

Структуры данных

- При решении конкретной задачи можно считать, что у нас в наличии имеется «черный ящик» (его мы и будем называть **структурой данных**), про который известно, что в нем хранятся данные некоторого рода, и который умеет выполнять некоторые операции над этими данными. Это позволяет отвлечься от деталей и сосредоточиться на характерных особенностях задачи.
- Внутри этот «черный ящик» может быть реализован различным образом, при этом следует стремиться к как можно более эффективной (быстрой и экономично расходующей память) реализации.

Очередь (queue)

Очередь — абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл — первый вышел» (FIFO, **First In — First Out**).

Структуру данных «очередь» (queue), удобно применять в ситуации, когда данные нужно обрабатывать в порядке их получения.

- `QUEUE-INIT` – инициализирует (создает пустую) очередь;
- `ENQUEUE (x)` – добавляет в очередь объект `x`;
- `DEQUEUE` – удаляет из очереди объект, который был добавлен раньше всех, и возвращает в качестве результата удаленный объект (предполагается, что очередь не пуста);
- `QUEUE-EMPTY` – возвращает `TRUE`, если очередь пуста (не содержит данных), и `FALSE` – в противном случае.

- Operation QUEUE-INIT
 - Head:= 0;
 - Tail:= 0;
- End;

- Operation ENQUEUE (x)
 - Head:= Head+1;
 - Q[Head]:= x;
- End;

- Operation DEQUEUE
 - Tail:= Tail+1;
 - Return Q[Tail];
- End;

- Operation QUEUE-EMPTY
 - If Tail < Head
 - Then Return FALSE
 - Else Return TRUE;
- End;

- Опишем реализацию очереди на базе массива. Будем использовать для хранения данных массив $Q[0..N]$, где число N достаточно велико. При этом данные всегда будут храниться в некотором интервале из последовательных ячеек этого массива. Мы будем использовать переменные $Head$ и $Tail$ для указания границ этого интервала. Более точно, данные хранятся в ячейках с индексами от $Tail+1$ до $Head$ (предполагается, что $Tail < Head$; 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. Это означает, что объект, который был добавлен в очередь раньше всех – это $Q[Tail+1]$.
- 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, то есть ситуации, когда $Head > N$. Это может приводить в некоторых задачах к неэффективному использованию памяти.

СТЕК

- Стек – список, организованный по принципу LIFO (**Last In, First Out** – "последним вошел, первым вышел").
- Удобно использовать, когда данные нужно обрабатывать в порядке, обратном порядку получения.

- `STACK-INIT` – инициализирует стек;
- `PUSH (x)` – добавляет в стек объект `x`;
- `POP` – удаляет из стека объект, который был добавлен позже всех, и возвращает в качестве результата удаленный объект (предполагается, что стек не пуст);
- `STACK-EMPTY` – возвращает `TRUE`, если стек пуст, и `FALSE` – в противном случае.

- Стек будем реализовывать также на базе массива $S[1..N]$. Данные будем хранить в некотором интервале последовательных ячеек массива (более точно, в ячейках с индексами от 1 до Top). Top – переменная, которая содержит текущее количество объектов в стеке. Как и в случае очереди, соблюдается правило: если $i < j \leq Top$, то объект $S[i]$ был добавлен в стек раньше, чем объект $S[j]$. Это гарантирует, что объект $S[Top]$ – тот объект, который был добавлен в стек позже всех.

- Operation STACK-INIT
 - Top:= 0;
- End;

- Operation PUSH (x)
 - Top:= Top+1;
 - S[Top]:= x;
- End;

- Operation POP
 - Top:= Top-1;
 - Return S[Top+1];
- End;

- Operation STACK-EMPTY
 - If Top > 0
 - Then Return FALSE
 - Else Return TRUE;
- End;

Дек (Deque)

- Дек (deque — double ended queue, «двусторонняя очередь») – структура данных, функционирующая одновременно по двум принципам организации данных: FIFO и LIFO.

Базовые операции

- добавление элемента в начало;
- добавление элемента в конец;
- удаление первого элемента;
- удаление последнего элемента;
- чтение первого элемента;
- чтение последнего элемента.

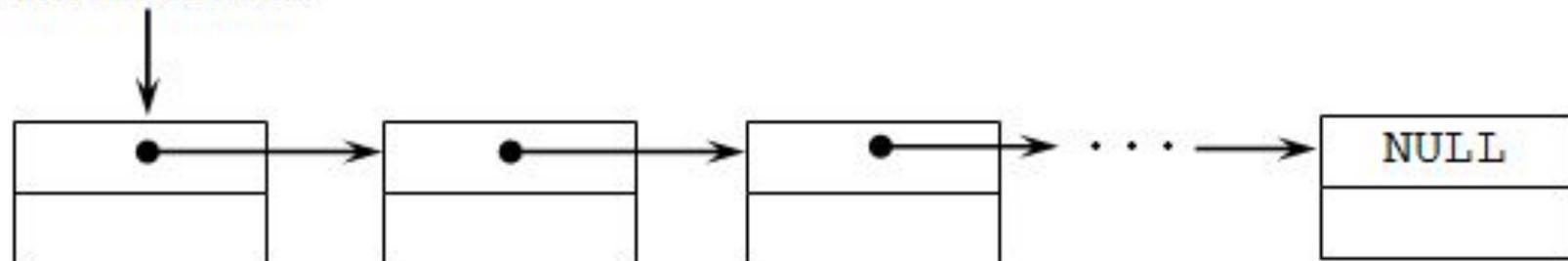
- На практике этот список может быть дополнен проверкой дека на пустоту, получением его размера и некоторыми другими операциями.
- В плане реализации двусторонняя очередь очень близка к стеку и обычной очереди: в качестве ее базиса приемлемо использовать как массив, так и список.

СПИСОК

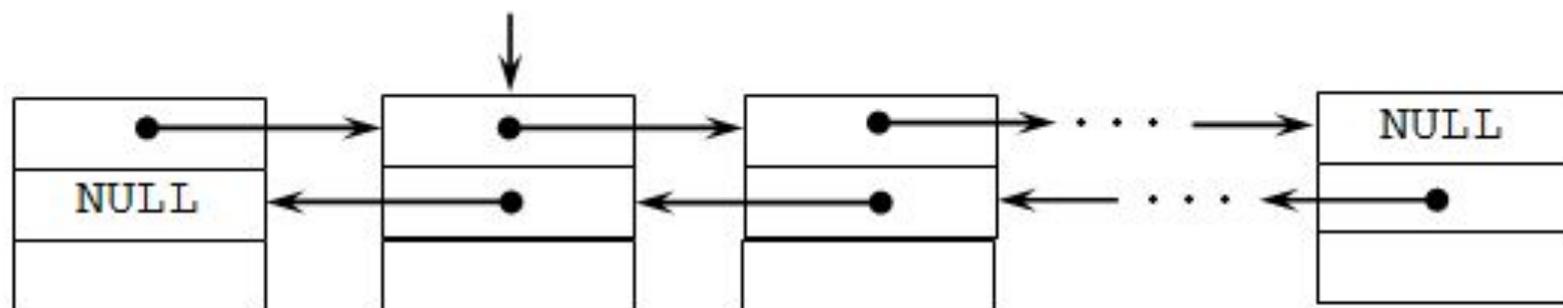
- Список – структура, в которой данные выписаны в некотором порядке. В отличие от массива, этот порядок определяется указателями, связывающими элементы списка в линейную цепочку. Обычно элемент списка представляет собой запись, содержащую **ключ (идентификатор)** хранящегося объекта, один или несколько **указателей** и необходимую **информацию** об объекте. Под ключом подразумевается какая-либо величина, идентифицирующая объект. К примеру, если мы храним информацию о городах, то ключом может быть название города.

- LIST-INIT – инициализирует список;
- LIST-FIND (k) – возвращает TRUE, если в списке есть объект с ключом k, иначе возвращает FALSE;
- LIST-INSERT (obj) – добавляет в список объект obj;
- LIST-DELETE (x) – удаляет из списка элемент x.

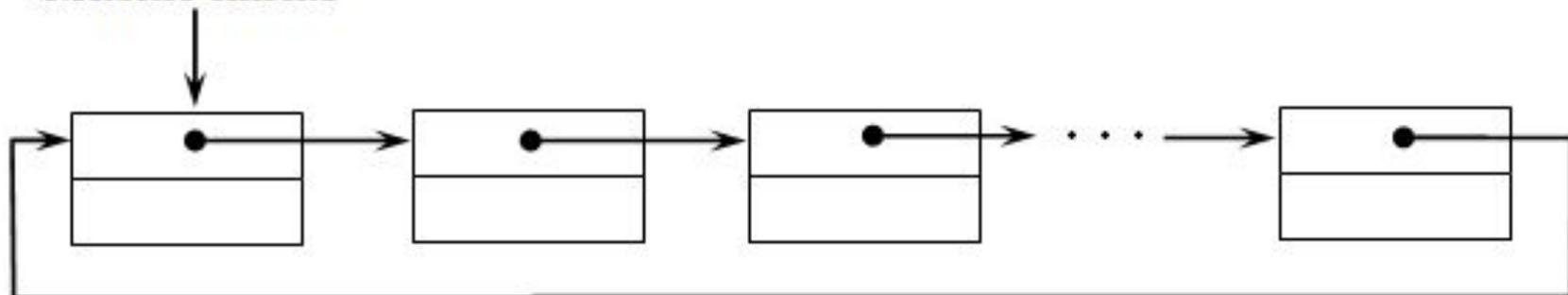
Указатель на первый элемент списка



Указатель на список



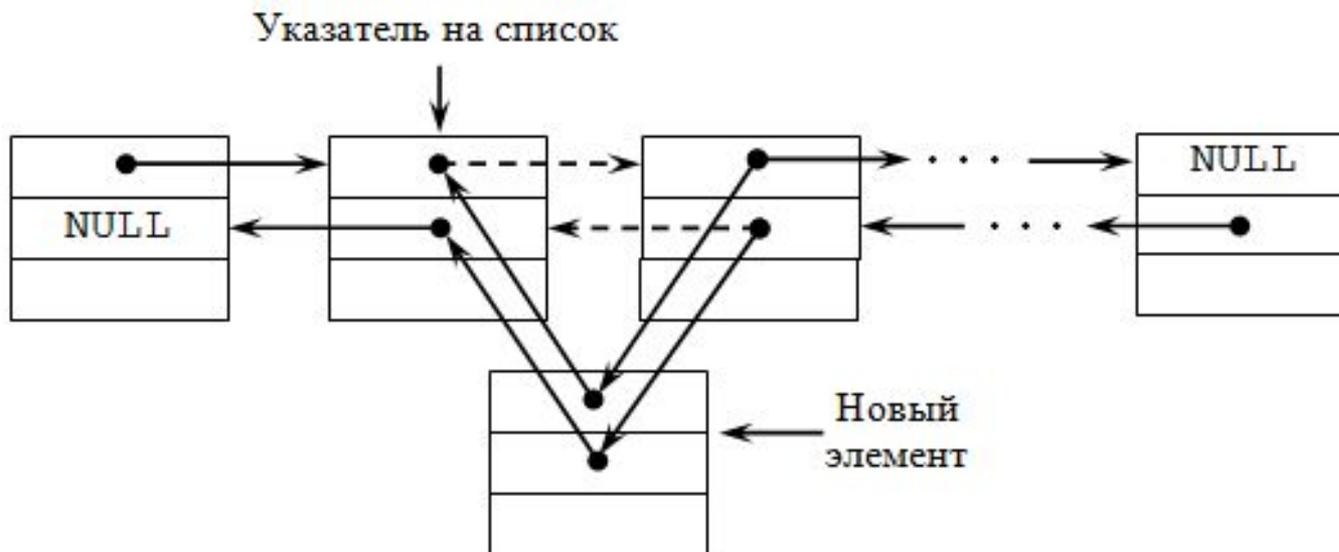
Указатель на «первый» элемент списка



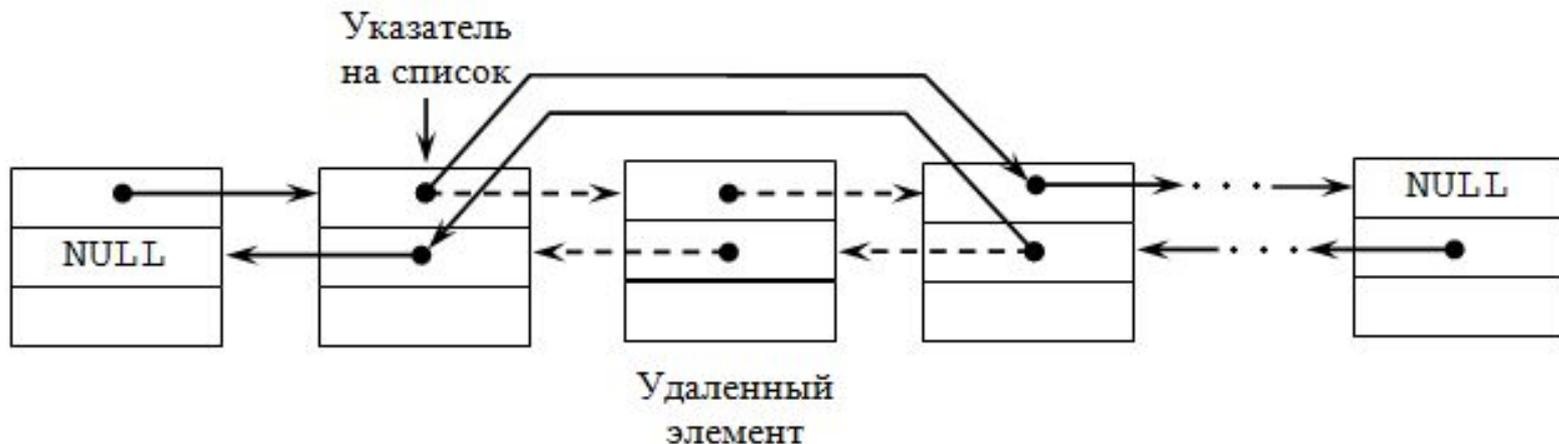
- Опишем реализацию всех операций для двусвязного некольцевого списка. Каждый элемент списка будем хранить как запись, содержащую следующие поля: Key – ключ объекта, Data – дополнительная информация об объекте, Next – указатель на следующий элемент списка (Next = NIL у последнего элемента списка), Prev – указатель на предыдущий элемент списка (Prev = NIL у первого элемента списка). Будем всегда поддерживать указатель Head на первый элемент списка (если список пуст, то Head = NIL). Будем считать, что добавляемые объекты – записи, содержащие поля Key – ключ и Data – дополнительная информация.

- Operation LIST-INIT
 - Head:= NIL;
- End;
- Operation LIST-FIND (k)
 - X:= Head;
 - While X <> NIL Do Begin
 - If X^.Key = k
 - Then Return TRUE;
 - X:= X^.Next;
- End;
- Return FALSE;
- End;
- Операция LIST-FIND выполняется за $O(n)$, где n – количество элементов в списке. Это означает, что список не является структурой, эффективно выполняющей поиск.

- При добавлении элемента в список можно вставить его в начало списка, при этом нужно переписать некоторые указатели.
- Operation LIST-INSERT (obj)
 - New(X);
 - X^.Key:= obj.Key;
 - X^.Data:= obj.Data;
 - X^.Next:= Head;
 - X^.Prev:= NIL;
 - If Head <> NIL
 - Then Head^.Prev:= X;
 - Head:= X;
- End;

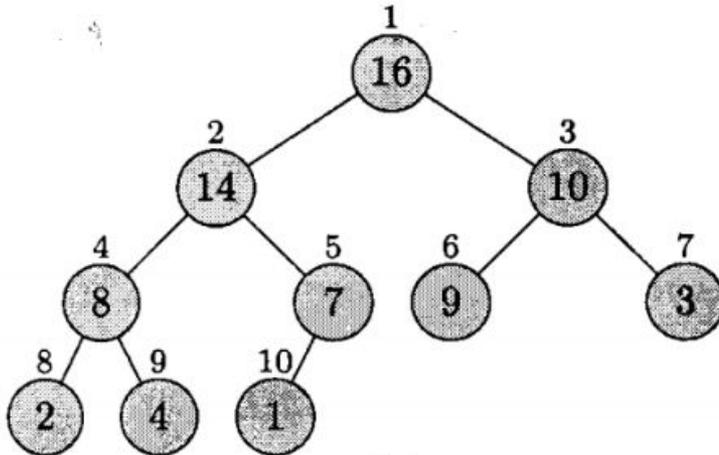


- При удалении элемента из списка нужно соединить указателями предыдущий и следующий за ним элементы. При этом нужно аккуратно обработать случай, когда элемент является первым или последним в списке.
- Operation LIST-DELETE (x)
 - If $x^{\wedge}.Prev \neq NIL$
 - Then $x^{\wedge}.Prev^{\wedge}.Next := x^{\wedge}.Next$
 - Else $Head := x^{\wedge}.Next$;
 - If $x^{\wedge}.Next \neq NIL$
 - Then $x^{\wedge}.Next^{\wedge}.Prev := x^{\wedge}.Prev$;
 - Dispose(x);
- End;

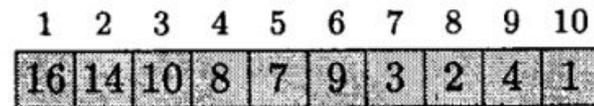


КУЧА

Куча - специализированная структура данных типа дерево, которая удовлетворяет свойству кучи: если B является узлом-потомком узла A , то ключ $(A) \geq$ ключ (B) .



(a)



(б)

Свойство 1. Высота полного двоичного дерева из N вершин (то есть максимальное количество ребер на пути от корня к листьям) есть $O(\log N)$.

Свойство 2. Рассмотрим вершину полного 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 $i \div 2$. 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.

Свойство 3. В бинарной куче объект $H[1]$ (или объект, хранящийся в корне дерева) имеет минимальное значение ключа из всех объектов.

Функции для работы с кучей

`a` – это массив, где хранится наша куча.

`size` – её текущий размер.

```
void shiftDown(int i)
```

```
{
    while (2 * i + 1 < size){
        int l = 2 * i + 1;
        int r = 2 * i + 2;
        int pos = l;
        if (r < size && a[pos] > a[r]){
            pos = r;
        }
        if (a[i] < a[pos]) break;
        swap(a[i], a[pos]);
        i = pos;
    }
}
```

```
void shiftUp(int i)
```

```
{
    while (i > 0){
        int pos = (i - 1) / 2;
        if (a[pos] < a[i]) break;
        swap(a[i], a[pos]);
        i = pos;
    }
}
```

Пирамидальная сортировка $O(n * \log n)$

a – массив, который нужно отсортировать

n – размер массива a

size – размер кучи

```
void HeapSort(int n)
```

```
{  
    size = n;  
    for (int i = 0; i < n / 2; i++)  
        shiftDown(i);  
  
    for (int i = n - 1; i > 0; i--){  
        size = i;  
        swap(a[0], a[i]);  
        shiftDown(0);  
    }  
}
```

Приоритетная очередь

Приоритетная очередь — это абстрактная структура данных на подобии стека или очереди, где у каждого элемента есть приоритет. Элемент с более высоким приоритетом находится перед элементом с более низким приоритетом. Если у элементов одинаковые приоритеты, они располагаются в зависимости от своей позиции в очереди. Обычно приоритетные очереди реализуются с помощью **кучи**.

Методы такие же, как и у обычной очереди. Ниже приведена реализация некоторых из них

```
void push(Type x){
    a[size] = x;
    shiftUp(size);
    size++;
}
```

```
Type pop(){
    Type res = a[0];
    size--;
    swap(a[0], a[size]);
    shiftDown(0);
    return res;
}
```

Дома решить задачи из контеста «Занятие 27.09.2016. Структуры данных» в группе на codeforces.com

Темы задач:

A – стек

B – стек

C – очередь

D – очередь

E – список

F – приоритетная очередь

G – работа с кучей

H – очередь

На основе решения домашних заданий будет составляться рейтинг.
Удачи.