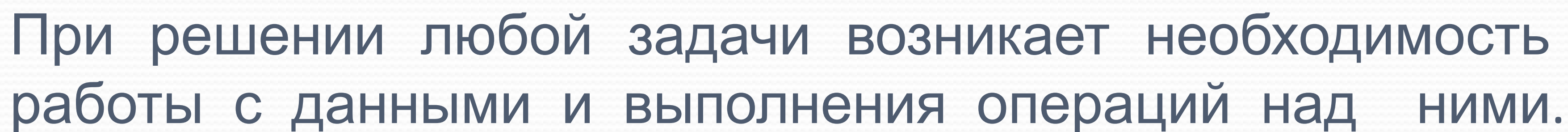
# Структуры данных



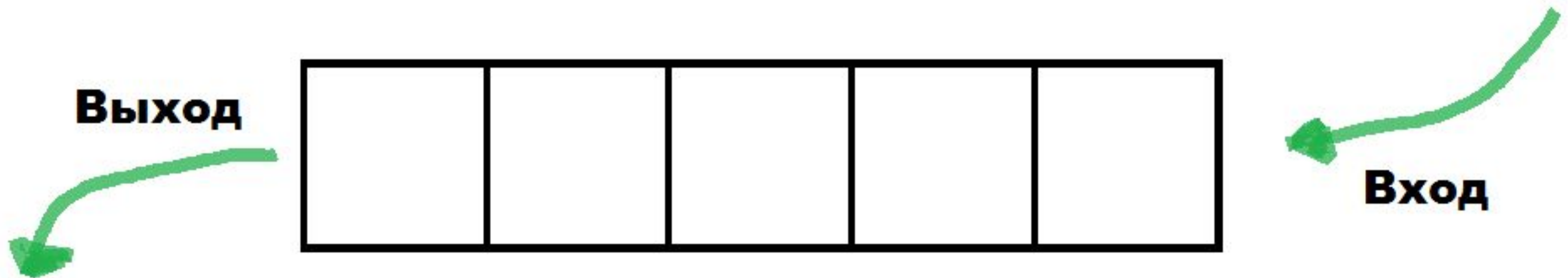
При решении любой задачи возникает необходимость работы с данными и выполнения операций над ними.

Некоторый набор операций часто используется при решении различных задач, поэтому полезно придумать способ организации данных, позволяющий выполнять именно эти операции как можно эффективнее. После того, как такой способ придуман, имеем структуру данных, про которую известно, что в ней хранятся данные некоторого рода, и могут выполняться некоторые операции над этими данными.

Это позволяет отвлечься от деталей и сосредоточиться на характерных особенностях задачи.

# Очередь

Данные обрабатываются в порядке их поступления, принцип FIFO (First In, First Out)



Очередь поддерживает следующие операции:

- queue_init – инициализирует (создает пустую) очередь
- queue_push(x) – добавляет в очередь элемент x
- queue_empty – возвращает true, если очередь пуста (не содержит элементов), и false – в противном случае
- queue_pop – удаляет первый элемент в очереди и возвращает в качестве результата удаленный элемент (предполагается, что очередь не пуста)

## Реализация на базе массива

- Для хранения данных используется массив Q[0..N], где число N достаточно велико.

- Данные всегда хранятся в некотором интервале из последовательных ячеек этого массива. Переменные Head и Tail используются для указания границ этого интервала. Более точно, данные хранятся в ячейках с индексами от Head до Tail(предполагается, что Head < Tail; если же Head = Tail, то очередь пуста).

- Соблюдается также следующее правило: чем позже был добавлен в очередь объект, тем большим будет индекс ячейки массива, в которую он помещается.

Псевдокод операций, работающих с очередью:

```
queue_init {
    Head=0
    Tail=0
}

queue_push(x) {
    Tail++
    Q[Tail]=x
}

queue_empty {
    if Head==Tail
    then return true
    return false
}

queue_pop {
    Head++
    return Q[Head-1]
}
```

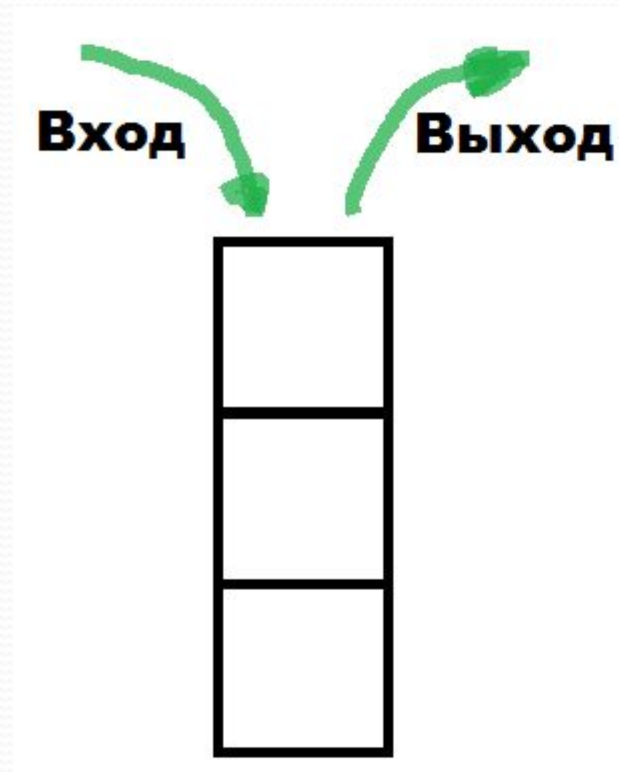
Все операции над очередью при такой реализации работают за O(1), следовательно, такая реализация эффективна по времени.

Однако, число N нужно выбирать достаточно большим (в зависимости от задачи), чтобы избежать «переполнения» очереди, т.е. ситуации, когда Head>N. Это может приводить в некоторых задачах к неэффективному использованию памяти.

Стоит реализовать цикличный доступ к ячейкам массива, чтобы избежать «переполнения» памяти, т.е. при заполнении последнего элемента массива, Tail переместить в начало массива.

# Стек

Данные обрабатываются в обратном порядке их поступления, принцип FILO (First In, Last Out)



Стек поддерживает следующие операции:

- stack_init – инициализирует (создает пустой) стек
- stack_push(x) – добавляет в стек элемент x
- stack_empty – возвращает true, если стек пуст (не содержит элементов), и false – в противном случае
- stack_pop – удаляет верхний элемент стека и возвращает в качестве результата удаленный элемент (предполагается, что стек не пуст)

## Реализация на базе массива

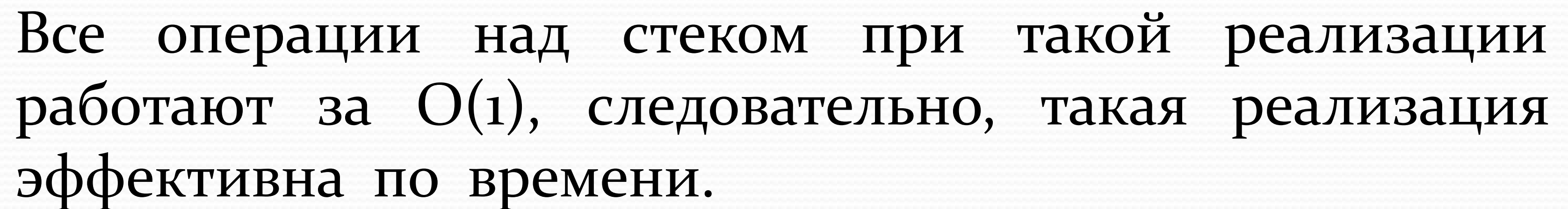
- Для хранения данных используется массив S[0..N], где число N достаточно велико.

- Данные всегда хранятся в некотором интервале из последовательных ячеек этого массива. Переменная Top содержит текущее количество объектов стека. S[Top] – объект, который был добавлен позже всех (предполагается, что Top>-1; если же Top = -1, то очередь пуста).

- Соблюдается также следующее правило: чем позже был добавлен в стек объект, тем большим будет индекс ячейки массива, в которую он помещается.

Псевдокод операций, работающих со стеком:

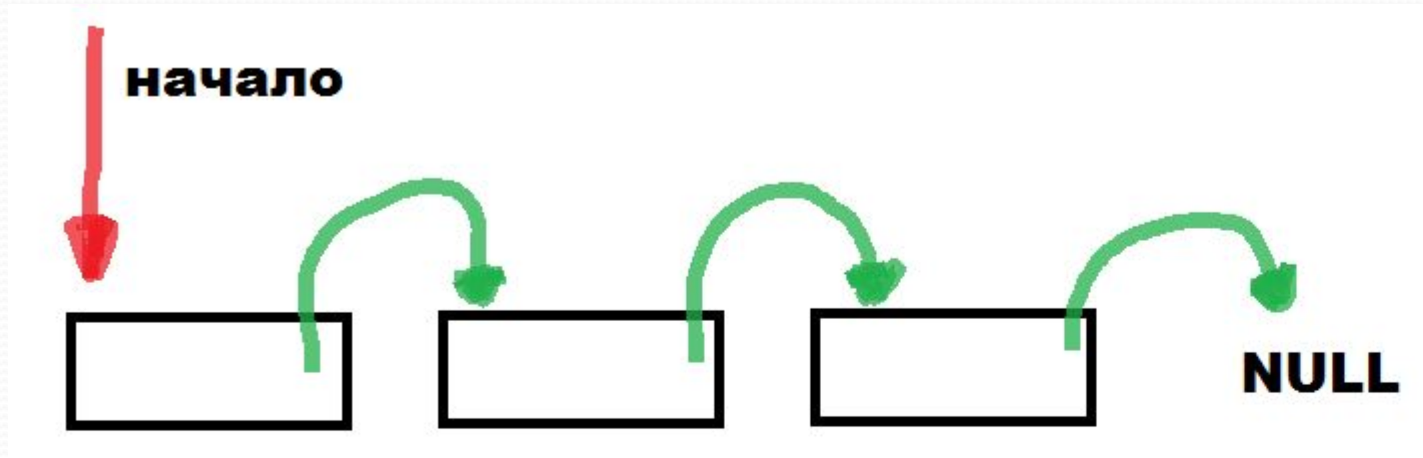
```
stack_init {
    Top=-1
}

stack_push(x) {
    Top++
    S[Top]=x
}

stack_empty {
    if Top==-1
    then return true
    return false
}

stack_pop {
    Top--
    return S[Top+1]
}
```


Все операции над стеком при такой реализации работают за O(1), следовательно, такая реализация эффективна по времени.

Однако, число N нужно выбирать достаточно большим (в зависимости от задачи), чтобы избежать «переполнения» стека, т.е. ситуации, когда Top>N.

# Список

Список – это структура, в которой данные выписаны в некотором порядке. Порядок определяется указателями, связывающими элементы списка в линейную цепочку.



Обычно элемент списка представляет собой запись, содержащую ключ (идентификатор) хранящегося объекта, один или несколько указателей и необходимую информацию об объекте.

Возможна различная связь данных в списке.
Если каждый элемент списка содержит указатель на элемент, следующий непосредственно за ним, то получаемый список называют односвязным.
Если в дополнение к этому каждый элемент списка содержит указатель на элемент, следующий непосредственно перед ним, то список называют двусвязным.

Обычно у последнего элемента списка указатель на следующий элемент равен NULL.

- list_init – инициализирует (создает пустой) список
- list_find(k) – возвращает true, если в списке есть объект с ключом k, и false – в противном случае
- list_insert(x) – добавляет в список объект x
- list_delete(k) – удаляет из списка объект с ключом k

## Реализация односвязного списка

- Каждый объект списка хранится как запись, содержащая следующие поля:
  - Key – ключ объекта
  - Data – дополнительная информация об объекте
  - Next – указатель на следующий объект списка
- Поддерживается указатель Head на первый элемент списка (если список пуст, то Head указывает на NULL)
- Новый элемент вставляется в начало списка
- Последний элемент списка указывает на NULL

Каждый объект может храниться в виде записи:

```
struct myStruct  {
    int Key;
    int Data;
    myStruct*Next;
};
```

Объявление указателя на начало списка:

```
myStruct*Head;
```

Инициализация списка:

```
void list_init() {
    Head = NULL;
}
```

Поиск объекта по ключу:

```
bool list_find(int k) {
    myStruct *x = Head;
    while(x != NULL){
        if(x->Key == k)
            return true;
        x = x->Next;
    }
    return false;
}
```

Операция list_find выполняется за O(n), где n – количество объектов в списке. Это означает, что список не является структурой, эффективно выполняющей поиск.

Вставка объекта в начало списка:

```
void list_insert(myStruct x) {
    myStruct*new_el= new myStruct;
    new_el->Key = x->Key;
    new_el->Data = x->Data;
    new_el->Next = Head;
    Head = new_element;
}
```

Операция list_insert выполняется за O(1).

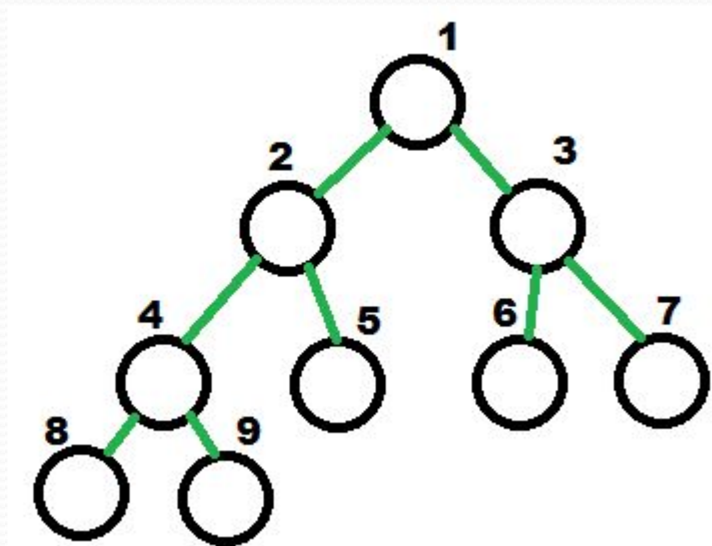
Удаление объекта из списка:

```
void list_delete(int k) {
    myStruct*el = Head;
    while(el != NULL && el->Next != NULL){
        if(el->Next->Key == k) {
            el->Next = el->Next->Next;
            return;
        }
        el = el->Next;
    }
}
```

Операция list_delete выполняется за O(n), т.к. для ее выполнения нужно найти указатель на удаляемый объект списка, для чего придется просмотреть (в худшем случае) весь список.

# Бинарная куча

Будем считать, что объекты хранятся в вершинах двоичного дерева. Пронумеруем вершины этого дерева слева направо сверху вниз



# Свойства:

- Высота двоичного дерева из $N$ вершин (т.е. максимальное количество ребер на пути от корня к листьям) есть $O(\log N)$

Рассмотрим вершину двоичного дерева из $N$ вершин, имеющую номер $i$:

- если $i = 1$, то у вершины $i$ нет отца
- если $i > 1$, то ее отец имеет номер $i / 2$
- если $2i < N$, то у вершины $i$ есть два сына с номерами $2i$ и $2i+1$
- если $2i = N$, то единственный сын вершины $i$ имеет номер $2i$
- если $2i > N$, то у вершины $i$ нет сыновей

Бинарная куча поддерживает следующие операции:

- heap_init – инициализирует бинарную кучу
- heap_minimum – возвращает объект с минимальным значением ключа
- heap_insert(x) – добавляет новый объект
- heap_extract – удаляет объект в корне бинарной кучи и возвращает в качестве результата удаленное значение

## Реализация на базе массива

- Для хранения данных используется массив H[0..N]
- Будем говорить что объекты, хранящиеся в дереве, образуют бинарную кучу, если ключ объекта, находящегося в любой вершине, всегда не превосходит ключей объектов в сыновьях этой вершины.
- В бинарной куче объект H[1] (или объект, хранящийся в корне дерева) имеет минимальное значение ключа из всех объектов.

Рассмотрим операцию heap_insert.
Сначала мы помещаем добавляемый объект на первое свободное место дерева. Если окажется, что ключ этого объекта больше (или равен) ключа его отца, то свойство кучи нигде не нарушено, и добавление проведено корректно. В противном случае, поменяем местами объект с его отцом. В результате вершина с добавляемым объектом «всплывает» на одну позицию вверх.
Это «всплытие» продолжается до тех пор, пока ключ объекта не станет больше (или равен) ключа его отца или пока объект не «всплывет» до самого корня дерева.
Время работы операции прямо пропорционально высоте дерева, оно равно O(log N).

```
heap_insert(x) {
    N++, H[N] = x, i = N
    while (i > 1 && H[i].Key < H[i/2].Key)
        swap(H[i], H[i/2])
        i/=2
}
```

Теперь рассмотрим операцию heap_extract.
Сначала перемещаем объект из листа с номером N в корень. Ставший свободным при этом лист удаляется. Если окажется, что ключ объекта в корне меньше (или равен) ключей объектов в его сыновьях, то свойство кучи нигде не нарушено, и удаление было проведено корректно. В противном случае, выберем сына корня с минимальным значением ключа и поменяем объект в корне с объектом в этом сыне. В результате объект, находившийся в корне, «спускается» на одну позицию вниз. Этот «спуск» продолжается до тех пор, пока объект не окажется в листе или его ключ не станет меньше (или равен) ключей объектов в его сыновьях.
Операция выполняется за O(log N).

```
heap_extract {
    res=H[1], H[1]=H[N], N- -, i=1
    while(2i<N) {
        if(2i<N || H[2i].Key<H[2i+1].Key)
            ind = 2i
        else ind = 2i+1
        if(H[i].Key<=H[ind].Key)
            break
        swap(H[i], H[ind])
    }
}
```

Спасибо за внимание