In our last episode

- We looked at the complexity of searching a binary tree
  - average complexity is $O(\log n)$, in most cases
- We pondered the question, “How many ways can you traverse a binary tree?”
  - and came up with some good ideas

Tree traversals

- For a tree with $n$ nodes, there are $n!$ possible traversals
- Most of these are random
  - not very sensible!
- We'll look at 2 methods
  - breadth-first
  - depth-first
- Complexity is $\Theta(n)$ (stack space is $O(n)$)

Breadth-first traversal

- Visit every node at level $i$ before visiting the nodes at level $i+1$ (or vice versa)
- Several possibilities:
  - top-down, left-to-right
  - top-down, right-to-left
  - bottom-up, left-to-right
  - bottom-up, right-to-left
- Implement with a queue
Implementation of breadth-first search

```java
public void breadthFirst() {
    BinaryTreeNode n = root;
    Queue queue = new Queue();
    if (n != null) {
        queue.enqueue(n);
        while (!queue.isEmpty()) {
            n = (BinaryTreeNode)queue.dequeue();
            n.visit();
            // if there are children, add them to the
            // end of the queue
            if (n.left != null)
                queue.enqueue(n.left);
            if (n.right != null)
                queue.enqueue(n.right);
        }
    }
}
```

Depth-first traversal

- Go down one path from top to bottom, then backtrack and repeat for all paths
- Nodes can be visited on the way down or on the way up
- 3 subtasks
  - visit node: V
  - traverse its left subtree: L
  - traverse its right subtree: R

Depth-first traversal

- 3 main types:
  - VLR: preorder traversal
  - LVR: inorder traversal
  - LRV: postorder traversal
- Initials refer to the order of operations on each node
  - e.g. VLR = “visit node, then traverse left subtree, then traverse right subtree”

Implementation of preorder traversal

```java
protected void preorder(BinaryTreeNode n) {
    if (n != null) {
        n.visit();
        preorder(n.left);
        preorder(n.right);
    }
}
```
Implementation of inorder traversal

```java
protected void inorder(BinaryTreeNode n) {
    if (n != null) {
        inorder(n.left);
        n.visit();
        inorder(n.right);
    }
}
```

Implementation of postorder traversal

```java
protected void postorder(BinaryTreeNode n) {
    if (n != null) {
        postorder(n.left);
        postorder(n.right);
        n.visit();
    }
}
```

Do we have to use recursion?

- All three traversals were defined recursively
- We could replace these with iterative methods
  - but, these methods make heavy use of stacks...
  - ...and so does recursion...
  - ...so our gains may be minimal

Do we have to use a stack when traversing a tree?

- There are ways to traverse a tree without using a stack (recursive or otherwise)
- Idea 1: each node contains references to predecessors and successors
  - Use these references to traverse the tree
- Idea 2: Alter the structure of the tree so that you have less children to visit
  - e.g., convert tree to one that has no left subtree (Morris algorithm in book)
Modifying a binary tree

- **Q:** How can we insert items in a tree, or remove items from a tree, without changing the nature of a binary tree?
- **Potential rough spots:**
  - preserving characteristics of left and right subtrees (left must be less than the node, right must be greater than the node)
  - inserting duplicates
  - deleting a parent node

Insertion

- **Idea:**
  - compare new node to current node
  - if new node is less than current node, check right subchild
  - else, check left subchild
  - if current node has no subchild,
    - insert new node as {left,right} subchild of current node
  - repeat until node is placed
- **Complexity is the same as searching**

Insertion: code

```java
public void insert(Comparable obj) {
    if (root == null)
        root = new BinaryTreeNode(obj);
    else {
        BinaryTreeNode n = root, prev = null;
        while (n != null) {
            prev = n;
            if (obj.compareTo(n.key) == 1)
                n = n.right;
            else
                n = n.left;
        }
        if (obj.compareTo(prev.key) == 1)
            prev.right = new BinaryTreeNode(obj);
        else
            prev.left = new BinaryTreeNode(obj);
    }
}
```

Note

- The insertion algorithm allows duplicates to be inserted into the tree
  - Duplicates will be left children
- This is not good practice!
- **Q:** How can we change this code to deal with duplicates?
Deletion

- Deletion is a bit more complicated than insertion
  - there is a chance you'll fragment the tree
- Complexity is proportional to the number of children the node to be deleted has

Deletion: 3 cases

- Case 1: Deleting a leaf node
  - parent's {left,right} reference is set to null, and we're done!
- Case 2: Deleting a node with one child
  - Reset parent's {left,right} reference to the child
  - child is “promoted” one level
- Case 3: Deleting a node with two children
  - most complicated case
  - several possible algorithms

Algorithm 1: Deletion by merging

- Idea:
  - Find the node with the largest value in the left subtree
    - this will be the rightmost node in the left subtree
  - Make this node the parent of the right subtree
  - Promote the left child node to the position vacated by the deleted node

Deletion by merging: code

```java
public void mergeDelete(Comparable obj) { 
    BinaryTreeNode temp, node, n = root, prev = null; 
    while (n != null && obj.compareTo(n.key) != 0) { 
        prev = n; 
        if (obj.compareTo(n.key) == 1) 
            n = n.right; 
        else 
            n = n.left; 
    } 
    node = n; 
    if (n != null && obj.compareTo(n.key) == 0) { 
        if (node.right == null) // node has no right child 
            node = node.left; // attach left node to parent 
        else if (node.left == null) // node has no left child 
            node = node.right; // attach right node to parent 
        // continued on next slide 
    }
```
Deletion by merging: code (cont.)

```java
else { // merge the subtrees
    temp = node.left;
    // find rightmost node of left subtree
    while (temp.right != null)
        temp = temp.right;
    // attach right subtree to rightmost node of left subtree
    temp.right = node.right;
    node = node.left; // original left child is now in node's
    // place
}
if (n == root)
    root = node;
else if (prev.left == n)
    prev.left = node;
else
    prev.right = node;
// continued on next slide
```

Deletion by merging: code (cont.)

```java
else if (root != null)
    System.out.println("key " + obj.toString() + " is not
        in the tree");
else
    System.out.println("tree is empty");
```

Notes

- Deletion by merging may change the height of the tree
  - increase or decrease
  - may lead to a very unbalanced tree (one subtree is heavier than the other)
- Q: Is there a way we can delete a node without changing the height of the tree?

Algorithm 2: Deletion by copying (Hibbard and Knuth)

- Idea:
  - Replace node being deleted with its immediate predecessor or successor
  - Predecessor = key in rightmost node of left subtree
  - Successor = key in leftmost node of right subtree
  - Node that held predecessor/successor is removed from the tree
  - Preserves the height of the tree
Deletion by copying: code

```java
public void copyDelete(Comparable obj) {
    BinaryTreeNode node, n=root, prev = null;
    while (n != null && obj.compareTo(n.key) != 0) {
        prev = n;
        if (obj.compareTo(n.key) == 1)
            n = n.right;
        else
            n = n.left;
    }
    node = n;
    if (n != null && obj.compareTo(n.key) == 0) {
        if (node.right == null)
            node = node.left;
        else if (node.left == null)
            node = node.right;
        else {
            BinaryTreeNode temp = node.left; // traverse left subtree
            BinaryTreeNode previous = node;
            while (temp.right != null) {
                previous = temp;
                temp = temp.right;  // move to rightmost node
            }
            node.key = temp.key;  // "copy" node here
            if (previous == node) // no right subtree
                previous.left = temp.left;
            else
                previous.right = temp.left;
        }
    }
    if (n == root)
        root = node;
    else if (prev.left == n)
        prev.left = node;
    else
        prev.right = node;
} // continued on next page
```

Deletion by copying: code (cont.)

```java
else {
    BinaryTreeNode temp = node.left; // traverse left subtree
    BinaryTreeNode previous = node;
    while (temp.right != null) {
        previous = temp;
        temp = temp.right;  // move to rightmost node
    }
    node.key = temp.key;  // "copy" node here
    if (previous == node) // no right subtree
        previous.left = temp.left;
    else
        previous.right = temp.left;
}
if (n == root)
    root = node;
else if (prev.left == n)
    prev.left = node;
else
    prev.right = node;
} // continued on next page
```

Deletion by copying: code (cont.)

```java
else if (root != null)
    System.out.println("key " + obj.toString() + " is not in the tree");
else
    System.out.println("tree is empty");
```

Notes

- Tree may still become unbalanced if we insert and delete many nodes ("asymmetric deletion")
  - workaround: alternate copy node between the predecessor and the successor ("symmetric deletion")
- Complexity: difficult to compute
  - usually computed via experiments
  - asymmetric: $O(n \log^3 n)$
  - symmetric: $O(n \log n)$