International Journal of Research in Engineering, Science and Management
Volume-1, Issue-11, November-2018
www.ijresm.com | ISSN (Online): 2581-5792

Fibonacci Heap and its Applications

K. N. Hemanth Rao[1], Raghavendra[2], Pratyush Singh[3], Sanket V. Salankimatt[4]
[1,2,3,4]*Student, Department of CSE, R. V. College of Engineering, Bengaluru, India*

***Abstract*: A Fibonacci heap is a specific implementation of the heap data structure that makes use of Fibonacci numbers. Fibonacci heaps are used to implement the priority queue element in Dijkstra's algorithm, giving the algorithm a very efficient running time. Fibonacci heaps have a faster amortized running time than other heap types. Fibonacci heaps are similar to binomial heaps but Fibonacci heaps have a less rigid structure. Binomial heaps merge heaps immediately but Fibonacci heaps wait to merge until the extract-min function is called. While Fibonacci heaps have very good theoretical complexities, in practice, other heap types such as pairing heaps are faster. This is because even in the simplest implementation, Fibonacci heaps require four pointers for each node, other heaps need two or three [1].**

***Keywords*: Nodes Fibonacci heaps and image**

1. Objectives

Fibonacci heaps but binary heaps are used in the priority queues. Priority queues are widely used in the real systems. One known example is process scheduling in the kernel. The highest priority process is taken first.

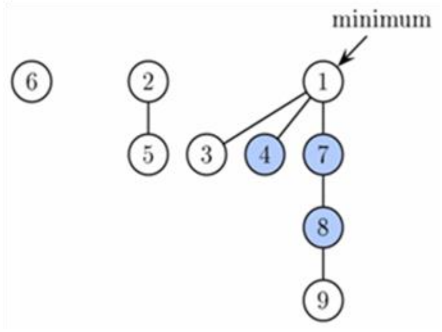


Fig. 1. Fibonacci heaps

2. Methodology

Here is how Fibonacci heaps implement the basic functionalities of heaps and the time complexity of each operation. The children of each node are also related using a linked list. For each node, the linked list maintains the number of children a node has and whether the node is marked. The linked list also maintains a pointer to the root containing the minimum key. A node is marked to indicate if any of its children were removed. This is important so the heap can keep track of how far removed its shape is becoming from a binomial heap. If a Fibonacci heap is too different from a binomial heap, it loses many of the efficient time operations that their binomial nature gives it. These operations are described in terms of a min Fibonacci heap, but they could easily be adapted to be max Fibonacci heap operations [2].

A. *Find Minimum*

The linked list has pointers and keeps track of the minimum node, so finding the minimum is simple and can be done in constant time [2].

B. *Merge*

In Fibonacci heaps, merging is accomplished by simply concatenating two lists containing the tree roots. Compare the roots of the two heaps to be merged, and whichever is smaller becomes the root of the new combined heap. The other tree is added as a sub tree to this root. This can be done in constant time [2].

C. *Extract Minimum*

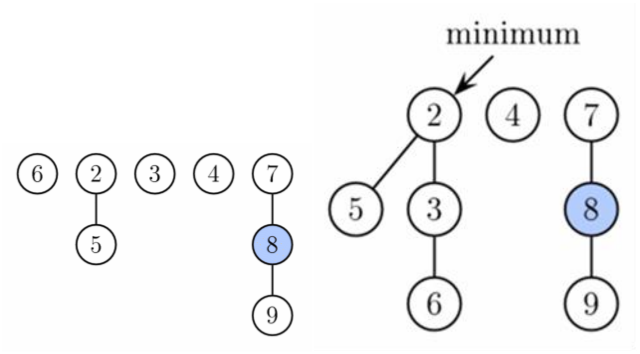


Fig. 2. Extract minimum

Extract-min is one of the most important operations regarding Fibonacci heaps. Much of a Fibonacci heap speed advantage comes from the fact that it delays consolidating heaps after operations until extract-min is called. Binomial heaps, on the other hand, consolidate immediately. Consolidation occurs when heap properties are violated, for example, if two heaps have the same order, the heaps must be adjusted to prevent this. [2] Deleting the minimum element is done in three steps. The node is removed from the root list and the node's children are added to the root list. Next, the minimum element is updated if needed. Finally, consolidate the trees so that there are no repeated orders. If any consolidation occurred, make sure to update the minimum element if needed. Delaying consolidation saves times. The two images below show the extract-min function on the Fibonacci heap shown in the introduction [2].

D. *Insert*

Insertion to a Fibonacci heap is similar to the insert operation of a binomial heap. A heap of one element is created and the two heaps are merged with the merge function. The minimum element pointer is updated if necessary. The total number of nodes in the tree increases by one. [2]

E. *Remove*

To delete an element, decrease the key using decrease key to negative infinity, and then call extract-min. When the node has a value of negative infinity, since the heap is a min heap, it will become the root of the tree. Extract-min will remove the top element, so doing this deletes the node in question. [2]

F. *Decrease Key*

There are two situations that can arise when decreasing the key the change will cause a heap violation or it will not.

- If the heap properties aren't violated, simply decrease xx.
- If a violation does occur, remove the node its parent. If the parent is not a root, mark it. If it has been marked already, it is removed as well and its parent is marked, and so on. Continue this process up the tree until either the root or an unmarked node is reached. Next, set the minimum pointer to the decreased value if it is the new minimum. [2].

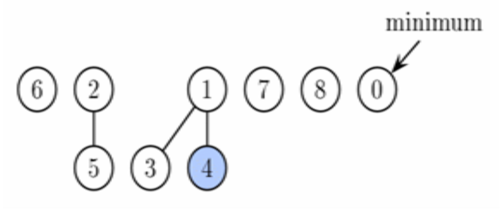


Fig. 3. Minimum decrease key

The decrease key function marks a node when its child is removed. This allows it to track some history about each node. Essentially the marking tracks if:

- The node has had no children removed (unmarked)
- The node has had a single child removed (marked)
- The node is about to have a second child removed (removing a child of a marked node.

3. Implementation

```
Make-Fibonacci-Heap()
n[H] := 0
min[H] := NIL
return H

Fibonacci-Heap-Minimum(H)
return min[H]

Fibonacci-Heap-Link(H,y,x)
remove y from the root list of H
make y a child of x
degree[x] := degree[x] + 1
mark[y] := FALSE
```

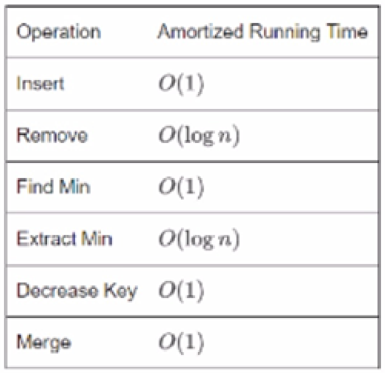
```
CONSOLIDATE(H)
for i:=0 to D(n[H])
    Do A[i] := NIL
for each node w in the root list of H
   do x:= w
      d:= degree[x]
      while A[d] <> NIL
         do y:=A[d]
            if key[x]>key[y]
              then exchange x<->y
            Fibonacci-Heap-Link(H, y, x)
            A[d]:=NIL
           d:=d+1
      A[d]:=x
min[H]:=NIL
for i:=0 to D(n[H])
   do if A[i]<> NIL
         then add A[i] to the root list of H
            if min[H] = NIL or key[A[i]]<key[min[H]]
              then min[H]:= A[i]

Fibonacci-Heap-Union(H1,H2)
H := Make-Fibonacci-Heap()
min[H] := min[H1]
Concatenate the root list of H2 with the root list of H
if (min[H1] = NIL) or (min[H2] <> NIL and min[H2] <
min[H1])
  then min[H] := min[H2]
n[H] := n[H1] + n[H2]
free the objects H1 and H2
return H


Fibonacci-Heap-Insert(H,x)
degree[x] := 0
p[x] := NIL
child[x] := NIL
left[x] := x
right[x] := x
mark[x] := FALSE
concatenate the root list containing x with root list H
if min[H] = NIL or key[x]<key[min[H]]
       then min[H] := x
n[H]:= n[H]+1

Fibonacci-Heap-Extract-Min(H)
z:= min[H]
if x <> NIL
       then for each child x of z
            do add x to the root list of H
               p[x]:= NIL
            remove z from the root list of H
            if z = right[z]
               then min[H]:=NIL
               else min[H]:=right[z]
                   CONSOLIDATE(H)
            n[H] := n[H]-1
```

International Journal of Research in Engineering, Science and Management
Volume-1, Issue-11, November-2018
www.ijresm.com | ISSN (Online): 2581-5792

```
return z

Fibonacci-Heap-Decrease-Key(H,x,k)
if k > key[x]
  then error "new key is greater than current key"
key[x] := k
y := p[x]
if y <> NIL and key[x]<key[y]
  then CUT(H, x, y)
       CASCADING-CUT(H,y)
if key[x]<key[min[H]]
  then min[H] := x

CUT(H,x,y)
Remove x from the child list of y, decrementing degree[y]
Add x to the root list of H
p[x]:= NIL
mark[x]:= FALSE

CASCADING-CUT(H,y)
z:= p[y]
if z <> NIL
  then if mark[y] = FALSE
       then mark[y]:= TRUE
       else CUT(H, y, z)
            CASCADING-CUT(H, z)

Fibonacci-Heap-Delete(H,x)
Fibonacci-Heap-Decrease-Key(H,x,-infinity)
Fibonacci-Heap-Extract-Min(H)        [3]
```

4. Conclusion and summary

Table 1
Running times of Fibonacci heaps



Operation	Amortized Running Time
Insert	$O(1)$
Remove	$O(\log n)$
Find Min	$O(1)$
Extract Min	$O(\log n)$
Decrease Key	$O(1)$
Merge	$O(1)$

References

[1] Stergiopoulos S. Algorithm for Fibonacci Heap Operations. Retrieved June 7, 2016.

[2] Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2001). Introduction to Algorithms (2nd edition) (pp. chapter 20). The MIT Press.

[3] Fredman, M., Sedgewick, R., Sleator, D., & Tarjan, R. The Pairing Heap: A New Form of Self-Adjusting Heap. Retrieved June 7, 2016.