Algorithms and Data Structures: Homework #8

Due on March 30, 2020 at 23:00

Henri Sota

Problem 8.1

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.

```
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) {
    // Check if stack is full
    if (this->current_size == this->size)
        std::cout << "Stack Overflow!" << std::endl;
    else {
        this->current_size += 1;
        if (this->current_size == 0) {
            // Else in case there are no elements in the stack
            StackNode *newElement = new StackNode();
            newElement->data = x;
            // Set next field of newElement to NULL as it is the only element
            newElement->next = NULL;
            // Set top to point to newElement as it is the only element
            this->top = newElement;
            std::cout << "Push element with value " << this->top->data
                      << " onto the stack." << std::endl;
        } else {
            StackNode *newElement = new StackNode();
            // Set next pointer of newElement to current top
            newElement->next = top;
            newElement->data = x;
            // Set top to point to newElement as it is the last element in
            this->top = newElement;
            std::cout << "Push element with value " << this->top->data
                      << " onto the stack." << std::endl;
        }
    }
}

template <typename T>
T Stack<T>::pop() {
    // Check if there are no elements in the stack
    if (this->current_size == 0) {
```

```
        std::cout << "Stack Underflow!" << std::endl;
        return this->top->data;
    } else {
        // Set the element to be popped to point at top
        StackNode *poppedElement = this->top;
        T data = this->top->data;
        // Update top to point to next of the current top
        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() {
    // Check if stack is empty
    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;
    }
}
```

Listing 1: Stack Implementation in C++ using Linked List

Run-time analysis:

- Push(T **x**)
  In order to correctly analyze the run-time of the Push procedure, all three logical branches must be analyzed on their own. The first one has a constant run-time $O(1)$ as in case when the Stack is full, the algorithm has to perform a simple constant time comparison between the current_size and the size of the Stack to determine if it 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, which altogether are constant time operations. The third one 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()
  In order to correctly analyze the run-time of the Pop procedure, the two branches of the conditional must be analyzed on their own. The first one has a constant run-time $O(1)$ as in case when the Stack is empty, the algorithm has to perform a simple constant time comparison between the current_size and 0 to determine if it is empty. The second branch 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. These operations take constant time. Therefore, the run-time of Pop() is $O(1)$.
- IsEmpty()
  The run-time of the IsEmpty procedure 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.
In order to simulate the behavior of a queue using two Stacks, elements that get enqueued will get pushed onto the first Stack. If these elements were to be popped directly from the first Stack, they would be returned into the reverse order they were entered due to the LIFO methodology of the Stack. To avoid this, we pop each element off the first Stack and push them into the second Stack. When popping elements from the second Stack, the order in which they get returned is the reverse of the order which they entered the second Stack and the same order they entered the first Stack. This simulates the behavior of the queue as the elements are dequeued in the same order they were enqueued.

```
template <typename T>
class Queue {
    private:
        Stack<T> firstStack, secondStack;
    public:
        void Enqueue(T x);
        T Dequeue();
        // Implementation of the constructor, parametric constructor
        //  and destructor is not necessary in this case
        // Queue(int new_size);
        // Queue();
        // ~Queue();
};

template <typename T>
void Queue<T>::Enqueue(T x) {
    // Push enqueued elements onto the first stack
    std::cout << "First Stack: ";
    this->firstStack.push(x);
}

template <typename T>
T Queue<T>::Dequeue() {
    // Check if queue is empty
    if ((this->firstStack.isEmpty()) && (this->secondStack.isEmpty())) {
        std::cout << "Queue is empty!" << std::endl;
        return {};
    }
    // Check if the second stack is empty
    if (this->secondStack.isEmpty()) {
        std::cout << "Transferring elements from stacks!" << std::endl;
        // Pop all elements of the first stack and push them onto the second
        // stack, thus reversing the order of the elements that entered in the
        // first stack
        while (!this->firstStack.isEmpty()) {
```

```
            T transfer = this->firstStack.pop();
            this->secondStack.push(transfer);
        }
        std::cout << std::endl;
    }
    // Pop one element from the second stack
    T dequeued = this->secondStack.pop();
    std::cout << "Dequeued element " << dequeued << " from the queue."
              << std::endl;
    return dequeued;
}
```

Listing 2: Queue Implementation in C++ using two Stacks

Problem 8.2

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.
The algorithm runs in $\Theta(n)$ time, because it has to go through the whole list of elements and change the pointers to the next node to refer to the previous node in the old order. This is reflected by the loop which stops until it meets the end of the list. a NIL element.

Algorithm 1 Reverse Linked List[List list]

```
// Check if there are no elements
if list.head == NIL then
  return list
end if
Node previous = NIL
Node current = list.head
Node next = NIL
while current != NIL do
  // Set next to the next node of the current node
  next = current.next
  // Reverse next pointer of current node to point to previous node
  current.next = previous
  // Set previous for the next iteration to current node
  previous = current
  // Move onto the next node of the list
  current = next
end while
// Return last element of the old order of the list
return previous
```

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 3 pointers to point to nodes of the list, previous, current and next. While traversing the list, it saves the next field of the current node being traversed, it changes the next field of current node to point to the previous node, it sets the previous node to the current element and sets the current element to the next element in the old order.

b) Implement an algorithm to convert a binary search tree to a sorted linked list and derive its asymptotic time complexity.
This problem has been implemented by creating 2 classes, one for the Binary Search Tree and one for the Linked List. The Binary Search Tree class has functions to insert an element, to print itself and to convert itself to a Linked List. The Linked List class has functions to push an element onto itself and to print itself. The main function creates a Binary Search Tree from an array of elements, prints the Binary Search Tree using inorder traversal, converts the BST to a Linked List and prints the Linked List.

```
class TreeNode {
    public:
        int data;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int data);
};

TreeNode::TreeNode(int data) {
    this->data = data;
    this->left = NULL;
    this->right = NULL;
}

class BinarySearchTree {
    public:
        TreeNode *root;
        BinarySearchTree();

        void insert(int data);
        void insertUnderRoot(TreeNode *root, int data);
        void toSortedLinkedList(LinkedList &list);
        void toSortedLinkedListUnderRoot(LinkedList &list, TreeNode *root);
        void print();
        void printUnderRoot(TreeNode *root);
};

BinarySearchTree::BinarySearchTree() {
    this->root = NULL;
}

void BinarySearchTree::insert(int data) {
    TreeNode *newElement = new TreeNode(data);
    // Check if root of the tree is NULL
    if (this->root == NULL)
        // Set root to the newElement
        this->root = newElement;
    else
        // Call the recursive insert function to place the element into the tree
        this->insertUnderRoot(this->root, data);
    std::cout << "Inserted element " << data
              << " into Binary Search Tree!" << std::endl;
}

void BinarySearchTree::insertUnderRoot(TreeNode *root, int data) {
    // Check if the element we are trying to place is bigger than the root
    if (data > root-> data) {
        // In case it is bigger, check if the right branch of root is empty
        if (root->right == NULL) {
            // Insert the element into the right branch of the root
            root->right = new TreeNode(data);
        } else {
```

```
            // Try to insert the element into the right subtree
            this->insertUnderRoot(root->right, data);
        }
    } else {
        // In case it is smaller or equal, check if the left branch of root is
        // empty
        if (root->left == NULL) {
            // Insert the element into the left branch of the root
            root->left = new TreeNode(data);
        } else {
            // Try to insert the element into the left subtree
            this->insertUnderRoot(root->left, data);
        }
    }
}

void BinarySearchTree::toSortedLinkedList(LinkedList &list) {
    // Check if root is empty
    if (this->root == NULL)
        std::cout << "Binary Search Tree has no elements inside!" << std::endl;
    else
        // Call the recursive transfer function to move the elements from the
        // BST to a Sorted Linked List
        this->toSortedLinkedListUnderRoot(list, this->root);
}

void BinarySearchTree::toSortedLinkedListUnderRoot(LinkedList &list,
                                                   TreeNode *root) {
    // Call itself recursively with the right subtree if it exists
    if (root->right != NULL)
        toSortedLinkedListUnderRoot(list, root->right);
    // Push the element at the root into the list
    list.pushList(root->data);
    // Call itself recursively with the left subtree if it exists
    if (root->left != NULL)
        toSortedLinkedListUnderRoot(list, root->left);
}

void BinarySearchTree::printUnderRoot(TreeNode *root) {
    // Print the elements of the left subtree if it exists
    if (root->left != NULL)
        printUnderRoot(root->left);
    // Print the data field of the root element
    std::cout << " " << root->data;
    // Print the elements of the right subtree if it exists
    if (root->right != NULL)
        printUnderRoot(root->right);
}

void BinarySearchTree::print() {
    // Check if Binary Search Tree is empty
    if (this->root != NULL) {
        std::cout << "Binary Search Tree:";
        // Call the recursive function for the root of the Binary Search Tree
        printUnderRoot(this->root);
        std::cout << std::endl;
    } else
        std::cout << "Binary Search Tree has no elements inside!" << std::endl;
}
```

Listing 3: Binary Search Tree Class implemented in C++

```
class ListNode {
    public:
        int data;
        ListNode *next;
        ListNode(int data);
        ListNode();
};

ListNode::ListNode(int data) {
    this->data = data;
    this->next = NULL;
}

ListNode::ListNode() {
    this->data = 2147483647; // Set it to INT_MAX value
    this->next = NULL;
}

class LinkedList {
    public:
        ListNode *head;
        LinkedList();
        void pushList(int data);
        void printList();
        void reverseList();
};

LinkedList::LinkedList() {
    this->head = NULL;
}

void LinkedList::pushList(int data) {
    ListNode *newElement = new ListNode(data);
    // Check if list is empty
    if (this->head != NULL)
        // Prepend the element before the start of the list
        newElement->next = this->head;
    else {} // Do nothing
    // Make newElement the new head of the list
    this->head = newElement;
    std::cout << "Pushed element " << data << " into the Linked List!"
              << std::endl;
}

void LinkedList::printList() {
    ListNode *cursor = this->head;
    std::cout << "Sorted Linked List: ";
    // Print the elements of the linked list by traversing the list
    while (cursor != NULL) {
        if (cursor == this->head)
            std::cout << cursor->data;
        else std::cout << " " << cursor->data;
        cursor = cursor->next;
    }
    std::cout << std::endl;
}
```

Listing 4: Linked List Class implemented in C++

```
int main() {
    int elements[10] = {4, 12, 9, 2, 13, 19, 21, 8, 8, 15};
    BinarySearchTree bst;
```

```
    LinkedList list;

    // The elements inside the Binary Search Tree
    for (auto element : elements)
        bst.insert(element);

    // Print the elements of the Binary Search Tree
    bst.print();

    bst.toSortedLinkedList(list);

    // Printing the elements of the list
    list.printList();

    return 0;
}
```

Listing 5: Main Function to convert a BST to a Sorted Linked List implemented in C++

Asymptotic Time Complexity of Conversion Procedure from BST to Sorted Linked List
The time complexity of the conversion procedure is linear, $O(n)$, where n is the number of elements in the Binary Search Tree. Due to the trait of the reverse inorder tree traversal, which returns the right child (if a subtree, then reads the subtree inorder), the root, and then the left child of a tree (if a subtree, then reads the subtree inorder), the algorithm has to just read all of the elements in order to convert/move them. Since this is a Binary Search Tree, reverse inorder tree traversal will yield a decreasingly sorted array of elements. This decreasingly sorted array will then be pushed onto the Linked List one by one. Therefore, this is linear time, as pushing an element onto the Linked List takes constant time and for n elements takes $O(n)$. In this implementation, "reverse" inorder traversal is used to yield a decreasingly sorted array, because after pushing all the elements onto the Linked List, they will be in the reverse order they came in, which means that the elements will be increasingly sorted in the end.