Algorithms and Data Structures: Homework #8

Due on March 30, 2020 at 23:00

Henri Sota

Problem 8.2

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.
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 and to print itself. The Linked List class has functions to push an element onto itself, to print itself, to count the number of ListNodes that are inside itself and to convert itself to a Binary Search Tree.

```
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 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::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;
}

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);
}
```

Listing 1: 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();
        int countListNodes();
        TreeNode* toBST();
        TreeNode* toBSTUnderRoot(ListNode **head, int n);
};

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;
}

int LinkedList::countListNodes() {
    int count = 0;
    ListNode *cursor = this->head;
    // Traverse through the list and count the number of elements
    while (cursor != NULL) {
        count++;
        cursor = cursor->next;
    }
    return count;
}

TreeNode* LinkedList::toBST() {
    // Calculate the number of ListNodes in the Linked List
    int n = this->countListNodes();

    // Construct the Binary Search Tree and return its root TreeNode
    return toBSTUnderRoot(&this->head, n);
}

TreeNode* LinkedList::toBSTUnderRoot(ListNode **head, int n) {
    if (n <= 0)
        return NULL;

    // Construct the left subtree by calling itself
    TreeNode *left = toBSTUnderRoot(head, n/2);
    // Set the root of the tree
    TreeNode *root = new TreeNode((*head)->data);
    // Connect the current root to its left subtree
    root->left = left;
    // Move on onto the next element
    *head = (*head)->next;
    // Construct the right subtree by calling itself and connect it to the root
    root->right = toBSTUnderRoot(head, n - n / 2 - 1);

    return root;
}
```

Listing 2: Linked List Class implemented in C++

```
int main() {
    int elements[10] = {42, 33, 27, 24, 17, 14, 9, 6, 4, 2};
    LinkedList list;
    BinarySearchTree bst;

    // Push elements sorted increasingly into the list
    for (auto element : elements)
        list.pushList(element);

    // Print the linked list
    list.printList();

    TreeNode *root = list.toBST();
    bst.root = root;

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

    return 0;
}
```

Listing 3: Main Function to convert a BST to a Sorted Linked List implemented in C++

Asymptotic Time Complexity of Conversion Procedure from Sorted Linked List to BST
The time complexity of the conversion procedure is $\Theta(n)$, where n is the number of elements in the Linked List. The other techniques have time complexities of $O(n \log n)$ or $O(n^2)$ as they need to traverse through the Linked List, which takes linear time for each element and build the BST from the root to the leaves. This technique works by building the Binary Search Tree from the leaves to the root. It recursively creates both subtrees by going down to the leaf level (by checking against the base case) and from there it creates the root for that subtree, connects it to the left child and the right child and so on it goes for the levels above it. This technique runs in $O(n)$, because it only has to through each element only once in the Linked List.
Asymptotic Time Complexity of BST Search
The time complexity of search procedure in this BST is $O(\log n)$, because this is a balanced binary tree. The BST search algorithm is exactly the binary search algorithm that is performed on an array, but with links and pointers that act as connections to the middle of the left subarray and to the middle of the right subarray. Each step of this search procedure removes half of the current tree nodes that are to be considered as a candidate in the search. In the worst case, the search procedure has to go from the root to a leaf level, when the element is not in the tree, which yields a $O(\log n)$. Therefore, it is lower than the time complexity of search procedure, $O(n)$, in a Sorted Linked List, as the search algorithm has to go through most of the elements in the average and all elements in the worst case to find the element that it is searching for.

---

* Implementation has been based upon the code found on this website.