# Heaps and HeapSort

*Author: Haarit R Chavda*
*Sources: GFG, CRLS*

## 1 Binary Heaps

A **binary heap** is a complete binary tree used to store data in such a way that the minimum or maximum element can be accessed with ease. A binary heap can either be a *max-heap* or a *min-heap*.

### 1.1 Properties of a Binary Heap

A binary heap has the following properties:

1. The root element is either the minimum (in a *min-heap*) or the maximum (in a *max-heap*) element.
2. All children of a node are smaller (in a max-heap) or larger (in a min-heap) than the node itself.
3. It forms a complete binary tree, meaning all levels are fully filled except possibly the last level, which is filled from left to right.
4. The parent-child relationships can be computed using the following array indices:
   - The parent of the node at index $i$ is located at index $\lfloor \frac{i-1}{2} \rfloor$ (using integer division).
   - The left child of the node at index $i$ is located at index $2i + 1$.
   - The right child of the node at index $i$ is located at index $2i + 2$.

### 1.2 Time and Space Complexity Analysis

The following tables summarize the time complexities for various operations on heaps under different scenarios.

#### Time Complexity

| Operation | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Insert | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| Delete (Root) | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Peek (Min/Max) | $O(1)$ | $O(1)$ | $O(1)$ |
| Build Heap | $O(n)$ | $O(n)$ | $O(n)$ |
| Heapify | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

#### Space Complexity

| Aspect | Space Complexity |
|---|---|
| Heap Storage (Array) | $O(n)$ |
| Auxiliary Space | $O(1)$ |

## 2 Fibonacci Heaps

A Fibonacci heap is a data structure used for implementing priority queues. It is a type of heap data structure, but with several improvements over the traditional binary heap and binomial heap data structures.

The key advantage of a Fibonacci heap over other heap data structures is its fast amortized running time for operations such as insert, merge, and extract-min, making it one of the most efficient data structures for these operations. The running time of these operations in a Fibonacci heap is:

- $O(1)$ for insert,
- $O(\log n)$ for extract-min,
- $O(1)$ amortized for merge.

A Fibonacci heap is a collection of trees, where each tree is a heap-ordered multi-tree, meaning that each tree has a single root node with its children arranged in a heap-ordered manner. The trees in a Fibonacci heap are organized in such a way that the root node with the smallest key is always at the front of the list of trees.

In a Fibonacci heap, when a new element is inserted, it is added as a singleton tree. When two heaps are merged, the root list of one heap is simply appended to the root list of the other heap. When the extract-min operation is performed, the tree with the minimum root node is removed from the root list, and its children are added to the root list.

One unique feature of a Fibonacci heap is the use of lazy consolidation, which is a technique for improving the efficiency of the merge operation. In lazy consolidation, the merging of trees is postponed until it is necessary, rather than performed immediately. This allows for the merging of trees to be performed more efficiently in batches, rather than one at a time.

In summary, a Fibonacci heap is a highly efficient data structure for implementing priority queues, with fast amortized running times for operations such as insert, merge, and extract-min. Its use of lazy consolidation and its multi-tree structure make it a superior alternative to traditional binary and binomial heaps in many applications.

In terms of time complexity, Fibonacci Heap beats both Binary and Binomial Heap.

Like the Binomial Heap, Fibonacci Heap is a collection of trees with min-heap or max-heap properties. In Fibonacci Heap, trees can have any shape, even single nodes, unlike the Binomial Heap where every tree must be a Binomial Tree.

## Interesting Facts about Fibonacci Heap

- The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With a Binary Heap, the time complexity of these algorithms is $O(V \log V + E \log V)$. If Fibonacci Heap is used, the time complexity improves to $O(V \log V + E)$.
- Although Fibonacci Heap looks promising in terms of time complexity, it has been found slow in practice due to high hidden constants (Source: Wikipedia).
- Fibonacci Heaps are named so because Fibonacci numbers are used in the running time analysis. Every node in a Fibonacci Heap has a degree at most $O(\log n)$, and the size of a subtree rooted in a node of degree $k$ is at least $F_{k+2}$, where $F_k$ is the $k$th Fibonacci number.

### Advantages of Fibonacci Heap

- **Fast amortized running time:** The running time of operations such as insert, extract-min, and merge in a Fibonacci heap is $O(1)$ for insert, $O(\log n)$ for extract-min, and $O(1)$ amortized for merge, making it one of the most efficient data structures for these operations.
- **Lazy consolidation:** The use of lazy consolidation allows for the merging of trees to be performed more efficiently in batches, rather than one at a time, improving the efficiency of the merge operation.
- **Efficient memory usage:** Fibonacci heaps have a relatively small constant factor compared to other data structures, making them a more memory-efficient choice in some applications.

### Disadvantages of Fibonacci Heap

- **Increased complexity:** The structure and operations of a Fibonacci heap are more complex than those of a binary or binomial heap, making it a less intuitive data structure for some users.
- **Less well-known:** Compared to other data structures, Fibonacci heaps are less well-known and widely used, making it more difficult to find resources and support for implementation and optimization.

## 2.1 Time and Space Complexity Analysis

### Time Complexity

| Operation | Fibonacci Heap | Binary Heap / Binomial Heap |
|---|---|---|
| Find Min | $\Theta(1)$ | $\Theta(1)$ / $O(\log n)$ |
| Delete Min | $O(\log n)$ | $\Theta(\log n)$ |
| Insert | $\Theta(1)$ | $\Theta(\log n)$ / $\Theta(1)$ |
| Decrease-Key | $\Theta(1)$ | $\Theta(\log n)$ |
| Merge | $\Theta(1)$ | $\Theta(m \log n)$ or $\Theta(m + n)$ / $\Theta(\log n)$ |

Table 1: Amortized Time Complexities of Fibonacci Heap

**Space Complexity**

| **Aspect** | **Space Complexity** |
|---|---|
| Heap Storage (Array) | $O(n)$ |
| Auxiliary Space | $O(1)$ |

## 3 Binomial Heaps

A Binomial Heap is a collection of Binomial Trees. What is a Binomial Tree ??

### 3.1 Binomial Trees

We give the inductive definition of a binomial tree. A **Binomial Tree** has the following structure:

- (Base Case) A Binomial Tree of order 0 has exactly 1 node.
- (Inductive Case) A Binomial Tree of order $k$ can be constructed by taking two Binomial Trees of order $k-1$ and making one the leftmost child of the other.

A Binomial Tree of order $k$ has the following properties:

1. It has exactly $2^k$ nodes.
2. Its depth is $k$.
3. There are exactly $\binom{k}{i}$ nodes at depth $i$ for $i = 0, 1, \ldots, k$.
4. The root has degree $k$, and the children of the root are themselves Binomial Trees of order $k-1, k-2, \ldots, 0$ from left to right.

### 3.2 Properties of Binomial Heaps

A Binomial Heap is a set of Binomial Trees where each Binomial Tree follows the Min Heap property. And there can be at most one Binomial Tree of any degree.

### 3.3 Time and Space Complexity Analysis

**Time Complexity**

| **Operation** | **Best Case** | **Average Case** | **Worst Case** |
|---|---|---|---|
| Insert | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| Find Min | $O(1)$ | $O(1)$ | $O(1)$ |
| Delete Min | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Decrease-Key | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| Merge | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

**Space Complexity**

| **Aspect** | **Space Complexity** |
|---|---|
| Heap Storage (Array) | $O(n)$ |
| Auxiliary Space | $O(1)$ |

## 4 Comparison Among Binary, Binomial and Fibonacci

| **Operation** | **Heap Type** | | |
|---|---|---|---|
| | **Binary Heap** | **Binomial Heap** | **Fibonacci Heap** |
| Making Heap | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| Inserting a node | $\Theta(\log n)$ | $O(\log n)$ | $\Theta(1)$ |
| Finding Minimum key | $\Theta(1)$ | $O(\log n)$ | $O(1)$ |
| Extract-Minimum key | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| Union or merging | $\Theta(n)$ | $O(\log n)$ | $\Theta(1)$ |
| Decreasing a Key | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(1)$ |
| Deleting a node | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |

## 5 D-ary Heaps

A d-ary heap is like a binary heap, but (with one possible exception) nonleaf nodes have d children instead of two children. In all parts of this problem, assume that the time to maintain the mapping between objects and heap elements is O(1) per operation.

### Time Complexity Analysis

| **Operation** | **Best Case** | **Average Case** | **Worst Case** |
|---|---|---|---|
| Insert | $O(1)$ | $O(\log_d n)$ | $O(\log_d n)$ |
| Delete (Root) | $O(\log_d n)$ | $O(\log_d n)$ | $O(\log_d n)$ |
| Peek (Min/Max) | $O(1)$ | $O(1)$ | $O(1)$ |
| Build Heap | $O(n)$ | $O(n)$ | $O(n)$ |
| Heapify | $O(\log_d n)$ | $O(\log_d n)$ | $O(\log_d n)$ |

## 6 Young Tableaus

A Young tableau is a matrix with specific properties. It is an $m \times n$ matrix where:

- The numbers in each row are sorted from left to right.
- The numbers in each column are sorted from top to bottom.

Some entries in a Young tableau can be marked as $\infty$, which represent nonexistent elements. This allows a Young tableau to store up to $r$ finite numbers, where $r \leq m \times n$.

### Time Complexity Analysis

Here's the time complexity for various operations on Young tableaus:
Correct the table .......

| **Operation** | **Best Case** | **Average Case** | **Worst Case** |
|---|---|---|---|
| Insert | $O(1)$ | $O(\log_d n)$ | $O(\log_d n)$ |
| Delete (Root) | $O(\log_d n)$ | $O(\log_d n)$ | $O(\log_d n)$ |
| Peek (Min/Max) | $O(1)$ | $O(1)$ | $O(1)$ |
| Build Heap | $O(n)$ | $O(n)$ | $O(n)$ |
| Heapify | $O(\log_d n)$ | $O(\log_d n)$ | $O(\log_d n)$ |

## 7 Monotone PQ

## 8 Radix Heaps

## 9 Van Embde Boas Tress

## 10 Fredman-Willard Data Structure

## 11 Thorup's Improvement with Randomized Hashing

Thorup [436] improved the time complexity of certain operations in priority queues by introducing **randomized hashing**. The use of hashing allows for more efficient handling of certain types of priority queue operations.

Improvement: Thorup's approach reduces the time complexity for certain operations to $O(loglogn)$. This is achieved using randomization techniques, where the keys are hashed to simplify certain operations, making them faster.