III. **ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ**

Структура, отражающая отношение соседства между своими элементами, называется линейной. К линейным структурам можно отнести массивы, линейные списки и их разновидности: стеки, очереди, деки.

**1. Массивы**

1.1. Структура данных «массив»

Массивы являются наиболее широко используемыми структурами данных и предусмотрены во всех языках программирования. Массив состоит из элементов одного типа, называемого *базовым*, поэтому структура массива однородна. Базовый тип может быть как скалярным, так и структурированным, т.е. элементами массива могут быть числа, символы, строки, структуры, в том числе и массивы. Число элементов массива фиксировано, поэтому объём занимаемой массивом памяти остаётся неизменным. С каждым элементом массива связан один или несколько индексов. Они однозначно определяют место элемента в массиве и обеспечивают прямой доступ к нему. Индексы массива относятся к определённому порядковому типу, поэтому индексы можно вычислять. Это обеспечивает, с одной стороны, гибкость обработки элементов массива, с другой стороны, создаёт опасность выхода за пределы массива, если не предусмотрены соответствующие средства контроля.

В зависимости от числа индексов различают одномерные и многомерные массивы. Допустимое число индексов массива (размерность массива) и диапазоны изменения их значений устанавливаются языком программирования. Трансляторы с языка программирования могут уточнить эти значения. По стандарту языка Си размерность массива не может превышать 31, а нумерация индексов начинается всегда с нуля, т.е. индекс изменяется от 0 до n-1, где n – количество значений индекса (мощность индекса).

Областями применения массивов являются:

* числовые массивы в вычислительных задачах;
* матричная алгебра, экстраполяция, интерполяция;
* таблицы – массивы с элементами типа «запись»;
* управляющие и информационные таблицы в операционных системах, трансляторах, системах управления базами данных (СУБД);
* представление других структур: графов, деревьев.

1.2. Структуры хранения массивов

Массив в памяти хранится в виде вектора, т.е. все элементы размещаются в смежных участках памяти подряд, начиная с адреса, соответствующего началу массива. Элементы одномерного массива размещаются последовательно друг за другом a0, a1,…, an-1. Элементы двухмерного массива размещаются по строкам, т.е. наиболее быстро меняется последний индекс b0,0, b0,1,…, b0,n-1, b1,0, b1,1,…, b1,n-1,…, bm-1,n-1 (в Си, Паскале, ПЛ/1).

Вся информация, необходимая для управления массивом, задаётся при его описании в программе. Описание содержит имя массива, тип элементов, который однозначно определяет длину элемента, диапазоны изменения индексов или число значений индексов, если нижние границы индексов фиксированы. Таким образом, общее количество элементов массива и размер памяти для массива полностью определяются описанием массива. Транслятор выделяет необходимую память и строит управляющий блок массива – *дескриптор*, или информационный вектор, например, такого содержания (для одномерного массива):

* тип структуры;
* адрес начала массива Aн;
* тип элемента (длина элемента массива L);
* нижняя граница индекса iн;
* верхняя граница индекса iв.

В случае многомерного массива для каждого индекса задаются нижняя и верхняя границы или же для каждой строки (каждого индекса) создаётся свой дескриптор.

Этой информации достаточно как для доступа к элементам массива, так и для контроля над тем, чтобы значения индексов не выходили за установленные диапазоны. Доступ к любому i-тому элементу в одномерном массиве осуществляется по адресу:

Ai = Aн + (i - iн) \* L.

То, что размеры массива, формируемого транслятором, фиксированы, может явиться ограничивающим фактором применения готовой программы. Действительно, требуется, чтобы память для массива выделялась в размерах, необходимых для решения конкретной задачи, а каковы будут её потребности, заранее может быть неизвестно. В таких ситуациях массив можно строить в динамической памяти, получаемой с помощью средств управления памятью операционной системы. Управление доступом к элементам таких массивов осуществляется самой программой по вычисляемым индексам (адресам) элементов с использованием указателя.

2. Стек

2.1. Структуры хранения стека

*Стек* – это динамически изменяемый упорядоченный набор элементов, в котором засылка и выборка элементов производятся с одного и того же конца стека, который называется *вершиной* стека. То есть выборка элементов из стека осуществляется в порядке, обратном их засылке. Это правило формулируется так: «последним пришёл, первым вышел» (LIFO: «Last Input – First Output»).

Области применения стека:

* передача параметров в функции и процедуре;
* трансляция (синтаксический и семантический анализы, генерация кодов и т.д.);
* реализация рекурсии в программировании;
* реализация управления динамической памятью и т.п.

Стеки могут представляться в памяти либо в виде вектора, либо в виде линейного списка.

При *векторном представлении* под стек отводится сплошная область памяти, достаточно большая, чтобы в ней можно было поместить некоторое максимальное число элементов, которое определяется решаемой задачей. Граничные адреса этой области являются параметрами физической структуры стека – вектора. В процессе заполнения стека место последнего элемента (его адрес) помещается в указатель вершины стека. Если указатель выйдет за верхнюю границу стека, то стек считается переполненным и включение нового элемента становится невозможным. Поэтому для стека надо отводить достаточно большую память. Однако если стек в процессе решения задачи заполняется только частично, то память используется неэффективно. Так как под стек отводится фиксированный объём памяти, а количество элементов переменно, то говорят, что стек в векторной памяти – это *полустатическая* структура данных. Обычно в стеке элементы имеют один и тот же тип, поэтому обработка такого стека достаточно проста.

Многие современные вычислительные машины содержат в своей конструкции аппаратные стеки или средства работы со стеками. Однако даже в этом случае при разработке программ часто приходится использовать свои программные стеки. Стек, представленный как вектор, имеет вид, показанный на рис. 2.1.



Рис. 2.1. Структура стека в векторной памяти.

При *списковом представлении* стека память под дескриптор и под каждый элемент стека получают динамически; включение и выборка элемента осуществляются с начала списка, которое одновременно является вершиной стека (рис. 2.2). Переполнения стека в этом случае не происходит, однако алгоритмы обработки сложнее, а время обработки удлиняется, так как операции включения и выборки элементов сопряжены с обращением к операционной системе для получения или освобождения памяти.

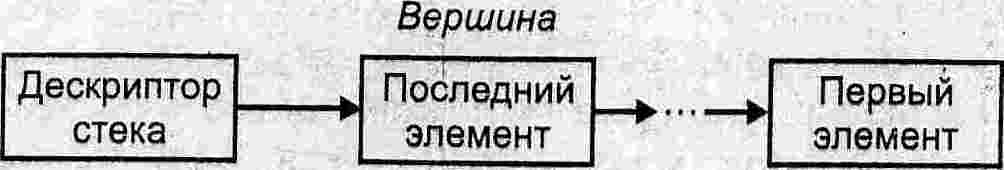


Рис 2.2 Списковая структура стека.

2.2. Операции над стеками

Основными операциями над стеками являются:

1) создание стека;

2) включение элемента в стек;

3) выборка элемента из стека;

4) извлечение данных;

5) уничтожение стека.

1) *Создание стека*.

Для стека в векторной памяти необходимо построить дескриптор и отвести память под элементы стека, исходя из максимально возможного их количества. Это может быть выполнено либо транслятором, либо путём получения динамической памяти в ходе выполнения программы. В любом случае при инициализации дескриптора в него обязательно заносятся указатели начала и конца стека, а в указателе вершины устанавливается признак «пусто».

Для стека списковой структуры создание стека сводится к построению только дескриптора. В нём достаточно наличия только указателя на конец списка, который одновременно является и указателем вершины списка, так как включение и выборка элемента осуществляются с вершины списка. Этот указатель устанавливается в NULL, обозначая, что стек пуст.

2) *Включение элемента в стек*.

В случае стека в векторной памяти, если нет переполнения стека, новый элемент включается в вершину стека, при переполнении – отказ от включения.

В случае стека списковой структуры переполнения, как правило, не бывает, поэтому включение сводится к получению динамической памяти под элемент и занесению туда данных. Очевидно, что в указатель в новом элементе заносится значение указателя на прежнюю вершину стека (для первого элемента это NULL), а в указатель вершины стека – адрес нового элемента.

3) *Выборка элемента из стека*. Она возможна только тогда, когда стек не пуст. Она означает, что выбирается значение из элемента, находящегося в вершине стека, а сам элемент исключается из стека. Исключение для векторного стека сводится к смещению указателя вершины на один элемент к началу стеку. При выборке из стека списковой структуры указатель на вершину (т.е. последний элемент) стека заменяется на адрес предпоследнего элемента, хранящийся в вершине стека, и освобождается память из-под прежней вершины стека.

4) *Извлечение данных* из любого элемента стека без удаления самого элемента. В запросе на выполнение этой операции необходимо указать номер элемента от вершины стека или какой-либо другой уникальный признак элемента. По нему отыскивается нужный элемент, если таковой имеется в стеке.

5) *Уничтожение стека*. Сводится к освобождению динамической памяти, полученной в процессе создания и обработки стека; если же элементы и дескриптор были построены транслятором, то сводится к очистке элементов списка и установке признака «пусто» в дескрипторе.

Рассмотрим *алгоритмы работы со стеками*, реализованными в виде однонаправленного списка.

1) *Процедура создания стека*.

Установка адреса текущего элемента стека Temp в Nil (пусто)

Ввод количества элементов стека n

Установка адреса вершины стека Top в Nil (пусто)

Цикл записи элементов стека i=1,…,n

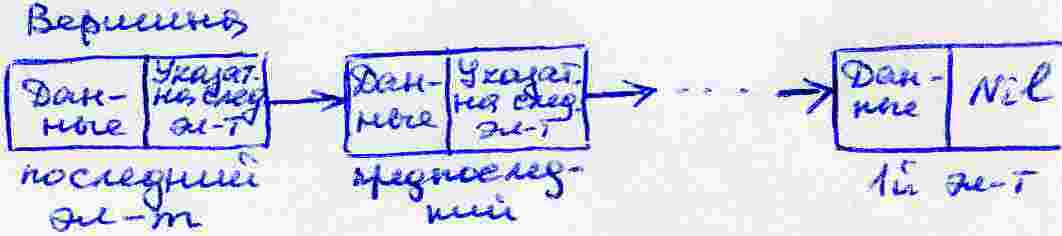
Выделение динамической памяти под текущий элемент стека с адресом Temp

Запись в область данных текущего элемента стека Temp^.Inf заданного значения

Присвоение адресу следующего элемента стека Temp^.Next значения адреса вершины стека

Присвоение адресу вершины стека Top адреса текущего элемента Temp (перенос вершины стека)

Вывод сообщения о создании стека



Type S2=^Rec2;

Rec2=record

Inf:integer; {значение элемента стека}

Next:S2; {адрес следующего элемента стека}

End;

Var Temp,Top:S2;

Procedure Creating\_of\_Stack;

Var i,n:byte;

Begin

Readln(n);

Temp:=Nil;

Top:=Nil;

For i:=1 to n do

Begin

New (Temp); { выделение памяти под новый элемент с адресом Temp }

Readln(Temp^.Inf);

Temp^.Next:=Top;

Top:=Temp;

End;

Writeln(‘Стек создан’);

End;

2) *Процедура включения элемента в стек*.

Ввод значения нового элемента стека Temp^.Inf

Присвоение адресу следующего элемента стека Temp^.Next значения адреса вершины стека Top

Присвоение адресу вершины стека Top адреса нового элемента стека Temp

Выделение динамической памяти для размещения нового элемента стека

Procedure Including\_in\_Stack;

Begin

New (Temp);

Readln(Temp^.Inf);

Temp^.Next:=Top;

Top:=Temp;

End;

4) *Процедура извлечения данных из стека* (с выводом значений на экран).

Присвоение адресу текущего элемента стека Temp адреса вершины стека Top

Стек пустой?

Текущий элемент существует?

да

нет

нет

Вывод значения информационного поля текущего элемента Temp^.Inf

да

Вывод соответствующего сообщения

Присвоение адресу текущего элемента стека Temp значения адреса следующего элемента Temp^.Next

Procedure View\_of\_Stack;

Begin

Temp:=Top;

If Temp=nil then

Writeln(‘Стек пустой’)

Else

While Temp<>nil do

Begin

Writeln(Temp^.Inf);

Temp:=Temp^.Next;

End;

End;

5*) Процедура удаления стека*.

Присвоение адресу текущего элемента стека Temp адреса вершины стека Top

Стек не пустой?

нет

да

Вывод соответствующего сообщения

Освобождение динамической памя- ти, занимаемой текущим элементом стека Temp; перенос вершины

Присвоение адресу вершины стека Top значения адреса следующего элемента стека Temp^.Next (перенос вершины стека на следующий элемент)

Procedure Delete\_Stack;

Begin

Temp:=Top;

While Top<>nil do

Begin

Top:=Temp^.Next;

Dispose (Temp);

Temp:=Top;

End;

Writeln(‘Стек очищен’);

End;

*Выборка* одного последнего элемента стека (3) выполняется аналогично, но без цикла (while заменить на if и убрать Temp:=Top в конце).

**3. Очередь (queue)**

*Очередь* – это динамически изменяемый упорядоченный набор элементов, в котором добавление элемента производится с одного конца (*хвоста* очереди), а выборка – с другого конца (*головы* очереди) в соответствии с правилом «Первым пришёл – первым ушёл» (FIFO: First Input – First Output). Такая очередь является простой очередью без приоритетов. Часто используются очереди с приоритетами, в них более приоритетные элементы включаются ближе к голове очереди, выборка осуществляется, как обычно, с головы очереди.

Очереди находят широкое применение в операционных системах (очереди задач, буфера ввода-вывода, буфера ввода с клавиатуры, очереди в сетях ЭВМ и т.п.), при моделировании реальных процессов и т.д.

Очереди могут иметь векторную или списковую структуру хранения. В свою очередь векторная структура может занимать статическую либо динамическую память. Очередь векторной структуры из-за ограниченности элементов имеет свойство переполнения, когда хвост очереди достигнет конца вектора. В этом случае добавление элементов становится невозможным, даже если в начальной части вектора будут свободные элементы из-под выбранных элементов. Для устранения такого недостатка образуют кольцевые очереди. При достижении конца вектора новые элементы добавляются в свободные элементы с начала вектора. Здесь также возможно переполнение, когда хвост догонит голову. Если же из-за выборки элементов голова догонит хвост, то очередь оказывается пустой. Изображение очереди в векторной памяти приведено на рис. 2.3.

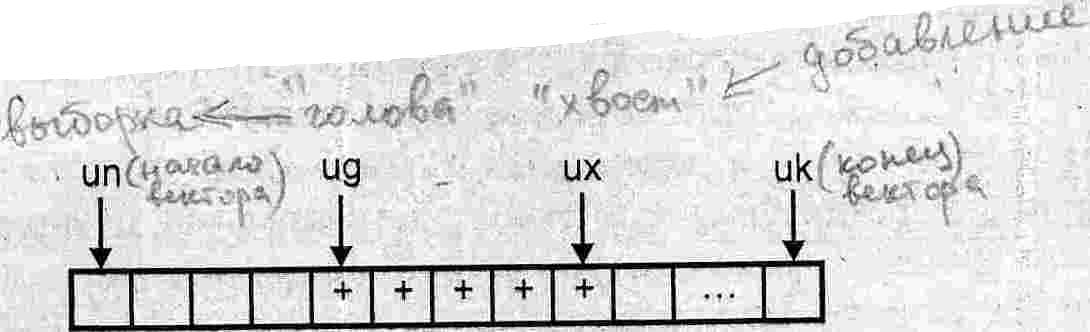


Рис. 3.1. Очередь в векторной памяти:

un – указатель начала вектора; uk – указатель конца вектора;

ug – указатель головы очереди; ux – указатель хвоста очереди;

+ - занятые ячейки очереди.

Основными операциями над очередью являются:

1) создание и освобождение очереди;

2) включение в очередь нового элемента;

3) выборка элемента из очереди.

При их выполнении используются такие вспомогательные операции, как проверка наличия элементов, проверка переполнения, организация перехода по кольцу, изменение приоритета и вызванная этим перестройка очереди.

Операции над очередями списковой структуры не вызывают никаких затруднений по сравнению со списками.

1) *Операции создания и освобождения очереди* векторной структуры такие же, как при работе со стеками. Дескриптор очереди должен содержать адрес начала вектора, указатели головы, хвоста и конца очереди в виде их адресов или индексов.

Опишем возможные состояния и соответствующие действия при выполнении операций включения и выборки элементов.

2) *Включение в очередь нового элемента*. 1. Очередь заполнена, тогда возврат признака переполнения. 2. Очередь пуста, тогда включение элемента в начало очереди, корректировка указателя хвоста. 3. Очередь заполнена частично, тогда добавление в первый свободный элемент с учётом кольцевой организации, корректировка указателя хвоста.

3) *Выборка элемента из очереди*. 1. Очередь пуста, тогда возврат соответствующего признака. 2. В очереди единственный элемент, тогда выбрать этот элемент и установить указатели в начальное состояние. 3. Очередь заполнена частично, тогда выбрать элемент с головы очереди с учётом кольцевой организации, скорректировать указатель головы.

*Очередь с приоритетами* – разновидность очереди, в которой на множестве элементов задана функция приоритета, т.е. для каждого элемента a множества можно вычислить функцию p(a) - приоритет элемента a, которая обычно принимает значения из множества действительных чисел или, в более общем случае, из некоторого линейно упорядоченного множества. При этом операция включения элемента в очередь с приоритетами осуществляется как в обычной очереди (т.е. с хвоста), а выборка элементов из очереди производится в соответствии с приоритетами всех элементов очереди, т.е. в первую очередь осуществляется выборка элемента с минимальным (или максимальным – как оговорено) значением его приоритета, независимо от его места в очереди.

Таким образом, отличие будет только в процедуре выборки элемента из очереди: сначала осуществляется поиск элемента с минимальным (максимальным) приоритетом, затем его выборка.

Возможны и другие операции, связанные с приоритетами: удаление из очереди элемента (элементов) с заданным приоритетом; поиск в очереди элемента (элементов) с заданным приоритетом и изменение значения приоритета и др.

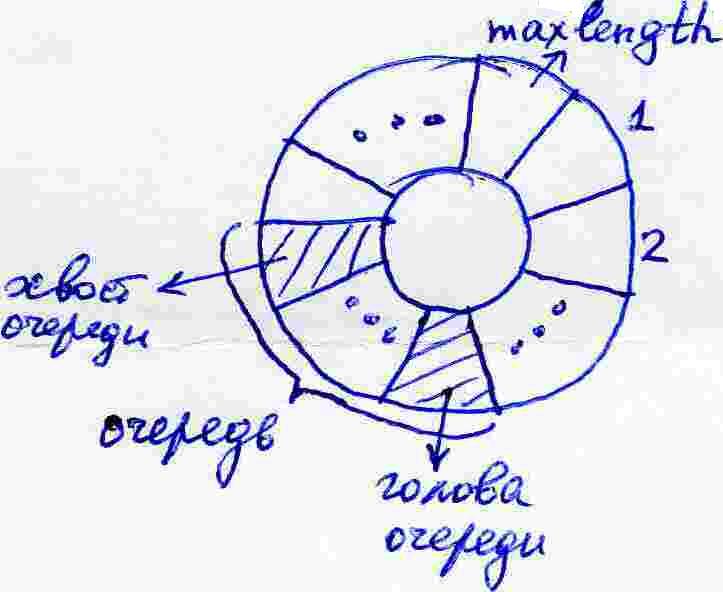
В качестве примера можно привести очередь с приоритетами, которая возникает среди множества вычислительных процессов, ожидающих обслуживания совместно используемыми ресурсами вычислительной системы. Обычно системные разработчики стараются сделать так, чтобы короткие вычислительные процессы выполнялись как можно скорее, поэтому такие процессы получают более высокие приоритеты, чем процессы, которые требуют (или уже израсходовали) значительное количество системного времени. Однако нельзя переусердствовать в применении этой стратегии, иначе процессы, требующие значительно больше времени, чем процессы средней длительности, вообще никогда не смогут получить машинного времени и будут находиться в режиме ожидания вечно.

Очередь при списковой реализации может быть представлена как однонаправленным, так и двунаправленным списком. В первом случае в элемент списка входит указатель на следующий элемент, что позволяет организовать перемещение по элементам очереди от головы до хвоста (т.е. от начала к концу) в отличие от стека, в котором перемещение возможно только от конца (вершины) к началу. Во втором случае в каждом элементе присутствуют указатели на следующий и на предыдущий элементы, что является избыточным. В случае очереди с приоритетом кроме поля данных в каждом элементе будет присутствовать поле приоритета.

*Реализация очередей с помощью циклических массивов*

Реализацию списков посредством массивов можно применить и для очередей, но в данном случае это было бы нерационально. Действительно, с помощью указателя на последний элемент очереди (хвост) можно вставить новый элемент в конец очереди за фиксированное число шагов, независимое от длины очереди (то есть перенести указатель хвоста), но удаление первого элемента очереди (головы) требует перемещения всех элементов очереди на одну позицию в массиве, то есть временная сложность операции удаления первого элемента очереди пропорциональна длине очереди.

Чтобы избежать этих вычислительных затрат, воспользуемся другим подходом, представив массив в виде циклической структуры, где первая ячейка массива следует за последней (maxlength).



Элементы очереди располагаются в «круге» ячеек последовательных позициях (причём последовательность позиций здесь понимается как циклическая непрерывность). Хвост очереди находится по часовой стрелке на определённом расстоянии от головы очереди. Теперь для вставки нового элемента в хвост очереди достаточно переместить указатель хвоста на одну позицию по часовой стрелке и записать значение элемента в эту позицию. При удалении элемента из головы очереди надо переместить указатель головы на одну позицию по часовой стрелке. Как видно, эта операция выполняется уже за фиксированное число шагов, не зависящее от длины очереди.

Для организации очереди следует ввести тип:

Type Queue = record

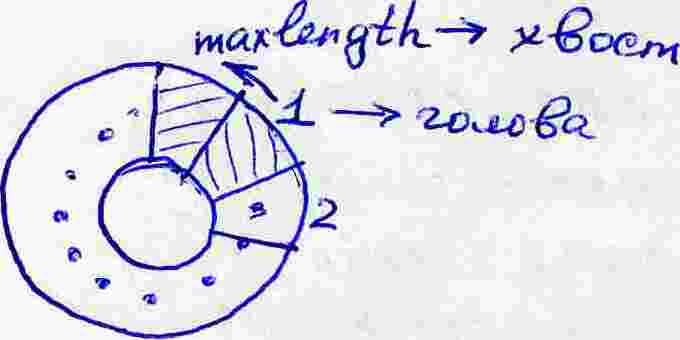
elements: array[1..maxlength]of <тип элементов>;

front, rear: integer {указатели головы и хвоста}

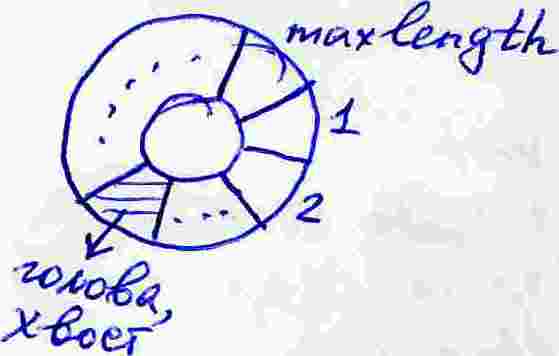
end;

При реализации очередей с помощью циклических массивов существует проблема, заключающаяся в невозможности только по формальному признаку взаимного расположения указателей головы и хвоста очереди судить о том, пуста ил очередь или заполнила весь массив.

Чтобы разобраться в этой проблеме, предположим, что очередь состоит из maxlength элементов, то есть полностью заполнила весь массив. Тогда указатель хвоста указывает на позицию рядом с головой, находящуюся против часовой стрелки.



Чтобы представить пустую очередь, сначала рассмотрим очередь, состоящую из одного элемента. Тогда указатели головы и хвоста указывают на одну и ту же позицию. Если удалить из очереди этот элемент, то указатель головы переместится на одну позицию по часовой стрелке, оформляя пустую очередь. Таким образом, в случае пустой очереди указатель хвоста указывает на позицию рядом с указателем головы, находящуюся против часовой стрелки, то есть точно так же, как и при полном заполнении массива.



Отсюда следует, что без введения каких-либо механизмов определения пустых очередей при максимальной длине массива maxlength нельзя позволить очереди иметь более maxlength-1 элементов.

Одним из таких механизмов является ведение переменной, которая принимает значение, например, TRUE, если очередь пуста.

Некоторые процедуры и функции, выполняющие операции над очередями, представленными в виде циклических массивов, можно найти в [3], с 65-66.

**4. Дек (deque)**

*Дек* – это разновидность очереди, в которой включение и выборка элементов возможны с обоих концов. Например, дек может использоваться при управлении памятью, когда распределение памяти производится и сверху, и снизу.

В свою очередь, существуют разновидности дека: *дек с ограниченным входом и дек с ограниченным выходом*. Дек с ограниченным входом допускает включение элементов только на одном конце, а дек с ограниченным выходом допускает выборку элементов только с одного конца.

Деки могут иметь как векторную, так и списковую структуру хранения. Операции над деками такие же, как и над очередями.

Рассмотрим операции с деком, реализованным в виде двунаправленного списка.

*1.* *Добавление элемента в конец дека*,

Список пустой?

нет

да

Выделение динамической памяти под последний элемент дека

Присвоение указателю на предыдущий элемент списка PointerLastMember^.Prev значения nil (т.к. список был пустой)

Ввод информационного поля последнего элемента дека PointerLastMember^.Inf

Присвоение указателю на последующий элемент списка PointerLastMember^.Next значения nil (т.к. элемент в списке всего один)

Присвоение начального значения nil адресу вспомогательной переменной MyPointer (буферу-указателю на добавляемый элемент) для определённости

Выделение динамической памяти под новый элемент списка с адресом MyPointer

Присвоение указателю на предыдущий элемент списка адреса последнего элемента PointerLastMember

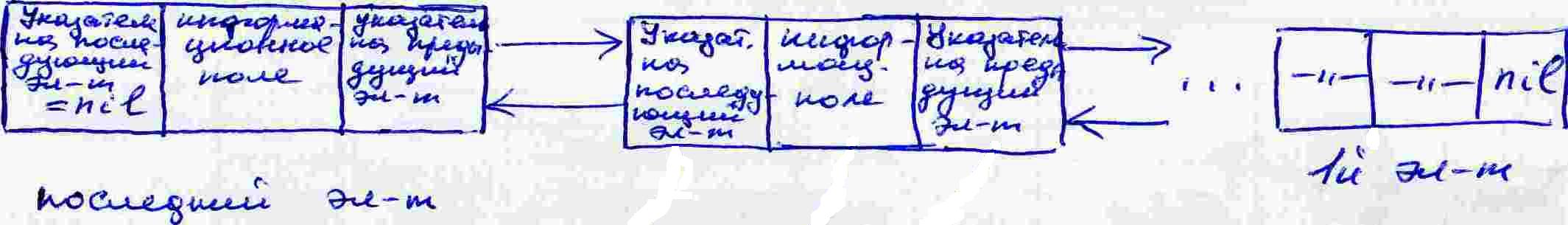
Ввод информационного поля нового элемента списка MyPointer^.Inf

Присвоение указателю на последующий элемент списка значения nil (т.к. элемент последний)

Присвоение указателю на последующий элемент для прежнего последнего элемента списка PointerLastMember^.Next адреса нового элемента

Присвоение указателю на последний элемент PointerLastMember адреса нового элемента MyPointer

Двунаправленный список:



Элемент списка будет иметь тип «запись»:

Type SpisokTwo=record

Prev:ListTwo; - указатель на предыдущий элемент

Inf:integer; - информационное поле

Next:ListTwo; - указатель на последующий элемент

End;

ListTwo=^SpisokTwo; - указатель на запись типа SpisokTwo

Глобальные переменные:

Var PointerFirstMember:ListTwo; - указатель на первый элемент списка

PointerLastMember:ListTwo; - указатель на последний элемент списка

Procedure AddToEndList (Var PointerLastMember:ListTwo);

Var MyPointer:ListTwo; {вводимый элемент списка}

Begin

If PointerLastMember=nil then {список пустой}

Begin

New(PointerLastMember);

PointerLastMember^.Prev:=nil;

Readln(PointerLastMember^.Inf);

PointerLastMember^.Next:=nil;

End;

Else {список непустой}

Begin

MyPointer:=nil;

New(MyPointer);

MyPointer^.Prev:=PointerLastMember;

Readln(MyPointer^.Inf);

MyPointer^.Next:=nil;

PointerLastMember^.Next:=MyPointer;

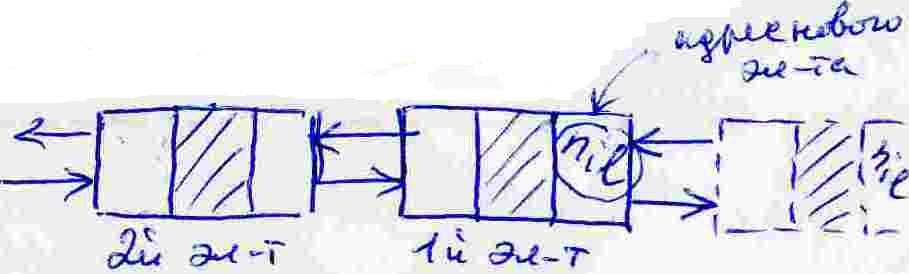
PointerLastMember:=MyPointer; {перенос указателя на последний элемент}

End;

End;

Добавление элемента в начало дека осуществляется аналогично.

*2. Добавления элемента в начало дека.*



Список пустой?

нет

да

Выделение динамической памяти под первый элемент дека

Присвоение указателю на предыдущий элемент списка PointerFirstMember^.Prev значения nil (т.к. список пустой)

Ввод информационного поля первого элемента PointerFirstMember^.Inf

Присвоение указателю на последующий элемент списка PointerFirstMember^.Next значения nil

Присвоение начального значения nil адресу MyPointer

Выделение динамической памяти под новый элемент c адресом MyPointer

Присвоение указателю на предыдущий элемент MyPointer^.Prev значения nil (т.к. это первый элемент списка)

Ввод информационного поля нового элемента списка MyPointer^.Inf

Присвоение указателю на последующий элемент списка MyPointer^.Next адреса прежнего первого элемента PointerFirstMember

Присвоение указателю на предыдущий элемент для прежнего первого элемента PointerFirstMember^.Prev адреса нового элемента MyPointer

Присвоение адресу нового первого элемента списка PointerFirstMember адреса нового элемента MyPointer

Procedure AddToStartList (var PointerFirstMember:ListTwo);

адрес нового первого элемента

Var MyPointer:ListTwo;

Begin

If PointerFirstMember=nil then

Begin

New(PointerFirsrMember);

PointerFirstMember^.Prev:=nil;

Readln(PointerFirstMember^.Inf);

PointerFirstMember^.Next:=nil;

End;

Else

Begin

MyPointer:=nil;

New(MyPointer);

MyPointer^.Prev:=nil;

Readln(MyPointer^.Inf);

MyPointer^.Next:=PointerFirstMember;

PointerFirstMember^.Prev:=MyPointer;

PointerFirstMember:=MyPointer;

End;

End;

*3. Процедура создания дека, состоящего из задаваемого числа N элементов*

i = 1

нет

да

Ввод количества элементов дека N

Присвоение начального значения nil адресу MyPointer (указателю на создаваемый элемент)

i = 1,…,N

Процедура добавления элемента в конец дека AddToEndList (MyPointer)

Присвоение указателю на первый элемент списка адреса первого созданного элемента MyPointer

Присвоение указателю на последний элемент списка адреса последнего созданного элемента MyPointer

Procedure CreateList (var PointerFirstMember:ListTwo;

var PointerLastMember:ListTwo);

Var i:byte;

MyPointer:ListTwo;

Begin

Readln(N); {N передаётся через глобальные переменные}

MyPointer:=nil;

For i:=1 to N do

Begin

AddToEndList(MyPointer);

If i=1 then PointerFirstMember:=MyPointer;

End;

PointerLastMember:=MyPointer;

End;

*4. Процедура выборки элемента из начала дека*.

Список пустой?

нет

да

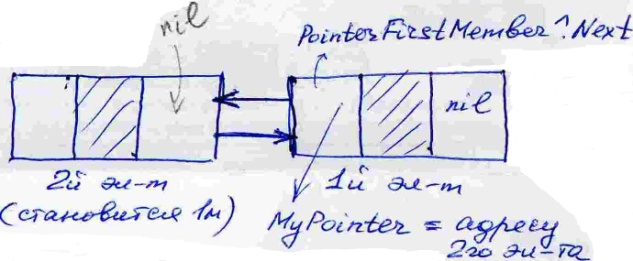
Вывод соответствующего сообщения «список пуст»

Освобождение динамической памяти, занимаемой первым элементом списка

Присвоить вспомогательной переменной MyPointer значение указателя на прежний второй элемент списка (который после выборки станет первым) PointerFirstMember^.Next

Присвоение указателю на первый элемент PointerFirstMember значения MyPointer

Присвоение указателю на элемент, предшествующий первому, значения nil



Procedure DelElemStartList (var PointerFirstMember:ListTwo);

Var MyPointer:ListTwo;

Begin

If PointerFirstMember=nil then writeln (‘Список пустой’);

Else

Begin

MyPointer:=PointerFirstMember^.Next;

Dispose(PointerFirstMember);

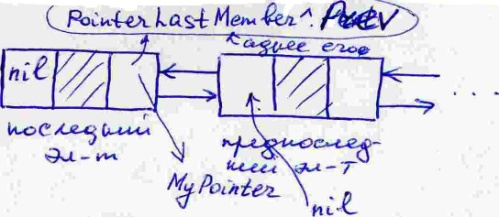
PointerFirstMember:=MyPointer;

PointerFirstMember^.Prev:=nil;

End;

End;

*5. Процедура выборки элемента из конца дека.*



Список пустой?

нет

да

Вывод соответствующего сообщения

Освобождение динамической памяти, занимаемой последним элементом списка

Присвоение вспомогательной переменной MyPointer значения указателя на предпоследний элемент списка (который после выборки станет последним) PointerLastMember^.Prev

Присвоение указателю на последний элемент PointerLastMember значения MyPointer

Присвоение значения nil указателю на последующий элемент для последнего элемента списка PointerLastMember^.Next

Procedure DelElemEndList (var PointerLastMember:ListTwo);

Var MyPointer:ListTwo;

Begin

If PointerLastMember=nil then writeln (‘Список пустой’);

Else

Begin

MyPointer:=PointerLastMember^.Prev;

Dispose(PointerLastMember);

PointerLastMember:=MyPointer;

PointerLastMember^.Next:=nil;

End;

End;

*6. Процедура нахождения элемента дека с заданным номером i.*

Вышли за границы списка?

нет

да

Вывод сообщения

Присвоение указателю на искомый элемент PointerFindMember начального значения, равного указателю на первый элемент списка PointerFirstMember

j = 1,…,i-1

Присвоение указателю на искомый элемент значения адреса следующего элемента списка PointerFindMember^.Next

Procedure FindElement (i:byte; PointerFirstMember:ListTwo;

var PointerFindMember:ListTwo);

Var j:byte;

Begin

PointerFindMember:=PointerFirstMember;

For j:=1 to i-1 do

Begin

PointerFindMember:=PointerFindMember^.Next;

If PointerFindMember=nil then

Begin

Writeln(‘Элемент с номером i выходит за границы списка’);

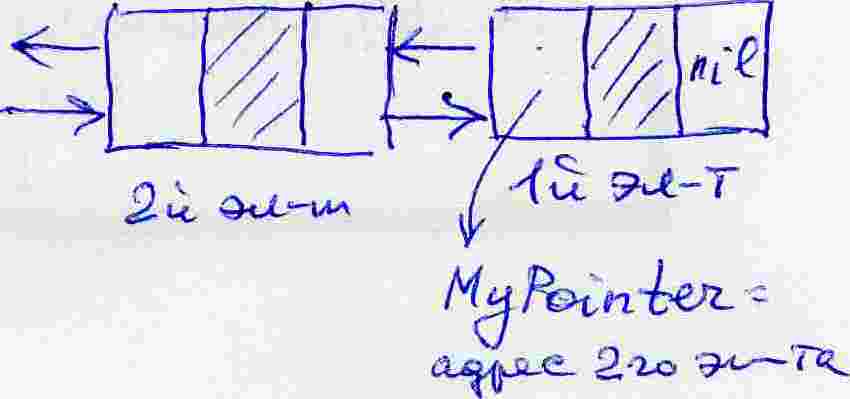
Break;

End;

End;

End;

*7. Процедура очистки дека (начиная с начала списка)*.



Список непустой?

нет

да

Освобождение динамической памяти, занимаемой первым элементом списка

Вспомогательной переменной MyPointer присвоить значение адреса элемента, следующего за удаляемым первым элементом PointerFirstMember^.Next

Присвоение указателю на первый элемент значения MyPointer (т.е. последующего элемента)

Список пустой?

да

нет

Procedure DelAllElem (var PointerFirstMember:ListTwo);

Var MyPointer:ListTwo;

Begin

If PointerFirstMember<>nil then

Repeat

MyPointer:=PointerFirstMember^.Next;

Dispose(PointerFirstMember);

PointerFirstMember:=MyPointer

Until PointerFirstMember=nil;

End;

Процедура очистки дека, начиная с конца списка.

Отличия:

Procedure DelAllElem (var PointerLastMember:ListTwo);

Var MyPointer:ListTwo;

Begin

If PointerLastMember<>nil then

Repeat

MyPointer:=PointerLastMember^.Prev;

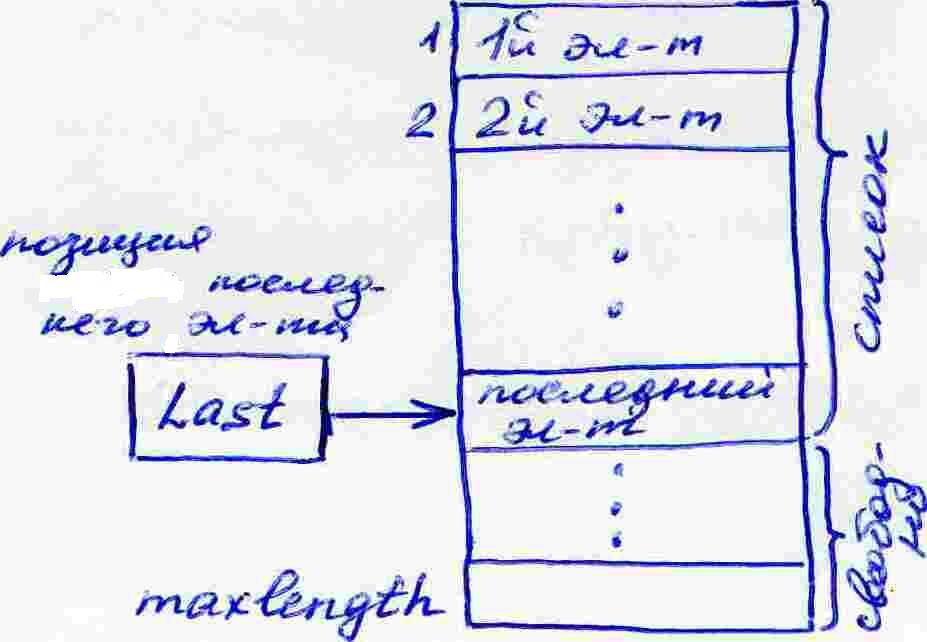
Dispose(PointerLastMember);

PointerLastMember:=MyPointer

Until PointerLastMember=nil;

End;

Векторная реализация списков



При векторной реализации списков (т.е. с помощью массивов) элементы списка располагаются в смежных ячейках массива. Это позволяет легко просматривать содержимое списка и вставлять новые элементы в его конец. Вставка нового элемента в середину списка требует перемещения всех последующих элементов на одну позицию к концу массива, чтобы освободить место для нового элемента. Удаление элемента также требует перемещения элементов, чтобы закрыть освободившуюся ячейку.

При использовании массива для реализации списка следует определить тип LIST как запись, содержащую два поля: первое поле – это элементы массива, чей размер maxlength считается достаточным для хранения списка любой длины, которые могут встретиться в данной программе; второе поле указывает на позицию последнего элемента списка в массиве.

Const maxlength = 100;

Type LIST=record

elements: array[1..maxlength]of <тип элемента массива>;

last: integer

end;

Рассмотрим некоторые процедуры работы со списками, реализованными посредством массивов.

1. Процедура вставки нового элемента списка в позицию р. При этом все элементы из позиций р, р+1, …, Last перемещаются в позиции р+1, р+2, …, Last+1, чтобы освободить место в позиции р под новый элемент.

Номер позиции последнего элемента списка ≥ максимального размера массива

нет

да

Перемещение j-того элемента массива в позицию j+1

j = <номер последней позиции списка>,…,p

Вывод сообщения «список полон»

Номер позиции p вставляемого элемента > номера позиции последнего элемента списка +1 или p<1

Вывод сообщения «такой позиции не существует»

да

нет

Увеличение номера позиции последнего элемента списка на 1

Ввод значения нового элемента списка

2. Процедура удаления элемента в позиции p списка. При этом элементы из позиций p+1,p+2,…,Last перемещаются в позиции p,p+1,…,Last-1.

Перемещение (j+1)-го элемента j-тую позицию

j = p,…,<номер последней позиции списка>

Номер позиции p удаляемого элемента > номера позиции последнего элемента списка +1 или p<1

Вывод сообщения «такой позиции не существует»

да

нет

Уменьшение номера позиции последнего элемента списка на 1

Сравнение реализаций списков на основе массивов и указателей

Ответ на вопрос, какую реализацию выгоднее использовать, зависит от операций, выполняемых над списками. Иногда аргументом в пользу выбора той или иной реализации служит максимальный размер обрабатываемых списков.

Приведём некоторые принципиальные соображения по этому поводу.

1. Реализация списков с помощью массивов требует указания максимального размера списка до начал выполнения программы. Если нельзя заранее ограничить сверху длину обрабатываемых списков, то более рациональным выбором будет реализация списков с помощью указателей.

2. Выполнение некоторых операторов в одной реализации требует б*о*льших вычислительных затрат, чем в другой. Например, процедуры добавления и выборки элемента выполняются за постоянное число шагов в случае связанных списков любого размера, а при использовании массивов требуют времени, пропорционального числу элементов, следующих за вставляемым или удаляемым элементом. В то же время функция определения длины списка имеет постоянное время выполнения при реализации списков посредством массивов, а в случае реализации с помощью указателей время пропорционально длине списка.

3. Реализация списков с помощью массивов неэкономна в отношении компьютерной памяти, так как резервируется объём памяти, достаточный для максимально возможного размера списка независимо от его реального размера в конкретный момент времени. Реализация с помощью указателей использует столько памяти, сколько необходимо для хранения текущего списка, но требует дополнительную память для указателя каждой ячейки.