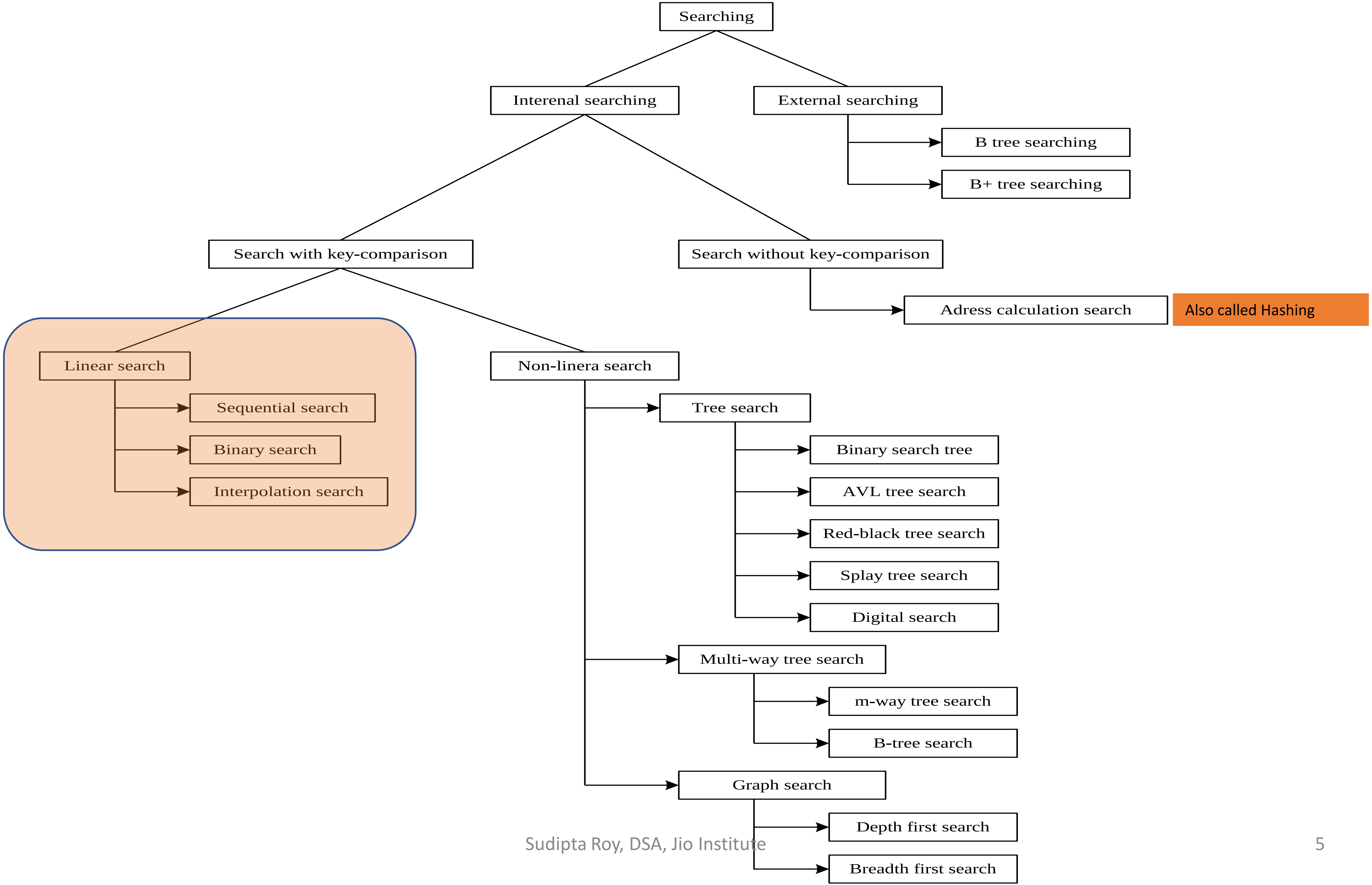
Searching: Linear and Binary Searching Interpolation Search

Searching Techniques

Today’s discussion…

- Searching Techniques
  - Sequential search with arrays
  - Binary search
  - Interpolation search
- Sequential search with linked lists

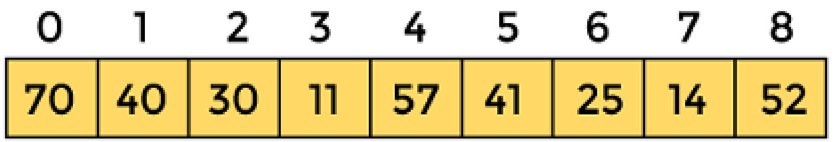
Searching Techniques


Searching
Interenal searching
External searching
B tree searching
B+ tree searching
Search with key-comparison
Search without key-comparison
Adress calculation search
Also called Hashing
Linear search
Sequential search
Binary search
Interpolation search
Non-linera search
Tree search
Binary search tree
AVL tree search
Red-black tree search
Splay tree search
Digital search
Multi-way tree search
m-way tree search
B-tree search
Graph search
Depth first search
Breadth first search

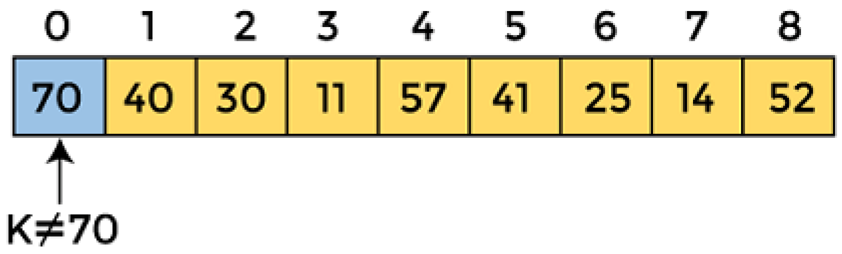
Linear Search

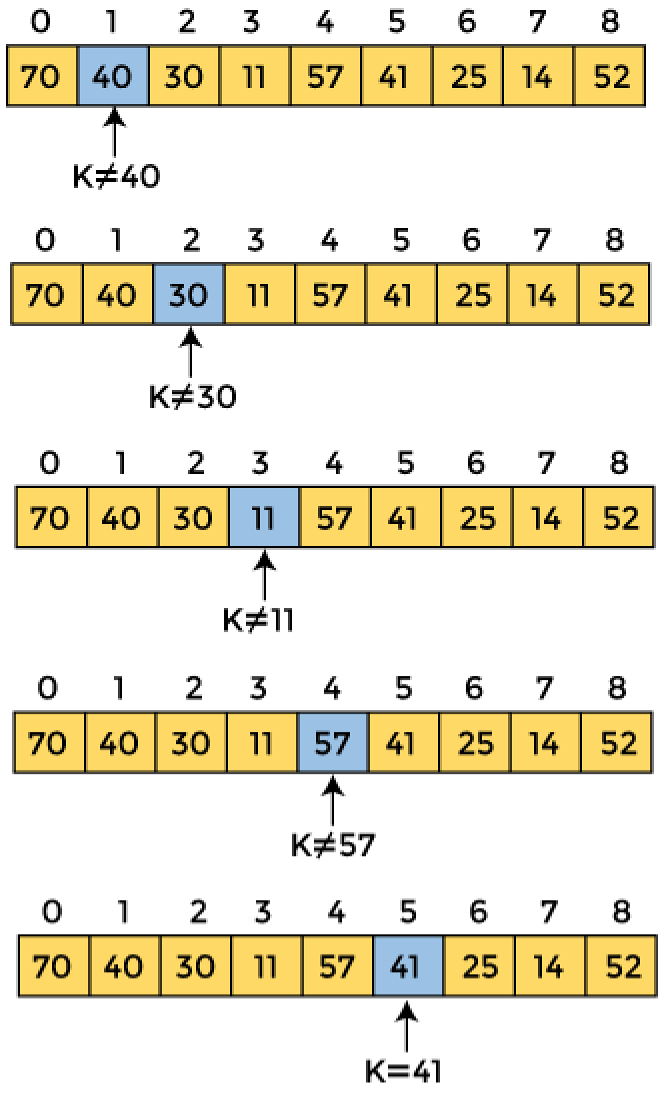
Sequential Search with Arrays

Example

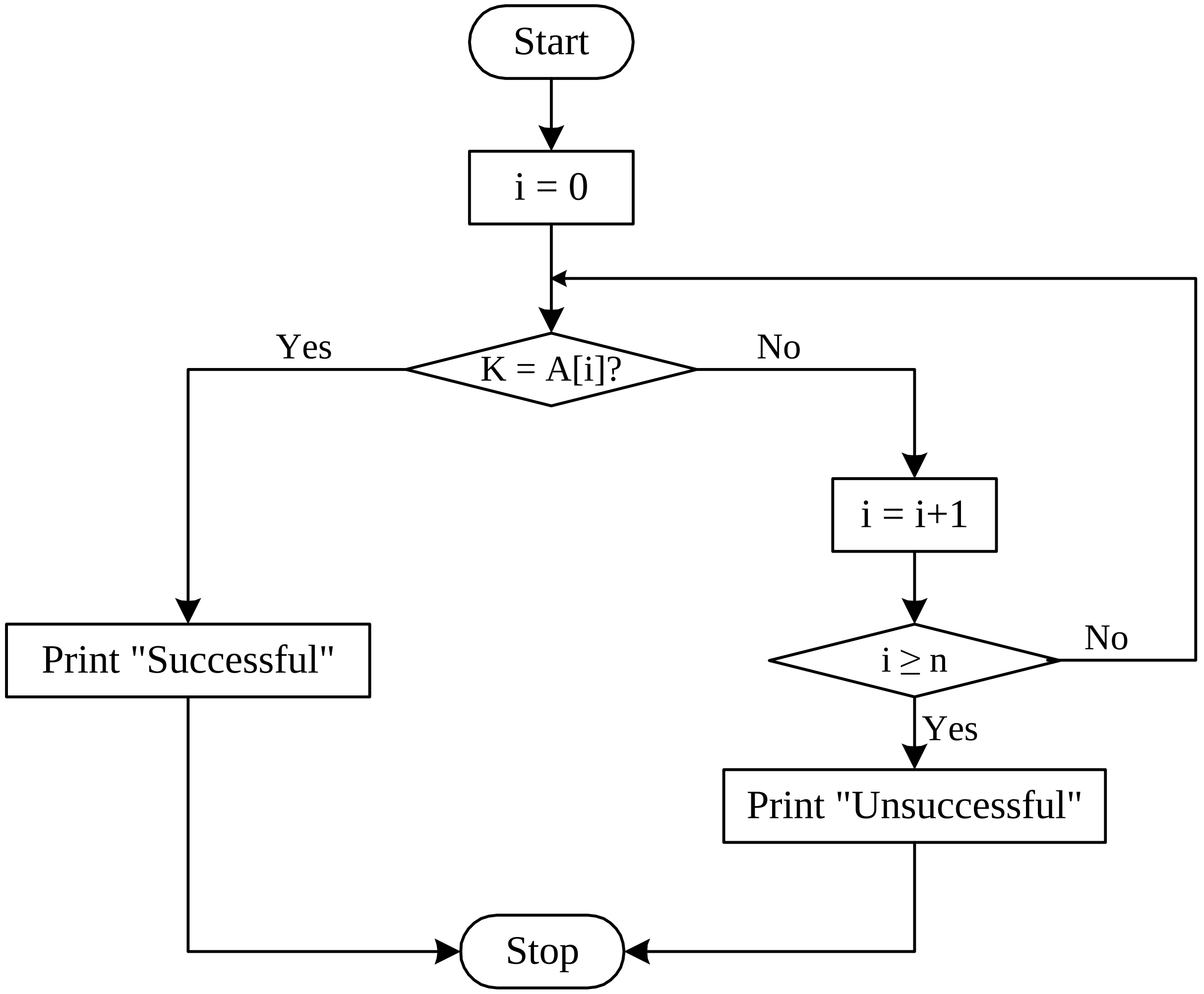


Let the element to be searched is **K = 41**





Flowchart: Sequential Search with Array



Algorithm / Procedure

Linear Search (Array Arr, Value a)
// Arr is the name of the array, and a is the searched element.

Step 1: Set i to 0 // i is the index of an array which starts from 0
Step 2: if i > n then go to step 7 // n is the number of elements in array
Step 3: if Arr[i] = a then go to step 6
    Step 4: Set i to i + 1
    Step 5: Goto step 2
Step 6: Print element a found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Example: Sequential Search with Array

```c
int main()
{
    int A[10], i, n, K, flag = 0;
    printf("Enter the size of an array: ");
    scanf("%d",&n);

    printf("Enter the elements of the array: ");
    for(i=0; i < n; i++)
         scanf("%d",&A[i]);
    printf("Enter the number to be searched: ");
    scanf("%d",&K);
    for(i=0;i<n;i++){
         if(a[i] == K){
             flag = 1; break;
         }
    }
    if(flag == 0)
        printf("The number is not in the list");
    else
        printf("The number is found at index %d",i);
    return 0;
}
```

```python
print("Enter 10 Numbers: ")
arr = []
for i in range(10):
        arr.insert(i, int(input()))  // arr.append(input())

print("Enter the Number to Search: ")
num = int(input())
for i in range(10):
        if num==arr[i]:
                index = i
                break
print("\nNumber Found at Index Number: ")
print(index)
```

Complexity Analysis

- Case 1: The key matches with the first element
  - T(n) = 1
- Case 2: Key does not exist
  - T(n) = n
- Case 3: The key is present at any location in the array

$$T(n) = \sum_{i=1}^{n} p_i \cdot i$$

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} i \qquad\qquad p_1 = p_2 = \cdots p_i = \cdots p_n = \frac{1}{n}$$

$$T(n) = \frac{n+1}{2}$$

Complexity Analysis : Summary

Case	Number of key comparisons	Asymptotic complexity	Remark
Case 1	$T(n) = 1$	$T(n) = O(1)$	Best case
Case 2	$T(n) = n$	$T(n) = O(n)$	Worst case
Case 3	$T(n) = \frac{n+1}{2}$	$T(n) = O(n)$	Average case

Uses

- **Advantages of Linear Search:**
  - Linear search is simple to implement and easy to understand.
  - Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.
  - Does not require any additional memory.
  - It is a well-suited algorithm for small datasets.
- **Drawbacks of Linear Search:**
  - Linear search has a time complexity of O(n), which in turn makes it slow for large datasets.
  - Not suitable for large array.
  - Linear search can be less efficient than other algorithms, such as hash tables.
- **When to use Linear Search:**
  - When we are dealing with a small dataset.
  - When you need to find an exact value.
  - When you are searching a dataset stored in contiguous memory.
  - When you want to implement a simple algorithm.

```
# Pseudocode : Sum(a, b) { return a + b }
a = 5
b = 6
def sum(a,b):
  return a+b
 # function call
print(sum(a,b))
```

Time Complexity:

- The above code will take 2 units of time(constant):
  - one for arithmetic operations and
  - one for return. (as per the above conventions).
- Therefore, total cost to perform sum operation (**Tsum)** = 1 + 1 = 2
- **Time Complexity = O(2) = O(1)**, since 2 is constant

```
Pseudocode : list_Sum(A, n)
{
total =0
for i=0 to n-1
    sum = sum + A[i]

return sum

}
```

```
Pseudocode : list_Sum(A, n)
{
total =0                    // cost=1  no of times=1
for i=0 to n-1             // cost=2  no of times=n+1 (+1 for the end false condition)
    sum = sum + A[i]        // cost=2  no of times=n
return sum                 // cost=1  no of times=1
}
```

O(n)

```python
n = 3
m = 3
arr = [[3, 2, 7], [2, 6, 8], [5, 1, 9]]
sum = 0

# Iterating over all 1-D arrays in 2-D array
for i in range(n):
    # Printing all elements in ith 1-D array
    for j in range(m):
        # Printing jth element of ith row
        sum += arr[i][j]

print(sum)
```

Time Complexity: O(n*m)
The program iterates through all the elements in the 2D array using two nested loops. The outer loop iterates n times and the inner loop iterates m times for each iteration of the outer loop. Therefore, the time complexity of the program is O(n*m).

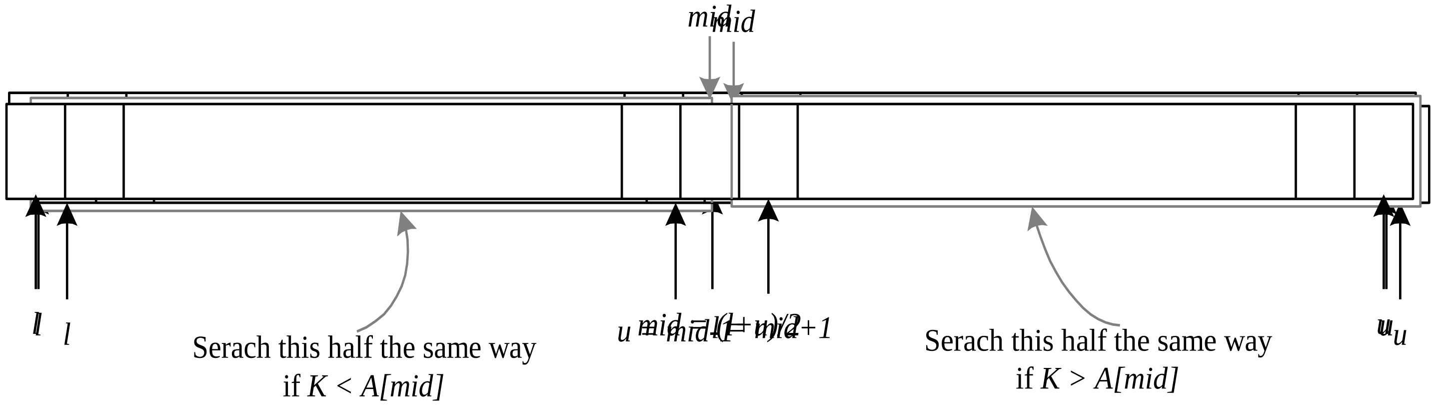
```c
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i=i*2) {
        printf("Hello World !!!\n");
    }
}
```

```python
n = 8
# for (i = 1; i <= n; i=i*2) {
for i in range(1,9,+2):
    print("Hello World !!!")
```

Time Complexity: $O(\log_2(n))$

Binary Search

The Technique



(a) An ordered array of elements with index values l, u and mid

(b) Search the entire list turns into the searching of left-half only

(c) Search the entire list turns into the searching of right-half only

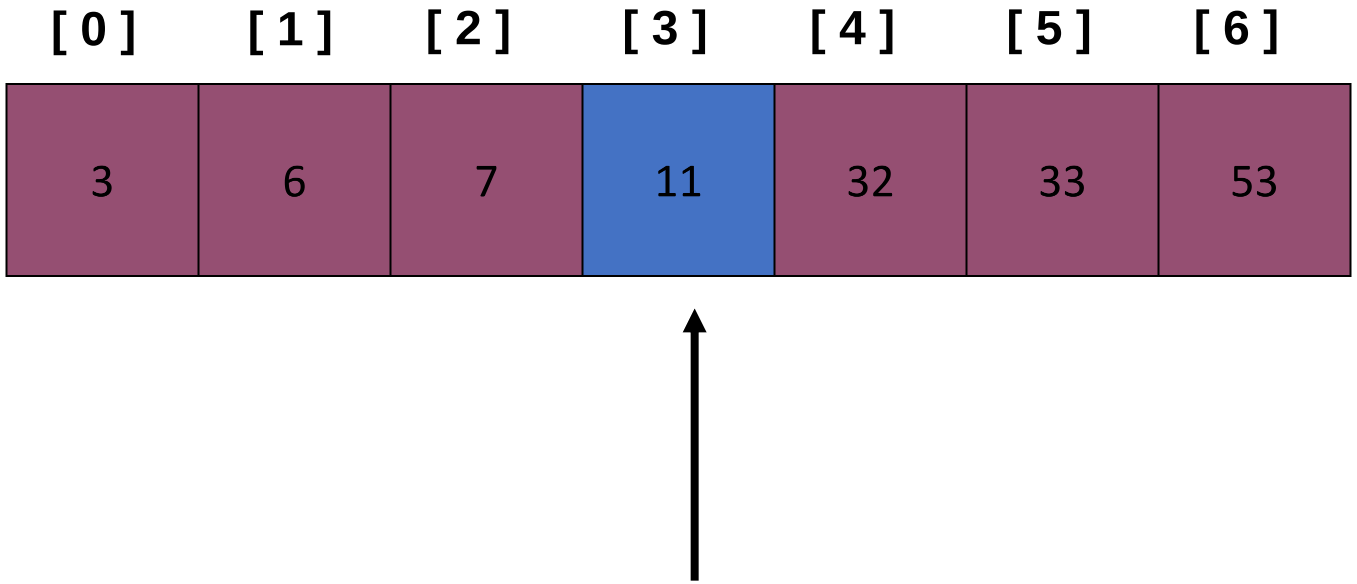
Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.



Find approximate midpoint

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is 7 = midpoint key? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is 7 < midpoint key? YES.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search for the target in the area before midpoint.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Target = key of midpoint? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Target < key of midpoint? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Target > key of midpoint? YES.

Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search for the target in the area after midpoint.

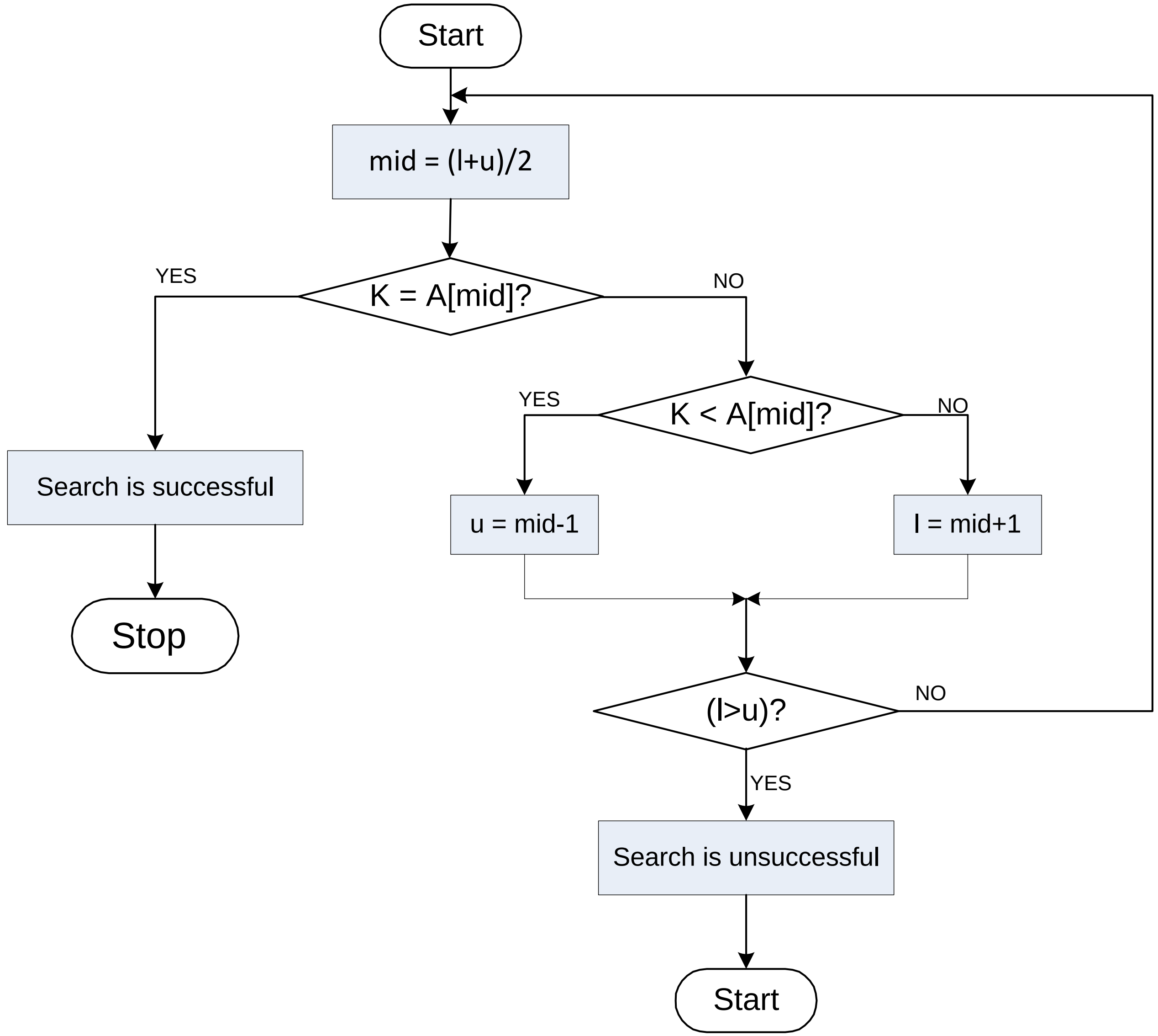
Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint.
Is target = midpoint key? YES.

Flowchart: Binary Search with Array



Algorithm/ Procedure

Given an array A of n elements with values or records $A_0, A_1, A_2, \ldots, A_{n-1}$ sorted such that $A_0 \leq A_1 \leq A_2 \leq \cdots \leq A_{n-1}$, and target value T, the following subroutine uses binary search to find the index of T in A.

1. Set L to 0 and R to $n-1$.
2. If $L > R$, the search terminates as unsuccessful.
3. Set m (the position of the middle element) to the floor of $\frac{L+R}{2}$, which is the greatest integer less than or equal to $\frac{L+R}{2}$.
4. If $A_m < T$, set L to $m+1$ and go to step 2.
5. If $A_m > T$, set R to $m-1$ and go to step 2.
6. Now $A_m = T$, the search is done; return m.

This iterative procedure keeps track of the search boundaries with the two variables L and R. The procedure may be expressed in pseudocode as follows, where the variable names and types remain the same as above,

```
function binary_search(A, n, T) is
    L := 0
    R := n − 1
    while L ≤ R do
        m := floor((L + R) / 2)
        if A[m] < T then
            L := m + 1
        else if A[m] > T then
            R := m − 1
        else:
            return m
    return unsuccessful
```

```
function binary_search_alternative(A, n, T) is
    L := 0
    R := n − 1
    while L != R do
        m := ceil((L + R) / 2)
        if A[m] > T then
            R := m − 1
        else:
            L := m
    if A[L] = T then
        return L
    return unsuccessful
```

Duplicate elements

The procedure may return any index whose element is equal to the target value, even if there are duplicate elements in the array. For example, if the array to be searched was [1,2,3,4,4,5,6,7] and the target was 4 4, then it would be correct for the algorithm to either return the 4th (index 3) or 5th (index 4) element

```
function binary_search_leftmost(A, n, T):
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] < T:
            L := m + 1
        else:
            R := m
    return L
```

```
function binary_search_rightmost(A, n, T):
    L := 0
    R := n
    while L < R:
        m := floor((L + R) / 2)
        if A[m] > T:
            R := m
        else:
            L := m + 1
    return R - 1
```

Binary Search (with Iteration)

```
#include <stdio.h>

int main()
{
   int i, l, u, mid, n, K, data[100];

   printf("Enter number of elements\n");
   scanf("%d",&n);

   printf("Enter %d integers in sorted order\n", n);

   for (i = 0; i < n; i++)
      scanf("%d",&array[i]);

   printf("Enter value to find\n");
   scanf("%d", &K);

   l = 0;
   u = n - 1;
   mid = (l+u)/2;
```

Contd…

Binary Search (with Iteration)

```
while (l <= u) {
      if (data[mid] < K)
         l = mid + 1;
      else if (data[mid] == K) {
         printf("%d found at location %d.\n", search, mid+1);
         break;
      }
      else
         u = mid - 1;

      mid = (l + u)/2;
   }
   if (l > u)
      printf("Not found! %d is not present in the list.\n", K);

   return 0;
}
```

Binary Search (with Recursion)

```
#include<stdio.h>
int main(){

    int data[100],i, n, K, flag, l, u;

    printf("Enter the size of an array: ");
    scanf("%d",&n);

    printf("Enter the elements of the array in sorted order: " );
    for(i=0;i<n;i++)
         scanf("%d",&a[i]);

    printf("Enter the number to be search: ");
    scanf("%d",&K);

    l=0,u=n-1;
    flag = binarySearch(data,n,K,l,u);
    if(flag==0)
         printf("Number is not found.");
    else
         printf("Number is found.");

    return 0;
 }
```

Contd…

Binary Search (with Recursion)

```
int binary(int a[],int n, int K, int l, int u){

      int mid;

      if(l<=u){
            mid=(l+u)/2;
            if(K==a[mid]){
                 return(1);
            }
            else if(m<a[mid]){
                 return binarySearch(a,n,K,l,mid-1);
            }
            else
                 return binarySearch(a,n,m,mid+1,u);
      }
      else return(0);
}
```

```python
nums = []
print(end="How Many Number to Enter ? ")
tot = int(input())
print(end="Enter " + str(tot) + " Numbers: ")
for i in range(tot):
        nums.insert(i, int(input()))

for i in range(tot-1):
    for j in range(tot-i-1):
        if nums[j]>nums[j+1]:
            temp = nums[j]
            nums[j] = nums[j+1]
            nums[j+1] = temp

print(end="\nThe List is: ")
for i in range(tot):
    print(end=str(nums[i]) + " ")
```

```python
print(end="\nEnter a Number to Search: ")
search = int(input())
first = 0
last = tot-1
middle = int((first+last)/2)
while first <= last:
    if nums[middle]<search:
        first = middle+1
    elif nums[middle]==search:
        print("\nThe Number Found at Position: " + str(middle+1))
        break
    else:
        last = middle-1
    middle = int((first+last)/2)
if first>last:
    print("\nThe Number is not Found in the List")
```

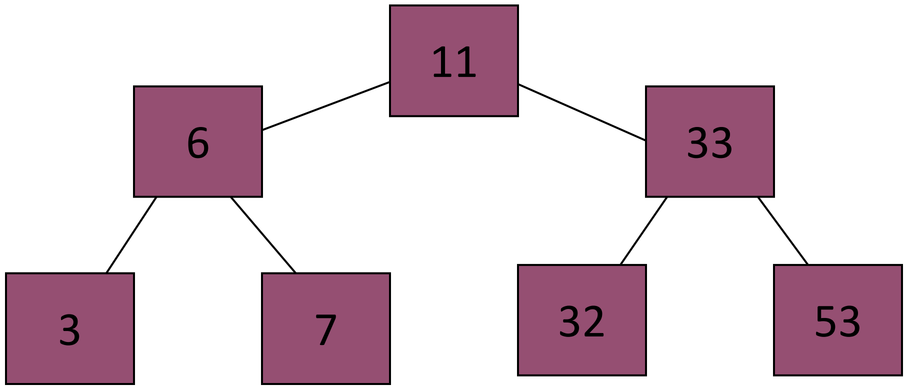
Relation to Binary Search Tree

Relation to Binary Search Tree

Array of previous example:

3	6	7	11	32	33	53

Corresponding complete binary search tree

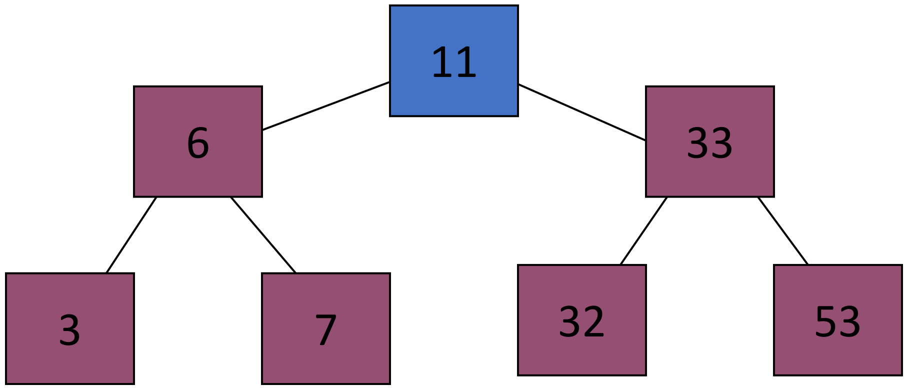


Search for target = 7

Find midpoint:

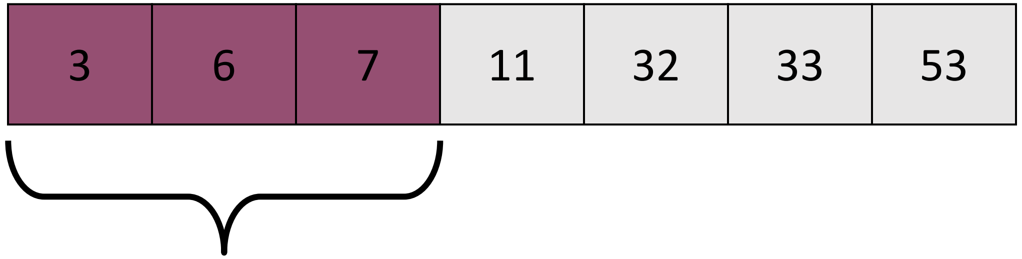
3	6	7	11	32	33	53

Start at root:

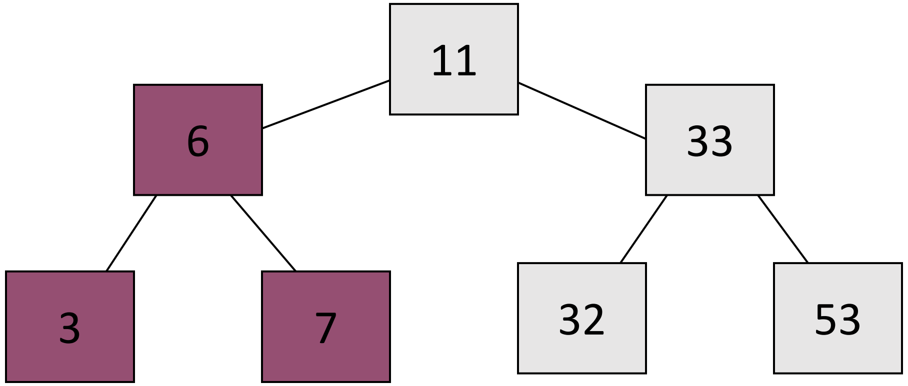


Search for target = 7

Search left subarray:

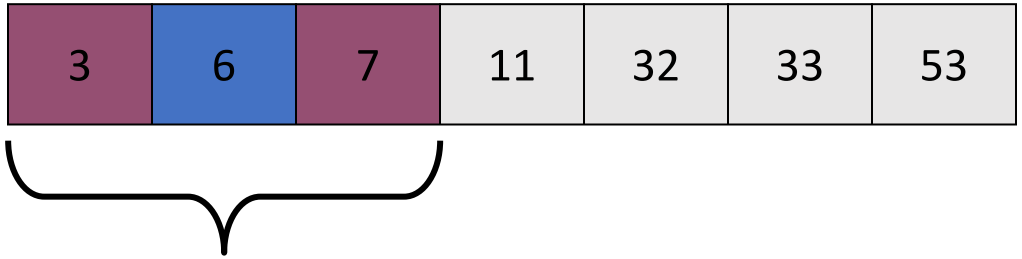


Search left subtree:

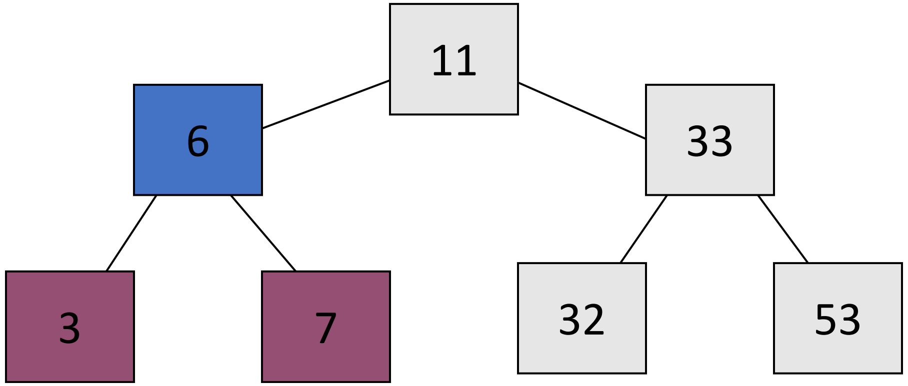


Search for target = 7

Find approximate midpoint of subarray:

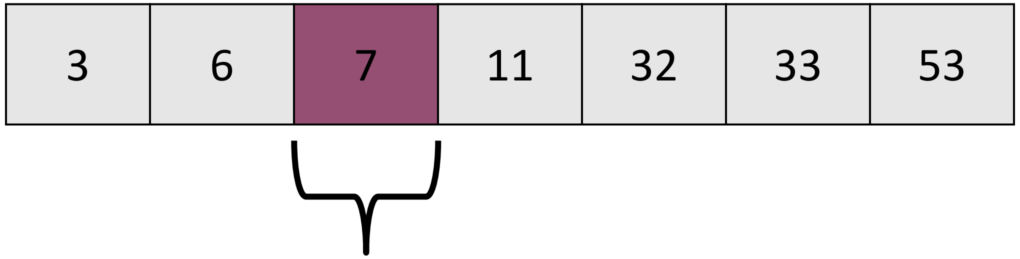


Visit root of subtree:

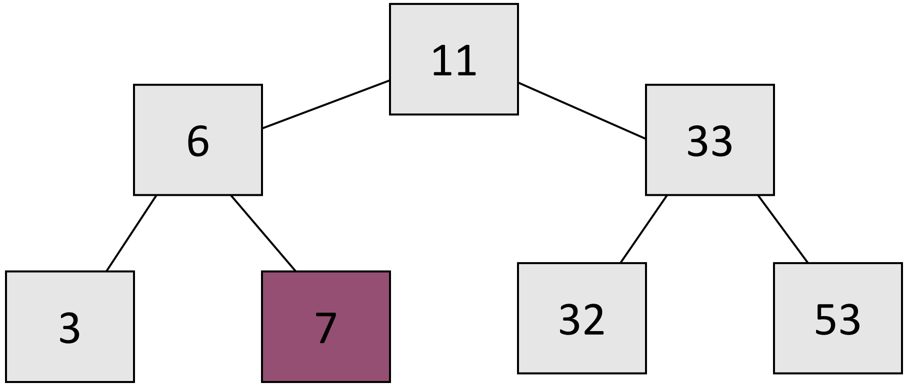


Search for target = 7

Search right subarray:



Search right subtree:



Another example of Binary Search

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.
- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo														↑ hi

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.
- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lo							mid							hi

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.
- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				lo		hi								

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑ lo hi										

Binary Search

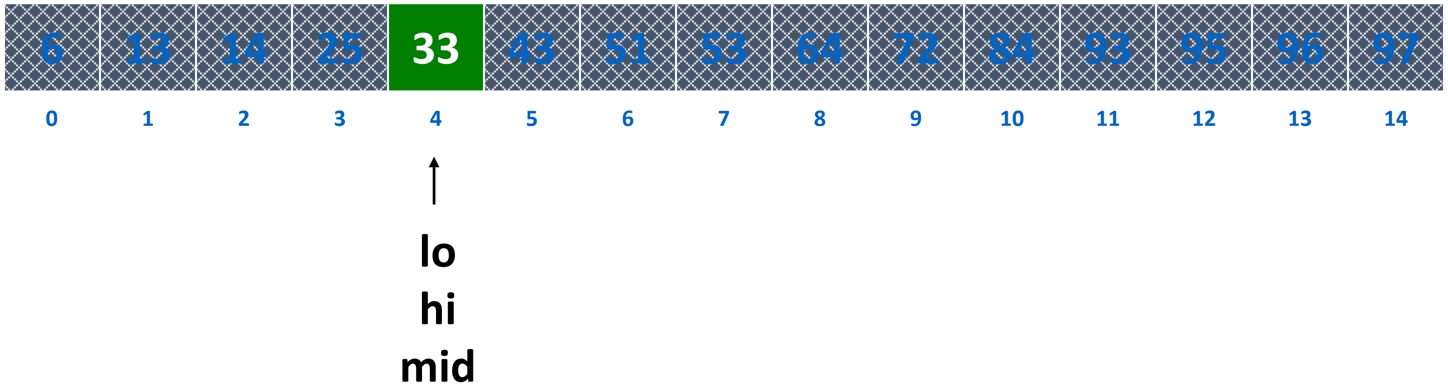
- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.
- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑ (index 4)
lo
hi
mid

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.
- Ex. Binary search for 33.



Binary Search: Analysis

- Worst case complexity?
- What is the maximum depth of recursive calls in binary search as function of *n*?
- Each level in the recursion, we split the array in half (divide by two).
- Therefore, maximum recursion depth is floor($\log_2 n$) and worst case = O($\log_2 n$).
- Average case is also = O($\log_2 n$).

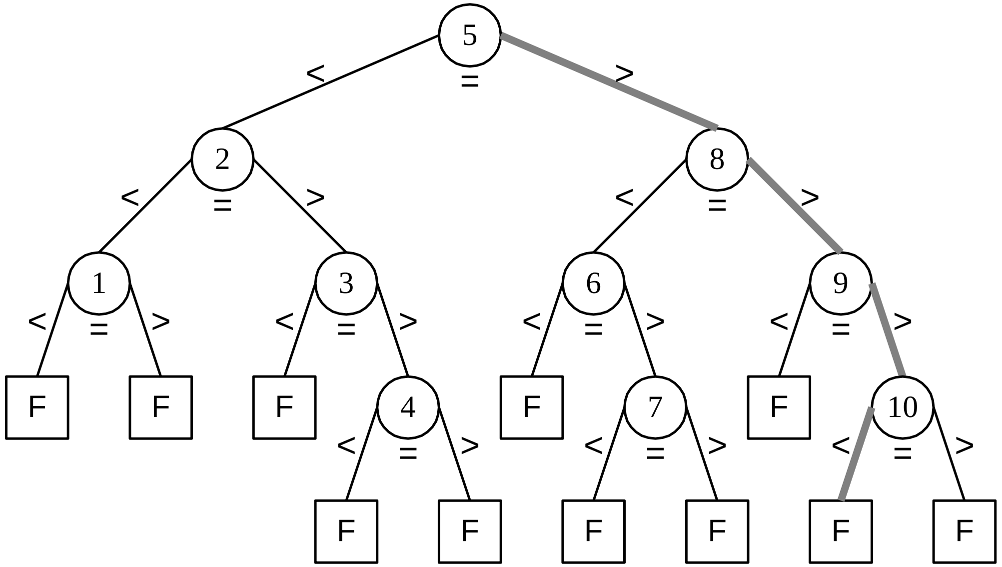
Binary search -Complexity

Récurrence relation is $T(n) = T(n/2) + 1$

where $T(n)$ is the time required for binary search in an array of size n.

$T(n) = T(n/2^k) + 1 + \cdots + 1$

Since $T(1) = 1$, when $n = 2^k$, $T(n) = T(1) + k = 1 + \log_2(n)$. $\log_2(n) \leq 1 + \log_2(n) \leq 2\log_2(n)$, $\forall\, n \geq 2$.
$T(n) = \Theta(\log_2(n))$



Similar to binary search, what will be the complexity for ternary search ?

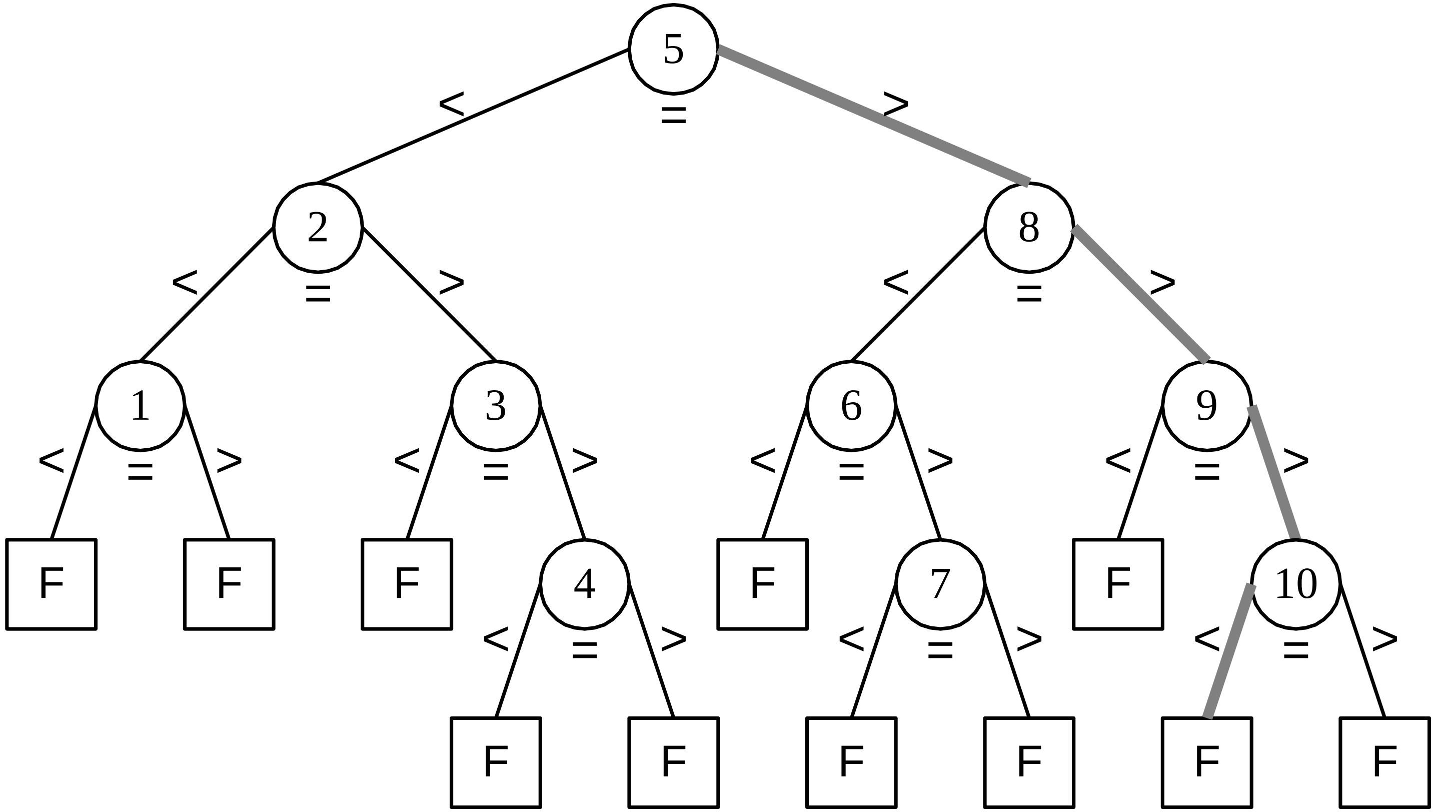
Can we do better than $O(\log_2 n)$?

- Average and worst case of serial search = $O(n)$
- Average and worst case of binary search = $O(\log_2 n)$

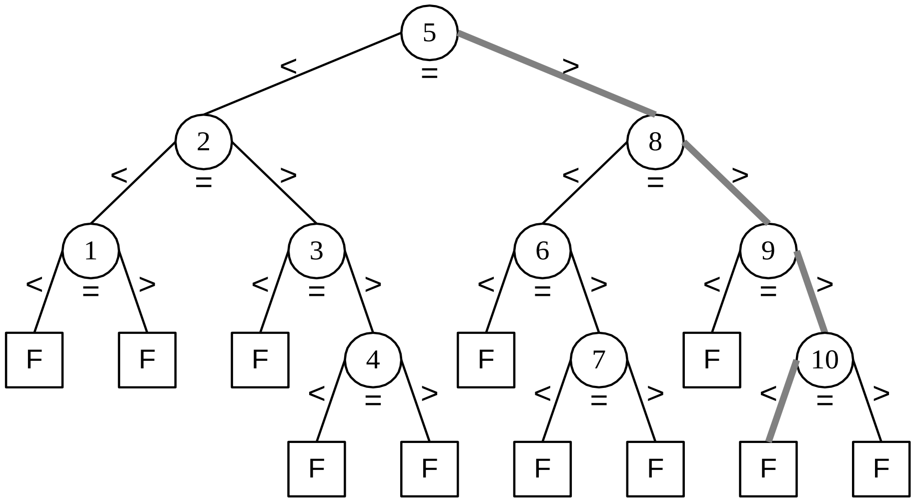
- Can we do better than this?

YES. Use a hash table!

Complexity Analysis



Complexity Analysis: Binary Search



Let n be the total number of elements in the list under search and there exist an integer k such that:-

- For successful search:-
  - If $2^{k-1} \le n < 2^k$, then the binary search algorithm requires at least one comparison and at most k comparisons.
- For unsuccessful search:-
  - If $n = 2^{k-1}$, then the binary search algorithm requires k comparisons.
  - If $2^{k-1} \le n < 2^k - 1$, then the binary search algorithm requires either k-1 or k number of comparisons.

Complexity-Hight of Binary search tree

- The total number of nodes, n, in the tree is equal to the sum of the nodes on all the levels: $1 + 2^1 + 2^2 + 2^3 + \ldots + 2^{h-1} = n$
- We know that: $1 + 2^1 + 2^2 + 2^3 + \ldots + 2^{h-1} = 2^h - 1$

  Therefore:

  $2^h - 1 = n$

  $2^h = n + 1$

  $\log_2 2^h = \log_2(n + 1)$

  $h \log_2 2 = \log_2(n + 1)$

  $h = \log_2(n + 1)$

- **Therefore, h is O(log n)**

Interpolation Search

Interpolation Search

$l = 1, u = n$ // Initialization: Range of searching

flag = FALSE // Hold the status of searching

While (flag = FALSE) **do**

    $loc = \left\lceil \frac{K - A[l]}{A[u] - A[l]} \right\rceil \times (u - l) + l$

    If ($l \leq loc \leq u$) **then** // If loc is within the range of the list

        Case: K < A[loc]

            u = loc -1

        Case: K = A[loc]

            flag = TRUE

        Case: K > A[loc]

            l = loc +1

    Else

        Exit()

    EndIf

EndWhile

If (flag) **then**

    Print “Successful at” loc

Else

    Print “Unsuccessful”

EndIf

Stop

K Element to be searched

Any question?

Problems to ponder…

1. What will be the time complexity of linear search with array if the array is already in sorted order?
2. What will be the outcome, if Binary search technique is applied to an array where the data are not necessarily in sorted order?
3. Whether the Binary search technique is applicable to a linked list? If so, how?
4. In Binary Search, the mid location is calculated and then either left or right part of the mid location is searched further, if there is no match at the middle is found. As an alternative to check at middle, the location at one-third (or two-third) position of the array is chosen. Such a searching can be termed as Ternary Search. Modify the Binary Search algorithm to Ternary Search algorithm. Which algorithm gives result faster?
5. If T(n) denotes the number of comparisons required to search a key element in a list of n elements, then a) express T(n) as recursion formula and b) solve T(n).

Problems to ponder…

6. Out of sequential search and binary search, which is faster? Why?

7. Whether binary search technique can be applied to search a string in a list od strings stored in an array? If so, revise the Binary search algorithm accordingly.

If you try to solve problems yourself, then you will learn many things automatically.

Spend few minutes and then enjoy the study.