1. Структури от данни – дефиниране на понятието.

Под СД се разбира организирана информацияа, като може да бъде описана, създадена и обработена с помощта на програма. За да се определи една СД е необходимо да се направи:

- логическо описание на структурата – описва я на базата на декомпозицията ѝ на по-прости структури, а също на декомпозицията на операциите на структурата на по-прости операции.

- физическо описание – дава метода за представяне на структурата в паметта на компютъра.

2. Списък. Логическо описание. Списък с една и две връзки. Характеристики на реализациите с една и две връзки. Сложност на операциите по добавяне, премахване и намиране на елемент. Дефиниране на клас за списък, използващ една от реализациите.

Списъкът е крайна редица от хомогенни елементи. Операциите за включване и изключване са допустими на произволно място в редицата. Възможен е достъп до всеки елемент в редицата (пряк или непряк в зависимост от реализацията на списъка).

Има два основни начина физически да се представи списъкът в компютъра – свързано с една връзка, свързано с две връзки или последователно (не се използва, крайно неефективно).

За свързаното представяне с една връзка клетките се съхраняват на различни места в паметта и не са последователно разположени. Връзката между отделни елементи се осъществява чрез указател към следващия (за край се използва nullptr). За поддържане на списък е достатъчно указател към началото му

За свързаното представяне с две връзки клетките се съхраняват на различни места в ОП и връзка между отделните елементи се осъществява чрез указатели към предния и следващия (за край се използва nullptr). За поддържане на списъка е достатъчно указател към началото му

Сложност:

- с една връзка – добавяне в начало O(1), добавяне на позиция O(n), премахване в началото O(1), премахване на позиция O(n), намиране O(n)

- с две връзки - добавяне в начало/край O(1), добавяне на позиция O(n), премахване в начало/край O(1), премахване на позиция O(n), намиране O(n)

```
template <typename T>

class LinkedList {

private:
```

```
    struct Node {
        T data;
        Node* next;
        Node(const T& newData) : data(newData), next(nullptr) {}
    };

    Node* head;

public:
    LinkedList() : head(nullptr) {}

    ~LinkedList() {
        while (head != nullptr) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }

    void push_front(const T& newData) {
        Node* newNode = new Node(newData);
        newNode->next = head;
        head = newNode;
    }

    void push_back(const T& newData) {
        Node* newNode = new Node(newData);
        if (head == nullptr) {
            head = newNode;
```

```
            return;
        }
        Node* current = head;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }

    void insert(const T& newData, int position) {
        if (position < 0)
            return;
        if (position == 0) {
            push_front(newData);
            return;
        }

        Node* newNode = new Node(newData);
        Node* current = head;
        for (int i = 0; i < position - 1 && current != nullptr; ++i) {
            current = current->next;
        }
        if (current == nullptr) return;
        newNode->next = current->next;
        current->next = newNode;
    }
};
```

3. Стек. Логическо описание. Характеристики на статичната, динамичната и свързаната реализация. Сложност на операциите по добавяне и премахване на елемент. Дефиниране на клас за стек, използващ една от реализациите.

Линейна динамична СД. Стекът е крайна редица от хомогенни елементи. Операции за включване и изключване на елементи само от върха на стека – LIFO(Last in first out). Има последователно и свързано представяне на стека. При последователното предварително се пази блок в паметта и стекът има ограничен капацитет. При свързаното представяне парчетата памет са разпръснати по ОП и връзката между отделните елементи е чрез указател към следващия. Стекът може да расте „неограничено“ (докато не свърши ОП). Достатъчен е указател към началото му. Краят се бележи с nullptr.

Сложност на добавяне и премахване на елемент:
- статична имплементация – добавянето може да отнеме O(n) ако има нужда от преоразмеряване на масива, затова можем да кажем, че има амортизирано O(1). Махането на елемент е O(1).

- динамична имплементация – добавяне и премахване на елемент е O(1)

```
template <typename T>

class Stack {

private:

    struct Node {

        T data;

        Node* next;

        Node(const T& newData) : data(newData), next(nullptr) {}

    };

    Node* topNode;


public:

    Stack() : topNode(nullptr) {}


    ~Stack() {

        while (!isEmpty()) {
```

```
        pop();
    }
}

void push(const T& newData) {
    Node* newNode = new Node(newData);
    if (isEmpty()) {
        topNode = newNode;
    } else {
        newNode->next = topNode;
        topNode = newNode;
    }
}

void pop() {
    if (!isEmpty()) {
        Node* temp = topNode;
        topNode = topNode->next;
        delete temp;
    } else {
        std::cout << "Error: Stack underflow\n";
    }
}

T peek() const {
    if (!isEmpty()) {
        return topNode->data;
    } else {
        std::cerr << "Error: Stack is empty\n";
```

```
            // Returning a default value. You might want to handle this differently.
            return T();
        }
    }

    bool isEmpty() const {
        return topNode == nullptr;
    }
};
```

4. Опашка. Логическо описание. Характеристики на статичната, динамичната и свързаната реализация. Сложност на операциите по добавяне и премахване на елемент. Дефиниране на клас за опашка, използващ една от реализациите.

Опашката е крайна редица от хомогенни елементи. Операция за включване е допустима в края ѝ, а за изключване от началото ѝ, т.е. е FIFO(First in first out). Може да се представи физически последователно или свързано. При последователно представяне първоначално се пази блок памет вътре в който опашката расте и се съкращава. Обикновено се счита, че блокът памет е цикличен, тоест, когато краят на опашката достигне края на блока, но има свободна памет в неговото начало, там може да се включи елемент. При свързаното представяне парчетата са разпръснати в ОП и достъп до отделните елементи се извършва чрез указател към следващ елемент. Достатъчни са ни 2 указателя за началото и края на опашката. Краят се бележи с nullptr.

Сложност:

- статична реализация – добавяне е O(1), премахване O(n) (може да изисква преместване на всички елементи на по-предни позиции)
- статична реализация(циклична) – добавяне е O(1), премахване O(1)
- динамична реализация – добавяне O(1), премахване O(1)

```
const int MAX_SIZE = 20; class Queue {
private:
    int arr[MAX_SIZE];
    int front, rear;
```

```
public:
    CircularQueue() {
        front = rear = -1;
    }

    void push(int value) {
        if (isFull()) {
            cout << "Queue is full" << endl;
            return;
        }
        if (isEmpty()) {
            front = 0;
            rear = 0;
        } else {
            rear = (rear + 1) % MAX_SIZE;
        }
        arr[rear] = value;
        cout << "Pushed element: " << value << endl;
    }

    void pop() {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return;
        }
        cout << "Popped element: " << arr[front] << endl;
        if (front == rear) {
            front = -1;
            rear = -1;
```

```
        } else {
            front = (front + 1) % MAX_SIZE;
        }
    }

    bool isEmpty() {
        return front == -1;
    }

    bool isFull() {
        return (rear + 1) % MAX_SIZE == front;
    }
};
```

5. Дървовидни структури от данни – кореново дърво и двоично кореново дърво. Логическо описание. Начини за представяне в паметта. Дефиниране на клас, реализиращ кореново дърво или двоично кореново дърво.

Двоично дърво от тип T е рекурсивна СД, която е или празна или е образувана от:

- данни тип T – корен на дърво;
- двоично дърво тип T, наречено {ляво, дясно} поддървно на двоично дърво

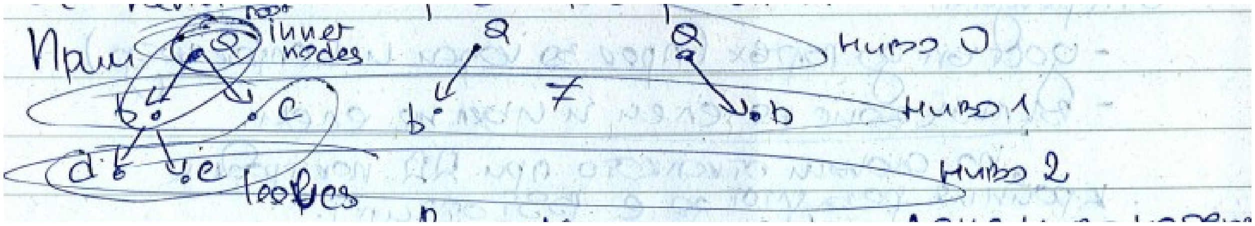
Множеството от върховете на ДД се определя рекурсивно:

- празното ДД няма върхове
- върховете на непразно ДД са неговият корен и върховете на двете му поддървета

Листата на ДД са върховете с две празни поддървета. Останалите се наричат вътрешни.

Височината на дърво: максималното ниво на дърво

Пример:



Операции:

- достъп до връх (пряк до корен или непряк до другите)
- включване и изключване на връх (от произволно място – резултатът е отново двоично дърво от същия тип)
- обхождане (ЛКД (смесено?), КЛД (възходящо), ЛДК (низходящо))

Три начина за представяне в паметта:

- свързано (реализира се чрез указател към кутия с 3 полета – inf, left и right указатели за поддържане)
- верижно – 3 масива a[N], b[N], c[N].

N = # върхове от 0 до N-1, елементите на a съдържат стойност на i-ти връх, елементите на b съдържат индекс на ляв налседник на i-ти връх (-1 ако няма) и елементите на c съдържат индекс на десен наследник на i-ти връх (-1 ако няма).

- чрез списък на бащите – 1 масив p[N]. N = #върхове в дървото от 0 до N-1. Елементът p[i] е единствения баща на върха i (-1 ако този връх е корен).

```
template<typename T>
class BinaryTree {
private:
    class Node {
    public:
        T value;
        Node* leftChild;
        Node* rightChild;
        Node(T val) {
            value = val;
            leftChild = nullptr;
            rightChild = nullptr;
```

```
        }
    };

    Node* root;
    Node* insertRecursive(Node* currentNode, T val) {
        if (currentNode == nullptr) {
            currentNode = new Node(val);
        } else {
            if (val <= currentNode->value) {
                currentNode->leftChild = insertRecursive(currentNode->leftChild, val);
            } else {
                currentNode->rightChild = insertRecursive(currentNode->rightChild, val);
            }
        }
        return currentNode;
    }
    void inorderTraversalRecursive(Node* currentNode) {
        if (currentNode == nullptr) return;
        inorderTraversalRecursive(currentNode->leftChild);
        cout << currentNode->value << " ";
        inorderTraversalRecursive(currentNode->rightChild);
    }

public:
    BinaryTree() {
        root = nullptr;
    }
    void insert(T val) {
        root = insertRecursive(root, val);
```

```
    }
    void inorderTraversal() {
        inorderTraversalRecursive(root);
    }
};
```

6. Двоично кореново дърво за търсене. Логическо описание. Начини за представяне в паметта. Сложност на операциите по добавяне, премахване и търсене на елемент. Дефиниране на клас реализиращ двоично кореново дърво за търсене.

Предполагаме, че има линейна наредба върху елементите. BST от тип T се дефинира рекурсивно:

- празното двоично дърво от тип T е наредено
- непразно двоично дърво от тип T е наредено ⇔ всички върхове на ЛПД са по-малки от корена и всички върхове в ДПД са по-големи от корена и ЛПД и ДПД са BST от тип T

Операции:

- достъп до връх (пряк за корен и непряк за др.)
- включване и изключване на елемент(по-сложни от при ДД, крайният резултат пак е BST от тип T)
- обхождане на дърво (като ЛКД ще даде елементите в сортиран вид, КЛД, ЛДК)

Представянето е свързано с кутийки в паметта с инф поле и по 2 указателя за двете поддървета.

```
template<typename T>
class BST {
private:
    struct Node {
        T data;
        Node* left;
        Node* right;
    };

    Node* root;
```

```
Node* createNode(T value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

Node* insertRecursive(Node* current, T value) {
    if (current == nullptr) {
        return createNode(value);
    }
    if (value < current->data) {
        current->left = insertRecursive(current->left, value);
    } else if (value > current->data) {
        current->right = insertRecursive(current->right, value);
    }
    return current;
}

// Recursive function to search for a value
bool searchRecursive(Node* current, T value) {
    if (current == nullptr) {
        return false;
    }
    if (current->data == value) {
        return true;
    }
```

```
        if (value < current->data) {
            return searchRecursive(current->left, value);
        } else {
            return searchRecursive(current->right, value);
        }
    }

public:
    BST() {
        root = nullptr;
    }
    void insert(T value) {
        root = insertRecursive(root, value);
    }
    bool search(T value) {
        return searchRecursive(root, value);
    }
};
```