# DESIGN OF ANALYSIS AND ALGORITHM

## ACTIVITY T3-PRIORITY QUEUE

GROUP MEMBERS(GROUP-6):

1929157-KUMAR PRATYUSH
1929172-SAURABH KUMAR PATHAK
1929174-SHANTANU ROY
1929256-YASH RAJ SHUKLA

Q1.)Given two sorted arrays A[] and B[] of sizes N and M respectively, the task is to merge them in a sorted manner.

->**Input: A[] = { 5, 6, 8 }, B[] = { 4, 7, 8 }**
**Output: 4 5 6 7 8 8**

implementation-Initialize a min priority queue say PQ to implement the Min Heap.
Traverse the array A[], and push all the elements of the array into the PQ.
Traverse the array B[], and push all the elements of the array into the PQ.
Now put all the elements of the PQ into an array, say res[] popping the top element of the PQ one by one.
Finally, we print the sorted array res[] as the answer.

C++ program for the above approach:

```
#include <bits/stdc++.h>
using namespace std;

// Function to merge two arrays
void merge(int A[], int B[], int N, int M)
{
    // Stores the merged array
    int res[N + M];

    // Create a min priority_queue
    priority_queue<int, vector<int>, greater<int> > pq;

    // Traverse the array A[]
    for (int i = 0; i < N; i++)
        pq.push(A[i]);

    // Traverse the array B[]
    for (int i = 0; i < M; i++)
        pq.push(B[i]);

    int j = 0;

    // Iterate until the
    // pq is not empty
```

```
    while (!pq.empty()) {

        // Stores the top element
        // of pq into res[j]
        res[j++] = pq.top();

        // Removes the top element
        pq.pop();
    }

    // Print the merged array
    for (int i = 0; i < N + M; i++)
        cout << res[i] << ' ';
}

int main()
{

    // Input
    int A[] = { 5, 6, 8 };
    int B[] = { 4, 7, 8 };

    int N = sizeof(A) / sizeof(A[0]);
    int M = sizeof(B) / sizeof(B[0]);

    // Function call
    merge(A, B, N, M);

    return 0;
}
```

Time Complexity: O((N+M)*log(N+M))
Auxiliary Space: O(N+M)

## Q2.)HOW PRIORITY QUEUE IS IMPLEMENTED USING BINARY HEAPS?

->One important variation of the queue is the priority queue. A priority queue acts like a queue in that items remain in it for some time before being dequeued. However, in a priority queue the logical order of items inside a queue is

determined by their “priority”. Specifically, the highest priority items are retrieved from the queue ahead of lower priority items.

The classic way to implement a priority queue is using a data structure called a binary heap. A binary heap will allow us to enqueue or dequeue items in O(\ log{n})O(logn).
The basic operations we will implement for our binary heap are:

BinaryHeap() creates a new, empty, binary heap.
insert(k) adds a new item to the heap.
find_min() returns the item with the minimum key value, leaving item in the heap.
del_min() returns the item with the minimum key value, removing the item from the heap.
is_empty() returns true if the heap is empty, false otherwise.
size() returns the number of items in the heap.
build_heap(list) builds a new heap from a list of keys.

## Q3.)Why is Binary Heap Preferred over BST for Priority Queue?

->A typical Priority Queue requires following operations to be efficient.

1. Get Top Priority Element (Get minimum or maximum)
2. Insert an element
3. Remove top priority element
4. Decrease Key

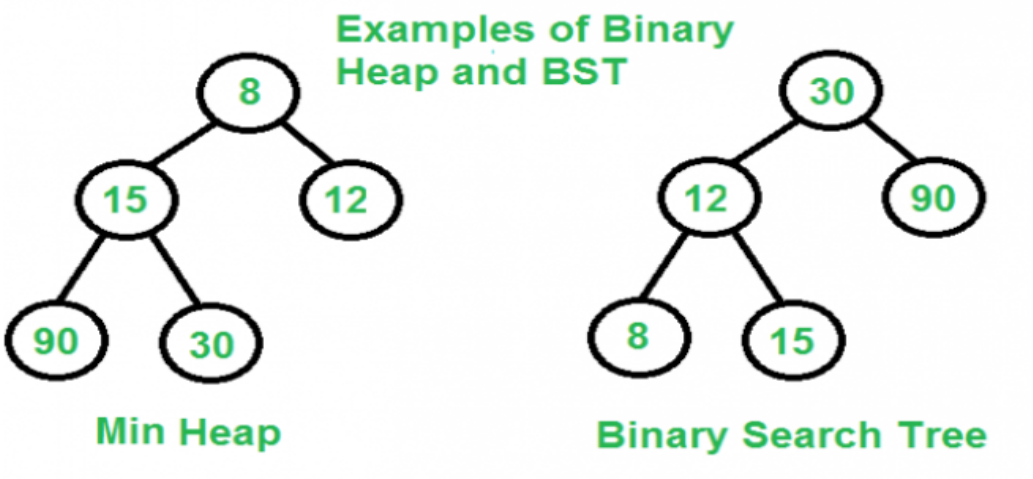
A Binary Heap supports above operations with following time complexities:

1. O(1)
2. O(Logn)
3. O(Logn)
4. O(Logn)

A Self Balancing Binary Search Tree like AVL Tree , Red black tree etc can also support above operations with same time complexities.

1. Finding minimum and maximum are not naturally O(1), but can be easily implemented in O(1) by keeping an extra pointer to minimum or maximum and updating the pointer with insertion and deletion if required. With deletion we can update by finding inorder predecessor or successor.

2. Inserting an element is naturally O(Logn)
3. Removing maximum or minimum are also O(Logn)
4. Decrease key can be done in O(Logn) by doing a deletion followed by insertion.




**So why is Binary Heap Preferred for Priority Queue?**

- Since Binary Heap is implemented using arrays, there is always better locality of reference and operations are more cache friendly.
- Although operations are of same time complexity, constants in Binary Search Tree are higher.
- We can build a Binary Heap in O(n) time. Self Balancing BSTs require O(nLogn) time to construct.
- Binary Heap doesn't require extra space for pointers.
- Binary Heap is easier to implement.
- There are variations of Binary Heap like Fibonacci Heap that can support insert and decrease-key in Θ(1) time

----------x----------x----------