Part II

Trees

Chapter 1

Binary search tree, the 'hello world' data structure

1.1 Introduction

Arrays or lists are typically considered the 'hello world' data structures. However, we'll see they are not actually particularly easy to implement. In some procedural settings, arrays are the most elementary data structures, and it is possible to implement linked lists using arrays (see section 10.3 in [2]). On the other hand, in some functional settings, linked lists are the elementary building blocks used to create arrays and other data structures.

Considering these factors, we start with Binary Search Trees (or BST) as the 'hello world' data structure using an interesting problem Jon Bentley mentioned in 'Programming Pearls' [2]. The problem is to count the number of times each word occurs in a large text. One solution in C++ is below:

```
int main(int, char** ){
  map<string, int> dict;
  string s;
  while(cin>>s)
    ++dict[s];
  map<string, int>::iterator it=dict.begin();
  for(; it!=dict.end(); ++it)
    cout<<it->first<<": "<<it->second<<"\n";
}
```

And we can run it to produce the result using the following UNIX commands [1].

```
$ g++ wordcount.cpp -o wordcount
$ cat bbe.txt | ./wordcount > wc.txt
```

The map provided in the standard template library is a kind of balanced BST with augmented data. Here we use the words in the text as the keys and the number of occurrences as the augmented data. This program is fast, and

[1]This is not a UNIX unique command, in Windows OS, it can be achieved by: *type bbe.txt|wordcount.exe > wc.txt*

it reflects the power of BSTs. We'll introduce how to implement BSTs in this section and show how to balance them in a later section.

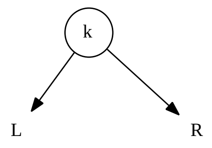
Before we dive into BSTs, let's first introduce the more general binary tree.

Binary trees are recursively defined. BSTs are just one type of binary tree. A binary tree is usually defined in the following way.

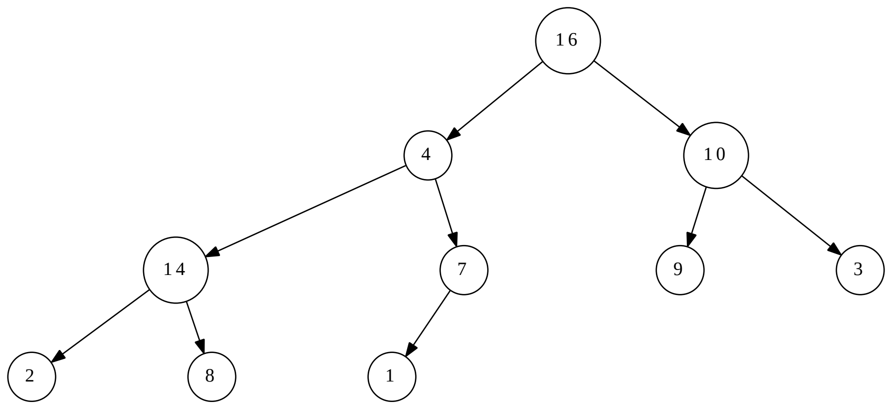
A binary tree is

- either an empty node;
- or a node containing 3 parts: a value, a left child which is a binary tree and a right child which is also a binary tree.

Figure 1.1 shows this concept and an example binary tree.



(a) Concept of binary tree



(b) An example binary tree

Figure 1.1: Binary tree concept and an example.

A BST is a binary tree where the following applies to each node:

- all the values in left child tree are less than the value of this node;
- the value of this node is less than any values in its right child tree.

Figure 1.2 shows an example of binary search tree. Comparing with Figure 1.1, we can see the differences in how keys are ordered between them.

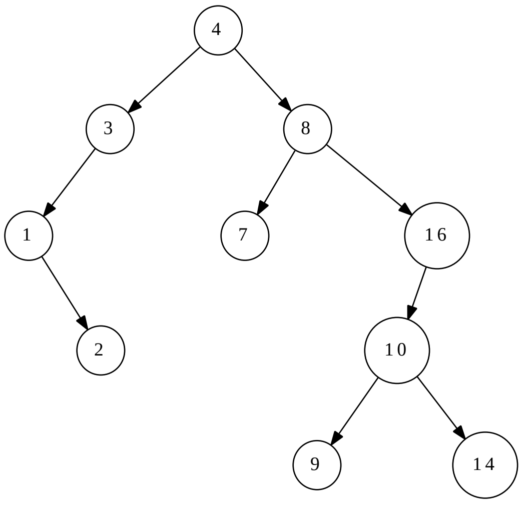


Figure 1.2: A Binary search tree example.

1.2 Data Layout

Based on the recursive definition of BSTs, we can draw the data layout in procedural setting with pointers as in Figure 1.3.

The node contains a field of key, which can be augmented with satellite data; a field contains a pointer to the left child and a field point to the right child. In order to back-track an ancestor easily, a parent field can be provided as well.

In this post, we'll ignore the satellite data for simple illustration purpose. Based on this layout, the node of binary search tree can be defined in a procedural language, such as C++ as the following.

```
template<class T>
struct node{
  node(T x):key(x), left(0), right(0), parent(0){}
  ~node(){
    delete left;
    delete right;
  }

  node* left;
  node* right;
  node* parent; //parent is optional, it's␣helpful␣for␣succ/pred
␣␣T␣key;
};
```

There is another setting, for instance in Scheme/Lisp languages, the elementary data structure is linked-list. Figure 1.4 shows how a binary search tree node can be built on top of linked-list.

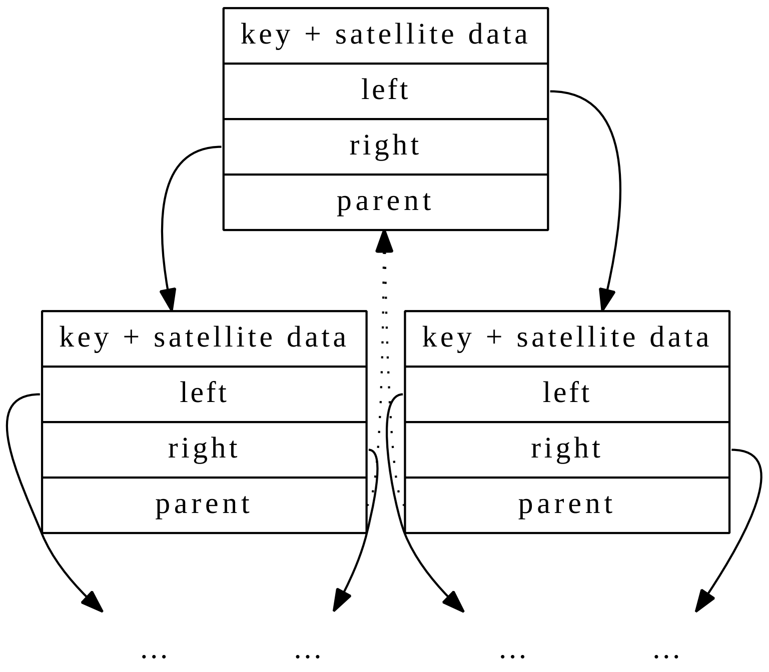


Figure 1.3: Layout of nodes with parent field.

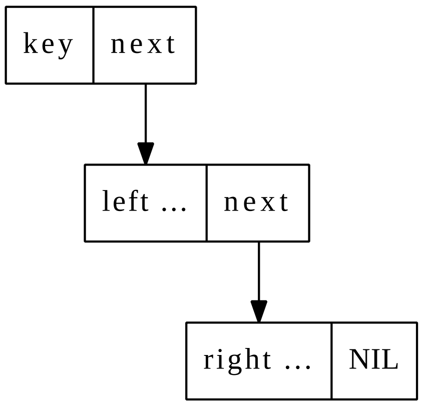


Figure 1.4: Binary search tree node layout on top of linked list. Where 'left...' and 'right ...' are either empty or binary search tree node composed in the same way.

Because in pure functional setting, It's hard to use pointer for back tracking the ancestors, (and typically, there is no need to do back tracking, since we can provide top-down solution in recursive way) there is not 'parent' field in such layout.

For simplified reason, we'll skip the detailed layout in the future, and only focus on the logic layout of data structures. For example, below is the definition of binary search tree node in Haskell.

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

1.3 Insertion

To insert a key k (may be along with a value in practice) to a binary search tree T, we can follow a quite straight forward way.

- If the tree is empty, then construct a leave node with key=k;
- If k is less than the key of root node, insert it to the left child;
- If k is greater than the key of root, insert it to the right child;

There is an exceptional case that if k is equal to the key of root, it means it has already existed, we can either overwrite the data, or just do nothing. For simple reason, this case is skipped in this post.

This algorithm is described recursively. It is so simple that is why we consider binary search tree is 'hello world' data structure. Formally, the algorithm can be represented with a recursive function.

$$insert(T, k) = \begin{cases} node(\phi, k, \phi) & : \quad T = \phi \\ node(insert(L, k), Key, R) & : \quad k < Key \\ node(L, Key, insert(R, k)) & : \quad otherwise \end{cases} \tag{1.1}$$

Where

$$\begin{array}{l} L = left(T) \\ R = right(T) \\ Key = key(T) \end{array}$$

The node function creates a new node with given left sub-tree, a key and a right sub-tree as parameters. ϕ means NIL or Empty. function $left$, $right$ and key are access functions which can get the left sub-tree, right sub-tree and the key of a node.

Translate the above functions directly to Haskell yields the following program.

```
insert::(Ord a) => Tree a -> a -> Tree a
insert Empty k = Node Empty k Empty
insert (Node l x r) k | k < x = Node (insert l k) x r
                      | otherwise = Node l x (insert r k)
```

This program utilized the pattern matching features provided by the language. However, even in functional settings without this feature, for instance, Scheme/Lisp, the program is still expressive.

```
(define (insert tree x)
  (cond ((null? tree) (list '() x '()))
        ((< x (key tree))
         (make-tree (insert (left tree) x)
                    (key tree)
                    (right tree)))
        ((> x (key tree))
         (make-tree (left tree)
                    (key tree)
                    (insert (right tree) x)))))
```

It is possible to turn the algorithm completely into imperative way without recursion.

```
1: function INSERT($T, k$)
2:     $root \leftarrow T$
3:     $x \leftarrow$ CREATE-LEAF($k$)
4:     $parent \leftarrow NIL$
5:     while $T \neq NIL$ do
6:         $parent \leftarrow T$
7:         if $k <$ KEY($T$) then
8:             $T \leftarrow$ LEFT($T$)
9:         else
10:            $T \leftarrow$ RIGHT($T$)
11:    PARENT($x$) $\leftarrow parent$
12:    if $parent = NIL$ then                              ▷ tree $T$ is empty
13:        return $x$
14:    else if $k <$ KEY($parent$) then
15:        LEFT($parent$) $\leftarrow x$
16:    else
17:        RIGHT($parent$) $\leftarrow x$
18:    return $root$

19: function CREATE-LEAF(k)
20:    $x \leftarrow$ EMPTY-NODE
21:    KEY($x$) $\leftarrow k$
22:    LEFT($x$) $\leftarrow NIL$
23:    RIGHT($x$) $\leftarrow NIL$
24:    PARENT($x$) $\leftarrow NIL$
25:    return $x$
```

Compare with the functional algorithm, it is obviously that this one is more complex although it is fast and can handle very deep tree. A complete C++ program and a python program are available along with this post for reference.

1.4 Traversing

Traversing means visiting every element one by one in a binary search tree. There are 3 ways to traverse a binary tree, pre-order tree walk, in-order tree walk, and post-order tree walk. The names of these traversing methods highlight the order of when we visit the root of a binary search tree.

Since there are three parts in a tree, as left child, the root, which contains the key and satellite data, and the right child. If we denote them as $(left, current, right)$, the three traversing methods are defined as the following.

- pre-order traverse, visit $current$, then $left$, finally $right$;
- in-order traverse, visit $left$, then $current$, finally $right$;
- post-order traverse, visit $left$, then $right$, finally $current$.

Note that each visiting operation is recursive. And we see the order of visiting $current$ determines the name of the traversing method.

For the binary search tree shown in figure 1.2, below are the three different traversing results.

- pre-order traverse result: 4, 3, 1, 2, 8, 7, 16, 10, 9, 14;
- in-order traverse result: 1, 2, 3, 4, 7, 8, 9, 10, 14, 16;
- post-order traverse result: 2, 1, 3, 7, 9, 14, 10, 16, 8, 4;

It can be found that the in-order walk of a binary search tree outputs the elements in increase order, which is particularly helpful. The definition of binary search tree ensures this interesting property, while the proof of this fact is left as an exercise of this post.

In-order tree walk algorithm can be described as the following:

- If the tree is empty, just return;
- traverse the left child by in-order walk, then access the key, finally traverse the right child by in-order walk.

Translate the above description yields a generic map function

$$map(f, T) = \begin{cases} \phi & : T = \phi \\ node(l', k', r') & : otherwise \end{cases} \tag{1.2}$$

where

$$\begin{array}{l} l' = map(f, left(T)) \\ r' = map(f, right(T)) \\ k' = f(key(T)) \end{array}$$

If we only need access the key without create the transformed tree, we can realize this algorithm in procedural way lie the below C++ program.

```
template<class T, class F>
void in_order_walk(node<T>* t, F f){
  if(t){
    in_order_walk(t->left, f);
    f(t->value);
    in_order_walk(t->right, f);
  }
}
```

The function takes a parameter f, it can be a real function, or a function object, this program will apply f to the node by in-order tree walk.

We can simplified this algorithm one more step to define a function which turns a binary search tree to a sorted list by in-order traversing.

$$toList(T) = \begin{cases} \phi & : T = \phi \\ toList(left(T)) \cup \{key(T)\} \cup toList(right(T)) & : otherwise \end{cases} \tag{1.3}$$

Below is the Haskell program based on this definition.

```
toList::(Ord a)⇒Tree a → [a]
toList Empty = []
toList (Node l x r) = toList l ++ [x] ++ toList r
```

This provides us a method to sort a list of elements. We can first build a binary search tree from the list, then output the tree by in-order traversing. This method is called as ‘tree sort’. Let’s denote the list $X = \{x_1, x_2, x_3, ..., x_n\}$.

$$sort(X) = toList(fromList(X)) \tag{1.4}$$

And we can write it in function composition form.

$$sort = toList.fromList$$

Where function $fromList$ repeatedly insert every element to a binary search tree.

$$fromList(X) = foldL(insert, \phi, X) \tag{1.5}$$

It can also be written in partial application form like below.

$$fromList = foldL \quad insert \quad \phi$$

For the readers who are not familiar with folding from left, this function can also be defined recursively as the following.

$$fromList(X) = \begin{cases} \phi & : X = \phi \\ insert(fromList(\{x_2, x_3, ..., x_n\}), x_1) & : otherwise \end{cases}$$

We’ll intense use folding function as well as the function composition and partial evaluation in the future, please refer to appendix of this book or [6] [7] and [8] for more information.

Exercise 1.1

- Given the in-order traverse result and pre-order traverse result, can you re-construct the tree from these result and figure out the post-order traversing result?

  Pre-order result: 1, 2, 4, 3, 5, 6; In-order result: 4, 2, 1, 5, 3, 6; Post-order result: ?

- Write a program in your favorite language to re-construct the binary tree from pre-order result and in-order result.

- Prove why in-order walk output the elements stored in a binary search tree in increase order?
- Can you analyze the performance of tree sort with big-O notation?

1.5 Querying a binary search tree

There are three types of querying for binary search tree, searching a key in the tree, find the minimum or maximum element in the tree, and find the predecessor or successor of an element in the tree.

1.5.1 Looking up

According to the definition of binary search tree, search a key in a tree can be realized as the following.

- If the tree is empty, the searching fails;
- If the key of the root is equal to the value to be found, the search succeed. The root is returned as the result;
- If the value is less than the key of the root, search in the left child.
- Else, which means that the value is greater than the key of the root, search in the right child.

This algorithm can be described with a recursive function as below.

$$lookup(T, x) = \begin{cases} \phi & : T = \phi \\ T & : key(T) = x \\ lookup(left(T), x) & : x < key(T) \\ lookup(right(T), x) & : otherwise \end{cases} \tag{1.6}$$

In the real application, we may return the satellite data instead of the node as the search result. This algorithm is simple and straightforward. Here is a translation of Haskell program.

```
lookup::(Ord a)⇒ Tree a → a → Tree a
lookup Empty _ = Empty
lookup t@(Node l k r) x | k == x = t
                        | x < k = lookup l x
                        | otherwise = lookup r x
```

If the binary search tree is well balanced, which means that almost all nodes have both non-NIL left child and right child, for N elements, the search algorithm takes $O(\lg N)$ time to perform. This is not formal definition of balance. We'll show it in later post about red-black-tree. If the tree is poor balanced, the worst case takes $O(N)$ time to search for a key. If we denote the height of the tree as h, we can uniform the performance of the algorithm as $O(h)$.

The search algorithm can also be realized without using recursion in a procedural manner.

```
1: function SEARCH($T, x$)
2:     while $T \neq NIL \land$ KEY($T$) $\neq x$ do
```

```
3:     if $x <$ KEY($T$) then
4:         $T \leftarrow$ LEFT($T$)
5:     else
6:         $T \leftarrow$ RIGHT($T$)
7: return $T$
```

Below is the C++ program based on this algorithm.

```
template<class T>
node<T>* search(node<T>* t, T x){
  while(t && t->key!=x){
    if(x < t->key) t=t->left;
    else t=t->right;
  }
  return t;
}
```

1.5.2 Minimum and maximum

Minimum and maximum can be implemented from the property of binary search tree, less keys are always in left child, and greater keys are in right.

For minimum, we can continue traverse the left sub tree until it is empty. While for maximum, we traverse the right.

$$min(T) = \begin{cases} key(T) & : left(T) = \phi \\ min(left(T)) & : otherwise \end{cases} \tag{1.7}$$

$$max(T) = \begin{cases} key(T) & : right(T) = \phi \\ max(right(T)) & : otherwise \end{cases} \tag{1.8}$$

Both function bound to $O(h)$ time, where h is the height of the tree. For the balanced binary search tree, *min*/*max* are bound to $O(\lg N)$ time, while they are $O(N)$ in the worst cases.

We skip translating them to programs, It’s also possible to implement them in pure procedural way without using recursion.

1.5.3 Successor and predecessor

The last kind of querying, to find the successor or predecessor of an element is useful when a tree is treated as a generic container and traversed by using iterator. It will be relative easier to implement if parent of a node can be accessed directly.

It seems that the functional solution is hard to be found, because there is no pointer like field linking to the parent node. One solution is to left ‘breadcrumbs’ when we visit the tree, and use these information to back-track or even re-construct the whole tree. Such data structure, that contains both the tree and ‘breadcrumbs’ is called zipper. please refer to [9] for details.

However, If we consider the original purpose of providing *succ*/*pred* function, ‘to traverse all the binary search tree elements one by one‘ as a generic container, we realize that they don’t make significant sense in functional settings because we can traverse the tree in increase order by *mapT* function we defined previously.

We'll meet many problems in this series of post that they are only valid in imperative settings, and they are not meaningful problems in functional settings at all. One good example is how to delete an element in red-black-tree[3].

In this section, we'll only present the imperative algorithm for finding the successor and predecessor in a binary search tree.

When finding the successor of element x, which is the smallest one y that satisfies $y > x$, there are two cases. If the node with value x has non-NIL right child, the minimum element in right child is the answer; For example, in Figure 1.2, in order to find the successor of 8, we search it's right sub tree for the minimum one, which yields 9 as the result. While if node x don't have right child, we need back-track to find the closest ancestors whose left child is also ancestor of x. In Figure 1.2, since 2 don't have right sub tree, we go back to its parent 1. However, node 1 don't have left child, so we go back again and reach to node 3, the left child of 3, is also ancestor of 2, thus, 3 is the successor of node 2.

Based on this description, the algorithm can be given as the following.

```
1: function Succ($x$)
2:     if Right($x$) $\neq NIL$ then
3:         return Min(Right($x$))
4:     else
5:         $p \leftarrow$ Parent($x$)
6:         while $p \neq NIL$ and $x =$ Right($p$) do
7:             $x \leftarrow p$
8:             $p \leftarrow$ Parent($p$)
9:         return $p$
```

The predecessor case is quite similar to the successor algorithm, they are symmetrical to each other.

```
1: function Pred($x$)
2:     if Left($x$) $\neq NIL$ then
3:         return Max(Left($x$))
4:     else
5:         $p \leftarrow$ Parent($x$)
6:         while $p \neq NIL$ and $x =$ Left($p$) do
7:             $x \leftarrow p$
8:             $p \leftarrow$ Parent($p$)
9:         return $p$
```

Below are the Python programs based on these algorithms. They are changed a bit in while loop conditions.

```
def succ(x):
    if x.right is not None: return tree_min(x.right)
    p = x.parent
    while p is not None and p.left != x:
        x = p
        p = p.parent
    return p

def pred(x):
    if x.left is not None: return tree_max(x.left)
    p = x.parent
```

```
while p is not None and p.right != x:
    x = p
    p = p.parent
return p
```

Exercise 1.2

- Can you figure out how to iterate a tree as a generic container by using $pred()/succ()$? What’s the performance of such traversing process in terms of big-O?
- A reader discussed about traversing all elements inside a range $[a, b]$. In C++, the algorithm looks like the below code:

  $\textbf{for\_each}(m.lower\_bound(12), m.upper\_bound(26), f);$

  Can you provide the purely function solution for this problem?

1.6 Deletion

Deletion is another ‘imperative only’ topic for binary search tree. This is because deletion mutate the tree, while in purely functional settings, we don’t modify the tree after building it in most application.

However, One method of deleting element from binary search tree in purely functional way is shown in this section. It’s actually reconstructing the tree but not modifying the tree.

Deletion is the most complex operation for binary search tree. this is because we must keep the BST property, that for any node, all keys in left sub tree are less than the key of this node, and they are all less than any keys in right sub tree. Deleting a node can break this property.

In this post, different with the algorithm described in [2], A simpler one from SGI STL implementation is used.[6]

To delete a node x from a tree.

- If x has no child or only one child, splice x out;
- Otherwise (x has two children), use minimum element of its right sub tree to replace x, and splice the original minimum element out.

The simplicity comes from the truth that, the minimum element is stored in a node in the right sub tree, which can’t have two non-NIL children. It ends up in the trivial case, the the node can be directly splice out from the tree.

Figure 1.5, 1.6, and 1.7 illustrate these different cases when deleting a node from the tree.

Based on this idea, the deletion can be defined as the below function.

$$delete(T, x) = \begin{cases} \phi & : T = \phi \\ node(delete(L, x), K, R) & : x < K \\ node(L, K, delete(R, x)) & : x > K \\ R & : x = K \land L = \phi \\ L & : x = K \land R = \phi \\ node(L, y, delete(R, y)) & : otherwise \end{cases} \tag{1.9}$$

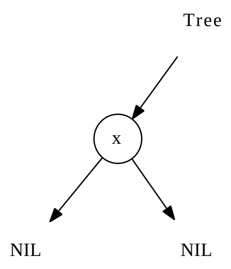
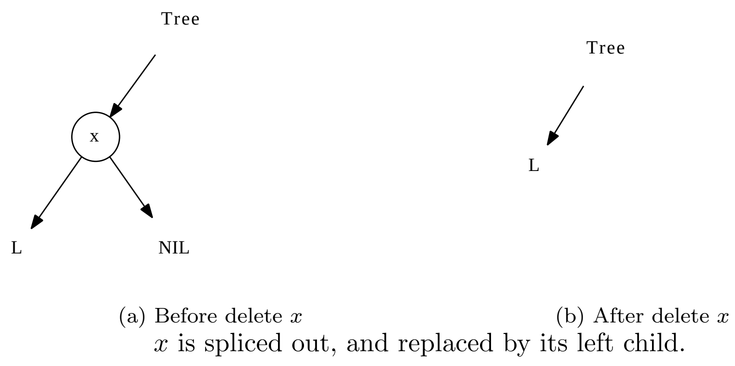
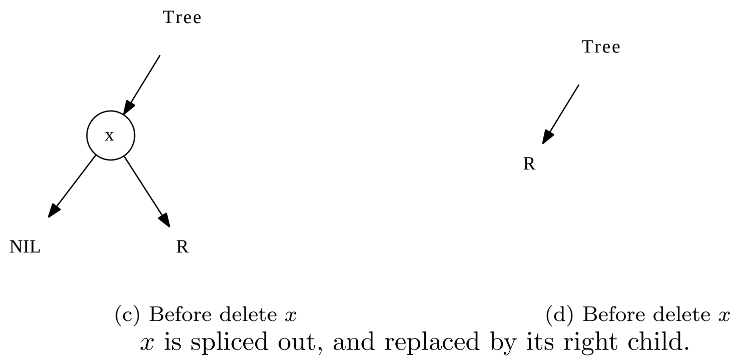


Figure 1.5: x can be spliced out.

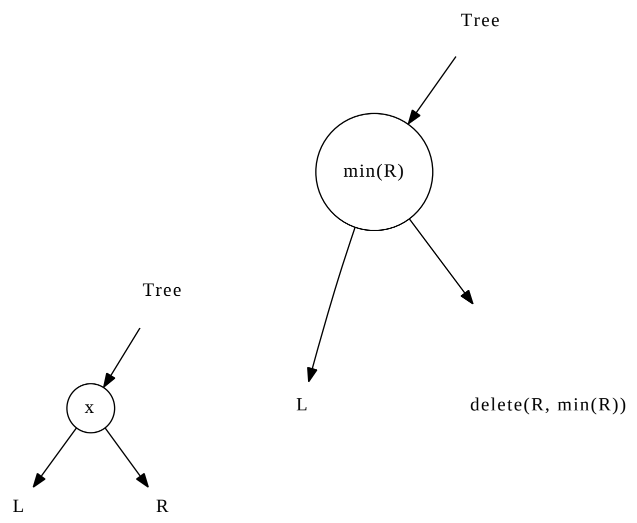


(a) Before delete x (b) After delete x
x is spliced out, and replaced by its left child.



(c) Before delete x (d) Before delete x
x is spliced out, and replaced by its right child.

Figure 1.6: Delete a node which has only one non-NIL child.



(a) Before delete x (b) After delete x
x is replaced by splicing the minimum element from its right child.

Figure 1.7: Delete a node which has both children.

Where

$$
\begin{array}{l}
L = left(T) \\
R = right(T) \\
K = key(T) \\
y = min(R)
\end{array}
$$

Translating the function to Haskell yields the below program.

```
delete::(Ord a)⇒ Tree a → a → Tree a
delete Empty _ = Empty
delete (Node l k r) x | x < k = (Node (delete l x) k r)
                      | x > k = (Node l k (delete r x))
                      -- x == k
                      | isEmpty l = r
                      | isEmpty r = l
                      | otherwise = (Node l k' (delete r k'))
                          where k' = min r
```

Function 'isEmpty' is used to test if a tree is empty (ϕ). Note that the algorithm first performs search to locate the node where the element need be deleted, after that it execute the deletion. This algorithm takes $O(h)$ time where h is the height of the tree.

It's also possible to pass the node but not the element to the algorithm for deletion. Thus the searching is no more needed.

The imperative algorithm is more complex because it need set the parent properly. The function will return the root of the result tree.

```
1: function DELETE($T, x$)
2:     $root \leftarrow T$
```

```
3:  $x' \leftarrow x$                                   ▷ save $x$
4:  $parent \leftarrow$ PARENT($x$)
5:  if LEFT($x$) $= NIL$ then
6:      $x \leftarrow$ RIGHT($x$)
7:  else if RIGHT($x$) $= NIL$ then
8:      $x \leftarrow$ LEFT($x$)
9:  else                                                ▷ both children are non-NIL
10:     $y \leftarrow$ MIN(RIGHT($x$))
11:     KEY($x$) $\leftarrow$ KEY($y$)
12:     Copy other satellite data from $y$ to $x$
13:     if PARENT($y$) $\neq x$ then                    ▷ $y$ hasn't left sub tree
14:         LEFT(PARENT($y$)) $\leftarrow$ RIGHT($y$)
15:     else                                            ▷ $y$ is the root of right child of $x$
16:         RIGHT($x$) $\leftarrow$ RIGHT($y$)
17:     Remove $y$
18:     return $root$
19: if $x \neq NIL$ then
20:     PARENT($x$) $\leftarrow parent$
21: if $parent = NIL$ then                              ▷ We are removing the root of the tree
22:     $root \leftarrow x$
23: else
24:     if LEFT($parent$) $= x'$ then
25:         LEFT($parent$) $\leftarrow x$
26:     else
27:         RIGHT($parent$) $\leftarrow x$
28: Remove $x'$
29: return $root$
```

Here we assume the node to be deleted is not empty (otherwise we can simply returns the original tree). In other cases, it will first record the root of the tree, create copy pointers to x, and its parent.

If either of the children is empty, the algorithm just splice x out. If it has two non-NIL children, we first located the minimum of right child, replace the key of x to y's, copy the satellite data as well, then splice y out. Note that there is a special case that y is the root node of x's left sub tree.

Finally we need reset the stored parent if the original x has only one non-NIL child. If the parent pointer we copied before is empty, it means that we are deleting the root node, so we need return the new root. After the parent is set properly, we finally remove the old x from memory.

The relative Python program for deleting algorithm is given as below. Because Python provides GC, we needn't explicitly remove the node from the memory.

```
def tree_delete(t, x):
    if x is None:
        return t
    [root, old_x, parent] = [t, x, x.parent]
    if x.left is None:
        x = x.right
    elif x.right is None:
        x = x.left
```

```
else:
    y = tree_min(x.right)
    x.key = y.key
    if y.parent != x:
        y.parent.left = y.right
    else:
        x.right = y.right
    return root
if x is not None:
    x.parent = parent
if parent is None:
    root = x
else:
    if parent.left == old_x:
        parent.left = x
    else:
        parent.right = x
return root
```

Because the procedure seeks minimum element, it runs in $O(h)$ time on a tree of height h.

Exercise 1.3

- There is a symmetrical solution for deleting a node which has two non-NIL children, to replace the element by splicing the maximum one out off the left sub-tree. Write a program to implement this solution.

1.7 Randomly build binary search tree

It can be found that all operations given in this post bound to $O(h)$ time for a tree of height h. The height affects the performance a lot. For a very unbalanced tree, h tends to be $O(N)$, which leads to the worst case. While for balanced tree, h close to $O(\lg N)$. We can gain the good performance.

How to make the binary search tree balanced will be discussed in next post. However, there exists a simple way. Binary search tree can be randomly built as described in [2]. Randomly building can help to avoid (decrease the possibility) unbalanced binary trees. The idea is that before building the tree, we can call a random process, to shuffle the elements.

Exercise 1.4

- Write a randomly building process for binary search tree.

1.8 Appendix

All programs are provided along with this post. They are free for downloading. We provided C, C++, Python, Haskell, and Scheme/Lisp programs as example.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[2] Jon Bentley. “Programming Pearls(2nd Edition)”. Addison-Wesley Professional; 2 edition (October 7, 1999). ISBN-13: 978-0201657883

[3] Chris Okasaki. “Ten Years of Purely Functional Data Structures”. http://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.html

[4] SGI. “Standard Template Library Programmer’s Guide”. http://www.sgi.com/tech/stl/

[5] http://en.literateprograms.org/Category:Binary_search_tree

[6] http://en.wikipedia.org/wiki/Foldl

[7] http://en.wikipedia.org/wiki/Function_composition

[8] http://en.wikipedia.org/wiki/Partial_application

[9] Miran Lipovaca. “Learn You a Haskell for Great Good! A Beginner’s Guide”. the last chapter. No Starch Press; 1 edition April 2011, 400 pp. ISBN: 978-1-59327-283-8

Chapter 2

The evolution of insertion sort

2.1 Introduction

In previous chapter, we introduced the 'hello world' data structure, binary search tree. In this chapter, we explain insertion sort, which can be think of the 'hello world' sorting algorithm [1]. It's straightforward, but the performance is not as good as some divide and conqueror sorting approaches, such as quick sort and merge sort. Thus insertion sort is seldom used as generic sorting utility in modern software libraries. We'll analyze the problems why it is slow, and trying to improve it bit by bit till we reach the best bound of comparison based sorting algorithms, $O(n \lg n)$, by evolution to tree sort. And we finally show the connection between the 'hello world' data structure and 'hello world' sorting algorithm.

The idea of insertion sort can be vivid illustrated by a real life poker game[2]. Suppose the cards are shuffled, and a player starts taking card one by one.

At any time, all cards in player's hand are well sorted. When the player gets a new card, he insert it in proper position according to the order of points. Figure 2.1 shows this insertion example.

Based on this idea, the algorithm of insertion sort can be directly given as the following.

```
function SORT($A$)
    $X \leftarrow \Phi$
    for each $x \in A$ do
        INSERT($X, x$)
    return $X$
```

It's easy to express this process with folding, which we mentioned in the chapter of binary search tree.

$$insert = foldL \quad insert \quad \Phi \tag{2.1}$$

[1]Some reader may argue that 'Bubble sort' is the easiest sort algorithm. Bubble sort isn't covered in this book as we don't think it's a valuable algorithm[1]

Figure 2.1: Insert card 8 to proper position in a deck.

Note that in above algorithm, we store the sorted result in X, so this isn't in-place sorting. It's easy to change it to in-place algorithm.

```
function Sort($A$)
    for $i \leftarrow 2$ to Length($A$) do
        insert $A_i$ to sorted sequence $\{A'_1, A'_2, ..., A'_{i-1}\}$
```

At any time, when we process the i-th element, all elements before i have already been sorted. we continuously insert the current elements until consume all the unsorted data. This idea is illustrated as in figure 9.3.

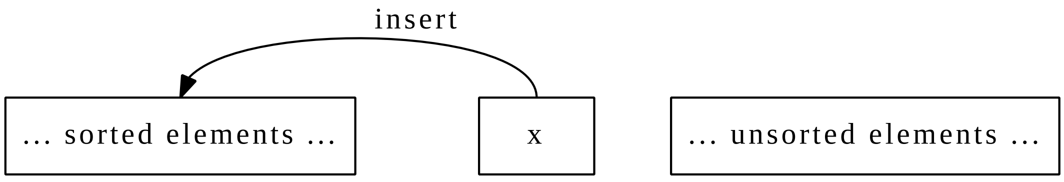


Figure 2.2: The left part is sorted data, continuously insert elements to sorted part.

We can find there is recursive concept in this definition. Thus it can be expressed as the following.

$$sort(A) = \begin{cases} \Phi & : A = \Phi \\ insert(sort(\{A_2, A_3, ...\}), A_1) & : otherwise \end{cases} \tag{2.2}$$

2.2 Insertion

We haven't answered the question about how to realize insertion however. It's a puzzle how does human locate the proper position so quickly.

For computer, it's an obvious option to perform a scan. We can either scan from left to right or vice versa. However, if the sequence is stored in plain array, it's necessary to scan from right to left.

```
function Sort($A$)
    for $i \leftarrow 2$ to Length($A$) do    ▷ Insert $A[i]$ to sorted sequence $A[1...i-1]$
    $x \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0 \land x < A[j]$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow x$
```

One may think scan from left to right is natural. However, it isn't as effect as above algorithm for plain array. The reason is that, it's expensive to insert an element in arbitrary position in an array. As array stores elements continuously. If we want to insert new element x in position i, we must shift all elements after i, including $i+1, i+2, ...$ one position to right. After that the cell at position i is empty, and we can put x in it. This is illustrated in figure 2.3.

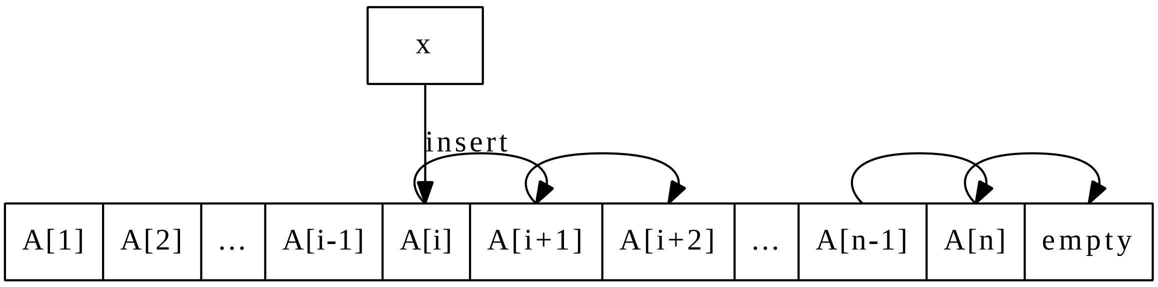


Figure 2.3: Insert x to array A at position i.

If the length of array is n, this indicates we need examine the first i elements, then perform $n - i + 1$ moves, and then insert x to the i-th cell. So insertion from left to right need traverse the whole array anyway. While if we scan from right to left, we totally examine the last $j = n - i + 1$ elements, and perform the same amount of moves. If j is small (e.g. less than $n/2$), there is possibility to perform less operations than scan from left to right.

Translate the above algorithm to Python yields the following code.

```
def isort(xs):
    n = len(xs)
    for i in range(1, n):
        x = xs[i]
        j = i - 1
        while j ≥ 0 and x < xs[j]:
            xs[j+1] = xs[j]
            j = j - 1
        xs[j+1] = x
```

It can be found some other equivalent programs, for instance the following ANSI C program. However this version isn't as effective as the pseudo code.

```
void isort(Key* xs, int n){
  int i, j;
  for(i=1; i<n; ++i)
    for(j=i-1; j≥0 && xs[j+1] < xs[j]; --j)
      swap(xs, j, j+1);
}
```

This is because the swapping function, which can exchange two elements typically uses a temporary variable like the following:

```
void swap(Key* xs, int i, int j){
  Key temp = xs[i];
  xs[i] = xs[j];
  xs[j] = temp;
}
```

So the ANSI C program presented above takes $3m$ times assignment, where m is the number of inner loops. While the pseudo code as well as the Python program use shift operation instead of swapping. There are $n + 2$ times assignment.

We can also provide INSERT() function explicitly, and call it from the general insertion sort algorithm in previous section. We skip the detailed realization here and left it as an exercise.

All the insertion algorithms are bound to $O(n)$, where n is the length of the sequence. No matter what difference among them, such as scan from left or from right. Thus the over all performance for insertion sort is quadratic as $O(n^2)$.

Exercise 2.1

- Provide explicit insertion function, and call it with general insertion sort algorithm. Please realize it in both procedural way and functional way.

2.3 Improvement 1

Let's go back to the question, that why human being can find the proper position for insertion so quickly. We have shown a solution based on scan. Note the fact that at any time, all cards at hands have been well sorted, another possible solution is to use binary search to find that location.

We'll explain the search algorithms in other dedicated chapter. Binary search is just briefly introduced for illustration purpose here.

The algorithm will be changed to call a binary search procedure.

```
function SORT($A$)
    for $i \leftarrow 2$ to LENGTH($A$) do
        $x \leftarrow A[i]$
        $p \leftarrow$ BINARY-SEARCH($A[1...i-1], x$)
        for $j \leftarrow i$ down to $p$ do
            $A[j] \leftarrow A[j-1]$
        $A[p] \leftarrow x$
```

Instead of scan elements one by one, binary search utilize the information that all elements in slice of array $\{A_1, ..., A_{i-1}\}$ are sorted. Let's assume the order is monotonic increase order. To find a position j that satisfies $A_{j-1} \leq x \leq A_j$. We can first examine the middle element, for example, $A_{\lfloor i/2 \rfloor}$. If x is less than it, we need next recursively perform binary search in the first half of the sequence; otherwise, we only need search in last half.

Every time, we halve the elements to be examined, this search process runs $O(\lg n)$ time to locate the insertion position.

```
function BINARY-SEARCH($A, x$)
```

```
  $l \leftarrow 1$
  $u \leftarrow 1+$ LENGTH($A$)
  **while** $l < u$ **do**
      $m \leftarrow \lfloor \frac{l+u}{2} \rfloor$
      **if** $A_m = x$ **then**
          **return** $m$                                  ▷ Find a duplicated element
      **else if** $A_m < x$ **then**
          $l \leftarrow m + 1$
      **else**
          $u \leftarrow m$
  **return** $l$
```

The improved insertion sort algorithm is still bound to $O(n^2)$, compare to previous section, which we use $O(n^2)$ times comparison and $O(n^2)$ moves, with binary search, we just use $O(n \lg n)$ times comparison and $O(n^2)$ moves.

The Python program regarding to this algorithm is given below.

```
def isort(xs):
    n = len(xs)
    for i in range(1, n):
        x = xs[i]
        p = binary_search(xs[:i], x)
        for j in range(i, p, -1):
            xs[j] = xs[j-1]
        xs[p] = x

def binary_search(xs, x):
    l = 0
    u = len(xs)
    while l < u:
        m = (l+u)/2
        if xs[m] == x:
            return m
        elif xs[m] < x:
            l = m + 1
        else:
            u = m
    return l
```

Exercise 2.2

Write the binary search in recursive manner. You needn't use purely functional programming language.

2.4 Improvement 2

Although we improve the search time to $O(n \lg n)$ in previous section, the number of moves is still $O(n^2)$. The reason of why movement takes so long time, is because the sequence is stored in plain array. The nature of array is continuously layout data structure, so the insertion operation is expensive. This hints us that we can use linked-list setting to represent the sequence. It can improve the insertion operation from $O(n)$ to constant time $O(1)$.

$$insert(A, x) = \begin{cases} \{x\} & : A = \Phi \\ \{x\} \cup A & : x < A_1 \\ \{A_1\} \cup insert(\{A_2, A_3, ...A_n\}, x) & : otherwise \end{cases} \tag{2.3}$$

Translating the algorithm to Haskell yields the below program.

```
insert :: (Ord a) ⇒ [a] → a → [a]
insert [] x = [x]
insert (y:ys) x = if x < y then x:y:ys else y:insert ys x
```

And we can complete the two versions of insertion sort program based on the first two equations in this chapter.

```
isort [] = []
isort (x:xs) = insert (isort xs) x
```

Or we can represent the recursion with folding.

```
isort = foldl insert []
```

Linked-list setting solution can also be described imperatively. Suppose function KEY(x), returns the value of element stored in node x, and NEXT(x) accesses the next node in the linked-list.

```
function INSERT($L, x$)
    $p \leftarrow NIL$
    $H \leftarrow L$
    while $L \neq NIL \wedge$ KEY($L$) $<$ KEY($x$) do
        $p \leftarrow L$
        $L \leftarrow$ NEXT($L$)
    NEXT($x$) $\leftarrow L$
    if $p \neq NIL$ then
        $H \leftarrow x$
    else
        NEXT($p$) $\leftarrow x$
    return $H$
```

For example in ANSI C, the linked-list can be defined as the following.

```
struct node{
  Key key;
  struct node* next;
};
```

Thus the insert function can be given as below.

```
struct node* insert(struct node* lst, struct node* x){
  struct node *p, *head;
  p = NULL;
  for(head = lst; lst && x→key > lst→key; lst = lst→next)
    p = lst;
  x→next = lst;
  if(!p)
    return x;
  p→next = x;
  return head;
}
```

Instead of using explicit linked-list such as by pointer or reference based structure. Linked-list can also be realized by another index array. For any array element A_i, $Next_i$ stores the index of next element follows A_i. It means A_{Next_i} is the next element after A_i.

The insertion algorithm based on this solution is given like below.

```
function Insert($A, Next, i$)
    $j \leftarrow \perp$
    while $Next_j \neq NIL \wedge A_{Next_j} < A_i$ do
        $j \leftarrow Next_j$
    $Next_i \leftarrow Next_j$
    $Next_j \leftarrow i$
```

Here $\perp$ means the head of the $Next$ table. And the relative Python program for this algorithm is given as the following.

```
def isort(xs):
    n = len(xs)
    next = [-1]*(n+1)
    for i in range(n):
        insert(xs, next, i)
    return next

def insert(xs, next, i):
    j = -1
    while next[j] != -1 and xs[next[j]] < xs[i]:
        j = next[j]
    next[j], next[i] = i, next[j]
```

Although we change the insertion operation to constant time by using linked-list. However, we have to traverse the linked-list to find the position, which results $O(n^2)$ times comparison. This is because linked-list, unlike array, doesn't support random access. It means we can't use binary search with linked-list setting.

Exercise 2.3

- Complete the insertion sort by using linked-list insertion function in your favorate imperative programming language.
- The index based linked-list return the sequence of rearranged index as result. Write a program to re-order the original array of elements from this result.

2.5 Final improvement by binary search tree

It seems that we drive into a corner. We must improve both the comparison and the insertion at the same time, or we will end up with $O(n^2)$ performance.

We must use binary search, this is the only way to improve the comparison time to $O(\lg n)$. On the other hand, we must change the data structure, because we can't achieve constant time insertion at a position with plain array.

This remind us about our 'hello world' data structure, binary search tree. It naturally support binary search from its definition. At the same time, We can insert a new leaf in binary search tree in $O(1)$ constant time if we already find the location.

So the algorithm changes to this.

```
function SORT($A$)
    $T \leftarrow \Phi$
    for each $x \in A$ do
        $T \leftarrow$ INSERT-TREE($T, x$)
    return TO-LIST($T$)
```

Where INSERT-TREE() and TO-LIST() are described in previous chapter about binary search tree.

As we have analyzed for binary search tree, the performance of tree sort is bound to $O(n \lg n)$, which is the lower limit of comparison based sort[3].

2.6 Short summary

In this chapter, we present the evolution process of insertion sort. Insertion sort is well explained in most textbooks as the first sorting algorithm. It has simple and straightforward idea, but the performance is quadratic. Some textbooks stop here, but we want to show that there exist ways to improve it by different point of view. We first try to save the comparison time by using binary search, and then try to save the insertion operation by changing the data structure to linked-list. Finally, we combine these two ideas and evolute insertion sort to tree sort.

Bibliography

[1] http://en.wikipedia.org/wiki/Bubble_sort

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[3] Donald E. Knuth. “The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)”. Addison-Wesley Professional; 2 edition (May 4, 1998) ISBN-10: 0201896850 ISBN-13: 978-0201896855

Chapter 3

Red-black tree, not so complex as it was thought

3.1 Introduction

3.1.1 Exploit the binary search tree

We have shown the power of binary search tree by using it to count the occurrence of every word in Bible. The idea is to use binary search tree as a dictionary for counting.

One may come to the idea that to feed a yellow page book [1] to a binary search tree, and use it to look up the phone number for a contact.

By modifying a bit of the program for word occurrence counting yields the following code.

```
int main(int, char** ){
  ifstream f("yp.txt");
  map<string, string> dict;
  string name, phone;
  while(f>>name && f>>phone)
    dict[name]=phone;
  for(;;){
    cout<<"λnname:␣";
    cin>>name;
    if(dict.find(name)==dict.end())
      cout<<"not␣found";
    else
      cout<<"phone:␣"<<dict[name];
  }
}
```

This program works well. However, if you replace the STL map with the binary search tree as mentioned previously, the performance will be bad, especially when you search some names such as Zara, Zed, Zulu.

This is because the content of yellow page is typically listed in lexicographic order. Which means the name list is in increase order. If we try to insert a

[1]A name-phone number contact list book

sequence of number 1, 2, 3, ..., n to a binary search tree. We will get a tree like in Figure 3.1.

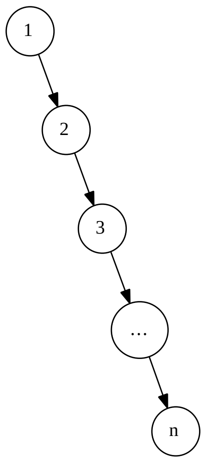


Figure 3.1: unbalanced tree

This is a extreme unbalanced binary search tree. The looking up performs $O(h)$ for a tree with height h. In balanced case, we benefit from binary search tree by $O(\lg N)$ search time. But in this extreme case, the search time downgraded to $O(N)$. It's no better than a normal link-list.

Exercise 3.1

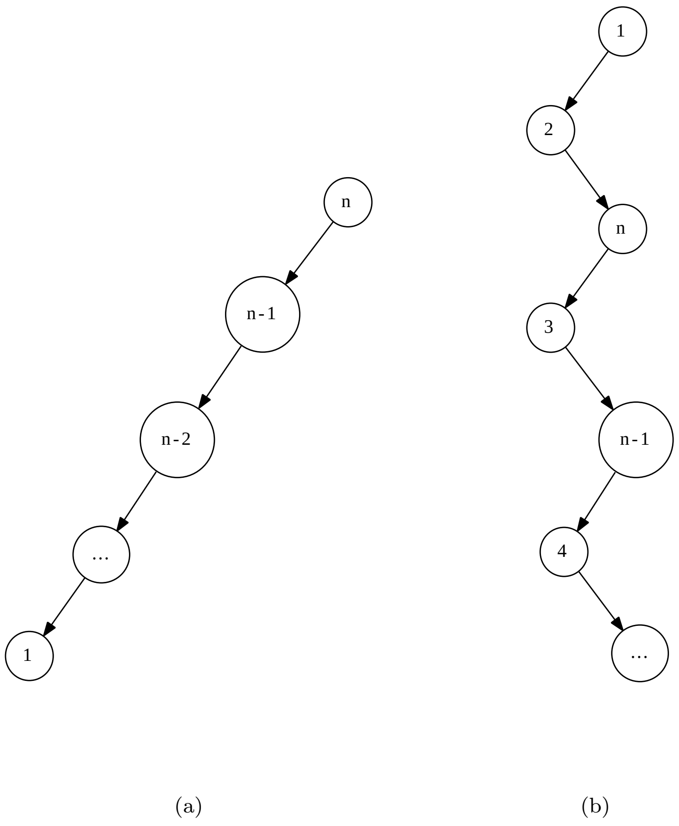
- For a very big yellow page list, one may want to speed up the dictionary building process by two concurrent tasks (threads or processes). One task reads the name-phone pair from the head of the list, while the other one reads from the tail. The building terminates when these two tasks meet at the middle of the list. What will be the binary search tree looks like after building? What if you split the the list more than two and use more tasks?

- Can you find any more cases to exploit a binary search tree? Please consider the unbalanced trees shown in figure 3.2.

3.1.2 How to ensure the balance of the tree

In order to avoid such case, we can shuffle the input sequence by randomized algorithm, such as described in Section 12.4 in [2]. However, this method doesn't always work, for example the input is fed from user interactively, and the tree need to be built/updated after each input.

There are many solutions people have ever found to make binary search tree balanced. Many of them rely on the rotation operations to binary search tree. Rotation operations change the tree structure while maintain the ordering of the elements. Thus it either improve or keep the balance property of the binary search tree.



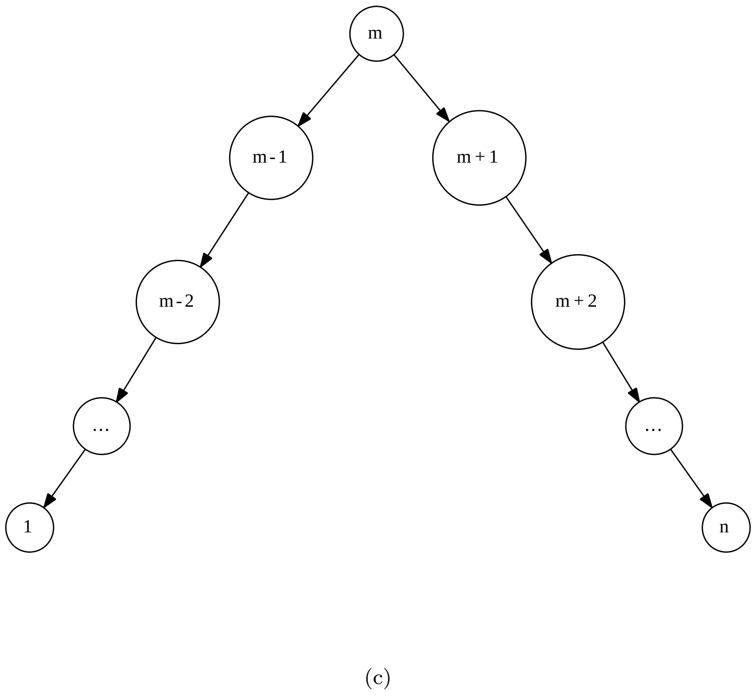


Figure 3.2: Some unbalanced trees

In this chapter, we'll first introduce about red-black tree, which is one of the most popular and widely used self-adjusting balanced binary search tree. In next chapter, we'll introduce about AVL tree, which is another intuitive solution; In later chapter about binary heaps, we'll show another interesting tree called splay tree, which can gradually adjust the the tree to make it more and more balanced.

3.1.3 Tree rotation

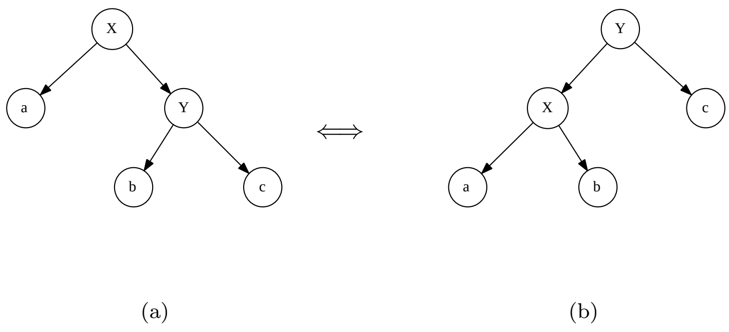


Figure 3.3: Tree rotation, 'rotate-left' transforms the tree from left side to right side, and 'rotate-right' does the inverse transformation.

Tree rotation is a kind of special operation that can transform the tree structure without changing the in-order traverse result. It based on the fact that for a specified ordering, there are multiple binary search trees correspond to it. Figure 3.3 shows the tree rotation. For a binary search tree on the left side, left rotate transforms it to the tree on the right, and right rotate does the inverse transformation.

Although tree rotation can be realized in procedural way, there exists quite simple function description if using pattern matching.

$$rotateL(T) = \begin{cases} node(node(a, X, b), Y, c) & : pattern(T) = node(a, X, node(b, Y, c)) \\ T & : otherwise \end{cases} \tag{3.1}$$

$$rotateR(T) = \begin{cases} node(a, X, node(b, Y, c)) & : pattern(T) = node(node(a, X, b), Y, c)) \\ T & : otherwise \end{cases} \tag{3.2}$$

However, the pseudo code dealing imperatively has to set all fields accordingly.

```
1: function LEFT-ROTATE($T, x$)
2:     $p \leftarrow$ PARENT($x$)
3:     $y \leftarrow$ RIGHT($x$)                          ▷ Assume $y \neq NIL$
4:     $a \leftarrow$ LEFT($x$)
5:     $b \leftarrow$ LEFT($y$)
6:     $c \leftarrow$ RIGHT($y$)
```

```
    REPLACE($x, y$)
    SET-CHILDREN($x, a, b$)
    SET-CHILDREN($y, x, c$)
    if $p = NIL$ then
        $T \leftarrow y$
    return $T$

function RIGHT-ROTATE($T, y$)
    $p \leftarrow$ PARENT($y$)
    $x \leftarrow$ LEFT($y$)                              ▷ Assume $x \neq NIL$
    $a \leftarrow$ LEFT($x$)
    $b \leftarrow$ RIGHT($x$)
    $c \leftarrow$ RIGHT($y$)
    REPLACE($y, x$)
    SET-CHILDREN($y, b, c$)
    SET-CHILDREN($x, a, y$)
    if $p = NIL$ then
        $T \leftarrow x$
    return $T$

function SET-LEFT($x, y$)
    LEFT($x$) $\leftarrow y$
    if $y \neq NIL$ then PARENT($y$) $\leftarrow x$

function SET-RIGHT($x, y$)
    RIGHT($x$) $\leftarrow y$
    if $y \neq NIL$ then PARENT($y$) $\leftarrow x$

function SET-CHILDREN($x, L, R$)
    SET-LEFT($x, L$)
    SET-RIGHT($x, R$)

function REPLACE($x, y$)
    if PARENT($x$) $= NIL$ then
        if $y \neq NIL$ then PARENT($y$) $\leftarrow NIL$
    else if LEFT(PARENT($x$)) $= x$ then SET-LEFT(PARENT($x$), $y$)
    elseSET-RIGHT(PARENT($x$), $y$)
    PARENT($x$) $\leftarrow NIL$
```

Compare these pseudo codes with the pattern matching functions, the former focus on the structure states changing, while the latter focus on the rotation process. As the title of this chapter indicated, red-black tree needn't be so complex as it was thought. Most traditional algorithm text books use the classic procedural way to teach red-black tree, there are several cases need to deal and all need carefulness to manipulate the node fields. However, by changing the mind to functional settings, things get intuitive and simple. Although there is some performance overhead.

Most of the content in this chapter is based on Chris Okasaki's work in [2].

3.2 Definition of red-black tree

Red-black tree is a type of self-balancing binary search tree[3]. [2] By using color changing and rotation, red-black tree provides a very simple and straightforward way to keep the tree balanced.

For a binary search tree, we can augment the nodes with a color field, a node can be colored either red or black. We call a binary search tree red-black tree if it satisfies the following 5 properties[2].

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Why this 5 properties can ensure the red-black tree is well balanced? Because they have a key characteristic, the longest path from root to a leaf can't be as 2 times longer than the shortest path.

Please note the 4-th property, which means there won't be two adjacent red nodes. so the shortest path only contains black nodes, any paths longer than the shortest one has interval red nodes. According to property 5, all paths have the same number of black nodes, this finally ensure there won't be any path is 2 times longer than others[3]. Figure 3.4 shows an example red-black tree.

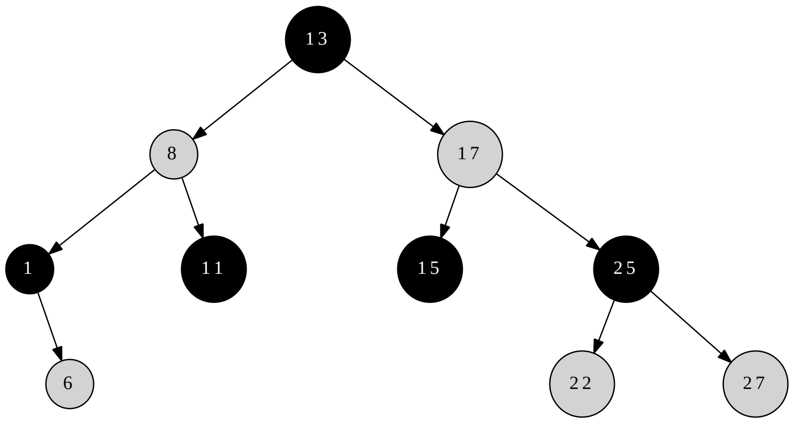


Figure 3.4: An example red-black tree

[2]Red-black tree is one of the equivalent form of 2-3-4 tree (see chapter B-tree about 2-3-4 tree). That is to say, for any 2-3-4 tree, there is at least one red-black tree has the same data order.

All read only operations such as search, min/max are as same as in binary search tree. While only the insertion and deletion are special.

As we have shown in word occurrence example, many implementation of set or map container are based on red-black tree. One example is the C++ Standard library (STL)[6].

As mentioned previously, the only change in data layout is that there is color information augmented to binary search tree. This can be represented as a data field in imperative languages such as C++ like below.

```
enum Color {Red, Black};

template <class T>
struct node{
  Color color;
  T key;
  node* left;
  node* right;
  node* parent;
};
```

In functional settings, we can add the color information in constructors, below is the Haskell example of red-black tree definition.

```
data Color = R | B
data RBTree a = Empty
              | Node Color (RBTree a) a (RBTree a)
```

Exercise 3.2

- Can you prove that a red-black tree with n nodes has height at most $2\lg(n+1)$?

3.3 Insertion

Inserting a new node as what has been mentioned in binary search tree may cause the tree unbalanced. The red-black properties has to be maintained, so we need do some fixing by transform the tree after insertion.

When we insert a new key, one good practice is to always insert it as a red node. As far as the new inserted node isn't the root of the tree, we can keep all properties except the 4-th one. that it may bring two adjacent red nodes.

Functional and procedural implementation have different fixing methods. One is intuitive but has some overhead, the other is a bit complex but has higher performance. Most text books about algorithm introduce the later one. In this chapter, we focus on the former to show how easily a red-black tree insertion algorithm can be realized. The traditional procedural method will be given only for comparison purpose.

As described by Chris Okasaki, there are total 4 cases which violate property 4. All of them has 2 adjacent red nodes. However, they have a uniformed form after fixing[2] as shown in figure 4.3.

Note that this transformation will move the redness one level up. So this is a bottom-up recursive fixing, the last step will make the root node red. According

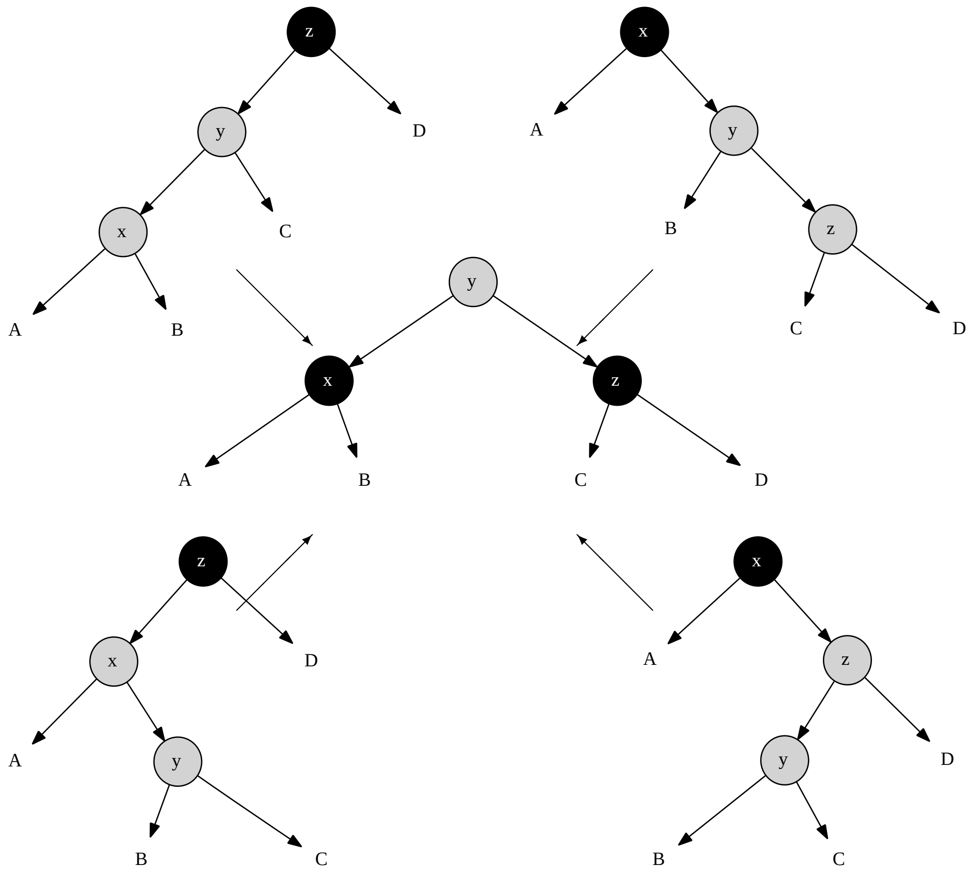


Figure 3.5: 4 cases for balancing a red-black tree after insertion

to property 2, root is always black. Thus we need final fixing to revert the root color to black.

Observing that the 4 cases and the fixed result have strong pattern features, the fixing function can be defined by using the similar method we mentioned in tree rotation. To avoid too long formula, we abbreviate *Color* as $\mathcal{C}$, *Black* as $\mathcal{B}$, and *Red* as $\mathcal{R}$.

$$balance(T) = \begin{cases} node(\mathcal{R}, node(\mathcal{B}, A, x, B), y, node(\mathcal{B}, C, z, D)) & : match(T) \\ T & : otherwise \end{cases} \tag{3.3}$$

where function *node*() can construct a red-black tree node with 4 parameters as color, the left child, the key and the right child. Function *match*() can test if a tree is one of the 4 possible patterns as the following.

$$match(T) = \begin{array}{l} pattern(T) = node(\mathcal{B}, node(\mathcal{R}, node(\mathcal{R}, A, x, B), y, C), z, D) \vee \\ pattern(T) = node(\mathcal{B}, node(\mathcal{R}, A, x, node(\mathcal{R}, B, y, C), z, D)) \vee \\ pattern(T) = node(\mathcal{B}, A, x, node(\mathcal{R}, B, y, node(\mathcal{R}, C, z, D))) \vee \\ pattern(T) = node(\mathcal{B}, A, x, node(\mathcal{R}, node(\mathcal{R}, B, y, C), z, D)) \end{array}$$

With function *balance*() defined, we can modify the previous binary search tree insertion functions to make it work for red-black tree.

$$insert(T, k) = makeBlack(ins(T, k)) \tag{3.4}$$

where

$$ins(T, k) = \begin{cases} node(\mathcal{R}, \phi, k, \phi) & : T = \phi \\ balance(node(ins(L, k), Key, R)) & : k < Key \\ balance(node(L, Key, ins(R, k))) & : otherwise \end{cases} \tag{3.5}$$

L, R, Key represent the left child, right child and the key of a tree.

$$\begin{array}{l} L = left(T) \\ R = right(T) \\ Key = key(T) \end{array}$$

Function *makeBlack*() is defined as the following, it forces the color of a non-empty tree to be black.

$$makeBlack(T) = node(\mathcal{B}, L, Key, R) \tag{3.6}$$

Summarize the above functions and use language supported pattern matching features, we can come to the following Haskell program.

```haskell
insert::(Ord a)⇒RBTree a → a → RBTree a
insert t x = makeBlack $ ins t where
    ins Empty = Node R Empty x Empty
    ins (Node color l k r)
        | x < k     = balance color (ins l) k r
        | otherwise = balance color l k (ins r) --[3]
    makeBlack(Node _ l k r) = Node B l k r
```

```
balance::Color → RBTree a → a → RBTree a → RBTree a
balance B (Node R (Node R a x b) y c) z d =
                Node R (Node B a x b) y (Node B c z d)
balance B (Node R a x (Node R b y c)) z d =
                Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R b y (Node R c z d)) =
                Node R (Node B a x b) y (Node B c z d)
balance B a x (Node R (Node R b y c) z d) =
                Node R (Node B a x b) y (Node B c z d)
balance color l k r = Node color l k r
```

Note that the 'balance' function is changed a bit from the original definition. Instead of passing the tree, we pass the color, the left child, the key and the right child to it. This can save a pair of 'boxing' and 'un-boxing' operations.

This program doesn't handle the case of inserting a duplicated key. However, it is possible to handle it either by overwriting, or skipping. Another option is to augment the data with a linked list[2].

Figure 3.6 shows two red-black trees built from feeding list 11, 2, 14, 1, 7, 15, 5, 8, 4 and 1, 2, ..., 8.

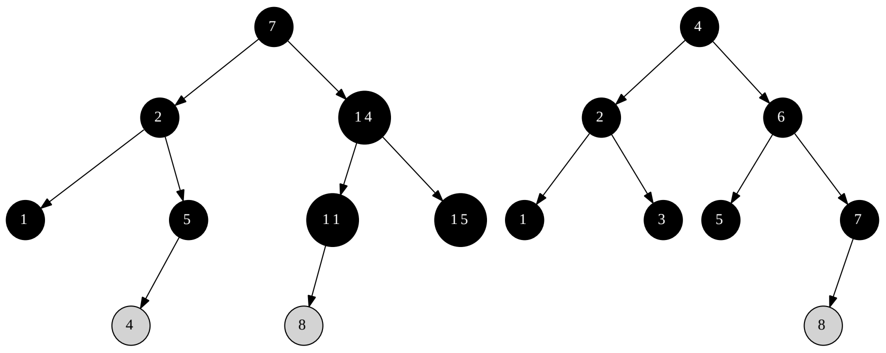


Figure 3.6: insert results generated by the Haskell algorithm

This algorithm shows great simplicity by summarizing the uniform feature from the four different unbalanced cases. It is expressive over the traditional tree rotation approach, that even in programming languages which don't support pattern matching, the algorithm can still be implemented by manually check the pattern. A Scheme/Lisp program is available along with this book can be referenced as an example.

The insertion algorithm takes $O(\lg N)$ time to insert a key to a red-black tree which has N nodes.

Exercise 3.3

- Write a program in an imperative language, such as C, C++ or python to realize the same algorithm in this section. Note that, because there is no language supported pattern matching, you need to test the 4 different cases manually.

3.4 Deletion

Remind the deletion section in binary search tree. Deletion is 'imperative only' for red-black tree as well. In typically practice, it often builds the tree just one time, and performs looking up frequently after that. Okasaki explained why he didn't provide red-black tree deletion in his work in [3]. One reason is that deletions are much messier than insertions.

The purpose of this section is just to show that red-black tree deletion is possible in purely functional settings, although it actually rebuilds the tree because trees are read only in terms of purely functional data structure. In real world, it's up to the user (or actually the programmer) to adopt the proper solution. One option is to mark the node be deleted with a flag, and perform a tree rebuilding when the number of deleted nodes exceeds 50% of the total number of nodes.

Not only in functional settings, even in imperative settings, deletion is more complex than insertion. We face more cases to fix. Deletion may also violate the red black tree properties, so we need fix it after the normal deletion as described in binary search tree.

The deletion algorithm in this book are based on top of a handout in [5]. The problem only happens if you try to delete a black node, because it will violate the 4-th property of red-black tree, which means the number of black node in the path may decreased so that it is not uniformed black-height any more.

When delete a black node, we can resume red-black property number 4 by introducing a 'doubly-black' concept[2]. It means that the although the node is deleted, the blackness is kept by storing it in the parent node. If the parent node is red, it turns to black, However, if it has been already black, it turns to 'doubly-black'.

In order to express the 'doubly-black node', The definition need some modification accordingly.

```
data Color = R | B | BB -- BB: doubly black for deletion
data RBTree a = Empty | BBEmpty -- doubly black empty
              | Node Color (RBTree a) a (RBTree a)
```

When deleting a node, we first perform the same deleting algorithm in binary search tree mentioned in previous chapter. After that, if the node to be sliced out is black, we need fix the tree to keep the red-black properties. Let's denote the empty tree as ϕ, and for non-empty tree, it can be decomposed to $node(Color, L, Key, R)$ for its color, left sub-tree, key and the right sub-tree. The delete function is defined as the following.

$$delete(T, k) = blackenRoot(del(T, k)) \tag{3.7}$$

where

$$del(T,k) = \left\{ \begin{array}{rcl} \phi & : & T = \phi \\ fixBlack^2(node(\mathcal{C}, del(L,k), Key, R)) & : & k < Key \\ fixBlack^2(node(\mathcal{C}, L, Key, del(R,k))) & : & k > Key \\ = \left\{ \begin{array}{rcl} mkBlk(R) & : & \mathcal{C} = \mathcal{B} \\ R & : & otherwise \end{array} \right. & : & L = \phi \\ = \left\{ \begin{array}{rcl} mkBlk(L) & : & \mathcal{C} = \mathcal{B} \\ L & : & otherwise \end{array} \right. & : & R = \phi \\ fixBlack^2(node(\mathcal{C}, L, k', del(R, k'))) & : & otherwise \end{array} \right. \tag{3.8}$$

The real deleting happens inside function *del*. For the trivial case, that the tree is empty, the deletion result is ϕ; If the key to be deleted is less than the key of the current node, we recursively perform deletion on its left sub-tree; if it is bigger than the key of the current node, then we recursively delete the key from the right sub-tree; Because it may bring doubly-blackness, so we need fix it.

If the key to be deleted is equal to the key of the current node, we need splice it out. If one of its children is empty, we just replace the node by the other one and reserve the blackness of this node. otherwise we cut and past the minimum element $k' = min(R)$ from the right sub-tree.

Function *delete* just forces the result tree of *del* to have a black root. This is realized by function *blackenRoot*.

$$blackenRoot(T) = \left\{ \begin{array}{rcl} \phi & : & T = \phi \\ node(\mathcal{B}, L, Key, R) & : & otherwise \end{array} \right. \tag{3.9}$$

Compare with the *makeBlack* function, which is defined in red-black tree insertion section, they are almost same, except for the case of empty tree. This is only valid in deletion, because insertion can't result an empty tree, while deletion may.

Function *mkBlk* is defined to reserved the blackness of a node. If the node to be sliced isn't black, this function won't be applied, otherwise, it turns a red node to black and turns a black node to doubly-black. This function also marks an empty tree to doubly-black empty.

$$mkBlk(T) = \left\{ \begin{array}{rcl} \Phi & : & T = \phi \\ node(\mathcal{B}, L, Key, R) & : & \mathcal{C} = \mathcal{R} \\ node(\mathcal{B}^2, L, Key, R) & : & \mathcal{C} = \mathcal{B} \\ T & : & otherwise \end{array} \right. \tag{3.10}$$

where Φ means doubly-black empty node and $\mathcal{B}^2$ is the doubly-black color.

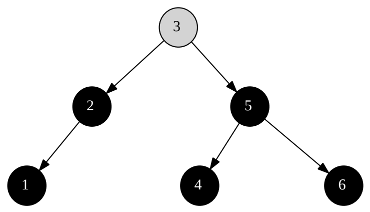
Summarizing the above functions yields the following Haskell program.

```haskell
delete::(Ord a)⇒RBTree a → a → RBTree a
delete t x = blackenRoot(del t x) where
    del Empty _ = Empty
    del (Node color l k r) x
        | x < k = fixDB color (del l x) k r
        | x > k = fixDB color l k (del r x)
        -- x == k, delete this node
        | isEmpty l = if color==B then makeBlack r else r
        | isEmpty r = if color==B then makeBlack l else l
        | otherwise = fixDB color l k' (del r k') where k'= min r
    blackenRoot (Node _ l k r) = Node B l k r
    blackenRoot _ = Empty

makeBlack::RBTree a → RBTree a
makeBlack (Node B l k r) = Node BB l k r -- doubly black
makeBlack (Node _ l k r) = Node B l k r
makeBlack Empty = BBEmpty
makeBlack t = t
```

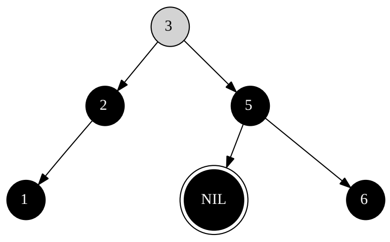
The final attack to the red-black tree deletion algorithm is to realize the *fixBlack*[2] function. The purpose of this function is to eliminate the 'doubly-black' colored node by rotation and color changing.

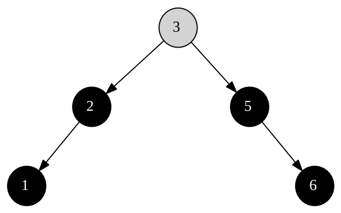
Let's solve the doubly-black empty node first. For any node, If one of its child is doubly-black empty, and the other child is non-empty, we can safely replace the doubly-black empty with a normal empty node.

Like figure 3.7, if we are going to delete the node 4 from the tree (Instead show the whole tree, only part of the tree is shown), the program will use a doubly-black empty node to replace node 4. In the figure, the doubly-black node is shown in black circle with 2 edges. It means that for node 5, it has a doubly-black empty left child and has a right non-empty child (a leaf node with key 6). In such case we can safely change the doubly-black empty to normal empty node. which won't violate any red-black properties.



(a) Delete 4 from the tree.

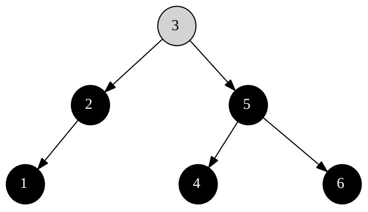




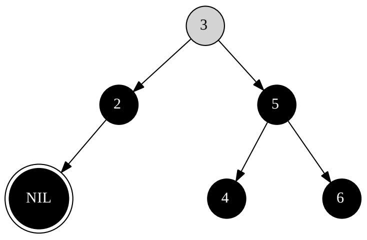
(b) After 4 is sliced off, it is doubly-black empty. (c) We can safely change it to normal NIL.

Figure 3.7: One child is doubly-black empty node, the other child is non-empty

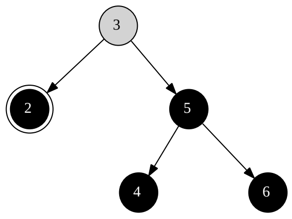
On the other hand, if a node has a doubly-black empty node and the other child is empty, we have to push the doubly-blackness up one level. For example in figure 3.8, if we want to delete node 1 from the tree, the program will use a doubly-black empty node to replace 1. Then node 2 has a doubly-black empty node and has an empty right node. In such case we must mark node 2 as doubly-black after change its left child back to empty.



(a) Delete 1 from the tree.



(b) After 1 is sliced off, it is doubly-black empty.



(c) We must push the doubly-blackness up to node 2.

Figure 3.8: One child is doubly-black empty node, the other child is empty.

Based on above analysis, in order to fix the doubly-black empty node, we define the function partially like the following.

$$
fixBlack^2(T) = \begin{cases} node(\mathcal{B}^2, \phi, Key, \phi) & : (L = \phi \land R = \Phi) \lor (L = \Phi \land R = \phi) \\ node(\mathcal{C}, \phi, Key, R) & : L = \Phi \land R \neq \phi \\ node(\mathcal{C}, L, Key, \phi) & : R = \Phi \land L \neq \phi \\ ... & : ... \end{cases} \tag{3.11}
$$

After dealing with doubly-black empty node, we need to fix the case that the sibling of the doubly-black node is black and it has one red child. In this situation, we can fix the doubly-blackness with one rotation. Actually there are 4 different sub-cases, all of them can be transformed to one uniformed pattern. They are shown in the figure 3.9. These cases are described in [2] as case 3 and case 4.

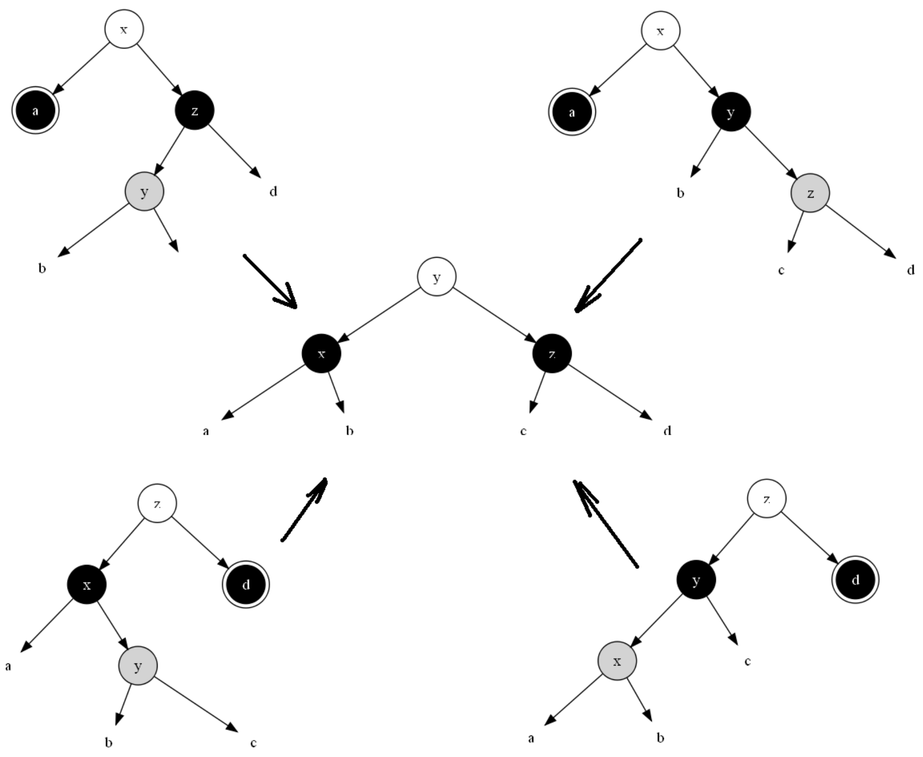


Figure 3.9: Fix the doubly black by rotation, the sibling of the doubly-black node is black, and it has one red child.

The handling of these 4 sub-cases can be defined on top of formula 3.11.

$$fixBlack^2(T) = \left\{ \begin{array}{rcl} ... & : & ... \\ node(\mathcal{C}, node(\mathcal{B}, mkBlk(A), x, B), y, node(\mathcal{B}, C, z, D)) & : & p1.1 \\ node(\mathcal{C}, node(\mathcal{B}, A, x, B), y, node(\mathcal{B}, C, z, mkBlk(D))) & : & p1.2 \\ ... & : & ... \end{array} \right. \tag{3.12}$$

where $p1.1$ and $p1.2$ each represent 2 patterns as the following.

$$p1.1 = \left\{ \begin{array}{l} node(\mathcal{C}, A, x, node(\mathcal{B}, node(\mathcal{R}, B, y, C), z, D)) \land Color(A) = \mathcal{B}^2 \\ \lor \\ node(\mathcal{C}, A, x, node(\mathcal{B}, B, y, node(\mathcal{R}, C, z, D))) \land Color(A) = \mathcal{B}^2 \end{array} \right\}$$

$$p1.2 = \left\{ \begin{array}{l} node(\mathcal{C}, node(\mathcal{B}, A, x, node(\mathcal{R}, B, y, C)), z, D) \land Color(D) = \mathcal{B}^2 \\ \lor \\ node(\mathcal{C}, node(\mathcal{B}, node(\mathcal{R}, A, x, B), y, C), z, D) \land Color(D) = \mathcal{B}^2 \end{array} \right\}$$

Besides the above cases, there is another one that not only the sibling of the doubly-black node is black, but also its two children are black. We can change the color of the sibling node to red; resume the doubly-black node to black and propagate the doubly-blackness one level up to the parent node as shown in figure 3.10. Note that there are two symmetric sub-cases. This case is described in [2] as case 2.

We go on adding this fixing after formula 3.12.

$$fixBlack^2(T) = \left\{ \begin{array}{rcl} ... & : & ... \\ mkBlk(node(\mathcal{C}, mkBlk(A), x, node(\mathcal{R}, B, y, C))) & : & p2.1 \\ mkBlk(node(\mathcal{C}, node(\mathcal{R}, A, x, B), y, mkBlk(C))) & : & p2.2 \\ ... & : & ... \end{array} \right. \tag{3.13}$$

where $p2.1$ and $p2.2$ are two patterns as below.

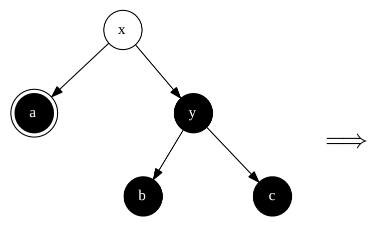
$$p2.1 = \left\{ \begin{array}{l} node(\mathcal{C}, A, x, node(\mathcal{B}, B, y, C)) \land \\ Color(A) = \mathcal{B}^2 \land Color(B) = Color(C) = \mathcal{B} \end{array} \right\}$$

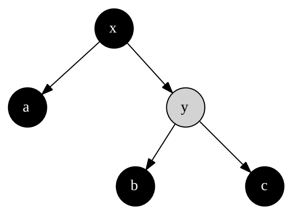
$$p2.2 = \left\{ \begin{array}{l} node(\mathcal{C}, node(\mathcal{B}, A, x, B), y, C) \land \\ Color(C) = \mathcal{B}^2 \land Color(A) = Color(B) = \mathcal{B} \end{array} \right\}$$

There is a final case left, that the sibling of the doubly-black node is red. We can do a rotation to change this case to pattern $p1.1$ or $p1.2$. Figure 3.11 shows about it.

We can finish formula 3.13 with 3.14.

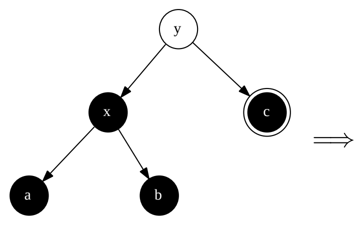
$$fixBlack^2(T) = \left\{ \begin{array}{rcl} ... & : & ... \\ fixBlack^2(\mathcal{B}, fixBlack^2(node(\mathcal{R}, A, x, B), y, C) & : & p3.1 \\ fixBlack^2(\mathcal{B}, A, x, fixBlack^2(node(\mathcal{R}, B, y, C)) & : & p3.2 \\ T & : & otherwise \end{array} \right. \tag{3.14}$$

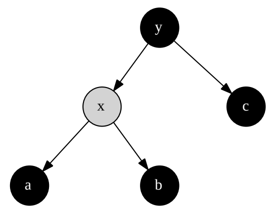




(a) Color of x can be either black or red.

(b) If x was red, then it becomes black, otherwise, it becomes doubly-black.





(c) Color of y can be either black or red.

(d) If y was red, then it becomes black, otherwise, it becomes doubly-black.

Figure 3.10: propagate the blackness up.

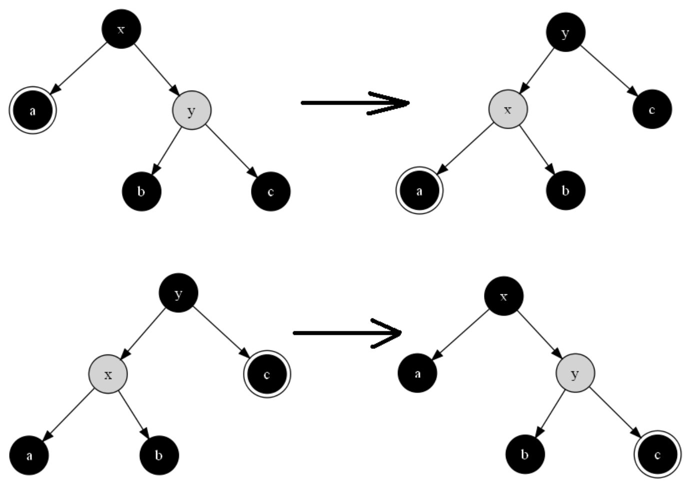


Figure 3.11: The sibling of the doubly-black node is red.

where $p3.1$ and $p3.2$ are two patterns as the following.

$$p3.1 = \{Color(T) = \mathcal{B} \wedge Color(L) = \mathcal{B}^2 \wedge Color(R) = \mathcal{R}\}$$

$$p3.2 = \{Color(T) = \mathcal{B} \wedge Color(L) = \mathcal{R} \wedge Color(R) = \mathcal{B}^2\}$$

This two cases are described in [2] as case 1.

Fixing the doubly-black node with all above different cases is a recursive function. There are two termination conditions. One contains pattern $p1.1$ and $p1.2$, The doubly-black node was eliminated. The other cases may continuously propagate the doubly-blackness from bottom to top till the root. Finally the algorithm marks the root node as black anyway. The doubly-blackness will be removed.

Put formula 3.11, 3.12, 3.13, and 3.14 together, we can write the final Haskell program.

```
fixDB::Color → RBTree a → a → RBTree a → RBTree a
fixDB color BBEmpty k Empty = Node BB Empty k Empty
fixDB color BBEmpty k r = Node color Empty k r
fixDB color Empty k BBEmpty = Node BB Empty k Empty
fixDB color l k BBEmpty = Node color l k Empty
-- the sibling is black, and it has one red child
fixDB color a@(Node BB _ _ _) x (Node B (Node R b y c) z d) =
      Node color (Node B (makeBlack a) x b) y (Node B c z d)
fixDB color a@(Node BB _ _ _) x (Node B b y (Node R c z d)) =
      Node color (Node B (makeBlack a) x b) y (Node B c z d)
fixDB color (Node B a x (Node R b y c)) z d@(Node BB _ _ _) =
      Node color (Node B a x b) y (Node B c z (makeBlack d))
fixDB color (Node B (Node R a x b) y c) z d@(Node BB _ _ _) =
      Node color (Node B a x b) y (Node B c z (makeBlack d))
-- the sibling and its 2 children are all black, propagate the blackness up
fixDB color a@(Node BB _ _ _) x (Node B b@(Node B _ _ _) y c@(Node B _ _ _))
    = makeBlack (Node color (makeBlack a) x (Node R b y c))
fixDB color (Node B a@(Node B _ _ _) x b@(Node B _ _ _)) y c@(Node BB _ _ _)
    = makeBlack (Node color (Node R a x b) y (makeBlack c))
-- the sibling is red
fixDB B a@(Node BB _ _ _) x (Node R b y c) = fixDB B (fixDB R a x b) y c
fixDB B (Node R a x b) y c@(Node BB _ _ _) = fixDB B a x (fixDB R b y c)
-- otherwise
fixDB color l k r = Node color l k r
```

The deletion algorithm takes $O(\lg N)$ time to delete a key from a red-black tree with N nodes.

Exercise 3.4

- As we mentioned in this section, deletion can be implemented by just marking the node as deleted without actually removing it. Once the number of marked nodes exceeds 50% of the total node number, a tree re-build is performed. Try to implement this method in your favorite programming language.

3.5 Imperative red-black tree algorithm ⋆

We almost finished all the content in this chapter. By induction the patterns, we can implement the red-black tree in a simple way compare to the imperative tree rotation solution. However, we should show the comparator for completeness.

For insertion, the basic idea is to use the similar algorithm as described in binary search tree. And then fix the balance problem by rotation and return the final result.

```
function INSERT($T, k$)
    $root \leftarrow T$
    $x \leftarrow$ CREATE-LEAF($k$)
    COLOR($x$) $\leftarrow RED$
    $parent \leftarrow NIL$
    while $T \neq NIL$ do
        $parent \leftarrow T$
        if $k <$ KEY($T$) then
            $T \leftarrow$ LEFT($T$)
        else
            $T \leftarrow$ RIGHT($T$)
    PARENT($x$) $\leftarrow parent$
    if $parent = NIL$ then                          ▷ tree $T$ is empty
        return $x$
    else if $k <$ KEY($parent$) then
        LEFT($parent$) $\leftarrow x$
    else
        RIGHT($parent$) $\leftarrow x$
    return INSERT-FIX($root, x$)
```

The only difference from the binary search tree insertion algorithm is that we set the color of the new node as red, and perform fixing before return. It is easy to translate the pseudo code to real imperative programming language, for instance Python [3].

```
def rb_insert(t, key):
    root = t
    x = Node(key)
    parent = None
    while(t):
        parent = t
        if(key < t.key):
            t = t.left
        else:
            t = t.right
    if parent is None: #tree is empty
        root = x
    elif key < parent.key:
        parent.set_left(x)
    else:
        parent.set_right(x)
    return rb_insert_fix(root, x)
```

[3]C, and C++ source codes are available along with this book

There are 3 base cases for fixing, and if we take the left-right symmetric into consideration. there are total 6 cases. Among them two cases can be merged together, because they all have uncle node in red color, we can toggle the parent color and uncle color to black and set grand parent color to red. With this merging, the fixing algorithm can be realized as the following.

```
1: function INSERT-FIX($T, x$)
2:     while PARENT($x$) $\neq NIL$ and COLOR(PARENT($x$)) $= RED$ do
3:         if COLOR(UNCLE($x$)) $= RED$ then          ▷ Case 1, x's uncle is red
4:             COLOR(PARENT($x$)) $\leftarrow BLACK$
5:             COLOR(GRAND-PARENT($x$)) $\leftarrow RED$
6:             COLOR(UNCLE($x$)) $\leftarrow BLACK$
7:             $x \leftarrow$ GRANDPARENT($x$)
8:         else                                        ▷ x's uncle is black
9:             if PARENT($x$) = LEFT(GRAND-PARENT($x$)) then
10:                if $x$ = RIGHT(PARENT($x$)) then  ▷ Case 2, x is a right child
11:                    $x \leftarrow$ PARENT($x$)
12:                    $T \leftarrow$ LEFT-ROTATE($T, x$)
                                                       ▷ Case 3, x is a left child
13:                COLOR(PARENT($x$)) $\leftarrow BLACK$
14:                COLOR(GRAND-PARENT($x$)) $\leftarrow RED$
15:                $T \leftarrow$ RIGHT-ROTATE($T$, GRAND-PARENT($x$))
16:            else
17:                if $x$ = LEFT(PARENT($x$)) then           ▷ Case 2, Symmetric
18:                    $x \leftarrow$ PARENT($x$)
19:                    $T \leftarrow$ RIGHT-ROTATE($T, x$)
                                                       ▷ Case 3, Symmetric
20:                COLOR(PARENT($x$)) $\leftarrow BLACK$
21:                COLOR(GRAND-PARENT($x$)) $\leftarrow RED$
22:                $T \leftarrow$ LEFT-ROTATE($T$, GRAND-PARENT($x$))
23:     COLOR($T$) $\leftarrow BLACK$
24:     return $T$
```

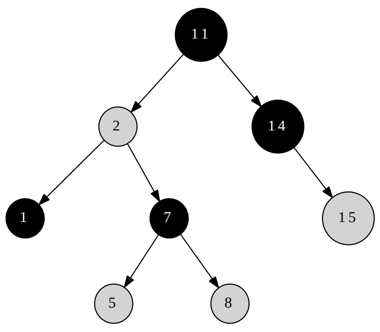
This program takes $O(\lg N)$ time to insert a new key to the red-black tree. Compare this pseudo code and the *balance* function we defined in previous section, we can see the difference. They differ not only in terms of simplicity, but also in logic. Even if we feed the same series of keys to the two algorithms, they may build different red-black trees. There is a bit performance overhead in the pattern matching algorithm. Okasaki discussed about the difference in detail in his paper[2].

Translate the above algorithm to Python yields the below program.

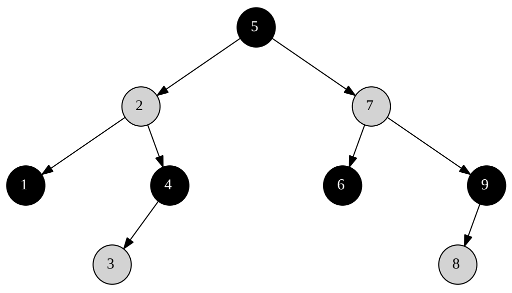
```
# Fix the red→red violation
def rb_insert_fix(t, x):
    while(x.parent and x.parent.color==RED):
        if x.uncle().color == RED:
            #case 1: ((a:R x:R b) y:B c:R) ==> ((a:R x:B b) y:R c:B)
            set_color([x.parent, x.grandparent(), x.uncle()],
                      [BLACK, RED, BLACK])
            x = x.grandparent()
        else:
            if x.parent == x.grandparent().left:
                if x == x.parent.right:
```

```
                #case 2: ((a x:R b:R) y:B c) ⟹ case 3
                x = x.parent
                t=left_rotate(t, x)
            # case 3: ((a:R x:R b) y:B c) ⟹ (a:R x:B (b y:R c))
            set_color([x.parent, x.grandparent()], [BLACK, RED])
            t=right_rotate(t, x.grandparent())
        else:
            if x == x.parent.left:
                #case 2': (a x:B (b:R y:R c)) ⟹ case 3'
                x = x.parent
                t = right_rotate(t, x)
            # case 3': (a x:B (b y:R c:R)) ⟹ ((a x:R b) y:B c:R)
            set_color([x.parent, x.grandparent()], [BLACK, RED])
            t=left_rotate(t, x.grandparent())
t.color = BLACK
return t
```

Figure 3.12 shows the results of feeding same series of keys to the above python insertion program. Compare them with figure 3.6, one can tell the difference clearly.



(a)



(b)

Figure 3.12: Red-black trees created by imperative algorithm.

We skip the red-black tree delete algorithm in imperative settings, because it is even more complex than the insertion. The implementation of deleting is left as an exercise of this chapter.

Exercise 3.5

- Implement the red-black tree deleting algorithm in your favorite imperative programming language. you can refer to [2] for algorithm details.

3.6 More words

Red-black tree is the most popular implementation of balanced binary search tree. Another one is the AVL tree, which we'll introduce in next chapter. Red-black tree can be a good start point for more data structures. If we extend the number of children from 2 to K, and keep the balance as well, it leads to B-tree, If we store the data along with edge but not inside node, it leads to Tries. However, the multiple cases handling and the long program tends to make new comers think red-black tree is complex.

Okasaki's work helps making the red-black tree much easily understand. There are many implementation in other programming languages in that manner [7]. It's also inspired me to find the pattern matching solution for Splay tree and AVL tree etc.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. ISBN:0262032937. The MIT Press. 2001

[2] Chris Okasaki. “FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting”. J. Functional Programming. 1998

[3] Chris Okasaki. “Ten Years of Purely Functional Data Structures”. http://okasaki.blogspot.com/2008/02/ten-years-of-purely-functional-data.html

[4] Wikipedia. “Red-black tree”. http://en.wikipedia.org/wiki/Red-black_tree

[5] Lyn Turbak. “Red-Black Trees”. cs.wellesley.edu/ cs231/fall01/red-black.pdf Nov. 2, 2001.

[6] SGI STL. http://www.sgi.com/tech/stl/

[7] Pattern matching. http://rosettacode.org/wiki/Pattern_matching

Chapter 4

AVL tree

4.1 Introduction

4.1.1 How to measure the balance of a tree?

Besides red-black tree, are there any other intuitive solutions of self-balancing binary search tree? In order to measure how balancing a binary search tree, one idea is to compare the height of the left sub-tree and right sub-tree. If they differs a lot, the tree isn't well balanced. Let's denote the difference height between two children as below

$$\delta(T) = |L| - |R| \tag{4.1}$$

Where $|T|$ means the height of tree T, and L, R denotes the left sub-tree and right sub-tree.

If $\delta(T) = 0$, The tree is definitely balanced. For example, a complete binary tree has $N = 2^h - 1$ nodes for height h. There is no empty branches unless the leafs. Another trivial case is empty tree. $\delta(\phi) = 0$. The less absolute value of $\delta(T)$ the more balancing the tree is.

We define $\delta(T)$ as the *balance factor* of a binary search tree.

4.2 Definition of AVL tree

An AVL tree is a special binary search tree, that all sub-trees satisfying the following criteria.

$$|\delta(T)| \leq 1 \tag{4.2}$$

The absolute value of balance factor is less than or equal to 1, which means there are only three valid values, -1, 0 and 1. Figure 4.1 shows an example AVL tree.

Why AVL tree can keep the tree balanced? In other words, Can this definition ensure the height of the tree as $O(\lg N)$ where N is the number of the nodes in the tree? Let's prove this fact.

For an AVL tree of height h, The number of nodes varies. It can have at most $2^h - 1$ nodes for a complete binary tree. We are interesting about how

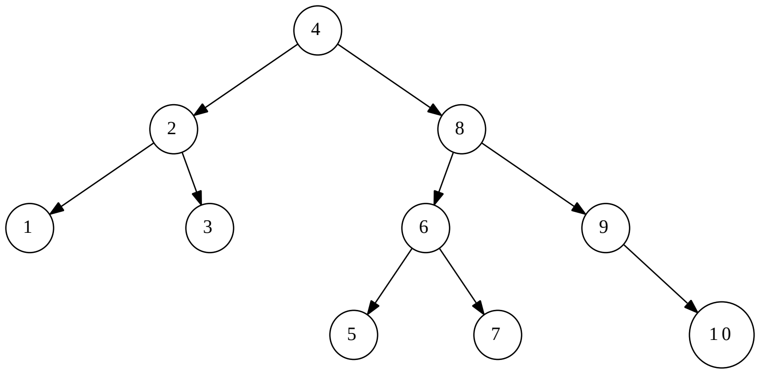


Figure 4.1: An example AVL tree

many nodes there are at least. Let's denote the minimum number of nodes for height h AVL tree as $N(h)$. It's obvious for the trivial cases as below.

- For empty tree, $h = 0$, $N(0) = 0$;
- For a singleton root, $h = 1$, $N(1) = 1$;

What's the situation for common case $N(h)$? Figure 4.2 shows an AVL tree T of height h. It contains three part, the root node, and two sub trees A, B. We have the following fact.

$$h = max(height(L), height(R)) + 1 \tag{4.3}$$

We immediately know that, there must be one child has height $h - 1$. Let's say $height(A) = h - 1$. According to the definition of AVL tree, we have. $|height(A) - height(B)| \leq 1$. This leads to the fact that the height of other tree B can't be lower than $h - 2$, So the total number of the nodes of T is the number of nodes in tree A, and B plus 1 (for the root node). We exclaim that.

$$N(h) = N(h - 1) + N(h - 2) + 1 \tag{4.4}$$

This recursion reminds us the famous Fibonacci series. Actually we can transform it to Fibonacci series by defining $N'(h) = N(h) + 1$. So equation 4.4 changes to.

$$N'(h) = N'(h - 1) + N'(h - 2) \tag{4.5}$$

Lemma 4.2.1. *Let $N(h)$ be the minimum number of nodes for an AVL tree with height h. and $N'(h) = N(h) + 1$, then*

$$N'(h) \geq \phi^h \tag{4.6}$$

Where $\phi = \frac{\sqrt{5}+1}{2}$ is the golden ratio.

Proof. For the trivial case, we have

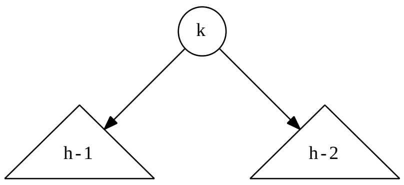


Figure 4.2: An AVL tree with height h, one of the sub-tree with height $h-1$, the other is $h-2$

- $h = 0$, $N'(0) = 1 \geq \phi^0 = 1$
- $h = 1$, $N'(1) = 2 \geq \phi^1 = 1.618...$

For the induction case, suppose $N'(h) \geq \phi^h$.

$$\begin{array}{rll} N'(h+1) & = N'(h) + N'(h-1) & \{Fibonacci\} \\ & \geq \phi^h + \phi^{h-1} & \\ & = \phi^{h-1}(\phi + 1) & \{\phi + 1 = \phi^2 = \frac{\sqrt{5}+3}{2}\} \\ & = \phi^{h+1} & \end{array}$$

□

From Lemma 4.2.1, we immediately get

$$h \leq log_\phi(N+1) = log_\phi(2) \cdot \lg(N+1) \approx 1.44 \lg(N+1) \tag{4.7}$$

It tells that the height of AVL tree is proportion to $O(\lg N)$, which means that AVL tree is balanced.

During the basic mutable tree operations such as insertion and deletion, if the balance factor changes to any invalid value, some fixing has to be performed to resume $|\delta|$ within 1. Most implementations utilize tree rotations. In this chapter, we'll show the pattern matching solution which is inspired by Okasaki's red-black tree solution[2]. Because of this modify-fixing approach, AVL tree is also a kind of self-balancing binary search tree. For comparison purpose, we'll also show the procedural algorithms.

Of course we can compute the δ value recursively, another option is to store the balance factor inside each nodes, and update them when we modify the tree. The latter one avoid computing the same value every time.

Based on this idea, we can add one data field δ to the original binary search tree as the following C++ code example [1].

```
template <class T>
struct node{
  int delta;
  T key;
  node* left;
```

[1]Some implementations store the height of a tree instead of δ as in [5]

```
  node* right;
  node* parent;
};
```

In purely functional setting, some implementation use different constructor to store the δ information. for example in [1], there are 4 constructors, E, N, P, Z defined. E for empty tree, N for tree with negative 1 balance factor, P for tree with positive 1 balance factor and Z for zero case.

In this chapter, we'll explicitly store the balance factor inside the node.

```
data AVLTree a = Empty
               | Br (AVLTree a) a (AVLTree a) Int
```

The immutable operations, including looking up, finding the maximum and minimum elements are all same as the binary search tree. We'll skip them and focus on the mutable operations.

4.3 Insertion

Insert a new element to an AVL tree may violate the AVL tree property that the δ absolute value exceeds 1. To resume it, one option is to do the tree rotation according to the different insertion cases. Most implementation is based on this approach

Another way is to use the similar pattern matching method mentioned by Okasaki in his red-black tree implementation [2]. Inspired by this idea, it is possible to provide a simple and intuitive solution.

When insert a new key to the AVL tree, the balance factor of the root may changes in range $[-1, 1]$, and the height may increase at most by one, which we need recursively use this information to update the δ value in upper level nodes. We can define the result of the insertion algorithm as a pair of data $(T', \Delta H)$. Where T' is the new tree and ΔH is the increment of height. Let's denote function $first(pair)$ which can return the first element in a pair. We can modify the binary search tree insertion algorithm as the following to handle AVL tree.

$$insert(T, k) = first(ins(T, k)) \tag{4.8}$$

where

$$ins(T, k) = \begin{cases} (node(\phi, k, \phi, 0), 1) & : T = \phi \\ (tree(ins(L, k), Key, (R, 0)), \Delta) & : k < Key \\ (tree((L, 0), Key, ins(R, k)), \Delta) & : otherwise \end{cases} \tag{4.9}$$

L, R, Key, Δ represent the left child, right child, the key and the balance factor of a tree.

$$\begin{array}{l} L = left(T) \\ R = right(T) \\ Key = key(T) \\ \Delta = \delta(T) \end{array}$$

When we insert a new key k to a AVL tree T, if the tree is empty, we just need create a leaf node with k, set the balance factor as 0, and the height is increased by one. This is the trivial case. Function $node()$ is defined to build a tree by taking a left sub-tree, a right sub-tree, a key and a balance factor.

If T isn't empty, we need compare the Key with k. If k is less than the key, we recursively insert it to the left child, otherwise we insert it into the right child.

As we defined above, the result of the recursive insertion is a pair like $(L', \Delta H_l)$, we need do balancing adjustment as well as updating the increment of height. Function $tree()$ is defined to dealing with this task. It takes 4 parameters as $(L', \Delta H_l)$, Key, $(R', \Delta H_r)$, and Δ. The result of this function is defined as $(T', \Delta H)$, where T' is the new tree after adjustment, and ΔH is the new increment of height which is defined as

$$\Delta H = |T'| - |T| \tag{4.10}$$

This can be further detailed deduced in 4 cases.

$$\begin{array}{rl} \Delta H & = |T'| - |T| \\ & = 1 + max(|R'|, |L'|) - (1 + max(|R|, |L|)) \\ & = max(|R'|, |L'|) - max(|R|, |L|) \\ & = \left\{ \begin{array}{ccc} \Delta H_r & : & \Delta \geq 0 \land \Delta' \geq 0 \\ \Delta + \Delta H_r & : & \Delta \leq 0 \land \Delta' \geq 0 \\ \Delta H_l - \Delta & : & \Delta \geq 0 \land \Delta' \leq 0 \\ \Delta H_l & : & otherwise \end{array} \right. \end{array} \tag{4.11}$$

To prove this equation, note the fact that the height can't increase both in left and right with only one insertion.

These 4 cases can be explained from the definition of balance factor definition that it equal to the difference from the right sub tree and left sub tree.

- If $\Delta \geq 0$ and $\Delta' \geq 0$, it means that the height of right sub tree isn't less than the height of left sub tree both before insertion and after insertion. In this case, the increment in height of the tree is only 'contributed' from the right sub tree, which is ΔH_r.
- If $\Delta \leq 0$, which means the height of left sub tree isn't less than the height of right sub tree before, and it becomes $\Delta' \geq 0$, which means that the height of right sub tree increases due to insertion, and the left side keeps same ($|L'| = |L|$). So the increment in height is

$$\begin{array}{rll} \Delta H & = max(|R'|, |L'|) - max(|R|, |L|) & \{\Delta \leq 0 \land \Delta' \geq 0\} \\ & = |R'| - |L'| & \{|L| = |L'|\} \\ & = |R| + \Delta H_r - |L| & \\ & = \Delta + \Delta H_r & \end{array}$$

- For the case $\Delta \geq 0 \land \Delta' \leq 0$, Similar as the second one, we can get.

$$\begin{array}{rll} \Delta H & = max(|R'|, |L'|) - max(|R|, |L|) & \{\Delta \geq 0 \land \Delta' \leq 0\} \\ & = |L'| - |R| & \\ & = |L| + \Delta H_l - |R| & \\ & = \Delta H_l - \Delta & \end{array}$$

- For the last case, the both Δ and Δ' is no bigger than zero, which means the height left sub tree is always greater than or equal to the right sub tree, so the increment in height is only 'contributed' from the right sub tree, which is ΔH_l.

The next problem in front of us is how to determine the new balancing factor value Δ' before performing balancing adjustment. According to the definition of AVL tree, the balancing factor is the height of right sub tree minus the height of right sub tree. We have the following facts.

$$\begin{array}{rl}\Delta' & = |R'| - |L'| \\ & = |R| + \Delta H_r - (|L| + \Delta H_l) \\ & = |R| - |L| + \Delta H_r - \Delta H_l \\ & = \Delta + \Delta H_r - \Delta H_l\end{array} \tag{4.12}$$

With all these changes in height and balancing factor get clear, it's possible to define the $tree()$ function mentioned in (4.9).

$$tree((L', \Delta H_l), Key, (R', \Delta H_r), \Delta) = balance(node(L', Key, R', \Delta'), \Delta H) \tag{4.13}$$

Before we moving into details of balancing adjustment, let's translate the above equations to real programs in Haskell.

First is the insert function.

```
insert::(Ord a)⇒AVLTree a → a → AVLTree a
insert t x = fst $ ins t where
    ins Empty = (Br Empty x Empty 0, 1)
    ins (Br l k r d)
        | x < k     = tree (ins l) k (r, 0) d
        | x == k    = (Br l k r d, 0)
        | otherwise = tree (l, 0) k (ins r) d
```

Here we also handle the case that inserting a duplicated key (which means the key has already existed.) as just overwriting.

```
tree::(AVLTree a, Int) → a → (AVLTree a, Int) → Int → (AVLTree a, Int)
tree (l, dl) k (r, dr) d = balance (Br l k r d', delta) where
    d' = d + dr - dl
    delta = deltaH d d' dl dr
```

And the definition of height increment is as below.

```
deltaH :: Int → Int → Int → Int → Int
deltaH d d' dl dr
       | d ≥0 && d' ≥0 = dr
       | d ≤0 && d' ≥0 = d+dr
       | d ≥0 && d' ≤0 = dl - d
       | otherwise = dl
```

4.3.1 Balancing adjustment

As the pattern matching approach is adopted in doing re-balancing. We need consider what kind of patterns violate the AVL tree property.

Figure 4.3 shows the 4 cases which need fix. For all these 4 cases the balancing factors are either -2, or +2 which exceed the range of $[-1, 1]$. After balancing adjustment, this factor turns to be 0, which means the height of left sub tree is equal to the right sub tree.

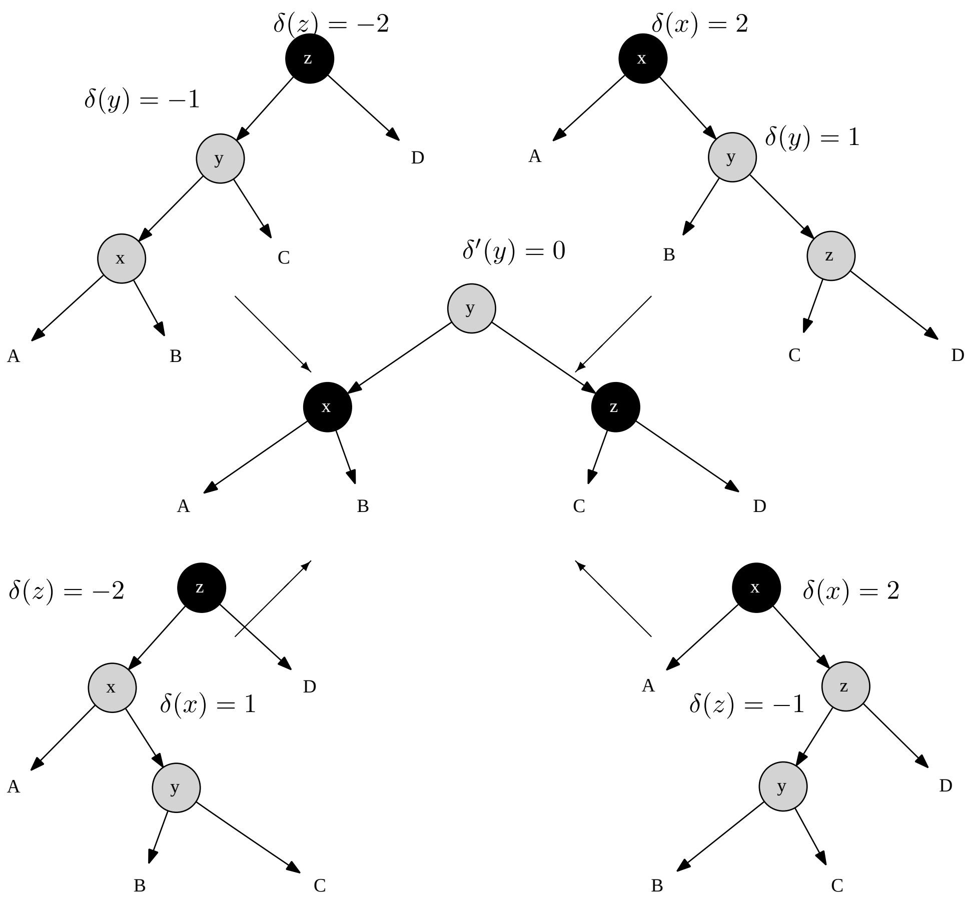


Figure 4.3: 4 cases for balancing a AVL tree after insertion

We call these four cases left-left lean, right-right lean, right-left lean, and left-right lean cases in clock-wise direction from top-left. We denote the balancing factor before fixing as $\delta(x), \delta(y)$, and $\delta(z)$, while after fixing, they changes to $\delta'(x), \delta'(y)$, and $\delta'(z)$ respectively.

We'll next prove that, after fixing, we have $\delta(y) = 0$ for all four cases, and we'll provide the result values of $\delta'(x)$ and $\delta'(z)$.

Left-left lean case

As the structure of sub tree x doesn't change due to fixing, we immediately get $\delta'(x) = \delta(x)$.

Since $\delta(y) = -1$ and $\delta(z) = -2$, we have

$$\begin{array}{l} \delta(y) = |C| - |x| = -1 \Rightarrow |C| = |x| - 1 \\ \delta(z) = |D| - |y| = -2 \Rightarrow |D| = |y| - 2 \end{array} \tag{4.14}$$

After fixing.

$$\begin{array}{rll} \delta'(z) & = |D| - |C| & \{From(4.14)\} \\ & = |y| - 2 - (|x| - 1) & \\ & = |y| - |x| - 1 & \{x \text{ is child of } y \Rightarrow |y| - |x| = 1\} \\ & = 0 & \end{array} \tag{4.15}$$

For $\delta'(y)$, we have the following fact after fixing.

$$\begin{array}{rll} \delta'(y) & = |z| - |x| & \\ & = 1 + max(|C|, |D|) - |x| & \{\text{By (4.15), we have} |C| = |D|\} \\ & = 1 + |C| - |x| & \{\text{By (4.14)}\} \\ & = 1 + |x| - 1 - |x| & \\ & = 0 & \end{array} \tag{4.16}$$

Summarize the above results, the left-left lean case adjust the balancing factors as the following.

$$\begin{array}{l} \delta'(x) = \delta(x) \\ \delta'(y) = 0 \\ \delta'(z) = 0 \end{array} \tag{4.17}$$

Right-right lean case

Since right-right case is symmetric to left-left case, we can easily achieve the result balancing factors as

$$\begin{array}{l} \delta'(x) = 0 \\ \delta'(y) = 0 \\ \delta'(z) = \delta(z) \end{array} \tag{4.18}$$

Right-left lean case

First let's consider $\delta'(x)$. After balance fixing, we have.

$$\delta'(x) = |B| - |A| \tag{4.19}$$

Before fixing, if we calculate the height of z, we can get.

$$\begin{array}{rll} |z| & = 1 + max(|y|, |D|) & \{\delta(z) = -1 \Rightarrow |y| > |D|\} \\ & = 1 + |y| & \\ & = 2 + max(|B|, |C|) & \end{array} \tag{4.20}$$

While since $\delta(x) = 2$, we can deduce that.

$$\begin{array}{lll} \delta(x) = 2 & \Rightarrow |z| - |A| = 2 & \{\text{By (4.20)}\} \\ & \Rightarrow 2 + max(|B|, |C|) - |A| = 2 & \\ & \Rightarrow max(|B|, |C|) - |A| = 0 & \end{array} \tag{4.21}$$

If $\delta(y) = 1$, which means $|C| - |B| = 1$, it means

$$max(|B|, |C|) = |C| = |B| + 1 \tag{4.22}$$

Take this into (4.21) yields

$$\begin{array}{l} |B| + 1 - |A| = 0 \Rightarrow |B| - |A| = -1 \quad \{\text{By (4.19) }\} \\ \Rightarrow \delta'(x) = -1 \end{array} \tag{4.23}$$

If $\delta(y) \neq 1$, it means $max(|B|, |C|) = |B|$, taking this into (4.21), yields.

$$\begin{array}{l} |B| - |A| = 0 \quad \{\text{By (4.19)}\} \\ \Rightarrow \delta'(x) = 0 \end{array} \tag{4.24}$$

Summarize these 2 cases, we get relationship of $\delta'(x)$ and $\delta(y)$ as the following.

$$\delta'(x) = \left\{ \begin{array}{rcl} -1 & : & \delta(y) = 1 \\ 0 & : & otherwise \end{array} \right. \tag{4.25}$$

For $\delta'(z)$ according to definition, it is equal to.

$$\begin{array}{lll} \delta'(z) & = |D| - |C| & \{\delta(z) = -1 = |D| - |y|\} \\ & = |y| - |C| - 1 & \{|y| = 1 + max(|B|, |C|)\} \\ & = max(|B|, |C|) - |C| & \end{array} \tag{4.26}$$

If $\delta(y) = -1$, then we have $|C| - |B| = -1$, so $max(|B|, |C|) = |B| = |C| + 1$. Takes this into (4.26), we get $\delta'(z) = 1$.

If $\delta(y) \neq -1$, then $max(|B|, |C|) = |C|$, we get $\delta'(z) = 0$.

Combined these two cases, the relationship between $\delta'(z)$ and $\delta(y)$ is as below.

$$\delta'(z) = \left\{ \begin{array}{rcl} 1 & : & \delta(y) = -1 \\ 0 & : & otherwise \end{array} \right. \tag{4.27}$$

Finally, for $\delta'(y)$, we deduce it like below.

$$\begin{array}{ll} \delta'(y) & = |z| - |x| \\ & = max(|C|, |D|) - max(|A|, |B|) \end{array} \tag{4.28}$$

There are three cases.

- If $\delta(y) = 0$, it means $|B| = |C|$, and according to (4.25) and (4.27), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$, and $\delta'(z) = 0 \Rightarrow |C| = |D|$, these lead to $\delta'(y) = 0$.

- If $\delta(y) = 1$, From (4.27), we have $\delta'(z) = 0 \Rightarrow |C| = |D|$.

$$\begin{array}{rll} \delta'(y) & = max(|C|, |D|) - max(|A|, |B|) & \{|C| = |D|\} \\ & = |C| - max(|A|, |B|) & \{\text{From (4.25): } \delta'(x) = -1 \Rightarrow |B| - |A| = -1\} \\ & = |C| - (|B| + 1) & \{\delta(y) = 1 \Rightarrow |C| - |B| = 1\} \\ & = 0 & \end{array}$$

- If $\delta(y) = -1$, From (4.25), we have $\delta'(x) = 0 \Rightarrow |A| = |B|$.

$$\begin{array}{rll} \delta'(y) & = max(|C|, |D|) - max(|A|, |B|) & \{|A| = |B|\} \\ & = max(|C|, |D|) - |B| & \{\text{From (4.27): } |D| - |C| = 1\} \\ & = |C| + 1 - |B| & \{\delta(y) = -1 \Rightarrow |C| - |B| = -1\} \\ & = 0 & \end{array}$$

Both three cases lead to the same result that $\delta'(y) = 0$.

Collect all the above results, we get the new balancing factors after fixing as the following.

$$\begin{array}{l} \delta'(x) = \left\{ \begin{array}{rcl} -1 & : & \delta(y) = 1 \\ 0 & : & otherwise \end{array} \right. \\ \delta'(y) = 0 \\ \delta'(z) = \left\{ \begin{array}{rcl} 1 & : & \delta(y) = -1 \\ 0 & : & otherwise \end{array} \right. \end{array} \tag{4.29}$$

Left-right lean case

Left-right lean case is symmetric to the Right-left lean case. By using the similar deduction, we can find the new balancing factors are identical to the result in (4.29).

4.3.2 Pattern Matching

All the problems have been solved and it's time to define the final pattern matching fixing function.

$$balance(T, \Delta H) = \left\{ \begin{array}{rcl} (node(node(A, x, B, \delta(x)), y, node(C, z, D, 0), 0), 0) & : & P_{ll}(T) \\ (node(node(A, x, B, 0), y, node(C, z, D, \delta(z)), 0), 0) & : & P_{rr}(T) \\ (node(node(A, x, B, \delta'(x)), y, node(C, z, D, \delta'(z)), 0), 0) & : & P_{rl}(T) \lor P_{lr}(T) \\ (T, \Delta H) & : & otherwise \end{array} \right. \tag{4.30}$$

Where $P_{ll}(T)$ means the pattern of tree T is left-left lean respectively. $\delta'(x)$ and $delta'(z)$ are defined in (4.29). The four patterns are tested as below.

$$\begin{array}{l} P_{ll}(T) = node(node(node(A, x, B, \delta(x)), y, C, -1), z, D, -2) \\ P_{rr}(T) = node(A, x, node(B, y, node(C, z, D, \delta(z)), 1), 2) \\ P_{rl}(T) = node(node(A, x, node(B, y, C, \delta(y)), 1), z, D, -2) \\ P_{lr}(T) = node(A, x, node(node(B, y, C, \delta(y)), z, D, -1), 2) \end{array} \tag{4.31}$$

Translating the above function definition to Haskell yields a simple and intuitive program.

```
balance :: (AVLTree a, Int) → (AVLTree a, Int)
balance (Br (Br (Br a x b dx) y c (-1)) z d (-2), _) =
        (Br (Br a x b dx) y (Br c z d 0) 0, 0)
balance (Br a x (Br b y (Br c z d dz)    1)    2, _) =
        (Br (Br a x b 0) y (Br c z d dz) 0, 0)
balance (Br (Br a x (Br b y c dy)    1) z d (-2), _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (Br a x (Br (Br b y c dy) z d (-1))    2, _) =
        (Br (Br a x b dx') y (Br c z d dz') 0, 0) where
    dx' = if dy ==  1 then -1 else 0
    dz' = if dy == -1 then  1 else 0
balance (t, d) = (t, d)
```

The insertion algorithm takes time proportion to the height of the tree, and according to the result we proved above, its performance is $O(\lg N)$ where N is the number of elements stored in the AVL tree.

Verification

One can easily create a function to verify a tree is AVL tree. Actually we need verify two things, first, it's a binary search tree; second, it satisfies AVL tree property.

We left the first verification problem as an exercise to the reader.

In order to test if a binary tree satisfies AVL tree property, we can test the difference in height between its two children, and recursively test that both children conform to AVL property until we arrive at an empty leaf.

$$avl?(T) = \begin{cases} True & : T = \Phi \\ avl?(L) \land avl?(R) \land ||R| - |L|| \leq 1 & : otherwise \end{cases} \tag{4.32}$$

And the height of a AVL tree can also be calculate from the definition.

$$|T| = \begin{cases} 0 & : T = \Phi \\ 1 + max(|R|, |L|) & : otherwise \end{cases} \tag{4.33}$$

The corresponding Haskell program is given as the following.

```
isAVL :: (AVLTree a) → Bool
isAVL Empty = True
isAVL (Br l _ r d) = and [isAVL l, isAVL r, abs (height r - height l) ≤ 1]

height :: (AVLTree a) → Int
height Empty = 0
height (Br l _ r _) = 1 + max (height l) (height r)
```

Exercise 4.1

Write a program to verify a binary tree is a binary search tree in your favorite programming language. If you choose to use an imperative language, please consider realize this program without recursion.

4.4 Deletion

As we mentioned before, deletion doesn't make significant sense in purely functional settings. As the tree is read only, it's typically performs frequently looking up after build.

Even if we implement deletion, it's actually re-building the tree as we presented in chapter of red-black tree. We left the deletion of AVL tree as an exercise to the reader.

Exercise 4.2

- Take red-black tree deletion algorithm as an example, write the AVL tree deletion program in purely functional approach in your favorite programming language.
- Write the deletion algorithm in imperative approach in your favorite programming language.

4.5 Imperative AVL tree algorithm $\star$

We almost finished all the content in this chapter about AVL tree. However, it necessary to show the traditional insert-and-rotate approach as the comparator to pattern matching algorithm.

Similar as the imperative red-black tree algorithm, the strategy is first to do the insertion as same as for binary search tree, then fix the balance problem by rotation and return the final result.

```
1: function INSERT($T, k$)
2:     $root \leftarrow T$
3:     $x \leftarrow$ CREATE-LEAF($k$)
4:     $\delta(x) \leftarrow 0$
5:     $parent \leftarrow NIL$
6:     while $T \neq NIL$ do
7:         $parent \leftarrow T$
8:         if $k <$ KEY($T$) then
9:             $T \leftarrow$ LEFT($T$)
10:        else
11:            $T \leftarrow$ RIGHT($T$)
12:    PARENT($x$) $\leftarrow parent$
13:    if $parent = NIL$ then                                ▷ tree $T$ is empty
14:        return $x$
15:    else if $k <$ KEY($parent$) then
16:        LEFT($parent$) $\leftarrow x$
17:    else
18:        RIGHT($parent$) $\leftarrow x$
19:    return AVL-INSERT-FIX($root, x$)
```

Note that after insertion, the height of the tree may increase, so that the balancing factor δ may also change, insert on right side will increase δ by 1, while insert on left side will decrease it. By the end of this algorithm, we need perform bottom-up fixing from node x towards root.

We can translate the pseudo code to real programming language, such as Python [2].

```
def avl_insert(t, key):
    root = t
    x = Node(key)
    parent = None
    while(t):
        parent = t
        if(key < t.key):
            t = t.left
        else:
            t = t.right
    if parent is None: #tree is empty
        root = x
    elif key < parent.key:
        parent.set_left(x)
    else:
        parent.set_right(x)
    return avl_insert_fix(root, x)
```

This is a top-down algorithm search the tree from root down to the proper position and insert the new key as a leaf. By the end of this algorithm, it calls fixing procedure, by passing the root and the new node inserted.

Note that we reuse the same methods of set_left() and set_right() as we defined in chapter of red-black tree.

In order to resume the AVL tree balance property by fixing, we first determine if the new node is inserted on left hand or right hand. If it is on left, the balancing factor δ decreases, otherwise it increases. If we denote the new value as δ', there are 3 cases of the relationship between δ and δ'.

- If $|\delta| = 1$ and $|\delta'| = 0$, this means adding the new node makes the tree perfectly balanced, the height of the parent node doesn't change, the algorithm can be terminated.
- If $|\delta| = 0$ and $|\delta'| = 1$, it means that either the height of left sub tree or right sub tree increases, we need go on check the upper level of the tree.
- If $|\delta| = 1$ and $|\delta'| = 2$, it means the AVL tree property is violated due to the new insertion. We need perform rotation to fix it.

```
1: function AVL-INSERT-FIX($T, x$)
2:     while PARENT($x$) $\neq NIL$ do
3:         $\delta \leftarrow \delta$(PARENT($x$))
4:         if $x =$ LEFT(PARENT($x$)) then
5:             $\delta' \leftarrow \delta - 1$
6:         else
7:             $\delta' \leftarrow \delta + 1$
8:         $\delta$(PARENT($x$)) $\leftarrow \delta'$
9:         $P \leftarrow$ PARENT($x$)
10:        $L \leftarrow$ LEFT($x$)
11:        $R \leftarrow$ RIGHT($x$)
```

[2] C and C++ source code are available along with this book

```
12:         if $|\delta| = 1$ and $|\delta'| = 0$ then      ▷ Height doesn't change, terminates.
13:             return $T$
14:         else if $|\delta| = 0$ and $|\delta'| = 1$ then      ▷ Go on bottom-up updating.
15:             $x \leftarrow P$
16:         else if $|\delta| = 1$ and $|\delta'| = 2$ then
17:             if $\delta' = 2$ then
18:                 if $\delta(R) = 1$ then      ▷ Right-right case
19:                     $\delta(P) \leftarrow 0$      ▷ By (4.18)
20:                     $\delta(R) \leftarrow 0$
21:                     $T \leftarrow$ LEFT-ROTATE$(T, P)$
22:                 if $\delta(R) = -1$ then      ▷ Right-left case
23:                     $\delta_y \leftarrow \delta($LEFT$(R))$      ▷ By (4.29)
24:                     if $\delta_y = 1$ then
25:                         $\delta(P) \leftarrow -1$
26:                     else
27:                         $\delta(P) \leftarrow 0$
28:                     $\delta($LEFT$(R)) \leftarrow 0$
29:                     if $\delta_y = -1$ then
30:                         $\delta(R) \leftarrow 1$
31:                     else
32:                         $\delta(R) \leftarrow 0$
33:                     $T \leftarrow$ RIGHT-ROTATE$(T, R)$
34:                     $T \leftarrow$ LEFT-ROTATE$(T, P)$
35:             if $\delta' = -2$ then
36:                 if $\delta(L) = -1$ then      ▷ Left-left case
37:                     $\delta(P) \leftarrow 0$
38:                     $\delta(L) \leftarrow 0$
39:                     RIGHT-ROTATE$(T, P)$
40:                 else      ▷ Left-Right case
41:                     $\delta_y \leftarrow \delta($RIGHT$(L))$
42:                     if $\delta_y = 1$ then
43:                         $\delta(L) \leftarrow -1$
44:                     else
45:                         $\delta(L) \leftarrow 0$
46:                     $\delta($RIGHT$(L)) \leftarrow 0$
47:                     if $\delta_y = -1$ then
48:                         $\delta(P) \leftarrow 1$
49:                     else
50:                         $\delta(P) \leftarrow 0$
51:                     LEFT-ROTATE$(T, L)$
52:                     RIGHT-ROTATE$(T, P)$
53:             break
54:     return $T$
```

Here we reuse the rotation algorithms mentioned in red-black tree chapter. Rotation operation doesn't update balancing factor δ at all, However, since rotation changes (actually improves) the balance situation we should update these factors. Here we refer the results from above section. Among the four cases, right-right case and left-left case only need one rotation, while right-left case and left-right case need two rotations.

The relative python program is shown as the following.

```
def avl_insert_fix(t, x):
    while x.parent is not None:
        d2 = d1 = x.parent.delta
        if x == x.parent.left:
            d2 = d2 - 1
        else:
            d2 = d2 + 1
        x.parent.delta = d2
        (p, l, r) = (x.parent, x.parent.left, x.parent.right)
        if abs(d1) == 1 and abs(d2) == 0:
            return t
        elif abs(d1) == 0 and abs(d2) == 1:
            x = x.parent
        elif abs(d1)==1 and abs(d2) == 2:
            if d2 == 2:
                if r.delta == 1:  # Right-right case
                    p.delta = 0
                    r.delta = 0
                    t = left_rotate(t, p)
                if r.delta == -1: # Right-Left case
                    dy = r.left.delta
                    if dy == 1:
                        p.delta = -1
                    else:
                        p.delta = 0
                    r.left.delta = 0
                    if dy == -1:
                        r.delta = 1
                    else:
                        r.delta = 0
                    t = right_rotate(t, r)
                    t = left_rotate(t, p)
            if d2 == -2:
                if l.delta == -1: # Left-left case
                    p.delta = 0
                    l.delta = 0
                    t = right_rotate(t, p)
                if l.delta == 1: # Left-right case
                    dy = l.right.delta
                    if dy == 1:
                        l.delta = -1
                    else:
                        l.delta = 0
                    l.right.delta = 0
                    if dy == -1:
                        p.delta = 1
                    else:
                        p.delta = 0
                    t = left_rotate(t, l)
                    t = right_rotate(t, p)
            break
    return t
```

We skip the AVL tree deletion algorithm and left this as an exercise to the reader.

4.6 Chapter note

AVL tree was invented in 1962 by Adelson-Velskii and Landis[3], [4]. The name AVL tree comes from the two inventor's name. It's earlier than red-black tree.

It's very common to compare AVL tree and red-black tree, both are self-balancing binary search trees, and for all the major operations, they both consume $O(\lg N)$ time. From the result of (4.7), AVL tree is more rigidly balanced hence they are faster than red-black tree in looking up intensive applications [3]. However, red-black trees could perform better in frequently insertion and removal cases.

Many popular self-balancing binary search tree libraries are implemented on top of red-black tree such as STL etc. However, AVL tree provides an intuitive and effective solution to the balance problem as well.

After this chapter, we'll extend the tree data structure from storing data in node to storing information on edges, which leads to Trie and Patrica, etc. If we extend the number of children from two to more, we can get B-tree. These data structures will be introduced next.

Bibliography

[1] Data.Tree.AVL http://hackage.haskell.org/packages/archive/AvlTree/4.2/doc/html/Data-Tree-AVL.html

[2] Chris Okasaki. “FUNCTIONAL PEARLS Red-Black Trees in a Functional Setting”. J. Functional Programming. 1998

[3] Wikipedia. “AVL tree”. http://en.wikipedia.org/wiki/AVL_tree

[4] Guy Cousinear, Michel Mauny. “The Functional Approach to Programming”. Cambridge University Press; English Ed edition (October 29, 1998). ISBN-13: 978-0521576819

[5] Pavel Grafov. “Implementation of an AVL tree in Python”. http://github.com/pgrafov/python-avl-tree

Chapter 5

Trie and Patricia

5.1 Introduction

The binary trees introduced so far store information in nodes. It's possible to store the information in edges. Trie and Patricia are important data structures in information retrieving and manipulating. They were invented in 1960s. And are widely used in compiler design[2], and bio-information area, such as DNA pattern matching [3].

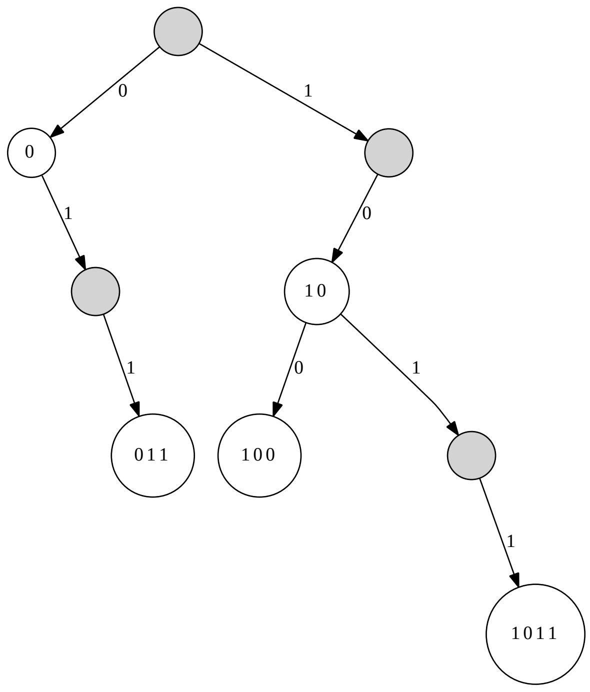


Figure 5.1: Radix tree.

Figure 5.1 shows a radix tree[2]. It contains the strings of bit 1011, 10, 011, 100 and 0. When searching a key $k = (b_0 b_1 ... b_n)_2$, we take the first bit b_0 (MSB from left), check if it is 0 or 1, if it is 0, we turn left; and turn right for 1. Then we take the second bit and repeat this search till either meet a leaf or finish all n bits.

The radix tree needn't store keys in node at all. The information is represented by edges. The nodes marked with keys in the above figure are only for illustration purpose.

It is very natural to come to the idea 'is it possible to represent key in integer instead of string?' Because integer can be written in binary format, such approach can save spaces. Another advantage is that the speed is fast because we can use bit-wise manipulation in most programming environment.

5.2 Integer Trie

The data structure shown in figure 5.1 is called as *binary trie*. Trie is invented by Edward Fredkin. It comes from "retrieval", pronounce as /'tri:/ by the inventor, while it is pronounced /'trai/ "try" by other authors [5]. Trie is also called prefix tree or radix tree. A binary trie is a special binary tree in which the placement of each key is controlled by its bits, each 0 means "go left" and each 1 means "go right"[2].

Because integers can be represented in binary format, it is possible to store integer keys rather than 0, 1 strings. When insert an integer as the new key to the trie, we change it to binary form, then examine the first bit, if it is 0, we recursively insert the rest bits to the left sub tree; otherwise if it is 1, we insert into the right sub tree.

There is a problem when treat the key as integer. Consider a binary trie shown in figure 5.2. If represented in 0, 1 strings, all the three keys are different. But they are identical when turn into integers. Where should we insert decimal 3, for example, to the trie?

One approach is to treat all the prefix zero as effective bits. Suppose the integer is represented with 32-bits, If we want to insert key 1, it ends up with a tree of 32 levels. There are 31 nodes, each only has the left sub tree. the last node only has the right sub tree. It is very inefficient in terms of space.

Okasaki shows a method to solve this problem in [2]. Instead of using big-endian integer, we can use the little-endian integer to represent key. Thus decimal integer 1 is represented as binary 1. Insert it to the empty binary trie, the result is a trie with a root and a right leaf. There is only 1 level. decimal 2 is represented as 01, and decimal 3 is $(11)_2$ in little-endian binary format. There is no need to add any prefix 0, the position in the trie is uniquely determined.

5.2.1 Definition of integer Trie

In order to define the little-endian binary trie, we can reuse the structure of binary tree. A binary trie node is either empty, or a branch node. The branch node contains a left child, a right node, and optional value as satellite data. The left sub tree is encoded as 0 and the right sub tree is encoded as 1.

The following example Haskell code defines the trie algebraic data type.

```
data IntTrie a = Empty
               | Branch (IntTrie a) (Maybe a) (IntTrie a)
```

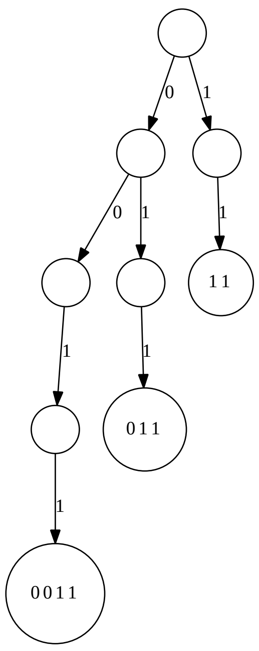


Figure 5.2: a big-endian trie

Below is another example definition in Python.

```
class IntTrie:
    def __init__(self):
        self.left = self.right = None
        self.value = None
```

5.2.2 Insertion

Since the key is little-endian integer, when insert a key, we take the bit one by one from the right most (LSB). If it is 0, we go to the left, otherwise go to the right for 1. If the child is empty, we need create a new node, and repeat this to the last bit (MSB) of the key.

```
function INSERT($T, k, v$)
    if $T$ = NIL then
        $T \leftarrow$ EMPTY-NODE
    $p \leftarrow T$
    while $k \neq 0$ do
        if EVEN?($k$) then
            if LEFT($p$) = NIL then
                LEFT($p$) $\leftarrow$ EMPTY-NODE
            $p \leftarrow$ LEFT($p$)
        else
            if RIGHT($p$) = NIL then
                RIGHT($p$) $\leftarrow$ EMPTY-NODE
            $p \leftarrow$ RIGHT($p$)
```

```
14:         $k \leftarrow \lfloor k/2 \rfloor$
15:     DATA$(p) \leftarrow v$
16:     **return** $T$
```

This algorithm takes 3 arguments, a Trie T, a key k, and the satellite date v. The following Python example code implements the insertion algorithm. The satellite data is optional, it is empty by default.

```
def trie_insert(t, key, value = None):
    if t is None:
        t = IntTrie()
    p = t
    while key != 0:
        if key & 1 == 0:
            if p.left is None:
                p.left = IntTrie()
            p = p.left
        else:
            if p.right is None:
                p.right = IntTrie()
            p = p.right
        key = key>>1
    p.value = value
    return t
```

Figure 5.2 shows a trie which is created by inserting pairs of key and value $\{1 \rightarrow a, 4 \rightarrow b, 5 \rightarrow c, 9 \rightarrow d\}$ to the empty trie.

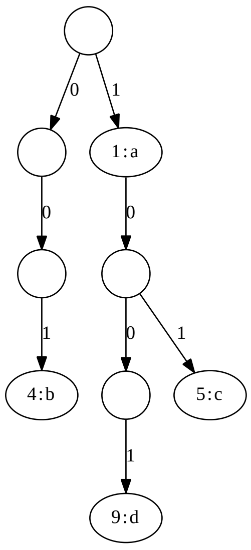


Figure 5.3: An little-endian integer binary trie for the map $\{1 \rightarrow a, 4 \rightarrow b, 5 \rightarrow c, 9 \rightarrow d\}$.

Because the definition of the integer trie is recursive, it's nature to define the insertion algorithm recursively. If the LSB is 0, it means that the key to be inserted is even, we recursively insert to the left child, we can divide the key by 2 to get rid of the LSB. If the LSB is 1, the key is odd number, the recursive insertion is applied to the right child. For trie T, denote the left and right children as T_l and T_r respectively. Thus $T = (T_l, d, T_r)$, where d is the optional satellite data. if T is empty, T_l, T_r and d are defined as empty as well.

$$insert(T, k, v) = \begin{cases} (T_l, v, T_r) & : k = 0 \\ (insert(T_l, k/2, v), d, T_r) & : even(k) \\ (T_l, d, insert(T_r, \lfloor k/2 \rfloor, v)) & : otherwise \end{cases} \tag{5.1}$$

If the key to be inserted already exists, this algorithm just overwrites the previous stored data. It can be replaced with other alternatives, such as storing data as with linked-list etc.

The following Haskell example program implements the insertion algorithm.

```
insert t 0 x = Branch (left t) (Just x) (right t)
insert t k x | even k = Branch (insert (left t) (k `div` 2) x) (value t) (right t)
             | otherwise = Branch (left t) (value t) (insert (right t) (k `div` 2) x)

left (Branch l _ _) = l
left Empty = Empty

right (Branch _ _ r) = r
right Empty = Empty

value (Branch _ v _) = v
value Empty = Nothing
```

For a given integer k with m bits in binary, the insertion algorithm reuses m levels. The performance is bound to $O(m)$ time.

5.2.3 Look up

To look up key k in the little-endian integer binary trie. We take each bit of k from the left (LSB), then go left if this bit is 0, otherwise, we go right. The looking up completes when all bits are consumed.

```
1: function LOOKUP($T, k$)
2:     while $x \neq 0 \land T \neq$NIL do
3:         if EVEN?($x$) then
4:             $T \leftarrow$ LEFT($T$)
5:         else
6:             $T \leftarrow$ RIGHT($T$)
7:         $k \leftarrow \lfloor k/2 \rfloor$
8:     if $T \neq$ NIL then
9:         return DATA($T$)
10:    else
11:        return not found
```

Below Python example code uses bit-wise operation to implements the looking up algorithm.

```
def lookup(t, key):
    while key != 0 and (t is not None):
        if key & 1 == 0:
            t = t.left
        else:
            t = t.right
        key = key>>1
    if t is not None:
        return t.value
    else:
        return None
```

Looking up can also be define in recursive manner. If the tree is empty, the looking up fails; If $k = 0$, the satellite data is the result to be found; If the last bit is 0, we recursively look up the left child; otherwise, we look up the right child.

$$lookup(T, k) = \begin{cases} \Phi & : T = \Phi \\ d & : k = 0 \\ lookup(T_l, k/2) & : even(k) \\ lookup(T_r, \lfloor k/2 \rfloor) & : otherwise \end{cases} \tag{5.2}$$

The following Haskell example program implements the recursive look up algorithm.

```
search Empty k = Nothing
search t 0 = value t
search t k = if even k then search (left t) (k `div` 2)
             else search (right t) (k `div` 2)
```

The looking up algorithm is bound to $O(m)$ time, where m is the number of bits for a given key.

5.3 Integer Patricia

Trie has some drawbacks. It wastes a lot of spaces. Note in figure 5.2, only leafs store the real data. Typically, the integer binary trie contains many nodes only have one child. One improvement idea is to compress the chained nodes together. Patricia is such a data structure invented by Donald R. Morrison in 1968. Patricia means practical algorithm to retrieve information coded in alphanumeric[3]. It is another kind of prefix tree.

Okasaki gives implementation of integer Patricia in [2]. If we merge the chained nodes which have only one child together in figure 5.3, We can get a Patricia as shown in figure 5.4.

From this figure, we can find that the key for the sibling nodes is the longest common prefix for them. They branches out at certain bit. Patricia saves a lot of spaces compare to trie.

Different from integer trie, using the big-endian integer in Patricia doesn't cause the padding zero problem mentioned in section 5.2. All zero bits before MSB are omitted to save space. Okasaki lists some significant advantages of big-endian Patricia[2].

5.3.1 Definition

Integer Patricia tree is a special kind of binary tree. It is either empty or is a node. There are two different types of node.

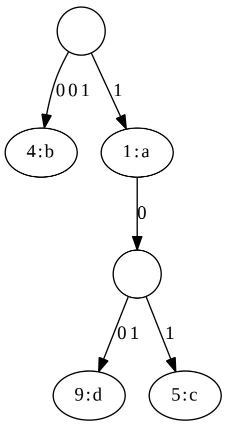


Figure 5.4: Little endian Patricia for the map $\{1 \rightarrow a, 4 \rightarrow b, 5 \rightarrow c, 9 \rightarrow d\}$.

- It can be a leaf contains integer key and optional satellite data;
- or a branch node, contains the left and the right children. The two children shares the longest common prefix bits for their keys. For the left child, the next bit of the key is zero, while the next bit is one for the right child.

The following Haskell example code defines Patricia accordingly.

```
type Key = Int
type Prefix = Int
type Mask = Int

data IntTree a = Empty
              | Leaf Key a
              | Branch Prefix Mask (IntTree a) (IntTree a)
```

In order to tell from which bit the left and right children differ, a mask is recorded in the branch node. Typically, a mask is power of 2, like 2^n for some non-negative integer n, all bits being lower than n don't belong to the common prefix.

The following Python example code defines Patricia as well as some auxiliary functions.

```
class IntTree:
    def __init__(self, key = None, value = None):
        self.key = key
        self.value = value
        self.prefix = self.mask = None
        self.left = self.right = None

    def set_children(self, l, r):
        self.left = l
        self.right = r

    def replace_child(self, x, y):
        if self.left == x:
            self.left = y
        else:
            self.right = y

    def is_leaf(self):
        return self.left is None and self.right is None

    def get_prefix(self):
        if self.prefix is None:
            return self.key
        else:
            return self.prefix
```

5.3.2 Insertion

When insert a key, if the tree is empty, we can just create a leaf node with the given key and satellite data, as shown in figure 5.5).

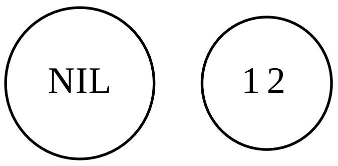


Figure 5.5: Left: the empty tree; Right: After insert key 12.

If the tree is just a singleton leaf node x, we can create a new leaf y, put the key and data into it. After that, we need create a new branch node, and set x and y as the two children. In order to determine if the y should be the left or right node, we need find the longest common prefix of x and y. For example if $key(x)$ is 12 ($(1100)_2$ in binary), $key(y)$ is 15 ($(1111)_2$ in binary), then the longest common prefix is $(11oo)_2$. Where o denotes the bits we don't care about. We can use another integer to mask those bits. In this case, the mask number is 4 (100 in binary). The next bit after the longest common prefix presents 2^1. This bit is 0 in $key(x)$, while it is 1 in $key(y)$. We should set x as the left child and y as the right child. Figure 5.6 shows this example.

In case the tree is neither empty, nor a singleton leaf, we need firstly check if the key to be inserted matches the longest common prefix recorded in the root. Then recursively insert the key to the left or the right child according to the next bit of the common prefix. For example, if insert key 14 ($(1110)_2$ in binary) to the result tree in figure 5.6, since the common prefix is $(11oo)_2$, and the next bit (the bit of 2^1) is 1, we need recursively insert to the right child.

If the key to be inserted doesn't match the longest common prefix stored in the root, we need branch a new leaf out. Figure 5.7 shows these two different cases.

For a given key k and value v, denote (k, v) is the leaf node. For branch node, denote it in form of (p, m, T_l, T_r), where p is the longest common prefix, m is the mask, T_l and T_r are the left and right children. Summarize the above

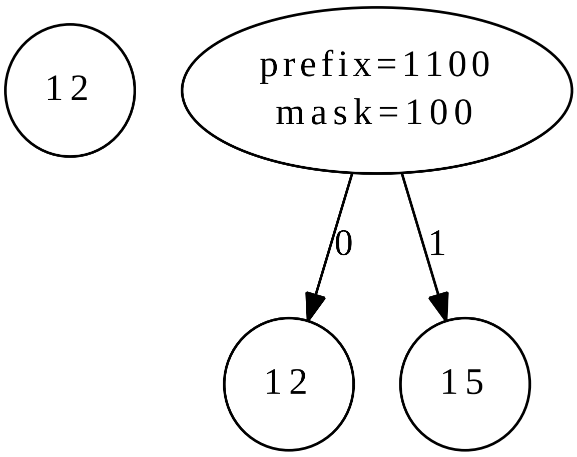
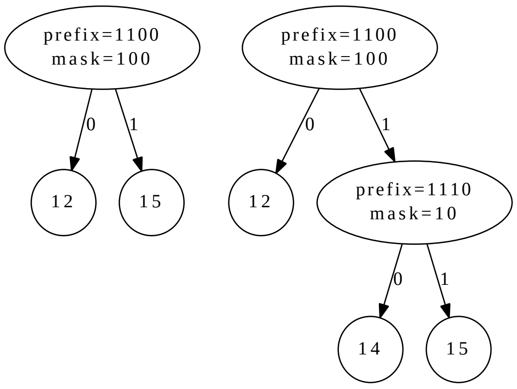
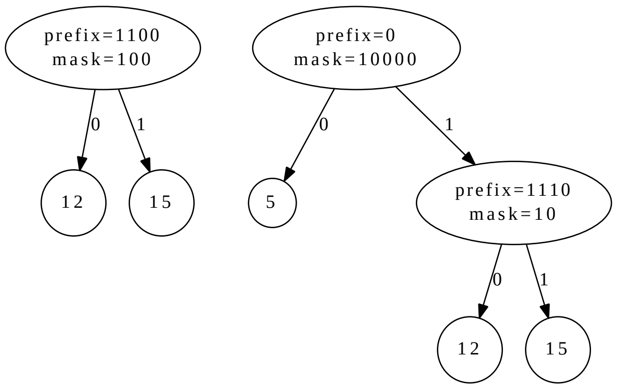


Figure 5.6: Left: A tree with singleton leaf 12; Right: After insert key 15.



(a) Insert key 14. It matches the longest common prefix $(1100)_2$; 14 is then recursively inserted to the right sub tree.



(b) Insert key 5. It doesn't match the longest common prefix $(1100)_2$, a new leaf is branched out.

Figure 5.7: Insert key to branch node.

cases, the insertion algorithm can be defined as the following.

$$insert(T,k,v) = \begin{cases} (k,v) & : T = \Phi \lor T = (k,v') \\ join(k,(k,v),k',T) & : T = (k',v') \\ (p,m,insert(T_l,k,v),T_r) & : T = (p,m,T_l,T_r), match(k,p,m), zero(k,m) \\ (p,m,T_l,insert(T_r,k,v)) & : T = (p,m,T_l,T_r), match(k,p,m), \neg zero(k,m) \\ join(k,(k,v),p,T) & : T = (p,m,T_l,T_r), \neg match(k,p,m) \end{cases} \tag{5.3}$$

The first clause deals with the edge cases, that either T is empty or it is a leaf node with the same key. The algorithm overwrites the previous value for the later case.

The second clause handles the case that T is a leaf node, but with different key. Here we need branch out another new leaf. We need extract the longest common prefix, and determine which leaf should be set as the left, and which should be set as the right child. Function $join(k_1, T_1, K_2, T_2)$ does this work. We'll define it later.

The third clause deals with the case that T is a branch node, the longest common prefix matches the key to be inserted, and the next bit to the common prefix is zero. Here we need recursively insert to the left child.

The fourth clause handles the similar case as the third clause, except that the next bit to the common prefix is one, but not zero. We need recursively insert to the right child.

The last clause is for the case that the key to be inserted doesn't match the longest common prefix stored in the branch. We need branch out a new leaf by calling the $join$ function.

We need define function $match(k,p,m)$ to test if the key k, has the same prefix p above the masked bits m. For example, suppose the prefix stored in a branch node is $(p_np_{n-1}...p_i...p_0)_2$ in binary, key k is $(k_nk_{n-1}...k_i...k_0)_2$ in binary, and the mask is $(100...0)_2 = 2^i$. They match if and only if $p_j = k_j$ for all $i \leq j \leq n$.

One solution to realize match is to test if $mask(k,m) = p$ is satisfied. Where $mask(x,m) = \overline{m-1}\&x$, that we perform bitwise-not of $m-1$, then perform bitwise-and with x.

Function $zero(k,m)$ test the next bit of the common prefix is zero. With the help of the mask m, we can shift m one bit to the right, then perform bitwise and with the key.

$$zero(k,m) = x\&shift_r(m,1) \tag{5.4}$$

If the mask $m = (100..0)_2 = 2^i$, $k = (k_nk_{n-1}...k_i1...k_0)_2$, because the bit next to k_i is 1, $zero(k,m)$ returns false value; if $k = (k_nk_{n-1}...k_i0...k_0)_2$, then the result is true.

Function $join(p_1,T_1,p_2,T_2)$ takes two different prefixes and trees. It extracts the longest common prefix of p_1 and p_2, create a new branch node, and set T_1 and T_2 as the two children.

$$join(p_1,T_1,p_2,T_2) = \begin{cases} (p,m,T_1,T_2) & : zero(p1,m), (p,m) = LCP(p_1,p_2) \\ (p,m,T_2,T_1) & : \neg zero(p1,m) \end{cases} \tag{5.5}$$

In order to calculate the longest common prefix of p_1 and p_2, we can firstly compute bitwise exclusive-or for them, then count the number of bits in this result, and generate a mask $m = 2^{|xor(p_1,p_2)|}$. The longest common prefix p can be given by masking the bits with m for either p_1 or p_2.

$$p = mask(p_1, m) \tag{5.6}$$

The following Haskell example code implements the insertion algorithm.

```
import Data.Bits

insert t k x
   = case t of
       Empty → Leaf k x
       Leaf k' x' → if k==k' then Leaf k x
                     else join k (Leaf k x) k' t -- t@(Leaf k' x')
       Branch p m l r
          | match k p m → if zero k m
                           then Branch p m (insert l k x) r
                           else Branch p m l (insert r k x)
          | otherwise → join k (Leaf k x) p t -- t@(Branch p m l r)

join p1 t1 p2 t2 = if zero p1 m then Branch p m t1 t2
                                else Branch p m t2 t1
    where
      (p, m) = lcp p1 p2

lcp :: Prefix → Prefix → (Prefix, Mask)
lcp p1 p2 = (p, m) where
    m = bit (highestBit (p1 ‘xor‘ p2))
    p = mask p1 m

highestBit x = if x == 0 then 0 else 1 + highestBit (shiftR x 1)

mask x m = (x ∘ &. complement (m-1)) -- complement means bit-wise not.

zero x m = x ∘ &. (shiftR m 1) == 0

match k p m = (mask k m) == p
```

The insertion algorithm can also be realized imperatively.

```
function INSERT($T, k, v$)
    if $T$ = NIL then
        $T \leftarrow$ CREATE-LEAF($k, v$)
        return $T$
    $y \leftarrow T$
    $p \leftarrow$ NIL
    while $y$ is not leaf, and MATCH($k$, PREFIX($y$), MASK($y$)) do
        $p \leftarrow y$
        if ZERO?($k$, MASK($y$)) then
            $y \leftarrow$ LEFT($y$)
        else
            $y \leftarrow$ RIGHT($y$)
```

```
13:     if $y$ is leaf, and $k = \textsc{Key}(y)$ then
14:         $\textsc{Data}(y) \leftarrow v$
15:     else
16:         $z \leftarrow \textsc{Branch}(y, \textsc{Create-Leaf}(k, v))$
17:         if $p =$ NIL then
18:             $T \leftarrow z$
19:         else
20:             if $\textsc{Left}(p) = y$ then
21:                 $\textsc{Left}(p) \leftarrow z$
22:             else
23:                 $\textsc{Right}(p) \leftarrow z$
24:     return $T$
```

Function $\textsc{Branch}(T_1, T_2)$ does the similar job as what *join* is defined. It creates a new branch node, extracts the longest common prefix, and sets T_1 and T_2 as the two children.

```
1: function $\textsc{Branch}(T_1, T_2)$
2:     $T \leftarrow \textsc{Empty-Node}$
3:     ( $\textsc{Prefix}(T)$, $\textsc{Mask}(T)$ ) $\leftarrow$ LCP($\textsc{Prefix}(T_1)$, $\textsc{Prefix}(T_2)$)
4:     if $\textsc{Zero?}(\textsc{Prefix}(T_1), \textsc{Mask}(T))$ then
5:         $\textsc{Left}(T) \leftarrow T_1$
6:         $\textsc{Right}(T) \leftarrow T_2$
7:     else
8:         $\textsc{Left}(T) \leftarrow T_2$
9:         $\textsc{Right}(T) \leftarrow T_1$
10:    return $T$
```

The following Python example program implements the insertion algorithm.

```
def insert(t, key, value = None):
    if t is None:
        t = IntTree(key, value)
        return t

    node = t
    parent = None
    while(True):
        if match(key, node):
            parent = node
            if zero(key, node.mask):
                node = node.left
            else:
                node = node.right
        else:
            if node.is_leaf() and key == node.key:
                node.value = value
            else:
                new_node = branch(node, IntTree(key, value))
                if parent is None:
                    t = new_node
                else:
                    parent.replace_child(node, new_node)
            break
    return t
```

The auxiliary functions, `match`, `branch`, `lcp` etc. are given as below.

```
def maskbit(x, mask):
    return x & (~(mask-1))

def match(key, tree):
    return (not tree.is_leaf()) and maskbit(key, tree.mask) == tree.prefix

def zero(x, mask):
    return x & (mask>>1) == 0

def lcp(p1, p2):
    diff = (p1 ^ p2)
    mask=1
    while(diff!=0):
        diff>>=1
        mask<<=1
    return (maskbit(p1, mask), mask)

def branch(t1, t2):
    t = IntTree()
    (t.prefix, t.mask) = lcp(t1.get_prefix(), t2.get_prefix())
    if zero(t1.get_prefix(), t.mask):
        t.set_children(t1, t2)
    else:
        t.set_children(t2, t1)
    return t
```

Figure 5.8 shows the example Patricia created with the insertion algorithm.

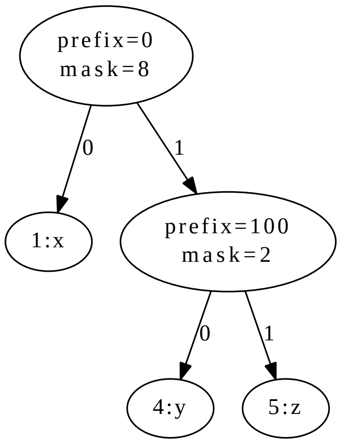


Figure 5.8: Insert map $1 \rightarrow x, 4 \rightarrow y, 5 \rightarrow z$ into the big-endian integer Patricia tree.

5.3.3 Look up

Consider the property of integer Patricia tree. When look up a key, if it has common prefix with the root, then we check the next bit. If this bit is zero, we recursively look up the left child; otherwise if the bit is one, we next look up the right child.

When reach a leaf node, we can directly check if the key of the leaf is equal to what we are looking up. This algorithm can be described with the following pseudo code.

```
function LOOK-UP($T, k$)
    if $T$ = NIL then
        return $NIL$                                          ▷ Not found
    while $T$ is not leaf, and MATCH($k$, PREFIX($T$), MASK($T$)) do
        if ZERO?($k$, MASK($T$)) then
            $T \leftarrow$ LEFT($T$)
        else
            $T \leftarrow$ RIGHT($T$)
    if $T$ is leaf, and KEY($T$) $= k$ then
        return DATA($T$)
    else
        return $NIL$                                          ▷ Not found
```

Below Python example program implements the looking up algorithm.

```python
def lookup(t, key):
    if t is None:
        return None
    while (not t.is_leaf()) and match(key, t):
        if zero(key, t.mask):
            t = t.left
        else:
            t = t.right
    if t.is_leaf() and t.key == key:
        return t.value
    else:
        return None
```

The looking up algorithm can also be realized in recursive approach. If the Patricia tree T is empty, or it's a singleton leaf with different key from what we are looking up, the result is empty to indicate not found error; If the tree is a singleton leaf, and the key of this leaf is equal to what we are looking up, we are done. Otherwise, T is a branch node, we need check if the common prefix matches the key to be looked up, and recursively look up the child according to the next bit. If the common prefix doesn't match the key, it means the key doesn't exist in the tree. We can return empty result to indicate not found error.

$$lookup(T, k) = \begin{cases} \Phi & : T = \Phi \lor (T = (k', v), k' \neq k) \\ v & : T = (k', v), k' = k \\ lookup(T_l, k) & : T = (p, m, T_l, T_r), match(k, p, m), zero(k, m) \\ lookup(T_r, k) & : T = (p, m, T_l, T_r), match(k, p, m), \neg zero(k, m) \\ \Phi & : otherwise \end{cases} \tag{5.7}$$

The following Haskell example program implements this recursive looking up algorithm.

```
search t k
  = case t of
      Empty → Nothing
      Leaf k' x → if k==k' then Just x else Nothing
      Branch p m l r
             | match k p m → if zero k m then search l k
                             else search r k
             | otherwise → Nothing
```

5.4 Alphabetic Trie

Integer based Trie and Patricia Tree can be a good start point. Such technical plays important role in Compiler implementation. Okasaki pointed that the widely used Glasgow Haskell Compiler, GHC, utilized the similar implementation for several years before 1998 [2].

If we extend the key from integer to alphabetic value, Trie and Patricia tree can be very powerful in solving textual manipulation problems.

5.4.1 Definition

It's not enough to just use the left and right children to represent alphabetic keys. Using English for example, there are 26 letters and each can be lower or upper case. If we don't care about the case, one solution is to limit the number of branches (children) to 26. Some simplified ANSI C implementations of Trie are defined by using the array of 26 letters. This can be illustrated as in Figure 5.9.

Not all branch nodes contain data. For instance, in Figure 5.9, the root only has three non-empty branches representing letter 'a', 'b', and 'z'. Other branches such as for letter c, are all empty. We don't show empty branches in the rest of this chapter.

If deal with case sensitive problems, or handle languages other than English, there can be more letters than 26. The problem of dynamic size of sub branches can be solved by using some collection data structures. Such as Hash table or map.

A alphabetic trie is either empty or a node. There are two types of node.

- A leaf node don't has any sub trees;
- A branch node contains multiple sub trees. Each sub tree is bound to a character.

Both leaf and branch may contain optional satellite data. The following Haskell code shows the example definition.

```
data Trie a = Trie { value :: Maybe a
                   , children :: [(Char, Trie a)]}

empty = Trie Nothing []
```

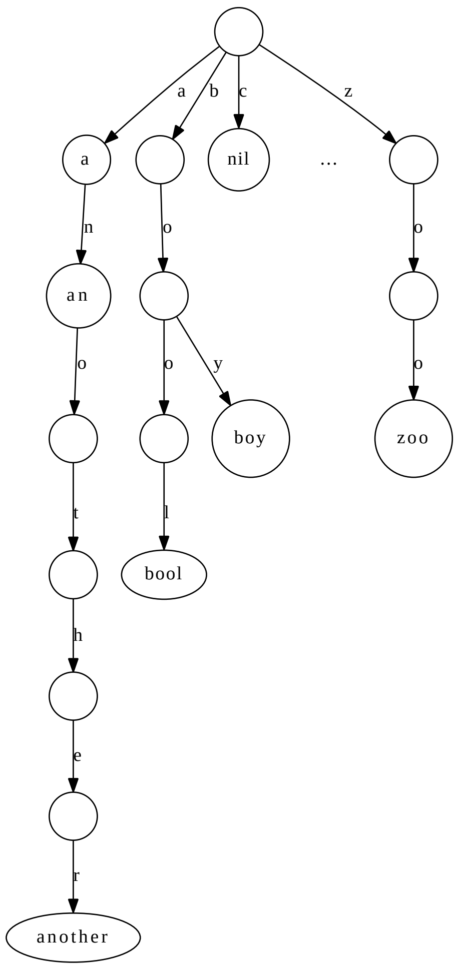


Figure 5.9: A Trie with 26 branches, containing key 'a', 'an', 'another', 'bool', 'boy' and 'zoo'.

Below ANSI C code defines the alphabetic trie. For illustration purpose only, we limit the character set to lower case English letters, from 'a' to 'z'.

```
struct Trie {
  struct Trie* children[26];
  void* data;
};
```

5.4.2 Insertion

When insert string as key, start from the root, we pick the character one by one from the string, examine which child represents the character. If the corresponding child is empty, a new empty node is created. After that, the next character is used to select the proper grand child.

We repeat this process for all the characters, and finally store the optional satellite data in the node we arrived at.

Below pseudo code describes the insertion algorithm.

```
1: function INSERT($T, k, v$)
2:     if $T$ = NIL then
3:         $T \leftarrow$ EMPTY-NODE
4:     $p \leftarrow T$
5:     for each $c$ in $k$ do
6:         if CHILDREN($p$)[c] = NIL then
7:             CHILDREN($p$)[c] $\leftarrow$ EMPTY-NODE
8:         $p \leftarrow$ CHILDREN($p$)[c]
9:     DATA($p$) $\leftarrow v$
10:    return $T$
```

The following example ANSI C program implements the insertion algorithm.

```
struct Trie* insert(struct Trie* t, const char* key, void* value){
    int c;
    struct Trie *p;
    if(!t)
        t = create_node();
    for (p = t; *key; ++key, p = p->children[c]) {
        c = *key - 'a';
        if (!p->children[c])
            p->children[c] = create_node();
    }
    p->data = value;
    return t;
}
```

Where function create_node creates new empty node, with all children initialized to empty.

```
struct Trie* create_node(){
    struct Trie* t = (struct Trie*) malloc(sizeof(struct Trie));
    int i;
    for (i=0; i<26; ++i)
        t->children[i] = NULL;
    t->data = NULL;
    return t;
}
```

The insertion can also be realized in recursive way. Denote the key to be inserted as $K = k_1 k_2 ... k_n$, where k_i is the i-th character. K' is the rest of characters except k_1. v' is the satellite data to be inserted. The trie is in form $T = (v, C)$, where v is the satellite data, $C = \{(c_1, T_1), (c_2, T_2), ..., (c_m, T_m)\}$ is the map of children. It maps from character c_i to sub-tree T_i. if T is empty, then C is also empty.

$$insert(T, K, v') = \begin{cases} (v', C) & : K = \Phi \\ (v, ins(C, k_1, K', v')) & : otherwise. \end{cases} \tag{5.8}$$

If the key is empty, the previous value v is overwritten with v'. Otherwise, we need check the children and perform recursive insertion. This is realized in function $ins(C, k_1, K', v')$. It examines key-sub tree pairs in C one by one. Let C' be the rest of pairs except for the first one. This function can be defined as below.

$$ins(C, k_1, K', v') = \begin{cases} \{(k_1, insert(\Phi, K', v'))\} & : C = \Phi \\ \{k_1, insert(T_1, K', v')\} \cup C' & : k_1 = c_1 \\ \{(c_1, T_1)\} \cup ins(C', k_1, K', v') & : otherwise \end{cases} \tag{5.9}$$

If C is empty, we create a pair, mapping from character k_1 to a new empty tree, and recursively insert the rest characters. Otherwise, the algorithm locates the child which is mapped from k_1 for further insertion.

The following Haskell example program implements the insertion algorithm.

```
insert t []     x = Trie (Just x)  (children t)
insert t (k:ks) x = Trie (value t) (ins (children t) k ks x) where
    ins [] k ks x = [(k, (insert empty ks x))]
    ins (p:ps) k ks x = if fst p == k
                        then (k, insert (snd p) ks x):ps
                        else p:(ins ps k ks x)
```

5.4.3 Look up

To look up a key, we also extract the character from the key one by one. For each character, we search among the children to see if there is a branch match this character. If there is no such a child, the look up process terminates immediately to indicate the not found error. When we reach the last character of the key, The data stored in the current node is what we are looking up.

```
1: function LOOK-UP($T, key$)
2:     if $T$ = NIL then
3:         return not found
4:     for each $c$ in $key$ do
5:         if CHILDREN($T$)[$c$] = NIL then
6:             return not found
7:         $T \leftarrow$ CHILDREN($T$)[$c$]
8:     return DATA($T$)
```

Below ANSI C program implements the look up algorithm. It returns NULL to indicate not found error.

```
void* lookup(struct Trie* t, const char* key) {
    while (*key && t && t→children[*key - 'a'])
        t = t→children[*key++ - 'a'];
    return (*key || !t) ? NULL : t→data;
}
```

The look up algorithm can also be realized in recursive manner. When look up a key, we start from the first character. If it is bound to some child, we then recursively search the rest characters in that child. Denote the trie as $T = (v, C)$, the key being searched as $K = k_1 k_2 ... k_n$ if it isn't empty. The first character in the key is k_1, and the rest characters are denoted as K'.

$$lookup(T, K) = \begin{cases} v & : K = \Phi \\ \Phi & : find(C, k_1) = \Phi \\ lookup(T', K') & : find(C, k_1) = T' \end{cases} \tag{5.10}$$

Where function $find(C, k)$ examine the pairs of key-child one by one to check if any child is bound to character k. If the list of pairs C is empty, the result is empty to indicate non-existence of such a child. Otherwise, let $C = \{(k_1, T_1), (k_2, T_2), ..., (k_m, T_m)\}$, the first sub tree T_1 is bound to k_1; the rest of pairs are represented as C'. Below equation defines the $find$ function.

$$find(C, k) = \begin{cases} \Phi & : C = \Phi \\ T_1 & : k_1 = k \\ find(C', k) & : otherwise \end{cases} \tag{5.11}$$

The following Haskell example program implements the trie looking up algorithm. It uses the `find` function provided in standard library[5].

```
find t [] = value t
find t (k:ks) = case lookup k (children t) of
                  Nothing → Nothing
                  Just t' → find t' ks
```

Exercise 5.1

- Develop imperative trie by using collection data structure to manage multiple sub trees in alphabetic trie.

5.5 Alphabetic Patricia

Similar to integer trie, alphabetic trie is not memory efficient. We can use the same method to compress alphabetic trie to Patricia.

5.5.1 Definition

Alphabetic Patricia tree is a special prefix tree, each node contains multiple branches. All children of a node share the longest common prefix string. As the result, there is no node with only one child, because it conflicts with the longest common prefix property.

If we turn the trie shown in figure 5.9 into Patricia by compressing all nodes which have only one child. we can get a Patricia prefix tree as in figure 5.10.

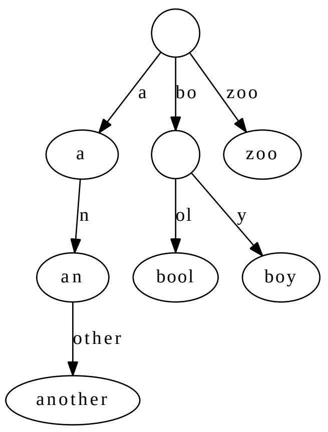


Figure 5.10: A Patricia prefix tree, with keys: 'a', 'an', 'another', 'bool', 'boy' and 'zoo'.

We can modify the definition of alphabetic trie a bit to adapt it to Patricia. The Patricia is either empty, or a node in form $T = (v, C)$. Where v is the optional satellite data; $C = \{(s_1, T_1), (s_2, T_2), ..., (s_n, T_n)\}$ is a list of pairs. Each pair contains a string s_i, which is bound to a sub tree T_i.

The following Haskell example code defines Patricia accordingly.

```haskell
type Key = String

data Patricia a = Patricia { value :: Maybe a
                           , children :: [(Key, Patricia a)]}

empty = Patricia Nothing []
```

Below Python code reuses the definition for trie to define Patricia.

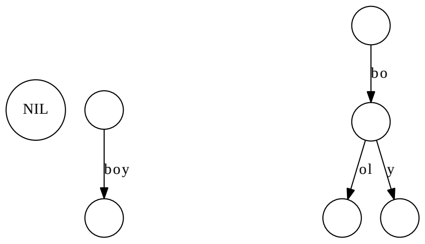
```python
class Patricia:
    def __init__(self, value = None):
        self.value = value
        self.children = {}
```

5.5.2 Insertion

When insert a key, s, if the Patricia is empty, we create a leaf node as shown in figure 5.11 (a). Otherwise, we need check the children. If there is some sub tree T_i bound to the string s_i, and there exists common prefix between s_i and s, we need branch out a new leaf T_j. The method is to create a new internal branch node, bind it with the common prefix. Then set T_i as one child of this branch, and T_j as the other child. T_i and T_j share the common prefix. This is shown in figure 5.11 (b). However, there are two special cases, because s may be the prefix of s_i as shown in figure 5.11 (c). And s_i may be the prefix of s as in figure 5.11 (d).

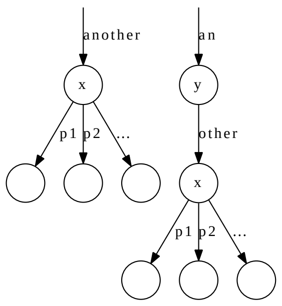
The insertion algorithm can be described as below.

```
1: function Insert($T, k, v$)
2:     if $T$ = NIL then
```

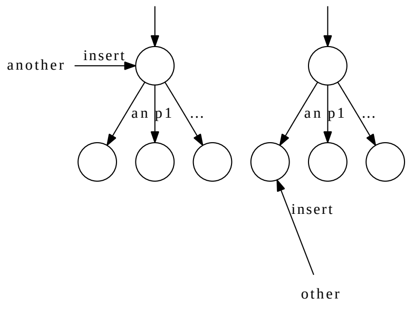


(a) Insert key 'boy' into the empty Patricia, the result is a leaf.

(b) Insert key 'bool'. A new branch with common prefix 'bo' is created.



(c) Insert key 'an' with value y into x with prefix 'another'.



(d) Insert 'another', into the node with prefix 'an'. We recursively insert key 'other' to the child.

Figure 5.11: Patricia insertion

```
        $T \leftarrow$ EMPTY-NODE
    $p \leftarrow T$
    loop
        $match \leftarrow$ FALSE
        for each $(s_i, T_i) \in$ CHILDREN($p$) do
            if $k = s_i$ then
                VALUE($p$) $\leftarrow v$
                return $T$
            $c \leftarrow$ LCP($k, s_i$)
            $k_1 \leftarrow k - c$
            $k_2 \leftarrow s_i - c$
            if $c \neq$ NIL then
                $match \leftarrow$ TRUE
                if $k_2 =$ NIL then                         ▷ $s_i$ is prefix of $k$
                    $p \leftarrow T_i$
                    $k \leftarrow k_1$
                    break
                else                                        ▷ Branch out a new leaf
                    CHILDREN($p$) $\leftarrow$ CHILDREN($p$) $\cup${ ($c$, BRANCH($k_1, v, k_2, T_i$))
}
                    DELETE(CHILDREN($p$), $(s_i, T_i)$)
                    return $T$
        if $\neg match$ then                                ▷ Add a new leaf
            CHILDREN($p$) $\leftarrow$ CHILDREN($p$) $\cup${ ($k$, CREATE-LEAF($v$)) }
            return $T$
    return $T$
```

In the above algorithm, LCP function finds the longest common prefix of two given strings, for example, string ‘bool’ and ‘boy’ has the longest common prefix ‘bo’. The subtraction symbol ’-’ for strings gives the different part of two strings. For example ‘bool’ - ‘bo’ = ‘ol’. BRANCH function creates a branch node and updates keys accordingly.

The longest common prefix can be extracted by checking the characters in the two strings one by one till there are two characters don’t match.

```
function LCP($A, B$)
    $i \leftarrow 1$
    while $i \leq |A| \wedge i \leq |B| \wedge A[i] = B[i]$ do
        $i \leftarrow i + 1$
    return $A[1...i-1]$
```

There are two cases when branch out a new leaf. BRANCH(s_1, T_1, s_2, T_2) takes two different keys and two trees. If s_1 is empty, we are dealing the case such as insert key ‘an’ into a child bound to string ‘another’. We set T_2 as the child of T_1. Otherwise, we create a new branch node and set T_1 and T_2 as the two children.

```
function BRANCH($s_1, T_1, s_2, T_2$)
    if $s_1 = \Phi$ then
        CHILDREN($T_1$) $\leftarrow$ CHILDREN($T_1$) $\cup\{(s_2, T_2)\}$
        return $T_1$
    $T \leftarrow$ EMPTY-NODE
```

```
6:     CHILDREN($T$) ← $\{(s_1, T_1), (s_2, T_2)\}$
7:     return $T$
```

The following example Python program implements the Patricia insertion algorithm.

```
def insert(t, key, value = None):
    if t is None:
        t = Patricia()
    node = t
    while True:
        match = False
        for k, tr in node.children.items():
            if key == k: # just overwrite
                node.value = value
                return t
            (prefix, k1, k2) = lcp(key, k)
            if prefix != "":
                match = True
                if k2 == "":
                    # example: insert "another" into "an", go on traversing
                    node = tr
                    key = k1
                    break
                else: #branch out a new leaf
                    node.children[prefix] = branch(k1, Patricia(value), k2, tr)
                    del node.children[k]
                    return t
        if not match: # add a new leaf
            node.children[key] = Patricia(value)
            return t
    return t
```

Where the functions to find the longest common prefix, and branch out are implemented as below.

```
# returns (p, s1', s2'), where p is lcp, s1'=s1-p, s2'=s2-p
def lcp(s1, s2):
    j = 0
    while j < len(s1) and j < len(s2) and s1[j] == s2[j]:
        j += 1
    return (s1[0:j], s1[j:], s2[j:])

def branch(key1, tree1, key2, tree2):
    if key1 == "":
        #example: insert "an" into "another"
        tree1.children[key2] = tree2
        return tree1
    t = Patricia()
    t.children[key1] = tree1
    t.children[key2] = tree2
    return t
```

The insertion can also be realized recursively. Start from the root, the program checks all the children to find if there is a node matches the key. Matching means they have the common prefix. For duplicated keys, the program overwrites previous value. There are also alternative solution to handle duplicated keys, such as using linked-list etc. If there is no child matches the key, the program creates a new leaf, and add it to the children.

For Patricia $T = (v, C)$, function $insert(T, k, v')$ inserts key k, and value v' to the tree.

$$insert(T, k, v') = (v, ins(C, k, v')) \tag{5.12}$$

This function calls another internal function $ins(C, k, v')$. If the children C is empty, a new leaf is created; Otherwise the children are examined one by one. Denote $C = \{(k_1, T_1), (k_2, T_2), ..., (k_n, T_n)\}$, C' holds all the prefix-sub tree pairs except for the first one.

$$ins(C, k, v') = \left\{ \begin{array}{rcl} \{(k, (v', \Phi))\} & : & C = \Phi \\ \{(k, (v', C_{T_1}))\} \cup C' & : & k_1 = k \\ \{branch(k, v', k_1, T_1)\} \cup C' & : & match(k_1, k) \\ \{(k_1, T_1)\} \cup ins(C', k, v') & : & otherwise \end{array} \right. \tag{5.13}$$

The first clause deals with the edge case of empty children. A leaf node containing v' which is bound to k will be returned as the only child. The second clause overwrites the previous value with v' if there is some child bound to the same key. Note the C_{T_1} means the children of sub tree T_1. The third clause branches out a new leaf if the first child matches the key k. The last clause goes on checking the rest children.

We define two keys A and B matching if they have non-empty common prefix.

$$match(A, B) = A \neq \Phi \land B \neq \Phi \land a_1 = b_1 \tag{5.14}$$

Where a_1 and b_1 are the first characters in A and B if they are not empty.

Function $branch(k_1, v, k_2, T_2)$ takes tow keys, a value and a tree. Extract the longest common prefix $k = lcp(k_1, k_2)$, Denote the different part as $k'_1 = k_1 - k$, $k'_2 = k_2 - k$. The algorithm firstly handles the edge cases that either k_1 is the prefix of k_2 or k_2 is the prefix of k_1. For the former case, It creates a new node containing v, bind this node to k, and set (k'_2, T_2) as the only child; For the later case, It recursively insert k'_1 and v to T_2. Otherwise, the algorithm creates a branch node, binds it to the longest common prefix k, and set two children for it. One child is (k'_2, T_2), the other is a leaf node containing v, and being bound to k'_1.

$$branch(k_1, v, k_2, T_2) = \left\{ \begin{array}{rcl} (k, (v, \{(k'_2, T_2)\})) & : & k = k_1 \\ (k, insert(T_2, k'_1, v)) & : & k = k_2 \\ (k, (\Phi, \{(k'_1, (v, \Phi)), (k'_2, T_2)\})) & : & otherwise \end{array} \right. \tag{5.15}$$

And function $lcp(A, B)$ keeps taking same characters from A and B one by one. Denote a_1 and b_1 as the first characters in A and B if they are not empty. A' and B' are the rest parts except for the first characters.

$$lcp(A, B) = \left\{ \begin{array}{rcl} \Phi & : & A = \Phi \lor B = \Phi \lor a_1 \neq b_1 \\ \{a_1\} \cup lcp(A', B') & : & a_1 = b_1 \end{array} \right. \tag{5.16}$$

The following Haskell example program implements the Patricia insertion algorithm.

```
insert t k x = Patricia (value t) (ins (children t) k x) where
    ins []     k x = [(k, Patricia (Just x) [])]
    ins (p:ps) k x
        | (fst p) == k
           = (k, Patricia (Just x) (children (snd p))):ps --overwrite
        | match (fst p) k
           = (branch k x (fst p) (snd p)):ps
        | otherwise
           = p:(ins ps k x)

match x y = x /= [] && y /= [] && head x == head y

branch k1 x k2 t2
    | k1 == k
        -- ex: insert "an" into "another"
        = (k, Patricia (Just x) [(k2', t2)])
    | k2 == k
        -- ex: insert "another" into "an"
        = (k, insert t2 k1' x)
    | otherwise = (k, Patricia Nothing [(k1', leaf x), (k2', t2)])
   where
      k = lcp k1 k2
      k1' = drop (length k) k1
      k2' = drop (length k) k2

lcp [] _ = []
lcp _ [] = []
lcp (x:xs) (y:ys) = if x == y then x:(lcp xs ys) else []
```

5.5.3 Look up

When look up a key, we can't examine the characters one by one as in trie. Start from the root, we need search among the children to see if any one is bound to a prefix of the key. If there is such a child, we update the key by removing the prefix part, and recursively look up the updated key in this child. If there aren't any children bound to any prefix of the key, the looking up fails.

```
1: function LOOK-UP($T, k$)
2:     if $T$ = NIL then
3:         return not found
4:     repeat
5:         $match \leftarrow$ FALSE
6:         for $\forall(k_i, T_i) \in$ CHILDREN($T$) do
7:             if $k = k_i$ then
8:                 return DATA($T_i$)
9:             if $k_i$ is prefix of $k$ then
10:                $match \leftarrow$ TRUE
11:                $k \leftarrow k - k_i$
12:                $T \leftarrow T_i$
13:                break
```

```
14:     until $\neg match$
15:     return not found
```

Below Python example program implements the looking up algorithm. It reuses the `lcp(s1, s2)` function defined previously to test if a string is the prefix of the other.

```
def lookup(t, key):
    if t is None:
        return None
    while True:
        match = False
        for k, tr in t.children.items():
            if k == key:
                return tr.value
            (prefix, k1, k2) = lcp(key, k)
            if prefix != "" and k2 == "":
                match = True
                key = k1
                t = tr
                break
        if not match:
            return None
```

This algorithm can also be realized recursively. For Patricia in form $T = (v, C)$, it calls another function to find among the children C.

$$lookup(T, k) = find(C, k) \tag{5.17}$$

If C is empty, the looking up fails; Otherwise, For $C = \{(k_1, T_1), (k_2, T_2), ..., (k_n, T_n)\}$, we firstly examine if k is the prefix of k_1, if not the recursively check the rest pairs denoted as C'.

$$find(C, k) = \begin{cases} \Phi & : C = \Phi \\ v_{T_1} & : k = k_1 \\ lookup(T_1, k - k_1) & : k_1 \sqsubset k \\ find(C', k) & : otherwise \end{cases} \tag{5.18}$$

Where $A \sqsubset B$ means string A is prefix of B. $find$ mutually calls $lookup$ if some child is bound to a string which is prefix of the key.

Below Haskell example program implements the looking up algorithm.

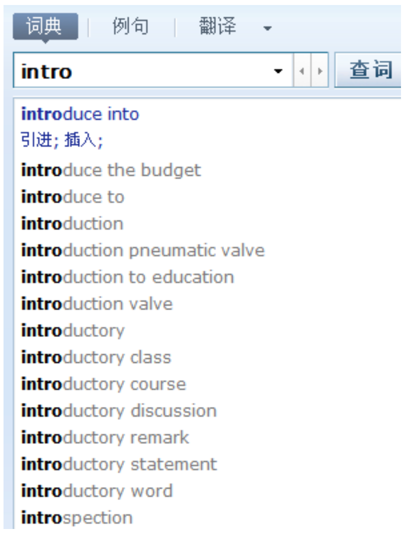
```
import qualified Data.List

find t k = find' (children t) k where
    find' [] _ = Nothing
    find' (p:ps) k
          | (fst p) == k = value (snd p)
          | (fst p) `Data.List.isPrefixOf` k = find (snd p) (diff (fst p) k)
          | otherwise = find' ps k
    diff k1 k2 = drop (length (lcp k1 k2)) k2
```

5.6 Trie and Patricia applications

Trie and Patricia can be used to solving some interesting problems. Integer based prefix tree is used in compiler implementation. Some daily used software applications have many interesting features which can be realized with trie or Patricia. In this section, some applications are given as examples, including, e-dictionary, word auto-completion, T9 input method etc. The commercial implementations typically do not adopt trie or Patricia directly. The solutions we demonstrated here are for illustration purpose only.

5.6.1 E-dictionary and word auto-completion

Figure 5.12 shows a screen shot of an English-Chinese E-dictionary. In order to provide good user experience, the dictionary searches its word library, and lists all candidate words and phrases similar to what user has entered.



intro

introduce into
引进; 插入;
introduce the budget
introduce to
introduction
introduction pneumatic valve
introduction to education
introduction valve
introductory
introductory class
introductory course
introductory discussion
introductory remark
introductory statement
introductory word
introspection

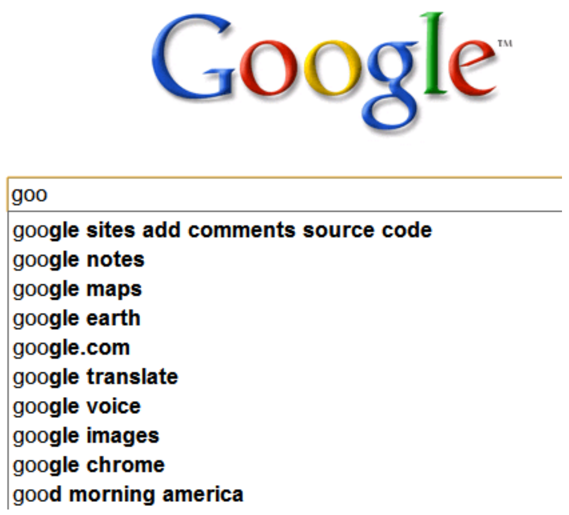
Figure 5.12: E-dictionary. All candidates starting with what the user input are listed.

A E-dictionary typically contains hundreds of thousands words. It's very expensive to performs a whole word search. Commercial software adopts complex approaches, including caching, indexing etc to speed up this process.

Similar with e-dictionary, figure 5.13 shows a popular Internet search engine. When user input something, it provides a candidate lists, with all items starting with what the user has entered. And these candidates are shown in the order of popularity. The more people search, the upper position it is in the list.

In both cases, the software provides a kind of word auto-completion mechanism. In some modern IDEs, the editor can even help users to auto-complete program code.

Let's see how to implementation of the e-dictionary with trie or Patricia. To simplify the problem, we assume the dictionary only supports English - English information.


goo

google sites add comments source code
google notes
google maps
google earth
google.com
google translate
google voice
google images
google chrome
good morning america

Figure 5.13: A search engine. All candidates starting with what user input are listed.

A dictionary stores key-value pairs, the keys are English words or phrases, the values are the meaning described in English sentences.

We can store all the words and their meanings in a trie, but it isn't space effective especially when there are huge amount of items. We'll use Patricia to realize e-dictionary.

When user wants to look up word 'a', the dictionary does not only return the meaning of 'a', but also provides a list of candidate words, which all start with 'a', including 'abandon', 'about', 'accent', 'adam', ... Of course all these words are stored in the Patricia.

If there are too many candidates, one solution is only displaying the top 10 words, and the user can browse for more.

The following algorithm reuses the looking up defined for Patricia. When it finds a node bound to a string which is the prefix of what we are looking for, it expands all its children until getting n candidates.

```
function LOOK-UP($T, k, n$)
    if $T$ = NIL then
        return $\Phi$
    $prefix \leftarrow$ NIL
    repeat
        $match \leftarrow$ FALSE
        for $\forall (k_i, T_i) \in$ CHILDREN($T$) do
            if $k$ is prefix of $k_i$ then
                return EXPAND($T_i, prefix, n$)
            if $k_i$ is prefix of $k$ then
                $match \leftarrow$ TRUE
                $k \leftarrow k - k_i$
                $T \leftarrow T_i$
                $prefix \leftarrow prefix + k_i$
```

```
15:                 break
16:     until $\neg match$
17:     return $\Phi$
```

Where function Expand($T, prefix, n$) picks n sub trees, which share the same prefix in T. It is realized as BFS (Bread-First-Search) traverse. Chapter search explains BFS in detail.

```
1: function Expand($T, prefix, n$)
2:     $R \leftarrow \Phi$
3:     $Q \leftarrow \{(prefix, T)\}$
4:     while $|R| < n \land |Q| > 0$ do
5:         $(k, T) \leftarrow$ Pop($Q$)
6:         if Data($T$) $\neq$ NIL then
7:             $R \leftarrow R \cup \{(k,$ Data($T$) $)\}$
8:         for $\forall(k_i, T_i) \in$ Children($T$) do
9:             Push($Q, (k + k_i, T_i)$)
```

The following example Python program implements the e-dictionary application. When testing if a string is prefix of another one, it uses the `find` function provided in standard string library.

```python
import string

def patricia_lookup(t, key, n):
    if t is None:
        return None
    prefix = ""
    while True:
        match = False
        for k, tr in t.children.items():
            if string.find(k, key) == 0: #is prefix of
                return expand(prefix+k, tr, n)
            if string.find(key, k) ==0:
                match = True
                key = key[len(k):]
                t = tr
                prefix += k
                break
        if not match:
            return None

def expand(prefix, t, n):
    res = []
    q = [(prefix, t)]
    while len(res)<n and len(q)>0:
        (s, p) = q.pop(0)
        if p.value is not None:
            res.append((s, p.value))
        for k, tr in p.children.items():
            q.append((s+k, tr))
    return res
```

This algorithm can also be implemented recursively, if the string we are looking for is empty, we expand all children until getting n candidates. Otherwise we recursively examine the children to find one which has prefix equal to this string.

In programming environments supporting lazy evaluation. An intuitive solution is to expand all candidates, and take the first n on demand. Denote the Patricia prefix tree in form $T = (v, C)$, below function enumerates all items starts with key k.

$$findAll(T, k) = \begin{cases} enum(C) & : k = \Phi, v = \Phi \\ \{(\Phi, v)\} \cup enum(C) & : k = \Phi, v \neq \Phi \\ find(C, k) & : k \neq \Phi \end{cases} \tag{5.19}$$

The first two clauses deal with the edge cases the the key is empty. All the children are enumerated except for those with empty values. The last clause finds child matches k.

For non-empty children, $C = \{(k_1, T_1), (k_2, T_2), ..., (k_m, T_m)\}$, denote the rest pairs except for the first one as C'. The enumeration algorithm can be defined as below.

$$enum(C) = \begin{cases} \Phi & : C = \Phi \\ mapAppend(k_1, findAll(T_1, \Phi)) \cup enum(C') & : \end{cases} \tag{5.20}$$

Where $mapAppend(k, L) = \{(k + k_i, v_i) | (k_i, v_i) \in L\}$. It concatenate the prefix k in front of every key-value pair in list L.

Function $find(C, k)$ is defined as the following. For empty children, the result is empty as well; Otherwise, it examines the first child T_1 which is bound to string k_1. If k_1 is equal to k, it calls $mapAppend$ to add prefix to the keys of all the children under T_1; If k_1 is prefix of k, the algorithm recursively find all children start with $k - k_1$; On the other hand, if k is prefix of k_1, all children under T_1 are valid result. Otherwise, the algorithm by-pass the first child and goes on find the rest children.

$$find(C, k) = \begin{cases} \Phi & : C = \Phi \\ mapAppend(k, findAll(T_1, \Phi)) & : k_1 = k \\ mapAppend(k_1, findAll(T_1, k - k_1)) & : k_1 \sqsubset k \\ findAll(T_1, \Phi) & : k \sqsubset k_1 \\ find(C', k) & : otherwise \end{cases} \tag{5.21}$$

Below example Haskell program implements the e-dictionary application according to the above equations.

```
findAll :: Patricia a → Key → [(Key, a)]
findAll t [] =
    case value t of
      Nothing → enum $ children t
      Just x  → ("", x):(enum $ children t)
    where
      enum [] = []
      enum (p:ps) = (mapAppend (fst p) (findAll (snd p) [])) ++ (enum ps)
findAll t k = find' (children t) k where
    find' [] _ = []
    find' (p:ps) k
          | (fst p) == k
             = mapAppend k (findAll (snd p) [])
          | (fst p) `Data.List.isPrefixOf` k
             = mapAppend (fst p) (findAll (snd p) (k `diff` (fst p)))
          | k `Data.List.isPrefixOf` (fst p)
             = findAll (snd p) []
          | otherwise = find' ps k
    diff x y = drop (length y) x

mapAppend s lst = map (λp→(s++(fst p), snd p)) lst
```

In the lazy evaluation environment, the top n candidates can be gotten like $take(n, findAll(T, k))$. Appendix A has detailed definition of *take* function.

5.6.2 T9 input method

Most mobile phones around year 2000 are equipped with a key pad. Users have quite different experience from PC when editing a short message or email. This is because the mobile-phone key pad, or so called ITU-T key pad has much fewer keys than PC. Figure 5.14 shows one example.

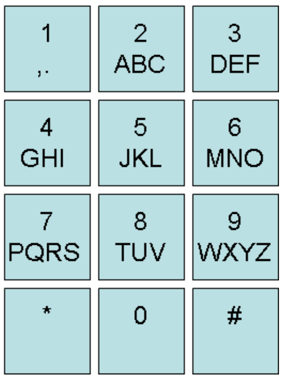


Figure 5.14: an ITU-T keypad for mobile phone.

There are typical two methods to input English word or phrases with ITU-T key pad. For instance, if user wants to enter a word 'home', He can press the key in below sequence.

- Press key '4' twice to enter the letter 'h';
- Press key '6' three times to enter the letter 'o';
- Press key '6' twice to enter the letter 'm';
- Press key '3' twice to enter the letter 'e';

Another much quicker way is to just press the following keys.

- Press key '4', '6', '6', '3', word 'home' appears on top of the candidate list;
- Press key '*' to change a candidate word, so word 'good' appears;

- Press key '*' again to change another candidate word, next word 'gone' appears;
- ...

Compare these two methods, we can see the second one is much easier for the end user. The only overhead is to store a dictionary of candidate words.

Method 2 is called as 'T9' input method, or predictive input method [6], [7]. The abbreviation 'T9' stands for 'textonym'. It start with 'T' with 9 characters. T9 input can also be realized with trie or Patricia.

In order to provide candidate words to user, a dictionary must be prepared in advance. Trie or Patricia can be used to store the dictionary. The commercial T9 implementations typically use complex indexing dictionary in both file system and cache. The realization shown here is for illustration purpose only.

Firstly, we need define the T9 mapping, which maps from digit to candidate characters.

$$
\begin{array}{rl}
M_{T9} = \{ & 2 \rightarrow abc, 3 \rightarrow def, 4 \rightarrow ghi, \\
 & 5 \rightarrow jkl, 6 \rightarrow mno, 7 \rightarrow pqrs, \\
 & 8 \rightarrow tuv, 9 \rightarrow wxyz\}
\end{array}
\tag{5.22}
$$

With this mapping, $M_{T9}[i]$ returns the corresponding characters for digit i.

Suppose user input digits $D = d_1 d_2 ... d_n$, If D isn't empty, denote the rest digits except for d_1 as D', below pseudo code shows how to realize T9 with trie.

```
1: function LOOK-UP-T9($T, D$)
2:     $Q \leftarrow \{(\Phi, D, T)\}$
3:     $R \leftarrow \Phi$
4:     while $Q \neq \Phi$ do
5:         $(prefix, D, T) \leftarrow$ POP($Q$)
6:         for each $c$ in $M_{T9}[d_1]$ do
7:             if $c \in$ CHILDREN($T$) then
8:                 if $D' = \Phi$ then
9:                     $R \leftarrow R \cup \{prefix + c\}$
10:                else
11:                    PUSH($Q, (prefix + c, D'$, CHILDREN($t$)[$c$]))
12:    return $R$
```

Where $prefix + c$ means appending character c to the end of string $prefix$. Again, this algorithm performs BFS search with a queue Q. The queue is initialized with a tuple $(prefix, D, T)$, containing empty prefix, the digit sequence to be searched, and the trie. It keeps picking the tuple from the queue as far as it isn't empty. Then get the candidate characters from the first digit to be processed via the T9 map. For each character c, if there is a sub-tree bound to it, we created a new tuple, update the prefix by appending c, using the rest of digits to update D, and use that sub-tree. This new tuple is pushed back to the queue for further searching. If all the digits are processed, it means a candidate word is found. We put this word to the result list R.

The following example program in Python implements this T9 search with trie.

```
T9MAP={'2':"abc", '3':"def", '4':"ghi", '5':"jkl", λ
       '6':"mno", '7':"pqrs", '8':"tuv", '9':"wxyz"}
```

```
def trie_lookup_t9(t, key):
    if t is None or key == "":
        return None
    q = [("", key, t)]
    res = []
    while len(q)>0:
        (prefix, k, t) = q.pop(0)
        i=k[0]
        if not i in T9MAP:
            return None #invalid input
        for c in T9MAP[i]:
            if c in t.children:
                if k[1:]=="":
                    res.append((prefix+c, t.children[c].value))
                else:
                    q.append((prefix+c, k[1:], t.children[c]))
    return res
```

Because trie is not space effective, we can modify the above algorithm with Patricia solution. As far as the queue isn't empty, the algorithm pops the tuple. This time, we examine all the prefix-sub tree pairs. For every pair (k_i, T_i), we convert the alphabetic prefix k_i back to digits sequence D' by looking up the T9 map. If D' exactly matches the digits of what user input, we find a candidate word; otherwise if the digit sequence is prefix of what user inputs, the program creates a new tuple, updates the prefix, the digits to be processed, and the sub-tree. Then put the tuple back to the queue for further search.

```
1: function LOOK-UP-T9($T, D$)
2:     $Q \leftarrow \{(\Phi, D, T)\}$
3:     $R \leftarrow \Phi$
4:     while $Q \neq \Phi$ do
5:         $(prefix, D, T) \leftarrow$ POP($Q$)
6:         for each $(k_i, T_i) \in$ CHILDREN($T$) do
7:             $D' \leftarrow$ CONVERT-T9($k_i$)
8:             if $D' \sqsubset D$ then                    ▷ $D'$ is prefix of $D$
9:                 if $D' = D$ then
10:                    $R \leftarrow R \cup \{prefix + k_i\}$
11:                else
12:                    PUSH($Q, (prefix + k_i, D - D', T_i)$)
13:    return $R$
```

Function CONVERT-T9(K) converts each character in K back to digit.

```
1: function CONVERT-T9($K$)
2:     $D \leftarrow \Phi$
3:     for each $c \in K$ do
4:         for each $(d \rightarrow S) \in M_{T9}$ do
5:             if $c \in S$ then
6:                 $D \leftarrow D \cup \{d\}$
7:                 break
8:     return $D$
```

The following example Python program implements the T9 input method with Patricia.

```
def patricia_lookup_t9(t, key):
    if t is None or key == "":
        return None
    q = [("", key, t)]
    res = []
    while len(q)>0:
        (prefix, key, t) = q.pop(0)
        for k, tr in t.children.items():
            digits = toT9(k)
            if string.find(key, digits)==0: #is prefix of
                if key == digits:
                    res.append((prefix+k, tr.value))
                else:
                    q.append((prefix+k, key[len(k):], tr))
    return res
```

T9 input method can also be realized recursively. Let's first define the trie solution. The algorithm takes two arguments, a trie storing all the candidate words, and a sequence of digits that is input by the user. If the sequence is empty, the result is empty as well; Otherwise, it looks up C to find those children which are bound to the first digit d_1 according to T9 map.

$$findT9(T, D) = \begin{cases} \{\Phi\} & : D = \Phi \\ fold(f, \Phi, lookupT9(d_1, C)) & : otherwise \end{cases} \tag{5.23}$$

Where folding is defined in Appendix A. Function f takes two arguments, an intermediate list of candidates which is initialized empty, and a pair (c, T'), where c is candidate character, to which sub tree T' is bound. It append character c to all the candidate words, and concatenate this to the result list.

$$f(L, (c, T')) = mapAppend(c, findT9(T', D')) \cup L \tag{5.24}$$

Note this *mapAppend* function is a bit different from the previous one defined in e-dictionary application. The first argument is a character, but not a string.

Function $lookupT9(k, C)$ checks all the possible characters mapped to digit k. If the character is bound to some child in C, it is record as one candidate.

$$lookupT9(d, C) = fold(g, \Phi, M_{T9}[k]) \tag{5.25}$$

Where

$$g(L, k) = \begin{cases} L & : find(C, k) = \Phi \\ \{(k, T')\} \cup L & : find(C, k) = T' \end{cases} \tag{5.26}$$

Below Haskell example program implements the T9 look up algorithm with trie.

```
mapT9 = [('2', "abc"), ('3', "def"), ('4', "ghi"), ('5', "jkl"),
         ('6', "mno"), ('7', "pqrs"), ('8', "tuv"), ('9', "wxyz")]

findT9 t [] = [("", value t)]
findT9 t (k:ks) = foldl f [] (lookupT9 k (children t))
    where
      f lst (c, tr) = (mapAppend' c (findT9 tr ks)) ++ lst
```

```
lookupT9 c children = case lookup c mapT9 of
        Nothing → []
        Just s  → foldl f [] s where
             f lst x = case lookup x children of
                 Nothing → lst
                 Just t  → (x, t):lst

mapAppend' x lst = map (λp→(x:(fst p), snd p)) lst
```

There are few modifications when change the realization from trie to Patricia. Firstly, the sub-tree is bound to prefix string, but not a single character.

$$findT9(T, D) = \begin{cases} \{\Phi\} & : D = \Phi \\ fold(f, \Phi, findPrefixT9(D, C)) & : otherwise \end{cases} \tag{5.27}$$

The list for folding is given by calling function $findPrefixT9(D, C)$. And f is also modified to reflect this change. It appends the candidate prefix D' in front of every result output by the recursive search, and then accumulates the words.

$$f(L, (D', T')) = mapAppend(D', findT9(T', D - D')) \cup L \tag{5.28}$$

Function $findPrefixT9(D, C)$ examines all the children. For every pair (k_i, T_i), if converting k_i back to digits yields a prefix of D, then this pair is selected as a candidate.

$$findPrefixT9(D, C) = \{(k_i, T_i) | (k_i, T_i) \in C, convertT9(k_i) \sqsubset D\} \tag{5.29}$$

Function $convertT9(k)$ converts every alphabetic character in k back to digits according to T9 map.

$$convertT9(K) = \{d | \forall c \in k, \exists (d \to S) \in M_{T9} \Rightarrow c \in S\} \tag{5.30}$$

The following example Haskell program implements the T9 input algorithm with Patricia.

```
findT9 t [] = [("", value t)]
findT9 t k = foldl f [] (findPrefixT9 k (children t))
    where
      f lst (s, tr) = (mapAppend s (findT9 tr (k `diff` s))) ++ lst
      diff x y = drop (length y) x

findPrefixT9 s lst = filter f lst where
    f (k, _) = (toT9 k) `Data.List.isPrefixOf` s

toT9 = map (λc → head $ [ d |(d, s) ← mapT9, c `elem` s])
```

Exercise 5.2

- For the T9 input, compare the results of the algorithms realized with trie and Patricia, the sequences are different. Why does this happen? How to modify the algorithm so that they output the candidates with the same sequence?

5.7 Summary

In this chapter, we start from the integer base trie and Patricia. The map data structure based on integer Patricia plays an important role in Compiler implementation. Alphabetic trie and Patricia are natural extensions. They can be used to manipulate text information. As examples, predictive e-dictionary and T9 input method are realized with trie or Patricia. Although these examples are different from the real implementation in commercial software. They show simple approaches to solve some problems. Other important data structure, such as suffix tree, has close relationship with them. Suffix tree is introduced in the next chapter.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. “Introduction to Algorithms, Second Edition”. Problem 12-1. ISBN:0262032937. The MIT Press. 2001

[2] Chris Okasaki and Andrew Gill. “Fast Mergeable Integer Maps”. Workshop on ML, September 1998, pages 77-86, http://www.cse.ogi.edu/ andy/pub-/finite.htm

[3] D.R. Morrison, “PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric”, Journal of the ACM, 15(4), October 1968, pages 514-534.

[4] Suffix Tree, Wikipedia. http://en.wikipedia.org/wiki/Suffix_tree

[5] Trie, Wikipedia. http://en.wikipedia.org/wiki/Trie

[6] T9 (predictive text), Wikipedia. http://en.wikipedia.org/wiki/T9_(predictive_text)

[7] Predictive text, Wikipedia. http://en.wikipedia.org/wiki/Predictive_text

Chapter 6

Suffix Tree

6.1 Introduction

Suffix Tree is an important data structure. It can be used to realize many important string operations particularly fast[3]. It is also widely used in bio-information area such as DNA pattern matching[4]. Weiner introduced suffix tree in 1973[2]. The latest on-line construction algorithm was found in 1995[1].

The suffix tree for a string S is a special Patricia. Each edge is labeled with some sub-string of S. Each suffix of S corresponds to exactly one path from the root to a leaf. Figure 6.1 shows the suffix tree for English word 'banana'.

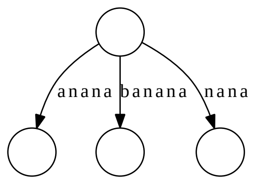


Figure 6.1: The suffix tree for 'banana'

All suffixes, 'banana', 'anana', 'nana', 'ana', 'na', 'a', '' can be found in the above tree. Among them the first 3 suffixes are explicitly shown; others are implicitly represented. The reason for why 'ana, 'na', 'a', and '' are not shown is because they are prefixes of the others. In order to show all suffixes explicitly, we can append a special pad terminal symbol, which doesn't occur in other places in the string. Such terminator is typically denoted as '\$'. By this means, there is no suffix being the prefix of the others.

Although the suffix tree for 'banana' is simple, the suffix tree for 'bananas', as shown in figure 6.2, is quite different.

To create suffix suffix tree for a given string, we can utilize the insertion algorithm explained in previous chapter for Patricia.

```
1: function SUFFIX-TREE($S$)
2:     $T \leftarrow$ NIL
3:     for $i \leftarrow 1$ to $|S|$ do
4:         $T \leftarrow$ PATRICIA-INSERT($T$, RIGHT($S, i$))
5:     return $T$
```

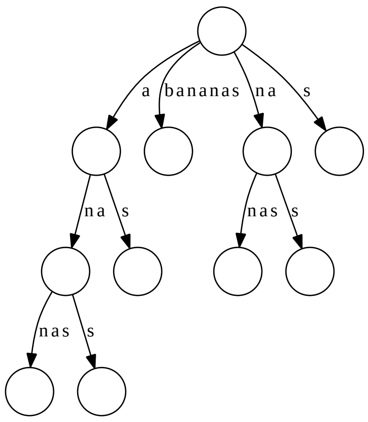


Figure 6.2: The suffix tree for 'bananas'

For non-empty string $S = s_1 s_2 ... s_i ... s_n$ of length $n = |S|$, function RIGHT(S, i) $= s_i s_{i+1 ... s_n}$. It extracts the sub-string of S from the i-th character to the last one. This straightforward algorithm can also be defined as below.

$$suffix_T(S) = fold(insert_{Patricia}, \Phi, suffixes(S)) \tag{6.1}$$

Where function $suffixes(S)$ gives all the suffixes for string S. If the string is empty, the result is one empty string; otherwise, S itself is one suffix, the others can be given by recursively call $suffixes(S')$, where S' is given by drop the first character from S.

$$suffixes(S) = \begin{cases} \{\Phi\} & : \quad S = \Phi \\ \{S\} \cup suffixes(S') & : \quad otherwise \end{cases} \tag{6.2}$$

This solution constructs suffix tree in $O(n^2)$ time, for string of length n. It totally inserts n suffixes to the tree, and each insertion takes linear time proportion to the length of the suffix. The efficiency isn't good enough.

In this chapter, we firstly explain a fast on-line suffix trie construction solution by using suffix link concept. Because trie isn't space efficient, we next introduce a linear time on-line suffix tree construction algorithm found by Ukkonen. and show how to solve some interesting string manipulation problems with suffix tree.

6.2 Suffix trie

Just likes the relationship between trie and Patricia, Suffix trie has much simpler structure than suffix tree. Figure 6.3 shows the suffix trie for 'banana'.

Compare with figure 6.1, we can find the difference between suffix tree and suffix trie. Instead of representing a word, every edge in suffix trie only represents a character. Thus suffix trie needs much more spaces. If we pack all nodes which have only one child, the suffix trie is turned into a suffix tree.

We can reuse the trie definition for suffix tree. Each node is bound to a character, and contains multiple sub trees as children. A child can be referred from the bounded character.

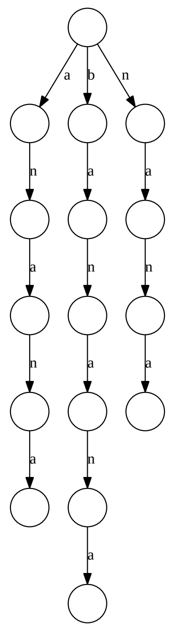


Figure 6.3: Suffix trie for 'banana'

6.2.1 Node transfer and suffix link

For string S of length n, define $S_i = s_1 s_2 ... s_i$. It is the prefix contains the first i characters.

In suffix trie, each node represents a suffix string. for example in figure 6.4, node X represents suffix 'a', by adding character 'c', node X transfers to Y which represents suffix 'ac'. We say node X transfer to Y with the edge of character 'c'[1].

$Y \leftarrow \textsc{Children}(X)[c]$

We also say that node X has a 'c'-child Y. Below Python expression reflects this concept.

```
y = x.children[c]
```

If node A in a suffix trie represents suffix $s_i s_{i+1} ... s_n$, and node B represents suffix $s_{i+1} s_{i+2} ... s_n$, we say node B represents *the suffix* of node A. We can create a link from A to B. This link is defined as *the suffix link* of node A[1]. Suffix link is drawn in dotted style. In figure 6.4, the suffix link of node A points to node B, and the suffix link of node B points to node C.

Suffix link is valid for all nodes except the root. We can add a suffix link field to the trie definition. Below Python example code shows this update.

```
class STrie:
    def __init__(self, suffix=None):
        self.children = {}
        self.suffix = suffix
```

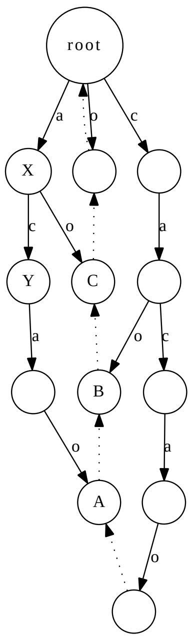


Figure 6.4: Suffix trie for string “cacao”. Node $X \leftarrow$ “a”, node $Y \leftarrow$ “ac”, X transfers to Y with character 'c'

suffix string
$s_1 s_2 s_3 ... s_i$
$s_2 s_3 ... s_i$
...
$s_{i-1} s_i$
s_i
“”

Table 6.1: suffixes for S_i

6.2.2 On-line construction

For string S, Suppose we have constructed suffix trie for the i-th prefix $S_i = s_1 s_2 ... s_i$. Denote it as $SuffixTrie(S_i)$. Let's consider how to obtain $SuffixTrie(S_{i+1})$ from $SuffixTrie(S_i)$.

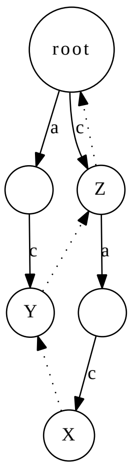
If list all suffixes corresponding to $SuffixTrie(S_i)$, from the longest (which is S_i) to the shortest (which is empty), we can get table 6.1. There are total $i + 1$ suffixes.

One solution is to append the character s_{i+1} to every suffix in this table, then add another empty string. This idea can be realized by adding a new child for every node in the trie, and binding all these new child with edge of character s_{i+1}.

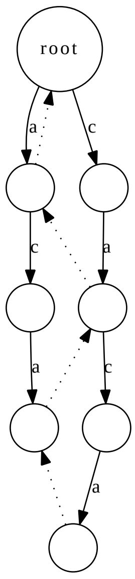
However, some nodes in $SuffixTrie(S_i)$ may have the s_{i+1}-child already. For example, in figure 6.5, node X and Y are corresponding for suffix 'cac' and 'ac' respectively. They don't have the 'a'-child. But node Z, which represents suffix 'c' has the 'a'-child already.

Algorithm 1 Update $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$, initial version.

```
1: for $\forall T \in SuffixTrie(S_i)$ do
2:     CHILDREN($T$)[$s_{i+1}$] ← CREATE-EMPTY-NODE
```



(a) Suffix trie for string “cac”.



(b) Suffix trie for string “caca”.

Figure 6.5: Suffix Trie of “cac” and “caca”

When append s_{i+1} to $SuffixTrie(S_i)$. In this example s_{i+1} is character 'a'. We need create new nodes for X and Y, but we needn't do this for Z.

If check the nodes one by one according to table 6.1, we can stop immediately when meet a node which has the s_{i+1}-child. This is because if node X in $SuffixTrie(S_i)$ has the s_{i+1}-child, according to the definition of suffix link, any suffix nodes X' of X in $SuffixTrie(S_i)$ must also have the s_{i+1}-child. In other words, let $c = s_{i+1}$, if wc is a sub-string of S_i, then every suffix of wc is also a sub-string of S_i [1]. The only exception is the root, which represents for empty string "".

According to this fact, we can refine the algorithm 1 to the following.

Algorithm 2 Update $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$, second version.

```
1: for each $T \in SuffixTrie(S_i)$ in descending order of suffix length do
2:     if CHILDREN($T$)[$s_{i+1}$] = NIL then
3:         CHILDREN($T$)[$s_{i+1}$] ← CREATE-EMPTY-NODE
4:     else
5:         break
```

The next question is how to iterate all nodes in descending order of the suffix length? Define the *top* of a suffix trie as the deepest leaf node. This definition ensures the top represents the longest suffix. Along the suffix link from the top to the next node, the length of the suffix decrease by one. This fact tells us that We can traverse the suffix tree from the top to the root by using the suffix links. And the order of such traversing is exactly what we want. Finally, there is a special suffix trie for empty string $SuffixTrie(NIL)$, We define the top equals to the root in this case.

```
function INSERT($top, c$)
    if $top$ = NIL then                              ▷ The trie is empty
        $top$ ← CREATE-EMPTY-NODE
    $T$ ← $top$
    $T'$ ← CREATE-EMPTY-NODE                         ▷ dummy init value
    while $T \neq$ NIL $\wedge$ CHILDREN($T$)[$c$] = NIL do
        CHILDREN($T$)[$c$] ← CREATE-EMPTY-NODE
        SUFFIX-LINK($T'$) ← CHILDREN($T$)[$c$]
        $T'$ ← CHILDREN($T$)[$c$]
        $T$ ← SUFFIX-LINK($T$)
    if $T \neq$ NIL then
        SUFFIX-LINK($T'$) ← CHILDREN($T$)[$c$]
    return CHILDREN($top$)[$c$]                       ▷ returns the new top
```

Function INSERT, updates $SuffixTrie(S_i)$ to $SuffixTrie(S_{i+1})$. It takes two arguments, one is the top of $SuffixTrie(S_i)$, the other is s_{i+1} character. If the top is NIL, it means the tree is empty, so there is no root. The algorithm creates a root node in this case. A sentinel empty node T' is created. It keeps tracking the previous created new node. In the main loop, the algorithm checks every node one by one along the suffix link. If the node hasn't the s_{i+1}-child, it then creates a new node, and binds the edge to character s_{i+1}. The algorithm repeatedly goes up along the suffix link until either arrives at the root, or find a node which has the s_{i+1}-child already. After the loop, if the node isn't empty, it means we stop at a node which has the s_{i+1}-child. The last suffix link then points to that child. Finally, the new top position is returned, so that we can further insert other characters to the suffix trie.

For a given string S, the suffix trie can be built by repeatedly calling INSERT function.

```
1: function SUFFIX-TRIE($S$)
2:     $t \leftarrow$ NIL
3:     for $i \leftarrow 1$ to $|S|$ do
4:         $t \leftarrow$ INSERT($t, s_i$)
5:     return $t$
```

This algorithm returns the top of the suffix trie, but not the root. In order to access the root, we can traverse along the suffix link.

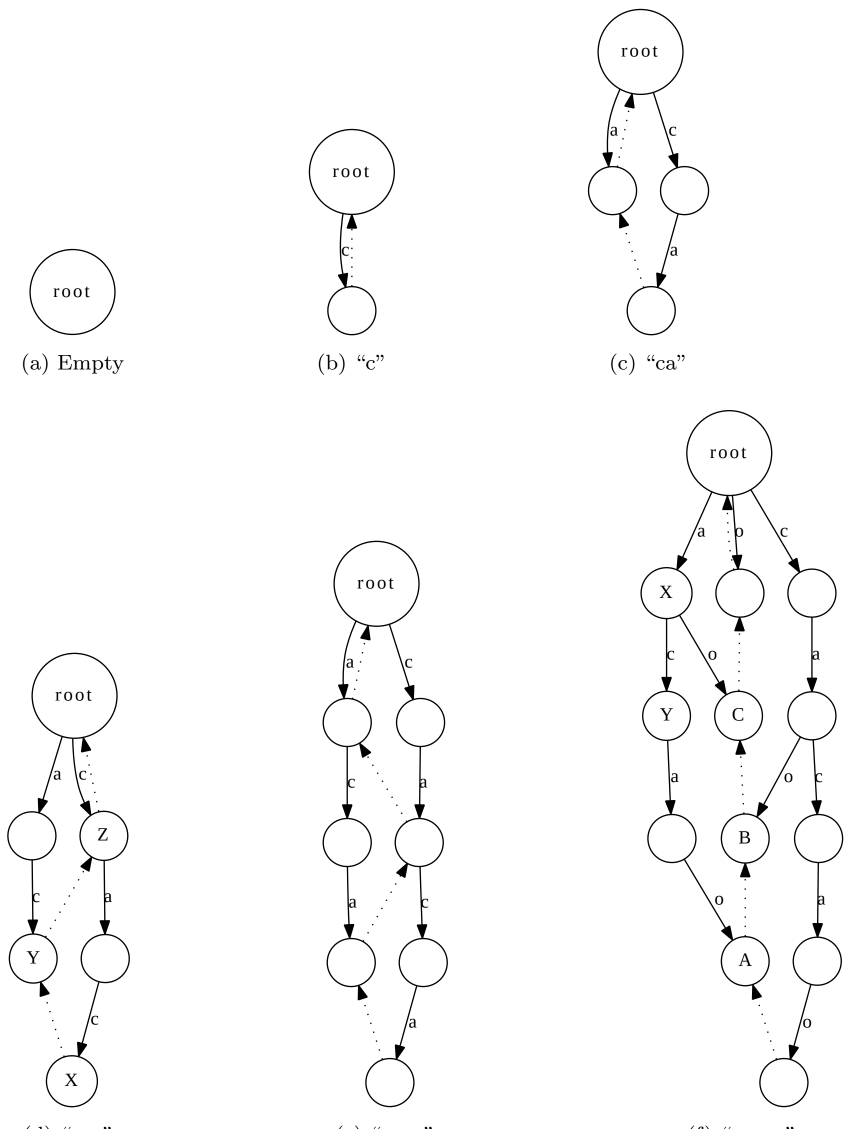
```
1: function ROOT($T$)
2:     while SUFFIX-LINK($T$) $\neq$ NIL do
3:         $T \leftarrow$ SUFFIX-LINK($T$)
4:     return $T$
```

Figure 6.6 shows the steps when construct suffix trie for “cacao”. Only the last layer of suffix links are shown.

For INSERT algorithm, the computation time is proportion to the size of suffix trie. In the worse case, the suffix trie is built in $O(n^2)$ time, where $n = |S|$. One example is $S = a^n b^n$, that there are n characters of a and n characters of b.

The following example Python program implements the suffix trie construction algorithm.

```python
def suffix_trie(str):
    t = None
    for c in str:
        t = insert(t, c)
    return root(t)

def insert(top, c):
    if top is None:
        top=STrie()
    node = top
    new_node = STrie() #dummy init value
    while (node is not None) and (c not in node.children):
        new_node.suffix = node.children[c] = STrie(node)
        new_node = node.children[c]
        node = node.suffix
    if node is not None:
        new_node.suffix = node.children[c]
    return top.children[c] #update top

def root(node):
    while node.suffix is not None:
        node = node.suffix
    return node
```



(a) Empty (b) "c" (c) "ca"

(d) "cac" (e) "caca" (f) "cacao"

Figure 6.6: Construct suffix trie for "cacao". There are 6 steps. Only the last layer of suffix links are shown in dotted arrow.

6.3 Suffix Tree

Suffix trie isn't space efficient, and the construction time is quadratic. If don't care about the speed, we can compress the suffix trie to suffix tree[6]. Ukkonen found a linear time on-line suffix tree construction algorithm in 1995.

6.3.1 On-line construction

Active point and end point

The suffix trie construction algorithm shows very important fact about what happens when $SuffixTrie(S_i)$ updates to $SuffixTrie(S_{i+1})$. Let's review the last two steps in figure 6.6.

There are two different updates.

1. All leaves are appended with a new node for s_{i+1};
2. Some non-leaf nodes are branched out with a new node for s_{i+1}.

The first type of update is trivial, because for all new coming characters, we need do this work anyway. Ukkonen defines leaf as the 'open' node.

The second type of update is important. We need figure out which internal nodes need branch out. We only focus on these nodes and apply the update.

Ukkonen defines the path along the suffix links from the top to the end as 'boundary path'. Denote the nodes in boundary path as, $n_1, n_2, ..., n_j, ..., n_k$. These nodes start from the leaf node (the first one is the top position), suppose that after the j-th node, they are not leaves any longer, we need repeatedly branch out from this time point till the k-th node.

Ukkonen defines the first none-leaf node n_j as 'active point' and the last node n_k as 'end point'. The end point can be the root.

Reference pair

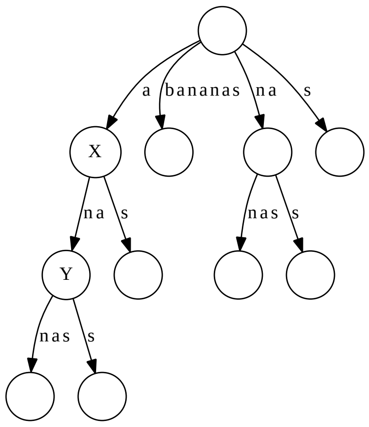


Figure 6.7: Suffix tree of "bananas". X transfer to Y with sub-string "na".

Figure 6.7 shows the suffix tree of English word "bananas". Node X represents suffix "a". By adding sub-string "na", node X transfers to node Y, which represents suffix “ana”. In other words, we can represent Y with a pair of node and sub-string, like (X, w), where $w =$ “na”. Ukkonen defines such kind of pair as *reference pair*. Not only the explicit node, but also the implicit position in suffix tree can be represented with reference pair. For example, (X, “n”) represents to a position which is not an explicit node. By using reference pair, we can represent every position in a suffix tree.

In order to save spaces, for string S, all sub-strings can be represented as a pair of index (l, r), where l is the left index and r is the right index of the character for the sub-string. For instance, if $S =$ “bananas”, and the index starts from 1, sub-string “na” can be represented with pair $(3, 4)$. As the result, there is only one copy of the complete string, and all positions in the suffix tree is defined as $(node, (l, r))$. This is the final form of reference pair.

With reference pair, node transfer for suffix tree can be defined as the following.

$\textsc{Children}(X)[s_l] \leftarrow ((l, r), Y) \iff Y \leftarrow (X, (l, r))$

If character $s_l = c$, we say that node X has a c-child, This child is Y. Y can be transferred from X with sub string (l, r) Each node can have at most one c-child.

canonical reference pair

It’s obvious that the one position in a suffix tree may have multiple reference pairs. For example, node Y in Figure 6.7 can be either denoted as $(X, (3, 4))$ or $(root, (2, 4))$. If we define empty string $\epsilon = (i, i - 1)$, Y can also be represented as (Y, ϵ).

The canonical reference pair is the one which has the closest node to the position. Specially, in case the position is an explicit node, the canonical reference pair is $(node, \epsilon)$, so (Y, ϵ) is the canonical reference pair of node Y.

Below algorithm converts a reference pair $(node, (l, r))$ to the canonical reference pair $(node', (l', r))$. Note that since r doesn’t change, the algorithm can only return $(node', l')$ as the result.

Algorithm 3 Convert reference pair to canonical reference pair

```
 1: function CANONIZE($node, (l, r)$)
 2:     if $node$ = NIL then
 3:         if $(l, r) = \epsilon$ then
 4:             return ( NIL, $l$)
 5:         else
 6:             return CANONIZE($root, (l + 1, r)$)
 7:     while $l \leq r$ do                                  ▷ $(l, r) isn't empty$
 8:         $((l', r'), node') \leftarrow$ CHILDREN($node$)[$s_l$]
 9:         if $r - l \geq r' - l'$ then
10:             $l \leftarrow l + r' - l' + 1$          ▷ Remove $|(l', r')|$ chars from $(l, r)$
11:             $node \leftarrow node'$
12:         else
13:             break
14:     return $(node, l)$
```

If the passed in node parameter is NIL, it means a very special case. The function is called like the following.

CANONIZE(SUFFIX-LINK(*root*), (l, r))

Because the suffix link of root points to NIL, the result should be $(root, (l+1, r))$ if (l, r) is not ϵ. Otherwise, (NIL, ϵ) is returned to indicate a terminal position.

We explain this special case in detail in later sections.

The algorithm

In 6.3.1, we mentioned, all updating to leaves is trivial, because we only need append the new coming character to the leaf. With reference pair, it means, when update $SuffixTree(S_i)$ to $SuffixTree(S_{i+1})$, all reference pairs in form $(node, (l, i))$, are leaves. They will change to $(node, (l, i+1))$ next time. Ukkonen defines leaf in form $(node, (l, \infty))$, here ∞ means "open to grow". We can skip all leaves until the suffix tree is completely constructed. After that, we can change all ∞ to the length of the string.

So the main algorithm only cares about *positions* from the active point to the end point. However, how to find the active point and the end point?

When start suffix tree construction, there is only a root node. There aren't any branches or leaves. The active point should be $(root, \epsilon)$, or $(root, (1, 0))$ (the string index starts from 1).

For the end point, it is a position where we can finish updating $SuffixTree(S_i)$. According to the suffix trie algorithm, we know it should be a *position* which has the s_{i+1}-child already. Because a position in suffix trie may not be an explicit node in suffix tree, if $(node, (l, r))$ is the end point, there are two cases.

1. $(l, r) = \epsilon$. It means the node itself is the end point. This node has the s_{i+1}-child, which means CHILDREN(*node*)$[s_{i+1}] \neq$ NIL;

2. Otherwise, $l \leq r$, the end point is an implicit position. It must satisfy $s_{i+1} = s_{l'+|(l,r)|}$, where CHILDREN(*node*)$[s_l]$= $((l', r'), node')$, $|(l, r)|$ means the length of sub-string (l, r). It equals to $r - l + 1$. This is illustrated in figure 6.8. We can also say that $(node, (l, r))$ has a s_{i+1}-child implicitly.

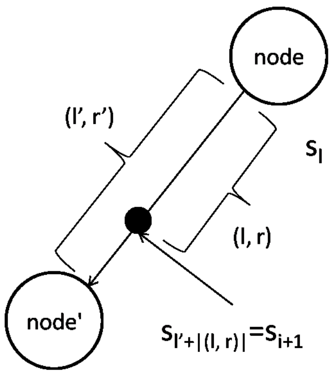


Figure 6.8: Implicit end point

Ukkonen finds a very important fact that if $(node, (l, i))$ is the end point of $SuffixTree(S_i)$, then $(node, (l, i+1))$ is the active point of $SuffixTree(S_{i+1})$.

This is because if $(node, (l, i))$ is the end point of $SuffixTree(S_i)$, it must have a s_{i+1}-child (either explicitly or implicitly). If this end point represents suffix $s_k s_{k+1} ... s_i$, it is the longest suffix in $SuffixTree(S_i)$ which satisfies $s_k s_{k+1} ... s_i s_{i+1}$ is a sub-string of S_i. Consider S_{i+1}, $s_k s_{k+1} ... s_i s_{i+1}$ must occur at least twice in S_{i+1}, so position $(node, (l, i+1))$ is the active point of $SuffixTree(S_{i+1})$. Figure 6.9 shows about this truth.

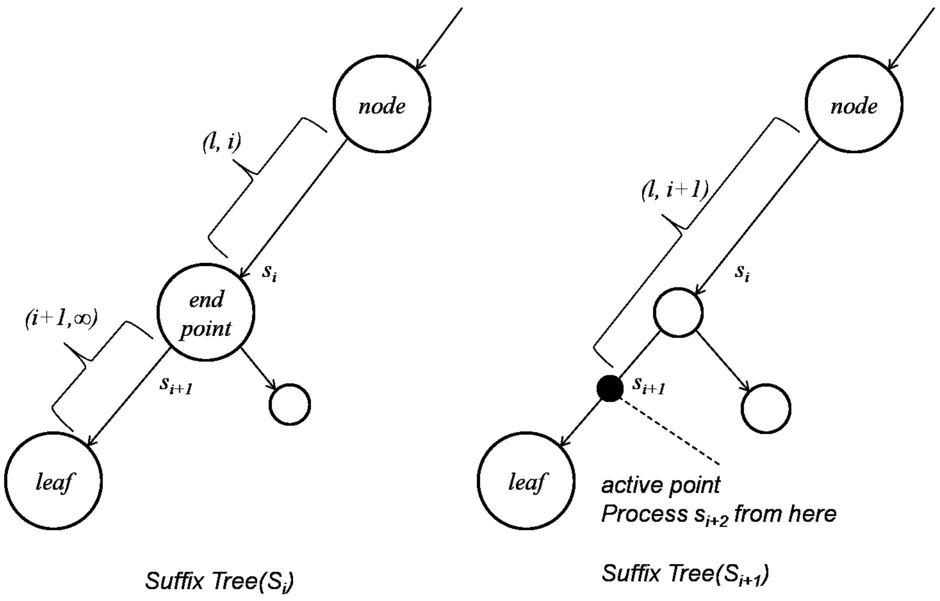


Figure 6.9: End point in $SuffixTree(S_i)$ and active point in $SuffixTree(S_{i+1})$.

Summarize the above facts, the algorithm of Ukkonen's on-line construction can be given as the following.

```
1: function UPDATE($node, (l, i)$)
2:     $prev \leftarrow$ CREATE-EMPTY-NODE                ▷ Initialized as sentinel
3:     loop                                  ▷ Traverse along the suffix links
4:         $(finish, node') \leftarrow$ END-POINT-BRANCH?($node, (l, i-1), s_i$)
5:         if $finish$ then
6:             break
7:         CHILDREN($node'$)[$s_i$] $\leftarrow$ (($i, \infty$), CREATE-EMPTY-NODE)
8:         SUFFIX-LINK($prev$) $\leftarrow node'$
9:         $prev \leftarrow node'$
10:        $(node, l)$ = CANONIZE(SUFFIX-LINK($node$), $(l, i-1)$)
11:    SUFFIX-LINK($prev$) $\leftarrow node$
12:    return $(node, l)$                                  ▷ The end point
```

This algorithm takes reference pair $(node, (l, i))$ as arguments, note that position $(node, (l, i-1)$ is the active point for $SuffixTree(S_{i-1})$. Then we start a loop, this loop goes along the suffix links until the current position $(node, (l, i-1))$ is the end point. Otherwise, function END-POINT-BRANCH?

returns a position, from where the new leaf branch out.

The END-POINT-BRANCH? algorithm is realized as below.

```
function END-POINT-BRANCH?($node, (l, r), c$)
    if $(l, r) = \epsilon$ then
        if $node$ = NIL then
            return (TRUE, $root$)
        else
            return (CHILDREN($node$)[$c$] = NIL, $node$)
    else
        $((l', r'), node') \leftarrow$ CHILDREN($node$)[$s_l$]
        $pos \leftarrow l' + |(l, r)|$
        if $s_{pos} = c$ then
            return (TRUE, $node$)
        else
            $p \leftarrow$ CREATE-EMPTY-NODE
            CHILDREN($node$)[$s_{l'}$] $\leftarrow ((l', pos - 1), p)$
            CHILDREN($p$)[$s_{pos}$] $\leftarrow ((pos, r'), node')$
            return (FALSE, $p$)
```

If the position is $(root, \epsilon)$, it means we have arrived at the root. It's definitely the end point, so that we can finish this round of updating. If the position is in form of $(node, \epsilon)$, it means the reference pair represents an explicit node, we can examine if this node has already the c-child, where $c = s_i$. If not, we need branch out a leaf.

Otherwise, the position $(node, (l, r))$ points to an implicit node. We need find the exact position next to it to see if there is a c-child. If yes, we meet an end point, the updating loop can be finished; else, we turn the position to an explicit node, and return it for further branching.

We can finalize the Ukkonen's algorithm as below.

```
1: function SUFFIX-TREE($S$)
2:     $root \leftarrow$ CREATE-EMPTY-NODE
3:     $node \leftarrow root, l \leftarrow 0$
4:     for $i \leftarrow 1$ to $|S|$ do
5:         $(node, l)$ = UPDATE($node, (l, i)$)
6:         $(node, l)$ = CANONIZE($node, (l, i)$)
7:     return $root$
```

Figure 6.10 shows the steps when constructing the suffix tree for string "cacao".

Note that we needn't set suffix link for leaf nodes, only branch nodes need suffix links.

The following example Python program implements Ukkonen's algorithm. First is the node definition.

```
class Node:
    def __init__(self, suffix=None):
        self.children = {} # 'c':(word, Node), where word = (l, r)
        self.suffix = suffix
```

Because there is only one copy of the complete string, all sub-strings are represent in $(left, right)$ pairs, and the leaf are open pairs as $(left, \infty)$. The suffix tree is defined like below.

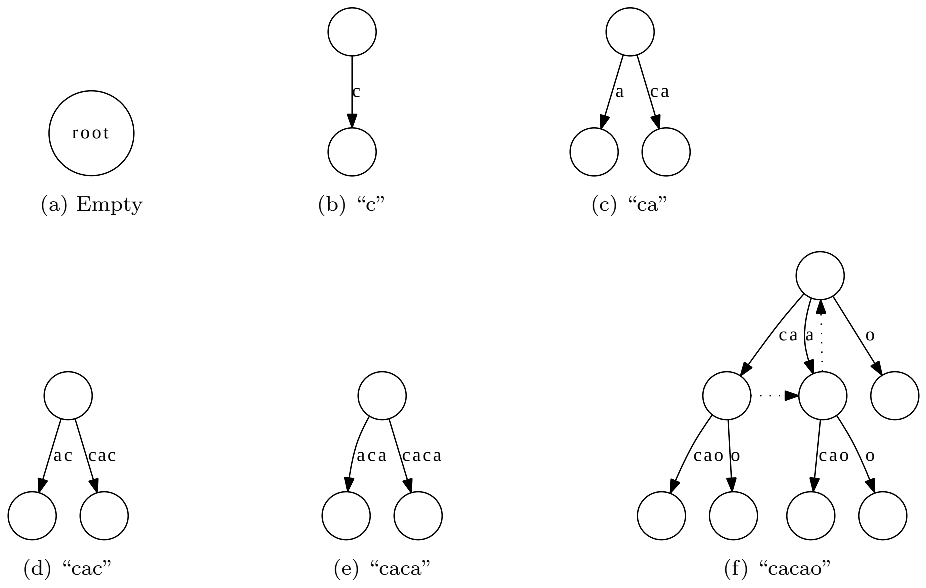


Figure 6.10: Construct suffix tree for "cacao". There are 6 steps. Only the last layer of suffix links are shown in dotted arrow.

```
class STree:
    def __init__(self, s):
        self.str = s
        self.infinity = len(s)+1000
        self.root = Node()
```

The infinity is defined as the length of the string plus a big number. Some auxiliary functions are defined.

```
def substr(str, str_ref):
    (l, r)=str_ref
    return str[l:r+1]

def length(str_ref):
    (l, r)=str_ref
    return r-l+1
```

The main entry for Ukkonen's algorithm is implemented as the following.

```
def suffix_tree(str):
    t = STree(str)
    node = t.root # init active point is (root, Empty)
    l = 0
    for i in range(len(str)):
        (node, l) = update(t, node, (l, i))
        (node, l) = canonize(t, node, (l, i))
    return t

def update(t, node, str_ref):
    (l, i) = str_ref
    c = t.str[i] # current char
    prev = Node() # dummy init
    while True:
        (finish, p) = branch(t, node, (l, i-1), c)
        if finish:
            break
        p.children[c]=((i, t.infinity), Node())
        prev.suffix = p
        prev = p
        (node, l) = canonize(t, node.suffix, (l, i-1))
    prev.suffix = node
    return (node, l)

def branch(t, node, str_ref, c):
    (l, r) = str_ref
    if length(str_ref)≤0: # (node, empty)
        if node is None: #_|_
            return (True, t.root)
        else:
            return ((c in node.children), node)
    else:
        ((l1, r1), node1) = node.children[t.str[l]]
        pos = l1+length(str_ref)
        if t.str[pos]==c:
            return (True, node)
        else:
            branch_node = Node()
            node.children[t.str[l1]]=((l1, pos-1), branch_node)
            branch_node.children[t.str[pos]] = ((pos, r1), node1)
            return (False, branch_node)

def canonize(t, node, str_ref):
    (l, r) = str_ref
    if node is None:
        if length(str_ref)≤0:
            return (None, l)
        else:
            return canonize(t, t.root, (l+1, r))
    while l≤r: # str_ref is not empty
        ((l1, r1), child) = node.children[t.str[l]]
        if r-l ≥ r1-l1:
            l += r1-l1+1
            node = child
        else:
            break
    return (node, l)
```

Functional suffix tree construction

Giegerich and Kurtz found Ukkonen's algorithm can be transformed to McCreight's algorithm[7]. The three suffix tree construction algorithms found by Weiner, McCreight, and Ukkonen are all bound to $O(n)$ time. Giegerich and Kurtz conjectured any sequential suffix tree construction method doesn't base on suffix links, active suffixes, etc., fails to meet the $O(n)$-criterion.

There is implementation in PLT/Scheme[10] based on Ukkonen's algorithm, However, it updates suffix links during the processing, which is not purely functional.

A lazy suffix tree construction method is discussed in [8]. And this method is contributed to Haskell Hackage by Bryan O'Sullivan. [9]. The method depends on the lazy evaluation property. The tree won't be constructed until it is traversed. However, it can't ensure the $O(n)$ performance if the programming environments or languages don't support lazy evaluation.

The following Haskell program defines the suffix tree. A suffix tree is either a leaf, or a branch containing multiple sub trees. Each sub tree is bound to a string.

```
data Tr = Lf | Br [(String, Tr)] deriving (Eq)
type EdgeFunc = [String]→(String, [String])
```

The edge function extracts a common prefix from a list of strings. The prefix returned by edge function may not be the longest one, empty string is also allowed. The exact behavior can be customized with different edge functions.

$$build(edge, X)$$

This defines a generic radix tree building function. It takes an edge function, and a set of strings. X can be all suffixes of a string, so that we get suffix trie or suffix tree. We'll also explain later that X can be all prefixes, which lead to normal prefix trie or Patricia.

Suppose all the strings are built from a character set Σ. When build the tree, if the string is empty, X only contains one empty string as well. The result tree is an empty leaf; Otherwise, we examine every character in Σ, group the strings in X with their initial characters, the apply the edge function to these groups.

$$build(edge, X) == \left\{ \begin{array}{lcl} leaf & : & X = \{\Phi\} \\ \begin{array}{ll} branch(\{(\{c\} \cup p, & build(edge, X'))| \\ & c \in \Sigma, \\ & G \in \{group(X, c)\}, \\ & (p, X') \in \{edge(G)\}\}) \end{array} & : & otherwise \end{array} \right. \tag{6.3}$$

The algorithm categorizes all suffixes by the first letter in several groups. It removes the first letter for each element in every group. For example, the suffixes {"acac", "cac", "ac", "c"} are categorized to groups {('a', ["cac", "c"]), ('c', ["ac", ""])}.

$$group(X, c) = \{C' | \{c_1\} \cup C' \in X, c_1 = c\} \tag{6.4}$$

Function *group* enumerates all suffixes in X, for each one, denote the first character as c_1, the rest characters as C'. If c_1 is same as the given character c, then C' is collected.

Below example Haskell program implements the generic radix tree building algorithm.

```
alpha = ['a'..'z']++['A'..'Z']

lazyTree::EdgeFunc → [String] → Tr
lazyTree edge = build where
    build [[]] = Lf
    build ss = Br [(a:prefix, build ss') |
                      a←alpha,
                      xs@(x:_) ←[[cs | c:cs←ss, c==a]],
                      (prefix, ss')←[edge xs]]
```

Different edge functions produce different radix trees. Since edge function extracts common prefix from a set of strings. The simplest one constantly uses the empty string as the common prefix. This edge function builds a trie.

$$edgeTrie(X) = (\Phi, X) \tag{6.5}$$

We can also realize an edge function, that extracts the longest common prefix. Such edge function builds a Patricia. Denote the strings as $X = \{x_1, x_2, ..., x_n\}$, for the each string x_i, let the initial character be c_i, and the rest characters in x_i as W_i. If there is only one string in X, the longest common prefix is definitely this string; If there are two strings start with different initial characters, the longest common prefix is empty; Otherwise,it means all the strings share the same initial character. This character definitely belongs to the longest common prefix. We can remove it from all strings, and recursively call the edge function.

$$edgeTree(X) = \begin{cases} (x_1, \{\Phi\}) & : X = \{x_1\} \\ (\Phi, X) & : |X| > 1, \exists x_i \in X, c_i \neq c_1 \\ (\{c_1\} \cup p, Y) & : (p, Y) = edgeTree(\{W_i | x_i \in X\}) \end{cases} \tag{6.6}$$

Here are some examples for $edgeTree$ function.

$$edgeTree(\{\text{“an”}, \text{“another”}, \text{“and”}\}) = (\text{“an”}, \{\text{“”}, \text{“other”}, \text{“d”}\})$$
$$edgeTree(\{\text{“bool”}, \text{“foo}, \text{“bar”}\}) = (\text{“”}, \{\text{“bool”}, \text{“fool”}, \text{“bar”}\})$$

The following example Haskell program implements this edge function.

```
edgeTree::EdgeFunc
edgeTree [s] = (s, [[]])
edgeTree awss@((a:w):ss) | null [c|c:_←ss, a/=c] = (a:prefix, ss')
                         | otherwise              = ("", awss)
                         where (prefix, ss') = edgeTree (w:[u| _:u←ss])
edgeTree ss = ("", ss)
```

For any given string, we can build suffix trie and suffix tree by feeding suffixes to these two edge functions.

$$suffixTrie(S) = build(edgeTrie, suffixes(S)) \tag{6.7}$$

$$suffixTree(S) = build(edgeTree, suffixes(S)) \tag{6.8}$$

Because the $build(edge, X)$ is generic, it can be used to build other radix trees, such as the normal prefix trie and Patricia.

$$trie(S) = build(edgeTrie, prefixes(S)) \tag{6.9}$$

$$tree(S) = build(edgeTree, prefixes(S)) \tag{6.10}$$

6.4 Suffix tree applications

Suffix tree can help to solve many string and DNA pattern manipulation problems particularly fast.

6.4.1 String/Pattern searching

There a plenty of string searching algorithms, such as the famous KMP(Knuth-Morris-Pratt algorithm is introduced in the chapter of search) algorithm. Suffix tree can perform at the same level as KMP[11]. the string searching in bound to $O(m)$ time, where m is the length of the sub-string to be search. However, $O(n)$ time is required to build the suffix tree in advance, where n is the length of the text[12].

Not only sub-string searching, but also pattern matching, including regular expression matching can be solved with suffix tree. Ukkonen summarizes this kind of problems as sub-string motifs: *For a string S, $SuffixTree(S)$ gives complete occurrence counts of all sub-string motifs of S in $O(n)$ time, although S may have $O(n^2)$ sub-strings.*

Find the number of sub-string occurrence

Every internal node in $SuffixTree(S)$ is corresponding to a sub-string occurs multiple times in S. If this sub-string occurs k times in S, then there are total k sub-trees under this node[13].

```
1: function LOOKUP-PATTERN($T, s$)
2:     loop
3:         $match \leftarrow$ FALSE
4:         for $\forall(s_i, T_i) \in$ VALUES(CHILDREN($T$)) do
5:             if $s \sqsubset s_i$ then
6:                 return MAX($|$CHILDREN($T_i$)$|$, 1)
7:             else if $s_i \sqsubset s$ then
8:                 $match \leftarrow$ TRUE
9:                 $T \leftarrow T_i$
10:                $s \leftarrow s - s_i$
11:                break
12:        if $\neg match$ then
13:            return 0
```

When look up a sub-string pattern s in text w, we build the suffix tree T from the text. Start from the root, we iterate all children. For every string reference pair s_i and sub-tree T_i, we check if the s is prefix of s_i. If yes, the number of sub-trees in T_i is returned as the result. There is a special case that T_i is a leaf without any children. We need return 1 but not zero. This is why we use the maximum function. Otherwise, if s_i is prefix of s, then we remove s_i part from s, and recursively look up in T_i.

The following Python program implements this algorithm.

```
def lookup_pattern(t, s):
    node = t.root
    while True:
        match = False
        for _, (str_ref, tr) in node.children.items():
            edge = substr(t, str_ref)
            if string.find(edge, s)==0: #s 'isPrefixOf' edge
                return max(len(tr.children), 1)
            elif string.find(s, edge)==0: #edge 'isPrefixOf' s
                match = True
                node = tr
                s = s[len(edge):]
                break
        if not match:
            return 0
    return 0 # not found
```

This algorithm can also be realized in recursive way. For the non-leaf suffix tree T, denote the children as $C = \{(s_1, T_1), (s_2, T_2), ...\}$. We search the sub string among the children.

$$lookup_{pattern}(T, s) = find(C, s) \tag{6.11}$$

If children C is empty, it means the sub string doesn't occurs at all. Otherwise, we examine the first pair (s_1, T_1), if s is prefix of s_1, then the number of sub-trees in T_1 is the result. If s_1 is prefix of s, we remove s_1 from s, and recursively look up it in T_1; otherwise, we go on to examine the rest children denoted as C'.

$$find(C, s) = \begin{cases} 0 & : C = \Phi \\ max(1, |C_1|) & : s \sqsubset s_1 \\ lookup_{pattern}(T_1, s - s_1) & : s_1 \sqsubset s \\ find(C', s) & : otherwise \end{cases} \tag{6.12}$$

The following Haskell example code implements this algorithm.

```
lookupPattern (Br lst) ptn = find lst where
    find [] = 0
    find ((s, t):xs)
         | ptn 'isPrefixOf' s = numberOfBranch t
         | s 'isPrefixOf' ptn = lookupPattern t (drop (length s) ptn)
         | otherwise = find xs
    numberOfBranch (Br ys) = length ys
    numberOfBranch _ = 1

findPattern s ptn = lookupPattern (suffixTree $ s++"$") ptn
```

We always append special terminator to the string (the '$' in above program), so that there won't be any suffix becomes the prefix of the other[3].

Suffix tree also supports searching pattern like "a**n", we skip it here. Readers can refer to [13] and [14] for details.

6.4.2 Find the longest repeated sub-string

After adding a special terminator character to string S, The longest repeated sub-string can be found by searching the deepest branches in suffix tree.

Consider the example suffix tree shown in figure 6.11

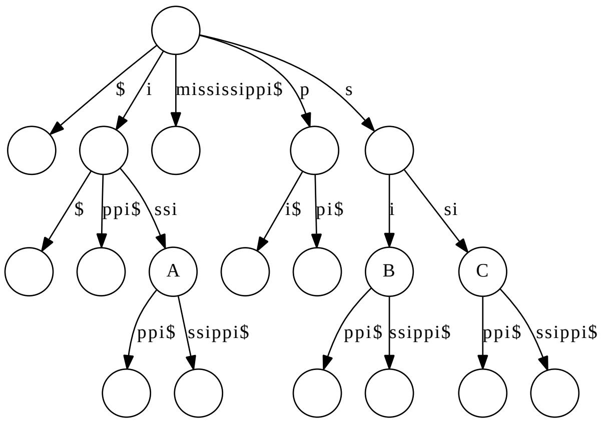


Figure 6.11: The suffix tree for 'mississippi$'

There are three branch nodes, A, B, and C with depth 3. However, A represents the longest repeated sub-string "issi". B and C represent for "si", "ssi", they are shorter than A.

This example tells us that the "depth" of the branch node should be measured by the number of characters traversed from the root. But not the number of explicit branch nodes.

To find the longest repeated sub-string, we can perform BFS in the suffix tree.

```
1: function LONGEST-REPEATED-SUBSTRING($T$)
2:     $Q \leftarrow$ (NIL, ROOT($T$))
3:     $R \leftarrow$ NIL
4:     while $Q$ is not empty do
5:         $(s, T) \leftarrow$ POP($Q$)
6:         for each $((l, r), T') \in$ CHILDREN($T$) do
7:             if $T'$ is not leaf then
8:                 $s' \leftarrow$ CONCATENATE($s, (l, r)$)
9:                 PUSH($Q, (s', T')$)
10:                $R \leftarrow$ UPDATE($R, s'$)
11:    return $R$
```

This algorithm initializes a queue with a pair of an empty string and the root. Then it repeatedly examine the candidate in the queue.

For each node, the algorithm examines each children one by one. If it is a branch node, the child is pushed back to the queue for further search. And the sub-string represented by this child will be treated as a candidate of the longest repeated sub-string.

Function UPDATE(R, s') updates the longest repeated sub-string candidates. If multiple candidates have the same length, they are all kept in a result list.

```
1: function UPDATE($L, s$)
2:     if $L =$ NIL $\vee |l_1| < |s|$ then
3:         return $l \leftarrow \{s\}$
4:     if $|l_1| = |s|$ then
5:         return APPEND($L, s$)
6:     return $L$
```

The above algorithm can be implemented in Python as the following example program.

```
def lrs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s+t.substr(str_ref)
                queue.append((s1, tr))
                res = update_max(res, s1)
    return res

def update_max(lst, x):
    if lst ==[] or len(lst[0]) < len(x):
        return [x]
    if len(lst[0]) == len(x):
        return lst + [x]
    return lst
```

Searching the deepest branch can also be realized recursively. If the tree is just a leaf node, empty string is returned, else the algorithm tries to find the longest repeated sub-string from the children.

$$LRS(T) = \begin{cases} \Phi & : leaf(T) \\ longest(\{s_i \cup LRS(T_i) | (s_i, T_i) \in C, \neg leaf(T_i)\}) & : otherwise \end{cases} \tag{6.13}$$

The following Haskell example program implements the longest repeated sub-string algorithm.

```
isLeaf Lf = True
isLeaf _ = False

lrs'::Tr->String
lrs' Lf = ""
lrs' (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = ""
    find ((s, t):xs) = maximumBy (compare `on` length) [s++(lrs' t), find xs]
```

6.4.3 Find the longest common sub-string

The longest common sub-string, can also be quickly found with suffix tree. The solution is to build a generalized suffix tree. If the two strings are denoted as txt_1 and txt_2, a generalized suffix tree is $SuffixTree(txt_1\$_1 txt_2\$_2)$. Where

$\$_1$ is a special terminator character for txt_1, and $\$_2 \neq \$_1$ is another special terminator character for txt_2.

The longest common sub-string is indicated by the deepest branch node, with two forks corresponding to both "...$\$_1$..." and "...$\$_2$"(no $\$_1$). The definition of the *deepest* node is as same as the one for the longest repeated sub-string, it is the number of characters traversed from root.

If a node has "...$\$_1$..." under it, the node must represent a sub-string of txt_1, as $\$_1$ is the terminator of txt_1. On the other hand, since it also has "...$\$_2$" (without $\$_1$), this node must represent a sub-string of txt_2 too. Because it's the deepest one satisfied this criteria, so the node represents the longest common sub-string.

Again, we can use BFS (bread first search) to find the longest common sub-string.

```
1: function LONGEST-COMMON-SUBSTRING($T$)
2:     $Q \leftarrow$ (NIL, ROOT($T$))
3:     $R \leftarrow$ NIL
4:     while $Q$ is not empty do
5:         $(s, T) \leftarrow$ POP($Q$)
6:         if MATCH-FORK($T$) then
7:             $R \leftarrow$ UPDATE($R, s$)
8:         for each $((l, r), T') \in$ CHILDREN($T$) do
9:             if $T'$ is not leaf then
10:                $s' \leftarrow$ CONCATENATE($s, (l, r)$)
11:                PUSH($Q, (s', T')$)
12:    return $R$
```

Most part is as same as the the longest repeated sub-sting searching algorithm. The function Match-Fork checks if the children satisfy the common sub-string criteria.

```
1: function MATCH-FORK($T$)
2:     if $|$ CHILDREN($T$) $| = 2$ then
3:         $\{(s_1, T_1), (s_2, T_2)\} \leftarrow$ CHILDREN($T$)
4:         return $T_1$ is leaf $\wedge T_2$ is leaf $\wedge$ XOR($\$_1 \in s_1, \$_1 \in s_2$)
5:     return FALSE
```

In this function, it checks if the two children are both leaf. One contains $\$_2$, while the other doesn't. This is because if one child is a leaf, it always contains $\$_1$ according to the definition of suffix tree.

The following Python program implement the longest common sub-string program.

```
def lcs(t):
    queue = [("", t.root)]
    res = []
    while len(queue)>0:
        (s, node) = queue.pop(0)
        if match_fork(t, node):
            res = update_max(res, s)
        for _, (str_ref, tr) in node.children.items():
            if len(tr.children)>0:
                s1 = s + t.substr(str_ref)
                queue.append((s1, tr))
    return res

def is_leaf(node):
    return node.children=={}

def match_fork(t, node):
    if len(node.children)==2:
        [(_, (str_ref1, tr1)), (_, (str_ref2, tr2))]=node.children.items()
        return is_leaf(tr1) and is_leaf(tr2) and
            (t.substr(str_ref1).find('#')!=-1) !=
            (t.substr(str_ref2).find('#')!=-1)
    return False
```

The longest common sub-string finding algorithm can also be realized recursively. If the suffix tree T is a leaf, the result is empty; Otherwise, we examine all children in T. For those satisfy the matching criteria, the sub-string are collected as candidates; for those don't matching, we recursively search the common sub-string among the children. The longest candidate is selected as the final result.

$$LCS(T) = \begin{cases} \Phi & : leaf(T) \\ longest(\begin{array}{l} \{s_i \mid (s_i, T_i) \in C, match(T_i)\} \cup \\ \{s_i \cup LCS(T_i) \mid (s_i, T_i) \in C, \neg match(T_i)\}) \end{array} & : otherwise \end{cases} \tag{6.14}$$

The following Haskell example program implements the longest common sub-string algorithm.

```
lcs Lf = []
lcs (Br lst) = find $ filter (not ∘ isLeaf ∘ snd) lst where
    find [] = []
    find ((s, t):xs) = maxBy (compare `on` length)
                       (if match t
                        then s:(find xs)
                        else  (map (s++) (lcs t)) ++ (find xs))

match (Br [(s1, Lf), (s2, Lf)]) = ("#" `isInfixOf` s1) /= ("#" `isInfixOf` s2)
match _ = False
```

6.4.4 Find the longest palindrome

A palindrome is a string, S, such that $S = reverse(S)$ For example, "level", "rotator", "civic" are all palindrome.

The longest palindrome in a string $s_1 s_2 ... s_n$ can be found in $O(n)$ time with suffix tree. The solution can be benefit from the longest common sub-string algorithm.

For string S, if sub-string w is a palindrome, then it must be sub-string of $reverse(S)$ too. for instance, "issi" is a palindrome, it is a sub-string of "mississippi". When reverse to "ippississim", "issi" is also a sub-string.

Based on this fact, we can find the longest palindrome by searching the longest common sub-string for S and $reverse(S)$.

$$palindrome_m(S) = LCS(suffixTree(S \cup reverse(S))) \tag{6.15}$$

The following Haskell example program finds the longest palindrome.

```
longestPalindromes s = lcs $ suffixTree (s++"#"++(reverse s)++"$")
```

6.4.5 Others

Suffix tree can also be used for data compression, such as Burrows-Wheeler transform, LZW compression (LZSS) etc. [3]

6.5 Notes and short summary

Suffix Tree was first introduced by Weiner in 1973 [?]. In 1976, McCreight greatly simplified the construction algorithm. McCreight constructs the suffix tree from right to left. In 1995, Ukkonen gave the first on-line construction algorithms from left to right. All the three algorithms are linear time ($O(n)$). And some research shows the relationship among these 3 algorithms. [7]

Bibliography

[1] Esko Ukkonen. "On-line construction of suffix trees". Algorithmica 14 (3): 249–260. doi:10.1007/BF01206331. http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf

[2] Weiner, P. "Linear pattern matching algorithms", 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1C11, doi:10.1109/SWAT.1973.13

[3] Suffix Tree, Wikipedia. http://en.wikipedia.org/wiki/Suffix_tree

[4] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[5] Trie, Wikipedia. http://en.wikipedia.org/wiki/Trie

[6] Suffix Tree (Java). http://en.literateprograms.org/Suffix_tree_(Java)

[7] Robert Giegerich and Stefan Kurtz. "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction". Science of Computer Programming 25(2-3):187-218, 1995. http://citeseer.ist.psu.edu/giegerich95comparison.html

[8] Robert Giegerich and Stefan Kurtz. "A Comparison of Imperative and Purely Functional Suffix Tree Constructions". Algorithmica 19 (3): 331–353. doi:10.1007/PL00009177. www.zbh.uni-hamburg.de/pubs/pdf/GieKur1997.pdf

[9] Bryan O'Sullivan. "suffixtree: Efficient, lazy suffix tree implementation". http://hackage.haskell.org/package/suffixtree

[10] Danny. http://hkn.eecs.berkeley.edu/ dyoo/plt/suffixtree/

[11] Zhang Shaojie. "Lecture of Suffix Trees". http://www.cs.ucf.edu/ shzhang/Combio09/lec3.pdf

[12] Lloyd Allison. "Suffix Trees". http://www.allisons.org/ll/AlgDS/Tree/Suffix/

[13] Esko Ukkonen. "Suffix tree and suffix array techniques for pattern analysis in strings". http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

[14] Esko Ukkonen "Approximate string-matching over suffix trees". Proc. CPM 93. Lecture Notes in Computer Science 684, pp. 228-242, Springer 1993. http://www.cs.helsinki.fi/u/ukkonen/cpm931.ps

Chapter 7

B-Trees

7.1 Introduction

B-Tree is important data structure. It is widely used in modern file systems. Some are implemented based on B+ tree, which is extended from B-tree. B-tree is also widely used in database systems.

Some textbooks introduce B-tree with the the problem of how to access a large block of data on magnetic disks or secondary storage devices[2]. It is also helpful to understand B-tree as a generalization of balanced binary search tree[2].

Refer to the Figure 7.1, It is easy to find the difference and similarity of B-tree regarding to binary search tree.

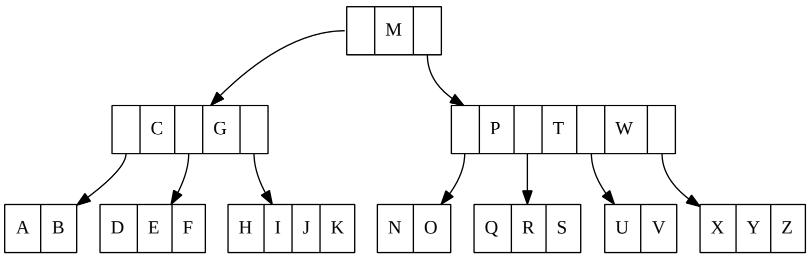


Figure 7.1: Example B-Tree

Remind the definition of binary search tree. A binary search tree is

- either an empty node;
- or a node contains 3 parts, a value, a left child and a right child. Both children are also binary search trees.

The binary search tree satisfies the constraint that.

- all the values on the left child are not greater than the value of of this node;
- the value of this node is not greater than any values on the right child.

For non-empty binary tree (L, k, R), where L, R and k are the left, right children, and the key. Function $Key(T)$ accesses the key of tree T. The constraint can be represented as the following.

$$\forall x \in L, \forall y \in R \Rightarrow Key(x) \leq k \leq Key(y) \tag{7.1}$$

If we extend this definition to allow multiple keys and children, we get the B-tree definition.

A B-tree

- is either empty;
- or contains n keys, and $n+1$ children, each child is also a B-Tree, we denote these keys and children as $k_1, k_2, ..., k_n$ and $c_1, c_2, ..., c_n, c_{n+1}$.

Figure 7.2 illustrates a B-Tree node.

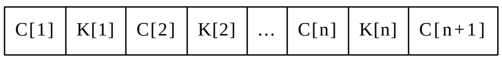


Figure 7.2: A B-Tree node

The keys and children in a node satisfy the following order constraints.

- Keys are stored in non-decreasing order. that $k_1 \leq k_2 \leq ... \leq k_n$;
- for each k_i, all elements stored in child c_i are not greater than k_i, while k_i is not greater than any values stored in child c_{i+1}.

The constraints can be represented as in equation (7.2) as well.

$$\forall x_i \in c_i, i = 0, 1, ..., n, \Rightarrow x_1 \leq k_1 \leq x_2 \leq k_2 \leq ... \leq x_n \leq k_n \leq x_{n+1} \tag{7.2}$$

Finally, after adding some constraints to make the tree balanced, we get the complete B-tree definition.

- All leaves have the same depth;
- We define integral number, t, as the *minimum degree* of B-tree;
  - each node can have at most $2t-1$ keys;
  - each node can have at least $t-1$ keys, except the root;

Consider a B-tree holds n keys. The minimum degree $t \geq 2$. The height is h. All the nodes have at least $t-1$ keys except the root. The root contains at least 1 key. There are at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, ..., finally, there are at least $2t^{h-1}$ nodes at depth h. Times all nodes with $t-1$ except for root, the total number of keys satisfies the following inequality.

$$\begin{aligned} n & \geq 1 + (t-1)(2 + 2t + 2t^2 + ... + 2t^{h-1}) \\ & = 1 + 2(t-1)\sum_{k=0}^{h-1} t^k \\ & = 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ & = 2t^h - 1 \end{aligned} \tag{7.3}$$

Thus we have the inequality between the height and the number of keys.

$$h \leq \log_t \frac{n+1}{2} \tag{7.4}$$

This is the reason why B-tree is balanced. The simplest B-tree is so called 2-3-4 tree, where $t = 2$, that every node except root contains 2 or 3 or 4 keys. red-black tree can be mapped to 2-3-4 tree essentially.

The following Python code shows example B-tree definition. It explicitly pass t when create a node.

```
class BTree:
    def __init__(self, t):
        self.t = t
        self.keys = []
        self.children = []
```

B-tree nodes commonly have satellite data as well. We ignore satellite data for illustration purpose.

In this chapter, we will firstly introduce how to generate B-tree by insertion. Two different methods will be explained. One is the classic method as in [2], that we split the node before insertion if it's full; the other is the modify-fix approach which is quite similar to the red-black tree solution [3] [2]. We will next explain how to delete key from B-tree and how to look up a key.

7.2 Insertion

B-tree can be created by inserting keys repeatedly. The basic idea is similar to the binary search tree. When insert key x, from the tree root, we examine all the keys in the node to find a position where all the keys on the left are less than x, while all the keys on the right are greater than x. If the current node is a leaf node, and it is not full (there are less then $2t - 1$ keys in this node), x will be insert at this position. Otherwise, the position points to a child node. We need recursively insert x to it.

Figure 7.3 shows one example. The B-tree illustrated is 2-3-4 tree. When insert key $x = 22$, because it's greater than the root, the right child contains key 26, 38, 45 is examined next; Since $22 < 26$, the first child contains key 21 and 25 are examined. This is a leaf node, and it is not full, key 22 is inserted to this node.

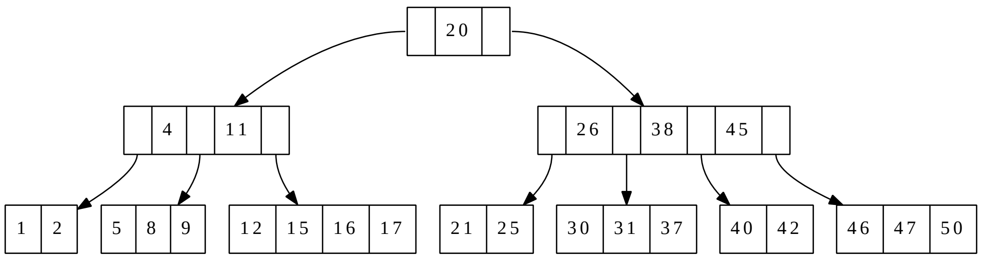
However, if there are $2t - 1$ keys in the leaf, the new key x can't be inserted, because this node is 'full'. When try to insert key 18 to the above example B-tree will meet this problem. There are 2 methods to solve it.

7.2.1 Splitting

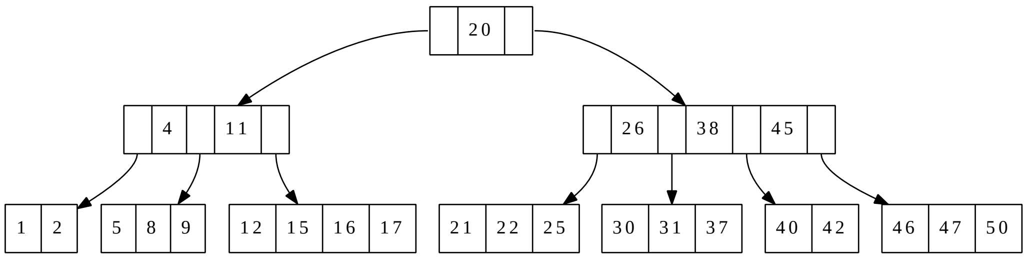
Split before insertion

If the node is full, one method to solve the problem is to split to node before insertion.

For a node with $at - 1$ keys, it can be divided into 3 parts as shown in Figure 7.4. the left part contains the first $t - 1$ keys and t children. The right part contains the rest $t - 1$ keys and t children. Both left part and right part are



(a) Insert key 22 to the 2-3-4 tree. 22 > 20, go to the right child; 22 < 26 go to the first child.



(b) 21 < 22 < 25, and the leaf isn't full.

Figure 7.3: Insertion is similar to binary search tree.

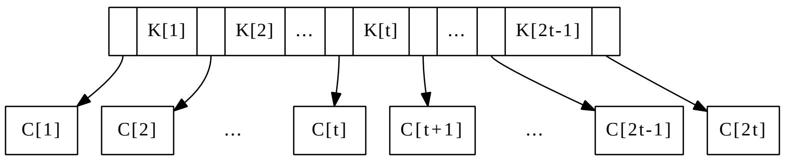
valid B-tree nodes. the middle part is the t-th key. We can push it up to the parent node (if the current node is root, then the this key, with the two children will be the new root).

For node x, denote $K(x)$ as keys, $C(x)$ as children. The i-th key as $k_i(x)$, the j-th child as $c_j(x)$. Below algorithm describes how to split the i-th child for a given node.

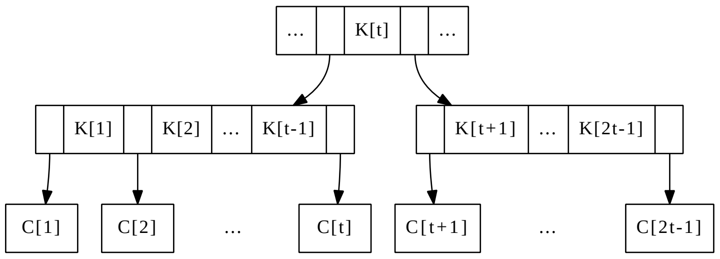
```
1: procedure SPLIT-CHILD($node, i$)
2:     $x \leftarrow c_i(node)$
3:     $y \leftarrow$ CREATE-NODE
4:     INSERT($K(node), i, k_t(x)$)
5:     INSERT($C(node), i+1, y$)
6:     $K(y) \leftarrow \{k_{t+1}(x), k_{t+2}(x), ..., k_{2t-1}(x)\}$
7:     $K(x) \leftarrow \{k_1(x), k_2(x), ..., k_{t-1}(x)\}$
8:     if $y$ is not leaf then
9:         $C(y) \leftarrow \{c_{t+1}(x), c_{t+2}(x), ..., c_{2t}(x)\}$
10:        $C(x) \leftarrow \{c_1(x), c_2(x), ..., c_t(x)\}$
```

The following example Python program implements this child splitting algorithm.

```
def split_child(node, i):
    t = node.t
    x = node.children[i]
    y = BTree(t)
    node.keys.insert(i, x.keys[t-1])
    node.children.insert(i+1, y)
    y.keys = x.keys[t:]
    x.keys = x.keys[:t-1]
```



(a) Before split



(b) After split

Figure 7.4: Split node

```
    if not is_leaf(x):
        y.children = x.children[t:]
        x.children = x.children[:t]
```

Where function is_leaf test if a node is leaf.

```
def is_leaf(t):
    return t.children == []
```

After splitting, a key is pushed up to its parent node. It is quite possible that the parent node has already been full. And this pushing violates the B-tree property.

In order to solve this problem, we can check from the root along the path of insertion traversing till the leaf. If there is any node in this path is full, the splitting is applied. Since the parent of this node has been examined, it is ensured that there are less than $2t - 1$ keys in the parent. It won't make the parent full if pushing up one key. This approach only need one single pass down the tree without any back-tracking.

If the root need splitting, a new node is created as the new root. There is no keys in this new created root, and the previous root is set as the only child. After that, splitting is performed top-down. And we can insert the new key finally.

```
1: function INSERT($T, k$)
2:     $r \leftarrow T$
3:     if $r$ is full then                                ▷ root is full
4:         $s \leftarrow$ CREATE-NODE
5:         $C(s) \leftarrow \{r\}$
6:         SPLIT-CHILD($s, 1$)
7:         $r \leftarrow s$
8:     return INSERT-NONFULL($r, k$)
```

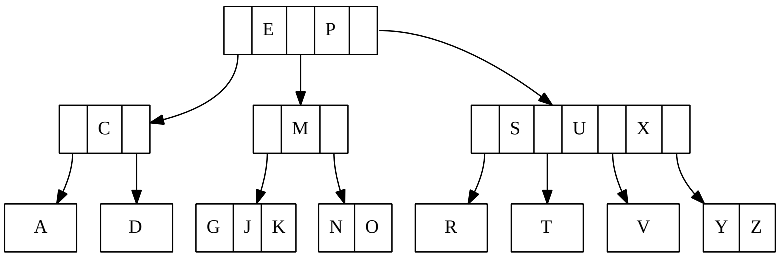
Where algorithm INSERT-NONFULL assumes the node passed in is not full.

If it is a leaf node, the new key is inserted to the proper position based on the order; Otherwise, the algorithm finds a proper child node to which the new key will be inserted. If this child is full, splitting will be performed.

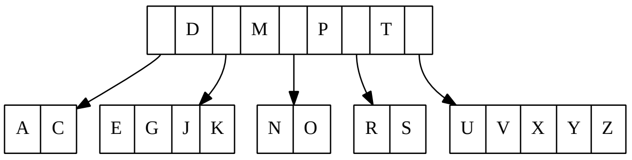
```
 1: function INSERT-NONFULL($T, k$)
 2:     if $T$ is leaf then
 3:         $i \leftarrow 1$
 4:         while $i \leq |K(T)| \land k > k_i(T)$ do
 5:             $i \leftarrow i + 1$
 6:         INSERT($K(T), i, k$)
 7:     else
 8:         $i \leftarrow |K(T)|$
 9:         while $i > 1 \land k < k_i(T)$ do
10:             $i \leftarrow i - 1$
11:         if $c_i(T)$ is full then
12:             SPLIT-CHILD($T, i$)
13:             if $k > k_i(T)$ then
14:                 $i \leftarrow i + 1$
15:         INSERT-NONFULL($c_i(T), k$)
16:     return $T$
```

This algorithm is recursive. In B-tree, the minimum degree t is typically relative to magnetic disk structure. Even small depth can support huge amount of data (with $t = 10$, maximum to 10 billion data can be stored in a B-tree with height of 10). The recursion can also be eliminated. This is left as exercise to the reader.

Figure 7.5 shows the result of continuously inserting keys G, M, P, X, A, C, D, E, J, K, N, O, R, S, T, U, V, Y, Z to the empty tree. The first result is the 2-3-4 tree ($t = 2$). The second result shows how it varies when $t = 3$.



(a) 2-3-4 tree.



(b) $t = 3$

Figure 7.5: Insertion result

Below example Python program implements this algorithm.

```
def insert(tr, key):
    root = tr
    if is_full(root):
        s = BTree(root.t)
        s.children.insert(0, root)
        split_child(s, 0)
        root = s
    return insert_nonfull(root, key)
```

And the insertion to non-full node is implemented as the following.

```
def insert_nonfull(tr, key):
    if is_leaf(tr):
        ordered_insert(tr.keys, key)
    else:
        i = len(tr.keys)
        while i>0 and key < tr.keys[i-1]:
            i = i-1
        if is_full(tr.children[i]):
            split_child(tr, i)
            if key>tr.keys[i]:
                i = i+1
        insert_nonfull(tr.children[i], key)
    return tr
```

Where function `ordered_insert` is used to insert an element to an ordered list. Function `is_full` tests if a node contains $2t - 1$ keys.

```
def ordered_insert(lst, x):
    i = len(lst)
    lst.append(x)
    while i>0 and lst[i]<lst[i-1]:
        (lst[i-1], lst[i]) = (lst[i], lst[i-1])
        i=i-1

def is_full(node):
    return len(node.keys) ≥ 2 * node.t - 1
```

For the array based collection, append on the tail is much more effective than insert in other position, because the later takes $O(n)$ time, if the length of the collection is n. The `ordered_insert` program firstly appends the new element at the end of the existing collection, then iterates from the last element to the first one, and checks if the current two elements next to each other are ordered. If not, these two elements will be swapped.

Insert then fixing

In functional settings, B-tree insertion can be realized in a way similar to red-black tree. When insert a key to red-black tree, it is firstly inserted as in the normal binary search tree, then recursive fixing is performed to resume the balance of the tree. B-tree can be viewed as extension to the binary search tree, that each node contains multiple keys and children. We can firstly insert the key without considering if the node is full. Then perform fixing to satisfy the minimum degree constraint.

$$insert(T, k) = fix(ins(T, k)) \tag{7.5}$$

Function $ins(T, k)$ traverse the B-tree T from root to find a proper position where key k can be inserted. After that, function fix is applied to resume the B-tree properties. Denote B-tree in a form of $T = (K, C, t)$, where K represents keys, C represents children, and t is the minimum degree.

Below is the Haskell definition of B-tree.

```
data BTree a = Node{ keys :: [a]
                   , children :: [BTree a]
                   , degree :: Int} deriving (Eq)
```

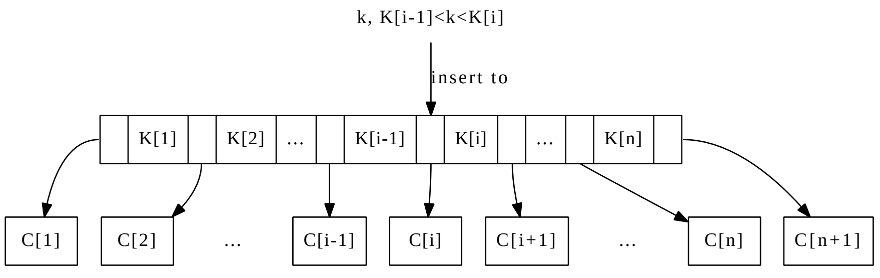
The insertion function can be provided based on this definition.

```
insert tr x = fixRoot $ ins tr x
```

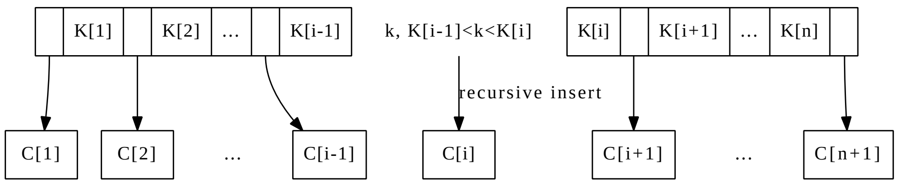
There are two cases when realize $ins(T, k)$ function. If the tree T is leaf, k is inserted to the keys; Otherwise if T is the branch node, we need recursively insert k to the proper child.

Figure 7.6 shows the branch case. The algorithm first locates the position. for certain key k_i, if the new key k to be inserted satisfy $k_{i-1} < k < k_i$, Then we need recursively insert k to child c_i.

This position divides the node into 3 parts, the left part, the child c_i and the right part.



(a) Locate the child to insert.



(b) Recursive insert.

Figure 7.6: Insert a key to a branch node

$$ins(T, k) = \begin{cases} (K' \cup \{k\} \cup K'', \Phi, t) & : C = \Phi, (K', K'') = divide(K, k) \\ make((K', C_1), ins(c, k), (K'', C_2')) & : (C_1, C_2) = split(|K'|, C) \end{cases} \tag{7.6}$$

The first clause deals with the leaf case. Function $divide(K, k)$ divide keys into two parts, all keys in the first part are not greater than k, and all rest keys are not less than k.

$$K = K' \cup K'' \land \forall k' \in K, k'' \in K'' \Rightarrow k' \leq k \leq k''$$

The second clause handle the branch case. Function $split(n, C)$ splits children in two parts, C_1 and C_2. C_1 contains the first n children; and C_2 contains the rest. Among C_2, the first child is denoted as c, and others are represented as C'_2.

Here the key k need be recursively inserted into child c. Function *make* takes 3 parameter. The first and the third are pairs of key and children; the second parameter is a child node. It examine if a B-tree node made from these keys and children violates the minimum degree constraint and performs fixing if necessary.

$$make((K', C'), c, (K'', C'')) = \begin{cases} fixFull((K', C'), c, (K'', C'')) & : full(c) \\ (K' \cup K'', C' \cup \{c\} \cup C'', t) & : otherwise \end{cases} \tag{7.7}$$

Where function $full(c)$ tests if the child c is full. Function $fixFull$ splits the the child c, and forms a new B-tree node with the pushed up key.

$$fixFull((K', C'), c, (K'', C'')) = (K' \cup \{k\} \cup K'', C' \cup \{c_1, c_2\} \cup C'', t) \tag{7.8}$$

Where $(c_1, k, c_2) = split(c)$. During splitting, the first $t-1$ keys and t children are extract to one new child, the last $t-1$ keys and t children form another child. The t-th key is pushed up.

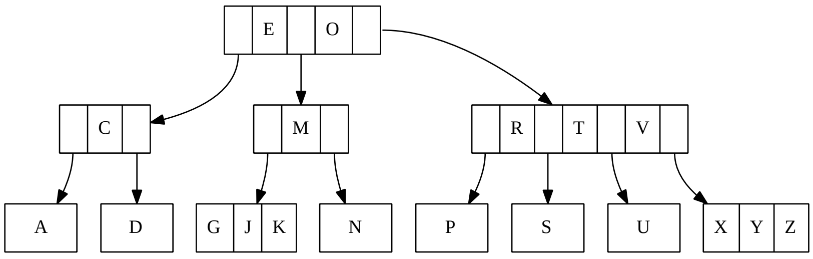
With all the above functions defined, we can realize $fix(T)$ to complete the functional B-tree insertion algorithm. It firstly checks if the root contains too many keys. If it exceeds the limit, splitting will be applied. The split result will be used to make a new node, so the total height of the tree increases by one.

$$fix(T) = \begin{cases} c & : T = (\Phi, \{c\}, t) \\ (\{k\}, \{c_1, c_2\}, t) & : full(T), (c_1, k, c_2) = split(T) \\ T & : otherwise \end{cases} \tag{7.9}$$

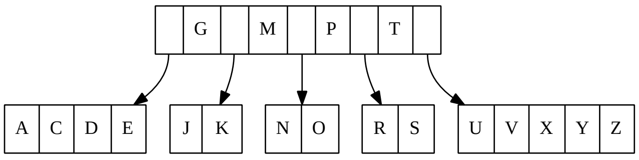
The following Haskell example code implements the B-tree insertion.

```
import qualified Data.List as L

ins (Node ks [] t) x = Node (L.insert x ks) [] t
ins (Node ks cs t) x = make (ks', cs') (ins c x) (ks'', cs'')
    where
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs

fixRoot (Node [] [tr] _) = tr -- shrink height
fixRoot tr = if full tr then Node [k] [c1, c2] (degree tr)
             else tr
    where
      (c1, k, c2) = split tr

make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)

fixFull (ks', cs') c (ks'', cs'') = Node (ks'++[k]++ks'')
                                         (cs'++[c1,c2]++cs'') (degree c)
    where
      (c1, k, c2) = split c

full tr = (length $ keys tr) > 2*(degree tr)-1
```

Figure 7.7 shows the varies of results of building B-trees by continuously inserting keys "GMPXACDEJKNORSTUVYZ".



(a) Insert result of a 2-3-4 tree.



(b) Insert result of a B-tree with $t = 3$

Figure 7.7: Insert then fixing results

Compare to the imperative insertion result as shown in figure 7.7 we can found that there are different. However, they are all valid because all B-tree properties are satisfied.

7.3 Deletion

Deleting a key from B-tree may violate balance properties. Except the root, a node shouldn't contain too few keys less than $t - 1$, where t is the minimum degree.

Similar to the approaches for insertion, we can either do some preparation so that the node from where the key being deleted contains enough keys; or do some fixing after the deletion if the node has too few keys.

7.3.1 Merge before delete method

We start from the easiest case. If the key k to be deleted can be located in node x, and x is a leaf node, we can directly remove k from x. If x is the root (the only node of the tree), we needn't worry about there are too few keys after deletion. This case is named as case 1 later.

In most cases, we start from the root, along a path to locate where is the node contains k. If k can be located in the internal node x, there are three sub cases.

- Case 2a, If the child y precedes k contains enough keys (more than t). We replace k in node x with k', which is the predecessor of k in child y. And recursively remove k' from y.

  The predecessor of k can be easily located as the last key of child y.

  This is shown in figure 7.8.

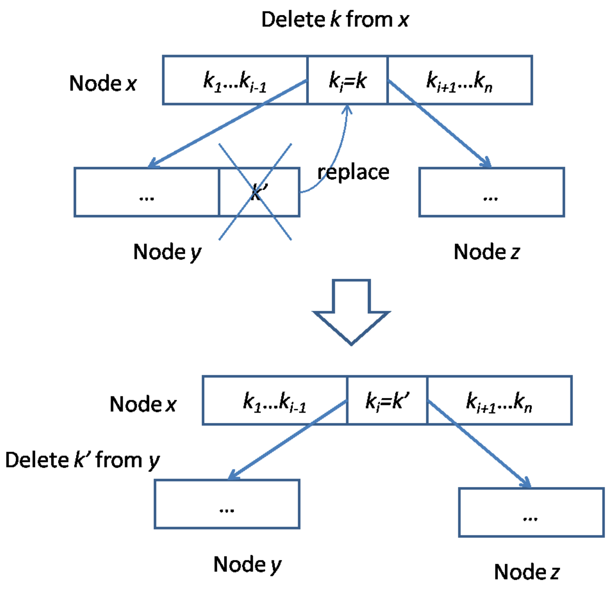


Figure 7.8: Replace and delete from predecessor.

- Case 2b, If y doesn't contain enough keys, while the child z follows k contains more than t keys. We replace k in node x with k'', which is the successor of k in child z. And recursively remove k'' from z.

  The successor of k can be easily located as the first key of child z.

  This sub-case is illustrated in figure 7.9.

- Case 2c, Otherwise, if neither y, nor z contains enough keys, we can merge y, k and z into one new node, so that this new node contains $2t - 1$ keys. After that, we can then recursively do the removing.

  Note that after merge, if the current node doesn't contain any keys, which means k is the only key in x. y and z are the only two children of x. we need shrink the tree height by one.

Figure 7.10 illustrates this sub-case.

the last case states that, if k can't be located in node x, the algorithm need find a child node c_i in x, so that the sub-tree c_i contains k. Before the deletion is recursively applied in c_i, we need make sure that there are at least t keys in c_i. If there are not enough keys, the following adjustment is performed.

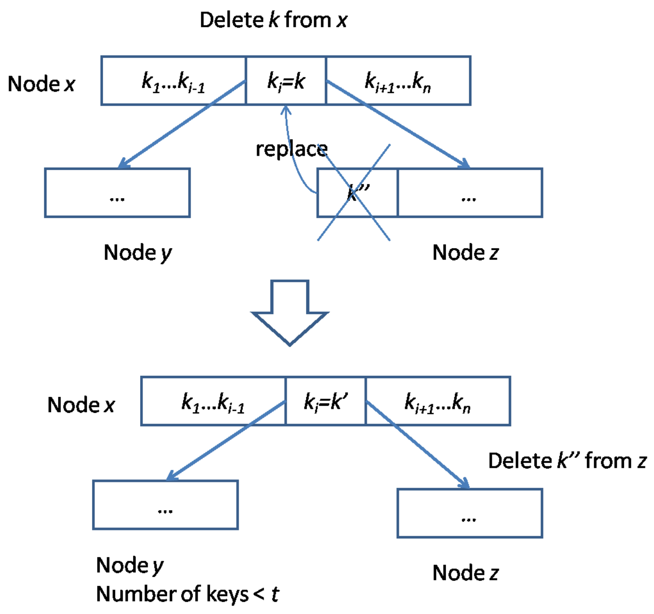


Figure 7.9: Replace and delete from successor.

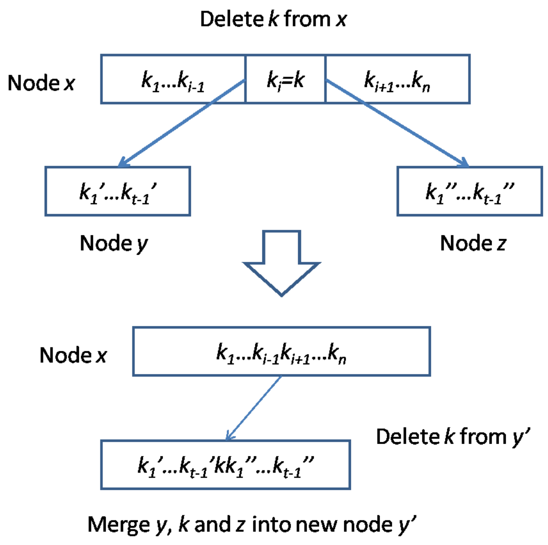


Figure 7.10: Merge and delete.

- Case 3a, We check the two sibling of c_i, which are c_{i-1} and c_{i+1}. If either one contains enough keys (at least t keys), we move one key from x down to c_i, and move one key from the sibling up to x. Also we need move the relative child from the sibling to c_i.

  This operation makes c_i contains enough keys for deletion. we can next try to delete k from c_i recursively.

  Figure 7.11 illustrates this case.

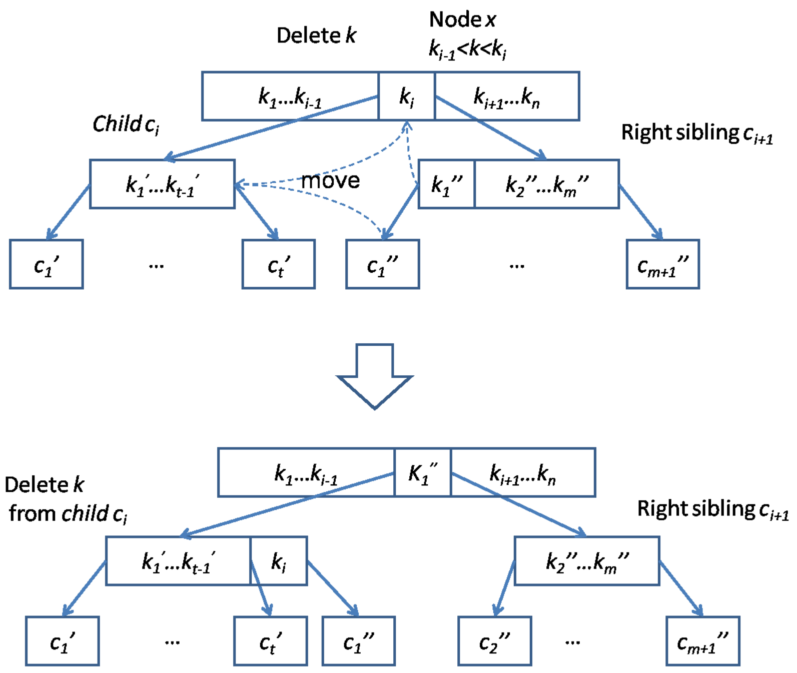


Figure 7.11: Borrow from the right sibling.

- Case 3b, In case neither one of the two siblings contains enough keys, we then merge c_i, a key from x, and either one of the sibling into a new node. Then do the deletion on this new node.

Figure 7.12 shows this case.

Before define the B-tree delete algorithm, we need provide some auxiliary functions. Function CAN-DEL tests if a node contains enough keys for deletion.

```
1: function CAN-DEL($T$)
2:     return $|K(T)| \geq t$
```

Procedure MERGE-CHILDREN(T, i) merges child $c_i(T)$, key $k_i(T)$, and child $c_{i+1}(T)$ into one big node.

```
1: procedure MERGE-CHILDREN($T, i$)        ▷ Merge $c_i(T)$, $k_i(T)$, and $c_{i+1}(T)$
2:     $x \leftarrow c_i(T)$
3:     $y \leftarrow c_{i+1}(T)$
4:     $K(x) \leftarrow K(x) \cup \{k_i(T)\} \cup K(y)$
5:     $C(x) \leftarrow C(x) \cup C(y)$
6:     REMOVE-AT($K(T), i$)
```

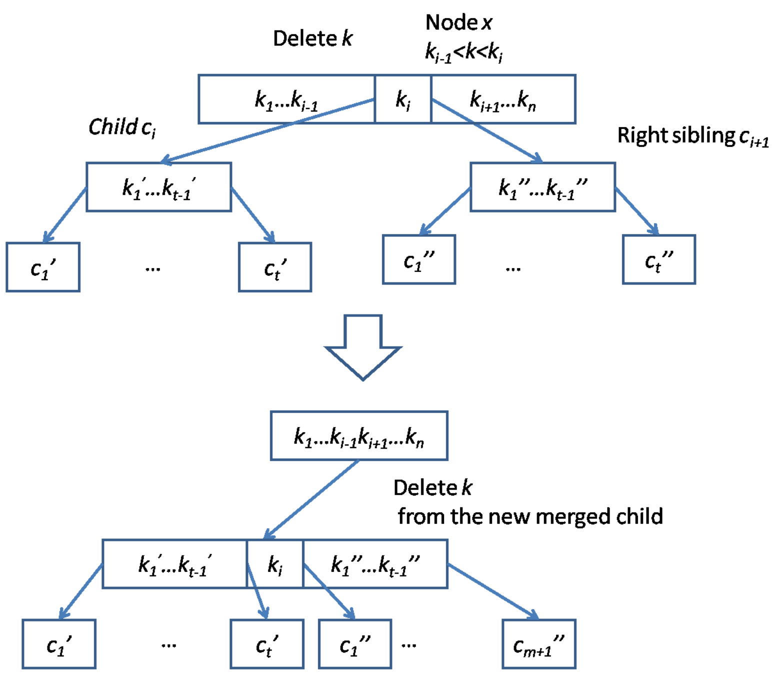


Figure 7.12: Merge c_i, k, and c_{i+1} to a new node.

```
7:     REMOVE-AT($C(T), i + 1$)
```

Procedure MERGE-CHILDREN merges the i-th child, the i-th key, and $i+1$-th child of node T into a new child, and remove the i-th key and $i+1$-th child from T after merging.

With these functions defined, the B-tree deletion algorithm can be given by realizing the above 3 cases.

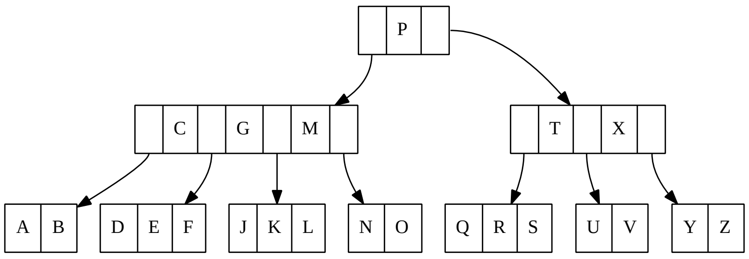
```
1: function DELETE($T, k$)
2:     $i \leftarrow 1$
3:     while $i \leq |K(T)|$ do
4:         if $k = k_i(T)$ then
5:             if $T$ is leaf then                              ▷ case 1
6:                 REMOVE($K(T), k$)
7:             else                                             ▷ case 2
8:                 if CAN-DEL($c_i(T)$) then                    ▷ case 2a
9:                     $k_i(T) \leftarrow$ LAST-KEY($c_i(T)$)
10:                    DELETE($c_i(T), k_i(T)$)
11:                else if CAN-DEL($c_{i+1}(T)$) then           ▷ case 2b
12:                    $k_i(T) \leftarrow$ FIRST-KEY($c_{i+1}(T)$)
13:                    DELETE($c_{i+1}(T), k_i(T)$)
14:                else                                         ▷ case 2c
15:                    MERGE-CHILDREN($T, i$)
16:                    DELETE($c_i(T), k$)
17:                    if $K(T) = NIL$ then
18:                        $T \leftarrow c_i(T)$                ▷ Shrinks height
```

```
19:             **return** $T$
20:         **else if** $k < k_i(T)$ **then**
21:             Break
22:         **else**
23:             $i \leftarrow i + 1$

24:     **if** $T$ is leaf **then**
25:         **return** $T$                                    ▷ $k$ doesn't exist in $T$.
26:     **if** $\neg$ CAN-DEL$(c_i(T))$ **then**                           ▷ case 3
27:         **if** $i > 1 \wedge$ CAN-DEL$(c_{i-1}(T))$ **then**          ▷ case 3a: left sibling
28:             INSERT$(K(c_i(T)), k_{i-1}(T))$
29:             $k_{i-1}(T) \leftarrow$ POP-BACK$(K(c_{i-1}(T)))$
30:             **if** $c_i(T)$ isn't leaf **then**
31:                 $c \leftarrow$ POP-BACK$(C(c_{i-1}(T)))$
32:                 INSERT$(C(c_i(T)), c)$
33:         **else if** $i \leq |C(T)| \wedge$ CAN-DEL$(c_{i_1}(T))$ **then**  ▷ case 3a: right sibling
34:             APPEND$(K(c_i(T)), k_i(T))$
35:             $k_i(T) \leftarrow$ POP-FRONT$(K(c_{i+1}(T)))$
36:             **if** $c_i(T)$ isn't leaf **then**
37:                 $c \leftarrow$ POP-FRONT$(C(c_{i+1}(T)))$
38:                 APPEND$(C(c_i(T)), c)$
39:         **else**                                                  ▷ case 3b
40:             **if** $i > 1$ **then**
41:                 MERGE-CHILDREN$(T, i - 1)$
42:             **else**
43:                 MERGE-CHILDREN$(T, i)$
44:     DELETE$(c_i(T), k)$                                      ▷ recursive delete
45:     **if** $K(T) = NIL$ **then**                                ▷ Shrinks height
46:         $T \leftarrow c_1(T)$
47:     **return** $T$
```

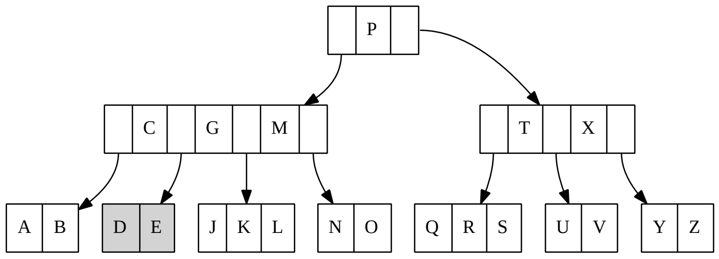
Figure 7.13, 7.14, and 7.15 show the deleting process step by step. The nodes modified are shaded.

The following example Python program implements the B-tree deletion algorithm.

```
def can_remove(tr):
    return len(tr.keys) ≥ tr.t

def replace_key(tr, i, k):
    tr.keys[i] = k
    return k

def merge_children(tr, i):
    tr.children[i].keys += [tr.keys[i]] + tr.children[i+1].keys
    tr.children[i].children += tr.children[i+1].children
    tr.keys.pop(i)
    tr.children.pop(i+1)

def B_tree_delete(tr, key):
    i = len(tr.keys)
```

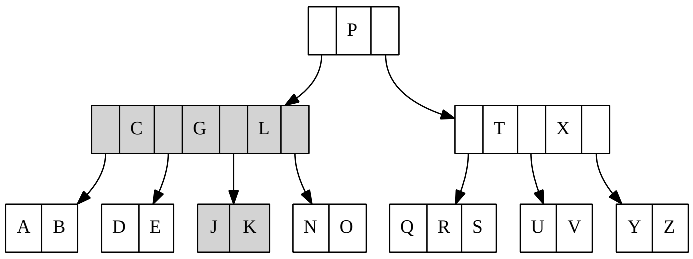


(a) A B-tree before deleting.

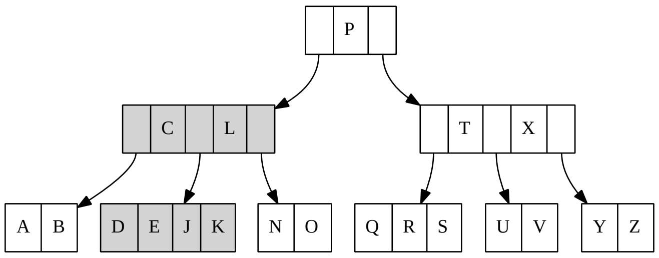


(b) After delete key 'F', case 1.

Figure 7.13: Result of B-tree deleting (1).

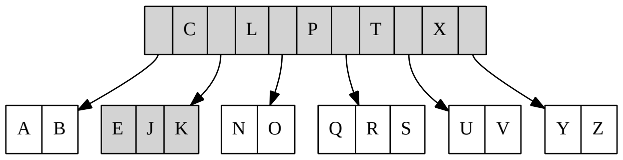


(a) After delete key 'M', case 2a.

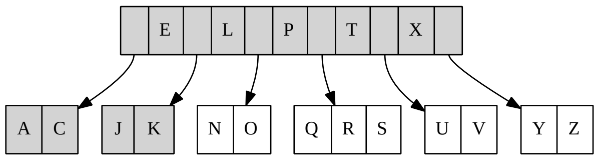


(b) After delete key 'G', case 2c.

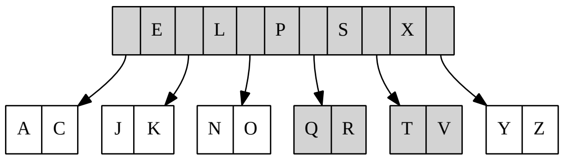
Figure 7.14: Result of B-tree deleting program (2)



(a) After delete key 'D', case 3b, and height is shrunk.



(b) After delete key 'B', case 3a, borrow from right sibling.



(c) After delete key 'U', case 3a, borrow from left sibling.

Figure 7.15: Result of B-tree deleting program (3)

```
while i>0:
    if key == tr.keys[i-1]:
        if tr.leaf:  # case 1 in CLRS
            tr.keys.remove(key)
        else: # case 2 in CLRS
            if tr.children[i-1].can_remove(): # case 2a
                key = tr.replace_key(i-1, tr.children[i-1].keys[-1])
                B_tree_delete(tr.children[i-1], key)
            elif tr.children[i].can_remove(): # case 2b
                key = tr.replace_key(i-1, tr.children[i].keys[0])
                B_tree_delete(tr.children[i], key)
            else: # case 2c
                tr.merge_children(i-1)
                B_tree_delete(tr.children[i-1], key)
                if tr.keys==[]: # tree shrinks in height
                    tr = tr.children[i-1]
        return tr
    elif key > tr.keys[i-1]:
        break
    else:
        i = i-1
# case 3
if tr.leaf:
    return tr #key doesn't exist at all
if not tr.children[i].can_remove():
    if i>0 and tr.children[i-1].can_remove(): #left sibling
        tr.children[i].keys.insert(0, tr.keys[i-1])
        tr.keys[i-1] = tr.children[i-1].keys.pop()
```

```
            if not tr.children[i].leaf:
                tr.children[i].children.insert(0, tr.children[i-1].children.pop())
        elif i<len(tr.children) and tr.children[i+1].can_remove(): #right sibling
            tr.children[i].keys.append(tr.keys[i])
            tr.keys[i]=tr.children[i+1].keys.pop(0)
            if not tr.children[i].leaf:
                tr.children[i].children.append(tr.children[i+1].children.pop(0))
        else: # case 3b
            if i>0:
                tr.merge_children(i-1)
            else:
                tr.merge_children(i)
    B_tree_delete(tr.children[i], key)
    if tr.keys==[]: # tree shrinks in height
        tr = tr.children[0]
    return tr
```

7.3.2 Delete and fix method

The merge and delete algorithm is a bit complex. There are several cases, and in each case, there are sub cases to deal.

Another approach to design the deleting algorithm is to perform fixing after deletion. It is similar to the insert-then-fix strategy.

$$delete(T, k) = fix(del(T, k)) \tag{7.10}$$

When delete a key from B-tree, we firstly locate which node this key is contained. We traverse from the root to the leaves till find this key in some node.

If this node is a leaf, we can remove the key, and then examine if the deletion makes the node contains too few keys to satisfy the B-tree balance properties.

If it is a branch node, removing the key breaks the node into two parts. We need merge them together. The merging is a recursive process which is shown in figure 7.16.

When do merging, if the two nodes are not leaves, we merge the keys together, and recursively merge the last child of the left part and the first child of the right part to one new node. Otherwise, if they are leaves, we merely put all keys together.

Till now, the deleting in performed in straightforward way. However, deleting decreases the number of keys of a node, and it may result in violating the B-tree balance properties. The solution is to perform fixing along the path traversed from root.

During the recursive deletion, the branch node is broken into 3 parts. The left part contains all keys less than k, includes $k_1, k_2, ..., k_{i-1}$, and children $c_1, c_2, ..., c_{i-1}$, the right part contains all keys greater than k, say $k_i, k_{i+1}, ..., k_{n+1}$, and children $c_{i+1}, c_{i+2}, ..., c_{n+1}$. Then key k is recursively deleted from child c_i. Denote the result becomes c'_i after that. We need make a new node from these 3 parts, as shown in figure 7.17.

At this time point, we need examine if c'_i contains enough keys. If there are to less keys (less than $t - 1$, but not t in contrast to the merge-and-delete approach), we can either borrow a key-child pair from the left or the right part,

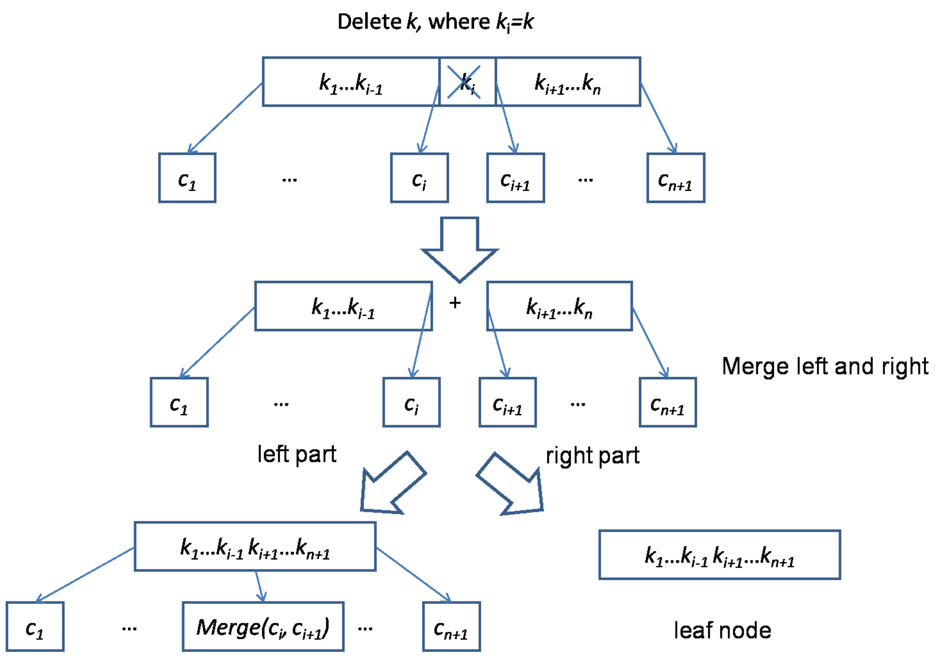


Figure 7.16: Delete a key from a branch node. Removing k_i breaks the node into 2 parts. Merging these 2 parts is a recursive process. When the two parts are leaves, the merging terminates.

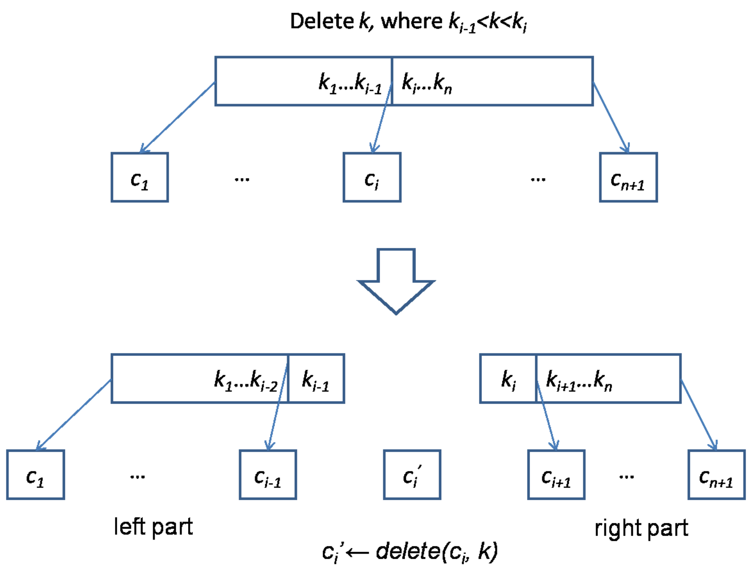


Figure 7.17: After delete key k from node c_i, denote the result as c_i'. The fixing makes a new node from the left part, c_i' and the right part.

and do inverse operation of splitting. Figure 7.18 shows example of borrowing from the left part.

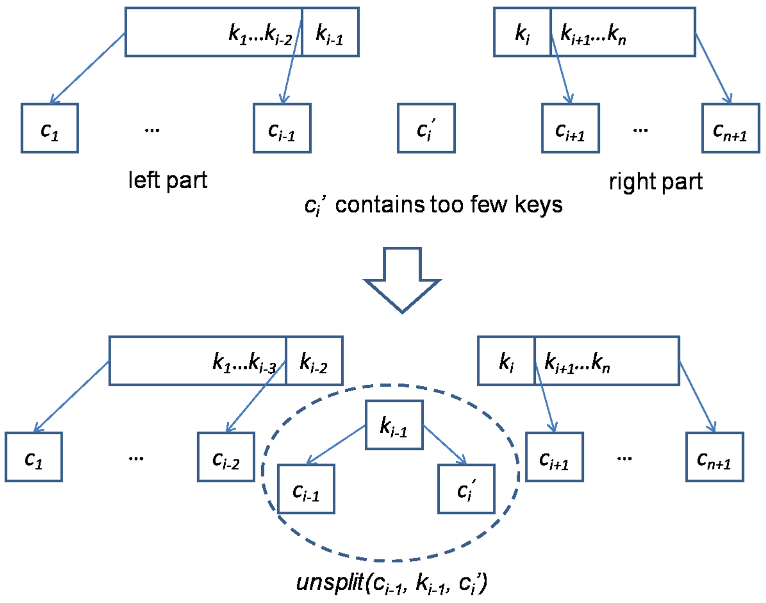


Figure 7.18: Borrow a key-child pair from left part and un-split to a new child.

If both left part and right part are empty, we can simply push c_i' up.

Denote the B-tree as $T = (K, C, t)$, where K and C are keys and children. The $del(T, k)$ function deletes key k from the tree.

$$del(T,k) = \begin{cases} (delete(K,k), \Phi, t) & : C = \Phi \\ merge((K_1, C_1, t), (K_2, C_2, t)) & : k_i = k \\ make((K_1', C_1'), del(c, k), (K_2', C_2')) & : k \notin K \end{cases} \quad (7.11)$$

If children $C = \Phi$ is empty, T is leaf. k is deleted from keys directly. Otherwise, T is internal node. If $k \in K$, removing it separates the keys and children in two parts (K_1, C_1) and (K_2, C_2). They will be recursively merged.

$$\begin{array}{l} K_1 = \{k_1, k_2, ..., k_{i-1}\} \\ K_2 = \{k_{i+1}, k_{i+2}, ..., k_m\} \\ C_1 = \{c_1, c_2, ..., c_i\} \\ C_2 = \{c_{i+1}, c_{i+2}, ..., c_{m+1}\} \end{array}$$

If $k \notin K$, we need locate a child c, and further delete k from it.

$$\begin{array}{l} (K_1', K_2') = (\{k' | k' \in K, k' < k\}, \{k' | k' \in K, k < k'\}) \\ (C_1', \{c\} \cup C_2') = splitAt(|K_1'|, C) \end{array}$$

The recursive merge function is defined as the following. When merge two trees $T_1 = (K_1, C_1, t)$ and $T_2 = (K_2, C_2, t)$, if both are leaves, we create a new leave by concatenating the keys. Otherwise, the last child in C_1, and the first child in C_2 are recursively merged. And we call *make* function to form the new tree. When C_1 and C_2 are not empty, denote the last child of C_1 as $c_{1,m}$, the rest as C_1'; the first child of C_2 as $C_{2,1}$, the rest as C_2'. Below equation defines the merge function.

$$merge(T_1, T_2) = \left\{ \begin{array}{rcl} (K_1 \cup K_2, \Phi, t) & : & C_1 = C_2 = \Phi \\ make((K_1, C_1'), merge(c_{1,m}, c_{2,1}), (K_2, C_2')) & : & otherwise \end{array} \right. \tag{7.12}$$

The *make* function defined above only handles the case that a node contains too many keys due to insertion. When delete key, it may cause a node contains too few keys. We need test and fix this situation as well.

$$make((K', C'), c, (K'', C'')) = \left\{ \begin{array}{rcl} fixFull((K', C'), c, (K'', C'')) & : & full(c) \\ fixLow((K', C'), c, (K'', C'')) & : & low(c) \\ (K' \cup K'', C' \cup \{c\} \cup C'', t) & : & otherwise \end{array} \right. \tag{7.13}$$

Where $low(T)$ checks if there are too few keys less than $t - 1$. Function $fixLow(P_l, c, P_r)$ takes three arguments, the left pair of keys and children, a child node, and the right pair of keys and children. If the left part isn't empty, we borrow a pair of key-child, and do un-splitting to make the child contain enough keys, then recursively call *make*; If the right part isn't empty, we borrow a pair from the right; and if both sides are empty, we return the child node as result. In this case, the height of the tree shrinks.

Denote the left part $P_l = (K_l, C_l)$. If K_l isn't empty, the last key and child are represented as $k_{l,m}$ and $c_{l,m}$ respectively. The rest keys and children become K_l' and C_l'; Similarly, the right part is denoted as $P_r = (K_r, C_r)$. If K_r isn't empty, the first key and child are represented as $k_{r,1}$, and $c_{r,1}$. The rest keys and children are K_r' and C_r'. Below equation gives the definition of $fixLow$.

$$fixLow(P_l, c, P_r) = \left\{ \begin{array}{rcl} make((K_l', C_l'), unsplit(c_{l,m}, k_{l,m}, c), (K_r, C_r)) & : & K_l \neq \Phi \\ make((K_r, C_r), unsplit(c, k_{r,1}, c_{r,1}), (K_r', C_r')) & : & K_r \neq \Phi \\ c & : & otherwise \end{array} \right. \tag{7.14}$$

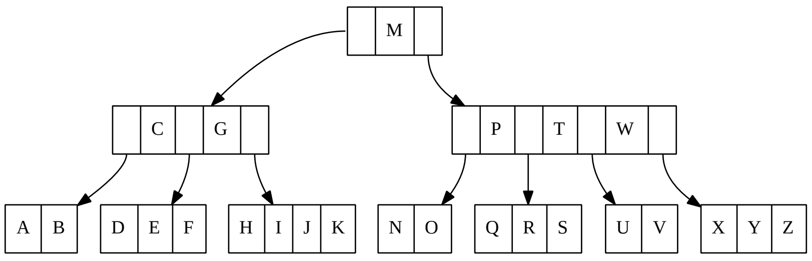
Function $unsplit(T_1, k, T_2)$ is the inverse operation to splitting. It forms a new B-tree nodes from two small nodes and a key.

$$unsplit(T_1, k, T_2) = (K_1 \cup \{k\} \cup K_2, C_1 \cup C_2, t) \tag{7.15}$$

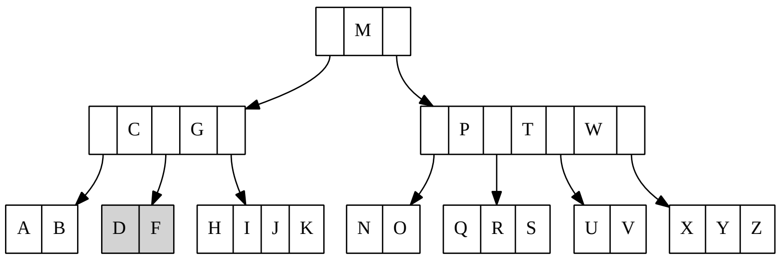
The following example Haskell program implements the B-tree deletion algorithm.

```haskell
import qualified Data.List as L

delete tr x = fixRoot $ del tr x

del:: (Ord a) ⇒ BTree a → a → BTree a
del (Node ks [] t) x = Node (L.delete x ks) [] t
del (Node ks cs t) x =
    case L.elemIndex x ks of
      Just i → merge (Node (take i ks) (take (i+1) cs) t)
                     (Node (drop (i+1) ks) (drop (i+1) cs) t)
      Nothing → make (ks', cs') (del c x) (ks'', cs'')
    where
      (ks', ks'') = L.partition (<x) ks
      (cs', (c:cs'')) = L.splitAt (length ks') cs

merge (Node ks [] t) (Node ks' [] _) = Node (ks++ks') [] t
merge (Node ks cs t) (Node ks' cs' _) = make (ks, init cs)
                                             (merge (last cs) (head cs'))
                                             (ks', tail cs')

make (ks', cs') c (ks'', cs'')
    | full c = fixFull (ks', cs') c (ks'', cs'')
    | low c  = fixLow  (ks', cs') c (ks'', cs'')
    | otherwise = Node (ks'++ks'') (cs'++[c]++cs'') (degree c)

low tr = (length $ keys tr) < (degree tr)-1

fixLow (ks'@(_:_), cs') c (ks'', cs'') = make (init ks', init cs')
                                              (unsplit (last cs') (last ks') c)
                                              (ks'', cs'')
fixLow (ks', cs') c (ks''@(_:_), cs'') = make (ks', cs')
                                              (unsplit c (head ks'') (head cs''))
                                              (tail ks'', tail cs'')
fixLow _ c _ = c

unsplit c1 k c2 = Node ((keys c1)++[k]++(keys c2))
                       ((children c1)++(children c2)) (degree c1)
```

When delete the same keys from the B-tree as in merge and fixing approach, the results are different. However, both satisfy the B-tree properties, so they are all valid.

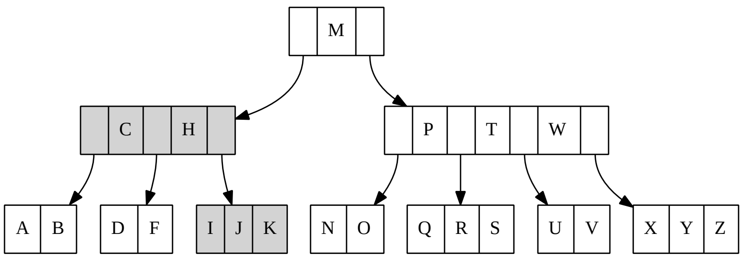


(a) B-tree before deleting

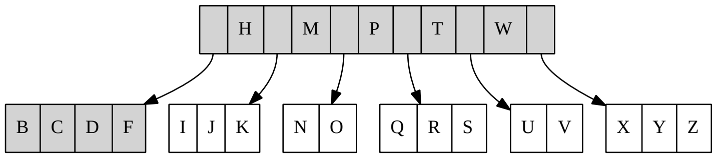


(b) After delete key 'E'.

Figure 7.19: Result of delete-then-fixing (1)

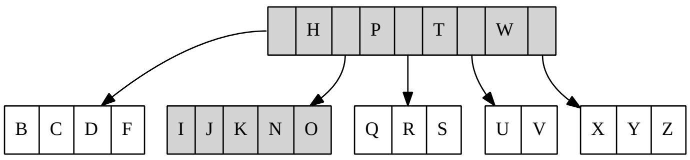


(a) After delete key 'G',

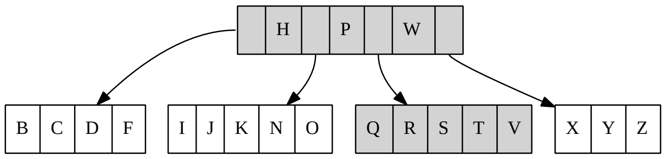


(b) After delete key 'A'.

Figure 7.20: Result of delete-then-fixing (2)



(a) After delete key 'M'.



(b) After delete key 'U'.

Figure 7.21: Result of delete-then-fixing (3)

7.4 Searching

Searching in B-tree can be considered as the generalized tree search extended from binary search tree.

When searching in the binary tree, there are only 2 different directions, the left and the right. However, there are multiple directions in B-tree.

```
1: function SEARCH($T, k$)
2:     loop
3:         $i \leftarrow 1$
4:         while $i \leq |K(T)| \land k > k_i(T)$ do
5:             $i \leftarrow i + 1$
6:         if $i \leq |K(T)| \land k = k_i(T)$ then
```

```
 7:             return $(T, i)$
 8:         if $T$ is leaf then
 9:             return $NIL$                                  ▷ $k$ doesn't exist
10:         else
11:             $T \leftarrow c_i(T)$
```

Starts from the root, this program examines each key one by one from the smallest to the biggest. In case it finds the matched key, it returns the current node and the index of this key. Otherwise, if it finds the position i that $k_i < k < k_{i+1}$, the program will next search the child node c_{i+1} for the key. If it traverses to some leaf node, and fails to find the key, the empty value is returned to indicate that this key doesn't exist in the tree.

The following example Python program implements the search algorithm.

```
def B_tree_search(tr, key):
    while True:
        for i in range(len(tr.keys)):
            if key ≤ tr.keys[i]:
                break
        if key == tr.keys[i]:
            return (tr, i)
        if tr.leaf:
            return None
        else:
            if key > tr.keys[-1]:
                i=i+1
            tr = tr.children[i]
```

The search algorithm can also be realized by recursion. When search key k in B-tree $T = (K, C, t)$, we partition the keys with k.

$$\begin{array}{l} K_1 = \{k' | k' < k\} \\ K_2 = \{k' | k \leq k'\} \end{array}$$

Thus K_1 contains all the keys less than k, and K_2 holds the rest. If the first element in K_2 is equal to k, we find the key. Otherwise, we recursively search the key in child $c_{|K_1|+1}$.

$$search(T, k) = \left\{ \begin{array}{rcl} (T, |K_1| + 1) & : & k \in K_2 \\ \Phi & : & C = \Phi \\ search(c_{|K_1|+1}, k) & : & otherwise \end{array} \right. \tag{7.16}$$

Below example Haskell program implements this algorithm.

```
search :: (Ord a)⇒ BTree a → a → Maybe (BTree a, Int)
search tr@(Node ks cs _) k
    | matchFirst k $ drop len ks = Just (tr, len)
    | otherwise = if null cs then Nothing
                  else search (cs !! len) k
    where
      matchFirst x (y:_) = x==y
      matchFirst x _ = False
      len = length $ filter (<k) ks
```

7.5 Notes and short summary

In this chapter, we explained the B-tree data structure as a kind of extension from binary search tree. The background knowledge of magnetic disk access is skipped, user can refer to [2] for detail. For the three main operations, insertion, deletion, and searching, both imperative and functional algorithms are given. They traverse from the root to the leaf. All the three operations perform in time proportion to the height of the tree. Because B-tree always maintains the balance properties. The performance is ensured to bound to $O(\lg n)$ time, where n is the number of the keys in B-tree.

Exercise 7.1

- Eliminate the recursion in imperative B-tree insertion algorithm.