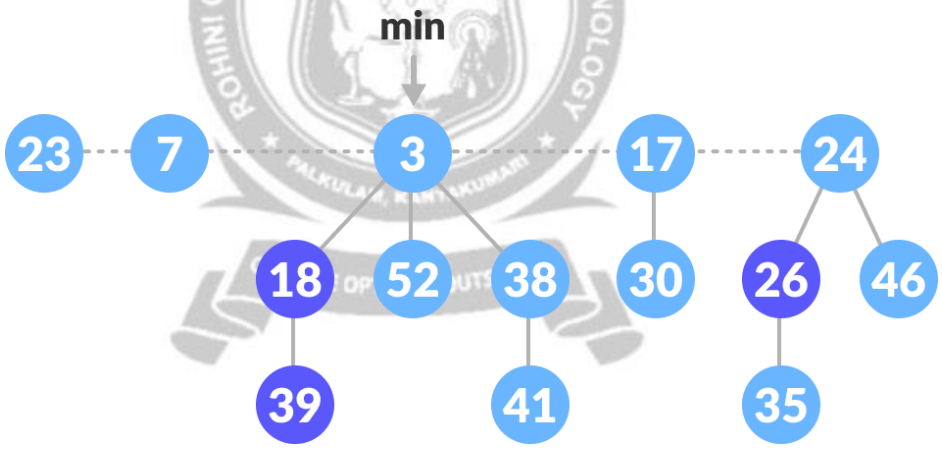
UNIT II - HIERARCHICAL DATA STRUCTURES

Binary Search Trees: Basics – Querying a Binary search tree – Insertion and Deletion- Red Black trees: Properties of Red-Black Trees – Rotations – Insertion – Deletion -B-Trees: Definition of B - trees – Basic operations on B-Trees – Deleting a key from a B-Tree- Heap – Heap Implementation – Disjoint Sets - Fibonacci Heaps: structure – Mergeable-heap operations- Decreasing a key and deleting a node-Bounding the maximum degree.

FIBONACCI HEAPS: STRUCTURE – MERGEABLE

A fibonacci heap is a data structure that consists of a collection of trees which follow min heap or max heap property. We have already discussed the min heap and max heap property in the In a fibonacci heap, a node can have more than two children or no children at all. Also, it has more efficient heap operations than that supported by the binomial and binary heaps.

The fibonacci heap is called a fibonacci heap because the trees are constructed in a way such that a tree of order n has at least Fn+2 nodes in it, where Fn+2 is the (n + 2)th Fibonacci number.



Properties of a Fibonacci Heap

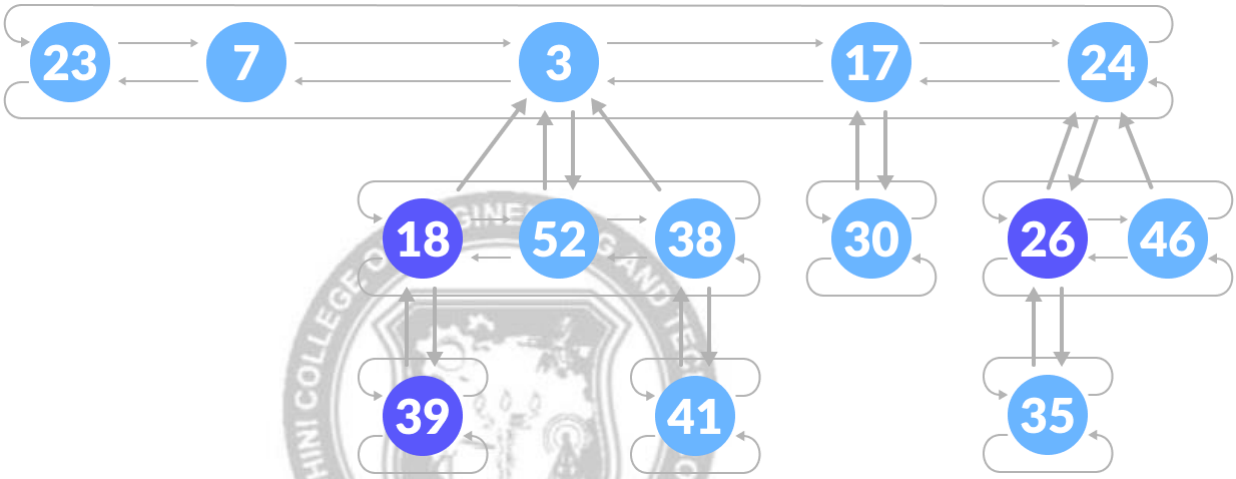
Important properties of a Fibonacci heap are:

- It is a set of min heap-ordered trees. (i.e. The parent is always smaller than the children.)
- A pointer is maintained at the minimum element node.
- It consists of a set of marked nodes. (Decrease key operation)
- The trees within a Fibonacci heap are unordered but rooted.

Memory Representation of the Nodes in a Fibonacci Heap

The roots of all the trees are linked together for faster access. The child nodes of a parent node are connected to each other through a circular doubly linked list as shown below. There are two main advantages of using a circular doubly linked list.

- Deleting a node from the tree takes O(1) time.
- The concatenation of two such lists takes O(1) time.



Fibonacci Heap Structure

Operations on a Fibonacci Heap

Insertion

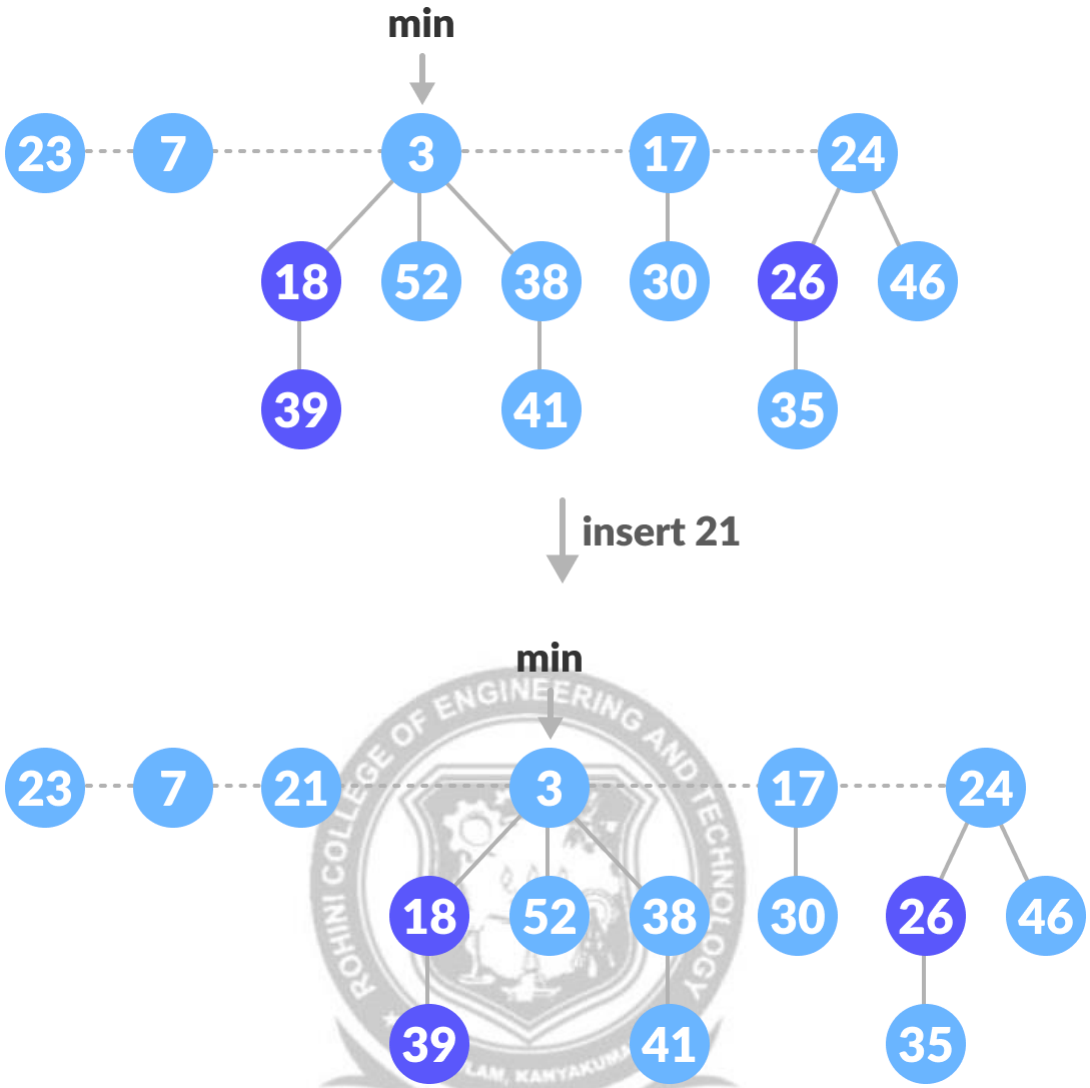
Algorithm

insert(H, x) degree[x] = 0 p[x] = NIL
child[x] = NIL left[x] = x right[x] = x mark[x] = FALSE
concatenate the root list containing x with root list H if min[H] == NIL or key[x] < key[min[H]]
then min[H] = x n[H] = n[H] + 1

Inserting a node into an already existing heap follows the steps below.

1. Create a new node for the element.
2. Check if the heap is empty.

3. If the heap is empty, set the new node as a root node and mark it min.
4. Else, insert the node into the root list and update min.



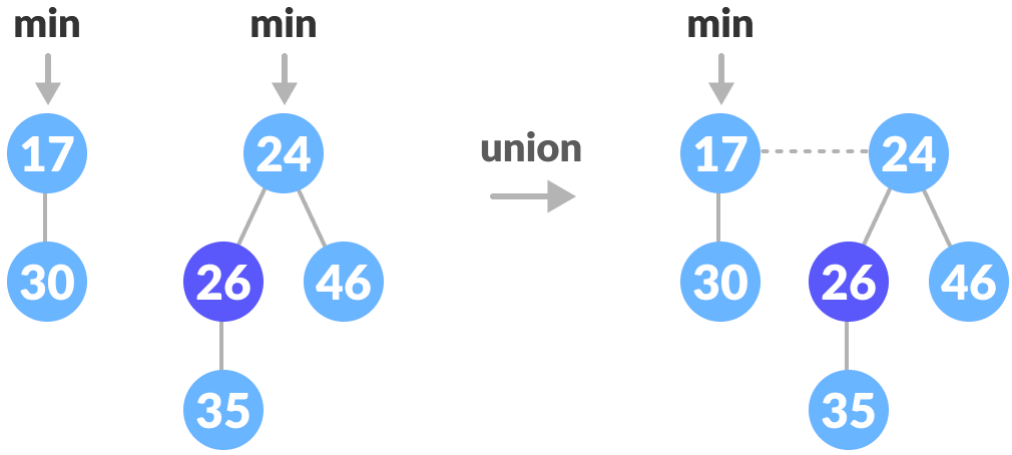
Find Min

The minimum element is always given by the min pointer.

Union

Union of two fibonacci heaps consists of the following steps.

1. Concatenate the roots of both the heaps.
2. Update min by selecting a minimum key from the new root lists.



Union of two heaps

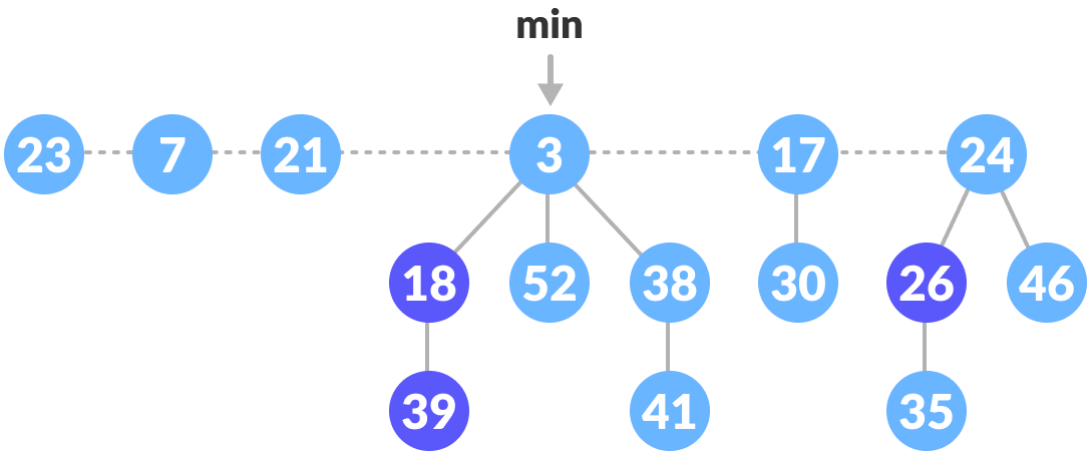
Extract Min

It is the most important operation on a fibonacci heap. In this operation, the node with minimum value is removed from the heap and the tree is re-adjusted.

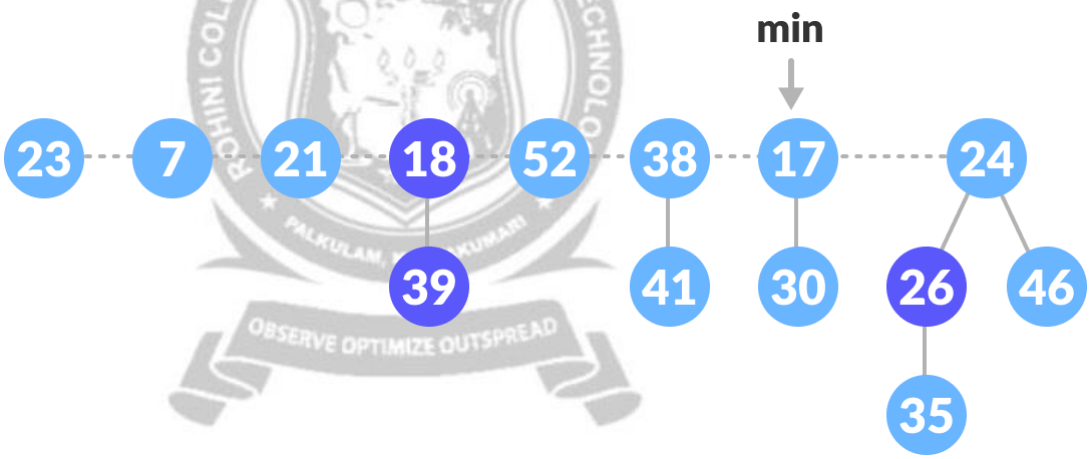
The following steps are followed:

1. Delete the min node.
2. Set the min-pointer to the next root in the root list.
3. Create an array of size equal to the maximum degree of the trees in the heap before deletion.
4. Do the following (steps 5-7) until there are no multiple roots with the same degree.
5. Map the degree of current root (min-pointer) to the degree in the array.
6. Map the degree of next root to the degree in array.
7. If there are more than two mappings for the same degree, then apply union operation to those roots such that the min-heap property is maintained (i.e. the minimum is at the root).

An implementation of the above steps can be understood in the example below. We will perform an extract-min operation on the heap below.

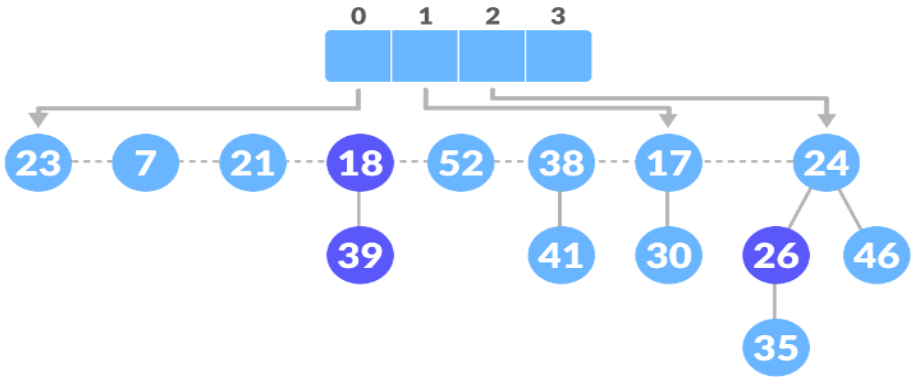


Delete the min node, add all its child nodes to the root list and set the min-pointer to the next root in the root list.



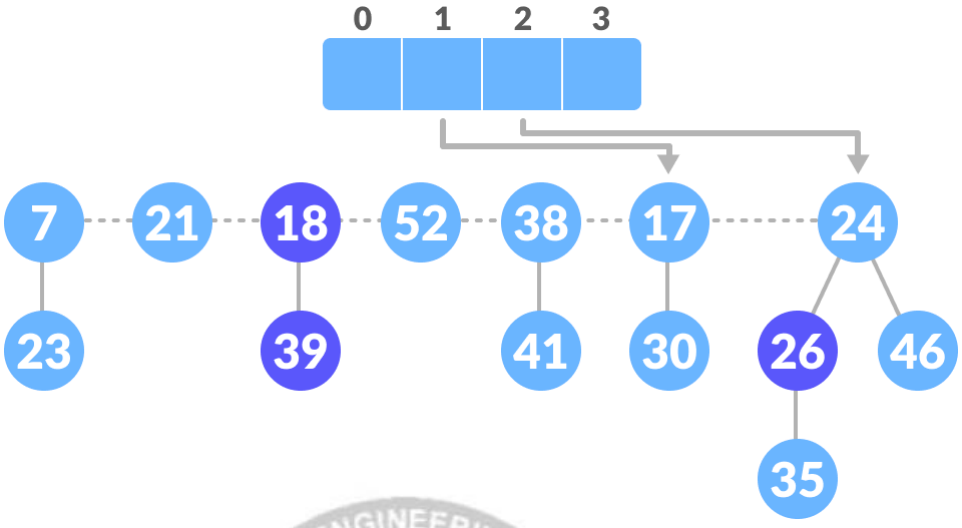
Delete the min node

The maximum degree in the tree is 3. Create an array of size 4 and map the degree of the next roots with the array.



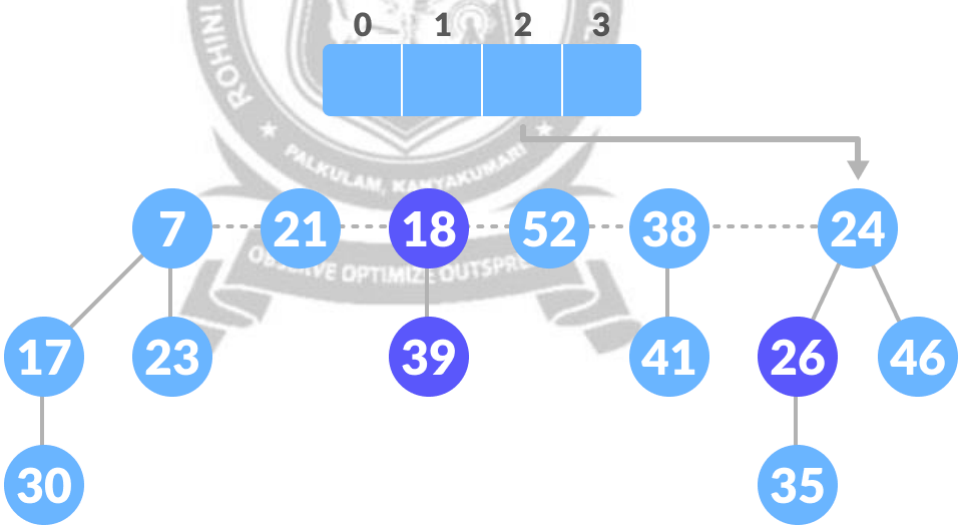
Create an array

Here, 23 and 7 have the same degrees, so unite them.



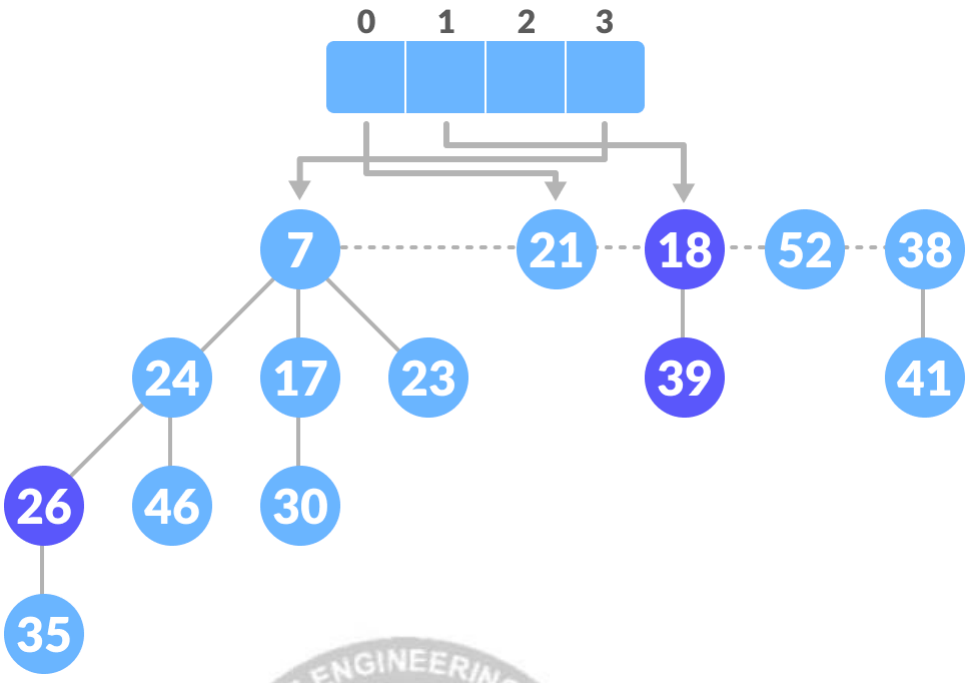
Unite those having the same degrees

Again, 7 and 17 have the same degrees, so unite them as well.



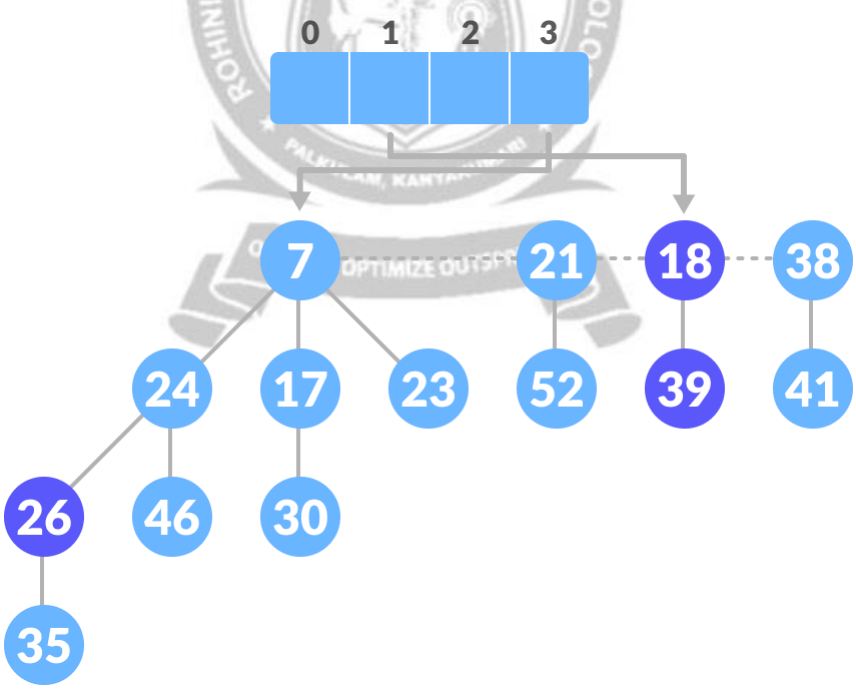
Unite those having the same degrees

Map the next nodes.



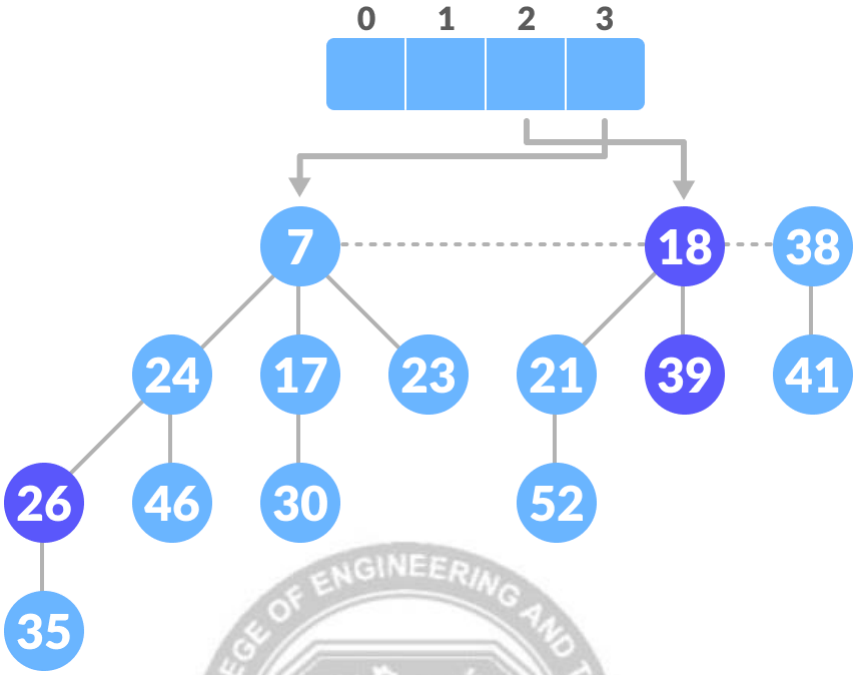
Map the remaining nodes

Again, 52 and 21 have the same degree, so unite them

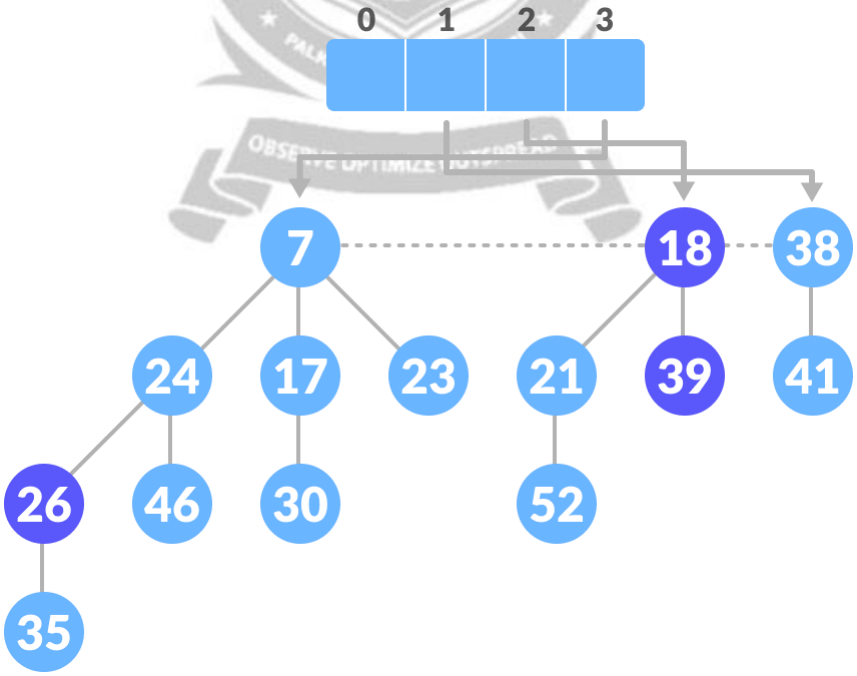


Unite those having the same degrees

Similarly, unite 21 and 18.

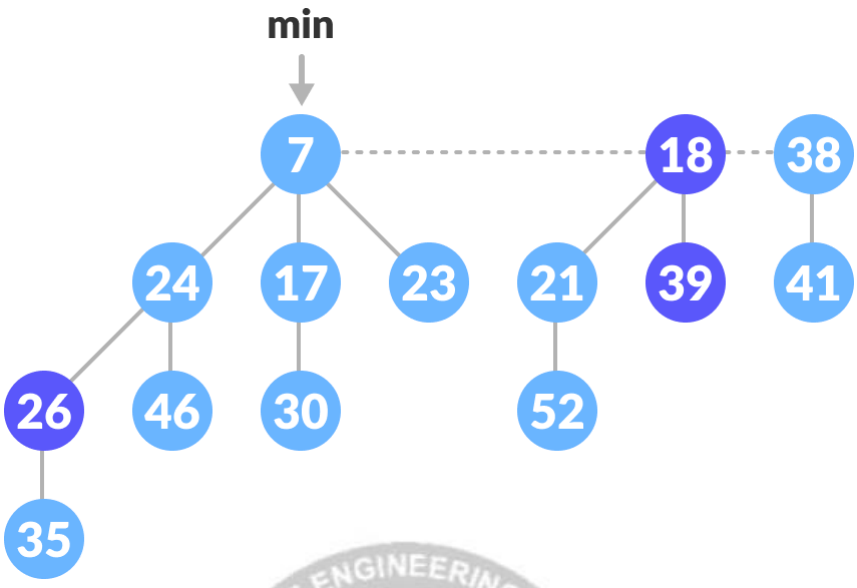


Unite those having the same degrees
Map the remaining root.



Map the remaining nodes

The final heap is.



Heap operations- Decreasing a key and deleting a node

Operations on Heaps

Heap is a very useful data structure that every programmer should know well. The heap data structure is used in Heap Sort, Priority Queues. The understanding of heaps helps us to know about memory management. In this blog, we will discuss the various operations of the heap data structure. We have already discussed what are heaps, its structure, types, and its representation in the array in the last blog. So let's get started with the operations on a heap.

Operations on Heaps

The common operation involved using heaps are:

- Heapify → Process to rearrange the heap in order to maintain heap-property.
- Find-max (or Find-min) → find a maximum item of a max-heap, or a minimum item of a min- heap, respectively.
- Insertion → Add a new item in the heap.
- Deletion → Delete an item from the heap.
- Extract Min-Max → Returning and deleting the maximum or minimum element in max-heap and min-heap respectively.

Heapify

It is a process to rearrange the elements of the heap in order to maintain the heap property. It is done when a certain node causes an imbalance in the heap due to some operation on that node.

The heapify can be done in two methodologies:

- up_heapify() → It follows the bottom-up approach. In this, we check if the nodes are following heap property by going in the direction of rootNode and if nodes are not following the heap property we do certain operations to let the tree follows the heap property.
- down_heapify() → It follows the top-down approach. In this, we check if the nodes are following heap property by going in the direction of the leaf nodes and if nodes are not following the heap property we do certain operations to let the tree follows the heap property.

Pseudo Code

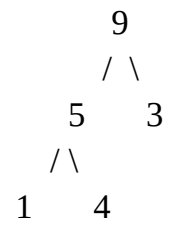
```
void down_heapify(int heap[], int parent, int size)
{
        largest = parent leftChild = 2*parent + 1
        rightChild = 2*parent + 2
        if(leftChild < size && heap[leftChild] > heap[largest])
                largest = leftChild
        if(rightChild < size && heap[rightChild] > heap[largest])
                largest = rightChild
        if(parent != largest)
        {
                swap(heap[parent], heap[largest]) down_heapify(heap,largest,size)
        }
}
void up_heapify(int heap[], int child)
{
        parent = (child-1)/2;
        if(heap[parent] < heap[child])
        {
                swap(heap[parent], heap[child]) up_heapify(heap,parent)
        }
}
```

Insertion

The insertion in the heap follows the following steps

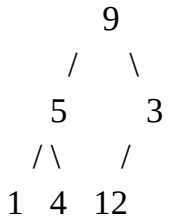
- Insert the new element at the end of the heap.
- Since the newly inserted element can distort the properties of the Heap. So, we need to perform up_heapify() operation, in order to keep the properties of the heap in a bottom-up approach.

Initially the heap is as (It follows max-heap property)

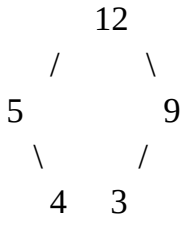


New element to be inserted is 12

Step 1: Insert the new element at the end.



Step 2: Heapify the new element following bottom-up approach.



Pseudo Code

```
void up_heapify(int heap[], int child)
{
        parent = (child-1)/2;
        if(heap[parent] < heap[child])
        {
                swap(heap[parent], heap[child]) up_heapify(heap,parent)
        }
}
void insert(int heap[],int size,int key)
{
        heap.append(key) up_heapify(heap,size+1,size)
}
```

Deletion

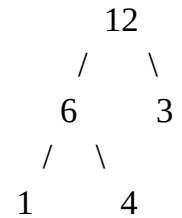
The deletion operations follow the following step:

- Replace the element to be deleted by the last element in the heap.
- Delete the last item from the heap.

- Now, the last element is placed at some position in heap, it may not follow the property of the heap, so we need to perform down_heapify() operation in order to maintain heap structure. The down_heapify() operation does the heapify in the top-bottom approach.

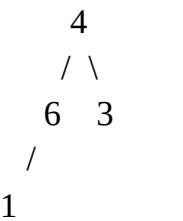
The standard deletion on Heap is to delete the element present at the root node of the heap.

Initially the heap is(It follows max-heap property)

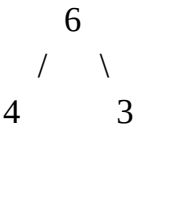


Element to be deleted is 12

Step 1: Replace the last element with root, and delete it.



Step 2: Heapify root.



Pseudo-Code

```
void down_heapify(int heap[], int parent, int size)
{
        largest = parent leftChild = 2*parent + 1
        rightChild = 2*parent + 2
        if(leftChild < size && heap[leftChild] > heap[largest])
                largest = leftChild
        if(rightChild < size && heap[rightChild] > heap[largest])
                largest = rightChild
        if(parent != largest)
        {
```

```
                swap(heap[parent], heap[largest]) down_heapify(heap,largest,size)
        }
}
void deleteRoot(int heap[], int size)
{
        swap(heap[0], heap[size-1]) // swap first and last element
        heap.pop_back(); // delete the last element down_heapify(heap,0,size-1)
}
```

Find-max (or Find-min)

The maximum element and the minimum element in the max-heap and min-heap is found at the root node of the heap.

```
int findMax(int heap[])
{
        return heap[0]
}
```

Extract Min-Max

This operation returns and deletes the maximum or minimum element in max-heap and min-heap respectively. The maximum element is found at the root node.

```
int extractMax(int heap[], int size)
{
        ans = heap[0] deleteRoot(heap, size) return ans
}
```

Bounding the maximum degree.

Bounding the maximum degree: To prove that the amortized time of FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELETE is O(lg n), we must show that the upper bound D(n) on the degree of any node of an n-node Fibonacci heap is O(lg n). The cuts that occur in FIB- HEAP-DECREASE-KEY, however, may cause trees within the Fibonacci heap to violate the unordered binomial tree properties. In this section, we shall show that because we cut a node from its parent as soon as it loses two children, D(n) is $O(\lg n)$. In particular, we shall show that $D(n) \le \lfloor \log_{\phi} n \rfloor$, where $\phi = (1+\sqrt{5})/2$.

The key to the analysis is as follows. For each node x within a Fibonacci heap, define size(x) to be the number of nodes, including x itself, in the subtree rooted at x. (Note that x need not be in the root list-it can be any node at all.) We shall show that size(x) is exponential in degree[x]. Bear in mind that degree[x] is always maintained as an accurate count of the degree of x.

Lemma 20.1

Let x be any node in a Fibonacci heap, and suppose that degree[x] = k. Let y1, y2, . . . , yk denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then, degree[y1] ≥ 0 and degree[yi] ≥ i - 2 for i = 2, 3, . . . , k.

Proof

Obviously, degree[y1] ≥ 0.

For i ≥ 2, we note that when yi was linked to x, all of y1, y2, . . . , yi-1 were children of x, so we must have had degree[x] = i - 1. Node yi is linked to x only if degree[x] = degree[yi], so we must have also had degree[yi] = i - 1 at that time. Since then, node yi has lost at most one child, since it would have been cut from x if it had lost two children. We conclude that degree[yi] ≥ i - 2.

We finally come to the part of the analysis that explains the name "Fibonacci heaps." Recall from Standard notations and common functions that for k = 0, 1, 2, . . . , the kth Fibonacci number is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k = 0, \\ 1 & \text{if } k = 1, \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2. \end{cases}$$

The following lemma gives another way to express Fk.

Lemma 20.2

For all integers k ≥ 0,

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

Proof

The proof is by induction on k. When k = 0,

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2 . \end{aligned}$$

We now assume the inductive hypothesis that $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, and we have

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i . \end{aligned}$$

The following lemma and its corollary complete the analysis. They use the in-equality Fk 2 ≥ φk