Tree Data Structures

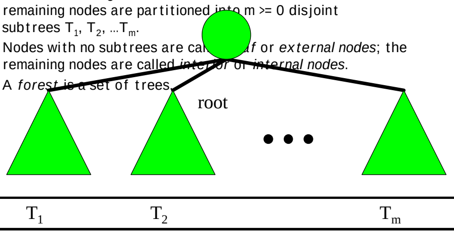
Reading Assignment
Chapters 6, 9.1

Applications

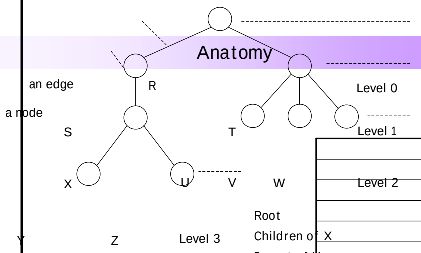
- A tree imposes a hierarchical structure on a collection of items. These items are represented as nodes and relationships among the nodes and edges.
- Examples
  - Pedigree
  - History of a tournament
  - Syntax of expressions, syntax trees, programs
  - Organization charts
- The number of children per node define tree types
  - binary, ternary, quaternary, and n-ary trees
- Other important tree structures types we will look at
  - Search trees, B-trees
  - Heaps
  - 2-3 trees, AVL-trees
  - Red-black trees, And-Or trees
  - Fibonacci heaps

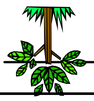
Definitions

- A *tree* is a recursive data structure.
- A finite rooted tree T is defined as a finite set of nodes such that there is a distinguished node, called the root of the tree, and remaining nodes are partitioned into m >= 0 disjoint subtrees T_1, T_2, …T_m.
- Nodes with no subtrees are called *leaf* or *external nodes*; the remaining nodes are called *interior* or *internal nodes*.
- A *forest* is a set of trees.



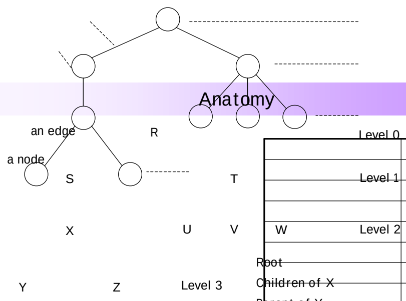
Anatomy

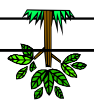




Root	
Children of X	
Parent of Y	
Descendants of S	
Ancestors of X	
Siblings of W	
Internal nodes	
Leaf or external	
Subtree rooted at S	

Anatomy





Root	**R**
Children of X	**Y, Z**
Parent of Y	**X**
Descendants of S	**X, Y, Z**
Ancestors of X	**S, R**
Siblings of W	**U, V**
Internal nodes	**R, S, X, T**
Leaves or external	**Y, Z, U, V, W**
Subtree rooted at S	**S, X, Y, Z**

Depth of a Node

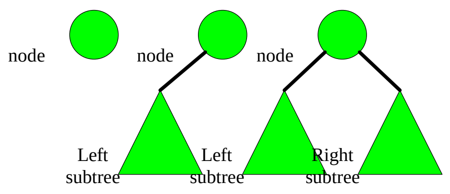
- Let v be a node of a tree T.
  - Non-recursive definition of *depth*
    - The depth of v is the number of ancestors of v, excluding v itself.
  - Recursive definition of *depth*
    - If v is the root, then the depth of v is zero.
    - Otherwise, the depth of v is one plus the depth of its parent.
  - Thus, the depth of root is zero.

Height of a Tree

- Let v be a node of a tree T.
  - ✍ Recursive definition of the *height* of a node
    - If v is an external node, then the height of v is zero.
    - Otherwise, the height of v is one plus the max. height of a child of v
- The height of a Tree is the height of the root of T.
  - ✍ Note: The *height* of tree T is equal to the maximum of all the depths of its nodes.

Binary trees

- A binary tree is an ordered tree in which every node has at most two children (i.e., 0, 1, or 2 children).
- Recursive definition
  - ✍ A binary tree is empty, or consists of a node, or a node and a left subtree, or a node and a left subtree and a right subtree.



Properties of Binary Trees

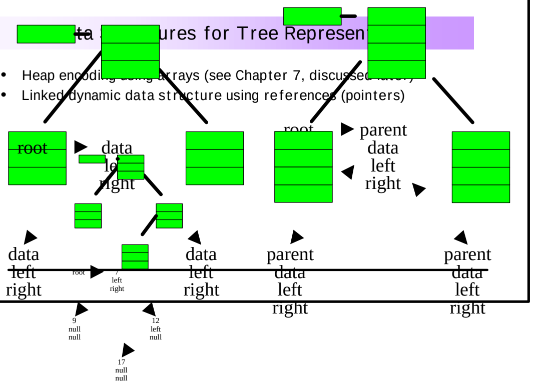
- The set of all nodes of a tree T at the same depth d is called the *level d* of T.
- In a binary tree,
  - ✍ Level 0 has 1 node
  - ✍ Level 1 has at most 2 nodes
  - ✍ Level 2 has at most 4 nodes
  - ✍ Level 3 has at most 8 nodes
  - ✍ Level 4 has at most 16 nodes
  - ✍ ...
  - ✍ Level d has at most 2^d nodes
  - ✍ A proper binary tree T is one where all the levels are full

Properties of Binary Trees

- Let T be a binary tree with n nodes, and let h denote the height of T. Then T has the following properties:
  - ✍ The number of leaf or external nodes in T is at least h+1 and at most 2^h
  - ✍ The number of internal nodes in T is at least h and at most 2^h-1
  - ✍ The total number of nodes in T is at least 2h + 1 and at most $2^{h+1} - 1$
  - ✍ The height of T is at least lg(n+1) -1 and at most (n-1)/2, that is, log(n+1)-1 <= (n-1)/2
  - ✍ The number of leaf nodes in T is one more than the number of internal nodes.
- A list is a degenerate binary tree.
- N-ary trees can always be converted to a binary tree.

ta S ures for Tree Represen

- Heap encoding using arrays (see Chapter 7, discussed later)
- Linked dynamic data structure using references (pointers)



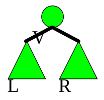
Binary Tree Node Class

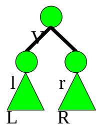
```
public class Node {

   private Node parent;
   private Object data;
   private Node left;
   private Node right;

public Node() {
   this(null, null, null, null);
}
public Node(Object data) {
   this(data, null, null, null);
}
public Node(Object data,
            Node parent,
            Node left,
            Node right) {
   this.data = data;
   this.parent = parent;
   this.left = left;
   this.right = right;
}
```

```
public Object getData() { return data; }
public Node getLeft() { return left; }
public Node getParent() { return parent; }
public Node getRight() { return right; }
public boolean isLeaf() {
   return left==null && right==null; }
public void setData(Object data) {
   this.data = data; }
public void setLeft(Node left) {
   this.left = left; }
public void setParent(Node parent) {
   this.parent = parent; }
public void setRight(Node right) {
   this.right = right; }
public String toString() { return data; }

}
```

Tree Traversals for Binary Trees

- Preorder (depth-first)
  1. V
  2. L
  3. R
- Inorder (symmetric)
  1. L
  2. V
  3. R
- Postorder (bottom-up)
  1. L
  2. R
  3. V
- Level order (breadth-first)
  1. V
  2. l
  3. r
  4. L
  5. R

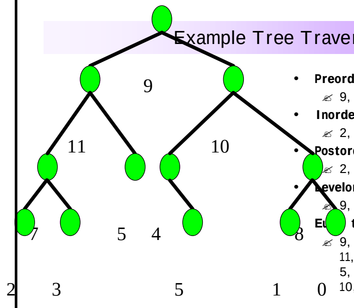




Euler Tour

- Euler tour
  1. V
  2. L
  3. V
  4. R
  5. V
- Euler tour integrates three traversals
  - Preorder
  - Inorder
  - Postorder

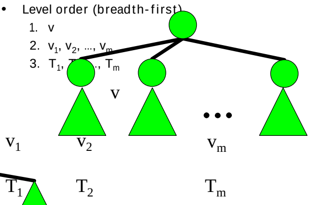
Example Tree Traversals

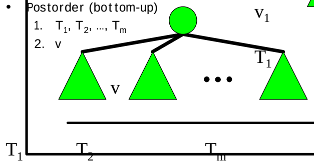


- **Preorder**
  - 9, 11, 7, 2, 3, 5, 10, 4, 5, 8, 1, 0
- **Inorder**
  - 2, 7, 3, 11, 5, 9, 4, 5, 10, 1, 8, 0
- **Postorder**
  - 2, 3, 7, 5, 11, 5, 4, , 0, 8, 10, 9
- **Levelorder**
  - 9, 11, 10, 7, 5, 4, 8, 2, 3, 5, 1, 0
- **Eu tour**
  - 9, 11, 7, 2, 2, 2, 7, 3, 3, 3, 7, 11, 5, 5, 5, 11, 9, 10, 4, 4, 5, 5, 5, 4, 10, 8, 1, 1, 1, 8, 0, 0, 0, 8, 10, 9

Tree Traversals for n-ary Trees

- Preorder (depth-first)
  1. v
  2. $T_1, T_2, \ldots, T_m$
- Inorder (symmetric)
  1. T_1
  2. v
  3. $T_2, T_3, \ldots, T_m$
- Postorder (bottom-up)
  1. $T_1, T_2, \ldots, T_m$
  2. v
- Level order (breadth-first)
  1. v
  2. $v_1, v_2, \ldots, v_m$
  3. $T_1, T_2, \ldots, T_m$





Pre-, In-, Postorder Binary Tree Traversal Algorithms

```
void preorder(Node t) {
   if (t != null) {
      processNode(t);
      preorder(t.getLeft());
      preorder(t.getRight());
   }
}
void inorder(Node t) {
   if (t != null) {
      inorder(t.getLeft());
      processNode(t);
      inorder(t.getRight());
   }
}
```

```
void postorder(Node t) {
   if (t != null) {
      postorder(t.getLeft());
      postorder(t.getRight());
      processNode(t);
   }
}
```

Levelorder Binary Tree Traversal Algorithm

```
void levelorder(Node t) {
   Queue q = new Queue();
   q.enqueue(t);
   while (!q.empty()) {
      t = q.dequeue();
      processNode(t);
      Node left = t.getLeft();
      if (left != null)
         q.enqueue(left);
      Node right = t.getRight();
      if (right != null)
         q.enqueue(right);
   }
}
```

Non-recursive Preorder Traversal Algorithm

```
void nrpreorder(Node t) {
   Stack s = new Stack();
   s.push(t);
   while (!s.empty()) {
      t = s.pop();
      processNode(t);
       Node right = t.getRight();
      if (right != null)
         s.push(right);
      Node left = t.getLeft();
      if (left != null)
         s.push(left);

   }
}
```

Time Complexity of Tree Traversals

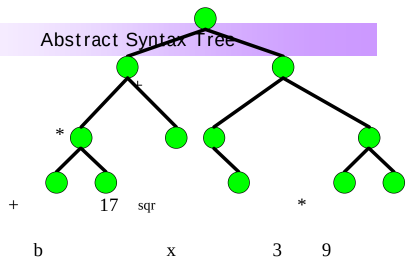
- The running times of tree traversals are easy to analyze
- Assuming a node takes O(1) to process
- Thus, we spend a constant amount of time at each node
- Thus, the overall running time is O(n) for a tree of n nodes

Theorem.
The time complexity of the tree traversals *preorder*, *inorder*, *postorder*, *levelorder*, and *Euler Tour* is *O(n).*

Binary Tree Interface

```
public interface BinaryTree {
  void add(Object s);
  void remove(Object s);
  Node getRoot();
  boolean isEmpty();
  void makeEmpty();
  Enumeration levelorderIterator();
  Enumeration postorderIterator();
  Enumeration inorderIterator();
  Enumeration preorderIterator();
  boolean search(Object s);
  int size();
  int height();
  String toString();
}
```

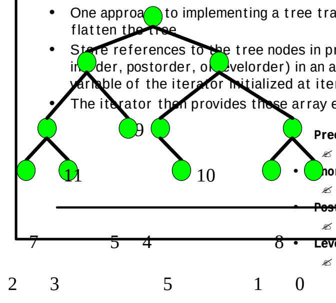
Abstract Syntax Tree



- **Preorder**
  - ✍ + * + a b 17 + sqr x * 3 9
- **Inorder**
  - ✍ a + b * 17 + sqr x + 3 * 9
- **Height of tree**
  - ✍ 3
- **Postorder**
  - ✍ a b + 17 * x sqr 3 9 * + +
- **Levelorder**
  - ✍ + * + + 17 sqr * a b x 3 9
- **Depth of x**
  - ✍ 3

Implementing Tree Iterators

- One approach to implementing a tree traversal iterator is to flatten the tree
- Store references to the tree nodes in proper order (i.e., preorder, inorder, postorder, or levelorder) in an array, which is an instance variable of the iterator initialized at iterator instantiation time
- The iterator then provides those array elements in order



- **Preorder array**
  - ✍ 9, 11, 7, 2, 3, 5, 10, 4, 5, 8, 1, 0
- **Inorder array**
  - ✍ 2, 7, 3, 11, 5, 9, 4, 5, 10, 1, 8, 0
- **Postorder array**
  - ✍ 2, 3, 7, 5, 11, 5, 4, , 0, 8, 10, 9
- **Levelorder array**
  - ✍ 9, 11, 10, 7, 5, 4, 8, 2, 3, 5, 1, 0

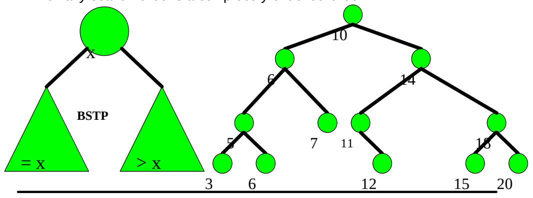
Inner Class Preorder Iterator

```
private class PreorderIterator
      implements Enumeration {
   private int curNode;
   private int curSize;
   private Node[] pre;

   public PreorderIterator() {
      pre = new Node[size];
      curNode = 0;
      curSize = size;
      preorder(root);
      curNode = 0;
   }
```

```
private void preorder(Node t) {
   if (t != null) {
      pre[curNode] = t;
      curNode++;
      preorder(t.getLeft());
      preorder(t.getRight());
   }
}
public boolean hasMoreElements() {
   return curNode < curSize;
}
public Object nextElement() {
   int c = curNode;
   curNode++;
   return pre[c];
}

} // private class
```

Binary Search Trees

- *Binary search tree property (BSTP)*
  - ✍ A binary search tree is a binary tree in which the nodes are labelled with elements of a set such that all elements in the left subtree of a node labelled x are less than or equal to x and all the elements in the right subtree are greater than x.
- A binary search tree is a completely ordered tree.



Implementing Dictionary Operations

- Dictionary operations
  - ✍ **member(), insert(), delete()**
- To implement these operations for the generic type Object, we would have to implement a Comparator object
- In the next few slides, we use String instead of Object to simplify the **member(), insert(),** and **delete()** operations

Implementing Dictionary Operations

- Options for supporting the **delete()** operation
  - ✍ Delete element, reorganize the tree until binary search tree property is re-established
    - If the node to be deleted has only one subtree, move that subtree into its place
    - If the node to be deleted has two subtrees, then move the node with the smallest or largest value in the subtree into the position of the deleted node
  - ✍ Mark nodes as deleted, but keep nodes in the tree for searching purposes
    - Add a **boolean** field **deleted** in the Node class
    - All routines must take deleted field into account

Search for a Node in a Binary Search Tree

```
boolean searchTree(Node t, String x) {
   if (t == null) {
      return false;
   } else if (!t.getDeleted() &&
      x.compareTo(t.getData()) == 0) {
      return true;
   } else if (x.compareTo(t.getData()) <= 0) {
      Node left = t.getLeft();
      if (left == null) return false;
      else return searchTree(left, x);
   } else {
      Node right = t.getRight();
      if (right == null) return false;
      else return searchTree(right, x);
   }
}
```

Mark a Node in Binary Search Tree as Deleted

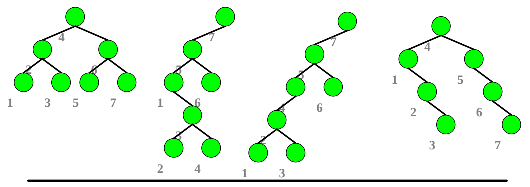
```
void delete(Node t, String x) {
  if (t == null) {
     return;
  } else if (x.equals(t.getData())) {
     t.setDeleted(true);
     size--;
  } else if (x.compareTo(t.getData()) <= 0) {
     delete(t.getLeft(), x);
  } else {
     delete(t.getRight(), x);
  }
}
```

Insert a Node into Binary Search Tree

```
void insert(Node t, Node x) {
  if (t == null) {
    return;
  } else if (t.getDeleted() &&
   x.getData().compareTo(t.getData()) == 0) {
     t.setDeleted(false);
  } else if (x.getData().compareTo(t.getData()) <= 0) {
     Node left = t.getLeft();
     if (left == null) {
          t.setLeft(x);
          x.setParent(t);
     } else insert(left, x);
  } else {
     Node right = t.getRight();
     if (right == null) {
          t.setRight(x);
          x.setParent(x);
     } else insert(right, x);
  }
}
```

Inorder Traversal of Binary Search Trees

- Consider the following sequence of integers
  - ✍ 1, 2, 3, 4, 5, 6, 7
- In which order do these integers have to be inserted into a binary search tree to produce the following trees?
- Execute inorder traversal on all of these binary search trees. What is the output? ✍



Balanced Binary Search Trees

- In a balanced binary search tree, the levels are full except the last one
- A list is a binary tree
  - ✍ **member, insert(),** and **delete()** operations take O(n) time in such a degenerate binary tree
- To guarantee O(log n) access times, the tree must be kept balanced
- There are many types of balanced binary search trees
  - ✍ AVL trees
  - ✍ 2-3 trees
  - ✍ Multi-way trees
  - ✍ Red-black trees
  - ✍ B-trees
- The time complexity of **member, insert(),** and **delete()** operations in a balanced binary search tree is O(log n)

Summary

- Tree applications
  - ✍ Abstract syntax trees
- Definitions and anatomy
- Trees, binary trees, n-ary trees, recursive data structures
- Traversals
  - ✍ Preorder, inorder, postorder
  - ✍ Levelorder, Euler tour
  - ✍ Recursive and iterative algorithms
- Representation
  - ✍ Linked data structures
  - ✍ Heap (next section)
- Iterator
  - ✍ Flatten tree
- Binary search trees
  - ✍ Binary search tree property
  - ✍ Balanced binary search trees