Unit 6

Hash Tables

	Syllabus	Guidelines	Suggested number of lectures
1	Introduction to hashing, hash tables and hashing functions, insertion, resolving collision by open addressing, deletion, searching and their analysis, properties of a good hash function.	Chapter 10, Section 10.1 - 10.4, Ref 1	4

References

1. Ref 1: . Drozdek, A., (2012), *Data Structures and algorithm in C++.* 3rd edition. Cengage Learning. Note: 4th edition is available. Ebook is 4th edition

2. Ref 2.: Goodrich, M., Tamassia, R., & Mount, D., (2011). *Data Structures and Algorithms Analysis in C+ +.* 2nd edition. Wiley.

3. Additional Resource 3: Sahni, S. (2011). Data Structures, Algorithms and applications in C++. 2ndEdition, Universities Press

4. Additional Resource 4: Tenenbaum, A. M., Augenstein, M. J., & Langsam Y., (2009), *Data Structures Using C and C++.* 2nd edition. PHI.

Note: Ref1, Additional resource etc. as per the LOCF syllabus for the paper.

ch-10. Hashing. (10.1 - 10.3) (w.e.f - Jan'12.) A. Drozdek

Hashing :- A function 'h' that transform a particular key 'K', into an index in the table used for storing items of the same type as 'K', then 'h' is called a 'hash function'.

OR.

Hash function :- A f^n that transform a key into a table index is called a hash function.

eg:- A co. maintains an inventory file in which each part is assigned a unique serial no. of 7 digits. A co. has less than 1000 parts & there is only a single record for each part. Then an array of 1000 elements is sufficient to contain entire file. Array is indexed from 0 to 999.

The last three digits of part no. are used as the index for the part's record in the array.

Position	Key (k)	Record.
0	4967000	
2	8421002	
396	4618396	
399	1286399	
990	0000990	
995	9846995	
997	4967997	
999	0001999	

Thus, hash f^n is:-

$$h(k) = k \% 1000$$

Drawback :-

Keys 4967999 & 2946999 points to the same table entry. one key cannot occupy the space in array already used.

This is called 'collission' or 'Hash collission' or 'Hash clash'.

Perfect Hash function :- If h transforms different keys into different, unique nos, it is called perfect Hash f^n.

Need of Hashing

Basic searching techniques taken O(n) [Linear search] or O($\log_2 n$) (Binary search) time as 'value' of key is only indication of posiˈbon.

Using Hashing technique, if key 'k' is known & hash funcˈbon 'h' is known, the posiˈbon in table can be accessed directly without making any test thus reducing the time from O(n) or (O($\log_2 n$)) to O(1).

10.1. Hash functions

The no. of Hash functions that can be used to assign positions to 'n' items in table of 'm' positions ($n \leq m$) is equal to m^n.

The no. of perfect hash f^n is no. of different placements of these items (n) in the Table (m) & is equal to $P(m,n) = \frac{m!}{(m-n)!}$

eg:- 50 elements (n) &

100-cell array (m)

Hash functions = 100^{50}

Perfect Hash $f^n = P(100,50) = \frac{100!}{50!} = 10^{94}$.

Types of Hash functions :-

1) Division.

A hash f^n must guarantee that the no. it returns is a valid index to one of the table cells.

If TSize = Sizeof(Table). , then

$h(K) = K \bmod TSize$.

Division method is a prefered choice if very little is known about the keys.

2) Folding.

Key 'K' is divided into several parts which are combined or folded together & transformed in a certain way to create the target address.

There are two types of folding :-

(a) shift folding.
(b) boundary folding.

Shift folding.

Eg:- A social security no. (SSN) 123-45-6789 can be divided in 3 parts 123, 456, 789 & then added. Result 1,368 is divided modulo TSize. If TSize=1000, then 1368 % 1000 = 368 can be used for address.

Division can be done in many different ways.

Also SSN no. can be divided in 5 parts as 12, 34, 56, 78 & 9, then added & divide the result modulo TSize.

Boundary folding

The key is seen as written on a peice of paper that is folded on the borders b/w different parts of keys. Thus, every other part is put in reverse order.

Eg:- SSN no. is again div. in three parts 123, 456, 789.

The addition becomes

```
  123
+ 654 (folded part)
  789
-------
 1566
```

Then 1,566 % 1000 = 586.

Bit-oriented version of shift folding is obtained by applying the exclusive-or operation.

In case of strings, process all characters of the strings by "Xor'ing" them together & using the result for the address.

eg:- h("abcd") = "a" xor "b" xor "c" xor "d".

or

h("abcd") = "ab" xor "cd"....

3) Mid-Square function

The key is squared & mid-part of the result is used as the address.

In this method, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys.

For eg:- If K = 3121, then

9740641 = $(3121)^2$

If TSize = 1000, then h(3121) = 406.

4) Extraction:-

In this method, only a part of key is used to compute the address.

eg:- SSN - 123-456-789

If first 4 digits are used, then, h(K) = 1234

" last " " " h(k) = 6789.

If first 2 & last 2 digits are used, then h(k) = 1289.

5) Radix Transformation

Key K is transformed into another number base.

eg:- if $K = (345)_{10}$ then its value in base 9 is 423. ie $(345)_{10} = (423)_9$

This value `423' is then divided modulo TSize & resulting no. is used as the address of the location to which K should be hashed.

collisions still cannot be avoided.

eg:- $K_1 = \Rightarrow (345)_{10} = (423)_9 = 423 \% 100 = 23$

$K_2 = (264)_{10} = (323)_9 = 323 \% 100 = 23$

Both Keys K_1 & K_2 hashed to the same address 23.

10.2. Collision Resolution

Factors that minimize the no. of collisions :-

1) hash function.
2) Table size.

Collision Resolution methods

(I) Open Addressing

In this method, when a key collides with another key, the collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed.

If position $h(k)$ is occupied, then the positions in the probing sequence

$norm(h(k) + p(1))$, $norm(h(k) + p(2))$, ..., $norm(h(k) + p(i))$, ... are tried until either an available cell is found or the same positions are tried repeatedly or the table is full.

Here function 'p' is the probing function,
'i' is a probe &
norm is a normalization f^n (most likely, division modulo TSize).

Methods of Probing

(A) Linear Probing :-

In Linear Probing method, $p(i) = i$, & thus for i^{th} probe, position tried is $(h(k) + i)$ mod TSize.

In this method, position in which key can be stored is found by sequentially searching all positions starting from the position calculated by the hash f^n until an empty cell is found.

If the end of table is reached & no empty cell has been found, the search is continued from the beginning of the table & stops in the cell preceding the one from which the search started.

Linear Probing method has a tendency to create clusters in the table.

eg:- Suppose K_i is hashed to position 'i'.

Insert: A_5, A_2, A_3

0	
1	
2	A_2
3	A_3
4	
5	A_5
6	
7	
8	
9	

Insert : B_5, A_9, B_2

0	
1	
2	A_2
3	A_3
4	B_2
5	A_5
6	B_5
7	
8	
9	A_9

Insert : B_9, C_2

0	B_9
1	
2	A_2
3	A_3
4	B_2
5	A_5
6	B_5
7	C_2
8	
9	A_9

To locate C_2 = 5 Probes (index 3 – index

A large cluster is created after every insertion step.

Empty cells following clusters have a much greater chance to be filled than other positions.

Probability = (size of (cluster) + 1) / TSize.

other empty cells have Probability = 1 / TSize.

cluster degrades the performance of a hash table for storing & retrieving data.

Problem at present is to avoid cluster buildup & sol^n is to found a careful choice of probing function. Quadratic Probing implements this choice.

(B) Quadratic Probing

use Quadratic function so that resulting formula is,

$$p(i) = h(K) + (-1)^{i-1} \left((i+1)/2 \right)^2 \text{ for } i = 1, 2, \ldots, Tsize-1.$$

This formula can also be expressed in a simpler form as a sequence of probes :

$$h(K) + i^2, \quad h(K) - i^2 \text{ for } i = 1, 2, \ldots, (TSize-1)/2$$

Including the first attempt to hash K, will result in sequence :-

$h(K)$,

$h(K) + 1$, $h(k) - 1$, $[i = 1]$

$h(K) + 4$, $h(k) - 4$, ... $[i = 2]$

$h(k) + \left((TSize-1)^2/4 \right)$, $h(k) - \left((TSize-1)^2/4 \right)$. $\left[i = \frac{(TSize-1)}{2} \right]$

all divided modulo TSize. #

The size of Table should not be an even no. because only even positions or only odd positions are tried depending on the value of $h(K)$.

eg:- TableSize should be a prime $4j+3$ (not necessarily) of an integer j, which gaurantees the inclusion of all positions in the probing sequence. For eg:- If $j = 4$, then TSize = 19. Assume, $h(K) = 9$ for some K, the resulting seq. of Probes is,

✓9,	(Home address)	
✓10,	$(h(k)+1)$	$(9+1)\%19$
8,	$(h(k)-1)$	$(9-1)\%19$
✓13,	$(h(k)+4)$	$(9+4)\%19$
5,	$(h(k)-4)$	$(9-4)\%19$
✓18,	$(h(k)+9)$	$(9+9)\%19$
0,	$(h(k)-9)$	$(9-9)\%19$
✓6,	$(h(k)+16)$	$(9+16)\%19$ [cyclic] $25\%19=6$
12,	$(h(k)-16)$	$(9-16)\%19$ ["]
✓15,	$(h(k)+25)$	$(9+25)\%19$ (")
3,	$(h(k)-25)$	$(9-25)\%19$ (")
✓7,		

11, 1, 17, 16, 2, 14, 4 . . $\%19$

0 1 2 3 4 5 6 7 8 (9) 10 11 12 13 14 15 16 17 18

$19\overline{)34}$ (1, -19, 15)

formula determining the seq. of probes chosen for quadratic probing is not,

$h(K) + i^2$ for $i = 1, 2, \ldots,$ TSize − 1 : because (not TSize−1/2)

the first half of the sequence

$$h(K)+1,\ h(k)+4,\ h(k)+9,\ +\ldots+,\ h(k)+(TSize-1)^2$$

covers only half of the Table, & second half of the sequence repeats the first half in the reverse order.

For eg :- TSize = 19, $h(K) = 9$, then the seq. is :-

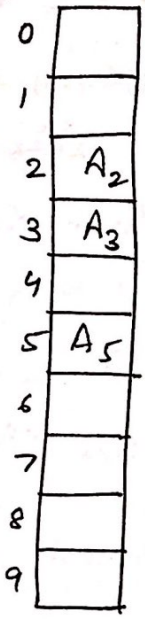
9, 10, 13, 18, 6, 15, 7, 1, 16, 14, 14, 16, 1, 7, 15, 6, 18, 13, 10.

9 → H.Add. ; 10, 13, 18, 6, 15, 7, 1, 16, 14 : first half ; 14, 16, 1, 7, 15, 6, 18, 13, 10 : second half. (last → $h(k)+(TSize-1)$)

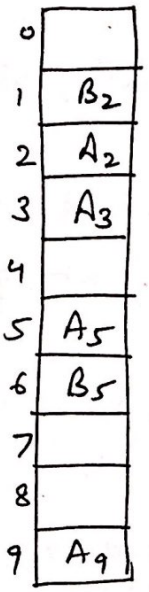
Thus, probes that renders the same address are of the form

$$i = TSize/2 + 1 \quad \& \quad j = TSize/2 - 1.$$

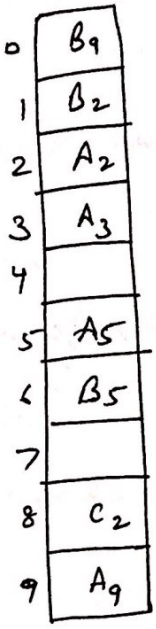
eg 2. Insert: A_5, A_2, A_3



Insert B_5, A_9, B_2



Insert : B_9, C_2



$C_2 \rightarrow 2$

$2-1^2$ & $2+1^2$

1 & 3

Already full.

$2-2^2$ & $2+2^2$

$2-4$ & $2+4$

8 & 6

$\therefore C_2 \rightarrow 8$ full

To locate B_2 → 2 Probes

To locate C_2 → 4 Probes. (5 probes in linear probing).

Quadratic Probing Also does not completely remove the problem of cluster Build up because keys hashed to same location uses the same probe sequence. Such clusters are called secondary clusters.

secondary clusters are less harmful than Primary clusters. (formed in Linear Probing).

(c) Double Hashing

The problem of secondary clustering is best addressed with double hashing.

This method utilizes two hash functions :-

1) one for accessing the primary position of a key, i.e. f^n 'h'.

2) and second f^n 'h_p' for resolving conflicts.

Thus, the Probing sequence becomes :-

$h(K)$, $h(K) + h_p(K)$, ... , $h(k) + i.h_p(K)$, ...

(all divided modulo TSize).

Table size should be a prime no, so that each position in the table can be included in the sequence.

If key K_1 is hashed to the position j , the Probing seq. is

j , $j + h_p(K_1)$, $j + 2.h_p(K_1)$, $j + 3.h_p(K_1)$, ...

(all divided modulo TSize).

If another key K_2 is hashed to $j + h_p(K_1)$, then the next position tried is $\underbrace{j + h_p(K_1)}_{i} + \underbrace{h_p(K_2)}_{\text{second } f^n}$ and not $j + 2.h_p(K_1)$, which avoids secondary clustering if h_p is carefully chosen.

Here, $i = j + h_p(K_1)$ i.e. $i + h_p(K)$, $i + 2h_p(K)$, $i + 3h_p(K)$ . . . . .

Thus, $j + h_p(K_1) + h_p(K_2)$

Key K_1 hashed to j i.e

$$h(K_1) = j \quad \&$$

probing f^n :- $h_p(K_1) = i.h(K_1)+1 = i.j+1$

Probing seq :-

(K_1) :- $h(K_1),\ h(K_1)+1.h_p(K_1),\ h(K_1)+2.h_p(K_2),\ \ldots$

i.e $j,\ j+1.(1.j+1),\ j+2(2j+1),\ \ldots$

i.e $j,\ 2j+1,\ 5j+2,\ \ldots$

If Key K_2 hashed to $2j+1$

i.e $h(K_2) = 2j+1$

Probing f^n :- $h_p(K_2) = i.h(K_2)+1 = i(2j+1)+1$

Probing seq

(K_2) :- $h(K_2),\ h(K_2)+1.h_p(K_2),\ h(K_2)+2.h_p(K_2),\ \ldots$

i.e $2j+1,\ 2j+1+1.(1.(2j+1)+1),\ 2j+1+2(2.(2j+1)+1),\ \ldots$

i.e $2j+1,\ 2j+1+2j+1+1,\ 2j+1+2(4j+2+1),\ \ldots$

i.e $2j+1,\ 4j+3,\ 2j+1+8j+4+2,\ \ldots$

i.e $2j+1,\ 4j+3,\ 10j+7,\ \ldots$

The efficiency of all these three methods can be seen when it is for unsuccessful searches i.e searching for elements not in the table.

Consider linear Probing used for collision resolution. If K is not in the table, then starting from the position h(K), all consecutively occupied cells are checked. Therefore search time increases with the no. of elements in the table.

Formulas approximating, for different hashing methods, the average no. of trials for successful & unsuccessful searches.

	Linear Probing.	Quadratic Probing	double hashing
successful search	$\frac{1}{2}\left(1+\frac{1}{1-LF}\right)$	$1-\ln(1-LF)-\frac{LF}{2}$	$\frac{1}{LF}\ln\frac{1}{1-LF}$
unsuccessful search	$\frac{1}{2}\left(1+\frac{1}{(1-LF)^2}\right)$	$\frac{1}{1-LF}-LF-\ln(1-LF)$	$\frac{1}{1-LF}$

where, Loading factor $LF = \frac{\text{no. of elements in table}}{\text{table size.}}$

II) Chaining (collision Resolution method).

Keys do not have to be stored in the table itself.

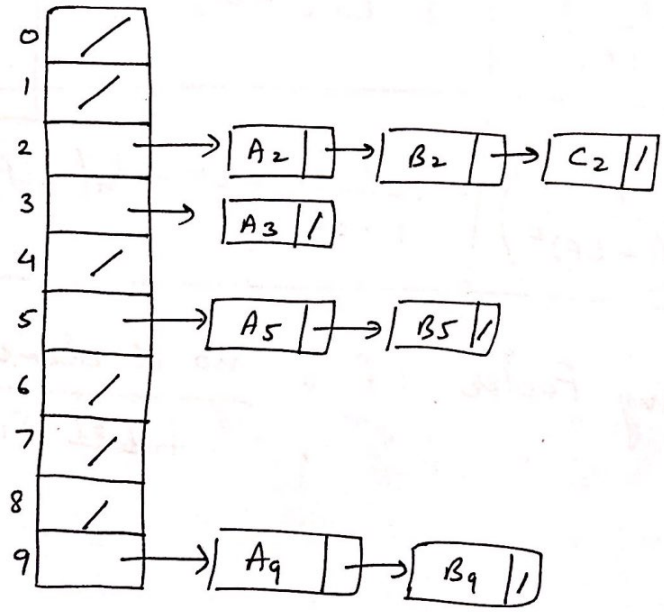
Each position of table is associated with a linked list or chain of structures whose 'info' field store keys or references to keys.

The method is called separate chaining & table of references is called a scatter table.

In this method, the table can never overflow because linked list are extended only upon arrival of new keys

The table stores only pointers, & each node requires one pointer field. Therefore, for n keys, n + TSize pointers are needed.

eg. :- Insert $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$.



→ as a whole, join together

A version of chaining called 'coalesced hashing' (or coalesced chaining) combines linear probing with chaining.

→ to come together to form larger groups

In this method, the first available position is found for a key colliding with another key (any probing method) & index of this position is stored with the key already in the table.

Table stores one key and a pointer pointing to element that should come at that place but kept somewhere else because it was already full.

Each position 'pos' of the table stores an object with two members: 'info' for key & 'next' with index of next key that is hashed to 'pos'.

This method requires TSize. sizeof(next) more space for the table in addition to the space required for the keys.

eg:- Insert: A_5, A_2, A_3

0		/
1		/
2	A_2	/
3	A_3	/
4		/
5	A_5	/
6		/
7		/
8		/
9		/

B_5, A_9, B_2

0		/
1		/
2	A_2	
3	A_3	/
4		/
5	A_5	
6		/
7	B_2	/
8	A_9	/
9	B_5	

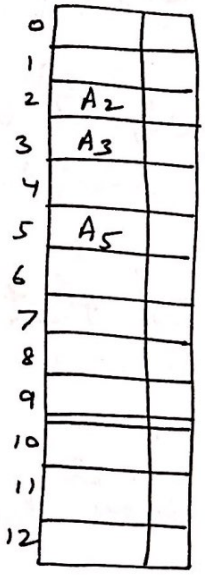
B_9, C_2

0		/
1		/
2	A_2	
3	A_3	/
4	C_2	/
5	A_5	
6	B_9	/
7	B_2	
8	A_9	
9	B_5	

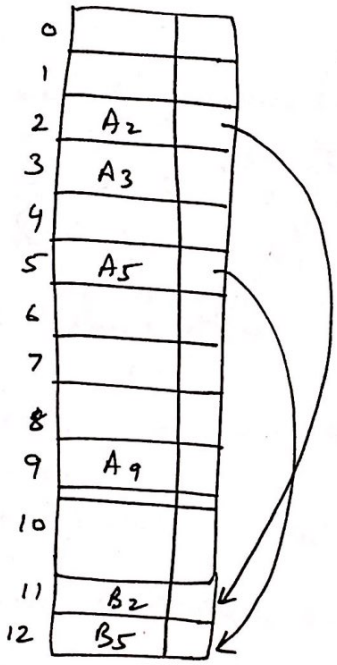
(Not actual Linear Probing is used).

An overflow area known as 'cellar' can be allocated to store keys for which there is no space in the table. This area should be located dynamically if implemented as a list of arrays.

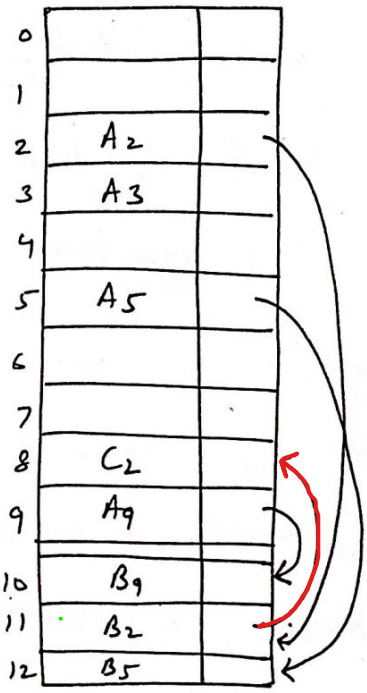
eg:- Insert A_5, A_2, A_3



B_5, A_9, B_2



B_9, C_2



Non colliding keys are stored in their home positions.

colliding keys are put in the last available slot of the cellar & added to the list starting from their home positions.

when cellar is full, an available cell is taken from the table.

(III) Bucket Addressing (collision Resolution methods).

In this method, store colliding elements in the same position in the table.

This can be achieved by associating a bucket with each address. A Bucket is a block of space large enough to store multiple items.

Now, if Bucket is already full, then item hashed to it has to be stored somewhere else.

They can be stored in next bucket if it has available slot (using linear probing.) OR in some other bucket (using Quadratic Probing).

eg:- Insert: $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$

0		
1		
2	A_2	B_2
3	A_3	C_2
4		
5	A_5	B_5
6		
7		
8		
9	A_9	B_9

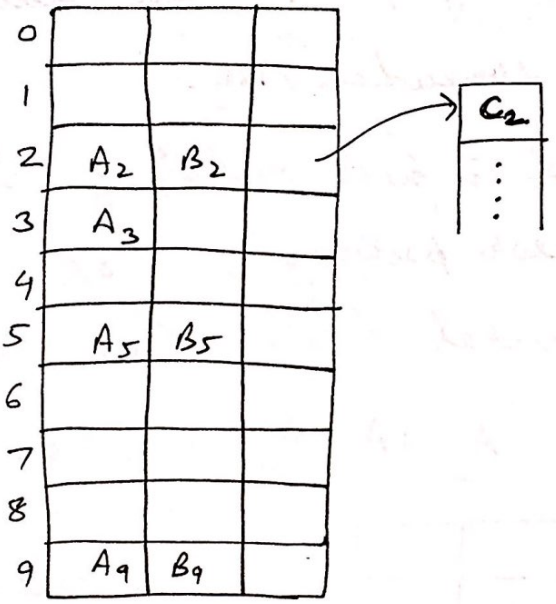
collision Resolution with buckets & linear Probing.

The colliding items can also be stored in an overflow area.

Here each Bucket includes a field that indicates whether the search should be continued or not.

This can be a Yes/No marker.

In conjunction with chaining, this marker can be the number indicating the position in which the beginning of the linked list associated with this bucket can be found in the overflow area.

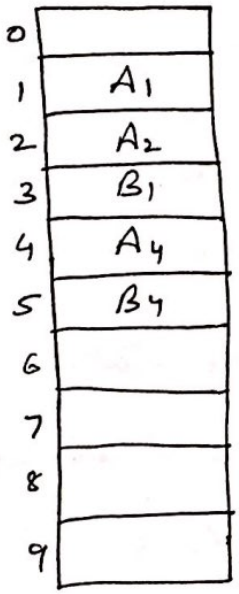


10.3 Deletion

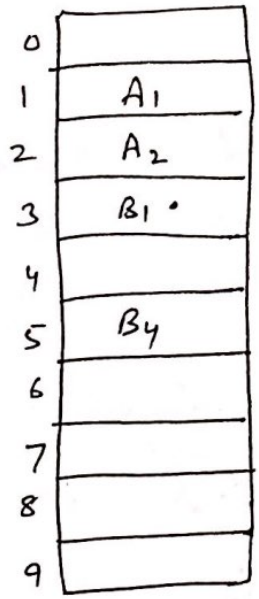
In chaining method, deleting an element leads to the deletion of a node from a linked list holding the element.

eg:- The keys have been entered in the following order:

Insert: A_1, A_4, A_2, B_4, B_1

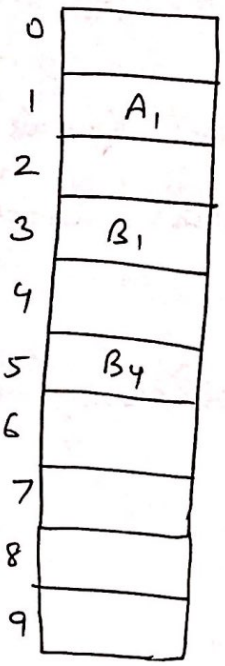


Now, Delete A_4. After A_4 is deleted position '4' is freed.



Problem:- If we try to find B_4, we first check position 4. But this position is now empty, so we may conclude that B_4 is not in the table which is not correct.

Now, Delete A_2.



A_2 is deleted & Position 2 is marked empty.

Then search for B_1 is unsuccessful, because if we are using linear probing, the search terminates at posibon 2.

Solution:-

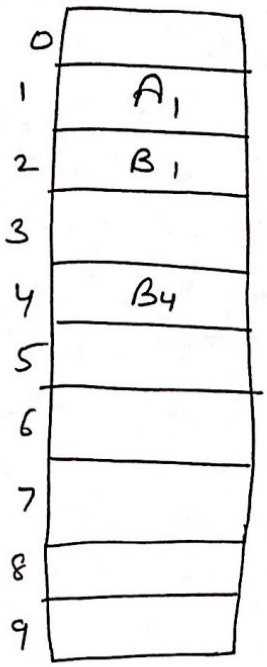
- If we leave deleted keys in the table with markers indicating that they are not valid elements of the table, any search for an element does not terminate prematurely. When a new key is inserted, it overwrites the non-valid key & mark it as valid key.

Problem:- If table has large no. of deletions & few insertions then table becomes overloaded with deleted records.

Thus, the table should be purged after a certain no. of deletions by moving undeleted elements to the cells occupied by deleted elements.

cells with deleted elements that are not overwritten

by this procedure are marked as free. ie contd. from previous diagram, we get, resulting diagram in which B_1 is moved one position up. Also B_4 is moved up but only till Position 4.



10.4. Perfect Hash functions (A. Drozdek). Ref-1.

Perfect Hash functions :- is one that maps the set of actual key values to the table without any collisions. i.e. a f^n that hashes key values at first attempt.

minimal Perfect Hash function :- A function that requires only as many cells in the table as the number of data so that no empty cell remains after hashing is completed. That is, table has only as many slots as there are key values to be hashed.

If the set of keys is known in advance, it is possible to construct a specialized hash function that is perfect. (perhaps even minimal perfect).

Algorithms for constructing perfect hash function tend to be tedious, but a number are known.

Cichelli's Method.

This method is used primarily when it is necessary to hash a relatively small collection of keys.
for eg:- set of reserved words for a programming language.

The basic formula is:

$$h(word) = [length(word) + g(firstletter(word)) + g(lastletter(word))] \bmod TSize.$$

where,
$g()$ is constructed using Cichelli's algorithm so that $h()$ will return a different hash value for each word in the set.

The values assigned by $g()$ to particular letters do not have to be unique.

The algorithm has three phases:-

Phase (I) Computation of the letter frequencies in the words.

Phase (II) ordering the words

Phase (III) searching.

Example:

Hash the following words given :- (Muses (goddesses in Greek mythology).

TSize = 9.
g-max = 4.

- Calliope
- Clio
- Erato
- Euterpe
- Melpomene
- Polyhymnia
- Terpsichore
- Thalia
- Urania

Phase I. - Count the occurrences of the first and last letters of each word. (disregard cases).

C - 2
E - 6
O - 2
M - 1
P - 1
A - 3
T - 2
U - 1

Phase-II Order the list from highest to lowest frequency.

E - 6
A - 3
C - 2
O - 2
T - 2
M - 1
P - 1
U - 1

Now, we'll calculate the value for each word using the first and last letters.

Calliope → 2 + 6 = 8] (2)
Clio → 2 + 2 = 4] (5)
Erato → 6 + 2 = 8] (2)
Euterpe → 6 + 6 = 12] (1)
Melpomene → 1 + 6 = 7] (3)
Polyhymnia → 1 + 3 = 4] (5)
Terpsichore → 2 + 6 = 8] (2)
Thelia → 2 + 3 = 5] (4)
Urania → 1 + 3 = 4.] (5)

C - 2
E - 6
O - 2
M - 1
P - 1
A - 3
T - 2
U - 1

Order the words by these values.

Euterpe
Calliope
Erato
Terpsichore
Melpomene
Thelia
Clio
Polyhymnia
Urania

$h(word) = [length(word) + g(first\ letter(word)) + g(last\ letter(word))] \bmod TSize$

Phase - III

First word :- Euterpe

length - 7

g(first letter = E) = 0

g(last letter = E) = 0

7 mod 9 = 7

g- values.

E - 0
A - 0
C - 0
O - 0
T - 0
M - 0
P - 0
U - 0

Second word :-

Calliope

length = 8

g(first letter = c) = 0

g(last letter = e) = 0

8 mod 9 = 8

Table.

Indexes	words.
0	Melpomene
1	
2	Terpsichore
3	
4	
5	Erato
6	
7	Euterpe
8	Calliope
9	

Third word :- Erato

length - 5

g(first letter = E) = 0

g(last letter = o) = 0

5 mod 9 = 5

fourth word :- Terpsichore

length = 11

g(first letter = T) = 0

g(last letter = e) = 0

11 mod 9 = 2

fifth word :-

Melpomene

length = 9

g(first letter = M) = 0

g(last letter = e) = 0

9 mod 9 = 0

sixth word :- Thalia

length = 6

g(first letter = t) = 0

g(last letter = a) = 0

6 mod 9 = 6

seventh word :- Clio

length = 4

g(first letter = c) = 0

g(last letter = o) = 0

4 mod 9 = 4

Eighth word :- Polyhymnia

length = 10

~~length~~ g(first letter = p) = 0

g(last letter = a) = 0

10 mod 9 = 1

Ninth word :- Urania.

length = 6

g(first letter = u) = 0

g(last letter = a) = 0

6 mod 9 = 6 (clash)

Hash

g-values

E - 0
A - 0
C - 0
O - 0
T - 0
M - 0
P - 0
U - ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4.
(reaches max)

Table

Indexes	words
0	Melpomene
1	Polyhymnia
2	Terpsichore
3	
4	Clio
5	Erato
6	Thalia
7	Euterpe
8	Calliope
~~9~~	

⇒ To fix this, we will start increasing the g-value of first letter (u)

length = 6

g(first letter = u) = 1

g(last letter = a) = 0

7 mod 9 = 7

(Hash clash)

length = 6

g(first letter = u) = 2

g(last letter = a) = 0

8 mod 9 = 8

(Hash clash)

length = 6

g(first letter = u) = 3

g(last letter = a) = 0

9 mod 9 = 0

(Hash clash)

length = 6
g(first Letter = u) = 4
g(last letter = a) = 0

10 mod 9 = 1 (Hash clash)

At this point, we have maxed out our g-values, but we still haven't found a solution.

Thus reset g-values and step back to 8th words ie.

Polyhymnia.

length = 10
g(first letter = P) = 1
g(last letter = a) = 0

11 mod 9 = 2
(Hash clash)

length = 10
g(first letter = P) = 2
g(last letter = a) = 0

12 mod 9 = 3

Back to 9th word - Urania

length = 6
g(first letter = u) = 0 (0 to 3 causes clash)
g(last letter = a) = 0

6 mod 9 = 6
(Hash clash)

g-values

E - 0
A - 0
C - 0
O - 0
T - 0
M - 0
P - ø ✗ 2
U - ø ✗ ✗ ✗ 4.

Table

Indexes	words
0	Melponene
1	
2	Terpsichore
3	Polyhymnia
4	Clio
5	Erato
6	Thalia
7	Euterpe
8	Calliope

Urania.

length = 6

g(firstLetter = u) = 4

g(lastLetter = a) = 0

10 mod 9 = 1.

Table

Index	words
0	Melpomene
1	Urania.
2	Terpsichore
3	Polyhymnia
4	Clio
5	Erato
6	Thelia
7	Euterpe
8	calliope.

Drawbacks:-

The algorithm is inapplicable to a large number of words.

The searching process is exponential because it uses an exhaustive search.

It does not gaurantee that a perfect hash f^n can be found.

Advantages

The program needs to be run only once.

The resulting hash f^n can be incorporated into another program.

Extensions / Modification in Cichelli's Algorithm

g-value of second to last letter is added to calculate 'h'.

Partitioning the body of data into separate buckets for which minimum perfect hash f^n is found.

$$h(word) = bucket_{gr(word)} + h_{gr(word)}(word)$$

⇒ Problem is difficult to find grouping f^n.