

**Подготовительная
программа по
программированию на
C/c++**

Занятие №2

Валентина Глазкова

Реализация структур данных на языке С

- Многомерные массивы
- Списки, стеки, очереди, деревья. Выделение динамической памяти

Многомерные массивы

- Двумерный массив — объект данных $T\ a[N][M]$, который:
 - содержит N последовательно расположенных в памяти строк по M элементов типа T в каждой;
 - в общем и целом инициализируется аналогично одномерным массивам;
 - по характеристикам выравнивания идентичен объекту $T\ a[N * M]$, что сводит его двумерный характер к удобному умозрительному приему, упрощающему обсуждение и визуализацию порядка размещения данных.
- Массивы размерности больше двух считаются многомерными, при этом $(N + 1)$ -мерные массивы индуктивно определяются как линеаризованные массивы N -мерных массивов, для которых справедливо все сказанное об одно- и двумерных массивах.



Многомерные массивы: пример

```
// определение двумерных массивов
int a[2][3] = { {0, 1}, // частичная инициализация строки
               {2, 3, 4}}; // полная инициализация строки
int b[2][3] = {0, 1, 2, 3, 4};

// результаты:
// a: {0, 1, 0, 2, 3, 4}; b: {0, 1, 2, 3, 4, 0}

// определение массивов размерности больше 2
double d[3][5][10];
int32_t k[5][4][3][2];
```

Двумерные массивы и векторы векторов

- **Двумерный массив** следует отличать от **вектора векторов**, работа с которым:
 - предполагает двухступенчатую процедуру создания и удаления;
 - гарантирует смежность хранения данных только в пределах одной строки (аналогичная гарантия предоставляется и в отношении указателей на строки);
 - ведёт к большей фрагментации памяти, но повышает вероятность успешного выделения в памяти непрерывных фрагментов (требование памяти объёма $\sim N^2$ заменяется требованием памяти объёма $\sim N$).
- Многомерные массивы и векторы векторов (векторов...) являются различными структурами данных с разной дисциплиной использования.



Двумерные массивы и векторы векторов: пример

```
// создание вектора векторов
```

```
int **v = (int**)malloc(N * sizeof(int*));
```

```
for(int i = 0; i < N; i++)
```

```
    // NB: в каждой строке значение M может быть разным
```

```
    v[i] = (int*)malloc(M * sizeof(int));
```

```
// ...
```

```
// удаление вектора векторов
```

```
for(int i = 0; i < N; i++)
```

```
    free(v[i]);
```

```
free(v);
```

Многомерные массивы и указатели

- Для многомерных массивов справедлив ряд тождеств, отражающих эквивалентность соответствующих выражений языка C. Так, для двумерного массива `T a[N][M]` справедливо:
 - `a == &a[0]; a + i == &a[i];`
 - `*a = a[0] == &a[0][0];`
 - `**a == *&a[0][0] == a[0][0];`
 - `a[i][j] == *(*a + i) + j.`
- Использование операции разыменования `*` не имеет каких-либо преимуществ перед доступом по индексу, и наоборот. Трансляция и первой, и второй формы записи в объектный код приводит в целом к одинаковым результатам.



Многомерные массивы и указатели: пример

```
// указатели на массивы и массивы указателей
```

```
int k[3][5];
```

```
int (*pk)[5];    // указатель на массив int[5]
```

```
int *p[5];       // массив указателей (int*)[5]
```

```
// примеры использования (все — допустимы)
```

```
pk    = k;       // аналогично: pk = &k[0];
```

```
pk[0][0] = 1;    // аналогично: k[0][0] = 1;
```

```
*pk[0] = 2; // аналогично: k[0][0] = 2;
```

```
**pk    = 3; // аналогично: k[0][0] = 3;
```


Многомерные массивы и функции

- В общем случае для передачи функции двумерного массива `T a[N][M]` используется формальный параметр `T **pa`. При этом информация о размерности массива закреплена в типе (`T**`), а размеры передаются как **дополнительные параметры**.
- Существование в языке объектов вида `T (*pt) [M]` открывает возможность **встраивания** информации о длине строк массива в тип данных и определения функций с прототипами вида:
 - `void foo(T a[][M], size_t N);`
 - `void foo(T (*a)[M], size_t N);` // комплект `[]` трактуется как указатель
- Для многомерных массивов аналогично имеем:
 - `void bar(T b[][SZ2][SZ3][SZ4], size_t sz1);`
 - `void bar(T (*b)[SZ2][SZ3][SZ4], size_t sz1);`

Динамические структуры данных

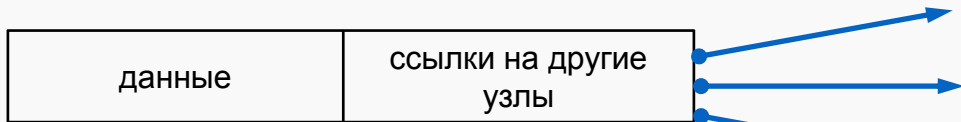
- **Динамические данные**
 - Размер заранее неизвестен, определяется во время работы программы
 - Память выделяется во время работы программы (для выделения памяти используются функции `malloc`, `calloc`, `realloc`; для освобождения – `free`)
- **Структура данных** – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных
- Типичные операции
 - Добавление данных
 - Изменение данных
 - Удаление данных
 - Поиск данных

Динамические структуры данных

Реализация: набор узлов, объединенных с помощью ссылок

Структура узла:

Типы структур:



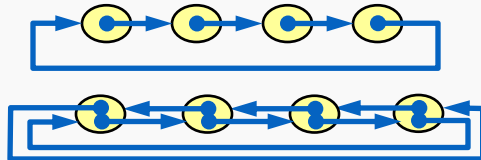
СПИСКИ
ОДНОСВЯЗНЫЙ



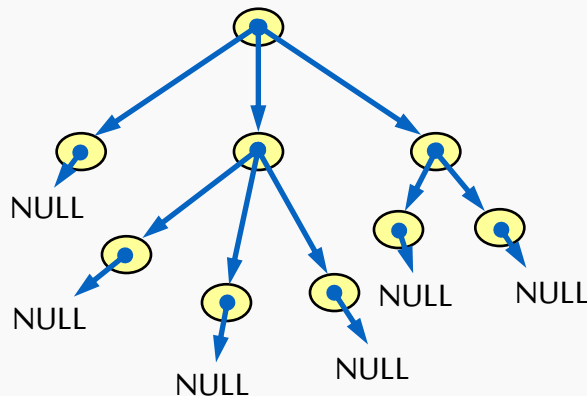
двунаправленный (двусвязный)



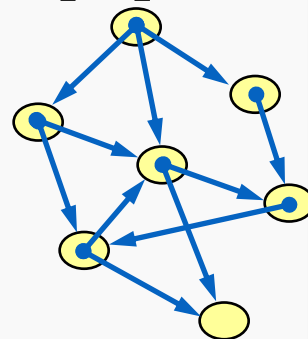
циклические списки (кольца)



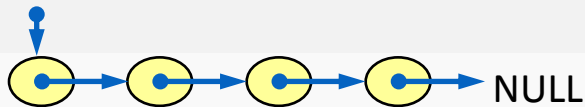
деревья



графы



Односвязный список



Определение

- Динамическая структура данных, состоящая из узлов, каждый из которых содержит данные и ссылку на следующий узел списка

Определение (рекурсивное)

- пустая структура – это список;
- список – это начальный узел (голова) и связанный с ним список.

Отличия от массивов

- Порядок элементов может не совпадать с порядком расположения элементов данных в памяти
- Порядок обхода списка всегда явно задаётся его внутренними связями (в односвязном списке можно передвигаться только в сторону конца списка)

Односвязный список

Основные операции со списками:

- Поиск, вставка, удаление элемента

Структура узла:

```
struct Node {  
    char word[20]; // данные  
    int count;    // данные  
    Node *next;  // ссылка на следующий элемент  
};  
typedef Node *PNode;
```

Адрес начала списка (головы списка)

```
PNode Head = NULL;
```

Создание узла списка

Функция `CreateNode` (*создать узел*):

ВХОД: новое слово, прочитанное из файла;

ВЫХОД: адрес нового узла, созданного в памяти.

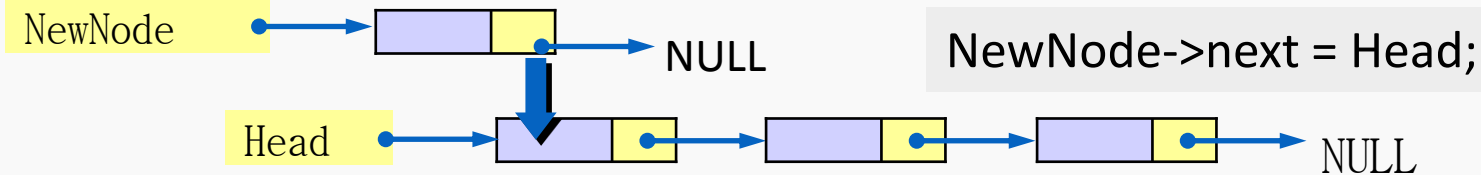
возвращает адрес
созданного узла

НОВОЕ СЛОВО

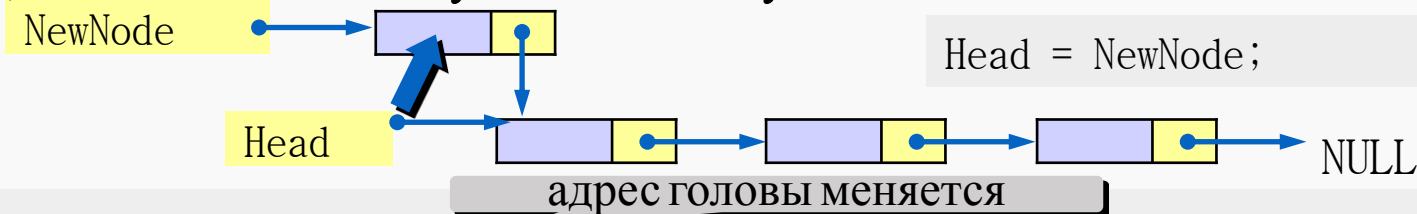
```
PNode CreateNode ( char NewWord[] )
{
    PNode NewNode = (PNode) malloc(sizeof(Node));
    strcpy(NewNode->word, NewWord);
    NewNode->count = 1;
    NewNode->next = NULL;
    return NewNode;
}
```

Добавление узла в начало списка

1) Установить ссылку нового узла на голову списка:



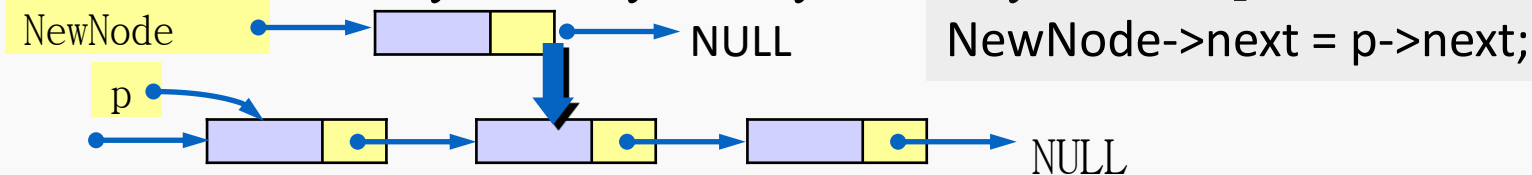
2) Установить новый узел как голову списка:



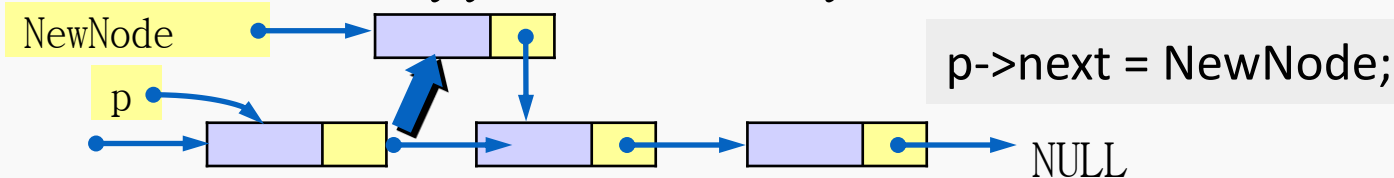
```
void AddFirst (Pnode *Head, PNode NewNode)
{
    NewNode->next = Head;
    Head = &NewNode;
}
```

Добавление узла после заданного

1) Установить ссылку нового узла на узел, следующий за p:



2) Установить ссылку узла p на новый узел:

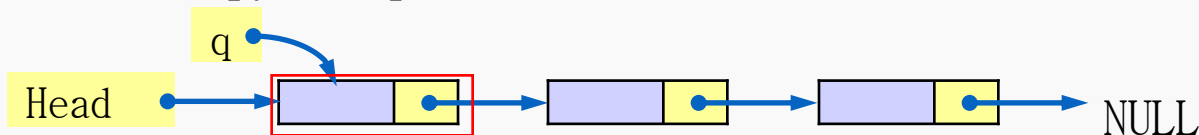


```
void AddAfter (PNode p, PNode NewNode)
{
    NewNode->next = p->next;
    p->next = NewNode;
}
```


Проход по списку

Задача:

выполнить некоторую операцию с каждым элементом списка.



Алгоритм:

- 1) установить вспомогательный указатель q на голову списка;
- 2) если указатель q равен $NULL$ (дошли до конца списка), то стоп;
- 3) выполнить действие над узлом с адресом q ;
- 4) перейти к следующему узлу, $q \rightarrow next$.

```
PNode q = Head;    // начали с головы
while ( q != NULL ) { // пока не дошли до конца
    ...             // делаем что-то с q
    q = q->next;     // переходим к следующему узлу
}
```

Добавление узла в конец списка

Задача: добавить новый узел в конец списка.

Алгоритм:

- 1) найти последний узел q , такой что $q \rightarrow \text{next}$ равен NULL ;
- 2) добавить узел после узла с адресом q (процедура `AddAfter`).

Особый случай: добавление в пустой список.

```
void AddLast ( PNode *Head, PNode NewNode )
```

```
{
```

```
    PNode q = *Head;
```

```
    if ( Head == NULL ) {
```

```
        AddFirst( Head, NewNode );
```

```
        return;
```

```
    }
```

```
    while ( q->next ) q = q->next;
```

```
    AddAfter ( q, NewNode );
```

```
}
```

особый случай – добавление в
пустой список

ищем последний узел

добавить узел после узла q

Поиск слова в списке

Задача:

найти в списке заданное слово или определить, что его нет.

Функция Find:

- вход: слово (символьная строка);
- выход: адрес узла, содержащего это слово или NULL.

Алгоритм: проход по списку.

результат – адрес узла

ищем это слово

```
PNode Find ( PNode Head, char NewWord[ ] )
```

```
{
```

```
    PNode q = Head;
```

```
    while (q && strcmp(q->word, NewWord))
```

```
        q = q->next;
```

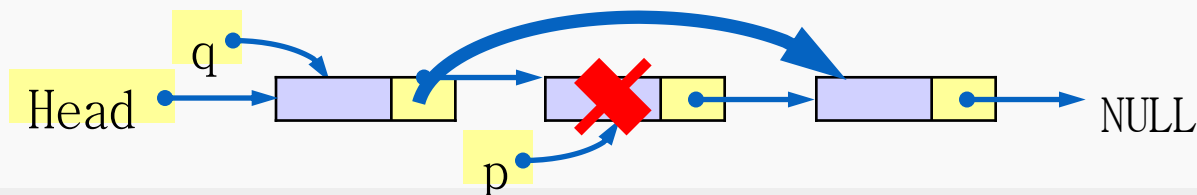
```
    return q;
```

```
}
```

пока не дошли до конца списка и слово
не равно заданному

Удаление узла списка

Проблема: нужно знать адрес узла перед удаляемым



```
void DeleteNode ( PNode *Head, PNode p )
```

```
{
```

```
    PNode q = *Head;
```

```
    if ( *Head == p )
```

```
        Head = &(p->next);
```

```
    else {
```

```
        while ( q && q->next != p )
```

```
            q = q->next;
```

```
        if ( q == NULL ) return;
```

```
        q->next = p->next;
```

```
    }
```

```
    free(p);
```

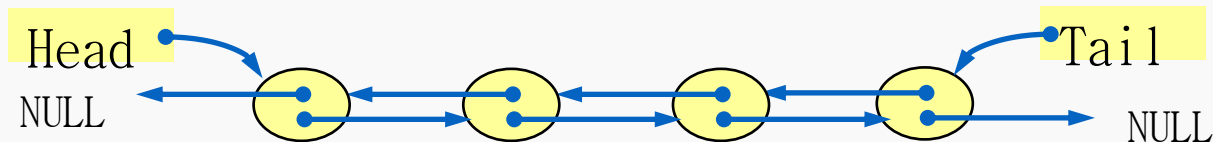
```
}
```

особый случай:
удаляем первый узел

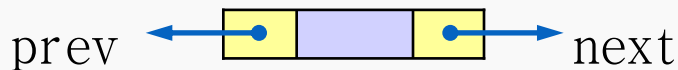
ищем предыдущий
узел, такой что
 $q \rightarrow \text{next} == p$

освобождение памяти

Двусвязные списки



Структура узла:



```
struct Node {  
    char word[40]; // слово  
    int count;    // счетчик повторений  
    Node *next;   // ссылка на следующий элемент  
    Node *prev;   // ссылка на предыдущий элемент  
};
```

Указатель на эту структуру:

```
typedef Node *PNode;
```

Адреса «ГОЛОВЫ» и «ХВОСТА»:

```
PNode Head = NULL;
```

```
PNode Tail = NULL;
```



можно двигаться в обе
стороны



нужно работать с двумя
указателями вместо одного

Стек

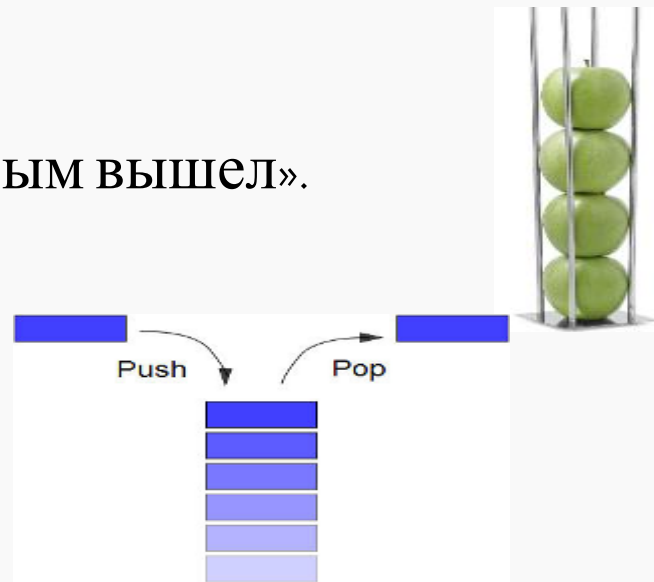
Стек – это линейная структура данных, в которой добавление и удаление элементов возможно только с одного конца (вершины стека).

LIFO = *Last In – First Out*

- «Кто последним вошел, тот первым вышел».

Операции со стеком:

- 1) добавить элемент на вершину (*Push* = втолкнуть);
- 2) снять элемент с вершины (*Pop* = вытолкнуть).



Стек: пример (1/2)

Задача: вводится символьная строка, в которой записано выражение со скобками трех типов: [], { } и (). Определить, верно ли расставлены скобки (не обращая внимания на остальные символы). Примеры:

[()] { }] [[({)] }

Задача: то же самое, но с одним видом скобок.

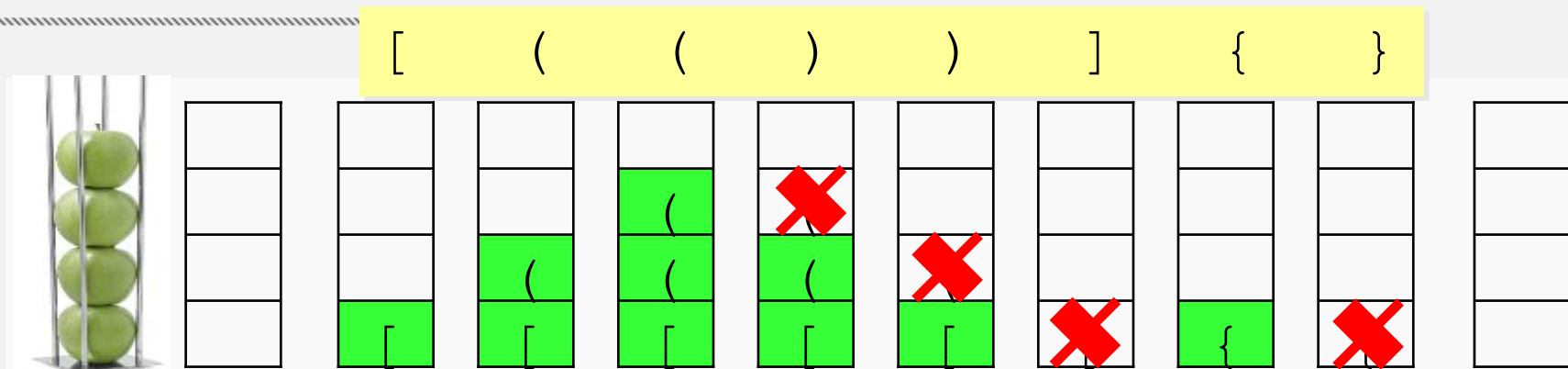
- Решение: счетчик вложенности скобок. Последовательность правильная, если в конце счетчик равен нулю и при проходе не разу не становился отрицательным.

(())	()
1	2	1	0	1	0

(()))	(
1	2	1	0	-1	0

(())	(
1	2	1	0	1
[({)] }
:	:	:	:	:
(:	:	:	:
[:	:	:	:
{	:	:	:	:

Стек: пример (2/2)



Алгоритм:

- 1) в начале стек пуст;
- 2) в цикле просматриваем все символы строки по порядку;
- 3) если очередной символ – открывающая скобка, заносим ее на вершину стека;
- 4) если символ – закрывающая скобка, проверяем вершину стека: там должна быть соответствующая открывающая скобка (если это не так, то ошибка);
- 5) если в конце стек не пуст, выражение неправильное.

Реализация стека (список)

Структура узла:

```
struct Node {  
    char data;  
    Node *next;  
};  
typedef Node *PNode;
```

Добавление элемента:

```
void Push (PNode *Head, char x)  
{  
    PNode NewNode = (PNode) malloc(sizeof(Node));  
    NewNode->data = x;  
    NewNode->next = *Head;  
    Head = &NewNode;  
}
```

Реализация стека (список)

Извлечение элемента с вершины:

```
char Pop (PNode *Head) {  
    char x;  
    PNode q = *Head;  
    if ( Head == NULL ) return char(255);  
    x = (*Head)->data;  
    *Head = (*Head)->next;  
    free(q);  
    return x;  
}
```



стек пуст

Стек: пример (1/2)

Вычисление арифметических выражений

$(a + b) / (c + d - 1)$

Инфиксная запись

(знак операции между операндами)



необходимы скобки

Префиксная запись (знак операции до операндов)

/

$a + b$

$c + d - 1$

польская нотация



скобки не нужны, можно однозначно вычислить

Постфиксная запись (знак операции после операндов)

$a + b$

$c + d - 1$

/

обратная польская нотация

Стек: пример (2/2)

Постфиксная форма

X =		a b + c d + 1 - /								
					d		1			
		b		c	c	c+d	c+d	c+d-1		
	a	a	a+b	a+b	a+b	a+b	a+b	a+b		x

Алгоритм:

- 1) взять очередной элемент;
- 2) если это не знак операции, добавить его в стек;
- 3) если это знак операции, то
 - взять из стека два операнда;
 - выполнить операцию и записать результат в стек;
- 4) перейти к шагу 1.

Очередь

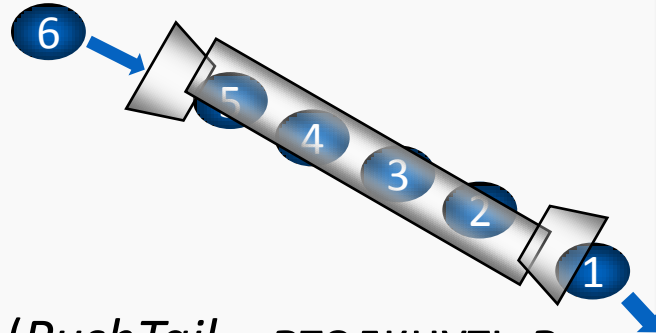
Очередь – это линейная структура данных, в которой добавление элементов возможно только с одного конца (конца очереди), а удаление элементов – только с другого конца (начала очереди).

FIFO = *First In – First Out*

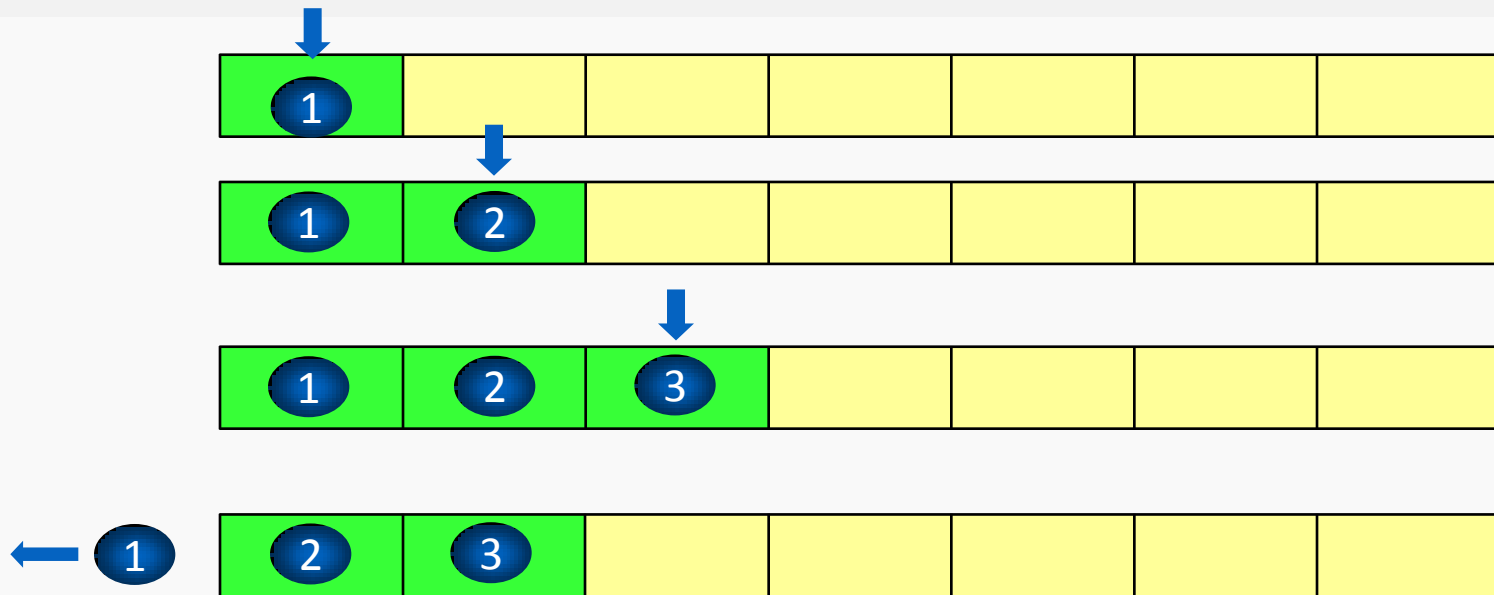
- «Кто первым вошел,
- тот первым вышел».

Операции с очередью:

- 1) добавить элемент в конец очереди (*PushTail* = втолкнуть в конец);
- 2) удалить элемент с начала очереди (*Pop*).



Реализация очереди (массив)



самый простой способ



- 1) нужно заранее выделить массив;
- 2) при выборке из очереди нужно сдвигать все элементы.

Реализация очереди (списки)

Структура узла:

```
struct Node {  
    int data;  
    Node *next;  
};
```

```
typedef Node *PNode;
```

Тип данных «очередь»:

```
struct Queue {  
    PNode Head, Tail;  
};
```

Реализация очереди (списки)

Добавление элемента:

```
void PushTail ( Queue *Q, int x )
```

```
{
```

```
    PNode NewNode;
```

```
    NewNode = (PNode)malloc(sizeof(Node));
```

```
    NewNode->data = x;
```

```
    NewNode->next = NULL;
```

```
    if ( (*Q).Tail )
```

```
        (*Q).Tail->next = NewNode;
```

```
    (*Q).Tail = NewNode;
```

```
    if ( (*Q).Head == NULL )
```

```
        (*Q).Head = (*Q).Tail;
```

```
}
```

создаем новый
узел

если в списке уже что-то
было, добавляем в конец

если в списке
ничего не было, ...

Реализация очереди (списки)

Выборка элемента:

```
int Pop ( Queue *Q )
```

```
{
```

```
    PNode top = (*Q).Head;
```

```
    int x;
```

```
    if ( top == NULL )
```

```
        return 32767;
```

```
    x = top->data;
```

```
    (*Q).Head = top->next;
```

```
    if ( (*Q).Head == NULL )
```

```
        (*Q).Tail = NULL;
```

```
    free(top);
```

```
    return x;
```

```
}
```

если список пуст,...

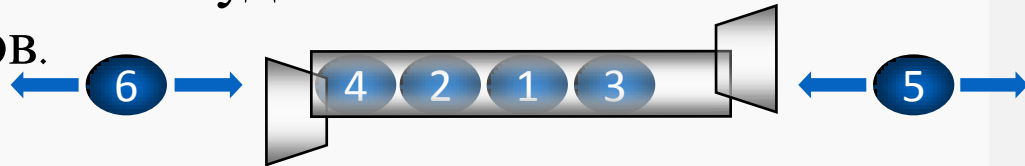
запомнили первый
элемент

если в списке
ничего не осталось

освободить память

Очередь с двумя концами (дек)

Дек (*deque = double ended queue*) – это линейная структура данных, в которой добавление и удаление элементов возможно с обоих концов.



Операции с деком:

- 1) добавление элемента в начало (*Push*);
- 2) удаление элемента с начала (*Pop*);
- 3) добавление элемента в конец (*PushTail*);
- 4) удаление элемента с конца (*PopTail*).

Реализация:

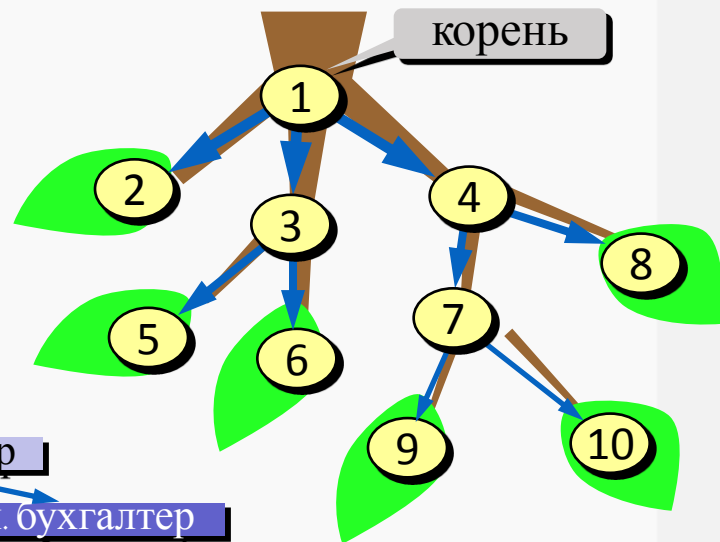
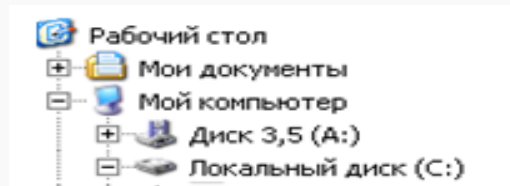
- 1) кольцевой массив;
- 2) двусвязный список.

Деревья

Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.

Корень – это начальный узел дерева.

Лист – это узел, из которого не выходит ни одной дуги.



Деревья

С помощью деревьев изображаются отношения подчиненности (иерархия, «старший – младший», «родитель – ребенок»).

Предок узла x – это узел, из которого существует путь по стрелкам в узел x .

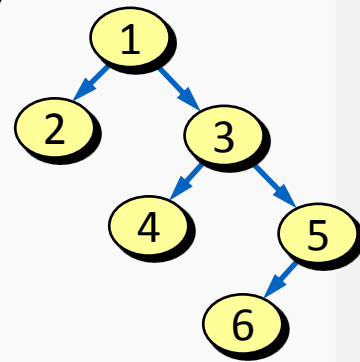
Потомок узла x – это узел, в который существует путь по стрелкам из узла x .

Родитель узла x – это узел, из которого существует дуга непосредственно в узел x .

Сын узла x – это узел, в который существует дуга непосредственно из узла x .

Брат узла x – это узел, у которого тот же родитель, что и у узла x .

Высота дерева – это наибольшее расстояние от корня до листа (количество дуг).



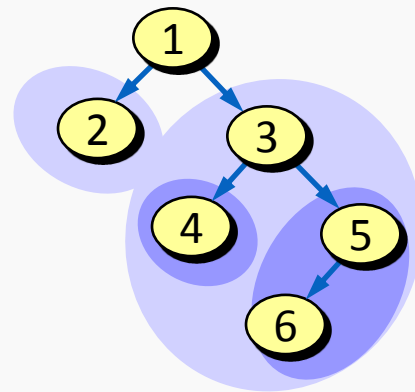
Дерево – рекурсивная структура данных

Рекурсивное определение:

1. Пустая структура – это дерево.
2. Дерево – это корень и несколько связанных с ним деревьев.

Двоичное (бинарное) дерево – это дерево, в котором каждый узел имеет не более двух сыновей.

1. Пустая структура – это двоичное дерево.
2. Двоичное дерево – это корень и два связанных с ним двоичных дерева (левое и правое поддеревья).



Двоичные деревья

Применение:

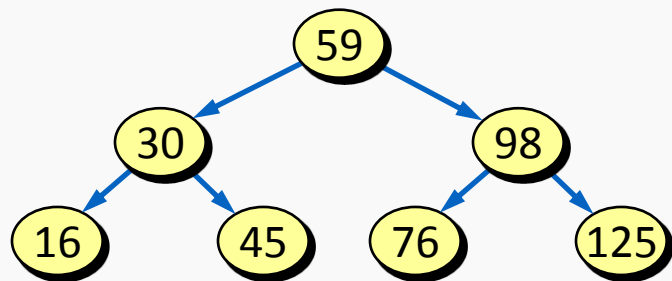
- 1) поиск данных в специально построенных деревьях (базы данных);
- 2) сортировка данных;
- 3) вычисление арифметических выражений

Структура узла:

```
struct Node {  
    int data;           // данные  
    Node *left, *right; // ссылки на левого и правого сыновей  
};  
typedef Node *PNode;
```

Двоичные деревья поиска

Ключ – это характеристика узла, по которой выполняется поиск (чаще всего – одно из полей структуры).



Слева от каждого узла находятся узлы с меньшими ключами, а справа – с бóльшими.

Как искать ключ, равный x :

- 1) если дерево пустое, ключ не найден;
- 2) если ключ узла равен x , то стоп.
- 3) если ключ узла меньше x , то искать x в левом поддереве;
- 4) если ключ узла больше x , то искать x в правом поддереве.

Двоичные деревья поиска

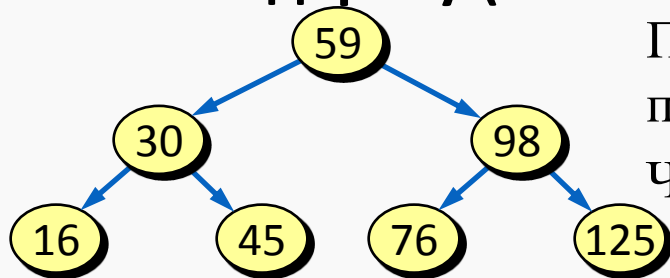
Поиск в массиве (N элементов):



При каждом сравнении отбрасывается 1 элемент.

Число сравнений – N .

Поиск по дереву (N элементов):



При каждом сравнении отбрасывается половина оставшихся элементов.

Число сравнений $\sim \log_2 N$.



быстрый поиск



нужно заранее построить дерево



Реализация алгоритма поиска

```
// Функция Search – поиск по дереву  
// Вход: Tree - адрес корня,  
//      x - искомый ключ  
// Выход: адрес узла или NULL (не нашли)
```

```
PNode Search (PNode Tree, int x)
```

```
{
```

```
    if ( ! Tree ) return NULL;
```

```
    if ( x == Tree->data )  
        return Tree;
```

```
    if ( x < Tree->data )  
        return Search(Tree->left, x);
```

```
    else  
        return Search(Tree->right, x);
```

```
}
```

дерево пустое:
ключ не нашли

нашли, возвращаем
адрес корня

искать в левом
поддереве

искать в правом
поддереве



Построение дерева поиска

```
// Функция AddToTree – добавить элемент к дереву
```

```
// Вход: Tree - адрес корня,
```

```
//      x - что добавляем
```

```
void AddToTree (PNode Tree, int x)
```

```
{
```

```
    if ( ! Tree ) {
```

```
        Tree = (PNode) malloc(sizeof(Node));
```

```
        Tree->data = x;
```

```
        Tree->left = NULL;
```

```
        Tree->right = NULL;
```

```
        return;
```

```
    }
```

```
    if ( x < Tree->data )
```

```
        AddToTree ( Tree->left, x );
```

```
    else AddToTree ( Tree->right, x );
```

```
}
```

дерево пустое: создаем
новый узел (корень)

добавляем к левому или
правому поддереву

Обход дерева

Обход дерева – просмотр всех узлов дерева в определенном порядке.

Обход «левый–корень–правый»



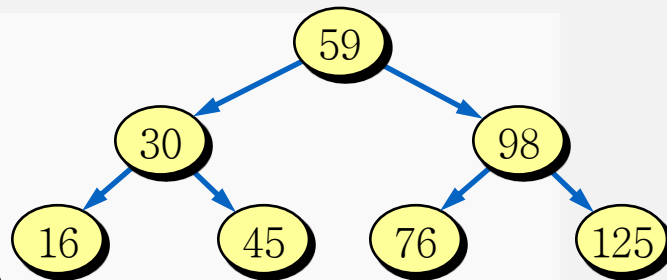
Обход «правый – корень – левый»



Обход «корень – левый – правый»



Обход «левый – правый – корень»





Обход дерева – реализация

```
// Функция LRR – обход дерева в порядке  
//      “левый – корень – правый”  
// Вход: Tree - адрес корня
```

```
void LRR( PNode Tree )
```

обход этой ветки закончен

```
{
```

```
if ( ! Tree ) return;
```

обход левого поддерева

```
LRR ( Tree->left );
```

```
printf ( "%d ", Tree->data );
```

вывод данных корня

```
LRR ( Tree->right );
```

обход правого поддерева

```
}
```



Для рекурсивной структуры удобно
применять рекурсивную обработку

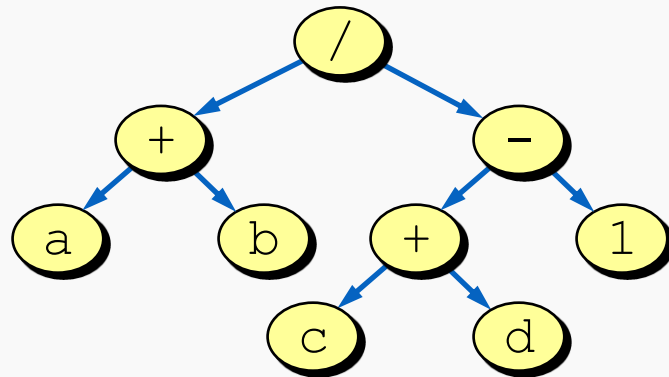
Вычисление арифметических выражений

Задача: в символьной строке записано правильное арифметическое выражение, которое может содержать только однозначные числа и знаки операций $+$ $-$ $*$ \backslash . Вычислить это выражение.

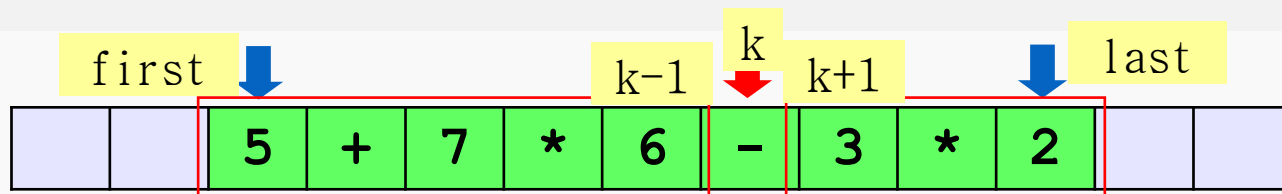
Алгоритм:

- 1) ввести строку;
- 2) построить дерево;
- 3) вычислить выражение по дереву.

$(a + b) / (c + d - 1)$



Построение дерева



Алгоритм:

- 1) если $first = last$ (остался один символ – число), то создать новый узел и записать в него этот элемент; иначе...
- 2) среди элементов от $first$ до $last$ включительно найти последнюю операцию с наименьшим приоритетом (элемент с номером k);
- 3) создать новый узел (корень) и записать в него знак операции;
- 4) рекурсивно применить этот алгоритм два раза:
 - построить левое поддерево, разобрав выражение из элементов массива с номерами от $first$ до $k-1$;
 - построить правое поддерево, разобрав выражение из элементов массива с номерами от $k+1$ до $last$.

Валентина Глазкова

Спасибо за внимание!