ЛЕКЦИЯ 7

ОСНОВЫ ПРОГРАММИРОВАНИЯ

ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ

- Пользовательские типы данных можно создать с помощью:

- ***структуры*** — группы переменных, имеющей одно имя и называемой *агрегатным* типом данных (*соединение* (*compound*), *конгломерат* (*conglomerate*).);
- ***объединения***, которое позволяет определять один и тот же участок памяти как два или более типов переменных;
- ***битового поля***, которое является специальным типом элемента структуры или объединения, позволяющим легко получать доступ к отдельным битам;
- ***перечисления*** — списка поименованных целых констант;
- ключевого слова **typedef**, которое определяет новое имя для существующего типа.

ОПЕРАТОР TYPEDEF

- Оператор **typedef** определяет новое имя для уже существующего типа.

- Общий вид декларации:
- typedef *type1 type2;*

- *type1* — это любой тип данных языка C++,
- *type2* — новое имя этого типа.

СТРУКТУРЫ

- ***Структура*** — это совокупность переменных, объединенных под одним именем.

- *Объявление структуры* создает шаблон, который можно использовать для создания ее объектов (то есть экземпляров этой структуры).

- Переменные, из которых состоит структура, называются ***членами*** (*элементами*, *полями*), и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него.

STRUCT

- **struct** *тег {*
- *тип имя-члена;*
- *тип имя-члена;*
- *тип имя-члена;*
- .
- .
- .
- *} переменные-структуры;*

- *тег* или *переменные-структуры* могут быть пропущены, но **только не оба одновременно**.

СТРУКТУРЫ

```
struct addr
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
};

struct addr addr_info;
addr_info.zip = 12345;
```

СТРУКТУРЫ

- Доступ к отдельным элементам структуры осуществляется с помощью оператора . (точка)
- *имя-объекта-структуры.имя-элемента*
- addr_info.zip = 191002;
- При обращении через указатель **->**
- pointer_addr_info->zip = 198504;

АНАЛИЗ ПРОГРАММЫ

```
int main()
{
  struct {
    int a;
    int b;
  } x, y, *z;
  x.a = 10;
  y=x;
  z=&x;
  cout<<y.a<<z->a;
  return 0;}
```

АНАЛИЗ ПРОГРАММЫ

- *// объявление массива структур*
- struct addr addr_list[100];

- *// объявление указателя на структуру*
- struct addr *addr_pointer;

- *// использование вложенных структур*
- struct emp {
- struct addr address; */* вложенная структура */*
- float salary;
- } worker;

АНАЛИЗ ПРОГРАММЫ

- struct Phone
- {
- char* name;
- long phoneNumber;
- };
- Phone *somePhone = new Phone;
- somePhone->name = "Иванов Иван";
- somePhone->phoneNumber= 1234567;
-

БИТОВЫЕ ПОЛЯ

- **Битовые поля** — это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет».
- При описании битового поля после имени через двоеточие указывается длина поля в битах.
- Доступ к полю осуществляется по имени. Адрес поля получить нельзя.

БИТОВЫЕ ПОЛЯ

```
struct Options {
    bool centerX:1;
    bool centerY:1;
    unsigned int shadow:2;
    unsigned int palette:4; };

struct rgb_color
{   unsigned red_value:3;
    unsigned green_value:3;
    unsigned blue_value:3; };
```

ОБЪЕДИНЕНИЯ

- Объединение (**union**) представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу.

- В каждый момент времени в переменной типа объединение хранится *только одно значение*.

ОБЪЕДИНЕНИЯ

```
struct Options {
    bool centerX:1;
    bool centerY:1;
    unsigned int shadow:2;
    unsigned int palette:4; };
union {
    unsigned char ch;
    Options bit;
} option = {0xC4};
cout << option.bit.palette;
option.ch &= 0xF0;      // наложение маски
```

ОБЪЕДИНЕНИЯ

- Ограничения объединений (по сравнению со структурами):

- объединение может инициализироваться только значением его первого элемента;
- объединение не может содержать битовые поля;
- объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;
- объединение не может входить в иерархию классов.

ПЕРЕЧИСЛЕНИЯ

- *Перечисление* — это набор именованных целых констант.

- enum *тег* {*список перечисления*}
- *список переменных*;

- *тег* или *переменные-структуры* могут быть пропущены, но **только не оба одновременно**.

- Каждому элементу дается значение, на единицу большее, чем у его предшественника. Первый элемент перечисления имеет значение 0.

ПЕРЕЧИСЛЕНИЯ

- */*
- *penny (пенни, монета в один цент)*
- *nickel (никель, монета в пять центов)*
- *dime (монета в 10 центов)*
- *quarter (25 центов, четверть доллара)*
- *half-dollar (полдоллара)*
- *dollar (доллар) */*
- enum coin { penny, nickel, dime, quarter, half_dollar, dollar};
- enum coin money;
- money = dime;

ПЕРЕЧИСЛЕНИЯ

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;
// ...
switch (error) {
    case ERR_READ:          /* операторы */
        break;
    case ERR_WRITE:         /* операторы */
        break;
    case ERR_CONVERT:       /* операторы */
        break;
}
```

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память выделяется по мере необходимости *отдельными блоками*, связанными друг с другом с помощью *указателей*.

- Такой способ организации данных называется **динамическими структурами данных**, поскольку их размер изменяется во время выполнения программы.

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- ***Механизм доступа*** (data engine) к данным – механизм *сохранения* и *получения* информации.

- **Очередь** (queue)
- **Стек** (stack)
- **Связанный список** (linked list)
- **Двоичное дерево** (binary tree)

- Каждый из этих методов дает возможность решать задачи определенного класса.

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- Динамические структуры данных *(списки, стеки, очереди, деревья)* различаются способами связи отдельных элементов и допустимыми операциями.
- Динамическая структура может занимать несмежные участки оперативной памяти.
- Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения **задач сортировки.**

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- Элемент любой динамической структуры данных представляет собой структуру (**struct**), содержащую по крайней мере два поля: для хранения данных и для указателя.
- Полей данных и указателей может быть несколько.
- Поля данных могут быть любого типа: основного, составного или типа указатель.

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

- Описание простейшего элемента (компоненты, узла) динамической структуры:
-
```
struct Node {
/*    тип данных Data должен быть
определен ранее      */
Data d;
Node *p:
  }:
```

СПИСКИ

- ***Списком*** называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения.
- Список, отражающий отношения соседства между элементами, называется ***линейным***.
- ***Линейные связные списки*** – простейшие динамические структуры данных.
- ***Длина списка*** равна числу элементов, содержащихся в списке.
- Список нулевой длины называется ***пустым списком***.

СПИСКИ

- Самый простой способ связать множество элементов — сделать так, чтобы каждый элемент содержал ссылку на следующий.
- Такой список называется однонаправленным (***односвязным***).
- Если добавить в каждый элемент вторую ссылку — на предыдущий элемент, получится двунаправленный список (***двусвязный***).
- Если последний элемент связать указателем с первым, получится ***кольцевой*** список.

СПИСКИ

- Каждый элемент списка содержит **ключ**, идентифицирующий этот элемент.
- Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных.
- В качестве ключа в процессе работы со списком могут выступать разные части поля данных.
- Ключи разных элементов списка могут совпадать.

СВЯЗНЫЕ ЛИНЕЙНЫЕ СПИСКИ

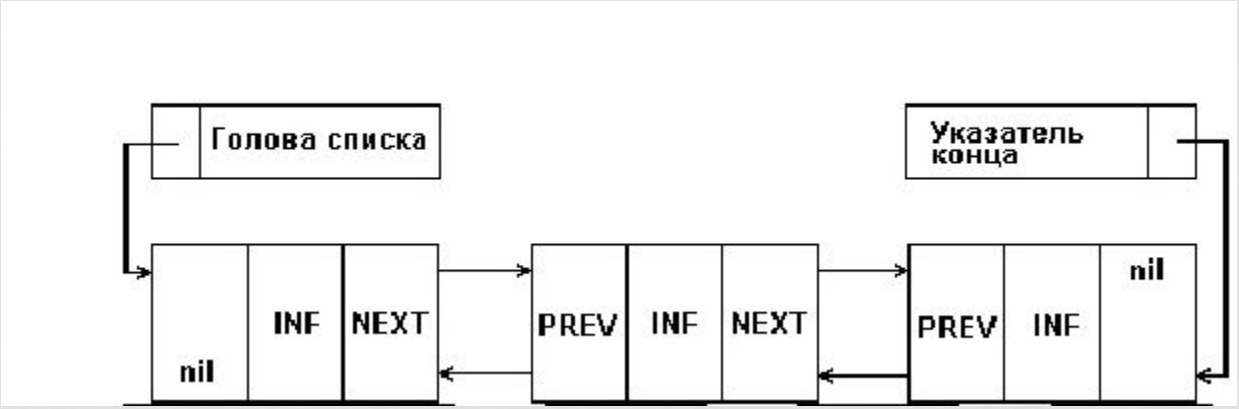
- INF - информационное поле, данные
- NEXT - указатель на следующий элемент списка
- nil - специальный признак, свидетельствующий о конце списка

- **Линейный односвязный список:**

- обработка односвязного списка не всегда удобна, так как отсутствует возможность продвижения в противоположную сторону.

СВЯЗНЫЕ ЛИНЕЙНЫЕ СПИСКИ

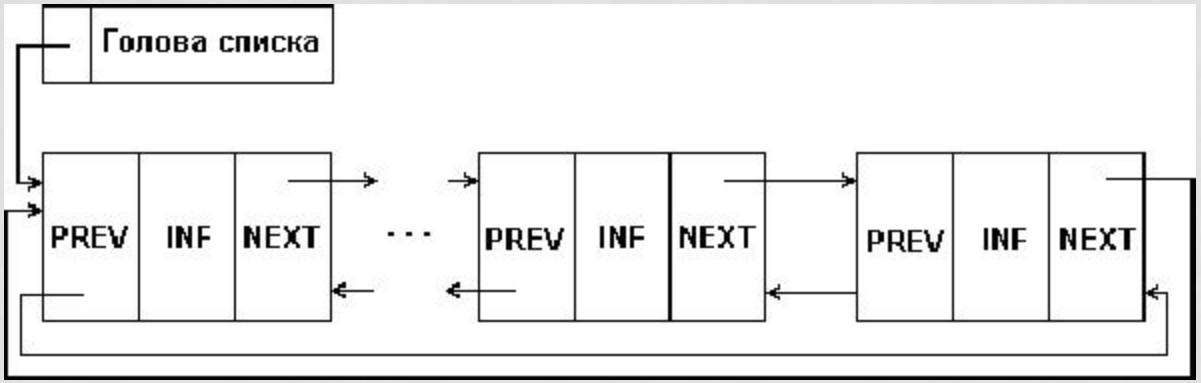
- **Линейный двусвязный список:**



- Наличие двух указателей в каждом элементе усложняет список и приводит к дополнительным затратам памяти, но в то же время обеспечивает более эффективное выполнение некоторых операций над списком.

СВЯЗНЫЕ ЛИНЕЙНЫЕ СПИСКИ

- **Кольцевой список** (может быть организован на основе как односвязного, так и двухсвязного)

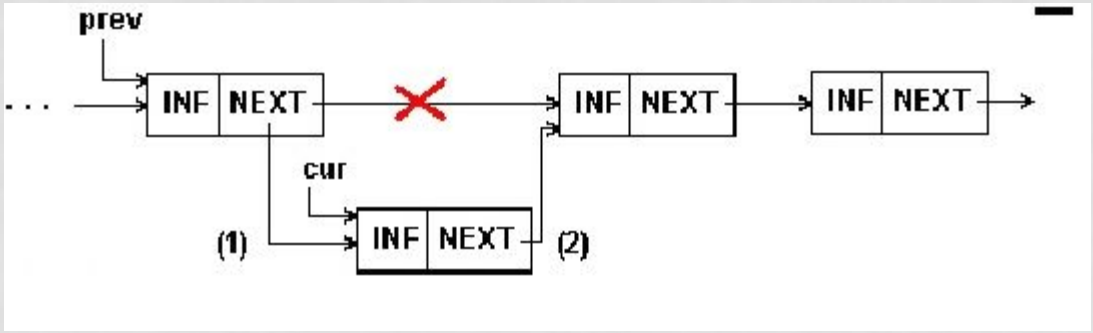


ОПЕРАЦИИ НАД СПИСКАМИ

- начальное формирование списка (создание первого элемента);
- добавление элемента в конец списка;
- чтение элемента с заданным ключом;
- вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- удаление элемента с заданным ключом;
- упорядочивание списка по ключу.

РЕАЛИЗАЦИЯ ОПЕРАЦИЙ НАД ЛИНЕЙНЫМИ СПИСКАМИ

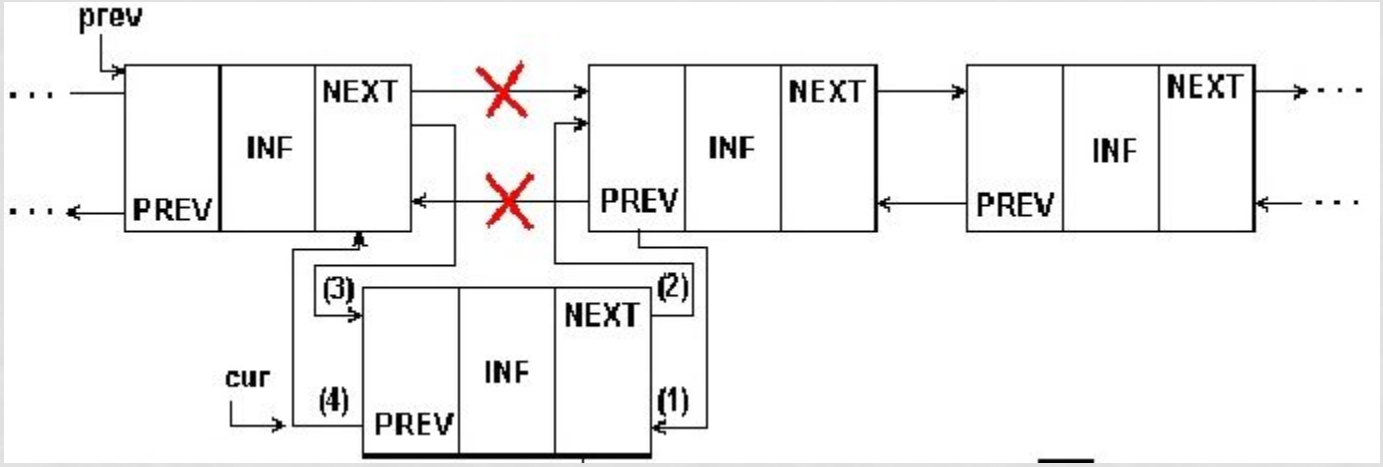
- **Вставка элемента в середину 1-связного списка**



- **Вставка элемента в начало 1-связного списка**

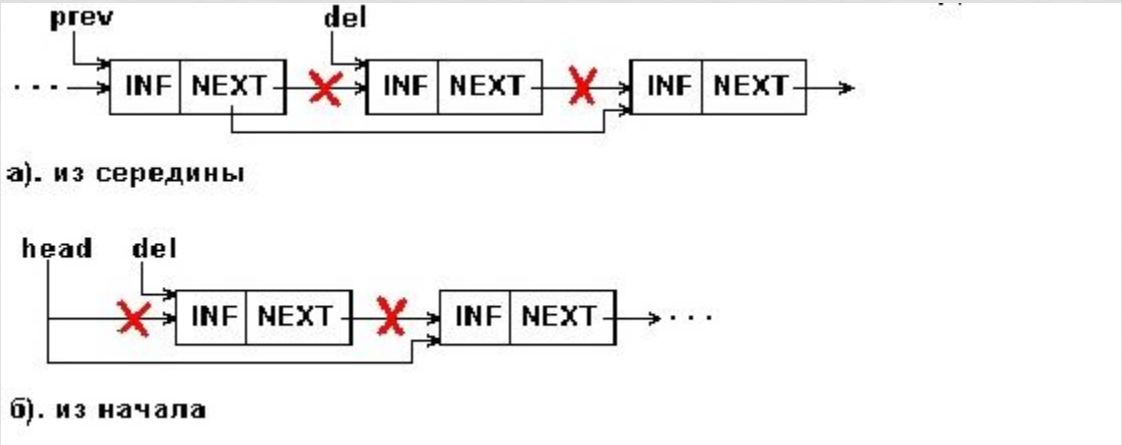
РЕАЛИЗАЦИЯ ОПЕРАЦИЙ НАД ЛИНЕЙНЫМИ СПИСКАМИ

- **Вставка элемента в середину 2-связного списка**

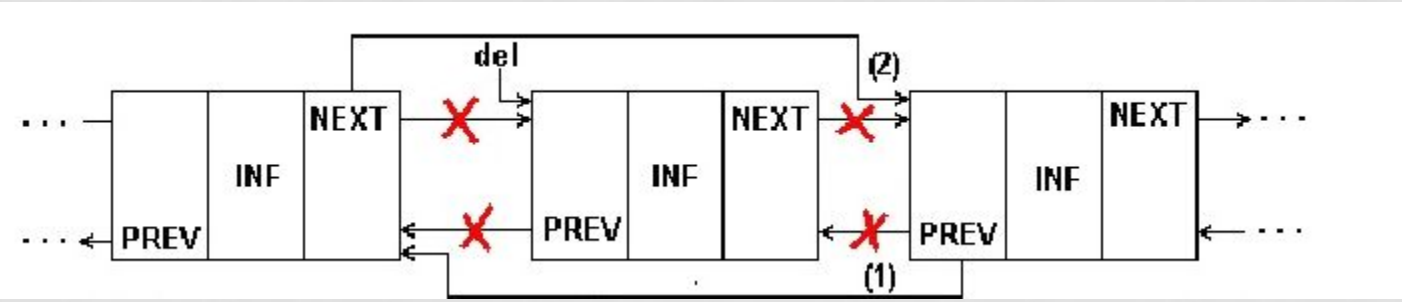


РЕАЛИЗАЦИЯ ОПЕРАЦИЙ НАД ЛИНЕЙНЫМИ СПИСКАМИ

- **Удаление элемента из 1-связного списка**

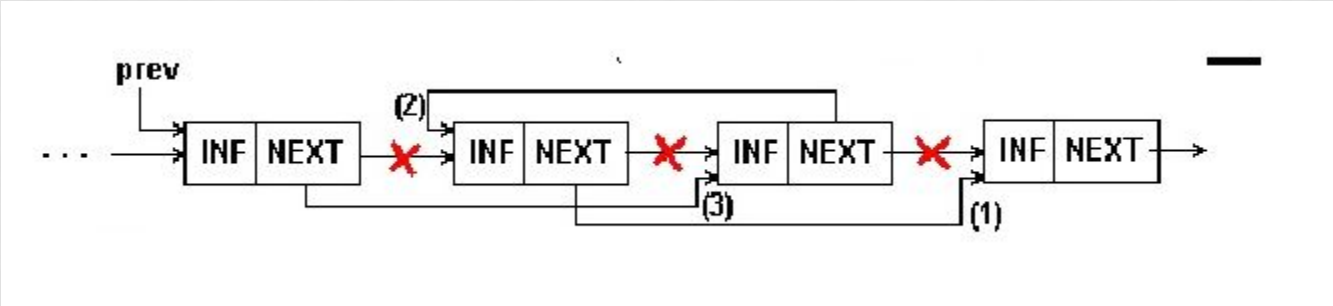


- **Удаление элемента из 2-связного списка**



РЕАЛИЗАЦИЯ ОПЕРАЦИЙ НАД ЛИНЕЙНЫМИ СПИСКАМИ

- **Перестановка элементов 1-связного списка**



- Изменчивость динамических структур данных предполагает не только изменения размера структуры, но и изменения связей между элементами.
- Для связных структур изменение связей не требует пересылки данных в памяти, а только изменения указателей в элементах связной структуры.

ВОЗМОЖНЫЕ ОПЕРАЦИИ НАД ТИПОМ ДАННЫХ «СПИСОК»

- *end(l)* — вернуть позицию, следующую за последним элементом списка l.
- *insert(x, p, l)* — добавить элемент x в позицию p списка l.
- *locate(x, l)* — возвращает позицию элемента x в списке l.
- *retrieve(p, l)* — возвращает значение элемента в позиции p списка l.
- *delete(p, l)* — удаляет элемент в позиции p
- *next(p, l)* и *previous(p, l)*, *makeNull(l)*, *first(l)*, *printList(l)* и т.п.
-

АНАЛИЗ ПРОГРАММЫ

```
//Описание структуры элемента списка
struct Node
{
int d;
Node *next;
Node *prev;
};
```

АНАЛИЗ ПРОГРАММЫ

```
// Формирование первого элемента

Node * first(int d)
{
Node *pv = new Node;
pv->d = d;
pv->next = 0;
pv->prev = 0;
return pv;
};
```

АНАЛИЗ ПРОГРАММЫ

- ***// Добавление в конец списка***

- void add(Node **pend, int d)
- {
- Node *pv = new Node;
- pv->d = d;
- pv->next = 0;
- pv->prev = *pend;
- (*pend)->next = pv;
- *pend = pv;
- }

АНАЛИЗ ПРОГРАММЫ

```
// Поиск элемента по ключу
Node * find(Node * const pbeg, int d)
{
Node *pv = pbeg;
while (pv)
{
if(pv->d == d)
break;
pv = pv->next;
}
return pv;}
```

АНАЛИЗ ПРОГРАММЫ

```
// Вставка элемента
Node * insert(Node * const pbeg, Node **pend, int key, int d)
{ if(Node *pkey = find(pbeg, key)){
Node *pv = new Node; pv->d = d;
// 1 - установление связи нового узла с последующим:
pv->next = pkey->next;
// 2 - установление связи нового узла с предыдущим:
pv->prev = pkey;
// 3 - установление связи предыдущего узла с новым:
pkey->next = pv;
// 4 - установление связи последующего узла с новым:
if( pkey != *pend) (pv->next)->prev = pv;
// Обновление указателя на конец списка,
// если узел вставляется в конец:
else *pend = pv; return pv;
} return 0; }
```

АНАЛИЗ ПРОГРАММЫ

```
// Удаление элемента
bool remove(Node **pbeg, Node **pend, int key)
{       if (Node *pkey = find(*pbeg, key))
{ if (pkey == *pbeg)
{*pbeg = (*pbeg)->next;
  (*pbeg)->prev =0;}
  else
  if (pkey == *pend)
  {     *pend = (*pend)->prev;
        (*pend)->next = 0;}
  else
  {     (pkey->prev)->next = pkey->next;
        (pkey->next)->prev = pkey->prev;}
  delete pkey; return true;}
return false;}
```

СРАВНЕНИЕ ОПЕРАЦИЙ НАД СТРУКТУРАМИ ДАННЫХ

- Сравнение операций над статическими массивами, динамическими массивами и связными списками:

	Linked list	Array	Dynamic array
Indexing	Θ(*n*)	Θ(1)	Θ(1)
Insertion/deletion at end	Θ(1)	N/A	Θ(1)
Insertion/deletion in middle	Θ(1)	N/A	Θ(*n*)
Wasted space (average)	Θ(*n*)	0	Θ(*n*)

ДОСТОИНСТВА СПИСКОВ

- лёгкость добавления и удаления элементов
- размер ограничен только объёмом памяти компьютера и разрядностью указателей
- динамическое добавление и удаление элементов

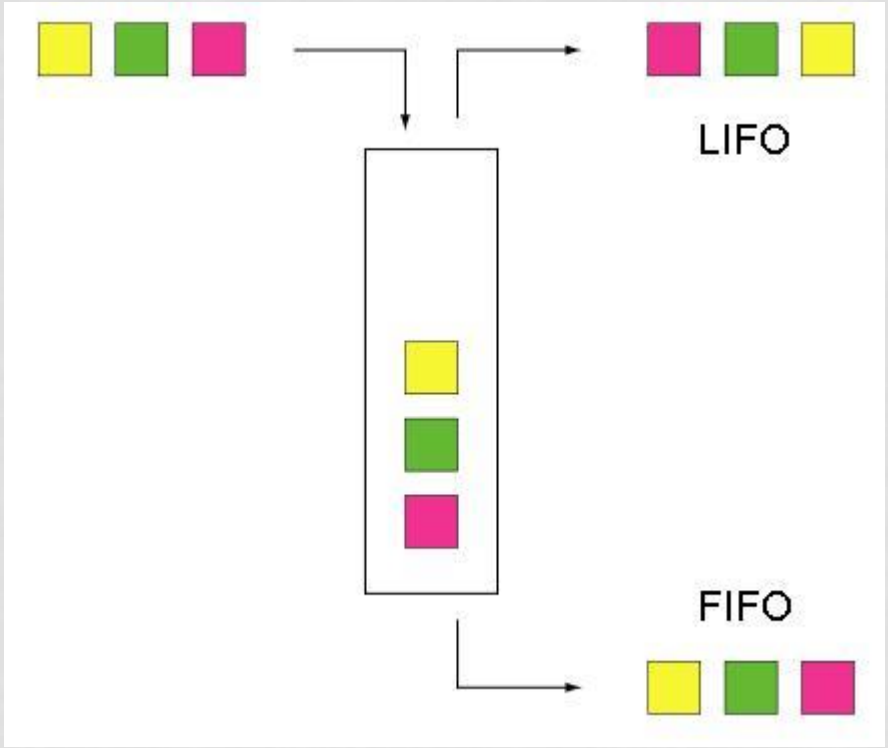
НЕДОСТАТКИ СПИСКОВ

- сложность определения адреса элемента по его индексу (номеру) в списке
- на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны)
- работа со списком медленнее, чем с массивами, так как к любому элементу списка можно обратиться, только пройдя все предшествующие ему элементы
- элементы списка могут быть расположены в памяти разреженно, что окажет негативный эффект на кэширование процессора
- над связными списками гораздо труднее (хотя и в принципе возможно) производить параллельные векторные операции, такие как вычисление суммы

АБСТРАКТНЫЕ ПРИНЦИПЫ ОБРАБОТКИ СПИСКОВ

- **FIFO** (*First In, First Out)*
- *«первым пришёл —*
- *первым ушёл»*

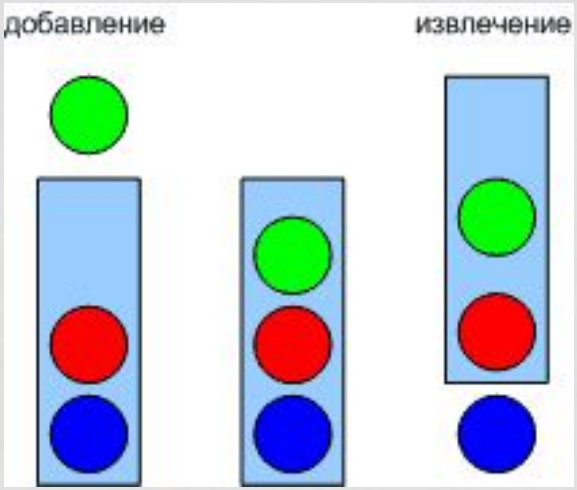
- **FIFO** (*First In, First Out)*
- *«последним пришёл —*
- *первым ушёл»*



ОЧЕРЕДИ

- **Очередь** — это частный случай линейного 1-связного списка, добавление элементов в который выполняется в один конец, а выборка — из другого конца.

- Очередь реализует принцип обслуживания **FIFO**.



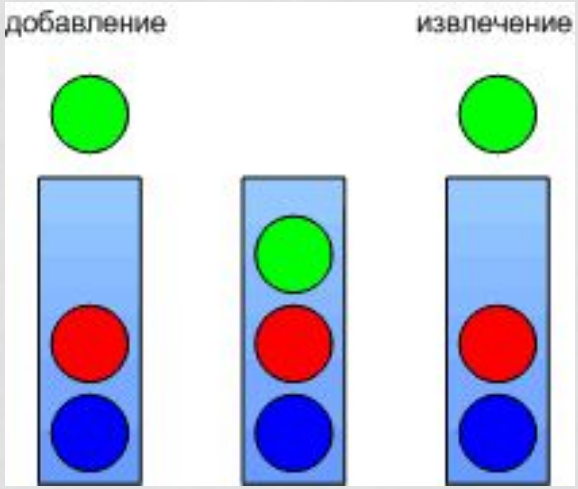
ОЧЕРЕДИ

- **Типичные операции:**

- void **push(**const value_type&**)** - добавить элемент
- void **pop()** - удалить первый элемент
- size_type **size()** - размер очереди
- bool **empty()** - true, если очередь пуста
- value_type& **front()** - получить первый элемент
- value_type& **back()** - получить последний элемент

СТЕКИ

- **Стек** — это частный случай линейного 1-связного списка, добавление элементов в который и извлечение из которого выполняются с одного конца, называемого вершиной стека. При извлечении элемент исключается из стека.
- Стек реализует принцип обслуживания **LIFO**.



СТЕКИ

Типичные операции:

- void **push(**const value_type&**)** - добавить элемент
- void **pop()** - удалить верхний элемент
- value_type& **top()** - получить верхний элемент
- size_type **size()** - размер стека
- bool **empty()** - true, если стек пуст

ЗАДАЧИ (СПИСКИ)

- 1) Написать программу, выводящую элемент списка по его номеру.
- 2) Определить длину списка (вывести длину списка). Список вводится с клавиатуры.
- 3) Переформировать список так, чтобы список стал в обратном порядке.
- 4) По заданному списку посчитать количество каждого из встречаемых в нем элементов.
- 5) В список после каждого вхождения элемента E вставить элемент F.
- 6) Реализовать стек и очередь на списках.