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"
   "x3"
   …
   "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 $O(n/size)$.

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 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 hashse 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 $\mathrm{r}_0\mathrm{key}[0] + \cdots + \mathrm{r}_{\mathrm{strlen}-1}\mathrm{key}[\mathrm{strlen} - 1] = \mathrm{HASH\_TARGET} \pmod{\mathrm{size}}$ where $\mathrm{strlen} \leq \mathrm{MAXSTRLEN}$ and $\mathrm{size} < 255$. This is equivalent to $\mathrm{r}_0\mathrm{key}[0] + \cdots + \mathrm{r}_{\mathrm{strlen}-1}\mathrm{key}[\mathrm{strlen} - 1] = \mathrm{HASH\_TARGET} + \mathrm{size} \pmod{\mathrm{size}}$.

My method is to make $\mathrm{r}_0\mathrm{key}[0] \,\%\, \mathrm{size} = \mathrm{HASH\_TARGET}, \ldots, \mathrm{r}_{\mathrm{strlen}-1}\mathrm{key}[\mathrm{strlen} - 1] \,\%\, \mathrm{size} = \mathrm{HASH\_TARGET}$. In other word, my method generates strings of which the value of each unsigned char mod size = `HASH_TARGET`.

My algorithm starts by generating string of length 1. It firstly sets the initial value of the last unsigned char (the only one) of this single length string to be $\mathrm{HASH\_TARGET} + \mathrm{multiple} * \mathrm{size}$, where multiple here is 1. So the first string has been generated. Then, if possible ($\mathrm{new\ multiple} * \mathrm{size} \leq \mathrm{UCHAR\_MAX}$, where new multiple = previous multiple +1), it will increase the value of the unsigned char by size. So another string has been generated. It keeps doing that until $\mathrm{new\ multiple} * \mathrm{size} > \mathrm{UCHAR\_MAX}$.

Then my algorithm will generate string of length 2, 3,… and so on until it has generated *n* strings. During the process of generating string(s) of each length, the algorithm firstly sets all unsigned chars of the string (except the last char) to be size, and then it does the same thing as explained in last paragraph for the last char of the string (i.e. increases its value by size to make more strings). If I did not explain clearly how my algorithm works, please see my pseudocode (continuing on next page):

```
set str_len = 0, multiple = 0;
while (not enough strings && str_len < MAXSTRLEN){
      new_val = HASH_TARGET + size*multiple;
      if (new_val <= UCHAR_MAX) {
            for (i=0; i<str_len -1; i++){
                  new_string[i]=size;
            }
            new_string[str_len-1]= new_val;
            multiple++;
      } else {
            str_len ++; // next round make longer strings
            multiple = 1;  // reset multiple
}
```

Assuming integer operations take a constant amount of time, the `while` loop above only takes linear time for each iteration. To generate n strings the `while` iteration above will run n times, so the expected complexity of my algorithm is $O(n)$. Note that my algorithm can only generate limited number of strings (even if there is no running time restriction), depending on $size$.