Hierarchical structures: Trees
Objectives

Discuss the following topics:

• Trees, Binary Trees, and Binary Search Trees
• Implementing Binary Trees
• Tree Traversal
• Searching a Binary Search Tree
• Insertion
• Deletion
Objectives (continued)

Discuss the following topics:

• Heaps
• Balancing a Tree
• Self-Adjusting Trees
Trees, Binary Trees, and Binary Search Trees

• A **tree** is a data type that consists of **nodes** and **arcs**

• These trees are depicted upside down with the root at the top and the **leaves** (**terminal nodes**) at the bottom

• The **root** is a node that has no parent; it can have only child nodes

• Leaves have no children (their children are null)
Each node has to be reachable from the root through a unique sequence of arcs, called a **path**.

The number of arcs in a path is called the **length** of the path.

The **level** of a node is the length of the path from the root to the node plus 1, which is the number of nodes in the path.

The **height** of a nonempty tree is the maximum level of a node in the tree.
Trees, Binary Trees, and Binary Search Trees (continued)

Figure 6-1 Examples of trees
Trees, Binary Trees, and Binary Search Trees (continued)

Figure 6-2 Hierarchical structure of a university shown as a tree
Trees: abstract/mathematical
important, great number of varieties

• terminology
  (knoop, wortel, vader, kind)
  node/vertex, root, father/parent, child
  (non) directed
  (non) orderly
  binary trees (left ≠ right)
  full (sometimes called decision trees, see Drozdek), complete (all
  levels are filled, except the last one)

• categorization
  structure
    number of children (binary, B-boom)
    height of subtrees (AVL-, B-trees)
    compleet (heap)
  Location of keys
    search tree, heap
Recall Definition of Tree

1. An empty structure is a tree
2. If $t_1, \ldots, t_k$ are disjoint trees, then the structure whose root has as its children the roots of $t_1, \ldots, t_k$ is also a tree
3. Only structures generated by rule 1 and 2 are trees

Alternatively: a connected graph which contains no cycles is a tree
Equivalent statements (see φ1)

• Let T be graph with n vertices then the following are equivalent:
  a) T is a tree ( = no cycles and connected )
  b) T contains no cycles, and has n-1 edges
  c) T is connected, and has n-1 edges
  d) T is connected, and every edge is a bridge
  e) Any two vertices are connected by exactly one path
  f) T contains no cycles, but the addition of any new edge creates exactly one circuit (cycle with no repeated edges).
Trees, Binary Trees, and Binary Search Trees (continued)

- An **orderly tree** is where all elements are stored according to some predetermined criterion of ordering.

![Diagram of an orderly tree](image)

Figure 6-3 Transforming (a) a linked list into (b) a tree
Binary Trees

• **A binary tree** is a tree whose nodes have two children (possibly empty), and each child is designated as either a left child or a right child.

Figure 6-4 Examples of binary trees
Binary Trees

• In a **complete binary tree**, all the nodes at all levels have two children except the last level.

• A **decision tree** is a binary tree in which all nodes have either zero or two nonempty children.
Remark on definition of Complete Binary Tree

• Drozdek Page218 uses the following definition: a complete binary tree is a binary tree of which all non-terminal nodes have both their children, and all leaves are at the same level.

• The more common definition is as follows: A complete binary tree is a binary tree in which every level, except possibly the last, is completed filled, and all nodes are as far left as possible.
Binary Trees

• At level i in binary tree at most $2^{i-1}$ nodes
• For non-empty binary tree whose nonterminal nodes (i.e., a full binary tree) have exactly two nonempty children: \# of leaves = 1+\#nonterminal nodes
• In a Drozdek-complete tree: \# of nodes = $2^{\text{height}}-1$; one way to see this is to use the statement \# of leaves = 1+\#nonterminal nodes; another way is to count how many nodes there are in each level and then sum the geometric progression;
Figure 6-5 Adding a leaf to tree (a), preserving the relation of the number of leaves to the number of nonterminal nodes (b)
ADT Binary Tree (more explicitly)

`createBinaryTree()`  // creates an empty binary tree

`createBinary(rootItem)`  // creates a one-node bin tree whose root contains rootItem

`createBinary(rootItem, leftTree, rightTree)`  // creates a bin tree whose root contains rootItem  // and has leftTree and rightTree, respectively, as its left and right subtrees

`destroyBinaryTree()`  // destroys a binary tree

`rootData()`  // returns the data portion of the root of a nonempty binary tree

`setRootData(newItem)`  // replaces the the data portion of the root of a nonempty bin tree with newItem. If the bin tree is empty, however, creates a root node whose data portion is newItem and inserts the new node into the tree

`attachLeft(newItem, success)`  // attaches a left child containing newItem to the root of a binary tree. Success indicates whether the operation was successful.

`attachRight(newItem, success)`  // analogous to attachLeft
attachLeftSubtree(leftTree, success) // Attaches leftTree as the left subtree to the root of a bin tree. Success indicates whether the operation was successful.

attachRightSubtree(rightTree, success) // analogous to attachLeftSubtree
detachLeftSubtree(leftTree, success) // detaches the left subtree of a bin tree’s root and retains it in leftTree. Success indicates whether the op was successful.

detachRightSubtree(rightTree, success) // similar to detachLeftSubtree
leftSubtree() // Returns, but does not detach, the left subtree of a bin tree’s root

rightSubtree() // analogous to leftSubtree

preorderTraverse(visit) // traverses a binary tree in preorder and calls the function visit once for each node
inorderTraverse(visit) // analogous: inorder
postorderTraverse(visit) // analogous: postorder
Implementing Binary Trees

• Binary trees can be implemented in at least two ways:
  – As arrays
  – As linked structures

• To implement a tree as an array, a node is declared as an object with an information field and two “reference” fields
Implementing Binary Trees (continued)

Implementing Binary Trees (continued)

Can do array for complete binary trees;
Level order storage;
Parent of A[i] is A[i div 2]:

Heapsort
Also for trees of max degree k (at most k children)
template <class T>
struct TreeNode {
    T info;
    TreeNode<T> *left, *right;
    int tag; // a.o. For threading

    TreeNode ( const T& i, 
        TreeNode<T> *left = NULL, 
        TreeNode<T> *right = NULL )
        : info(i)
    { left = l; right = r; tag = 0; }
};

See the next slide for the proof of concept; type T=int, is hardwired
The programmed ADT Binary Tree (refers to slide 20, 21: ADT Binary Tree) not parametrized: itemType = int

```cpp
#include <iostream>

using namespace std;

struct TreeNode {
    TreeNode * left;
    int data;
    TreeNode * right;
};

class Tree {
public:
    Tree(); // creates empty tree
    Tree(int rootItem);
    Tree(int rootItem, Tree leftTree, Tree rightTree);
    void setRootData(int newItem);
    void attachLeft(int newItem);
    void attachRight(int newItem);
    void attachLeftSubtree(Tree leftTree); //void attachRightSubtree(Tree rightTree);
    void detachLeftSubtree(Tree & leftTree); //void detachRightSubtree(Tree & rightTree);

private
    TreeNode * root;
};
```

// Client

```cpp
#include "tree.h"
#include <iostream>

using namespace std;

int main (){
    Tree t;
    t.setRootData(5);
    t.attachLeft(3);
    t.attachRight(7);

    // temporarily made everything public in order to inspect
    cout << "t.root->data: " << t.root->data << "n";
    cout << "t.root->left->data: " << t.root->left->data << "n";
    cout << "t.root->right->data: " << t.root->right->data << "n";
    return 1;
}
```

// Impl.

```cpp
#include "tree.h"
#include <iostream>

using namespace std;

Tree::Tree() { root = 0; }
Tree::Tree(int rootItem) {
    TreeNode * root = new TreeNode();
    root -> left = 0;
    root -> right = 0;
    root -> data = rootItem;
}

Tree::Tree(int rootItem, Tree leftTree, Tree rightTree) {
    TreeNode * root = new TreeNode();
    root -> data = rootItem;
    root -> left = 0;
    root -> right = 0;
    // attachLeftSubtree(leftTree);
    //attachRightSubtree(rightTree);
}

void Tree::setRootData(int newItem) {
    if (root != 0) {
        root -> data = newItem;
    } else {
        root = new TreeNode();
        root -> data = newItem;
        root -> left = 0;
        root -> right = 0;
    }
}

void Tree::attachLeft(int newItem) {
    if (root != 0) {
        if (root -> left == 0) {
            root -> left = new TreeNode();
            root -> left -> data = newItem;
            root -> left -> left = 0;
            root -> left -> right = 0;
        }
    }
}

void Tree::attachRight(int newItem) {
    if (root != 0) {
        if (root -> right == 0) {
            root -> right = new TreeNode();
            root -> right -> data = newItem;
            root -> right -> left = 0;
            root -> right -> right = 0;
        }
    }
}
```
// ********************** genBST.h **********************

//
// generic binary search tree

#include <queue>
#include <stack>

using namespace std;

#ifndef BINARY_SEARCH_TREE
#define BINARY_SEARCH_TREE

template<class T>
class Stack : public stack<T> { /*...*/ } // as in Figure 4.21

template<class T>
class Queue : public queue<T> {
public:

    T dequeue() {
        T tmp = front();
        queue<T>::pop();
        return tmp;
    }

    void enqueue(const T& el) {
        push(el);
    }
};

template<class T>
class BSTNode {
public:

    BSTNode() {
        left = right = 0;
    }

    BSTNode(const T& el, BSTNode *l = 0, BSTNode *r = 0) {
        key = el; left = l; right = r;
    }

    T key;
    BSTNode *left, *right;
};
template<class T>
class BST {
public:
    BST() {
        root = 0;
    }
    ~BST() {
        clear();
    }
    void clear() {
        clear(root); root = 0;
    }
    bool isEmpty() const {
        return root == 0;
    }
    void preorder() {
        preorder(root); // Figure 6.11
    }
    void inorder() {
        inorder(root); // Figure 6.11
    }
    void postorder() {
        postorder(root); // Figure 6.11
    }
    T* search(const T& el) const {
        return search(root, el); // Figure 6.9
    }
    void breadthFirst(); // Figure 6.10
    void iterativePreorder(); // Figure 6.15
    void iterativeInorder(); // Figure 6.17
    void iterativePostorder(); // Figure 6.16
    void MorrisInorder(); // Figure 6.20
    void insert(const T&); // Figure 6.23
    void deleteByMerging(BSTNode<T>*&); // Figure 6.29
    void findAndDeleteByMerging(const T&); // Figure 6.29
    void deleteByCopying(BSTNode<T>*&); // Figure 6.32
    void balance(T*, int, int); // Section 6.7
    .................

protected:
    BSTNode<T>* root;
void clear(BSTNode<T>*);
T* search(BSTNode<T>* const T&) const; // Figure 6.9
void preorder(BSTNode<T>*); // Figure 6.11
void inorder(BSTNode<T>*); // Figure 6.11
void postorder(BSTNode<T>*); // Figure 6.11
virtual void visit(BSTNode<T>* p) {
    cout << p->key << ' ';
}
    . . . . . . . . . . . . . . . .
};

#endif
Traversal of Binary Trees
Traversal of Binary Trees
Traversals of Binary Trees

• Is the process of visiting each node (precisely once) in a systematic way (visiting has technical meaning, a visit can possibly ‘write’ to the node, or change the structure of the tree, so you need to do it precisely once for each node; you can ‘pass’ by a node many times when only reading, for instance)

• Why?
  – Get info, updates
  – Check for structural properties, updating
  – Definitely can be extended to graphs (with cycles)!

• Methods:
  – Depth first (recursively or iteratively with stacks):
    • preorder (VLR),
    • inorder (symmetric) - LVR,
    • postorder (LRV)

in level order (breadth first)
Traversals of Binary Trees

- Recursively
- Iteratively: stacks (Depth First)
- Queues for Breadth First
- Threaded Trees
- Tree Transformation (e.g., Morris)
Tree Traversal: breadth-first

- **Breadth-first traversal** is visiting each node starting from the lowest (or highest) level and moving down (or up) level by level, visiting nodes on each level from left to right (or from right to left)
Tree Traversal: breadth-first

**URE 6.10** Top-down, left-to-right, breadth-first traversal implementation.

```cpp
template<class T>
void BST<T>::breadthFirst() {
    Queue<BSTNode<T>>* queue;
    BSTNode<T> *p = root;
    if (p != 0) {
        queue.enqueue(p);
        while (!queue.empty()) {
            p = queue.dequeue();
            visit(p);
            if (p->left != 0)
                queue.enqueue(p->left);
            if (p->right != 0)
                queue.enqueue(p->right);
        }
    }
}
```
Depth-First Traversal

• **Depth-first traversal** proceeds as far as possible to the left (or right), then backs up until the first crossroad, goes one step to the right (or left), and again as far as possible to the left (or right)
  
  – **V** — Visiting a node
  – **L** — Traversing the left subtree
  – **R** — Traversing the right subtree
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
    if (p != 0) {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}

template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}

template<class T>
void BST<T>::postorder(BSTNode<T>* p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
Inorder Tree Traversal

**FIGURE 6.12**  Inorder tree traversal.

(a)  
(b)  
(c)  
(d)  
(e)
template<class T>
void BST<T>::iterativeInorder() {
    Stack<BSTNode<T>*>* travStack;
    BSTNode<T>* p = root;
    while (p != 0) {
        while (p != 0) { // stack the right child (if any)
            if (p->right) // and the node itself when going
                travStack.push(p->right); // to the left;
            travStack.push(p);
            p = p->left;
        }
        p = travStack.pop(); // pop a node with no left child
        while (!travStack.empty() && p->right == 0) { // visit it
            visit(p); // and all nodes with no right
            p = travStack.pop(); // child;
        }
        visit(p); // visit also the first node with
        if (!travStack.empty()) // a right child (if any);
            p = travStack.pop();
        else p = 0;
    }
}
Preorder Traversal – iterative uses a stack

S.create();
S.push(root);

while (not S.isEmpty()) {
    current = S.pop() // a retrieving pop
    while (current ≠ NULL) {
        visit(current);
        S.push(current -> right);
        current = current -> left
    } // end while
} // end while
template<class T>
void BST<T>::iterativePreorder() {
    Stack<BSTNode<T>*> travStack;
    BSTNode<T> *p = root;
    if (p != 0) {
        travStack.push(p);
        while (!travStack.empty()) {
            p = travStack.pop();
            visit(p);
            if (p->right != 0)
                travStack.push(p->right);
            if (p->left != 0)  // left child pushed after right
                travStack.push(p->left);  // to be on the top of
                // the stack;
        }
    }
}
Stackless Depth-First Traversal

• **Threads** are references to the predecessor and successor of the node according to an inorder traversal
• Trees whose nodes use threads are called **threaded trees**
A threaded tree; an inorder traversal´s path in a threaded tree with Right successors only
A threaded tree; right pointers: successors; left pointers: predecessors
MorrisInOrder()
while not finished
   if node has **NO** left descendant
      visit it;
      go to the right;
   else make this node the right child of the rightmost node
       in its left descendant;
       go to this left descendant

FIGURE 6.21  Tree traversal with the Morris method.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram (a)" /></td>
<td><img src="image2" alt="Diagram (b)" /></td>
<td><img src="image3" alt="Diagram (c)" /></td>
<td><img src="image4" alt="Diagram (d)" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(e)</th>
<th>(f)</th>
<th>(g)</th>
<th>(h)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5" alt="Diagram (e)" /></td>
<td><img src="image6" alt="Diagram (f)" /></td>
<td><img src="image7" alt="Diagram (g)" /></td>
<td><img src="image8" alt="Diagram (h)" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
<th>(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image9" alt="Diagram (i)" /></td>
<td><img src="image10" alt="Diagram (j)" /></td>
<td><img src="image11" alt="Diagram (k)" /></td>
</tr>
</tbody>
</table>
Figure 6-20 Implementation of the Morris algorithm for inorder traversal

```cpp
template<class T>
void BST<T>::MorrisInorder() {
    BSTNode<T> *p = root, *tmp;
    while (p != 0)
        if (p->left == 0) {
            visit(p);
            p = p->right;
        }
        else {
            tmp = p->left;
            while (tmp->right != 0 && // go to the rightmost node
                   tmp->right != p) // of the left subtree or
                tmp = tmp->right; // to the temporary parent
            if (tmp->right == 0) { // of p; if 'true'
                tmp->right = p; // rightmost node was
                p = p->left; // reached, make it a
                            // temporary parent of the
            } else { // current root, else
                visit(p); // a temporary parent has
                            // been found; visit node p
                tmp->right = 0; // and then cut the right
                                  // pointer of the current
                p = p->right; // parent, whereby it
                         // ceases to be a parent;
            }
        }
    
}```
Binary Search Trees

Figure 6-6 Examples of binary search trees
Searching a Binary Search Tree (continued)

- The \textbf{internal path length (IPL)} is the sum of all path lengths of all nodes.
- It is calculated by summing $\Sigma (i - 1)l_i$ over all levels $i$, where $l_i$ is the number of nodes on level $i$.
- A depth of a node in the tree is determined by the path length.
- An average depth, called an \textbf{average path length}, is given by the formula $\text{IPL}/n$, which depends on the shape of the tree.
Insertion

Figure 6-22 Inserting nodes into binary search trees
Insertion (continued)

Figure 6-23 Implementation of the insertion algorithm.

```cpp
template<class T>
void BST<T>::insert(const T& el) {
    BSTNode<T> *p = root, *prev = 0;
    while (p != 0) {                      // find a place for inserting new node;
        prev = p;
        if (p->key < el)
            p = p->right;
        else p = p->left;
    }
    if (root == 0)                      // tree is empty;
        root = new BSTNode<T>(el);
    else if (prev->key < el)
        prev->right = new BSTNode<T>(el);
    else prev->left = new BSTNode<T>(el);
}
```
Insertion (continued)

Figure 6-25 Inserting nodes into a threaded tree
Deletion in BSTs

• There are three cases of deleting a node from the binary search tree:
  – The node is a leaf; it has no children
  – The node has one child
  – The node has two children
Deletion (continued)

Figure 6-26 Deleting a leaf

Figure 6-27 Deleting a node with one child
Deletion by Merging

• Making one tree out of the two subtrees of the node and then attaching it to the node’s parent is called **deleting by merging**

Figure 6-28 Summary of deleting by merging
Deletion by Copying

• If the node has two children, the problem can be reduced to:
  – The node is a leaf
  – The node has only one nonempty child
• Solution: replace the key being deleted with its immediate predecessor (or successor)
• A key’s predecessor is the key in the rightmost node in the left subtree