Assignments week 4

Son Cao

2025-09-23

Assignment 1 - Maintaining a sorted tree

My code:

```
#include <iostream>
#include <string>
#include "bintree.h"

sax::binary_tree_node<int>* insert(sax::binary_tree_node<int>* node, int x){
    if(!node){
        return new sax::binary_tree_node<int>{x, nullptr, nullptr};
    }

    if(x < node->data){
        node->left = insert(node->left, x);
    } else if (x > node->data){
        node->right = insert(node->right, x);
    }
    //if x == node->data
    return node;
}

//remove smallest node from subtree

sax::binary_tree_node<int>* remove_min(sax::binary_tree_node<int>* node, int &min_val){
    if(node->left == nullptr){
        //this node is a min
        min_val = node->data;
        sax::binary_tree_node<int>* right_subtree = node->right;
        delete node;
        return right_subtree;
    }
    node->left = remove_min(node->left, min_val);
    return node;
```

```
}

//remove x  from BST
sax::binary_tree_node<int>* remove(sax::binary_tree_node<int>* node, int x){
    if(!node) return nullptr;

    if(x < node->data){
        node->left = remove(node->left, x);
    } else if (x > node->data){
        node->right = remove(node->right, x);
    } else {
        //found node to remove

        //leaf node(no children) -> delete
        if(!node->left && !node->right){
            delete node;
            return nullptr;

        } else if (!node->left){ //only right child
            sax::binary_tree_node<int>* tmp = node->right;
            delete node;
            return tmp;
        } else if (!node->right){ //only left child
            sax::binary_tree_node<int>* tmp = node->left;
            delete node;
            return tmp;

        } else {
            //2 children -> replace with inorder successor(smallest value in right subtree)
            int successor_val;
            node->right = remove_min(node->right, successor_val);
            node->data = successor_val;
        }
    }
    return node;
}

int main() {
    /* TODO:
    Write a program that reads a binary search tree with integers from the standard input,
    followed by a command to either insert or delete an integer from the tree.
    The program should then perform the specified operation and print the resulting tree.
    */
   sax::binary_tree_node<int>* root = nullptr;
   std::string op;
   int x;
```

```
    std::cin >> root;
    std::cin >> op >> x;

    if(op == "insert"){
     root = insert(root, x);
    } else if (op == "remove"){
         root = remove(root, x);
    }

    std::cout << root << std::endl;
    sax::binary_tree_node<int>::cleanup(root);
     return 0;
}
```

Time complexity: this algorithm has a time complexity of O(h), because in both the insert and remove operations, we traverse from the root down to a specific node, visiting at most one node per level of the tree.

Assignment 2 - Tree slicing without recursion

My code:

```
#ifndef NODE_PTR_H
#define NODE_PTR_H

#include <stack>    // for std::stack
#include "bintree.h"

namespace sax {

template<typename T>
class node_ptr {
    public:
        //constructor: initialize the smallest element
        explicit node_ptr(binary_tree_node<T>* root){
            push_left_path(root);
        }
        //Arrow operator: access current node's data
        binary_tree_node<T>* operator->(){
            return current;
        }
        //Conversion to bool: true if valid
        operator bool() const{
            return current != nullptr;
        }
```

```
        //Advance to next (in-order successor)
        void move_next(){
            if(!current) return;

            if(current->right){ //if there is a right subtree, go down its leftmost path
                push_left_path(current->right);
            } else {
                // backtrack using the stack
                if(!stk.empty()){
                    current = stk.top();  //top() gives a reference to the element currentl
                    stk.pop();  //to remove
                } else {
                    current = nullptr;
                }
            }
        }

    private:
        binary_tree_node<T>* current = nullptr;
        std::stack<binary_tree_node<T>*> stk;

        //Helper: push path of left children,set current
        void push_left_path(binary_tree_node<T>* node){
            while(node){
                stk.push(node);
                node = node->left;
            }
            if(!stk.empty()){
                current = stk.top();
                stk.pop();
            } else {
                current = nullptr;
            }
        }
};

} //namespace sax


#endif // NODE_PTR_H
```

```
#include <iostream>
#include "bintree.h"
#include "node_ptr.h"

int main()
```

```
{
    /* TODO:
        Write a program that reads a binary search tree from the standard input,
        and prints the values in the tree in reverse sorted order.

        Example input: ((1 3 4) 5 (7 8 ()))
        Example output: 8 7 5 4 3 1

        Use the binary_tree_node struct from bintree.h to represent the tree. This structure
        contains input and output operators that you can use to read (and write) trees.
    */
    sax::binary_tree_node<int>* root = nullptr;
    int k;

    std::cin >> root;
    std::cin >> k;

    //iterate with node_ptr
    sax::node_ptr<int> it(root);
    while (it && it->data <= k){
        std::cout << it->data << " ";
        it.move_next();
    }
    std::cout << std::endl;

    sax::binary_tree_node<int>::cleanup(root);
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because it visits each node in the binary search tree at most once while traversing it in-order using the node_ptr iterator. The move_next() operation advances to the next node in sorted order without revisiting any node, and the stack ensures that each node is pushed and popped exactly once, giving linear time proportional to the number of nodes n.

Assignment 3 - Subs, Sets and Trees

My code:

```
#include <iostream>
#include <string>
#include "bintree.h" // for binary_tree_node
#include "node_ptr.h"

bool is_subset(sax::binary_tree_node<int>* A, sax::binary_tree_node<int>* B){
```

```
    sax::node_ptr<int> itA(A);
    sax::node_ptr<int> itB(B);

    while (itA){
        if(!itB) return false; //if B is exhauted but A not -> false

        if(itA->data == itB->data){  //match -> advance both
            itA.move_next();
            itB.move_next();
        } else if(itA->data > itB->data){
            itB.move_next();  //B smaller -> advance B
        } else {
            return false; //A smaller but not in B => false
        }               //cuz both array is sorted thru node_ptr if A is smaller than B the
    }
    return true;
}

int main() {
    /* TODO:
        Write a program that reads two binary search trees from standard input, which repres
        integers (no duplicates within each tree), and checks if the first one is a subset

        The program must output "true" if the first tree is a subset of the second one, and

        The time complexity of your program must be O(n + m), where n and m are the sizes o
    */
    sax::binary_tree_node<int>* A = nullptr;
    sax::binary_tree_node<int>* B = nullptr;

    std::cin >> A >> B;

    bool result = is_subset(A, B);
    std::cout << (result ? "true" : "false") << std::endl;

    sax::binary_tree_node<int>::cleanup(A);
    sax::binary_tree_node<int>::cleanup(B);
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because each node in tree A and tree B is visited at most once while traversing the trees in sorted order using the node_ptr iterators. The two-pointer technique ensures that we never revisit nodes, so the total number of operations is proportional to the sum of the number of nodes in both trees.

Assignment 4 - Tree intersection

My code:

```
#include <iostream>
#include "utils.h"
#include "bintree.h"
#include "node_ptr.h"

int main() {
    /* TODO:
        Write a program that reads two binary search trees from standard input, which repres
        integers (no duplicates within each tree), and builds a vector containing the inters

        The program must write the resulting vector to standard output.

        The time complexity of your program must be O(n + m), where n and m are the sizes o

        Example input:
            ((2 3 4) 5 (6 7 8)) (3 5 10)
        Example output:
            [3, 5]
    */
    sax::binary_tree_node<int>* A = nullptr;
    sax::binary_tree_node<int>* B = nullptr;

    std::cin >> A >> B;

    sax::node_ptr<int> itA(A);
    sax::node_ptr<int> itB(B);

    std::vector<int> intersection;

    while(itA && itB){
        if(itA->data == itB->data){
            intersection.push_back(itA->data);
            itA.move_next();
            itB.move_next();
        } else if (itA->data < itB->data){
            itA.move_next();
        } else {
            itB.move_next();
        }
    }

    std::cout << "[";
```

```
    for (size_t i = 0; i < intersection.size(); i++){
        std::cout << intersection[i];
        if(i + 1 < intersection.size()) std::cout << ", ";
    }
    std::cout << "]" << std::endl;

    sax::binary_tree_node<int>::cleanup(A);
    sax::binary_tree_node<int>::cleanup(B);
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n + m), because each node in tree A and tree B is visited at most once while traversing the trees in sorted order using the node_ptr iterators. The two-pointer technique ensures that we never revisit nodes, so the total number of operations is proportional to the sum of the number of nodes in both trees.

Assignment 5 - Array to balanced tree

My code:

```
#include <iostream>
#include "utils.h"
#include "bintree.h"
#include "string"
#include <vector>

sax::binary_tree_node<std::string>* build_balanced_bst(const std::vector<std::string>& arr,
    if(left > right) return nullptr;

    int mid = left + (right - left) / 2;  //Mathematically equivalent to (left + right)/2, l
    sax::binary_tree_node<std::string>* node = new sax::binary_tree_node<std::string>{arr[mi

    //recursivelybuild left and right subtrees
    node->left = build_balanced_bst(arr, left, mid - 1);
    node->right = build_balanced_bst(arr, mid + 1, right);

    return node;
}

int main(){
    /* TODO:
        Write a program that reads a sorted array of strings from standard input,
        constructs a balanced binary search tree containing the strings, and
        prints the tree to standard output.
```

```
        Example input:
            [apple, banana, cherry, date, elderberry, fig, grape]
        Example output:
            ((apple banana cherry) date (elderberry fig grape))
    */
    std::vector<std::string> arr;
    std::cin >> arr;

    sax::binary_tree_node<std::string>* root = build_balanced_bst(arr, 0, arr.size() - 1);

    std::cout << root << std::endl;
    sax::binary_tree_node<std::string>::cleanup(root);
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because each element in the input array is visited exactly once to create a node. The recursive calls divide the array into halves at each step, and each element is allocated and assigned to a node exactly one time.

Assignment 6 - Non-recursive tree height

My code:

```
#include <iostream>
#include "bintree.h"
#include <stack> // for std::stack
#include <utility> //std::pair

int main() {
    /* TODO:
        Write a program that reads a binary tree from standard input, and prints its height
        Your program must not use recursion.

        Example input: ((2 3 4) 5 (6 7 8))
        Example output: 3
    */
    sax::binary_tree_node<int>* root = nullptr;
    std::cin >> root;

    if(!root){
        std::cout << 0 << std::endl; //empty tree has height 0
        return 0;
    }

    std::stack<std::pair<sax::binary_tree_node<int>*, int>> stk;
```

```
    stk.push({root, 1});// root is at depth 1
    int max_height = 0;

    while(!stk.empty()){
        auto [node, depth] = stk.top();
        stk.pop();

        if(node){
            //update max height
            if (depth > max_height) max_height = depth;

            //push children  with incrmented height
            if(node->left) stk.push({node->left, depth + 1});
            if(node->right) stk.push({node->right, depth + 1});
        }
    }
    std::cout << max_height<< std::endl;
    sax::binary_tree_node<int>::cleanup(root);
    return 0;
}
```

Time complexity: this algorithm has a time complexity of O(n), because each node in the tree is visited exactly once. We push each node onto the stack once and pop it once, performing a constant amount of work per node.