Fibonacci Heap

November 7, 2022

Hardik Aggarwal (2021CSB1173) ,
Harshit Kumar Ravi (2021CSB1093) ,
Gyanendra Mani (2021CSB1090)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Vinay Sajja Kumar

Summary: Implemented a Fibonacci heap and performed amortized analysis of all the operations implemented using a potential function. The various operations implemented are insertion, deletion, extracting minimum, decrease key and union. Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.

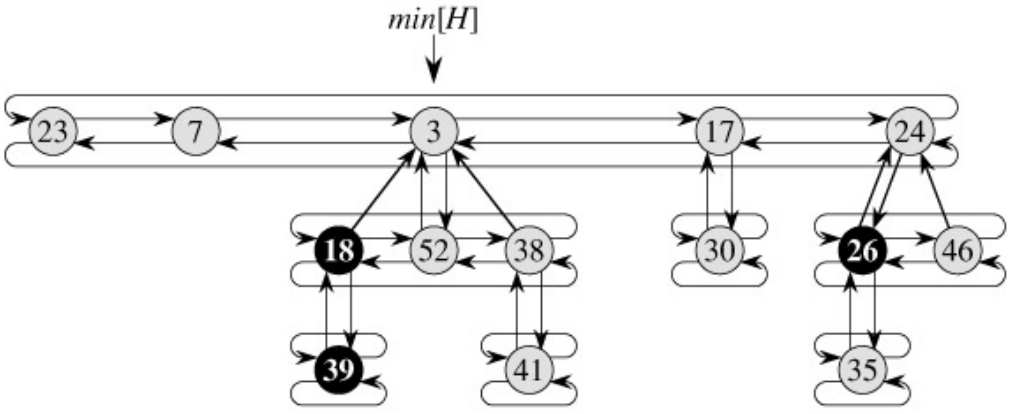
---

1. Introduction

Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes. Fibonacci heap maintains a min pointer which points to the node having minimum value. It takes constant amortized time in most of its operations.

1.1. Structure of Fibonacci heaps

A Fibonacci heap is a collection of rooted trees that satisfy the heap properties.
Each heap structure has two properties: min is a pointer to node with minimum key value and n is the number of nodes in the heap.
Each node in the heap has a pointer to its parent and a pointer to any of its children. The root nodes are connected by a doubly linked list called the root list and similarly the children of all the nodes are connected within a doubly linked list with their siblings.
Each node has a degree and a mark attribute. The degree attribute represents the number of children of that node and the mark is a boolean attribute which shows whether the node is marked or not. A node is marked if that node has lost a child since the last time it was made the child of another node.



Structure of a Fibonacci heap

2. Algorithms

2.1. Insertion

The following algorithm is used to insert a node x into a Fibonacci heap H, where the node x has already been allocated along with its key. It simply inserts the node x into the root list.

Algorithm 1 FIB-HEAP-INSERT(H,x)

```
1: x.degree=0
2: x.p = NIL
3: x.child = NIL
4: x.mark = FALSE
5: if H.min == NIL then
6:     create a root list for H containing just X
7:     H.min = x
8: else
9:     insert x into H's root list
10:    if x.key < H.min.key then
11:        H.min = x
12:    end if
13: end if
14: H.n = H.n + 1
```

2.2. Union

The following algorithm unites Fibonacci heaps H_1 and H_2 into a single heap. It simply concatenates the root lists of H_1 and H_2 and then determines the new minimum node.

Algorithm 2 FIB-HEAP-UNION(H_1,H_2)

```
1: H = MAKE-FIB-HEAP()
2: H.min = H1.min
3: concatenate the root list of H2 with the root list of H
4: if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key) then
5:     H.min = H2.min
6: end if
7: H.n = H1.n + H2.n
8: return H
```

2.3. Decrease Key

The following algorithm reduces the key of x to a new key value k. It uses the functions CUT and CASCADING-CUT to cut those nodes from the parent that are violating the heap's property.

CUT: This function cuts the target node from its parent node and inserts it into the root list.

CASCADING-CUT: This function recursively removes those nodes that are marked and marks the unmarked nodes.

Algorithm 3 FIB-HEAP-DECREASE-KEY(H,x,k)

```
1: if k > x.key then
2:     error "new key is greater than current key"
3: end if
4: x.key = k
5: y = x.p
6: if y ≠ NIL and x.key < y.key then
7:     CUT(H,x,y)
8:     CASCADING-CUT(H,y)
9: end if
10: if x.key < H.min.key then
11:     H.min = x
12: end if
```

Algorithm 4 CUT(H,x,y)

```
1: remove x from the child of y, decrementing y.degree
2: add x to the root list of H
3: x.p = NIL
4: x.mark = FALSE
```

Algorithm 5 CASCADING-CUT(H,y)

```
1: z = y.p
2: if z ≠ NIL then
3:     if y.mark == FALSE then
4:         y.mark = TRUE
5:     else
6:         CUT(H,y,z)
7:         CASCADING-CUT(H,z)
8:     end if
9: end if
```

2.4. Extract Min

The following algorithm extracts the minimum node. It removes the minimum node and puts all its children nodes into the root list. It also consists of the CONSOLIDATE function which consolidates the root list which results in a root list with each root node having a distinct degree.

CONSOLIDATE: This function is used to consolidate the root list of heap. It consists of repeatedly executing the following steps until every root in the root list has a distinct degree:

1. Find two roots x and y in the root list with the same degree. Without loss of generality, let x.key $\leq$ y.key.

2. Link y to x: It is performed by the FIB-HEAP-LINK function.

FIB-HEAP-LINK: This function is used to link two heaps of same degree into a single heap of degree one greater than the previous. The nodes x and y are the nodes of same degree which are to be linked by this function assuming x.key $\leq$ y.key for generality. It removes y from the root list and makes y a child of x. Then it increments x.degree and clears the mark of y.

Algorithm 6 FIB-HEAP-EXTRACT-MIN(H)

```
1: z = H.min
2: if z! = NIL then
3:    for each child x of z do
4:       add x to the root list of H
5:       x.p = NIL
6:    end for
7:    remove z from the root list of H
8:    if z == z.right then
9:       H.min = NIL
10:   else
11:      H.min = z.right
12:      CONSOLIDATE(H)
13:   end if
14:   H.n = H.n - 1
15: end if
16: return  z
```

Algorithm 7 CONSOLIDATE

```
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 the 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 = NIL
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 8 FIB-HEAP-LINK

1: remove y from the root list of H
2: make y a child of x, incrementing x.degree
3: y.mark = FALSE

2.5. Deletion

The following algorithm deletes a node from a heap. It is performed by first decreasing that key to -∞ and then removing the minimum node which will be the current node. We assume that initially there is no key value of -∞ in the heap.

Algorithm 9 FIB-HEAP-DELETE(H,x)

1: FIB-HEAP-DECREASE-KEY(H,x,$-\infty$)
2: FIB-HEAP-EXTRACT-MIN(H)

3. Analysis

The potential method is used to analyze the time complexities of Fibonacci heap operations . For a given Fibonacci heap H, the potential function is given as
$\phi(H) = t(H) + 2m(H)$
where t(H) is the number of trees in the root list of H and m(H) is the number of marked nodes in H.
We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, the subsequent potential function value will be non negative. Thus , the potential function value of subsequent forms of the Fibonacci heap have greater values than that of its initial form.

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Figure 1: Comparison of running times for operations on binary and Fibonacci heap

3.1. Insertion

Fibonacci heap tends to be lazy in its insertion operation To insert an element in Fibonacci heap , the element is inserted in the root list which is created as a doubly linked list.

The amortized cost of insertion:
Let H be the input Fibonacci heap and H' be the resulting Fibonacci heap.
Then, t(H') = t(H) + 1 and m(H') = m(H) , the increase in potential is: $\phi(H') - \phi(H)$
(t(H) + 1 + 2m(H)) - (t(H) + 2m(H)) = 1
Since the actual cost is O(1), the amortized cost is O(1)+1 = **O(1)**.

3.2. Extract-Min

The root list after CONSOLIDATE is at most D(n) + t(H) - 1, since it consists of the original t(H) root list nodes, extracted min node and added children of the extracted node.

The actual cost of extracting the minimum node is O(D(n) + t(H))
Now, the change in potential after the extract-min operation is:

ϕ_{before} = t(H) + 2m(H)
ϕ_{after} = D(n) + 1 + 2m(H)

There will be atmost (D(n) + 1) children in the root list after the extract-min operation, because after CONSOLIDATING, the nodes can have a maximum of (D(n) + 1) degree.
No nodes are marked during this operation.

For calculating the amortized cost of the extract-min operation,
Amortized cost = Actual cost + Change in potential
= O(D(n) + t(H)) + ((D(n) + 1 + 2m(H)) - (t(H) + 2m(H)))
= O(D(n)) + O(t(H)) - t(H)
= O(D(n))

For any node x in n-node Fibonacci heap, taking k = x.degree
By Corollary 1, D(n) = O(log n). So, the amortized cost of extracting the minimum node is **O(log n)**.

3.3. Union

Union operation of Fibonacci heap unites two Fibonacci heap (H1 and H2) into one Fibonacci heap (H) , destroying the individual Fibonacci heap (H1 and H2) in the process. The Union operation is performed by simply concatenating the root list of both the Fibonacci heap and then updating the min pointer.
The change in potential after Union operation : final state of Fibonacci heap - initial state of Fibonacci heap

$\phi(H) - (\phi(H_1) + \phi(H_2))$
= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2)))
= (t(H) - t(H_1) - t(H_2)) + 2(m(H) - m(H_1) - m(H_2))
= 0
where t(H) = t(H_1) + t(h_2) and m(H) = m(H_1) + m(h_2)
Therefore , the amortized cost of union operation is **O(1)**

3.4. Decrease-Key

This operation decreases the key value of a node x to a new key value k. If it violates the heap property, then it cuts the node from parent and places it in the root list and clears its mark. Then it recursively checks through the parent of this node removing the marked nodes clearing their marks and marking the unmarked nodes.
After performing all the CASCADING-CUT operations, the Fibonacci heap contains t(H) + c trees: The original t(H) trees, c-1 trees produced by cascading cuts,and the tree rooted at x and at most m(H) - c + 2 marked nodes: c-1 were unmarked by cascading cuts ans the last call of CASCADING-CUT may have marked a node.
Hence the change in potential is therefore at most ((t(H) + c) + 2(m(H) -c +2)) - (t(H) + 2m(H)) = 4-c. Thus the amortized cost of FIB-HEAP-DECREASE-KEY is at most O(c) + 4 - c = **O(1)**.

3.5. Deletion

The amortized time of this operation is the sum of the O(1) amortized time of FIB-HEAP-DECREASE-KEY and the O(log n)/(O(D(n)) amortized time of FIB-HEAP-EXTRACT-MIN. Hence the amortized time of FIB-HEAP-DELETE is **O(log n)**.

4. Lemmas

Lemma 1
Let x be any node in a Fibonacci heap, and suppose that x.degree = k. Let y_1, y_2,...,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 ≥ 0 and y_i.degree $\geq$ i - 2 for i = 2, 3, ... , k.

Lemma 2 For all integers k ≥ 0,

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

Lemma 3 For all integers k ≥ 0, the (k + 2)th 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 $\phi = (1 + \sqrt{5})/2$

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

5. Conclusion

Fibonacci heap has much better amortized time complexity compared to binary and binomial heap. Fibonacci heaps support deletion and decrease key in O(log n) amortized time and all other standard heap operations in O(1) amortized time. It can be used to speed up various algorithms using heaps such as Dijkstra's algorithm, giving the algorithm a very efficient running time.

6. Bibliography and Citations

Introduction to Algorithms [1], University of Cambridge [2]: Amortized analysis and lemmas

Wikipedia [3] : Introduction and structure of Fibonacci heaps

Acknowledgements

We are thankful to our Teaching Assistant *Vinay Sajja Kumar* for guiding us throughout the project and instructor *Dr. Anil Shukla* for motivating and inspiring us to make this project.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition, 2009.

[2] Frank Stajano and Thomas Sauerwald. Fibonacci heaps analysis.

[3] Wikipedia. Fibonacci heap.

A. Appendix A

Proof of Corollary 1

Statement: 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 $\geq \phi^k$.
Taking log with base ϕ gives us k $\leq log_\phi$n.
Thus, the maximum degree D(n) of any node is O(log n).