Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Филиал

«Минский радиотехнический колледж»

Учебная дисциплина «Структуры и алгоритмы обработки данных»

**Инструкция**

по выполнению лабораторной работы

«Реализация очереди и основных алгоритмовее обработки»

Минск

2018

**Лабораторная работа № 6**

**Тема работы: «Реализация очереди и основных алгоритмов ее обработки»**

**1. Цель работы**

Изучение принципов работы со структурой данных очередь.

**2. Задание**

**Вариант 1.** За один просмотр файла действительных чисел напечатать элементы файла в следующем порядке: сначала – все числа, меньшие а, затем – все числа из отрезка [а, b], и наконец – все остальные числа, сохраняя исходный порядок в каждой из этих трех групп чисел. Числа а и b задает пользователь.

**Вариант 2.** Из заданного текста перенести все цифры в конец каждой строки, сохранив их порядок.

**3. Оснащение работы**

ПК, Pascal.

**4. Основные теоретические сведения**

**Очередь** – последовательный список переменной длины, в котором включение элементов выполняется только с одной стороны списка (конец или хвост очереди), а исключение – с другой стороны (начало или голова очереди). Очередь организована согласно дисциплине обслуживания FIFO:

**FIRST** – первый;

**INPUT** – вошел;

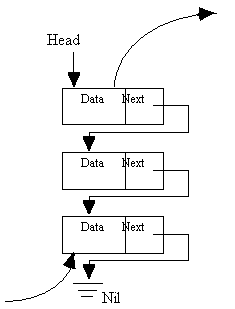
**FIRST** – первый;

**OUTPUT** – вышел.

*В отличие от обычной очереди, которую всегда можно при желании покинуть, из середины программистской очереди удалять элементы нельзя.*

Как и в «живой» очереди, здесь первым будет обслужен тот, кто пришел первым.

*Изобразим очередь графически:*



Очередь является динамической структурой – с течением времени изменяется и количество, и набор составляющих ее элементов.

Существует несколько способов реализации очереди:

- с помощью одномерного массива ;

- с помощью связанного списка;

- с помощью класса объектно-ориентированного программирования.

Стандартный набор операций, выполняемых над очередями:

- добавление элемента;

- удаление элемента;

- чтение первого элемента *(неразрушающее чтение)*;

- определение размера очереди;

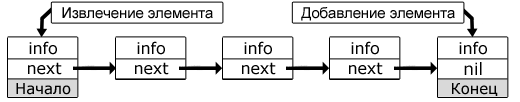
- очистка очереди.

**Реализация очереди с помощью связанного списка**

*Каждый элемент очереди должен иметь ссылку на следующий за ним элемент, поэтому элемент очереди объявляется как запись с двумя полями - информационное поле и связующее поле.*

*Для реализации операций с очередью необходимы уже две переменные: указатель* ***pFirst*** *на начало очереди и указатель* ***pLast*** *на конец очереди.*

*Приведенная ниже схема элементов очереди отражает логический порядок следования элементов, физически же элементы могут находиться в любых свободных областях памяти.*

[](http://kvodo.ru/wp-content/uploads/queue.png)

Опишем очередь на языке программирования:

**Type**

**EXO = ^O;**

**O = record**

**Data : integer;**

**Next : EXO;**

**end;**

*В очереди, в силу ее определения, доступны две позиции: ее конец, куда заносятся новые элементы, и ее начало, откуда извлекаются элементы. Поэтому для работы с очередью необходимо описать две переменные:*

**Var**

**BeginO, EndO : EXO;**

*где* ***BeginO*** *– соответствует началу очереди и будет использоваться для удаления элемента из очереди,* ***EndO*** *– соответствует концу очереди и будет использоваться для добавления новых элементов в очередь.*

*Создание пустой очереди включает в себя:*

*· выделение памяти для заголовка с помощью указателя* ***BeginO;***

*· занесение в ссылочную часть заголовка пустого указателя* ***nil;***

*· установка указателя конца очереди* ***EndO*** *=* ***BeginO.***

**Занесение** элемента в очередь реализуется как добавление элемента в конец списка.

*Пример:*

**Procedure writeO(Var BeginO, EndO : EXO; c : integer);**

**Var**

**u : EXO;**

**Begin**

**new(u);**

**u^.Data := c;**

**u^.Next := Nil;**

**if BeginO = Nil {проверяем, пуста ли очередь}**

**then**

**BeginO := u {ставим указатель начала очереди на первый созданный элемент}**

**Else**

**EndO^.Next := u; {ставим созданный элемент в конец очереди}**

**EndO := u; {переносим указатель конца очереди на последний элемент}**

**End;**

*Добавление нового элемента немного по-разному реализуется для пустой и непустой очереди. Аналогично, по-разному выполняется удаление из очереди, содержащей один или более одного элемента.*

Процедура **извлечения** элемента из очереди аналогична удалению элемента из начала списка. *Поскольку извлечение элемента из пустой очереди осуществить нельзя, опишем логическую функцию, проверяющую, есть ли элементы в очереди.*

*Пример:*

**Procedure readO(Var BeginO : EXO; Var c : integer);**

**Var**

**u : EXO;**

**Function FreeO(x1 : EXO): boolean;**

**Begin**

**FreeO := (x1 = Nil);**

**End;**

**Begin**

**if FreeO(BeginO)**

**then**

**writeln('Очередь пуста')**

**else**

**begin**

**c := BeginO^.Data; {считываем искомое значение в переменную с}**

**u := BeginO; {ставим промежуточный указатель на первый элемент очереди}**

**BeginO := BeginO^.Next;{указатель начала переносим на следующий элемент}**

**dispose(u); {освобождаем память, занятую уже ненужным первым элементом}**

**end;**

**End;**

*Алгоритмы работы с очередью:*

type **pQueueItem = ^ TQueueItem;**

**{ссылочный тип для адресации элементов очереди}**

**TQueueItem = record {базовый тип: структура элемента очереди}**

**inf : integer; {информационная часть}**

**next : pQueueItem;**

**{ссылочная часть: адрес следующего элемента}**

**end;**

**var pFirst, pLast : pQueueItem;**

*Тогда условие пустой очереди можно записать следующим образом:*

**pFirst^.next = nil**

*Для прохода по очереди от первого реального элемента к последнему необходимо:*

*- ввести вспомогательную ссылочную переменную* ***pTemp***

*- установить pTemp в адрес первого реального элемента:* ***pTemp := pFirst^.next***

*- организовать цикл по условию достижения конца очереди*

*- в цикле обработать очередной элемент с помощью указателя* ***pTemp*** *и изменить этот указатель:* ***pTemp := pTemp^.next***

*Добавление элемента в конец очереди выполняется следующим образом:*

*· выделить память для нового элемента с помощью стандартной функции New и вспомогательной ссылочной переменной pTemp:*

*· заполнить поля нового элемента, в частности в связующую часть установить значение nil: pTemp^.next := nil*

*· изменить связующую часть бывшего последнего элемента таким образом, чтобы она адресовала новый добавленный элемент: pLast^.next := pTemp;*

*· изменить значение указателя pLast так, чтобы он указывал новый последний элемент: pLast := pTemp;*

*Удаление элемента из начала очереди (но после заголовка!) выполняется следующим образом:*

*· адресуем удаляемый элемент с помощью вспомогательной переменной pTemp : pTemp := pFirst^.next;*

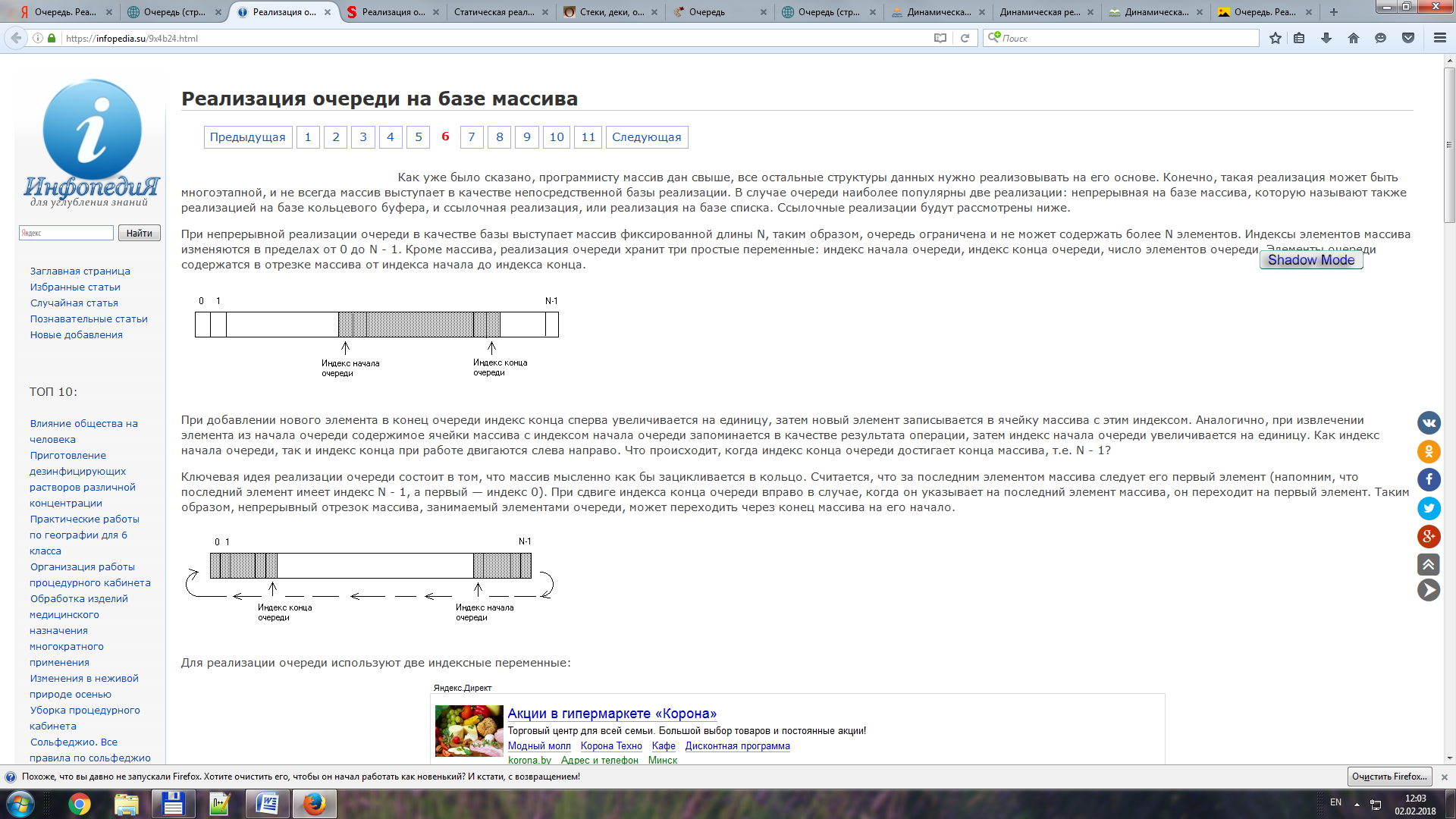
*· изменить связующую часть заголовка так, чтобы она указывала на второй элемент очереди, который теперь должен стать первым: pFirst^.next := pTemp^.next*

*· если после удаления в списке не остаётся реальных элементов, то необходимо изменить указатель pLast: pLast := pFirst*

*· обработать удаленный элемент, например - освободить занимаемую им память с помощью стандартной подпрограммы Dispose (pTemp) или включить его во вспомогательную очередь удаленных элементов.*

**Реализация очереди на базе массива**

При непрерывной реализации очереди в качестве базы выступает массив фиксированной длины N, таким образом, очередь ограничена и не может содержать более N элементов. Индексы элементов массива изменяются в пределах от 0 до N - 1. Кроме массива, реализация очереди хранит три простые переменные: индекс начала очереди, индекс конца очереди, число элементов очереди. Элементы очереди содержатся в отрезке массива от индекса начала до индекса конца.



При добавлении нового элемента в конец очереди индекс конца сперва увеличивается на единицу, затем новый элемент записывается в ячейку массива с этим индексом. Аналогично, при извлечении элемента из начала очереди содержимое ячейки массива с индексом начала очереди запоминается в качестве результата операции, затем индекс начала очереди увеличивается на единицу. Как индекс начала очереди, так и индекс конца при работе двигаются слева направо.

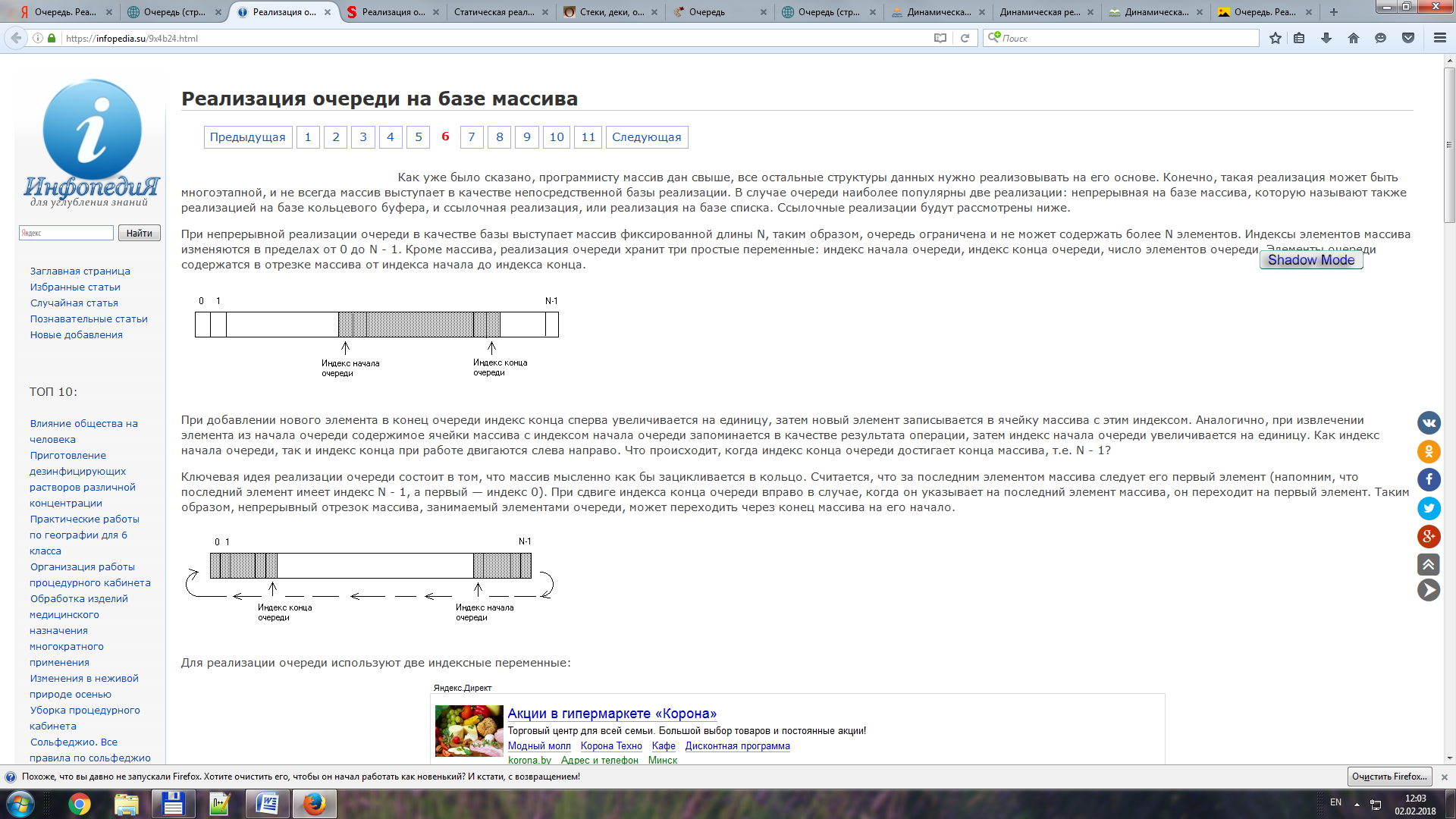
Очевидно, что со временем указатель на конец структуры данных при очередном включении элемента достигнет верхней границы той области памяти, которая выделена для очереди, т.е. индекс конца очереди достигает конца массива N – 1. Однако, если операции включения чередовались с операциями исключения элементов, то в начальной части отведенной под очередь памяти имеется свободное место.

*Рассмотренная выше простейшая реализация очереди-массива имеет один существенный недостаток: освобождающиеся при удалении ячейки в начале массива НЕ используются при последующих добавлениях, и поэтому при интенсивном использовании очереди быстро может возникнуть ситуация, когда указатель Last выходит за пределы массива, тогда как в начале массива есть свободные ячейки.*

Для устранения этого недостатка можно использовать два подхода:

1) при очередном удалении элемента из начала очереди сдвигать все элементы влево на одну ячейку, что при большом числе элементов в очереди может привести к большим вычислительным затратам;

2) более эффективно использовать так называемую кольцевую очередь, в которой при достижении указателем **Last** конца массива добавление производится в начало массива:



Ключевая идея реализации очереди состоит в том, что массив мысленно как бы зацикливается в кольцо. Считается, что за последним элементом массива следует его первый элемент (напомним, что последний элемент имеет индекс N - 1, а первый — индекс 0). При сдвиге индекса конца очереди вправо в случае, когда он указывает на последний элемент массива, он переходит на первый элемент. Таким образом, непрерывный отрезок массива, занимаемый элементами очереди, может переходить через конец массива на его начало.

*Таким образом, для того, чтобы места, занимаемые исключенными элементами, могли быть повторно использованы, очередь замыкается в кольцо: указатели (на начало и на конец), достигнув конца выделенной области памяти, переключаются на ее начало. Такая организация очереди в памяти называется кольцевой очередью. Реализацию очереди на базе массива называют также реализацией на базе кольцевого буфера.*

*Если в процессе работы с кольцевой очередью число операций включения превышает число операций исключения, то может возникнуть ситуация, в которой указатель конца "догонит" указатель начала. Это ситуация заполненной очереди, но если в этой ситуации указатели сравняются, эта ситуация будет неотличима от ситуации пустой очереди. Для различения этих двух ситуаций к кольцевой очереди предъявляется требование, чтобы между указателем конца и указателем начала оставался "зазор" из свободных элементов. Когда этот "зазор" сокращается до одного элемента, очередь считается заполненной и дальнейшие попытки записи в нее блокируются.*

Для программной реализации удобно ввести переменную-счетчик числа элементов в очереди, с помощью которой легко отслеживаются состояния пустой и заполненной очереди.

*Очистка очереди сводится к записи одного и того же (не обязательно начального) значения в оба указателя.*

*Определение размера состоит в вычислении разности указателей с учетом кольцевой природы очереди.*

*Программный пример иллюстрирует организацию очереди и операции на ней:*

**unit Queue; { Очередь FIFO - кольцевая }**

**Interface**

**const SIZE=...; { предельный размер очереди }**

**type data = ...; { эл-ты могут иметь любой тип }**

**Procesure QInit;**

**Procedure Qclr;**

**Function QWrite(a: data) : boolean;**

**Function QRead(var a: data) : boolean;**

**Function Qsize : integer;**

**Implementation { Очередь на кольце }**

**var QueueA : array[1..SIZE] of data; { данные очереди }**

**top, bottom : integer; { начало и конец }**

**Procedure QInit; {\*\* инициализация - начало=конец=1 }**

**begin top:=1; bottom:=1; end;**

**Procedure Qclr; {\*\*очистка - начало=конец }**

**begin top:=bottom; end;**

**Function QWrite(a : data) : boolean; {\*\* запись в конец }**

**begin**

**if bottom mod SIZE+1=top then { очередь полна } QWrite:=false**

**else begin**

**{ запись, модификация указ.конца с переходом по кольцу }**

**Queue[bottom]:=a; bottom:=bottom mod SIZE+1; QWrite:=true;**

**end; end; { QWrite }**

**Function QRead(var a: data) : boolean; {\*\* выборка из начала }**

**begin**

**if top=bottom then QRead:=false else**

**{ запись, модификация указ.начала с переходом по кольцу }**

**begin a:=Queue[top]; top:=top mod SIZE + 1; QRead:=true;**

**end; end; { QRead }**

**Function QSize : integer; {\*\* определение размера }**

**begin**

**if top <= bottom then QSize:=bottom-top**

**else QSize:=bottom+SIZE-top;**

**end; { QSize }**

**END.**

*Программа работы с кольцевой очередью на языке Паскаль*

**Programm Queue\_Test2;**

**Var**

**q:array[0..30] of Integer;**

**qnext,qindex,qlength:Integer;**

**procedure qstore(i:Integer);**

**begin**

**if (qnext+1)<>qlength then**

**begin**

**q[qnext]:=i;**

**qnext:=qnext+1;**

**if qnext=qlength then**

**qnext:=0        (\* Циклический переход \*)**

**end**

**else**

**writeln('Мест нет')**

**end;**

**function qretrieve():Integer;**

**begin**

**if qindex=qlength then**

**qindex:=0;        (\* Циклический переход \*)**

**if qindex=qnext then**

**begin**

**writeln('Очередь пуста');**

**qretrieve:=0;**

**end**

**else**

**begin**

**qretrieve:=q[qindex];**

**qindex:=qindex+1**

**end;**

**end;**

*Индексы сохранения* ***qnext*** *и извлечения* ***qindex*** *циклически возвращаются к началу массива (т.е. к 0). В такую очередь можно поместить любое количество элементов (если они не только помещаются, но и извлекаются из очереди).*

*Если* ***qindex*** *на 1 больше* ***qnext*** *– очередь заполнена и запись новых элементов невозможна, пока не будут прочитаны элементы, хранящиеся в очереди в данный момент. Если* ***qindex*** *равен* ***qnext*** *– очередь пуста. В остальных случаях существует место для помещения по крайней мере еще одного элемента.*

**Массовые операции**

Массовые операции — это операции, затрагивающие значительную часть всех элементов структуры данных. Пусть нужно добавить или удалить один элемент. Если при этом приходится, например, переписывать значительную часть остальных элементов с одного места на другое, то говорят, что добавление или удаление приводит к массовым операциям. Массовые операции — это бедствие для программиста, то, чего он всегда стремится избежать. Хорошая реализация структуры данных — та, в которой массовых операций либо нет совсем, либо они происходят очень редко. Например, добавление элемента должно выполняться за ограниченное число шагов, независимо от того, содержит ли структура десять или десять тысяч элементов.

В непрерывных реализациях добавление или удаление элементов в середине структуры неизбежно приводит к массовым операциям. Поэтому структуры, в которых можно удалять или добавлять элементы в середине, обязательно должны быть реализованы ссылочным образом.

**Приоритетные очереди. Понятие отношения порядка.**

**Приоритетная очередь** отличается от обычной тем, что имеет дополнительные позиции считывания (одну или несколько), расположенные ближе к хвосту очереди, или дополнительные позиции записи (одну или несколько), расположенные ближе к её голове. Чем ближе позиция считывания находится к хвосту очереди, тем выше приоритет этой позиции. Чем ближе позиция записи к голове очереди, тем выше приоритет. Предельным (но обычно не используемым) случаем такой очереди является одномерный массив, в котором приоритет позиции определяется её номером.

Процедуры, обслуживающие приоритетную очередь, должны изменять несколько индексов записи и/или считывания - основной (как в выше приведённых примерах), имеющий самый низкий приоритет, и один или несколько индексов с более высоким приоритетом (возрастающим, если таких индексов несколько). Данные могут быть считаны из дополнительной позиции (или записаны в неё), только если соответствуют уровню приоритета.

Но такие реализации неэффективны по времени выполнения основных операций. Так, например, поиск элемента с минимальным ключом в неупорядоченном массиве или списке требует последовательного просмотра всех его элементов. Если поддерживать упорядоченность массива или списка по ключу, то "неудобной" окажется операция вставки нового элемента.

Чаще всего приоритетная очередь представляется с помощью корневого дерева или набора корневых деревьев с определенными свойствами. При этом узлам дерева ставятся во взаимно однозначное соответствие элементы рассматриваемого множества.

**5. Порядок выполнения работы**

1. Создать файл, в соответствии с вариантом.

2. Используя основные алгоритмы работы с очередью, выполнить задание по варианту.

3. Результат вывести на экран.

**6. Форма отчета о работе**

*Лабораторная работа № \_\_\_*

*Номер учебной группы \_\_\_\_\_\_\_\_\_\_\_\_*

*Фамилия, инициалы учащегося \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Дата выполнения работы \_\_\_\_\_\_\_\_\_\_\_\_\_*

*Тема работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Цель работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Оснащение работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Результат выполнения работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

**7. Контрольные вопросы и задания**

1. Что представляет собой очередь?

2. Какие известны виды очередей?

3. На основе каких структур данных могут организовываться очереди?

4. Какой характер имеет операция считывания для очередей?

5. Какими свойствами обладают очереди?

6. Каким недостатком обладает простая очередь? Каков способ борьбы с этим недостатком?

7. Чем отличается приоритетная очередь от простой?

8. К каким структурам данных относятся очереди?

**8. Рекомендуемая литература**

1.Ахо, А.В.Структуры данных и алгоритмы / А.В. Ахо, Дж.Хопкрофт, Дж. Д.Ульман. – пер. с англ. – М.: Издательский дом "Вильямс", 2007.-400 с.

2. Вирт,Н. Алгоритмы и структуры данных / Н. Вирт. –СПб.:Невский диалект, 2008. – 352с.

3. Вирт, Н. Алгоритмы и структуры данных с примерами на Паскале / Н. Вирт[перевод с английского Д. Б. Подшивалова]; – 2-е изд., испр., – СПб.: Невский Диалект, 2005. – 352с.

4. Гагарина, Л.Г. Алгоритмы и структуры данных / Л.Г. Гагарина, В.Д. Колдаев. – учеб.пособие – М: Финансы и статистика, 2009. – 304с.

5. Котов, В.М. Алгоритмы и структуры данных: учеб.пособие / В.М. Котов, Е.П. Соболевская, А.А. Толстиков – Минск: БГУ, 2011. – 267с.

6. Макконнелл, Дж. Основы современных алгоритмов / Дж. Макконнелл – 2-е дополненное издание – М.:Техносфера, 2006. – 368с.

7. Окулов, С.М. Программирование в алгоритмах / С.М. Окулов. – 3-е изд. – М.: БИНОМ. Лаборатория знаний, 2007. – 383с.