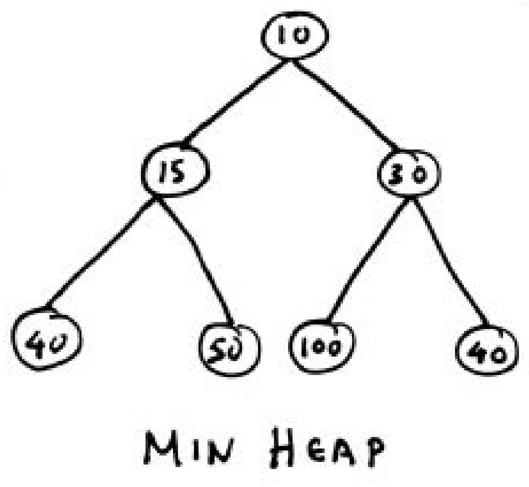
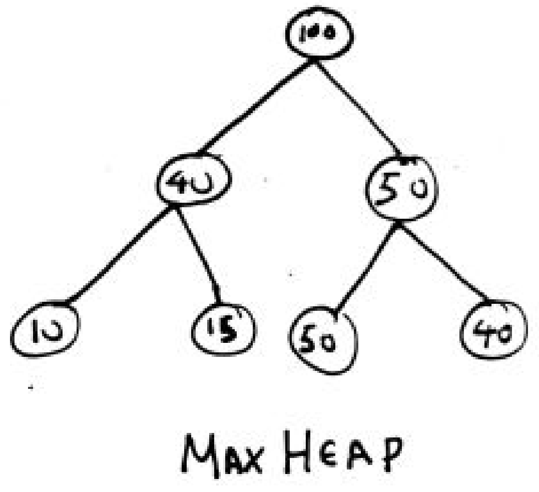
Heaps using STL

Heap: A heap is a specialized tree-based data structure which is essentially an almost complete tree that satisfies the heap property. In a max heap, for any given node C, if P is a parent node of C, then the key (the value) of P is less than or equal to the key of C. Like the tree, the heap has a single root - a node with no parent.

A common implementation of a heap is the binary heap. In that the tree is a binary tree. Generally, Heaps can be of two types:

- **Max-Heap:** in a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
- **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.





Common Operations of Heaps

The most common operations involving heaps are the following:

Basic:

- *find-max (find-min):* find a maximum item of a max-heap (or a minimum item of a min-heap) [aka, *peek*]
- *insert:* adding a new key to the heap [aka, *push*]
- *extract-max (extract-min):* returns the node of maximum value from a max-heap (or minimum value from a min-heap) after removing it from the heap [aka, *pop*]
- *delete-max (delete-min):* removing the root node of a max heap (or a min heap)

- *replace:* pop root and push a new key. More efficient than followed by push, since only need to baance once, not twice, and appropriate for fixed-size heaps.

Creation:

- *create-heap:* create an empty heap
- *heapify:* create a heap out of given array of elements
- *merge [aka, union]:* joining two heaps to form a valid new heap containing all the elements of both, preserving the original heaps.
- *meld:* joining two heaps to form a valid new heap containing all the elements of both, destroying the original heaps

Inspection:

- *size:* return the number of items in the heap
- *is-empty:* return true of the heap is empty, false otherwise

Internal:

- *increase-key (decrease-key):* updating a key within a max- (or min-)heap
- *delete:* delete an arbitrary node (followed by moving last node and sifting to maintain heap)
- *sift-up:* move a node up in the tree, as long as needed; used to restore heap condition after insertion. Called "sift" because node moves up the tree until it reaches the correct level.
- *sift-down:* move a node down in the tree, similar to sift-up; used to restore heap condition after deletion or replacement.

There are multiple ways of implementing the Heap ADT. The selection of the method has impact in the associated computational complexity. The table below summarizes the complexities within most important tasks.

Operation	**find-min**	**delete-min**	**insert**	**decrease-key**	**meld**
Binary	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$
Fibonacci	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Pairing	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
Brodal	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Strict Fibonacci	Θ(1)	O(logn)	Θ(1)	Θ(1)	Θ(1)
2-3 heap	O(logn)	O(logn)	O(logn)	Θ(1)	X

Example of using std::make_heap - *Make Heap from Range:* Rearranges the elements in the range [first,last) in such a way that they form a heap.

This will create a heap data structure and will thus allow a fast way to organize the elements based on their range and allow fast retrieval of the element with the highest value at any moment (with pop_heap), while allowing for fast insertion of new elements (with push_heap).

The element with the highest value is always pointed by first. The order of the other elements depends on the particular implementation, but it is consistent throughout all heap-related functions of this header.

The elements are compared using operator<(for the first version), or comp (for the second): The element with the highest value is an element for which this would return false when compared to every other element in the range.

```
// range heap example
#include <iostream>     // std::cout
#include <algorithm>          // std::make_heap, std::pop_heap,
std::push_heap, std::sort_heap
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,5,15};
  std::vector<int> v(myints,myints+5);

  std::make_heap (v.begin(),v.end());
  std::cout << "initial max heap   : " << v.front() << '\n';

  std::pop_heap (v.begin(),v.end()); v.pop_back();
  std::cout << "max heap after pop : " << v.front() << '\n';

  v.push_back(99); std::push_heap (v.begin(),v.end());
  std::cout << "max heap after push: " << v.front() << '\n';

  std::sort_heap (v.begin(),v.end());
  std::cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++)
    std::cout << ' ' << v[i];

  std::cout << '\n';

  return 0;
}
```

Example of using std::push_heap - *Push element into heap range:* Given a heap in the range [first,last-1), this function extends the range considered a heap to [first,last) by placing the value in (last-1) into its corresponding location within it.

```
// range heap example
#include <iostream>     // std::cout
#include  <algorithm>           //  std::make_heap,  std::pop_heap,
std::push_heap, std::sort_heap
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,5,15};
  std::vector<int> v(myints,myints+5);

  std::make_heap (v.begin(),v.end());
  std::cout << "initial max heap   : " << v.front() << '\n';

  std::pop_heap (v.begin(),v.end()); v.pop_back();
  std::cout << "max heap after pop : " << v.front() << '\n';

  v.push_back(99); std::push_heap (v.begin(),v.end());
  std::cout << "max heap after push: " << v.front() << '\n';

  std::sort_heap (v.begin(),v.end());

  std::cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++)
    std::cout << ' ' << v[i];

  std::cout << '\n';

  return 0;
}
```

Example of using std::pop_heap - *Pop element from heap range:* Rearranges the elements in the heap range [first,last) in such a way that the part considered a heap is shortened by one: The element with the highest value is moved to (last-1).

```
// range heap example
#include <iostream>     // std::cout
#include   <algorithm>           //   std::make_heap,   std::pop_heap,
std::push_heap, std::sort_heap
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,5,15};
  std::vector<int> v(myints,myints+5);

  std::make_heap (v.begin(),v.end());
  std::cout << "initial max heap   : " << v.front() << '\n';

  std::pop_heap (v.begin(),v.end()); v.pop_back();
  std::cout << "max heap after pop : " << v.front() << '\n';

  v.push_back(99); std::push_heap (v.begin(),v.end());
  std::cout << "max heap after push: " << v.front() << '\n';

  std::sort_heap (v.begin(),v.end());

  std::cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++)
    std::cout << ' ' << v[i];

  std::cout << '\n';

  return 0;
}
```

Example of using std::sort_heap - Sort elements of heap: Sorts the elements in the heap range [first,last) into ascending order.

```
// range heap example
#include <iostream>     // std::cout
```

```
#include   <algorithm>           //   std::make_heap,   std::pop_heap,
std::push_heap, std::sort_heap
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,5,15};
  std::vector<int> v(myints,myints+5);

  std::make_heap (v.begin(),v.end());
  std::cout << "initial max heap   : " << v.front() << '\n';

  std::pop_heap (v.begin(),v.end()); v.pop_back();
  std::cout << "max heap after pop : " << v.front() << '\n';

  v.push_back(99); std::push_heap (v.begin(),v.end());
  std::cout << "max heap after push: " << v.front() << '\n';

  std::sort_heap (v.begin(),v.end());

  std::cout << "final sorted range :";
  for (unsigned i=0; i<v.size(); i++)
    std::cout << ' ' << v[i];

  std::cout << '\n';

  return 0;
}
```

Example of using std::is_heap - *Test if range is heap:* Returns true if the range [first,last) forms a heap, as if constructed with make_heap.

```
// is_heap example
#include <iostream>     // std::cout
#include   <algorithm>            //   std::is_heap,   std::make_heap,
std::pop_heap
#include <vector>       // std::vector

int main () {
  std::vector<int> foo {9,5,2,6,4,1,3,8,7};

  if (!std::is_heap(foo.begin(),foo.end()))
    std::make_heap(foo.begin(),foo.end());
  std::cout << "Popping out elements:";
  while (!foo.empty()) {
    std::pop_heap(foo.begin(),foo.end());    // moves largest element
to back
    std::cout << ' ' << foo.back();          // prints back
    foo.pop_back();                               // pops element out of
container
  }
  std::cout << '\n';

  return 0;
}
```

Example of using std::is_heap_until - *Find first element not in heap order:* Returns an iterator to the first element in the range [first,last) which is not in a valid position if the range is considered a heap (as if constructed with make_heap).

```
// is_heap example
#include <iostream>     // std::cout
#include   <algorithm>           //   std::is_heap_until,   std::sort,
std::reverse
#include <vector>       // std::vector

int main () {
  std::vector<int> foo {2,6,9,3,8,4,5,1,7};

  std::sort(foo.begin(),foo.end());
  std::reverse(foo.begin(),foo.end());

  auto last = std::is_heap_until (foo.begin(),foo.end());

  std::cout << "The " << (last-foo.begin()) << " first elements are a
valid heap:";
  for (auto it=foo.begin(); it!=last; ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Example of using std::reverse - *Reverse range:* Reverses the order of the elements in the range [first,last).

```
// reverse algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::reverse
#include <vector>       // std::vector

int main () {
  std::vector<int> myvector;

  // set some values:
  for (int i=1; i<10; ++i) myvector.push_back(i);   // 1 2 3 4 5 6 7
8 9

  std::reverse(myvector.begin(),myvector.end());    // 9 8 7 6 5 4 3
2 1

  // print out content:
  std::cout << "myvector contains:";
        for     (std::vector<int>::iterator     it=myvector.begin();
it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Priority Queue using STL

Priority queues: an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a “priority” associated with it. In a priority queue, an element with high priority is served before an element with low priority. In some implementations, if two elements have the same priority, they are served according to the order in which they were enqueued, while in other implementations, ordering of elements with the same priority is undefined. While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A prirority queue is a concept like “a list” or “a map”; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods such as an unordered array.

Priority Queue in STL: Overall, priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.

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

Priority queues are implemented as container adaptors, which are classes that use an encapsulated object of a specific set of member functions to access its elements. Elements are popped from the “back” of the specific container, which is known as the top of the priority queue.

The standard container adaptor priority_queue is based on heaps. It calls make_heap, push_heap and pop_heap automatically to maintain heap properties for a container.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall be accessible through random access iterators and support the following operations:

- empty()
- size()
- front()
- push_back()
- pop_back()

The standard container classes vector and deque fulfill these requirements. By default, if no container class is specified for a particular priority_que class instantiation, the standard container vector is used.

Using a priority queue to sort: The semantics of priority queues naturally suggest a sorting method - insert all the elements to be sorted into a priority queue, and sequentially remove them; they will come out in sorted order. But this is the process also used by several sorting algorithms. This sorting process is concept-wise equivalent to the following sorting algorithms:

Name	**Priority Queue Implementation**	**Best**	**Average**	**Worst**
Heapsort	Heap	nlogn	nlogn	nlogn
Smoothsort	Leonardo Heap	n	nlogn	nlogn
Selection sort	Unordered Array	n^2	n^2	n^2
Insertion sort	Ordered Array	n	n^2	n^2
Tree sort	Self-balancing BST	nlogn	nlogn	nlogn

Example of using a priority queue through STL

```
// Note that by default C++ creates a max-heap
// for priority queue
#include <iostream>
#include <queue>
 using namespace std;
 void showpq(priority_queue <int> gq)
{
   priority_queue <int> g = gq;
   while (!g.empty())
   {
       cout << '\t' << g.top();
       g.pop();
   }
   cout << '\n';
}
 int main ()
{
   priority_queue <int> gquiz;
   gquiz.push(10);
   gquiz.push(30);
```

```
    gquiz.push(20);
    gquiz.push(5);
    gquiz.push(1);
     cout << "The priority queue gquiz is : ";
    showpq(gquiz);
     cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.top() : " << gquiz.top();

    cout << "\ngquiz.pop() : ";
    gquiz.pop();
    showpq(gquiz);
     return 0;
}
```

Priority Queue based on min-heap

```
// C++ program to demonstrate min heap
#include <iostream>
#include <queue>
using namespace std;

void showpq(priority_queue <int, vector<int>, greater<int>> gq)
{
    priority_queue <int, vector<int>, greater<int>> g = gq;
    while (!g.empty())
    {
        cout << '\t' << g.top();
        g.pop();
    }
    cout << '\n';
}

int main ()
{
    priority_queue <int, vector<int>, greater<int>> gquiz;
    gquiz.push(10);
    gquiz.push(30);
    gquiz.push(20);
    gquiz.push(5);
    gquiz.push(1);

    cout << "The priority queue gquiz is : ";
    showpq(gquiz);

    cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.top() : " << gquiz.top();
    cout << "\ngquiz.pop() : ";
    gquiz.pop();
    showpq(gquiz);
    return 0;
}
```

Task: Given a priority queue of integers, find the number of prime and non prime numbers.

Indicative Solution

```cpp
// CPP program to illustrate
// Application of top() function
#include <iostream>
#include <queue>
using namespace std;
 int main()
{
   int prime = 0, nonprime = 0, size;
   priority_queue<int> pqueue;
   pqueue.push(1);
   pqueue.push(8);
   pqueue.push(3);
   pqueue.push(6);
   pqueue.push(2);
   size = pqueue.size();
    // Priority queue becomes 1, 8, 3, 6, 2
    while (!pqueue.empty()) {
       for (int i = 2; i <= pqueue.top() / 2; ++i) {
           if (pqueue.top() % i == 0) {
               prime++;
               break;
           }
       }
       pqueue.pop();
   }
   cout << "Prime - " << prime << endl;
   cout << "Non Prime - " << size - prime;
   return 0;
}
```

References:

- http://www.cplusplus.com/
- https://www.geeksforgeeks.org
- https://en.wikipedia.org/wiki/Priority_queue
- https://en.wikipedia.org/wiki/Heap_(data_structure)