Deletion

In the lesson, we'll see how to delete a key from a BST.

We'll cover the following

Deletion

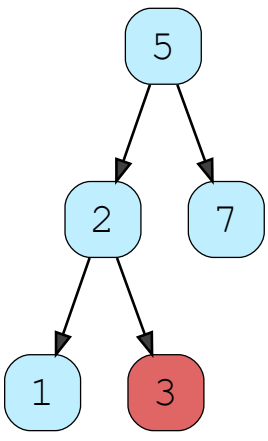
While removing a node from a BST, three cases are possible:

1. Node is a leaf
2. Node has only one child
3. Node has two children

In any case, we could end up traversing the entire tree in case of a skewed binary tree. So, the time complexity is $O(N)$.

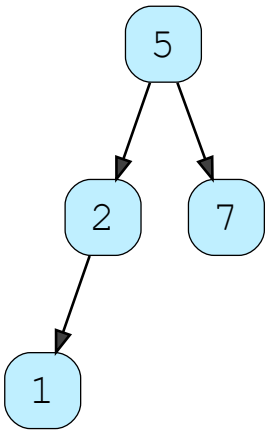
Node is a leaf

In this case, the node can simply be removed from the tree.



3 is a leaf, simply mark right child of 2 as NULL

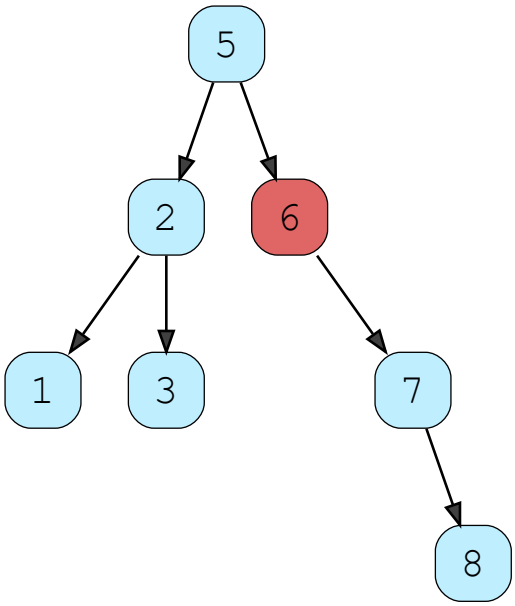
1 of 2



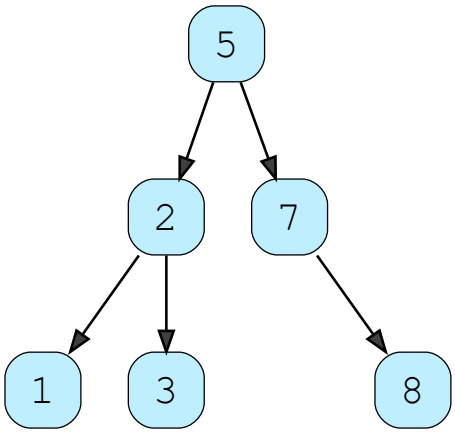
2 of 2

Node has only one child

The parent of this node now points to the child instead of this node. Then we just remove the parent node.



6 has only one child

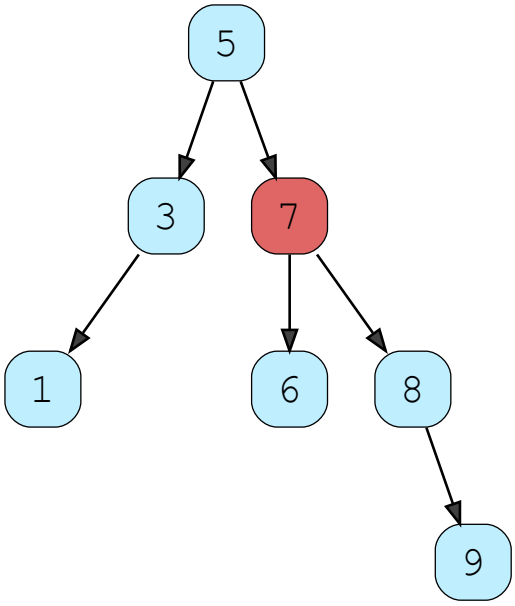


5 now points to 7 directly, Delete 6

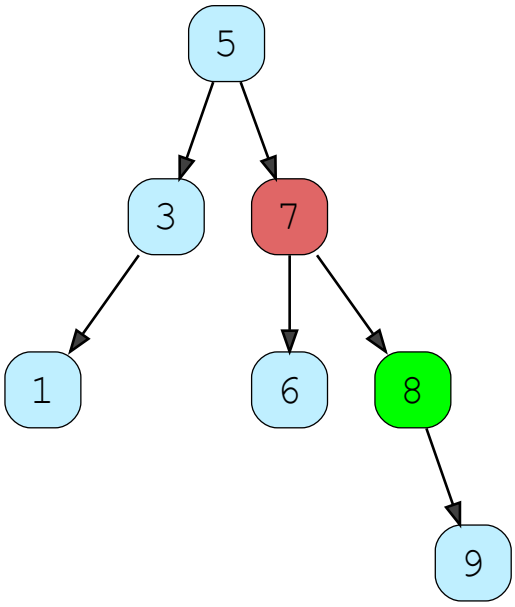
Node has two children

In this case, the node to be deleted is replaced by the in-order successor of this node (smallest value node in the right child).

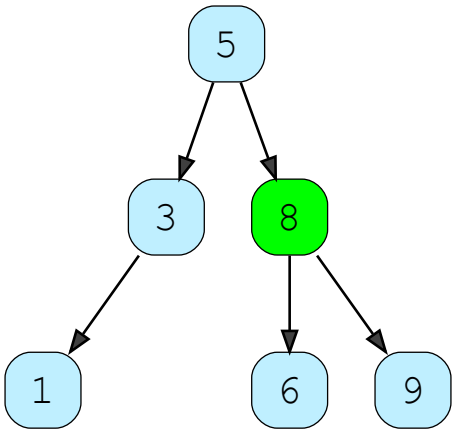
Replacing the node with the in-order successor will not break the BST property. After replacing, we will need to delete the in-order successor from the tree. To do that we can call this *delete* operation on that key in the right subtree.



To delete 7, find in-order successor of 7



in-order successor is 8, swap value and delete this node




```
#include <iostream>
using namespace std;

struct Node {
  int val;
  Node *left, *right;

  Node(int val){
    this -> val = val;
    this -> left = NULL;
    this -> right = NULL;
  }
};

void insert(struct Node* &root, int val) {
  if (root == NULL) {
    root = new Node(val);
    return;
  }

  Node* pCrawl = root;
  Node* pCrawlParent;
  while(pCrawl) {
    pCrawlParent = pCrawl;

    if (val < pCrawl->val)
      pCrawl = pCrawl->left;
    else
      pCrawl = pCrawl->right;
  }
```

```
  if (val < pCrawlParent->val)
    pCrawlParent->left = new Node(val);
  else

    pCrawlParent->right = new Node(val);
}

void in_order(struct Node* node)  {
    if (node == NULL)
        return;

    in_order(node -> left);
    cout << node -> val << " ";
    in_order(node -> right);
}

struct Node* min_value_node(struct Node* node) {
    struct Node* pCrawl = node;

    while (pCrawl->left != NULL)
        pCrawl = pCrawl->left;

    return pCrawl;
}

Node* delete_node(struct Node* root, int val) {
  if (root == NULL) return root;

  if (val < root->val) // The key to be deleted is in the left subtree
      root->left = delete_node(root->left, val);

  else if (val > root->val) // The key to be deleted is in the right subtree
      root->right = delete_node(root->right, val);

  else {  // The current node is to be deleted
    if (root->left == NULL && root->right == NULL) { // Case 1
        // Returning null here will make the parent's pointer to this node null
        // effectively removing this node from the tree
        return NULL;
    }
    else if (root->right == NULL) {  // Case 2
        // Parent's pointer to this node is replaced with left child of this node
        return root->left;
    }
    else if (root->left == NULL) {  // Case 3
        // Parent's pointer to this node is replaced with right child of this node
        return root->right;
    }
    else { // Case 3
      // Find minimum value in the right subtree (in-order successor)
      // Copy to this node
      // Delete the inorder successor
      struct Node* temp = min_value_node(root->right);
      root->val = temp->val;
      root->right = delete_node(root->right, temp->val);
    }
  }
  return root;
}

int main() {
  // Creating the same tree as in the illustration above for Case 1
  Node* root1 = NULL;
```

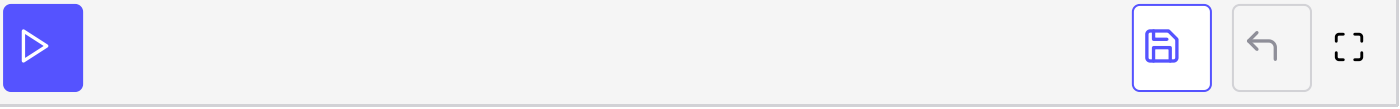
```
  insert(root1, 5);insert(root1, 2);insert(root1, 7);
  insert(root1, 1);insert(root1, 3);
  in_order(root1); cout << "\n";
  delete_node(root1, 3);
  in_order(root1); cout << "\n\n";


  // Creating the same tree as in the illustration above for Case 2
  Node* root2 = NULL;
  insert(root2, 5);insert(root2, 2);insert(root2, 6);
  insert(root2, 1);insert(root2, 3);insert(root2, 7);
  insert(root2, 8);
  in_order(root2); cout << "\n";
  delete_node(root2, 6);
  in_order(root2); cout << "\n\n";


  // Creating the same tree as in the illustration above for Case 3
  Node* root3 = NULL;
  insert(root3, 5);insert(root3, 3);insert(root3, 7);
  insert(root3, 1);insert(root3, 6);insert(root3, 8);
  insert(root3, 9);
  in_order(root3); cout << "\n";
  delete_node(root3, 7);
  in_order(root3); cout << "\n\n";



  return 0;
}
```



In the next lesson, we'll learn more about BSTs.