Fibonacci Heap: A Data Structure Used for Improved Network Optimization Algorithms

Jarin Tasneem
Student ID: 2005019
Swastika Pandit
Student ID: 2005027
Fairuz Mubashwera
Student ID: 2005030

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

March 9, 2024

Contents

List of Figures

1 Introduction

Fibonacci heap is a data structure for implementing priority queues. It is an optimized extension of binomial heap. While binomial heap supports all heap operations in $O(logn)$ worst case time complexity for a n-item heap, Fibonacci heap supports arbitrary deletion in $O(logn)$ amortized time, and all other heap operations in $O(logn)$ time. It is particularly useful where number of insert and decrease-key operations is significantly larger than delete and extract-min operations.

2 History of Using Heap structures in Computer Science

In the 1960s, heaps were introduced as a data structure in computer science. Initially, the term "heap" referred to a memory region used for dynamic memory allocation. This era marked the inception of structured heap data structures, laying the foundation for their evolution in subsequent decades.

The 1970s witnessed the rise of binary heaps, including both binary min-heaps and max-heaps, as fundamental data structures. Their efficient representation using arrays and straightforward operations made them integral for implementing priority queues, significantly impacting algorithm design and efficiency during this period.

In the 1980s, Bernard O. Tree introduced binomial heaps, a notable advancement in heap structures. Binomial heaps addressed certain limitations of binary heaps, providing more efficient merge operations. These heaps found applications in various algorithms, notably contributing to the optimization of Dijkstra's shortest path algorithm.

Figure 1: Michael L. Fredman and Robert E. Tarjan

A pivotal moment occurred in 1986 when Michael L. Fredman and Robert E. Tarjan introduced Fibonacci Heaps through their paper *"Fibonacci Heaps and Their Uses."* Designed to enhance the efficiency of decrease key operations, Fibonacci Heaps proved particularly suitable for algorithms such as Dijkstra's and Prim's. This marked a significant milestone in the ongoing evolution of heap data structures, showcasing the continual quest for optimized solutions in computer science.

3 Motivation

A Comparison with Other Data Structures

Operation	**Binary heap**	**Binomial heap**	**Fibonacci heap**
Insert	O(log(n))	O(log(n))	O(1)
Decrease key	O(log(n))	O(log(n))	O(1)
Extract min	O(log(n))	O(log(n))	O(log(n))
Delete	O(log(n))	O(log(n))	O(log(n))
Find min	O(1)	O(log(n))	O(1)
Union	O(n)	O(log(n))	O(1)

Table 1: Time complexity comparison of Fibonacci heap with other heap structures

In binomial heap, most of the operations can be done in $O(logn)$ time. But when we try to extract-min or delete, decrease key operation is needed to often. Since in binomial heap, each decrease-key operation would need a heapify operation of time-complexity $O(logn)$, the overall performance becomes slow. To solve this problem, Fibonacci heap was introduced, where the prime motivation was to gain $O(1)$ amortized time-complexity in decrease-key operation.

Fibonacci heap structure is much more relaxed and less strict than binomial heap data structure. Due to "Lazy Consolidation" we can merge and decrease a key without having to heapify in each of these operations. Owing to this feature and relaxation in its structure, Fibonacci heap has been able to bring significant performance improvement in optimization of graph and network algorithms.[5]

4 Structure

- A Fibonacci heap is a collection of item-disjoint *heap-ordered trees*
  A heap-ordered tree is a rooted tree where for any node x, the key of the children of x is no less than key of x, provided x has children. Thus the root contains the minimum key node.
- The number of children of a node is called its *rank r(x)*.
- Each node is either *marked* or *unmarked*. [5]

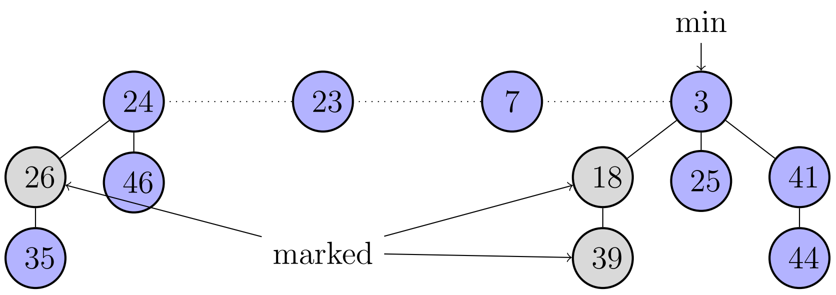

Figure 2: A Fibonacci heap

5 Memory Representation of the Nodes of Fibonacci Heap

- Each node contains a pointer to its parent (or *null* if it has no parent) and a pointer to one of its children.
- The children of each are stored in a doubly-linked list.
- Each node contains its rank, and a bit to indicate whether it is marked.
- The roots of all the trees in a heap are stored in a doubly-linked list.
- Heap is accessed by a pointer to a root containing an item of minimum key, which is called the *minimum node* of heap. A minimum node of null indicates an empty heap.

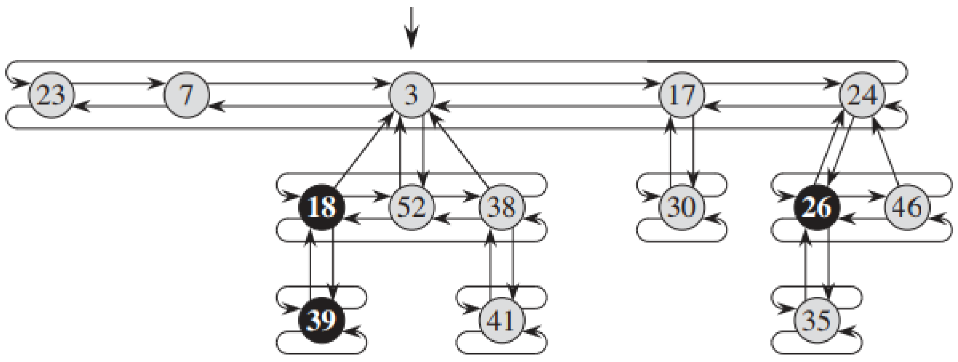

Figure 3: complete representation showing pointers p (up arrows), child (down arrows), and left and right (sideways arrows)[2].

6 Fibonacci Heap Operations

Fibonacci heap offers a diverse array of operations, comparable to other common structures like binary heaps and binomial heaps. Some of the most common operations associated with Fibonacci heaps are:

- Make-Heap
- Insert
- Union
- Find-min
- Extract-Min
- Decrease-key
- Delete

We'll see implementations of each operation one by one.

6.1 Make-Heap

This operation creates an empty heap. A heap is represented by a single tree rooted at the minimum element in the heap. Initially, this tree is empty.

6.2 Insert

This operation adds a new element to the heap. The new element is added as a single tree rooted at the new element and linked to the existing trees in the heap. The minimum element of the tree is compared to the newly inserted element. If the new element is smaller than the element pointed by *min pointer*, then the *min pointer* is updated to point to the new element, otherwise it remains unchanged.

- Make a new tree with the new key.

- Add the new tree to the heap (usually next to the min pointer).

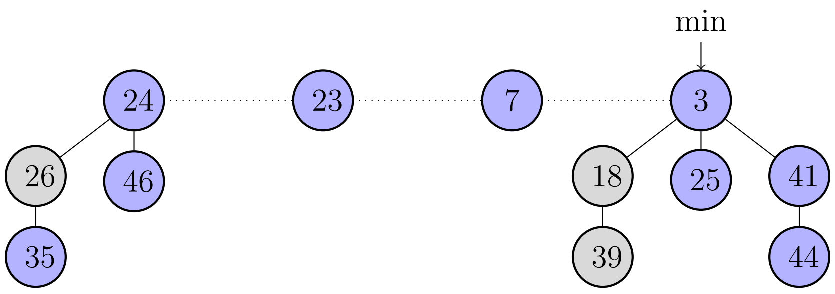

Figure 4: In this example, 21 is the new key to be inserted in the Fibonacci heap. A new tree is created with a single node 21.

- Update the min pointer if required.

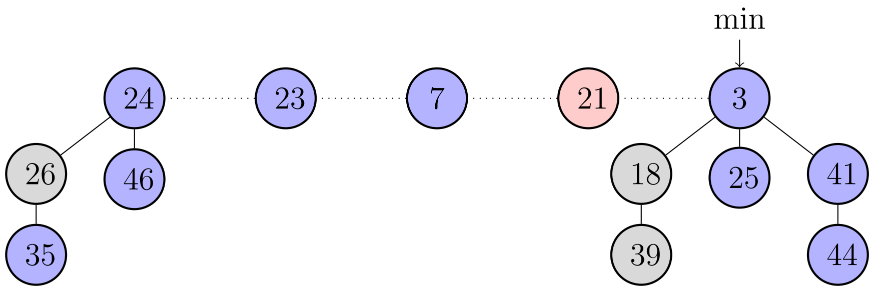

Figure 5: The newly created tree is added to the Fibonacci heap. Since 21 is greater than min node 3, no need to update the min-pointer

6.3 Union

Union operation takes two Fibonacci heaps H1 and H2 and merges them to one heap H

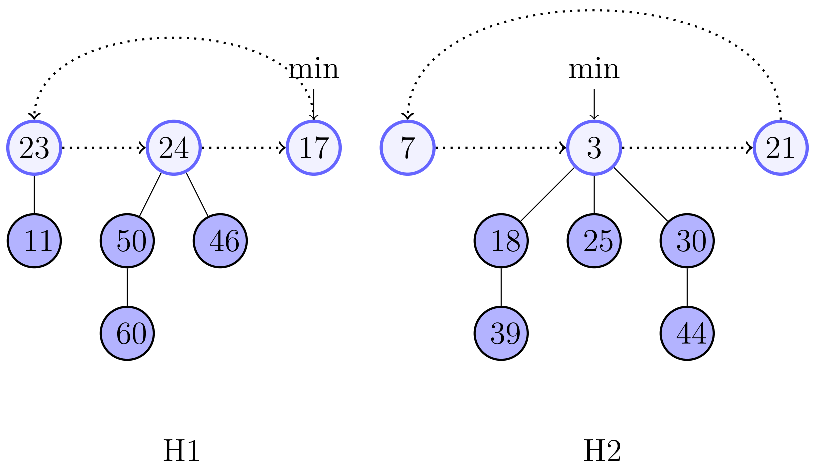

Figure 6: Two heaps H1 and H2.

- Merge the two rootlists (usually rootlists are stored in a doubly-linked list)

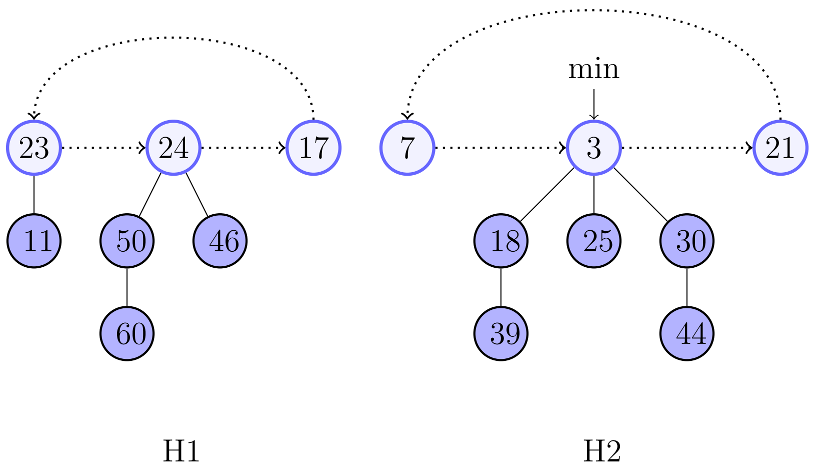

Figure 7: Root lists are merged together to form a single heap H

- Make the min pointer with smaller value the min pointer of the merged heap.

6.4 Find-min

The find-min operation in a Fibonacci heap is a simple operation that returns the minimum key value in the heap without removing it. In the basic structure of Fibonacci heap, we have a min pointer that always points to the updated minimum value of the entire heap.

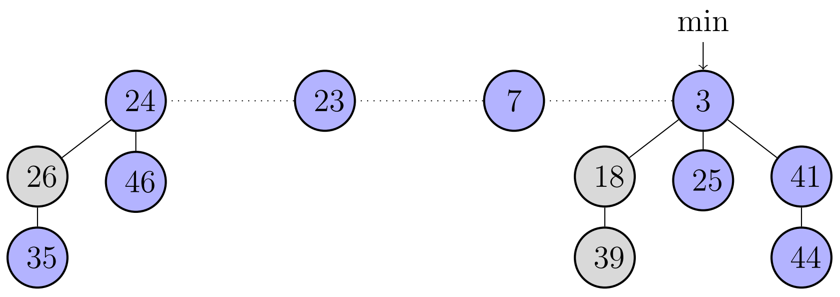

Figure 8: A Fibonacci heap with a min-pointer pointed to the minimum value

In find-min operation, we just return the minimum value pointed by the min-pointer.

6.5 Decrease-key

An amortized time complexity O(1) in case of completing decrease-key operation was the prime motivation behind the introduction of Fibonacci heap data structure. The major two obstacles of achieving this goal was

1. Decreasing a key might break the heap property
2. Preserving heap property after any operation needs an extra heapify operation that take $O(logn)$ running time

To solve these two issues, decrease-key operation of Fibonacci heap is described differently than that of other heaps. The steps generally followed are:

1. If the decreased node violates the heap order, cut the sub-tree rooted at the note and add it to the root list.
2. Since the newly added sub-tree has a minimum value as its node, no need to heapify. Problem Solved in O(1) amortized time!

Let's see a simulation of decrease-key.

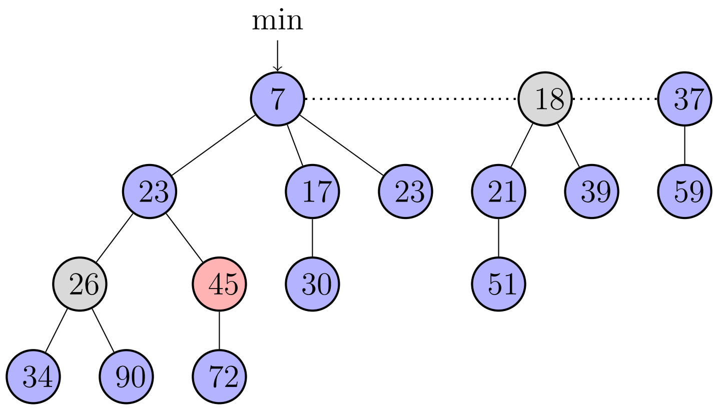

Figure 9: In this Fibonacci heap, the node 45 will be decreased with a smaller value 15

- **Decrease a Key with Smaller Value:**

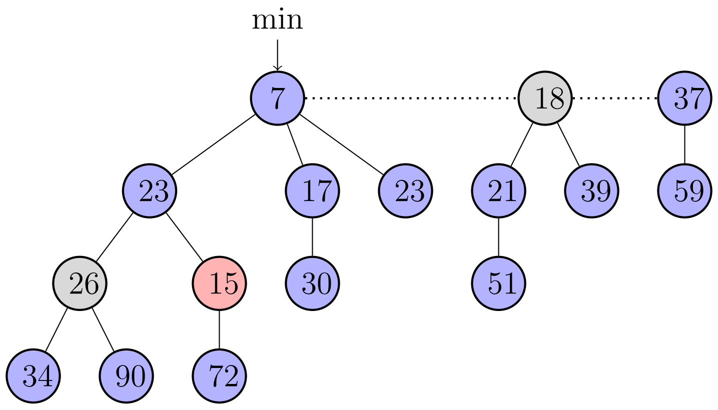

Figure 10: A violation of heap order.

- **Cut the Rooted Subtree and Add it to Root-list:**

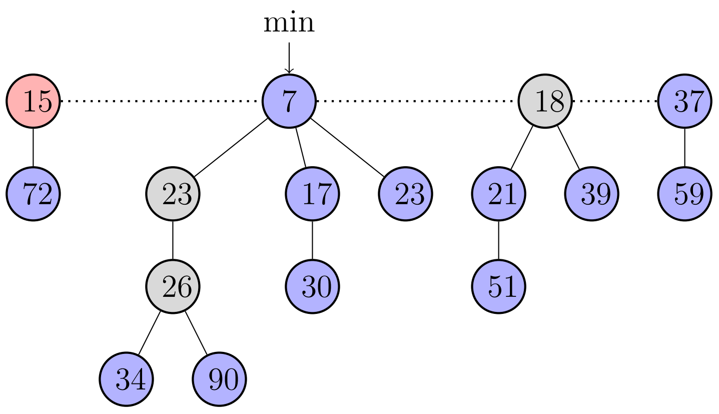

Figure 11: The subtree rooted at 15 is cut and added as a new tree to the root-list.

In this process the parent, 23 of the root, 15 of the cut subtree was marked.

6.5.1 Why parents need to be marked when a child is cut?

So that we can keep a maximum degree of a node bounded. If the degree of a node is not bounded, a node can have as many children as possible.

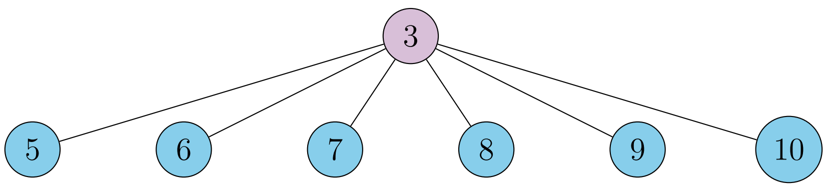

Figure 12: A problematic structure if maximum degree of a not is not bounded

In this example, root 3 has 6 children. This kind of structure would significantly impact the performance increasing it to O(n) of many operations like decrease-key, extract-min, delete etc. Cascading cut prevents these kind of structures.

- Nodes are initially unmarked.
- When a child is cut from a node, the node becomes marked.
- If a child is cut from a marked node, cut the marked node itself and do it recursively.

Let's see a simulation of cascading cut!

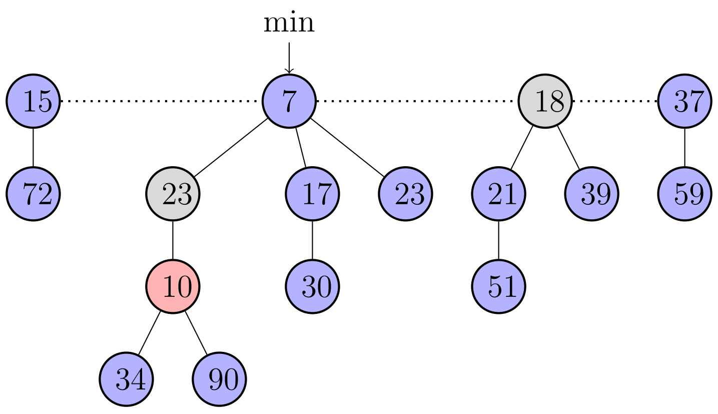

Figure 13: Cascading cut: Child cut from a marked node 23

- An already marked node 23 is losing another child. So we need to cut this node 23 as well and make it a newly added tree. This is called recursively cascading cut.

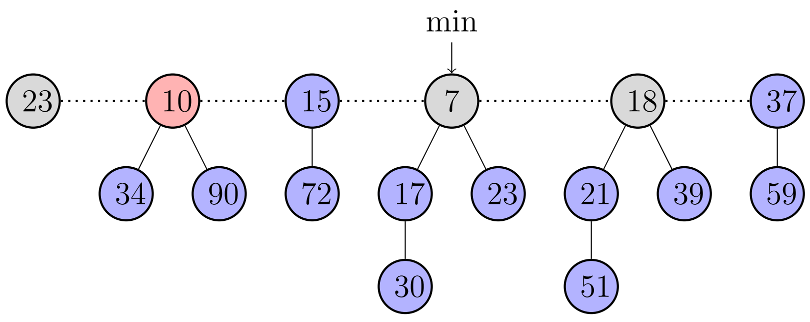

Figure 14: Cascading cut: Marked node is cut as well.

6.6 Extract-min

"Extract-min" is one of the key operations of Fibonacci heap, which removes the minimum node from the heap.As this data structure maintains "lazy union",union operation happens after extract-min.The process is-

1. **Find the Minimum Node:** Traverse the root list of the Fibonacci Heap to find the node with the minimum key. This node represents the minimum element in the heap.

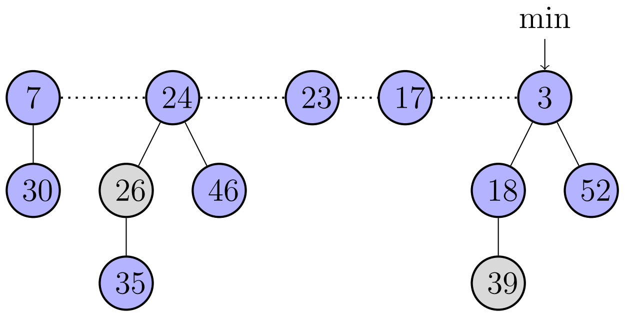

Figure 15: A Fibonacci heap with minimum node 3

2. **Remove Minimum Node from Root List:** Remove the minimum node from the root list of the heap. This involves updating the pointers of its neighboring nodes to bypass the node being removed.

3. **Merge Children with Root List:** Combine the children of the minimum node with the root list. This step involves updating the pointers of the minimum node's children to remove the parent link and then merging the resulting list with the root list.

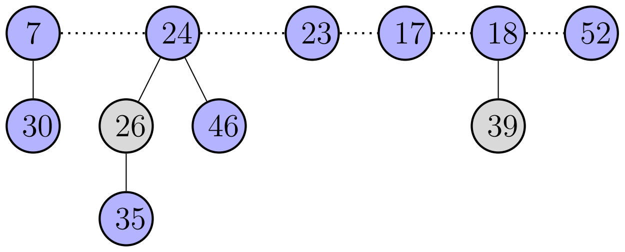

Figure 16: Removing minimum node and merging its children

4. **Consolidate Trees:** After merging the children with the root list, there might be multiple trees with the same degree (number of children) in the root list. To ensure the Fibonacci Heap maintains its structural properties, consolidate these trees by combining trees of the same degree. During this process, a table (array) is used to keep track of trees with different degrees.

5. **Update Minimum Pointer:** Traverse the consolidated root list to find the new minimum node and update the minimum pointer accordingly.

6. **Update Marked Status:** If necessary, update the marked status of nodes in the consolidated heap. A node in a Fibonacci Heap is marked when it loses a child during a decrease key operation. This information is used to maintain the amortized time complexity of the decrease key operation.

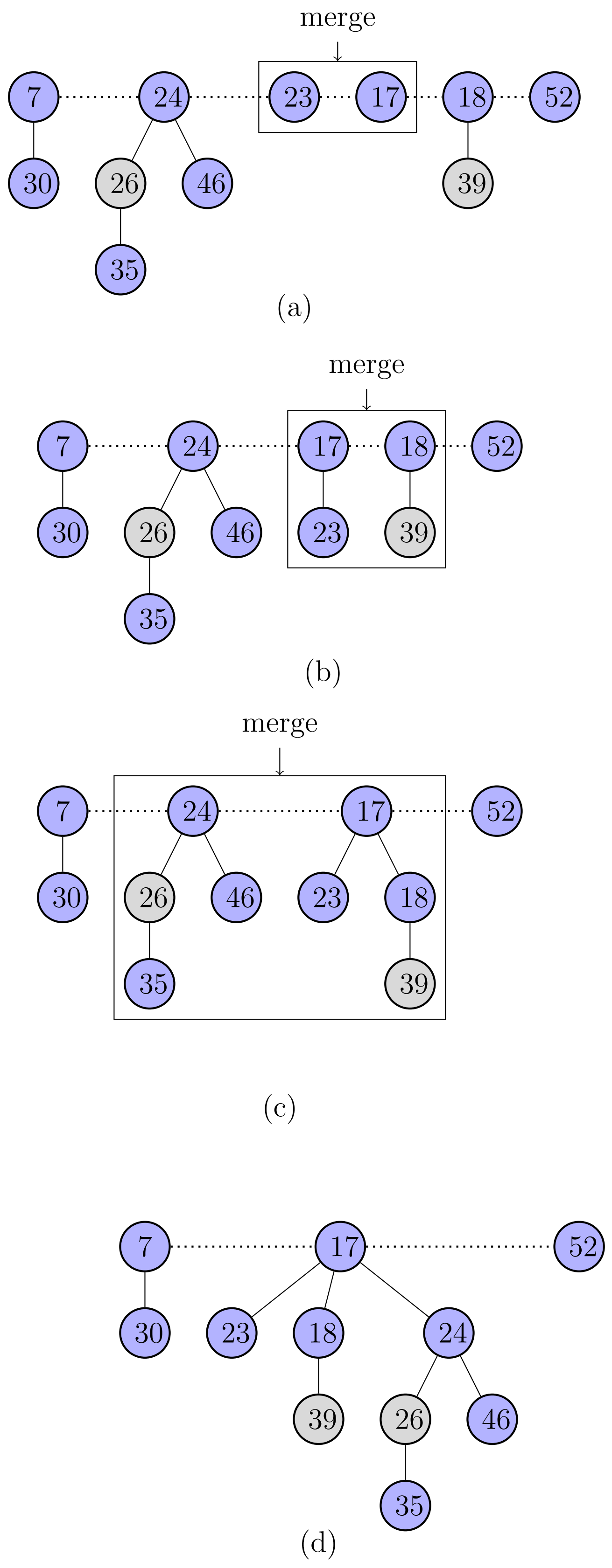

Figure 17: Consecutive Consolidations

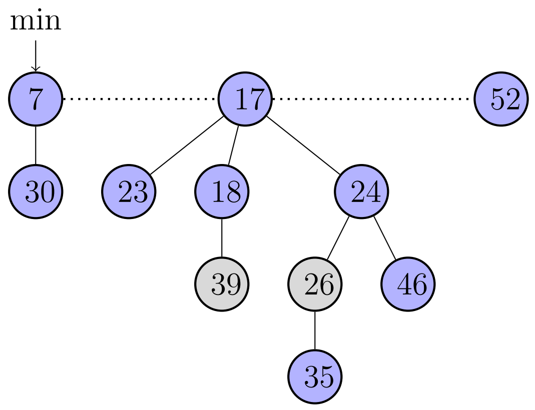

Figure 18: Final Fibonacci Heap after extracting min

In this process the amortized time complexity of extract-min is $O(logn)$. But there is a pitfall. The worst case running time of extract-min can be $O(n)$. This will happen if there is a series of insert operations without any extract-min, then the heap will be a linear data structure. Then in the first step of extract-min where we need to find the minimum node, it will take linear time complexity[6].

6.7 Delete

The delete operation of Fibonacci heap is based on the extract-min operation. When we are to delete a node x, we decrease its value so that it becomes the minimum and then we do the extract-min operation.The steps are-

1. Decrease Key of the node x to $-\infty$.
2. Extract min of the Fibonacci heap

Thus we can delete a node of Fibonacci heap. Amortized running time for delete will be $O(logn)$ because the decrease key operation will be in O(N) and extract min operation will be in $O(logn)$ time.

7 Time Complexity of Fibonacci Heap Operations

We shall use potential method to analyze the time complexity of Fibonacci heap operations. For a given Fibonacci heap H, *t(H)* denote the number of trees in the root list of H, and *m(H)* denote the number of marked nodes in H. The potential $\phi(H)$ of Fibonacci heap H is defined as

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

7.1 Make-Heap

To make an empty Fibonacci heap, the Make-Heap operation allocates and returns the Fibonacci heap object H. Since there is no element in H, *t(H)* $= 0$ and *m(H)* $= 0$. So potential $\phi(H) = 0$. The amortized cost of Make-Heap is thus equal to its actual cost, *O(1)*.

7.2 Insert

To analyze time complexity of Insert operation, let H be the heap before inserting an element x and H' be the heap after inserting it. Since a new tree is created with the new element, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is

$$\begin{aligned} \phi(H') - \phi(H) &= t(H') + 2m(H') - (t(H) + 2m(H)) \\ &= t(H) + 1 + 2m(H) - (t(H) + 2m(H) \\ &= 1. \end{aligned}$$

Since the actual cost is *O(1)*, the amortized cost is $O(1) + 1 = O(1)$.

7.3 Union

The union operation on Fibonacci heap H_1 and H_2, concatenates root lists of H_1 and H_2 and then determines the new minimum node. Let the resulting heap be H. The change in potential is

$$\begin{aligned} \phi(H) - (\phi(H_1) + \phi(H_2)) &= (t(H) + 2m(H)) \\ &= ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\ &= 0 \end{aligned}$$

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost of Union operation is equal to its actual cost $O(1)$.

7.4 Find-Min

The minimum node of a Fibonacci heap H is given by the *min pointer*, so the minimum node can be found in *O(1)* actual time. Since potential of H does not change, the amortized cost of this operation is equal to its actual cost $o(1)$.

7.5 Decrease-Key

The algorithm for decreasing the key of node x to k in heap H is as follows:

```
DECREASE-KEY($H, x, k$)
1  **if** $k > x.key$
2      **error** "new key is greater than current key"
3  $x.key = k$
4  $y = x.parent$
5  **if** $y != x$ and $x.key < y.key$
6      CUT($H, x, y$)
7      CASCADING-CUT($H, y$)
8  **if** $x.key < H.min.key$
9      $H.min = x$
```

```
CUT($H, x, y$)
1  remove $x$ from the child list of $y$, decrementing $y.rank$
2  add $x$ to the root list of $H$
3  $x.parent = NULL$
4  $x.mark = FALSE$
```

```
CASCADING-CUT($H, y$)
1  $z = y.parent$
2  **if** $z != NULL$
3      **if** $y.mark == FALSE$
4          $y.mark = TRUE$
5      **else** CUT($H, y, z$)
6          CASCADING-CUT($H, z$)
```

Now, to determine the amortized cost of the operation, first the actual cost will be determined. The Decrease-Key procedure takes $O(1)$ time, along with the time to perform the cascading cuts. Let c be the number of calls to Cascading-Cut made by a given invocation of Decrease-Key. Thus the actual cost of Decrease-Key is $O(c)$.

Let H denote the Fibonacci heap just before Decrease-Key is called. The call to Cut inside the Decrease-Key procedure creates a new tree with root x and clears x's mark bit. Each call of Cascading-Cut, except the last one, cuts a marked node and clears the mark bit. After this, The Fibonacci heap contains $t(H) + c$ trees and at most $m(h) - c + 2$ marked nodes ($c - 1$ were unmarked by cascading cuts and the last Cascading-Cut may have marked a node). The change in potential is at most

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c$$

Thus, the amortized cost of Decrease-Key is at most

$$O(c) + 4 - c = O(1)$$

7.6 Extract-Min

The algorithm for extracting the minimum element of a Fibonacci heap H is as follows:

```
Extract-Min($H$)
1   $z = H.min$
2   **if** $z != NULL$
3       **for** each child $x$ of $z$
4           add $x$ to the root list of $H$
5           $x.parent = NULL$
6       remove $z$ from the root list of $H$
7       **if** $z == z.right$
8           $H.min = NULL$
9       **else** $H.min = z.right$
10            Consolidate($H$)
11        $H.n = H.n - 1$ // H.n denotes the number of elements in heap H
12   **return** $z$
```

```
Consolidate($H$)
1   let $A[0..D(H.n)]$ be a new array
2   **for** $i = 0$ **to** $D(H.n)$
3       $A[i] = NULL$
4   **for** each node $w$ in the root list of $H$
5       $x = w$
6       $d = x.rank$
```

```
7          while $A[d] != NULL$
8             $y = A[d]$
9             if $x.key > y.key$
10                exchange $x$ with $y$
11            Merge($H, y, x$)
12            $A[d] = NULL$
13            $d = d + 1$
14         $A[d] = x$
15    $H.min = NULL$
16    for $i = 0$ $D(H.n)$
17       if $A[i] != NULL$
18          if $H.min == NULL$
19             create a root list for $H$ containing just $A[i]$
20             $H.min = A[i]$
21          else insert $A[i]$ into $H$'s root list
22             if $A[i].key < H.min.key$
23                if $H.min = A[i]$
```

```
Merge($H$, $y$, $x$)
1   remove $y$ from the root list of $H$
2   $y$ a child of $x$, incrementing $x.rank$
3   $y.mark = FALSE$
```

The procedure Consolidate uses an auxiliary array $A[0..D(H.n)]$ to keep track of roots according to their ranks. $D(H.n)$ is the upper bound on the maximum rank of H. If $A[i] = y$, then y is currently a root with $y.degree = i$. At the start of each iteration of the while loop in Consolidate, $d = x.rank$. An $O(D(n))$ time complexity comes from Extract-Min processing at most $D(n)$ children of the minimum node in line 2-3 and line 16-23 of Consolidate. For analyzing the contribution of line 4-14 of Consolidate, aggregation method will be used. The size of root list when Consolidate is called is at most $D(n) + t(H) - 1$, for original $t(H)$ root list nodes, minus the extracted root node, plus the children of the extracted node, which is at most $D(n)$. The total number of iterations of the while loop over all iterations of the for

loop is at most the number of roots in the root list. So the total work performed in the for loop is at most proportional to $D(n) + t(H)$. Thus the actual work in Extract-Min is $O(D(n) + t(H))$.
The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential after extracting is at most $(D(n) + 1) + 2m(H)$, since at most $D(n) + 1$ roots remain and no node is marked. The amortized cost is thus

$$\begin{aligned} O(D(n)+t(H)) &+ ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &= O(D(n)) + O(t(H)) - t(H) \\ &= O(D(n)) \end{aligned}$$

Now it will be shown that $D(n) = O(logn)$. So the amortized cost of extracting the minimum node is $O(logn)$.

LEMMA 1. *Let x be any node in an F-heap. Arrange the children of x in the order they were linked to x, from earliest to latest. Then the i-th child of x has a rank of at least i - 2.*
PROOF. Let y be i-th child of x, and consider the time when y was linked to x. Just before the linking, x has at least i - 1 children (some of which it may have lost after the linking). Since x and y had the same rank just before the linking, they both had a rank of at least i - 1 at that time. After the linking, the rank of y could have decreased by at most one without causing y to be cut as a child of x.

LEMMA 2. *A node of rank k in an F-heap has at least $F_{k+2} \geq \phi^k$ descendants, including itself, where F_k is the k-th Fibonacci number ($F_0 = 0, F_1 = 1, F_k = F_{k-2} + F_{k-1}$ for $K \geq 2$, and $\phi = (1 + \sqrt{5})/2$ is the golden ratio.*
PROOF. Let S_k be the minimum possible number of descendants of a node of rank k. Obviously, $S_0 = 1$, and $S_1 = 2$. Lemma 1 implies that $S_k \geq \sum_{i=0}^{k-2} S_i + 2$ for $k \geq 2$. The Fibonacci numbers satisfy $F_{k+2} = \sum_{i=2}^{k} F_i + 2$ for $k \geq 2$, from which $S_k \geq F_{k+2}$ for $k \geq 0$ follows by induction on k. The inequality $F_{k+2} \geq \phi^k$ is well known. [5]

COROLLARY 1. *The maximum rank D(n) of any node in and n-node Fibonacci heap is $O(logn)$.*

PROOF. Let x be any node in an n-node Fibonacci heap, and let $k = x.rank$. By LEMMA 1, $\phi^k \leq S_k$. $S_k \leq size(x)$ and size$(x) \leq n$. Therefor $\phi^k \leq n$. Taking base-ϕ logarithms gives $k \leq \log_\phi n$. (In fact, because k is an integer, $K \leq \lfloor \log_\phi n \rfloor$.) The maximum degree $D(n)$ of any node is thus $O(logn)$.

8 Advantages and Disadvantages

8.1 Advantages

1. **Amortized Constant-Time Operations:** Fibonacci Heaps provide amortized constant-time operations for decrease key, insert, and merge operations, ensuring efficient performance on average. This means that, even though individual operations might occasionally take longer, their average time complexity over a sequence of operations remains constant.

2. **Efficient Decrease Key Operation:** The decrease key operation in Fibonacci Heaps is particularly efficient, making it suitable for dynamic priority changes. This is crucial in algorithms where the priority of elements needs to be updated frequently.

3. **Lazy Merging:** Fibonacci Heaps support lazy merging, allowing efficient combination of multiple heaps without immediate consolidation. This laziness in merging can lead to better performance in certain dynamic programming algorithms.

4. **Support for Decrease Key Without Extract Min:** Unlike some other heap structures, Fibonacci Heaps allow the decrease key operation without requiring immediate extraction of the element. This flexibility is beneficial in scenarios where you want to update the priority without extracting the minimum element.

5. **Efficient Merge Operation:** The merge operation in Fibonacci Heaps is more efficient than in some other heap structures, facilitating efficient heap combination during algorithms. This can be advantageous when combining heaps is a frequent operation.

6. **Potential for Better Performance in Some Algorithms:** In specific algorithms like Dijkstra's for shortest paths and Prim's for minimum spanning trees, Fibonacci Heaps can offer better overall performance. This is due to their efficient decrease key and merge operations.

7. **Dynamic Structure:** Fibonacci Heaps are well-suited for algorithms involving dynamic changes, making them adaptable to evolving heap structures. This dynamic nature is beneficial in algorithms where the heap structure changes over time.[2]

8.2 Disadvantages

1. **Large Constant Factors:** Fibonacci Heaps have larger constant factors in their time and space complexities compared to simpler heap structures. This can make them less efficient in practice for small or moderately sized problem instances.[2]

2. **Complexity of Implementation:** Implementing and maintaining a Fibonacci Heap is more complex compared to simpler heap structures. This complexity can lead to more challenging debugging and maintenance.

3. **Higher Memory Overhead:** Fibonacci Heaps may have a higher memory overhead due to the additional pointers and structures required for their advanced features. This can lead to increased space consumption.[2]

4. **Lower Cache Locality:** The structure of Fibonacci Heaps, with its use of pointer-based data structures, may result in lower cache locality compared to simpler heap structures. This can impact the efficiency of memory access.

5. **Variable Performance:** While Fibonacci Heaps have favorable average-case time complexity, their performance can vary depending on factors such as the characteristics of the input data and the sequence of operations. In certain scenarios, other heap structures might perform more consistently.

6. **Limited Practical Performance Gains:** In many practical applications, the advantages of Fibonacci Heaps may not always

translate to significant performance gains. Simpler heap structures might perform adequately for certain problem instances without the added complexity of a Fibonacci Heap.

7. **Trade-off in Constant Factors:** The benefits of amortized constant-time operations come with trade-offs in constant factors, making Fibonacci Heaps less suitable for situations where minimizing overhead is critical.

8. **Difficulty in Implementation and Debugging:** Due to their complexity, implementing and debugging algorithms using Fibonacci Heaps can be more challenging compared to using simpler heap structures.

9 Application

9.1 Optimizing Dijkstra Algorithm

The Fibonacci Heap is commonly employed to enhance the efficiency of Dijkstra's algorithm. In the context of Dijkstra's algorithm, let the number of vertices in the graph G be denoted as 'n,' and the number of edges as 'm.' We make the assumption that there exists a path from the source vertex 's' to any other vertex, implying that the number of edges (m) is greater than or equal to the number of vertices minus one (m >= n-1). Dijkstra's algorithm addresses the shortest path problem by utilizing a tentative distance function 'd' that assigns real numbers to vertices, possessing properties as-

1. For any vertex v such that d(v) is finite, there is a path from s to v of length d(v).

2. When the algorithm terminates, d(v) is the distance from s to v.

At the outset, set $d(s) = 0$ and $d(v) = \infty$ for $v \neq s$. Throughout the algorithm execution, each vertex falls into one of three states: unlabeled, labeled, or scanned. Initially, the source vertex s is labeled, and all other vertices are unlabeled. The algorithm iterates through the following step until there are no labeled vertices (all vertices are scanned).
While scanning , we choose a labeled vertex v with the minimum

$d(v)$. Transition v from the labeled state to the scanned state. For every edge (v, w) such that $d(v) + Z(v, w) < d(w)$, update $d(w)$ to $d(v) + l(v, w)$ and designate w as labeled.
The non-negativity of the edge lengths ensures that once a vertex is scanned, it can never become labeled again. This property guarantees the correctness of distance computations by the algorithm.[3]
To implement Dijkstra's algorithm, we utilize a heap to store the set of labeled vertices, where the tentative distance of a vertex serves as its key. The initialization process involves one `make-heap` operation and one `insert` operation. Each scanning step necessitates one `delete-min` operation. Additionally, for each edge (v, w) such that $d(v) + l(v, w) < d(w)$, an `insert` operation is required if $d(w) = \infty$, or a `decrease-key` operation if $d(w) < \infty$. In this context, there is one `make-heap` operation, n `insert` operations, n `delete-min` operations, and at most m `decrease-key` operations. The maximum heap size is $n - 1$. If we use a Fibonacci heap (`F-heap`), the total time for heap operations is $O(n \log n + m)$, while the time for other tasks is $O(n + m)$. Consequently, Dijkstra's algorithm achieves a running time of $O(n \log n + m)$.[5]

9.2 Optimizing Minimum Spanning Tree Algorithm

Fibonacci heaps find application in computing minimum spanning trees also. For the discussion of minimum spanning tree algorithms, we will consider a connected undirected graph G with n vertices and m edges (v, w), each endowed with a nonnegative cost $c(v, w)$. In this context, a minimum spanning tree of G refers to a spanning tree with the least possible total edge cost.
A minimum spanning tree can be determined through a generalized greedy approach (Prim's algorithm). The process involves maintaining a forest defined by the edges selected thus far to form the minimum spanning tree. We start by initializing the forest to include each of the n vertices of G as a one-vertex tree. Subsequently, we iterate through the following step $n - 1$ times, progressing until there is only one n-vertex tree. For this purpose, select any tree T in the forest. Find a minimum-cost edge with exactly one endpoint in T and add it to the forest. This connects two trees to one.
For each vertex v not yet in T, we maintain a key representing the

tentative cost of connecting v to T. If $v \neq s$ and key$(v) < \infty$, we also keep track of an edge e(v) by which the connection can be made. Initially, set key$(s) = 0$ and key$(v) = w$ for $v \neq s$. Then, iterate through the following step until no vertex has a finite key. For iteration, we select a vertex v with key(v) minimum among vertices with a finite key. Replace key(v) by $-\infty$. For each edge (v, w) such that $c(v, w) <$ key(w), replace key(w) by $c(v, w)$ and define e$(w) = (v, w)$. Upon termination of this algorithm, the set of edges $e(v)$ with $v \neq s$ defines a minimum spanning tree. Setting key$(v) = -\infty$ in the connecting step serves to mark v as being in T.[1] If we store the vertices with finite key in an Fibonacci heap, the algorithm requires n `delete-min` operations and $O(m)$ other heap operations, none of which involve deletions.The running time is $O(n \log n + m)$. [5]

9.3 Optimizing Network Flow Algorithms

Fibonacci Heaps can be employed to enhance the efficiency of the scaling algorithm of Edmonds and Karp for minimum-cost network flow. The scaling algorithm improves upon basic network flow algorithms by utilizing a technique known as "scaling," where the algorithm initiates with a small scale factor and gradually increases it until the optimal solution is found.
The original time complexity of the scaling algorithm by Edmonds and Karp is $O(m^2 \cdot (\log(\Delta) + 2) \cdot n \cdot \log(N))$, where,

- m is the number of edges,
- n is the number of vertices,
- Δ is the maximum cost of an augmenting path,
- N is the maximum capacity.

The use of Fibonacci Heaps can reduce the time complexity to $O(m \cdot (n \cdot \log(n) + m) \cdot \log(N))$, assuming integer capacities.[4] Fibonacci heaps improve the efficiency of operations like `decrease key` in the context of the scaling algorithm. This is crucial in the inner loop of the algorithm where augmenting paths are repeatedly discovered and residual capacities are updated.

10 Conclusions

Summary

This report on Fibonacci Heap begins with an introduction, setting the context for exploring this data structure and its significance in computer science. It delves into the historical evolution of heap structures, starting from their introduction in the 1960s to the prominence of binary and binomial heaps in subsequent decades. This historical overview serves as a foundation for understanding the motivation behind incorporating Fibonacci Heaps into the field of computer science. The core sections of the report focus on the structure of Fibonacci Heaps, detailing the memory representation of nodes, and thoroughly examining each operation they support, including Make-Heap, Insert, Union, Find-min, Decrease-key, Extract-min, and Delete. The time complexity of these operations is scrutinized to provide insights into their efficiency. The report then discusses the advantages and disadvantages of Fibonacci Heaps, offering a balanced perspective on their strengths and potential limitations. Finally, the application section explores how Fibonacci Heaps optimize algorithms like Dijkstra's, Minimum Spanning Tree, and Network Flow algorithms, showcasing their practical utility in various computational scenarios.

References

[1] ANDTARJAN R.E CHERITON, D. Finding minimum spanning trees. *SIAM J. Comput. 5*, pages 724–742, 1976.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2022.

[3] E. W. DIJKSTRA. A note on two problems in connexion with graphs. *Numer. Math. I,*, pages 269–271, 1959.

[4] J. EDMONDS and R. M KARP. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of ACM*, pages 248–264, 1972.

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

[6] David Kempe. Lecture notes on fibonacci heaps—cs 570. 2006.