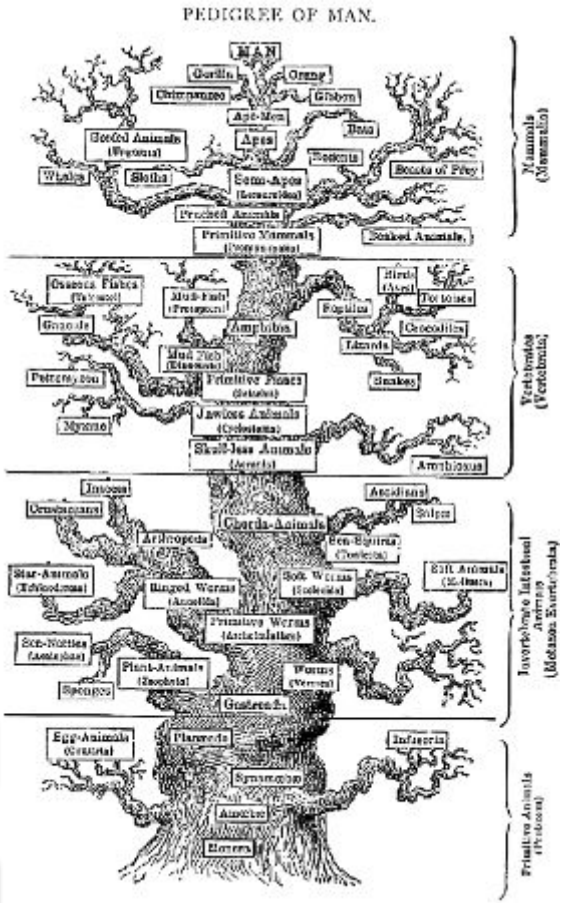
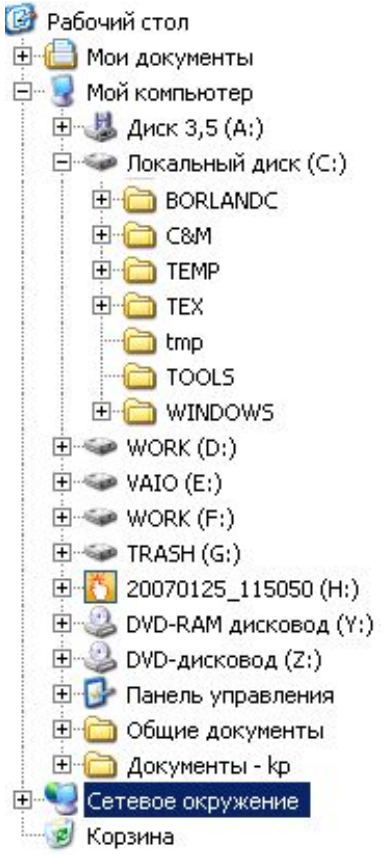
Динамические структуры данных

Деревья

Деревья





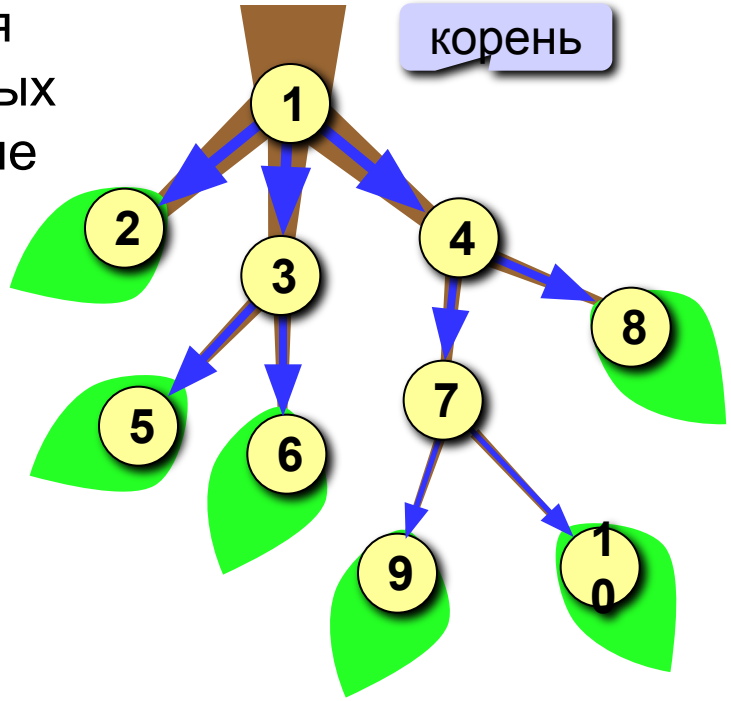
Что общего во всех примерах?

Деревья

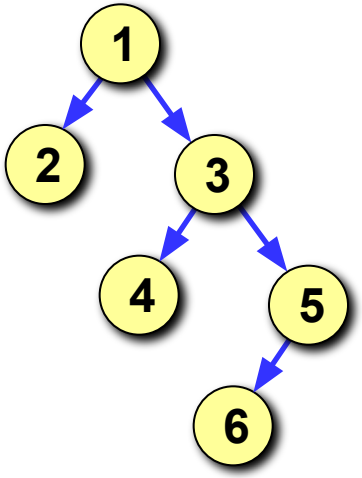
Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.

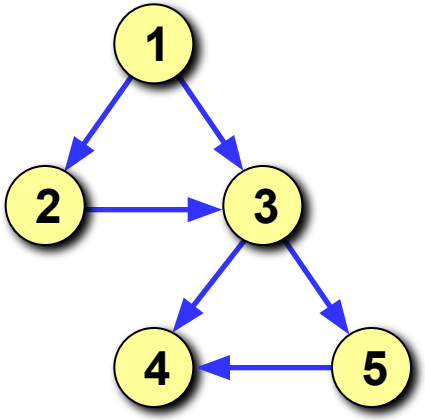
Корень – это начальный узел дерева.

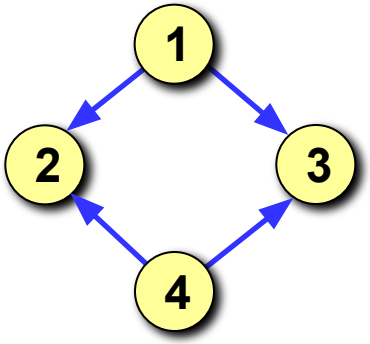
Лист – это узел, из которого не выходит ни одной дуги.



Какие структуры – не деревья?







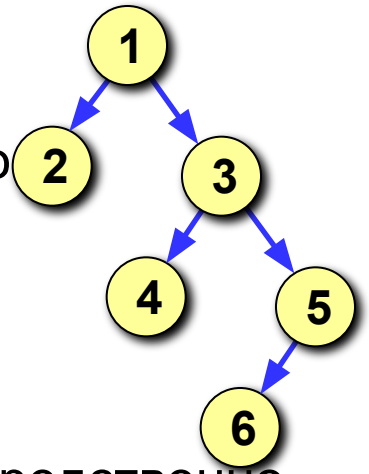
Деревья

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***.

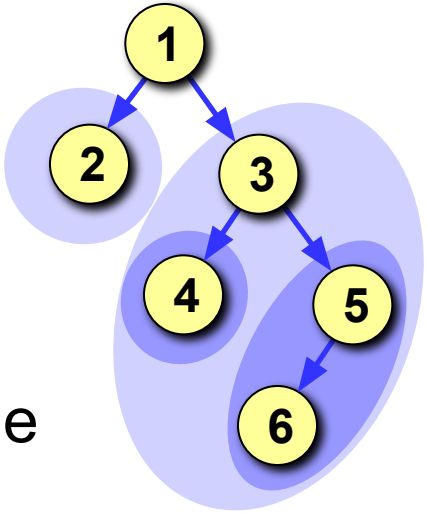
Брат узла *x (sibling)* – это узел, у которого тот же родитель, что и у узла ***x***.

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

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;
4) кодирование (метод Хаффмана).

Структура узла:

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

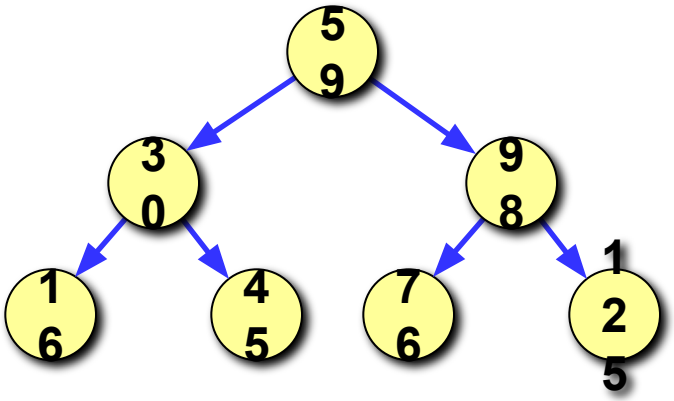
Двоичные деревья

Многие полезные структуры данных основаны на двоичном дереве:

- Двоичное дерево поиска
- Двоичная куча
- АВЛ-дерево
- Красно-чёрное дерево
- Матричное дерево
- Дерево Фибоначчи
- Суффиксное дерево

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 …?

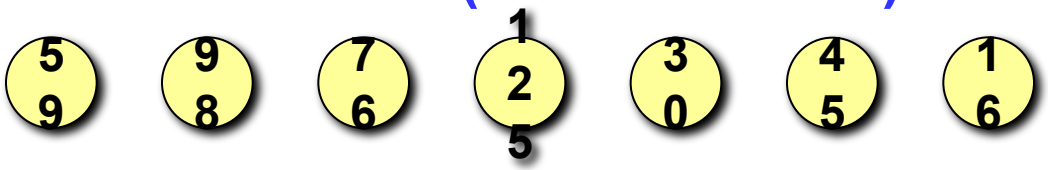
Двоичные деревья поиска

Двоичное дерево поиска— это двоичное дерево, для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

- ✔ **Оба поддерева — левое и правое, являются двоичными деревьями поиска.**
- ✔ **У всех узлов левого поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных узла X.**
- ✔ **У всех узлов правого поддерева произвольного узла X значения ключей данных *не меньше*, нежели значение ключа данных узла X.**

Двоичные деревья поиска

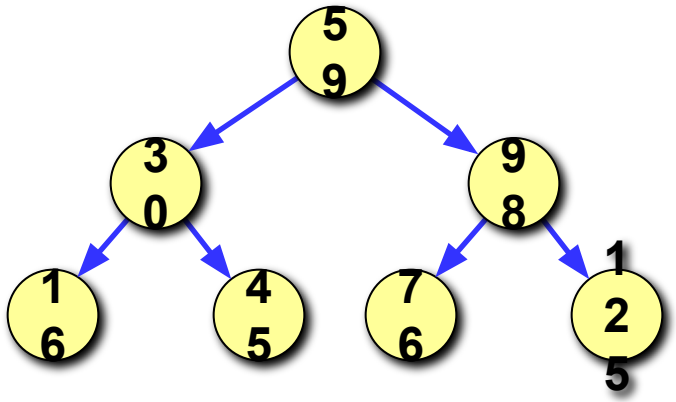
Поиск в массиве (N элементов):



При каждом сравнении отбрасывается 1 элемент.

Число сравнений – N.

Поиск по дереву (N элементов):



При каждом сравнении отбрасывается половина оставшихся элементов.

Число сравнений ~ $\log_2 N$.

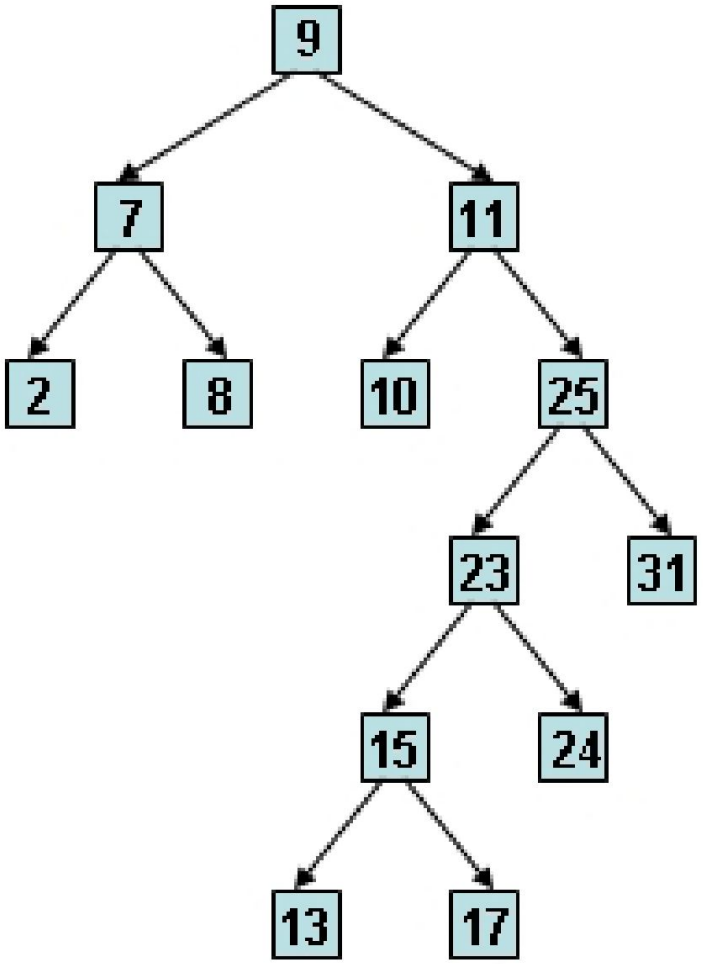
быстрый поиск

1) нужно заранее построить дерево;
2) желательно, чтобы дерево было минимальной высоты.

Несбалансированные двоичные деревья поиска (unbalanced)

Это такие деревья, высота правого и левого поддеревьев которых отличаются более, чем на 1.

Дерево двоичного поиска становится несбалансированным, когда в него постоянно добавляются элементы большего или меньшего размера

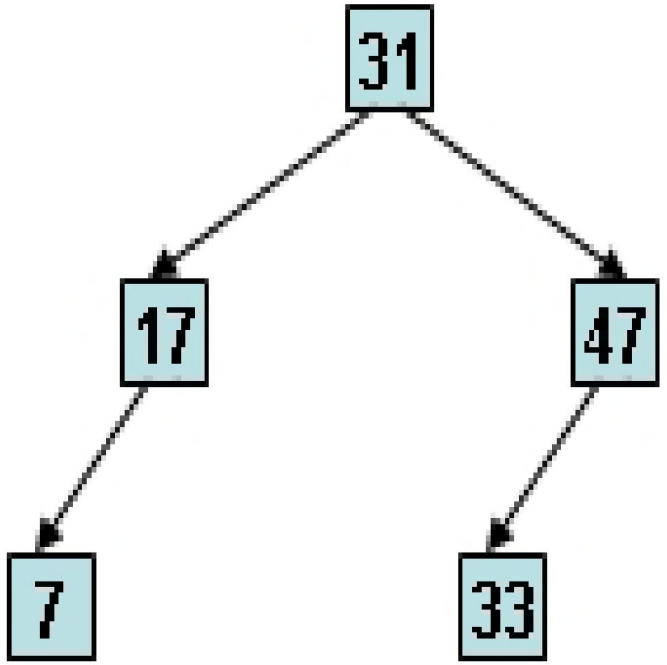


Неполные двоичные деревья поиска (incomplete)

Каждый узел дерева двоичного поиска должен содержать не более 2 детей.

Но он может иметь 1 ребенка или не иметь детей.

Если в дереве есть такие хотя бы один такой узел, дерево называют неполным.



Основные операции в двоичном дереве поиска

Базовый интерфейс двоичного дерева поиска состоит из трех операций:

- **Поиск** узла, в котором хранится пара (key, value) с key = K.
- **Добавление** в дерево пары (key, value) = (K, V).
- **Удаление** узла, в котором хранится пара (key, value) с key = K.

Поиск элемента

Дано: дерево T и ключ K.

Задача: проверить, есть ли узел с ключом K в дереве T, и если да, то вернуть ссылку на этот узел.

Алгоритм:

- Если дерево пусто, сообщить, что узел не найден, и остановиться.
- Иначе сравнить K со значением ключа корневого узла X.
  - Если K=X, выдать ссылку на этот узел и остановиться.
  - Если K>X, рекурсивно искать ключ K в правом поддереве T.
  - Если K<X, рекурсивно искать ключ K в левом поддереве T.

Добавление элемента

Дано: дерево T и пара (K,V).
Задача: добавить пару (K, V) в дерево T.
Алгоритм:

- Если дерево пусто, заменить его на дерево с одним корневым узлом ((K,V), null, null) и остановиться.
- Иначе сравнить K с ключом корневого узла X.
  - Если K>=X, рекурсивно добавить (K,V) в правое поддерево T.
  - Если K<X, рекурсивно добавить (K,V) в левое поддерево T.

Удаление узла

Дано: дерево T с корнем n и ключом K.
Задача: удалить из дерева T узел с ключом K (если такой есть).
Алгоритм:

- Если дерево T пусто, остановиться
- Иначе сравнить K с ключом X корневого узла n.
  - Если K>X, рекурсивно удалить K из правого поддерева T.
  - Если K<X, рекурсивно удалить K из левого поддерева T.
  - Если K=X, то необходимо рассмотреть три случая.
    - Если обоих детей нет, то удаляем текущий узел и обнуляем ссылку на него у родительского узла.
    - Если одного из детей нет, то значения полей второго ребёнка m ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом m.
    - Если оба ребёнка присутствуют, то
      - найдём узел m, являющийся самым левым узлом правого поддерева с корневым узлом Right(n);
      - присвоим ссылке Left(m) значение Left(n)
      - ссылку на узел n в узле Parent(n) заменить на Right(n);
      - освободим память, занимаемую узлом n (на него теперь никто не указывает).

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

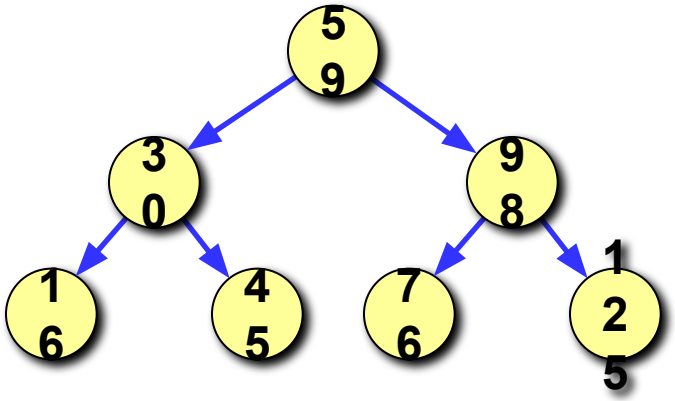
дерево пустое: создаем новый узел (корень)

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»):

16 30 45 59 76 98 125

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

125 98 76 59 45 30 16

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

59 30 16 45 98 76 125

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

16 45 30 76 125 98 59

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

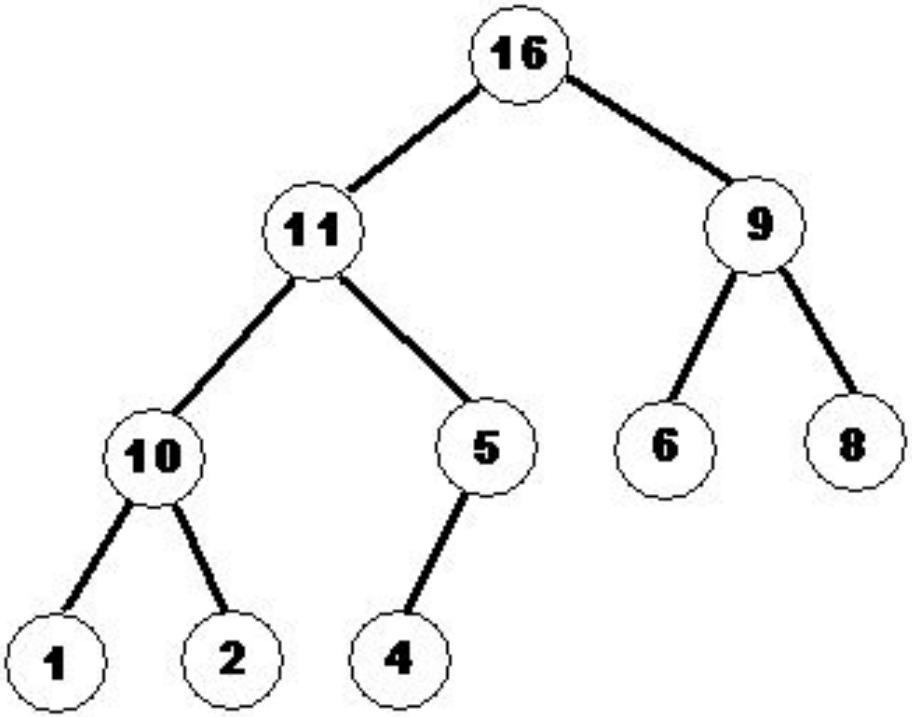
обход левого поддерева

вывод данных корня

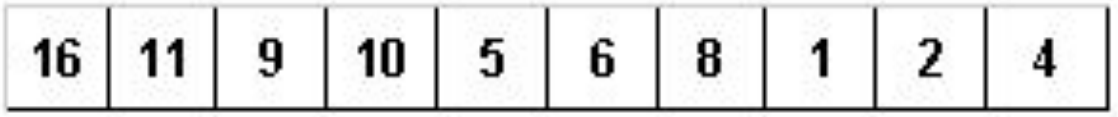
обход правого поддерева

Для рекурсивной структуры удобно применять рекурсивную обработку!

Двоичная куча



Двоичная куча



Структура данных для хранения двоичной кучи

Двоичная куча

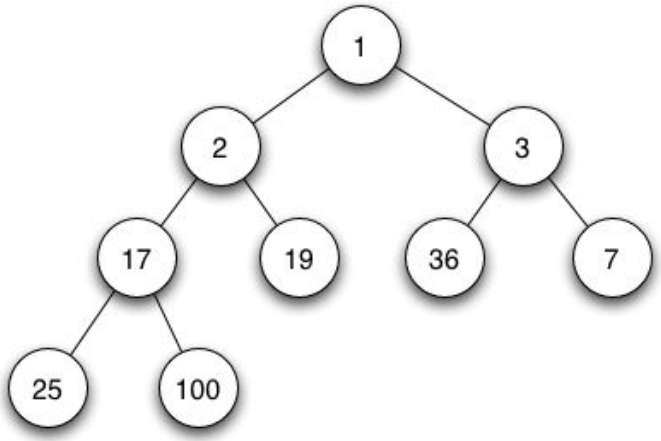
Двоичная куча *(пирамида)* — такое **двоичное дерево**, для которого выполнены три условия:

- Значение в любой вершине больше, чем значения её потомков.
- Каждый лист имеет глубину (расстояние до корня) либо *d* либо *d-1*. Иными словами, если назвать *слоем* совокупность листьев, находящемся на определённой глубине, то все слои, кроме, может быть, последнего, заполнены полностью.
- Последний слой заполняется слева направо.

Существуют также кучи, где значение в любой вершине, наоборот, меньше, чем значения её потомков. Такие кучи называются min-heap, а кучи, описанные выше — max-heap.

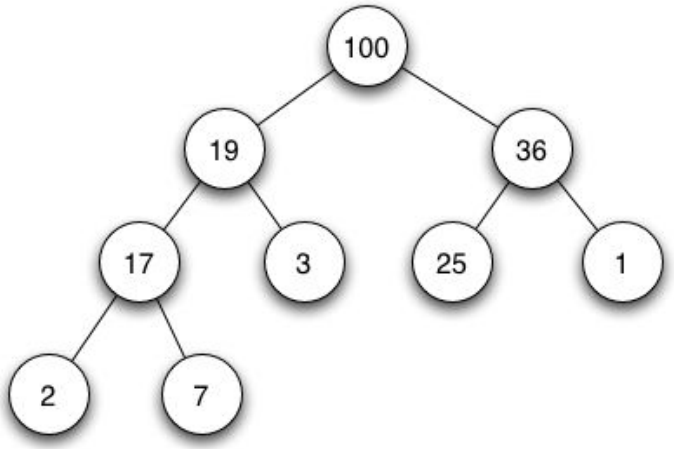
Кучи

- Min-heap



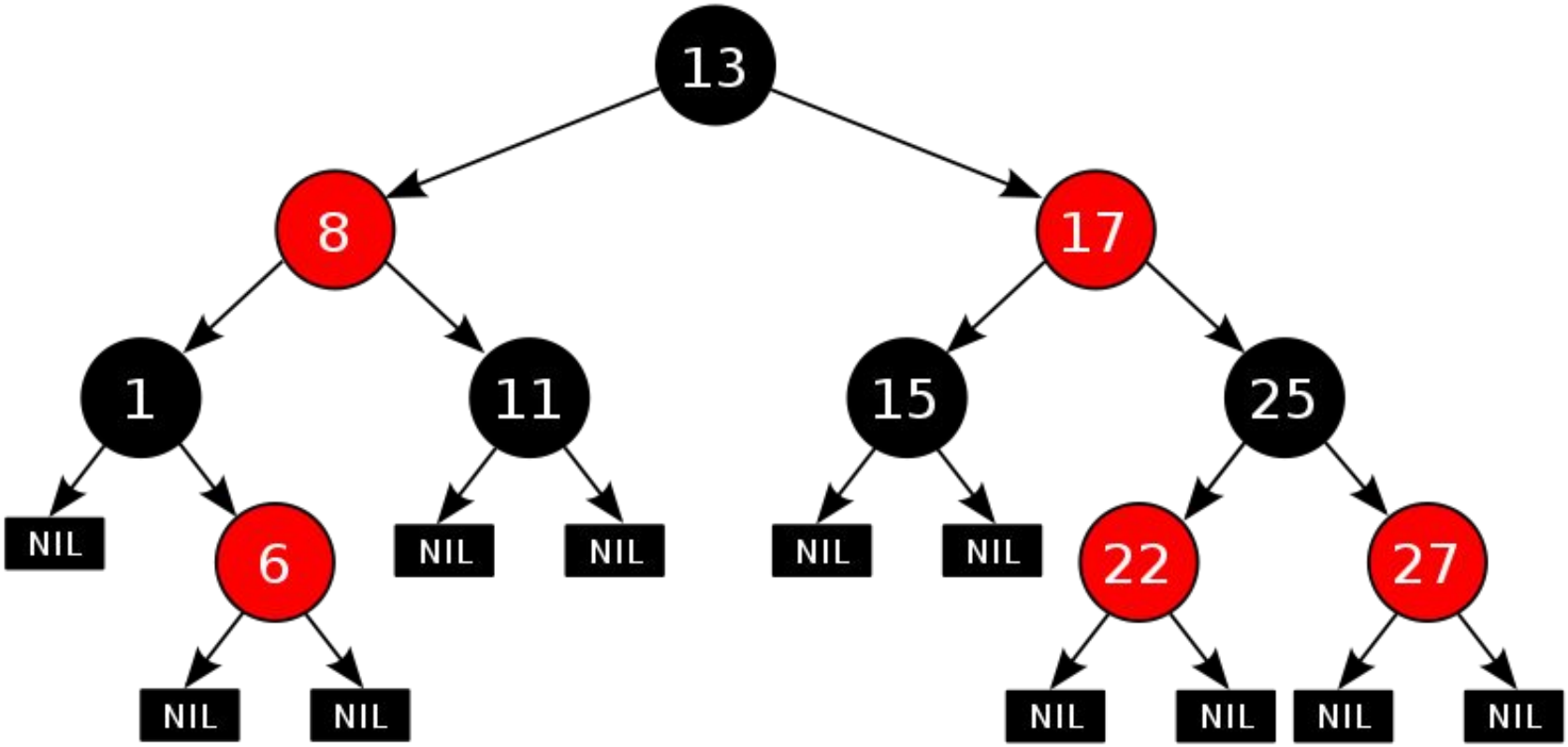
Значение в любой вершине меньше, чем значения ее потомков

- Max-heap



Значение в любой вершине больше, чем значения ее потомков

Красно-чёрное дерево



Красно-чёрное дерево

Красно-чёрное дерево (Red-Black-Tree, RB-Tree) — это одно из самобалансирующихся **двоичных деревьев поиска**, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла.

Сбалансированность достигается за счет введения дополнительного атрибута узла дерева — «цвет». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный».

Красно-чёрное дерево обладает следующими свойствами:

- Все листья черные.
- Все потомки красных узлов черные (т.е. запрещена ситуация с двумя красными узлами подряд).
- На всех ветвях дерева, ведущих от его корня к листьям, число чёрных узлов одинаково. Это число называется **чёрной высотой** дерева.

При этом для удобства листьями красно-чёрного дерева считаются фиктивные «нулевые» узлы, не содержащие данных.

АВЛ-дерево

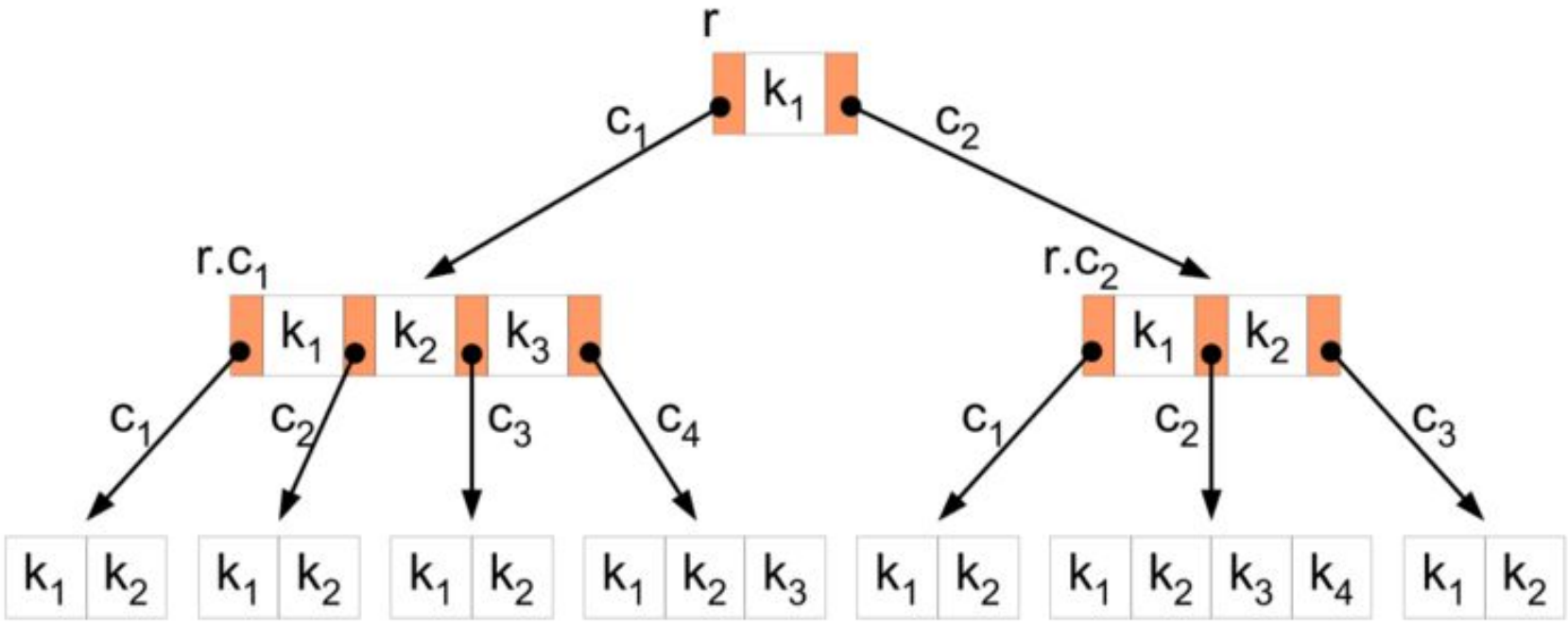
- **АВЛ-дерево** — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.
- АВЛ-деревья названы по первым буквам фамилий их изобретателей, Г. М. Адельсона-Вельского и Е. М. Ландиса, которые впервые предложили использовать АВЛ-деревья в 1962

B-дерево

B-дерево (по-русски произносится как **Б-дерево**) — структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево во внешней памяти.

- *Сбалансированность* означает, что длина пути от корня дерева к любому его листу одна и та же.
- *Ветвистость дерева* — это свойство каждого узла дерева ссылаться на большое число узлов-потомков.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, то есть каждому узлу дерева соответствует блок внешней памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.



Пример B-дерева степени 2

2-3-дерево

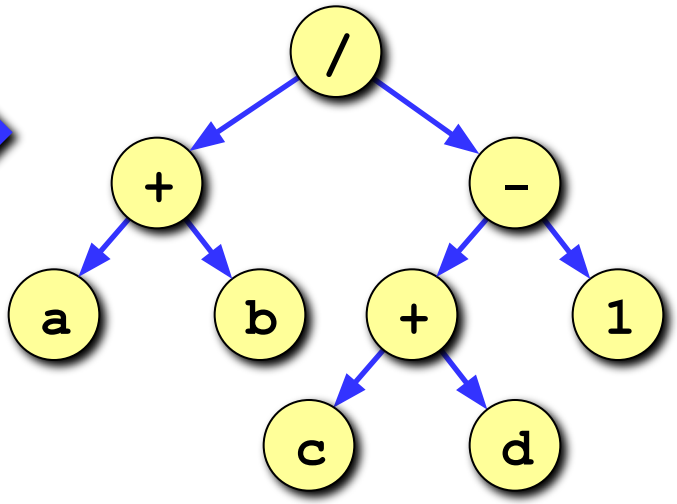
2-3 дерево — структура данных являющаяся B-деревом степени 1, страницы которого могут содержать только 2-вершины (вершины с одним полем и 2-мя детьми) и 3-вершины (вершины с 2-мя полями и 3-мя детьми).

Листовые вершины являются исключением — у них нет детей (но может быть одно или два поля). 2-3 деревья сбалансированы, то есть каждое левое, правое, и центральное поддерево одинаковой высоты, и таким образом содержат равное (или почти равное) число данных.

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)

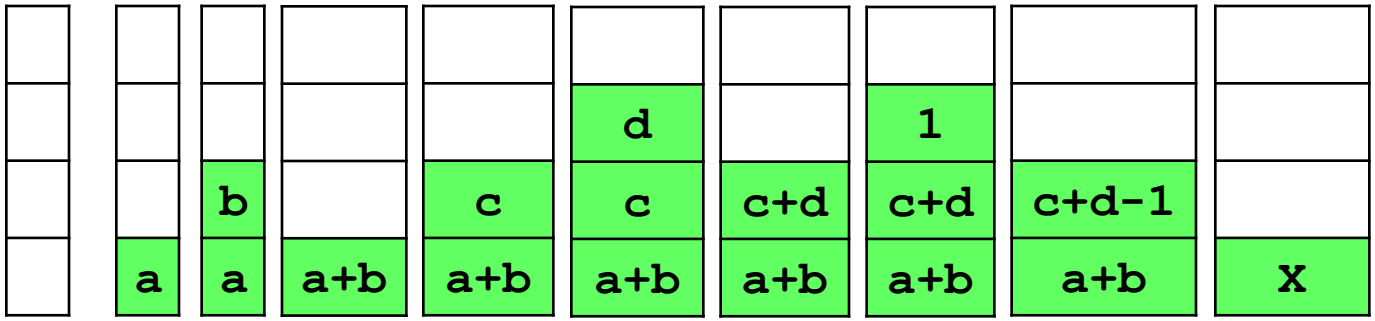
```
a b + c d + 1 - /
```

обратная польская нотация,
F. L. BauerF. L. Bauer and E. W. Dijkstra

Вычисление выражений

Постфиксная форма:

X =	a b + c d + 1 - /



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 `+-*\`. Вычислить это выражение.

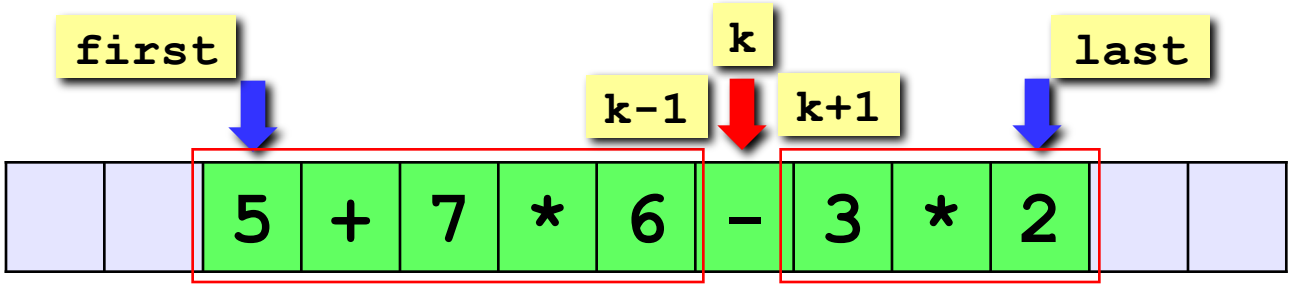
Алгоритм:

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.

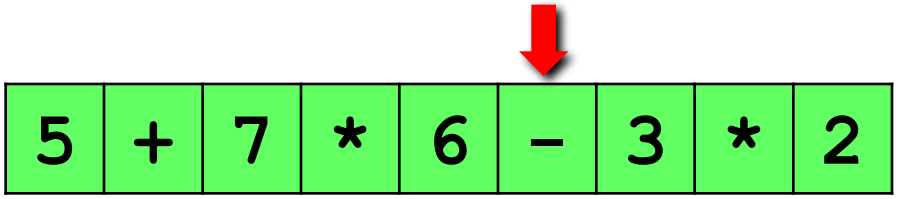
Построение дерева



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 (поддеревья)

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.

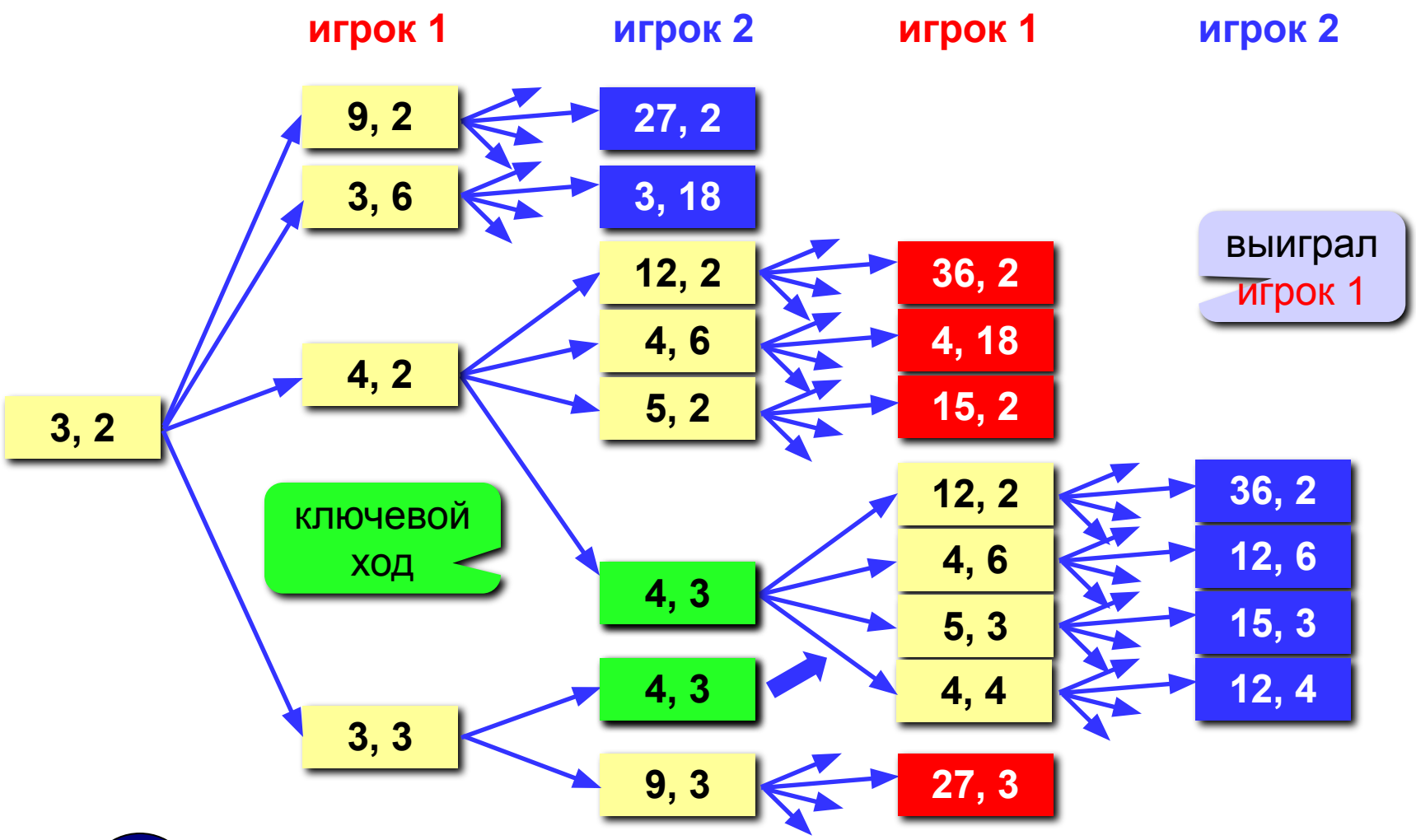
Перед двумя игроками лежат две кучки камней, в первой из которых 3, а во второй – 2 камня. У каждого игрока неограниченно много камней.

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!