COMP2160/2860: Data Structures

The University of Sydney
School of Information Technologies

Week 3:
- Trees
- Binary search trees
- Balanced trees

Terminology

- Definition of a general tree
  - A general tree T is a set of one or more nodes such that T is partitioned into disjoint subsets:
    - A single node r, the root
    - Sets that are general trees, called subtrees of r

- Definition of a binary tree
  - A binary tree is a set T of nodes such that either
    - 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

Terminology

- A binary search tree
  - A binary tree that has the following properties for each node n
    - n’s value is greater than all values in its left subtree TL
    - n’s value is less than all values in its right subtree TR
    - Both TL and TR are binary search trees

Terminology

- The height of trees
  - Level of a node n in a tree T
    - If n is the root of T, it is at level 1
    - If n is not the root of T, its level is 1 greater than the level of its parent
  - Height of a tree T defined in terms of the levels of its nodes
    - If T is empty, its height is 0
    - If T is not empty, its height is equal to the maximum level of its nodes

Terminology

- Terminology
  - Definition of a general tree
    - A general tree T is a set of one or more nodes such that T is partitioned into disjoint subsets:
      - A single node r, the root
      - Sets that are general trees, called subtrees of r
  - Definition of a binary tree
    - A binary tree is a set T of nodes such that either
      - 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

Figure 10.4
Binary trees that represent algebraic expressions

Figure 10.5
A binary search tree of names

Figure 10.6
Binary trees with the same nodes but different heights
Terminology

• Full, complete, and balanced binary trees
  – Recursive definition of a full binary tree
    • 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

Figure 10.7
A full binary tree of height 3

• Complete binary trees
  – A binary tree T of height h is complete if
    All nodes at level h – 2 and above have two children each, and
    When a node at level h – 1 has children, all nodes to its left at
    the same level have two children each, and
    When a node at level h – 1 has one child, it is a left child

Figure 10.8
A complete binary tree

• Balanced binary trees
  – A binary tree is 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

• Full binary trees are complete
• Complete binary trees are balanced

Summary of tree terminology (Continued)

• Leaf
  • A node with no children
• Siblings
  • Nodes with a common parent
• Ancestor of node n
  • A node on the path from the root to n
• Descendant of node n
  • A node on a path from n to a leaf
• Subtree of node n
  • A tree that consists of a child (if any) of n and the child’s
descendants

• Height
  • The number of nodes on the longest path from the root to
    a leaf
• Binary tree
  • A set of nodes that is either empty or partitioned into a
    root node and one or two subsets that are binary
    subtrees of the root
  Each node has at most two children, the left child and the
  right child
• Left (right) child of node n
  • A node directly below node n in the tree
  • A node directly below node n in the tree
  – Root
  • The only node in the tree with no parent

• Summary of tree terminology
  – General tree
  • A set of one or more nodes, partitioned into a root node
    and subsets that are general subtrees of the root
  – Parent of node n
  • The node directly above node n in the tree
  – Child of node n
  • A node directly below node n in the tree
  – Root
  • The only node in the tree with no parent
Terminology

• Summary of tree terminology (Continued)
  – Left (right) subtree of node n
    • In a binary tree, the left (right) child (if any) of node n plus its descendants
  – Binary search tree
    • A binary tree where the value in any node n is greater than the value in every node in n’s left subtree, but less than the value of every node in n’s right subtree
  – Empty binary tree
    • A binary tree with no nodes

• Summary of tree terminology (Continued)
  – Full binary tree
    • A binary tree of height h with no missing nodes
    • All leaves are at level h and all other nodes each have two children
  – Complete binary tree
    • A binary tree of height h that is full to level h – 1 and has level h filled in from left to right
    • Balanced binary tree
      • A binary tree in which the left and right subtrees of any node have heights that differ by at most 1

Properties of trees

• No cycles
  – a tree is a connected acyclic graph
• Every two nodes are connected by exactly one path
  – simplest way to connect given nodes
• Connected graph with n nodes and n-1 edges

General Operations of the ADT Binary Tree

• General operations of the ADT binary tree
  – `createBinaryTree(rootItem, leftTree, rightTree)`
  – `setRootItem(newItem)`
  – `attachLeft(newItem)` throws `TreeException`
  – `attachRight(newItem)` throws `TreeException`
  – `attachLeftSubtree(leftTree)` throws `TreeException`
  – `attachRightSubtree(rightTree)` throws `TreeException`
  – `detachLeftSubtree()` throws `TreeException`
  – `detachRightSubtree()` throws `TreeException`

The ADT Binary Tree: Basic Operations of the ADT Binary Tree

• The operations available for a particular ADT binary tree depend on the type of binary tree being implemented
• Basic operations of the ADT binary tree
  – `createBinaryTree()`
  – `createBinaryTree(rootItem)`
  – `makeEmpty()`
  – `isEmpty()`
  – `getRootItem()` throws `TreeException`

Traversals of a Binary Tree

• A traversal algorithm for a binary tree visits each node in the tree
• Recursive traversal algorithms
  – Preorder traversal
  – Inorder traversal
  – Postorder traversal
• Traversal is O(n)
Traversal of a Binary Tree

Possible Representations of a Binary Tree

- An array-based representation
  - A Java class is used to define a node in the tree
  - A binary tree is represented by using an array of tree nodes
  - Each tree node contains a data portion and two indexes (one for each of the node's children)
  - Requires the creation of a free list which keeps track of available nodes

Possible Representations of a Binary Tree

- An array-based representation of a complete tree
  - If the binary tree is complete and remains complete
    - A memory-efficient array-based implementation can be used

Possible Representations of a Binary Tree

- An array-based representation of a complete tree
  - A node $\text{tree}[i]$
  - left child is in $\text{tree}[2i+1]$
  - right child in $\text{tree}[2i+2]$
  - parent in $\text{tree}[(i-1)/2]$
Possible Representations of a Binary Tree

• An array-based representation of a complete tree
• Very efficient for complete trees
• Wastes space if inner nodes are missing

Possible Representations of a Binary Tree

• A reference-based representation
  – Java references can be used to link the nodes in the tree

Figure 10.13
A reference-based implementation of a binary tree

A Reference-Based Implementation of the ADT Binary Tree

• Classes which provide a reference-based implementation for the ADT binary tree
  – TreeNode
    • Represents a node in a binary tree
  – TreeException
    • An exception class
  – BinaryTreeBasis
    • An abstract class of basic tree operation
  – BinaryTree
    • Provides the general operations of a binary tree
    • Extends BinaryTreeBasis

A Reference-Based Binary Tree

public class TreeNode {
    private Object item;
    private TreeNode leftChild;
    private TreeNode rightChild;
    ...
    // constructors, getLeft(),
    // getRight(), getItem() etc
}

A Reference-Based Binary Tree

public class BinaryTreeBasis {
    protected TreeNode root;
    // constructors...
    public boolean isEmpty() {...};
    public void makeEmpty() {...};
    public Object getRootItem() {...};
}

A Reference-Based Binary Tree

public class BinaryTree
    extends BinaryTreeBasis {
    // constructors...
    public void attachLeftSubtree
            (BinaryTree leftTree) {...};
    ...
}
Tree Traversals

- Basic tree traversals
  - In-order, pre-order, post-order
- Recursive implementations
- Tree iterator implements these tree traversals

Tree Traversals Using an Iterator

- TreeIterator
  - Implements the Java `Iterator` interface
  - Provides methods to set the iterator to the type of traversal desired
  - Uses a queue to maintain the current traversal of the nodes in the tree
- Nonrecursive traversal
  - An iterative method and an explicit stack can be used to mimic actions at a return from a recursive call to `inorder`

Iterators

```java
public interface Iterator {
    public boolean hasNext(); // Returns true if the iteration has more elements.
    public Object next() throws java.util.NoSuchElementException; // Returns the next element in the iteration.
    public void remove() throws UnsupportedOperationException, IllegalStateException; // Removes from the underlying collection the last element returned by the iterator (optional operation).
} // end Iterator
```

```java
public class BasicListIterator implements java.util.ListIterator {
    private BasicListInterface list;
    private int loc; // location of iterator in list

    public BasicListIterator(BasicListInterface list) {
        this.list = list;
        loc = 0;
    } // end constructor

    public boolean hasNext() {
        return (loc < list.size());
    } // end hasNext

    public Object next() throws java.util.NoSuchElementException {
        // Returns the next element in the list.
        if (hasNext()) {
            Object item = list.get(loc+1);
            loc++;
            return item;
        } else {throw new java.util.NoSuchElementException();}
    }
} // end BasicListIterator
```

Tree Iterators

- Tree iterators can traverse a tree
  - in-order
  - post-order
  - pre-order
- Recursive implementation
- `TreeIterator` extends java `Iterator` interface

```java
TreeIterator
    treeIt = new TreeIterator(tree1);
    treeIt.setPostorder();
    ...
    while (treeIt.hasNext()) {
        System.out.println(treeIt.next());
    }
```
Tree Iterators

public class TreeIterator implements java.util.Iterator {
    private BinaryTreeBasis binTree;
    private TreeNode currentNode;
    private QueueInterface queue;

    public TreeIterator(BinaryTreeBasis bTree) {
        binTree = bTree;
        currentNode = null;
        queue = new QueueReferenceBased();
    }

    public void setPreorder() {
        queue.dequeueAll();
        preorder(binTree.root);
    }

    public void setInorder() {
        queue.dequeueAll();
        inorder(binTree.root);
    }

    public void setPostorder() {
        queue.dequeueAll();
        postorder(binTree.root);
    }

    private void preorder(TreeNode treeNode) {
        if (treeNode != null) {
            queue.enqueue(treeNode);
            preorder(treeNode.getLeft());
            preorder(treeNode.getRight());
        }
    }

    private void inorder(TreeNode treeNode) {
        if (treeNode != null) {
            inorder(treeNode.getLeft());
            queue.enqueue(treeNode);
            inorder(treeNode.getRight());
        }
    }

    private void postorder(TreeNode treeNode) {
        if (treeNode != null) {
            postorder(treeNode.getLeft());
            postorder(treeNode.getRight());
            queue.enqueue(treeNode);
        }
    }

    private void preorder(TreeNode treeNode) {
        if (treeNode != null) {
            queue.enqueue(treeNode);
            preorder(treeNode.getLeft());
            preorder(treeNode.getRight());
        }
    }

    private void inorder(TreeNode treeNode) {
        if (treeNode != null) {
            inorder(treeNode.getLeft());
            queue.enqueue(treeNode);
            inorder(treeNode.getRight());
        }
    }

    private void postorder(TreeNode treeNode) {
        if (treeNode != null) {
            postorder(treeNode.getLeft());
            postorder(treeNode.getRight());
            queue.enqueue(treeNode);
        }
    }
}
Tree Iterators – post-order

```java
private void postorder(TreeNode treeNode) {
    if (treeNode != null) {
        postorder(treeNode.getLeft());
        postorder(treeNode.getRight());
        queue.enqueue(treeNode);
    } // end if
} // end postorder
} // end TreeIterator
```

The ADT Binary Search Tree

- **Record**
  - A group of related items, called fields, that are not necessarily of the same data type
- **Field**
  - A data element within a record
- A data item in a binary search tree has a specially designated search key
  - A search key is the part of a record that identifies it within a collection of records
- **KeyedItem class**
  - Contains the search key as a data field and a method for accessing the search key
  - Must be extended by classes for items that are in a binary search tree

Algorithms for the Operations of the ADT Binary Search Tree

- Since the binary search tree is recursive in nature, it is natural to formulate recursive algorithms for its operations
- **A search algorithm**
  - ```java
    search(bst, searchKey)
    ```
  - Searches the binary search tree `bst` for the item whose search key is `searchKey`

Binary Search Tree: Insertion

- ```java
  insertItem(treeNode, newItem)
  ```
  - Inserts `newItem` into the binary search tree of which `treeNode` is the root

The ADT Binary Search Tree

- **ADT binary search tree**
  - Searching for a particular item
- Each node `n` in a binary search tree satisfies the following properties
  - `n`’s value is greater than all values in its left subtree `T_L`
  - `n`’s value is less than all values in its right subtree `T_R`
  - Both `T_L` and `T_R` are binary search trees
Binary Search Tree: Insertion

Figure 10.21c

c) insertion at a leaf

Binary Search Tree: Deletion

- Steps for deletion
  - Use the search algorithm to locate the item with the specified key
  - If the item is found, remove the item from the tree
- Three possible cases for node N containing the item to be deleted
  - N is a leaf
  - N has only one child
  - N has two children

Binary Search Tree: Deletion

- Strategies for deleting node N
  - If N is a leaf
    - Set the reference in N’s parent to null
  - If N has only one child
    - Let N’s parent adopt N’s child
  - If N has two children
    - Locate another node M that is easier to remove from the tree than the node N
    - Copy the item that is in M to N
    - Remove the node M from the tree

Binary Search Tree: Retrieval

- Retrieval operation can be implemented by refining the search algorithm
  - Return the item with the desired search key if it exists
  - Otherwise, return a null reference

Binary Search Tree: Traversal

- Traversals for a binary search tree are the same as the traversals for a binary tree
- The inorder traversal of a binary search tree T will visit its nodes in sorted search-key order

A Reference-Based Implementation of the ADT Binary Search Tree

- BinarySearchTree
  - Extends BinaryTreeBasis
  - Inherits the following from BinaryTreeBasis
    - isEmpty()
    - makeEmpty()
    - getRootItem()
  - The use of the constructors
- TreeIterator
  - Can be used with BinarySearchTree
The Efficiency of Binary Search Tree Operations

- The maximum number of comparisons for a retrieval, insertion, or deletion is the height of the tree
- The maximum and minimum heights of a binary search tree – \( n \) is the maximum height of a binary tree with \( n \) nodes

The Efficiency of Binary Search Tree Operations

- Theorem 10-2
  A full binary tree of height \( h \geq 0 \) has \( 2^h - 1 \) nodes
- Theorem 10-3
  The maximum number of nodes that a binary tree of height \( h \) can have is \( 2^h - 1 \)

Treesort

- Treesort
  - Uses the ADT binary search tree to sort an array of records into search-key order
  - Efficiency
    - Average case: \( O(n \log n) \)
    - Worst case: \( O(n^2) \)

General Trees

- An \( n \)-ary tree
  - A generalization of a binary tree whose nodes each can have no more than \( n \) children

Balanced Search Trees

- The efficiency of the binary search tree implementation of the ADT table is related to the tree’s height
  - Height of a binary search tree of \( n \) items
    - Maximum: \( n \)
    - Minimum: \( \lceil \log_2(n+1) \rceil \)
  - Height of a binary search tree is sensitive to the order of insertions and deletions
  - Variations of the binary search tree
    - Can retain their balance despite insertions and deletions
2-3 Trees

- A 2-3 tree
  - Has 2-nodes and 3-nodes
    - A 2-node
      - A node with one data item and two children
    - A 3-node
      - A node with two data items and three children
  - Is not a binary tree
  - Is never taller than a minimum-height binary tree
- A 2-3 tree with n nodes never has height greater than \( \lceil \log_2(n + 1) \rceil \)

2-3 Trees

- Rules for placing items in the nodes of a 2-3 tree
  - A 2-node must contain a single data item whose search key is
    - Greater than the left child's search key(s)
    - Less than the right child's search key(s)
  - A 3-node must contain two data items whose search keys S and L satisfy the following
    - S is
      - Greater than the left child's search key(s)
      - Less than the middle child's search key(s)
    - L is
      - Greater than the middle child's search key(s)
      - Less than the right child's search key(s)
  - A leaf may contain either one or two data items

2-3 Trees

- Traversing a 2-3 tree
  - To traverse a 2-3 tree
    - Perform the analogue of an inorder traversal
- Searching a 2-3 tree
  - Searching a 2-3 tree is as efficient as searching the shortest binary search tree
  - Searching a 2-3 tree is \( O(\log_2 n) \)
  - Number of comparisons required to search a 2-3 tree for a given item
    - Approximately equal to the number of comparisons required to search a binary search tree that is as balanced as possible

2-3 Trees

- Advantage of a 2-3 tree over a balanced binary search tree
  - Maintaining the balance of a binary search tree is difficult
  - Maintaining the balance of a 2-3 tree is relatively easy

2-3 Trees: Inserting Into a 2-3 Tree

- Insertion into a 2-node leaf is simple
- Insertion into a 3-node causes it to divide
2-3 Trees: The Insertion Algorithm

- To insert an item $I$ into a 2-3 tree:
  - Locate the leaf at which the search for $I$ would terminate
  - Insert the new item $I$ into the leaf
  - If the leaf now contains only two items, you are done
  - If the leaf now contains three items, split the leaf into two nodes, $n_1$ and $n_2$

![Splitting a leaf in a 2-3 tree](image)

2-3 Trees: The Insertion Algorithm

- When an internal node contains three items:
  - Split the node into two nodes
  - Accommodate the node’s children

![Splitting an internal node in a 2-3 tree](image)

2-3 Trees: The Insertion Algorithm

- When the root contains three items:
  - Split the root into two nodes
  - Create a new root node

![Splitting the root of a 2-3 tree](image)

2-3 Trees: Deleting from a 2-3 Tree

- Deletion from a 2-3 tree:
  - Does not affect the balance of the tree
- Deletion from a balanced binary search tree:
  - May cause the tree to lose its balance

2-3 Trees: The Deletion Algorithm

![Redistributing values; merging a leaf](image)

![Redistributing values and children; merging internal nodes](image)
2-3 Trees: The Deletion Algorithm

- When analyzing the efficiency of the `insertItem` and `deleteItem` algorithms, it is sufficient to consider only the time required to locate the item.
- A 2-3 operation is $O(\log_2 n)$.
- A 2-3 tree is a compromise.
  - Searching a 2-3 tree is not quite as efficient as searching a binary search tree of minimum height.
  - A 2-3 tree is relatively simple to maintain.

2-3-4 Trees

- Rules for placing data items in the nodes of a 2-3-4 tree:
  - A 2-node must contain a single data item whose search keys satisfy the relationships pictured in Figure 12-3a.
  - A 3-node must contain two data items whose search keys satisfy the relationships pictured in Figure 12-3b.
  - A 4-node must contain three data items whose search keys $S$, $M$, and $L$ satisfy the relationships pictured in Figure 12-21.
  - A leaf may contain either one, two, or three data items.

2-3-4 Trees: Inserting into a 2-3-4 Tree

- The insertion algorithm for a 2-3-4 tree:
  - Splits a node by moving one of its items up to its parent node.
  - Splits 4-nodes as soon as its encounters them on the way down the tree from the root to a leaf.
  - Result: when a 4-node is split and an item is moved up to the node’s parent, the parent cannot possibly be a 4-node and can accommodate another item.

2-3-4 Trees: Splitting 4-nodes During Insertion

- A 4-node is split as soon as it is encountered during a search from the root to a leaf.
- The 4-node that is split will:
  - Be the root, or
  - Have a 2-node parent, or
  - Have a 3-node parent.
**2-3-4 Trees: Splitting 4-nodes During Insertion**

- Figure 12.29
  - Splitting a 4-node whose parent is a 2-node during insertion

- Figure 12.30
  - Splitting a 4-node whose parent is a 3-node during insertion

**2-3-4 Trees: Deleting from a 2-3-4 Tree**

- The deletion algorithm for a 2-3-4 tree
  - Locate the node $n$ that contains the item $\text{theItem}$
  - Find $\text{theItem}$'s inorder successor and swap it with $\text{theItem}$ (deletion will always be at a leaf)
  - If that leaf is a 3-node or a 4-node, remove $\text{theItem}$
  - To ensure that $\text{theItem}$ does not occur in a 2-node
    - Transform each 2-node encountered into a 3-node or a 4-node

**2-3-4 Trees: Remarks**

- Advantage of 2-3 and 2-3-4 trees
  - Easy-to-maintain balance
  - Trees grow or shrink at the root
  - Insertion and deletion algorithms for a 2-3-4 tree require fewer steps than those for a 2-3 tree
  - 2-3-4 tree operations are one-pass
  - Allowing nodes with more than four children is counterproductive

**Trees**

- Covered
  - Trees
  - Tree traversals
  - Binary search trees
  - Balanced trees
    - 2-3 trees, 2-3-4 trees