Lecture 9: Trees

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
  - Cells without children are called leaves
  - 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, I, N, and S are leaves
- N is the left child of P; S is the right child
- P is the parent of N
- G and W are siblings
- M and G are ancestors of O
- 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 for Binary Tree Cells

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

Class for General Trees

```java
class GTreeCell {
    private Object datum;
    private GTreeCell left;
    private GTreeCell sibling;
    appropriate getter and setter methods
}
```

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
  - $(2 + 3)$

- 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;
    else if (node.datum.compareTo(x) > 0) return treeSearch(x, node.left);
    else return 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

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 (i.e. jan, feb, mar, apr, may, jun, jul, ...)

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)
  - Maximally high tree \rightarrow search just as slow as for linked list.
- BST works great if data arrives in random order
Printing Contents of BST

- 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

```
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);
}
```

Output: apr Feb jan jun mar may

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
```

Reading and Writing Trees

- Write t to file in pre-order:
  if t==null THEN
  print null
  ELSE
  Print root
  Recurse left subtree
  Recurse right subtree

- Read from file in pre-order:
  next_token = read
  IF next_token == null THEN
  return null
  ELSE
  root = next_token
  left = Recurse left subtree
  right = Recurse right subtree
  return new TreeCell(root,left,right)

```
// 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 = \(2^h + 2^{h-1} + \ldots + 2^0 = 2^{h+1} - 1\)
- Complete binary tree
  - All levels of tree down to a certain depth are completely filled

```
Useful Methods
```

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 delete data from a BST?

• A BST works great as long as it’s 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 non-null 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

• 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 Compression of “Ulysses”

<table>
<thead>
<tr>
<th>Char</th>
<th>Code</th>
<th>ascii</th>
<th>Bits and Huffman code</th>
</tr>
</thead>
<tbody>
<tr>
<td>' '</td>
<td>000</td>
<td>3</td>
<td>000</td>
</tr>
<tr>
<td>'a'</td>
<td>001</td>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>'b'</td>
<td>010</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>'c'</td>
<td>011</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>'d'</td>
<td>100</td>
<td>4</td>
<td>111</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
<td>5</td>
<td>111</td>
</tr>
<tr>
<td>'f'</td>
<td>110</td>
<td>6</td>
<td>111</td>
</tr>
<tr>
<td>'g'</td>
<td>111</td>
<td>7</td>
<td>111</td>
</tr>
<tr>
<td>'h'</td>
<td>000</td>
<td>8</td>
<td>111</td>
</tr>
<tr>
<td>'i'</td>
<td>001</td>
<td>9</td>
<td>111</td>
</tr>
<tr>
<td>'j'</td>
<td>010</td>
<td>10</td>
<td>111</td>
</tr>
<tr>
<td>'k'</td>
<td>011</td>
<td>11</td>
<td>111</td>
</tr>
<tr>
<td>'l'</td>
<td>100</td>
<td>12</td>
<td>111</td>
</tr>
<tr>
<td>'m'</td>
<td>101</td>
<td>13</td>
<td>111</td>
</tr>
<tr>
<td>'n'</td>
<td>110</td>
<td>14</td>
<td>111</td>
</tr>
<tr>
<td>'o'</td>
<td>111</td>
<td>15</td>
<td>111</td>
</tr>
<tr>
<td>'p'</td>
<td>000</td>
<td>16</td>
<td>111</td>
</tr>
<tr>
<td>'q'</td>
<td>001</td>
<td>17</td>
<td>111</td>
</tr>
<tr>
<td>'r'</td>
<td>010</td>
<td>18</td>
<td>111</td>
</tr>
<tr>
<td>'s'</td>
<td>011</td>
<td>19</td>
<td>111</td>
</tr>
<tr>
<td>'t'</td>
<td>100</td>
<td>20</td>
<td>111</td>
</tr>
<tr>
<td>'u'</td>
<td>101</td>
<td>21</td>
<td>111</td>
</tr>
<tr>
<td>'v'</td>
<td>110</td>
<td>22</td>
<td>111</td>
</tr>
<tr>
<td>'w'</td>
<td>111</td>
<td>23</td>
<td>111</td>
</tr>
<tr>
<td>'x'</td>
<td>000</td>
<td>24</td>
<td>111</td>
</tr>
<tr>
<td>'y'</td>
<td>001</td>
<td>25</td>
<td>111</td>
</tr>
<tr>
<td>'z'</td>
<td>010</td>
<td>26</td>
<td>111</td>
</tr>
</tbody>
</table>

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

Huffman encoding
197*1 + 63*2 + 40*3 + 26*3 = 521 bits

Original size: 31049900
Compressed size: 462351
42.7% compression
**Decision Trees**

- Classification:
  - Attributes (e.g. is CC used more than 200 miles from home?)
  - Values (e.g. yes/no)
  - Follow branch of tree based on value of attribute.
  - Leaves provide decision.

- Example:
  - Should credit card transaction be denied?
    - Remote Use?
      - yes
        - Freq Trav?
          - yes
            - Total?
              - yes
              - no
            - no
        - no
      - no
    - Freq Trav?
      - yes
      - > $10,000?
        - yes
        - Deny
        - no
        - Deny
      - no
      - Deny

**BSP Trees**

- BSP = Binary Space Partition
  - Used to render 3D images composed of polygons (see demo)
  - 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

- Paint image from back to front. 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