Binary Trees

- Unlike previous data structures, trees are *nonlinear*.
- A binary tree is a finite set of nodes. The set might be empty (no nodes, which is called the empty tree). But if the tree is not empty, it follows the following rules:
  1. There is a distinguished node, called the root.
  2. Each node may be associated with two child nodes, called the left child and the right child.
  3. Each node, except the root, has exactly one parent; the root has no parent.

Examples

```
    A
   /|
  B  C
 /|
D  E F  G
```
Terminology

- leaf (terminal) – node with no children
- parent, child, sibling, ancestor, descendant
- level of a node – root at level 1, the child(ren) of root at level 2, and so forth.
- height of a tree – the maximum level of any of its nodes
- full binary trees – every leaf is at the same level and every non-leaf node has two children
- complete binary trees – a complete binary tree of height \( K \) is a full binary tree of height \( K - 1 \) plus a set of leaves at level \( K \) occupying the leftmost positions.
Binary Tree Properties

- The maximum number of nodes on level $i$ of a binary tree is $2^{i-1}, i \geq 1$
- The maximum number of nodes in a binary tree of height $K$ is $2^K - 1, k \geq 1$
- The minimum number of nodes in a binary tree of height $K$ is ?

Another Definition: Recursive Thinking

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees, called the left subtree and the right subtree.
CS 310 Binary Trees, Page 9

**Pointer Representation**

A

B C

D E F G

H I J

CS 310 Binary Trees, Page 10

**C++ Implementation**

```cpp
struct Binary_node {
    int data;
    Binary_node *left;
    Binary_node *right;
    Binary_node() { data = 0; left = right = NULL; }
    Binary_node (const int x) {
        data = x; left = right = NULL; }
};
```

Collapsing the two constructors using default parameter value:

```cpp
Binary_node (const int x = 0) {
    data = x; left = right = NULL; }
```
class Binary_tree {
private:
    Binary_node* root;
    // We may need to add auxiliary functions later on.

public:
    Binary_tree () { root=NULL; }
    ~Binary_tree ();
    bool empty() const { return root==NULL; }
    int height() const;
    void insert (const int); // not discussed in this talk
    // Tree traversal methods
    void preorder ();
    void inorder ();
    void postorder ();
};

Computing Tree Height

First, we add an auxiliary (and private) function:

int Binary_tree::height (Binary_node *r)
{
    if (r == NULL) return 0;
}
Let us consider the tree on page 3.

The public \texttt{height()} function simply calls upon its private counterpart to get the real job done.

\begin{verbatim}
int Binary_tree::height ()
{
    return height (root);
}
\end{verbatim}
**Traversals**

- Inorder – left subtree, root, right subtree
- Preorder – root, left subtree, right subtree
- Postorder – left subtree, right subtree, root

**Inorder**

```cpp
void Binary_tree::inorder (Binary_node* p)
{
    if (p != NULL)
    {
        inorder (p->left);
        cout << p->data << '
';
        inorder (p->right);
    }
}
```
void Binary_tree::preorder (Binary_node* p) 
{ 
    if (p != NULL) 
    { 
        cout << p->data << '\n';
        preorder (p->left);
        preorder (p->right);
    }
}

void Binary_tree::postorder (Binary_node* p) 
{ 
    if (p != NULL) 
    { 
        postorder (p->left);
        postorder (p->right);
        cout << p->data << '\n';
    }
}
Examples

Let use the tree shown on page 3.

- inorder:
- preorder:
- postorder:

And the tree shown in page 9.

- inorder:
- preorder:
- postorder:

Binary Search Tree

A binary search tree is a binary tree that is either empty or satisfies the following conditions:

- The root is greater than all nodes in its left subtree.
- The root is less than all nodes in its right subtree.
- The left and right subtrees are also binary search trees.

We do not allow identical nodes.
Examples

```
    7
   / \
  3   10
 / \   / \
2   4  8  13
   \   \   
    5  12  14
```

C++ Implementation

```cpp
class Search_tree {
    private:
        Binary_node* root;
        // We may need to add auxiliary functions later on.
    
    public:
        Search_tree () { root=NULL; }
        ~Search_tree ();
        void search (const int) const;
        void insert (const int);
        void remove (const int);
        // height, empty, traversals, etc.
};
```
Searching in a BST

Again, the public search function simply calls an auxiliary, private function, using root as the first parameter.

```cpp
bool Search_tree::search (Binary_node *p, int k) {
}
```

Insertion into A BST

```cpp
bool Search_tree::insert (Binary_node*& p, int k) {
}
```
**Example**

Draw the BST resulted from inserting the sequence 25, 49, 30, 8, 66, 33, 45, 79, 12, and 29.
Do the same for 0, 1, 2, 3, 4, and 5.

**Deleting from A BST**

Case 1: No Children

Case 2: No Left Child
Case 3: No Right Child
Case 4: Two Children

bool Search_tree::remove (Binary_node*& root, int target) {
    // Search for target

    if (root == NULL)
        return false;

    if (root->data > target)
        return remove (root->left, target);

    if (root->data < target)
        return remove (root->right, target);

    if (root->data == target) {
        // Handle removal...
    }

    return true;  // Return true to indicate successful removal
}
/ root->_data == target (target found) 
// Let us see how many children the root has

if (root->left==NULL)
{
    root = root->right;
    return true;
}

if (root->right==NULL)
{
    root = root->left;
    return true;
}

// root has two children
Binary_node* p = root->left;
while (p->right != NULL)
    p = p->right;

int tmp = p->data;
p->data = root->data;
r->data = tmp;
return remove (root->left, target);
} /* end of remove */
Handling Two-Children Cases without Copying

Binary_node* p = root->left;

while (p->right != NULL)
{
    p = p->right;
}

ListNode* r=p->left;