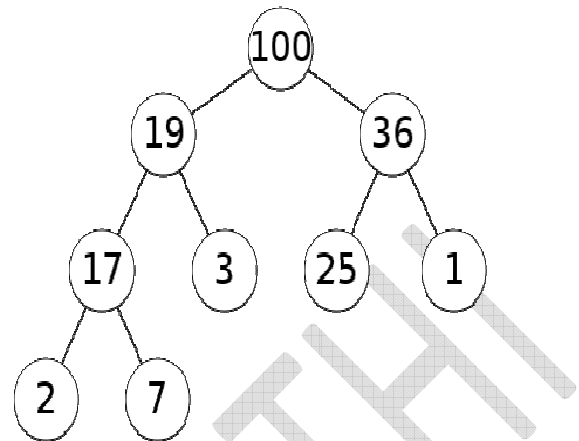


# Heap (data structure)

Example of a complete binary max-heap

In computer science, a **heap** is a specialized tree-based data structure that satisfies the *heap property*: If  $A$  is a parent node of  $B$  then  $\text{key}(A)$  is ordered with respect to  $\text{key}(B)$  with the same ordering applying across the heap. In a *max heap* the keys of parent nodes are always greater than or equal to those of the children and the highest key exists in the root node. In a *min heap* the keys of parent nodes are always less than or equal to those of the children and the root node has the lowest key.



Note that, as shown in the graphic, there is no implied ordering between siblings or cousins and no implied sequence for an in-order traversal (as there would be in, e.g., a binary search tree). The heap relation mentioned above applies only between nodes and their immediate parents.

The maximum number of children each node can have depends on the type of heap, but in many types it is at most two. The heap is one maximally efficient implementation of an abstract data type called a priority queue. Heaps are crucial in several efficient graph algorithms such as Dijkstra's algorithm, and in the sorting algorithm heapsort. A *heap* data structure should not be confused with *the heap* which is a common name for dynamically allocated memory. The term was originally used only for the data structure.

## Implementation and operations

Heaps are usually implemented in an array, and do not require pointers between elements.

The operations commonly performed with a heap are:

- *create-heap*: create an empty heap
- *find-max* or *find-min*: find the maximum item of a max-heap or a minimum item of a min-heap, respectively
- *delete-max* or *delete-min*: removing the root node of a max- or min-heap, respectively
- *increase-key* or *decrease-key*: updating a key within a max- or min-heap, respectively
- *insert*: adding a new key to the heap
- *merge*: joining two heaps to form a valid new heap containing all the elements of both.

Different types of heaps implement the operations in different ways, but notably, insertion is often done by adding the new element at the end of the heap in the first available free space. This will tend to violate the heap property, and so the elements are then reordered until the heap property has been re-established.

## Variants

- 2-3 heap
- Beap
- Binary heap
- **Binomial heap**
- Brodal queue
- D-ary heap
- **Fibonacci heap**
- **Leftist heap**
- **Lazy Binomial heap**
- **Min-Max heap**
- Pairing heap

- **Skew heap**
- Soft heap
- Weak heap
- Leaf heap
- Radix heap
- Randomized meldable heap

### Comparison of theoretic bounds for variants

The following time complexities are amortized (worst-time) time complexity for entries marked by an asterisk, and regular worst case time complexities for all other entries.  $O(f)$  gives asymptotic upper bound and  $\Theta(f)$  is asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	Binary	Binomial	Fibonacci	Pairing	Brodal***	RP
find-min	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$O(1)^*$	$O(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)$	$O(\log n)^*$
insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	$O(1)^*$	$O(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$O(\log n)^*$	$O(1)$	$\Theta(1)^*$
merge	$\Theta(n)$	$O(\log n)^{**}$	$\Theta(1)$	$O(1)^*$	$O(1)$	$\Theta(1)$

(\*)Amortized time

(\*\*)Where  $n$  is the size of the larger heap

(\*\*\*)Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with  $n$  elements can be constructed bottom-up in  $O(n)$ .

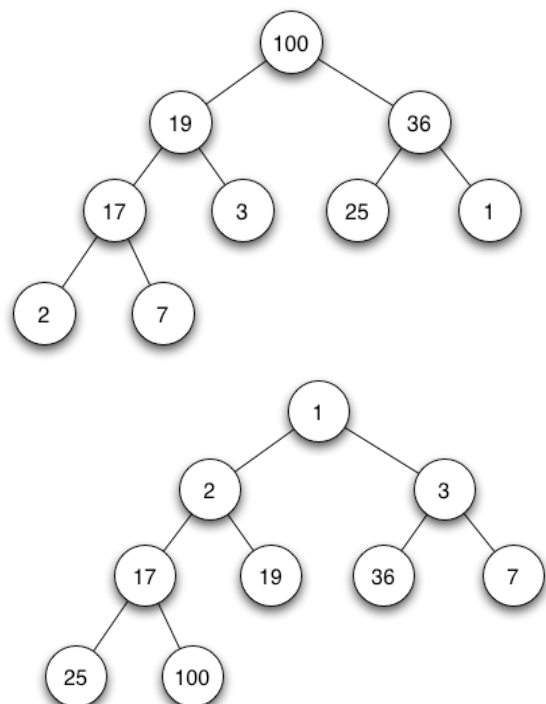
## Binary heap

Binary Heap		
Type Tree		
Time complexity in big O notation		
	Average	Worst case
Space	$O(n)$	$O(n)$
Search	N/A Operation	N/A Operation
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

A **binary heap** is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:

- The *shape property*: the tree is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.
- The *heap property*: each node is greater than or equal to each of its children according to a comparison predicate defined for the data structure.

Heaps with a mathematical "greater than or equal to" comparison function are called *max-heaps*; those with a



mathematical "less than or equal to" comparison function are called *min-heaps*. Min-heaps are often used to implement priority queues.

Since the ordering of siblings in a heap is not specified by the heap property, a single node's two children can be freely interchanged unless doing so violates the shape property (compare with treap). The binary heap is a special case of the d-ary heap in which  $d = 2$ .

### Heap operations

Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take  $O(\log n)$  time.

### Heap operations

Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take  $O(\log n)$  time.

### Insert

To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle up*, *heapify-up*, or *cascade-up*), by following this algorithm:

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a time complexity of  $O(\log n)$ .

### Delete

The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, *cascade-down* and *extract-min/max*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

The downward-moving node is swapped with the *larger* of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position. This functionality is achieved by the **Max-Heapify** function as defined below in pseudocode for an array-backed heap *A*. Note that "*A*" is indexed starting at 1, not 0 as is common in many programming languages.

For the following algorithm to correctly re-heapify the array, the node at index *i* and its two direct children must violate the heap property. If they do not, the algorithm will fall through with no change to the array.

```
Max-Heapify (A, i):  
  left  $\leftarrow 2i$   
  right  $\leftarrow 2i + 1$   
  largest  $\leftarrow i$   
  if left  $\leq$  heap_length[A] and A[left] > A[largest] then:  
    largest  $\leftarrow$  left  
  if right  $\leq$  heap_length[A] and A[right] > A[largest] then:  
    largest  $\leftarrow$  right  
  if largest  $\neq$  i then:  
    swap A[i]  $\leftrightarrow$  A[largest]  
    Max-Heapify(A, largest)
```

The down-heap operation (without the preceding swap) can also be used to modify the value of the root, even when an element is not being deleted.

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or  $O(\log n)$ .

### Building a heap

A heap could be built by successive insertions. This approach requires  $O(n \log n)$  time because each insertion takes  $O(\log n)$  time and there are  $n$  elements. However this is not the optimal method. The optimal method starts by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be represented by an

array, see below). Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored. More specifically if all the subtrees starting at some height (measured from the bottom) have already been "heapified", the trees at height  $h+1$  can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap. This process takes  $O(h)$  operations (swaps) per node. In this method most of the heapification takes place in the lower levels. Since the height of the heap is  $\lceil \log n \rceil$ , the number of nodes at

height  $h$  is  $\leq \left\lceil \frac{2^{(\lg n - h) - 1}}{2^{h+1}} \right\rceil = \left\lceil \frac{2^{\lg n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$ . Therefore, the cost of heapifying all subtrees is:

$$\begin{aligned} \sum_{h=0}^{\lceil \lg n \rceil} \frac{n}{2^{h+1}} O(h) &= O \left( n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^{h+1}} \right) \\ &\leq O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) \end{aligned}$$

This uses the fact that the given infinite series  $\sum h / 2^h$  converges to 2.

The **Build-Max-Heap** function that follows, converts an array  $A$  which stores a complete binary tree with  $n$  nodes to a max-heap by repeatedly using **Max-Heapify** in a bottom up manner. It is based on the observation that the array elements indexed by  $\text{floor}(n/2) + 1, \text{floor}(n/2) + 2, \dots, n$  are all leaves for the tree, thus each is a one-element heap.

**Build-Max-Heap** runs **Max-Heapify** on each of the remaining tree nodes.

**Build-Max-Heap**[3] ( $A$ ):

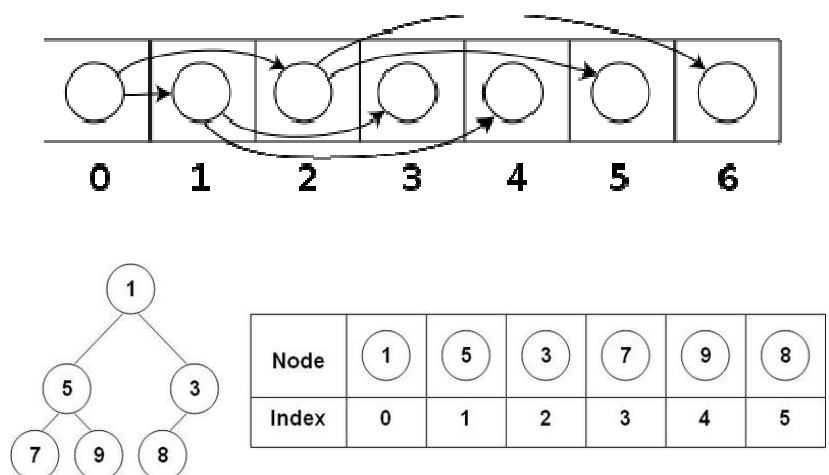
$\text{heap\_length}[A] \leftarrow \text{length}[A]$

**for**  $i \leftarrow \text{floor}(\text{length}[A]/2)$  **downto** 1 **do**

**Max-Heapify**( $A, i$ )

### Heap implementation

Heaps are commonly implemented with an array. Any binary tree can be stored in an array, but because a heap is always an almost complete binary tree, it can be stored compactly. No space is required for pointers; instead, the parent and children of each node can be found by arithmetic on array indices. These properties make this heap implementation a simple example



of an implicit data structure or

Ahnentafel list. Details depend on the root position, which in turn may depend on constraints of a programming language used for implementation, or programmer preference. Specifically, sometimes the root is placed at index 1, wasting space in order to simplify arithmetic.

Let  $n$  be the number of elements in the heap and  $i$  be an arbitrary valid index of the array storing the heap. If the tree root is at index 0, with valid indices 0 through  $n-1$ , then each element  $a[i]$  has

- children  $a[2i+1]$  and  $a[2i+2]$
- parent  $a[\text{floor}((i-1)/2)]$

Alternatively, if the tree root is at index 1, with valid indices 1 through  $n$ , then each element  $a[i]$  has

- children  $a[2i]$  and  $a[2i+1]$
- parent  $a[\text{floor}(i/2)]$ .

This implementation is used in the heapsort algorithm, where it allows the space in the input array to be reused to store the heap (i.e. the algorithm is done in-place). The implementation is also useful for use as a Priority queue where use of a dynamic array allows insertion of an unbounded number of items.

#### Derivation of children's index in an array implementation

This derivation will show how for any given node (starts from zero), its children would be found at and

#### Mathematical proof

From the figure in "Heap Implementation" section, it can be seen that any node can store its children only after its right siblings and its left siblings' children have been stored. This fact will be used for derivation.

Total number of elements from root to any given level  $l = 2^{l+1} - 1$ , where starts at zero.

Suppose the node is at level  $l$ .

So, the total number of nodes from root to previous level would be  $= 2^{(l-1)+1} - 1 = 2^l - 1$

Total number of nodes stored in the array till the index  $i = i + 1$  (Counting too)

So, total number of siblings on the left of  $i$  is

$=$  Number of nodes including  $i$   $-$  Number of nodes through the previous level  $-$  One node for  $i$  itself

$$= (i + 1) - (2^l - 1) - 1$$

$$= i + 1 - 2^l + 1 - 1$$

$$= i - 2^l + 1$$

Hence, total number of children of these siblings  $= 2(i - 2^l + 1)$

Number of elements at any given level  $L = 2^L$

So, total siblings to right of  $i$  is:-

$$= \text{Total nodes in level } l - (\text{Total siblings on left} + 1)$$

$$= (2^l) - (i - 2^l + 2)$$

$$= 2^l + 2^l - i - 2$$

$$= 2^{l+1} - i - 2$$

So, index of 1st child of node would be:-

$$\begin{aligned}
 &= i + \text{Total siblings on right} + 2 * \text{Total siblings on left} + 1 \\
 &= i + (2^{l+1} - i - 2) + 2(i - 2^l + 1) + 1 \\
 &= i + 2^{l+1} - i - 2 + 2i - 2^{l+1} + 2 + 1 \\
 &= i - i + 2i + 2^{l+1} - 2^{l+1} - 2 + 2 + 1 \\
 &= 2i + 1
 \end{aligned}$$

**From: Geeksforgeeks**

## Binary Heap

A Binary Heap is a Binary Tree with following properties.

1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

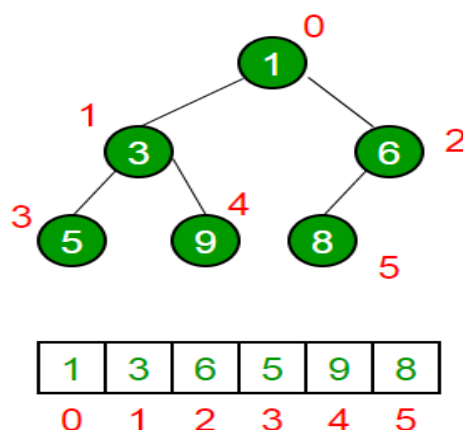
### How is Binary Heap represented?

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the  $i^{\text{th}}$  node, i.e., Arr[i]:

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node

The traversal method use to achieve Array representation is **Level Order**



### Applications of Heaps:

- 1) **Heap Sort**: Heap Sort uses Binary Heap to sort an array in  $O(n \log n)$  time.
- 2) **Priority Queue**: Priority queues can be efficiently implemented using Binary Heap because it supports `insert()`, `delete()` and `extractmax()`, `decreaseKey()` operations in  $O(\log n)$  time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
- 3) **Graph Algorithms**: The priority queues are especially used in Graph Algorithms like **Dijkstra's Shortest Path** and **Prim's Minimum Spanning Tree**.
- 4) Many problems can be efficiently solved using Heaps. See following for example.
  - a) **K'th Largest Element in an array**.
  - b) **Sort an almost sorted array/**
  - c) **Merge K Sorted Arrays**.

### Operations on Min Heap:

- 1) `getMini()`: It returns the root element of Min Heap. Time Complexity of this operation is  $O(1)$ .
- 2) `extractMin()`: Removes the minimum element from MinHeap. Time Complexity of this Operation is  $O(\log n)$  as this operation needs to maintain the heap property (by calling `heapify()`) after removing root.
- 3) `decreaseKey()`: Decreases value of key. The time complexity of this operation is  $O(\log n)$ . If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- 4) `insert()`: Inserting a new key takes  $O(\log n)$  time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- 5) `delete()`: Deleting a key also takes  $O(\log n)$  time. We replace the key to be deleted with minus infinite by calling `decreaseKey()`. After `decreaseKey()`, the minus infinite value must reach root, so we call `extractMin()` to remove the key.

Below is the implementation of basic heap operations.

### Python Code

```
# A Python program to demonstrate common binary heap operations
# Import the heap functions from python library
from heapq import heappush, heappop, heapify

# heappop - pop and return the smallest element from heap
```

```
# heappush - push the value item onto the heap, maintaining
# heap invariant
# heapify - transform list into heap, in place, in linear time
```

```
# A class for Min Heap
```

```
class MinHeap:
```

```
    # Constructor to initialize a heap
```

```
    def __init__(self):
```

```
        self.heap = []
```

```
    def parent(self, i):
```

```
        return (i-1)/2
```

```
    # Inserts a new key 'k'
```

```
    def insertKey(self, k):
```

```
        heappush(self.heap, k)
```

```
    # Decrease value of key at index 'i' to new_val
```

```
    # It is assumed that new_val is smaller than heap[i]
```

```
    def decreaseKey(self, i, new_val):
```

```
        self.heap[i] = new_val
```

```
        while(i != 0 and self.heap[self.parent(i)] > self.heap[i]):
```

```
            # Swap heap[i] with heap[parent(i)]
```

```
            self.heap[i], self.heap[self.parent(i)] = (
```

```
                self.heap[self.parent(i)], self.heap[i])
```

```
    # Method to remove minimum element from min heap
```

```
    def extractMin(self):
```

```
        return heappop(self.heap)
```

```
    # This function deletes key at index i. It first reduces
```

```
    # value to minus infinite and then calls extractMin()
```

```
    def deleteKey(self, i):
```

```
        self.decreaseKey(i, float("-inf"))
```

```
        self.extractMin()
```

```
    # Get the minimum element from the heap
```

```
    def getMin(self):
```

```
        return self.heap[0]
```

```
# Driver program to test above function
```

```
heapObj = MinHeap()
```

```
heapObj.insertKey(3)
```

```
heapObj.insertKey(2)
```

```
heapObj.deleteKey(1)
```

```
heapObj.insertKey(15)
```

```
heapObj.insertKey(5)
```



```
heapObj.insertKey(4)
heapObj.insertKey(45)
```

```
print heapObj.extractMin(),
print heapObj.getMin(),
heapObj.decreaseKey(2, 1)
print heapObj.getMin()
```

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

### **C++ Code**

```
// A C++ program to demonstrate common Binary Heap Operations
```

```
#include<iostream>
```

```
#include<climits>
```

```
using namespace std;
```

```
// Prototype of a utility function to swap two integers
```

```
void swap(int *x, int *y);
```

```
// A class for Min Heap
```

```
class MinHeap
```

```
{
```

```
    int *harr; // pointer to array of elements in heap
```

```
    int capacity; // maximum possible size of min heap
```

```
    int heap_size; // Current number of elements in min heap
```

```
public:
```

```
    // Constructor
```

```
    MinHeap(int capacity);
```

```
    // to heapify a subtree with the root at given index
```

```
    void MinHeapify(int i);
```

```
    int parent(int i) { return (i-1)/2; }
```

```
    // to get index of left child of node at index i
```

```
    int left(int i) { return (2*i + 1); }
```

```
    // to get index of right child of node at index i
```

```
    int right(int i) { return (2*i + 2); }
```

```
    // to extract the root which is the minimum element
```

```
    int extractMin();
```

```
    // Decreases key value of key at index i to new_val
```

```
    void decreaseKey(int i, int new_val);
```

```
    // Returns the minimum key (key at root) from min heap
```

```
    int getMin() { return harr[0]; }
```

```

// Deletes a key stored at index i
void deleteKey(int i);

// Inserts a new key 'k'
void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

// Decreases value of key at index 'i' to new_val. It is assumed that
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] > harr[i])
    {
        swap(&harr[i], &harr[parent(i)]);
        i = parent(i);
    }
}

```

```
}
```

```
// Method to remove minimum element (or root) from min heap
```

```
int MinHeap::extractMin()
```

```
{
```

```
    if (heap_size <= 0)
```

```
        return INT_MAX;
```

```
    if (heap_size == 1)
```

```
    {
```

```
        heap_size--;
```

```
        return harr[0];
```

```
    }
```

```
    // Store the minimum value, and remove it from heap
```

```
    int root = harr[0];
```

```
    harr[0] = harr[heap_size-1];
```

```
    heap_size--;
```

```
    MinHeapify(0);
```

```
    return root;
```

```
}
```

```
// This function deletes key at index i. It first reduced value to minus
```

```
// infinite, then calls extractMin()
```

```
void MinHeap::deleteKey(int i)
```

```
{
```

```
    decreaseKey(i, INT_MIN);
```

```
    extractMin();
```

```
}
```

```
// A recursive method to heapify a subtree with the root at given index
```

```
// This method assumes that the subtrees are already heapified
```

```
void MinHeap::MinHeapify(int i)
```

```
{
```

```
    int l = left(i);
```

```
    int r = right(i);
```

```
    int smallest = i;
```

```
    if (l < heap_size && harr[l] < harr[i])
```

```
        smallest = l;
```

```
    if (r < heap_size && harr[r] < harr[smallest])
```

```
        smallest = r;
```

```
    if (smallest != i)
```

```
    {
```

```
        swap(&harr[i], &harr[smallest]);
```

```
        MinHeapify(smallest);
```

```
    }
```

```
}
```

```
// A utility function to swap two elements
```

```
void swap(int *x, int *y)
```

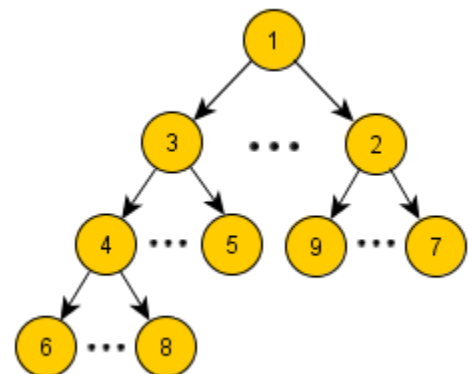
```
{  
    int temp = *x;  
    *x = *y;  
    *y = temp;  
}
```

```
// Driver program to test above functions
```

```
int main()  
{  
    MinHeap h(11);  
    h.insertKey(3);  
    h.insertKey(2);  
    h.deleteKey(1);  
    h.insertKey(15);  
    h.insertKey(5);  
    h.insertKey(4);  
    h.insertKey(45);  
    cout << h.extractMin() << " ";  
    cout << h.getMin() << " ";  
    h.decreaseKey(2, 1);  
    cout << h.getMin();  
    return 0;  
}
```

## d-ary heap

D-ary heap is a complete d-ary tree filled in left to right manner, in which holds, that every parent node has a higher (or equal value) than all of its descendands. Heap respecting this ordering is called max-heap, because the node with the maximal value is on the top of the tree. Analogously min-heap is a heap, in which every parent node has a lower (or equal) value than all of its descendands.



Thanks to these properties, d-ary heap behaves as a priority queue. Special case of d-ary heap (d=2) is binary heap.

## Implementation

D-ary heap is usually implemented using array (let's suppose it is indexed starting at 0). Than for every node of the heap placed at index  $n$  holds, that its parent is placed at index  $(n - 1)/d$  and its descendands are placed at indexes  $d \cdot k + 1, \dots, d \cdot k + d$ . It is also convenient, if the heap arity is a power of 2, because than we can easily replace multiplications used in the tree traversal by binary shifts.

## Problem Solution

1. Create a class D\_aryHeap with instance variables items set to an empty list and d. The list items is used to store the d-ary heap while d represents the number of children each node can have in the heap.
2. Define methods size, parent, child, get, get\_max, extract\_max, max\_heapify, swap and insert.
3. The method size returns the number of elements in the heap.
4. The method parent takes an index as argument and returns the index of the parent.
5. The method child takes an index and position as arguments and returns the index of the child at that position from the left.
6. The method get takes an index as argument and returns the key at the index.
7. The method get\_max returns the maximum element in the heap by returning the first element in the list items.
8. The method extract\_max returns the the maximum element in the heap and removes it.
9. The method max\_heapify takes an index as argument and modifies the heap structure at and below the node at this index to make it satisfy the heap property.
10. The method swap takes two indexes as arguments and swaps the corresponding elements in the heap.
11. The method insert takes a key as argument and adds that key to the heap.

## Program/Source Code

Here is the source code of a Python program to implement a d-ary heap. The program output is shown below.

```
class D_aryHeap:
    def __init__(self, d):
        self.items = []
        self.d = d

    def size(self):
        return len(self.items)

    def parent(self, i):
        return (i - 1) // self.d

    def child(self, index, position):
        return index * self.d + (position + 1)

    def get(self, i):
        return self.items[i]

    def get_max(self):
        if self.size() == 0:
            return None
        return self.items[0]

    def extract_max(self):
        if self.size() == 0:
            return None
        largest = self.get_max()
        self.items[0] = self.items[-1]
        del self.items[-1]
        self.max_heapify(0)
        return largest

    def max_heapify(self, i):
        largest = i
        for j in range(self.d):
            c = self.child(i, j)
            if (c < self.size() and self.get(c) > self.get(largest)):
                largest = c
        if (largest != i):
```

```

        self.swap(largest, i)
        self.max_heapify(largest)

    def swap(self, i, j):
        self.items[i], self.items[j] = self.items[j], self.items[i]

    def insert(self, key):
        index = self.size()
        self.items.append(key)
        while (index != 0):
            p = self.parent(index)
            if self.get(p) < self.get(index):
                self.swap(p, index)
            index = p

d = int(input('Enter the value of D: '));
dheap = D_aryHeap(d)

print('Menu (this assumes no duplicate keys)')
print('insert <data>')
print('max get')
print('max extract')
print('quit')

while True:
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'insert':
        data = int(do[1])
        dheap.insert(data)
    elif operation == 'max':
        suboperation = do[1].strip().lower()
        if suboperation == 'get':
            print('Maximum value: {}'.format(dheap.get_max()))
        elif suboperation == 'extract':
            print('Maximum value removed: {}'.format(dheap.extract_max()))

    elif operation == 'quit':
        break

```

## Program Explanation

1. The user is prompted to enter the value of the number of children for each node in the heap.
2. An instance of D\_aryHeap is created.
3. The user is presented with a menu to perform various operations on the heap.
4. The corresponding methods are called to perform each operation.

## Runtime Test Cases

```

Case 1:
Enter the value of D: 5
Menu (this assumes no duplicate keys)
insert <data>
max get
max extract
quit
What would you like to do? insert 3
What would you like to do? insert 4
What would you like to do? insert 11
What would you like to do? insert 7

```

```

What would you like to do? max get
Maximum value: 11
What would you like to do? max extract
Maximum value removed: 11
What would you like to do? max extract
Maximum value removed: 7
What would you like to do? max extract
Maximum value removed: 4
What would you like to do? max extract
Maximum value removed: 3
What would you like to do? max extract
Maximum value removed: None
What would you like to do? quit

```

```

Case 2:
Enter the value of D: 2
Menu (this assumes no duplicate keys)
insert <data>
max get
max extract
quit
What would you like to do? insert 1
What would you like to do? insert 3
What would you like to do? insert 2
What would you like to do? max extract
Maximum value removed: 3
What would you like to do? max extract
Maximum value removed: 2
What would you like to do? max extract
Maximum value removed: 1
What would you like to do? quit

```

## Binomial Heap

The main application of [Binary Heap](#) is as implement priority queue. Binomial Heap is an extension of [Binary Heap](#) that provides faster union or merge operation together with other operations provided by Binary Heap. *A Binomial Heap is a collection of Binomial Trees*

### **What is a Binomial Tree?**

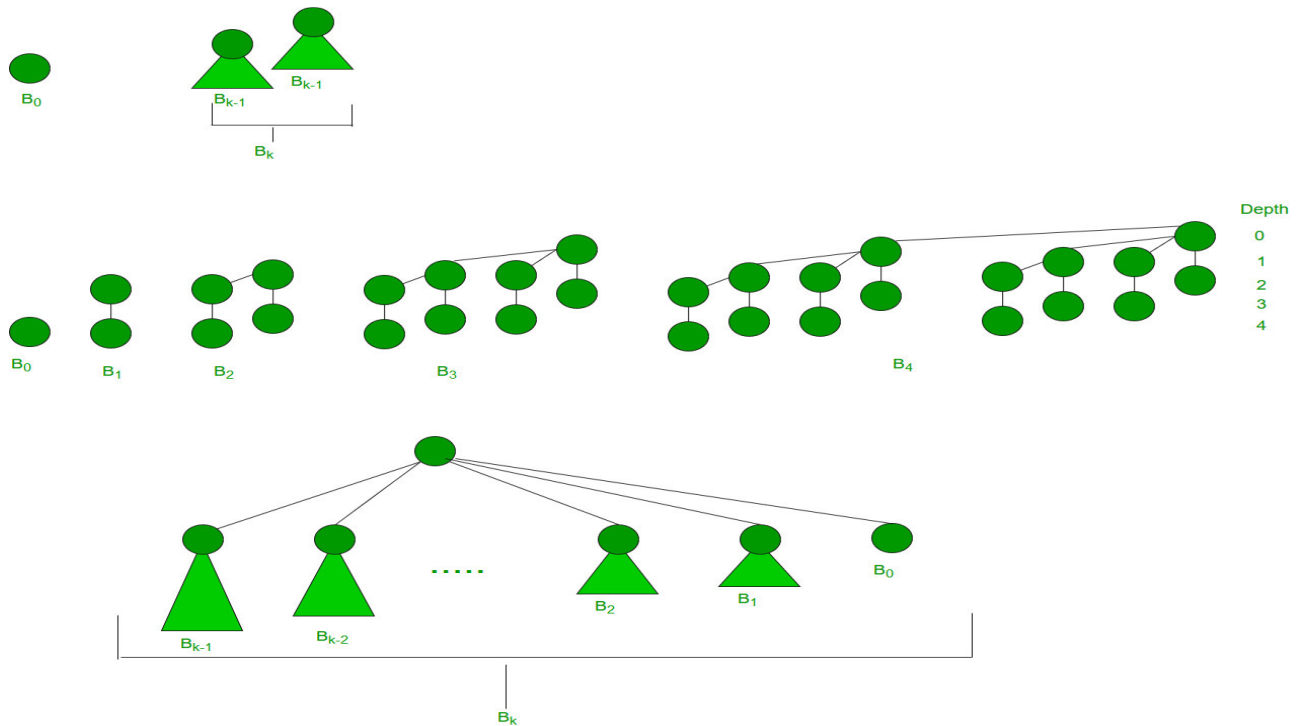
A Binomial heap is implemented as a collection of binomial trees (compare with a binary heap, which has a shape of

a single binary tree). A **binomial tree** is defined recursively

A Binomial Tree of order 0 has 1 node. A Binomial Tree of order k can be constructed by taking two binomial trees of order k-1 and making one as leftmost child or other.

A Binomial Tree of order k has following properties.

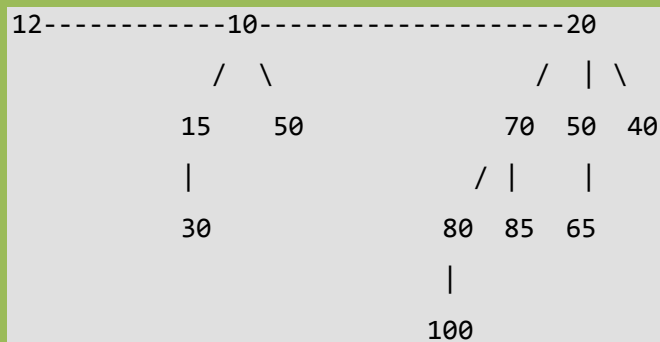
- a) It has exactly  $2^k$  nodes.
- b) It has depth as k.
- c) There are exactly  ${}^kC_i$  nodes at depth i for  $i = 0, 1, \dots, k$ .
- d) The root has degree k and children of root are themselves Binomial Trees with order k-1, k-2,... 0 from left to right.



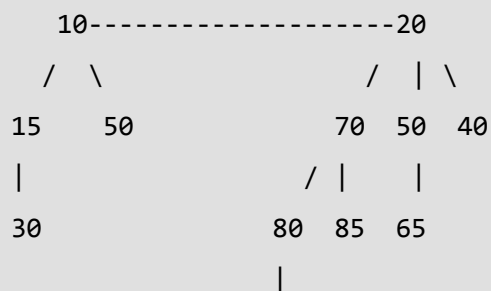
### Binomial Heap:

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

### Examples Binomial Heap:



A Binomial Heap with 13 nodes. It is a collection of 3 Binomial Trees of orders 0, 2 and 3 from left to right.





A Binomial Heap with 12 nodes. It is a collection of 2 Binomial Trees of orders 2 and 3 from left to right.

### Binary Representation of a number and Binomial Heaps

A Binomial Heap with  $n$  nodes has the number of Binomial Trees equal to the number of set bits in the Binary representation of  $n$ . For example let  $n$  be 13, there 3 set bits in the binary representation of  $n$  (00001101), hence 3 Binomial Trees. We can also relate the degree of these Binomial Trees with positions of set bits. With this relation, we can conclude that there are  $O(\text{Log}n)$  Binomial Trees in a Binomial Heap with ' $n$ ' nodes.

### Operations of Binomial Heap:

The main operation in Binomial Heap is `union()`, all other operations mainly use this operation. The `union()` operation is to combine two Binomial Heaps into one. Let us first discuss other operations, we will discuss `union` later.

- 1) `insert(H, k)`: Inserts a key ' $k$ ' to Binomial Heap ' $H$ '. This operation first creates a Binomial Heap with single key ' $k$ ', then calls `union` on  $H$  and the new Binomial heap.
- 2) `getMin(H)`: A simple way to `getMin()` is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires  $O(\text{Log}n)$  time. It can be optimized to  $O(1)$  by maintaining a pointer to minimum key root.
- 3) `extractMin(H)`: This operation also uses `union()`. We first call `getMin()` to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally, we call `union()` on  $H$  and the newly created Binomial Heap. This operation requires  $O(\text{Log}n)$  time.
- 4) `delete(H)`: Like Binary Heap, delete operation first reduces the key to minus infinite, then calls `extractMin()`.
- 5) `decreaseKey(H)`: `decreaseKey()` is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for the parent. We stop when we either reach a node whose parent has a smaller key or we hit the root node. Time complexity of `decreaseKey()` is  $O(\text{Log}n)$ .

### Union operation in Binomial Heap:

Given two Binomial Heaps  $H1$  and  $H2$ , `union(H1, H2)` creates a single Binomial Heap.

- 1) The first step is to simply merge the two Heaps in non-decreasing order of degrees. In the following diagram, figure(b) shows the result after merging.
- 2) After the simple merge, we need to make sure that there is at most one Binomial Tree of any order. To do this, we need to combine Binomial Trees of the same order. We traverse the list of merged roots, we keep track of three-pointers, `prev`, `x` and `next-x`. There can be following 4 cases when we traverse the list of roots.

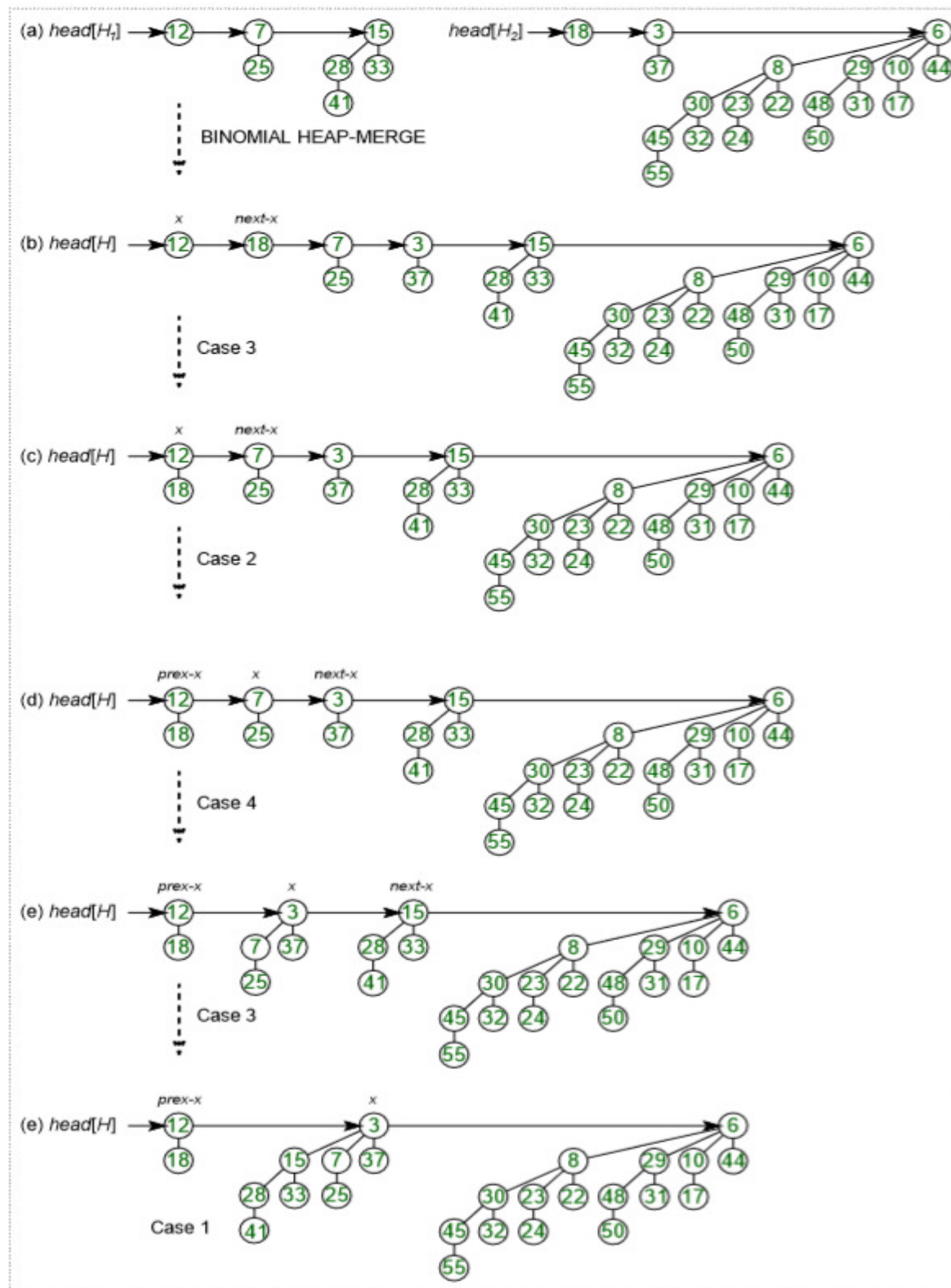
—Case 1: Orders of  $x$  and  $\text{next-}x$  are not same, we simply move ahead.

In following 3 cases orders of  $x$  and  $\text{next-}x$  are same.

—Case 2: If the order of  $\text{next-next-}x$  is also same, move ahead.

—Case 3: If the key of  $x$  is smaller than or equal to the key of  $\text{next-}x$ , then make  $\text{next-}x$  as a child of  $x$  by linking it with  $x$ .

—Case 4: If the key of  $x$  is greater, then make  $x$  as the child of  $\text{next-}x$ .



## How to represent Binomial Heap?

A Binomial Heap is a set of Binomial Trees. A Binomial Tree must be represented in a way that allows sequential access to all siblings, starting from the leftmost sibling (We need this in `insertMin()` and `delete()`). The idea is to represent Binomial Trees as the leftmost child and right-sibling representation, i.e., every node stores two pointers, one to the leftmost child and other to the right sibling.

## Implementation of Binomial Heap

1. **insert(H, k):** Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.
2. **getMin(H):** A simple way to `getMin()` is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires  $O(\log n)$  time. It can be optimized to  $O(1)$  by maintaining a pointer to minimum key root.
3. **extractMin(H):** This operation also uses `union()`. We first call `getMin()` to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call `union()` on H and the newly created Binomial Heap. This operation requires  $O(\log n)$  time.

```
// C++ program to implement different operations
// on Binomial Heap
#include<bits/stdc++.h>
using namespace std;

// A Binomial Tree node.
struct Node
{
    int data, degree;
    Node *child, *sibling, *parent;
};

Node* newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->degree = 0;
    temp->child = temp->parent = temp->sibling = NULL;
    return temp;
}

// This function merge two Binomial Trees.
Node* mergeBinomialTrees(Node *b1, Node *b2)
{
    // Make sure b1 is smaller
```

```

        if (b1->data > b2->data)
            swap(b1, b2);

        // We basically make larger valued tree
        // a child of smaller valued tree
        b2->parent = b1;
        b2->sibling = b1->child;
        b1->child = b2;
        b1->degree++;

    return b1;
}

// This function perform union operation on two
// binomial heap i.e. l1 & l2
list<Node*> unionBinomialHeap(list<Node*> l1,
                                list<Node*> l2)
{
    // _new to another binomial heap which contain
    // new heap after merging l1 & l2
    list<Node*> _new;
    list<Node*>::iterator it = l1.begin();
    list<Node*>::iterator ot = l2.begin();
    while (it!=l1.end() && ot!=l2.end())
    {
        // if D(l1) <= D(l2)
        if ((*it)->degree <= (*ot)->degree)
        {
            _new.push_back(*it);
            it++;
        }
        // if D(l1) > D(l2)
        else
        {
            _new.push_back(*ot);
            ot++;
        }
    }

    // if there remains some elements in l1
    // binomial heap
    while (it != l1.end())
    {
        _new.push_back(*it);
    }
}

```

```

        it++;
    }

    // if there remains some elements in l2
    // binomial heap
    while (ot!=l2.end())
    {
        _new.push_back(*ot);
        ot++;
    }
    return _new;
}

// adjust function rearranges the heap so that
// heap is in increasing order of degree and
// no two binomial trees have same degree in this heap
list<Node*> adjust(list<Node*> _heap)
{
    if (_heap.size() <= 1)
        return _heap;
    list<Node*> new_heap;
    list<Node*>::iterator it1,it2,it3;
    it1 = it2 = it3 = _heap.begin();

    if (_heap.size() == 2)
    {
        it2 = it1;
        it2++;
        it3 = _heap.end();
    }
    else
    {
        it2++;
        it3=it2;
        it3++;
    }
    while (it1 != _heap.end())
    {
        // if only one element remains to be processed
        if (it2 == _heap.end())
            it1++;

        // If  $D(it1) < D(it2)$  i.e. merging of Binomial
        // Tree pointed by it1 & it2 is not possible
    }
}

```

```

        // then move next in heap
        else if ((*it1)->degree < (*it2)->degree)
        {
            it1++;
            it2++;
            if(it3!=_heap.end())
                it3++;
        }

        // if D(it1),D(it2) & D(it3) are same i.e.
        // degree of three consecutive Binomial Tree are same
        // in heap
        else if (it3!=_heap.end() &&
                ((*it1)->degree == (*it2)->degree &&
                ((*it1)->degree == (*it3)->degree))
        {
            it1++;
            it2++;
            it3++;
        }

        // if degree of two Binomial Tree are same in heap
        else if ((*it1)->degree == (*it2)->degree)
        {
            Node *temp;
            *it1 = mergeBinomialTrees(*it1,*it2);
            it2 = _heap.erase(it2);
            if(it3 != _heap.end())
                it3++;
        }
    }
    return _heap;
}

```

```

// inserting a Binomial Tree into binomial heap
list<Node*> insertATreeInHeap(list<Node*> _heap,
                             Node *tree)
{
    // creating a new heap i.e temp
    list<Node*> temp;

    // inserting Binomial Tree into heap
    temp.push_back(tree);
}

```

```

        // perform union operation to finally insert
        // Binomial Tree in original heap
        temp = unionBionomialHeap(_heap,temp);

        return adjust(temp);
    }

// removing minimum key element from binomial heap
// this function take Binomial Tree as input and return
// binomial heap after
// removing head of that tree i.e. minimum element
list<Node*> removeMinFromTreeReturnBHeap(Node *tree)
{
    list<Node*> heap;
    Node *temp = tree->child;
    Node *lo;

    // making a binomial heap from Binomial Tree
    while (temp)
    {
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
    }
    return heap;
}

// inserting a key into the binomial heap
list<Node*> insert(list<Node*> _head, int key)
{
    Node *temp = newNode(key);
    return insertATreeInHeap(_head,temp);
}

// return pointer of minimum value Node
// present in the binomial heap
Node* getMin(list<Node*> _heap)
{
    list<Node*>::iterator it = _heap.begin();
    Node *temp = *it;
    while (it != _heap.end())
    {
        if ((*it)->data < temp->data)
    
```

```

        temp = *it;
        it++;
    }
    return temp;
}

list<Node*> extractMin(list<Node*> _heap)
{
    list<Node*> new_heap,lo;
    Node *temp;

    // temp contains the pointer of minimum value
    // element in heap
    temp = getMin(_heap);
    list<Node*>::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        if (*it != temp)
        {
            // inserting all Binomial Tree into new
            // binomial heap except the Binomial Tree
            // contains minimum element
            new_heap.push_back(*it);
        }
        it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    new_heap = unionBionomialHeap(new_heap,lo);
    new_heap = adjust(new_heap);
    return new_heap;
}

// print function for Binomial Tree
void printTree(Node *h)
{
    while (h)
    {
        cout << h->data << " ";
        printTree(h->child);
        h = h->sibling;
    }
}

```



```

// print function for binomial heap
void printHeap(list<Node*> _heap)
{
    list<Node*> ::iterator it;
    it = _heap.begin();
    while (it != _heap.end())
    {
        printTree(*it);
        it++;
    }
}

// Driver program to test above functions
int main()
{
    int ch,key;
    list<Node*> _heap;

    // Insert data in the heap
    _heap = insert(_heap,10);
    _heap = insert(_heap,20);
    _heap = insert(_heap,30);

    cout << "Heap elements after insertion:\n";
    printHeap(_heap);

    Node *temp = getMin(_heap);
    cout << "\nMinimum element of heap "
         << temp->data << "\n";

    // Delete minimum element of heap
    _heap = extractMin(_heap);
    cout << "Heap after deletion of minimum element\n";
    printHeap(_heap);

    return 0;
}

```

### Implementation of Binomial Heap | Set – 2 (delete() and decreaseKey())

1. **insert(H, k):** Inserts a key 'k' to Binomial Heap 'H'. This operation first creates a Binomial Heap with single key 'k', then calls union on H and the new Binomial heap.

2. **getMin(H):** A simple way to getMin() is to traverse the list of root of Binomial Trees and return the minimum key. This implementation requires  $O(\log n)$  time. It can be optimized to  $O(1)$  by maintaining a pointer to minimum key root.
3. **extractMin(H):** This operation also uses union(). We first call getMin() to find the minimum key Binomial Tree, then we remove the node and create a new Binomial Heap by connecting all subtrees of the removed minimum node. Finally we call union() on H and the newly created Binomial Heap. This operation requires  $O(\log n)$  time.
4. **delete(H):** Like Binary Heap, delete operation first reduces the key to minus infinite, then calls extractMin().
5. **decreaseKey(H):** decreaseKey() is also similar to Binary Heap. We compare the decreases key with it parent and if parent's key is more, we swap keys and recur for parent. We stop when we either reach a node whose parent has smaller key or we hit the root node. Time complexity of decreaseKey() is  $O(\log n)$

// C++ program for implementation of

// Binomial Heap and Operations on it

#include <bits/stdc++.h>

using namespace std;

// Structure of Node

struct Node

{

int val, degree;

Node \*parent, \*child, \*sibling;

};

// Making root global to avoid one extra

// parameter in all functions.

Node \*root = NULL;

// link two heaps by making h1 a child

// of h2.

int binomialLink(Node \*h1, Node \*h2)

{

h1->parent = h2;

h1->sibling = h2->child;

h2->child = h1;

h2->degree = h2->degree + 1;

}

// create a Node

Node \*createNode(int n)

{

Node \*new\_node = new Node;

new\_node->val = n;

new\_node->parent = NULL;

new\_node->sibling = NULL;

```

new_node->child = NULL;
new_node->degree = 0;
return new_node;
}

// This function merge two Binomial Trees
Node *mergeBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;

    // define a Node
    Node *res = NULL;

    // check degree of both Node i.e.
    // which is greater or smaller
    if (h1->degree <= h2->degree)
        res = h1;

    else if (h1->degree > h2->degree)
        res = h2;

    // traverse till if any of heap gets empty
    while (h1 != NULL && h2 != NULL)
    {
        // if degree of h1 is smaller, increment h1
        if (h1->degree < h2->degree)
            h1 = h1->sibling;

        // Link h1 with h2 in case of equal degree
        else if (h1->degree == h2->degree)
        {
            Node *sib = h1->sibling;
            h1->sibling = h2;
            h1 = sib;
        }

        // if h2 is greater
        else
        {
            Node *sib = h2->sibling;
            h2->sibling = h1;

```

```

        h2 = sib;
    }
}
return res;
}

// This function perform union operation on two
// binomial heap i.e. h1 & h2
Node *unionBHeaps(Node *h1, Node *h2)
{
    if (h1 == NULL && h2 == NULL)
        return NULL;

    Node *res = mergeBHeaps(h1, h2);

    // Traverse the merged list and set
    // values according to the degree of
    // Nodes
    Node *prev = NULL, *curr = res,
        *next = curr->sibling;
    while (next != NULL)
    {
        if ((curr->degree != next->degree) ||
            ((next->sibling != NULL) &&
             (next->sibling)->degree ==
             curr->degree))
        {
            prev = curr;
            curr = next;
        }
        else
        {
            if (curr->val <= next->val)
            {
                curr->sibling = next->sibling;
                binomialLink(next, curr);
            }
            else
            {
                if (prev == NULL)
                    res = next;
                else
                    prev->sibling = next;
            }
        }
    }
}

```

```

        binomialLink(curr, next);
        curr = next;
    }
}
next = curr->sibling;
}
return res;
}

```

// Function to insert a Node

```

void binomialHeapInsert(int x)
{
    // Create a new node and do union of
    // this node with root
    root = unionBHeaps(root, createNode(x));
}

```

// Function to display the Nodes

```

void display(Node *h)
{
    while (h)
    {
        cout << h->val << " ";
        display(h->child);
        h = h->sibling;
    }
}

```

// Function to reverse a list

// using recursion.

```

int revertList(Node *h)
{
    if (h->sibling != NULL)
    {
        revertList(h->sibling);
        (h->sibling)->sibling = h;
    }
    else
        root = h;
}

```

// Function to extract minimum value

```

Node *extractMinBHeap(Node *h)
{

```

```

if (h == NULL)
return NULL;

Node *min_node_prev = NULL;
Node *min_node = h;

// Find minimum value
int min = h->val;
Node *curr = h;
while (curr->sibling != NULL)
{
    if ((curr->sibling)->val < min)
    {
        min = (curr->sibling)->val;
        min_node_prev = curr;
        min_node = curr->sibling;
    }
    curr = curr->sibling;
}

// If there is a single Node
if (min_node_prev == NULL &&
    min_node->sibling == NULL)
    h = NULL;

else if (min_node_prev == NULL)
    h = min_node->sibling;

// Remove min node from list
else
    min_node_prev->sibling = min_node->sibling;

// Set root (which is global) as children
// list of min node
if (min_node->child != NULL)
{
    revertList(min_node->child);
    (min_node->child)->sibling = NULL;
}

// Do union of root h and children
return unionBHeaps(h, root);
}

```

```

// Function to search for an element
Node *findNode(Node *h, int val)
{
    if (h == NULL)
        return NULL;

    // check if key is equal to the root's data
    if (h->val == val)
        return h;

    // Recur for child
    Node *res = findNode(h->child, val);
    if (res != NULL)
        return res;

    return findNode(h->sibling, val);
}

// Function to decrease the value of old_val
// to new_val
void decreaseKeyBHeap(Node *H, int old_val,
                      int new_val)
{
    // First check element present or not
    Node *node = findNode(H, old_val);

    // return if Node is not present
    if (node == NULL)
        return;

    // Reduce the value to the minimum
    node->val = new_val;
    Node *parent = node->parent;

    // Update the heap according to reduced value
    while (parent != NULL && node->val < parent->val)
    {
        swap(node->val, parent->val);
        node = parent;
        parent = parent->parent;
    }
}

// Function to delete an element

```

```

Node *binomialHeapDelete(Node *h, int val)
{
    // Check if heap is empty or not
    if (h == NULL)
        return NULL;

    // Reduce the value of element to minimum
    decreaseKeyBHeap(h, val, INT_MIN);

    // Delete the minimum element from heap
    return extractMinBHeap(h);
}

// Driver code
int main()
{
    // Note that root is global
    binomialHeapInsert(10);
    binomialHeapInsert(20);
    binomialHeapInsert(30);
    binomialHeapInsert(40);
    binomialHeapInsert(50);

    cout << "The heap is:\n";
    display(root);

    // Delete a particular element from heap
    root = binomialHeapDelete(root, 10);

    cout << "\nAfter deleting 10, the heap is:\n";

    display(root);

    return 0;
}

```

## Python Program to Implement Binomial Heap

### Problem Solution

1. Create a class BinomialTree with instance variables key, children and order. children is set to an empty list and order is set to 0 when an object is instantiated.
2. Define method add\_at\_end which takes a binomial tree of the same order as argument and adds it to the current tree, increasing its order by 1.



3. Create a class BinomialHeap with an instance variable trees set to an empty list. This list will contain the set of binomial trees.
4. Define methods get\_min, extract\_min, combine\_roots, merge and insert.
5. The method get\_min returns the minimum element in the heap by returning the key of the smallest root in the list trees.
6. The method merge takes a heap as argument and merges it with the current heap. It iterates through the sorted (by order of each tree) list of trees and merges any two trees with the same order. It also checks for the case for three consecutive trees of the same order and merges the last two trees.
7. The method combine\_roots takes a heap as argument and combines the current heap's list of trees with its list of trees and sorts them by order of each tree.
8. The method extract\_min removes and returns the minimum element in the current heap. It does so by removing the tree with the smallest root from the current heap's list of trees and creating a heap with the children of the smallest root as its list of trees. This new heap is then merged with the current heap.
9. The method insert takes a key as argument and adds a node with that key to the heap. It does so by creating an order 0 heap with that key and then merging it with the current heap.

## Program/Source Code

```
class BinomialTree:
    def __init__(self, key):
        self.key = key
        self.children = []
        self.order = 0

    def add_at_end(self, t):
        self.children.append(t)
        self.order = self.order + 1

class BinomialHeap:
    def __init__(self):
        self.trees = []

    def extract_min(self):
        if self.trees == []:
            return None
        smallest_node = self.trees[0]
        for tree in self.trees:
            if tree.key < smallest_node.key:
                smallest_node = tree
        self.trees.remove(smallest_node)
        h = BinomialHeap()
        h.trees = smallest_node.children
        self.merge(h)

        return smallest_node.key

    def get_min(self):
        if self.trees == []:
            return None
        least = self.trees[0].key
```

```

    for tree in self.trees:
        if tree.key < least:
            least = tree.key
    return least

def combine_roots(self, h):
    self.trees.extend(h.trees)
    self.trees.sort(key=lambda tree: tree.order)

def merge(self, h):
    self.combine_roots(h)
    if self.trees == []:
        return
    i = 0
    while i < len(self.trees) - 1:
        current = self.trees[i]
        after = self.trees[i + 1]
        if current.order == after.order:
            if (i + 1 < len(self.trees) - 1
                and self.trees[i + 2].order == after.order):
                after_after = self.trees[i + 2]
                if after.key < after_after.key:
                    after.add_at_end(after_after)
                    del self.trees[i + 2]
            else:
                after_after.add_at_end(after)
                del self.trees[i + 1]
        else:
            if current.key < after.key:
                current.add_at_end(after)
                del self.trees[i + 1]
            else:
                after.add_at_end(current)
                del self.trees[i]
        i = i + 1

def insert(self, key):
    g = BinomialHeap()
    g.trees.append(BinomialTree(key))
    self.merge(g)

```

```

bheap = BinomialHeap()

```

```

print('Menu')
print('insert <data>')
print('min get')
print('min extract')
print('quit')

```

```

while True:
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'insert':
        data = int(do[1])
        bheap.insert(data)
    elif operation == 'min':
        suboperation = do[1].strip().lower()
        if suboperation == 'get':
            print('Minimum value: {}'.format(bheap.get_min()))
        elif suboperation == 'extract':
            print('Minimum value removed: {}'.format(bheap.extract_min()))

```

```
elif operation == 'quit':  
    break
```

## Program Explanation

1. Create an instance of BinomialHeap.
2. The user is presented with a menu to perform various operations on the heap.
3. The corresponding methods are called to perform each operation.

## Runtime Test Cases

Case 1:

```
Menu  
insert <data>  
min get  
min extract  
quit  
What would you like to do? insert 3  
What would you like to do? insert 7  
What would you like to do? insert 1  
What would you like to do? insert 4  
What would you like to do? min get  
Minimum value: 1  
What would you like to do? min extract  
Minimum value removed: 1  
What would you like to do? min extract  
Minimum value removed: 3  
What would you like to do? min extract  
Minimum value removed: 4  
What would you like to do? min extract  
Minimum value removed: 7  
What would you like to do? min extract  
Minimum value removed: None  
What would you like to do? quit
```

Case 2:

```
Menu  
insert <data>  
min get  
min extract  
quit  
What would you like to do? insert 10  
What would you like to do? insert 12  
What would you like to do? insert 5  
What would you like to do? insert 6  
What would you like to do? min get  
Minimum value: 5  
What would you like to do? insert 3  
What would you like to do? min get  
Minimum value: 3  
What would you like to do? insert 8  
What would you like to do? min extract  
Minimum value removed: 3  
What would you like to do? min extract  
Minimum value removed: 5  
What would you like to do? insert 1  
What would you like to do? min extract  
Minimum value removed: 1  
What would you like to do? quit
```

# Fibonacci Heap

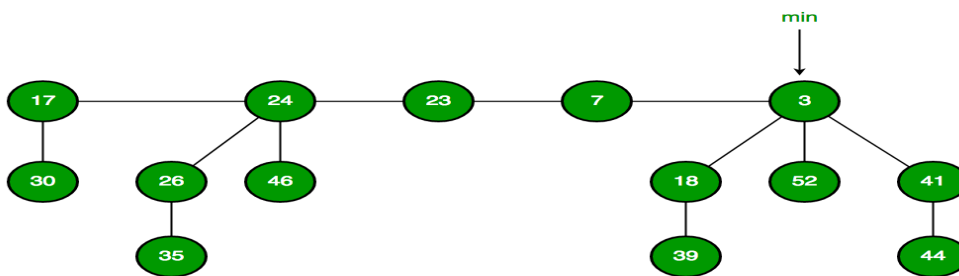
Heaps are mainly used for implementing priority queue. In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are [amortized time complexities](#) of **Fibonacci Heap**.

1) Find Min:	$\Theta(1)$	[Same as both Binary and Binomial]
2) Delete Min:	$\Theta(\log n)$	$[\Theta(\log n)$ in both Binary and Binomial]
3) Insert:	$\Theta(1)$	$[\Theta(\log n)$ in Binary and $\Theta(1)$ in Binomial]
4) Decrease-Key:	$\Theta(1)$	$[\Theta(\log n)$ in both Binary and Binomial]
5) Merge:	$\Theta(1)$	$[\Theta(m \log n)$ or $\Theta(m+n)$ in Binary and $\Theta(\log n)$ in Binomial]

Like [Binomial Heap](#), 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 (This is unlike Binomial Heap where every tree has to be Binomial Tree).

Below is an example Fibonacci Heap taken from [here](#).



Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single 'min' pointer.

The main idea is to execute operations in "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.

## **Below are some interesting facts about Fibonacci Heap**

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is  $O(V \log V + E \log V)$ . If Fibonacci Heap is used, then time complexity is improved to  $O(V \log V + E)$
2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high (Source [Wiki](#)).
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has 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.

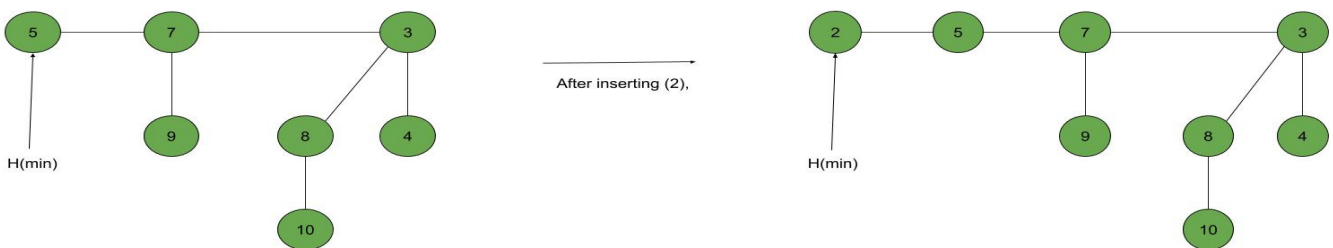
Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to

be Binomial Tree). In this article, we will discuss Insertion and Union operation on Fibonacci Heap.

**Insertion:** To insert a node in a Fibonacci heap  $H$ , the following algorithm is followed:

1. Create a new node 'x'.
2. Check whether heap  $H$  is empty or not.
3. If  $H$  is empty then:
  - Make  $x$  as the only node in the root list.
  - Set  $H(\min)$  pointer to  $x$ .
4. Else:
  - Insert  $x$  into root list and update  $H(\min)$ .

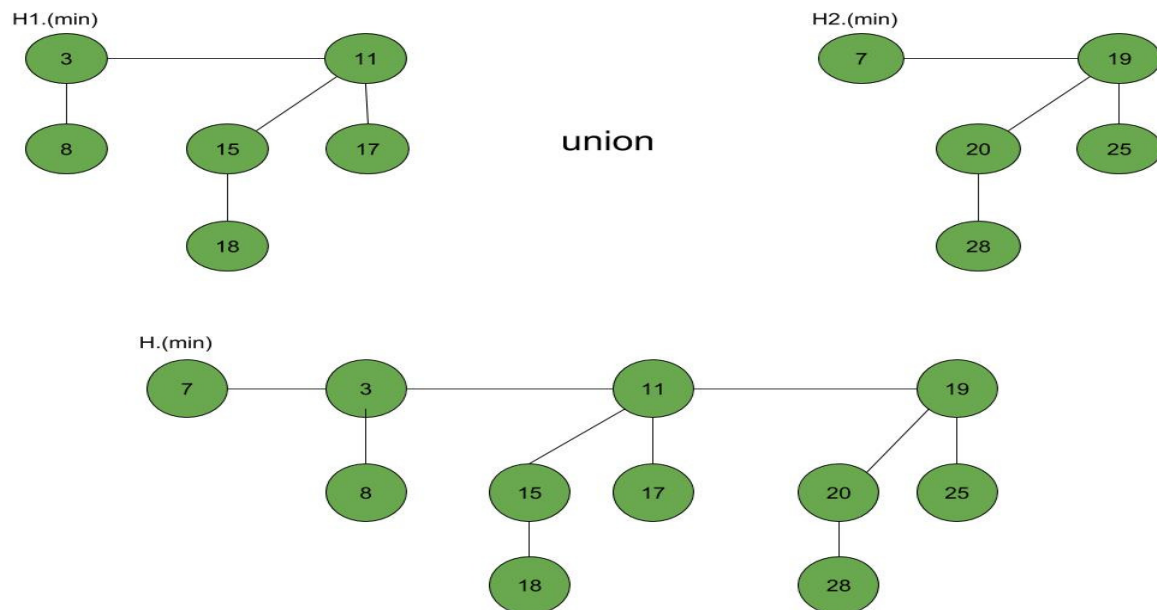
**Example:**



**Union:** Union of two Fibonacci heaps  $H_1$  and  $H_2$  can be accomplished as follows:

1. Join root lists of Fibonacci heaps  $H_1$  and  $H_2$  and make a single Fibonacci heap  $H$ .
2. If  $H_1(\min) < H_2(\min)$  then:
  - $H(\min) = H_1(\min)$ .
3. Else:
  - $H(\min) = H_2(\min)$ .

**Example:**



```
// C++ program to demonstrate building
// and inserting in a Fibonacci heap
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;

struct node {
    node* parent;
    node* child;
    node* left;
    node* right;
    int key;
};

// Creating min pointer as "mini"
struct node* mini = NULL;

// Declare an integer for number of nodes in the heap
int no_of_nodes = 0;

// Function to insert a node in heap
void insertion(int val)
{
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
    new_node->key = val;
    new_node->parent = NULL;
    new_node->child = NULL;
    new_node->left = new_node;
    new_node->right = new_node;
    if (mini != NULL) {
        (mini->left)->right = new_node;
    }
}
```

```

        new_node->right = mini;
        new_node->left = mini->left;
        mini->left = new_node;
        if (new_node->key < mini->key)
            mini = new_node;
    }
    else {
        mini = new_node;
    }
}

// Function to display the heap
void display(struct node* mini)
{
    node* ptr = mini;
    if (ptr == NULL)
        cout << "The Heap is Empty" << endl;

    else {
        cout << "The root nodes of Heap are: " << endl;
        do {
            cout << ptr->key;
            ptr = ptr->right;
            if (ptr != mini) {
                cout << "-->";
            }
        } while (ptr != mini && ptr->right != NULL);
        cout << endl
            << "The heap has " << no_of_nodes << " nodes" << endl;
    }
}

// Function to find min node in the heap
void find_min(struct node* mini)
{
    cout << "min of heap is: " << mini->key << endl;
}

// Driver code
int main()
{
    no_of_nodes = 7;
    insertion(4);
    insertion(3);
    insertion(7);
    insertion(5);
    insertion(2);
}

```

```

insertion(1);
insertion(10);

display(mini);

find_min(mini);

return 0;
}

```

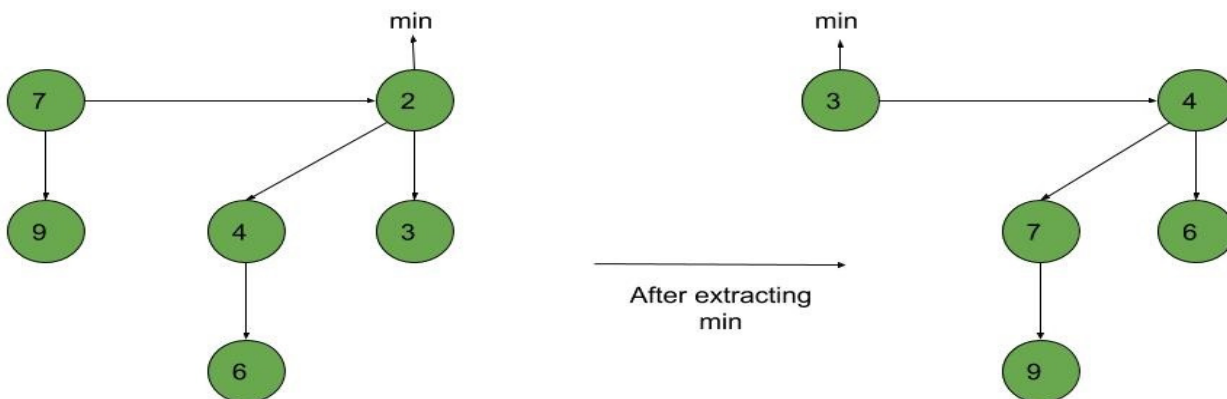
## Fibonacci Heap – Deletion, Extract min and Decrease key

We will discuss Extract\_min(), Decrease\_key() and Deletion() operations on Fibonacci heap.

**Extract\_min():** We create a function for deleting the minimum node and setting the min pointer to the minimum value in the remaining heap. The following algorithm is followed:

1. Delete the min node.
2. Set head to the next min node and add all the tree of the deleted node in root list.
3. Create an array of degree pointers of the size of the deleted node.
4. Set degree pointer to current node.
5. Move to the next node.
  - If degrees are different then set degree pointer to next node.
  - If degrees are same then join the Fibonacci trees by union operation.
6. Repeat steps 4 and 5 until the heap is completed.

**Example:**



**Decrease\_key():** To decrease the value of any element in the heap, we follow the following algorithm:  
Decrease the value of the node 'x' to the new chosen value.

CASE 1) If min heap property is not violated,

- Update min pointer if necessary.

CASE 2) If min heap property is violated and parent of 'x' is unmarked,

- Cut off the link between 'x' and its parent.
- Mark the parent of 'x'.
- Add tree rooted at 'x' to the root list and update min pointer if necessary.

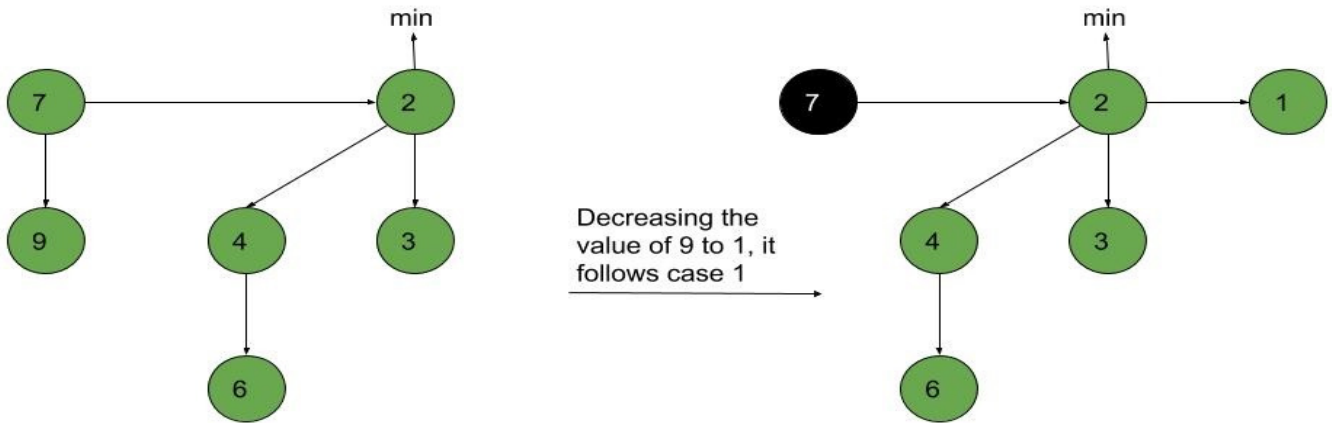
CASE 3) If min heap property is violated and parent of 'x' is marked,

- Cut off the link between 'x' and its parent p[x].



- Add 'x' to the root list, updating min pointer if necessary.
- Cut off link between p[x] and p[p[x]].
- Add p[x] to the root list, updating min pointer if necessary.
- If p[p[x]] is unmarked, mark it.
- Else, cut off p[p[x]] and repeat steps 4.2 to 4.5, taking p[p[x]] as 'x'.

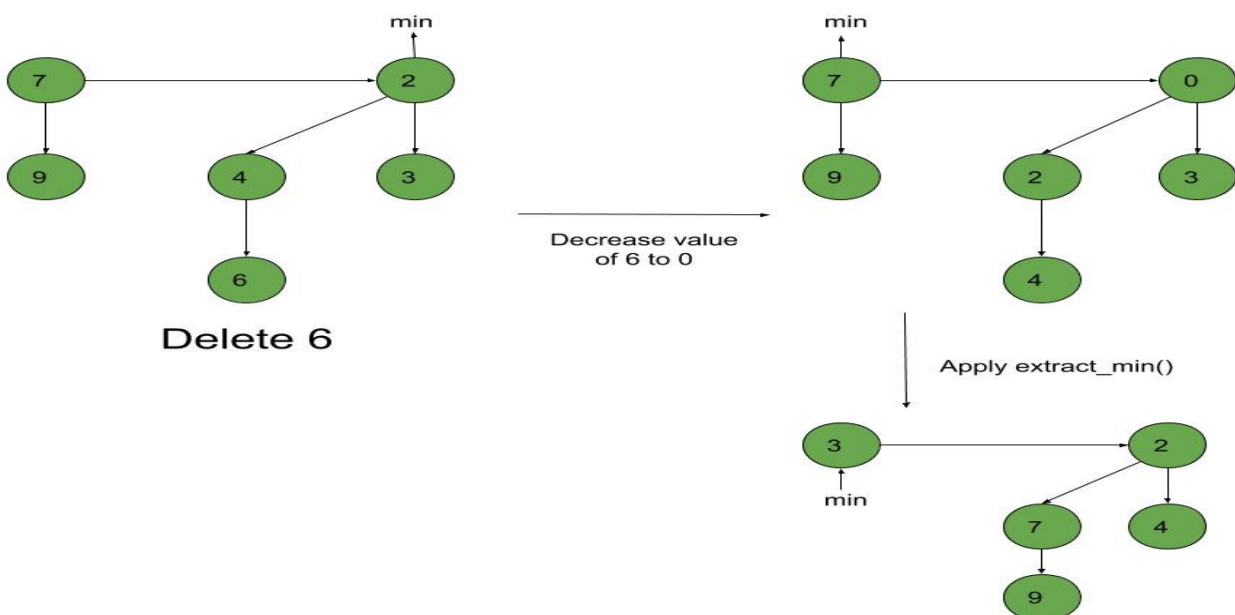
**Example:**



**Deletion():** To delete any element in a Fibonacci heap, the following algorithm is followed:

1. Decrease the value of the node to be deleted 'x' to minimum by Decrease\_key() function.
2. By using min heap property, heapify the heap containing 'x', bringing 'x' to the root list.
3. Apply Extract\_min() algorithm to the Fibonacci heap.

**Example:**



Following is a program to demonstrate Extract min(), Deletion() and Decrease key() operations on a Fibonacci Heap:

```
// C++ program to demonstrate Extract min, Deletion()
// and Decrease key() operations in a fibonacci heap
```

```

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;

// Creating a structure to represent a node in the heap
struct node {
    node* parent; // Parent pointer
    node* child; // Child pointer
    node* left; // Pointer to the node on the left
    node* right; // Pointer to the node on the right
    int key; // Value of the node
    int degree; // Degree of the node
    char mark; // Black or white mark of the node
    char c; // Flag for assisting in the Find node function
};

// Creating min pointer as "mini"
struct node* mini = NULL;

// Declare an integer for number of nodes in the heap
int no_of_nodes = 0;

// Function to insert a node in heap
void insertion(int val)
{
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
    new_node->key = val;
    new_node->degree = 0;
    new_node->mark = 'W';
    new_node->c = 'N';
    new_node->parent = NULL;
    new_node->child = NULL;
    new_node->left = new_node;
    new_node->right = new_node;
    if (mini != NULL) {
        (mini->left)->right = new_node;
        new_node->right = mini;
        new_node->left = mini->left;
        mini->left = new_node;
        if (new_node->key < mini->key)
            mini = new_node;
    }
}

```

```

        else {
            mini = new_node;
        }
        no_of_nodes++;
    }
// Linking the heap nodes in parent child relationship
void Fibonnaci_link(struct node* ptr2, struct node* ptr1)
{
    (ptr2->left)->right = ptr2->right;
    (ptr2->right)->left = ptr2->left;
    if (ptr1->right == ptr1)
        mini = ptr1;
    ptr2->left = ptr2;
    ptr2->right = ptr2;
    ptr2->parent = ptr1;
    if (ptr1->child == NULL)
        ptr1->child = ptr2;
    ptr2->right = ptr1->child;
    ptr2->left = (ptr1->child)->left;
    ((ptr1->child)->left)->right = ptr2;
    (ptr1->child)->left = ptr2;
    if (ptr2->key < (ptr1->child)->key)
        ptr1->child = ptr2;
    ptr1->degree++;
}
// Consolidating the heap
void Consolidate()
{
    int temp1;
    float temp2 = (log(no_of_nodes)) / (log(2));
    int temp3 = temp2;
    struct node* arr[temp3];
    for (int i = 0; i <= temp3; i++)
        arr[i] = NULL;
    node* ptr1 = mini;
    node* ptr2;
    node* ptr3;
    node* ptr4 = ptr1;
    do {
        ptr4 = ptr4->right;
        temp1 = ptr1->degree;
        while (arr[temp1] != NULL) {
            ptr2 = arr[temp1];
            if (ptr1->key > ptr2->key) {

```

```

        ptr3 = ptr1;
        ptr1 = ptr2;
        ptr2 = ptr3;
    }
    if (ptr2 == mini)
        mini = ptr1;
    Fibonnaci_link(ptr2, ptr1);
    if (ptr1->right == ptr1)
        mini = ptr1;
    arr[temp1] = NULL;
    temp1++;
}
arr[temp1] = ptr1;
ptr1 = ptr1->right;
} while (ptr1 != mini);
mini = NULL;
for (int j = 0; j <= temp3; j++) {
    if (arr[j] != NULL) {
        arr[j]->left = arr[j];
        arr[j]->right = arr[j];
        if (mini != NULL) {
            (mini->left)->right = arr[j];
            arr[j]->right = mini;
            arr[j]->left = mini->left;
            mini->left = arr[j];
            if (arr[j]->key < mini->key)
                mini = arr[j];
        }
        else {
            mini = arr[j];
        }
        if (mini == NULL)
            mini = arr[j];
        else if (arr[j]->key < mini->key)
            mini = arr[j];
    }
}
}
}

```

```

// Function to extract minimum node in the heap
void Extract_min()
{
    if (mini == NULL)
        cout << "The heap is empty" << endl;
}

```

```

else {
    node* temp = mini;
    node* pntr;
    pntr = temp;
    node* x = NULL;
    if (temp->child != NULL) {

        x = temp->child;
        do {
            pntr = x->right;
            (mini->left)->right = x;
            x->right = mini;
            x->left = mini->left;
            mini->left = x;
            if (x->key < mini->key)
                mini = x;
            x->parent = NULL;
            x = pntr;
        } while (pntr != temp->child);
    }
    (temp->left)->right = temp->right;
    (temp->right)->left = temp->left;
    mini = temp->right;
    if (temp == temp->right && temp->child == NULL)
        mini = NULL;
    else {
        mini = temp->right;
        Consolidate();
    }
    no_of_nodes--;
}
}

```

// Cutting a node in the heap to be placed in the root list

```
void Cut(struct node* found, struct node* temp)
```

```

{
    if (found == found->right)
        temp->child = NULL;

    (found->left)->right = found->right;
    (found->right)->left = found->left;
    if (found == temp->child)
        temp->child = found->right;
}

```

```

temp->degree = temp->degree - 1;
found->right = found;
found->left = found;
(mini->left)->right = found;
found->right = mini;
found->left = mini->left;
mini->left = found;
found->parent = NULL;
found->mark = 'B';
}

```

// Recursive cascade cutting function

```

void Cascase_cut(struct node* temp)
{
    node* ptr5 = temp->parent;
    if (ptr5 != NULL) {
        if (temp->mark == 'W') {
            temp->mark = 'B';
        }
        else {
            Cut(temp, ptr5);
            Cascase_cut(ptr5);
        }
    }
}

```

// Function to decrease the value of a node in the heap

```

void Decrease_key(struct node* found, int val)
{
    if (mini == NULL)
        cout << "The Heap is Empty" << endl;

    if (found == NULL)
        cout << "Node not found in the Heap" << endl;

    found->key = val;

    struct node* temp = found->parent;
    if (temp != NULL && found->key < temp->key) {
        Cut(found, temp);
        Cascase_cut(temp);
    }
    if (found->key < mini->key)
        mini = found;
}

```

```
}
```

```
// Function to find the given node
```

```
void Find(struct node* mini, int old_val, int val)
```

```
{
```

```
    struct node* found = NULL;
```

```
    node* temp5 = mini;
```

```
    temp5->c = 'Y';
```

```
    node* found_ptr = NULL;
```

```
    if (temp5->key == old_val) {
```

```
        found_ptr = temp5;
```

```
        temp5->c = 'N';
```

```
        found = found_ptr;
```

```
        Decrease_key(found, val);
```

```
    }
```

```
    if (found_ptr == NULL) {
```

```
        if (temp5->child != NULL)
```

```
            Find(temp5->child, old_val, val);
```

```
        if ((temp5->right)->c != 'Y')
```

```
            Find(temp5->right, old_val, val);
```

```
    }
```

```
    temp5->c = 'N';
```

```
    found = found_ptr;
```

```
}
```

```
// Deleting a node from the heap
```

```
void Deletion(int val)
```

```
{
```

```
    if (mini == NULL)
```

```
        cout << "The heap is empty" << endl;
```

```
    else {
```

```
        // Decreasing the value of the node to 0
```

```
        Find(mini, val, 0);
```

```
        // Calling Extract_min function to
```

```
        // delete minimum value node, which is 0
```

```
        Extract_min();
```

```
        cout << "Key Deleted" << endl;
```

```
    }
```

```
}
```

```
// Function to display the heap
```

```
void display()
```

```

{
    node* ptr = mini;
    if (ptr == NULL)
        cout << "The Heap is Empty" << endl;

    else {
        cout << "The root nodes of Heap are: " << endl;
        do {
            cout << ptr->key;
            ptr = ptr->right;
            if (ptr != mini) {
                cout << "-->";
            }
        } while (ptr != mini && ptr->right != NULL);
        cout << endl
            << "The heap has " << no_of_nodes << " nodes" << endl
            << endl;
    }
}

```

// Driver code

```
int main()
```

```

{
    // We will create a heap and insert 3 nodes into it
    cout << "Creating an initial heap" << endl;
    insertion(5);
    insertion(2);
    insertion(8);

    // Now we will display the root list of the heap
    display();

    // Now we will extract the minimum value node from the heap
    cout << "Extracting min" << endl;
    Extract_min();
    display();

    // Now we will decrease the value of node '8' to '7'
    cout << "Decrease value of 8 to 7" << endl;
    Find(mini, 8, 7);
    display();

    // Now we will delete the node '7'
    cout << "Delete the node 7" << endl;
}

```



```

        Deletion(7);
        display();

    return 0;
}

```

## Python Program to Implement Fibonacci Heap

### Problem Solution

1. Create a class FibonacciTree with instance variables key, children and order. children is set to an empty list and order is set to 0 when an object is instantiated.
2. Define method add\_at\_end which takes a Fibonacci tree of the same order as argument and adds it to the current tree, increasing its order by 1.
3. Create a class FibonacciHeap with instance variables trees, least and count. The variable trees is set to an empty list, least to None and count to 0 on instantiation. The list will contain the set of Fibonacci trees, least will point to the tree with the least element and count will contain the number of nodes in the heap.
4. Define methods get\_min, extract\_min, consolidate and insert.
5. The method get\_min returns the minimum element in the heap by returning the key of the variable least.
6. The method extract\_min removes and returns the minimum element in the current heap. It does so by removing the tree that least points to from the current heap's list of trees and then appending the children of the removed node to the list of trees. The method consolidate is then called before returning the key of the least element.
7. The method consolidate combines the trees in the heap such that there is at most one tree of any order. It also sets the variable least of the heap to the tree with the smallest element.
8. The method insert takes a key as argument and adds a node with that key to the heap. It does so by creating an order 0 Fibonacci tree with that key and appending it to list of trees of the heap. It then updates count and, if required, least.
9. Define the function floor\_log2 which takes a number as argument and returns the floor of its base 2 logarithm.

### Program/Source Code

```

import math

class FibonacciTree:
    def __init__(self, key):
        self.key = key
        self.children = []
        self.order = 0

    def add_at_end(self, t):
        self.children.append(t)
        self.order = self.order + 1

class FibonacciHeap:
    def __init__(self):
        self.trees = []
        self.least = None
        self.count = 0

```

```

def insert(self, key):
    new_tree = FibonacciTree(key)
    self.trees.append(new_tree)
    if (self.least is None or key < self.least.key):
        self.least = new_tree
    self.count = self.count + 1

def get_min(self):
    if self.least is None:
        return None
    return self.least.key

def extract_min(self):
    smallest = self.least
    if smallest is not None:
        for child in smallest.children:
            self.trees.append(child)
        self.trees.remove(smallest)
        if self.trees == []:
            self.least = None
        else:
            self.least = self.trees[0]
            self consolidate()
        self.count = self.count - 1
    return smallest.key

def consolidate(self):
    aux = (floor_log2(self.count) + 1)*[None]

    while self.trees != []:
        x = self.trees[0]
        order = x.order
        self.trees.remove(x)
        while aux[order] is not None:
            y = aux[order]
            if x.key > y.key:
                x, y = y, x
            x.add_at_end(y)
            aux[order] = None
            order = order + 1
        aux[order] = x

    self.least = None
    for k in aux:
        if k is not None:
            self.trees.append(k)
            if (self.least is None
                or k.key < self.least.key):
                self.least = k

def floor_log2(x):
    return math.frexp(x)[1] - 1

fheap = FibonacciHeap()

print('Menu')
print('insert <data>')
print('min get')
print('min extract')
print('quit')

```

```

while True:
    do = input('What would you like to do? ').split()

    operation = do[0].strip().lower()
    if operation == 'insert':
        data = int(do[1])
        fheap.insert(data)
    elif operation == 'min':
        suboperation = do[1].strip().lower()
        if suboperation == 'get':
            print('Minimum value: {}'.format(fheap.get_min()))
        elif suboperation == 'extract':
            print('Minimum value removed: {}'.format(fheap.extract_min()))

    elif operation == 'quit':
        break

```

### Program Explanation

1. Create an instance of FibonacciHeap.
2. The user is presented with a menu to perform various operations on the heap.
3. The corresponding methods are called to perform each operation.

### Runtime Test Cases

```

Case 1:
Menu
insert <data>
min get
min extract
quit
What would you like to do? insert 3
What would you like to do? insert 2
What would you like to do? insert 7
What would you like to do? min get
Minimum value: 2
What would you like to do? min extract
Minimum value removed: 2
What would you like to do? min extract
Minimum value removed: 3
What would you like to do? min extract
Minimum value removed: 7
What would you like to do? min extract
Minimum value removed: None
What would you like to do? quit

```

```

Case 2:
Menu
insert <data>
min get
min extract
quit
What would you like to do? insert 1
What would you like to do? insert 2
What would you like to do? insert 3
What would you like to do? insert 4
What would you like to do? insert 0
What would you like to do? min extract
Minimum value removed: 0
What would you like to do? min extract
Minimum value removed: 1

```

```

What would you like to do? min extract
Minimum value removed: 2
What would you like to do? min extract
Minimum value removed: 3
What would you like to do? min extract
Minimum value removed: 4
What would you like to do? quit

```

## Leftist Tree / Leftist Heap

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a complete binary tree), a leftist tree may be very unbalanced.

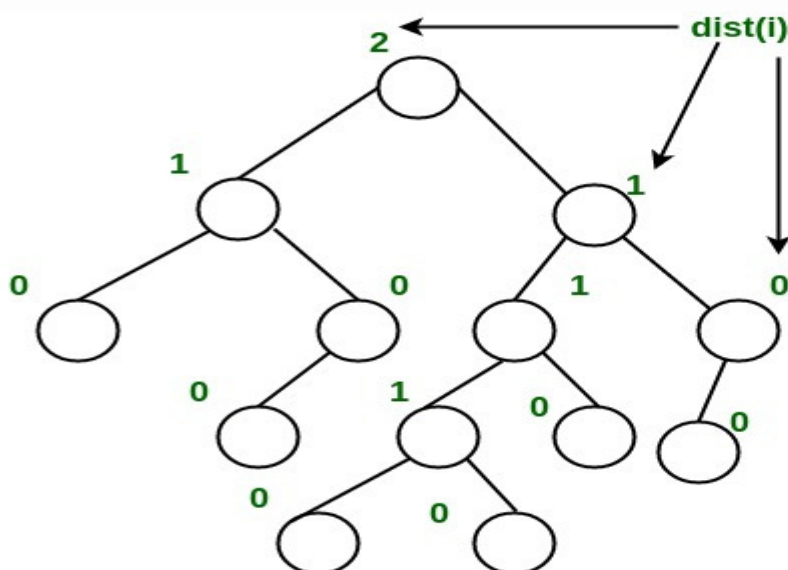
Below are time complexities of **Leftist Tree / Heap**.

Function	Complexity	Comparison
1) Get Min:	$O(1)$	[same as both Binary and Binomial]
2) Delete Min:	$O(\log n)$	[same as both Binary and Binomial]
3) Insert:	$O(\log n)$	[ $O(\log n)$ in Binary and $O(1)$ in Binomial and $O(\log n)$ for worst case]
4) Merge:	$O(\log n)$	[ $O(\log n)$ in Binomial]

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property** :  $\text{key}(i) \geq \text{key}(\text{parent}(i))$
2. **Heavier on left side** :  $\text{dist}(\text{right}(i)) \leq \text{dist}(\text{left}(i))$ . Here,  $\text{dist}(i)$  is the number of edges on the shortest path from node  $i$  to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and  $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$ .

**Example:** The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right subtree of this node is null and its parent has a distance of 1 by  $\text{dist}(i) = 1 + \text{dist}(\text{right}(i))$ . The same is followed for each node and their s-value (or rank) is calculated.



**Node:**

data	
dist	
L	R

From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has  $x$  nodes, then leftist heap has atleast  $2^x - 1$  nodes. This means the length of path to rightmost leaf is  $O(\log n)$  for a leftist heap with  $n$  nodes.

#### Operations :

1. The main operation is merge().
2. deleteMin() (or extractMin()) can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done by create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

#### Idea behind Merging :

Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion:
  - Update dist() of merged root.
  - Swap left and right subtrees just below root, if needed, to keep leftist property of merged result

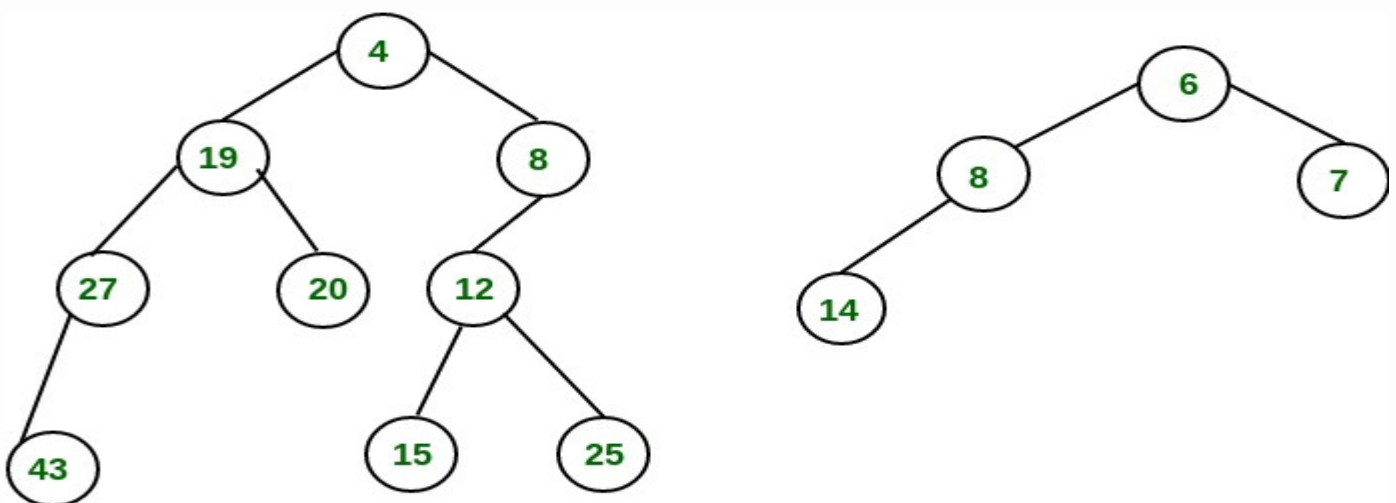
Source : <http://courses.cs.washington.edu/courses/cse326/08sp/lectures/05-leftist-heaps.pdf>

#### Detailed Steps for Merge:

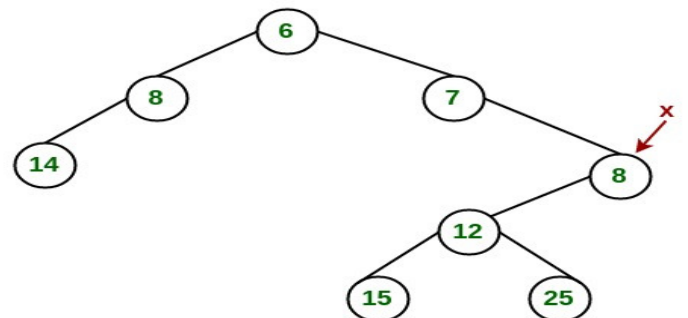
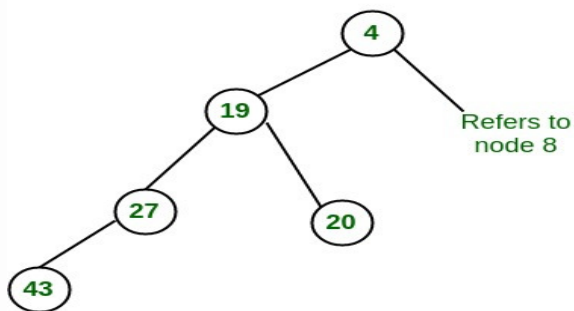
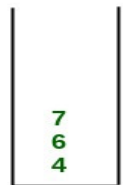
1. Compare the roots of two heaps.
2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

#### Example:

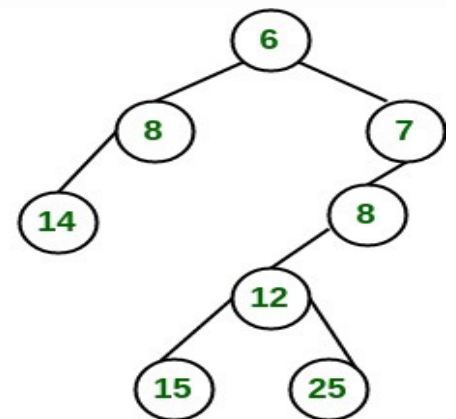
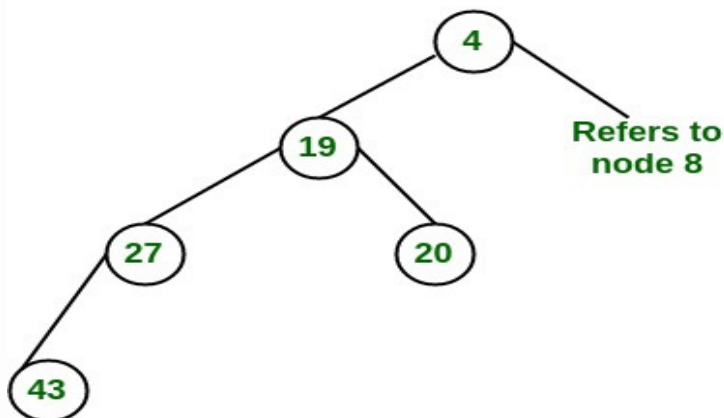
Consider two leftist heaps given below:



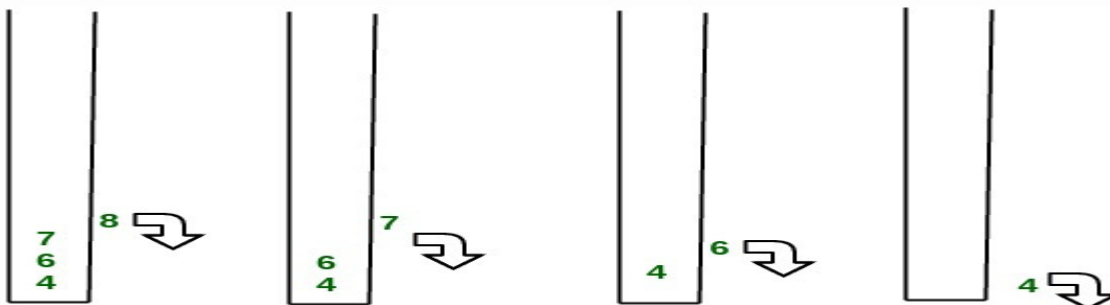
Compare(4,6)  
 Push 4  
 Compare(8,6)  
 Push 6  
 Compare(8,7)  
 Push 7  
 Compare(8,null)  
 As null is encountered, we make node 8 as right sub-tree of stack top, i.e. 7

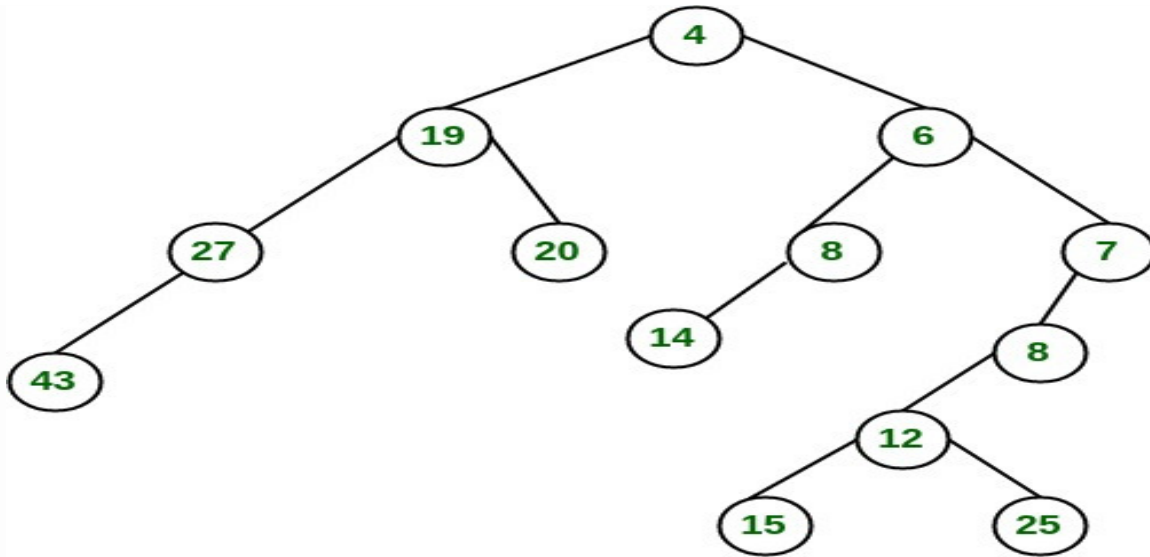


The subtree at node 7 violates the property of leftist heap so we swap it with the left child and retain the property of leftist heap.



Convert to leftist heap. Repeat the process

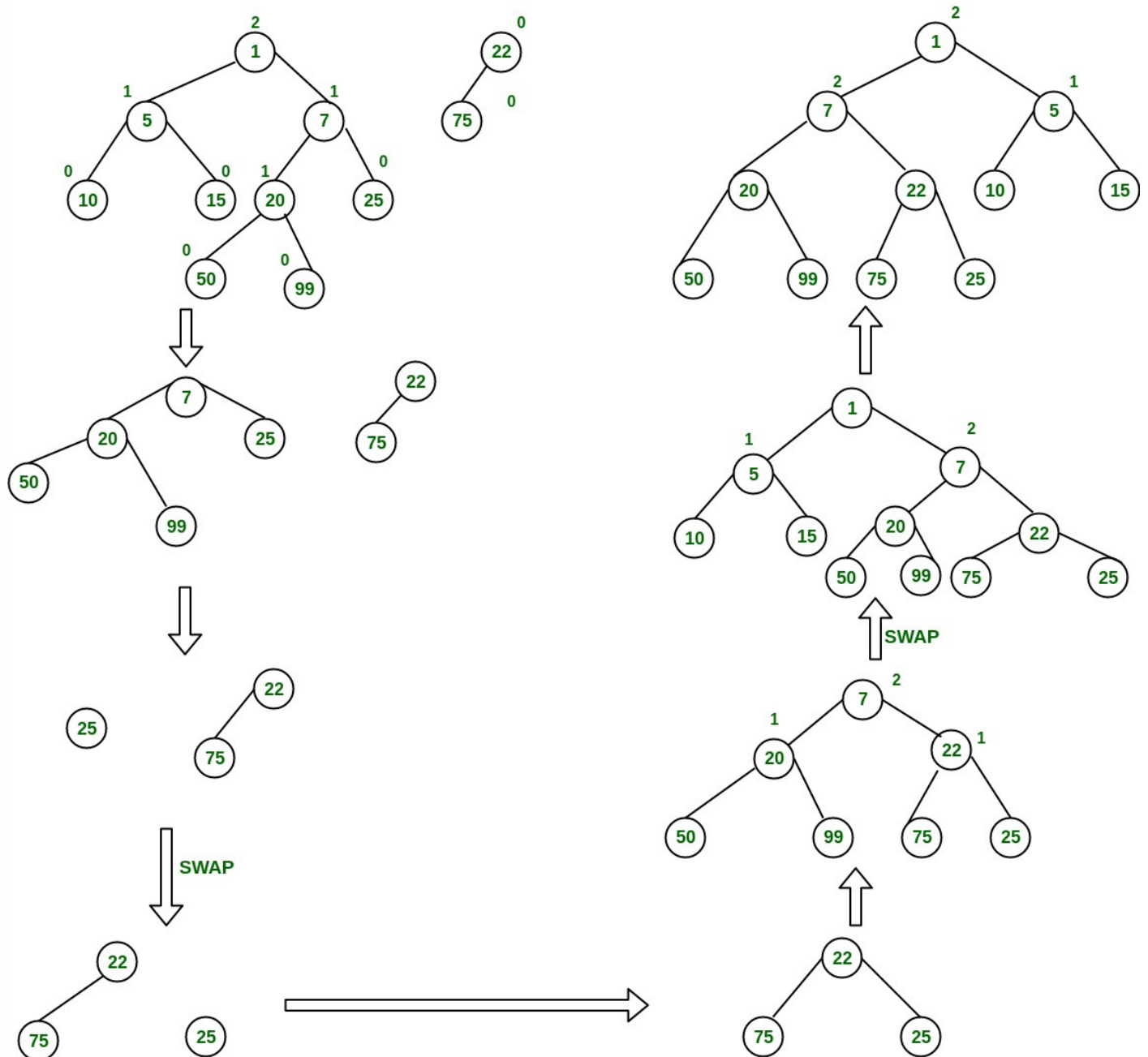




**Final leftist heap**

The worst case time complexity of this algorithm is  $O(\log n)$  in the worst case, where  $n$  is the number of nodes in the leftist heap.

**Another example of merging two leftist heap:**



### Implementation of leftist Tree / leftist Heap:

//C++ program for leftist heap / leftist tree

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

// Node Class Declaration

```
class LeftistNode
```

```
{
```

```
public:
```

```
    int element;
```

```
    LeftistNode *left;
```

```
    LeftistNode *right;
```

```
    int dist;
```



```

LeftistNode(int & element, LeftistNode *lt = NULL,
            LeftistNode *rt = NULL, int np = 0)
{
    this->element = element;
    right = rt;
    left = lt,
    dist = np;
}
};

```

//Class Declaration

```

class LeftistHeap
{
public:
    LeftistHeap();
    LeftistHeap(LeftistHeap &rhs);
    ~LeftistHeap();
    bool isEmpty();
    bool isFull();
    int &findMin();
    void Insert(int &x);
    void deleteMin();
    void deleteMin(int &minItem);
    void makeEmpty();
    void Merge(LeftistHeap &rhs);
    LeftistHeap & operator =(LeftistHeap &rhs);
private:
    LeftistNode *root;
    LeftistNode *Merge(LeftistNode *h1,
                      LeftistNode *h2);
    LeftistNode *Merge1(LeftistNode *h1,
                       LeftistNode *h2);

    void swapChildren(LeftistNode * t);
    void reclaimMemory(LeftistNode * t);
    LeftistNode *clone(LeftistNode *t);
};

```

// Construct the leftist heap

```

LeftistHeap::LeftistHeap()
{
    root = NULL;
}

```

// Copy constructor.

```

LeftistHeap::LeftistHeap(LeftistHeap &rhs)
{
    root = NULL;
    *this = rhs;
}

// Destruct the leftist heap
LeftistHeap::~LeftistHeap()
{
    makeEmpty( );
}

/* Merge rhs into the priority queue.
rhs becomes empty. rhs must be different
from this.*/
void LeftistHeap::Merge(LeftistHeap &rhs)
{
    if (this == &rhs)
        return;
    root = Merge(root, rhs.root);
    rhs.root = NULL;
}

/* Internal method to merge two roots.
Deals with deviant cases and calls recursive Merge1.*/
LeftistNode *LeftistHeap::Merge(LeftistNode * h1,
                                LeftistNode * h2)
{
    if (h1 == NULL)
        return h2;
    if (h2 == NULL)
        return h1;
    if (h1->element < h2->element)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}

/* Internal method to merge two roots.
Assumes trees are not empty, and h1's root contains
smallest item.*/
LeftistNode *LeftistHeap::Merge1(LeftistNode * h1,
                                  LeftistNode * h2)
{

```

```

        if (h1->left == NULL)
            h1->left = h2;
        else
        {
            h1->right = Merge(h1->right, h2);
            if (h1->left->dist < h1->right->dist)
                swapChildren(h1);
            h1->dist = h1->right->dist + 1;
        }
        return h1;
    }

// Swaps t's two children.
void LeftistHeap::swapChildren(LeftistNode * t)
{
    LeftistNode *tmp = t->left;
    t->left = t->right;
    t->right = tmp;
}

/* Insert item x into the priority queue, maintaining
heap order.*/
void LeftistHeap::Insert(int &x)
{
    root = Merge(new LeftistNode(x), root);
}

/* Find the smallest item in the priority queue.
Return the smallest item, or throw Underflow if empty.*/
int &LeftistHeap::findMin()
{
    return root->element;
}

/* Remove the smallest item from the priority queue.
Throws Underflow if empty.*/
void LeftistHeap::deleteMin()
{
    LeftistNode *oldRoot = root;
    root = Merge(root->left, root->right);
    delete oldRoot;
}

/* Remove the smallest item from the priority queue.

```

Pass back the smallest item, or throw Underflow if empty.\*/

```
void LeftistHeap::deleteMin(int &minItem)
{
    if (isEmpty())
    {
        cout<<"Heap is Empty"<<endl;
        return;
    }
    minItem = findMin();
    deleteMin();
}
```

/\* Test if the priority queue is logically empty.

Returns true if empty, false otherwise\*/

```
bool LeftistHeap::isEmpty()
{
    return root == NULL;
}
```

/\* Test if the priority queue is logically full.

Returns false in this implementation.\*/

```
bool LeftistHeap::isFull()
{
    return false;
}
```

// Make the priority queue logically empty

```
void LeftistHeap::makeEmpty()
{
    reclaimMemory(root);
    root = NULL;
}
```

// Deep copy

LeftistHeap &LeftistHeap::operator =(LeftistHeap & rhs)

```
{
    if (this != &rhs)
    {
        makeEmpty();
        root = clone(rhs.root);
    }
    return *this;
}
```

```

// Internal method to make the tree empty.
void LeftistHeap::reclaimMemory(LeftistNode * t)
{
    if (t != NULL)
    {
        reclaimMemory(t->left);
        reclaimMemory(t->right);
        delete t;
    }
}

// Internal method to clone subtree.
LeftistNode *LeftistHeap::clone(LeftistNode * t)
{
    if (t == NULL)
        return NULL;
    else
        return new LeftistNode(t->element, clone(t->left),
                                clone(t->right), t->dist);
}

//Driver program
int main()
{
    LeftistHeap h;
    LeftistHeap h1;
    LeftistHeap h2;
    int x;
    int arr[] = {1, 5, 7, 10, 15};
    int arr1[] = {22, 75};

    h.Insert(arr[0]);
    h.Insert(arr[1]);
    h.Insert(arr[2]);
    h.Insert(arr[3]);
    h.Insert(arr[4]);
    h1.Insert(arr1[0]);
    h1.Insert(arr1[1]);

    h.deleteMin(x);
    cout<< x <<endl;

    h1.deleteMin(x);
    cout<< x <<endl;
}

```

```

h.Merge(h1);
h2 = h;

h2.deleteMin(x);
cout<< x << endl;

return 0;
}

```

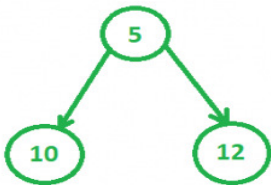
## Skew Heap

A **skew heap** (or self – adjusting heap) is a heap data structure implemented as a **binary tree**. Skew heaps are advantageous because of their ability to **merge more quickly** than binary heaps. In contrast with [binary heaps](#), there are no structural constraints, so there is no guarantee that the height of the tree is logarithmic. Only two conditions must be satisfied :

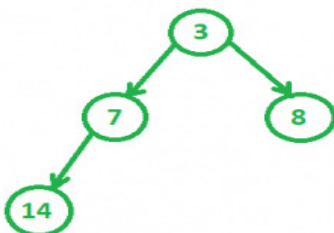
1. The general heap order must be there (root is minimum and same is recursively true for subtrees), but balanced property (all levels must be full except the last) is not required.
2. Main operation in Skew Heaps is Merge. We can implement other operations like insert, extractMin(), etc using Merge only.

### Example :

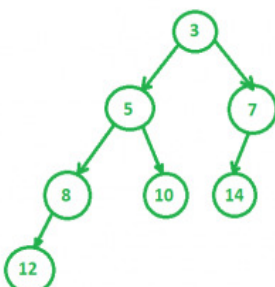
1. Consider the skew heap 1 to be



2. The second heap to be considered



4. And we obtain the final merged tree as



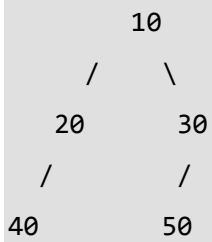
## Recursive Merge Process :

merge(h1, h2)

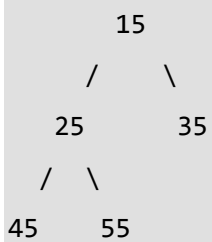
1. Let h1 and h2 be the two min skew heaps to be merged. Let h1's root be smaller than h2's root (If not smaller, we can swap to get the same).
2. We swap h1->left and h1->right.
3. h1->left = merge(h2, h1->left)

Examples :

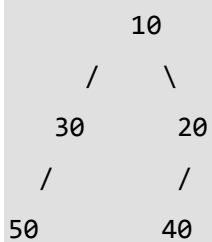
Let h1 be



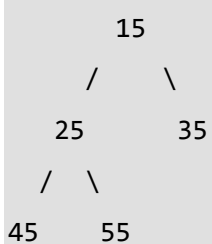
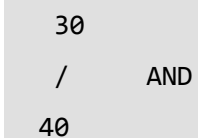
Let h2 be



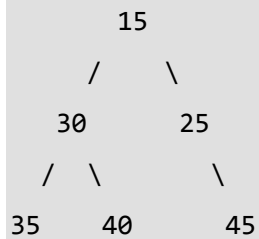
After swapping h1->left and h1->right, we get



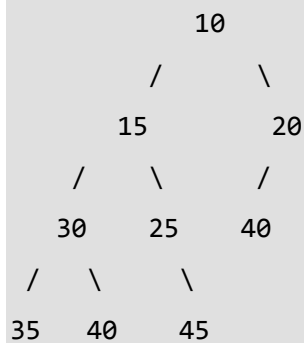
Now we recursively Merge



After recursive merge, we get (Please do it using pen and paper).



We make this merged tree as left of original h1 and we get following result.



```
// CPP program to implement Skew Heap
```

```
// operations.
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
struct SkewHeap
```

```
{
```

```
    int key;
```

```
    SkewHeap* right;
```

```
    SkewHeap* left;
```

```
    // constructor to make a new
```

```
    // node of heap
```

```
    SkewHeap()
```

```
    {
```

```
        key = 0;
```



```

        right = NULL;

        left = NULL;
    }

// the special merge function that's
// used in most of the other operations
// also
SkewHeap* merge(SkewHeap* h1, SkewHeap* h2)
{
    // If one of the heaps is empty
    if (h1 == NULL)
        return h2;

    if (h2 == NULL)
        return h1;

    // Make sure that h1 has smaller
    // key.
    if (h1->key > h2->key)
        swap(h1, h2);

    // Swap h1->left and h1->right
    swap(h1->left, h1->right);

    // Merge h2 and h1->left and make
    // merged tree as left of h1.
    h1->left = merge(h2, h1->left);

    return h1;
}

// function to construct heap using

```

```

// values in the array
SkewHeap* construct(SkewHeap* root,
                    int heap[], int n)
{
    SkewHeap* temp;
    for (int i = 0; i < n; i++) {
        temp = new SkewHeap;
        temp->key = heap[i];
        root = merge(root, temp);
    }
    return root;
}

// fucntion to print the Skew Heap,
// as it is in form of a tree so we use
// tree traversal algorithms
void inorder(SkewHeap* root)
{
    if (root == NULL)
        return;
    else {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
    return;
}
};

```

// Driver Code

```
int main()
```

```

{

// Construct two heaps

SkewHeap heap, *temp1 = NULL, *temp2 = NULL;

/*
    5
   /\
  /\
10  12 */
int heap1[] = { 12, 5, 10 };

/*
    3
   /\
  /\
 7   8
/
/
14 */
int heap2[] = { 3, 7, 8, 14 };
int n1 = sizeof(heap1) / sizeof(heap1[0]);
int n2 = sizeof(heap2) / sizeof(heap2[0]);
temp1 = heap.construct(temp1, heap1, n1);
temp2 = heap.construct(temp2, heap2, n2);
// Merge two heaps
temp1 = heap.merge(temp1, temp2);
/*
    3
   /\
  /\
 5   7
 /\  /
8 10 14
/
12 */
cout << "Merged Heap is: " << endl;
heap.inorder(temp1);

}

```

PARTHA SARATHI