Міністерство освіти і науки України

Національний університет «Львівська політехніка»

Кафедра систем штучного інтелекту

Звіт

про виконання лабораторних та практичних робіт блоку № 6

На тему: «Динамічні структури (Черга, Стек, Списки, Дерево). Алгоритми обробки динамічних структур.»

з дисципліни: «Основи програмування»

до:

ВНС Лабораторної Роботи № 10

Алготестер Лабораторних Робіт № 5, 7-8

Практичних Робіт до блоку № 6

Виконав:

Студент групи ШІ-12

Токарик Сергій

Львів 2024

Тема роботи

Динамічні структури, види динамічних структур, їх використання, алгоритми їх обробки.

Мета роботи

1. Навчитись створювати та використовувати динамічні структури, такі як: Черга, Стек, Списки, Дерево.
2. Навчитись виконувати алгоритми обробки динамічний структур.

Теоретичні відомості

1. Перенавантаження операторі виводу

   https://acode.com.ua/urok-141-perevantazhennya-operatoriv-vvodu-i-vyvodu/#toc-0

2. Класи

   https://acode.com.ua/urok-121-klasy-ob-yekty-i-metody/#toc-1

   https://acode.com.ua/urok-183-shablony-klasiv/

3. Черга

   https://www.bestprog.net/uk/2019/09/26/c-queue-general-concepts-ways-to-implement-the-queue-implementing-a-queue-as-a-dynamic-array-ua/

4. Стек

   https://www.bestprog.net/uk/2019/09/18/c-the-concept-of-stack-operations-on-the-stack-an-example-implementation-of-the-stack-as-a-dynamic-array-ua/

5. Списки

   https://codelessons.dev/ru/spisok-list-v-s-polnyj-material/

6. Дерево

https://www.bing.com/ck/a?!&&p=d92fe6ba3879a4e47f751a99580f5f4fe2193328a2a4e32cef85f6aa2f6603bcJmltdHM9MTczMjc1MjAwMA&ptn=3&ver=2&hsh=4&fclid=1c78f629-a87f-6de6-3f44-e249a96d6cf2&psq=%d0%b4%d0%b5%d1%80%d0%b5%d0%b2%d0%b0+%d1%83+%d1%81%2b%2b&u=a1aHR0cHM6Ly9wdXJlY29kZWNwcC5jb20vdWsvYXJjaGl2ZXMvMjQ4Mw&ntb=1

Виконання роботи

Завдання №1 -VNS lab 10 - варіант 24

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _Node {
    char *pc;
    struct _Node *next;
    struct _Node *prev;
} Node;

typedef struct _DblLinkedList {
    size_t size;
    Node *head;
    Node *tail;
} DblLinkedList;

DblLinkedList* createDblLinkedList() {
    DblLinkedList *tmp = (DblLinkedList*)malloc(sizeof(DblLinkedList));
    tmp->size = 0;
    tmp->head = tmp->tail = NULL;
    return tmp;
}

void deleteDblLinkedList(DblLinkedList **list) {
    Node *tmp = (*list)->head;
    Node *next = NULL;
    while (tmp) {
        next = tmp->next;
        free(tmp->pc); // Звільняємо пам'ять, виділену для рядка
        free(tmp);
        tmp = next;
    }
    free(*list);
    *list = NULL;
}

void pushFront(DblLinkedList *list, const char *data) {
    Node *tmp = (Node*)malloc(sizeof(Node));
    tmp->pc = strdup(data); // Копіюємо рядок
    tmp->next = list->head;
    tmp->prev = NULL;
    if (list->head) {
        list->head->prev = tmp;
    }
    list->head = tmp;
    if (list->tail == NULL) {
        list->tail = tmp;
    }
    list->size++;
}

void pushBack(DblLinkedList *list, const char *value) {
    Node *tmp = (Node*)malloc(sizeof(Node));
    tmp->pc = strdup(value); // Копіюємо рядок
    tmp->next = NULL;
    tmp->prev = list->tail;
    if (list->tail) {
        list->tail->next = tmp;
    }
    list->tail = tmp;
    if (list->head == NULL) {
        list->head = tmp;
    }
    list->size++;
}

void* deleteNth(DblLinkedList *list, size_t index) {
```

```c
void* deleteNth(DblLinkedList *list, size_t index) {
    Node *tmp = list->head;
    size_t i = 0;
    while (tmp && i < index) {
        tmp = tmp->next;
        i++;
    }
    if (!tmp) return NULL;
    if (tmp->prev) tmp->prev->next = tmp->next;
    if (tmp->next) tmp->next->prev = tmp->prev;
    if (!tmp->prev) list->head = tmp->next;
    if (!tmp->next) list->tail = tmp->prev;
    char *data = tmp->pc;
    free(tmp);
    list->size--;
    return data;
}

Node* getNth(DblLinkedList *list, size_t index) {
    Node *tmp = list->head;
    size_t i = 0;
    while (tmp && i < index) {
        tmp = tmp->next;
        i++;
    }
    return tmp;
}

void insert(DblLinkedList *list, size_t index, const char *value) {
    Node *elm = getNth(list, index);
    if (!elm) return;
    Node *newNode = (Node*)malloc(sizeof(Node));
    newNode->pc = strdup(value); // Копіюємо рядок
    newNode->prev = elm;
    newNode->next = elm->next;
    if (elm->next) elm->next->prev = newNode;
    elm->next = newNode;
    if (!elm->prev) list->head = elm;
    if (!elm->next) list->tail = elm;
    list->size++;
}

void listPrint(DblLinkedList *list) {
    Node *tmp = list->head;
    while (tmp) {
        printf("%s ", tmp->pc);
        tmp = tmp->next;
    }
    printf("\n");
}

int main() {
    DblLinkedList *list = createDblLinkedList();
    pushBack(list, "one");
    pushBack(list, "two");
    pushBack(list, "three");
    listPrint(list);

    deleteNth(list, 1);
    listPrint(list);

    Node *target = getNth(list, 1);
    if (target) {
        insert(list, 1, "inserted-before");
        insert(list, 2, "inserted-after");
    }
    listPrint(list);

    deleteDblLinkedList(&list);
    return 0;
}
```

```
one two three
one three
one three inserted-before inserted-after
```

Завдання №2 Algotester lab 5 - варіант 3

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

int main() {
    int N, M, x, y;
    cin >> N >> M >> x >> y;
    x--; y--;

    vector<vector<int>> mount(N, vector<int>(M, -1));
    queue<pair<int, int>> q;
    q.push({x, y});

    int max_height = max(x, N - 1 - x) + max(y, M - 1 - y);
    mount[x][y] = max_height;

    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};

    while (!q.empty()) {
        int cx = q.front().first;
        int cy = q.front().second;
        q.pop();
        for (int i = 0; i < 4; i++) {
            int nx = cx + dx[i];
            int ny = cy + dy[i];
            if (nx >= 0 && nx < N && ny >= 0 && ny < M && mount[nx][ny] == -1) {
                mount[nx][ny] = mount[cx][cy] - 1;
                q.push({nx, ny});
            }
        }
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            cout << mount[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

```
3 4
2 2
1 2 1 0
2 3 2 1
1 2 1 0
```

Завдання №3 Algotester lab 7-8 - варіант 1

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

template <typename T>
class DoublyLinkedList {
private:
    struct Node {
        T data;
        Node* prev;
        Node* next;

        Node(const T& value) : data(value), prev(nullptr), next(nullptr) {}
    };

    Node* head;
    Node* tail;
    int size;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr), size(0) {}

    ~DoublyLinkedList() {
        clear();
    }

    void clear() {
        while (head) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
        tail = nullptr;
        size = 0;
    }

    void insert(int index, int n, const T* elements) {
        if (index < 0 || index > size) return;

        for (int i = 0; i < n; i++) {
            Node* newNode = new Node(elements[i]);

            if (index == 0) {
                newNode->next = head;
                if (head) head->prev = newNode;
                head = newNode;
                if (!tail) tail = head;
            } else {
                Node* current = head;
                for (int j = 0; j < index - 1; j++) current = current->next;

                newNode->next = current->next;
                if (current->next) current->next->prev = newNode;
                current->next = newNode;
                newNode->prev = current;

```

```
 5  class DoublyLinkedList {
36      void insert(int index, int n, const T* elements) {
39          for (int i = 0; i < n; i++) {
47              } else {
                    newNode->prev = current;

56                  if (!newNode->next) tail = newNode;
57              }
58              index++;
59              size++;
60          }
61      }
62
63      void erase(int index, int n) {
64          if (index < 0 || index >= size || n <= 0) return;
65
66          for (int i = 0; i < n; i++) {
67              if (index >= size) break;
68
69              Node* current = head;
70              for (int j = 0; j < index; j++) current = current->next;
71
72              if (current->prev) current->prev->next = current->next;
73              if (current->next) current->next->prev = current->prev;
74
75              if (current == head) head = current->next;
76              if (current == tail) tail = current->prev;
77
78              delete current;
79              size--;
80          }
81      }
82
83      int getSize() const {
84          return size;
85      }
86
87      T get(int index) const {
88          if (index < 0 || index >= size) throw out_of_range("Index out of range");
89
90          Node* current = head;
91          for (int i = 0; i < index; i++) current = current->next;
92          return current->data;
93      }
94
95      void set(int index, const T& value) {
96          if (index < 0 || index >= size) throw out_of_range("Index out of range");
97
98          Node* current = head;
99          for (int i = 0; i < index; i++) current = current->next;
100         current->data = value;
101     }
102
103     friend ostream& operator<<(ostream& os, const DoublyLinkedList& list) {
104         Node* current = list.head;
```

```
        Node* current = list.head;
        while (current) {
            os << current->data << " ";
            current = current->next;
        }
        return os;
    }
};

int main() {
    DoublyLinkedList<int> list;
    int q;
    cin >> q;

    while (q--) {
        string command;
        cin >> command;

        if (command == "insert") {
            int index, n;
            cin >> index >> n;
            int* elements = new int[n];
            for (int i = 0; i < n; i++) cin >> elements[i];
            list.insert(index, n, elements);
            delete[] elements;
        } else if (command == "erase") {
            int index, n;
            cin >> index >> n;
            list.erase(index, n);
        } else if (command == "size") {
            cout << list.getSize() << endl;
        } else if (command == "get") {
            int index;
            cin >> index;
            try {
                cout << list.get(index) << endl;
            } catch (const out_of_range& e) {
                cout << e.what() << endl;
            }
        } else if (command == "set") {
            int index, value;
            cin >> index >> value;
            try {
                list.set(index, value);
            } catch (const out_of_range& e) {
                cout << e.what() << endl;
            }
        } else if (command == "print") {
            cout << list << endl;
        }
    }

    return 0;
}
```

```
5

insert
0 3
1 2 3
erase
0 2

set
0 10

size
1

print
10
```

Завдання №4 Class Practice Work

```cpp
#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* next;
    Node(int value) : data(value), next(nullptr) {}
};

Node* reverseList(Node* head) {
    Node* previous = nullptr;
    Node* current = head;
    Node* following = nullptr;
    while (current != nullptr) {
        following = current->next;
        current->next = previous;
        previous = current;
        current = following;
    }
    return previous;
}

void printList(Node* head) {
    while (head != nullptr) {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

void append(Node*& head, int value) {
    Node* newNode = new Node(value);
    if (!head) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

bool listsEqual(Node* h1, Node* h2) {
    while (h1 && h2) {
        if (h1->data != h2->data) {
            return false;
        }
        h1 = h1->next;
        h2 = h2->next;
    }
    return h1 == nullptr && h2 == nullptr;
}

```

```cpp
Node* addLists(Node* n1, Node* n2) {
    Node* result = nullptr;
    Node** tail = &result;
    int carry = 0;

    while (n1 || n2 || carry) {
        int sum = carry;
        if (n1) {
            sum += n1->data;
            n1 = n1->next;
        }
        if (n2) {
            sum += n2->data;
            n2 = n2->next;
        }
        carry = sum / 10;
        *tail = new Node(sum % 10);
        tail = &((*tail)->next);
    }
    return result;
}

struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

TreeNode* mirrorTree(TreeNode* root) {
    if (!root) {
        return nullptr;
    }
    TreeNode* mirrored = new TreeNode(root->data);
    mirrored->left = mirrorTree(root->right);
    mirrored->right = mirrorTree(root->left);
    return mirrored;
}

void printTree(TreeNode* root) {
    if (root) {
        printTree(root->left);
        cout << root->data << " ";
        printTree(root->right);
    }
}

TreeNode* insertIntoTree(TreeNode* root, int value) {
    if (!root) {
        return new TreeNode(value);
    }
    if (value < root->data) {
        root->left = insertIntoTree(root->left, value);
```

```
        root->left = insertIntoTree(root->left, value);
    } else {
        root->right = insertIntoTree(root->right, value);
    }
    return root;
}

int calculateSubtreeSum(TreeNode* root) {
    if (!root) return 0;
    if (!root->left && !root->right) return root->data;
    int leftSum = calculateSubtreeSum(root->left);
    int rightSum = calculateSubtreeSum(root->right);
    root->data = leftSum + rightSum;
    return root->data;
}

int main() {
    Node* list1 = nullptr;
    Node* list2 = nullptr;

    append(list1, 1);
    append(list1, 2);
    append(list1, 3);
    append(list2, 1);
    append(list2, 2);
    append(list2, 3);

    cout << "List 1: ";
    printList(list1);
    cout << "List 2: ";
    printList(list2);

    cout << (listsEqual(list1, list2) ? "Lists are identical" : "Lists are different") << endl;

    list1 = reverseList(list1);
    cout << "Reversed List 1: ";
    printList(list1);

    Node* sumList = addLists(list1, list2);
    cout << "Sum List: ";
    printList(sumList);

    TreeNode* root = nullptr;
    root = insertIntoTree(root, 4);
    root = insertIntoTree(root, 2);
    root = insertIntoTree(root, 6);
    root = insertIntoTree(root, 1);
    root = insertIntoTree(root, 3);
    root = insertIntoTree(root, 5);
    root = insertIntoTree(root, 7);

    cout << "Original Tree: ";
    printTree(root);
    cout << endl;

    calculateSubtreeSum(root);
    cout << "Tree after subtree sums: ";
    printTree(root);
    cout << endl;

    return 0;
}
```

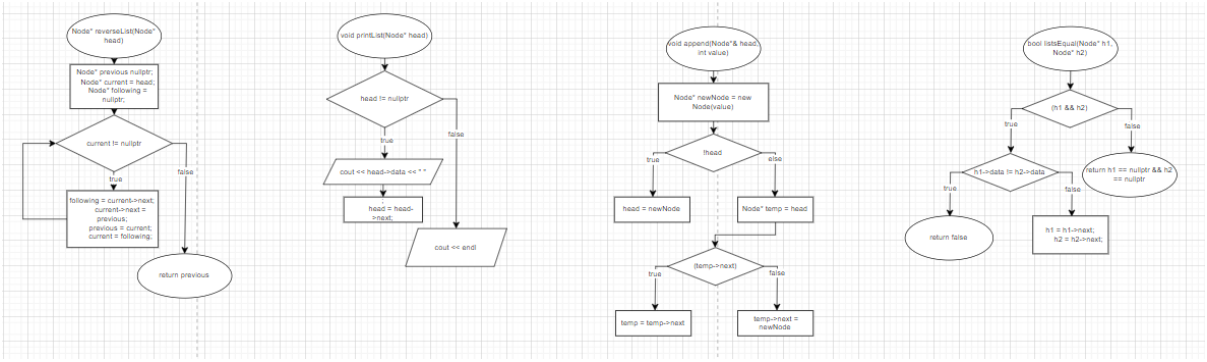
```
List 1: 1 2 3
List 2: 1 2 3
Lists are identical
Reversed List 1: 3 2 1
Sum List: 4 4 4
Original Tree: 1 2 3 4 5 6 7
Tree after subtree sums: 1 4 3 16 5 12 7
```

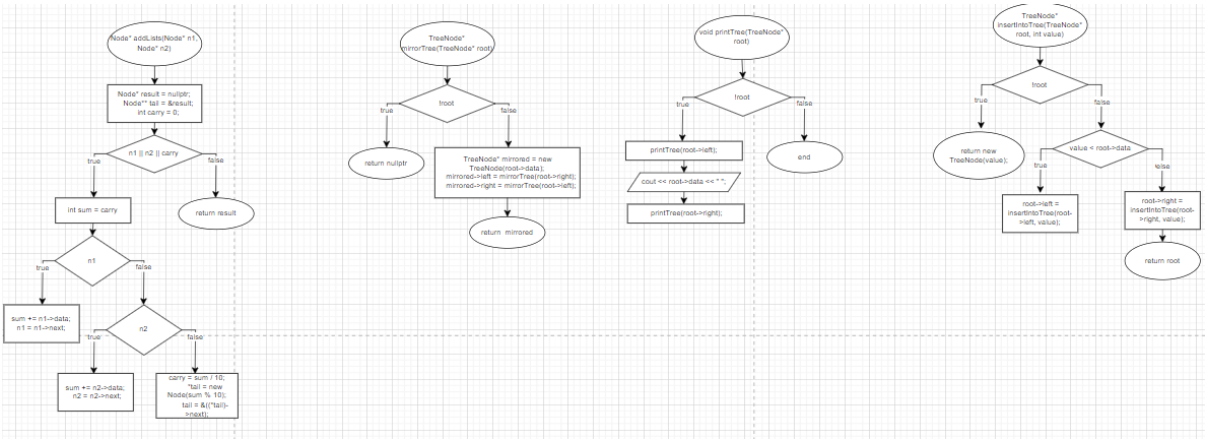
Завдання №5 Self Practice Work

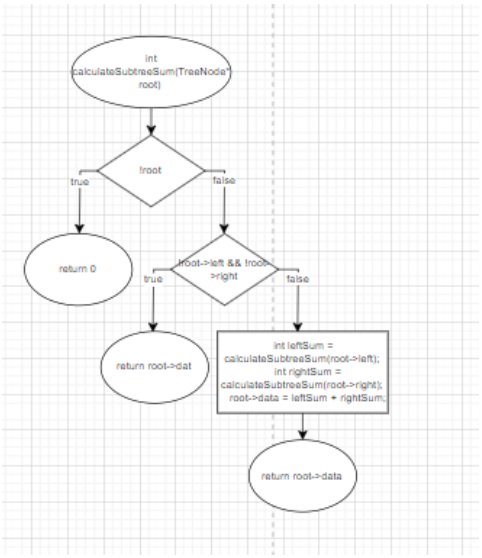
```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int N, K;
    cin >> N >> K;

    vector<int> A(N);
    for (int i = 0; i < N; i++) {
        cin >> A[i];
    }

    int maxLength = 0;
    int currentLength = 0;

    for (int i = 0; i < N; i++) {
        if (A[i] >= K) {
            currentLength++;
            maxLength = max(maxLength, currentLength);
        } else {
            currentLength = 0;
        }
    }

    cout << maxLength << endl;
    return 0;
}

```

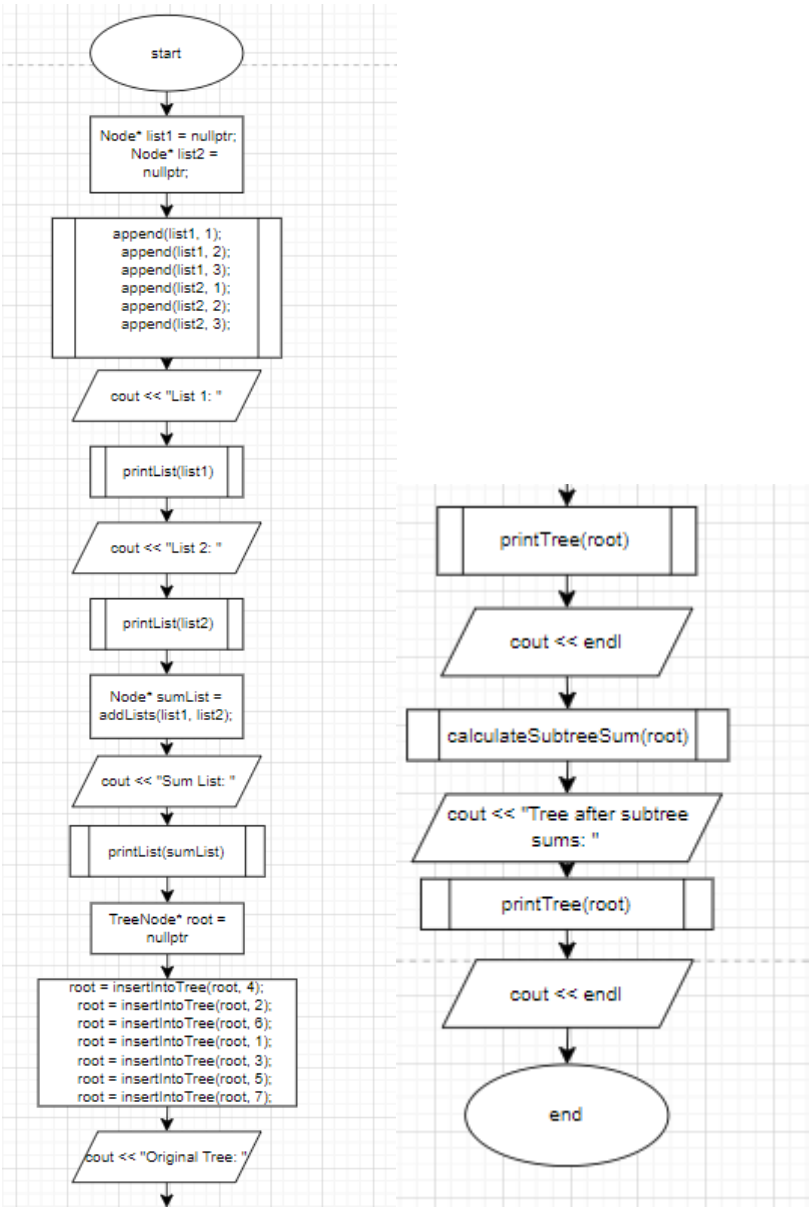
```
8
5
3 4 4 5  1 6 7 8
3
```

Діаграма:


Node* reverseList(Node* head)
Node* previous nullptr;
Node* current = head;
Node* following = nullptr;
current != nullptr
true
false
following = current->next;
current->next = previous;
previous = current;
current = following;
return previous
void printList(Node* head)
head != nullptr
true
false
cout << head->data << " "
head = head->next;
cout << endl
void append(Node*& head, int value)
Node* newNode = new Node(value)
head
else
head = newNode
Node* temp = head
(temp->next)
temp = temp->next
temp->next = newNode
bool listsEqual(Node* h1, Node* h2)
(h1 && h2)
h1->data != h2->data
return h1 == nullptr && h2 == nullptr
return false
h1 = h1->next;
h2 = h2->next;


Node* addLists(Node* n1, Node* n2)
Node* result = nullptr;
Node** tail = &result;
int carry = 0;
n1 || n2 || carry
int sum = carry
return result
n1
sum += n1->data;
n1 = n1->next;
n2
sum += n2->data;
n2 = n2->next;
carry = sum / 10;
*tail = new Node(sum % 10);
tail = &((*tail)->next);
TreeNode* mirrorTree(TreeNode* root)
!root
return nullptr
TreeNode* mirrored = new TreeNode(root->data);
mirrored->left = mirrorTree(root->right);
mirrored->right = mirrorTree(root->left);
return mirrored
void printTree(TreeNode* root)
!root
printTree(root->left);
cout << root->data << " ";
printTree(root->right);
end
TreeNode* insertIntoTree(TreeNode* root, int value)
!root
return new TreeNode(value);
value < root->data
root->left = insertIntoTree(root->left, value);
root->right = insertIntoTree(root->right, value);
return root


int calculateSubtreeSum(TreeNode* root)
!root
return 0
root->left && !root->right
return root->dat
int leftSum = calculateSubtreeSum(root->left);
int rightSum = calculateSubtreeSum(root->right);
root->data = leftSum + rightSum;
return root->data


start
Node* list1 = nullptr;
Node* list2 =
nullptr;
append(list1, 1);
append(list1, 2);
append(list1, 3);
append(list2, 1);
append(list2, 2);
append(list2, 3);
cout << "List 1: "
printList(list1)
cout << "List 2: "
printList(list2)
Node* sumList =
addLists(list1, list2);
cout << "Sum List: "
printList(sumList)
TreeNode* root =
nullptr
root = insertIntoTree(root, 4);
root = insertIntoTree(root, 2);
root = insertIntoTree(root, 6);
root = insertIntoTree(root, 1);
root = insertIntoTree(root, 3);
root = insertIntoTree(root, 5);
root = insertIntoTree(root, 7);
cout << "Original Tree: "
printTree(root)
cout << endl
calculateSubtreeSum(root)
cout << "Tree after subtree
sums: "
printTree(root)
cout << endl
end

Зустрічі з командою

З командою зустрічалися двічі, на зустрічах обговорювали питання по епіку.

Висновок:

Завдяки цій роботі я зрозумів принципи динамічних структур даних, їх ключові операції та відмінності між статичним і динамічним виділенням пам’яті. Практичні вправи зі стеком, чергою, зв’язним списком і деревом допомогли мені опанувати алгоритми обробки даних у пам'яті та ефективне управління ресурсами. Цей досвід дозволив мені створювати адаптивні рішення для роботи з великими та змінними обсягами даних, підвищуючи продуктивність програм.