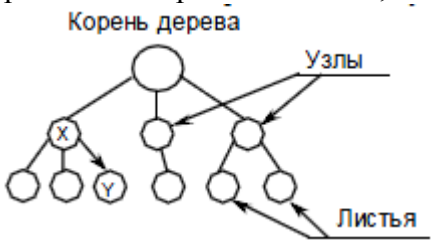
Лабораторная работа №6. Программирование с использованием деревьев

Теория.

Довольно часто при работе с данными бывает удобно использовать структуры с иерархическим представлением, изображенные следующим образом:



Такая конструкция данных получила название «дерево».

Дерево состоит из элементов, называемых узлами (вершинами). Узлы соединены между собой направленными дугами. В случае X→Y вершина X называется предком (родителем), а Y – потомком (сыном).

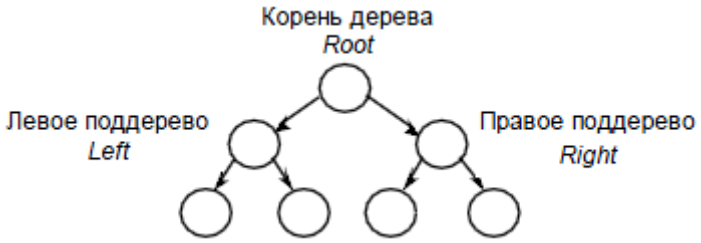
Дерево имеет единственный узел, у которого нет предков. Такой узел называют корнем. Любой другой узел имеет ровно одного предка. Узел, не имеющий потомков, называется листом.

Внутренний узел – это узел, не являющийся ни листом ни корнем. Порядок узла – количество его узлов-потомков. Степень дерева – максимальный порядок его узлов. Глубина узла равна числу его предков плюс один. Глубина дерева – это наибольшая глубина его узлов.

Бинарные деревья

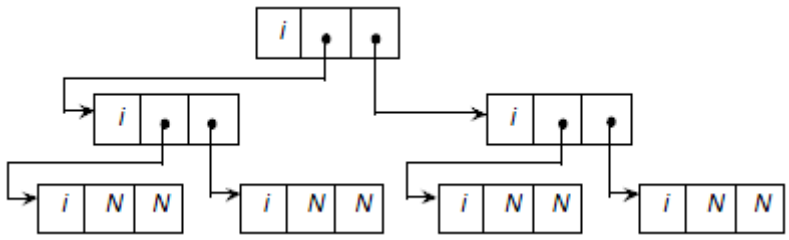
Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на бинарные поддеревья, т.е. у каждого узла (предка) может быть не более двух потомков – левый и правый. На каждый узел бинарного дерева имеется ровно одна ссылка.

Начальный же узел, как уже известно, называется корнем дерева.

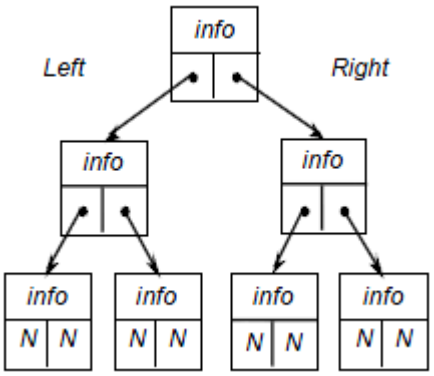


На рисунке приведен пример бинарного дерева (корень обычно изображается сверху, хотя изображение можно и перевернуть).

Такая структура данных организуется следующим образом:



или, соблюдая более привычную форму:



Высота дерева, как и раньше, определяется количеством уровней, на которых располагаются его узлы.

Если дерево организовано так, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева – больше, оно называется деревом поиска. Одинаковые ключи здесь не допускаются.

Представление динамических данных в виде древовидных структур оказывается довольно удобным и эффективным для решения задач быстрого поиска информации.

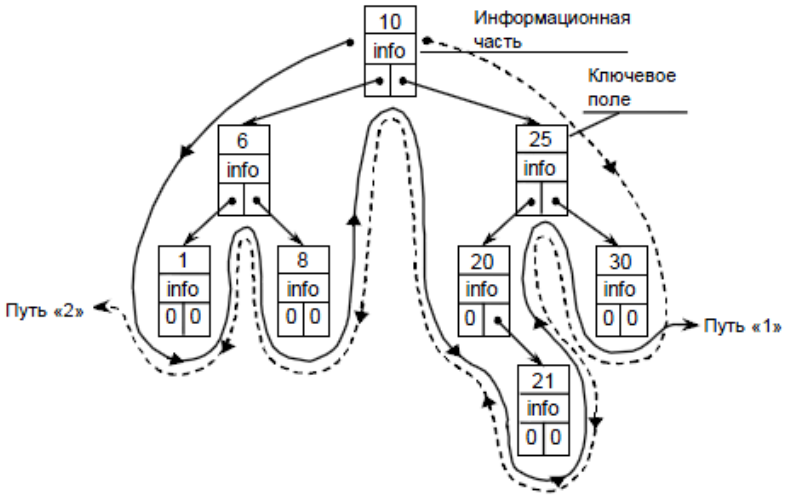
В дереве поиска можно найти элемент по ключу, двигаясь от корня, и, переходя на левое или правое поддерево, в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна логарифму количества узлов – log2N. Время поиска по сравнению с линейной структурой сокращается с N до log2N.

Для так называемого сбалансированного дерева, в котором количество узлов справа и слева отличается не более чем на единицу, высота дерева как раз и равна двоичному логарифму количества узлов. Линейный список можно представить как вырожденное бинарное дерево, в котором каждый узел имеет не более одной ссылки. Для списка среднее время поиска равно ровно половине длины этого списка.

Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. Действия с такими структурами изящнее всего описываются с помощью рекурсивных алгоритмов. Например, функцию обхода всех узлов дерева можно в общем виде описать так:

```
type way_around (дерево) {
        way_around (левое поддерево);
  посещение корня;
        way_around (правое поддерево);
        }
```

Можно обходить дерево и в другом порядке, например, сначала корень, потом поддеревья, но приведенная функция позволяет получить на выходе отсортированную последовательность ключей, поскольку сначала посещаются вершины с меньшими ключами, расположенные в левом поддереве, а потом вершины правого поддерева с большими ключами. На приведенном ниже рисунке это Путь «1».



Результат обхода дерева, изображенного на рисунке:

Путь «1»: 1→ 6→ 8→ 10→ 20→ 21→ 25→ 30

Если в функции обхода первое обращение идет к правому поддереву, результат обхода будет таким:

Путь «2»: 30→ 25→ 21→ 20→ 10→ 8→ 6→ 1

Таким образом, деревья поиска можно применять для сортировки значений (при обходе дерева узлы не удаляются).

Рассмотрим основные алгоритмы работы с бинарным деревом

Необходимо уметь:

– построить (создать) дерево;

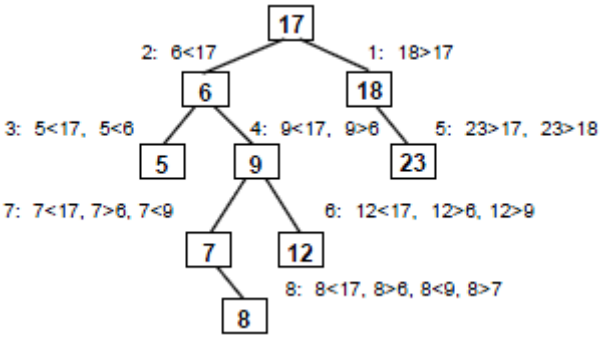
– вставить новый элемент;

– обойти все элементы дерева, например, для просмотра или чтобы произвести некоторую операцию;

– выполнить поиск элемента с указанным значением в узле;

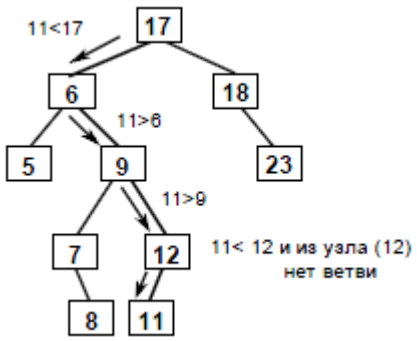
– удалить заданный элемент.

Обычно бинарное дерево строится сразу упорядоченным, т.е. узел левого сына имеет значение меньшее, чем значение родителя, а узел правого сына – большее. Например, если приходят числа 17, 18, 6, 5, 9, 23, 12, 7, 8, то построенное по ним дерево будет выглядеть следующим образом (для упрощения приводим только ключи):

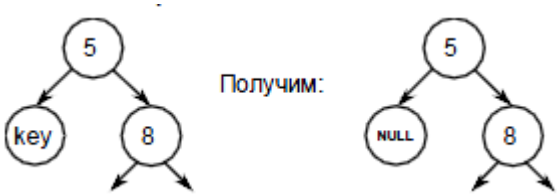


Для того чтобы вставить новый элемент в дерево, необходимо найти для него место. Для этого, начиная с корня, сравниваем значения узлов (Y) со значением нового элемента (New). Если New<Y, то идем по левой ветви, в противном случае – по правой ветви. Когда дойдем до узла, из которого не выходит нужная ветвь для дальнейшего поиска, это означает, что место под новый элемент найдено.

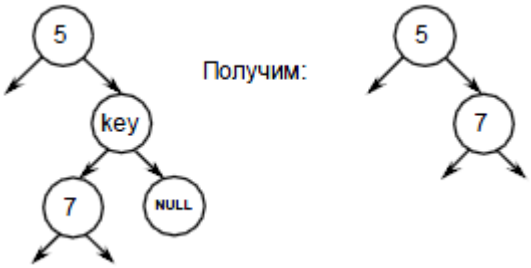
Путь поиска места в построенном дереве для числа 11:



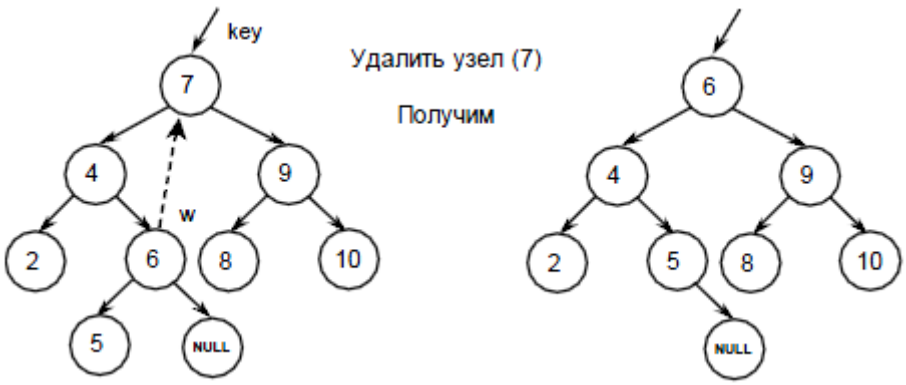
При удалении узла из дерева возможны три ситуации:
1. Удаляемый узел является листом – просто удаляем ссылку на него;
Удаление листа с ключом key:



2. Из удаляемого узла выходит только одна ветвь;
Удаление узла имеющего одного потомка:

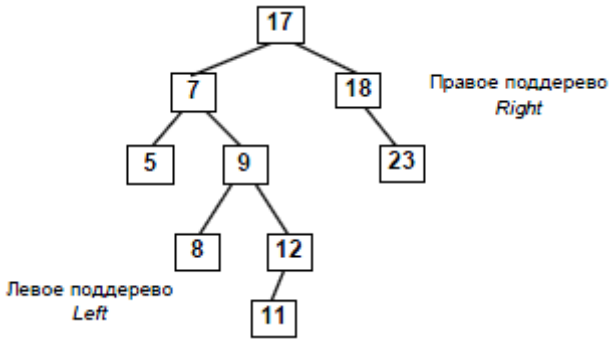


3. Удаление узла, имеющего двух потомков, значительно сложнее. Если key – исключаемый узел, то его следует заменить узлом w, который содержит либо наибольший ключ в левом поддереве, либо наименьший ключ в правом поддереве. Такой узел w является либо листом, либо самым правым узлом поддерева key, у которого имеется только левый потомок:



Таким образом, если из удаляемого узла выходит две ветви (в данном случае на место удаляемого узла надо поставить либо самый правый узел левой ветви, либо самый левый узел правой ветви для сохранения упорядоченности дерева).

Например, построенное дерево после удаления узла 6 может стать таким:



Рассмотрим задачу обхода дерева. Существуют три алгоритма обхода деревьев, которые естественно следуют из самой структуры дерева.

1) Обход слева направо: Left-Root-Right (сначала посещаем левое поддерево, затем – корень и, наконец, правое поддерево).

2) Обход сверху вниз: Root-Left-Right (посещаем корень до поддеревьев).

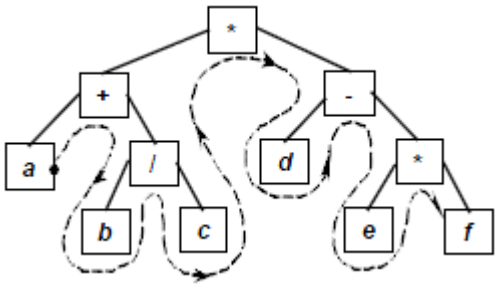
3) Обход снизу вверх: Left-Right-Root (посещаем корень после поддеревьев).

Интересно проследить результаты этих трех обходов на примере записи формулы в виде дерева, так как они и позволяют получить различные формы записи арифметических выражений.

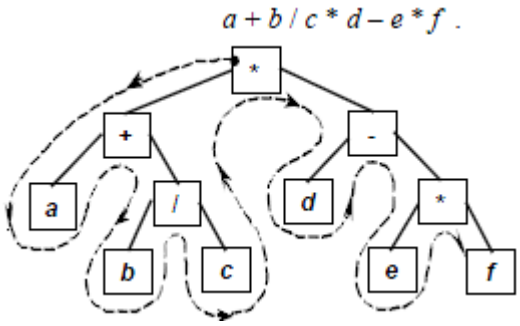
Рассмотрим на примере формулы: ((a+b/c)*(d–e*f)). Дерево формируется по принципу:

– в корне размещаем операцию, которая выполнится последней;

– далее узлы – операции, операнды – листья дерева.

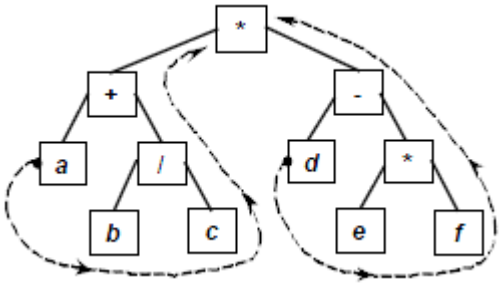


Обход 1 (Left-Root-Right) дает обычную инфиксную запись выражения (без скобок):



Обход 2 (Root-Left-Right) – префиксную запись выражения (без скобок):

* + a / b c – d * e f



Обход 3 (Left-Right-Root) – постфиксную (ПОЛИЗ – польская инверсная запись):
a b c / + d e f * – * .

Пример. Сформировать дерево из целых чисел, выполнить просмотр, добавление, удаление по ключу и освобождение памяти при выходе из программы.

Для каждого рекурсивного алгоритма можно создать его не рекурсивный эквивалент, и в приведенной ниже программе реализован не рекурсивный алгоритм поиска по дереву с включением и рекурсивная функция обхода дерева. Пример построения бинарного дерева, осуществляя поиск элемента с заданным ключом и, если элемент не найден – включает его в соответствующее место дерева.

Для включения элемента необходимо помнить пройденный по дереву путь на один шаг назад и знать, выполняется ли включение нового элемента в левое или правое поддерево его предка.

Рекурсии удалось избежать, сохранив всего одну переменную (Prev), и, повторив, при включении операторы, определяющие, к какому поддереву присоединяется новый узел.

В функции обхода дерева вторым параметром передается переменная, определяющая, на каком уровне (level) находится узел. Корень находится на уровне «0». Значения узлов выводятся по горизонтали так, что корень находится слева. Перед значением узла для имитации структуры дерева выводится количество пробелов, пропорциональное уровню узла. Если закомментировать цикл печати пробелов, вводимые значения ключей будут выведены в столбик.

Текст программы:

```
#include <stdio.h>
#include <conio.h>
struct Tree {
int info;
Tree *Left, *Right;
} *Root; // Root – указатель на корень
void Make(int);
void Print (Tree*, int);
void Del(int);
void Del_All(Tree*);
Tree* List(int);
void main() {
int b, found, key;
// b – для ввода ключей, found – код поиска, key – удаляемый ключ
while(1) {
puts(" Creat - 1\n View - 2\n Add - 3 \nDel Key - 4\n EXIT - 0");
switch (getch()) {
case '1': Make(0); break;
case '2': if( Root == NULL ) puts ("\t END TREE !");
else Print(Root, 0);
break;
case '3': if(Root==NULL) Make(0);
else Make(1);
```

```
break;
case '4': puts("\n Input Del Info "); scanf("%d", &key);
Del(key);
break;
case '0': Del_All(Root);
puts("\n Tree Delete!");

return;
} // End switch
} // End while(1)
}
//============ Создание дерева ==========================
void Make(int kod) {
Tree *Prev, *t, *t1;
int b,found;
if ( kod == 0 ) { // Формирование первого элемента
puts( "\n Input Root :");
scanf("%d", &b);
Root = List(b); // Установили указатель корня
}
//============== Вставка остальных элементов ===============
while(1) {
puts( "\n Input Info :"); scanf("%d", &b);
if (b<0) break;
t = Root;
found = 0;
while ( t && !found ) {
Prev = t;
if( b == t->info) found = 1;
else
if ( b < t -> info ) t = t -> Left;
else t = t -> Right;
}
if (!found) {
t1 = List(b); // Создаем новый узел
if ( b < Prev -> info ) Prev -> Left = t1;
else Prev -> Right = t1;
}
} // Конец цикла
}
//============== Удаление элемента по ключу (не корень) ===============
void Del(int key) {
Tree *Del,*Prev_Del,*R,*Prev_R;
// Del, Prev_Del - удаляемый элемент и его предок;
// R, Prev_R - элемент, на который заменяется удаленный и его предок;
Del = Root; Prev_Del = NULL;
//-------- Поиск удаляемого элемента и его предка по ключу key----------------------
while (Del != NULL && Del -> info != key) {
Prev_Del = Del;
if (Del->info >key) Del=Del->Left;
else Del=Del->Right;
```

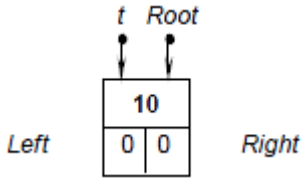
```
}
if (Del==NULL){ // В дереве такого ключа нет
puts ( "\n NO Key!");
return;
}
//---------------------- Поиск элемента для замены R ---------------------------------------
//----------1. Если удаляемый элемент имеет одного потомка, или ЛИСТ -----------
if (Del -> Right == NULL) R = Del->Left;
else
if (Del -> Left == NULL) R = Del->Right;
else {
//----------------Иначе, ищем самый правый узел в левом поддереве------------------
Prev_R = Del;
R = Del->Left;
while (R->Right != NULL) {
Prev_R=R;
R=R->Right;
}
//------------------2. Если удаляемый элемент имеет одного потомка------------------
if( Prev_R == Del) R->Right=Del->Right;
else {
//------------------3. Если удаляемый элемент имеет двух потомков ------------------
R->Right=Del->Right;
Prev_R->Right=R->Left;
R->Left=Prev_R;
}
}
// Устанавливаем связь с предком удаляемого (Prev_Del) и заменой (R):
if (Prev_Del==NULL) { Root = Prev_Del = R; }
else
if (Del->info < Prev_Del->info) Prev_Del->Left=R;
else Prev_Del->Right=R;
printf("\n Delete %d element ",Del->info);
delete Del;
}
//============== Формирование (создание) элемента - листа ==============
Tree* List(int i) {
Tree *t = new Tree; // Захват памяти
t -> info = i;
t -> Left = t -> Right = NULL;
return t;
}
//============ Функция вывода на экран ========================
void Print ( Tree *p, int level ) {
if ( p ) {
Print ( p -> Right , level+1); // Вывод левого поддерева
for ( int i=0; i<level; i++) printf(" ");
printf("%d \n", p->info);
Print( p -> Left , level+1); // Вывод правого поддерева
}
}
//==================== Освобождение памяти ==================
```

```
void Del_All(Tree *t) {
if ( t != NULL) {
Del_All ( t -> Left);
Del_All ( t -> Right);
delete t;
}
}
```

Пояснение к программе

Вводим ключи: 10, 25, 20, 6, 21, 8, 1, 30.

1. Формирование первого элемента (корня) со значением 10 приведет к следующему виду (0 – NULL):



Далее, при определении, в какое место (в левое или правое поддерево) добавлять очередной элемент, воспользуемся тем, что бинарное дерево строится упорядоченным по ключевому полю:

– узел левого сына (потомка) должен быть меньше узла родителя (предка);

– узел правого сына – больше узла родителя.

Рассмотрим несколько шагов цикла while.

Шаг 1: b = 25;

1. Установили текущий указатель на корень: Tree *t = Root;

2. Флаг для контроля поиска int found=0; чтобы исключить повторение ключей.

3. Внутренний цикл while, выполнять пока текущий указатель t != NULL и искомый ключ не найден (!found);

4. Установили указатель на предка на текущий элемент: Prev = t;

5. Проверяем значение текущего ключа:

1) если b = t -> info, то found = 1; else – игнорируется, вставку не производим, т.к. такой уже есть;

2) если found=1, уходим на новый виток цикла;

3) иначе (25 не равно 10),

если b< t->info, то создаем левую ветвь (t = t->Left), иначе – создаем правую ветвь (t = t->Right).

6. Создаем новый элемент:

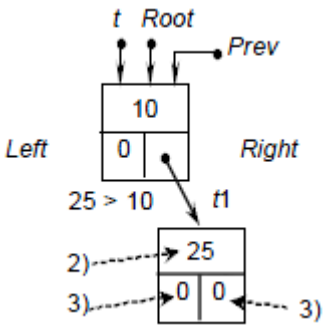
1) захватываем память: Tree *t1 = new Tree;

2) формируем информационную часть: t1 -> info = b; (25)

3) указатели на потомков в NULL:

t1 -> Left = t1->Right = NULL;

На этом этапе получили следующее:



Шаг 2: i = 2, b = 20;

1. Tree *t = Root;

2. found = 0;

3. Внутренний цикл while, выполнять пока t!=0 и !found;

Prev = t;

4. Проверяем значение текущего ключа:

1) (шаг 1.1): если b[i] ?= t -> info (10?=20), – Нет!

иначе: если b[i] ?< t->info (20?<10) – Нет! значит:

t = t->Right – создаем правую ветвь;

2) (шаг 1.2): если b[i] ?= t -> info (20?=25), – Нет!

иначе: если b[i] ?< t->info (20?<25) – Да! значит:

t = t->Left = NULL; – указатель на правую ветвь;

3) конец цикла while.

5. Создается по уже рассмотренному алгоритму новый элемент и присоединяется слева от последнего:

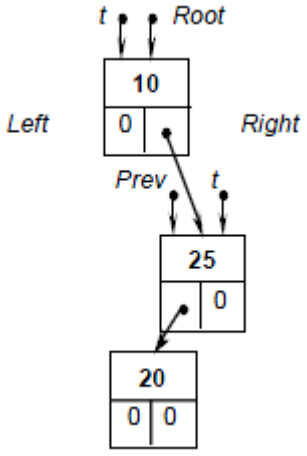
t1 = new Tree;

t1 -> info = b; (20)

t1 -> Left = t1 -> Right = NULL;

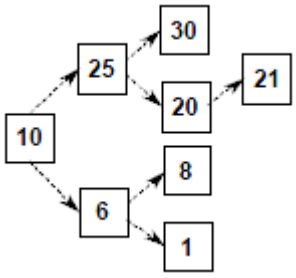
Prev -> Left = t1;

Графически это выглядит следующим образом:



и т.д.

Результат работы программы для ключей: 10, 25, 20, 6, 21, 8, 1, 30 (стрелками показана связь между выводимыми значениями):



Задание.

Исходная информация в виде массива находится в компоненте StringGrid. Каждый элемент массива содержит строку текста и целочисленный ключ (например, Ф.И.О. и номер паспорта).

Разработать класс для работы с деревом поиска, содержащий следующие методы **(не использовать шаблоны STL)**:

- внести информацию из массива в дерево поиска;
- сбалансировать дерево поиска;
- добавить в дерево поиска новую запись;
- по заданному ключу найти информацию в дереве поиска и отобразить ее;
- удалить из дерева поиска информацию с заданным ключом;
- распечатать информацию прямым, обратным обходом и в порядке возрастания ключа.

На основе родительского класса создать производный класс для решения задачи выбранного варианта.

Написать программу, иллюстрирующую все методы работы с деревом поиска. Результат формирования и преобразования дерева отображать в компонентах TTreeView и TMemo. Написать обработчик события, реализующий работу с методом решения своего варианта.

Индивидуальные задания

1. Поменять местами информацию, содержащую максимальный и минимальный ключи.
2. Подсчитать число листьев в дереве. (Лист – это узел, из которого нет ссылок на другие узлы дерева.)
3. Удалить из дерева ветвь с вершиной, имеющей заданный ключ.
4. Определить максимальную глубину дерева, т.е. число узлов в самом длинном пути от корня дерева до листьев.
5. Определить число узлов на каждом уровне дерева.
6. Удалить из левой ветви дерева узел с максимальным значением ключа и все связанные с ним узлы.
7. Определить количество символов во всех строках, находящихся в узлах дерева.
8. Определить число листьев на каждом уровне дерева.
9. Определить число узлов в дереве, в которых есть указатель только на одну ветвь.
10. Определить число узлов в дереве, у которых есть две дочери.
11. Определить количество записей в дереве, начинающихся с определенной буквы (например а).
12. Найти среднее значение всех ключей дерева и найти узел, имеющий ближайший к этому значению ключ.
13. Найти запись с ключом, ближайшим к среднему значению между максимальным и минимальным значениями ключей.
14. Определить количество узлов в левой ветви дерева.
15. Определить количество узлов в правой ветви дерева.