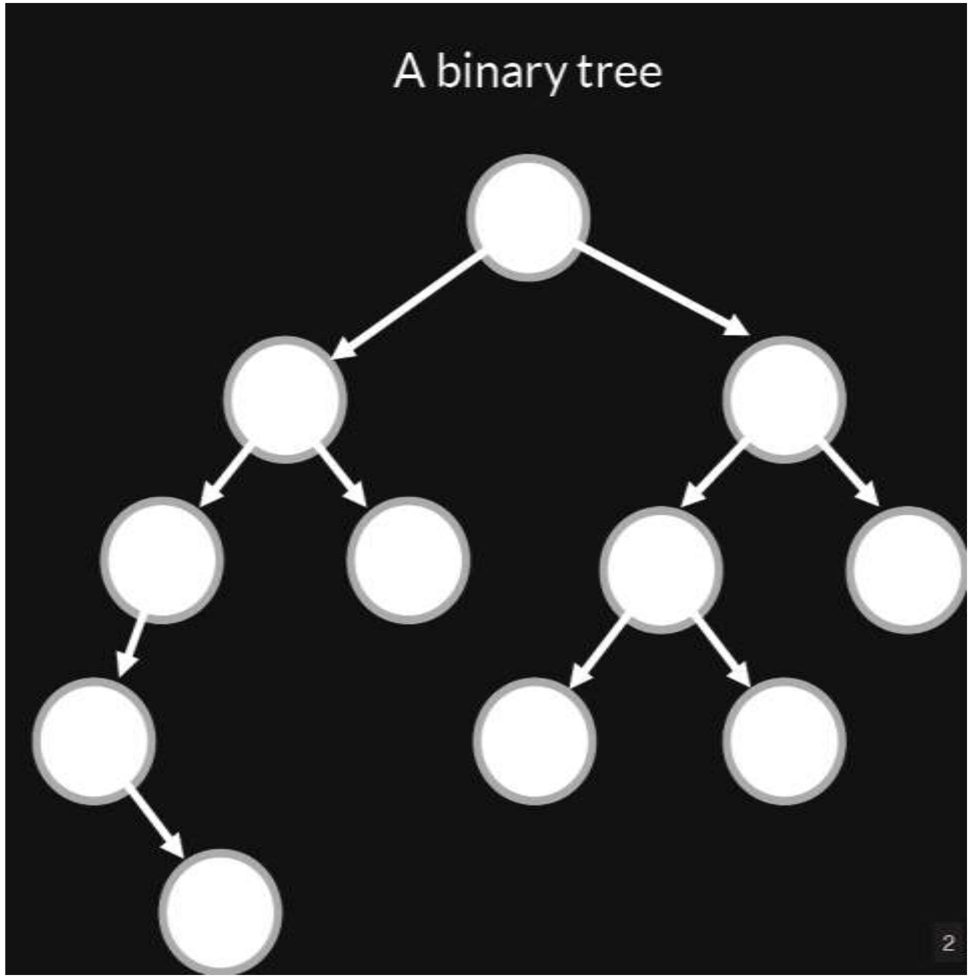
# Lecture 16: Binary Search Trees

Kaleel Mahmood
Department of Computer Science and Engineering
University of Connecticut

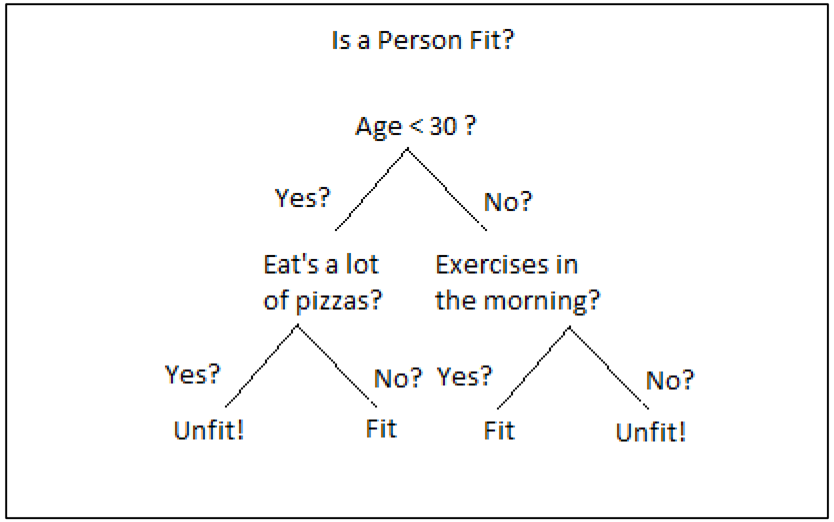
# Trees in Scheme




Pictured: A binary tree in Scheme

- The basic pair structure we have introduced is extremely flexible.
- A natural extension: trees.
- A tree is a natural hierarchical data structure.
- We'll focus on a variant called binary trees.

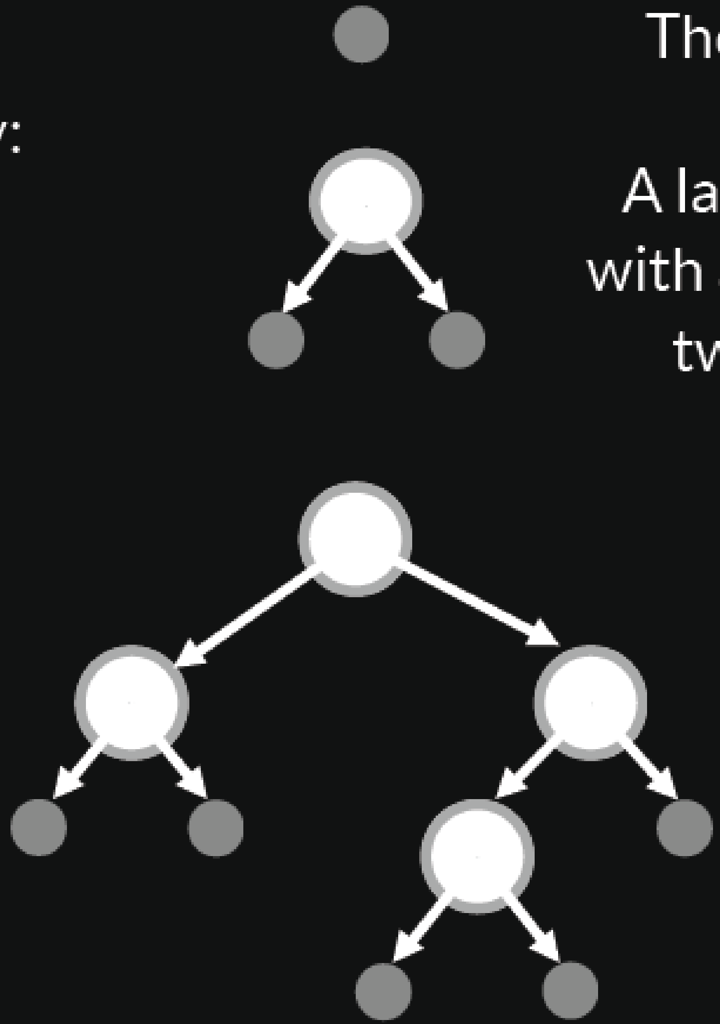
# Motivation: Why do we care about the tree data structure?




- Can be used to store data and do a faster look up than searching through an entire array (we may explain this more later).
- Modeling any kind of hierarchal data- e.g. trees data structures are used in many bioinformatics algorithms.
- Machine learning – A decision tree is one artificial intelligence algorithm that can be used to make classifications.

# TREES, A DEFINITION

- The notion of tree can be defined recursively:
- A tree is either:
  - The *empty* tree.
  - A *node, with arrows pointing to two trees.*



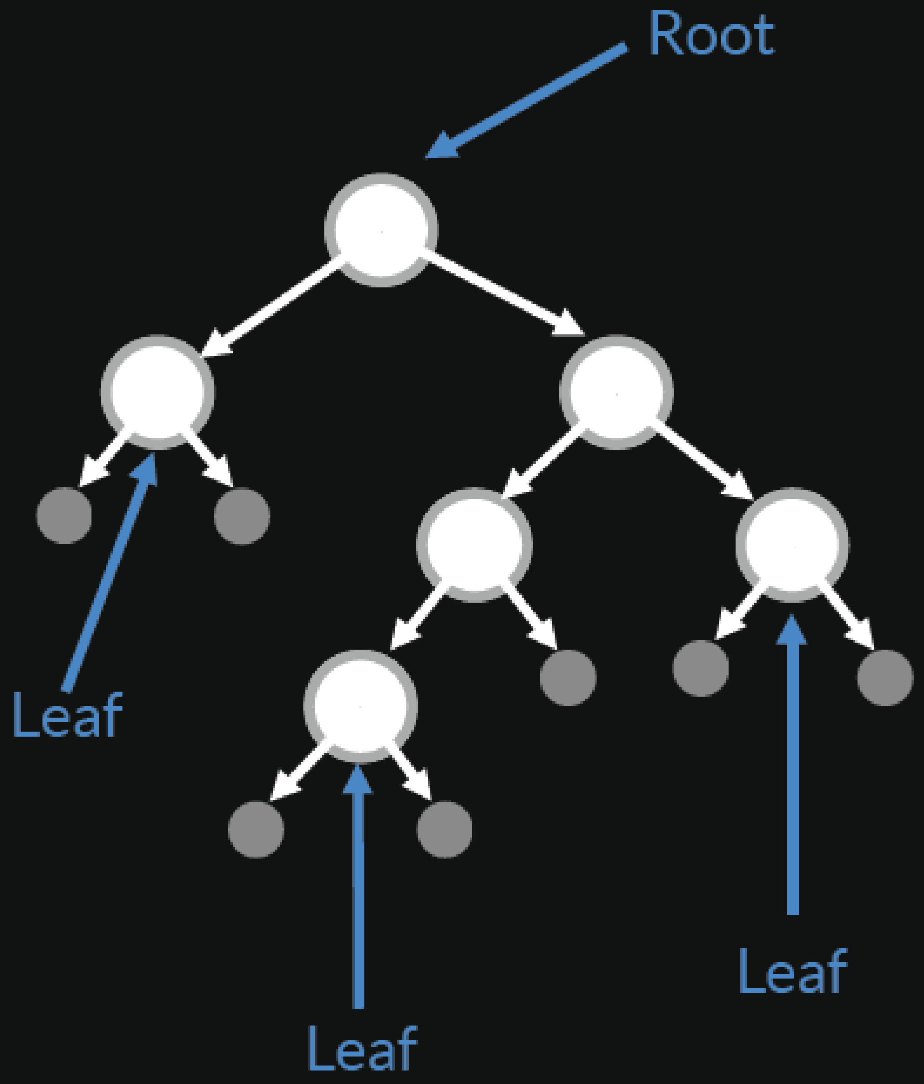
The empty tree

A larger tree: A node with arrows pointing to two empty trees.

A more complicated tree with four nodes

# TERMINOLOGY

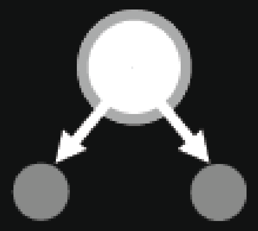
- The top of the tree is the root.
- The children of a node are the roots of the trees to which it points.
- A leaf is a node with no children (so it points to two empty trees).



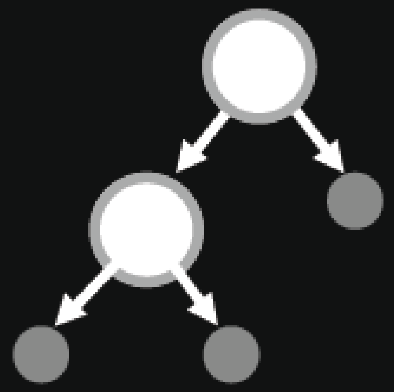
# DEPTH

- The depth of a tree is length of the longest path from the root to a leaf.
- We do not define the depth of the empty tree.
- A tree with a single node has depth 0.
- Otherwise, notice that the depth of a tree is one more than the maximum depth of the trees rooted at its children.

Depth undefined



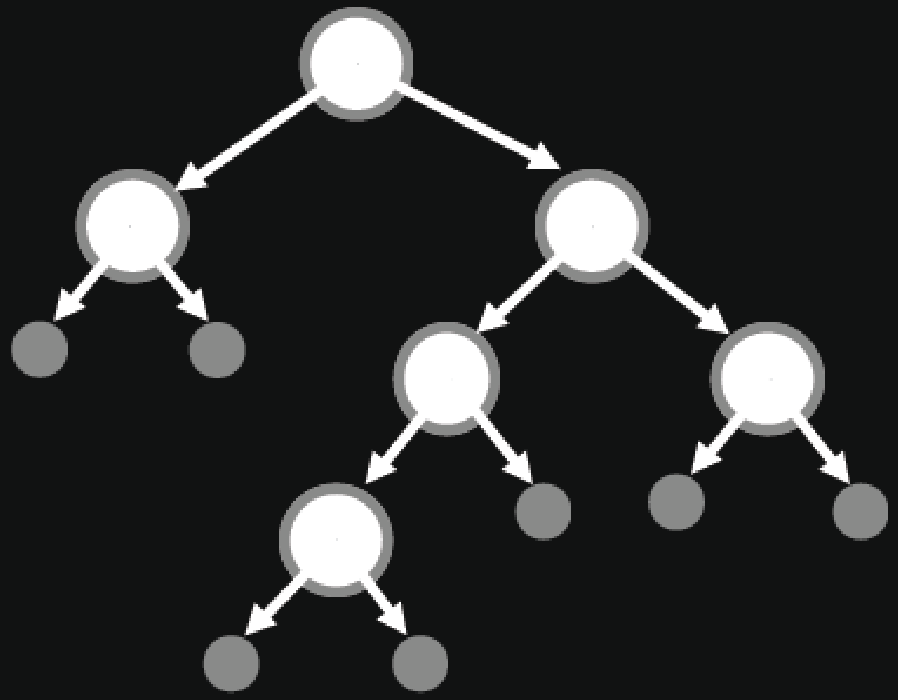
Depth 0



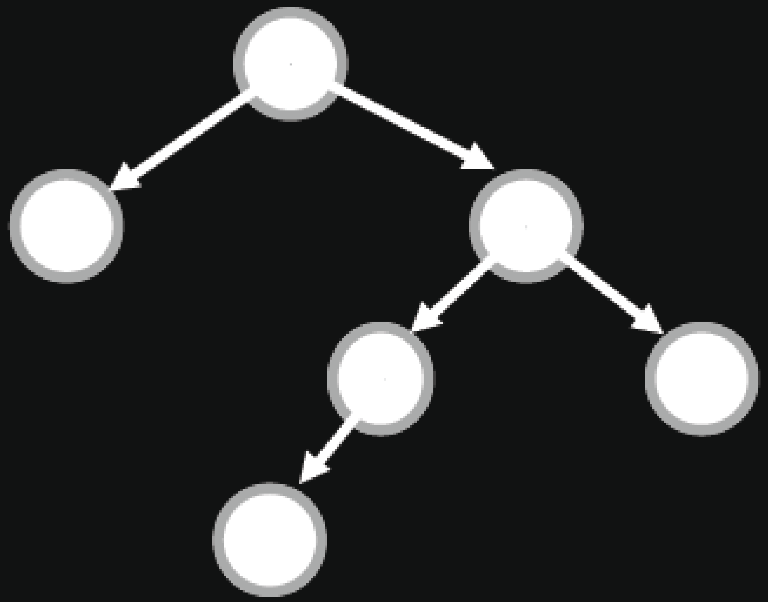
Depth 1

# CONVENTIONS

- When we draw trees, we typically suppress the empty trees.
- Thus:



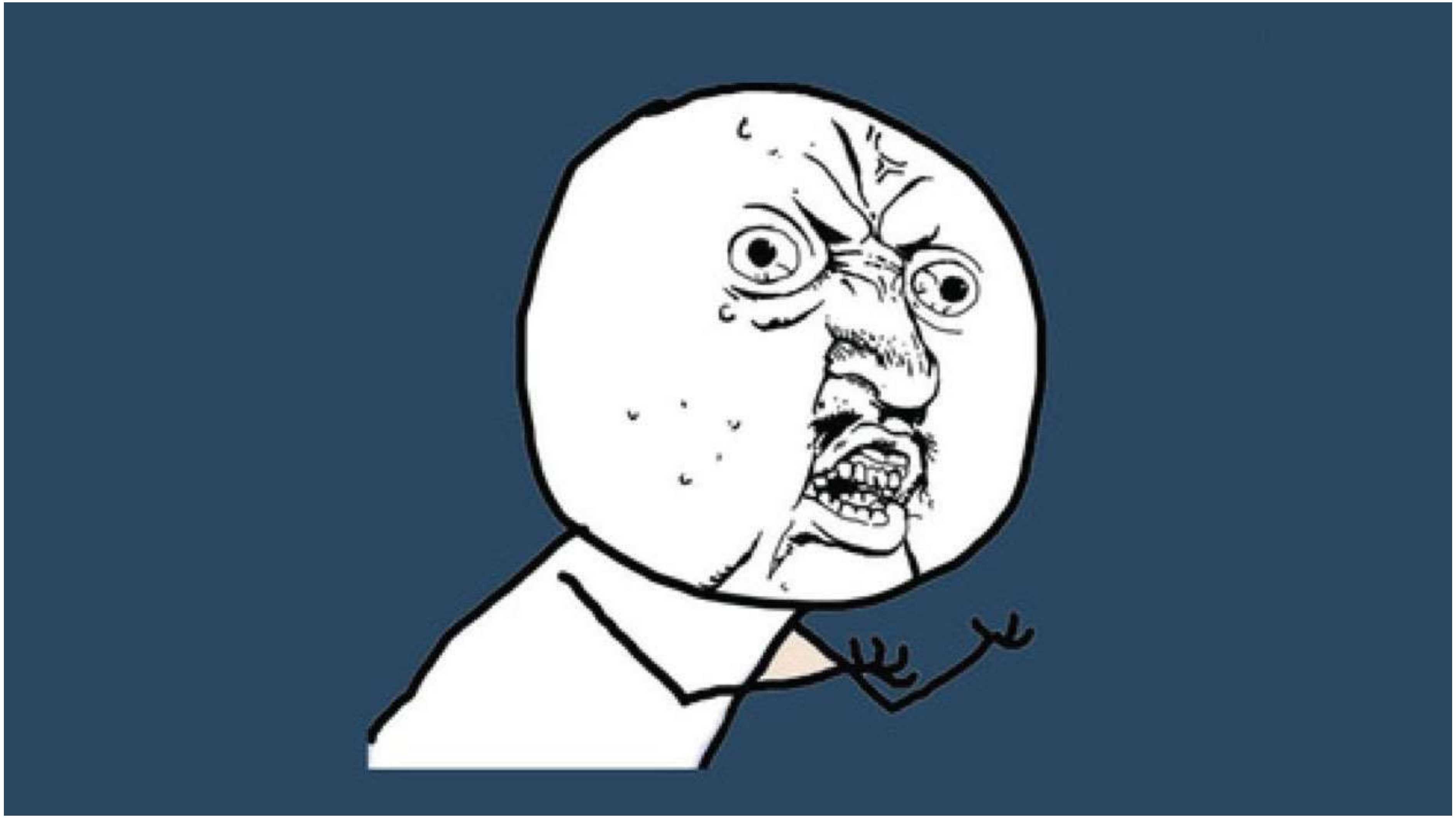
is drawn



# MOTIVATING TREES: MAINTAINING A SET

- Why might you wish to store data this way?
- Consider the basic task of maintaining a set of numbers. We'd like to be able to
  - **insert** a number into our set, and
  - **test membership**: Determine if a given number is an element of our set.

Need to insert elements



Why you no use List?

# MAINTAINING A SET WITH A LIST

- What's the problem? We have a data structure we can use for this purpose: a list.
  - To insert a number: Easy! Just add it at the beginning of the list unless it is already in the set.
  - To test membership: Easy! Just scan the list from left to right.

```
(define (insert element set)
  (if (ismember? element set)
      set
      (cons element set)))

(define (ismember? element set)
  (cond ((null? set) #f)
        ((equal? element (car set)) #t)
        (else (ismember? element (cdr set)))))
```

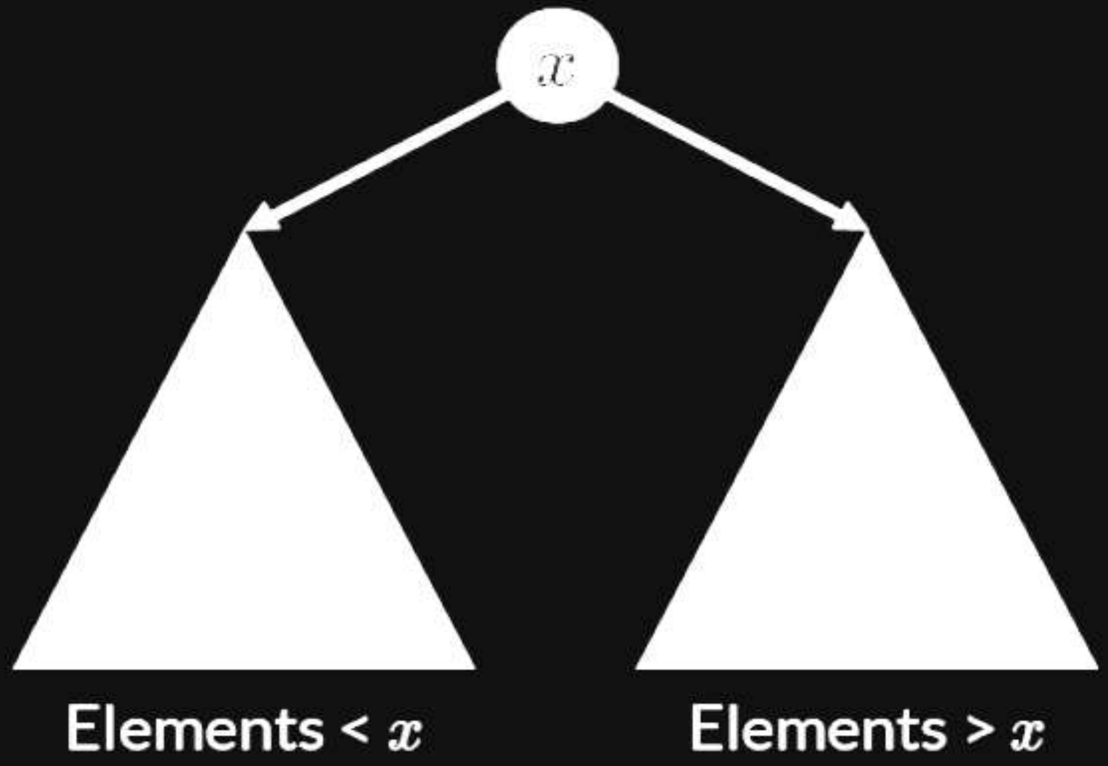
Note… this returns the new set

# WHAT'S THE PROBLEM?

- So...
  - Insertion can be *FAST if not a member*. A single function call to `cons`.
  - But... testing membership is *SLOW*.
    - We may have to scan the entire list each time.
- Unfortunately, in practice, most processing with sets tends to be membership queries.
- We'd like a way to maintain sets so that both insertion and membership are fast.

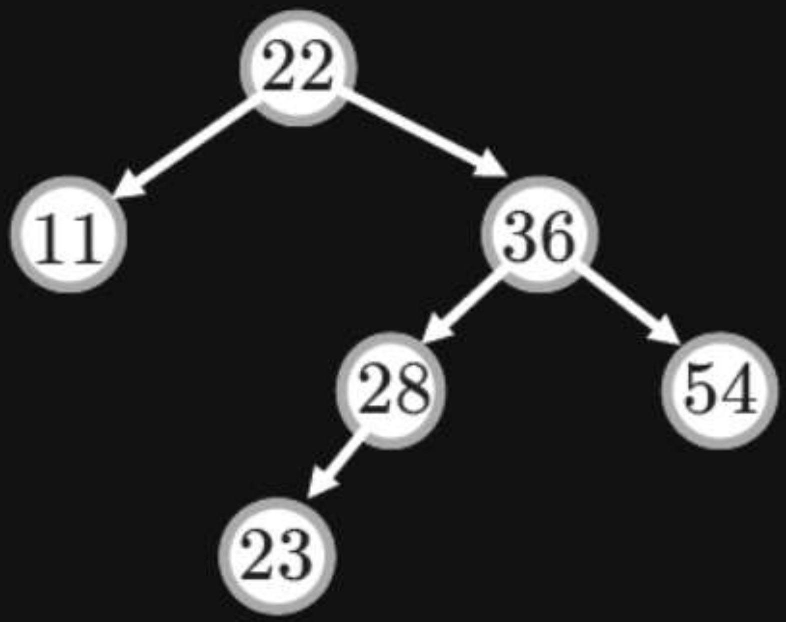
# IDEA: MAINTAIN THE SET IN A BINARY SEARCH TREE

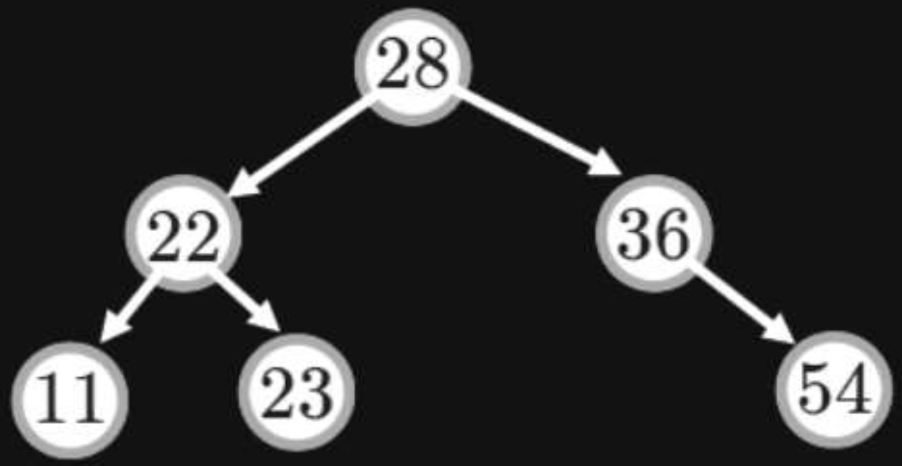
- A binary search tree for the set $S$:
  - Elements of $S$ are placed on the nodes in such a way that...
  - **Rule**:
    - elements in the left subtree of $x$ are smaller than $x$;
    - elements in the right subtree are larger than $x$.



# BINARY SEARCH TREES

- Note that a given set can have many binary search trees.
- Consider the set $\{11, 22, 23, 28, 36, 54\}$.
  - Two binary search trees are shown below:



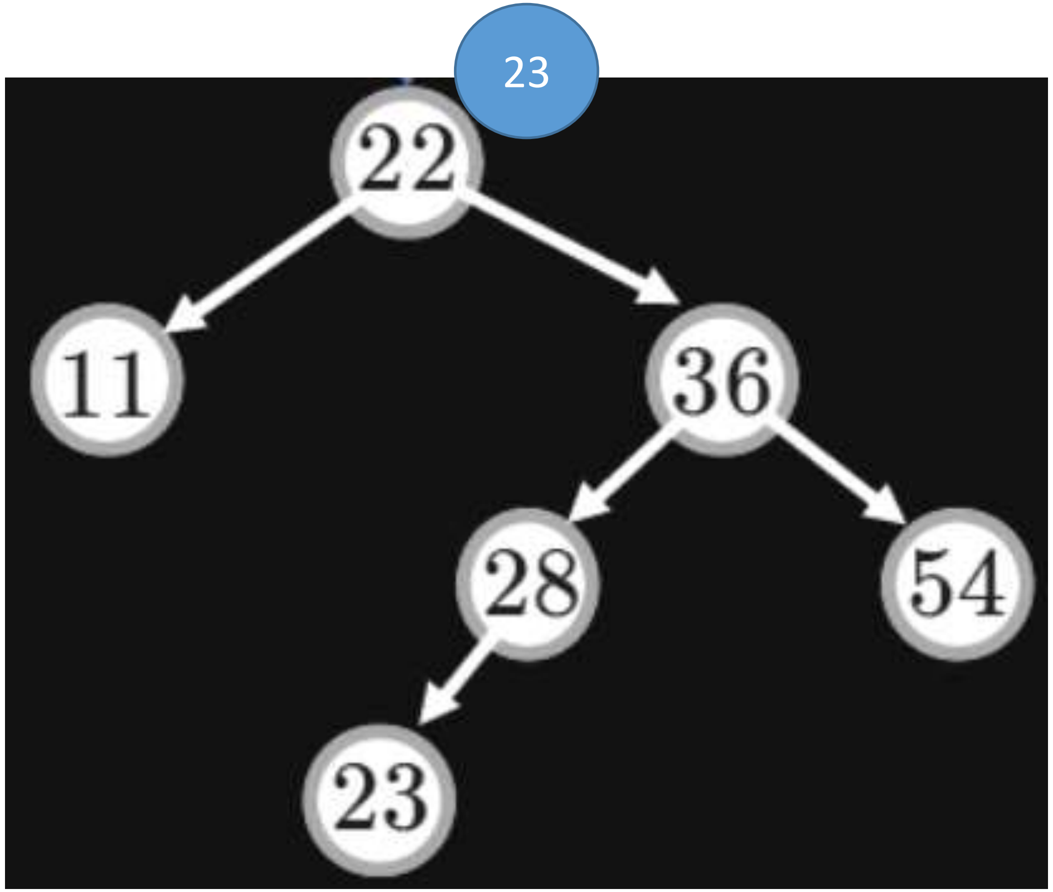
# SEARCHING FOR AN ELEMENT IN A BINARY SEARCH TREE

- Element?(x,T)
  - If T is empty, return #f.
  - If x = root(T), return #t.
  - Otherwise:
    - if x > root(T) return Element?(x, RightChild(T));
    - if x < root(T) return Element?(x, LeftChild(T)).

A more formal Pseudocode of the above algorithm:

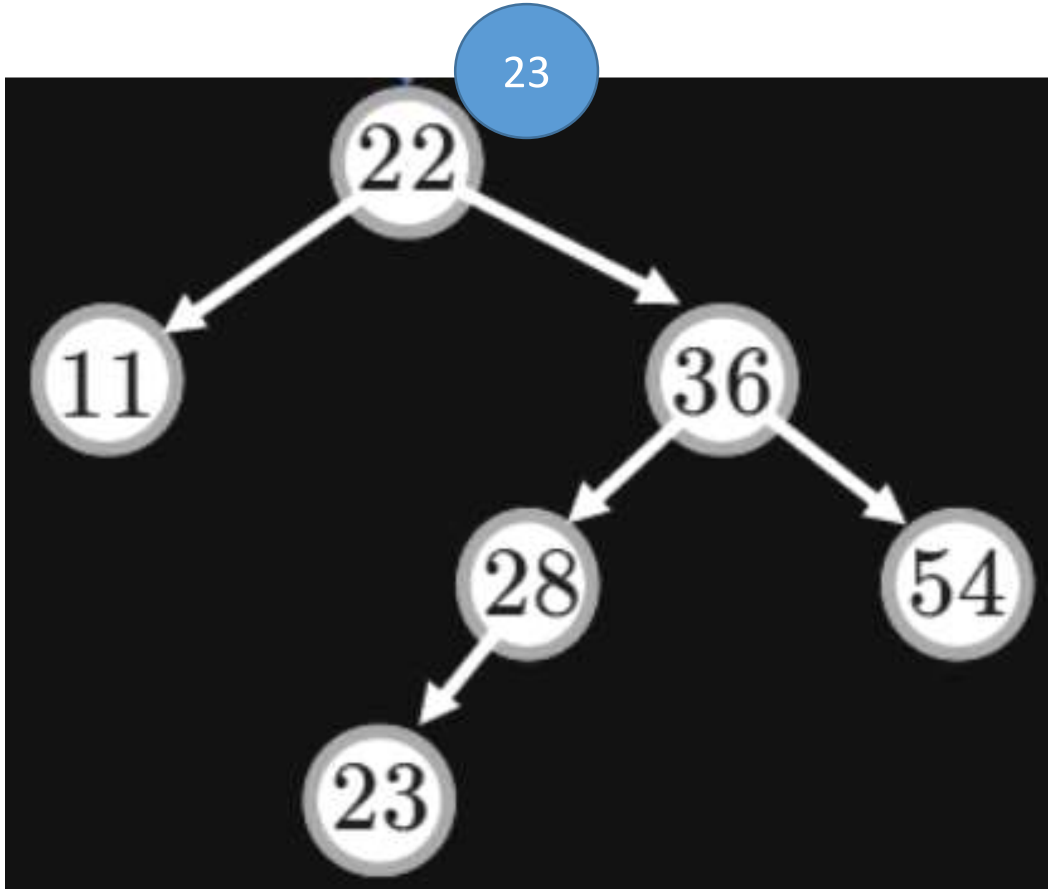
```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree



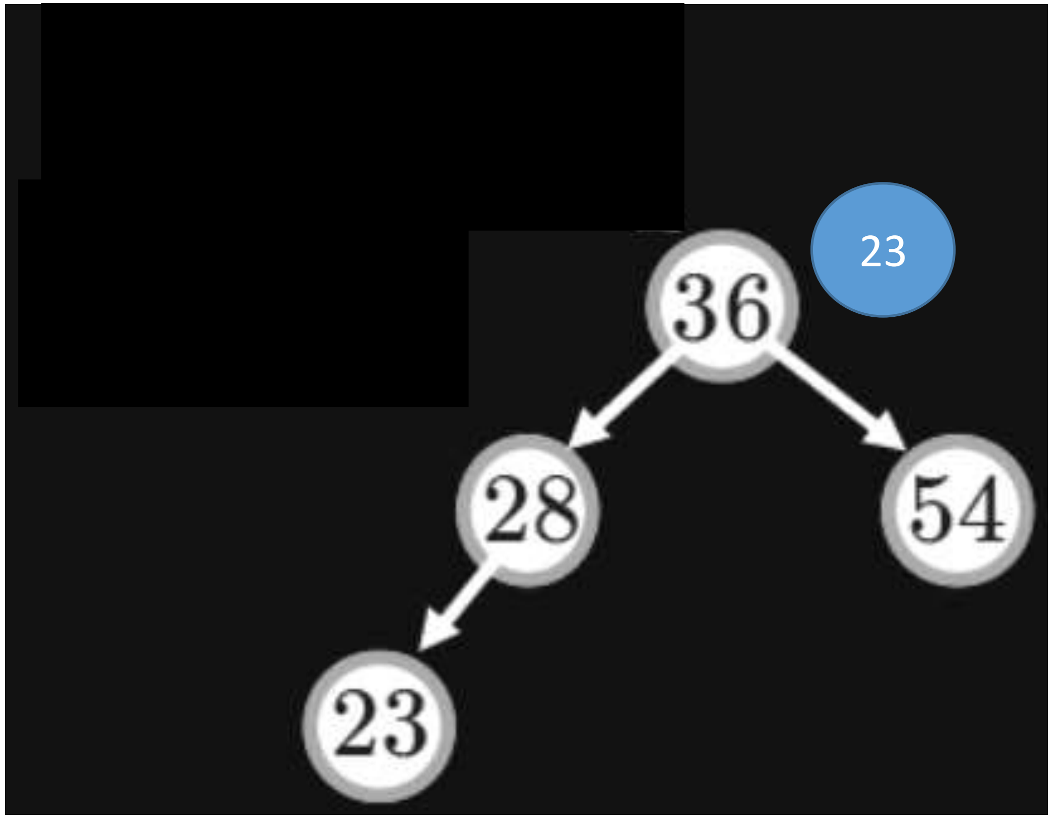

```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree



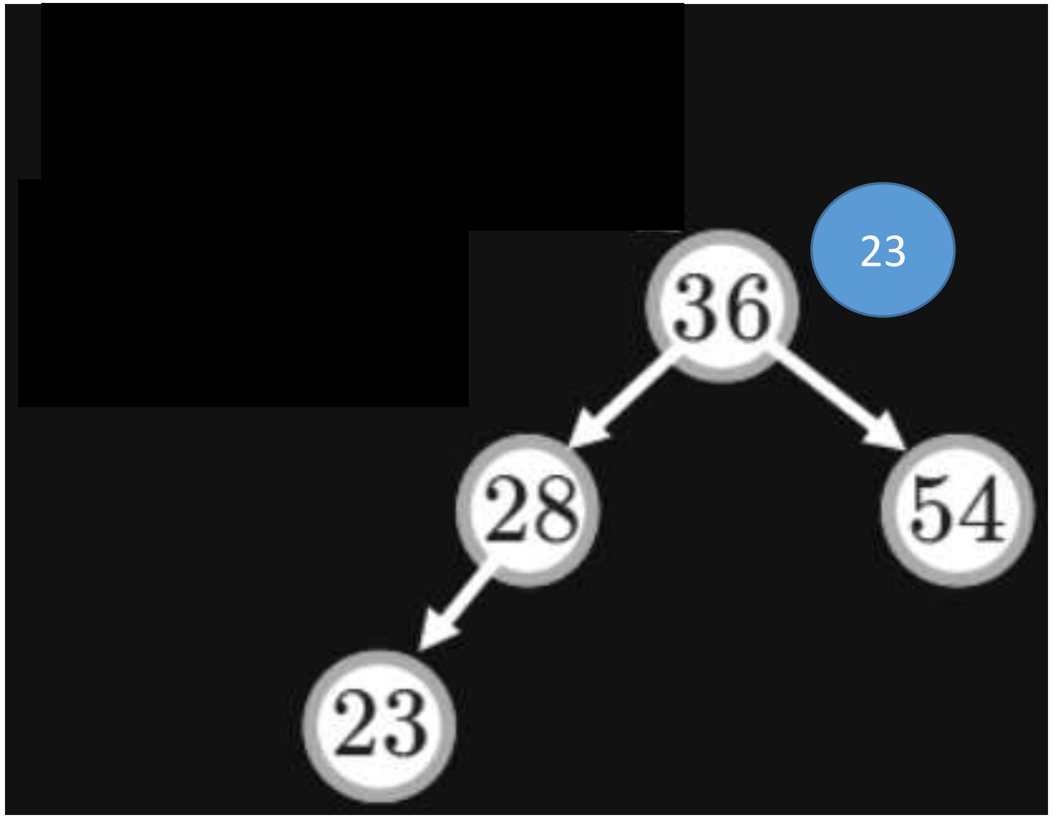

```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree



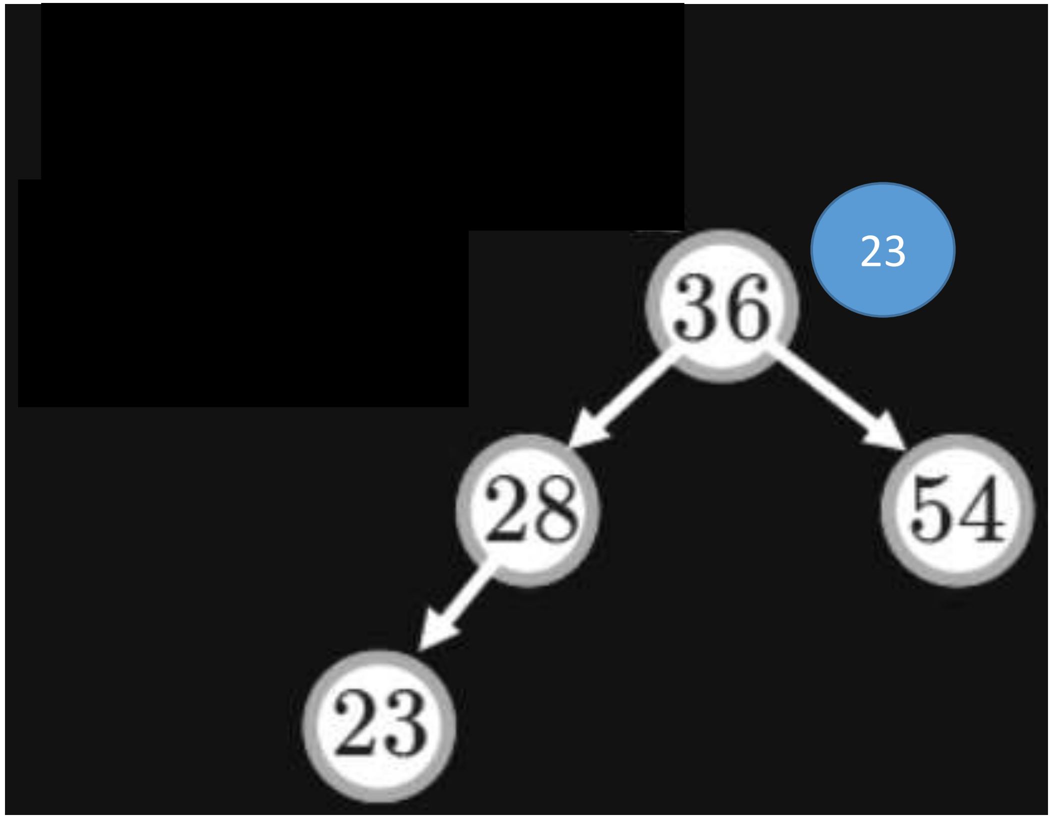

```
Element?(x,T)
    If T is empty,
        return #f.
    If x = root(T),
        return #t.
    Otherwise:
        if x > root(T)
            return Element?(x, RightChild(T));
        if x < root(T)
            return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree



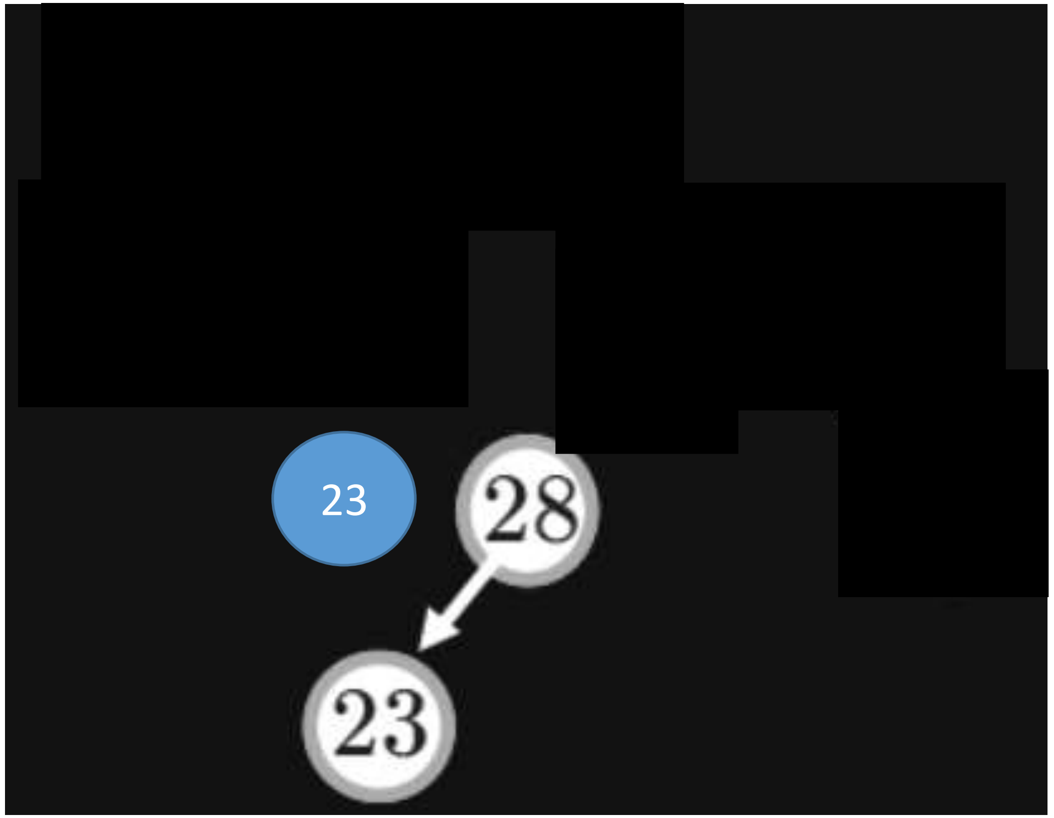

```
Element?(x,T)
    If T is empty,
        return #f.
    If x = root(T),
        return #t.
    Otherwise:
        if x > root(T)
            return Element?(x, RightChild(T));
        if x < root(T)
            return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree



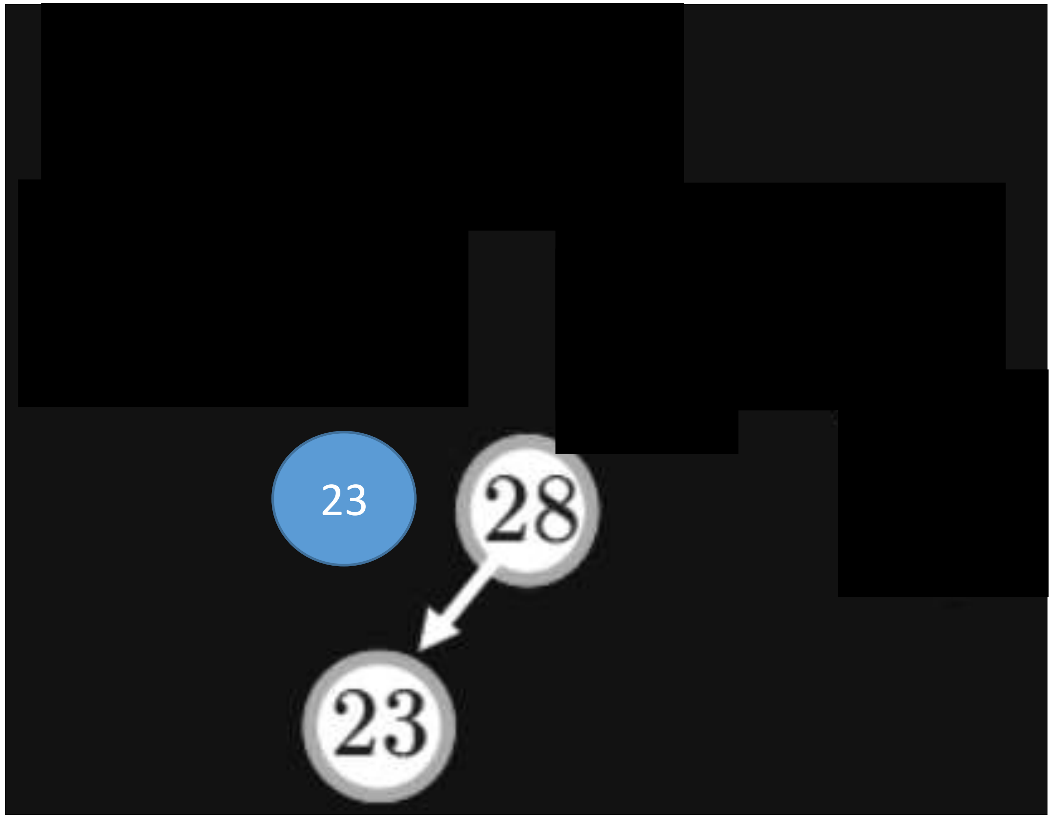

```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree



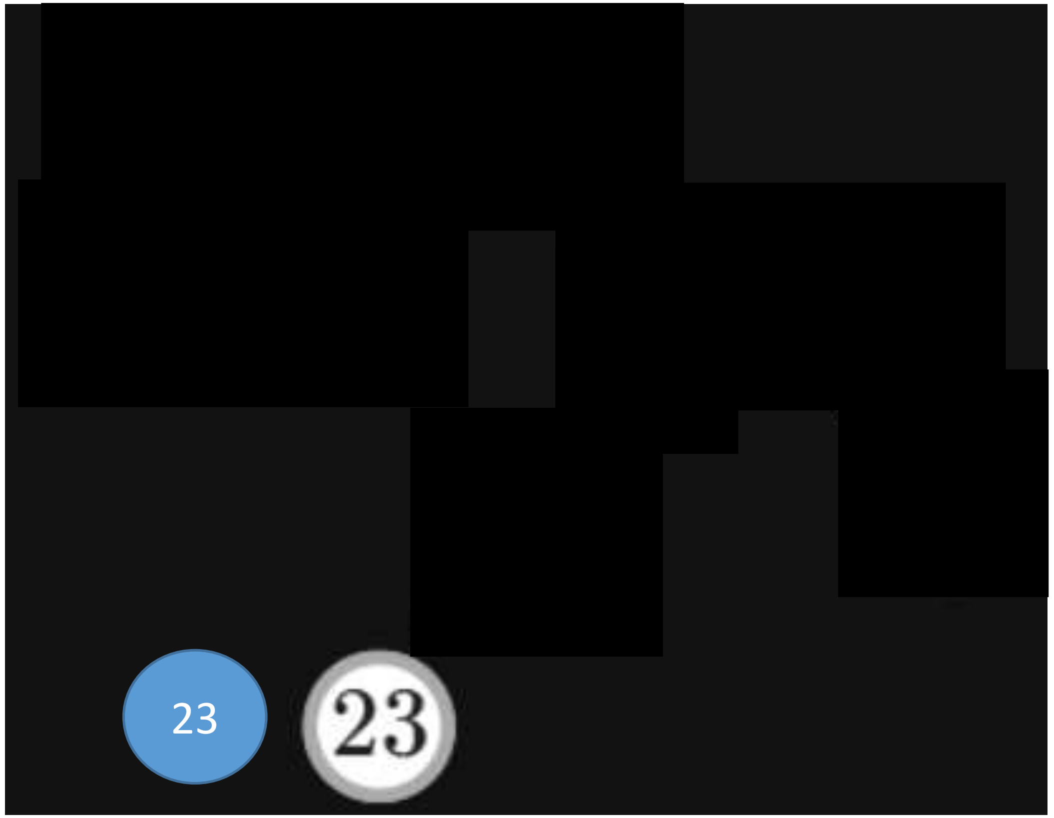

```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree



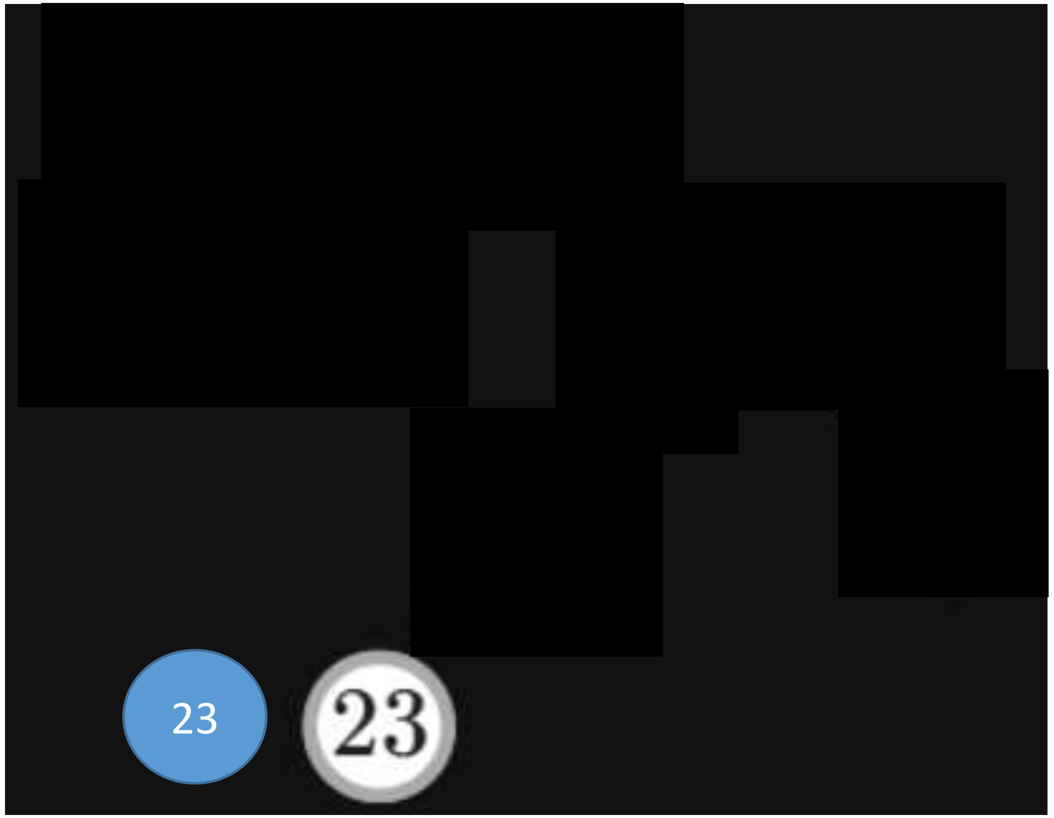

```
Element?(x,T)
    If T is empty,
        return #f.
    If x = root(T),
        return #t.
    Otherwise:
        if x > root(T)
            return Element?(x, RightChild(T));
        if x < root(T)
            return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree




```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

# An Example: Check if 23 is in the tree




Finally we found x is equal to a root so we can return true!

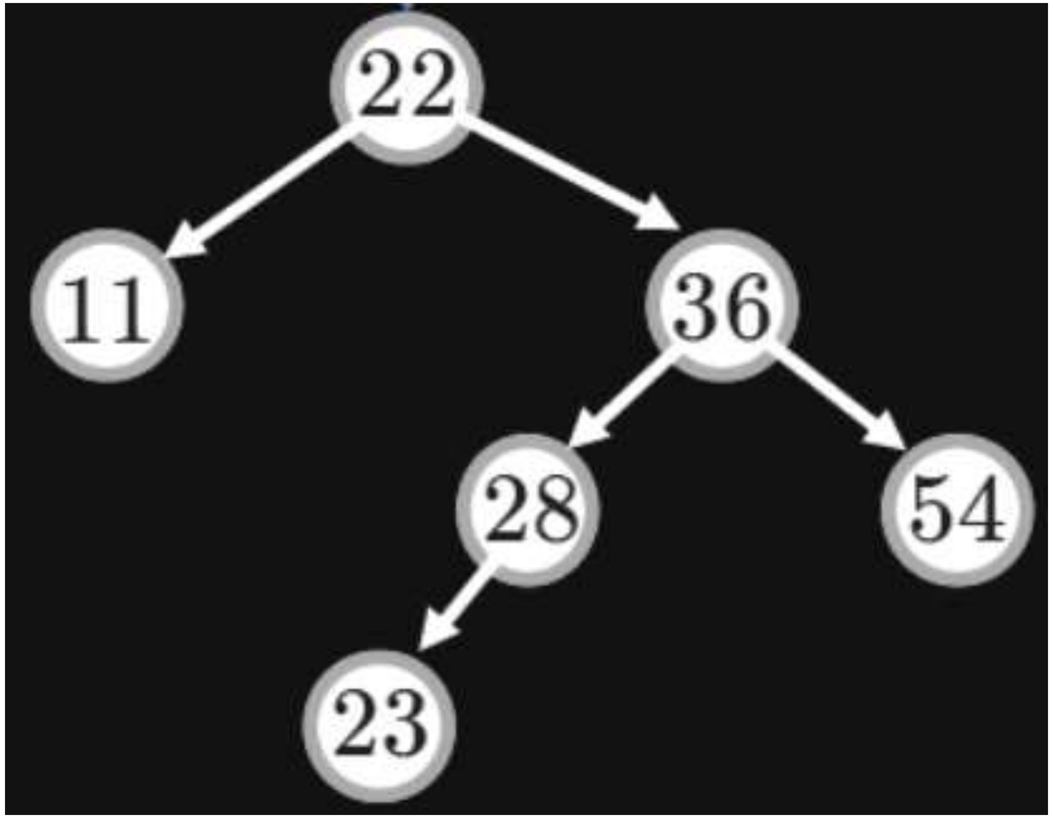
```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

# Why is inserting in a tree faster than inserting in a list?

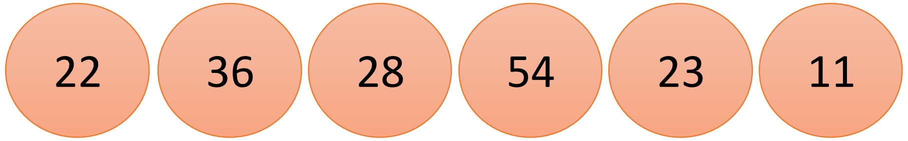
- <u>Its not magic.</u>
- For a list we have to go through every element and make a comparison.
- This means for a list, every comparison operation we make reduce the amount of elements we have left to compare by ONE.
- When using a tree every comparison operation reduces the amount of elements we have left to compare by AT LEAST one, but often more.

Head to head comparison: Find out if element 10 is in the data structure?

## Binary Tree



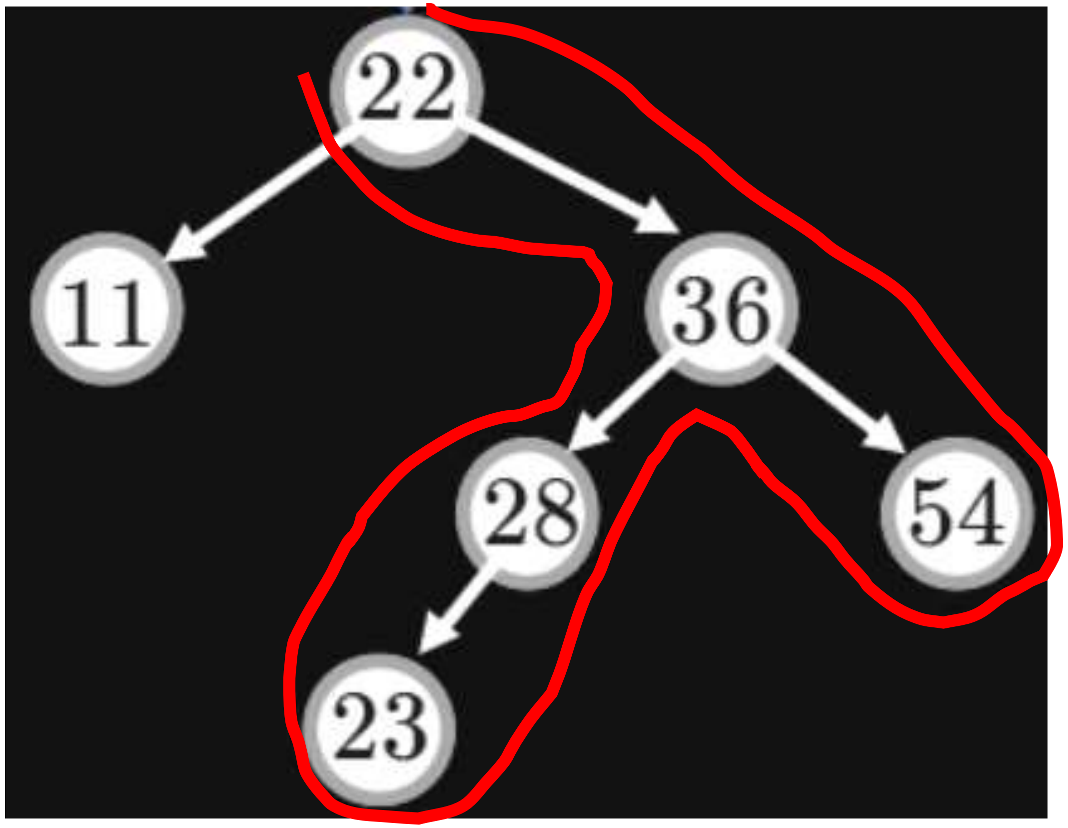

## List



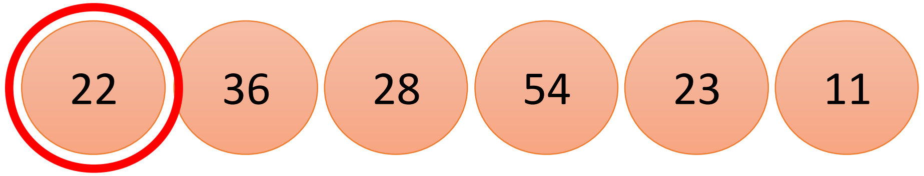
Head to head comparison: Find out if element 10 is in the data structure?

*Comparison 1*

Binary Tree: 5 Elements Eliminated




List: 1 Element Eliminated

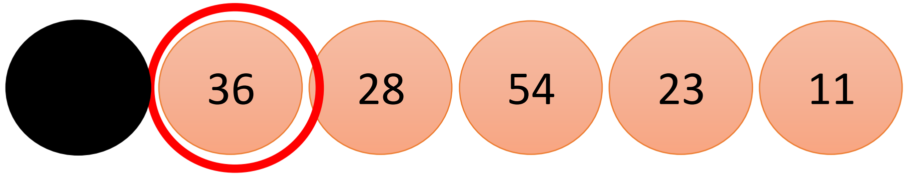


Head to head comparison: Find out if element 10 is in the data structure?

# *Comparison 2*

## Binary Tree: 1 Element Eliminated and done!


## List: 1 Element Eliminated



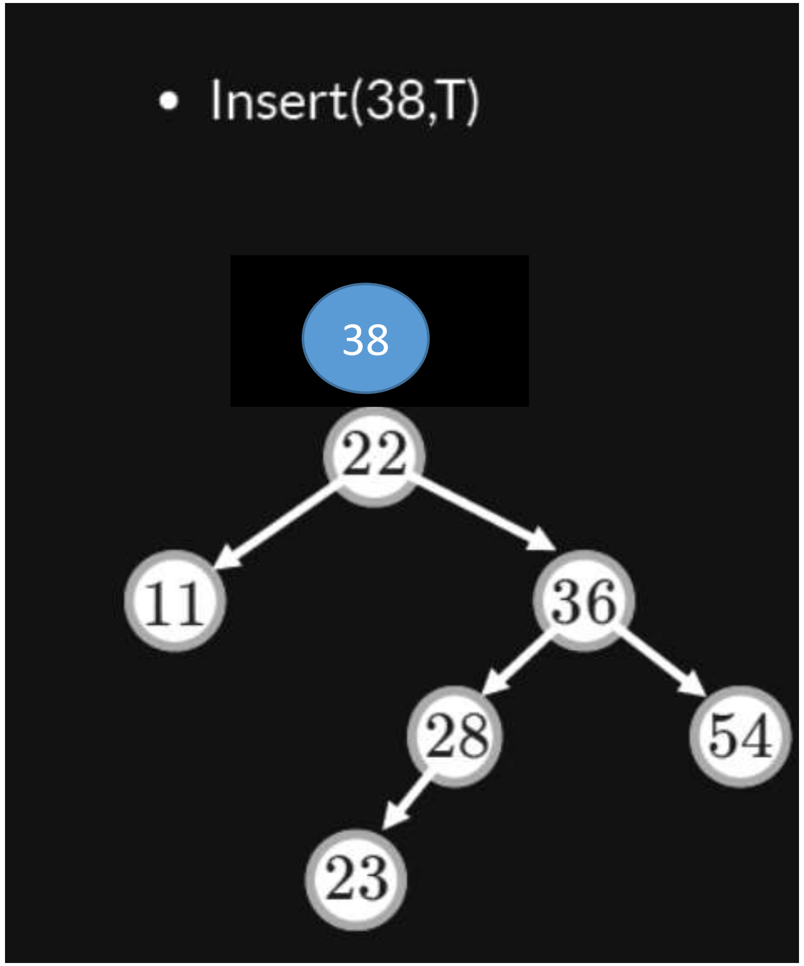

And 5 more comparisons to go…

# INSERTING AN ELEMENT INTO A BINARY SEARCH TREE

- `Insert(x, T)`. To insert an element $x$ into a tree $T$:
  - If $T$ is empty, return a new node containing $x$.
  - Otherwise,
    - if $x$ < root($T$), insert into the left subtree of $T$
    - if $x$ > root($T$), insert into the right subtree of $T$.
  - Return the resulting $T$.
- Note: The tree $T$ is "traversed" in the same way by both Insertion and Element?.

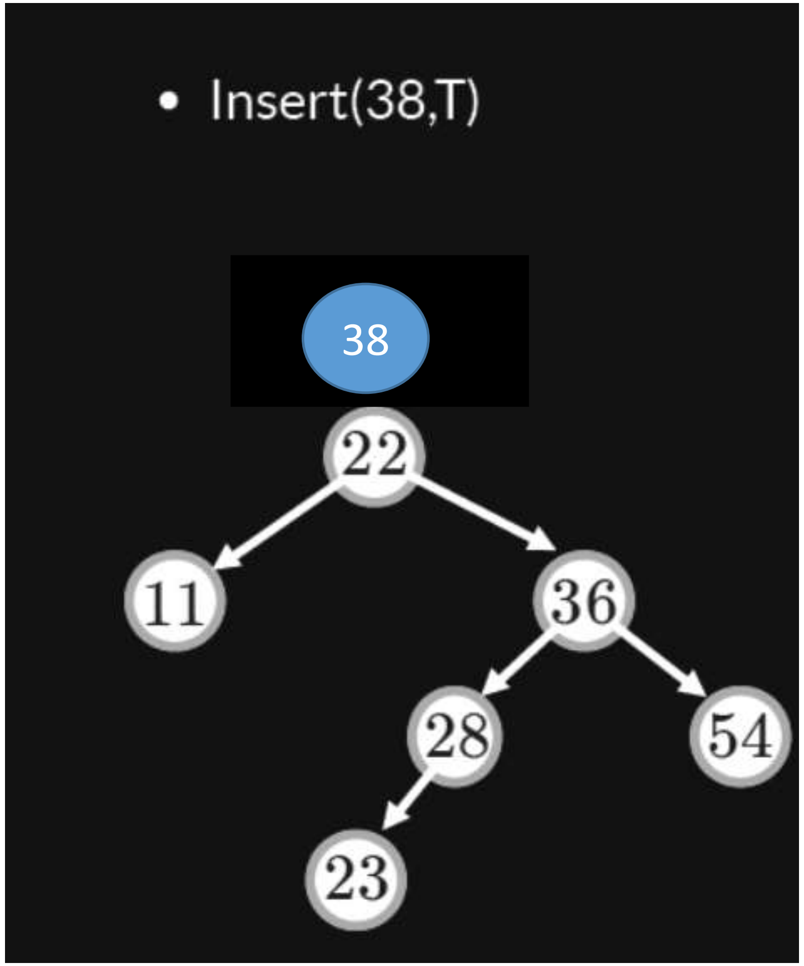
```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# Example: Insert 38 into the tree



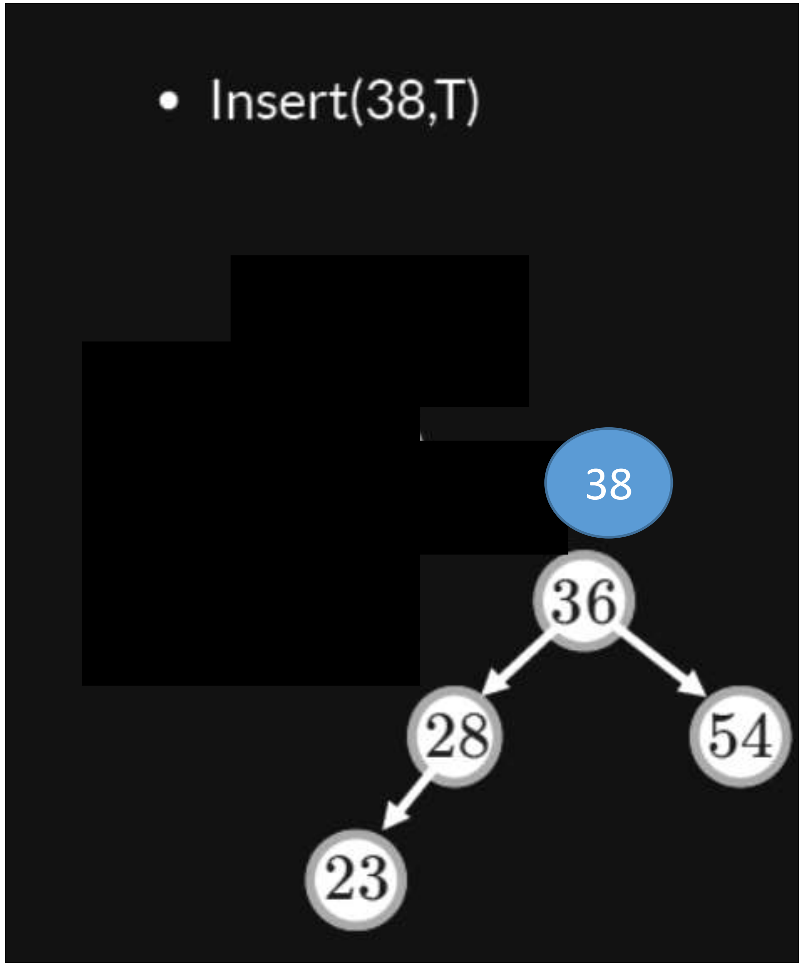

```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# Example: Insert 38 into the tree



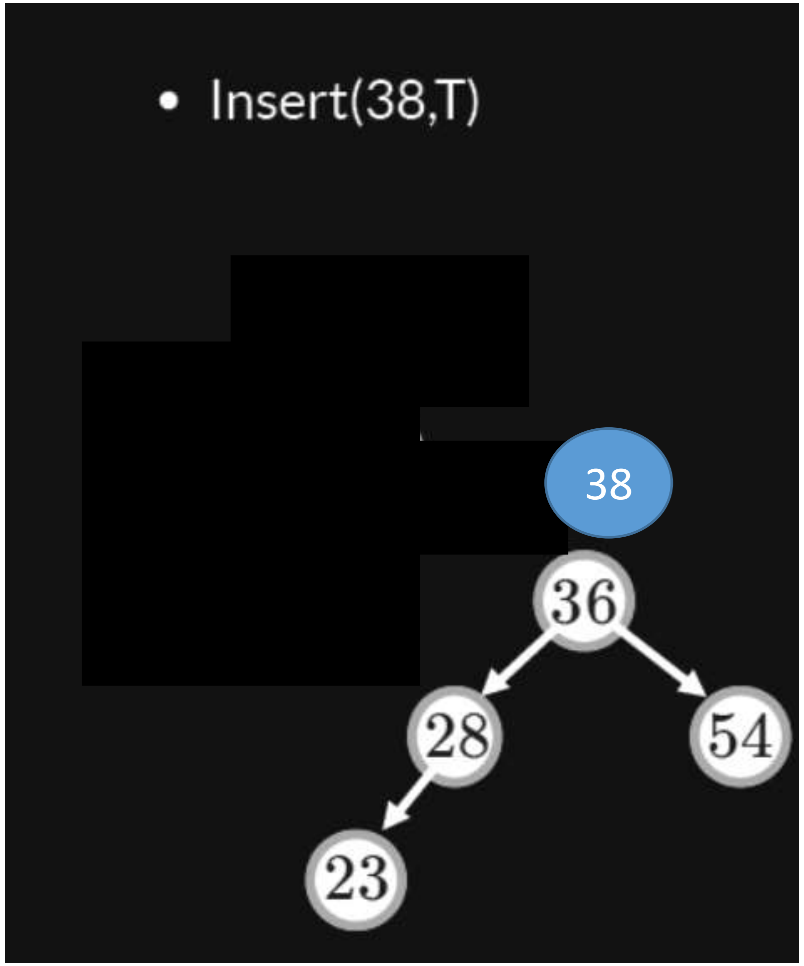

```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# Example: Insert 38 into the tree



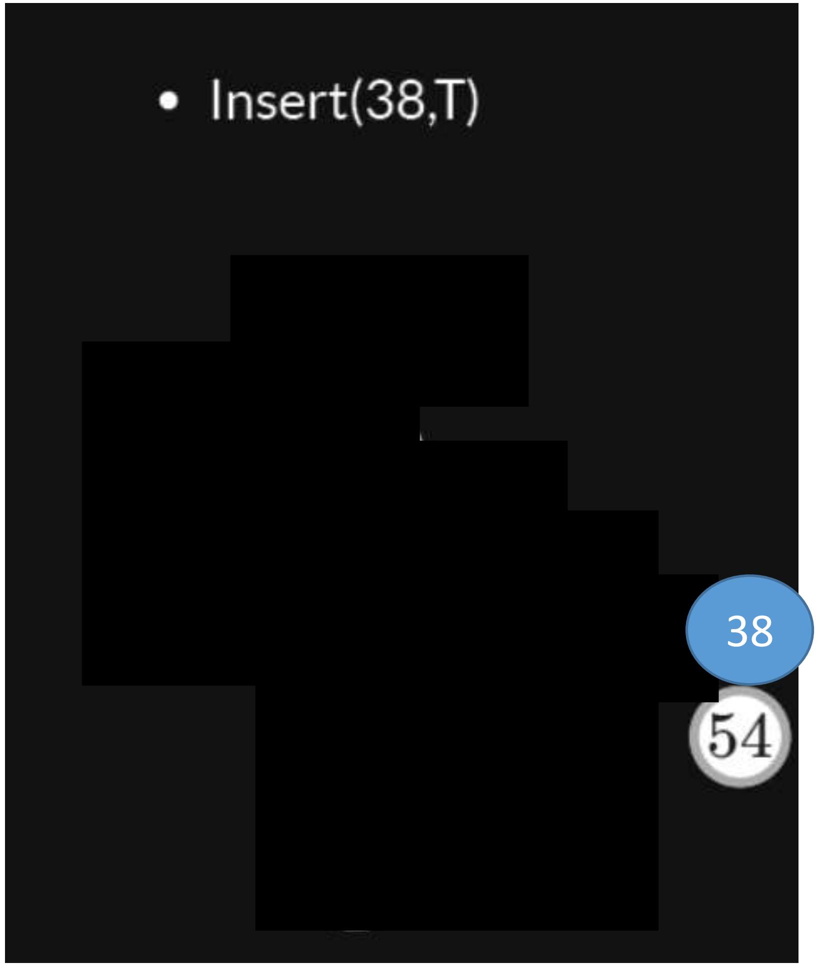

```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# Example: Insert 38 into the tree



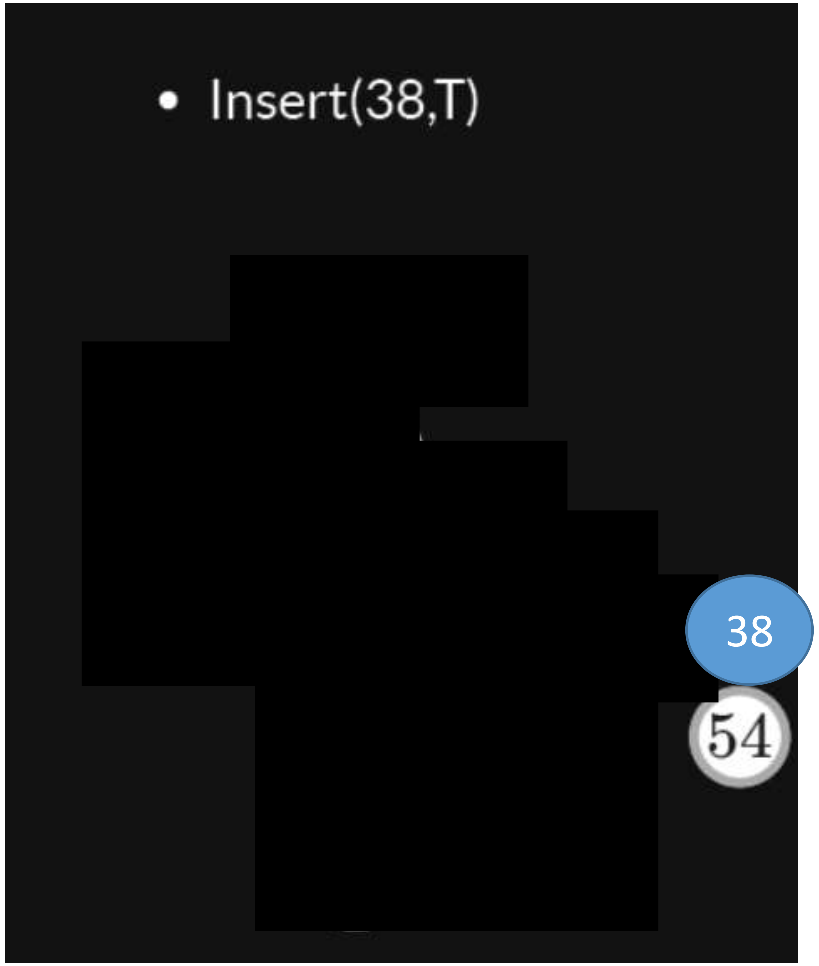

```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# Example: Insert 38 into the tree



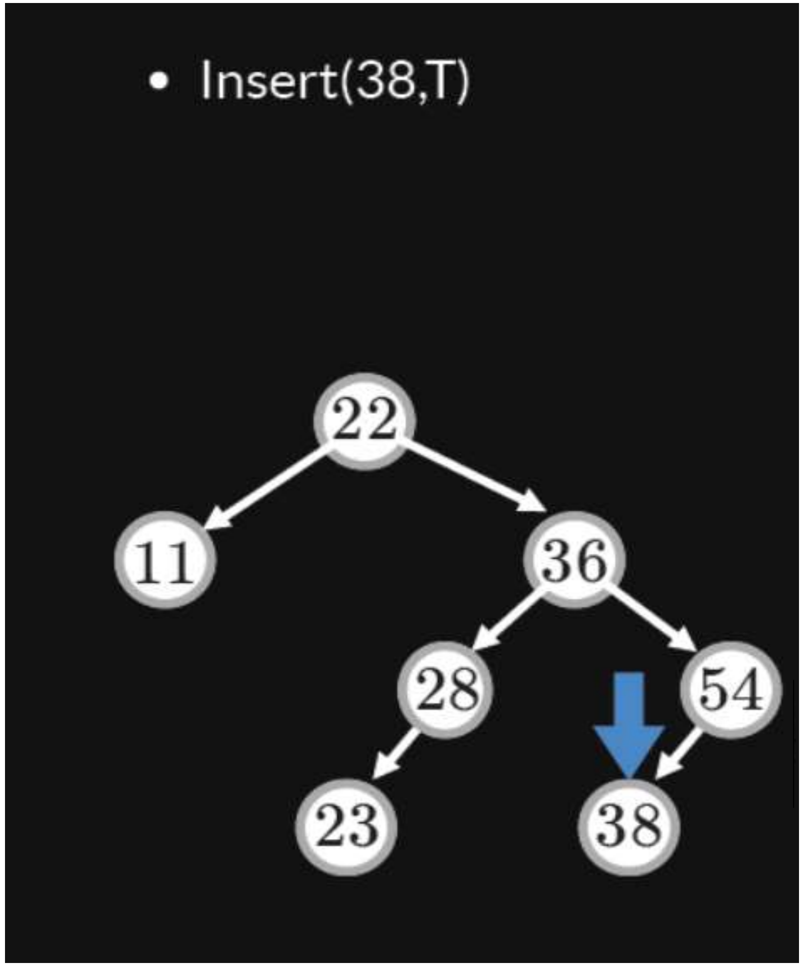

```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# Example: Insert 38 into the tree




```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# Example: Insert 38 into the tree




```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

# HOW EXPENSIVE IS A MEMBERSHIP QUERY?

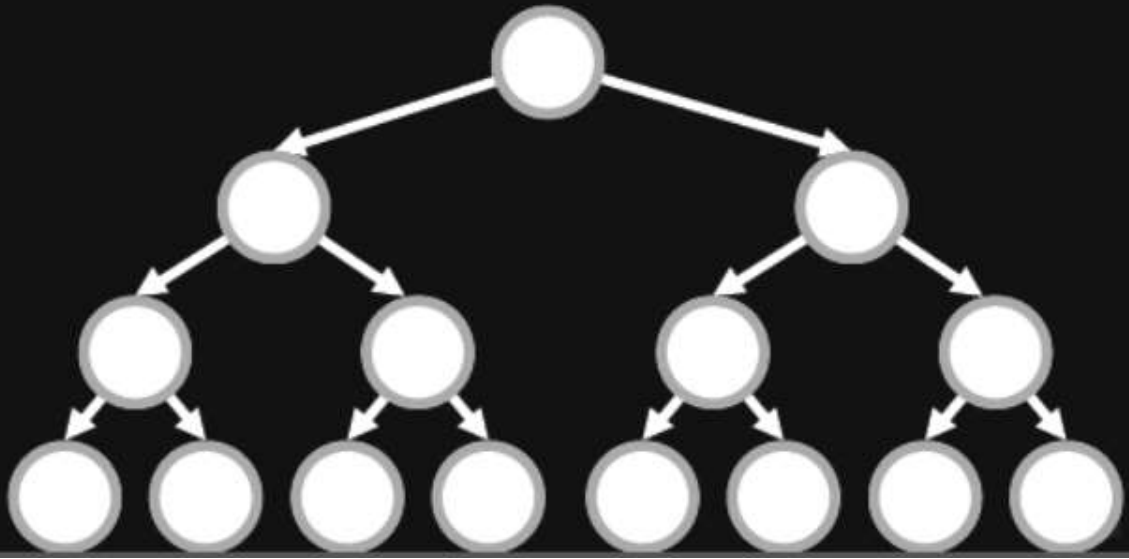
- In general, to determine membership (or insert an element), we may have to traverse the tree from the root to a leaf. In the worst case, this will involve a number of function calls proportional to

The DEPTH of the tree.

- Question: How can we expect depth to scale with the number of elements?

# DEPTH VERSUS SIZE

- **Depth**: longest path in a tree.
- **Size**: total number of nodes in a tree.
- These could be comparable: a long skinny tree.
- Size could be much larger: a bushy tree.

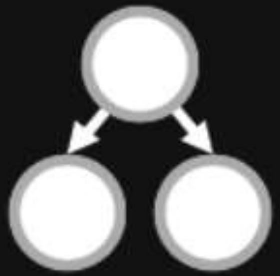


# THE SIZE OF A BUSHY TREE

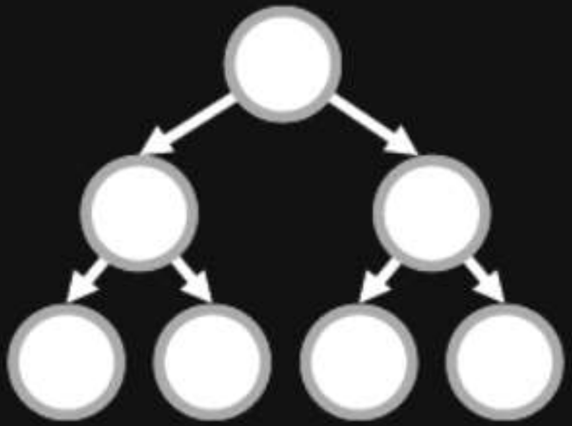
- Consider "complete" trees of various depths. How many elements do they have?

depth: 0
size: 1

depth: 1
size: 3



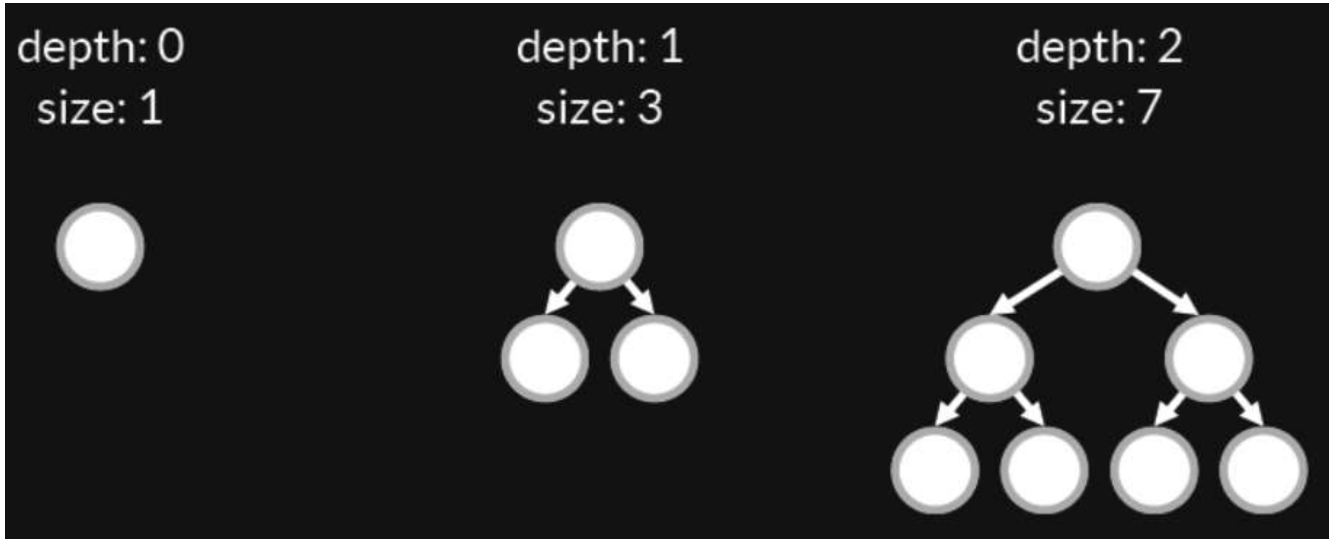
depth: 2
size: 7



# IN GENERAL...

- Let $S_d$ be the size for depth $d$. Then $S_{d+1} = 2 \cdot S_d + 1$.

So if we look at base case $S_0 = 1$ (when the depth is 0) from here we can verify the relationship:




For example $S_2 = 2S_1 + 1 = 2 * (2S_0 + 1) + 1$

$S_2 = 2 * (2 * 1 + 1) + 1 = 7$

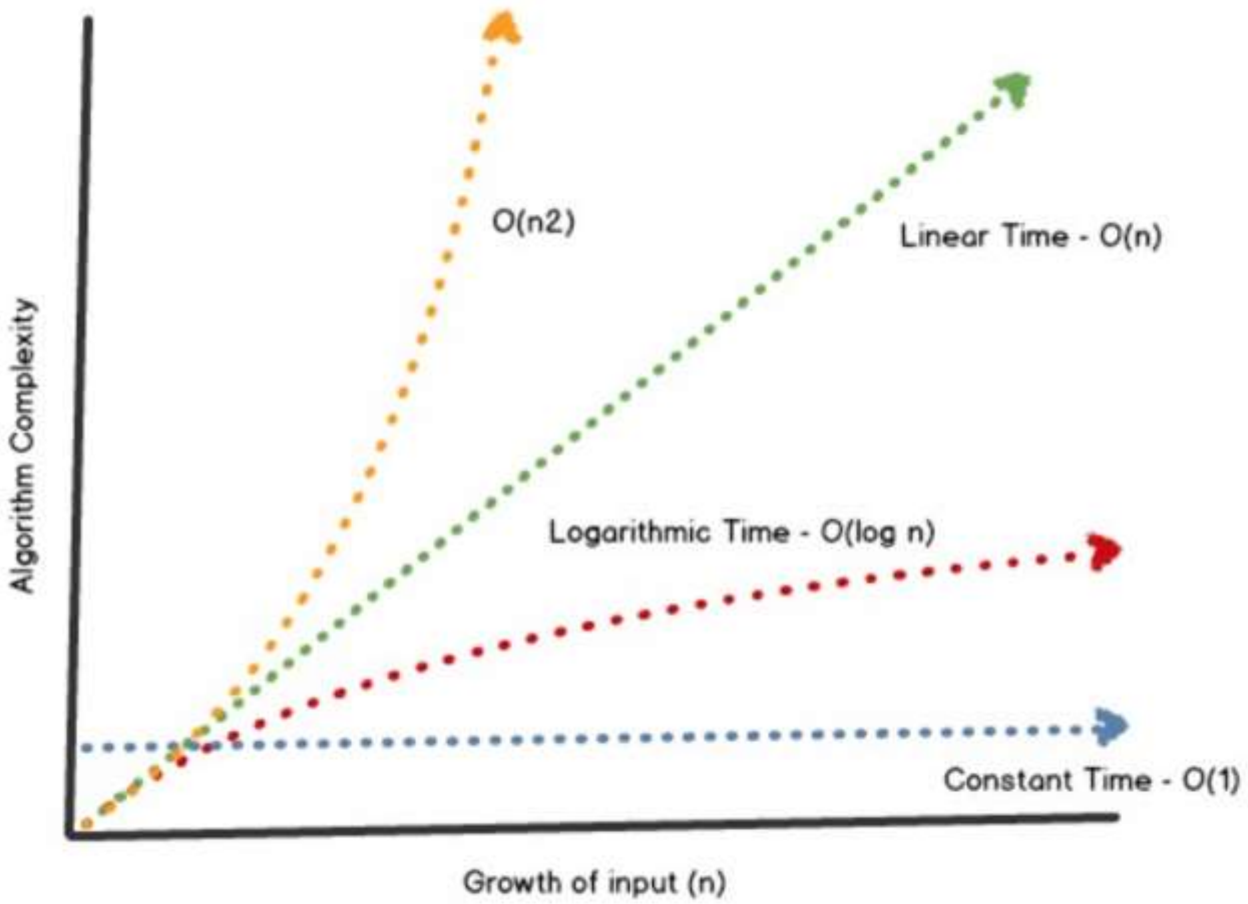
# IN GENERAL...

- Let $S_d$ be the size for depth $d$. Then $S_{d+1} = 2 \cdot S_d + 1$.
- The sequence: $1, 3, 7, 15, 31, 63, \ldots$ may be familiar.
  - It is the sequence $2, 4, 8, 16, 32, 64$, less one. Or:

$$S_d = 2^{d+1} - 1 \quad \text{(approximately } 2^{d+1}\text{)}$$

- It follows that we can pack an $n$ element set into a very shallow tree.
  - If $2^d > n$, we can do it with depth $d$. We can have $d$ about $\log_2 n$.
- $\log_2 n$ grows very slowly.
  - If $n = 100,000,000$ then $\log_2 n \sim 27$.

# Again, why is $\log(n)$ depth for size $n$ elements important?




- If the most common operation is membership query, a list will take $n$ time for $n$ elements.
- A tree will take $\sim\log(n)$ time for $n$ elements. This means the tree scales better than a basic list.

# IMPLEMENTING BINARY SEARCH TREES IN SCHEME

- To represent a node of a tree, we need to maintain 3 data items: the **value** at the node, the **left subtree**, and the **right subtree**.
- We choose to represent these as a list:

```
(value left-subtree right-subtree)
```

- Define helper functions to create trees and extract these fields:

```
(define (make-tree value left right)
  (list value left right))
                                        [redacted]
(define (value T) (car T))
(define (left T) (cadr T))
(define (right T)  (caddr T))
```

# What the heck is a cadr?

cadr = (car (cdr (list))

- Then you can also have: caddr

(car (cdr (cdr (list))))

- You can also have cadddr.
- However: caddddr (Using four ds, is not allowed)

# Scheme: Checking for membership in a Tree

Pseudocode

```
Element?(x,T)
   If T is empty,
      return #f.
   If x = root(T),
      return #t.
   Otherwise:
      if x > root(T)
         return Element?(x, RightChild(T));
      if x < root(T)
         return Element?(x, LeftChild(T)).
```

Scheme Code

```scheme
(define (element? x T)
  (cond ((null? T) #f)
        ((eq? x (value T)) #t)
        ((< x (value T)) (element? x (left T)))
        (else (element? x (right T)))))
```

# Scheme: Inserting into a Tree

Pseudocode

```
Insert(x, T)
If T is empty
        return make-tree(x, (list) (list))
Otherwise
        if x < root(T)
                return Insert(x, left(T))
        if x > root(T)
                return Insert(x, right(T))
```

Scheme Code

```
(define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((eq? x (value T)) T)
        ((< x (value T)) (make-tree (value T)
[redacted]                            (insert x (left T))
                                      (right T)))
        (else (make-tree (value T)
                         (left T)
                         (insert x (right T))))))
```

# MAINTAINING SETS WITH TREES: PERFORMANCE

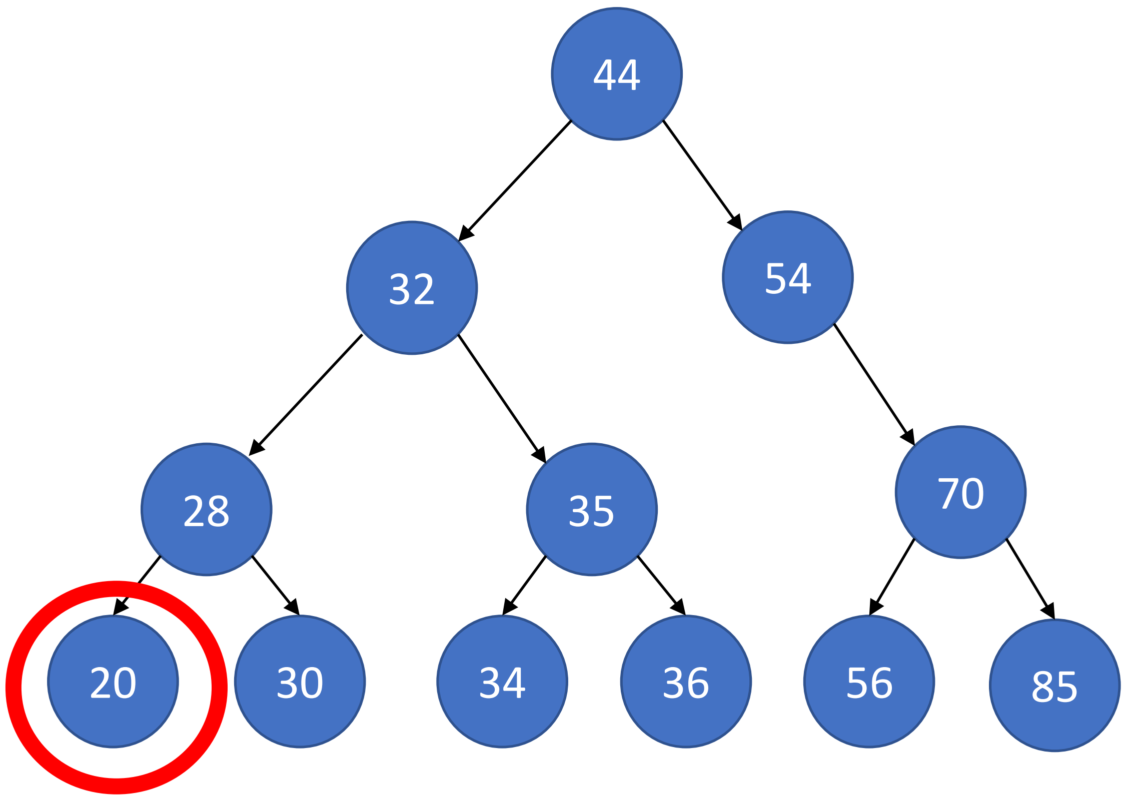
- What happens if you insert elements into a tree in sorted order?
  - You get a skinny tree.
- However, one can prove that if elements are inserted into a tree in random order, the tree is near-bushy with high probability.
- There are also fancier ways of insuring that trees remain balanced.
  - 2-3 trees, splay trees, AVL trees, etc.
  - You'll learn about some of them in your algorithms course.

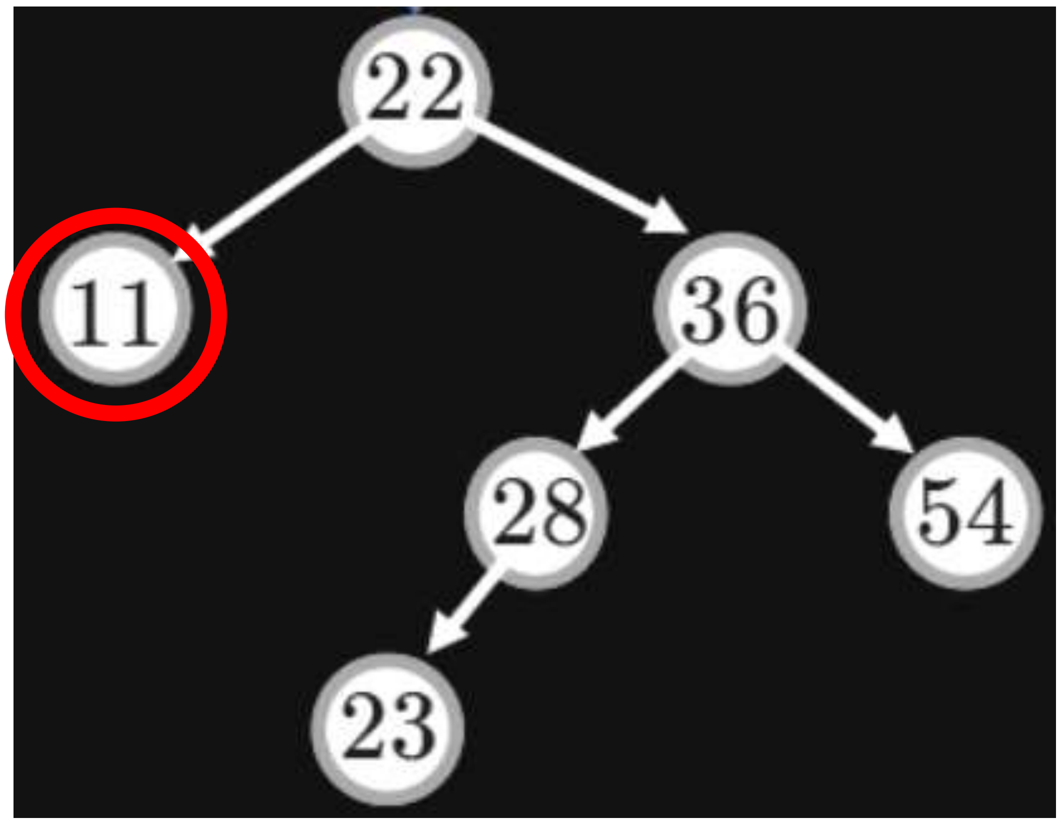
- Note that it is easy to extract the a sorted list of element from a binary search tree:

```
(define (extract-sorted T)
  (if (null? T)
      '()
      (append (extract-sorted (left T))
              (list (value T))
              (extract-sorted (right T)))))
```

(append list1 list2 ...) appends the lists together

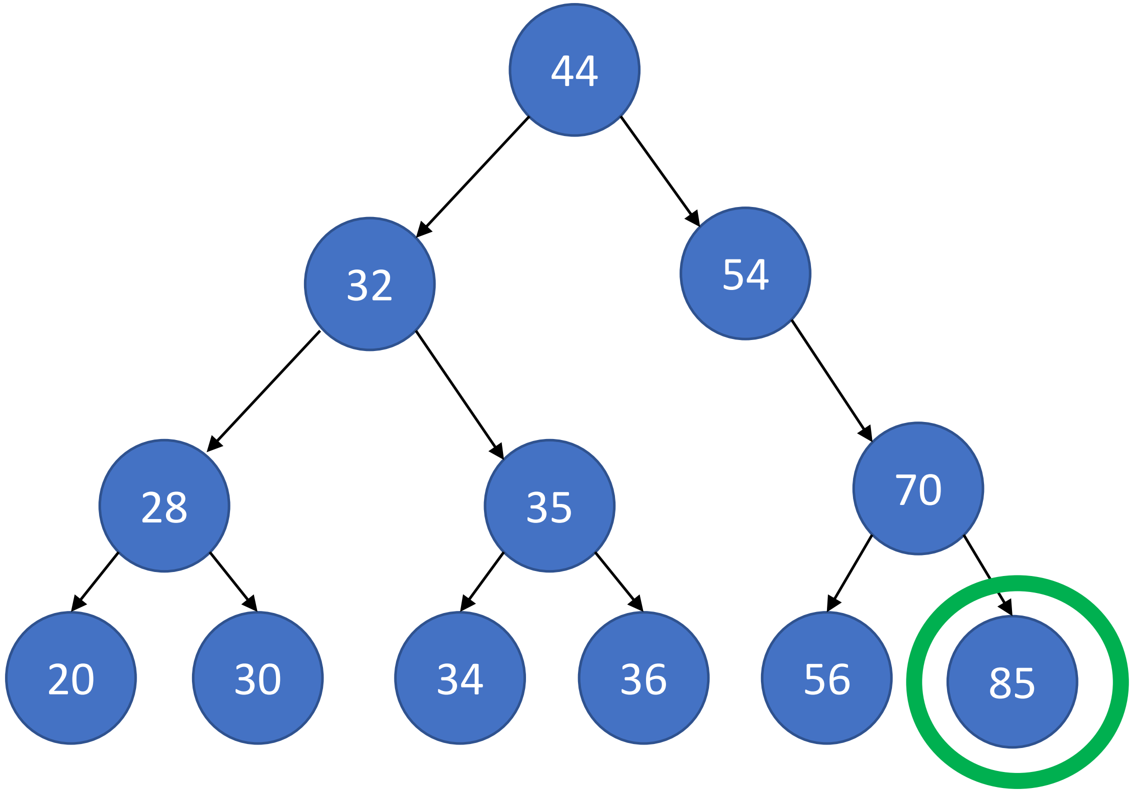
Consider the following trees. Which element is smallest?

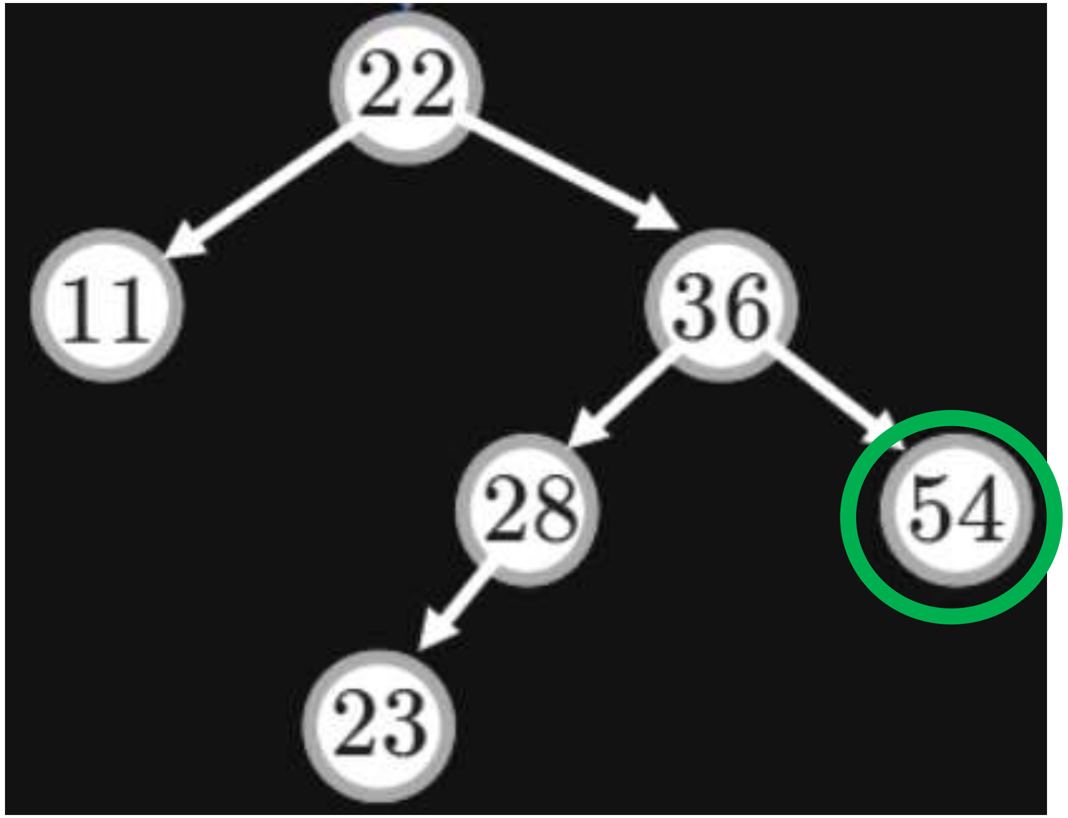



*Do you see any pattern in finding the smallest element in a binary search tree?*

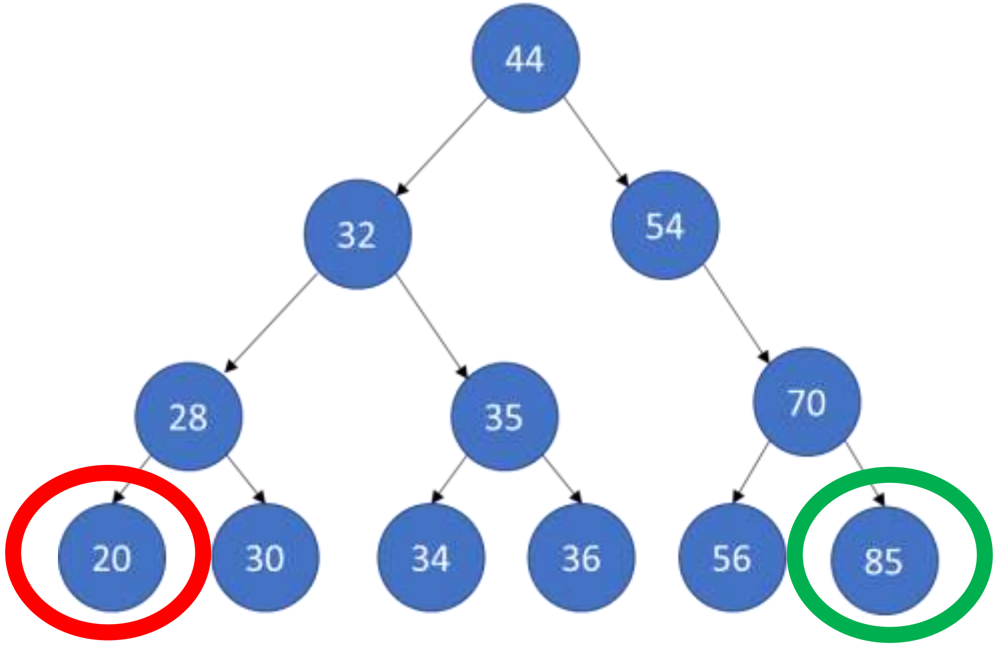
Consider the following trees. Which element is largest?

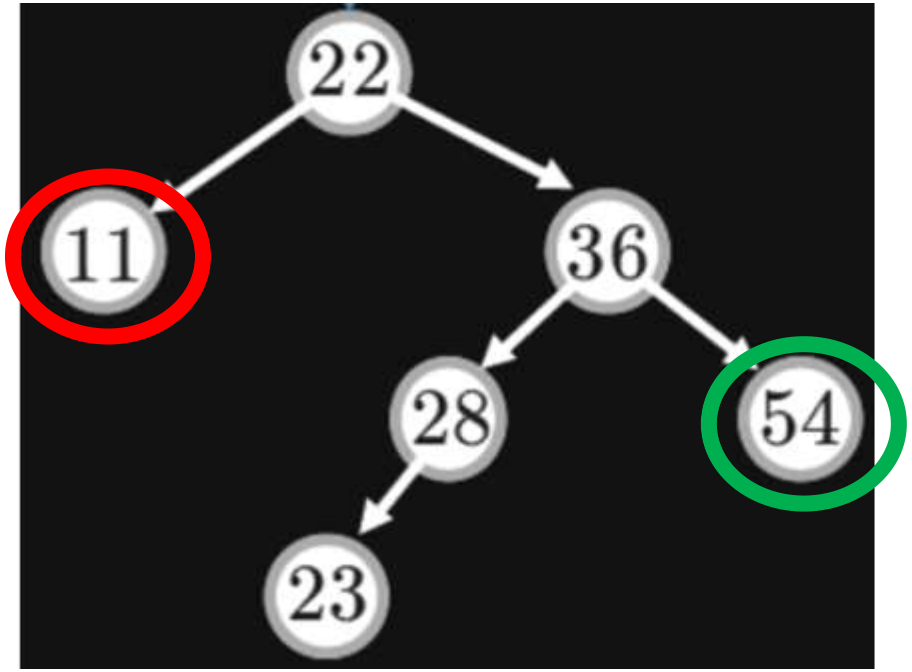



*Do you see any pattern in finding the largest element in a binary search tree?*

# Finding the smallest and largest elements in a binary search tree




- It should be obvious that the leftmost element (circled in red) in a binary search tree is always the smallest.
- Likewise it should be obvious that the rightmost element (circled in gree) in a binary search tree is always the largest.

# Figure Sources

- https://static0.gamerantimages.com/wordpress/wp-content/uploads/2021/08/lord-of-the-rings-ents-feature-treebeard-picture-bright.jpg?q=50&fit=crop&w=1800&dpr=1.5
- https://www.xoriant.com/sites/default/files/uploads/2017/08/Decision-Trees-modified-1.png
- https://i.kym-cdn.com/entries/icons/original/000/004/006/YUNO.jpg
- https://static.wikia.nocookie.net/the-fairy-world/images/2/25/Wizard_by_adam_brown-d3iiyfb.jpg/revision/latest?cb=20171109173637&path-prefix=hr
- https://techsupport.cambridgeaudio.com/hc/article_attachments/360000169198/log_lin_approx.jpg