ФГБОУ ВО РГУПС

Алгоритмизация и программирование

Динамические структуры данных. Деревья

Лекция 9

© Составление,
О.В. Игнатьева

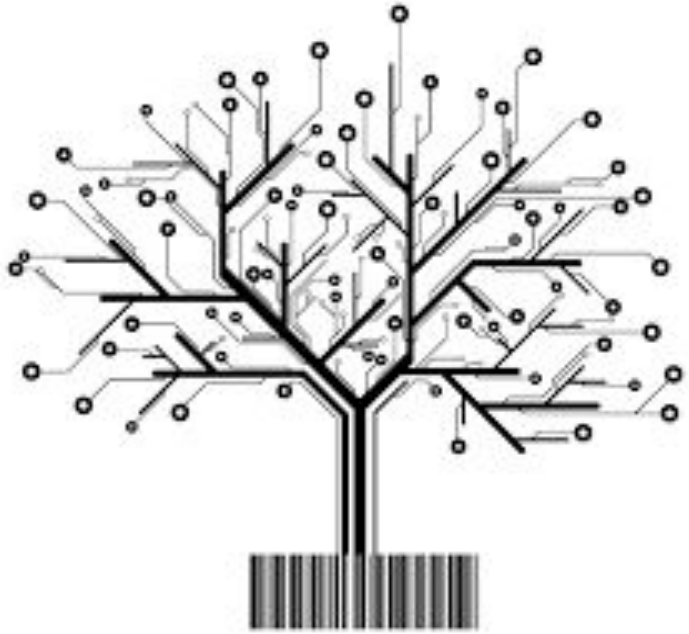
Ростов-на-Дону
2020

План лекции

- Понятие дерева
- Классификация
- Бинарное дерево

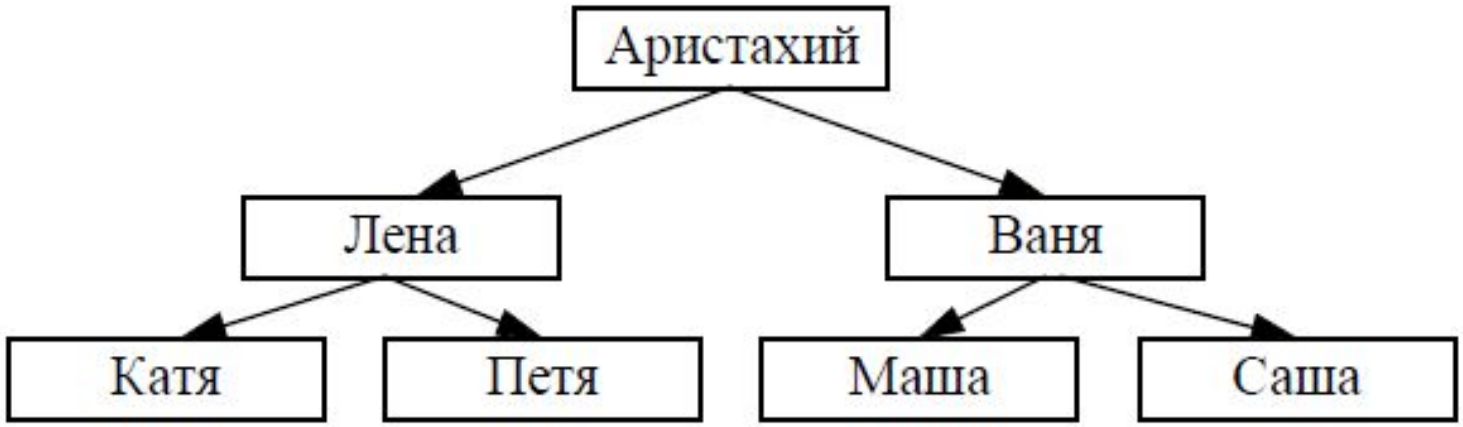
Деревья

- **Дерево –** это совокупность **узлов** (**вершин**) и соединяющих их направленных **ребер** (**дуг**), причем в каждый узел (за исключением одного **- корня**) ведет ровно одна дуга.
- **Корень –** это начальный узел дерева, в который не ведет ни одной дуги.



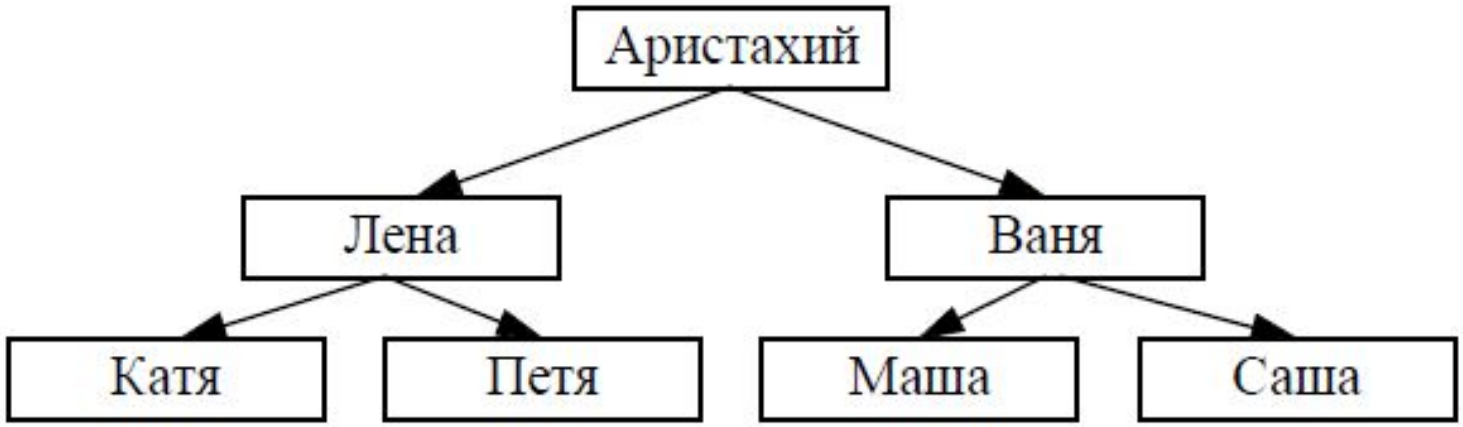
Деревья

- Примером может служить **генеалогическое дерево** - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.



Деревья

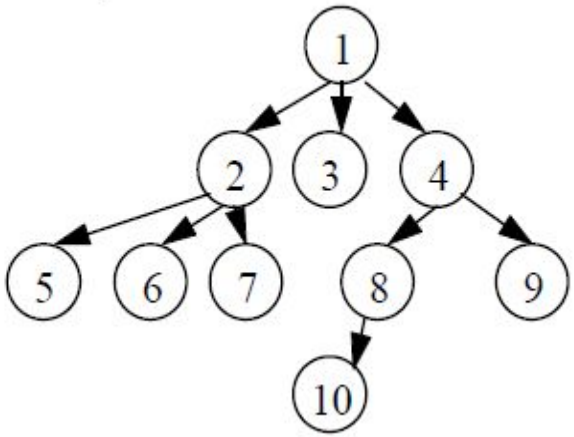
- Примером может служить **генеалогическое дерево** - в корне дерева находитесь вы сами, от вас идет две дуги к родителям, от каждого из родителей - две дуги к их родителям и т.д.



Деревья

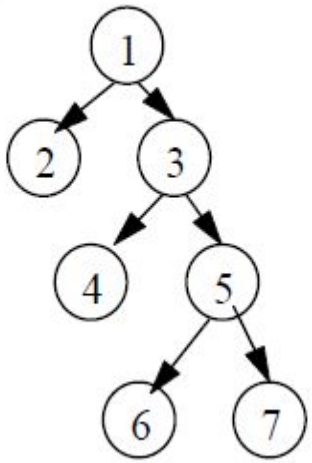
- Например, на рисунке структуры – кто из них является деревом?

а)



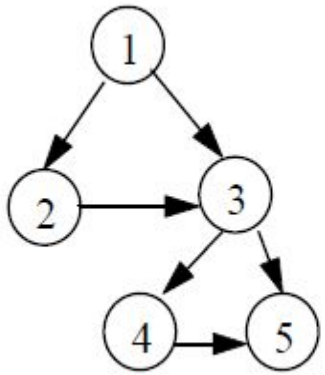
Дерево

б)



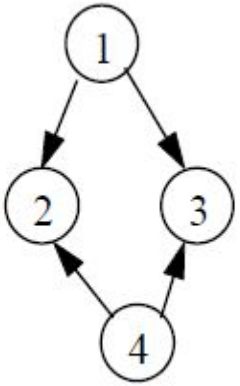
Дерево

в)



Нет

г)



Нет

Деревья. Базовые понятия

- **Предком** для узла ***x*** называется узел дерева, из которого существует путь в узел ***x***.
- **Потомком** узла ***x*** называется узел дерева, в который существует путь (по стрелкам) из узла ***x***.
- **Родителем** для узла ***x*** называется узел дерева, из которого существует непосредственная дуга в узел ***x***.
- **Сыном** узла ***x*** называется узел дерева, в который существует непосредственная дуга из узла ***x***.
- **Уровнем** узла ***x*** называется длина пути (количество дуг) от корня к данному узлу. Считается, что корень находится на уровне 0.

Деревья. Базовые понятия

- **Листом дерева** называется узел, не имеющий потомков.
- **Внутренней вершиной** называется узел, имеющий потомков.
- **Высотой дерева** называется максимальный уровень листа дерева.
- **Упорядоченным деревом** называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

Деревья. Рекурсивное определение

- **Дерево представляет собой типичную рекурсивную структуру (определяемую через саму себя).**
- Как и любое рекурсивное определение, определение дерева состоит из двух частей – первая определяет условие окончания рекурсии, а второе – механизм ее использования.

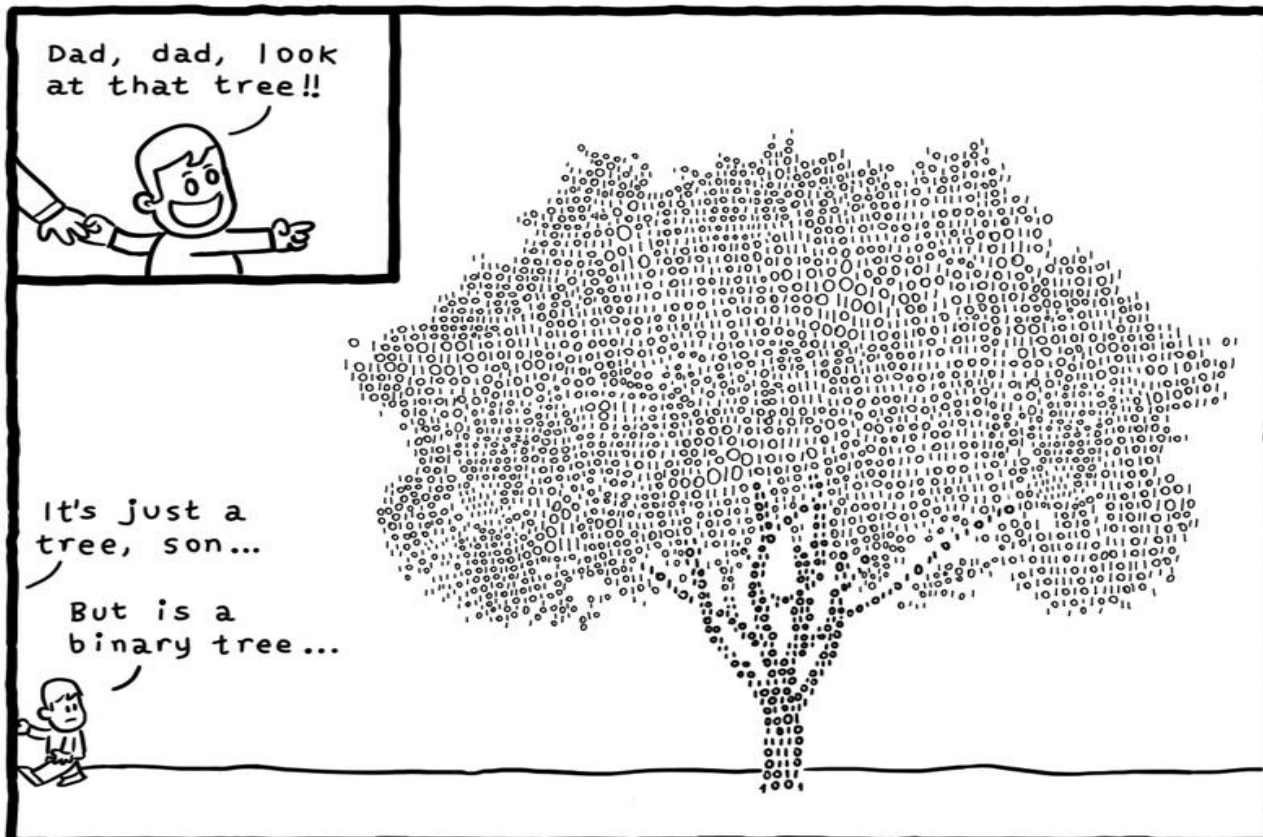
  1) пустая структура является деревом;

  2) дерево – это корень и несколько связанных с ним деревьев (поддеревьев).
- Таким образом, размер памяти, необходимый для хранения дерева, заранее неизвестен, потому что неизвестно, сколько узлов будет в него входить.

Двоичные (бинарные) деревья. Определение

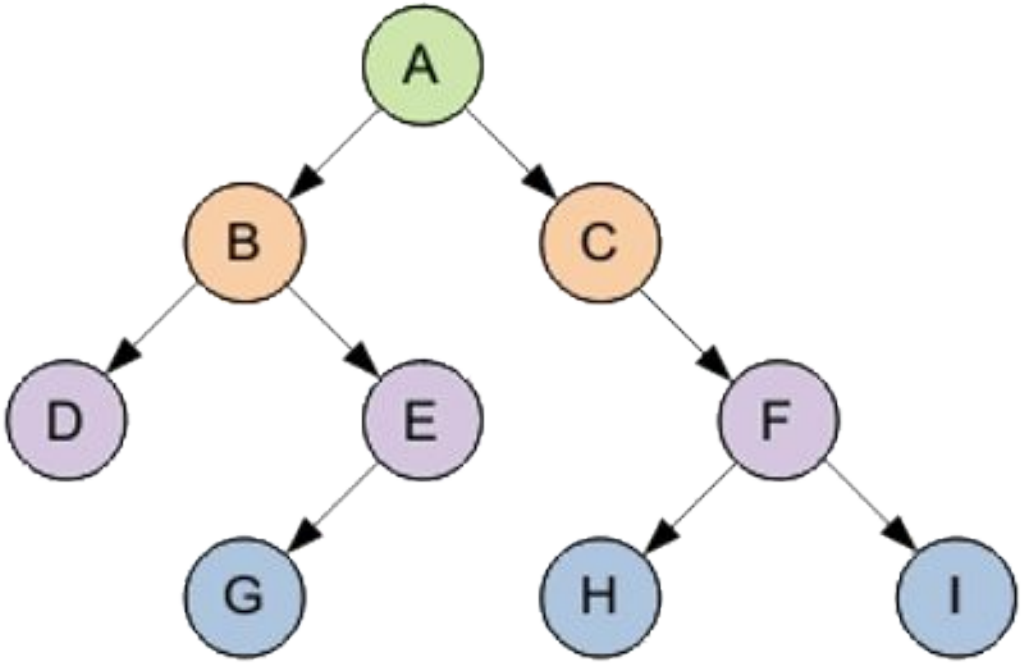
Двоичные деревья

- **Двоичным деревом** называется дерево, каждый узел которого имеет не более двух сыновей.



Двоичные деревья

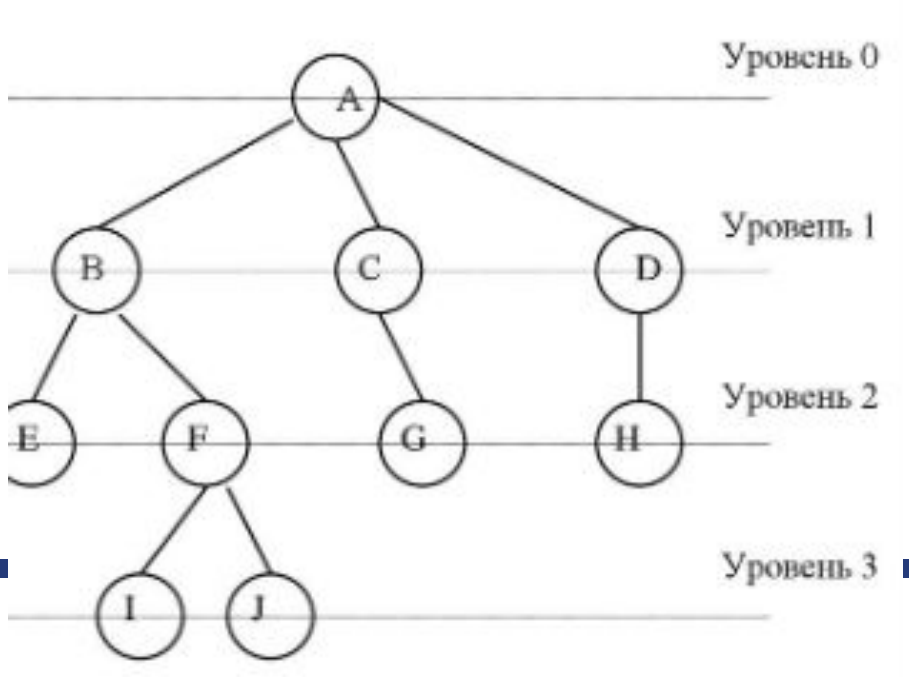
- На практике используются главным образом деревья особого вида, называемые **двоичными (бинарными).**



Двоичные деревья

Можно определить двоичное дерево и рекурсивно:

- 1) пустая структура является двоичным деревом;
- 2) дерево – это корень и два связанных с ним двоичных дерева, которые называют **левым** и **правым** поддеревом .



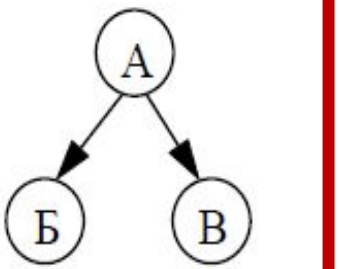
Двоичные деревья

- Двоичные деревья **упорядочены**, то есть различают левое и правое поддеревья.
- Двоичные деревья используются тогда, когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных.
- **Строго двоичным деревом** называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья. Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево.
- **Полным двоичным деревом** называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Двоичные деревья

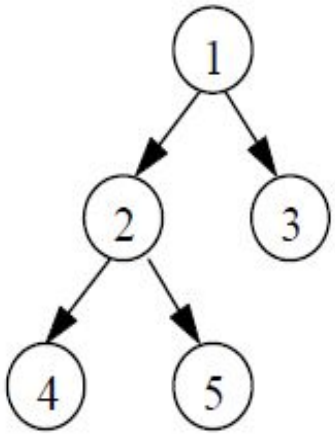
- На рисунке даны деревья. Какие из них строго двоичные ?
- Какие из них полные двоичные деревья?

а)



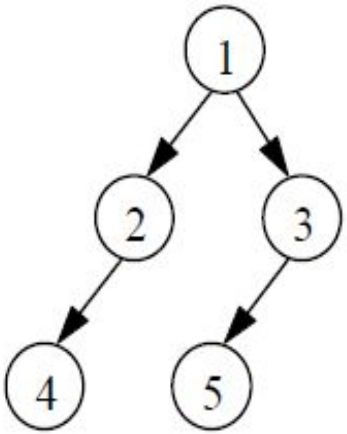
да

б)



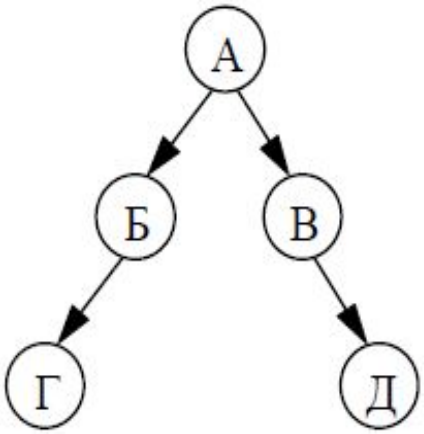
да

в)



нет

г)



нет

Реализация двоичных деревьях в C++

Двоичные деревья

- Вершина дерева, как и узел любой динамической структуры, имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним.
- Для двоичного дерева таких ссылок будет две – ссылка на **левого сына** и ссылка на **правого сына**.

Синтаксис, описывающий вершину и включает полезные данные и указатели на левое и правое поддерево:

```
struct Node
{
    Тип1 Поле1;
    Тип2 Поле2;
    ……..
    Node *left;
    Node *right;
};
typedef Node *PNode;
```

// полезные данные

// указатели на сыновей

// указатель на вершину

Двоичные деревья

- Например, опишем дерево в виде последовательности чисел. В результате получаем структуру, описывающую вершину:

```
struct Node
{
    int key;          // полезные данные (ключ)
    Node *left;       // указатели на сыновей
    Node *right;
};
typedef Node *PNode;  // указатель на вершину
```

Деревья минимальной высоты

- Для большинства практических задач наиболее интересны такие деревья, которые имеют минимально возможную высоту при заданном количестве вершин **n**.
- Очевидно, что минимальная высота достигается тогда, когда на каждом уровне (кроме, возможно, последнего) будет максимально возможное число вершин.

Деревья минимальной высоты

Предположим, что задано ***n*** чисел (их количество заранее известно). Требуется построить из них дерево минимальной высоты.

Алгоритм решения этой задачи:

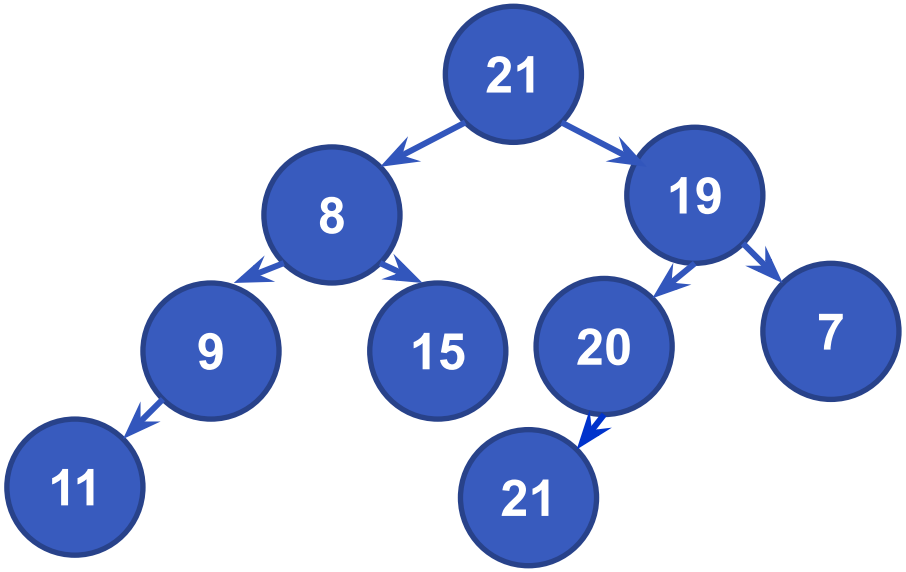
- 1. Взять одну вершину в качестве корня и записать в нее первое нерассмотренное число.
- 2. Построить этим же способом левое поддерево из **n1 = n/2** вершин (деление нацело!).
- 3. Построить этим же способом правое поддерево из **n2 = n-n1-1** вершин.

Деревья минимальной высоты

- Для массива данных

21, 8, 9, 11, 15, 19, 20, 21, 7

по этому алгоритму строится дерево



Заметим, что по построению левое поддерево всегда будет содержать столько же вершин, сколько правое поддерево, или на 1 больше.

Деревья минимальной высоты. Алгоритм создания

- Вершины дерева должны создаваться динамически, надо **выделить память** под вершину и записать в поле данных нужное число. Затем из оставшихся чисел построить левое и правое поддеревья.
- В основной программе нам надо объявить указатель на корень нового дерева, задать мас-сив данных и вызвать функцию, возвращающую указатель на построенное дерево.

Деревья минимальной высоты. Алгоритм создания

В основной программе объявляем указатель на корень нового дерева, заем массив данных и вызываем функцию, возвращающую указатель на построенное дерево.

```
int data[] = { 21, 8, 9, 11, 15, 19, 20, 21, 7 };
PNode Tree;                          // указатель на корень дерева
n = sizeof(data) / sizeof(int) - 1; // размер массива

Tree = MakeTree (data, 0, n);   // использовать n элементов,
                                    // начиная с номера 0
```

Деревья минимальной высоты. Алгоритм создания

Функция MakeTree принимает три параме
первого неиспользованного элемента и коли
дереве. Возвращает она указатель на новое

```
PNode MakeTree (int data[], int from, ir
{
        PNode Tree;
        int n1, n2;
        if ( n == 0 ) return NULL;              //
        Tree = new Node;        // выделит
        Tree->key = data[from];             // з
        n1 = n / 2;                          // размеры поддеревьев
        n2 = n - n1 - 1;
        Tree->left = MakeTree(data, from+1, n1);
        Tree->right = MakeTree(data, from+1+n1, n2);
        return Tree;
}
```

Выделенные строчки программы содержат рекурсивные вызовы. При этом левое поддерево содержит **n1** элементов массива начиная с номера **from+1**, тогда как правое – **n2** элементов начиная с номера **from+1+n1**.

Деревья минимальной высоты. Обход дерева

- Одной из необходимых операций при работе с деревьями является **обход дерева**, во время которого надо посетить каждый узел по одному разу и (возможно) вывести информацию, содержащуюся в вершинах.
- Пусть в результате обхода надо напечатать значения поля данных всех вершин в определенном порядке. Существуют **три варианта обхода**:
  - 1) **КЛП (*корень – левое – правое*)**: сначала посещается корень (выводится информация о нем), затем левое поддерево, а затем – правое;
  - 2) **ЛКП (*левое – корень – правое*)**: сначала посещается левое поддерево, затем корень, а затем – правое;
  - 3) **ЛПК (*левое – правое – корень*)**: сначала посещается левое поддерево, затем правое, а затем – корень.

Деревья минимальной высоты. Обход дерева

- Алгоритм обхода дерева - рекурсивная функция просмотра дерева в порядке ЛКП.

```
void PrintLKP(PNode Tree)
{
if ( ! Tree ) return;    // пустое дерево – окончание рекурсии

    PrintLKP(Tree->left);        // обход левого поддерева

    cout<< Tree->key;            // вывод информации о корне

    PrintLKP(Tree->right);       // обход правого поддерева
}
```

Остальные варианты обхода программируются аналогично.

Деревья поиска

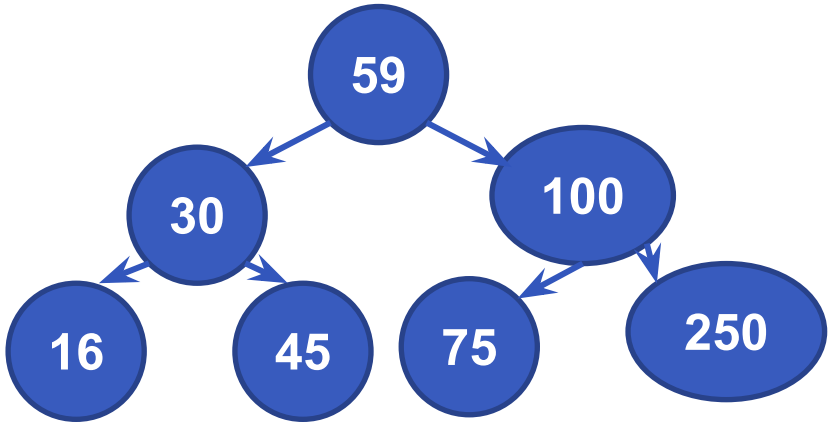
- Деревья очень удобны для поиска в них информации. Однако для быстрого поиска требуется предварительная подготовка – дерево надо построить специальным образом.
- **Дерево поиска** обладает следующим важным свойством: значения ключей всех вершин левого поддерева вершины **x** меньше ключа **x**, а значения ключей всех вершин правого поддерева **x** больше или равно ключу вершины **x.**

Деревья поиска

- Предположим, что существует массив данных и с каждым элементом связан ключ - число, по которому выполняется поиск. Пусть ключи для элементов таковы

59, 100, 75, 30, 16, 45, 250

по этому алгоритму строится дерево



Для этих данных нам надо много раз проверять, есть ли среди ключей заданный ключ x, и если есть, то вывести всю связанную с этим элементом информацию.

Если данные организованы в виде массива (без сортировки), то для поиска в худшем случае надо сделать **n** сравнений элементов.

Деревья поиска. Алгоритм построения

- 1. Сравнить ключ очередного элемента массива с ключом корня.
- 2. Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен, то в правое.
- 3. Если текущее дерево пустое, создать новую вершину и включить в дерево.

Деревья поиска. Алгоритм построения

```
void AddToTree (PNode &Tree, int data)
{
    if ( ! Tree ) {
     Tree = new Node;          // создать новый узел
     Tree->key = data;
     Tree->left = NULL;
     Tree->right = NULL;
      return;
    }
    if ( data < Tree->key )
        AddToTree ( Tree->left, data );
    else AddToTree ( Tree->right, data );
}
```

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента меньше, включить его в левое поддерево, если больше или равен, то в правое.

Деревья поиска. Алгоритм построения

- В результате работы этого алгоритма не всегда получается **дерево минимальной высоты** – все зависит от порядка выбора элементов.
- Для оптимизации поиска используют так называемые **сбалансированные** или **АВЛ-деревья** деревья, у которых для любой вершины высоты левого и правого поддеревьев отличаются не более, чем на 1.
- Добавление в них нового элемента иногда сопровождается некоторой **перестройкой дерева**.
- Сбалансированные деревья называют так в честь изобретателей этого метода **Г.М. Адельсона-Вельского и Е.М. Ландиса.**

Деревья поиска. Алгоритм поиска

- Теперь, когда дерево сортировки построено, очень легко искать элемент с заданным ключом.
- ***Алгоритм:***
  - сначала проверяем ключ корня, если он равен искомому, то нашли его;
  - если он меньше искомого, ищем в левом поддереве корня, если больше – то в правом.
- Приведенная функция возвращает адрес нужной вершины, если поиск успешный, и **NULL**, если требуемый элемент не найден.

Деревья поиска. Алгоритм поиска

Функция для поиска вершины в дереве поиска

```
PNode Search (PNode Tree, int what)
{
    if ( ! Tree ) return NULL;              // ключ не найден
    if ( what == Tree->key )
        return Tree;                        // ключ найден!
    if ( what < Tree->key )          // искать в поддеревьях
        return Search ( Tree->left, what );
    else return Search ( Tree->right, what );
}
```

Деревья поиска. Сортировка с помощью дерева поиска

Если дерево поиска построено, то очень просто вывести отсортированные данные.
Так,

- обход типа **ЛКП (*левое поддерево – корень – правое поддерево*)** даст ключи в порядке возрастания,
- а обход типа **ПКЛ (*правое поддерево – корень – левое поддерево*)** – в порядке убывания.

Деревья поиска. Поиск одинаковых элементов

- Приведенный алгоритм можно модифицировать так, чтобы быстро искать одинаковые элементы в массиве чисел. Конечно, можно перебрать все элементы массива и сравнить каждый со всеми остальными. Однако для этого требуется очень большое число сравнений.
- С помощью двоичного дерева можно значительно ускорить поиск.
- Для этого надо в структуру вершины включить еще одно поле – счетчик найденных дубликатов **count**.

Деревья поиска. Поиск одинаковых элементов

```
struct Node {
   int key;
   int count;              // счетчик дубликатов
   Node *left, *right;
};
```

При создании узла в счетчик записывается единица (найден один элемент).

Поиск дубликатов происходит по следующему алгоритму:

1. Сравнить ключ очередного элемента массива с ключом корня.
2. Если ключ нового элемента равен ключу корня, то увеличить счетчик корня и стоп.
3. Если ключ нового элемента меньше, чем у корня, включить его в левое поддерево, если
больше или равен – в правое.
4. Если текущее дерево пустое, создать новую вершину (со значением счетчика 1) и включить в дерево.

Пример. Создать дерево минимальной высоты

```
//структура с описанием вершины дерева
struct Node
{
  string name;              // область данных
  int year;
  Node *left, *right;       // ссылки на левое и правое дерево
  };
typedef Node *PNode;       //  указатель на дерево
// добавление элемента в дерево поиска, по годам издания
void AddToTree(PNode &Tree,  string Newname, int Newyear) {
……
}
//Вывод дерева по левому-корень-правое ЛКП
void PrintLKP(PNode Tree)
{
….
}
```

Пример. Создать дерево минимальной высоты

```
//Вывод дерева  с отступами
void Print(PNode Tree, int n)
{
    if ( Tree == NULL ) return;          // пустое дерево – окончание рекурсии

    if (Tree->left !=NULL) Print(Tree->left, n+1);

    for (int i = 0; i < n; i++) cout<<"***";

    cout<< Tree->year<<"\n";

    if (Tree->right !=NULL) Print(Tree->right, n+1);

}
```

Пример. Создать дерево минимальной высоты

```
// поиск по году
PNode Search (PNode Tree, int what)
{
if ( ! Tree ) return NULL;                          // ключ не найден

if ( what == Tree->year ) return Tree;              // ключ найден!

if ( what < Tree->year )                            // искать в поддеревьях
    return Search ( Tree->left, what );
else
      return Search ( Tree->right, what );
}
```

Пример. Создать дерево минимальной высоты

```
int _tmain(int argc, _TCHAR* argv[])
{
	//Объявляем переменную, тип которой структура  Дерево
	PNode Tree=NULL;

	PNode pnew, pfind;
	int t; string newname; int newyear;
	do
	{
	cout<<"введите от 1 до 5 или 0 - выход"<<endl;
	cout<<" 0 - выход "<<endl;
	cout<<" 1 - добавить новый узел в дерево по году издания"<<endl;
	cout<<" 2 - вывод дерева "<<endl;
	cout<<" 3 - поиск информации по году  "<<endl;
	cout<<" 4 - уровень дерева  "<<endl;
	cout<<" 5 - удаление узла "<<endl;
	cout<<" 6 - удаление дерева "<<endl;
	cout<<" 7 -   "<<endl;
	cout<<endl;
		cin>>t;
```

Пример. Создать дерево минимальной высоты

```
switch (t)
      {
      case 1 :

            cout<<"введите название книги = ";  cin>>newname;
            cout<<"введите год издания = ";       cin>>newyear;
            AddToTree(Tree, newname, newyear);
            break;
      case 2 :
            Print(Tree,0);
            break;

      case 3 :
            //поиск
            cout<<"введите год издания = ";  cin>>newyear;
                  pfind=Search(Tree, newyear);
                  cout<<"результат поиска"<<endl;
                  cout<<"название "<<pfind->name<<endl;
                  cout<<"год "<<pfind->year<<endl;
                        break;
```

Пример. Создать дерево минимальной высоты

Самостоятельно разработать функции для:

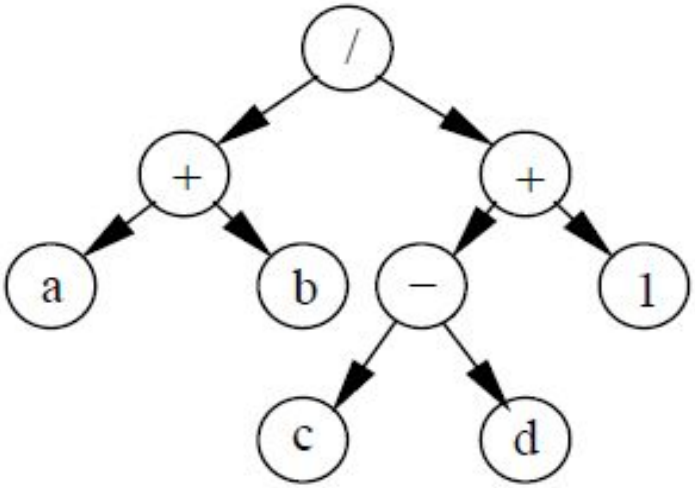
- Удаления узла из дерева
- Удаления поддерева
- Определения уровня дерева

Разбор арифметического выражения с помощью деревьев

- Как транслятор обрабатывает и выполняет арифметические и логические выражения, которые он встречает в программе? Один из вариантов – представить это выражение в виде двоичного дерева.
- Например, выражению

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

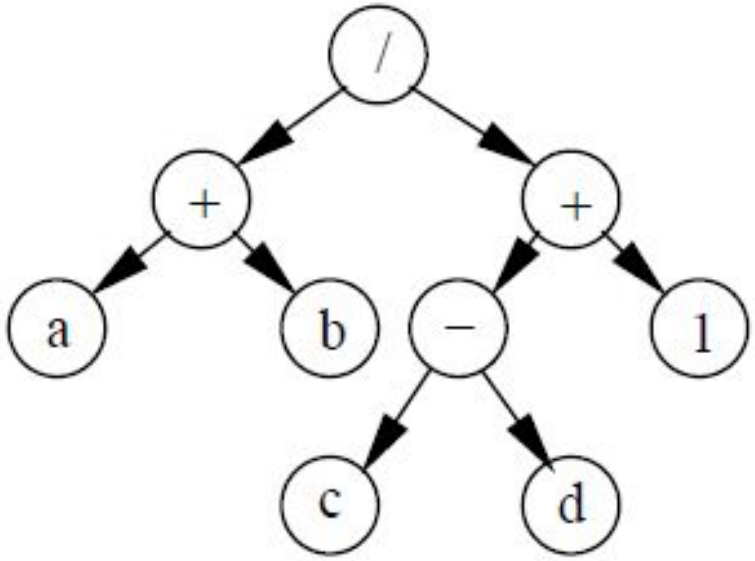
- соответствует дерево, показанное на рисунке.



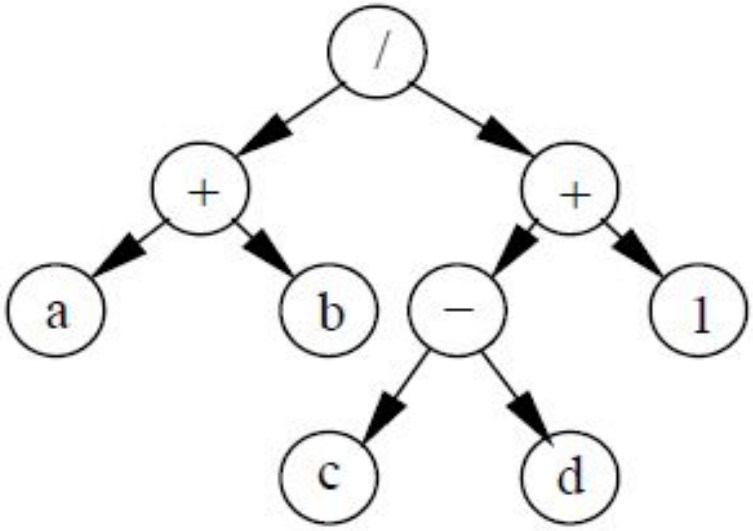
Разбор арифметического выражения с помощью деревьев

- Листья содержат числа и имена переменных (операндов), а внутренние вершины и корень – арифметические действия и вызовы функций. Вычисляется такое выражение снизу, начиная с листьев. Скобки отсутствуют, и дерево полностью определяет порядок выполнения операций.

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



Формы записи арифметического



- Теперь посмотрим, что

получается при прохождении

таких двоичных деревьев. Прохождение дерева в ширину (корень – левое – правое) дает

/ + a b + - c d 1

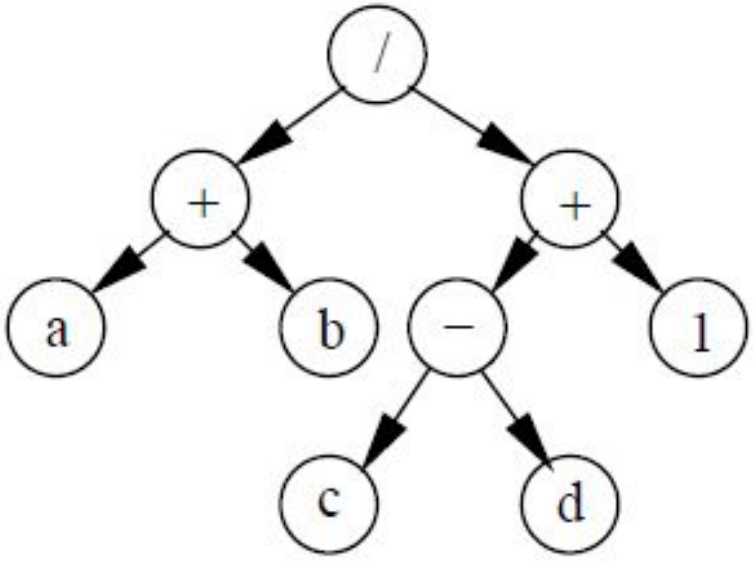
- то есть знак операции (корень) предшествует своим операндам. Такая форма записи арифметических выражений называется **префиксной**. Проход в прямом порядке (левое – корень – правое) дает **инфиксную форму**, которая совпадает с обычной записью, но без скобок:

a + b / c - d + 1

Формы записи арифметического выражения

- Поскольку скобок нет, по инфиксной записи невозможно восстановить правильный порядок операций.
- В трансляторах широко используется **постфиксная запись** выражений, которая получается в результате обхода в порядке **ЛПК (левое – правое – корень)**. В ней знак операции стоит **после** обоих операндов:

a b + c d - 1 /



Формы записи арифметического выражения

Порядок выполнения такого выражения однозначно определяется следующим алгоритмом, который использует стек:

Пока в постфиксной записи есть невыбранные элементы,

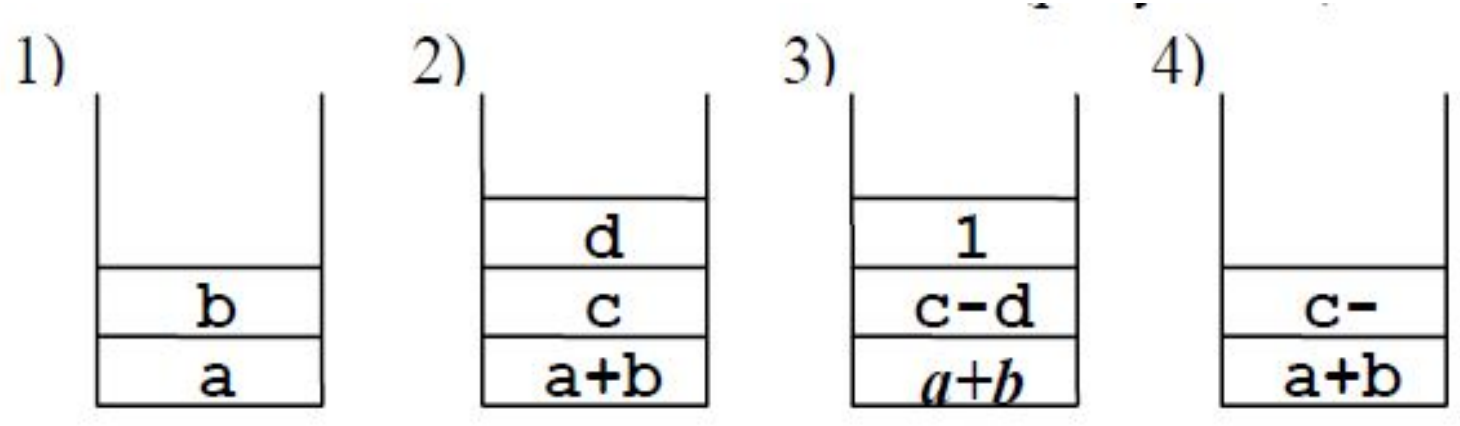
- 1) взять очередной элемент;
- 2) если это операнд (не знак операции), то записать его в стек;
- 3) если это знак операции, то
  - выбрать из стека второй операнд;
  - выбрать из стека первый операнд;
  - выполнить операцию с этими данными и результат записать в стек.

Формы записи арифметического выражения

- Проиллюстрируем на примере вычисление выражения в постфиксной форме

a b + c d - 1 /

- Сначала запишем в стек **a**, а затем **b** (рис. 1).
- Результат выполнения операции **a+b** запишем обратно в стек, а сверху – выбранные из входного потока значения переменных **c** и **d** (рисунок 2).
- Далее рис. 3 и 4. Выполнение последней операции (деления) для стека на рисунке 4 дает искомый результат.



Алгоритм построения дерева

- Пусть задано арифметическое выражение. Надо построить для него дерево синтаксического разбора и различные формы записи.
- Чтобы не слишком усложнять задачу, рассмотрим самый простой вариант, введя **следующие упрощения**.
  - 1. В выражении могут присутствовать только однозначные целые числа знаки операций **+ - * /**.
  - 2. Запрещается использование вызовов функций, скобок, унарных знаков плюс и минус (например, запрещено выражение **-a+5**, вместо него надо писать **0-a+5**).
  - 3. Предполагается, что выражение записано верно, то есть не делается проверки на правильность.

Алгоритм построения дерева

- Вспомним, что порядок выполнения операций в выражении определяется **приоритетом операций** – первыми выполняются операции с более высоким приоритетом. Например, умножение и деление выполняются раньше, чем сложение и вычитание.
- Будем правильное арифметическое выражение записано в виде символьной строки **Expr** длиной **N**. Построим дерево для элементов массива с номерами от **first** до **last** (полное дерево дает применение этого алгоритма ко всему массиву, то есть при **first=0** и **last=N-1**).

Алгоритм построения дерева

- В словесном виде алгоритм выглядит так:
- 1. Если **first=last** (остался один элемент – переменная или 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 (пусть найденный элемент имеет номер **k**).
- 3. Создать новый узел (корень) и записать в него знак операции **Expr[k]**.
- 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**

Алгоритм построения дерева

- Объявим структуру, описывающую узел такого дерева. Так как мы используем только однозначные целые числа и знаки, область данных может содержать один символ.

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

Далее надо определить функцию, возвращающую приоритет операции, которая ей передана. Определим приоритет 1 для сложения и вычитания и приоритет 2 для умножения и деления.

```
int Priority ( char c )
{
   switch ( c )
  {
     case '+': case '-': return 1;
     case '*': case '/': return 2;
   }
return 100;
}
```

Алгоритм построения дерева

- Функция **MakeTree** строит требуемое дерево.
- Используя эту функцию, и возвращает адрес построенного дерева в памяти.
- При сравнении приоритета текущей операции с минимальным предыдущим используется условие **<=**.
- За счет этого мы ищем именно **последнюю** операцию с минимальным приоритетом, то есть, операцию, которая будет выполняться самой последней. Если бы мы использовали знак **<**, то нашли бы **первую** операцию с наименьшим приоритетом, и дерево было бы построено неверно (вычисления дают неверный результат, если встречаются два знака вычитания или деления).

```
PNode MakeTree (char Expr[], int first, int last)
{
int MinPrt, i, k, prt;
PNode Tree = new Node; // создать в памяти новую вершину
if ( first == last ) { // конечная вершина: число или
Tree->data = Expr[first]; //
Tree->left = NULL;
Tree->right = NULL;
return Tree;
}
MinPrt = 100;
for ( i = first; i <= last; i ++ ) {
prt = Priority ( Expr[i] );
if ( prt <= MinPrt ) { // ищем последнюю операцию
MinPrt = prt; // с наименьшим приоритетом
k = i;
}
}
Tree->data = Expr[k]; // внутренняя вершина (операция)
Tree->left = MakeTree (Expr,first,k-1); // рекурсивно строим
Tree->right = MakeTree (Expr,k+1,last); // поддеревья
return Tree;
}
```

Теперь обход этого дерева разными способами дает различные формы представления соответствующего арифметического выражения.

Вычисление выражения по дереву

- Пусть для некоторого арифметического выражения построено дерево и известен его адрес **Tree**.
- Напишем функцию, которая возвращает целое число – результат вычисления этого выражения. Учтем, что деление выполняется нацело (остаток отбрасывается).

Вычисление выражения по дереву

```
int CalcTree (PNode Tree)
{
int num1, num2;
if ( ! Tree->left )                    // если нет потомков,
return Tree->data - '0';               // вернули число
num1 = CalcTree(Tree->left);           // вычисляем поддеревья
num2 = CalcTree(Tree->right);
switch ( Tree->data )                  // выполняем операцию
{
   case '+': return num1+num2;
   case '-': return num1-num2;
   case '*': return num1*num2;
   case '/': return num1/num2;
}
return 32767;                  // неизвестная операция, ошибка!
}
```

Вычисление выражения по дереву

Если дерево не имеет потомков, значит это число. Чтобы получить результат как целое число, из кода этой цифры надо вычесть код цифры **'0'**. Если потомки есть, вычисляем левое и правое поддеревья (рекурсивно!) и выполняем операцию, записанную в корне дерева. Основная программа может выглядеть так, как показано ниже.

```
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();
}
```

Спасибо за внимание!

Литературные источники

- 1) К. Поляков. Программирование на языке C++.
- 2). Харви Дейтел, Пол Дейтел. Как программировать на C++. - М: Вильямс, - 1011 с.
- 3). Структруп Б. Программирование: принципы и практика использования C++. – М. : Вильямс, 2011. – 1248 с.
- 4). Структруп Б. Язык программирования C++. – М.: Бином. - 1054 с.
- 5). Лафоре Р. Объектно-ориентированное программирование в C++. Питер, 2004. – 922 с.
- 6). Шилдт Г. C++: руководство для начинающих, 2-е издание. – М: Вильямс, 2005. -672 с.