CS 61B Data Structures and Programming Methodology

July 15, 2008

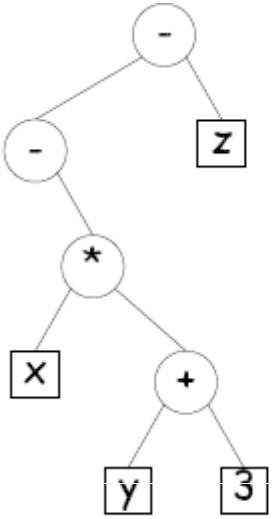
David Sun

Announcements

- Project 2 spec is out.
  - You can work individually or a team of two.
  - Due 7/28.
- Midter1 Regrades:

  If you believe we misgraded questions on a midterm, return the paper to me (or your TA) with a written note of the problem on a separate piece of paper. Upon receiving a regrade request, the entire exam will be regraded, so be sure to check the solution to confirm that you will not loose more points, which has happened in the past.

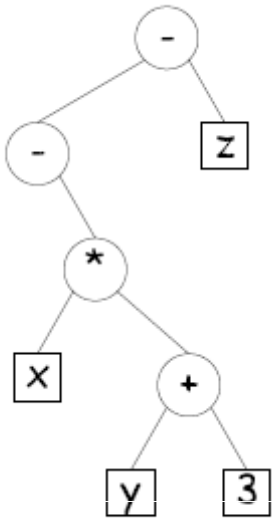
Inorder Traversal and Infix Expression



$\Rightarrow$ ((-(x*(y+3)))-z)

```
static String toInfix (Tree<String> T) {
  if (T == null)
    return "";
  if (T.degree () == 0)
    return T.label ();
  else {
    return String.format ("(%s%s%s)",
        toInfix (T.left ()), T.label (), toInfix (T.right ())
  }
}
```

Preorder Traversal and Prefix Expression



$\Rightarrow$ (- (- (* x (+ y 3))) z)

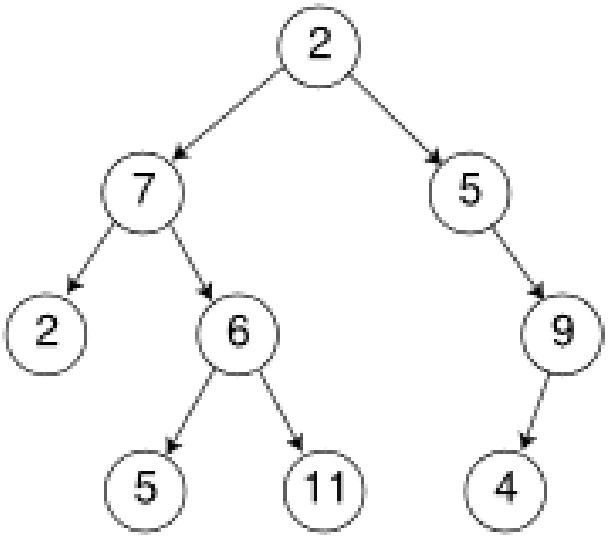
```
static String toLisp (Tree<String> T) {
        if (T == null)
                return "";
        else if (T.degree () == 0)
                return T.label ();
        else {
                String R; R = "";
                for (int i = 0; i < T.numChildren (); i += 1)
                        R += " " + toLisp (T.child (i));
                        return String.format ("(%s%s)", T.label (), R);
        }
}
```

Time

- Tree traversal is linear: O*(N)*, where *N* is the # of nodes.
  - there is one visit at the root, and
  - one visit for every edge in the tree
  - since every node but the root has exactly one parent, and the root has none, must be *N − 1* edges in any non-empty tree.
- For *k*-ary tree (max # children is *k*), $h + 1 \leq N \leq \frac{k^{h+1}-1}{k-1}$, where *h* is height.
- So $h \in \Omega(\log_k N) = \Omega(\lg N)$ and $h \in O(N)$
- Many tree algorithms look at one child only. For them, time is proportional to the height of the tree, and this is $\Theta(\lg N)$ assuming that tree is bushy—each level has about as many nodes as possible.

Binary Tree

- A binary tree is a tree in which no node has more than two children, and every child is either a ***left child*** or a ***right child***, even if it's the only child its parent has.



Representing Binary Trees

- Each tree node has three references to neighboring tree nodes: a "parent" reference, and "left" and "right" references for the two children. Each node also has an "item" reference.

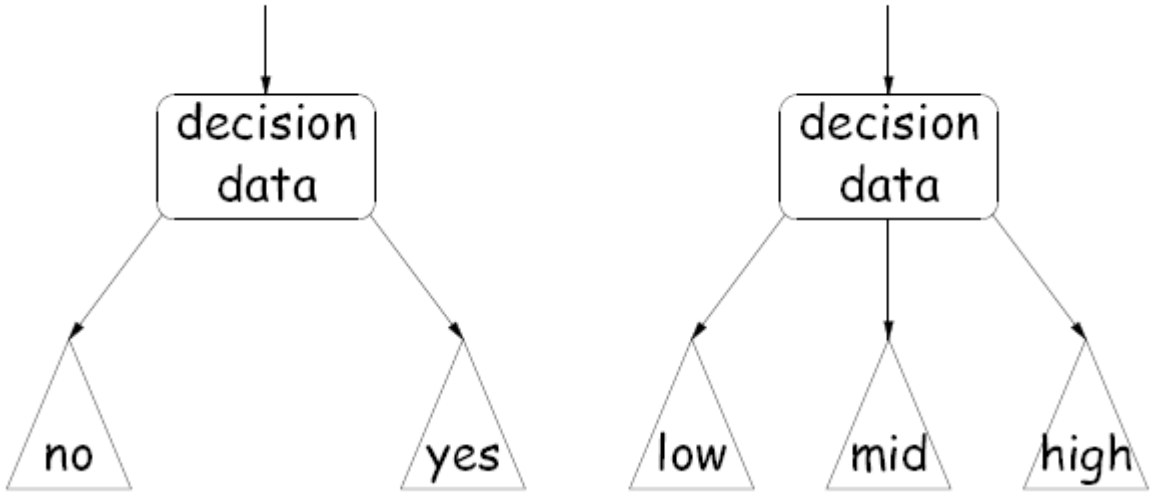
```
class BinaryTree{
    BinaryTreeNode root;
    int size;
}

class BinaryTreeNode{
    Object item;
    SibTreeNode parent;
    SibTreeNode left;
    SibTreeNode right;
}

public void inorder() {
   if (left != null) {
        left.inorder();
   }
   this.visit();
   if (right != null) {
        right.inorder();
   }
}
```

Divide and Conquer

- Much computation is devoted to finding things in response to various forms of query.
- Linear search for response can be expensive, especially when data set is too large for primary memory.
- Preferable to have criteria for dividing data to be searched into pieces recursively
- Tree is a natural framework for the representation:



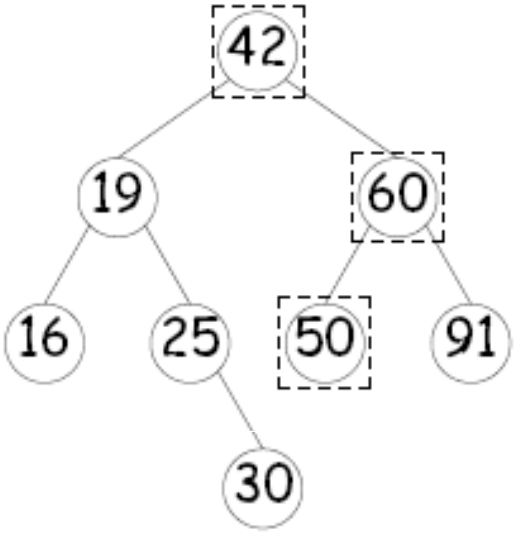
Binary Search Tree

- Binary Search Tree is a binary tree.
- Generally, each node of the Binary Search Tree contains an <Key, Value> pair called an Entry.
  - The key can be the same as the value.
- Binary Search Tree satisfies the following property, called the *binary search tree invariant*: For any node X,
  - every key in the *left* subtree of X is *less than or equal* to X's key, and
  - every key in the *right* subtree of X is *greater than or equal* to X's key.
  - A key equal to the parent's key can go into either subtree.
- Example.

Finding

```
public Entry find (Comparable aKey) {
      BinaryTreeNode T = findHelper(root, aKey);
      if (T == null)
           return null;
      else
           return T.entry;
}
private static BinaryTreeNode findHelper(BinaryTreeNode T,
           Comparable aKey){
      if (T == null)
           return T;
      //compare the Key with the current node
      int comp = aKey.compareTo(T.entry.key());
      //aKey is smaller, look in the right subtree
      if (comp < 0)
           return find(T.left, aKey);
      //aKey is larger, look in the right subtree
      else if (comp < 0)
           return find(T.right, aKey);
      else
           //return when find a match
           return T;
}
```

Search for key = 50:



- *Dashed boxes show which node labels we look at.*
- *Number looked at proportional to height of tree.*

Iterative Version

```
public Entry find(Comparable aKey) {
    //start at the root
    BinaryTreeNode T = root;
    while (node != null) {
          //compare the Key with the current node
          int comp = aKey.compareTo(T.entry.key);
          //aKey is smaller, look in the left subtree
          if (comp < 0)
                    node = node.left;
          //aKey is larger, look in the right subtree
          else if (comp > 0)
                    node = node.right;
          //Stop and return when find a match
          else
                    return node.entry;
    }
    //Return null on failure
    return null;
}
```

Finding

- What if we want to find the smallest key greater than or equal to k, or the largest key less than or equal to k?
- When searching downward through the tree for a key k that is not in the tree, we are certain to encounter both
  - the node containing the smallest key greater than k (if any key is greater)
  - the node containing the largest key less than k (if any key is less).
  - Implement a method smallestKeyNotSmaller(k):
    - Use a variable to track the smallest key not smaller than k found so far.
    - Pretend you are looking for key k, if you find it return immediately
    - When you encounter a null pointer, return the value of the variable

First and Last

- Entry first() – find the smallest key in the binary tree
  - If the tree is empty, return null. Otherwise, start at the root. Repeatedly go to the left child until you reach a node with no left child.
- Entry last() – find the largest key in the binary tree
  - If the tree is empty, return null. Repeatedly go to the right child until you reach a node with no right child.

Inserting

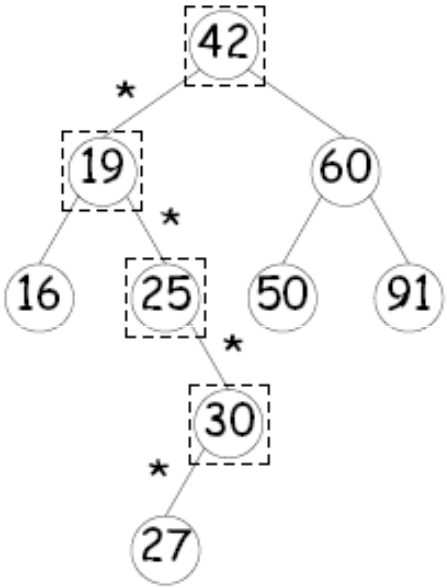
- `insert()` starts by following the same path through the tree as `find()`.
- When it reaches a `null` reference, replace the `null` with a reference to a new node <Key, Value> .
- Duplicate keys are allowed.
  - If `insert()` finds a node that already has the key, it puts it the new entry in the left subtree of the older one.
  - We could just as easily choose the right subtree; it doesn't matter.

Inserting

```
public insert(Entry e) {
   insertHelper(root, E);
}

BinaryTreeNode static insertHelper(BinaryTreeNode T, Ent
     if (T == null)
         return new BinaryTreeNode(E);
     //compare the Key with the current node
     int comp = aKey.compareTo(E.key);
     //aKey is smaller or equal, insert in the left subtr
     if (comp <= 0)
         T.left = insert(T.left, E);
     //aKey is larger, insert in the right subtree
     else //if (comp > 0)
          T.left = insert(T.right, E);
     return T;
}
```

Insert with Key = 27



- *Starred edges are set (to themselves, unless initially null).*
- *Again, time proportional to height.*
- *Try writing an iterative version.*

Reading

- Objects, Abstraction, Data Structures and Design using Java 5.0
  - Chapter 8 pp408 - 433