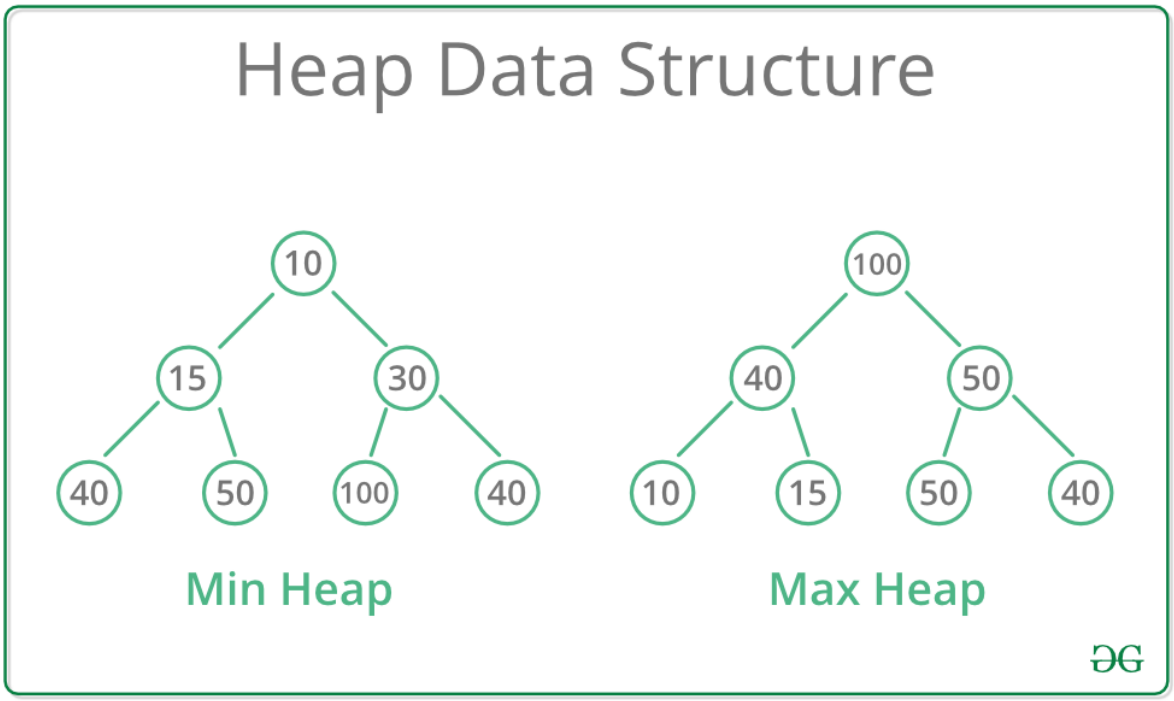
Heap Data Structure

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



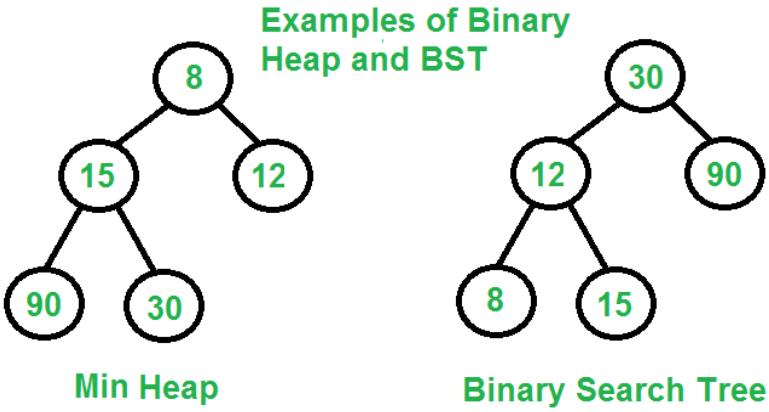
Why is Binary Heap Preferred over BST for Priority Queue?

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)



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

Is Binary Heap always better?

Although Binary Heap is for Priority Queue, BSTs have their own advantages and the list of advantages is in-fact bigger compared to binary heap.

- Searching an element in self-balancing BST is O(Logn) which is O(n) in Binary Heap.
- We can print all elements of BST in sorted order in O(n) time, but Binary Heap requires O(nLogn) time.
- Floor and ceil can be found in O(Logn) time.
- K'th largest/smallest element be found in O(Logn) time by augmenting tree with an additional field.

Applications of Priority Queue

A Priority Queue is different from a normal queue, because instead of being a "first-in-first-out", values come out in order by priority. It is an abstract data type that captures the idea of a container whose elements have "priorities" attached to them. An element of highest priority always appears at the front of the queue. If that element is removed, the next highest priority element advances to the front.

APPLICATIONS:

Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.

Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.

Data compression : It is used in Huffman codes which is used to compresses data.

Artificial Intelligence : A* Search Algorithm : The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first. The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.

Heap Sort : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.

Operating systems: It is also use in Operating System for load balancing (load balancing on server), interrupt handling.

IMPLEMENTATION OF PRIORITY QUEUES:

MIN HEAP:

```
#include <vector>


class PriorityQueue {
        vector<int> pq;
        public :
        bool isEmpty() {
                return pq.size() == 0;
        }

        // Return the size of priorityQueue - no of elements present
        int getSize() {
                return pq.size();
        }
        int getMin() {
                if(isEmpty()) {
                        return 0;                       // Priority Queue is empty
                }
                return pq[0];
        }
```

```
void insert(int element) {
	pq.push_back(element);
	int childIndex = pq.size() - 1;
	//up-heapify
	while(childIndex > 0) {
		int parentIndex = (childIndex - 1) / 2;
		if(pq[childIndex] < pq[parentIndex]) {
			int temp = pq[childIndex];
			pq[childIndex] = pq[parentIndex];
			pq[parentIndex] = temp;
		}
		else {
			break;
		}
		childIndex = parentIndex;
	}

}

int removeMin() {
	if(isEmpty()) {
		return 0;			// Priority Queue is empty
	}
	int ans = pq[0];
	pq[0] = pq[pq.size() - 1];
	pq.pop_back();

	// down-heapify

	int parentIndex = 0;
	int leftChildIndex = 2 * parentIndex + 1;
```

```
                int rightChildIndx = 2 * parentIndex + 2;

                while(leftChildIndex < pq.size()) {
                        int minIndex = parentIndex;
                        if(pq[minIndex] > pq[leftChildIndex]) {
                                minIndex = leftChildIndex;
                        }
                        if(rightChildIndx < pq.size() && pq[rightChildIndx] < pq[minIndex]) {
                                minIndex = rightChildIndx;
                        }
                        if(minIndex == parentIndex) {
                                break;
                        }
                        int temp = pq[minIndex];
                        pq[minIndex] = pq[parentIndex];
                        pq[parentIndex] = temp;

                        parentIndex = minIndex;
                        leftChildIndex = 2 * parentIndex + 1;
                        rightChildIndx = 2 * parentIndex + 2;
                }

                return ans;
        }
};
#include <iostream>
using namespace std;
int main() {
        PriorityQueue p;
        p.insert(1);
        p.insert(19);
```

```
        p.insert(14);
        p.insert(98);
        p.insert(143);
        p.insert(2;
        p.insert(6);
        cout << p.getSize() << endl;
        cout << p.getMin() << endl;
        while(!p.isEmpty()) {
                cout << p.removeMin() << " " ;
        }
        cout << endl
}
```

MAX HEAP:

```
#include <vector>
class PriorityQueue {
        vector<int> pq;
        public :
        bool isEmpty() {
                return pq.size() == 0;
        }
        // Return the size of priorityQueue - no of elements present
        int getSize() {
                return pq.size();
        }
        int getMax() {
                if(isEmpty()) {
                        return 0;                       // Priority Queue is empty
                }
                return pq[0];
        }
```

```
void insert(int element) {
	pq.push_back(element);
	int childIndex = pq.size() - 1;
	while(childIndex > 0) {
		int parentIndex = (childIndex - 1) / 2;
		if(pq[childIndex] > pq[parentIndex]) {
			int temp = pq[childIndex];
			pq[childIndex] = pq[parentIndex];
			pq[parentIndex] = temp;
		}
		else {
			break;
		}
		childIndex = parentIndex;
	}

}
int removeMax() {
	if(isEmpty()) {
		return 0;			// Priority Queue is empty
	}
	int ans = pq[0];
	pq[0] = pq[pq.size() - 1];
	pq.pop_back();

	// down-heapify

	int parentIndex = 0;
	int leftChildIndex = 2 * parentIndex + 1;
	int rightChildIndex = 2 * parentIndex + 2;
```

```
            while(leftChildIndex < pq.size()) {
                int maxIndex = parentIndex;
                if(pq[maxIndex] <pq[leftChildIndex]) {
                    maxIndex = leftChildIndex;
                }
                if(rightChildIndex < pq.size() && pq[rightChildIndx] > pq[maxIndex]) {
                    maxIndex = rightChildIndex;
                }
                if(maxIndex == parentIndex) {
                    break;
                }
                int temp = pq[maxIndex];
                pq[maxIndex] = pq[parentIndex];
                pq[parentIndex] = temp;

                parentIndex = maxIndex;
                leftChildIndex = 2 * parentIndex + 1;
                rightChildIndx = 2 * parentIndex + 2;
            }

            return ans;
        }
};
```

INPLACE HEAP SORT:

```
#include <iostream>
using namespace std;
void inplaceHeapSort(int pq[], int n) {
```

```
// Build the heap in input array
for(int i = 1; i < n; i++) {
	int childIndex = i;
	while(childIndex > 0) {
		int parentIndex = (childIndex - 1) / 2;

		if(pq[childIndex] < pq[parentIndex]) {
			int temp = pq[childIndex];
			pq[childIndex] = pq[parentIndex];
			pq[parentIndex] = temp;
		}
		else {
			break;
		}
		childIndex = parentIndex;
	}
}
// Remove elements
int size = n;
while(size > 1) {
	int temp = pq[0];
	pq[0] = pq[size - 1];
	pq[size-1] = temp;

	size--;

	int parentIndex = 0;
	int leftChildIndex = 2 * parentIndex + 1;
	int rightChildIndex = 2 * parentIndex + 2;
```

```
            while(leftChildIndex < size) {
                int minIndex = parentIndex;
                if(pq[minIndex] > pq[leftChildIndex]) {
                    minIndex = leftChildIndex;
                }
                if(rightChildIndex < size && pq[rightChildIndex] < pq[minIndex]) {
                    minIndex = rightChildIndx;
                }
                if(minIndex == parentIndex) {
                    break;
                }
                int temp = pq[minIndex];
                pq[minIndex] = pq[parentIndex];
                pq[parentIndex] = temp;

                parentIndex = minIndex;
                leftChildIndex = 2 * parentIndex + 1;
                rightChildIndex = 2 * parentIndex + 2;
            }
        }
}


int main() {
        int input[] = {4, 10, 21, 43, 100,6};
        inplaceHeapSort(input, 6);
        for(int i = 0; i < 6; i++)
        {
```

```
            cout << input[i] << " ";
        }
        cout << endl;
}
```

INBUILT PRIORITY QUEUE IN STL:

Methods of priority queue are:

- priority_queue::empty() in C++ STL– **empty()** function returns whether the queue is empty.
- priority_queue::size() in C++ STL– **size()** function returns the size of the queue.
- priority_queue::top() in C++ STL– Returns a reference to the top most element of the queue
- priority_queue::push() in C++ STL**– push(g)** function adds the element 'g' at the end of the queue.
- priority_queue::pop() in C++ STL– **pop()** function deletes the first element of the queue.
- priority_queue::swap() in C++ STL– This function is used to swap the contents of one priority queue with another priority queue of same type and size.
- priority_queue::emplace() in C++ STL – This function is used to insert a new element into the priority queue container, the new element is added to the top of the priority queue.
- priority_queue value_type in C++ STL– Represents the type of object stored as an element in a priority_queue. It acts as a synonym for the template parameter.

It supports the usual push(), pop(), top() etc operations, but is specifically designed so that its first element is always the greatest of the elements it contains, i.e. max heap.

In STL, priority queues take three template parameters:

```
template <class T,
        class Container = vector<T>,
        class Compare = less<typename Container::value_type>
        >
class priority_queue;
```

- The first element of the template defines the class of each element. It can be user-defined classes or primitive data-types. Like in you case

it can be int, float or double.

- The second element defines the container to be used to store the elements. The standard container classes std::vector and std::dequeue fulfil these requirements. It is usually the vector of the class defined in the first argument. Like in your case it can be vector<int>, vector<float>, vector<double>.

- The third element is the comparative class. By default it is less<T> but can be changed to suit your need. For min heap it can be changed to greater<T>

The priority_queue uses the function inside Comp class to maintain the elements sorted in a way that preserves *heap properties*(i.e., that the element popped is the last according to this *strict weak ordering*).

Example: (BY DEFAULT THE INBUILT PQ IS OF MAX HEAP TYPE)

```
#include <iostream>

using namespace std;

#include <queue>

int main() {

        priority_queue<int> p;

        p.push(100);

        p.push(21);

        p.push(7);

        p.push(165);

        p.push(78);

        p.push(4);


        cout << p.size() << endl;

        cout << p.empty() << endl;

        cout << p.top() << endl;
```

```
	while(!p.empty()) {
		cout << p.top() << endl;
		p.pop();
	}
}
```

Example:(WHEN WE WANT TO USE INBUILT MIN HEAP)

```
#include <iostream>
using namespace std;
#include <queue>



int main() {
	priority_queue<int, vector<int>, greater<int> > p;

	p.push(100);
	p.push(21);
	p.push(7);
	p.push(165);
	p.push(78);
	p.push(4);

	cout << p.size() << endl;
	cout << p.empty() << endl;
	cout << p.top() << endl;

	while(!p.empty()) {
		cout << p.top() << endl;
		p.pop();
	}
```

```
}
```

EXAMPLE: (Inplace heap sort by using PQ (STL))

```
#include<queue>
#include<vector>
using namespace std;
void inplaceHeapSort(int input[], int n)
{
  priority_queue<int> pq;
  for(int i=0; i<n; i++)
  {
    pq.push(input[i]);
  }

  for(int i=0 ; i<n ;i++)
  {
    input[i] = pq.top();
    pq.pop();
  }
}
int main() {

        int size;
        cin >> size;
        int *input = new int[1 + size];
        for(int i = 0; i < size; i++)
                cin >> input[i];
```

```
	inplaceHeapSort(input, size);


	for(int i = 0; i < size; i++)
		cout << input[i] << " ";
	return 0;
}
```

EXAMPLE: (K sorted array):

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in O(n log k) time. For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array. It may be assumed that k < n.

```
Input : arr[] = {6, 5, 3, 2, 8, 10, 9}
            k = 3
Output : arr[] = {2, 3, 5, 6, 8, 9, 10}


Input : arr[] = {10, 9, 8, 7, 4, 70, 60, 50}
         k = 4
Output : arr[] = {4, 7, 8, 9, 10, 50, 60, 70}
```

A **simple solution** is to sort the array using any standard sorting algorithm. The time complexity of this solution is O(n Log n)
A better solution is to use priority queue (or heap data structure).

```
#include <iostream>

using namespace std;

#include <queue>


void kSortedArray(int input[], int n, int k) {
	priority_queue<int> pq;
	for(int i = 0; i < k; i++) {
		pq.push(input[i]);
```

```
        }

        int j = 0;
        for(int i = k; i < n; i++) {
                input[j] = pq.top();
                pq.pop();
                pq.push(input[i]);
                j++;
        }
        while(!pq.empty()) {
                input[j] = pq.top();
                pq.pop();
        }
}
int main() {
        int input[] = {9, 11, 43, 21, 90,42};
        int k = 3;
        kSortedArray(input, 6, k);
        for(int i = 0; i < 6; i++) {
                cout << input[i] << " ";
        }
}
```

EXAMPLE: (K SMALLEST ELEMENTS):

You are given with an integer k and an array of integers that contain numbers in random order. Write a program to find k smallest numbers from given array.

```
#include <iostream>
#include<queue>
Using namespace std;
```

```
vector<int> kSmallest(int *input, int n, int k){
  priority_queue<int> pq;
  vector<int> output;
  for(int i=0; i<k; i++)
    pq.push(input[i]);

  for(int i=k;i<n;i++){
     if(input[i]<pq.top()){
        pq.pop();
        pq.push(input[i]);
     }
  }
    while(!pq.empty()){
       output.push_back(pq.top());
       pq.pop();
       }

    return output;
  }
#include <vector>
int main() {
          int n;
          cin >> n;
          int *input = new int[n];
          for(int i = 0; i < n; i++){
                    cin >> input[i];
          }
          int k;
          cin >> k;
```

```
        vector<int> output = kSmallest(input, n, k);
        for(int i = 0; i < output.size(); i++){
                cout << output[i] << endl;
        }
}
```

EXAMPLE: (K LARGEST ELEMENTS):

You are given with an integer k and an array of integers that contain numbers in random order. Write a program to find k largest numbers from given array.

```
#include<iostream>
#include<queue>
#include<vector>
Using namespace std;
vector<int> kLargest(int input[], int n, int k){

 priority_queue<int, vector<int>, greater<int> > pq;
 int i = 0;
 for(; i<k; i++)
 {
  pq.push(input[i]);
 }

 for(; i<n;i++)
 {
  if(pq.top() < input[i])
  {
   pq.pop();
   pq.push(input[i]);
  }
```

```
 }
 vector<int> v;
  int j=0;
  while(j<k)
  {
    v.push_back(pq.top());
    pq.pop();
    j++;
  }
  return v;
}
```

EXAMPLE: (CHECK MAX HEAP):

Given an array of integers, check whether it represents max-heap or not.

```
#include <iostream>
using namespace std;

bool checkMaxHeap(int arr[], int n)
{

  int max= arr[0];
  for(int i=0; i<n ; i++)
  {
    if(arr[i] > max)
      return false;
```

```
  }
  return true;
}
int main() {
    int n;
    cin>>n;
    int *arr = new int[n];
    for(int i=0; i<n; i++){
        cin >> arr[i];
    }
    bool ans = checkMaxHeap(arr, n);
    if(ans)
        cout << "true" << endl;
    else
        cout << "false" << endl;
   delete [] arr;
}
```

EXAMPLE: (Merge K sorted arrays):

Given k different arrays, which are sorted individually (in ascending order). You need to merge all the given arrays such that output array should be sorted (in ascending order).

```
#include <iostream>
#include<queue>
#include<vector>
Using namespace std;
vector<int> mergeKSortedArrays(vector<vector<int>*> input)
{
 priority_queue <int, vector<int>, greater<int>> pq;
```

```
 for(int i=0; i< input.size(); i++)
 {
  for(int j=0; j<input[i]->size(); j++)
    pq.push((*input[i])[j]);
 }
 vector<int> v;
 while(pq.size())
 {
  v.push_back(pq.top());
  pq.pop();
 }
 return v;
 }
int main() {

          int k;
          cin >> k;
          vector<vector<int>*> input;
  for(int j = 1; j <= k; j++) {
      int size;
      cin >> size;
      vector<int> *current = new vector<int>;

      for(int i = 0; i < size; i++) {
         int a;
         cin >> a;
         current -> push_back(a);
      }
      input.push_back(current);
```

```
    }
  vector<int> output = mergeKSortedArrays(input);


        for(int i = 0; i < output.size(); i++)
                cout << output[i] << " ";


        return 0;
}
```

EXAMPLE: (Kth largest element):

Given an array A of random integers and an integer k, find and return the kth largest element in the array.

```
#include <iostream>
#include<queue>
#include<vector>
using namespace std;
int kthLargest (vector<int> arr, int n, int k)
{

 priority_queue<int, vector<int>, greater<int> > pq;
 int i=0;
 for(; i<k;i++)
   pq.push(arr[i]);

 for(; i<n;i++)
 {
   if(pq.top() < arr[i])
   {
```

```
      pq.pop();
      pq.push(arr[i]);
    }
  }
   return pq.top();
}
int main(){
    int n, k, s;
    vector<int> arr;
    cin>>n;
    for(int i = 0; i < n; i++){
        cin>>s;
        arr.push_back(s);
    }
    cin >> k;
    cout << kthLargest(arr, n, k) << endl;
}
```

EXAMPLE(Buy the ticket):

You want to buy a ticket for a well-known concert which is happening in your city. But the number of tickets available is limited. Hence the sponsors of the concert decided to sell tickets to customers based on some priority.

A queue is maintained for buying the tickets and every person has attached with a priority (an integer, 1 being the lowest priority). The tickets are sold in the following manner -

1. The first person (pi) in the queue asked to comes out.

2. If there is another person present in the queue who has higher priority than pi, then ask pi to move at end of the queue without giving him the ticket.

3. Otherwise, give him the ticket (and don't make him stand in queue again).

Giving a ticket to a person takes exactly 1 minutes and it takes no time for removing and adding a person to the queue. And you can assume that no new person joins the queue.

Given a list of priorities of N persons standing in the queue and the index of your priority (indexing starts from 0). Find and return the time it will take until you get the ticket.

Input Format :

Line 1 : Integer N (Total number of people standing in queue)

Line 2 : Priorities of every person (n space separated integers)

Line 3 : Integer k (index of your priority)

Output Format :

Time required

Sample Input 1 :

3

3 9 4

2

Sample Output 1 :

2

Sample Output 1 Explanation :

Person with priority 3 comes out. But there is a person with higher priority than him. So he goes and then stands in the queue at the end. Queue's status : {9, 4, 3}. Time : 0 secs.

Next, the person with priority 9 comes out. And there is no person with higher priority than him. So he'll get the ticket. Queue's status : {4, 3}. Time : 1 secs.

Next, the person with priority 4 comes out (which is you). And there is no person with higher priority than you. So you'll get the ticket. Time : 2 secs.

```
#include <iostream>
#include<queue>
void shift(int *arr, int n)
{
        int ans = arr[0];
        for(int i=0; i<n; i++)
          arr[i] = arr[i + 1];
          arr[n-1] = ans;
```

```
}
int buyTicket (int *arr, int n, int k){
  int me= arr[k];
 int count=0;
 priority_queue<int, vector<int>, less<int>> pq;

 for(int i=0; i<n; i++)
 {
  pq.push(arr[i]);
 }

 while(pq.top() != me)
 {
   if(pq.top() != arr[0])
     shift(arr, n);
  else if(pq.top()==arr[0])
  {
    pq.pop();
    count++;
  }
}
return count + 1;
}
 int main(){
   int n, k, s;
   cin>>n;
   int *arr = new int[n];
   for(int i = 0; i < n; i++){
```

```
        cin >> arr[i];
    }
    cin >> k;
    cout << buyTicket(arr, n, k) << endl;
}
```