Foundations of Computer Science (Semester 2) – 2015

Assessed Exercise Sheet 8 – 10% of Continuous Assessment Mark

Deadline : 11pm Sunday 15th March, via Canvas

Question 1 (24 marks)

Suppose you have a hash table of size 13, the keys are words, and the hash function is defined as follows: Each letter is assigned a number according to its position in the alphabet, i.e.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

and the numbers corresponding to all the letters in the key word are added and their remainder after division by 13 is computed. [Hint: Rather than performing tedious computations like this by hand, it will probably be better to write a small computer program to do it for you.]

Insert the following list of words into an initially empty hash table using linear probing:
`[computer, science, in, birmingham, dates, back, to, the, sixties]`

0	1	2	3	4	5	6	7	8	9	10	11	12
back			sixties	the	dates	birmingham	to	in			science	computer

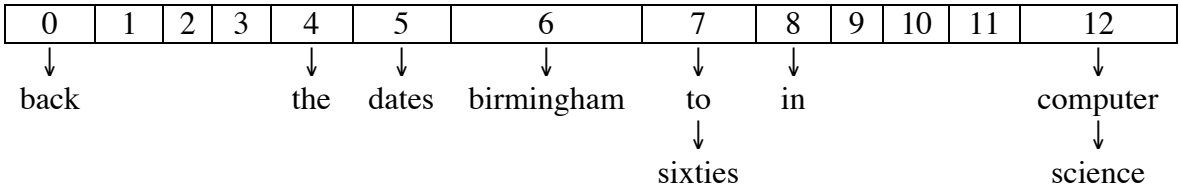
What is the load factor of the resulting table, and how many collisions occurred?

The load factor is 9/13 ~ 0.69 and 5 collisions occurred.

What is the effort (i.e. number of comparisons) involved in checking whether each of the following words are in the hash table: `teaching, research, admin`?

Checking for `teaching` takes 6 comparisons, `research` takes 3 comparisons, and `admin` takes 1 comparisons.

Show what the resulting hash table would look like if direct chaining had been used rather than linear probing.



Now what is the effort (i.e. number of comparisons, not including the processing of NULL pointers) involved in checking whether each of the following words are in the hash table: `teaching, research, admin`?

Checking for `teaching` takes 2 comparisons, `research` takes 1 comparisons, and `admin` takes 0 comparisons.

Question 2 (20 marks)

Suppose you have a hash table of size 19 to accommodate words. Each letter is assigned a number according to its position in the alphabet as in Question 1. The primary hash function is "*x* modulo 19", where *x* is the number corresponding to the first letter of the word.

Why is this hash function not ideal?

Taking just one letter as the basis of the hash function means that words with the same first letter will be mapped to the same position, and since the letters in words are far from equally distributed, that will result in many collisions and a clustered table. This will be made worse because several consecutive positions correspond to two letters, and the rest only to one.

Fill an initially empty hash table using linear probing with the following words:
`[all, human, beings, are, born, free, equal, in, dignity, and, rights]`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
all	beings		dignity	equal	free		human	in							rights	and	born	are

Next, instead of using linear probing, insert the same words into an initially empty hash table with the conflicts resolved using the secondary hash function "1 + *y* modulo 11", where *y* is the number corresponding to the last character in the word.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
all	beings		dignity	equal	free		human	in	rights					are	and		born	

Why is this particular secondary hash function not ideal?

As with the primary hash function, the distribution will be uneven.

What is the load factor of the table, and how many collisions occurred with each of the two approaches to filling it?

The load factor is 11/19 ~ 0.57. Linear probing resulted in 9 collisions, and secondary hashing resulted in 4 collisions.

Question 3 (18 marks)

Suppose the School's main first year modules are given the following two-letter codes: IM (Introduction to Mathematics), CS (Foundations of CS), LL (Language & Logic), AI (Introduction to AI), SE (Introduction to Software Engineering), IW (Information & Web), RP (Robot Programming), and SW (Software Workshop). Each code letter is assigned a number according to its position in the alphabet as in Question 1. The modules are to be placed into and initially empty hash table of size 11 using linear probing with hash function defined as the sum modulo 11 of the numbers corresponding to the two code letters.

What is the resulting hash table, and what is its load factor?

0	1	2	3	4	5	6	7	8	9	10
LL				SW	RP	IW	AI	CS	IM	SE

The load factor is 8/11 ~ 0.73.

What is the effort involved in checking whether the modules CM (Computational Machines) and HC (Holistic Computing) are in the table?

Checking for CM takes 1 comparison, and HC takes 7 comparisons.

How would the effort of checking for CM and HC change if IM had been deleted?

Deleting IM makes no difference, because a deletion marker would have to be left in its place.

Question 4 (18 marks)

A call centre uses a hash table with open addressing to store customer orders taken during its working hours of 7am to 11pm every day of the year. Orders are kept in the hash table for 5 years and then deleted. It currently has about 5 million entries, and handles about 1 million transactions per year (i.e. between 2 and 3 a minute).

What size hash table would you suggest?

About two or three times the number of entries would be appropriate, i.e. 10 to 15 million.

The performance of the hash table has degraded. Why might this have happened?

Since markers need to be left after an entry is deleted, it can become very costly to search for keys after a number of deletions. In this case, the rate of deletions will be similar to the number of transactions, and there could be as many deletion markers as real entries in the table, and that will have a large negative effect on the performance when searching for entries. In the worst case, the whole table may have to be checked to find if an entry exists. This is particularly likely if the table was set up to be smaller than is optimal now because the number of entries was much smaller then.

Suppose someone suggested copying the entries in the hash table into a new hash table, and it takes 2 milliseconds to copy one item. Would that be a sensible thing to do?

It would take at most 5 or 6 hours to create the whole new hash table (since the old table cannot contain more entries than its size, and it is unlikely to take more time to process a deletion entry than copying a real data entry), so it could easily be completed overnight while the call centre is closed. The current build up of deletions would be removed, and the size of the table could be increased. So, it would probably be a very sensible thing to do.

Question 5 (20 marks)

Suppose you have a hash table of size m, with a hash function that distributes the keys as evenly as possible. Then you insert n items with keys $k_1, k_2, \ldots, k_n$ into the initially empty hash table using linear probing, where $n < m$.

Derive an expression for the probability of having no collisions.

The probability of a collision when inserting item k_i is the load factor at that time, i.e. $(i-1)/m$. So the probability of no collision is $1-(i-1)/m = (m-i+1)/m$. Thus the total probability of having no collisions at all is the product of the individual probabilities, i.e.:

$$p = \prod_{i=1}^{n} \frac{m-i+1}{m} = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \ldots\ldots \cdot \frac{m-i+1}{m} \cdot \ldots\ldots \cdot \frac{m-n+1}{m} = \frac{m!}{m^n (m-n)!}$$

If the size of the hash table is 1,000,000, how many items can you insert before the probability of at least one collision rises above 1%? How many items can you insert before it rises above 50%? Show how you got your answers. [Hint: The numbers involved are too large to compute by hand, so implement and run a small computer program to do it.]

```
int i=1;
float p=1, m=1000000;
while (p > 0.99) p*=(m-i+++1)/m; printf("%d ",i-2);
while (p > 0.50) p*=(m-i+++1)/m; printf("and %d\n",i-2);
```

142 and 1177