Lecture 1

Introduction to Data Structure and Arrays
(Traversal, Insertion, Deletion, Searching)

Data Structure

- A particular way of storing and organizing data in a computer so that it can be used efficiently.
- They provide a means to manage large amounts of data efficiently, such as large databases.
- Data are simply values or set of values and Database is organized collection of data.

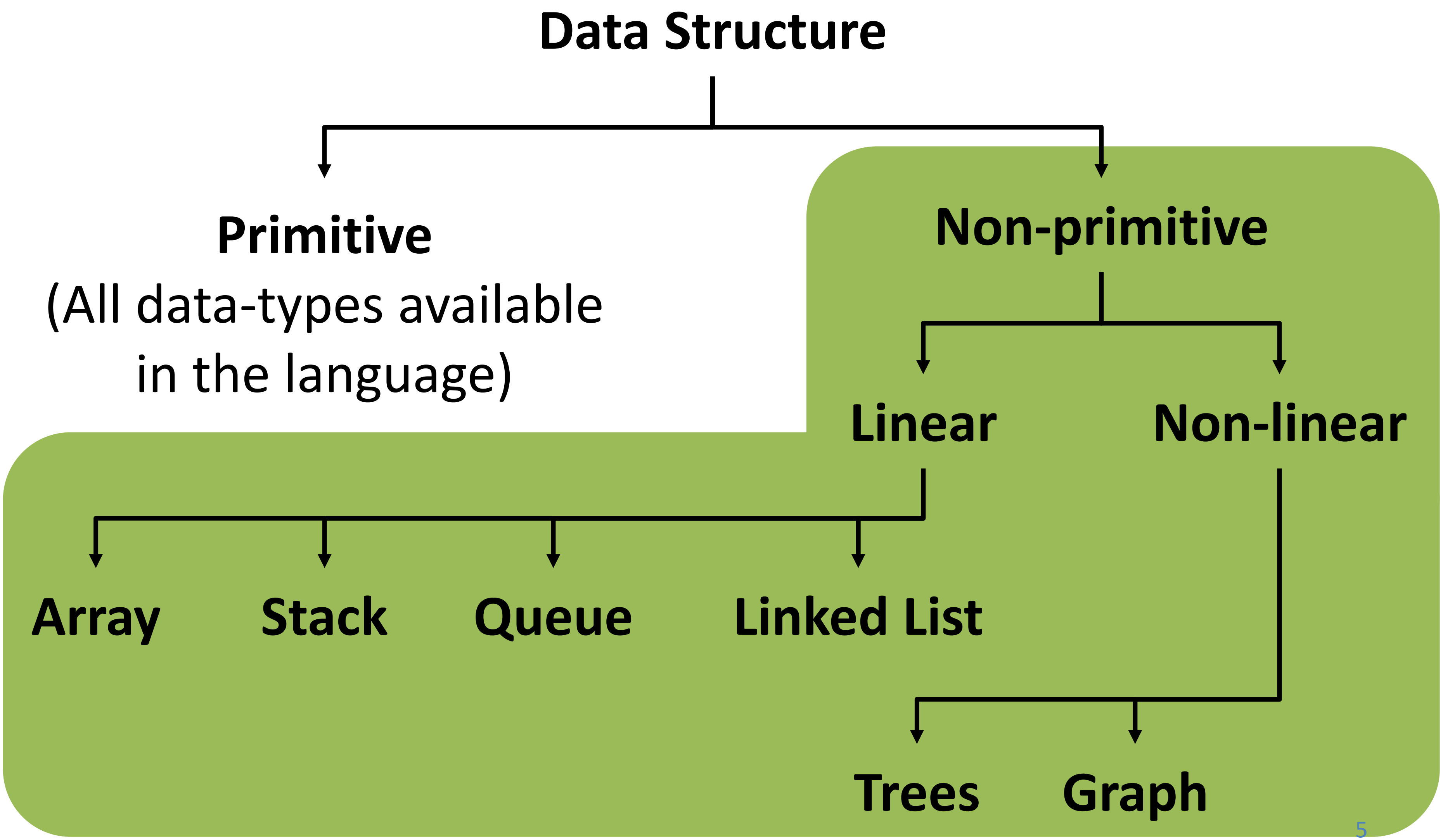
Abstract Data Type (ADT)

- Realization of a data type as a software component, i.e. focus on **what** can be done with the data, not **how** it is done.
- The interface of the ADT is defined in terms of a type and a set of operations on that type.
- Implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation.

Study of Data Structure Includes

- Logical description of data structure.
- Implementation of data structure.
- Quantitative analysis of data structure, this include amount of memory, processing time

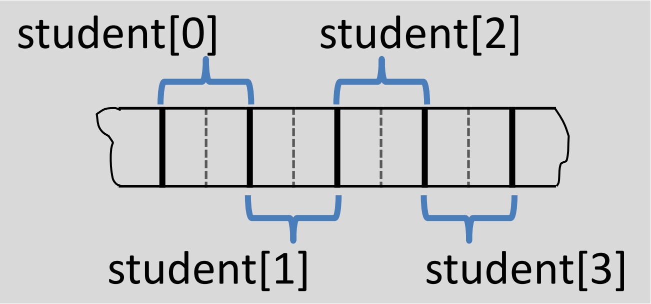
Classification of Data Structures



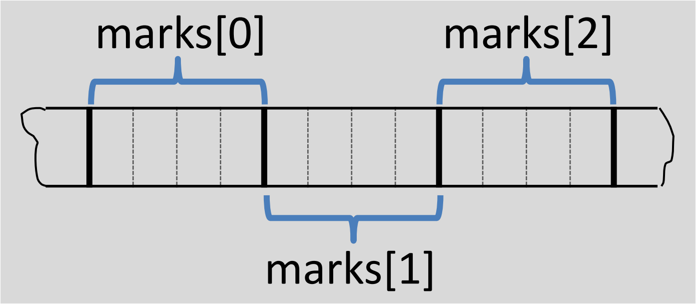
Array

Memory Storage – One Dimensional Array

```
int student[4];
```




```
float marks[3];
```



Memory Storage – Two Dimensional Array

```
int marks[3][5];
```

- Can be visualized in the form of a matrix as

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	marks[0][0]	marks[0][1]	marks[0][2]	marks[0][3]	marks[0][4]
Row 1	marks[1][0]	marks[1][1]	marks[1][2]	marks[1][3]	marks[1][4]
Row 2	marks[2][0]	marks[2][1]	marks[2][2]	marks[2][3]	marks[2][4]

Contd…

- Row-major order

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)

Row0 Row1 Row2

- Column-major order

(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)	(0,3)	(1,3)	(2,3)	(0,4)	(1,4)	(2,4)

Co10 Col1 Col2 Col3 Col4

Array ADT

- The simplest but useful data structure.
- Assign single name to a homogeneous collection of instances of one abstract data type.
  - All array elements are of same type, so that a pre–defined equal amount of memory is allocated to each one of them.
- Individual elements in the collection have an associated index value that depends on array dimension.

Contd…

- One-dimensional and two-dimensional arrays are commonly used.
- Multi–dimensional arrays can also be defined.

- Usage:
  – Used frequently to store relatively permanent collections of data.
  – Not suitable if the size of the structure or the data in the structure are constantly changing.

Operations on Linear Data Structures

- Traversal
- Search
- Insertion
- Deletion
- Merging
- Sorting

Traversal

- Processing each element in the array.
- Example: Find minimum element in the array

Algorithm arrayMinElement(A,n)
Input: An array **A** containing **n** integers.
Output: The minimum element in **A**.

		cost	time		
1.	min = 0	c1	1	$\rightarrow$	c1
2.	for i = 1 to n-1 do	c2	n	$\rightarrow$	c2 n
3.	if A[min] > A[i]	c3	n – 1	$\rightarrow$	c3 (n – 1)
4.	min = i	c4	$\sum_{i=1}^{n-1} t_i$	$\rightarrow$	$c4 \sum_{i=1}^{n-1} t_i$
5.	return A[min]	c5	1	$\rightarrow$	c5

Contd…

- $T(n) = c1 + c2\ n + c3\ (n - 1) + c4 \sum_{i=1}^{n-1} t_i\ + c5$

  $= (c1 - c3 + c5) + (c2 + c3)\ n + c4 \sum_{i=1}^{n-1} t_i$

- Best case: Ω(n) ⟶ summation evaluates to 0.
- Worst case: O(n) ⟶ summation evaluates to n.
- Average case: θ(n) ⟶ summation evaluates to n/2.

Example – Traversal

```
//Determine smallest element in an array.
#include<iostream>
using namespace std;
int arrayMinElement(int arr[], int n)
{
   int min = 0;
   for (int i = 1; i < n; i++)
   {
      if (arr[i] < arr[min])
         min = i;
   }
   return arr[min];    }
```

Contd...

```
int main()
{
    int i, n, a[10];
    cout << "Enter the array size (<=10): ";
    cin >> n;
    for (i = 0; i < n; i++)
    { cout << "Enter " << i+1 << " element: ";
       cin >> a[i];
    }
    cout << "\nSmallest element is " << arrayMinElement(a,n);
    return 0;
}
```

Insertion and Deletion

	0	1	2	3	4	5	6	7	8	9
a[]	8	6	3	4	5					

- Insert 2 at index 1

	0	1	2	3	4	5	6	7	8	9
a[]	8	~~6~~	~~3~~	~~4~~	~~5~~					
		2	6	3	4	5				

- Delete the value at index 2

	0	1	2	3	4	5	6	7	8	9
a[]	8	2	~~6~~	~~3~~	~~4~~	5				
			3	4	5					

Algorithm – Insertion

Algorithm insertElement(A,n,num,indx)

Input: An array **A** containing **n** integers and the number **num** to be inserted at index **indx**.

Output: Successful insertion of **num** at **indx**.

		cost	time
1.	for i = n – 1 to indx do	c1	n – indx + 1
2.	A[i + 1] = A[i]	c2	n – indx
3.	A[indx] = num	c3	1
4.	n = n + 1	c4	1

Contd…

- $T(n) = c1\ (n - indx + 1) + c2\ (n - indx) + c3 + c4$

- Best case: $\Omega(1) \longrightarrow indx = n - 1$
- Worst case: $O(n) \longrightarrow indx = 0$
- Average case: $\theta(n) \longrightarrow indx = n/2$

Algorithm – Deletion

Algorithm deleteElement(A,n,indx)

Input: An array **A** containing **n** integers and the index **indx** whose value is to be deleted.

Output: Deleted value stored initially at **indx**.

		cost	time
1.	temp = A[indx]	c1	1
2.	for i = indx to n – 2 do	c2	n – indx
3.	A[i] = A[i + 1]	c3	n – indx – 1
4.	n = n – 1	c4	1
5.	return temp	c5	1

Contd…

- $T(n) = c1 + c2\ (n - indx) + c3\ (n - indx - 1) + c4 + c5$

- Best case: $\Omega(1) \rightarrow indx = n - 1$
- Worst case: $O(n) \rightarrow indx = 0$
- Average case: $\theta(n) \rightarrow indx = n/2$

Example – Insertion and Deletion

```
#include<stdio.h>
int n;
void insert(int a[], int num, int pos)
{   for(int i = n-1; i >= pos; i--)                a[i+1] = a[i];
     a[pos] = num;
      n++;  }
int deleteElement(int a[], int pos)
{   int temp = a[pos];
     for(int i = pos; i <= n-2; i++)               a[i] = a[i+1];
     n--;
      return temp;  }
void printArray(int a[])
{     for (i = 0; i < n; i++)
            printf("%d ",a[i]);  }
```

Contd…

```
int main()
{  int i, a[10], num, pos;
   printf("Enter the size of an array (<10): ");
   scanf("%d",&n);
   for (i = 0; i < n; i++)
   {    printf("Enter element at index %d: ",i);
        scanf("%d",&a[i]);    }
  printf("\nEnter number to be inserted: ");
  scanf("%d",&num);
  printf("Enter the desired index: ");
  scanf("%d",&pos);
  insert(a,num,pos);
```

Contd...

```
    printf("\nArray after insertion is...\n");
    printArray(a);
    printf("\nEnter the index whose value is to be deleted: ");
    scanf("%d",&pos);
    printf("\nThe deleted element is %d.", deleteElement(a,pos));
    printf("\nArray after deletion is...\n");
    printArray(a);
    return 0;
}
```

Search

Find the location of the element with a given value.

Linear Search

- Used if the array is unsorted.
- Example:

Search 7 in the following array

i→0→1→2→3→4→5→6

a[]	10	5	1	6	2	9	7	8	3	4

Found at index 6

Search 11 in the following array

i→0→1→2→3→4→5→6→7→8→9→10

a[]	10	5	1	6	2	9	7	8	3	4

Not found

Contd…

Algorithm linearSearch(A,n,num)
Input: An array **A** containing **n** integers and number **num** to be searched.
Output: Index of **num** if found, otherwise -1.

	cost	time	
1. for i = 0 to n-1 do	c1	$\sum_{i=0}^{n} t_i$	$\rightarrow$ c1 $\sum_{i=0}^{n} t_i$
2. if A[i] == num	c2	$\sum_{i=0}^{n-1} t_i$	$\rightarrow$ c2 $\sum_{i=0}^{n-1} t_i$
3. return i	c3	1	$\rightarrow$ c3
4. return -1	c4	1	$\rightarrow$ c4

Contd…

- $T(n) = c1\sum_{i=0}^{n} t_i + c2\sum_{i=0}^{n-1} t_i + (c3 + c4)$

- Best case: $\Omega(1) \longrightarrow$ summation evaluates to 1.
- Worst case: $O(n) \longrightarrow$ summation evaluates to n.
- Average case: $\theta(n) \longrightarrow$ summation evaluates to n/2.

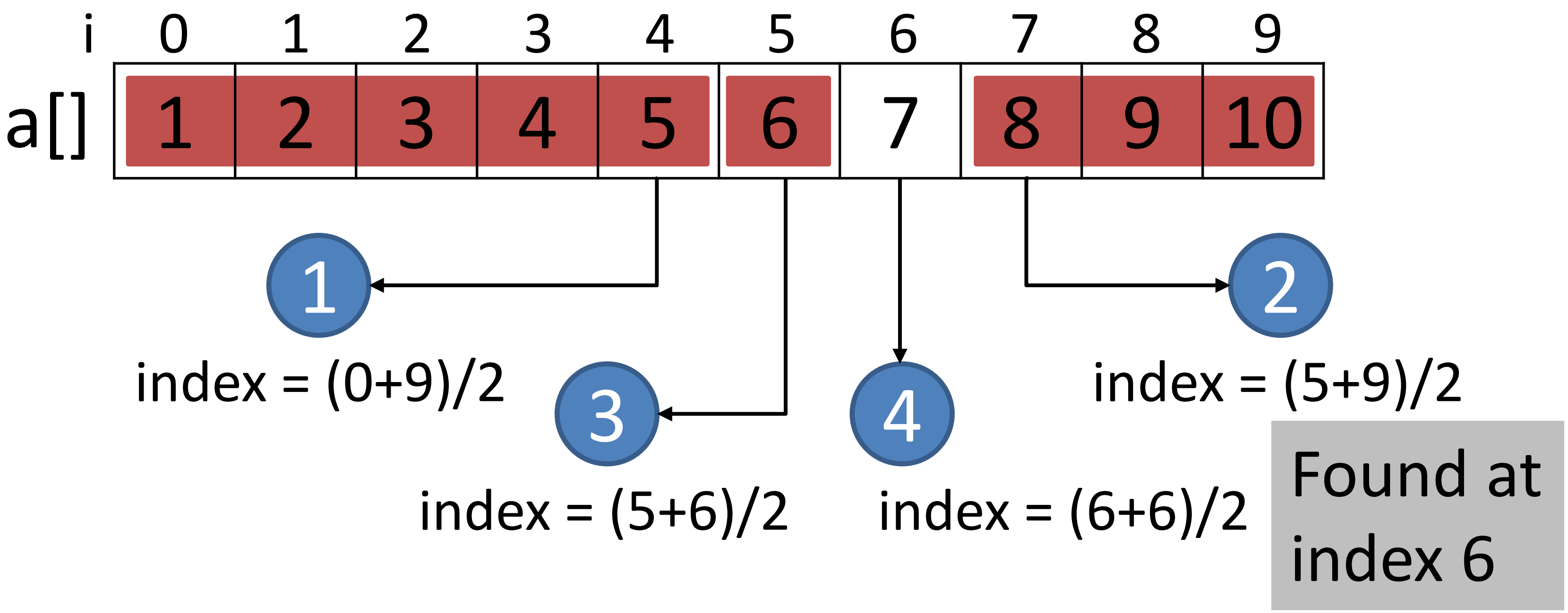
Example – Linear Search

```
#include<stdio.h>
int linearSearch(int a[], int n, int num)
{  for (int i = 0; i < n; i++)
       if (a[i] == num)  return i;
     return -1;  }
int main()
{  int i, n, a[10], num;
   printf("Enter the size of an array (<=10): ");      scanf("%d",&n);
   for (i = 0; i < n; i++)
   {  printf("Enter element at index %d: ",i);          scanf("%d", &a[i]);  }
      printf("\nEnter number to search: ");             scanf("%d",&num);
      int found = linearSearch(a,n,num);
      if(found != -1)   printf("Element found at index %d",found);
      else              printf("Element not found.");
      return 0;  }
```

Binary Searching

- Used if the array is sorted.
- Example:

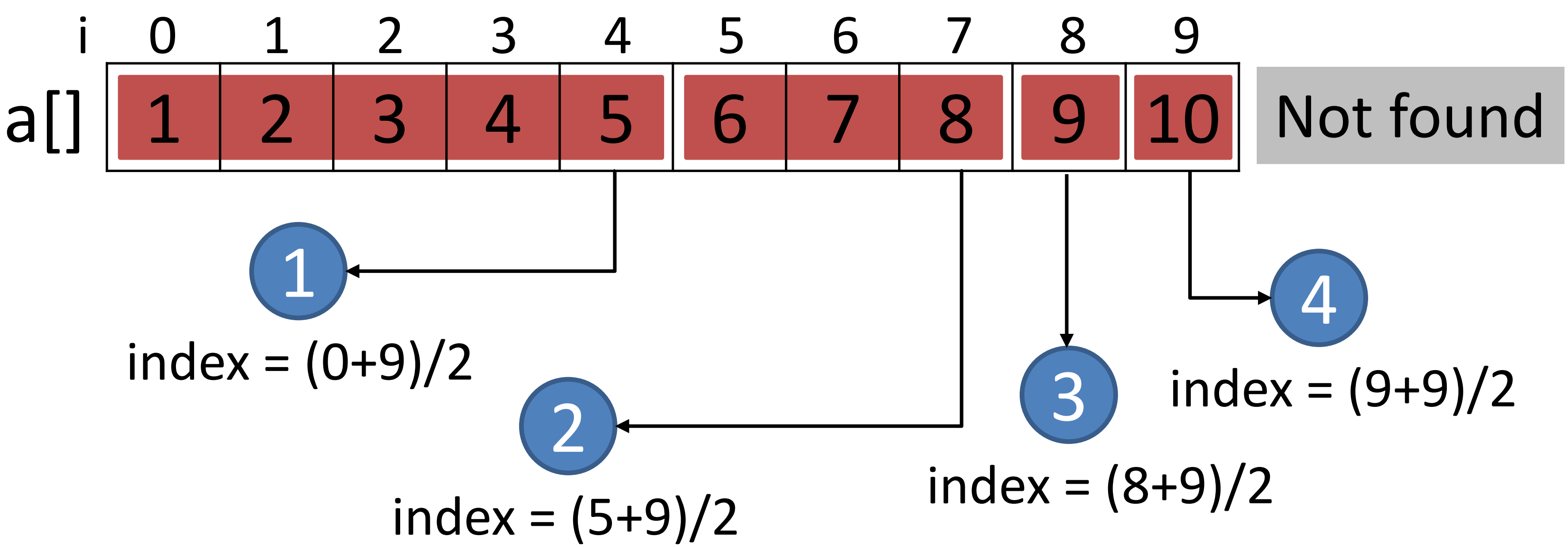
Search 7 in the following array



Contd…

- Used if the array is sorted.
- Example:

Search 11 in the following array



Contd...

Algorithm binarySearch(A,n,num)

Input: An array **A** containing **n** integers and number **num** to be searched.

Output: Index of **num** if found, otherwise -1.

```
beg = 0, end = n-1
while  beg <= end  do
        mid = (beg + end)/2
        if  A[mid] == num
                return mid
        else if   A[mid] > num
                end = mid - 1
        else if   A[mid] < num
                beg = mid + 1
return -1
```

Contd…

- In each iteration, the number of elements to be searched from gets reduced by half. This process continues till the number of elements (to be searched) reaches to 1.
  - 1^{st} iteration – $n/2^0$
  - 2^{nd} iteration – $n/2^1$
  - 3^{rd} iteration – $n/2^2$
  - …
  - $(m+1)^{th}$ iteration – $n/2^m$

Let the termination condition (i.e. single element to search from) reaches at the $(m+1)^{th}$ iteration, thus

$$1 = \frac{n}{2^m}$$

$$2^m = n$$

$$m \log_2 2 = \log_2 n$$

$$m = \lg n$$

$$\Rightarrow \text{complexity} \quad \text{is} \quad O(\lg n)$$

Example – Binary Search

```
#include<stdio.h>
int binarySearch(int arr[], int n, int num)
{   int beg = 0, end = 9;
    while (beg <= end)
    {   int mid = (beg + end)/2;
        if (arr[mid] == num)
            return mid;
        else if (arr[mid] > num)
            end = mid - 1;
        else if (arr[mid] < num)
            beg = mid + 1;
    }
    return -1;
}
```

Contd…

```
int main()
{  int i, n, a[10], num;
   printf("Enter the size of an array (<=10): ");
   scanf("%d",&n);
   for (i = 0; i < n; i++)
     {  printf("Enter element at index %d: ",i);
        scanf("%d", &a[i]);  }
     printf("\nEnter number to search: ");
     scanf("%d",&num);
     int found = binarySearch(a,n,num);
     if(found != -1)
       printf("Element found at index %d",found);
     else
       printf("Element not found.");
     return 0;  }
```

Thankyou