Priority Queue in C++

A **priority_queue** in C++ is always implemented using a heap data structure. In C++ this default priority queue we create is always MAX_HEAP. The maximum element by default is always at the top. Priority queues are a type of container adapters, specifically designed such that the first element of the queue is the greatest of all elements in the queue and elements are in non-increasing order.

Syntax:

```
priority_queue pq;


This creates an empty priority_queue pq

to accept integer values.
```

Few important functions of Priority Queue:

1. **priority_queue::push()** - The **push()** function is used to insert an element in the priority queue. The element is added to the priority queue container and the size of the queue is increased by 1. Firstly, the element is added at the back and at the same time the elements of the priority queue reorder themselves according to priority. By default, the maximum element is brought to the top.

   Syntax:

   ```
   priority_queue.push(value)
   ```

   Parameters: The value of the element to be inserted is passed as the parameter.

   Examples:

   ```
   Input: pqueue = {5, 2, 1}

   Operation: pqueue.push(3);

   Output: 5, 3, 2, 1
   ```

2. **priority_queue::pop()** - The **pop()** function is used to remove the top element from the priority queue.
   Syntax:

   ```
   priority_queue.pop()
   ```

   Parameters: No parameters are passed.

   Examples:

   ```
   Input: pqueue = {5, 3, 2, 1}
   Operation: pqueue.pop();
   Output: pqueue = {3, 2, 1}
   ```

3. **priority_queue::top()** - The **top()** function is used to reference the top(or the largest) element of the priority queue.
   Syntax:

   ```
   priority_queue.top()
   ```

   Parameters: No value is needed to pass as the parameter.
   Return Value: Direct reference to the top(or the largest) the element of the priority queue container.

   Examples:

   ```
   Input: pqueue.push(5);
          pqueue.push(1);
          pqueue.top();
   Output: 5

   Input: pqueue.push(5);
          pqueue.push(1);
          pqueue.push(7);
          pqueue.top();
   ```

```
        pqueue.top();
Output: 7
```

4. **priority_queue::empty()** - The **empty()** function is used to check if the priority queue container is empty or not.
   Syntax:

   ```
   pqueuename.empty()
   ```

   Parameters: No parameters are passed
   Returns: The method returns True, if priority queue is empty, else False.

   Examples:

   ```
   Input :  pqueue = {3, 2, 1}

          pqueue.empty();

   Output : False



   Input :  pqueue

          pqueue.empty();

   Output : True
   ```

Program 1: This program shows the implementation of PriorityQueue along with various important above mentioned functions.

```

// CPP code to illustrate
// priorit_queue and its
// various function
#include <iostream>
#include <queue>
using namespace std;

// Drivers Method
int main()
{

    // Creating a priority queue
    priority_queue<int> pq;

    // Pushing elements into
    // the priority_queue
    pq.push(10);
    pq.push(15);
    pq.push(5);

    // Displaying th e size of
    // the queue
    cout << pq.size() << " ";

    // Displaying the top elements
    // of the queue
    cout << pq.top() << " ";

    while (pq.empty() == false) {
```

Run

Output:

```
3 15 15 10 5
```

Program 2: Creating a priority_queue where the minimum element is always at the top (construction of a MIN_HEAP).

Syntax:

```
priority_queue,
           greater> pq;
```

Parameters: In the syntax, the *int* represents the data type of the elements of the priority queue. We have also passed a *vector*, which is *used by the priority_queue as an underlined container.* The priority_queue uses the *concept of Heap* but it is built on vector or a dynamic array as a base. Here the *greater()* function has been used to revere the order of the Heap. This converts the MAX_HEAP into a MIN_HEAP.

```

// CPP code to illustrate
// priority_queue with MIN_HEAP
#include <iostream>
#include <queue>
using namespace std;

// Drivers Method
int main()
{

    // Creating a priority queue
    priority_queue<int, vector<int>,
                          greater<int> >
        pq;

    // Pushing elements into
    // the priority_queue
    pq.push(10);
    pq.push(15);
    pq.push(5);

    // Displaying th e size of
    // the queue
    cout << pq.size() << " ";

    // Displaying the top elements
    // of the queue
    cout << pq.top() << " ";

```

Run

Output:

```
3 5 5 10 15
```

Program 3: Implementing priority queue with an existing vector or an array.
Syntax:

```
priority_queue

pq(begin_iterator, last_iterator);
```

Parameters: Here *begin_iterator* and *last_iterator* refers to the first and the last position of the array respectively. Here the last position refers to the index beyond the last element.

```

// CPP code to illustrate
// priority_queue
#include <iostream>
#include <queue>
using namespace std;

// Drivers Method
int main()
{
    // Sample array
    int arr[] = { 10, 5, 15 };

    // Creating priority_queue
    // out of the array
    priority_queue<int> pq(arr, arr + 3);

    // Displaying the priority_queue
    while (pq.empty() == false) {
        cout << pq.top() << " ";
        pq.pop();
    }
}

```

Run

Output:

```
15 10 5
```

Note: Internally, the priority_queue calls make_heap() function and creates a MAX_HEAP by default out of the passed array. This method is more optimal than pushing each element into the queue as the construction of the heap takes O(n) time, where n is the number of elements in the array.

Time Complexities:

- top(): O(1) as the top element can be accessed directly in constant time.
- empty(): O(1) as the checking can be done in constant time.
- push(): O(logn)
- pop(): O(logn)

Program 4: Alternate way to create a MIN_HEAP using a simple method. The trick lies in converting all the array elements into its negative counterpart. This code uses the syntax of MAX_HEAP but the effect is seen that of MIN_HEAP.

```

// CPP code to illustrate
// priority_queue and MIN_HEAP
#include <iostream>
#include <queue>
using namespace std;

// Drivers Method
int main()
{
    // Sample array
    int arr[] = { 10, 5, 15 };
    int n = 3;

    // Multiplying the array elements
    // with -1
    for (int i = 0; i < n; i++)
        arr[i] = -arr[i];

    // Creating priority_queue
    // out of the array
    priority_queue<int> pq(arr, arr + 3);

    // Displaying the priority_queue
    while (pq.empty() == false) {
        cout << (-pq.top()) << " ";
        pq.pop();
    }
}

```

Run

Output:

```
5 10 15
```

Note: If you create a priority queue of pairs, elements would be considered according to the first element of pairs which is the default method, be it sorting, priority_queue, map or unordered map of pairs. To make any change to the order, you can use the customised comparator function.

Program 5: Taking the age and height of a person as a structure, write a program to create a priority queue of persons such that all the person can be traversed in decreasing order of heights.

Example:

```
Input:
Person p1: age - 21
           height - 5

Person p2: age - 22
           height - 6

Person p3: age - 23
           height - 7

Output: 7 6 5
```

```
// CPP code to illustrate// priorit_queue and its// various function#include <iostream>#include <queue>
```

Run

Output:

```
7 6 5
```

Applications: The priority_queue can be used in various standard algorithms like,

- Dijkstra's Algorithm
- Prims Algorithm
- Huffman Algorithm
- Heap Sort
- Any other place where the heaps can be used.

Sample Problem: Sorting an array using priority queue

Problem: Given an array arrange them in an increasing order using the priority_queue.

Examples:

```
Input: arr[] = {1, 3, 7, 2, 4}
       n = 5
Output: {1, 2, 3, 4, 7}

Input: arr[] = {2, 7, 1, 3, 9, 5, 4}
```

```
Output: {1, 2, 3, 4, 5, 7, 9}
```

Approach: Since we are using priority_queue to sort the array, we know that priority_queue stores maximum element at the top of the heap or at the root of the tree. For further detail and working of important functions associated with priority_queue, you can go through this link.

> **Step 1:** Initially we'll insert all the elements of array from index 0 to n-1 in heap or priority_queue. Now the elements of the array have been stored in heap in decreasing order.
>
> **Step 2:** Remove topmost element from priority_queue and store it in array at (n-1)th position.
>
> **Step 3:** Repeat the process of *Step2.*

Time complexity: Time taken in Step1 is O(N) because constructing heap from a given array or vector take O(N) time. Removing an element from the top of the heap takes O(logN) time since we are performing this N times, thus the overall time taken in removing and storing in an array takes O(NlogN).

```cpp

// Program to sort the array
// using priority_queue
#include<bits/stdc++.h>
using namespace std;

int main()
{
    // The array to sort
    int arr[] = {1, 3, 7, 2, 4};

    // Size of the array
    int n = 5;

    // Creating the priority_queue
    priority_queue<int>pq(arr, arr+n);
    int i = n - 1;
    int x;

    // Sorting the elements
    while(i >= 0)
    {
        arr[i] = pq.top();
        pq.pop();
        i--;
    }

    // Displaying the array
    for(int i=0; i<n;i++)
        cout<<arr[i]<<" ";
```

Run

- Sample Problem : Kth largest element in an array

Problem: Given an array and an integer **k**, find the K-th largest element in the array using priority queue.

Examples:

```
Input: arr[] = {1, 9, 7, 2, 4}
        n = 5
        k = 3
Output: 4
Explanation: The 3rd largest element
in the array is 4.
```

Approach: Since we are using priority_queue to sort the array, we know that priority_queue stores maximum element at the top of the heap or at the root of the tree. So we just need to pop the elements K times to get the K-th largest element. For more detail and working of important functions associated with priority_queue, you can go through the lectures in priority_queue track of the course.

Step 1: Initially we'll insert all the elements of array from index 0 to n-1 in heap or priority_queue. Now the elements of the array have been stored in heap in decreasing order.

Step 2: Pop the top elements from the priority_queue.

Step 3: Repeat *Step2* K-times to get the K-th largest element.

Time complexity: Time taken in Step1 is O(N) because constructing heap from a given array or vector take O(N) time. Removing an element from the top of the heap in Step2 takes O(logN) time and since we are performing this K-times, the overall time taken will be **O(KlogN) times**. In the worst case, this would be similar to O(NlogN) times.

```cpp

// Program to find the K-th largest element
#include<bits/stdc++.h>
using namespace std;

// Function to find the k-th largest element
int find_kth_largest(int arr[], int n, int k)
{

    // Creating the priority_queue out
    priority_queue<int> pq(arr, arr+n);
    int x;

    // Finding the k-th largest element
    while (k > 0)
    {
        x = pq.top(); pq.pop();
        k--;
    }
    return x;
}

// Driving code
int main()
{
    // Base array
    int arr[] = {1, 9, 7, 2, 4};

    // Size and value of k
    int n = 5, k = 3;
```

Run

Output:

```
4
```

- Sample Problem : Find k most frequent numbers

Problem: Given an array of size n and a value k, the task is to find k numbers with most occurrences. If there are similar occurrences, then the priority will be given to the lesser number in the number line.

Examples:

```
Input: arr[] = {3, 1, 4 , 4, 5, 2, 6, 1}
         n = 8
         k = 2
Output: 1 4
Explanation: As 1 and 4 occurs 2 times

Input: arr[] = {1, 1, 4, 4}
         n = 8
         k = 1
Output: 1
Explanation: Although both 1 and 4 occurs 2 times,
but priority would be given to the lesser number
```

```
which is 1 in this case.
```

Approach: We will be using the concept of Unordered Map, Vector and Priority Queue to solve this problem efficiently. The logic is to find the number of times each element is appearing in the given array and storing them in an unordered_map. Now create a Vector of pair out of this unordered_map by assigning the value as the first element of the pair and the frequency of it occurring as the second. Next, we need to convert this Vector pair into a priority_queue and store them in a MAX_HEAP wherein the arrangements would be done based on the frequency of occurrences. We will also be using a compare function, that would solve the problem of similar occurrences and would help with the arrangement of the queue in a prioritized manner where the lesser number will be given a priority when it shares the same frequency with another number. Finally, pop the element k times to get the k most frequent elements in the array.

Step 1: Create an unordered_map.

Step 2: Store in the unordered_map the elements and their frequency. For an array {3, 1, 4 , 4, 5, 2, 6, 1} and k = 2, the unordered map will look like this:

Number	Frequency
1	2
3	1
2	1
4	2
5	1
6	1

Step 3: Create a vector pair and assign the numbers to the first element of the pair and the frequency of the number to the next element. The Vector_pair out of the above-mentioned unordered_map will look like this:

```
(1, 2), (3, 1), (2, 1), (4, 2), (5, 1), (6, 1)
```

Step 4: Convert the vector pair into the priority queue with each index of the queue storing the number and their frequency of occurrence. Here the MAX_HEAP is sorted in a decreasing manner based on the frequency.

Step 5: A compare function will be used to prioritize two pairs when they have the same frequency and priority needs to be given to the lesser element as in Example 2. If the frequency does not match then the queue will be arranged according to the frequency.

Step 6: Run a loop k-times through the priority_queue to find the k most frequent numbers occurring in the array.

```cpp


// C++ program to find the k-most frequent number.
#include <bits/stdc++.h>
using namespace std;

// Compare method to arrange the priority_queue
// according to the frequency or lesser value
// in case the frequency of two pair matches.
struct compare {
    bool operator()(pair<int, int> p1,
                    pair<int, int> p2)
    {
        // if the frequency matches
        // return the lesser value
        if (p1.second == p2.second)
            return p1.first > p2.first;
        else
            return p1.second < p2.second;
    }
};

// Method to find the k-most frequent number
void k_most_frequent(int arr[], int n, int k)
{
    // Create an unordered_map
    unordered_map<int, int> mp;

    // Store the numbers
    // and frequency of occurrence
```

Run

Output:

```
1 4
```