Binary Search Trees

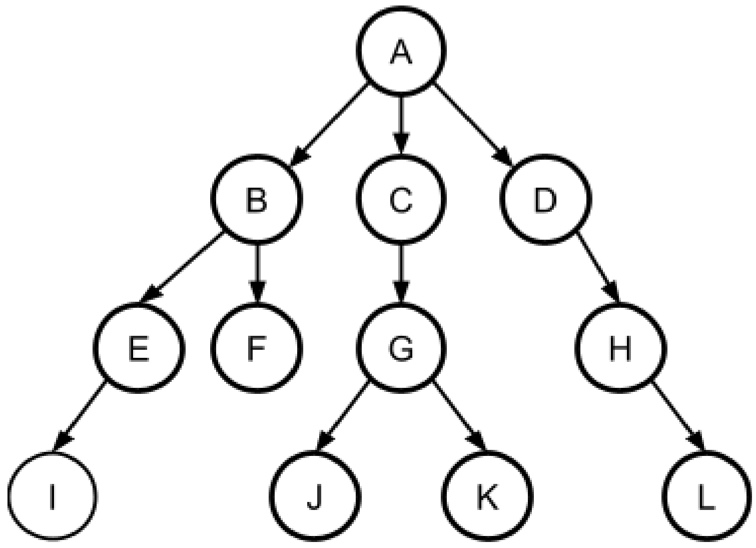
- ***Trees*** are the first major class of nonlinear data structures we'll explore in this course.

- In general, a tree represents a collection of data as a hierarchical structure, encoding the hierarchical relationships between different data elements.

- In a tree, each individual data element is represented as a ***node***, which contains the data element itself and also points to other nodes representing related elements.

- An encoded relationship between data elements in a tree is called an ***edge*** or an ***arc***. Edges in trees represent ***directed*** relationships.

- Visually, a tree is depicted like this, where the nodes are represented here as circles and the edges as directed arrows:



- Trees are ubiquitous in computer science. For example, a computer's filesystem, a browser's document object model representation of a web page, and a compiler's abstract syntax tree representation of a program are all examples of trees. Trees also show up all over the place in computer vision, machine learning, natural language processing, game theory, and more.

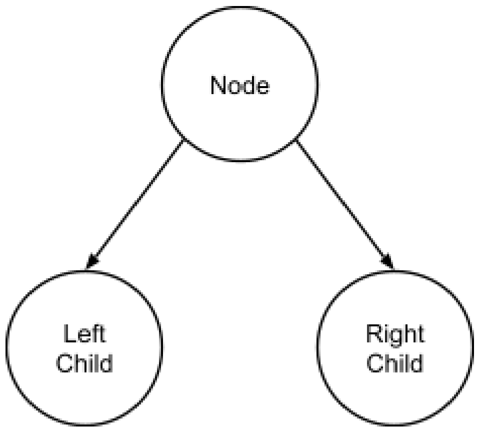
- When working with trees, there's a lot of important terminology you'll need to be familiar with, so you can correctly communicate with other people about trees. Here are a few of the most important terms to know:
    - ***Parent*** – A node *P* in a tree is called the parent of another node *C* if *P* has an edge that points directly to *C*.
        - In the tree depicted above, for example, node A is the parent of nodes B, C, and D, and node B is the parent of nodes E and F.
    - ***Child*** – A node *C* in a tree is called the child of another node *P* if *P* is *C*'s parent.
        - In the tree depicted above, nodes B, C, and D are children of node A, and nodes J and K are children of node 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*.
        - In the tree depicted above, nodes B, C, and D are siblings, and nodes J and K are siblings.
    - ***Descendant*** – The descendants of a node *N* are all of *N*'s children, plus its children's children, plus the children of those nodes, and so forth.
        - In the tree depicted above, nodes 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*.
        - In the tree depicted above, nodes E, B, and A are ancestors of node I, and nodes G, C, and A are ancestors of node K.
    - ***Root*** – The ancestor of all other nodes in the tree is called the root. Each tree has exactly one root.
        - In the tree depicted above, node A is the root.
    - ***Interior node*** – A node in a tree is an interior node if it has at least one child.
        - In the tree depicted above, nodes A, B, C, D, E, G, and H are interior nodes.
    - ***Leaf node*** – A node in a tree is a leaf node (or just leaf) if it has no children. When implementing a tree, a leaf node conventionally has all NULL children.
        - In the tree depicted above, nodes F, I, J, K, and L are leaves.
    - ***Subtree*** – A subtree is the portion of a tree that consists of a single node *N*, all of *N*'s descendants, and the edges joining these nodes. We call this the subtree rooted at *N*.
        - In the tree above, the subtree rooted at node B contains the nodes B, E, F, and I and the edges joining those nodes.

    - ***Path*** – A path is the collection of edges in a tree joining a node to one of its descendants.
    - ***Path length*** – The length of a path in a tree is equal to the number of edges in that path.
        - In the tree depicted above, the path from node C to node 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*. By definition, the root node has depth 0.
        - In the tree above, the depth of node K is 3.
    - ***Height*** – A tree's height is the maximum depth of any node in the tree.
        - The tree depicted above has height 3.

- Importantly, a structure must obey the following 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. In other words, there cannot be a path from any node to itself.

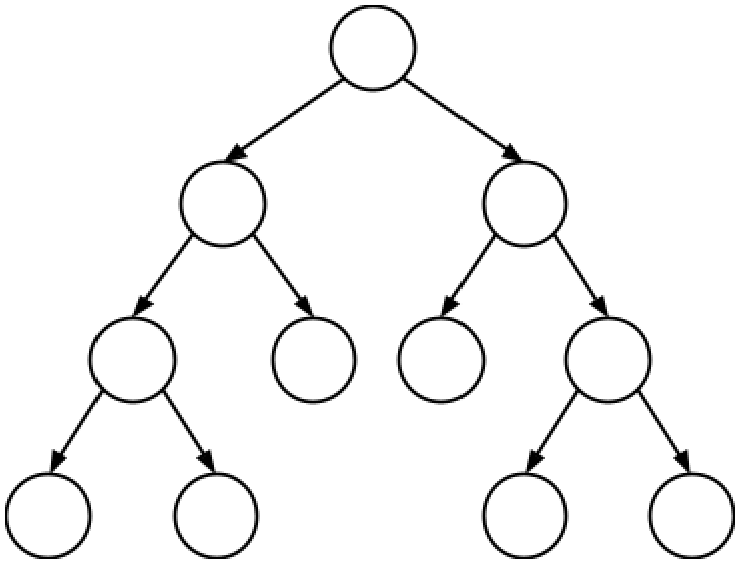
Binary trees

- A ***binary tree*** is a special type of tree in which each node can have at most two children. The children of a binary tree node are typically thought of as ordered, and we refer to them as the ***left child*** and the ***right child***:

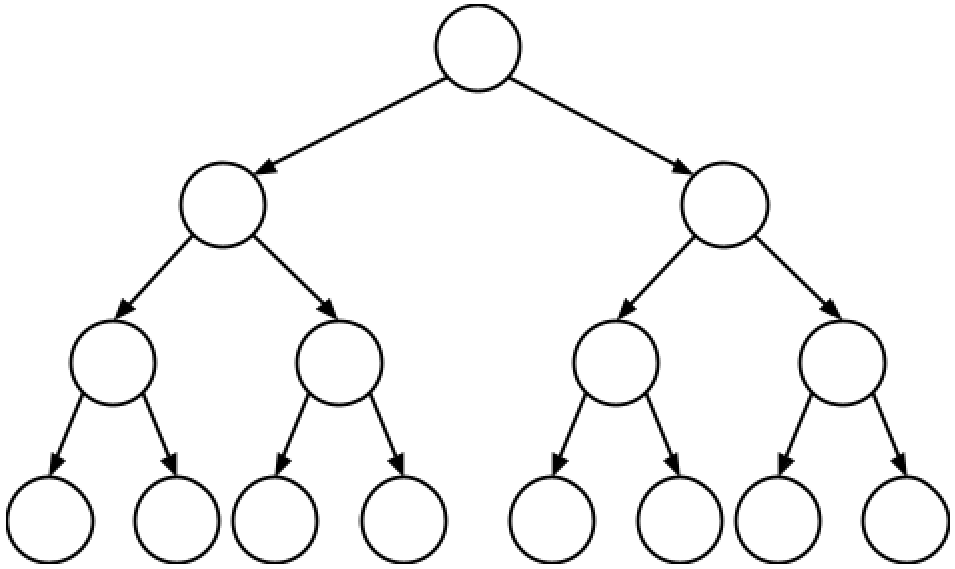


- Within a binary tree, we can further talk of a node's ***left subtree*** as the subtree rooted at that node's left child and its ***right subtree*** as the subtree rooted at its right child.

- There are many special forms of binary tree that are important to be aware of. For example, a ***full*** binary tree is a binary tree in which every interior node has exactly two children:



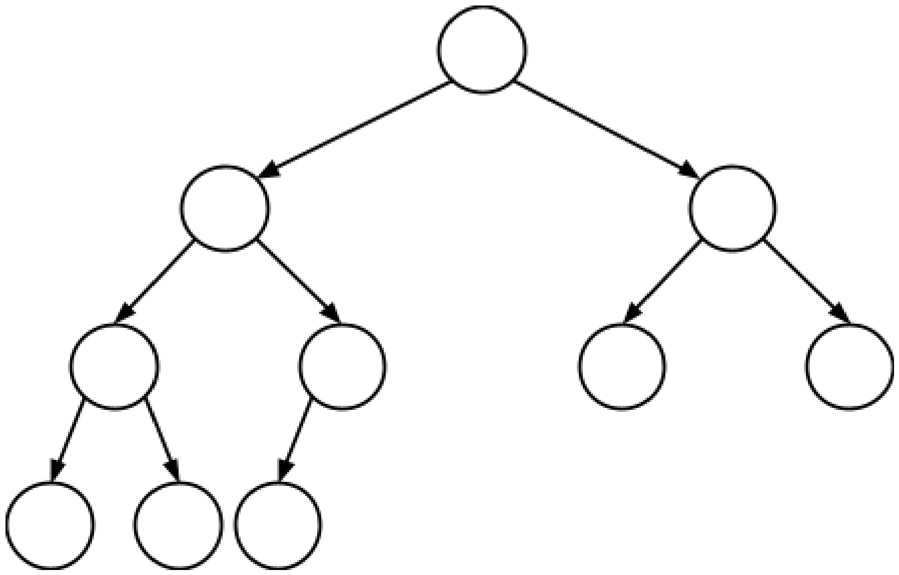
- A ***perfect*** binary tree is a full binary tree (i.e. one where all interior nodes have two children) where all the leaves are at the same depth:



- Importantly, if we know a perfect binary tree has height *h*, then we know:
  - It has 2^h leaves.
  - It has $2^{h+1} - 1$ total nodes.

- Conversely, if we know that a perfect binary tree has *n* nodes, then we know its height is approximately *log(n)*.

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



Binary search trees

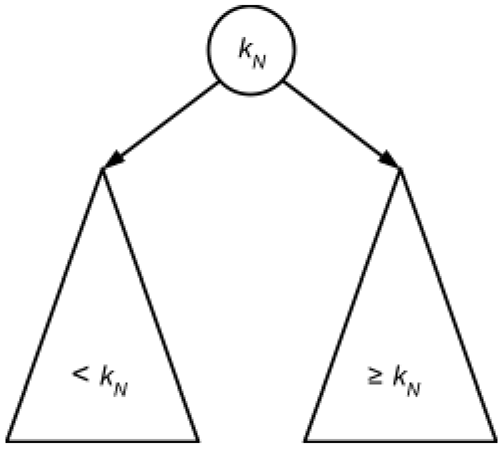
- Though we haven't really been depicting it in the trees above, it's important to remember that each node in a tree represents a data element. In what follows, we'll actually consider each data element can be represented using a ***key***, or identifier (the data element may also contain other data, which we can refer to as its ***value***).

- Importantly, we'll assume that these keys can be ordered in relation to each other. For example, integer keys can be ordered numerically, and string keys can be ordered alphabetically.

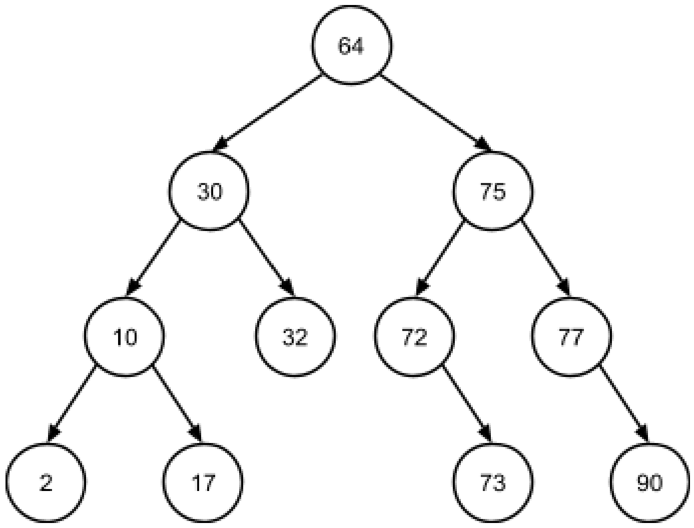
- A ***binary search tree*** (or ***BST***), then, is a binary tree that satisfies the following property (known as the "binary search tree property"):

  *Within a binary search tree, 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.*

- In other words, at each BST node *N* (whose key is k_N), the following relationship holds (where each triangle represents an entire subtree beneath *N*):



- For example, here's a valid BST containing integer keys:



- Note that there is no restriction on the *shape* of a BST. For example, a BST does *not* have to be full, perfect, complete, etc. The only requirement of a BST is that the BST property holds at all of its nodes.

BST operations

- Let's look at how we can perform some of the important operations on a BST, including finding an element, inserting a new element, and removing an element.

- As we explore these operations, remember the important convention that 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. Thus, a leaf node in a BST is one where both the left and right child are NULL.

Finding an element in a BST

- Elements in a BST are located based on their keys. For instance, when a user wants to locate an element, they will need to provide the key of the element they want to find.

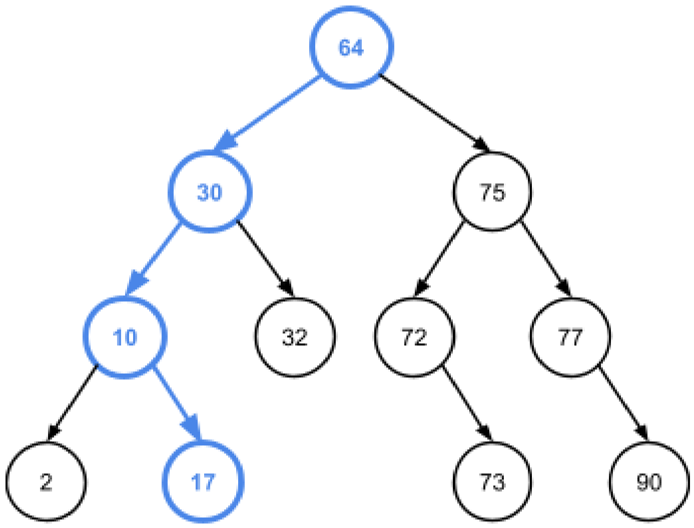
- Once we know the key of the element we want to find in the BST, searching for that element proceeds much like binary search within an ordered array.

- Given a query key k_q, here's how it works at a high level:
    - Search proceeds by examining one node at a time. Keep a pointer to the current node, starting at the root.
    - If the current node is NULL, the key k_q doesn't exist in the tree, and the search has failed. Halt.
    - If the current node's key is equal to k_q, the search has succeeded. Halt.
    - Otherwise, if k_q is less than the current node's key, move the current node to point to its *left* child and repeat.
    - Otherwise, k_q is greater than the current node's key, so we move the current node to point to its *right* child and repeat.

- This can be expressed in pseudocode as follows:

```
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
```

- For example, when a search for the key 17 in the BST depicted above would proceed from the root node (whose key is 64), to the node with key 30, to 10, to 17:



- The find operation also has a straightforward recursive implementation.

Inserting a new element into a BST

- New elements are always inserted into a BST as leaves.
    - In this way, we avoid the need to restructure the tree by inserting a new interior node.

- The key then, to inserting a new element into a BST is finding the right location for the new leaf node representing that element. In particular, the new leaf node has to be inserted at a location that maintains the BST property at all nodes in the tree.

- Assuming each node in a BST contains both a key *k* and a value *v*, we can find the location at which to insert the new node in much the same way as if we were simply searching through the tree for key *k*.

- However, when inserting a new element, 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. In particular, the new node will become the child of the NULL node's parent.

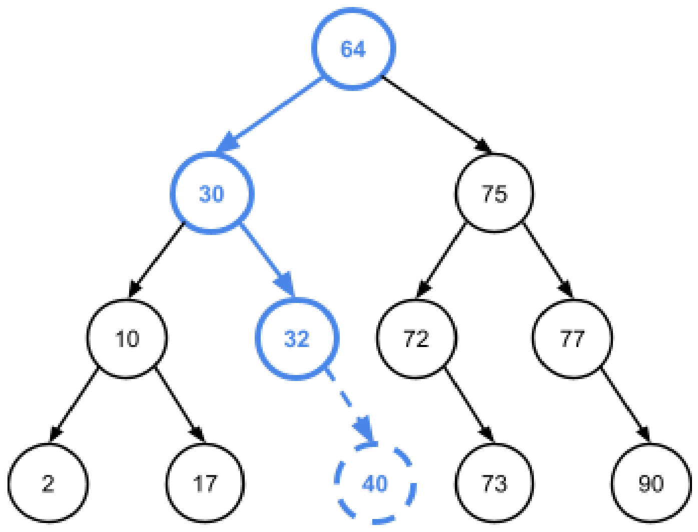
- In pseudocode, insertion into a BST looks like this:

```
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
```

- In the pseudocode above, **P** is used to track the location of the new node's parent, but otherwise the search proceeds as it does in the `find()` method.

- Importantly, if **P** is NULL at the end of the search here, this is an indication that 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.

- For example, if we wanted to insert the key 40 into the BST depicted above, the search for the new node's location would start at the root (whose key is 64) and proceed to the node with key 30, and then to the node with key 32. Then, after finding that 32's right child was NULL, the new node with key 40 would be inserted at that location:



- Just like searching for an element in a BST, inserting an element has an elegant recursive implementation.

Removing an element from a BST

- Removing an element from a BST poses interesting challenges. In particular, it is easier to remove some elements from a BST than it is to remove other ones.

- For example, as you might suspect, removing the element with key 2 from the BST depicted above will be rather straightforward. However, removing the element at the root node (whose key is 64) will be less so.

- How we handle removing an element will specifically depend on the number of children that element's BST node has.

- If the element to be removed is stored in a leaf node (i.e. a node with no children, like the one with key 2 in the BST depicted above), we can 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 (like the node with key 72 in the BST depicted above), we can simply free that node and move its child to become a child of the node's parent.

- If the element to be removed is stored in a node with two children (like the root node, with key 64, in the BST depicted above), things become more complicated.

- Before we remove an element in a node with two children, we first need to find what's known as that node's ***in-order successor***.
    - If, when inserting an element with a key equal to one already in the tree, we branched left at that equal key instead of right, then when removing a node with two children, we'd want instead to find its ***in-order predecessor***. Can you explain why this is?

- To understand what a node's in-order successor is, it's helpful to just imagine all of the elements stored in the tree lined up in ascending order, by key. For example, in the BST depicted above, this order (of the keys) would look like this:

```
2      10     17     30     32     64     72     73     75     77     90
```

- The in-order successor, then, for a node with key *k*, is simply the node corresponding to the very next key *after k* in this ordered list of keys.
    - The same node's in-order predecessor is similarly the node corresponding to the key *before k* in the ordered list of keys.

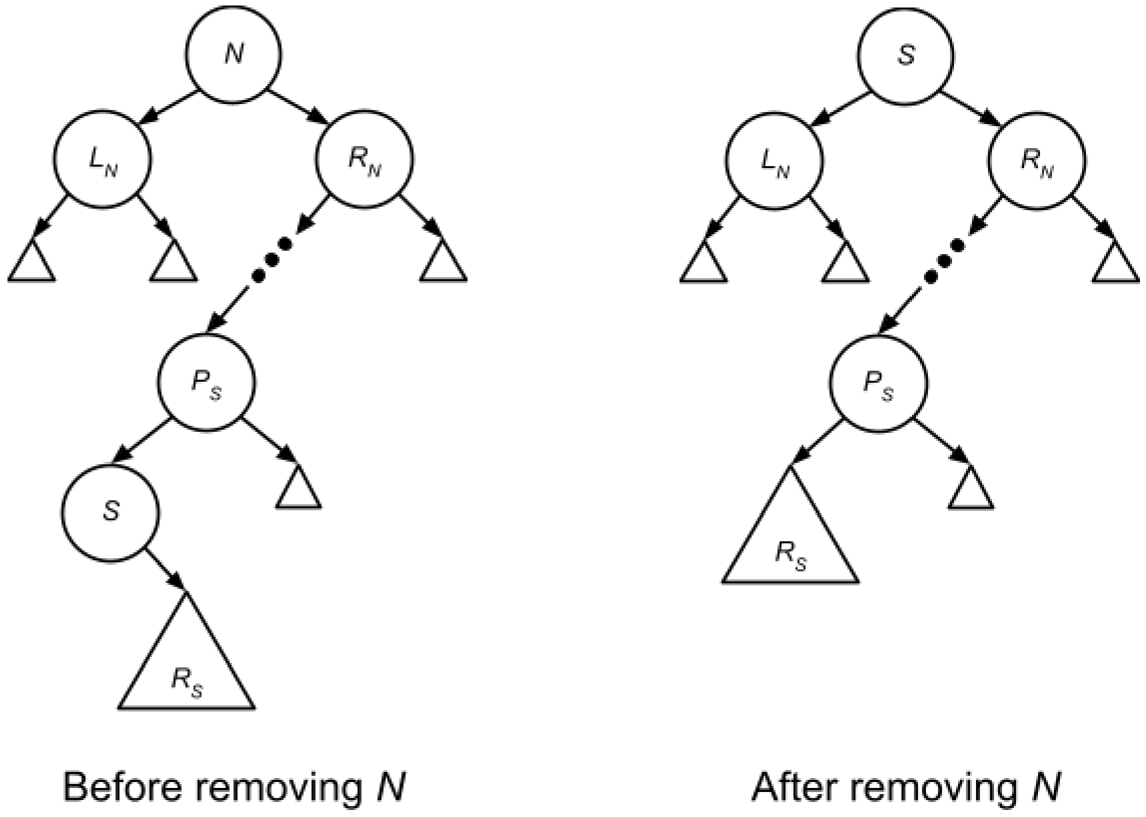
- For example, the in-order successor of the root node in our example BST (whose key is 64) is the node with key 72, since 72 comes immediately after 64 in the sorted list of keys.

- Now that we know what an in-order successor *is*, how do we actually *find* the in-order successor of a given node in a BST? It's actually quite easy. Specifically, a node *N*'s in-order successor is always the *leftmost* node in *N*'s *right* subtree.
    - In other words, to find *N*'s in order successor, we branch right in the tree from *N*, and then continue to branch left until we can no longer do so (i.e. until we reach a node with no left child). The last node we reach this way (i.e. the one with no left child) will be *N*'s in-order successor.

- We can verify that this works in the BST depicted above. Indeed, the node with key 72 is the leftmost node in the right subtree of the root node (whose key, again, is 64), just as predicted. The same will hold for the in-order successor of any other node in the tree.

- Now, knowing how to *find* a node's in-order successor, we can turn our attention back to the task of removing a node with two children from a BST. At a high level, here's how to remove a node *N* that has two children:
    - 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* and remove *N* from the tree:
        - *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*.

- Visualized, the tree will look like this before and after *N*'s removal:



- In these visualizations, the small triangles represent locations where parent/child relationships are not affected during the removal of *N*. Any of these might correspond to a single node, a larger subtree, or NULL. Again, R_S, which represents *S*'s right subtree, might also be a single node, a larger subtree, or NULL in this situation.

- Note that the actual shape of the tree here changes near *S*, not near *N*. In particular, structurally, *S* is the node that's removed from the tree, with its (possibly NULL) right subtree becoming the *left* subtree of P_S.
    - Importantly, this restructuring maintains the BST property. You should be able to explain why.

- It's important to note, too, that *N*'s in-order successor could simply be *N*'s own right child (R_N in the visualizations above). In this case, *N*'s original left child (L_N above) simply becomes the left child of *N*'s right child (i.e. R_N).

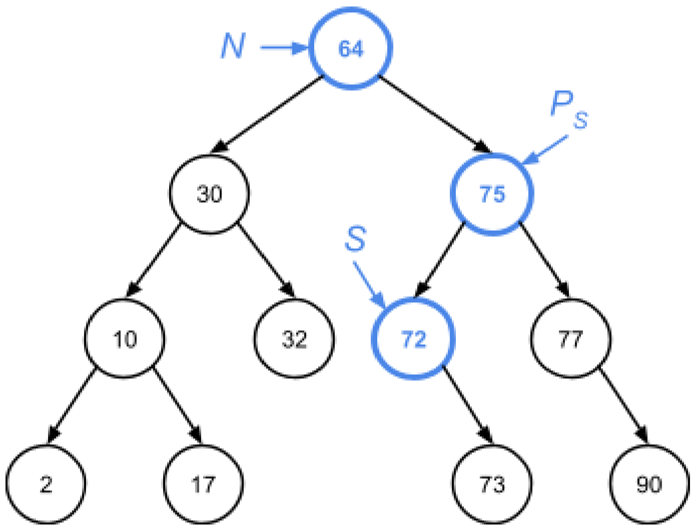
- The pseudocode below accomplishes all three removal cases (i.e. removing a node with zero, one, or two children). As when finding a node in a BST, a node is removed based on its key:

```
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
```

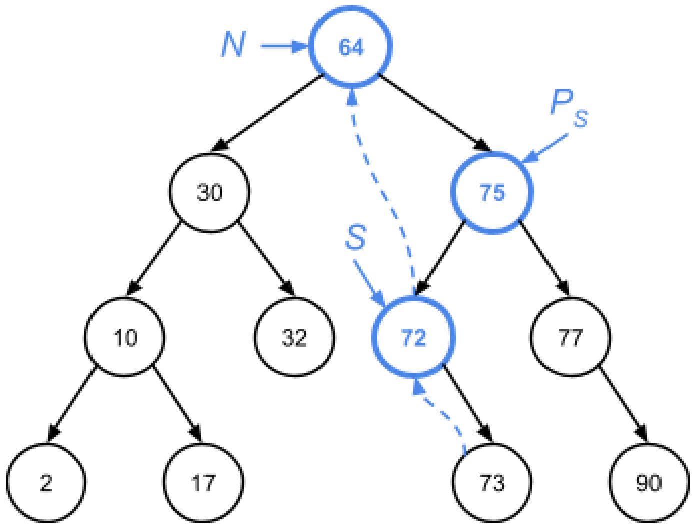
- See also Algorithm D in Volume 3 of The Art of Computer Programming for an elegant binary search tree removal function.

- As a concrete example of how removing a BST node with two children works, let's explore what it would look like to remove the root node (with key 64) from the example tree we've been working with.

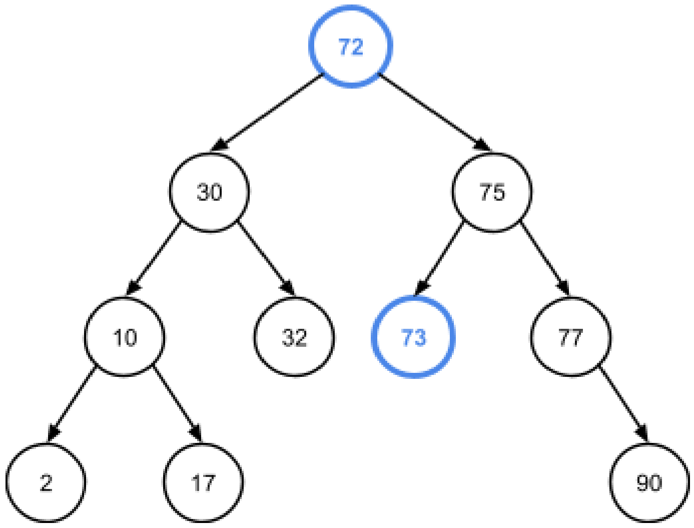
- First we must identify that node's in-order successor (*S*) and its parent (P_S):



- Next, we need to update pointers so that *S* replaces *N* and *S*'s right child replaces *S* as P_S's child:



- The end result is a tree with the root node (i.e. *N*) removed. Again, note that the BST property is maintained by this removal:



Runtime complexity of BST operations

- The main factor in the computational complexity of all three main BST operations (find, insert, and remove) arises from the need to search within the tree.
    - In the case of the find operation, we search for the query key.
    - In the case of the insert operation, we search for the correct location at which to insert the new node.
    - In the case of the remove operation, we search for both the query key (i.e. the node to be removed) and its in-order successor.

- In all three operations, search begins at the root of the tree, and at each iteration, moves down the tree one level, proceeding until it reaches the bottom of the tree (or until it finds the node it's searching for).
    - During the remove operation, the combination of the search for the node to be removed and the search for its in-order successor proceeds from the root node to the bottom of the tree.

- In other words, each operation performs a number of search iterations equal at most to the height of the tree, *h*.

- It should be straightforward to recognize that each iteration of these searches performs constant-time work, i.e. making a comparison and branching left or right (or halting) based on that comparison.

- Thus, the total amount of work done searching in all three of these operations is *O(h)*, where *h*, again, is the height of the tree.

- What about the extra work done during the insert and remove operations?
    - In the case of the insert operation, we need to allocate the new node and update its new parent to point to it. This can be done in constant time.
    - In the case of the remove operation, we need to update at most a few pointers to replace the node being removed with its child or in-order successor before freeing the removed node. Again, this can all be done in constant time.

- Thus, each of these three operations (find, insert, and remove) has runtime complexity *O(h)*.

- Of course for a fixed number of nodes *n*, the height *h* of the BST can vary greatly, depending on the order in which nodes are inserted. It can be as little as *log(n)* or as great as *n*, so the actual runtimes of the main BST operations can be great or small.

- Soon, we'll explore strategies for limiting the height of the BST to be close to *log(n)*, thereby limiting the runtimes of the main BST operations to be *O(log n)*.