**Сбалансированные деревья, хеш–таблицы**

**Цель работы** – построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах. Сравнить эффективность устранения коллизий при внешнем и внутреннем хешировании

**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, балансированное двоичное дерево без без повторяющихся букв строки .

Хэш таблицы, коллизии в которых разрешены при помощи метода цепочек и метода внутреннего хэширования.

**Особенность работы программы**

Для деревьев, состоящих только из корня, отсутствует графическая реализация. Под хэш таблицу с открытой адресации выделена статическая область памяти.

**Внутренние структуры данных**

Двоичное дерево поиска

**struct** tree\_node

{

**char** **name**; -ключ

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

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

};

Сбалансированное дерево

**struct** Node

{

**char** **key**;

**char** **height**; - высота элемента

**struct** Node \***right**;

**struct** Node \***left**;

};

Хэш таблицы

**typedef** **struct** T1

{

**char** **data**; - ключ

**int** **number**; - число повторений буквы в строке

} T

**typedef** **struct** Node\_ {

**struct** Node\_ \***next**;

T **key**;

} Node1;

**Основные алгоритмы**

**Добавление элемента в хэш-таблицу с открытой адресацией(ОА)**

**void** put(**char** letter,**int** number, Node1 z[], **int** size)

{

**int** i=hash1(letter);

**while**((z[i].**key**.**data**!=0)&&(i<size)) i++;

**if** (i==size)

{

i=0;

**while**(z[i].**key**.**data**!=0) i++;

}

z[i].**key**.**data**=letter;

z[i].**key**.**number**=number;

**return**;

}

**Поиск элемента в ОА хэш-таблице**

Node1 search\_op(**char** letter, Node1 z[], **int** size)

{

Node1 \*p;

**int** i=hash1(letter);

**while**((z[i].**key**.**data**!=letter)&&(i<size))

{

i++;

hash\_op++;

}

**if**(z[i].**key**.**data**==letter) hash\_op++;

**if** (i==size)

{

i=0;

**while**(z[i].**key**.**data**!=letter)

i++;

}

**return** z[i];

}

**Добавление элемента в хэш-таблицу МП(метод цепочек)**

Node1 \*insertNode(**char** data1,**int** number) {

Node1 \*p, \*p0;

hashTableIndex bucket;

bucket = hash(data1);

**if** ((p = malloc(**sizeof**(Node1))) == 0) {

fprintf (**stderr**, **"out** **of** **memory** **(insertNode)\n"**);

exit(1);

}

p0 = hashTable[bucket];

hashTable[bucket] = p;

p->**next** = p0;

p->**key**.**data** = data1;

p->**key**.**number**=number;

**return** p;

}

**Поиск элемента в МП хэш-таблице**

Node1 \*findNode (**char** data) {

Node1 \*p;

p = hashTable[hash(data)];

**while** (p && !**compEQ**(p->**key**.**data**, data))

{

hash\_open++;

p = p->**next**;

}

**if** (p) hash\_open++;

**return** p;

}

**Сравнение эффективности поиска**

В двоичном дереве поиска после удаления повторяющих элементов не может содержаться более 26 элементов.

Двоичное дерево поиска

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Память | Время поиска(cекунд) | Количество сравнений | Искомый элемент |
| 26 | 234 bytes | 2.913793e-004 | 1 | a |
| 26 | 234 bytes | 3.503448e-004 | 2 | b |
| 26 | 234 bytes | 5.296552e-004 | 3 | c |
| 26 | 234 bytes | 5.862069e-004 | 4 | d |
| 26 | 234 bytes | 8.548276e-004 | 5 | e |
| 26 | 234 | 8.882759e-004 | 6 | f |
|  |  | 8.900000e-004 | 7 | g |
|  |  | 1.030000e-003 | 8 | h |
|  |  | 1.034828e-003 | 9 | I |
|  |  |  | 10 | J |
|  |  |  | 11 | k |
|  |  |  |  | l |
|  |  |  |  | m |
|  |  |  |  | n |
|  |  |  |  | o |
|  |  |  |  | p |
|  |  |  |  | q |
|  |  |  |  | r |
|  |  |  |  | s |
|  |  |  |  | t |
|  |  |  |  | u |
|  |  |  |  | v |
|  |  |  |  | w |
|  |  |  |  | x |
|  |  |  |  | y |
|  |  |  |  | z |