МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

РГЗ

по дисциплине: Алгоритмы и структуры данных

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили: асс. Солонченко Роман
Евгеньевич

Белгород 2023г.

РГЗ

Common.h

```c
// Содержит общие объявления

#pragma once

typedef void* BaseType;
```

1. Реализовать стек на массиве
   Интерфейс

```c
// Стек как отображение на статический массив

#pragma once

#include <stdbool.h>

// Максимальное количество элементов стека на массиве.
#define ArrayStackBufferSize 100

// Операция выполнена успешно.
#define ArrayStackOk 0
// Стек пуст.
#define ArrayStackEmpty 1
// Стек заполнен.
#define ArrayStackFull 2

// Код ошибки, обновляется каждый раз после выполнения операции над стеком.
extern int ArrayStackError;

#include <stdint.h>

#include <CNuke/Common.h>

typedef struct {
    BaseType Buf[ArrayStackBufferSize];
    size_t End;
} ArrayStack;

// Инициализирует пустой стек и возвращает его.
ArrayStack ArrayStackInit();

// Если стек S не переполнен, добавляет в него элемент E.
void ArrayStackPut(ArrayStack *S, BaseType E);

// Если стек S не пуст, исключает верхний элемент и возвращает его.
BaseType ArrayStackGet(ArrayStack *S);

// Возвращает true, если стек S пуст, иначе - false.
bool ArrayStackIsEmpty(ArrayStack S);
```

```
// Возвращает true, если стек S заполнен, иначе - false.
bool ArrayStackIsFull(ArrayStack S);
```

Реализация

```
#include <CNuke/container/ArrayStack.h>

int ArrayStackError = ArrayStackOk;

ArrayStack ArrayStackInit() {
    ArrayStackError = ArrayStackOk;
    return (ArrayStack) {{0}, 0};
}

void ArrayStackPut(ArrayStack *S, BaseType E) {
    ArrayStackError = ArrayStackOk;
    if (ArrayStackIsFull(*S)) {
        ArrayStackError = ArrayStackFull;
        return;
    }

    S->Buf[S->End++] = E;
}

BaseType ArrayStackGet(ArrayStack *S) {
    ArrayStackError = ArrayStackOk;
    if (ArrayStackIsEmpty(*S)) {
        ArrayStackError = ArrayStackFull;
        return;
    }

    return S->Buf[--S->End];
}

bool ArrayStackIsEmpty(ArrayStack S) {
    return S.End == 0;
}

bool ArrayStackIsFull(ArrayStack S) {
    return S.End == ArrayStackBufferSize;
}
```

2. Реализовать стек на ОЛС
Интерфейс

```
// Стек как отображение на односвязный линейный список

#pragma once

#include <CNuke/list/SinglyLinkedList.h>
```

```c
// Операция выполнена успешно.
#define LinkedStackOk SinglyLinkedListOk
// Стек пуст.
#define LinkedStackEmpty SinglyLinkedListEmpty
// Недостаточно памяти для нового элемента.
#define LinkedStackNotMem SinglyLinkedListNotMem

// Код ошибки, обновляется каждый раз после выполнения операции над стеком.
extern int LinkedStackError;

#include <stdint.h>

#include <CNuke/Common.h>

typedef SinglyLinkedList LinkedStack;

// Инициализирует пустой стек и возвращает его.
LinkedStack LinkedStackInit();

// Добавляет в стек S элемент E, если это возможно.
void LinkedStackPut(LinkedStack *S, BaseType E);

// Если стек S не пуст, исключает элемент и возвращает его.
BaseType LinkedStackGet(LinkedStack *S);

// Возвращает true, если стек S пуст, иначе - false.
bool LinkedStackIsEmpty(LinkedStack S);

// Освобождает память, занятую стеком S
void LinkedStackDone(LinkedStack *S);
```

Реализация

```c
#include <CNuke/container/LinkedStack.h>

int LinkedStackError = LinkedStackOk;

LinkedStack LinkedStackInit() {
    LinkedStack stack = SinglyLinkedListInit();
    LinkedStackError = SinglyLinkedListError;

    return stack;
}

void LinkedStackPut(LinkedStack *S, BaseType E) {
    SinglyLinkedListPut(S, E);
    LinkedStackError = SinglyLinkedListError;
}

BaseType LinkedStackGet(LinkedStack *S) {
    BaseType result = SinglyLinkedListGet(S);
    LinkedStackError = SinglyLinkedListError;
```

```
    return result;
}

bool LinkedStackIsEmpty(LinkedStack S) {
    bool result = SinglyLinkedListIsEmpty(S);
    LinkedStackError = SinglyLinkedListError;

    return result;
}

void LinkedStackDone(LinkedStack *S) {
    SinglyLinkedListDone(S);
    LinkedStackError = SinglyLinkedListError;
}
```

3. Реализовать очередь на массиве
   Интерфейс

```
// Очередь как отображение на статический массив

#pragma once

#include <stdbool.h>
#include <stdint.h>

// Максимальное количество элементов очереди на массиве.
#define ArrayQueueBufferSize 100

// Операция выполнена успешно.
#define ArrayQueueOk 0
// Очередь пуста.
#define ArrayQueueEmpty 1
// Очередь заполнена.
#define ArrayQueueFull 2

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
extern int ArrayQueueError;

#include <CNuke/Common.h>

typedef struct {
    BaseType Buf[ArrayQueueBufferSize];
    size_t Begin;
    size_t End;
    size_t Size;
} ArrayQueue;

// Инициализирует пустую очередь и возвращает её.
ArrayQueue ArrayQueueInit();

// Если очереь Q не переполнена, добавляет наверх в неё элемент E.
void ArrayQueuePut(ArrayQueue *Q, BaseType E);
```

```
// Если очередь Q не пуста, исключает нижний элемент и возвращает его.
BaseType ArrayQueueGet(ArrayQueue *Q);

// Возвращает true, если очередь Q пуста, иначе - false.
bool ArrayQueueIsEmpty(ArrayQueue Q);

// Возвращает true, если очередь Q заполнена, иначе - false.
bool ArrayQueueIsFull(ArrayQueue Q);
```

Реализация

```
#include <CNuke/container/ArrayQueue.h>

int ArrayQueueError = ArrayQueueOk;

ArrayQueue ArrayQueueInit() {
    ArrayQueueError = ArrayQueueOk;
    return (ArrayQueue) {{0}, 0, 0, 0};
}

void ArrayQueuePut(ArrayQueue *Q, BaseType E) {
    ArrayQueueError = ArrayQueueOk;
    if (ArrayQueueIsFull(*Q)) {
        ArrayQueueError = ArrayQueueFull;
        return;
    }

    Q->Buf[Q->End] = E;
    Q->End = (Q->End + 1) % ArrayQueueBufferSize;
    Q->Size++;
}

BaseType ArrayQueueGet(ArrayQueue *Q) {
    ArrayQueueError = ArrayQueueOk;
    if (ArrayQueueIsEmpty(*Q)) {
        ArrayQueueError = ArrayQueueEmpty;
        return 0;
    }

    BaseType result = Q->Buf[Q->Begin];
    Q->Begin = (Q->Begin + 1) % ArrayQueueBufferSize;
    Q->Size--;

    return result;
}

bool ArrayQueueIsEmpty(ArrayQueue Q) {
    return Q.Size == 0;
}

bool ArrayQueueIsFull(ArrayQueue Q) {
    return Q.Size == ArrayQueueBufferSize;
```

```
}
```

4. Реализовать очередь на ОЛС

Интерфейс

```
// Очередь как отображение на односвязный линейный список

#pragma once

#include <CNuke/list/SinglyLinkedList.h>

// Операция выполнена успешно.
#define LinkedQueueOk SinglyLinkedListOk
// Очередь пуста.
#define LinkedQueueEmpty SinglyLinkedListEmpty
// Недостаточно памяти для нового элемента.
#define LinkedQueueNotMem SinglyLinkedListNotMem

// Код ошибки, обновляется каждый раз после выполнения операции над стеком.
extern int LinkedQueueError;

#include <stdint.h>

#include <CNuke/Common.h>

typedef SinglyLinkedList LinkedQueue;

// Инициализирует пустую очередь и возвращает её.
LinkedQueue LinkedQueueInit();

// Добавляет в очередь Q элемент E, если это возможно.
void LinkedQueuePut(LinkedQueue *Q, BaseType E);

// Если очередь Q не пуста, исключает элемент и возвращает его.
BaseType LinkedQueueGet(LinkedQueue *Q);

// Возвращает true, если очередь Q пуста, иначе - false.
bool LinkedQueueIsEmpty(LinkedQueue Q);

// Освобождает память, занятую очередью Q
void LinkedQueueDone(LinkedQueue *Q);
```

Реализация

```
#include <CNuke/container/LinkedQueue.h>

int LinkedQueueError = LinkedQueueOk;

LinkedQueue LinkedQueueInit() {
    LinkedQueue stack = SinglyLinkedListInit();
    LinkedQueueError = SinglyLinkedListError;
```

```
    return stack;
}

void LinkedQueuePut(LinkedQueue *Q, BaseType E) {
    SinglyLinkedListEndPtr(Q);
    SinglyLinkedListPut(Q, E);
    LinkedQueueError = SinglyLinkedListError;
}

BaseType LinkedQueueGet(LinkedQueue *Q) {
    SinglyLinkedListBeginPtr(Q);
    BaseType result = SinglyLinkedListGet(Q);
    LinkedQueueError = SinglyLinkedListError;

    return result;
}

bool LinkedQueueIsEmpty(LinkedQueue Q) {
    bool result = SinglyLinkedListIsEmpty(Q);
    LinkedQueueError = SinglyLinkedListError;

    return result;
}

void LinkedQueueDone(LinkedQueue *Q) {
    SinglyLinkedListDone(Q);
    LinkedQueueError = SinglyLinkedListError;
}
```

5. Реализовать ОЛС
Интерфейс

```
// Структура данных односвязный линейный список

#pragma once

#include <stdbool.h>

// Операция выполнена успешно.
#define SinglyLinkedListOk 0
// Лист пуст.
#define SinglyLinkedListEmpty 1
// Память для нового элемента выделить не удалось.
#define SinglyLinkedListNotMem 2
// Рабочий указатель стоит на последнем элементе списка.
#define SinglyLinkedListEnd 3

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
extern int SinglyLinkedListError;

#include <CNuke/Common.h>
```

```c
typedef struct SinglyLinkedListElement {
    BaseType Value;
    struct SinglyLinkedListElement* Next;
} SinglyLinkedListElement;

typedef SinglyLinkedListElement* SinglyLinkedListElementPtr;

typedef struct {
    SinglyLinkedListElementPtr Begin;
    SinglyLinkedListElementPtr Ptr;
} SinglyLinkedList;

// Инициализирует пустой односвязный линейный список и возвращает его.
SinglyLinkedList SinglyLinkedListInit();

// Включает в односвязный линейный список элемент после рабочего указателя.
void SinglyLinkedListPut(SinglyLinkedList *L, BaseType E);

// Возвращает значение элемента идущего после рабочего указателя и удаляет его из списка L.
BaseType SinglyLinkedListGet(SinglyLinkedList *L);

// Передвигает рабочий указатель на следующий элемент в списке L.
void SinglyLinkedListMovePtr(SinglyLinkedList *L);

// Возвращает true, если односвязный линейный список L пуст, иначе - false.
bool SinglyLinkedListIsEmpty(SinglyLinkedList L);

// Освобождает память, занятую односвязным линейным списком L.
void SinglyLinkedListDone(SinglyLinkedList *L);

// Перемещает рабочий указатель L в начало
void SinglyLinkedListBeginPtr(SinglyLinkedList *L);

// Перемещает рабочий указатель L в конец.
void SinglyLinkedListEndPtr(SinglyLinkedList *L);
```

Реализация

```c
#include <CNuke/list/SinglyLinkedList.h>

#include <malloc.h>

int SinglyLinkedListError = SinglyLinkedListOk;

SinglyLinkedList SinglyLinkedListInit() {
    SinglyLinkedListError = SinglyLinkedListOk;
    SinglyLinkedListElementPtr newElement = (SinglyLinkedListElementPtr) malloc(sizeof(SinglyLinkedListElement));
    if (newElement == NULL) {
        SinglyLinkedListError = SinglyLinkedListNotMem;
        return;
    }
    newElement->Next = NULL;
```

```
    return (SinglyLinkedList) {newElement, newElement};
}

void SinglyLinkedListPut(SinglyLinkedList *L, BaseType E) {
    SinglyLinkedListError = SinglyLinkedListOk;
    SinglyLinkedListElementPtr newElement = (SinglyLinkedListElementPtr) malloc(sizeof(SinglyLinkedListElement));
    if (newElement == NULL) {
        SinglyLinkedListError = SinglyLinkedListNotMem;
        return;
    }

    newElement->Value = E;
    newElement->Next = NULL;

    SinglyLinkedListElementPtr currentElement = L->Ptr;
    SinglyLinkedListElementPtr nextElement = currentElement->Next;
    currentElement->Next = newElement;
    newElement->Next = nextElement;
}

BaseType SinglyLinkedListGet(SinglyLinkedList *L) {
    SinglyLinkedListError = SinglyLinkedListOk;
    if (SinglyLinkedListIsEmpty(*L)) {
        SinglyLinkedListError = SinglyLinkedListEmpty;
        return;
    }

    if (L->Ptr->Next == NULL) {
        SinglyLinkedListError = SinglyLinkedListEnd;
        return;
    }

    SinglyLinkedListElementPtr currentElement = L->Ptr;
    BaseType result = currentElement->Next->Value;

    SinglyLinkedListElementPtr nextNextElement = currentElement->Next->Next;
    free(currentElement->Next);
    currentElement->Next = nextNextElement;

    return result;
}

void SinglyLinkedListMovePtr(SinglyLinkedList *L) {
    SinglyLinkedListError = SinglyLinkedListOk;
    if (L->Ptr->Next == NULL) {
        SinglyLinkedListError = SinglyLinkedListEnd;
        return;
    }

    L->Ptr = L->Ptr->Next;
}

bool SinglyLinkedListIsEmpty(SinglyLinkedList L) {
```

```
    SinglyLinkedListError = SinglyLinkedListOk;
    return L.Begin->Next == NULL;
}

void SinglyLinkedListFreeElement(SinglyLinkedListElementPtr element) {
    if (element == NULL)
        return;

    SinglyLinkedListFreeElement(element->Next);
    free(element);
}

void SinglyLinkedListDone(SinglyLinkedList *L) {
    SinglyLinkedListError = SinglyLinkedListOk;
    SinglyLinkedListFreeElement(L->Begin);
    *L = (SinglyLinkedList){NULL, NULL};
}

void SinglyLinkedListBeginPtr(SinglyLinkedList *L) {
    SinglyLinkedListError = SinglyLinkedListOk;
    L->Ptr = L->Begin;
}

void SinglyLinkedListEndPtr(SinglyLinkedList *L) {
    SinglyLinkedListError = SinglyLinkedListOk;
    while (L->Ptr->Next != NULL) {
        L->Ptr = L->Ptr->Next;
    }
}
```

6. Реализовать ДЛС
Интерфейс

```
// Структура данных двусвязный линейный список

#pragma once

#include <stdbool.h>

// Операция выполнена успешно.
#define DoublyLinkedListOk 0
// Лист пуст.
#define DoublyLinkedListEmpty 1
// Не удалось выделить память для нового элемента.
#define DoublyLinkedListNotMem 2
// Рабочий указатель находится в конце списка
#define DoublyLinkedListEnd 3
// Рабочий указатель находится в начале списка
#define DoublyLinkedListBegin 4

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
extern int DoublyLinkedListError;
```

```c
#include <CNuke/Common.h>

typedef struct DoublyLinkedListElement {
    BaseType Value;
    struct DoublyLinkedListElement* Next;
    struct DoublyLinkedListElement* Prev;
} DoublyLinkedListElement;

typedef DoublyLinkedListElement* DoublyLinkedListElementPtr;

typedef struct {
    DoublyLinkedListElementPtr Begin;
    DoublyLinkedListElementPtr End;
    DoublyLinkedListElementPtr Ptr;
} DoublyLinkedList;

// Инициализирует пустой двусвязный линейный список и возвращает его.
DoublyLinkedList DoublyLinkedListInit();

// Включает в двусвязный линейный список L элемент E до рабочего указателя.
void DoublyLinkedListPutBefore(DoublyLinkedList* L, BaseType E);

// Включает в двусвязный линейный список L элемент E после рабочего указателя.
void DoublyLinkedListPutAfter(DoublyLinkedList* L, BaseType E);

// Возвращает значение элемента идущего до рабочего указателя и удаляет его из списка L.
BaseType DoublyLinkedListGetBefore(DoublyLinkedList *L);

// Возвращает значение элемента идущего после рабочего указателя и удаляет его из списка L.
BaseType DoublyLinkedListGetAfter(DoublyLinkedList *L);

// Передвигает рабочий указатель на предыдущий элемент в списке L.
void DoublyLinkedListMoveL(DoublyLinkedList *L);

// Передвигает рабочий указатель на следующий элемент в списке L.
void DoublyLinkedListMoveR(DoublyLinkedList *L);

// Освобождает память, занятую двусвязным линейным списком L.
void DoublyLinkedListDone(DoublyLinkedList *L);

// Перемещает рабочий указатель L в начало
void DoublyLinkedListBeginPtr(DoublyLinkedList *L);

// Перемещает рабочий указатель L в конец.
void DoublyLinkedListEndPtr(DoublyLinkedList *L);
```

Реализация

```c
#include <CNuke/list/DoublyLinkedList.h>

#include <stdlib.h>
```

```
int DoublyLinkedListError = DoublyLinkedListOk;

DoublyLinkedList DoublyLinkedListInit() {
    DoublyLinkedListError = DoublyLinkedListOk;

    return (DoublyLinkedList) {NULL, NULL, NULL};
}

void DoublyLinkedListPutBefore(DoublyLinkedList* L, BaseType E) {
    DoublyLinkedListError = DoublyLinkedListOk;
    DoublyLinkedListElementPtr newElement = (DoublyLinkedListElementPtr) malloc(sizeof(DoublyLinkedListElement));
    if (newElement == NULL) {
        DoublyLinkedListError = DoublyLinkedListNotMem;
        return;
    }

    newElement->Value = E;

    if (L->Begin == NULL) {
        L->Begin = newElement;
        L->End = newElement;
        L->Ptr = newElement;
        newElement->Next = NULL;
        newElement->Prev = NULL;

        return;
    }

    DoublyLinkedListElementPtr oldPrev = L->Ptr->Prev;
    L->Ptr->Prev = newElement;
    newElement->Prev = oldPrev;

    if (oldPrev == NULL) {
        L->Begin = newElement;
    }
}

void DoublyLinkedListPutAfter(DoublyLinkedList* L, BaseType E) {
    DoublyLinkedListError = DoublyLinkedListOk;
    DoublyLinkedListElementPtr newElement = (DoublyLinkedListElementPtr) malloc(sizeof(DoublyLinkedListElement));
    if (newElement == NULL) {
        DoublyLinkedListError = DoublyLinkedListNotMem;
        return;
    }

    newElement->Value = E;

    if (L->Begin == NULL) {
        L->Begin = newElement;
        L->End = newElement;
        L->Ptr = newElement;
        newElement->Next = NULL;
        newElement->Prev = NULL;
```

```
        return;
    }

    DoublyLinkedListElementPtr oldNext = L->Ptr->Next;
    L->Ptr->Next = newElement;
    newElement->Next = oldNext;

    if (oldNext == NULL) {
        L->End = newElement;
    }
}

BaseType DoublyLinkedListGetBefore(DoublyLinkedList *L) {
    DoublyLinkedListError = DoublyLinkedListOk;
    if (L->Begin == L->End && L->Begin != NULL) {
        BaseType result = L->Begin->Value;

        free(L->Begin);
        L->Begin = NULL;
        L->End = NULL;
        L->Ptr = NULL;

        return result;
    }

    if (L->Ptr == NULL || L->Ptr->Prev == NULL) {
        DoublyLinkedListError = DoublyLinkedListBegin;
        return;
    }

    DoublyLinkedListElementPtr prevElement = L->Ptr->Prev;
    DoublyLinkedListElementPtr prevPrevElement = prevElement->Prev;

    BaseType result = prevElement->Value;

    free(prevElement);
    L->Ptr->Prev = prevPrevElement;

    if (prevPrevElement == NULL) {
        L->Begin = L->Ptr;
    }

    return result;
}

BaseType DoublyLinkedListGetAfter(DoublyLinkedList *L) {
    DoublyLinkedListError = DoublyLinkedListOk;
    if (L->Begin == L->End && L->Begin != NULL) {
        BaseType result = L->Begin->Value;

        free(L->Begin);
        L->Begin = NULL;
        L->End = NULL;
        L->Ptr = NULL;
```

```
        return result;
    }

    if (L->Ptr == NULL || L->Ptr->Next == NULL) {
        DoublyLinkedListError = DoublyLinkedListEnd;
        return;
    }

    DoublyLinkedListElementPtr nextElement = L->Ptr->Next;
    DoublyLinkedListElementPtr nextNextElement = nextElement->Next;

    BaseType result = nextElement->Value;

    free(nextElement);
    L->Ptr->Next = nextNextElement;

    if (nextNextElement == NULL) {
        L->End = L->Ptr;
    }

    return result;
}

void DoublyLinkedListMoveL(DoublyLinkedList *L) {
    DoublyLinkedListError = DoublyLinkedListOk;
    if (L->Ptr == NULL || L->Ptr->Prev == NULL) {
        DoublyLinkedListError = DoublyLinkedListBegin;
        return;
    }

    L->Ptr = L->Ptr->Prev;
}

void DoublyLinkedListMoveR(DoublyLinkedList *L) {
    DoublyLinkedListError = DoublyLinkedListOk;
    if (L->Ptr == NULL || L->Ptr->Next == NULL) {
        DoublyLinkedListError = DoublyLinkedListEnd;
        return;
    }

    L->Ptr = L->Ptr->Next;
}

void DoublyLinkedListDone(DoublyLinkedList *L) {
    DoublyLinkedListError = DoublyLinkedListOk;
    DoublyLinkedListElementPtr currentElement = L->Begin;
    while (currentElement != NULL) {
        DoublyLinkedListElementPtr next = currentElement->Next;
        free(currentElement);
        currentElement = next;
    }

}
```

```
void DoublyLinkedListBeginPtr(DoublyLinkedList *L) {
    DoublyLinkedListError = DoublyLinkedListOk;
    L->Ptr = L->Begin;
}

void DoublyLinkedListEndPtr(DoublyLinkedList *L) {
    DoublyLinkedListError = DoublyLinkedListOk;
    L->Ptr = L->End;
}
```

7. Реализовать дэк как отображение на ДЛС
Интерфейс

```
// Дек как отображение на двусвязный линейный список

#pragma once

#include <CNuke/list/DoublyLinkedList.h>

// Операция выполнена успешно.
#define LinkedDequeOk DoublyLinkedListOk
// Не удалось выделить память для нового элемента.
#define LinkedDequeNotMem DoublyLinkedListNotMem
// Дек пуст
#define LinkedDequeEmpty DoublyLinkedListEmpty

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
extern int LinkedDequeError;

typedef DoublyLinkedList LinkedDeque;

// Инициализирует пустой дек и возвращает его.
LinkedDeque LinkedDequeInit();

// Освобождает память, занятую деком D.
void LinkedDequeDone(LinkedDeque* D);

// Включает в дек D элемент E в конец списка.
void LinkedDequePutEnd(LinkedDeque* D, BaseType E);

// Включает в дек D элемент E в начало списка.
void LinkedDequePutBegin(LinkedDeque* D, BaseType E);

// Возвращает значение элемента в начале и удаляет его из дека D.
BaseType LinkedDequeGetBegin(LinkedDeque* D);

// Возвращает значение элемента в конце и удаляет его из дека D.
BaseType LinkedDequeGetEnd(LinkedDeque* D);

// Возвращает true, если дек D пуст, иначе - false.
```

```
bool LinkedDequeIsEmpty(LinkedDeque* D);
```

Реализация

```
#include <CNuke/container/LinkedDeque.h>

#include <stdint.h>

int LinkedDequeError = LinkedDequeOk;

LinkedDeque LinkedDequeInit() {
    LinkedDeque result = DoublyLinkedListInit();
    LinkedDequeError = DoublyLinkedListError;

    return result;
}

void LinkedDequeDone(LinkedDeque* D) {
    DoublyLinkedListDone(D);
    LinkedDequeError = DoublyLinkedListError;
}

void LinkedDequePutEnd(LinkedDeque* D, BaseType E) {
    DoublyLinkedListEndPtr(D);
    DoublyLinkedListPutAfter(D, E);
    LinkedDequeError = DoublyLinkedListError;
}

void LinkedDequePutBegin(LinkedDeque* D, BaseType E) {
    DoublyLinkedListBeginPtr(D);
    DoublyLinkedListPutBefore(D, E);
    LinkedDequeError = DoublyLinkedListError;
}

BaseType LinkedDequeGetBegin(LinkedDeque* D) {
    DoublyLinkedListBeginPtr(D);
    DoublyLinkedListMoveR(D);
    BaseType result = DoublyLinkedListGetBefore(D);
    LinkedDequeError = DoublyLinkedListError;

    return result;
}

BaseType LinkedDequeGetEnd(LinkedDeque* D) {
    DoublyLinkedListEndPtr(D);
    DoublyLinkedListMoveL(D);
    BaseType result = DoublyLinkedListGetAfter(D);
    LinkedDequeError = DoublyLinkedListError;

    return result;
}

bool LinkedDequeIsEmpty(LinkedDeque* D) {
```

```
    LinkedDequeError = LinkedDequeOk;
    return D->Begin == NULL;
}
```

8. Реализовать таблицу как отображение на неупорядоченный массив
Интерфейс

```
// Структура данных таблица

#pragma once

#include <stdbool.h>
#include <stdint.h>

// Максимальное количество элементов таблицы.
#define TableBufferSize 100

// Операция выполнена успешно.
#define TableOk 0
// Таблица пуста.
#define TableEmpty 1
// Таблица заполнена.
#define TableFull 2
// Ключ в таблице не найден
#define TableNoKey 3

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
extern int TableError;

#include <CNuke/Common.h>

// Элемент таблицы, содержит ключ Key и значение Value
typedef struct {
    int Key;
    BaseType Value;
} TableElement;

typedef struct {
    TableElement Buf[TableBufferSize];
    size_t End;
} Table;

// Инициализирует пустую таблицу и возвращает её.
Table TableInit();

// Если в таблице T есть элемент с ключом E.Key, то обновляет его значение
// Если в таблице T нет элемента с таким ключом и T не переполнена, включает элемент
void TablePut(Table *T, TableElement E);

// Если в таблице T есть элемент с ключом E.Key, возвращает его и удаляет из таблицы
TableElement TableGet(Table *T, int key);
```

```
// Возвращает true, если очередь T пуста, иначе - false.
bool TableIsEmpty(Table T);

// Возвращает true, если очередь T заполнена, иначе - false.
bool TableIsFull(Table T);
```

Реализация

```
#include <CNuke/Table.h>

int TableError = TableOk;

Table TableInit() {
    TableError = TableOk;
    return (Table) {{0}, 0};
}

void TablePut(Table *T, TableElement E) {
    TableError = TableOk;
    for (int i = 0; i < T->End; i++) {
        if (E.Key == T->Buf[i].Key) {
            T->Buf[i].Value = E.Value;

            return;
        }
    }

    if (TableIsFull(*T))  {
        TableError = TableFull;
        return;
    }

    T->Buf[T->End++] = E;
}

TableElement TableGet(Table *T, int key) {
    if (TableIsEmpty(*T)) {
        TableError = TableEmpty;

        return (TableElement){0};
    }

    TableError = TableOk;
    for (int i = 0; i < T->End; i++) {
        if (key == T->Buf[i].Key) {
            TableElement result = T->Buf[i];
            T->Buf[i] = T->Buf[--T->End];

            return result;
        }
    }

    TableError = TableNoKey;
```

```
    return (TableElement){0};
}

bool TableIsEmpty(Table T) {
    return T.End == 0;
}

bool TableIsFull(Table T) {
    return T.End >= TableBufferSize;
}
```

9. Реализовать ОЛС на массиве
Интерфейс

```
// Структура данных односвязный линейный список на статическом массиве

#pragma once

#include <stdbool.h>

// Максимальное количество элементов в списке
#define SinglyLinkedArrayListBufferSize 100

// Операция выполнена успешно.
#define SinglyLinkedArrayListOk 0
// Лист пуст.
#define SinglyLinkedArrayListEmpty 1
// Память для нового элемента выделить не удалось.
#define SinglyLinkedArrayListFull 2
// Рабочий указатель стоит на последнем элементе списка.
#define SinglyLinkedArrayListEnd 3

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
extern int SinglyLinkedArrayListError;

#include <CNuke/Common.h>

typedef size_t SinglyLinkedArrayListElementPtr;

typedef struct {
    BaseType Value;
    SinglyLinkedArrayListElementPtr Next;
} SinglyLinkedArrayListElement;

typedef struct {
    SinglyLinkedArrayListElement Buf[SinglyLinkedArrayListBufferSize];
    bool Taken[SinglyLinkedArrayListBufferSize];
    SinglyLinkedArrayListElementPtr Begin;
    SinglyLinkedArrayListElementPtr Ptr;
} SinglyLinkedArrayList;

// Инициализирует пустой односвязный линейный список и возвращает его.
```

```
SinglyLinkedArrayList SinglyLinkedArrayListInit();

// Включает в односвязный линейный список элемент после рабочего указателя.
void SinglyLinkedArrayListPut(SinglyLinkedArrayList *L, BaseType E);

// Возвращает значение элемента идущего после рабочего указателя.
BaseType SinglyLinkedArrayListGet(SinglyLinkedArrayList *L);

// Передвигает рабочий указатель на следующий элемент.
void SinglyLinkedArrayListMovePtr(SinglyLinkedArrayList *L);

// Возвращает true, если односвязный линейный список L пуст, иначе - false.
bool SinglyLinkedArrayListIsEmpty(SinglyLinkedArrayList L);

// Перемещает рабочий указатель L в начало
void SinglyLinkedArrayListBeginPtr(SinglyLinkedArrayList *L);

// Перемещает рабочий указатель L в конец.
void SinglyLinkedArrayListEndPtr(SinglyLinkedArrayList *L);
```

Реализация

```
#include <CNuke/list/SinglyLinkedArrayList.h>

#include <stdint.h>

int SinglyLinkedArrayListError = SinglyLinkedArrayListOk;

SinglyLinkedArrayList SinglyLinkedArrayListInit() {
    SinglyLinkedArrayListError = SinglyLinkedArrayListOk;
    return (SinglyLinkedArrayList) {{0}, {true, false}, 0, 0};
}

void SinglyLinkedArrayListPut(SinglyLinkedArrayList *L, BaseType E) {
    SinglyLinkedArrayListError = SinglyLinkedArrayListOk;
    SinglyLinkedArrayListElementPtr newElement = 0;
    for (size_t i = 0; i < SinglyLinkedArrayListBufferSize && newElement == 0; i++) {
        if (!L->Taken[i])
            newElement = i;
    }

    if (newElement == 0) {
        SinglyLinkedArrayListError = SinglyLinkedArrayListFull;
        return;
    }

    L->Taken[newElement] = true;

    SinglyLinkedArrayListElementPtr oldElement = L->Buf[L->Ptr].Next;
    L->Buf[L->Ptr].Next = newElement;
    L->Buf[newElement] = (SinglyLinkedArrayListElement) {E, oldElement};
}
```

```
BaseType SinglyLinkedArrayListGet(SinglyLinkedArrayList *L) {
    SinglyLinkedArrayListError = SinglyLinkedArrayListOk;
    if (L->Buf[L->Ptr].Next == 0) {
        SinglyLinkedArrayListError = SinglyLinkedArrayListEnd;
        return;
    }

    SinglyLinkedArrayListElementPtr readElement = L->Buf[L->Ptr].Next;

    BaseType result = L->Buf[readElement].Value;
    L->Buf[L->Ptr].Next = L->Buf[readElement].Next;

    L->Taken[readElement] = false;

    return result;
}

void SinglyLinkedArrayListMovePtr(SinglyLinkedArrayList *L) {
    SinglyLinkedArrayListError = SinglyLinkedArrayListOk;
    if (L->Buf[L->Ptr].Next == 0) {
        SinglyLinkedArrayListError = SinglyLinkedArrayListEnd;
        return;
    }

    L->Ptr = L->Buf[L->Ptr].Next;
}

bool SinglyLinkedArrayListIsEmpty(SinglyLinkedArrayList L) {
    SinglyLinkedArrayListError = SinglyLinkedArrayListOk;
    return L.Buf[L.Begin].Next == 0;
}

void SinglyLinkedArrayListBeginPtr(SinglyLinkedArrayList *L) {
    SinglyLinkedArrayListError = SinglyLinkedArrayListOk;
    L->Ptr = L->Begin;
}

void SinglyLinkedArrayListEndPtr(SinglyLinkedArrayList *L) {
    SinglyLinkedArrayListError = SinglyLinkedArrayListOk;
    while (L->Buf[L->Ptr].Next != 0) {
        L->Ptr = L->Buf[L->Ptr].Next;
    }
}
```

10. Реализовать файл
Интерфейс

```
// Структура данных файл

#pragma once

#include <CNuke/container/ArrayQueue.h>
```

```c
// Операция выполнена успешно.
#define FileOk 0
// Достигнут конец файла.
#define FileEnd 1
// Недостаточно памяти для нового элемента.
#define FileNotMem 2

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
extern int FileError;

typedef struct {
    // Прочитанные данные
    ArrayQueue Read;
    // Непрочитанные данные
    ArrayQueue Buf;
} File;


// Инициализирует файл и возвращает его.
File FileInit();

// Прочитывает элемент из F и возвращает его, если можно выполнить это действие.
BaseType FileRead(File *F);

// Включает в файл F элемент E.
void FilePut(File* F, BaseType E);

// Устанавливает указатель в начало.
void FileBeginPtr(File* F);
```

Реализация

```c
#include <CNuke/File.h>

int FileError = FileOk;

File FileInit() {
    FileError = FileOk;
    return (File) {ArrayQueueInit(), ArrayQueueInit()};
}

BaseType FileRead(File *F) {
    FileError = FileOk;
    BaseType element = ArrayQueueGet(&F->Buf);

    if (ArrayQueueError != ArrayQueueOk) {
        FileError = FileEnd;
        return element;
    }

    ArrayQueuePut(&F->Read, element);
    if (ArrayQueueError != ArrayQueueOk) {
```

```
        FileError = FileNotMem;
        return element;
    }

    return element;
}

void FilePut(File* F, BaseType E) {
    FileError = FileOk;
    ArrayQueuePut(&F->Buf, E);
    if (ArrayQueueError != ArrayQueueOk) {
        FileError = FileNotMem;
    }
}

void FileBeginPtr(File* F) {
    FileError = FileOk;
    while (!ArrayQueueIsEmpty(F->Buf)) {
        ArrayQueuePut(&F->Read, ArrayQueueGet(&F->Buf));

        if (ArrayQueueError != ArrayQueueOk) {
            FileError = FileNotMem;
            return;
        }
    }

    ArrayQueue T = F->Buf;
    F->Buf = F->Read;
    F->Read = T;
}
```

11. Реализовать ДЛС при помощи двух стеков
Интерфейс

```
// Структура данных двусвязный линейный список на двух очередях

#pragma once

#include <stdbool.h>

// Операция выполнена успешно.
#define DoublyLinkedStackListOk 0
// Лист пуст.
#define DoublyLinkedStackListEmpty 1
// Не удалось выделить память для нового элемента.
#define DoublyLinkedStackListNotMem 2
// Рабочий указатель находится в конце списка
#define DoublyLinkedStackListEnd 3
// Рабочий указатель находится в начале списка
#define DoublyLinkedStackListBegin 4

// Код ошибки, обновляется каждый раз после выполнения операции над очередью.
```

```c
extern int DoublyLinkedStackListError;

#include <CNuke/Common.h>
#include <CNuke/container/LinkedStack.h>

typedef struct {
    LinkedStack Left;
    LinkedStack Right;
} DoublyLinkedStackList;

// Инициализирует пустой двусвязный линейный список и возвращает его.
DoublyLinkedStackList DoublyLinkedStackListInit();

// Включает в двусвязный линейный список L элемент E до текущего элемента.
void DoublyLinkedStackListPutBefore(DoublyLinkedStackList* L, BaseType E);

// Включает в двусвязный линейный список L элемент E после текущего элемента.
void DoublyLinkedStackListPutAfter(DoublyLinkedStackList* L, BaseType E);

// Возвращает значение элемента идущего до текущего элемента и удаляет его из списка L.
BaseType DoublyLinkedStackListGetBefore(DoublyLinkedStackList *L);

// Возвращает значение элемента идущего после текущего элемента и удаляет его из списка L.
BaseType DoublyLinkedStackListGetAfter(DoublyLinkedStackList *L);

// Передвигает текущий элемент на следующий в списке L.
void DoublyLinkedStackListMoveL(DoublyLinkedStackList *L);

// Передвигает текущий элемент на предыдущий в списке L.
void DoublyLinkedStackListMoveR(DoublyLinkedStackList *L);

// Освобождает память, занятую двусвязным линейным списком L.
void DoublyLinkedStackListDone(DoublyLinkedStackList *L);

// Перемещает текущий элемент в L в начало
void DoublyLinkedStackListBeginPtr(DoublyLinkedStackList *L);

// Перемещает текущий элемент в L в конец.
void DoublyLinkedStackListEndPtr(DoublyLinkedStackList *L);
```

Реализация

```c
#include <CNuke/list/DoubleLinkedStackList.h>

int DoublyLinkedStackListError = DoublyLinkedStackListOk;

DoublyLinkedStackList DoublyLinkedStackListInit() {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    return (DoublyLinkedStackList) {LinkedStackInit(), LinkedStackInit()};
}

void DoublyLinkedStackListPutBefore(DoublyLinkedStackList* L, BaseType E) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
```

```
    LinkedStackPut(&L->Left, E);

    if (LinkedStackError == LinkedStackNotMem)
        DoublyLinkedStackListError = DoublyLinkedStackListNotMem;
}

void DoublyLinkedStackListPutAfter(DoublyLinkedStackList* L, BaseType E) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    LinkedStackPut(&L->Right, E);

    if (LinkedStackError == LinkedStackNotMem)
        DoublyLinkedStackListError = DoublyLinkedStackListNotMem;
}

BaseType DoublyLinkedStackListGetBefore(DoublyLinkedStackList *L) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    BaseType result = LinkedStackGet(&L->Left);

    if (LinkedStackError == LinkedStackEmpty)
        DoublyLinkedStackListError = DoublyLinkedStackListBegin;

    return result;
}

BaseType DoublyLinkedStackListGetAfter(DoublyLinkedStackList *L) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    BaseType result = LinkedStackGet(&L->Right);

    if (LinkedStackError == LinkedStackEmpty)
        DoublyLinkedStackListError = DoublyLinkedStackListBegin;

    return result;
}

void DoublyLinkedStackListMoveL(DoublyLinkedStackList *L) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    BaseType element = DoublyLinkedStackListGetBefore(L);
    if (DoublyLinkedStackListError != DoublyLinkedStackListOk)
        return;

    DoublyLinkedStackListPutAfter(L, element);
}

void DoublyLinkedStackListMoveR(DoublyLinkedStackList *L) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    BaseType element = DoublyLinkedStackListGetAfter(L);
    if (DoublyLinkedStackListError != DoublyLinkedStackListOk)
        return;

    DoublyLinkedStackListPutBefore(L, element);
}

void DoublyLinkedStackListDone(DoublyLinkedStackList *L) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
```

```
    LinkedStackDone(&L->Left);
    LinkedStackDone(&L->Right);
}

void DoublyLinkedStackListBeginPtr(DoublyLinkedStackList *L) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    while (!LinkedStackIsEmpty(L->Left) && DoublyLinkedStackListError == DoublyLinkedStackListOk) {
        DoublyLinkedStackListMoveL(L);
    }
}

void DoublyLinkedStackListEndPtr(DoublyLinkedStackList *L) {
    DoublyLinkedStackListError = DoublyLinkedStackListOk;
    while (!LinkedStackIsEmpty(L->Right) && DoublyLinkedStackListError == DoublyLinkedStackListOk) {
        DoublyLinkedStackListMoveR(L);
    }
}
```

Ссылка на репозиторий