

Динамические структуры данных (язык Си)

Тема 5. Стеки, очереди, деки

Стек



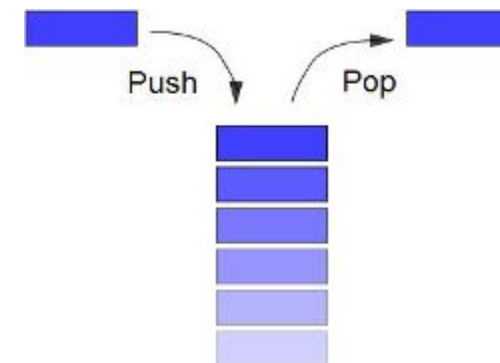
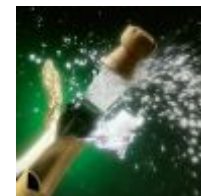
Стек – это линейная структура данных, в которой добавление и удаление элементов возможно только с одного конца (**вершины стека**). *Stack* = кipa, куча, стопка (англ.)

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* = вылететь со звуком).



Пример задачи

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:

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

Упрощенная 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

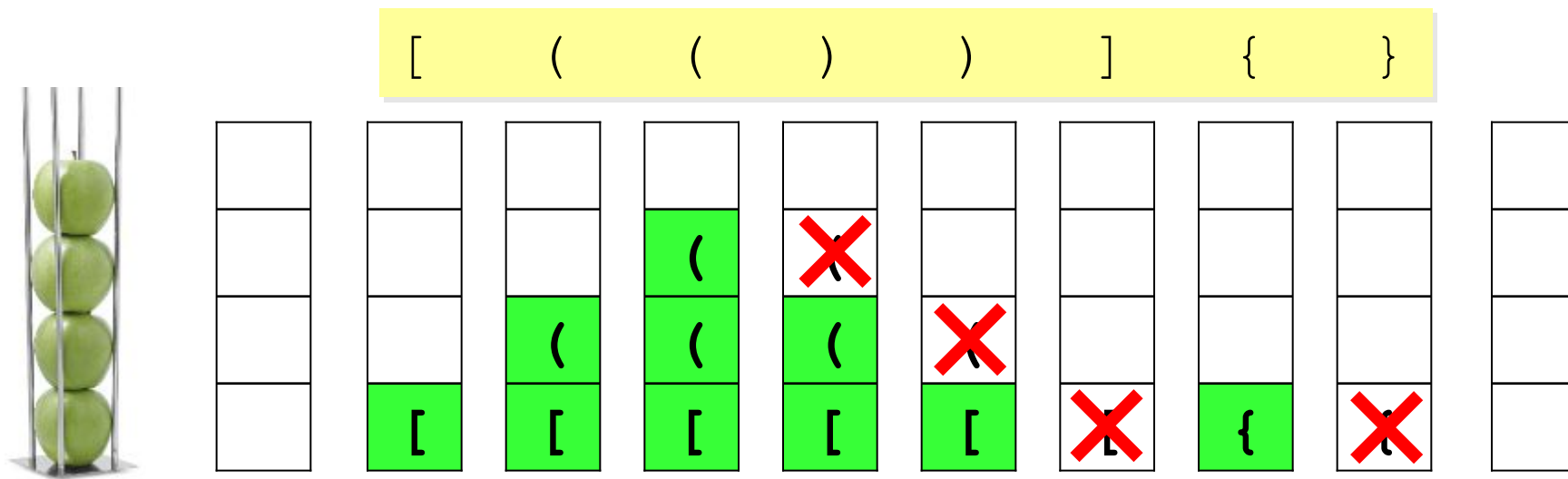
(())	(
1	2	1	0	1



Можно ли решить исходную задачу так же, но с тремя счетчиками?

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

Решение задачи со скобками



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 (массив)

Структура-стек:

```
const MAXSIZE = 100;
struct Stack {
    char data[MAXSIZE]; // стек на 100 символов
    int  size;           // число элементов
};
```

Добавление элемента:

```
int Push ( Stack &S, char x )
{
    if ( S.size == MAXSIZE ) return 0;
    S.data[S.size] = x;
    S.size ++;
    return 1;
}
```

ошибка:
переполнение
стека

добавить элемент

нет ошибки

Реализация стека (массив)

Снятие элемента с вершины:

```
char Pop ( Stack &S )  
{  
    if ( S.size == 0 ) return char(255);  
    S.size --;  
    return S.data[S.size];  
}
```

ошибка:
стек пуст

Пустой или нет?

```
int isEmpty ( Stack &S )  
{  
    if ( S.size == 0 )  
        return 1;  
    else return 0;  
}
```

```
int isEmpty ( Stack &S )  
{  
    return (S.size == 0);  
}
```

Программа

```
void main()
{
    char br1[3] = { '(', '[', '{' };
    char br2[3] = { ')', ']', '}' };
    char s[80], upper;
    int i, k, error = 0;
    Stack S;
    S.size = 0;
    printf("Введите выражение со скобками > ");
    gets ( s );
    ... // здесь будет основной цикл обработки
    if ( ! error && (S.size == 0) )
        printf("\nВыражение правильное\n");
    else printf("\nВыражение неправильное\n");
}
```

открывающие
скобки

закрывающие
скобки

то, что сняли со стека

признак ошибки

Обработка строки (основной цикл)

```
for ( i = 0; i < strlen(s); i++ )
{
    for ( k = 0; k < 3; k++ )
    {
        if ( s[i] == br1[k] )    // если открывающая скобка
        {
            Push ( S, s[i] ); // втолкнуть в стек
            break;
        }
        if ( s[i] == br2[k] ) // если закрывающая скобка
        {
            upper = Pop ( S ); // снять верхний элемент
            if ( upper != br1[k] ) error = 1;
            break;
        }
    }
    if ( error ) break;
}
```

ЦИКЛ ПО ВСЕМ
СИМВОЛАМ СТРОКИ S

ЦИКЛ ПО ВСЕМ ВИДАМ СКОБОК

ошибка: стек пуст или
не та скобка

была ошибка: дальше нет
смысла проверять

Реализация стека (список)

Структура узла:

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

Добавление элемента:

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

Реализация стека (список)

Снятие элемента с вершины:

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

стек пуст

Изменения в основной программе:

```
Stack S;  
S.size = 0;  
...  
if ( ! error && (S.size == 0) )  
    printf("\nВыражение правильное\n");  
else printf("\nВыражение неправильное \n");
```

PNode S = NULL;

(S == NULL)

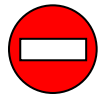
Вычисление арифметических выражений

Как вычислять автоматически:

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

Инфиксная запись

(знак операции **между** операндами)



необходимы скобки!

Префиксная запись (знак операции **до** операндов)

$/ \quad \begin{array}{c} a + \\ b \end{array} \quad c + d - 1$

польская нотация,
[Jan Łukasiewicz](#) (1920)



скобки не нужны, можно однозначно
вычислить!

Постфиксная запись (знак операции **после** операндов)

$\begin{array}{c} a + \\ b \end{array} \quad c + d - 1 \quad /$

обратная польская нотация,
[F. L. Bauer](#) and [E. W. Dijkstra](#)

Запишите в постфиксной форме

$$(32 * 6 - 5) * (2 * 3 + 4) / (3 + 7 * 2)$$

$$(2 * 4 + 3 * 5) * (2 * 3 + 18 / 3 * 2) * (12 - 3)$$

$$(4 - 2 * 3) * (3 - 12 / 3 / 4) * (24 - 3 * 12)$$

Вычисление выражений

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.

Системный стек (*Windows* – 1 Мб)

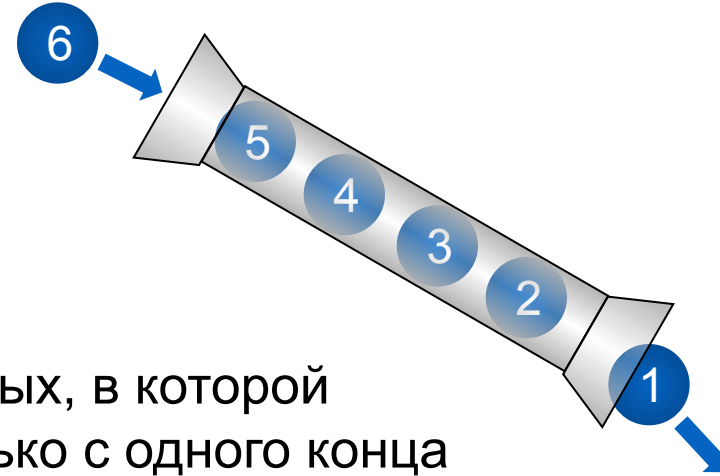
Используется для

- 1) размещения **локальных переменных**;
- 2) хранения **адресов возврата** (по которым переходит программа после выполнения функции или процедуры);
- 3) передачи **параметров** в функции и процедуры;
- 4) временного хранения данных (в программах на языке *Ассемблер*).

Переполнение стека (*stack overflow*):

- 1) слишком много локальных переменных
(**выход** – использовать динамические массивы);
- 2) очень много рекурсивных вызовов функций и процедур
(**выход** – переделать алгоритм так, чтобы уменьшить глубину рекурсии или отказаться от нее вообще).

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**).

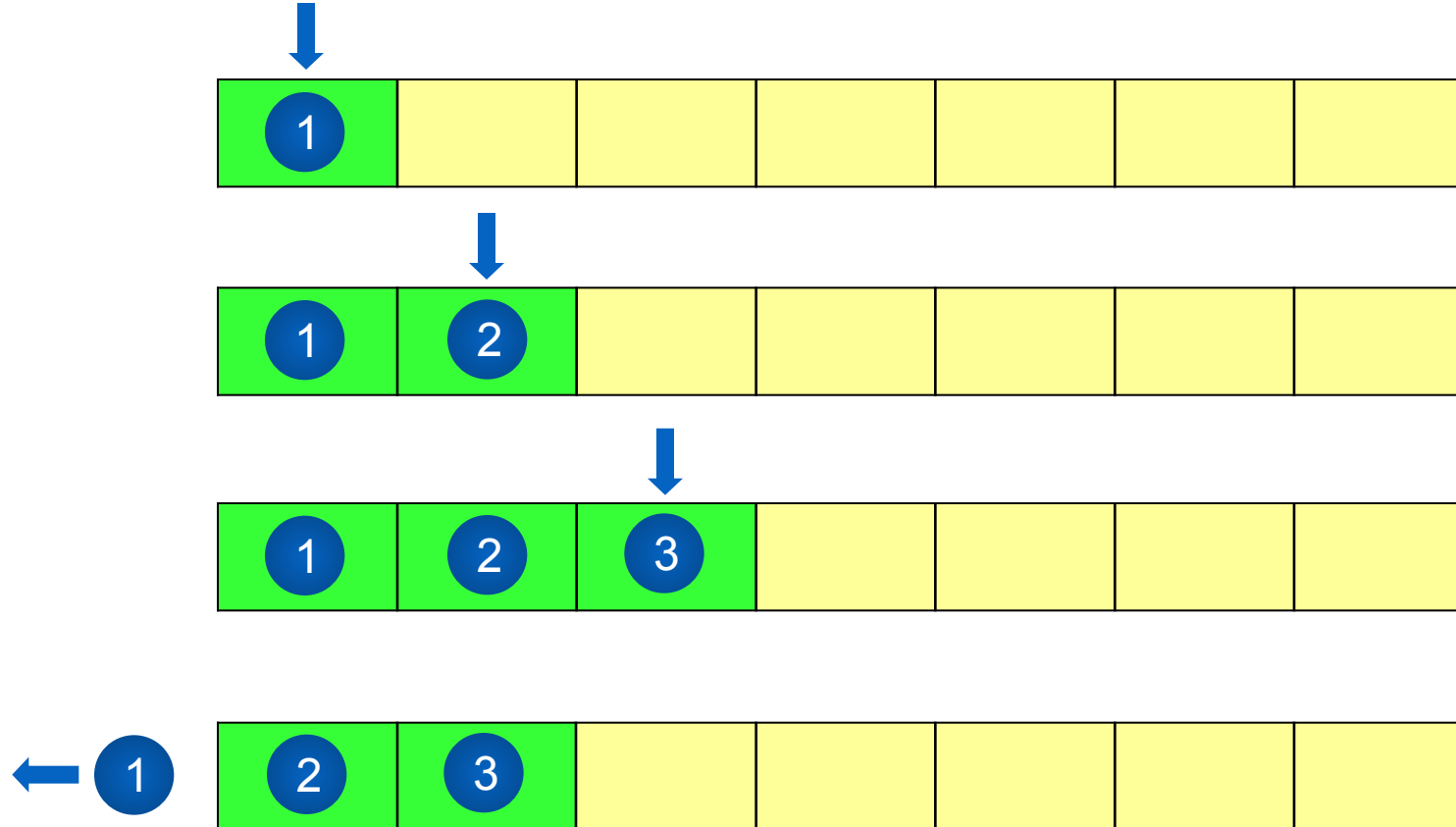
FIFO = *First In – First Out*

«Кто первым вошел, тот первым вышел».

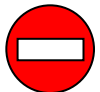
Операции с очередью:

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

Реализация очереди (массив)

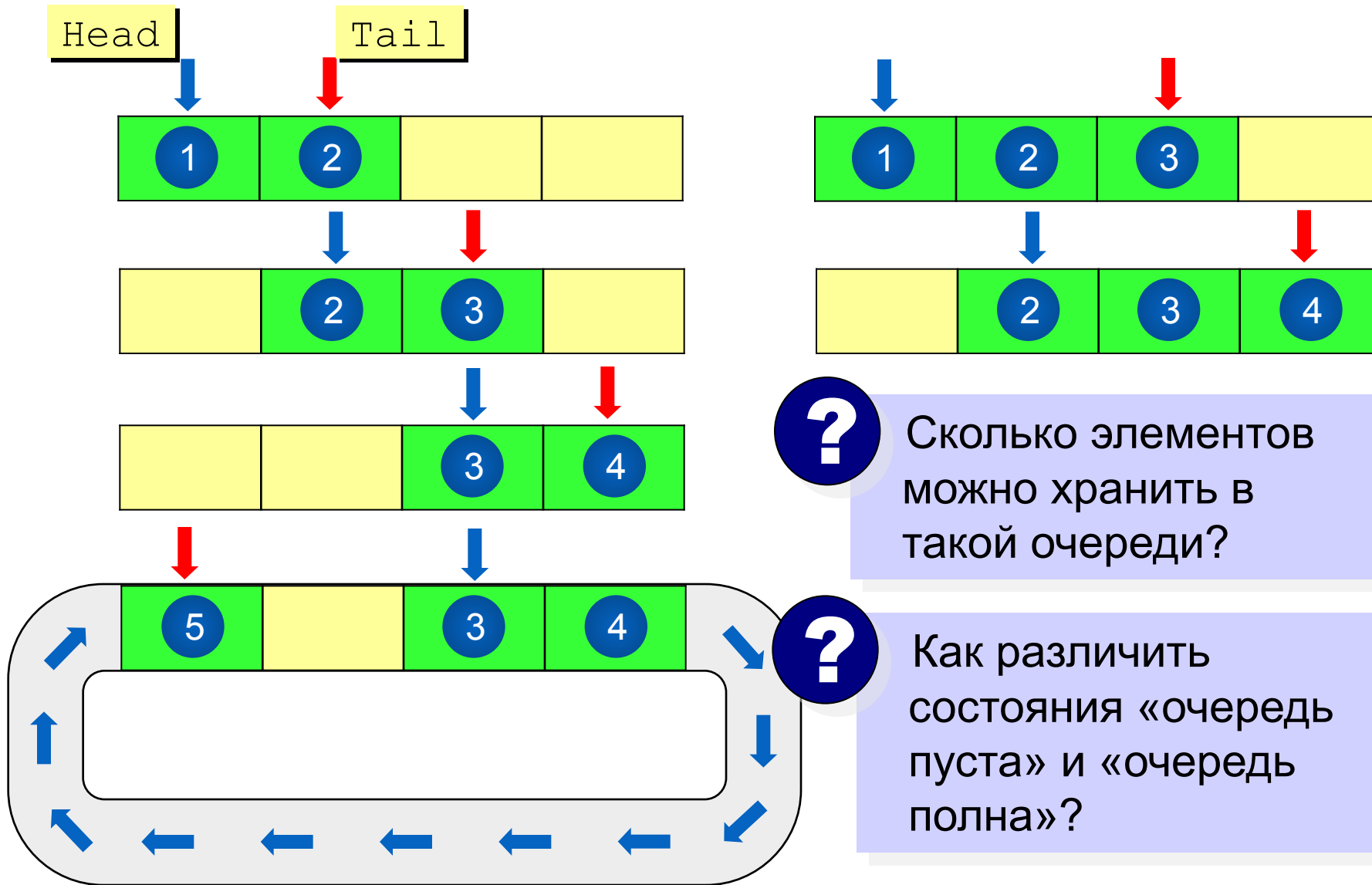


самый простой способ



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

Реализация очереди (кольцевой массив)



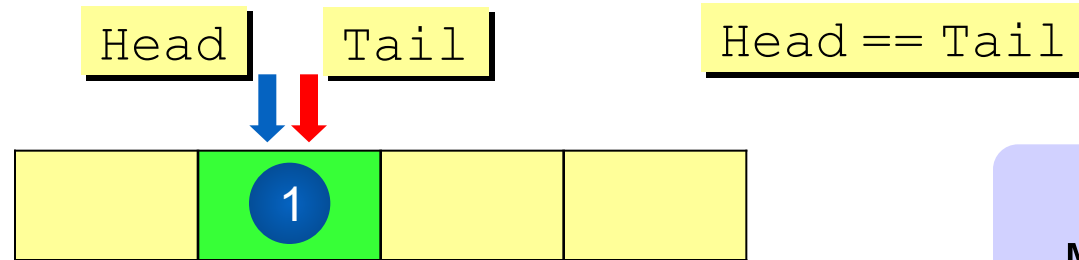
Сколько элементов можно хранить в такой очереди?



Как различить состояния «очередь пуста» и «очередь полна»?

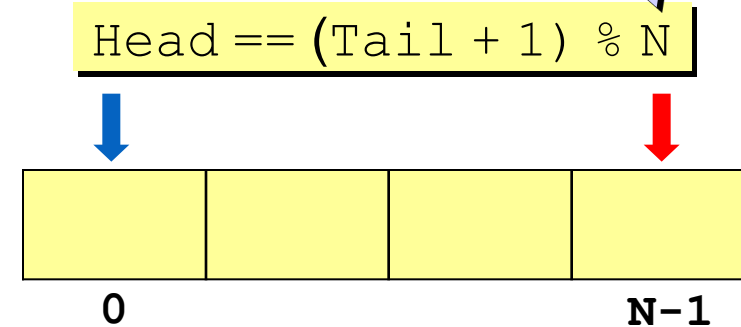
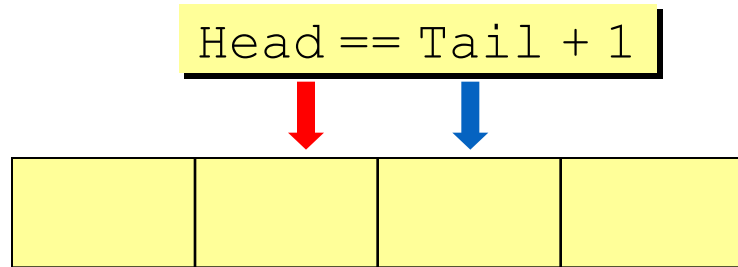
Реализация очереди (кольцевой массив)

В очереди 1 элемент:

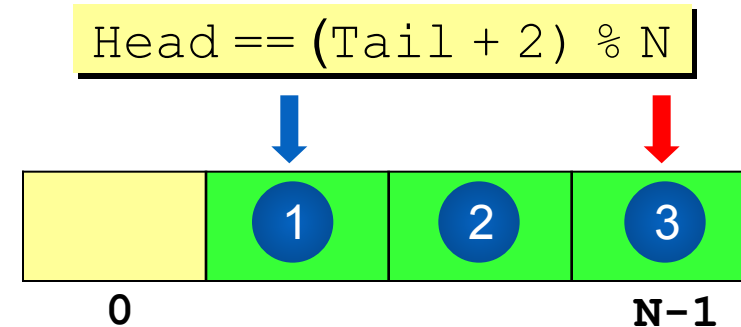
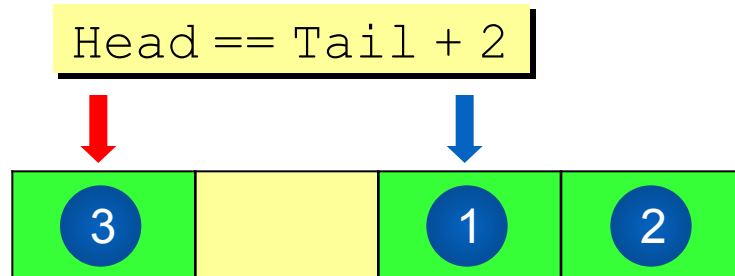


размер
массива

Очередь пуста:



Очередь полна:



Реализация очереди (кольцевой массив)

Структура данных:

```
const MAXSIZE = 100;
struct Queue {
    int data[MAXSIZE];
    int head, tail;
};
```

Добавление в очередь:

```
int PushTail ( Queue &Q, int x )
{
    if ( Q.head == (Q.tail+2) % MAXSIZE )
        return 0;
    Q.tail = (Q.tail + 1) % MAXSIZE;
    Q.data[Q.tail] = x;
    return 1;
}
```

замкнуть в
кольцо

очередь
полна, не
добавить

удачно добавили

Реализация очереди (кольцевой массив)

Выборка из очереди:

```
int Pop ( Queue &Q )  
{  
    int temp;  
    if ( Q.head == (Q.tail + 1) % MAXSIZE )  
        return 32767;  
    temp = Q.data[Q.head];  
    Q.head = (Q.head + 1) % MAXSIZE;  
    return temp;  
}
```

очередь пуста

взять первый
элемент

удалить его из
очереди

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 = new 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;  
    delete top;  
    return x;  
}
```

если список
пуст, ...

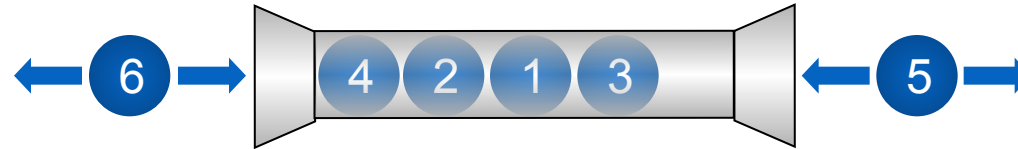
запомнили
первый элемент

если в списке
ничего не осталось,
...

освободить
память

Дек

Дек (*deque* = ***double ended queue***, очередь с двумя концами) – 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.

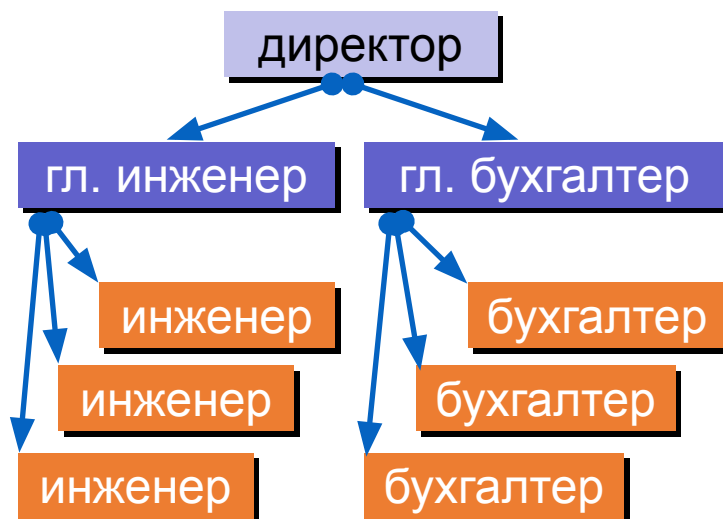
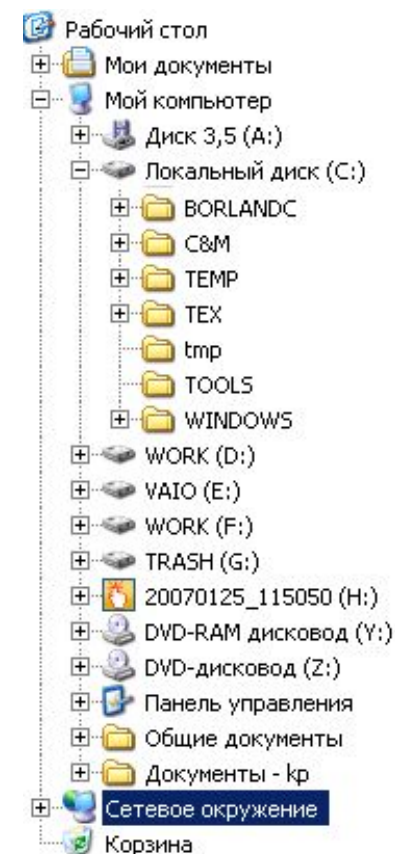
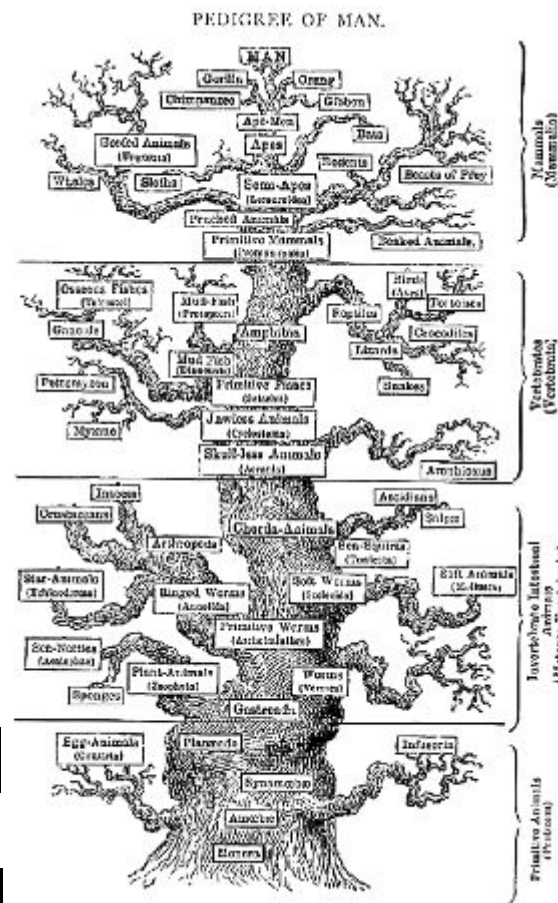
Задания

- «4»:** В файле `input.dat` находится список чисел (или слов). Переписать его в файл `output.dat` в обратном порядке.
- «5»:** Составить программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме, с помощью стека. Выражение правильное, допускаются только однозначные числа и знаки `+`, `-`, `*`, `/`.
- «6»:** То же самое, что и на «5», но допускаются многозначные числа.

Динамические структуры данных (язык Си)

Тема 6. Деревья

Деревья



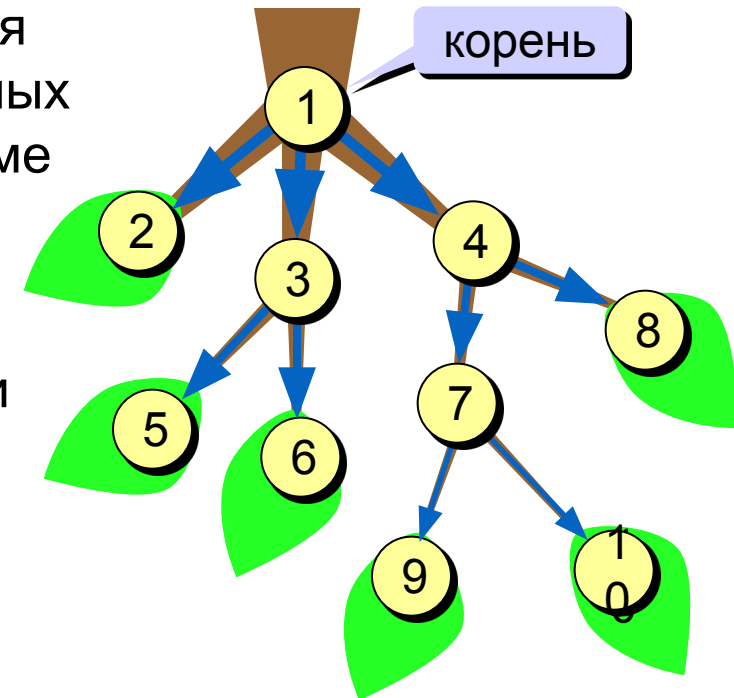
Что общего во всех примерах?

Деревья

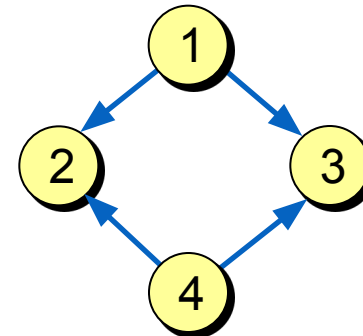
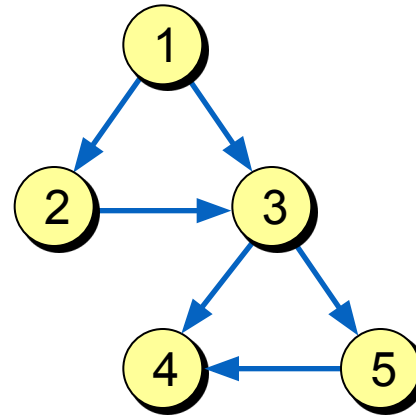
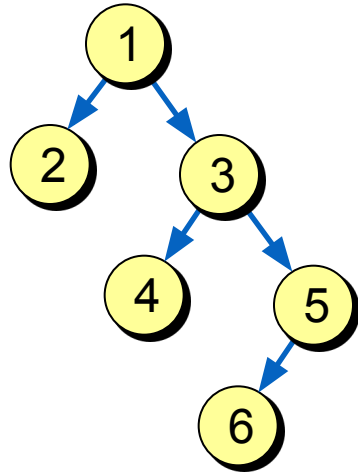
Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.

Корень – это начальный узел дерева.

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



Какие структуры – не деревья?



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 .

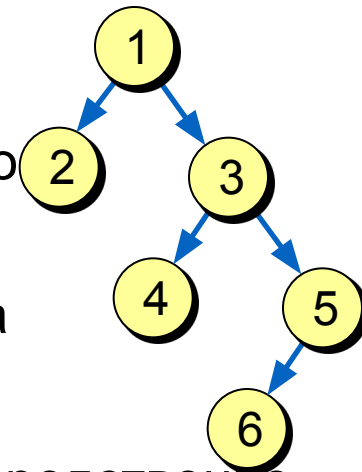
Потомок узла x – это узел, в который существует путь по стрелкам из узла x .

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 (*sibling*) – это узел, у которого тот же родитель, что и у узла x .

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



Двоичные деревья

Применение:

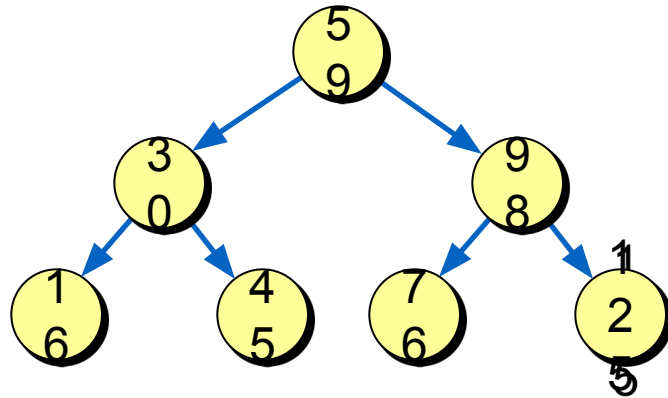
- 1) поиск данных в специально построенных деревьях (базы данных);
- 2) сортировка данных;
- 3) вычисление арифметических выражений;
- 4) кодирование (метод Хаффмана).

Структура узла:

```
struct Node {  
    int data;           // полезные 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).



Какая закономерность?

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.



Сведение задачи к такой же задаче меньшей размерности – это ...?

Реализация алгоритма поиска

```
//-----  
// Функция Search – поиск по дереву  
// Вход: Tree – адрес корня,  
//       x – что ищем  
// 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

Как построить дерево поиска?

```
//-----  
// Функция AddToTree - добавить элемент к дереву  
// Вход: Tree - адрес корня,  
//       x   - что добавляем  
//-----  
void AddToTree (PNode &Tree, int x)  
{  
    if ( ! Tree ) {  
        Tree = new Node;  
        Tree->data = x;  
        Tree->left = NULL;  
        Tree->right = NULL;  
        return;  
    }  
    if ( x < Tree->data )  
        AddToTree ( Tree->left, x );  
    else AddToTree ( Tree->right, x );  
}
```

адрес корня может
измениться

дерево пустое: создаем
новый узел (корень)

добавляем к левому или
правому поддереву

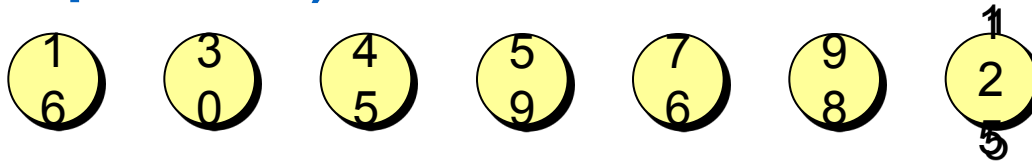


Минимальная высота не гарантируется!

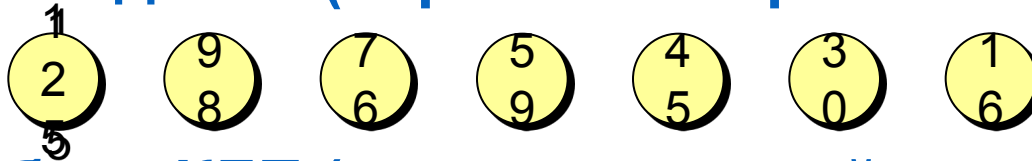
Обход дерева

Обход дерева – это перечисление всех узлов в определенном порядке.

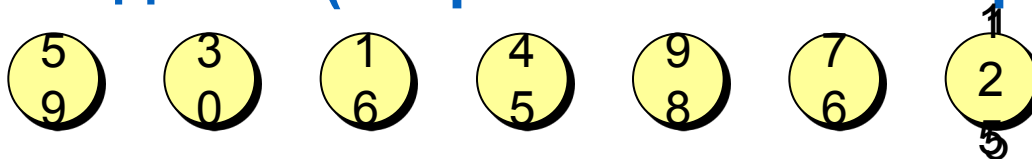
Обход ЛКП («левый – корень – правый»):



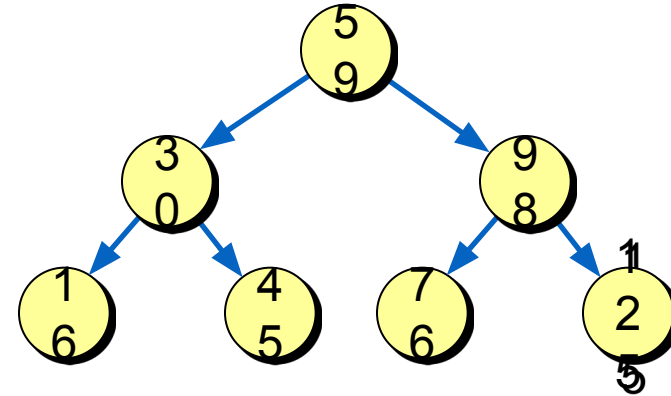
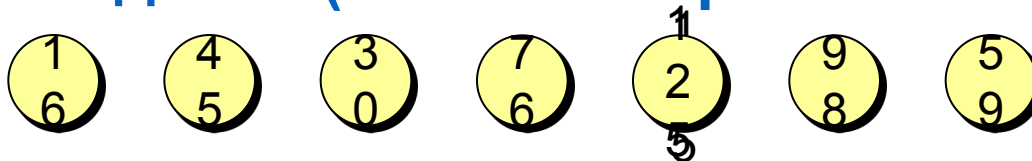
Обход ПКЛ («правый – корень – левый»):



Обход КЛП («корень – левый – правый»):



Обход ЛПК («левый – правый – корень»):



Обход дерева – реализация

```
//-----  
//  Функция LKP – обход дерева в порядке ЛКП  
//                      (левый – корень – правый)  
//  Вход: Tree – адрес корня  
//-----  
void LKP ( PNode Tree )  
{  
    if ( ! Tree ) return;  
    LKP ( Tree->left );  
    printf ( "%d ", Tree->data );  
    LKP ( Tree->right );  
}
```

обход этой ветки закончен

обход левого поддерева

вывод данных корня

обход правого поддерева

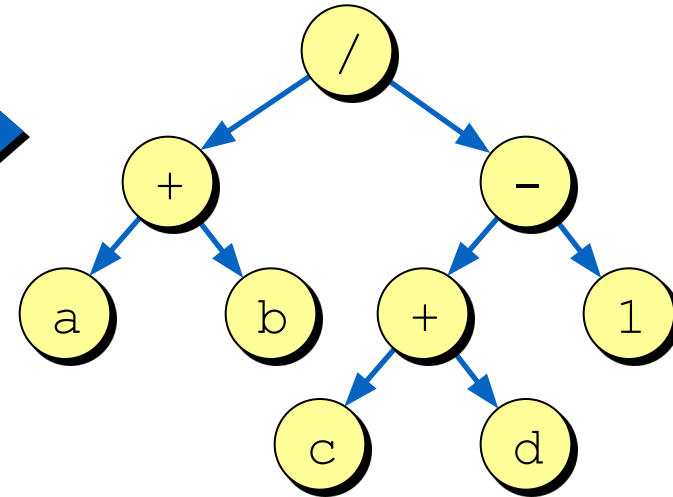


Для рекурсивной структуры удобно применять рекурсивную обработку!

Разбор арифметических выражений

Как вычислять автоматически:

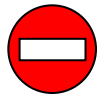
$(a + b) / (c + d - 1)$



Инфиксная запись, обход ЛКП

(знак операции **между** операндами)

$a + b / c + d - 1$



необходимы скобки!

Префиксная запись, КЛП (знак операции **до** операндов)

$/ + a b - + c d 1$

польская нотация,
[Jan Łukasiewicz](#) (1920)



скобки не нужны, можно однозначно вычислить!

Постфиксная запись, ЛПК (знак операции **после** операндов)

$a b + c d + 1 - /$

обратная польская нотация,
[F. L. Bauer](#) and [E. W. Dijkstra](#)

Вычисление выражений

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.

Вычисление выражений

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

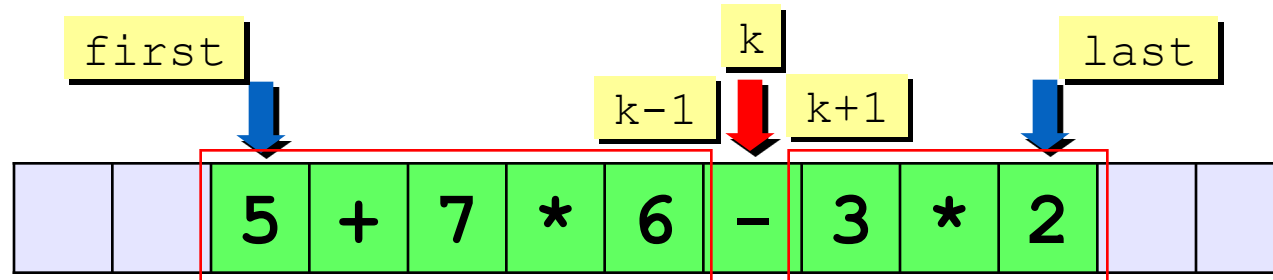
Алгоритм:

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

Ограничения:

- 1) ошибки не обрабатываем;
- 2) многозначные числа не разрешены;
- 3) дробные числа не разрешены;
- 4) скобки не разрешены.

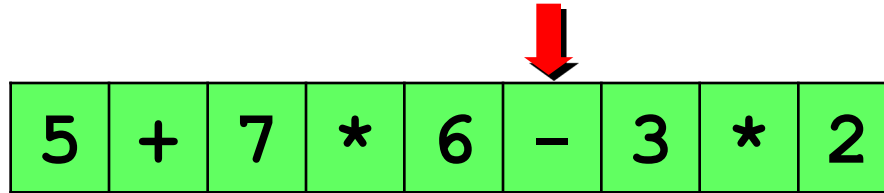
Построение дерева



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 (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`.

Как найти последнюю операцию?



Порядок выполнения операций

- умножение и деление;
- сложение и вычитание.

Приоритет (старшинство) – число, определяющее последовательность выполнения операций: раньше выполняются операции с большим приоритетом:

- умножение и деление (приоритет **2**);
- сложение и вычитание (приоритет **1**).



Нужно искать последнюю операцию с наименьшим приоритетом!

Приоритет операции

```
//-----  
// Функция Priority - приоритет операции  
// Вход:  символ операции  
// Выход: приоритет или 100, если не операция  
//-----  
int Priority ( char c )  
{  
    switch ( c ) {  
        case '+': case '-':  
            return 1;  
        case '*': case '/':  
            return 2;  
    }  
    return 100;  
}
```

сложение и вычитание:
приоритет 1

умножение и деление:
приоритет 2

это вообще не операция

Номер последней операции

```
//-----  
// Функция LastOperation – номер последней операции  
// Вход: строка, номера первого и последнего  
//       символов рассматриваемой части  
// Выход: номер символа – последней операции  
//-----  
int LastOperation ( char Expr[], int first, int last )  
{  
    int MinPrt, i, k, prt;  
    MinPrt = 100;  
    for( i=first; i <= last; i++ ) {  
        prt = Priority ( Expr[i] );  
        if ( prt <= MinPrt ) {  
            MinPrt = prt;  
            k = i;  
        }  
    }  
    return k;  
}
```

проверяем все
СИМВОЛЫ

нашли операцию с
минимальным
приоритетом

вернуть номер
символа

Построение дерева

Структура узла

```
struct Node {  
    char data;  
    Node *left, *right;  
};  
typedef Node *PNode;
```

Создание узла для числа (без потомков)

```
PNode NumberNode ( char c )  
{  
    PNode Tree = new Node;  
    Tree->data = c;  
    Tree->left = NULL;  
    Tree->right = NULL;  
    return Tree;  
}
```

ОДИН СИМВОЛ, ЧИСЛО

возвращает адрес
созданного узла

Построение дерева

```
//-----  
//  Функция MakeTree - построение дерева  
//  Вход:  строка, номера первого и последнего  
//         символов рассматриваемой части  
//  Выход: адрес построенного дерева  
//-----  
PNode MakeTree ( char Expr[], int first, int last )  
{  
    PNode Tree;  
    int k;  
    if ( first == last )  
        return NumberNode ( Expr[first] );  
    k = LastOperation ( Expr, first, last );  
    Tree = new Node;  
    Tree->data  = Expr[k];  
    Tree->left  = MakeTree ( Expr, first, k-1 );  
    Tree->right = MakeTree ( Expr, k+1, last );  
    return Tree;  
}
```

осталось
только число

новый узел:
операция

Вычисление выражения по дереву

```
//-----  
// Функция CalcTree - вычисление по дереву  
// Вход:  адрес дерева  
// Выход: значение выражения  
//-----  
int CalcTree (PNode Tree)  
{  
    int num1, num2;  
    if ( ! Tree->left ) return Tree->data - '0';  
    num1 = CalcTree( Tree->left);  
    num2 = CalcTree(Tree->right);  
    switch ( Tree->data ) {  
        case '+': return  num1+num2;  
        case '-': return  num1-num2;  
        case '*': return  num1*num2;  
        case '/': return  num1/num2;  
    }  
    return 32767;  
}
```

вернуть число,
если это лист

вычисляем
операнды
(поддерева)

выполняем
операцию

некорректная
операция

Основная программа

```
//-----  
// Основная программа: ввод и вычисление  
// выражения с помощью дерева  
//-----  
void main()  
{  
    char s[80];  
    PNode Tree;  
    printf ( "Введите выражение > " );  
    gets(s);  
    Tree = MakeTree ( s, 0, strlen(s)-1 );  
    printf ( "= %d \n", CalcTree ( Tree ) );  
    getch();  
}
```

Дерево игры

Задача.

Перед двумя игроками лежат две кучки камней, в первой из которых 3, а во второй – 2 камня. У каждого игрока неограниченно много камней.

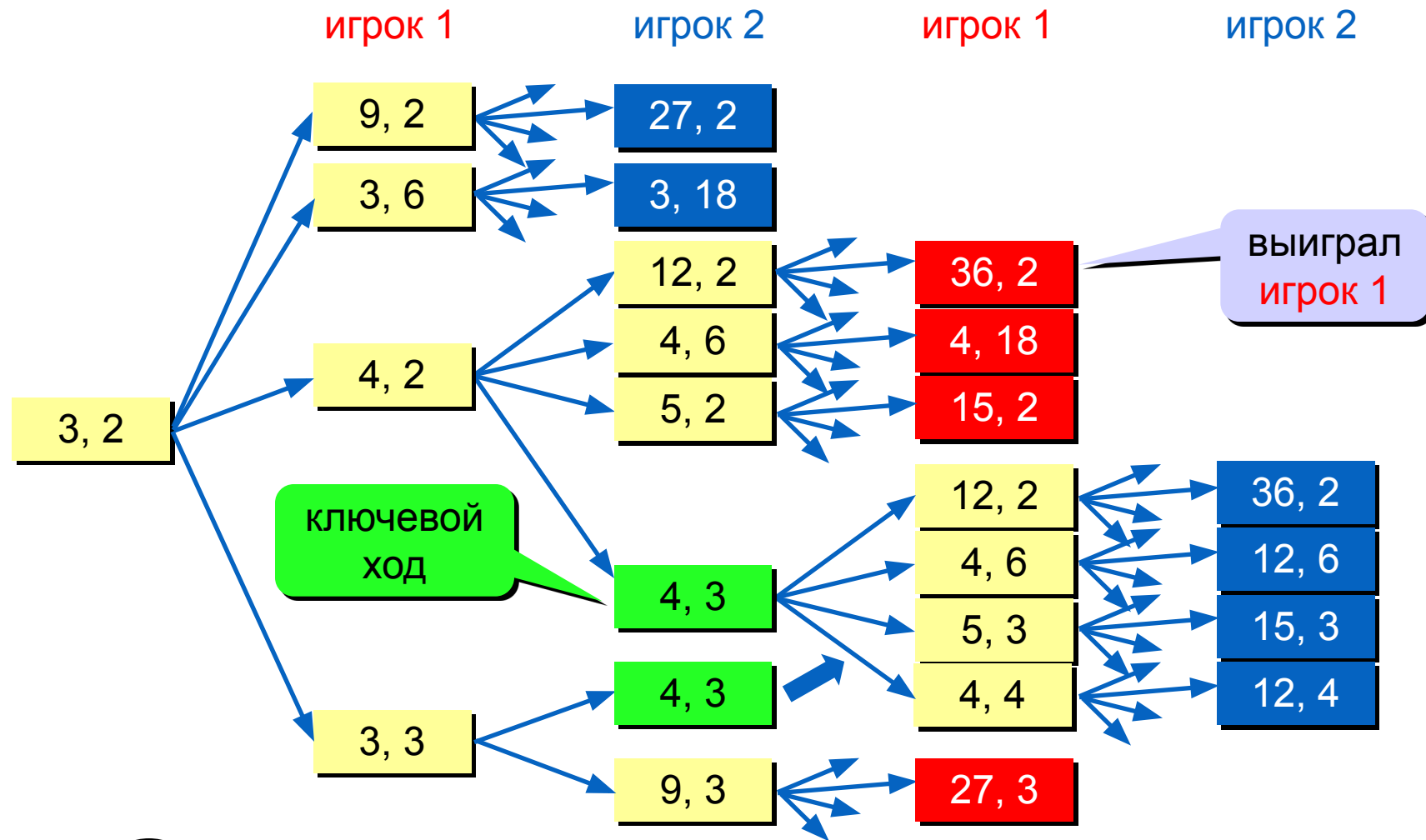
Игроки ходят по очереди. Ход состоит в том, что игрок или **увеличивает в 3 раза** число камней в какой-то куче, или **добавляет 1 камень** в какую-то кучу.

Выигрывает игрок, после хода которого общее число камней в двух кучах становится **не менее 16**.

Кто выигрывает при безошибочной игре – игрок, делающий первый ход, или игрок, делающий второй ход? Как должен ходить выигрывающий игрок?



Дерево игры



При правильной игре выиграет игрок 2!

Задания

- «4»:** «Собрать» программу для вычисления правильного арифметического выражения, включающего только однозначные числа и знаки операций $+$, $-$, $*$, $/$.
- «5»:** То же самое, но допускаются также многозначные числа и скобки.
- «6»:** То же самое, что и на «5», но с обработкой ошибок (должно выводиться сообщение).