Deletion

There are three main cases for deletion of a node from a BST. The first two are easy to handle. Only the third is tricky:

1. The node to delete is a leaf. The appropriate node of the parent is set to `null` and the node is deallocated:

2. The node to delete has one child. The node’s parent can “adopt” the child. The node itself is then deallocated:

   (think about why this operation is ok)

3. The node has two children. What do we do here?

Deletion by merging

Assume we are in the third case. The node to delete has two subtrees. Our first approach is to merge one subtree into the other, we would then be in case 2 (node to delete has one child). Consider removing the 10-node from the following BST:

The 8-node is the inorder predecessor to the 10-node because it is visited just before the 10 in an inorder traversal. Similarly, the 12-node is the inorder successor. Either one of these nodes can “adopt” the entirety of the other subtree, as shown above.

Balanced trees

- AVL trees
Deletion by merging

Assume we are in the third case. The node to delete has two subtrees. Our first approach is to merge one subtree into the other, we would then be in case 2 (node to delete has one child). Consider removing the 10-node from the following BST:

```
   8
  / \   
 3   15
 / \    
2   12   18
 /   
1   20
```

The 8-node is the **inorder predecessor** to the 10-node because it is visited just before the 10 in an inorder traversal. Similarly, the 12-node is the **inorder successor**. Either one of these nodes can “adopt” the entirety of the other subtree, as shown above.

Deletion by merging

Assume we are in the third case. The node to delete has two subtrees. Our first approach is to merge one subtree into the other, we would then be in case 2 (node to delete has one child). Consider removing the 10-node from the following BST:

```
   8
  /   
 3   12
 / \   
2   18   20
```

The 8-node is the **inorder predecessor** to the 10-node because it is visited just before the 10 in an inorder traversal. Similarly, the 12-node is the **inorder successor**. Either one of these nodes can “adopt” the entirety of the other subtree, as shown above.
Deletion by merging

Assume we are in the third case. The node to delete has two subtrees. Our first approach is to merge one subtree into the other, we would then be in case 2 (node to delete has one child). Consider removing the 10-node from the following BST:

```
        10
       / \
      8   12
     /   / \
    6   8  12
   / /   /   /
  4 5  6  8  12
```

The 8-node is the inorder predecessor to the 10-node because it is visited just before the 10 in an inorder traversal. Similarly, the 12-node is the inorder successor. Either one of these nodes can “adopt” the entirety of the other subtree, as shown above.

Deletion by merging

Assume we are in the third case. The node to delete has two subtrees. Our first approach is to merge one subtree into the other, we would then be in case 2 (node to delete has one child). Consider removing the 10-node from the following BST:

```
        10
       / \
      8   12
     /   / \
    6   8  12
   / /   /   /
  4 5  6  8  12
```

The 8-node is the inorder predecessor to the 10-node because it is visited just before the 10 in an inorder traversal. Similarly, the 12-node is the inorder successor. Either one of these nodes can “adopt” the entirety of the other subtree, as shown above.
We will choose to find the inorder predecessor of the node to delete and make its right child the right subtree of the node to delete. This works because the inorder predecessor cannot have a right child. The following summarizes this process:

How do we find the inorder predecessor? It is the rightmost node in the left subtree. Start at node->left and go right until node->right is null.

The function `deleteByMerging` implements deletion by merging. It assumes that `node` already points to the node we wish to delete. The easy cases (1 and 2) are also handled.

Following this, we present `findAndDeleteByMerging` which first finds the node we wish to delete, then calls `deleteByMerging`.

```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T>* tmp = node;
    if (node != 0) {
        if (!node->right) { // no right child
            node = node->left;
        } else if (node->left == 0) { // no left child
            node = node->right;
        } else {
            tmp = node->left; // 1. move left
            while (tmp->right != 0) { // 2. and then right
                tmp = tmp->right;
                tmp->right = node->right; // 3. merge in right subtree
            }
            tmp = node; // 4.
            node = node->left; // 5.
            delete tmp; // 6.
        }
    }
}
```

The following illustrates the steps in `deleteByMerging`:
In our earlier example, the merging step increased the height of the tree. This is not always the case as illustrated in (b) below:

Deletion by copying

The first step in any sort of deletion is finding the pointer node which points to the node we wish to delete. But we don’t actually care about deleting the node itself. We wish to remove the value stored by this node from the tree.

Deletion by copying proceeds as follows:

- Find a pointer to the node containing the value we wish to remove: node
- If the node is easy to delete, we delete it (case 1 or 2)
- If the node has two children, we find another node that is easier to delete: tmp
- We copy the value from tmp into node
- Delete the node pointed at by tmp

What nodes are both easy to delete and swappable with node? The inorder predecessor and successor. We choose the inorder predecessor.
The following figure illustrates the operation of \texttt{deleteByCopying}:

We must call \texttt{findAndDeleteByCopying} to both find and remove the given value from the tree. This function is identical to \texttt{findAndDeleteByMerging} except that it calls \texttt{deleteByCopying} instead of \texttt{deleteByMerging}.

A binary tree is \textbf{perfectly balanced} if it is balanced and all leaves are found on the highest two levels of the tree.

A complete binary tree is perfectly balanced by definition. We have already established a link between the number of nodes \( n \) in a complete binary tree and its height: \( h = \lg(n + 1) \).

Consider searching for a node in a complete BST versus a BST where no node has more than one child (i.e. a linked list).

Let \( n = 2^{20} - 1 = 1,048,575 \). What is the number of comparisons required in the worst case?

- Complete BST: 20
- Linked-list type tree: 1,048,575

\textbf{AVL trees}

AVL trees are balanced BST’s where any insertions or deletions to the tree that would unbalance it are corrected by local transformations known as \textit{rotations}.

AVL trees are not perfectly balanced, but being balanced (i.e. balance factor is -1, 0, or +1) is enough to ensure that operations on the tree are performed efficiently (i.e. worst-case complexity of \( O(\lg n) \) instead of \( O(n) \)).

Thus, AVL trees (and other similar balanced trees) are the data structure of choice for many applications.

AVL trees are the oldest and best-known balanced trees. They were named after their inventors: Adel’son-Vel’skii and Landis.
Any BST with 0, 1, or 2 nodes is balanced, and is therefore an AVL tree. The trick is to maintain a tree’s balance when inserting or removing nodes. To make balance calculations efficient, we will store in each node the current height of the subtree rooted at that node.

After an insertion or a removal in a tree that is currently balanced, the tree might no longer be balanced. Since we have inserted or removed only one node, the largest any balance factor can be is +2, and the smallest is -2. We consider only the case of a balance factor of +2. The -2 case is symmetric.

Inserting or removing one node can put more than one ancestor node out of balance. Consider the deepest node that is out of balance. This node has a balance factor of +2 (by assumption). Call this node \( a \) (this is a label, and does not indicate that the node actually stores “a”).

Let \( h \) be the height of the left subtree of node \( a \). The minimum height of the left subtree is zero, therefore the minimum height of the right subtree is 2. Thus, the right subtree cannot be empty. Call its root \( b \). There are three choices for the balance factor of the right subtree: -1, 0, +1 (why?). We will consider each case separately...

**Case 0:** The balance factor of node \( b \) is 0. We have the following situation.

As this remains a BST, the following is true:
- All nodes in \( x < a < \) All nodes in \( y < b < \) All nodes in \( z \)

Therefore, we can apply the following transformation, known as a left rotation:

The rotation has restored balance. Note that the height of \( a \) has decreased by one, while the height of \( b \) has increased by one.
Case +1: The balance factor of node \( b \) is +1. We have the following situation.

The same relations hold as above:
All nodes in \( x < a < y < b < All nodes in z \)
Therefore the same left rotation can be applied to correct this imbalance...

In this case, however, the changes in height are slightly different. The rotation decreases the height of \( a \) by two, while the height of \( b \) has stayed the same.

Case -1: The balance factor of \( b \) is -1.

Unfortunately, the left rotation will not help us (why not?).
We need to shift the subtree \( y \) up somehow. In order to do this, we must look more deeply into this subtree...

We know for sure that \( y \) has at least one node. Let \( c \) be the root of \( y \).

There are three possibilities for the balance factor of \( c \). Fortunately, it turns out that we can handle all three in the same way.
\( x < a < y_0 < c < y_1 < b < z \)
Therefore, we can perform the following transform, known as a **double left rotation**:

- The height of \( a \) has decreased by 2.
- The height of \( b \) has decreased by 1.
- The height of \( c \) has increased by 1.

We now present the pseudocode for insertion into an AVL tree.

We add one field to the regular BST node class, allowing the height of the subtree rooted at that node to be stored. Knowing the heights allows us to determine balance factors:

```cpp
int AVLTree::getHeight(Node *node) {
    if (node == NULL) return 0;
    else return node->height;
}
```

```cpp
int AVLTree::getBalanceFactor(Node *node) {
    if (node == NULL) return 0;
    else return getHeight(node->right) - getHeight(node->left);
}
```

We now can insert or remove as if this were a regular BST... Except...

After performing the insertion or removal of a node we rebalance its parent if necessary.

We will begin with general recursive insertion into a BST:

```cpp
void AVLTree::insert(int data, Node *&node) {
    if (node == NULL) {
        node = new Node; // Sets both children to NULL
        node->item = data;
    } else if (data < node->item) {
        insert(data, node->left);
    } else {
        insert(data, node->right);
    }
}
```

This must be called initially with `node` set to the root.
We now make the appropriate additions for AVL tree insertion:

```cpp
void AVLTree::insert(int data, Node * &node) {
    if (node == NULL) {
        node = new Node; // Sets both children to NULL
        node->item = data;
        node->height = 1;
    } else if (data < node->item) {
        insert(data, node->left);
        node->height = 1 + max(getHeight(node->left), getHeight(node->right));
        rebalance(node);
    } else {
        insert(data, node->right);
        node->height = 1 + max(getHeight(node->left), getHeight(node->right));
        rebalance(node);
    }
}
```

The code given above updates heights and performs rebalancing (if necessary) all the way up along the path of nodes that were visiting during the downwards portion of the insertion (i.e. when trying to find out where to insert. Note that only one rebalance is necessary.

E.g. Build an AVL tree by inserting the following nodes in order: 50, 80, 90.

A left rotation is required.

Continue: 100, 95.

A double left rotation is required.

The rebalancing takes no more than a constant amount of time at each level of the tree.

How many levels does an AVL tree have? The height of an AVL tree is bounded by $O(\lg n)$ (justification for this is beyond our scope—see text for details). Hence insertion and removal will have a worst-case time complexity of $O(\lg n)$.

There are a number of similar self-balancing trees: 2-3 trees, 2-3-4 trees, red-black trees. They all have pros and cons with respect to implementation complexity and performance. However, they all exhibit the same worst-case time complexity of $O(\lg n)$ for search, insertion, and removal.