Part III

Heaps

Chapter 8

Binary Heaps

8.1 Introduction

Heap is one of the elementary data structure. It is widely used to solve some practical problems, such as sorting, prioritized scheduling, and graph algorithms[2].

Most popular implementations of heap are using a kind of implicit binary heap by array, which is described in [2]. Examples include C++/STL heap and Python heapq. The most efficient heap sort algorithm is also realized with binary heap as proposed by R. W. Floyd [3] [5].

However, heaps can be general and realized with varies of other data structures besides array. In this chapter, explicit binary tree is used. It leads to Leftist heaps, Skew heaps, and Splay heaps, which are suitable for purely functional implementation as shown by Okasaki[6].

A heap is a data structure that satisfies the following *heap property*.

- Top operation always returns the minimum (maximum) element;
- Pop operation removes the top element from the heap while the heap property should be kept, so that the new top element is still the minimum (maximum) one;
- Insert a new element to heap should keep the heap property. That the new top is still the minimum (maximum) element;
- Other operations including merge etc should all keep the heap property.

This is a kind of recursive definition, while it doesn't limit the under ground data structure.

We call the heap with the minimum element on top as *min-heap*, while if the top keeps the maximum element, we call it *max-heap*.

8.2 Implicit binary heap by array

Considering the heap definition in previous section, one option to implement heap is by using trees. A straightforward solution is to store the minimum (maximum) element in the root of the tree, so for 'top' operation, we simply return the root as the result. And for 'pop' operation, we can remove the root and rebuild the tree from the children.

If binary tree is used to implement heap the heap, we can call it *binary heap*. This chapter explains three different realizations for binary heap.

8.2.1 Definition

The first one is implicit binary tree. Consider the problem how to represent a complete binary tree with array. (For example, try to represent a complete binary tree in the programming language doesn't support structure or record data type. Only array can be used). One solution is to pack all elements from top level (root) down to bottom level (leaves).

Figure 8.1 shows a complete binary tree and its corresponding array representation.

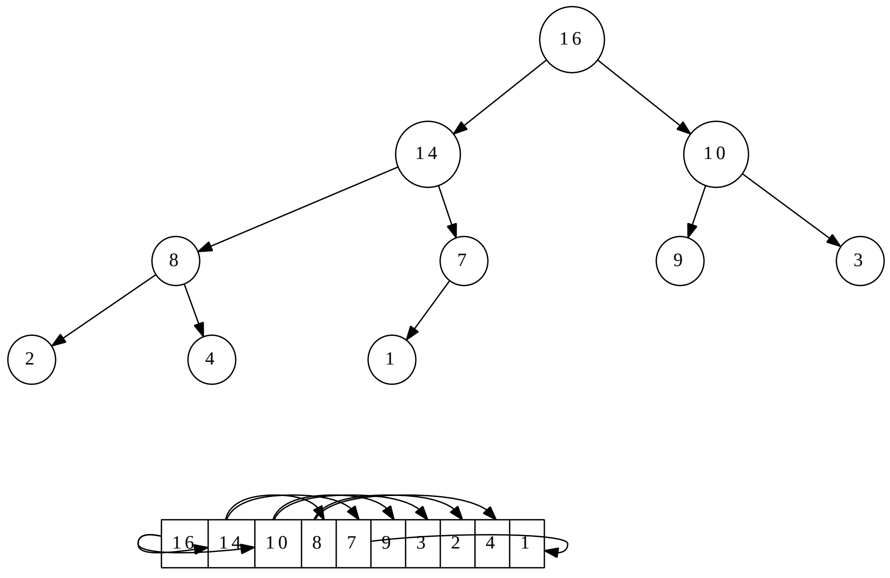


Figure 8.1: Mapping between a complete binary tree and array

This mapping between tree and array can be defined as the following equations (The array index starts from 1).

```
1: function PARENT($i$)
2:     return $\lfloor \frac{i}{2} \rfloor$

3: function LEFT($i$)
4:     return $2i$

5: function LEFT($i$)
6:     return $2i + 1$
```

For a given tree node which is represented as the i-th element of the array, since the tree is complete, we can easily find its parent node as the $\lfloor i/2 \rfloor$-th element; Its left child with index of $2i$ and right child of $2i + 1$. If the index of the child exceeds the length of the array, it means this node does not have such a child (leaf for example).

In real implementation, this mapping can be calculated fast with bit-wise operation like the following example ANSI C code. Note that, the array index starts from zero in C like languages.

```
#define PARENT(i) ((((i) + 1) >> 1) - 1)

#define LEFT(i) (((i) << 1) + 1)

#define RIGHT(i) (((i) + 1) << 1)
```

8.2.2 Heapify

The most important thing for heap algorithm is to maintain the heap property, that the top element should be the minimum (maximum) one.

For the implicit binary heap by array, it means for a given node, which is represented as the i-th index, we need develop a method to check if both its two children conform to this property. In case there is violation, we need swap the parent and child recursively [2] to fix the problem.

Below algorithm shows the iterative solution to enforce the min-heap property from the given index of the array.

```
1: function HEAPIFY($A, i$)
2:     $n \leftarrow |A|$
3:     loop
4:         $l \leftarrow$ LEFT($i$)
5:         $r \leftarrow$ RIGHT($i$)
6:         $smallest \leftarrow i$
7:         if $l < n \land A[l] < A[i]$ then
8:             $smallest \leftarrow l$
9:         if $r < n \land A[r] < A[smallest]$ then
10:            $smallest \leftarrow r$
11:        if $smallest \neq i$ then
12:            EXCHANGE $A[i] \leftrightarrow A[smallest]$
13:            $i \leftarrow smallest$
14:        else
15:            return
```

For array A and the given index i, None its children should be bigger than $A[i]$, in case there is violation, we pick the smallest element as $A[i]$, and swap the previous $A[i]$ to child. The algorithm traverses the tree top-down to fix the heap property until either reach a leaf or there is no heap property violation.

The HEAPIFY algorithm takes $O(\lg n)$ time, where n is the number of elements. This is because the loop time is proportion to the height of the binary tree.

When implement this algorithm, the comparison method can be passed as a parameter, so that both min-heap and max-heap can be supported. The following ANSI C example code uses this approach.

```
typedef int (*Less)(Key, Key);
int less(Key x, Key y) { return x < y; }
int notless(Key x, Key y) { return !less(x, y); }

void heapify(Key* a, int i, int n, Less lt) {
    int l, r, m;
    while (1) {
        l = LEFT(i);
        r = RIGHT(i);
        m = i;
        if (l < n && lt(a[l], a[i]))
            m = l;
        if (r < n && lt(a[r], a[m]))
            m = r;
        if (m != i) {
            swap(a, i, m);
            i = m;
        }
        else
            break;
    }
}
```

Figure 8.2 illustrates the steps when Heapify processing the array $\{16, 4, 10, 14, 7, 9, 3, 2, 8, 1\}$. The array changes to $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$.

8.2.3 Build a heap

With Heapify algorithm defined, it is easy to build a heap from the arbitrary array. Observe that the numbers of nodes in a complete binary tree for each level is a list like below:

$1, 2, 4, 8, ..., 2^i, ...$

The only exception is the last level. Since the tree may not full (note that complete binary tree doesn't mean full binary tree), the last level contains at most 2^{p-1} nodes, where $2^p \leq n$ and n is the length of the array.

The Heapify algorithm doesn't have any effect on leave node. We can skip applying Heapify for all leaves. In other words, all leaf nodes have already satisfied the heap property. We only need start checking and maintain the heap property from the last branch node. the index of the last branch node is no greater than $\lfloor n/2 \rfloor$.

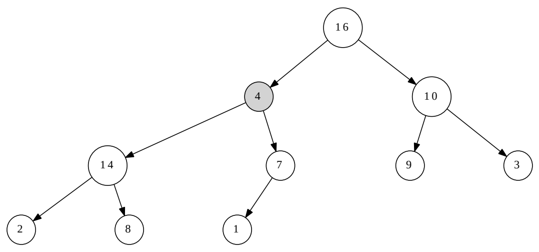
Based on this fact, we can build a heap with the following algorithm. (Assume the heap is min-heap).

```
1: function Build-Heap($A$)
2:     $n \leftarrow |A|$
3:     for $i \leftarrow \lfloor n/2 \rfloor$ down to 1 do
4:         Heapify($A, i$)
```

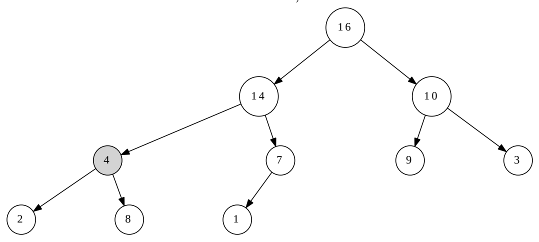
Although the complexity of Heapify is $O(\lg n)$, the running time of Build-Heap is not bound to $O(n \lg n)$ but $O(n)$. This is a linear time algorithm. [2] provides the detailed proof.

Below ANSI C example program implements this heap building function.

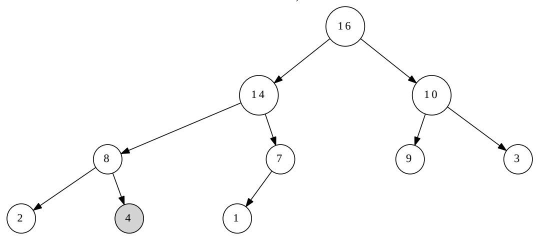
```
void build_heap(Key* a, int n, Less lt) {
    int i;
    for (i = (n-1) >> 1; i ≥ 0; --i)
```



a. Step 1, 14 is the biggest element among 4, 14, and 7. Swap 4 with the left child;



b. Step 2, 8 is the biggest element among 2, 4, and 8. Swap 4 with the right child;

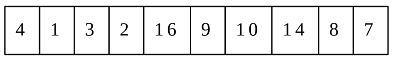


c. 4 is the leaf node. It hasn't any children. Process terminates.

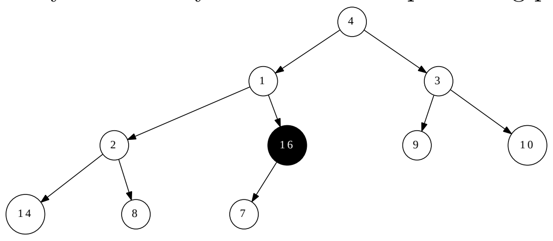
Figure 8.2: Heapify example, a max-heap case.

```
        heapify(a, i, n, lt);
}
```

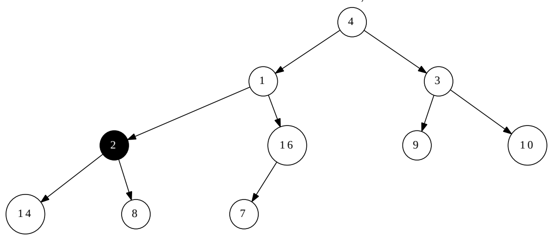
Figure 8.3 and 8.4 show the steps when building a heap from array $\{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$. The node in black color is the one HEAPIFY being applied, the nodes in gray color are swapped during to keep the heap property.



a. An array in arbitrary order before heap building process;



b. Step 1, The array is mapped to binary tree. The first branch node, which is 16 is examined;



c. Step 2, 16 is the largest element in current sub tree, next is to check node with value 2;

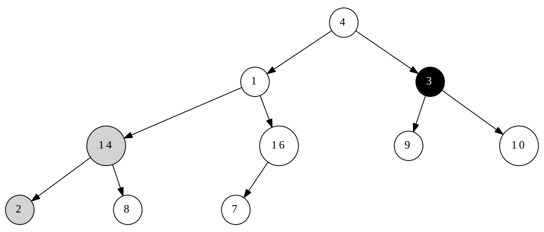
Figure 8.3: Build a heap from the arbitrary array. Gray nodes are changed in each step, black node will be processed next step.

8.2.4 Basic heap operations

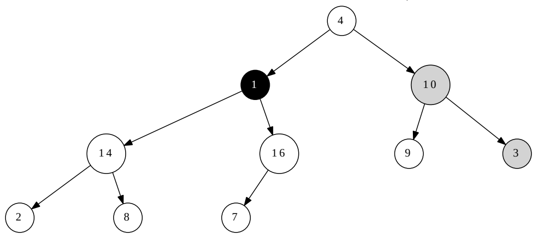
From the generic definition of heap (not necessarily the binary heap), It's essential to provides basic operations so that user can access the data and modify it.

The most important operations include accessing the top element (find the minimum or maximum one), popping the top element from the heap, finding the top n elements, decreasing a key (this is for min-heap, and it is increasing a key for max-heap), and insertion.

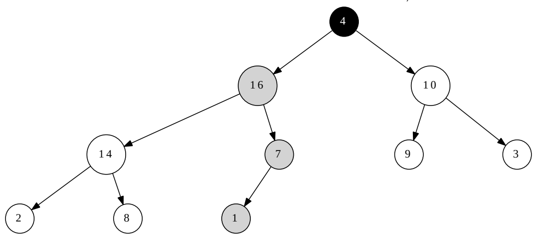
For the binary tree, most of operations are bound to $O(\lg n)$ in worst-case, some of them, such as top is $O(1)$ constant time.



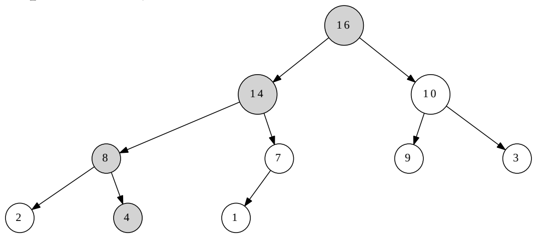
d. Step 3, 14 is the largest value in the sub-tree, swap 14 and 2; next is to check node with value 3;



e. Step 4, 10 is the largest value in the sub-tree, swap 10 and 3; next is to check node with value 1;



f. Step 5, 16 is the largest value in current node, swap 16 and 1 first; then similarly, swap 1 and 7; next is to check the root node with value 4;



g. Step 6, Swap 4 and 16, then swap 4 and 14, and then swap 4 and 8; And the whole build process finish.

Figure 8.4: Build a heap from the arbitrary array. Gray nodes are changed in each step, black node will be processed next step.

Access the top element (the minimum)

We need provide a way to access the top element efficiently. For the binary tree realization, it is the root stores the minimum (maximum) value. This is the first element in the array.

```
1: function TOP($A$)
2:     return $A[1]$
```

This operation is trivial. It takes $O(1)$ time. Here we skip the error handling for empty case. If the heap is empty, one option is to raise an error.

Heap Pop

Pop operation is more complex than accessing the top, because the heap property has to be maintained after the top element is removed.

The solution is to apply HEAPIFY algorithm to the next element after the root is removed.

One simple but slow method based on this idea looks like the following.

```
1: function POP-SLOW($A$)
2:     $x \leftarrow$ TOP($A$)
3:     REMOVE($A$, 1)
4:     if $A$ is not empty then
5:         HEAPIFY($A$, 1)
6:     return $x$
```

This algorithm firstly records the top element in x, then it removes the first element from the array, the size of this array is reduced by one. After that if the array isn't empty, HEAPIFY will applied to the new array from the first element (It was previously the second one).

Removing the first element from array takes $O(n)$ time, where n is the length of the array. This is because we need shift all the rest elements one by one. This bottle neck slows the whole algorithm to linear time.

In order to solve this problem, one alternative is to swap the first element with the last one in the array, then shrink the array size by one.

```
1: function POP($A$)
2:     $x \leftarrow$ TOP($A$)
3:     $n \leftarrow$ HEAP-SIZE($A$)
4:     EXCHANGE $A[1] \leftrightarrow A[n]$
5:     REMOVE($A, n$)
6:     if $A$ is not empty then
7:         HEAPIFY($A$, 1)
8:     return $x$
```

Removing the last element from the array takes only constant $O(1)$ time, and HEAPIFY is bound to $O(\lg n)$. Thus the whole algorithm performs in $O(\lg n)$ time. The following example ANSI C program implements this algorithm[1].

```
Key pop(Key∗ a, int n, Less lt) {
    swap(a, 0, --n);
    heapify(a, 0, n, lt);
```

[1]This program does not actually remove the last element, it reuse the last cell to store the popped result

```
    return a[n];
}
```

Since the top element is removed from the array, instead swapping, this program overwrites it with the last element before applying HEAPIFY.

Find the top k elements

With pop defined, it is easy to find the top k elements from array. we can build a max-heap from the array, then perform pop operation k times.

```
1: function TOP-K($A, k$)
2:     $R \leftarrow \Phi$
3:     BUILD-HEAP($A$)
4:     for $i \leftarrow 1$ to MIN(k, LENGTH($A$)) do
5:         APPEND($R$, POP($A$))
6:     return $R$
```

If k is greater than the length of the array, we need return the whole array as the result. That's why it calls the MIN function to determine the number of loops.

Below example Python program implements the top-k algorithm.

```
def top_k(x, k, less_p = MIN_HEAP):
    build_heap(x, less_p)
    return [heap_pop(x, less_p) for _ in range(min(k, len(x)))]
```

Decrease key

Heap can be used to implement priority queue. It is important to support key modification in heap. One typical operation is to increase the priority of a tasks so that it can be performed earlier.

Here we present the decrease key operation for a min-heap. The corresponding operation is increase key for max-heap. Figure 8.5 illustrate such a case for a max-heap. The key of the 9-th node is increased from 4 to 15.

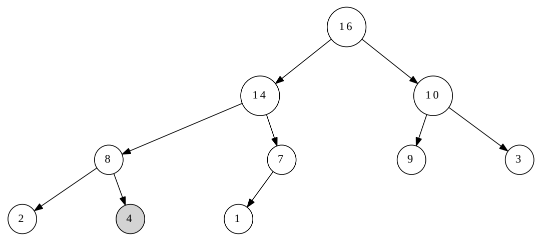
Once a key is decreased in a min-heap, it may make the node conflict with the heap property, that the key may be less than some ancestor. In order to maintain the invariant, the following auxiliary algorithm is defined to resume the heap property.

```
1: function HEAP-FIX($A, i$)
2:     while $i > 1 \wedge A[i] < A[$ PARENT($i$) $]$ do
3:         EXCHANGE $A[i] \leftrightarrow A[$ PARENT($i$) $]$
4:         $i \leftarrow$ PARENT($i$)
```

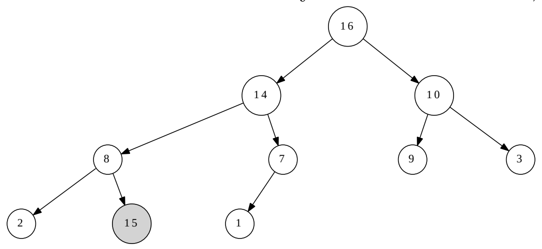
This algorithm repeatedly compares the keys of parent node and current node. It swap the nodes if the parent contains the smaller key. This process is performed from the current node towards the root node till it finds that the parent node holds the smaller key.

With this auxiliary algorithm, decrease key can be realized as below.

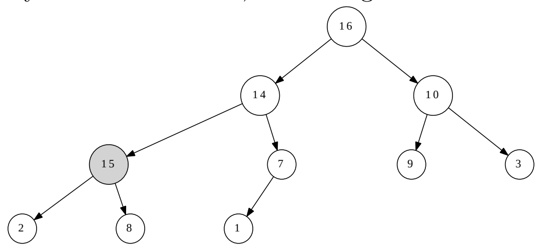
```
1: function DECREASE-KEY($A, i, k$)
2:     if $k < A[i]$ then
3:         $A[i] \leftarrow k$
4:         HEAP-FIX($A, i$)
```



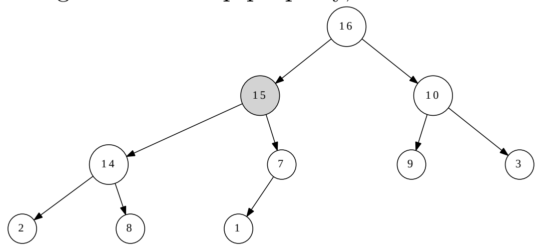
a. The 9-th node with key 4 will be modified;



b. The key is modified to 15, which is greater than its parent;



c. According the max-heap property, 8 and 15 are swapped.



d. Since 15 is greater than its parent 14, they are swapped. After that, because 15 is less than 16, the process terminates.

Figure 8.5: Example process when increase a key in a max-heap.

This algorithm is only triggered when the new key is less than the original key. The performance is bound to $O(\lg n)$. Below example ANSI C program implements the algorithm.

```
void heap_fix(Key* a, int i, Less lt) {
    while (i > 0 && lt(a[i], a[PARENT(i)])) {
        swap(a, i, PARENT(i));
        i = PARENT(i);
    }
}

void decrease_key(Key* a, int i, Key k, Less lt) {
    if (lt(k, a[i])) {
        a[i] = k;
        heap_fix(a, i, lt);
    }
}
```

Insertion

In some materials like [2], insertion is implemented by using DECREASE-KEY. A new node with ∞ as key is created. According to the min-heap property, it should be the last element in the under ground array. After that, the key is decreased to the value to be inserted, and DECREASE-KEY is called to fix any violation to the heap property.

Alternatively, we can reuse HEAP-FIX to implement insertion. The new key is directly appended at the end of the array, and the HEAP-FIX is applied to this new node.

```
1: function HEAP-PUSH($A, k$)
2:     APPEND($A, k$)
3:     HEAP-FIX($A, |A|$)
```

The following example Python program implements the heap insertion algorithm.

```
def heap_insert(x, key, less_p = MIN_HEAP):
    i = len(x)
    x.append(key)
    heap_fix(x, i, less_p)
```

8.2.5 Heap sort

Heap sort is interesting application of heap. According to the heap property, the min(max) element can be easily accessed by from the top of the heap. A straightforward way to sort a list of values is to build a heap from them, then continuously pop the smallest element till the heap is empty.

The algorithm based on this idea can be defined like below.

```
1: function HEAP-SORT($A$)
2:     $R \leftarrow \Phi$
3:     BUILD-HEAP($A$)
4:     while $A \neq \Phi$ do
5:         APPEND($R$, HEAP-POP($A$))
```

```
6:     return $R$
```

The following Python example program implements this definition.

```
def heap_sort(x, less_p = MIN_HEAP):
    res = []
    build_heap(x, less_p)
    while x!=[]:
        res.append(heap_pop(x, less_p))
    return res
```

When sort n elements, the BUILD-HEAP is bound to $O(n)$. Since pop is $O(\lg n)$, and it is called n times, so the overall sorting takes $O(n \lg n)$ time to run. Because we use another list to hold the result, the space requirement is $O(n)$.

Robert. W. Floyd found a fast implementation of heap sort. The idea is to build a max-heap instead of min-heap, so the first element is the biggest one. Then the biggest element is swapped with the last element in the array, so that it is in the right position after sorting. As the last element becomes the new top, it may violate the heap property. We can shrink the heap size by one and perform HEAPIFY to resume the heap property. This process is repeated till there is only one element left in the heap.

```
1: function HEAP-SORT($A$)
2:     BUILD-MAX-HEAP($A$)
3:     while $|A| > 1$ do
4:         EXCHANGE $A[1] \leftrightarrow A[n]$
5:         $|A| \leftarrow |A| - 1$
6:         HEAPIFY($A, 1$)
```

This is in-place algorithm, it needn't any extra spaces to hold the result. The following. The following ANSI C example code implements this algorithm.

```
void heap_sort(Key* a, int n) {
    build_heap(a, n, notless);
    while(n > 1) {
        swap(a, 0, --n);
        heapify(a, 0, n, notless);
    }
}
```

Exercise 8.1

- Somebody considers one alternative to realize in-place heap sort. Take sorting the array in ascending order as example, the first step is to build the array as a minimum heap A, but not the maximum heap like the Floyd's method. After that the first element a_1 is in the correct place. Next, treat the rest $\{a_2, a_3, ..., a_n\}$ as a new heap, and perform HEAPIFY to them from a_2 for these $n - 1$ elements. Repeating this advance and HEAPIFY step from left to right would sort the array. The following example ANSI C code illustrates this idea. Is this solution correct? If yes, prove it; if not, why?

```
void heap_sort(Key* a, int n) {
    build_heap(a, n, less);
    while(--n)
```

```
        heapify(++a, 0, n, less);
}
```

- Because of the same reason, can we perform HEAPIFY from left to right k times to realize in-place top-k algorithm like below ANSI C code?

```
int tops(int k, Key* a, int n, Less lt) {
    build_heap(a, n, lt);
    for (k = MIN(k, n) - 1; k; --k)
        heapify(++a, 0, --n, lt);
    return k;
}
```

8.3 Leftist heap and Skew heap, the explicit binary heaps

Instead of using implicit binary tree by array, it is natural to consider why we can't use explicit binary tree to realize heap?

There are some problems must be solved if we turn into explicit binary tree as the under ground data structure.

The first problem is about the HEAP-POP or DELETE-MIN operation. Consider the binary tree is represented in form of left, key, and right as (L, k, R), which is shown in figure 8.6

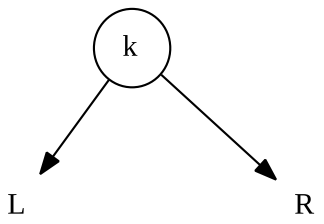


Figure 8.6: A binary tree, all elements in children are smaller than k.

If k is the top element, all elements in left and right children are less than k. After k is popped, only left and right children are left. They have to be merged to a new tree. Since heap property should be maintained after merge, the new root is still the smallest element.

Because both left and right children are binary trees conforming heap property, the two trivial cases can be defined immediately.

$$merge(H_1, H_2) = \begin{cases} H_2 & : H_1 = \Phi \\ H_1 & : H_2 = \Phi \\ ? & : otherwise \end{cases}$$

Where Φ means empty heap.

If neither left nor right child is empty, because they all fit heap property, the top elements of them are all the minimum respectively. We can compare these two roots, and select the smaller as the new root of the merged heap.

For instance, let $L = (A, x, B)$ and $R = (A', y, B')$, where A, A', B, and B' are all sub trees. If $x < y$, x will be the new root. We can either keep A, and recursively merge B and R; or keep B, and merge A and R, so the new heap can be one of the following.

- $(merge(A, R), x, B)$
- $(A, x, merge(B, R))$

Both are correct. One simplified solution is to only merge the right sub tree. *Leftist* tree provides a systematically approach based on this idea.

8.3.1 Definition

The heap implemented by Leftist tree is called Leftist heap. Leftist tree is first introduced by C. A. Crane in 1972[6].

Rank (S-value)

In Leftist tree, a rank value (or S value) is defined for each node. Rank is the distance to the nearest external node. Where external node is the NIL concept extended from the leaf node.

For example, in figure 8.7, the rank of NIL is defined 0, consider the root node 4, The nearest leaf node is the child of node 8. So the rank of root node 4 is 2. Because node 6 and node 8 both only contain NIL, so their rank values are 1. Although node 5 has non-NIL left child, However, since the right child is NIL, so the rank value, which is the minimum distance to leaf is still 1.

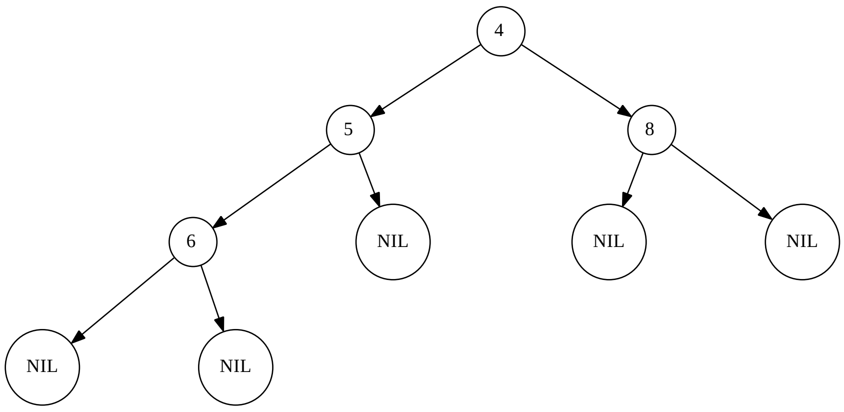


Figure 8.7: $rank(4) = 2$, $rank(6) = rank(8) = rank(5) = 1$.

Leftist property

With rank defined, we can create a strategy when merging.

- Every time when merging, we always merge to right child; Denote the rank of the new right sub tree as r_r;
- Compare the ranks of the left and right children, if the rank of left sub tree is r_l and $r_l < r_r$, we swap the left and the right children.

We call this ‘Leftist property’. In general, a Leftist tree always has the shortest path to some external node on the right.

Leftist tree tends to be very unbalanced, However, it ensures important property as specified in the following theorem.

Theorem 8.3.1. *If a Leftist tree T contains n internal nodes, the path from root to the rightmost external node contains at most $\lfloor \log(n+1) \rfloor$ nodes.*

We skip the proof here, readers can refer to [7] and [1] for more information. With this theorem, algorithms operate along this path are all bound to $O(\lg n)$.

We can reuse the binary tree definition, and augment with a rank field to define the Leftist tree. For example in form of (r, k, L, R) for non-empty case. The below Haskell defines Leftist tree.

```
data LHeap a = E -- Empty
             | Node Int a (LHeap a) (LHeap a) -- rank, element, left, right
```

For empty tree, the rank is defined as zero. Otherwise, it’s the value of the augmented field. A $rank(H)$ function can be given to cover both cases.

$$rank(H) = \begin{cases} 0 & : H = \Phi \\ r & : otherwise, H = (r, k, L, R) \end{cases} \tag{8.1}$$

Here is the example Haskell rank function.

```
rank E = 0
rank (Node r _ _ _) = r
```

In the rest of this section, we denote $rank(H)$ as r_H

8.3.2 Merge

In order to realize ‘merge’, we need develop the auxiliary algorithm to compare the ranks and swap the children if necessary.

$$mk(k, A, B) = \begin{cases} (r_A + 1, k, B, A) & : r_A < r_B \\ (r_B + 1, k, A, B) & : otherwise \end{cases} \tag{8.2}$$

This function takes three arguments, a key and two sub trees A, and B. if the rank of A is smaller, it builds a bigger tree with B as the left child, and A as the right child. It increment the rank of A by 1 as the rank of the new tree; Otherwise if B holds the smaller rank, then A is set as the left child, and B becomes the right. The resulting rank is $r_b + 1$.

The reason why rank need be increased by one is because there is a new key added on top of the tree. It causes the rank increasing.

Denote the key, the left and right children for H_1 and H_2 as k_1, L_1, R_1, and k_2, L_2, R_2 respectively. The $merge(H_1, H_2)$ function can be completed by using this auxiliary tool as below

$$merge(H_1, H_2) = \begin{cases} H_2 & : H_1 = \Phi \\ H_1 & : H_2 = \Phi \\ mk(k_1, L_1, merge(R_1, H_2)) & : k_1 < k_2 \\ mk(k_2, L_2, merge(H_1, R_2)) & : otherwise \end{cases} \tag{8.3}$$

The *merge* function is always recursively called on the right side, and the Leftist property is maintained. These facts ensure the performance being bound to $O(\lg n)$.

The following Haskell example code implements the merge program.

```
merge E h = h
merge h E = h
merge h1@(Node _ x l r) h2@(Node _ y l' r') =
    if x < y then makeNode x l (merge r h2)
    else makeNode y l' (merge h1 r')

makeNode x a b = if rank a < rank b then Node (rank a + 1) x b a
                 else Node (rank b + 1) x a b
```

Merge operation in implicit binary heap by array

Implicit binary heap by array performs very fast in most cases, and it fits modern computer with cache technology well. However, merge is the algorithm bounds to $O(n)$ time. The typical realization is to concatenate two arrays together and make a heap for this array [13].

```
1: function MERGE-HEAP($A, B$)
2:     $C \leftarrow$ CONCAT($A, B$)
3:     BUILD-HEAP($C$)
```

8.3.3 Basic heap operations

Most of the basic heap operations can be implemented with *merge* algorithm defined above.

Top and pop

Because the smallest element is always held in root, it's trivial to find the minimum value. It's constant $O(1)$ operation. Below equation extracts the root from non-empty heap $H = (r, k, L, R)$. The error handling for empty case is skipped here.

$$top(H) = k \tag{8.4}$$

For pop operation, firstly, the top element is removed, then left and right children are merged to a new heap.

$$pop(H) = merge(L, R) \tag{8.5}$$

Because it calls *merge* directly, the pop operation on Leftist heap is bound to $O(\lg n)$.

Insertion

To insert a new element, one solution is to create a single leaf node with the element, and then merge this leaf node to the existing Leftist tree.

$$insert(H, k) = merge(H, (1, k, \Phi, \Phi)) \tag{8.6}$$

It is $O(\lg n)$ algorithm since insertion also calls *merge* directly.

There is a convenient way to build the Leftist heap from a list. We can continuously insert the elements one by one to the empty heap. This can be realized by folding.

$$build(L) = fold(insert, \Phi, L) \tag{8.7}$$

Figure 8.8 shows one example Leftist tree built in this way.

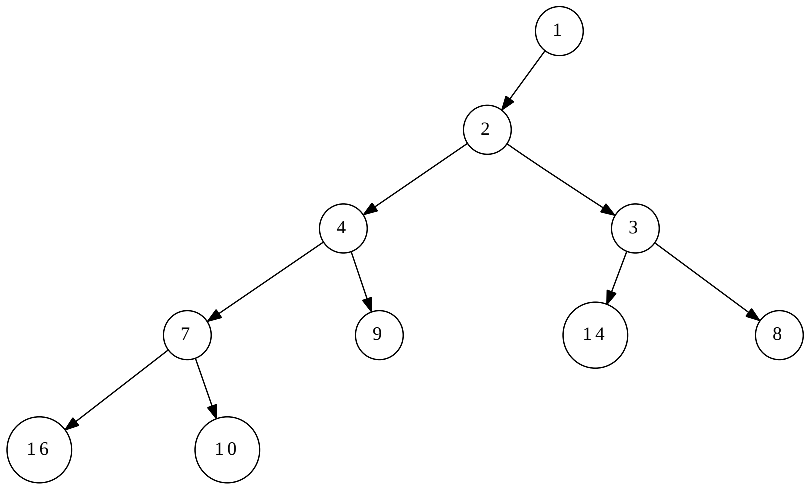


Figure 8.8: A Leftist tree built from list $\{9, 4, 16, 7, 10, 2, 14, 3, 8, 1\}$.

The following example Haskell code gives reference implementation for the Leftist tree operations.

```
insert h x = merge (Node 1 x E E) h

findMin (Node _ x _ _) = x

deleteMin (Node _ _ l r) = merge l r

fromList = foldl insert E
```

8.3.4 Heap sort by Leftist Heap

With all the basic operations defined, it's straightforward to implement heap sort. We can firstly turn the list into a Leftist heap, then continuously extract the minimum element from it.

$$sort(L) = heapSort(build(L)) \tag{8.8}$$

$$heapSort(H) = \begin{cases} \Phi & : H = \Phi \\ \{top(H)\} \cup heapSort(pop(H)) & : otherwise \end{cases} \tag{8.9}$$

Because pop is logarithm operation, and it is recursively called n times, this algorithm takes $O(n \lg n)$ time in total. The following Haskell example program implements heap sort with Leftist tree.

```
heapSort = hsort ◦ fromList where
    hsort E = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

8.3.5 Skew heaps

Leftist heap performs poor in some cases. Figure 8.9 shows one example. The Leftist tree is built by folding on list $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$.

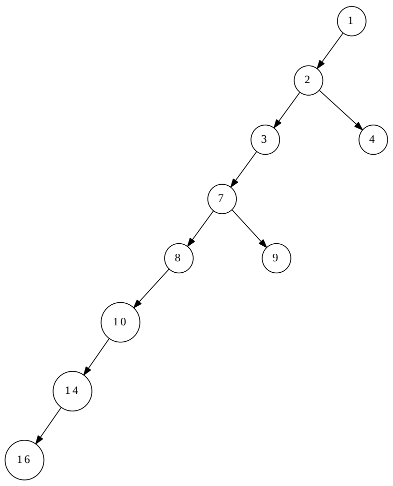


Figure 8.9: A very unbalanced Leftist tree build from list $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$.

The binary tree almost turns to a linked-list. The worst case happens when feed the ordered list to build Leftist tree. Because the tree downgrades to linked-list, the performance drops from $O(\lg n)$ to $O(n)$.

Skew heap (or *self-adjusting heap*) simplifies Leftist heap realization and can solve the performance issue[9] [10].

When construct the Leftist heap, we swap the left and right children during merge if the rank on left side is less than the right side. This comparison-and-swap strategy doesn't work when either sub tree has only one child. Because in such case, the rank of the sub tree is always 1 no matter how big it is. A 'Brute-force' approach is to swap the left and right children every time when merge. This idea leads to Skew heap.

Definition of Skew heap

Skew heap is the heap realized with Skew tree. Skew tree is a special binary tree. The minimum element is stored in root. Every sub tree is also a skew tree.

It needn't keep the rank (or S-value) field. We can reuse the binary tree definition for Skew heap. The tree is either empty, or in a pre-order form (k, L, R). Below Haskell code defines Skew heap like this.

```
data SHeap a = E -- Empty
             | Node a (SHeap a) (SHeap a) -- element, left, right
```

Merge

The merge algorithm tends to be very simple. When merge two non-empty Skew trees, we compare the roots, and pick the smaller one as the new root, then the other tree contains the bigger element is merged onto one sub tree, finally, the tow children are swapped. Denote $H_1 = (k_1, L_1, R_1)$ and $H_2 = (k_2, L_2, R_2)$ if they are not empty. if $k_1 < k_2$ for instance, select k_1 as the new root. We can either merge H_2 to L_1, or merge H_2 to R_1. Without loss of generality, let's merge to R_1. And after swapping the two children, the final result is $(k_1, merge(R_1, H_2), L_1)$. Take account of edge cases, the merge algorithm is defined as the following.

$$merge(H_1, H_2) = \begin{cases} H_1 & : H_2 = \Phi \\ H_2 & : H_1 = \Phi \\ (k_1, merge(R_1, H_2), L_1) & : k_1 < k_2 \\ (k_2, merge(H_1, R_2), L_2) & : otherwise \end{cases} \tag{8.10}$$

All the rest operations, including insert, top and pop are all realized as same as the Leftist heap by using merge, except that we needn't the rank any more.

Translating the above algorithm into Haskell yields the following example program.

```
merge E h = h
merge h E = h
merge h1@(Node x l r) h2@(Node y l' r') =
    if x < y then Node x (merge r h2) l
    else Node y (merge h1 r') l'

insert h x = merge (Node x E E) h

findMin (Node x _ _) = x

deleteMin (Node _ l r) = merge l r
```

Different from the Leftist heap, if we feed ordered list to Skew heap, it can build a fairly balanced binary tree as illustrated in figure 8.10.

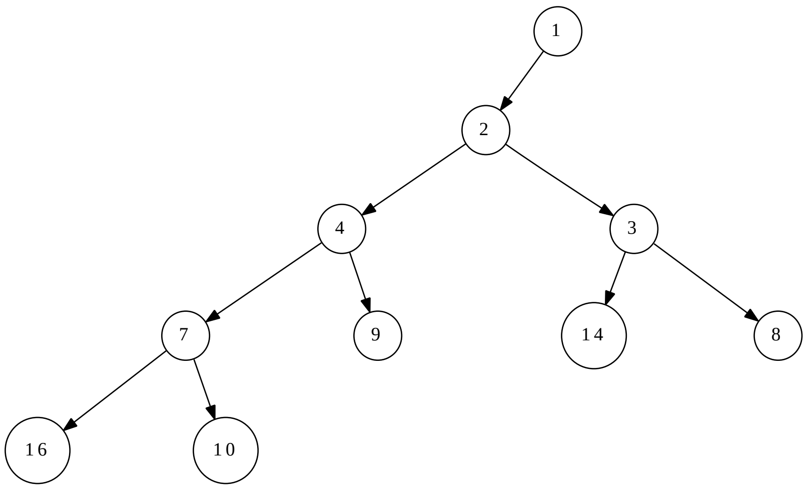


Figure 8.10: Skew tree is still balanced even the input is an ordered list $\{1, 2, ..., 10\}$.

8.4 Splay heap

The Leftist heap and Skew heap show the fact that it's quite possible to realize heap data structure with explicit binary tree. Skew heap gives one method to solve the tree balance problem. Splay heap on the other hand, use another method to keep the tree balanced.

The binary trees used in Leftist heap and Skew heap are not Binary Search tree (BST). If we turn the underground data structure to binary search tree, the minimum(or maximum) element is not root any more. It takes $O(\lg n)$ time to find the minimum(or maximum) element.

Binary search tree becomes inefficient if it isn't well balanced. Most operations degrade to $O(n)$ in the worst case. Although red-black tree can be used to realize binary heap, it's overkill. Splay tree provides a light weight implementation with acceptable dynamic balancing result.

8.4.1 Definition

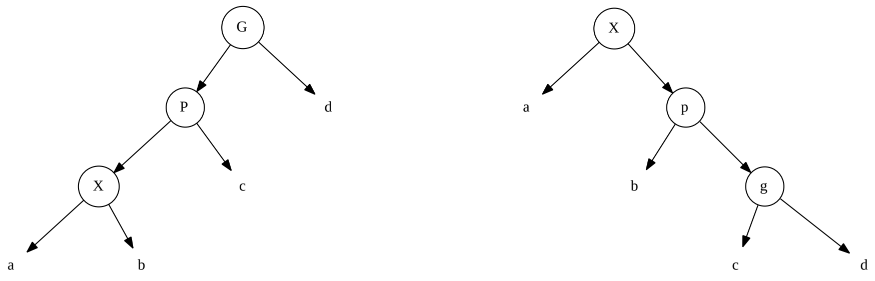
Splay tree uses cache-like approach. It keeps rotating the current access node close to the top, so that the node can be accessed fast next time. It defines such kinds of operation as "Splay". For the unbalanced binary search tree, after several splay operations, the tree tends to be more and more balanced. Most basic operations of Splay tree perform in amortized $O(\lg n)$ time. Splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985[11] [12].

Splaying

There are two methods to do splaying. The first one need deal with many different cases, but can be implemented fairly easy with pattern matching. The second one has a uniformed form, but the implementation is complex.

Denote the node currently being accessed as X, the parent node as P, and the grand parent node as G (If there are). There are 3 steps for splaying. Each step contains 2 symmetric cases. For illustration purpose, only one case is shown for each step.

- *Zig-zig step.* As shown in figure 8.11, in this case, X and P are children on the same side, either both on the left or on the right. By rotating 2 times, X becomes the new root.



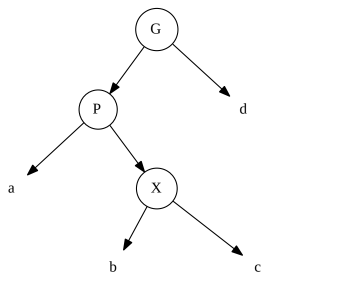
(a) X and P are either left or right children. (b) X becomes new root after rotating 2 times.

Figure 8.11: Zig-zig case.

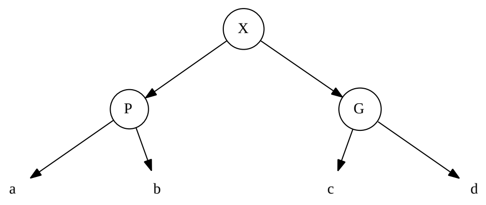
- *Zig-zag step.* As shown in figure 8.12, in this case, X and P are children on different sides. X is on the left, P is on the right. Or X is on the right, P is on the left. After rotation, X becomes the new root, P and G are siblings.

- *Zig step.* As shown in figure 8.13, in this case, P is the root, we rotate the tree, so that X becomes new root. This is the last step in splay operation.

Although there are 6 different cases, they can be handled in the environments support pattern matching. Denote the binary tree in form (L, k, R). When

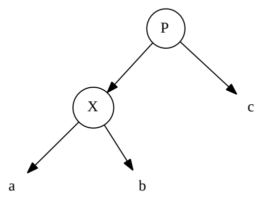


(a) X and P are children on different sides.

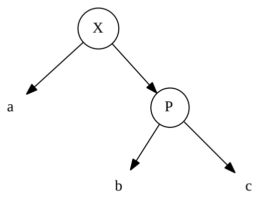


(b) X becomes new root. P and G are siblings.

Figure 8.12: Zig-zag case.



(a) P is the root.



(b) Rotate the tree to make X be new root.

Figure 8.13: Zig case.

access key Y in tree T, the splay operation can be defined as below.

$$splay(T, X) = \begin{cases} (a, X, (b, P, (c, G, d))) & : T = (((a, X, b), P, c), G, d), X = Y \\ (((a, G, b), P, c), X, d) & : T = (a, G, (b, P, (c, X, d))), X = Y \\ ((a, P, b), X, (c, G, d)) & : T = (a, P, (b, X, c), G, d), X = Y \\ ((a, G, b), X, (c, P, d)) & : T = (a, G, ((b, X, c), P, d)), X = Y \\ (a, X, (b, P, c)) & : T = ((a, X, b), P, c), X = Y \\ ((a, P, b), X, c) & : T = (a, P, (b, X, c)), X = Y \\ T & : otherwise \end{cases} \tag{8.11}$$

The first two clauses handle the 'zig-zig' cases; the next two clauses handle the 'zig-zag' cases; the last two clauses handle the zig cases. The tree aren't changed for all other situations.

The following Haskell program implements this splay function.

```
data STree a = E -- Empty
             | Node (STree a) a (STree a) -- left, key, right

-- zig-zig
splay t@(Node (Node (Node a x b) p c) g d) y =
    if x == y then Node a x (Node b p (Node c g d)) else t
splay t@(Node a g (Node b p (Node c x d))) y =
    if x == y then Node (Node (Node a g b) p c) x d else t
-- zig-zag
splay t@(Node (Node a p (Node b x c)) g d) y =
    if x == y then Node (Node a p b) x (Node c g d) else t
splay t@(Node a g (Node (Node b x c) p d)) y =
    if x == y then Node (Node a g b) x (Node c p d) else t
-- zig
splay t@(Node (Node a x b) p c) y = if x == y then Node a x (Node b p c) else t
splay t@(Node a p (Node b x c)) y = if x == y then Node (Node a p b) x c else t
-- otherwise
splay t _ = t
```

With splay operation defined, every time when insert a new key, we call the splay function to adjust the tree. If the tree is empty, the result is a leaf; otherwise we compare this key with the root, if it is less than the root, we recursively insert it into the left child, and perform splaying after that; else the key is inserted into the right child.

$$insert(T, x) = \begin{cases} (\Phi, x, \Phi) & : T = \Phi \\ splay((insert(L, x), k, R), x) & : T = (L, k, R), x < k \\ splay(L, k, insert(R, x)) & : otherwise \end{cases} \tag{8.12}$$

The following Haskell program implements this insertion algorithm.

```
insert E y = Node E y E
insert (Node l x r) y
    | x > y     = splay (Node (insert l y) x r) y
    | otherwise = splay (Node l x (insert r y)) y
```

Figure 8.14 shows the result of using this function. It inserts the ordered elements $\{1, 2, ..., 10\}$ one by one to the empty tree. This would build a very poor result which downgrade to linked-list with normal binary search tree. The splay method creates more balanced result.

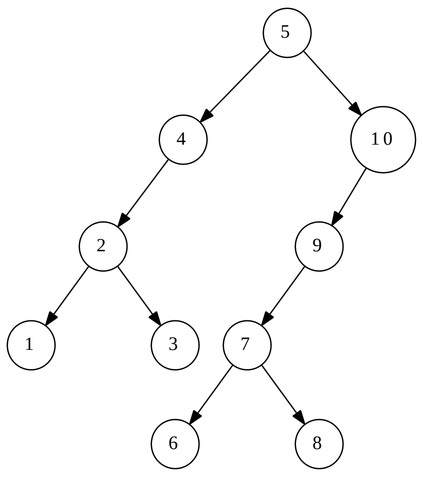


Figure 8.14: Splaying helps improving the balance.

Okasaki found a simple rule for Splaying [6]. Whenever we follow two left branches, or two right branches continuously, we rotate the two nodes.

Based on this rule, splaying can be realized in such a way. When we access node for a key x (can be during the process of inserting a node, or looking up a node, or deleting a node), if we traverse two left branches or two right branches, we partition the tree in two parts L and R, where L contains all nodes smaller than x, and R contains all the rest. We can then create a new tree (for instance in insertion), with x as the root, L as the left child, and R being the right child.

The partition process is recursive, because it will splay its children as well.

$$
partition(T, p) = \begin{cases}
(\Phi, \Phi) & : T = \Phi \\
(T, \Phi) & : T = (L, k, R) \land R = \Phi \\
((L, k, L'), k', A, B) & : \begin{array}{l} T = (L, k, (L', k', R')) \\ k < p, k' < p \\ (A, B) = partition(R', p) \end{array} \\
((L, k, A), (B, k', R')) & : \begin{array}{l} T = (L, K, (L', k', R')) \\ k < p \leq k' \\ (A, B) = partition(L', p) \end{array} \\
(\Phi, T) & : T = (L, k, R) \land L = \Phi \\
(A, (L', k', (R', k, R)) & : \begin{array}{l} T = ((L', k', R'), k, R) \\ p \leq k, p \leq k' \\ (A, B) = partition(L', p) \end{array} \\
((L', k', A), (B, k, R)) & : \begin{array}{l} T = ((L', k', R'), k, R) \\ k' \leq p \leq k \\ (A, B) = partition(R', p) \end{array}
\end{cases} \tag{8.13}
$$

Function $partition(T, p)$ takes a tree T, and a pivot p as arguments. The first clause is edge case. The partition result for empty is a pair of empty left and right trees. Otherwise, denote the tree as (L, k, R). we need compare the pivot p and the root k. If $k < p$, there are two sub-cases. one is trivial case that R is empty. According to the property of binary search tree, All elements are less than p, so the result pair is (T, Φ); For the other case, $R = (L', k', R')$, we need further compare k' with the pivot p. If $k' < p$ is also true, we recursively partition R' with the pivot, all the elements less than p in R' is held in tree A, and the rest is in tree B. The result pair can be composed with two trees, one is $((L, k, L'), k', A)$; the other is B. If the key of the right sub tree is not less than the pivot. we recursively partition L' with the pivot to give the intermediate pair (A, B), the final pair trees can be composed with (L, k, A) and (B, k', R'). There is a symmetric cases for $p \leq k$. They are handled in the last three clauses.

Translating the above algorithm into Haskell yields the following partition program.

```
partition E _ = (E, E)
partition t@(Node l x r) y
    | x < y =
        case r of
          E → (t, E)
          Node l' x' r' →
              if x' < y then
                  let (small, big) = partition r' y in
                  (Node (Node l x l') x' small, big)
              else
                  let (small, big) = partition l' y in
                  (Node l x small, Node big x' r')
```

```
  | otherwise =
      case l of
        E → (E, t)
        Node l' x' r' →
            if y < x' then
                let (small, big) = partition l' y in
                (small, Node l' x' (Node r' x r))
            else
                let (small, big) = partition r' y in
                (Node l' x' small, Node big x r)
```

Alternatively, insertion can be realized with *partition* algorithm. When insert a new element k into the splay heap T, we can first partition the heap into two trees, L and R. Where L contains all nodes smaller than k, and R contains the rest. We then construct a new node, with k as the root and L, R as the children.

$$insert(T, k) = (L, k, R), (L, R) = partition(T, k) \tag{8.14}$$

The corresponding Haskell example program is as the following.

```
insert t x = Node small x big where (small, big) = partition t x
```

Top and pop

Since splay tree is just a special binary search tree, the minimum element is stored in the left most node. We need keep traversing the left child to realize the top operation. Denote the none empty tree $T = (L, k, R)$, the $top(T)$ function can be defined as below.

$$top(T) = \left\{ \begin{array}{rcl} k & : & L = \Phi \\ top(L) & : & otherwise \end{array} \right. \tag{8.15}$$

This is exactly the $min(T)$ algorithm for binary search tree.

For pop operation, the algorithm need remove the minimum element from the tree. Whenever there are two left nodes traversed, the splaying operation should be performed.

$$pop(T) = \left\{ \begin{array}{rcl} R & : & T = (\Phi, k, R) \\ (R', k, R) & : & T = ((\Phi, k', R')k, R) \\ (pop(L'), k', (R', k, R)) & : & T = ((L', k', R'), k, R) \end{array} \right. \tag{8.16}$$

Note that the third clause performs splaying without explicitly call the *partition* function. It utilizes the property of binary search tree directly.

Both the top and pop algorithms are bound to $O(\lg n)$ time because the splay tree is balanced.

The following Haskell example programs implement the top and pop operations.

```
findMin (Node E x _) = x
findMin (Node l x _) = findMin l

deleteMin (Node E x r) = r
deleteMin (Node (Node E x' r') x r) = Node r' x r
deleteMin (Node (Node l' x' r') x r) = Node (deleteMin l') x' (Node r' x r)
```

Merge

Merge is another basic operation for heaps as it is widely used in Graph algorithms. By using the *partition* algorithm, merge can be realized in $O(\lg n)$ time.

When merging two splay trees, for non-trivial case, we can take the root of the first tree as the new root, then partition the second tree with this new root as the pivot. After that we recursively merge the children of the first tree to the partition result. This algorithm is defined as the following.

$$merge(T_1, T_2) = \begin{cases} T_2 & : T_1 = \Phi \\ (merge(L, A), k, merge(R, B)) & : T_1 = (L, k, R), (A, B) = partition(T_2, k) \end{cases} \tag{8.17}$$

If the first heap is empty, the result is definitely the second heap. Otherwise, denote the first splay heap as (L, k, R), we partition T_2 with k as the pivot to yield (A, B), where A contains all the elements in T_2 which are less than k, and B holds the rest. We next recursively merge A with L; and merge B with R as the new children for T_1.

Translating the definition to Haskell gives the following example program.

```
merge E t = t
merge (Node l x r) t = Node (merge l l') x (merge r r')
    where (l', r') = partition t x
```

8.4.2 Heap sort

Since the internal implementation of the Splay heap is completely transparent to the heap interface, the heap sort algorithm can be reused. It means that the heap sort algorithm is generic no matter what the underground data structure is.

8.5 Notes and short summary

In this chapter, we define binary heap more general so that as long as the heap property is maintained, all binary representation of data structures can be used to implement binary heap.

This definition doesn't limit to the popular array based binary heap, but also extends to the explicit binary heaps including Leftist heap, Skew heap and Splay heap. The array based binary heap is particularly convenient for the imperative implementation because it intensely uses random index access which can be mapped to a completely binary tree. It's hard to find directly functional counterpart in this way.

However, by using explicit binary tree, functional implementation can be achieved, most of them have $O(\lg n)$ worst case performance, and some of them even reach $O(1)$ amortize time. Okasaki in [6] shows detailed analysis of these data structures.

In this chapter, only purely functional realization for Leftist heap, Skew heap, and Splay heap are explained, they can all be realized in imperative approaches.

It's very natural to extend the concept from binary tree to k-ary (k-way) tree, which leads to other useful heaps such as Binomial heap, Fibonacci heap and pairing heap. They are introduced in the next chapter.

Exercise 8.2

- Realize the imperative Leftist heap, Skew heap, and Splay heap.

Bibliography

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

[2] Heap (data structure), Wikipedia. http://en.wikipedia.org/wiki/Heap_(data_structure)

[3] Heapsort, Wikipedia. http://en.wikipedia.org/wiki/Heapsort

[4] Chris Okasaki. “Purely Functional Data Structures”. Cambridge university press, (July 1, 1999), ISBN-13: 978-0521663502

[5] Sorting algorithms/Heapsort. Rosetta Code. http://rosettacode.org/wiki/Sorting_algorithms/Heapsort

[6] Leftist Tree, Wikipedia. http://en.wikipedia.org/wiki/Leftist_tree

[7] Bruno R. Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in Java. http://www.brpreiss.com/books/opus5/index.html

[8] Donald E. Knuth. “The Art of Computer Programming. Volume 3: Sorting and Searching.”. Addison-Wesley Professional; 2nd Edition (October 15, 1998). ISBN-13: 978-0201485417. Section 5.2.3 and 6.2.3

[9] Skew heap, Wikipedia. http://en.wikipedia.org/wiki/Skew_heap

[10] Sleator, Daniel Dominic; Jarjan, Robert Endre. “Self-adjusting heaps” SIAM Journal on Computing 15(1):52-69. doi:10.1137/0215004 ISSN 00975397 (1986)

[11] Splay tree, Wikipedia. http://en.wikipedia.org/wiki/Splay_tree

[12] Sleator, Daniel D.; Tarjan, Robert E. (1985), “Self-Adjusting Binary Search Trees”, Journal of the ACM 32(3):652 - 686, doi: 10.1145/3828.3835

[13] NIST, “binary heap”. http://xw2k.nist.gov/dads//HTML/binaryheap.html

Chapter 9

From grape to the world cup, the evolution of selection sort

9.1 Introduction

We have introduced the ‘hello world’ sorting algorithm, insertion sort. In this short chapter, we explain another straightforward sorting method, selection sort. The basic version of selection sort doesn’t perform as good as the divide and conqueror methods, e.g. quick sort and merge sort. We’ll use the same approaches in the chapter of insertion sort, to analyze why it’s slow, and try to improve it by varies of attempts till reach the best bound of comparison based sorting, $O(N \lg N)$, by evolving to heap sort.

The idea of selection sort can be illustrated by a real life story. Consider a kid eating a bunch of grapes. There are two types of children according to my observation. One is optimistic type, that the kid always eats the biggest grape he/she can ever find; the other is pessimistic, that he/she always eats the smallest one.

The first type of kids actually eat the grape in an order that the size decreases monotonically; while the other eat in a increase order. The kid *sorts* the grapes in order of size in fact, and the method used here is selection sort.

Based on this idea, the algorithm of selection sort can be directly described as the following.

In order to sort a series of elements:

- The trivial case, if the series is empty, then we are done, the result is also empty;
- Otherwise, we find the smallest element, and append it to the tail of the result;

Note that this algorithm sorts the elements in increase order; It’s easy to sort in decrease order by picking the biggest element instead; We’ll introduce about passing a comparator as a parameter later on.

Figure 9.1: Always picking the smallest grape.

This description can be formalized to a equation.

$$sort(A) = \begin{cases} \Phi & : A = \Phi \\ \{m\} \cup sort(A') & : otherwise \end{cases} \tag{9.1}$$

Where m is the minimum element among collection A, and A' is all the rest elements except m:

$$m = min(A)$$
$$A' = A - \{m\}$$

We don't limit the data structure of the collection here. Typically, A is an array in imperative environment, and a list (singly linked-list particularly) in functional environment, and it can even be other data struture which will be introduced later.

The algorithm can also be given in imperative manner.

```
function SORT($A$)
    $X \leftarrow \Phi$
    while $A \neq \Phi$ do
        $x \leftarrow$ MIN($A$)
        $A \leftarrow$ DEL($A, x$)
        $X \leftarrow$ APPEND($X, x$)
    return $X$
```

Figure 9.2 depicts the process of this algorithm.

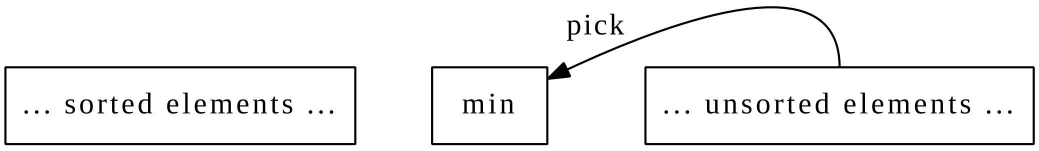


Figure 9.2: The left part is sorted data, continuously pick the minimum element in the rest and append it to the result.

We just translate the very original idea of 'eating grapes' line by line without considering any expense of time and space. This realization stores the result in

X, and when an selected element is appended to X, we delete the same element from A. This indicates that we can change it to 'in-place' sorting to reuse the spaces in A.

The idea is to store the minimum element in the first cell in A (we use term 'cell' if A is an array, and say 'node' if A is a list); then store the second minimum element in the next cell, then the third cell, ...

One solution to realize this sorting strategy is swapping. When we select the i-th minimum element, we swap it with the element in the i-th cell:

```
function SORT($A$)
    for $i \leftarrow 1$ to $|A|$ do
        $m \leftarrow$ MIN($A[i...]$)
        EXCHANGE $A[i] \leftrightarrow m$
```

Denote $A = \{a_1, a_2, ..., a_N\}$. At any time, when we process the i-th element, all elements before i, as $\{a_1, a_2, ..., a_{i-1}\}$ have already been sorted. We locate the minimum element among the $\{a_i, a_{i+1}, ..., a_N\}$, and exchange it with a_i, so that the i-th cell contains the right value. The process is repeatedly executed until we arrived at the last element.

This idea can be illustrated by figure 9.3.

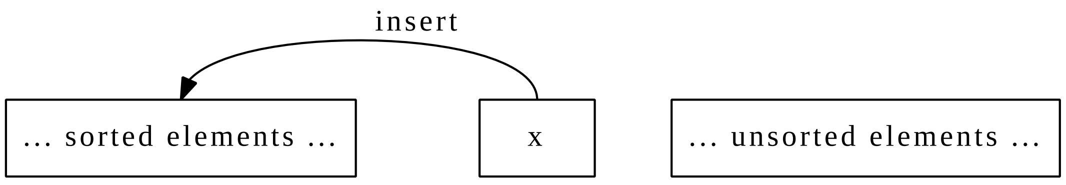


Figure 9.3: The left part is sorted data, continuously pick the minimum element in the rest and put it to the right position.

9.2 Finding the minimum

We haven't completely realized the selection sort, because we take the operation of finding the minimum (or the maximum) element as a black box. It's a puzzle how does a kid locate the biggest or the smallest grape. And this is an interesting topic for computer algorithms.

The easiest but not so fast way to find the minimum in a collection is to perform a scan. There are several ways to interpret this scan process. Consider that we want to pick the biggest grape. We start from any grape, compare it with another one, and pick the bigger one; then we take a next grape and compare it with the one we selected so far, pick the bigger one and go on the take-and-compare process, until there are not any grapes we haven't compared.

It's easy to get loss in real practice if we don't mark which grape has been compared. There are two ways to to solve this problem, which are suitable for different data-structures respectively.

9.2.1 Labeling

Method 1 is to label each grape with a number: $\{1, 2, ..., N\}$, and we systematically perform the comparison in the order of this sequence of labels. That we first compare grape number 1 and grape number 2, pick the bigger one; then we take grape number 3, and do the comparison, ... We repeat this process until arrive at grape number N. This is quite suitable for elements stored in an array.

```
function Min($A$)
    $m \leftarrow A[1]$
    for $i \leftarrow 2$ to $|A|$ do
        if $A[i] < m$ then
            $m \leftarrow A[i]$
    return $m$
```

With Min defined, we can complete the basic version of selection sort (or naive version without any optimization in terms of time and space).

However, this algorithm returns the value of the minimum element instead of its location (or the label of the grape), which needs a bit tweaking for the in-place version. Some languages such as ISO C++, support returning the reference as result, so that the swap can be achieved directly as below.

```
template<typename T>
T& min(T* from, T* to) {
    T* m;
    for (m = from++; from != to; ++from)
        if (*from < *m)
            m = from;
    return *m;
}

template<typename T>
void ssort(T* xs, int n) {
    int i;
    for (i = 0; i < n; ++i)
        std::swap(xs[i], min(xs+i, xs+n));
}
```

In environments without reference semantics, the solution is to return the location of the minimum element instead of the value:

```
function Min-At($A$)
    $m \leftarrow$ First-Index($A$)
    for $i \leftarrow m + 1$ to $|A|$ do
        if $A[i] < A[m]$ then
            $m \leftarrow i$
    return $m$
```

Note that since we pass $A[i...]$ to Min-At as the argument, we assume the first element $A[i]$ as the smallest one, and examine all elements $A[i+1], A[i+2], ...$ one by one. Function First-Index() is used to retrieve i from the input parameter.

The following Python example program, for example, completes the basic in-place selection sort algorithm based on this idea. It explicitly passes the range information to the function of finding the minimum location.

```
def ssort(xs):
    n = len(xs)
    for i in range(n):
        m = min_at(xs, i, n)
        (xs[i], xs[m]) = (xs[m], xs[i])
    return xs

def min_at(xs, i, n):
    m = i;
    for j in range(i+1, n):
        if xs[j] < xs[m]:
            m = j
    return m
```

9.2.2 Grouping

Another method is to group all grapes in two parts: the group we have examined, and the rest we haven't. We denote these two groups as A and B; All the elements (grapes) as L. At the beginning, we haven't examine any grapes at all, thus A is empty (Φ), and B contains all grapes. We can select arbitrary two grapes from B, compare them, and put the loser (the smaller one for example) to A. After that, we repeat this process by continuously picking arbitrary grapes from B, and compare with the winner of the previous time until B becomes empty. At this time being, the final winner is the minimum element. And A turns to be $L - \{min(L)\}$, which can be used for the next time minimum finding.

There is an invariant of this method, that at any time, we have $L = A \cup \{m\} \cup B$, where m is the winner so far we hold.

This approach doesn't need the collection of grapes being indexed (as being labeled in method 1). It's suitable for any traversable data structures, including linked-list etc. Suppose b_1 is an arbitrary element in B if B isn't empty, and B' is the rest of elements with b_1 being removed, this method can be formalized as the below auxiliary function.

$$min'(A, m, B) = \begin{cases} (m, A) & : B = \Phi \\ min'(A \cup \{m\}, b_1, B') & : b_1 < m \\ min'(A \cup \{b_1\}, m, B') & : otherwise \end{cases} \tag{9.2}$$

In order to pick the minimum element, we call this auxiliary function by passing an empty A, and use an arbitrary element (for instance, the first one) to initialize m:

$$extractMin(L) = min'(\Phi, l_1, L') \tag{9.3}$$

Where L' is all elements in L except for the first one l_1. The algorithm $extractMin$) doesn't not only find the minimum element, but also returns the updated collection which doesn't contain this minimum. Summarize this minimum extracting algorithm up to the basic selection sort definition, we can create a complete functional sorting program, for example as this Haskell code snippet.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs
```

```
extractMin (x:xs) = min' [] x xs where
  min' ys m [] = (m, ys)
  min' ys m (x:xs) = if m < x then min' (x:ys) m xs else min' (m:ys) x xs
```

The first line handles the trivial edge case that the sorting result for empty list is obvious empty; The second clause ensures that, there is at least one element, that's why the `extractMin` function needn't other pattern-matching.

One may think the second clause of `min'` function should be written like below:

```
min' ys m (x:xs) = if m < x then min' ys ++ [x] m xs
                            else min' ys ++ [m] x xs
```

Or it will produce the updated list in reverse order. Actually, it's necessary to use 'cons' instead of appending here. This is because appending is linear operation which is proportion to the length of part A, while 'cons' is constant $O(1)$ time operation. In fact, we needn't keep the relative order of the list to be sorted, as it will be re-arranged anyway during sorting.

It's quite possible to keep the relative order during sorting, while ensure the performance of finding the minimum element not degrade to quadratic. The following equation defines a solution.

$$extractMin(L) = \begin{cases} (l_1, \Phi) & : |L| = 1 \\ (l_1, L') & : l_1 < m, (m, L'') = extractMin(L') \\ (m, l_1 \cup L'') & : otherwise \end{cases} \tag{9.4}$$

If L is a singleton, the minimum is the only element it contains. Otherwise, denote l_1 as the first element in L, and L' contains the rest elements except for l_1, that $L' = \{l_2, l_3, ...\}$. The algorithm recursively finding the minimum element in L', which yields the intermediate result as (m, L''), that m is the minimum element in L', and L'' contains all rest elements except for m. Comparing l_1 with m, we can determine which of them is the final minimum result.

The following Haskell program implements this version of selection sort.

```
sort [] = []
sort xs = x : sort xs' where
  (x, xs') = extractMin xs

extractMin [x] = (x, [])
extractMin (x:xs) = if x < m then (x, xs) else (m, x:xs') where
  (m, xs') = extractMin xs
```

Note that only 'cons' operation is used, we needn't appending at all because the algorithm actually examines the list from right to left. However, it's not free, as this program need book-keeping the context (via call stack typically). The relative order is ensured by the nature of recursion. Please refer to the appendix about tail recursion call for detailed discussion.

9.2.3 performance of the basic selection sorting

Both the labeling method, and the grouping method need examine all the elements to pick the minimum in every round; and we totally pick up the minimum element N times. Thus the performance is around $N+(N-1)+(N-2)+...+1$ which is $\frac{N(N+1)}{2}$. Selection sort is a quadratic algorithm bound to $O(N^2)$ time.

Compare to the insertion sort, which we introduced previously, selection sort performs same in its best case, worst case and average case. While insertion sort performs well in best case (that the list has been reverse ordered, and it is stored in linked-list) as $O(N)$, and the worst performance is $O(N^2)$.

In the next sections, we'll examine, why selection sort performs poor, and try to improve it step by step.

Exercise 9.1

- Implement the basic imperative selection sort algorithm (the none in-place version) in your favorite programming language. Compare it with the in-place version, and analyze the time and space effectiveness.

9.3 Minor Improvement

9.3.1 Parameterize the comparator

Before any improvement in terms of performance, let's make the selection sort algorithm general enough to handle different sorting criteria.

We've seen two opposite examples so far, that one may need sort the elements in ascending order or descending order. For the former case, we need repeatedly finding the minimum, while for the later, we need find the maximum instead. They are just two special cases. In real world practice, one may want to sort things in varies criteria, e.g. in terms of size, weight, age, ...

One solution to handle them all is to passing the criteria as a compare function to the basic selection sort algorithms. For example:

$$sort(c, L) = \begin{cases} \Phi & : L = \Phi \\ m \cup sort(c, L'') & : otherwise, (m, L'') = extract(c, L') \end{cases} \tag{9.5}$$

And the algorithm $extract(c, L)$ is defined as below.

$$extract(c, L) = \begin{cases} (l_1, \Phi) & : |L| = 1 \\ (l_1, L') & : c(l_1, m), (m, L'') = extract(c, L') \\ (m, \{l_1\} \cup L'') & : \neg c(l_1, m) \end{cases} \tag{9.6}$$

Where c is a comparator function, it takes two elements, compare them and returns the result of which one is preceding of the other. Passing 'less than' operator ($<$) turns this algorithm to be the version we introduced in previous section.

Some environments require to pass the total ordering comparator, which returns result among 'less than', 'equal', and 'greater than'. We needn't such strong condition here, that c only tests if 'less than' is satisfied. However, as the minimum requirement, the comparator should meet the strict weak ordering as following [16]:

- Irreflexivity, for all x, it's not the case that $x < x$;

- Asymmetric, For all x and y, if $x < y$, then it's not the case $y < x$;
- Transitivity, For all x, y, and z, if $x < y$, and $y < z$, then $x < z$;

The following Scheme/Lisp program translates this generic selection sorting algorithm. The reason why we choose Scheme/Lisp here is because the lexical scope can simplify the needs to pass the 'less than' comparator for every function calls.

```
(define (sel-sort-by ltp? lst)
  (define (ssort lst)
    (if (null? lst)
        lst
        (let ((p (extract-min lst)))
          (cons (car p) (ssort (cdr p))))))
  (define (extract-min lst)
    (if (null? (cdr lst))
        lst
        (let ((p (extract-min (cdr lst))))
          (if (ltp? (car lst) (car p))
              lst
              (cons (car p) (cons (car lst) (cdr p)))))))
  (ssort lst))
```

Note that, both `ssort` and `extract-min` are inner functions, so that the 'less than' comparator `ltp?` is available to them. Passing '<' to this function yields the normal sorting in ascending order:

```
(sel-sort-by < '(3 1 2 4 5 10 9))
;Value 16: (1 2 3 4 5 9 10)
```

It's possible to pass varies of comparator to imperative selection sort as well. This is left as an exercise to the reader.

For the sake of brevity, we only consider sorting elements in ascending order in the rest of this chapter. And we'll not pass comparator as a parameter unless it's necessary.

9.3.2 Trivial fine tune

The basic in-place imperative selection sorting algorithm iterates all elements, and picking the minimum by traversing as well. It can be written in a compact way, that we inline the minimum finding part as an inner loop.

```
procedure SORT($A$)
    for $i \leftarrow 1$ to $|A|$ do
        $m \leftarrow i$
        for $j \leftarrow i + 1$ to $|A|$ do
            if $A[i] < A[m]$ then
                $m \leftarrow i$
        EXCHANGE $A[i] \leftrightarrow A[m]$
```

Observe that, when we are sorting N elements, after the first $N-1$ minimum ones are selected, the left only one, is definitely the N-th big element, so that we need NOT find the minimum if there is only one element in the list. This indicates that the outer loop can iterate to $N-1$ instead of N.

Another place we can fine tune, is that we needn't swap the elements if the i-th minimum one is just $A[i]$. The algorithm can be modified accordingly as below:

```
procedure SORT($A$)
    for $i \leftarrow 1$ to $|A| - 1$ do
        $m \leftarrow i$
        for $j \leftarrow i + 1$ to $|A|$ do
            if $A[i] < A[m]$ then
                $m \leftarrow i$
        if $m \neq i$ then
            EXCHANGE $A[i] \leftrightarrow A[m]$
```

Definitely, these modifications won't affects the performance in terms of big-O.

9.3.3 Cock-tail sort

Knuth gave an alternative realization of selection sort in [1]. Instead of selecting the minimum each time, we can select the maximum element, and put it to the last position. This method can be illustrated by the following algorithm.

```
procedure SORT'($A$)
    for $i \leftarrow |A|$ down-to 2 do
        $m \leftarrow i$
        for $j \leftarrow 1$ to $i - 1$ do
            if $A[m] < A[i]$ then
                $m \leftarrow i$
        EXCHANGE $A[i] \leftrightarrow A[m]$
```

As shown in figure 13.1, at any time, the elements on right most side are sorted. The algorithm scans all unsorted ones, and locate the maximum. Then, put it to the tail of the unsorted range by swapping.

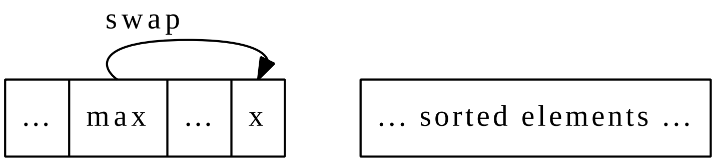


Figure 9.4: Select the maximum every time and put it to the end.

This version reveals the fact that, selecting the maximum element can sort the element in ascending order as well. What's more, we can find both the minimum and the maximum elements in one pass of traversing, putting the minimum at the first location, while putting the maximum at the last position. This approach can speed up the sorting slightly (halve the times of the outer loop).

```
procedure SORT($A$)
    for $i \leftarrow 1$ to $\lfloor \frac{|A|}{2} \rfloor$ do
        $min \leftarrow i$
        $max \leftarrow |A| + 1 - i$
```

```
  if $A[max] < A[min]$ then
      EXCHANGE $A[min] \leftrightarrow A[max]$
  for $j \leftarrow i + 1$ to $|A| - i$ do
      if $A[j] < A[min]$ then
          $min \leftarrow j$
      if $A[max] < A[j]$ then
          $max \leftarrow j$
  EXCHANGE $A[i] \leftrightarrow A[min]$
  EXCHANGE $A[|A| + 1 - i] \leftrightarrow A[max]$
```

This algorithm can be illustrated as in figure 9.5, at any time, the left most and right most parts contain sorted elements so far. That the smaller sorted ones are on the left, while the bigger sorted ones are on the right. The algorithm scans the unsorted ranges, located both the minimum and the maximum positions, then put them to the head and the tail position of the unsorted ranges by swapping.

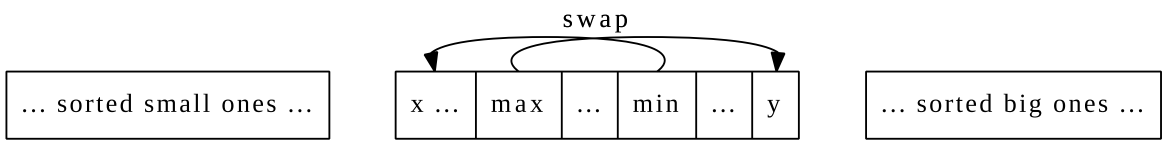


Figure 9.5: Select both the minimum and maximum in one pass, and put them to the proper positions.

Note that it's necessary to swap the left most and right most elements before the inner loop if they are not in correct order. This is because we scan the range excluding these two elements. Another method is to initialize the first element of the unsorted range as both the maximum and minimum before the inner loop. However, since we need two swapping operations after the scan, it's possible that the first swapping moves the maximum or the minimum from the position we just found, which leads the second swapping malfunctioned. How to solve this problem is left as exercise to the reader.

The following Python example program implements this cock-tail sort algorithm.

```
def cocktail_sort(xs):
    n = len(xs)
    for i in range(n / 2):
        (mi, ma) = (i, n - 1 -i)
        if xs[ma] < xs[mi]:
            (xs[mi], xs[ma]) = (xs[ma], xs[mi])
        for j in range(i+1, n - 1 - i):
            if xs[j] < xs[mi]:
                mi = j
            if xs[ma] < xs[j]:
                ma = j
        (xs[i], xs[mi]) = (xs[mi], xs[i])
        (xs[n - 1 - i], xs[ma]) = (xs[ma], xs[n - 1 - i])
    return xs
```

It's possible to realize cock-tail sort in functional approach as well. An intuitive recursive description can be given like this:

- Trivial edge case: If the list is empty, or there is only one element in the list, the sorted result is obviously the origin list;
- Otherwise, we select the minimum and the maximum, put them in the head and tail positions, then recursively sort the rest elements.

This algorithm description can be formalized by the following equation.

$$sort(L) = \begin{cases} L & : |L| \leq 1 \\ \{l_{min}\} \cup sort(L'') \cup \{l_{max}\} & : otherwise \end{cases} \tag{9.7}$$

Where the minimum and the maximum are extracted from L by a function $select(L)$.

$$(l_{min}, L'', l_{max}) = select(L)$$

Note that, the minimum is actually linked to the front of the recursive sort result. Its semantic is a constant $O(1)$ time 'cons' (refer to the appendix of this book for detail). While the maximum is appending to the tail. This is typically a linear $O(N)$ time expensive operation. We'll optimize it later.

Function $select(L)$ scans the whole list to find both the minimum and the maximum. It can be defined as below:

$$select(L) = \begin{cases} (min(l_1, l_2), max(l_1, l_2)) & : L = \{l_1, l_2\} \\ (l_1, \{l_{min}\} \cup L'', l_{max}) & : l_1 < l_{min} \\ (l_{min}, \{l_{max}\} \cup L'', l_1) & : l_{max} < l_1 \\ (l_{min}, \{l_1\} \cup L'', l_{max}) & : otherwise \end{cases} \tag{9.8}$$

Where $(l_{min}, L'', l_{max}) = select(L')$ and L' is the rest of the list except for the first element l_1. If there are only two elements in the list, we pick the smaller as the minimum, and the bigger as the maximum. After extract them, the list becomes empty. This is the trivial edge case; Otherwise, we take the first element l_1 out, then recursively perform selection on the rest of the list. After that, we compare if l_1 is less then the minimum or greater than the maximum candidates, so that we can finalize the result.

Note that for all the cases, there is no appending operation to form the result. However, since selection must scan all the element to determine the minimum and the maximum, it is bound to $O(N)$ linear time.

The complete example Haskell program is given as the following.

```haskell
csort [] = []
csort [x] = [x]
csort xs = mi : csort xs' ++ [ma] where
  (mi, xs', ma) = extractMinMax xs

extractMinMax [x, y] = (min x y, [], max x y)
extractMinMax (x:xs) | x < mi = (x, mi:xs', ma)
                     | ma < x = (mi, ma:xs', x)
                     | otherwise = (mi, x:xs', ma)
  where (mi, xs', ma) = extractMinMax xs
```

We mentioned that the appending operation is expensive in this intuitive version. It can be improved. This can be achieved in two steps. The first step is to convert the cock-tail sort into tail-recursive call. Denote the sorted small ones as A, and sorted big ones as B in figure 9.5. We use A and B as accumulators. The new cock-tail sort is defined as the following.

$$sort'(A, L, B) = \begin{cases} A \cup L \cup B & : L = \Phi \lor |L| = 1 \\ sort'(A \cup \{l_{min}\}, L'', \{l_{max}\} \cup B) & : otherwise \end{cases} \tag{9.9}$$

Where l_{min}, l_{max} and L'' are defined as same as before. And we start sorting by passing empty A and B: $sort(L) = sort'(\Phi, L, \Phi)$.

Besides the edge case, observing that the appending operation only happens on $A \cup \{l_{min}\}$; while l_{max} is only linked to the head of B. This appending occurs in every recursive call. To eliminate it, we can store A in reverse order as $\overleftarrow{A}$, so that l_{max} can be 'cons' to the head instead of appending. Denote $cons(x, L) = \{x\} \cup L$ and $append(L, x) = L \cup \{x\}$, we have the below equation.

$$\begin{aligned} append(L, x) &= reverse(cons(x, reverse(L))) \\ &= reverse(cons(x, \overleftarrow{L})) \end{aligned} \tag{9.10}$$

Finally, we perform a reverse to turn $\overleftarrow{A}$ back to A. Based on this idea, the algorithm can be improved one more step as the following.

$$sort'(A, L, B) = \begin{cases} reverse(A) \cup B & : L = \Phi \\ reverse(\{l_1\} \cup A) \cup B & : |L| = 1 \\ sort'(\{l_{min}\} \cup A, L'', \{l_{max}\} \cup B) & : \end{cases} \tag{9.11}$$

This algorithm can be implemented by Haskell as below.

```
csort' xs = cocktail [] xs [] where
  cocktail as [] bs = reverse as ++ bs
  cocktail as [x] bs = reverse (x:as) ++ bs
  cocktail as xs bs = let (mi, xs', ma) = extractMinMax xs
                      in cocktail (mi:as) xs' (ma:bs)
```

Exercise 9.2

- Realize the imperative basic selection sort algorithm, which can take a comparator as a parameter. Please try both dynamic typed language and static typed language. How to annotate the type of the comparator as general as possible in a static typed language?
- Implement Knuth's version of selection sort in your favorite programming language.
- An alternative to realize cock-tail sort is to assume the i-th element both the minimum and the maximum, after the inner loop, the minimum and maximum are found, then we can swap the the minimum to the i-th position, and the maximum to position $|A|+1-i$. Implement this solution in your favorite imperative language. Please note that there are several special edge cases should be handled correctly:

- $A = \{max, min, ...\}$;
- $A = \{..., max, min\}$;
- $A = \{max, ..., min\}$.

Please don't refer to the example source code along with this chapter before you try to solve this problem.

9.4 Major improvement

Although cock-tail sort halves the numbers of loop, the performance is still bound to quadratic time. It means that, the method we developed so far handles big data poorly compare to other divide and conquer sorting solutions.

To improve selection based sort essentially, we must analyze where is the bottle-neck. In order to sort the elements by comparison, we must examine all the elements for ordering. Thus the outer loop of selection sort is necessary. However, must it scan all the elements every time to select the minimum? Note that when we pick the smallest one at the first time, we actually traverse the whole collection, so that we know which ones are relative big, and which ones are relative small partially.

The problem is that, when we select the further minimum elements, instead of re-using the ordering information we obtained previously, we drop them all, and blindly start a new traverse.

So the key point to improve selection based sort is to re-use the previous result. There are several approaches, we'll adopt an intuitive idea inspired by football match in this chapter.

9.4.1 Tournament knock out

The football world cup is held every four years. There are 32 teams from different continent play the final games. Before 1982, there were 16 teams compete for the tournament finals[4].

For simplification purpose, let's go back to 1978 and imagine a way to determine the champion: In the first round, the teams are grouped into 8 pairs to play the game; After that, there will be 8 winner, and 8 teams will be out. Then in the second round, these 8 teams are grouped into 4 pairs. This time there will be 4 winners after the second round of games; Then the top 4 teams are divided into 2 pairs, so that there will be only two teams left for the final game.

The champion is determined after the total 4 rounds of games. And there are actually $8 + 4 + 2 + 1 = 16$ games. Now we have the world cup champion, however, the world cup game won't finish at this stage, we need to determine which is the silver medal team.

Readers may argue that isn't the team beaten by the champion at the final game the second best? This is true according to the real world cup rule. However, it isn't fair enough in some sense.

We often heard about the so called 'group of death', Let's suppose that Brazil team is grouped with Deutch team at the very beginning. Although both teams are quite strong, one of them must be knocked out. It's quite possible that even the team loss that game can beat all the other teams except for the champion. Figure 9.6 illustrates such case.

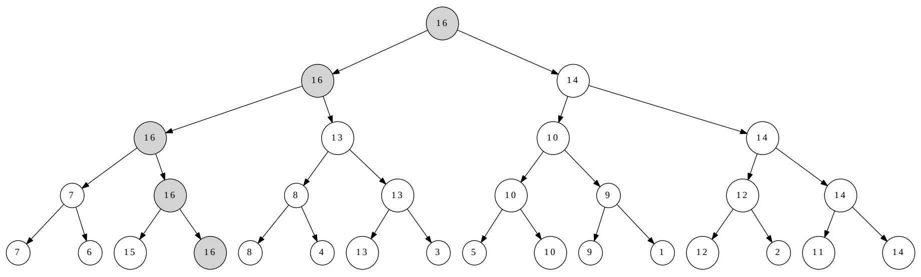


Figure 9.6: The element 15 is knocked out in the first round.

Imagine that every team has a number. The bigger the number, the stronger the team. Suppose that the stronger team always beats the team with smaller number, although this is not true in real world. But this simplification is fair enough for us to develop the tournament knock out solution. This maximum number which represents the champion is 16. Definitely, team with number 14 isn't the second best according to our rules. It should be 15, which is knocked out at the first round of comparison.

The key question here is to find an effective way to locate the second maximum number in this tournament tree. After that, what we need is to apply the same method to select the third, the fourth, ..., to accomplish the selection based sort.

One idea is to assign the champion a very small number (for instance, $-\infty$), so that it won't be selected next time, and the second best one, becomes the new champion. However, suppose there are 2^m teams for some natural number m, it still takes $2^{m-1} + 2^{m-2} + ... + 2 + 1 = 2^m$ times of comparison to determine the new champion. Which is as slow as the first time.

Actually, we needn't perform a bottom-up comparison at all since the tournament tree stores plenty of ordering information. Observe that, the second best team must be beaten by the champion at sometime, or it will be the final winner. So we can track the path from the root of the tournament tree to the leaf of the champion, examine all the teams along with this path to find the second best team.

In figure 9.6, this path is marked in gray color, the elements to be examined are $\{14, 13, 7, 15\}$. Based on this idea, we refine the algorithm like below.

1. Build a tournament tree from the elements to be sorted, so that the champion (the maximum) becomes the root;
2. Extract the root from the tree, perform a top-down pass and replace the maximum with $-\infty$;
3. Perform a bottom-up back-track along the path, determine the new champion and make it as the new root;
4. Repeat step 2 until all elements have been extracted.

Figure 9.7, 9.8, and 9.9 show the steps of applying this strategy.

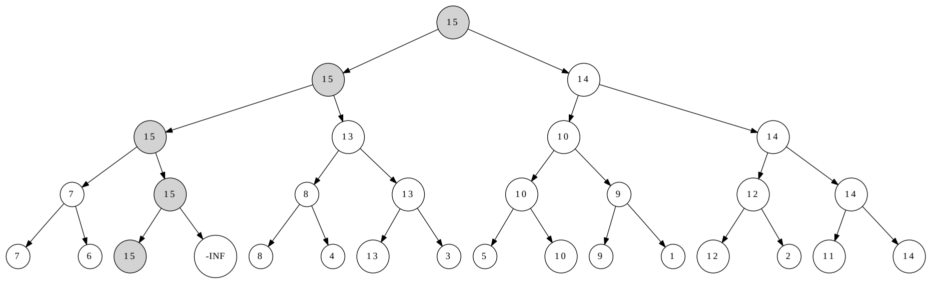


Figure 9.7: Extract 16, replace it with $-\infty$, 15 sifts up to root.

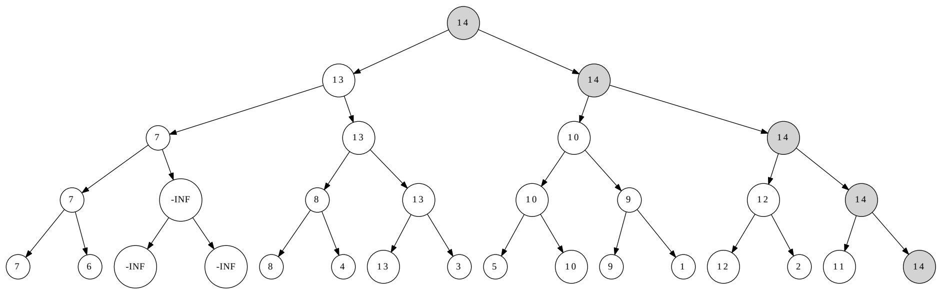


Figure 9.8: Extract 15, replace it with $-\infty$, 14 sifts up to root.

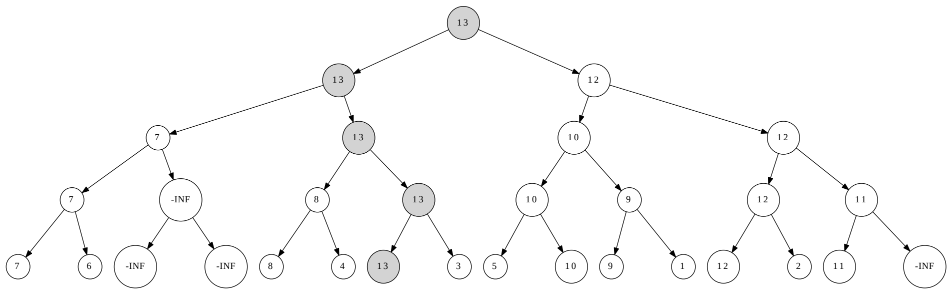


Figure 9.9: Extract 14, replace it with $-\infty$, 13 sifts up to root.

We can reuse the binary tree definition given in the first chapter of this book to represent tournament tree. In order to back-track from leaf to the root, every node should hold a reference to its parent (concept of pointer in some environment such as ANSI C):

```
struct Node {
  Key key;
  struct Node *left, *right, *parent;
};
```

To build a tournament tree from a list of elements (suppose the number of elements are 2^m for some m), we can first wrap each element as a leaf, so that we obtain a list of binary trees. We take every two trees from this list, compare their keys, and form a new binary tree with the bigger key as the root; the two trees are set as the left and right children of this new binary tree. Repeat this operation to build a new list of trees. The height of each tree is increased by 1. Note that the size of the tree list halves after such a pass, so that we can keep reducing the list until there is only one tree left. And this tree is the finally built tournament tree.

```
function BUILD-TREE($A$)
    $T \leftarrow \Phi$
    for each $x \in A$ do
        $t \leftarrow$ CREATE-NODE
        KEY($t$) $\leftarrow x$
        APPEND($T, t$)
    while $|T| > 1$ do
        $T' \leftarrow \Phi$
        for every $t_1, t_2 \in$ T do
            $t \leftarrow$ CREATE-NODE
            KEY($t$) $\leftarrow$ MAX(KEY($t_1$), KEY($t_2$))
            LEFT($t$) $\leftarrow t_1$
            RIGHT($t$) $\leftarrow t_2$
            PARENT($t_1$) $\leftarrow t$
            PARENT($t_2$) $\leftarrow t$
            APPEND($T', t$)
        $T \leftarrow T'$
    return $T[1]$
```

Suppose the length of the list A is N, this algorithm firstly traverses the list to build tree, which is linear to N time. Then it repeatedly compares pairs, which loops proportion to $N + \frac{N}{2} + \frac{N}{4} + ... + 2 = 2N$. So the total performance is bound to $O(N)$ time.

The following ANSI C program implements this tournament tree building algorithm.

```
struct Node* build(const Key* xs, int n) {
    int i;
    struct Node *t, **ts = (struct Node**) malloc(sizeof(struct Node*) * n);
    for (i = 0; i < n; ++i)
        ts[i] = leaf(xs[i]);
    for (; n > 1; n /= 2)
        for (i = 0; i < n; i += 2)
```

```
            ts[i/2] = branch(max(ts[i]→key, ts[i+1]→key), ts[i], ts[i+1]);
    t = ts[0];
    free(ts);
    return t;
}
```

The type of key can be defined somewhere, for example:

```
typedef int Key;
```

Function `leaf(x)` creats a leaf node, with value `x` as key, and sets all its fields, left, right and parent to `NIL`. While function `branch(key, left, right)` creates a branch node, and links the new created node as parent of its two children if they are not empty. For the sake of brevity, we skip the detail of them. They are left as exercise to the reader, and the complete program can be downloaded along with this book.

Some programming environments, such as Python provides tool to iterate every two elements at a time, for example:

```
for x, y in zip(*[iter(ts)]*2):
```

We skip such language specific feature, readers can refer to the Python example program along with this book for details.

When the maximum element is extracted from the tournament tree, we replace it with $-\infty$, and repeatedly replace all these values from the root to the leaf. Next, we back-track to root through the parent field, and determine the new maximum element.

```
function EXTRACT-MAX($T$)
    $m \leftarrow$ KEY($T$)
    KEY($T$) $\leftarrow -\infty$
    while $\neg$ LEAF?($T$) do                          ▷ The top down pass
        if KEY(LEFT($T$)) $= m$ then
            $T \leftarrow$ LEFT($T$)
        else
            $T \leftarrow$ RIGHT($T$)
        KEY($T$) $\leftarrow -\infty$
    while PARENT($T$) $\neq \Phi$ do                     ▷ The bottom up pass
        $T \leftarrow$ PARENT($T$)
        KEY($T$) $\leftarrow$ MAX(KEY(LEFT($T$)), KEY(RIGHT($T$)))
    return $m$
```

This algorithm returns the extracted maximum element, and modifies the tournament tree in-place. Because we can't represent $-\infty$ in real program by limited length of word, one approach is to define a relative negative big number, which is less than all the elements in the tournament tree, for example, suppose all the elements are greater than -65535, we can define negative infinity as below:

```
#define N_INF -65535
```

We can implements this algorithm as the following ANSI C example program.

```
Key pop(struct Node* t) {
    Key x = t→key;
    t→key = N_INF;
    while (!isleaf(t)) {
        t = t→left→key == x ? t→left : t→right;
        t→key = N_INF;
    }
    while (t→parent) {
        t = t→parent;
        t→key = max(t→left→key, t→right→key);
    }
    return x;
}
```

The behavior of Extract-Max is quite similar to the pop operation for some data structures, such as queue, and heap, thus we name it as pop in this code snippet.

Algorithm Extract-Max process the tree in tow passes, one is top-down, then a bottom-up along the path that the 'champion team wins the world cup'. Because the tournament tree is well balanced, the length of this path, which is the height of the tree, is bound to $O(\lg N)$, where N is the number of the elements to be sorted (which are equal to the number of leaves). Thus the performance of this algorithm is $O(\lg N)$.

It's possible to realize the tournament knock out sort now. We build a tournament tree from the elements to be sorted, then continuously extract the maximum. If we want to sort in monotonically increase order, we put the first extracted one to the right most, then insert the further extracted elements one by one to left; Otherwise if we want to sort in decrease order, we can just append the extracted elements to the result. Below is the algorithm sorts elements in ascending order.

```
procedure Sort($A$)
    $T \leftarrow$ Build-Tree($A$)
    for $i \leftarrow |A|$ down to 1 do
        $A[i] \leftarrow$ Extract-Max($T$)
```

Translating it to ANSI C example program is straightforward.

```
void tsort(Key* xs, int n) {
    struct Node* t = build(xs, n);
    while(n)
        xs[--n] = pop(t);
    release(t);
}
```

This algorithm firstly takes $O(N)$ time to build the tournament tree, then performs N pops to select the maximum elements so far left in the tree. Since each pop operation is bound to $O(\lg N)$, thus the total performance of tournament knock out sorting is $O(N \lg N)$.

Refine the tournament knock out

It's possible to design the tournament knock out algorithm in purely functional approach. And we'll see that the two passes (first top-down replace the champion with $-\infty$, then bottom-up determine the new champion) in pop operation can be combined in recursive manner, so that we needn't the parent field any more. We can re-use the functional binary tree definition as the following example Haskell code.

```
data Tr a = Empty | Br (Tr a) a (Tr a)
```

Thus a binary tree is either empty or a branch node contains a key, a left sub tree and a right sub tree. Both children are again binary trees.

We've use hard coded big negative number to represents $-\infty$. However, this solution is ad-hoc, and it forces all elements to be sorted are greater than this pre-defined magic number. Some programming environments support algebraic type, so that we can define negative infinity explicitly. For instance, the below Haskell program setups the concept of infinity [1].

```
data Infinite a = NegInf | Only a | Inf deriving (Eq, Ord)
```

From now on, we switch back to use the $min()$ function to determine the winner, so that the tournament selects the minimum instead of the maximum as the champion.

Denote function $key(T)$ returns the key of the tree rooted at T. Function $wrap(x)$ wraps the element x into a leaf node. Function $tree(l, k, r)$ creates a branch node, with k as the key, l and r as the two children respectively.

The knock out process, can be represented as comparing two trees, picking the smaller key as the new key, and setting these two trees as children:

$$branch(T_1, T_2) = tree(T_1, min(key(T_1), key(T_2)), T_2) \tag{9.12}$$

This can be implemented in Haskell word by word:

```
branch t1 t2 = Br t1 (min (key t1) (key t2)) t2
```

There is limitation in our tournament sorting algorithm so far. It only accepts collection of elements with size of 2^m, or we can't build a complete binary tree. This can be actually solved in the tree building process. Remind that we pick two trees every time, compare and pick the winner. This is perfect if there are always even number of trees. Considering a case in football match, that one team is absent for some reason (sever flight delay or whatever), so that there left one team without a challenger. One option is to make this team the winner, so that it will attend the further games. Actually, we can use the similar approach.

To build the tournament tree from a list of elements, we wrap every element into a leaf, then start the building process.

$$build(L) = build'(\{wrap(x) | x \in L\}) \tag{9.13}$$

The $build'(\mathbb{T})$ function terminates when there is only one tree left in $\mathbb{T}$, which is the champion. This is the trivial edge case. Otherwise, it groups every two trees in a pair to determine the winners. When there are odd numbers of trees, it just makes the last tree as the winner to attend the next level of tournament and recursively repeats the building process.

$$build'(\mathbb{T}) = \begin{cases} \mathbb{T} & : |\mathbb{T}| \leq 1 \\ build'(pair(\mathbb{T})) & : otherwise \end{cases} \tag{9.14}$$

[1]The order of the definition of 'NegInf', regular number, and 'Inf' is significant if we want to derive the default, correct comparing behavior of 'Ord'. Anyway, it's possible to specify the detailed order by make it as an instance of 'Ord'. However, this is Language specific feature which is out of the scope of this book. Please refer to other textbook about Haskell.

Note that this algorithm actually handles another special cases, that the list to be sort is empty. The result is obviously empty.

Denote $\mathbb{T} = \{T_1, T_2, ...\}$ if there are at least two trees, and $\mathbb{T}'$ represents the left trees by removing the first two. Function $pair(\mathbb{T})$ is defined as the following.

$$pair(\mathbb{T}) = \left\{ \begin{array}{rcl} \{branch(T_1, T_2)\} \cup pair(\mathbb{T}') & : & |\mathbb{T}| \geq 2 \\ \mathbb{T} & : & otherwise \end{array} \right. \tag{9.15}$$

The complete tournament tree building algorithm can be implemented as the below example Haskell program.

```
fromList :: (Ord a) ⇒ [a] → Tr (Infinite a)
fromList = build ∘ (map wrap) where
  build [] = Empty
  build [t] = t
  build ts = build $ pair ts
  pair (t1:t2:ts) = (branch t1 t2):pair ts
  pair ts = ts
```

When extracting the champion (the minimum) from the tournament tree, we need examine either the left child sub-tree or the right one has the same key, and recursively extract on that tree until arrive at the leaf node. Denote the left sub-tree of T as L, right sub-tree as R, and K as its key. We can define this popping algorithm as the following.

$$pop(T) = \left\{ \begin{array}{rcl} tree(\Phi, \infty, \Phi) & : & L = \Phi \wedge R = \Phi \\ tree(L', min(key(L'), key(R)), R) & : & K = key(L), L' = pop(L) \\ tree(L, min(key(L), key(R')), R') & : & K = key(R), R' = pop(R) \end{array} \right. \tag{9.16}$$

It's straightforward to translate this algorithm into example Haskell code.

```
pop (Br Empty _ Empty) = Br Empty Inf Empty
pop (Br l k r) | k == key l = let l' = pop l in Br l' (min (key l') (key r)) r
               | k == key r = let r' = pop r in Br l (min (key l) (key r')) r'
```

Note that this algorithm only removes the current champion without returning it. So it's necessary to define a function to get the champion at the root node.

$$top(T) = key(T) \tag{9.17}$$

With these functions defined, tournament knock out sorting can be formalized by using them.

$$sort(L) = sort'(build(L)) \tag{9.18}$$

Where $sort'(T)$ continuously pops the minimum element to form a result list

$$sort'(T) = \left\{ \begin{array}{rcl} \Phi & : & T = \Phi \vee key(T) = \infty \\ \{top(T)\} \cup sort'(pop(T)) & : & otherwise \end{array} \right. \tag{9.19}$$

The rest of the Haskell code is given below to complete the implementation.

```
top = only ∘ key

tsort :: (Ord a) ⇒ [a] → [a]
tsort = sort' ∘ fromList where
    sort' Empty = []
    sort' (Br _ Inf _) = []
    sort' t = (top t) : (sort' $ pop t)
```

And the auxiliary function `only`, `key`, `wrap` accomplished with explicit infinity support are list as the following.

```
only (Only x) = x
key (Br _ k _ ) = k
wrap x = Br Empty (Only x) Empty
```

Exercise 9.3

- Implement the helper function `leaf()`, `branch`, `max()` `lsleaf()`, and `release()` to complete the imperative tournament tree program.
- Implement the imperative tournament tree in a programming language support GC (garbage collection).
- Why can our tournament tree knock out sort algorithm handle duplicated elements (elements with same value)? We say a sorting algorithm stable, if it keeps the original order of elements with same value. Is the tournament tree knock out sorting stable?
- Design an imperative tournament tree knock out sort algorithm, which satisfies the following:
  - Can handle arbitrary number of elements;
  - Without using hard coded negative infinity, so that it can take elements with any value.
- Compare the tournament tree knock out sort algorithm and binary tree sort algorithm, analyze efficiency both in time and space.
- Compare the heap sort algorithm and binary tree sort algorithm, and do same analysis for them.

9.4.2 Final improvement by using heap sort

We manage improving the performance of selection based sorting to $O(N \lg N)$ by using tournament knock out. This is the limit of comparison based sort according to [1]. However, there are still rooms for improvement. After sorting, there lefts a complete binary tree with all leaves and branches hold useless infinite values. This isn't space efficient at all. Can we release the nodes when popping?

Another observation is that if there are N elements to be sorted, we actually allocate about $2N$ tree nodes. N for leaves and N for branches. Is there any better way to halve the space usage?

The final sorting structure described in equation 9.19 can be easily uniformed to a more general one if we treat the case that the tree is empty if its root holds infinity as key:

$$sort'(T) = \begin{cases} \Phi & : T = \Phi \\ \{top(T)\} \cup sort'(pop(T)) & : otherwise \end{cases} \tag{9.20}$$

This is exactly as same as the one of heap sort we gave in previous chapter. Heap always keeps the minimum (or the maximum) on the top, and provides fast pop operation. The binary heap by implicit array encodes the tree structure in array index, so there aren't any extra spaces allocated except for the N array cells. The functional heaps, such as leftist heap and splay heap allocate N nodes as well. We'll introduce more heaps in next chapter which perform well in many aspects.

9.5 Short summary

In this chapter, we present the evolution process of selection based sort. selection sort is easy and commonly used as example to teach students about embedded looping. It has simple and straightforward structure, but the performance is quadratic. In this chapter, we do see that there exists ways to improve it not only by some fine tuning, but also fundamentally change the data structure, which leads to tournament knock out and heap sort.

Bibliography

[1] 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

[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] Wikipedia. “Strict weak order”. http://en.wikipedia.org/wiki/Strict_weak_order

[4] Wikipedia. “FIFA world cup”. http://en.wikipedia.org/wiki/FIFA_World_Cup

Chapter 10

Binomial heap, Fibonacci heap, and pairing heap

10.1 Introduction

In previous chapter, we mentioned that heaps can be generalized and implemented with varies of data structures. However, only binary heaps are focused so far no matter by explicit binary trees or implicit array.

It's quite natural to extend the binary tree to K-ary [1] tree. In this chapter, we first show Binomial heaps which is actually consist of forest of K-ary trees. Binomial heaps gain the performance for all operations to $O(\lg N)$, as well as keeping the finding minimum element to $O(1)$ time.

If we delay some operations in Binomial heaps by using lazy strategy, it turns to be Fibonacci heap.

All binary heaps we have shown perform no less than $O(\lg N)$ time for merging, we'll show it's possible to improve it to $O(1)$ with Fibonacci heap, which is quite helpful to graph algorithms. Actually, Fibonacci heap achieves almost all operations to good amortized time bound as $O(1)$, and left the heap pop to $O(\lg N)$.

Finally, we'll introduce about the pairing heaps. It has the best performance in practice although the proof of it is still a conjecture for the time being.

10.2 Binomial Heaps

10.2.1 Definition

Binomial heap is more complex than most of the binary heaps. However, it has excellent merge performance which bound to $O(\lg N)$ time. A binomial heap is consist of a list of binomial trees.

Binomial tree

In order to explain why the name of the tree is 'binomial', let's review the famous Pascal's triangle (Also know as the Jia Xian's triangle to memorize the Chinese methematican Jia Xian (1010-1070).) [4].

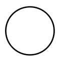
```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
...
```

In each row, the numbers are all binomial coefficients. There are many ways to gain a series of binomial coefficient numbers. One of them is by using recursive composition. Binomial trees, as well, can be defined in this way as the following.

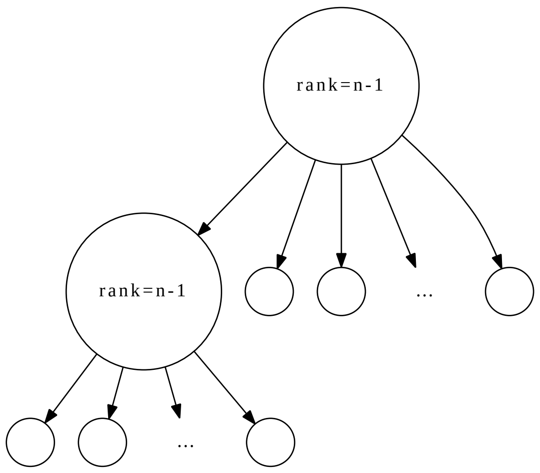
- A binomial tree of rank 0 has only a node as the root;
- A binomial tree of rank N is consist of two rank $N-1$ binomial trees, Among these 2 sub trees, the one has the bigger root element is linked as the leftmost child of the other.

We denote a binomial tree of rank 0 as B_0, and the binomial tree of rank n as B_n.

Figure 10.1 shows a B_0 tree and how to link 2 B_{n-1} trees to a B_n tree.



(a) A B_0 tree.



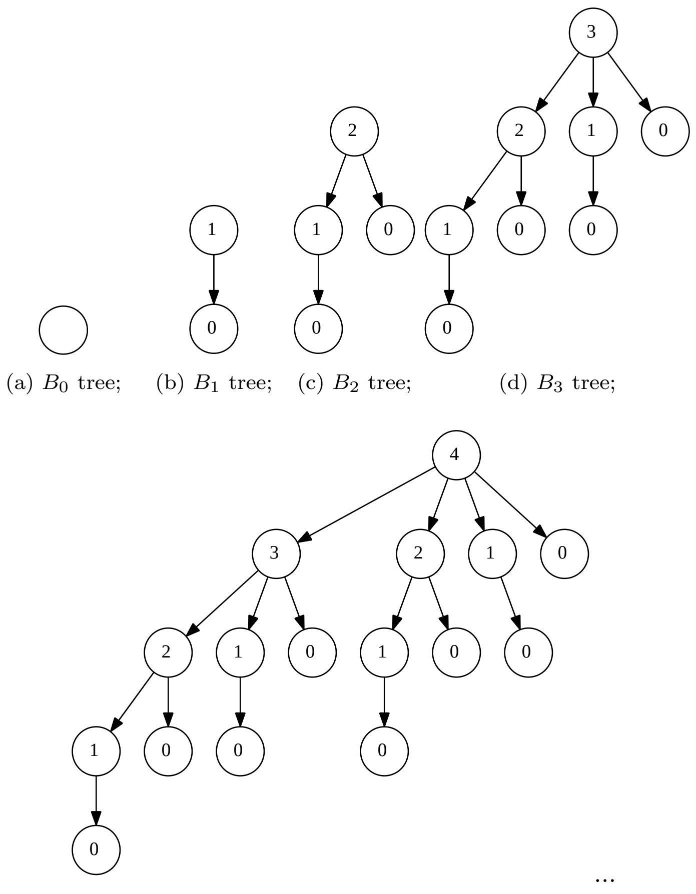
(b) Linking 2 B_{n-1} trees yields a B_n tree.

Figure 10.1: Recursive definition of binomial trees

With this recursive definition, it easy to draw the form of binomial trees of rank 0, 1, 2, ..., as shown in figure 10.2

Observing the binomial trees reveals some interesting properties. For each rank N binomial tree, if counting the number of nodes in each row, it can be found that it is the binomial number.

For instance for rank 4 binomial tree, there is 1 node as the root; and in the second level next to root, there are 4 nodes; and in 3rd level, there are 6 nodes;



(a) B_0 tree; (b) B_1 tree; (c) B_2 tree; (d) B_3 tree;

...

(e) B_4 tree;

Figure 10.2: Forms of binomial trees with rank = 0, 1, 2, 3, 4, ...

and in 4-th level, there are 4 nodes; and the 5-th level, there is 1 node. They are exactly 1, 4, 6, 4, 1 which is the 5th row in Pascal's triangle. That's why we call it binomial tree.

Another interesting property is that the total number of node for a binomial tree with rank N is 2^N. This can be proved either by binomial theory or the recursive definition directly.

Binomial heap

With binomial tree defined, we can introduce the definition of binomial heap. A binomial heap is a set of binomial trees (or a forest of binomial trees) that satisfied the following properties.

- Each binomial tree in the heap conforms to *heap property*, that the key of a node is equal or greater than the key of its parent. Here the heap is actually min-heap, for max-heap, it changes to 'equal or less than'. In this chapter, we only discuss about min-heap, and max-heap can be equally applied by changing the comparison condition.
- There is at most one binomial tree which has the rank r. In other words, there are no two binomial trees have the same rank.

This definition leads to an important result that for a binomial heap contains N elements, and if convert N to binary format yields $a_0, a_1, a_2, ..., a_m$, where a_0 is the LSB and a_m is the MSB, then for each $0 \leq i \leq m$, if $a_i = 0$, there is no binomial tree of rank i and if $a_i = 1$, there must be a binomial tree of rank i.

For example, if a binomial heap contains 5 element, as 5 is '(LSB)101(MSB)', then there are 2 binomial trees in this heap, one tree has rank 0, the other has rank 2.

Figure 10.3 shows a binomial heap which have 19 nodes, as 19 is '(LSB)11001(MSB)' in binary format, so there is a B_0 tree, a B_1 tree and a B_4 tree.

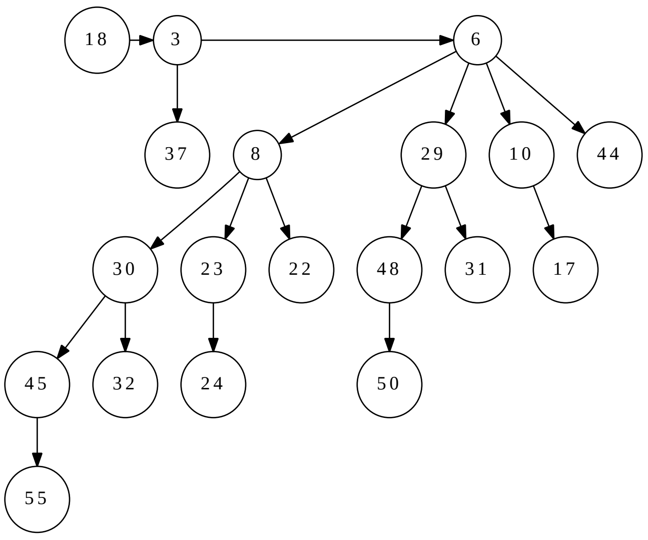


Figure 10.3: A binomial heap with 19 elements

Data layout

There are two ways to define K-ary trees imperatively. One is by using 'left-child, right-sibling' approach[2]. It is compatible with the typical binary tree structure. For each node, it has two fields, left field and right field. We use the left field to point to the first child of this node, and use the right field to point to the sibling node of this node. All siblings are represented as a single directional linked list. Figure 10.4 shows an example tree represented in this way.

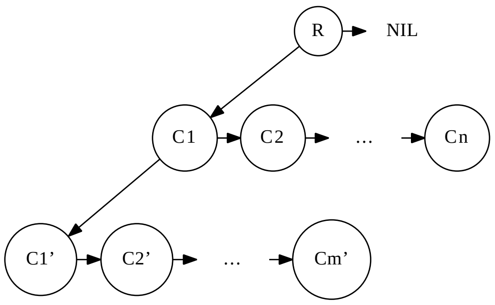


Figure 10.4: Example tree represented in 'left-child, right-sibling' way. R is the root node, it has no sibling, so it right side is pointed to NIL. $C_1, C_2, ..., C_n$ are children of R. C_1 is linked from the left side of R, other siblings of C_1 are linked one next to each other on the right side of C_1. $C'_2, ..., C'_m$ are children of C_1.

The other way is to use the library defined collection container, such as array or list to represent all children of a node.

Since the rank of a tree plays very important role, we also defined it as a field.

For 'left-child, right-sibling' method, we defined the binomial tree as the following.[1]

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.child = None
        self.sibling = None
```

When initialize a tree with a key, we create a leaf node, set its rank as zero and all other fields are set as NIL.

It quite nature to utilize pre-defined list to represent multiple children as below.

```
class BinomialTree:
    def __init__(self, x = None):
        self.rank = 0
        self.key = x
        self.parent = None
        self.children = []
```

[1]C programs are also provided along with this book.

For purely functional settings, such as in Haskell language, binomial tree are defined as the following.

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

While binomial heap are defined as a list of binomial trees (a forest) with ranks in monotonically increase order. And as another implicit constraint, there are no two binomial trees have the same rank.

```
type BiHeap a = [BiTree a]
```

10.2.2 Basic heap operations

Linking trees

Before dive into the basic heap operations such as pop and insert, We'll first realize how to link two binomial trees with same rank into a bigger one. According to the definition of binomial tree and heap property that the root always contains the minimum key, we firstly compare the two root values, select the smaller one as the new root, and insert the other tree as the first child in front of all other children. Suppose function $Key(T)$, $Children(T)$, and $Rank(T)$ access the key, children and rank of a binomial tree respectively.

$$link(T_1, T_2) = \begin{cases} node(r+1, x, \{T_2\} \cup C_1) & : x < y \\ node(r+1, y, \{T_1\} \cup C_2) & : otherwise \end{cases} \tag{10.1}$$

Where

$$\begin{array}{l} x = Key(T_1) \\ y = Key(T_2) \\ r = Rank(T_1) = Rank(T_2) \\ C_1 = Children(T_1) \\ C_2 = Children(T_2) \end{array}$$

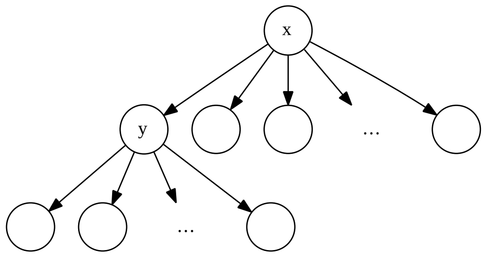


Figure 10.5: Suppose $x < y$, insert y as the first child of x.

Note that the link operation is bound to $O(1)$ time if the $\cup$ is a constant time operation. It's easy to translate the link function to Haskell program as the following.

```
link :: (Ord a) ⇒ BiTree a → BiTree a → BiTree a
link t1@(Node r x c1) t2@(Node _ y c2) =
    if x<y then Node (r+1) x (t2:c1)
    else Node (r+1) y (t1:c2)
```

It's possible to realize the link operation in imperative way. If we use 'left child, right sibling' approach, we just link the tree which has the bigger key to the left side of the other's key, and link the children of it to the right side as sibling. Figure 10.6 shows the result of one case.

```
1: function LINK($T_1, T_2$)
2:     if KEY($T_2$) < KEY($T_1$) then
3:         Exchange $T_1 \leftrightarrow T_2$
4:     SIBLING($T_2$) ← CHILD($T_1$)
5:     CHILD($T_1$) ← $T_2$
6:     PARENT($T_2$) ← $T_1$
7:     RANK($T_1$) ← RANK($T_1$) + 1
8:     return $T_1$
```

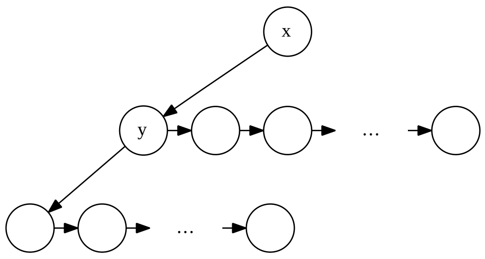


Figure 10.6: Suppose $x < y$, link y to the left side of x and link the original children of x to the right side of y.

And if we use a container to manage all children of a node, the algorithm is like below.

```
1: function LINK'($T_1, T_2$)
2:     if KEY($T_2$) < KEY($T_1$) then
3:         Exchange $T_1 \leftrightarrow T_2$
4:     PARENT($T_2$) ← $T_1$
5:     INSERT-BEFORE(CHILDREN($T_1$), $T_2$)
6:     RANK($T_1$) ← RANK($T_1$) + 1
7:     return $T_1$
```

It's easy to translate both algorithms to real program. Here we only show the Python program of LINK' for illustration purpose [2].

```
def link(t1, t2):
    if t2.key < t1.key:
        (t1, t2) = (t2, t1)
    t2.parent = t1
    t1.children.insert(0, t2)
    t1.rank = t1.rank + 1
    return t1
```

[2]The C and C++ programs are also available along with this book

Exercise 10.1

Implement the tree-linking program in your favorite language with left-child, right-sibling method.

We mentioned linking is a constant time algorithm and it is true when using left-child, right-sibling approach, However, if we use container to manage the children, the performance depends on the concrete implementation of the container. If it is plain array, the linking time will be proportion to the number of children. In this chapter, we assume the time is constant. This is true if the container is implemented in linked-list.

Insert a new element to the heap (push)

As the rank of binomial trees in a forest is monotonically increasing, by using the *link* function defined above, it's possible to define an auxiliary function, so that we can insert a new tree, with rank no bigger than the smallest one, to the heap which is a forest actually.

Denote the non-empty heap as $H = \{T_1, T_2, ..., T_n\}$, we define

$$insertT(H, T) = \begin{cases} \{T\} & : H = \phi \\ \{T\} \cup H & : Rank(T) < Rank(T_1) \\ insertT(H', link(T, T_1)) & : otherwise \end{cases} \tag{10.2}$$

where

$$H' = \{T_2, T_3, ..., T_n\}$$

The idea is that for the empty heap, we set the new tree as the only element to create a singleton forest; otherwise, we compare the ranks of the new tree and the first tree in the forest, if they are same, we link them together, and recursively insert the linked result (a tree with rank increased by one) to the rest of the forest; If they are not same, since the pre-condition constraints the rank of the new tree, it must be the smallest, we put this new tree in front of all the other trees in the forest.

From the binomial properties mentioned above, there are at most $O(\lg N)$ binomial trees in the forest, where N is the total number of nodes. Thus function *insertT* performs at most $O(\lg N)$ times linking, which are all constant time operation. So the performance of *insertT* is $O(\lg N)$. [3]

The relative Haskell program is given as below.

```
insertTree :: (Ord a) ⇒ BiHeap a → BiTree a → BiHeap a
insertTree [] t = [t]
insertTree ts@(t':ts') t = if rank t < rank t' then t:ts
                           else insertTree ts' (link t t')
```

[3]There is interesting observation by comparing this operation with adding two binary numbers. Which will lead to topic of *numeric representation*[6].

With this auxiliary function, it's easy to realize the insertion. We can wrap the new element to be inserted as the only leaf of a tree, then insert this tree to the binomial heap.

$$insert(H, x) = insertT(H, node(0, x, \phi)) \tag{10.3}$$

And we can continuously build a heap from a series of elements by folding. For example the following Haskell define a helper function 'fromList'.

```
fromList = foldl insert []
```

Since wrapping an element as a singleton tree takes $O(1)$ time, the real work is done in *insertT*, the performance of binomial heap insertion is bound to $O(\lg N)$.

The insertion algorithm can also be realized with imperative approach.

Algorithm 4 Insert a tree with 'left-child-right-sibling' method.

```
1: function Insert-Tree($H, T$)
2:     while $H \neq \phi \land$ Rank(Head($H$)) = Rank($T$) do
3:         $(T_1, H) \leftarrow$ Extract-Head($H$)
4:         $T \leftarrow$ Link($T, T_1$)
5:     Sibling($T$) $\leftarrow H$
6:     return $T$
```

Algorithm 4 continuously linking the first tree in a heap with the new tree to be inserted if they have the same rank. After that, it puts the linked-list of the rest trees as the sibling, and returns the new linked-list.

If using a container to manage the children of a node, the algorithm can be given in Algorithm 5.

Algorithm 5 Insert a tree with children managed by a container.

```
1: function Insert-Tree'($H, T$)
2:     while $H \neq \phi \land$ Rank($H[0]$) = Rank($T$) do
3:         $T_1 \leftarrow$ Pop($H$)
4:         $T \leftarrow$ Link($T, T_1$)
5:     Head-Insert($H, T$)
6:     return $H$
```

In this algorithm, function Pop removes the first tree $T_1 = H[0]$ from the forest. And function Head-Insert, insert a new tree before any other trees in the heap, so that it becomes the first element in the forest.

With either Insert-Tree or Insert-Tree' defined. Realize the binomial heap insertion is trivial.

Algorithm 6 Imperative insert algorithm

```
1: function Insert($H, x$)
2:     return Insert-Tree($H$, Node($0, x, \phi$))
```

The following python program implement the insert algorithm by using a container to manage sub-trees. the 'left-child, right-sibling' program is left as an exercise.

```
def insert_tree(ts, t):
    while ts !=[] and t.rank == ts[0].rank:
        t = link(t, ts.pop(0))
    ts.insert(0, t)
    return ts

def insert(h, x):
    return insert_tree(h, BinomialTree(x))
```

Exercise 10.2

Write the insertion program in your favorite imperative programming language by using the 'left-child, right-sibling' approach.

Merge two heaps

When merge two binomial heaps, we actually try to merge two forests of binomial trees. According to the definition, there can't be two trees with the same rank and the ranks are in monotonically increasing order. Our strategy is very similar to merge sort. That in every iteration, we take the first tree from each forest, compare their ranks, and pick the smaller one to the result heap; if the ranks are equal, we then perform linking to get a new tree, and recursively insert this new tree to the result of merging the rest trees.

Figure 10.7 illustrates the idea of this algorithm. This method is different from the one given in [2].

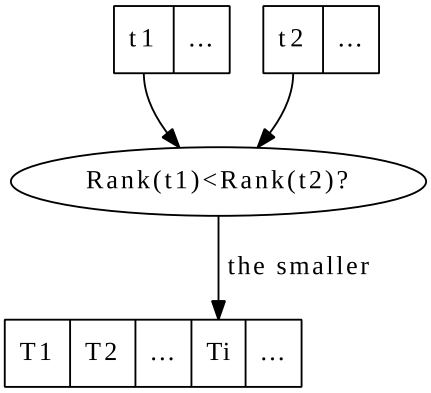
We can formalize this idea with a function. For non-empty cases, we denote the two heaps as $H_1 = \{T_1, T_2, ...\}$ and $H_2 = \{T'_1, T'_2, ...\}$. Let $H'_1 = \{T_2, T_3, ...\}$ and $H'_2 = \{T'_2, T'_3, ...\}$.

$$merge(H_1, H_2) = \begin{cases} H_1 & : H_2 = \phi \\ H_2 & : H_1 = \phi \\ \{T_1\} \cup merge(H'_1, H_2) & : Rank(T_1) < Rank(T'_1) \\ \{T'_1\} \cup merge(H_1, H'_2) & : Rank(T_1) > Rank(T'_1) \\ insertT(merge(H'_1, H'_2), link(T_1, T'_1)) & : otherwise \end{cases} \tag{10.4}$$

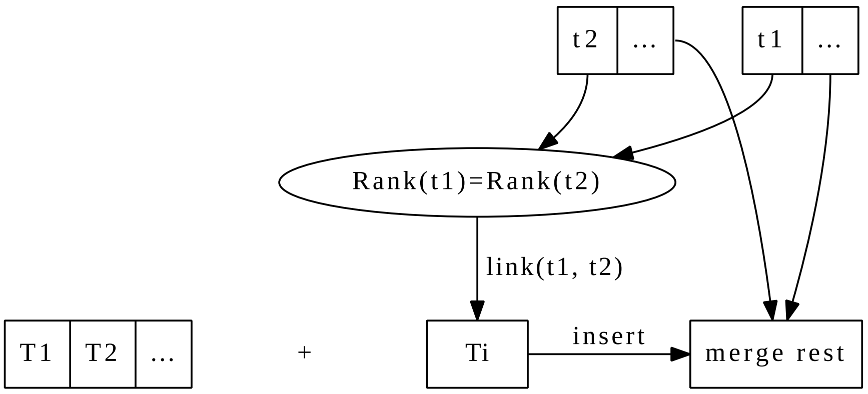
To analysis the performance of merge, suppose there are m_1 trees in H_1, and m_2 trees in H_2. There are at most $m_1 + m_2$ trees in the merged result. If there are no two trees have the same rank, the merge operation is bound to $O(m_1 + m_2)$. While if there need linking for the trees with same rank, *insertT* performs at most $O(m_1 + m_2)$ time. Consider the fact that $m_1 = 1 + \lfloor \lg N_1 \rfloor$ and $m_2 = 1 + \lfloor \lg N_2 \rfloor$, where N_1, N_2 are the numbers of nodes in each heap, and $\lfloor \lg N_1 \rfloor + \lfloor \lg N_2 \rfloor \leq 2\lfloor \lg N \rfloor$, where $N = N_1 + N_2$, is the total number of nodes. the final performance of merging is $O(\lg N)$.

Translating this algorithm to Haskell yields the following program.

```
merge:: (Ord a) ⇒ BiHeap a → BiHeap a → BiHeap a
merge ts1 [] = ts1
merge [] ts2 = ts2
merge ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1:(merge ts1' ts2)
    | rank t1 > rank t2 = t2:(merge ts1 ts2')
    | otherwise = insertTree (merge ts1' ts2') (link t1 t2)
```



(a) Pick the tree with smaller rank to the result.



(b) If two trees have same rank, link them to a new tree, and recursively insert it to the merge result of the rest.

Figure 10.7: Merge two heaps.

Merge algorithm can also be described in imperative way as shown in algorithm 7.

Algorithm 7 imperative merge two binomial heaps

```
1: function Merge($H_1, H_2$)
2:     if $H_1 = \phi$ then
3:         return $H_2$
4:     if $H_2 = \phi$ then
5:         return $H_1$
6:     $H \leftarrow \phi$
7:     while $H_1 \neq \phi \land H_2 \neq \phi$ do
8:         $T \leftarrow \phi$
9:         if Rank($H_1$) < Rank($H_2$) then
10:            $(T, H_1) \leftarrow$ Extract-Head($H_1$)
11:        else if Rank($H_2$) < Rank($H_1$) then
12:            $(T, H_2) \leftarrow$ Extract-Head($H_2$)
13:        else                                              ▷ Equal rank
14:            $(T_1, H_1) \leftarrow$ Extract-Head($H_1$)
15:            $(T_2, H_2) \leftarrow$ Extract-Head($H_2$)
16:            $T \leftarrow$ Link($T_1, T_2$)
17:        Append-Tree($H, T$)
18:    if $H_1 \neq \phi$ then
19:        Append-Trees($H, H_1$)
20:    if $H_2 \neq \phi$ then
21:        Append-Trees($H, H_2$)
22:    return $H$
```

Since both heaps contain binomial trees with rank in monotonically increasing order. Each iteration, we pick the tree with smallest rank and append it to the result heap. If both trees have same rank we perform linking first. Consider the Append-Tree algorithm, The rank of the new tree to be appended, can't be less than any other trees in the result heap according to our merge strategy, however, it might be equal to the rank of the last tree in the result heap. This can happen if the last tree appended are the result of linking, which will increase the rank by one. In this case, we must link the new tree to be inserted with the last tree. In below algorithm, suppose function Last(H) refers to the last tree in a heap, and Append(H, T) just appends a new tree at the end of a forest.

```
1: function Append-Tree($H, T$)
2:     if $H \neq \phi \land$ Rank($T$) = Rank(Last($H$)) then
3:         Last($H$) $\leftarrow$ Link($T$, Last($H$))
4:     else
5:         Append($H, T$)
```

Function Append-Trees repeatedly call this function, so that it can append all trees in a heap to the other heap.

```
1: function Append-Trees($H_1, H_2$)
2:     for each $T \in H_2$ do
3:         $H_1 \leftarrow$ Append-Tree($H_1, T$)
```

The following Python program translates the merge algorithm.

```
def append_tree(ts, t):
    if ts != [] and ts[-1].rank == t.rank:
        ts[-1] = link(ts[-1], t)
    else:
        ts.append(t)
    return ts

def append_trees(ts1, ts2):
    return reduce(append_tree, ts2, ts1)

def merge(ts1, ts2):
    if ts1 == []:
        return ts2
    if ts2 == []:
        return ts1
    ts = []
    while ts1 != [] and ts2 != []:
        t = None
        if ts1[0].rank < ts2[0].rank:
            t = ts1.pop(0)
        elif ts2[0].rank < ts1[0].rank:
            t = ts2.pop(0)
        else:
            t = link(ts1.pop(0), ts2.pop(0))
        ts = append_tree(ts, t)
    ts = append_trees(ts, ts1)
    ts = append_trees(ts, ts2)
    return ts
```

Exercise 10.3

The program given above uses a container to manage sub-trees. Implement the merge algorithm in your favorite imperative programming language with 'left-child, right-sibling' approach.

Pop

Among the forest which forms the binomial heap, each binomial tree conforms to heap property that the root contains the minimum element in that tree. However, the order relationship of these roots can be arbitrary. To find the minimum element in the heap, we can select the smallest root of these trees. Since there are $\lg N$ binomial trees, this approach takes $O(\lg N)$ time.

However, after we locate the minimum element (which is also know as the top element of a heap), we need remove it from the heap and keep the binomial property to accomplish heap-pop operation. Suppose the forest forms the binomial heap consists trees of $B_i, B_j, ..., B_p, ..., B_m$, where B_k is a binomial tree of rank k, and the minimum element is the root of B_p. If we delete it, there will be p children left, which are all binomial trees with ranks $p-1, p-2, ..., 0$.

One tool at hand is that we have defined $O(\lg N)$ merge function. A possible approach is to reverse the p children, so that their ranks change to monotonically increasing order, and forms a binomial heap H_p. The rest of trees is still a binomial heap, we represent it as $H' = H - B_p$. Merging H_p and H' given the final result of pop. Figure 10.8 illustrates this idea.

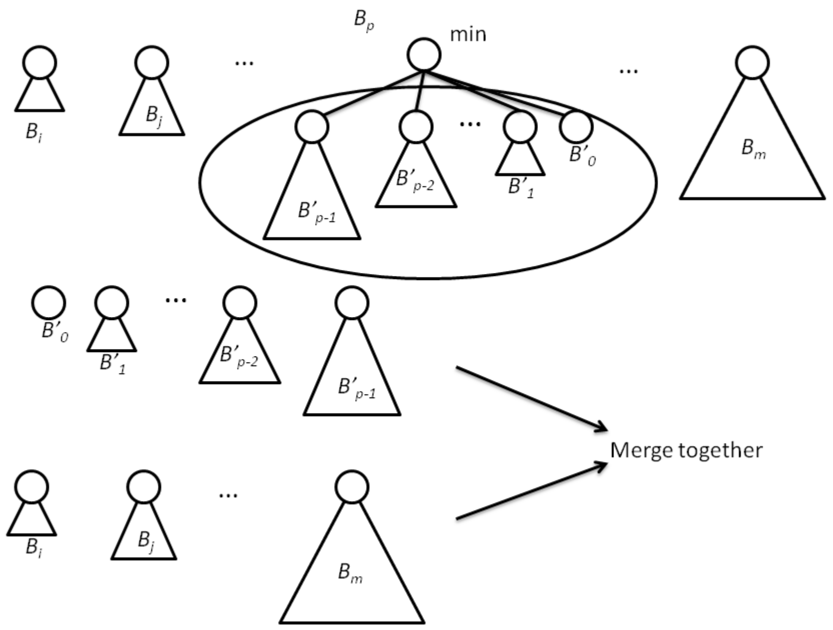


Figure 10.8: Pop the minimum element from a binomial heap.

In order to realize this algorithm, we first need to define an auxiliary function, which can extract the tree contains the minimum element at root from the forest.

$$extractMin(H) = \begin{cases} (T, \phi) & : \text{H is a singleton as } \{T\} \\ (T_1, H') & : Root(T_1) < Root(T') \\ (T', \{T_1\} \cup H'') & : otherwise \end{cases} \quad (10.5)$$

where

$$\begin{array}{ll} H = \{T_1, T_2, ...\} & \text{for the non-empty forest case;} \\ H' = \{T_2, T_3, ...\} & \text{is the forest without the first tree;} \\ (T', H'') = extractMin(H') & \end{array}$$

The result of this function is a tuple. The first part is the tree which has the minimum element at root, the second part is the rest of the trees after remove the first part from the forest.

This function examine each of the trees in the forest thus is bound to $O(\lg N)$ time.

The relative Haskell program can be give respectively.

```
extractMin :: (Ord a) ⇒ BiHeap a → (BiTree a, BiHeap a)
extractMin [t] = (t, [])
extractMin (t:ts) = if root t < root t' then (t, ts)
                    else (t', t:ts')
    where
      (t', ts') = extractMin ts
```

With this function defined, to return the minimum element is trivial.

```
findMin :: (Ord a) ⇒ BiHeap a → a
findMin = root ∘ fst. extractMin
```

Of course, it's possible to just traverse forest and pick the minimum root without remove the tree for this purpose. Below imperative algorithm describes it with 'left child, right sibling' approach.

```
1: function Find-Minimum($H$)
2:     $T \leftarrow$ Head($H$)
3:     $min \leftarrow \infty$
4:     while $T \neq \phi$ do
5:         if Key($T$)$<$ $min$ then
6:             $min \leftarrow$ Key($T$)
7:         $T \leftarrow$ Sibling($T$)
8:     return $min$
```

While if we manage the children with collection containers, the link list traversing is abstracted as to find the minimum element among the list. The following Python program shows about this situation.

```
def find_min(ts):
    min_t = min(ts, key=lambda t: t.key)
    return min_t.key
```

Next we define the function to delete the minimum element from the heap by using *extractMin*.

$$delteMin(H) = merge(reverse(Children(T)), H') \tag{10.6}$$

where

$$(T, H') = extractMin(H)$$

Translate the formula to Haskell program is trivial and we'll skip it.

To realize the algorithm in procedural way takes extra efforts including list reversing etc. We left these details as exercise to the reader. The following pseudo code illustrate the imperative pop algorithm

```
1: function Extract-Min($H$)
2:     $(T_{min}, H) \leftarrow$ Extract-Min-Tree($H$)
3:     $H \leftarrow$ Merge($H$, Reverse(Children($T_{min}$)))
4:     return (Key($T_{min}$), $H$)
```

With pop operation defined, we can realize heap sort by creating a binomial heap from a series of numbers, than keep popping the smallest number from the heap till it becomes empty.

$$sort(xs) = heapSort(fromList(xs)) \tag{10.7}$$

And the real work is done in function *heapSort*.

$$heapSort(H) = \begin{cases} \phi & : H = \phi \\ \{findMin(H)\} \cup heapSort(deleteMin(H)) & : otherwise \end{cases} \tag{10.8}$$

Translate to Haskell yields the following program.

```
heapSort :: (Ord a) ⇒ [a] → [a]
heapSort = hsort ∘ fromList where
    hsort [] = []
    hsort h = (findMin h):(hsort $ deleteMin h)
```

Function fromList can be defined by folding. Heap sort can also be expressed in procedural way respectively. Please refer to previous chapter about binary heap for detail.

Exercise 10.4

- Write the program to return the minimum element from a binomial heap in your favorite imperative programming language with 'left-child, right-sibling' approach.
- Realize the Extract-Min-Tree() Algorithm.
- For 'left-child, right-sibling' approach, reversing all children of a tree is actually reversing a single-direct linked-list. Write a program to reverse such linked-list in your favorite imperative programming language.

More words about binomial heap

As what we have shown that insertion and merge are bound to $O(\lg N)$ time. The results are all ensure for the *worst case*. The amortized performance are $O(1)$. We skip the proof for this fact.

10.3 Fibonacci Heaps

It's interesting that why the name is given as 'Fibonacci heap'. In fact, there is no direct connection from the structure design to Fibonacci series. The inventors of 'Fibonacci heap', Michael L. Fredman and Robert E. Tarjan, utilized the property of Fibonacci series to prove the performance time bound, so they decided to use Fibonacci to name this data structure.[2]

10.3.1 Definition

Fibonacci heap is essentially a lazy evaluated binomial heap. Note that, it doesn't mean implementing binomial heap in lazy evaluation settings, for instance Haskell, brings Fibonacci heap automatically. However, lazy evaluation setting does help in realization. For example in [5], presents a elegant implementation.

Fibonacci heap has excellent performance theoretically. All operations except for pop are bound to amortized $O(1)$ time. In this section, we'll give an algorithm different from some popular textbook[2]. Most of the ideas present here are based on Okasaki's work[6].

Let's review and compare the performance of binomial heap and Fibonacci heap (more precisely, the performance goal of Fibonacci heap).

operation	Binomial heap	Fibonacci heap
insertion	$O(\lg N)$	$O(1)$
merge	$O(\lg N)$	$O(1)$
top	$O(\lg N)$	$O(1)$
pop	$O(\lg N)$	amortized $O(\lg N)$

Consider where is the bottleneck of inserting a new element x to binomial heap. We actually wrap x as a singleton leaf and insert this tree into the heap which is actually a forest.

During this operation, we inserted the tree in monotonically increasing order of rank, and once the rank is equal, recursively linking and inserting will happen, which lead to the $O(\lg N)$ time.

As the lazy strategy, we can postpone the ordered-rank insertion and merging operations. On the contrary, we just put the singleton leaf to the forest. The problem is that when we try to find the minimum element, for example the top operation, the performance will be bad, because we need check all trees in the forest, and there aren't only $O(\lg N)$ trees.

In order to locate the top element in constant time, we must remember where is the tree contains the minimum element as root.

Based on this idea, we can reuse the definition of binomial tree and give the definition of Fibonacci heap as the following Haskell program for example.

```
data BiTree a = Node { rank :: Int
                     , root :: a
                     , children :: [BiTree a]}
```

The Fibonacci heap is either empty or a forest of binomial trees with the minimum element stored in a special one explicitly.

```
data FibHeap a = E | FH { size :: Int
                        , minTree :: BiTree a
                        , trees :: [BiTree a]}
```

For convenient purpose, we also add a size field to record how many elements are there in a heap.

The data layout can also be defined in imperative way as the following ANSI C code.

```
struct node{
  Key key;
  struct node *next, *prev, *parent, *children;
  int degree; /* As known as rank */
  int mark;
};

struct FibHeap{
  struct node *roots;
  struct node *minTr;
  int n; /* number of nodes */
};
```

For generality, Key can be a customized type, we use integer for illustration purpose.

```
typedef int Key;
```

In this chapter, we use the circular doubly linked-list for imperative settings to realize the Fibonacci Heap as described in [2]. It makes many operations easy and fast. Note that, there are two extra fields added. A *degree* also known as *rank* for a node is the number of children of this node; Flag *mark* is used only in decreasing key operation. It will be explained in detail in later section.

10.3.2 Basic heap operations

As we mentioned that Fibonacci heap is essentially binomial heap implemented in a lazy evaluation strategy, we'll reuse many algorithms defined for binomial heap.

Insert a new element to the heap

Recall the insertion algorithm of binomial tree. It can be treated as a special case of merge operation, that one heap contains only a singleton tree. So that the inserting algorithm can be defined by means of merging.

$$insert(H, x) = merge(H, singleton(x)) \tag{10.9}$$

where singleton is an auxiliary function to wrap an element to a one-leaf-tree.

$$singleton(x) = FibHeap(1, node(1, x, \phi), \phi)$$

Note that function $FibHeap()$ accepts three parameters, a size value, which is 1 for this one-leaf-tree, a special tree which contains the minimum element as root, and a list of other binomial trees in the forest. The meaning of function $node()$ is as same as before, that it creates a binomial tree from a rank, an element, and a list of children.

Insertion can also be realized directly by appending the new node to the forest and updated the record of the tree which contains the minimum element.

```
1: function INSERT($H, k$)
2:     $x \leftarrow$ SINGLETON($k$)                    ▷ Wrap $x$ to a node
3:     append $x$ to root list of $H$
4:     if $T_{min}(H) = NIL \lor k <$ KEY($T_{min}(H)$) then
5:         $T_{min}(H) \leftarrow x$
6:     N($H$) $\leftarrow$ N($H$)+1
```

Where function $T_{min}()$ returns the tree which contains the minimum element at root.

The following C source snippet is a translation for this algorithm.

```
struct FibHeap* insert_node(struct FibHeap* h, struct node* x){
  h = add_tree(h, x);
  if(h→minTr == NULL || x→key < h→minTr→key)
    h→minTr = x;
  h→n++;
  return h;
}
```

Exercise 10.5

Implement the insert algorithm in your favorite imperative programming language completely. This is also an exercise to circular doubly linked list manipulation.

Merge two heaps

Different with the merging algorithm of binomial heap, we post-pone the linking operations later. The idea is to just put all binomial trees from each heap together, and choose one special tree which record the minimum element for the result heap.

$$merge(H_1, H_2) = \begin{cases} H_1 & : H_2 = \phi \\ H_2 & : H_1 = \phi \\ FibHeap(s_1 + s_2, T_{1min}, \{T_{2min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : root(T_{1min}) < root(T_{2min}) \\ FibHeap(s_1 + s_2, T_{2min}, \{T_{1min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : otherwise \end{cases} \tag{10.10}$$

where s_1 and s_2 are the size of H_1 and H_2; T_{1min} and T_{2min} are the special trees with minimum element as root in H_1 and H_2 respectively; $\mathbb{T}_1 = \{T_{11}, T_{12}, ...\}$ is a forest contains all other binomial trees in H_1; while $\mathbb{T}_2$ has the same meaning as $\mathbb{T}_1$ except that it represents the forest in H_2. Function $root(T)$ return the root element of a binomial tree.

Note that as long as the $\cup$ operation takes constant time, these *merge* algorithm is bound to $O(1)$. The following Haskell program is the translation of this algorithm.

```haskell
merge:: (Ord a) ⇒ FibHeap a → FibHeap a → FibHeap a
merge h E = h
merge E h = h
merge h1@(FH sz1 minTr1 ts1) h2@(FH sz2 minTr2 ts2)
    | root minTr1 < root minTr2 = FH (sz1+sz2) minTr1 (minTr2:ts2++ts1)
    | otherwise = FH (sz1+sz2) minTr2 (minTr1:ts1++ts2)
```

Merge algorithm can also be realized imperatively by concatenating the root lists of the two heaps.

```
1: function MERGE($H_1, H_2$)
2:     $H \leftarrow \Phi$
3:     ROOT($H$) ← CONCAT(ROOT($H_1$), ROOT($H_2$))
4:     if KEY($T_{min}(H_1)$) < KEY($T_{min}(H_2)$) then
5:         $T_{min}(H) \leftarrow T_{min}(H_1)$
6:     else
7:         $T_{min}(H) \leftarrow T_{min}(H_2)$
       N($H$) = N($H_1$) + N($H_2$)
8:     release $H_1$ and $H_2$
9:     return $H$
```

This function assumes neither H_1, nor H_2 is empty. And it's easy to add handling to these special cases as the following ANSI C program.

```c
struct FibHeap* merge(struct FibHeap* h1, struct FibHeap* h2){
  struct FibHeap* h;
  if(is_empty(h1))
```

```
    return h2;
  if(is_empty(h2))
    return h1;
  h = empty();
  h→roots = concat(h1→roots, h2→roots);
  if(h1→minTr→key < h2→minTr→key)
    h→minTr = h1→minTr;
  else
    h→minTr = h2→minTr;
  h→n = h1→n + h2→n;
  free(h1);
  free(h2);
  return h;
}
```

With *merge* function defined, the $O(1)$ insertion algorithm is realized as well. And we can also give the $O(1)$ time top function as below.

$$top(H) = root(T_{min}) \tag{10.11}$$

Exercise 10.6

Implement the circular doubly linked list concatenation function in your favorite imperative programming language.

Extract the minimum element from the heap (pop)

The pop (delete the minimum element) operation is the most complex one in Fibonacci heap. Since we postpone the tree consolidation in merge algorithm. We have to compensate it somewhere. Pop is the only place left as we have defined, insert, merge, top already.

There is an elegant procedural algorithm to do the tree consolidation by using an auxiliary array[2]. We'll show it later in imperative approach section.

In order to realize the purely functional consolidation algorithm, let's first consider a similar number puzzle.

Given a list of numbers, such as $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$, we want to add any two values if they are same. And repeat this procedure till all numbers are unique. The result of the example list should be $\{8, 16\}$ for instance.

One solution to this problem will as the following.

$$consolidate(L) = fold(meld, \phi, L) \tag{10.12}$$

Where $fold()$ function is defined to iterate all elements from a list, applying a specified function to the intermediate result and each element. it is sometimes called as *reducing*. Please refer to the chapter of binary search tree for it.

$L = \{x_1, x_2, ..., x_n\}$, denotes a list of numbers; and we'll use $L' = \{x_2, x_3, ..., x_n\}$ to represent the rest of the list with the first element removed. Function $meld()$ is defined as below.

$$meld(L, x) = \begin{cases} \{x\} & : L = \phi \\ meld(L', x + x_1) & : x = x_1 \\ \{x\} \cup L & : x < x_1 \\ \{x_1\} \cup meld(L', x) & : otherwise \end{cases} \tag{10.13}$$

Table 10.1: Steps of consolidate numbers

number	intermediate result	result
2	2	2
1	1, 2	1, 2
1	(1+1), 2	4
4	(4+4)	8
8	(8+8)	16
1	1, 16	1, 16
1	(1+1), 16	2, 16
2	(2+2), 16	4, 16
4	(4+4), 16	8, 16

The *consolidate*() function works as the follows. It maintains an ordered result list L, contains only unique numbers, which is initialized from an empty list ϕ. Each time it process an element x, it firstly check if the first element in L is equal to x, if so, it will add them together (which yields $2x$), and repeatedly check if $2x$ is equal to the next element in L. This process won't stop until either the element to be melt is not equal to the head element in the rest of the list, or the list becomes empty. Table 10.1 illustrates the process of consolidating number sequence $\{2, 1, 1, 4, 8, 1, 1, 2, 4\}$. Column one lists the number 'scanned' one by one; Column two shows the intermediate result, typically the new scanned number is compared with the first number in result list. If they are equal, they are enclosed in a pair of parentheses; The last column is the result of meld, and it will be used as the input to next step processing.

The Haskell program can be give accordingly.

```
consolidate = foldl meld [] where
    meld [] x = [x]
    meld (x':xs) x | x == x' = meld xs (x+x')
                   | x < x'  = x:x':xs
                   | otherwise = x': meld xs x
```

We'll analyze the performance of consolidation as a part of pop operation in later section.

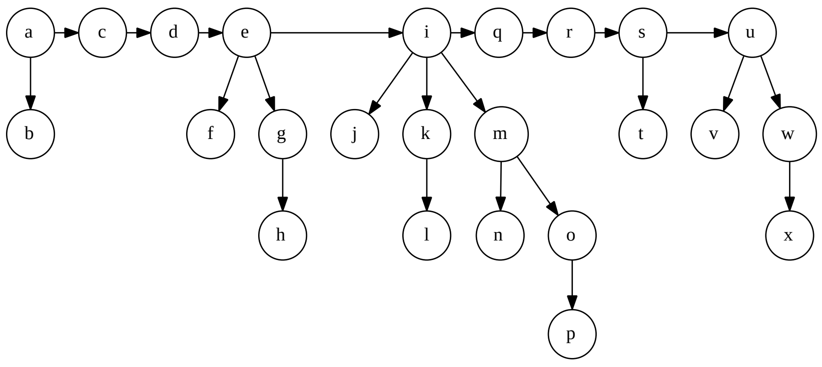
The tree consolidation is very similar to this algorithm except it performs based on rank. The only thing we need to do is to modify *meld*() function a bit, so that it compare on ranks and do linking instead of adding.

$$meld(L, x) = \begin{cases} \{x\} & : L = \phi \\ meld(L', link(x, x_1)) & : rank(x) = rank(x_1) \\ \{x\} \cup L & : rank(x) < rank(x_1) \\ \{x_1\} \cup meld(L', x) & : otherwise \end{cases} \quad (10.14)$$

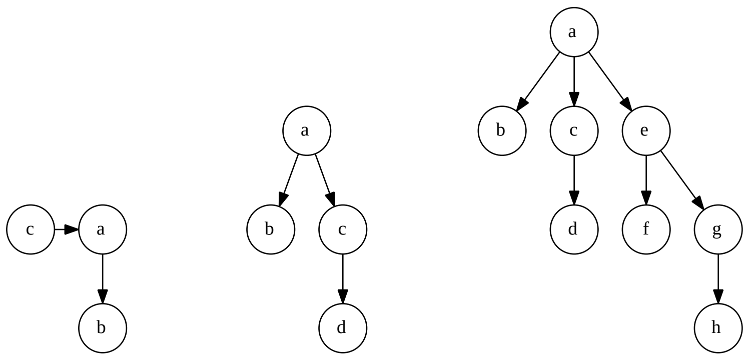
The final consolidate Haskell program changes to the below version.

```
consolidate :: (Ord a) => [BiTree a] -> [BiTree a]
consolidate = foldl meld [] where
    meld [] t = [t]
    meld (t':ts) t | rank t == rank t' = meld ts (link t t')
                   | rank t <  rank t' = t:t':ts
                   | otherwise = t' : meld ts t
```

Figure 10.9 and 10.10 show the steps of consolidation when processing a Fibonacci Heap contains different ranks of trees. Comparing with table 10.1 reveals the similarity.



(a) Before consolidation



(b) Step 1, 2 (c) Step 3, 'd' is firstly linked to 'c', then repeatedly linked to 'a'. (d) Step 4

Figure 10.9: Steps of consolidation

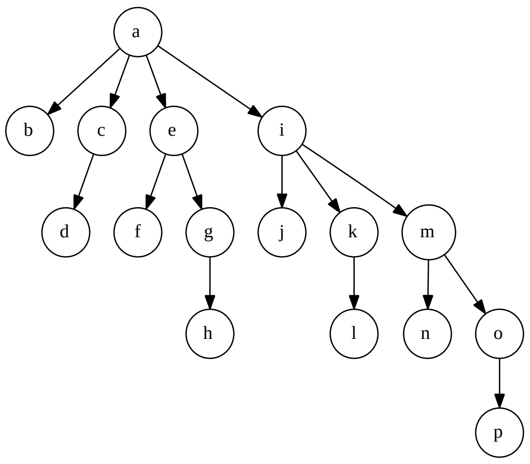
After we merge all binomial trees, including the special tree record for the minimum element in root, in a Fibonacci heap, the heap becomes a Binomial heap. And we lost the special tree, which gives us the ability to return the top element in $O(1)$ time.

It's necessary to perform a $O(\lg N)$ time search to resume the special tree. We can reuse the function $extractMin()$ defined for Binomial heap.

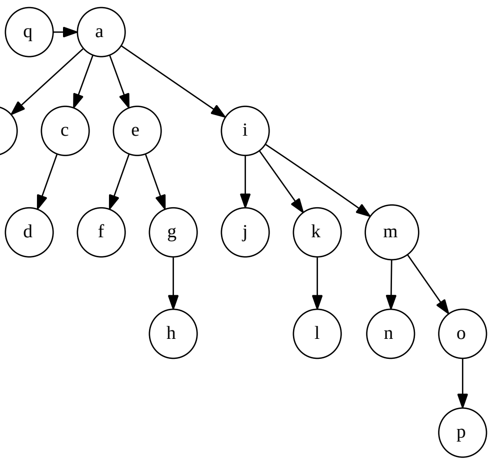
It's time to give the final pop function for Fibonacci heap as all the sub problems have been solved. Let T_{min} denote the special tree in the heap to record the minimum element in root; $\mathbb{T}$ denote the forest contains all the other trees except for the special tree, s represents the size of a heap, and function $children()$ returns all sub trees except the root of a binomial tree.

$$deleteMin(H) = \begin{cases} \phi & : \mathbb{T} = \phi \wedge children(T_{min}) = \phi \\ FibHeap(s-1, T'_{min}, \mathbb{T}') & : otherwise \end{cases} \tag{10.15}$$

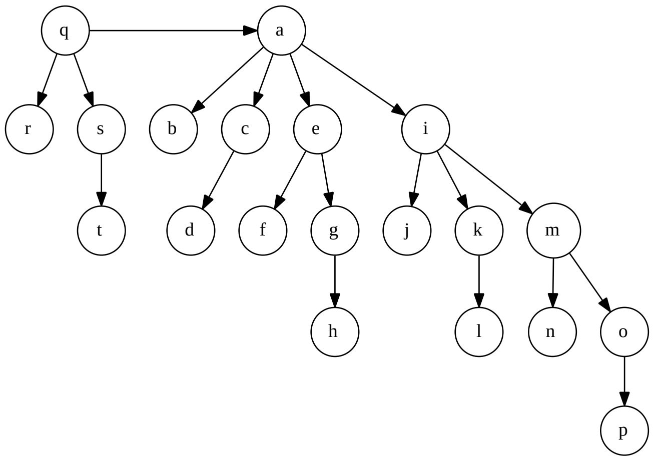
Where



(a) Step 5



(b) Step 6



(c) Step 7, 8, 'r' is firstly linked to 'q', then 's' is linked to 'q'.

Figure 10.10: Steps of consolidation

$$(T'_{min}, \mathbb{T}') = extractMin(consolidate(children(T_{min}) \cup \mathbb{T}))$$

Translate to Haskell yields the below program.

```
deleteMin :: (Ord a) ⇒ FibHeap a → FibHeap a
deleteMin (FH _ (Node _ x []) []) = E
deleteMin h@(FH sz minTr ts) = FH (sz-1) minTr' ts' where
    (minTr', ts') = extractMin $ consolidate (children minTr ++ ts)
```

The main part of the imperative realization is similar. We cut all children of T_{min} and append them to root list, then perform consolidation to merge all trees with the same rank until all trees are unique in term of rank.

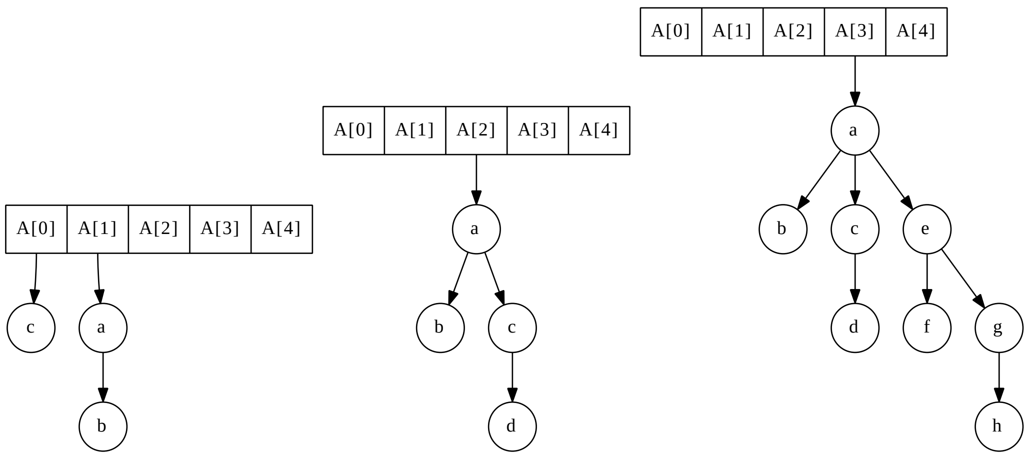
```
function DELETE-MIN($H$)
    $x \leftarrow T_{min}(H)$
    if $x \neq NIL$ then
        for each $y \in$ CHILDREN($x$) do
            append $y$ to root list of $H$
            PARENT($y$) $\leftarrow NIL$
        remove $x$ from root list of $H$
        N($H$) $\leftarrow$ N($H$) - 1
        CONSOLIDATE($H$)
    return $x$
```

Algorithm CONSOLIDATE utilizes an auxiliary array A to do the merge job. Array $A[i]$ is defined to store the tree with rank (degree) i. During the traverse of root list, if we meet another tree of rank i, we link them together to get a new tree of rank $i + 1$. Next we clean $A[i]$, and check if $A[i + 1]$ is empty and perform further linking if necessary. After we finish traversing all roots, array A stores all result trees and we can re-construct the heap from it.

```
function CONSOLIDATE($H$)
    $D \leftarrow$ MAX-DEGREE(N($H$))
    for $i \leftarrow 0$ to $D$ do
        $A[i] \leftarrow NIL$
    for each $x \in$ root list of $H$ do
        remove $x$ from root list of $H$
        $d \leftarrow$ DEGREE($x$)
        while $A[d] \neq NIL$ do
            $y \leftarrow A[d]$
            $x \leftarrow$ LINK($x, y$)
            $A[d] \leftarrow NIL$
            $d \leftarrow d + 1$
        $A[d] \leftarrow x$
    $T_{min}(H) \leftarrow NIL$                    ▷ root list is NIL at the time
    for $i \leftarrow 0$ to $D$ do
        if $A[i] \neq NIL$ then
            append $A[i]$ to root list of $H$.
            if $T_{min} = NIL \vee$ KEY($A[i]$) $<$ KEY($T_{min}(H)$) then
                $T_{min}(H) \leftarrow A[i]$
```

The only unclear sub algorithm is MAX-DEGREE, which can determine the upper bound of the degree of any node in a Fibonacci Heap. We'll delay the realization of it to the last sub section.

Feed a Fibonacci Heap shown in Figure 10.9 to the above algorithm, Figure 10.11, 10.12 and 10.13 show the result trees stored in auxiliary array A in every steps.



(a) Step 1, 2

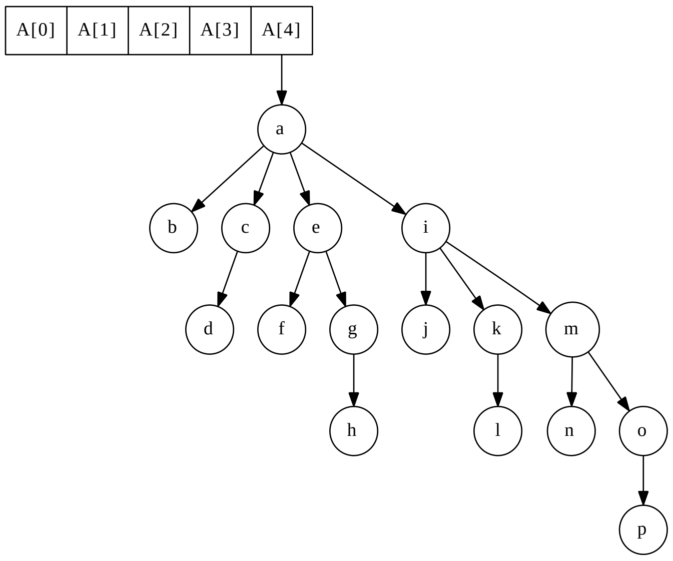
(b) Step 3, Since $A_0 \neq NIL$, 'd' is firstly linked to 'c', and clear A_0 to NIL. Again, as $A_1 \neq NIL$, 'c' is linked to 'a' and the new tree is stored in A_2.

(c) Step 4

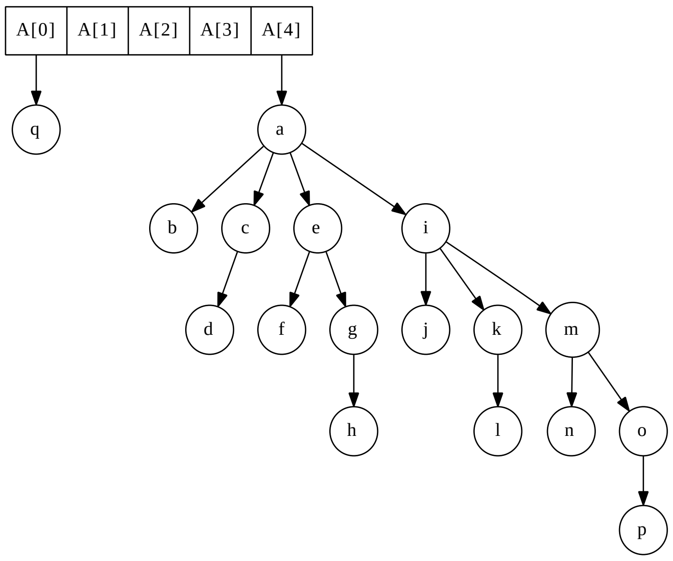
Figure 10.11: Steps of consolidation

Translate the above algorithm to ANSI C yields the below program.

```
void consolidate(struct FibHeap* h){
  if(!h→roots)
    return;
  int D = max_degree(h→n)+1;
  struct node *x, *y;
  struct node** a = (struct node**)malloc(sizeof(struct node*)*(D+1));
  int i, d;
  for(i=0; i≤D; ++i)
    a[i] = NULL;
  while(h→roots){
    x = h→roots;
    h→roots = remove_node(h→roots, x);
    d= x→degree;
    while(a[d]){
      y = a[d];  /* Another node has the same degree as x */
      x = link(x, y);
      a[d++] = NULL;
    }
    a[d] = x;
  }
  h→minTr = h→roots = NULL;
  for(i=0; i≤D; ++i)
    if(a[i]){
```

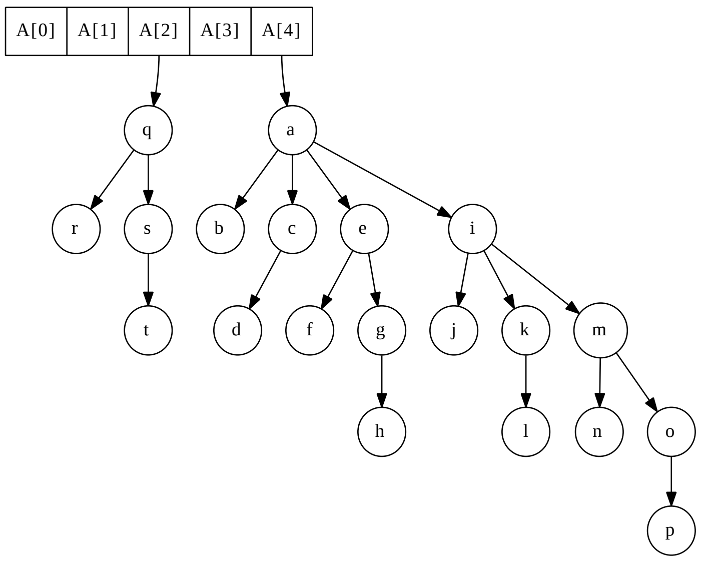


(a) Step 5



(b) Step 6

Figure 10.12: Steps of consolidation



(a) Step 7, 8, Since $A_0 \neq NIL$, 'r' is firstly linked to 'q', and the new tree is stored in A_1 (A_0 is cleared); then 's' is linked to 'q', and stored in A_2 (A_1 is cleared).

Figure 10.13: Steps of consolidation

```
        h→roots = append(h→roots, a[i]);
        if(h→minTr == NULL || a[i]→key < h→minTr→key)
          h→minTr = a[i];
    }
  free(a);
}
```

Exercise 10.7

Implement the remove function for circular doubly linked list in your favorite imperative programming language.

10.3.3 Running time of pop

In order to analyze the amortize performance of pop, we adopt potential method. Reader can refer to [2] for a formal definition. In this chapter, we only give a intuitive illustration.

Remind the gravity potential energy, which is defined as

$$E = M \cdot g \cdot h$$

Suppose there is a complex process, which moves the object with mass M up and down, and finally the object stop at height h'. And if there exists friction resistance W_f, We say the process works the following power.

$$W = M \cdot g \cdot (h' - h) + W_f$$

Figure 10.14 illustrated this concept.

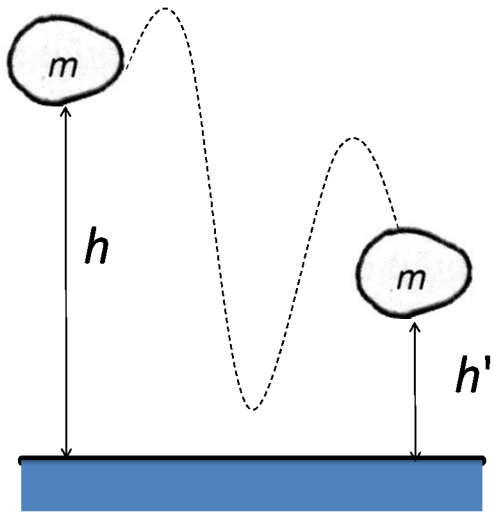


Figure 10.14: Gravity potential energy.

We treat the Fibonacci heap pop operation in a similar way, in order to evaluate the cost, we firstly define the potential $\Phi(H)$ before extract the minimum element. This potential is accumulated by insertion and merge operations executed so far. And after tree consolidation and we get the result H', we then calculate the new potential $\Phi(H')$. The difference between $\Phi(H')$ and $\Phi(H)$ plus the contribution of consolidate algorithm indicates the amortized performance of pop.

For pop operation analysis, the potential can be defined as

$$\Phi(H) = t(H) \tag{10.16}$$

Where $t(H)$ is the number of trees in Fibonacci heap forest. We have $t(H) = 1 + length(\mathbb{T})$ for any non-empty heap.

For the N-nodes Fibonacci heap, suppose there is an upper bound of ranks for all trees as $D(N)$. After consolidation, it ensures that the number of trees in the heap forest is at most $D(N) + 1$.

Before consolidation, we actually did another important thing, which also contribute to running time, we removed the root of the minimum tree, and concatenate all children left to the forest. So consolidate operation at most processes $D(N) + t(H) - 1$ trees.

Summarize all the above factors, we deduce the amortized cost as below.

$$\begin{array}{rcl} T & = & T_{consolidation} + \Phi(H') - \Phi(H) \\ & = & O(D(N) + t(H) - 1) + (D(N) + 1) - t(H) \\ & = & O(D(N)) \end{array} \tag{10.17}$$

If only insertion, merge, and pop function are applied to Fibonacci heap. We ensure that all trees are binomial trees. It is easy to estimate the upper limit $D(N)$ if $O(\lg N)$. (Suppose the extreme case, that all nodes are in only one Binomial tree).

However, we'll show in next sub section that, there is operation can violate the binomial tree assumption.

10.3.4 Decreasing key

There is a special heap operation left. It only makes sense for imperative settings. It's about decreasing key of a certain node. Decreasing key plays important role in some Graphic algorithms such as Minimum Spanning tree algorithm and Dijkstra's algorithm [2]. In that case we hope the decreasing key takes O(1) amortized time.

However, we can't define a function like $Decrease(H, k, k')$, which first locates a node with key k, then decrease k to k' by replacement, and then resume the heap properties. This is because the time for locating phase is bound to $O(N)$ time, since we don't have a pointer to the target node.

In imperative setting, we can define the algorithm as DECREASE-KEY(H, x, k). Here x is a node in heap H, which we want to decrease its key to k. We needn't perform a search, as we have x at hand. It's possible to give an amortized $O(1)$ solution.

When we decreased the key of a node, if it's not a root, this operation may violate the property Binomial tree that the key of parent is less than all keys of children. So we need to compare the decreased key with the parent node, and if this case happens, we can cut this node and append it to the root list. (Remind the recursive swapping solution for binary heap which leads to $O(\lg N)$)

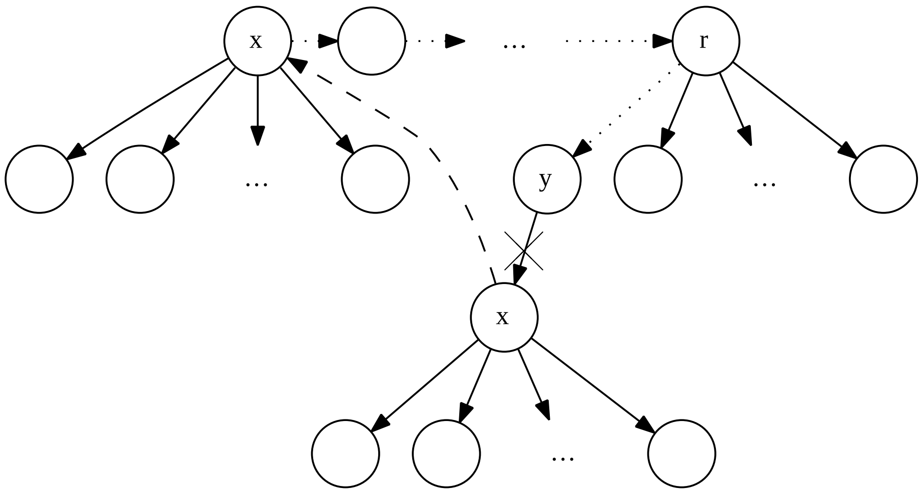


Figure 10.15: $x < y$, cut tree x from its parent, and add x to root list.

Figure 10.15 illustrates this situation. After decreasing key of node x, it is less than y, we cut x off its parent y, and 'past' the whole tree rooted at x to root list.

Although we recover the property of that parent is less than all children, the tree isn't any longer a Binomial tree after it losses some sub tree. If a tree losses too many of its children because of cutting, we can't ensure the performance of merge-able heap operations. Fibonacci Heap adds another constraints to avoid such problem:

If a node losses its second child, it is immediately cut from parent, and added to root list

The final DECREASE-KEY algorithm is given as below.

```
1: function DECREASE-KEY($H, x, k$)
2:     KEY($x$) $\leftarrow k$
3:     $p \leftarrow$ PARENT($x$)
4:     if $p \neq NIL \land k <$ KEY($p$) then
5:         CUT($H, x$)
6:         CASCADING-CUT($H, p$)
7:     if $k <$ KEY($T_{min}(H)$) then
8:         $T_{min}(H) \leftarrow x$
```

Where function CASCADING-CUT uses the mark to determine if the node is losing the second child. the node is marked after it losses the first child. And the mark is cleared in CUT function.

```
1: function CUT($H, x$)
2:     $p \leftarrow$ PARENT($x$)
3:     remove $x$ from $p$
4:     DEGREE($p$) $\leftarrow$ DEGREE($p$) - 1
5:     add $x$ to root list of $H$
6:     PARENT($x$) $\leftarrow NIL$
7:     MARK($x$) $\leftarrow FALSE$
```

During cascading cut process, if x is marked, which means it has already lost one child. We recursively performs cut and cascading cut on its parent till reach to root.

```
1: function CASCADING-CUT($H, x$)
2:     $p \leftarrow$ PARENT($x$)
3:     if $p \neq NIL$ then
4:         if MARK($x$) $= FALSE$ then
5:             MARK($x$) $\leftarrow TRUE$
6:         else
7:             CUT($H, x$)
8:             CASCADING-CUT($H, p$)
```

The relevant ANSI C decreasing key program is given as the following.

```
void decrease_key(struct FibHeap* h, struct node* x, Key k){
  struct node* p = x→parent;
  x→key = k;
  if(p && k < p→key){
    cut(h, x);
    cascading_cut(h, p);
  }
  if(k < h→minTr→key)
    h→minTr = x;
}

void cut(struct FibHeap* h, struct node* x){
  struct node* p = x→parent;
  p→children = remove_node(p→children, x);
  p→degree--;
  h→roots = append(h→roots, x);
  x→parent = NULL;
  x→mark = 0;
}

void cascading_cut(struct FibHeap* h, struct node* x){
  struct node* p = x->parent;
  if(p){
    if(!x->mark)
      x->mark = 1;
    else{
      cut(h, x);
      cascading_cut(h, p);
    }
  }
}
```

Exercise 10.8

Prove that Decrease-Key algorithm is amortized $O(1)$ time.

10.3.5 The name of Fibonacci Heap

It's time to reveal the reason why the data structure is named as 'Fibonacci Heap'.

There is only one undefined algorithm so far, Max-Degree(N). Which can determine the upper bound of degree for any node in a N nodes Fibonacci Heap. We'll give the proof by using Fibonacci series and finally realize Max-Degree algorithm.

Lemma 10.3.1. *For any node x in a Fibonacci Heap, denote $k = degree(x)$, and $|x| = size(x)$, then*

$$|x| \geq F_{k+2} \tag{10.18}$$

Where F_k is Fibonacci series defined as the following.

$$F_k = \begin{cases} 0 & : \quad k = 0 \\ 1 & : \quad k = 1 \\ F_{k-1} + F_{k-2} & : \quad k \geq 2 \end{cases}$$

Proof. Consider all k children of node x, we denote them as $y_1, y_2, ..., y_k$ in the order of time when they were linked to x. Where y_1 is the oldest, and y_k is the youngest.

Obviously, $y_i \geq 0$. When we link y_i to x, children $y_1, y_2, ..., y_{i-1}$ have already been there. And algorithm LINK only links nodes with the same degree. Which indicates at that time, we have

$$degree(y_i) = degree(x) = i - 1$$

After that, node y_i can at most lost 1 child, (due to the decreasing key operation) otherwise, if it will be immediately cut off and append to root list after the second child loss. Thus we conclude

$$degree(y_i) \geq i - 2$$

For any $i = 2, 3, ..., k$.

Let s_k be the *minimum possible size* of node x, where $degree(x) = k$. For trivial cases, $s_0 = 1$, $s_1 = 2$, and we have

$$\begin{aligned}
|x| &\geq s_k \\
&= 2 + \sum_{i=2}^{k} s_{degree(y_i)} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2}
\end{aligned}$$

We next show that $s_k > F_{k+2}$. This can be proved by induction. For trivial cases, we have $s_0 = 1 \geq F_2 = 1$, and $s_1 = 2 \geq F_3 = 2$. For induction case $k \geq 2$. We have

$$\begin{aligned}
|x| &\geq s_k \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
&\geq 2 + \sum_{i=2}^{k} F_i \\
&= 1 + \sum_{i=0}^{k} F_i
\end{aligned}$$

At this point, we need prove that

$$F_{k+2} = 1 + \sum_{i=}^{k} F_i \tag{10.19}$$

This can also be proved by using induction:

- Trivial case, $F_2 = 1 + F_0 = 2$
- Induction case,

$$\begin{aligned}
F_{k+2} &= F_{k+1} + F_k \\
&= 1 + \sum_{i=0}^{k-1} F_i + F_k \\
&= 1 + \sum_{i=0}^{k} F_i
\end{aligned}$$

Summarize all above we have the final result.

$$N \geq |x| \geq F_k + 2 \tag{10.20}$$

□

Recall the result of AVL tree, that $F_k \geq \Phi^k$, where $\Phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. We also proved that pop operation is amortized $O(\lg N)$ algorithm.

Based on this result. We can define Function $MaxDegree$ as the following.

$$MaxDegree(N) = 1 + \lfloor \log_{\Phi} N \rfloor \tag{10.21}$$

The imperative MAX-DEGREE algorithm can also be realized by using Fibonacci sequences.

```
1: function MAX-DEGREE($N$)
2:     $F_0 \leftarrow 0$
3:     $F_1 \leftarrow 1$
4:     $k \leftarrow 2$
5:     repeat
6:         $F_k \leftarrow F_{k_1} + F_{k_2}$
7:         $k \leftarrow k + 1$
8:     until $F_k < N$
9:     return $k - 2$
```

Translate the algorithm to ANSI C given the following program.

```
int max_degree(int n){
  int k, F;
  int F2 = 0;
  int F1 = 1;
  for(F=F1+F2, k=2; F<n; ++k){
    F2 = F1;
    F1 = F;
    F = F1 + F2;
  }
  return k-2;
}
```

10.4 Pairing Heaps

Although Fibonacci Heaps provide excellent performance theoretically, it is complex to realize. People find that the constant behind the big-O is big. Actually, Fibonacci Heap is more significant in theory than in practice.

In this section, we'll introduce another solution, Pairing heap, which is one of the best heaps ever known in terms of performance. Most operations including insertion, finding minimum element (top), merging are all bounds to $O(1)$ time, while deleting minimum element (pop) is conjectured to amortized $O(\lg N)$ time [7] [6]. Note that this is still a conjecture for 15 years by the time I write this chapter. Nobody has been proven it although there are much experimental data support the $O(\lg N)$ amortized result.

Besides that, pairing heap is simple. There exist both elegant imperative and functional implementations.

10.4.1 Definition

Both Binomial Heaps and Fibonacci Heaps are realized with forest. While a pairing heaps is essentially a K-ary tree. The minimum element is stored at root. All other elements are stored in sub trees.

The following Haskell program defines pairing heap.

```
data PHeap a = E | Node a [PHeap a]
```

This is a recursive definition, that a pairing heap is either empty or a K-ary tree, which is consist of a root node, and a list of sub trees.

Pairing heap can also be defined in procedural languages, for example ANSI C as below. For illustration purpose, all heaps we mentioned later are minimum-heap, and we assume the type of key is integer [4]. We use same linked-list based left-child, right-sibling approach (aka, binary tree representation[2]).

```
typedef int Key;

struct node{
  Key key;
  struct node *next, *children, *parent;
};
```

Note that the parent field does only make sense for decreasing key operation, which will be explained later on. we can omit it for the time being.

10.4.2 Basic heap operations

In this section, we first give the merging operation for pairing heap, which can be used to realize the insertion. Merging, insertion, and finding the minimum element are relative trivial compare to the extracting minimum element operation.

Merge, insert, and find the minimum element (top)

The idea of merging is similar to the linking algorithm we shown previously for Binomial heap. When we merge two pairing heaps, there are two cases.

- Trivial case, one heap is empty, we simply return the other heap as the result;
- Otherwise, we compare the root element of the two heaps, make the heap with bigger root element as a new children of the other.

Let H_1, and H_2 denote the two heaps, x and y be the root element of H_1 and H_2 respectively. Function $Children()$ returns the children of a K-ary tree. Function $Node()$ can construct a K-ary tree from a root element and a list of children.

$$merge(H_1, H_2) = \begin{cases} H_1 & : H_2 = \phi \\ H_2 & : H_1 = \phi \\ Node(x, \{H_2\} \cup Children(H_1)) & : x < y \\ Node(y, \{H_1\} \cup Children(H_2)) & : otherwise \end{cases} \quad (10.22)$$

Where

$$x = Root(H_1)$$
$$y = Root(H_2)$$

[4]We can parametrize the key type with C++ template, but this is beyond our scope, please refer to the example programs along with this book

It's obviously that merging algorithm is bound to $O(1)$ time [5]. The *merge* equation can be translated to the following Haskell program.

```
merge :: (Ord a) ⇒ PHeap a → PHeap a → PHeap a
merge h E = h
merge E h = h
merge h1@(Node x hs1) h2@(Node y hs2) =
    if x < y then Node x (h2:hs1) else Node y (h1:hs2)
```

Merge can also be realized imperatively. With left-child, right sibling approach, we can just link the heap, which is in fact a K-ary tree, with larger key as the first new child of the other. This is constant time operation as described below.

```
1: function MERGE($H_1, H_2$)
2:     if $H_1$ = NIL then
3:         return $H_2$
4:     if $H_2$ = NIL then
5:         return $H_1$
6:     if KEY($H_2$) < KEY($H_1$) then
7:         EXCHANGE($H_1 \leftrightarrow H_2$)
8:     Insert $H_2$ in front of CHILDREN($H_1$)
9:     PARENT($H_2$) $\leftarrow H_1$
10:    return $H_1$
```

Note that we also update the parent field accordingly. The ANSI C example program is given as the following.

```
struct node* merge(struct node* h1, struct node* h2){
  if(h1 == NULL)
    return h2;
  if(h2 == NULL)
    return h1;
  if(h2→key < h1→key)
    swap(&h1, &h2);
  h2→next = h1→children;
  h1→children = h2;
  h2→parent = h1;
  h1→next = NULL; /*Break previous link if any*/
  return h1;
}
```

Where function swap() is defined in a similar way as Fibonacci Heap.

With merge defined, insertion can be realized as same as Fibonacci Heap in Equation 10.9. Definitely it's $O(1)$ time operation. As the minimum element is always stored in root, finding it is trivial.

$$top(H) = Root(H) \tag{10.23}$$

Same as the other two above operations, it's bound to $O(1)$ time.

Exercise 10.9

Implement the insertion and top operation in your favorite programming language.

[5] Assume $\cup$ is constant time operation, this is true for linked-list settings, including 'cons' like operation in functional programming languages.

Decrease key of a node

There is another trivial operation, to decrease key of a given node, which only makes sense in imperative settings as we explained in Fibonacci Heap section.

The solution is simple, that we can cut the node with the new smaller key from it's parent along with all its children. Then merge it again to the heap. The only special case is that if the given node is the root, then we can directly set the new key without doing anything else.

The following algorithm describes this procedure for a given node x, with new key k.

```
1: function DECREASE-KEY($H, x, k$)
2:     KEY($x$) $\leftarrow k$
3:     if PARENT($x$) $\neq$ NIL then
4:         Remove $x$ from CHILDREN(PARENT($x$))
       PARENT($x$) $\leftarrow$ NIL
5:     return MERGE($H, x$)
```

The following ANSI C program translates this algorithm.

```
struct node* decrease_key(struct node* h, struct node* x, Key key){
  x→key = key; /* Assume key ≤ x→key */
  if(x→parent)
    x→parent→children = remove_node(x→parent→children, x);
  x→parent = NULL;
  return merge(h, x);
}
```

Exercise 10.10

Implement the program of removing a node from the children of its parent in your favorite imperative programming language. Consider how can we ensure the overall performance of decreasing key is O(1) time? Is left-child, right sibling approach enough?

Delete the minimum element from the heap (pop)

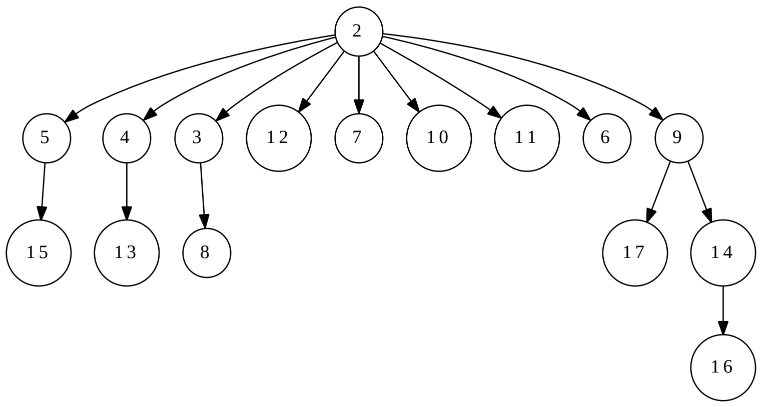
Since the minimum element is always stored at root, after delete it during popping, the rest things left are all sub-trees. These trees can be merged to one big tree.

$$pop(H) = mergePairs(Children(H)) \tag{10.24}$$

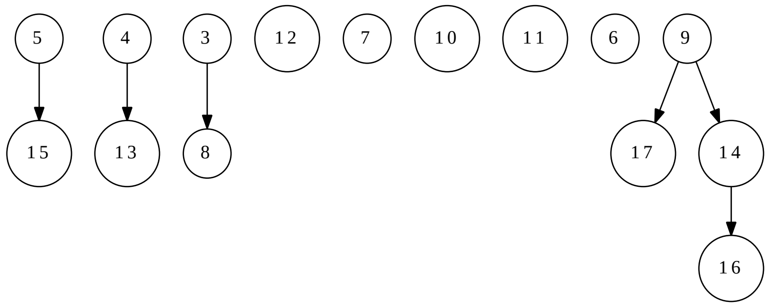
Pairing Heap uses a special approach that it merges every two sub-trees from left to right in pair. Then merge these paired results from right to left which forms a final result tree. The name of 'Pairing Heap' comes from the characteristic of this pair-merging.

Figure 10.16 and 10.17 illustrate the procedure of pair-merging.

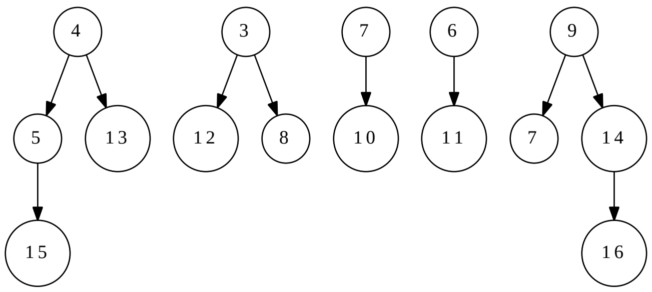
The recursive pair-merging solution is quite similar to the bottom up merge sort[6]. Denote the children of a pairing heap as A, which is a list of trees of $\{T_1, T_2, T_3, ..., T_m\}$ for example. The $mergePairs()$ function can be given as



(a) A pairing heap before pop.

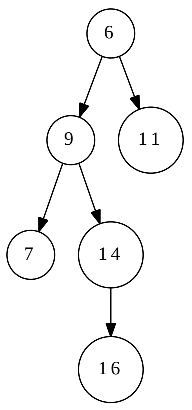


(b) After root element 2 being removed, there are 9 sub-trees left.

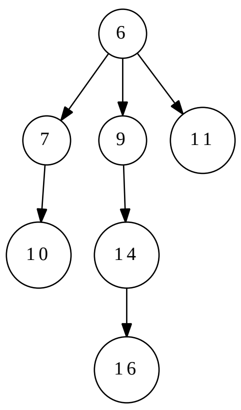


(c) Merge every two trees in pair, note that there are odd number trees, so the last one needn't merge.

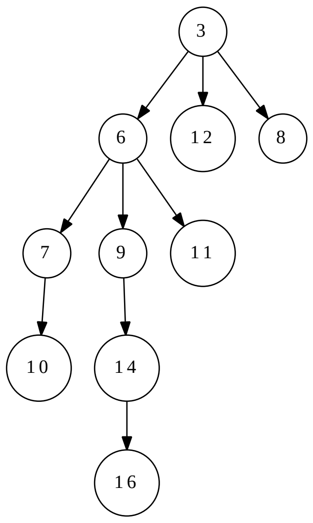
Figure 10.16: Remove the root element, and merge children in pairs.



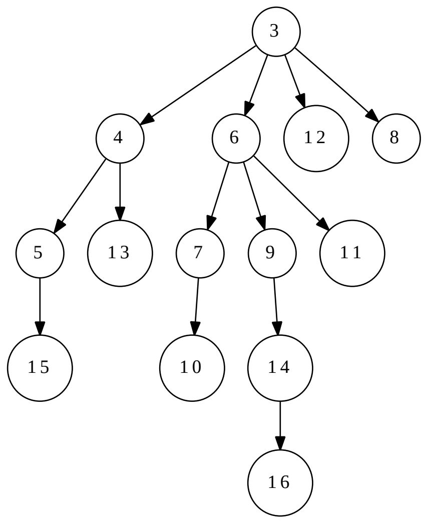
(a) Merge tree with 9, and tree with root 6.



(b) Merge tree with root 7 to the result.



(c) Merge tree with root 3 to the result.



(d) Merge tree with root 4 to the result.

Figure 10.17: Steps of merge from right to left.

below.

$$mergePairs(A) = \begin{cases} \Phi & : A = \Phi \\ T_1 & : A = \{T_1\} \\ merge(merge(T_1, T_2), mergePairs(A')) & : otherwise \end{cases} \tag{10.25}$$

where

$$A' = \{T_3, T_4, ..., T_m\}$$

is the rest of the children without the first two trees.

The relative Haskell program of popping is given as the following.

```haskell
deleteMin :: (Ord a) ⇒ PHeap a → PHeap a
deleteMin (Node _ hs) = mergePairs hs where
    mergePairs [] = E
    mergePairs [h] = h
    mergePairs (h1:h2:hs) = merge (merge h1 h2) (mergePairs hs)
```

The popping operation can also be explained in the following procedural algorithm.

```
1: function POP($H$)
2:     $L \leftarrow NIL$
3:     for every 2 trees $T_x$, $T_y \in$ CHILDREN($H$) from left to right do
4:         Extract $x$, and $y$ from CHILDREN($H$)
5:         $T \leftarrow$ MERGE($T_x, T_y$)
6:         Insert $T$ at the beginning of $L$
7:     $H \leftarrow$ CHILDREN($H$)                ▷ $H$ is either $NIL$ or one tree.
8:     for $\forall T \in L$ from left to right do
9:         $H \leftarrow$ MERGE($H, T$)
10:    return $H$
```

Note that L is initialized as an empty linked-list, then the algorithm iterates every two trees in pair in the children of the K-ary tree, from left to right, and performs merging, the result is inserted at the beginning of L. Because we insert to front end, so when we traverse L later on, we actually process from right to left. There may be odd number of sub-trees in H, in that case, it will leave one tree after pair-merging. We handle it by start the right to left merging from this left tree.

Below is the ANSI C program to this algorithm.

```c
struct node* pop(struct node* h){
  struct node *x, *y, *lst = NULL;
  while((x = h→children) != NULL){
    if((h→children = y = x→next) != NULL)
      h→children = h→children→next;
    lst = push_front(lst, merge(x, y));
  }
  x = NULL;
  while((y = lst) != NULL){
    lst = lst→next;
    x = merge(x, y);
  }
  free(h);
  return x;
}
```

The pairing heap pop operation is conjectured to be amortized $O(\lg N)$ time [7].

Exercise 10.11

Write a program to insert a tree at the beginning of a linked-list in your favorite imperative programming language.

Delete a node

We didn't mention delete in Binomial heap or Fibonacci Heap. Deletion can be realized by first decreasing key to minus infinity ($-\infty$), then performing pop. In this section, we present another solution for delete node.

The algorithm is to define the function $delete(H, x)$, where x is a node in a pairing heap H [6].

If x is root, we can just perform a pop operation. Otherwise, we can cut x from H, perform a pop on x, and then merge the pop result back to H. This can be described as the following.

$$delete(H,x) = \left\{ \begin{array}{rcl} pop(H) & : & x \text{ is root of } H \\ merge(cut(H,x), pop(x)) & : & otherwise \end{array} \right. \tag{10.26}$$

As delete algorithm uses pop, the performance is conjectured to be amortized $O(1)$ time.

Exercise 10.12

- Write procedural pseudo code for delete algorithm.
- Write the delete operation in your favorite imperative programming language
- Consider how to realize delete in purely functional setting.

10.5 Notes and short summary

In this chapter, we extend the heap implementation from binary tree to more generic approach. Binomial heap and Fibonacci heap use Forest of K-ary trees as under ground data structure, while Pairing heap use a K-ary tree to represent heap. It's a good point to post pone some expensive operation, so that the over all amortized performance is ensured. Although Fibonacci Heap gives good performance in theory, the implementation is a bit complex. It was removed in some latest textbooks. We also present pairing heap, which is easy to realize and have good performance in practice.

The elementary tree based data structures are all introduced in this book. There are still many tree based data structures which we can't covers them all and skip here. We encourage the reader to refer to other textbooks about them. From next chapter, we'll introduce generic sequence data structures, array and queue.

[6]Here the semantic of x is a reference to a node.