Searching

Searching refers to finding a given data item in a set, list or array. It is not necessary that the data item we are
searching for, must be present in the list.

- If the searched item is present in the list, then the searching algorithm (or program) can find that data item, in which case we say that the search is **successful** and return the location of the data item in the list.
- But if the searched item is not present in the list, then it cannot be found and we say that the search is **unsuccessful**.

Linear search or sequential search:

In sequential search we search for a key in a sequential manner, accessing each element only once from beginning of the data structure.

- In a sequential search, the table that stores the elements is searched successively, and the key comparison determines whether an element has been found.
- If element is found then search is said to be successful else unsuccessful.
- These include arrays, lists a sequential file.

The Java code for the Linear search

```
class searching
{
        public static boolean Linearsearch(int [] a, int n)
        {
                int i, t=0;
                for (i=0; i<a.length; i++)
                {
                        if(a[i]==n)
                        {
                                t = 1;
                                break;

                        }

                }
                if(t==1)
                        return true;
                else
                        return false;
        }
        public static void main (String [] args)
        {
                int arr[] = {1,20,30,40,10,22};
                System.out.println(Linearsearch (arr, 22));
        }
}
```

Complexity of Linear Search

In case element is found within the very first comparison, it will be the best case for searching (least probable) and time complexity will be O(1). In the worst-case element may be at the end of the list so that number of comparisons will be N. Time complexity in this case will be O(N). On an average case the number of comparisons will be approximately (N+1)/2. But still complexity will be O(N). This result is obtained from the probabilistic theory.

Binary Search:

In binary search over the array of N elements, we first find out the middle position of the array. We then compare the middle element of the array with the data to be searched. If data is equal to the middle element, it is found. If data element is less than middle element, it resides into the lower half of the array else it resides into the upper half of the array. The process is repeated for the other half of the array (lower and upper) and finally the element will be found as the middle element or search will be unsuccessful.

Complexity of Binary Search

It is clearly visible from the code for binary search that each comparison divides the sub list into two halves. Hence the complexity of binary search will be O(log N). Due to this complexity, binary search is considered as one of the most efficient searching techniques. But there are some limitations of binary search. The first one is that original list must be sorted. In case list is not sorted, we need to apply some sorting algorithm to sort the list first.

The following Java program shows binarysearch:

```
class BSearch
 {
        public static oolean BinarySearch(int [] arr, int val)
        {
                int
                low,high,mid;
                oolean   b   =
                false; low = 0;
                high =
                arr.length-1;
                while(low<=high
                )
                {
                        mid = (low+high)/2;
                        if(arr[mid]==val)
                        {
                                b =
                                true;
                                break;
                         }
                         if(arr[mid]>val)
                                high = mid-
                          1; else
                                 low = mid+1;
                }
                return b;
        }
        public static void main (String [] args)
        {
                int arr[] = {1,2,4,5,12,15,20,23,25,27,31};
                if(BinarySearch(arr,12))
                        System.out.println ("The element is found");
                else
                System.out.println ("Element is not found");
        }

 }
```

Hash

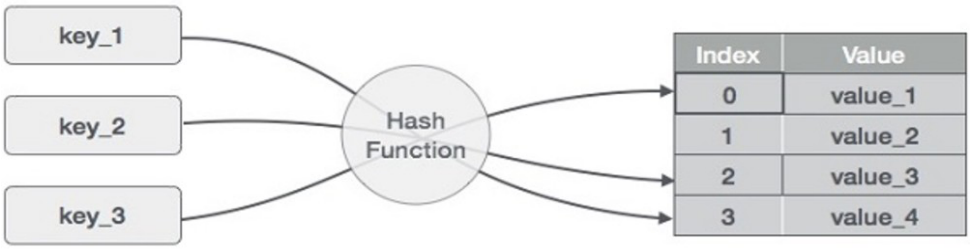
Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

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

Hashing

- ***Hashing*** is an efficient searching technique in which key is placed in direct accessible address for rapid search.
- ***Hashing*** provides direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a ***hashing function*** say h which maps the key with its corresponding key address or location or index.
- ***Hash function*** is a function that transforms a key into its corresponding table index.
- The data values are stored into a ***hash table*** by the help of a ***hash function.***
- A ***hash table*** is a data structure (generally implemented as an array) where key values are stored after applying the ***hash function***.
- Each key value occupies a unique location in the ***hash table***.

Consider an example of hash table of size 20, and the following items are to be stored.



Key	Hash	Array Index
1	1 % 20 = 1	1
2	2 % 20 = 2	2
42	42 % 20 = 2	2
4	4 % 20 = 4	4
12	12 % 20 = 12	12

Hash Functions

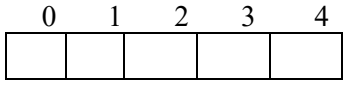
The function *h* is said to be hash function such that *h* can transform a particular key *K* into an index in the table for storing items of the same type as *K*.
The number of hash functions that can be used to assign positions to n items in a table of m positions ($n \leq m$) is equal to m^n. The number of perfect hash functions is the same as the number of different placement of these items in the table and is equal to m!/(m-n)!. For example, for 50 elements and a 100 cell array, there are $100^{50} = 10^{100}$ has functions, out of which "only" 10^{94} (one in one million) are perfect. Most of these functions are too unwieldy for practical applications and cannot be expressed with a formula.

Types of Hash Functions:

1. Division

A hash function must guarantee that the number it returns is valid index to one of the table cells. The simplest way to accomplish this is to use division modulo TSize = *sizeof (table),* as in *h(K) = K* mod *TSize*, if *K* is a number. It is best if *TSize* is a prime number; otherwise, *h (K)* = (*K* mod *p*) mod *TSize* for some prime *p>TSize* can be used. However, nonprime divisors may work equally well as prime divisors provided that they do not have prime factors less than 20.

Example: Suppose 1, 13, 37, 29 are the keys to be stored in a hash table with size 5 using Division.
Solution:
here, ***Tsize = 5*** and hash function is ***h(k) = k mod Tsize***
h(1) = 1 % 5 = 1
h(13) = 13 % 5 = 3
h(37) =37 % 5 = 2
h(29) = 29 % 5 = 4



2. Folding

In this method, the key is divided into several parts (which conveys the true meaning of the word hash). These parts are combined or folded together and are often transformed in a certain way to create the target address. There are two types of folding: Shift folding and boundary folding.

The key is divided into several parts and these parts are then processed using a simple operation such as addition to combine them in a certain way. For example, a social security number (SSN) 123-45-6789 can be divided into three parts, 123, 456, 789 and then these parts can be added.

a) Shift folding:

For example, a social security number (SSN) 123-45-6789 can be divided into three parts, 123, 456, 789 and then these parts can be added i.e., 123+456+789 = 1380. The resulting number 1368 is divided modulo TSize or, if the size of the table is 1000(i.e., 1380%1000 = 138), the first three digits can be used for the address. To be sure, the division can be done in many different ways.

Example: Consider a hash table with 100 slots i.e., m=100 and key values ***K*** be: 7325, 76321, 1623, 7613.

k	*parts*	Sum of parts	h(k)
7325	73,25	98	98
76321	76,32,1	109%100=9	09
1623	16,23	39	39
7613	76,13	89	89

b) Boundary folding:

Go one step further and reverse every other piece before the addition.

k	*parts*	Sum of parts	h(k)
7325	73,52	125%100=25	25
76321	76,23,1	100%100=0	0
1623	16,32	48	48
7613	76,31	107%100=7	7

3. Mid Square Function

In the mid square method, the key is squared and the middle or mid part of the result is used as the address. The mid part is computed on the basis of table size. For Decimal number use 10^x = Tsize, where x is the digits contained mid part. If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding. In a mid-square hash function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys.

- For example, if the key is 3121 then 3121^2 = 9740641 and for the 1000 cell table, h (3121) = 406 which is the middle part of 31212.
- In practice, it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key. If we assume that the size of the table is 1024, then, in this example, the binary representation of 3121^2 is the bit string 1001010***0101100001***01100001, with the middle part shown in bold italics. This middle part, the binary number 01011000010 is equal to 322. This part can be easily extracted by using a mask and shift operation.

4. Extraction

In the extraction method, only a part of the key is used to compute the address. For the social security number 123-456-789, this method might use:

- the first four digits, 1234
- the last four 6789
- the first two combined with last two, 1289 or some other combination.

Each time only a portion of the key is used. If this portion is carefully chosen, it can be sufficient for hashing, provided the omitted portion distinguishes the keys only in significant way. For example, in some university settings, all international students' ID numbers start with 999. Therefore, the first three digits can be safely omitted in hash function that uses student IDs for computing table positions.

Hash collision and Collision Resolution Techniques:

Hash collision is the condition in which multiple elements are hashed to the same index. The systematic method for placing colliding key at a unique index in the hash table is called collision resolution.

There are following hash collision resolution techniques are available:

a) Open Addressing
   (i) Linear Probing
   (ii) Quadratic Probing
   (iii) Double Hashing
b) Rehashing
c) Chaining
d) Bucket Addressing

a) Open Addressing

In the open addressing 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 repeatedly until table is full.

The function p is a probing function, i is a probe, and norm is a normalization function, most likely, division modulo the size of the table.

(i) Linear Probing:

The simplest method is the **linear probing,** for which p(i) = i, and for the ith probe, the position to be tried is ***(h(K)+i) mod TSize***. In linear probing, the position in which a key can be tried is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found.

In case if key has same hash address then it will find the next empty position in the hash table we take the hash table as circular array. If table size in n then after n-1 position it will search form zeroth position in the array.

Example: Elements are: 29, 18, 23, 10, 36, 26, 46, 43, 2 and Table size(Tsize) is 11.
h(29) = 29 % 11 = 7
h(18) = 18 % 11 = 7, collision occurs since 29 is already at position 7. So, ***(h(k)+i) % Tsize*** = (7+1)%11 = 8.
h(23) = 23 % 11 = 1
h(10) = 10 % 11 = 10
h(36) = 36 % 11 = 3
h(26) = 26 % 11 = 4
h(46) = 46 % 11 = 2
h(43) = 43 % 11 = 10, collision since 10 is already at index 10. So, ***(h(k)+i) % Tsize*** = (10+1) %11 = 0
h(2) = 2 % 11 = 2, collision occurs since 46 is at index 2. So, ***(h(k)+i) % Tsize*** = (2+1) %11 = 3.
Collision occurs again since 36 is already at index 3.
now try i =2
(h(k)+i) % Tsize = (2+2) %11 = 4.
Collision occurs again since 26 is already at index 4.
now try i =3
(h(k)+i) % Tsize = (2+3) %11 = 5.

0	1	2	3	4	5	6	7	8	9	10
43	23	46	36	26	2		29	18		10

Example 2:

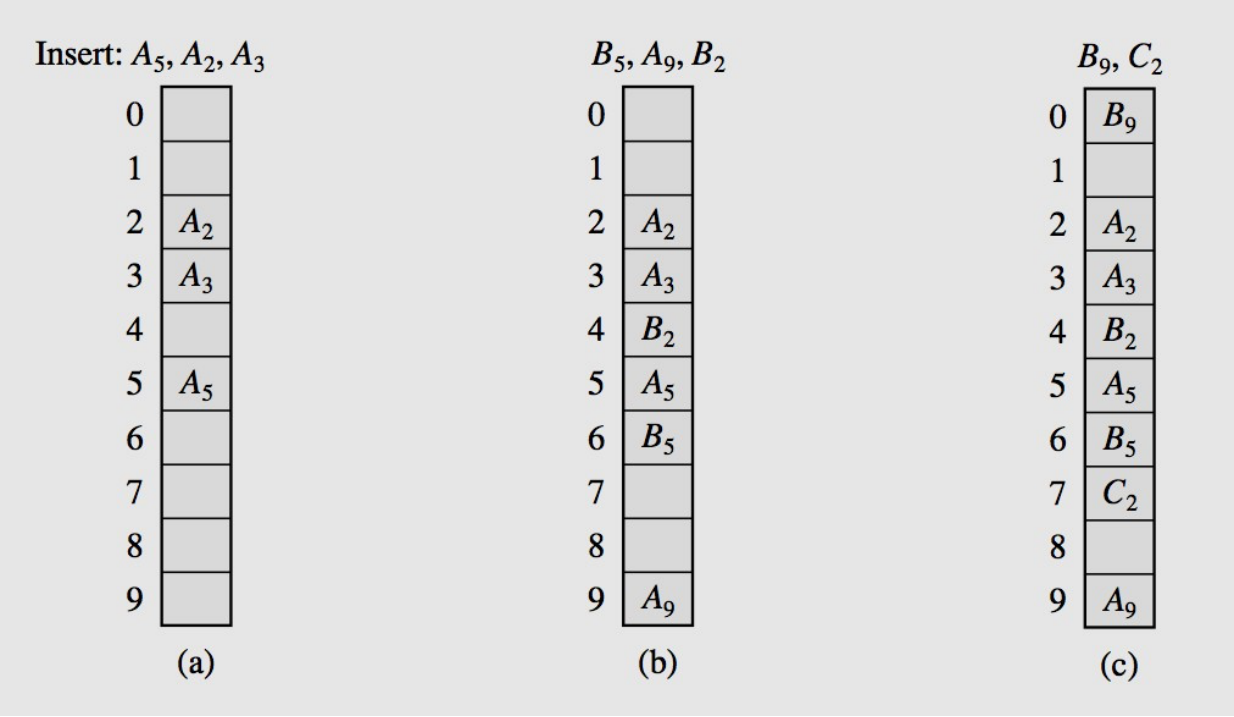


Figure: Resolving collisions with linear probing method. Subscripts indicate the home positions of the keys being hashed.

(ii) Quadratic Probing:
While collision occurs, the quadratic probing works as follows:

***(h(K)+i²) mod Tsize, (*h(K)-i²) mod Tsize** for i=1,2,,,,,,(TSize-1)/2.

Including the first attempt to hash K, this results in the sequence:
h(K), h(K)+1, h(K)+4……h(K)+(TSize-1)²/4 all divided modulo Tsize.

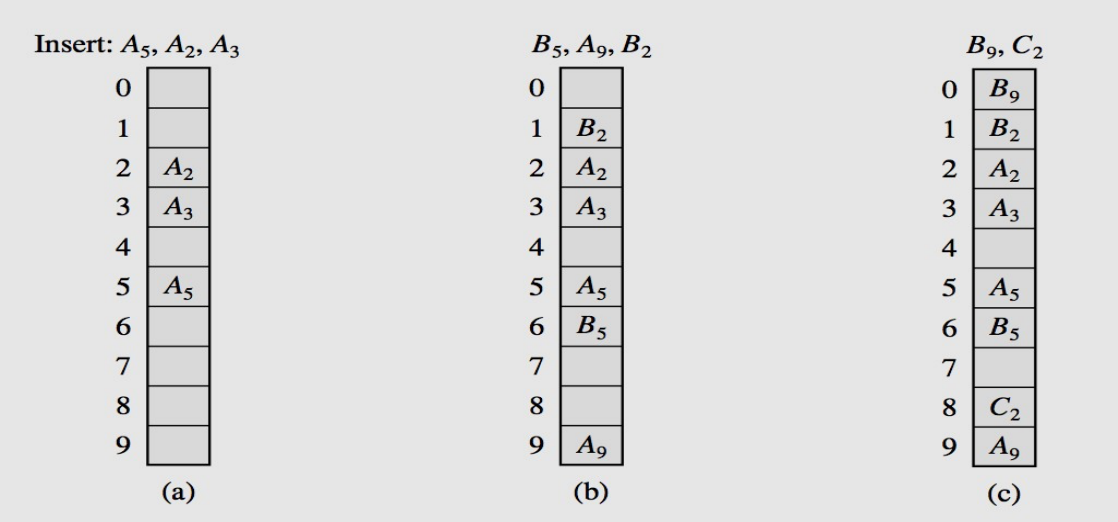
Example: Insert following numbers or elements in a hash table of size 10.
Insert 5, 2, 3:

0	1	2	3	4	5	6	7	8	9
		2	3		5				

Inserting 6, 9, 12:
Here 12 collides with 2 at index 2 since h(12) = 12%10 = 2.
(h(K)+i²) mod Tsize = (2+1²) %10 = 3. Again 3 is at index 3.
So, try at ***(h(K)-i²) mod Tsize =*** (2-1²) %10 = 1.

0	1	2	3	4	5	6	7	8	9
	12	2	3		5	6			9

Example 2:



(iii) <u>Double Hashing:</u>

Double hashing is a hash collision resolution technique that eliminates the primary clustering problem take place in a linear probing. When collision occurs then the double hashing define new hash function as follows:

$h_2(k)$ = R-(k mod R), where R is a prime number smaller than hash table size.
The colliding element can be placed in the position by using below function:

Position = (original hash value + i*h2(k)) % Tsize

Example 1: Insert 79, 13, 59, 33, 109 in a hash table of size 10 using double hashing technique.
Here, Tsize = 10.
H(79) = 79%10 = 9
H(13) = 13%10 = 3
H(59) = 59%10 = 9, Collision occurs since 79 is already at 9 index.
Now, $h_2(k) = R-(k \bmod R)$
$h_2(59) = 7-(59\%7) = 7-3 = 4$
So, position = (original hash value + $i*h_2(k)$) % Tsize [here, i=1]
*= (9+1*4)%10 = 13%10 = 3, collision occurs since 3 is already at 3 index.*
position = (original hash value + $i*h_2(k)$) % Tsize [here, i=2]
*= (9+2*4)%10 = 17%10 = 7, so 59 will be placed at 7 index.*
H(33) = 33%10 = 3, collision occurs since 13 is already at 3 index.
Now, $h_2(k) = R-(k \bmod R)$
$h_2(33) = 7-(33\%7) = 7-5 = 2$
So, position = (original hash value + $i*h_2(k)$) % Tsize [here, i=1]
*= (3+1*2)%10 = 5%10 = 5*
H(109) = 109%10 = 9, Collision occurs since 79 is already at 9 index.
Now, $h_2(k) = R-(k \bmod R)$
$h_2(109) = 7-(109\%7) = 7-4 = 3$
So, position = (original hash value + $i*h_2(k)$) % Tsize [here, i=1]
*= (9+1*3)%10 = 12%10 = 2*

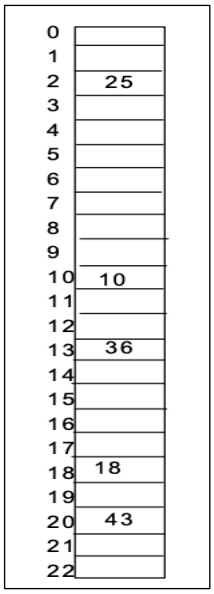
0	1	2	3	4	5	6	7	8	9
		109	12		33		59		79

b) Rehashing:
There are chances of insertion failure when hash table is full, so the solution for this particular case is to create a new hash table with the more than double size of previous hash table. Here, we will use new hash function and we will insert all the elements of the pervious hash table. So, we will scan the elements of previous hash table one by open & calculate the hash key with new hash function & insert them into new hash table.
Example: Consider the table size 11 & elements are 7, 18, 43, 10, 36, 25……..
Solution:
Applying linear probing we get
h(7) = 7%11= 7
h(18) = 18%11= 7, collision so move to next empty cell i.e. 8
h(43) = 43%11= 10
h(10) = 10%11= 10, collision so move to next empty cell i.e. 0
h(36) = 36%11= 3
h(25) = 25%11= 3, collision occur so move to next empty cell i.e. 4

0	1	2	3	4	5	6	7	8	9	10
10			**36**	**25**			**7**	**18**		**43**

.

Now, if we want to insert six more element then size will not be sufficient. In order to fit all the elements or key with in a table we take new table of size more than double with prime number. Thus, total size is 23. Applying linear
probing, we get,
h(7) = 7%23= 7
h(18) = 18%23 = 18
h(43) = 43%23 = 20
h(10) = 10%23 = 10
h(36) = 36%23= 13
h(25) = 25%23 = 2 .. …. .. ..



c) Chaining

Keys do not have to be stored in the table itself. In chaining, each position of the table is associated with a linked list or chain of structures whose info fields store keys or reference to keys. This method is called separate chaining, and a table of references is called a scatter table. In this method, the table can never overflow, because the linked lists are extended only upon the arrival of the new keys. For short linked lists, this is a very fast method, but increasing the length of these lists can significantly degrade retrieval performance. Performance can be improved by maintaining an order on all these lists so that, for unsuccessful searches, an exhaustive search is not required in most cases or by using self-organizing linked list.

This method requires additional space for maintaining references. The table stores only references, and each node requires one reference field. Therefore, for k keys, n+TSize references are needed, which for large n can be a very demanding requirement.

Example: **Inserting** 5, 2, 3, 52, 15, 29, 59, 22

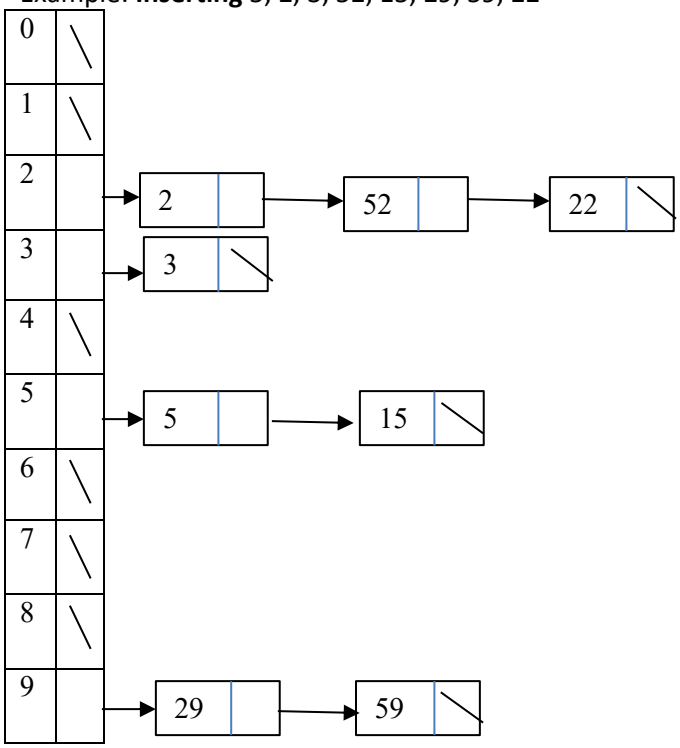


Figure: In chaining, colliding keys are put on the same linked list.

Example 2:

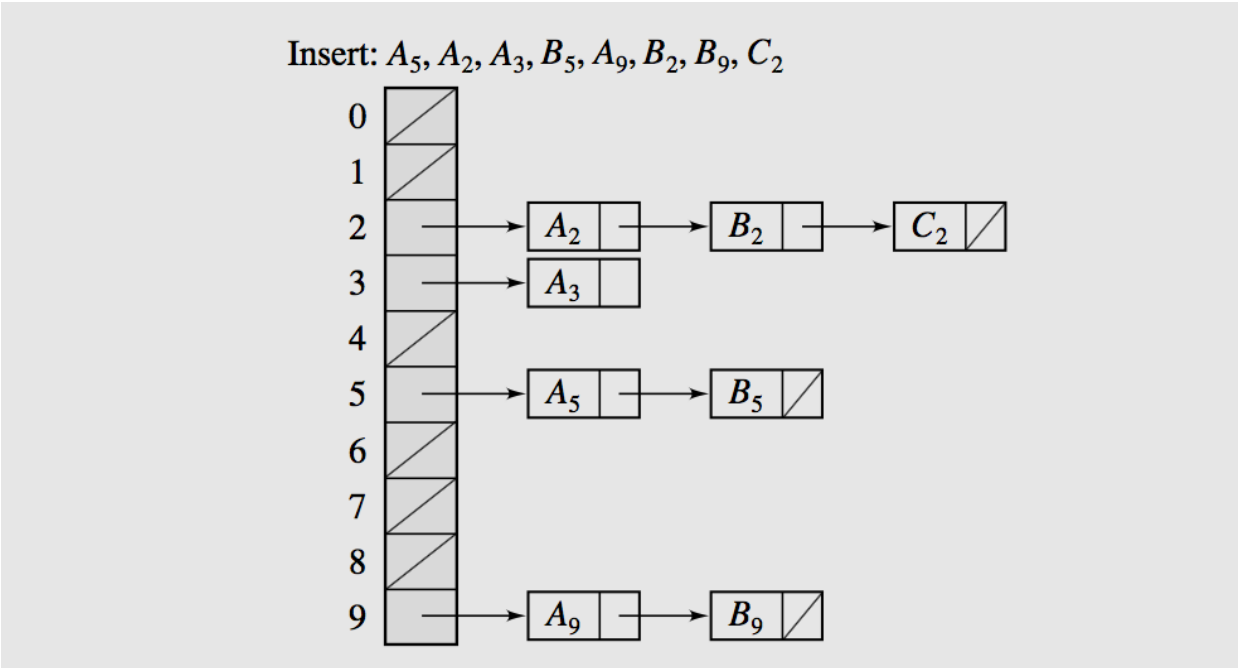


Figure: In chaining, colliding keys are put on the same linked list.

Coalesced Hashing

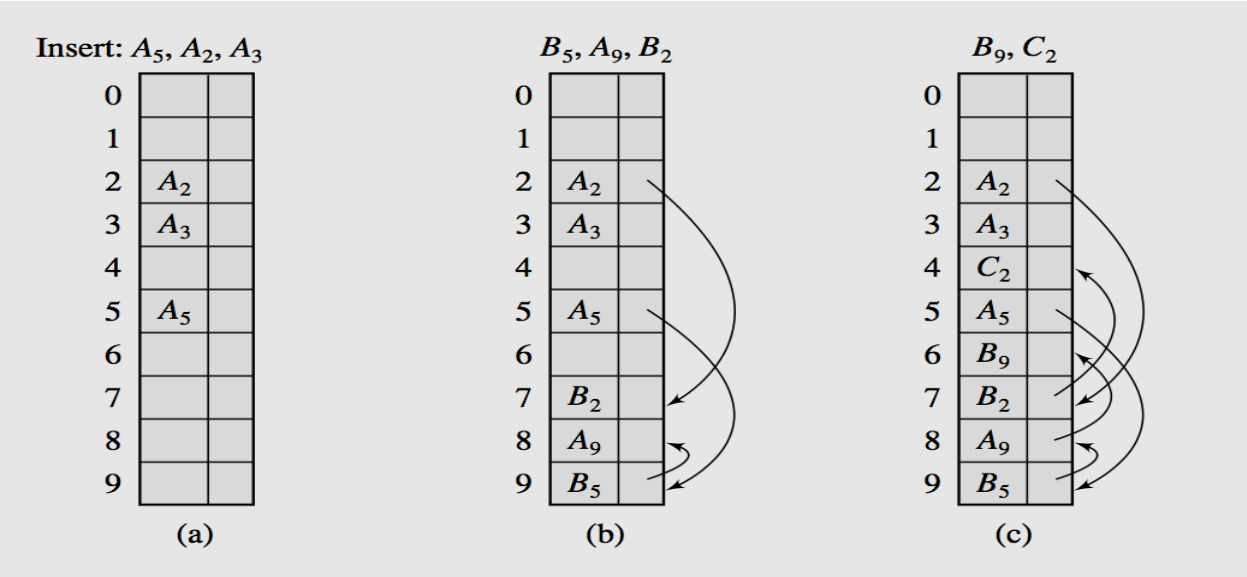


Figure: Coalesced hashing puts a colliding key in the last available position of the table.

Coalesced Hashing with Celler

- Celler is an overflow area.

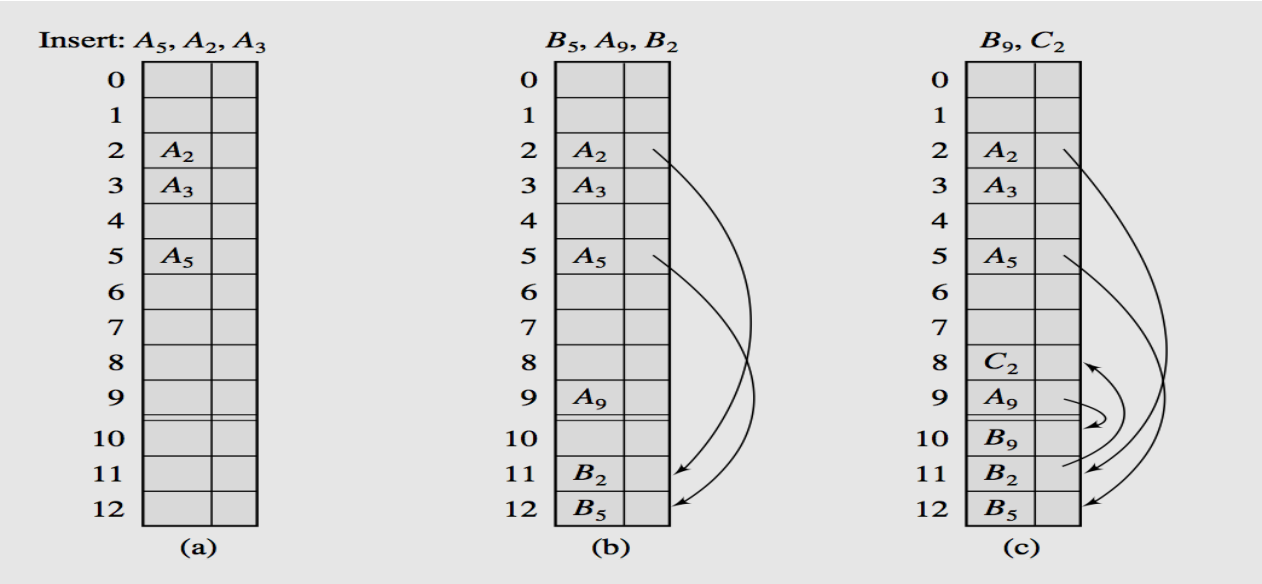


Figure: Coalesced hashing with an overflow area(celler) puts a colliding key in the last available position of the celler.

d) Bucketing Addressing

Another solution to the collision problem is to 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.

By using buckets, the problem of collisions is not totally avoided. If a bucket is already full, then an item hashed to it has to be stored somewhere else. By incorporating the open addressing approach, the colliding item can be stored in the next bucket if it has an available slot when using linear probing or it can be stored in some other bucket when, say, quadratic probing is used.

Insert: 5, 2, 3, 25, 19, 32, 9, 12, 49

49		0
		1
2	32	2
3	12	3
		4
5	25	5
		6
		7
		8
19	9	9

Figure: Collision resolution with buckets and linear probing method

Example 2:

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

Figure: Collision resolution with buckets and linear probing method

Example 3:

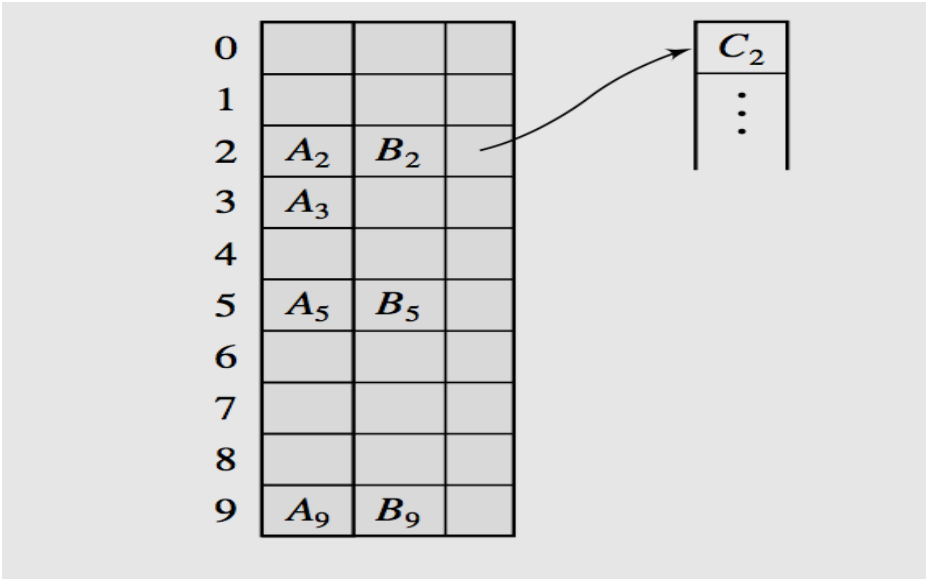


Figure: Collision resolution with buckets and an overflow area

Deletion

When we delete data from a hash table, we have to maintain this table. With a chaining method, deleting an element leads to the deletion of a node from a linked list holding the element. For other methods, a deletion operation may require a more careful treatment of collision resolution, except for the rare occurrence when a perfect hash function is used.

Consider the following table, in which the keys are stored using linear probing. The keys have been entered in the following order: A_1, A_4, A_2, B_4, B_1. After A_4 is deleted and position 4 is freed (Figure b), we try to find B_4 by first checking position 4 but this position is now empty, so we may conclude that B_4 is not in the table. The same result occurs after deleting A_2 and making cell 2 as empty (Figure c). Then, the search for B_1 is unsuccessful, because if we are using linear probing, the search terminates at position 2. The situation is the same for the other open addressing methods.

If we leave deleted keys in the table with markers indicating that they are not valid elements of the table, any subsequent search for an element does not terminate prematurely. When a new key is inserted, it overwrites a key that is only space filler. However, for a large number of deletions and a small number of additional insertions, the table becomes overloaded with deleted records, which increases the search time because the open addressing methods require testing the deleted elements. Therefore, the table should be purged after a certain number 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.