Algorithms and Data Structures: Homework 9

Due on April 17th, 2023, 23:00

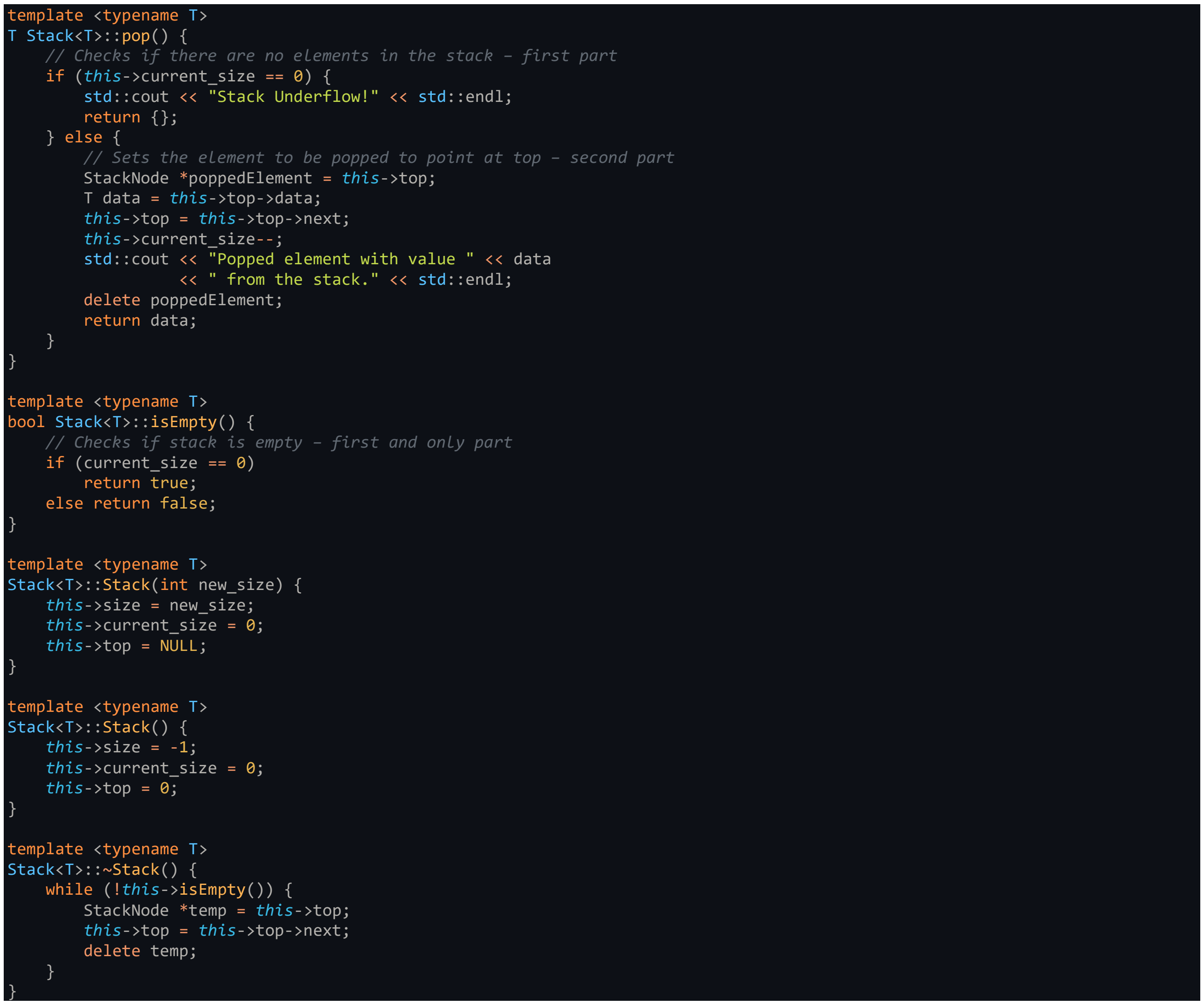
Problem 9.1: Stacks & Queues

(a) Implement using C++, Python or Java the data structure of a stack backed up by a linked list, that can store data of any type, and analyze the running time of each specific operation. Implement the stack such that you have the possibility of setting a fixed size but not necessarily have to (size should be -1 if unset). Your functions should print suggestive messages in cases of underflow or overflow. You can assume that if the size is passed, it will have a valid value.

```cpp
#include <iostream>
using namespace std;
template <typename T>
class Stack {
    private:
        struct StackNode {      // linked list
            T data;
            StackNode *next;
        };
        StackNode *top;         // top of stack
        int size;               // -1 if not set, value otherwise
        int current_size;       // unused if size = -1
    public:
        void push(T x);         // if size set, check for overflow
        T pop();                // return element if not in underflow
        bool isEmpty();         // true if empty, false otherwise
        Stack(int new_size);    // Parametrized constructor
        Stack();                // Constructor
        ~Stack();               // Destructor
};




template <typename T>
void Stack<T>::push(T x) {
    // Checks if the stack is full - first part
    if (this->current_size == this->size)
        std::cout << "Stack Overflow!" << std::endl;
    else {
        this->current_size += 1;
        if (this->current_size == 0) {          // - second part
            StackNode *newElement = new StackNode();
            newElement->data = x;
            // Set next field of newElement to NULL as it is the only element
            newElement->next = NULL;
            this->top = newElement;
            std::cout << "Push element with value " << this->top->data
                      << " onto the stack." << std::endl;
        } else {
            StackNode *newElement = new StackNode();    // - third part
            // Set next pointer of newElement to current top
            newElement->next = top;
            newElement->data = x;
            this->top = newElement;
            std::cout << "Push element with value " << this->top->data
                      << " onto the stack." << std::endl;
        }
    }
}
```



```cpp
template <typename T>
T Stack<T>::pop() {
    // Checks if there are no elements in the stack - first part
    if (this->current_size == 0) {
        std::cout << "Stack Underflow!" << std::endl;
        return {};
    } else {
        // Sets the element to be popped to point at top - second part
        StackNode *poppedElement = this->top;
        T data = this->top->data;
        this->top = this->top->next;
        this->current_size--;
        std::cout << "Popped element with value " << data
                  << " from the stack." << std::endl;
        delete poppedElement;
        return data;
    }
}

template <typename T>
bool Stack<T>::isEmpty() {
    // Checks if stack is empty - first and only part
    if (current_size == 0)
        return true;
    else return false;
}

template <typename T>
Stack<T>::Stack(int new_size) {
    this->size = new_size;
    this->current_size = 0;
    this->top = NULL;
}

template <typename T>
Stack<T>::Stack() {
    this->size = -1;
    this->current_size = 0;
    this->top = 0;
}

template <typename T>
Stack<T>::~Stack() {
    while (!this->isEmpty()) {
        StackNode *temp = this->top;
        this->top = this->top->next;
        delete temp;
    }
}
```

Stack Implementation in C++ using Linked List

Run-time analysis:

- Push (T x):

The first part has a constant run-time $O(1)$, as in case when the Stack is full The second one has a constant run-time $O(1)$ as in case when the Stack is empty, the algorithm has to create a new element with data x and set the top of the Stack to refer to this new element,, constant time operations. The third part has a constant run-time $O(1)$ as the only difference from the second branch is just setting the next field of the new element to be the old top node of the Stack. Therefore, the run-time of Push is $O(1)$.

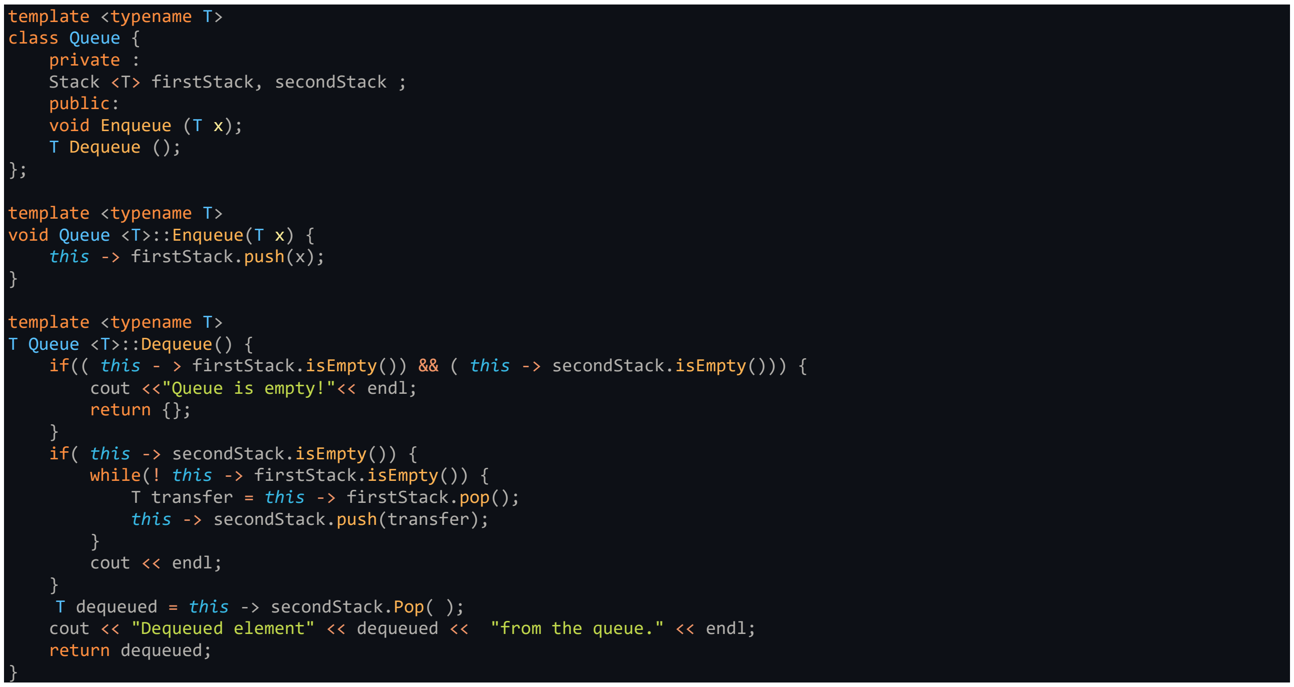
- Pop():

The first one has a constant run-time $O(1)$ as in case when the Stack is empty. The second part of the conditional has a constant run-time $O(1)$ as the algorithm has to save a pointer pointing to the current top and move the top to point to the next field of itself, constant time. Therefore, the run-time of Pop() is $O(1)$.

- IsEmpty():

The run-time of the IsEmpty is constant, $O(1)$. The algorithm has to perform just a comparison between the current size field and 0 to check if the Stack is empty or not.

(b) Implement a queue which uses two stacks to simulate the queue behavior:



```
template <typename T>
class Queue {
    private :
    Stack <T> firstStack, secondStack ;
    public:
    void Enqueue (T x);
    T Dequeue ();
};

template <typename T>
void Queue <T>::Enqueue(T x) {
    this -> firstStack.push(x);
}

template <typename T>
T Queue <T>::Dequeue() {
    if(( this - > firstStack.isEmpty()) && ( this -> secondStack.isEmpty())) {
        cout <<"Queue is empty!"<< endl;
        return {};
    }
    if( this -> secondStack.isEmpty()) {
        while(! this -> firstStack.isEmpty()) {
            T transfer = this -> firstStack.pop();
            this -> secondStack.push(transfer);
        }
        cout << endl;
    }
     T dequeued = this -> secondStack.Pop( );
    cout << "Dequeued element" << dequeued <<  "from the queue." << endl;
    return dequeued;
}
```

Queue Implementation in C++ using two Stacks

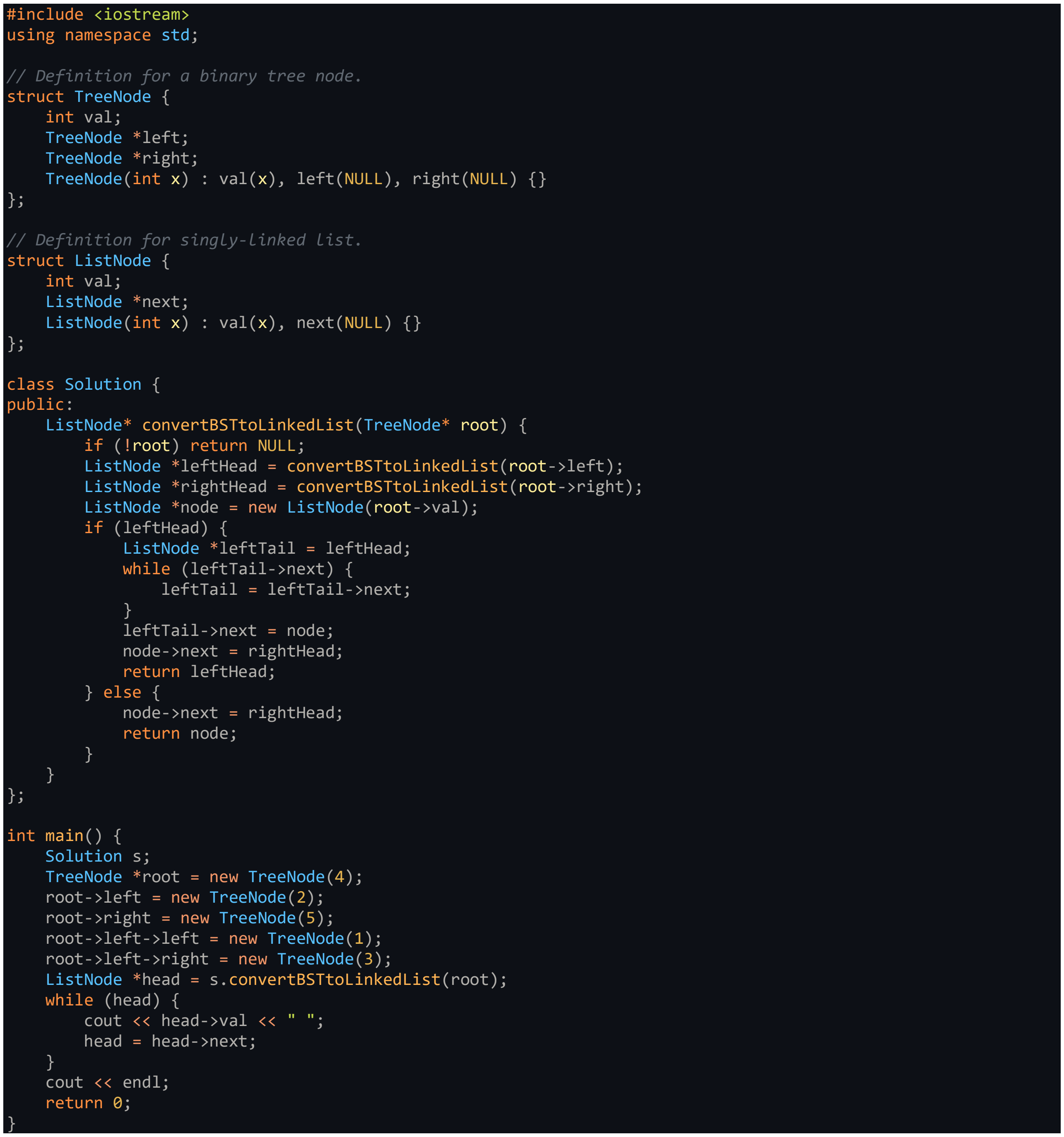
Problem 9.2: Linked Lists & Rooted Trees

a) Write down the pseudocode for an in-situ algorithm that reverses a linked list of n elements in $\Theta(n)$. Explain why it is an in-situ algorithm:

```
Algorithm   Reverse_linked_list( List L )
    if L.head == NIL then
        return list                // Check if the list is empty
    end if
    Node previous = NIL
    Node current = L.head
    Node next = NIL
    while current != NIL do
        next = current.next        // Set next to the next node of the current node
        current.next = previous    // Reverse next pointer of current node to point to previous node
        previous = current         // Set previous for the next iteration to current node
        current = next             // Move onto the next node of the list
    end while
    return previous                // Return last element of the old order of the list
```

The algorithm is an in-situ algorithm as it only uses constant space, $O(1)$, to reverse the order of the elements in a list. It uses only 3 pointers to point to nodes of the list, previous, current and next.

b) Implement an algorithm to convert a binary search tree to a sorted linked list and derive its asymptotic time complexity:



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

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

class Solution {
public:
    ListNode* convertBSTtoLinkedList(TreeNode* root) {
        if (!root) return NULL;
        ListNode *leftHead = convertBSTtoLinkedList(root->left);
        ListNode *rightHead = convertBSTtoLinkedList(root->right);
        ListNode *node = new ListNode(root->val);
        if (leftHead) {
            ListNode *leftTail = leftHead;
            while (leftTail->next) {
                leftTail = leftTail->next;
            }
            leftTail->next = node;
            node->next = rightHead;
            return leftHead;
        } else {
            node->next = rightHead;
            return node;
        }
    }
};

int main() {
    Solution s;
    TreeNode *root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(5);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);
    ListNode *head = s.convertBSTtoLinkedList(root);
    while (head) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
    return 0;
}
```

Algorithm to convert a BST to a Sorted Linked List implemented in C++

<u>Asymptotic Time Complexity of Conversion from BST to Sorted Linked List</u>:

The basic idea is to recursively convert the left and right subtrees to linked lists and then merge them with the current node. If there is a left subtree, we find the tail of its linked list and append the current node to it, then append the right subtree to the current node.

Otherwise, we just append the right subtree to the current node.
The time complexity of this algorithm is O(n), where n is the number of nodes in the binary search tree. This is because each node is visited once, and each operation takes O(1) time.

c) Implement an algorithm to convert a sorted linked list to a binary search tree and derive its asymptotic time complexity. The search time complexity of the resulting binary search tree should be lower than the one for the equivalent sorted linked list.

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

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        if (!head) return NULL;
        vector<int> nums;
        while (head) {
            nums.push_back(head->val);
            head = head->next;
        }
        return buildBST(nums, 0, nums.size() - 1);
    }

    TreeNode* buildBST(vector<int>& nums, int start, int end) {
        if (start > end) return NULL;
        int mid = start + (end - start) / 2;
        TreeNode *root = new TreeNode(nums[mid]);
        root->left = buildBST(nums, start, mid - 1);
        root->right = buildBST(nums, mid + 1, end);
        return root;
    }

    void printTree(TreeNode* root) {
        if (!root) return;
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                TreeNode *node = q.front();
                q.pop();
                cout << node->val << " ";
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
            cout << endl;
        }
    }
};
```

```
int main() {
    Solution s;
    ListNode *head = new ListNode(-10);
    head->next = new ListNode(-3);
    head->next->next = new ListNode(0);
    head->next->next->next = new ListNode(5);
    head->next->next->next->next = new ListNode(9);
    TreeNode *root = s.sortedListToBST(head);
    s.printTree(root);
    return 0;
}
```

Asymptotic Time Complexity of Conversion from a Sorted Linked List to a BST:

The basic idea is to first convert the linked list to a vector and then recursively build a binary search tree from the vector. To build the binary search tree, we find the middle element of the vector and make it the root of the current subtree. We then recursively build the left and right subtrees using the left and right halves of the vector, respectively.
The time complexity of this algorithm is O(n), where n is the number of nodes in the linked list. This is because we visit each node once to convert the linked list to a vector, and then each node is visited once again to build the binary search tree. The search time complexity of the resulting binary search tree is O(log n), which is lower than the O(n) search time complexity of the equivalent sorted linked list.