There are many situations, both in real life and in computing applications, where we wish to choose the next "most important" from a collection of people, tasks, or objects. For example, doctors in a hospital emergency room often choose to see next the "most critical" patient rather than the one who arrived first. When scheduling programs for execution in a multitasking operating system, at any given moment there might be several programs (usually called [**jobs**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-job)) ready to run. The next job selected is the one with the highest [**priority**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-priority). Priority is indicated by a particular value associated with the job (and might change while the job remains in the wait list). When a collection of objects is organized by importance or priority, we call this a [**priority queue**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-priority-queue).

 A normal queue data structure will not implement a priority queue efficiently because search for the element with highest priority will take Θ(n)Θ(n) time. A list, whether sorted or not, will also require Θ(n)Θ(n) time for either insertion or removal. A BST that organizes records by priority could be used, with the total of nn inserts and nn remove operations requiring Θ(nlogn)Θ(nlog⁡n) time in the average case. However, there is always the possibility that the BST will become unbalanced, leading to bad performance.

This section presents the [**heap**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-heap) data structure. [**[1]**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Heaps.html#id2) A heap is defined by two properties. First, it is a complete binary tree, so heaps are nearly always implemented using the [**array representation for complete binary trees**](https://traky.cs.hut.fi/Books/CSE-A1141/html/CompleteTree.html#completetree). Second, the values stored in a heap are [**partially ordered**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-partial-order). This means that there is a relationship between the value stored at any node and the values of its children. There are two variants of the heap, depending on the definition of this relationship.

A [**max heap**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-max-heap) has the property that every node stores a value that is greater than or equal to the value of either of its children. Because the root has a value greater than or equal to its children, which in turn have values greater than or equal to their children, the root stores the maximum of all values in the tree.

A [**min heap**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-min-heap) has the property that every node stores a value that is less than or equal to that of its children. Because the root has a value less than or equal to its children, which in turn have values less than or equal to their children, the root stores the minimum of all values in the tree.

Note that there is no necessary relationship between the value of a node and that of its sibling in either the min heap or the max heap. For example, it is possible that the values for all nodes in the left subtree of the root are greater than the values for every node of the right subtree.

A BST defines a [**total order**](https://traky.cs.hut.fi/Books/CSE-A1141/html/Glossary.html#term-total-order) on its nodes in that, given the positions for any two nodes in the tree, the one to the "left" (equivalently, the one appearing earlier in an inorder traversal) has a smaller key value than the one to the "right". In contrast, a heap implements a partial order. Given their positions, we can determine the relative order for the key values of two nodes in the heap only if one is a descendant of the other.

Min heaps and max heaps both have their uses. For example, the Heapsort uses the max heap, while the Replacement Selection algorithm used for external sorting uses a min heap.

Be sure not to confuse the logical representation of a heap with its physical implementation by means of the array-based complete binary tree. The two are not synonymous because the logical view of the heap is actually a tree structure, while the typical physical implementation uses an array.

PARENT (i) = i / 2 Returns the index of the node's father (splitting is off)

LEFT (i) = 2i Returns the node's left child index

RIGHT (i) = 2i + 1 Returns the index of the right child of the node

Adding an Item

Adding a new item to your kekoon:

Put the new embryo into the last embryo of the bow (the levels are filled from left to right)

Raise the embedded item to the correct position by replacing the nose with the knife until the knockout is resumed

With the addition operations, the construction of the keel is quite straightforward. It only adds each item individually to the last embryo of the bow and then raises the added element into the right place.

Algorithm 2 Max-Heapify(A, i)

l ← Left-child-index(i)

r ← Right-child-index(i)

if l < heap-size[A] and A[l] > A[i] then

greatest ← l

else

greatest ← i

end if

if r < heap-size[A] and A[r] > A[greatest] then

greatest ← r

end if

if greatest ≠ i then

Swap(A[i],A[greatest])

Max-Heapify(A, greatest)

end if

Since the heap is a complete binary tree, its height is guaranteed to be the minimum possible. In particular, a heap containing nn nodes will have a height of Θ(logn)Θ(log⁡n). Intuitively, we can see that this must be true because each level that we add will slightly more than double the number of nodes in the tree (the ii th level has 2i2i nodes, and the sum of the first ii levels is 2i+1−12i+1−1). Starting at 1, we can double only lognlog⁡n times to reach a value of nn. To be precise, the height of a heap with nn nodes is ⌈logn+1⌉⌈log⁡n+1⌉.

Each call to insert takes Θ(logn)Θ(log⁡n) time in the worst case, because the value being inserted can move at most the distance from the bottom of the tree to the top of the tree. Thus, to insert nn values into the heap, if we insert them one at a time, will take Θ(nlogn)Θ(nlog⁡n) time in the worst case.

If all nn values are available at the beginning of the building process, we can build the heap faster than just inserting the values into the heap one by one. Consider this example, with two possible ways to heapify an initial set of values in an array.

 it is clear that the heap for any given set of numbers is not unique, and we see that some rearrangements of the input values require fewer exchanges than others to build the heap.