Temasek Junior College

JC H2 Computing

Problem Solving & Algorithm Design 16 – Hashing

Hashing

1 Concept: building a data structure that can be searched in ***O(1)*** time.

The technique used for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to identify its location in the list.

In order to do this, need to know more about where the items in the collection might be when go to look for them in the collection. If every item is where it should be, then the search can use a *single* comparison to discover the presence of an item.

2 Hash Table

Hash Table is a data structure made up of two parts, a table (or array) of data, and a key which identifies the location of the data within the table. The data structure stores data in an *associative* manner. Each data value has its own unique index value. Access of data becomes very fast if the index of the desired data was given. Such data is known as **key/value pair.**

Hash Table: a data structure that stores key/value pairs based on an *index calculated from an algorithm*.
Key/value pair: the key and its associated data.

A **hash table** is a collection of items which are stored in such a way as to make it easy to find them later.

Example 1 Student records with NRIC as unique key

	Data [Values]
Record no.	**NRIC [Key] | Name | Gender**
109	**G1923358Q, TAN JUN HUI, F**
..	**…..**
211	**S8744135I, JAKAY TEO, M**
..	**…..**
234	**S7721536G, ADLI ZUHAILY, F**

Key → **Key/Value pair**

StudentID	Index	NRIC [Key] | Name |Gender
'G1923358Q'	19	G1923358Q, TAN JUN HUI, F
	..	…..
'S8744135I'	21	S8744135I, JAKAY TEO, M
	..	…..
'S7721536G'	24	S7721536G, ADLI ZUHAILY, F

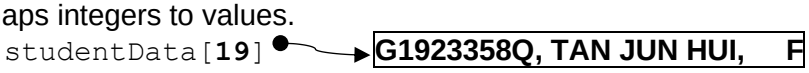
Hash table can be viewed as two-dimensional arrays, or tables with one dimension being the data and the other being the key that identifies the location of the data in the table. Each **key/value** combination is unique within the data structure.

3 Hash technique

A hashing algorithm is carried out on the key, which then acts as an index to the specific location of that data item within the array. It serves as a look-up table that uses a key/value combination.

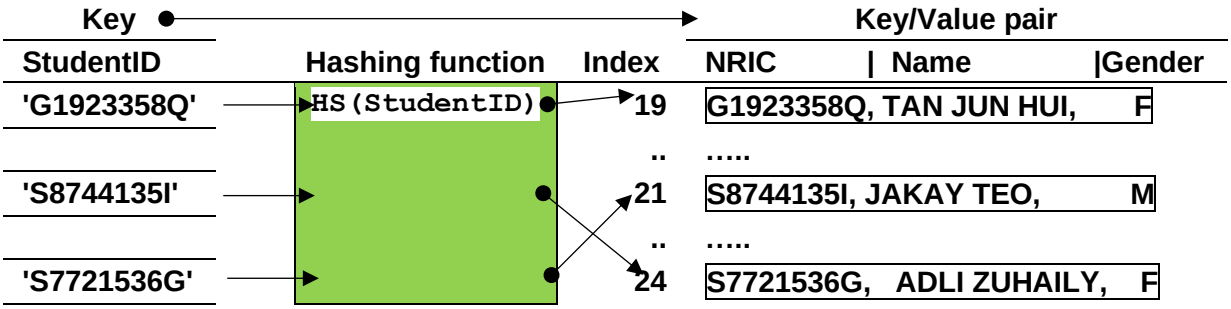
Hashing is a technique to convert a range of key values into a range of indexes of an array. Hashing algorithms must create a range of keys sufficient to assign unique values to each item of data.

- An array maps integers to values.



integer → Key / value pair	
Index	**NRIC | Name |Gender**
19	G1923358Q, TAN JUN HUI, F
..	…..
21	S8744135I, JAKAY TEO, M
..	…..
24	S7721536G, ADLI ZUHAILY, F

- A hash table maps keys to values.



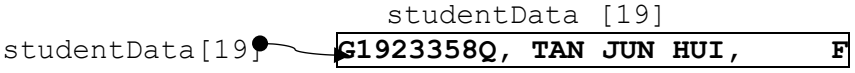
- Key can be of any data type.
- **Each key is unique**.
- Accesses are performed **via keys** rather than positions, thus making searching and retrieval more efficient.

3.1 Insertion and searching using hashing

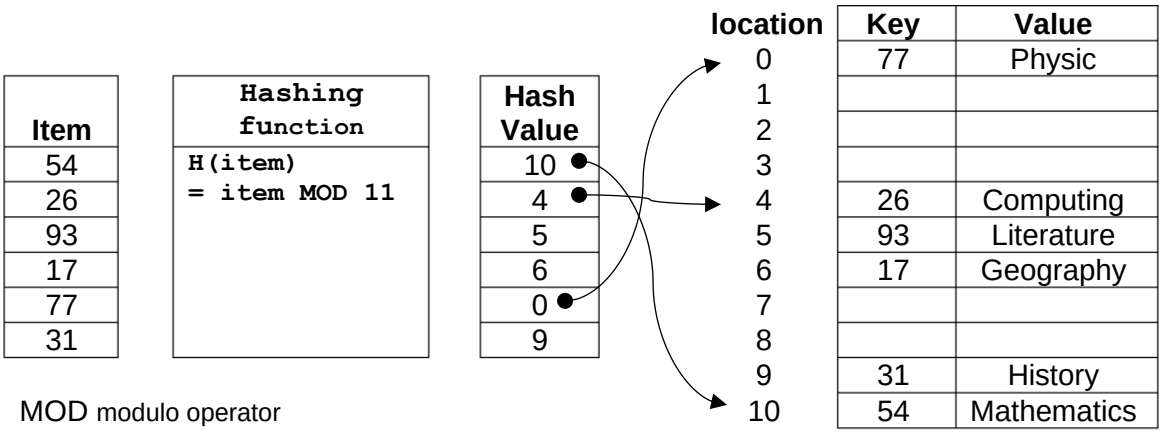
Hash, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses *hash technique* to generate an index where an element is to be inserted or is to be located from.

When the data need to be retrieved, for example, if a search is carried out on the data, the *same* hashing algorithm is used on the key being searched to calculate the index and therefore retrieve the data in one step. This is a very efficient search technique and it is why hashing tables are used extensively in applications where large datasets need to be accessed or where speed of access is critical.

```
Key = 'G1923358Q'
studentDataFound ← studentData [HashFunction(Key)]
                   studentData [HashFunction('G1923358Q')]
                   studentData [19]
```



Example 2 Simple Hash Function Using Remainders



MOD modulo operator

4 A hashing function takes in the data key and returns an INTEGER index.

There is no universally accepted method for creating suitable hashing algorithms and the design of the algorithm depends to a large extent on the application.

Example 3

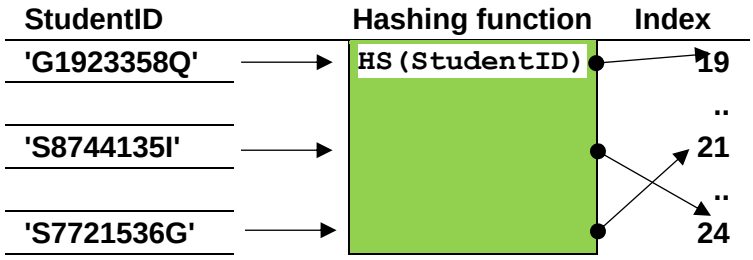
Write program code with the following hashing algorithm:

```
FUNCTION HS(studentID)
      DECLARE HashTotal, CHAR, Hash AS INTEGER
      Set HashTotal to 0
      FOR CHAR 1 TO LENGTH(studentID)
            Get the ASCII code for studentID[CHAR]
            Add it to the HashTotal
      ENDFOR
      Calculate Hash as HashTotal MOD 500
      RETURN Hash
ENDFUNCTION

MAIN()
      DECLARE No AS INTEGER
      studentIDList = ['G1923358Q','S8744135I','S7721536G']

      FOR No = 1 TO LENGTH(studernIDList)
            PRINT studernIDList[No], '->', HS(studernIDList[No])
      ENDFOR
ENDMAIN
```

```
G1923358Q -> 19
S8744135I -> 24
S7721536G -> 21
>>>
```



5 Collisions

- Collision occurs when a hashing algorithm produces the same index for two or more different keys.
- It is theoretically possible to create a perfect hashing algorithm that avoids collisions but in practice, they are unavoidable.
- Rehashing or chaining must be carried out in the event of a collision.

Example 4

Data set:

```
studentIDList = [
'G1923358Q','S8744135I','S7721536G',
'S7733010G','S8740525E','S6709440C',
'S8671460B','S8742506Z','S8734053F',
'S9535433B','S9634621F','S9414585G',
'S9603708F','S9309384I','S9234153B',
'S7847448F','S8747447H','S7713451J',
'S9819719I','S7612861D','S8728919J',
'S9942155B','S9914425G','S9519664D',
'S9632366F','S9239107F','S7742882D',
'S7622993C','S8628526D','S7620076E']
```

0	G1923358Q	19☼
1	S8744135I	24≈
2	S7721536G	21
3	S7733010G	11
4	S8740525E	19☼
5	S6709440C	16►
6	S8671460B	17
7	S8742506Z	41
8	S8734053F	19☼
9	S9535433B	17
10	S9634621F	20
11	S9414585G	26
12	S9603708F	22
13	S9309384I	28
14	S9234153B	12
15	S7847448F	31
16	S8747447H	32
17	S7713451J	21
18	S9819719I	36
19	S7612861D	18
20	S8728919J	37
21	S9942155B	20
22	S9914425G	24≈
23	S9519664D	27
24	S9632366F	24≈
25	S9239107F	20
26	S7742882D	25
27	S7622993C	24≈
28	S8628526D	24≈
29	S7620076E	16►

3	S7733010G	11	
14	S9234153B	12	
5	S6709440C	16►	collide
29	S7620076E	16►	collide
6	S8671460B	17	
9	S9535433B	17	
19	S7612861D	18	
0	G1923358Q	19☼	collide
4	S8740525E	19☼	collide
8	S8734053F	19☼	collide
10	S9634621F	20	
21	S9942155B	20	
25	S9239107F	20	
2	S7721536G	21	collide
17	S7713451J	21	collide
12	S9603708F	22	
1	S8744135I	24≈	collide
22	S9914425G	24≈	collide
24	S9632366F	24≈	collide
27	S7622993C	24≈	collide
28	S8628526D	24≈	collide
26	S7742882D	25	
11	S9414585G	26	
23	S9519664D	27	
13	S9309384I	28	
15	S7847448F	31	
16	S8747447H	32	
18	S9819719I	36	
20	S8728919J	37	
7	S8742506Z	41	

5.1 Three ways to handle collisions

- Rehashing/Linear probing (also known as open addressing)
- Chaining
- Values that collide are stored serially after end of file/in a bucket

5.2 Rehashing/Linear Probing (Open addressing)

Rehashing - Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key.

- Store colliding keys elsewhere in the table.
- If the hash table index is used by another key, store value in the next available space.
- If the key is not found at the generated hash table index, it does not mean it is not in the hash table.
- Keep searching until the key is found or an empty location is encountered.

SN	Key	Hash value
0	G1923358Q	19☼
1	S8744135I	24≈
2	S7721536G	21
3	S7733010G	11
4	S8740525E	19☼
5	S6709440C	16▶
6	S8671460B	17
7	S8742506Z	41
8	S8734053F	19☼
9	S9535433B	17
10	S9634621F	20
11	S9414585G	26
12	S9603708F	22
13	S9309384I	28
14	S9234153B	12
15	S7847448F	31
16	S8747447H	32
17	S7713451J	21
18	S9819719I	36
19	S7612861D	18
20	S8728919J	37
21	S9942155B	20
22	S9914425G	24≈
23	S9519664D	27
24	S9632366F	24≈
25	S9239107F	20
26	S7742882D	25
27	S7622993C	24≈
28	S8628526D	24≈
29	S7620076E	16▶

Table index	Value [data]	Order of insertion
0		
1		
2		
	::::::::	
6	S7733010G	3
7	S9234153B	
8		
	::::::::	
14		
15		
16	S6709440C	5
17	S8671460B	6
18	S9535433B	9
19	G1923358Q[19☼-1]	0
20	S8740525E[19☼-2]	4
21	S7721536G	2
22	S8734053F[19☼-3]	8
23	S9634621F[20-1]	10
24	S8744135I	1
25	S9603708F	12
26	S9414585G	11
27	S7713451J	
28	S9309384I	
29	S7612861D	
30	S9942155B	
31	S7847448F	
32	S8747447H	
33	S9914425G	
34	S9519664D	
35	S9632366F	
36	S9819719I	
37	S8728919J	
38	S9239107F	
39	S7742882D	
40	S7622993C	
41	S8742506Z	7
42	S8628526D	
43	S7620076E	
…	…	

Example 5

Write a program code for hashing algorithm with linear probing using data set from **Example 4** in Jupyter Notebook.

5.2.1 More on linear probing

The technique of linear probing discussed so far is an example of collision resolution by rehashing. If the hash function produces a collision, the hash value is used as the input to a *rehash function* to compute a new hash value. In the previous section, add 1 to the hash value to create a new hash value; that is,

The rehash function Re_Hash(HashValue) = (HashValue + 1) % 100

For rehashing with linear probing, in general:

Re_Hash(HashValue) = (HashValue + constant) % array_size

As long as constant and array_size are relatively prime-that is, if the largest number that divides both of them evenly is 1.

5.2.2 Quadratic probing

Resolving a hash collision by using the rehashing formula (HashValue + $\boldsymbol{T}^2$) % array_size
Where ***T*** is the number of times that the rehash function has been applied.

Makes the result of rehashing dependent on how many times the refresh function has been applied. In the ***Tth*** rehash, the function is

Re_Hash(HashValue) = (HashValue + $\boldsymbol{T}^2$) % array_size

5.3 Records removing

- Removing a record from an open hash table is problematic.
- Replacing the removed record with NULL values may interfere with subsequent search operations.
- Possible solutions:
  - Do not allow removal of records.
  - Replace the value of deleted key with a **"tombstone"** value that will not be used for any key.

5.4 Clustering

- The tendency of elements to become unevenly distributed in the hash table, with many elements clustering around a single hash location.
- Clustering occurs when a hashing algorithm produces indices that are not randomly distributed.
- This increases the possibility of collisions as well as the number of comparisons needed when searching for a record.

SN	Key	Hash value
6	S8671460B	17
7	S8742506Z	41
8	S8734053F	19☼
9	S9535433B	17
10	S9634621F	20
11	S9414585G	26
12	S9603708F	22
13	S9309384I	28
14	S9234153B	12

15	….	
16		
17		
18	S9535433B	9
19	G1923358Q[19☼-1]	0
20	S8740525E[19☼-2]	4
21	S7721536G	2
22	S8734053F[19☼-3]	8
23	S9634621F[20-1]	10
24	S8744135I	1
25	S9603708F	12
26	S9414585G	11
27		
28		
29	…	

Example 6

Write program code for hashing algorithm with linear probing using data set from **Example 4.**

Calculate the index using an algorithm that adds the numbers (digits) in the key together and then performs a modulo 28 sum on the result,

For the first data item the value of the key might be '**G1923358Q**'

- Select the characters (digits) from 2nd to 7th position, e.g. **192335**.
- Add the numbers (digits) together 1+9+2+3+3+5= 23.
- Perform modulo 28 calculation so divide by 28 = 0 with remainder 23.
- Therefore the Index = 23.
- The data is placed in slot 23.

SN	Key	Hash value and linear probing
0	'G1923358Q'	23
1	'S8744135I'	27
2	'S7721536G'	25
3	'S7733010G'	21
4	'S8740525E'	26
5	'S6709440C'	2
6	'S8671460B'	4
7	'S8742506Z'	26 27 28
8	'S8734053F'	27 28 29
9	'S9535433B'	1
10	'S9634621F'	2 3
11	'S9414585G'	3 4 5
12	'S9603708F'	25 26 27 28 29 30
13	'S9309384I'	4 5 6
14	'S9234153B'	24
15	'S7847448F'	6 7
16	'S8747447H'	6 7 8
17	'S7713451J'	27 28 29 30 31
18	'S9819719I'	7 8 9
19	'S7612861D'	2 3 4 5 6 7 8 9 10
20	'S8728919J'	7 8 9 10 11
21	'S9942155B'	2 3 4 5 6 7 8 9 10 11 12
22	'S9914425G'	1 2 3 4 5 6 7 8 9 10 11 12 13
23	'S9519664D'	8 9 10 11 12 13 14
24	'S9632366F'	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
25	'S9239107F'	24 25 26 27 28 29 30 31 32
26	'S7742882D'	8 9 10 11 12 13 14 15 16
27	'S7622993C'	7 8 9 10 11 12 13 14 15 16 17
28	'S8628526D'	3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
29	'S7620076E'	22

Table index	Value [data]
0	
1	S9535433B
2	S6709440C
3	S9634621F
4	S8671460B
5	S9414585G
6	S9309384I
7	S7847448F
8	S8747447H
9	S9819719I
10	S7612861D
11	S8728919J
12	S9942155B
13	S9914425G
14	S9519664D
15	S9632366F
16	S7742882D
17	S7622993C
18	S8628526D
19	
20	
21	S7733010G
22	S7620076E
23	G1923358Q
24	S9234153B
25	S7721536G
26	S8740525E
27	S8744135I
28	S8742506Z
29	S8734053F
30	S9603708F
31	S7713451J
32	S9239107F
33	
34	

5.5 Load Factor

- Hash tables have a load factor which is the number of keys divided by the size of database.

Load factor of Example 6

number of keys = 30
size of database = 31
Load factor = number of keys divided by the size of database
= 30 / 31
= 0.96

Example 7

For each Hash value-x given below:

a. Estimate the number or records used in database
b. Calculate the load factors
c. Explain in details will clashes occur.

SN	Key	Hash value-1	Hash value-2	Hash value-3	Hash value-4	Hash value-4
0	**'G1923358Q'**	**58**	**358**	**23358**	**923358**	1923358
1	**'S8744135I'**	**35**	**135**	**44135**	**744135**	8744135
2	**'S7721536G'**	**36**	**536**	**21536**	**721536**	7721536
3	**'S7733010G'**	10	010	33010	733010	7733010
4	**'S8740525E'**	25	525	40525	740525	8740525
5	**'S6709440C'**	40	440	09440	709440	6709440
6	**'S8671460B'**	60	460	71460	671460	8671460
7	**'S8742506Z'**	06	506	42506	742506	8742506
8	**'S8734053F'**	53	053	34053	734053	8734053
9	**'S9535433B'**	33	433	35433	535433	9535433
10	**'S9634621F'**	21	621	34621	634621	9634621
11	**'S9414585G'**	85	585	14585	414585	9414585
12	**'S9603708F'**	08	708	03708	603708	9603708
13	**'S9309384I'**	84	384	09384	309384	9309384
14	**'S9234153B'**	53	153	34153	234153	9234153
15	**'S7847448F'**	48	448	47448	847448	7847448
16	**'S8747447H'**	47	447	47447	747447	8747447
17	**'S7713451J'**	51	451	13451	713451	7713451
18	**'S9819719I'**	19	719	19719	719719	9819719
19	**'S7612861D'**	61	861	12861	812861	7612861
20	**'S8728919J'**	19	919	28919	728919	8728919
21	**'S9942155B'**	55	155	42155	942155	9942155
22	**'S9914425G'**	25	425	14425	914425	9914425
23	**'S9519664D'**	64	664	19664	519664	9519664
24	**'S9632366F'**	66	366	32366	632366	9632366
25	**'S9239107F'**	07	107	39107	239107	9239107
26	**'S7742882D'**	82	882	42882	742882	7742882
27	**'S7622993C'**	93	993	22993	622993	7622993
28	**'S8628526D'**	26	526	28526	628526	8628526
29	**'S7620076E'**	76	076	20076	620076	7620076

5.6 Redundancy

With **Example 7**, the problem of clashes has been solved, but at the expense of using up vast amounts of memory (in fact more memory than the computer has at its disposal). This is known as redundancy. Having so much redundancy in the algorithm is obviously not acceptable. A sensible hashing algorithm needs to be a compromise that minimises redundancy without producing too many clashes or collisions.

Example 8

Write program code for hashing algorithm with linear probing for checking Redundancy in each case given in **Example 7**.

5.7 Designing a Hashing Algorithm

- A perfect hashing algorithm should map each given key to a unique location in the table.
- A good hashing algorithm should:
    - reduce the chances of collision,
    - distribute keys uniformly over the table to avoid clustering, and
    - have a suitable load factor.

5.8 Choosing a hashing algorithm

The basic example above demonstrates a few features and associated problems when creating a suitable algorithm:

- A numeric value needs to be generated from the data in order to perform the calculation. For non-numeric keys such as text and other characters, the ASCII or Unicode value of each character is normally used.

- It must generate unique indices. For example, if the 5th data item was 974052 [S9740525E], the algorithm would generate the index of 27 again. There is already data stored in this location so this has created a **collision**. It is theoretically possible to create a perfect hashing algorithm that avoids collisions, but in practice, they are unavoidable. A good algorithm will create as few as possible and needs a mechanism to cope with collisions as they occur.

- It needs to create a uniform spread of indices. For example, if you were storing millions of items of data into millions of slots the algorithm needs to provide an even spread of index values from the data and avoid **clustering**. This cuts down the possibility of collisions.

- There must be enough slots to store the volume of data. For example, if a database is going to store 1 million records, the algorithm must be capable of producing at least 1 million indices. In fact it would need more than this to avoid collisions as the table fills up. Hash tables have a **load factor** which is the number of keys divided by the number of slots. A high load factor means that it will become increasingly difficult for the algorithm to produce a unique result.

- It has to balance the issues of speed with complexity. For example, an algorithm for a database needs to calculate the index very quickly. An algorithm for encryption needs to be very complex, but may not need to calculate the index quickly.

5.9 Chaining

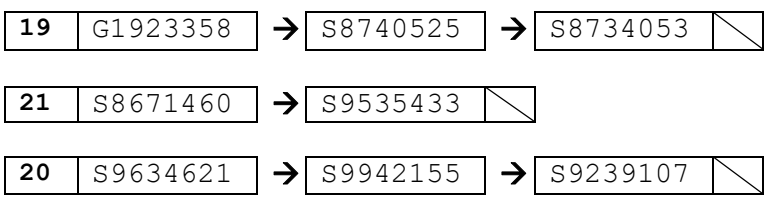
- Another method used to resolve collisions.
- Colliding keys are stored in a list at the same index.
- New record (storing key/value pair) is added to the list at either the head or tail of the list.

Example 9 Chaining [Linked lists]

SN	Key	Hash value
0	'G1923358Q'	19☼
1	'S8744135I'	24≈
2	'S7721536G'	21
3	'S7733010G'	11
4	'S8740525E'	19☼
5	'S6709440C'	16▶
6	'S8671460B'	17
7	'S8742506Z'	41
8	'S8734053F'	19☼
9	'S9535433B'	17
10	'S9634621F'	20
11	'S9414585G'	26
12	'S9603708F'	22
13	'S9309384I'	28
14	'S9234153B'	12
15	'S7847448F'	31
16	'S8747447H'	32
17	'S7713451J'	21
18	'S9819719I'	36
19	'S7612861D'	18
20	'S8728919J'	37
21	'S9942155B'	20
22	'S9914425G'	24≈
23	'S9519664D'	27
24	'S9632366F'	24≈
25	'S9239107F'	20
26	'S7742882D'	25
27	'S7622993C'	24≈
28	'S8628526D'	24≈
29	'S7620076E'	16▶

index	Value [data]
0	
1	
:	:
:	:
9	
10	
11	S7733010G
12	S9234153B
13	
14	
15	
16	S6709440C|S7620076E
17	S8671460B|S9535433B
18	S7612861D
19	G1923358Q|S8740525E|S8734053F
20	S9634621F|S9942155B|S9239107F
21	S7721536G|S7713451J
22	S9603708F
23	
24	S8744135I|S9914425G|S9632366F|S7622993C|S8628526D
25	S7742882D
26	S9414585G
27	S9519664D
28	S9309384I
29	
30	
31	S7847448F
32	S8747447H
33	
34	
35	
36	S9819719I
37	S8728919J
38	
39	
40	
41	S8742506Z
42	
43	::::::

Store the following key/value pairs in multiple linked lists:



Example 10

Write a program code for hashing algorithm with chaining

5.10 Comparisons of linear probing and chaining

5.10.1 Advantages of chaining over linear probing

- Linear probing has slower insertion as it may require several attempts to find an empty slot.
- Chaining may implement smaller hash table size compared to linear probing.
- Array index overflow is not a problem.
- Deletion is simpler in a chained table (no “tombstone” value needed).

5.10.2 Disadvantages of chaining over linear probing

- List could get long and negatively impact performance.
- Worst case occurs when all records are stored in one list – typically result of a poorly designed hashing algorithm.

6 Use of hashing algorithms

Hashing algorithms have many applications:

- Databases: Used to create indices for databases enabling quick storage and retrieval of data.
- Memory addressing: Used to generate memory addresses where data will be stored. It is particularly useful for cache memory, where data is placed temporarily allowing the user fast access to programs and data stored in the cache.
- Operating systems: As an example of memory addressing, some operating systems use hashing tables to store and locate the executable files of all its programs and utilities.
- Encryption: Used to encrypt data, hence the term ‘encryption key’. In this case the algorithm must be complex so that if data is intercepted it is not possible to reverse-engineer it.
- Checksums: A value calculated from a set of data that is going to be transmitted. On receipt the algorithm is run again on the data and the two results are compared as a way of checking whether the data has been corrupted during transmission. [Networking]
- Digital signature is method of ensuring the authenticity of the sender in Internet communication [Networking].
- Programming: Used to index keywords and identifiers as the compiler will need access to these quickly as it compiles the program.
- Random access to data means that data is stored in random order but that a series of indexes keeps the address of the data so that a search can go straight to it. Data such as a customer file that requires fast access to a particular customer record is held in a random access fashion.
- Random organisation of files may be implemented using a hashing algorithm. The hashing algorithm calculates a number which files the address of the record. This same number is used to retrieve the record.
    - Random (file) access: Each record is allocated a record key and the record is stored at an address calculated from this record key using a hashing function

Tutorial 16 [Hashing] Q11, Q12

1. What is a hash table data structure?
2. Suggest a suitable hashing algorithm that could be used to store the names of everyone in your class. Write code to implement your solution.
3. Identify three possible applications for hashing algorithms.
4. What are the main features of a good hashing algorithm?
5. What can you do to minimise the likelihood of collisions when creating a hash table?
6. What is load factor?
7. What is clustering and how is it caused?
8. Explain in detail how a hashing algorithm can deal with collisions.
9. Is it possible to write a perfect hashing function?
10. Establish a hashing algorithm that could allow random access to a student file using the student name as the key field.
11. Describe what is meant by a hashing algorithm and explain why such an algorithm can lead to clashes or collisions. [3]
12. Each part stored by the parts department [in a garage] has a 6 digit key.
    - The first two digits refer to the model of the car for which it is designed.
    - The third digit is used to indicate the year of manufacture.
    - The last three digits are used to ensure the key for this part is unique.

    (i) Devise a hashing algorithm which would be suitable for storing 10,000 parts. [2]

    (ii) State **two** key values which hash to the same address. [1]

    (iii) Describe a method of handling collisions when using a hashing algorithm. [2]

[Question 12b Cambridge AS Level & A Level Computing 9691 Paper 1 June 2007]