Стеки и очереди

Стеки

Стек представляет собой структуру данных, в которой операции добавления и удаления осуществляются с одного конца, который называется вершиной стека. Доступ всегда имеется только к тому элементу, который был добавлен последним. Если возникла необходимость обратиться к предпоследнему элементу, то сначала надо будет удалить элемент, хранящийся в вершине стека. Такую дисциплину обслуживания часто называют LIFO - last in first out.

В качестве аналогии стека удобно рассматривать стопку бумаг на столе: человек, сидящий за столом видит перед собой только тот лист, который был положен в стопку последним. Также довольно часто стек сравнивают с оружейным магазином, оснащённым пружиной: стрельба всегда начинается с патрона, помещённого в магазин последним. Хоть это сравнение и не является корректным (в магазине при добавлении патрона происходит сдвиг всех ранее заряженных патронов, а в стеке такого перемещения не происходит), но именно пружинная аналогия породила устоявшиеся названия для процедур добавления (`push`) и удаления (`pop`) элементов.

Помимо процедур `push` и `pop` для стеков часто определяют следующие процедуры:

- `Top` - возвращает элемент, хранящийся на вершине стека, не удаляя его;
- `IsEmpty` - возвращает значение `true`, если стек пуст, и `false` в противном случае;
- `IsFull` - возвращает значение true, если стек заполнен, и false в противном случае;
- `ClearStack` - очищает стек и делает его пустым.

Рассмотрим реализацию этих процедур.

Реализация стека на основе массива

Прежде всего опишем тип, который будет представлять стек. Для простоты будем считать, что стек хранит целые числа, тогда объявление типа будет выглядеть следующим образом:

```
const
  MaxStackLength=100;
type
  TStack = record
    ind: word;
    elems: array [1.. MaxStackLength][1] of integer;
  end;
```

При такой организации ind указывает на вершину стека, а сами элементы хранятся в elems, причём первый добавленный элемент хранится в 1 ячейке, второй - во второй и т.д.

Реализация процедур для работы со стеком не представляет особой сложности.

---

[1] Очевидно, что стек, объявленный таким образом, может хранить не более 100 элементов, но ничто не мешает построить стек на основе динамического массива

Очистка стека (ClearStack)

Как и в случае с массивом, нет необходимости действительно удалять элементы, можно просто сбросить указатель вершины стека.

```
procedure ClearStack;
begin
  S.ind:=0;
end;
```

Проверка на заполненность (isFull)

Стек будет заполнен тогда, когда индекс ind будет равен MaxStackLength. Если используется статический массив, добавление элемента приведёт к ошибке, а в случае динамического массива, к расширению массива.

```
function IsFull: boolean;
begin
  IsFull:=S.ind=MaxStackLength;
end;
```

Добавление элемента в стек (push)

При добавлении элемента стек необходимо сдвинуть указатель ind на очередную пустую позицию в массиве elems и записать новый элемент в эту позицию.

```
procedure push(newelem: integer);
begin
  S.ind:=S.ind+1;
  S.elems[S.ind]:=newelem;
end;
```

Проверка на пустоту (IsEmpty)

Для того, чтобы узнать, есть ли элементы в стеке достаточно проверить значение ind. Если оно не равно нулю, то в стеке ещё есть элементы. Более того, в ind хранится их количество, но, работая со стеком, мы не должны знать сколько в нём элементов, поэтому функция isempty возвращает только значения true, если стек пуст, и false в противном случае.

```
function isempty: boolean;
begin
  isempty:=S.ind=0;
end;
```

Чтение с вершины стека (top)

Элемент-вершина всегда имеет индекс ind, поэтому для того, чтобы прочитать элемент с вершины надо вернуть элемент с индексом ind. Важно помнить, что функцию top нельзя вызывать, если массив пуст.

```
function top: integer;
begin
  top:=S.elems[S.ind];
end;
```

Чтение и удаление с вершины стека (pop)

Для того, чтобы прочитать все элементы стека, необходимо поочерёдно считывать элементы с вершины и удалять их. Ещё раз отметим, что вместо удаления выполняется простое уменьшение ind.

```
function pop: integer;
begin
  pop:=S.elems[S.ind];
  S.ind:=S.ind-1;
end;
```

Процедуру добавления и удаления элементов можно вызывать только в том случае, если стек не полон (для добавления) и не пуст (для удаления). Очевидно, что сложность всех описанных процедур не зависит от количества элементов в стеке, а значит, оценивается как $O(C)$.

Очереди

Очередь, как и стек, это АТД. В отличие от стека, где элементы добавляются и удаляются с одного и того же конца (вершины стека), в очереди элементы удаляются из одного конца (переднего), а добавляются в другой конец (задний). Принятую в очередях дисциплину обслуживания называют FIFO - first in first out.

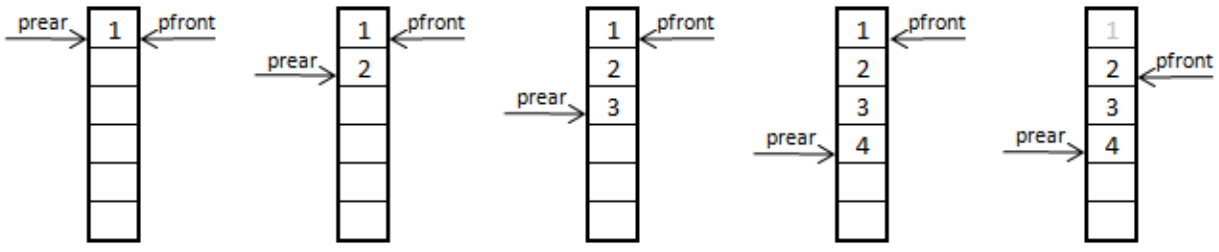
Самой естественной аналогией для очереди является обычная очередь в кассу магазина: самым первым оплатит покупки тот, кто первым встал в очередь.

Для работы с очередями принято использовать следующие процедуры и функции:

- ClearQueue - процедура очистки очереди;
- IsFull - возвращает значение true, если очередь заполнена, и false в противном случае;
- enqueue - добавление элемента в очередь;
- IsEmpty - возвращает значение true, если очередь пуста, и false в противном случае;
- front - возвращает элемент, хранящийся в переднем конце очереди;
- dequeue - возвращает элемент, хранящийся в переднем конце очереди, и удаляет его.

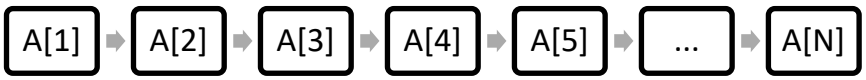
Реализация очереди на основе кольцевого массива

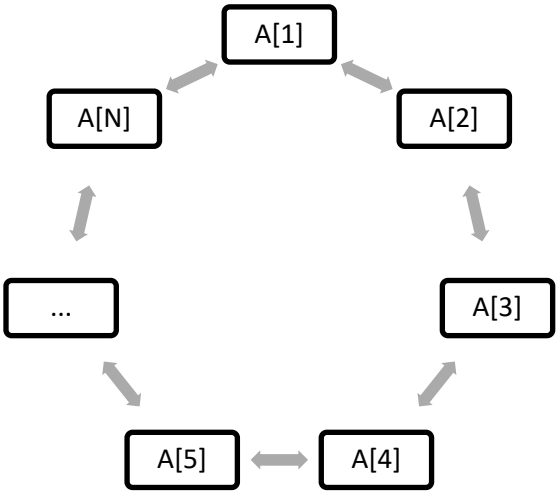
Очевидным решением для создания очереди является использование массива. Тогда при каждом добавлении элементы будут вставляться в конец массива (prear), а удаление будет происходить с первой позиции. Основным недостатком данной реализации является необходимость сдвигать весь массив на одну позицию при удалении элемента. Обойти это узкое место можно, введя второй указатель на первый элемент очереди pfront. На рисунке показано положение указателей при последовательном добавлении в очередь элементов $1,2,3,4$ и удалении из очереди самого старого элемента.



Использовании двух индексов (для начала и конца очереди) позволяет избавиться от необходимости сдвигать массив при каждом удалении, но возникает другая проблема: в очереди, представленной на рисунке, есть ещё три свободные ячейки, но если к prear добавить 3, то произойдёт выход за границы массива, что приведёт к ошибке. Самым распространённым

способом борьбы с этой проблемой является использование кольцевых массивов. В кольцевых массивах при выходе индекса за пределы массива происходит сброс индекса в начальную позицию, т.е. массив оказывается как бы закольцован.





Вычислить индекс в кольцевом массиве с количеством элементов равным MaxQueueLength можно с помощью следующего простого фрагмента кода:

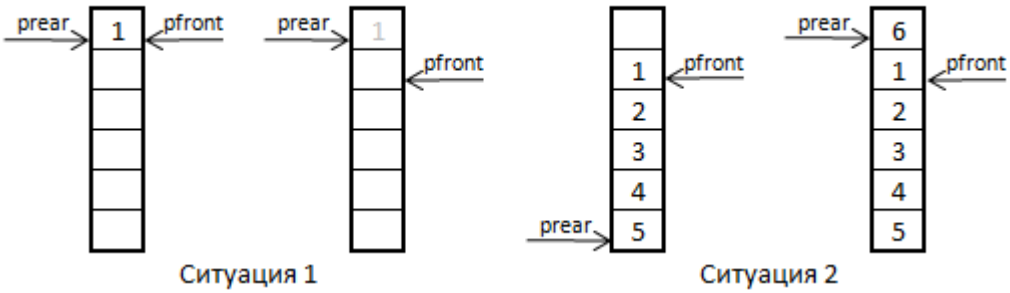
```
if i=MaxQueueLength then
  i:=1
else
  i:=i+1;
```

Этот код можно записать более компактно с использованием операции вычисления остатка:

```
i:=(i+1) mod MaxQueueLength;
```

Прежде чем переходить к кодированию очередей, необходимо обсудить ещё один важный вопрос. Рассмотрим две ситуации:

1. Очередь, состоящая из одного элемента, из которой мы удаляем элемент;
2. Очередь состоящая из MaxQueueLength-1 элементов, куда мы добавляем ещё один элемент.



По рисункам видно, что с помощью одних только индексов pfront и prear невозможно отличить пустую очередь от полной. Существует несколько способов решения этой проблемы:

- Самый простой способ заключается в том, чтобы добавлять в очередь не более MaxQueueLength -1 элементов. Тогда очередь будет полной, если prear находится за две позиции до pfront.
- Другой способ заключается в ведении дополнительной переменной, указывающей либо на полноту очереди, либо на количество элементов в ней.

Рассмотрим реализацию очереди, в которой используется дополнительная переменная, хранящие количество элементов.

С учётом всех вспомогательных элементов объявление очереди будет выглядеть так:

```
const
  MaxQueueLength = 100;
type
  TQueue = record
    pfront,prear: word;
    elems: array [1..MaxQueueLength] of integer;
    elemscount: word;
  end;
```

Дополнительно объявим функцию, которая будет возвращать следующий индекс в кольцевом массиве:

```
function GetNextInd(ind: word): word;
begin
  GetNextInd:=(ind mod MaxQueueLength)+1;
end;
```

Тогда процедуры для работы с очередью будут выглядеть следующим образом:

Очистка очереди (ClearQueue)

При очистке очереди мы должны, во-первых, сбросить в ноль количество элементов, а индексы конца и начала очереди установить в 0 и 1 соответственно.

```
procedure ClearQueue;
begin
  Q.elemscount:=0;
  Q.prear:=0;
  Q.pfront:=1;
end;
```

Проверка заполненность и пустоту (IsFull, IsEmpty)

Так как было решено использовать дополнительную переменную для хранения количества элементов, проверять критерии заполненности и пустоты очереди можно, используя эту переменную. Очередь полна, если в ней MaxQueueLength элементов, и пуста, если в ней 0 элементов.

```
function IsFull: boolean;
begin
  IsFull:=Q.elemscount=MaxQueueLength;
end;
function IsEmpty: boolean;
begin
  IsEmpty:=Q.elemscount=0;
end;
```

Добавление элемента в очередь (enqueue)

При добавлении элемента в очередь, сдвигается указатель prear, и увеличивается количество элементов в очереди. Новый элемент при этом записывается в ячейку с индексом prear.

```
procedure enqueue(newelem: integer);
begin
  Q.prear:=GetNextInd(Q.prear);
  Q.elems[Q.prear]:=newelem;
  Q.elemscount:=Q.elemscount+1;
end;
```

Чтение элемента из очереди (front)

Функция front возвращает первый элемент очереди, но не удаляет его. Индекс первого элемента очереди хранится в переменной pfront.

```
function front: integer;
begin
  front:=Q.elems[Q.pfront];
end;
```

Чтение и удаление элемента из очереди (dequeue)

Удаление элемента из очереди приводит к перемещению указателя pfront и уменьшению количества элементов, хранящихся в очереди.

```
function dequeue: integer;
begin
  dequeue:=Q.elems[q.pfront];
  Q.pfront:=GetNextInd(Q.pfront);
  Q.elemscount:=Q.elemscount-1;
end;
```

Как и в случае со стеком, сложность процедур для работы с очередями не зависит от количества хранящихся элементов, и оценивается величиной $O(C)$. Тем не менее очереди в общем случае работают медленнее стеков, т.к. тратится дополнительное время на редактирование счётчика элементов, и процедура вычисления индексов содержит медленную операцию получения остатка от деления.

Ограничения на применение процедур и функций такие же, как и в случае стека: нельзя вызывать процедуру удаления для пустой очереди и процедура добавления элементов для полной.

Особые типы очередей

Приоритетные очереди

Приоритетная очередь - специализированная структура данных, построенная на основе обычной очереди. Главным отличием является то, что элементы извлекаются не в порядке добавления, а в порядке увеличения (или уменьшения) значений элементов.

При реализации приоритетной очереди на основе массивов существует два основных подхода:

- Элементы вставляются так, чтобы сохранить упорядоченность очереди. При этом извлечение выполняется за константное время.
- Элементы вставляются в конец очереди. При этом для извлечения минимального элемента необходимо просмотреть всю очередь целиком, а потом сдвинуть элементы на освободившееся место.

В зависимости от нужд программиста может быть выбран как первый, так и второй подход. Кроме того, приоритетную очередь можно построить на основе структуры, называемой кучей, которая будет рассмотрена далее.

Многопоточные очереди

Ещё одним интересным типом очередей являются многопоточные очереди. Они используются, когда есть несколько обработчиков элементов очереди. Например, когда есть список процессов, которые должны быть выполнены, и несколько процессоров, которые могут эти процессы выполнять.

Хорошей аналогией из реального мира является очередь в банке. Все клиенты ждут в одной очереди, а сотрудники банка забирают клиентов из общей очереди по мере освобождения. Такая организация эффективнее отдельных очередей, т.к. все сотрудники заняты, пока в очереди есть клиенты.

Многопоточную очередь очень легко построить на основе обычной. Все элементы записываются в одну очередь, а потом каждый освободившийся обработчик извлекает из очереди очередной элемент.

Рассмотренные выше структуры стеки, очереди, приоритетные и многопоточные очереди могут быть реализованы не только на основе смежных структур (массивов), но и на основе связных структур (списков). Такие варианты реализации будут рассмотрены далее.

Реализация стеков и очередей на основе списков

Напомним, что стек - это абстрактный тип данных, добавление и удаление элементов в которой возможно только с одного конца. Легко заметить, что односвязный список практически идеально подходит для организации стека (см. процедуры InsertFirst, DeleteFirst). Преимуществом по сравнению с массивом является теоретическая бесконечность стека, т.к. для списков нет ограничения на количество хранящихся элементов. Недостатком является необходимость хранить указатели, а значит, менее эффективное использование памяти.

Очередь, как и стек, это АТД. В отличие от стека, где элементы добавляются и удаляются с одного и того же конца, в очереди элементы удаляются из одного конца, а добавляются в другой конец. Легко заметить, что очереди могут быть легко организованы на основе двусвязных списков. Добавление элементов организуется с помощью функции InsertAfter(DLF,<>), а удаление с помощью процедуры Delete(DLL^.prev). Такая организация удобнее массива, т.к. позволяет избежать нетривиальных алгоритмов работы с циклическими массивами. Традиционным плюсом является отсутствие ограничения на размер очереди, а минусом - дополнительный расход памяти на хранения ссылок.