Practice Quiz Out of 33 Marks

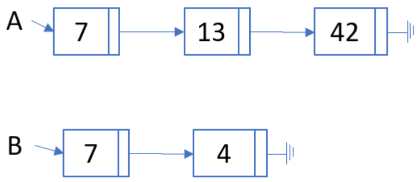
Linked Lists

Suppose head pointer `A` points to the first node of a singly-linked list, and header pointer `B` points to the first node of a different singly-linked list. If a list is empty, then its head pointer is `nullptr`.

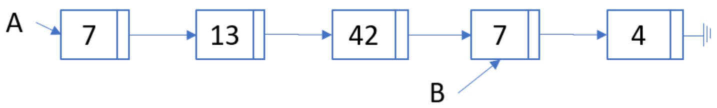
Nodes have this type:

```
struct Node {
    int data;
    Node* next;
};
```

Part A. (4 marks) Assuming A and B are both initially `nullptr`, write C++ code that makes lists `A` and `B` like this:



Part B. (6 marks) Write C++ code that starts with the lists above and appends `B` to the end of `A` so it looks like this:



Note that no nodes are added or removed. **Important**: Your code should work for *any* lists `A` and `B`, not just the particular ones in this question.

Part C. (4 marks) Write C++ code that correctly de-allocates all the nodes so there are no memory leaks or other problems.

Sample Solution

```
//
// Part A
//
Node *A = nullptr; // 7 13 42
A = new Node{42, A};
A = new Node{13, A};
A = new Node{7, A};

Node *B = nullptr; // 7 4
B = new Node{4, B};
B = new Node{7, B};

//
// Part B: append B to A
//
if (A == nullptr) {
    A = B;
} else {
    Node *lastA = A;
    while (lastA->next != nullptr) {
        lastA = lastA->next;
    }
    lastA->next = B;

}

//
// Part C: de-allocate all the nodes
//

while (A != nullptr) {
    Node *temp = A;
    A = A->next;
    delete temp;
}
```

Marking Scheme

- **Part A**
  - **+4 marks**: 2 marks each for correctly initializing each list
- **Part B**
  - **+2 marks**: correctly handling case when A is empty
  - **+4 marks**: correctly handling case when A is not empty (approximately 1 mark per line of the sample solution)
- **Part C**
  - **+4 marks**: correctly de-allocating all nodes
  - **-2 marks** (or more) if nodes are deleted more than once
- **+1 mark**: correct and sensible use of C++ features

Up to -1 mark deducted for each case where the code is very inefficient, or does anything unnecessary.

Notes

- **C++-like pseudocode is okay**. It's okay if little syntactic details are missing from the code, as long as it is clear what the student means, and what they write can be easily translated to C++.
- Sometimes answers can be different than what the marking scheme expects, and in that case you may need to "wing it". In such cases, first decide if it is a failing or passing answer. If it's failing, don't give more than 50%.

Binary Trees

(8 marks) Write a function that returns the number of leaf (external) nodes in a binary tree. Use detailed C++-like pseudocode.

Assume `T` points to the root node, and if the tree is empty then `T` is `nullptr`. Nodes are based on this struct:

```
struct Node {
   int data;
   Node* left;
   Node* right;
};
```

Solution

```
def count_leaves(Node* p)
    if p == nullptr then
        return 0
    else if p->left == nullptr and p->right == nullptr then
        return 1
    else
        return count_leaves(p->left) + count_leaves(p->right)
```

Marking Scheme

- **+2 marks**: correctly recognizing handling empty tree
- **+2 marks**: correctly recognizing and handling case when `p` points to a root
- **+3 marks**: correctly handling the other cases
- **+1 mark**: correct and sensible use of C++ features

Up to -1 mark deducted for each case where the code is very inefficient, or does anything unnecessary.

Notes

- **C++-like pseudocode is okay**. It's okay if little syntactic details are missing from the code, as long as it is clear what the student means, and what they write can be easily translated to C++.
- Sometimes answers can be different than what the marking scheme expects, and in that case you may need to "wing it". In such cases, first decide if it is a failing or passing answer. If it's failing, don't give more than 50%.

Binary Search Trees

Part A. (3 marks) Give the definition of a **binary search tree** (**BST**). Assume unique keys.

Solution

A binary search tree is a binary tree where each internal node has a unique key that can be compared using `<`, and for every node `p` of the tree:

- All keys in the left sub-tree of `p` are less the `p`'s key
- All keys in the right sub-tree of `p` are greater the `p`'s key

Marking Scheme

- **+1 mark**: for mentioning keys can be compared with <
- **+2 marks**: for a correct definition; read the definition and judge it holistically, and give it marks proportional to its correctness

Up to -1 mark deducted for each instance of something unnecessary.

Part B. (4 marks) Give the definition of an **AVL tree**. Assume unique keys.

Solution

An AVL tree is a binary search tree that satisfies the **height-balance property**. A tree satisfies the height-balance property if for every node `p` with a key, the heights of the children of `p` differ by at most 1.

Marking Scheme

- **+1 mark**: saying it's a BST
- **+3 marks**: for a correct definition of height-balance property

Up to -1 mark deducted for each instance of something unnecessary.

Part C. (4 marks) *Prove* or *dis-prove* the following:

> Suppose r is the root node of an AVL tree. If the left sub-tree of r is non-empty, then that sub-tree is also an AVL tree.

Write your proof in answer, readable English that would make your math teacher proud.

Solution

The statement is true.

First, both sub-trees of r are also BSTs. By the definition of a BST, each sub-tree is also a BST.

Second, call a node *bad* if the heights of its children differ by more than 1. A BST with no bad nodes is thus AVL trees. For the sake of contradiction, suppose the left sub-tree of r is *not* an AVL tree. That means it has at least one bad node somewhere in it. Since every node in that sub-tree is also in the original tree, then the original tree must have a bad node, implying it is not an AVL tree. This is a contradiction, and so the left sub-tree must be an AVL tree.

Marking Scheme

- **+1 mark**: mentioning that both sub-trees are BSTs
- **+3 marks**: *Clearly* explaining why the left subtree of an AVL tree is also AVL. This can be done in various ways, so read the answer carefully.

0 marks if claimed the statement is false.

Up to -1 mark deducted for each instance of something unnecessary.