**Обработка деревьев**

**Цель работы** – получить навыки применения двоичных деревьев, реализовать основные операции над деревьями: обход деревьев, включение, исключение и поиск узлов.

**Входные данные:**

Строка букв, длиною не более 279.

**Выходные данные:**

Двоичное дерево поиска, составленное из букв данной строки, представленное в графическом виде; узлы, значения которых встречается в данной строке более одного раза, выделены цветом. Двоичное дерево поиска без повторяющихся букв строки.

**Особенность работы программы**

Для деревьев, состоящих только из корня, отсутствует графическая реализация.

**Внутренние структуры данных**

**struct** tree\_node

{

**char** **name**; -ключ

**struct** tree\_node \***left**; - указатель на левый потомок

**struct** tree\_node \***right**; -указатель на правый потомок

};

**Основные алгоритмы**

**Добавление элемента (рекурсивная реализация)**

**struct** tree\_node\* insert(**struct** tree\_node \*tree, **struct** tree\_node \*node)

{

**int** cmp;

**if** (tree == **NULL**)

**return** node;

**if** ( node->**name**> tree->**name**) cmp=1 ; **else** cmp=-1;

**if** (cmp == 0)

{

**assert**(1);

}

**else** **if** (cmp < 0)

{

tree->**left** = insert(tree->**left**, node);

}

**else**

{

tree->**right** = insert(tree->**right**, node);

}

**return** tree;

}

**Поиск элемента(рекурсивная реализация)**

**struct** tree\_node\* lookup\_1(**struct** tree\_node \*tree,**char** name)

{

**int** cmp;

**if** (tree == **NULL**)

**return** **NULL**;

**if** (name==tree->**name**) {cmp=0;}

**else**

**if** ( name> tree->**name**) cmp=1 ;

**else** cmp=-1;

**if** (cmp == 0)

**return** tree;

**else** **if** (cmp < 0)

**return** lookup\_1(tree->**left**, name);

**else**

**return** lookup\_1(tree->**right**, name);

}

**Обход дерева(постфиксный)**

**void** apply\_post(**struct** tree\_node \*tree, **void** (\*f)(**struct** tree\_node\*, **void**\*), **void** \*arg)

{

**if** (tree == **NULL**)

**return**;

apply\_post(tree->**left**, f, arg);

apply\_post(tree->**right**, f, arg);

f(tree, arg);

}

**Удаление элемента(рекурсивная реализация)**

**struct** tree\_node \*dtree(**struct** tree\_node \*root, **char** key)

{

**struct** tree\_node \*p,\*p2;

**if**(!root) **return** root; */\** *вершина* *не* *найдена* *\*/*

**if**(root->**name** == key) { */\** *удаление* *корня* *\*/*

*/\** *это* *означает* *пустое* *дерево* *\*/*

**if**(root->**left** == root->**right**){

free(root);

**return** **NULL**;

}

*/\** *или* *если* *одно* *из* *поддеревьев* *пустое* *\*/*

**else** **if**(root->**left** == **NULL**) {

p = root->**right**;

free(root);

**return** p;

}

**else** **if**(root->**right** == **NULL**) {

p = root->**left**;

free(root);

**return** p;

}

*/\** *или* *есть* *оба* *поддерева* *\*/*

**else** {

p2 = root->**right**;

p = root->**right**;

**while**(p->**left**) p = p->**left**;

p->**left** = root->**left**;

free(root);

**return** p2;

}

}

**if**(root->**name** < key) root->**right** = dtree(root->**right**, key);

**else** root->**left** = dtree(root->**left**, key);

**return** root;

}

**Эффективность поиска в зависимости от высоты и степени ветвления дерева**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество элементов** | **Вид** | **Затраченное время(секунд)** | **Глубина**  **искомого элемента** | **Высота дерева** |
| 26 | Правостороннее | 1.854828e-003 | 25 | 25 |
| 26 | Правостороннее | 1.635172e-003 | 20 | 25 |
| 26 | Правостороннее | 1.234828e-003 | 15 | 25 |
| 26 | Правостороннее | 6.200000e-004 | 5 | 25 |
| 26 | Левостороннее | 2.210000e-003 | 25 | 25 |
| 26 | Левостороннее | 1.614828e-003 | 20 | 25 |
| 26 | Левостороннее | 1.540000e-003 | 15 | 25 |
| 26 | Левостороннее | 1.024828e-003 | 5 | 25 |
| 26 | Сильноветвящееся | 1.464828e-003 | 9 | 9 |
| 26 | Сильноветвящееся | 9.348276e-004 | 4 | 9 |
| 18 | Сильноветвящееся | 1.145172e-003 | 6 | 6 |
| 18 | Сильноветвящееся | 9.648276e-004 | 3 | 6 |

**Эффективность сортировки в зависимости от высоты и степени ветвления(tree sort)**

|  |  |  |
| --- | --- | --- |
| **Количество элементов** | **Вид** | **Затраченное время(секунд)** |
| 20 | Правостороннее | 1.864448e-001 |
| 20 | Левостороннее | 1.881752e-001 |
| 20 | Сильноветвящееся | 1.915300e-001 |
| 10 | Правостороннее | 1.395517e-002 |
| 10 | Левостороннее | 1.412000e-002 |
| 10 | Сильноветвящееся | 1.485000e-002 |

**Сравнение времени удаления элементов из строки и из дерева**

|  |  |  |  |
| --- | --- | --- | --- |
| Количество повторяющихся элементов | Глубина элемента в дерева | Время удаления из строки(секунд) | Время удаления из графа(секунд) |
| 2 | 0 | 3.015172e-003 | 2.914828e-003 |
| 5 | 1 | 3.360000e-003 | 3.045172e-003 |
| 10 | 2 | 4.785172e-003 | 3.210000e-003 |
| 20 | 4 | 6.789655e-003 | 2.803448e-003 |

**Вывод:** Время удаления элемента из строки всегда больше времени удаления из графа, за счет того, что при удаления элемента из строки, происходит сдвиг массива символов, а так же за счет того, что если символ n раз встречается в строке, то из строки удалится n элементов, а из графа удалится всего лишь один элемент. Выигрыш по времени для большого количества повторяющихся элементов тем значительней, чем больше количество повторяющихся элементов.

**Вывод(общий):** Время, затрачиваемое на поиск в бинарном дереве, не зависит от количества элементов и степени ветвления дерева, а зависит только от глубины элемента дерева.

Деревья имеет смысл применять для поиска элемента.

Время, затраченное на сортировку, при помощи дерева зависит только от количества элементов дерева.