

Worksheet Hashing and Searching

Mike Fraser, Alex Stanhope

Week 9

In this worksheet you will be faced with problems to do with hashing, lists, and trees.

1. Suppose we want to store names, or words, consisting of lower case letters only. We will consider three hash functions, all hashing to a hashtable with 26 entries:

```
unsigned int hash1( char *s ) {  
    return s[0] - 'a' ;  
}
```

```
unsigned int hash2( char *s ) {  
    int h = 0, i ;  
    for( i=0 ; s[i] != '\0' ; i++ ) {  
        h = h + s[i] ;  
    }  
    return h % 26 ;  
}
```

```
unsigned int hash2( char *s ) {  
    int h = 0, i ;  
    for( i=0 ; s[i] != '\0' ; i++ ) {  
        h = h * 3 + s[i] ;  
    }  
    return h % 26 ;  
}
```

Suppose we wish to store the following words: *break*, *brake*, *dear*, *dare*, *bristol*, *bristle*, *not* and *ton*

- (a) How many collision will we have when using hash1?

Answer:

4 (brake, bristol and bristle collide with break, and dear with dare)

- (b) How many collision will we have when using hash2?

Answer:

3 (brake with break, hear with hare, and ton with not)

(c) How many collision will we have when using hash3 (guess, don't compute the answer)?

Answer:

0, there is a good chance that there is a collision already, birthday paradox.

(d) What is the strength of hash1?

Answer:

simple and fast!

(e) What is the strength of hash3?

Answer:

few collisions

(f) assume that we use hash function 1 and we use open addressing to resolve this. Sketch the contents of the hash table.

Answer:

slot 0 is empty;
slot 1 contains break;
slot 2 contains brake;
slot 3 contains dear;
slot 4 contains dare;
slot 5 contains bristol;
slot 6 contains bristle;
slot 7-12 are empty;
slot 13 contains not;
slot 14-18 are empty;
slot 19 contains ton;
slot 20-25 are empty;

(g) We will need a string comparison to compare the contents of a hash cell with a word when we are looking a word up. how many string comparisons are required to check whether the following words are in the hashtable: aardvark, chicken, bristol, peanuts, gorilla.

Answer:

aardvark: 0 (slot 0 is empty), chicken: 5 (compare with all values from slots 2-6 inclusive), bristol: 5 (compare with all values from slots 1-5 inclusive), peanuts: 0 (slot 16 is empty), gorilla: 0 (slot 7 is empty)

(h) Repeat the previous questions, assuming that we use direct chaining.

Answer:

slot 0 is empty;
slot 1 contains bristle, bristol, brake, break.
slot 2 is empty
slot 3 contains dare, dear.
slot 4-12 are empty;
slot 13 contains not;
slot 14-18 are empty;

slot 19 contains ton;
slot 20-25 are empty;
0, 0, 2, 0, 0

2. Assume we have an array storing the values 3, 9, 12, 13, 24, 25, 26, 200, 202, 208, 215, 300, 314, 31415, 80000. The lowest value is stored at index 0, the highest at index 14.

- (a) Suppose we want to check whether 14 is in the array or not. Which values does a binary chop need to compare with?

Answer:

200, 13, 25, 24

- (b) How many comparisons would you need if you did a linear search?

Answer:

Five: 3, 9, 12, 13, 24

- (c) What are the minimum and the maximum number of comparisons for the binary chop on this array?

Answer:

One and Four

- (d) What are the minimum and the maximum number of comparisons for a linear search on this array?

Answer:

One and Fifteen

- (e) Could you store this in a hash-table with, say, 17 elements? What would a suitable hash function be?

Answer:

Yeah, you could. modulo 17 is the simplest hash-function

3. Below is a definition of a tree, and a function that tests whether a value is in a tree or not. Note: the tree node has three members: a pointer to the tree with all bigger values, the contents of the node, and a pointer to the tree with all smaller values. They are stored in memory in that order. We assume that a pointer and an integer fit in one memory cell.

```
typedef struct Carol {
    struct Carol *bigger ;
    int contents ;
    struct Carol *smaller ;
} Tree ;

int is_in_tree( Tree *t, int value ) {
    if( t == NULL )          return 0 ;
    if( t->contents == value ) return 1 ;
    if( t->contents > value ) {
        return is_in_tree( t->bigger, value ) ;
    } else {
        return is_in_tree( t->smaller, value ) ;
    }
}
```

We have built a tree with the root at cell 35, as shown in the table with memory addresses below. A NULL pointer is represented with the number 0.

	0	1	2	3	4	5	6	7	8	9
10	0	108	0	0	0	0	0	0	45	0
20	0	0	46	240	0	41	118	10	0	12
30	0	5	0	0	0	38	8	30	22	200
40	25	0	119	0	0	0	0	255	0	0

So, given that the first node is at cell 35, we see that the three members of that cell are 38, 8 and 30. That is, the field `contents` is 8, the pointer to the `smaller` subtree is 30, and the pointer to the `bigger` subtree is 38. (following the order of the fields above).

- (a) Which nodes are visited when `is_in_tree(35, 22)` is called? Ie, we are searching for an element with value 22, the root of the tree is at 35.

Answer:

35, 38, 25, 10; the values that we inspect on the way are 8, 200, 118, 108.

- (b) How many values are stored in the tree? Which values?

Answer:

Eight: 255, 240, 200, 119, 118, 108, 8, 5.

- (c) If you haven't done so already, draw the tree on a piece of paper.

Answer:

