Fibonacci heaps analysis and comparison with Binary Heap

November 21, 2021

Sourabh Sanganwar (2020csb1121) ,
Sanyam Walia (2020csb1122) ,
Sukhmeet Singh (2020csb1129)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sarthak Joshi

Summary: **Fibonacci heap** has much better amortized time complexity compared to binary and binomial heap. According to [2] Fibonacci heaps support deletion and decrease key$O(\log n)$ amortized time and all other standard heap operations in $o(1)$ amortized time..Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.

1. Introduction

This document focuses on the **structure and run time of Fibonacci Heaps**. Unlike the basic binary heaps which are limited to have 2 leaves, Fibonacci Heap is a collection of many heaps with their root nodes linked to each other via a doubly linked circular list. The amortized run time of these Fibonacci Heaps is much better than the basic binary heaps as we will see in our analysis. Thus, all the algorithms such as Djikstra's, Prim's Min Cost Spanning Tree, can be made even more efficient by using Fibonacci heaps in place of the Binary Heap. Following operations are supported by heap data structure:

1. **Make-Heap** - Return a new, empty heap.
2. **Insert (H, i)** - Insert a new item i with predefined key into heap h.
3. **Union**(H_1,H_2) - Return the heap formed by taking the union of the item disjoint heaps h1 and h2.
4. **Minimum(H)** - Returns an item of minimum key.
5. **Extract-Min(H)** - Returns minimum element and deletes it from heap.
6. **Decrease-Key(H, x, k)** - decreases value of key in x to k in heap H.
7. **Delete(H,x)** - Deletes item x from H. This function previously knows position of x.

2. Structure of Fibonacci heaps

A Fibonacci heap is a collection of rooted trees that are min-heap ordered.

Each heap object has 2 attributes. H.min is a pointer to node with minimum key in heap and H.n is number of nodes in heap.

Each node x in the Fibonacci Heap has a pointer x.p to its parent and a pointer x.child to any of its children. The children of x are connected in a circular doubly linked list with left and right pointers of each node pointing to its siblings. Such lists are called the child list of x if x is parent of elements in list. If node is an only child, then node.left = node.right = node may appear in a child list in any order.

Nodes also contain x.degree, x.mark attributes. We store number of children in x.degree. The Boolean-valued attribute x.mark indicates whether node x has lost a child since the last time x was made the child of another node.
Potential method is used for time complexity analysis

Below figure is representation of Fibonacci Heap:

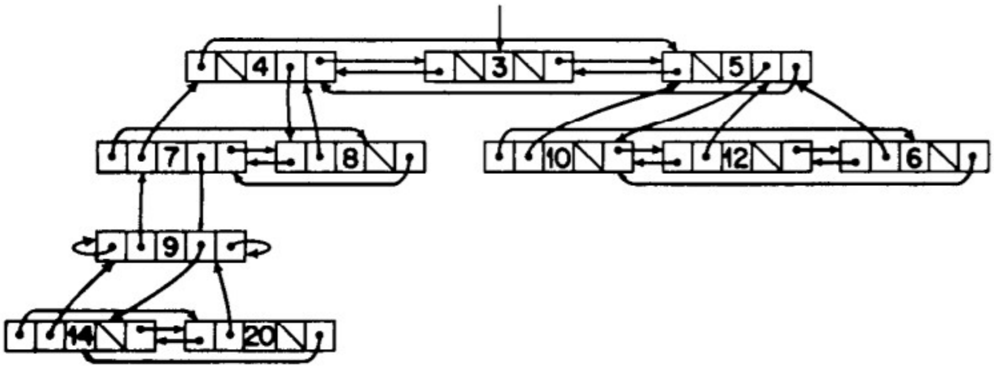

Figure 1: Representation of Fibonacci Heap.

2.1. Potential Function

let $t(H)$ be number of trees in root list of H.
$m(H)$ be number of marked nodes
Then according to [1] Potential function is defined as follows:
$\Phi(H) = t(H) + 2m(H)$
As initially heap is empty hence $t(H) = 0$ and $m(H) = 0$. Hence $\Phi(H) = 0$

2.2. Tables

Comparison Of time complexity

	Binary Heap	**Fibonacci Heap**
Make-Heap	$O(1)$	$O(1)$
Insert	$O(\log n)$	O(1)
Minimum	O(1)	O(1)
Extract-Min	$O(\log n)$	$O(\log n)$
Union	O(n)	O(1)
Decrease-Key	$O(\log n)$	O(1)
Delete	$O(\log n)$	$O(\log n)$

Table 1: Time complexity of Fibonacci Heap and Binary Heap

From above table we observe that insert, union and decrease-key operations have better time complexities hence Fibonacci heaps have better amortized run time than binary heap.

2.3. Algorithms

Following are Algorithms of different functions in Fibonacci heap:

Algorithm 1 Make-Heap()

```
H.min=NULL
H.n=0
return H
```

Make-Heap returns empty heap.
The potential of the empty Fibonacci heap is $\Phi(H) = 0$.
Hence amortized cost of Make-Heap is $O(1)$.

Algorithm 2 Insert(H,node)

```
Initialize node
Add node in root list of H
if x.key < H.min.key then
    H.min = x
end if
H.n=H.n+1
```

Let H be heap before inserting and $t(H')$ be heap after inserting node Insert function insert element in root list. Hence number of trees in root list increases by 1 and number of marked nodes doesn't change.
Change in potential $= t(H') - t(H) = 1$
Hence amortized cost of Make-Heap is $O(1)$.

Algorithm 3 Union(H_1,H_2)

```
H = MAKE-FIB-HEAP()
H.min = H1.min
concatenate the root list of H2 with the root list of H.
H.min=minimum(H1.min,H2.min)
H.n=H1.n + H2.n
return H
```

Union returns heap with after uniting both heaps.
Let $\Phi(H)$ be final potential and $\Phi(H_1), \Phi(H_2)$ be potential of initial 2 heaps.
As number of trees in H are sum of number of trees in H_1 and H_2
hence $t(H) = t(H_1) + t(H_2)$ similarly $m(H) = m(H_1) + m(H_2)$
Hence $\Phi(H) = \Phi(H_1) + \Phi(H_2)$
Change in potential $= \Phi(H) - (\Phi(H_1) + \Phi(H_2)) = 0$
Hence amortized cost of Make-Heap is $O(1)$.

Algorithm 4 Extract-Min(H)

```
temp=H.min
if temp ≠ NULL then
    Add child of temp to root list of H
    remove temp from root list of H
    if temp == temp.right then
        H.min=NULL
    else
        H.min=temp.right
        CONSOLIDATE(H)
    end if
    H.n=H.n-1
end if
return temp
```

Extract-Min first adds children of minimum node to root list and removes minimum node from the root list. Then it consolidates the root list by linking all trees of same degree.

Algorithm 5 CONSOLIDATE(H)

```
1: let A[0...D(H.n)] be a new array
2: for i = 0 to D(H.n)  do
3:   A[i]=NIL
4: end for
5: for each node w in root list of H do
6:   x=w
7:   d=x.degree
8:   while A[d] ≠ NIL do
9:     y=A[d]
10:    if x.key > y.key then
11:      exchange x with y
12:    end if
13:    FIB-HEAP-LINK(H,y,x)
14:    A[d]=NIL
15:    d=d+1
16:  end while
17:  A[d]=x
18: end for
19: H.min=NULL
20: for i = 0 to D(H.n)  do
21:  if A[i] ≠ NIL then
22:    if H.min==NIL then
23:      Create a root list for H containing just A[i]
24:      H.min=A[i]
25:    else
26:      Insert A[i] into H's root list
27:      if A[i].key<H.min.key then
28:        H.min=A[i]
29:      end if
30:    end if
31:  end if
32: end for
```

Algorithm 6 FIB-HEAP-LINK(H,y,x)

```
remove y from the root list of H
make y a child of x, increment x.degree
y.mark=FALSE
```

Consolidate links trees of same degree such that tree with greater key in root linked in child list of other tree.
Potential before extracting minimum node is: $t(H) + 2m(H)$
Let $D(n)$ be maximum degree of any node.
From Corollary 1 $D(n) = O(\log(n))$
Extract-Min process at-max of $D(n)$ child.
The size of the root list upon calling CONSOLIDATE is at most $D(n) + t(H) - 1$
Total amount of work performed in for loop in line 5 of consolidate function is $O(D(n) + t(H))$ [1]
The amortized cost is thus at most
$O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H))$
$=O(D(n)) = O(\log(n))$

Algorithm 7 Decrease-Key(H, x, k)

if $k > x.key$ **then**
  "new key is greater than current key"
end if
x.key= k
y=x.p
if $y \neq NULL\ \ and\ \ x.key < y.key$ **then**
  CUT(H, x, y)
  CASCADING-CUT(H, y)
end if
if $x.key < H.min.key$ **then**
  H.min = x
end if

Algorithm 8 CUT(H,x,y)

remove x from the child list of y, decrementing y.degree
add x to the root list of H
x.p=NIL
x.mark=FALSE

Algorithm 9 CASCADING-CUT(H,y)

z=y.p
if $z \neq NIL$ **then**
  if y.mark==FALSE **then**
    y.mark=TRUE
  else
    CUT(H,y,z)
    CASCADING-CUT(H,z)
  end if
end if
add x to the root list of H
x.p=NIL
x.mark=FALSE

Analysis of Decrease key:

By analysis of Decrease-Key algorithm all operations are of constant time.
Cut function runs in constant time as there is no looping and assignment take $O(1)$ time.
Let t(H) be initial number of trees and m(H) be initial marked keys. Decrease-key creates a new tree rooted at node x and clears x's mark bit. Each call of CASCADING-CUT, except for the last one, cuts a marked node and clears the mark bit. Afterward, the Fibonacci heap contains $t(H) + c$ trees, c is a constant. New heap has at most $m(H) - c + 2$ marked nodes.
Hence amortized cost is:
$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c$
Amortized cost $= O(c) + 4 - c = O(1)$

Algorithm 10 Delete(H,x)

FIB-HEAP-DECREASE-KEY(H, x,- ∞)
FIB-HEAP-EXTRACT-MIN(H)

As Delete calls 2 functions of O(1) complexity and O($\log n$)
Amortized Cost = O(1) + O($\log n$) = O($\log n$)

3. Lemmas

Lemma 1. *Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, \ldots y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, \ldots, k$.*

Lemma 2. *For all integers $k \geq 0$,*

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Lemma 3. *For all integers $k \geq 0$, the $(k+2)$nd Fibonacci number satisfies $F_{k+2} \geq \phi^k$.*

Lemma 4. *Let x be any node in a Fibonacci heap, and let $k = x.degree$. Then $size(x) \geq F_{k+2} \geq \phi^k$, where $\phi = \frac{(1+\sqrt{5})}{2}$.*

Corollary 1. *The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\log n)$.*

4. Conclusion

Through the course of this project, we looked deeper into the structure and run time of Fibonacci Heaps. Thus we can conclude that Fibonacci heaps have a better run time complexity as compared to Binary Heaps and so, they can find their future applications in the field of Dynamic programming and in algorithms related to Graphs and their traversal. Many in progress research papers also describes the application of Fibonacci heaps to the problems of single-source shortest paths, all-pairs shortest paths, weighted bipartite matching, and the minimum-spanning tree problem.

5. Bibliography and citations

https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/
https://www.geeksforgeeks.org/fibonacci-heap-insertion-and-union/

Acknowledgements

We would like to thank Dr. Anil Shukla for inspiring us and making our basics clear which made this project possible and thanks to our TA Sarthak Joshi for guiding us.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[2] Michael Fredman and Robert Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 07 1987.

A. Appendix A

Corollary 1 Proof:

The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\log n)$.

Proof: Let x be any node in an n-node Fibonacci heap, and let $k = x.degree$. By Lemma 4, we have $n \geq size(x) \geq \phi^k$. Taking base-$\phi$ logarithms gives us $k \leq \log_\phi n$. (In fact, because k is an integer, $k \leq \lfloor \log_\phi n \rfloor$.) The maximum degree $D(n)$ of any node is thus $O(\log n)$.

Following is comparison of run time of different heaps:

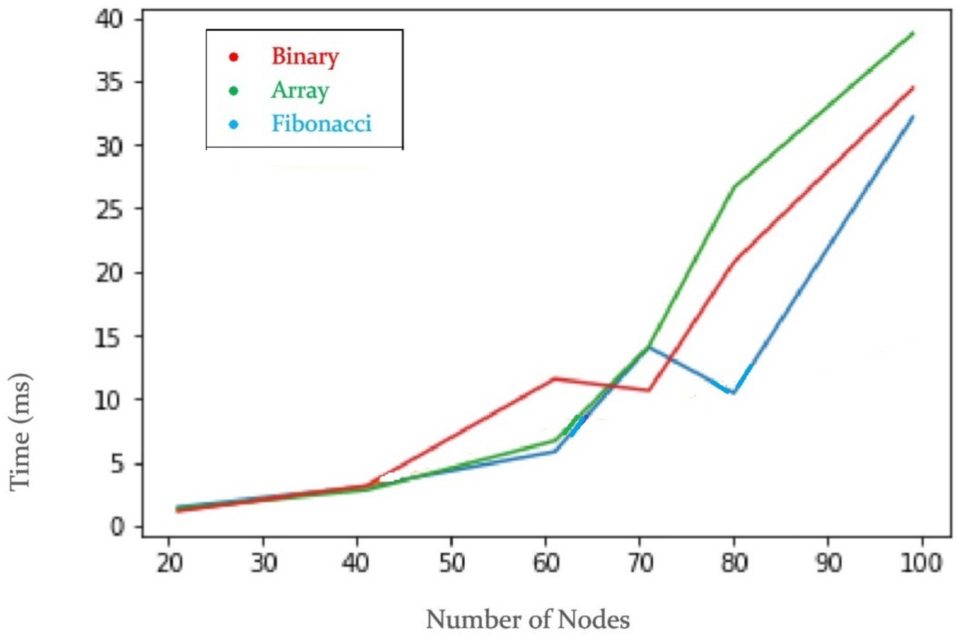

Figure 2: Run time comparison of heaps.

Following is implementation of insert:

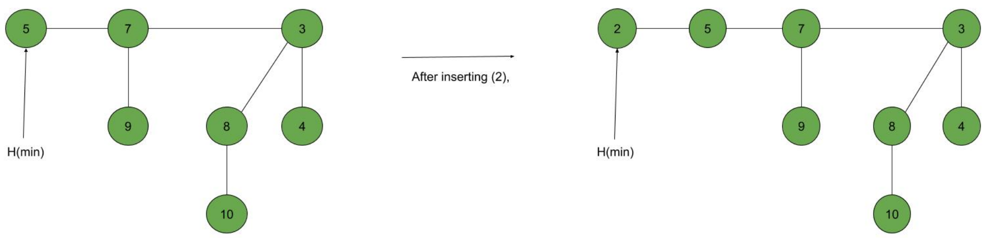

Figure 3: Fibonacci insert.

Following is implementation of union:

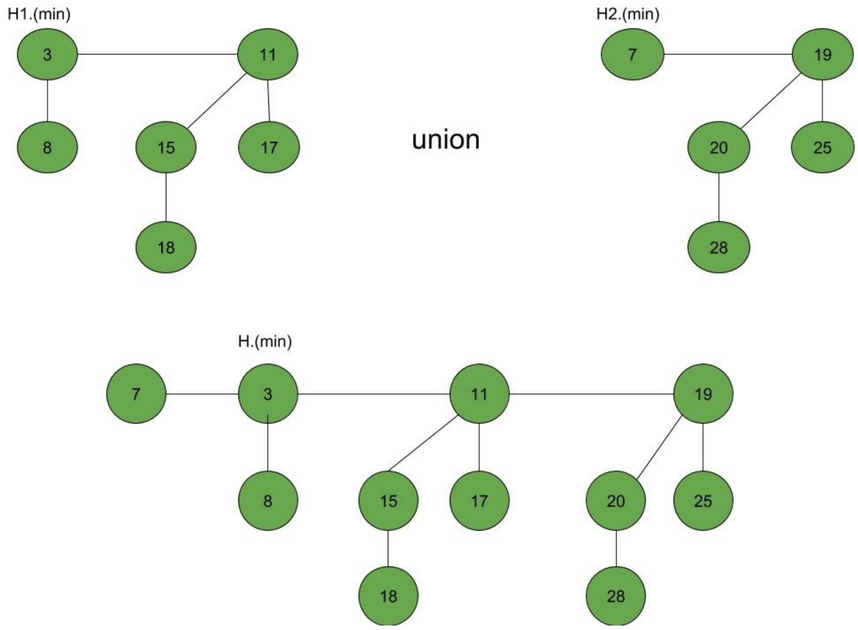

Figure 4: Fibonacci Union.

Following is implementation of extract-min:

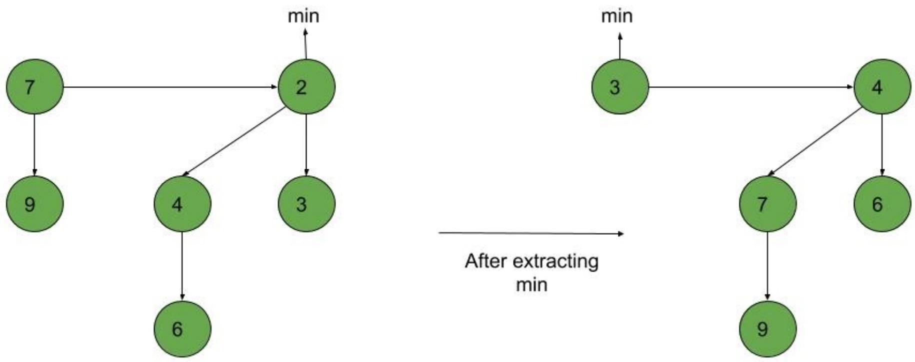

Figure 5: Fibonacci extract min.

Following is implementation of dec-key:

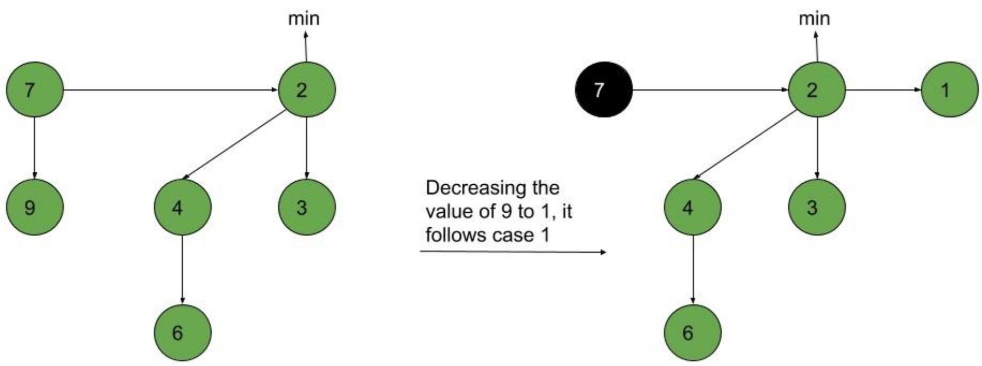

Figure 6: Fibonacci decrease-key.