Data Structures and Algorithms

Lesson 5: *Make Friends with Trees*

Binary trees, binary search trees, sets, tree iterators

Outline

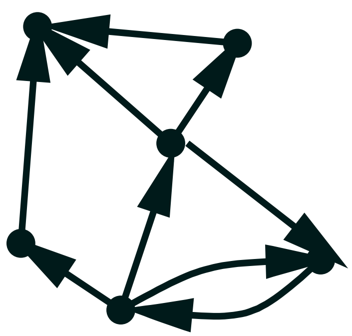
1. **Trees**
2. Binary Trees
   - Implementing Binary Trees
3. Binary Search Trees
   - Definition
   - Implementing a Set
4. Tree Iterators

Trees

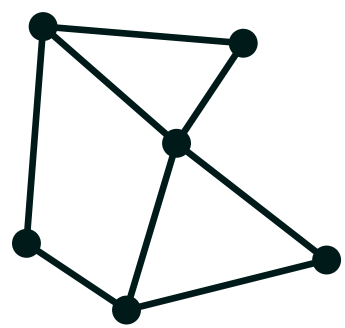
- Trees are one of the major ways of structuring data
- They are used in a vast number of data structures
  - ⋆ Binary search trees
  - ⋆ B-trees
  - ⋆ splay trees
  - ⋆ heaps
  - ⋆ tries
  - ⋆ suffix trees
- We shall cover most of these

Defining Trees

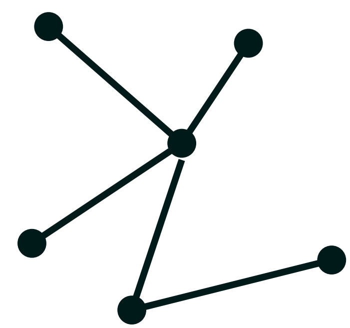
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges have no ”direction”
  - ⋆ **acyclic**: there are no cycles in the graph



graph



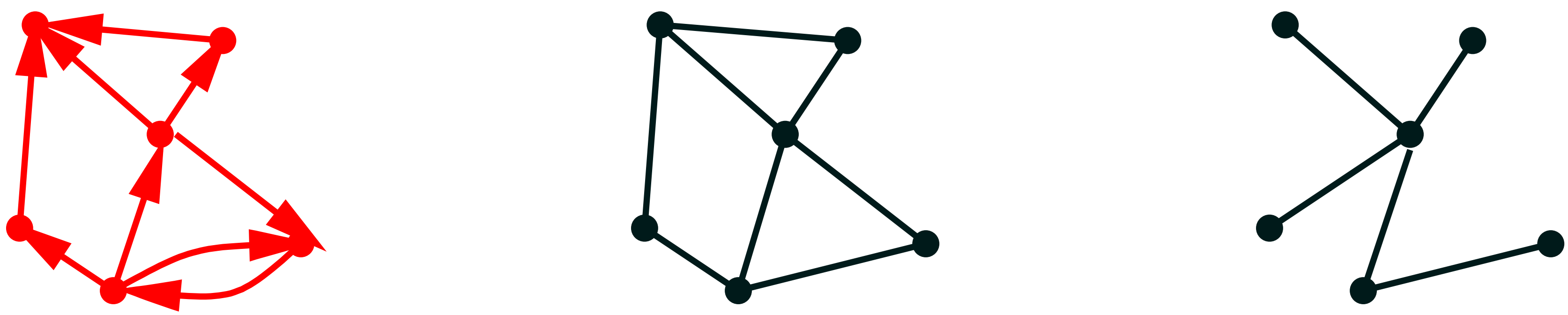
undirected graph



tree = acyclic undirected graph

Defining Trees

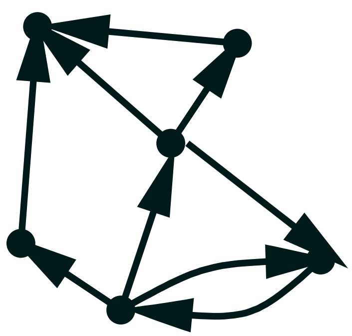
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges have no "direction"
  - ⋆ **acyclic**: there are no cycles in the graph



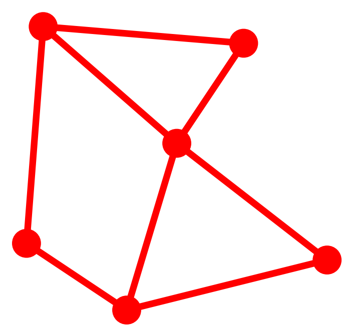
graph undirected graph tree = acyclic undirected graph

Defining Trees

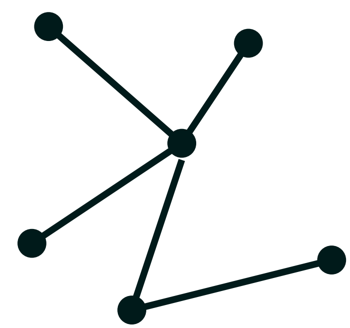
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges have no "direction"
  - ⋆ **acyclic**: there are no cycles in the graph



graph



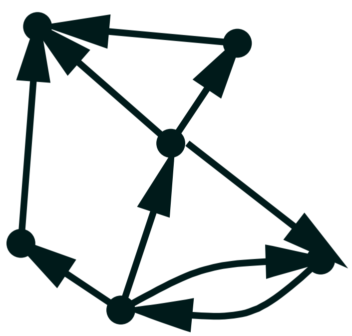
undirected graph



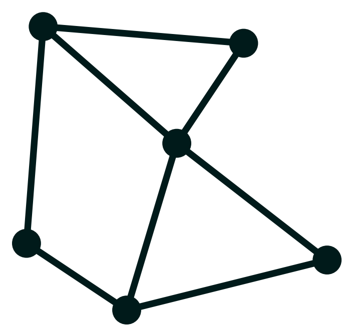
tree = acyclic undirected graph

Defining Trees

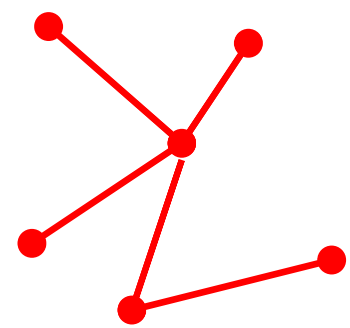
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges have no ”direction”
  - ⋆ **acyclic**: there are no cycles in the graph



graph



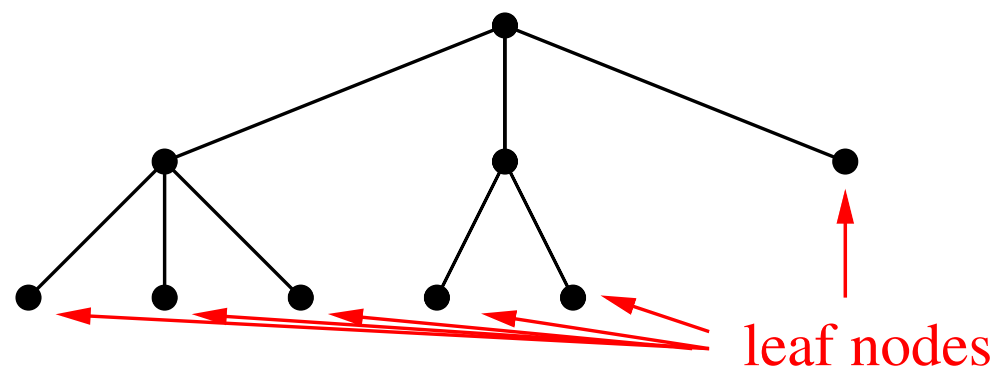
undirected graph



tree = acyclic undirected graph

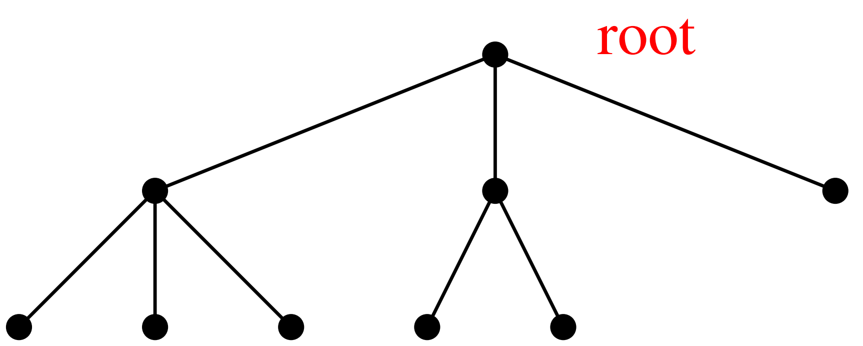
Borrowing from Nature

- We often impose an ordering on the nodes (or a direction on the edges) – known as a rooted tree
- Borrowing from nature, we recognise one node as the **root** node
- Nodes have **children** nodes living immediately beneath them
- Each node has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes



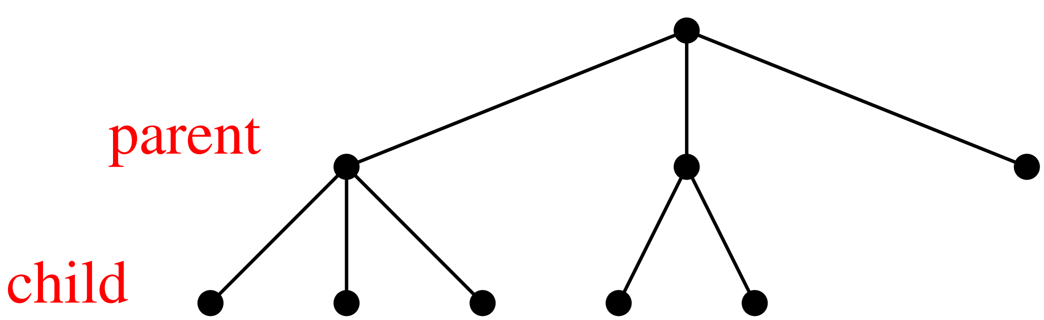
Borrowing from Nature

- We often impose an ordering on the nodes (or a direction on the edges) – known as a rooted tree
- Borrowing from nature, we recognise one node as the **root** node
- Nodes have **children** nodes living immediately beneath them
- Each node has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes



Borrowing from Nature

- We often impose an ordering on the nodes (or a direction on the edges) – known as a rooted tree
- Borrowing from nature, we recognise one node as the **root** node
- Nodes have **children** nodes living immediately beneath them
- Each node has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes

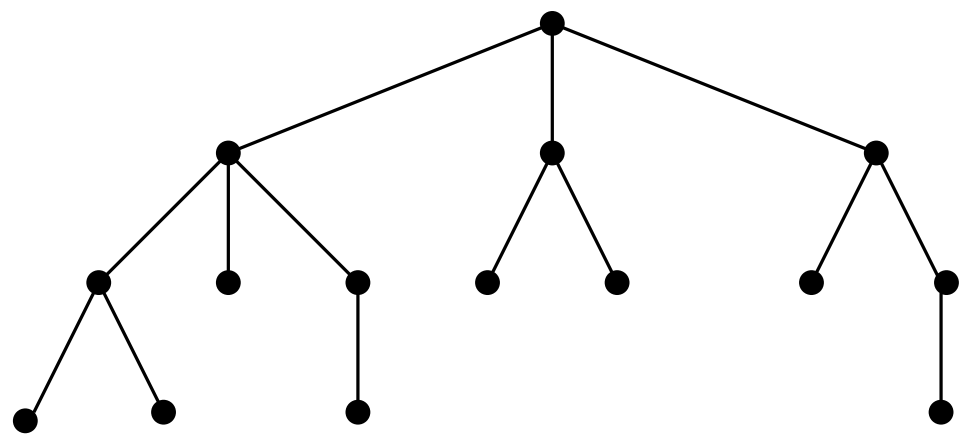


Spot the Error

- One small inconsistency with biology . . .
- . . . computer scientists draw there trees upside down

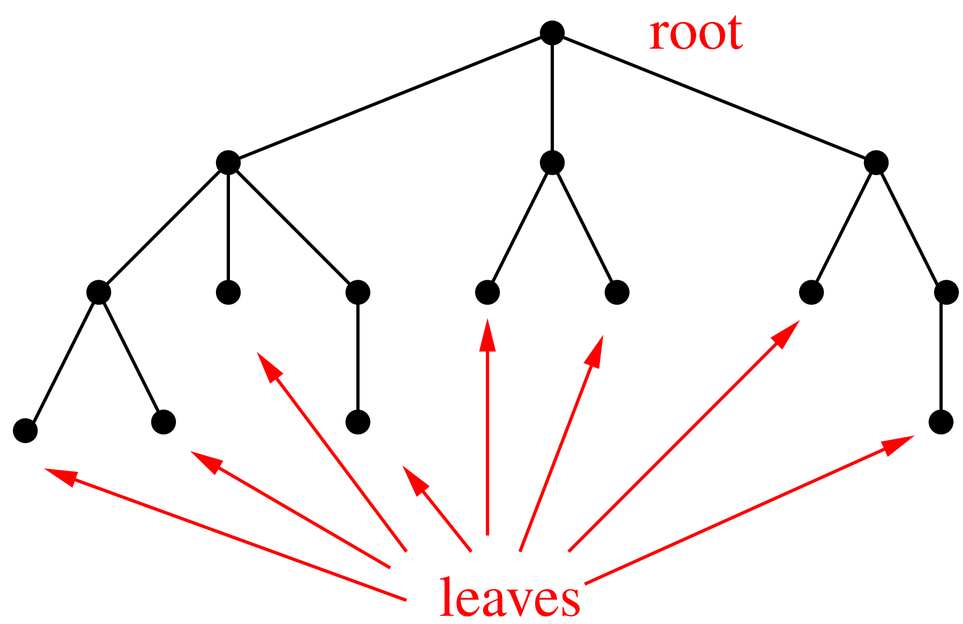
Spot the Error

- One small inconsistency with biology . . .
- . . . computer scientists draw there trees upside down
  - ⋆ root at the top
  - ⋆ leaves at the bottom



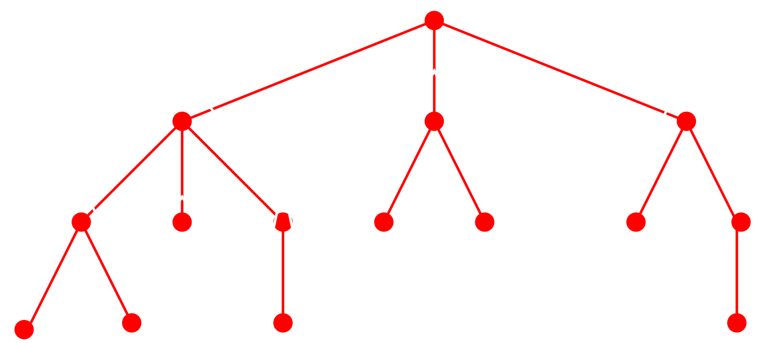
Spot the Error

- One small inconsistency with biology . . .
- . . . computer scientists draw there trees upside down
  - ⋆ root at the top
  - ⋆ leaves at the bottom



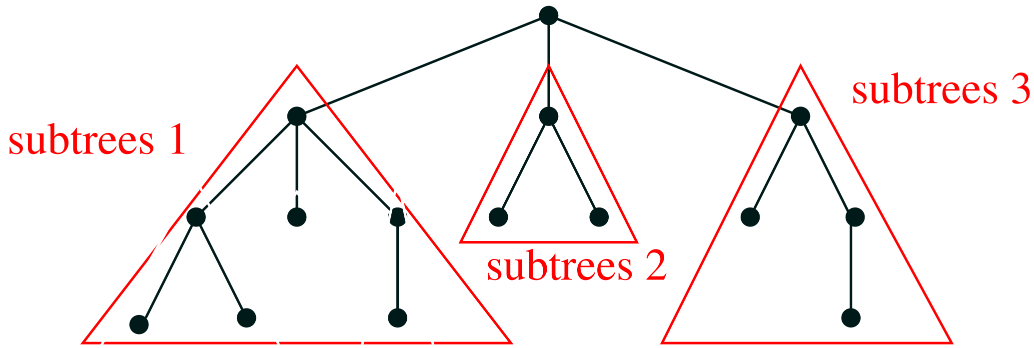
Subtrees

- We can think of a tree as being made up of **subtrees** (plus the root)



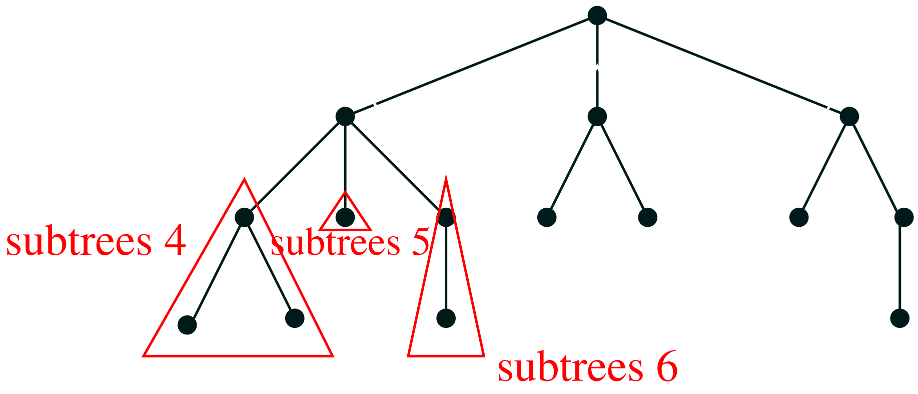
Subtrees

- We can think of a tree as being made up of **subtrees** (plus the root)



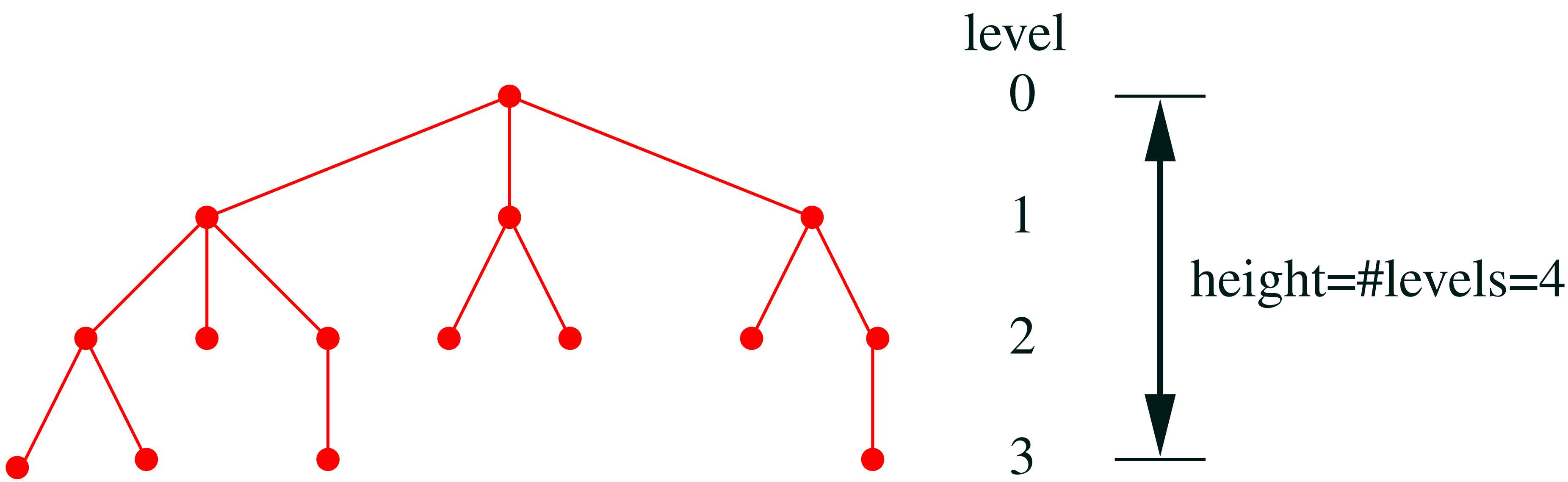
Subtrees

- We can think of a tree as being made up of **subtrees** (plus the root)



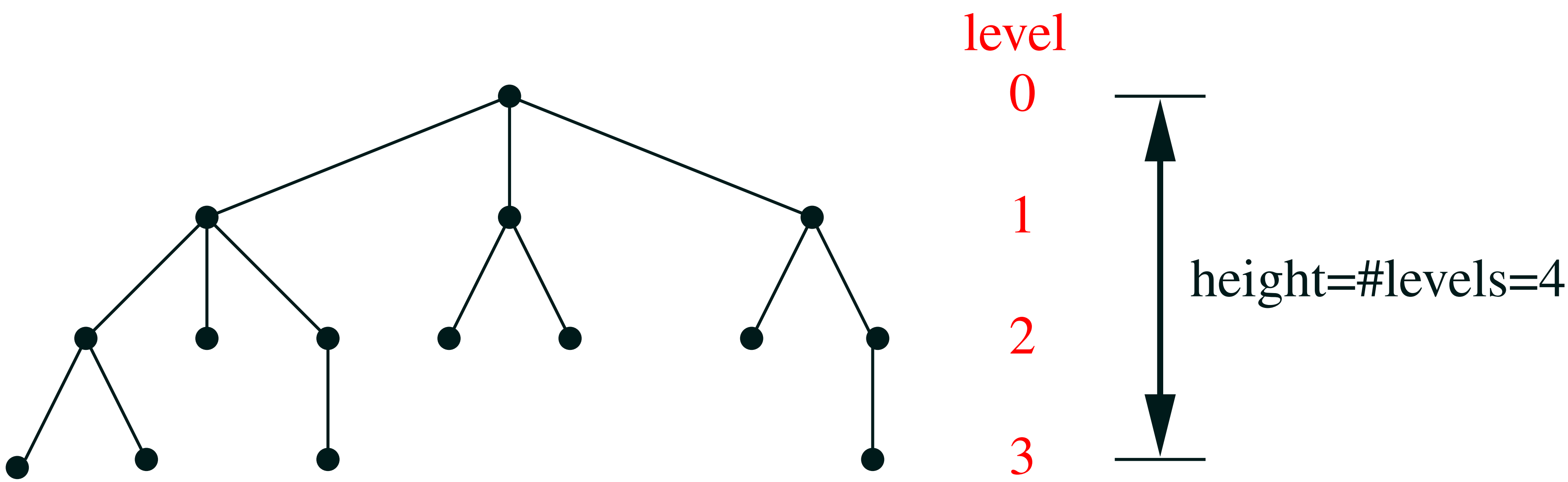
Level of Nodes

- It is useful to label different levels of the tree
- We take the **level** of a node in a tree as its distance from the root
- We take the **height** of a tree to be the number of levels



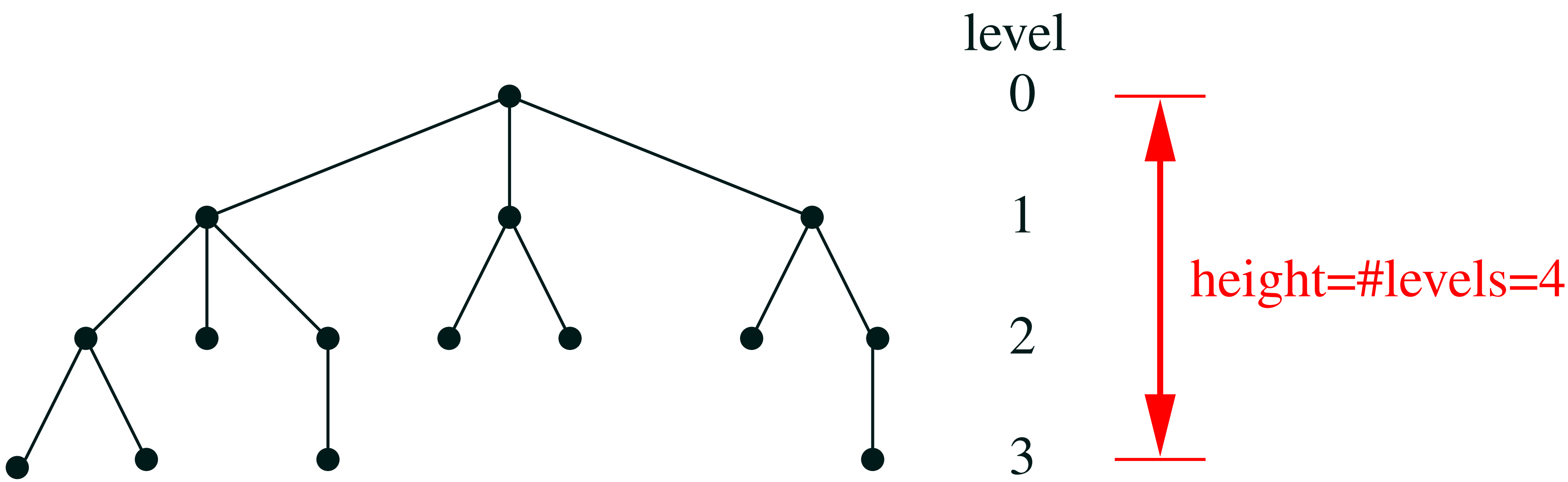
Level of Nodes

- It is useful to label different levels of the tree
- We take the **level** of a node in a tree as its distance from the root
- We take the **height** of a tree to be the number of levels



Level of Nodes

- It is useful to label different levels of the tree
- We take the **level** of a node in a tree as its distance from the root
- We take the **height** of a tree to be the number of levels



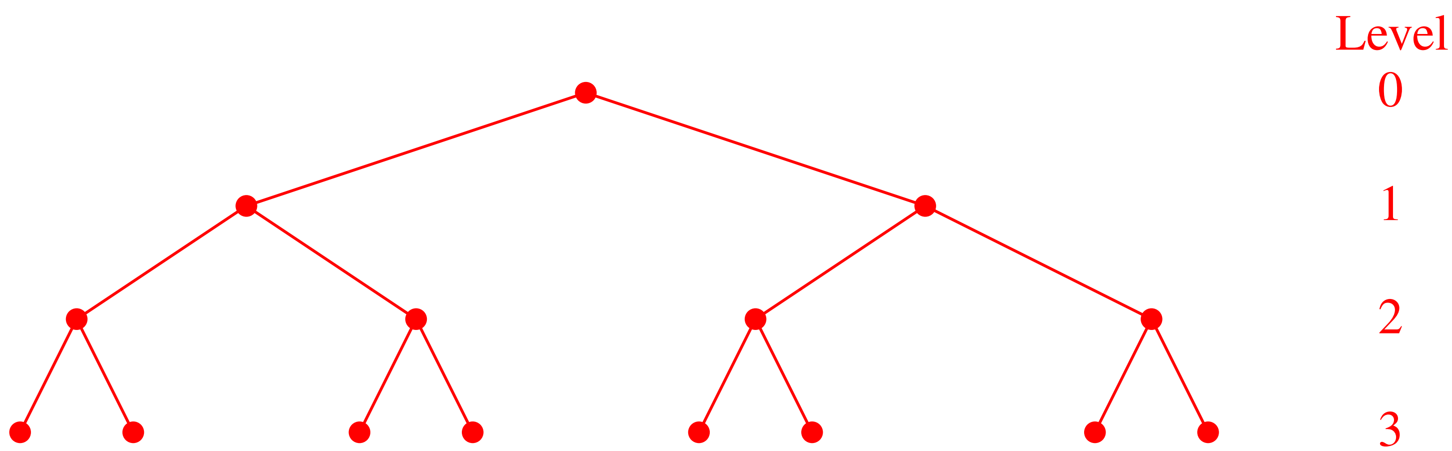
Outline

1. Trees
2. **Binary Trees**
   - Implementing Binary Trees
3. Binary Search Trees
   - Definition
   - Implementing a Set
4. Tree Iterators

Binary Trees

- A **binary tree** is a tree where each node can have zero, one or two children
- The total number of possible nodes at level l is 2^l
- The total number of possible nodes of a tree of height h is

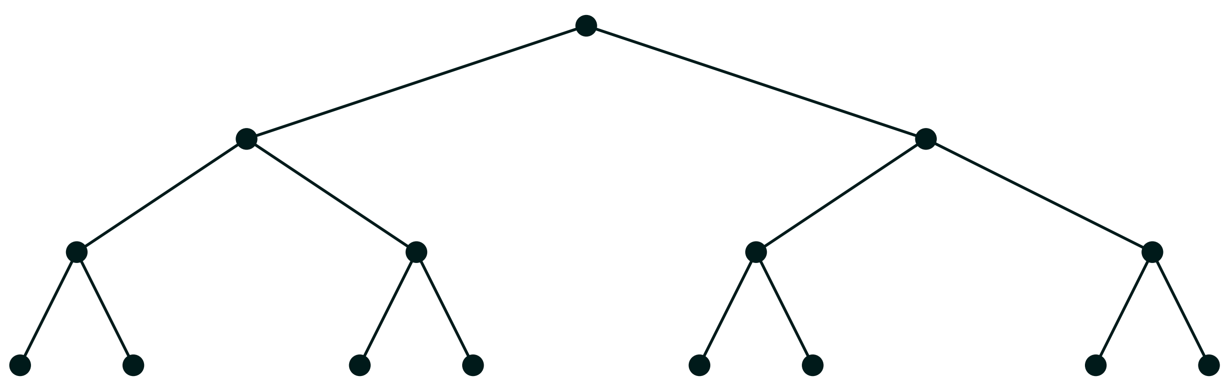
$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$



Binary Trees

- A **binary tree** is a tree where each node can have zero, one or two children
- The total number of possible nodes at level l is 2^l
- The total number of possible nodes of a tree of height h is

$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$

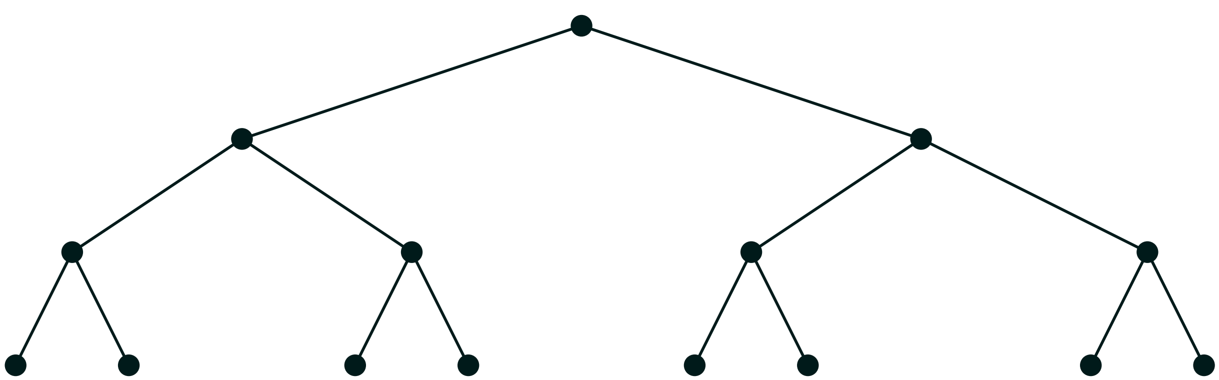


Level	# Nodes
0	1
1	2
2	4
3	8

Binary Trees

- A **binary tree** is a tree where each node can have zero, one or two children
- The total number of possible nodes at level l is 2^l
- The total number of possible nodes of a tree of height h is

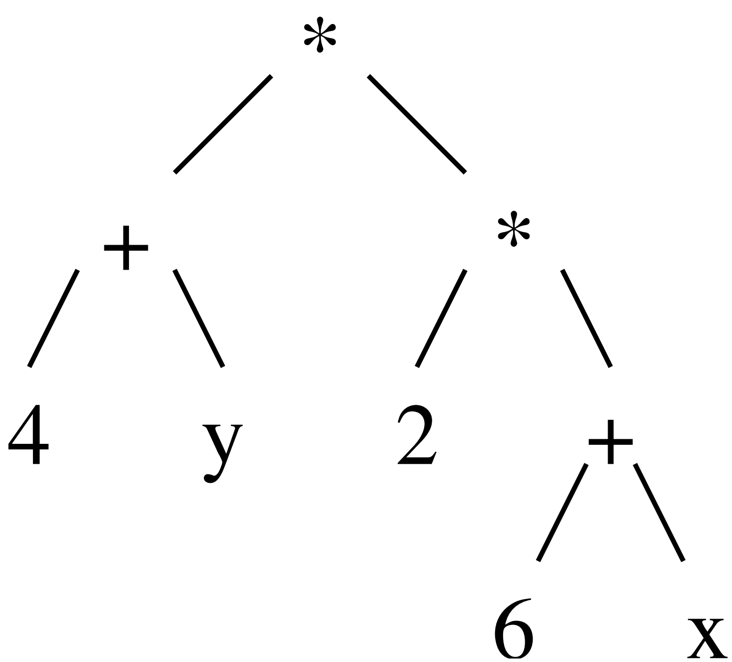
$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$



Level	# Nodes
0	1
1	2
2	4
3	8
	15

Uses of Binary Trees

- Binary trees have a huge number of applications
- For example, they are used as **expression trees** to represent expressions



$$(4+y)*(2*(6+x))$$

Implementation

- We wish to build a generic binary tree class with each node housing an element
- Again we use a `Node<T>` class as the building block for our data structure – in this case a node of the tree
- The `Node<T>` class will contain a reference to left and right children
- To help navigate the tree each node will contain a reference to its parent

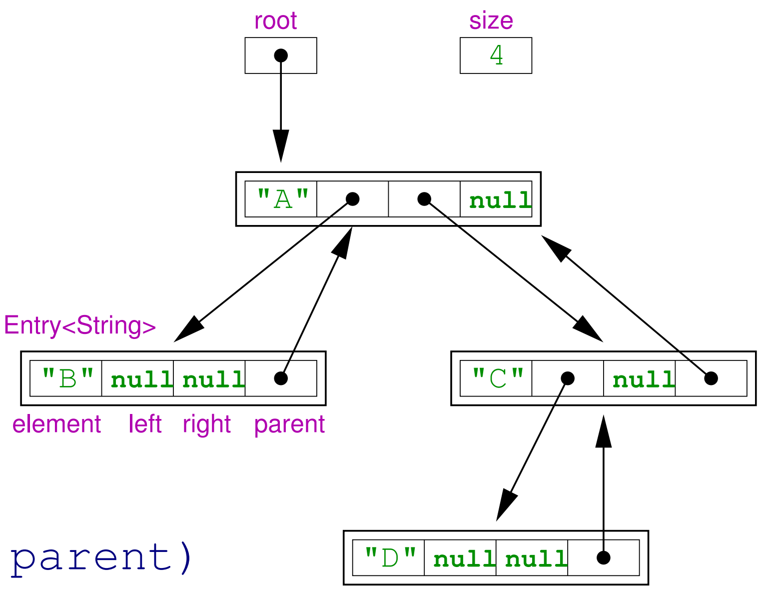
Java Code

```
public class BinaryTree<E>
{
    private Node<E> root;
    private int size;

    private static class Node<T>
    {
        private T element;
        private Node<T> left = null;
        private Node<T> right = null;
        private Node<T> parent;

        private Node(T element, Node<T> parent)
        {
            this.element = element;
            this.parent = parent;
        }
    }

}
```



Outline

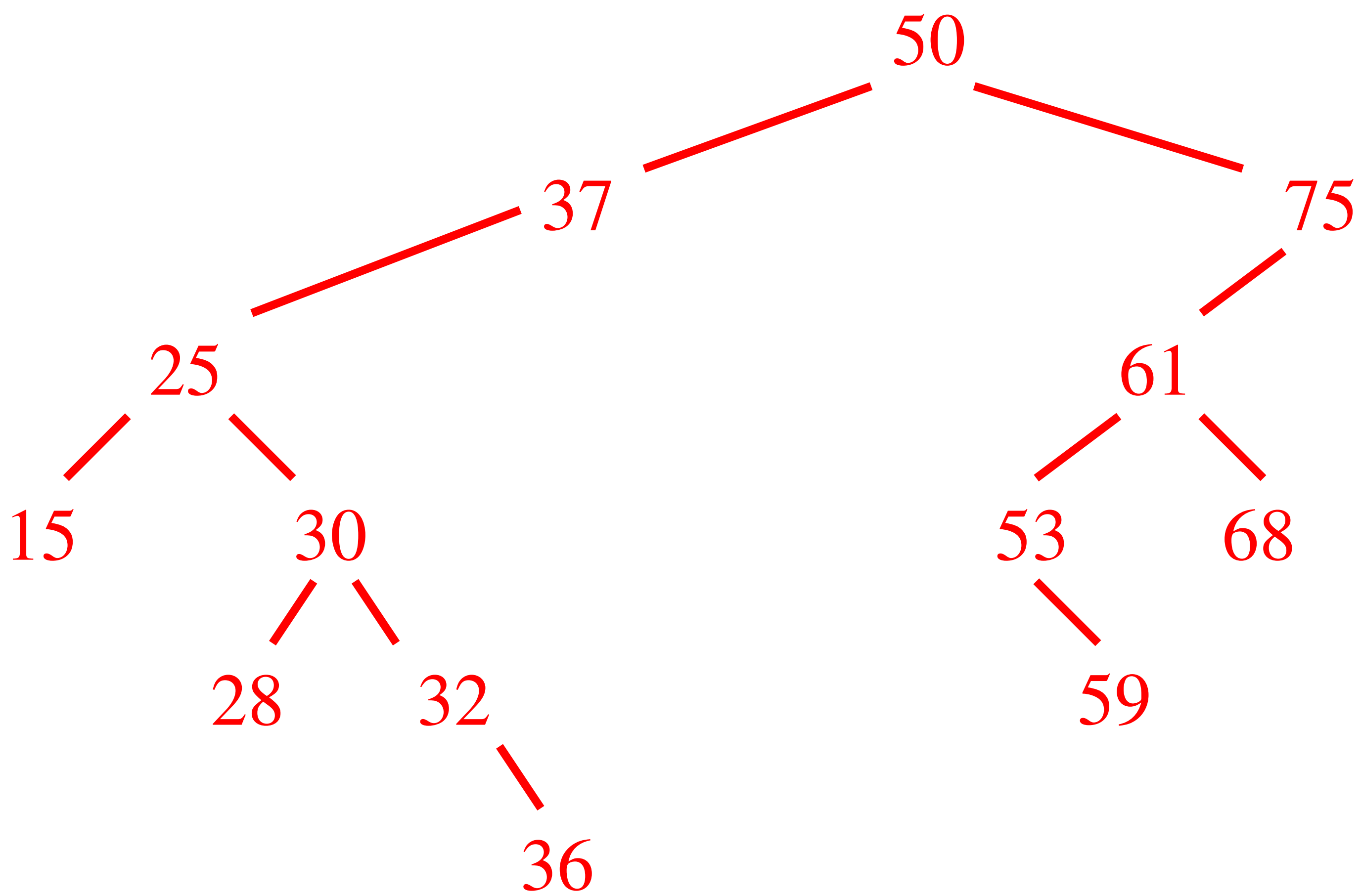
1. Trees
2. Binary Trees
   - Implementing Binary Trees
3. **Binary Search Trees**
   - Definition
   - Implementing a Set
4. Tree Iterators

Binary Search Trees

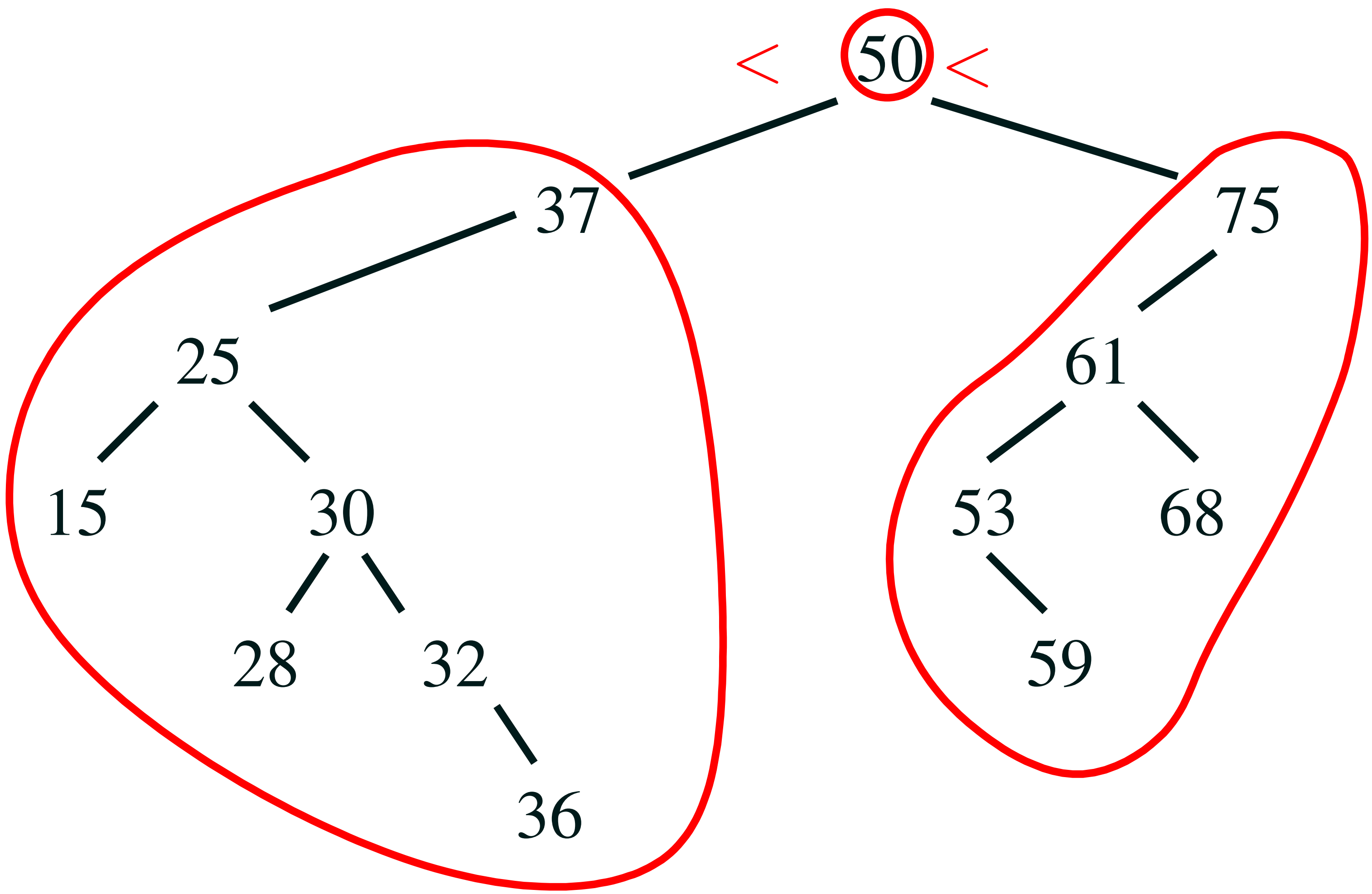
- We will concentrate on one of the most important binary trees, namely the **binary search tree**
- The binary search tree keeps the elements ordered
- We can define a binary search tree recursively

1. Each element in the left subtree is less than the root element
2. Each element in the right subtree is greater than the root element
3. Both left and right subtrees are binary search trees

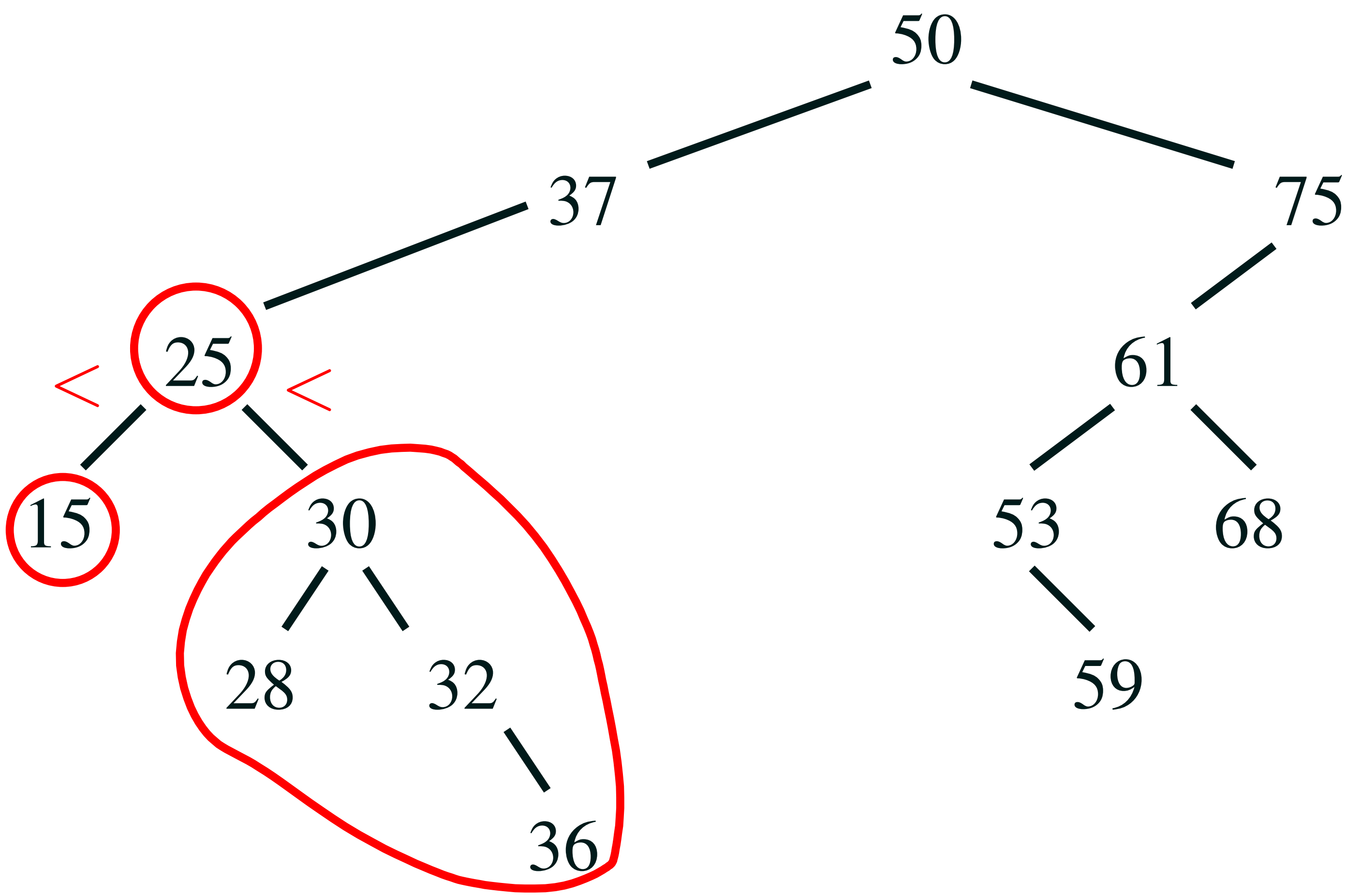
Example Binary Search Tree



Example Binary Search Tree

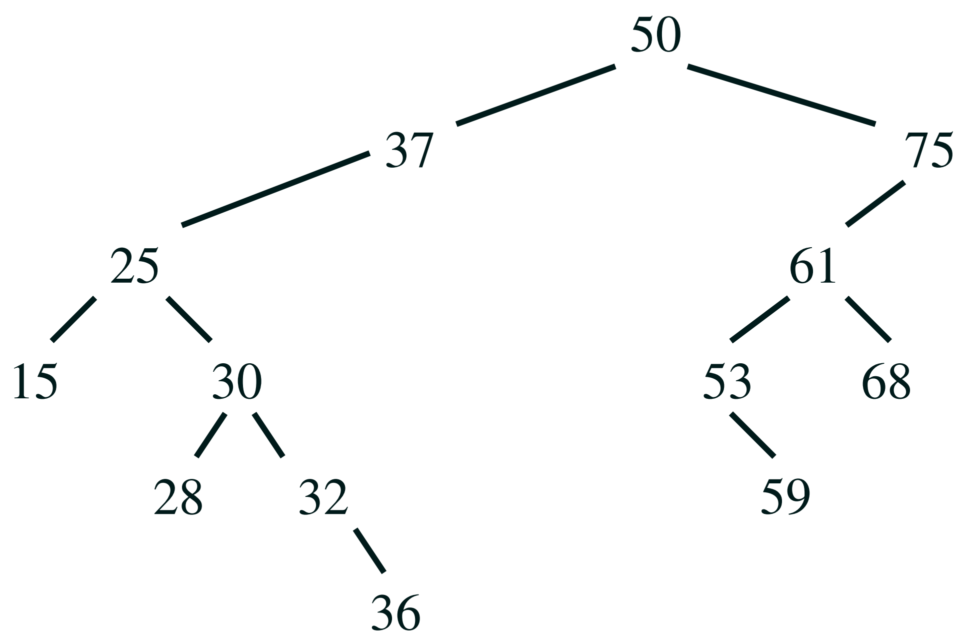


Example Binary Search Tree



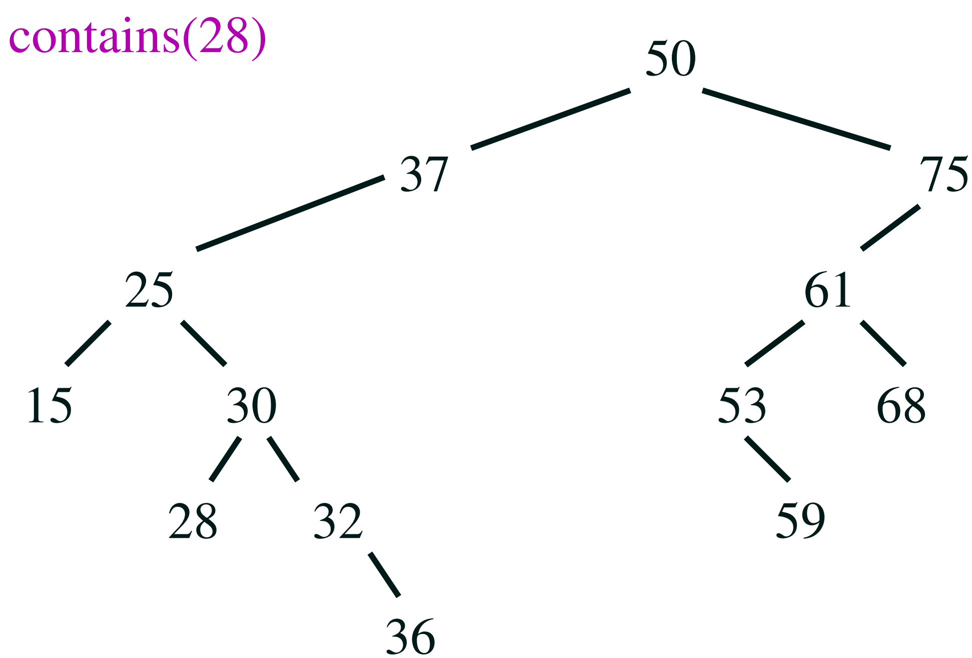
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



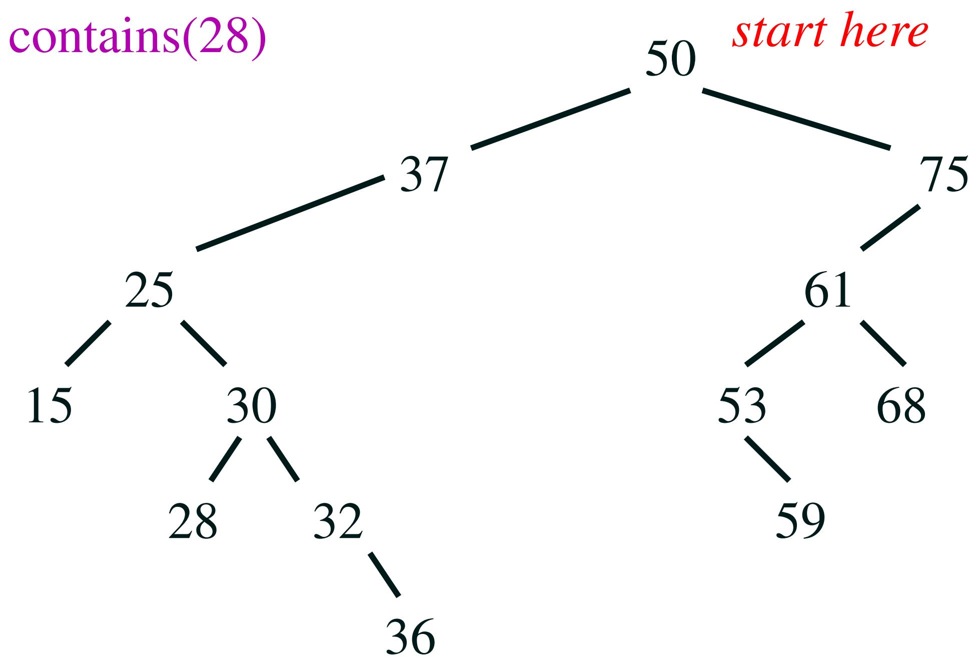
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



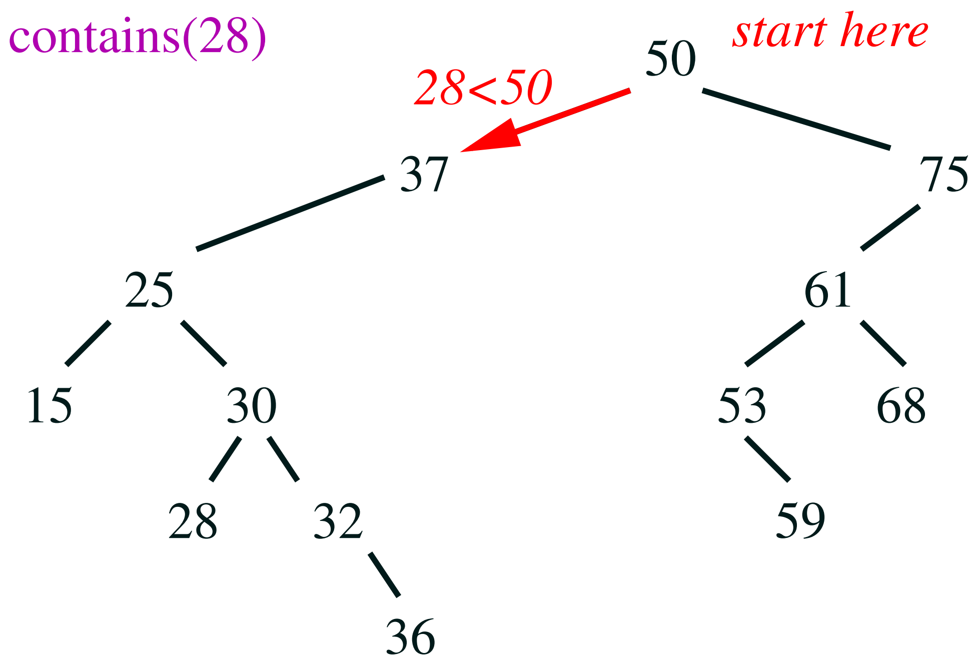
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



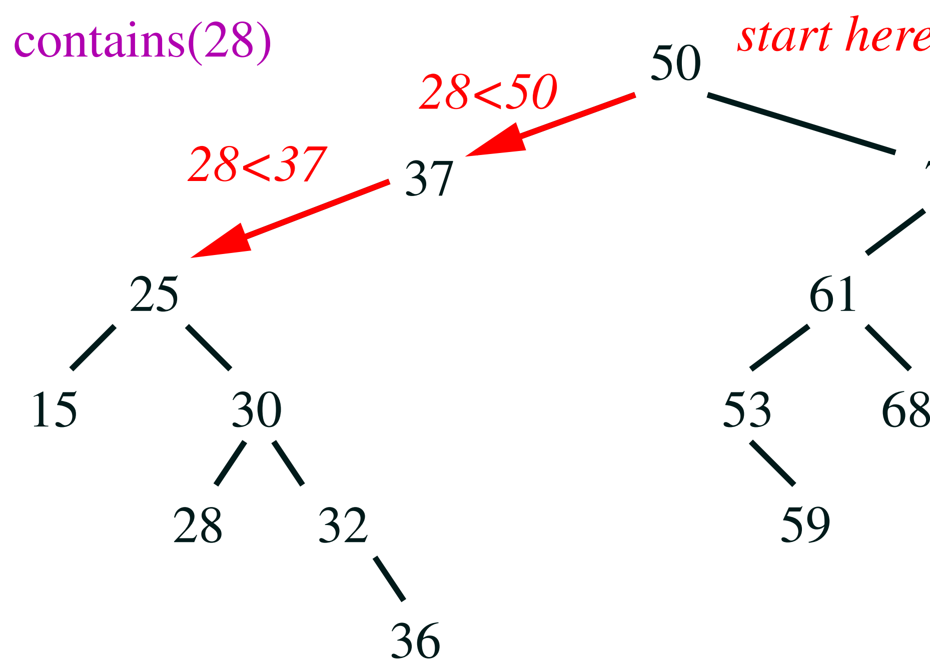
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



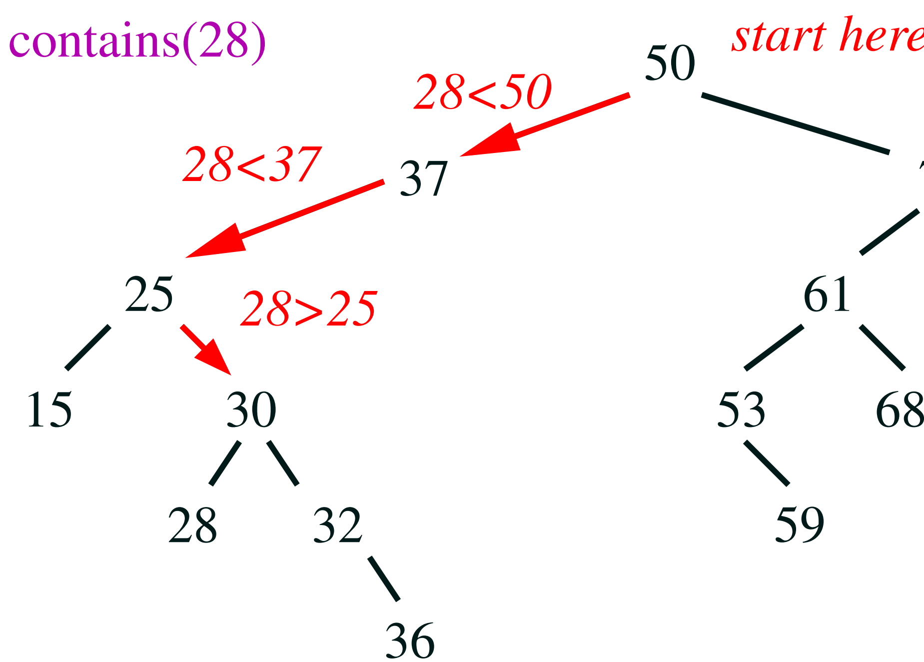
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



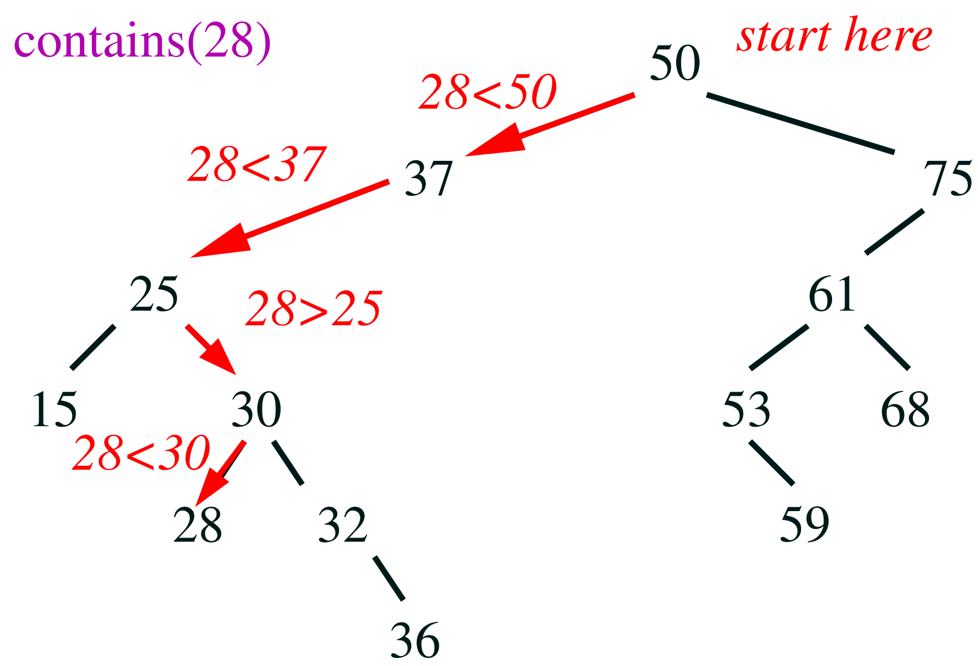
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



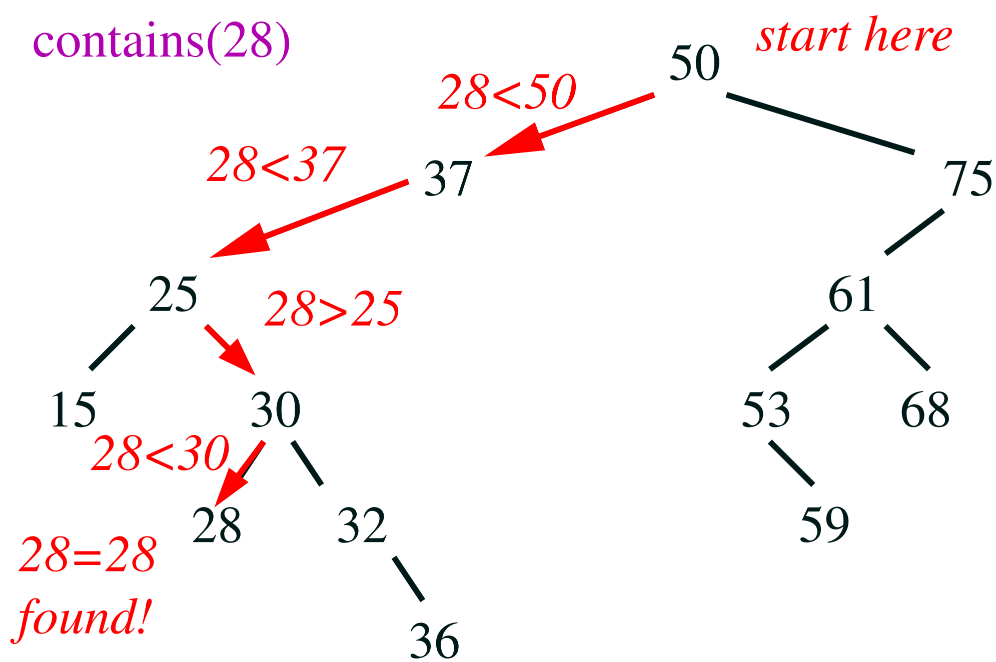
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found



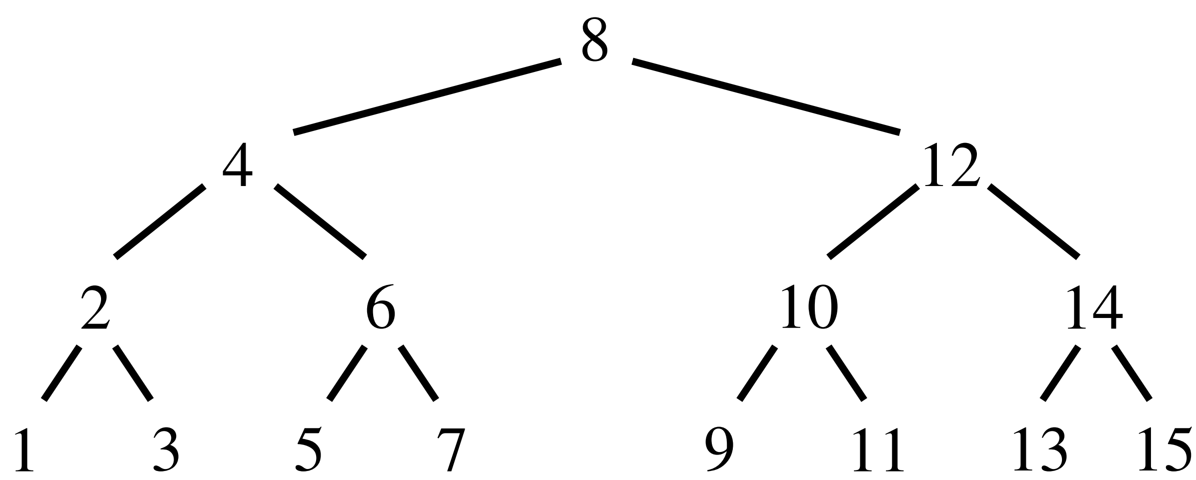
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - ⋆ If less than element go left
  - ⋆ If greater than element go right
  - ⋆ If equal to element found

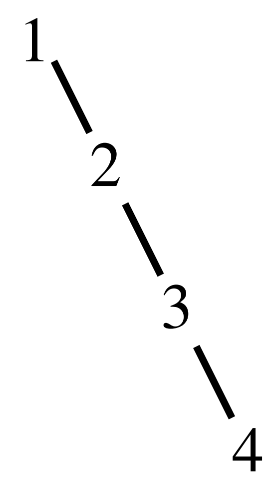


Speed of Search

- The number of comparisons necessary to find an element in a binary tree depends on the level of the node in the tree
- The worst case number of comparisons is therefore the height of the tree
- This depends on the density of the tree



full tree



sparse tree

Inorder Tree Walk

- We can print the elements in a binary search tree in sorted order using a simple recursive algorithm:

```
public void print(Node<E> e)
{
  if (e != null)
  {
    print(e.left);
    System.out.println(e.element);
    print(e.right);
  }
}
```

- For an n-node tree, the above algorithm takes $\Theta(n)$ time.
- The above algorithm is useful even if our binary tree is not a binary search tree – e.g. when the tree is an expression tree (see slide 11) !

Inorder Tree Walk – A Non-recursive Algorithm

- idea: use a stack to remember where we are

Inorder Tree Walk – A Non-recursive Algorithm

- idea: use a stack to remember where we are
- ```
  Node<E> e = root;
  Stack s = new Stack();
  while(e != null || s.hasElement()){
    if (e != null){
      s.push(e);
      e = e.left;
    }
    else{
      e = s.pop();
      System.out.println(e.element);
      e = e.right;
    }
  }
  ```
  ```

Implementing a Set

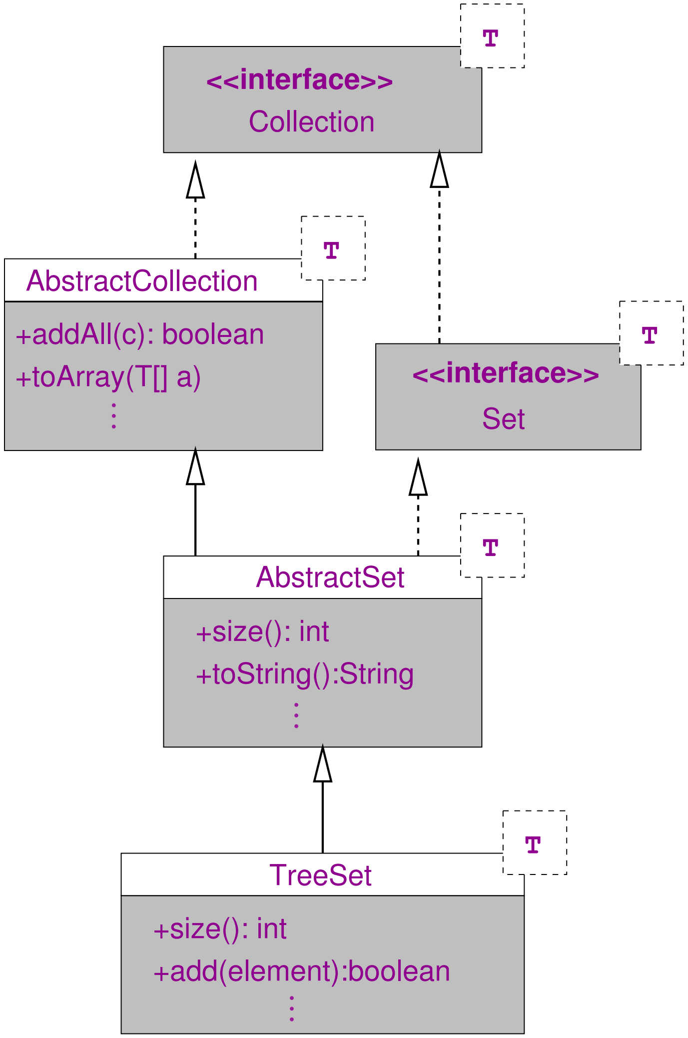
- A set is a fundamental **abstract data type**
- It is a collection of things with no repetition and no order
- Ironically because order doesn't matter we can order the elements

$$\{1, 3, 5, 5, 3, 4\} = \{5, 3, 4, 1\} = \{1, 3, 4, 5\}$$

- This allows rapid search – a feature we care about
- Binary trees are one of the efficient ways of implementing a set

Fitting In

- We can make our binary search tree class part of the Collections framework
- To do so we have to implement
  - ⋆ Constructors
  - ⋆ `size()`
  - ⋆ `add(T o)`
  - ⋆ `contains(Object o)`
  - ⋆ `remove(Object o)`
  - ⋆ `iterator()`



Comparable

- To sort any objects they must be comparable
- In Java this means they have to implement the `Comparable<T>` interface

```java
public interface Comparable<T>
{
   public int compareTo(T o);
}
```

- where `x.compareTo(y)` returns
  - ⋆ **0** if $x = y$
  - ⋆ **a negative integer** if x precedes y
  - ⋆ **a positive integer** if x succeeds y

Find an Element

- One of the core operations of a binary tree is to find a node

```java
private Node<E> getNode(Object obj)
{
    Node<E> e = root;
    while (e != null) {
        int comp = ((Comparable)(obj)).compareTo(e.element);
        if (comp == 0)
            return e;
        else if (comp < 0)
            e = e.left;
        else
            e = e.right;
    }
    return null;
}
```

Contains

- The idea is to navigate down the tree until either you reach the node or you reach a null

- We can use `getNode` to implement `contains`

```
public boolean contains(Object obj)
{ return getNode(obj) != null; }
```

- We can use a similar method to add an element to the tree

- Although rather long it is relatively simple

Add an Element

```
public boolean add(E element)
```

Add an Element

```
public boolean add(E element)
{
    if (root == null) {
        root = new Node<E>(element, null);
        size++;
        return true;
    }
```

Add an Element

```
public boolean add(E element)
{
    if (root == null) {
        root = new Node<E>(element, null);
        size++;
        return true;
    } else {
        Node<E> temp = root;
```

Add an Element

```
public boolean add(E element)
{
    if (root == null) {
        root = new Node<E>(element, null);
        size++;
        return true;
    } else {
        Node<E> temp = root;
        while (true) {
            int comp = ((Comparable)(element)).compareTo(temp.element);
```

Add an Element

```
public boolean add(E element)
{
    if (root == null) {
        root = new Node<E>(element, null);
        size++;
        return true;
    } else {
        Node<E> temp = root;
        while (true) {
            int comp = ((Comparable)(element)).compareTo(temp.element);
            if (comp == 0)
                return false;
```

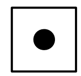
```
            if (comp<0) {
                if (temp.left != null)
                    temp = temp.left;









            }
        }
    }
}
```

```
        if (comp<0) {
            if (temp.left != null)
                temp = temp.left;
            else {
                temp.left = new Node<T>(element, temp);
                size++;
                return true;









            }
        }
    }
}
```

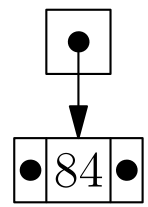
```
        if (comp<0) {
            if (temp.left != null)
                temp = temp.left;
            else {
                temp.left = new Node<T>(element, temp);
                size++;
                return true;
            }
        } else {
            if (temp.right != null)
                temp = temp.right;
            else {
                temp.right = new Node<T>(element, temp);
                size++;
                return true;
            }
        }
    }
  }
}
```

Tree in Action

add(84)

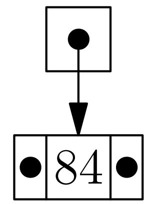


Tree in Action

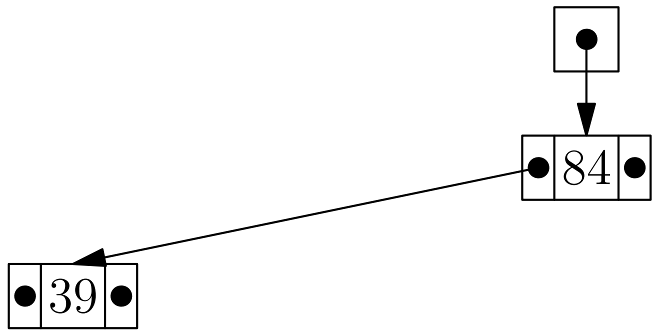


Tree in Action

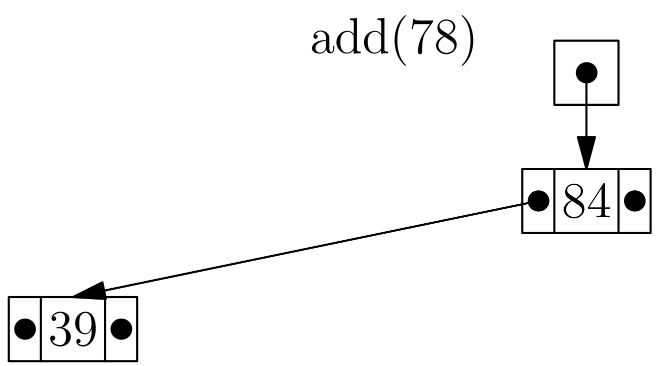
add(39)



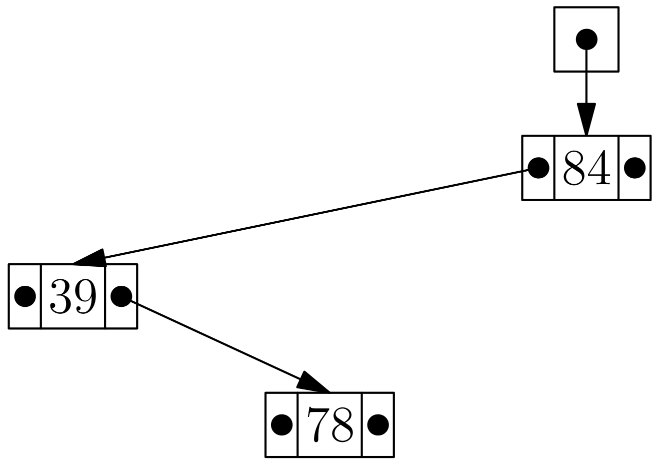
Tree in Action



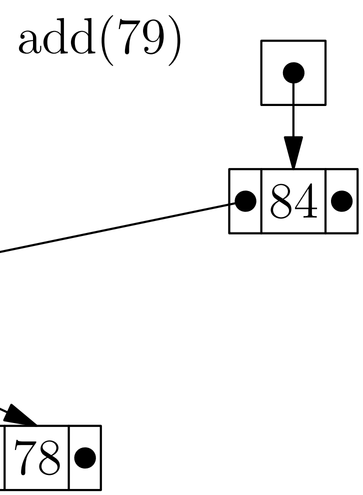
Tree in Action



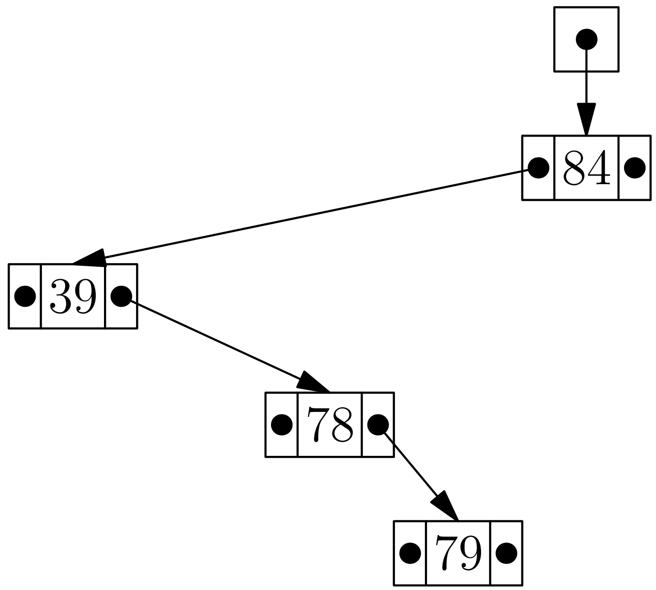
Tree in Action



Tree in Action

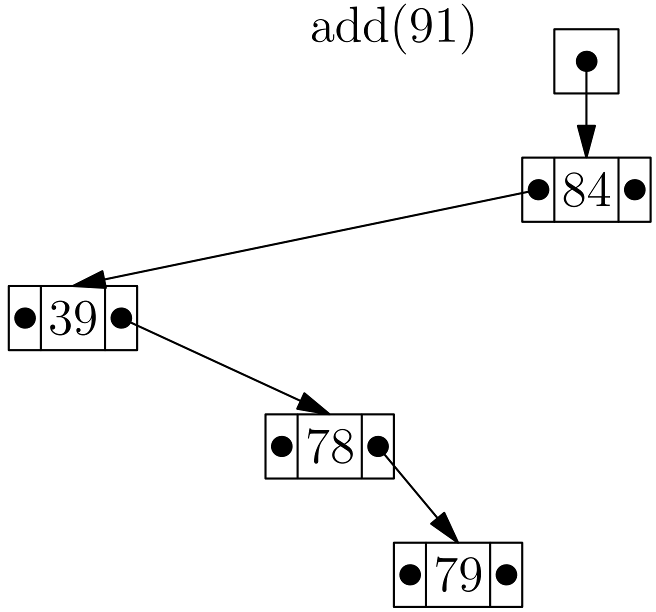


Tree in Action

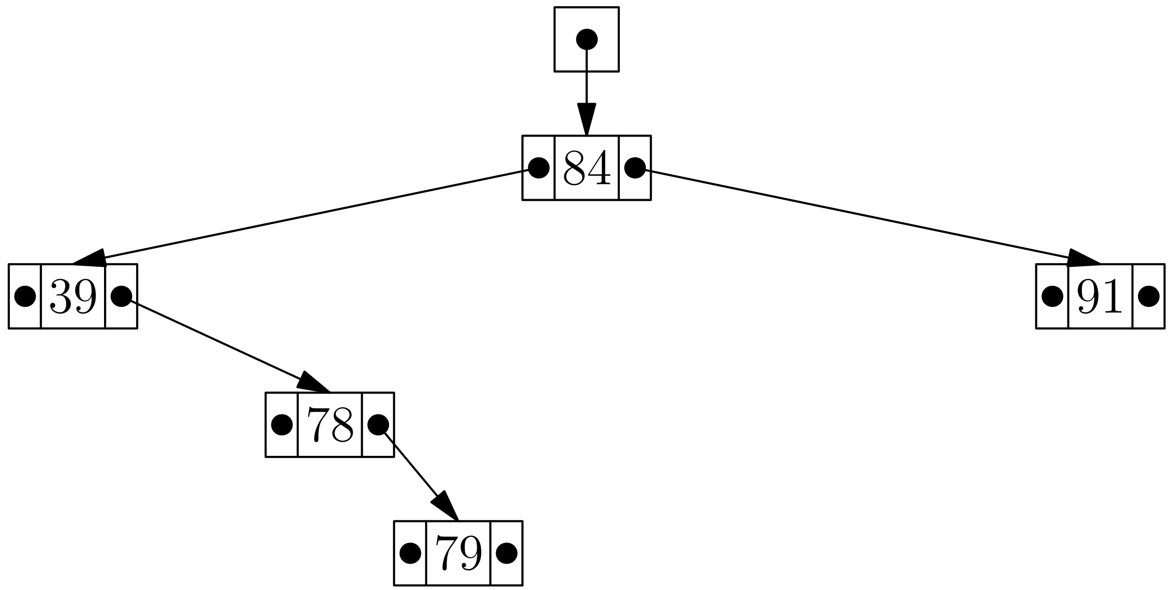


Tree in Action

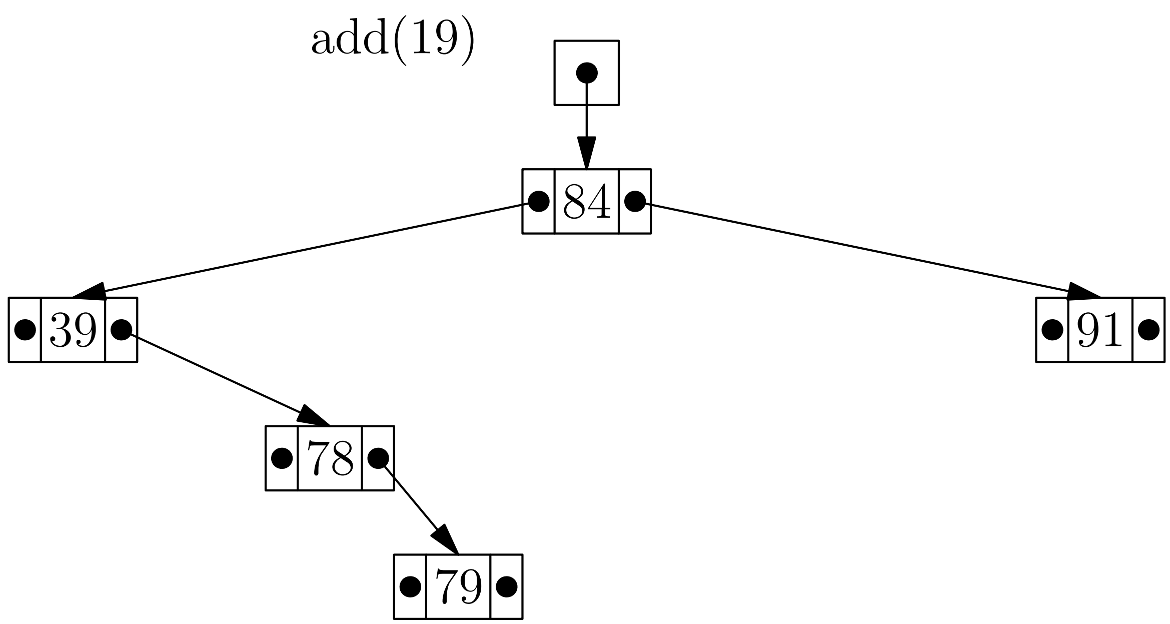
add(91)



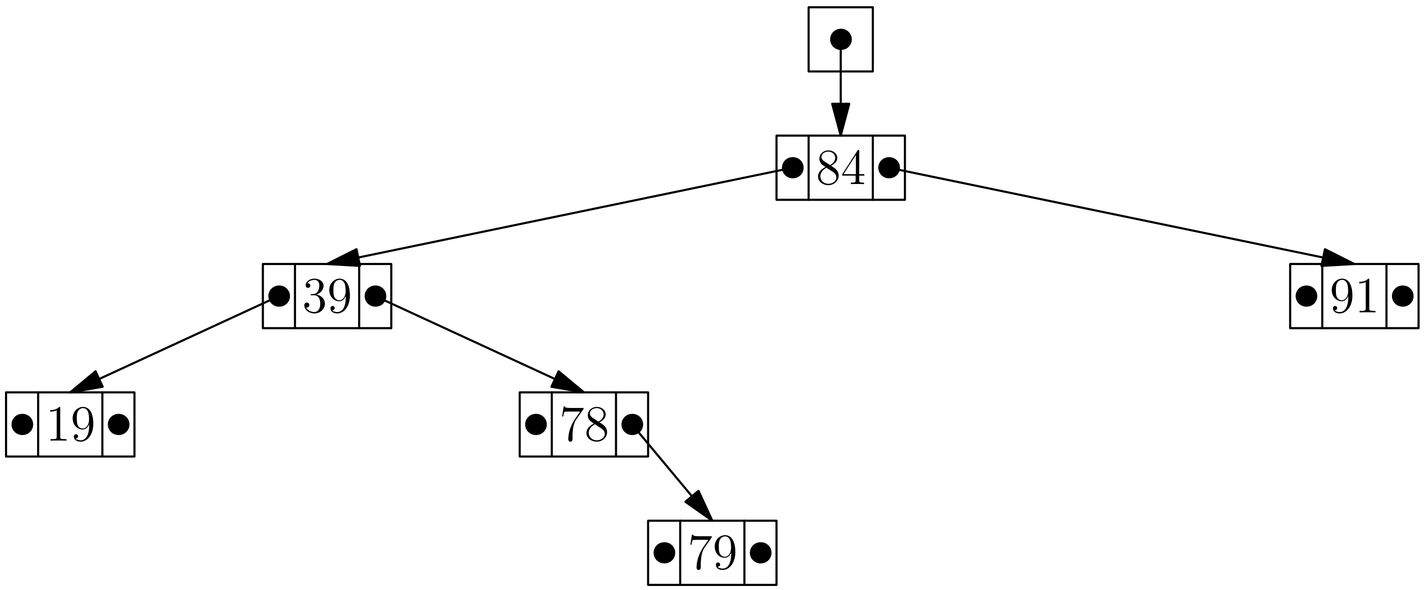
Tree in Action



Tree in Action

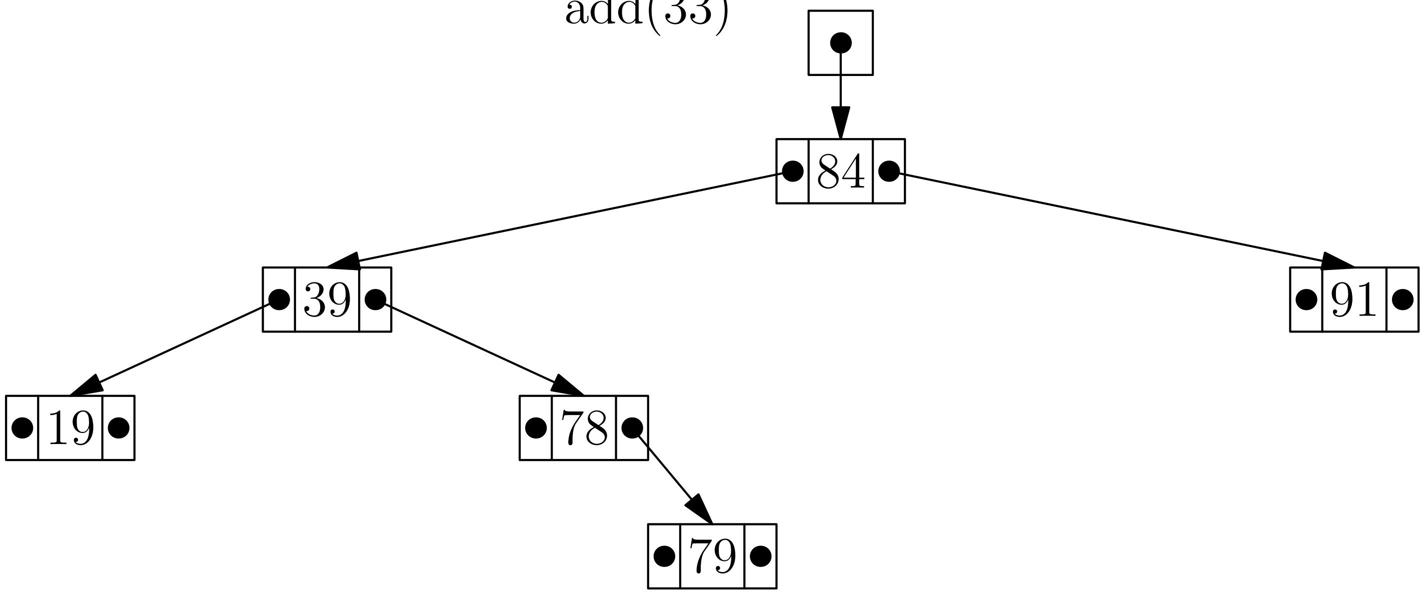


Tree in Action

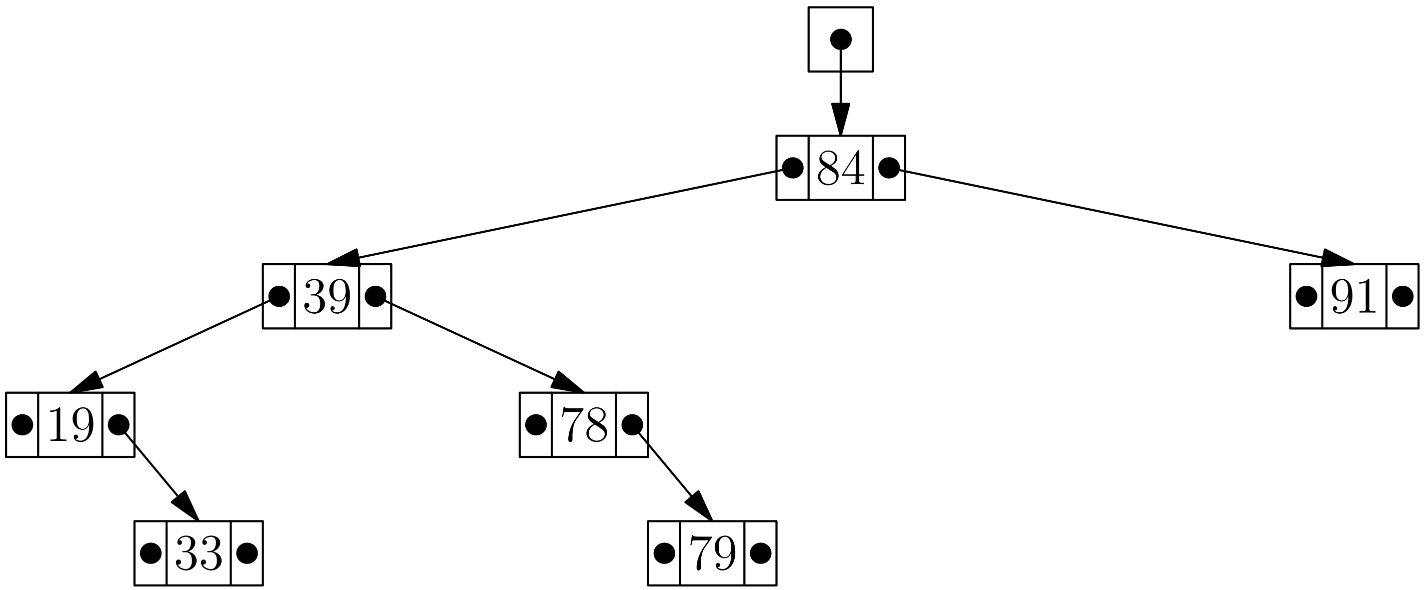


Tree in Action

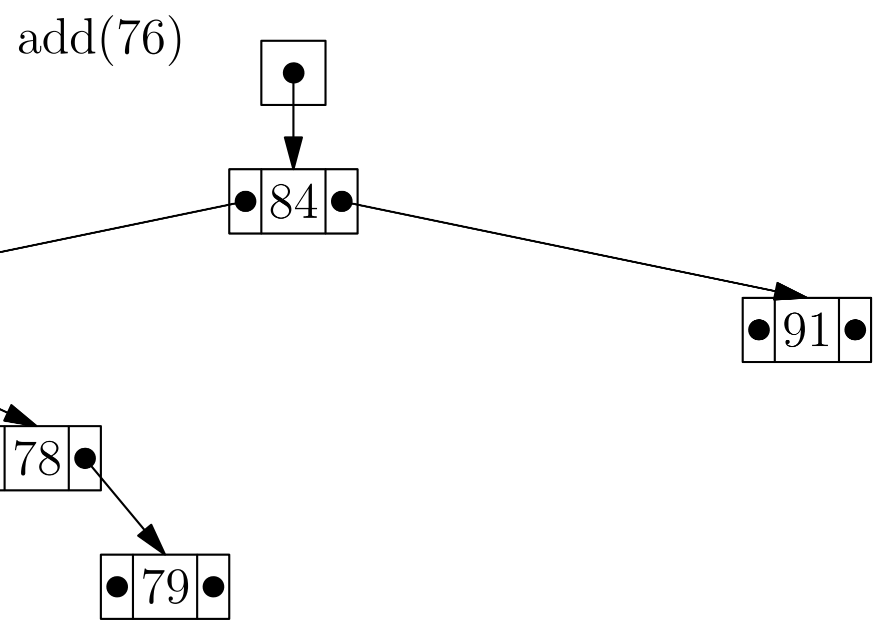
add(33)



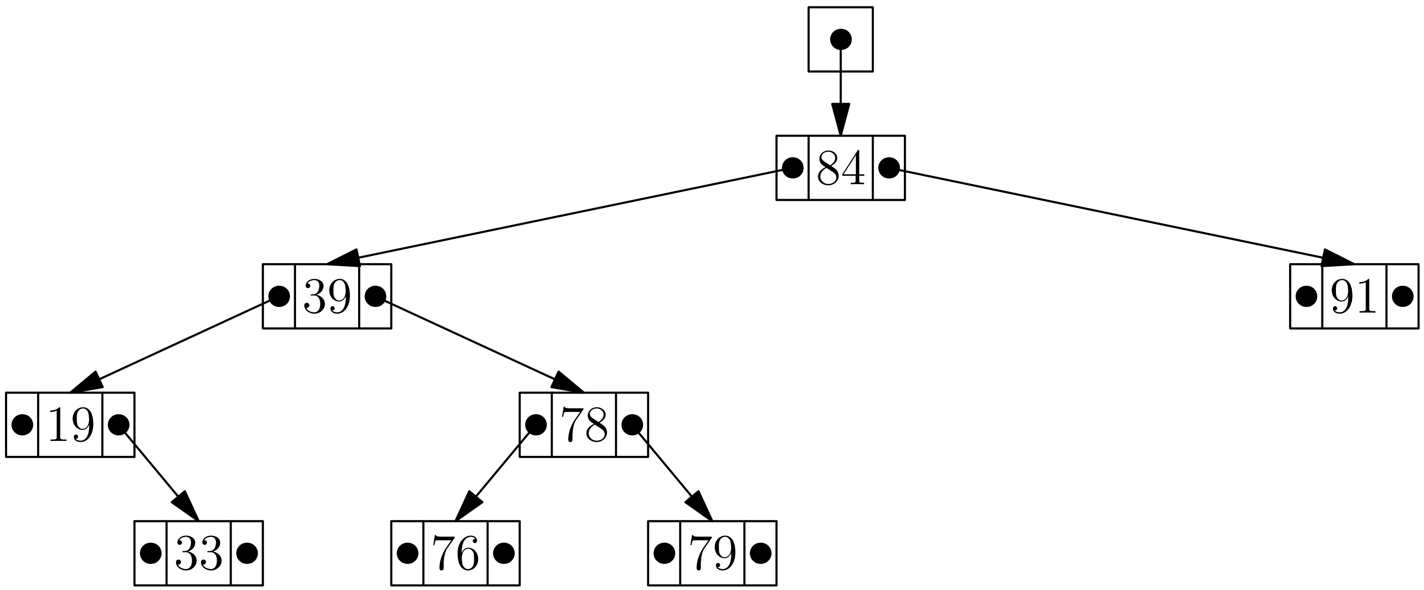
Tree in Action



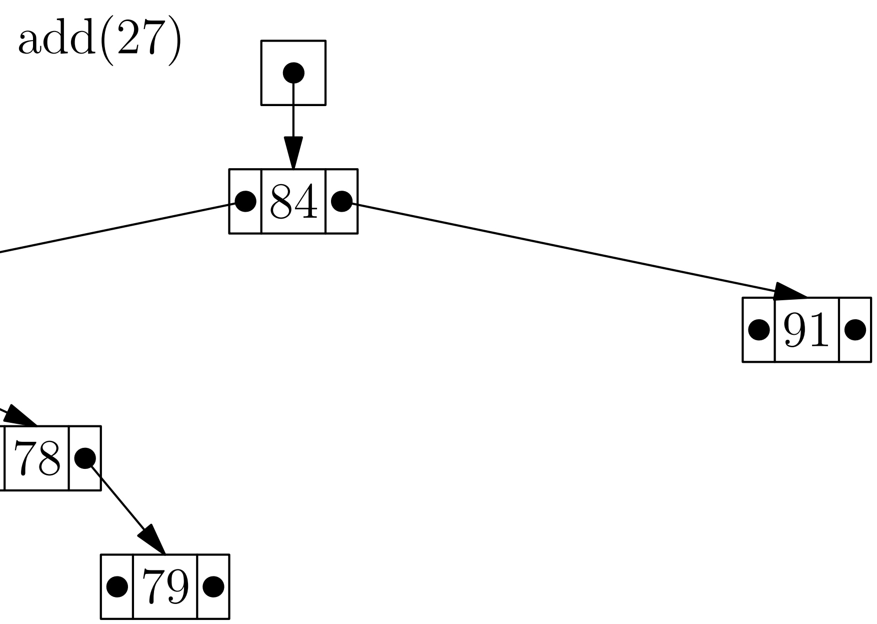
Tree in Action



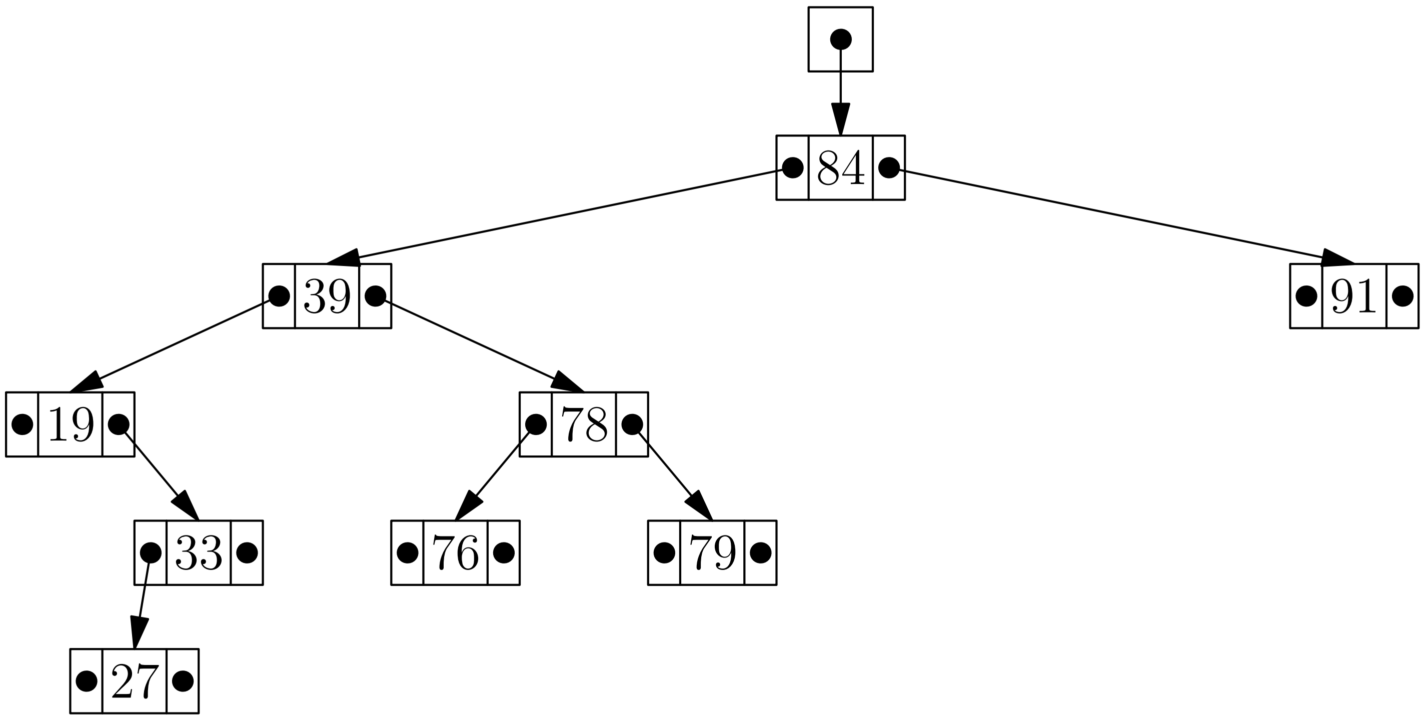
Tree in Action



Tree in Action

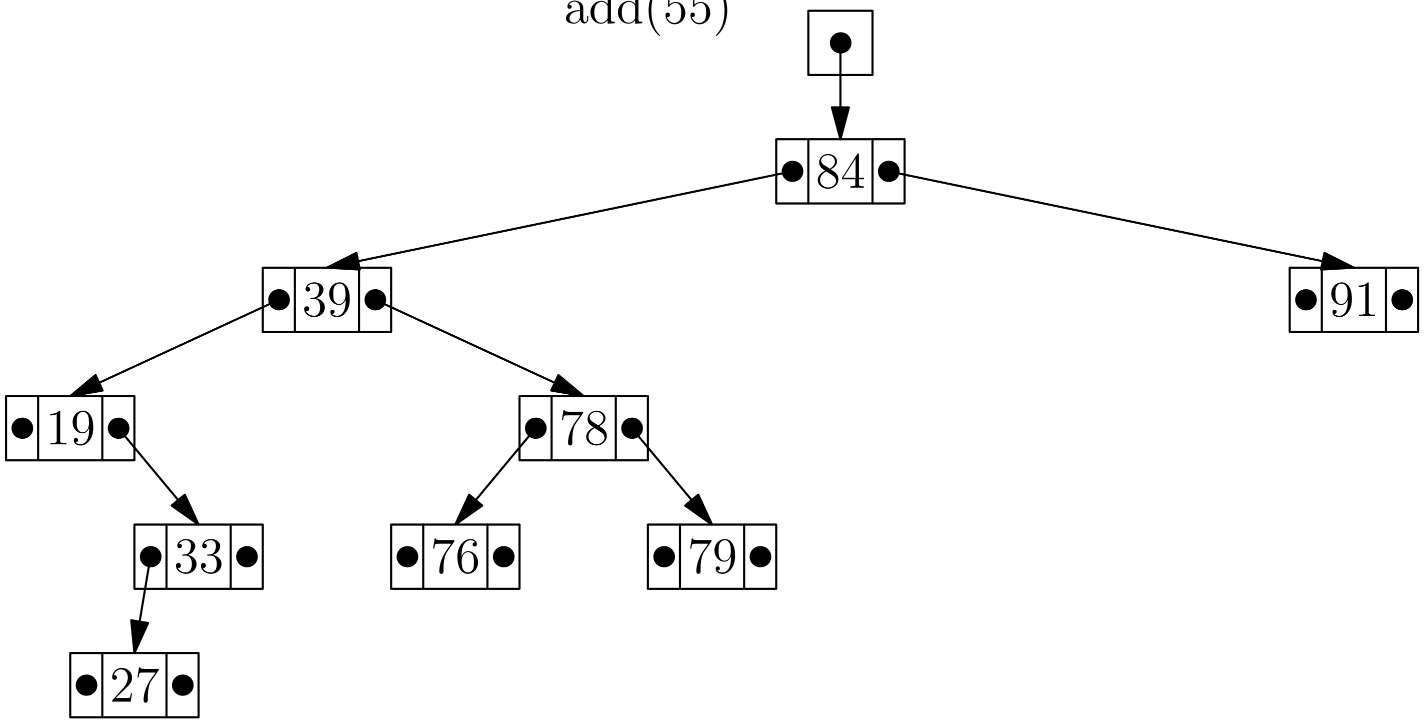


Tree in Action

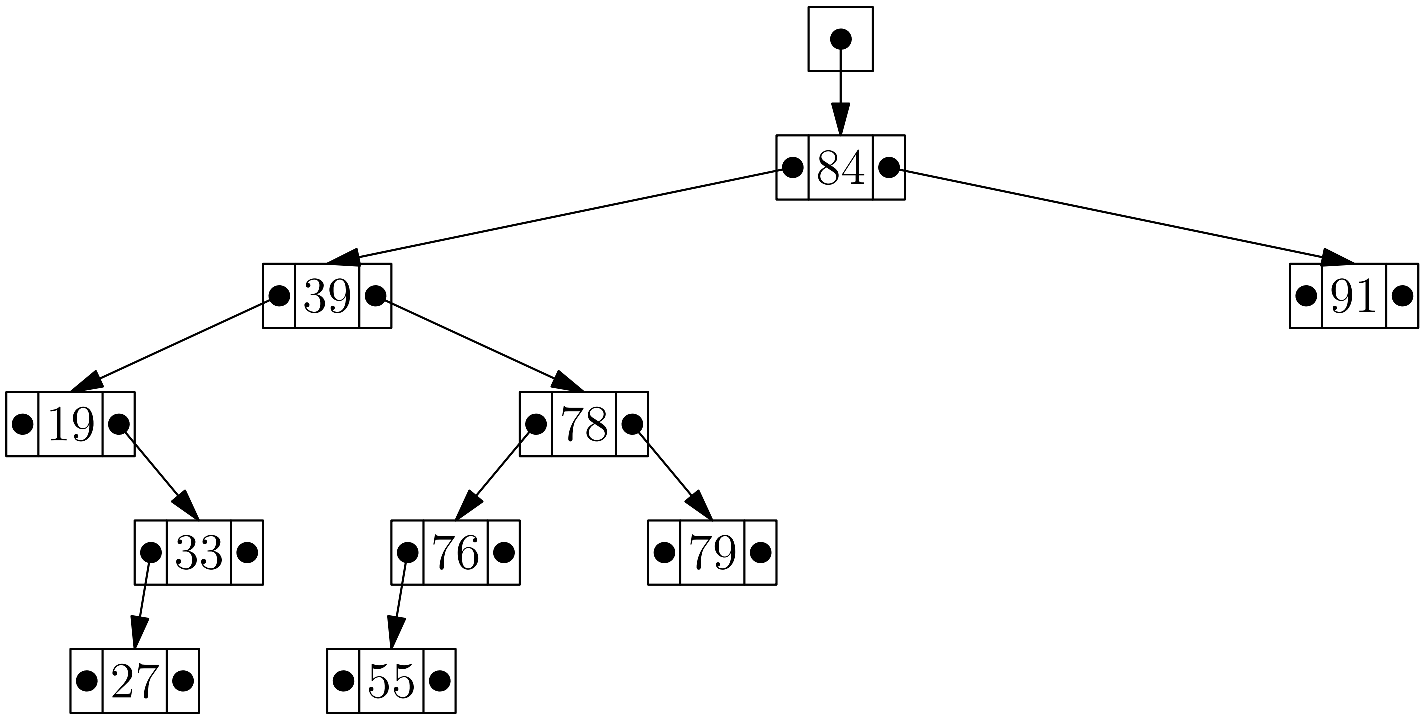


Tree in Action

add(55)



Tree in Action



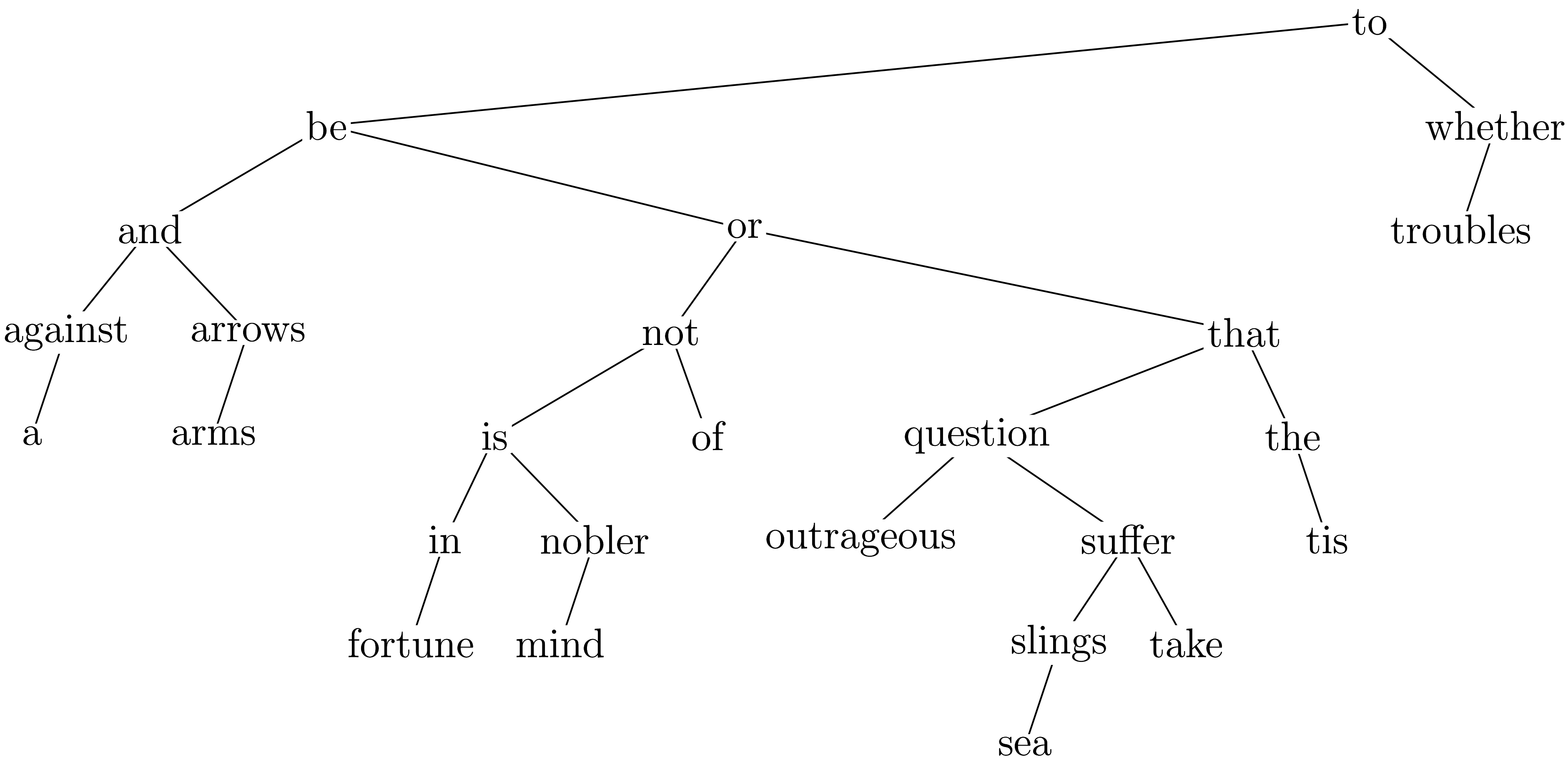
Shape of Tree

- The structure of the tree depends on the order in which we add elements to it
- Suppose we add

  To be, or not to be: that is the question:
  Whether 'tis nobler in the mind to suffer
  The slings and arrows of outrageous fortune,
  Or to take arms against a sea of troubles,

- Ignoring punctuation we get the following tree

Hamlet


to
be
whether
and
or
troubles
against
arrows
not
that
a
arms
is
of
question
the
in
nobler
outrageous
suffer
tis
fortune
mind
slings
take
sea

Outline

1. Trees
2. Binary Trees
   - Implementing Binary Trees
3. Binary Search Trees
   - Definition
   - Implementing a Set
4. **Tree Iterators**

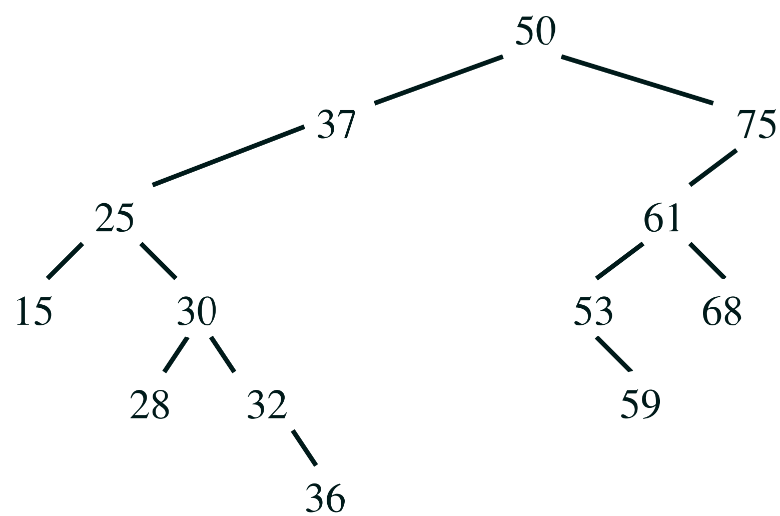
Tree Iterators

- We follow the usual pattern to create a tree iterator
- `public Iterator<E> iterator() {return new TreeIterator<E>();}`
- Where `TreeIterator<T>` is a private nested class within the `BinarySearchTree` class
- `TreeIterator` extends the Iterator interface and requires implementation of
  - ⋆ `boolean hasNext()`
  - ⋆ `T next()`
  - ⋆ `void remove()`
- `T next()` needs to find the successor node

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go *up* to the left as far as possible and then move up right

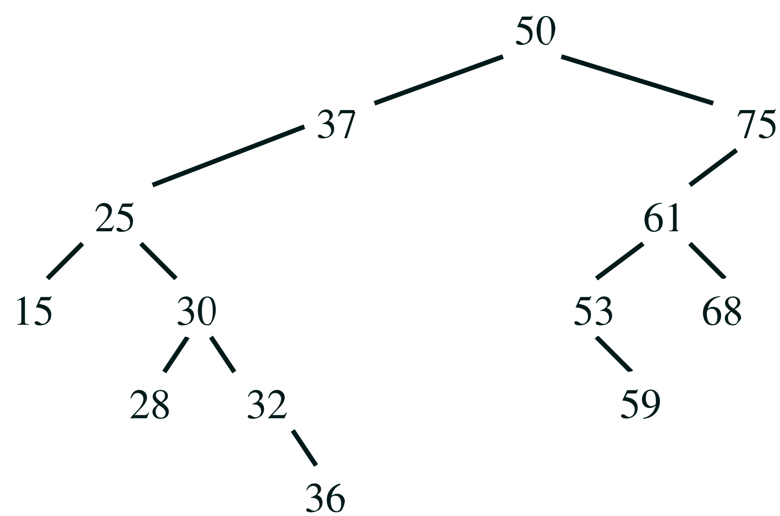


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

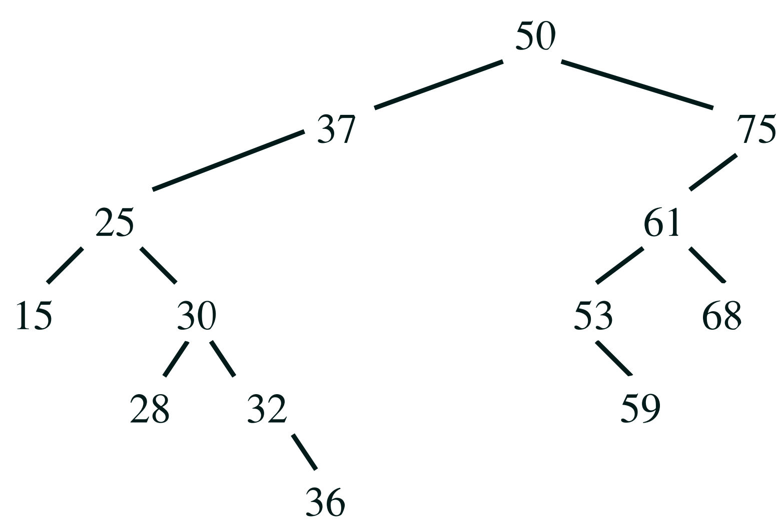


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

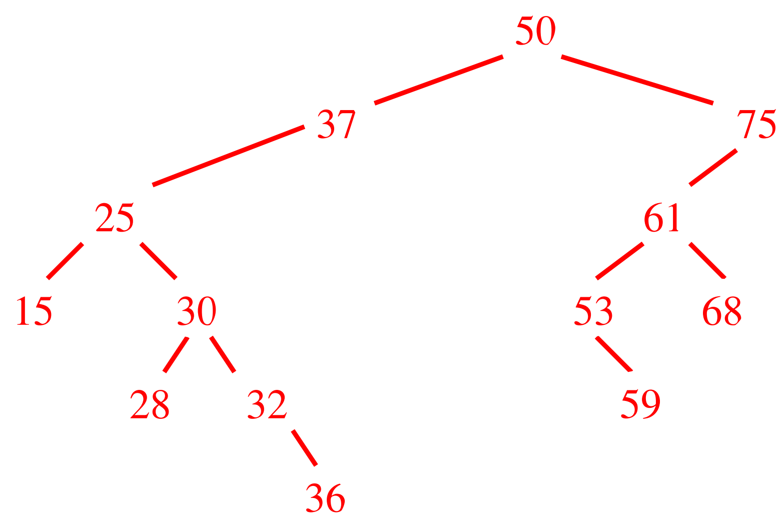


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

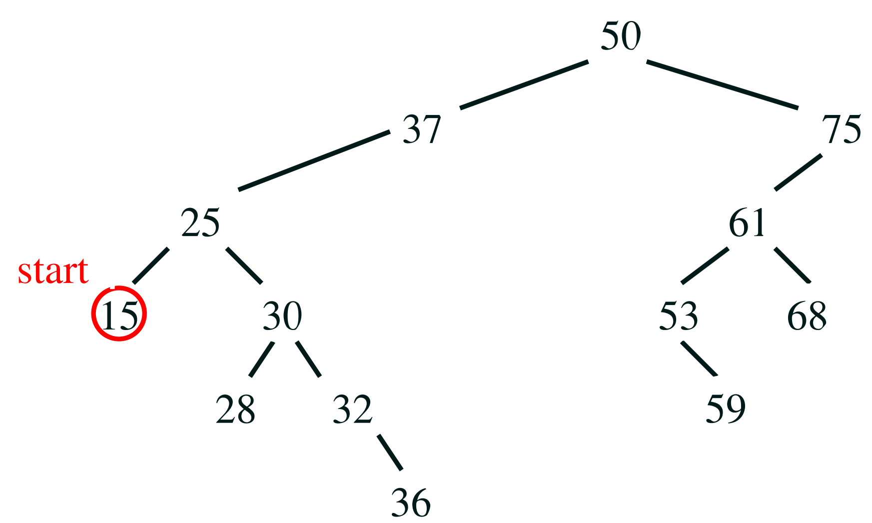


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

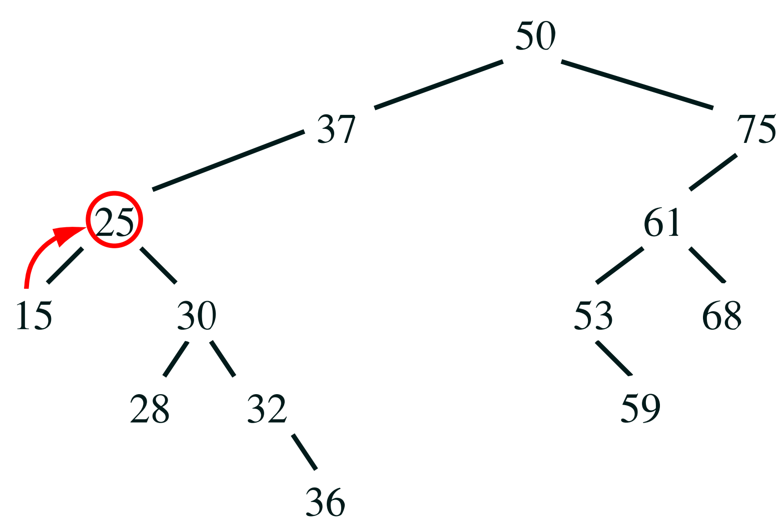


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

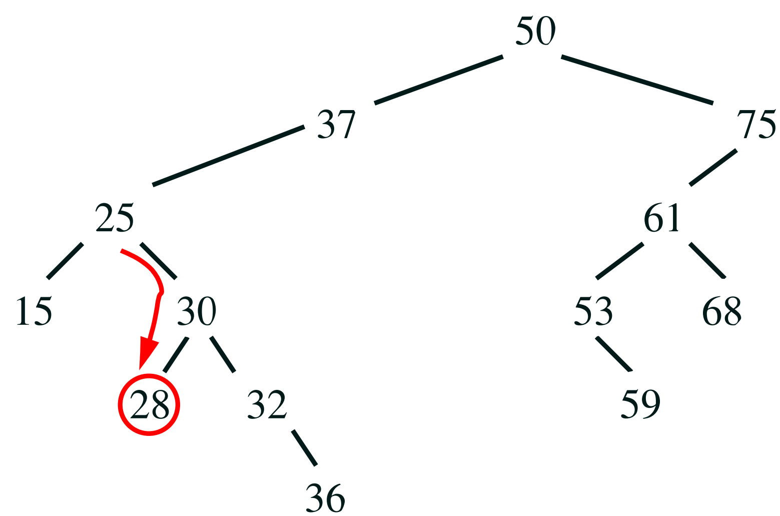


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

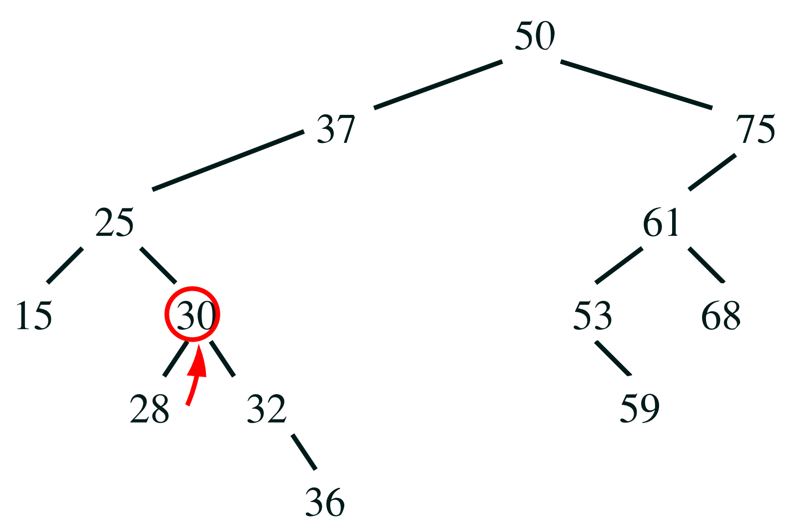


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go *up* to the left as far as possible and then move up right

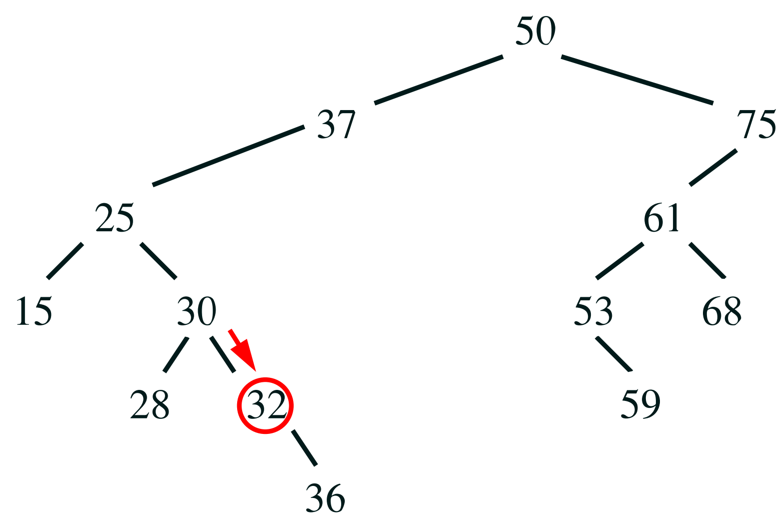


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

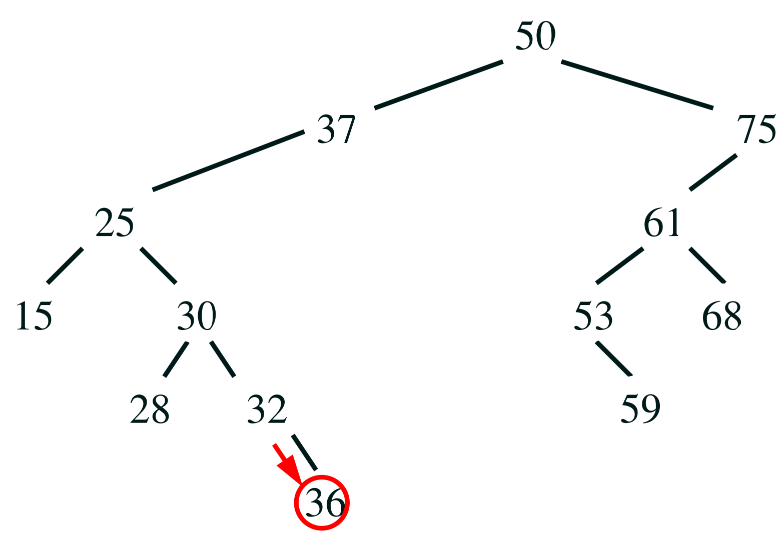


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

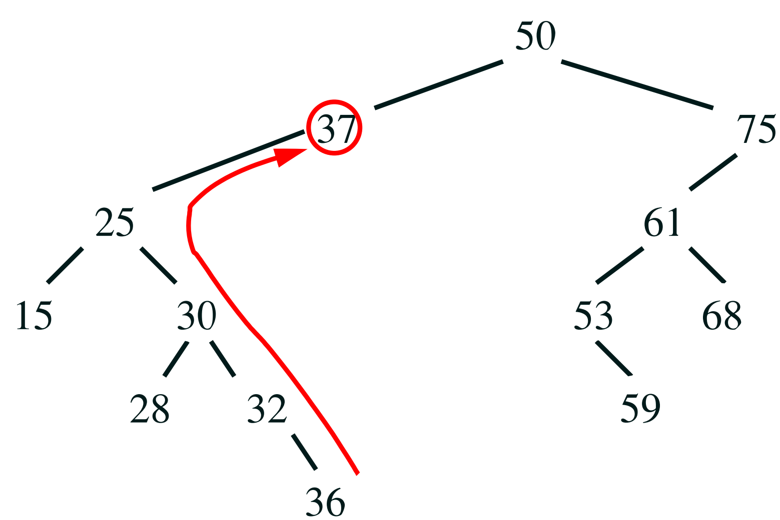


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

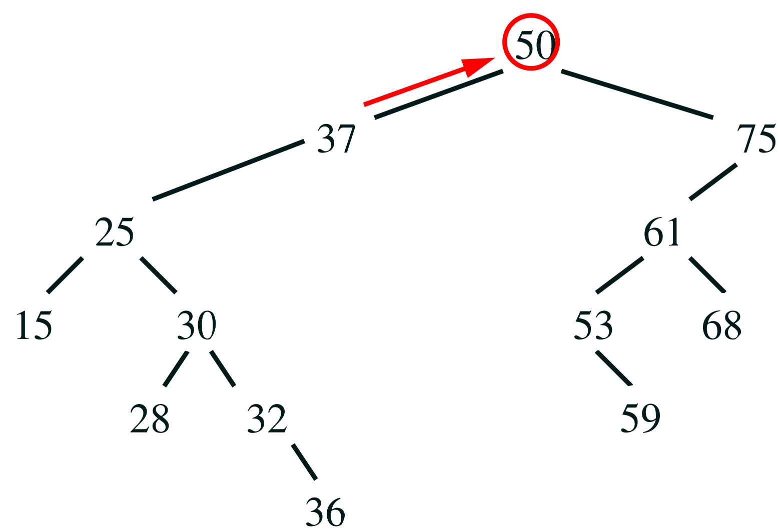


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

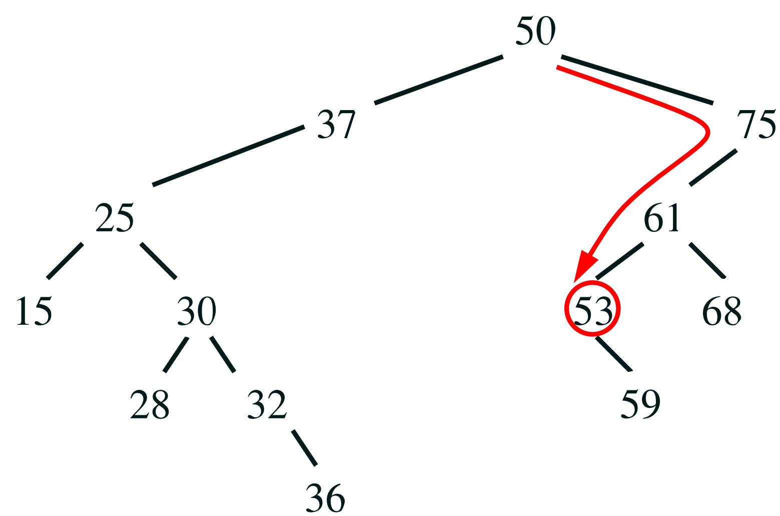


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

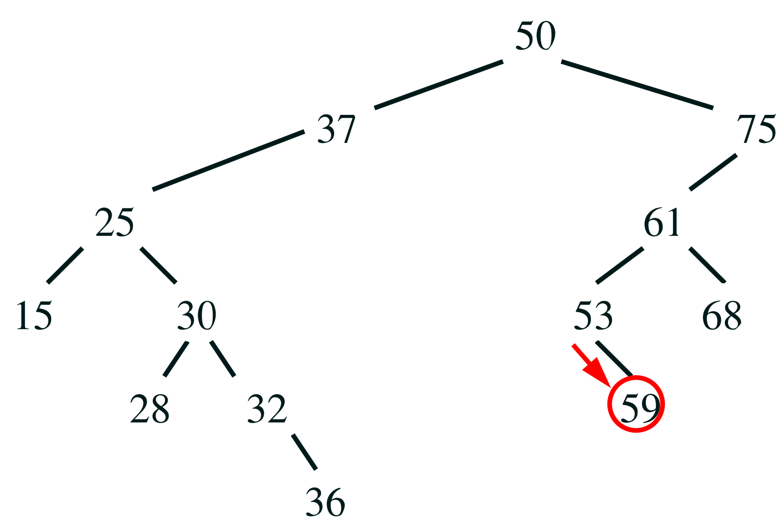


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

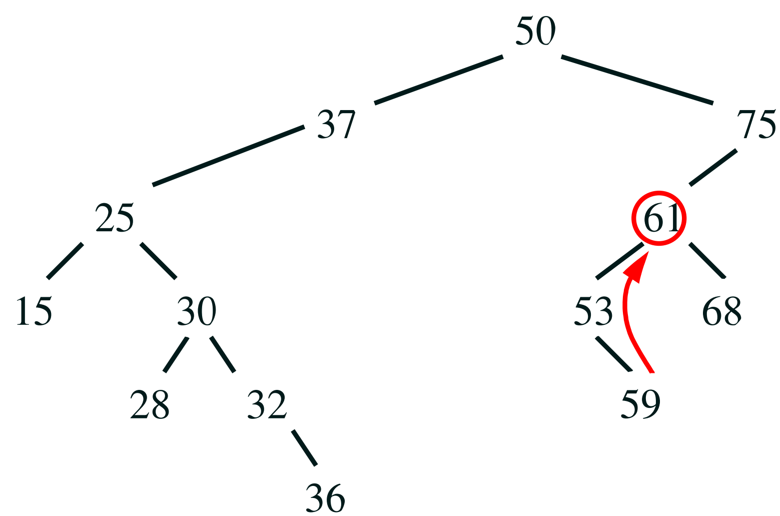


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

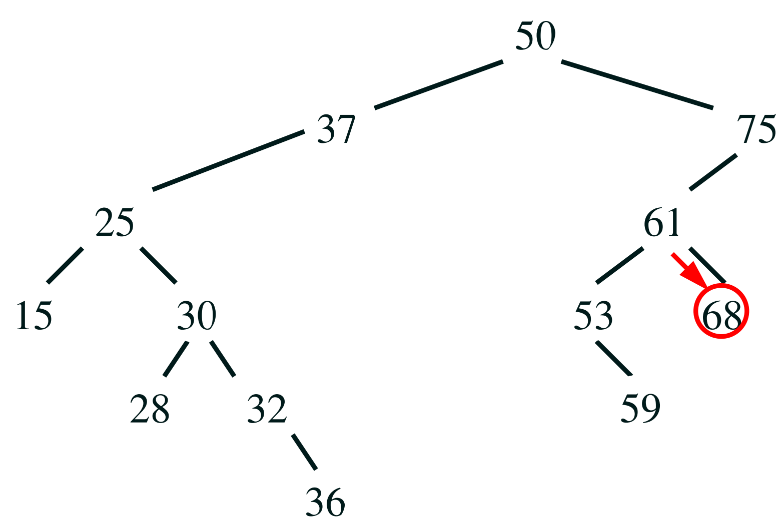


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

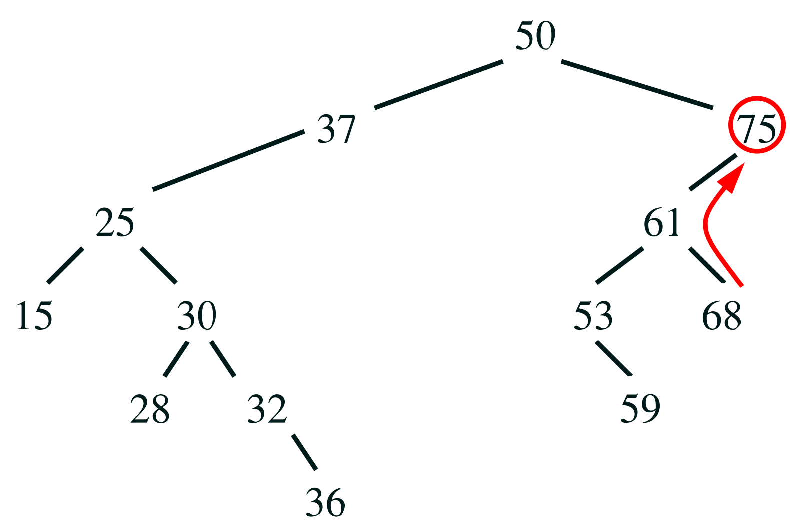


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right

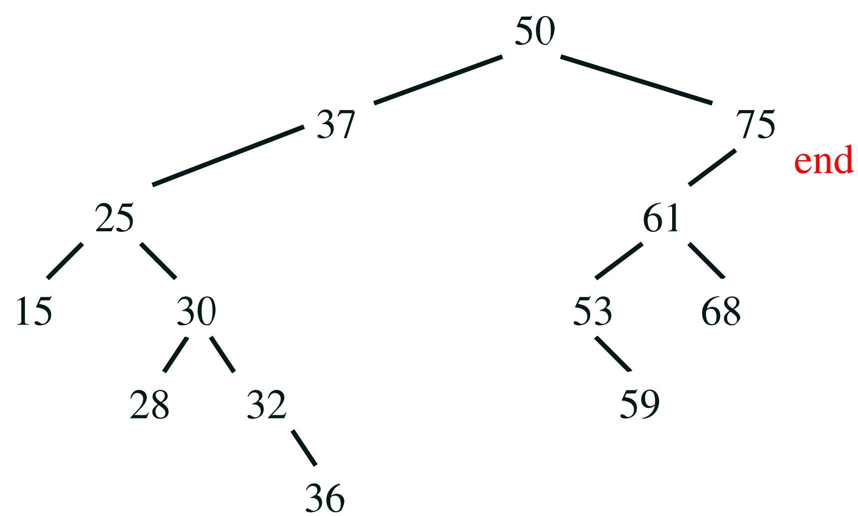


$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Tree Iterator

- To iterate through the elements we start in the left most branch
- To find the successor of the current element we follow two rules

1. **If** right child exist **then** move right once and then move as far left as possible
2. **else** go up to the left as far as possible and then move up right



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

Lessons

- Trees and particularly binary trees are one of the most important tools of a computer scientist
- Conceptually they are quite simple
- However, there are a lot of details that need to be understood
- Coding even simple trees needs great care
- As we will see things get more complicated