Fibonacci Heaps

Saem Hasan
1705027
Samira
1705028

Department of Computer Science & Engineering

Bangladesh University of Engineering and Technology

July 27, 2021

Contents

1 Introduction **2**

2 Binary Heap **3**
2.1 Binary Heap: Properties 3
2.2 Building Binary Heap 4
2.3 Merge Binary Heaps 4

3 Mergeable Heaps **6**

4 Binomial Heap **7**
4.1 Binomial Tree 7

5 Fibonacci Heap **8**
5.1 Running Time Comparison 8
5.2 Fibonacci Heap: Structure 9
5.3 Heap Representation 9
5.4 Node Representation 10
5.5 Basic Operations 10
5.5.1 Insert 10
5.5.2 Union 11
5.5.3 Decrease key 12
5.5.4 Extract Min 14
5.5.5 Delete 17
5.5.6 Find min 17
5.6 Maximally Damaged tree 17
5.7 Amortized Running time 20

6 Conclusion **21**

Chapter 1

Introduction

Fibonacci heap is a data structure which is applied on a set of items. Fibonacci heap is similar to Binomial heap. Fibonacci heap is used where merging heap is costly. In other heap , merging takes O(n) time where fibonacci heap takes O(1). Other than that it has some other operation like insertion, Decrease key, Union, Delete, Find minimum ,Extract min. Fibonacci heap has a less rigid structure than other heaps.It lazily consolidates until the next Extract-min operation. Fibonacci is faster for network optimization problem.

Chapter 2

Binary Heap

A binary heap is a heap data structure that takes the form of a binary tree. Binary heaps are a common way of implementing priority queues.

A binary heap is defined as a binary tree with two additional constraints:

- **Structural property:** A binary heap is an almost complete binary tree. It is complete on all levels except the last.
- **Order property:** A binary heap maintains a minimum/maximum heap order property. The minimum/maximum key is stored in the root. The key stored in each node is either less than or equal to or greater than or equal to the keys of the node's children.

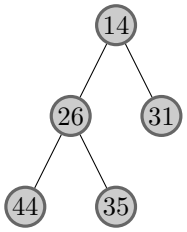


Figure 2.1: Binary Heap

2.1 Binary Heap: Properties

- Binary heap stores maximum/minimum key in the root.
- Binary heap maintains a pointer to the root node.
- Each node maintains pointers to it's child node.
- Binary heap with n elements has floor(logn) height where n equals to the number of nodes of the heap.

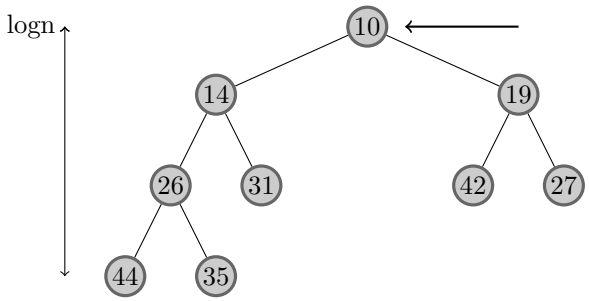


Figure 2.2: Binary Heap

2.2 Building Binary Heap

To build a binary heap with n keys, we can take two approaches:

1. We can call insert operation of binary heap for n keys. Each insert operation takes O(logn) time. Then to build a heap, we have to call insert operation n times.
   - Running time to build a heap by calling insert operation : O(nlogn)
2. We can construct a binary tree. For i=n to 1, we can repeatedly exchange the key in node i with it's smaller/larger child until the heap property is achieved. We can do this in linear time.
   - Running time to build a heap by constructing a binary tree : O(n)

2.3 Merge Binary Heaps

There is no easy solution to merge two binary heaps. We can take a approach. We can merge all keys of two heaps and then we can call build heap operation. Build heap operation takes O(n) time.

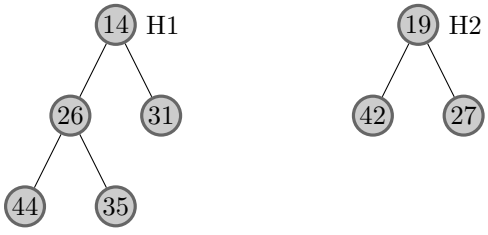


Figure 2.3: Merge Heaps

Figure 2.4: All elements of H1 & H2

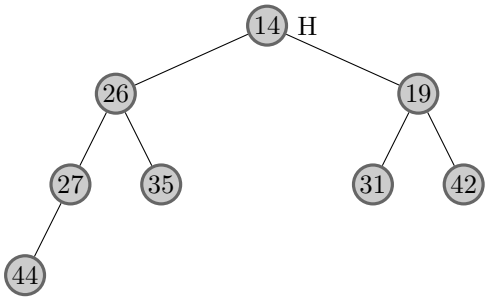


Figure 2.5: Merged Heap

- Running time to merge two heaps: O(n)

Chapter 3

Mergeable Heaps

We can merge two heaps by taking all keys of two heaps and calling build heap operation. We can do this in O(n) time. But we can merge two heaps faster than O(n). There are two well-known mergeable heap data structures:

1. Binomial Heap
2. **Fibonacci Heap**

In this report, we will mainly focus on Fibonacci Heap.

Chapter 4

Binomial Heap

Binomial Heap is a collection of binomial trees. There is at most one binomial tree of order in a binomial heap.

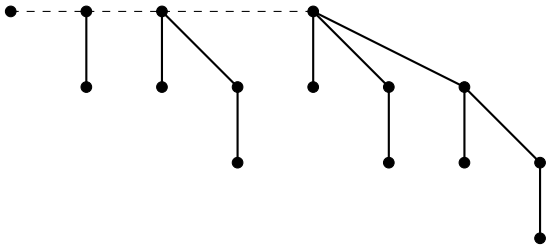


Figure 4.1: Binomial Heap

4.1 Binomial Tree

The Binomial Tree B_K is an ordered tree defined recursively.

- B_0 : Consists of a single node.
- B_K : Consists of two binomial trees B_{K-1} linked together.

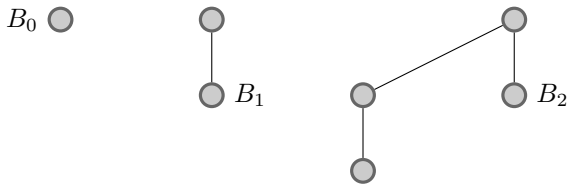


Figure 4.2: Binomial Trees

Chapter 5

Fibonacci Heap

Fibonacci heap is a data structure for priority queue operations, consisting of a collection of heap-ordered trees. It has a better amortized running time than many other priority queue data structures including the binary heap and binomial heap. Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.

Fibonacci Heap is very similar to binomial heap, but it has a less rigid structure. It lazily defers consolidation until the next Extract-min operation.

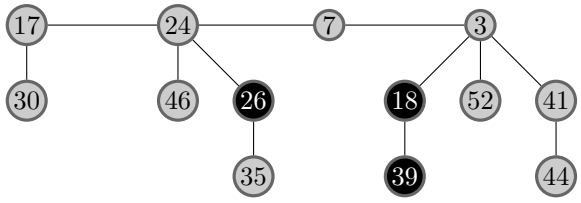


Figure 5.1: Fibonacci Heap

5.1 Running Time Comparison

For the Fibonacci heap, the find-minimum operation takes constant (O(1)) amortized time. The insert and decrease key operations also work in constant amortized time. Deleting an element (most often used in the special case of deleting the minimum element) works in O(log n) amortized time, where n is the size of the heap. The union operation takes constant time.

For the Binomial heap, all the operations take $O(\log n)$ time. So, Fibonacci heap is better than Binomial/Binary heap. If we use Fibonacci heaps for implementing priority queues, it will improve the asymptotic running time of important algorithms, such as Dijkstra's algorithm for computing the shortest path between two nodes in a graph, compared to the same algorithm using other slower priority queue data structures.

Operation	*Binary Heap*	*Binomial Heap*	*Fibonacci Heap*
Insert	O(logn)	O(logn)	O(1)
Decrease Key	O(logn)	O(logn)	O(1)
Extract Min	O(logn)	O(logn)	O(logn)
Delete	O(logn)	O(logn)	O(logn)
Find Min	O(1)	O(logn)	O(1)
Union	O(n)	O(logn)	O(1)

Table 5.1: Running Time Comparison

5.2 Fibonacci Heap: Structure

- Fibonacci heap has a set of heap-ordered trees. Each tree is maintaining minimum heap order property in Figure 5.2.
- Fibonacci heap has a set of marked nodes. If any node is marked, then it means that one child node of this node is cut.

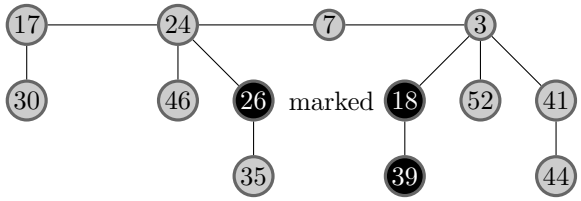


Figure 5.2: Fibonacci Heap : Structure

5.3 Heap Representation

- Fibonacci heap stores a pointer to the minimum node.
- Fibonacci heap's root list maintains a doubly circular linked-list.

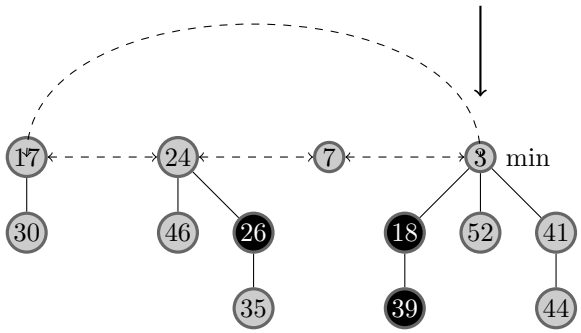


Figure 5.3: Heap Representation

5.4 Node Representation

- Each node stores a pointer to it's parent.
- Each node stores a pointer to any of it's children.
- Each node stores a pointer to it's left & right siblings.
- Each node stores it's rank where rank means the number of children of the node.
- Each node stores whether it is marked or not.

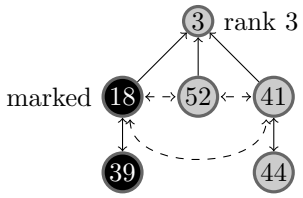


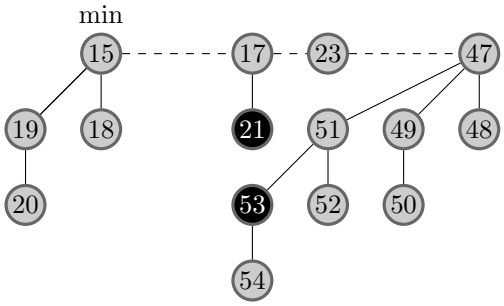
Figure 5.4: Node Representation

5.5 Basic Operations

5.5.1 Insert

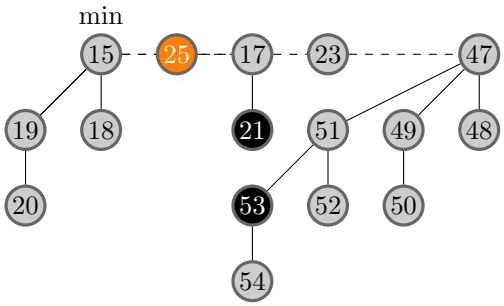
Let's say we want to insert 25 to the Fibonacci heap. To insert 25, we need to:

1. Create a new node
2. Add the newly created node to the root list.
3. Update the minimum pointer if necessary.



(a) Fibonacci heap

(b) Creating a new node



(c) Fibonacci Heap after inserting 25

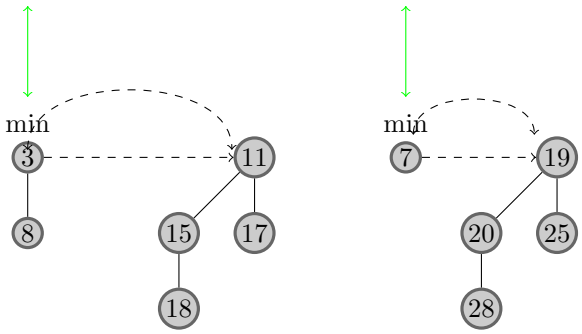
Figure 5.5: Insert Operation

Running Time Analysis:

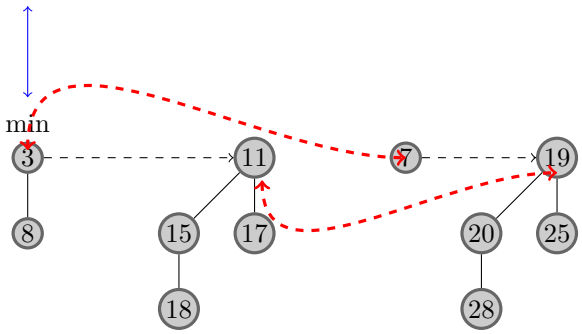
- We can create a new node and add it to the root list in constant time.
- So, the running time of insert operation : $O(\log n)$

5.5.2 Union

Union is performed by joining two roots of two different heaps. To join the two roots, we need to delete two edges and add two edges between two roots of two heaps. Then we need to update the minimum pointer.



(a) Fibonacci Heaps H1 & H2



(b) Fibonacci Heap after Union operation

Figure 5.6: Union Operation

Running Time Analysis:

- We can delete two edges and two edges between two roots in constant time.
- So, the running time of union operation : $O(\log n)$

5.5.3 Decrease key

Suppose x is the node which we will decrease. Select the key and change the value with new value. There are 3 cases which are-
1.**When Parent is unmarked :** If the new key is smaller(if it maintains min heap order property) than its parent(y) and parent(y) is not marked, then we cut the node from the tree. We meld the node into the root list. If the node is marked, then we unmark it. Lastly We update the minimum of the root list if necessary.
2.**When parent is marked :** If the new value is smaller(if it maintains min heap order property than its parent(y) and it is marked, we cut the node from the tree. We meld it into the rootlist.
Since parent's 2nd child is cut, we cut the parent from the tree, meld it into the root list and unmark it.
We do this cascading cut for its parent and continue it
3.**When Parent is greater :** It parent is greater than new value, then no modification is necessary.

Running time of decrease key is O(1)

Example-
1.Decrease-key of x from 15 to 3

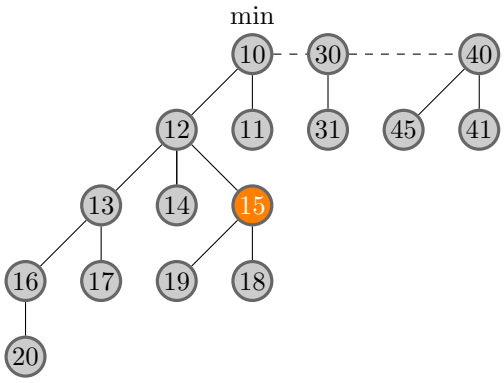


Figure 5.7: Fibonacci tree

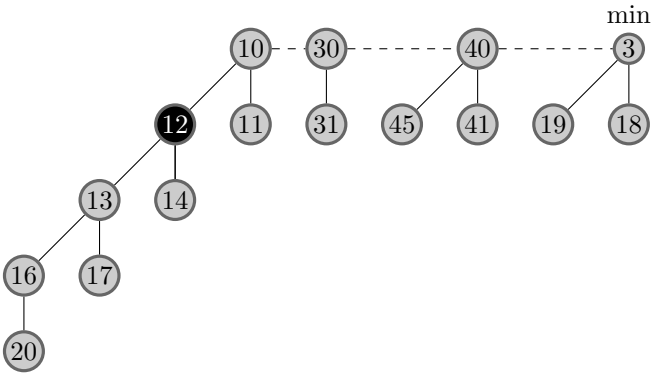


Figure 5.8: Case 1 is applied

2.decrease-key of x from 13 to 4

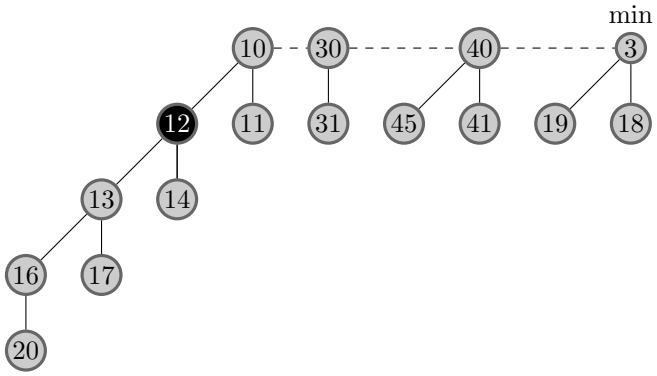


Figure 5.9: Fibonacci tree

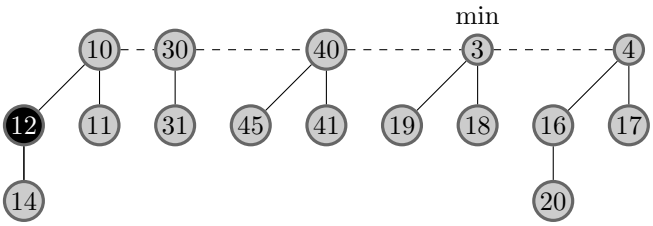


Figure 5.10: Case 2 is applied.Child is cut and melded into the tree.

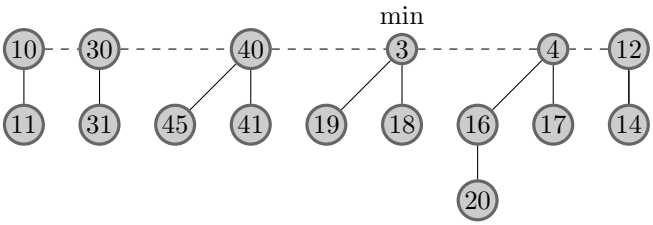


Figure 5.11: After Cascading of marked node

5.5.4 Extract Min

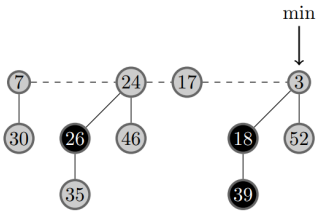
It is the most important operation on a fibonacci heap. In this operation, we remove the node with minimum value from the heap and re-adjust the tree. Later on, We consolidate trees in the heap so that no two roots have the same rank. We follow the following processes to implement extract min-

1. We delete the min pointer.
2. We meld its children into the root list.
3. We update the minimum pointer.

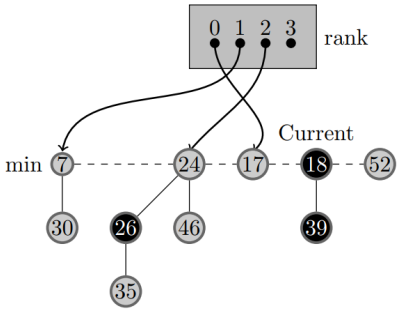
4. We create a current pointer which points at min pointer initially. .
5. We set the rank to the current pointer.
6. We move the current pointer forward.
    - if rank is different from any other trees, then move forward.
    - if rank is same with any other tree, then merge those two trees into one tree and update the rank of the current tree.
7. We repeat 5 and 6 process til there is no trees left.

Running time of extract min is O(logn)

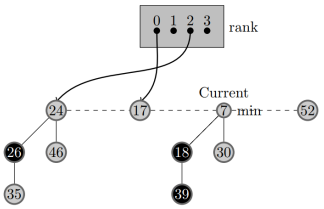
Example-



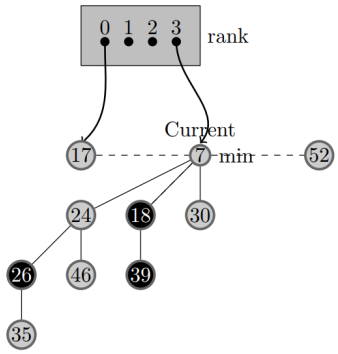
(a)



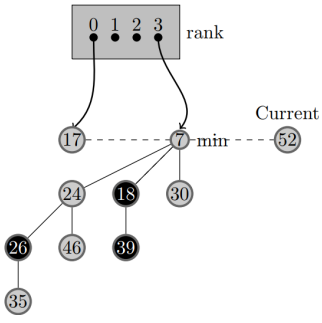
(b)



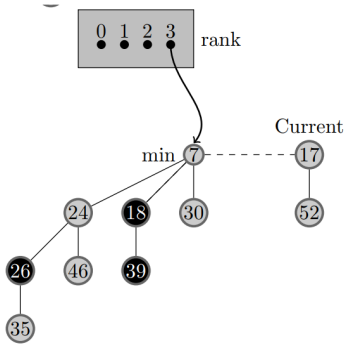
(c)



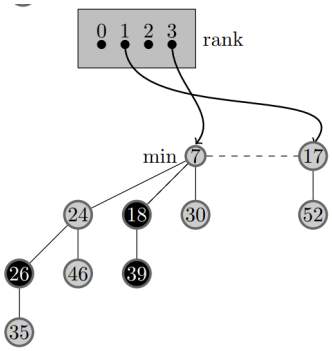
(d)



(e)



(f)



(g)

Figure 5.12: Extract min

5.5.5 Delete

To delete any node from fibonacci heap, previous two operation are applied. These are -

1. We decrease key
2. We extract min

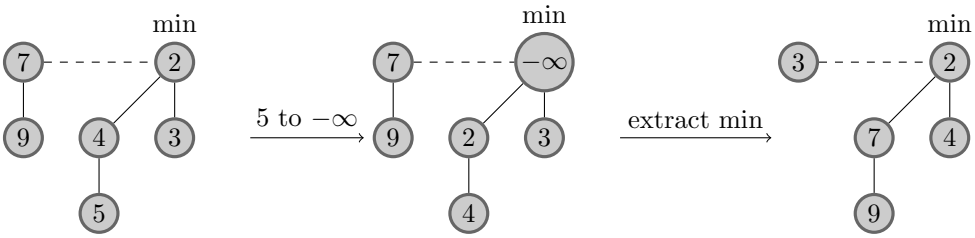
Following algorithm is followed-

1. We decrease the node to minus infinity.
2. We bring the node into the root list by heapifying the heap containing the minus infinity.
3. We perform the extract min operation.

Running time of decrease key : O(1)
Running time of extract min : O(logn)

Running time of Delete key : O(logn)

Example-



5.5.6 Find min

Finding minimum is easy in fibonacci heap. Minimum element is always given by Min pointer. So this operation takes constant time.

Running time of Find Min : O(1)

5.6 Maximally Damaged tree

Binomial heap with order k has 2^k number of nodes. But in fibonacci heap 2^k property are not satisfied always.When we cut the children in fibonacci heap, 2^k property does not apply on fibonacci heap.
If a tree does not grow with height, height does not remain logn. After cutting a children in fibonacci heap, we want to show that fibonacci tree with k height has c^k nodes where $c > 1$. we do not want to delete too many nodes from a

tree, as we still want a tree of rank k to have c^k nodes. The reason for wanting this property is so that the searching part of delete-min should take only O(log n) steps.
Here is an example of tree with rank 5. We will cut its higher degree nodes without cutting the root node.

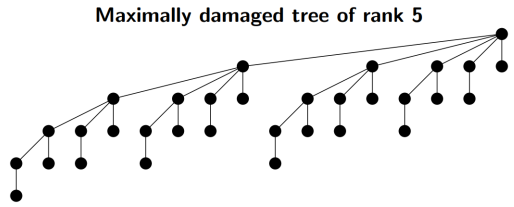


Figure 5.13: Binomial tree of rank 5

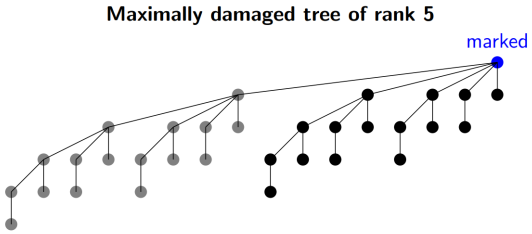


Figure 5.14

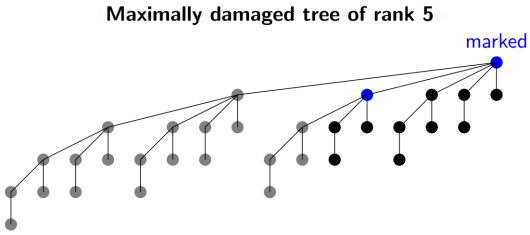


Figure 5.15

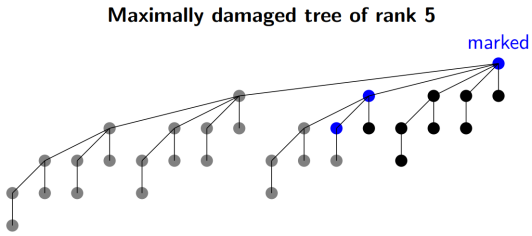


Figure 5.16

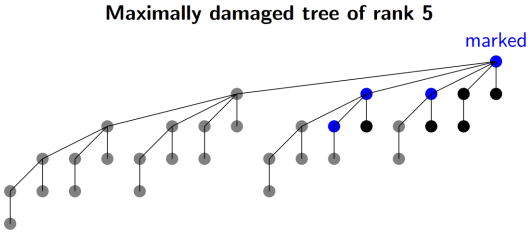


Figure 5.17

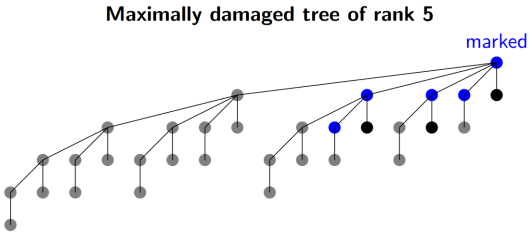


Figure 5.18

So this is the maximally damaged tree with rank 5.This is a fibonacci tree. This tree has c^k number of node where c>1 and $c < 2$. If we damaged binomial tree with different rank , we will get fibonacci tree corrresponding to the binomial tree.

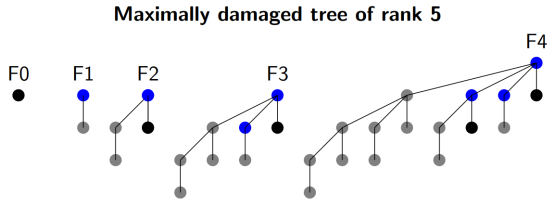


Figure 5.19

In Figure 21 , we can see the formation of fibonacci tree. Current tree is consist of its first and second preceding. This is like fibonacci number series.That is why this heap is called fibonacci heap.

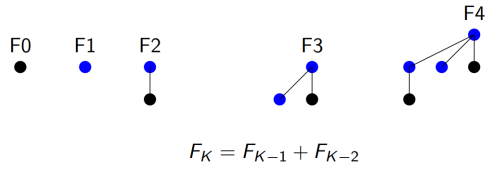


Figure 5.20

Size of f_k is: $S_k >= c^k$
Fibonacci trees still grow exponentially fast in k.

$$c = (1 + \sqrt{5})/2$$

rank = $O(log_c k) = O(log n)$

5.7 Amortized Running time

Amortized running time is a method to compute complexity of an algorithm. Amortized running time is that looking at the worst running time per operation. Here is an example of amortized running time -
If a number of insertion, b number of decrease key and c number of extract min are taken,
the amortized running time : **O(a+b+clogn)**

Chapter 6

Conclusion

Heaps have a variety of application in network optimization and in many such applications. Thus Fibonacci heap is also faster for several well-known network optimization problems. Purpose of fibonacci heap is to speed up Dijkstra's algorithm for single source shortest path problem with non negative edges.Now Djkstra's algorithm runs O(nlogn+m) improved from $O(mlog_{m/n+2}n)$[1].
Various other network use Dijkstra's algorithm as a subroutine and for each of these, we obtain a corresponding improvement.

Bibliography

[1] Michael L. Fredman and Robert Endree Tarjan. *"Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms"*. Journal of the ACM, 34(3):596-615, 1987.