Chapter 10

Trees
Outline

■ Terminology

■ ADT Binary Tree
  • “Traversal” algorithm - print data, search data, etc.
  • Pointer-based implementation

■ ADT Binary Search Tree (BST)
  • Algorithm for BST operations
  • Pointer-based implementation
  • Efficiency
Binary Tree

Position-oriented structures
  • e.g., List, Stack, Queue

Value-oriented structures
  • e.g., Binary search tree

Data Management Operations
  • insert
  • delete
  • search
Trees

- A very important data structure for organizing data.

- As name says, data is organized in hierarchical fashion in which data items are related by the “branches”
Trees or hierarchy as a way of organizing data

Figure 10-3
(a) An organization chart; (b) a family tree
Definition: A tree is a finite set of one or more nodes such that:

- There is a specially designated code called the root.
- The remaining nodes are partitioned into disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree. We call $T_1, \ldots, T_n$ the subtrees of the root. (recursive definition)
A Sample General Tree


Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
Trees Terminologies

- **The “degree” of a node:**
  - The number of subtrees of the node

- **The “degree” of a tree:**
  - The maximum degree of the nodes in the tree
A “leaf” node (or a terminal node):
- A node with degree zero

“parent” and “child”
- A node that has subtrees is the “parent” of the roots of the subtrees, and the roots of the subtrees are the “children”.

Trees Terminologies
Trees Terminologies (cont’d)

“sibling”

- Children of the same parent are “siblings”.

(no order among children)
“ancestors”
- The “ancestors” of a node are all the nodes along the path from the root to the node.

“descendants”
- The “descendants” of a node are all the nodes that are in its subtrees.
The “level” of a node

- Let the root be at level 1.
- For all subsequent nodes, level is the level of the node’s parent plus one.
The “height” or “depth” of a tree

- The maximum level of any node in the tree.
A very important data structure in computer science

**Definition:** A **binary tree** is a finite set of nodes that is either empty or consists of a **root** and two disjoint binary trees called the **left subtree** and the **right subtree**.

- The degree of any node must not exceed two.
- A binary tree may have zero nodes (different from a tree).
- Children of a node has order (while it is not in trees).
Binary Tree

Recursive definition

- T is empty, or
- T is partitioned into three disjoint subsets:
  - A single node r, the root
  - Two possibly empty sets that are binary trees called left and right subtrees of r.

T is a binary tree if either

- T has no node, or
- T is of the form

```
    r
   /|
  /  \
TL   TR
```
Binary Trees that represent "algebraic expressions"

Figure 10-4
Binary trees that represent algebraic expressions

operator
operand
precedence (evaluation) order
Lemma 1 - Maximum number of nodes

- 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 depth $k$ is $2^k - 1, k \geq 1$

Proof by induction.
Binary Trees: properties (cont’d)

Lemma 2 - Relation between number of leaf nodes and nodes of degree 2

- For any nonempty binary tree, $T$, if $n_0$ is the number of leaf nodes and $n_2$ the number of nodes of degree 2, then $n_0 = n_2 + 1$

- $n = n_0 + n_1 + n_2$ (n: total number of nodes)
- $n = B + 1$ (Every node except the root has a branch leading to it. $B$: Number of branches)
- $B = n_1 + 2n_2$ --> $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$ --> $n_0 = n_2 + 1$
Binary Trees: properties (cont’d)

Lemma 3 Relationship between numbers of nodes in a sequentially numbered binary tree

- if a complete binary tree with \( n \) nodes (depth = \( \left\lceil \log_2 n + 1 \right\rceil \)) is represented sequentially, then for any node with index \( i \), \( 1 \leq i \leq n \), we have:
  1. parent \((i)\) is at \( \left\lfloor i/2 \right\rfloor \) if \( i \neq 1 \). If \( i = 1 \), \( i \) is at the root and has no parent.
  2. Left-child\((i)\) is at \( 2i \) if \( 2i \leq n \). If \( 2i > n \), then \( i \) has no left child.
  3. right-child\((i)\) is at \( 2i + 1 \) if \( 2i + 1 \leq n \). If \( 2i + 1 > n \), then \( i \) has no left child.

- The Proof is omitted.
“Full” Binary Tree

Recursive definition:

- If $T$ is empty, $T$ is a full binary tree of height 0.

- 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

Definition: A full binary tree of depth \( k \) is a binary tree of depth \( k \) having \( 2^k - 1 \) nodes, \( k \geq 0 \).

Sequential numbering nodes of a full binary tree
- Start with root on level 1
- Continue with the nodes on level 2, and so on
- At any level, number nodes from left to right.
Full Binary Tree (cont’d)

- Theorem 10-2: A full binary tree of height $h \ (h \geq 0)$ has $2^h - 1$ nodes.

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

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

- The maximum height of a binary tree with $N$ nodes is $N$. 
“Complete” Binary Tree

Definition: A binary tree with \( N \) nodes and depth \( k \) is "complete" iff its nodes correspond to the nodes numbered from 1 to \( n \) in the full binary tree of depth \( k \).

Height

\[
\left\lfloor \log_2 (N + 1) \right\rfloor
\]
General Trees - transform into a binary tree
A Sample Tree

A Tree of degree 2, i.e. a Binary Tree
ADT Binary Tree Operations

// Creates an empty binary tree.
createBinaryTree ()

// Creates a one-node binary tree whose root contains rootItem.
createBinaryTree (in rootItem:TreeItemType)

// Creates a binary tree whose root contains rootItem and has
// leftTree and rightTree, respectively, as its left and right
// subtrees.
createBinaryTree(in rootItem, inout leftTree, inout rightTree)
ADT Binary Tree Operations (cont’d)

DestroyBinaryTree ()
// Destroys a binary tree.

isEmpty(): boolean {query}
// Determines whether a binary tree is empty.

getRootData(): TreeItemType throw TreeException
// Returns the data item in the root of a nonempty binary tree. If empty, throw an exception.

setRootData (in newItem: TreeItemType) throw TreeException
// Replaces the data item in the root of a binary tree with newItem, if the tree is not empty.
// If the tree is empty, creates a root node whose data item is newItem and inserts the new node into the tree.
// Otherwise, throw an exception.
attachLeft(in newItem: TreeItemType)
    throw TreeException
// Attaches a left child containing newItem to
the root of a binary tree.
// Throws TreeException if a new child node
cannot be allocated.
// Also throws TreeException if the binary tree
is empty
// (no root node to attach to) or
// a left subtree already exists (should
explicitly detach it first).

attachRight(in newItem: TreeItemType)
    throw TreeException
// Attaches a right child containing newItem to
the root of a binary tree.
ADT Binary Tree Operations (cont’d)

attachLeftSubtree (inout leftTree:BinaryTree)
throw TreeException

// Attaches leftTree as the left subtree of the root of a binary tree and make leftTree empty so that it cannot be used to access this tree.

// Throws TreeException if the binary tree is empty
// (no root node to attach to) or
// a left subtree already exists (should explicitly detach it first).

attachRightSubtree (inout rightTree:BinaryTree) throw TreeException

// Attaches rightTree as the right subtree of the root of a binary tree
ADT Binary Tree Operations (cont'd)

detachLeftSubtree (out leftTree: BinaryTree) throw TreeException
// Detaches the left subtree of a binary tree's root
// and retains it in LeftTree.
// Throw TreeException if the binary tree is empty (no
// root node to detach from).

detachRightSubtree (out rightTree: BinaryTree) throw TreeException
// Detaches the right subtree of a binary tree's root
// and retains it in rightTree.
// Throw TreeException if the binary tree is empty (no
// root node to detach from).
ADT Binary Tree Operations (cont'd)

leftSubtree():BinaryTree
// Returns a copy of the left subtree of a binary tree's
// root without detaching the subtree.

rightSubtree():BinaryTree
// Returns a copy of the right subtree of a binary
// tree's
// root without detaching the subtree.
// Returns an empty tree if the binary tree is empty
// (no root node to copy from).
Binary Tree Traversals

- A commonly used operation on trees
- “Traversing” a tree:
  
  Visit each node in the tree exactly once.

  "visit a node" means to “retrieve” or access the node's data item.

- A full traversal produces a “linear” order for the information in a tree
- Possible traversals
  - 6 ways (LVR, LRV, VLR, RVL, RLV, VRL)
Three Ways of Binary Tree Traversals

- "Inorder": LVR
- "Postorder": LRV
- "Preorder": VLR
ADT Binary Tree Operations

preorderTraverse(in visit:FunctionType) VLR
// Traverses a binary tree in preorder and calls the
// function visit() once for each node.

inorderTraverse(in visit:FunctionType) LVR
// Traverses a binary tree in inorder and calls the
// function visit() once for each node.

postorderTraverse(in visit:FunctionType) LRV
// Traverses a binary tree in postorder and calls the
// function visit() once for each node.

"visit a node" means to display the node's data item.
Preorder Traversal

Output: 60 20 10 40 30 50 70

(a) Preorder: 60, 20, 10, 40, 30, 50, 70
Preorder Traversal: recursive function

preorder(T)
  if (T is not empty)
  {
    Display the data in the root of T
    preorder(left subtree of T's root)
    preorder(right subtree of T's root)
  }
Inorder Traversal

Output:

(b) Inorder: 10, 20, 30, 40, 50, 60, 70

Output a sorted list!
Inorder Traversal: recursive function

inorder(T)

if (T is not empty)
{
    inorder(Left subtree of T's root)
    Display the data in the root of T
    inorder(Right subtree of T's root)
}

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
Postorder Traversal

Output:

10 30 50 40 20 70 60

Depth-first traversal.

(c) Postorder: 10, 30, 50, 40, 20, 70, 60
Postorder Traversal: recursive function

\[ \text{postorder}(T) \]

if (T is not empty)
{
  \text{postorder}(\text{Left subtree of } T'\text{'s root})
  \text{postorder}(\text{Right subtree of } T'\text{'s root})
  \text{Display the data in the root of } T
}
Postorder traversal of expressions

(a) \( a - b \)

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

(c) \( (a - b) * c \)

Postfix form of the expression:
- ab-
- abc/-
- ab-c*
Binary Tree Traversal

- preorder, inorder and postorder
- $O(\_\_\_\_)$

$O(N)$

$N$ is the number of nodes of the tree
Representations of a Binary Tree

- Array-based implementation
- Pointers – a typical implementation
Array-based Implementation

- Use a C++ class `TreeNode` to define a node in the tree.
- Use an array of tree nodes to represent the binary tree.
- Declare `BinaryTree` as a friend class to class `TreeNode` so that members of `BinaryTree` have direct access to all of the members of `TreeNode`.
- Note that the `TreeNode` class is used only by the implementation of the `BinaryTree` class.
- It is inappropriate for other classes to have access to the members of the `TreeNode` class.
Definition of class TreeNode

const int MAX_NODES = 100; // maximum number of nodes
typedef string treeItemType;
class TreeNode // node in the tree
{
    private:
        TreeNode();
        TreeNode(const TreeItemType & nodeItem, int left, int right)
        TreeItemType item; // data portion
        int leftChild=-1; // index to left child
        int rightChild=-1; // index to right child

        // friend class – can access private parts
        friend class BinaryTree;
}; // end class TreeNode
Definition of a Binary Tree

TreeNode[MAX_NODES] tree;    // array of tree nodes

int root=-1;     // index of root

int free;        // index of free list
Array-based Implementation of a Binary Tree

(1/2)

**Figure 10-11**
Level-by-level numbering of a complete binary tree

A complete binary tree: Compact use of array space

<table>
<thead>
<tr>
<th>Item</th>
<th>LChild</th>
<th>RChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Jane</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Tom</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Alan</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Ellen</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>Nancy</td>
<td>-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Root</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>LChild</th>
<th>RChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>?</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>-1</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>-1</td>
</tr>
</tbody>
</table>

Free list

**Figure 10-10**
(b) its array-based implementation
Array-based Implementation of a Binary Tree (2/2)

A skewed binary tree: Inefficient use of array space

leftChild = 2i + 1
rightChild = 2i + 2
An alternate array-based implementation

- As the tree changes due to insertions and deletions, its nodes may not be in contiguous elements of the array.

- Use a free list to keep track of available nodes.
An alternate array-based implementation example

<table>
<thead>
<tr>
<th>item</th>
<th>leftChild</th>
<th>rightChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>Tom</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>zeth</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>?</td>
<td>-1</td>
<td>4</td>
</tr>
<tr>
<td>free</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

root: 0
free: 3
An alternate array-based implementation example (cont’d)

<table>
<thead>
<tr>
<th>item</th>
<th>leftChild</th>
<th>rightChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Tom</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>zeth</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>bob</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

The array-based implementation example continues with the following structure:

- **jane**: jane
- **bob**: bob
- **tom**: tom
- **zeth**: zeth

The corresponding array values are as follows:

- **root**: 0
- **free**: 4

The array is structured to accommodate the tree's nodes, with each index representing a node and its children as indicated by the leftChild and rightChild values.
Pointer-Based Binary Tree

- Use pointers to link the nodes in the tree.

![Diagram of a pointer-based binary tree with root, Item, LChildPtr, RChildPtr labels.](image-url)
typedef string treeItemType;
class TreeNode // node in the tree
{
    private:
        TreeNode();
        TreeNode(const TreeItemType & nodeItem,  
            TreeNode *left=NULL, TreeNode *right=NULL) 
            item(nodeItem); // data portion
            TreeNode * leftChildPtr; // pointer to left child
            TreeNode * rightChildPtr; // pointer to right child

    friend class BinaryTree;
}; // end class TreeNode
Pointer-Based Binary Tree: BinaryTree.h

// ********************************************
// Header file BinaryTree.h for the ADT binary tree.
// ********************************************

#include "TreeException.h"
#include "TreeNode.h"

typedef void (*FunctionType)(TreeItemType& anItem);
class BinaryTree
{
private:
    TreeNode *root; // pointer to root of tree
public:
    // constructors and destructor:
    BinaryTree();
    BinaryTree(const TreeItemType& rootItem);
    BinaryTree(const TreeItemType& rootItem,
                const BinaryTree & leftTree, const BinaryTree & rightTree)
    BinaryTree(const BinaryTree & tree);
    root
    leftTree
    rightTree
Constructors

- Construct a binary tree
  - an empty binary tree
    - `BinaryTree tree1;`
  - a binary tree with a root node only
    - `BinaryTree tree2(root2);`
    - `BinaryTree tree3(root3);`
  - a binary tree with a root and the root’s two subtrees
    - `BinaryTree tree4(root4, tree2, tree3);`

![Diagram of binary tree constructors]
Constructor Implementation

BinaryTree :: BinaryTree() : root(NULL)
{
}  // end default constructor

BinaryTree :: BinaryTree(const TreeItemType& rootItem);
{
    root = new TreeNode(rootItem, NULL, NULL);
    assert(root != NULL);
}
}  // end constructor

Note:
- Make an input argument const is to avoid expensive copying of the object.
- A constant reference argument instead of value arguments.
Constructor Implementation

```cpp
BinaryTree :: BinaryTree(const TreeItemType& rootItem,
        const BinaryTree & leftTree,
        const BinaryTree & rightTree) {
    root = new TreeNode(rootItem, NULL, NULL);
    assert(root != NULL);
    attachLeftSubtree(leftTree);
    attachRightSubtree(rightTree);
}  // end constructor
```
Constructor Implementation

BinaryTree :: BinaryTree(const BinaryTree & tree)
{
    copyTree(tree.root, root);
} // end copy constructor

BinaryTree :: BinaryTree(TreeNode *nodePtr):
    root(nodePtr)
{
} // end protected constructor
**copyTree - a recursive function**

```cpp
void BinaryTree :: copyTree(TreeNode *treePtr,
    TreeNode *& newTreePtr) const
{
    // preorder traversal; returns the address of a pointer pointing to the
    // root node of the new tree.
    if (treePtr != NULL)
    {
        // copy node
        newTreePtr = new TreeNode(treePtr->item, NULL, NULL);
        if (newTreePtr == NULL)
            throw TreeException("TreeException: cannot allocate memory");

        copyTree(treePtr->leftChildPtr, newTreePtr->leftChildPtr);
        copyTree(treePtr->rightChildPtr, newTreePtr->rightChildPtr);
    } else
        newTreePtr = NULL;  // copy empty tree
}
```

*const function*
copyTree - a recursive function

![Diagram of copyTree function]

- `newTreePtr`
- `Root Node`
- `left`
- `right`
virtual ~ BinaryTree( ); // destructor

// A virtual function of a class is a function that a derived class can redefine.
Destructor implementation

void BinaryTree::destroyTree(TreeNode * & treePtr)
{
    // postorder traversal
    if (treePtr != NULL)
    {
        destroyTree(treePtr->leftChildPtr);
        destroyTree(treePtr->rightChildPtr);
        delete treePtr;
        treePtr = NULL;
    }
    // end if
}  // end destroyTree

Destructor implementation

BinaryTree::~BinaryTree()
{
    destroyTree(root);
}  // end destructor

Root Node

```cpp
TreeNode * root
```

```
root
```

```
leftChildPtr rightChildPtr
```

```
Root Node
```

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
// binary tree operations:
virtual bool BinaryTreeIsEmpty() const;

virtual treeItemType RootData() Const;
virtual void setRootData(const treeItemType& newItem) throw TreeException;

virtual void attachLeft(const treeItemType& newItem) throw TreeException;

virtual void attachRight(const treeItemType& newItem) throw TreeException;
virtual void attachLeftSubtree(const binTreeClass& leftTree, bool& success);

virtual void attachRightSubtree(const binTreeClass& rightTree, bool& success);

virtual void detachLeftSubtree(binTreeClass& leftTree, bool& success);
virtual void detachRightSubtree(binTreeClass& rightTree, bool& success);

virtual binTreeClass LeftSubtree() const;
virtual binTreeClass RightSubtree() const;
Pointer-Based Binary Tree: BinaryTree.h (cont’d)

virtual void preorderTraverse(functionType visit);
virtual void inorderTraverse(functionType visit);
virtual void postorderTraverse(functionType visit);
Tree Traversal Implementation (1/2)

void BinaryTree::inorderTraverse(FunctionType visit)
{
    inorder(root, visit);
} // end inorderTraverse

- When implement a recursive traversal operation, need to be careful not to violate the wall of the ADT.
- void BinaryTree::inorder(TreeNode root, FunctionType visit)
  
  - The input argument the pointer root eventually points to every node in the tree.
  - Because this argument depends on the tree’s pointer-based implementation, it is not suitable as a public member, so we make inorder as a protected member.
void BinaryTree::preorderTraverse(FunctionType visit)
{
    preorder(root, visit);
} // end preorderTraverse

void BinaryTree::postorderTraverse(FunctionType visit)
{
    postorder(root, visit);
} // end postorderTraverse
void BinaryTree::inorder(TreeNode treePtr, FunctionType visit)
{
    if (treePtr != NULL)
    {
        inorder(treePtr->leftChildPtr, visit);
        visit(treePtr->Item);
        inorder(treePtr->rightChildPtr, visit);
    } // end if
} // end Inorder
Note: Function \texttt{visit} as a formal argument in inorder.

- The function \texttt{visit} has the type \texttt{FunctionType} defined as follows:

  \begin{verbatim}
  typedef void (*FunctionType) (TreeItemType& anItem);
  \end{verbatim}

- \texttt{visit} specifies the tree item as a \texttt{reference parameter} which allows the calling program to \texttt{view} the item as well as \texttt{modify} the item.
visit implementation

- void display(TreeItemType& anItem);

Example

tree4.inorderTraverse(display);

void BinaryTree::inorderTraverse(FunctionType visit)
{
    inorder(root, visit);
}
// end inorderTraverse
Display ()

template <class Type>
void NewClass <Type>::display()
{
    cout << theData;
} // end display

• The standard types such as int, char and string are assumed.
• Does not work for user-defined types (e.g., struct …)
Pointer-Based Binary Tree (1/5)

// overloaded operator:
virtual BinaryTree& operator=(const BinaryTree& rhs);

Implementation

BinaryTree& BinaryTree :: operator=(const BinaryTree& rhs)
{
    if (this != &rhs)
    {
        destroyTree(root); // deallocate left-hand side
        copyTree(rhs.root, root); // copy right-hand side
    } // end if
    return *this;
} // end operator=
ptrType BinaryTree::rootPtr() const
{
  return root;
}
// end rootPtr
void binTreeClass::setRootPtr(PtrType newRoot)
{
    root = newRoot;
} // end setRoot

void binTreeClass::GetChildPtrs
    (PtrType nodePtr, PtrType& leftPtr, PtrType& rightPtr) const
{
    leftPtr = nodePtr->lChildPtr;
    rightPtr = nodePtr->rChildPtr;
} // end getChildPtrs
void binTreeClass::setchildPtrs
        (PtrType nodePtr, PtrType leftPtr, PtrType rightPtr)
    {  nodePtr->lChildPtr = leftPtr;
        nodePtr->rChildPtr = rightPtr;
    }  // end setChildPtrs

void binTreeClass::preorder(PtrType treePtr, FunctionType visit)
    {  if (treePtr != NULL)
        {  visit(treePtr->item);
            preorder(treePtr->lChildPtr, visit);
            preorder(treePtr->rChildPtr, visit);
        }  // end if
    }  // end preorder
void binTreeClass::postorder
    (PtrType treePtr, FunctionType visit)
{
    if (treePtr != NULL)
    {
        postorder(treePtr->lChildPtr, visit);
        postorder(treePtr->rchildPtr, visit);
        visit(treePtr->item);
    } // end if
} // end postorder

// End of implementation file.
#include "BinaryTree.h"
#include <iostream>
using namespace std;

void display(TreeItemType& anItem);

int main(){
    BinaryTree tree1, tree2, left; //empty trees
    BinaryTree tree3(70); //tree with only a root node 70

    // build the tree in Figure 10-10
Example Use of BinaryTree class

(2/3)

tree1.setRootData(40);
tree1.attachLeft(30);
tree1.attachRight(50);

tree2.setRootData(20);
tree2.attachLeft(10);

Tree2.attachRightSubtree(tree1);

BinaryTree binTree(60, tree2, tree3);

(b) Inorder: 10, 20, 30, 40, 50, 60, 70
Example Use of BinaryTree class
(3/3)

binTree.inorderTraverse(display);
  // 10, 20, 30, 40, 50, 60, 70
binTree.getLeftSubtree().inorderTraverse(display);
  // 10, 20, 30, 40, 50
binTree.detachLeftSubtree(left);
left.inorderTraverse(display);
  // 10, 20, 30, 40, 50
binTree.inorderTraverse(display);
  // 60, 70
Return 0;
} // end main
Non-recursive Traversal

- Example: inorder traversal

- Use **stack** to keep track of **pointers** to the nodes along the path from the tree’s root to the current node \( n \)
  - the pointer to \( n \) at the top of the stack and
  - the pointer to the **root** at the bottom.
Trace the execution of recursive inorder traversal

```cpp
void BinaryTree::inorder(TreeNode treePtr, FunctionType visit)
{
    if (treePtr != NULL)
    {
        inorder(treePtr->leftChildPtr, visit);
        visit(treePtr->item);
        inorder(treePtr->rightChildPtr, visit);
    } // end if
} // end Inorder
```

Used to find nodes to go back

**Figure 10-14**

Contents of the implicit stack as `TreePtr` progresses through a given tree during a recursive inorder traversal.
Non-recursive Traversal

Figure 10-15
Traversing (a) the left and (b) the right subtrees of 20

(a) Left subtree of 20 has been traversed. Pop pointer to 10 from stack, visit 20.

(b) Right subtree of 20 has been traversed. Pop pointer to 40 from stack.
Nonrecursive inorderTraverse Implementation

traverse (in visit:FunctionType)
// Nonrecursively traverses a binary tree inorder.
   // initialize
Create an empty stack s
cur = rootPtr()    // start at root
done = false
while (!done)
{  if (cur != NULL)
{    // place pointer to node on stack before
    // traversing node's left subtree
    s.push(cur)
    // traverse the left subtree
    cur = cur->leftChildPtr
}
else // backtrack from the empty subtree and visit the node at the
    // top of the stack; however, if the stack is empty, you are done
{ if (!s.isEmpty())
    { s.getTop(cur)
        visit(cur->item)
        s.pop()
        // traverse the right subtree of the node just visited
        cur = cur->rightChildPtr
    }
    else
        done = true
} // end else backtrack
} // end while
Binary Search Tree
Figure 10-5
A binary search tree of names
A way of organizing data

- **Record** and **field**, e.g.,
  - A student ID database.
  - Each student’s data is a **record**.
  - **Fields**: name, ID number, address, telephone number, department, grade, and so on.

- **Insert/Delete/Search** – information retrieval.
ADT Binary Search Tree
Binary Search Tree

Definition

- Each node \( n \) in a binary search tree satisfies the following properties:
  - \( n \)’s search key is **greater** than all search keys in its **left** subtree \( T_L \).
  - \( n \)’s search key is **less** than all search keys in its **right** subtree \( T_R \).
  - Both \( T_L \) and \( T_R \) are binary search trees.

- A binary search tree allows users to **search** for a particular data item, given its **search key value** instead of its position.
Records and Fields in Binary Search Tree

- **TreeItemType** is the type of the items stored in the binary search tree.

- It should be based on **KeyItem** which has a search key field of type **KeyType**
ADT Binary Search Tree Operations

createSearchTree()
// Creates an empty binary search tree.

destroySearchTree()
// Destroys a binary search tree.

isEmpty()
// Determines whether a binary search tree is empty.

searchTreeInsert(in
ADT Binary Search Tree Operations (cont’d)

searchTreeDelete (in searchKey:KeyType) throw TreeException

// Deletes from a binary search tree the item whose search key equals searchKey. The operation fails if no such item exists and throw TreeException.

searchTreeRetrieve (in searchKey:KeyType, out treeItem:TreeItemType) throw TreeException

// Retrieves into treeItem the item in a binary search tree whose search key equals searchKey. The operation fails if no such item exists and throw TreeException.
ADT Binary Search Tree Operations (cont’d)

preorderTraverse (in visit:FunctionType)
// Traverses a binary search tree in preorder and calls the // function Visit once for each item.

inorderTraverse (in visit:FunctionType)
// Traverses a binary search tree in inorder and calls the // function Visit once for each item.

postorderTraverse (in visit:FunctionType)
// Traverses a binary search tree in postorder and calls the // function Visit once for each item.
Example

- `nameTree.searchTreeRetrieve("Nancy", nameRecord);`
- `nameTree.searchTreeInsert(HalRecord);`
- `nameTree.searchTreeDelete("Jane");`
- `nameTree.inorderTraverse(displayName);`
BST Search pseudo algorithm

search(in binTree:BinarySearchTree, in searchKey:KeyType)
// Searches the binary search tree binTree for the item
// whose search key is searchKey.

if (binTree is empty)
    The desired record is not found
else if (searchKey == search key of root's item)
    The desired record is found
else if (searchKey < search key of root's item)
    search(Left subtree of binTree, searchKey)
else
    Search (Right subtree of binTree, searchKey)
Different Shapes of Binary Search Tree with the same set of data

(a)

(b)

(c)
Different Shapes of Binary Search Tree with the same set of data

- The shape depends on the sequence of data insertion/deletion.
- The search algorithms work more efficiently on some trees than on others.
- O(N)
Definition of class KeyedItem

#include <string>
Using namespace std;

typedef string KeyType;
class KeyedItem
{
public:
    KeyedItem() {};
    KeyedItem(const KeyType & keyValue): searchKey(KeyValue) {}
    KeyType getKey() const
    {
        return searchKey;
    } // getKey
private:
    KeyType searchKey;
}; // end class
BST Insertion Algorithm

insertItem(inout treePtr: TreeNodePtr, in newItem: TreeItemType)

// Inserts newItem into the binary search tree to which treePtr points.

if (treePtr is NULL)
{
    Create a new node and let treePtr point to it
    Copy newItem into new node's data portion
    Set the pointers in the new node to NULL
}
else if (newItem.getKey() < treePtr->item.getKey())
    insertItem(treePtr->leftChildPtr, newItem)
else
    insertItem(treePtr->rightChildPtr, newItem)
Insert a record with searchKey = “Frank”
Figure 10-21
(c) insertion at a leaf
**BST Deletion Algorithm**

- The essential task here is: Remove item \(i\) from the tree.

- Assume that `deleteItem` locates item \(i\) in a particular node \(N\), there are three cases to consider:
  1. \(N\) is a leaf.
  2. \(N\) has only one child.
  3. \(N\) has two children.
BST Deletion Algorithm:
delete a leaf node

- Set the pointer in its parent to NULL.
BST Deletion Algorithm: delete a node with one child

- N has only a left child.
  -> let N’s parent points to N’s child

- N has only a right child.
  -> let N’s parent points to N’s child

*Figure 10-22*
(a) N with only a left child—N can be either the left or right child of P; 
(b) after deleting node N
BST Deletion Algorithm:
delete a node with two children

Figure 10-26
Not any node will do

The resulting tree
must be
a “binary search tree”.
**Figure 10-27**

Search key $x$ can be replaced by $y$
Delete a Node with Two Children

- \( \text{L}_{\text{max}} < N < \text{every node in rightSubtree} \)
  
  \( \text{-> replace } N \text{ with the rightmost node in the leftSubtree} \)

- Every node in leftSubtree
  
  \( \text{< } N \text{ < R}_{\text{min}} \)
  
  \( \text{-> replace } N \text{ with the leftmost node in the rightSubtree} \)
Figure 10-28

Copying the item whose search key is the inorder successor of N's search key.
BST Deletion Algorithm (1/4)

deleteItem (inout treePtr: TreeNodePtr, in searchKey: KeyType) throw TreeException

// Deletes from the binary search tree to which treePtr points the item whose search key equals searchKey.
// The operation fails if no such item exists; throw exception.

if (treePtr == NULL)
    throw TreeException // item not found
else if (searchKey == treePtr->item.getKey())
    // item is in the root of some subtree
    deleteNodeItem(treePtr) // delete the item
else if (searchKey < treePtr->itein.getKey())
    // search the left subtree
    deleteItem(treePtr->leftChildPtr, searchKey)

else // search the right subtree
    deleteItem(treePtr->rightChildPtr, searchKey)
deleteNodeItem(inout nodePtr:TreeNodePtr)

// Deletes the item in node N to which nodePtr points.
if (N is a leaf)
{
    // remove leaf from the tree
    delete nodePtr
    nodePtr = NULL
}
else if (N has only one child C)
{
    // C replaces N as the child of N's parent
    delPtr = nodePtr
    if ( C is the left child of N)
        nodePtr = nodePtr->leftChildPtr
    else
        nodePtr = nodePtr->rightChildPtr
    delete delPtr
}

Figure 10-22
(a) N with only a left child—N can be either the left or right child of P;
(b) after deleting node N
BST Deletion Algorithm (4/4)

else  // N has two children
{
    // find the inorder successor of the search key in N:
    // it is in the leftmost node of the subtree
    // rooted at N's right child
    processLeftmost(nodePtr->rightChildPtr, replacementItem)
    put replacementItem in node N
}  // end if
ProcessLeftmost()

processLeftmost(inout nodePtr: TreeNodePtr, out treeItem: TreeItemType)
// Retrieves into treeItem the item in the leftmost descendant of the node to which nodePtr points.
// Deletes the node that contains this item.

if (nodePtr->leftChildPtr == NULL)
{  // this is the node you want; it has no left child, but it might have a right subtree
    treeItem = nodePtr->item
    delPtr = nodePtr
ProcessLeftmost() (cont'd)

// the actual argument corresponding to nodePtr is a child pointer of nodePtr's parent; thus, the following "moves up" nodePtr's right subtree

nodePtr = nodePtr->rightChildPtr

delete delPtr

} else

processLeftmost(nodePtr->leftChildPtr, treetItem)
Recursive deletion of Node N

Any change to `treePtr` while deleting node N (Bob) changes `leftChildPtr` of Jane

*Figure 10-29*

Recursive deletion of node N
Binary Search Tree

- N's left subtree: Search keys are $< y$
- N's right subtree: Search keys are $\geq y$
BST Retrieval Algorithm

retrieveltem(in treePtr: TreeNodePtr,
in searchKey: KeyType,
out treeItem: TreeItemType)

// Retrieves into treeItem the item whose search key equals
// searchKey from the binary search tree to which treePtr points.
// The operation fails if no such item exists.

if (treePtr == NULL)
    throw TreeException // tree is empty
else if (searchKey == treePtr->item.getKey())
    // item is in the root of some subtree
    treeltem = treePtr->item
BST Retrieval Algorithm (cont'd)

else if (searchKey < treePtr->item.getKey())
    // search the left subtree
    retrieveItem(treePtr->leftChildPtr, searchKey, treeItem)

else // search the right subtree
    retrieveItem (treePtr->rightChildPtr, searchKey, treeItem)
“Sorted” search-key order

- **Theorem 10-1**
  The *inorder* traversal of a binary search tree $T$ will visit its nodes in sorted search-key order.

- **Proof by induction.**
Figure 10-19
An array of names in sorted order
typedef desired-type-of-search-key KeyType;

class KeyedItem
{
  public:
    KeyedItem() {};
    KeyedItem(const KeyType & keyValue): searchKey(KeyValue) {}
    KeyType getKey() const
    {
      return searchKey;
    } // getKey

  private:
    KeyType searchKey;
    // and other data about the person
}; // end class
TreeNode.h

#include "KeyedItem.h"
typedef KeyedItem TreeItemType;
class TreeNode // node in the tree
{
  private:
    TreeNode();
    TreeNode(const TreeItemType & nodeItem,
      TreeNode *left=NULL, TreeNode *right=NULL):
      item(nodeItem), leftChildPtr(left), rightChildPtr(right) {}

    TreeItemType item;  // data portion
    TreeNode       *leftChildPtr, *rightChildPtr;

  friend class BinarySearchTree;
};  // end class TreeNode
// *************************************************
// Header file BST.h for the ADT binary search tree.
// Assumptions: A tree contains at most one item with a given
// search key at any time.
// *************************************************

#include "TreeNode.h"

typedef void (*FunctionType)(TreeItemType& anItem);
class BinarySearchTree
{
    public:
    // constructors and destructor:
    bstClass();                // default constructor
    bstClass(const bstClass& Tree); // copy constructor
    virtual ~bstClass();       // destructor

    // binary search tree operations:
    // Precondition for all methods: **No** two items in a binary
    // search tree have the **same** search key.
Efficiency of BST Operations

1. **Complete** trees and full trees have *minimum* height.

2. The height of an N-node binary search tree ranges from $\left\lceil \log_2(N + 1) \right\rceil$ to N.

3. Insertion in search-key *order* produces a *maximum-height* binary search tree.

*Figure 10-28*

A maximum-height binary tree with seven nodes
4. Insertion in *random* order produces a *near-minimum-height* binary tree.

*Figure 10-35*

A tree of minimum height that is not complete
# Efficiency of BST Operations

<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>

*Figure 10-35*

A tree of minimum height that is not complete
Homework 2008

Page 580  #6, #7, #10, #14, #19
Treesort

See chapter 9-2 Sorting
Figure 10-33
(a) A binary search tree \( T \); (b) the sequence of insertions that result in this tree:

```
T.SearchTreeInsert(60, Success);
T.SearchTreeInsert(20, Success);
T.SearchTreeInsert(10, Success);
T.SearchTreeInsert(40, Success);
T.SearchTreeInsert(30, Success);
T.SearchTreeInsert(50, Success);
T.SearchTreeInsert(70, Success);
```
Figure 10-34
A full tree saved in a file by using inorder traversal
Examples

```c
binTreeClass T1;
binTreeClass T2(Root2);
binTreeClass T3(Root3);
binTreeClass T5(NodePtr);
```

T2: root2

T3: root3

T4: root4

T5: NodePtr

```c
binTreeClass T4(Root4, T2, T3);
```
constant variable

const int max_size = 256;
- A symbolic constant (i.e. a read-only variable)
- Any attempt to change the value in a program will result in a compiler time error.

const double minWage = 4.80;
const double *pc; // a pointer to a const object of type double
- pc can be changed to address a different variable.
- The value of the object addressed cannot be modified.
const reference argument

- The reference argument type is NOT allowed to be modified within the called program.
- It is a good practice to declare an input argument as const.
const member function

Nonclass object

An attempt to modify a nonclass `const` object in a program is flagged as a compiler-time error, e.g.,

```cpp
const char blank ="";
blank = '\0'; //error
```
const member function (cont’d)

class object

- A class object is typically modified by invoking its public member functions.
- To enforce the “constness” of a class object, a compiler needs to distinguish “safe” and “unsafe” member functions.
- Example,

```
const Screen blankScreen;
blankScreen.display(); //safe
blankScreen.set("*"); //unsafe
```
Safe member function

To indicate which member functions are safe by specifying them as const, e.g.,

```cpp
class Screen{
    public:
        char get() const {return *cursor;}
        //…
};
```

Only const member functions can be invoked by a const class object.

```cpp
const Screen cs;
char ch=cs.get();
```
const member function (cont’d)

- A const member function cannot modify a data member.

```cpp
Class Screen{
    public:
        void ok(char ch) const { *cursor = ch; } //legal
        void error(char *pch) const { cursor = pch; } //illegal
    //…
    private:
        char * cursor;
    //…
};
```

Copyright 2008 Yeali S. Sun. All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form, or by any means without the prior written permission of the author.
A pointer to a function: motivation

- Be able to change algorithms without requiring a change to user code.
- Allow the code to be fine-tuned after it is up and running.
- Typically done by having user code manipulate a function argument or pointer.
- The argument is a pointer to a function.
A pointer to a function

- Declare a function
  ```c
  void quickSort(int *, int, int);
  ```

- Declare a pointer to a function
  ```c
  void (*pf) (int *, int, int); // a pointer of the same type as quickSort
  ```