CH08-320143: Assignment #8

Harit Krishan
h.krishan@jacobs-university.de

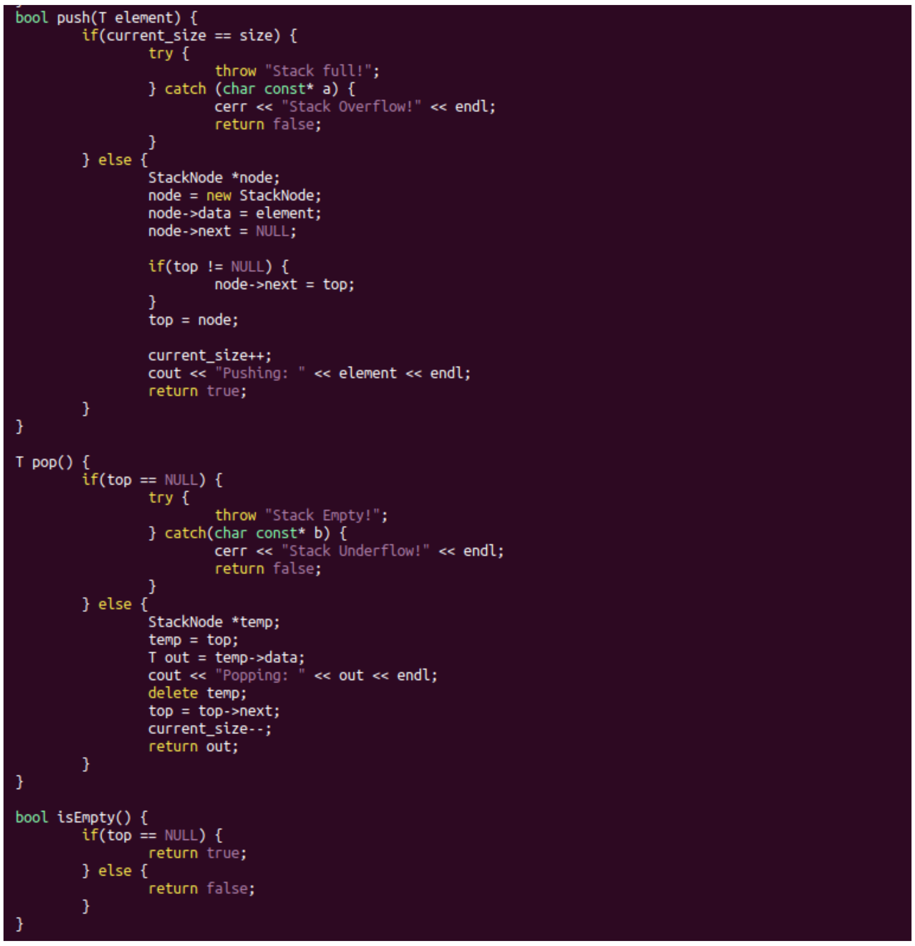
Jacobs University Bremen — March 30, 2020

1 Stacks and Queues

A stack is a data structure which supports a last-in-first-out (LIFO) file system. A queue is a data structure which supports a first-in-first-out file system.

1.1 Linked-List Backed Stack

It is possible to create a stack which is backed up by a linked-list. The implementation of this data structure can be found in *stack.h* in the root directory of this submission. There are three main member functions of this class - $push()$, $pop()$ & $isEmpty()$. The following image shows the function definitions of the respective functions in C++.



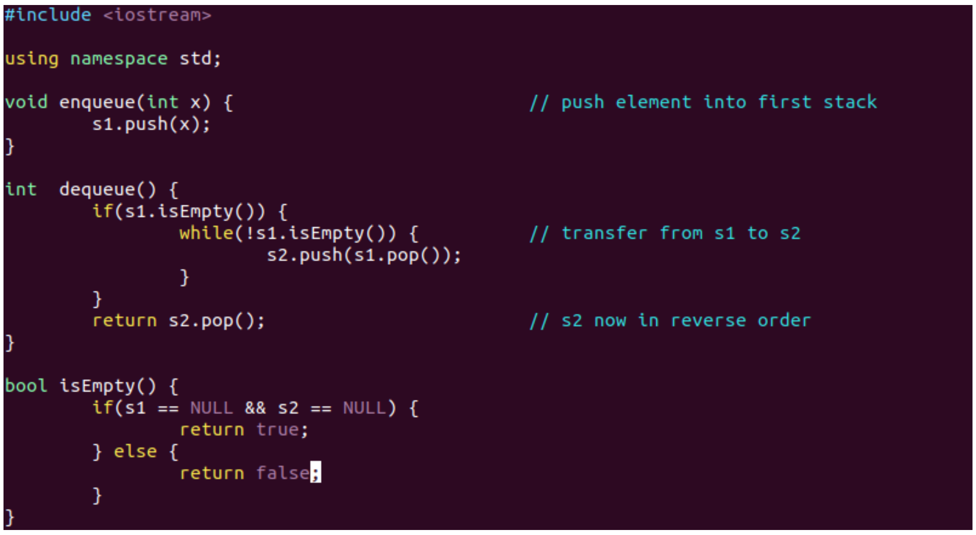
```
bool push(T element) {
        if(current_size == size) {
                try {
                        throw "Stack full!";
                } catch (char const* a) {
                        cerr << "Stack Overflow!" << endl;
                        return false;
                }
        } else {
                StackNode *node;
                node = new StackNode;
                node->data = element;
                node->next = NULL;

                if(top != NULL) {
                        node->next = top;
                }
                top = node;

                current_size++;
                cout << "Pushing: " << element << endl;
                return true;
        }
}

T pop() {
        if(top == NULL) {
                try {
                        throw "Stack Empty!";
                } catch(char const* b) {
                        cerr << "Stack Underflow!" << endl;
                        return false;
                }
        } else {
                StackNode *temp;
                temp = top;
                T out = temp->data;
                cout << "Popping: " << out << endl;
                delete temp;
                top = top->next;
                current_size--;
                return out;
        }
}

bool isEmpty() {
        if(top == NULL) {
                return true;
        } else {
                return false;
        }
}
```

Figure 1: Function Definitions for Stack in C++

The time complexity for $push()$ is $O(1)$ since adding a new element to the stack is a constant timed operation. The function $pop()$ has time complexity $O(1)$. Popping an element from the stack takes constant time also regardless of the current position loaded from the stack. $isEmpty()$ checks if the stack is empty by verifying the value of the top pointer. This operation also takes constant time, therefore $O(1)$.

1.2 Stack Backed Queue

It is possible to implement a queue using two stacks such that it simulates the behaviour of a queue. For this to happen, the enque and deque operations must be modified. This set of modifications is available in the file $queue_stack.h$. Figure 2 shows a snapshot of this file. We can see the respective modifications for the implementation to work.



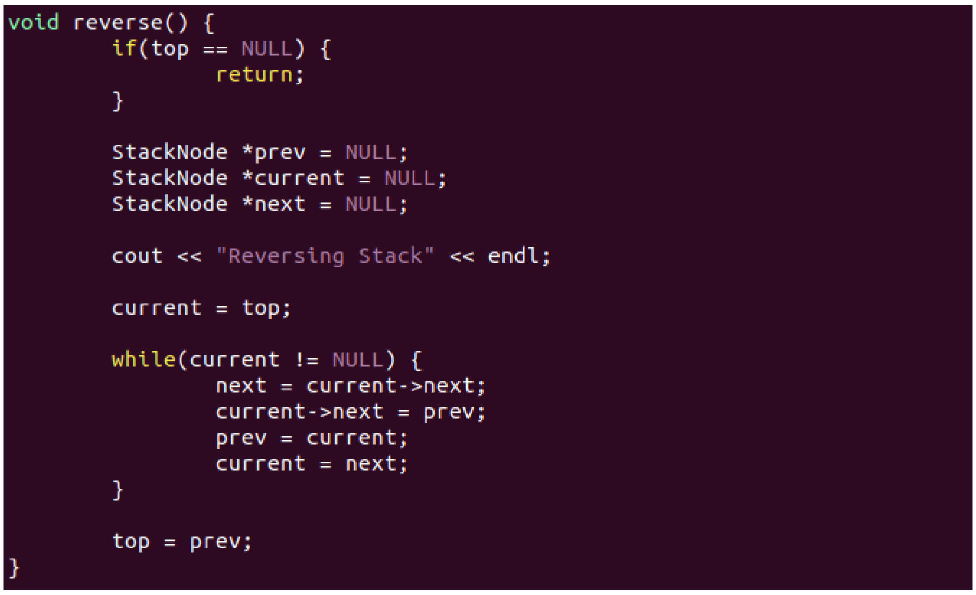
```
#include <iostream>

using namespace std;

void enqueue(int x) {                           // push element into first stack
        s1.push(x);
}

int  dequeue() {
        if(s1.isEmpty()) {
                while(!s1.isEmpty()) {          // transfer from s1 to s2
                        s2.push(s1.pop());
                }
        }
        return s2.pop();                        // s2 now in reverse order
}

bool isEmpty() {
        if(s1 == NULL && s2 == NULL) {
                return true;
        } else {
                return false;
        }
}
```

Figure 2: Modifications of Queue with 2 Stacks

2 Linked Lists and Rooted Trees

2.1 Reversing Linked List

The reverse algorithm is implemented in $stack.h$ as a member function of the stack class. The implementation of this algorithm can be seen in Figure 3.

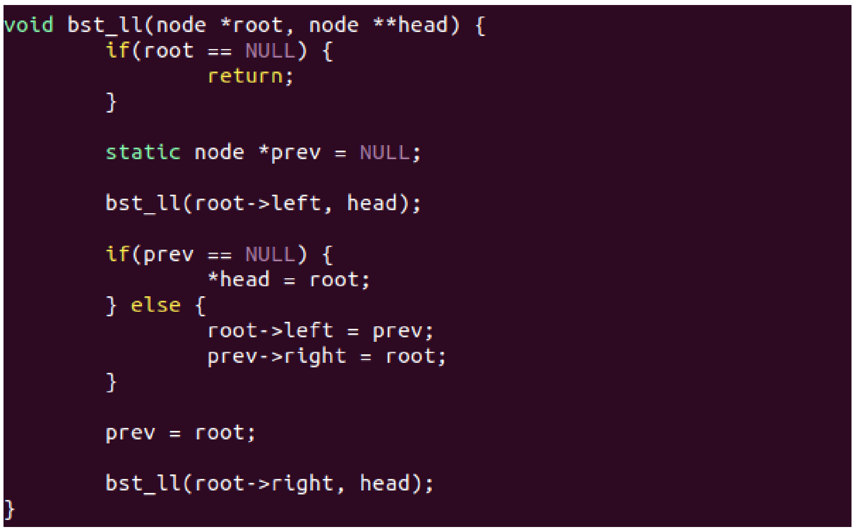


```
void reverse() {
        if(top == NULL) {
                return;
        }

        StackNode *prev = NULL;
        StackNode *current = NULL;
        StackNode *next = NULL;

        cout << "Reversing Stack" << endl;

        current = top;

        while(current != NULL) {
                next = current->next;
                current->next = prev;
                prev = current;
                current = next;
        }

        top = prev;
}
```

Figure 3: Implementation of Reverse Function in Stack Class in C++

2.2 BST to Sorted Linked List

It is possible to convert a binary search tree to a sorted linked list. This function has been implemented in $bst_l l.h$ by the same name. Figure 4 shows a snapshot of this algorithm implementation in C++.



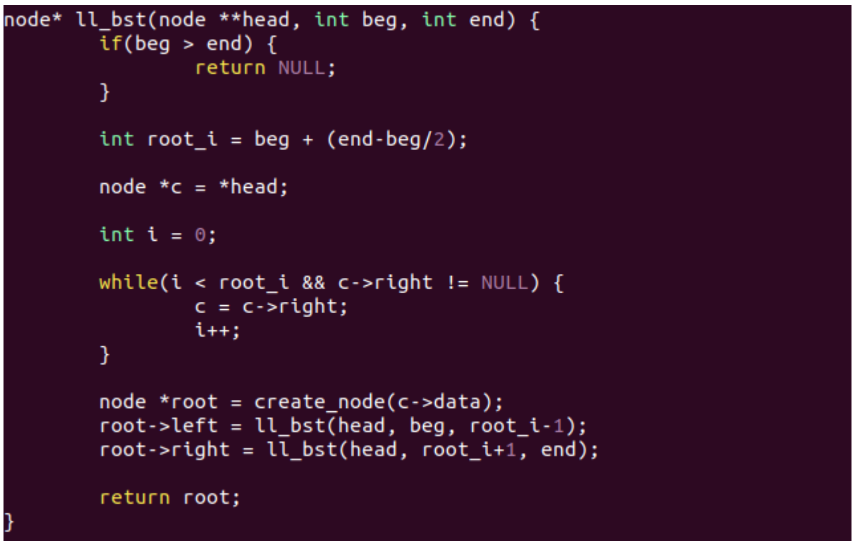
```
void bst_ll(node *root, node **head) {
        if(root == NULL) {
                return;
        }

        static node *prev = NULL;

        bst_ll(root->left, head);

        if(prev == NULL) {
                *head = root;
        } else {
                root->left = prev;
                prev->right = root;
        }

        prev = root;

        bst_ll(root->right, head);
}
```

Figure 4: Implementation of BST to LL Conversion in C++

This algorithm works by recursively calling itself in an ascending order. By ascending order, it is implied that the traversal is in the order or node, left child and right child. This ensures that the elements are stored in order. We take advantage of the BST property of ordering.

2.3 Linked List to BST

It is possible to convert a sorted linked list to a binary search tree. This function has been implemented in $bst_l l.h$ by the name $ll_b st()$. Figure 5 shows a snapshot of this algorithm implementation in C++. A similar principle is followed for this problem like the previous one. We exploit the given data structure's property of ordering and use that to convert the structure.



```
node* ll_bst(node **head, int beg, int end) {
        if(beg > end) {
                return NULL;
        }

        int root_i = beg + (end-beg/2);

        node *c = *head;

        int i = 0;

        while(i < root_i && c->right != NULL) {
                c = c->right;
                i++;
        }

        node *root = create_node(c->data);
        root->left = ll_bst(head, beg, root_i-1);
        root->right = ll_bst(head, root_i+1, end);

        return root;
}
```

Figure 5: Implementation of LL to BST Conversion in C++