# Лекция 23

## Тема:Структуры данных

## *Стеки. Абстрактный тип данных "стек"*

Стек - это специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом вершиной (TOP или HEAD).

***Стеком*** называется совокупность однотипных элементов, над которой определены две основных операции:

- занесение, или заталкивание в стек (**`push`**);
- извлечение из стека (**`pop`**). При этом извлекается тот элемент, который был занесен в стек последним. В соответствии с правилами этой операции стек еще называют структурой типа LIFO ("last in – first out").

В классическом стеке недопустимы никакие другие операции, кроме **push** и **pop**. В частности, доступ к хранящимся в стеке элементам возможет только после их извлечения из стека.

Число хранящихся в стеке элементов также нельзя определить. Вместо этого можно обрабатывать нештатную ситуацию, возникающую при выполнении операции **pop** – пустоту стека.

Физическая реализация стека может быть организована по-разному.

*Первый способ* организации основан на организации стека в виде списка, тогда операции **push** и **pop** сводятся к вставке элемента в начало списка и удалению первого элемента.

Однако этот способ, несмотря на кажущуюся простоту, обладает существенными недостатками:

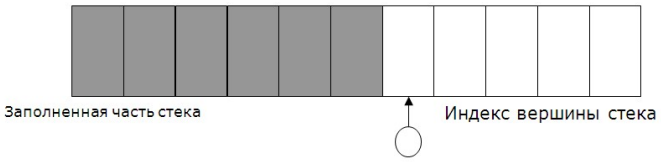
- требуется дополнительная память на хранение ссылок, хотя их значение изменяется редко;
- операции вставки и удаления являются довольно трудоемкими.

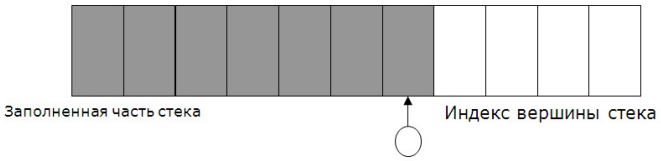
*Второй способ* организации стека предполагает использование массива, в котором будут храниться элементы стека. Дополнительно (как и в случае реализации списка в виде массива) используется целочисленная переменная – *вершина* стека. Значение, хранящееся в ней, соответствует либо индексу последнего занесенного в стек элемента, либо индексу первого свободного элемента.

В любом случае при занесении в стек значение вершины увеличивается, а при извлечении из стека – уменьшается (естественно, после проверки на пустоту стека). В дальнейшем будем использовать в качестве значения вершины индекс первого свободного места.

Этот способ также не лишен недостатков.

# Варианты реализации стека на массивах




Во-первых, необходимо ограничить количество хранящихся в стеке элементов, а при попытке вставки проверять еще одну нештатную ситуацию – переполнение стека.

Во-вторых, организация массива в статической памяти требует, чтобы это количество было определено в момент написания программы, а не в момент ее выполнения.

Второй недостаток в принципе может быть устранен за счет использования динамической памяти.

***Программная реализация стека***

```
int *stack, Top  = 0;
…
stack = new int[N];
// push – заносится значение k
if (Top > N-1)
      throw "Stack is full!";
stack[Top++] = k;
// pop – извлекается значение в k
if (Top == 0)
      throw "Stack is empty!";
k = stack[--Top];
```

Рассмотрим в качестве примера классическую задачу, использующую принципы работы со стеками (хотя термин «стек» в условии не присутствует и даже не подразумевается!).

***Пример.*** *Входная строка содержит алгебраическое выражение, включающее круглые, квадратные и фигурные скобки. Будем считать, что скобки расставлены правильно, если:*

- *каждой открывающей скобке конкретного типа соответствует следующая за ней закрывающая скобка того же типа;*
- *нет закрывающих скобок любого типа без предшествующих им открывающих скобок того же типа;*
- *не допускается рассогласование типов скобок, когда открывающей скобке одного типа соответствует закрывающая скобка другого типа.*

*Требуется проверить правильность расстановки скобок во входной строке.*

## Польская инверсная запись

***История.*** Польская инверсная запись (обратная польская нотация) была разработана австралийским философом и специалистом в области теории вычислительных машин Чарльзом Хэмблином в середине 1950-х на основе польской нотации, которая была предложена в 1920 году польским математиком Яном Лукасевичем. Работа Хэмблина была представлена в июне 1957 года.

Справка из «Википедии»

### *Формы записи бинарных операций*

Пусть задана бинарная операция ◊ над операндами A и B.

- *инфиксная (обычная) форма* A ◊ B
- *префиксная форма* ◊ A B
- *постфиксная форма (польская инверсная запись)* A B ◊

### *Формы записи унарных операций*

Пусть задана унарная операция ◊ над операндом A.

- *префиксная (обычная) форма* ◊ A
- *постфиксная форма (польская инверсная запись)* A ◊

Инфиксная ((A+B*(D-E))/(F+G)
Префиксная /+A*B-DE+FG
Постфиксная ABDE-*+FG+/

### *Особенности постфиксной записи*

- Порядок выполнения операций однозначно задаётся порядком следования знаков операций в выражении, поэтому отпадает необходимость использования скобок и введения приоритетов операций.
- Выражение читается слева направо. Когда в выражении встречается знак операции, выполняется соответствующая операция над двумя последними встретившимися перед ним операндами в порядке их

записи. Результат операции заменяет в выражении последовательность её операндов и её знак, после чего выражение вычисляется дальше по тому же правилу.

- Результатом вычисления выражения становится результат последней вычисленной операции.
- В отличие от инфиксной записи, невозможно использовать одни и те же знаки для записи унарных и бинарных операций.

***Примеры записи выражений в постфиксной форме***

См. конспект

***Задачи, возникающие при работе с польской инверсной записью***

- перевод выражения из инфиксной формы в постфиксную
- вычисление выражения, записанного в постфиксной форме

***Алгоритм перевода выражений в постфиксную форму***

1. Операнды (обозначены буквами) записываются в строку том же порядке, в каком встречаются в исходном выражении.

2. Знак арифметической операции заносится в стек при условии, что приоритет данной операции выше приоритета предыдущей операции. В противном случае в строку записывается предыдущий символ операции, а найденный заносится в стек.

3. Открывающая скобка заносится в стек. Считается, что ее приоритет ниже приоритета всех арифметических операций.

4. При нахождении закрывающей скобки все содержимое стека до первой открывающей скобки удаляется и записывается в строку.

Открывающая скобка удаляется из стека и в строку не записывается.

***Алгоритм перевода выражений в постфиксную форму (псевдокод)***

**(для случая односимвольных операндов)**

```
// Стек и выходная строка изначально пусты
while (входная строка не закончилась) {
  с = очередной символ входной строки;
  switch (с) of {
   операнд: переносим с в выходную строку;
        break;
   '(': заносим с в стек;
        break;
   ')': переносим из стека в выходную строку все
          символы, вплоть до '(';
        открывающую скобку выталкиваем из стека;
        break;
```

```
знак операции:
    if (стек пуст)
      заносим с в стек;
   else {
     do {
      извлекаем из стека символ t;
      if ((t == '(') ||
            (приоритет(с) > приоритет(t)){
        заносим t в стек;
        break;
      }
      else
        переносим t в выходную строку;
     } while (стек не пуст);
     заносим с в стек;
   }  // else
 } // switch
} // while
переносим оставшиеся в стеке символы с выходную строку,
извлекая их из стека по одному;
```

***Пример работы алгоритма***

*для выражения* (a+b)/((c+d)*(e-f))

см. конспект …

***Алгоритм вычисления выражения, находящегося в постфиксной форме***

*Предполагается, что операнды- односимвольные, а их значения известны*

```
// Стек изначально пуст
while (входная строка не закончилась) {
  с = очередной символ входной строки;
  switch (с) of {
   операнд: заносим с в стек; break;
   знак операции:
      извлекаем из стека нужное для выполнения
      операции количество операндов;
      вычисляем значение операции;
      заносим его в стек;
  } // switch
} // while
извлекаем из стека полученное значение выражения;
```

***Упражнения***

- Реализовать стек с помощью указателей.
- Написать программу сортировки вагонов.

***Задача сортировки вагонов***

*Имеется **2n** вагонов (**n** черных и **n** белых). Нужно составить состав так, чтобы вагоны чередовались. Можно использовать операции **ВТупик**, **ИзТупика**, **Мимо**. В тупике может поместиться **n** вагонов*

***Абстрактный тип данных "очередь"***

Очередь - это специальный тип списка, в котором все вставки выполняются в одном конце, называемом последним (**REAR**),
а удаления выполняются в другом конце, называемом передним (**FRONT**).

***Очередь***

Это структура данных, представляющая собой совокупность однотипных элементов, над которой определены две основных операции:

- Вставка в очередь;
- Извлечение из очереди. При этом извлекается тот элемент, который был первым вставлен в очередь. В соответствии с правилами этой операции очередь еще называют структурой типа FIFO ("first in – first out").

Никакие другие операции над классической очередью недопустимы.

Из механизма FIFO следует, что в очереди доступны два элемента – первый и последний.

Физическая организация очереди может быть различная:

Первый способ – в виде списков с указателями на первый и последний элемент списка.

Другой способ организации очереди связан с хранением ее элементов в виде массива. При этом нам надо дополнительно хранить еще два индекса – т.н. ***голову*** и ***хвост*** очереди.

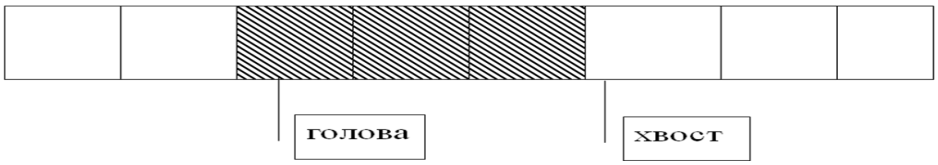
Как и в случае со стеками, в этих переменных можно хранить как индекс занятого, так и индекс свободного элемента. Однако наиболее удобно хранить в качестве хвоста очереди индекс свободного элемента массива, в который будет вставлен новый элемент очереди, а в качестве головы – индекс занятого элемента, который будет удален из очереди первым.

При работе с очередью необходимо обрабатывать следующие нештатные ситуации:

- очередь пуста, а делается попытка удаления элемента из очереди;
- очередь заполнена, а делается попытка вставить новый элемент;

- Кроме того, необходимо решить проблему сдвига элементов: если при добавлении очередного элемента указатель хвоста очереди станет больше, чем размер массива, а указатель на голову указывает не на 1-ый элемент массива, все элементы передвигаются в начало массива.

***Схематическое изображение линейной очереди***




Чтобы избежать ситуации перемещения элементов используют так называемую ***кольцевую*** или ***циклическую*** очередь: после того, как заполнены элементы массива с бо'льшими номерами, заполнение очереди продолжается с области меньших номеров.

Тогда становится понятным, как различить ситуации ***«очередь полна»*** (свободен единственный элемент) и ***«очередь пуста».***

***Программная реализация циклической очереди***

```
int *Queue,
    Front=0, //голова очереди
  Rear=0,  //хвост очереди
  N;       //размер очереди
…
Queue = new int[N+1]; // Выделение памяти
int k;
//push(заносится значение k)
if ((Rear+1 == Front)||
             ((Rear==N)&&(Front==0))
throw "Queue is full";
Queue[Rear++] = k;
if (Rear > N) Rear = 0;

//pop(значение извлекается в k)
if (Rear == Front)
   throw "Queue is empty";
k = Queue[Front++];
if (Front > N) Front = 0;
```

Если в программе неоднократно выполняются действия над очередью, то красивее описать очередь как структуру и все действия реализовать как функции:

```
struct T {     // тип элементов очереди
   int x;
   int y;
};
struct TQueue // тип циклическая очередь
{
    T* Q;
    int Front;
    int Rear;
    int N;
};
TQueue Qu; // переменная циклическая очередь

// функция создания пустой очереди
void Create_Queue(TQueue &Queue,int count)
{
   Queue.N=count;
   Queue.Q=new T [Queue.N+1];
   Queue.Front=0;
   Queue.Rear=0;
}

void Push(TQueue &Queue,const T &k)
// добавление нового элемента в  очередь
{
   if ((Queue.Rear+1==Queue.Front)||
    ((Queue.Rear==Queue.N)&&(Queue.Front==0)))
   throw "Queue is full!";
   Queue.Q[Queue.Rear++]=k;
   if (Queue.Rear>Queue.N)
      Queue.Rear=0;
}

T Pop(TQueue &Queue)
// извлечение элемента из очереди
{
   if (Queue.Rear == Queue.Front)
     throw "Queue is empty!";
```

```
   T k = Queue.Q[Queue.Front++];
   if (Queue.Front > Queue.N)
     Queue.Front = 0;
   return k;
}
```

***Задача.***Из листа клетчатой бумаги размером NxN удалили некоторые клетки размером 1x1. Определить на сколько связных кусков распадается оставшаяся часть листа, если известно K - количество удаленных клеток и их координаты: $(x_1,y_1)$, $(x_2,y_2)$, …, $(x_k,y_k)$.

Алгоритм есть в книге *Котов В.М. , Мельников О.И. Информатика. Методы алгоритмизации. 10-11 кл. Минск, "Народная асвета", 2000 г.*

Конец лекции