# Algorithms and Analysis

## Lesson 9: *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

# Trees

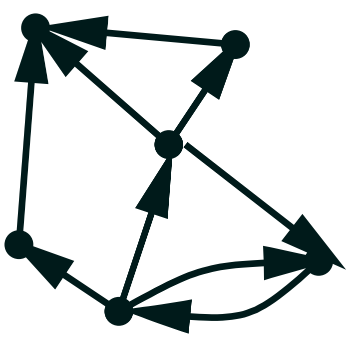
- Trees are one of the major ways of structuring data
- They are used in a vast number of data structures
  - ⋆ Binary search trees
  - ⋆ B-trees
  - ⋆ splay trees
  - ⋆ heaps
  - ⋆ tries
  - ⋆ suffix trees
- We shall cover most of these

# Trees

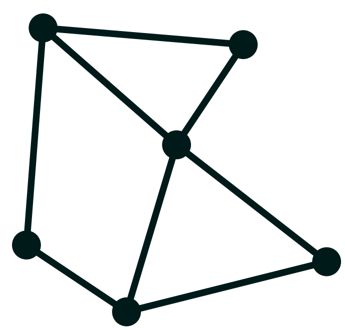
- Trees are one of the major ways of structuring data
- They are used in a vast number of data structures
  - ⋆ Binary search trees
  - ⋆ B-trees
  - ⋆ splay trees
  - ⋆ heaps
  - ⋆ tries
  - ⋆ suffix trees
- We shall cover most of these

# Defining Trees

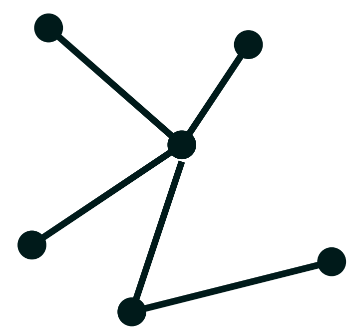
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges goes both ways
  - ⋆ **acyclic**: there are no cycles in the graph



graph



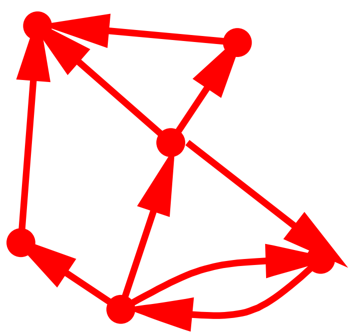
undirected graph



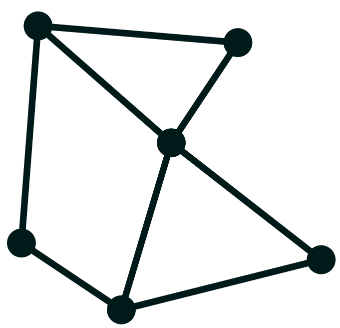
tree = acyclic undirected graph

# Defining Trees

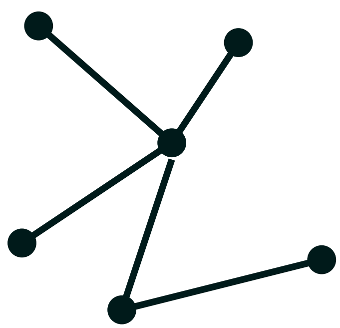
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges goes both ways
  - ⋆ **acyclic**: there are no cycles in the graph



graph



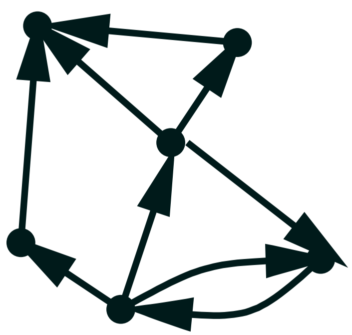
undirected graph



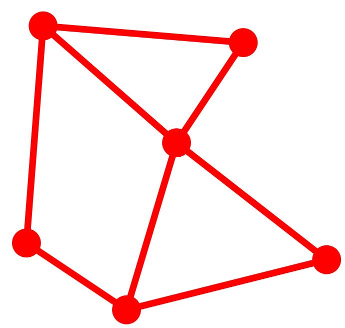
tree = acyclic undirected graph

# Defining Trees

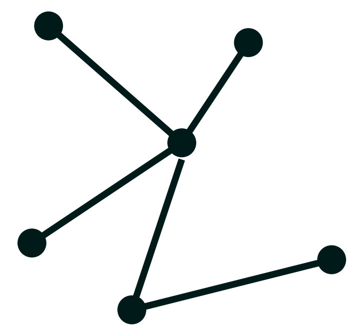
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges goes both ways
  - ⋆ **acyclic**: there are no cycles in the graph



graph



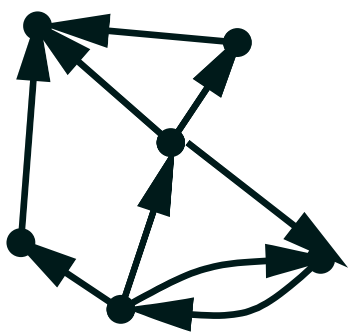
undirected graph



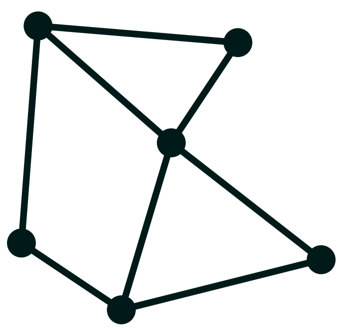
tree = acyclic undirected graph

# Defining Trees

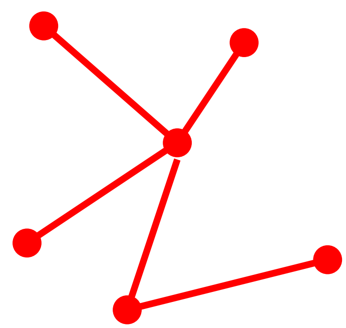
- Mathematically a tree is an **acyclic undirected graph**
  - ⋆ **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - ⋆ **undirected**: the edges goes both ways
  - ⋆ **acyclic**: there are no cycles in the graph



graph



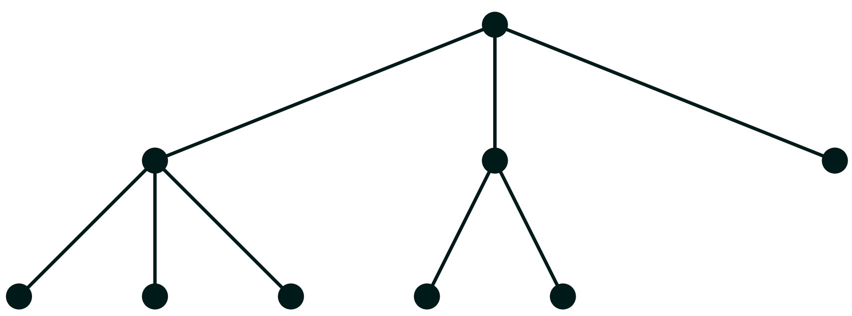
undirected graph



tree = acyclic undirected graph

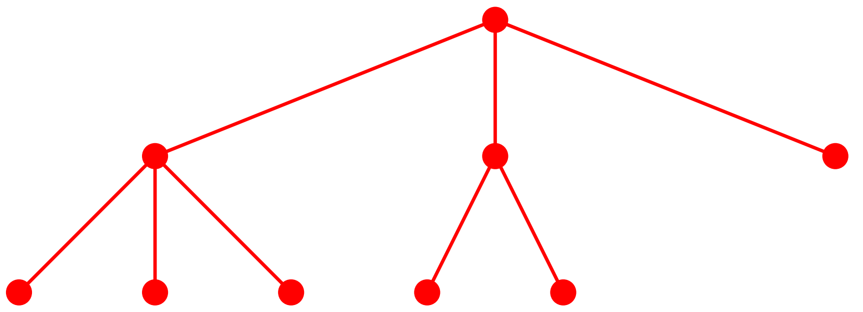
# Borrowing from Nature

- 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 beneath them
- Each child has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes



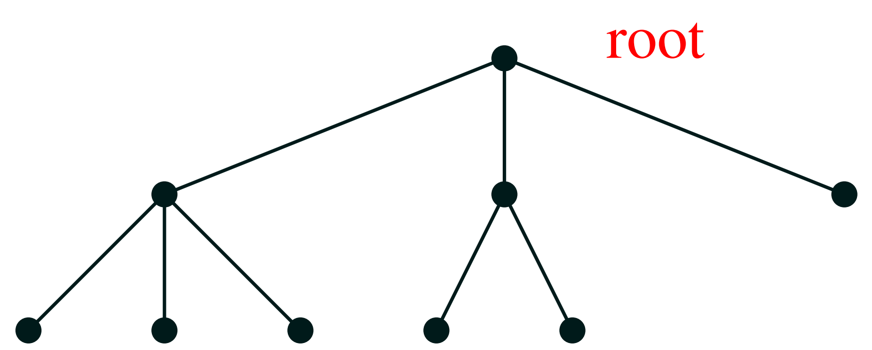
# Borrowing from Nature

- 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 beneath them
- Each child has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes



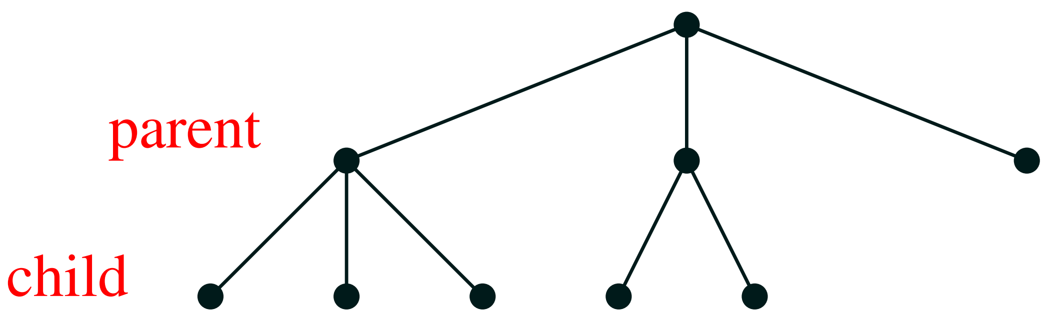
# Borrowing from Nature

- 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 beneath them
- Each child has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes



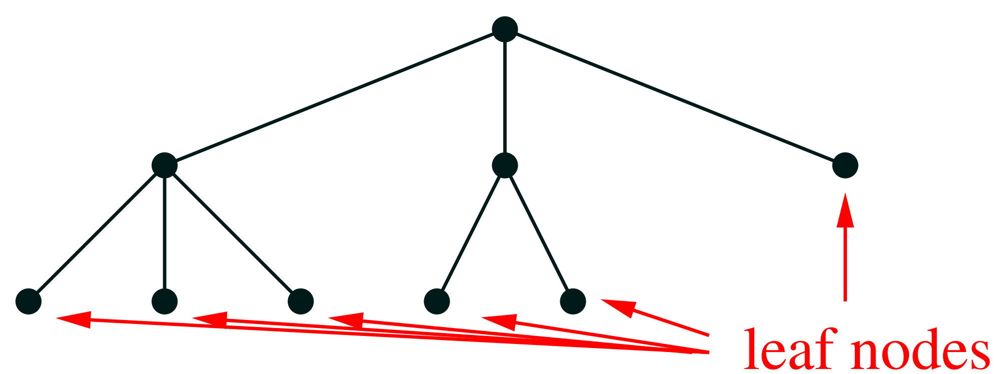
# Borrowing from Nature

- 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 beneath them
- Each child 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 beneath them
- Each child has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes

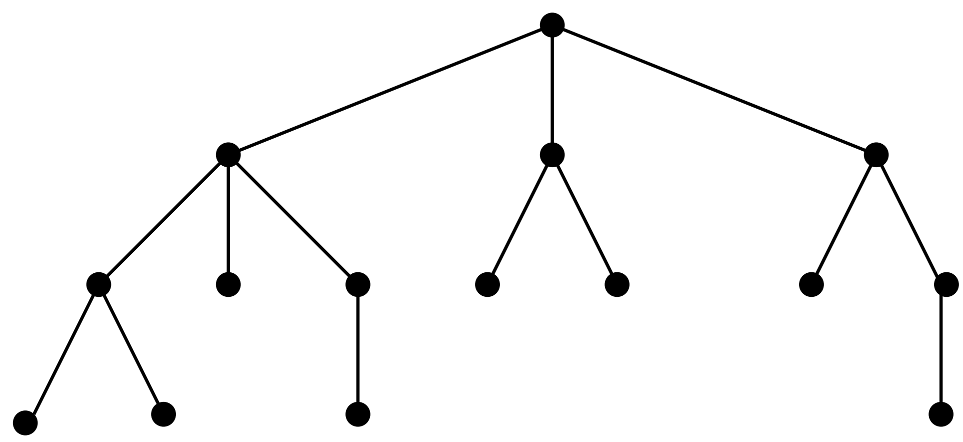


# Spot the Error

- One small biological inconsistency

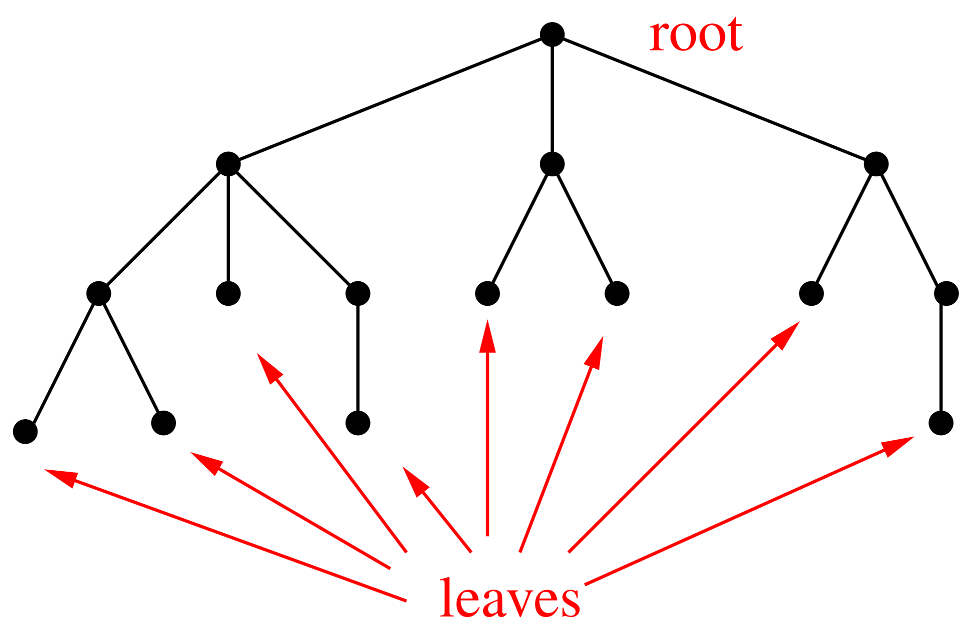
# Spot the Error

- One small biological inconsistency
- Yep!, computer scientists draw there trees upside down



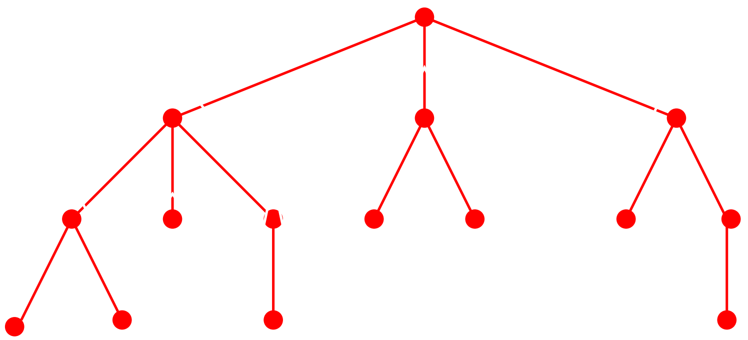
# Spot the Error

- One small biological inconsistency
- Yep!, computer scientists draw there trees upside down
  - ⋆ root at the top
  - ⋆ leaves at the bottom



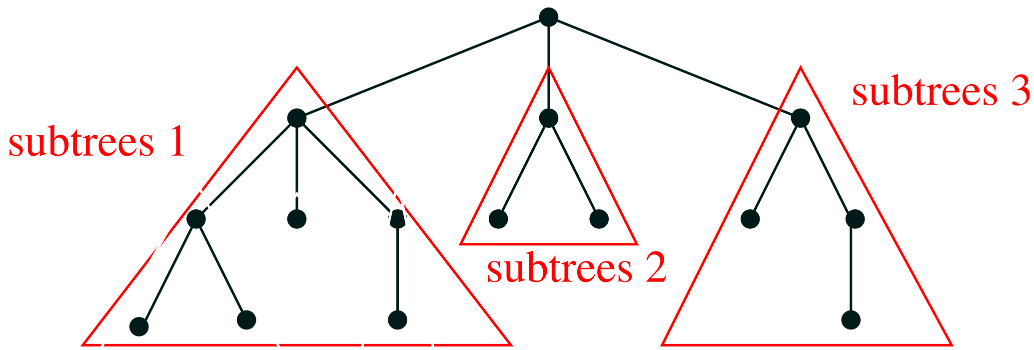
# Subtrees

- We can think of the tree made up of **subtrees**



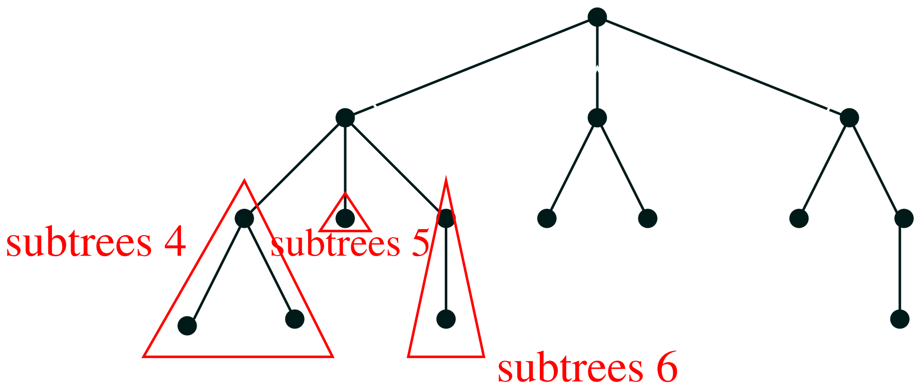
# Subtrees

- We can think of the tree made up of **subtrees**



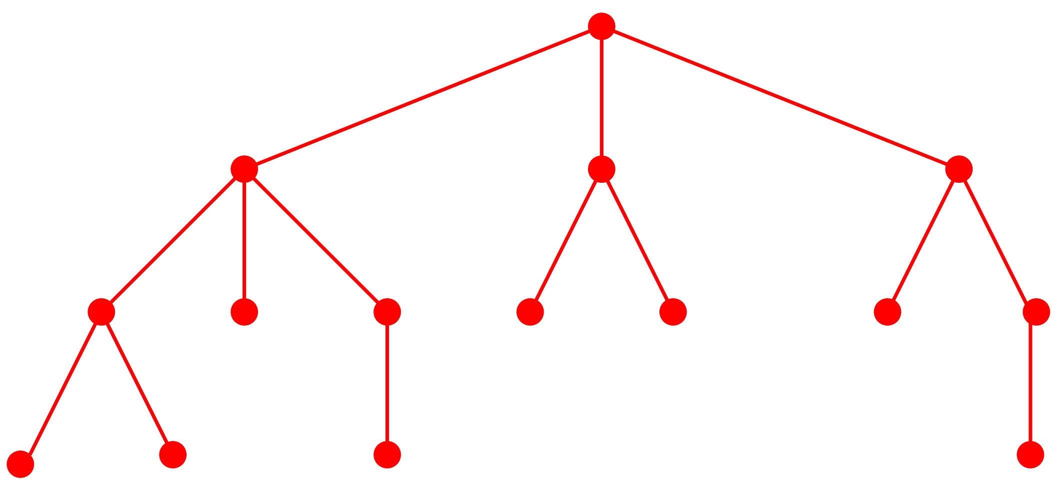
# Subtrees

- We can think of the tree made up of **subtrees**



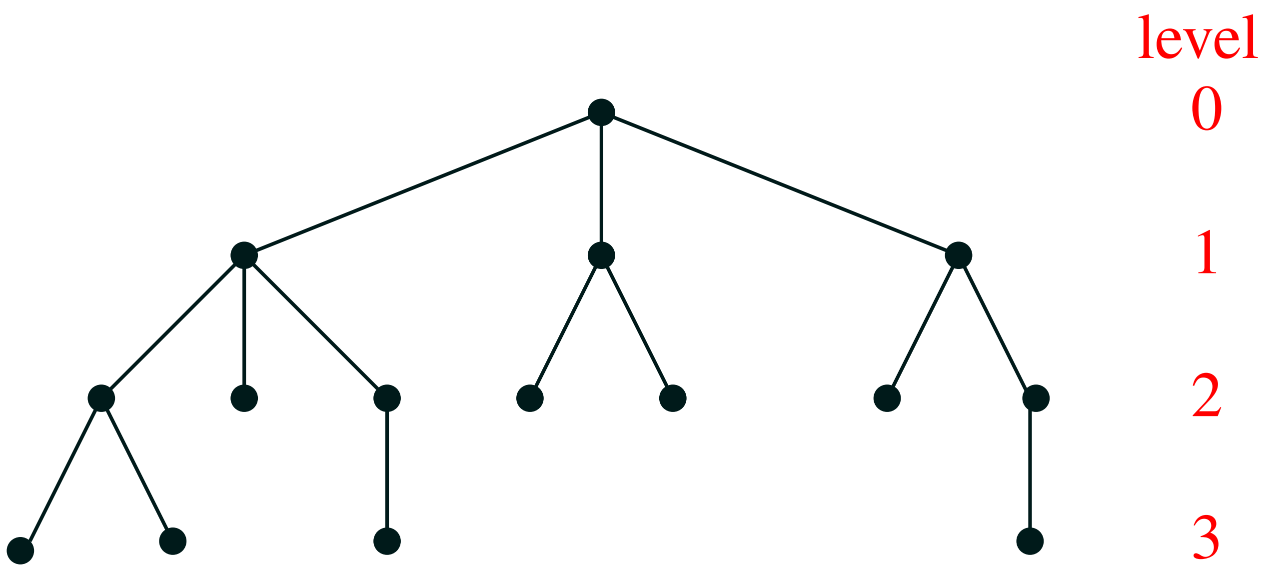
# Level of Nodes

- 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



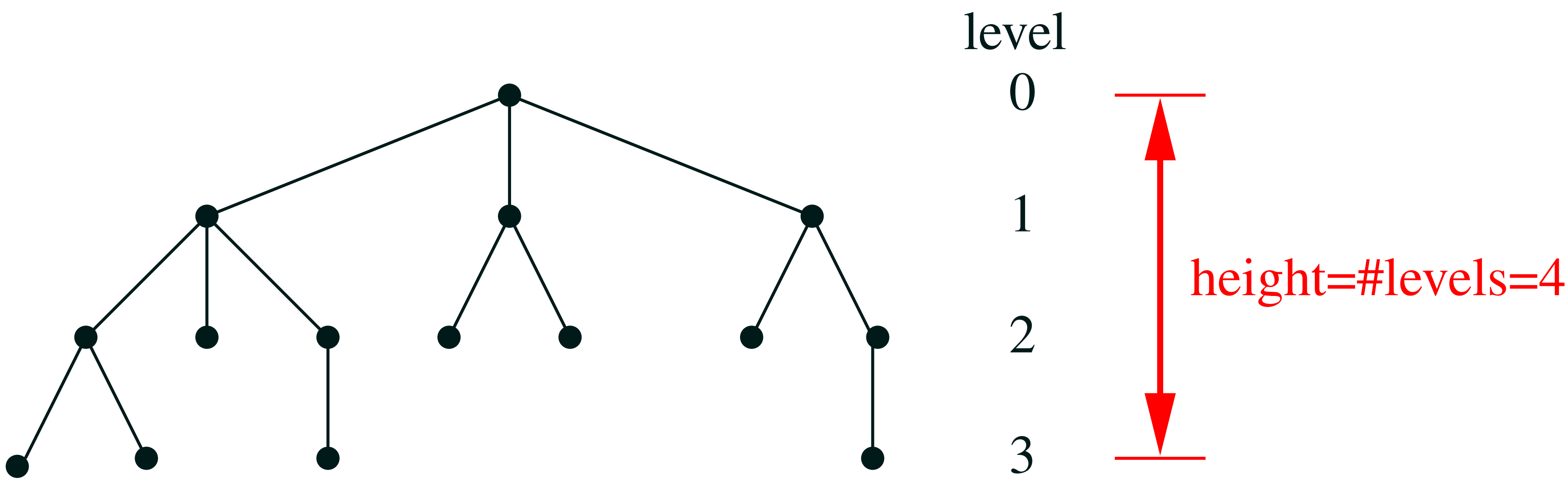
# Level of Nodes

- 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



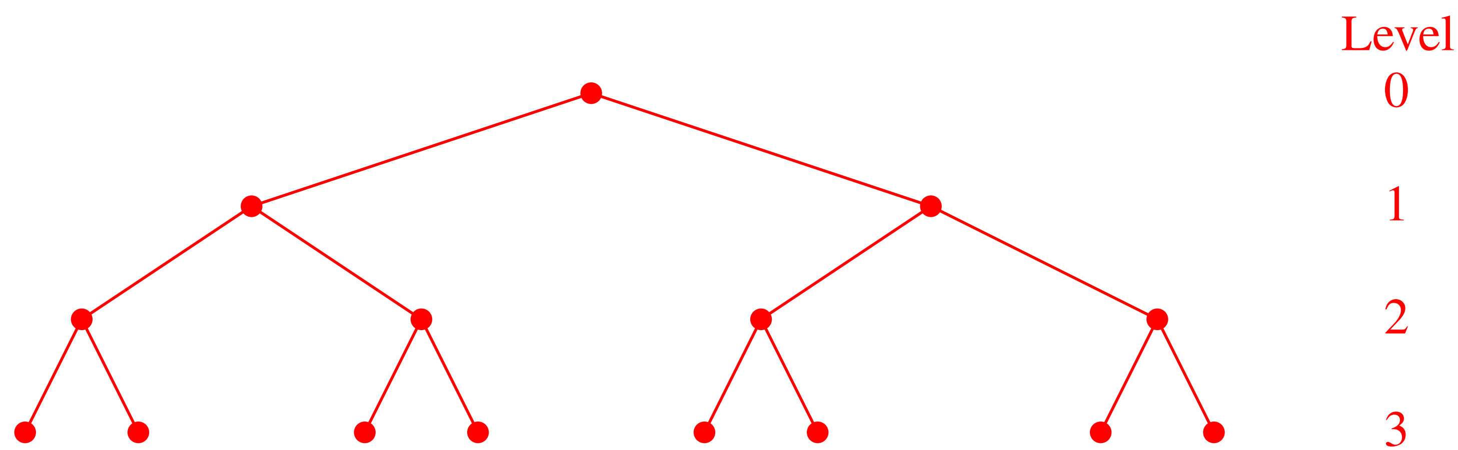
# Outline

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

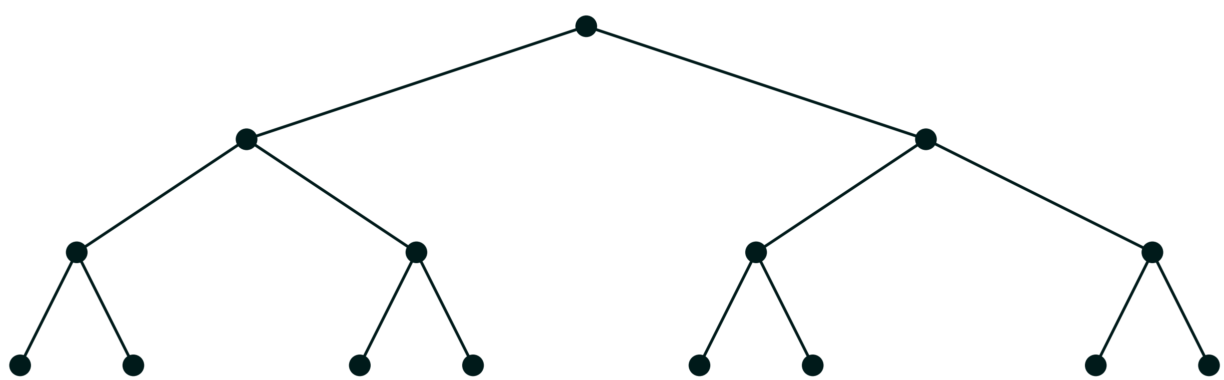
$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$



# 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

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

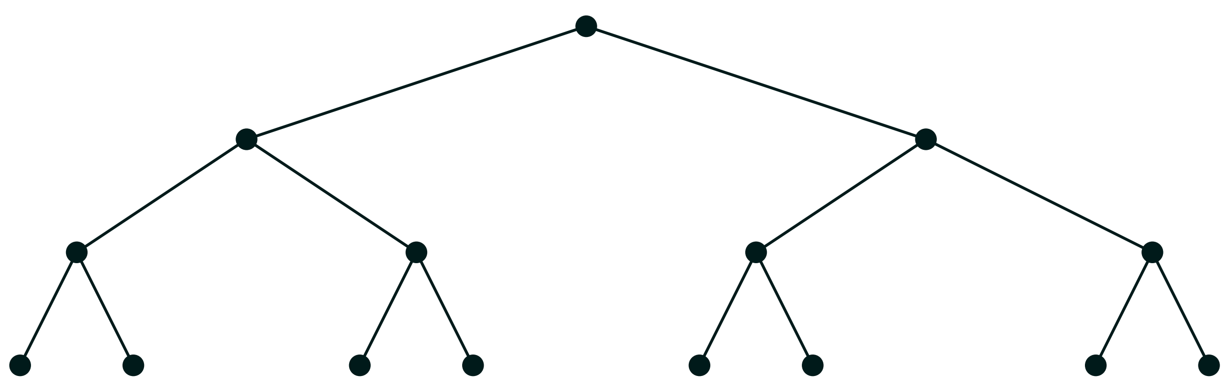


| Level | # Nodes |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

# 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

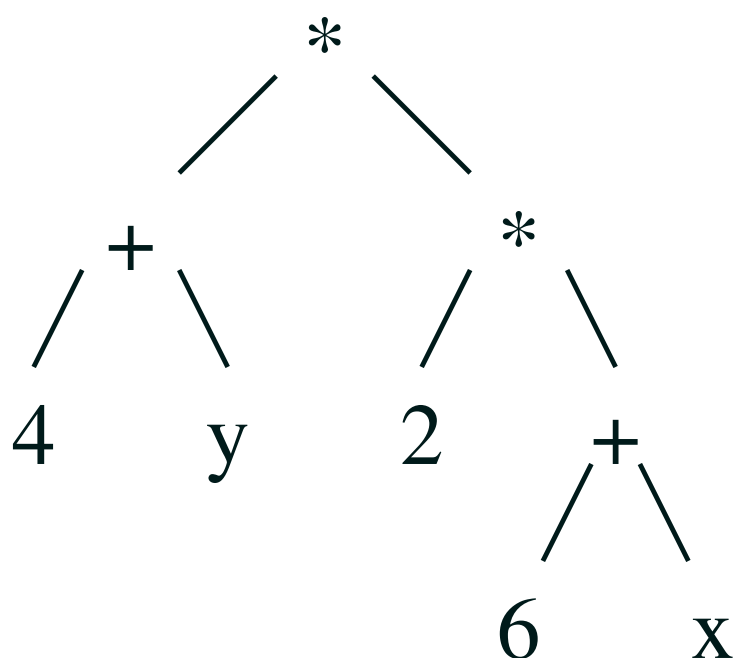
$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$



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

# Uses of Binary Trees

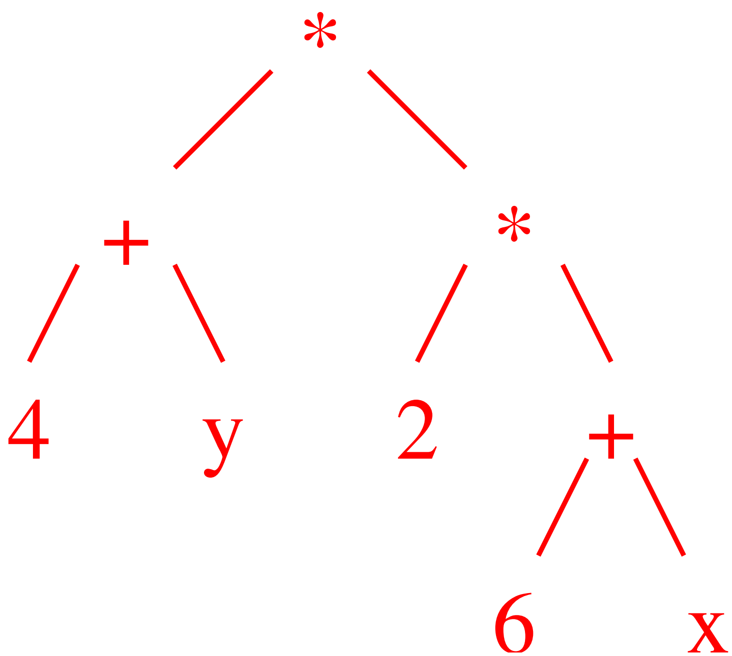
- Binary trees have a huge number of applications
- For example, they are used as **expression trees** to represent formulae




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

# Uses of Binary Trees

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




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

# 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 pointer to left and right children
- To help navigate the tree each node will contain a pointer to its parent

# 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 pointer to left and right children
- To help navigate the tree each node will contain a pointer to its parent

# Implementation

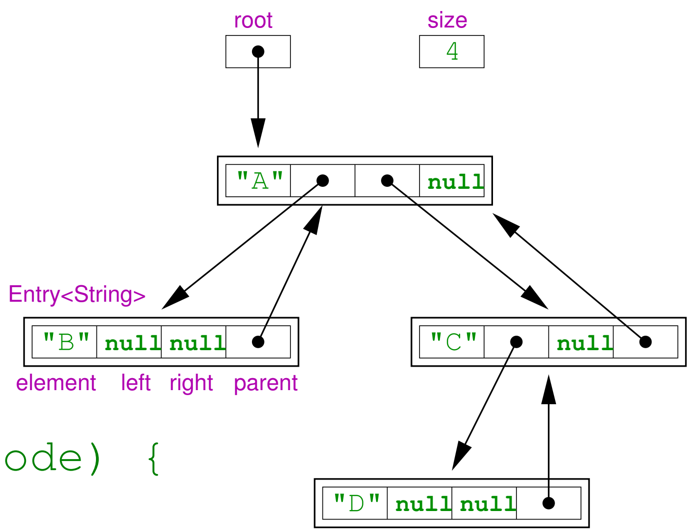
- We wish to build a generic binary tree class with each node housing an element
- Again we use a `Node<T>` class as the building block for our data structure—in this case a node of the tree
- The `Node<T>` class will contain a pointer to left and right children
- To help navigate the tree each node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

# 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 pointer to left and right children
- To help navigate the tree each node will contain a pointer to its parent

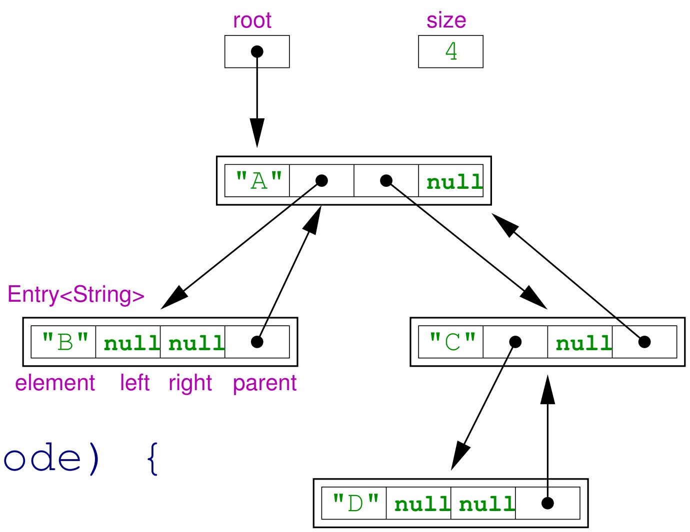
# C++ Code

```
template <typename T>
class binary_tree {
private:
  class Node {
  public:
    T element;
    Node* parent;
    Node* left = nullptr;
    Node* right = nullptr;

    Node(const T& value, Node* parent_node) {
      element = value;
      parent = parent_node;
    }
  };

  unsigned no_elements = 0;
  Node* root = nullptr;

public:
  ...
```



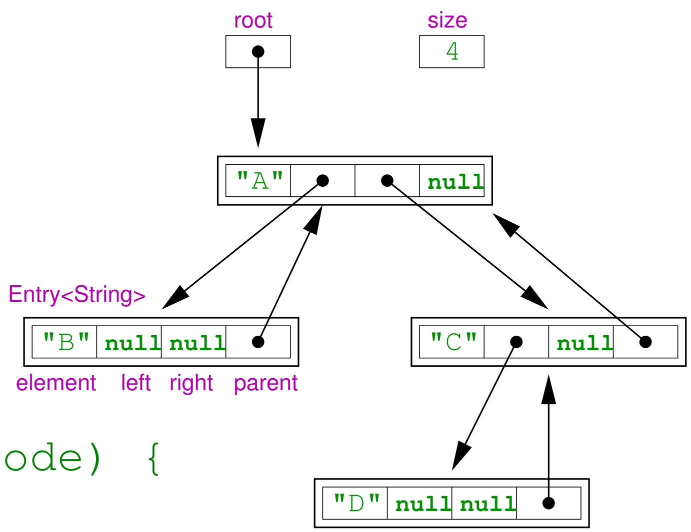
# C++ Code

```
template <typename T>
class binary_tree {
private:
  class Node {
  public:
    T element;
    Node* parent;
    Node* left = nullptr;
    Node* right = nullptr;

    Node(const T& value, Node* parent_node) {
      element = value;
      parent = parent_node;
    }
  };

  unsigned no_elements = 0;
  Node* root = nullptr;

public:
  ...
```



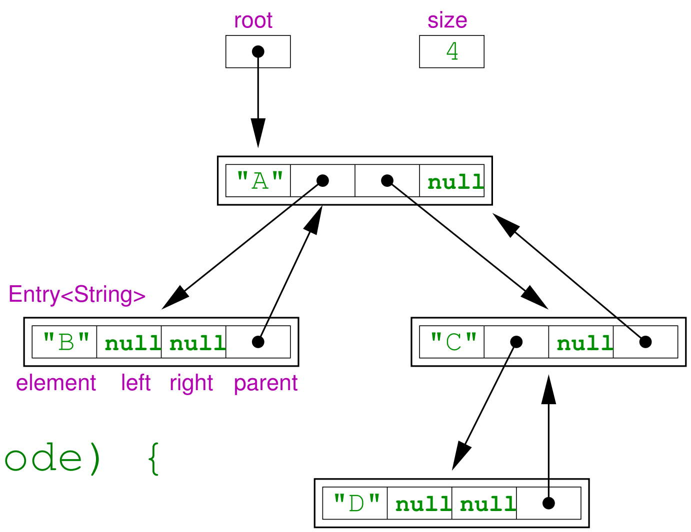
# C++ Code

```
template <typename T>
class binary_tree {
private:
  class Node {
  public:
    T element;
    Node* parent;
    Node* left = nullptr;
    Node* right = nullptr;

    Node(const T& value, Node* parent_node) {
      element = value;
      parent = parent_node;
    }
  };

  unsigned no_elements = 0;
  Node* root = nullptr;

public:
  ...
```



# C++ Code

```
template <typename T>
class binary_tree {
private:
  class Node {
  public:
    T element;
    Node* parent;
    Node* left = nullptr;
    Node* right = nullptr;

    Node(const T& value, Node* parent_node) {
      element = value;
      parent = parent_node;
    }
  };

  unsigned no_elements = 0;
  Node* root = nullptr;

public:
  ...
```



# 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

# 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

# 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

# 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

# Binary Search Trees

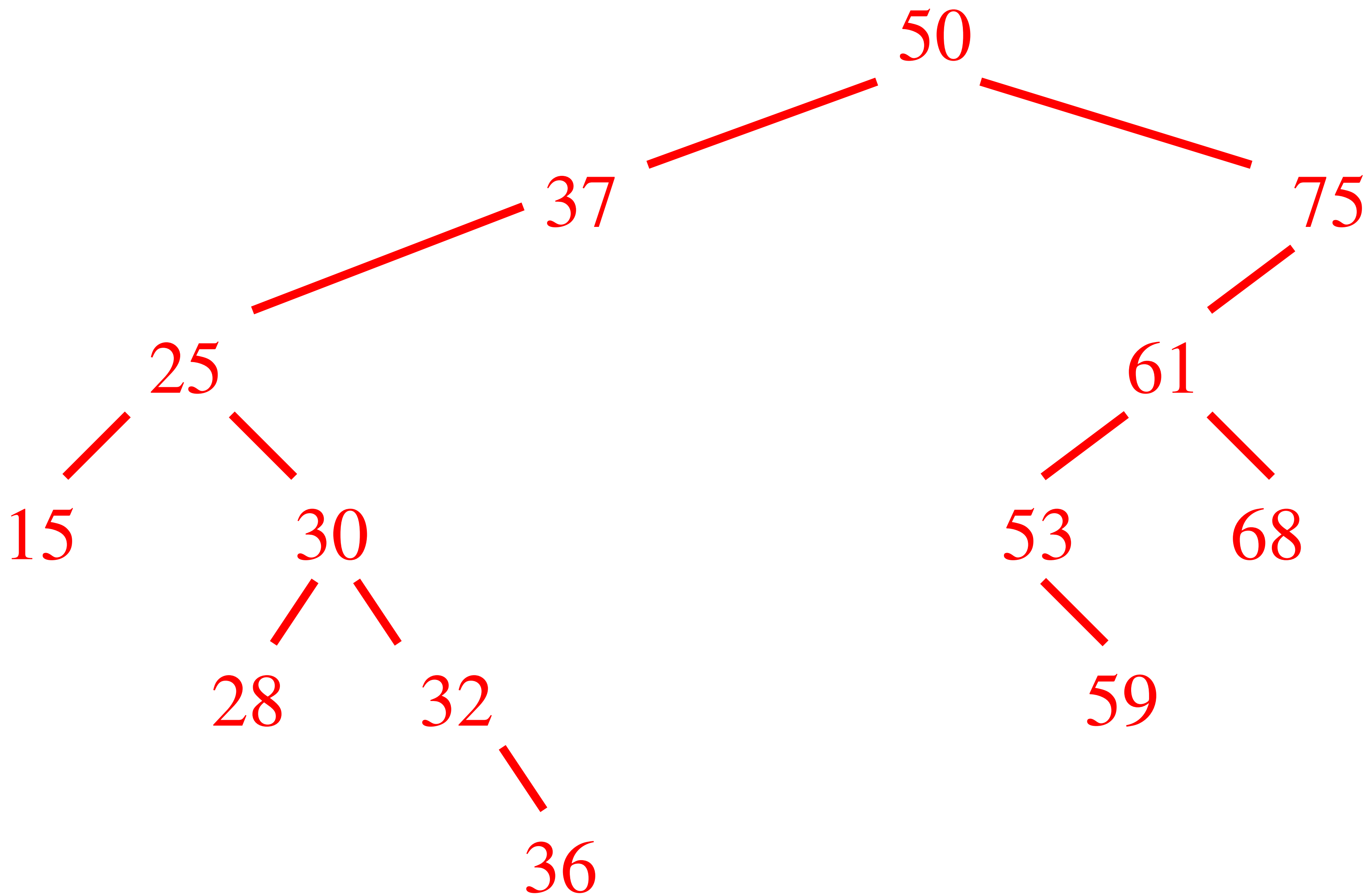
- We will concentrate on one of the most important binary trees, namely the **binary search tree**
- The binary search tree keeps the elements ordered
- We can define a binary search tree recursively

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

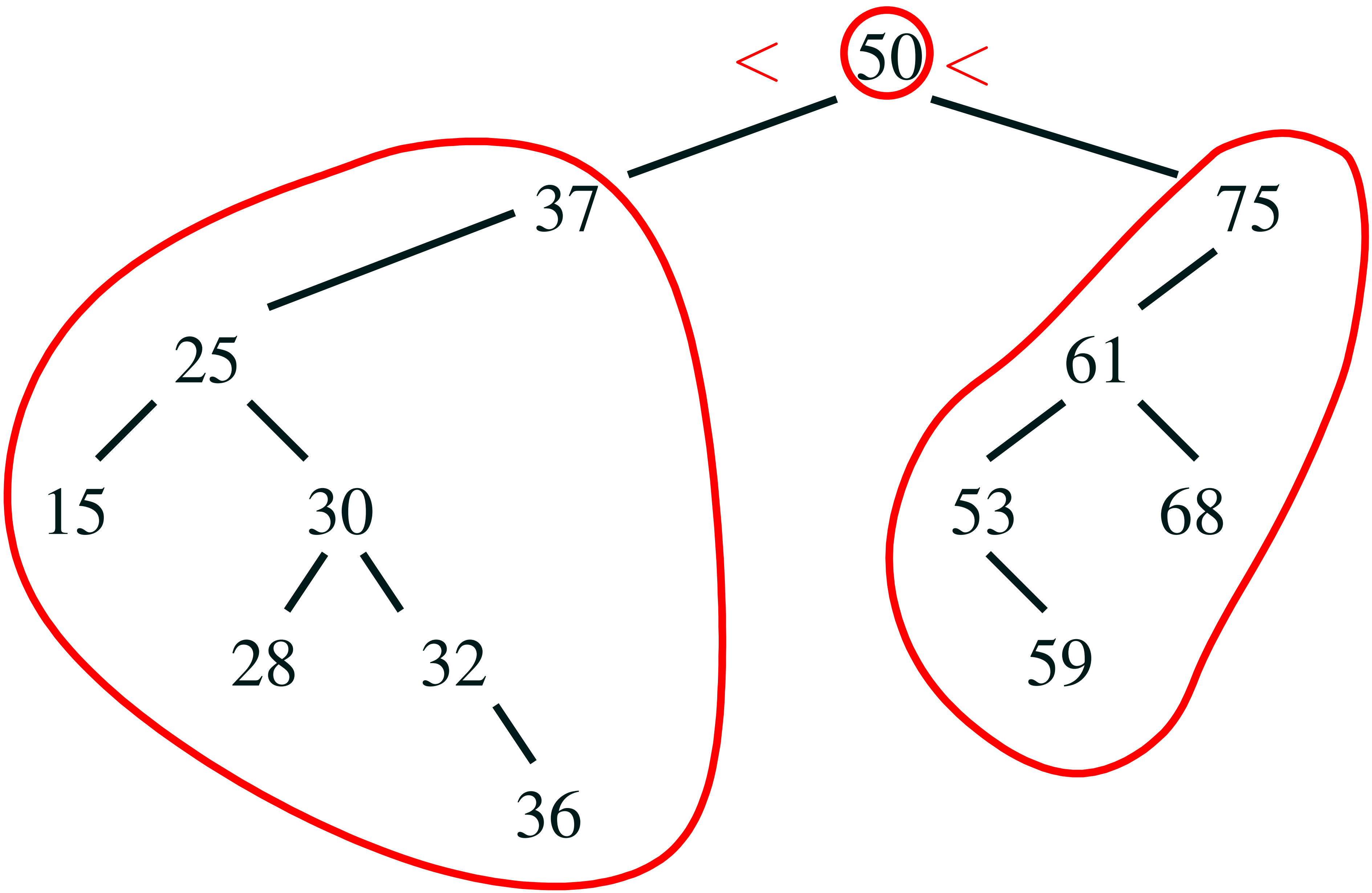
# Binary Search Trees

- 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

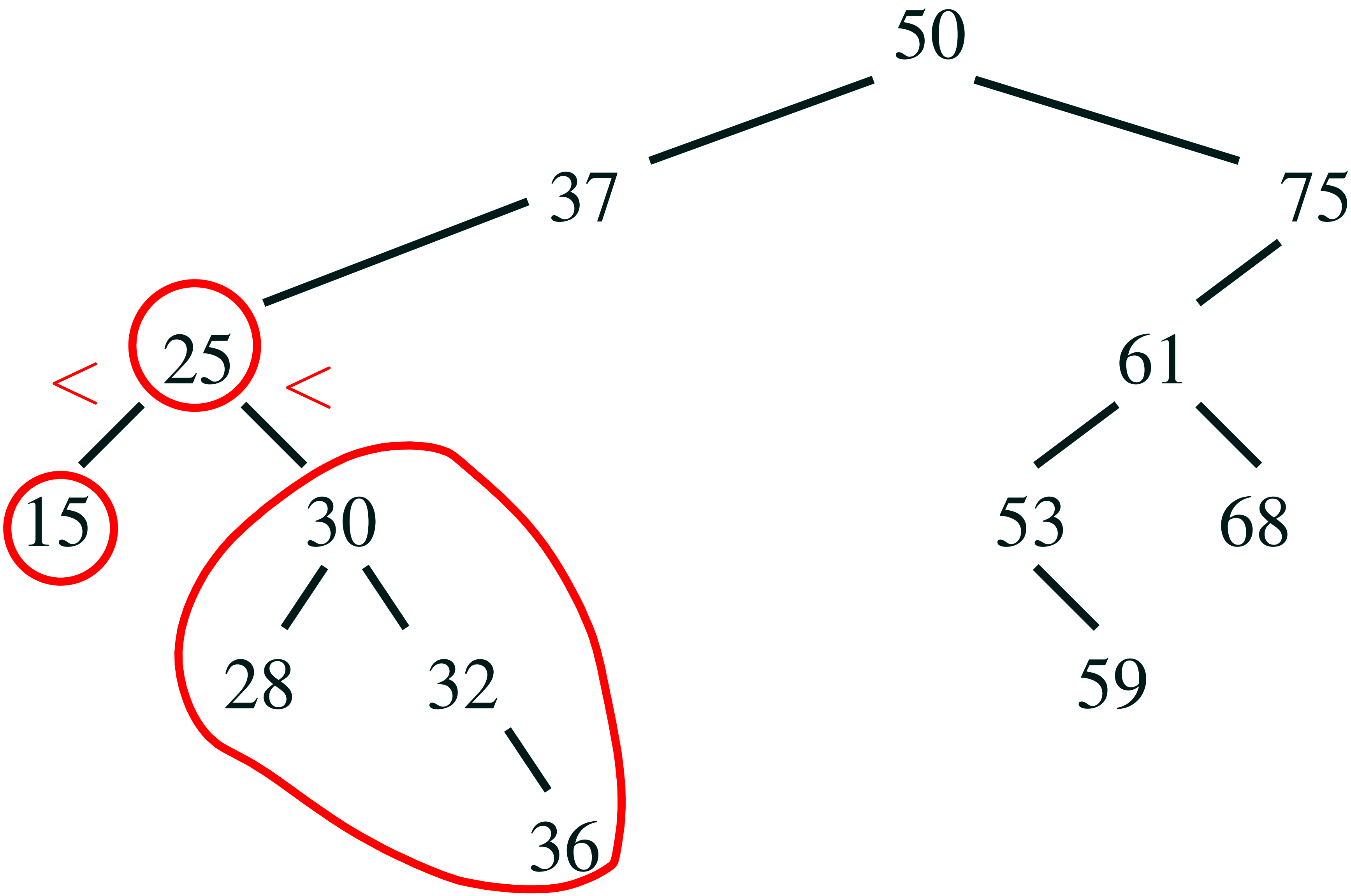
# Example Binary Search Tree



# Example Binary Search Tree

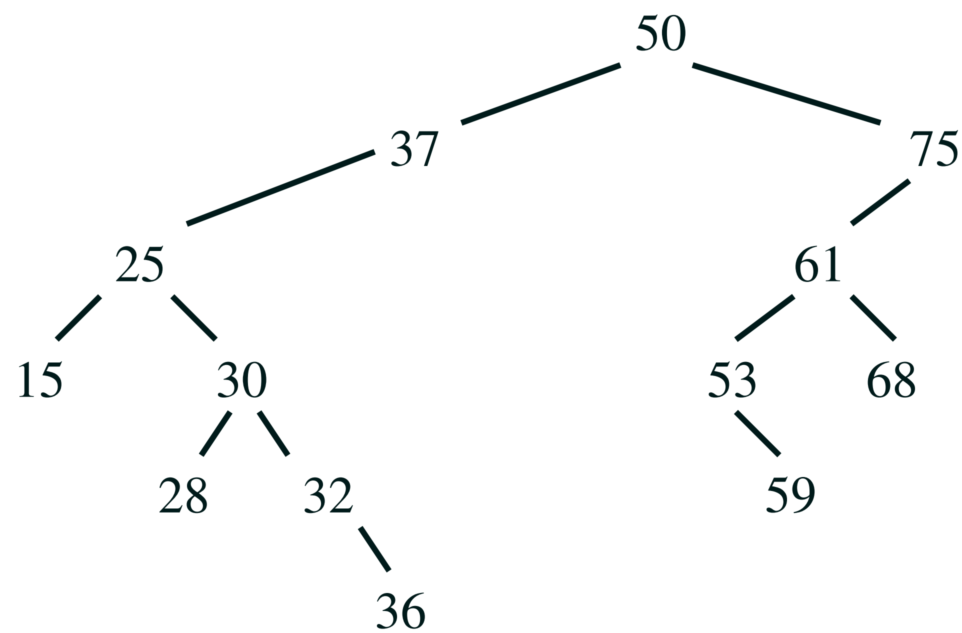


# Example Binary Search Tree



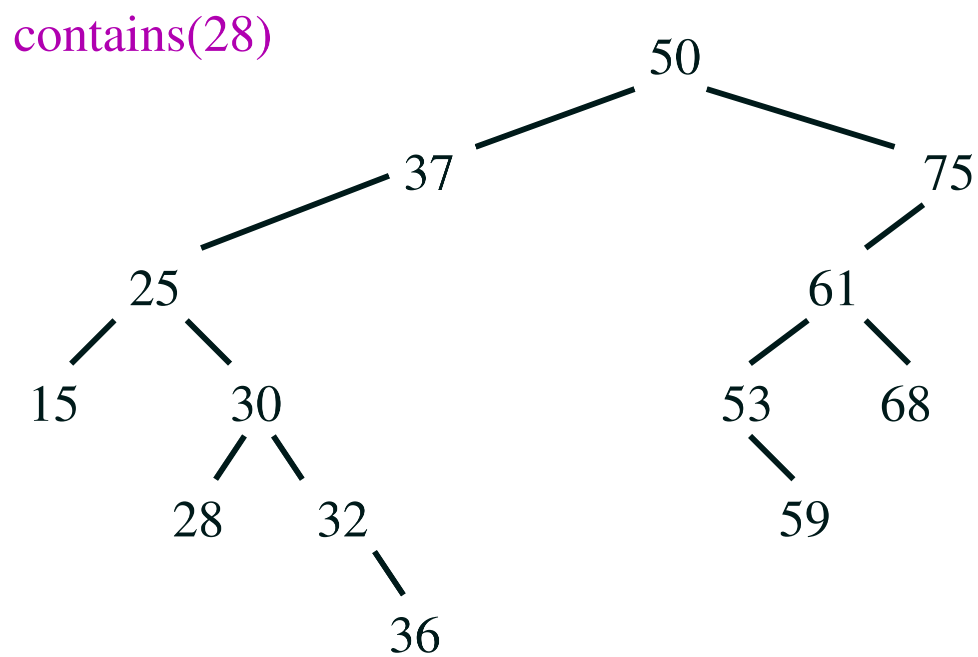
# Searching A Binary Search Tree

- 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



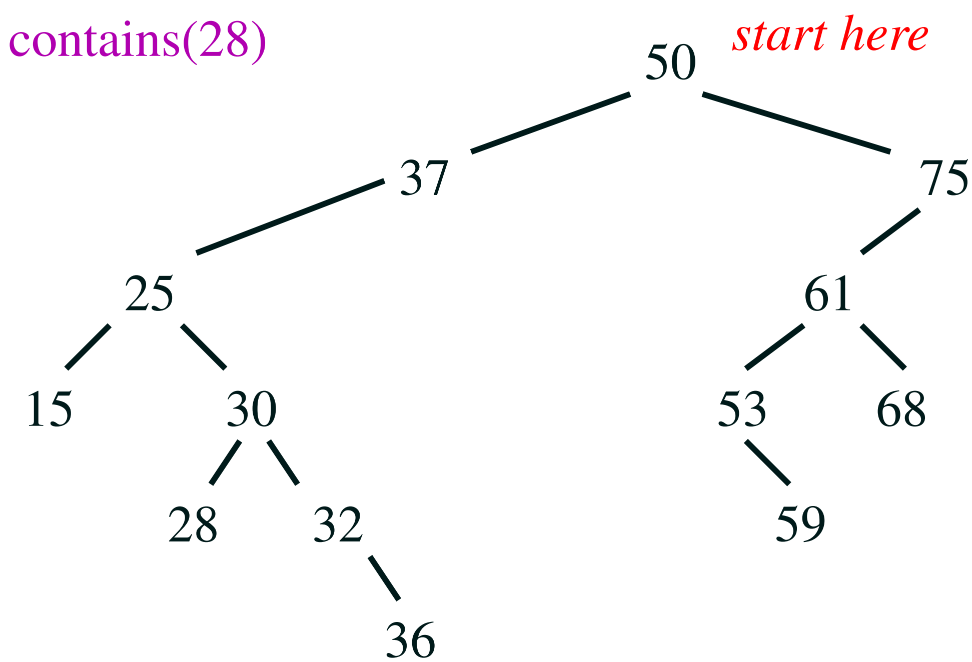
# Searching A Binary Search Tree

- 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



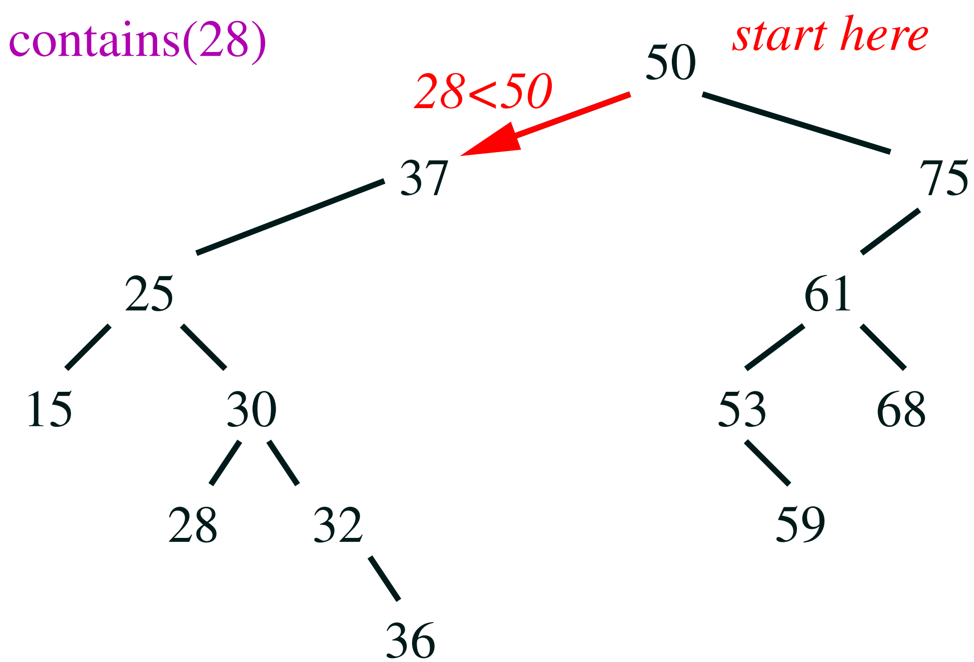
# Searching A Binary Search Tree

- 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



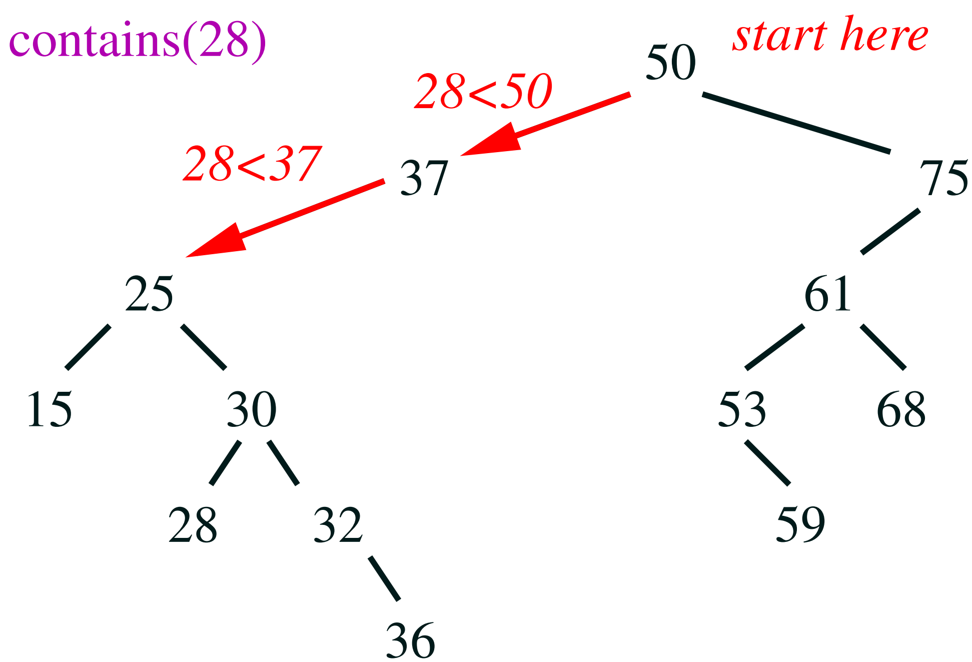
# Searching A Binary Search Tree

- 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



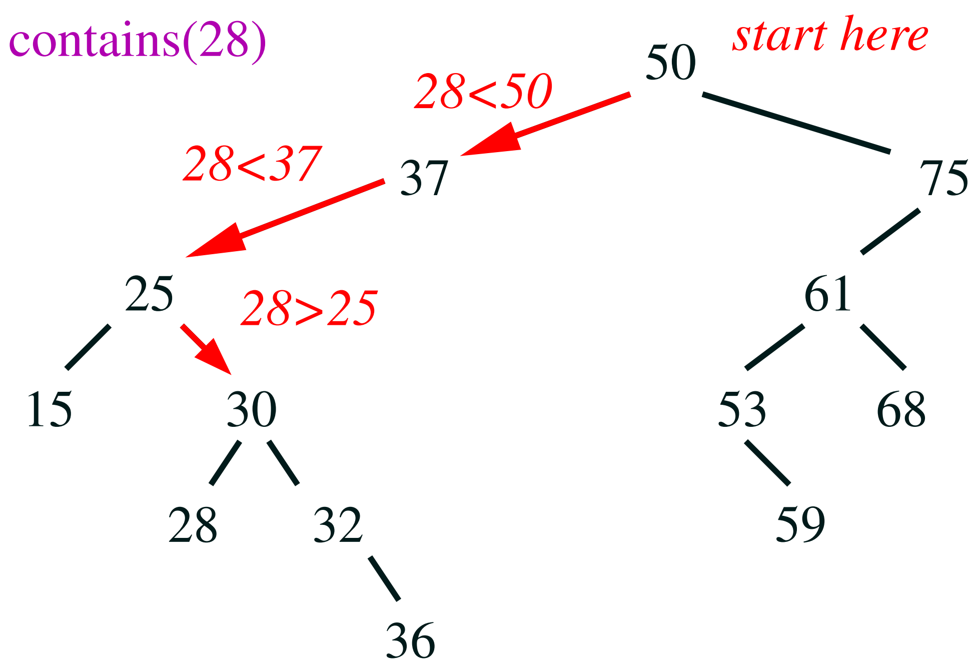
# Searching A Binary Search Tree

- 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



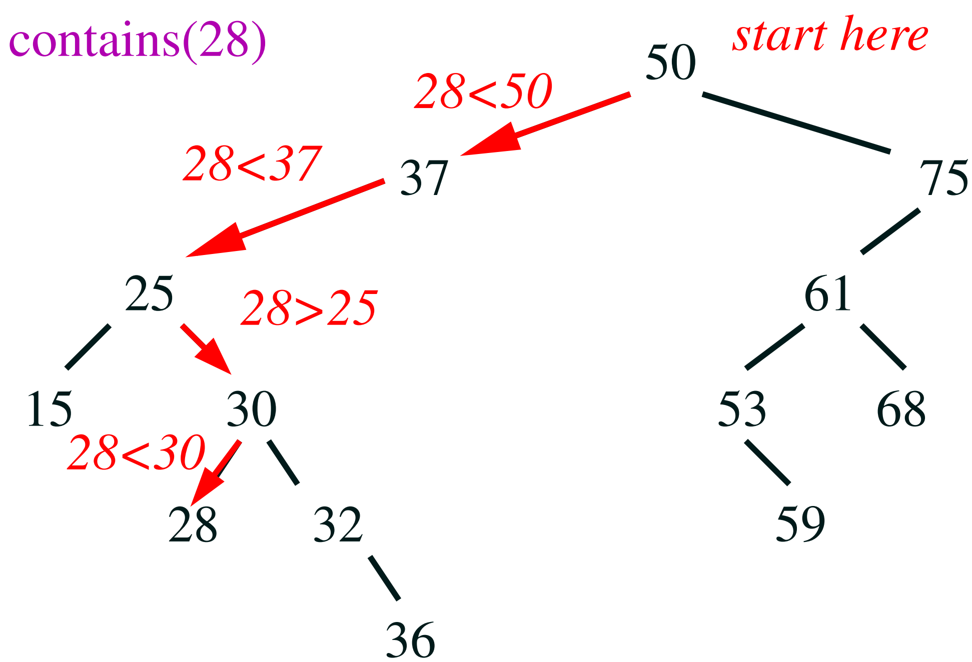
# Searching A Binary Search Tree

- 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



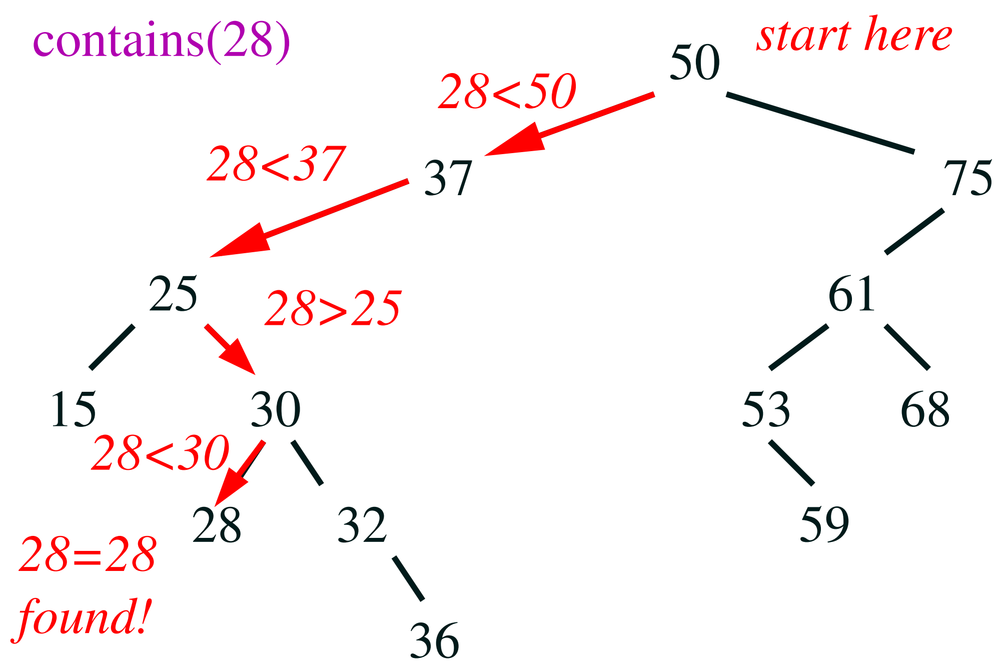
# Searching A Binary Search Tree

- 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



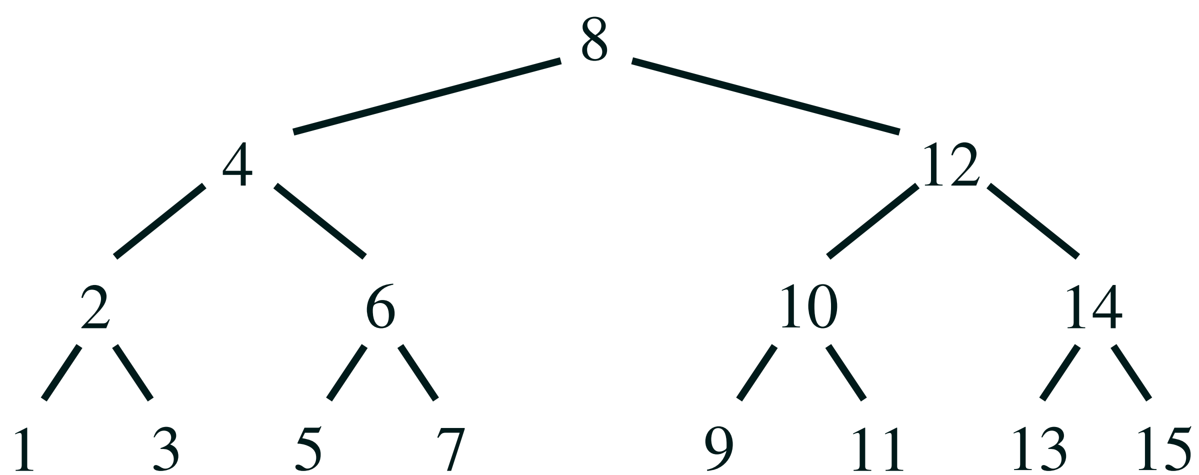
# Searching A Binary Search Tree

- 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

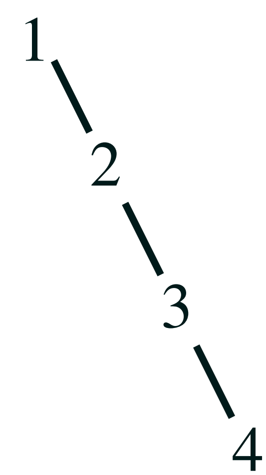


# Speed of Search

- 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



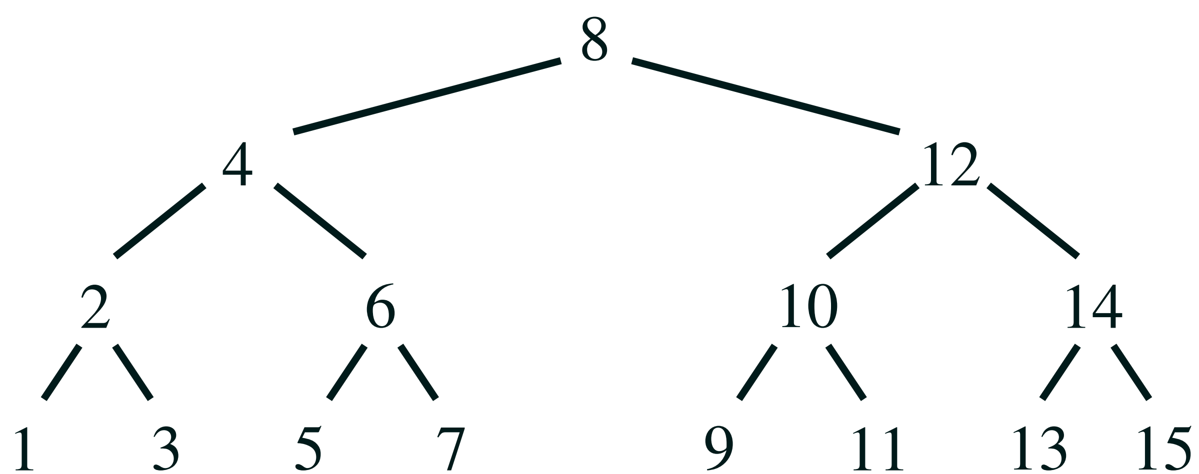

full tree



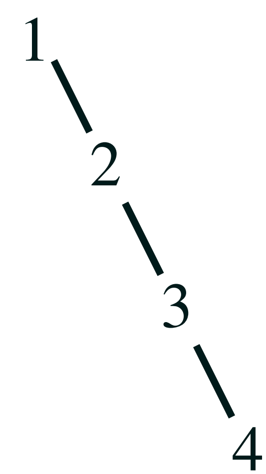

sparse tree

# 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



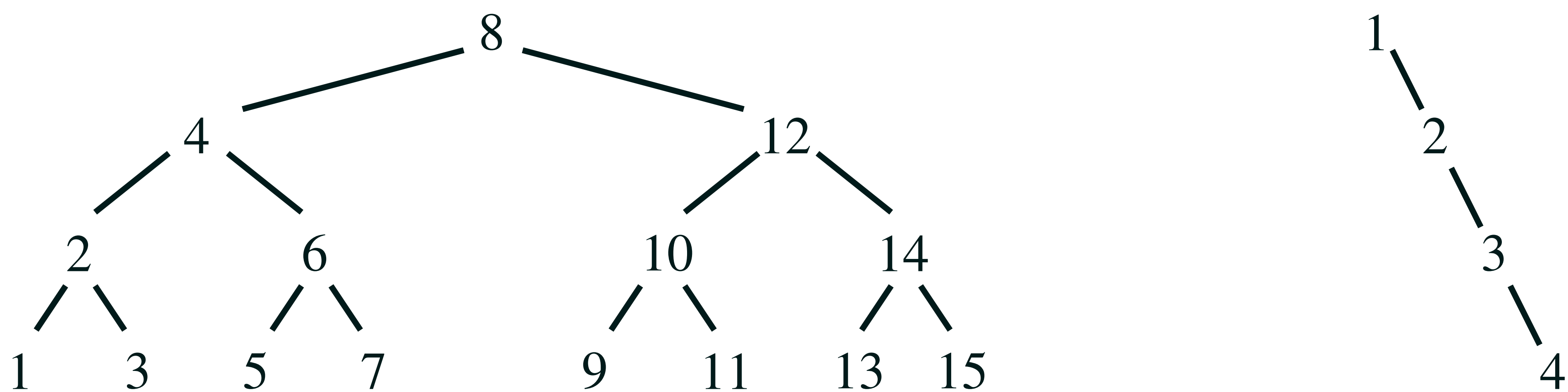

full tree




sparse tree

# 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

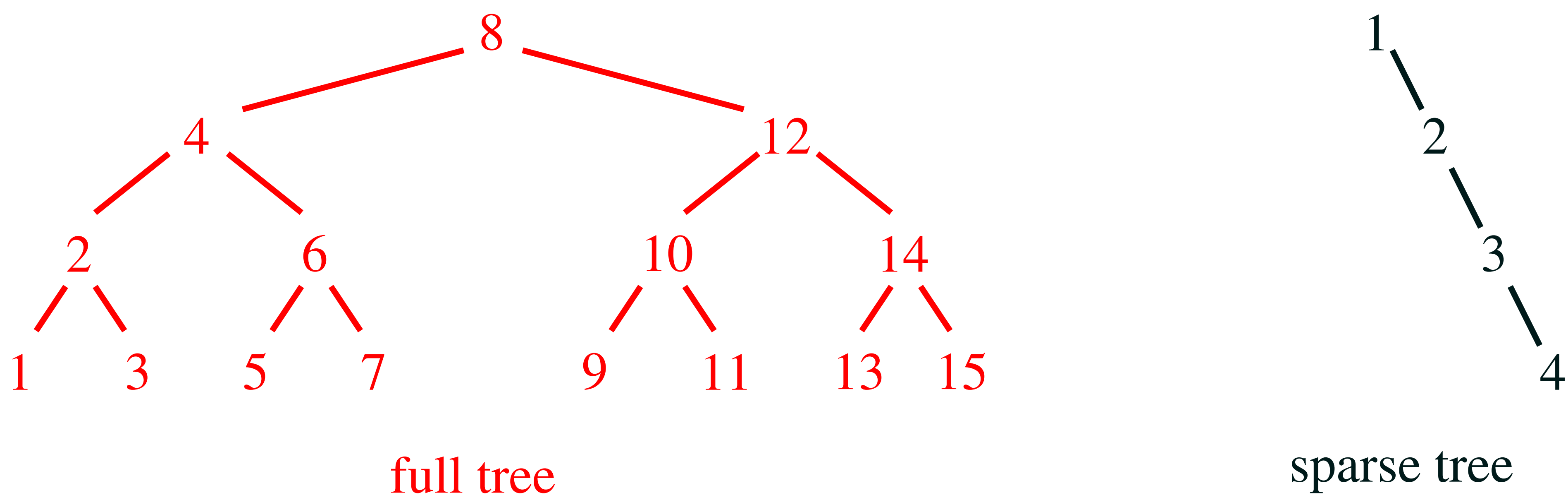



full tree

sparse tree

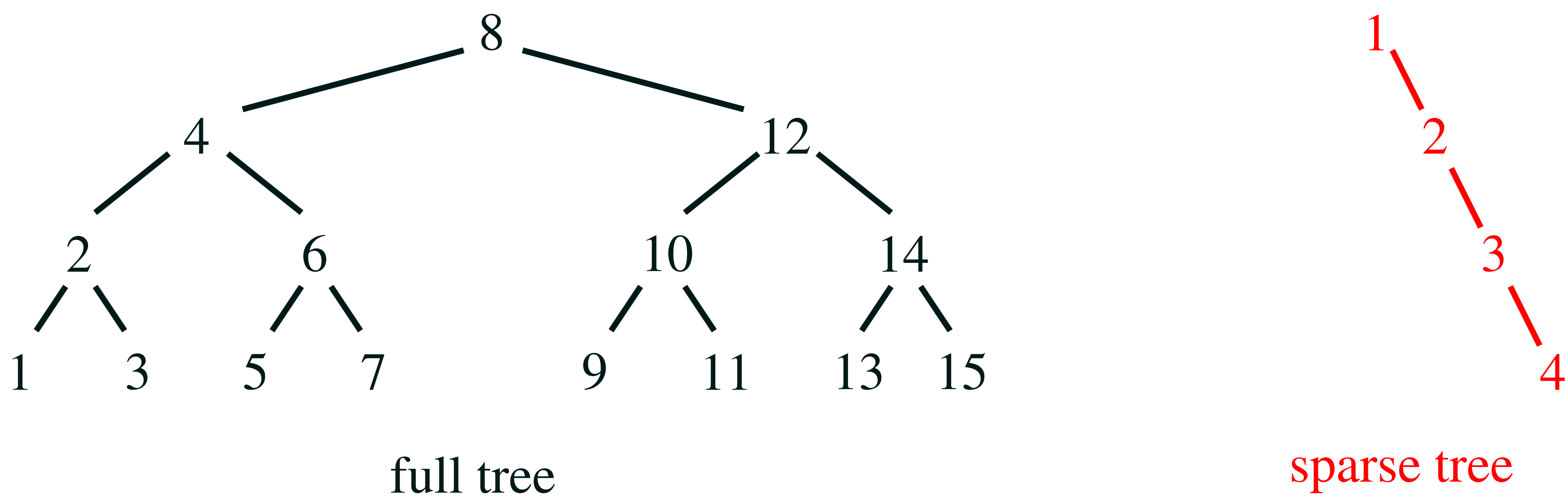
# Speed of Search

- 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



# 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



# 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

# 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

# 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

# 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

# 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

- The standard template library provides a class `std:set<T>`
- This contains many functions like
  - ⋆ Constructors
  - ⋆ `size()`
  - ⋆ `insert(T o)`
  - ⋆ `find(T o)`
  - ⋆ `erase(T o)`
  - ⋆ `begin()` and `end()`

# Fitting In

- The standard template library provides a class `std:set<T>`
- This contains many functions like
  - ⋆ Constructors
  - ⋆ `size()`
  - ⋆ `insert(T o)`
  - ⋆ `find(T o)`
  - ⋆ `erase(T o)`
  - ⋆ `begin()` and `end()`

# Comparable

- To sort any objects they must be comparable
- In the STL the set implementation has a second template parameter: `std::set<T, Compare = less<T>>`
- by default this is defined to be `less<T>` (which is a function already defined for most common types) which you can define
- If you have a set of complex objects you will have to define `Compare`

```
bool MyCompare(MyObject left, MyObject right) {
  return something
}

mySet = set<MyObject, MyCompare>;
```

# Comparable

- To sort any objects they must be comparable
- In the STL the set implementation has a second template parameter: `std::set<T, Compare = less<T>>`
- by default this is defined to be `less<T>` (which is a function already defined for most common types) which you can define
- If you have a set of complex objects you will have to define `Compare`

```
bool MyCompare(MyObject left, MyObject right) {
  return something
}

mySet = set<MyObject, MyCompare>;
```

# Comparable

- To sort any objects they must be comparable
- In the STL the set implementation has a second template parameter: `std::set<T, Compare = less<T>>`
- by default this is defined to be `less<T>` (which is a function already defined for most common types) which you can define
- If you have a set of complex objects you will have to define `Compare`

```
bool MyCompare(MyObject left, MyObject right) {
  return something
}

mySet = set<MyObject, MyCompare>;
```

# Comparable

- To sort any objects they must be comparable
- In the STL the set implementation has a second template parameter: `std::set<T, Compare = less<T>>`
- by default this is defined to be `less<T>` (which is a function already defined for most common types) which you can define
- If you have a set of complex objects you will have to define `Compare`

```
bool MyCompare(MyObject left, MyObject right) {
  return something
}

mySet = set<MyObject, MyCompare>;
```

# Find an Element

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

```
iterator find(const T& element) {
  Node* current = root;
  while (current!=nullptr) {
    if (current->element == element) {
      return iterator(current);
    }
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
  return iterator(nullptr);
}
```

# Find an Element

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

```
iterator find(const T& element) {
  Node* current = root;
  while (current!=nullptr) {
    if (current->element == element) {
      return iterator(current);
    }
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
  return iterator(nullptr);
}
```

# Find an Element

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

```
iterator find(const T& element) {
  Node* current = root;
  while (current!=nullptr) {
    if (current->element == element) {
      return iterator(current);
    }
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
  return iterator(nullptr);
}
```

# Find an Element

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

```
iterator find(const T& element) {
  Node* current = root;
  while (current!=nullptr) {
    if (current->element == element) {
      return iterator(current);
    }
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
  return iterator(nullptr);
}
```

# Find an Element

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

```
iterator find(const T& element) {
  Node* current = root;
  while (current!=nullptr) {
    if (current->element == element) {
      return iterator(current);
    }
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
  return iterator(nullptr);
}
```

# Add an Element

```
pair<iterator, bool> insert(const T& element) {
  if (no_elements==0) {
    root = new Node(element, nullptr);
    ++no_elements;
    return pair<iterator, bool>(iterator(root), true);
  }
  Node* parent = nullptr;
  Node* current = root;
  while(current != nullptr) {
    if (current->element == element) {
      return pair<iterator, bool>(iterator(nullptr), false);
    }
    parent = current;
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
```

# Add an Element

```
pair<iterator, bool> insert(const T& element) {
  if (no_elements==0) {
    root = new Node(element, nullptr);
    ++no_elements;
    return pair<iterator, bool>(iterator(root), true);
  }
  Node* parent = nullptr;
  Node* current = root;
  while(current != nullptr) {
    if (current->element == element) {
      return pair<iterator, bool>(iterator(nullptr), false);
    }
    parent = current;
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
```

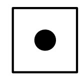
# Add an Element

```
pair<iterator, bool> insert(const T& element) {
  if (no_elements==0) {
    root = new Node(element, nullptr);
    ++no_elements;
    return pair<iterator, bool>(iterator(root), true);
  }
  Node* parent = nullptr;
  Node* current = root;
  while(current != nullptr) {
    if (current->element == element) {
      return pair<iterator, bool>(iterator(nullptr), false);
    }
    parent = current;
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
```

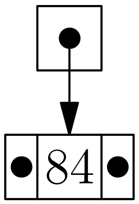
```
  current = new Node(element, parent);
  if (element < parent->element) {
    parent->left = current;
  } else {
    parent->right = current;
  }
  ++no_elements;
  return pair<iterator, bool>(iterator(current), true);
}
```

# Tree in Action

add(84)

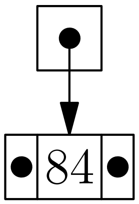


# Tree in Action

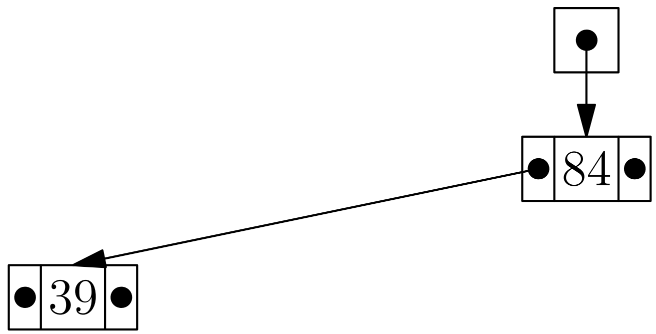


# Tree in Action

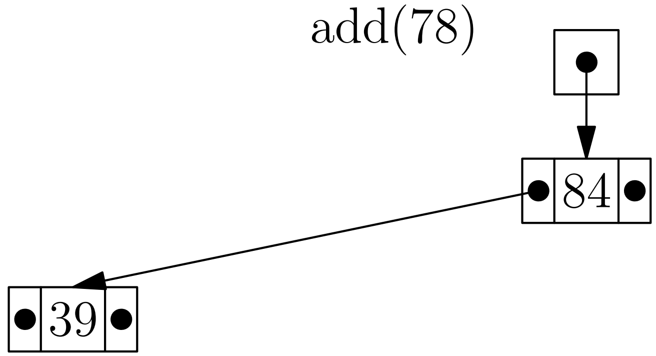
add(39)



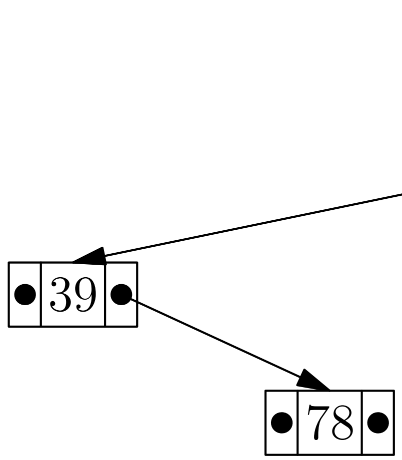
# Tree in Action



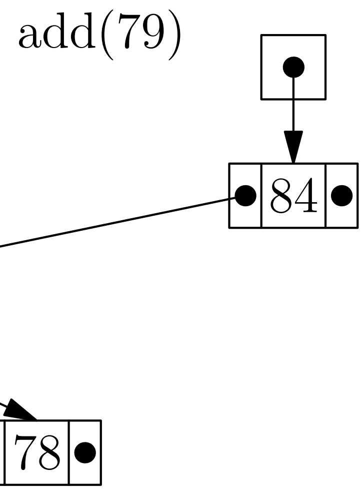
# Tree in Action



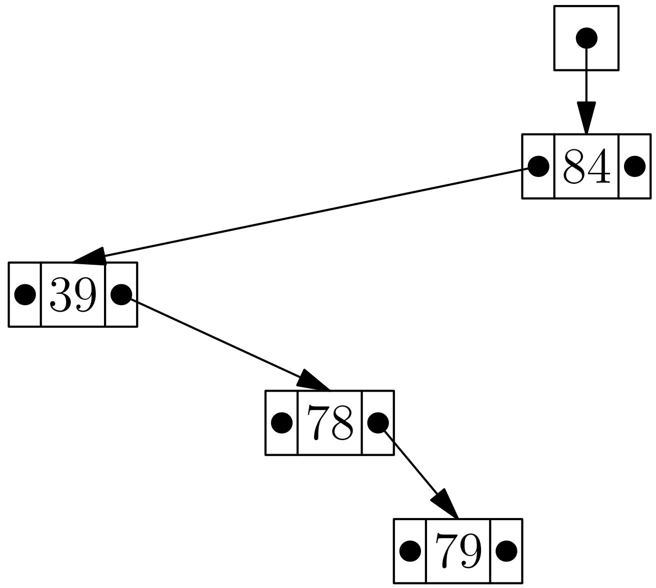
# Tree in Action



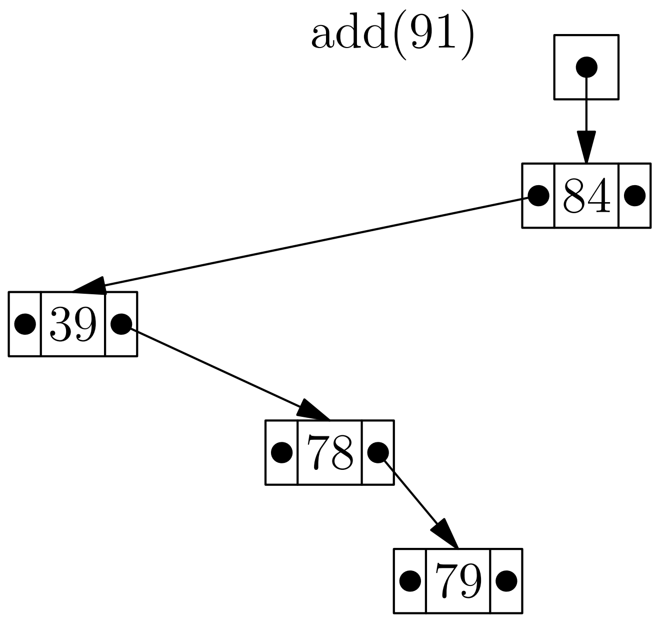
# Tree in Action



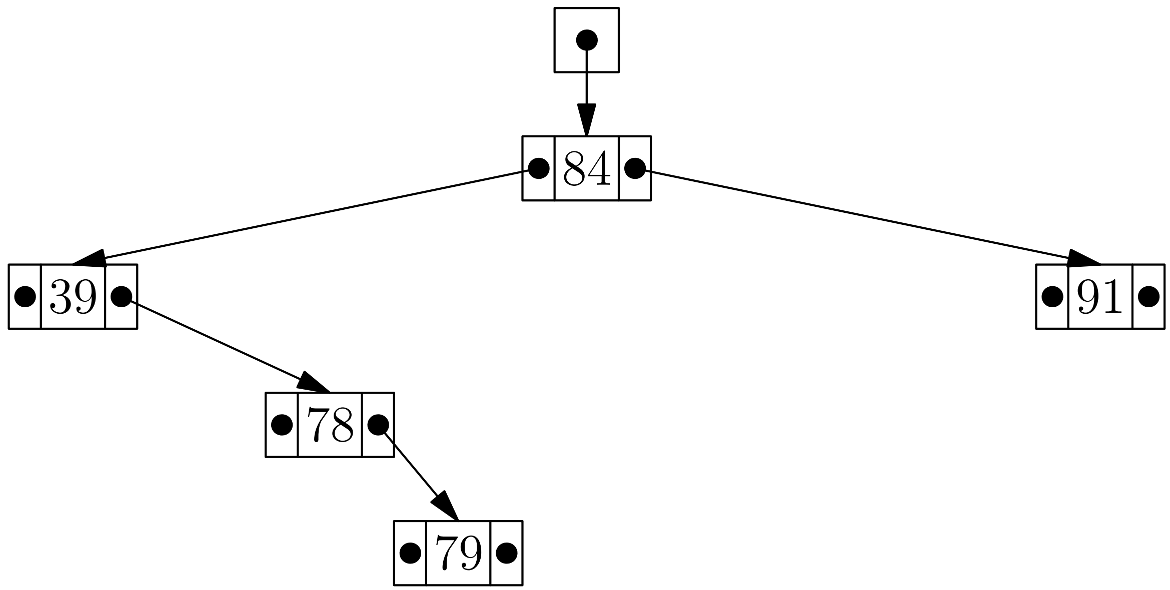
# Tree in Action



# Tree in Action

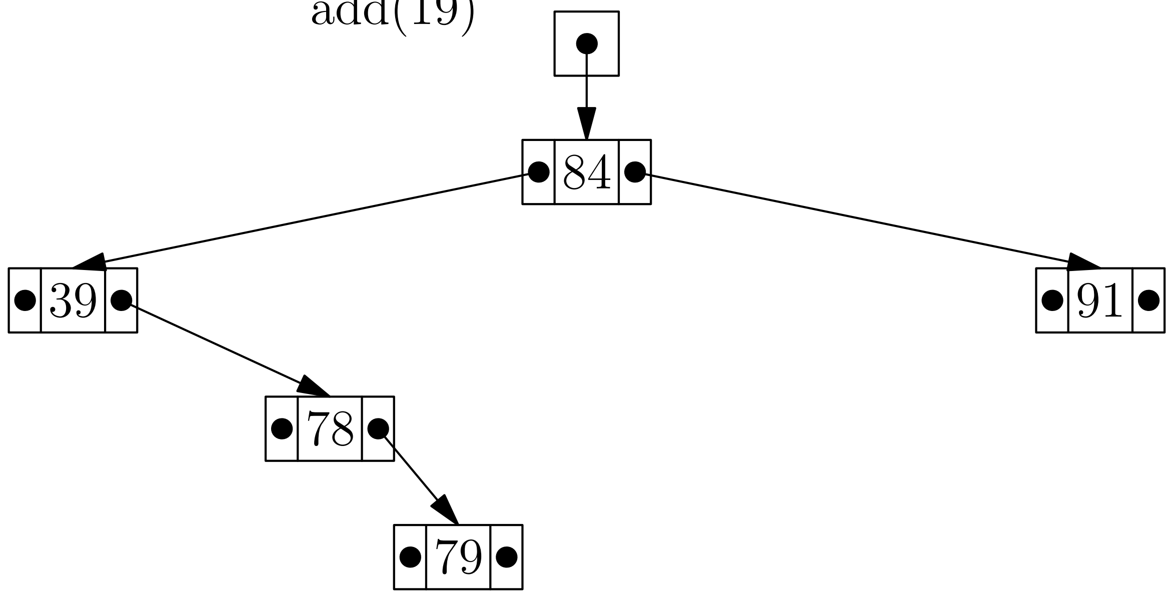


# Tree in Action

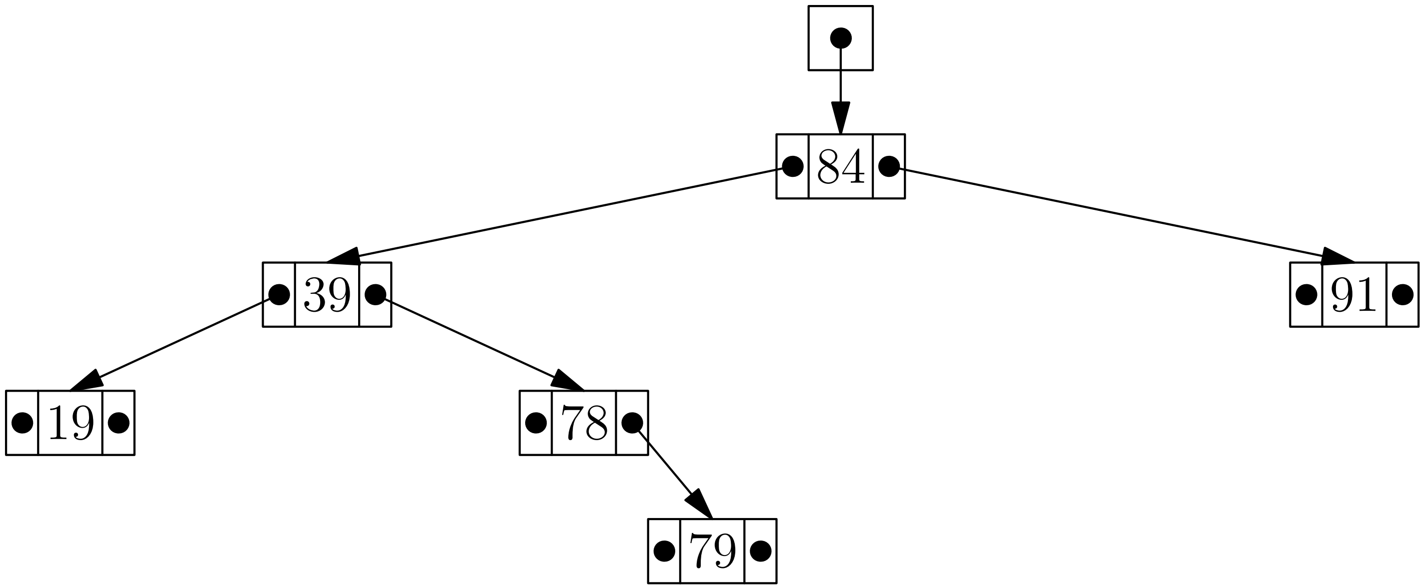


# Tree in Action

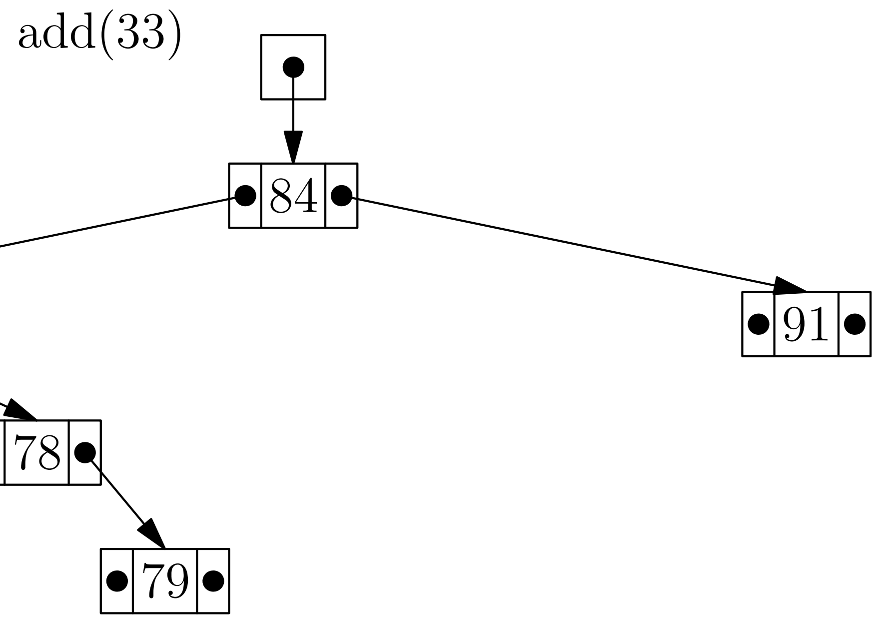
add(19)



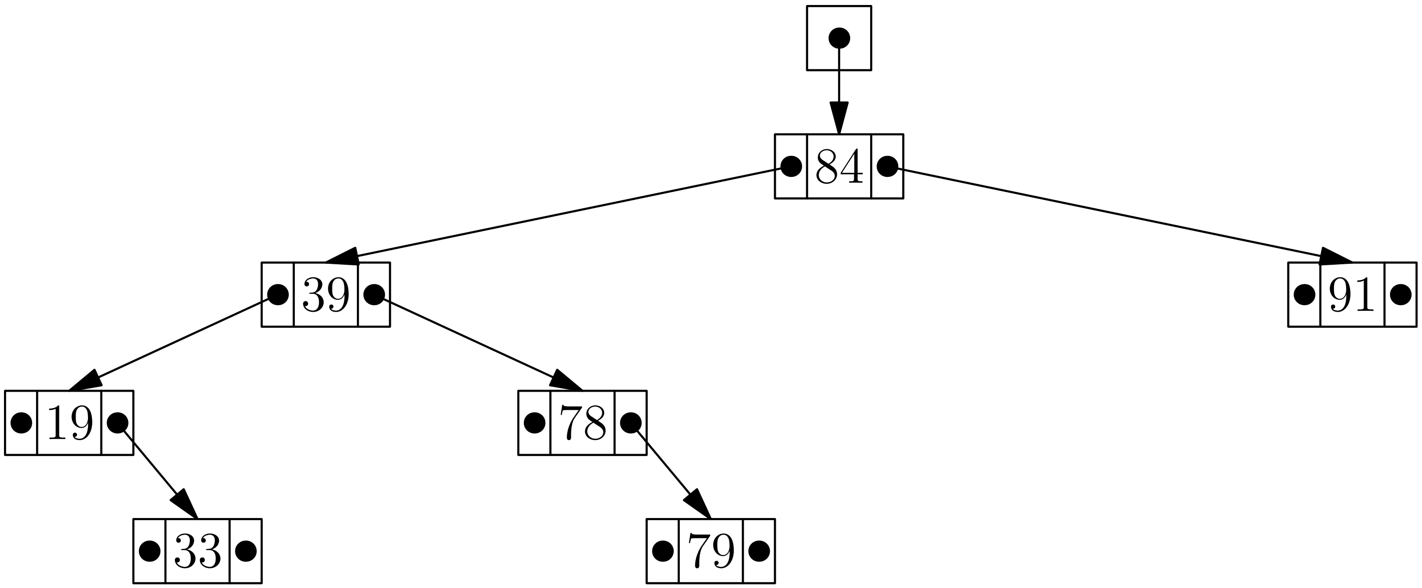
# Tree in Action



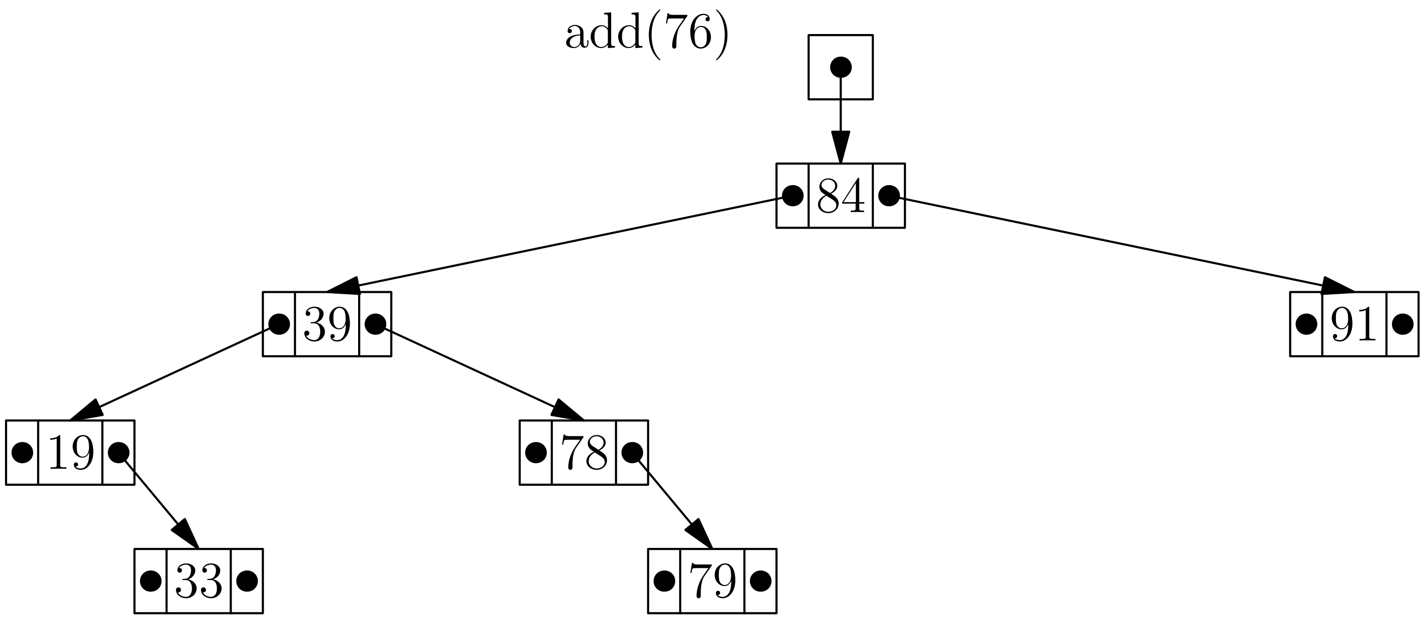
# Tree in Action



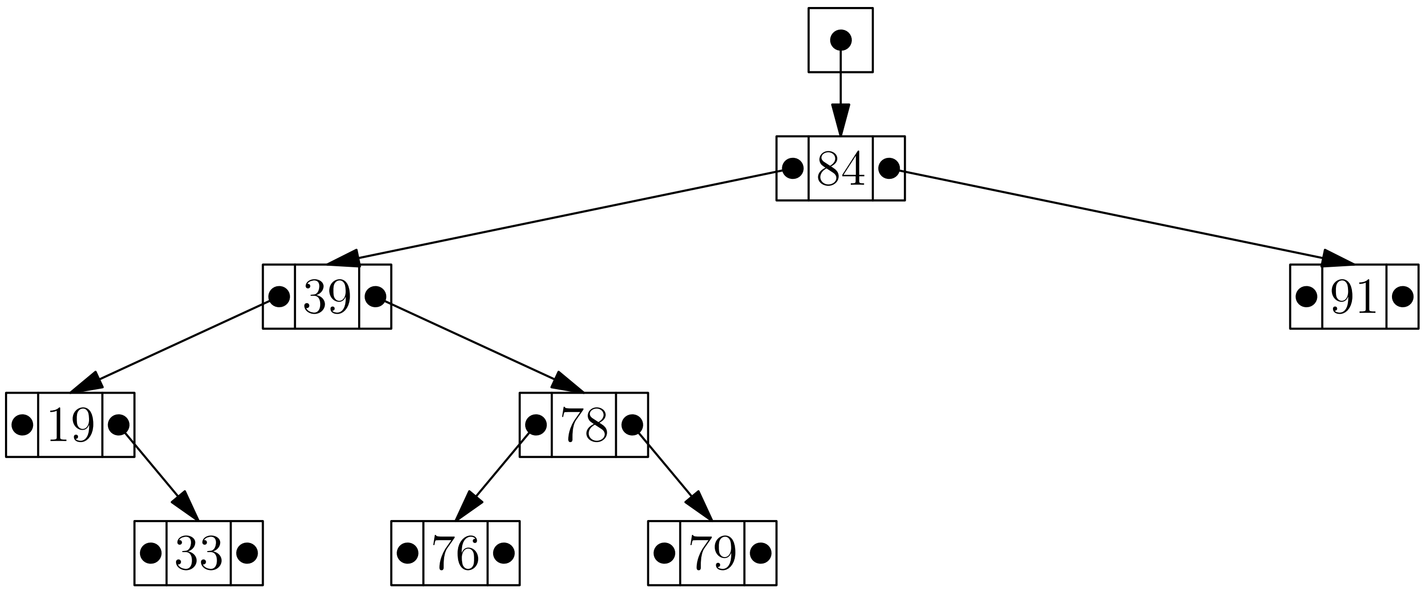
# Tree in Action



# Tree in Action

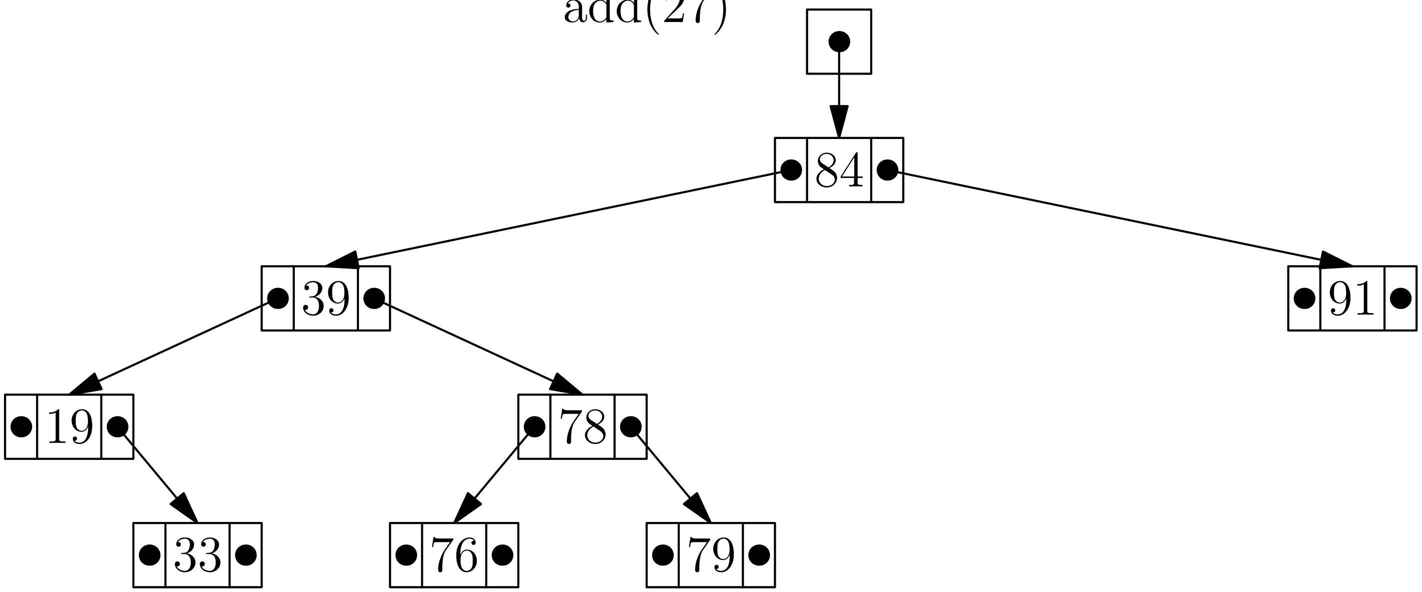


# Tree in Action

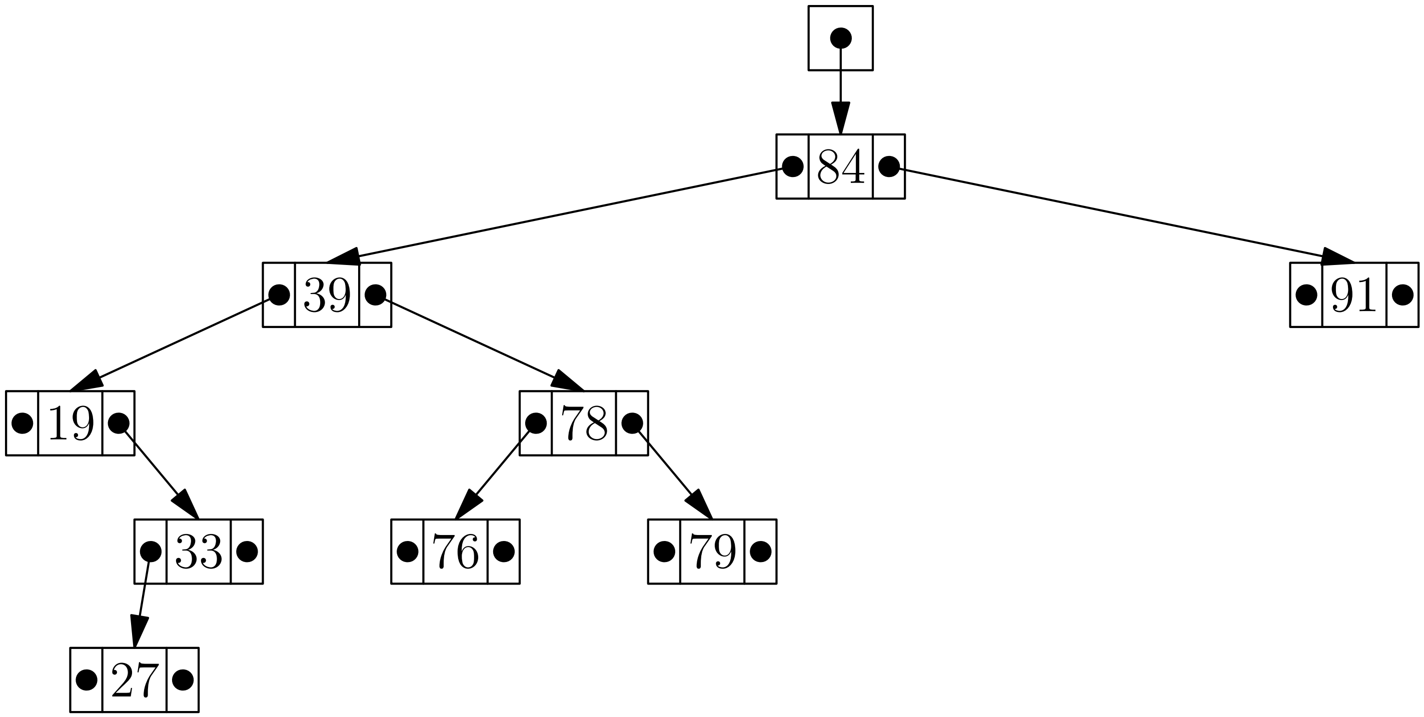


# Tree in Action

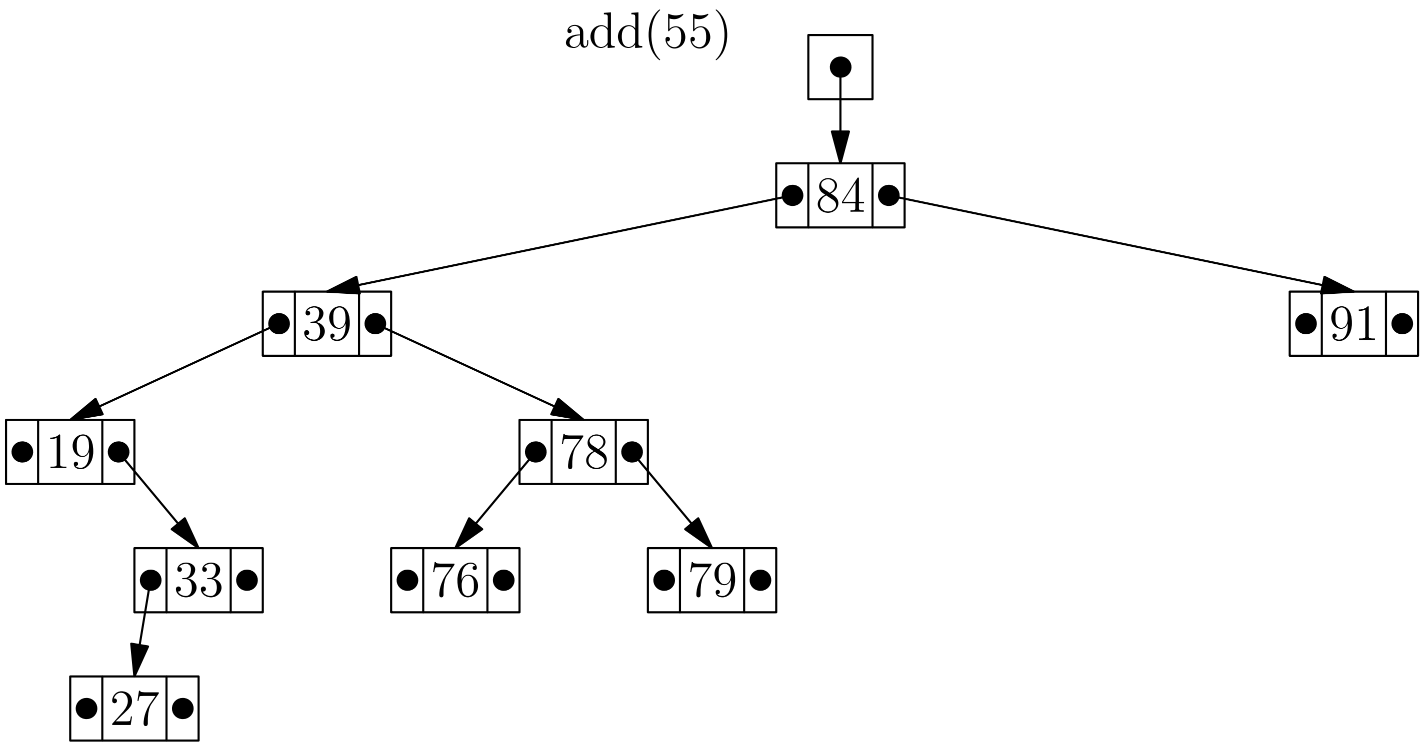
add(27)



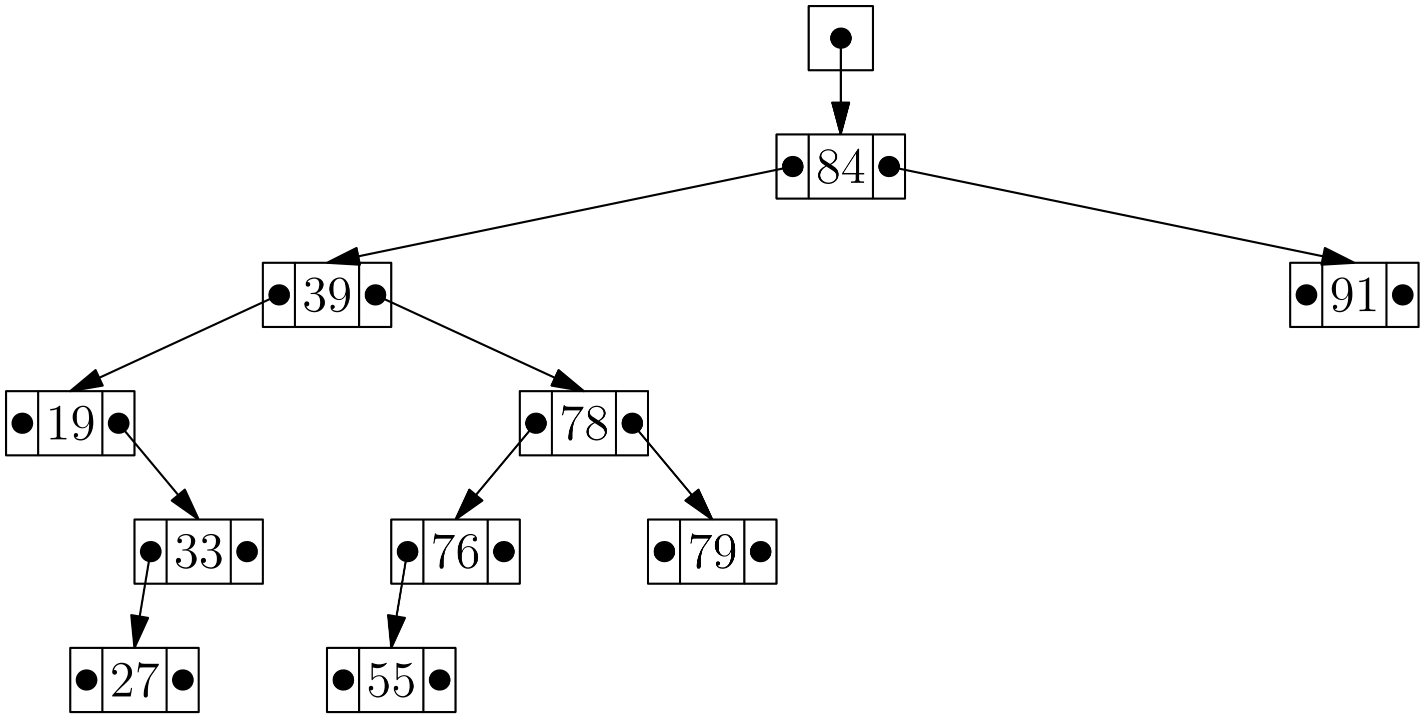
# Tree in Action



# Tree in Action



# 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

# 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

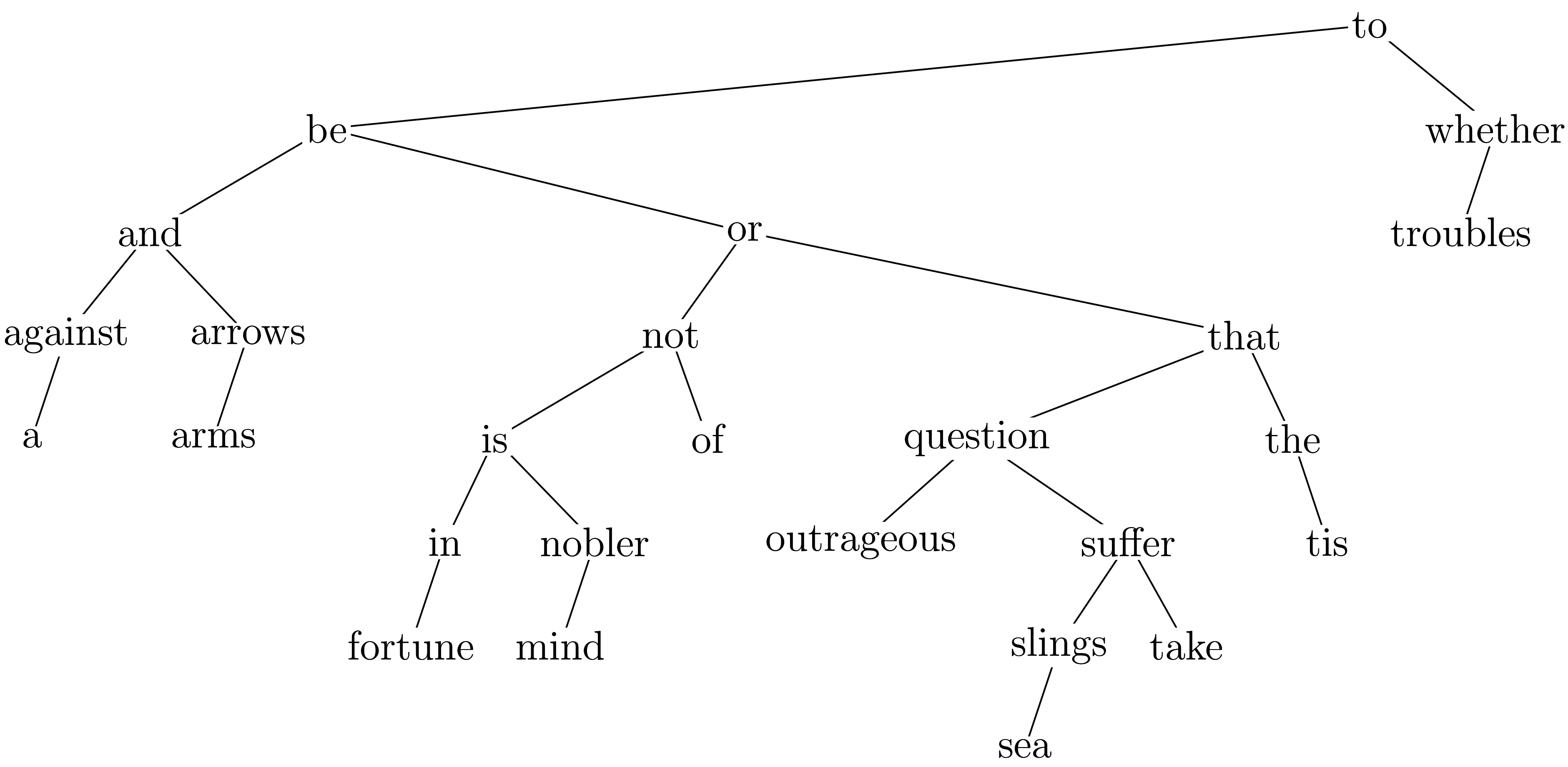
# Shape of Tree

- 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

- As with most container classes it is very useful to define iterators
- `begin()` should return a “pointer” to the start of the tree
- `end()` provides a “pointer” past the end
- `operator*()` returns the element
- `opeator++()` increments the “pointer”
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```

# Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a "pointer" to the start of the tree
- `end()` provides a "pointer" past the end
- `operator*()` returns the element
- `opeator++()` increments the "pointer"
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```

# Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a "pointer" to the start of the tree
- `end()` provides a "pointer" past the end
- `operator*()` returns the element
- `opeator++()` increments the "pointer"
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```

# Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a "pointer" to the start of the tree
- `end()` provides a "pointer" past the end
- `operator*()` returns the element
- `opeator++()` increments the "pointer"
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```

# Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a “pointer” to the start of the tree
- `end()` provides a “pointer” past the end
- `operator*()` returns the element
- `opeator++()` increments the “pointer”
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```

# Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a "pointer" to the start of the tree
- `end()` provides a "pointer" past the end
- `operator*()` returns the element
- `opeator++()` increments the "pointer"
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```

# Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a "pointer" to the start of the tree
- `end()` provides a "pointer" past the end
- `operator*()` returns the element
- `opeator++()` increments the "pointer"
- `operator!=(lhs, rhs)` is used to compare iterators

```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```

# C++ Code

```
class binary_tree {

private:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(nullptr)}

};
```

# C++ Code

```
class binary_tree {

private:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(nullptr)}

};
```

# C++ Code

```
class binary_tree {

private:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(nullptr)}

};
```

# C++ Code

```
class binary_tree {

private:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(nullptr)}

};
```

# C++ Code

```
class binary_tree {

private:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(nullptr)}

};
```

# C++ Code

```
class binary_tree {

private:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(nullptr)}

};
```

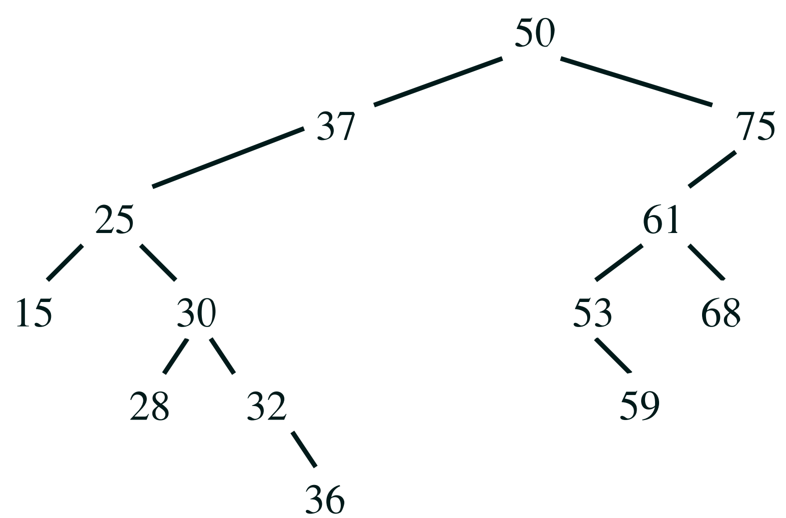
# C++ Code

```
class binary_tree {

private:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(nullptr)}

};
```

# Successor

- To find the successor we first start in the left most branch
- 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

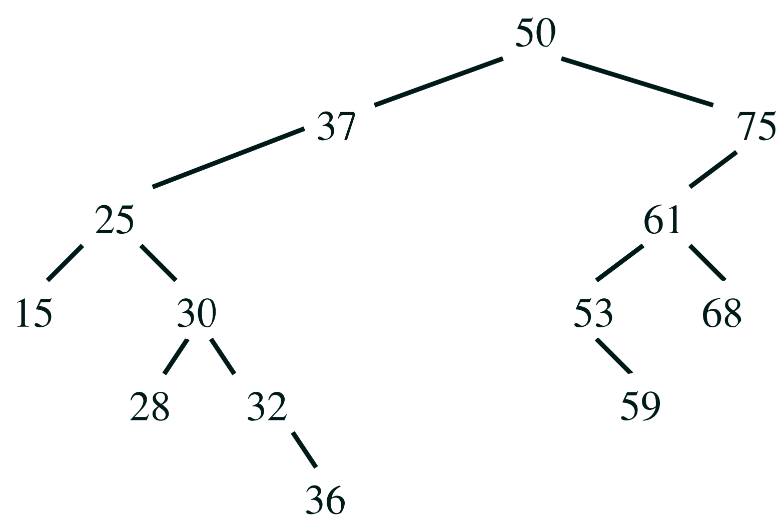



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

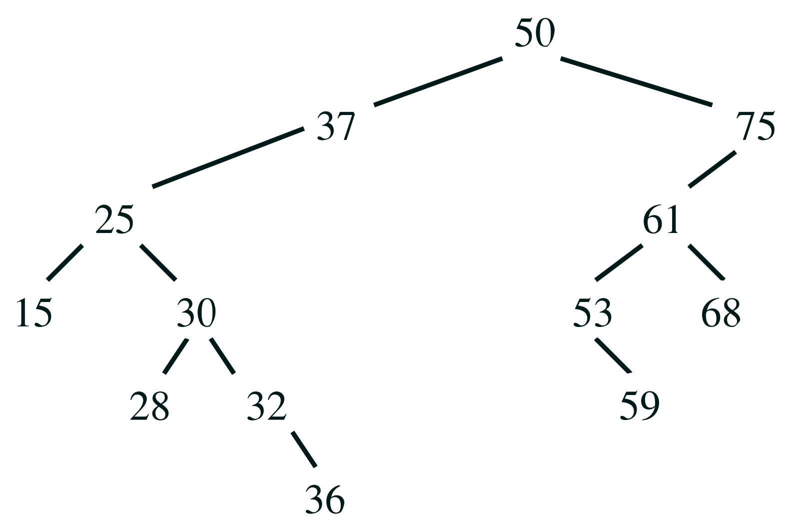



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

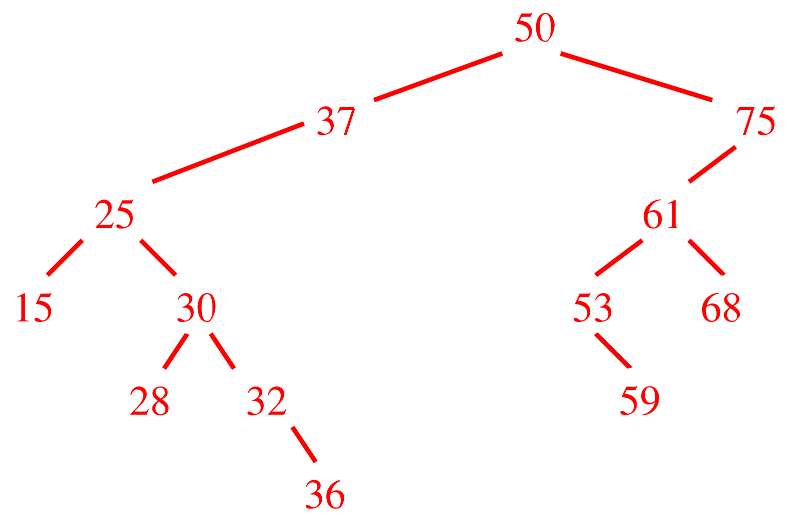



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

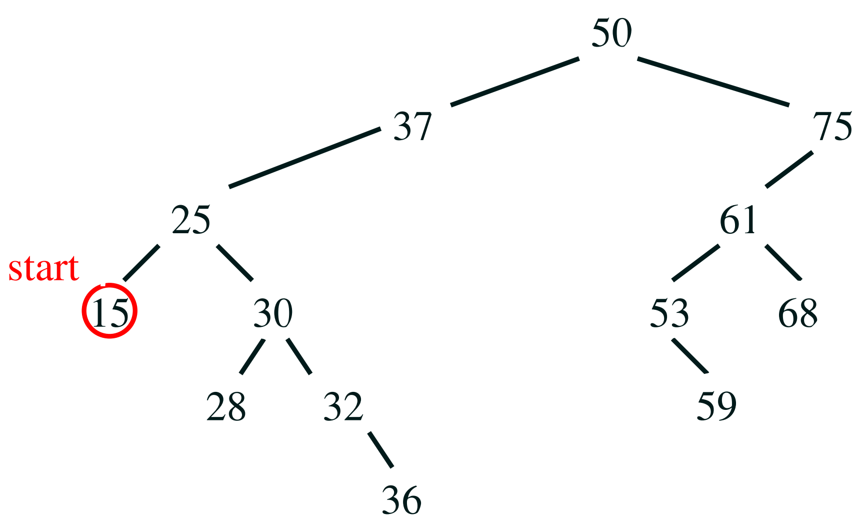



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

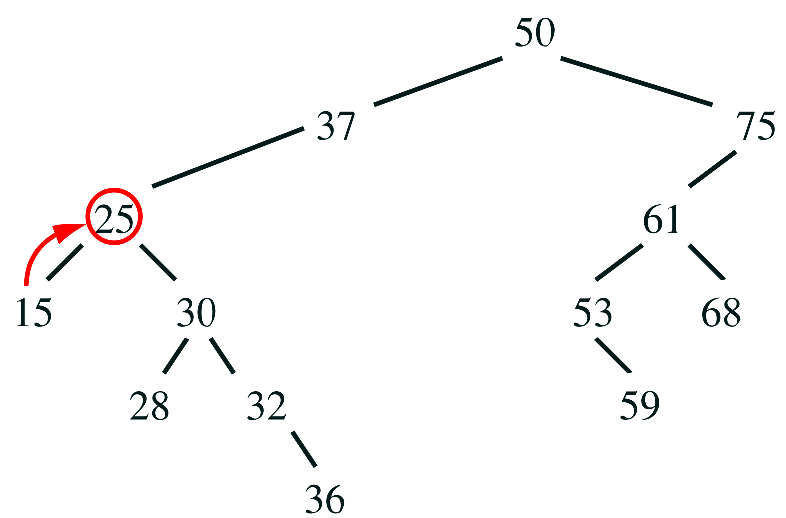



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

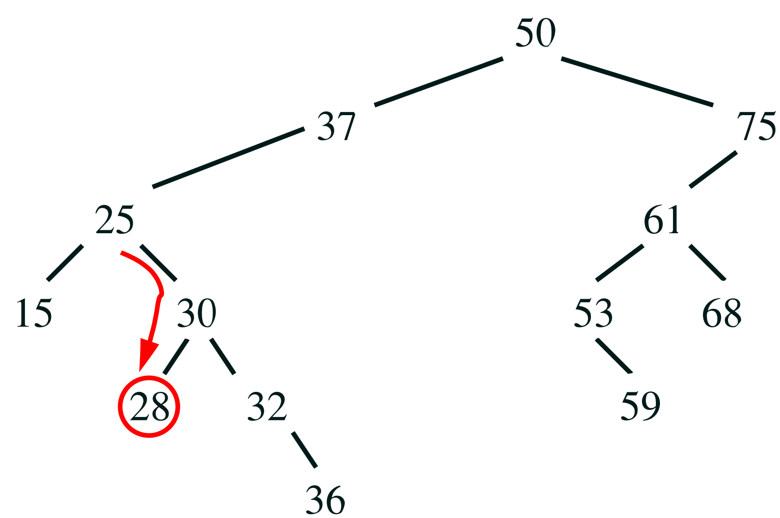



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

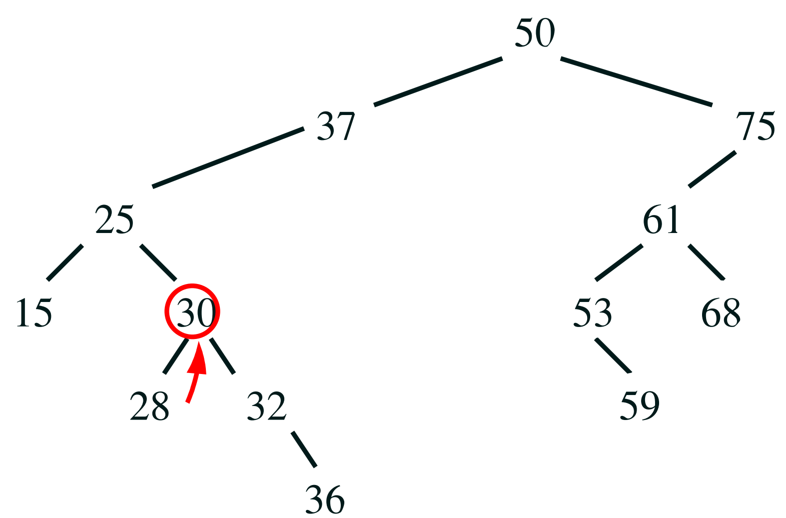



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

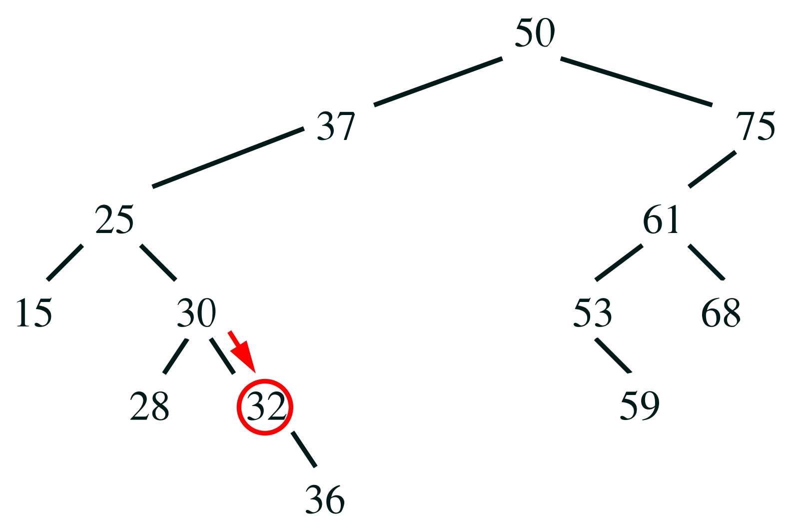



$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$

# Successor

- To find the successor we first start in the left most branch
- 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

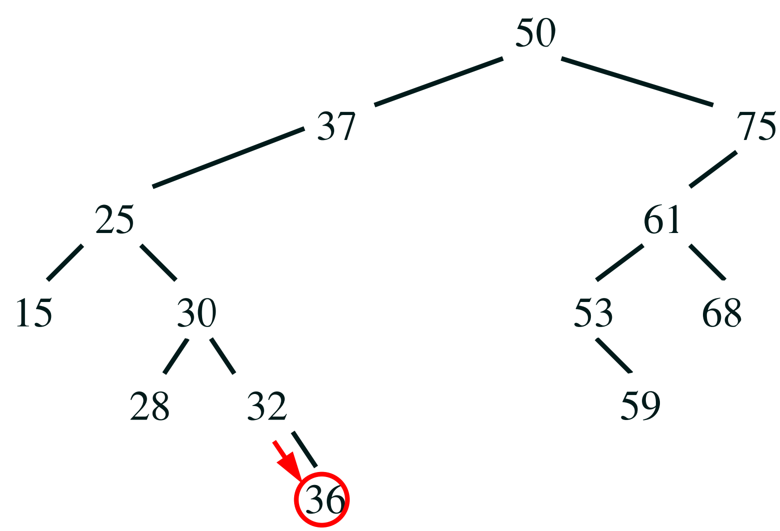



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

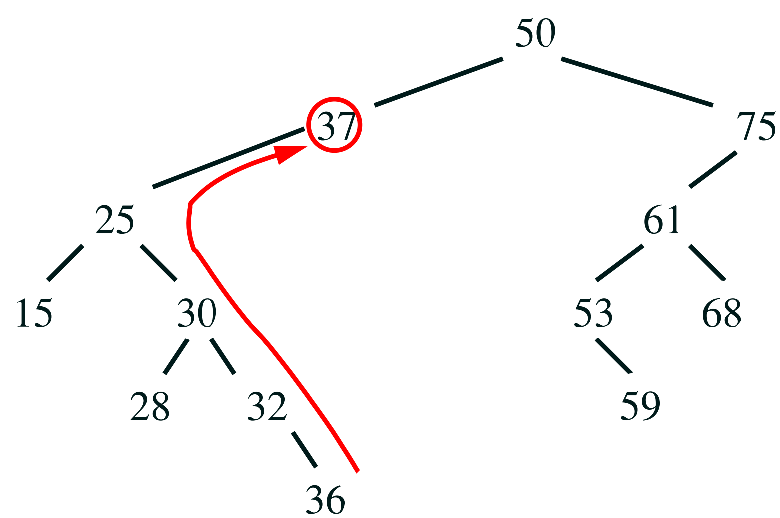



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

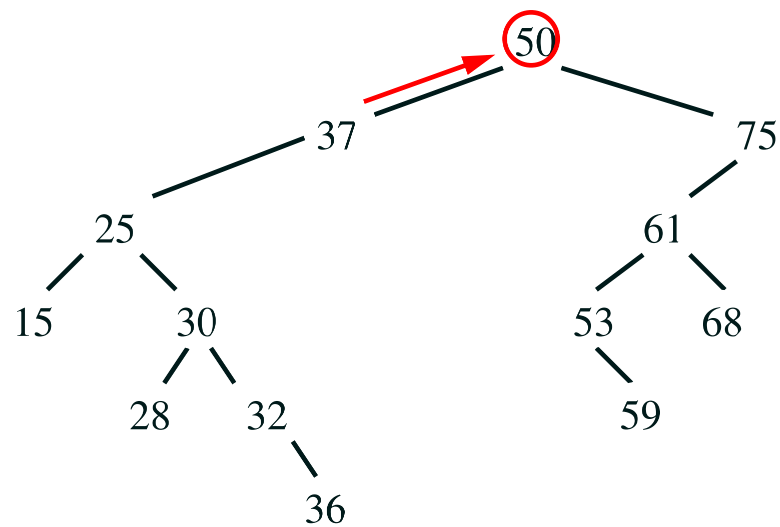



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

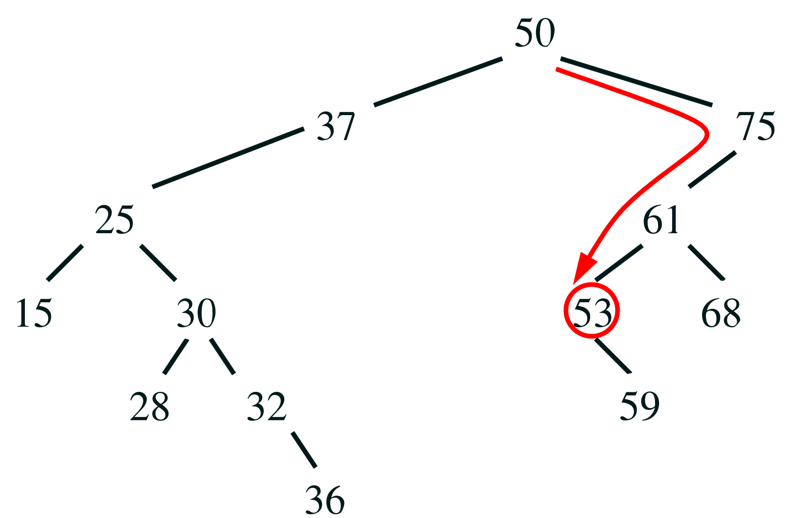



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

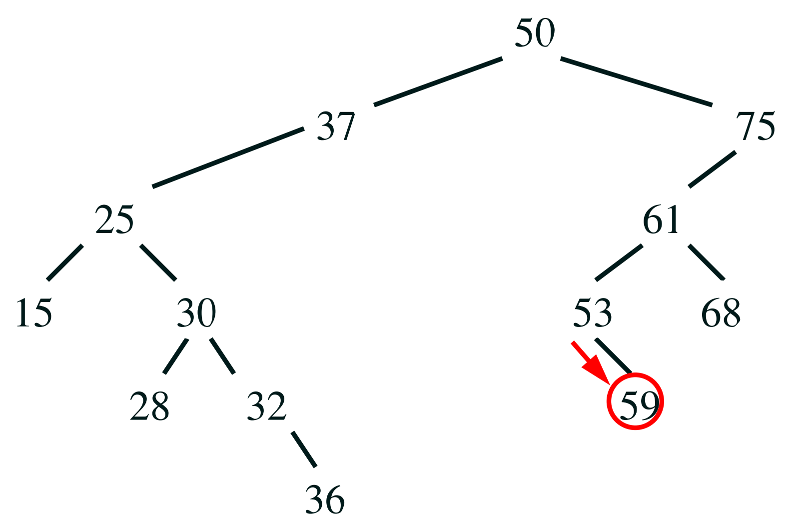



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

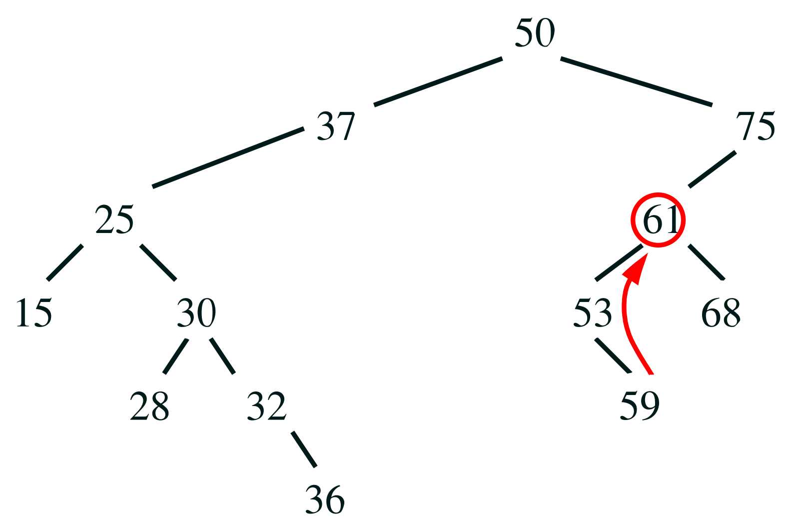



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ \color{red}{59}\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

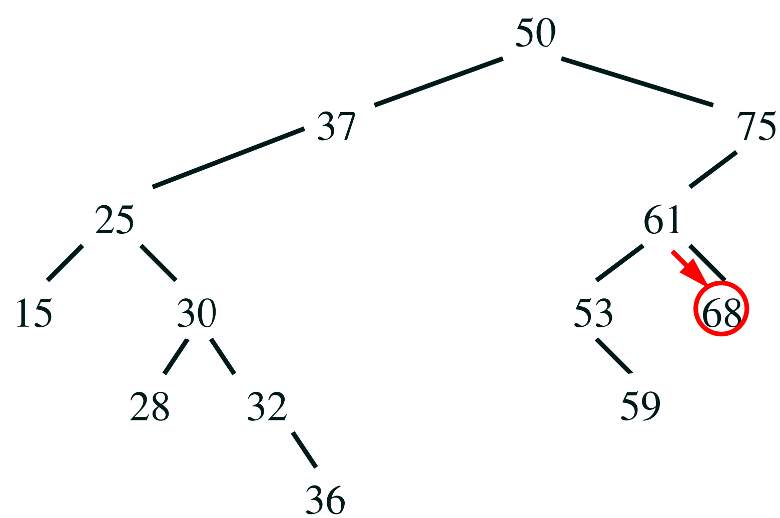



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

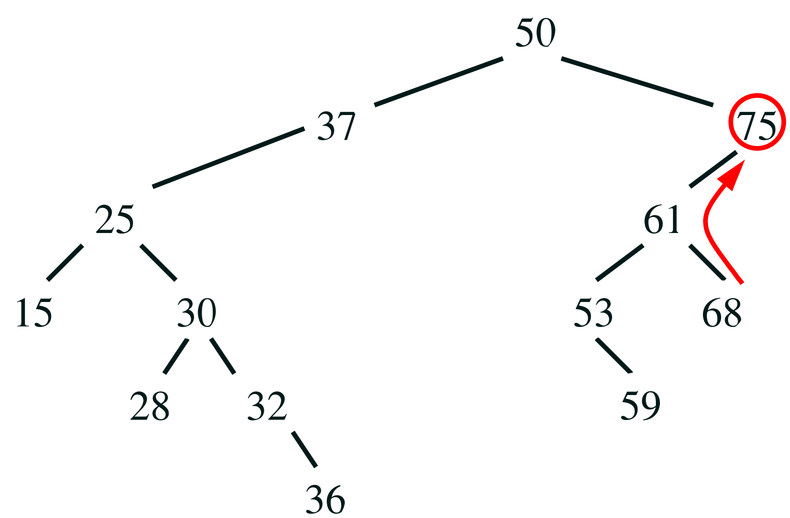



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

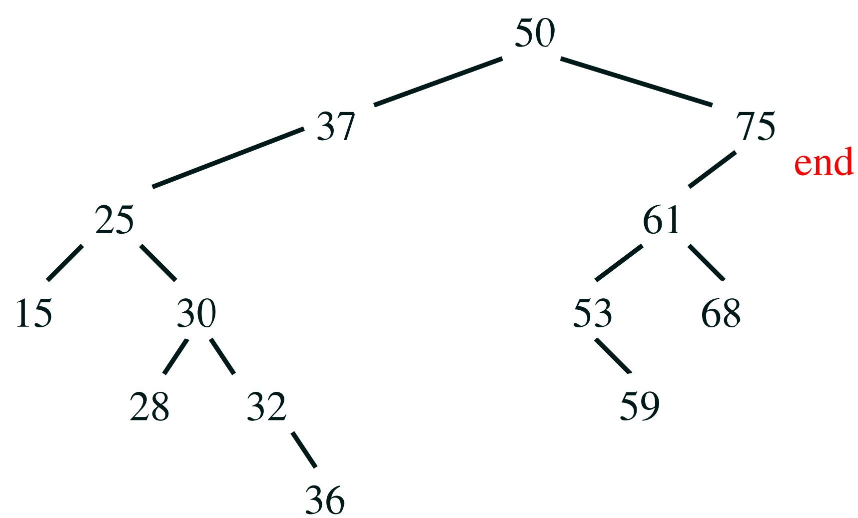



$$\{15\ 25\ 28\ 30\ 32\ 36\ 37\ 50\ 53\ 59\ 61\ 68\ 75\}$$

# Successor

- To find the successor we first start in the left most branch
- 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

# 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

# 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

# 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

# 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