МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации

**

**ЛАБОРАТОРНАЯ РАБОТА №1**

**по дисциплине: «Программирование»**

**на тему: «Абстрактные структуры данных»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБ-420», «АВТФ»,  *Силютин Данил Иванович*  «\_30\_» \_мая\_ 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Проверил:  *Ассистент*  *Исаев Глеб Андреевич*  «\_\_\_» \_\_\_\_\_\_ 2025г  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) |

Новосибирск 2025

# ВВЕДЕНИЕ

Цели и задачи работы: изучение алгоритмов формирования и работы с абстрактными структурами данных.

Ход выполнения

## Задание 1

1. Реализовать структуры данных с базовым набором операций:

a. Массив. Операции: создание, добавление элемента (по индексу

и в конец массива), получение элемента по индексу, удаление

элемента по индексу, замена элемента по индексу, длина

массива, чтение.

b. Односвязный список. Операции: добавление и удаление элемента

(4 способа (до после голова хвост)), чтение (разное, несколько

способов), удаление элемента по значению, поиск элемента по

значению.

c. Двусвязный список. Операции: добавление и удаление элемента

(4 способа (до после голова хвост)), чтение (разное, несколько

способов), удаление элемента по значению, поиск элемента по

значению.

d. Стек. Операции: добавление и удаление элемента (push и pop),

чтение.

e. Очередь. Операции: добавление и удаление элемента (push и

pop), чтение.

f. Деревья (вариант 11 Красно-черное дерево, вариант 2 -

Сбалансированное двоичное дерево (АВЛ-дерево), вариант 3

Complete Binary Tree, вариант 4 Full Binary Tree, вариант 5

– бинарное дерево поиска. Операции: добавление элемента,

поиск элемента, удаление элемента с сохранением структуры

дерева, чтение. Если Full Binary Tree или Complete Binary

Tree, то вместо удаления элемента добавить операцию проверки

на Full или Complete соответственно.

Код программы на языке GO

Array.go

package array

type Array struct {

values []int

}

func NewArray() \*Array {

return &Array{values: []int{}}

}

func (a \*Array) Append(value int) {

a.values = append(a.values, value)

}

func (a \*Array) Insert(index int, value int) {

if index < 0 || index > len(a.values) {

return

}

a.values = append(a.values[:index], append([]int{value}, a.values[index:]...)...)

}

func (a \*Array) Get(index int) (int, bool) {

if index < 0 || index >= len(a.values) {

return 0, false

}

return a.values[index], true

}

func (a \*Array) Remove(index int) {

if index < 0 || index >= len(a.values) {

return

}

a.values = append(a.values[:index], a.values[index+1:]...)

}

func (a \*Array) Set(index int, value int) {

if index < 0 || index >= len(a.values) {

return

}

a.values[index] = value

}

func (a \*Array) Length() int {

return len(a.values)

}

func (a \*Array) Read() []int {

out := make([]int, len(a.values))

copy(out, a.values)

return out

}

Avltree.go

package avltree

import "fmt"

type Node struct {

Key int

Height uint8

Left \*Node

Right \*Node

}

func NewNode(key int) \*Node {

return &Node{

Key: key,

Height: 1,

}

}

*// HeightVal возвращает высоту узла (0, если узел nil)*

func (n \*Node) HeightVal() uint8 {

if n == nil {

return 0

}

return n.Height

}

*// BalanceFactor возвращает баланс-фактор узла*

func (n \*Node) BalanceFactor() int {

if n == nil {

return 0

}

return int(n.Left.HeightVal()) - int(n.Right.HeightVal())

}

*// FixHeight пересчитывает высоту узла*

func (n \*Node) FixHeight() {

if n == nil {

return

}

hl := n.Left.HeightVal()

hr := n.Right.HeightVal()

if hl > hr {

n.Height = hl + 1

} else {

n.Height = hr + 1

}

}

*// RotateRight выполняет правый поворот*

func (p \*Node) RotateRight() \*Node {

if p == nil || p.Left == nil {

return p

}

q := p.Left

p.Left = q.Right

q.Right = p

p.FixHeight()

q.FixHeight()

return q

}

*// RotateLeft выполняет левый поворот*

func (q \*Node) RotateLeft() \*Node {

if q == nil || q.Right == nil {

return q

}

p := q.Right

q.Right = p.Left

p.Left = q

q.FixHeight()

p.FixHeight()

return p

}

*// Balance балансирует узел*

func (p \*Node) Balance() \*Node {

if p == nil {

return nil

}

p.FixHeight()

*// Правый дисбаланс*

if p.BalanceFactor() == 2 {

if p.Right != nil && p.Right.BalanceFactor() < 0 {

p.Right = p.Right.RotateRight()

}

return p.RotateLeft()

}

*// Левый дисбаланс*

if p.BalanceFactor() == -2 {

if p.Left != nil && p.Left.BalanceFactor() > 0 {

p.Left = p.Left.RotateLeft()

}

return p.RotateRight()

}

return p

}

*// Insert добавляет ключ в дерево*

func (p \*Node) Insert(key int) \*Node {

if p == nil {

return NewNode(key)

}

if key < p.Key {

p.Left = p.Left.Insert(key)

} else if key > p.Key {

p.Right = p.Right.Insert(key)

} else {

return p

}

return p.Balance()

}

*// Search ищет ключ в дереве*

func (n \*Node) Search(key int) bool {

if n == nil {

return false

}

if key == n.Key {

return true

} else if key < n.Key {

return n.Left.Search(key)

} else {

return n.Right.Search(key)

}

}

*// findMin ищет узел с минимальным ключом*

func (n \*Node) findMin() \*Node {

if n.Left == nil {

return n

}

return n.Left.findMin()

}

*// removeMin удаляет узел с минимальным ключом*

func (n \*Node) removeMin() \*Node {

if n.Left == nil {

return n.Right

}

n.Left = n.Left.removeMin()

n.FixHeight()

return n.Balance()

}

*// Delete удаляет ключ из дерева*

func (n \*Node) Delete(key int) \*Node {

if n == nil {

return nil

}

if key < n.Key {

n.Left = n.Left.Delete(key)

} else if key > n.Key {

n.Right = n.Right.Delete(key)

} else {

left := n.Left

right := n.Right

if right == nil {

return left

}

min := right.findMin()

n.Key = min.Key

n.Right = n.Right.removeMin()

}

n.FixHeight()

return n.Balance()

}

*// InOrder печатает дерево в порядке возрастания*

func (n \*Node) InOrder() {

if n == nil {

return

}

n.Left.InOrder()

fmt.Print(n.Key, " ")

n.Right.InOrder()

}

Doublylinkedlist.go

*// c. Двусвязный список. Операции: добавление и удаление элемента*

*// (4 способа (до после голова хвост)), чтение (разное, несколько*

*// способов), удаление элемента по значению, поиск элемента по*

*// значению.*

package doubleLinkedlist

import "fmt"

type Node struct {

Next \*Node

Prev \*Node

Data int

}

type DoublyLinkedList struct {

Head \*Node

Tail \*Node

}

func (list \*DoublyLinkedList) AppendAtBack(Data int) {

newNode := &Node{Data: Data}

if list.Tail == nil {

list.Head = newNode

list.Tail = newNode

return

}

list.Tail.Next = newNode

newNode.Prev = list.Tail

list.Tail = newNode

}

func (list \*DoublyLinkedList) AppendAtStart(Data int) {

newNode := &Node{Data: Data}

if list.Head == nil {

list.Head = newNode

list.Tail = newNode

return

}

newNode.Next = list.Head

list.Head.Prev = newNode

list.Head = newNode

}

func (list \*DoublyLinkedList) AppendAfterValue(value, Data int) {

current := list.Head

for current != nil && current.Data != value {

current = current.Next

}

if current == nil {

fmt.Printf("элемент %d не найден\n", value)

return

}

newNode := &Node{Data: Data}

newNode.Next = current.Next

newNode.Prev = current

if current.Next != nil {

current.Next.Prev = newNode

} else {

list.Tail = newNode

}

current.Next = newNode

}

func (list \*DoublyLinkedList) AppendBeforeValue(value, Data int) {

current := list.Head

for current != nil && current.Data != value {

current = current.Next

}

if current == nil {

fmt.Printf("элемент %d не найден\n", value)

return

}

newNode := &Node{Data: Data}

newNode.Next = current

newNode.Prev = current.Prev

if current.Prev != nil {

current.Prev.Next = newNode

} else {

list.Head = newNode

}

current.Prev = newNode

}

func (list \*DoublyLinkedList) DeleteHead() {

if list.Head == nil {

return

}

list.Head = list.Head.Next

if list.Head != nil {

list.Head.Prev = nil

} else {

list.Tail = nil

}

}

func (list \*DoublyLinkedList) DeleteTail() {

if list.Tail == nil {

return

}

list.Tail = list.Tail.Prev

if list.Tail != nil {

list.Tail.Next = nil

} else {

list.Head = nil

}

}

func (list \*DoublyLinkedList) DeleteForValue(Data int) {

current := list.Head

for current != nil && current.Data != Data {

current = current.Next

}

if current == nil {

return

}

if current.Prev != nil {

current.Prev.Next = current.Next

} else {

list.Head = current.Next

}

if current.Next != nil {

current.Next.Prev = current.Prev

} else {

list.Tail = current.Prev

}

}

func (list \*DoublyLinkedList) FindElementForValue(Data int) \*Node {

current := list.Head

for current != nil {

if current.Data == Data {

return current

}

current = current.Next

}

return nil

}

func (list \*DoublyLinkedList) ReadForward() {

current := list.Head

for current != nil {

fmt.Println(current.Data)

current = current.Next

}

}

func (list \*DoublyLinkedList) ReadBackward() {

current := list.Tail

for current != nil {

fmt.Println(current.Data)

current = current.Prev

}

}

func (list \*DoublyLinkedList) ToSlice() []int {

collector := []int{}

current := list.Head

for current != nil {

collector = append(collector, current.Data)

current = current.Next

}

return collector

}

Linkedlist.go

*// b. Односвязный список. Операции: добавление и удаление элемента*

*// (4 способа (до после голова хвост)), чтение (разное, несколько*

*// способов), удаление элемента по значению, поиск элемента по*

*// значению.*

package linkedlist

import "fmt"

type Node struct {

Next \*Node

Data int

}

type LinkedList struct {

Head \*Node

}

func (list \*LinkedList) AppendAtBack(Data int) {

newNode := &Node{Data: Data}

if list.Head == nil {

list.Head = newNode

return

}

current := list.Head

for current.Next != nil {

current = current.Next

}

current.Next = newNode

}

func (list \*LinkedList) AppendAtStart(Data int) {

newNode := &Node{Data: Data, Next: list.Head}

list.Head = newNode

}

func (list \*LinkedList) AppendAfterValue(value, Data int) {

if list.Head == nil {

fmt.Println("список пуст")

return

}

current := list.Head

for current != nil && current.Data != value {

current = current.Next

}

if current == nil {

fmt.Printf("элемент %d не найден\n", value)

return

}

newNode := &Node{Data: Data, Next: current.Next}

current.Next = newNode

}

func (list \*LinkedList) AppendBeforeValue(value, Data int) {

if list.Head == nil {

fmt.Println("список пуст")

return

}

if list.Head.Data == value {

list.AppendAtStart(Data)

return

}

prev := list.Head

current := list.Head.Next

for current != nil && current.Data != value {

prev = current

current = current.Next

}

if current == nil {

fmt.Printf("элемент %d не найден\n", value)

return

}

newNode := &Node{Data: Data, Next: current}

prev.Next = newNode

}

func (list \*LinkedList) DeleteTail() {

if list.Head == nil {

fmt.Println("нечего удалять")

return

}

if list.Head.Next == nil {

list.Head = nil

return

}

current := list.Head

for current.Next.Next != nil {

current = current.Next

}

current.Next = nil

}

func (list \*LinkedList) DeleteHead() {

if list.Head == nil {

fmt.Println("нечего удалять")

return

}

list.Head = list.Head.Next

}

func (list \*LinkedList) DeleteAfterValue(Data int) {

if list.Head == nil {

fmt.Println("нечего удалять")

return

}

current := list.Head

for current.Data != Data {

if current.Next == nil {

return

}

current = current.Next

}

if current.Next == nil {

fmt.Println("нечего удалять")

return

}

current.Next = current.Next.Next

}

func (list \*LinkedList) DeleteBeforeValue(Data int) {

if list.Head == nil || list.Head.Next == nil {

return

}

var previous \*Node

current := list.Head

for current.Next != nil {

if current.Next.Data == Data {

if previous == nil {

list.Head = current.Next

} else {

previous.Next = current.Next

}

return

}

previous = current

current = current.Next

}

}

func (list \*LinkedList) DeleteForValue(Data int) {

if list.Head == nil {

return

}

if list.Head.Data == Data {

list.Head = list.Head.Next

return

}

current := list.Head

for current.Next != nil {

if current.Next.Data == Data {

current.Next = current.Next.Next

return

}

current = current.Next

}

}

func (list \*LinkedList) FindElementForValue(Data int) \*Node {

current := list.Head

for current != nil {

if current.Data == Data {

return current

}

current = current.Next

}

return nil

}

func (list \*LinkedList) ReadLinkedListCycle() {

current := list.Head

for current != nil {

fmt.Println(current.Data)

current = current.Next

}

}

func (list \*LinkedList) ReadLinkedListForSlice() []int {

collector := []int{}

current := list.Head

for current != nil {

collector = append(collector, current.Data)

current = current.Next

}

return collector

}

Queue.go

*// e. Очередь. Операции: добавление и удаление элемента (push и*

*// pop), чтение.*

package queue

import doubleLinkedlist "BaseStruct/baseStruct/doubleLinkedList"

type Queue struct {

list \*doubleLinkedlist.DoublyLinkedList

}

func NewQueue() \*Queue {

return &Queue{list: &doubleLinkedlist.DoublyLinkedList{}}

}

func (q \*Queue) Enqueue(data int) {

q.list.AppendAtBack(data)

}

func (q \*Queue) Dequeue() (int, bool) {

if q.list.Head == nil {

return 0, false

}

data := q.list.Head.Data

q.list.DeleteHead()

return data, true

}

func (q \*Queue) Peek() (int, bool) {

if q.list.Head == nil {

return 0, false

}

return q.list.Head.Data, true

}

func (q \*Queue) IsEmpty() bool {

return q.list.Head == nil

}

func (q \*Queue) ToSlice() []int {

return q.list.ToSlice()

}

Stack.go

*// d. Стек. Операции: добавление и удаление элемента (push и pop),*

*// чтение.*

package stack

import "fmt"

type Stack struct {

values []int

}

func (s \*Stack) Push(value int) {

s.values = append(s.values, value)

}

func (s \*Stack) Pop() (int, bool) {

if len(s.values) == 0 {

return 0, false

}

lastIndex := len(s.values) - 1

value := s.values[lastIndex]

s.values = s.values[:lastIndex]

return value, true

}

func (s \*Stack) isEmpty() bool {

if len(s.values) == 0 {

fmt.Println("stack isempty")

return true

} else {

return false

}

}

func (s \*Stack) CheckAllElement() {

fmt.Println(s.values)

}

Main,go

*// main.go*

package main

import (

convertstruct "BaseStruct/convertStruct"

"encoding/json"

"flag"

"fmt"

"os"

"strconv"

"strings"

)

type DBFile struct {

Arrays map[string][]int `json:"arrays"`

Lists map[string][]int `json:"lists"`

DlLists map[string][]int `json:"dllists"`

Stacks map[string][]int `json:"stacks"`

Queues map[string][]int `json:"queues"`

Trees map[string][]int `json:"trees"`

}

func NewEmptyDB() \*DBFile {

return &DBFile{

Arrays: map[string][]int{},

Lists: map[string][]int{},

DlLists: map[string][]int{},

Stacks: map[string][]int{},

Queues: map[string][]int{},

Trees: map[string][]int{},

}

}

func loadDB(path string) (\*DBFile, error) {

data, err := os.ReadFile(path)

if err != nil {

if os.IsNotExist(err) {

return NewEmptyDB(), nil

}

return nil, err

}

var db DBFile

if err := json.Unmarshal(data, &db); err != nil {

return nil, err

}

if db.Arrays == nil {

db.Arrays = map[string][]int{}

}

if db.Lists == nil {

db.Lists = map[string][]int{}

}

if db.DlLists == nil {

db.DlLists = map[string][]int{}

}

if db.Stacks == nil {

db.Stacks = map[string][]int{}

}

if db.Queues == nil {

db.Queues = map[string][]int{}

}

if db.Trees == nil {

db.Trees = map[string][]int{}

}

return &db, nil

}

func saveDB(path string, db \*DBFile) error {

b, err := json.MarshalIndent(db, "", " ")

if err != nil {

return err

}

return os.WriteFile(path, b, 0644)

}

func parseInt(s string) (int, error) {

return strconv.Atoi(s)

}

func sliceToString(s []int) string {

if s == nil {

return "[]"

}

parts := make([]string, 0, len(s))

for \_, v := range s {

parts = append(parts, strconv.Itoa(v))

}

return "[" + strings.Join(parts, " ") + "]"

}

func removeFirstByValue(s []int, val int) ([]int, bool) {

for i := 0; i < len(s); i++ {

if s[i] == val {

return append(s[:i], s[i+1:]...), true

}

}

return s, false

}

func contains(s []int, val int) bool {

for \_, v := range s {

if v == val {

return true

}

}

return false

}

func treeInsert(s []int, v int) ([]int, bool) {

if contains(s, v) {

return s, false

}

return append(s, v), true

}

func treeDelete(s []int, v int) ([]int, bool) {

return removeFirstByValue(s, v)

}

func main() {

filePath := flag.String("file", "file.data", "path to data file")

query := flag.String("query", "", "query string, e.g. 'SPUSH mystack 10'")

flag.Parse()

if \*query == "" {

fmt.Println("no query provided")

return

}

db, err := loadDB(\*filePath)

if err != nil {

fmt.Println("Error loading file:", err)

return

}

*// parse query*

*// split by spaces but preserve quoted tokens (simple approach)*

parts := strings.Fields(\*query)

if len(parts) < 1 {

fmt.Println("invalid query")

return

}

verb := parts[0]

name := ""

args := []string{}

if len(parts) >= 2 {

name = parts[1]

}

if len(parts) > 2 {

args = parts[2:]

}

verb = strings.ToUpper(verb)

*// dispatch based on verb prefix (letter)*

var changed bool

var output string

switch {

*// M - массив*

case strings.HasPrefix(verb, "M"):

switch verb {

case "MPUSH":

if name == "" || len(args) < 1 {

fmt.Println("ERROR: MPUSH requires name and value")

return

}

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

db.Arrays[name] = append(db.Arrays[name], v)

changed = true

output = strconv.Itoa(v)

case "MDEL":

if name == "" || len(args) < 1 {

fmt.Println("ERROR: MDEL requires name and value")

return

}

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

newSlice, ok := removeFirstByValue(db.Arrays[name], v)

if ok {

db.Arrays[name] = newSlice

changed = true

output = strconv.Itoa(v)

} else {

output = "NOTFOUND"

}

case "MGET":

if name == "" {

fmt.Println("ERROR: MGET requires name")

return

}

if len(args) >= 1 {

*// treat arg as index*

idx, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: index must be integer")

return

}

if idx < 0 || idx >= len(db.Arrays[name]) {

output = "NULL"

} else {

output = strconv.Itoa(db.Arrays[name][idx])

}

} else {

output = sliceToString(db.Arrays[name])

}

default:

fmt.Println("Unknown M command:", verb)

return

}

*// F - односвязный список*

case strings.HasPrefix(verb, "F"):

if name == "" {

fmt.Println("ERROR: F command requires container name")

return

}

ll := convertstruct.LinkedListFromSlice(db.Lists[name])

switch verb {

case "FPUSH":

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

ll.AppendAtBack(v)

db.Lists[name] = convertstruct.SliceFromLinkedList(ll)

changed = true

output = strconv.Itoa(v)

case "FADDHEAD":

v, \_ := parseInt(args[0])

ll.AppendAtStart(v)

db.Lists[name] = convertstruct.SliceFromLinkedList(ll)

changed = true

output = strconv.Itoa(v)

case "FADDAFTER":

target, \_ := parseInt(args[0])

v, \_ := parseInt(args[1])

ll.AppendAfterValue(target, v)

db.Lists[name] = convertstruct.SliceFromLinkedList(ll)

changed = true

output = strconv.Itoa(v)

case "FADDBEFORE":

target, \_ := parseInt(args[0])

v, \_ := parseInt(args[1])

ll.AppendBeforeValue(target, v)

db.Lists[name] = convertstruct.SliceFromLinkedList(ll)

changed = true

output = strconv.Itoa(v)

case "FDEL":

v, \_ := parseInt(args[0])

ll.DeleteForValue(v)

db.Lists[name] = convertstruct.SliceFromLinkedList(ll)

changed = true

output = strconv.Itoa(v)

case "FDELHEAD":

ll.DeleteHead()

db.Lists[name] = convertstruct.SliceFromLinkedList(ll)

changed = true

output = "OK"

case "FDELTAIL":

ll.DeleteTail()

db.Lists[name] = convertstruct.SliceFromLinkedList(ll)

changed = true

output = "OK"

case "FGET":

if len(args) >= 1 {

idx, \_ := parseInt(args[0])

slice := convertstruct.SliceFromLinkedList(ll)

if idx < 0 || idx >= len(slice) {

output = "NULL"

} else {

output = strconv.Itoa(slice[idx])

}

} else {

output = sliceToString(convertstruct.SliceFromLinkedList(ll))

}

case "FEXISTS":

v, \_ := parseInt(args[0])

node := ll.FindElementForValue(v)

if node != nil {

output = "TRUE"

} else {

output = "FALSE"

}

default:

fmt.Println("Unknown F command:", verb)

return

}

*// L - двусвязный список (dllists)*

case strings.HasPrefix(verb, "L"):

if name == "" {

fmt.Println("ERROR: L command requires container name")

return

}

*// Получаем DLL из среза, если контейнера нет — создаём новый*

listl := convertstruct.DllFromSlice(db.DlLists[name])

switch verb {

case "LPUSH": *// добавить в конец*

if len(args) < 1 {

fmt.Println("ERROR: LPUSH requires value")

return

}

v, \_ := parseInt(args[0])

listl.AppendAtBack(v)

changed = true

output = strconv.Itoa(v)

case "LUNSHIFT": *// добавить в начало*

if len(args) < 1 {

fmt.Println("ERROR: LUNSHIFT requires value")

return

}

v, \_ := parseInt(args[0])

listl.AppendAtStart(v)

changed = true

output = strconv.Itoa(v)

case "LAFTER": *// добавить после значения*

if len(args) < 2 {

fmt.Println("ERROR: LAFTER requires target value and new value")

return

}

target, \_ := parseInt(args[0])

v, \_ := parseInt(args[1])

listl.AppendAfterValue(target, v)

changed = true

output = strconv.Itoa(v)

case "LBEFORE": *// добавить перед значением*

if len(args) < 2 {

fmt.Println("ERROR: LBEFORE requires target value and new value")

return

}

target, \_ := parseInt(args[0])

v, \_ := parseInt(args[1])

listl.AppendBeforeValue(target, v)

changed = true

output = strconv.Itoa(v)

case "LDEL": *// удалить по значению*

if len(args) < 1 {

fmt.Println("ERROR: LDEL requires value")

return

}

v, \_ := parseInt(args[0])

listl.DeleteForValue(v)

changed = true

output = strconv.Itoa(v)

case "LDELHEAD": *// удалить голову*

listl.DeleteHead()

changed = true

output = "OK"

case "LDELTAIL": *// удалить хвост*

listl.DeleteTail()

changed = true

output = "OK"

case "LGET": *// вывести весь список*

output = sliceToString(convertstruct.SliceFromDLL(listl))

case "LEXISTS": *// проверить наличие элемента*

if len(args) < 1 {

fmt.Println("ERROR: LEXISTS requires value")

return

}

v, \_ := parseInt(args[0])

node := listl.FindElementForValue(v)

if node != nil {

output = "TRUE"

} else {

output = "FALSE"

}

default:

fmt.Println("Unknown L command:", verb)

return

}

*// Сохраняем обратно в базу*

db.DlLists[name] = convertstruct.SliceFromDLL(listl)

*// Q - очередь*

case strings.HasPrefix(verb, "Q"):

switch verb {

case "QPUSH":

if name == "" || len(args) < 1 {

fmt.Println("ERROR: QPUSH requires name and value")

return

}

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

db.Queues[name] = append(db.Queues[name], v)

changed = true

output = strconv.Itoa(v)

case "QPOP":

if name == "" {

fmt.Println("ERROR: QPOP requires name")

return

}

if len(db.Queues[name]) == 0 {

output = "NULL"

} else {

val := db.Queues[name][0]

db.Queues[name] = db.Queues[name][1:]

changed = true

output = strconv.Itoa(val)

}

default:

fmt.Println("Unknown Q command:", verb)

return

}

*// S - стек*

case strings.HasPrefix(verb, "S"):

switch verb {

case "SPUSH":

if name == "" || len(args) < 1 {

fmt.Println("ERROR: SPUSH requires name and value")

return

}

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

db.Stacks[name] = append(db.Stacks[name], v)

changed = true

output = strconv.Itoa(v)

case "SPOP":

if name == "" {

fmt.Println("ERROR: SPOP requires name")

return

}

if len(db.Stacks[name]) == 0 {

output = "NULL"

} else {

last := len(db.Stacks[name]) - 1

val := db.Stacks[name][last]

db.Stacks[name] = db.Stacks[name][:last]

changed = true

output = strconv.Itoa(val)

}

default:

fmt.Println("Unknown S command:", verb)

return

}

*// T - дерево (set-like)*

case strings.HasPrefix(verb, "T"):

switch verb {

case "TINSERT":

if name == "" || len(args) < 1 {

fmt.Println("ERROR: TINSERT requires name and value")

return

}

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

newSlice, added := treeInsert(db.Trees[name], v)

if added {

db.Trees[name] = newSlice

changed = true

}

if added {

output = strconv.Itoa(v)

} else {

output = "ALREADY"

}

case "TDEL":

if name == "" || len(args) < 1 {

fmt.Println("ERROR: TDEL requires name and value")

return

}

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

newSlice, ok := treeDelete(db.Trees[name], v)

if ok {

db.Trees[name] = newSlice

changed = true

output = strconv.Itoa(v)

} else {

output = "NOTFOUND"

}

case "TGET":

*// TGET name value -> TRUE/FALSE (membership)*

if name == "" || len(args) < 1 {

fmt.Println("ERROR: TGET requires name and value")

return

}

v, err := parseInt(args[0])

if err != nil {

fmt.Println("ERROR: value must be integer")

return

}

if contains(db.Trees[name], v) {

output = "TRUE"

} else {

output = "FALSE"

}

default:

fmt.Println("Unknown T command:", verb)

return

}

default:

fmt.Println("Unknown command prefix:", verb)

return

}

fmt.Println("->", output)

if changed {

if err := saveDB(\*filePath, db); err != nil {

fmt.Println("Error saving file:", err)

}

}

}

Convert.go

package convertstruct

import (

doubleLinkedlist "BaseStruct/baseStruct/doubleLinkedList"

linkedlist "BaseStruct/baseStruct/linkedList"

)

*// Конвертация среза в LinkedList*

func LinkedListFromSlice(slice []int) \*linkedlist.LinkedList {

ll := &linkedlist.LinkedList{}

for \_, v := range slice {

ll.AppendAtBack(v)

}

return ll

}

*// Конвертация LinkedList обратно в срез*

func SliceFromLinkedList(ll \*linkedlist.LinkedList) []int {

return ll.ReadLinkedListForSlice()

}

*// Конвертация среза в двусвязный список*

func DllFromSlice(slice []int) \*doubleLinkedlist.DoublyLinkedList {

dll := &doubleLinkedlist.DoublyLinkedList{}

for \_, v := range slice {

dll.AppendAtBack(v)

}

return dll

}

*// Конвертация DLL обратно в срез*

func SliceFromDLL(dll \*doubleLinkedlist.DoublyLinkedList) []int {

return dll.ToSlice()

}

Go mod

module BaseStruct

go 1.25.1

Код программы на языке С++

Array.hpp

#ifndef ARRAY\_HPP

#define ARRAY\_HPP

#include <string>

#include <vector>

#include <memory>

using namespace std;

struct Array {

vector<string> data;

};

unique\_ptr<Array> array\_create();

void array\_append(Array\* arr, const string& value);

void array\_insert(Array\* arr, size\_t index, const string& value);

string array\_get(const Array\* arr, size\_t index);

void array\_remove(Array\* arr, size\_t index);

void array\_set(Array\* arr, size\_t index, const string& value);

bool array\_remove\_by\_value(Array\* arr, const string& value);

size\_t array\_size(const Array\* arr);

bool array\_empty(const Array\* arr);

#endif *// ARRAY\_HPP*

AVLtree.hpp

#ifndef AVL\_TREE\_HPP

#define AVL\_TREE\_HPP

#include <memory>

#include <cstdint>

using namespace std;

struct TreeNode {

int key;

uint8\_t height;

unique\_ptr<TreeNode> left;

unique\_ptr<TreeNode> right;

};

struct AVLTree {

unique\_ptr<TreeNode> root;

};

unique\_ptr<AVLTree> avl\_create();

void avl\_insert(AVLTree\* tree, int key);

void avl\_delete(AVLTree\* tree, int key);

bool avl\_search(const AVLTree\* tree, int key);

#endif *// AVL\_TREE\_HPP*

Database.hpp

#ifndef DATABASE\_HPP

#define DATABASE\_HPP

#include "Array.hpp"

#include "LinkedList.hpp"

#include "DoubleLinkedList.hpp"

#include "Stack.hpp"

#include "Queue.hpp"

#include "AVLTree.hpp"

#include <string>

#include <memory>

#include <unordered\_map>

using namespace std;

struct Database {

unordered\_map<string, unique\_ptr<Array>> arrays;

unordered\_map<string, unique\_ptr<LinkedList>> lists;

unordered\_map<string, unique\_ptr<DoubleLinkedList>> dlists;

unordered\_map<string, unique\_ptr<Stack>> stacks;

unordered\_map<string, unique\_ptr<Queue>> queues;

unordered\_map<string, unique\_ptr<AVLTree>> trees;

};

unique\_ptr<Database> db\_create();

Array\* db\_get\_array(Database\* db, const string& name);

LinkedList\* db\_get\_list(Database\* db, const string& name);

DoubleLinkedList\* db\_get\_dlist(Database\* db, const string& name);

Stack\* db\_get\_stack(Database\* db, const string& name);

Queue\* db\_get\_queue(Database\* db, const string& name);

AVLTree\* db\_get\_tree(Database\* db, const string& name);

bool db\_load(Database\* db, const string& filepath);

bool db\_save(const Database\* db, const string& filepath);

#endif *// DATABASE\_HPP*

Doublelinkedlist.hpp

#ifndef DOUBLE\_LINKED\_LIST\_HPP

#define DOUBLE\_LINKED\_LIST\_HPP

#include <string>

#include <memory>

using namespace std;

struct DNode {

string data;

unique\_ptr<DNode> next;

DNode\* prev;

};

struct DoubleLinkedList {

unique\_ptr<DNode> head;

DNode\* tail;

};

unique\_ptr<DoubleLinkedList> dll\_create();

void dll\_append\_back(DoubleLinkedList\* list, const string& data);

void dll\_append\_front(DoubleLinkedList\* list, const string& data);

void dll\_append\_after(DoubleLinkedList\* list, const string& target, const string& data);

void dll\_append\_before(DoubleLinkedList\* list, const string& target, const string& data);

void dll\_delete\_head(DoubleLinkedList\* list);

void dll\_delete\_tail(DoubleLinkedList\* list);

void dll\_delete\_by\_value(DoubleLinkedList\* list, const string& value);

bool dll\_find(const DoubleLinkedList\* list, const string& value);

bool dll\_empty(const DoubleLinkedList\* list);

DNode\* dll\_begin(const DoubleLinkedList\* list);

#endif *// DOUBLE\_LINKED\_LIST\_HPP*

Linkedlist.hpp

#ifndef LINKED\_LIST\_HPP

#define LINKED\_LIST\_HPP

#include <string>

#include <memory>

using namespace std;

struct Node {

string data;

unique\_ptr<Node> next;

};

struct LinkedList {

unique\_ptr<Node> head;

};

unique\_ptr<LinkedList> ll\_create();

void ll\_append\_back(LinkedList\* list, const string& data);

void ll\_append\_front(LinkedList\* list, const string& data);

void ll\_append\_after(LinkedList\* list, const string& target, const string& data);

void ll\_append\_before(LinkedList\* list, const string& target, const string& data);

void ll\_delete\_head(LinkedList\* list);

void ll\_delete\_tail(LinkedList\* list);

void ll\_delete\_by\_value(LinkedList\* list, const string& value);

bool ll\_find(const LinkedList\* list, const string& value);

string ll\_get\_by\_index(const LinkedList\* list, size\_t index);

bool ll\_empty(const LinkedList\* list);

Node\* ll\_begin(const LinkedList\* list);

#endif *// LINKED\_LIST\_HPP*

Queue.hpp

#ifndef QUEUE\_HPP

#define QUEUE\_HPP

#include "DoubleLinkedList.hpp"

#include <memory>

#include <optional>

using namespace std;

struct Queue {

unique\_ptr<DoubleLinkedList> list;

};

unique\_ptr<Queue> queue\_create();

void queue\_enqueue(Queue\* queue, const string& data);

optional<string> queue\_dequeue(Queue\* queue);

bool queue\_is\_empty(const Queue\* queue);

#endif *// QUEUE\_HPP*

Stack.hpp

#ifndef STACK\_HPP

#define STACK\_HPP

#include <string>

#include <vector>

#include <optional>

#include <memory>

using namespace std;

struct Stack {

vector<string> data;

};

unique\_ptr<Stack> stack\_create();

void stack\_push(Stack\* stack, const string& value);

optional<string> stack\_pop(Stack\* stack);

bool stack\_is\_empty(const Stack\* stack);

size\_t stack\_size(const Stack\* stack);

#endif *// STACK\_HPP*

Array.cpp

*/\**

*\* Реализация динамического массива строк на C++*

*\* Использует vector для автоматического управления памятью*

*\*/*

#include "../include/Array.hpp"

#include <algorithm>

using namespace std;

unique\_ptr<Array> array\_create() {

return make\_unique<Array>();

}

void array\_append(Array\* arr, const string& value) {

arr->data.push\_back(value);

}

void array\_insert(Array\* arr, size\_t index, const string& value) {

if (index > arr->data.size()) return;

arr->data.insert(arr->data.begin() + index, value);

}

string array\_get(const Array\* arr, size\_t index) {

if (index >= arr->data.size()) return "";

return arr->data[index];

}

void array\_remove(Array\* arr, size\_t index) {

if (index >= arr->data.size()) return;

arr->data.erase(arr->data.begin() + index);

}

void array\_set(Array\* arr, size\_t index, const string& value) {

if (index >= arr->data.size()) return;

arr->data[index] = value;

}

bool array\_remove\_by\_value(Array\* arr, const string& value) {

auto it = find(arr->data.begin(), arr->data.end(), value);

if (it != arr->data.end()) {

arr->data.erase(it);

return true;

}

return false;

}

size\_t array\_size(const Array\* arr) {

return arr->data.size();

}

bool array\_empty(const Array\* arr) {

return arr->data.empty();

}

AVLtree.cpp

*/\**

*\* Реализация AVL-дерева (самобалансирующееся бинарное дерево поиска) на C++*

*\* Использует unique\_ptr для автоматического управления памятью*

*\*/*

#include "../include/AVLTree.hpp"

using namespace std;

*// Вспомогательные функции*

static uint8\_t max\_height(uint8\_t a, uint8\_t b) {

return a > b ? a : b;

}

static uint8\_t node\_height(const TreeNode\* node) {

return node ? node->height : 0;

}

static int balance\_factor(const TreeNode\* node) {

return node ? static\_cast<int>(node\_height(node->left.get())) -

static\_cast<int>(node\_height(node->right.get())) : 0;

}

static void fix\_height(TreeNode\* node) {

if (!node) return;

node->height = max\_height(node\_height(node->left.get()), node\_height(node->right.get())) + 1;

}

static unique\_ptr<TreeNode> rotate\_right(unique\_ptr<TreeNode> p) {

if (!p || !p->left) return p;

auto q = move(p->left);

p->left = move(q->right);

fix\_height(p.get());

fix\_height(q.get());

q->right = move(p);

return q;

}

static unique\_ptr<TreeNode> rotate\_left(unique\_ptr<TreeNode> q) {

if (!q || !q->right) return q;

auto p = move(q->right);

q->right = move(p->left);

fix\_height(q.get());

fix\_height(p.get());

p->left = move(q);

return p;

}

static unique\_ptr<TreeNode> balance(unique\_ptr<TreeNode> p) {

if (!p) return nullptr;

fix\_height(p.get());

int bf = balance\_factor(p.get());

if (bf == 2) {

if (balance\_factor(p->left.get()) < 0) {

p->left = rotate\_left(move(p->left));

}

return rotate\_right(move(p));

}

if (bf == -2) {

if (balance\_factor(p->right.get()) > 0) {

p->right = rotate\_right(move(p->right));

}

return rotate\_left(move(p));

}

return p;

}

static unique\_ptr<TreeNode> insert\_node(unique\_ptr<TreeNode> node, int key) {

if (!node) {

auto newNode = make\_unique<TreeNode>();

newNode->key = key;

newNode->height = 1;

newNode->left = nullptr;

newNode->right = nullptr;

return newNode;

}

if (key < node->key) {

node->left = insert\_node(move(node->left), key);

} else if (key > node->key) {

node->right = insert\_node(move(node->right), key);

} else {

return node;

}

return balance(move(node));

}

static TreeNode\* find\_min(TreeNode\* node) {

return node->left ? find\_min(node->left.get()) : node;

}

static unique\_ptr<TreeNode> remove\_min(unique\_ptr<TreeNode> node) {

if (!node->left) return move(node->right);

node->left = remove\_min(move(node->left));

return balance(move(node));

}

static unique\_ptr<TreeNode> delete\_node(unique\_ptr<TreeNode> node, int key) {

if (!node) return nullptr;

if (key < node->key) {

node->left = delete\_node(move(node->left), key);

} else if (key > node->key) {

node->right = delete\_node(move(node->right), key);

} else {

auto left = move(node->left);

auto right = move(node->right);

if (!right) return left;

TreeNode\* minNode = find\_min(right.get());

auto newNode = make\_unique<TreeNode>();

newNode->key = minNode->key;

newNode->height = 1;

newNode->right = remove\_min(move(right));

newNode->left = move(left);

return balance(move(newNode));

}

return balance(move(node));

}

static bool search\_node(const TreeNode\* node, int key) {

if (!node) return false;

if (key == node->key) return true;

if (key < node->key) return search\_node(node->left.get(), key);

return search\_node(node->right.get(), key);

}

*// Публичные функции*

unique\_ptr<AVLTree> avl\_create() {

auto tree = make\_unique<AVLTree>();

tree->root = nullptr;

return tree;

}

void avl\_insert(AVLTree\* tree, int key) {

tree->root = insert\_node(move(tree->root), key);

}

void avl\_delete(AVLTree\* tree, int key) {

tree->root = delete\_node(move(tree->root), key);

}

bool avl\_search(const AVLTree\* tree, int key) {

return search\_node(tree->root.get(), key);

}

Database.cpp

*/\**

*\* Модуль управления базой данных структур на C++*

*\* Использует unordered\_map для хранения контейнеров*

*\*/*

#include "../include/Database.hpp"

using namespace std;

#include <fstream>

#include <sstream>

unique\_ptr<Database> db\_create() {

return make\_unique<Database>();

}

Array\* db\_get\_array(Database\* db, const string& name) {

auto it = db->arrays.find(name);

if (it != db->arrays.end()) {

return it->second.get();

}

auto newArray = array\_create();

Array\* ptr = newArray.get();

db->arrays[name] = move(newArray);

return ptr;

}

LinkedList\* db\_get\_list(Database\* db, const string& name) {

auto it = db->lists.find(name);

if (it != db->lists.end()) {

return it->second.get();

}

auto newList = ll\_create();

LinkedList\* ptr = newList.get();

db->lists[name] = move(newList);

return ptr;

}

DoubleLinkedList\* db\_get\_dlist(Database\* db, const string& name) {

auto it = db->dlists.find(name);

if (it != db->dlists.end()) {

return it->second.get();

}

auto newDList = dll\_create();

DoubleLinkedList\* ptr = newDList.get();

db->dlists[name] = move(newDList);

return ptr;

}

Stack\* db\_get\_stack(Database\* db, const string& name) {

auto it = db->stacks.find(name);

if (it != db->stacks.end()) {

return it->second.get();

}

auto newStack = stack\_create();

Stack\* ptr = newStack.get();

db->stacks[name] = move(newStack);

return ptr;

}

Queue\* db\_get\_queue(Database\* db, const string& name) {

auto it = db->queues.find(name);

if (it != db->queues.end()) {

return it->second.get();

}

auto newQueue = queue\_create();

Queue\* ptr = newQueue.get();

db->queues[name] = move(newQueue);

return ptr;

}

AVLTree\* db\_get\_tree(Database\* db, const string& name) {

auto it = db->trees.find(name);

if (it != db->trees.end()) {

return it->second.get();

}

auto newTree = avl\_create();

AVLTree\* ptr = newTree.get();

db->trees[name] = move(newTree);

return ptr;

}

bool db\_save(const Database\* db, const string& filepath) {

ofstream file(filepath);

if (!file.is\_open()) return false;

*// Сохраняем массивы*

file << "ARRAYS " << db->arrays.size() << "\n";

for (const auto& [name, arr] : db->arrays) {

file << name << " " << array\_size(arr.get()) << "\n";

for (size\_t i = 0; i < array\_size(arr.get()); i++) {

file << array\_get(arr.get(), i) << "\n";

}

}

*// Сохраняем односвязные списки*

file << "LISTS " << db->lists.size() << "\n";

for (const auto& [name, list] : db->lists) {

file << name << " ";

int count = 0;

Node\* curr = ll\_begin(list.get());

while (curr) {

count++;

curr = curr->next.get();

}

file << count << "\n";

curr = ll\_begin(list.get());

while (curr) {

file << curr->data << "\n";

curr = curr->next.get();

}

}

*// Сохраняем двусвязные списки*

file << "DLISTS " << db->dlists.size() << "\n";

for (const auto& [name, dlist] : db->dlists) {

file << name << " ";

int count = 0;

DNode\* curr = dll\_begin(dlist.get());

while (curr) {

count++;

curr = curr->next.get();

}

file << count << "\n";

curr = dll\_begin(dlist.get());

while (curr) {

file << curr->data << "\n";

curr = curr->next.get();

}

}

*// Сохраняем стеки*

file << "STACKS " << db->stacks.size() << "\n";

for (const auto& [name, stack] : db->stacks) {

file << name << " " << stack\_size(stack.get()) << "\n";

for (size\_t i = 0; i < stack\_size(stack.get()); i++) {

file << stack->data[i] << "\n";

}

}

*// Сохраняем очереди*

file << "QUEUES " << db->queues.size() << "\n";

for (const auto& [name, queue] : db->queues) {

file << name << " ";

int count = 0;

DNode\* curr = dll\_begin(queue->list.get());

while (curr) {

count++;

curr = curr->next.get();

}

file << count << "\n";

curr = dll\_begin(queue->list.get());

while (curr) {

file << curr->data << "\n";

curr = curr->next.get();

}

}

*// Сохраняем деревья*

file << "TREES " << db->trees.size() << "\n";

for (const auto& [name, tree] : db->trees) {

file << name << "\n";

}

file.close();

return true;

}

bool db\_load(Database\* db, const string& filepath) {

ifstream file(filepath);

if (!file.is\_open()) return false;

string type;

int count;

*// Загружаем массивы*

file >> type >> count;

if (type == "ARRAYS") {

for (int i = 0; i < count; i++) {

string name;

size\_t size;

file >> name >> size;

file.ignore(); *// пропускаем \n*

Array\* arr = db\_get\_array(db, name);

for (size\_t j = 0; j < size; j++) {

string value;

getline(file, value);

array\_append(arr, value);

}

}

}

*// Загружаем односвязные списки*

file >> type >> count;

if (type == "LISTS") {

for (int i = 0; i < count; i++) {

string name;

int size;

file >> name >> size;

file.ignore();

LinkedList\* list = db\_get\_list(db, name);

for (int j = 0; j < size; j++) {

string value;

getline(file, value);

ll\_append\_back(list, value);

}

}

}

*// Загружаем двусвязные списки*

file >> type >> count;

if (type == "DLISTS") {

for (int i = 0; i < count; i++) {

string name;

int size;

file >> name >> size;

file.ignore();

DoubleLinkedList\* dlist = db\_get\_dlist(db, name);

for (int j = 0; j < size; j++) {

string value;

getline(file, value);

dll\_append\_back(dlist, value);

}

}

}

*// Загружаем стеки*

file >> type >> count;

if (type == "STACKS") {

for (int i = 0; i < count; i++) {

string name;

size\_t size;

file >> name >> size;

file.ignore();

Stack\* stack = db\_get\_stack(db, name);

for (size\_t j = 0; j < size; j++) {

string value;

getline(file, value);

stack\_push(stack, value);

}

}

}

*// Загружаем очереди*

file >> type >> count;

if (type == "QUEUES") {

for (int i = 0; i < count; i++) {

string name;

int size;

file >> name >> size;

file.ignore();

Queue\* queue = db\_get\_queue(db, name);

for (int j = 0; j < size; j++) {

string value;

getline(file, value);

queue\_enqueue(queue, value);

}

}

}

*// Загружаем деревья*

file >> type >> count;

if (type == "TREES") {

for (int i = 0; i < count; i++) {

string name;

file >> name;

db\_get\_tree(db, name);

}

}

file.close();

return true;

}

Doublelinkedlist.cpp

*/\**

*\* Реализация двусвязного списка на C++*

*\* Использует unique\_ptr для автоматического управления памятью*

*\*/*

#include "../include/DoubleLinkedList.hpp"

using namespace std;

unique\_ptr<DoubleLinkedList> dll\_create() {

auto list = make\_unique<DoubleLinkedList>();

list->head = nullptr;

list->tail = nullptr;

return list;

}

void dll\_append\_back(DoubleLinkedList\* list, const string& data) {

auto newNode = make\_unique<DNode>();

newNode->data = data;

newNode->next = nullptr;

newNode->prev = list->tail;

if (!list->tail) {

list->tail = newNode.get();

list->head = move(newNode);

} else {

list->tail->next = move(newNode);

list->tail = list->tail->next.get();

}

}

void dll\_append\_front(DoubleLinkedList\* list, const string& data) {

auto newNode = make\_unique<DNode>();

newNode->data = data;

newNode->next = move(list->head);

newNode->prev = nullptr;

if (!newNode->next) {

list->head = move(newNode);

list->tail = list->head.get();

} else {

newNode->next->prev = newNode.get();

list->head = move(newNode);

}

}

void dll\_append\_after(DoubleLinkedList\* list, const string& target, const string& data) {

DNode\* current = list->head.get();

while (current && current->data != target) {

current = current->next.get();

}

if (!current) return;

auto newNode = make\_unique<DNode>();

newNode->data = data;

newNode->next = move(current->next);

newNode->prev = current;

if (newNode->next) {

newNode->next->prev = newNode.get();

} else {

list->tail = newNode.get();

}

current->next = move(newNode);

}

void dll\_append\_before(DoubleLinkedList\* list, const string& target, const string& data) {

DNode\* current = list->head.get();

while (current && current->data != target) {

current = current->next.get();

}

if (!current) return;

if (!current->prev) {

dll\_append\_front(list, data);

return;

}

auto newNode = make\_unique<DNode>();

newNode->data = data;

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

newNode->prev = current->prev;

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

current->prev = current->prev->next.get();

}

void dll\_delete\_head(DoubleLinkedList\* list) {

if (!list->head) return;

list->head = move(list->head->next);

if (list->head) {

list->head->prev = nullptr;

} else {

list->tail = nullptr;

}

}

void dll\_delete\_tail(DoubleLinkedList\* list) {

if (!list->tail) return;

if (list->tail->prev) {

list->tail = list->tail->prev;

list->tail->next.reset();

} else {

list->head.reset();

list->tail = nullptr;

}

}

void dll\_delete\_by\_value(DoubleLinkedList\* list, const string& value) {

DNode\* current = list->head.get();

while (current && current->data != value) {

current = current->next.get();

}

if (!current) return;

if (current->prev) {

current->prev->next = move(current->next);

if (current->next) {

current->next->prev = current->prev;

} else {

list->tail = current->prev;

}

} else {

dll\_delete\_head(list);

}

}

bool dll\_find(const DoubleLinkedList\* list, const string& value) {

DNode\* current = list->head.get();

while (current) {

if (current->data == value) {

return true;

}

current = current->next.get();

}

return false;

}

bool dll\_empty(const DoubleLinkedList\* list) {

return list->head == nullptr;

}

DNode\* dll\_begin(const DoubleLinkedList\* list) {

return list->head.get();

}

Linkedlist.cpp

*/\**

*\* Реализация односвязного списка на C++*

*\* Использует unique\_ptr для автоматического управления памятью*

*\*/*

#include "../include/LinkedList.hpp"

using namespace std;

unique\_ptr<LinkedList> ll\_create() {

auto list = make\_unique<LinkedList>();

list->head = nullptr;

return list;

}

void ll\_append\_back(LinkedList\* list, const string& data) {

auto newNode = make\_unique<Node>();

newNode->data = data;

newNode->next = nullptr;

if (!list->head) {

list->head = move(newNode);

return;

}

Node\* current = list->head.get();

while (current->next) {

current = current->next.get();

}

current->next = move(newNode);

}

void ll\_append\_front(LinkedList\* list, const string& data) {

auto newNode = make\_unique<Node>();

newNode->data = data;

newNode->next = move(list->head);

list->head = move(newNode);

}

void ll\_append\_after(LinkedList\* list, const string& target, const string& data) {

Node\* current = list->head.get();

while (current && current->data != target) {

current = current->next.get();

}

if (!current) return;

auto newNode = make\_unique<Node>();

newNode->data = data;

newNode->next = move(current->next);

current->next = move(newNode);

}

void ll\_append\_before(LinkedList\* list, const string& target, const string& data) {

if (!list->head) return;

if (list->head->data == target) {

ll\_append\_front(list, data);

return;

}

Node\* prev = list->head.get();

Node\* current = list->head->next.get();

while (current && current->data != target) {

prev = current;

current = current->next.get();

}

if (!current) return;

auto newNode = make\_unique<Node>();

newNode->data = data;

newNode->next = move(prev->next);

prev->next = move(newNode);

}

void ll\_delete\_head(LinkedList\* list) {

if (!list->head) return;

list->head = move(list->head->next);

}

void ll\_delete\_tail(LinkedList\* list) {

if (!list->head) return;

if (!list->head->next) {

list->head.reset();

return;

}

Node\* current = list->head.get();

while (current->next && current->next->next) {

current = current->next.get();

}

current->next.reset();

}

void ll\_delete\_by\_value(LinkedList\* list, const string& value) {

if (!list->head) return;

if (list->head->data == value) {

ll\_delete\_head(list);

return;

}

Node\* prev = list->head.get();

Node\* current = list->head->next.get();

while (current && current->data != value) {

prev = current;

current = current->next.get();

}

if (!current) return;

prev->next = move(current->next);

}

bool ll\_find(const LinkedList\* list, const string& value) {

Node\* current = list->head.get();

while (current) {

if (current->data == value) {

return true;

}

current = current->next.get();

}

return false;

}

string ll\_get\_by\_index(const LinkedList\* list, size\_t index) {

Node\* current = list->head.get();

size\_t i = 0;

while (current && i < index) {

current = current->next.get();

i++;

}

return current ? current->data : "";

}

bool ll\_empty(const LinkedList\* list) {

return list->head == nullptr;

}

Node\* ll\_begin(const LinkedList\* list) {

return list->head.get();

}

Queue.cpp

*/\**

*\* Реализация очереди (FIFO - First In First Out) на C++*

*\* Использует двусвязный список для эффективных операций с обоих концов*

*\*/*

#include "../include/Queue.hpp"

using namespace std;

unique\_ptr<Queue> queue\_create() {

auto queue = make\_unique<Queue>();

queue->list = dll\_create();

return queue;

}

void queue\_enqueue(Queue\* queue, const string& data) {

dll\_append\_back(queue->list.get(), data);

}

optional<string> queue\_dequeue(Queue\* queue) {

if (dll\_empty(queue->list.get())) {

return nullopt;

}

DNode\* node = dll\_begin(queue->list.get());

string result = node->data;

dll\_delete\_head(queue->list.get());

return result;

}

bool queue\_is\_empty(const Queue\* queue) {

return dll\_empty(queue->list.get());

}

Stack.cpp

*/\**

*\* Реализация стека (LIFO - Last In First Out) на C++*

*\* Использует vector для автоматического управления памятью*

*\*/*

#include "../include/Stack.hpp"

using namespace std;

unique\_ptr<Stack> stack\_create() {

return make\_unique<Stack>();

}

void stack\_push(Stack\* stack, const string& value) {

stack->data.push\_back(value);

}

optional<string> stack\_pop(Stack\* stack) {

if (stack->data.empty()) {

return nullopt;

}

string value = stack->data.back();

stack->data.pop\_back();

return value;

}

bool stack\_is\_empty(const Stack\* stack) {

return stack->data.empty();

}

size\_t stack\_size(const Stack\* stack) {

return stack->data.size();

}

Main.cpp

using namespace std;

#include <iostream>

#include <string>

#include <sstream>

#include <algorithm>

#include <cctype>

#include "include/Database.hpp"

*// Преобразование строки в верхний регистр*

string toUpper(string str) {

transform(str.begin(), str.end(), str.begin(),

[](unsigned char c) { return toupper(c); });

return str;

}

*// Вывод содержимого массива*

void printArray(const Array\* arr) {

cout << "[";

for (size\_t i = 0; i < array\_size(arr); i++) {

cout << array\_get(arr, i);

if (i < array\_size(arr) - 1) cout << " ";

}

cout << "]";

}

*// Вывод содержимого односвязного списка*

void printList(const LinkedList\* list) {

cout << "[";

bool first = true;

Node\* curr = ll\_begin(list);

while (curr) {

if (!first) cout << " ";

cout << curr->data;

first = false;

curr = curr->next.get();

}

cout << "]";

}

*// Вывод содержимого двусвязного списка*

void printDList(const DoubleLinkedList\* dlist) {

cout << "[";

bool first = true;

DNode\* curr = dll\_begin(dlist);

while (curr) {

if (!first) cout << " ";

cout << curr->data;

first = false;

curr = curr->next.get();

}

cout << "]";

}

int main(int argc, char\* argv[]) {

string filepath = "file.data";

string query;

*// Парсинг аргументов*

for (int i = 1; i < argc; i++) {

if (string(argv[i]) == "--file" && i + 1 < argc) {

filepath = argv[++i];

} else if (string(argv[i]) == "--query" && i + 1 < argc) {

query = argv[++i];

}

}

if (query.empty()) {

cout << "ERROR: no query provided\n";

cout << "Usage: " << argv[0] << " --file <filepath> --query <query>\n";

return 1;

}

auto db = db\_create();

db\_load(db.get(), filepath);

*// Парсинг команды*

istringstream iss(query);

string verb, name, arg1, arg2;

iss >> verb >> name >> arg1 >> arg2;

verb = toUpper(verb);

bool changed = false;

*// Массивы (M)*

if (verb[0] == 'M') {

Array\* arr = db\_get\_array(db.get(), name);

if (verb == "MPUSH") {

array\_append(arr, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "MDEL") {

if (array\_remove\_by\_value(arr, arg1)) {

changed = true;

cout << "-> " << arg1 << "\n";

} else {

cout << "-> NOTFOUND\n";

}

}

else if (verb == "MGET") {

if (!arg1.empty()) {

int idx = stoi(arg1);

string val = array\_get(arr, idx);

if (!val.empty()) {

cout << "-> " << val << "\n";

} else {

cout << "-> NULL\n";

}

} else {

cout << "-> ";

printArray(arr);

cout << "\n";

}

}

}

*// Односвязный список (F)*

else if (verb[0] == 'F') {

LinkedList\* list = db\_get\_list(db.get(), name);

if (verb == "FPUSH") {

ll\_append\_back(list, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "FADDHEAD") {

ll\_append\_front(list, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "FADDAFTER") {

ll\_append\_after(list, arg1, arg2);

changed = true;

cout << "-> " << arg2 << "\n";

}

else if (verb == "FADDBEFORE") {

ll\_append\_before(list, arg1, arg2);

changed = true;

cout << "-> " << arg2 << "\n";

}

else if (verb == "FDEL") {

ll\_delete\_by\_value(list, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "FDELHEAD") {

ll\_delete\_head(list);

changed = true;

cout << "-> OK\n";

}

else if (verb == "FDELTAIL") {

ll\_delete\_tail(list);

changed = true;

cout << "-> OK\n";

}

else if (verb == "FGET") {

if (!arg1.empty()) {

int idx = stoi(arg1);

string val = ll\_get\_by\_index(list, idx);

if (!val.empty()) {

cout << "-> " << val << "\n";

} else {

cout << "-> NULL\n";

}

} else {

cout << "-> ";

printList(list);

cout << "\n";

}

}

else if (verb == "FEXISTS") {

cout << "-> " << (ll\_find(list, arg1) ? "TRUE" : "FALSE") << "\n";

}

}

*// Двусвязный список (L)*

else if (verb[0] == 'L') {

DoubleLinkedList\* dlist = db\_get\_dlist(db.get(), name);

if (verb == "LPUSH") {

dll\_append\_back(dlist, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "LUNSHIFT") {

dll\_append\_front(dlist, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "LAFTER") {

dll\_append\_after(dlist, arg1, arg2);

changed = true;

cout << "-> " << arg2 << "\n";

}

else if (verb == "LBEFORE") {

dll\_append\_before(dlist, arg1, arg2);

changed = true;

cout << "-> " << arg2 << "\n";

}

else if (verb == "LDEL") {

dll\_delete\_by\_value(dlist, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "LDELHEAD") {

dll\_delete\_head(dlist);

changed = true;

cout << "-> OK\n";

}

else if (verb == "LDELTAIL") {

dll\_delete\_tail(dlist);

changed = true;

cout << "-> OK\n";

}

else if (verb == "LGET") {

cout << "-> ";

printDList(dlist);

cout << "\n";

}

else if (verb == "LEXISTS") {

cout << "-> " << (dll\_find(dlist, arg1) ? "TRUE" : "FALSE") << "\n";

}

}

*// Стек (S)*

else if (verb[0] == 'S') {

Stack\* stack = db\_get\_stack(db.get(), name);

if (verb == "SPUSH") {

stack\_push(stack, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "SPOP") {

auto val = stack\_pop(stack);

if (val.has\_value()) {

changed = true;

cout << "-> " << val.value() << "\n";

} else {

cout << "-> NULL\n";

}

}

}

*// Очередь (Q)*

else if (verb[0] == 'Q') {

Queue\* queue = db\_get\_queue(db.get(), name);

if (verb == "QPUSH") {

queue\_enqueue(queue, arg1);

changed = true;

cout << "-> " << arg1 << "\n";

}

else if (verb == "QPOP") {

auto val = queue\_dequeue(queue);

if (val.has\_value()) {

changed = true;

cout << "-> " << val.value() << "\n";

} else {

cout << "-> NULL\n";

}

}

}

*// Дерево (T)*

else if (verb[0] == 'T') {

AVLTree\* tree = db\_get\_tree(db.get(), name);

if (verb == "TINSERT") {

int key = stoi(arg1);

if (!avl\_search(tree, key)) {

avl\_insert(tree, key);

changed = true;

cout << "-> " << key << "\n";

} else {

cout << "-> ALREADY\n";

}

}

else if (verb == "TDEL") {

int key = stoi(arg1);

if (avl\_search(tree, key)) {

avl\_delete(tree, key);

changed = true;

cout << "-> " << key << "\n";

} else {

cout << "-> NOTFOUND\n";

}

}

else if (verb == "TGET") {

int key = stoi(arg1);

cout << "-> " << (avl\_search(tree, key) ? "TRUE" : "FALSE") << "\n";

}

}

if (changed) {

db\_save(db.get(), filepath);

}

return 0;

}

Makefile

*# Makefile для C++ версии проекта*

CXX = g++

CXXFLAGS = -std=c++17 -Wall -Wextra -I.

TARGET = dbms\_cpp

SRC\_DIR = src

INCLUDE\_DIR = include

*# Исходные файлы*

SOURCES = main.cpp \

$(SRC\_DIR)/Array.cpp \

$(SRC\_DIR)/LinkedList.cpp \

$(SRC\_DIR)/DoubleLinkedList.cpp \

$(SRC\_DIR)/Stack.cpp \

$(SRC\_DIR)/Queue.cpp \

$(SRC\_DIR)/AVLTree.cpp \

$(SRC\_DIR)/Database.cpp

*# Объектные файлы*

OBJECTS = $(SOURCES:.cpp=.o)

*# Заголовочные файлы*

HEADERS = $(INCLUDE\_DIR)/Array.hpp \

$(INCLUDE\_DIR)/LinkedList.hpp \

$(INCLUDE\_DIR)/DoubleLinkedList.hpp \

$(INCLUDE\_DIR)/Stack.hpp \

$(INCLUDE\_DIR)/Queue.hpp \

$(INCLUDE\_DIR)/AVLTree.hpp \

$(INCLUDE\_DIR)/Database.hpp

*# Основная цель*

all: $(TARGET)

*# Компиляция исполняемого файла*

$(TARGET): $(OBJECTS)

$(CXX) $(CXXFLAGS) -o $@ $^

*# Компиляция объектных файлов*

%.o: %.cpp $(HEADERS)

$(CXX) $(CXXFLAGS) -c $< -o $@

*# Очистка*

clean:

rm -f $(OBJECTS) $(TARGET)

*# Перекомпиляция*

rebuild: clean all

.PHONY: all clean rebuild

# Команды для работы программы

*Формат запуска:*

./dbms --query "КОМАНДА контейнер [аргументы]"

*1,Массив (Array) - префикс M*

MPUSH <имя> <значение> - Добавить элемент в конец массива

MDEL <имя> <значение> - Удалить первое вхождение элемента

MGET <имя> [индекс] - Получить элемент по индексу или весь массив

*2,Односвязный список (Linked List) - префикс F*

FPUSH <имя> <значение> - Добавить в конец списка

FADDHEAD <имя> <значение> - Добавить в начало списка

FADDAFTER <имя> <цель> <значение> - Добавить после элемента

FADDBEFORE <имя> <цель> <значение> - Добавить перед элементом

FDEL <имя> <значение> - Удалить элемент по значению

FDELHEAD <имя> - Удалить голову списка

FDELTAIL <имя> - Удалить хвост списка

FGET <имя> [индекс] - Получить элемент или весь список

FEXISTS <имя> <значение> - Проверить наличие элемента

*3. Двусвязный список (double linked list) - префикс l*

*LPUSH <имя> <значение> - Добавить в конец списка*

*LUNSHIFT <имя> <значение> - Добавить в начало списка*

*LAFTER <имя> <цель> <значение> - Добавить после элемента*

*LBEFORE <имя> <цель> <значение> - Добавить перед элементом*

*LDEL <имя> <значение> - Удалить элемент по значению*

*LDELHEAD <имя> - Удалить голову списка*

*LDELTAIL <имя> - Удалить хвост списка*

*LGET <имя> - Получить весь список*

*LEXISTS <имя> <значение> - Проверить наличие элемента*

*4. СТЕК (Stack) - префикс S (LIFO - Last In First Out)*

*SPUSH <имя> <значение> - Добавить элемент в стек (push)*

*SPOP <имя> - Извлечь элемент из стека (pop)*

*5. ОЧЕРЕДЬ (Queue) - префикс Q (FIFO - First In First Out)*

*QPUSH <имя> <значение> - Добавить элемент в очередь (enqueue)*

*QPOP <имя> - Извлечь элемент из очереди (dequeue)*

*6.AVL-ДЕРЕВО (AVL Tree) - префикс T (только числа)*

*TINSERT <имя> <число> - Вставить число в дерево*

*TDEL <имя> <число> - Удалить число из дерева*

*TGET <имя> <число> - Проверить наличие числа (TRUE/FALSE)*

# Таблица Big O для операциям и структурам

Массив

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Средняя сложность** | **Худшая сложность** | **Комментарий** |
| Создание | O(n) | O(n) | Выделение памяти под n элементов |
| Добавление в конец | O(1)\* | O(n)\* | Амортизированная вставка при увеличении ёмкости |
| Добавление по индексу | O(n) | O(n) | Сдвиг элементов |
| Получение по индексу | O(1) | O(1) | Прямой доступ по адресу |
| Удаление по индексу | O(n) | O(n) | Сдвиг элементов влево |
| Замена по индексу | O(1) | O(1) | Прямой доступ |
| Длина массива | O(1) | O(1) | Хранится в поле |
| Чтение (всего массива) | O(n) | O(n) | Последовательный обход |

Односвязный список

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Средняя сложность** | **Худшая сложность** | **Комментарий** |
| Добавление в начало | O(1) | O(1) | Изменение указателя head |
| Добавление в конец | O(n) | O(n) | Проход до последнего элемента |
| Добавление после узла | O(1) | O(1) | Переназначение ссылок |
| Добавление до узла | O(n) | O(n) | Поиск предыдущего элемента |
| Удаление головы | O(1) | O(1) | Смена head |
| Удаление хвоста | O(n) | O(n) | Поиск предпоследнего узла |
| Удаление по значению | O(n) | O(n) | Последовательный поиск |
| Поиск по значению | O(n) | O(n) | Последовательный проход |
| Чтение по индексу | O(n) | O(n) | Последовательный проход |

Двусвязный список

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Средняя сложность** | **Худшая сложность** | **Комментарий** |
| Добавление в начало | O(1) | O(1) | Меняем ссылки head/prev |
| Добавление в конец | O(1) | O(1) | Если храним tail |
| Удаление по значению | O(n) | O(n) | Поиск элемента |
| Поиск по значению | O(n) | O(n) | Проход по узлам |
| Чтение всех элементов | O(n) | O(n) | Проход вперёд или назад |

Стек

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Средняя сложность** | **Худшая сложность** | **Комментарий** |
| Push (добавление) | O(1) | O(1) | Добавление в вершину |
| Pop (удаление) | O(1) | O(1) | Удаление вершины |
| Peek (чтение вершины) | O(1) | O(1) | Прямой доступ |
| Чтение всех элементов | O(n) | O(n) | Последовательный обход |

Очередь

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Средняя сложность** | **Худшая сложность** | **Комментарий** |
| Push / Enqueue | O(1) | O(1) | Добавление в хвост |
| Pop / Dequeue | O(1) | O(1) | Удаление из головы |
| Peek (чтение головы) | O(1) | O(1) | Прямой доступ |

Бинарное дерево поиска (BST)

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Средняя сложность** | **Худшая сложность** | **Комментарий** |
| Добавление | O(log n) | O(n) | В несбалансированном виде — вырожденное дерево |
| Поиск | O(log n) | O(n) | Зависит от высоты дерева |
| Удаление | O(log n) | O(n) | Требует поиска и перестройки |
| Обход (чтение) | O(n) | O(n) | Все узлы |

AVL-дерево

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **Средняя сложность** | **Худшая сложность** | **Комментарий** |
| Добавление | O(log n) | O(log n) | Балансировка поддерживает логарифмическую высоту |
| Поиск | O(log n) | O(log n) | Сбалансированная глубина |
| Удаление | O(log n) | O(log n) | После удаления — балансировка |
| Обход (чтение) | O(n) | O(n) | Все узлы |

# Недостатки и достоинства структур и их применение.

**Массив**  
Достоинства:  
– Быстрый доступ к элементам по индексу (O(1)).  
– Простая реализация и эффективное использование памяти.  
– Хорошо подходит для хранения фиксированного количества данных.  
Недостатки:  
– Фиксированный размер (если не динамический массив).  
– Вставка и удаление требуют сдвига элементов (O(n)).  
– Неэффективное использование памяти при частых изменениях размера.  
Применение:  
– Хранение и обработка числовых данных, таблиц, изображений, матриц.  
– Используется в качестве основы для реализации других структур (стек, очередь, хэш-таблицы).

**Односвязный список**  
Достоинства:  
– Динамическое выделение памяти (размер не ограничен заранее).  
– Быстрое добавление и удаление элементов в начале списка.  
– Простая реализация.  
Недостатки:  
– Нет прямого доступа к элементу по индексу (только последовательный поиск).  
– Увеличенные накладные расходы памяти из-за хранения ссылок.  
– Медленный поиск и обход (O(n)).  
Применение:  
– Используется, когда часто добавляются или удаляются элементы.  
– Реализация очередей, стеков, кэш-структур и других динамических контейнеров.

**Двусвязный список**  
Достоинства:  
– Доступ к элементам как вперёд, так и назад.  
– Быстрая вставка и удаление в любом месте при известном узле.  
– Удобен для реализации структур с частым удалением и вставкой.  
Недостатки:  
– Дополнительная память на хранение двух ссылок (prev и next).  
– Более сложная реализация, чем у односвязного списка.  
– Медленный поиск элемента (O(n)).  
Применение:  
– Реализация двусторонних очередей (deque).  
– Используется в браузерах (история «назад/вперёд»), списках воспроизведения, LRU-кэше.

**Стек**  
Достоинства:  
– Простая структура с быстрыми операциями добавления и удаления (O(1)).  
– Естественная реализация принципа LIFO (последний вошёл — первый вышел).  
– Использует минимальные накладные расходы памяти.  
Недостатки:  
– Ограничен доступ только к верхнему элементу.  
– Невозможно произвольное удаление или поиск элементов.  
Применение:  
– Реализация рекурсии (стек вызовов).  
– Проверка правильности скобок, обратная польская запись, отмена действий в программах.

**Очередь**  
Достоинства:  
– Простая реализация принципа FIFO (первый вошёл — первый вышел).  
– Быстрые операции добавления и удаления (O(1)).  
– Эффективна при потоковой обработке данных.  
Недостатки:  
– Нет произвольного доступа к элементам.  
– Возможны накладные расходы при реализации через массив без кольцевой структуры.  
Применение:  
– Планирование задач, очереди печати, обработка сетевых запросов.  
– Реализация буферов данных, например, в очередях сообщений или потоках.

**AVL-дерево**  
Достоинства:  
– Гарантированная сбалансированность, глубина всегда O(log n).  
– Быстрый поиск, вставка и удаление.  
– Эффективно при большом количестве операций поиска.  
Недостатки:  
– Сложная реализация балансировки.  
– Дополнительные затраты на пересчёт высоты и повороты при изменении.  
Применение:  
– Реализация баз данных, индексов и словарей, где требуется частый поиск.  
– Используется в системах, где важна скорость поиска, а вставки редки.

# Контрольный пример работы программы

Использование команд для своей базы данных(Рисунок 1).

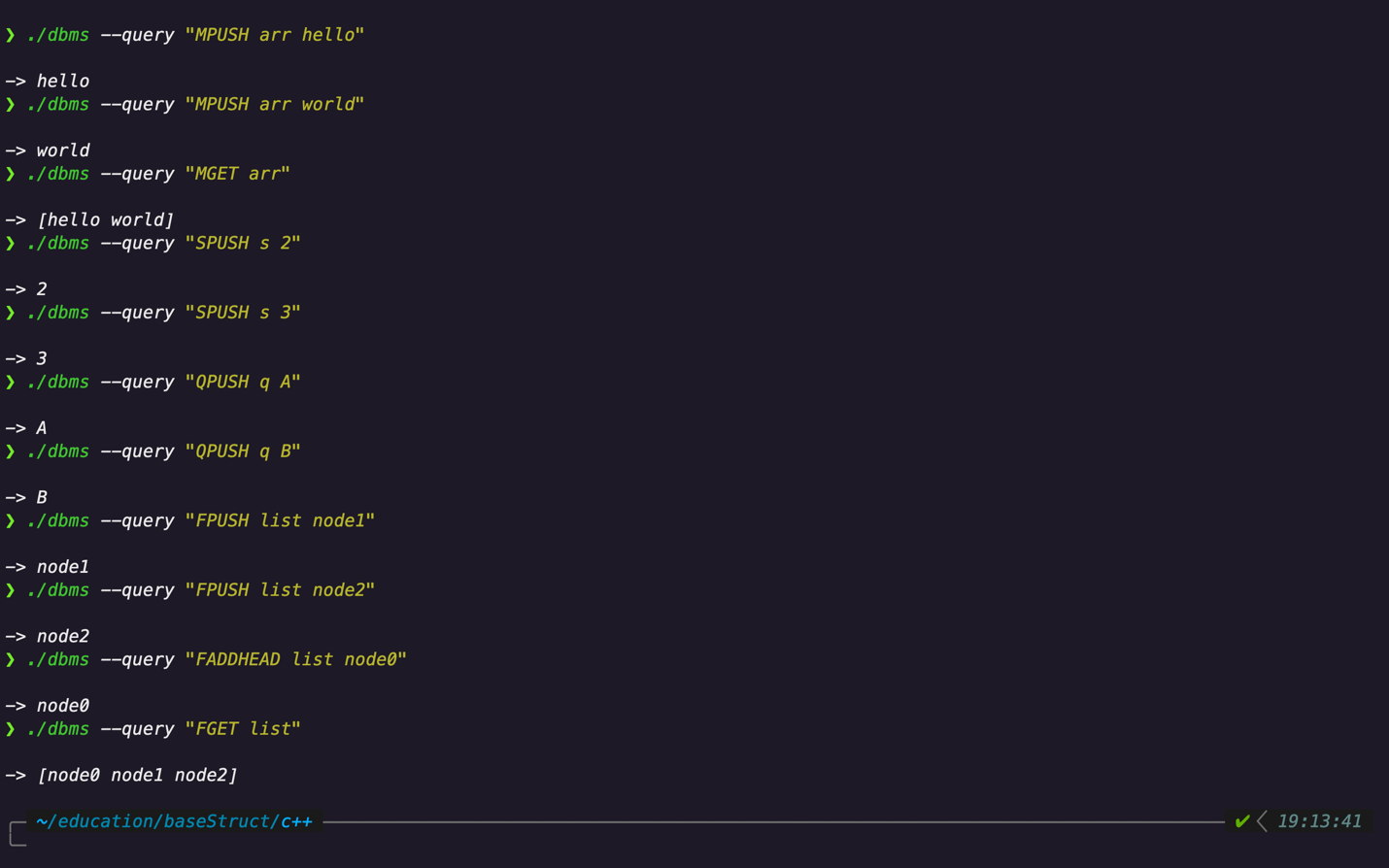


Рисунок 1 – команды

Результат работы программы (Рисунок 2).

  
Рисунок 2 – результат работы программы

# Вывод

В ходе работы были реализованы и протестированы базовые структуры данных: массив, стек, очередь и односвязный список. Проверка операций добавления, удаления и чтения показала корректное выполнение команд и ожидаемое поведение каждой структуры.

**Ссылка на github**: https://github.com/Narotan