Алгоритмы
и
структуры данных

ДЕРЕВЬЯ.

ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ.

БИНАРНЫЕ ДЕРЕВЬЯ ПОИСКА

Деревья. Основные понятия и определения

1. Определения
2. Способы изображения структуры дерева
3. Основные понятия и определения
4. Обходы дерева

Бинарные деревья поиска

1. Основные понятия и определения
2. Работа с бинарным деревом поиска
3. Обход бинарного дерева поиска
4. Идеально сбалансированное бинарное дерево поиска
5. Реализация бинарного дерева поиска на С++

Деревья. Основные понятия и определения

1. Определения
2. Способы изображения структуры дерева
3. Основные понятия и определения
4. Обходы дерева

Рекурсивное определение.

Дерево с базовым типом *T* — это либо:

1) пустое дерево,
2) либо некоторый узел типа *T* с некоторым конечным числом связанных с ним деревьев типа *T*, называемых поддеревьями.

Дерево второй степени – двоичное (бинарное) дерево –

конечное множество элементов(вершин), которое

1) либо пусто
2) либо состоит из корня(вершины) с двумя отдельными бинарными деревьями, которые называются левым и правым поддеревом этого корня.

Бинарное дерево, не содержащее вершин, является пустым.

Элемент дерева – узел - вершина

Двоичное (бинарное) дерево может быть представлено при помощи связной структуры данных, в которой каждый узел является объектом, содержащим поля:

- Ключ (key) и сопутствующие поля данных
- Поля left, right, p – левый, правый, родительский узел соответственно.

Если дочерний или родительский узел отсутствует, то поле – NIL.

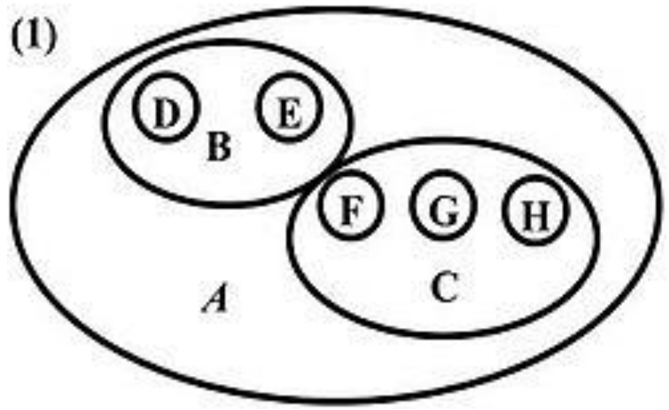
Указатель p для корневого узла дерева – NIL.

2. Способы изображения структуры дерева

1) вложенные множества
2) вложенные скобки
3) отступы
4) граф

Дерево, элементами которого являются буквы латинского алфавита

Вложенные множества:

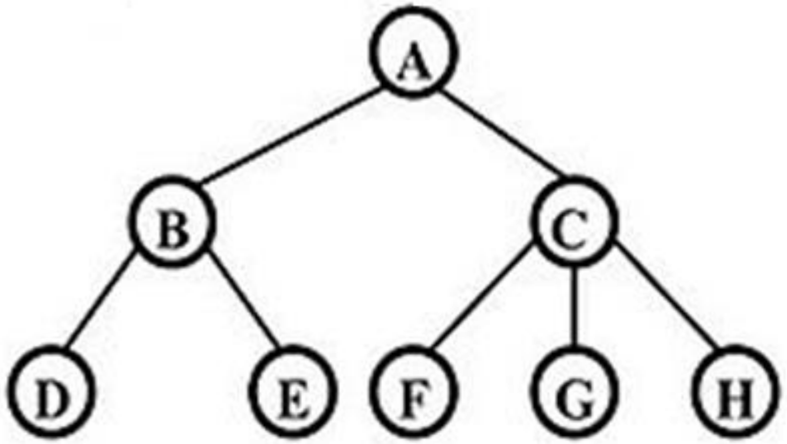


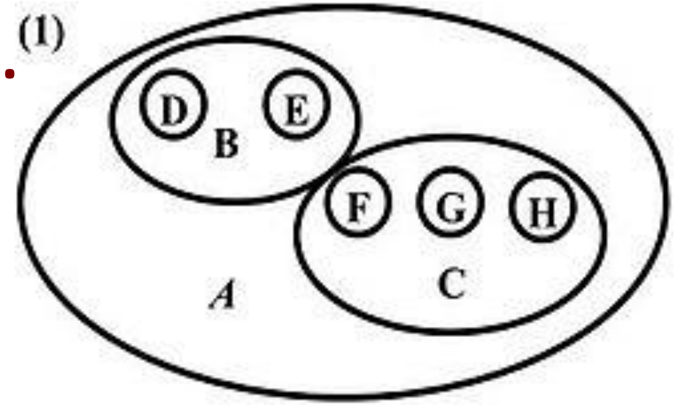
Вложенные скобки: (A (B (D, E)) (C (F, G, H)))

Отступы

Дерево, элементами которого являются буквы латинского алфавита

Граф:



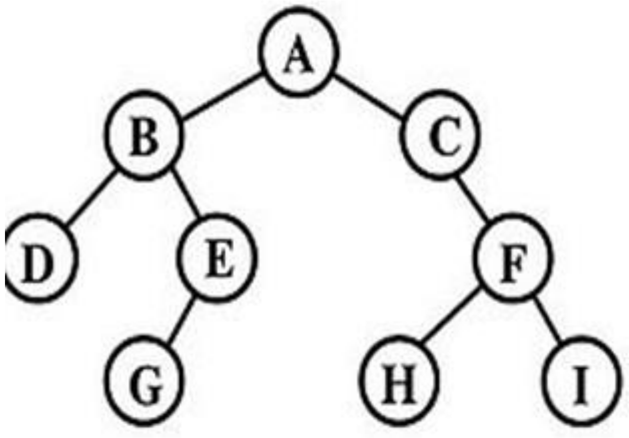


3. Основные понятия и определения

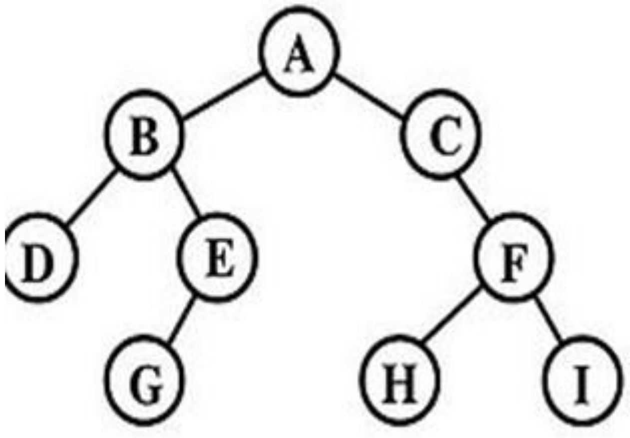
Корневой узел — самый верхний узел дерева.

Лист, листовой или терминальный узел — узел, не имеющий дочерних элементов.

Внутренний узел — любой узел дерева, имеющий потомков, и таким образом, не являющийся листовым узлом.



Каждый узел дерева имеет ноль или более узлов-потомков, которые располагаются ниже по дереву.



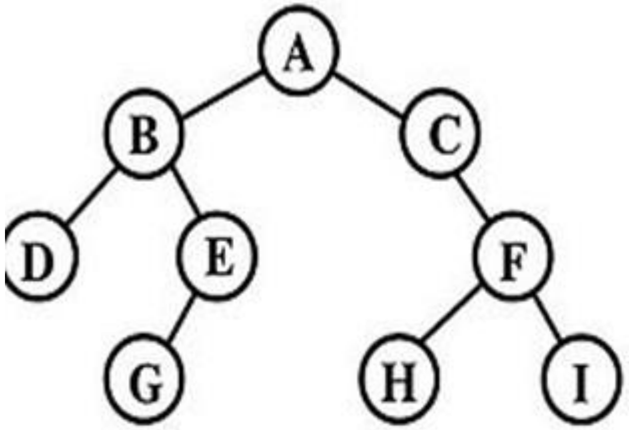
Узел, имеющий потомка, называется узлом-родителем относительно своего потомка (или *узлом-предшественником*, или старшим).

Каждый узел имеет не больше одного узла-родителя (предка).

Высота узла — это максимальная длина нисходящего пути от этого узла к самому нижнему узлу (краевому узлу – *листу)*.

Высота корневого узла равна высоте всего дерева.

Глубина вложенности узла равна длине пути до корневого узла.

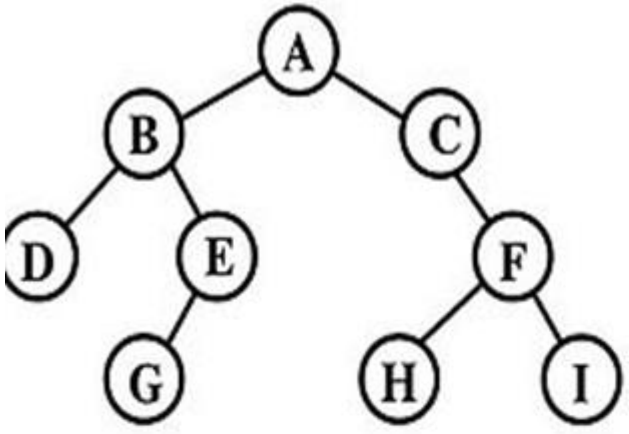


Дерево принято изображать растущим вниз,

его верхний узел - корень.

Все узлы дерева разбивают на уровни.

Корень имеет нулевой уровень.



Если узел x лежит на уровне n, а y связан с x и лежит на уровне n + 1, то говорят, что y — потомок (сын) x, а x — предок (отец) y.

Если элемент не имеет потомков, то его называют терминальным узлом, или листом,

в противном случае узел называют внутренним.

Число непосредственных потомков внутреннего узла называют степенью узла.

Максимальная степень всех узлов есть степень дерева.

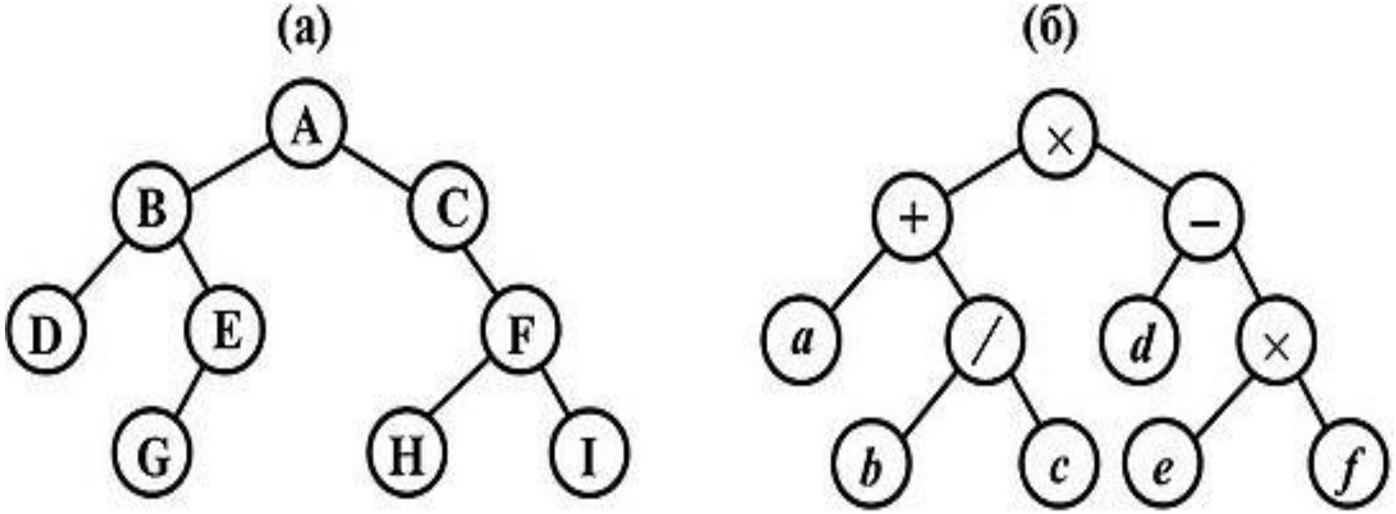
Упорядоченное дерево — это дерево, у которого ветви, исходящие из каждого узла, упорядочены. Поэтому два упорядоченных дерева — это разные объекты

Упорядоченные деревья второй степени = двоичные (бинарные) деревья**.**

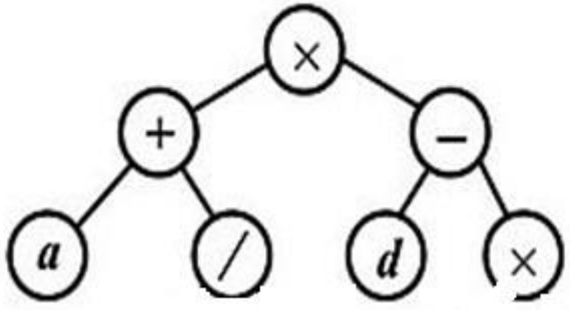
Если каждый узел бинарного дерева, не являющийся листом, имеет непустые правые и левые поддеревья, то дерево называется строго бинарным.

Каждый узел имеет степень 0 или 2.

Пример строго бинарного дерева (б):



Строго бинарное дерево, все листья которого расположены на одном уровне, называется совершенным.



Совершенное дерево высотой h содержит

$2^h - 1$ узлов.

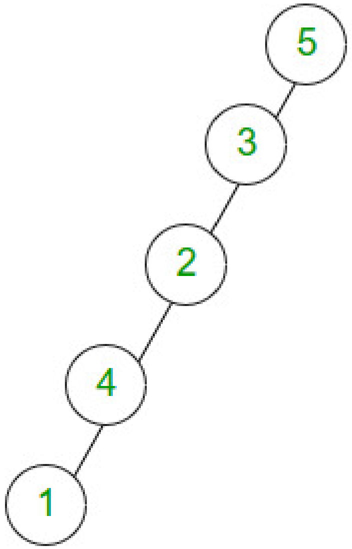
На каждом его уровне расположено максимальное количество узлов, равное 2^h,

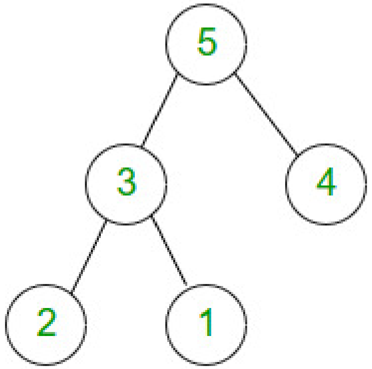
где *h* — номер уровня.

Если в бинарном дереве **n** узлов, максимальная высота двоичного дерева равна n – 1**,**

минимальная высота — floor ($log_2 n$)

Например, левое «перекошенное» двоичное дерево с 5 узлами, имеет высоту 5-1 = 4.



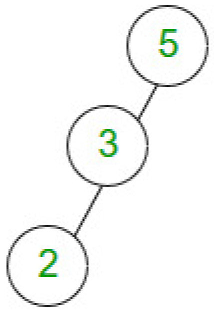


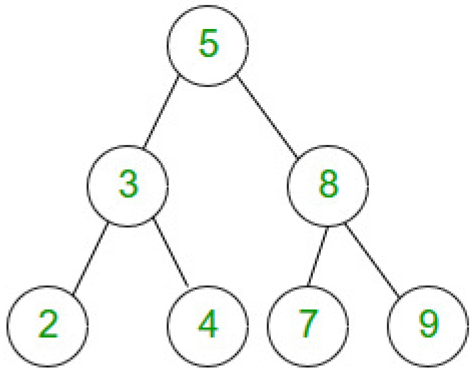
Если бинарное дерево имеет высоту h , минимальное количество узлов равно n + 1 (в случае левостороннего и правостороннего двоичного дерева).

Если бинарное дерево имеет высоту h, максимальное количество узлов будет, когда все уровни полностью заполнены.

Общее количество узлов будет $2^0 + 2^1 + \ldots 2^h = 2^{h+1} - 1$

Например, двоичное дерево с высотой 2, имеет $2^{2+1} - 1 = 7$ узлов.

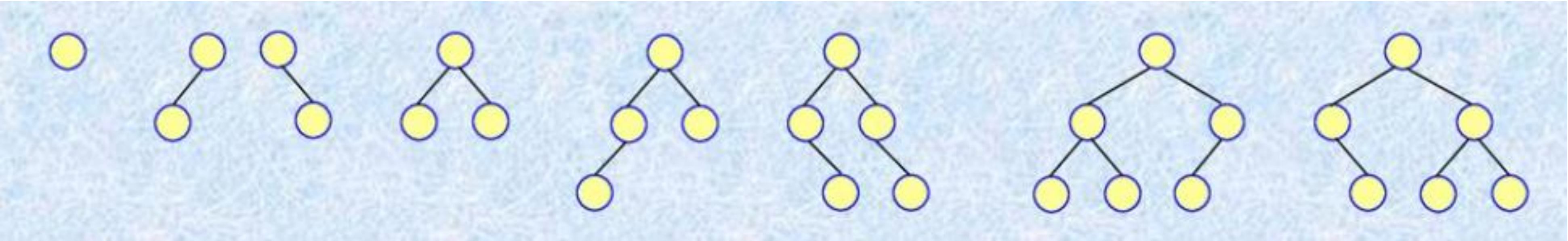


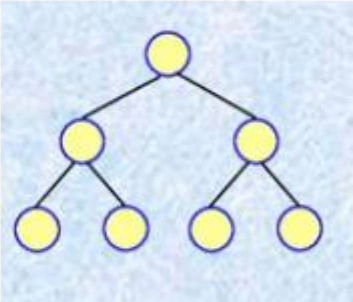


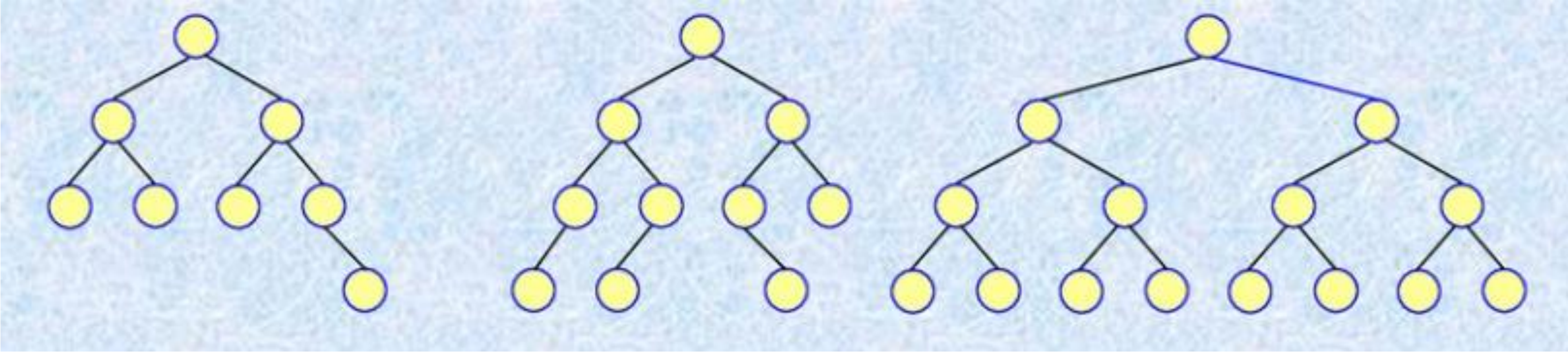
Бинарное дерево называется сбалансированным, если для любой его вершины ***v* высоты** левого и правого поддерева, выходящих из v (т.е. поддеревьев с корнями left [v]и right [v]), отличаются не более чем **на 1.**

Идеально сбалансированными деревьями будем называть деревья, для которых для каждой вершины **количество** элементов в левом и правом поддереве отличается не более, чем на 1.

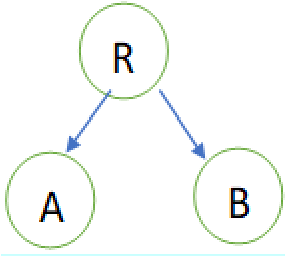
Идеально сбалансированными деревьями будем называть деревья, для которых для каждой вершины количество элементов в левом и правом поддереве отличается не более, чем на 1.







4. Обходы дерева



Пусть имеем дерево, где

R — корень,

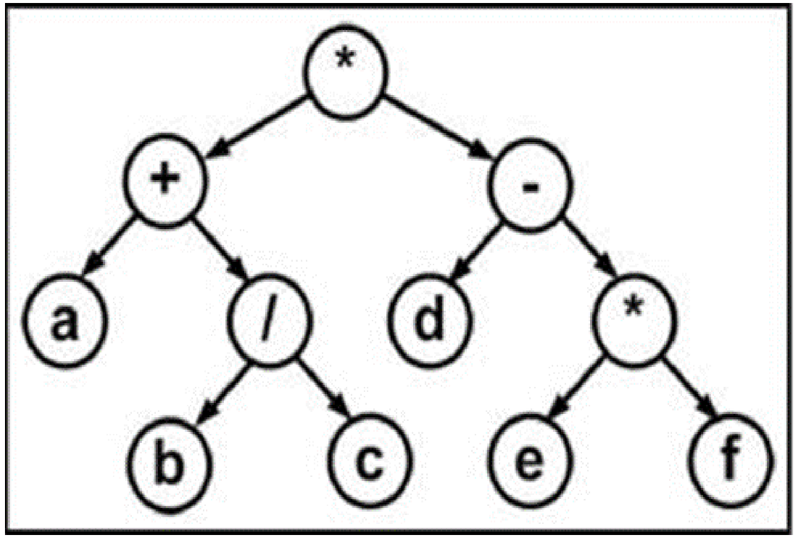
A и B — левое и правое поддеревья.

Рассмотрим три способа обхода дерева:

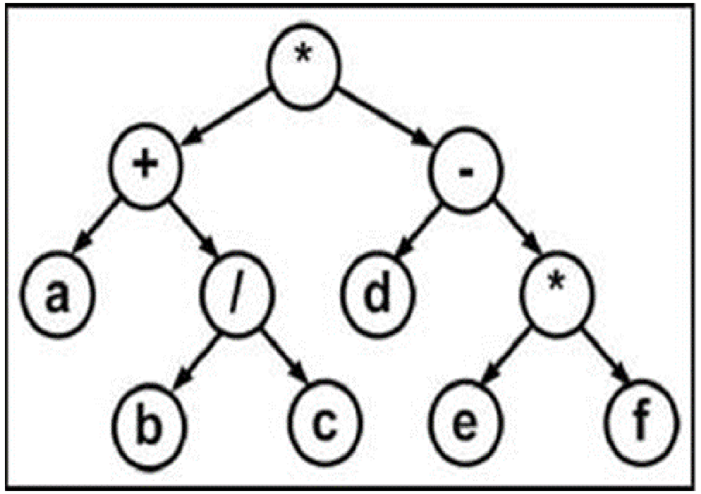
1. Обход дерева в прямом порядке (сверху вниз): R, A, B.
2. В симметричном порядке (слева направо): A, R, B.
3. В обратном порядке (снизу вверх): A, B, R.

Функции обхода дерева удобно выразить рекурсивно.

Дерево для арифметического выражения: (a + b/c)*(d-e*f)



Дерево для арифметического выражения: (a + b/c)*(d-e*f)



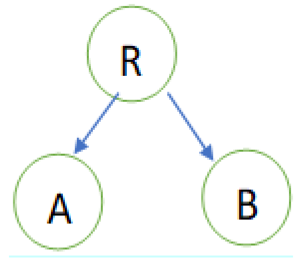
Обходя дерево и выписывая символы в узлах, получаем:

1. Сверху вниз: * + a / b c – d * e f — префиксная запись выражения.
2. Слева направо: a + b / c * d – e * f — инфиксная запись (без скобок).
3. Снизу вверх: a b c / + d e j * - * — постфиксная запись.

Обход дерева симметричном порядке (слева направо, инфиксный): A, R, B.

INORDER_TREE_WALK(x)

1. **If x ≠ NIL then**
2. INORDER_TREE_WALK (**left [x]**)
3. print **key[x]**
4. INORDER_TREE_WALK (**right [x]**)
5. **end if**



Если x – корень поддерева, в котором имеется n узлов, то процедура INORDER_TREE_WALK(x) выполняется за время Θ(n)

Бинарные деревья **поиска**

1. Основные понятия и определения

2. Работа с бинарным деревом поиска

3 . Обходы бинарного дерева поиска

3. Идеально сбалансированное бинарное дерево поиска.

4. Реализация бинарного дерева поиска на C++

Двоичное (бинарное) дерево может быть представлено при помощи связной структуры данных, в которой каждый узел является объектом, содержащим поля:

- Ключ (key) и сопутствующие поля данных
- Поля left, right, p – левый, правый, родительский узел соответственно.

Если дочерний или родительский узел отсутствует, то поле – NIL.

Указатель p для корневого узла дерева – NIL.

1. Основные понятия и определения

Ключи в *двоичном дереве **поиска*** хранятся так, чтобы в любой момент удовлетворять свойству бинарного дерева поиска:

Если x - узел бинарного дерева поиска,

а узел y находится в левом поддереве x, то key[y] < key[x].

Если узел y находится в правом поддереве x, то key[x] ≤ key[y].

key[x] < key[y]

2. Работа с бинарным деревом поиска

1. Поиск определенного ключа
2. Поиск минимального и максимального ключа
3. Поиск предшествующего и последующего ключа
4. Вставка нового узла
5. Удаление узла

Запросы:

1) Поиск определенного ключа
2) Поиск минимального и максимального ключа
3) Поиск предшествующего и последующего ключа

Модифицирующие операции:

1) Вставка нового узла
2) Удаление узла

Все эти операции в бинарном дереве поиска высотой h выполняются за время O(h)

Дерево поиска может использоваться как словарь и очередь с приоритетами…

2.1 Поиск определенного ключа

Для поиска узла с заданным ключом используется TREE_SEARCH ,

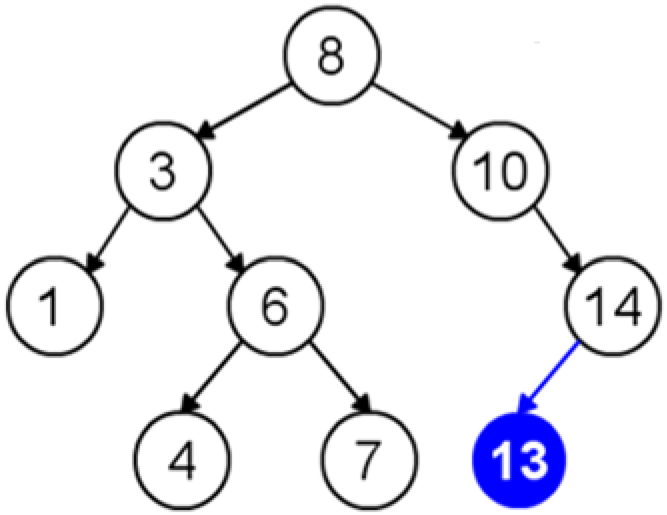
параметры - указатель на корень бинарного дерева и ключ,

Результат - указатель на узел с этим ключом (если такой существует; в противном случае – NIL).

2.1 Поиск определенного ключа

TREE_SEARCH(x, k)

```
if  x = NIL or  k = key[x] then
    return x
end if
if k < key[x] then
    return TREE_SEARCH( left[x], k)
else
    return TREE_SEARCH( right[x], k)
end if
```



Узлы, которые посещаются при рекурсивном поиске, образуют нисходящий путь от корня дерева, следовательно, время работы TREE_SEARCH равно O(h),где h - высота дерева

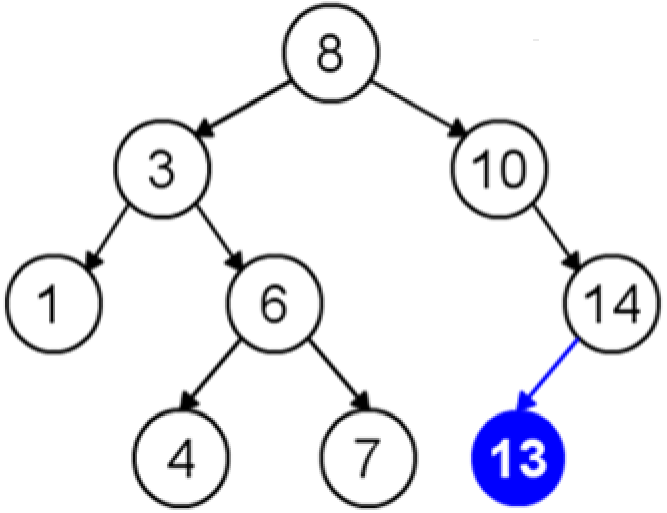
2.1 Поиск определенного ключа

Процедуру TREE_SEARCH можно записать **итеративно,** «разворачивая» рекурсию в цикл while.

Часто такая версия оказывается более эффективной.

TREE_SEARCH_ ITERATIVE_(x, k)

```
while x ≠ NIL and k ≠ key[x] do
    if k < key[x] then
        x ← left[x]
    else
        x ← right [x]
    end if
end while
return x
```



Эффективность ITERATIVE_TREE_SEARCH (x, k) по времени и памяти по сравнению с рекурсивной реализацией TREE_SEARCH(x, k) достигается за счет того, что не нужно на каждом шаге заново вызывать функцию и хранить информацию, связанную с текущим ее вызовом

Высота дерева – это максимальная длина пути от корня до листа.

При поиске узла на каждой итерации мы спускаемся на один уровень вниз, поэтому время поиска узла в двоичном дереве поиска, то есть, количество шагов, ограничено сверху высотой этого дерева. Используя O-символику, можно обозначить время поиска в худшем случае как **O(h)**,

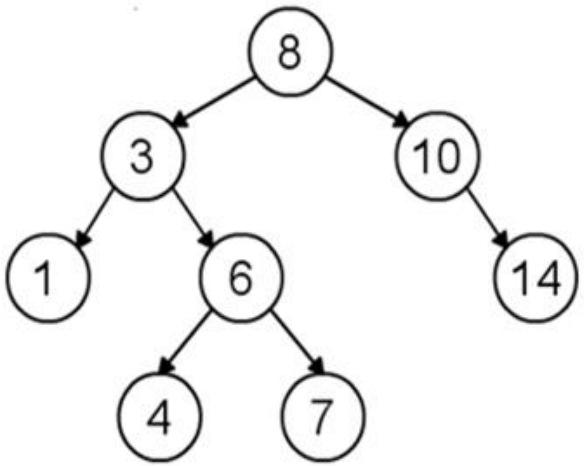
где h – высота дерева.

2.2 Поиск минимального и максимального ключа

Элемент с минимальным значением ключа можно найти, следуя по указателям left до тех пор пока не встретится NIL. Считаем, что $x \neq NIL$

TREE_MINIMUM (X)

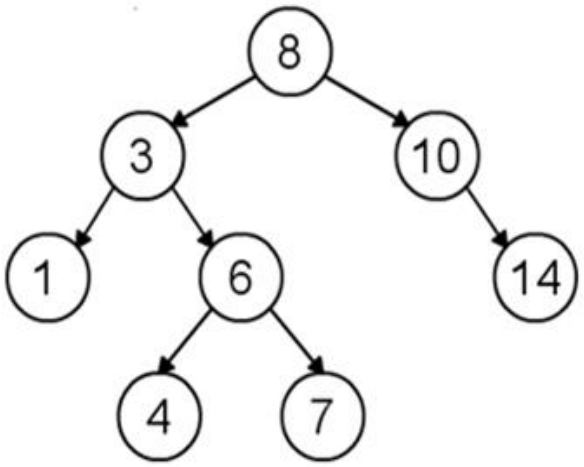
1. **while** left[x] ≠ NIL **do**
2. x ← left [x]
3. **end while**
4. **return** x



Элемент с максимальным значением ключа можно найти, следуя по указателям right до тех пор пока не встретится NIL. Считаем, что $x \neq NIL$

TREE_MAXIMUM (X)

1. **while** right [x] ≠ NIL **do**
2. x ← right [x]
3. **end while**
4. **return** x



Обе процедуры находят минимальный (максимальный) элемент дерева за время ***O(h)***,где h - высота дерева, т.к. последовательность проверяемых узлов образует нисходящий путь от корня дерева.

2.3 Поиск предшествующего и последующего ключа

Пусть известен узел в бинарном дереве поиска.

Определить какой узел следует за ним в отсортированной последовательности при центрированном (прямом, инфиксном) обходе дерева.

Определить какой узел предшествует ему в отсортированной последовательности при центрированном (прямом, инфиксном) обходе дерева

Если все ключи в дереве различны, последующим по отношению к узлу x является узел с наименьшим ключом, большим key[x].

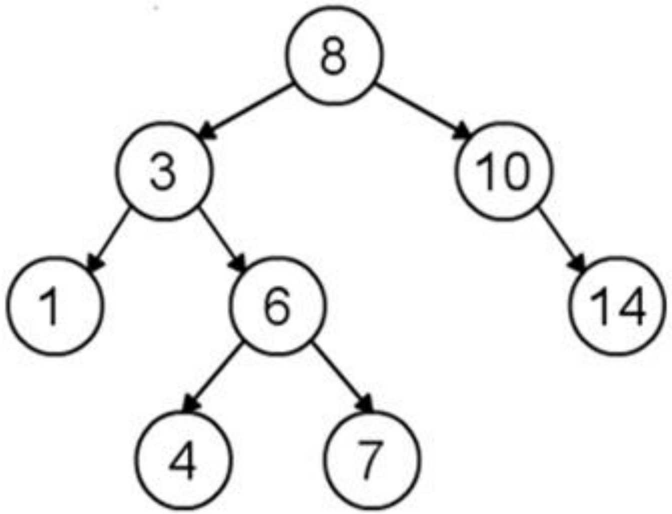
Найдем этот узел используя структуру бинарного дерева поиска, не выполняя сравнение ключей.

2.3 Поиск предшествующего и последующего ключа

Процедура TREE_SUCCESSOR возвращает узел, следующий за узлом x, если такой узел существует, и NIL, если key[x] - наибольший ключ. Считаем, что x ≠ NIL

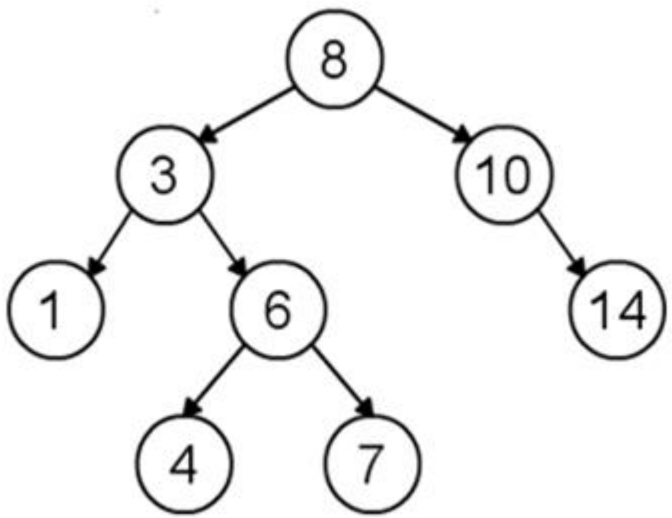
TREE_SUCCESSOR (x)

1. **if** right[x] ≠ NIL **then**
2. **return** TREE_MINIMUM (right [x])
3. **end if**



Процедура TREE_SUCCESSOR_ITERATIVE

возвращает узел, следующий за узлом x, если такой узел существует, и NIL, если key[x] - наибольший ключ.



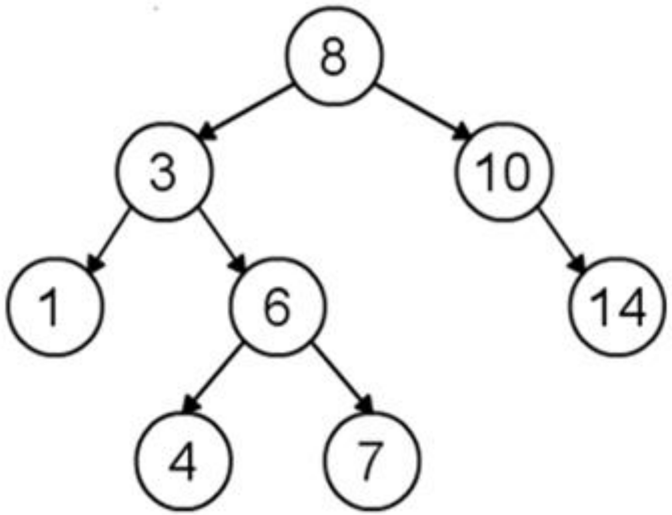
Если правое поддерево x – пустое: Поднимаемся вверх по дереву до тех пор, пока не встретим узел, который является левым дочерним узлом своего родителя.

2.3 Поиск предшествующего и последующего ключа

Считаем, что $x \neq NIL$

TREE_SUCCESSOR_ ITERATIVE (x)

```
if right[x] ≠ NIL then
    return TREE_MINIMUM ( right [ x ] )
end if
y ← p[x]
while y ≠ NIL and x = right [ y ] do
    x ← y
    y ← p[ y ]
end while
return y
```



Процедура TREE_SUCCESSOR_ ITERATIVE находит следующий узел за время O(h),

где h - высота дерева,

т.к. мы движемся либо по пути вниз от исходного узла, либо по пути вверх.

Процедура TREE_PREDECESSOR симметрична TREE_SUCCESSOR.

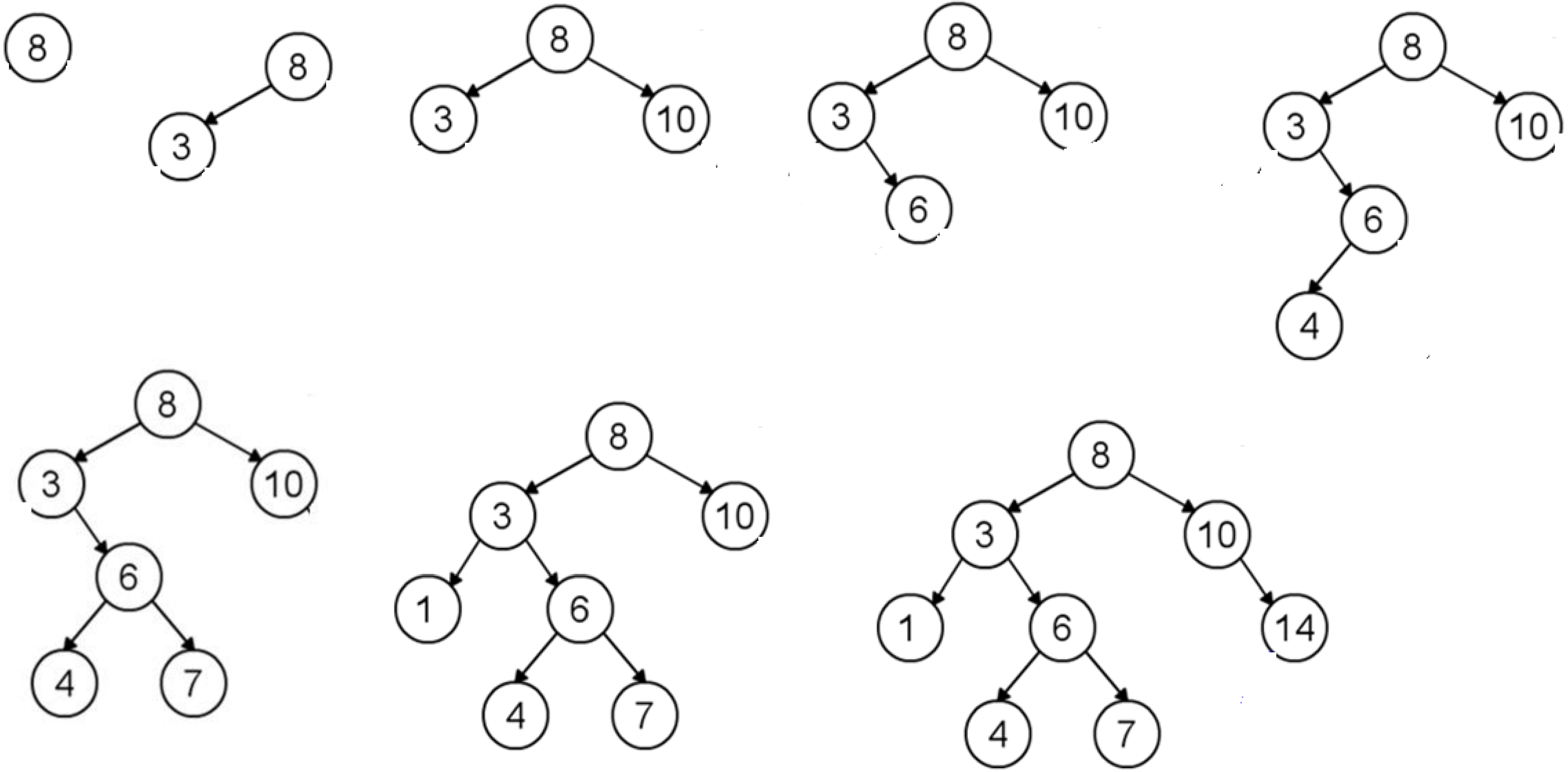
Если в дереве есть узлы с одинаковыми ключами, то мы можем определить последующий и предшествующий узлы как те, что возвращаются процедурами TREE_SUCCESSOR и TREE_PREDECESSOR.

2.4 Вставка нового узла

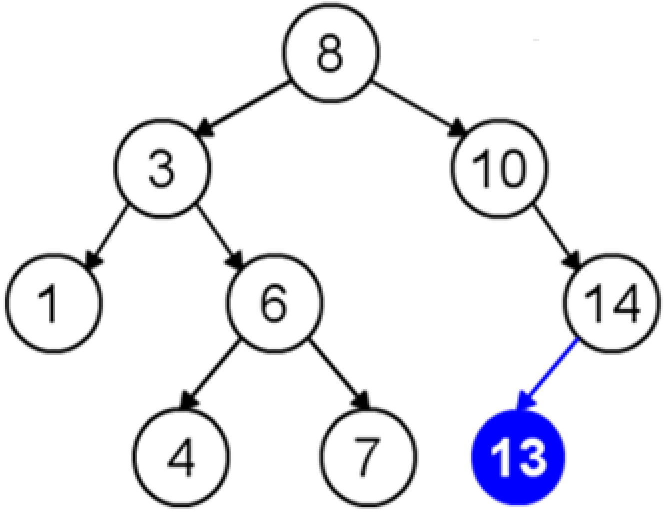
8, 3, 10, 6, 4, 7, 1, 14, 13

8, 3, 10, 6, 4, 7, 1, 14, 13

2.4 Вставка нового узла



8, 3, 10, 6, 4, 7, 1, 14, 13



При добавлении нового узла возможны следующие ситуации:

1) Дерево пустое. В этом случае новый узел становится корнем.

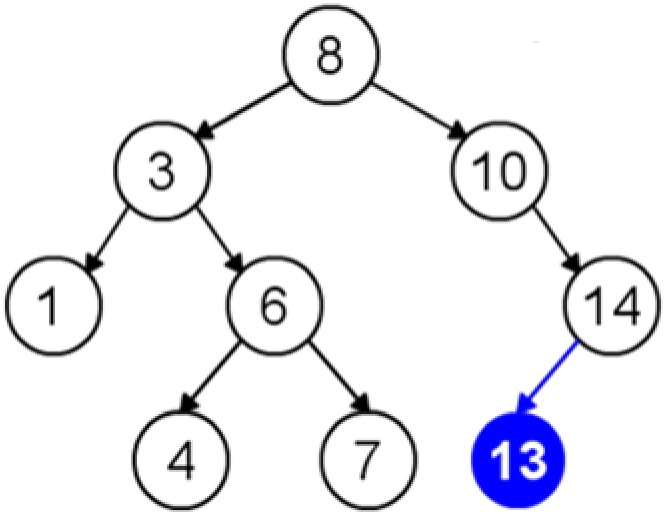
2) Новое значение меньше корневого. В этом случае значение должно быть вставлено слева. Если слева уже стоит элемент, то повторяем эту же операцию, только в качестве корневого узла рассматриваем левый узел. Если слева нет элемента, то добавляем новый узел.

3) Новое значение больше корневого. В этом случае новое значение должно быть вставлено справа. Если справа уже стоит элемент, то повторяем операцию, только в качестве корневого рассматриваем правый узел. Если справа узла нет, то вставляем новый узел.

Параметр – узел z, у которого key[z]=v, left[z]=NIL, right[z] = NIL

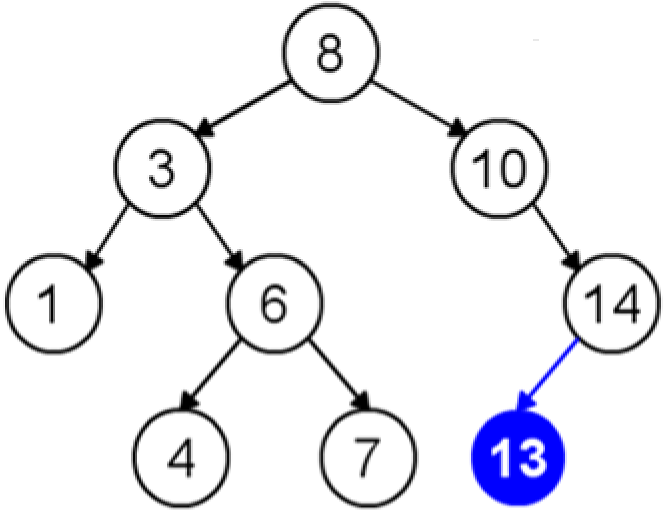
TREE_INSERT (T, z)

```
x ← root[ T ]
while   x ≠  NIL  do
      if key [ z ] < key [ x ]  then
          x ← left[x]
      else
            x   ← right [x]
       end if
end while
. . . ▶ нужен родитель x = NIL  и нужен родитель x!
```



Параметр – узел z, у которого key[z]=v, left[z]=NIL, right[z] = NIL

```
TREE_INSERT ( T, z )
    y ← NIL
    x ← root[ T ]
    while   x ≠ NIL do
      y ← x
        if key [ z ] < key [ x ] then
            x ← left[x]
         else
               x ← right [x]
        end if
    end while
    p[ z ]  ← y
    if y = NIL    then ▶  T - пустое
        root[ T ] ← z
     else if  key [ z ] < key [ y ] then
                left [ y ] ← z
            else
                right [ y ] ← z
            end if
    end if
```



Результат вставки узлов в дерево:

структура дерева будет зависит от порядка вставки элементов

(форма дерева зависит от порядка вставки элементов).

Если элементы не упорядочены и их значения распределены равномерно,

то дерево будет *достаточно сбалансированным*, путь от вершины до листьев будет *почти* одинаковый.

В таком случае максимальное время доступа до листа равно log(n),

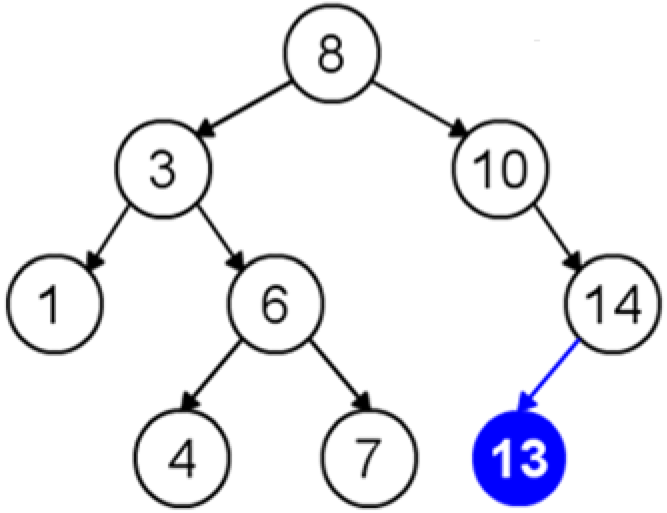
где n – это число узлов, то есть равно высоте дерева.

Если же элементы упорядочены, то дерево не будет сбалансировано и может растянуться в одну сторону, как список; тогда время доступа до последнего узла будет порядка n. Из-за этого применение этой структуры ограничено.

2.5 Удаление узла

Три возможных ситуации:

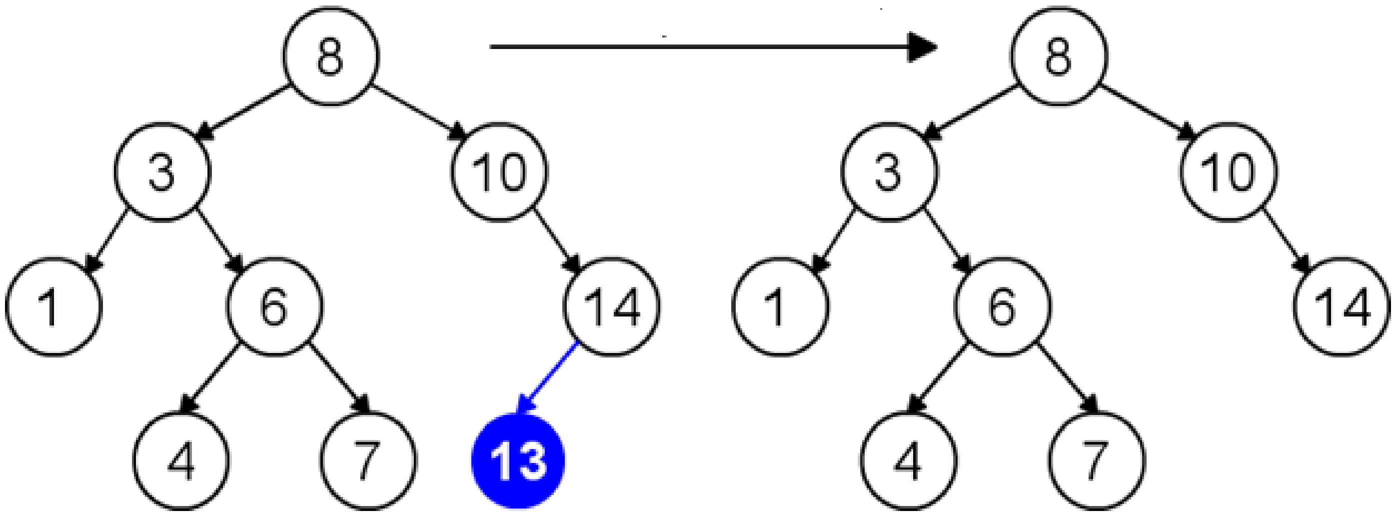
1) У узла **нет наследников** (удаляем лист). Тогда он просто удаляется, а его родитель обнуляет указатель на него.
2) У узла один **наследник**. В этом случае узел подменяется своим наследником.
3) У узла **оба наследника**. В этом случае узел не удаляем, а заменяем его значение на максимум левого поддерева (минимум правого поддерева). После этого удаляем максимум левого поддерева (минимум правого поддерева).



1) У узла нет наследников (удаляем лист).

Тогда он просто удаляется, а ссылка на него у родителя становится - NIL.

Если только корень = лист ?



Удаляемый узел z

left [p[z]] ← NIL или right[p[z]] ← NIL

if left [p[z]] = z

then // удаляется левый ребенок . .

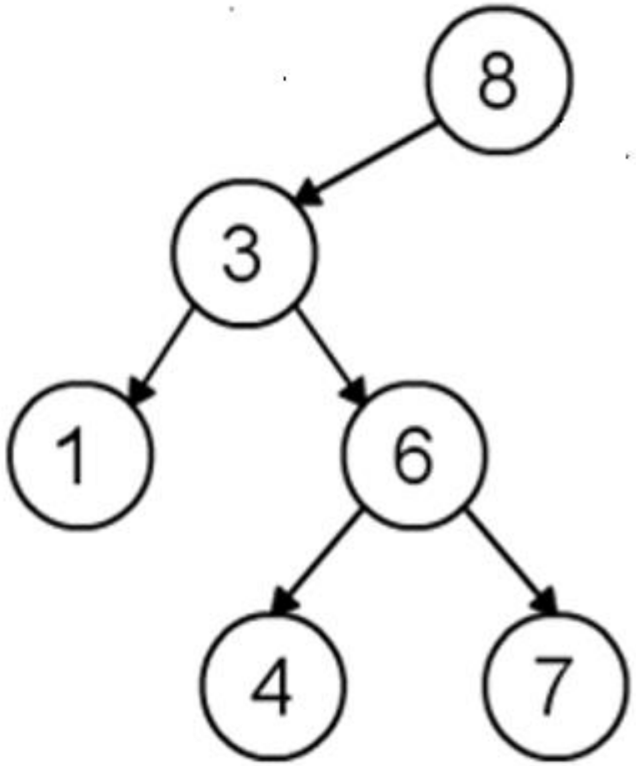
else // удаляется правый ребенок

2) У узла один наследник.

В этом случае узел подменяется своим наследником.

(left [z] ≠ NIL и right[z] = NIL) или

(left [z] = NIL и right[z] ≠ NIL)



Удаляемый узел z (8) - корень

if right[z] ≠ NIL **and** left [z] = NIL **and** p[z] = NIL **then**

root ← right[z]

p[right[z]] ← NIL

end if

if right[z] = NIL **and** left [z] ≠ NIL **and** p[z] = NIL **then**

root ← left [z]

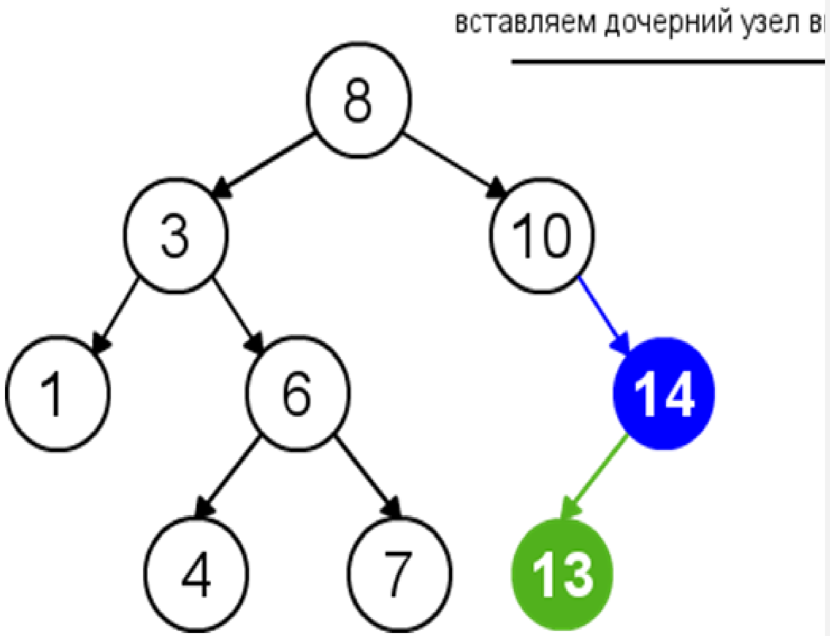
p[left[z]] ← NIL

end if

2) У узла один наследник.

В этом случае узел подменяется своим наследником.

(left [z] ≠ NIL и right[z] = NIL) или
(left [z] = NIL и right[z] ≠ NIL)



Удаляемый узел z (14) – **не корень**

if right[z] = NIL **and** left [z] ≠ NIL **and** p[z] ≠ NIL **then**

▶ есть левый ребенок

if right[p[z]] = z **then**

right[p[z]] ← left[z] ▶ удаляемый узел справа

else

left [p[z]] ← left[z] ▶ удаляемый узел слева

end if

else

if right[z] ≠ NIL **and** left[z] = NIL **and** p[z] ≠ NIL **then**

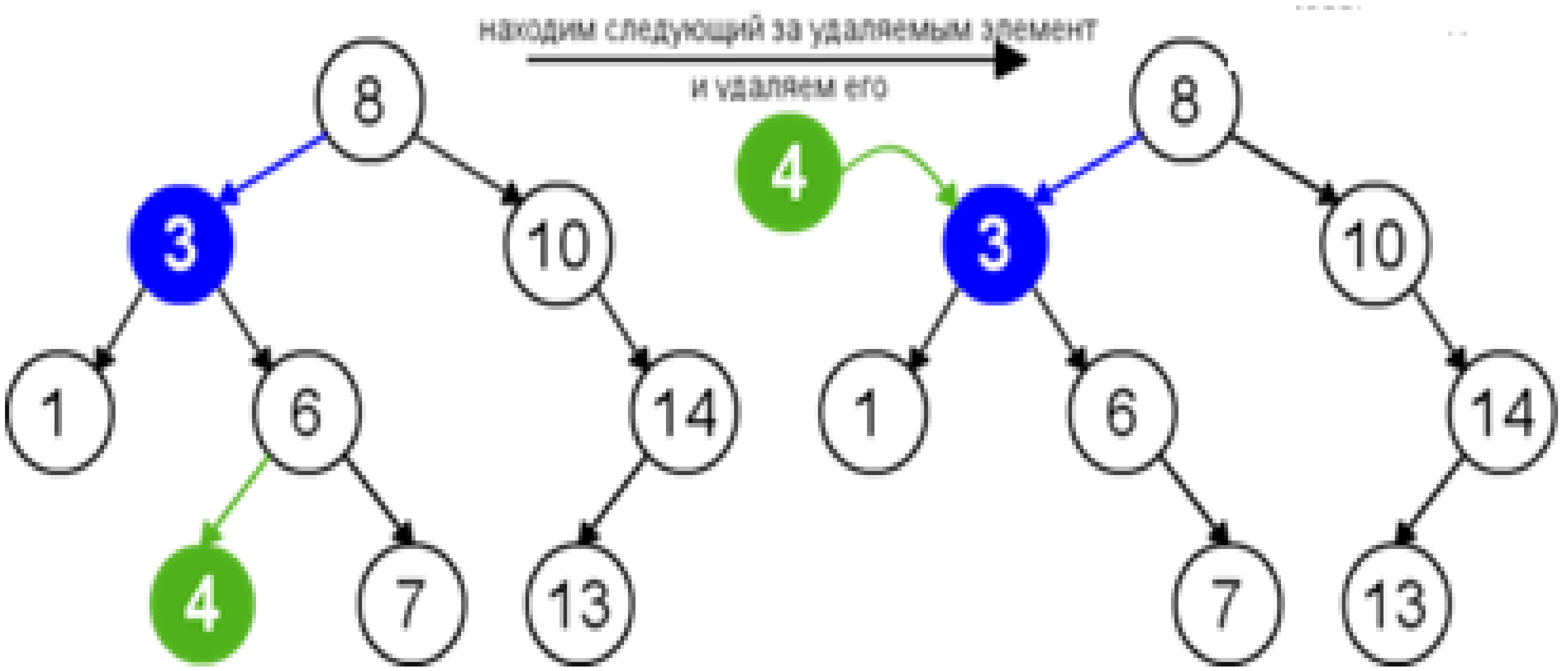
▶ есть правый ребенок

. . .

3) У узла оба наследника. В этом случае узел не удаляем, а заменяем его значение на минимум правого поддерева максимум левого поддерева). После этого удаляем минимум правого поддерева (максимум левого поддерева).

Минимум правого поддерева имеет не более одного наследника, так что он удаляется просто. Известно, что все значения справа от корня больше корня. Соответственно, минимум правого поддерева (максимум левого поддерева) будет, с одной стороны, меньше всех элементов правого поддерева, с другой стороны больше всех значений левого поддерева.

У узла оба наследника. В этом случае узел не удаляем,
а заменяем его значение на минимум правого поддерева
(максимум левого поддерева).



Удаляемый узел z key[z] = 3

(left [z] ≠ NIL и right[z] ≠ NIL)

y ← TREE_SUCCESSOR(z)

if left [y] = NIL **and** right[y] = NIL ***then***

key[z] ← key [y]

if p[y] ≠ z ***then***

left[p[y]]← NIL

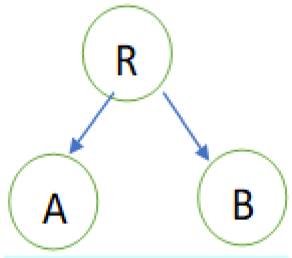
else

right [z] ← NIL

end if

end if

3 . Обход бинарного дерева поиска



Пусть имеем дерево, где

R — корень,

A и B — левое и правое поддеревья.

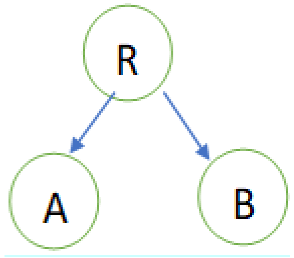
Рассмотрим три способа обхода дерева:

1. Обход дерева в прямом порядке (сверху вниз): R, A, B - префиксный

2. В симметричном порядке (слева направо):   A, R, B - инфиксный

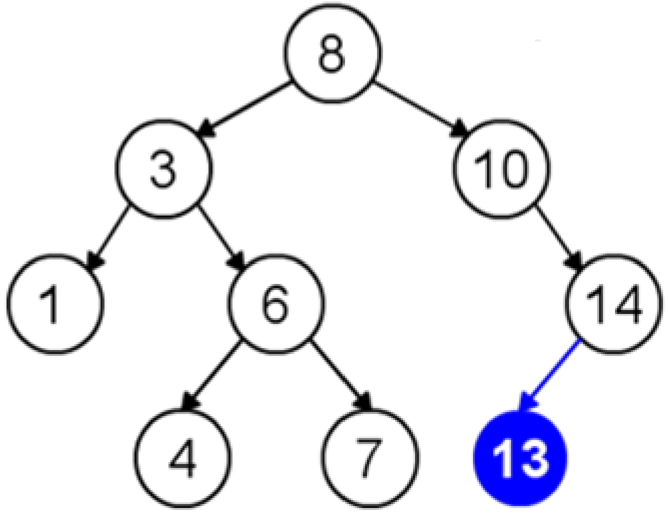
3. В обратном порядке (снизу вверх):   A, B, R - постфиксный

Обход дерева симметричном порядке (слева направо, инфиксный): A, R, B.



INORDER_TREE_WALK(x)

1. **If x ≠ NIL then**
2. INORDER_TREE_WALK (**left [x]**)
3. print **key[x]**
4. INORDER_TREE_WALK (**right [x]**)
5. **end if**



Ключи дерева в порядке возрастания!

1 3 4 6 7 8 10 13 14

4. Идеально сбалансированное бинарное дерево поиска

1. Создание шаблона для идеально сбалансированного дерева
2. Сортировка ключей по возрастанию
3. Инфиксный обход дерева и расстановка ключей

4.1 Создание шаблона для идеально сбалансированного дерева

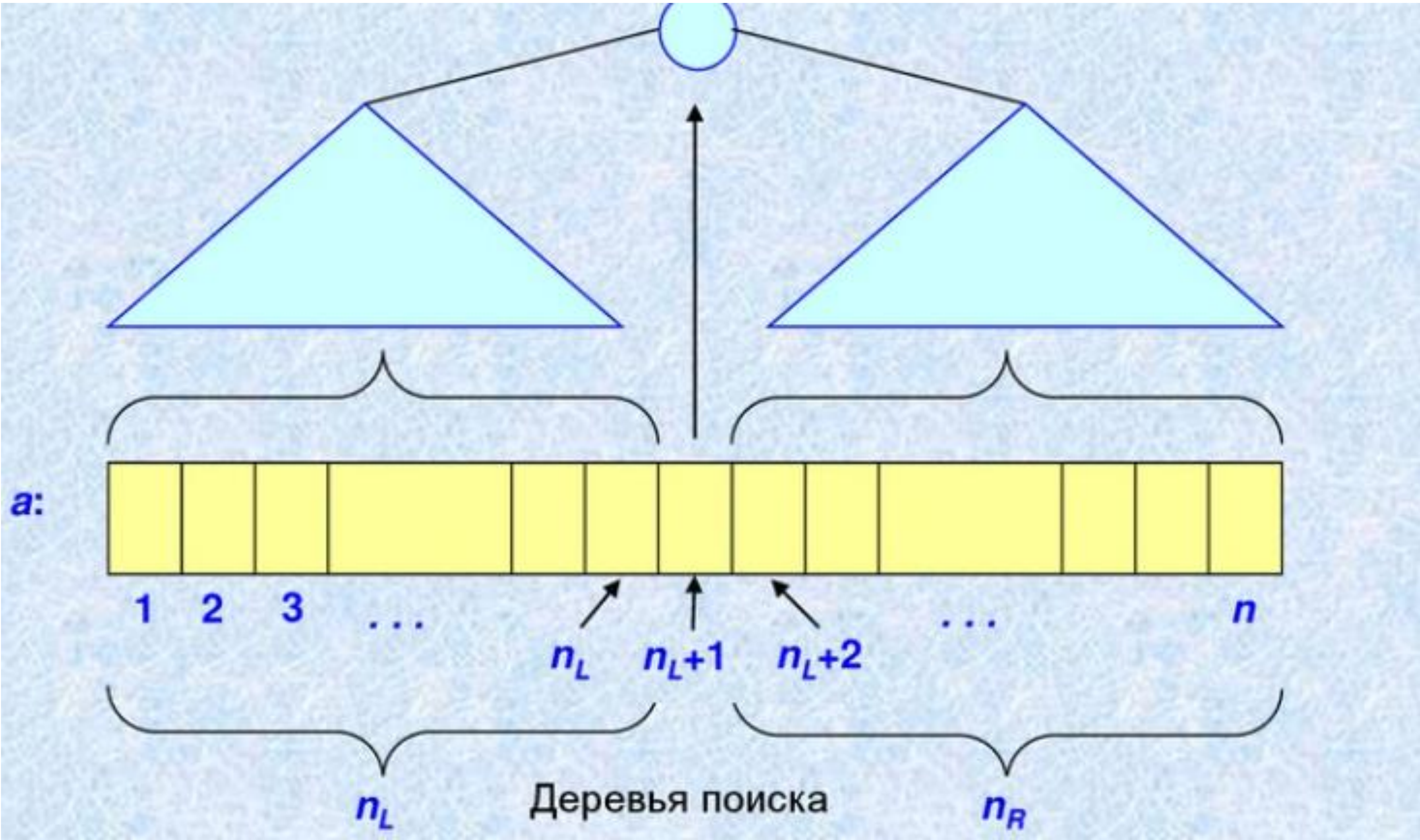
Идеально сбалансированными деревьями будем называть деревья, для которых для каждой вершины количество элементов в левом и правом поддереве отличается не более, чем на 1.

$$a_1, a_2, \dots, a_n$$

$$n_L = n \text{ div } 2$$

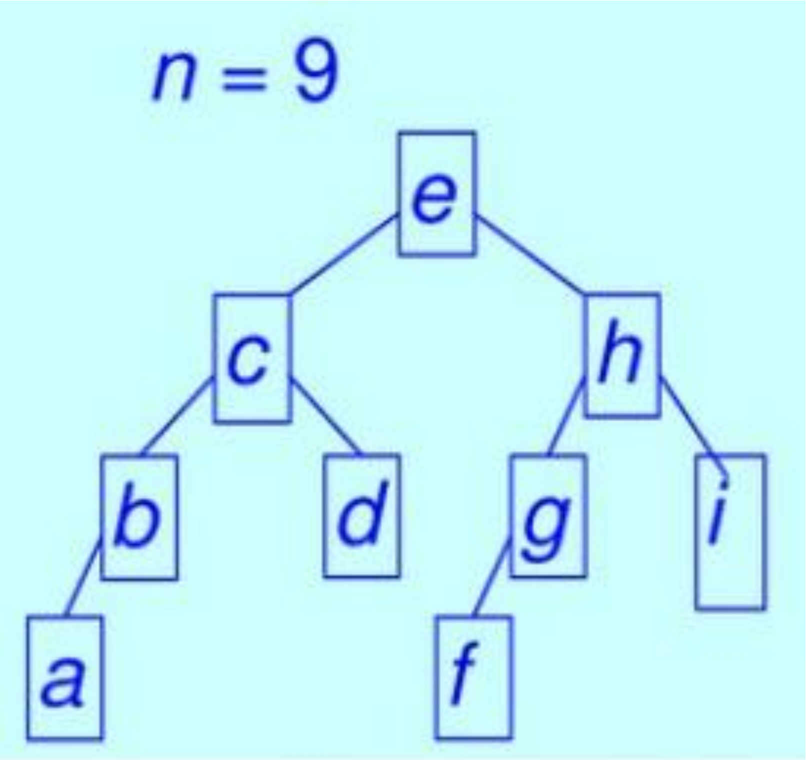
$$n_R = n - n_L - 1$$

$$n = n_L + n_R + 1$$

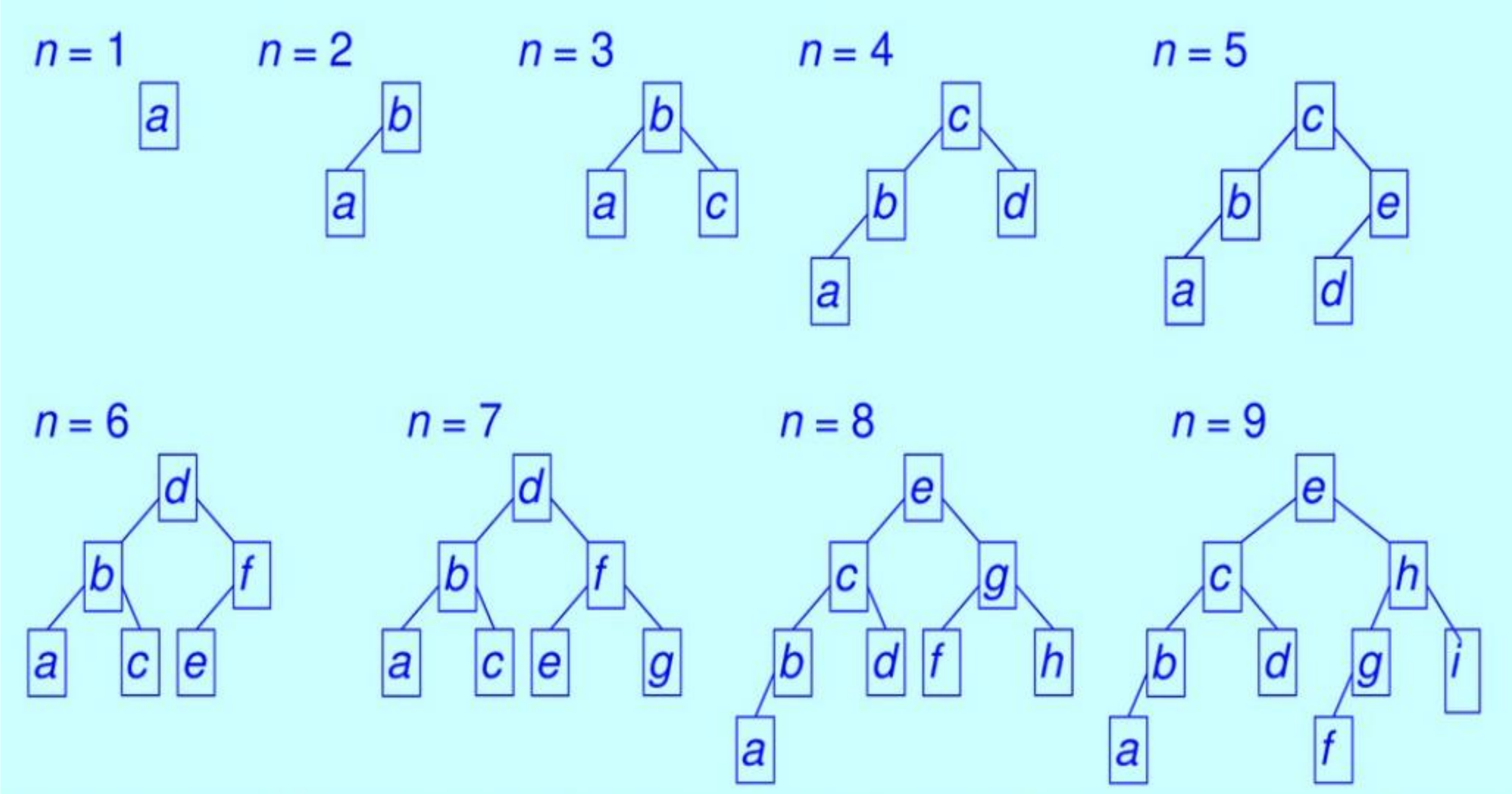


4.3 Обход инфиксный обход дерева и расстановка ключей

a, b, c, d, e, f, g, h, i.




a, b, c, d, e, f, g, h, i.


n = 1
a
n = 2
b
a
n = 3
b
a
c
n = 4
c
b
d
a
n = 5
c
b
e
a
d
n = 6
d
b
f
a
c
e
n = 7
d
b
f
a
c
e
g
n = 8
e
c
g
b
d
f
h
a
n = 9
e
c
h
b
d
g
i
a
f

5. Реализация бинарного дерева поиска на С++

Для описания узла дерева использовать тип Node, в котором

key_ - значение ключа узла,

left_ - указатель на левое поддерево,

right_ - указатель на правое поддерево,

p_ - указатель на родителя (может не использоваться).

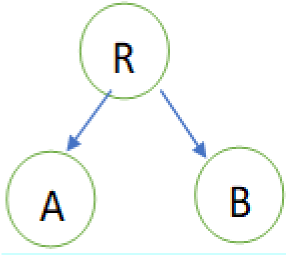
Тип Node может использоваться только в классе BinarySearchTree

```
#ifndef __BINARY_SEARCH_TREE_H
#define __BINARY_SEARCH_TREE_H
template <class T>
class BinarySearchTree
{
private:
  // Описание структуры узла со ссылками на детей
  struct Node {  . . . .
  };
  // Дерево реализовано в виде указателя на корневой узел.
  Node *root_;
```

```
// Описание структуры узла со ссылками на детей
 struct Node {
   T  key_;          // значение ключа, содержащееся в узле
   Node *left_;      // указатель на левое поддерево
   Node *right_;     // указатель на правое поддерево
   Node *p_;         // указатель на  родителя !!! не используется
   // Конструктор узла
   Node(T key, Node *left = nullptr, Node *right= nullptr,Node *p =nullptr):
             key_(key), left_ (left),  right_(right),  p_(p)
     {  }
 };
```

5. Реализация бинарного дерева поиска на C++

```
public:

  // Конструктор "по умолчанию" создает пустое дерево
  BinarySearchTree() : root_( nullptr) {}

  // Деструктор освобождает память, занятую узлами дерева
  virtual ~BinarySearchTree();

  // Печать строкового изображения дерева в выходной поток out
  void print (ostream & out) const ;

  // Функция поиска по ключу в бинарном дереве поиска
  bool searchIterative (const T & key)  const ;

  // Вставка нового элемента в дерево, не нарушающая порядка элементов.
  void insert (const T& key);
....
```




```
// private: Рекурсивная функция определения количества узлов дерева
size_t getCountSubTree (Node *node) const
{
    if (node == nullptr) {
       return 0;
    }
    return (1 + getCountSubTree(node->left_) + getCountSubTree(node->right_));
  }
 //
 // public: Определение количества узлов дерева
 size_t getCount () const
 {
     return getCountSubTree(this->root_);
 );
```

```
int main() {
  BinarySearchTree <int> t; // создаем пустое дерево
  t.insert(15);             // добавляем узлы
  t.insert(3);
  t.insert(20);
  // Поиск в дереве по ключу
  int  keyFound = t.searchIterative(15); // поиск должен быть успешным
  cout << "Key:" << 15 <<
              (keyFound ? " found successfully" : " not found")  <<
              " in the tree" << endl;
  // Определение числа узлов дерева
  cout << "count =  " << t.getCount() << endl;
```

```
 // public
template<class T>
void BinarySearchTree <T>::inorder (void (*visit)(T))
{
  inorder (root_, visit);
}
// private
template<class T>
void BinarySearchTree ::inorder (Node<T> *node, void(*visit) (T))
{
  if (node != nullptr) {
    inorder (node->left_, visit);
    (*visit) (node->key);
    inorder (node->right_, visit);
  }
}
```

```
 // public
template<class T>
void BinarySearchTree <T>::inorder (void (*visit)(T))
{
  inorder (root_, visit);
}
// private
template<class T>
void BinarySearchTree ::inorder (Node<T> *node, void(*visit) (T))
{
  if (node != nullptr) {
    inorder (node->left_, visit);
    (*visit) (node->key);
    inorder (node->right_, visit);
  }
}
```

```
void printKey(int x) {
  cout << x << "\n";
}

BinarySearchTree <int> treeTest; . . .

treeTest. inorder (printKey);
```