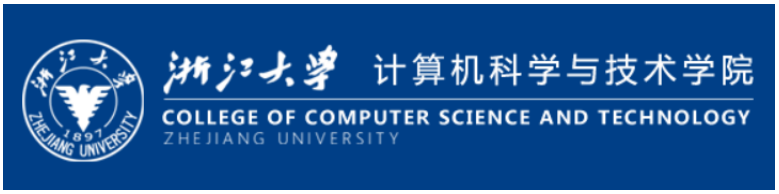

Zhejiang University

College of Computer Science and Technology

Skip List

Project7 Report by Group1

Author:
Wu Yihang
Sun Xinjie
Guo Jiahao

Supervisor:
Mao Yuchen

Abstract:

In this project, we implement a probabilisitc data structure that is built upon the general idea of a linked list named skip list. We not only ensure the correctness of our implementation for searching, inserting and deleting, but analyze the the time complexity, which is $O(\log n)$, of these operations theoretically based on probability theory and pratically based on experiments on inputs of different size.

An Assignment submitted for the ZJU:

21120491 Advanced Data Structure and Algorithm Analysis

May 31, 2022

1 Introduction of the Project

In this project, we need to implement a probabilisitc data structure that is built upon the general idea of a linked list named skip list. We need to implement three basic operations: searching, inserting, and deleting. Except that, we need to analyze these operations not only theoretically but also pratically. We need to measure the practical performance of skip list on inputs of different size.

2 Introduction to the Skip List

2.1 Structure Property

Fibonacci heap is a data structure for priority queue operations, which is is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent.

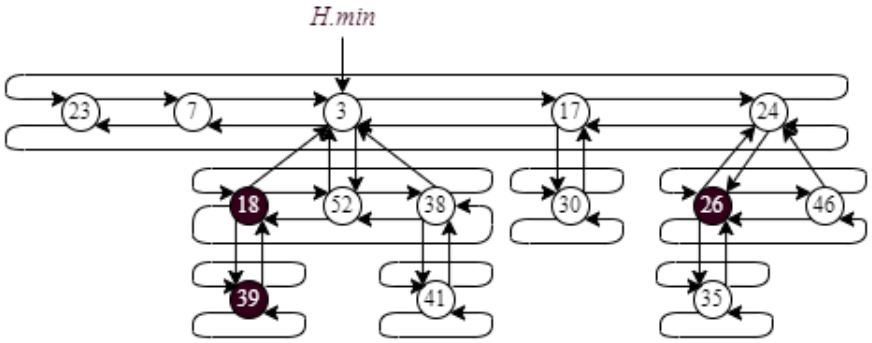

Figure 1: An example of a Fibonacci heap

As Figure 1 shows, we access a given Fibonacci heap H by a pointer **H.min** to the root of a tree containing the minimum key. When a Fibonacci heap H is empty, H.min is NULL. The roots of all the trees in a Fibonacci heap are linked together using their **left** and **right** pointers into a circular, doubly linked list called **root list**.

We also see that each node x contains a pointer **x.p** to its parent and a pointer **x.child** to any one of its children. Each child y of x are linked together in a circular, doubly linked list, which we call the **child list of x**, using pointers **y.left** and **y.right** that point to ys left and right siblings. If node y is an only child, then y.left=y.right=y. Siblings may appear in a child list in any order.

Except that, other property we need for the node are:

1. **x.degree**: store the number of children in the child list of node x.
2. **x.mark**: indicates whether node x has lost a child since the last time x was made the child of another node.

We also need **H.n** as a heap attribute, which represent the number of nodes currently in H.

2.2 Implementation of Operations

2.2.1 Insertion

Insertion for Fibonacci heap is a lazy operation. The following procedure inserts node x into Fibonacci heap H:

Algorithm 1: Insertion for Fibonacci heap

```
Input: Fibonacci heap H, node x
x.degree = 0;
x.p = x.child = NULL;
x.mark = false;
if H.min == NULL then
    create a root list for H containing just x;
    H.min = x;
else
    insert x into H's root list;
    if x.key < H.min.key then
        H.min = x;
    end
end
H.n = H.n + 1
```

First we initialize some attributes of node x. Next, we test if Fibonacci heap H is empty. If it is, we make x be the only node in Hs root list and set $H.min$ to point to x. Otherwise, we insert x into Hs root list and update $H.min$ if necessary. Finally, we increment $H.n$.

2.2.2 Make heap

For the making heap operation of Fibonacci heap, we just need to insert n keys continuously.

2.2.3 FindMin

Because we maintain a pointer to the minimum node of the heap, so we only need to return $H.min.data$.

2.2.4 DeleteMin

DeleteMin is much more complicated than insertion. It works by first making a root out of each of the minimum nodes children and removing the minimum node from the root list. It then consolidate the root list by linking roots of equal degree until at most one root remains of each degree. The pseudocode of Algorithm 2 extracts the minimum node.

The procedure consolidate uses an auxiliary array A, whose size can be limited by the bounding the maximum degree

$$D(n) \leq \lfloor \log_{\phi} n \rfloor, \quad \phi = (1+\sqrt{5})/2$$

, to keep track of roots according to their degrees. If $A[i] = y$, then y is currently a root with $y.degree = i$.

In detail, let's see the pseudocode of Algorithm 3, the consolidate procedure works as follows. We first allocate array A, then we link roots together, w may be linked to some other node and no longer be a root. Nevertheless, w is always in a tree rooted at some node x, which may or may not be w itself. Because we want at most one root with each degree, we look in the array A to see whether it contains a root y with the same degree as x. If it does, then we link the roots x and y but guaranteeing that x remains a root after linking. That is, we link y to x after first exchanging the pointers to the two roots if ys key is smaller than xs key. After we link y to x, the degree of x has increased by 1, and so we continue this process, linking x and another root whose degree equals xs new degree, until no other root that we have processed has the same degree as x. We then set the appropriate entry of A to point to x, so that as we process roots later on, we have recorded that x is the unique root of its degree that we have already processed. When this for loop terminates, at most one root of each degree will remain, and the array A will point to each remaining root.

Algorithm 2: DeleteMin for Fibonacci heap

```
   Input: Fibonacci heap H
 1 z = H.min;
 2 if z != NULL then
 3 |   for each child x of z do
 4 |   |   add x to the root list of H;
 5 |   |   x.p = NULL
 6 |   end
 7 |   remove z from the root list of H;
 8 |   if z == z.right then
 9 |   |   H.min = NULL;
10 |   else H.min = z.right
11 |   |   consolidate(H);
12 |   end
13 |   H.n = H.n - 1
14 end
```

2.2.5 Merge

We don't need this procedure in this project, so I just describe this procedure without pseudocode. To merge two Fibonacci heaps H_1 and H_2, we first concatenate the root lists of H_1 and H_2 into a new root list H. Then we set the minimum node of H and new $H.n$.

2.2.6 DecreseKey

DecreseKey operation is crucial for the optimization for Dijkstra's algorithm because it has only $O(1)$ amortized time. To implement the operation, we first judge if the min-heap order has not been violated, if it is, we don't need to adjust the position of the node. If min-heap order has been violated, many changes may occur. We start by cutting procedure, which cuts the link between x and its parent y, making x a root.

Algorithm 3: Consolidate for Fibonacci heap

```
   Input: Fibonacci heap H, node x
 1 let A be a new array;
 2 for i = 0 to size of A do
 3 |   A[i] = NULL;
 4 end
 5 for each node w in the root list of H do
 6 |   x = w;
 7 |   d = x.degree;
 8 |   while A[d] != NULL do
 9 |   |   y = A[d];
10 |   |   if x.key > y.key then
11 |   |   |   exchange x with y;
12 |   |   end
13 |   |   remove y from the root list of H;
14 |   |   make y a child of x, incrementing x.degree;
15 |   |   y.mark = false;
16 |   |   A[d] = NULL;
17 |   |   d = d + 1;
18 |   end
19 |   A[d] = x;
20 end
21 H.min = NULL;
22 for i = 0 to size of A do
23 |   if A[d] != NULL then
24 |   |   if H.min == NULL then
25 |   |   |   create a root list for H containing just A[i];
26 |   |   |   H.min = A[i];
27 |   |   else
28 |   |   |   insert A[i] into H's root list;
29 |   |   |   if A[i].key < H.min.key then
30 |   |   |   |   H.min = A[i];
31 |   |   |   end
32 |   |   end
33 |   end
34 end
```

We use the mark attributes to obtain the desired time bounds. As soon as the second child has been lost, we cut x from its parent, making it a new root, and sometimes we need cascading-cut operation. Once all the cascading cuts have occurred, the procedure can finish up by updating *H.min* if necessary. The only node whose key changed was the node x whose key decreased. Thus, the new minimum node is either the original minimum node or node x.

Algorithm 4: DecreseKey for Fibonacci heap

```
Input: Fibonacci heap H, node x, new key value k
if k > x.key then
    error"new key is greater than current key";
end
x.key = k;
y = x.p;
if y!= NULL and x.key < y.key then
    remove x from the child list of y, decrementing y.degree;
    add x to the root list of H;
    x.p = NULL;
    x.mark = false;
    cascading_cut(H, y);
end
if x.key < H.min.key then
    H.min = x;
end
```

Algorithm 5: Cascade cut for Fibonacci heap

```
Input: Fibonacci heap H, node y
z = y.p;
if z!= NULL then
    if y.mark == false then
        y.mark = true;
    else
        remove y from the child list of z, decrementing z.degree;
        add y to the root list of H;
        y.p = NULL;
        y.mark = false;
        cascading_cut(H, z);
    end
end
```

2.2.7 Delete

For deletion, we just need to use decrease key function to decrease the data stored in the node we want to delete to a small value that normal node won't store.

3 Theoretical Analysis

3.1 Heap Operations

In this section, we define n as the number of elements in priority queue. And the result in the table represent the worst time, **except the column for Fibonacci heap, which represent amortized time**.

Operation	**Binary heap**	**Leftist heap**	**Binomial heap**	**Fibonacci heap**
Make heap	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Find min	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Delete min	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Merge	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Decrese key	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$

Table 1: Running times of all operations of different heaps

3.2 optimization for Dijkstra's Algorithm

We have learned that theoretically, the total running time of Dijkstra's algorithm without priority queue is $O(|E|+|V|^2)$. So if the graph is dense, with $|E| = \Theta(|V|^2)$, the algorithm without priority queue is good enough.

But in practice, also in this project, the grapg is always sparse, which means $|E| = \Theta(|V|)$, the algorithm is too slow. So we consider the heap operation DeleteMin, whose running time is $O(n\log n)$ in all the heaps we use in the project, to find and delete the minimum vertex. And we use DecreseKey, or in practice insertion, whose running time is $O(n\log n)$ for binary heap, leftist heap, binomial heap, and $O(1)$ for Fibonacci heap.

The improtant observation is that the running time of Dijkstra algorithm is dominated by $|E|$ DecreseKey/Insertion operation and $|V|$ DeleteMin operations. So theoretically, for binary heap, leftist heap and binomial heap, the time bound for the optimized Dijkstra algorithm is $O(|E|\log|V| + |V|\log|V|) = O(|E|\log|V|)$, and for Fibonacci heap, the time bound will be decreased to $O(|E| + |V|\log|V|)$.

4 Experiment and Result

5 Discussion and Conclusion