Unit 9: Hashing

Searching

Searching refers to finding a given data item in a set, list or array. In general searching is of two types: Linear search and binary search.

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.

These include arrays, lists and sequential files.

This search will take O(n) worst case

If element is found then search is said to be successful else unsuccessful. The C code for the same is give below. Here array is assumed to be unsorted.

```
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));
	}
}
```

For searching an element in the array using linear search, we search the entire array from the beginning and comparing every array element with the data to be searched. As soon as matching occurs we set the found variable to 1(which was 0 in the beginning) and break from top. Outside the loop it is checked whether found is 1 or not and depending upon result is displayed.

Complexity of Linear Search

Complexity of any searching method depends on the number of comparisons performed to search for a specific element in the array of N elements. 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.

The following Java program shows binary search:

```
class BSearch
{
        public static boolean BinarySearch(int [] arr, int val)
        {
                int low,high,mid;
                boolean 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");
        }
}
```

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 technique. 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 main operation used by all searching methods was comparison of keys. In a sequential search, the table that stores the elements is searched successively, and the key comparison determines whether an element has been found. In a binary search, the table that stores the elements is divided successively into halves to determine which cell of the table to check, and again, the key comparison determines whether an element has been found. Similarly, the decision to continue the search in a binary search tree in a particular direction is accomplished by comparing keys.

A different approached to searching calculates the position of the key in the table based on the value of the key. The value of the key is only indication of the position. When the key is known, the position in the table can be accessed directly, without making any preliminary tests, as required in a binary search or when searching a tree. This means that the search time is reduced from O(n), as in a sequential search, or from O(logn), as in binary search to 1 or at least O(1); regardless of the number of elements being searched, the run time is always the same.

We need to find a function h that can transform a particular key K, be it s string, number, record or the like, into an index in the table used for storing items of the same type as K. The function h is called a hash function. If h transforms different keys into different numbers, it is called a perfect hash function. To create a perfect hash function, which is always the goal, the table has to contain at least the same number of positions as the number of elements being hashed. But the number of elements is not always known ahead of time.

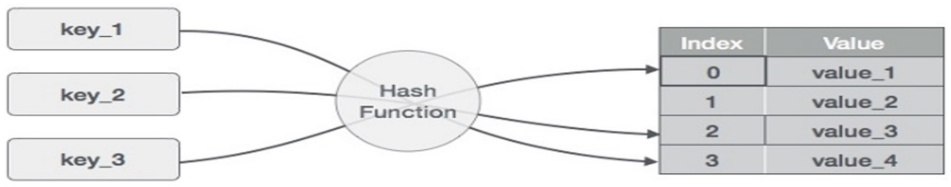
Hash

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 a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. 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

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Key	Hash	Array Index	After Linear Probing, Array Index
1	1 % 20 = 1	1	1
2	2 % 20 = 2	2	2
42	42 % 20 = 2	2	3
4	4 % 20 = 4	4	4
12	12 % 20 = 12	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≤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.

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.
(i)Shift folding: they are put underneath one another and then processed.

123-456-789 ---> 123+456+789 = 1368

The resulting number is, 1368 can be divided modulo TSize or, if the size of the table is 1000, the first three digits can be used for the address. To be sure, the division can be done in many different ways.

In the case of strings, one approach processes all characters of the string by "xor"ing them together and using the results for the address. For example, for the string "abcd", h("abcd") = "a"^"b"^"c"^"c". However, this simple method results in addresses between the numbers 0 and 127

(ii) With boundary folding, the key is seen as being written on a piece of paper that is folded on the borders between different parts of the key. In this way, every other part will be put in the reverse order.

123-456-789 ---> 123+654+789 = 1566

In both versions, the key is usually divided into even parts of some fixed size plus some remainder and then added.

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. 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***0101000010***1100001, 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.
A string would first be transformed into a number, say by folding

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, 11234; 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.
Similarly, the starting digits of the ISBN code are the same for all books published by the same publisher (e.g., 0534 for Brooks/Cole Publishing Company). Therefore, they should be excluded from the computation of addresses if a data table contains only books from one publisher.

Collision Resolution

The straightforward hashing is not without its problems, because for almost all hash functions, more than one key can be assigned to the same position. For example, if the hash function h1 applied to names returns the ASCII value of the first letter of each name (i.e., h1(name) = name[0]), then all names starting with the same letter are hashed to the same position. This problem can be solved by finding a function that distributes the names more uniformly in the table. For example, the function h2 could add the first two letters (i.e., h2(names) = name [0] + name [1]), which is better than h1. Even if all the letters are considered (i.e., h3(name)=name [0]+……name[length(name)-1), the probability of hashing different names to the same location still exist. The function h3 is the best of the three because it distributes the names most uniformly for the defined function, but it also tacitly assumes that the size of the table has been increased.

(i) 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.

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.

Another method is a quadratic probing and the resulting formula is:

P(i) = h(K)+i^2, h(K)-i^2 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)-1, h(K)+4,h(K)-4,…………………h(K)+$(TSize-1)^2/4$, h(K)- $(TSize-1)^2/4$
all divided modulo TSize.

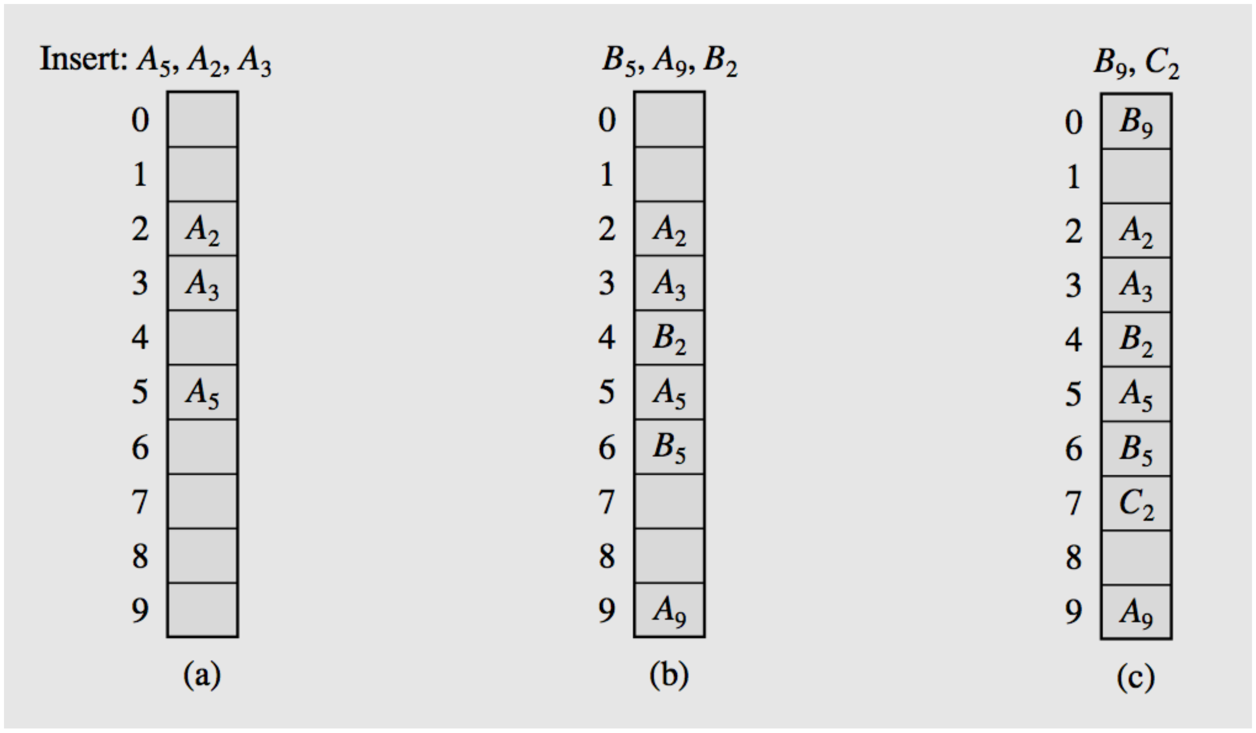


Figure: Resolving collisions with linear probing method. Subscripts indicate the home positions of the keys being hashed.

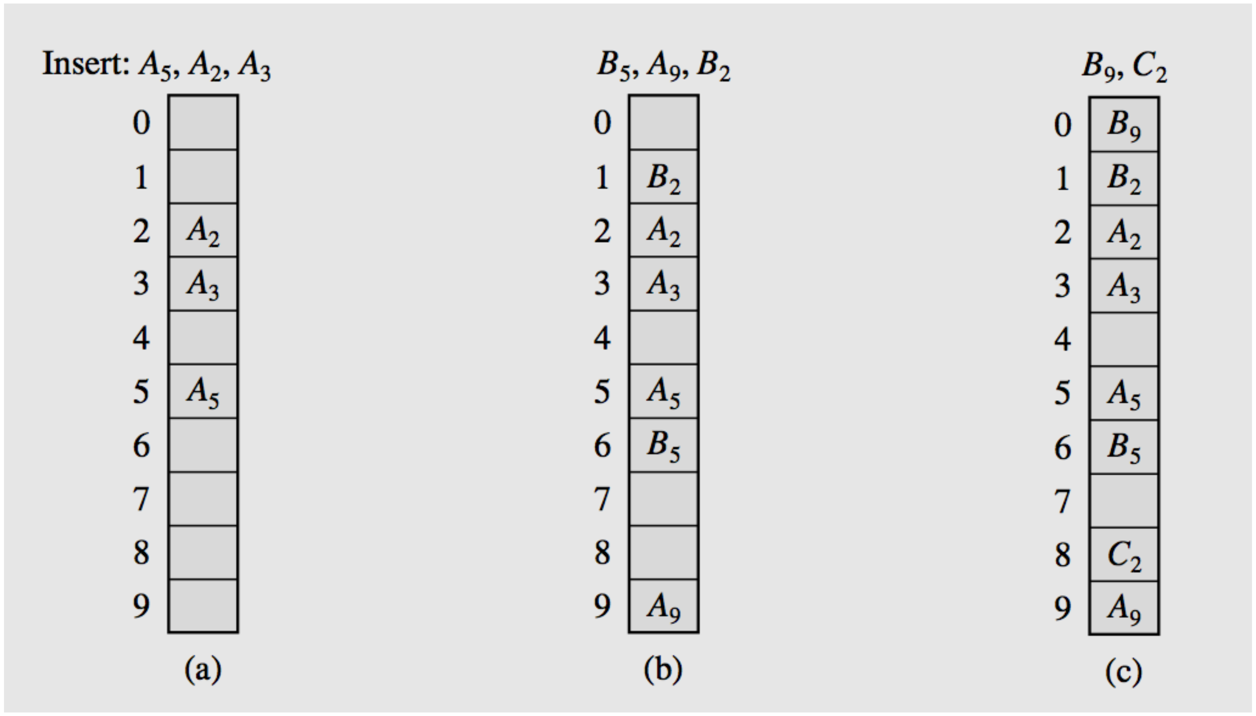


Figure: Resolving collisions with quadratic probing method

(ii) 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 n keys, n+TSize references are needed, which for large n can be a very demanding requirement.
A version of chaining called coalesced hashing (or coalesced chaining) combines linear probing with chaining. In this method, the first available position is found for a key colliding with another key, and the index of this position is stored with the key already in the table. In this way, a sequential search down the table can be avoided by directly accessing the next element on the linked list. Each position pos of the table includes two fields : an info field for a key and next field with the index of the next key that is hashed to pos. Available positions can be marked by , say, -2 in next; -1 can be used to indicate the end of a chain. This method requires TSize. (sizeof(reference)+sizeof(next)) more space for the table is addition to the space required for the keys. This is less than for chaining, but the table size limits the number of keys that can be hashed into table.
An overflow area known as a cellar can be allocated to store keys for which there are no room in the table.

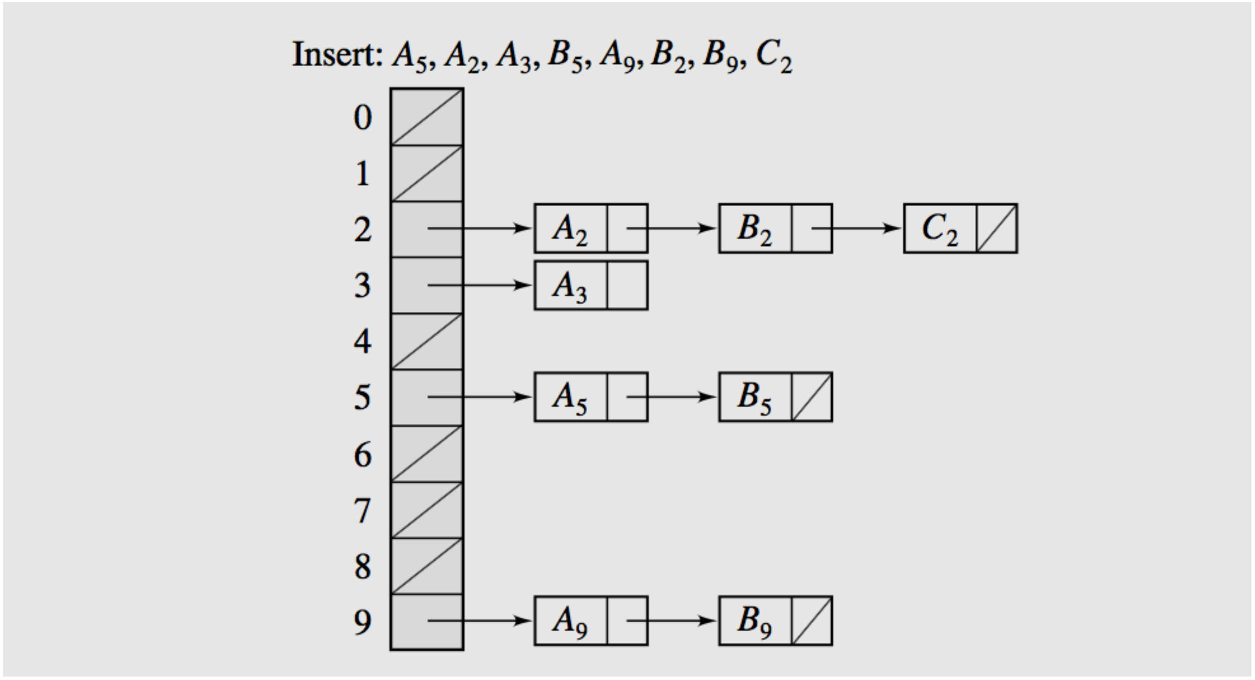


Figure: In chaining, colliding keys are put on the same linked list.

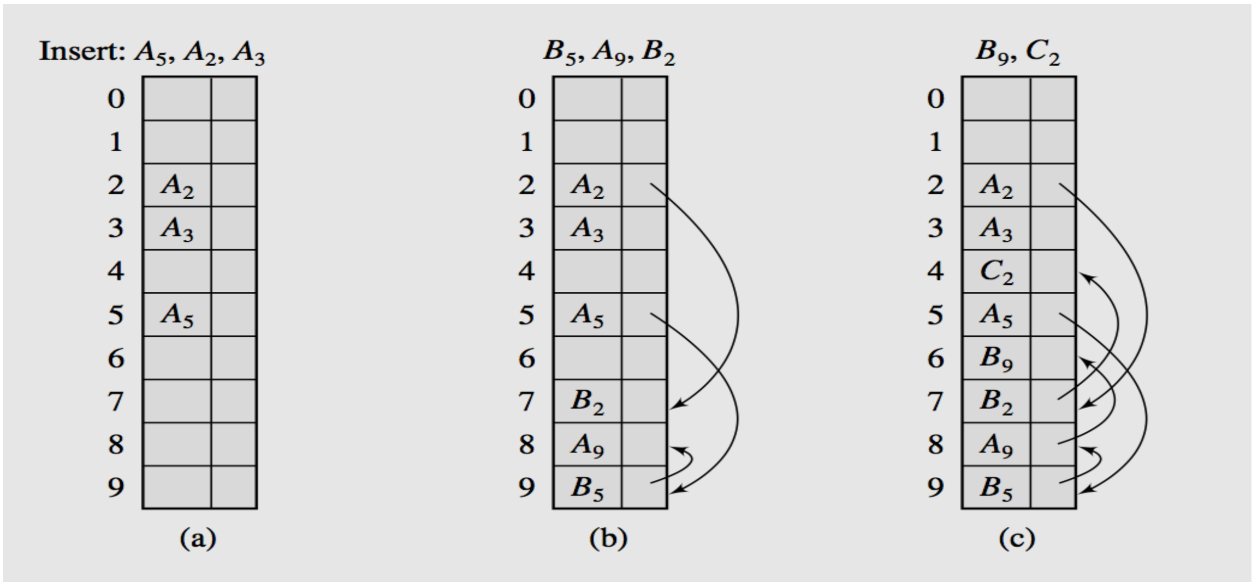


Figure: Coalesced hashing puts a colliding key in the last available position of the table.

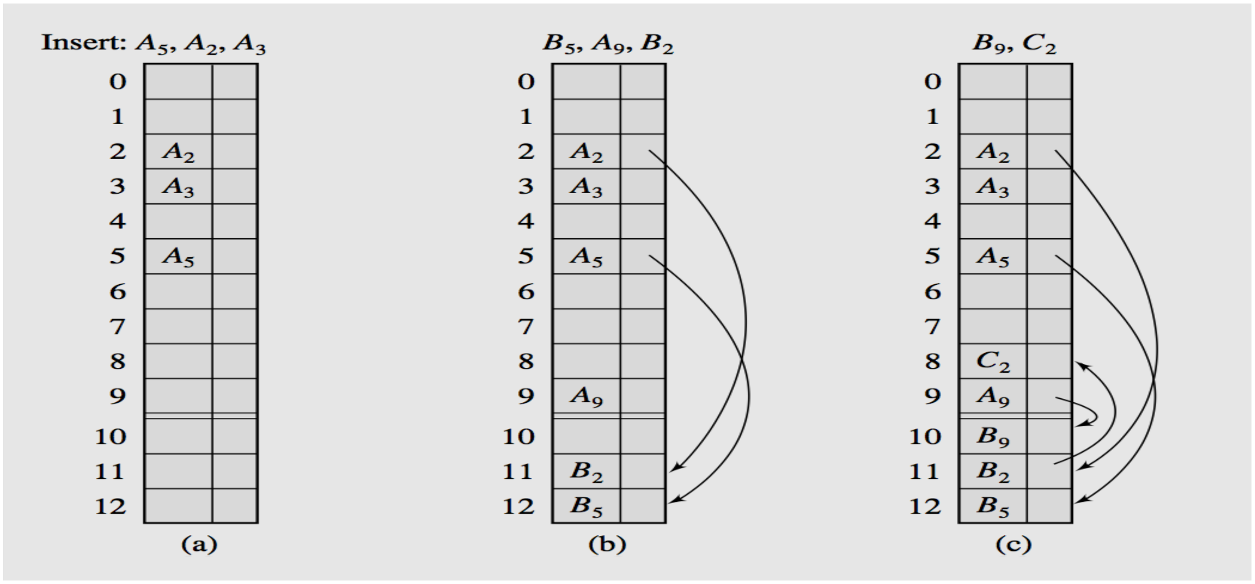


Figure: Coalesced hashing that uses a cellar.

(iii) Bucket 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.
In this implementation, hash table slots are grouped into buckets. The M slots of table are divided into B buckets, with each bucket consisting of M/B slots.

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.
The colliding items can also be stored in an overflow area. In this case, each bucket includes a field that indicates whether the search should be continued in this area or not. It can be simply a yes/no marker. In conjunction with chaining 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.

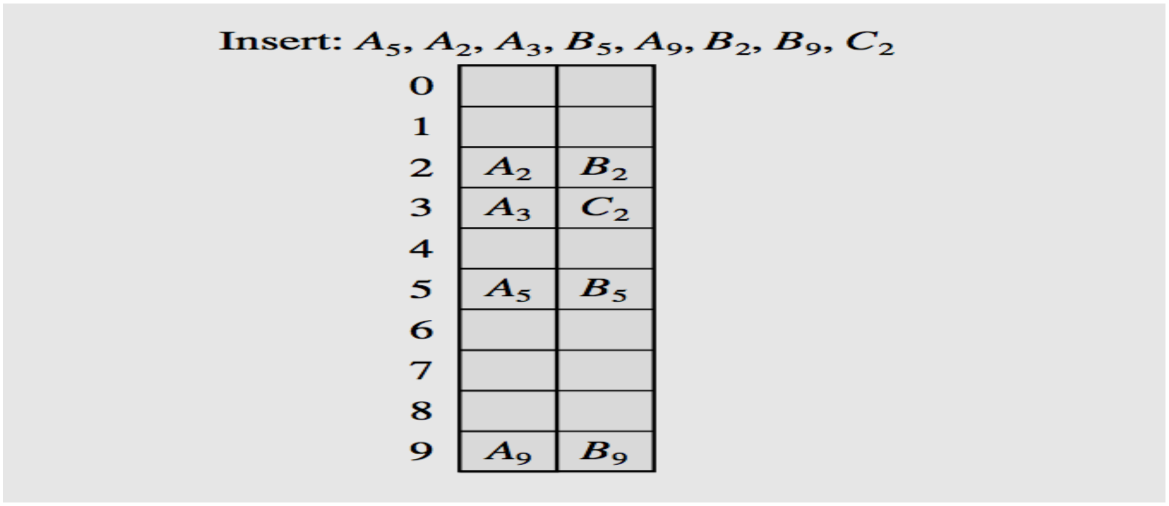


Figure: Collision Resolution with buckets and linear probing method

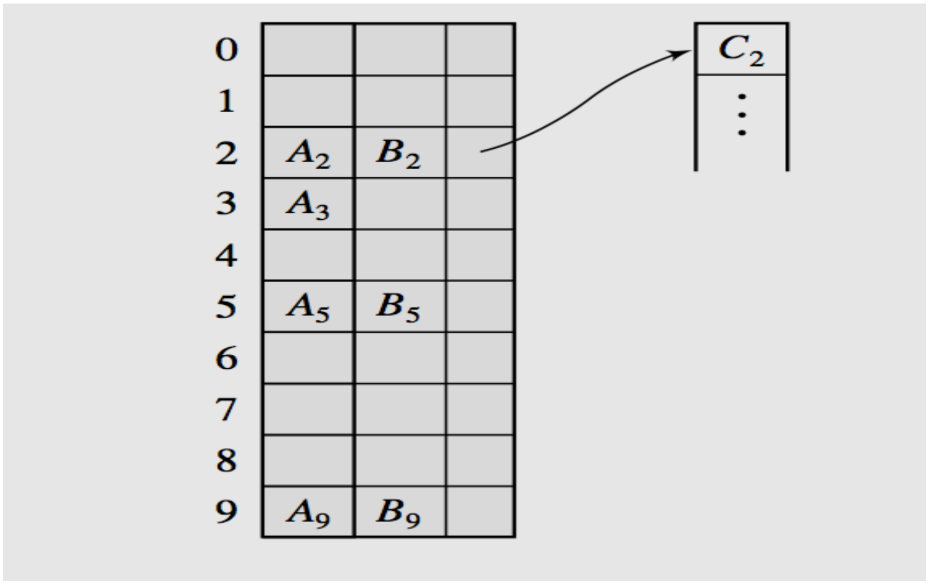


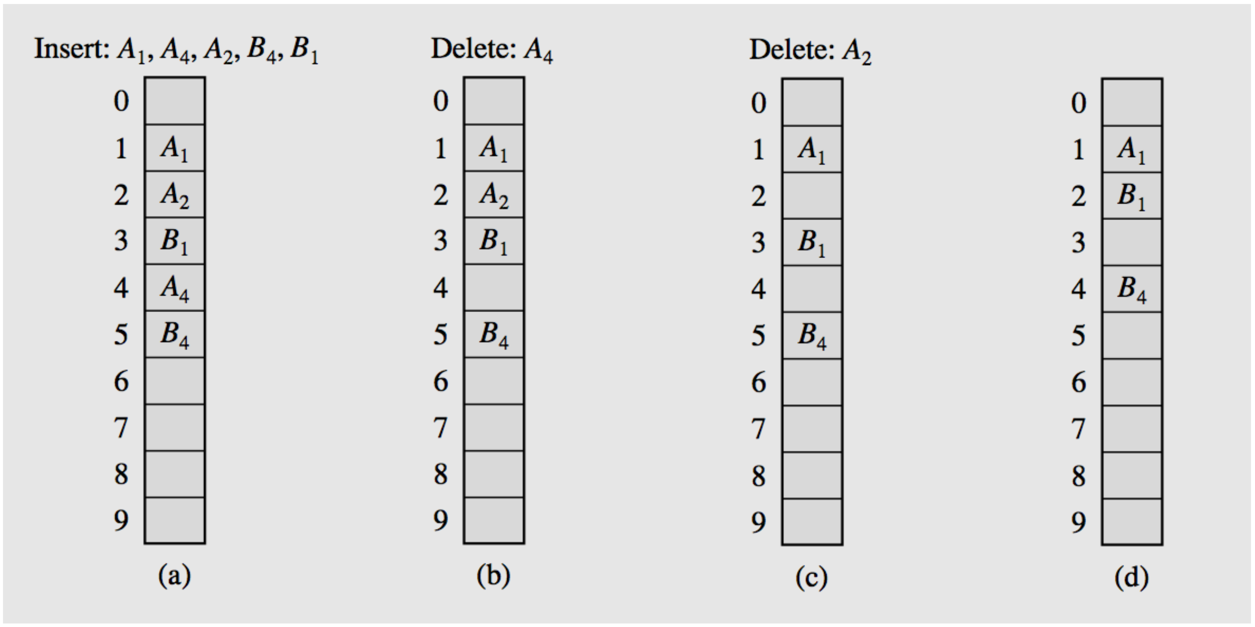
Figure: Collision Resolution with buckets and overflow area (cellar)

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.



Program for collision detection using bucket Addressing

```
class Hash{
    int data[][];
    int d1;
    int d2;
    public Hash(){
        data = new int[0][0];
    }
    public Hash(int m,int n){
        data = new int[m][n];
        d1 = m;
        d2 = n;
    }
    public void initialize(){
        for(int i=0;i<d1;i++)
            for(int j=0;j<d2;j++)
            data[i][j]=-1;
    }
    public void insert(int num)
    {
        int N = data.length;
        int i = num%N;
        int t=0, j=i;
        if(data[i][0] ==-1)
            data[i][0] = num;
        else
        {
            do
            {
                        int k=1;
                        do{
                                if(data[j][k]==-1)
                                {
                                        data[j][k] = num;
                                        t=1;
                                        break;
                                }
                                k++;
                                if(k>2)break;
                            }while(k<d1);
                            j = (j+1)%d1;
            }while(t<1);
        }
    }
 public void printAll(){
        for(int i=0;i<d1;i++){
         System.out.print("Bucket "+ i+" :");
        for(int j=0;j<d2;j++)
        {
        System.out.print(data[i][j]+" ");
        }
       System.out.println();
  }
  }
  }
public class HashingBucket {
public static void main(String[] args) {
        int arr[]={14,28,35,21,5,14,7};
        Hash h = new Hash(arr.length,3);
        h.initialize();
        for(int i=0;i<arr.length;i++)
        h.insert(arr[i]);
        h.printAll();
}
}
```

The output of the above program is:

```
Bucket 0 :14 28 35
Bucket 1 :-1 21 14
Bucket 2 :-1 7 -1
Bucket 3 :-1 -1 -1
Bucket 4 :-1 -1 -1
Bucket 5 :5 -1 -1
Bucket 6 :-1 -1 -1
```