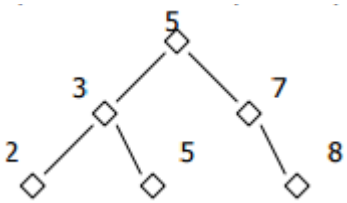
Задание №5.
Самостоятельная работа
«Реализация операций над упорядоченными бинарными деревьями (BST-деревья)»

Упорядоченные бинарные деревья называют деревьями поиска – *binary search trees* – BST-дерево.



Это дерево, которое может иметь (или не иметь) две вершины – левую, правую. Значения в вершинах отвечают свойству упорядоченности. Определим его так: Пусть x – произвольная вершина двоичного дерева поиска. Если вершина *y* находится в левом поддереве вершины *x*, то $y < x$:

$$y \rightarrow d <= x \rightarrow d.$$

Если в правом, то

$$y \rightarrow d >= x \rightarrow d.$$

Структура узла при реализации на базе списка:

```
struct Node {
  public:
  int d;
  Node* father;
  Node* left;
  Node* right;
  Node(){father=left=right=0;};
};
Node* root=0; //корень дерева
```

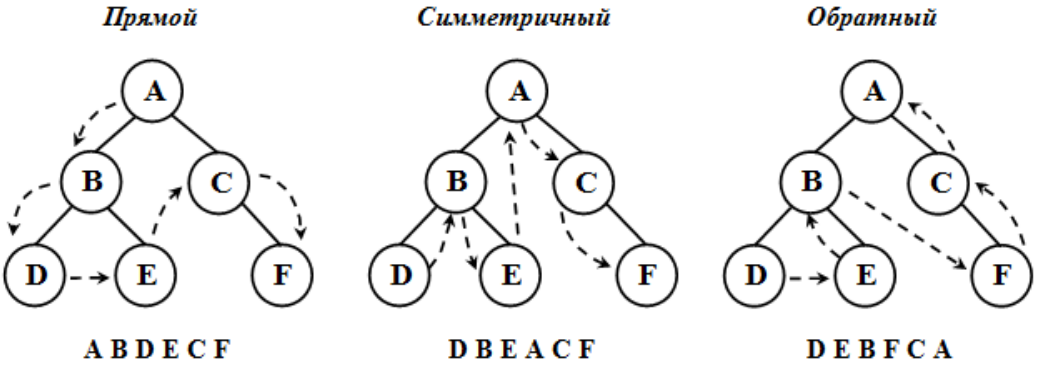
Деревья поиска позволяют выполнить следующие операции над множеством, которое они представляют:

- обход и печать всех значений дерева;
- поиск (search);
- максимум, минимум;
- получение следующего и предыдущего по значению;
- вставка;
- удаление;
- удаление всего дерева (destroy).

Свойство упорядоченности позволяет напечатать все значения в неубывающем порядке, т.е. отсортировать значения.

Операции обхода могут быть трех видов:

1. прямой (preorder) – порядок обхода, при котором корень предшествует обоим поддеревьям.
2. обратный (postorder) – порядок обхода, при котором корень следует за поддеревьями.
3. симметричный (inorder) - сначала самая левая вершина, корень, правая.



Обход реализуют рекурсивными алгоритмами:

Прямой обход:

```
void preorder_Tree (Node * p)
{
  if (p)
  { printf (“% i”, p→d);
    preorder_Tree (p→left);
    preorder_Tree (p→right);
  }
}
```

Обратный обход:

```
void postorder_Tree (Node * p)
{
  if (p)
  { postorder_Tree (p→left);
    postorder_Tree (p→right);
    printf (“% i”, p→d);
  }
}
```

Симметричный обход:

```
void inorder_Tree (Node * p)
{  // самостоятельно !!! }
```

Проверьте: какой из видов обхода позволит вывести отсортированную последовательность вершин.

Операция поиск Search.

Эту операцию тоже можно реализовать рекурсивно и итерационно.

Рекурсивно:

```
  //d – искомое значение
Node * Tree_Search (Node * p, int d)
{
  if (p = = 0 или p→d = = d) return p;
  if (d < p→d) Tree_Search (p→left, d);
    else Tree_Search (p→right, d);
}
```

Итеративная версия (более эффективна – быстрая):

```
Node * Iterative_Search_Tree (Node * p, int d)
{
  while (p!=0 и d!=p→d)
  if (d < p→d) p = p→left;
    else p =p→right;
  return p;
}
```

Операции поиска max и min значений.

Минимальное значение можно найти, пройдя по указателям *left*.

```
// x – root при вызове или корень поддерева
Node Tree_Minimum (Node* x)
{
  while (x→left != 0)
    x = x→left;
  return x;
}
```

Алгоритм для поиска *max* симметричен, но теперь уже надо идти по указателям *right*.

Следующий Tree_Succ и предыдущий Tree_Pred элемент по значению ключа.

Итераторы.

```
Node * Tree_Succ (Node *p)
{
  if (p→right) return Tree_Minimum (p→right);
  y = p→father;
  while (y != 0 и p == y→right)
  {
    p = y;
    y = y→father;
  }
  return y;
}
```

Два случая:

- если правое поддерево *p* – не пусто, то следующий за p элемент – минимальный в этом поддереве;
- если правое поддерево – пусто, тогда мы идём вверх от *p* пока не найдём вершину, от которой мы в последний раз пошли «влево».

Функция Tree_Pred – симметрична.

Добавление элемента (Tree_Insert)

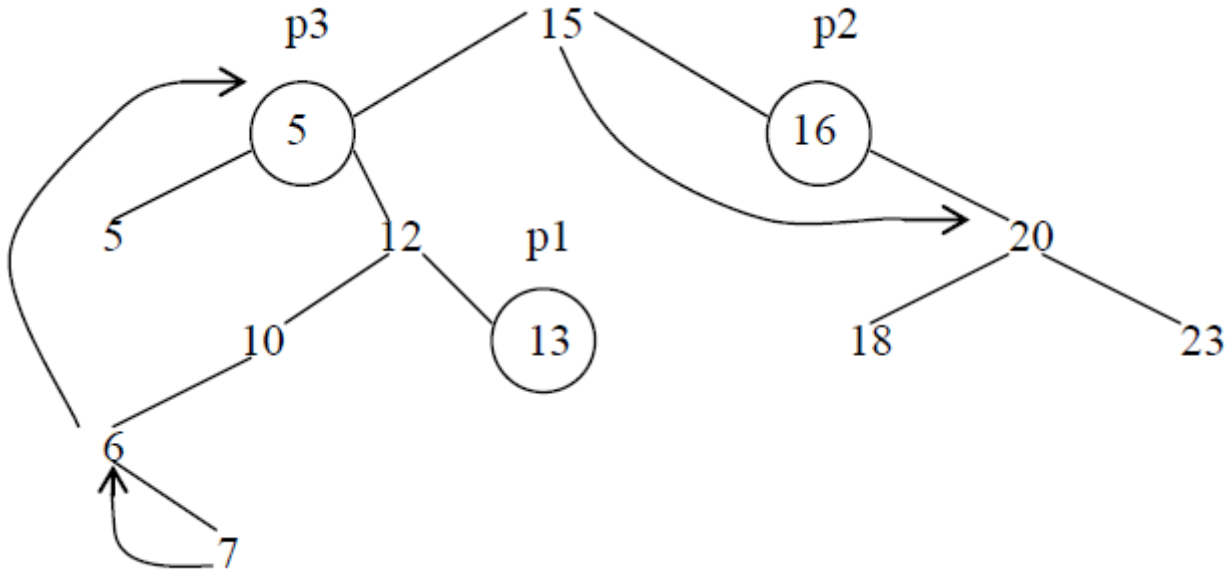
Вставляет элемент в подходящее место, сохраняя свойство упорядоченности.

```
void Tree_Insert (int d)
{
  Node * z = new Node;
  z→d = d;
  if (root == 0)
  {
    root = z;
    return;
  }
  Node *y, *x = root;
  while (x)
  {
    y =x;
    if (z→d < x→d) x = x→left;
      else x = x→right;
  }
  z→father = y;
  if (y→d > z→d) y→left = z;
    else y→right = z;
}
```

Удаление элемента (Tree_Delete)

Рассматривается три случая p1, p2, p3.

Разберите самостоятельно и проверьте её на корректность!




```
void Tree_Delete (Node * p)
{
  Node * x, *y;
  if (root && p)
  {
    if ( p→left == 0 || p→right == 0) y = p;
      else y = Tree_Succ(p);
    if ( y→left) x = y→left;  //p1
      else x = y→right;
    if ( x ) x→father = y→father;  //p2
    if (y→father == 0)  root = x;  //если корень
      else {
        if (y == y→father→left)  y→father→left = x;
        //если он - левая вершина отца
          else y→father→right = x; }
    if (y != p) p→d = y→d;  //p3
    delete y;
  }
}
```

Удаление дерева (Tree_Erase)

Алгоритм удаления дерева использует обратный обход, так как при обратном обходе сначала посещаются наследники, а потом родительский узел. Это дает возможность удалить сначала дочерние вершины, а затем корень. При этом для удаления очередной вершины используем операцию *Tree_Delete*.

Реализуйте самостоятельно.

Задания самостоятельной работы

1. Реализуйте все типовые операции над двоичным деревом поиска на базе динамического списка.
2. Реализуйте все типовые операции над двоичным деревом на базе одномерного массива.

Дополнительные задания

1. Найдите количество четных элементов *бинарного дерева*. Укажите эти элементы и их уровни.
2. Найдите сумму элементов *сбалансированного дерева*, находящихся на уровне `k`.
3. Оператор мобильной связи организовал базу данных *абонентов*, содержащую сведения о телефонах, их владельцах и используемых тарифах, в виде *бинарного дерева*. Составьте программу, которая:
   - обеспечивает начальное формирование базы данных в виде *бинарного дерева*;
   - производит вывод всей базы данных;
   - производит поиск владельца по номеру телефона;
   - выводит наиболее востребованный тариф (по наибольшему числу *абонентов*).

Литература

1. Альфред Ахо, Джон Э. Хопкрофт, Д. Ульман Структуры данных и алгоритмы. - М. - СПб - Киев: "Вильямс", 2000 г. - 384 с.
2. Н. Вирт Алгоритмы + структуры данных = программы. - М.: "Мир", 1985 г. - 406 с.
3. Фрэнк М. Каррано, Джанет Дж. Причард. Абстракция данных и решение задач на C++. Стены и зеркала. - М. - СПб - Киев: "Вильямс", 2003 г. - 848 с.
4. Д. Кнут. Искусство программирования для ЭВМ. Т.1. Основные алгоритмы. - М: "Мир", 1976 г. (переиздание - М., Изд. "Вильямс", 2000 г.) - 735 с.
5. Д. Кнут. Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. - М: "Мир", 1978 г. (переиздание - М., Изд. Изд. Вильямс", 2000 г.) - 844 с.
6. Т. Кормен, Ч. Лейзерсон, Р. Ривест Алгоритмы. Анализ и построение. - М: "БИНОМ", 2000 г. - 960 с
7. Дж. Макконелл. Анализ алгоритмов. Вводный курс. - М: "Техносфера", 2002 г. - 304 с.