Белгородский государственный технологический университет им. В. Г. Шухова

Кафедра ПОВТиАС

Расчетно-графическое задание на тему   
«Линейные структуры данных»

Выполнил:  
студент группы ПВ-21  
Адаменко Игорь

Проверил:  
профессор  
Синюк В. Г.

Белгород  
2013

# Задание

Реализовать программно следующие линейные структуры данных:

1. Стек
   1. с использованием массива;
   2. с использованием списка;
2. Очередь
   1. с использованием массива (с приоритетами);
   2. с использованием списка;
3. Неупорядоченная таблица
   1. с использованием массива;
   2. с использованием списка;
4. Односвязный линейный список в динамической памяти
5. Двусвязный линейный список в динамической памяти
6. Дек, как отображение на двусвязный список

# Стек

Стек — это последовательность, в которой включение и исключение элемента осуществляется с одной стороны последовательности (со стороны вершины стека). Таким же образом осуществляется и операция доступа. Структура функционирует по принципу LIFO (последний пришедший обслуживается первым).

## Реализация с использованием массива

### StackArray.h

|  |  |
| --- | --- |
| 01 | **#ifndef** STACK\_ARRAY\_H |
| 02 | **#define** STACK\_ARRAY\_H |
| 03 |  |
| 04 | **#include** <stdio.h> |
| 05 | **#include** <stdlib.h> |
| 06 |  |
| 07 | **const short** StackSize = 1000; |
| 08 | **const short** StackOk = 0; |
| 09 | **const short** StackEmpty = 1; |
| 10 | **const short** StackFull = 2; |
| 11 | **static short** StackError; |
| 12 |  |
| 13 | **typedef** **int** BaseType; |
| 14 | **typedef** **struct** { |
| 15 | BaseType Buf[StackSize]; |
| 16 | **unsigned** Top; |
| 17 | } Stack; |
| 18 |  |
| 19 | *// Инициализация стека* |
| 20 | **void** InitStack(Stack \*S); |
| 21 |  |
| 22 | *// Вставка в стек* |
| 23 | **void** PutStack(Stack \*S, BaseType E); |
| 24 |  |
| 25 | *// Взятие из стека* |
| 26 | **void** GetStack(Stack \*S, BaseType \*E); |
| 27 |  |
| 28 | *// Проверка стека на заполненность* |
| 29 | **bool** FullStack(Stack \*S); |
| 30 |  |
| 31 | *// Проверка стека на пустоту* |
| 32 | **bool** EmptyStack(Stack \*S); |
| 33 |  |
| 34 | **#endif** |

### StackArray.c

|  |  |
| --- | --- |
| 01 | **#include** "StackArray.h" |
| 02 |  |
| 03 | *// Инициализация стека* |
| 04 | **void** InitStack(Stack \*S) { |
| 05 | S->Top = StackSize; |
| 06 | }; |
| 07 |  |
| 08 | *// Вставка в стек* |
| 09 | **void** PutStack(Stack \*S, BaseType E) { |
| 10 | **if** (S->Top == 0) { |
| 11 | StackError = StackFull; |
| 12 | **return**; |
| 13 | } |
| 14 |  |
| 15 | S->Buf[--S->Top] = E; |
| 16 | }; |
| 17 |  |
| 18 | *// Взятие из стека* |
| 19 | **void** GetStack(Stack \*S, BaseType \*E) { |
| 20 | **if** (S->Top == StackSize) { |
| 21 | StackError = StackEmpty; |
| 22 | **return**; |
| 23 | } |
| 24 |  |
| 25 | \*E = S->Buf[S->Top++]; |
| 26 | }; |
| 27 |  |
| 28 | *// Проверка стека на заполненность* |
| 29 | **bool** FullStack(Stack \*S) { |
| 30 | **if** (S->Top == 0) { |
| 31 | StackError = StackFull; |
| 32 | **return** **true**; |
| 33 | } |
| 34 | **return** **false**; |
| 35 | } |
| 36 |  |
| 37 | *// Проверка стека на пустоту* |
| 38 | **bool** EmptyStack(Stack \*S) { |
| 39 | **if** (S->Top == StackSize) { |
| 40 | StackError = StackEmpty; |
| 41 | **return** **true**; |
| 42 | } |
| 43 | **return** **false**; |
| 44 | }; |

## Реализация с использованием списка

### StackList.h

|  |  |
| --- | --- |
| 01 | **#ifndef** STACK\_LIST\_H |
| 02 | **#define** STACK\_LIST\_H |
| 03 |  |
| 04 | **#include** "SLL.h" |
| 05 |  |
| 06 | **const short** StackOk = ListOk; |
| 07 | **const short** StackEmpty = ListEmpty; |
| 08 | **const short** StackNotMem = ListNotMem; |
| 09 | **static short** StackError; |
| 10 |  |
| 11 | **typedef** List Stack; |
| 12 |  |
| 13 | *// Инициализация стека* |
| 14 | **void** InitStack(Stack \*S); |
| 15 |  |
| 16 | *// Вставка в стек* |
| 17 | **void** PutStack(Stack \*S, BaseType E); |
| 18 |  |
| 19 | *// Взятие из стека* |
| 20 | **void** GetStack(Stack \*S, BaseType \*E); |
| 21 |  |
| 22 | *// Очистка стека* |
| 23 | **void** DoneStack(Stack \*S); |
| 24 |  |
| 25 | *// Проверка на пустоту стека* |
| 26 | **bool** EmptyStack(Stack \*S); |
| 27 |  |
| 28 | **#endif** |

### StackList.c

|  |  |
| --- | --- |
| 01 | **#include** "StackList.h" |
| 02 |  |
| 03 | *// Инициализация стека* |
| 04 | **void** InitStack(Stack \*S) { |
| 05 | InitList(S); |
| 06 | StackError = ListError; |
| 07 | } |
| 08 |  |
| 09 | *// Вставка в стек* |
| 10 | **void** PutStack(Stack \*S, BaseType E) { |
| 11 | PutList(S, E); |
| 12 | StackError = ListError; |
| 13 | } |
| 14 |  |
| 15 | *// Взятие из стека* |
| 16 | **void** GetStack(Stack \*S, BaseType \*E) { |
| 17 | GetList(S, E); |
| 18 | StackError = ListError; |
| 19 | } |
| 20 |  |
| 21 | *// Очистка стека* |
| 22 | **void** DoneStack(Stack \*S) { |
| 23 | ClearList(S); |
| 24 | StackError = ListError; |
| 25 | } |
| 26 |  |
| 27 | *// Проверка на пустоту* |
| 28 | **bool** EmptyStack(Stack \*S) { |
| 29 | **return** EmptyList(S); |
| 30 | } |

# Очередь

Очередь — последовательность, в которую включают элементы с одной стороны, а исключают — с другой. Структура функционирует по принципу FIFO (поступивший первым обслуживается первым).

## Реализация с использованием массива

### QueueArray.h

|  |  |
| --- | --- |
| 01 | **#ifndef** QUEUE\_ARRAY\_H |
| 02 | **#define** QUEUE\_ARRAY\_H |
| 03 |  |
| 04 | **#include** <stdio.h> |
| 05 | **#include** <stdlib.h> |
| 06 |  |
| 07 | **const short** QueueSize = 1000; |
| 08 | **const short** QueueOk = 0; |
| 09 | **const short** QueueEmpty = 1; |
| 10 | **const short** QueueFull = 2; |
| 11 | **static short** QueueError; |
| 12 |  |
| 13 | **typedef** **int** BaseType; |
| 14 | **typedef** **struct** base { |
| 15 | BaseType data; |
| 16 | **int** priority; |
| 17 | } base; |
| 18 | **typedef** **struct** { |
| 19 | base Buf[QueueSize]; |
| 20 | **unsigned** Top; |
| 21 | **unsigned** Bottom; |
| 22 | } Queue; |
| 23 |  |
| 24 | *// Инициализация очереди* |
| 25 | **void** InitQueue(Queue \*Q); |
| 26 |  |
| 27 | *// Вставка в очередь* |
| 28 | **void** PutQueue(Queue \*Q, BaseType E, **int** P); |
| 29 |  |
| 30 | *// Взятие из очереди* |
| 31 | **void** GetQueue(Queue \*Q, BaseType \*E); |
| 32 |  |
| 33 | *// Проверка на заполненность* |
| 34 | **bool** FullQueue(Queue \*Q); |
| 35 |  |
| 36 | *// Проверка на пустоту* |
| 37 | **bool** EmptyQueue(Queue \*Q); |
| 38 |  |
| 39 | **#endif** |

### QueueArray.c

|  |  |
| --- | --- |
| 01 | **#include** "QueueArray.h" |
| 02 |  |
| 03 | *// Инициализация очереди* |
| 04 | **void** InitQueue(Queue \*Q) { |
| 05 | Q->Top = Q->Bottom = 0; |
| 06 | QueueError = QueueOk; |
| 07 | } |
| 08 |  |
| 09 | *// Вставка в очередь* |
| 10 | **void** PutQueue(Queue \*Q, BaseType E, **int** P) { |
| 11 | base tmp; |
| 12 | tmp.data = E; |
| 13 | tmp.priority = P; |
| 14 |  |
| 15 | **if** (FullQueue(Q)) { |
| 16 | **return**; |
| 17 | } |
| 18 | Q->Buf[Q->Bottom] = tmp; |
| 19 | Q->Bottom = (Q->Bottom + 1) % QueueSize; |
| 20 | } |
| 21 |  |
| 22 | *// Взятие из очереди* |
| 23 | **void** GetQueue(Queue \*Q, BaseType \*E) { |
| 24 | **int** i = Q->Top; |
| 25 | **int** maxPriority = Q->Buf[Q->Top].priority, maxIndex = i;; |
| 26 | **if** (EmptyQueue(Q)) { |
| 27 | **return**; |
| 28 | } |
| 29 | **while** (i != Q->Bottom) { |
| 30 | i = (i + 1) % QueueSize; |
| 31 | **if** (Q->Buf[i].priority > maxPriority) { |
| 32 | maxPriority = Q->Buf[i].priority; |
| 33 | maxIndex = i; |
| 34 | } |
| 35 | } |
| 36 |  |
| 37 | \*E = Q->Buf[maxIndex].data; |
| 38 | **while** (maxIndex != Q->Top) { |
| 39 | **int** newIndex; |
| 40 | newIndex = (maxIndex - 1 + QueueSize) % QueueSize; |
| 41 | Q->Buf[maxIndex] = Q->Buf[newIndex]; |
| 42 | maxIndex = newIndex; |
| 43 | } |
| 44 | Q->Top = (Q->Top + 1) % QueueSize; |
| 45 | } |
| 46 |  |
| 47 | *// Проверка на заполненность* |
| 48 | **bool** FullQueue(Queue \*Q) { |
| 49 | **if** (((Q->Bottom + 1) % QueueSize) == Q->Top) { |
| 50 | QueueError = QueueFull; |
| 51 | **return** **true**; |
| 52 | } |
| 53 | **return** **false**; |
| 54 | } |
| 55 |  |
| 56 | *// Проверка на пустоту* |
| 57 | **bool** EmptyQueue(Queue \*Q) { |
| 58 | **if** (Q->Top == Q->Bottom) { |
| 59 | QueueError = QueueEmpty; |
| 60 | **return** **true**; |
| 61 | } |
| 62 | **return** **false**; |
| 63 | } |

## Реализация с использованием списка

### QueueList.h

|  |  |
| --- | --- |
| 01 | **#ifndef** QUEUE\_LIST\_H |
| 02 | **#define** QUEUE\_LIST\_H |
| 03 |  |
| 04 | **#include** "SLL.h" |
| 05 |  |
| 06 | **const short** QueueOk = ListOk; |
| 07 | **const short** QueueEmpty = ListEmpty; |
| 08 | **const short** QueueNotMem = ListNotMem; |
| 09 | **static short** QueueError; |
| 10 |  |
| 11 | **typedef** List Queue; |
| 12 |  |
| 13 | *// Инициализация очереди* |
| 14 | **void** InitQueue(Queue \*Q); |
| 15 |  |
| 16 | *// Вставка в очередь* |
| 17 | **void** PutQueue(Queue \*Q, BaseType E); |
| 18 |  |
| 19 | *// Взятие из очереди* |
| 20 | **void** GetQueue(Queue \*Q, BaseType \*E); |
| 21 |  |
| 22 | *// Очистка очереди* |
| 23 | **void** DoneQueue(Queue \*Q); |
| 24 |  |
| 25 | *// Проверка на пустоту* |
| 26 | **bool** EmptyQueue(Queue \*Q); |
| 27 |  |
| 28 | **#endif** |

### QueueList.c

|  |  |
| --- | --- |
| 01 | **#include** "QueueList.h" |
| 02 |  |
| 03 | *// Инициализация очереди* |
| 04 | **void** InitQueue(Queue \*Q) { |
| 05 | InitList(Q); |
| 06 | QueueError = ListError; |
| 07 | } |
| 08 |  |
| 09 | *// Вставка в очередь* |
| 10 | **void** PutQueue(Queue \*Q, BaseType E) { |
| 11 | BeginPtr(Q); |
| 12 | PutList(Q, E); |
| 13 | QueueError = ListError; |
| 14 | } |
| 15 |  |
| 16 | *// Взятие из очереди* |
| 17 | **void** GetQueue(Queue \*Q, BaseType \*E) { |
| 18 | EndPtr(Q); |
| 19 | GetList(Q, E); |
| 20 | QueueError = ListError; |
| 21 | } |
| 22 |  |
| 23 | *// Очистка очереди* |
| 24 | **void** DoneQueue(Queue \*Q) { |
| 25 | ClearList(Q); |
| 26 | QueueError = ListError; |
| 27 | } |
| 28 |  |
| 29 | *// Проверка на пустоту* |
| 30 | **bool** EmptyQueue(Queue \*Q) { |
| 31 | **return** EmptyList(Q); |
| 32 | } |

# Неупорядоченная таблица

Таблица — последовательность записей, которые имеют одну и ту же организацию.

## Реализация с использованием массива

### UnsortedTableArray.h

|  |  |
| --- | --- |
| 01 | **#ifndef** UNSORTED\_TABLE\_ARRAY\_H |
| 02 | **#define** UNSORTED\_TABLE\_ARRAY\_H |
| 03 |  |
| 04 | **#include** <stdio.h> |
| 05 | **#include** <stdlib.h> |
| 06 |  |
| 07 | **const short** TableSize = 1000; |
| 08 | **const short** TableOk = 0; |
| 09 | **const short** TableEmpty = 1; |
| 10 | **const short** TableFull = 2; |
| 11 | **const short** TableNotContent = 3; |
| 12 | **const short** TableAlreadyContent = 4; |
| 13 | **static short** TableError; |
| 14 |  |
| 15 | **typedef** **int** DataType; |
| 16 | **typedef** **int** KeyType; |
| 17 | **typedef** **struct** base { |
| 18 | DataType data; |
| 19 | KeyType key; |
| 20 | } base; |
| 21 | **typedef** **struct** { |
| 22 | base Buf[TableSize]; |
| 23 | **unsigned** Top; |
| 24 | } Table; |
| 25 |  |
| 26 | *// Инициализация таблицы* |
| 27 | **void** InitTable(Table \*T); |
| 28 |  |
| 29 | *// Вставка в таблицу* |
| 30 | **void** PutTable(Table \*T, DataType E, KeyType K); |
| 31 |  |
| 32 | *// Взятие из таблицы* |
| 33 | **void** GetTable(Table \*T, DataType \*E, KeyType K); |
| 34 |  |
| 35 | *// Чтение из таблицы* |
| 36 | **void** ReadTable(Table \*T, DataType \*E, KeyType K); |
| 37 |  |
| 38 | *// Проверка на пустоту* |
| 39 | **bool** EmptyTable(Table \*T); |
| 40 |  |
| 41 | *// Проверка на заполненность* |
| 42 | **bool** FullTable(Table \*T); |
| 43 |  |
| 44 | **#endif** |

### UnsortedTableArray.c

|  |  |
| --- | --- |
| 01 | **#include** "UnsortedTableArray.h" |
| 02 |  |
| 03 | *// Инициализация таблицы* |
| 04 | **void** InitTable(Table \*T) { |
| 05 | T->Top = 0; |
| 06 | TableError = TableOk; |
| 07 | } |
| 08 |  |
| 09 | *// Вставка в таблицу* |
| 10 | **void** PutTable(Table \*T, DataType E, KeyType K) { |
| 11 | base tmp; |
| 12 | **if** (FullTable(T)) { |
| 13 | **return**; |
| 14 | } |
| 15 | ReadTable(T, &tmp.data, K); |
| 16 | **if** ((TableError == TableNotContent) || (TableError == TableEmpty) ) { |
| 17 | tmp.data = E; |
| 18 | tmp.key = K; |
| 19 | T->Buf[T->Top++] = tmp; |
| 20 | TableError = TableOk; |
| 21 | } **else** { |
| 22 | TableError = TableAlreadyContent; |
| 23 | } |
| 24 | } |
| 25 |  |
| 26 | *// Взятие из таблицы* |
| 27 | **void** GetTable(Table \*T, DataType \*E, KeyType K) { |
| 28 | **int** i = 0; |
| 29 | **if** (EmptyTable(T)) { |
| 30 | **return**; |
| 31 | } |
| 32 | **while** (i < T->Top) { |
| 33 | **if** (T->Buf[i].key == K) { |
| 34 | **int** j; |
| 35 | \*E = T->Buf[i].data; |
| 36 | for (j = i + 1; j < T->Top; j++) { |
| 37 | T->Buf[j - 1] = T->Buf[j]; |
| 38 | } |
| 39 | T->Top--; |
| 40 | **return**; |
| 41 | } |
| 42 | i++; |
| 43 | } |
| 44 | TableError = TableNotContent; |
| 45 | } |
| 46 |  |
| 47 | *// Чтение из таблицы* |
| 48 | **void** ReadTable(Table \*T, DataType \*E, KeyType K) { |
| 49 | **int** i = 0; |
| 50 | **if** (EmptyTable(T)) { |
| 51 | **return**; |
| 52 | } |
| 53 | **while** (i < T->Top) { |
| 54 | **if** (T->Buf[i].key == K) { |
| 55 | \*E = T->Buf[i].data; |
| 56 | **return**; |
| 57 | } |
| 58 | i++; |
| 59 | } |
| 60 | TableError = TableNotContent; |
| 61 | } |
| 62 |  |
| 63 | *// Проверка на пустоту* |
| 64 | **bool** EmptyTable(Table \*T) { |
| 65 | **if** (T->Top == 0) { |
| 66 | TableError = TableEmpty; |
| 67 | **return** **true**; |
| 68 | } |
| 69 | **return** **false**; |
| 70 | } |
| 71 |  |
| 72 | *// Проверка на заполненность* |
| 73 | **bool** FullTable(Table \*T) { |
| 74 | **if** (T->Top == TableSize) { |
| 75 | TableError = TableFull; |
| 76 | **return** **true**; |
| 77 | } |
| 78 | **return** **false**; |
| 79 | } |

## Реализация с использованием списка

### UnsortedTableList.h

|  |  |
| --- | --- |
| 01 | **#ifndef** UNSORTED\_TABLE\_LIST\_H |
| 02 | **#define** UNSORTED\_TABLE\_LIST\_H |
| 03 |  |
| 04 | **#include** <stdio.h> |
| 05 | **#include** <stdlib.h> |
| 06 | **#include** "SLL.h" |
| 07 |  |
| 08 | **const short** TableOk = ListOk; |
| 09 | **const short** TableEmpty = ListEmpty; |
| 10 | **const short** TableNotMem = ListNotMem; |
| 11 | **const short** TableEnd = ListEnd; |
| 12 | **const short** TableNotContent = 10; |
| 13 | **const short** TableAlreadyContent = 11; |
| 14 | **static short** TableError = ListError; |
| 15 |  |
| 16 | **typedef** List Table; |
| 17 |  |
| 18 | *// Инициализация таблицы* |
| 19 | **void** InitTable(Table \*T); |
| 20 |  |
| 21 | *// Вставка в таблицу* |
| 22 | **void** PutTable(Table \*T, DataType E, KeyType K); |
| 23 |  |
| 24 | *// Взятие из таблицы* |
| 25 | **void** GetTable(Table \*T, DataType \*E, KeyType K); |
| 26 |  |
| 27 | *// Чтение из таблицы* |
| 28 | **void** ReadTable(Table \*T, DataType \*E, KeyType K); |
| 29 |  |
| 30 | *// Проверка на пустоту* |
| 31 | **bool** EmptyTable(Table \*T); |
| 32 |  |
| 33 | *// Очистка таблицы* |
| 34 | **void** DoneTable(Table \*T); |
| 35 |  |
| 36 | **#endif** |

### UnsortedTableList.c

|  |  |
| --- | --- |
| 01 | **#include** "UnsortedTableList.h" |
| 02 |  |
| 03 | *// Инициализация таблицы* |
| 04 | **void** InitTable(Table \*T) { |
| 05 | InitList(T); |
| 06 | TableError = ListError; |
| 07 | } |
| 08 |  |
| 09 | *// Вставка в таблицу* |
| 10 | **void** PutTable(Table \*T, DataType E, KeyType K) { |
| 11 | BaseType tmp, tmp2; |
| 12 |  |
| 13 | tmp.data = E; |
| 14 | tmp.key = K; |
| 15 | BeginPtr(T); |
| 16 | **while** (TableError != TableEnd) { |
| 17 | MovePtr(T); |
| 18 | ReadList(T, &tmp2); |
| 19 | **if** (tmp2.key == tmp.key) { |
| 20 | TableError = TableAlreadyContent; |
| 21 | **return**; |
| 22 | } |
| 23 | TableError = ListError; |
| 24 | } |
| 25 | EndPtr(T); |
| 26 | PutList(T, tmp); |
| 27 | TableError = ListError; |
| 28 | } |
| 29 |  |
| 30 | *// Взятие из таблицы* |
| 31 | **void** GetTable(Table \*T, DataType \*E, KeyType K) { |
| 32 | BaseType tmp; |
| 33 |  |
| 34 | TableError = ListError; |
| 35 | BeginPtr(T); |
| 36 | **while** (TableError != TableEnd) { |
| 37 | MovePtr(T); |
| 38 | ReadList(T, &tmp); |
| 39 | **if** (tmp.key == K) { |
| 40 | GetList(T, &tmp); |
| 41 | \*E = tmp.data; |
| 42 | **return**; |
| 43 | } |
| 44 | TableError = ListError; |
| 45 | } |
| 46 | TableError = TableNotContent; |
| 47 | } |
| 48 |  |
| 49 | *// Чтение из таблицы* |
| 50 | **void** ReadTable(Table \*T, DataType \*E, KeyType K) { |
| 51 | BaseType tmp; |
| 52 |  |
| 53 | TableError = ListError; |
| 54 | BeginPtr(T); |
| 55 | **while** (TableError != TableEnd) { |
| 56 | MovePtr(T); |
| 57 | ReadList(T, &tmp); |
| 58 | **if** (tmp.key == K) { |
| 59 | \*E = tmp.data; |
| 60 | **return**; |
| 61 | } |
| 62 | TableError = ListError; |
| 63 | } |
| 64 | TableError = TableNotContent; |
| 65 | } |
| 66 |  |
| 67 | *// Проверка на пустоту* |
| 68 | **bool** EmptyTable(Table \*T) { |
| 69 | **return** EmptyList(T); |
| 70 | } |
| 71 |  |
| 72 | *// Очистка таблицы* |
| 73 | **void** DoneTable(Table \*T) { |
| 74 | ClearList(T); |
| 75 | } |

# Односвязный линейный список в динамической памяти

Связный список — структура данных, элементами которой являются записи, связанные друг с другом с помощью указателей, хранящихся в самих элементах. В односвязном линейном списке каждый элемент состоит из различных по назначению полей: содержательного и поля-указателя. В поле-указателе находится адрес следующего элемента. Поле последнего указателя имеет указание на конец списка — признак nil.

### SLL.h

|  |  |
| --- | --- |
| 01 | **#ifndef** SLL\_H |
| 02 | **#define** SLL\_H |
| 03 |  |
| 04 | **#include** <stdio.h> |
| 05 | **#include** <stdlib.h> |
| 06 |  |
| 07 | **const short** ListOk = 0; |
| 08 | **const short** ListNotMem = 1; |
| 09 | **const short** ListEmpty = 2; |
| 10 | **const short** ListEnd = 3; |
| 11 | **static short** ListError; |
| 12 |  |
| 13 | **typedef** **int** BaseType; |
| 14 | **typedef** **struct** element \* ptrel; |
| 15 | **typedef** **struct** element { |
| 16 | BaseType data; |
| 17 | ptrel next; |
| 18 | } element; |
| 19 | **typedef** **struct** { |
| 20 | ptrel Start; |
| 21 | ptrel ptr; |
| 22 | } List; |
| 23 |  |
| 24 | *// Инициализация списка* |
| 25 | **void** InitList(List \*L); |
| 26 |  |
| 27 | *// Вставка в список* |
| 28 | **void** PutList(List \*L, BaseType E); |
| 29 |  |
| 30 | *// Взятие из списка* |
| 31 | **void** GetList(List \*L, BaseType \*E); |
| 32 |  |
| 33 | *// Чтение из списка* |
| 34 | **void** ReadList(List \*L, BaseType \*E); |
| 35 |  |
| 36 | *// Проверка на пустоту* |
| 37 | **bool** EmptyList(List \*L); |
| 38 |  |
| 39 | *// Нахождение количества элементов в списке* |
| 40 | **unsigned** **int** Count(List \*L); |
| 41 |  |
| 42 | *// Перемещение текущего указателя в начало списка* |
| 43 | **void** BeginPtr(List \*L); |
| 44 |  |
| 45 | *// Перемещение текущего указателя в конец списка* |
| 46 | **void** EndPtr(List \*L); |
| 47 |  |
| 48 | *// Перемещение текущего указателя на один элемент вперед* |
| 49 | **void** MovePtr(List \*L); |
| 50 |  |
| 51 | *// Перемещение текущего указателя на заданную позицию* |
| 52 | **void** MoveTo(List \*L, **unsigned** **int** n); |
| 53 |  |
| 54 | *// Очистка списка* |
| 55 | **void** ClearList(List \*L); |
| 56 |  |
| 57 | *// Копирование первого списка во второй* |
| 58 | **void** CopyList(List \*L1, List \*L2); |
| 59 |  |
| 60 | **#endif** |

### SLL.c

|  |  |
| --- | --- |
| 01 | **#include** "SLL.h" |
| 02 |  |
| 03 | *// Инициализация списка* |
| 04 | **void** InitList(List \*L) { |
| 05 | L->Start = (ptrel)malloc(sizeof(element)); |
| 06 |  |
| 07 | **if** (L->Start == NULL) { |
| 08 | ListError = ListNotMem; |
| 09 | **return**; |
| 10 | } |
| 11 |  |
| 12 | L->ptr = L->Start; |
| 13 | L->ptr->next = NULL; |
| 14 | ListError = ListOk; |
| 15 | } |
| 16 |  |
| 17 | *// Вставка в список* |
| 18 | **void** PutList(List \*L, BaseType E) { |
| 19 | ptrel tmp = (ptrel)malloc(sizeof(element)); |
| 20 |  |
| 21 | **if** (tmp == NULL) { |
| 22 | ListError = ListNotMem; |
| 23 | **return**; |
| 24 | } |
| 25 |  |
| 26 | tmp->data = E; |
| 27 | tmp->next = L->ptr->next; |
| 28 | L->ptr->next = tmp; |
| 29 | L->ptr = L->ptr->next; |
| 30 | } |
| 31 |  |
| 32 | *// Взятие из списка* |
| 33 | **void** GetList(List \*L, BaseType \*E) { |
| 34 | ptrel tmp = L->ptr; |
| 35 |  |
| 36 | **if** (EmptyList(L)) { |
| 37 | **return**; |
| 38 | } |
| 39 |  |
| 40 | L->ptr = L->Start; |
| 41 | **while** (L->ptr->next != tmp) { |
| 42 | MovePtr(L); |
| 43 | } |
| 44 | \*E = L->ptr->next->data; |
| 45 | L->ptr->next = L->ptr->next->next; |
| 46 | **free**(tmp); |
| 47 | } |
| 48 |  |
| 49 | *// Чтение из списка* |
| 50 | **void** ReadList(List \*L, BaseType \*E) { |
| 51 | **if** (EmptyList(L)) { |
| 52 | **return**; |
| 53 | } |
| 54 | \*E = L->ptr->data; |
| 55 | } |
| 56 |  |
| 57 | *// Проверка на пустоту* |
| 58 | **bool** EmptyList(List \*L) { |
| 59 | **if** (L->Start->next == NULL) { |
| 60 | ListError = ListEmpty; |
| 61 | **return** **true**; |
| 62 | } |
| 63 | **return** **false**; |
| 64 | } |
| 65 |  |
| 66 | *// Нахождение количества элементов в списке* |
| 67 | **unsigned** **int** Count(List \*L) { |
| 68 | **unsigned** **int** counter = 0; |
| 69 | ptrel tmp = L->Start; |
| 70 |  |
| 71 | **while** (tmp->next != NULL) { |
| 72 | counter++; |
| 73 | tmp = tmp->next; |
| 74 | } |
| 75 |  |
| 76 | **return** counter; |
| 77 | } |
| 78 |  |
| 79 | *// Перемещение текущего указателя в начало списка* |
| 80 | **void** BeginPtr(List \*L) { |
| 81 | L->ptr = L->Start; |
| 82 | } |
| 83 |  |
| 84 | *// Перемещение текущего указателя в конец списка* |
| 85 | **void** EndPtr(List \*L) { |
| 86 | **if**(EmptyList(L)) { |
| 87 | **return**; |
| 88 | } |
| 89 | L->ptr = L->Start; |
| 90 | **while** (L->ptr->next != NULL) { |
| 91 | L->ptr = L->ptr->next; |
| 92 | } |
| 93 | } |
| 94 |  |
| 95 | *// Перемещение текущего указателя на один элемент вперед* |
| 96 | **void** MovePtr(List \*L) { |
| 97 | **if** (L->ptr->next == NULL) { |
| 98 | ListError = ListEnd; |
| 99 | **return**; |
| 100 | } |
| 101 | L->ptr = L->ptr->next; |
| 102 | } |
| 103 |  |
| 104 | *// Перемещение текущего указателя на указанную позицию* |
| 105 | **void** MoveTo(List \*L, **unsigned** **int** n) { |
| 106 | **unsigned** **int** counter = 0; |
| 107 | ptrel tmp = L->Start; |
| 108 |  |
| 109 | **while** (counter++ != n) { |
| 110 | tmp = tmp->next; |
| 111 | **if** (tmp == NULL) { |
| 112 | ListError = ListEnd; |
| 113 | **return**; |
| 114 | } |
| 115 | } |
| 116 | L->ptr = tmp; |
| 117 | } |
| 118 |  |
| 119 | *// Очистка списка* |
| 120 | **void** ClearList(List \*L) { |
| 121 | ptrel tmp, tmpnext; |
| 122 | tmp = L->Start; |
| 123 | tmpnext = tmp->next; |
| 124 |  |
| 125 | **while** (EmptyList(L)) { |
| 126 | tmp = tmpnext; |
| 127 | tmpnext = tmpnext->next; |
| 128 | **free**(tmp); |
| 129 | } |
| 130 |  |
| 131 | **free**(L->Start); |
| 132 | ListError = ListEmpty; |
| 133 | } |
| 134 |  |
| 135 | *// Копирование первого списка во второй* |
| 136 | **void** CopyList(List \*L1, List \*L2) { |
| 137 | ptrel tmp = L1->Start; |
| 138 |  |
| 139 | **while**(tmp->next != NULL) { |
| 140 | PutList(L2, tmp->next->data); |
| 141 | tmp = tmp->next; |
| 142 | } |
| 143 | } |

# Двусвязный линейный список в динамической памяти

Двусвязный линейный список — это такой связный список, в котором указатель можно перемещать как в одну, так и в другую сторону, поскольку в дескрипторе, помимо поля, указывающего на следующий элемент, есть также поле, указывающее на предыдущий элемент.

### DLL.h

|  |  |
| --- | --- |
| 01 | **#ifndef** DLL\_H |
| 02 | **#define** DLL\_H |
| 03 |  |
| 04 | **#include** <stdio.h> |
| 05 | **#include** <stdlib.h> |
| 06 |  |
| 07 | **const short** DListOk = 0; |
| 08 | **const short** DListNotMem = 1; |
| 09 | **const short** DListEmpty = 2; |
| 10 | **const short** DListEnd = 3; |
| 11 | **static short** DListError; |
| 12 |  |
| 13 | **typedef** **int** BaseType; |
| 14 | **typedef** **struct** element \* ptrel; |
| 15 | **typedef** **struct** element { |
| 16 | BaseType data; |
| 17 | ptrel next; |
| 18 | ptrel prev; |
| 19 | } element; |
| 20 | **typedef** **struct** { |
| 21 | ptrel Start; |
| 22 | ptrel ptr; |
| 23 | ptrel End; |
| 24 | } DList; |
| 25 |  |
| 26 | *// Инициализация списка* |
| 27 | **void** InitDList(DList \*D); |
| 28 |  |
| 29 | *// Вставка в список до указателя* |
| 30 | **void** PutDListL(DList \*D, BaseType E); |
| 31 |  |
| 32 | *// Вставка в список после указателя* |
| 33 | **void** PutDListR(DList \*D, BaseType E); |
| 34 |  |
| 35 | *// Взятие из списка до указателя* |
| 36 | **void** GetDListL(DList \*D, BaseType \*E); |
| 37 |  |
| 38 | *// Взятие из списка после указателя* |
| 39 | **void** GetDListR(DList \*D, BaseType \*E); |
| 40 |  |
| 41 | *// Чтение из списка* |
| 42 | **void** ReadDList(DList \*D, BaseType \*E); |
| 43 |  |
| 44 | *// Проверка на пустоту* |
| 45 | **bool** EmptyDList(DList \*D); |
| 46 |  |
| 47 | *// Подсчет количества элементов в списке* |
| 48 | **unsigned** **int** Count(DList \*D); |
| 49 |  |
| 50 | *// Перемещение текущего указателя в начало списка* |
| 51 | **void** BeginPtr(DList \*D); |
| 52 |  |
| 53 | *// Перемещение текущего указателя в конец списка* |
| 54 | **void** EndPtr(DList \*D); |
| 55 |  |
| 56 | *// Перемещение текущего указателя на один элемент вперед* |
| 57 | **void** MoveForward(DList \*D); |
| 58 |  |
| 59 | *// Перемещение текущего указателя на один элемент назад* |
| 60 | **void** MoveBack(DList \*D); |
| 61 |  |
| 62 | *// Перемещение текущего указателя на указанную позицию* |
| 63 | **void** MoveTo(DList \*D, unsigned int n); |
| 64 |  |
| 65 | *// Очистка списка* |
| 66 | **void** ClearDList(DList \*D); |
| 67 |  |
| 68 | *// Копирование первого списка во второй* |
| 69 | **void** CopyDList(DList \*D1, DList \*D2); |
| 70 |  |
| 71 | **#endif** |

### DLL.c

|  |  |
| --- | --- |
| 001 | **#include** "DLL.h" |
| 002 |  |
| 003 | *// Инициализация списка* |
| 004 | **void** InitDList(DList \*D) { |
| 005 | D->Start = (ptrel)malloc(sizeof(element)); |
| 006 | D->End = (ptrel)malloc(sizeof(element)); |
| 007 |  |
| 008 | **if** ((D->Start == NULL) || (D->End == NULL)) { |
| 009 | DListError = DListNotMem; |
| 010 | free(D->Start); |
| 011 | free(D->End); |
| 012 | **return**; |
| 013 | } |
| 014 |  |
| 015 | D->Start->next = D->End; |
| 016 | D->Start->prev = NULL; |
| 017 | D->End->prev = D->Start; |
| 018 | D->End->next = NULL; |
| 019 | D->ptr = D->Start; |
| 020 |  |
| 021 | DListError = DListOk; |
| 022 | } |
| 023 |  |
| 024 | *// Вставка в список до указателя* |
| 025 | **void** PutDListL(DList \*D, BaseType E) { |
| 026 | **if** (D->ptr != D->Start) { |
| 027 | ptrel tmp = (ptrel)malloc(sizeof(element)); |
| 028 |  |
| 029 | **if** (tmp == NULL) { |
| 030 | DListError = DListNotMem; |
| 031 | **return**; |
| 032 | } |
| 033 |  |
| 034 | tmp->data = E; |
| 035 | tmp->next = D->ptr; |
| 036 | tmp->prev = D->ptr->prev; |
| 037 |  |
| 038 | D->ptr->prev->next = tmp; |
| 039 | D->ptr->prev = tmp; |
| 040 | } |
| 041 | } |
| 042 |  |
| 043 | *// Вставка в список после указателя* |
| 044 | **void** PutDListR(DList \*D, BaseType E) { |
| 045 | **if** (D->ptr != D->End) { |
| 046 | ptrel tmp = (ptrel)malloc(sizeof(element)); |
| 047 |  |
| 048 | **if** (tmp == NULL) { |
| 049 | DListError = DListNotMem; |
| 050 | **return**; |
| 051 | } |
| 052 |  |
| 053 | tmp->data = E; |
| 054 | tmp->next = D->ptr->next; |
| 055 | tmp->prev = D->ptr; |
| 056 |  |
| 057 | D->ptr->next->prev = tmp; |
| 058 | D->ptr->next = tmp; |
| 059 | } |
| 060 | } |
| 061 |  |
| 062 | *// Взятие из списка до указателя* |
| 063 | **void** GetDListL(DList \*D, BaseType \*E) { |
| 064 | **if** (EmptyDList(D)) { |
| 065 | **return**; |
| 066 | } |
| 067 |  |
| 068 | **if** (D->ptr != D->Start) { |
| 069 | ptrel tmp = D->ptr->prev; |
| 070 | \*E = tmp->data; |
| 071 | D->ptr->prev = tmp->prev; |
| 072 | tmp->prev->next = D->ptr; |
| 073 | free(tmp); |
| 074 | } |
| 075 | } |
| 076 |  |
| 077 | *// Взятие из списка после указателя* |
| 078 | **void** GetDListR(DList \*D, BaseType \*E) { |
| 079 | **if** (EmptyDList(D)) { |
| 080 | **return**; |
| 081 | } |
| 082 |  |
| 083 | **if** (D->ptr != D->End) { |
| 084 | ptrel tmp = D->ptr->next; |
| 085 | \*E = tmp->data; |
| 086 | D->ptr->next = tmp->next; |
| 087 | tmp->next->prev = D->ptr; |
| 088 | free(tmp); |
| 089 | } |
| 090 | } |
| 091 |  |
| 092 | *// Чтение из списка* |
| 093 | **void** ReadDList(DList \*D, BaseType \*E) { |
| 094 | **if** (EmptyDList(D)) { |
| 095 | **return**; |
| 096 | } |
| 097 | \*E = D->ptr->data; |
| 098 | } |
| 099 |  |
| 100 | *// Проверка на пустоту* |
| 101 | bool EmptyDList(DList \*D) { |
| 102 | **if** (D->Start == D->End) { |
| 103 | DListError = DListEmpty; |
| 104 | **return** true; |
| 105 | } |
| 106 | **return** false; |
| 107 | } |
| 108 |  |
| 109 | *// Подсчет количества элементов в списке* |
| 110 | **unsigned int** Count(DList \*D) { |
| 111 | **unsigned int** counter = 0; |
| 112 | ptrel tmp = D->Start; |
| 113 |  |
| 114 | **while** (tmp->next != D->End) { |
| 115 | counter++; |
| 116 | tmp = tmp->next; |
| 117 | } |
| 118 |  |
| 119 | **return** counter; |
| 120 | } |
| 121 |  |
| 122 | *// Перемещение текущего указателя в начало списка* |
| 123 | **void** BeginPtr(DList \*D) { |
| 124 | D->ptr = D->Start; |
| 125 | } |
| 126 |  |
| 127 | *// Перемещение текущего указателя в конец списка* |
| 128 | **void** EndPtr(DList \*D) { |
| 129 | D->ptr = D->End; |
| 130 | } |
| 131 |  |
| 132 | *// Перемещение текущего указателя на один элемент вперед* |
| 133 | **void** MoveForward(DList \*D) { |
| 134 | **if** (D->ptr != D->End) { |
| 135 | DListError = DListEnd; |
| 136 | **return**; |
| 137 | } |
| 138 | D->ptr = D->ptr->next; |
| 139 | } |
| 140 |  |
| 141 | *// Перемещение текущего указателя на один элемент назад* |
| 142 | **void** MoveBack(DList \*D) { |
| 143 | **if** (D->ptr != D->Start) { |
| 144 | DListError = DListEnd; |
| 145 | **return**; |
| 146 | } |
| 147 | D->ptr = D->ptr->prev; |
| 148 | } |
| 149 |  |
| 150 | *// Перемещение текущего указателя на указанную позицию* |
| 151 | **void** MoveTo(DList \*D, **unsigned int** n) { |
| 152 | **unsigned int** counter = 0; |
| 153 | ptrel tmp = D->Start; |
| 154 |  |
| 155 | **while** (counter++ != n) { |
| 156 | tmp = tmp->next; |
| 157 | **if** (tmp == NULL) { |
| 158 | DListError = DListEnd; |
| 159 | **return**; |
| 160 | } |
| 161 | } |
| 162 | D->ptr = tmp; |
| 163 | } |
| 164 |  |
| 165 | *// Очистка списка* |
| 166 | **void** ClearDList(DList \*D) { |
| 167 | ptrel tmp, tmpnext; |
| 168 | tmp = D->Start; |
| 169 | tmpnext = tmp->next; |
| 170 |  |
| 171 | **while** (!EmptyDList(D)) { |
| 172 | tmp = tmpnext; |
| 173 | tmpnext = tmpnext->next; |
| 174 | free(tmp); |
| 175 | } |
| 176 |  |
| 177 | free(D->Start); |
| 178 | free(D->End); |
| 179 | DListError = DListEmpty; |
| 180 | } |
| 181 |  |
| 182 | *// Копирование первого списка во второй* |
| 183 | **void** CopyDList(DList \*D1, DList \*D2) { |
| 184 | ptrel tmp = D1->Start; |
| 185 |  |
| 186 | **while**(tmp->next != NULL) { |
| 187 | PutDList(D2, tmp->next->data); |
| 188 | tmp = tmp->next; |
| 189 | } |
| 190 | } |

# Дек, как отображение на двусвязный список

Дек (двусвязная очередь) — линейная структура (последовательность), в которой операции включения и исключения элементов могут выполняться как с одного, так и с другого конца последовательности.

### Deque.h

|  |  |
| --- | --- |
| 01 | **#ifndef** DEQUE\_DLL\_H |
| 02 | **#define** DEQUE\_DLL\_H |
| 03 |  |
| 04 | **#include** "DLL.h" |
| 05 |  |
| 06 | **const short** DequeOk = DListOk; |
| 07 | **const short** DequeNotMem = DListNotMem; |
| 08 | **const short** DequeEmpty = DListEmpty; |
| 09 | **const short** DequeEnd = DListEnd; |
| 10 | **static short** DequeError; |
| 11 |  |
| 12 | **typedef** DList Deque; |
| 13 |  |
| 14 | *// Инициализация дека* |
| 15 | **void** InitDeque(Deque \*D); |
| 16 |  |
| 17 | *// Вставка в начало дека* |
| 18 | **void** PutDequeFront(Deque \*D, BaseType E); |
| 19 |  |
| 20 | *// Вставка в конец дека* |
| 21 | **void** PutDequeEnd(Deque \*D, BaseType E); |
| 22 |  |
| 23 | *// Взятие из начала дека* |
| 24 | **void** GetDequeFront(Deque \*D, BaseType \*E); |
| 25 |  |
| 26 | *// Взятие из конца дека* |
| 27 | **void** GetDequeBack(Deque \*D, BaseType \*E); |
| 28 |  |
| 29 | *// Проверка на пустоту* |
| 30 | **bool** EmptyDeque(Deque \*D); |
| 31 |  |
| 32 | *// Очистка дека* |
| 33 | **void** DoneDeque(Deque \*D); |
| 34 |  |
| 35 | **#endif** |

### Deque.c

|  |  |
| --- | --- |
| 01 | **#include** "Deque.h" |
| 02 |  |
| 03 | *// Инициализация дека* |
| 04 | **void** InitDeque(Deque \*D) { |
| 05 | InitDList(D); |
| 06 | DequeError = DListError; |
| 07 | } |
| 08 |  |
| 09 | *// Вставка в начало дека* |
| 10 | **void** PutDequeFront(Deque \*D, BaseType E) { |
| 11 | BeginPtr(D); |
| 12 | PutDListR(D, E); |
| 13 | DequeError = DListError; |
| 14 | } |
| 15 |  |
| 16 | *// Вставка в конец дека* |
| 17 | **void** PutDequeEnd(Deque \*D, BaseType E) { |
| 18 | EndPtr(D); |
| 19 | PutDListL(D, E); |
| 20 | DequeError = DListError; |
| 21 | } |
| 22 |  |
| 23 | *// Взятие из начала дека* |
| 24 | **void** GetDequeFront(Deque \*D, BaseType \*E) { |
| 25 | MoveTo(D, 1); |
| 26 | GetDListR(D, E); |
| 27 | DequeError = DListError; |
| 28 | } |
| 29 |  |
| 30 | *// Взятие из конца дека* |
| 31 | **void** GetDequeBack(Deque \*D, BaseType \*E) { |
| 32 | EndPtr(D); |
| 33 | GetDListL(D, E); |
| 34 | DequeError = DListError; |
| 35 | } |
| 36 |  |
| 37 | *// Проверка на пустоту* |
| 38 | **bool** EmptyDeque(Deque \*D) { |
| 39 | **return** EmptyDList(D); |
| 40 | } |
| 41 |  |
| 42 | *// Очистка дека* |
| 43 | **void** DoneDeque(Deque \*D) { |
| 44 | ClearDList(D); |
| 45 | DequeError = DListError; |
| 46 | } |