505 22240 / ESOE 2012 Data Structures: Lecture 8
Tree Traversals, Binary Trees and Priority Queues

§ Tree Traversal

· Traversal: a manner of visiting each node in a tree once.

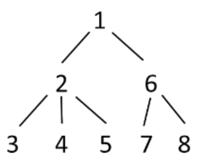
· There are several different traversals, each of which orders the nodes differently.

◎Preorder traversal: Visit each node before recursively visiting its children, which are visited from left to right. Root is visited first.

· Code: (**SibTreeNode class**)

```
void preorder( ) {
    this->visit( );
    // Whatever method to visit the node, e.g., print
    if (firstChild != NULL) {
        firstChild->preorder( );
    }
    if (nextSibling != NULL) {
        nextSibling->preorder( );
    }
}
```

· Visits nodes in this order: (suppose **visit( )** numbers the nodes in the order they're visited)



· Each node is visited only once, so a preorder traversal takes O(n) time, when n is # of nodes in tree. All the traversals we will consider take O(n) time.

· Output directory structure (Preorder):

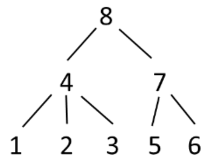
```
~esoe/ds2012
    hw
        hw1
        hw2
    index.html
    lab
        lab1
        lab2
    lec
        01
        02
        03
        04
        05
```

◎Postorder traversal: Visit each node's children in left-to-right order before the node itself.

· Code: (**SibTreeNode class**)

```
void postorder( ) {
    if (firstChild != NULL) {
        firstChild->postorder( );
    }
    this->visit( );
    if (nextSibling != NULL) {
        nextSibling->postorder( );
    }
}
```

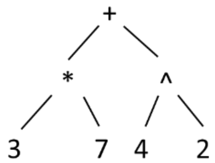
・Visiting order:



・Natural way to sum total disk space used in the root directory and its descendants.

◎<u>Inorder</u> traversal: (for binary trees) visit left child, then node, then right child.

・e.g. expression tree:



Inorder: 3 * 7 + 4 ^ 2 **easy to read for human**

Preorder: + * 3 7 ^ 4 2 **good for computers**

Postorder: 3 7 * 4 2 ^ + **good for computers**

◎<u>Level-order</u> traversal: visit root, then all depth-1 nodes (left to right), then all depth-2 nodes, etc.

・e.g. expression tree ⇨ "+ * ^ 3 7 4 2" **no meaning**

・Not recursive

⇨ Use a <u>queue</u>, which initially contains only the root.

<u>Repeat</u>:

- ・Dequeue a node.
- ・Visit it.
- ・Enqueue its children (left to right).

<u>Until</u> queue is empty

・e.g. expression tree + * ^ 3 7 4 2

・<u>Remark</u>: if you use a stack instead of a queue, and push each node's children in reverse order (from right to left), so that they pop off the stack in order from left to right. — you perform a preorder traversal. Think about *why*.

§ Binary Tree Construction

・Suppose that the elements in a binary tree are distinct.

・Can you construct the binary tree from which a given traversal sequence came?

・When a traversal sequence has more than one element, the binary tree is NOT uniquely defined.

・Therefore, the tree from which the sequence was obtained cannot be reconstructed uniquely.

・Can you construct the binary tree, given two traversal sequences?

・It depends on which two sequences are given.

◎Preorder and Postorder

・preorder = ab

・postorder = ba

・Preorder and postorder do not uniquely define a binary tree.

・Nor do preorder and level order (same example).

・Nor do postorder and level order (same example).

◎Inorder and Preorder

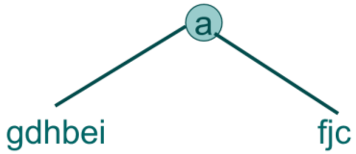
・Given two sequences of inorder and preorder:

inorder = **g d h b e i a f j c**

preorder = **a b d g h e i c f j**

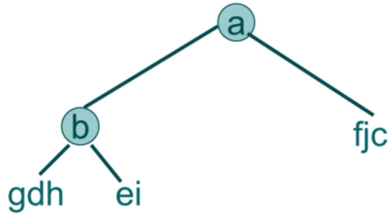
・Scan the preorder left to right using the inorder to separate left and right subtrees.

· **a** is the root of the tree; **gdhbei** are in the left subtree; **fjc** are in the right subtree.



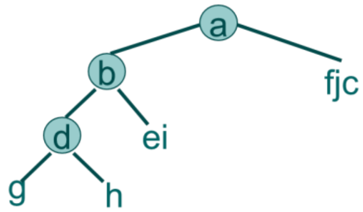
· preorder = **a b d g h e i c f j**

· **b** is the next root; **gdh** are in the left subtree; **ei** are in the right subtree.



· preorder = **a b d g h e i c f j**

· **d** is the next root; **g** is in the left subtree; **h** is in the right subtree.

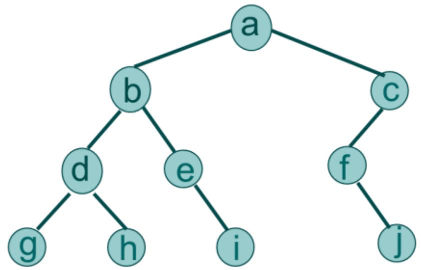


· preorder = **a b d g h e i c f j**

· **c** is the next root; **f** is in the left subtree; **j** is in the right subtree of **f**.

· inorder = **g d h b e i a f j c**

· **e** is the next root; **i** is in the right subtree.



◎Inorder and Postorder

· Scan postorder from right to left using inorder to separate left and right subtrees.

· inorder = **g d h b e i a f j c**

· postorder = **g h d i e b j f c a**

· Tree root is **a**; **gdhbei** are in left subtree; **fjc** are in right subtree.

◎Inorder And Level-Order

· Scan level order from left to right using inorder to separate left and right subtrees.

· inorder = **g d h b e i a f j c**

· level order = **a b c d e f g h i j**

· Tree root is **a**; **gdhbei** are in left subtree; **fjc** are in right subtree.

§ Priority Queues

· A priority queue is an ADT for storing a collection of prioritized elements (according to their priorities).

· Contains Entries, each consists of *key* and *value*.

· A total order (either in increasing or decreasing) is defined on the keys.

◎Operations

・Identify or remove entry whose key is lowest (minimum), but no other entry.

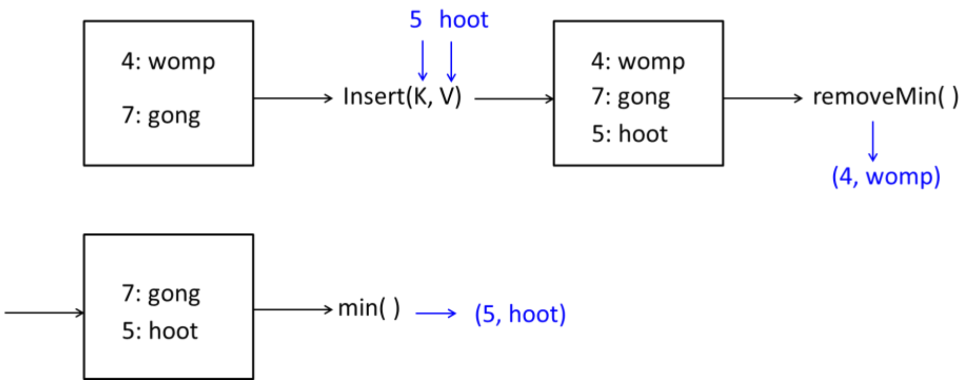
・Any key may be inserted at any time.

—`insert( )` adds entry to the priority queue.

—`min( )` returns entry with minimum key.

—`removeMin( )` both removes and returns entry with minimum key.

・e.g.



★Commonly used as "event queues" in simulations.

・Key is the time event takes place.

・Value is description of event.

・A simulation operates by removing successive events from the queue.

・This is why most priority queues return the minimum, rather than maximum, key.

・We want to simulate the events that occur first first.

◎Priority Queue Interface

```
template <typename K, typename V >
class PriorityQueue {
public:
      int size( ) const;
      bool isEmpty( ) const;
      Entry& insert(const K& k, const V& v);
      const Entry& min( ) const throw(QueueEmpty);
      Entry& removeMin( ) throw(QueueEmpty);
};
```