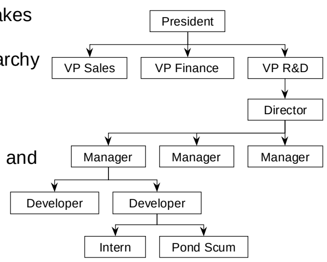
Trees [Chapter 13]

Data Structures So Far...

- Most have been linear (ordered)
  - The items in the collection can be arranged in a line
  - Lists, Stacks, Queues, Arrays
- Some have been unordered
  - There's no specific arrangement of the items relative to each other
  - Sets, Tables
- Are there other kinds of data structures?

Hierarchical Data

- Sometimes, data makes more sense when arranged into a hierarchy
- Example: the organization of a company
- Example: directories and files
- Example: a class hierarchy

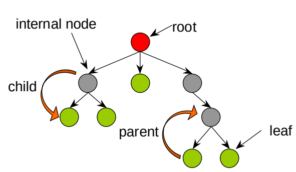


The Tree ADT

- In a hierarchical collection, every item may have more than one successor
- We use the Tree ADT to describe hierarchical data
  - Every node in the tree maintains some number of pointers to its children

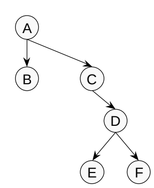
Tree Terminology

- Every node has zero or more children
  - >0 children: internal node
  - No children: leaf node
- Every node has exactly one parent, except a distinguished node called the "root"
- If B is a child of A, then A is the parent of B



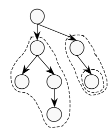
Ancestors And Descendants

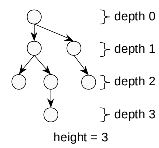
- Descendant: recursive definition!
  - A is a descendant of A
  - If B is a descendant of A and B is a parent of C, then C is a descendant of A
  - What are C's descendants? B's?
- Ancestor: similar definition
  - Or: A is an ancestor of B precisely when B is a descendant of A
  - What are E's ancestors?



Subtrees, Depth And Height

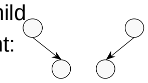
- A subtree is a node plus all of its descendants (part of tree with that node as root)
- Depth (of a node):
  - Depth of root node is 0
  - Depth of a node is 1 + depth of its parent
  - Different from "level" in textbook
- Height (of a tree):
  - Height is maximum depth for any node in the tree
  - Differs by one from book definition!





Binary Trees

- A restriction on the general form of trees
  - General form is complicated: must maintain a list of pointers to children
  - Restriction often gives us all the power we need
- Every node has at most two children
- The children have special names and places: the *left* child and the *right* child
- These two trees are different:



A Binary Tree ADT

- Like linked lists, we'll use two structures: a **`struct`** to hold data for a single node, and a **`class`** that abstracts the tree itself
- Another recursive data structure
  - So expect the algorithms to be highly recursive!
- We'd probably store size and height explicitly, but let's compute them…

```
struct Node {
    int data;

    Node *left;
    Node *right;
};

class BinaryTree
{
public:
    BinaryTree();
    ~BinaryTree();

    int calcSize();
    int calcHeight();
    …
private:
    Node *root;
};
```


Counting Nodes

- Add a private helper function: **`BinaryTree::calcSizeFrom`**

```
int BinaryTree::calcSize()
{
    return calcSizeFrom( root );
}

int BinaryTree::calcSizeFrom( Node *cur )
{









}
```


Measuring Height

- Assume that height of empty tree is -1
- Same plan as counting size:

```
int BinaryTree::calcHeight()
{
    return calcHeightFrom( root );
}

int BinaryTree::calcHeightFrom( Node *cur )
{









}
```


Destroying A Tree

```
int BinaryTree::~BinaryTree()
{
    delTree( root );
}

void BinaryTree::delTree( Node *cur )
{
    if( cur != NULL ) {
        delTree( cur->left );
        delTree( cur->right );
        delete cur;
    }
}
```

Performance Analysis

- The operations are similar, so let's analyze them at the same time
  - How much work is done at each recursive call?
  - How many recursive calls are there?
- Bonus question: how would you write these without recursion?

Searching A Binary Tree

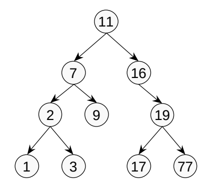
```
bool BinaryTree::isInTree( int item )
{
    return isInTreeFrom( root, item );
}

bool BinaryTree::isInTreeFrom( Node *cur, int item )
{
    if( cur == NULL ) {
        return false;
    } else if( cur->data == item ) {
        return true;
    }
        return isInTreeFrom( cur->left, item ) ||
               isInTreeFrom( cur->right, item );
    }
}
```

- What's the running time of this algorithm?
  - Can we do better?

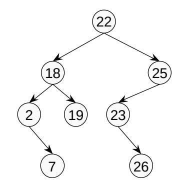
Binary Search Trees

- A very common use of binary trees
- A restriction that makes dictionary-style operations fast
- Definition:
  - No duplicates: every node contains a unique value
  - The type of the data values can be compared using <, == and >
  - For any node in the tree
    - Items in left subtree are < node
    - Items in right subtree are > node

Examples



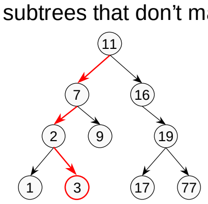
a binary search tree



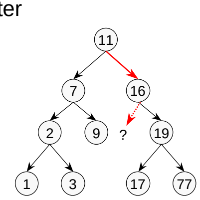
NOT a binary search tree
(why not?)

Searching A BST

- Like binary search on an array: throw away subtrees that don't matter



searching for 3



searching for 13

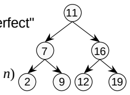
A Fast Search For BSTs

```
bool BinarySearchTree::isInTree( int item )
{
    return isInTreeFrom( root, item );
}

bool BinarySearchTree::isInTreeFrom( Node *cur, int item )
{
    if( cur == NULL ) {
        return false;
    } else if( item == cur->data ) {
        return true;
    } else if( item < cur->data ) {
        return isInTreeFrom( cur->left, item );
    } else {
        return isInTreeFrom( cur->right, item );
    }
}
```

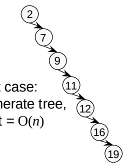
Performance Analysis Of BST Searching

- How much work is done in each recursive call?
- How many recursive calls are there?
  - Worst case: you have to walk all the way down the tree to the furthest leaf
  - Number of calls is height of tree
  - What's the height of a BST?

Best case: "perfect" tree, highly balanced, height = O(log n)



Worst case: degenerate tree, height = O(n)



BST Insertion

- A very elegant recursive algorithm
  - If tree is empty, return new one-node tree with item as root
  - If item equals root, stop
  - If item belongs in left subtree, recursively insert there
  - If item belongs in right subtree, recursively insert there

BST Insertion Algorithm

```
void BinarySearchTree::insert( int item )
{
    root = insertFrom( root, item );
}

Node *BinarySearchTree::insertFrom( Node *cur, int item )
{
    if( cur == NULL ) {
        Node *nn = new Node;
        nn->data = item;
        nn->left = NULL;
        nn->right = NULL;
        return nn;
    } else if( item < cur->data ) {
        cur->left = insertFrom( cur->left, item );
    } else if( item > cur->data ) {
        cur->right = insertFrom( cur->right, item );
    }
    return cur;
}
```


Analysis Of BST Insertion

- Just like searching: worst case number of recursive calls is height of tree
  - Analysis over random insertions gives probabilistic runtime of O(log n)
- Structure of resulting tree depends heavily on order of insertions!
  - More intelligent binary search tree implementations can fix this problem by rebalancing the tree (red-black trees, AVL trees, etc.)

BST Deletion

- Deleting a node from a BST can be complicated
  - Easy case: node is leaf (has no children), just remove it
  - Harder case: node has one child, must attach that child to node's parent
  - Hardest case: node has two children

Updated BST Class

```
struct Node {
    int data;

    Node *left;
    Node *right;
};

class BinarySearchTree
{
public:
    BinarySearchTree();        // construct an empty tree
    BinarySearchTree( const BinarySearchTree& other );
    ~BinaryTree();             // delete the tree

    int getSize();

    void insert( int item );   // Add to the tree
    void remove( int item );   // Remove from the tree
    bool isInTree( int item ); // Find item in tree

    // Other public methods …
private:
    // Other private data and helper functions …
    Node *root;
};
```

BST As Set Implementation

- The **`BinarySearchTree`** class just given is a very effective **`IntSet`** implementation:

```
class IntSet
{
public:
    IntSet() : bst() {}
    IntSet( const IntSet& other ) : bst( other.bst ) {}

    void addMember( int item ) { bst.insert( item ); }
    void removeMember( int item ) { bst.remove( item ); }
    bool isMember( int item ) { return bst.isInTree( item ); }

    IntSet union( const IntSet& other );
    IntSet intersection( const IntSet& other );
    IntSet difference( const IntSet& other );

private:
    BinarySearchTree bst;
};
```


BST As Table Implementation

- BSTs can also be used to implement the table ADT by changing the item type:

```
struct Node {
    Key key;
    Data data;

    Node *left;
    Node *right;
};
```

- **`Key`** and **`Data`** can be two arbitrary types (as long as **`Key`**s can be compared using <, ==, >)
- All BST operations compare just the **`Key`**s, not the **`Data`**s

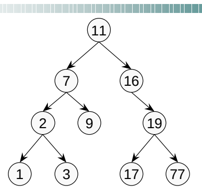
Binary Tree Traversal

- Question: how to implement **`IntSet::union`**?
- A binary tree is a collection
- We need a way to iterate across its contents
  - Like **`start`**, **`isEnd`**, **`advance`** for lists
- How will we do this?
  - What's the right order?
- Not just for binary search trees; works on binary trees in general

Recursive Traversal

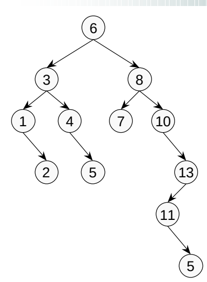
- Easiest way to visit every node in a tree is recursive (like **`calcSize`**, **`calcHeight`**):
  - At each node
    - Visit left subtree
    - Visit right subtree
    - Do something to item at current node
  - Order for these three steps leads to different traversal algorithms; We'll look at the three big ones
    - Preorder: item, left subtree, right subtree
    - Inorder: left subtree, item, right subtree
    - Postorder: left subtree, right subtree, item

Traversal Example 1



- Preorder traversal:
- Inorder traversal:
- Postorder traversal:

Traversal Example 2



- Preorder:
- Inorder:
- Postorder:

Traversal Implementation: Printing The Tree

```
void printPreorderFrom( Node *cur )
{
    if( cur != NULL ) {
        cout << cur->data << " ";
        printPreorderFrom( cur->left );
        printPreorderFrom( cur->right );
    }
}
```

```
void printPostorderFrom( Node *cur )
{
    if( cur != NULL ) {
        printPostorderFrom( cur->left );
        printPostorderFrom( cur->right );
        cout << cur->data << " ";
    }
}
```

```
void printInorderFrom( Node *cur )
{
    if( cur != NULL ) {
        printInorderFrom( cur->left );
        cout << cur->data << " ";
        printInorderFrom( cur->right );
    }
}

void BinarySearchTree::printInorder()
{
    printInorderFrom( root );
    cout << endl;
}
```


Abstract Traversal

- Sometimes it's not data that needs to be abstracted, it's an algorithm
- Example: these two functions are identical except for what to do at every node
- How can we abstract the idea of postorder traversal?

```
void printPostorderFrom( Node *cur )
{
    if( cur != NULL ) {
        printPostorderFrom( cur->left );
        printPostorderFrom( cur->right );
        cout << cur->data << " ";
    }
}

void deleteFrom( Node *cur )
{
    if( cur != NULL ) {
        deleteFrom( cur->left );
        deleteFrom( cur->right );
        delete cur;
    }
}
```


Abstract Traversal Idea 1

- Yes, this version is abstract, but you can only use it once per program
- Copying and pasting might help, but that defeats the goal of writing as little code as possible!

```
void traversePostorder( Node *cur )
{
    if( cur != null ) {
        traversePostorder( cur->left );
        traversePostorder( cur->right );
        doSomethingAtThisNode( cur );
    }
}
```


Abstract Traversal Idea 2

- This is the "correct" solution in the C world: pointers to functions!
- This can get very ugly very fast
- Hard to debug, hard to understand

```
void traversePostorder( Node *cur,
    void (*visit)( Node *node ) )
{
    if( cur != null ) {
        traversePostorder( cur->left );
        traversePostorder( cur->right );
        visit( cur );
    }
}

void printNode( Node *node )
{
    cout << node->data << " ";
}

void printPostorderFrom( Node *root )
{
    traversePostorder( root, printNode );
}
```


Abstract Abstract Traversal

- In C++, we can get the behaviour of "pointers-to-functions" using virtual functions
- A little more code at first, but easier to write more complicated algorithms, scales better, less ugly

```
class TraversalFunction {
public:
    virtual void visit( Node *node ) = 0;
};

void traversePostorder( Node *cur,
    TraversalFunction &func )
{
    if( cur != null ) {
        traversePostorder( cur->left );
        traversePostorder( cur->right );
        func.visit( cur );
    }
}

class PrintFunction : public TraversalFunction
{
public:
     virtual void visit( Node *node );
};

void PrintFunction::visit( Node * node ) {
    cout << node->data << " ";
}
```


BST With Abstract Traversal

```
struct Node { … };
class TraversalFunction { public: virtual void visit( Node *cur ) = 0; };

class BinarySearchTree
{
public:
    BinarySearchTree();         // construct an empty tree
    BinarySearchTree( const BinarySearchTree& other );
    ~BinaryTree();              // delete the tree

    int getSize();

    void insert( int item );   // Add to the tree
    void remove( int item );   // Remove from the tree
    bool isInTree( int item ); // Find item in tree

    void preorderTraversal( TraversalFunction& func );
    void inorderTraversal( TraversalFunction& func );
    void postorderTraversal( TraversalFunction& func );
private:
    // Other private data and helper functions …
    Node *root;
};
```

`IntSet` using Traversal

```
class UnionMaker : public TraversalFunction
{
public:
    UnionMaker( IntSet& s ) : addTo( s ) {}
    virtual void visit( Node *cur );
private:
    IntSet& addTo;
};

void UnionMaker::visit( Node *cur )
{
    addTo.insert( cur->data );
}

IntSet IntSet::union( const IntSet& other )
{
    IntSet result = other;
    UnionMaker maker( result );
    bst.inorderTraversal( maker );
}
```


Summary (I)

- Tree as new hierarchical data structure
  - Recursive definition and recursive data structure
- Tree parts and terminology
  - Made up of nodes
  - Root node, leaf nodes
  - Children, parents, ancestors, descendants
  - Depth of node, height of tree

Summary (II)

- Binary Trees
  - Either 0, 1, or 2 children at any node
  - Recursive functions to manipulate them
- Binary Search Trees
  - Binary Trees with ordering invariant
  - Recursive BST search
  - Recursive Insert, Delete functions

Summary (III)

- Binary Tree Traversals
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal
- Abstract classes as the basis for abstract traversal algorithms