Trees

Lecture 8
CS2110 – Fall 2011

Tree Overview
- Tree: recursive data structure (similar to list)
  - Each cell may have two or more successors (or children)
  - Each cell has at most one predecessor (or parent)
  - Distinguished cell called root has no parent
  - All cells reachable from root
- Binary tree: tree in which each cell can have at most two children: a left child and a right child

Class for Binary Tree Cells
```java
class TreeCell<T> {
    private T datum;
    private TreeCell<T> left, right;

    public TreeCell(T x) {
        datum = x;
    }

    public TreeCell(T x, TreeCell<T> l, TreeCell<T> r) {
        datum = x;
        left = l;
        right = r;
    }

    more methods: getDatum, setDatum, getLeft, setLeft, getRight, setRight
}
```

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

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

Class for Binary Tree Cells
```java
class TreeCell<T> {
    private T datum;
    private TreeCell<T> left, right;

    public TreeCell(T x) {
        datum = x;
    }

    public TreeCell(T x, TreeCell<T> l, TreeCell<T> r) {
        datum = x;
        left = l;
        right = r;
    }

    more methods: getDatum, setDatum, getLeft, setLeft, getRight, setRight
}
```

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

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

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

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

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

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

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

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

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

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

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
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

- 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

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
Printing Contents of BST

- Because of the ordering rules for a BST, it is 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

Some Useful Methods

```java
//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 = $2^0 + 2^1 + ... + 2^{h-1}$
  - Maximum number of nodes in tree = $2^h - 1$
- Complete binary tree
  - All levels of tree down to a certain depth are completely filled

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 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

Huffman 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)

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