Heap Data Structure

A *heap* is a binary tree in which each node's value is less than or equal to the values of nodes in its left and right subtrees.

A *dense* (*complete*) binary tree of height m is a full binary tree satisfying these conditions:

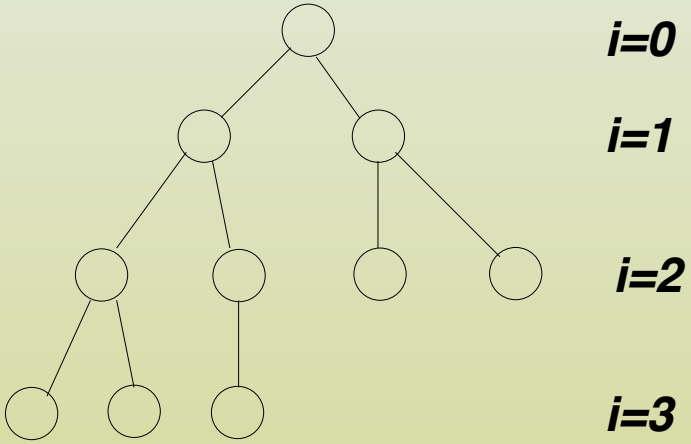
1. There are exactly 2^i nodes at depth $i = 0, 1, \ldots, m-2$ and there are at most 2^{m-1} nodes at depth $m-1$.

2. All nodes with two children at depth $m-1$ appear to the left of any node with only one child at depth $m-1$.

   Any node with only one child at depth $m-1$ appears to the left of all nodes with no children at depth $m-1$.

   There is at most one node with only one child at depth $m-1$.

Dense binary tree (height m=4)



A dense binary tree is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right—there are no missing nodes in the tree. The height of a dense tree of N nodes is at most $\lfloor \log_2 N \rfloor$.

Heap Property

In a complete tree the `left` and `right` pointers are not needed, so we can use an array to implement a heap.

- We place the root in position 0. We also need to maintain an integer that tells us how many nodes are currently in the tree.
- The left child of a node in array position k is found in position $2*k+1$, and the right child is found in position $2*k+2$. We test against the actual tree size to make sure whether the left or right child exists.
- The parent of a child in position k is in position $\lfloor (k-1)/2 \rfloor$. Every node (except the root) has a parent.

Using an array to store a tree is called *implicit representation*. Child references and the operations required to traverse are extremely simple and very fast for this representation. The heap class will consist of an array of objects and an integer representing the current heap size.

Heaps are drawn as trees so as to make the algorithms easier to visualize. The implementation of these trees always uses an array. From the property below we see that the minimum element can always be found at the root.

> *In a heap, for every node X with parent P, the key in P is smaller than or equal to the key in X.*

The Class BinaryHeap

```
class BinaryHeap {
private: int curSize; //number of elements in heap
   bool orderOK; //true if array has heap-order property
   int *array; //(dynamic) heap array
   int length; //the length of the array
   static const int DEF_SIZE = 8;
   void getNewArray(int newSize) {
      array = new int[newSize];
      length = newSize;
   }
   // other private functions
```

```
public: //class BinaryHeap (cont)
   BinaryHeap(int size = DEF_SIZE) { //constructor
      curSize = 0;
      orderOK = true;
      getNewArray(size);
   }
   int findMin() throw(EmptyHeap) {
      if ( isEmpty() ) throw EmptyHeap();
      if (! orderOK) buildHeap();
      return array[0];
   }
   bool isEmpty( ) const { return curSize == 0; }
   void buildHeap();
   void insert(int x);
   void deleteMin() throw(EmptyHeap);
};
```

Insert Operation

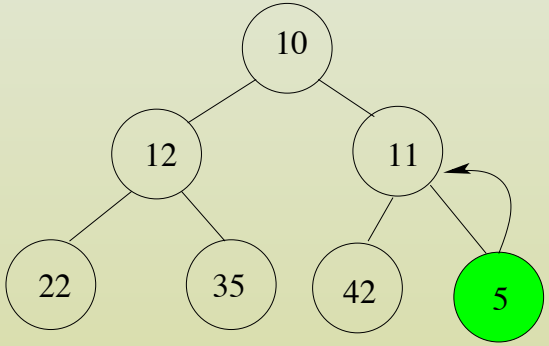
To insert an element X into a heap, we must add a node to the tree. The only option is to create a new position (hole) in the next available locations (after the last element of the array). If X is placed and heap order is not violated, we are done. Otherwise, we move down the element in the parent node in the hole, and the hole is moved in the parent's node. We continue the process until X is placed in the proper node and the tree has the heap-order property. This general strategy is called a *walk-up* or *upheap*.

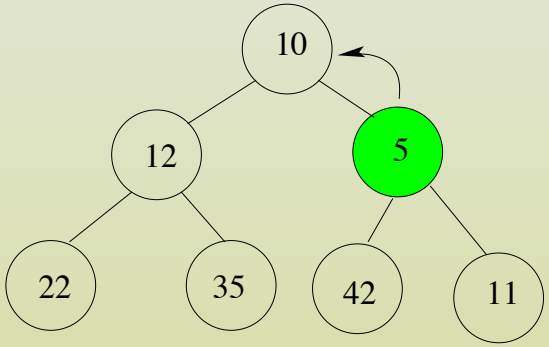
```
void BinaryHeap::insert(int x) {
   if (!orderOK) { //establish heap order later
      toss(x);
      return;
   }
   checkSize( );
   //walk up (establish heap order now)
   int hole = curSize++;
   for ( ; hole > 0 && array[(hole-1)/2] > x; hole = (hole-1)/2)
      array[hole] = array[(hole-1)/2];
   array[hole] = x;
}
```

Insert Operation (cont.)

```
// double the heap array if it is full
void BinaryHeap::checkSize( ) {
   if (curSize == length) {
      int *oldArray = array;
      getNewArray(2*curSize);
      for (int i = 0; i < curSize; i++)
         array[i] = oldArray[i];
      delete [] oldArray;
   }
}
// insert into array without maintaining heap order
void BinaryHeap::toss(int x) {
   checkSize( );
   array[ curSize++ ] = x;
   if (array[(curSize-1)/2] > x) orderOK = false;
}
```

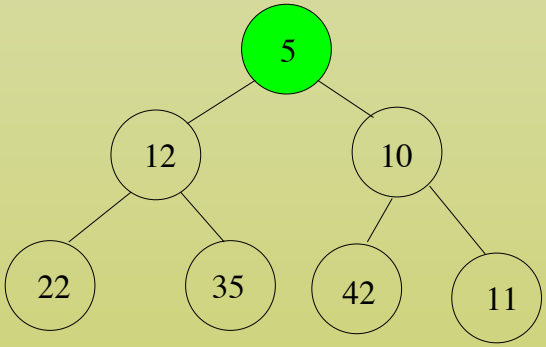
Insert Operation (cont.)





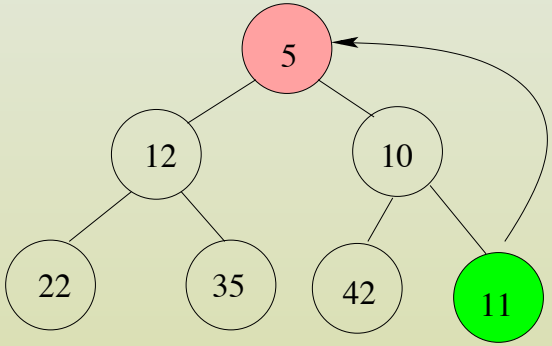
Insert 5 in the minimum heap

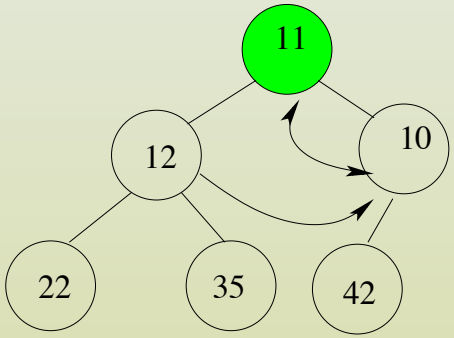
Walk up with 5



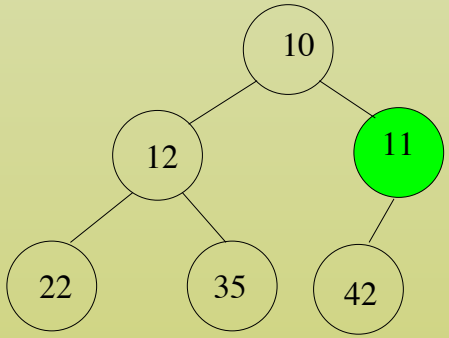
Delete Operation

When the minimal element is removed from the heap (from the root = `array[0]`), a hole is created at the root. The heap now becomes smaller, so the last node must be eliminated. The last element is placed in the root, and the size of the array is decreased. Unfortunately, in most cases, the array now violates the heap-order property. To establish it we need to apply the process called a *walk-down* or *downheap*. It is similar to the walk-up process, but this time the hole moves down instead of moving up. Notice that we cannot assume that there are two children; the last node can have only one child.





Remove 5 from the minimum heap

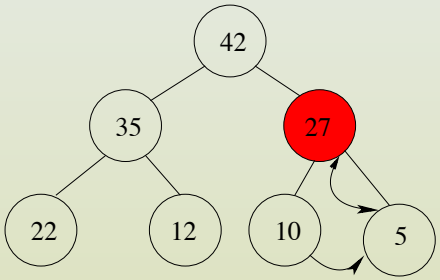


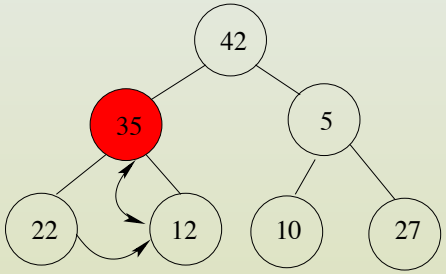
```
void BinaryHeap::deleteMin( ) throw(EmptyHeap) {
   array[0] = array[ --curSize ]; //decrease size
   walkDown(0);
}
void BinaryHeap::walkDown(int hole) {
   int child, key = array[hole];
   for ( ; 2*hole+1 < curSize; hole = child) {
      child = 2*hole+1;
      if (child != curSize-1 && array[child] > array[child+1])
         child++; // right child = 2*hole+2
      if (key > array[child]) array[hole] = array[child];
      else break;
   }
   array[hole] = key;
}
```

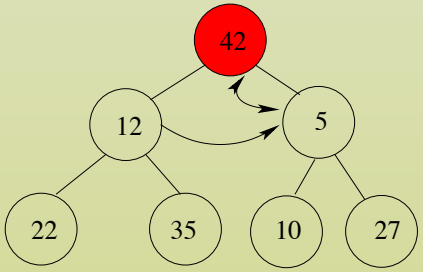
Building a Heap

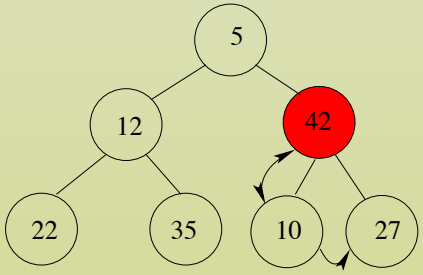
The `buildHeap` method takes an array that does not have heap order and converts it into a heap (with heap order). We want it to be a linear time operation (that is, $O(N)$ for N elements). Note that N insertions could be done in $O(N \log_2 N)$ time, so `insert()` cannot be used. We can establish heap order by recursively calling `buildHeap()` on the left and right subheaps and calling `walkDown()` for the root. But the recursion is not necessary, since if we call `walkDown()` on nodes in reverse level order, then at the point when `walkDown(i)` is processed, all descendants of node `i` will have been processed by a prior call to `walkDown()`. Notice that `walkDown()` on a leaf need not be performed.

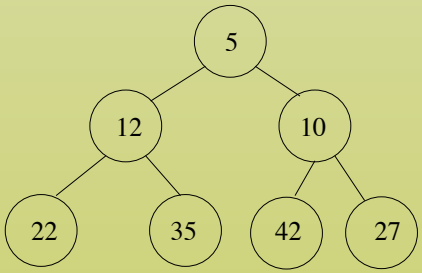
```
void BinaryHeap::buildHeap( ) {
   for (int i = (curSize-2)/2; i >= 0; i--) walkDown(i);
   orderOK = true;
}
```











Build minimum heap

Efficiency of Insert and Delete Operations

The number of comparisons required to add or delete a node in this implementation of a heap is proportional to the length of the path the item travels as it is "walked up" or "walked down". Since the binary tree is dense (complete), it implies that this path length is $O(\log_2 N)$ for a tree with N nodes. Therefore, the runtime efficiency of the operations insert and delete is $O(\log_2 N)$.

Sorting: Heapsort

The priority queue can be used to sort N items as follows:

1. Insert every item into a binary tree
2. Extract every item by calling `deleteMin()` N times. The resulting array is sorted.

Therefore we can use the `BinaryHeap` functions as follows:

1. `toss` each item into a binary heap
2. Apply `buildHeap`
3. Call `deleteMin()` N times: the items are sorted

Steps 1 and 2 take $O(N)$ time, and step 3 takes $N \log_2 N$ time. Consequently, we have $O(N \log_2 N)$ worst-case sorting algorithm.

In order to get a stand-alone sorting algorithm, we need to rewrite some parts of the above methods. First of all, we should not use the `BinaryHeap` class directly; it carries unnecessary overhead. Moreover, using the above approach, it seems that we need an additional array where we can store deleted elements. But we can notice that after deleting an item we shrink the array (by removing the last node), so we can reuse the last node and place the removed item there.

The Heapsort Algorithm

The heapsort algorithm has many advantages:

1. Guarantees $O(N \log_2 N)$ efficiency.
2. Avoids the overhead associated with recursion.
3. Sorts the array in place, not requiring an extra copy of the values.

The algorithm has 2 phases:

1. In the first phase, the array containing N data items is transformed into a heap.
2. Phase 2 sorts the elements by removing roots from the heap.

To transform the tree structure into a heap (**phase 1**):

1. Process the node that is the parent of the rightmost node on the lowest level. Call `walkDown()` with this node.
2. Move left on the same level. Again `walkDown()` is called for this new node.
3. When the left end of this level is reached, move up a level, and, beginning with the rightmost parent node, repeat step 2.

Steps for the **phase 2**:

1. Swap the root node (containing the largest element in the heap) with the bottom rightmost child. Decrease the length of the array.
2. Call `walkDown()` on the new root value to restore a heap.
3. Repeat steps 1 and 2 until only one element is left.

It creates a sorted array in increasing order.

Heapsort Implementation

```
// length = the number of elements in A
void heapsort(int A[], int length) {
   for (int i = (length-2)/2; i >= 0; i--) //build heap
      walkDown(A, i, length);
   for (int i = length-1; i > 0; i--) {
      swapElems(A, 0, i); // swap A[0] and A[i]
      walkDown(A, 0, i);
   }
}
void swapElems(int A[], int i1, int i2) {
   int tmp = A[i1];
   A[i1] = A[i2];
   A[i2] = tmp;
}
```

```
void walkDown(int A[], int index, int size) {
   int k, child;
   int key = A[index];
   for (k = index ; 2*k+1 < size; k = child) {
      child = 2*k+1; //left child
      if (child != size-1 && A[child] < A[child+1])
         child++;  // right child = 2*k+2
      if (key < A[child]) A[k] = A[child];
      else break;
   }
   A[k] = key;
}
```