МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ

образовательное учреждение

высшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации



**ЛАБОРАТОРНАЯ РАБОТА №1**

**«**Абстрактные структуры данных**»**

**по дисциплине: «*Программирование*»**

|  |  |
| --- | --- |
| Выполнил:  Студент гр. «АБ-221», «АВТФ»  Стеклов Данил Евгеньевич  «16» октября 2023г | Проверил:  *Ассистент ЗИ*  *Исаев Глеб Андреевич*  «\_\_\_» \_\_\_\_\_\_ 2023г |

Новосибирск 2023

**Цели и задачи работы:** изучение алгоритмов формирования и

работы с абстрактными структурами данных.

**Задание к работе:**

Самостоятельно решить задачи в соответствии с индивидуальным вариантом.

**Вариант 21:**

**Задание на языке C:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

*//-------------------------ТИПЫ ДАННЫХ------------------------*

**typedef** **struct** Array {

int capacity;

int size;

int\* data;

}Array;

**typedef** **struct** Node {

int data;

**struct** Node\* next;

}Node;

**typedef** **struct** Shackles {

int data;

**struct** Shackles\* next;

**struct** Shackles\* prev;

}Shackles;

**typedef** **struct** Stack {

int data;

**struct** Stack\* next;

}Stack;

**typedef** **struct** Queue {

int data;

**struct** Queue\* next;

}Queue;

**typedef** **struct** HashTable {

char key[15];

char value[30];

char is\_exist;

} HashTable;

**typedef** **struct** TreeNode {

int data;

**struct** TreeNode\* left;

**struct** TreeNode\* right;

} TreeNode;

**typedef** **struct** CBT {

TreeNode\* root;

int size;

int capacity;

TreeNode\*\* nodes;

} CBT;

*//--------------------------ПРОТОТИПЫ-------------------------*

void InitArray(Array\* arr);

void PrintArray(Array\* arr);

void GetArray(Array\* arr, int i);

void AddToArrayBack(Array\* arr, int value);

void AddToArrayFront(Array\* arr, int value);

void RemoveFromArrayBack(Array\* arr);

void RemoveFromArrayFront(Array\* arr);

void ChangeValueByIndex(Array\* arr, int index, int value);

void FreeArray(Array\* arr);

void ExampleArray();

int CountSingly(Node\* list\_copy);

void PrintListSingly(Node\* list\_copy);

void PushToHeadSingly(Node\*\* list, int data);

void PushToTailSingly(Node\*\* list, int data);

int PopFromHeadSingly(Node\*\* list);

int PopFromTailSingly(Node\* list\_copy);

void PopValueSingly(Node\* list\_copy, int value);

int EditByPositionSingly(Node\* list\_copy, int position, int data);

void FreeSinglyList(Node\*\* list);

void ExampleSinglyList();

int CountDoubly(Shackles\* head);

void PrintListDoubly(Shackles\* head);

void PushToHeadDoubly(Shackles\*\* head, Shackles\*\* tail, int data);

void PushToTailDoubly(Shackles\*\* head, Shackles\*\* tail, int data);

int PopFromHeadDoubly(Shackles\*\* head);

int PopFromTailDoubly(Shackles\*\* tail);

int EditByPositionDoubly(Shackles\* head, int position, int data);

void FreeDoulbyList(Shackles\*\* head, Shackles\*\* tail);

void ExampleDoublyList();

void PushStack(Stack\*\* head, int data);

int PopStack(Stack\*\* head);

void FreeStack(Stack\*\* head);

void ExampleStack();

void PushQueue(Queue\*\* head, Queue\*\* tail, int data);

int PopQueue(Queue\*\* head);

void FreeQueue(Queue\*\* head);

void ExampleQueue();

void HashSet(HashTable\* row, char\* key, char\* value);

void HashDelete(HashTable\* row, char\* key);

void HashGet(HashTable\* row, char\* key);

int FuncHash(char\* key);

int FuncHashSec(char ch);

void ExampleHash();

TreeNode\* createTreeNode(int value);

TreeNode\* insert(TreeNode\* root, int value);

TreeNode\* search(TreeNode\* root, int value);

void freeTree(TreeNode\* root);

bool isCompleteBinaryTree(TreeNode\* root);

TreeNode\* findMinValueNode(TreeNode\* node);

TreeNode\* deleteLastNodeFromCBT(TreeNode\* root, int valueToDelete);

void printBinaryTree(TreeNode\* root, int level);

void ExampleTree();

*//----------------------------МЭЙН----------------------------*

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

setlocale(0, "Ru");

*//ExampleArray();*

*//ExampleSinglyList();*

*//ExampleDoublyList();*

*//ExampleStack();*

*//ExampleQueue();*

*//ExampleHash();*

ExampleTree();

exit(EXIT\_SUCCESS);

}

*//---------------------------МАССИВ---------------------------*

void InitArray(Array\* arr) {

arr->capacity = 0;

arr->size = 0;

arr->data = NULL;

}

void PrintArray(Array\* arr) {

**for** (int i = 0; i < arr->size; ++i) {

printf("%d ", arr->data[i]);

}

printf("**\n**");

}

void GetArray(Array\* arr, int i) {

printf("%d**\n**", arr->data[i]);

}

void AddToArrayBack(Array\* arr, int value) {

**if** (arr->size == arr->capacity) {

arr->capacity = (arr->capacity == 0) ? 2 : arr->capacity \* 2;

int\* new\_data = (int\*)realloc(arr->data, arr->capacity \* **sizeof**(int));

**if** (new\_data == NULL) perror("Ошибка перезаписи");

arr->data = new\_data;

}

arr->data[arr->size++] = value;

**return**;

}

void AddToArrayFront(Array\* arr, int value) {

**if** (arr->size == arr->capacity) {

arr->capacity = (arr->capacity == 0) ? 2 : arr->capacity \* 2;

int\* new\_data = (int\*)realloc(arr->data, arr->capacity \* 4);

**if** (new\_data == NULL) perror("Ошибка перезаписи");

arr->data = new\_data;

}

**for** (int i = arr->size; i > 0; --i) {

arr->data[i] = arr->data[i - 1];

}

arr->data[0] = value;

++arr->size;

**return**;

}

void RemoveFromArrayBack(Array\* arr) {

arr->data[--arr->size] = 0;

}

void RemoveFromArrayFront(Array\* arr) {

**for** (int i = 0; i < arr->size - 1; ++i) {

arr->data[i] = arr->data[i + 1];

}

--arr->size;

}

void ChangeValueByIndex(Array\* arr, int index, int value) {

**if** ((index >= 0) && (index <= arr->size)) arr->data[index] = value;

**else** printf("Введен не правильный индекс, всего ячеек - %d**\n**", arr->capacity);

**return**;

}

void FreeArray(Array\* arr) {

arr->size = 0;

arr->capacity = 0;

free(arr->data);

}

void ExampleArray() {

Array arr;

InitArray(&arr);

AddToArrayBack(&arr, 12);

AddToArrayBack(&arr, 22);

AddToArrayFront(&arr, 11);

AddToArrayBack(&arr, 32);

AddToArrayFront(&arr, 21);

PrintArray(&arr);

RemoveFromArrayBack(&arr);

RemoveFromArrayFront(&arr);

ChangeValueByIndex(&arr, 1, 666);

PrintArray(&arr);

FreeArray(&arr);

}

*//----------------------ОДНОСВЯЗНЫЙ СПИСОК----------------------*

int CountSingly(Node\* list\_copy) {

int x = 0;

**if** (list\_copy == NULL) **return** 0;

**for** (; list\_copy != NULL; list\_copy = list\_copy->next, ++x);

**return** x;

}

void PrintListSingly(Node\* list\_copy) {

**if** (list\_copy == NULL) {

printf("Лист пуст**\n**");

**return**;

}

**while** (list\_copy != NULL) {

printf("%d ", list\_copy->data);

list\_copy = list\_copy->next;

}

printf("**\n**");

}

void PushToHeadSingly(Node\*\* list, int data) {

Node\* tmp = (Node\*)malloc(**sizeof**(Node));

tmp->data = data;

tmp->next = \*list;

\*list = tmp;

}

void PushToTailSingly(Node\*\* list, int data) {

Node\* tmp = (Node\*)malloc(**sizeof**(Node));

Node\* list\_copy = \*list;

tmp->data = data;

tmp->next = NULL;

**if** (\*list == NULL) {

\*list = tmp;

}

**else** {

**while** ((list\_copy->next) != NULL) list\_copy = list\_copy->next;

list\_copy->next = tmp;

}

}

int PopFromHeadSingly(Node\*\* list) {

**if** (\*list == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

Node\* tmp = \*list;

int res = tmp->data;

\*list = tmp->next;

free(tmp);

**return** res;

}

int PopFromTailSingly(Node\* list\_copy) {

**if** (list\_copy == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

Node\* tmp = list\_copy;

**while** (list\_copy->next != NULL) {

tmp = list\_copy;

list\_copy = list\_copy->next;

}

int res = list\_copy->data;

tmp->next = NULL;

free(list\_copy);

**return** res;

}

void PopValueSingly(Node\* list\_copy, int value) {

**if** (list\_copy == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

Node\* tmp = list\_copy;

**while** (list\_copy->data != value) {

tmp = list\_copy;

list\_copy = list\_copy->next;

**if** (list\_copy->next == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return**;

}

}

tmp->next = list\_copy->next;

free(list\_copy);

}

int EditByPositionSingly(Node\* list\_copy, int position, int data) {

**if** (list\_copy == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

**if** (position < 1 || position > CountSingly(list\_copy)) {

printf("Несуществующая позиция элемента**\n**");

**return** NULL;

}

**for** (int i = 1; i < position; i++, list\_copy = (list\_copy)->next);

int res = list\_copy->data;

list\_copy->data = data;

**return** res;

}

void FreeSinglyList(Node\*\* list) {

**while** (\*list != NULL) {

Node\* next = (\*list)->next;

free(\*list);

\*list = next;

}

}

void ExampleSinglyList() {

Node\* list = NULL;

PushToHeadSingly(&list, 1);

PushToHeadSingly(&list, 2);

PushToTailSingly(&list, 3);

PushToTailSingly(&list, 4);

PrintListSingly(list);

printf("Удаленный элемент = %d**\n**", PopFromHeadSingly(&list));

printf("Удаленный элемент = %d**\n**", PopFromTailSingly(list));

EditByPositionSingly(list, 2, 666);

PrintListSingly(list);

FreeSinglyList(&list);

}

*//----------------------ДВУХСВЯЗНЫЙ СПИСОК----------------------*

int CountDoubly(Shackles\* head) {

int x = 0;

**for** (; head != NULL; head = head->next, ++x);

**return** x;

}

void PrintListDoubly(Shackles\* head) {

**if** (head == NULL) {

printf("Лист пуст**\n**");

**return**;

}

**while** (head != NULL) {

printf("%d ", head->data);

head = head->next;

}

printf("**\n**");

}

void PushToHeadDoubly(Shackles\*\* head, Shackles\*\* tail, int data) {

Shackles\* tmp = (Shackles\*)malloc(**sizeof**(Shackles));

tmp->data = data;

tmp->next = \*head;

tmp->prev = NULL;

**if** ((\*tail) == NULL) {

\*tail = tmp;

}

**else** (\*head)->prev = tmp;

\*head = tmp;

}

void PushToTailDoubly(Shackles\*\* head, Shackles\*\* tail, int data) {

Shackles\* tmp = (Shackles\*)malloc(**sizeof**(Shackles));

tmp->data = data;

tmp->next = NULL;

tmp->prev = \*tail;

**if** ((\*head) == NULL) {

\*head = tmp;

}

**else** (\*tail)->next = tmp;

\*tail = tmp;

}

int PopFromHeadDoubly(Shackles\*\* head) {

**if** (\*head == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

Shackles\* tmp = \*head;

int res = tmp->data;

\*head = tmp->next;

(\*head)->prev = NULL;

free(tmp);

**return** res;

}

int PopFromTailDoubly(Shackles\*\* tail) {

**if** (\*tail == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

Shackles\* tmp = \*tail;

int res = tmp->data;

\*tail = tmp->prev;

(\*tail)->next = NULL;

free(tmp);

**return** res;

}

void PopValueDoubly(Shackles\* head\_copy, Shackles\* tail\_copy, int data) {

**if** (head\_copy == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

**while** (head\_copy->data != data) {

head\_copy = head\_copy->next;

**if** (head\_copy->next == NULL) {

printf("Элемент не найден.**\n**");

**return**;

}

}

head\_copy->prev->next = head\_copy->next;

head\_copy->next->prev = head\_copy->prev;

free(head\_copy);

}

int EditByPositionDoubly(Shackles\* head, int position, int data) {

**if** (head == NULL) {

printf("Элемент не найден. Лист пуст.**\n**");

**return** NULL;

}

**if** (position < 1 || position > CountDoubly(head)) {

printf("Несуществующая позиция элемента**\n**");

**return** NULL;

}

**for** (int i = 1; i < position; i++, head = (head)->next);

int res = head->data;

head->data = data;

**return** res;

}

void FreeDoulbyList(Shackles\*\* head, Shackles\*\* tail) {

**while** (\*head != NULL) {

Shackles\* next = (\*head)->next;

free(\*head);

\*head = next;

}

\*tail = NULL;

}

void ExampleDoublyList() {

Shackles\* head = NULL;

Shackles\* tail = NULL;

PushToHeadDoubly(&head, &tail, 1);

PushToHeadDoubly(&head, &tail, 2);

PushToTailDoubly(&head, &tail, 3);

PushToTailDoubly(&head, &tail, 4);

PrintListDoubly(head);

printf("Удаленный элемент = %d**\n**", PopFromHeadDoubly(&head));

printf("Удаленный элемент = %d**\n**", PopFromTailDoubly(&tail));

EditByPositionDoubly(head, 2, 666);

PrintListDoubly(head);

FreeDoulbyList(&head, &tail);

}

*//----------------------------СТЭК----------------------------*

void PushStack(Stack\*\* head, int data) {

Stack\* tmp = (Stack\*)malloc(**sizeof**(Stack));

tmp->data = data;

tmp->next = (\*head);

\*head = tmp;

}

int PopStack(Stack\*\* head) {

Stack\* tmp = (\*head);

(\*head) = (\*head)->next;

int res = tmp->data;

free(tmp);

**return** res;

}

void FreeStack(Stack\*\* head) {

**while** (\*head != NULL) {

Stack\* next = (\*head)->next;

free(\*head);

\*head = next;

}

}

void ExampleStack() {

Stack\* head = NULL;

PushStack(&head, 1);

PushStack(&head, 2);

PushStack(&head, 3);

printf("Удаленный элемент = %d**\n**", PopStack(&head));

printf("Удаленный элемент = %d**\n**", PopStack(&head));

FreeStack(&head);

}

*//--------------------------ОЧЕРЕДЬ--------------------------*

void PushQueue(Queue\*\* head, Queue\*\* tail, int data) {

Queue\* tmp = (Queue\*)malloc(**sizeof**(Queue));

tmp->data = data;

tmp->next = NULL;

**if** ((\*head) == NULL) {

\*head = tmp;

}

**else** (\*tail)->next = tmp;

\*tail = tmp;

}

int PopQueue(Queue\*\* head) {

Queue\* tmp = (\*head);

(\*head) = (\*head)->next;

int res = tmp->data;

free(tmp);

**return** res;

}

void FreeQueue(Queue\*\* head) {

**while** (\*head != NULL) {

Queue\* next = (\*head)->next;

free(\*head);

\*head = next;

}

}

void ExampleQueue() {

Queue\* head = NULL;

Queue\* tail = NULL;

PushQueue(&head, &tail, 1);

PushQueue(&head, &tail, 2);

PushQueue(&head, &tail, 3);

printf("Удаленный элемент = %d**\n**", PopQueue(&head));

printf("Удаленный элемент = %d**\n**", PopQueue(&head));

FreeQueue(&head);

}

*//-------------------------ХЕШ ТАБЛИЦА-------------------------*

void HashSet(HashTable\* row, char\* key, char\* value) {

int index = FuncHash(key);

char is\_coincidence = '0';

char is\_del = '0';

int index\_del;

**while** (row[index].value[0] != '\0') {

**if** (row[index].is\_exist == '0') {

**if** (is\_del == '0') {

index\_del = index;

is\_del = '1';

}

**else** **break**;

}

**if** (strcmp(row[index].key, key) == 0) {

is\_coincidence = '1';

**break**;

}

index = (index + FuncHashSec(key[0])) % 100;

}

**if** (is\_coincidence == '1') {

strcpy\_s(row[index].key, 15, key);

strcpy\_s(row[index].value, 30, value);

row[index].is\_exist = '1';

}

**else** **if** (is\_del == '1') {

strcpy\_s(row[index\_del].key, 15, key);

strcpy\_s(row[index\_del].value, 30, value);

row[index\_del].is\_exist = '1';

}

**else** {

strcpy\_s(row[index].key, 15, key);

strcpy\_s(row[index].value, 30, value);

row[index].is\_exist = '1';

}

}

void HashDelete(HashTable\* row, char\* key) {

int index = FuncHash(key);

**while** (row[index].value[0] != '\0') {

**if** (strcmp(row[index].key, key) == 0) {

row[index].is\_exist = '0';

**break**;

}

index = (index + FuncHashSec(key[0])) % 100;

}

}

void HashGet(HashTable\* row, char\* key) {

int index = FuncHash(key);

char is\_coincidence = '0';

**while** (row[index].value[0] != '\0') {

**if** ((strcmp(row[index].key, key) == 0) && (row[index].is\_exist == '1')) {

is\_coincidence = '1';

**break**;

}

index = (index + FuncHashSec(key[0])) % 100;

}

**if** (is\_coincidence == '1') printf("%s**\n**", row[index].value);

**else** printf("Не найдено значение для этого ключа = %s**\n**", key);

}

int FuncHash(char\* key) {

int sum = 0;

**for** (int i = 0; key[i]; ++i)

sum += key[i];

**return** sum % 256;

}

int FuncHashSec(char ch) {

int sum = ch \* 37;

**return** sum % 256;

}

void ExampleHash() {

HashTable\* row = (HashTable\*)calloc(256, **sizeof**(HashTable));

HashSet(row, "123", "Danil");

HashSet(row, "1342", "Sasha");

HashSet(row, "1124", "Lisa");

HashGet(row, "123");

HashDelete(row, "1342");

HashGet(row, "1342");

HashGet(row, "1124");

free(row);

}

*//--------------------------БИНАРНОЕ ДЕРЕВО--------------------------*

*// Функция для создания нового узла бинарного дерева*

TreeNode\* createTreeNode(int value) {

TreeNode\* newNode = (TreeNode\*)malloc(**sizeof**(TreeNode));

**if** (newNode) {

newNode->data = value;

newNode->left = NULL;

newNode->right = NULL;

}

**return** newNode;

}

*// Функция для вставки нового узла в бинарное дерево*

TreeNode\* insert(TreeNode\* root, int value) {

**if** (root == NULL) {

**return** createTreeNode(value);

}

**if** (value < root->data) {

root->left = insert(root->left, value);

}

**else** **if** (value > root->data) {

root->right = insert(root->right, value);

}

**return** root;

}

*// Функция для поиска узла по значению в бинарном дереве*

TreeNode\* search(TreeNode\* root, int value) {

**if** (root == NULL || root->data == value) {

**return** root;

}

**if** (value < root->data) {

**return** search(root->left, value);

}

**return** search(root->right, value);

}

*// Функция для освобождения памяти, занятой узлами бинарного дерева*

void freeTree(TreeNode\* root) {

**if** (root) {

freeTree(root->left);

freeTree(root->right);

free(root);

}

}

*// Функция для проверки, является ли дерево полным бинарным деревом*

bool isCompleteBinaryTree(TreeNode\* root) {

**if** (root == NULL) {

**return** true; *// Пустое дерево считается полным*

}

TreeNode\* queue[100]; *//размер очереди*

int front = 0, rear = 0;

bool missingNodeFound = false;

queue[rear++] = root;

**while** (front < rear) {

TreeNode\* current = queue[front++];

**if** (current == NULL) {

missingNodeFound = true;

}

**else** {

**if** (missingNodeFound) {

**return** false;

}

queue[rear++] = current->left;

queue[rear++] = current->right;

}

}

**return** true;

}

TreeNode\* findMinValueNode(TreeNode\* node) {

TreeNode\* current = node;

**while** (current && current->left) {

current = current->left;

}

**return** current;

}

*// Функция для удаления последнего узла из полного бинарного дерева*

TreeNode\* deleteLastNodeFromCBT(TreeNode\* root, int valueToDelete) {

**if** (root == NULL) {

**return** root;

}

**if** (valueToDelete < root->data) {

root->left = deleteLastNodeFromCBT(root->left, valueToDelete);

}

**else** **if** (valueToDelete > root->data) {

root->right = deleteLastNodeFromCBT(root->right, valueToDelete);

}

**else** {

**if** (root->left == NULL) {

TreeNode\* temp = root->right;

free(root);

**return** temp;

}

**else** **if** (root->right == NULL) {

TreeNode\* temp = root->left;

free(root);

**return** temp;

}

TreeNode\* temp = findMinValueNode(root->right);

root->data = temp->data;

root->right = deleteLastNodeFromCBT(root->right, temp->data);

}

**return** root;

}

void printBinaryTree(TreeNode\* root, int level) {

**if** (root == NULL) {

**return**;

}

printBinaryTree(root->right, level + 1);

**for** (int i = 0; i < level; i++) {

printf(" ");

}

printf("%d**\n**", root->data);

printBinaryTree(root->left, level + 1);

}

void ExampleTree() {

TreeNode\* root = createTreeNode(5);

root = insert(root, 2);

root = insert(root, 1);

root = insert(root, 6);

root = insert(root, 4);

root = insert(root, 9);

root = insert(root, 5);

root = insert(root, 7);

root = deleteLastNodeFromCBT(root, 9);

printf("**\n**");

printBinaryTree(root, 3);

bool isComplete = isCompleteBinaryTree(root); *//проверка на CBT*

**if** (isComplete) {

printf("it CBT.**\n**");

}

**else** {

printf("Not CBT.**\n**");

}

TreeNode\* result = search(root, 4);

*// Выводим результат*

**if** (result != NULL) {

printf("Node %d find.**\n**", 4);

}

**else** {

printf("Node %d not found.**\n**", 4);

}

freeTree(root);

}

**Пример работы:**

1. Динамический массив

Последовательность действий:

1. Добавить в конец 12

2. Добавить в конец 22

3. Добавить в начало 11

4. Добавить в конец 32

5. Добавить в начало 21

6. Вывести

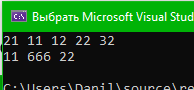
7. Удалить с конца

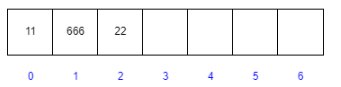
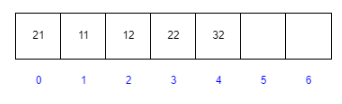
8. Удалить с начала

9. Заменить элемент под 1 индексом на 666

10. Вывести массив

11. Очистить память





2. Односвязный список

Последовательность действий:

1. Добавить в голову списка 1

2. Добавить в голову списка 2

3. Добавить в хвост списка 3

4. Добавить в хвост списка 4

5. Вывести список

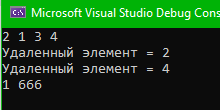
6. Удалить с головы и вывести

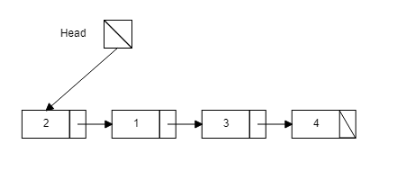
7. Удалить с хвоста и вывести

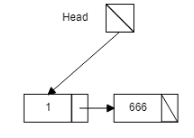
8. Изменить второй элемент на 666

9. Вывести список

10. Отчистить память







2. Двухсвязный список

Последовательность действий:

1. Добавить в голову списка 1

2. Добавить в голову списка 2

3. Добавить в хвост списка 3

4. Добавить в хвост списка 4

5. Вывести список

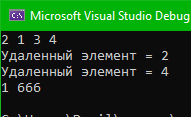
6. Удалить с головы и вывести

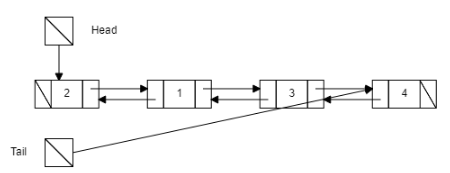
7. Удалить с хвоста и вывести

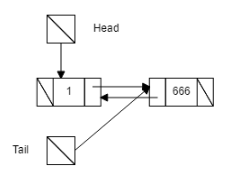
8. Изменить второй элемент на 666

9. Вывести список

10. Отчистить память







4. Стек

Последовательность действий:

1. Добавить 1

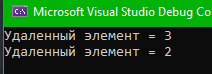
2. Добавить 2

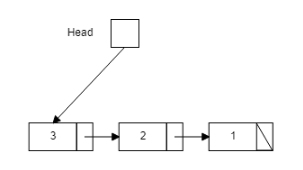
3. Добавить 3

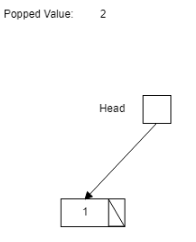
4. Удалить и вывести

5. Удалить и вывести

6. Очистить стек







4. Очередь

Последовательность действий:

1. Добавить 1

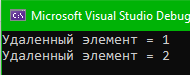
2. Добавить 2

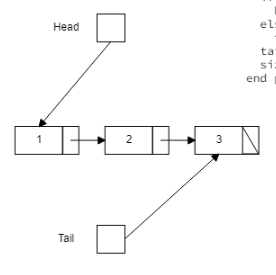
3. Добавить 3

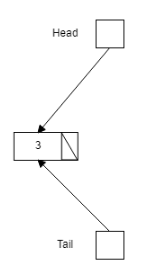
4. Удалить и вывести

5. Удалить и вывести

6. Очистить очередь







5. Хеш-таблица

Последовательность действий:

1. Добавить 123 по ключу Danil

2. Добавить 1342 по ключу Sasha

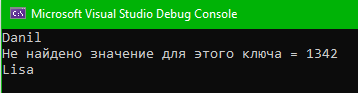
3. Добавить 1124 по ключу Lisa

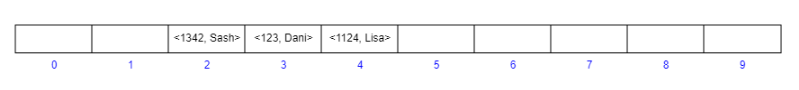
4. Вернуть значение по ключу 123

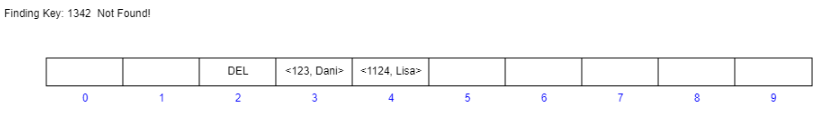
5. Удалить значение по ключу 1342

6. Вернуть значение по ключу 1342

7. Вернуть значение по ключу 1124







6. Complete Binary Tree

Последовательность действий:

1. Добавить элемент 2

2. Добавить элемент 1

3. Добавить элемент 6

4. Добавить элемент 4

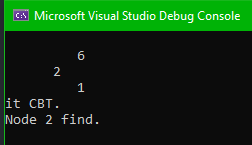
5. Удалить элемент 4

6. Вывести дерево

7. Проверить на CBT

8. Найти 6

9. Очистить память



Отсутствует визуализация CBT на сайте