**Applications of Heaps:**

**1)** [Heap Sort](http://quiz.geeksforgeeks.org/heap-sort/): Heap Sort uses Binary Heap to sort an array in O(nLogn) time.

**2)** Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial 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](http://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/" \t "_blank)and[Prim’s Minimum Spanning Tree](http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/" \t "_blank).

**4)** Many problems can be efficiently solved using Heaps. See following for example.  
a) [K’th Largest Element in an array](http://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/" \t "_blank).  
b) [Sort an almost sorted array/](http://www.geeksforgeeks.org/nearly-sorted-algorithm/)  
c) [Merge K Sorted Arrays](http://www.geeksforgeeks.org/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 Min Heap. Time Complexity of this Operation is O(Logn) as this operation needs to maintain the heap property (by calling heapify()) after removing root.

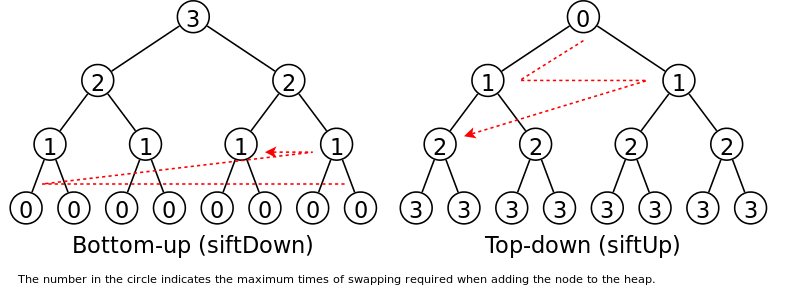
**3)** decreaseKey(): Decreases value of key. Time complexity of this operation is O(Logn). If the decreases key value of a node is greater than parent of the node, then we don’t need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

**4)**insert(): Inserting a new key takes O(Logn) 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(Logn) time. We replace the key to be deleted with minum infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove key.

BuildHeap takes O(n)

the difference lies in siftDown vs. siftUp that I think somewhat misses the point (or is just too brief to make it obvious). Indeed, you don't need to use siftUp at all when implementing a heap sort (although it is required for a priority queue, for example, to implement the insert operation).



The buildHeap function takes an array of unsorted items and moves them until it they all satisfy the heap property. There are two approaches one might take for buildHeap. One is to start at the top of the heap (the beginning of the array) and call siftUp on each item. At each step, the previously sifted items (the items before the current item in the array) form a valid heap, and sifting the next item up places it into a valid position in the heap. After sifting up each node, all items satisfy the heap property. The second approach goes in the opposite direction: start at the end of the array and move backwards towards the front. At each iteration, you sift an item down until it is in the correct location.

Both of these solutions will produce a valid heap. The question is: which implementation for buildHeap is more efficient? Unsurprisingly, it is the second operation that uses siftDown. If h = log n is the height, then the work required for the siftDown approach is given by the sum

(0 \* n/2) + (1 \* n/4) + (2 \* n/8) + ... + (h \* 1).

Each term in the sum has the maximum distance a node at the given height will have to move (zero for the bottom layer, h for the root) multiplied by the number of nodes at that height. In contrast, the sum for calling siftUp on each node is

(h \* n/2) + ((h-1) \* n/4) + ((h-2)\*n/8) + ... + (0 \* 1).

It should be clear that the second sum is larger. The first term alone is hn/2 = 1/2 n log n, so this approach has complexity at best O(n log n). However, the sum for the siftDown approach can be bounded by extending it to a Taylor series to show that it is indeed O(n). If there is interest, I can edit my answer to include the details. Obviously, O(n) is the best you could hope for.

----

for (int i = N/2; i >= 0; --i)

push\_heap(heap + i, N - i);

(push\_heap is a function that accepts a pointer to a heap and the heap size and pushes the top of the heap until the heap conditions are respected or the node reaches the bottom of the heap).

To get why this is O(N) look at the complete binary tree:

* 1/2 elements (last level, i > N/2) are pushed down at most 0 steps -> N/2 \* 0 operations
* 1/4 elements (last-1 level, i > N/4) are pushed down at most 1 step -> N/4 \* 1 operations
* 1/8 elements (last-2 level, i > N/8) are pushed down at most 2 steps -> N/8 \* 2 operations ...
* N/4 \* 1 + N/8 \* 2 + N/16 \* 3 + ... =
* N/4 \* 1 + N/8 \* 1 + N/16 \* 1 + ... +
* N/8 \* 1 + N/16 \* 2 + ... =
* N/4 \* 1 + N/8 \* 1 + N/16 \* 1 + ... + // < N/2
* N/8 \* 1 + N/16 \* 1 + ... + // < N/4
* N/16 \* 1 + ... + // < N/8
* ... = // N/2 + N/4 + N/8 + ... < N

Hope that math is not really too complicated. If you look on the tree and add how much each node can be pushed down you'll see the upper bound O(N).

Why is Binary Heap Preferred over BST for Priority Queue?

A typical [Priority Queue](http://geeksquiz.com/priority-queue-set-1-introduction/) 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](http://geeksquiz.com/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](http://www.geeksforgeeks.org/avl-tree-set-1-insertion/), [Red-Black Tree,](http://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/) 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. See [this](http://geeksquiz.com/how-to-implement-decrease-key-or-change-key-in-binary-search-tree/) for details.

**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](http://www.geeksforgeeks.org/floor-and-ceil-from-a-bst/) can be found in O(Logn) time.
* [K’th largest/smallest element](http://www.geeksforgeeks.org/find-k-th-smallest-element-in-bst-order-statistics-in-bst/)be found in O(Logn) time by augmenting tree with an additional field.

**std::priority\_queue**

If you are using std::priority\_queue as your priority queue class, the standard container class std::vector is used for its underlying container class, by default.

Generally, it is less efficient to push first than pop first.

**>** Pushing an element in priority\_queue will envoke vector::push\_back which can potentially reallocate the underlying buffer if it exceeds it current capacity.

**>**

priority\_queue::pop

When you pop an element from priority\_queue, it calls the pop\_heap algorithm to keep the heap property of priority\_queues, and then calls the member function pop\_back of the underlying container object to remove the element.

priority\_queue::push

When you push an element to priority\_queue, it calls the member function push\_back of the underlying container object, and then calls the push\_heap algorithm to keep the heap property of priority\_queues.

Assume there are now **N** elements in priority queue.

If you push first, the algorithm push\_heap is called two times, to adjust **N+1** and **N+1** elements, respectively.

If you pop first, the algorithm push\_heap is called two times, to adjust **N** and **N** elements, respectively.