CS 261-020
Data Structures

Lecture 9

Midterm Report

Binary Trees

2/15/24, Thursday

Oregon State University

Odds and Ends

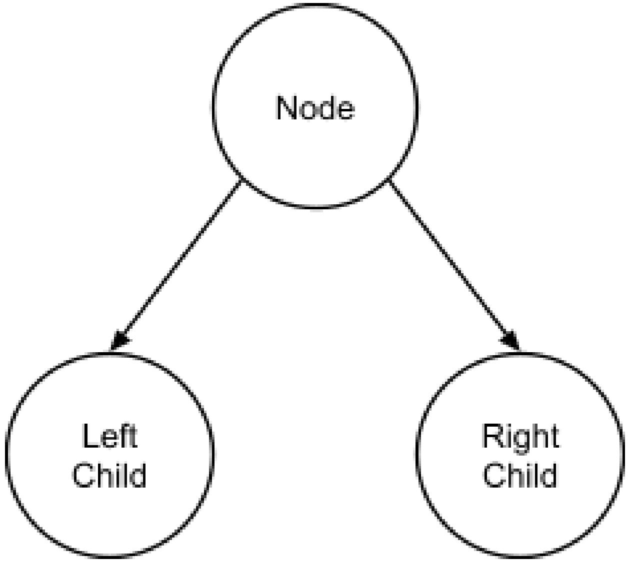
- Assignment 3 posted
- No quiz this week
- Don’t forget to demo your assignment 2!

Lecture Topics:

- Midterm Report
- Binary Trees

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

Lecture Topics:

- BST Operations:
  - Finding an element
  - Inserting a new element
  - Removing an element

- Runtime Complexity of BST operations
- BST traversals

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

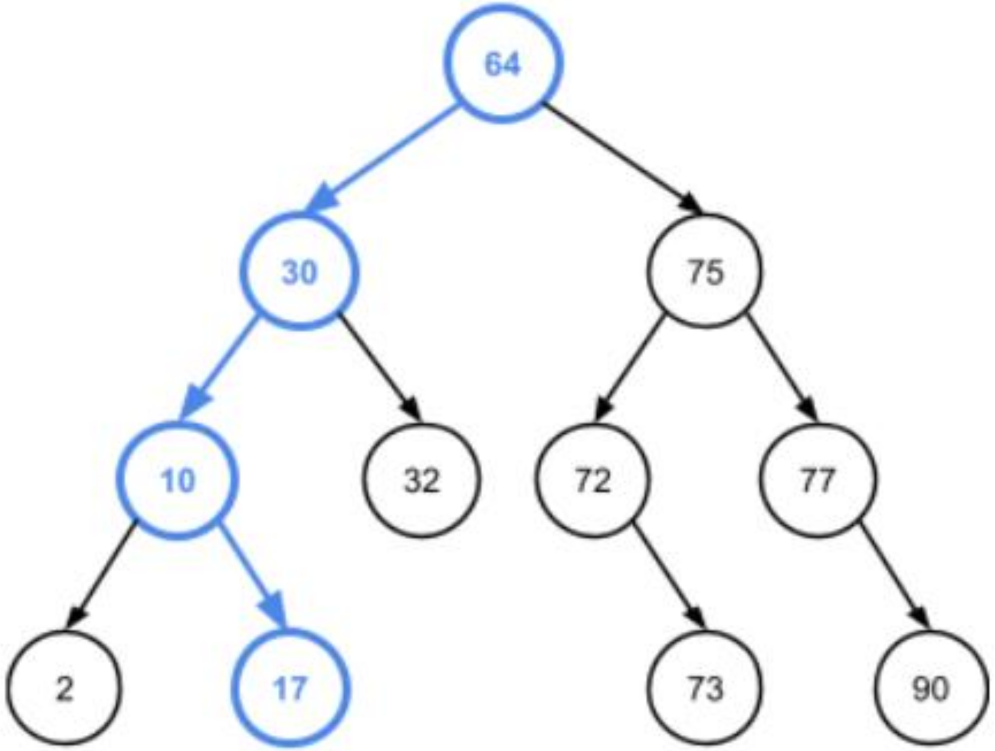
- How does it work?
  - Keep a pointer to the current node N, starting at the root. Examining one node at a time
  - If N is NULL, the key k_q doesn't exist in the tree, and the search has failed. Break.
  - If N's key is equal to k_q, the search has succeeded. Break.
  - If k_q is less than the N's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

- Hint: 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
}
```

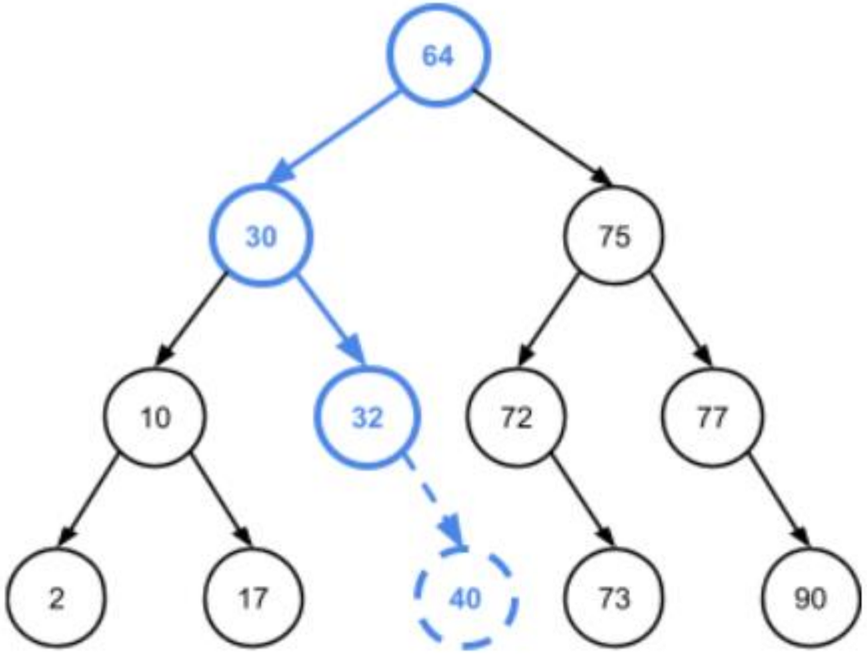
- **P** is used to track the location of the new node's parent
- if **P** is NULL at the end of the search here, then the BST is empty, and the new node should be inserted as the root of the tree
- If **P** is not NULL, then the new node will be inserted as either the left or right child of **P**, depending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

BST Operations: Inserting a new element

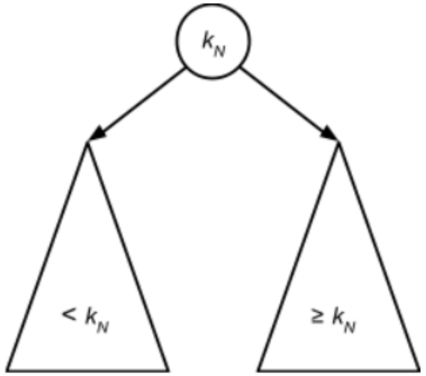
- Pseudocode:

```
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
}
```

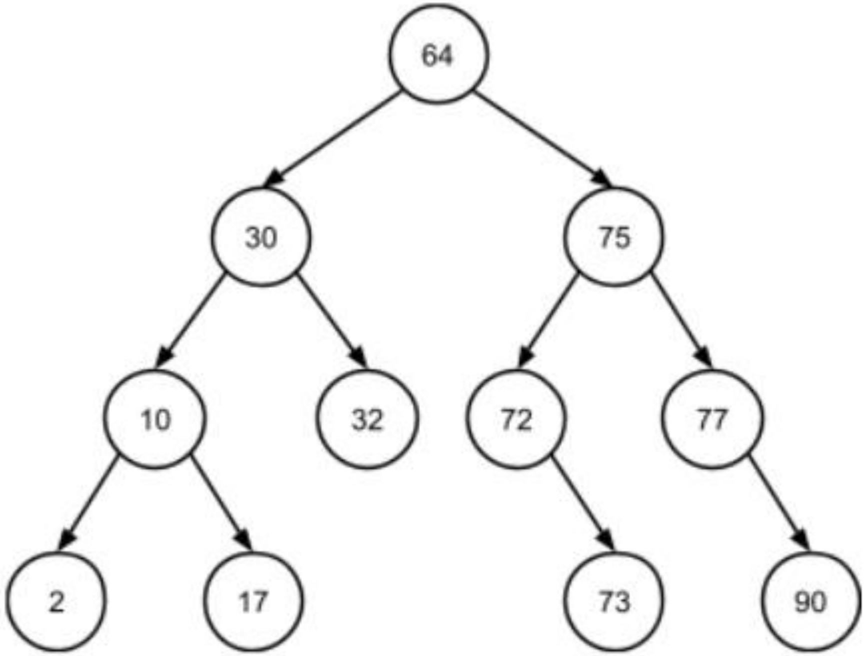
- Example: insert the key 40

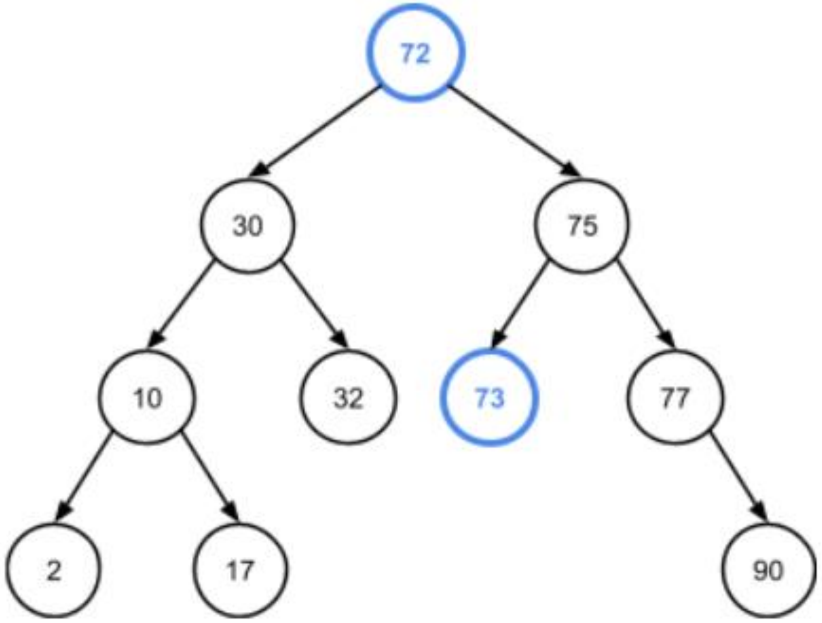


BST Operations: Removing an element



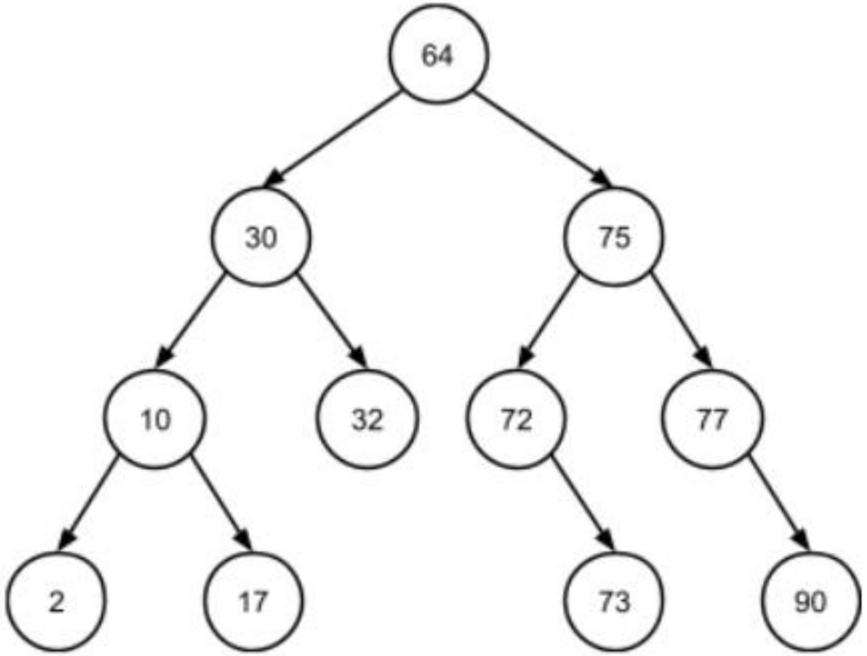
- How to remove the element with a key 2?
  - Easy! Simply remove it since it is a leaf node
- How to remove the element with a key 64?
  - Umm, then which node should be our new root, so it maintains BST after removal?





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

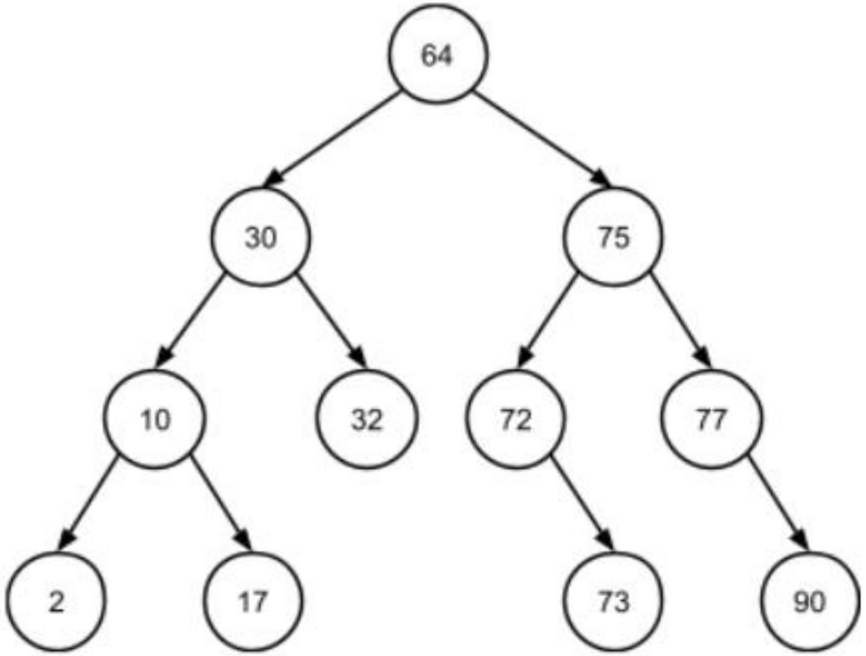


BST Operations: Removing an element

- 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).

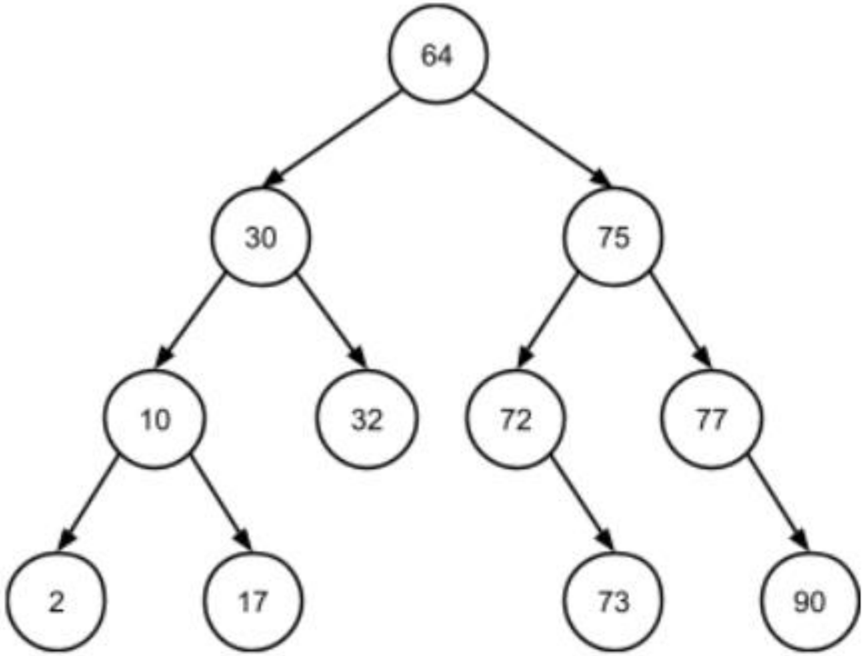
  - Line up all keys in ascending order:
  - 2 10 17 30 32 64 72 73 75 77 90

  - 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



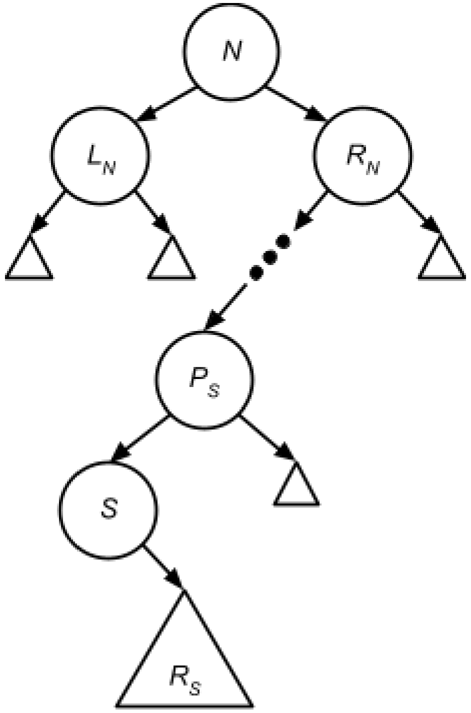
BST Operations: Removing an element

- 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

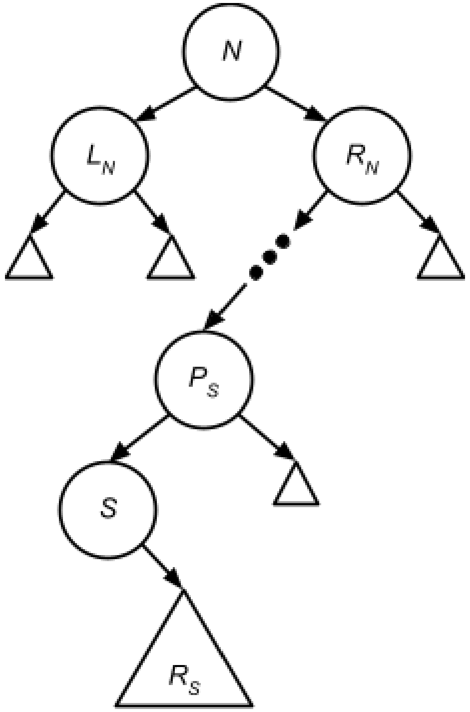
- If the element to be removed is stored in a node with two children: (i.e., 64):
  - *Denote N's parent node as P_N (if N is the root node, P_N will represent the root pointer for the entire tree)*
  - *Find N's in-order successor S. Denote S's parent node as P_S.*
  - *Update pointers to give N's children 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.*



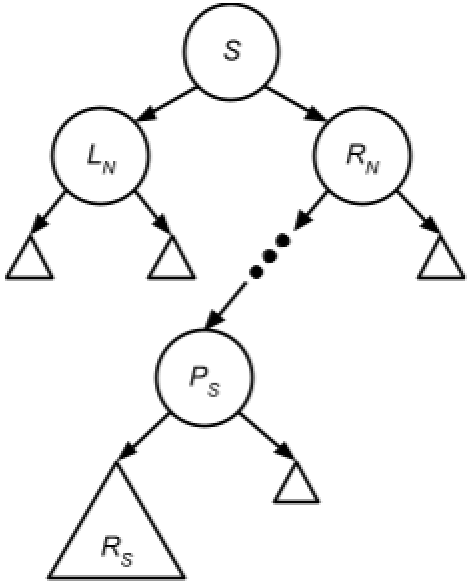
Before removing *N*

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*

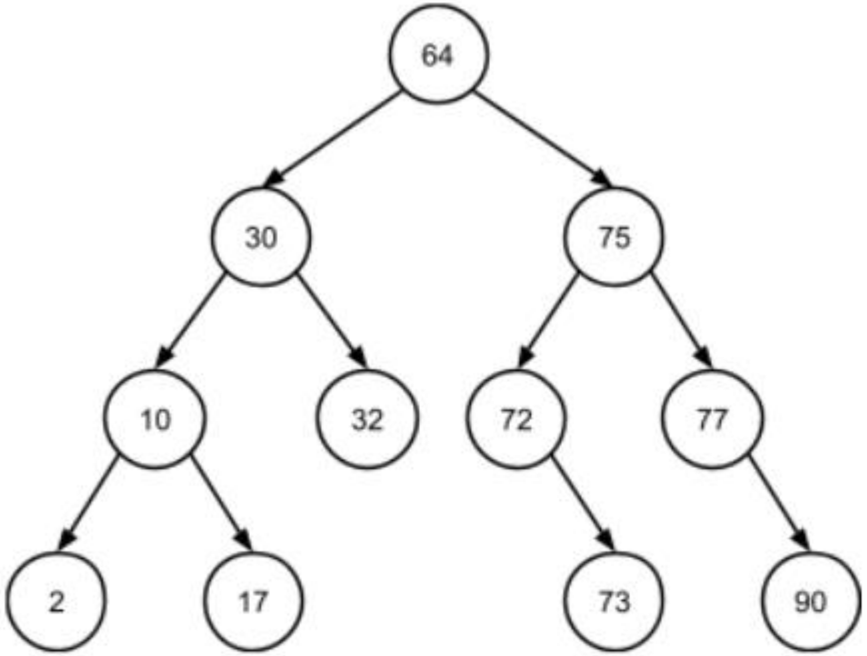
BST Operations: Removing an element

- 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
```

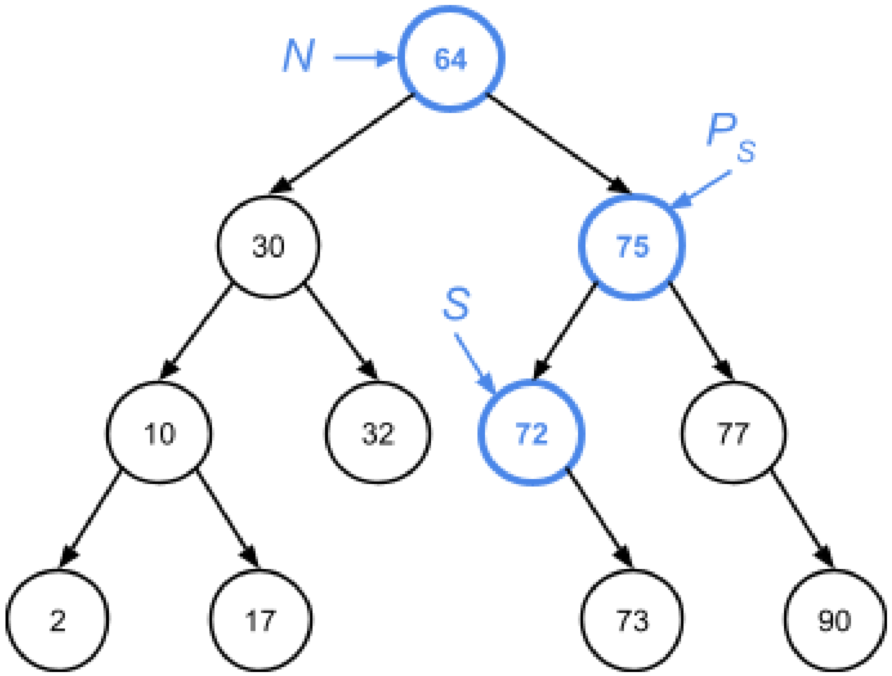
BST Operations: Removing an element

- Example: Remove the root node (64)



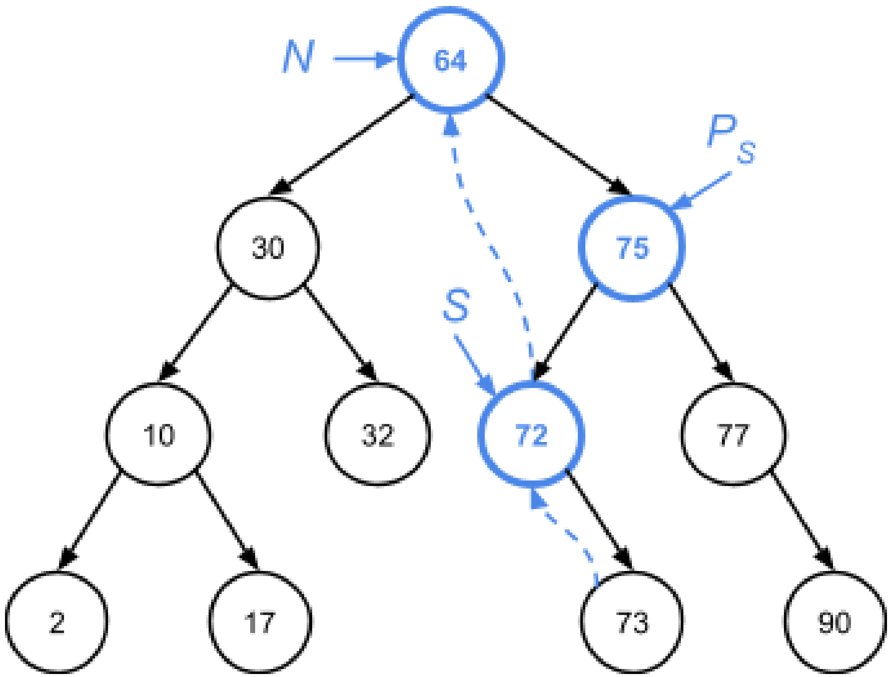
BST Operations: Removing an element

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



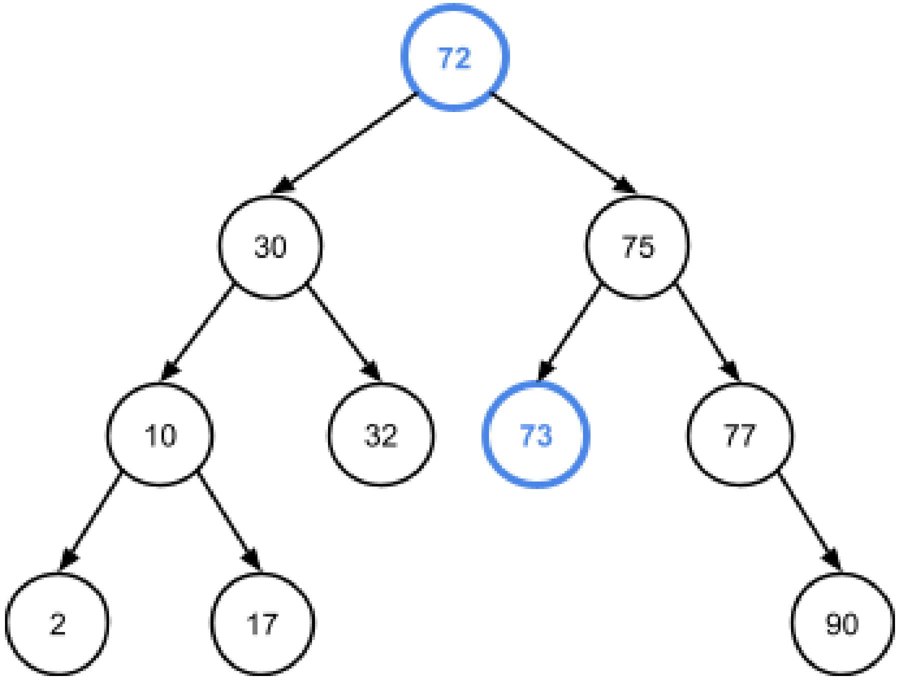
BST Operations: Removing an element

- 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:

Lecture Topics:

- BST Operations:
  - Finding an element
  - Inserting a new element
  - Removing an element

- Runtime Complexity of BST operations
- BST traversals

Runtime Complexity of BST Operations

- Main factor of all 3 BST operations: search within the tree
  - find(): search for the query key
  - insert(): search for the location at which to insert
  - remove(): search for both query key and its in-order successor

- Search begins at the root, moves down one level at each iteration, until reaches the bottom (or finds the node it is searching for)
  - Number of search iteration == the height of the tree, h

- Thus, runtime complexity for searching in all 3 operations: O(h)

Runtime Complexity of BST Operations

- Extra work done besides searching:
  - find(): none
  - insert(): allocate the new node, and update its new parent → O(1)
  - remove(): update a few pointers → O(1)

- Thus, the runtime complexity:
  - find() – O(h)
  - insert() – O(h)
  - remove() – O(h)

- What is the range of h if the BST has n nodes?
  - Depending on the order of insertion, h can be [log(n), n]

→ limit the height of the BST! (more later)

Lecture Topics:

- BST Operations:
  - Finding an element
  - Inserting a new element
  - Removing an element

- Runtime Complexity of BST operations
- BST traversals

Binary Tree Traversal

- How to print the value stored at each node in a binary tree?

- **A tree traversal**: a method for visiting each node in a tree exactly once and performing some operation or processing at each node when it's visited

Binary Tree Traversal

- Two types of tree traversal:
  - **Depth-first**: explores a tree subtree by subtree, visiting all of a node’s descendants before visiting any of its siblings.
    - moves as far downward in the tree as it can go before moving across in the tree
  - **Breadth-first**: explores a tree level by level, visiting every node at a given depth in the tree before moving downward
    - moves as far across the tree as it can go before moving down in the tree

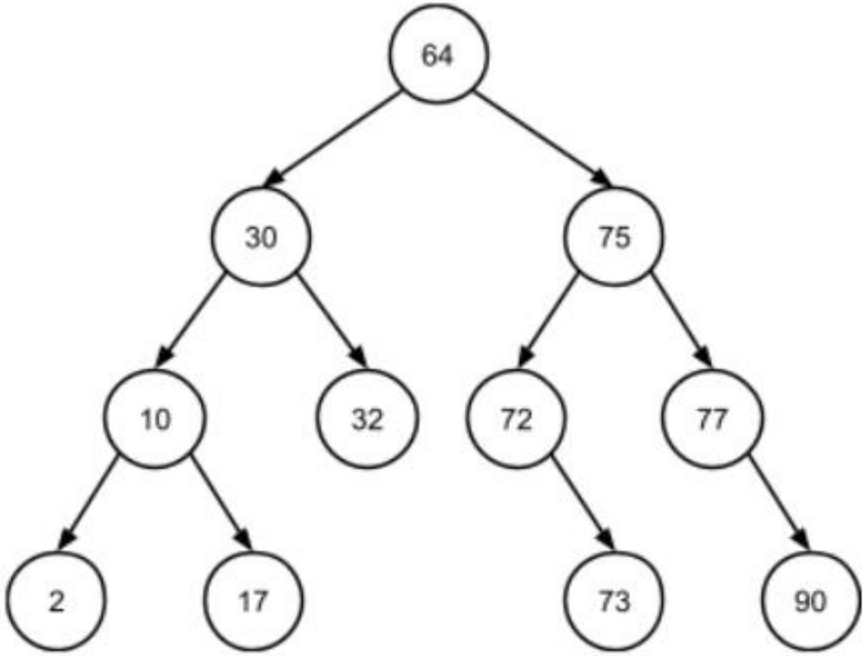
Binary Tree Traversal: Depth-first

- Denote using N, L, and R:
  - N – visit/process the current node itself
  - L – traverse the left subtree of the current node
  - R – traverse the right subtree of the current node

- Three kinds of depth-first traversal:
  - Pre-order traversal (NLR): process the current node before traversing either of its subtrees
  - In-order traversal (LNR): traverse the current node’s left subtree before processing the node itself, and then traverse the node’s right subtree
  - Post-order traversal (LRN): traverse both of the current node’s subtrees (left, then right) before processing the node itself

Binary Tree Traversal: Depth-first

- Three kinds of depth-first traversal:
  - Pre-order traversal (NLR)
    - 64 30 10 2 17 32 75 72 73 77 90
  - In-order traversal (LNR)
    - 2 10 17 30 32 64 72 73 75 77 90
  - Post-order traversal (LRN)
    - 2 17 10 32 30 73 72 90 77 75 64



- Note: in-order traversal processes the nodes in sorted order!

Binary Tree Traversal: Depth-first

- Pseudocode of three kinds of depth-first traversal: using recursion
  - Pre-order traversal (NLR)

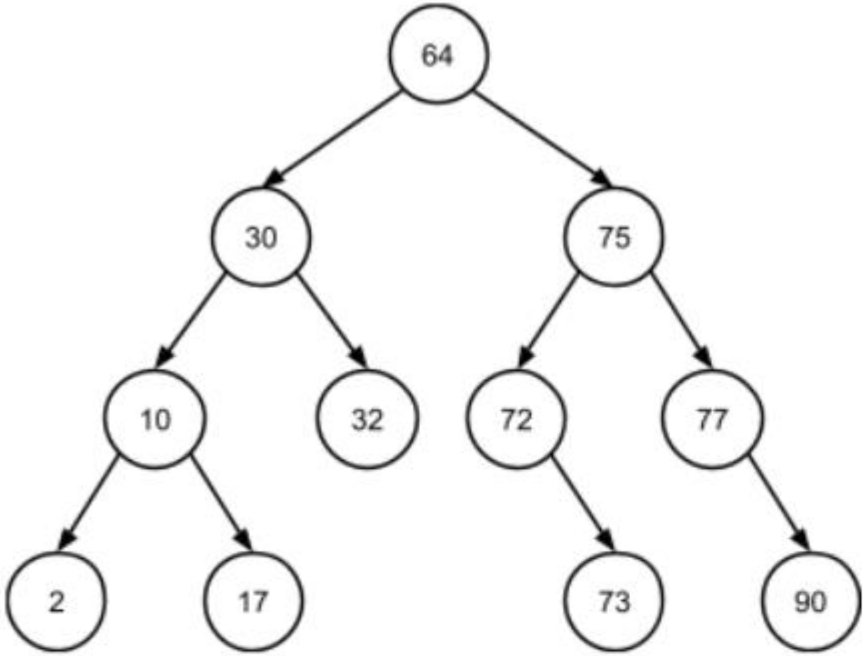
```
preOrder(N):
        if N is not NULL:
                process N
                preOrder(N.left)
                preOrder(N.right)
```

  - In-order traversal (LNR)

```
inOrder(N):
        if N is not NULL:
                inOrder(N.left)
                process N
                inOrder(N.right)
```

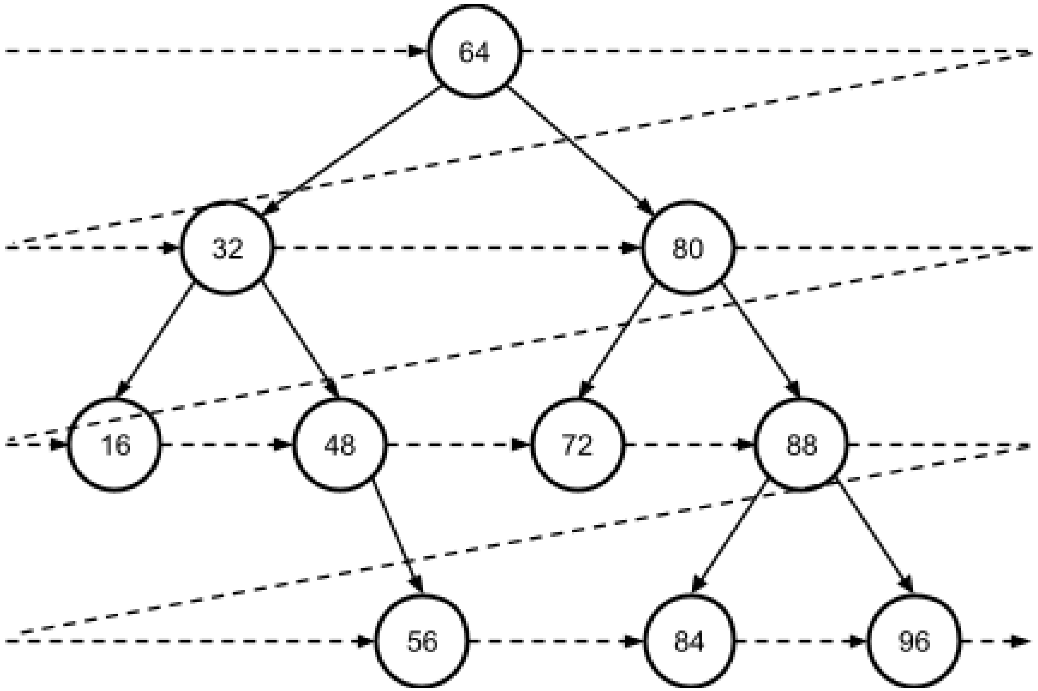
  - Post-order traversal (LRN)

```
postOrder(N):
        if N is not NULL:
                preOrder(N.left)
                preOrder(N.right)
                process N
```



Binary Tree Traversal: Breadth-first

- One main kind of breadth-first traversal: level-order traversal



- Using a level-order traversal, the nodes are processed in this order: 64, 32, 80, 16, 48, 72, 88, 56, 84, 96.

Binary Tree Traversal: Breadth-first

- Pseudocode of level-order traversal: using a queue

```
levelOrder(bst):
    q = new, empty queue
    enqueue(q, bst.root)
    while q is not empty:
            N = dequeue(q)
            if N is not NULL:
                    process N
                    enqueue(q, N.left)
                    enqueue(q, N.right)
```