Trees

Lecture 9
CS2110 – Fall 2008
Announcements

• A3 will be up shortly – check the website

• A2 is graded
  ▪ Submit regrades online
  ▪ Regrades accepted until 10/3

• Please include Cornell netId in email correspondence
  ▪ e.g., dancingGurl47@gmail.com does not help us
Tree Overview

- **Tree**: recursive data structure (similar to list)
  - Each cell may have zero or more successors (children)
  - Each cell has exactly one predecessor (parent) except the root, which has none
  - All cells are reachable from root

- **Binary tree**: tree in which each cell can have at most two children: a left child and a right child
Tree Terminology

- M is the root of this tree
- G is the root of the left subtree of M
- B, H, J, N, and S are leaves
- N is the left child of P; S is the right child
- P is the parent of N
- M and G are ancestors of D
- P, N, and S are descendants of W
- Node J is at depth 2 (i.e., depth = length of path from root = number of edges)
- Node W is at height 2 (i.e., height = length of longest path to a leaf)
- A collection of several trees is called a ...?
class TreeCell<T> {
    private T datum;
    private TreeCell<T> left, right;

    public TreeCell(T x) { datum = x; }
    public TreeCell(T x, TreeCell<T> lft, TreeCell<T> rgt) {
        datum = x;
        left = lft;
        right = rgt;
    }
    more methods: getDatum, setDatum, getLeft, setLeft, getRight, setRight
}

... new TreeCell<String>("hello") ...
Class for General Trees

class GTreeCell {
    private Object datum;
    private GTreeCell left;
    private GTreeCell sibling;

    appropriate getter and setter methods
}

• Parent node points directly only to its leftmost child
• Leftmost child has pointer to next sibling, which points to next sibling, etc.
Applications of Trees

- Most languages (natural and computer) have a recursive, hierarchical structure.
- This structure is *implicit* in ordinary textual representation.
- Recursive structure can be made *explicit* by representing sentences in the language as trees: Abstract Syntax Trees (ASTs).
- ASTs are easier to optimize, generate code from, etc. than textual representation.
- A parser converts textual representations to AST.
Example

• Expression grammar:
  \[ E \rightarrow \text{integer} \]
  \[ E \rightarrow (E + E) \]

• In textual representation
  ▪ Parentheses show hierarchical structure

• In tree representation
  ▪ Hierarchy is explicit in the structure of the tree
Recursion on Trees

- Recursive methods can be written to operate on trees in an obvious way

- Base case
  - empty tree
  - leaf node

- Recursive case
  - solve problem on left and right subtrees
  - put solutions together to get solution for full tree
Searching in a Binary Tree

```java
public static boolean treeSearch(Object x, TreeCell node) {
    if (node == null) return false;
    if (node.datum.equals(x)) return true;
    return treeSearch(x, node.left) ||
           treeSearch(x, node.right);
}
```

- Analog of linear search in lists: given tree and an object, find out if object is stored in tree
- Easy to write recursively, harder to write iteratively
Binary Search Tree (BST)

- If the tree data are ordered – in any subtree,
  - All left descendents of node come before node
  - All right descendents of node come after node
- This makes it much faster to search

```java
public static boolean treeSearch (Object x, TreeCell node) {
    if (node == null) return false;
    if (node.datum.equals(x)) return true;
    if (node.datum.compareTo(x) > 0)
        return treeSearch(x, node.left);
    else return treeSearch(x, node.right);
}
```
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree

- This can be done using either recursion or iteration

- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree

- This can be done using either recursion or iteration

- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*
Building a BST

- To insert a new item
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree

- This can be done using either recursion or iteration

- Example
  - Tree uses *alphabetical order*
  - Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
Building a BST

• To insert a new item
  ▪ Pretend to look for the item
  ▪ Put the new node in the place where you fall off the tree

• This can be done using either recursion or iteration

• Example
  ▪ Tree uses *alphabetical order*
  ▪ Months appear for insertion in *calendar order*
What Can Go Wrong?

• A BST makes searches very fast, *unless*…
  ▪ Nodes are inserted in alphabetical order
  ▪ In this case, we’re basically building a linked list (with some extra wasted space for the *left* fields that aren’t being used)

• BST works great if data arrives in random order
Because of the ordering rules for a BST, it’s easy to print the items in alphabetical order:

- Recursively print everything in the left subtree
- Print the node
- Recursively print everything in the right subtree

```java
/**
 * Show the contents of the BST in alphabetical order.
 */
public void show () {
    show(root);
    System.out.println();
}

private static void show(TreeNode node) {
    if (node == null) return;
    show(node.lchild);
    System.out.print(node.datum + " ");
    show(node.rchild);
}
```
Tree Traversals

• “Walking” over the whole tree is a *tree traversal*
  ▪ This is done often enough that there are standard names
  ▪ The previous example is an *inorder traversal*
    ▪ Process left subtree
    ▪ Process node
    ▪ Process right subtree

• Note: we’re using this for printing, but any kind of processing can be done

• There are other standard kinds of traversals
  ▪ Preorder traversal
    ▪ Process node
    ▪ Process left subtree
    ▪ Process right subtree
  ▪ Postorder traversal
    ▪ Process left subtree
    ▪ Process right subtree
    ▪ Process node
  ▪ Level-order traversal
    ▪ Not recursive
    ▪ Uses a queue
Some Useful Methods

//determine if a node is a leaf
public static boolean isLeaf(TreeCell node) {
    return (node != null) && (node.left == null)
        && (node.right == null);
}

//compute height of tree using postorder traversal
public static int height(TreeCell node) {
    if (node == null) return -1; //empty tree
    if (isLeaf(node)) return 0;
    return 1 + Math.max(height(node.left),
        height(node.right));
}

//compute number of nodes using postorder traversal
public static int nNodes(TreeCell node) {
    if (node == null) return 0;
    return 1 + nNodes(node.left) + nNodes(node.right);
Useful Facts about Binary Trees

- \(2^d\) = maximum number of nodes at depth \(d\)

- If height of tree is \(h\)
  - Minimum number of nodes in tree = \(h + 1\)
  - Maximum number of nodes in tree = \(2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1\)

- Complete binary tree
  - All levels of tree down to a certain depth are completely filled

\[
\begin{align*}
\text{depth} \\
0 & \quad \quad \quad \quad 5 \\
1 & \quad \quad \quad 4 \quad \quad 2 \\
2 & \quad 7 \quad 8 \quad 0 \quad 4 \\
\end{align*}
\]

Height 2, maximum number of nodes

\[
\begin{align*}
\text{height} \\
5 \quad 2 \\
\quad \quad \quad 4 \\
\end{align*}
\]

Height 2, minimum number of nodes
Tree with Parent Pointers

- In some applications, it is useful to have trees in which nodes can reference their parents
- Analog of doubly-linked lists
Things to Think About

• What if we want to \textit{delete} data from a BST?

• A BST works great as long as it’s \textit{balanced}
  ▪ How can we keep it balanced?
Suffix Trees

• Given a string $s$, a suffix tree for $s$ is a tree such that

  • each edge has a unique label, which is a nonnull substring of $s$
  • any two edges out of the same node have labels beginning with different characters
  • the labels along any path from the root to a leaf concatenate together to give a suffix of $s$
  • all suffixes are represented by some path
  • the leaf of the path is labeled with the index of the first character of the suffix in $s$

• Suffix trees can be constructed in linear time
Suffix Trees

abra\{cadabra\$}

dabra$

\{cadabra\$

bra

dabra$

ra

cadabra$

cadabra$

cadabra$

cadabra$

cadabra$

abr\{cadabra\$
Suffix Trees

- Useful in string matching algorithms (e.g., longest common substring of 2 strings)
- Most algorithms linear time
- Used in genomics (human genome is ~4GB)
Huffman Trees

Fixed length encoding
197*2 + 63*2 + 40*2 + 26*2 = 652

Huffman encoding
197*1 + 63*2 + 40*3 + 26*3 = 521
Huffman Compression of “Ulysses”

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
<th>Code Length</th>
<th>Code</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>242125</td>
<td>3</td>
<td>110</td>
<td>3</td>
</tr>
<tr>
<td>'e'</td>
<td>139496</td>
<td>3</td>
<td>000</td>
<td>3</td>
</tr>
<tr>
<td>'t'</td>
<td>95660</td>
<td>4</td>
<td>1010</td>
<td>4</td>
</tr>
<tr>
<td>'a'</td>
<td>89651</td>
<td>4</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>'o'</td>
<td>88884</td>
<td>4</td>
<td>0111</td>
<td>4</td>
</tr>
<tr>
<td>'n'</td>
<td>78465</td>
<td>4</td>
<td>0101</td>
<td>4</td>
</tr>
<tr>
<td>'i'</td>
<td>76505</td>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>'s'</td>
<td>73186</td>
<td>4</td>
<td>0011</td>
<td>4</td>
</tr>
<tr>
<td>'h'</td>
<td>68625</td>
<td>5</td>
<td>11111</td>
<td>5</td>
</tr>
<tr>
<td>'r'</td>
<td>68320</td>
<td>5</td>
<td>11110</td>
<td>5</td>
</tr>
<tr>
<td>'l'</td>
<td>52657</td>
<td>5</td>
<td>10111</td>
<td>5</td>
</tr>
<tr>
<td>'u'</td>
<td>32942</td>
<td>6</td>
<td>111011</td>
<td>6</td>
</tr>
<tr>
<td>'g'</td>
<td>26201</td>
<td>6</td>
<td>101101</td>
<td>6</td>
</tr>
<tr>
<td>'f'</td>
<td>25248</td>
<td>6</td>
<td>101100</td>
<td>6</td>
</tr>
<tr>
<td>'.'</td>
<td>21361</td>
<td>6</td>
<td>011010</td>
<td>6</td>
</tr>
<tr>
<td>'p'</td>
<td>20661</td>
<td>6</td>
<td>011001</td>
<td>6</td>
</tr>
</tbody>
</table>

... 

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
<th>Code Length</th>
<th>Code</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>'7'</td>
<td>68</td>
<td>15</td>
<td>111010101001111</td>
<td>15</td>
</tr>
<tr>
<td>'/'</td>
<td>58</td>
<td>15</td>
<td>111010101001110</td>
<td>15</td>
</tr>
<tr>
<td>'X'</td>
<td>19</td>
<td>16</td>
<td>0110000000100011</td>
<td>16</td>
</tr>
<tr>
<td>'&amp;'</td>
<td>3</td>
<td>18</td>
<td>0110000000100010</td>
<td>18</td>
</tr>
<tr>
<td>'%'</td>
<td>3</td>
<td>19</td>
<td>01100000000100011</td>
<td>19</td>
</tr>
<tr>
<td>'+'</td>
<td>2</td>
<td>19</td>
<td>01100000000100010</td>
<td>19</td>
</tr>
</tbody>
</table>

original size   11904320  
compressed size   6822151  
42.7% compression
BSP Trees

- BSP = Binary Space Partition
- Used to render 3D images composed of polygons
- Each node \( n \) has one polygon \( p \) as data
- Left subtree of \( n \) contains all polygons on one side of \( p \)
- Right subtree of \( n \) contains all polygons on the other side of \( p \)
- Order of traversal determines occlusion!
Tree Summary

- A tree is a recursive data structure
  - Each cell has 0 or more successors (children)
  - Each cell except the root has at exactly one predecessor (parent)
  - All cells are reachable from the root
  - A cell with no children is called a leaf

- Special case: binary tree
  - Binary tree cells have a left and a right child
  - Either or both children can be null

- Trees are useful for exposing the recursive structure of natural language and computer programs