# CS 261
# Data Structures

Lecture 13

Binary Search

Binary Trees and Operations

7/18/22, Monday

Oregon State University

# Odds and Ends

- Assignment 3 posted
  - Due Sunday 5/24 11:59 pm (This Sunday!)

- Recitation 5 posted

# Lecture Topics:

- Binary Search
- Binary Trees and their Operations

# Ordered Array

- Note: Binary Search can only work within an ordered (sorted) array
  - The assumption that allows binary search to eliminate half of the array at each iteration


- Make sure the array is sorted before using binary search!
  - Using a sorting algorithm
  - Using binary search

# Ordering Array using a sorting algorithm

- Using a sorting algorithm to order an array

- Runtime complexity of the best general-purpose sorting algorithm
  - O (n log n)
  - Best if we limit the number of times to “sort”

- Examples:
  - Look up in a phone book
  - Look up a word in a dictionary

- What if we expected new elements to be inserted frequently?

# Ordering Array using Binary Search

- If data is frequently changing (i.e., insertion), run binary search after each insertion to maintain an ordered array
  - Recall: `binary_search()` may return the index where an element should be inserted

- Thus, the cost of each insertion:
  - O(log n) to identify the index to be inserted using binary search
  - O(n) to shift the subsequent elements back one spot
  - Since O(n) dominates O(log n), the cost of each insertion is O(n)

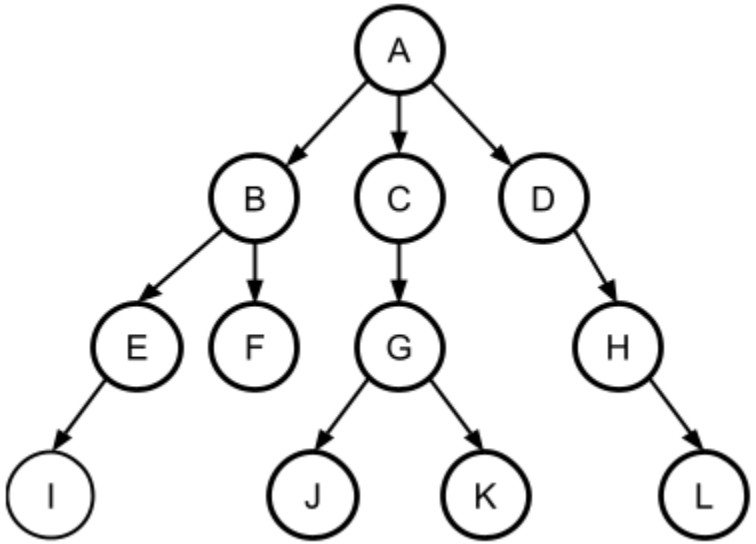
- The total cost of n insertions is $O(n*n) = O(n^2)$

# Lecture Topics:

- Binary Search (cont. )
- Binary Trees

# Trees

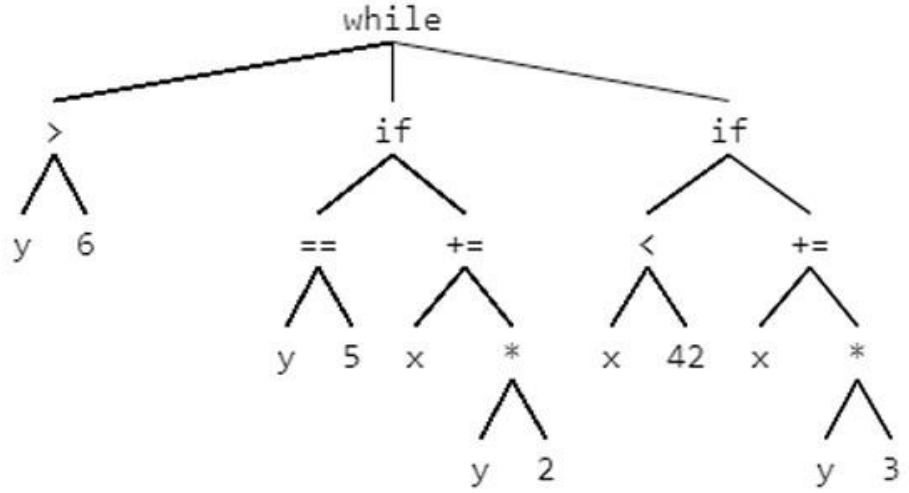
- Tree: non-linear data structure, represents data as a hierarchical structure, encoding the hierarchical relationships between different elements

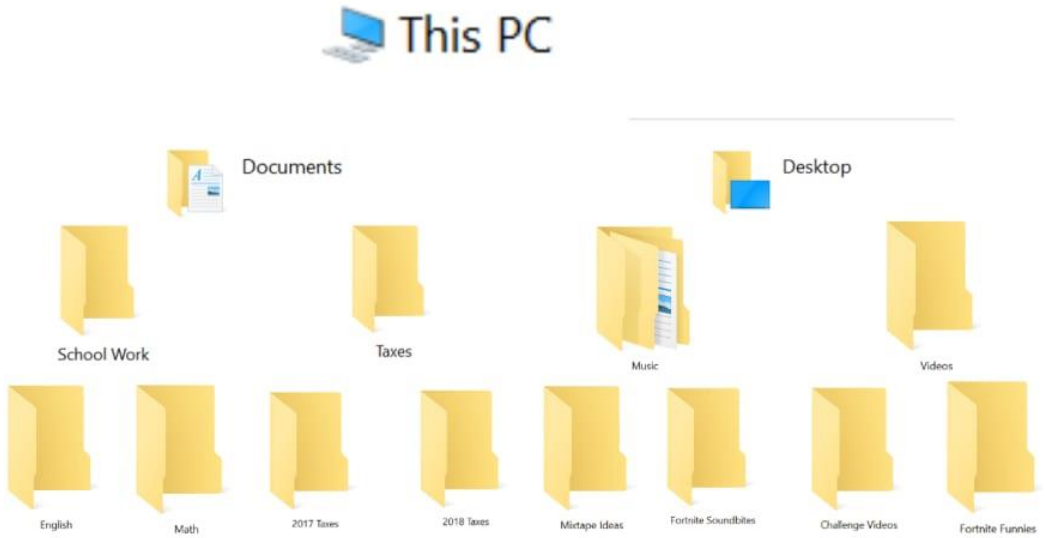


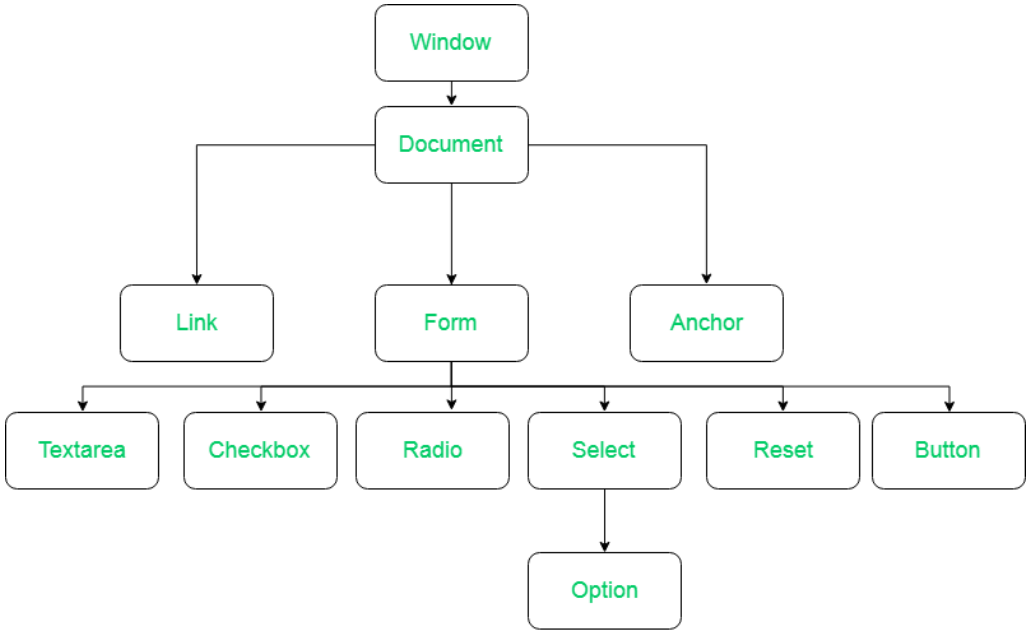

- Node: each individual data element in a tree
  - Contains the data element and points to other nodes
- Edge (arc): an encoded relationship between data elements
  - Represents directed relationships

# Tree Examples

- Examples:
  - Computer's filesystem
  - Object model of a web page
  - Compiler's abstract syntax tree of a program

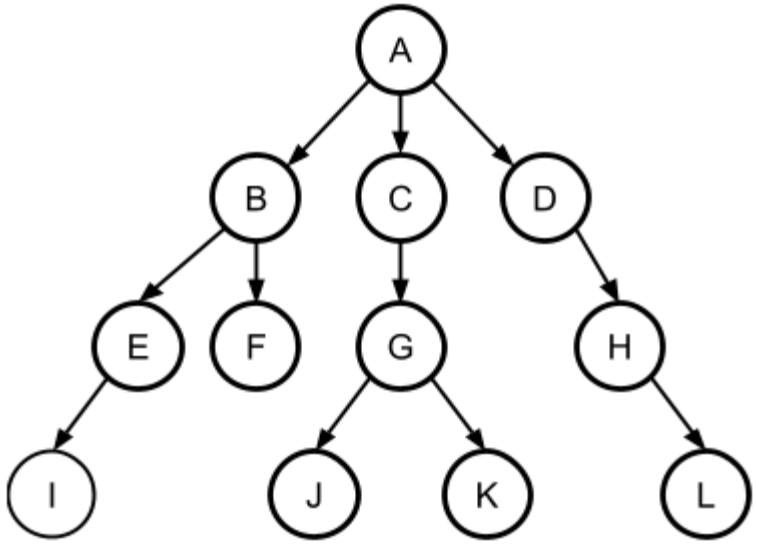


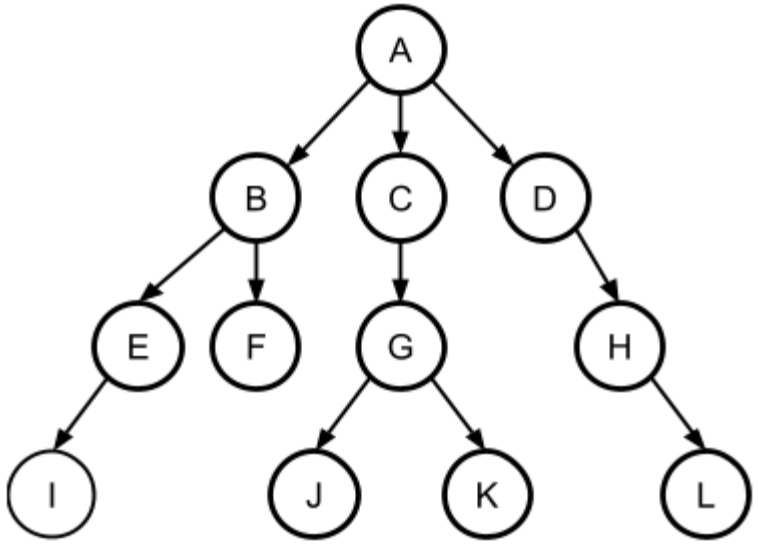
# Trees

- ***Parent***: A node P in a tree is called the parent of another node C if P has an edge that points directly to C.
  - A is parent of B, C, D; B is parent of E and F
- ***Child:*** A node C in a tree is called the child of another node P if P is C's parent.
  - B, C, D are children of A; J, K are children of G
- ***Sibling:*** A node $S_1$ is the sibling of another node $S_2$ if $S_1$ and $S_2$ share the same parent node P
  - B, C, D are siblings; J, K are siblings
- ***Descendant:*** The descendants of a node N are all of N's children, plus its children's children, and so forth.
  - E, F, and I are descendants of node B, and nodes H and L are descendants of node D
- ***Ancestor***: A node A is the ancestor of another node D if D is a descendant of A
  - E, B, and A are ancestors of I, and G, C, and A are ancestors of node K



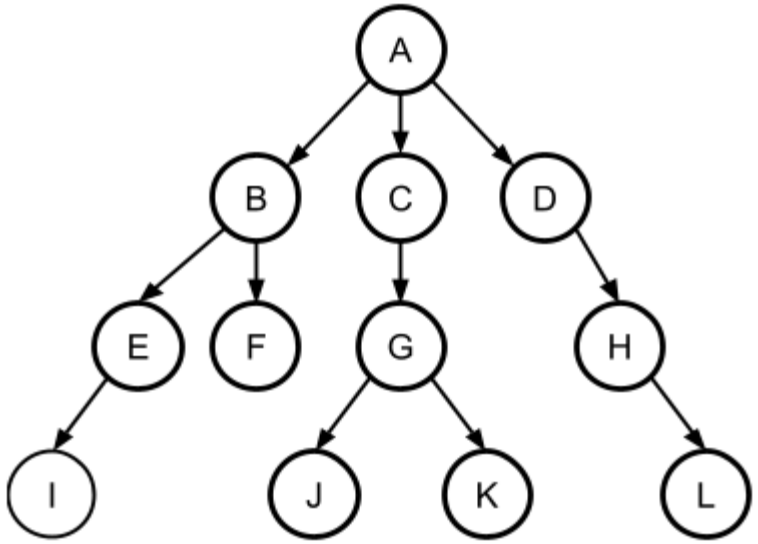
# Trees

- ***Root***: Ancestor of all other nodes in the tree. Each tree has exactly one root.
  - node A is the root.
- ***Interior (node)***: A node has at least one child.
  - A, B, C, D, E, G, and H are interior nodes.
- ***Leaf (node):*** A node has no children.
  - F, I, J, K, and L are leaves.
- ***Subtree***: the portion of a tree that consists of a single node *N*, all of *N*'s descendants, and the edges joining these nodes.
  - the subtree rooted at node B contains the nodes B, E, F, and I and the edges joining those nodes.



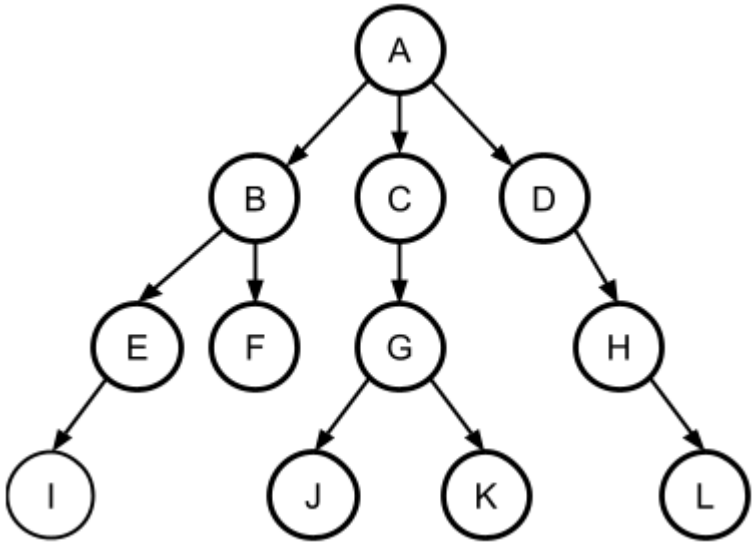
# Trees

- ***Path***: the collection of edges in a tree joining a node to one of its descendants.
- ***Path length:*** the number of edges in that path.
  - the path from C to K has length 2, since it contains 2 edges.
- ***Depth:*** The depth of a node *N* in a tree is the length of the path from the root to *N*.
  - the depth of K is 3.
  - The depth of A (root) is 0.
- ***Height:*** The maximum depth of any node in the tree.
  - The tree has height 3



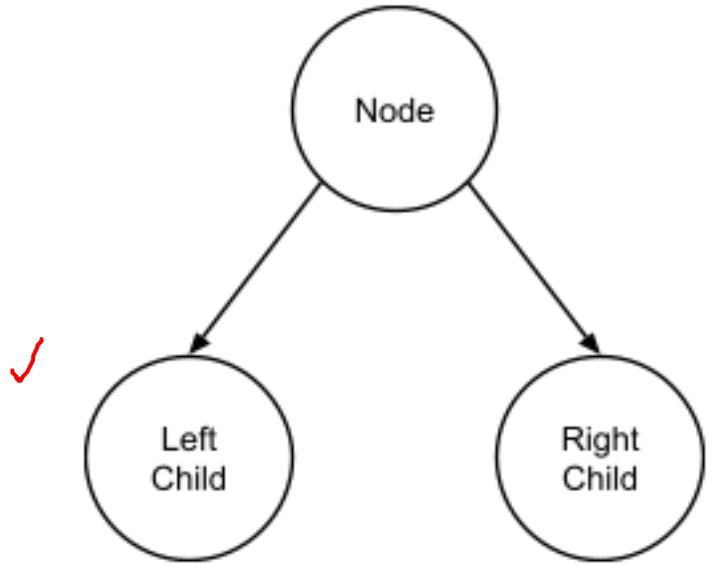
# Trees

- Constraints to be counted as a tree:
  - Each node in the structure may have only one parent.
  - The edges of the structure many not form any cycles.
    - there cannot be a path from any node to itself.



# Binary Trees

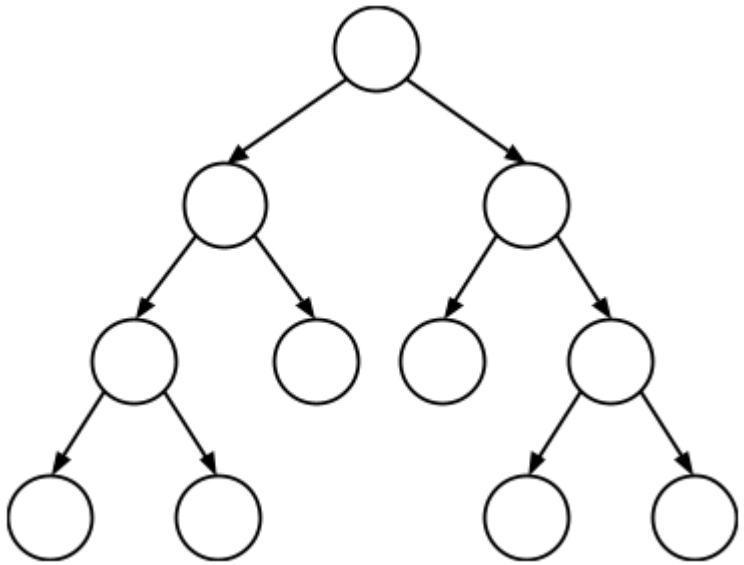
- *Binary Tree*: a tree in which each node can have at most two children (left child and right child).

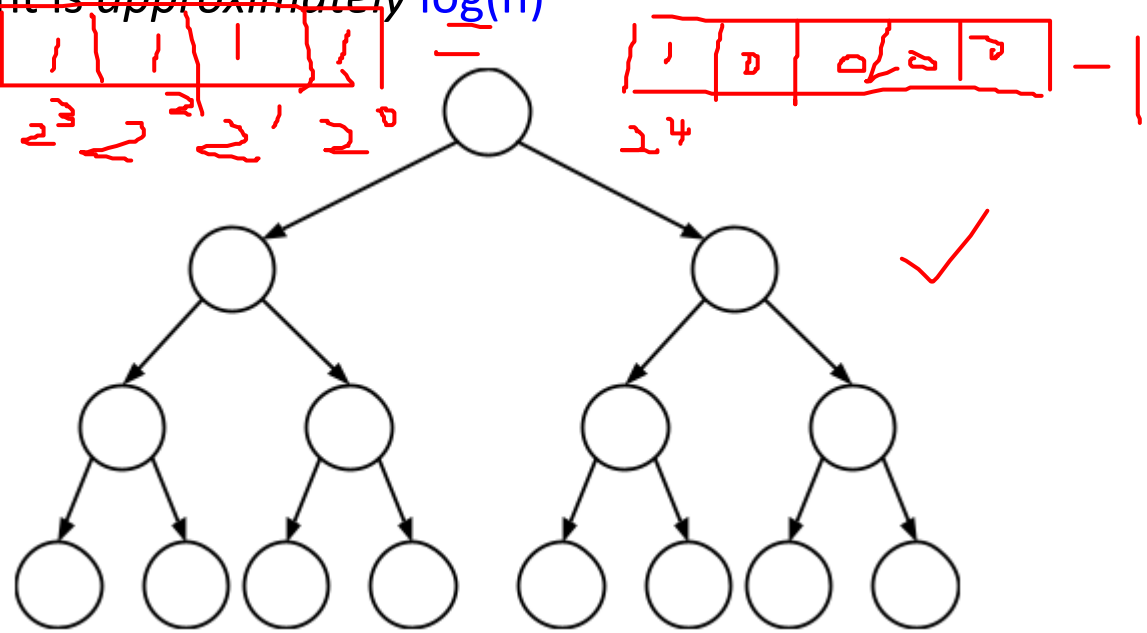



- *Left subtree*: the subtree rooted at that node's left child
- *Right subtree*: the subtree rooted at that node's right child

# Binary Trees

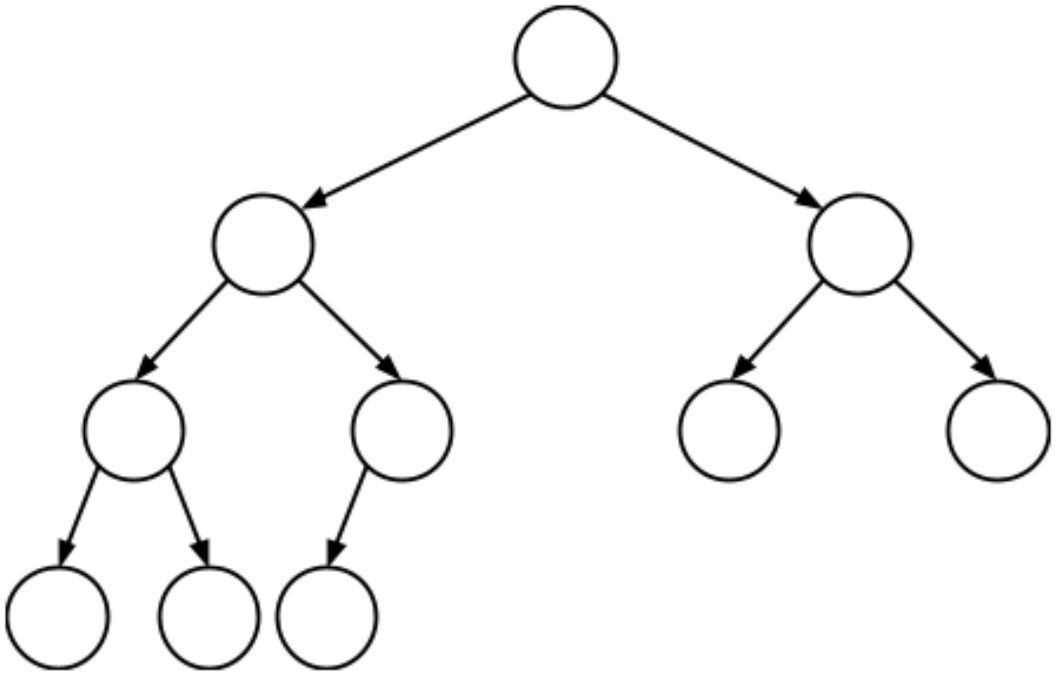
- *Full Binary Tree*: a binary tree that every interior node has exactly two children.
- *Perfect Binary Tree*: a full binary tree where all the leaves are at the same depth.
  - If a perfect binary tree has height h, then
    - It has $2^h$ leaves
    - It has $2^{h+1} - 1$ total nodes
  - If a perfect binary tree has n nodes, then its height is *approximately* log(n)





# Binary Trees

- *Complete Binary Tree*: a binary tree that is perfect except for the deepest level, whose nodes are all as far left as possible



# Binary Search Trees

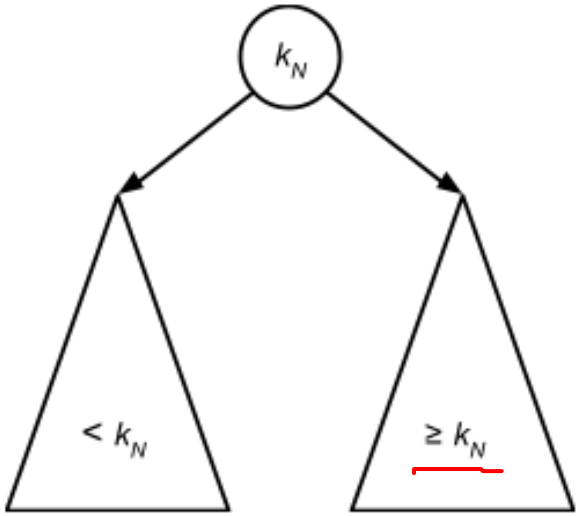
- Recall: each node in a tree represents a data element.

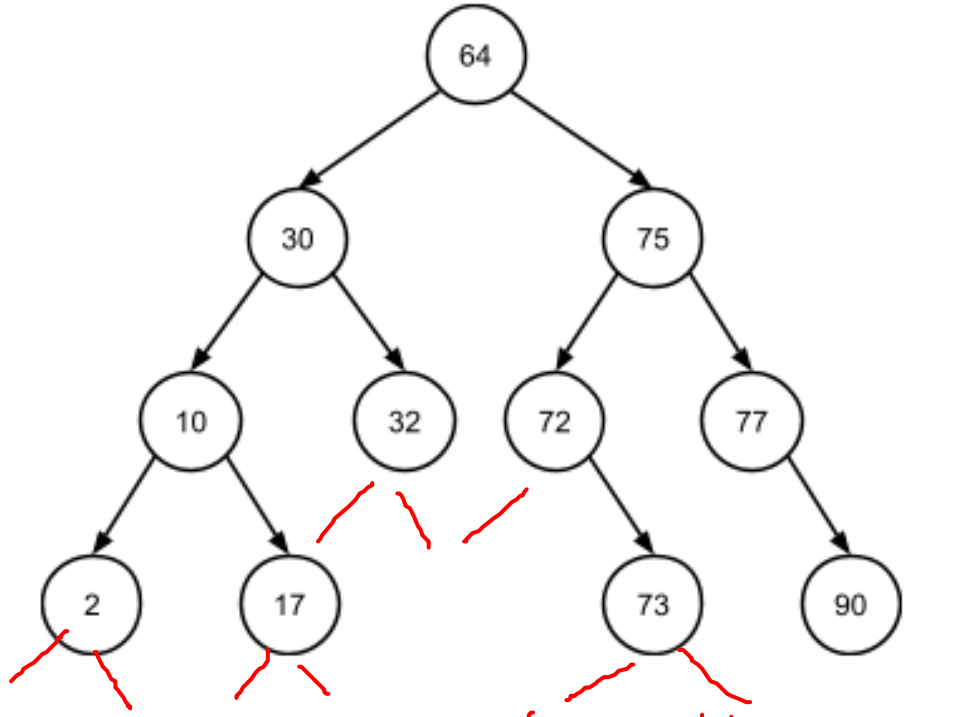
- Represent each data element using a **key** (identifier)
  - The data element may also contain other data, which we can refer to as its **value**

- Assuming these keys can be ordered in relation to others
  - i.e., integer keys can be ordered numerically, string keys can be ordered alphabetically

# Binary Search Trees

- A *binary search tree* (**BST**) is a binary tree that:
  - *the key of each node N is **greater than** all the keys in N's left subtree and **less than or equal to** all the keys in N's right subtree*




- **Note: A BST does NOT have to be full, perfect, complete, etc.*

# BST Operations

- *Remember:*
  - when a given node does not have a subtree on either the left or right side, the node's child on that side will be NULL.
  - a leaf node in a BST is one where both the left and right child are NULL.

# BST Operations: Finding an element

- Elements in a BST are located based on their keys
  - When a user wants to locate an element, they will need to provide the key of the element

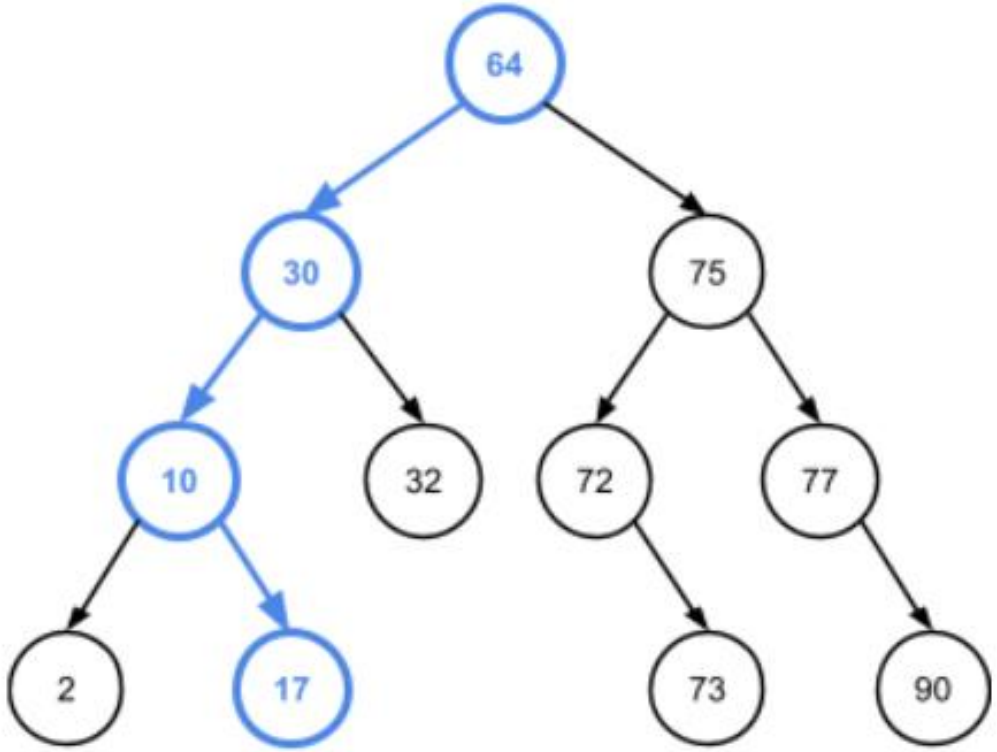
- How does it work?
  - Keep a pointer to the current node N, starting at the root. Examining one node at a time
  - If N is NULL, the key $k_q$ doesn't exist in the tree, and the search has failed. Break.
  - If N's key is equal to $k_q$, the search has succeeded. Break.
  - If $k_q$ is less than the N's key, move the current node to point to its *left* child and repeat.
  - If $k_q$ is greater than N's key , move the current node to point to its *right* child and repeat.

# BST Operations: Finding an element

- Pseudocode: iteration

```
find(bst, k_q){
    N = bst.root
    while N is not NULL{
        if N.key equals k_q
            return success
        else if k_q < N.key
            N = N.left
        else:
            N = n.right
    }
    return failure
}
```

- Example: search for key 17



# BST Operations: Inserting a new element

- New elements are always inserted into a BST as leaves.
  - avoid to restructure the tree

- Key: find the location for the new element that maintains the BST property at all nodes in the tree.

- find the location → using search/find function!
  - Instead of stopping the search if/when *k* is found in the tree, insertion always proceeds until reaching a NULL node
  - The location of this NULL node, then, is the location at which to insert the new node
  - The new node will become the child of the NULL node's parent

# BST Operations: Inserting a new element

- Pseudocode:

```
insert(bst, k, v){
    P = NULL
    N = bst.root
    while N is not NULL{
            P = N
            if k < N.key:
                    N = N.left
            else:
                    N = N.right
    }
    create a new node as the child of P containing k, v
}
```

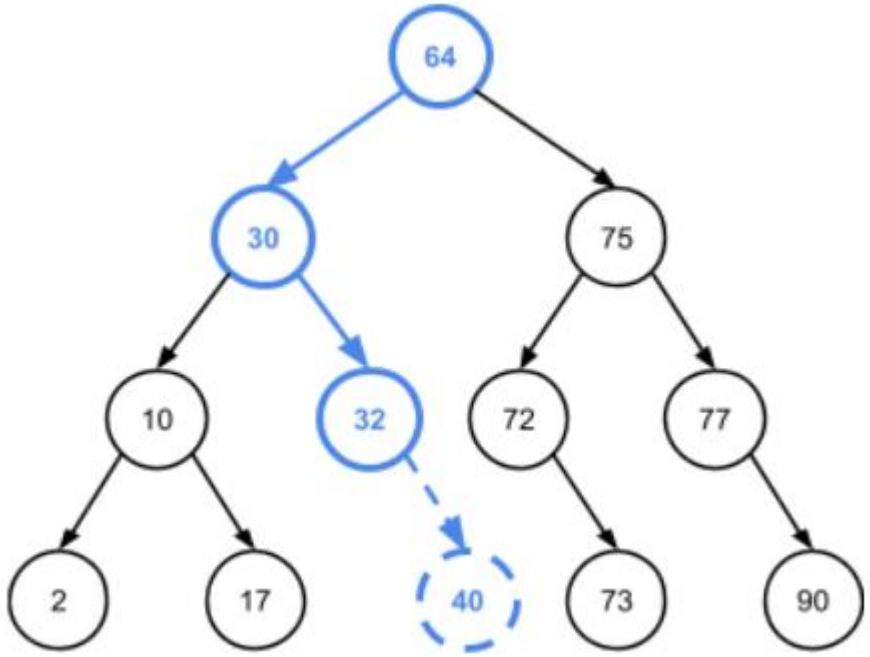
- **P** is used to track the location of the new node's parent
- if **P** is NULL at the end of the search here, then the BST is empty, and the new node should be inserted as the root of the tree
- If **P** is not NULL, then the new node will be inserted as either the left or right child of **P**, depending on whether **k** is less than or greater than (or equal to) **P**'s key

# BST Operations: Inserting a new element

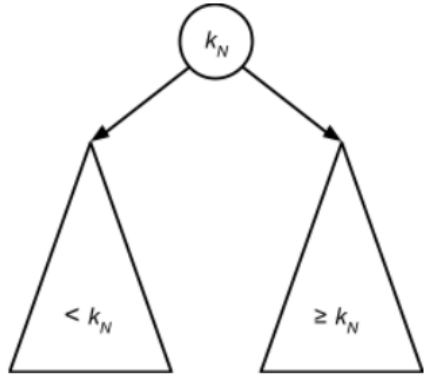
- Pseudocode:

```
insert(bst, k, v){
    P = NULL
    N = bst.root
    while N is not NULL{
            P = N
            if k < N.key:
                    N = N.left
            else:
                    N = N.right
    }
    create a new node as the child
    of P containing k, v
}
```

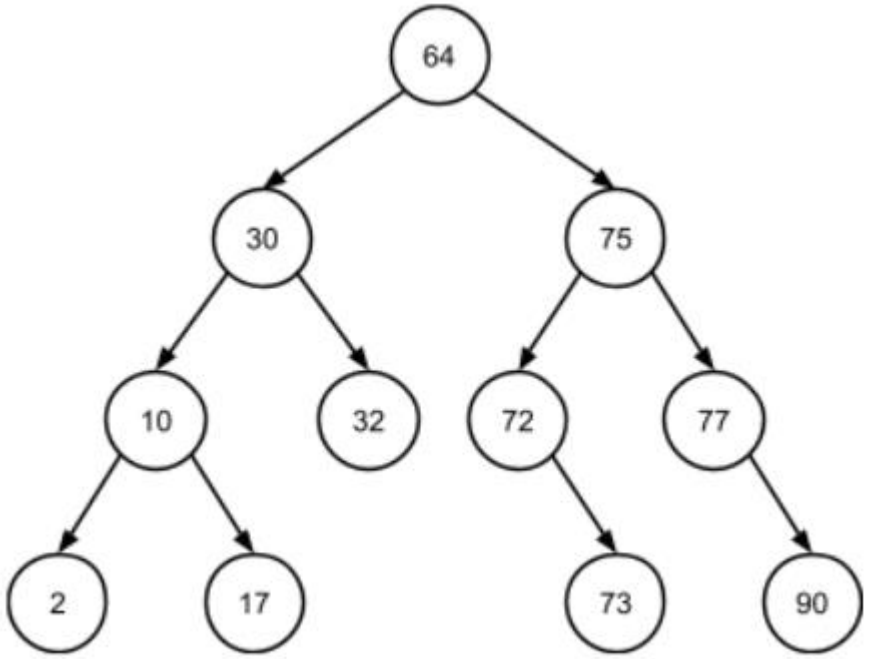
- Example: insert the key 40

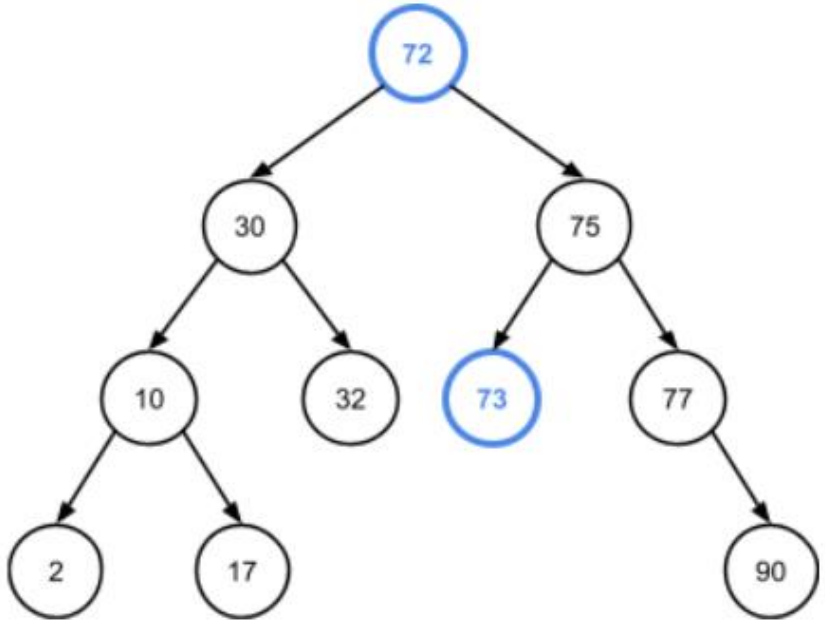


# BST Operations: Removing an element



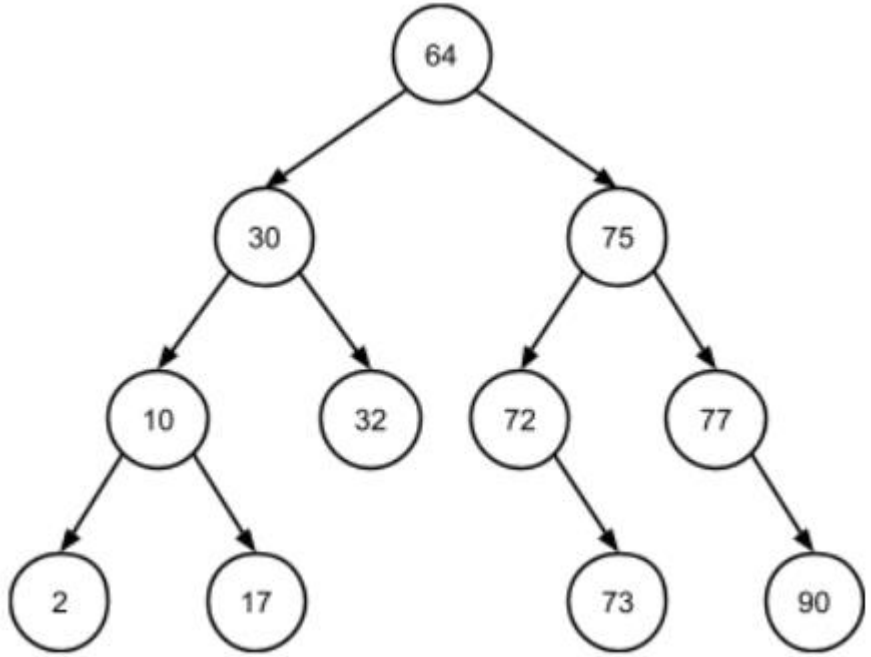

- How to remove the element with a key 2?
  - Easy! Simply remove it since it is a leaf node
- How to remove the element with a key 64?
  - Umm, then which node should be our new root, so it maintains BST after removal?



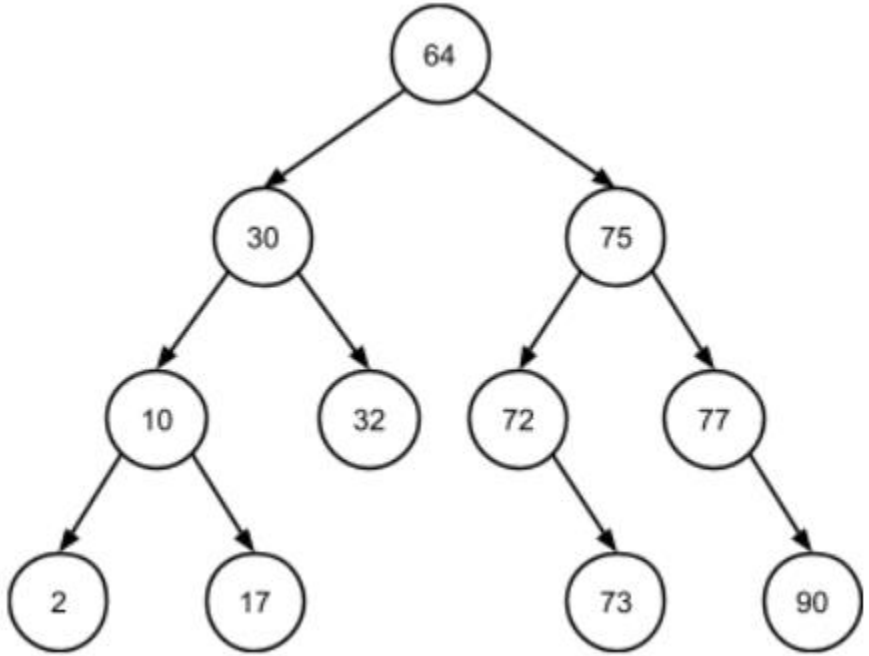
# BST Operations: Removing an element

- BST removal: depend on the number of children that element's BST node has
- If the element to be removed is a leaf node: (i.e., 2)
  - simply free that node and update its parent to have a NULL child
- If the element to be removed is stored in a node with just a single child: (i.e., 72)
  - simply free that node and move its child to become a child of the node's parent



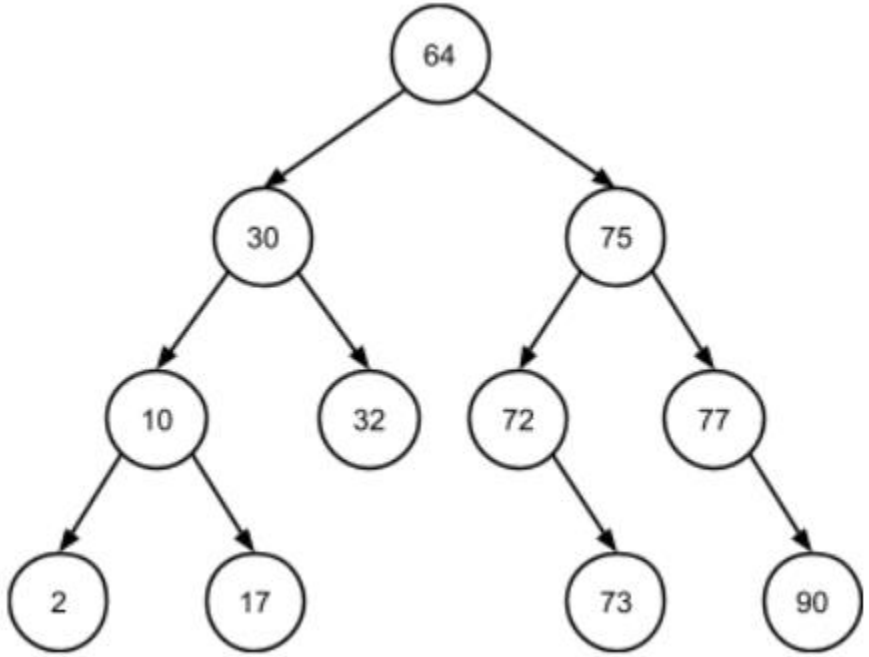
# BST Operations: Removing an element

- If the element to be removed is stored in a node with two children: (i.e., 64):
  - need to find that node's ***in-order successor*** (the next node in in-order traversal of the BST).
  - Line up all keys in ascending order:
  - 2 10 17 30 32 64 72 73 75 77 90
  - The in-order successor for a node with key k, is the node to the very next key after k in this ordered list of keys
    - i.e., the in-order successor of root (64) is the node with key 72



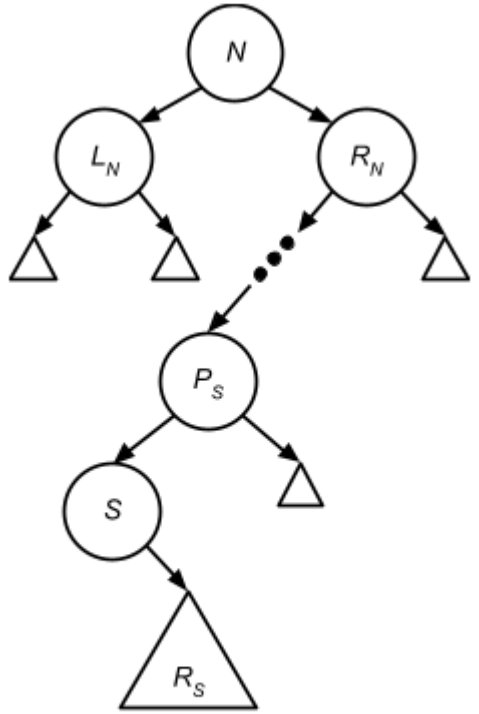
# BST Operations: Removing an element

- If the element to be removed is stored in a node with two children: (i.e., 64):
  - In BST, a node N's in-order successor is always **the leftmost node in N's right subtree**.
    - branch right in the tree from N, and then continue to branch left until we can no longer do so, The last node we reach will be N's in-order successor



# BST Operations: Removing an element

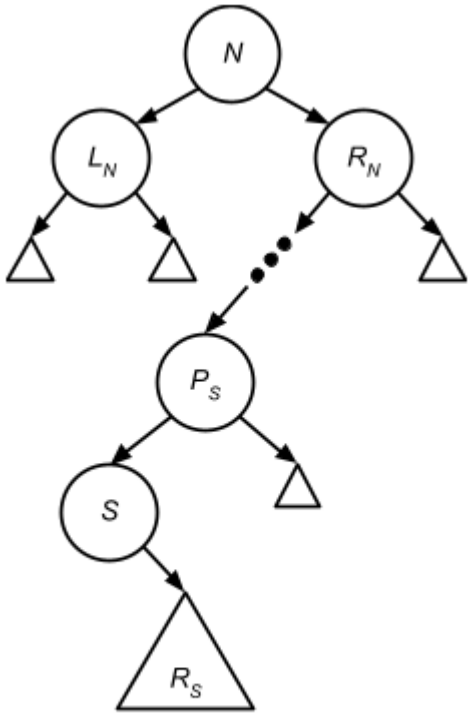
- If the element to be removed is stored in a node with two children: (i.e., 64):
  - *Denote N's parent node as $P_N$ (if N is the root node, $P_N$ will represent the root pointer for the entire tree)*
  - *Find N's in-order successor S. Denote S's parent node as $P_S$.*
  - *Update pointers to give N's children to S*
    - *N's left child becomes S's left child.*
    - *S's right child (which might be NULL) becomes $P_S$'s left child.*
    - *N's right child becomes S's right child.*
    - *Update $P_N$ to replace N with S.*
      - *Specifically, S becomes $P_N$'s left or right child, as appropriate, or the root of the tree, if N was the root.*
  - *Free the node N.*



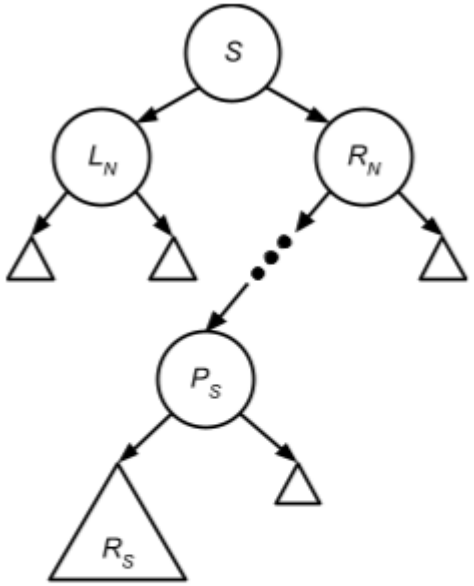

Before removing *N*

# BST Operations: Removing an element

- If the element to be removed is stored in a node with two children: (i.e., 64):




Before removing *N*




After removing *N*

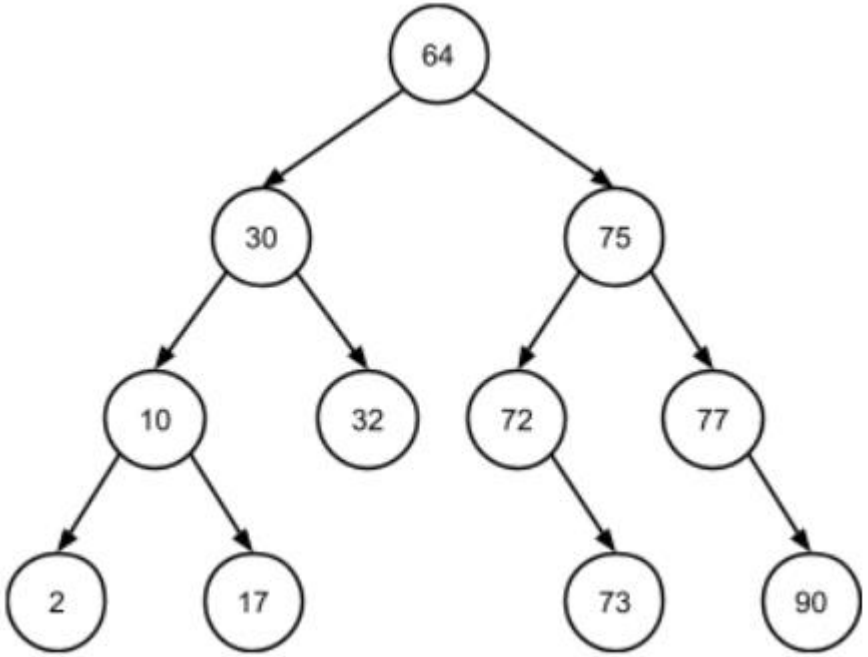
# BST Operations: Removing an element

- Pseudocode:

```
remove(bst, k):
        N, P_N ← find the node to be removed and its parent
                  based on key k, as in the find() function
        if N has no children:
                update P_N to point to NULL instead of N
        else if N has one child:
                update P_N to point to N's child instead of N
        else:
                S, P_S ← find N's in-order successor and its
                          parent, as described above
                S.left ← N.left
                if S is not N.right:
                        P_S.left ← S.right
                        S.right ← N.right
                update P_N to point to S instead of N
        free N
```

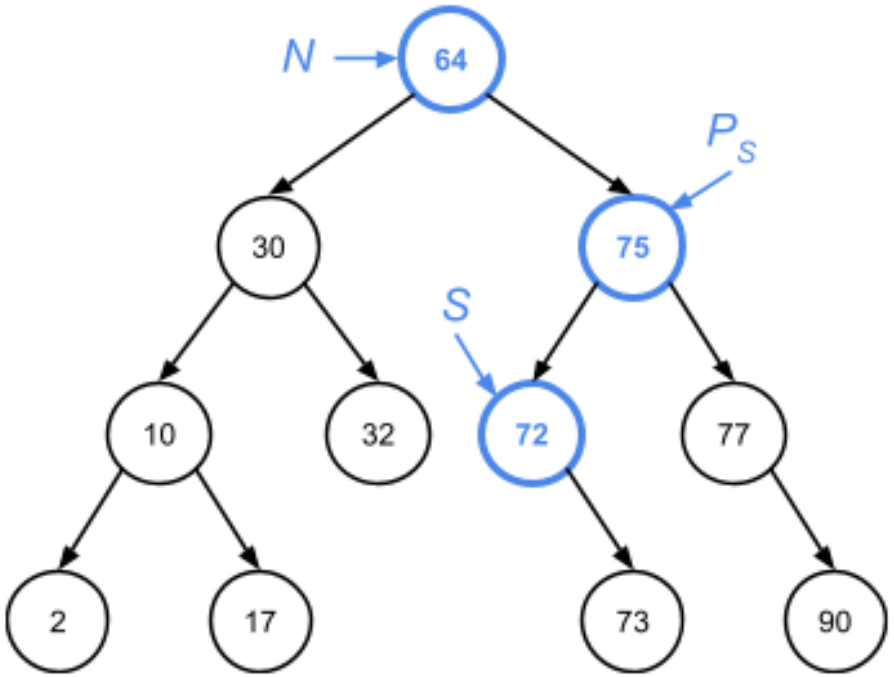
# BST Operations: Removing an element

- Example: Remove the root node (64)



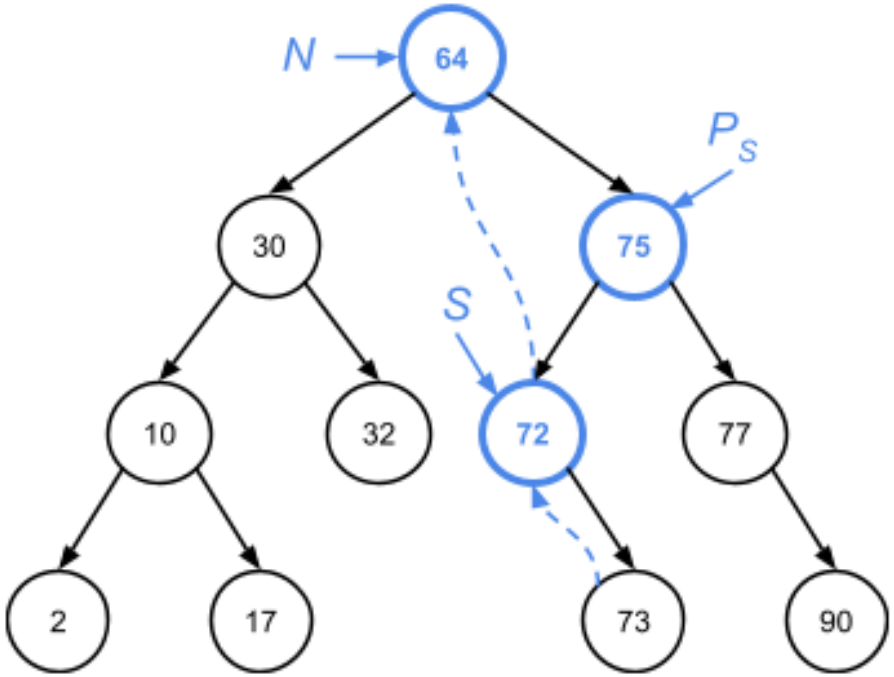
# BST Operations: Removing an element

- Example: Remove the root node (64)
  - 1. identify that node's in-order successor (S) and its parent ($P_S$):



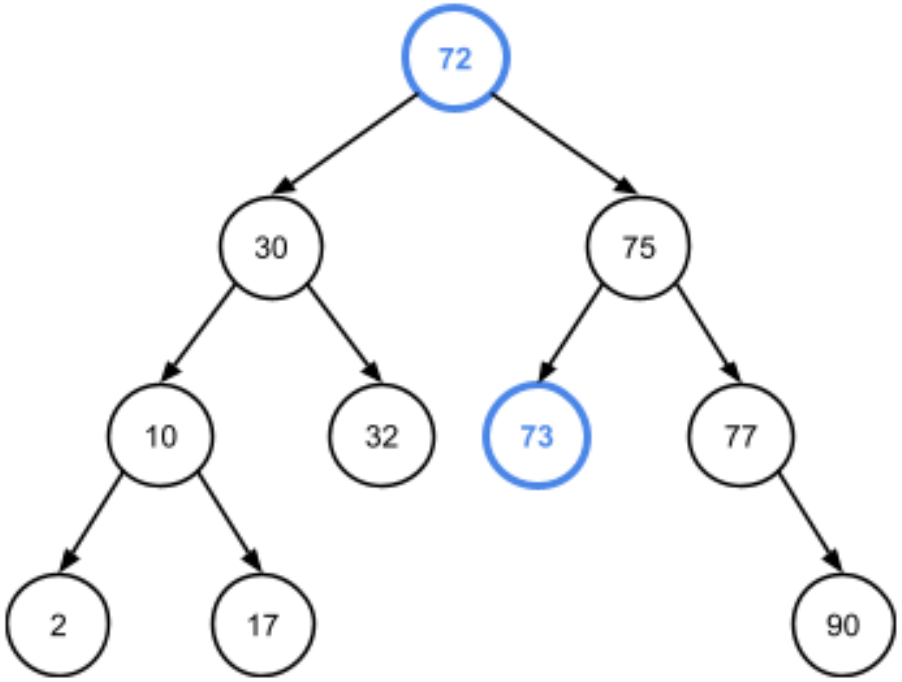
# BST Operations: Removing an element

- Example: Remove the root node (64)
  - 2. update pointers so that *S* replaces *N* and *S*'s right child replaces *S* as $P_S$'s child:



# BST Operations: Removing an element

- Example: Remove the root node (64)
  - 3. The end result is a tree with the root node (i.e. N) removed.




  - note that the BST property is maintained by this removal: