Chapter 9: Searching (5 hours)

Introduction

- Searching is a process of checking & finding an element within the list of elements stored in any order.
- There are several types of searching techniques. Some of them are:
  1. Linear or sequential searching
  2. Binary Searching
  3. Hashing
- Linear Searching is the basic and simple method of searching.
- Binary Search takes some less time to search an element from the **sorted list** of element.
- So we can say that binary search method is more efficient then the linear search only drawback is that binary search work on the sorted list where there is no prerequisite for the linear search.

Linear or sequential searching:

In linear search, we access each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found. Therefore linear search can be defined as the technique which traverses the array sequentially to locate the given item.

Algorithm for Linear search

Let '**a**' be the linear array with '**n**' elements, and '**item**' is an element to be searched. The algorithm finds the location '**loc**' of item in '**a**' or give the failure message.

```
1. Read the item to be searched.
2. [Initialize counter] set loc= -1,j=0
3. [Search for item]
   Repeat while j<n
           If a[j] = item
               Set loc = j
               Break
           Else
               Set j = j+1
       Wend
4. [Successful] if loc>=0
            Print the searched value's (item's) position (loc)
5. [Unsuccessful] else
           Print the searching item is not found.
6. [END]
```

2. Binary Search:

Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do the binary search, first we have to sort the array elements. The logic behind this technique is given below.

1. First find the middle element of the array.
2. Compare the middle element with an item.
3. There are three cases.
    a). If it is a desired element then search is successful,
    b). If it is less than the desired item then search only in the first half of the array.
    c). If it is greater than the desired item, search in the second half of the array.
4. Repeat the same steps until an element is found or search area is exhausted.

In this way, at each step we reduce the length of the list to be searched by half.

Algorithm for binary search

Let **'a'** be the array of size **'maxsize'** and **'beg', 'end'** and **'mid'** are variables to denote first, last and middle location of a segment.

This algorithm finds the location **'Loc'** of '**item**' in array '**a**' or return fail (sets loc = NULL).

```
1. Initialize the starting variables,
    Set beg = first element, end = last element and mid = int ((beg + end)/2)
2. Repeat steps 3 and 4 while beg<=end and a[mid]!=item
3. if item < a[mid], then
            Set end=mid – 1
    Else
            Set beg=mid + 1
   [End if]
4. set mid=int ((beg + end)/2)
5. if a [mid]=item then
            Set loc=mid
            Print the value and its position
    else
            Set loc=NULL
            Print search unsuccessful
    [End if]
6. Exit
```

Question 1:

Search 14 from the data given below using binary search. 2 7 8 10 12 14 16 18

Solution: Let, a be the array

beg = 0 (first element 2),

end = 7 (last element 18),

item = 14

0	1	2	3	4	5	6	7
2	**7**	**8**	**10**	**12**	**14**	**16**	**18**

Step 1:

mid = int ((beg + end)/2) = int (0 + 7)/2 = int (3.5) = 3

So, a [3] = 10 is the mid-point

0	1	2	3	4	5	6	7
2	**7**	**8**	**10**	**12**	**14**	**16**	**18**

First half Array Mid Second half array

Step 2: Compare mid-point with the item to be searched.

if a [mid]! =item && beg<=end → true

go to step 3 & 4

if 14 < a [mid] → false

So, item must be in second half of array.

And, set beg = mid + 1= 3 + 1= 4

0	1	2	3	4	5	6	7
2	**7**	**8**	**10**	**12**	**14**	**16**	**18**

Beg end

mid = int ((beg + end)/2) = int (4 + 7)/2 = int (5.5) = 5

So, a [5] = 14 is the mid-point

0	1	2	3	4	5	6	7
2	**7**	**8**	**10**	**12**	**14**	**16**	**18**

Mid

Step 3: Compare mid-point with the item to be searched.

if a[mid]!=item && beg<=end → false

go to step 5

if a[mid] = item (if 14 ==14)

Item is located.

Question 2: Search 30 from the data given below using binary search. **9 13 19 25 28 30 33 35 42 44**

Sequential search efficiency

The number of comparisons of keys done in sequential search of a list of length ***n*** is

i. Unsuccessful search: ***n*** comparisons

ii. Successful search, best case: **1** comparison

iii. Successful search, worst case: ***n*** comparisons

iv. Successful search, average case: **(*n* + 1)/2** comparisons

v. In any case, the number of comparison is ***O*(*n*)**

Binary search efficiency

i. In all cases, the no. of comparisons in proportional to n

ii. Hence, no. of comparisons in binary search is *O* (log n), where n is the no of items in the list

iii. Obviously binary search is faster then sequential search, but there is an extra overhead in maintaining the list ordered

iv. For small lists, better to use sequential search

v. Binary search is best suited for lists that are constructed and sorted once, and then repeatedly searched

Binary Search Procedure

```
void binsrch( int a[], int beg, int end, int item)
{
        int mid;
        int count = 0;
        mid = (beg + end)/2;
        count ++;
        while(beg <= end && item != a[mid])
        {
                If(item < a[mid])
                        end = end + 1;
                else
                        beg = beg + 1;
                mid = (beg + end)/2;
                count ++;
        }
        if(item == a[mid])
        {
                printf("\nSearch Successful!!!\n It took %d iterations to find this item", cnt);
                printf("\nThe position is %d",mid);
        }
        else
                printf("\n Search Unsuccessful");
        getch();
}
```

Hashing

- "Derive a number from the key information given to you and use that number to access all information related to the key".

This is the basic principle of hashing. This way, we can access any record in a specific time without making any comparison, irrespective of the number of record in the file.

Hashing is a scheme of searching that will use some function say hash to directly find out the location of the record in a constant search time, no matter where the record is in the file.

In order words the process of mapping large amount of data into a smaller table is called hashing.

This searching scheme is easy to program compared to trees. But it is difficult to expand, since it is based on arrays.

Hash Table

A hash table is simply an array that is address via a hash function. It is a data structure made up of

- A table of some fixed size to hold a collection of records each uniquely identified by some key
- A function called hash function that is used to generate index values in the table.

Hash Function

The basic idea in hashing is the transformation of a key into the corresponding location in the hash table. This is done by a hash function. A hash function can be defined as a function that takes key as input and transforms it into a hash table index usually denoted by H.

Some popular hash functions are

- Folding
- Mid square method
- Division method

Hash of Key

Let H be a hash function and k is a key then H(k) is called hash-of-key. The hash-of-key is the index at which a record with the key values k must be kept.

Store and retrieve operation using a Hash Function

The result of a hash function tells us where to look for a particular element in order to retrieve, modify or delete the element. We could use the following algorithm for retrieving a record.

```
retrieve(key)
{
        int location;
        location = hash(key);           //The result of hash function gives the location
        record = info[location];        //you get the record from the info array
}
```

We must put each new record into the correct slot according to the hash function when you insert a record. The algorithm for inserting a record follows;

```
insert(key)
{
        int location;
        location = hash(key);           //hash function gives the location loc
        info[location] = record;        //insert record into info array at array position loc.
}
```

Popular Hashing Functions

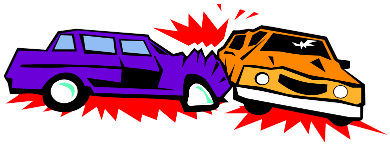
The two principle criteria in selection of a good hash function are:

1. It must be very easy and quick to compute hash value.
2. It must minimize collision

What is collision?

There are a finite number of indices in a table. But there are large numbers of keys so it is clearly impossible to get two different indexes for two distinct keys.

A situation in which two different keys k1 and k2 hash to the same index of the table i.e. h (k1) = h (k2), it is called collision or hash clash.



We have, in general following methods.

- Folding method
- Mid square method
- Division method

Folding method

- E.g. Let the keys be of four digits, chopping the key into two parts and adding yields
- Hash (5421) = 54 + 21 = 75

So 75 is the index at which we should store or retrieve record with key 5421

Mid square method: The key is squared. We defined the hash function in this case as

- Hash (key) = p;

Where p is obtained by deleting digits from both ends of $(\text{key})^2$. We emphasize that the same positions of $(\text{key})^2$ must be used for all the keys.

The following calculation are performed

Key	= 5421	1825
(Key) 2	=$293\underline{87}241$	$33\underline{30}625$
Hash (key)	= 87	30

Leaving 3 digits from the last and then taking two preceding digits.

Division Remainder method

Convert the key to an integer, divide by the size of the index range and take the remainder as the result.

E.g., hash (key) = hash (1029)
Let 100 be the table size (index range)
Then,
Hash (1029) = 1029%100 = 29

29 is the hash value (i.e. index of the array)

To get a good distribution of indices, prime number makes the best table size.

Collision Resolution Techniques

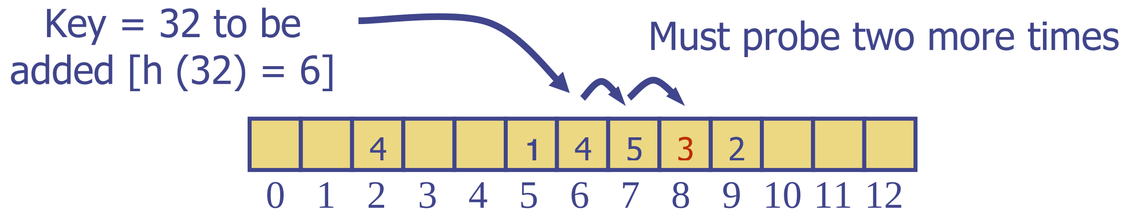
When a collision occurs, alternative locations in the table are tried until an empty location is found. Looking for the next available position is called probing.

Techniques are as follows:

- Linear probing
- Quadratic probing
- Double hashing
- Chaining

Linear probing:

A simple approach to resolve collision is to store the colliding element in the next available space. This technique is known as linear probing.



E.g. Let the hash function be h (key) = key % 10 (10 is the table size)
Then using linear probing, let’s insert following keys in the hash table

20, 15, 89, 18, 49, 48, 79, 21

9		9		**9**	**89**	9	89
8		8		8		**8**	**18**
7		7		7		7	
6		6		6		6	
5		**5**	**15**	5	15	5	15
4		4		4		4	
3		3		3		3	
2		2		2		2	
1		1		1		1	
0	**20**	0	20	0	20	0	20
	20%10=**0**		**15**%10=**5**		**89**%10=**9**		**18**%10=**8**

9	89 (collision)	9	89 (find next)	**9**	89 (collision)	9	89
8	18	**8**	18 (collision)	8	18	8	18
7		7		7		7	
6		6		6		6	
5	15	5	15	5	15	5	15
4		4		4		4	**21**
3		3		3	**79**	3	79 (find next)
2		2	**48**	2	48 (find next)	2	48 (find next)
1	**49**	1	49 (find next)	1	49 find next)	**1**	49 (collision)
0	20 (find next)	0	20 (find next)	0	20 (find next)	0	20
	49%10=**9**		**48**%10=**8**		**79**%10=**9**		**21**%10=**1**

Disadvantages of linear probing

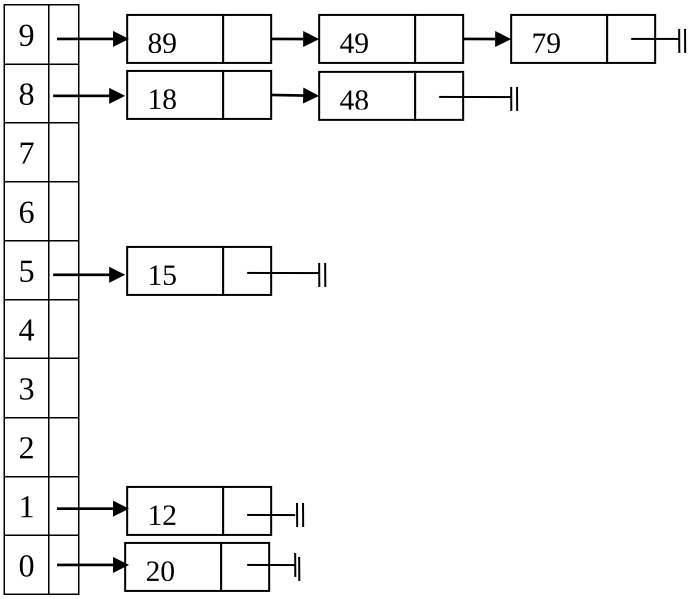
One man disadvantage of linear probing is that, records tend to cluster, that is, the records appear next to one another when the table is about half full.

Clustering occurs when a hash function is biased towards the placement of keys into a given region within the storage space. When the linear method is used to resolve collisions, this clustering problem is compounded, because keys that collide are loaded relatively close to the initial collision point.

Chaining

- It is another technique to deal with collision. In this method, for each location in the table, we keep a linked list of records that hash to the same index.
- The hash table contains pointers to linked list nodes; we can view these as the head pointers.
- Each time a record is inserted, it is added to the list at the location given by the hash function.
- When a collision occurs, the record is simply added to the list at the collision site.

For the above keys and hash function, chaining yields



Quadratic Probing

This method makes an attempt to correct the problem of clustering with linear probing. It forces the problem key to move quickly a considerable distance from the initial collision. When a key value hashes and collision occurs for key, this method probes the table location at

$(h (k) + 1^2)$ % table-size,
$(h (k) + 2^2)$ % table-size,
$(h (k) + 3^3)$ % table-size and so on.

That is, the first rehash adds 1 to the hash value. The second rehash adds 4, the third add 9 and so on.
It reduces primary clustering but suffers from secondary clustering: keys that hash to some initial slot will probe the same alternative cells.
There is no guarantee to finding an empty location once the table gets more than half full

9		9		**9**	**89**	9	89
8		8		8		**8**	**18**
7		7		7		7	
6		6		6		6	
5		**5**	**15**	5	15	5	15
4		4		4		4	
3		3		3		3	
2		2		2		2	
1		1		1		1	
0	**20**	0	20	0	20	0	20
	20%10=**0**		**15**%10=**5**		**89**%10=**9**		**18**%10=**8**

9	89 ((49+1)%10)	9	89 ((48+4)%10)	**9**	89 ((79+1)%10)	9	89
8	18	**8**	18 ((48+1)%10)	8	18 ((79+16)%10)	8	18
7		7		7		7	
6		6		6		6	
5	15	5	15	5	15 ((79+25)%10)	5	15
4		4		4	**79**	4	79
3	**49**	3	49	3	49 ((79+9)%10)	3	49
2		2	**48**	2	48	2	48
1		1		1		**1**	**21**
0	20 ((49+4)%10)	0	20	0	20 ((79+4)%10)	0	20
	49%10=**9**		**48**%10=**8**		**79**%10=**9**		**21**%10=**1**

Double Hashing

When collision occurs, a new hash function is defined.

$$H_2(key) = R - (key\%R)$$

Where R is the nearest prime number from its table size.

Rehashing

If at any stage the hash table become almost full (when packing density is more than 70%) then it will be difficult to find the free slot which will increase execution time. If the hash function produces collisions, we create a new hash table of double size. We may use the old hash value on input to a rehash function and compute a new hash value. For rehashing with linear probing, we can use the rehash function as:

(Old hash value + constant) % array-size

Where constant and array size are relatively prime, i.e., the largest number that divides both of them is 1. For e.g., given 100 slots array, we may use constant in rehash function:

(Old hash value + 3) % 100

Procedures of some Hash Resolutions:

An array is declared and each cell is assigned -1 to denote free cell.

```
int keys[10];
for (int i=0; i<10; i++)
    keys[i] = -1;
```

```
void linear_probing(int keys, int target)
{
    int rem;
    rem = target%10;
    if(keys[rem] == -1;
        keys[rem] = target;
    else
    {
        while(keys[rem] != -1)
        {
            If(rem >= 10)
                rem = 0;
            rem++;
        }
        keys[rem] = target;
    }
}
```

```
void double_hashing(int keys[], int target)
{
    int rem;
    rem = key%10;
    if(keys[rem] == -1)
        keys[rem] = target;
    else
    {
        Rem = 7 – (key%7)
        while(keys[rem] != -1)
        {
            if(rem >= 10)
                rem = 0;
            rem++;
        }
        keys[rem]=target;
    }
}
```

```
void quadratic_probing(int keys[], int target)
{
    int rem;
    rem = key%10;
    if(keys[rem] == -1)
        keys[rem] = target;
    else
    {
        int x = 0;
        int h = rem;
        while(keys[rem] != -1)
        {
            x++;
            int rem1 = h + pow(x,2);
            rem = rem1%10;
        }
        keys[rem] = target;
    }
}
```

Procedure for chaining:

```
struct chain
{
    int data;
    struct chain *next;
};
typedef struct chain node;
node *table[10];  Here, initially all the node pointers of the table are initialized to NULL.
void chaining(int key)
{
    int rem;
    rem = key%10;
    node *p;
    p = (node *)malloc(sizeof(node));
    p->data = key;
    p->next = NULL;
    if(table[rem] == NULL)
        table[rem] = p;
    else
    {
        node *temp;
        temp = table[rem];
        while(temp->next != NULL)
            temp = temp->next;
        temp->next = p;
    }
}
```