Search Algorithms

Searching Definition

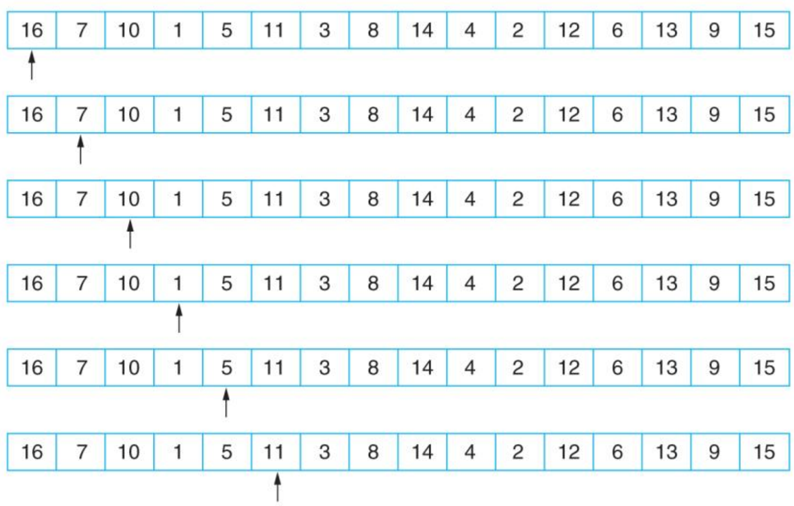
Search is used to find an element that has a particular value (10, 101, …). There are two classes of searching algorithms. The first one is applied on array like sequential search and binary search algorithms, and the second is applied on graph (data structure) like blind state search algorithms (depth and breadth) and heuristic search algorithms (best first search, A*, GA, …)

I. ***Sequential Search***

1. Looks for the target from the first to the last element of the list
2. The later in the list the target occurs the longer it takes to find it
3. Does not assume anything about the order of the elements in the list, so it can be used with an unsorted list

Sequential Search Example
(looking for a key value of 11)

FIGURE 3.1
Sequential search



Sequential Search Algorithm

```
for i = 0 to N-1 do
          if (target == list[i])
                    return i
          end if
end for
return -1  // Not found
```

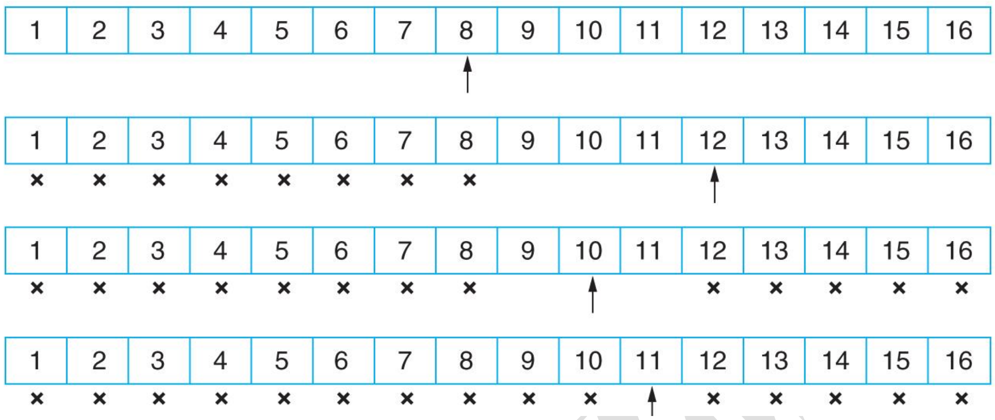
II. ***Binary Search Algorithm***

The **binary search** gets its name because the algorithm continually divides the list into two parts. It is used with a sorted list as follows:

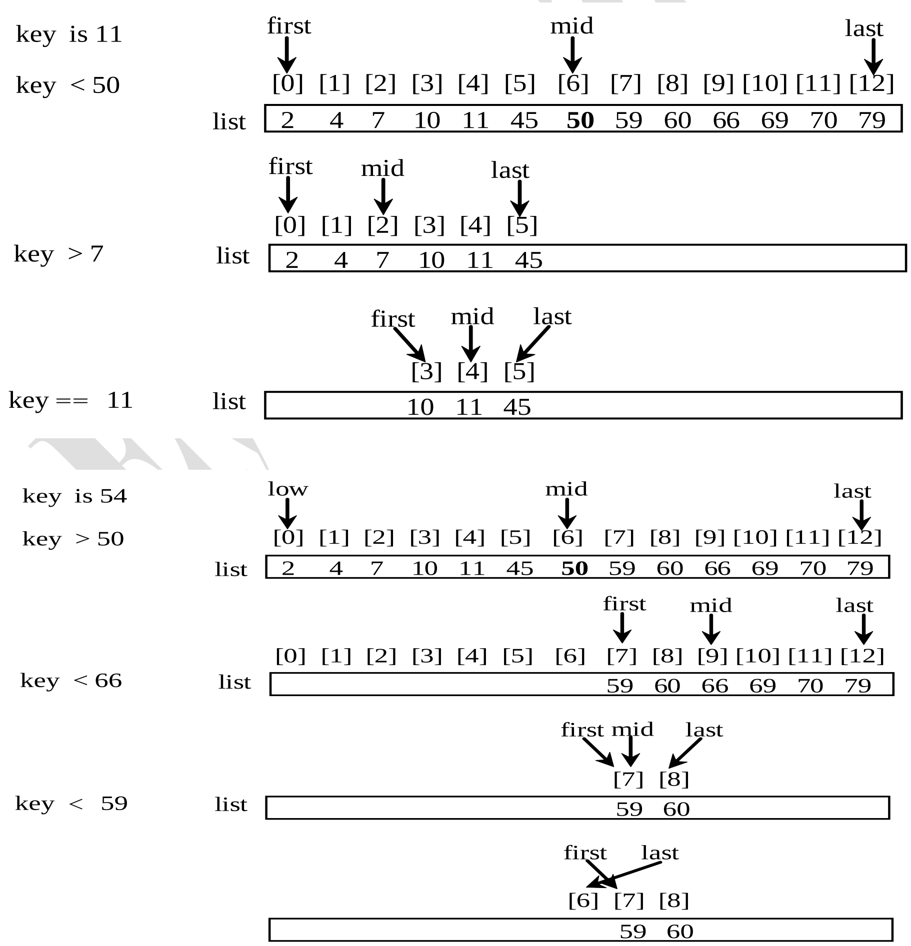
- First check the middle list element
- If the target **matches** the middle element, we are done
- If the target is **less than** the middle element, the key must be in the first half

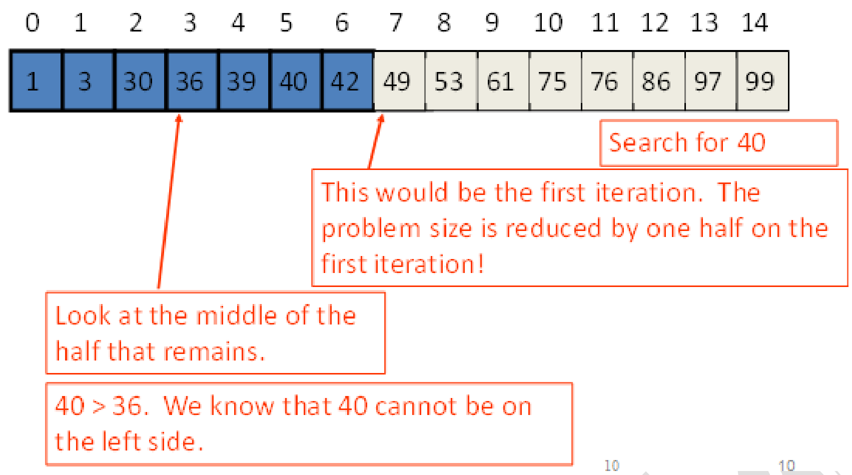
- If the target is **larger than** the middle element, the key must be in the second half

Binary Search Example
(key to search: 11)



Example:.

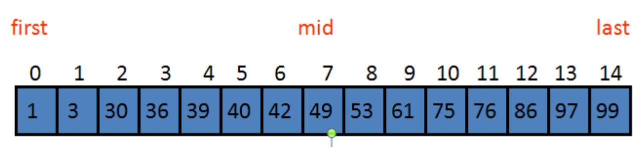




Binary Search Algorithm

```
 int BSA (int itemToFind, int[] A)
{
int first = 0;
int size ;
int last = size – 1;
int middle;
int result = -1;
boolean found = false;
while ( first <= last && !found ) {
            middle = ( first + last ) /2 ;
            if (itemToFind == A [middle])
                        { found = true; result=middle;}
            else if ( itemToFind < A[ middle ] )
                        last = middle – 1;
            else // itemToFind > A[ middle ]
                        first = middle + 1;
            }
Return result;
}
```


Example:




```
while ( first <= last && !found ) {
      middle = ( first + last ) /2 ;
      if ( itemToFind == A[ middle ] )
            found = true;
```

itemToFind: 42

found: false

```
    else if ( itemToFind < A[ middle ] )
        last = middle – 1;
    else // itemToFind > A[ middle ]
        first = middle + 1;
}
```

first mid last

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	30	36	39	40	42	49	53	61	75	76	86	97	99

```
while ( first <= last && !found ) {
    middle = ( first + last ) /2 ;
    if ( itemToFind == A[ middle ] )
        found = true;
    else if ( itemToFind < A[ middle ] )
        last = middle – 1;
    else // itemToFind > A[ middle ]
        first = middle + 1;
}
```

itemToFind: 42

found: false

first mid last

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	30	36	39	40	42	49	53	61	75	76	86	97	99

```
while ( first <= last && !found ) {
    middle = ( first + last ) /2 ;
    if ( itemToFind == A[ middle ] )
        found = true;
    else if ( itemToFind < A[ middle ] )
        last = middle – 1;
    else // itemToFind > A[ middle ]
        first = middle + 1;
}
```

itemToFind: 42

found: false

first
middle
last

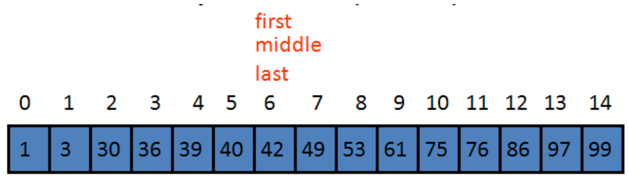
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	3	30	36	39	40	42	49	53	61	75	76	86	97	99

```
while ( first <= last && !found ) {
    middle = ( first + last ) /2 ;
    if ( itemToFind == A[ middle ] )
        found = true;
```

itemToFind: 42

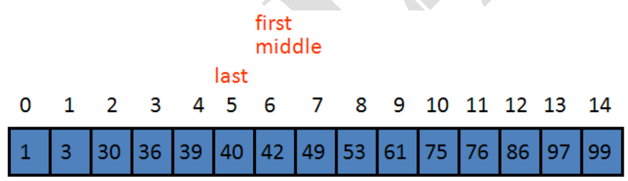
found: true

```
    else if ( itemToFind < A[ middle ] )
        last = middle – 1;
    else // itemToFind > A[ middle ]
        first = middle + 1;
    }
```




```
while ( first <= last && !found ) {
    middle = ( first + last ) /2 ;
    if ( itemToFind == A[ middle ] )
        found = true;
    else if ( itemToFind < A[ middle ] )
        last = middle – 1;
    else // itemToFind > A[ middle ]
        first = middle + 1;
    }
```

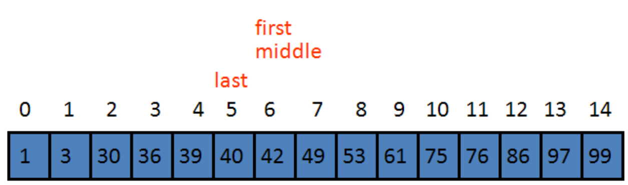
If itemToFind were 41 (not in array), then this would be true




```
while ( first <= last && !found ) {
    middle = ( first + last ) /2;
    if ( itemToFind == A[ middle ] )
        found = true;
    else if ( itemToFind < A[ middle ] )
        last = middle – 1;
    else // itemToFind > A[ middle ]
        first = middle + 1;
    }
```

Last would be made less than first

Then the loop would exit




```
while ( first <= last && !found ) {
    middle = ( first + last ) /2 ;
    if ( itemToFind == A[ middle ] )
        found = true;
    else if ( itemToFind < A[ middle ] )
```

Then the loop would exit

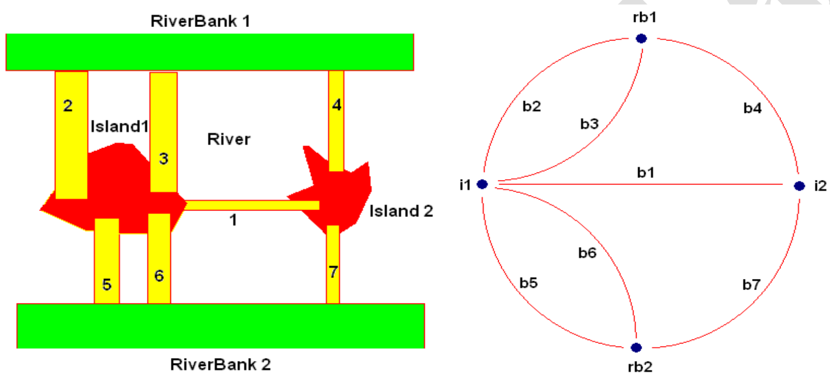
```
            last = middle – 1;
    else // itemToFind > A [middle]
            first = middle + 1;
    }
```

<u>The Theory of State Space Search</u>

Representing a problem is achieved by using State Space Graph (Graph Theory) to analyze the structure and complexity of both the problem and the search procedures that we employ to solve it. A graph consists of a set of nodes and a set of arcs or links connecting pairs of nodes.

<u>Example</u>: The “Bridges of Konigsberg problem”

“The city of Konigsberg occupied both banks and two islands of a river. The islands and the riverbanks were connected by seven bridges”, is indicated in the following figure:



A state space search is represented by a four tuple [N,A,S,GD], where:

- **N** is the set of nodes or states of the graph. These correspond to the states in a problem-solving process.
- **A** is the set of arcs (or links) between nodes. These correspond to the steps in a problem-solving process
- **S**, a nonempty subset of N, contains the start state(s) of the problem.
- **GD**, a nonempty subset of N, contains the goal state(s) of the problem. The states in GD are described using either:
    1. A measurable property of the states (nodes) encountered in the search.
    2. A property of the path (subset of arcs) developed in the search, for example, the transition costs for the arcs of the path.

Solution path is a path through this graph from a node in S to a node in GD

III. <u>Depth First Search (DFS)</u>

It is an algorithm for traversing or searching a tree, which is classified as an uninformed search that does not take into account the specific nature of the problem. The drawback is that most search spaces are extremely large, and an uninformed search (especially of a tree) will take a reasonable amount of time only for small examples. DFS is implemented using stack

Function DFS

Begin

Open: = [start]; Closed: = [ ];

```
While open <> [ ] do
Begin
    Remove leftmost state from open, call it X;
    If X is a goal then return SUCCESS
    Else Begin
        Generate children of X;
        Put X on closed;
        Discard children of X if already in open OR closed;
        Put remaining children on left end of open
        End
    End;
Return FAIL;
End.
```

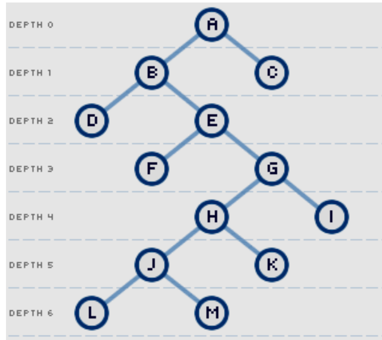
IV. *Breadth First Search (BFS)*

An algorithm for traversing or searching a tree, tree structure, or graph. BFS is implemented using queue.

```
Function BFS;
Begin
Open:=[start]; Closed:=[ ];
While open <> [ ] do
Begin
Remove leftmost state from open, call it X;
If X is a goal then return SUCCESS
Else begin
    Generate children of X;
    Put X on closed;
    Discard children of X if already in open OR closed;
    Put remaining children on right end of open
    End
End;
Return FAIL;
End.
```

Example:

Consider the following State Space, A is the start node and G is the goal. Trace DFS and BFS , writing the contents of both open and closed sets, as well as the output result.


DEPTH 0
DEPTH 1
DEPTH 2
DEPTH 3
DEPTH 4
DEPTH 5
DEPTH 6
A
B
C
D
E
F
G
H
I
J
K
L
M