Part 6. Trees (1)

CS 200 Algorithms and Data Structures

Outline

- Terminology
- Binary Tree
  - Basic operations
  - Traversals of a Binary Tree
  - Representations of a Binary Tree
  - Implementations
- Binary Search Tree
  - Algorithms
  - Implementations
  - Efficiency
  - TreeSort
- General Tree

Data Representation

<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
  <to>Elmo</to>
  <from>Zoe</from>
  <heading>Reminder</heading>
  <body>Let's meet at Sesame Street</body>
</note>

Terminology

- Represent **hierarchical relationships**

The parent child relationship is generalized to the relationship of ancestor and descendant.
**Terminology**

- **Parent node**
- **Child node**
- **Siblings**

**Binary Tree**

- Set $T$ of nodes such that either,
  - $T$ is empty, or
  - $T$ is partitioned into three disjoint subsets
    - A single node $r$, **root**
    - Two possibly empty sets that are binary trees, called **left** and **right subtrees** of $r$
- Each node in a binary tree has no more than two children.

**General Tree**

- 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 $r$**
# Degree, depth, and Height

- **Degree**
  - Degree of a node: number of children
  - Degree of a tree: maximum degree of nodes.
- **Depth**: length of path from node to root.
- **Height** of a tree
  - The number of nodes on the longest path from the root to a leaf.

## Height of a Tree

- If $T$ is empty, its height is 0.
- If $T$ is not empty, its height is equal to the maximum level of its nodes.

## Height of a Binary Tree

- If $T$ is empty, its height is 0.
- If $T$ is a non empty binary tree, $height(T) = 1 + \max\{height(T_L), height(T_R)\}$

## Binary trees with same nodes but different heights

![Binary trees with same nodes but different heights](image)

## Full Binary Tree

- A full binary tree is a tree in which every node other than the leaves has two children.

## Definition of 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 $h - 1$. 
### Complete Binary Tree

- Is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

![Complete Binary Tree](image)

### Formal Definition of a Complete Binary Tree

- 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.
- Full binary tree is complete.

### 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**
  - The height of the two subtrees of every node never differ by more than 1


![Binary Tree](image)

### Outline

- Terminology
- **Binary Tree**
  - Basic operations
  - Traversals of a Binary Tree
  - Representations of a Binary Tree
  - Implementations
- Binary Search Tree
  - Algorithms
  - Implementations
  - Efficiency
  - TreeSort
- General Tree

### Operations of the Binary Tree

- Add and remove node and subtrees
- Retrieve and set the data in the root
- Determine whether the tree is empty
General operations

| Root  |  
|------|---
| left subtree | 
| right subtree | 

createBinaryTree();  
makeEmpty();  
isEmpty();  
getRootItem();  
setRootItem();  
attachLeft();  
attachRight();  
attachLeftSubtree();  
attachRightSubtree();  
detachLeftSubtree();  
detachRightSubtree();  
getLeftSubtree();  
getRightSubtree();

Example

tree1.setRootItem("F")
tree1.attachLeft("G")
tree2.setRootItem("D")
tree2.attachLeftSubtree(tree1)
tree3.setRootItem("B")
tree3.attachLeftSubtree(tree2)
tree3.attachRight("E")
tree4.setRootItem("C")
binTree.createBinaryTree("A", tree3, tree4)

Traversal Algorithms

• The traversal of a tree is the process of “visiting” every node of the tree  
  – Display a portion of the data in the node.  
  – Process the data in the node

• Because a tree is not linear, there are many ways that this can be done.

Breadth-first traversal

• Breadth-first processes the tree **level by level** starting at the root and handling all the nodes at a particular level from **left to right**.

Breadth-first traversal

• Three choices of when to visit the root r.  
  – These methods are recursive methods for tree traversal.  
  1. **Before** it traverses both of r’s subtrees  
  2. After it has traversed r’s **left** subtree (before it traverses r’s right subtree)  
  3. After it has traversed **both** of r’s subtrees

  • **Preorder**, **inorder**, and **postorder**
**Depth First: Preorder traversal**

- **Preorder traversal** processes the information at the root, followed by the entire left subtree and concluding with the entire right subtree.

**Depth First: Preorder traversal**

- **Preorder traversal** processes the information at the root, followed by the entire left subtree and concluding with the entire right subtree.

**Depth First: Inorder traversal**

- **Inorder traversal** processes all the information in the left subtree before processing the root.
- It finishes by processing all the information in the right subtree.

**Depth First: Inorder traversal**

- **Inorder traversal** processes all the information in the left subtree before processing the root.
- It finishes by processing all the information in the right subtree.

**Depth First: Postorder traversal**

- **Postorder traversal** processes the left subtree, then the right subtree and finishes by processing the root.

**Depth First: Postorder traversal**

- **Postorder traversal** processes the left subtree, then the right subtree and finishes by processing the root.
Use case: postorder traversal

- When destroying a tree, we can’t delete the root node until we have destroyed the left and right subtree
  – So a postorder traversal makes sense.

Example of Binary sorting

**Create Tree**

```
60 20 10 40 70 50 30
```

**Traversal Tree**

```
10 20 30 40 50 60 70
```

Use case: Preorder/Inorder traversal (Creating and Browsing of Binary Sorting)

<table>
<thead>
<tr>
<th>YES</th>
<th>value &lt; value at root?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value must be in the left subtree</td>
</tr>
<tr>
<td>NO</td>
<td>Value must be in the right subtree</td>
</tr>
</tbody>
</table>

Preorder algorithm

```
preorder (in binTree:BinaryTree)
if (binTree is not empty){
    display the data in the root of binTree
    preorder(Left subtree of binTree’s root)
    preorder(Right subtree of binTree’s root)
}
```

Postorder algorithm

```
postorder (in binTree:BinaryTree)
if (binTree is not empty){
    postorder(Left subtree of binTree’s root)
    display the data in the root of binTree
    postorder(Right subtree of binTree’s root)
}
```

Inorder algorithm

```
inorder (in binTree:BinaryTree)
if (binTree is not empty){
    inorder(Left subtree of binTree’s root)
    display the data in the root of binTree
    inorder(Right subtree of binTree’s root)
}
```
Analysis of Tree Traversal

• \( n \) visits occur for a tree of \( n \) nodes.
  \(- O(n) \)

An array-based representation (1/2)

<table>
<thead>
<tr>
<th>index</th>
<th>item</th>
<th>leftChild</th>
<th>rightChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Jane</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Bob</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Tom</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>Alan</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>Ellen</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>Nancy</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>?</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>?</td>
<td>-1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>?</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

A binary tree of names

Free list: Array-based Linked List

An array-based representation (2/2)

```java
public class TreeNode<T>{
  private T item;
  private int leftChild;
  private int rightChild;
  ...
  ... setters
}
```

```java
public class BinaryTreeArrayBased<T> {
  protected final int MAX_NODES = 100;
  protected ArrayList<TreeNode<T>> tree;
  protected int root;
  protected int free; //index of next unused array location
  ...
  public BinaryTreeArrayBased<T> () {
    tree  = new ArrayList<TreeNode<T>>()
  ...
  public void creatTree(TreeNode<T> _root){
    root = 0;
    tree.set(0, _root);
    free++;
  }
  ...
  ...
  public TreeNode<T> getRootItem(){
    return tree.get(root);
  }
  ...
}
```

An array-based representation (2/2)

```java
public class TreeNode<T>{
  ...
  ... setters
}
```

```java
public class BinaryTreeArrayBased<T> {
  ...
  ...
  public void creatTree(TreeNode<T> _root){
    root = 0;
    tree.set(0, _root);
    free++;
  }
  ...
  public TreeNode<T> getRootItem(){
    return tree.get(root);
  }
  ...
}
```

Complete Binary Tree

Level-by-level numbering of a complete binary tree

If the binary tree is complete and remains complete
A memory-efficient array-based implementation can be used
### A reference-based representation

**Step 1. TreeNode**

```java
class TreeNode<T> {
    T item;
    TreeNode<T> leftChild;
    TreeNode<T> rightChild;
    public TreeNode(T newItem) {
        item = newItem;
        leftChild = null;
        rightChild = null;
    }
    public TreeNode(T newItem, TreeNode<T> left, TreeNode<T> right) {
        item = newItem;
        leftChild = left;
        rightChild = right;
    }
}
```

**Step 2. Tree (BinaryTree)**

```java
class BinaryTree<T> {
    public BinaryTree() {
    }
    public BinaryTree(T rootItem, BinaryTree<T> leftTree, BinaryTree<T> rightTree) {
        root = new TreeNode<T>(rootItem, null, null);
        attachLeftSubtree(leftTree);
        attachRightSubtree(rightTree);
    }
    public void setRootItem(T newItem) {
        if (root != null) {
            root.item = newItem;
        } else {
            root = new TreeNode<T>(newItem, null, null);
        }
    }
    public void attachLeft(T newItem) {
        if (!isEmpty() && root.leftChild == null) {
            root.leftChild = new TreeNode<T>(newItem, null, null);
        }
    }
    public void attachRight(T newItem) {
        if (!isEmpty() && root.leftChild == null) {
            root.rightChild = new TreeNode<T>(newItem, null, null);
        }
    }
    public void attachLeftSubtree(BinaryTree<T> leftTree) throws TreeException {
        if (isEmpty()) {
            throw new TreeException("TreeException: Empty tree.");
        } else if (root.leftChild != null) {
            throw new TreeException("TreeException: cannot overwrite left subtree.");
        }
        root.leftChild = leftTree.root;
        leftTree.makeEmpty();
    }
    public void attachRightSubtree() throws TreeException {
        // similar to attachLeftSubtree()
    }
}
```

### A reference-based representation (1/6)

### A reference-based representation (2/6)

### A reference-based representation (3/6)

### A reference-based representation (4/6)
A reference based representation (5/6)

```java
public BinaryTree<T> detachLeftSubtree(BinaryTree<T> leftTree)
    throws TreeException{
    if (isEmpty()) {
        throw new TreeException("TreeException:Empty tree.");
    }
    else {
        BinaryTree<T> leftTree;
        leftTree = new BinaryTree<T>(root.leftChild);
        root.leftChild = null;
        return leftTree;
    }
    public void detachRightSubtree(...
similar to detachLeftSubtree();
```}

A reference based representation (6/6)

```java
public class TreeException extends RuntimeException{
    public TreeException(String s){
        super(s)
    }
}
```

Tree Traversals Using an Iterator

- **Iterator interface**
  - `next()`, `hasNext()`, and `remove()`
- 1. Use a queue to order the nodes according to the type of traversal.
- 2. Initialize iterator by type (pre, post or in) and enqueue all nodes, in order, necessary for traversal
- 3. dequeue in the `next` operation

What is Java Iterator?

- An iterator allows going over all the element of the collection in sequence
- Unlike Enumeration, iterator allows the caller to remove element from the underlying collection
  - `java.util.Iterator`
    - `boolean hasNext()`
    - `Object next()`
    - `void remove()`
  - `java.utilEnumeration`
    - `Boolean hasMoreElement()`
    - `Object nextElement()`

How to use Java Iterator?

```java
import java.util.*;
class IteratorDemo { 
    public static void main(String args[]) {
        // create an array list
        ArrayList al = new ArrayList();
        // add elements to the array list
        al.add("C"); al.add("A"); al.add("E");
al.add("B"); al.add("D"); al.add("F");
```

How to use Java Iterator?

```java
// use iterator to display contents of al
Iterator ite = al.iterator();
while(ite.hasNext()) {
    Object element = ite.next();
    System.out.print(element + " ");
}
```
Using TreeIterator for the Preorder

```
Using TreeIterator for the Inorder

Using TreeIterator for the Postorder

TreeIterator

import java.util.LinkedList;
public class TreeIterator<T> implements java.util.Iterator<T> {
    private BinaryTreeBasic<T> binTree;
    private TreeNode<T> currentNode;
    private LinkedList<TreeNode<T>> queue;
    public TreeIterator(BinaryTreeBasic<T> bTree) {
        binTree = bTree;
        currentNode = null;
        queue = new LinkedList<TreeNode<T>>();
        return !queue.isEmpty();
        currentNode = queue.remove();
        return currentNode.getItem();
        queue.clear();
        preorder(binTree.root);
        queue.clear();
        inorder(binTree.root);
        queue.clear();
        postorder(binTree.root);
    }
    public boolean hasNext() {}
    public T next() throws java.util.NoSuchElementException {}
    public void setPreorder() {}
    public void setInorder() {}
    public void setPostorder() {}
    private void preorder(TreeNode<T> treeNode) {}
    private void inorder(TreeNode<T> treeNode) {}
    private void postorder(TreeNode<T> treeNode) {}
}
```

Outline

- Terminology
- Binary Tree
  - Basic operations
  - Traversals of a Binary Tree
  - Representations of a Binary Tree
  - Implementations
- Binary Search Tree
  - Algorithms
  - Implementations
  - Efficiency
  - TreeSort
- General Tree

Find the record for the person whose ID number is 1212121212.
**Binary Search Trees**

- A binary tree $T$ is a **binary search tree** if for every node $n$ in $T$:
  - $n$'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 $T_R$
  - $T_R$ and $T_L$ are binary search trees

---

**Is this a Binary Search Tree?**

- Valid
- Not Valid

---

**Organizing Items in BST**

- BST uses **inorder** traversal.
- The sequence of adding and removing nodes influences the shape of the tree.

---

**BST operations (1/2)**

- Insert a new item
  - `insert(in newItem:TreeItemType)`
- Delete the item with a given search key
  - `delete(in searchKey:KeyType) throws TreeException`
- Retrieve the item with a given search key
  - `retrieve(in searchKey:KeyType):TreeItemType`

---

**Example of nodes in BST**

```java
public class Person {
  private String id;
  private String phoneNo;
  private String address;
  public Person( String _id, String _phoneNo, String _address){
    id = _id;
    phoneNo = _phoneNo;
    ...
  }
  //other methods appear here
}
```

---

**Search in BST**

- BST is a recursive algorithm.
- Search key should be unique.
- Using BST's properties.
  - If search key == search key of root's item
    - The record is found
  - If search key < search key of root's item
    - Search in the left subtree → recursive method
  - If search key > search key of root's item
    - Search in the right subtree → recursive method
**Pseudocode**

search(in bst: BinarySearchTree, in searchKey: KeyType)
if (bst is empty){
    The desired record is not found
} else if (searchKey == searchKey of root’s item){
    The desired record is found
} else if (searchKey < search key of root’s item) {
    search (Left subtree of bst, searchKey)
} else {
    search(Right subtree of bst, searchKey)
}

---

**Example: Find the person with ID 50**

- 50 is smaller than 60
  - Search in the left subtree

- 50 is bigger than 60
  - Search in the right subtree

- 50 is bigger than 40
  - Search in the left subtree

- 50 is bigger than 50
  - Search in the right subtree

---

**Example: Insert a person with ID 55**

- Use search to determine where in the tree to insert a new ID, 55.
- New item will be inserted as a new leaf.
- Different orders of insertion can get a different binary search tree.
**Pseudocode (high-level)**

```plaintext
insertItem (in TreeNode treeNode, in newItem:TreeItemType)
Let parentNode be the parent of the empty subtree at which search terminates when it seeks newItem's search key
if (search terminated at parentNode's left subtree){
    Set leftChild of parentNode to reference newItem
} else{
    Set rightChild of parentNode to reference newItem
}
```

**Reference to the new item**

![Reference to the new item diagram]

**Pseudocode**

```plaintext
insertItem (in TreeNode treeNode, in newItem:TreeItemType)
if (treeNode is null)
    create a new node and let treeNode reference it
    create a new node with newItem as the data portion
    set the reference in the new node to null
else if (newItem.getKey() < treeNode.item.getKey()){
    treeNode.leftChild = insertItem(treeNode.leftChild, newItem)
} else{
    treeNode.rightChild = insertItem(treeNode.rightChild, newItem)
}
return treeNode
```

**Deletion in BST**

- Emptying a tree Vs. Deleting a node from a tree?
- Use search algorithm to locate the item with the specified search key.
- Three cases to consider:
  1. N is a leaf
  2. N has only one child
  3. N has two children

**Case 1. N is a leaf**

- Set the reference in its parent to null.
Case 2. N has only one child (left child)

- Let N’s parent adopt N’s child.

- Let N’s parent adopt N’s child.

Case 2. N has only one child (right child)

Case 3. N has two children

1. Find the inorder successor of N’s search key.
   - The node whose search key comes immediately after N’s search key
   - The inorder successor is in the leftmost node in N’s right subtree.

2. Copy the item of the inorder successor, M to the deleting node N.

3. Remove the node M from the tree.

Remove 20

Does the inorder traversal of a BST visit its nodes in sorted-key order?

Theorem 6-1
The inorder traversal of a binary search tree $T$ will visit its nodes in sorted search-key order.

Proof

Retrieval in BST

```cpp
retrieveItem(treeNode: TreeNode, in searchKey: KeyType): TreeItemType
if (treeNode == null){
    treeItem = null
} else if (searchKey < treeNode.item.getKey){
    treeItem = retrieveItem(treeNode.leftChild, searchKey)
} else{
    treeItem = retrieveItem(treeNode.rightChild, searchKey)
} return treeItem
```
The Efficiency of BST Operations

- Searching, inserting, deleting, and retrieving data
- Compare the specified value searchKey to the search keys in the nodes along a path through the tree.
- The maximum number of comparisons that these operations can require is equal to the height of the binary search tree.

The maximum and minimum heights of a BST

- Maximum height: each internal node has exactly one child.

Counting the nodes in a full binary tree of height $h$

- Level 1: $1$ node ($2^0$), Total number of nodes until this level: $1 = 2^1 - 1$
- Level 2: $2$ nodes ($2^1$), Total number of nodes until this level: $3 = 2^2 - 1$
- Level 3: $4$ nodes ($2^2$), Total number of nodes until this level: $7 = 2^3 - 1$
- Level 4: $8$ nodes ($2^3$), Total number of nodes until this level: $15 = 2^4 - 1$
- Level $h$: $2^{h-1}$ nodes, Total number of nodes until this level: $2^h - 1$

Theorem 6-2
The maximum number of nodes that a binary tree on the level $h > 0$ can have is $2^{h-1}$ nodes

Proof

Theorem 6-3
A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes

Proof

Theorem 6-4
The minimum height of a binary tree with $n$ nodes is $\lceil \log(n + 1) \rceil$

Proof
• Complete trees and full trees with \( n \) nodes have heights of \( \lceil \log(n+1) \rceil \)
• The height of an \( n \)-node BST ranges from \( \lceil \log(n+1) \rceil \) to \( n \)
• Insertion in search-key order will need maximum-height BST.
  – Adding the node to the leaf node.

Treesort

• Sort an array of records efficiently into a search-key order.
• Each insertion into a BST requires \( O(\log n) \) operations
  in the average case, and \( O(n) \) for the worst case.
• \( n \) insertions requires \( O(n \log n) \) operations in the average case and \( O(n^2) \) in the worst case.
• Sorted items in the tree are traversed and copied to the array: \( O(n) \)
• Average case: \( O(n \log n) + O(n) = O(n \log n) \)
• Worst case: \( O(n^2) + O(n) = O(n^2) \)

Storing a BST in a file

• Storing and restoring
• Original shape vs. balanced shape?
  – There is NO difference in the ADT operations.
  – Efficient operations are assured if the binary search tree is balanced.

Example: store

Example: restore

How to guarantee a restored tree of minimum height

• Create complete binary search tree with \( n \) nodes

\[
\begin{align*}
\text{readTree} & \quad \text{in inputFile:FileType,} \\
& \quad \text{in n:integer):Treenode} \\
\text{if} \quad & \quad (n > 0) \{ \\
\quad \text{treeNode} = \text{reference to new node} \\
\quad \text{with null child references} \\
\quad \text{Set treeNode’s left child to} \\
\quad \text{readTree(inputFile, n/2)} \\
\quad \text{Read Item from file into treeNode’s item} \\
\quad \text{Set treeNode’s right child to} \\
\quad \text{readFull(inputFile, (n-1)/2)} \\
\} \\
\text{return treeNode}
\end{align*}
\]
Comparison

\[
\begin{array}{c}
\text{General tree} \\
• Tree with no more than } n \text{ children.} \\
• How can we implement it?
\end{array}
\]

\[
\begin{array}{c}
\text{Comparison} \\
\text{n-ary} \\
\end{array}
\]

\[
\begin{array}{c}
\text{n = 3} \\
\text{Case 1: using 2 references} \\
\text{Case 2: using 3 references} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Case 1: Using 2 references} \\
\text{Case 2: Using 3 references} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Case 2: Using 3 references} \\
\end{array}
\]