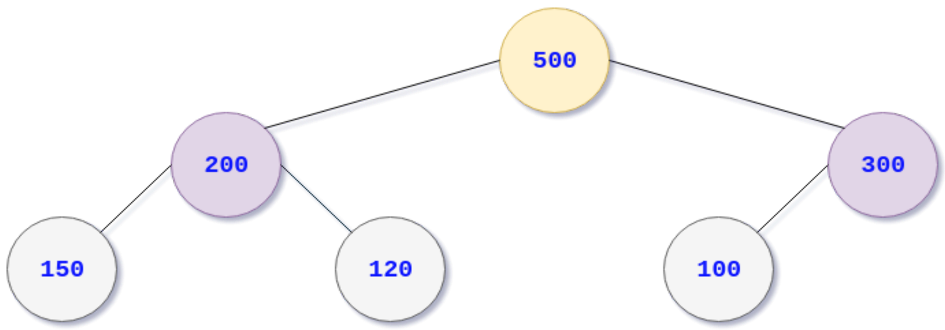
Maximum Heap

[0]	[1]	[2]	[3]	[4]	[5]	[6]
Number Of Nodes	Root Node	Left Child of Root	Right Child of Root	Left Child of Node 2	Right Child of Node 2	Left Child of Node 3

A maximum heap (a minimum heap is the opposite of a maximum heap) is similar to a Binary Search Tree, but the arrangement of the nodes follow a different set of rules. The data in a node is never less than the data of its children and the structure of a heap must be a complete binary tree. This means that a parent node could have a left child with a value greater than its sibling or vise versa. Furthermore, a heap is usually implemented as an array.

One use for a maximum heap is the implementation of a priority queue (see the **Queue** pages for more information). See **MaximumHeap.cpp** for an example program and further explanation via comments.

Here is an example of a maximum heap represented as a Binary Tree.



Here is the above maximum heap represented as an array:

[0]	[1]	[2]	[3]	[4]	[5]	[6]
6	500	200	300	150	120	100

This heap will be implemented such that the first element (**index** 0) contains the total number of used nodes (array elements) that are currently in the heap and such that the root node starts at the second element (**index** 1). Keep in mind that the number at **index** 0 is **not** the total size of the array.

Using an array in this manner makes it easy to calculate a node's children as well as to calculate a node's parent. The way this is done is via integer division (where anything on the right of the decimal is truncated). On the following page, let **P** denote **Parent**, **LC** denote **Left Child**, and **RC** denote **Right Child**. Basic algebra will be used in the calculation.

Starting at some parent node **P**, **LC** = **2P** and **RC** = **2P + 1**. For example, suppose the parent node is at index 2. Therefore its **LC** = **2*2** = **4** and its **RC** = **2*2 + 1** = **5**.

If you start at some left child node **LC**, you are able to manipulate the equation **LC** = **2P** to find its parent. Divide both sides by 2 and you have **P** = **LC/2**. For example, suppose the left child node is at index 6 and want to find its parent node. Therefore, **P** = **6/2** = **3**. Remember left children will always be at an even index.

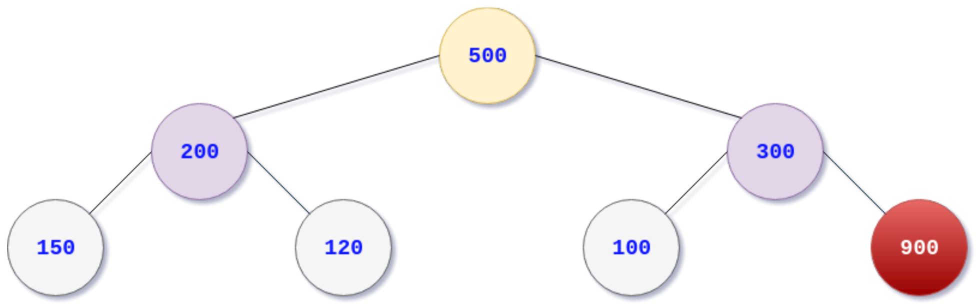
If you start at some right child node **RC**, you are able to manipulate the equation **RC** = **2P + 1** to find its parent. Subtract 1 from both sides, divide both sides by 2, and then you have **P** = **(RC - 1)/2**. For example, suppose the right child node is 7 and want to find its parent node. Therefore, **P** = **(7-1)/2** = **3**. Remember right children will always be at an odd index.

Insertion

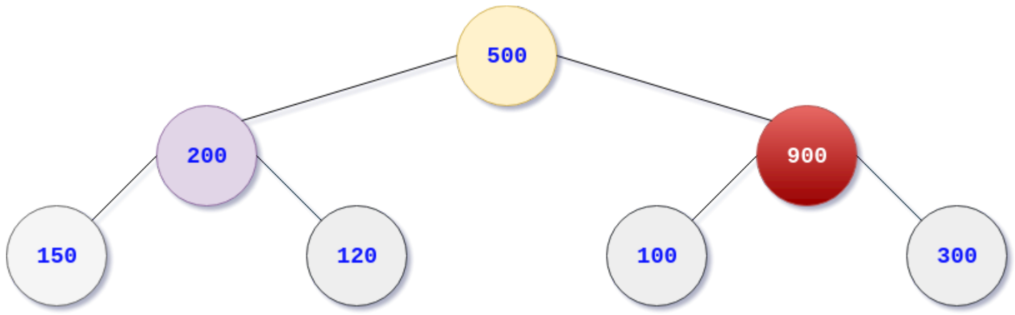
Your array will have some beginning size, and the maximum number of nodes is **size - 1** (size may need to change during program). The first element of the array will be 0 as the heap is empty. Inserting a value into the heap the first time through will increment the first element by one and place the value into the second element (**index 1**).

From there, each insertion will be placed into the first available location (while incrementing the value at **index 0**). That is, the next available location after the first entry is at **index 2**, then **index 3**, and so forth. This keeps the structure a complete binary tree. However, it may not meet the requirements to be called a heap. Remember, a parent node must always be greater than its children.

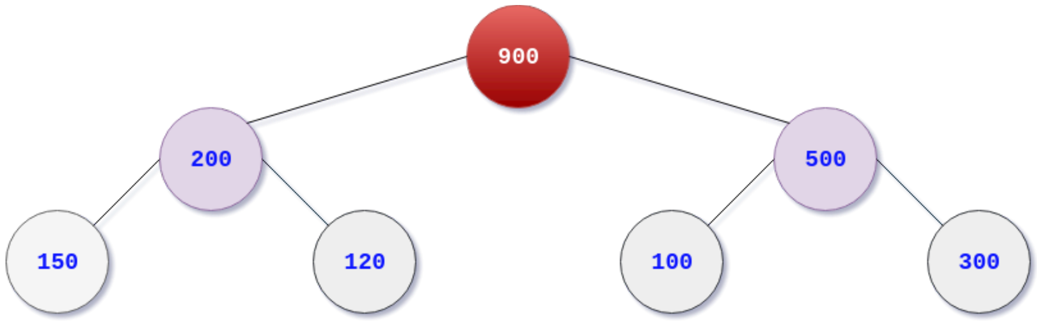
Using the previous page's Binary Tree representation of a maximum heap, the next insertion will be the leftmost available node in the deepest level of the Binary Tree, which is the right child of the node containing 300. This is at **index 7** in the array and is the node with the **red** background containing 900:



Looking at the heap, you should notice 300 is less than 900. A process called **reheapification upward**, or **sift-up** needs to take place. This process means to swap the newly inserted value with its parent's value so long as the newly inserted value is greater than its parent's value. This process continues up until the root node if necessary. Continuing our example, **reheapification upward** takes place and 900 swaps with its parent's value 300.



Another check is made to see if the new parent of 900 is lesser in value. 500 is indeed less than 900, so **reheapification upward** takes place again swapping 900 and 500.



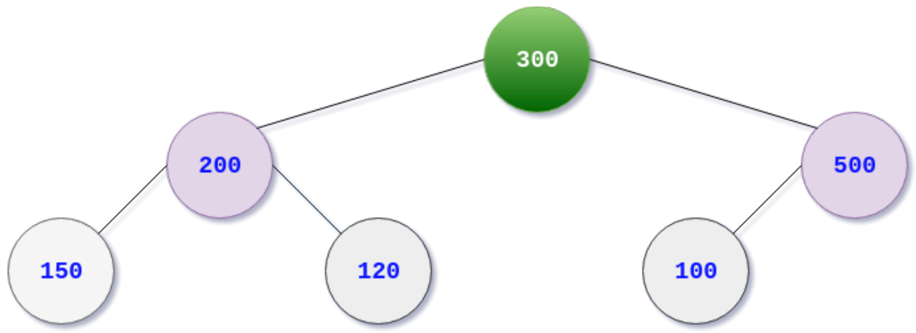
Since the root node has no parent to compare, **reheapification upward** terminates. The current Binary Tree is now a true maximum heap. Every parent is greater than its children again. Keep in mind that **reheapification upward** stops either when the newly inserted value reaches the root node or when the parent node of the newly inserted value is greater than the newly inserted value.

Deletion

In a maximum heap (as well as a priority queue), the root node (value at **index** 1) is the highest priority and will always be removed first. You cannot specify a random value to be deleted. If you want to delete a value and only the root node exists, decrement the value at **index** 0 and no further action is required.

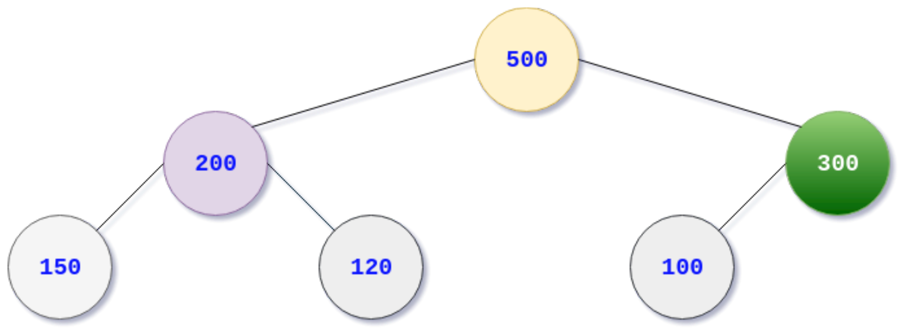
To continue the example from above, if the delete function were invoked, 900 would be deleted. However, if you delete the root node (value at **index** 1) and did no further action, you broke the heap. What needs to be done are the following actions:

1) Take the value in the last node (appropriate index is the value of **index** 0) of the deepest level and replace the root node with said value. Then, decrement the value at **index** 0 by one (this deletes the last node of the heap). Here is what the heap would look like after this step:



2) Now, you have to make sure that the root node is greater than both of its children as the highest number always goes in the root node. If any swapping needs to take place, this is called **reheapification downward**, or **sift-down**. It is important to check both children, because if both children are greater than its parent, its parent's value gets swapped with the highest valued child. This is to ensure the property of the maximum heap is met. If the lesser of the two children were swapped with its parent, then the new parent's value will still be smaller than the other non-swapped child breaking the maximum heap property.

In this example, 300 is greater than 200 so no need to potentially swap with the left child. However, 300 is less than 500 and we need to swap those two values. Our new maximum heap looks as such:



3) **Reheapification downward** continues down the maximum heap until the parent's value is greater than both children starting from the root node in step 1 or when the value that was swapped in step 1 reaches a leaf node. In this example, 300 is greater than 100, which is its only child. Therefore, **reheapification downward** is complete and the maximum heap has met all of its properties again.