# Array Representation of a Binary Heap

* For all out operations, our overall strategy is:
  + Preserve **complete tree structure** property
  + This may break **heap order property**
  + Percolate to restore **heap order property**
* An important observation is that because a complete binary tree contains no gaps (other than the right side of the array), it can be represented in an array and no links are necessary.
* The array in Figure 6.3 corresponds to the heap in Figure 6.2.

Shape

Description automatically generated

* For any element in array position *i*:
  + the left child is in position 2*i*,
  + the right child is in the cell after the left child (2*i* + 1),

and

* + the parent is in position [*i*/2].
* Thus, not only are links not required, but the operations required to traverse the tree are extremely simple.
* Throughout this chapter, we shall draw the heaps as trees, with the implication that an actual implementation will use simple arrays.
* **The only problem with this implementation is that an estimate of the maximum heap size is required in advance.**
* However, this is typically not a problem because we can always resize the array if necessary.

# **Heap Operations**

* A heap data structure will consist of the following private data members:
  + An array (of Comparable objects) called **items**.
  + An integer representing the current heap **size**.
  + A constant variable representing the default capacity for the array **DEFAULT\_CAPACITY**.

## **Heap Insertion**

To insert in a heap:

1. Insert element *X* into the heap at the last leaf position to preserve completeness.
2. While heap order is violated, percolate *X* up by repeatedly exchanging key values with its parent until the heap order is restored.

* To insert an element *X* into the heap, we create a ‘hole’ in the next available location. This location could be called the next available leaf position, or size + 1 in the items array. We need to do this to maintain a complete tree.
* If X can be placed in the hole without violating heap order (is X’s value smaller than its parents?), then we do so and are done.
* Otherwise, we slide (swap) the element that is in the hole’s parent node into the hole, thus bubbling the hole up toward the root.
* We continue this process until X can be placed (assigned) in the hole.

**Example 1: Insert 14**

* Figure 6.6 shows that to insert 14, we create a hole in the next available heap location.
* Inserting 14 in the hole would violate the heap-order property, so 31 is slid down into the hole.
* This strategy is continued in Figure 6.7 until the correct location for 14 is found.

A picture containing text, watch

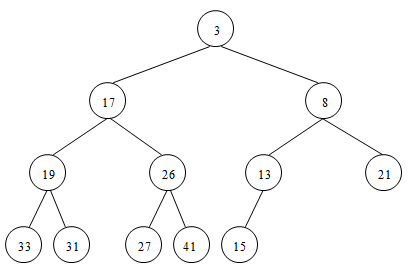
Description automatically generated

A picture containing text, watch

Description automatically generated

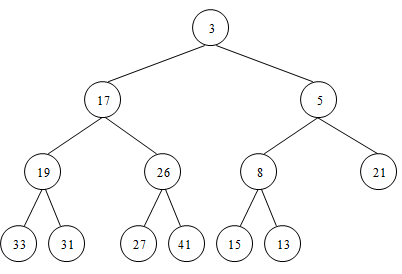
**Example 2: Insert c (say priority 5)**

1. Put 5 at the last leaf position.
2. This isn’t a heap, since the 5 is smaller than its parent.
3. We swap it with the 13 to fix this.
4. Since it still isn’t smaller than its parent is, we then swap it with the 8.
5. Heap-order is now restored.





We now get the following figure:



**Percolate Up**

* This general strategy of repeatedly exchanging key values with a parent node until the heap order is restored is known as **percolating up**.
* The new element is percolated up the heap until the correct location is found.
* Insertion is easily implemented with the code shown in Figure 6.8.

Text

Description automatically generated

**Optimizations**

1. **Do not insert the new item until the end.**

* We could have implemented the percolation in the insert routine by performing repeated swaps until the correct order was established, but a swap requires three assignment statements.
* If an element is percolated up *d* levels, the number of assignments performed by the swaps would be 3*d*. Our method uses *d* + 1 assignments.
* If the element to be inserted is the new minimum, it will be pushed all the way to the top.
* In is the case, then at some point, *hole* will be 1 and we will want to break out of the loop.

1. **Initialize the value at position zero in the heap to the new item.**

* We could create an explicit test to test if we have reached the top of the array, or we can put a reference to the inserted item in position 0 in order to make the loop terminate.
* The loop will terminate when the item in position 0 is compared against itself, resulting in a loop terminating condition.
* We elect to place *x* into position 0 in our implementation.

**Analysis**

* The time to do the insertion could be as much as *O*(log *N*), if the element to be inserted is the new minimum and is percolated all the way to the root.
* On average, the percolation terminates early.
* It has been shown that 2.607 comparisons are required on average to perform an insert, so the average insert moves an element up 1.607 levels.

## **Heap Removal**

* A heap removal operation **removes and returns the minimum element in the heap**.

To remove the top element in a heap:

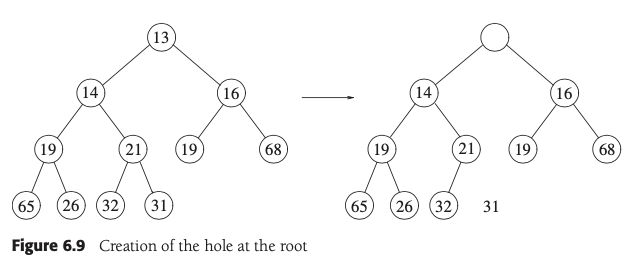
1. Save then delete the minimum (root) element from the heap.
2. Replace the now deleted root item with the last item *X* (rightmost leaf).
3. If *X* can be placed in the hole while maintain the tree’s heap-order, then we are done.
4. Else, percolate root element *X* down by swapping with the smaller of its two children repeatedly until the heap order is restored.

* When the minimum is removed, a hole is created at the root.
* In order to fill the hole, and to maintain completeness of the tree, we remove the last leaf in the tree. Since the heap now becomes one smaller, it follows that the last element *X* in the heap must move somewhere in the heap.
* If *X* can be placed in the hole and not violate minheap-order, then we are done.
* If it does, then we slide the smaller of the hole’s children into the hole, thus pushing the hole down one level. We repeat this step until *X* can be placed in the hole.
* Thus, our action is to place *X* in its correct spot along a path from the root containing *minimum* children.

**Example: A call to removeMin()**

1. Save and remove the minimum value 13 from the tree.

After removing the minimum value, a hole is created at the root.



1. Since the heap now becomes one smaller, it follows that the last element *31* in the heap must move to the hole.
2. The heap-order property is not maintained, since 31 is greater than both of its children.
3. Percolate the smaller of *X*’s children into the hole. Push the hole down one level.

A picture containing text, watch, clock

Description automatically generated

1. We repeat this step until *X* can be placed in the hole.

A picture containing text, watch, map

Description automatically generated

1. Return 13.

**Common Pitfall**

* A frequent implementation error in heaps occurs when there are an even number of elements in the heap, and the one node that has only one child is encountered.
* You must make sure **not to assume that there are always two children**, so this usually involves an extra test.
* In the code depicted in Figure 6.12, we’ve done this test at line 29.
* The worst-case running time for this operation is *O*(log *N*).
* On average, the element that is placed at the root is percolated almost to the bottom of the heap (which is the level it came from), so the average running time is *O*(log *N*).

A picture containing text

Description automatically generated