Алгоритмы и структуры данных

Практическая работа№8

Нелинейные динамические структуры данных. Бинарные деревья

Цель.Закрепить материал лекции4.

Отрабатываемые вопросы.

- Алгоритм построения бинарных деревьев поиска
- Алгоритм построения идеально сбалансированного бинарного дерева
- Алгоритмы обхода дерева
- Алгоритм вывода структуры дерева на экран

Задание.

Необходимо создать структуру с полями

1. ФИО - фамилия и инициалы, символьный массив.
2. Год рождения - целочисленная переменная.
3. Регион
4. Номер телефона - символьный массив.
5. Группа

Требуется создать четыре линейных списка, состоящих из 6 таких структур. Имя каждого файла должно начинаться с номера группы.

1) Ознакомиться с теоретическим материалом задания.
2) Разработать программу, которая:
    a. Считывает данные из примера ранее рассматриваемого текстового файла (*.txt) и формирует четыре линейные динамические структуры (либо связные списки, либо очереди, либо стеки), ключом формирования является номер группы (ИСТ-111, ИСТ-112, ИСТ-113, ИСТ-114). Данные считываются в том порядке, в каком они представлены в текстовом файле*.txt.
    b. Для каждого динамического списка (для каждой группы) программа строит и выводит на экран бинарное дерево поиска и сбалансированное бинарное деревопо ключу Фамилия.

Теоретический материал

Справочная литература:

1. Вирт Н. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.

2. Кнут Д. Искусство программирования для ЭВМ. Т.1: Основные алгоритмы. - М.: Мир,1976. - 736 с.

Дополнительные источники:

1. Язык программирования С++. Динамические структуры данных/Сайткафедры Информационных технологий КГУниверситета. [Электронный ресурс] - http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/oglav.html (дата просмотра 10.11.20

1 Структура вершины

Каждая вершина бинарного дерева является структурой, состоящей из четырех полей. Содержимым этих полей будут, соответственно:

- ***информационное поле*** (ключ вершины),
- ***служебное поле*** (их может быть несколько!),
- ***указатель на левое поддерево***,
- ***указатель на правое поддерево***.

```
struct node
    {
int Key; // Ключвершины.
intCount; // Счетчик количества вершин с одинаковыми ключами.
node *Left; // Указатель на "левого" сына.
node *Right; // Указатель на "правого" сына.
    };
```

Все функции по обработке узлов дерева реализуем через класс TREE и его методы (см. ниже).

```
class TREE
{
  private:
node *root; //Кореньдерева.
  public:
    …
}
```

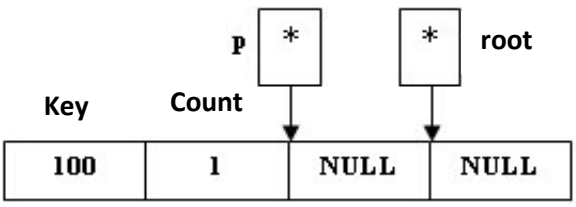
2 Алгоритм построения бинарного дерева поиска

Нерекурсивный алгоритм.

1. `root = NULL; //Построение пустого дерева`

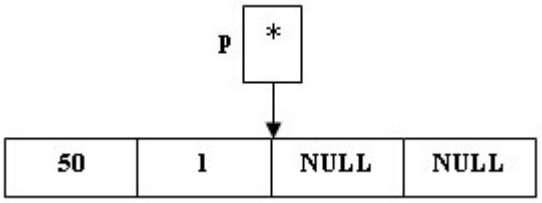
Пусть ключ первой, поступающей в дерево, вершины равен 100. Создаем первую вершину.

```
2.      p = new(node);
3.      (*p).Key = 100; (*p).Count = 1;
4.      (*p).Left = NULL; (*p).Right = NULL;
5. root = p;//эта вершина назначается корнем
```



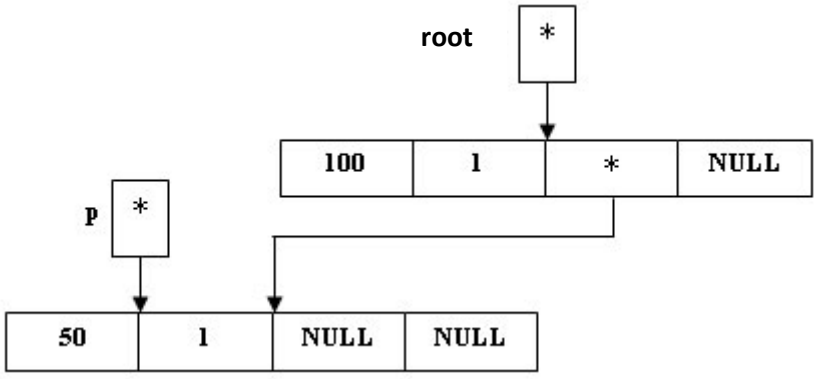
Пусть ключ второй, поступающей в дерево, вершины равен 50. Создаем новую вершину:

```
o  p = new(node);
o      (*p).Key = 50; (*p).Count = 1;
o      (*p).Left = NULL; (*p).Right = NULL;
```



так как 100>50, то по определению бинарного дерева поиска мы должны сделать вновь поступившую вершину левым сыном корня дерева:

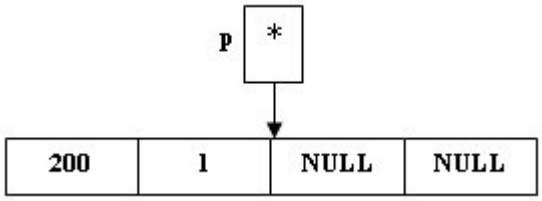
```
o       (*root).Left = p;
```



Пусть ключ третьей вершины, поступающей в дерево, равен 200. Порядок действий:

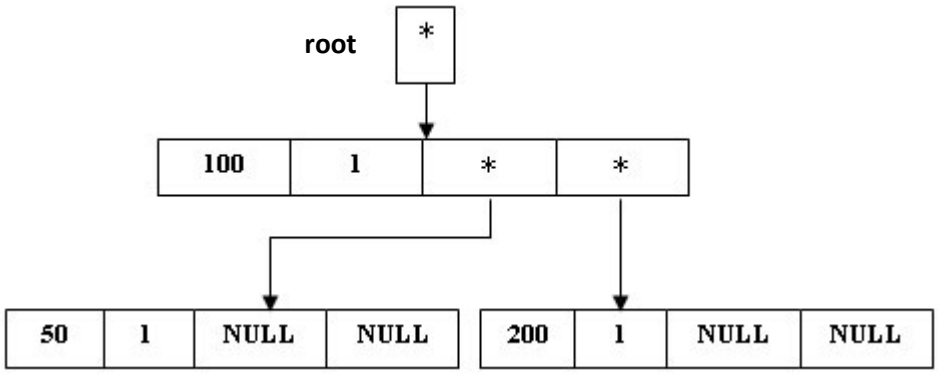
- создаем новую вершину:

```
o       p = new(node);
o       (*p).Key = 200; (*p).Count = 1;
o       (*p).Left = NULL; (*p).Right = NULL;
```



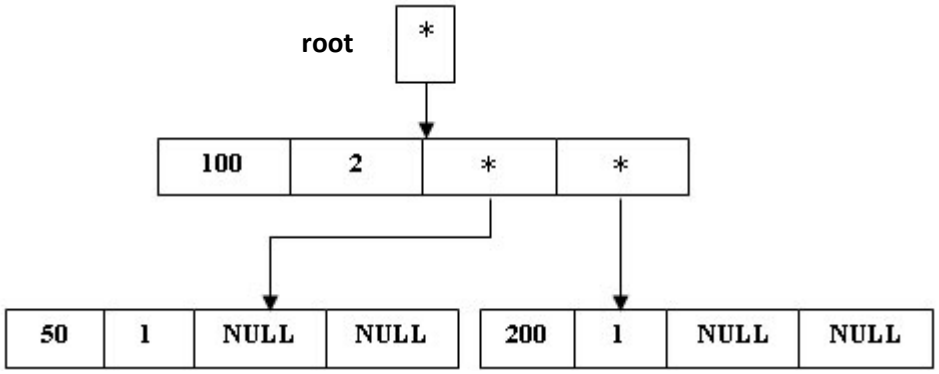
так как 200>100, то по определению бинарного дерева поиска, мы должны сделать вновь поступившую вершину правым сыном корня дерева:

```
o       (*root).Right = p;
```



Осталось определить, как действовать в случае поступления, например, в дерево вновь вершины с ключом 100. Оказывается, что поле **Count** и применяется для учета повторяющихся ключей!

Точнее, в этом случае выполняется оператор присваивания **(*root).Count = (*root).Count + 1;**, результат выполнения которого изобразим на схеме:



Таким образом, последовательное поступление вершин с ключами 100, 50, 200, 100 приведет к созданию структуры данных - ***бинарного дерева поиска***.

Рекурсивный алгоритм.

Опишем класс и методы построения дерева:

```
class TREE1
{
private:
node *Root; //Указатель на корень дерева.
voidSearch(int,node**); //метод поиска вершины с данным ключом
public:
TREE1() {Root=NULL;}
node** GetTree () {return&Root;} //Полученивершиныдерева.
voidBuildTree (); //метод построения дерева

}
voidTREE1::BuildTree()
// Построение бинарного дерева (рекурсивный алгоритм).
// Root - указатель на корень дерева.
{
intel;//переменная для ввода данных (целых чисел) элементов дерева

cout<<" Вводите ключи (данные) вершин дерева (0 - окончание ввода)...\n";
cin>>el;
while  (el!=0)//пока не введено значение 0
```

```
{ Search(el,&Root); //вызов метода вставки вершины в дерево
cin>>el; }
}

void TREE1::Search(int x,node **p)
//  Поиск вершины с ключом X в дереве со вставкой(рекурсивный алгоритм)
// *p - указатель на корень дерева.
{
if  (*p==NULL)
  {// Вершины в дереве нет; включить ее.
*p = new(node);
    (**p).Key = x;     (**p).Count = 1;
    (**p).Left = NULL; (**p).Right = NULL; }
else // проверяем место вставки (слева/справа) и наличие такой же вершины
if  (x<(**p).Key)
Search(x,&((**p).Left));//если новое значение меньше - левое поддерево
else
if  (x>(**p).Key)
Search(x,&((**p).Right)); //если значение больше - правое поддерево
else  (**p).Count = (**p).Count + 1; //если вершина уже есть
}
```

3 Алгоритм построения идеально сбалансированного дерева

Бинарное дерево называется ***идеально сбалансированным***, если для ***каждой его вершины количество вершин в левом и правом поддереве различаются не более чем на 1***.

Алгоритм построения идеально сбалансированного дерева при известном числе вершин **n** лучше всего формулируется с помощью рекурсии. При этом необходимо лишь учесть, что для достижения минимальной высоты при заданном числе вершин, нужно располагать максимально возможное число вершин на всех уровнях, кроме самого нижнего. Это можно сделать очень просто, если распределять все поступающие в дерево вершины поровну слева и справа от каждой вершины.

Суть алгоритма:

- взять одну вершину в качестве корня (например, первую).
- построить левое поддерево с **nl = n DIV 2** вершинами тем же способом.
- построить правое поддерево с **nr = n-nl-1** вершинами тем же способом.

Для реализации опишем класс и его методы:

```
class TREE
{
  private:
    node *root; //Кореньдерева.
  public:
TREE() { root = NULL; }
    node **Get_root() { return &root; }//полученииеадресавершины
    node *Tree (int, node **); //построенидерева
    void Vyvod(node **, int); //выводдерева
};
node *TREE::Tree (int n,node **p)
// Построение идеально сбалансированного дерева с n вершинами.
// *p - указатель на корень дерева.
{
  node *now;
  int nl,nr;//int x
int x;
  now = *p;
if  (n==0) *p = NULL;
  else
{
nl = n/2; nr = n - nl - 1;//находим середину
cin>>x;
    now = new(node);
(*now).Key = x;//первый узел - корень
Tree(nl,&((*now).Left));//половина - по левым ветвям поддеревьев
Tree(nr,&((*now).Right));//оставшиеся - по правым ветвям поддеревьев
```

```
*p = now;
  }
}
```

4 Алгоритм вывода дерева на экран

(1. Пример выводаидеально сбалансированного дерева)

Рекурсивно выводятся сначала правые вершины поддеревьев, начиная с листьев, затем - левые вершины поддеревьев, начиная с листьев.

```
void TREE::Vyvod(node **w,intl)
// Изображение бинарного дерева, заданногоуказателем *w на экране ПК.
{
if  (*w!=NULL)
  {
Vyvod(&((**w).Right),l+1);
for  (int i=1; i<=l; i++) cout<<"   ";
cout<<(**w).Key<<endl;
Vyvod(&((**w).Left),l+1);
}
}
```

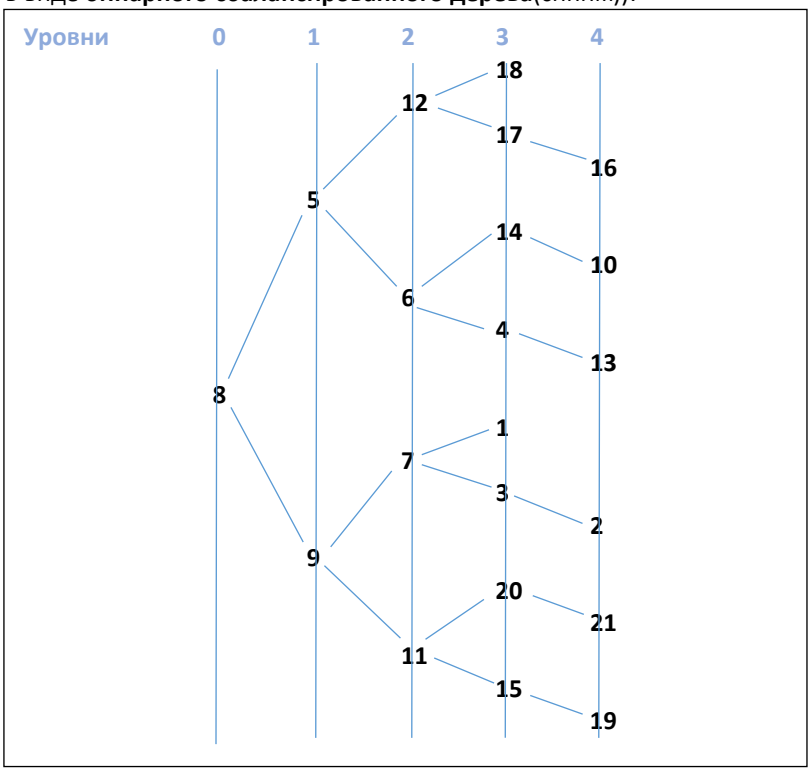
Результат вывода.

Если вершин 21 и они введены в таком порядке:

8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18

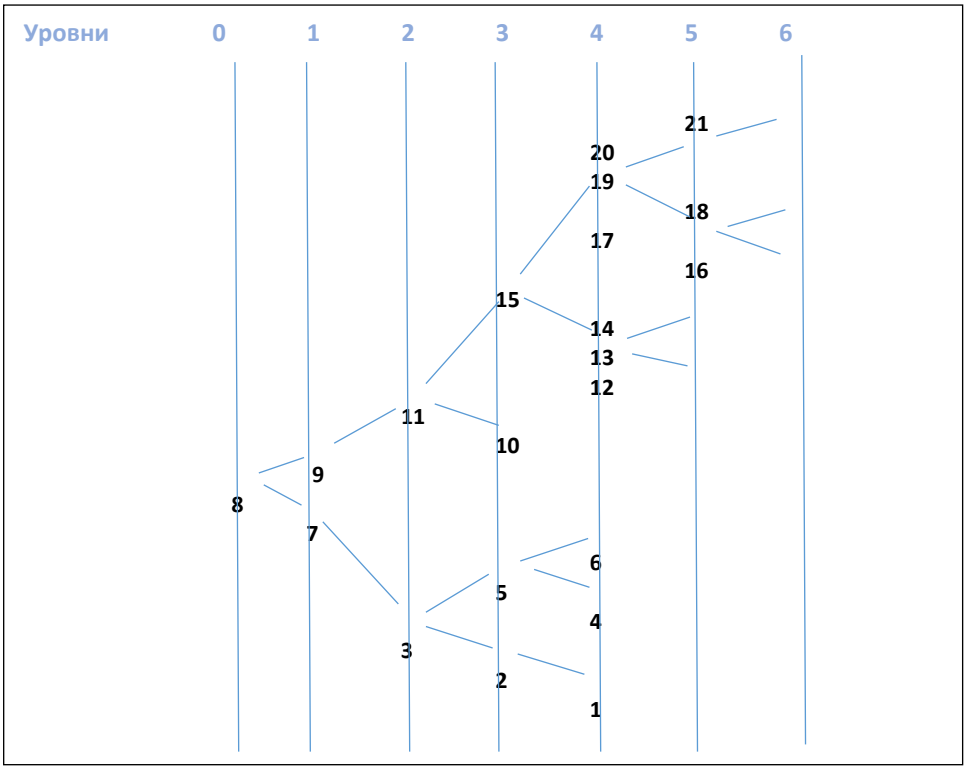
то при выводе на экран корень располагается в крайней левой позиции, а все его ветви –справа от него с отступом на каждом уровне.

Построение и вывод сбалансированного дерева (фактический вывод (черным) и условная схема структуры вывода в виде **бинарного сбалансированного дерева**(синим)):



(2. Пример выводабинарного дерева поиска)

Аналогичный метод для дерева бинарного поиска при тех же исходных данных выведет **бинарное дерево поиска,** но **не сбалансированное**:


Уровни
0
1
2
3
4
5
6
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1