

ECE 264 Spring 2023
***Advanced* C Programming**

Aravind Machiry
Purdue University

Linked List

Dynamic Structures

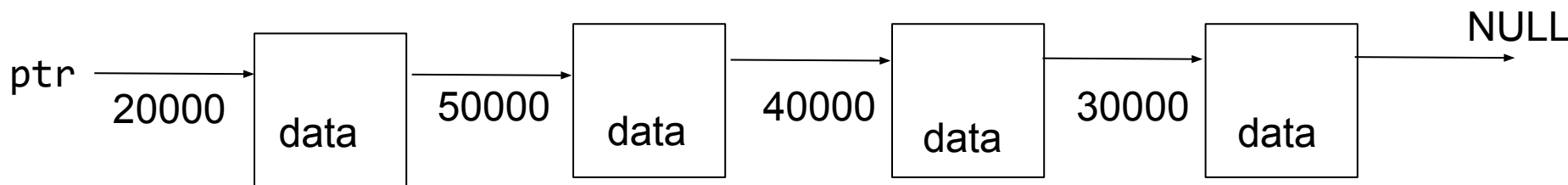
- Memory management:
 - Allocate memory when writing a program
 - Allocate memory after a program starts. Free before the program ends
- Allocate memory when needed. Free when no longer needed.
- Dynamic structures are used widely for problems whose sizes may change over time: database, web users, text editor, ...

General Concept

- a pointer ptr in the stack memory
- ptr points to heap memory
- The structure has a pointer and contains data
- The last piece points to NULL
- Each piece is called a node.

Stack Memory		
	Address	Value
ptr	100	20000

Heap Memory	
Address	Value
	data
50000	40000
	data
40000	30000
	data
30000	NULL
	data
20000	50000



Why Heap or Stack Memory

- Heap memory can be allocated / freed. Stack memory cannot.
- Local variables and arguments are in stack memory
- Heap memory can be accessed by different functions
- malloc returns the allocated heap memory. malloc does not necessary return increasing or decreasing orders
- After malloc / free several times, the memory may be scattered

List Manipulation Functions

- Print List
 - Insert Into List
 - Delete an Element
 - Destroy List
-
- Can we do these in recursion?

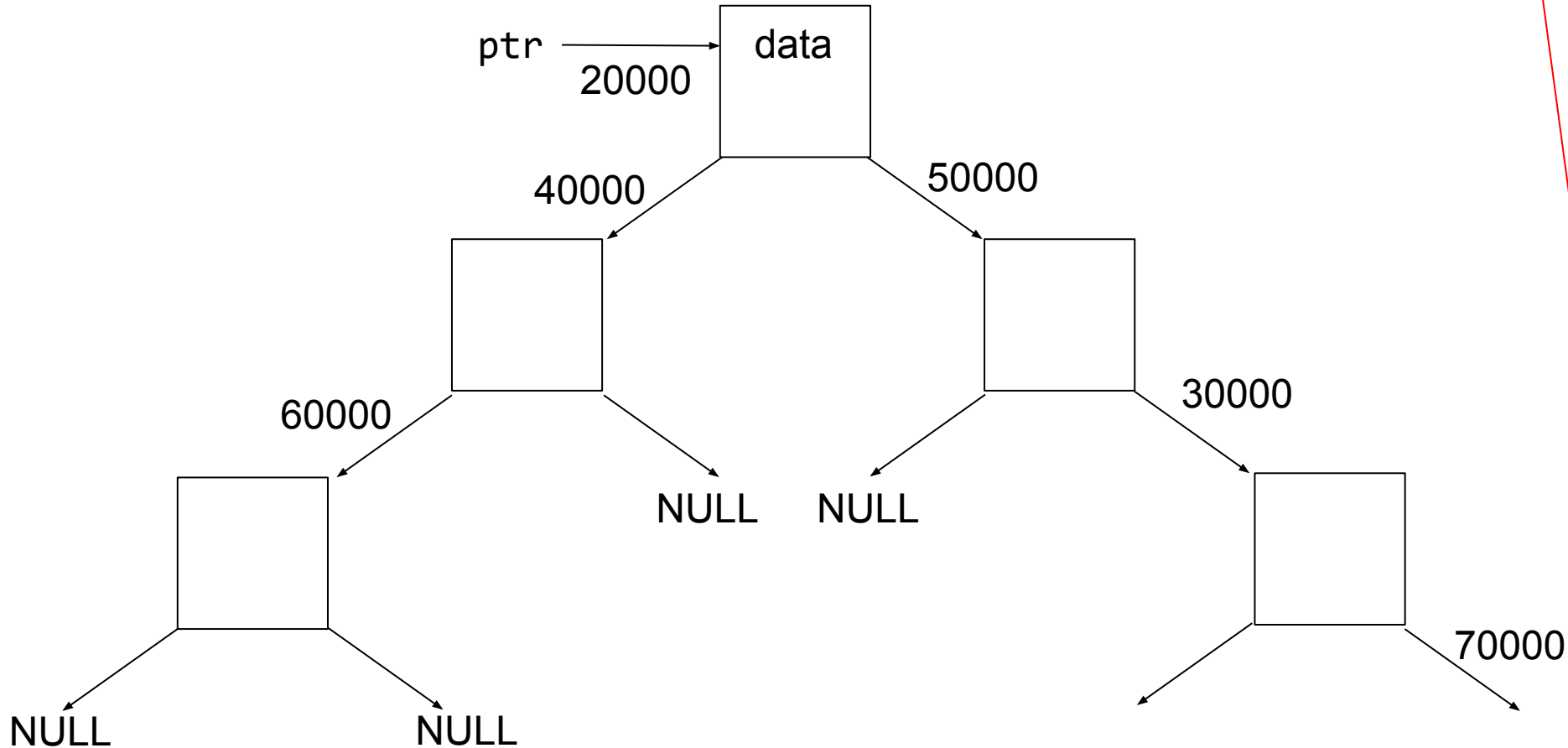
Binary Tree

Binary tree (Review)

Each piece of memory has two pointers

Stack Memory		
	Address	Value
ptr	100	20000

Heap Memory	
Address	Value
	data
	NULL
70000	NULL
	data
	NULL
60000	NULL
	data
	30000
50000	NULL
	data
	NULL
40000	60000
	data
	70000
30000	NULL
	data
	50000
20000	40000

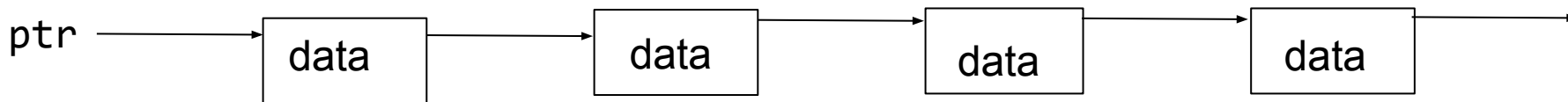


Difference between one and two

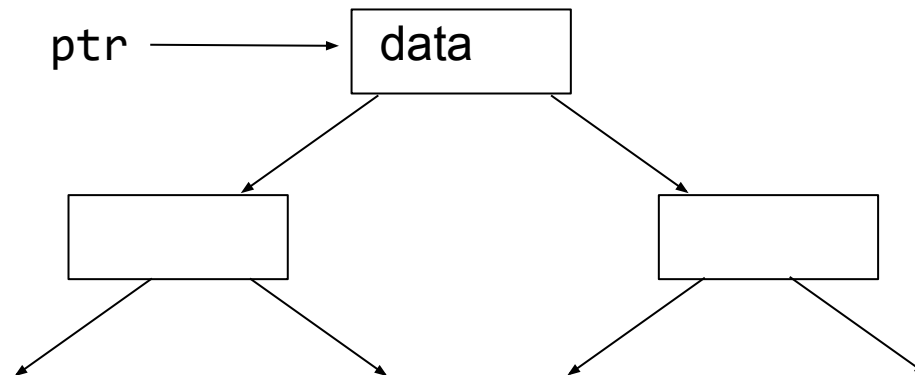
-

Linked List vs Binary Tree

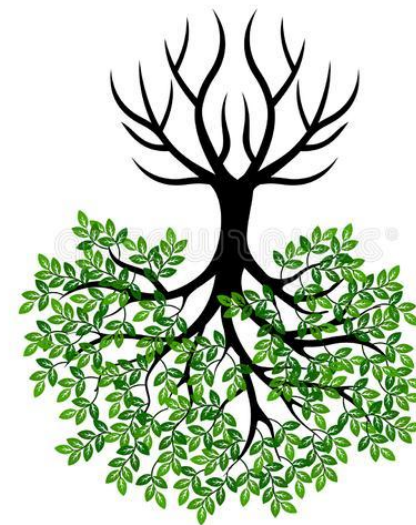
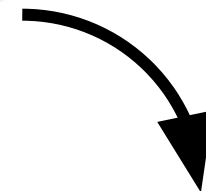
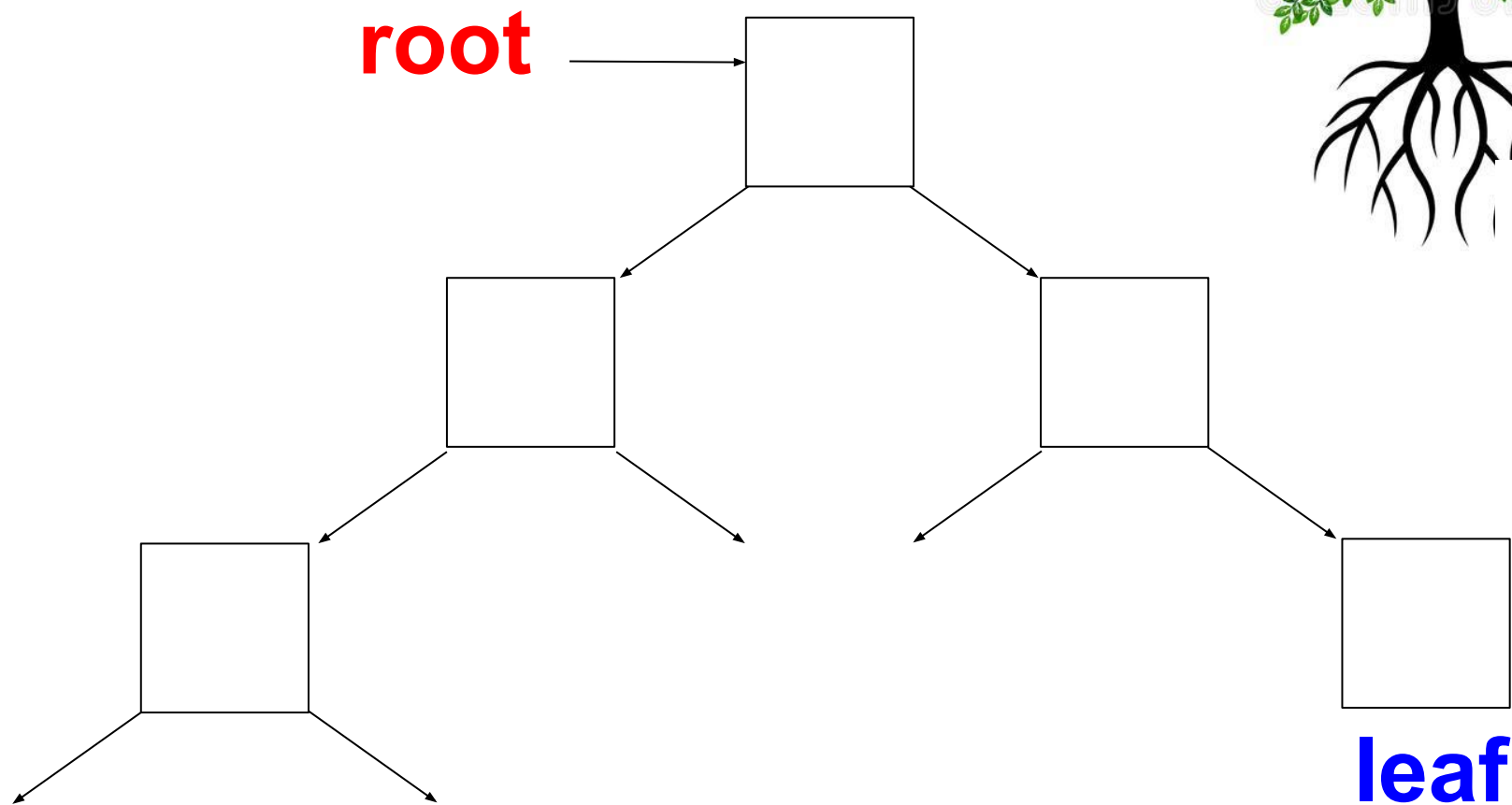
- Linked list is one-dimensional. Going to the middle has to pass half of the list.



- Binary tree is two dimensional and can eliminate (about) half data in a single step.



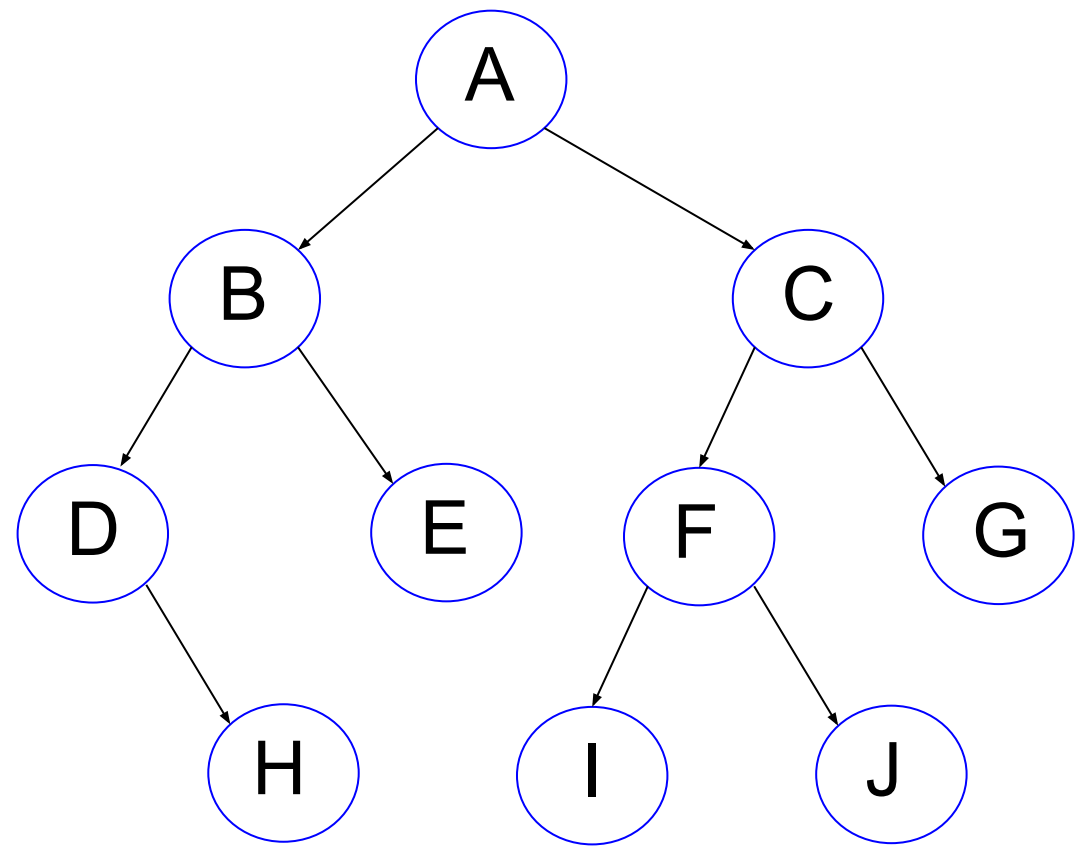
“Upside Down” Tree



leaf

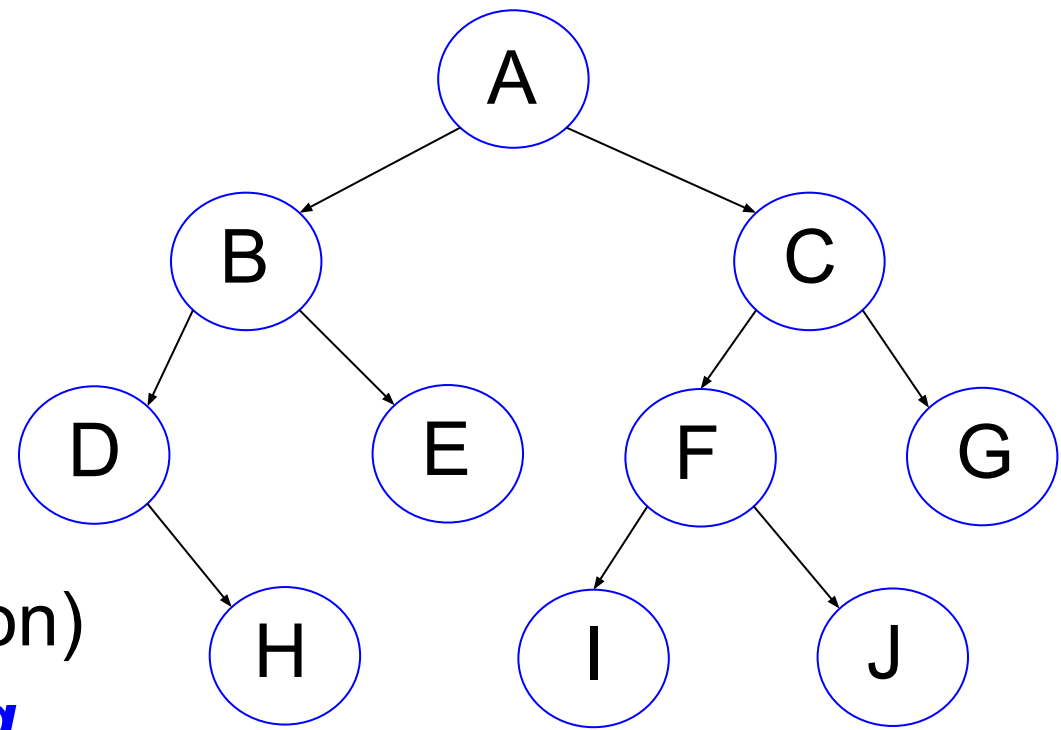
Terminology

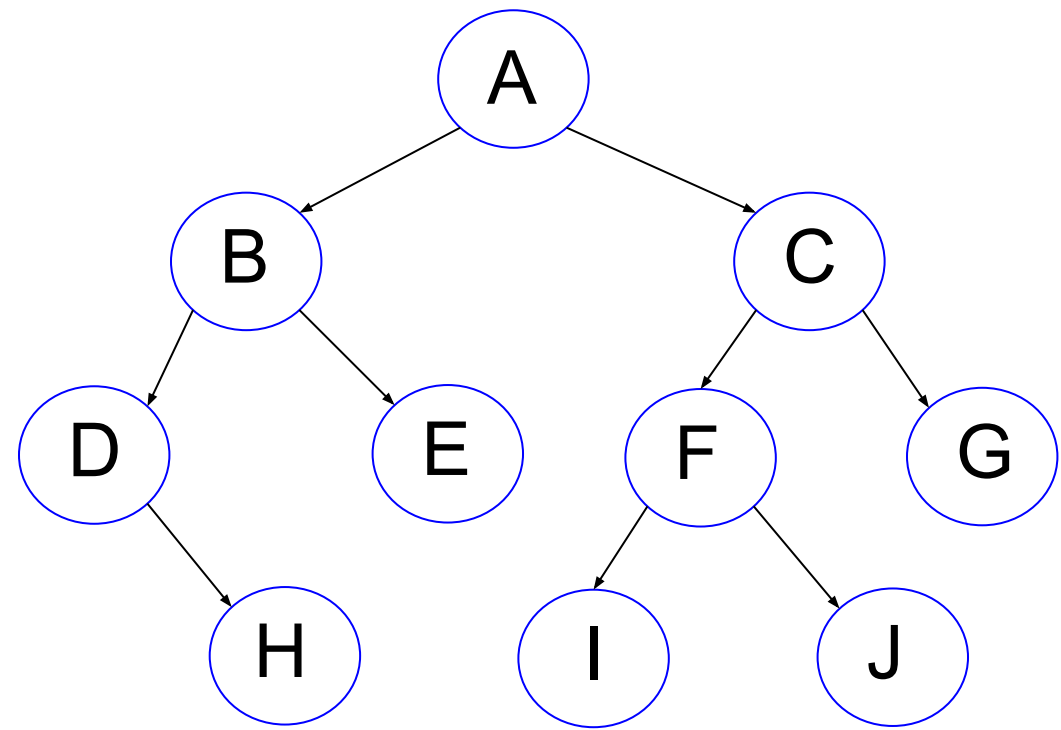
- A, B, C ... : each is a **node**
- An **edge** connects from A to B
- Do not draw edges point to NULL
- A is the **parent** node of B and C
- B and C are A's **child** nodes
- B and C are **siblings**
- **Binary tree**: each node has at most two children
- If a node has no parent node, this node is the tree's **root**
- If a node has no child node, this is a **leaf** node



Terminology

- If A is B's parent, A is B's **ancestor**.
- If A is B's parent, B is D's ancestor, A is D's ancestor (recursive definition)
- If A is B's ancestor, B is A's **offspring**.
- A **path** is the sequence of edges from an ancestor to an offspring.
- The **height** of a node is the length of the longest path to a leaf.
The heights of E, D, B are 0, 1, 2 respectively.
- The **height** of a tree is the height of the root.
The height of the tree is 3.



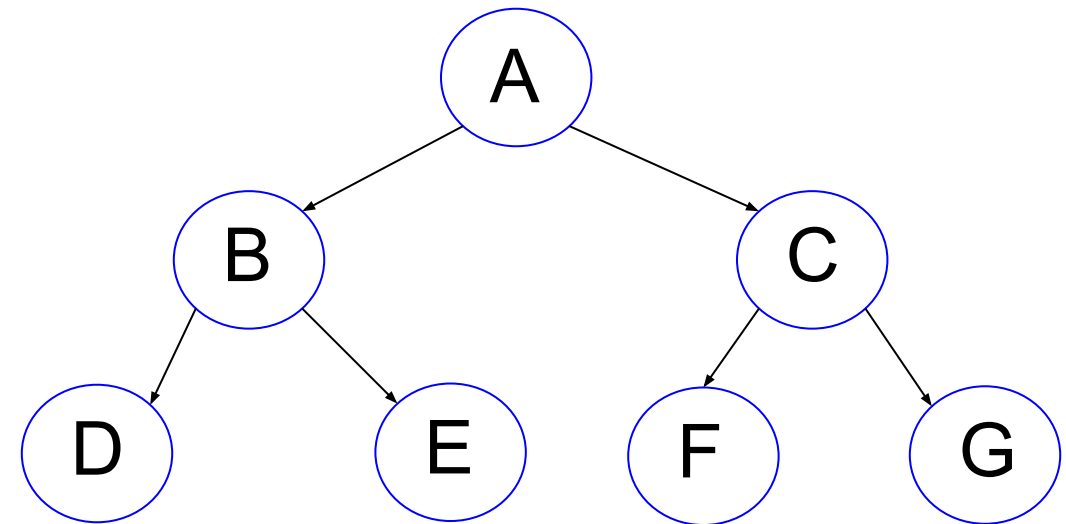


- The **depth** of a node is the distance to the root
- **Full** binary tree: nodes have 2 children or 0 child
- **Perfect** binary tree: full + leaf nodes of the same distance to root
- B and B's offsprings are A's left **subtree**.
- C and C's offsprings are A's right subtree.

Why Are Binary Tree Important

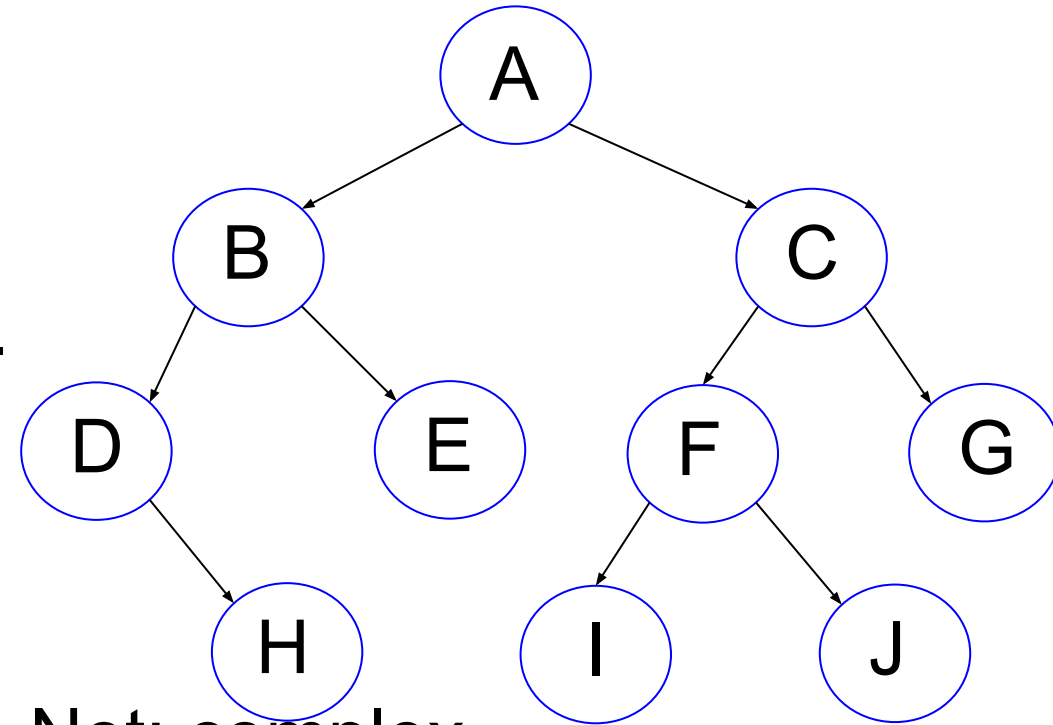
- Power of logarithm: $\log(n)$ grows very slowly. 2^n grows very fast
- A perfect binary tree of height n has $2^{n+1} - 1$ nodes
- In a single step, a program decides to go left or right:

```
if (condition)
{
    go left
}
else
{
    go right
}
```

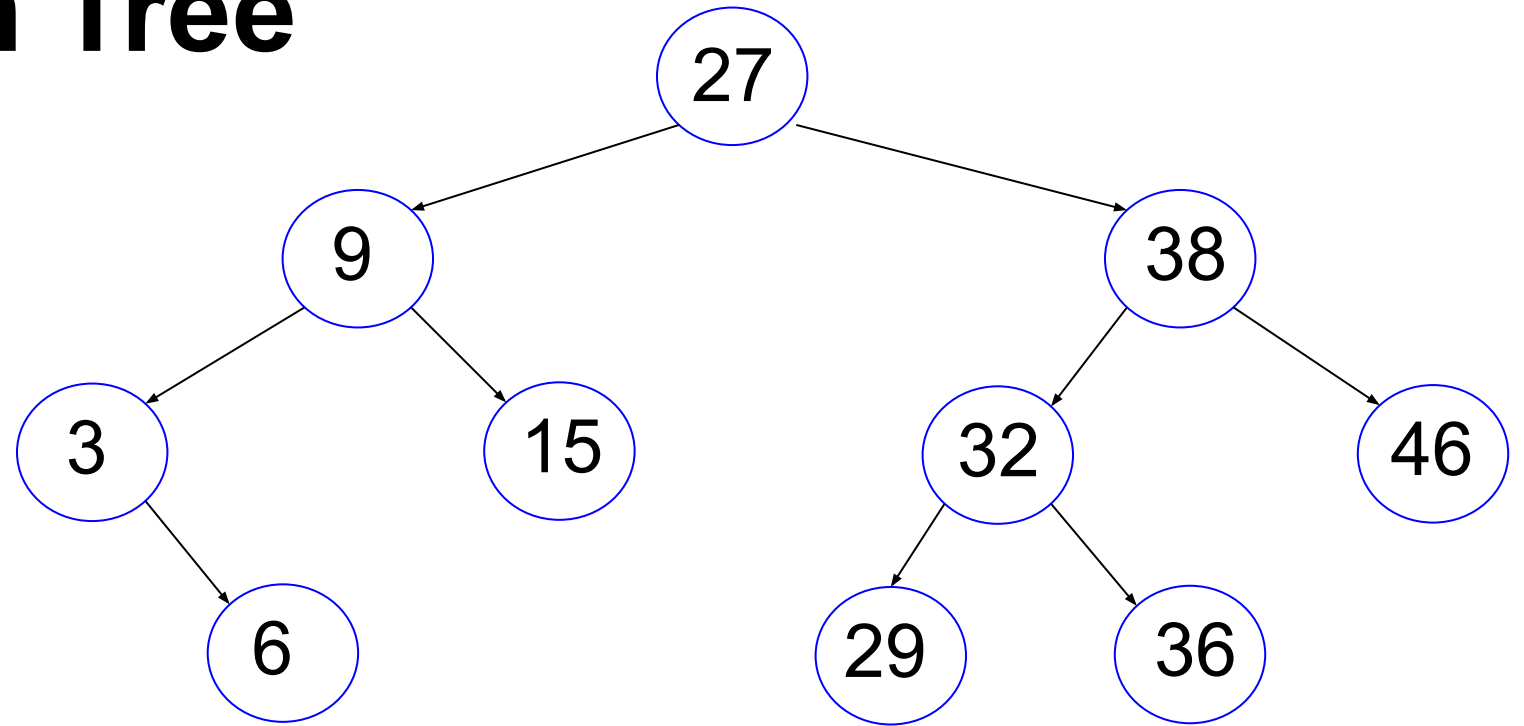


Binary Search Tree

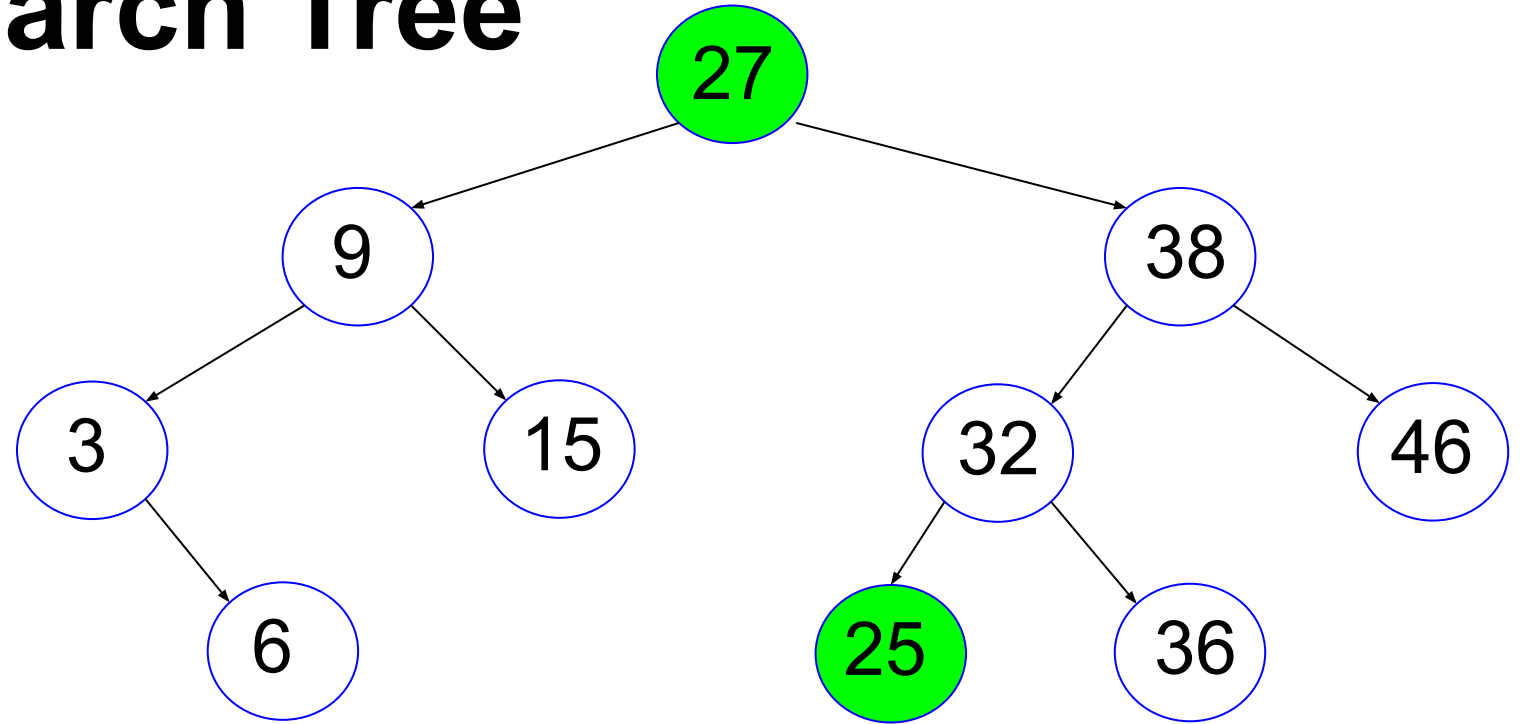
- Every node stores a value as the **key**.
- The keys must be **totally ordered**:
 - if $a \leq b$ and $b \leq a$ then $a = b$
 - if $a \leq b$ and $b \leq c$ then $a \leq c$
 - either $a \leq b$ or $b \leq a$
- Totally ordered: integer, real numbers. Not: complex.
- For **every node**, the following is tree:
 - Keys of all nodes of the left subtree of a node $<$ this node's key
 - Keys of all nodes of the right subtree of a node $>$ this node's key



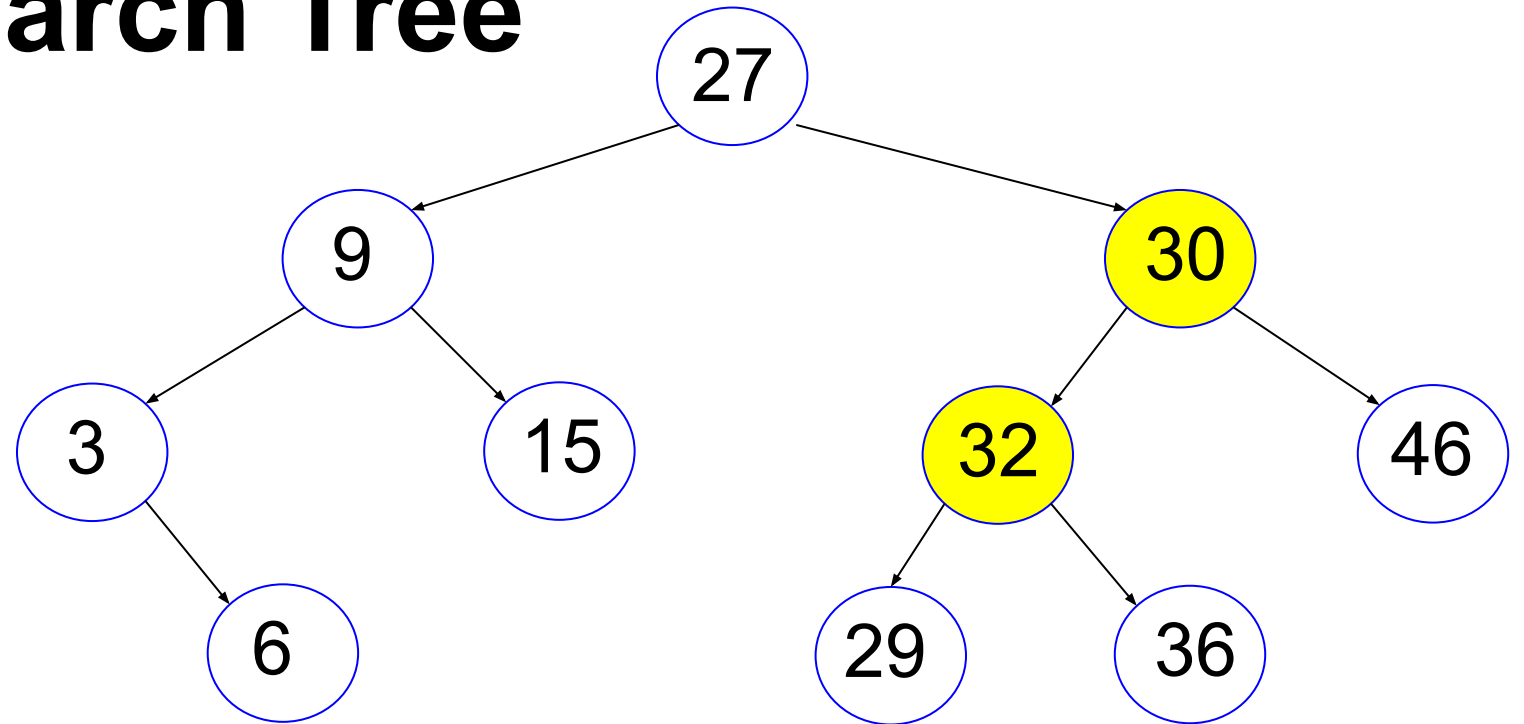
Binary Search Tree



Not Binary Search Tree



Not Binary Search Tree

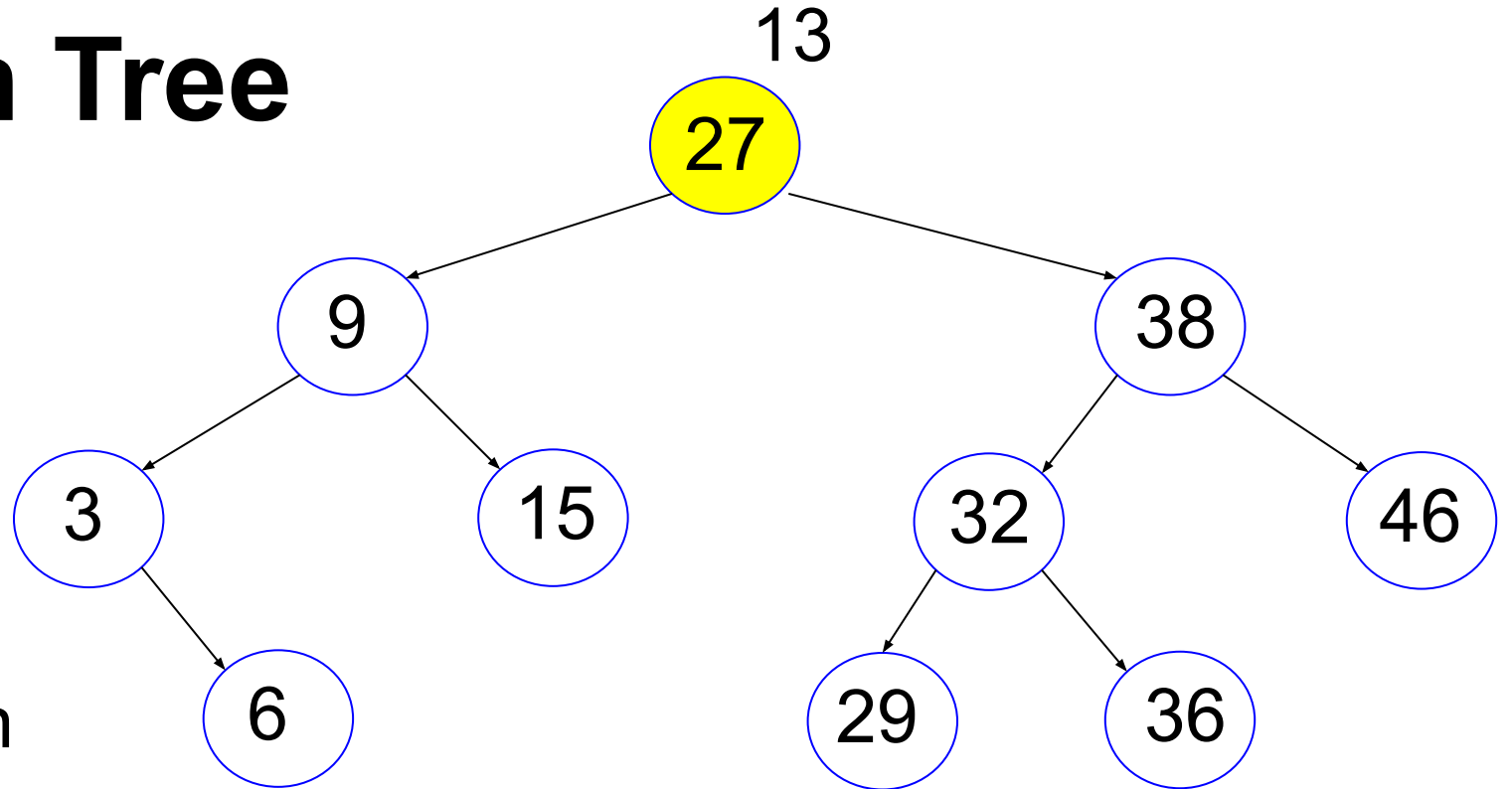


Binary Tree is a Container Structure

- insert: insert data
- delete: delete (a single piece of) data
- search: is a piece of data stored
- destroy: delete all data

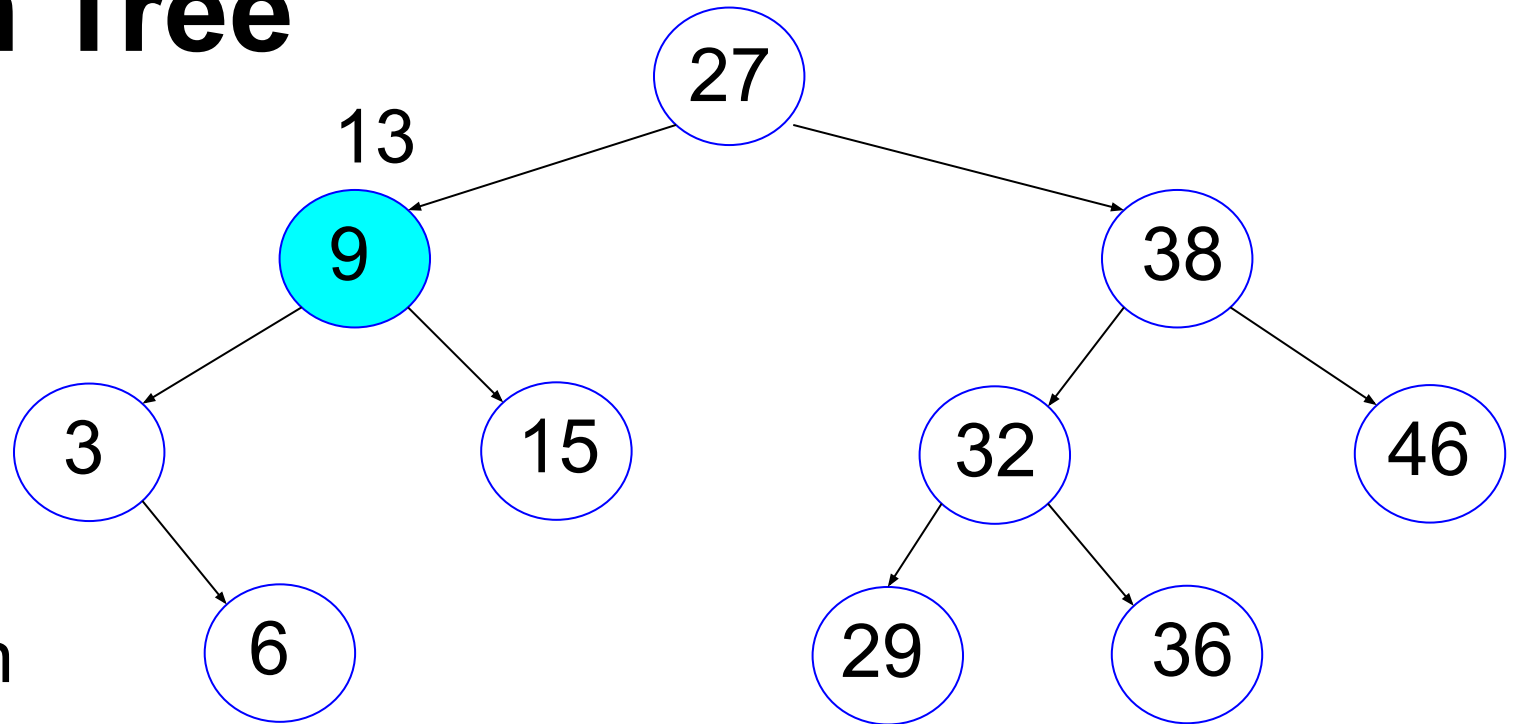
Binary Search Tree

- Is 13 stored?
- $13 < 27 \Rightarrow$ go left
- $13 > 9 \Rightarrow$ go right
- $13 > 15 \Rightarrow$ go left
- Nothing \Rightarrow 13 is not in



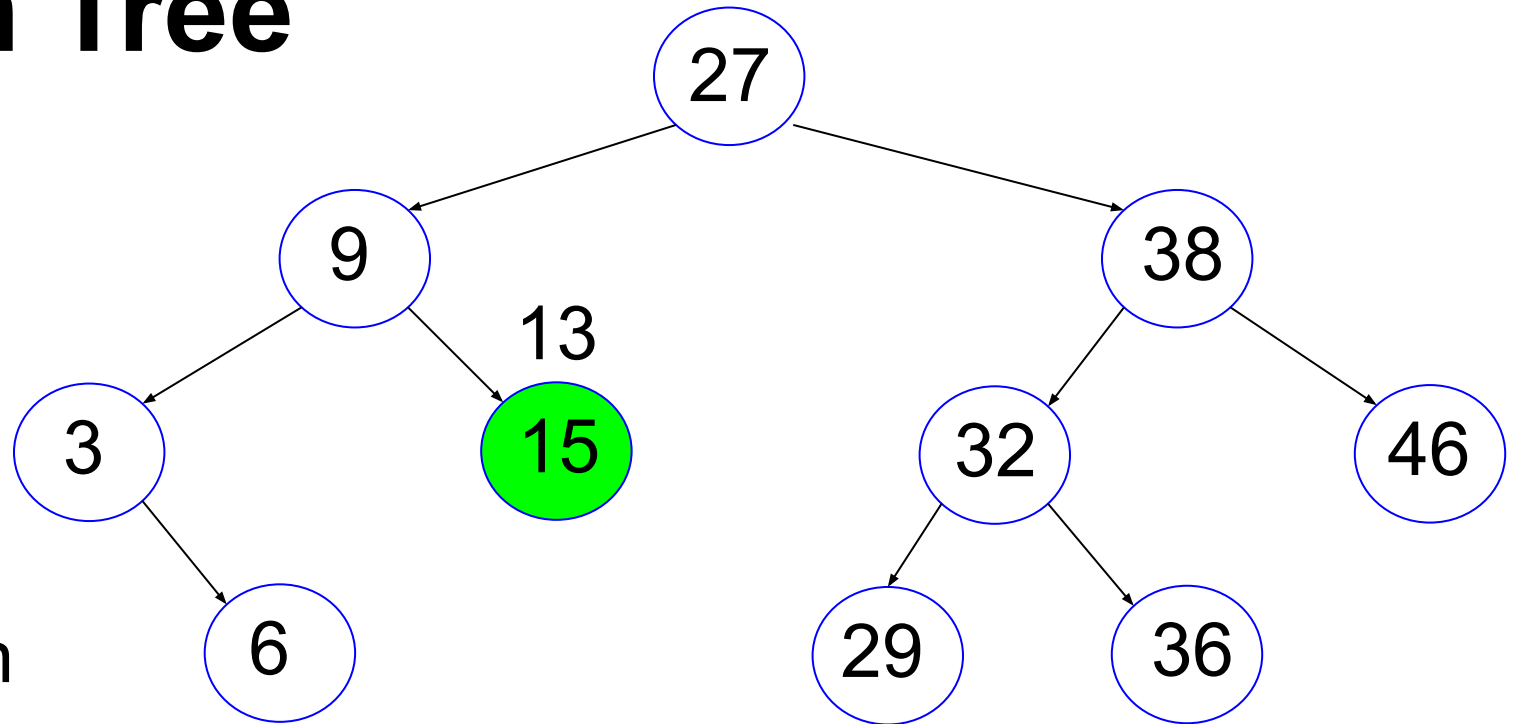
Binary Search Tree

- Is 13 stored?
- $13 < 27 \Rightarrow$ go left
- $13 > 9 \Rightarrow$ go right
- $13 > 15 \Rightarrow$ go left
- Nothing \Rightarrow 13 is not in



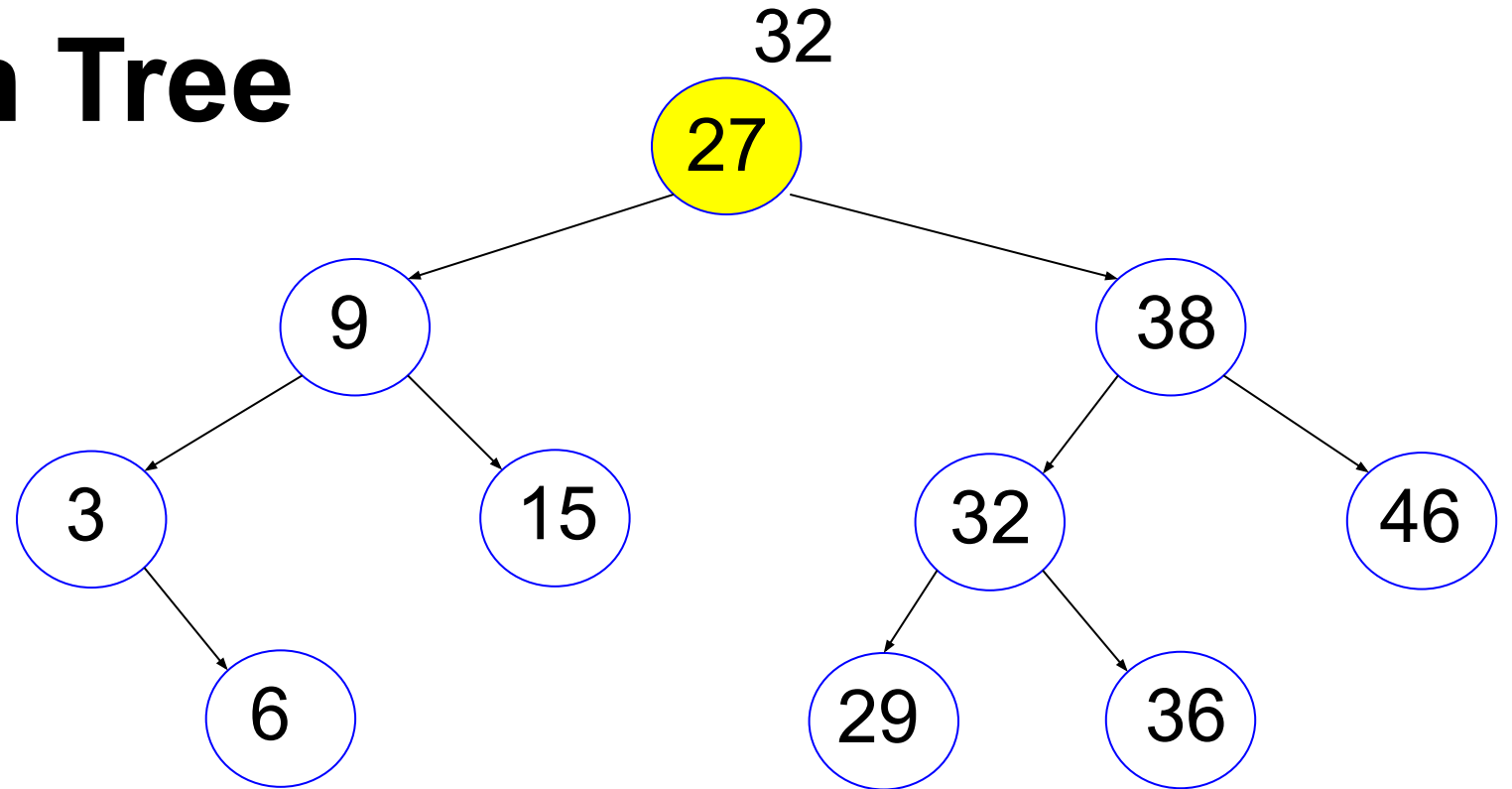
Binary Search Tree

- Is 13 stored?
- $13 < 27 \Rightarrow$ go left
- $13 > 9 \Rightarrow$ go right
- **$13 > 15 \Rightarrow$ go left**
- Nothing \Rightarrow 13 is not in



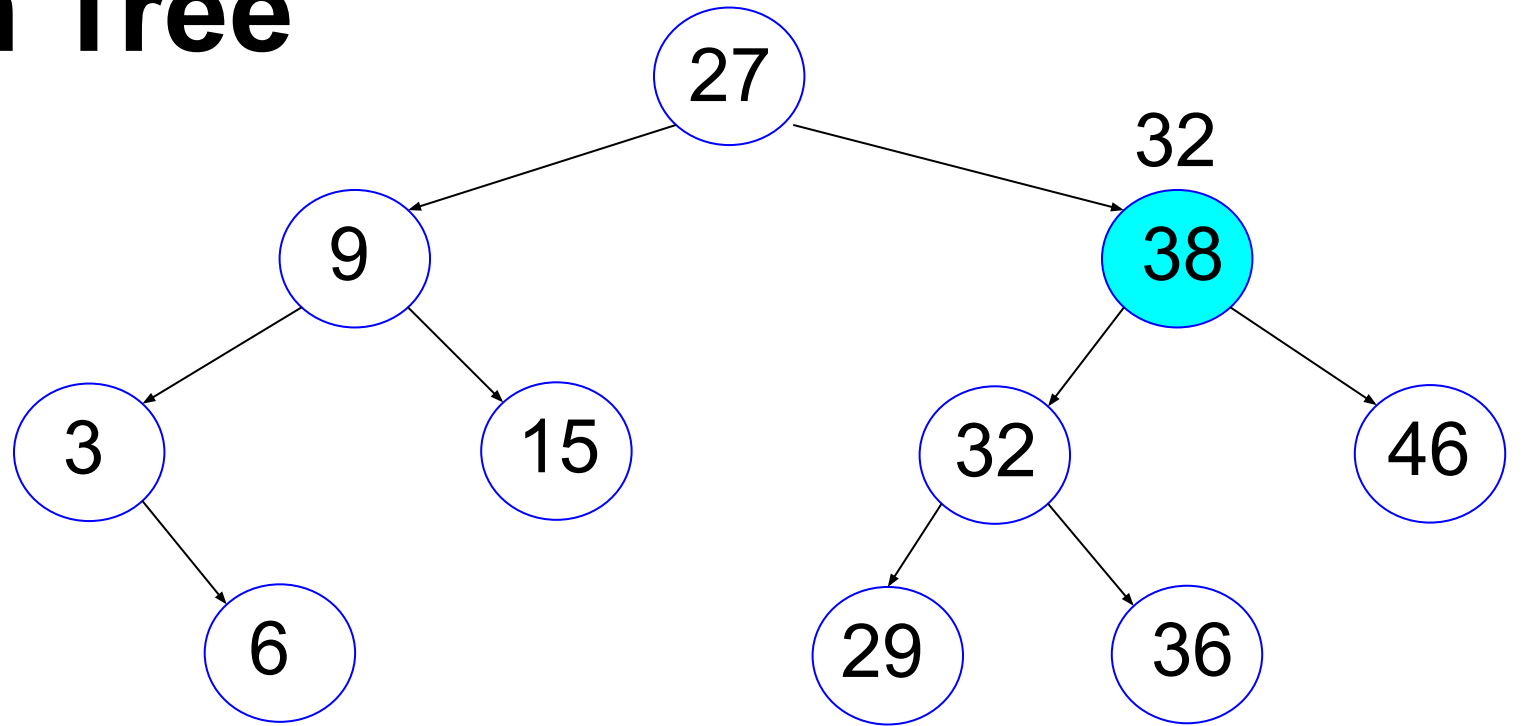
Binary Search Tree

- Is 32 stored?
- $32 > 27 \Rightarrow$ go right
- $32 < 38 \Rightarrow$ go left
- $32 = 32 \Rightarrow$ found



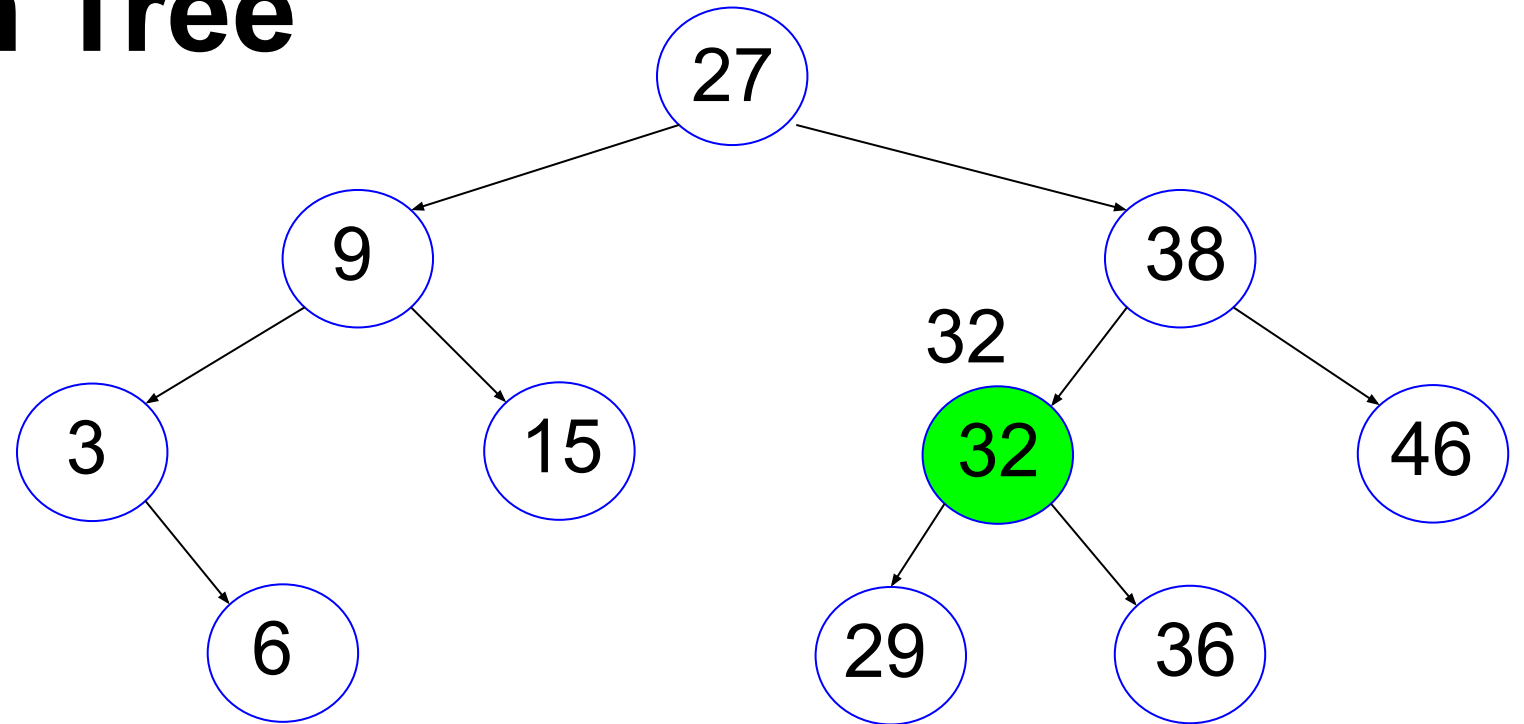
Binary Search Tree

- Is 32 stored?
- $32 > 27 \Rightarrow$ go right
- $32 < 38 \Rightarrow$ go left
- $32 = 32 \Rightarrow$ found



Binary Search Tree

- Is 32 stored?
- $32 > 27 \Rightarrow$ go right
- $32 < 38 \Rightarrow$ go left
- **$32 = 32 \Rightarrow$ found**



```
typedef struct tnode
{
    struct tnode * left;
    struct tnode * right;
    // data, must have a way to compare keys
    // may be a structure
    int value; // use int for simplicity
} TreeNode;
// search a value in a binary search tree starting
// with r, return the node whose value is v,
// or NULL if no such node exists
TreeNode * Tree_search(TreeNode * tn, int v);
```

```
TreeNode * Tree_search(TreeNode * tn, int val)
{
    if (tn == NULL) { return NULL; } // cannot find
    if (val == (tn -> value)) // found
        { return tn;}
    if (val < (tn -> value))
        {
            // search the left side
            return Tree_search(tn -> left, val);
        }
    return Tree_search(tn -> right, val);
}
```

three components of recursion:

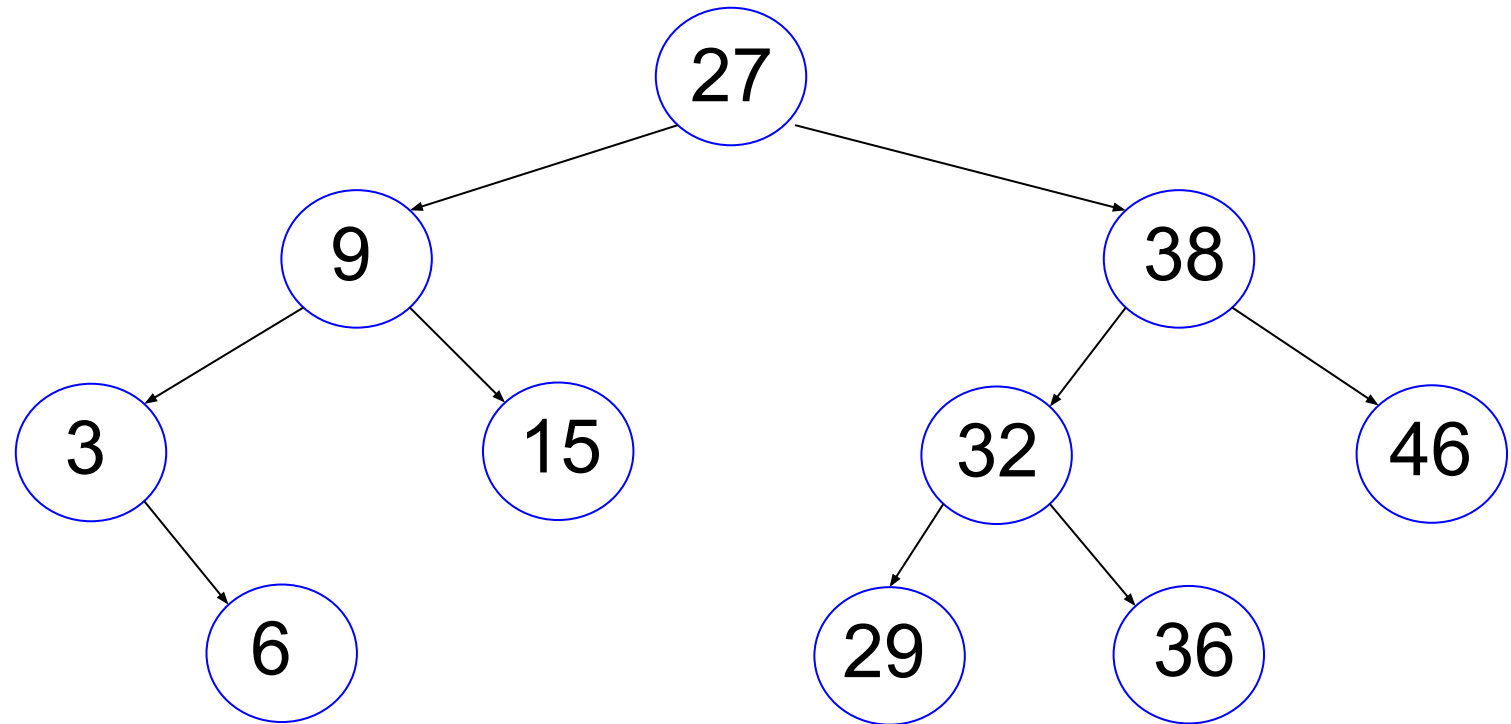
1. stop condition: NULL
2. change: go to child
3. recurring pattern: same method to search

Binary Tree Insert

Binary Search Tree

How to create a tree like this?

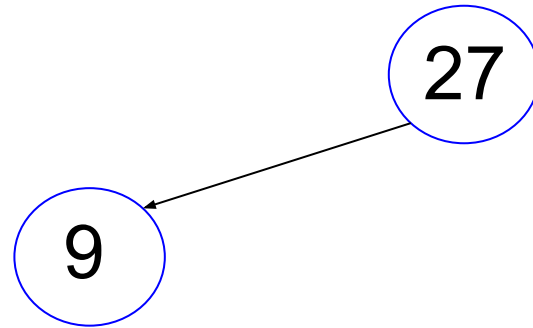
The insert function



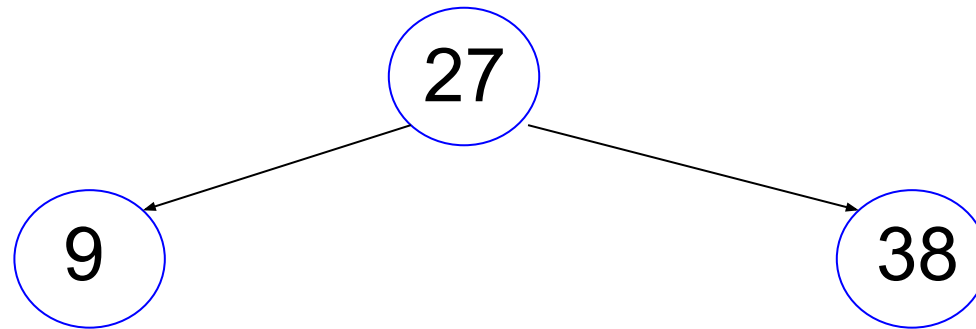
insert 27

27

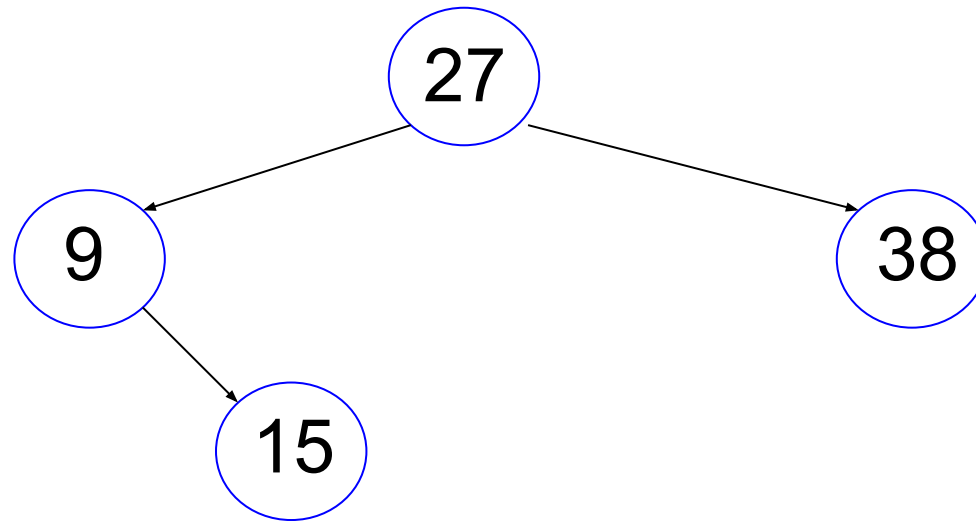
insert 27, 9



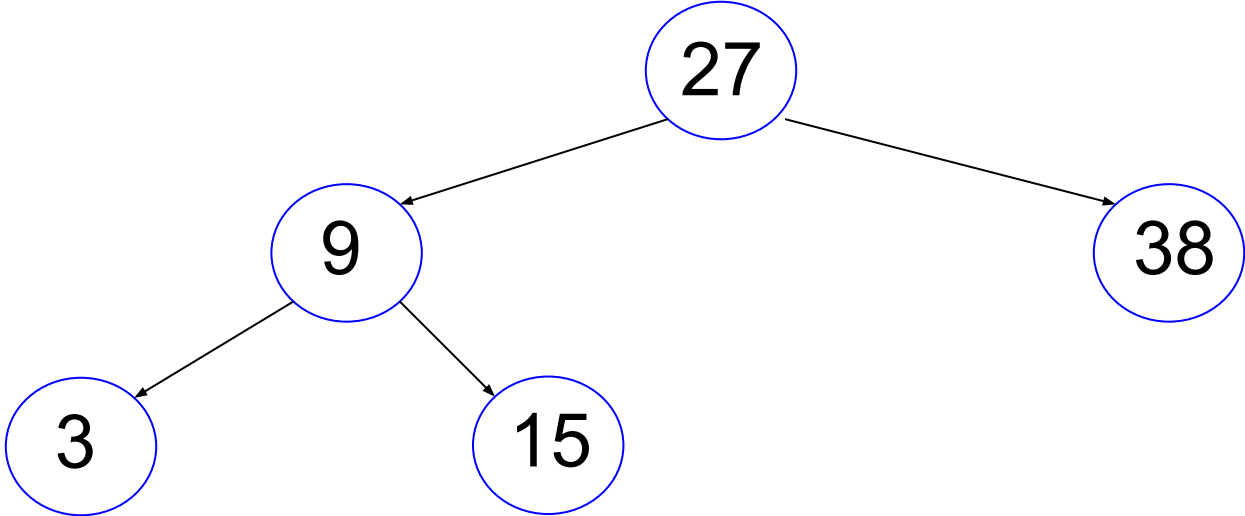
insert 27, 9, 38



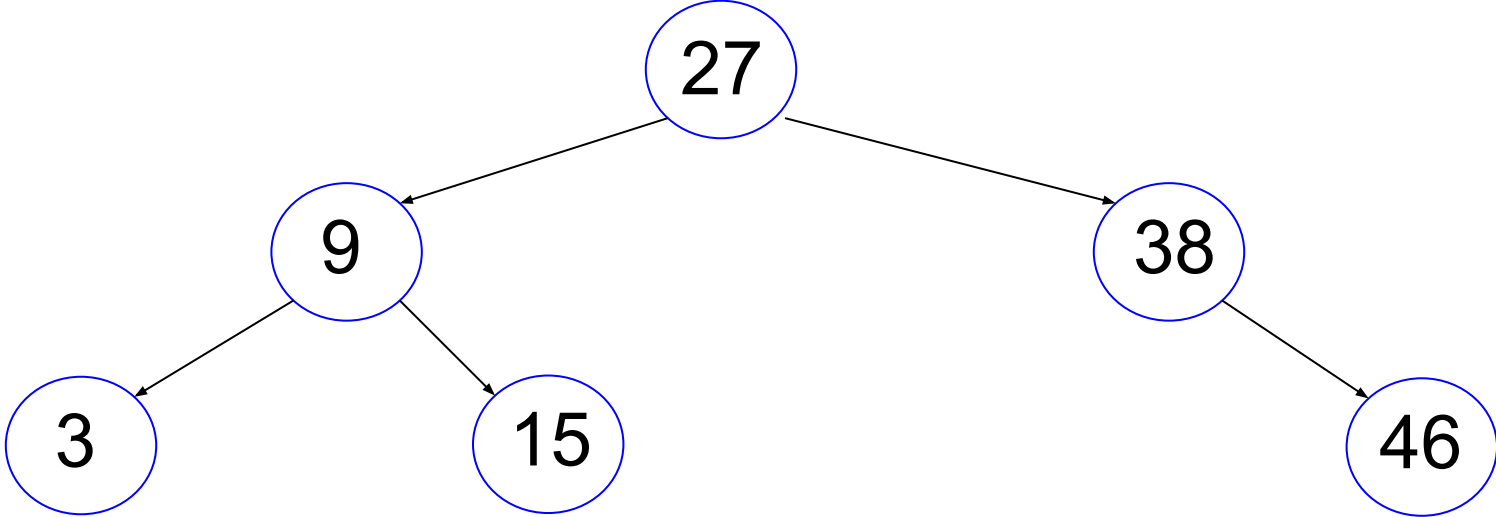
insert 27, 9, 38, 15



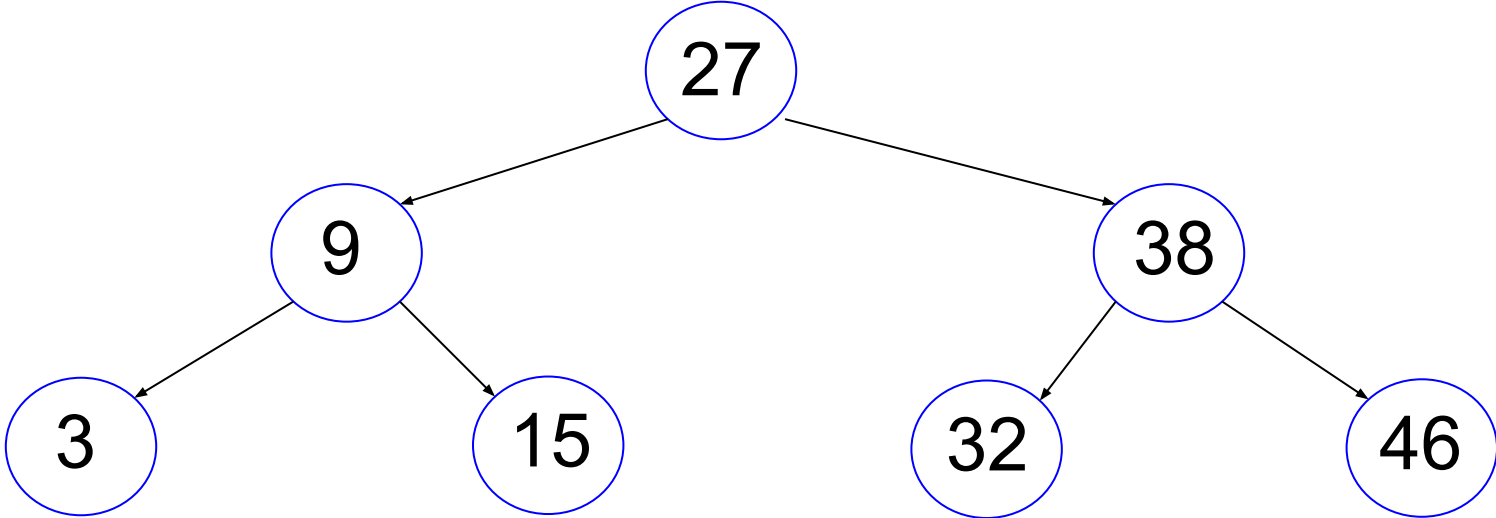
insert 27, 9, 38, 15, 3



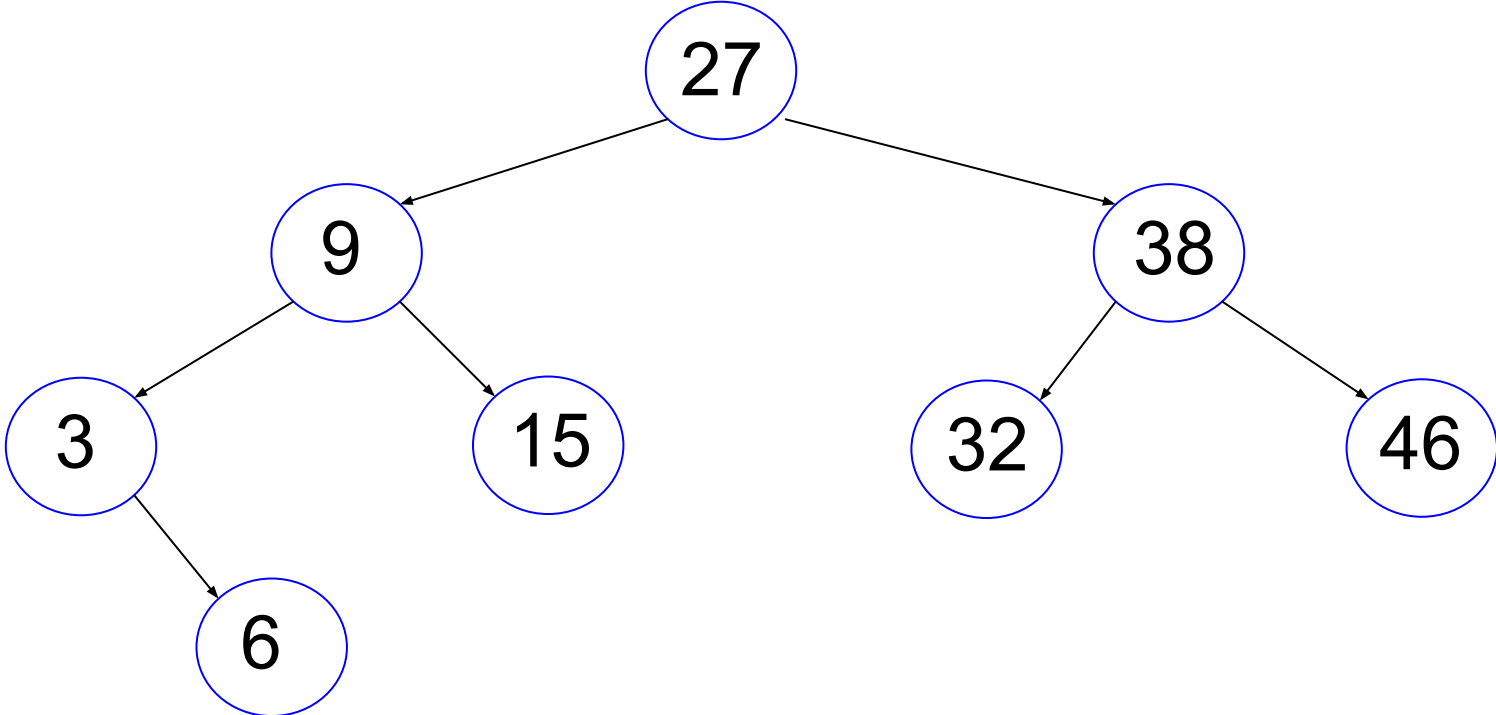
insert 27, 9, 38, 15, 3, 46



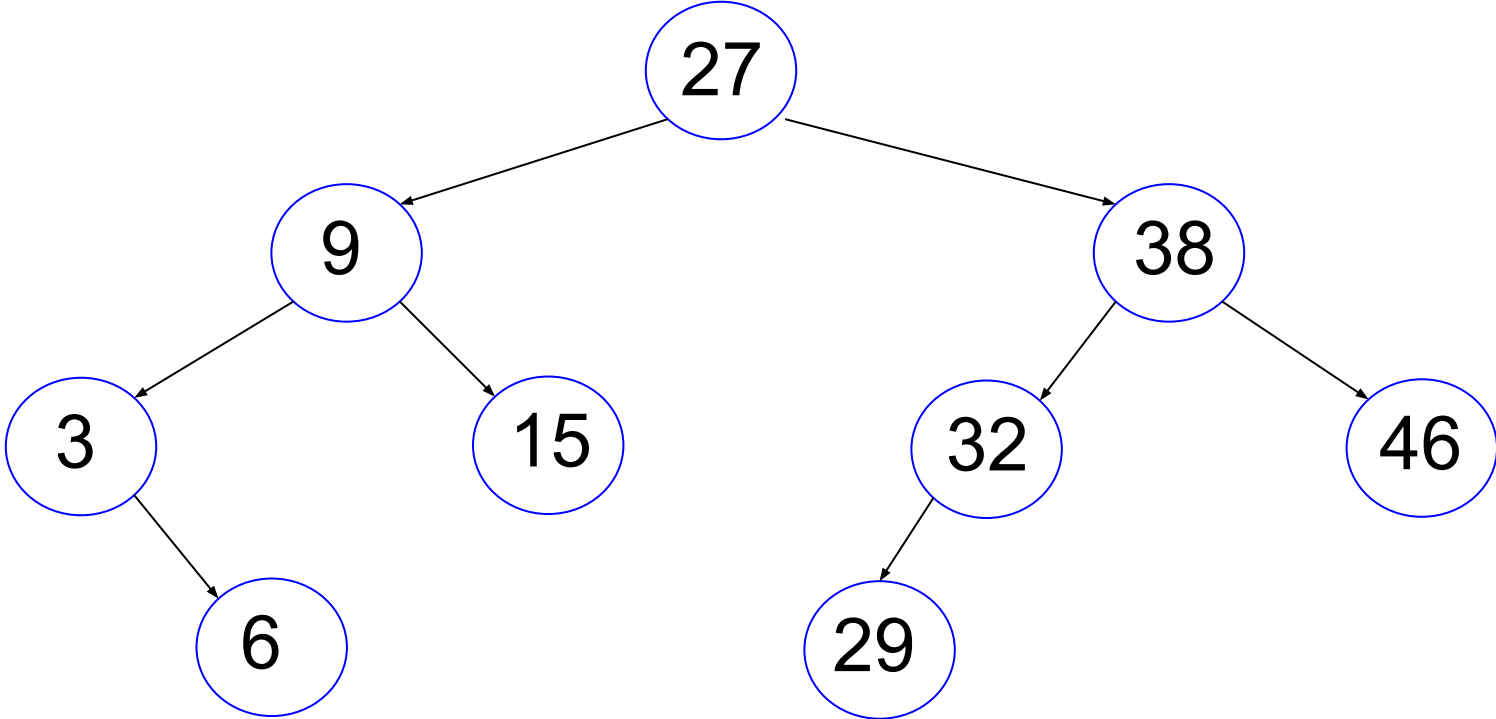
insert 27, 9, 38, 15, 3, 46, 32



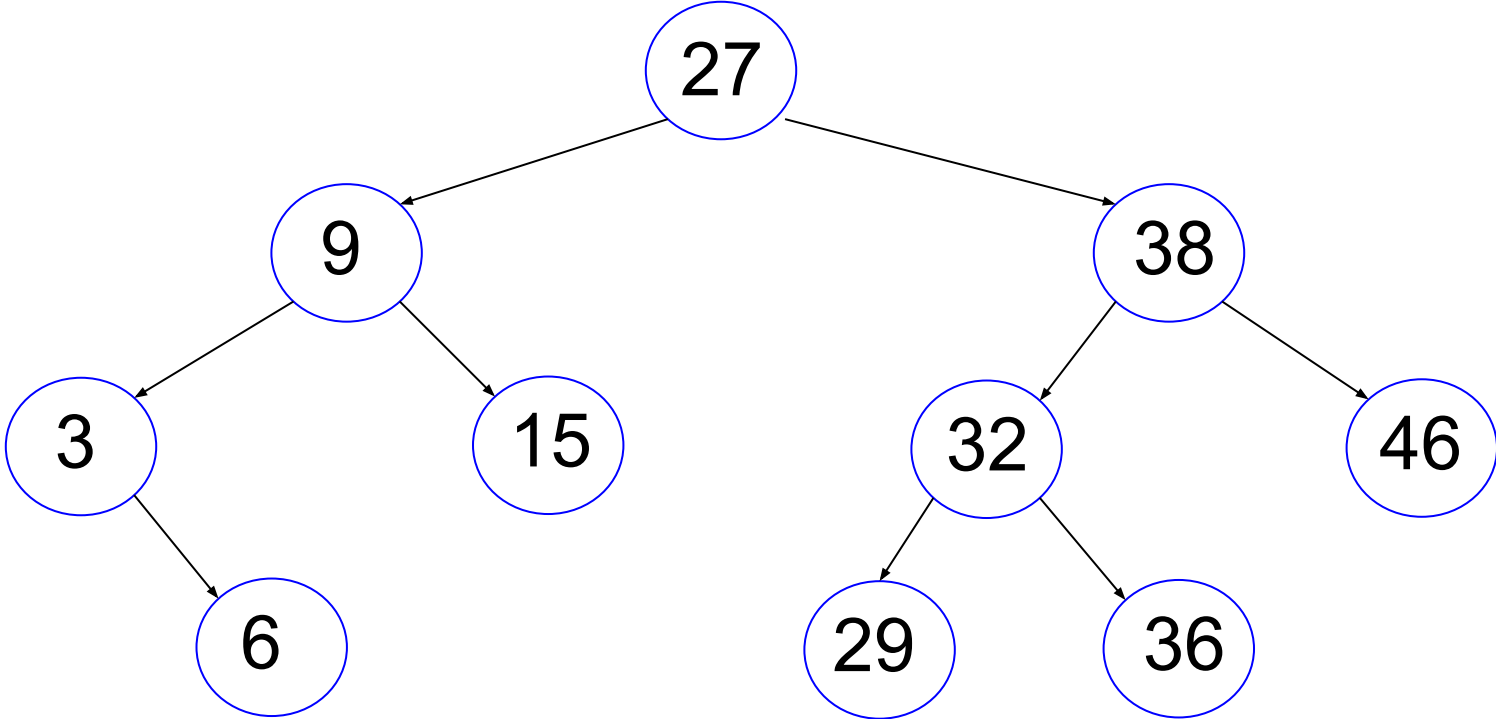
insert 27, 9, 38, 15, 3, 46, 32, 6



insert 27, 9, 38, 15, 3, 46, 32, 6, 29

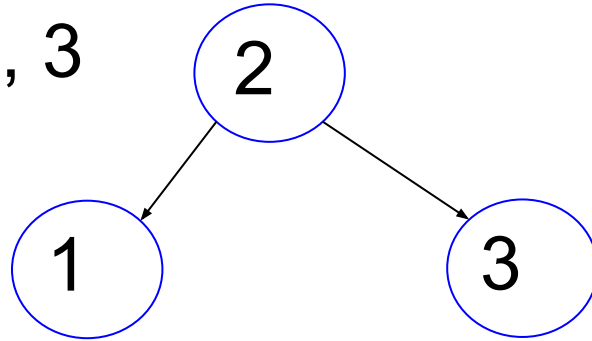


insert 27, 9, 38, 15, 3, 46, 32, 6, 29, 36



Order of insertion may change tree

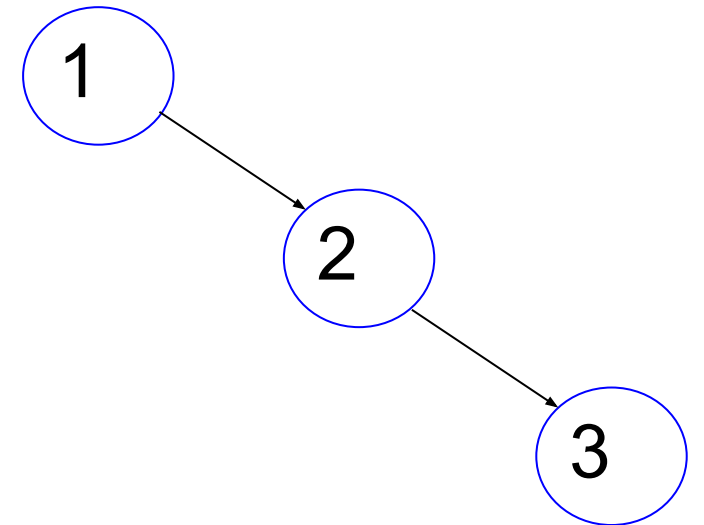
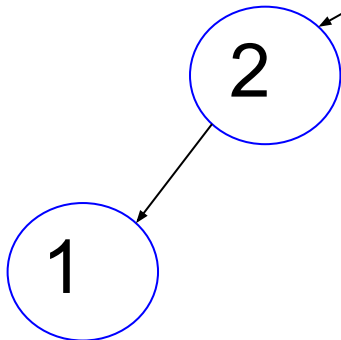
insert 2, 3, 1 or 2, 1, 3



insert 1, 2, 3



insert 3, 2, 1



```
static TreeNode * TreeNode_construct(int val)
{
    TreeNode * tn;
    tn = malloc(sizeof(TreeNode));
    tn -> left = NULL; // remember to initialize
    tn -> right = NULL; // remember to initialize
    tn -> value = val;
    return tn;
}
```

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

TreeNode * root = NULL; // must be initialized to NULL
root = Tree_insert(root, 27)

```

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```



```

TreeNode * root = NULL;
root = Tree_insert(root, 27)

```

Stack Memory			
Frame	Symbol	Address	Value
main	root	100	NULL

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```


```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	27
	tn	200	NULL
	value address 100		
main	root	100	NULL

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
     if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	27
	tn	200	NULL
	value address 100		
main	root	100	NULL

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	27
	tn	200	NULL
	value address 100		
main	root	100	NULL

A70000

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}


TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
main	root	100	A70000


```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
     if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```


TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
     if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    → if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```


TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        {  tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	9
	tn	300	NULL
	value address 70000		
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	70016	27
right	70008	NULL
left	70000	NULL

Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	9
	tn	300	NULL
	value address 70000		
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	9
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

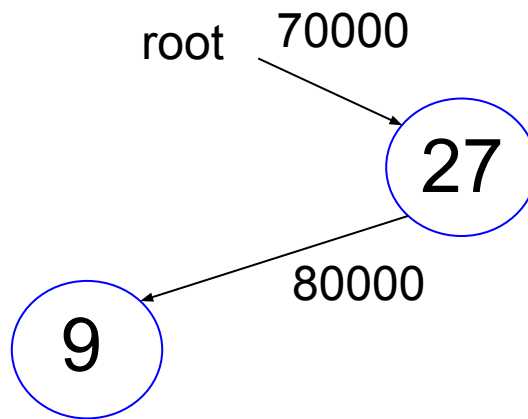
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
main	root	100	A70000

```

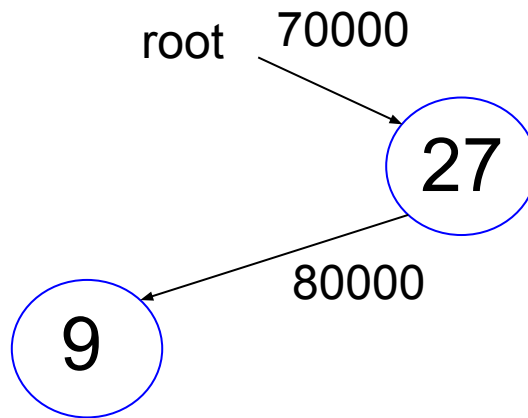
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



yunglu@purdue.edu

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000


```

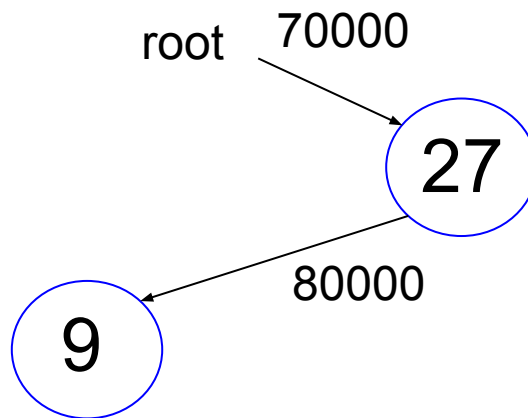
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    → if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

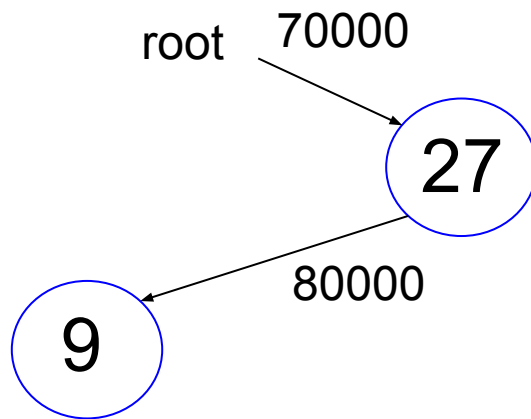
```



```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

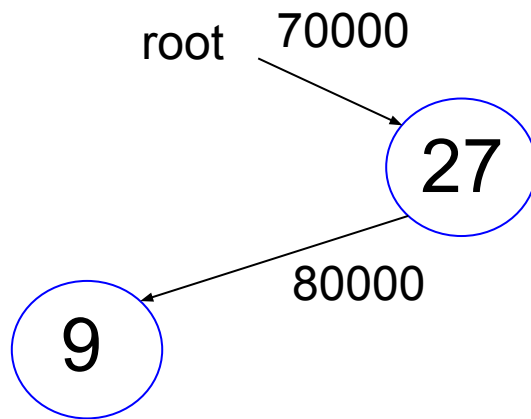
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

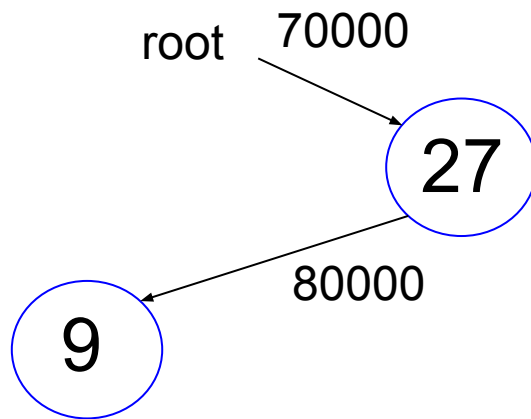
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

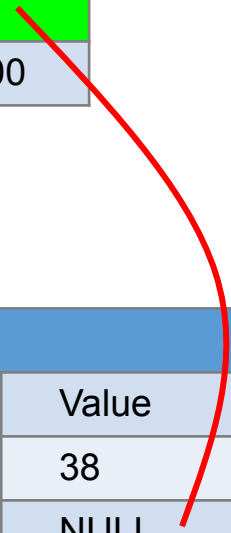
```



yunglu@purdue.edu

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	38
	tn	300	NULL
	value address 70008		
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000



```

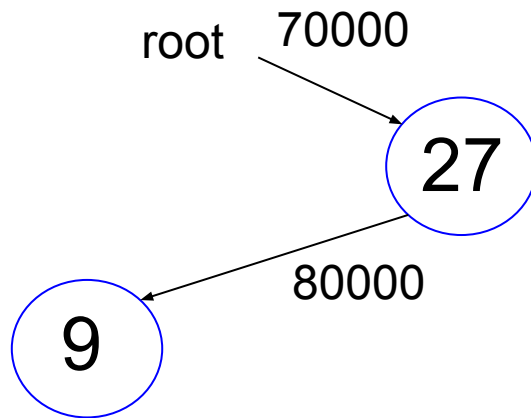
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



yunglu@purdue.edu

Heap Memory		
symbol	Address	Value
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	NULL
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	38
	tn	300	NULL
	value address 70008		
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

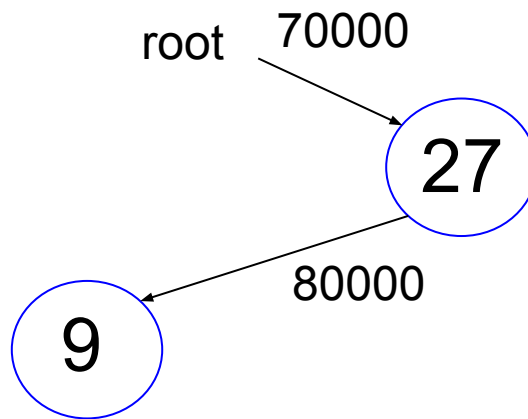
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



yunglu@purdue.edu

Heap Memory		
symbol	Address	Value
value	90016	38
right	90008	NULL
left	90000	NULL
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	A90000
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
insert	val	308	38
	tn	300	NULL
	value address 70008		
insert	val	208	38
	tn	200	A70000
	value address 100		
main	root	100	A70000

```

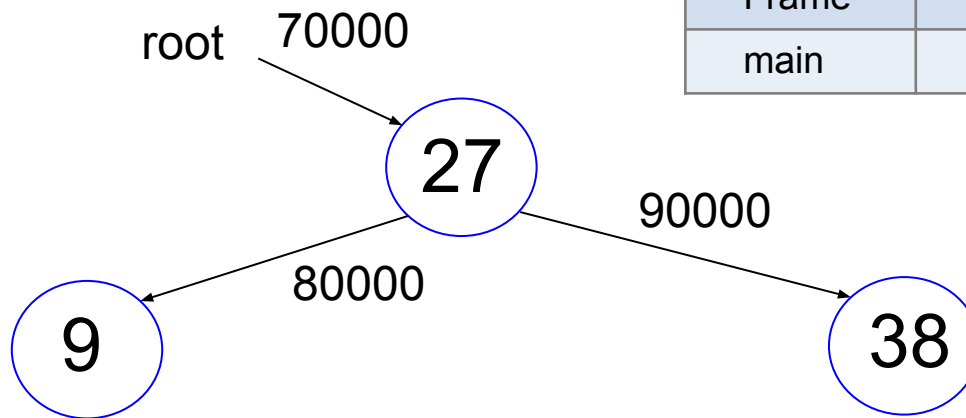
TreeNode * Tree_insert(TreeNode * tn, int val)
{
    if (tn == NULL) // empty, create a node
        { return TreeNode_construct(val); }
    // not empty
    if (val == (tn -> value)) // do not insert the same value
        { return tn; }
    if (val < (tn -> value))
        { tn -> left = Tree_insert(tn -> left, val); }
    else
        { tn -> right = Tree_insert(tn -> right, val); }
    return tn;
}

```

```

TreeNode * root = NULL;
root = Tree_insert(root, 27);
root = Tree_insert(root, 9);
root = Tree_insert(root, 38);

```



yunglu@purdue.edu

Heap Memory		
symbol	Address	Value
value	90016	38
right	90008	NULL
left	90000	NULL
value	80016	9
right	80008	NULL
left	80000	NULL
value	70016	27
right	70008	A90000
left	70000	A80000

Stack Memory			
Frame	Symbol	Address	Value
main	root	100	A70000