Binomial heap, Fibonacci heap, and pairing heap

Liu Xinyu *

February 21, 2012

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.

2 Binomial Heaps

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.

2.1.1 Binomial tree

In order to explain why the name of the tree is 'binomial', let's review the famous Pascal's triangle (in China, we call it Yang Hui's triangle) [4].

*Liu Xinyu
Email: liuxinyu95@gmail.com

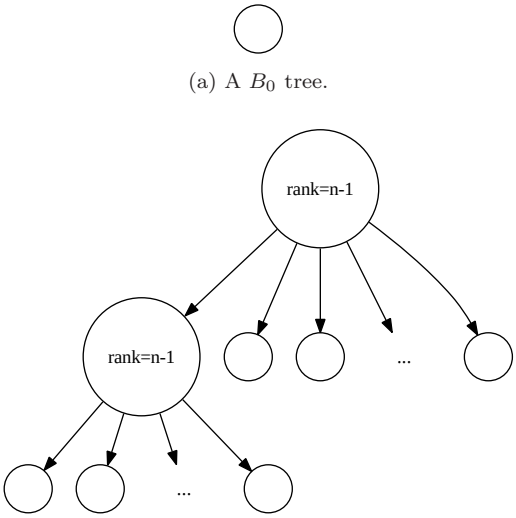
```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
...
```

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



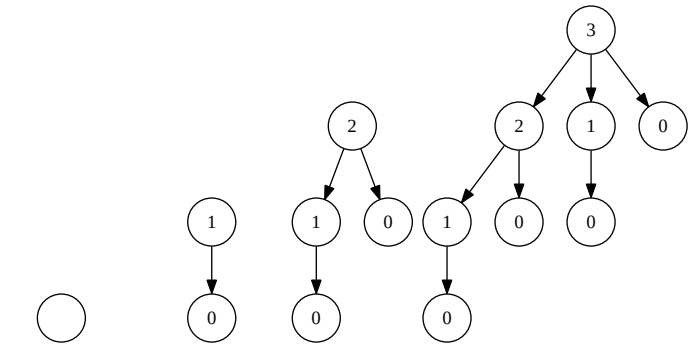
(a) A B_0 tree.

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

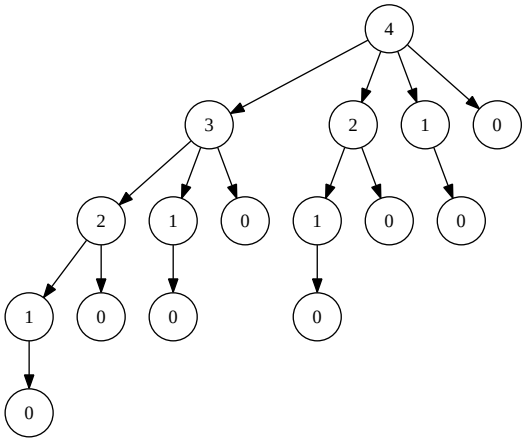
Figure 1: Recursive definition of binomial trees

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



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



...

(e) B_4 tree;

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

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; 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.

2.1.2 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 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.

2.1.3 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 4 shows an example tree represented in this way.

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.

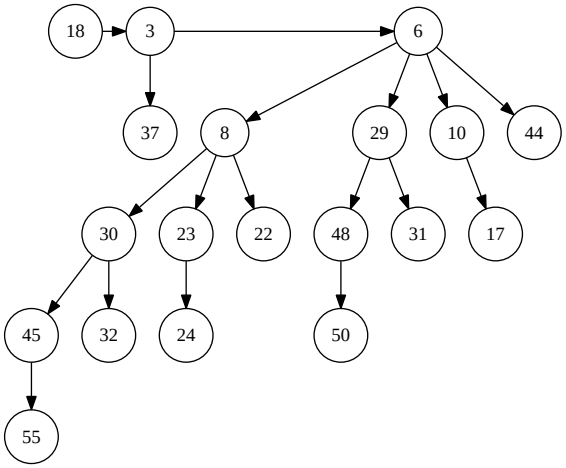


Figure 3: A binomial heap with 19 elements

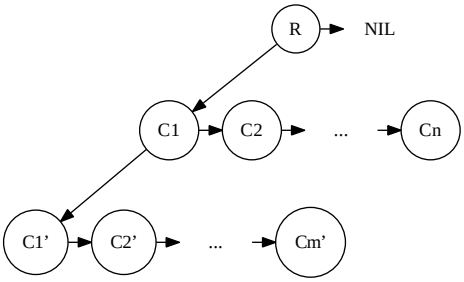


Figure 4: Example tree represented in 'left-child, right-sibling' way. R is the root node, it has no sibling, so it right side is pointed to NIL. $C_1, C_2, ..., C_n$ are children of R. C_1 is linked from the left side of R, other siblings of C_1 are linked one next to each other on the right side of C_1. $C'_2, ..., C'_m$ are children of C_1.

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 = []
```

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]
```

2.2 Basic heap operations

2.2.1 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

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

of all other children. Suppose function $Key(T)$, $Children(T)$, and $Rank(T)$ access the key, children and rank of a binomial tree respectively.

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

Where

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

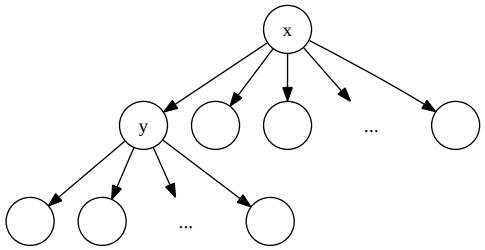


Figure 5: Suppose $x < y$, insert y as the first child of x.

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 6 shows the result of one case.

```
1: function LINK(T1, T2)
2:     if KEY(T2) < KEY(T1) then
3:         Exchange T1 ↔ T2
4:     SIBLING(T2) ← CHILD(T1)
5:     CHILD(T1) ← T2
6:     PARENT(T2) ← T1
7:     RANK(T1) ← RANK(T1) + 1
8:     return T1
```

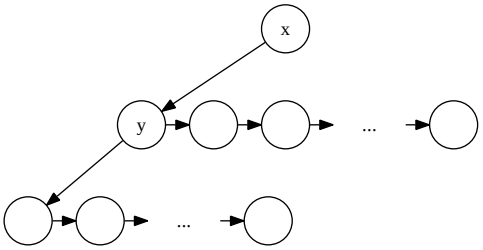


Figure 6: Suppose $x < y$, link y to the left side of x and link the original children of x to the right side of y.

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) $\leftarrow T_1$
5: INSERT-BEFORE(CHILDREN(T_1), T_2)
6: RANK(T_1) $\leftarrow$ 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
```

Exercise 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.

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

2.2.2 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

$$insertT(H, T) = \begin{cases} \{T\} & : & H = \phi \\ \{T\} \cup H & : & Rank(T) < Rank(T_1) \\ insertT(H', link(T, T_1)) & : & otherwise \end{cases} \quad (2)$$

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')
```

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.

$$insert(H, x) = insertT(H, node(0, x, \phi)) \quad (3)$$

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 []
```

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

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

```
1: function INSERT-TREE(H, T)
2:     while H ≠ φ∧ RANK(HEAD(H)) = RANK(T) do
3:         (T_1, H) ← EXTRACT-HEAD(H)
4:         T ← LINK(T, T_1)
5:     SIBLING(T) ← H
6:     return T
```

Algorithm 1 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 2.

Algorithm 2 Insert a tree with children managed by a container.

```
1: function INSERT-TREE'(H, T)
2:     while H ≠ φ∧ RANK(H[0]) = RANK(T) do
3:         T_1 ← POP(H)
4:         T ← 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 3 Imperative insert algorithm

```
1: function INSERT(H, x)
2:     return INSERT-TREE(H, NODE(0, x, φ))
```

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 2

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

2.2.3 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 7 illustrates the idea of this algorithm. This method is different from the one given in [2].

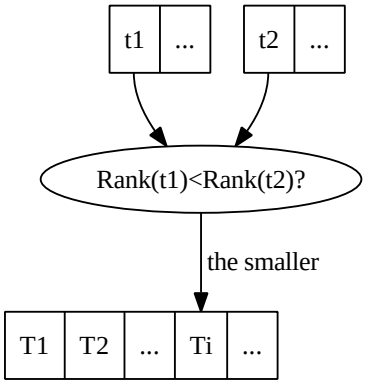
We can formalize this idea with a function. For non-empty cases, we denote the two heaps as $H_1 = \{T_1, T_2, ...\}$ and $H_2 = \{T'_1, T'_2, ...\}$. Let $H'_1 = \{T_2, T_3, ...\}$ and $H'_2 = \{T'_2, T'_3, ...\}$.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ \{T_1\} \cup merge(H'_1, H_2) & : & Rank(T_1) < Rank(T'_1) \\ \{T'_1\} \cup merge(H_1, H'_2) & : & Rank(T_1) > Rank(T'_1) \\ insertT(merge(H'_1, H'_2), link(T_1, T'_1)) & : & otherwise \end{cases} \tag{4}$$

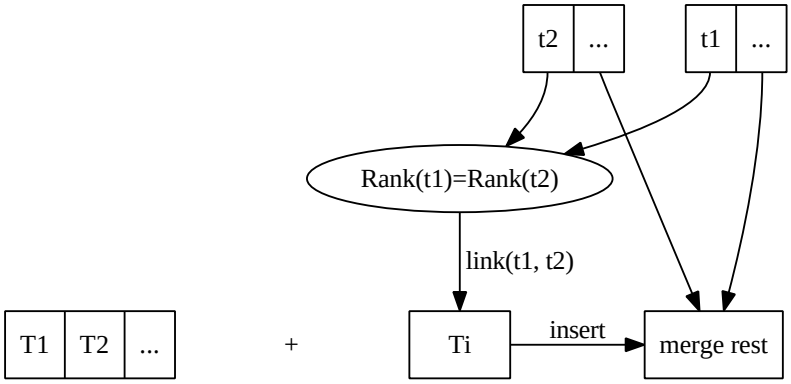
To analysis the performance of merge, suppose there are m_1 trees in H_1, and m_2 trees in H_2. There are at most $m_1 + m_2$ trees in the merged result. If there are no two trees have the same rank, the merge operation is bound to $O(m_1 + m_2)$. While if there need linking for the trees with same rank, $insertT$ 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)
```



(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 7: Merge two heaps.

```
  | rank t1 > rank t2 = t2:(merge ts1 ts2')
  | otherwise = insertTree (merge ts1' ts2') (link t1 t2)
```

Merge algorithm can also be described in imperative way as shown in algorithm 4.

Algorithm 4 imperative merge two binomial heaps

```
1:  function MERGE(H_1, H_2)
2:      if H_1 = φ then
3:          return H_2
4:      if H_2 = φ then
5:          return H_1
6:      H ← φ
7:      while H_1 ≠ φ ∧ H_2 ≠ φ do
8:          T ← φ
9:          if RANK(H_1) < RANK(H_2) then
10:             (T, H_1) ← EXTRACT-HEAD(H_1)
11:         else if RANK(H_2) < RANK(H_1) then
12:             (T, H_2) ← EXTRACT-HEAD(H_2)
13:         else                                          ▷ Equal rank
14:             (T_1, H_1) ← EXTRACT-HEAD(H_1)
15:             (T_2, H_2) ← EXTRACT-HEAD(H_2)
16:             T ← LINK(T_1, T_2)
17:         APPEND-TREE(H, T)
18:     if H_1 ≠ φ then
19:         APPEND-TREES(H, H_1)
20:     if H_2 ≠ φ 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 ≠ φ∧ RANK(T) = RANK(LAST(H)) then
3:         LAST(H) ← 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 ∈ H_2 do
3:         H_1 ← 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 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.

2.2.4 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 8 illustrates this idea.

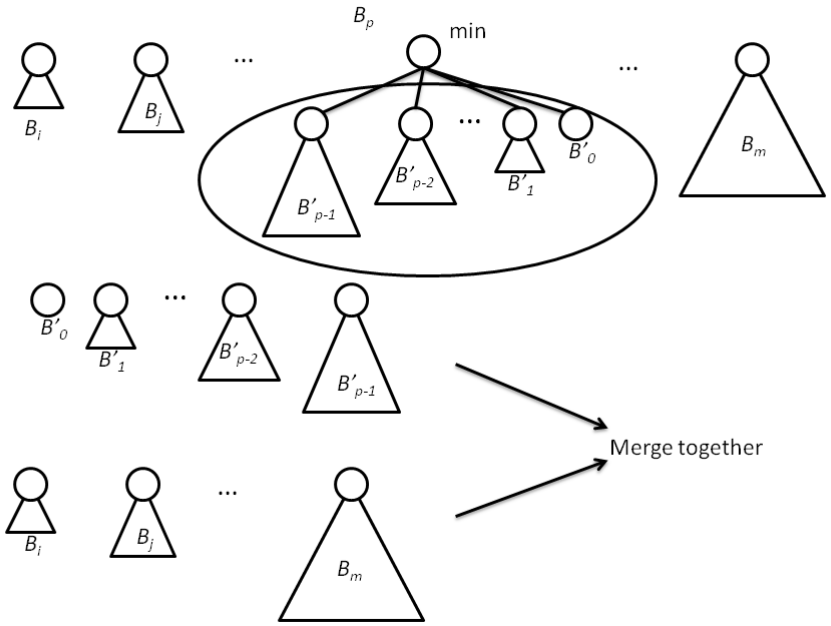


Figure 8: Pop the minimum element from a binomial heap.

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.

$$extractMin(H) = \begin{cases} (T, \phi) & : \text{H is a singleton as } \{T\} \\ (T_1, H') & : Root(T_1) < Root(T') \\ (T', \{T_1\} \cup H'') & : otherwise \end{cases} \quad (5)$$

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 ← HEAD(H)
3:     min ← ∞
4:     while T ≠ φ do
5:         if KEY(T)< min then
6:             min ← KEY(T)
7:         T ← 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 $extractMin$.

$$delteMin(H) = merge(reverse(Children(T)), H') \tag{6}$$

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) ← EXTRACT-MIN-TREE(H)
3:     H ← 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.

$$sort(xs) = heapSort(fromList(xs)) \tag{7}$$

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

$$heapSort(H) = \begin{cases} \phi & : \quad H = \phi \\ \{findMin(H)\} \cup heapSort(deleteMin(H)) & : \quad otherwise \end{cases} \tag{8}$$

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 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.

2.2.5 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.

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]

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.

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.

3.2.1 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.

$$insert(H, x) = merge(H, singleton(x)) \tag{9}$$

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 ← SINGLETON(k)                          ▷ Wrap x to a node
3:     append x to root list of H
4:     if T_min(H) = NIL ∨ k < KEY(T_min(H)) then
5:         T_min(H) ← x
6:     N(H) ← 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 5

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

3.2.2 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.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ FibHeap(s_1 + s_2, T_{1min}, \{T_{2min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & root(T_{1min}) < root(T_{2min}) \\ FibHeap(s_1 + s_2, T_{2min}, \{T_{1min}\} \cup \mathbb{T}_1 \cup \mathbb{T}_2) & : & otherwise \end{cases} \tag{10}$$

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.

```
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(H1, H2)
2:     H ← Φ
3:     ROOT(H) ← CONCAT(ROOT(H1), ROOT(H2))
4:     if KEY(Tmin(H1)) < KEY(Tmin(H2)) then
5:         Tmin(H) ← Tmin(H1)
6:     else
7:         Tmin(H) ← Tmin(H2)
       N(H) = N(H1) + N(H2)
8:     release H1 and H2
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.

```
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.

$$top(H) = root(T_{min}) \tag{11}$$

Exercise 6

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

3.2.3 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.

$$consolidate(L) = fold(meld, \phi, L) \tag{12}$$

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 $meld()$

Table 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

is defined as below.

$$meld(L, x) = \begin{cases} \{x\} & : \quad L = \phi \\ meld(L', x + x_1) & : \quad x = x_1 \\ \{x\} \cup L & : \quad x < x_1 \\ \{x_1\} \cup meld(L', x) & : \quad otherwise \end{cases} \tag{13}$$

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 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.

$$meld(L, x) = \begin{cases} \{x\} & : & L = \phi \\ meld(L', link(x, x_1)) & : & rank(x) = rank(x_1) \\ \{x\} \cup L & : & rank(x) < rank(x_1) \\ \{x_1\} \cup meld(L', x) & : & otherwise \end{cases} \tag{14}$$

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 9 and 10 show the steps of consolidation when processing a Fibonacci Heap contains different ranks of trees. Comparing with table 1 reveals the similarity.

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.

$$deleteMin(H) = \begin{cases} \phi & : & \mathbb{T} = \phi \wedge children(T_{min}) = \phi \\ FibHeap(s - 1, T'_{min}, \mathbb{T}') & : & otherwise \end{cases} \tag{15}$$

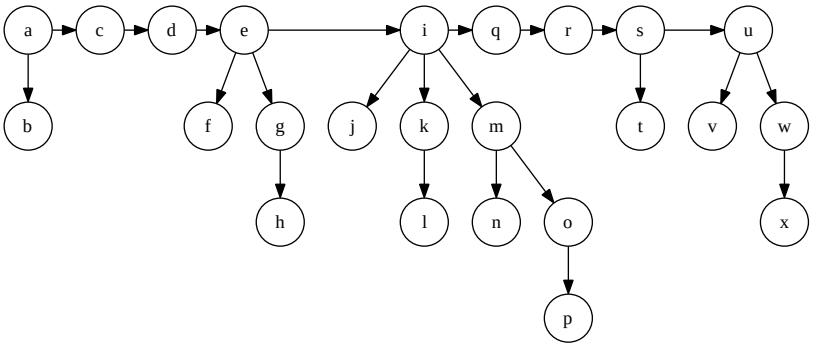
Where

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

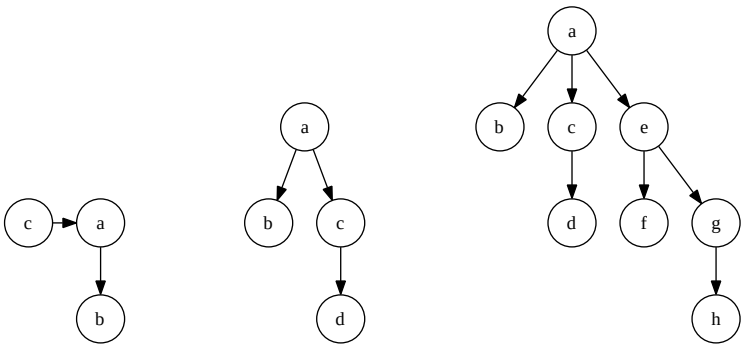
Translate to Haskell yields the below program.

```
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.

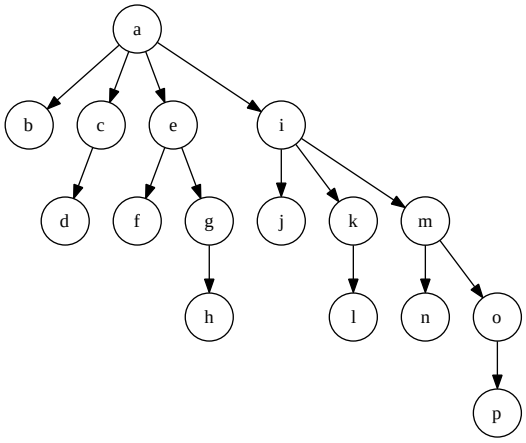


(a) Before consolidation

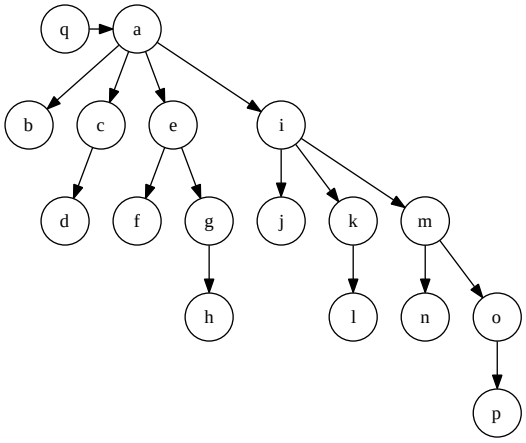


(b) Step 1, 2 (c) Step 3, 'd' is firstly linked to 'c', then repeatedly linked to 'a'. (d) Step 4

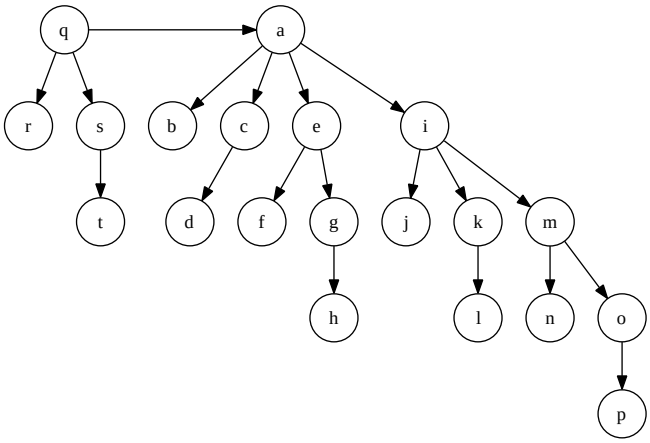
Figure 9: Steps of consolidation



(a) Step 5



(b) Step 6



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

Figure 10: Steps of consolidation

```
1: function DELETE-MIN(H)
2:     x ← T_min(H)
3:     if x ≠ NIL then
4:         for each y ∈ CHILDREN(x) do
5:             append y to root list of H
6:             PARENT(y) ← NIL
7:         remove x from root list of H
8:         N(H) ← N(H) - 1
9:         CONSOLIDATE(H)
10:    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.

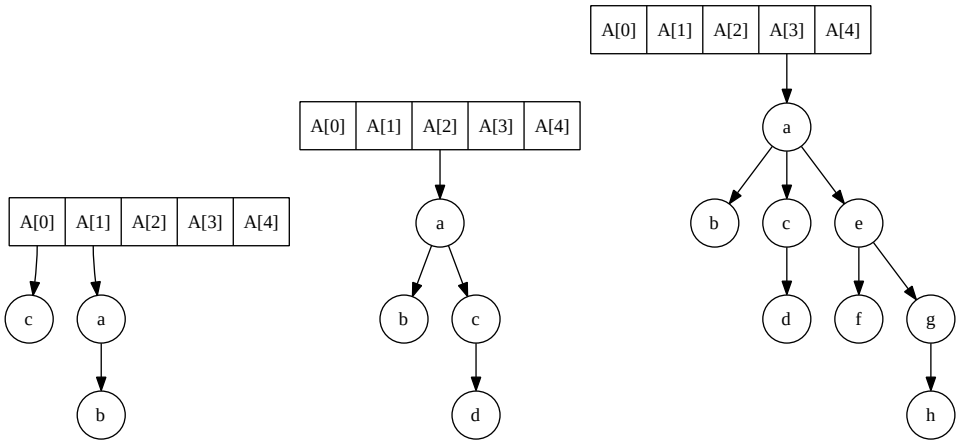
```
1: function CONSOLIDATE(H)
2:     D ← MAX-DEGREE(N(H))
3:     for i ← 0 to D do
4:         A[i] ← NIL
5:     for each x ∈ root list of H do
6:         remove x from root list of H
7:         d ← DEGREE(x)
8:         while A[d] ≠ NIL do
9:             y ← A[d]
10:            x ← LINK(x, y)
11:            A[d] ← NIL
12:            d ← d + 1
13:        A[d] ← x
14:    T_min(H) ← NIL                    ▷ root list is NIL at the time
15:    for i ← 0 to D do
16:        if A[i] ≠ NIL then
17:            append A[i] to root list of H.
18:            if T_min = NIL∨ KEY(A[i]) < KEY(T_min(H)) then
19:                T_min(H) ← 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 9 to the above algorithm, Figure 11, 12 and 13 show the result trees stored in auxiliary array A in every steps.

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

```
void consolidate(struct FibHeap* h){
  if(!h->roots)
    return;
```



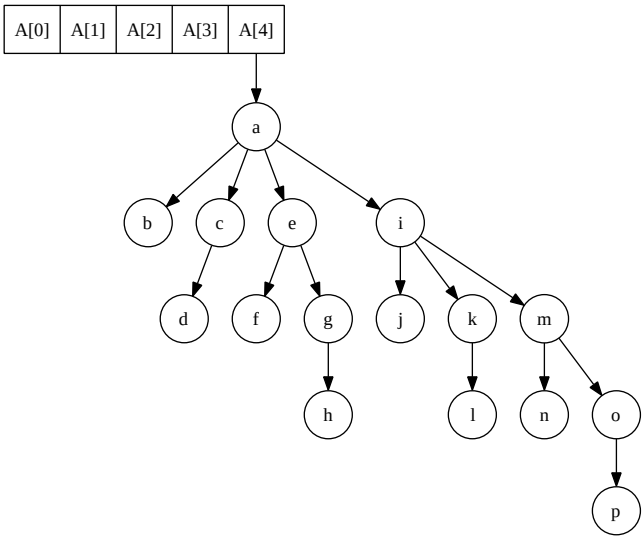
(a) Step 1, 2

(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.

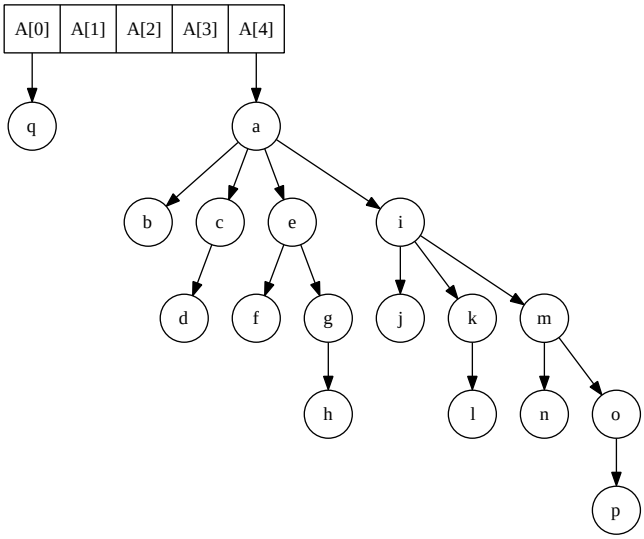
(c) Step 4

Figure 11: Steps of consolidation

```
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]){
    h->roots = append(h->roots, a[i]);
    if(h->minTr == NULL || a[i]->key < h->minTr->key)
```

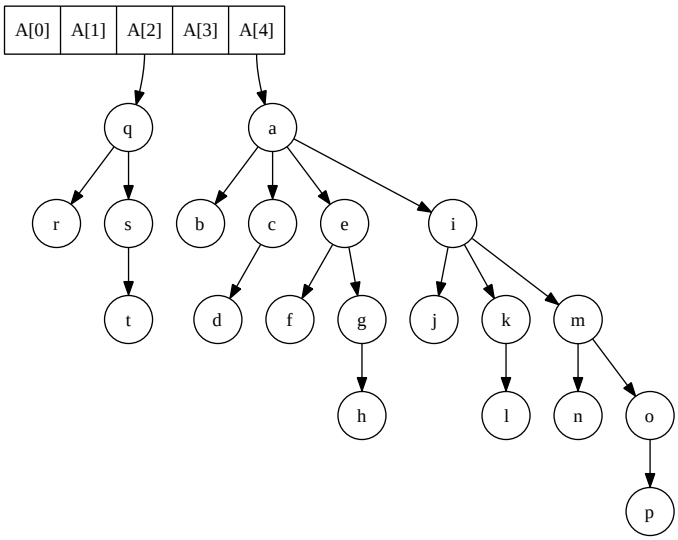


(a) Step 5



(b) Step 6

Figure 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 13: Steps of consolidation

```
            h->minTr = a[i];
        }
    free(a);
}
```

Exercise 7

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

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.

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

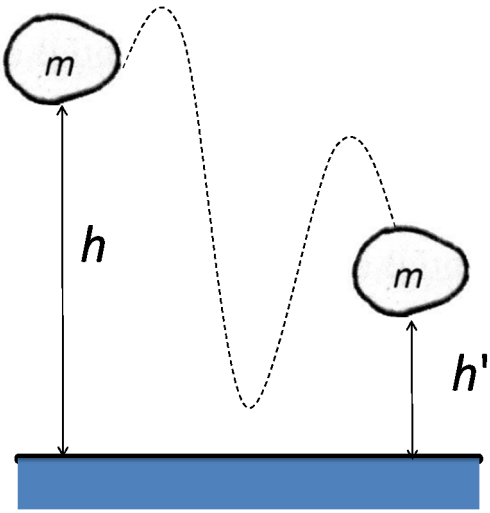


Figure 14: Gravity potential energy.

Figure 14 illustrated this concept.

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

$$\Phi(H) = t(H) \tag{16}$$

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.

$$\begin{aligned} T &= T_{consolidation} + \Phi(H') - \Phi(H) \\ &= O(D(N) + t(H) - 1) + (D(N) + 1) - t(H) \\ &= O(D(N)) \end{aligned} \tag{17}$$

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.

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 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) ← k
3:     p ← PARENT(x)
4:     if p ≠ NIL ∧ k < KEY(p) then
5:         CUT(H, x)
6:         CASCADING-CUT(H, p)
7:     if k < KEY(T_min(H)) then
```

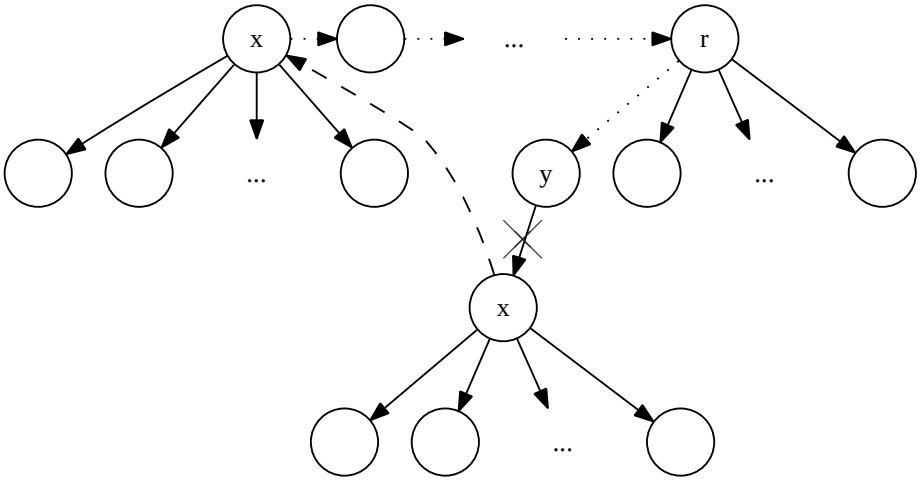


Figure 15: $x < y$, cut tree x from its parent, and add x to root list.

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 8

Prove that DECREASE-KEY algorithm is amortized $O(1)$ time.

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

$$|x| \geq F_{k+2} \tag{18}$$

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

$$F_{k+2} = 1 + \sum_{i=}^{k} F_i \tag{19}$$

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.

$$N \geq |x| \geq F_k + 2 \tag{20}$$

□

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.

$$MaxDegree(N) = 1 + \lfloor \log_\Phi N \rfloor \tag{21}$$

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

```
1: function MAX-DEGREE(N)
2:     F_0 ← 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;
}
```

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] [3]. 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.

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.

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.

4.2.1 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.

$$merge(H_1, H_2) = \begin{cases} H_1 & : & H_2 = \phi \\ H_2 & : & H_1 = \phi \\ Node(x, \{H_2\} \cup Children(H_1)) & : & x < y \\ Node(y, \{H_1\} \cup Children(H_2)) & : & otherwise \end{cases} \quad (22)$$

[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

Where

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

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(H1, H2)
2:     if H1 = NIL then
3:         return H2
4:     if H2 = NIL then
5:         return H1
6:     if KEY(H2) < KEY(H1) then
7:         EXCHANGE(H1 ↔ H2)
8:     Insert H2 in front of CHILDREN(H1)
9:     PARENT(H2) ← H1
10:    return H1
```

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.

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

With merge defined, insertion can be realized as same as Fibonacci Heap in Equation 9. Definitely it's $O(1)$ time operation. As the minimum element is always stored in root, finding it is trivial.

$$top(H) = Root(H) \tag{23}$$

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

Exercise 9

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

4.2.2 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) ← k
3:     if PARENT(x) ≠ NIL then
4:         Remove x from CHILDREN(PARENT(x))
           PARENT(x) ← 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

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?

4.2.3 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.

$$pop(H) = mergePairs(Children(H)) \tag{24}$$

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 16 and 17 illustrate the procedure of pair-merging.

The recursive pair-merging solution is quite similar to the bottom up merge sort[3]. 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 below.

$$mergePairs(A) = \begin{cases} \Phi & : & A = \Phi \\ T_1 & : & A = \{T_1\} \\ merge(merge(T_1, T_2), mergePairs(A')) & : & otherwise \end{cases} \tag{25}$$

where

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

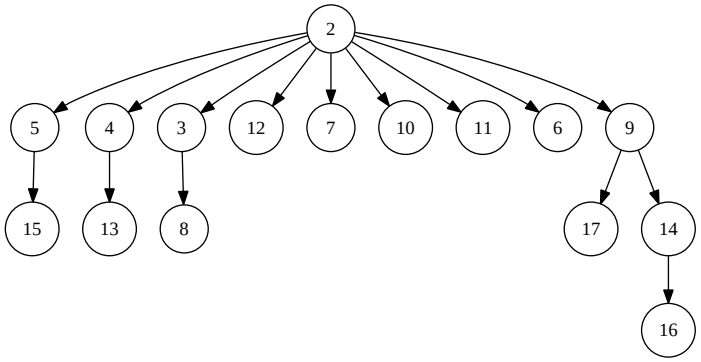
is the rest of the children without the first two trees.

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

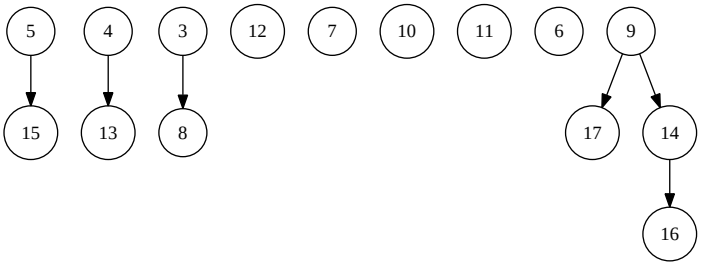
```
deleteMin :: (Ord a) => PHeap a -> PHeap a
deleteMin (Node _ hs) = mergePairs hs where
    mergePairs [] = E
    mergePairs [h] = h
    mergePairs (h1:h2:hs) = merge (merge h1 h2) (mergePairs hs)
```

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

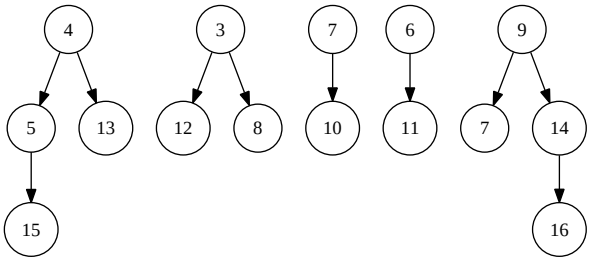
```
1: function POP(H)
2:     L ← NIL
3:     for every 2 trees T_x, T_y ∈ CHILDREN(H) from left to right do
4:         Extract x, and y from CHILDREN(H)
5:         T ← MERGE(T_x, T_y)
6:         Insert T at the beginning of L
7:     H ← CHILDREN(H)                    ▷ H is either NIL or one tree.
8: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



(a) A pairing heap before pop.

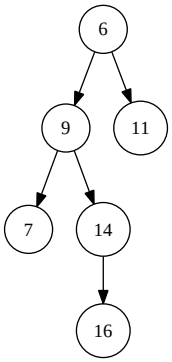


(b) After root element 2 being removed, there are 9 sub-trees left.

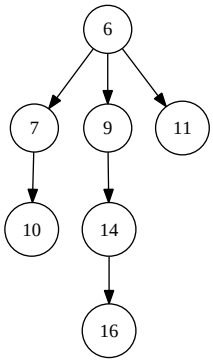


(c) Merge every two trees in pair, note that there are odd number trees, so the last one needn't merge.

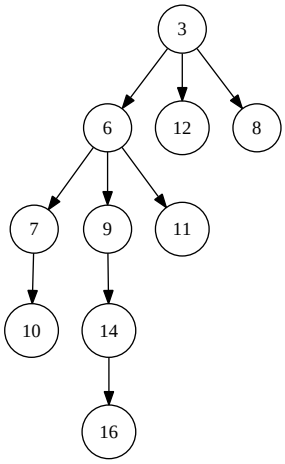
Figure 16: Remove the root element, and merge children in pairs.



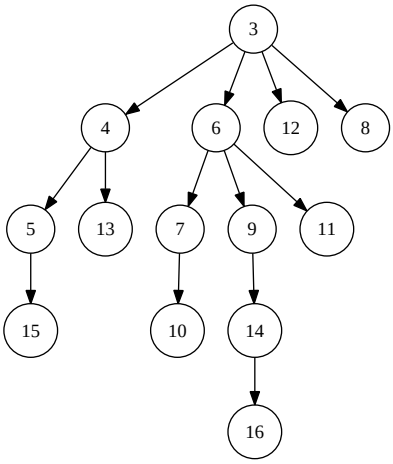
(a) Merge tree with 9, and tree with root 6.



(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 17: Steps of merge from right to left.

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.

```
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 11

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

4.2.4 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.

$$delete(H, x) = \begin{cases} pop(H) & : \quad x \quad \text{is root of} \quad H \\ merge(cut(H, x), pop(x)) & : \quad otherwise \end{cases} \tag{26}$$

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

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

Exercise 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.

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.

References

[1] K-ary tree, Wikipedia. http://en.wikipedia.org/wiki/K-ary_tree

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

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

[4] Wikipedia, "Pascal's triangle". http://en.wikipedia.org/wiki/Pascal's_triangle

[5] Hackage. "An alternate implementation of a priority queue based on a Fibonacci heap.", http://hackage.haskell.org/packages/archive/pqueue-mtl/1.0.7/doc/html/src/Data-Queue-FibQueue.html

[6] Chris Okasaki. "Fibonacci Heaps." http://darcs.haskell.org/nofib/gc/fibheaps/orig

[7] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. "The Pairing Heap: A New Form of Self-Adjusting Heap" Algorithmica (1986) 1: 111-129.