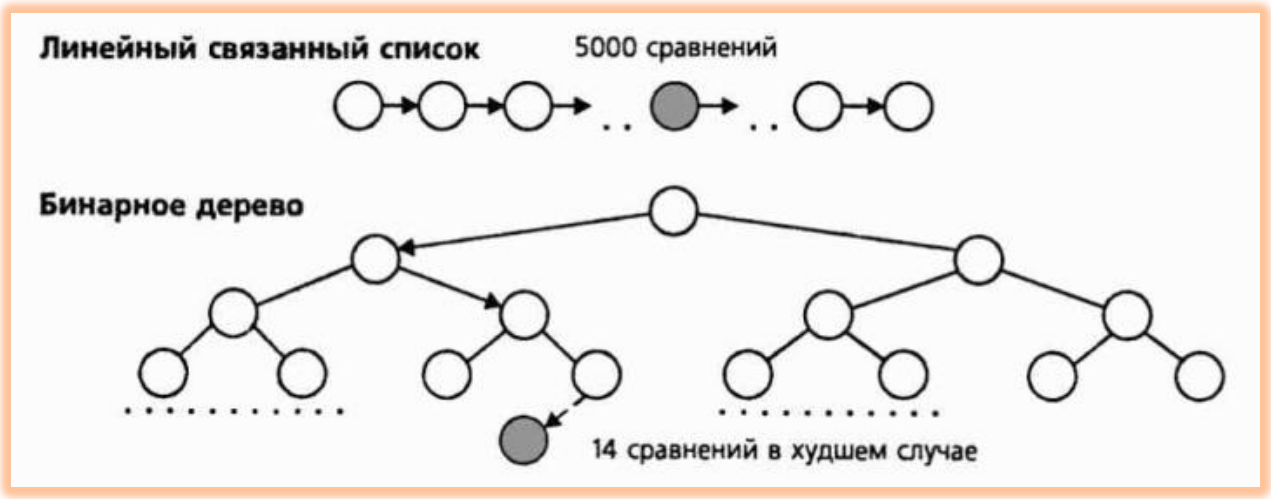
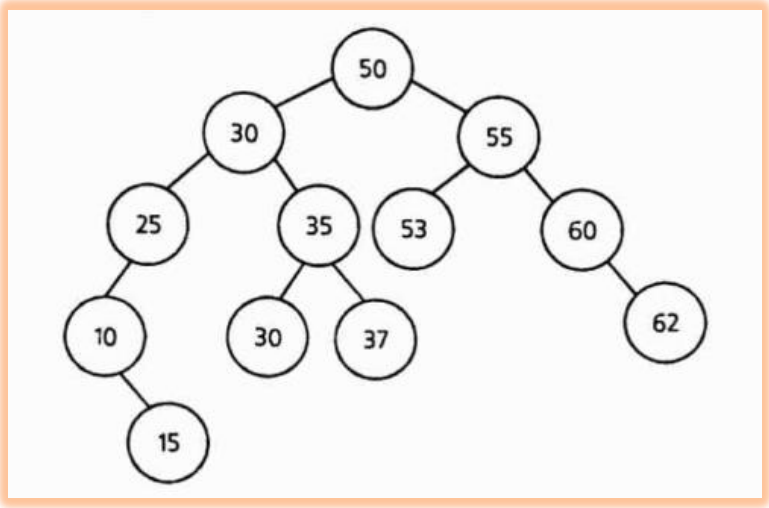
Отчёт №3. Бинарные деревья поиска

Обычное бинарное дерево может содержать большую коллекцию данных и все же обеспечивать быстрый поиск, добавление или удаление элементов. Одним из наиболее важных приложений деревьев является построение классов коллекций. Нам уже знакомы проблемы, возникающие при построении общего класса коллекций из класса SeqList и его реализации с помощью массива или связанного списка. Главную роль в классе SeqList играет метод Find, реализующий последовательный поиск. Для линейных структур сложность этого алгоритма равна O(N), что неэффективно для больших коллекций. В общем случае древовидные структуры обеспечивают значительно большую производительность, так как путь к любым данным не превышает глубины дерева. Эффективность поиска максимизируется при законченном бинарном дереве и составляет O(log2N). Например, в списке из 10000элементов предполагаемое число сравнений при последовательном поиске равно 5000. Поиск же на законченном дереве потребовал бы не более 14 сравнений. Бинарное дерево представляет большие потенциальные возможности в качестве структуры хранения списка.



Чтобы запомнить элементы в виде дерева с целью эффективного доступа, мы должны разработать поисковую структуру, которая указывает путь к элементу. Эта структура, называемая бинарным деревом поиска, упорядочивает элементы посредством оператора отношения <. Чтобы сравнить узлы дерева, мы подразумеваем, что часть или все поле данных определенно в качестве ключа и оператор < сравнивает ключи, когда размещает элемент на дереве. Бинарное дерево поиска строится по следующему правилу:

Для каждого узла значения данных в левом поддереве меньше, чем в этом узле, а в правом поддереве – больше или равны.



На рисунке показан пример бинарного поискового дерева. Это дерево называется поисковым потому, что в поисках некоторого элемента (ключа) мы можем идти лишь по совершенно конкретному пути. Начав с корня, мы сканируем левое поддерево, если значение ключа меньше текущего узла. В противном случае сканируется правое поддерево. Метод создания дерева позволяет осуществить поиск элемента по кратчайшему пути от корня. Например, поиск числа 37 требует четыре сравнения, начиная с корня.

Текущий узел	Действие
Корень = 50	Сравнить ключ = 37 и 50 поскольку 37 < 50, перейти в левое поддерево
Узел = 30	Сравнить ключ = 37 и 30 поскольку 37 >= 30, перейти в правое поддерево
Узел = 35	Сравнить ключ = 37 и 35 поскольку 37 >= 35, перейти в правое поддерево
Узел = 37	Сравнить ключ = 37 и 37. Элемент найден.

Ключ в узле бинарного поиска дерева

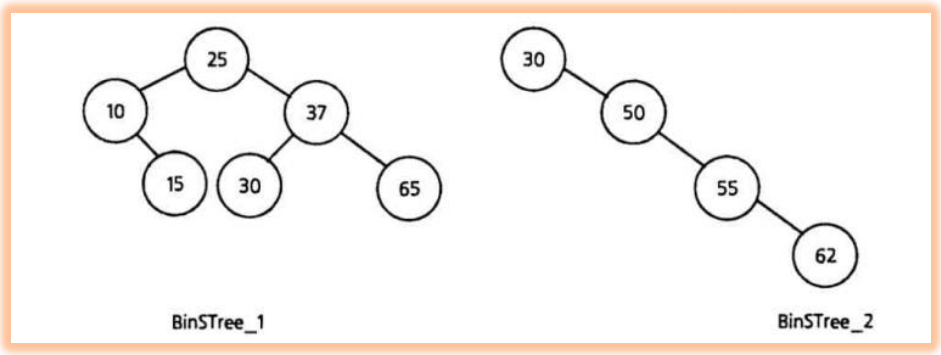
Ключ в поле данных работает как этикетка, с помощью которой можно идентифицировать узел. Во многих приложениях элементы данных являются записями, состоящими из отдельных полей. Ключ – одно из этих полей. Например, номер социальной страховки является ключом, идентифицирующим студента.

Номер социальной страховки (9-символьная строка)	Имя студента (строка)	Средний балл (число с плавающей точкой)

Ключевое поле

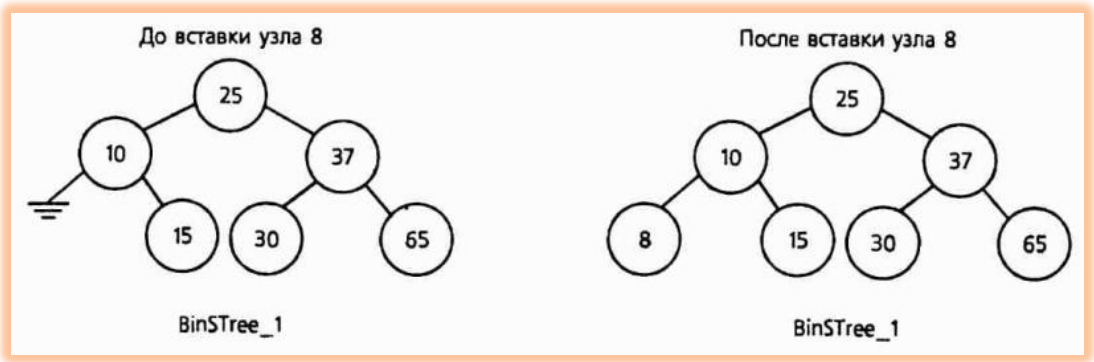
```
struct Student
{
  String ssn;
  String name;
  float  gpa;
}
```

Ключом может быть всё поле данных и только его часть. На рисунке узлы содержат единственное целочисленное значение, которое и является ключом. В этом случае узел 25 имеет ключ 25, и мы сравниваем два узла путем сравнения целых чисел. Сравнение производится с помощью целочисленных операторов отношения < и ==.

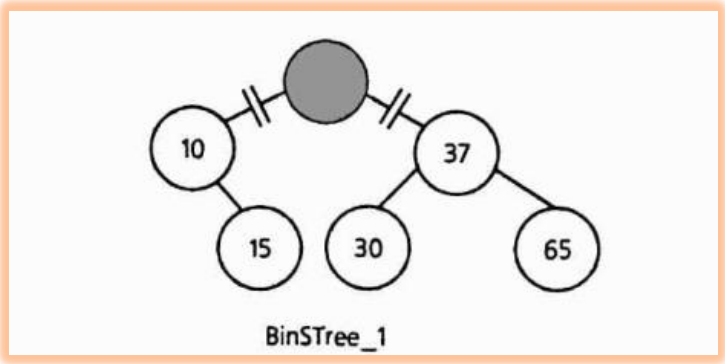


Операции на бинарном дереве поиска

Бинарное дерево поиска является нелинейной структурой для хранения множества элементов. Как и любая списковая структура, дерево должно допускать включение, удаление и поиск элементов. Для поискового дерева требуется такая операция включения (вставки), которая правильно располагает новый элемент. Рассмотрим, например, включение узла 8 в дерево BinSTree_1. Начав с корневого узла 25, определяем, что узел 8 должен быть в левом поддереве узла 10, которое в данный момент пусто. Узел 8 включается в дерево в качестве левого сына узла 10.

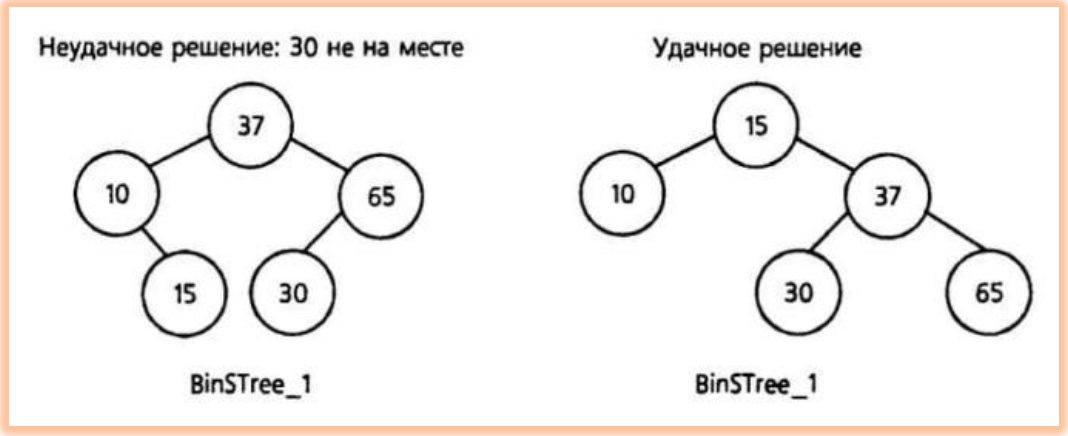


До каждого вставляемого в дерево узла существует конкретный путь. Тот же путь может использоваться для поиска элемента. Поисковый алгоритм берет ключ и ищет его в левом или в правом поддереве каждого узла, составляющего путь. Например, поиск элемента 30 на дереве BinSTree_1 начинается в корневом узле 25 и переходит в правое поддерево (30 >=25), а затем левое поддерево (30<37). Поиск прекращается на третьем сравнении, когда ключ совпадает с числом 30, хранящимся в узле.



В связанном списке операция удаления отсоединяет узел и соединяет его предшественника со следующим узлом. На бинарном дереве поиска подобная операция намного сложнее, так как узел может нарушить упорядочение элементов дерева. Рассмотрим задачу удаления корня 25 из BinSTree_1. В результате появляются два разобщенных поддерева, которым требуется новый корень.

На первый взгляд напрашивается решение выбрать сына узла 25 – скажем, 37 – и заменить его родителя. Однако это простое решение терпит неудачу, так как некоторые узлы оказываются не с той стороны корня. Поскольку данное дерево относительно невелико, мы можем установить, что 15 или 30 являются допустимой заменой корневому узлу.



Объявление АДТ деревьев

АДТ для списка строится по образцу класса SeqList. Тот факт, что бинарное дерево поиска хранит элементы данных в виде нелинейного списка, становится существенной деталью реализации его методов. Заметим, что этот АДТ является зеркальным отражением АДТ для класса SeqList, но имееет дополнительный метод Update, позволяющий обновлять поле данных, и метод GetRoot, предоставляющий доступ к корневому узлу, а следовательно, и функциям прохождения из treescan.h и к функциям печати из treeprint.h. Обратите внимание, что метод GetData класса SeqList отсутствует, так как он относится к линейному списку.

ADT для бинарных деревьев поиска

Данные

Список элементов, хранящийся в виде бинарного дерева, и значение size, определяющее текущее число элементов в списке. Дерево содержит указатель на корень и ссылку на последний обработанный узел — текущую позицию.

Операции

Конструктор <Тот же, что и в ADT для класса SeqList>

ListSize <Тот же, что и в ADT для класса SeqList>

ListEmpty <Тот же, что и в ADT для класса SeqList>

ClearList <Тот же, что и в ADT для класса SeqList>

Find

Вход: Ссылка на значение данных

Предусловия: Нет

Процесс: Осуществить поиск на дереве путем сравнения элемента с данными, хранящимися в узле. Если происходит совпадение, выбрать данные из узла.

Выход: Возвратить 1 (True), если произошло совпадение, и присвоить данные из совпавшего узла параметру. В противном случае возвратить 0 (False).

Постусловия: Текущая позиция соответствует совпавшему узлу.

Insert

Вход: Элемент данных

Предусловия: Нет

Процесс: Найти подходящее для вставки место на дереве. Добавить новый элемент данных.

Выход: Нет

Постусловия: Текущая позиция соответствует новому узлу.

Delete

Вход: Элемент данных

Предусловия: Нет

Процесс: Найти на дереве первый попавшийся узел, содержащий элемент данных. Удалить этот узел и связать все его поддеревья так, чтобы сохранить структуру бинарного дерева поиска.

Выход: Нет

Постусловия: Текущая позиция соответствует узлу, заменившему удаленный.

Update

Вход: Элемент данных

Предусловия: Нет

Процесс: Если ключ в текущей позиции совпадает с ключом элемента данных, присвоить элемент данных узлу. В противном случае вставить элемент данных в дерево.

Выход: Нет

Постусловия: В списке может оказаться новое значение.

GetRoot

Вход: Нет

Предусловия: Нет

Процесс: Получить указатель на корень.

Выход: Возвратить указатель на корень.

Постусловия: Не изменяется

Конец ADT для бинарных поисковых деревьев

Объявление класса BinSTree. Мы реализовали АДТ для бинарных поисковых деревьев в виде класса с динамическими списковыми структурами. Этот класс содержит стандартный деструктор, конструктор копирования и перегруженные операторы присваивания, позволяющие инициализировать объекты и играющие роль операторов присваивания. Деструктор отвечает за очистку списка, когда закрывается область действия объекта. Деструктор и операторы присваивания вместе с методом ClearList вызывают закрытый метод DeleteTree. Мы также включили сюда закрытый метод CopyTree для использования в конструкторе копирования и перегруженном операторе.

Спецификация класса BinSTree

ОБЪЯВЛЕНИЕ

```
#include <iostream.h>
#include <stdlib.h>

#include "treenode.h"
```

```
template <class T>
class BinSTree
{
  protected: // требуется для наследования в гл. 12
    // указатели на корень и на текущий узел
    TreeNode<T> *root;
    TreeNode<T> *current;

    // число элементов дерева
    int size;

    // распределение/освобождение памяти
    TreeNode<T> *GetTreeNode(const T& item,
                    TreeNode<T> *lptr, TreeNode<T> *rptr);
    void FreeTreeNode(TreeNode<T> *p);

    // используется конструктором копирования и оператором присваивания
    void DeleteTree(TreeNode<T> *t);

    // используется деструктором, оператором присваивания
    // и функцией ClearList
    TreeNode<T> *FindNode(const T& item,
                          TreeNode<T>* & parent) const;
  public:
    // конструктор и деструктор
    BinSTree(void);
    BinSTree(const BinSTree<T>& tree);
    ~BinSTree(void);

     // оператор присваивания
     BinSTree<T>& operator= (const BinSTree<T>& rhs);

     // стандартные методы обработки списков
     int Find(T& item);
     void Insert(const T& item);
     void Delete(const T& item);
     void ClearList(void);
     int ListEmpty(void) const;
     int ListSize(void) const;

     // методы, специфичные для деревьев
     void Update(const T& item);
     TreeNode<T> *GetRoot(void) const;

};
```

ОПИСАНИЕ

Этот класс имеет защищенные данные. Они предоставляют конструкцию наследования. Защищенный доступ функционально эквивалентен закрытому доступу для данного класса. Переменная root указывает на корневой узел дерева. Указатель current ссылается на точку последнего изменения в списке. Например, current указывает положение нового узла после операции включения, а метод Find заносит в current ссылку на узел, совпавший с элементом данных.

Стандартные операции обработки списков используют те же имена и параметры, что и определенные в классе SeqList.

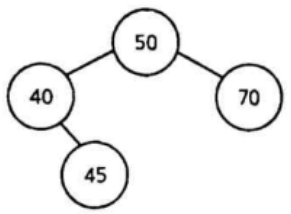
Класс BinSTree содержит две операции, специфические для деревьев. Метод Update присваивает новый элемент данных текущему узлу или включает в дерево новый элемент, если тот не совпадает с данными в текущей позиции. Метод GetRoot предоставляет доступ к корню дерева. Имея корень дерева, пользователь получает доступ к библиотечным функциям из treelib.h, treescan.h и treeprint.h. Это расширяет возможности класса для привлечения различных алгоритмов обработки деревьев, в том числе распечатки дерева.

ПРИМЕР

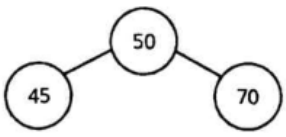
```
BinSTree<int> T;        // дерево с целочисленными данными

T.Insert(50);           // создать дерево с четырьмя узлами (A)
T.Insert(40);
T.Insert(70);
T.Insert(45);

T.Delete(40);           // удалить узел 40 (B)
T.ClearList();          // удалить узлы дерева
```



(A)



(B)

```
// дерево univInfo содержит информацию о студентах.
// Поле ssn является ключевым
BinSTree<Student> univInfo;
Student stud;

// назначить ключ "9876543789" и найти его на дереве
stud.ssn = "9876543789";
if (univInfo.Find(stud))
{
  // студент найден. присвоить новый средний балл и обновить узел
  stud.gpa = 3.86;
  univInfo.Update(stud);
}
else
  cout << "Студент отсутствует в базе данных." << endl;
```

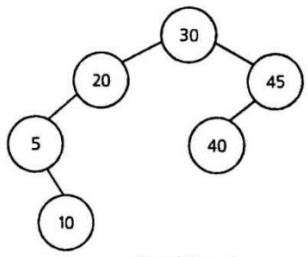
Использование бинарных деревьев поиска

Класс BinStree – мощная структура данных, которая используется для обработки динамических списков. Практическая задача построения конкодерданка (алфавитный список всех слов заданного текста с указателями на места их появлений) иллюстрирует типичное применение поисковых деревьев.

Создание примеров деревьев поиска. Ранее функция MakeCharTree использовалась для создания ряда бинарных деревьев с символьными данными. Похожая функция MakeSearchTree строит бинарные деревья поиска с целочисленными данными, применяя метод Insert. Например, дерево SearchTree_0 использует шесть элементов заранее определенного массива arr0 чтобы сконструировать дерево с помощью объекта T класса BinStree.

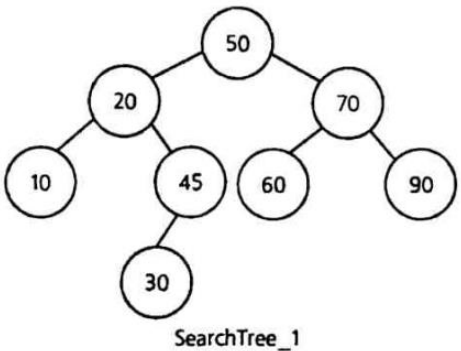
```
int arr0[6] = {30, 20, 45, 5, 10, 40};

for (i = 0; i < 6; i++)
  T.Insert(arr0[i]);          // добавить элемент к дереву
```

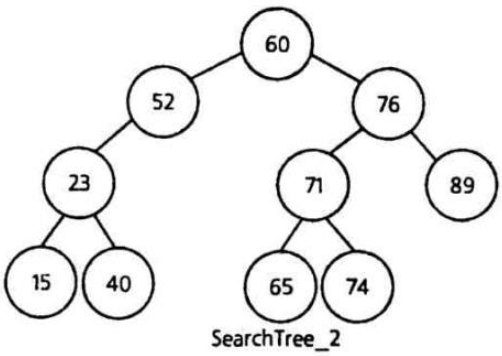


SearchTree_0

MakeSearchTree создает второе восьмиэлементное дерево и дерево с десятью случайными числами из диапазона 10-99. Параметры функции содержат объект класса BinSTree и параметр type (0 <= type <= 2), служащий для обозначения дерева. Код MakeSearchTree находится в файле makestch.h.



SearchTree_1



SearchTree_2

Симметричный метод прохождения. При симметричном прохождении бинарного дерева сначала посещается левое поддерево узла, затем сам узел, потом правое поддерево. Когда этот метод прохождения применяется к бинарному дереву поиска, узлы посещаются в сортированном порядке. Этот факт становится очевидным, когда вы сравниваете узлы в поддеревья текущего узла. Все узлы левого поддерева текущего узла имеют меньшие значения, чем текущий узел, и все узлы правого поддерева текущего узла больше или равны текущему узлу. Симметричное прохождение бинарного дерева гарантирует, что для каждого узла, который мы посещаем впервые, меньшие узлы находятся в левом поддереве, а большие – в правом. В результате узлы проходятся в возрастающем порядке.

Эта программа использует функцию MakeSearchTree для создания дерева SearchTree_1, содержащего числа 50, 20, 45, 70, 10, 60, 90, 30.

С помощью метода GetRoot мы получаем доступ к корню этого дерева, что позволяет вызвать функцию PrintVTree. Метод GetRoot позволяет также распечатать элементы по возрастанию, используя функцию Inorder с параметром-функцией PrintInt. Программа заканчивается удаление элементов 50 и 70 и повторной печатью дерева.

```
#include "makesrch.h"            // функция MakeSearch
#include "treescan.h"
#include "treeprnt.h"            // функция PrintVTree
#include "bstree.h"              // функция Inorder

// печать целого числа. используется функцией Inorder
void PrintInt(int& item)
{
  cout << item << "  ";
}

void main(void)
{
   // объявить целочисленное дерево
   BinSTree<int> Tree;

   // создать дерево поиска #1 и распечатать его вертикально
   // при ширине в 40 символов
   MakeSearchTree(Tree, 1);
   PrintVTree(Tree.GetRoot(), 2, 40);

   // симметричное прохождение обеспечивает
   // посещение узлов по возрастанию
   // хранящихся в них чисел
   cout << endl << endl   << "Сортированный список: ";
   Inorder(Tree.GetRoot(),   PrintInt);
   cout << endl;

   cout << endl << "Удаление   узлов 70 и 50." << endl;
   Tree.Delete(70);
   Tree.Delete(50);
   PrintVTree(Tree.GetRoot(), 2, 40);
   cout << endl;
}
/*
```

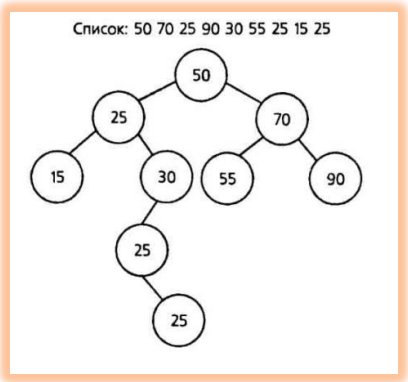
```
<Выполнение программы 11.4>
                50
        20                  70
    10      45          60      90
            30
Сортированный список: 10  20  30  45  50  60  70  90
Удаление узлов 70 и 50.
```

```
                45
        20                  60
    10      30                  90
*/
```

Дублированные узлы

Бинарное дерево поиска может иметь дублированные узлы. В операции включения мы продолжаем сканировать правое поддерево, если наш новый элемент совпадает с данными с текущем узле. В результате в правом поддереве совпавшего узла возникают дублированные узлы. Например, следующее дерево генерируется из списка 50 70 25 90 30 55 25 15 25.

Многие приложения не допускают дублирования узлов, а используют в данных поле счетчика экземпляров элемента. Это – принцип конкорданса, когда отслеживаются номера строк, в которых встречается некоторое слово. Вместо того чтобы несколько раз размещать слово на дереве, мы обрабатываем повторные случаи употребления этого слова путем помещения номеров строк в список. Следующая программа иллюстрирует лобовой подход, когда счетчик дубликатов хранится как отдельный элемент даннных.



Программа. Счетчики появлений.

Запись IntegerCount содержит целую переменную number и поле count, которое используется для запоминания частоты появлений целого числа в списке. Поле number работает в качестве ключа в перегруженных операторах < и ==, позволяющих сравнить две записи IntegerCount. Эти операторы используются в функциях Find и Insert.

Программа генерирует 100000 случайных чисел в диапазоне 0-9 и связывает каждое число с записью IntegerCount. Метод Find сначала определяет, есть ли уже данное число на дереве. Если есть, то значение поля count увеличивается на единицу и мы обновляем запись. В противном случае новая запись включается в дерево. Программа завершается симметричным прохождением узлов, в процессе которого происходит печать чисел и их счетчиков. Все генерируемые случайным образом числа от 0 до 9 равновероятны. Следовательно, каждый элемент может появиться приблизительно 10000 раз. Запись IntegerCount и два ее оператора находятся в файле intcount.h.

```
#include <iostream.h>

#include "random.h"     // генератор случайных чисел
#include "bstree.h"     // класс BinSTree
#include "treescan.h"   // функция Inorder
#include "intcount.h"   // запись IntegerCount

// вызывается функцией Inorder для распечатки записи IntegerCount
void PrintNumber(IntegerCount& N)
{
  cout << N.number << ':' << N.count << endl;
}

void main(void)
{
  // объявить дерево, состоящее из записей IntegerCount
  BinSTree<IntegerCount> Tree;

  // сгенерировать 100000 случайный целых чисел в диапазоне 0..9
  for (n = 0; n < 100000L; n++);
  {
    // сгенерировать запись IntegerCount со случайным ключом
    N.number = rnd.Random(10);

    // искать ключ на дереве
    if (Tree.Find(N))
    {
      // ключ найден, увеличить count и обновить запись
      N.count++;
      Tree.Update(N);
    }
    else
    {
      // это число встретилось впервые. вставить его с count=1
      N.count = 1;
      Tree.Insert(N);
    }
  }

  // симметричной прохождение для распечатки ключей по возрастанию
  Inorder(Tree.GetRoot(), PrintNumber);
}
/*
```

```
/*
<Выполнение программы 11.5>

0:10116
1:9835
2:9826
3:10028
4:10015
5:9975
6:9983
7:10112
8:10082
9:10028
*/
```

Источники:

1. У. Топп, У.Форд – Структуры данных в с++
2. https://habr.com/ru/post/267855/