Data Structures

BST Deletion 3- Design Consequences

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

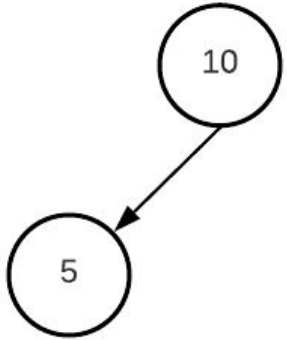
Design Limitations

- In the last lecture, we learned we must need 2+ nodes before deletion
  - Why? Compare to linkdelist, we have only 1 class. No internal class for the node
  - A single node represents **this** object, and we can't destroy yourself
  - In fact, the code is even buggy as there is a way to destroy ourself!

```
void delete_value(int target) {
    if (target == data && !left && !right)
        return; // can't remove root in this structure
    delete_node(target, this);
}
```

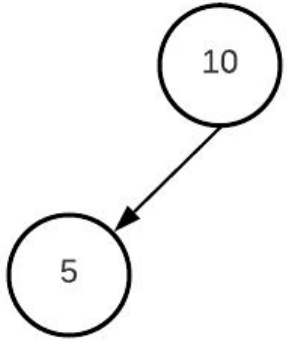
Delete 10

- Please trace the code to delete 10
- What is wrong?!



Deleting this!

- This is 2 nodes tree. Assume we want to delete 10
- Starting from the main code, we are this
  - Now we call this->delete
- 10 has single left child
- According to the code, we delete node(10) and return node(5)
- But node(10) is the original caller! The this
- This function will destroy itself!
- This is another design issue
- How to handle?!




```
else if (!node->right)  // case 2: has left only
    node = node->left;      // connect with child
```

Delete always my child

- We will write a simple function that takes the intended child to return
- Then perform a virtual deletion

```
else {
    if (!node->left && !node->right)    // case 1: no child
        node = nullptr;
    else if (!node->right)  // case 2: has left only
        node->special_delete(node->left);       // connect with child
    else if (!node->left)   // case 2: has right only
        node->special_delete(node->right);
    else {  // 2 children: Use successor
        BinarySearchTree* mn = node->right->min_node();
        node->data = mn->data;  // copy & go delete
        node->right = delete_node(node->data, node->right);
    }
}
```

Virtual delete

- Instead of deleting node and return its child (left/right)
- We will copy the child data to me first (data / pointers)
- Then delete the child itself
- This way we guarantee the this is never deleted!

```
void special_delete(BinarySearchTree* child) {
    // Instead of deleting node and return its child
    // copy child data and remove him
    data = child->data;
    left = child->left;
    right = child->right;
    delete child;
}
```

Rewriting Binary Search Tree

- Early in Binary Trees, our design was based on a single class (not 2)
  - That was very nice in coding, but we faced 2 issues in deletion
  - Tree must have at least one node. You must create. You can't destroy
  - We found deletion could be tricky as it might need to delete this
  - Later in AVL tree, it will force us during rotation for serious copy operation
  - Observe: how a **design choice**, which seemed great early can be source of pain later
- Now, it is time to return go back to the linked list coding style
  - I hope we learned the sensitivity of issues
  - In industry, we should think in as many as possible scenarios for our design effect
    - Clever seniors usually has pretty good sense of what might be a wrong path
- @Homework: You will rewrite the BST

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."