06-0: **Ordered List ADT**

Operations:

- Insert an element in the list
- Check if an element is in the list
- Remove an element from the list
- Print out the contents of the list, in order

06-1: **Implementing Ordered List**

Using an Ordered Array – Running times:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>( \Theta(\lg n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Remove</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Print</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>

06-2: **Implementing Ordered List**

Using an Ordered Array – Running times:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>( \Theta(\lg n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Remove</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Print</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>

06-3: **Implementing Ordered List**

Using an *Unordered* Array – Running times:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Remove</td>
<td>( \Theta(n) ) (Need to find element first!</td>
</tr>
<tr>
<td>Print</td>
<td>( \Theta(n \lg n) ) (Given a fast sorting algorithm)</td>
</tr>
</tbody>
</table>

06-4: **Implementing Ordered List**

Using an *Unordered* Array – Running times:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Remove</td>
<td>( \Theta(n) ) Need to find element first!</td>
</tr>
<tr>
<td>Print</td>
<td>( \Theta(n \lg n) )</td>
</tr>
</tbody>
</table>

06-5: **Implementing Ordered List**

Using an Ordered Linked List – Running times:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Remove</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Print</td>
<td>( \Theta(n \lg n) )</td>
</tr>
</tbody>
</table>

06-6: **Implementing Ordered List**

Using an Ordered Linked List – Running times:
Check $\Theta(n)$
Insert $\Theta(n)$
Remove $\Theta(n)$
Print $\Theta(n)$

06-7: The Best of Both Worlds

- Linked Lists – Insert fast / Find slow
- Arrays – Find fast / Insert slow
- The only way to examine nth element in a linked list is to traverse (n-1) other elements

![Linked List Diagram]

- If we could leap to the middle of the list ...

06-8: The Best of Both Worlds

![Linked List Diagram]

06-9: The Best of Both Worlds

Move the initial pointer to the middle of the list:

![Linked List Diagram]

We’ve cut our search time in half! Have we changed the $\Theta()$ running time?

06-10: The Best of Both Worlds

Move the initial pointer to the middle of the list:

![Linked List Diagram]

We’ve cut our search time in half! Have we changed the $\Theta()$ running time?

Repeat the process!

06-11: The Best of Both Worlds
06-12: **The Best of Both Worlds**
Grab the first element of the list:

Give it a good shake -

06-13: **Binary Trees**
Binary Trees are Recursive Data Structures
- Base Case: Empty Tree
- Recursive Case: Node, consisting of:
  - Left Child (Tree)
  - Right Child (Tree)
  - Data

06-14: **Binary Tree Examples**
The following are all Binary Trees (Though not Binary Search Trees)

06-15: **Tree Terminology**
- Parent / Child
- Leaf node
- Root node
- Edge (between nodes)
- Path
- Ancestor / Descendant
- Depth of a node \( n \)
  - Length of path from root to \( n \)
- Height of a tree
  - (Depth of deepest node) + 1

06-16: **Full Binary Tree**
- Each node has 0 or 2 children
- Full Binary Trees

06-17: **Complete Binary Tree**
- Can be built by starting at the root, and filling the tree by levels from left to right
- Complete Binary Trees
- Not Complete Binary Trees
06-18: Binary Search Trees

- Binary Trees
- For each node n, (value stored at node n) ≥ (value stored in left subtree)
- For each node n, (value stored at node n) < (value stored in right subtree)

06-19: Example Binary Search Trees

```
        D
       / \
      C   B
     / \  / \n    A   F E
    |   |   |
   D  C  B
```

06-20: Implementing BSTs

- Each Node in a BST is implemented as a class:

```java
public class Node {
    public Comparable data;
    public Node left;
    public Node right;
}
```

06-21: Implementing BSTs

```java
public class Node {
    // Constructor
    public Node(Comparable data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }

    public Node left() {
        return left;
    }

    public Node setLeft(Node newLeft) {
        left = newLeft;
    }

    // Other methods...
}
```

06-22: Finding an Element in a BST

- Binary Search Trees are recursive data structures, so most operations on them will be recursive as well
• Recall how to write a recursive algorithm ...

06-23: **Writing a Recursive Algorithm**

• Determine a small version of the problem, which can be solved immediately. This is the *base case*
• Determine how to make the problem smaller
• Once the problem has been made smaller, we can assume that the function that we are writing *will work correctly on the smaller problem* (Recursive Leap of Faith)
  • Determine how to use the solution to the smaller problem to solve the larger problem

06-24: **Finding an Element in a BST**

• First, the Base Case – when is it easy to determine if an element is stored in a Binary Search Tree?

06-25: **Finding an Element in a BST**

• First, the Base Case – when is it easy to determine if an element is stored in a Binary Search Tree?
  • If the tree is empty, then the element can’t be there
  • If the element is stored at the root, then the element is there

06-26: **Finding an Element in a BST**

• Next, the Recursive Case – how do we make the problem smaller?

06-27: **Finding an Element in a BST**

• Next, the Recursive Case – how do we make the problem smaller?
  • Both the left and right subtrees are smaller versions of the problem. Which one do we use?

06-28: **Finding an Element in a BST**

• Next, the Recursive Case – how do we make the problem smaller?
  • Both the left and right subtrees are smaller versions of the problem. Which one do we use?
  • If the element we are trying to find is < the element stored at the root, use the left subtree. Otherwise, use the right subtree.

06-29: **Finding an Element in a BST**

• Next, the Recursive Case – how do we make the problem smaller?
  • Both the left and right subtrees are smaller versions of the problem. Which one do we use?
  • If the element we are trying to find is < the element stored at the root, use the left subtree. Otherwise, use the right subtree.
  • How do we use the solution to the subproblem to solve the original problem?

06-30: **Finding an Element in a BST**

• Next, the Recursive Case – how do we make the problem smaller?
Both the left and right subtrees are smaller versions of the problem. Which one do we use?
- If the element we are trying to find is < the element stored at the root, use the left subtree. Otherwise, use the right subtree.

How do we use the solution to the subproblem to solve the original problem?
- The solution to the subproblem is the solution to the original problem (this is not always the case in recursive algorithms)

06-31: Finding an Element in a BST
To find an element $e$ in a Binary Search Tree $T$:

- If $T$ is empty, then $e$ is not in $T$
- If the root of $T$ contains $e$, then $e$ is in $T$
- If $e < \text{the element stored in the root of } T$:
  - Look for $e$ in the left subtree of $T$
  Otherwise
  - Look for $e$ in the right subtree of $T$

06-32: Finding an Element in a BST

```java
boolean find(Node tree, Comparable elem) {
    if (tree == null)
        return false;
    if (elem.compareTo(tree.element()) == 0)
        return true;
    if (elem.compareTo(tree) < 0)
        return find(tree.left(), elem);
    else
        return find(tree.right(), elem);
}
```

06-33: Printing out a BST
To print out all element in a BST:

- Print all elements in the left subtree, in order
- Print out the element at the root of the tree
- Print all elements in the right subtree, in order

06-34: Printing out a BST
To print out all element in a BST:

- Print all elements in the left subtree, in order
- Print out the element at the root of the tree
- Print all elements in the right subtree, in order
• Each subproblem is a smaller version of the original problem – we can assume that a recursive call will work!

06-35: **Printing out a BST**

• What is the base case for printing out a Binary Search Tree – what is an easy tree to print out?

06-36: **Printing out a BST**

• What is the base case for printing out a Binary Search Tree – what is an easy tree to print out?
  • An empty tree is extremely easy to print out – do nothing!
  • Code for printing a BST ...

06-37: **Printing out a BST**

```java
void print(Node tree) {
  if (tree != null) {
    print(tree.left());
    System.out.println(tree.element());
    print(tree.right());
  }
}
```

06-38: **Printing out a BST**

Examples

06-39: **Tree Traversals**

• PREORDER Traversal
  • Do operation on root of the tree
  • Traverse left subtree
  • Traverse right subtree

• INORDER Traversal
  • Traverse left subtree
  • Do operation on root of the tree
  • Traverse right subtree

• POSTORDER Traversal
  • Traverse left subtree
  • Traverse right subtree
  • Do operation on root of the tree

06-40: **PREORDER Examples**

06-41: **POSTORDER Examples**

06-42: **INORDER Examples**

06-43: **BST Minimal Element**

To find the minimal element in a BST:
- Base Case: When is it easy to find the smallest element in a BST?
- Recursive Case: How can we make the problem smaller?
  
  How can we use the solution to the smaller problem to solve the original problem?

06-44: **BST Minimal Element**
To find the minimal element in a BST:
Base Case:

- When is it easy to find the smallest element in a BST?

06-45: **BST Minimal Element**
To find the minimal element in a BST:
Base Case:

- When is it easy to find the smallest element in a BST?
  
  - When the left subtree is empty, then the element stored at the root is the smallest element in the tree.

06-46: **BST Minimal Element**
To find the minimal element in a BST:
Recursive Case:

- How can we make the problem smaller?

06-47: **BST Minimal Element**
To find the minimal element in a BST:
Recursive Case:

- How can we make the problem smaller?
  
  - Both the left and right subtrees are smaller versions of the same problem

- How can we use the solution to a smaller problem to solve the original problem?

06-48: **BST Minimal Element**
To find the minimal element in a BST:
Recursive Case:

- How can we make the problem smaller?
  
  - Both the left and right subtrees are smaller versions of the same problem

- How can we use the solution to a smaller problem to solve the original problem?
  
  - The smallest element in the left subtree is the smallest element in the tree

06-49: **BST Minimal Element**

```java
Comparable minimum(Node tree) {
    if (tree == null)
        return null;
    if (tree.left() == null)
        return tree.element();
    else
        return minimum(tree.left());
}
```
06-50: **BST Minimal Element**

**Iterative Version**

`Comparable minimum(Node tree) {
      if (tree == null)
        return null;
      while (tree.left() != null)
        tree = tree.left();
      return tree.element();
}`

06-51: **Inserting `e` into BST `T`**

- What is the base case – an easy tree to insert an element into?

06-52: **Inserting `e` into BST `T`**

- What is the base case – an easy tree to insert an element into?
  - An empty tree
  - Create a new tree, containing the element `e`

06-53: **Inserting `e` into BST `T`**

- Recursive Case: How do we make the problem smaller?

06-54: **Inserting `e` into BST `T`**

- Recursive Case: How do we make the problem smaller?
  - The left and right subtrees are smaller versions of the same problem.
  - How do we use these smaller versions of the problem?

06-55: **Inserting `e` into BST `T`**

- Recursive Case: How do we make the problem smaller?
  - The left and right subtrees are smaller versions of the same problem
  - Insert the element into the left subtree if `e <=` value stored at the root, and insert the element into the right subtree if `e >` value stored at the root

06-56: **Inserting `e` into BST `T`**

- Base case – `T` is empty:
  - Create a new tree, containing the element `e`

- Recursive Case:
  - If `e` is less than the element at the root of `T`, insert `e` into left subtree
  - If `e` is greater than the element at the root of `T`, insert `e` into the right subtree

06-57: **Tree Manipulation in Java**
• Tree manipulation functions return trees
• Insert method takes as input the old tree and the element to insert, and returns the new tree, with the element inserted
  • Old value (pre-insertion) of tree will be destroyed
• To insert an element $e$ into a tree $T$:
  • $T = \text{insert}(T, e)$

06-58: Inserting $e$ into BST $T$

```java
Node insert(Node tree, Comparable elem) {
    if (tree == null) {
        return new Node(elem);
    }
    if (elem.compareTo(tree.element()) <= 0) {
        tree.setLeft(insert(tree.left(), elem));
        return tree;
    } else {
        tree.setRight(insert(tree.right(), elem));
        return tree;
    }
}
```

06-59: Deleting From a BST

• Removing a leaf:

06-60: Deleting From a BST

• Removing a leaf:
  • Remove element immediately

06-61: Deleting From a BST

• Removing a leaf:
  • Remove element immediately
  • Removing a node with one child:

06-62: Deleting From a BST

• Removing a leaf:
  • Remove element immediately
  • Removing a node with one child:
    • Just like removing from a linked list
    • Make parent point to child

06-63: Deleting From a BST
• Removing a leaf:
  • Remove element immediately

• Removing a node with one child:
  • Just like removing from a linked list
  • Make parent point to child

• Removing a node with two children:

06-64: **Deleting From a BST**

• Removing a leaf:
  • Remove element immediately

• Removing a node with one child:
  • Just like removing from a linked list
  • Make parent point to child

• Removing a node with two children:
  • Replace node with largest element in left subtree, or the smallest element in the right subtree

06-65: **Comparable vs. .key() method**

• We have been storing “Comparable” elements in BSTs

• Alternately, could use a “key()” method – elements stored in BSTs must implement a key() method, which returns an integer.

• We can combine the two methods
  • Each element stored in the tree has a key() method
  • key() method returns Comparable class

06-66: **BST Implementation Details**

• Use BSTs to implement Ordered List ADT

• Operations
  • Insert
  • Find
  • Remove
  • Print in Order

• The specification (interface) should not specify an implementation
  • Allow several different implementations of the same interface

06-67: **BST Implementation Details**

• BST functions require the root of the tree be sent in as a parameter
• Ordered list functions should *not* contain implementation details!

• What should we do?

06-68: BST Implementation Details

• BST functions require the root of the tree be sent in as a parameter

• Ordered list functions should *not* contain implementation details!

• What should we do?
  
  • Private variable, holds root of the tree
  
  • Private recursive methods, require root as an argument
  
  • Public methods call private methods, passing in private root