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]`

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

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`?

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?

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

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.

Why is this particular secondary hash function not ideal?

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

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?

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

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

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?

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

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?

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.

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.]