COMP20007 – ASSIGNMENT 2 – DAFU AI 766586

1. **Suppose the hash function hashes everything to the same bucket. Using big-O notation, write the complexity of insert and search in terms of size and n. Justify your answer.**
   If the hash function hashes everything to the same bucket, then all the items will be inserted into only one linked list out of all, so there are n nodes in the linked list associated with the always-targeting bucket. Since `list_insert & list_find` of the given linked list implementation both take linear time, the complexity for both insert and search is therefore $O(n)$.

2. **Suppose the hash function spreads the input perfectly evenly over all the buckets. Using big-O notation, write the complexity of insert and search in terms of size and n. Justify your answer.**
   If the hash function spreads the input perfectly evenly over all the buckets, then there will be $n/size$ items inserted to each bucket, in other words there is $n/size$ nodes in each linked list associated with each bucket. So the complexity for both insert and search is $O(n/size)$.

3. **Suppose that the hash function never hashes two different inputs to the same bucket. Using big-O notation, write the complexity of insert and search in terms of size and n. Justify your answer.**
   If the hash function never hashes two different inputs to the same bucket, then there will be at most one item in each bucket, in other words there is at most one node in each linked list associated with each bucket. This results the best case for the complexity of the insert and search function of the given linked list implementation. Therefore it takes $O(1)$ for both insert and search.

4. **Why is this a bad hash function? Give some example input on which the hash table would behave badly if it was using this has function.**
   In the returning value (i.e. `a * key[0] % size`), `a` and `size` are fixed and key[0] is uncertain each time the function is called. So the return value of this function would be the same for the keys with the same first element (i.e. same `key[0]`). For example, if we have the following keys to be hashed (here we assume the keys are string):
   `"x1"`
   `"x2"`
   …
   `"x10000"` etc
   Then `key[0]` of all the keys would be `"x"`. In this example all keys will be hashed into same bucket and thus the hash table would behave badly.

5. **Average-case analysis**
   For average case, the universal hash function will hash keys **approximately** evenly over all the buckets. Then there will be **approximately** $n/size$ items inserted to each bucket, in other words there is **approximately** $n/size$ nodes in each linked list associated with each bucket. So the complexity for both insert and search is $O(n/size)$ using the given linked list implementation.

6. **Generating collisions by trial and error**

My algorithm enumerates random trial keys of fixed length (I define `TRIAL_LENGTH=10` in the program) to generate collisions. It enumerates random strings one by one by assigning a random value (in the range of printable values of unsigned char) to each unsigned char of the current string. Each time a trial string has been enumerated, its hashed value will be checked using `universal_hash`. If the hashed value of current trial is 0, then the function outputs it. The function stops enumeration when it has generated enough (i.e. n) keys which will `universal_hash` to 0.

The probability of enumerating a string which will `universal_hash` to 0 is $1/\text{size}$. So it is expecting to have at least one string which hashes to 0 among size number of strings enumerated. So expected running time of my algorithm for generating 2 hashes to 0 is $O(2\text{size})$.

7. **Generating collisions by clever maths**

Firstly, in the program I define `HASH_TARGET = 0` which is the hashed value required for the strings we are going to generate. By changing `HASH_TARGET`, my method could work for other values of `HASH_TARGET`.

If we want a string of certain length which hashes to 0 (by universal_hash), then we need

$$\mathrm{r_0key[0]} + \cdots + \mathrm{r_{strlen-1}key[strlen-1]} = \mathrm{HASH\_TARGET}\ (\mathrm{mod\ size})$$

where

$$\mathrm{strlen} \leq \mathrm{MAXSTRLEN} \quad \text{and} \quad \mathrm{size} < 255$$

This is equivalent to

$$\mathrm{r_0key[0]} + \cdots + \mathrm{r_{strlen-1}key[strlen-1]} = \mathrm{HASH\_TARGET} + \mathrm{size}\ (\mathrm{mod\ size})$$

My method is to generate keys such that

$$\mathrm{r_0key[0]\ \%\ size} = \mathrm{HASH\_TARGET}, \ldots, \mathrm{r_{strlen-1}key[strlen-1]\ \%\ size} = \mathrm{HASH\_TARGET}$$

In other word, my method is to generate strings of which the value of each unsigned char mod size = `HASH_TARGET`. If this condition is satisfied, then

$$(\mathrm{r_0key[0]} + \cdots + \mathrm{r_{strlen-1}key[strlen-1]})\mathrm{mod\ size} = \mathrm{HASH\_TARGET}$$

as required.

In my algorithm, the sequence of generating strings follows from length 1 to maximum length. My algorithm stops generating strings until enough strings has been done OR it has generated the maximum possible number of strings this algorithm can generate (depending on the given size and HASH_TARGET, but it is always more than 2 as the specification requires). The following piece is my pseudocode:

```
set HASH_TARGET = the desired value of strings to be generated;
set MIN_PRINTABLE/MAX_PRINTABLE = min/max value of printable uchar;
set str_len = 0; // the string length
set min_multiple = floor(MIN_PRINTABLE/size) + 1;
set multiple = min_multiple;
while (not enough strings && str_len < MAXSTRLEN){
	new_val = HASH_TARGET + size*multiple;
	declare a new string of length str_len as new_string;
	if (new_val <= UCHAR_MAX) {
		for (i=0; i<str_len; i++){
			new_string[i]=size;
		}
		print new_string;
		multiple++;
	} else {
		str_len ++; // next round make longer strings
		multiple = min_multiple;  // reset multiple
}
```

Assuming integer operations take a constant amount of time, the `while` loop above only takes constant time for each iteration. To generate n strings (assume n within the capability of my algorithm, i.e. < max number of strings my algorithm can generate as explained previously) the `while` iteration above will run n times, so the expected complexity of my algorithm is $O(n)$.