Data Structures and Algorithms

Week 3

Lists
List ADT

- Defines an ADT that specifies a general list data structure.
- The location of an element is determined by an *index*.
- The index of an element *e* is the number of elements before *e* in the list.
- So, the index of the first element is 0 and that of the last element is $n - 1$, assuming that there are *n* elements in the list.
- The ADT supports the operations in the following slide.

Lists
List ADT

- *size*( ): Returns the number of elements currently in the list.
- *isEmpty*( ): Returns *true* if the list is empty. Returns *false* otherwise.
- *get*(*i*): Returns the element whose index is *i*.
- *set*(*i*, *e*): The element at index *i* is replaced with a new element *e* and the old, replaced element is returned.
- *add*(*i*, *e*): Inserts a new element *e* at location with index *i*, and the element which is currently at index *i* and subsequent elements are moved one index later in the list.
- *remove*(*i*): Removes and returns the element at index *i*. The elements that are currently in [*i*+1 .. *size*( ) – 1] are moved one index earlier in the list.
- An error occurs if *i* is not in the range [0 .. *size*( ) – 1], except for the *add* method, for which a valid range is [0 .. *size*( )].
- *List.java* (interface)

Lists
List ADT

- Illustration

Operation	Return Value	List Contents
add(0, 25)	none	(25)
add(0, 32)	none	(32, 25)
add(2, 12)	none	(32, 25, 12)
add(2, 15)	none	(32, 25, 15, 12)
get(2)	15	(32, 25, 15, 12)
get(4)	“error”	(32, 25, 15, 12)
size( )	4	(32, 25, 15, 12)
remove(2)	15	(32, 25, 12)
remove(3)	“error”	(32, 25, 12)
size( )	3	(32, 25, 12)
get(1)	25	(32, 25, 12)
set(0, 10)	32	(10, 25, 12)
size( )	3	(10, 25, 12)
get(1)	25	(10, 25, 12)
set(4, 29)	“error”	(10, 25, 12)

Lists
Array Lists

- A list is implemented using an array as an underlying storage.
- Advantage: direct access to elements
- Disadvantage:
  - Adding or removing elements may require restructuring (shifting of elements) of the array.
  - Size is fixed

Lists

Array Lists with Bounded Array

- ArrayList class

```
public class ArrayList<E> implements List<E> {
  // instance variables
  public static final int CAPACITY=16;     // default array capacity
  private E[ ] data;                       // generic array used for storage
  private int size = 0;                    // current number of elements
  // constructors
  public ArrayList() {this(CAPACITY);}
  public ArrayList(int capacity) {
    data = (E[ ]) new Object[capacity];
  }

. . .
```

Lists
Array Lists with Bounded Array

- Methods

```
public int size() { return size; }

public boolean isEmpty() { return size == 0; }

public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];
}

public E set(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

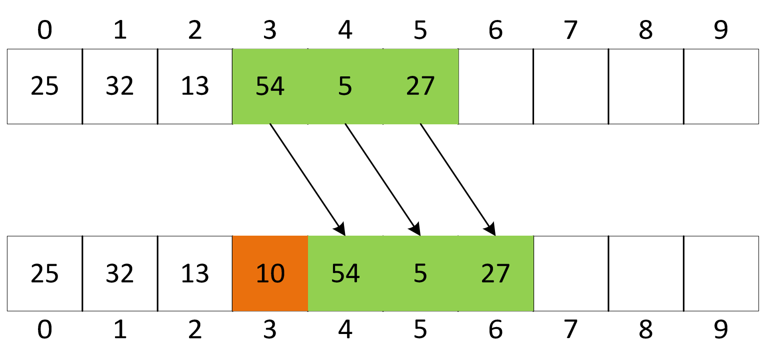
Lists

Array Lists with Bounded Array

- Methods (continued)

```
public void add(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size + 1);
  if (size == data.length)              // not enough capacity
    throw new IllegalStateException("Array is full");
  for (int k=size-1; k >= i; k--)       // start by shifting rightmost
    data[k+1] = data[k];
  data[i] = e;                          // ready to place the new element
  size++;
}
```

add(3, 10)

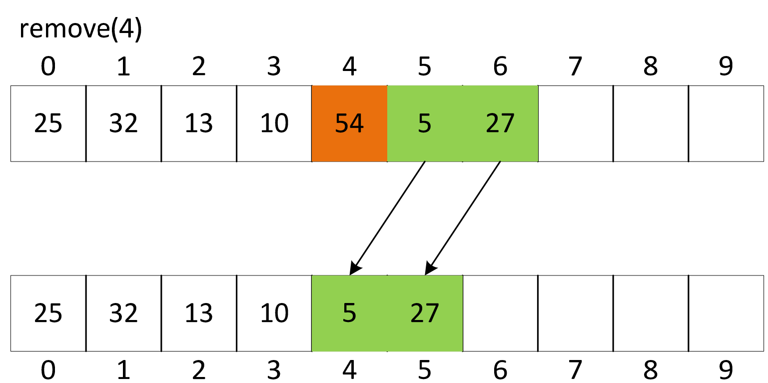


Lists

Array Lists with Bounded Array

- Methods (continued)

```
public E remove(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  for (int k=i; k < size-1; k++)        // shift elements to fill hole
    data[k] = data[k+1];
  data[size-1] = null;                  // help garbage collection
  size--;
  return temp;
}
```



Lists

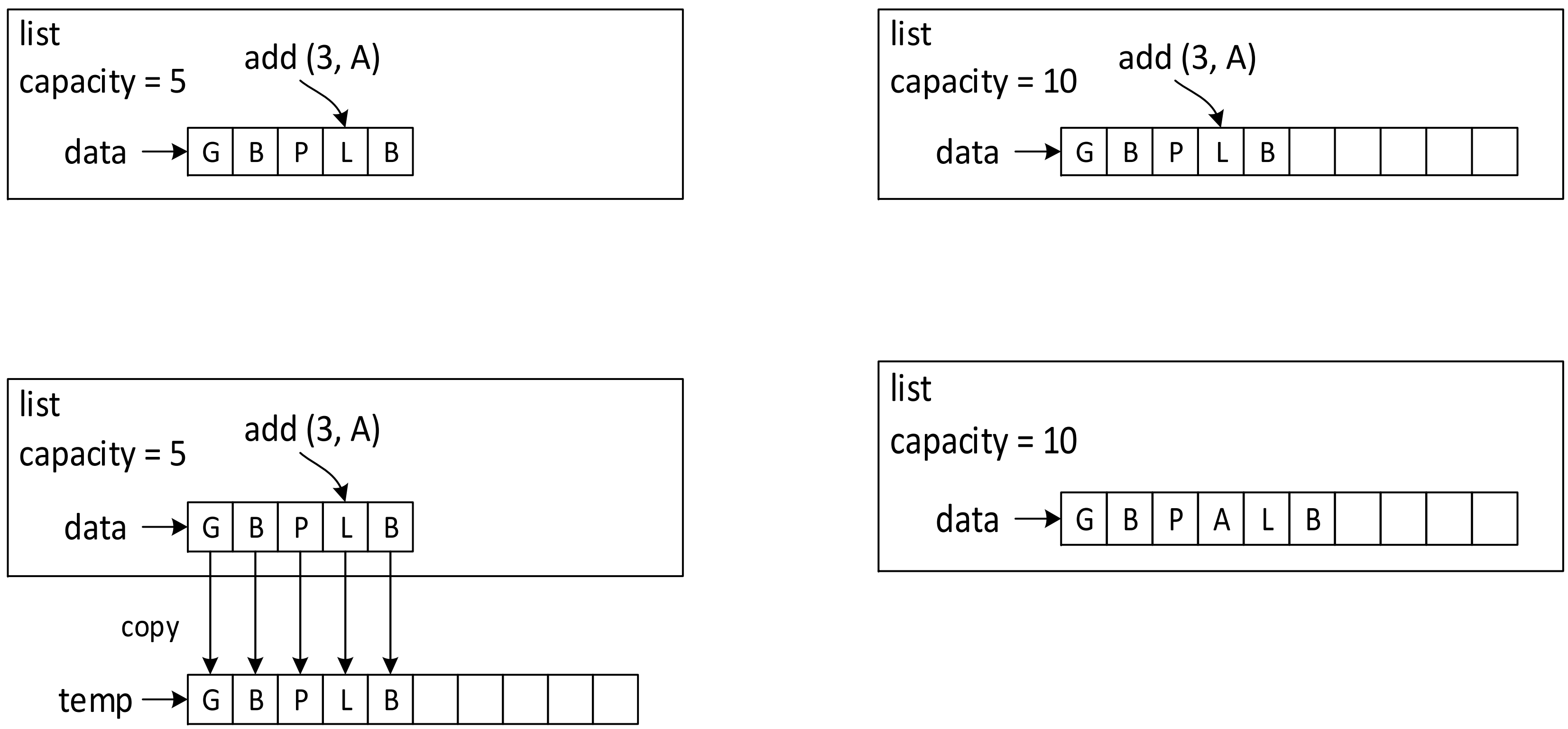
Array Lists with Bounded Array

- Running time analysis

Method	Running Time
size( )	O(1)
isEmpty( )	O(1)
get(i)	O(1)
set(i, e)	O(1)
add(i, e)	O(n)
remove(i)	O(n)

Lists
Array Lists with Dynamic Array

- Resize the internal array when the array is full.



Lists
Array Lists with Dynamic Array

- Resize method

```
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity];
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;
}
```

Lists
Array Lists with Dynamic Array

- Revised add method

```
public void add(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size + 1);
  if (size == data.length)              // not enough capacity, overflow
    resize(2 * data.length);            // increase the capacity
  for (int k=size-1; k >= i; k--)       // start by shifting rightmost
    data[k+1] = data[k];
  data[i] = e;                          // ready to place the new element
  size++;
}
```

- *ArrayList.java*

Lists

Positional Lists

- A *position* is an abstraction that represents a location of an element in a list.
- A position hides internal nodes (or details) of lists.
- A position allows a user to refer to any element in a list regardless of its location.
- We can perform local operations such as *add before* and *add after*.
- An example: a *cursor* in a text document.

- A position ADT has only one method:

  getElement( ): Returns the element stored at this position.

Lists
Positional Lists

- When you write a method that uses *position* for this course:
  - If the method receives an argument of *Position* type, convert it to a local variable of *Node* type by invoking the *validate* method .
  - Use the local variable (which is a *Node*) within your method.
  - If the return type is *Position*, you can return the local variable as it is (no need to convert the type to *Position*).

Lists
Positional List ADT

- Accessor methods:
  - first( ): Returns the position of the first element of *L* (or null if empty)
  - last( ): Returns the position of the last element of *L* (or null if empty)
  - before(*p*): Returns the position of *L* immediately before position *p* (or null if *p* is the first position)
  - after(*p*): Returns the position of *L* immediately after position *p* (or null if *p* is the last position)
  - isEmpty( ): Returns true if *L* does not have any element.
  - size( ): Returns the number of elements in *L*.

Lists
Positional List ADT

- Update methods
  - addFirst(*e*): A new element *e* is added at the front of the list, and the position of the new element is returned.
  - addLast(*e*): A new element *e* is added at the back of the list, and the position of the new element is returned.
  - addBefore(*p*, *e*): A new element *e* is added immediately before the position *p*, and the position of the new element is returned.
  - addAfter(*p*, *e*): A new element *e* is added immediately before the position *p*, and the position of the new element is returned.
  - set(*p*, *e*): The element at position *p* is replaced with the new element *e*, and the element that was in that position before the replacement is returned.
  - remove(*p*): The element at position *p* is removed and the removed element is returned. The position *p* is invalidated.

Lists

Positional List Interface in Java

- Refer to *PositionalList.java* (interface)

Lists

Positional List using Doubly Linked List

- *LinkedPositionalList* class defines a nested class *Node*.
- The *Node* class is a concrete implementation of the *Position* ADT.
- So, nodes are positions in this implementation.

Lists
Positional List using Doubly Linked List

- Part of the code:

```
public class LinkedPositionalList<E> implements PositionalList<E>
{
    private static class Node<E> implements Position<E> {
        private E element;  // reference to the element in this node
        private Node<E> prev;   // reference to the previous node
        private Node<E> next;  // reference to the next node
        public Node(E e, Node<E> p, Node<E> n) {
            element = e;
            prev = p;
            next = n;
    }
. . .
```

Lists

Positional List using Doubly Linked List

- Complete code of *LinkedPositionalList.java*

Lists
Positional List using Doubly Linked List

- Running times

Method	Running Time
size( )	O(1)
isEmpty( )	O(1)
first( ), last( )	O(1)
before(p), after(p)	O(1)
addFirst(e), addLast(e)	O(1)
addBefore(p, e), addAfter(p, e)	O(1)
set(p, e)	O(1)
remove(p)	O(1)

Lists

Java Iterator and Iterable

- An *Iterator* object is an abstraction.
- It provides a uniform way of traversing collections regardless of their internal organizations.
- The *Iterator* interface has the following methods:

  – hasNext( ): Returns true if there is at least one additional element in the collection.

  – next( ): Returns the next element in the collection.

  – remove( ): Removes from the collection the element returned by the most recent call to next( ). (optional operation)

Lists

Java Iterator and Iterable

- We create an *Iterator* object by invoking the *iterator*( ) method that is defined in the *Iterable* interface.
- Example

```
ArrayList<String> stringList = new ArrayList<>( );
// population of the list omitted
Iterator<String> stringIterator = stringList.iterator( );
While (stringIterator.hasNext( ))
  System.out.println(stringIterator.next( ));
```

- Java *Collection* interface extends the *Iterable* interface so all collection objects can invoke the *iterator*( ) method to create an iterator.

Lists
Java Iterator and Iterable

- Simpler syntax:

for (*ElementType variable* : *collection*) {
    loopBody
}

The previous example is equivalent to:

```
for (String s : stringList) {
        System.out.println(s);
}
```

Lists

Java ListIterator

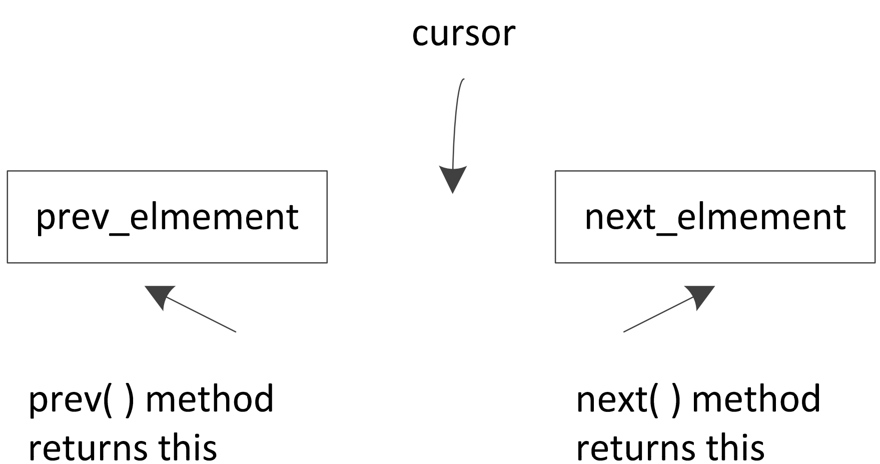
- Java’s *ListIterator* interface extends the *Iterator* interface
- Adds bi-directional traversal of a list.
- A list iterator can move forward and backward.
- A list iterator is assumed to be located before the first element, between two consecutive elements, or after the last element.
- A list iterator is obtained by invoking the *listIterator*( ) method of a *List* interface.
- It inherits all operations of *Iterator* and it also defines additional local update operations.

Lists
Java ListIterator

- add(*e*): Inserts the element *e* at the current position of the iterator.
- hasNext( )
- hasPrevious( )
- previous( ): Returns the element *e* before the current iterator position and sets the current position to be before *e*.
- next( ): Returns the element *e* after the current iterator position and sets the current position to be after *e*.
- nextIndex( ): Returns the index of the next element.
- previoustIndex( ): Returns the index of the previous element.
- remove( ): Removes the element returned by the most recent *next* or *previous* operation.
- set(*e*): Replaces the element returned by the most recent *next* or *previous* operation with *e*.

Lists
Java ListIterator

- Extends the *Iterator* interface
- Allows bidirectional traversal of a list
- *Cursor* is between two elements, say *prev_element* and *next_element*
- *previous*( ) methods returns *prev_element*
- *next*( ) methods returns *next_element*



Lists
Java ListIterator

```
LinkedList<Integer> intList = new LinkedList<>();
intList.add(20); intList.add(40); intList.add(60);
ListIterator<Integer> li;
li = intList.listIterator(); // cursor right before the first element
while (li.hasNext()){     // if there is next element
      System.out.print(li.next() + " "); // walk forward
}
System.out.println();
li = intList.listIterator(intList.size()); // cursor right after the last elem.
while (li.hasPrevious()){                  // if there is previous element
      System.out.print(li.previous() + " "); // walk backward
}
```

Lists
Java ListIterator

- The out put is:

  20 40 60

  60 40 20

- If we execute the following statements:

```
li = intList.listIterator(2); // cursor is between 2nd and 3rd
                              // elements
li.add(100); // add right before next element
```

  The list will have: 20 40 100 60

- remove( ) method removes the last element that was returned by next( ) or previous( )

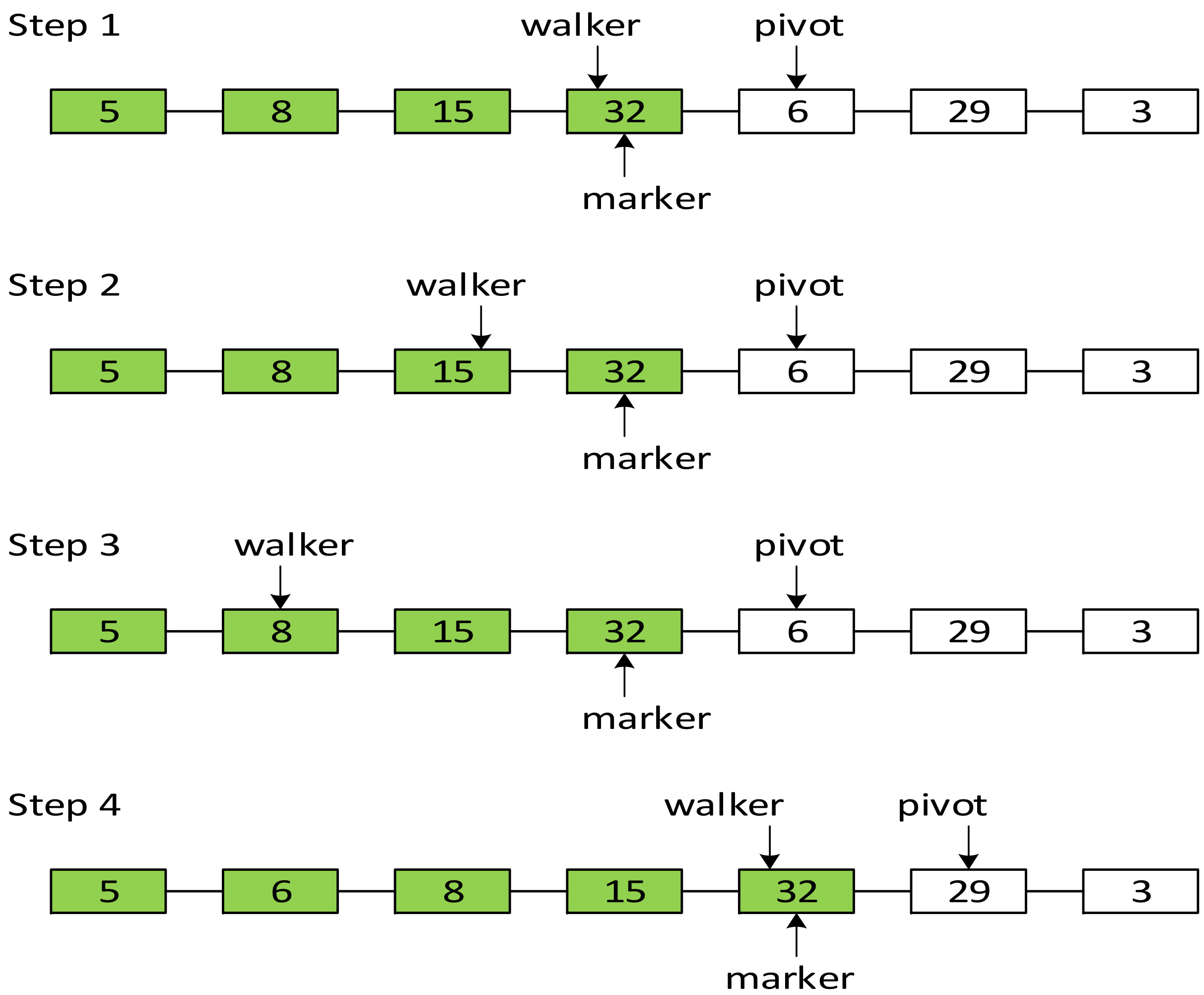
Lists

Sorting a Positional List

- Sorts an elements in a positional list using the insertion-sort algorithm.
- Uses three variables: *marker*, *pivot*, and *walk*.
- During sorting, the list has two parts.
- One part (on the left): already sorted
- The other part (on the right): has elements not explored
- *marker* is the rightmost position in the already sorted.
- *pivot* is the position of the element to the immediate right of *marker*, and represents the first element in the unsorted part.
- The *walk* is used to traverse the already sorted part of the array to decide the correct position of *pivot*.

Lists

Sorting a Positional List



Lists
Sorting a Positional List

- Java code

```
public static void insertionSort(PositionalList<Integer> list) {
  Position<Integer> marker = list.first(); // last position known to be sorted
  while (marker != list.last()) {
      Position<Integer> pivot = list.after(marker);
      int value = pivot.getElement();      // number to be placed
      if (value > marker.getElement())     // pivot is already sorted
          marker = pivot;
      else {                               // must relocate pivot
          Position<Integer> walk = marker;   // find leftmost item greater than value
          while (walk != list.first() && list.before(walk).getElement() > value)
              walk = list.before(walk);
          list.remove(pivot);                // remove pivot entry and
          list.addBefore(walk, value);       // reinsert value in front of walk
       }
   }
}
```

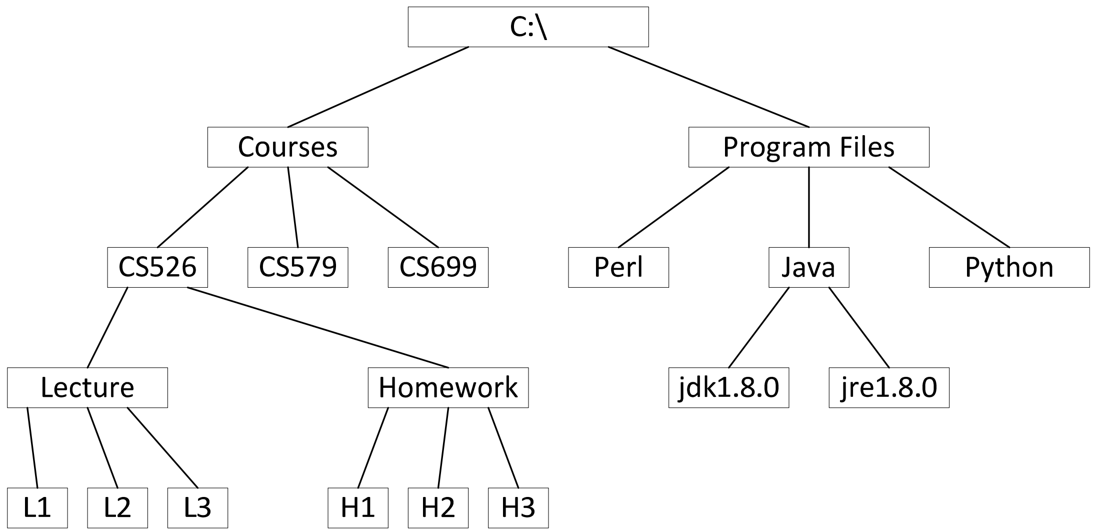
General Trees
Basics

- A *graph* is a set of nodes and a set of edges.
- Formally, a graph $G = (V, E)$, where V is a set of nodes (or vertices) and E is a set of edges.
- Each edge connects two nodes, and is represented as (u, v), where u and v are nodes.

- A ***tree*** is a connected, acyclic, undirected graph with a distinguished node called *root*.
- *Connected*: There is a path from every node to every other node.
- *Acyclic*: There is no cycle
- *Undirected*: Edges have no direction

General Trees
Basics

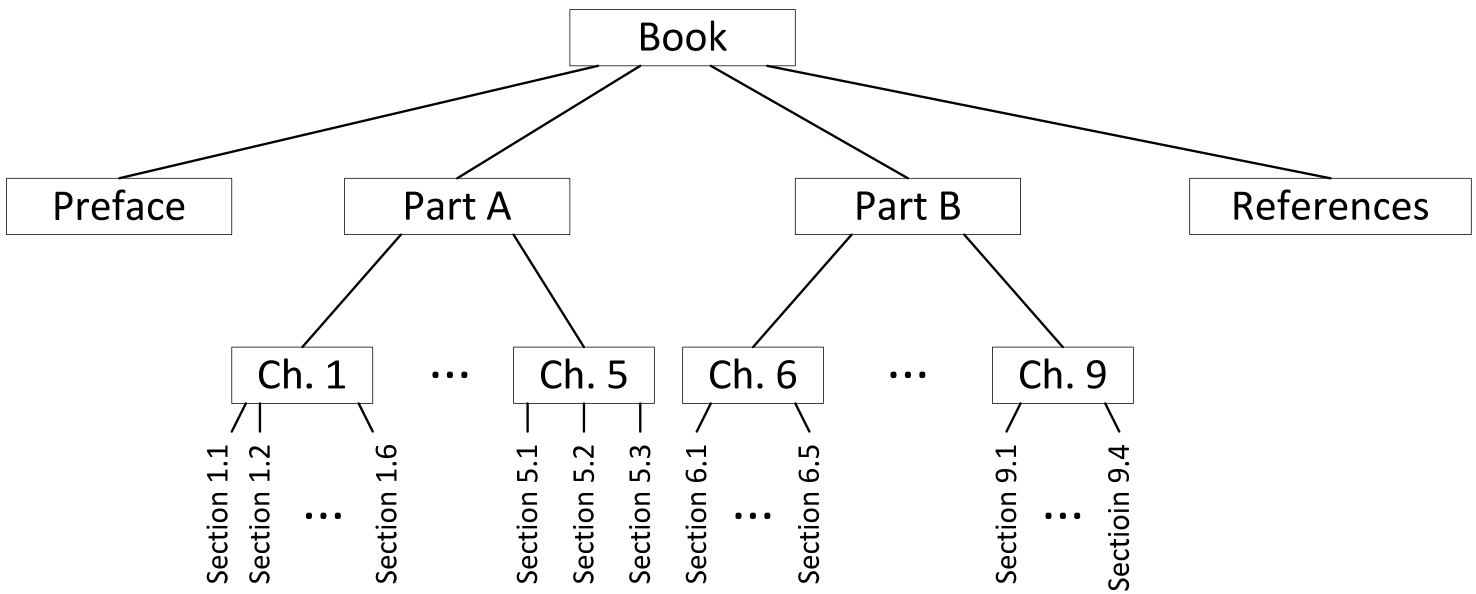
- Example



- Root, parent, child, siblings
- Internal node, external node (or leaf node)
- Ancestor, descendant
- Path

General Trees
Basics

- Ordered tree: There is meaningful ordering among siblings:



General Trees
Tree ADT

- Accessor methods
  - root( ): Returns the position of the root of the tree, or null if the tree is empty.
  - parent(*p*): Returns the position of the parent of position *p*, or null if *p* is the root.
  - children(*p*): Returns the children of position *p*, if any. If the tree is an ordered tree, children are ordered in the result.
  - numChildren(*p*): Returns the number of children of position *p*.

General Trees
Tree ADT

- Query methods
  - isInternal(*p*): Returns true if position *p* is an internal node.
  - isExternal(*p*): Returns true if position *p* is an external node (or a leaf node).
  - isRoot(*p*): Returns true if position *p* is the root of the tree.

General Trees
Tree ADT

- Other general methods

  - size( ): Returns the number of positions (or the elements) in the tree.
  - isEmpty( ): Returns true if the tree does not have any position (or element).
  - iterator( ): Returns an iterator for all elements in the tree. So, the tree is *Iterable*.
  - positions( ): Returns an iterable collection of all positions of the tree.

General Trees
Tree ADT

- Tree interface

```
public interface Tree<E> extends Iterable<E> {
  Position<E> root();
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                              throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size();
  boolean isEmpty();
  Iterator<E> iterator();
  Iterable<Position<E>> positions();
}
```

General Trees
Tree ADT

- AbstractTree abstract class

```
public abstract class AbstractTree<E> implements Tree<E> {

public boolean isInternal(Position<E> p)
      { return numChildren(p) > 0; }
public boolean isExternal(Position<E> p)
      { return numChildren(p) == 0; }
public boolean isRoot(Position<E> p) { return p == root(); }
public boolean isEmpty() { return size() == 0; }
}
```

General Trees
Depth and Height

- Depth
  - If *p* is the root, the depth of *p* is 0.
  - Otherwise, the depth of *p* is one plus the depth of its parent.

```
public int depth(Position<E> p) throws IllegalArgumentException {
  if (isRoot(p))
      return 0;
  else
      return 1 + depth(parent(p));
}
```

Running time = $O(d_p + 1)$
d_p is the depth of p

General Trees
Depth and Height

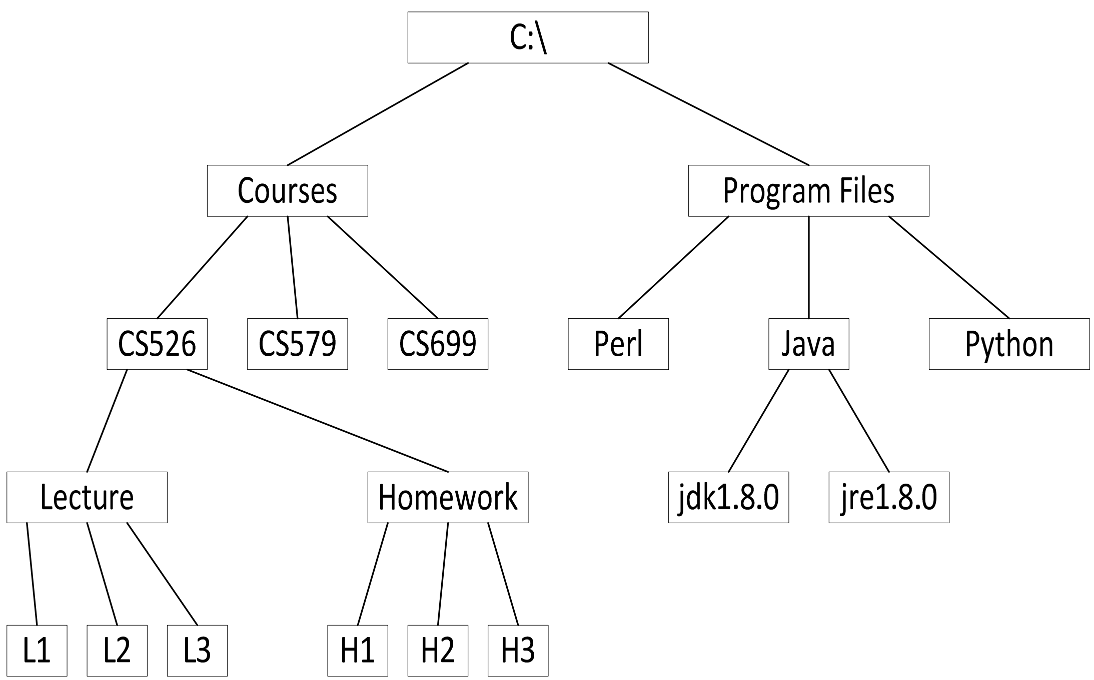
- The *height* of a tree is the length of the longest path from the root downward to an external node.
- Recursive definition:
  - If *p* is a leaf, then the height of *p* is 0.
  - Otherwise, the height of *p* is one more than the maximum of the heights of *p’s* children.

```
public int height(Position<E> p) throws IllegalArgumentException {
  int h = 0;                          // base case if p is external
  for (Position<E> c : children(p))
     h = Math.max(h, 1 + height(c));
  return h;
}
```

Running time = O(n)
n is the number of positions

General Trees
Depth and Height

- Example



- c:\
  - depth 0
  - height 4
- CS526
  - depth 2
  - height 2
- Program Files
  - depth 1
  - height 2

Binary Trees

- A binary tree is an ordered tree with the following properties:
  - Every node has at most two children.
  - Each child node is labeled as being a *left child* or a *right child*.
  - A left child precedes a right child in the order of children of a node

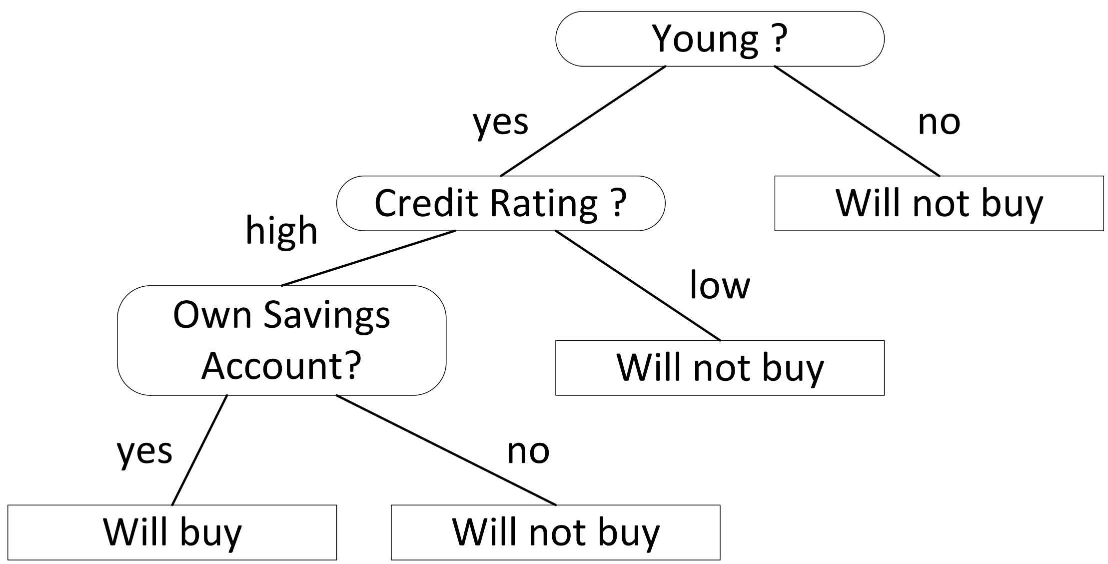
Binary Trees

- The subtree rooted at the left or right child of an internal node *v* is called the *left subtree* or the *right subtree*, respectively, of *v*.

- A binary tree is *proper* if each node has either zero or two children. (also referred to as *full binary tree*).
- So, in a proper binary tree, every internal node has exactly two children.
- A binary tree that is not proper is *improper*.

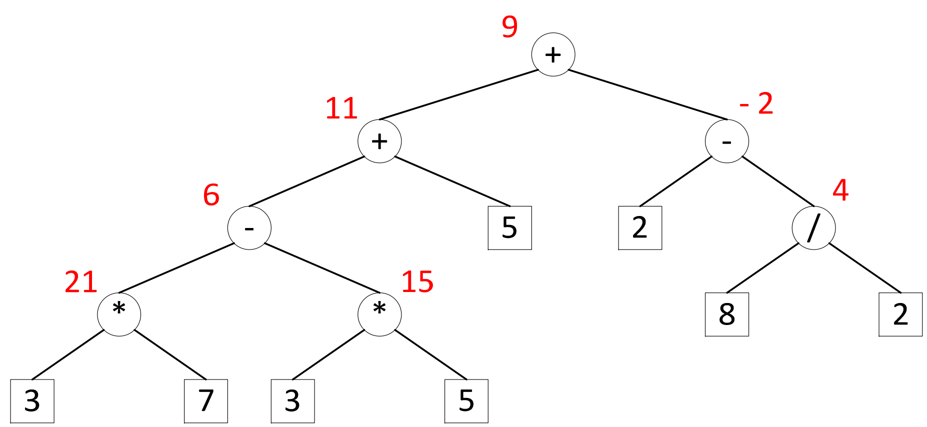
Binary Trees

- Example (a decision tree)



Binary Trees

- Example (arithmetic expression tree)



- ((((3 * 7) – (3 * 5)) + 5) + (2 – (8 / 2)))

Binary Trees

- A binary tree can be recursively defined as follows:

  - A binary tree is either
    - An empty tree, or
    - A nonempty tree with a root node *r* and two binary trees that are the left subtree and the right subtree of *r*. One or both of these subtrees can be empty, by definition.

Binary Trees
ADT

- The binary tree ADT is a specialization of the *Tree ADT.*
- Following additional methods are defined:
  - left(*p*): Returns the position of the left child of *p*. Returns null if *p* has no left child.
  - right(*p*): Returns the position of the right child of *p*. Returns null if *p* has no right child.
  - sibling(*p*): Returns the position of the sibling of *p*. Returns null if *p* has no sibling.

Binary Trees
ADT

- BinaryTree interface

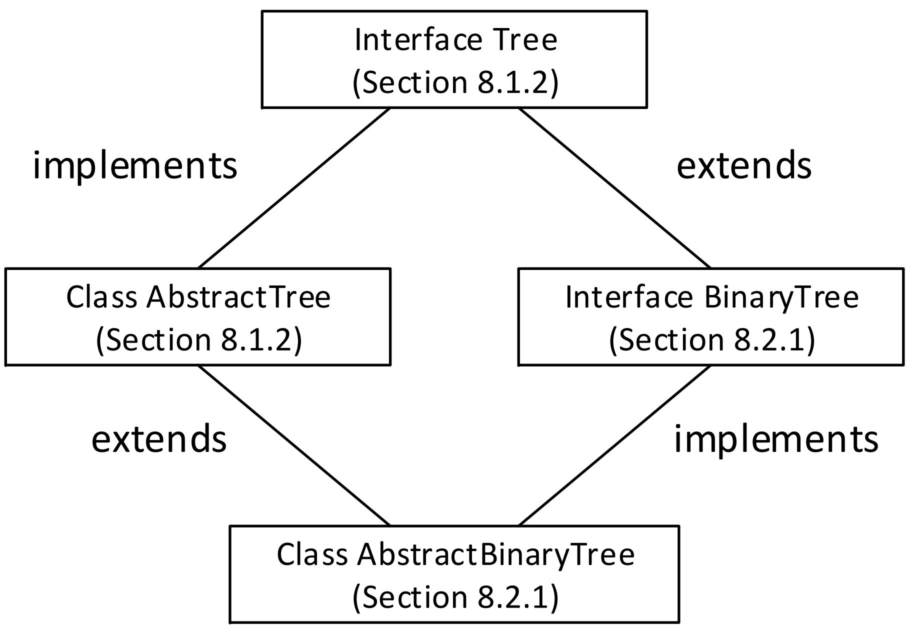
```
public interface BinaryTree<E> extends Tree<E> {

  Position<E> left(Position<E> p) throws
                              IllegalArgumentException;
  Position<E> right(Position<E> p) throws
                              IllegalArgumentException;
  Position<E> sibling(Position<E> p) throws
                              IllegalArgumentException;
}
```

Binary Trees
ADT

- AbstractBinaryTree: extends AtstractTree and implements BinaryTree

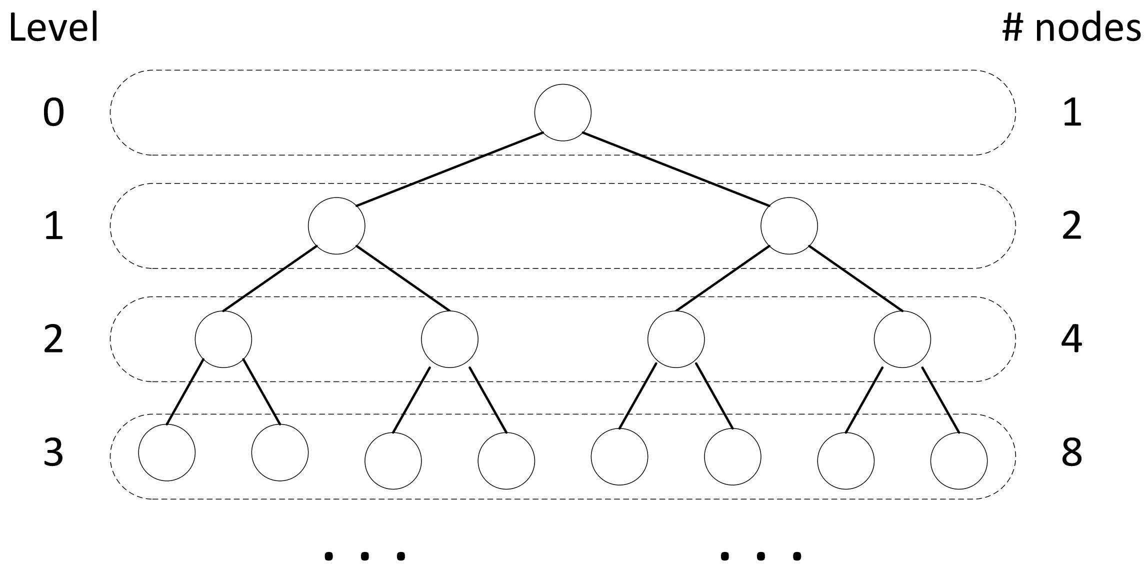
- Additional methods:
  - sibling
  - numChildren
  - children



- AbstractBinaryTree.java

Binary Trees
Binary Tree Properties

- Let *level d* of a binary tree *T* be the set of nodes at depth *d* of *T*.



- The maximum number of nodes at level d is 2^d.

Binary Trees

Binary Tree Properties

- n: the number of nodes in T
- n_E: the number of external nodes in T
- n_I: the number of internal nodes in T
- h: the height of T

- $h + 1 \le n \le 2^{h+1} - 1$
- $1 \le n_E \le 2^h$
- $h \le n_I \le 2^h - 1$
- $\log(n + 1) - 1 \le h \le n - 1$

Binary Trees

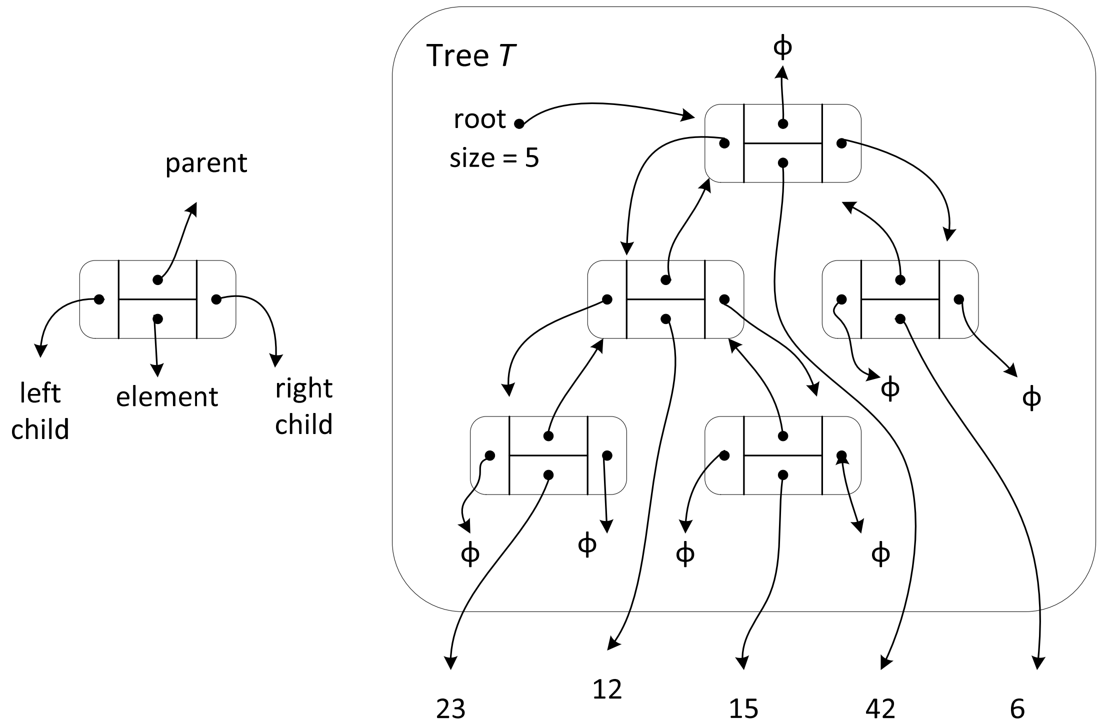
Binary Tree Properties

- If T is a proper binary tree:

- $2h + 1 \le n \le 2^{h+1} - 1$
- $h + 1 \le n_E \le 2^h$
- $h \le n_I \le 2^h - 1$
- $\log(n + 1) - 1 \le h \le (n - 1)/2$
- $n_E = n_I + 1$

Binary Trees
Implementation Using Linked Structure

- A node has the following linked structure.



Binary Trees
Implementation Using Linked Structure

- LinkedBinaryTree extends AbstractBinaryTree abstract class with the following update methods:

  - addRoot(*e*): Creates a new node with element *e* and make it the root of an empty tree. Returns the position of the root. An error occurs if the tree is not empty.
  - addLeft(*p*, *e*): Creates a new node with element *e* and make it a left child of position *p*. Returns the position of the new node (left child). An error occurs if *p* already has a left child.
  - addRight(*p*, *e*): Creates a new node with element *e* and make it a right child of position *p*. Returns the position of the new node (right child). An error occurs if *p* already has a right child.

Binary Trees
Implementation Using Linked Structure

- Update methods (continued):
  - set(*p*, *e*): Replaces the element of *p* with element *e.* Returns the previously stored element.
  - attach(*p*, T_1, T_2): Attaches internal structure of T_1 and T_2 as the left subtree and the right subtree, respectively, of a leaf node position *p* and resets T_1 and T_2 to empty trees. If *p* is not a leaf node, an error occurs.
  - remove(*p*): Removes the node at position *p*, replacing it with its child (if any). Returns the element that had been stored at *p*. An error occurs if *p* has two children.

Binary Trees
Implementation Using Linked Structure

- A node is a *position* (instance variables shown below)

```
protected static class Node<E> implements Position<E> {
  private E element;          // an element stored at this node
  private Node<E> parent;     // a reference to the parent node (if any)
  private Node<E> left;       // a reference to the left child (if any)
  private Node<E> right;      // a reference to the right child (if any)
```

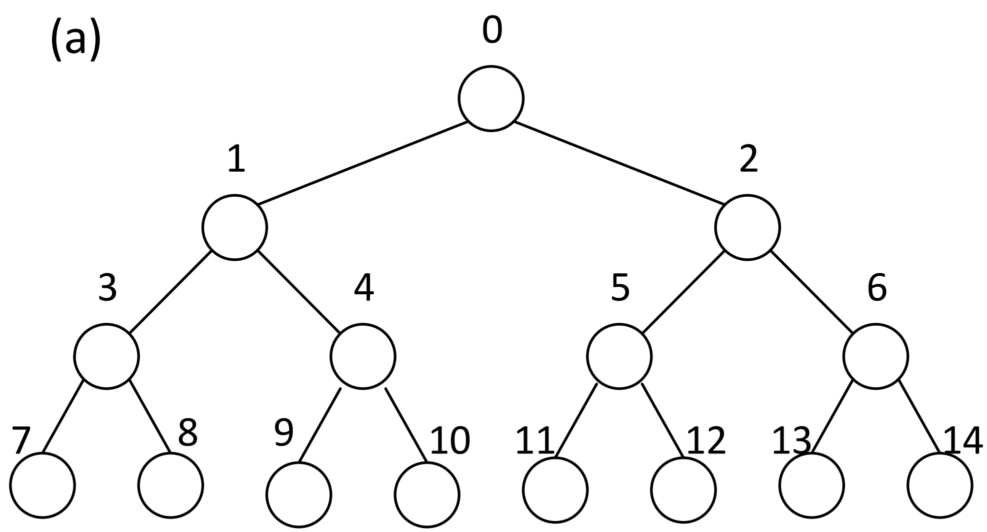
- LinkedBinaryTree has two instance variables

```
protected Node<E> root = null;
private int size = 0;
```

- LinkedBinaryTree.java

Binary Trees
Implementation Using Array

- Nodes are stored in an array.
- *Level numbering* scheme is used.

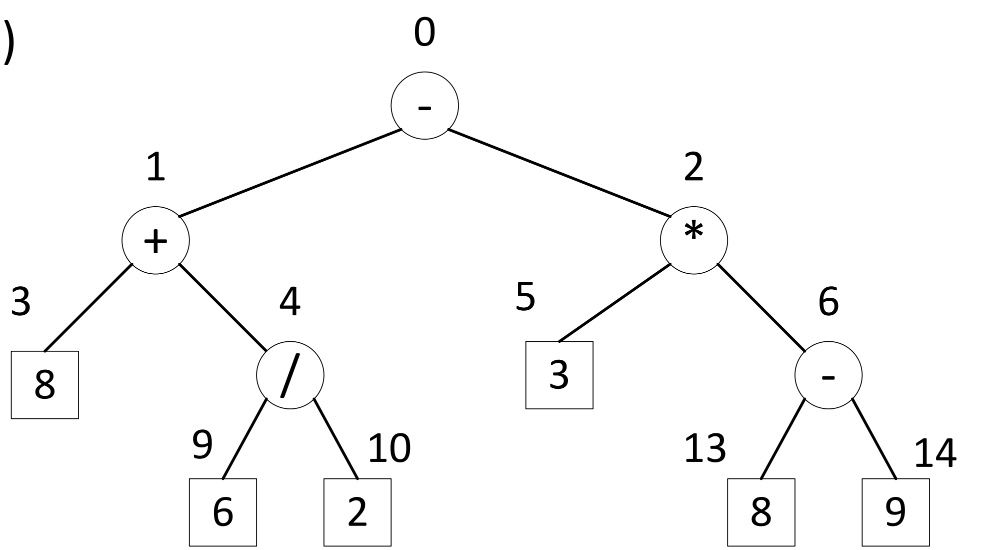


- A number above a node is the index in the array.

Binary Trees
Implementation Using Array

- Example

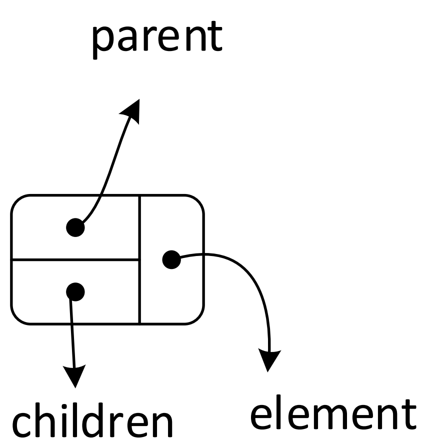
(b)

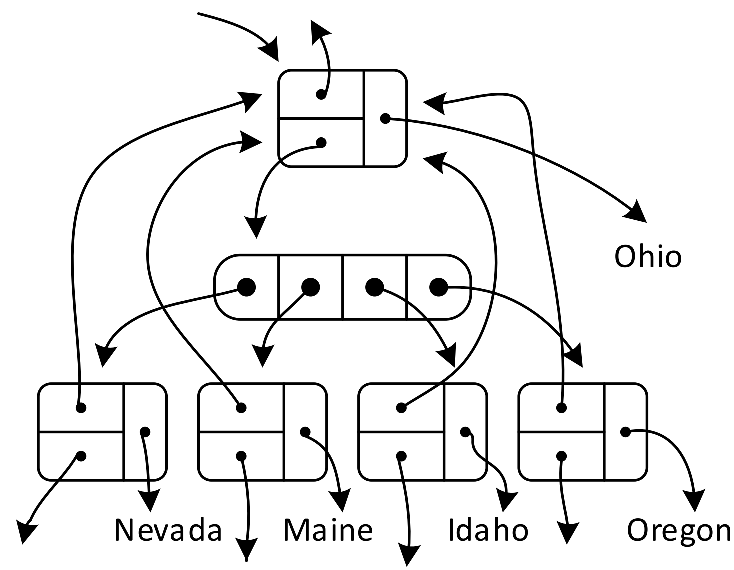


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-	+	*	8	/	3	-			6	2			8	9

Binary Trees

Linked Structure for General Trees





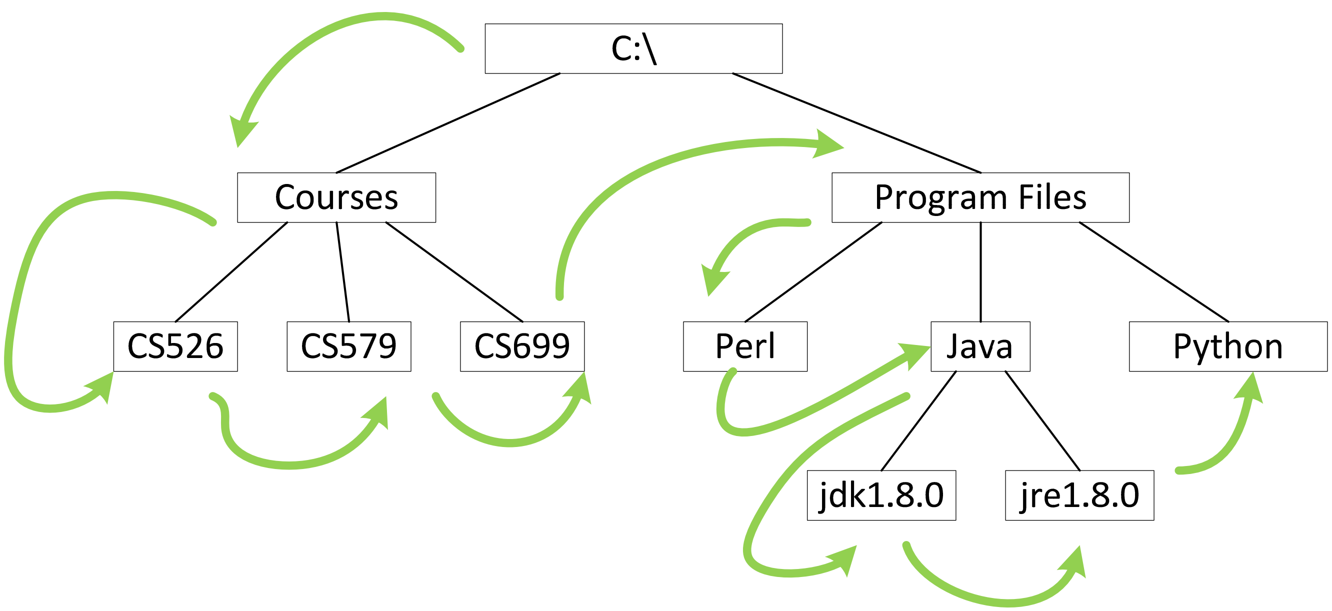
Binary Trees
Tree Traversal

- A *traversal* of a tree *T* is a systematic way of visiting all positions in *T*.
- Preorder tree traversal:
  – visit the root
  – visit all children

```
Algorithm preorder(p)
    visit p
    for each child c in children(p)
        preorder(c)
```

Binary Trees
Tree Traversal

- Preorder tree traversal illustration:



Binary Trees

Tree Traversal

- Postorder tree traversal:
  - Visit all children (recursively)
  - Visit the root

Algorithm postorder(*p*)

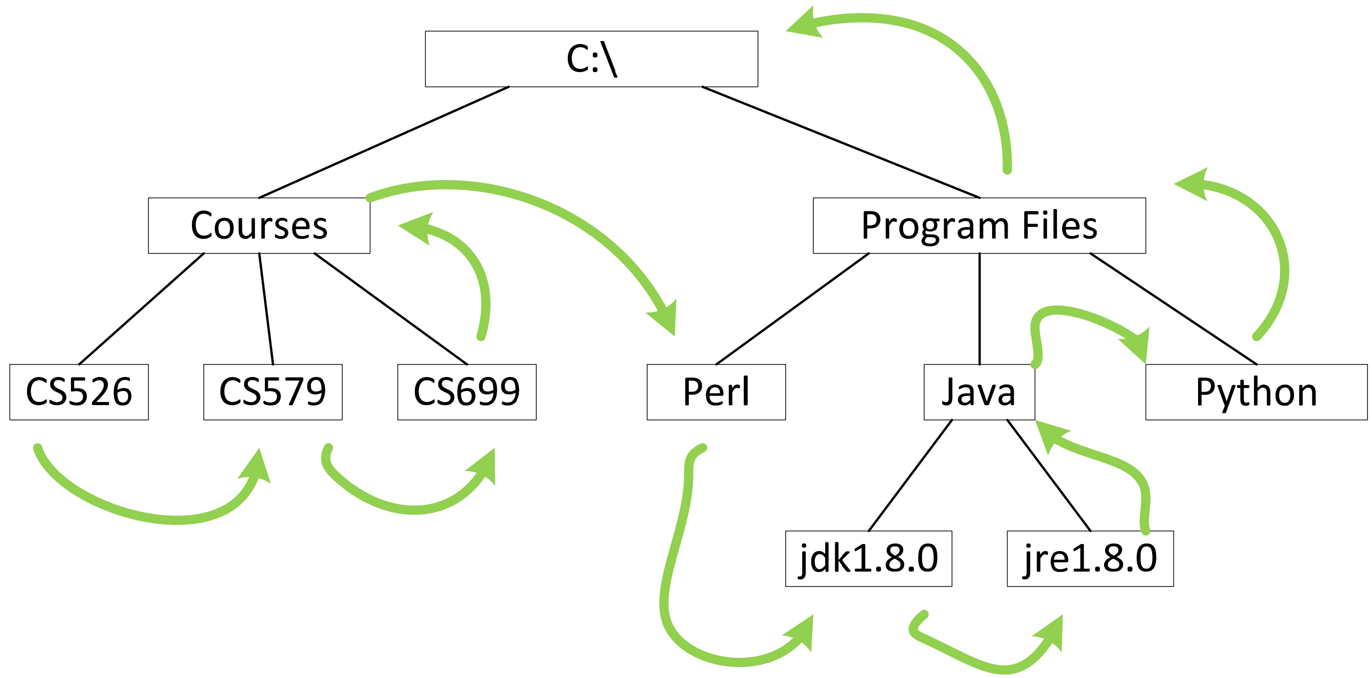
    for each child *c* in *children*(*p*)

        postorder(*c*)

    visit *p*

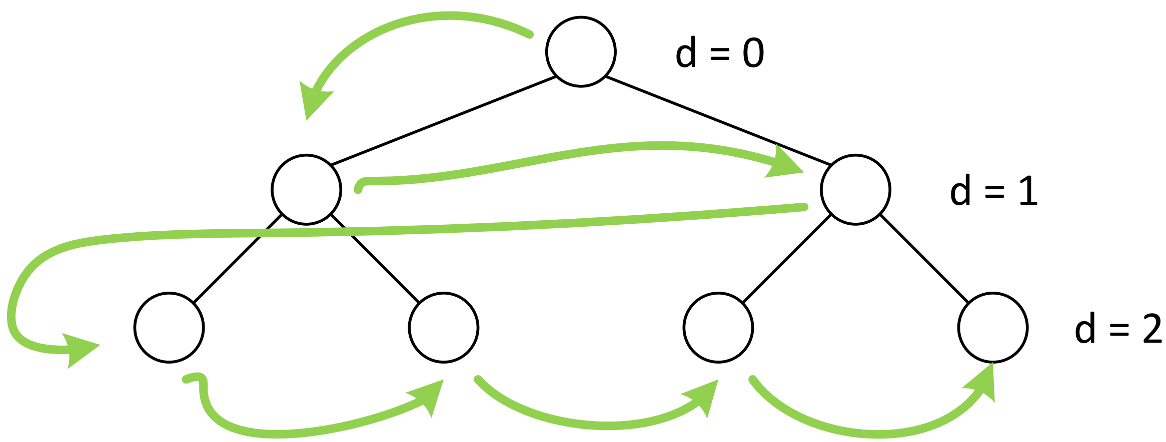
Binary Trees
Tree Traversal

- Postorder tree traversal illustration



Binary Trees
Tree Traversal

- Breadth-first tree traversal
  - Also called *breadth-first search* or *BFS*
  - Visits all positions at depth *d* before visiting positions at depth *d* + 1.



Binary Trees
Tree Traversal

- Breadth-first tree traversal (continued)

```
Algorithm breadthfirst( )
      initialize Q to contain the root of the tree
      while Q is not empty
         p = Q.dequeue( )  // remove the oldest entry in Q
         visit p
         for each child c in children(p)
            Q.enqueue(c)    // add all children of p to the rear of Q
```

- Running time
  - Each node is enqued and dequeued once each.
  - O(n)

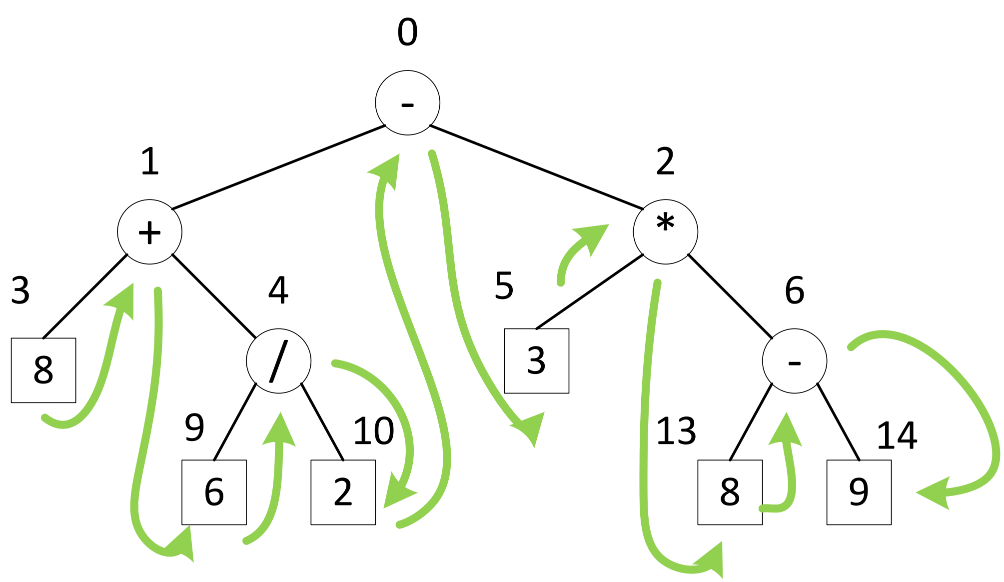
Binary Trees
Tree Traversal

- Inorder tree traversal of binary tree
  - Visit the left subtree
  - Visit the root
  - Visit the right subtree

```
Algorithm inorder(p)
    if p has a left child lc    // visit left subtree
        inorder(lc)
    visit p
    if p has a right child rc   // visit right subtree
        inorder(rc)
```

Binary Trees
Tree Traversal

- Inorder tree traversal of binary tree illustration:



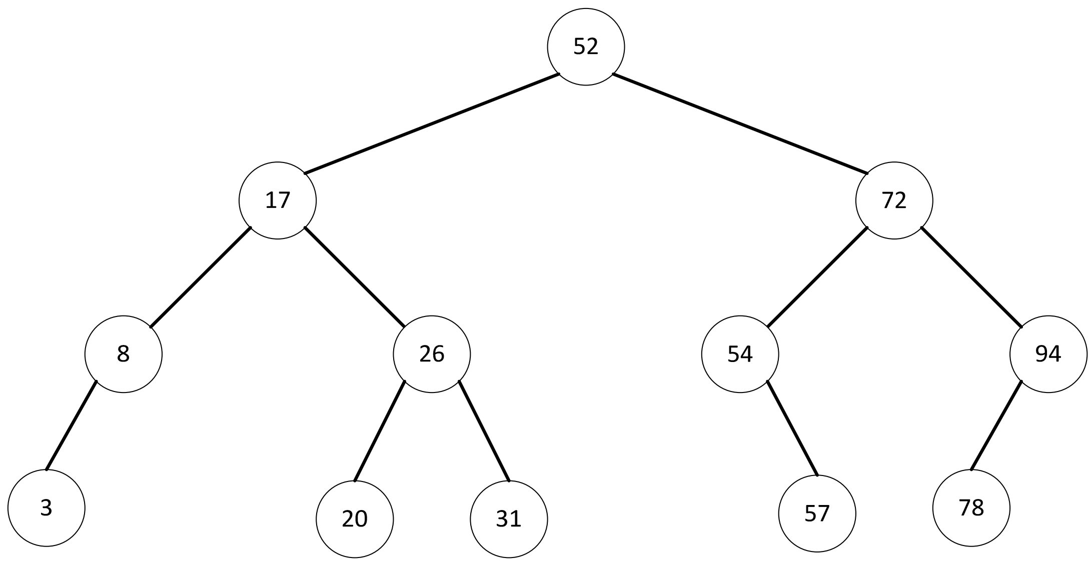
- Inorder tree traversal generates: 8 + 6 / 2 – 3 * 8 – 9
- Correct expression without parentheses

Binary Trees
Binary Search Tree

- A binary search tree is a binary tree with additional properties:
  - Each position p stores an element, denoted as $e(p)$.
  - All elements in the left subtree of a position p (if any) are less than $e(p)$.
  - All elements in the right subtree of a position p (if any) are greater than $e(p)$.

Binary Trees
Binary Search Tree

- A binary search tree example:



- Inorder tree traversal generates:

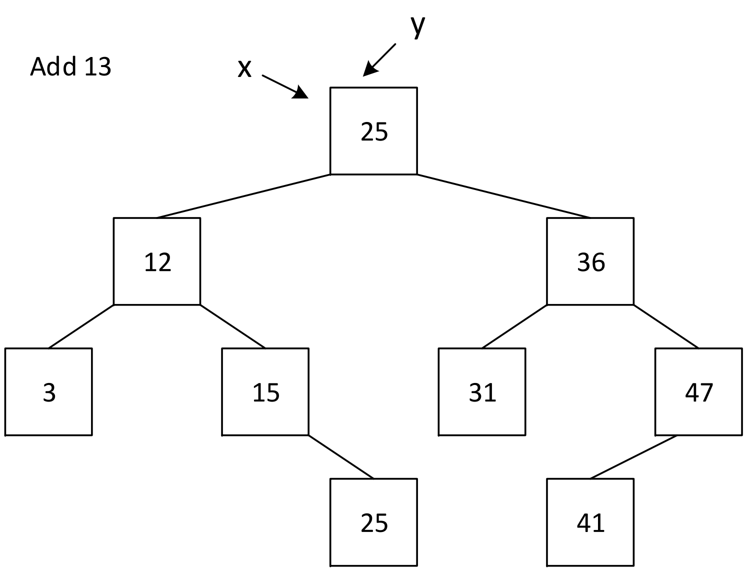
  3, 8, 17, 20, 26, 31, 52, 54, 57, 72, 78, 94

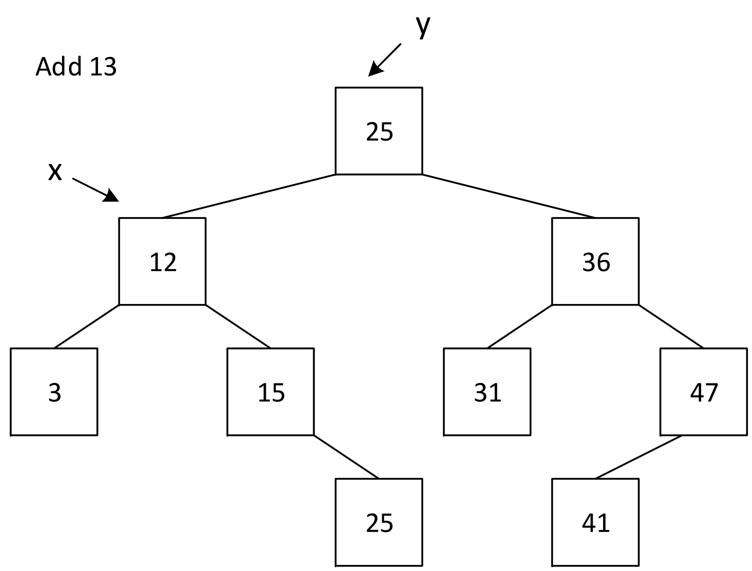
Binary Trees
Binary Search Tree

```
Algorithm add(p, e) // an incomplete code
    if p == null // this is an empty tree
        create a new node with e and make it the root of the tree
    x = p; y = x; // y follows x
    while (x is not null) {
      if (the element of x) is the same as e, return null
      else if (the element of x) > e{
            y = x;        x = left child of x;
        }
        else {
            y = x;        x = right child of x;
        }
    } // end of while
    // add a new node here
```

Binary Trees
Binary Search Tree

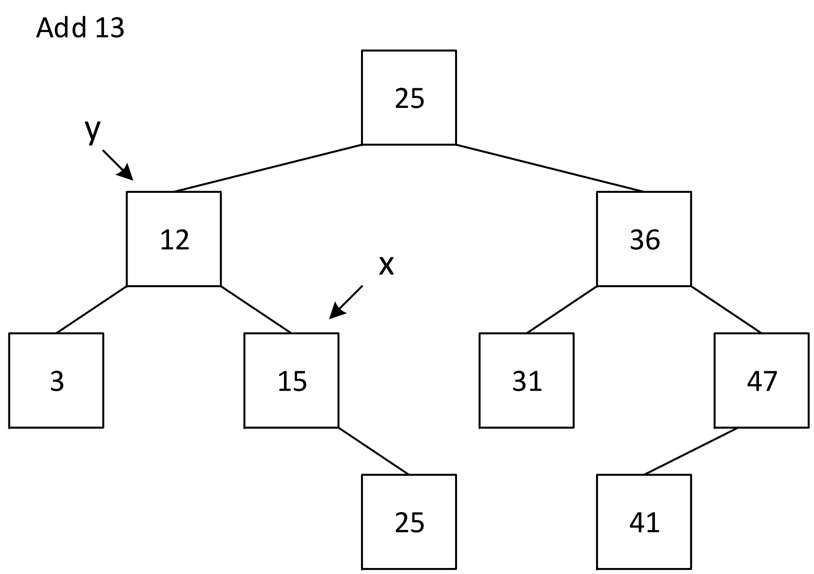
add(root, 13)

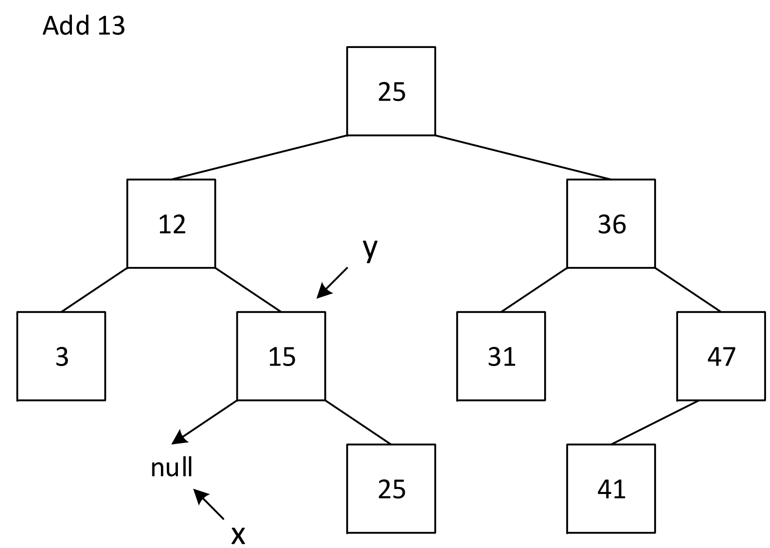




Binary Trees
Binary Search Tree

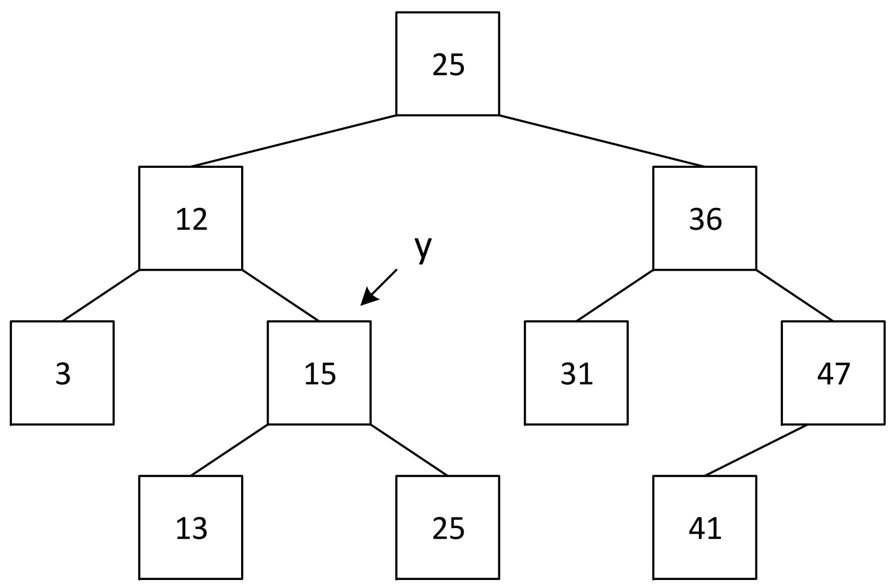
add(root, 13)





Binary Trees
Binary Search Tree

add(root, 13)



References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.