

Программирование и алгоритмизация

Лекция 2

Линейные списки

- Линейные списки – структуры данных, в которых элементы (переменные) следуют друг за другом, и каждому можно поставить в соответствие порядковый номер

Операции с линейными списками

- i) Получение доступа к k -му узлу списка для проверки и/или изменения содержимого его полей.
- ii) Вставка нового узла сразу после или до k -го узла.
- iii) Удаление k -го узла.
- iv) Объединение в одном списке двух (или более) линейных списков.
- v) Разбиение линейного списка на два (или более) списка.
- vi) Создание копии линейного списка.
- vii) Определение количества узлов в списке.
- viii) Сортировка узлов в порядке возрастания значений в определенных полях этих узлов.
- ix) Поиск узла с заданным значением в некотором поле.



$k-1$ k $k+1$

Виды линейных списков

Стек — это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются только на одном из концов списка.

Очередь или *односторонняя очередь* — это линейный список, в котором все операции вставки выполняются на одном из концов списка, а все операции удаления (и, как правило, операции доступа к данным) — на другом.

Дек или *двусторонняя очередь* (double-ended queue) это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются на обоих концах списка.

Примеры применения списков

Линейные списки:

- В операционных системах
 - Очередь задач, готовых к выполнению
 - Очередь документов к принтеру
 - Стек состояний прерванных процессов (задач)
- В системах имитационного моделирования
 - Очередь заявок на обслуживание какой-либо системой массового обслуживания, например,
 - Очередь деталей на обслуживание станком или промышленным роботом

Деревья используются при грамматическом разборе (синтаксическом анализе) искусственных и естественных языков

Сложные списковые структуры используются для представления графов:

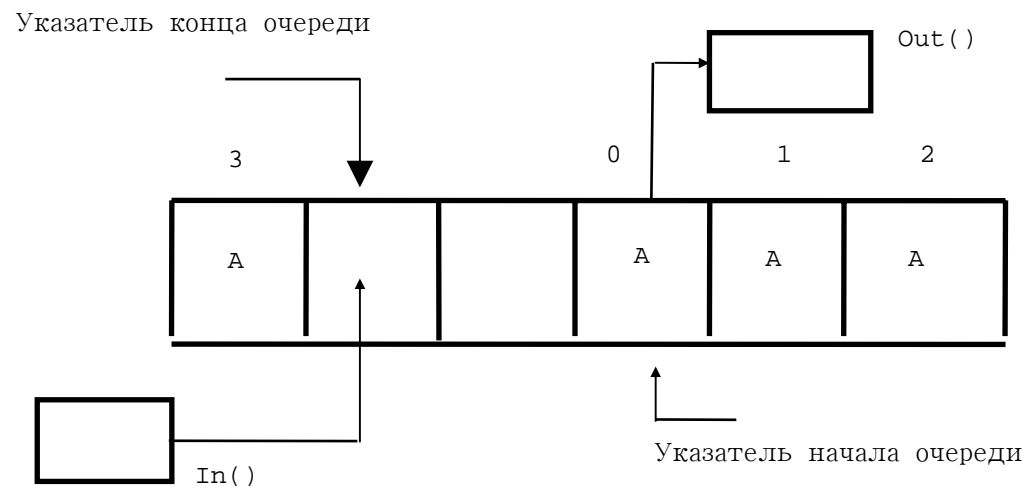
- В навигационных системах роботов или транспортных средств – граф возможных путей между узлами
- В искусственном интеллекте – графы для принятия решений,
- В научном ПО

Линейные списки с последовательным распределением памяти

- Размещается в массиве

```
//-----Основные операции со стеком
#define SIZE 100 // Размерность стека
int Stack[SIZE]; // Массив для размещения стека
int SP; // Указатель стека
void Init() // Очистка стека
{ SP=-1; } // Стек пуст
void Push(int val) // Запись в стек
{ SP++; // Указатель к следующему
  Stack[SP]=val; // Запись по указателю стека
}
int Pop() // Исключение из стека
{
if (SP < 0 ) return(0); // Стек пуст
return ( Stack[SP--]); // Возвратить элемент по указателю
} // Указатель к предыдущему
```

Очереди



```

//-----Основные операции с очередью
#define   SIZE   100           // Максимальная длина очереди
int  QUEUE[SIZE];           // Массив элементов очереди
int  fst;                   // Указатели на первый
                               // элемент очереди (индекс)
int  lst;                   // Указатель на следующий
                               // свободный за последним (индекс)

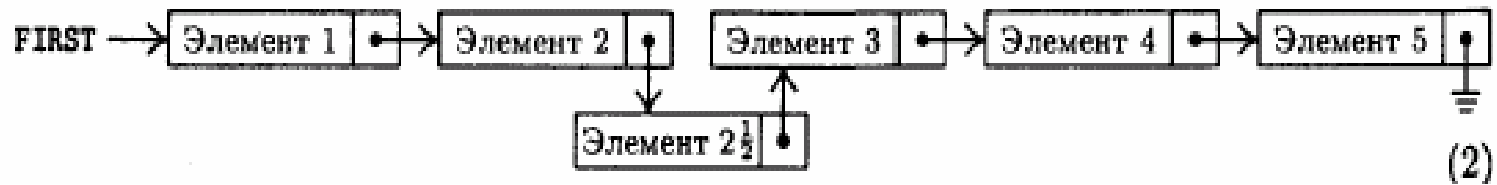
void   Clear()
{ fst = lst = 0; }           // Очистить очередь
int  In(int val)            // Поставить в конец очереди
{
int  next;
if ((next = (lst+1) % SIZE) == fst)
    return(0);              // Переполнение очереди
QUEUE[lst] = val;
lst = next;
return(1);
}
int  Out()                  // Взять из начала очереди
{
int  val;
if (fst == lst) return(0);  // Очередь пуста
val = QUEUE[fst++];        //
fst %= SIZE;               // По достижении fst==SIZE
return(val);               // сбрасывается в 0
}

```


Связанные списки

Основаны на:

динамическом распределении памяти
и использовании указателей



Недостатки статического определения переменных

- внешние переменные занимают области сегмента данных программы и увеличивают соответственно размер программного файла;
- размерность таких массивов как для внешних переменных (сегмент данных), так и для автоматических (стек) бывает зачастую ограничена;
- если в программе несколько переменных большой размерности, то заранее установленные размерности могут не соответствовать текущим соотношениям объемов входных данных разного типа. При этом перераспределение памяти невозможно.

- Можно выделить три варианта возможных изменений размерности данных в программе:
 - рассмотренный ранее статический вариант, когда размерность данных устанавливается при трансляции;
 - размерность данных становится известной в некоторый момент работы программы, затем установленная размерность данных не изменяется;
 - размерность данных меняется в течение всего времени работы программы.

- Два последних варианта могут быть реализованы при наличии механизма создания и уничтожения переменных работающей программой.
- Такие переменные называются **ДИНАМИЧЕСКИМИ**, а область памяти, в которой они создаются - **ДИНАМИЧЕСКОЙ ПАМЯТЬЮ** или "кучей".
- Средства работы с динамической памятью могут быть реализованы транслятором в виде внешних (библиотечных) функций (низкий уровень), либо в виде операторов (высокий уровень).

- **Динамическая память** — оперативная память компьютера, предоставляемая программе (процедуре, подпрограмме) при её работе.
- Динамическое размещение данных означает распределение динамической памяти непосредственно при работе программы или процедуры (подпрограммы).
- В отличие от этого статическое размещение (выделение памяти) осуществляется в момент запуска программы или процедуры.
- На этапе компиляции не известны ни тип, ни количество динамически размещаемых данных. В C++ эта память называется кучей (heap).

Динамическое распределение памяти.

Динамические переменные

- Динамические переменные создаются и уничтожаются работающей программой путем выполнения специальных операторов или вызовов функций;
- Количество и размерность динамических переменных может меняться в процессе работы программы и зависит от количества вызовов соответствующих функций и передаваемых при вызове параметров;
- Динамическая переменная не имеет имени, доступ к ней возможен только через указатель;
- При выполнении функции создания динамической переменной в "куче" выделяется свободная память необходимого размера и возвращается указатель на нее (адрес);
- Функции уничтожения динамической переменной передается указатель на уничтожаемую переменную.

Особенности работы с динамическими переменными

- Если динамическая переменная создана, а указатель на нее "потерян" программой, то такая переменная представляет собой "вещь в себе" - существует, но недоступна для использования; такие переменные обычно называются «мусором» (проблема сбора мусора);
- Динамическая переменная может, в свою очередь, содержать один или несколько указателей на другие динамические переменные. В этом случае мы получаем динамические структуры данных, в которых количество переменных и связи между ними могут меняться в процессе работы программы (списки, деревья, виртуальные массивы);
- Управление динамической памятью построено обычно таким образом, что ответственность за корректное использование указателей на динамические переменные несет программа (точнее, программист, написавший ее). Ошибки в процессе создания, уничтожения и работы с динамическими переменными (повторная попытка уничтожения динамической переменной, попытка уничтожения переменной, не являющейся динамической и т.д.), трудно обнаруживаются и приводят к непредсказуемым последствиям в работе программы.

Основные функции для динамического распределения памяти в С

```
void *malloc(int size);  
    // выделить область памяти размером  
    // в size байтов и вернуть адрес  
void free(void *p);  
    // освободить область памяти,  
    // выделенную по адресу p  
void *realloc(void *p, int size);  
    // расширить выделенную область памяти  
    // до размера size, при изменении адреса  
    // переписать старое содержимое блока
```


Пример создания динамической переменной

```
#include <alloc.h>
double *pd;
pd = malloc(sizeof(double));
if (pd !=NULL)
{
    *pd = 5;
    free(pd);
}
```

Операторы для динамического распределения памяти в C++

- Для работы с динамическими переменными в C++ введены операторы управления динамической памятью **new** и **delete**. Их отличительные особенности:
 - при создании динамической переменной в операторе **new** указывается тип создаваемой переменной, а не размерность, при этом оператор возвращает указатель того же самого типа;
 - если выделяется память под массив динамических переменных, то в операторе **new** добавляются квадратные скобки

Пример создания динамической переменной и массива

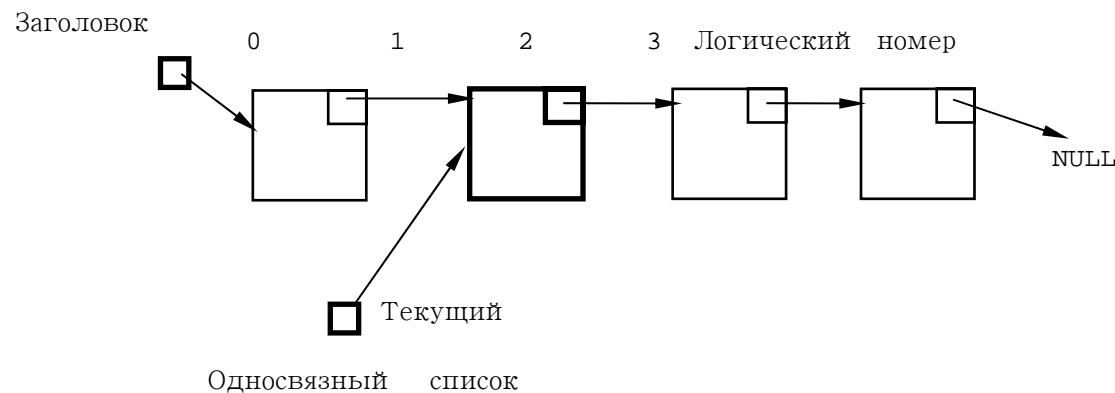
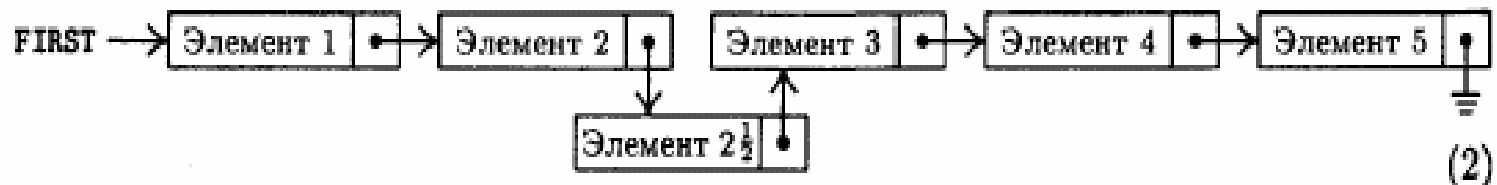
```
double *pd;
pd = new double;           // Обычная динамическая переменная
if (pd !=NULL)
{
    *pd = 5;

    delete pd;
}
double *pd;                // Массив динамических переменных
pd = new double[20];
if (pd !=NULL)
{
    for (i=0; i<20; i++) pd[i]=0;
    delete pd;
}
```

Связанные (динамические) СПИСКИ

Основаны на:

динамическом распределении памяти
и использовании указателей



Фрагменты для работы со связанным списком

// элемент списка

```
struct list
{
    list *next;
    int    val;
}        *ph;
```

```
struct list *p; // указатель на текущий элемент
p = ph; // текущий указатель - на первый элемент
p->next ... // указатель на следующий элемент
p = p->next; // переход к следующему элементу
p !=NULL ... // проверка на конец списка
p->next ==NULL ... // проверка на последний элемент
for (p=ph; p !=NULL; p=p->next)... // просмотр элементов списка

if (ph !=NULL)
{
    for (p=ph; p->next !=NULL; p=p->next); // поиск последнего элемента
}
list *pred; // просмотр с сохранением указателя на предыдущий элемент
for (pred=NULL, p=ph; p !=NULL; pred=p, p=p->next) ... // указатель на второй элемент от текущего
ph->next = pnew; // включение в начало непустого списка
ph = pnew;

//
pnew->next = NULL; // включение в конец непустого списка
pred->next = pnew; // списка
pred->next = p->next; // исключение текущего элемента списка (если он не первый)
pnew->next = p->next; // включение после текущего элемента
p->next = pnew;
```

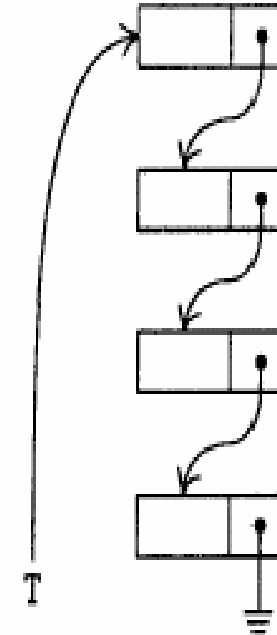
Стек

```
// элемент стека и указатель
// на вершину стека
struct list
{
    void *next;
    int val;
} *top=NULL;
// помещение в стек
void push(struct list *p)
{ p->next=top;
  top=p;
};
// извлечение из стека
list *pop()
{ list *p;
  if (top!=NULL)
  { p=top;
    top=top->next;
    return p; }
  else
  return NULL;
}

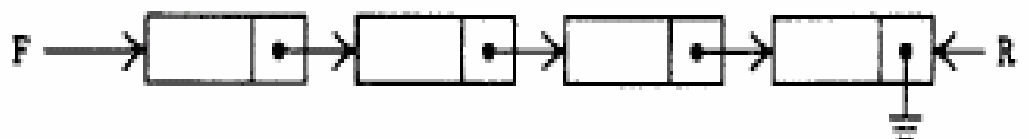
int main(int argc, char* argv[])
{
    return 0;
}
```

Фрагмент с
ИСПОЛЬЗОВАНИЕМ

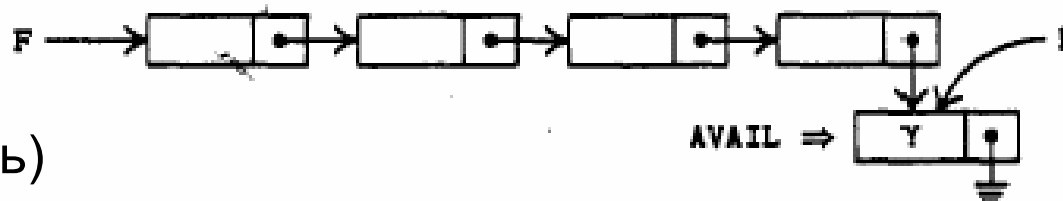
```
struct list *p, *pp;
p=new list;
strcpy(p->val,"Hello\n");
push(p);
printf("%s",top->val);
pp=pop();
printf("%s",pp->val);
```



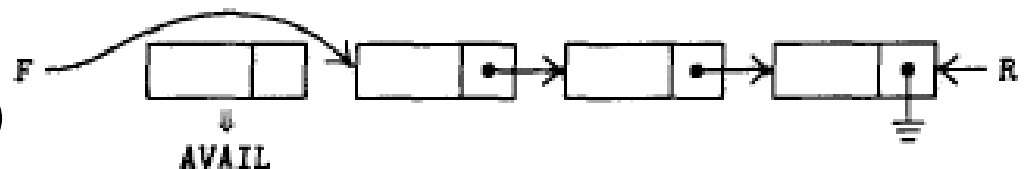
Очереди



Включение нового
Элемента
(постановка в очередь)



Удаление элемента
(извлечение из очереди)

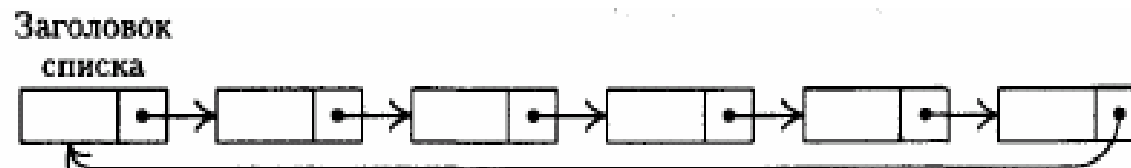
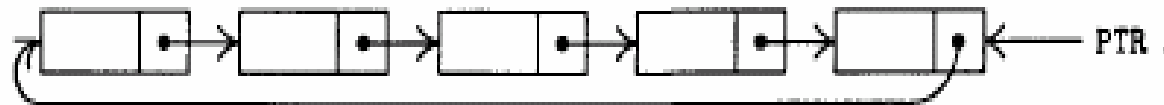


```

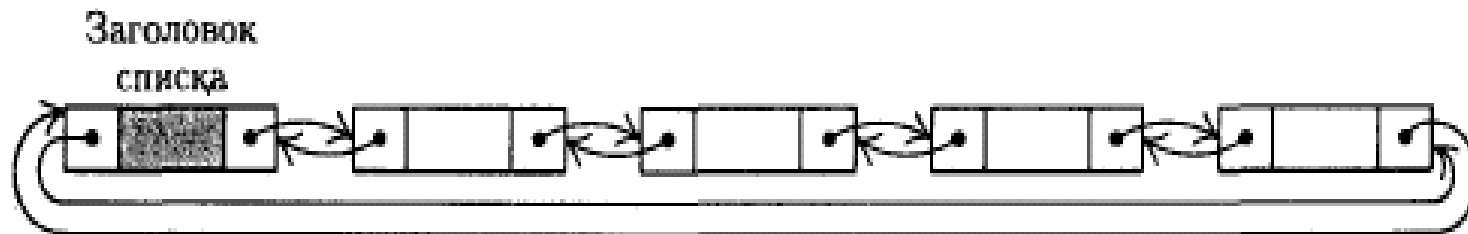
//----- Постановка элемента в конец очереди
// list *PH[2]; - заголовок очереди
void intoFIFO(list *ph[], int v)
{
list *p= new list; // создать элемент списка;
p->val = v; // и заполнить его
p->next = NULL; // новый элемент - последний
if (ph[0] == NULL) // включение в пустую
    ph[0] = ph[1] = p; // очередь
else // включение за последним
    {
    // элементом
    ph[1]->next = p; // следующий за последним = новый
    ph[1] = p; // новый = последний
    }
}
//----- Извлечение из очереди
int fromFIFO(list *ph[])
{ list *q;
if (ph[0] ==NULL) return -1; // очередь пуста
q = ph[0]; // исключение первого
ph[0] = q->next; // элемента
if (ph[0] ==NULL)
    ph[1] = NULL; // элемент единственный
int v = q->val;
delete q;
}

```

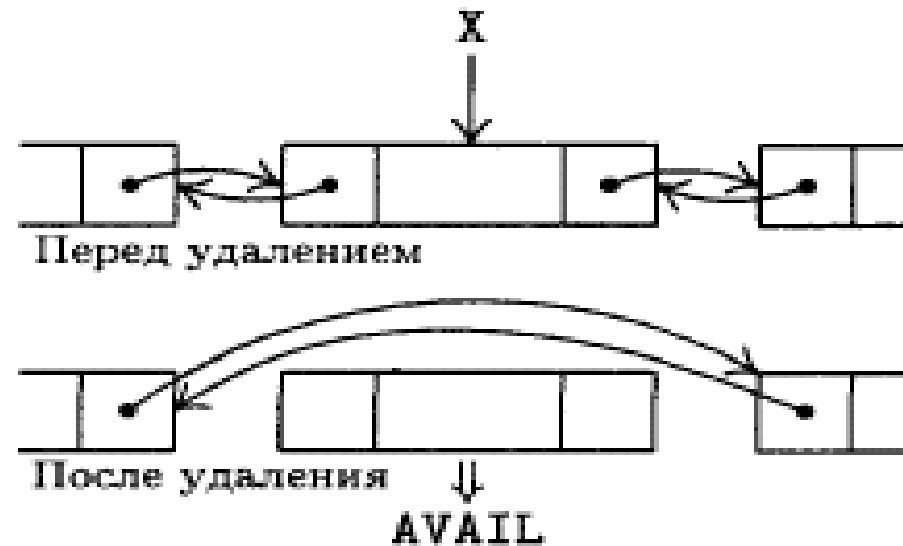

Циклические списки



Дважды связанные (двусвязные) СПИСКИ



Удаление элемента из дважды связанного списка



```

struct    list2
{
    list2 *next, *pred;
    int    val;
};
list *InsertSort(list2 *ph, int v)
{
list2 *q, *p=new list;
p->val = v;
p->pred = p->next = NULL;
if (ph == NULL) return p;// включение в пустой список
for (q=ph; q !=NULL && v > q->val; q=q->next);
// поиск места включения - q
if (q == NULL)                                // включение в конец списка
{                                              // восстановить указатель на
    for (q=ph; q->next!=NULL; q=q->next);
    p->pred = q;                               // последний
    q->next = p;
    return ph;
}                                              // включить перед текущим
p->next=q;                                    // следующий за новым = текущий
p->pred=q->pred;                              // предыдущий нового = предыдущий
// текущего
if (q->pred == NULL) // включение в начало списка
    ph = p;
else                                          // включение в середину
    q->pred->next = p;// следующий за предыдущим = новый
q->pred=p;                                  // предыдущий текущего = новый
return ph;
}

```

Включение в двусвязный список нового
элемента с сохранением
упорядоченности значений

Проблемы динамического распределения памяти

- Проблема сбора мусора – удаления свободных переменных, к которым утерян доступ (добавление их к общему объему динамически распределяемой памяти)
- При выделении участков памяти произвольной (требуемой) длины возникает фрагментация памяти (разбиение всего объема на много маленьких участков, мало пригодных для использования).
Дефрагментация (в отличие от файловой системы) затруднительна.