Advanced Object-oriented Programming

Lecture 27

Class Invariants:
Binary Search
Trees
Linear Search

Searching for an element in an array (‘look-up’) involves searching through each element in turn until we find the one we’re looking for.

Example class

class ArraySearch {

    private int[] values;

    public boolean isIn(int v) {
        for (int i = 0; i < values.length; i++) {
            if (values[i] == v) return true;
        }
        return false;
    }

    ...

}
Linear Search $\rightarrow$ Binary Search

The disadvantage of this is that it might take (in the worst case, where the element to be found is right at the end of the array) as many steps as there are elements in the array.

I.e., the bigger the array, the longer it takes to search the array.

We can improve look-up times by searching in a sorted array (i.e., the values in the array are increasing; or in other words, each element in the array is less than all the other elements that come after it).

The idea is that we can search through the array in much the same way as we would search through a telephone directory:

- look (roughly) in the middle;
- have we gone too far, or not far enough?
- look in the middle of the first or second half; ...
Linear Search $\rightarrow$ Binary Search

The disadvantage of this is that it might take (in the worst case, where the element to be found is right at the end of the array) as many steps as there are elements in the array. I.e., the bigger the array, the longer it takes to search the array.

We can improve look-up times by searching in a sorted array (i.e., the values in the array are increasing; or in other words, each element in the array is less than all the other elements that come after it).

The idea is that we can search through the array in much the same way as we would search through a telephone directory:

- look (roughly) in the middle;
- have we gone too far, or not far enough?
- look in the middle of the first or second half; ...
Linear Search $\rightarrow$ Binary Search

The disadvantage of this is that it might take (in the worst case, where the element to be found is right at the end of the array) as many steps as there are elements in the array.

I.e., the bigger the array, the longer it takes to search the array.

We can improve look-up times by searching in a sorted array (i.e., the values in the array are increasing; or in other words, each element in the array is less than all the other elements that come after it).

The idea is that we can search through the array in much the same way as we would search through a telephone directory:

1. look (roughly) in the middle;
2. have we gone too far, or not far enough?
3. look in the middle of the first or second half; 
   ...
Linear Search → Binary Search

The disadvantage of this is that it might take (in the worst case, where the element to be found is right at the end of the array) as many steps as there are elements in the array. I.e., the bigger the array, the longer it takes to search the array.

We can improve look-up times by searching in a sorted array (i.e., the values in the array are increasing; or in other words, each element in the array is less than all the other elements that come after it).

The idea is that we can search through the array in much the same way as we would search through a telephone directory:

- look (roughly) in the middle;
- have we gone too far, or not far enough?
- look in the middle of the first or second half; . . .
Linear Search $\rightarrow$ Binary Search

The disadvantage of this is that it might take (in the worst case, where the element to be found is right at the end of the array) as many steps as there are elements in the array.

I.e., the bigger the array, the longer it takes to search the array.

We can improve look-up times by searching in a sorted array (i.e., the values in the array are increasing; or in other words, each element in the array is less than all the other elements that come after it).

The idea is that we can search through the array in much the same way as we would search through a telephone directory:

- look (roughly) in the middle;
- have we gone too far, or not far enough?
- look in the middle of the first or second half; . . .
Linear Search $\rightarrow$ Binary Search

The disadvantage of this is that it might take (in the worst case, where the element to be found is right at the end of the array) as many steps as there are elements in the array.

I.e., the bigger the array, the longer it takes to search the array.

We can improve look-up times by searching in a sorted array (i.e., the values in the array are increasing; or in other words, each element in the array is less than all the other elements that come after it).

The idea is that we can search through the array in much the same way as we would search through a telephone directory:

- look (roughly) in the middle;
- have we gone too far, or not far enough?
- look in the middle of the first or second half; . . .
Linear Search $\rightarrow$ Binary Search

The disadvantage of this is that it might take (in the worst case, where the element to be found is right at the end of the array) as many steps as there are elements in the array. I.e., the bigger the array, the longer it takes to search the array.

We can improve look-up times by searching in a sorted array (i.e., the values in the array are increasing; or in other words, each element in the array is less than all the other elements that come after it).

The idea is that we can search through the array in much the same way as we would search through a telephone directory:

- look (roughly) in the middle;
- have we gone too far, or not far enough?
- look in the middle of the first or second half; ...
Binary Search

First, we’ll give a search method that uses a recursive helper method that looks in smaller and smaller segments of the array. Initially, the segment to search will be the entire array (from 0 to `values.length`); each recursive call will look in either the first or second half of the segment (depending on whether the middle of the segment is larger or smaller than the element we’re looking for).

```java
public boolean isIn(int v) {
    return isBetween(v, 0, values.length);
}
```

the helper method
initially, search the entire array
Binary Search

First, we’ll give a search method that uses a recursive helper method that looks in smaller and smaller segments of the array.

Initially, the segment to search will be the entire array (from 0 to `values.length`); each recursive call will look in either the first or second half of the segment (depending on whether the middle of the segment is larger or smaller than the element we’re looking for)

```java
public boolean isIn(int v) {
    return isBetween(v, 0, values.length);
}
```

the helper method
initially, search the entire array
Binary Search

First, we’ll give a search method that uses a recursive helper method that looks in smaller and smaller segments of the array.

Initially, the segment to search will be the entire array (from 0 to `values.length`); each recursive call will look in either the first or second half of the segment (depending on whether the middle of the segment is larger or smaller than the element we’re looking for)

```
public boolean isIn(int v) {
    return isBetween(v, 0, values.length);
}
```

a better search routine

the helper method

initially, search the entire array
Binary Search

First, we’ll give a search method that uses a recursive helper method that looks in smaller and smaller segments of the array.

Initially, the segment to search will be the entire array (from 0 to `values.length`); each recursive call will look in either the first or second half of the segment (depending on whether the middle of the segment is larger or smaller than the element we’re looking for).

```
public boolean isIn(int v) {
    return isBetween(v, 0, values.length);
}
```

the helper method

initially, search the entire array
The specification of the helper method:

```java
/**
 * Find a given element within a segment of a sorted array.
 *
 * @param v the element to find
 * @param l the first index of the segment
 * @param u the upper bound on the segment
 * @return true if v is in the segment values[l..u]
 */

boolean isBetween(int v, int l, int u) { ... }
```
boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2;  // roughly the middle
    if (v == values[i]) {
        return true;  // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else {  // values[i] < v
        return isBetween(v, i+1, u);
    }
}

nothing can be in the "empty" segment
too far; if v is in the array, it must be in the segment values[l..i]
not far enough: can only be in values[i+1..u]
boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2; // roughly the middle
    if (v == values[i]) {
        return true; // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else {
        return isBetween(v, i+1, u);
    }
}

nothing can be in the "empty" segment too far; if v is in the array, it must be in the segment values[l..i]
not far enough: can only be in values[i+1..u]
boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2;  // roughly the middle
    if (v == values[i]) {
        return true;  // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else {  // values[i] < v
        return isBetween(v, i+1, u);
    }
}

nothing can be in the "empty" segment too far; if v is in the array, it must be in the segment values[l..i]
not far enough: can only be in values[i+1..u]
boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2; // roughly the middle
    if (v == values[i]) {
        return true; // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else { // values[i] < v
        return isBetween(v, i+1, u);
    }
}

nothing can be in the "empty" segment
too far; if v is in the array, it must be in the segment values[l..i]
not far enough: can only be in values[i+1..u]
 boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2;    // roughly the middle
    if (v == values[i]) {
        return true;    // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else {    // values[i] < v
        return isBetween(v, i+1, u);
    }
}
boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2;  // roughly the middle
    if (v == values[i]) {
        return true;  // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else {  // values[i] < v
        return isBetween(v, i+1, u);
    }
}

toofar; if v is in the array, it must be in the segment values[l..i]
notfar enough: can only be in values[i+1..u]
bool isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2; // roughly the middle
    if (v == values[i]) {
        return true; // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else { // values[i] < v
        return isBetween(v, i+1, u);
    }
}

nothing can be in the "empty" segment too far; if v is in the array, it must be in the segment values[l..i]
not far enough: can only be in values[i+1..u]
boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2;  // roughly the middle
    if (v == values[i]) {
        return true;  // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else {  // values[i] < v
        return isBetween(v, i+1, u);
    }
}

nothing can be in the "empty" segment too far; if v is in the array, it must be in the segment values[l..i]
not far enough: can only be in values[i+1..u]
boolean isBetween(int v, int l, int u) {
    if (u <= l) return false;
    int i = (l + u) / 2;  // roughly the middle
    if (v == values[i]) {
        return true;  // found it
    } else if (v < values[i]) {
        return isBetween(v, l, i);
    } else {  // values[i] < v
        return isBetween(v, i+1, u);
    }
}
We can do the same sort of thing with a loop:

```java
public boolean isIn(int v) {
    /* if v is in the array, *
    * its index must be at least l
    */
    int l = 0;
    /* if v is in the array, *
    * its index must be less than u
    */
    int u = values.length;
    // used for half-way between l and u
    int i;
}
We can do the same sort of thing with a loop:

```java
public boolean isIn(int v) {
    /* if v is in the array,
       * its index must be at least l
       */
    int l = 0;
    /* if v is in the array,
       * its index must be less than u
       */
    int u = values.length;
    // used for half-way between l and u
    int i;
```
Binary Search: Loopy Remix

We can do the same sort of thing with a loop:

```java
public boolean isIn(int v) {
    /* if v is in the array,
     * its index must be at least l
     */
    int l = 0;
    /* if v is in the array,
     * its index must be less than u
     */
    int u = values.length;
    // used for half-way between l and u
    int i;
```
Binary Search: Loopy Remix

while (l < u) {
    i = (l + u) / 2;  // roughly half-way
    if (values[i] == v) {  // found it
        return true;
    } else if (v < values[i]) {  // too far
        u = i;  // search values[l..i]
    } else {  // not far enough
        l = i+1;  // search values[i+1..u]
    }
}

return false;

return true if we find \( v \) in \( \text{values}[l..u] \), exit loop and return false when \( \text{values}[l..u] \) is the empty segment
Binary Search: Loopy Remix

while (l < u) {
  i = (l + u) / 2;  // roughly half-way
  if (values[i] == v) {  // found it
    return true;
  } else if (v < values[i]) {  // too far
    u = i;  // search values[1..i]
  } else {  // not far enough
    l = i + 1;  // search values[i+1..u]
  }
}
return false;

return true if we find \( v \) in \( \text{values}[l..u] \), exit loop and return false when \( \text{values}[l..u] \) is the empty segment
Binary Search: Loopy Remix

```c
while (l < u) {
    i = (l + u) / 2;   // roughly half-way
    if (values[i] == v) {   // found it
        return true;
    } else if (v < values[i]) {   // too far
        u = i;   // search values[l..i]
    } else {    // not far enough
        l = i+1;   // search values[i+1..u]
    }
}
return false;
```

return true if we find \( v \) in \( \text{values}[l..u] \), exit loop and return false when \( \text{values}[l..u] \) is the empty segment.
Binary Search: Loopy Remix

while (l < u) {
    i = (l + u) / 2;       // roughly half-way
    if (values[i] == v) {   // found it
        return true;
    } else if (v < values[i]) {  // too far
        u = i;                // search values[l..i]
    } else {                // not far enough
        l = i+1;              // search values[i+1..u]
    }
}

return false;

return true if we find v in values[l..u], exit loop and return false when values[l..u] is the empty segment
Binary Search: Loopy Remix

while (l < u) {
  i = (l + u) / 2;  // roughly half-way
  if (values[i] == v) {  // found it
    return true;
  } else if (v < values[i]) {
    u = i;  // search values[l..i]
  } else {
    l = i+1;  // search values[i+1..u]
  }
}

return false;

return true if we find \( v \) in \( \text{values}[l..u] \), exit loop and return false when \( \text{values}[l..u] \) is the empty segment.
Binary Search: Loopy Remix

```
while (l < u) {
    i = (l + u) / 2; // roughly half-way
    if (values[i] == v) { // found it
        return true;
    } else if (v < values[i]) { // too far
        u = i; // search values[l..i]
    } else { // not far enough
        l = i+1; // search values[i+1..u]
    }
}
return false;
```
while (l < u) {
    i = (l + u) / 2;  // roughly half-way
    if (values[i] == v) {  // found it
        return true;
    }
    else if (v < values[i]) {  // too far
        u = i;  // search values[l..i]
    }
    else {  // not far enough
        l = i+1;  // search values[i+1..u]
    }
}
return false;

return true if we find v in values[l..u], exit loop and return false when values[l..u] is the empty segment.
return true if we find \( v \) in \( \text{values}[l..u] \), exit loop and return false when \( \text{values}[l..u] \) is the empty segment
Efficiency Issues

Binary search is a lot more efficient than linear search. In the worst case, we don’t need to search through the entire array element-by-element. The number of steps required is logarithmic: we could double the length of the array, and only require a fixed number of additional steps to find any given element.

However, the search algorithm does require the array to be sorted, which means it will take longer to add elements to the array (‘insertion’).

The approach is worthwhile if we expect to perform more look-up operations than insertion operations.
Binary search is a lot more efficient than linear search. In the worst case, we don’t need to search through the entire array element-by-element. The number of steps required is logarithmic: we could double the length of the array, and only require a fixed number of additional steps to find any given element.

However, the search algorithm does require the array to be sorted, which means it will take longer to add elements to the array (‘insertion’).

The approach is worthwhile if we expect to perform more look-up operations than insertion operations.
We can get quick look-up operations and quick insertion operations by representing a sorted list of numbers as a **Binary Search Tree**.

A **Binary Search Tree** is a binary tree with the following property:

<table>
<thead>
<tr>
<th>BST class invariant</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>For every node in the tree:</th>
</tr>
</thead>
</table>

- every value in the left subtree is strictly less than the internal label; and
- every value in the right subtree is strictly greater than the internal label.
We can get quick look-up operations and quick insertion operations by representing a sorted list of numbers as a Binary Search Tree.

A Binary Search Tree is a binary tree with the following property:

**BST class invariant**

For every node in the tree:

- every value in the left subtree is strictly less than the internal label; and
- every value in the right subtree is strictly greater than the internal label.
A Binary Search Tree

Here’s a picture of a tree that satisfies this class invariant:
Maintaining the Invariant

In order for the defining property of Binary Search Trees to be a class invariant, we need:

- the constructor to make this property true, and
- all public methods to preserve the property.

We can achieve this by allowing Binary Search Trees to be built up using only:

- a constructor that gives the empty tree, and
- one public method, `insert(int)`, that inserts a given integer at the appropriate point in the Binary Search Tree.

(We can modify our earlier implementation of binary trees to get the underlying tree structure.)
Maintaining the Invariant

In order for the defining property of Binary Search Trees to be a class invariant, we need:

- the constructor to make this property true, and
- all public methods to preserve the property.

We can achieve this by allowing Binary Search Trees to be built up using only:

- a constructor that gives the empty tree, and
- one public method, `insert(int)`, that inserts a given integer at the appropriate point in the Binary Search Tree.

(We can modify our earlier implementation of binary trees to get the underlying tree structure.)
Maintaining the Invariant

In order for the defining property of Binary Search Trees to be a class invariant, we need:

- the constructor to make this property true, and
- all public methods to preserve the property.

We can achieve this by allowing Binary Search Trees to be built up using only:

- a constructor that gives the empty tree, and
- one public method, `insert(int)`, that inserts a given integer at the appropriate point in the Binary Search Tree.

(We can modify our earlier implementation of binary trees to get the underlying tree structure.)
Maintaining the Invariant

In order for the defining property of Binary Search Trees to be a class invariant, we need:

- the constructor to make this property true, and
- all public methods to preserve the property.

We can achieve this by allowing Binary Search Trees to be built up using only:

- a constructor that gives the empty tree, and
- one public method, `insert(int)`, that inserts a given integer at the appropriate point in the Binary Search Tree.

(We can modify our earlier implementation of binary trees to get the underlying tree structure.)
insert(22):
Insertion in a BST

insert(22):

22 < 24
Insertion in a BST

insert(22):

22 > 19
Insertion in a BST

insert(22):
22 < 23
Insertion in a BST

insert(22):

\[
22 < 23
\]
Specifying BSTs

Abbreviated from ‘BinarySearchTree’
the empty tree
In Maude, this is a way of writing what a tree looks like; in Java, this will correspond to a TreeNode instance
The same as for Binary Trees
Specifying BSTs

Abbreviated from ‘BinarySearchTree’
the empty tree
In Maude, this is a way of writing what a tree looks like; in Java, this will correspond to a TreeNode instance
The same as for Binary Trees
Specifying BSTs

file: BinarySearchTree.maude

mod BINARY_SEARCH_TREE is
  protecting INT .
  sort BSTree .
  op null : → BSTree .
  op _/\_/ : BSTree Int BSTree → BSTree . *** private
  op insert : Int BSTree → BSTree .
  op height : BSTree → Int .
  op isIn : BSTree → Bool .

Abbreviated from ‘BinarySearchTree’
the empty tree
In Maude, this is a way of writing what a tree looks like; in Java,
this will correspond to a TreeNode instance
The same as for Binary Trees
Specifying BSTs

mod BINARY_SEARCH_TREE is
  protecting INT.

sort BSTree.

op null : -> BSTree.

op _/\_ : BSTree Int BSTree -> BSTree. *** private

op insert : Int BSTree -> BSTree.

op height : BSTree -> Int.

op isIn : BSTree -> Bool.

Abbreviated from ‘BinarySearchTree’
the empty tree
In Maude, this is a way of writing what a tree looks like; in Java,
this will correspond to a TreeNode instance
The same as for Binary Trees
Specifying BSTs

Abbreviated from ‘BinarySearchTree’
the empty tree
In Maude, this is a way of writing what a tree looks like; in Java, this will correspond to a TreeNode instance
The same as for Binary Trees

```maude
mod BINARY_SEARCH_TREE is
  protecting INT.
  sort BSTree.
  op null : → BSTree.
  op _/../_ : BSTree Int BSTree → BSTree. *** private
  op insert : Int BSTree → BSTree.
  op height : BSTree → Int.
  op isIn : BSTree → Bool.
```
Specifying BSTs

```
mod BINARY_SEARCH_TREE is
    protecting INT .
    sort BSTree .
    op null : -> BSTree .
    op _/\_ : BSTree Int BSTree -> BSTree . *** private
    op insert : Int BSTree -> BSTree .
    op height : BSTree -> Int .
    op isIn : BSTree -> Bool .
```

Abbreviated from ‘BinarySearchTree’
the empty tree
In Maude, this is a way of writing what a tree looks like; in Java, this will correspond to a TreeNode instance
The same as for Binary Trees
Specifying BSTs

```maude
mod BINARY_SEARCH_TREE is
  protecting INT.
  sort BSTree.
  op null : -> BSTree.
  op _/\_ : BSTree Int BSTree -> BSTree. *** private
  op insert : Int BSTree -> BSTree.
  op height : BSTree -> Int.
  op isIn : BSTree -> Bool.

Abbreviated from ‘BinarySearchTree’
the empty tree
In Maude, this is a way of writing what a tree looks like; in Java,
this will correspond to a TreeNode instance
The same as for Binary Trees
```
Specifying BSTs

```maude
vars I J : Int.
vars T1 T2 : BSTree.
eq height(null) = 0.
eq height(T1 / I \ T2) = max(height(T1), height(T2)) + 1.
```

Remember this? — two equations defining \( \text{height} \) the argument is either

- null (the empty tree: in which case the height is 0)
- or not the empty tree: in Java, this will be a TreeNode with internal label and left and right subtrees
Remember this? — two equations defining height

the argument is either

- null (the empty tree: in which case the height is 0)
- or not the empty tree: in Java, this will be a TreeNode with
  internal label and left and right subtrees
Remember this? — two equations defining the `height` of the argument is either:

- null (the empty tree: in which case the height is 0)
- or not the empty tree: in Java, this will be a `TreeNode` with internal label and left and right subtrees
file:BinarySearchTree.maude

\textbf{vars} | J : \text{Int}.
\textbf{vars} T1 T2 : BSTree.

\textbf{eq} \text{height(null)} = 0.
\textbf{eq} \text{height(T1 / I \setminus T2)} = \max(\text{height(T1)}, \text{height(T2)}) + 1.

Remember this? — two equations defining \text{height}
the argument is either

- \textbf{null} (the empty tree: in which case the height is 0)
- \textbf{or} not the empty tree: in Java, this will be a \text{TreeNode} with
  internal label and left and right subtrees
Specifying BSTs

file: BinarySearchTree.maude

vars I J : Int.
vars T1 T2 : BSTree.
eq height(null) = 0.
eq height(T1 / I \ T2) = max(height(T1), height(T2)) + 1.

Remember this? — two equations defining height:
the argument is either
- null (the empty tree: in which case the height is 0)
- or not the empty tree: in Java, this will be a TreeNode with
  internal label and left and right subtrees
Specifying BSTs

vars I J : Int.
vars T1 T2 : BSTree.
eq height(null) = 0.
eq height(T1 / I \ T2) = max(height(T1), height(T2)) + 1.

Remember this? — two equations defining `height` the argument is **either**

- null (the empty tree: in which case the height is 0)
- **or** not the empty tree: in Java, this will be a `TreeNode` with internal label and left and right subtrees
Specifying BSTs

To define insert, we look at the form of the tree that is the second argument: either null or a node with subtrees:

```
file:BinarySearchTree.maude

eq  insert(I, null) = null / I \ null .

cq  insert(I, T1 / J \ T2) = insert(I, T1) / J \ T2 if I < J .

cq  insert(I, T1 / J \ T2) = T1 / J \ insert(I, T2) if I > J .
```

insert into the left subtree if I is less than the internal label
insert into the right subtree if I is greater than the internal label
Specifying BSTs

To define insert, we look at the form of the tree that is the second argument: either null or a node with subtrees:

```
file:BinarySearchTree.maude

eq insert(I, null) = null / I \ null .

cq insert(I, T1 / J \ T2) = insert(I, T1) / J \ T2 if I < J .

cq insert(I, T1 / J \ T2) = T1 / J \ insert(I, T2) if I > J .
```

insert into the left subtree if I is less than the internal label
insert into the right subtree if I is greater than the internal label
To define insert, we look at the form of the tree that is the second argument: either null or a node with subtrees:

\[
\begin{align*}
\text{eq} \quad \text{insert}(I, \text{null}) &= \text{null} / I \setminus \text{null} . \\
\text{cq} \quad \text{insert}(I, T1 / J \setminus T2) &= \text{insert}(I, T1) / J \setminus T2 \quad \text{if} \quad I < J . \\
\text{cq} \quad \text{insert}(I, T1 / J \setminus T2) &= T1 / J \setminus \text{insert}(I, T2) \quad \text{if} \quad I > J .
\end{align*}
\]

insert into the left subtree if \( I \) is less than the internal label
insert into the right subtree if \( I \) is greater than the internal label
Specifying BSTs

To define insert, we look at the form of the tree that is the second argument: either null or a node with subtrees:

\[
\begin{align*}
\text{eq} & \quad \text{insert}(I, \text{null}) = \text{null} / I \setminus \text{null} . \\
\text{cq} & \quad \text{insert}(I, T_1 / J \setminus T_2) = \text{insert}(I, T_1) / J \setminus T_2 \quad \text{if} \quad I < J . \\
\text{cq} & \quad \text{insert}(I, T_1 / J \setminus T_2) = T_1 / J \setminus \text{insert}(I, T_2) \quad \text{if} \quad I > J .
\end{align*}
\]

insert into the left subtree if \( I \) is less than the internal label
insert into the right subtree if \( I \) is greater than the internal label
Specifying BSTs

To define insert, we look at the form of the tree that is the second argument: either null or a **node with subtrees**:

```
file:BinarySearchTree.maude

**eq** insert(I, null) = null / I \ null .

**cq** insert(I, T1 / J \ T2) = insert(I, T1) / J \ T2 if I < J .

**cq** insert(I, T1 / J \ T2) = T1 / J \ insert(I, T2) if I > J .
```

- insert into the left subtree if I is less than the internal label
- insert into the right subtree if I is greater than the internal label
Specifying BSTs

To define insert, we look at the form of the tree that is the second argument: either null or a node with subtrees:

```
file:BinarySearchTree.maude

eq  insert(I, null) = null / I \ null .
cq  insert(I, T1 / J \ T2) = insert(I, T1) / J \ T2  if I < J .
cq  insert(I, T1 / J \ T2) = T1 / J \ insert(I, T2)  if I > J .
```

insert into the left subtree if I is less than the internal label
insert into the right subtree if I is greater than the internal label
Specifying BSTs

To define insert, we look at the form of the tree that is the second argument: either null or a node with subtrees:

```
file:BinarySearchTree.maude

\textbf{eq} \quad \text{insert}(I, \text{null}) = \text{null} / I \backslash \text{null} .

\textbf{cq} \quad \text{insert}(I, T_1 / J \backslash T_2) = \text{insert}(I, T_1) / J \backslash T_2 \quad \text{if} \quad I < J .

\textbf{cq} \quad \text{insert}(I, T_1 / J \backslash T_2) = T_1 / J \backslash \text{insert}(I, T_2) \quad \text{if} \quad I > J .
```

insert into the left subtree if $I$ is less than the internal label
insert into the right subtree if $I$ is greater than the internal label
Specifying BSTs

isIn is defined similarly

```maude
eq  isIn(I, null) = false .
cq  isIn(I, T1 / J \ T2) = isIn(I, T1)  if  I < J .
cq  isIn(I, T1 / J \ T2) = isIn(I, T2)  if  I > J .
cq  isIn(I, T1 / J \ T2) = true  if  I == J .
endm
```

search in the left subtree if I is less than the internal label
search in the right subtree if I is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

```maude
file:BinarySearchTree.maude

eq  isIn(I, null) = false .
cq  isIn(I, T1 / J \ T2) = isIn(I, T1)  if  I < J .
cq  isIn(I, T1 / J \ T2) = isIn(I, T2)  if  I > J .
cq  isIn(I, T1 / J \ T2) = true     if  I == J .
endm
```

search in the left subtree if I is less than the internal label
search in the right subtree if I is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

```maude
file:BinarySearchTree.maude

eq  isIn(I, null) = false .

cq  isIn(I, T1 / J \ T2) = isIn(I, T1)  if  I < J .

cq  isIn(I, T1 / J \ T2) = isIn(I, T2)  if  I > J .

cq  isIn(I, T1 / J \ T2) = true  if  I == J .

endm
```

search in the left subtree if I is less than the internal label
search in the right subtree if I is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

```
file:BinarySearchTree.maude

eq isIn(l, null)  = false .
cq isIn(l, T1 / J \ T2) = isIn(l, T1) if l < J .
cq isIn(l, T1 / J \ T2) = isIn(l, T2) if l > J .
cq isIn(l, T1 / J \ T2) = true if l == J .
endm
```

search in the left subtree if l is less than the internal label
search in the right subtree if l is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

```maude
eq isIn(I, null) = false .
cq isIn(I, T1 / J \ T2) = isIn(I, T1) if I < J .
cq isIn(I, T1 / J \ T2) = isIn(I, T2) if I > J .
cq isIn(I, T1 / J \ T2) = true if I == J .
endm
```

search in the left subtree if I is less than the internal label
search in the right subtree if I is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

```maude
eq  isIn(I, null) = false .
cq  isIn(I, T1 / J \ T2) = isIn(I, T1) if I < J .
cq  isIn(I, T1 / J \ T2) = isIn(I, T2) if I > J .
cq  isIn(I, T1 / J \ T2) = true  if I == J .
endm
```

search in the left subtree if I is less than the internal label
search in the right subtree if I is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

```
file:BinarySearchTree.maude

eq  isIn(l, null) = false .
ceq  isIn(l, T1 / J \ T2) = isIn(l, T1) if l < J .
ceq  isIn(l, T1 / J \ T2) = isIn(l, T2) if l > J .
ceq  isIn(l, T1 / J \ T2) = true if l == J .
endm
```

search in the left subtree if l is less than the internal label
search in the right subtree if l is greater than the internal label
found it!
isIn is defined similarly

```maude
eq isIn(I, null) = false .
cq isIn(I, T1 / J \ T2) = isIn(I, T1)  if  I < J .
cq isIn(I, T1 / J \ T2) = isIn(I, T2)  if  I > J .
cq isIn(I, T1 / J \ T2) = true  if  I == J .
endm
```

search in the left subtree if I is less than the internal label
search in the right subtree if I is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

file:BinarySearchTree.maude

\[
\text{eq} \quad \text{isIn}(I, \text{null}) = \text{false}.
\]

\[
\text{cq} \quad \text{isIn}(I, T_1 / J \setminus T_2) = \text{isIn}(I, T_1) \quad \text{if} \quad I < J.
\]

\[
\text{cq} \quad \text{isIn}(I, T_1 / J \setminus T_2) = \text{isIn}(I, T_2) \quad \text{if} \quad I > J.
\]

\[
\text{cq} \quad \text{isIn}(I, T_1 / J \setminus T_2) = \text{true} \quad \text{if} \quad I == J.
\]

endm

search in the left subtree if I is less than the internal label
search in the right subtree if I is greater than the internal label
found it!
Specifying BSTs

isIn is defined similarly

```maude
file:BinarySearchTree.maude

eq  isIn(l, null)  =  false .
cq  isIn(l, T1 / J \ T2)  =  isIn(l, T1)  if  l < J .
cq  isIn(l, T1 / J \ T2)  =  isIn(l, T2)  if  l > J .
cq  isIn(l, T1 / J \ T2)  =  true  if  l == J .
endm
```

search in the left subtree if l is less than the internal label
search in the right subtree if l is greater than the internal label
found it!
We use the `TreeNode` class from Lecture 14 (slightly modified):

```java
public class BinarySearchTree {
    private TreeNode root;

    public BinarySearchTree() {
    }

    public void insert(int v) {
        ...
    }

    public int height() {
        ...
    }

    public boolean isIn(int v) {
        ...
    }
}
```
### Nodes in Java

#### class BinarySearchTree, contd.

```java
private class TreeNode {
    private int value;
    private TreeNode leftSubTree;
    private TreeNode rightSubTree;

    public TreeNode(int v) {
        value = v;
    }
}
```
Insertion

in class BinarySearchTree

public void insert(int v) {
    if (root == null) {
        root = new TreeNode(v);
    } else {
        root.insert(v);
    }
}
public void insert(v) {
    if (v < value) {
        if (leftSubtree == null) {
            leftSubtree = new TreeNode(v);
        } else {
            leftSubtree.insert(v);
        }
    } else if (v > value) {
        if (rightSubtree == null) {
            rightSubtree = new TreeNode(v);
        } else {
            rightSubtree.insert(v);
        }
    }
}
public boolean isIn(int v) {
    TreeNode tn = root;
    while (tn != null) {
        if (v == value) {
            return true;
        } else if (v < value) {
            tn = leftSubtree;
        } else {
            tn = rightSubtree;
        }
    }
    return false;
}
Loops v Recursion

`isIn()` and `insert(int)` can both be implemented using either a loop or recursive calls (see the loopy version of `insert(int)` in the source-code from the module web-page).

Loops are generally more efficient than recursive methods, which have an additional overhead in calling the method (placing it on the method-call stack, looking up the code, etc.).

We’ve already seen a recursive implementation of `height()`. Implementing it using a loop is more difficult, as both left- and right-subtrees need to be considered.

One way to improve the efficiency of `height()` is to add a field `height` to the `TreeNode` class, and update this field whenever the height of the tree changes (e.g., by inserting values).

The class invariant would be:

`height` really is the height of the tree.
Loops v Recursion

`isIn()` and `insert(int)` can both be implemented using either a loop or recursive calls (see the loopy version of `insert(int)` in the source-code from the module web-page).

Loops are generally more efficient than recursive methods, which have an additional overhead in calling the method (placing it on the method-call stack, looking up the code, etc.).

We’ve already seen a recursive implementation of `height()`. Implementing it using a loop is more difficult, as both left- and right-subtrees need to be considered.

One way to improve the efficiency of `height()` is to add a field `height` to the `TreeNode` class, and update this field whenever the height of the tree changes (e.g., by inserting values).

The class invariant would be: `height` really is the height of the tree.
That’s All, Folks!

Summary

- Linear and binary search
- Binary Search Trees
- Loops v recursion

Next:

height