Fibonacci Heap

University of Colorado Boulder

CSCI 5454 - Design and Analysis of Algorithms

2016

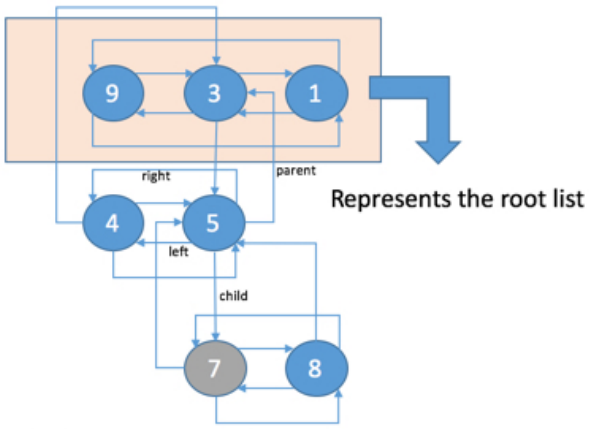
Mahesh Kumar Ravindranathan

1. INTRODUCTION TO FIBONACCI HEAP

Fibonacci Heap is a data structure that is generally used for priority queue operations, consisting of a collection of heap-ordered trees [2, 4] . The data structure was developed by Micheal L.Fredman and Robert E. Tarjan in 1984. They are named Fibonacci Heaps because their running time analysis is based on Fibonacci numbers.

Fibonacci heap is a collection of min-heap-ordered trees (The data contained in each node is less than or equal to the data in the node's children). In Fibonacci heaps, the trees are rooted but unordered. Each node contains a pointer to its parent and a pointer to any one of its children. The children are linked together in a circular, doubly linked list. The order in which the siblings appear in a child list is arbitrary. The number of children in the child list of a node is stored as degree. Boolean value is used to mark a node which indicates whether the node has lost a child since the last time it was made the child of another node.

For simplicity, the future figures won't have the representation of the doubly linked list left and right pointer. I will be representing just the parent pointer. The orange layer indicates the root list.



Each Node has 4 pointers namely
Parent: Root List Nodes have their parent pointer set to None
Child: Leaf Nodes have their child set to None
Parent point to only one child among their children
Left & Right: These pointers are used to represent the circular, doubly linked list at each level

Each Node has 3 properties namely
Key: Indicates the value of the key/data
Degree: Indicates the number of children
Mark: Stores history with the help of a Boolean value.

For node with key 5, the degree is 2 and it is not marked (False)
Marked nodes are colored grey

Figure 1: Fibonacci Heap Example

2. OPERATIONS OF FIBONACCI HEAP

The various operations of the Fibonacci Heaps are as follows:

- Inserting a node
- Union of Fibonacci Heaps
- Extracting the Minimum Node
- Decreasing a key
- Deleting a node

Let us consider n to be the number of nodes in the Fibonacci Heap.

2.1 Inserting a node

In Fibonacci Heap, inserting a node is $O(1)$ operation since it is equivalent to adding a node to a circular doubly linked list called the root list Figure 2 and Figure 3 . The added node has no children and is unmarked.

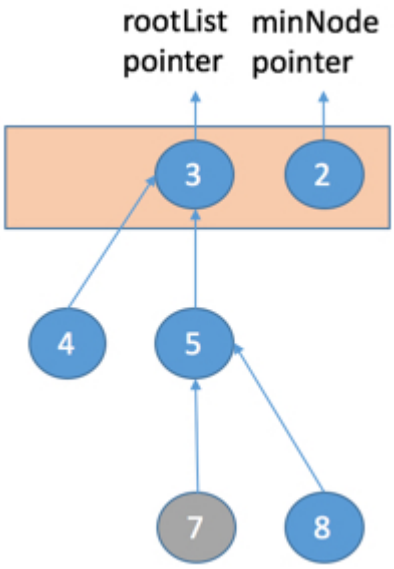


Figure 2: Fibonacci Heap before Insertion

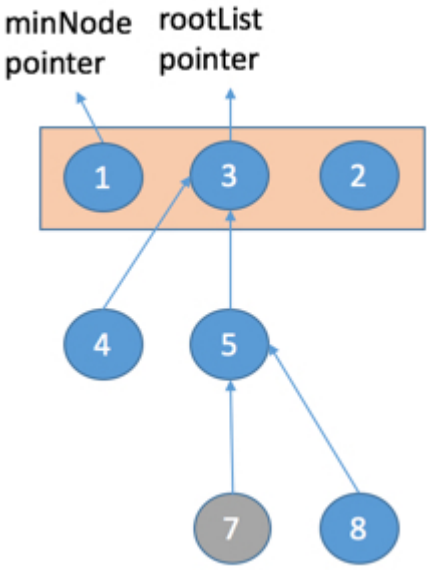


Figure 3: Fibonacci Heap post Insertion of a node of key 1

2.2 Union of Fibonacci Heaps

In Union operation, the root list of both the Fibonacci Heaps are united and the new minimum node is determined Figure 4 and Figure 5 . The amortized cost of the union operation is $O(1)$ as well.

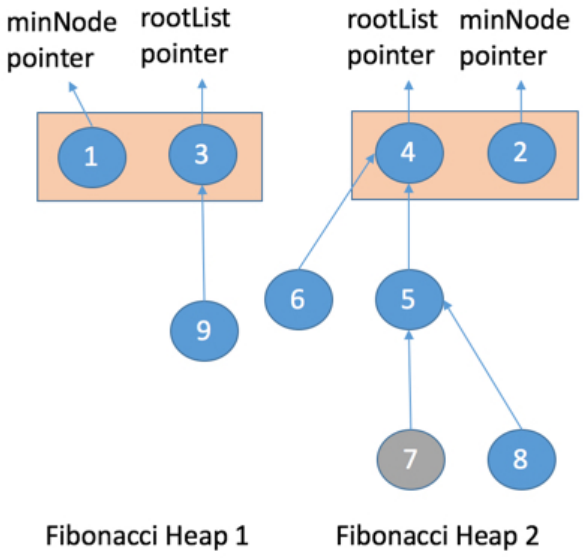


Figure 4: Two Fibonacci Heap before Union Operation

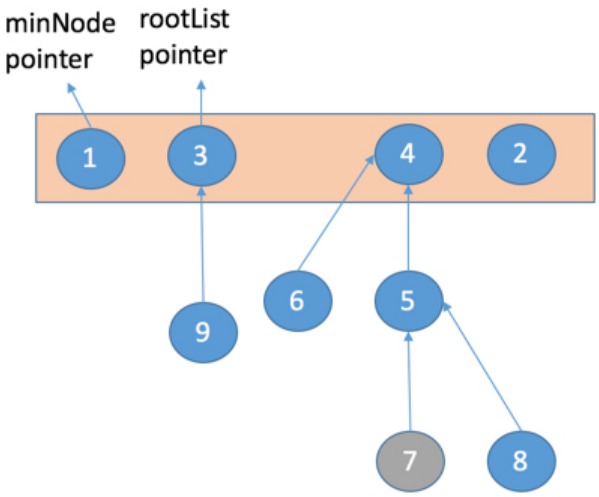


Figure 5: Fibonacci Heap after Union Operation

2.3 Extracting the Minimum Node

In the process of extracting the minimum node, the minimum node is extracted from the Fibonacci Heap. The procedure followed is as follows:

- The minimum node's children are added to the root list.Figure 6
- Consolidate the root list by linking roots when they are of equal degree.Figure7
- Repeat the previous step until at most one root remains of each degree.Figure8

- Update the minimum node with the minimum key in the root list.Figure 9

The amortized cost of extracting a minimum node is $O(\lg n)$

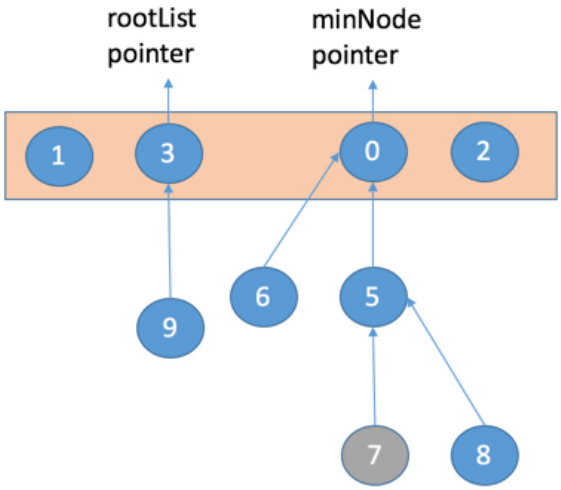


Figure 6: The minimum node's children are added to the root list

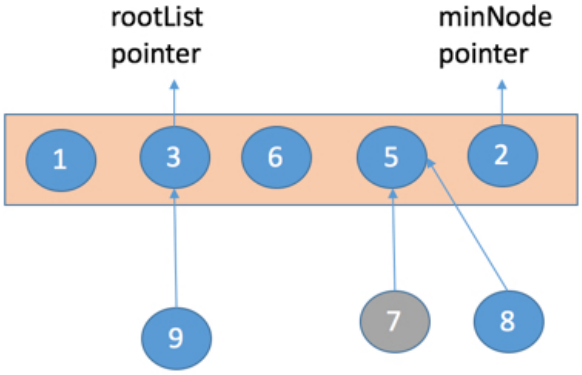


Figure 7: Consolidate Operation

2.4 Decreasing a key

In this process, we decrease the value of the key for an existing node. If the min-heap order is not violated, we need to update the minimum node if required Figure 10 and Figure 11 . If the min-heap order is violated, many changes can occur. The procedure followed is as follows:

- We cut the node from it's parent Figure 12 and Figure 13 .
- Add the node to the root list Figure 14 .
  - If the parent is not marked, the parent is marked Figure 14.

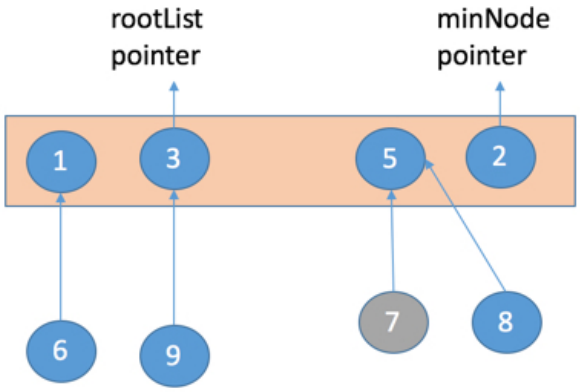


Figure 8: Repeat the previous step until at most one root remains of each degree.

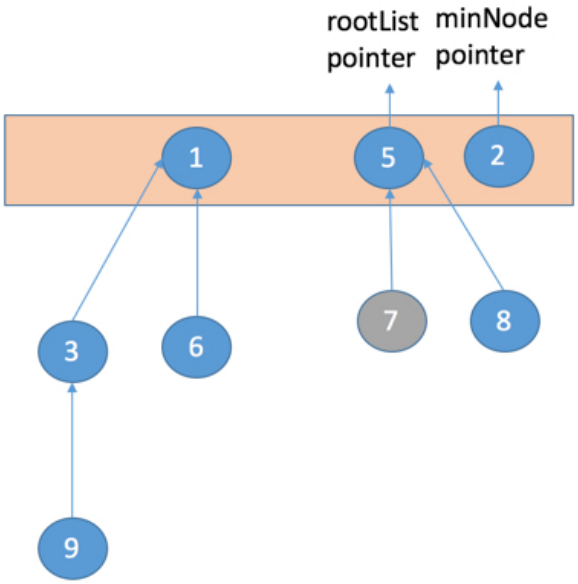


Figure 9: Updating the Minimum Node pointer

  - If the parent was already marked, it would be added to the root list as well and this step occurs recursively up the tree until either a root or an unmarked node is found Figure 14 .

- If the decreased key is less than the minimum node's key, we update the minimum node Figure 15 .

The amortized cost of decreasing a key is $O(1)$

2.5 Deleting a node

Deleting a node is equivalent to decreasing a key to -∞ and then extracting the minimum node (The node whose key was updated to -∞) for a Fibonacci Heap and hence it has an amortized cost of $O(\lg n)$

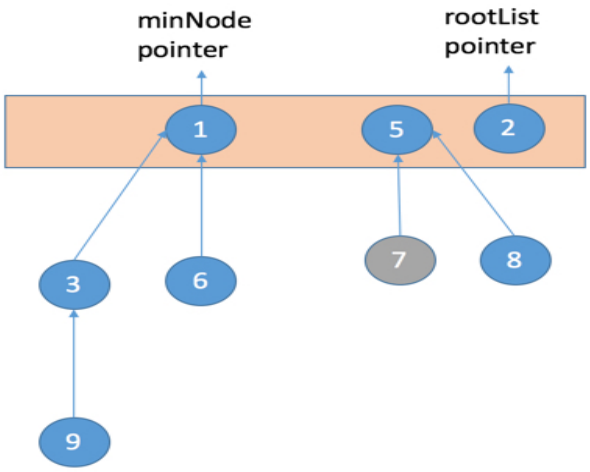


Figure 10: Fibonacci Heap before decreasing the key 5 to 0

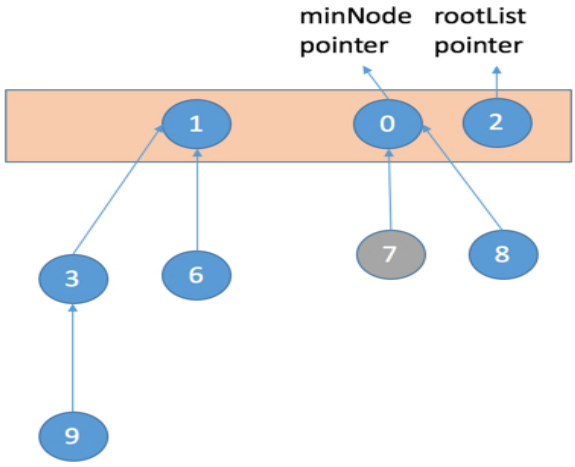


Figure 11: Fibonacci Heap after decreasing the key 5 to 0

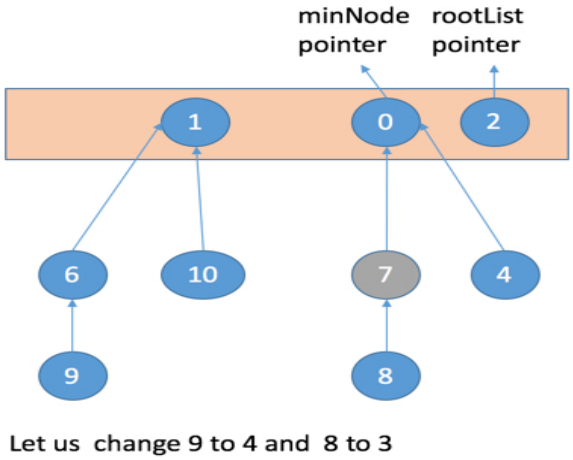


Figure 12: Fibonacci Heap before decreasing a key which violates the min-heap property

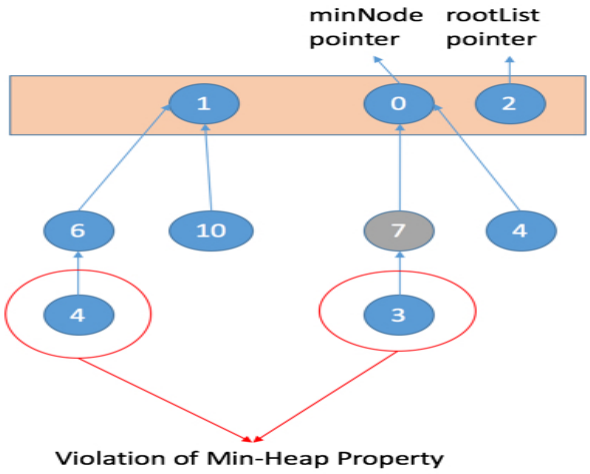


Figure 13: Nodes with key 4 and 3 violate the min-heap property

3. FIBONACCI HEAP IMPLEMENTATION

For the implementation of the Fibonacci Heap, I created a FibonacciHeap class and a Node class. The FibonacciHeap Node class has the following variables:

- key: Indicates the value of the key(data)
- degree: The number of immediate child nodes
- parent: The parent of the node
- child: The child of the node
- left & right: The siblings if any are pointed using these pointers (circular, doubly linked list)
- mark: Used for tracking history and restricting the amortized cost for extracting minimum and deleting a node to $O(\lg n)$

The FibonaaciHeap class has the following variables:

- rootList: Pointer to a node of the root list
- minNode: Node with the minimum key value
- totalNode: Total number of nodes within the Fibonacci Heap

The FibonacciHeap class has multiple functions defined, their uses are as follows:

3.1 insert (Insert a node to the Fibonacci Heap)

- We create a new node

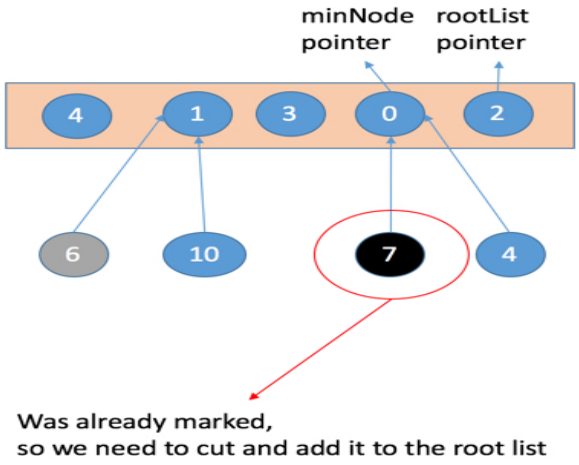


Figure 14: Node 4 and 3 are added to the root list. Node 6 is marked but node 7 was already marked so we unmark it and add it to the root list

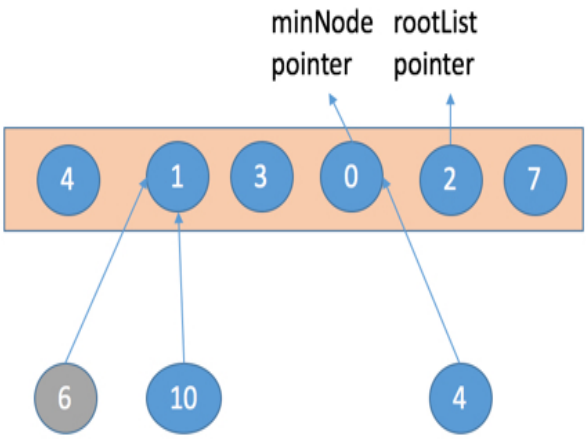


Figure 15: Fibonacci Heap after decreasing the key 5 to 0

- Make left and right pointer point to itself
- Call the mergeNewNodeWithRootList function
- If minNode was undefined or it has key greater than the new node, we update minNode to point to the new node
- We increase the totalNode count by 1 indicating that a new node was added to the Fibonacci Heap.

3.2 extractMin (Extracting the Minimum Node)

- We check if the minNode is undefined or not
- If minNode does exist, we check if the node has child node
  - If there exist a child Node, then we get the list of all the children using the iterateNodeDoubleLinkList function
  - We add all the children to the root list using the mergeNewNodeWithRootList function
- We remove the minimum node from the root list using the removeFromRootList function
- We update the minNode to a random node in the root list
- We call the consolidate function
- We decrease the total nodes by 1 as the minimum node is removed from the Fibonacci Heap

3.3 consolidate (Reduce the number of min-heap trees from the Fibonacci Heap)

- We create an empty list of size (total number of nodes in the Fibonacci Heap) called A
- We get all the nodes in the root list using the iterateNodeDoubleLinkList function
- For each node, we iterate and get it's degree
  - If the list A, has any node corresponding to the degree
  - We figure out which node has lower key value and add the larger key value node as a child to the lower key value node using the heapLink function
  - We update the list A with index as degree of the node to empty
  - We increment the degree of the node by 1
- We update the minNode with the node which is minimum among the root list

3.4 decreaseKey (Decreasing a key and updating the Fibonacci Heap)

- If the min-heap property is violated
  - We run the cut function
  - We run the cascadingCut function
- We update the minNode appropriately.

3.5 cut (If the child node is smaller than its parent after the decrease key operation, we cut the child and bring it back to the root list)

- We call the removeFromChildList function
- We decrease the degree by 1 and call the mergeNewNodeWithRootList function to add the removed child to the root list
- We mark the parent to empty and unmark it when it is added to the root list

3.6 cascadingCut (Based on the history (mark property), we decide if we need to recursively go up the tree, post a cut operation)

- If the node is not marked, we mark it
- If the node has been marked already, we recursively iterate up to the root or until a node is not marked by calling the cut and cascadingCut function.

3.7 heapLink (Remove a node from the rootList and add it as a child to another node of the rootList, maintaining the circular double linked list properties)

- Call the removeFromRootList function for the node that needs to be removed from the root list.
- Call the mergeWithChild function to merge the node as a child of another node
- Increase the degree of the node to which the node is added as child, update hat parent pointer and update the mark to False

3.8 removeFromRootList (Remove a node from the circular double linked root list)

- If the pointer rootList is pointing to it, update the pointer to another node in the root list
- Update the pointer of the doubly linked list to remove the node

3.9 iterateNodeDoubleLinkList (Iterating through the circular double linked list for a given node)

- If the node is undefined, return none.
- If the node is defined, iterate through a particular direction of the doubly linked list till we reach back to the original node.
- Return all the nodes that were traversed through.

3.10 mergeRootListWithExistingRootList (Joining two circular doubly linked root list)

- Update the pointers to join both the doubly linked lists of root list

3.11 mergeNewNodeWithRootList (Adding a node to doubly linked root list)

- If the rootList pointer is empty update the pointer to point to the node
- If the rootList pointer is not empty, add the node to the circular doubly linked root list

3.12 mergeWithChild (Adding a node as the child of a given parent node)

- If the parent has no child, just update the child pointer to point to the node
- If the parent has a child already, update the pointers to add the node to circular doubly linked list that contains all its children.

3.13 removeFromChildList (Removing a node from its parent)

- If it is the only child, update the parent's child pointer as empty
- If the parent had multiple child, update parent pointer with another child and the child's parent pointer with the parent node
- Update the pointers to remove the node from the circular doubly linked list

4. MATHEMATICAL ANALYSIS

We use potential method to analyze the performance of Fibonacci Heap Operations. For a Fibonacci Heap H, let us indicate the number of trees in the root list of H by $t(H)$ and the number of marked nodes in H by $m(H)$.
The potential of Fibonacci Heap H is defined by

$$\Phi(H) \ = \ t(H) + 2m(H)$$

Clearly,

- The potential when there is no heaps is zero.
- The potential is non-negative at all subsequent times.
- The amortized cost upper bounds the actual cost

The Mathematical Analysis for the run time of the various operations are as follows:

4.1 Inserting a node

Initial potential:

$$\Phi(H) \;=\; t(H) + 2m(H)$$

After insertion of a node, let the Fibonacci heap be denoted by H'.

$$\begin{aligned} t(H') &= t(H) + 1 \\ m(H') &= m(H) \end{aligned}$$

Increase in potential = New potential - Old potential

$$\begin{aligned} \triangle\Phi &= [t(H') + 2m(H')] - [t(H) + 2m(H)] \\ &= [t(H) + 1 + 2m(H)] - [t(H) + 2m(H)] \\ &= 1 \end{aligned} \tag{1}$$

Amortized cost = Actual cost + Increased potential

$$\begin{aligned} \textit{Amortized cost} &= O(1) + 1 \\ &= O(1) \end{aligned} \tag{2}$$

4.2 Union of Fibonacci Heaps

Initial potential:

$$\begin{aligned} \Phi(H_1) &= t(H_1) + 2m(H_1) \\ \Phi(H_2) &= t(H_2) + 2m(H_2) \end{aligned}$$

Let Fibonacci Heap tree formed by the union of H_1 and H_2 be denoted by H'

$$\begin{aligned} t(H') &= t(H_1) + t(H_2) \\ m(H') &= m(H_1) + m(H_2) \\ \Phi(H') &= t(H') + 2m(H') \end{aligned}$$

Increase in potential is given by

$$\begin{aligned} \triangle\Phi &= [t(H') + 2m(H')] \\ &\quad - [(t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))] \\ &= [(t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))] \\ &\quad - [(t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))] \\ &= 0 \end{aligned} \tag{3}$$

Amortized cost = Actual cost + Increased potential

$$\begin{aligned} \textit{Amortized cost} &= O(1) + 0 \\ &= O(1) \end{aligned} \tag{4}$$

4.3 Extracting the Minimum Node

Initial potential:

$$\Phi(H) \;=\; t(H) + 2m(H)$$

The actual cost of extracting the minimum node can be accounted as follows:

- An $O(D(n))$ contribution comes from there being at most $D(n)$ children of the minimum node that are added to the root list and processed.
- The size of the root list on calling the consolidate function is at most $D(n) + t(H) - 1$, since there were originally $t(H)$ trees, at most $D(n)$ of nodes can be the children to the node that is being extracted and one (1) since that is the node that is being extracted. Thus the total actual work done in extracting the minimum node is $O(D(n)+t(H))$

The change in potential is given as follows:

- Potential at most, post extracting the node is $(D(n) + 1 + 2m(H))$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation.

Amortized cost = Actual cost + Increased potential

$$\begin{aligned} \textit{Amortized cost} &= O(D(n) + t(H)) \\ &\quad + [[(D(n) + 1 + 2m(H)] - [t(H) + 2m(H)]] \\ &= O(D(n)) + O(t(H)) + D(n) + 1 - t(H) \\ &= O(D(n)) \end{aligned} \tag{5}$$

4.4 Decreasing a key

Initial potential:

$$\Phi(H) \;=\; t(H) + 2m(H)$$

The actual cost for decreasing a key can be accounted as follows:

- It takes $O(1)$ time, plus the time to perform cascading cuts.
- Suppose the cascading cut is recursively called c times and each call takes $O(1)$ time exclusive of recursive call, it takes $O(c)$ including all recursive call.

The change in potential is given as follows:

- Each recursive call of cascading cut, except for the last one, cuts a marked node and clears the marked bit.
- There are hence, $(t(H) + c)$ trees
- There are at most $(m(H) - c + 2)$ marked nodes

Amortized cost = Actual cost + Increased potential

$$\begin{aligned} \textit{Amortized cost} &= O(c) \\ &\quad + [(t(H) + c) + 2(m(H) - c + 2) - [t(H) + 2m(H)]] \\ &= O(c) - c + 4 \\ &= O(1) \end{aligned} \tag{6}$$

4.5 Deleting a node

Deleting a node is equivalent to decreasing a key and then extracting the minimum.
Hence,

$$\begin{aligned} Amortized\ cost &= O(D(n)) + O(1) \\ &= O(D(n)) \end{aligned} \tag{7}$$

4.6 Bounding the maximum degree

To show that the Extracting the minimum node operation and the Deleting a node operation takes $O(\lg n)$ we need to prove that the upper bound of $D(n)$ on the degree of any node of a n-node Fibonacci heap is $O(\lg n)$.

Let x be any node in a Fibonacci Heap, and suppose the $degree[x] = k$. If $y_1, y_2, y_3, ..., y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $degree[y_1] \geq 0$ and $degree[y_i] \geq i-2$ for $i = 2, 3, ..., k$ because when y_i was linked to x, all of $y_1, y_2, ..., y_{i-1}$ were children of x, so we must have had $degree[x] \geq i-1$. Node y_i is linked to x_i only if their degree are the same and hence $degree[y_i] = degree[x] \geq i-1$. Since then, y_i has lost at most one child, since it would have been cut from x if it had lost two children. We conclude that $degree[y_i] \geq i-2$

From the property of Fibonacci number: We have

$$\begin{aligned} F_k &= 0\ if\ k = 0 \\ F_k &= 1\ if\ k = 1 \\ F_k &= F_{k-1} + F_{k+2}\ if\ k \geq 2 \end{aligned}$$

For all integers $k \geq 0$ indicate the smallest possible tree of degree k , we get number of nodes in the sequence of Fibonacci numbers using the Fibonacci property: $n \geq \Phi^k$, where $\Phi \approxeq 1.1618$ and is the golden ratio. $F_{k+2} \geq \Phi^k$

For any node in a Fibonacci heap, let $k = degree[x]$. Then, we can show that the $size(x) \geq F_{k+2} \geq \Phi^k$, $where\ \Phi = \frac{(1+\sqrt{5})}{2}$.

Assuming the number of nodes as n.Taking base-Φ log on both the sides,we get $k \leq \log_\Phi n$.
The maximum degree $D(n)$ of any node is thus $O(\lg n)$.

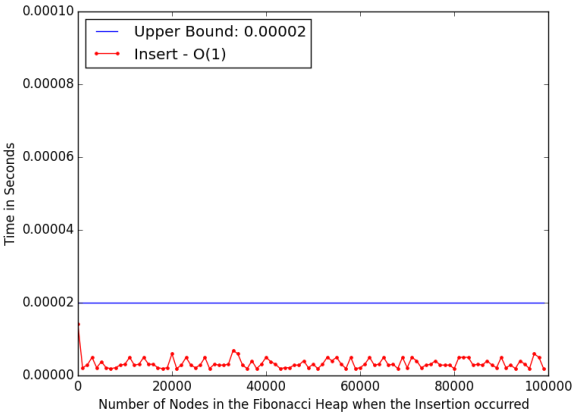


Figure 16: Insert Operation: $O(1)$
For Insertion operation, I created a new Fibonacci Heap with $n-1$ nodes and inserted the n^{th} node. Since it is constant time operation, we can observe that irrespective of the number of nodes added, the time taken to add the n^{th} node takes a constant time.

Time Complexity for Fibonacci Heap Operations

Operation & Time Complexity	
Find Min	$O(1)$
Insert	$O(1)$
Union	$O(1)$
Extract-Min	$O(\lg n)^*$
Decrease-Key	$O(1)^*$
Delete	$O(\lg n)^*$

* indicates amortized

5. EXPERIMENTATION, OBSERVATION AND RESULTS

6. PERFORMANCE FOR DIFFERENT INPUT

- Fibonacci heap has best performance when insertion or union operations are performed as it is equivalent to adding a node to a doubly linked list and it takes constant time. Hence the best performance can be obtained from Fibonacci heap when the frequent requirement is to use only insert and union operation. It takes constant time $O(1)$ to perform these operations and this is the main reason Fibonacci heap is used frequently in Network optimization algorithms. Fibonacci heap

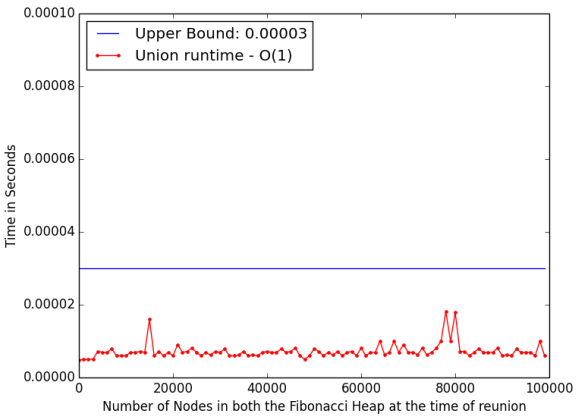


Figure 17: Union Operation: $O(1)$
For the Union operation, I created two Fibonacci Heap with n nodes and then merged them. Since it is a constant time operation, we can observe that irrespective of the number of the nodes in both the Fibonacci Heap at the time of merging, the operation completes in a constant time.

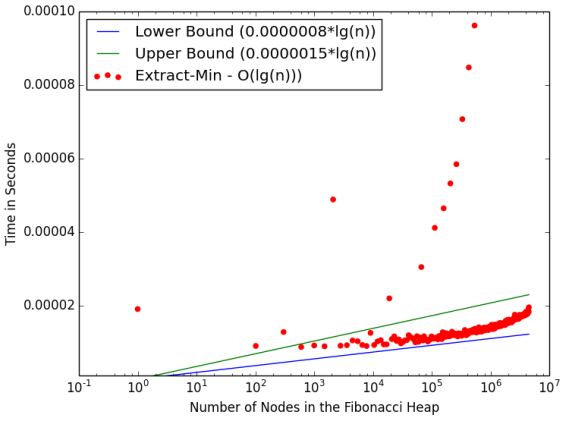


Figure 18: Extract Min Operation: $O(\log n)^*$
For the Extract-Min operation, I create just one Fibonacci-Heap and keep adding nodes to it and perform Extract-Min operation at various stages. Since it has an amortized time bound , the extract-min operation at any stage is dependent on the previous stage when the extract-min operation occurred and hence I decided to use just one Fibonacci Heap for the entire analysis. I have a lower and upper bound with a constant time $\log n$ which shows that Extract-Min has an amortized time bound of $O(\log n)$.
Note: The X-axis is in log scale.

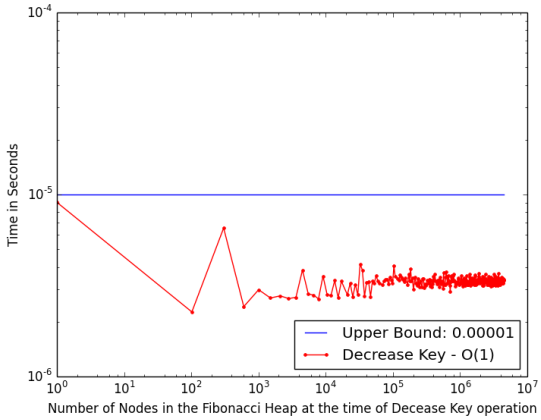


Figure 19: Decrease Key Operation: $O(1)^*$
For the Decrease-Key operation, I create a Fibonacci-Heap and add nodes to it. At various stages, I perform the decrease-key operation. I use a single Fibonacci-Heap since decrease-key runtime bound is amortized. The current decrease key operation runtime is based on when the previous decrease key operation was performed.
Note: The X-axis is in log scale.

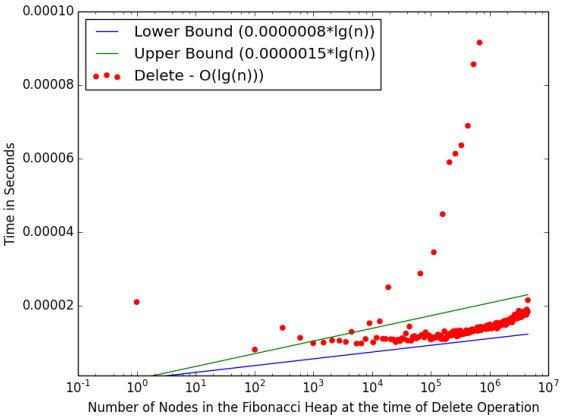


Figure 20: Delete Operation: $O(\log n)^*$
For the Delete operation, I create just one Fibonacci-Heap and keep adding nodes to it and perform delete operation at various stages. Since it has an amortized time bound, the delete operation at any stage is dependent on the previous stage when the delete operation occurred and hence I decided to use just one Fibonacci Heap for the entire analysis. I have a lower and upper bound with a constant time $\log n$ which shows that Delete operation has an amortized time bound of $O(\log n)$.
Note: The X-axis is in log scale.

has good performance even when it is used for decrease key operation where the key/data of a node is modified. Depending on the key/data of the parent node the restructure may or may not happen (heap order violation). If the restructure does not happen it is a $O(1)$ operation, but if the restructuring occurs then it's amortized cost is $O(1)$

- Fibonacci heap has a heap data structure similar to that of Binomial Heap, with slight modifications and a looser structure. Fibonacci heap defers from Binomial heap in the context of the consolidate operation which occurs only at the time of extract minimum and delete operation. Due to the deferred clean up, the worst case time complexity of delete and extract minimum operations is $O(n)$, however they are $O(\log n)$ amortized. Hence the worst case performance for the Fibonacci heap is when delete operation or extract min operation are performed and the time bound would be $O(\log n)$.
- Fibonacci heap has average performance when major operation performed frequently is that of insert/union and decrease-key and we use delete and extract-min operation once in a while.
- Although the total running time of operations starting with an empty structure is bounded by the bounds given, some operations in the sequence can take very long to complete (especially for extract min and delete minimum operation which have linear running time in the worst case). Hence Fibonacci Heaps may not be appropriate for real-time systems.
- The space complexity is higher than Binomial heaps, because it uses two pointer to the siblings and one pointer to its parent. If we have n nodes, than the space needed to store them is around $(O(n) + 4p)$, since each node in the Fibonacci Heap has four pointers namely child, parent, left and right.

7. APPLICATIONS

Generally in Network Optimization, the number of deletion operations is relatively small. Thus, Fibonacci Heaps can be used to obtain asymptotically faster algorithms. It is used in the following implementations [5] (Let n be the number of vertices and m be the number of edges):

- Dijkstra's algorithm for the single-source shortest path program with non-negative edges. It improves the time bound to $O(n \log n + m)$ from previously best known time bound $O(m \log_{\frac{m}{n+2}} n)$
- All-pairs shortest path problem had a improvement of time bound to $O(n^2 \log n + nm)$ from $O(nm \log_{\frac{m}{n+2}} n)$
- Weight bipartite matching had improvement of time to $O(n^2 \log n + nm)$ from $O(nm \log_{\frac{m}{n+2}} n)$
- Minimum spanning tree problem had an improvement to $O(m\beta(m, n))$ from $O(m \log \log_{\frac{m}{n+2}} n)$, where $\beta(m, n) = min\{i | \log^{(i)} n\} \leq \frac{m}{n}$. Note that $\beta(m, n) \leq \log^* n \; if \; m \geq n$

8. CONCLUSION

Although Fibonacci heaps looks efficient, they are not generally used in practical application because:

- It is complicated to code them. They are not as efficient in practice when compared with the theoretically less efficient forms of heaps, since in their simplest version they required storage and manipulation of four pointer per node, when compared to the two or three pointers per node needed for other structures.
- They have the reputation for being slow in practice due to large memory consumption per node and high constant factors on all operations.

References

[1] Princeton university fibonacci heap. `https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf`.

[2] Wikipedia fibonacci heap. `https://en.wikipedia.org/wiki/Fibonacci_heap`.

[3] D. Abuaiadh and J. H. Kingston. *Are Fibonacci heaps optimal?* Springer, 1994.

[4] T. H. Cormen. *Introduction to algorithms.* MIT press, 2009.

[5] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.