Лабораторная работа №6 по дисциплине «Типы и структуры данных»

Обработка деревьев

Цель работы:

Получить навыки применения двоичных деревьев, реализовать основные операции над деревьями: обход деревьев, включение, исключение и поиск узлов.

Задание (Вариант 1):

Построить дерево в соответствии со своим вариантом задания. Вывести его на экран в виде дерева. Реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов. Сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления.

В текстовом файле содержатся целые числа. Построить двоичное дерево из чисел файла. Вывести его на экран в виде дерева. Используя подпрограмму, определить количество узлов дерева на каждом уровне. Добавить число в дерево и в файл. Сравнить время добавление в указанные структуры.

Исходные данные:

Программа предназначена для работы с текстовым файлом. Файл может содержать произвольное количество целых чисел. Также, слова в дерево можно добавлять через меню.

Тесты

Если дерево не содержит удаляемого числа будет выведено предупреждение об отсутствии числа в дереве.

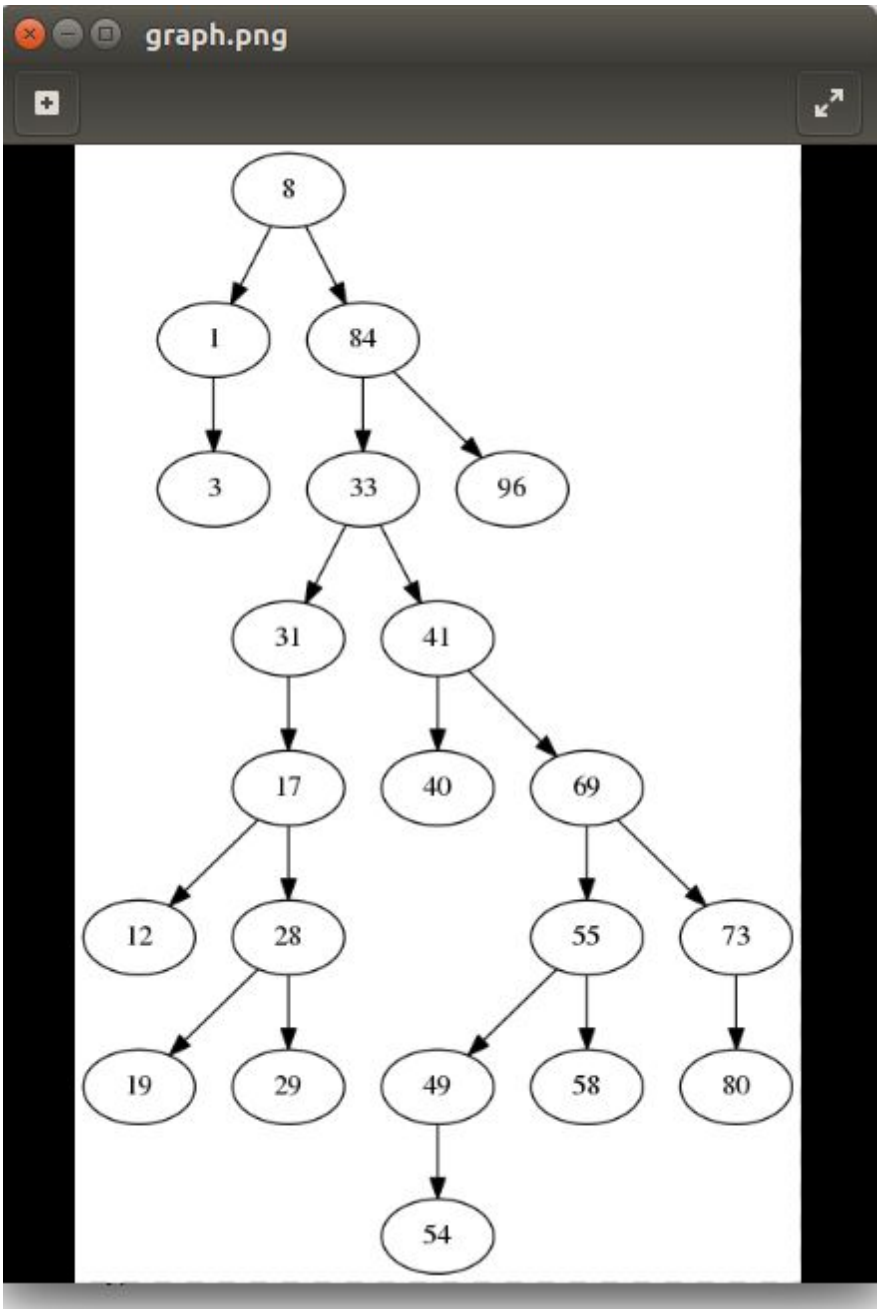
В случае некорректного выбора меню, будет выведено предупреждение **"Error command"**

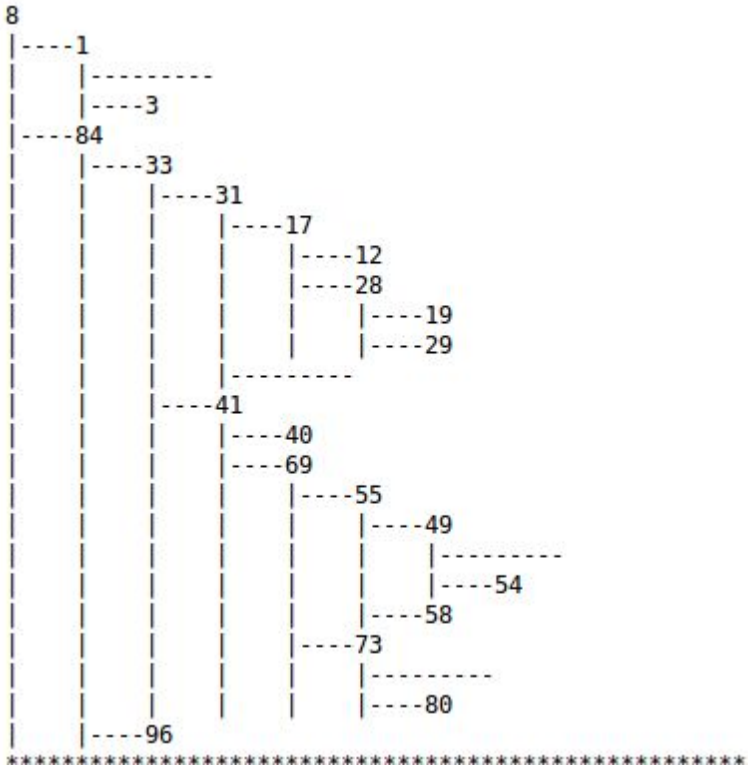
Интерфейс программы:

1)Главное меню

```
Выберите одно из действий:
1: Отобразить дерево (graphviz)
2: Отобразить дерево (tree)
3: Добавить число в дерево
4: Удалить число из дерева
5: Поиск числа
6: Сравнение времени добавления числа
7: Вывести количество элеметов на уровне
8: Обход дерева
9: Закончить работу
```

2) Отображение дерева


graph.png
8
1
84
3
33
96
31
41
17
40
69
12
28
55
73
19
29
49
58
80
54



Внутренние структуры данных:

Структура элемента дерева

```
template<typename T>
struct element{
  T value;
  element<T>*left = NULL;
  element<T>*right = NULL;

  element(T x) {
     value = x;
     left = NULL;
     right = NULL;
  }

};
```

Структура дерева

```
template<typename T>
class BT {
private:
  element<T> *head = NULL;
  void operator_copy(element<T> **head, element<T> *tmp);
  void LRootR(element<T> *tmp);
  void LRRoot(element<T> *tmp);
  void RootLR(element<T> *tmp);
  void delete_tree(element<T> *tmp);
  void remove(element<T> *prev, element<T> *tmp);
```

```
    void count_levels(element<T>*tmp, int deep,vector<int>& vector1);
    void print(element<T>* tmp, int deep, bool flag);
    void printer(element<T>*tmp, element<T>*parent, FILE *graph);
  public:
    BT(const BT &obj);
    BT();
    BT &operator=(const BT &obj);
    void levels() ;
    void insert(T x);
    void printer();
    bool remove(T x);
    ~BT() ;
    void show_as_tree();
    bool find(T x);
    void LeftRootRight() ;
    void LeftRightRoot();
    void RootLeftRight();
  };
```

Алгоритм функции подсчета количества элементов на уровне

```
  void BT<T>::count_levels(element<T>*tmp, int deep,vector<int>& vector1) {

    if(tmp) {
       if(vector1.size()<=deep)
       {
          vector1.push_back(0);
       }
       vector1[deep]+=1;
       count_levels(tmp->left, deep + 1, vector1);
       count_levels(tmp->right, deep + 1, vector1);
    }
  }
```

При удаление вершины на ее место становится лист дерева с максимального из левого поддерева.

Тесты

Операция	**Исходное дерево (префиксный обход)**	**Получаемо дерево(префиксный обход)**
Удаление 8	8 1 3 12 17 19 28 29 31 33 40 41 49 54 55 58 69 73 80 84 96	12 1 3 17 19 28 29 31 33 40 41 49 54 55 58 69 73 80 84 96
Удаление 100	12 1 3 17 19 28 29 31 33 40 41 49 54 55 58 69 73 80 84 96	Дерево не содержит числа
Удаление 33	12 1 3 17 19 28 29 31 33 40 41 49 54 55 58 69 73 80 84 96	12 1 3 17 19 28 29 31 40 41 49 54 55 58 69 73 80 84 96
Поиск 100	12 1 3 17 19 28 29 31 33 40 41 49 54 55 58 69 73 80 84 96	Дерево не содержит числа
Поиск 31	12 1 3 17 19 28 29 31 33 40 41 49 54 55 58 69 73 80 84 96	Число есть в дереве
Удалить 1	1	(Пусто)

Удалить 4	4 3 5	5 3

Сравнение времени добавления числа в дерево и файл

Добавление числа в файл имеет линейную сложность, а добавление элемента в дерево в худшем случае nlog(n), где n - количество чисел

№	Количество элементов	Время добавления в дерево	Время добавления в файл	повторное добавление в файл
1	**100**	**4**	**76**	**49**
2	**200**	**3**	**32**	**25**
3	**300**	**2**	**76**	**40**
4	**400**	**1**	**70**	**27**
5	**500**	**3**	**69**	**70**
6	**600**	**3**	**40**	**31**
7	**700**	**3**	**52**	**27**
8	**800**	**4**	**41**	**68**
9	**900**	**3**	**39**	**39**
10	**1000**	**5**	**42**	**48**

Операционная система производит кэширование файла в оперативной памяти ПК, поэтому повторное открытие и дозапись происходят намного быстрее чем в первый раз.

При модификации элементов данных в кэше выполняется их обновление в основной памяти. Задержка во времени между модификацией данных в кэше и обновлением основной памяти управляется так называемой политикой записи.
В кэше с немедленной записью каждое изменение вызывает синхронное обновление данных в основной памяти. В кэше с отложенной записью (или обратной записью) обновление происходит в случае вытеснения элемента данных, периодически или по запросу клиента. Для отслеживания модифицированных элементов данных записи кэша хранят признак модификации (изменённый или «грязный»). Промах в кэше с отложенной записью может потребовать два обращения к основной памяти: первое для записи заменяемых данных из кэша, второе для чтения необходимого элемента

данных. Поэтому измерить точное время при добавление в файл не представляется возможным, так как ОС сама решает в какой момент ей выгрузить данные на диск.

Вывод:

Основным преимуществом двоичного дерева перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки. Хранение и обработка дерева требует аккуратного обращение с памятью.

Контрольные вопросы

1. Что такое дерево?

Дерево – нелинейная структура данных, которая используется для представления иерархических связей «один ко многим». Дерево с базовым типом T определяется рекурсивно: это либо пустая структура (пустое дерево), либо узел типа T с конечным числом древовидных структур того же типа – поддеревьев.

2. Как выделяется память под представление деревьев?

Выделение памяти под деревья определяется типом их представления. Это может быть таблица связей с предками (№ вершины - № родителя), или связный список сыновей. Оба представления можно реализовать как с помощью матрицы, так и с помощью списков. При динамическом представлении деревьев (когда элементы можно удалять и добавлять) целесообразнее использовать списки – т.е. выделять память под каждый элемент динамически.

3. Какие бывают типы деревьев?

АВЛ-деревья, сбалансированные деревья, двоичные, двоичного поиска.

4. Какие стандартные операции возможны над деревьями?

Основные операции с деревьями: обход (инфиксный, префиксный, постфиксный), поиск элемента по дереву, включение и исключение элемента из дерева.

5. Что такое дерево двоичного поиска?

Дерево двоичного поиска – дерево, в котором все левые потомки «моложе» предка, а все правые – «старше». Это свойство выполняется для любого узла, включая корень.