Chapter 6

Searching

12130CH06

"Even though most people won't be directly involved with programming, everyone is affected by computers, so an educated person should have a good understanding of how computer hardware, software, and networks operate."

— Brian Kernighan

In this Chapter

» *Introduction*
» *Linear Search*
» *Binary Search*
» *Search by Hashing*

6.1 Introduction

We store many things in our home and find them out later as and when required. Sometimes we remember the exact location of a required item. But, sometimes we do not remember the exact location and in that case we need to search for the required item. A computer also stores lots of data to be retrieved later as and when demanded by a user or a program.

Searching means locating a particular element in a collection of elements. Search result determines whether that particular element is present in the collection or not. If it is present, we can also find out the position of that element in the given collection. Searching is an important technique in computer science. In order to design algorithms, programmers need to understand the different ways in which a collection of data can be searched for retrieval.

6.2 Linear Search

Linear search is the most fundamental and the simplest search method. It is an exhaustive searching technique where every element of a given list is compared with the item to be searched (usually referred to as 'key'). So, each element in the list is compared one by one with the key. This process continues until an element matching the key is found and we declare that the search is successful. If no element matches the key and we have traversed the entire list, we declare the search is unsuccessful i.e., the key is not present in the list. This item by item comparison is done in the order, in which the elements are present in the list, beginning at the first element of the list and moving towards the last. Thus, it is also called sequential search or serial search. This technique is useful for collection of items that are small in size and are unordered.

Given a list `numList` of n elements and key value K, Algorithm 6.1 uses a linear search algorithm to find the position of the key K in `numList`.

Algorithm 6.1 : Linear Search

LinearSearch(numList, key, n)

Step 1: SET index = 0

Step 2: WHILE index < n, REPEAT Step 3

Step 3: IF numlist[index]= key THEN

```
        PRINT "Element found at position", index+1
        STOP
    ELSE
        index = index+1
```

Step 4: PRINT "Search unsuccessful"

Example 6.1 Assume that the numList has seven elements [8, -4, 7, 17, 0, 2, 19] so, n = 7. We need to search for the key, say 17 in numList. Table 6.1 shows the elements in the given list along with their index values.

Table 6.1 Elements in numList alongwith their index value

Index in numList	0	1	2	3	4	5	6
Value	8	-4	7	17	0	2	19

The step-by-step process of linear search using Algorithm 6.1. is given in Table 6.2.

Activity 6.1

Consider a list of 15 elements: L=[2,3,9,7,-6,11,12,17,45,23,29, 31,-37,41,43]. Determine the number of comparisons linear search makes to search for key = 12.

Table 6.2 Linear search for key 17 in numList of Table 6.1

index	index < n	numList[index]= key	index=index+1
0	0 < 7 ? Yes	8 = 17? No	1
1	1 < 7 ? Yes	-4 = 17? No	2
2	2 < 7 ? Yes	7 = 17? No	3
3	3 < 7 ? Yes	17 = 17? Yes	

Observe that after four comparisons, the algorithm found the key 17 and will display 'Element found at position 4'.

Let us now assume another arrangement of the elements in numList as [17, 8, -4, 7, 0, 2, 19] and search for the key K=17 in numList.

Table 6.3 Elements in numList alongwith their index value

Index in numList	0	1	2	3	4	5	6
Value	**17**	**8**	**-4**	**7**	**0**	**2**	**19**

Table 6.4 Linear search for key 17 in numList given in Table 6.3

index	index < n	numList[index]= key	index=index+1
0	0 < 7 ? Yes	17 = 17? Yes	1

From Table 6.4, it is clear that the algorithm had to make only 1 comparison to display 'Element found at position 1'. Thus, if the key to be searched is the first element in the list, the linear search algorithm will always have to make only 1 comparison. This is the minimum amount of work that the linear search algorithm would have to do.

Let us now assume another arrangement of the elements in numList as [8, -4, 7, 0, 2, 19, 17] and search for the key K =17 in numList.

On a dry run, we can find out that the linear search algorithm has to compare each element in the list till the end to display 'Element found at position 7'. Thus, if the key to be searched is the last element in the list, the linear search algorithm will have to make n comparisons, where n is the number of elements in the list. This is in fact the maximum amount of work the linear search algorithm would have to do.

Activity 6.2

In the list : L = [7,-1, 11,32,17,19,23,29,31, 37,43]
Determine the number of comparisons linear search takes to search for key = 43.

Let us now assume another case, where the key being searched is not present in the list. For example, we are searching for key = 10 in the numList.

In this case also, the algorithm has to compare each element in the list till the end to display 'Element is not found in the list'. Thus, if the key is not present in the list, the linear search algorithm will have to make n comparisons. This again is a case where maximum work is done. Let us now understand the program of a Linear Search. It takes a list of elements and the key to be searched as input and returns either the position of the element in the list or display that the key is not present in the list.

Program 6-1 Linear Search

```
def linearSearch(list, key):       #function to perform the search
    for index in range(0,len(list)):
        if list[index] == key:     #key is present
            return index+1         #position of key in list
    return None  #key is not in list
#end of function

list1 = []   #Create an empty list
maximum = int(input("How many elements in your list? "))
print("Enter each element and press enter: ")
for i in range(0,maximum):
    n = int(input())
    list1.append(n)     #append elements to the list
print("The List contents are:", list1)

key = int(input("Enter the number to be searched:"))
position = linearSearch(list1, key)
if position is None:
    print("Number",key,"is not present in the list")
else:
    print("Number",key,"is present at position",position)
```

Output

```
How many elements in your list? 4
Enter each element and press enter:
12
23
3
-45
The List contents are: [12, 23, 3, -45]
Enter the number to be searched:23
Number 23 is present at position 2
```

6.3 BINARY SEARCH

Consider a scenario where we have to find the meaning of the word Zoology in an English dictionary. Where do we search it in the dictionary?

1. in the first half?
2. around the middle?
3. in the second half?

It is certainly more prudent to look for the word in the second half of the dictionary as the word starts with the alphabet 'Z'. On the other hand, if we were to find the meaning of the word Biology, we would have searched in the first half of the dictionary.

We were able to decide where to search in the dictionary because we are aware of the fact that all words in an English dictionary are placed in alphabetical order. Taking advantage of this, we could avoid unnecessary comparison through each word beginning from the first word of the dictionary and moving towards the end till we found the desired word. However, if the words in the dictionary were not alphabetically arranged, we would have to do linear search to find the meaning of a word.

The binary search is a search technique that makes use of the ordering of elements in the list to quickly search a key. For numeric values, the elements in the list may be arranged either in ascending or descending order of their key values. For textual data, it may be arranged alphabetically starting from a to z or from z to a.

In binary search, the key to be searched is compared with the element in the middle of a sorted list. This could result in either of the three possibilities:

i) the element at the middle position itself matches the key or

ii) the element at the middle position is greater than the key or

iii) the element at the middle position is smaller than the key

If the element at the middle position matches the key, we declare the search successful and the searching process ends.

We are using the term iteration and not comparison in binary search, because after every unsuccessful comparison, we change the search area redefining the first, mid and last position before making subsequent comparisons.

If the middle element is greater than the key it means that if the key is present in the list, it must surely be in the first half. We can thus straight away ignore the second half of the list and repeat the searching process only in the first half.

If the middle element is less than the key, it means if the key is present in the list, it must be in the second half. We can thus straight away ignore the first half of the list and repeat the searching process only in the second half. This splitting and reduction in list size continued till the key is either found or the remaining list consists of only one item. If that item is not the key, then the search is unsuccessful as the key is not in the list.

Thus, it is evident that unlike linear search of elements one-by-one, we can search more efficiently using binary search provided the list from which we want to search is arranged in some order. That is, the list needs to be sorted.

Activity 6.3

Consider the numList [17, 8, -4, 7, 0, 2, 19]. Sort it using the sort() function of Python's Lists. Now apply binary search to search for the key 7. Determine the number of iterations required.

If the list to be searched contains an even number of elements, the mid value is calculated using the floor division (//) operator. If there are 10 elements in the list, then the middle position (mid) = 10//2 = 5. Therefore, the sixth element in the list is considered the middle element as we know that the first element in list has index value 0. If required, the list is further divided into two parts where the first half contains 5 elements and the second half contains 4 elements.

It is interesting to note that the intermediate comparisons which do not find the key still give us information about the part of the list where the key may be found! They reveal whether the key is before or after the current middle position in the list, and we use this information to narrow down or reduce our search area. Thus, each unsuccessful comparison reduces the number of elements remaining to be searched by half, hence the name binary search. Let us now discuss the algorithm of binary search.

Activity 6.4

Consider a list [-4, 0, 2, 7, 8, 17, 19]. Apply binary search to find element -4. Determine the number of key comparisons required.

Given a list numList of n elements and key value K, Algorithm 6.2 shows steps for finding position of the key K in the numList using binary search algorithm.

Algorithm 6.2: Binary Search

BinarySearch(numList, key)

Step 1: SET first = 0, last = n-1

Step 2: Calculate mid = (first+last)//2

Step 3: WHILE first <= last REPEAT Step 4

Step 4: IF numList[mid] = key

PRINT "Element found at position", " mid+1

STOP

ELSE

IF numList[mid] > key, THEN last = mid-1

ELSE first = mid + 1

Step 5: PRINT "Search unsuccessful"

The binary search algorithm does not change the list. Rather, after every pass of the algorithm, the search area gets reduced by half. That is, only the index of the element to be compared with the key changes in each iteration.

Example 6.2 Consider a sorted list comprising of 15 elements:

```
numList = [2,3,5,7,10,11,12,17,19,23,29,31,37,41,43]
```

We need to search for the key, say 17 in numList. The first, middle and last element identified in numList alongwith their index values are shown in Table 6.5.

Table 6.5 Elements in sorted numList alongwith their index value

	first						mid						last
Index in numList	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Value	2	3	5	7	10	11	12	17	19	23	29	31	37	41	43

Table 6.6 Working of binary search using steps given in Algorithm 6.2.

	first	last	mid	$numList_{[mid]}$ == K	key < $L_{mid?}$	first <= last
At Start	0	14	(0+14)//2=7	Not known	Not known	0 <= 14? True
Iteration 1	0	14	7	17 = 17? Yes	Key is found. The search terminates	

Note that the algorithm had to make only 1 iteration to display 'Element found at position 8'. This is because the key being searched is the middle element in the list. Thus, binary search requires only 1 iteration when the key to be searched is the middle value in the list. This

is the minimum amount of work binary search would have to do to confirm that a key is present in the list.

Now, let us search for key 2 in the list.

```
numList = [2,3,5,7,10,11,12,17,19,23,29,31,37,41,43]
```

Activity 6.5

For L = [2,3,5,7,10,11,12,17,19,23,29,31,37,41,43]. Fill up the Table 6.8 for the given key values 2, 43, 17 and 9. What do you infer from Table 6.8 regarding performance of both the algorithms in different cases?

In the first iteration, we have the mid value as 17. As 2 is smaller than the mid value (17), we have to search for the first half of the list in the second iteration. We now consider only 7 elements. As 2 is smaller than the new mid value (7), we have to search for the first half of the remaining list in the third iteration. We now consider only 3 elements. Observe that the number of elements in the numList is halved each time. It reduces from 15 elements in iteration 1 to 7 elements in iteration 2, and to 3 elements in iteration 3. In the 3rd iteration, the algorithm finds that key 2 is smaller than the new mid value (3), we have to search in the first half of the remaining list. The list now has only 1 element in the fourth iteration and on comparison, it is found that the element is the same as key. Hence, the search terminates successfully and returns the position of key. Steps followed for binary search are given in Table 6.7.

Table 6.7 Searching key = 2 in the numList using binary search

	first	last	mid	numList[$_{mid}$] == K	K < numList[$_{mid}$]	first <= last
At Start	0	14	(0+14)//2= 7	Not known	Not known	True
Iteration 1	0	14	(0+14)//2= 7	17 = 2? No	2 < 17? True	0 <= 14? True
Iteration 2	0	6	(0+6)//2= 3	7 = 2? No	2 < 7? True	0 <= 6? True
Iteration 3	0	2	(0+2)//2= 1	3 = 2? No	2 < 3? True	0 <= 2? True
Iteration 4	0	0	(0+0)//2= 0	2 = 2? Yes	Key found, search Terminates, return position as (mid+1)=1	

As we can see, the binary search algorithm had to make 4 iterations to narrow down the list to a single element and decide that the search key is the first element of list. This is clearly the maximum work required to find a key in the given list.

Program 6-2 Binary Search

```
def binarySearch(list, key):
        first = 0
        last = len(list) - 1
        while(first <= last):
                mid = (first + last)//2
                if list[mid] == key:
                        return mid
                elif key > list[mid]:
                        first = mid + 1
                elif key < list[mid]:
                        last = mid - 1
        return -1

list1 = []   #Create an empty list
print ("Create a list by entering elements in ascending order")
print ("press enter after each element, press -999 to stop")
num = int(input())
while num!=-999:
    list1.append(num)
    num = int(input())
n = int(input("Enter the key to be searched: "))
pos = binarySearch(list1,n)
if(pos != -1):
    print( n,"is found at position", pos+1)
else:
    print (n,"is not found in the list ")
```

Output

```
Create a list by entering elements in ascending order
press enter after each element, press -999 to stop
1
3
4
5
-999
Enter the number to be searched: 4
4 is found at position 3

Second run of the program with different data:
Create a list by entering elements in ascending order
press enter after each element, press -999 to stop
12
8
3
-999
Enter the number to be searched: 4
4 is not found in the list
```

6.3.1 Applications of Binary Search

- Binary search has numerous applications including – searching a dictionary or a telephone directory, finding the element with minimum value or maximum value in a sorted list, etc.
- Modified binary search techniques have far reaching applications such as indexing in databases, implementing routing tables in routers, data compression code, etc.

6.4 Search by Hashing

Hashing is a technique which can be used to know the presence of a key in a list in just one step. The idea is if we already know the value at every index position in a list, it would require only a single comparison to check the presence or absence of a key in that list. Hashing makes searching operations very efficient. A formula called hash function is used to calculate the value at an index in the list.

Think and Reflect

Suppose a list has more than one element whose modulo division results in same remainder value. In such situations, what kind of hashing may be useful?

Thus, a *hash function* takes elements of a list one by one and generates an index value for every element. This will generate a new list called the hash table. Each index of the hash table can hold only one item and the positions are indexed by integer values starting from 0. Note that the size of the hash table can be larger than the size of the list.

A simple hash function that works with numeric values is known as the *remainder method*. It takes an element from a list and divides it by the size of the hash table. The remainder so generated is called the hash value.

h(element) = element % size(hash table)

We can easily implement a hash table using a Python's List. Let us consider an empty hash table having 10 positions as shown in Table 6.8:

Table 6.8 An Empty hash table with 10 positions

Index/ position	0	1	2	3	4	5	6	7	8	9
Value	None	None	None	None	None	None	None	None	None	None

Let us consider a list of numbers (34, 16, 2, 93, 80, 77, 51). We can use the hash function remainder

method explained earlier to create a hash table as shown in Table 6.9.

Table 6.9 hash function element % 10 applied on the elements of list

Element	34	16	2	93	80	77	51
Hash Value	34%10=4	16%10=6	2%10=2	93%10=3	80%10=0	77%10=7	51%10=1

After computing the hash values, each element is inserted at its designated position in the hash table shown in Table 6.10

Table 6.10 hash table generated for elements given in Table 6.10

index	0	1	2	3	4	5	6	7	8	9
Value	80	51	2	93	34	None	16	77	None	None

Now, to search for a key, we can calculate its index using the hashing function and compare the element at that index with the key to declare whether the element is present in the list or not. This search operation involves just one comparison and hence the same amount of time is always required to search a key irrespective of the size of the list.

Program 6-3 Use of hashing to find a key in the given list L

```
#Function to check if a key is present or not
def hashFind(key,hashTable):
    if (hashTable[key % 10] == key): #key is present
        return ((key % 10)+1)   #return the position
    else:
        return None          #key is not present
#end of function

#create hashTable with 10 empty positions
hashTable=[None, None, None, None, None, None, None, None, None,
None]
print("We have created a hashTable of 10 positions:")
print(hashTable)

L = [34, 16, 2, 93, 80, 77, 51]
print("The given list is", L[::] )

# Apply hash function
for i in range(0,len(L)):
    hashTable[L[i]%10] = L[i]
```

```
print("The hash table contents are: " )
for i in range(0,len(hashTable)):
    print("hashindex=", i," , value =", hashTable[i])

key = int(input("Enter the number to be searched:"))

position = hashFind(key,hashTable)
if position is None:
    print("Number",key,"is not present in the hash table")
else:
    print("Number ",key," present at ",position, " position")
```

Output:

```
We have created a hashTable of 10 positions:
[None, None, None, None, None, None, None, None, None, None]
The given list is [34, 16, 2, 93, 80, 77, 51]
The hash table contents are:
hashindex= 0  , value = 80
hashindex= 1  , value = 51
hashindex= 2  , value = 2
hashindex= 3  , value = 93
hashindex= 4  , value = 34
hashindex= 5  , value = None
hashindex= 6  , value = 16
hashindex= 7  , value = 77
hashindex= 8  , value = None
hashindex= 9  , value = None
Enter the number to be searched:16
Number  16  present at  7  position
```

6.4.1 COLLISION

The hashing technique works fine if each element of the list maps to a unique location in the hash table. Consider a list `[34, 16, 2, 26, 80]`. While applying the hash function say, list `[i]%10`, two elements (16 and 26) would have a hash value 6. This is a problematic situation, because according to our definition, two or more elements cannot be in the same position in the list. This situation is called *collision* in hashing.

We must have a mechanism for placing the other items with the same hash value in the hash table. This process is called *collision resolution*. Collision can be resolved in many ways, but it is beyond the scope of this book to discuss collision resolution methods.

If every item of the list maps to a unique index in the hash table, the hash function is called a *perfect hash function*. If a hash function is perfect, collision will never occur.

Apart from modulo division method, hash functions may be based on several other techniques like integer division, shift folding, boundary folding, mid-square function, extraction, radix transformation, etc. Again, it is beyond the scope of this book to discuss these methods.

The time taken by different hash functions may be different, but it remains constant for a particular hash function. The advantage of hashing is that the time required to compute the index value is independent of the number of items in the search list. It is to remember that the cost of computing a hash function must be small enough to make a hashing-based searching more efficient than other search methods.

SUMMARY

- Searching means trying to locate a particular element called key in a collection of elements. Search specifies whether that key is present in the collection or not. Also, if the key is present, it tells the position of that key in the given collection.
- Linear search checks the elements of a list, one at a time, without skipping any element. It is useful when we need to search for an item in a small unsorted list, but it is slow and time-consuming when the list contains a large number of items. The time taken to search the list increases as the size of the list increases.
- Binary search takes a sorted/ordered list and divides it in the middle. It then compares the middle element with the key to be searched. If the middle element matches the key, the search is declared successful and the program ends. If the middle element is greater than the key, the search repeats only in the first half of the list. If the middle element is lesser than the key, the search repeats only in the second half of the list.

This splitting and reduction in list size continue till the key is found or the remaining list consists of only one item.

- In binary search, comparisons that do not find the key still give us idea about the location where the key may probably be found! They reveal whether the key is before or after the current middle position in the list, and we can use this information to narrow down or reduce our searching efforts.
- Hash based searching requires only one key comparison to discover the presence or absence of a key, provided every element is present at its designated position decided by a hash function. It calculates the position of the key in the list using a formula called the hash function and the key itself.
- When two elements map to the same slot in the hash table, it is called collision.
- The process of identifying a slot for the second and further items in the hash table in the event of collision, is called collision resolution.
- A perfect hash function maps every input key to a unique index in the hash table. If the hash function is perfect, collisions will never occur.

EXERCISE

1. Using linear search determine the position of 8, 1, 99 and 44 in the list:

```
[1, -2, 32, 8, 17, 19, 42, 13, 0, 44]
```

   Draw a detailed table showing the values of the variables and the decisions taken in each pass of linear search.

2. Use the linear search program to search the key with value 8 in the list having duplicate values such as `[42, -2, 32, 8, 17, 19, 42, 13, 8, 44]`. What is the position returned? What does this mean?

3. Write a program that takes as input a list having a mix of 10 negative and positive numbers and a key value.

Apply linear search to find whether the key is present in the list or not. If the key is present it should display the position of the key in the list otherwise it should print an appropriate message. Run the program for at least 3 different keys and note the result.

4. Write a program that takes as input a list of 10 integers and a key value and applies binary search to find whether the key is present in the list or not. If the key is present it should display the position of the key in the list otherwise it should print an appropriate message. Run the program for at least 3 different key values and note the results.

5. Following is a list of unsorted/unordered numbers:

   `[50, 31, 21, 28, 72, 41, 73, 93, 68, 43, 45, 78, 5, 17, 97, 71, 69, 61, 88, 75, 99, 44, 55,9]`

   - Use linear search to determine the position of 1, 5, 55 and 99 in the list. Also note the number of key comparisons required to find each of these numbers in the list.
   - Use a Python function to sort/arrange the list in ascending order.
   - Again, use linear search to determine the position of 1, 5, 55 and 99 in the list and note the number of key comparisons required to find these numbers in the list.
   - Use binary search to determine the position of 1, 5, 55 and 99 in the sorted list. Record the number of iterations required in each case.

6. Write a program that takes as input the following unsorted list of English words:

   `[Perfect, Stupendous, Wondrous, Gorgeous, Awesome, Mirthful, Fabulous, Splendid, Incredible, Outstanding, Propitious, Remarkable, Stellar, Unbelievable, Super, Amazing].`

   - Use linear search to find the position of `Amazing, Perfect, Great` and `Wondrous` in the list. Also note the number of key comparisons required to find these words in the list.
   - Use a Python function to sort the list.
   - Again, use linear search to determine the position of `Amazing, Perfect, Great` and `Wondrous` in the list and note the number of key comparisons required to find these words in the list.
   - Use binary search to determine the position of `Amazing, Perfect, Great` and `Wondrous` in the sorted list. Record the number of iterations required in each case.

7. Estimate the number of key comparisons required in binary search and linear search if we need to find the details of a person in a sorted database having 230 (1,073,741,824) records when details of the person being searched lies at the middle position in the database. What do you interpret from your findings?

8. Use the hash function: `h(element)= element%11` to store the collection of numbers: `[44, 121, 55, 33, 110, 77, 22, 66]` in a hash table. Display the hash table created. Search if the values 11, 44, 88 and 121 are present in the hash table, and display the search results.

9. Write a Python program by considering a mapping of list of countries and their capital cities such as:

```
CountryCapital= {'India':'New Delhi','UK':
                 'London','France':'Paris',
                 'Switzerland': 'Berne',
                 'Australia': 'Canberra'}
```

Let us presume that our hash function is the length of the Country Name. Take two lists of appropriate size: one for keys (Country) and one for values (Capital). To put an element in the hash table, compute its hash code by counting the number of characters in Country, then put the key and value in both the lists at the corresponding indices. For example, India has a hash code of 5. So, we store India at the 5th position (index 4) in the keys list, and New Delhi at the 5th position (index 4) in the values list and so on. So that we end up with:

hash index = length of key - 1	List of Keys	List of Values
0	None	None
1	UK	London
2	None	None
3	Cuba	Havana
4	India	New Delhi
5	France	Paris
6	None	None
7	None	None
8	Australia	Canberra
9	None	None
10	Switzerland	Berne

Now search the capital of India, France and the USA in the hash table and display your result.