# 505 22240 / ESOE 2012　Data Structures: Lecture 09

## Binary Heap and Binary Search Trees

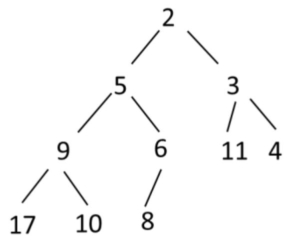
### § Binary Heap

・An implementation of priority queues.

・A binary heap is: ① a <u>complete</u> binary tree

・Complete binary tree: every row (level) is full, except bottom row, which is filled from left to right.

・e.g.



② Entries satisfy the <u>heap-order property</u>: no child has a key less than its parent's key.

・Often stored as arrays of entries by <u>level-order</u> traversal.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| ☒ | 2 | 5 | 3 | 9 | 6 | 11 | 4 | 17 | 10 | 8 |

left empty

・Mapping of nodes to indices: <u>level numbering</u>

Node i's children are 2i & 2i+1;

Node i's parent is *floor*(i / 2).

・Each tree node has two references (key, value), OR references an "Entry" object.

◎Operations

① Entry min( ); → Return entry at root. (heap-order property ensures this).

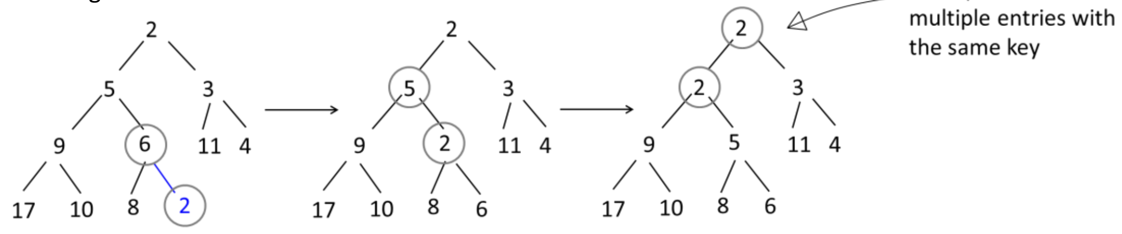
② template <typename K, typename V>

Entry insert(const K& k, const V& v)

Let x be a new entry (k, v), whose key is k and whose value is v.

Place x in bottom level of tree, at the first free spot from left; i.e., the first free location in array. (If bottom level is full, start a new level with x at the far left.)

・e.g.




★May violate the heap-order property

⇨ Solution:

・Entry bubbles up the tree until heap-order property is satisfied.

・<u>Repeat</u>:

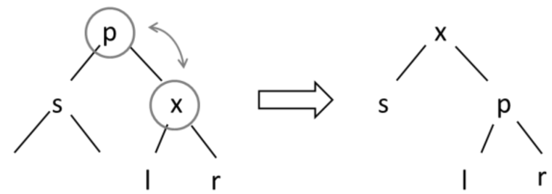
Compare x's key with its parent's key.

If x's key is less, exchange.

★After insertion, is the heap-order property satisfied?

⇨ YES, if the heap-order property was satisfied before the insertion.

・Exchange of x with a parent p during the insertion operation:



· We know that p ≤ s, where s is x's sibling.

p ≤ l
,where l and r are x's children.
p ≤ r

· We only swap if x < p → x < s.

⇨ p is the parent of l and r.

∴Tree rooted at x has the heap-order property.

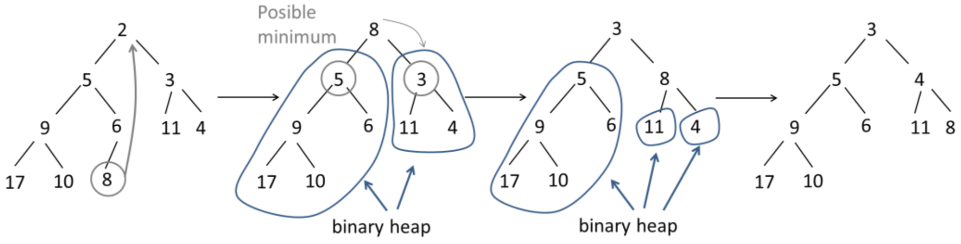
③ Entry removeMin( );

· Return null or throw an exception if the heap is empty.

· Otherwise, remove entry at root; save for return value.

· Fill hole with the last entry in tree, called "x".

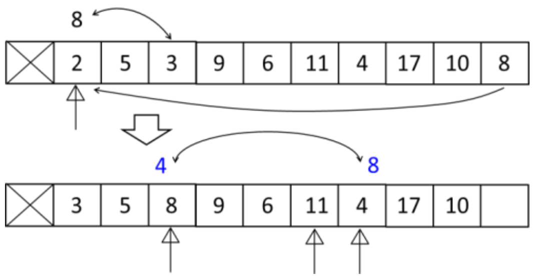
· e.g.




⇨Bubble x down the heap.

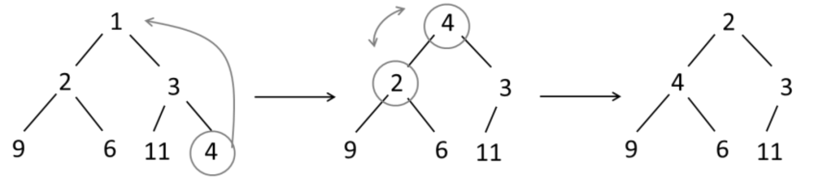
· Repeat:

If x > one or both of its children, swap x with its minimum child.



· Every subtree of a binary heap is a binary heap.

· Another example, not bubbling to leaf.



## ◎Running Times

| | Binary Heap | Sorted List | Unsorted List |
|---|---|---|---|
| min( ) | Θ(1) | Θ(1) | Θ(n) |
| insert( ) | | | |
| worse-case | Θ(log n) | Θ(n) | Θ(1) |
| best-case | Θ(1) | Θ(1) | Θ(1) |
| removeMin( ) | | | |
| worse-case | Θ(log n) | Θ(1) | Θ(n) |
| best-case | Θ(1) | Θ(1) | Θ(n) |

## ◎Binary Heap Running Times:

· insert( ):

· Takes O(1) time to compare x with parent.

· Complete binary tree: has at most $1+\log_2 n$ levels; n is # of entries in heap.

⇨Θ(log n) worst-case time.

## ◎Bottom-Up Heap Construction

· Given a bunch of randomly ordered entries, make a heap out of them.

⇨ Insert them one by one: O(n log n) time.

⇨ A faster way is described as follows:

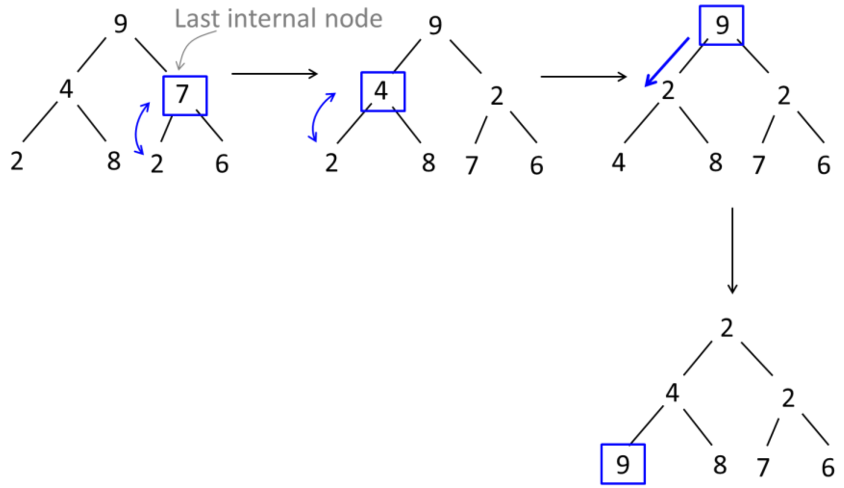
④ void bottomUpHeap( );

· Make a complete tree out of entries, in any order.

· Walk backward from the last internal node (non-leaf node) to root, i.e., reverse order in array.

· When we visit a node, bubble it down as in removeMin( ).

· e.g.




· The running time of bottomUpHeap() is tricky to derive.

· If each internal node bubbles all the way down, then the running time is proportional to the sum of the heights of all the nodes in the tree.

· It can be shown that this sum is less than n, where n is the number of entries being coalesced into a heap.

· Hence, the running time is in O(n), assuming two keys can be compared in O(1) time.

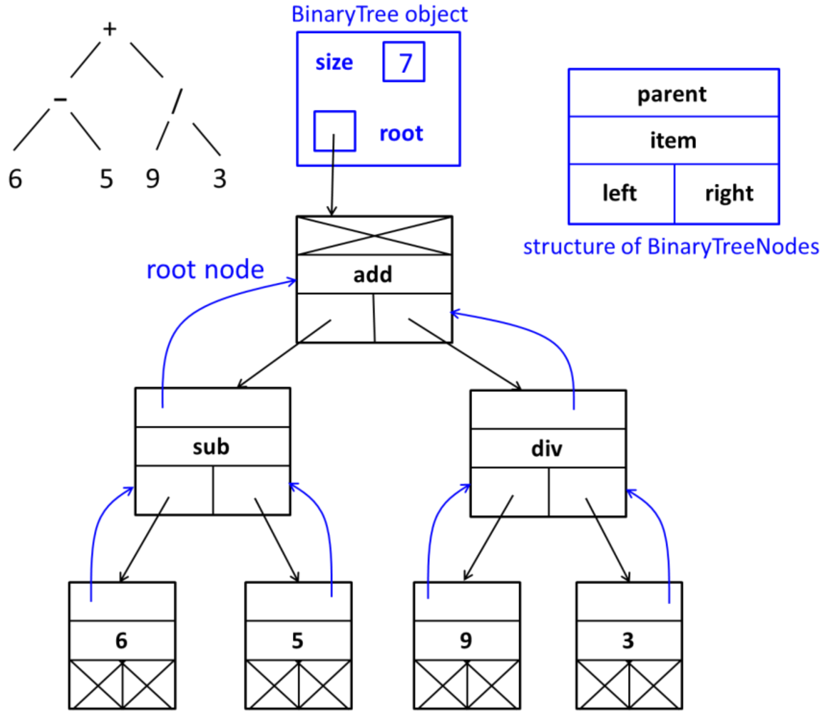
## § Binary Search Trees

· Binary Tree: a rooted tree wherein no node has more than two children.

### ◎Representing Binary Trees

```
template <typename E>
class BinaryTree {
public:
      BinaryTreeNode<E>* root;
      int size;
};
template <typename E>
class BinaryTreeNode {
public:
      E item;         // Entry entry;
      BinaryTreeNode<E>* parent;
      BinaryTreeNode<E>* left;
      BinaryTreeNode<E>* right;
      void inorder( ) {
            if (left != NULL) {
                  left->inorder( );
            }
            this->visit( );
            if (right != NULL) {
                  right->inorder( );
            }
      }
};
```

・e.g.




## ◎Binary Search Trees

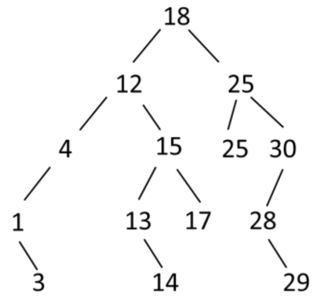
・<u>Ordered dictionary</u>: a dictionary in which keys have a total order, like in a heap.

・Insert, find, remove entries.

・Quickly find entry with minimum <u>or</u> maximum key, or entry nearest another entry.

・Simplest implementation of ordered dictionary is a binary search tree.

・e.g.




・left subtree of a node is the subtree rooted at the node's left child.

・right subtree is defined similarly.

★<u>Binary search tree invariant:</u>

For any node x,

every key in left subtree of x is ≦ x's key;

every key in right subtree of x is ≧ x's key.

・Inorder traversal of a binary search tree visits nodes in sorted order.