Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

Search Tree Structures

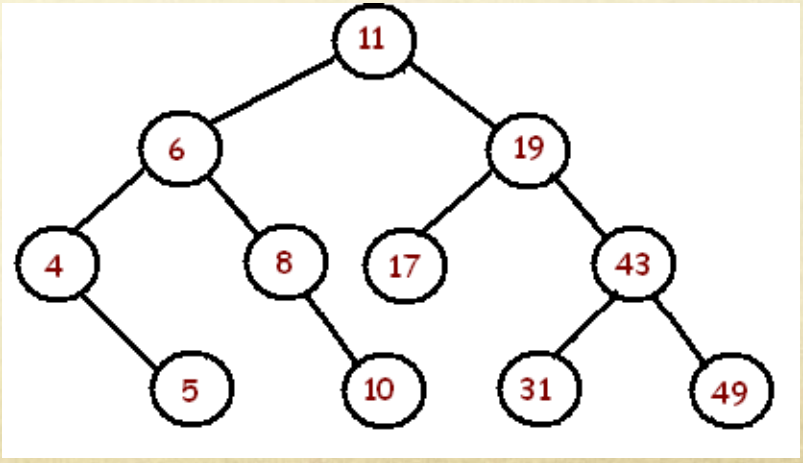
Binary Tree Operations

- Tree Traversals
  - O(n) calls to visit()
  - Why?
    - Every recursive has one call to visit() and spends O(1) time
    - Number of recursive calls = number of nodes + number of NULL pointers = n + (n+1) = O(n)
- Search
  - O(N): Tree Traversal with O(1) time for visit() to compare
- Insert and Delete
  - Needs to be defined properly and can be O(1) time

Too expensive!

Binary Search Trees

- Helps efficient search
- Binary Tree where each node stores a value
- Value stored at node is larger than all values stored in nodes of left subtree
- Value stored at node is smaller than all values stored in nodes of right subtree



https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/pix/insertEx.bmp

Implementations: BST

```
Private static class BinaryNode<AnyType> {
    AnyType element;      // data in node
    BinaryNode<AnyType> left; // left child
    BinaryNode<AnyType> right; // right child
}
```

Operations: BST

void insert(x) --> Insert x

void remove(x) --> Remove x

boolean contains(x) --> Return true if x is present

Comparable findMin( ) --> Return smallest item

Comparable findMax( ) --> Return largest item

boolean isEmpty( ) --> Return true if empty; else false

void makeEmpty( ) --> Remove all items

void printTree( ) --> Print tree in sorted order

Search: contains()

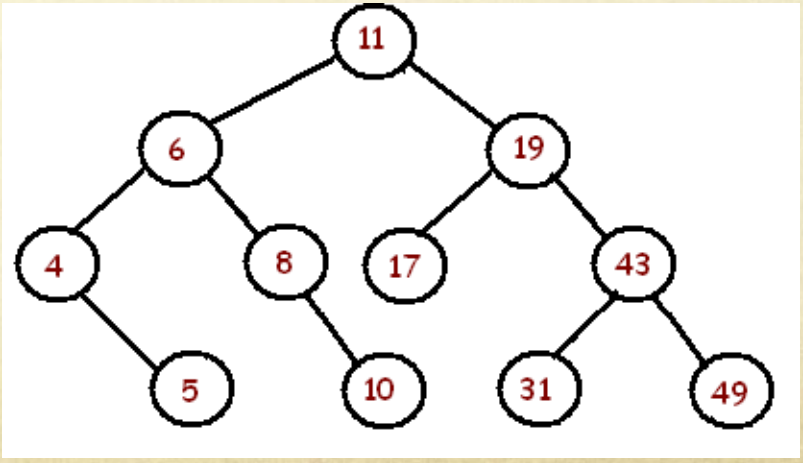
```
public boolean contains( AnyType x ) {
      return contains( x, root );
   }

private boolean contains( AnyType x, BinaryNode<AnyType> t ) {
      if( t == null ) return false;   // empty tree

      int compareResult = x.compareTo( t.element );

      if( compareResult < 0 )    // x is smaller than t.element
         return contains( x, t.left );
      else if( compareResult > 0 ) // x is larger than t.element
         return contains( x, t.right );
      else                                  // x = t.element
         return true;    // Match
   }
```

Time Complexity = O(h)

findMin() and findMax()

```
private BinaryNode<AnyType> findMin (BinaryNode<AnyType> t) {
      if( t == null )
         return null;
      else if( t.left == null )
         return t;
      return findMin( t.left );
   }
```

```
private BinaryNode<AnyType> findMax (BinaryNode<AnyType> t) {
      if( t == null )
         return null;
      else if( t.left == null )
         return t;
      return findMax( t.right);
   }
```



Time Complexity = O(h)

Insert into BST

- **Idea**:
  - First search and then insert at point of failure

```
private BinaryNode<AnyType> insert(AnyType x, BinaryNode<AnyType> t) {
    if( t == null )
        return new BinaryNode<>( x, null, null );

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = insert( x, t.left );      // Note left hand side of stmt
    else if( compareResult > 0 )
        t.right = insert( x, t.right );
    else
        ;  // Duplicate; do nothing
    return t;
}
```

Time Complexity = O(h)

Delete from BST

- **Idea**:
  - First search and then insert at point of failure

```
private BinaryNode<AnyType> remove(AnyType x, BinaryNode<AnyType> t) {
      if( t == null ) return t;   // item not found; do nothing

      int compareResult = x.compareTo( t.element );

      if( compareResult < 0 )
         t.left = remove( x, t.left );       // Note left hand side of stmt
      else if( compareResult > 0 )
         t.right = remove( x, t.right );
      else if( t.left != null && t.right != null ) { // Two children
          t.element = findMin( t.right ).element;
          t.right = remove( t.element, t.right );
      } else t = ( t.left != null ) ? t.left : t.right;

      return t;
   }
```

Time Complexity = O(h)

Static vs Dynamic Structures

Static vs Dynamic Structures

- Static data structures
  - Build data structures once and few updates
  - Lots of queries
  - E.g., Static Lists such as
- Dynamic data structures
  - Data structure constantly changing
  - Answers to queries depends on current state
  - Lots of inserts and deletes
  - E.g., Stacks and Queues

Linear Data Structures

- Lists
  - ArrayList
  - LinkedList
- Stacks & Queues
- Sorted Lists
  - List is sorted according to some key value in the data
  - Inserts and deletes must maintain sorted order
  - Search is an important operation
  - Implementations
    - ArrayList
    - LinkedList

SortedList: Time Complexity

	ArrayList	LinkedList
add(x)	O(N)	O(N)
remove(x)	O(N)	O(N)
find(x)	O(log N)	O(N)

	ArrayList	LinkedList
add(x) after find(x)	O(N)	O(1)
remove(x) after find(x)	O(N)	O(1)
find(x)	O(log N)	O(N)

Tradeoffs!

	BinaryTree	BinarySearchTree
insert(x)	O(1)	O(h)
remove(x)	O(1)	O(h)
contains(x)	O(N)	O(h)