Unit-5

Searching: List Searches- Sequential Search- Variations on Sequential Searches- Binary Search- Analyzing Search Algorithm- Hashed List Searches- Basic Concepts- Hashing Methods- Collision Resolutions- Open Addressing- Linked List Collision Resolution- Bucket Hashing.

Sequential Search- Variations on Sequential Searches:

Linear search in c programming: The following code implements linear search (Searching algorithm) which is used to find whether a given number is present in an array and if it is present then at what location it occurs. It is also known as sequential search. It is very simple and works as follows: We keep on comparing each element with the element to search until the desired element is found or list ends.

1) Write a program to perform Linear Search on the elements of a given array.(non-recursive)

PROGRAM:

```
#include<stdio.h>
int main()
{
int a[50],n,key,ans,i;
printf("enter size\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("values in the list are\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
printf("enter key to search\n");
scanf("%d",&key);
ans=lsnr(a,n,key);
if(ans==-1)
{
printf("element not found\n");
}
else
{
printf("key element present at index number=%d\n",ans);
```

```
}
return 0;
}

int lsnr(int a[],int n,int key)
{
int i;
for(i=0;i<n;i++)
{
if(a[i]==key)
{
return i;
break;

}
}
return -1;
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-MS-7528:~$ gcc lsnr.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter size
5
enter values
50
30
20
40
10
values in the list are

50      30      20      40      10
enter key to search
20
key element present at index number=2
gpcetadmin@gpcetadmin-MS-7528:~$ gcc lsnr.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter size
5
enter values
50
40
30
```

```
20
10
values in the list are

50      40      30      20      10
enter key to search
60
element not found
```

2) Write a program to perform Linear Search on the elements of a given array.(recursive)

PROGRAM:

```
#include<stdio.h>
int main()
{
int a[50],n,key,ans,i;
printf("enter size\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("values in the list are\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
printf("enter key to search\n");
scanf("%d",&key);
ans=lsr(a,n,key);
if(ans==-1)
{
printf("element not found\n");
}
else
{
printf("key element present at index number=%d\n",ans);
}
return 0;
}

int lsr(int a[],int n,int key)
{
```

```
if(n==-1)
{
return -1;
}
if(a[n]==key)
{
return n;
}
else
{
lsr(a,n-1,key);
}
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-MS-7528:~$ gedit lsr.c
gpcetadmin@gpcetadmin-MS-7528:~$ gcc lsr.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter size
5
enter values
50
30
20
40
10
values in the list are

50      30      20      40      10
enter key to search
20
key element present at index number=2
gpcetadmin@gpcetadmin-MS-7528:~$ gcc lsr.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter size
5
enter values
50
40
30
10
20
values in the list are
```

```
50      40      30      10      20
enter key to search
60
element not found
```

Binary Search- Analyzing Search Algorithm:

C program for binary search: This code implements binary search in c language. It can only be used for sorted arrays, but it's fast as compared to linear search. If you wish to use binary search on an array which is not sorted then you must sort it using some sorting technique say merge sort and then use binary search algorithm to find the desired element in the list. If the element to be searched is found then its position is printed.

3) Write a program to perform Binary Search on the elements of a given array(non-recursive)

PROGRAM:

```
#include<stdio.h>
int main()
{
int a[50],n,key,ans,i;
printf("enter size\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("values in the list are before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}

bubblesort(a,n);
printf("values after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
printf("enter key to search\n");
scanf("%d",&key);
ans=bsnr(a,n,key);
```

```
if(ans==-1)
{
printf("element not found\n");
}
else
{
printf("key element present at index number=%d\n",ans);
}
return 0;
}




bubblesort(int a[],int n)
{
int i,j,temp;
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
}


int bsnr(int a[],int n,int key)
{
int mid,first=0,last=n-1;
while(first<=last)
{
mid=(first+last)/2;
if(key==a[mid])
return mid;
else if(key>a[mid])

first=mid+1;
else
```

```
last=mid-1;
}
return -1;
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-MS-7528:~$ gedit bsnr.c
gpcetadmin@gpcetadmin-MS-7528:~$ gcc bsnr.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter size
5
enter values
50
40
30
20
10
values in the list are before sorting
50      40      30      20      10
Values after sorting
10      20      30      40      50
enter key to search
30
key element present at index number=2
gpcetadmin@gpcetadmin-MS-7528:~$ gcc bsnr.c
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter size
5
enter values
50
40
3020
10
5
values in the list are before sorting

50      40      3020  10      5
Values after sorting
5       10      40      50      3020  e
enter key to search
3000
element not found
```

4) Write a program to perform Binary Search on the elements of a given array(recursive)

PROGRAM:

```
#include<stdio.h>
int main()
{
int a[50],n,key,ans,i;
printf("enter size\n");
scanf("%d",&n);
printf("enter values\n");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("values in the list are before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
bubblesort(a,n);
printf("values after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
printf("enter key to search\n");
scanf("%d",&key);
ans=bsr(a,n,key,0,n-1);
if(ans==-1)
{
printf("element not found\n");
}
else
{
printf("key element present at index number=%d\n",ans);
}
return 0;
}
```

```
bubblesort(int a[],int n)
{
int i,j,temp;
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
}

int bsr(int a[],int n,int key,int first,int last)
{
int mid;
first=0;
last=n-1;
if(first<=last)
{
mid=(first+last)/2;
if(key==a[mid])
return mid;
else if(key>a[mid])
bsr(a,n,key,mid+1,last);
else if(key<a[mid])
bsr(a,n,key,first,mid-1);
}
return -1;
}
```

INPUT/OUTPUT:

```
gpcetadmin@gpcetadmin-MS-7528:~$ gcc bsr.c
```

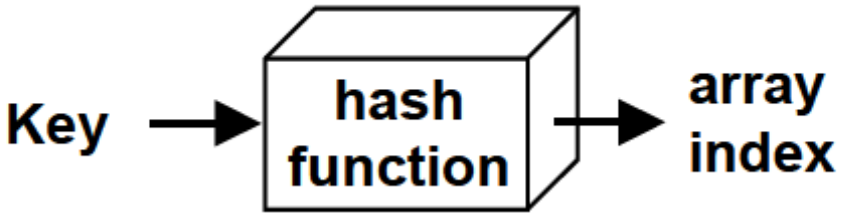
```
gpcetadmin@gpcetadmin-MS-7528:~$ ./a.out
enter size
3

enter values
30
20
10
values in the list are before sorting
30      20      10
values after sorting
10      20      30
enter key to search
20
key element present at index number=1
```

Hashing

Having an insertion, find and removal of O(log(N)) is good but as the size of the table
becomes larger, even this value becomes significant. We would like to be able to use an
algorithm for finding of O(1). This is when hashing comes into play!

Hashing using Arrays

When implementing a hash table using arrays, the nodes are not stored consecutively,
instead the location of storage is computed using the key and a ***hash*** function. The computation of the array index can be visualized as shown below:



The value computed by applying the hash function to the key is often referred to as the
hashed key. The entries into the array, are scattered (not necessarily sequential) as can be
seen in figure below

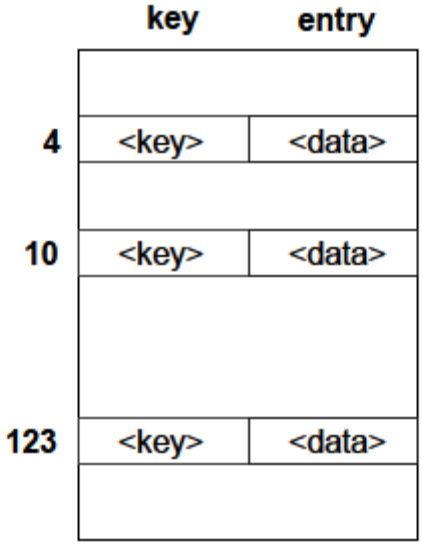


Figure 6. Hashed Array

The cost of the insert, find and delete operations is now only O(1). Can you think ofwhy?

Hash tables are very good if you need to perform a lot of search operations on a relatively stable table (i.e. there are a lot fewer insertion and deletion operations than search operations).

One the other hand, if traversals (covering the entire table), insertions, deletions are a lot more frequent than simple search operations, then ordered binary trees (also called AVL trees) are the preferred implementation choice.

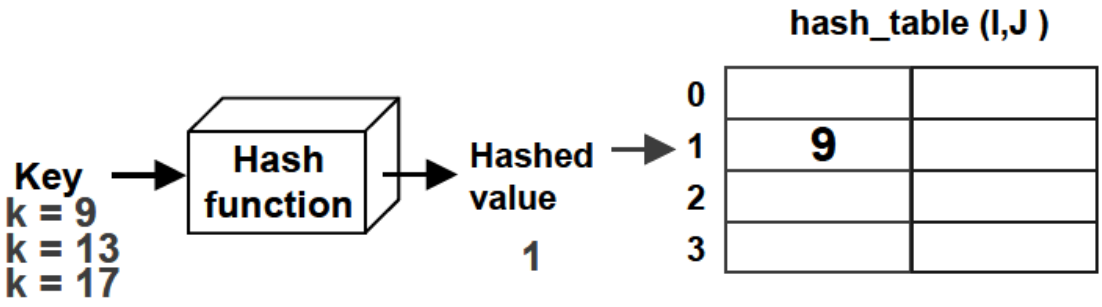
Hashing Performance
There are three factors the influence the performance of hashing:
__Hash function
o should distribute the keys and entries evenly throughout the entire table
o should minimize collisions
__Collision resolution strategy
o Open Addressing: store the key/entry in a different position
o Separate Chaining: chain several keys/entries in the same position
__Table size
o Too large a table, will cause a wastage of memory
o Too small a table will cause increased collisions and eventually force *rehashing* (creating a new hash table of larger size and copying the contents of the current hash table into it)
o The size should be appropriate to the hash function used and should typically be a prime number. Why?

Selecting Hash Functions

The hash function converts the key into the table position. It can be carried out using:
Modular Arithmetic: Compute the index by dividing the key with some value and use the remainder as the index. This forms the basis of the next two techniques.
For Example: index := keyMOD table_size
Truncation: Ignoring part of the key and using the rest as the array index. The problem with this approach is that there may not always be an even distribution throughout the table.
For Example: If student id's are the key 928324312 then select just the last three digits as the index i.e. 312 as the index. => the table size has to be atleast 999. Why?
Folding: Partition the key into several pieces and then combine it in some convenient way.
For Example:
o For an 8 bit integer, compute the index as follows:
Index := (Key/10000 + KeyMOD 10000) MOD Table_Size.
o For character strings, compute the index as follows:

```
Index :=0
For I in 1.. length(string)
Index := Index + ascii_value(String(I))
```

Collision
Let us consider the case when we have a single array with four records, each with two
fields, one for the key and one to hold data (we call this a *single slot bucket*). Let the hashing function be a simple modulus operator i.e. array index is computed by finding the
remainder of dividing the key by 4.
Array Index := key MOD 4
Then key values 9, 13, 17 will all hash to the same index. When two(or more) keys hash
to the same value, a **collision** is said to occur.



Collision Resolution
The hash table can be implemented either using
Buckets: An array is used for implementing the hash table. The array has size m*p where m is the number of hash values and p (³ 1) is the number of slots (a slot can hold one entry) as shown in figure below. The *bucket* is said to have p slots.

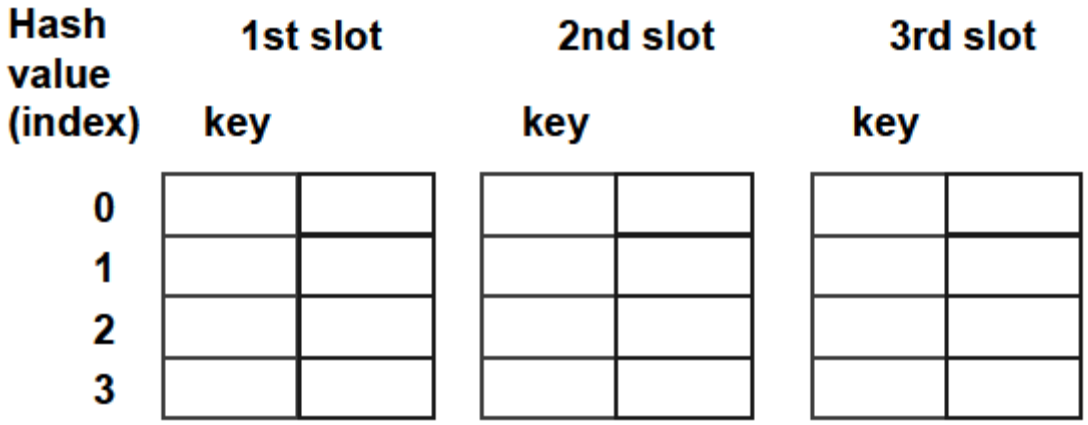


Figure 8. Hash Table with Buckets

Chaining: An array is used to hold the key and a pointer to a liked list (either singly or doubly linked) or a tree. Here the number of nodes is not restricted(unlike with buckets). Each node in the chain is large enough to hold one entry as shown in figure below.

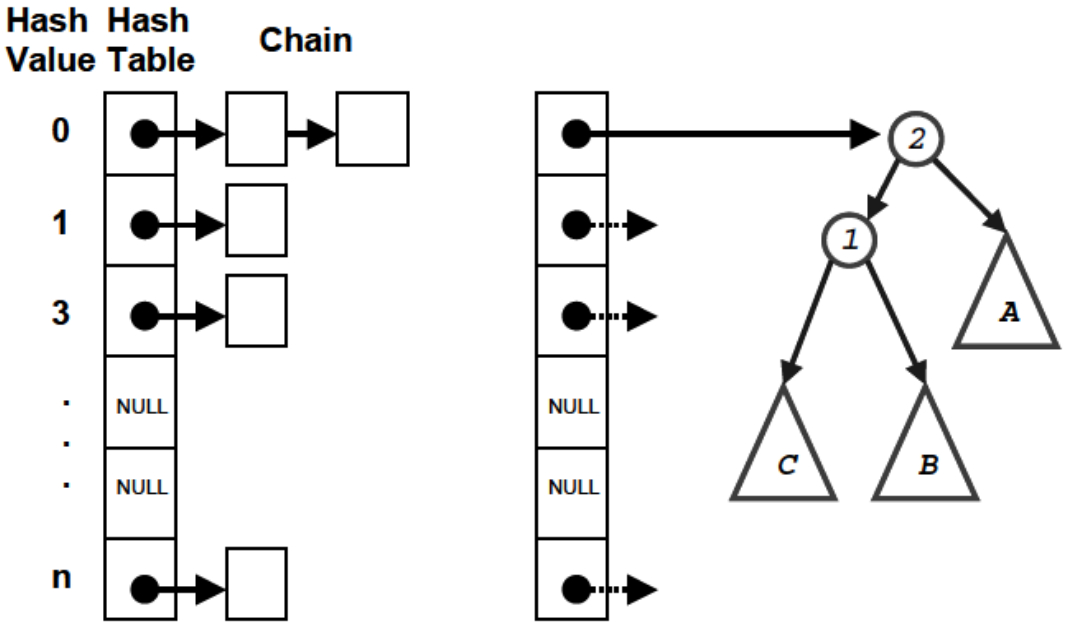


Figure 9. Chaining using Linked Lists / Trees

Open Addressing (Probing)
Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets). If the index given by the hash function is occupied, then increment the table position by some number.
There are three schemes commonly used for probing:
Linear Probing: The linear probing algorithm is detailed below:

Index := hash(key)
While Table(Index) Is Full do
index := (index + 1) MOD Table_Size
if (index = hash(key))
return table_full
else
Table(Index) := Entry
Quadratic Probing: increment the position computed by the hash function in quadratic fashion i.e. increment by 1, 4, 9, 16, ...

Double Hash: compute the index as a function of two different hash functions.

Chaining

In chaining, the entries are inserted as nodes in a linked list. The hash table itself is an
array of head pointers.

The advantages of using chaining are

- Insertion can be carried out at the head of the list at the index
- The array size is not a limiting factor on the size of the table

The prime disadvantage is the memory overhead incurred if the table size is small.