

Searching

Mohammad Asad Abbasi
Lecture 6

Searching

- Searching is a process of checking and finding an element from a list of elements.
- Following are the two important searching techniques :
 1. Linear or Sequential Searching
 2. Binary Searching

LINEAR OR SEQUENTIAL SEARCHING

- A **sequential search** of a list/array begins at the beginning of the list/array and continues until the item is found or the entire list/array has been searched.
- The linear (or sequential) search algorithm on an array is:
 - Sequentially scan the array, comparing each array item with the searched value.
 - If a match is found; return the index of the matched element; otherwise return -1 .

ALGORITHM FOR LINEAR SEARCH

- Let A be an array of n elements, $A[1], A[2], A[3], \dots, A[n]$. “data” is the element to be searched. Then this algorithm will find the location “loc” of data in A . Set $loc = -1$, if the search is unsuccessful.
1. Input an array A of n elements and “data” to be searched and initialize $loc = -1$.
 2. Initialize $i = 0$; and repeat through step 3 if $(i < n)$ by incrementing i by one.
 3. If $(data = A[i])$
 - (a) $loc = i$
 - (b) GOTO step 4
 4. If $(loc > 0)$
 - (a) Display “data is found and searching is successful”
 5. Else
 - (a) Display “data is not found and searching is unsuccessful”
 6. Exit

Search()

```
1.  printf (“\nEnter the element to be searched : ”);
2.  scanf (“%d",&item); //Input the item to be searched
3.  for(i=0;i < n;i++)
4.  {
5.  if item == arr[i])
6.  {
7.  printf (“\n%d found at position %d\n”,item,i+1);
8.  break;
9.  }
10. }
11. if (i == n)
12. printf (“\nItem %d not found in array\n”,item);
13. printf (“\n\nPress (Y/y) to continue : ”);5
```

TIME COMPLEXITY

- Time Complexity of the linear search is found by number of comparisons made in searching a record.
- In the best case, the desired element is present in the first position of the array, *i.e.*, only one comparison is made. So $f(n) = O(1)$.
- In the Average case, the desired element is found in the half position of the array, then $f(n) = O[(n + 1)/2]$.
- But in the worst case the desired element is present in the n th (or last) position of the array, so n comparisons are made. So $f(n) = O(n + 1)$.

Linear Search Tradeoffs

Benefits

- Easy algorithm to understand
- Array can be in any order

Disadvantage

- Inefficient (slow): for array of N elements, examines $N/2$ elements on average for value in array, N elements for value not in array

BINARY SEARCH

- Divide-and-conquer strategy.
- It uses a recursive method to search an array to find a specified value.
- The array must be a sorted array:
 - $a[0] \leq a[1] \leq a[2] \leq \dots \leq a[\text{finalIndex}]$
- If the value is found, its index is returned.
- If the value is not found, -1 is returned.
- Note: Each execution of the recursive method reduces the search space by about a half.



BINARY SEARCH

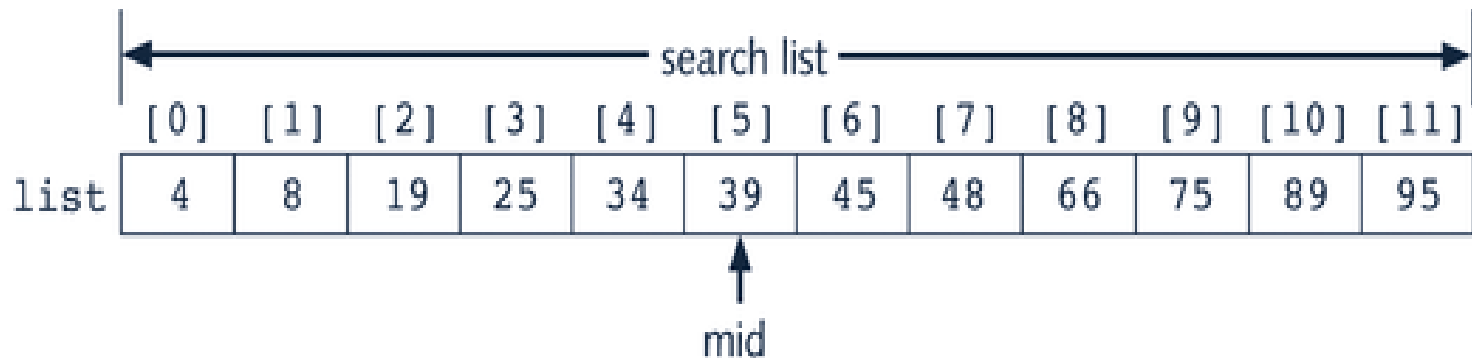
1. Find the middle element of the array (i.e., $n/2$ is the middle element if the array or the sub-array contains n elements).
2. Compare the middle element with the data to be searched, then there are following three cases.
 - a) If it is a desired element, then search is successful.
 - b) If it is less than desired data, then search only the first half of the array, i.e., the elements which come to the left side of the middle element.
 - c) If it is greater than the desired data, then search only the second half of the array, i.e., the elements which come to the right side of the middle element.

Repeat the same steps until an element is found or exhaust the search area.

Binary Search

$$\text{mid} = \frac{\text{left} + \text{right}}{2}$$

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95



Algorithm For Binary Search

1. Input an array A of n elements and “data” to be sorted
2. $LB = 0, UB = n; mid = \text{int}((LB+UB)/2)$
3. Repeat step 4 and 5 while $(LB \leq UB)$ and $(A[mid] \neq \text{data})$
4. If $(\text{data} < A[mid])$
(a) $UB = mid - 1$
5. Else
(a) $LB = mid + 1$
6. $Mid = \text{int}((LB + UB)/2)$
7. If $(A[mid] == \text{data})$
(a) Display “the data found”
8. Else
(a) Display “the data is not found”
9. Exit

Binary Search: Example

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Table 9-1 Values of `first`, `last`, and `middle` and the Number of Comparisons for Search Item 89

Iteration	first	last	mid	list[mid]	Number of Comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found is true)

Execution of the Method `search`

(Part 1 of 2)

key is 63

a[0]	15	← first == 0
a[1]	20	
a[2]	35	
a[3]	41	
a[4]	57	← mid = (0 + 9)/2
a[5]	63	
a[6]	75	
a[7]	80	
a[8]	85	
a[9]	90	← last == 9

next

a[0]	15	
a[1]	20	
a[2]	35	← Not in this half
a[3]	41	
a[4]	57	
a[5]	63	← first == 5
a[6]	75	
a[7]	80	← mid = (5 + 9)/2
a[8]	85	
a[9]	90	← last == 9

Execution of the Method **search**

(Part 2 of 2)

a[0]	15
a[1]	20
a[2]	35
a[3]	41
a[4]	57
a[5]	63
a[6]	75
a[7]	80
a[8]	85
a[9]	90

first == 5

last == 6

Not here

next

$\text{mid} = (5 + 6) / 2$ which is 5
a[mid] is a[5] == 63
key was found.
return 5.

Binary Search

[0]	ant
[1]	cat
[2]	chicken
[3]	cow
[4]	deer
[5]	dog
[6]	fish
[7]	goat
[8]	horse
[9]	camel
[10]	snake

Searching for cat

BinarySearch(0, 10)	middle: 5	cat < dog
BinarySearch(0, 4)	middle: 2	cat < chicken
BinarySearch(0, 1)	middle: 0	cat > ant
BinarySearch(1, 1)	middle: 1	cat = cat Return: true

Searching for zebra

BinarySearch(0, 10)	middle: 5	zebra > dog
BinarySearch(6, 10)	middle: 8	zebra > horse
BinarySearch(9, 10)	middle: 9	zebra > camel
BinarySearch(10, 10)	middle: 10	zebra > snake
BinarySearch(11, 10)		last > first Return: false

Searching for fish

BinarySearch(0, 10)	middle: 5	fish > dog
BinarySearch(6, 10)	middle: 8	fish < horse
BinarySearch(6, 7)	middle: 6	fish = fish Return: true

Efficiency of Binary Search

- The binary search algorithm is extremely fast compared to an algorithm that tries all array elements in order.
 - About half the array is eliminated from consideration right at the start.
 - Then a quarter of the array, then an eighth of the array, and so forth.
- Given an array with 1,000 elements, the binary search will only need to compare about 10 array elements to the key value, as compared to an average of 500 for a serial search algorithm.
- The binary search algorithm has a worst-case running time that is logarithmic: $O(\log n)$
 - A serial search algorithm is linear: $O(n)$

PROGRAM FOR BINARY SEARCH

```
1. void main()
2. {
3.   char opt;
4.   int arr[20],start,end,middle,n,i,item;
5.   clrscr();
6.   printf ("\nHow many elements you want to enter in the array : ");
7.   scanf ("%d",&n);
8.   for(i=0; i < n; i++)
9.   {
10.    printf ("\nEnter element %d : ",i+1);
11.    scanf ("%d",&arr[i]);
12.  }
13.  printf ("\n\nPress any key to continue...");
14.  getch();
15.  do
16.  {
17.    clrscr();
18.    printf ("\nEnter the element to be searched : ");
19.    scanf ("%d",&item);
20.    start=0;
21.    end=n - 1;
22.    middle=(start + end)/2;
```

PROGRAM FOR BINARY SEARCH

```
23) while(item != arr[middle] && start <= end)
24) {
25) if (item > arr[middle])
26) start=middle+1;
27) else
28) end=middle-1;
29) middle=(start+end)/2;
30) }
31) if (item==arr[middle])
32) printf("\n%d found at position %d\n",item,middle + 1);
33) if (start>end)
34) printf ("\n%d not found in array\n",item);
35) printf ("\n\nPress (Y/y) to continue : ");
36) scanf ("%c",&opt);
37) }while(opt == 'Y' || opt == 'y');
38) }/*End of main()*/
```