Comp 282 Review

Why do we use Binary search trees?

- Combines the advantage of two other structures an ordered array and linked list.

Slow insertion in an Ordered Array

- Search the center of the array; if the object you're looking for is greater than what you find there, you narrow your search to the bottom half. → Object in O(log N)

```
23    public void insert(int id, double data) {
24        Node newNode = new Node(id, data);
25        insert(newNode);
26    }
27
28    //Todo: Implement the Node insertion
29    public void insert(Node newNode) {
30        if (root == null) {
31            root = newNode;
32            return;
33        }
34
35        Node current = root;
36        while (current != null) {
37            if (newNode.id == current.id) {
38                //No duplicates allowed
39                throw new ArrayIndexOutOfBoundsException();
40            }
41
42            if (newNode.id < current.id) {
43                if (current.leftChild == null) {
44                    current.leftChild = newNode;
45                    break;
46                } else {
47                    current = current.leftChild;
48                }
49            } else {
50                if (current.rightChild == null) {
51                    current.rightChild = newNode;
52                    break;
53                } else {
54                    current = current.rightChild;
55                }
56            }
57        }
58    }
```

inserting the node

recursively inserting New node

Path

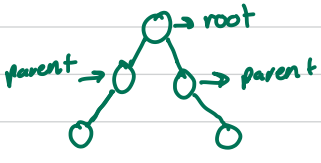
- Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a path.

Root

- The node at the top of the tree is called a root.
- There is only one root in a tree.

Parent

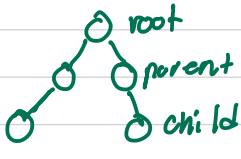
- Any node (except the root) has exactly one edge running upward to another node.
- The node above is called the parent



Child

Any node may have one or more lines running downward to other nodes.

- these nodes below a given node are called its children.



leaf

A node that has no children is called a leaf node

subtree

- Any node can be considered to be root of a subtree, which consist of its children and its children's children, and so on. Contains all its descendants

Visiting

- A node visited when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data fields or displaying it.

Traversing

To traverse a tree means to visit all the nodes in some specific order
For example you might visit all the nodes in order of ascending key values.

Levels

The levels of a particular node reference to how many generations the node is from the root.

- root = level 0 then its children will be level 1 its grandchildren will be level 2 and so on.

Keys

- key values are used to search for the item or preform other operation

Binary trees

- tree can have at most two children, the tree is called Binary tree
- The two children are called left child and Right child

Finding a node

Java code to Find a node

```
    public Node find(int key) {
        Node current = root;

        while (current != null) {
            if (key == current.id) {
                return current;
            } else if (key < current.id) {
                current = current.leftChild;
            } else {
                current = current.rightChild;
            }
        }

        return null;
    }
```

→ set the current to the root
→ while loop as long as the current =! null or empty
→ key must = the current num we will return the key
→ key < the current we will set current to left child
→ else current will be right child

→ if the key not found return null

Inserting a node

```
    public void insert(int id, double data) {
        Node newNode = new Node(id, data);
        insert(newNode);
    }
```

→ method created to add a new node.

```
    //Todo: Implement the Node insertion
    public void insert(Node newNode) {
        if (root == null) {
            root = newNode;
            return;
        }

        Node current = root;
        while (current != null) {
            if (newNode.id == current.id) {
                //No duplicates allowed
                throw new ArrayIndexOutOfBoundsException();
            }

            if (newNode.id < current.id) {
                if (current.leftChild == null) {
                    current.leftChild = newNode;
                    break;
                } else {
                    current = current.leftChild;
                }
            } else {
                if (current.rightChild == null) {
                    current.rightChild = newNode;
                    break;
                } else {
                    current = current.rightChild;
                }
            }
        }
    }
```

→ inserting new node
→ root == null or empty
→ root is the new node
→ return the root
→ set current to new node
→ while current does not equall null
→ will check if the new node equall the current if so no duplicates
→ throw an Exception
→ if the new node < current
→ current is set to left child and if it null
→ if left child is empty set new node to left child
→ End if statment
→ else if current left child is not empty we set it to current node
→ if the current right child equals null
→ current right child = new node
→ break if else
→ current = current right child.

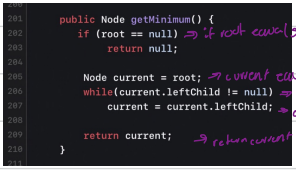
inorder traversal

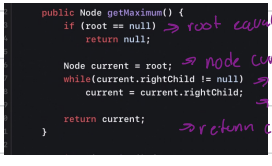
- all nodes will be visited in ascending order base on their Key values

Pre order and post order traversal

finding max and min values

- min values go to left child of root; then go left child of that child until you come to the child that has no left child this node is the min





Deleting a node

chapter 9) Red - Black Trees

Balanced and Unbalanced Trees

- During insertion balance is achieved

Red - Black Tree characteristic

0 The nodes are colored

0 During insertion and deletion, rules are followed that preserves various arrangement of those color

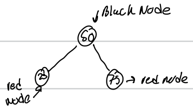
Colored Nodes

- In a red - Black, every node is either Black or red

Red - Black Rules

- Every node is either red or Black
- The root is always Black
- If a node is red, its children must be Black.
- Every path from the root to a leaf, or to a null child, must contain the same number of Black nodes

Inserting two Red Nodes



Rotation

```java
private void rotateLeft(RedBlackNode top, RedBlackNode parent) {
    if (top.rightChild == null)     → return null
        return;

    RedBlackNode newTop = top.rightChild;     → New Black Node = right child
    top.rightChild = newTop.leftChild;  //Cross over node
    newTop.leftChild = top;     //Move down
    if (top == root) {
        setRoot(newTop);
    } else {
        if (top == parent.leftChild)    //Move up
            parent.leftChild = newTop;
        else
            parent.rightChild = newTop;
    }
}

private void rotateRight(RedBlackNode top, RedBlackNode parent) {
    if (top.leftChild == null)
        return;

    RedBlackNode newTop = top.leftChild;
    top.leftChild = newTop.rightChild;      //Corss over node
    newTop.rightChild = top;        //Move down
    if (top == root) {
        setRoot(newTop);
    } else {
        if (top == parent.leftChild)    //Move up
            parent.leftChild = newTop;
        else
            parent.rightChild = newTop;
    }
}
```

Color flips

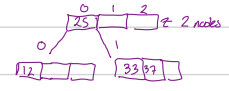
```java
private void flipColors(RedBlackNode current) {
    if (current.leftChild == null ||  current.rightChild == null)
        return;

    if (current.leftChild.isRed && current.rightChild.isRed) {
        current.leftChild.isRed = false;
        current.rightChild.isRed = false;

        if (current != root)
            current.isRed = true;
    }
}
```

Deletion

```java
boolean delete(int key) {   //We are not really deleting the RedBlackNodes (too complex). Only H
    RedBlackNode current = root;

    while (current != null && current.id != key) { //Look for RedBlackNode to be deleted
        if (key < current.id) {
            current = current.leftChild;
        } else {
            current = current.rightChild;
        }
    }

    if (current == null)
        return false;

    current.isDeleted = true;

    return true;
}
```

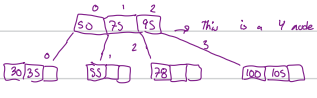
Chapter 10) 2-3-4 Trees and External Storage

- A node with one data item always has two children
- A node with two data items always has three children.
- A node with three data items always has four children

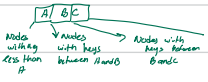


← 2 nodes → 2 nodes

← 3 nodes → This is 3 nodes



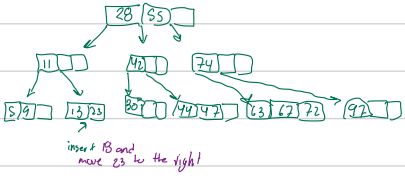
→ This is a 4 node



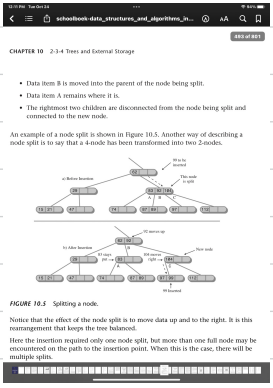
Searching 2-3-4 trees

Same as binary search you go thru every child corresponding

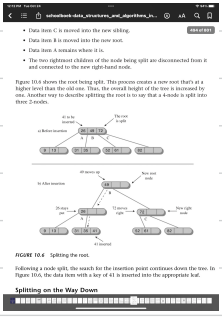
Insertion



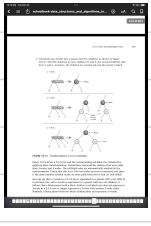
Node splits



Splitting the Root



Transforming from 2-3-4 to Red Black



External storage

```
private boolean isBalancedByNodeCount(){
int leftHand;
int rightHand;

     if(root == null){
    return true;
     }
      leftHand = height(leftChild);
      rightHand = height(rightChild);

     if(Math.abs(leftHand - rightHand) <= 1
&& leftChild.isBalancedByNodeCount() &&
rightChild.isBalancedByNodeCount()){
        return true;
    }
        return false;
   }
```

```
int height(Node node){

  if(node == null){
    return 0;
   }

  if(height(leftChild) > height(rightChild)){
     return 1 + height(leftChild);
   }
    else{
     return 1 + heiaght(rightChild);
   }
```