Trees
Outline

• Preliminaries
  – What is Tree?
  – Implementation of Trees using C++
  – Tree traversals and applications

• Binary Trees

• Binary Search Trees
  – Structure and operations
  – Analysis
What is a Tree?

• A tree is a collection of nodes with the following properties:
  – The collection can be empty.
  – Otherwise, a tree consists of a distinguished node \( r \), called root, and zero or more nonempty sub-trees \( T_1, T_2, \ldots, T_k \), each of whose roots are connected by a directed edge from \( r \).

• The root of each sub-tree is said to be child of \( r \), and \( r \) is the parent of each sub-tree root.

• If a tree is a collection of \( N \) nodes, then it has \( N-1 \) edges.
Node A has 6 children: B, C, D, E, F, G.

B, C, H, I, P, Q, K, L, M, N are leaves in the tree above.

K, L, M are siblings since F is parent of all of them.
Preliminaries (continued)

• A *path* from node $n_1$ to $n_k$ is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that $n_i$ is parent of $n_{i+1}$ ($1 \leq i < k$)
  – The *length* of a path is the number of edges on that path.
  – There is a path of length zero from every node to itself.
  – There is exactly one path from the root to each node.

• The *depth* of node $n_i$ is the length of the path from *root* to node $n_i$.

• The *height* of node $n_i$ is the length of longest path from node $n_i$ to a *leaf*.

• If there is a path from $n_1$ to $n_2$, then $n_1$ is *ancestor* of $n_2$, and $n_2$ is *descendent* of $n_1$.
  – If $n_1 \neq n_2$ then $n_1$ is *proper ancestor* of $n_2$, and $n_2$ is *proper descendent* of $n_1$. 
Figure 1
A tree, with height and depth information
Implementation of Trees

```c
struct TreeNode {
    Object   element;
    struct TreeNode *firstChild;
    struct TreeNode *nextSibling;
};
```
Figure 2: The Unix directory with file sizes
Listing a directory

// Algorithm (not a complete C code)
listAll ( struct TreeNode *t, int depth)
{
    printName ( t, depth );
    if (isDirectory())
        for each file c in this directory (for each child)
            listAll(c, depth+1 );
}

- printName() function prints the name of the object after “depth” number of tabs-indentation. In this way, the output is nicely formatted on the screen.
- The order of visiting the nodes in a tree is important while traversing a tree.
  - Here, the nodes are visited according to preorder traversal strategy.
Figure 3: The directory listing for the tree shown in Figure 2

```
mark
  books
    dsaa
      ch1
      ch2
    ecp
      ch1
      ch2
    ipps
      ch1
      ch2
  courses
    cop3223
      syl
    cop3530
      syl
  .login
```
Size of a directory

```cpp
int FileSystem::size () const
{
    int totalSize = sizeOfThisFile();

    if (isDirectory())
        for each file c in this directory (for each child)
            totalSize += c.size();
    return totalSize;
}
```

- The nodes are visited using *postorder* strategy.
- The work at a node is done after processing each child of that node.
Figure 18.9
A trace of the size method

```
ch1       9
ch2       7
dsaa      17
    ch1    4
    ch2    6
ecp       11
    ch1    3
    ch2    8
ipps      12
  books   41
    syl    2
cop3223   3
    syl    3
cop3530   4
courses   8
    .login  2
mark      52
```
Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm $preOrder(v)$

$visit(v)$

for each child $w$ of $v$

$preorder(w)$
In a postorder traversal, a node is visited after its descendants.

Application: compute space used by files in a directory and its subdirectories.

Algorithm postOrder(v)
for each child w of v
    postOrder (w)
visit(v)
Binary Trees

- A binary tree is a tree in which no node can have more than two children.
- The depth can be as large as $N-1$ in the worst case.

A binary tree consisting of a root and two subtrees $T_L$ and $T_R$, both of which could possibly be empty.
Binary Tree Terminology

*Left Child* – The left child of node n is a node directly below and to the left of node n in a binary tree.

*Right Child* – The right child of node n is a node directly below and to the right of node n in a binary tree.

*Left Subtree* – In a binary tree, the left subtree of node n is the left child (if any) of node n plus its descendants.

*Right Subtree* – In a binary tree, the right subtree of node n is the right child (if any) of node n plus its descendants.
Binary Tree -- Example

- A is the root.
- B is the left child of A, and C is the right child of A.
- D doesn’t have a right child.
- H doesn’t have a left child.
- B, F, G and I are leaves.
Binary Tree – Representing Algebraic Expressions

(a) \( a - b \)

(b) \( a - b / c \)

(c) \( (a - b) * c \)
The height of a binary tree $T$ can be defined recursively as:

- If $T$ is empty, its height is -1.
- If $T$ is non-empty tree, then since $T$ is of the form $r \ T_L \ T_R$

the height of $T$ is 1 greater than the height of its root’s taller subtree; i.e.

$$\text{height}(T) = 1 + \max\{\text{height}(T_L), \text{height}(T_R)\}$$
Height of Binary Tree (cont.)

Binary trees with the same nodes but different heights
Number of Binary trees with Same # of Nodes

n=0 ➞ empty tree

n=1 ➞  (1 tree)

n=2 ➞  (2 trees)

n=3 ➞  (5 trees)
Full Binary Tree

- In a *full binary tree* of height \( h \), all nodes that are at a level less than \( h \) have two children each.
- Each node in a full binary tree has left and right subtrees of the same height.
- Among binary trees of height \( h \), a full binary tree has as many leaves as possible, and they all are at level \( h \).
- A full binary has no missing nodes.
- Recursive definition of full binary tree:
  - If \( T \) is empty, \( T \) is a full binary tree of height -1.
  - If \( T \) is not empty and has height \( h > 0 \), \( T \) is a full binary tree if its root’s subtrees are both full binary trees of height \( h - 1 \).
Full Binary Tree – Example

A full binary tree of height 2
Complete Binary Tree

- A complete binary tree of height $h$ is a binary tree that is full down to level $h-1$, with level $h$ filled in from left to right.
- A binary tree $T$ of height $h$ is complete if
  1. All nodes at level $h-2$ and above have two children each, and
  2. When a node at level $h-1$ has children, all nodes to its left at the same level have two children each, and
  3. When a node at level $h-1$ has one child, it is a left child.

- A full binary tree is a complete binary tree.
Complete Binary Tree – Example
Balanced Binary Tree

- A binary tree is *height balanced* (or *balanced*), if the height of any node’s right subtree differs from the height of the node’s left subtree by no more than 1.
- A complete binary tree is a balanced tree.
- Other height balanced trees:
  - AVL trees
  - Red-Black trees
  - B-trees
  ....
A Pointer-Based Implementation of Binary Trees

struct BinaryNode {
    Object element;
    struct BinaryNode *left;
    struct BinaryNode *right;
};
Binary Tree Traversals

• **Preorder Traversal**
  – the node is visited before its left and right subtrees,

• **Postorder Traversal**
  – the node is visited after both subtrees.

• **Inorder Traversal**
  – the node is visited between the subtrees,
  – Visit left subtree, visit the node, and visit the right subtree.
Binary Tree Traversals

(a) Preorder: 60, 20, 10, 40, 30, 50, 70
(b) Inorder: 10, 20, 30, 40, 50, 60, 70
(c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)
void preorder(struct tree_node * p) {
    if (p != NULL) {
        printf("%d\n", p->data);
        preorder(p->left_child);
        preorder(p->right_child);
    }
}
void inorder(struct tree_node *p)
{
    if (p != NULL) {
        inorder(p->left_child);
        printf("%d\n", p->data);
        inorder(p->right_child);
    }
}

void postorder(struct tree_node *p)
{
    if (p != NULL) {
        postorder(p->left_child);
        postorder(p->right_child);
        printf("%d\n", p->data);
    }
}

Finding the maximum value in a binary tree

```c
int FindMax(struct tree_node *p) {
    int root_val, left, right, max;
    max = -1; // Assuming all values are positive integers

    if (p != NULL) {
        root_val = p->data;
        left = FindMax(p->left_child);
        right = FindMax(p->right_child);

        // Find the largest of the three values.
        if (left > right)
            max = left;
        else
            max = right;

        if (root_val > max)
            max = root_val;
    }
    return max;
}
```
Adding up all values in a Binary Tree

```c
int add(struct tree_node *p)
{
    if (p == NULL)
        return 0;
    else
        return (p->data + add(p->left_child) +
                add(p->right_child));
}
```
Exercises

1. Write a function that will count the leaves of a binary tree.
2. Write a function that will find the height of a binary tree.
3. Write a function that will interchange all left and right subtrees in a binary tree.
Binary Search Trees

• An important application of binary trees is their use in searching.

• *Binary search tree* is a binary tree in which every node X contains a data value that satisfies the following:
  a) all data values in its left subtree are smaller than the data value in X
  b) the data value in X is smaller than all the values in its right subtree.
  c) the left and right subtrees are also binary search trees.
Example

A binary search tree

Not a binary search tree, but a binary tree
Binary Search Trees – containing same data
Operations on BSTs

• Most of the operations on binary trees are $O(\log N)$.
  – This is the main motivation for using binary trees rather than using ordinary lists to store items.

• Most of the operations can be implemented using recursion.
  – we generally do not need to worry about running out of stack space, since the average depth of binary search trees is $O(\log N)$.
The BinaryNode class

template <class Comparable>
class BinaryNode {
    Comparable element; // this is the item stored in the node
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const Comparable & theElement, BinaryNode *lt,
                BinaryNode *rt ) : element( theElement ), left( lt ),
                right( rt ) { }
};
/**
 * Method to find an item in a subtree.
 * x is item to search for.
 * t is the node that roots the tree.
 * Return node containing the matched item.
 */

template <class Comparable>
BinaryNode<Comparable> *
find( const Comparable & x, BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    else if( x < t->element )
        return find( x, t->left );
    else if( t->element < x )
        return find( x, t->right );
    else
        return t;  // Match
findMin (recursive implementation)

/**
 * method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */

template <class Comparable>
BinaryNode<Comparable> *
findMin( BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
findMax (nonrecursive implementation)

/**
 * method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */

template <class Comparable>
BinaryNode<Comparable> *
findMax( BinaryNode<Comparable> *t ) const
{
    if( t != NULL )
        while( t->right != NULL )
            t = t->right;
    return t;
}
Insert operation

Algorithm for inserting X into tree T:
- Proceed down the tree as you would with a find operation.
- if X is found
  do nothing, (or “update” something)
else
  insert X at the last spot on the path traversed.
Example

- What about duplicates?
Insertion into a BST

/* method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 * Set the new root.
 */

template <class Comparable>
void insert( const Comparable & x,
             BinaryNode<Comparable> * & t ) const
{
    if( t == NULL )
        t = new BinaryNode<Comparable>( x, NULL, NULL );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ; // Duplicate; do nothing
}
Deletion operation

There are three cases to consider:

1. Deleting a leaf node
   - Replace the link to the deleted node by NULL.

2. Deleting a node with one child:
   - The node can be deleted after its parent adjusts a link to bypass the node.

3. Deleting a node with two children:
   - The deleted value must be replaced by an existing value that is either one of the following:
     - The largest value in the deleted node’s left subtree
     - The smallest value in the deleted node’s right subtree.
Deletion – Case 1: A Leaf Node

To remove the leaf containing the item, we have to set the pointer in its parent to NULL.

Delete 70 (A leaf node)
Deletion – Case2: A Node with only a left child

Delete 45 (A node with only a left child)

```
          50
         /   \
       40     60
      /     /   \
    30     45    70
       /     /  \
      42   50   

  ➔
```

```
          50
         /   \
       40     60
      /     /   \
    30     42    70
```
Deletion – Case 2: A Node with only a right child

Delete 60 (A node with only a right child)
Deletion – Case 3: A Node with two children

• Locate the inorder successor of the node.
• Copy the item in this node into the node which contains the item which will be deleted.
• Delete the node of the inorder successor.

Delete 40 (A node with two children)
Deletion – Case 3: A Node with two children
Deletion routine for BST

template <class Comparable>
void remove( const Comparable & x,

    BinaryNode<Comparable> * & t ) const
{
    if( t == NULL )
        return;   // Item not found; do nothing
    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != NULL && t->right != NULL ) {
        t->element = findMin( t->right )->element;
        remove( t->element, t->right );
    }
    else   {
        BinaryNode<Comparable> *oldNode = t;
        t = ( t->left != NULL ) ? t->left : t->right;
        delete oldNode;
    }
}
Analysis of BST Operations

• The cost of an operation is proportional to the depth of the last accessed node.

• The cost is logarithmic for a well-balanced tree, but it could be as bad as linear for a degenerate tree.

• In the best case we have logarithmic access cost, and in the worst case we have linear access cost.
Figure 19.19
(a) The balanced tree has a depth of $\log N$; (b) the unbalanced tree has a depth of $N - 1$. 
Maximum and Minimum Heights of a Binary Tree

• The efficiency of most of the binary tree (and BST) operations depends on the height of the tree.
• The maximum number of key comparisons for retrieval, deletion, and insertion operations for BSTs is the height of the tree.
• The maximum of height of a binary tree with n nodes is n-1.
• Each level of a minimum height tree, except the last level, must contain as many nodes as possible.
Maximum and Minimum Heights of a Binary Tree

A maximum-height binary tree with seven nodes

Some binary trees of height 2
## Counting the nodes in a full binary tree

A full binary tree is a binary tree in which every node has either 0 or 2 children. The number of nodes at each level of a full binary tree can be calculated using the formula $2^n$, where $n$ is the level number.

<table>
<thead>
<tr>
<th>Level</th>
<th>Number of nodes at this level</th>
<th>Number of nodes at this and previous levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1 = 2^0$</td>
<td>$1 = 2^1 - 1$</td>
</tr>
<tr>
<td>2</td>
<td>$2 = 2^1$</td>
<td>$3 = 2^2 - 1$</td>
</tr>
<tr>
<td>3</td>
<td>$4 = 2^2$</td>
<td>$7 = 2^3 - 1$</td>
</tr>
<tr>
<td>4</td>
<td>$8 = 2^3$</td>
<td>$15 = 2^4 - 1$</td>
</tr>
<tr>
<td>$h$</td>
<td>$2^{h-1}$</td>
<td>$2^h - 1$</td>
</tr>
</tbody>
</table>
Some Height Theorems

Theorem 10-2: A full binary of height $h \geq 0$ has $2^{h+1} - 1$ nodes.

Theorem 10-3: The maximum number of nodes that a binary tree of height $h$ can have is $2^{h+1} - 1$.

$\rightarrow$ We cannot insert a new node into a full binary tree without increasing its height.
Some Height Theorems

**Theorem 10-4:** The minimum height of a binary tree with \( n \) nodes is \( \lceil \log_2(n+1) \rceil \).

**Proof:** Let \( h \) be the smallest integer such that \( n \leq 2^{h+1} - 1 \). We can establish the following facts:

*Fact 1* – A binary tree whose height is \( \leq h-1 \) has \( \leq n \) nodes.
  - Otherwise \( h \) cannot be the smallest integer in our assumption.

*Fact 2* – There exists a complete binary tree of height \( h \) that has exactly \( n \) nodes.
  - A full binary tree of height \( h-1 \) has \( 2^{h-1} \) nodes.
  - Since a binary tree of height \( h \) cannot have more than \( 2^{h+1} - 1 \) nodes.
  - At level \( h \), we will reach \( n \) nodes.

*Fact 3* – The minimum height of a binary tree with \( n \) nodes is the smallest integer \( h \) such that \( n \leq 2^{h+1} - 1 \).

So, \( \Rightarrow 2^{h-1} < n \leq 2^{h+1} - 1 \)
\( \Rightarrow 2^h < n+1 \leq 2^{h+1} \)
\( \Rightarrow h < \log_2(n+1) \leq h+1 \)

Thus, \( \Rightarrow h = \lceil \log_2(n+1) \rceil \) is the minimum height of a binary tree with \( n \) nodes.
Minimum Height

- Complete trees and full trees have minimum height.
- The height of an n-node binary search tree ranges from $\lceil \log_2(n+1) \rceil$ to n-1.
- Insertion in search-key order produces a maximum-height binary search tree.
- Insertion in random order produces a near-minimum-height binary tree.
- That is, the height of an n-node binary search tree
  - **Best Case** – $\lceil \log_2(n+1) \rceil$ ➔ $O(\log_2 n)$
  - **Worst Case** – n-1 ➔ $O(n)$
  - **Average Case** – close to $\lceil \log_2(n+1) \rceil$ ➔ $O(\log_2 n)$
    - In fact, $1.39\log_2 n$
Average Height

Suppose we’re inserting n items into an empty binary search tree to create a binary search tree with n nodes,

- How many different binary search trees with n nodes, and
- What are their probabilities,

There are n! different orderings of n keys.
But how many different binary search trees with n nodes?

n=0 ➔ 1 BST (empty tree)
n=1 ➔ 1 BST (a binary tree with a single node)
n=2 ➔ 2 BSTs
n=3 ➔ 5 BSTs
Average Height (cont.)

\( n=3 \) ➔

Probabilities: \( \frac{1}{6} \) \( \frac{1}{6} \) \( \frac{2}{6} \) \( \frac{1}{6} \) \( \frac{1}{6} \)

Insertion Order: \( 3,2,1 \) \( 3,1,2 \) \( 2,1,3 \) \( 1,3,2 \) \( 1,2,3 \) \( 2,3,1 \)
## Order of Operations on BSTs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieval</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insertion</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Deletion</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Traversal</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Treesort

• We can use a binary search tree to sort an array.

treesort(inout anArray:ArrayType, in n:integer)
// Sorts n integers in an array anArray
// into ascending order
    Insert anArray’s elements into a binary search
tree bTree

    Traverse bTree in inorder. As you visit bTree’s
nodes,
copy their data items into successive locations of
anArray
Treesort Analysis

- Inserting an item into a binary search tree:
  - Worst Case: \( O(n) \)
  - Average Case: \( O(\log_2 n) \)

- Inserting \( n \) items into a binary search tree:
  - Worst Case: \( O(n^2) \) \( \Rightarrow \) \( (1+2+\ldots+n) = O(n^2) \)
  - Average Case: \( O(n*\log_2 n) \)

- Inorder traversal and copy items back into array \( \Rightarrow O(n) \)

- Thus, treesort is
  \( \Rightarrow O(n^2) \) in worst case, and
  \( \Rightarrow O(n*\log_2 n) \) in average case.

- Treesort makes exactly the same comparisons of keys as quicksort when the pivot for each sublist is chosen to be the first key.
Saving a BST into a file, and restoring it to its original shape

• Save:
  – Use a preorder traversal to save the nodes of the BST into a file.

• Restore:
  – Start with an empty BST.
  – Read the nodes from the file one by one, and insert them into the BST.
Saving a BST into a file, and restoring it to its original shape

Preorder: 60 20 10 40 30 50 70

```java
bst.searchTreeInsert(60);
bst.searchTreeInsert(20);
bst.searchTreeInsert(10);
bst.searchTreeInsert(40);
bst.searchTreeInsert(30);
bst.searchTreeInsert(50);
bst.searchTreeInsert(70);
```
Saving a BST into a file, and restoring it to a minimum-height BST

• Save:
  – Use an inorder traversal to save the nodes of the BST into a file. The saved nodes will be in ascending order.
  – Save the number of nodes (n) in somewhere.

• Restore:
  – Read the number of nodes (n).
  – Start with an empty BST.
  – Read the nodes from the file one by one to create a minimum-height binary search tree.
Building a minimum-height BST

readTree(out treePtr:TreeNodePtr, in n:integer)
// Builds a minimum-height binary search tree from n sorted
// values in a file. treePtr will point to the tree’s root.

if (n>0) {
    // construct the left subtree
    treePtr = pointer to new node with NULL child pointers
    readTree(treePtr->leftChildPtr, n/2)

    // get the root
    Read item from file into treePtr->item

    // construct the right subtree
    readTree(treePtr->rightChildPtr, (n-1)/2)
}
A full tree saved in a file by using inorder traversal