Heaps
Chapter 21

CS 3358
Summer I 2012

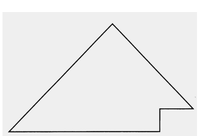
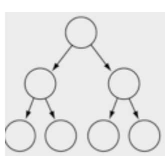
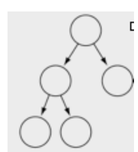
Jill Seaman

Binary heap data structure

- A binary heap is a special kind of binary tree
  - has a restricted structure (must be complete)
  - has an ordering property (parent value is smaller than child values)
- Used in the following applications
  - Priority queue implementation: supports enqueue and deleteMin operations in O(log N)
  - Heap sort: another O(N log N) sorting algorithm.

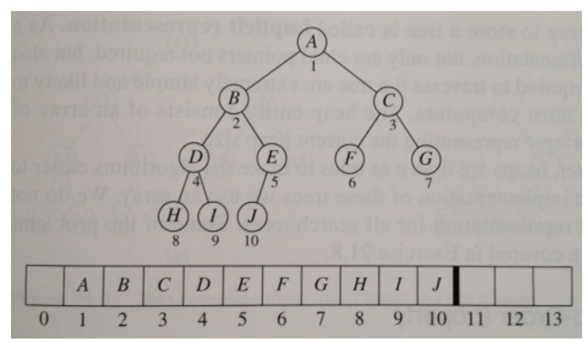
Binary Heap:
structure property

- **Complete binary tree**: a tree that is completely filled
  - every level except the last is completely filled.
  - the bottom level is filled left to right (the leaves are as far left as possible).

Complete Binary Trees

- A complete binary tree can be easily stored in an array
  - place the root in position 1 (for convenience)



Complete Binary Trees
Properties

- The height of a complete binary tree is floor($\log_2$ N)
  (floor = biggest int less than)
- In the array representation:
  - put root at location 1
  - use an int variable (size) to store number of nodes
  - for a node at position i:
    - left child at position `2i` (if 2i <= size, else i is leaf)
    - right child at position `2i+1` (if 2i+1 <= size, else i is leaf)
    - parent is in position `floor(i/2)` (or use integer division)

Binary Heap:
ordering property

- In a heap, if X is a parent of Y, value(X) is less than or equal to value(Y).
  - the minimum value of the heap is always at the root.

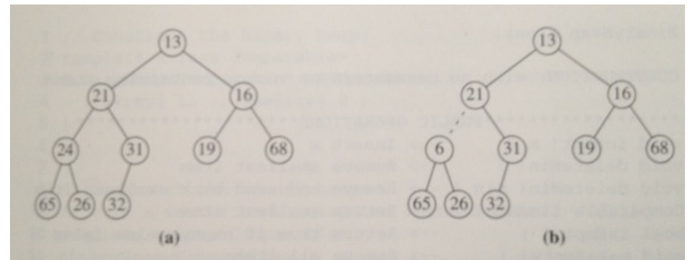


Figure 21.3 Two complete trees: (a) a heap; (b) not a heap.

Binary Heap: operations

- constructor, destructor
- isEmpty() (returns bool)
- makeEmpty()
- insert(x)
- findMin() (returns ItemType)
- deleteMin()

- Goal: logarithmic time (O(log n)) or better
- Must maintain heap properties after each operation

Heap class declaration

```
template<class ItemType>
class Heap_3358 {
public:

    Heap_3358();

    void makeEmpty();
    bool isEmpty() const;
    void insert(const ItemType &);
    ItemType findMin();
    void deleteMin();

private:
    int theSize;              //number of nodes in tree
    vector<ItemType> array;   //tree stored as array

};
```

Heap: simple methods

```
template<class ItemType>
Heap_3358<ItemType>::Heap_3358 ()
: array(11), theSize(0)
{ }

template<class ItemType>
void Heap_3358 <ItemType>::makeEmpty() {

    theSize = 0;
}

template<class ItemType>
bool Heap_3358 <ItemType>::isEmpty() const {

    return theSize==0;
}

template<class ItemType>
ItemType Heap_3358 <ItemType>::findMin() {

    assert(!isEmpty());
    return array[1];
}
```


Heap: insert(x)

- First: add a node to tree.
  - must be at next available location, size+1, in order to maintain a complete tree.
- Now maintain the ordering property:
  - if x is greater than its parent: done
  - else swap with parent
  - repeat
- Called “percolate up” or “reheap up”
- preserves ordering property
- O(log n), work is proportional to path length

Heap: insert(x)

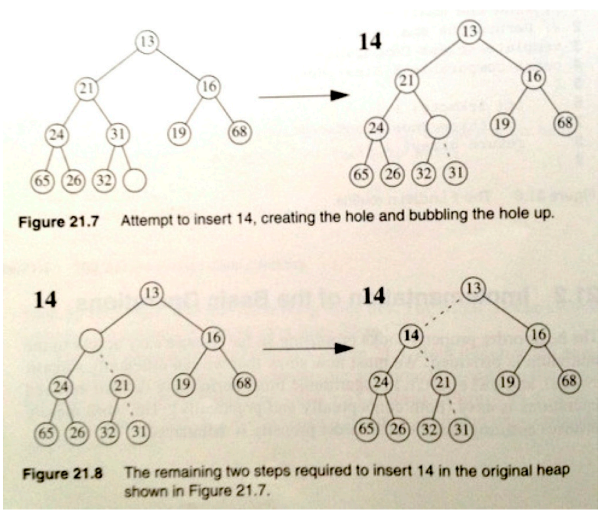


Figure 21.7 Attempt to insert 14, creating the hole and bubbling the hole up.

Figure 21.8 The remaining two steps required to insert 14 in the original heap shown in Figure 21.7.

Heap: insert(x)

```
template<class ItemType>
void Heap_3358 <ItemType>::insert(const ItemType& newItem)
{
   //make newItem the sentinel
   array[0] = newItem;

   //resize if necessary
   if (theSize+1 == array.size())
      array.resize(array.size()*2 + 1);

   //Percolate up
   theSize++;           //increment size
   int hole = theSize;  //the new location

   for (  ; newItem < array[hole/2]; hole=hole/2)  // hole/2=parent
     array[hole] = array[hole/2];    // move value down path ("swap")

   array[hole] = newItem;           // place in final spot

}
```

Places newItem in position 0, the parent of the root. Makes the loop stop if newItem is the new minimum.

Heap: deleteMin()

- Minimum is at the root, removing it leaves a hole.
- The last element in the tree must be relocated:
  - move last element up to the root
  - find smaller of the two children
  - if the smaller child is smaller than the parent: swap it with the parent, repeat
  - otherwise, we are done
- Called “percolate down” or “reheap down”
- preserves ordering property
- O(log n), work is proportional to path length

Heap: deleteMin()

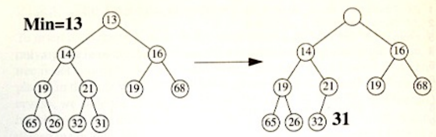


Figure 21.10 Creation of the hole at the root.

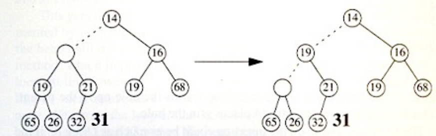


Figure 21.11 The next two steps in the `deleteMin` operation.

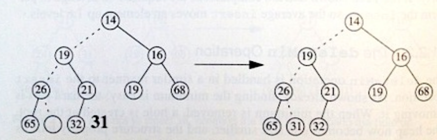


Figure 21.12 The Last two steps in the `deleteMin` operation.

Heap: deleteMin()

```
template<class ItemType>
void Heap_3358 <ItemType>::deleteMin()
{
   assert(!isEmpty());
   ItemType tmp = array[theSize]; //save this for final swap
   theSize--;

   //Percolate down
   int hole, child;
   for (hole = 1 ; hole*2 <= theSize; hole = child) {
      child = hole * 2;    // the left child

      // if there's a right child, compare and pick lesser
      if (child != theSize && array[child+1] < array[child])
         child++;

      if (array[child] < tmp)     // compare lesser child to parent
         array[hole] = array[child];    // if lesser, swap
      else
         break;
   }
   array[hole] = tmp;           // complete last swap
}
```


Heapsort

- Using a heap to sort a list:
  1. `insert` every item into a binary heap
  2. extract every item by calling `deleteMin` N times.
- Runtime Analysis: O(N log N)
  - step 1: `insert` is O(log N) and it’s done N times, so it’s O(N log N)
  - step 2: `deleteMin` is O(log N), and it’s done N times, so it’s O(N log N)

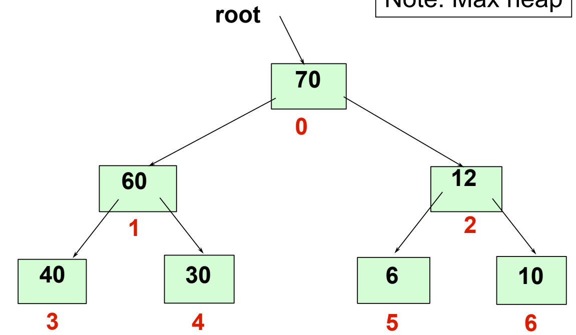
Heapsort

- Space analysis:
  - currently two arrays are needed:
  - one for heap, one for sorted list.
- If we use a Max heap (parent is always greater than children) then we can re-use the empty part of array for the sorted elements.
  - Then we need only one array.

After inserting all items into the heap

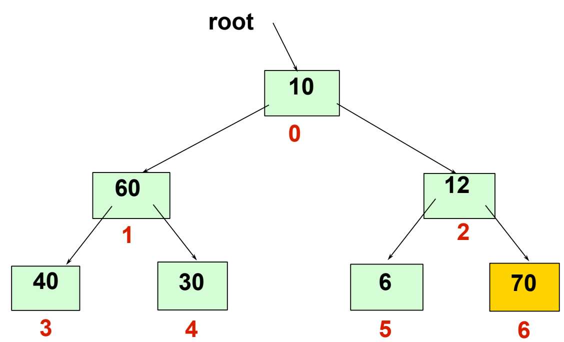
	values
[0]	70
[1]	60
[2]	12
[3]	40
[4]	30
[5]	6
[6]	10

Note: Max heap



After swapping root element into its place

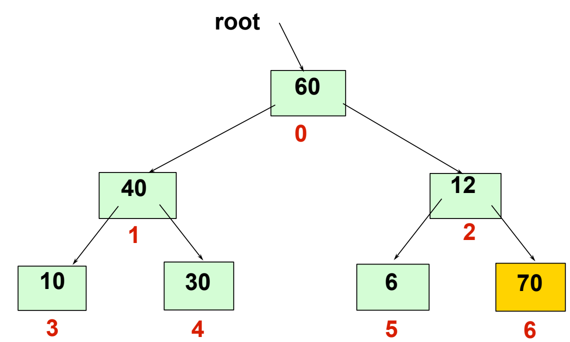
	values
[0]	10
[1]	60
[2]	12
[3]	40
[4]	30
[5]	6
[6]	70



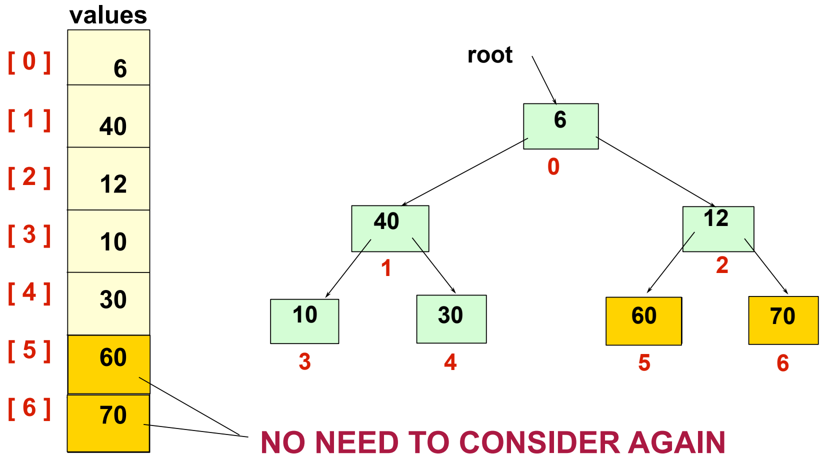
NO NEED TO CONSIDER AGAIN

After percolate down

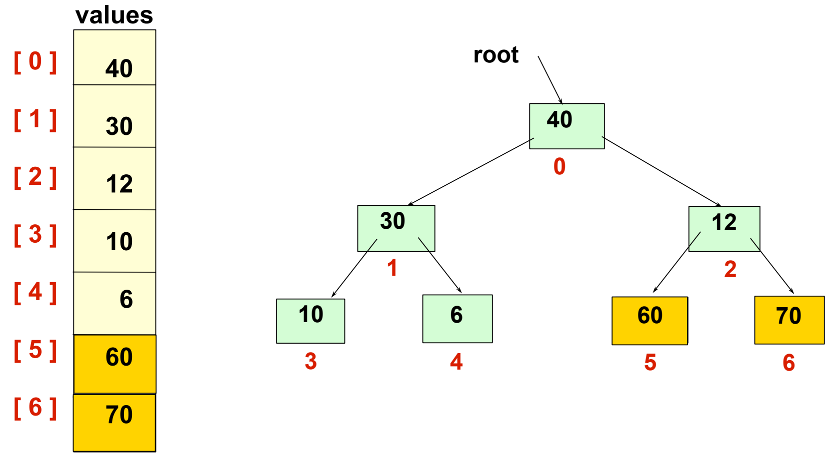
	values
[0]	60
[1]	40
[2]	12
[3]	10
[4]	30
[5]	6
[6]	70



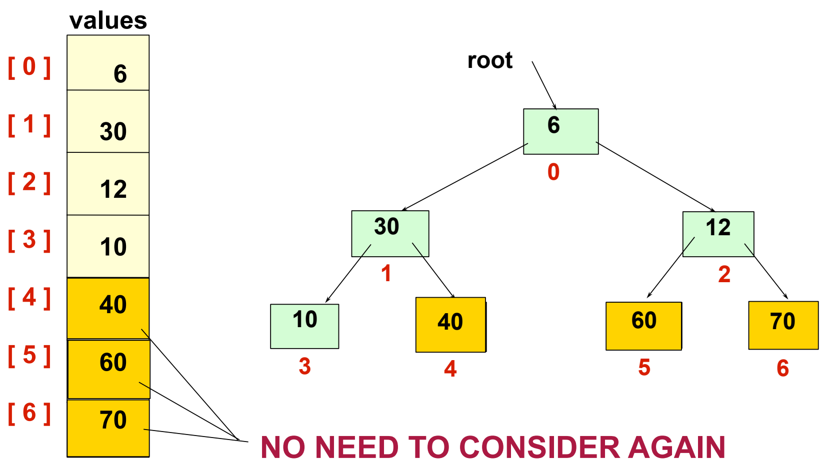
After swapping root element into its place



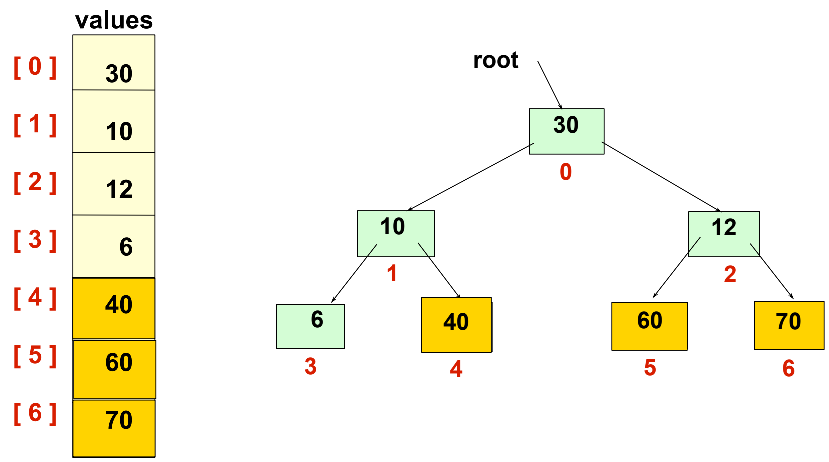
After percolate down



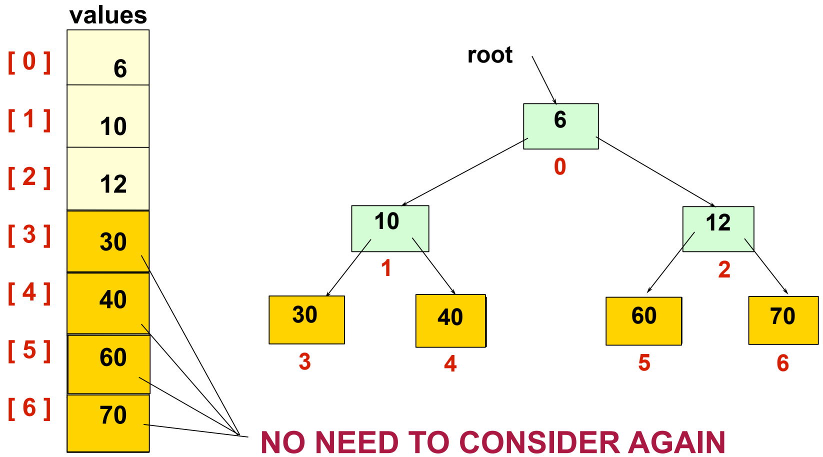
After swapping root element into its place



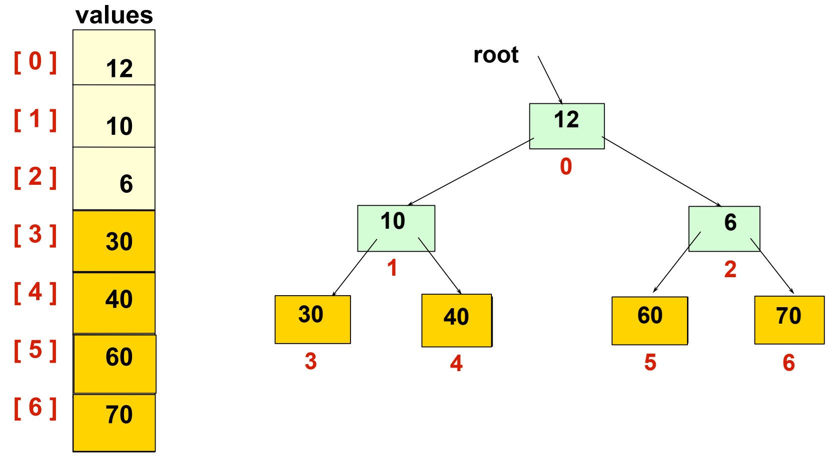
After percolate down



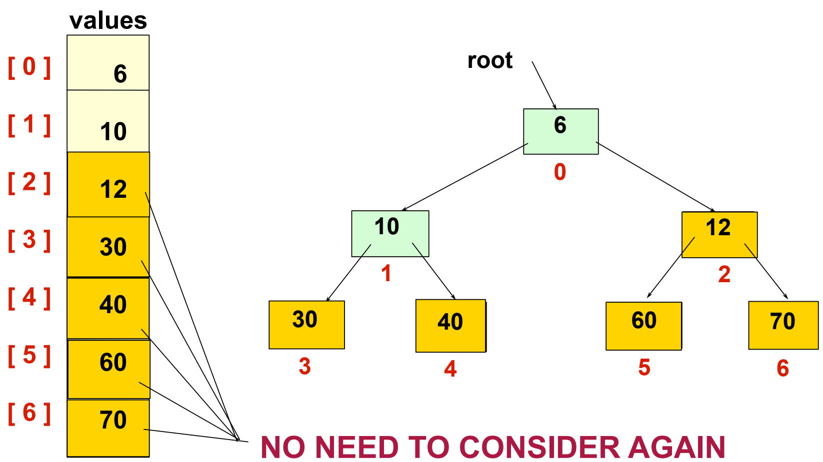
After swapping root element into its place



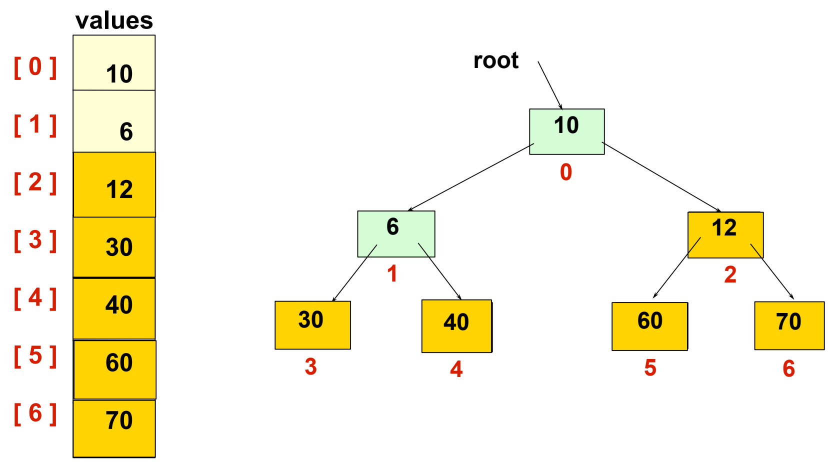
After percolate down



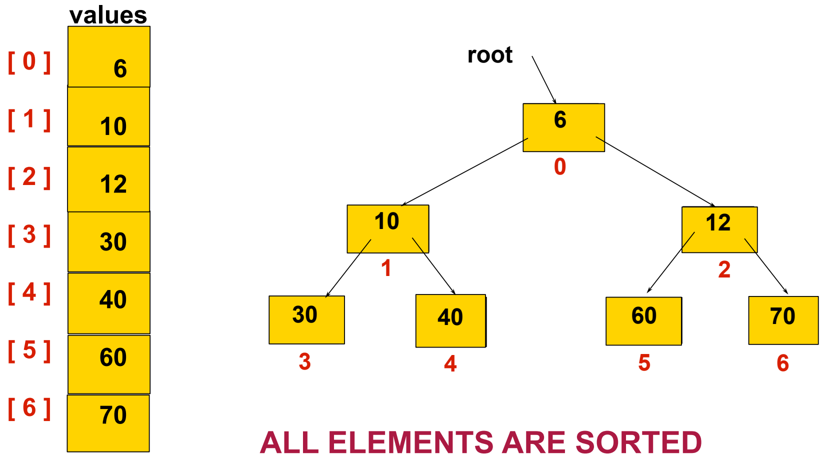
After swapping root element into its place



After percolate down



After swapping root element into its place



ALL ELEMENTS ARE SORTED