CS200: Trees

Walls Ch. 11

Tree Terminology

- **Node**
- **Edge**
- **Parent**
- **Root**
- **Leaf**
- **Interior Node**
- **Path**
- **Degree?**
- **Depth/Level?**
- **Height?**

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

All the defs are in page 525 of the textbook.

Binary Trees

- 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

- **Right Subtree**
- **Root**
- **Left Subtree**

Trees - more definitions

- **m-ary tree**
  - Every internal vertex has no more than m children
  - Our main focus will be binary trees

- **Full m-ary tree**
  - All interior nodes have m children

- **Perfect m-ary tree**
  - Full m-ary tree where all leaves are at the same level

- **Perfect binary tree**
  - Total number of nodes: $2^h - 1$
  - Recurrence relations for the # of leaf nodes and total # of nodes?

More definitions

- **Complete binary tree of height h**
  - Zero or more rightmost leaves not present at level h
  - 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
More definitions

- balanced tree
  - Height of any node's right subtree differs from left subtree by 0 or 1
- A complete tree is balanced

Applications - Expression Trees

- unambiguously represent infix expressions

Applications - Parse Trees

- Used in compilers to check syntax

Applications - Search Trees

- AI: search trees

  Example: a game tree

Applications - Decision Trees

- Example: a tree for deciding whether to wait for a table at a restaurant

Binary search trees

- A very efficient way of storing data!
Binary Tree ADT

- Create
  - createBinaryTree()
  - createBinaryTree(in rootItem:TreeItemType)

- Add/Modify
  - setRootItem(in rootItem:TreeItemType) throws UnsupportedOperationException

- Remove
  - makeEmpty()

- Ask
  - isEmpty():boolean {query}
  - getRootItem():TreeItemType throws TreeException {query}

Extensions to Binary Tree ADT

- Create
  - createBinaryTree(in rootItem:TreeItemType, in leftTree:BinaryTree, in rightTree:BinaryTree)

- Add/Modify
  - attachLeft(in newItem:TreeItemType) throws TreeException
  - attachLeftSubtree(in leftTree:BinaryTree) throws TreeException

- Remove
  - detachLeftSubtree():BinaryTree throws TreeException

- Ask
  - getLeftSubtree():BinaryTree throws TreeException

Show example of how to use the ADT

Complete Binary Tree

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

Indices of left/right child and parent node?

Reference Implementation

- TreeNode
  - item
  - left child
  - right child
  - parent (optional)

- Tree
  - root
  - size (optional)

Traversing a binary tree

- How to traverse a tree?
Traversing a Binary Tree

- **Pre order**
  - visit the node
  - go left
  - go right
- **Post order**
  - go left
  - go right
  - visit the node
- **In order**
  - go left
  - visit the node
  - go right
- **Level order / breadth first**
  - for \( d \geq 0 \) to height
    - visit nodes at level \( d \)

Traversal Examples

<table>
<thead>
<tr>
<th>Pre order</th>
<th>In order</th>
<th>Post order</th>
<th>Level order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B D G H C E F I</td>
<td>G D H B A E C F I</td>
<td>G H D B E I F C A</td>
<td>A B C D E F G H I</td>
</tr>
</tbody>
</table>

Traversal Implementation

- **recursive implementation of preorder**
  - The steps:
    - visit node
    - preorder(left child)
    - preorder(right child)
  - base case?

```java
void preorder(TreeNode<T> node) {
    doSomething(node);
    preorder(node.getLeft());
    preorder(node.getRight());
}
```

Implementing Traversal with Iterators

- Use a queue to order the nodes according to the type of traversal.
- Initialize iterator by type (pre, post or in) and enqueue all nodes in order necessary for traversal
- dequeue in next operation

LevelOrder Algorithm

- Use a queue to track unvisited nodes
- For each node that is dequeued,
  - enqueue each of its children
  - until queue empty
- Also called: breadth first traversal

```
<table>
<thead>
<tr>
<th>Step</th>
<th>Queue</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[B,E]</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>[C,D,E,F,G,H]</td>
<td>A B</td>
</tr>
<tr>
<td>3</td>
<td>[D,E,F,G,H]</td>
<td>A B C</td>
</tr>
<tr>
<td>4</td>
<td>[E,F,G,H]</td>
<td>A B C D</td>
</tr>
<tr>
<td>5</td>
<td>[F,G,H]</td>
<td>A B C D E</td>
</tr>
<tr>
<td>6</td>
<td>[G,H]</td>
<td>A B C D E F</td>
</tr>
<tr>
<td>7</td>
<td>[H]</td>
<td>A B C D E F G</td>
</tr>
<tr>
<td>8</td>
<td>[]</td>
<td>A B C D E F G H</td>
</tr>
<tr>
<td>9</td>
<td>[]</td>
<td>A B C D E F G H I</td>
</tr>
</tbody>
</table>
```
Categories of Data Structures

- Position-oriented data structures: access is by position.
- Value-oriented structures: access is by value.
- Examples?

Binary Search Trees

- **Definition:** 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

(Binary Tree Example)

Valid  Not Valid  Valid

BST

- **Organization**
  - the sequence of adding and removing influences the shape of the tree
- **Search / Retrieval**
  - Using inorder traversal
  - On a search key

BST ADT

- `insert(item: TreeItemType)`
  - inserts `newItem` into a BST whose items have distinct search keys that differ from `newItem`
- `delete(searchKey: KeyType)` throws `TreeException`
  - Deletes the item whose search key equals `searchKey`. If none exists, the operation fails.
- `retrieve(searchKey: KeyType): TreeItemType`
  - Returns the item whose search key equals `searchKey`. Returns null if not found.

BST - Search

- compare value with node
  - empty: not found
  - ==: found
  - <: search in the left sub-tree
  - >: search in the right sub-tree

(Binary Tree Example)

Why doesn’t BinarySearchTree extend Binary Tree?
BST – Insert

- Always add as a leaf – in the position where the search method would look for it
- Find leaf location
  - `<` : add to the left sub-tree
  - `>` : add to the right sub-tree
- Special Cases:
  - already there
  - empty tree

Inserting an item

```c
insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
// Inserts newItem into the binary search tree of which
// treeNode is the root
if (treeNode is null) {
    create new node with newItem as data
    return new node
} else if (newItem.getKey() < treeNode.getItem().getKey()) {
    treeNode.setLeft(insertItem(treeNode.getLeft(), newItem))
    return treeNode
} else {
    treeNode.setRight(insertItem(treeNode.getRight(), newItem))
    return treeNode
}
```

How is insertItem used in the code?
BST – Insert

```java
if (treeNode is null) {
    create new node with newItem as data
    return new node
}
newItem.getKey() <- 6
```

```
treeNode.setRight(insertItem(treeNode.getRight(), newItem))
return treeNode
```

Delete: Cases to Consider

- Delete something that is not there
  - Throw exception
- Delete a leaf
  - Easy, just set link from parent to null
- Delete a node with one child
- Delete a node with two children

Delete

- Case 1: one child
  - delete(5)
- Case 2: Left child has no right child
  - left child becomes root
    - (or vice-versa for right child with no left child)
  - delete(5)
- Case 2: two children
  - Strategy: replace node with a node that is easier to remove!
Digression: inorder traversal of BST

- In order:
  - go left
  - visit the node
  - go right
- The keys of an inorder traversal of a BST are in sorted order!

Delete Case 2: two children

Delete Pseudo Code I

Delete Pseudo Code II

Delete Pseudo Code III

Complexity of BST Operations
Tree Sort

- Uses the binary search tree ADT to sort an array of records according to search-key
- Efficiency
  - Average case: O(n * log n)
  - Worst case: O(n²)