Binary Search Trees
The Problem with Linked Lists

- Accessing an item from a linked list takes $O(N)$ time for an arbitrary element.
- Binary trees can improve upon this and reduce access to $O(\log N)$ time for the average case.
- Expands on the binary search technique and allows insertions and deletions.
- Worst case degenerates to $O(N)$ but this can be avoided by using balanced trees (AVL, Red-Black).
Binary Search Trees

- A binary tree is a tree where each node has at most two children, referred to as the left and right child.
- A binary search tree is a binary tree in which every node's left subtree holds values less than the root, and every right subtree holds values greater than the root's value.
- A new node is added as a leaf.
Binary Search Tree has some special properties

- Each element in the left subtree of an element is less than the root element of that subtree.

- The left and right subtrees are themselves Binary Search Trees.
Binary Search Tree (example)
Not a Binary Search Tree

“60” cannot be in the right branch of 80, since it is less than 80.
Implementation of Binary Node

```java
public class BSTNode {
    private Comparable myData;
    private BSTNode myLeft;
    private BSTNode myRight;

    public BinaryNode(Comparable item) {
        myData = item;
    }

    public Object getValue() {
        return myData;
    }

    public BinaryNode getLeft() {
        return myLeft;
    }

    public BinaryNode getRight() {
        return myRight;
    }

    public void setLeft(BSTNode b) {
        myLeft = b;
    }
    // setRight not shown
}
```
A binary search tree need not be full, complete or a two-tree, but it could be any of those

- If a binary search tree is **full** or **complete**, its height is \( \log_2 n \).

- If a binary search tree is a chain, its height is **linear** in \( n \).
Building a Binary Search Tree

- Insertion strategy:
  
  - If the value of the new element is less than the value of the current node, proceed to the left subtree (left child) of the node. If null, insert the node as a left child of the parent.
  
  - If the value of the new element is greater than the value of the current node, proceed to the right subtree (right child) of the node. If null, insert the node as a left child of the parent.
Building a Binary Search Tree (continued)
Sample Insertion

- 100, 164, 130, 189, 244, 42, 141, 231, 20, 153
Worst Case Performance

- In the worst case a BST can degenerate into a singly linked list.
- Performance goes to $O(N)$
- $2 \ 3 \ 5 \ 7 \ 11 \ 13 \ 17$
public void insert(int id, double dd) {
    Node newNode = new Node(); // make new node
    newNode.iData = id; // insert data
    newNode.dData = dd;
    if(root == null) // no node in root
        root = newNode;
    else // root occupied
        Node current = root; // start at root
    Node parent;
    while(true) // (exits internally)
        { parent = current;
            if(id < current.iData) // go left?
                { current = current.leftChild;
                    if(current == null) // if end of the line,
                        { // insert on left
                            parent.leftChild = newNode;
                            return;
                        } // end if go left
                } // end else go left
            else // or go right?
                { current = current.rightChild;
                    if(current == null) // if end of the line
                        { // insert on right
                            parent.rightChild = newNode;
                            return;
                        } // end else go right
                } // end else go right
        } // end while
    } // end else n
Searching a Binary Search Tree

Beginning at the root of the tree, recursively perform the following:

```java
public Node find(int key) // find node with given key
{
    // (assumes non-empty tree)
    Node current = root; // start at root
    while(current.iData != key) // while no match,
    {
        if(key < current.iData) // go left?
            current = current.leftChild;
        else // or go right?
            current = current.rightChild;
        if(current == null) // if no child,
            return null; // didn't find it
    }
    return current; // found it
}
```
Removing elements from a Binary Search Tree

- Strategy: search for the target to be removed, and then delete it
- There are three cases to consider, where the element to be removed
  - Is a leaf
  - Has a single child
  - Has two children
Case 1: Removing a leaf element from a Binary Search Tree

- **Strategy:** search for the target to be removed, and then delete it
- **Implementation similar to search()**
  
  ```java
  when target is found, 
  current.equals(target)
  if current == parent.left
    parent.left = null
  if current == parent.right
    parent.right = null
  ```

- **This effectively removes the target object**
  - Must be a leaf!
  - **And we need a way to access the parent!**
Case 2: Removing an element with a single child from a Binary Search Tree

First search()
When target is found, current.equals(target)
if current == parent.left
    parent.left = child;
if current == parent.right
    parent.right = child;

For a single child
    child = current.left
or
    child = current.right
Case 2 Example: Removing 92
Case 3: Removing an element with two children from a Binary Search Tree

Consider removing element 92
Case 3: Removing an element with two children from a Binary Search Tree (1)

- To preserve ordering, replace the element to be removed with
  - It’s immediate predecessor
  - It’s immediate successor

- Predecessor
  - Right-most element in left subtree

- Successor (symmetric)
  - Left-most element in right subtree

- We need a way (a method) to find the predecessor or successor

Note: The left child of the left-most entry will be null, and the right child of the right-most entry will be null.
Case 3: Removing an element with two children from a Binary Search Tree (2a)

- First, copy successor element to target element
  - Assume target is 92
    - 95 replaces 92
- Next, delete “old” 95
  - Treat this as case of an element with a single child
Case 3: Removing an element with two children from a Binary Search Tree (2b)

- Alternately, copy predecessor element to target element
  - Assume target is 92
    - 87 replaces 92
- Next, delete “old” 87
  - Treat this as case of an element with no children
Case 3: Removing an element with two children from a Binary Search Tree (3)

Replace 92 with successor

Replace 92 with predecessor
private node getSuccessor(node delNode) {
    Node successorParent = delNode;
    Node successor = delNode;
    Node current = delNode.rightChild; // go to right child
    while(current != null) // until no more
    {
        // left children,
        successorParent = successor;
        successor = current;
        current = current.leftChild; // go to left child
    }
    // if successor not
    if(successor != delNode.rightChild) // right child,
    {
        // make connections
        successorParent.leftChild = successor.rightChild;
        successor.rightChild = delNode.rightChild;
    }
    return successor;
}
Finding Minimum

- For the minimum, go to the left child of the root; then go to the left child of that child, and so on, until you come to a node that has no left child.

```java
public Node minimum() // returns node with minimum key value
{
    Node current, last;
    current = root; // start at root
    while(current != null) // until the bottom,
    {
        last = current; // remember node
        current = current.leftChild; // go to left child
    }
    return last;
}
```
Performance of Binary Trees

- For the three core operations (add, access, remove) a binary search tree (BST) has an average case performance of $O(\log N)$
Finding Maximum

- For the *maximum value in the tree, follow the same procedure but go from right child to right child until you find a node with no right child. This node is the maximum.*

```java
public Node minimum() // returns node with minimum key value
{
    Node current, last;
    current = root; // start at root
    while(current != null) // until the bottom,
    {
        last = current; // remember node
        current = current.rightChild; // go to left child
    }
    return last;
}
```
Summary

- Trees consist of nodes (circles) connected by edges (lines).
- The root is the topmost node in a tree; it has no parent.
- In a binary tree, a node has at most two children.
- In a binary search tree, all the nodes that are left descendants of node A have key values less than A; all the nodes that are A's right descendants have key values greater than (or equal to) A.
- Trees perform searches, insertions, and deletions in $O(\log N)$ time.
- Nodes represent the data-objects being stored in the tree.
- Edges are most commonly represented in a program by references to a node's children (and sometimes to its parent).
- Traversing a tree means visiting all its nodes in some order.
- The simplest traversals are preorder, inorder, and postorder.
- An unbalanced tree is one whose root has many more left descendants than right descendants, or vice versa.
• Searching for a node involves comparing the value to be found with the key value of a node, and going to that node's left child if the key search value is less, or to the node's right child if the search value is greater.

• Insertion involves finding the place to insert the new node, and then changing a child field in its new parent to refer to it.

• An inorder traversal visits nodes in order of ascending keys.

• Preorder and postorder traversals are useful for parsing algebraic expressions.

• When a node has no children, it can be deleted by setting the child field in its parent to null.

• When a node has one child, it can be deleted by setting the child field in its parent to point to its child.
• When a node has two children, it can be deleted by replacing it with its successor.

• The successor to a node A can be found by finding the minimum node in the subtree whose root is A's right child.

• In a deletion of a node with two children, different situations arise, depending on whether the successor is the right child of the node to be deleted or one of the right child's left descendants.

• Nodes with duplicate key values may cause trouble in arrays because only the first one can be found in a search.

• Trees can be represented in the computer's memory as an array, although the reference-based approach is more common

• a node with a duplicate key will be inserted as the right child of its twin.