Name: Pattarapon Buathong Student ID 62070504012

M05 - Space to Time Trade-offs

1. Objectives / Outcome

1) Students will be able to recognize the concept of space and time trade-offs, especially string matching and hashing.
2) Students will be able to design and implement the string matching and hashing for the simple problem with programming language.

2. Related Contents

Space to time trade-offs are an algorithm to speed the computing. Two varieties of space-for-time algorithms are:

1) **Input enhancement —** preprocess the input (or its part) to store some info to be used later in solving the problem**.**
2) **Pre-structuring —** preprocess the input to make accessing its elements easier.

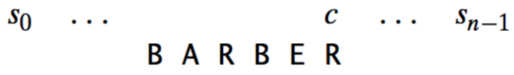
2.1. Input enhancement: String Matching

Several string searching algorithms are based on the input enhancement idea of preprocessing the pattern. Recall that the problem of string matching requires finding an occurrence of a given string of m characters called the pattern in a longer string of **n** characters called the text. The brute-force algorithm for this problem simply matches corresponding pairs of characters in the pattern and the text left to right and, if a mismatch occurs, shifts the pattern one position to the right for the next trial. Since the maximum number of such trials is n − m + 1 and, in the worst case, m comparisons need to be made on each of them, the **worst-case efficiency** of the brute-force algorithm is in the **O(nm)** class. On average however, we should expect just a few comparisons before a pattern’s shift, and for random natural-language texts, the average-case efficiency indeed turns out to be in **O(n+m).**

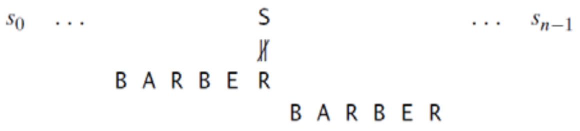
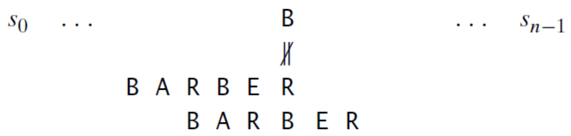
Several faster algorithms have been discovered. Most of them exploit the input-enhancement idea: preprocess the pattern to get some information about it, store this information in a table, and then use this information during an actual search for the pattern in a given text. This is exactly the idea behind the three best-known algorithms of this type:

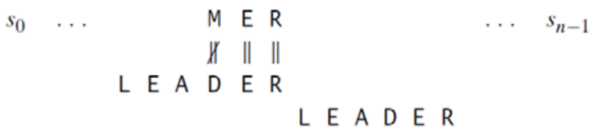
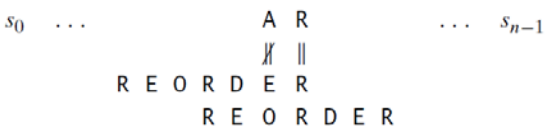
- **Knuth-Morris-Pratt algorithm (KMP)**: It preprocesses pattern *left to right* to get useful information for later searching. The worst-case efficiency of the brute-force algorithm is in the O(n+m).
- **Boyer -Moore algorithm**: It preprocesses pattern *right to left* and store information into two tables. The worst-case efficiency of the brute-force algorithm is in the O(n+m).
- **Horspool's Algorithm**: It is a simplified version of Boyer-Moore algorithm by using just one table, which preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs. It always makes a shift based on the text's *character c* aligned with the last compared (mismatched) character in the pattern according to the shift table's entry for *c.* Note that, this is not necessarily less efficient than the Boyer-Moore algorithm on random strings.

For example, searching for the pattern **BARBER** in some text:



Make as large a shift as possible when mismatching by looking at the last character *c* of the text aligning with the last character of the pattern. In general, the following four possibilities can occur. **How far to shift?**

Case 1: Character c is not in the pattern.  s_0 . . . S . . . s_{n-1} B A R B E R B A R B E R	**Case 2**: Character c in the pattern but not the last one.  s_0 . . . B . . . s_{n-1} B A R B E R B A R B E R

Case 3: Character c matches the last position of the pattern and shows up only once in the pattern.  s_0 . . . M E R . . . s_{n-1} L E A D E R L E A D E R	**Case 4**: Character c matches the last position and shows up many times in the pattern.  s_0 . . . A R . . . s_{n-1} R E O R D E R R E O R D E R

These examples clearly demonstrate that right-to-left character comparison can lead to farther shifts of the pattern than the shifts by only one position always made by the brute-force algorithm. The table's entries will indicate the shift sizes computed by the formula:

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \qquad \textbf{(7.1)}$$

By scanning pattern before search begins and stored in a table called shift table. After the shift, the right end of pattern is *t(c)* positions to the right of the last compared character in text. Here is a simple algorithm for computing the shift table entries. Initialize all the entries to the pattern's length m and scan the pattern left to right repeating the following step m – 1 times. Note that since the algorithm scans the pattern from left to right, the last overwrite will happen for the character's rightmost occurrence—exactly as we would like it to be.

```
ALGORITHM ShiftTable ( P[0 … m-1] )
// Input : Pattern P[0..m – 1] and an alphabet of possible characters
//Output : Table[0..size – 1] indexed by the alphabet's characters and filled with shift
sizes computed by formula (7.1)

for i <- 0 to size -1 do Table[i] <- m
for j <- 0 to m-2 do Table[P[j]] <- m-1-j
return Table
```

So Horspool’s algorithm can summarize the algorithm as follows:

```
ALGORITHM HorspoolMatching ( P[0 … m-1], T[0 … n-1] )
// Input : Pattern P[0..m − 1] and text T [0..n − 1]
//Output : The index of the left end of the first matching substring or −1 if there are
no matches
ShiftTable(P[0 … m-1])
i <- m-1
while i <= n-1 do
    k <- 0
    while k <= m-1 and P[m-1-k] = T[i-k] do
        k <- k + 1
   If k = m
        Return i – m + 1
   else i <- i + Table[T[i]]
Return -1
```

As an example of a complete application of Horspool’s algorithm, consider searching for the pattern **BARBER** in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

Character c	A	B	C	D	E	F	….	R	…..	Z	_
Shift t(c)	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

JIM_SAW_ME_IN_A_BARBERSHOP

BARBER
BARBER
BARBER
BARBER
BARBER
BARBER

A simple example can demonstrate that the worst-case efficiency of Horspool’s algorithm is in **O(nm).** But for random texts, it is in **Θ(n),** and, although in the same efficiency class, Horspool's algorithm is obviously faster on average than the brute-force algorithm. In fact, as mentioned, it is often at least as efficient as its more sophisticated predecessor discovered by R. Boyer and J. Moore.

2.2. Pre-structuring: Hashing

Hashing is a very efficient method for implementing a dictionary, a set with the operations: ***find, insert, delete.*** Based on representation-change and space-for-time trade-off ideas, Important applications: **symbol tables** and **databases** (extendible hashing). Hashing is based on the idea of distributing keys among a one-dimensional array H [0..m − 1] called a **hash table**. The distribution is done by computing, for each of the keys, the value of some predefined function h called the **hash function**. This function assigns an integer between 0 and m − 1, called the **hash address**, to a key. The idea of hashing is to map keys of a given file of size n into a table of size m, called the hash table, by using a predefined function, called the hash function,

h: K →location (cell) in the hash table

For example, student records, key = SSN. Hash function: h(K) = K mod m, where m is some integer (typically, prime) If m = 1000, where is record with SSN= 314159265 stored?

In general, a hash function needs to satisfy somewhat conflicting requirements:

- A hash function has to be easy to compute.
- A hash table's size should not be excessively large compared to the number of keys, but it should be sufficient to not jeopardize the implementation's time efficiency.
- A hash function needs to distribute keys among the cells of the hash table as evenly as possible. (This requirement makes it desirable, for most applications, to have a hash function dependent on all bits of a key, not just some of them.)

Collision Resolution

Obviously, if we choose a hash table's size m to be smaller than the number of keys n, such as h(K1) = h(K2), we will get **Collisions**—a phenomenon of two (or more) keys being hashed into the same cell of the hash table. Good hash functions result in fewer collisions, but some collisions should be expected (birthday paradox).

Name: Pattarapon Buathong Student ID 62070504012

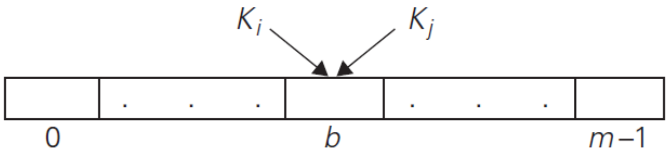


Fig 1. Collision of two keys in hashing: h(Ki) = h(Kj).

Two principal hashing schemes handle collisions differently: open hashing (also called separate chaining), which is each cell is a header of linked list of all keys hashed to it, and closed hashing (also called open addressing), which is one key per cell and in case of collision, finds another cell by linear probing and double hashing.

- Open Hashing (Separate Chaining) or m << n

  Keys are stored in linked lists outside a hash table whose elements serve as the lists' headers, for example, A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED. So, **h(K) = sum of K's letters' positions in the alphabet MOD 13**

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

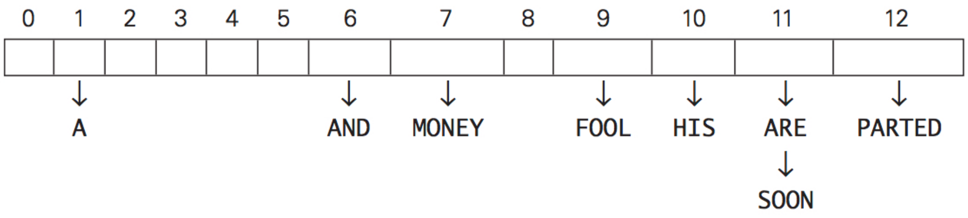
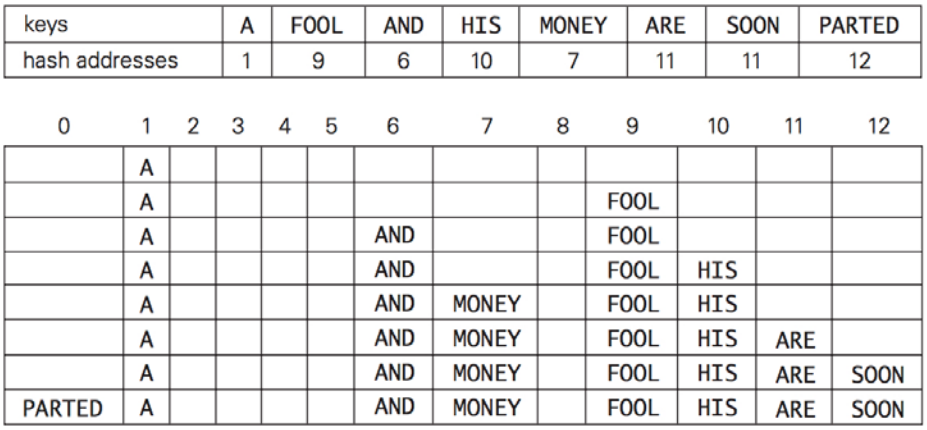


Fig 2. Example of a hash table construction with separate chaining

If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called load factor. For ideal hash functions, the average numbers of probes in successful, S, and unsuccessful searches, U: $S \approx 1+\alpha/2$, $U = \alpha$, where load α is typically kept small (ideally, about 1) and open hashing still works if n > m

- Closed Hashing (Open Addressing) or m ≥ n



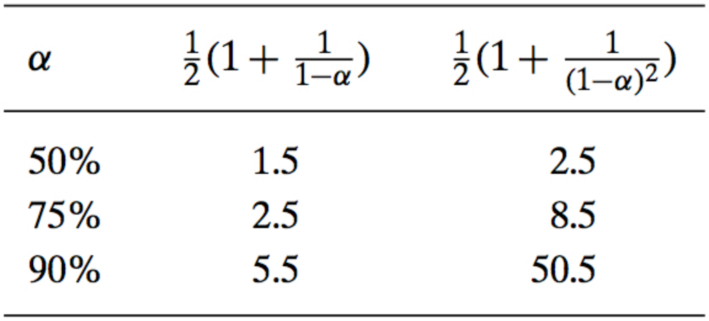
keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Fig 3. Example of a hash table construction with linear probing.

All keys are stored in the hash table, inside its, without using the linked list (m ≥ n). To **insert** for a given key K, Hash key to location h(K) if it is empty. Otherwise, linearly probe the next empty cell for insertion, wrapping around if reaching the end. To **search** for a given key K, we start by computing h(K) where h is the hash function used in the table construction. It will unsuccessful if location h(K) is empty and reaching an empty cell. Otherwise, keeping comparing K to successive non-empty cells

The limitation is it does not work if n > m and deletions are not straightforward. Number of probes to find/insert/delete a key depends on load factor α = n/m (hash table density) and collision resolution strategy. For linear probing: **S = (½) (1+ 1/(1- α)) and U = (½) (1+ 1/(1-α)^2).** As the table gets filled (α approaches 1), number of probes in linear probing increases dramatically:



α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

2.3. Summary

- **Space and time trade-offs** in algorithm design are a well-known issue for both theoreticians and practitioners of computing. As an algorithm design technique, trading space for time is much more prevalent than trading time for space.

- **Input enhancement** is one of the two principal varieties of trading space for time in algorithm design. Its idea is to pre-process the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. Sorting by distribution counting and several important algorithms for **string matching** are examples of algorithms based on this technique.
- **Pre-structuring**—the second type of technique that exploits space-for-time trade-offs—uses extra space to facilitate a faster and/or more flexible access to the data. Hashing and B+-trees are important examples of pre-structuring.
- **Hashing** is a very efficient approach to implementing dictionaries. It is based on the idea of mapping keys into a one-dimensional table. The size limitations of such a table make it necessary to employ a collision resolution mechanism.

3. Group Discussion (Exercises)

3.1. **Hashing:** For the input 40, 60, 37, 83, 42, 18, and hash function h(K) = K mod 11

- Construct the <u>open hash table.</u>

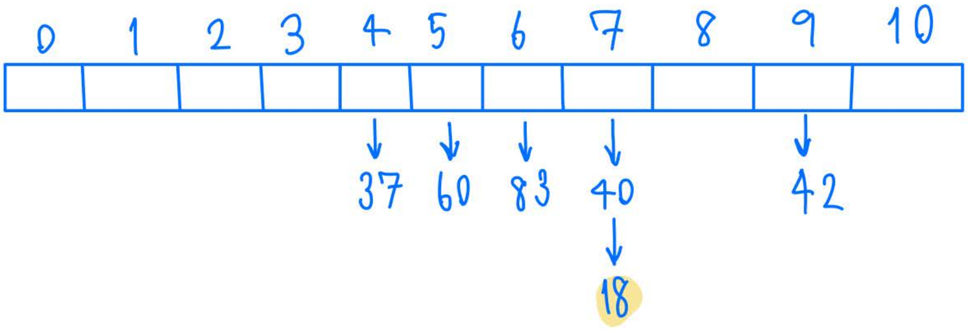
The list of keys : 40, 60, 37, 83, 42, 18

Hash function : h(K) = Kmod11

Hash Address :

K	40	60	37	83	42	18
h(K)	7	5	4	6	9	7

The open hash table :

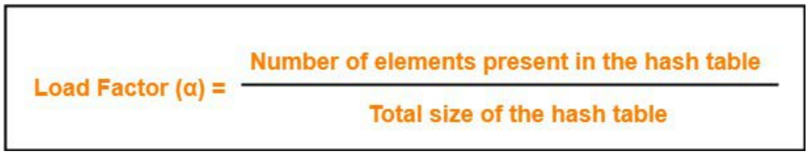


Name: Pattarapon Buathong   Student ID 62070504012

- Find the **largest number** of key comparisons in a successful search in this table.

The largest number of key comparisons in a successful search in this table is **2** (search for K=18)

- Find the **average number** of key comparisons in a successful search in this table.



$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

Load factor : $\alpha = \frac{6}{11} \approx 0.55$

Average number of key comparisons in a sucessful search

$S \approx 1 + \frac{\alpha}{2} \approx 1 + \frac{0.55}{2} = 1.275$

Name: Pattarapon Buathong Student ID 62070504012

3.2. **Hashing:** For the input 40, 60, 37, 83, 42, 18, and hash function h(K) = K mod 11

- Construct the <u>closed hash table</u>

The list of keys : 40, 60, 37, 83, 42, 18

Hash function : h(K) = Kmod11

Hash Address:

K	40	60	37	83	42	18
h(K)	7	5	4	6	9	7

The closed hash table

0	1	2	3	4	5	6	7	8	9	10
							40			
					60		40			
				37	60		40			
				37	60	83	40			
				37	60	83	40		42	
				37	60	83	40	18	42	

- Find **the largest number** of key comparisons in a successful search in this table.

The largest number of key comparisons in a successful search in this table is **6** (search for K=40)

- Find **the average number** of key comparisons in a successful search in this table.

Load factor : $\alpha = \frac{6}{11} = 0.55$

Average number of key comparisons in a sucessful search

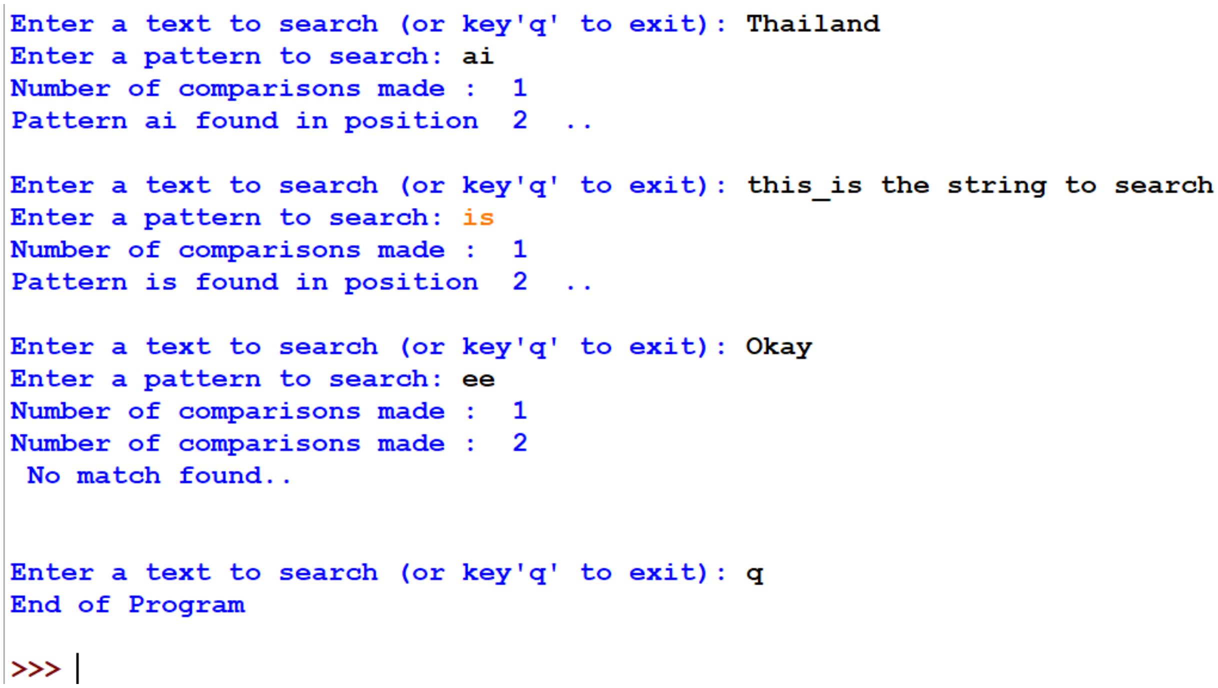
$$S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \approx \frac{1}{2}\left(1 + \frac{1}{0.45}\right) = 1.61$$

Name: Pattarapon Buathong   Student ID 62070504012

4. Lab Assignments (Ass 03)

4.1. **HorspoolStringMatching:** Complete the function **ShiftTable(P[0...m-1])** that generates the amount of shifts to be used by Horspool's string matching algorithm and function **HorspoolMatch (P[0...m-1], T[0...n-1])** that implements the matching algorithm itself. **HorspoolMatch (P[0...m-1], T[0...n-1])** returns the array index where the match is found, and the **number of comparisons** made is updated in the variable **cnt** passed by reference to the function. The program accepts a pattern to search from the user input.

- The sample run is shown below:



```
Enter a text to search (or key'q' to exit): Thailand
Enter a pattern to search: ai
Number of comparisons made :  1
Pattern ai found in position  2  ..

Enter a text to search (or key'q' to exit): this_is the string to search
Enter a pattern to search: is
Number of comparisons made :  1
Pattern is found in position  2  ..

Enter a text to search (or key'q' to exit): Okay
Enter a pattern to search: ee
Number of comparisons made :  1
Number of comparisons made :  2
 No match found..


Enter a text to search (or key'q' to exit): q
End of Program

>>> |
```

Name: Pattarapon Buathong Student ID 62070504012

5. Additional Questions

5.1. Would the Boyer-Moore algorithm work correctly with just the bad-symbol table to guide pattern shifts?

Ans: Yes, the algorithm can word correctly with just the bad-symbol shift table.

5.2. Would the Boyer-Moore algorithm work correctly with just the good-suffix table to guide pattern shifts?

Ans: No, the bad-symbol table is required. Because it is the only one table used by the Boyer-Moore algorithm if the first group of characters does not match.

5.3. Why is it not a good idea for a hash function to depend on just one letter (say, the first one) of a natural-language word?

Ans: The size of the alphabet will limit the number of different values. Apart from that, the probability of a word is not always start with a specific letter is the same for all of the letters.

Name: Pattarapon Buathong Student ID 62070504012

5.4. Birthday paradox The birthday paradox asks how many people should be in a room so that the chances are better than even that two of them will have the same birthday (month and day). Find the quite unexpected answer to this problem. What implication for hashing does this result have?

1 year = 365 days ⟶ ∴ 365 birthday

P(S) = probability for the same birthday

p (d) = probability for distinct birthday

$P(s) + P(d) = 1$

$\therefore P(S) = 1 - P(d)$

1 person in a room, (365/365) 100% chance for a distinct birthday

2 people in a room, $(1 - 1/365) = (364/365)$ chance for a distinct birthday

3 people in a room, $(1 - 2/365) = (363/365)$ 〃 ——— 〃

4 people in a room, $(1 - 3/365) = (362/365)$ 〃 ——— 〃

Ex. If there are 23 people in a room, $(1 - 23/365) = (342/365)$

∴ Probability for the 23 people having distinct birthday

$$P(d) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \dots \times \frac{342}{365}$$

$$= \frac{365!}{342! \times 365^{23}} \quad \left(\frac{365!}{365^{n}(365-n)!} \right)$$

$$= 0.4927$$

$\therefore P(d) \approx 49.3\%$

∴ Chance for at least 2 people having the same b.d.

$P(S) = 1 - P(d) = 0.5073 \approx 50.7\%$

The implication for hashing is that we should expect collisions, even though the size of a hash table is much larger than number of keys.

Name: Pattarapon Buathong Student ID 62070504012

5.5. Answer the following questions for the separate-chaining version of hashing.

5.5.1. Where would you insert keys if you knew that all the keys in the dictionary are distinct? Which dictionary operations, if any, would benefit from this modification?

Ans: If all the keys are distinct, we can insert a new key at the beginning of the linked list. Also, this will make the insertion operation $\theta(1)$, and it will not change the efficiencies of search and delete.

5.5.2. We could keep keys of the same linked list sorted. Which of the dictionary operations would benefit from this modification? How could we take advantage of this if all the keys stored in the entire table need to be sorted?

Ans: Once a key large than the search key is meet, the searching in a sorted list can be stopped. There will be a benefit from insertion and deletion. We could merge the k non-empty lists to get the whole dictionary sorted, to sort a dictionary sorted in linked lists of a hash table; $k - way\ merge$. It is convenient that we arrange the current first elements of the lists in a min-heap, to do this efficiently.

6. Reference

Levitin, A. (2012) Introduction to the Design & Analysis of Algorithms (3rd edition). Pearson. (Chapter 7.2 and 7.3)