Outline : Lecture 4

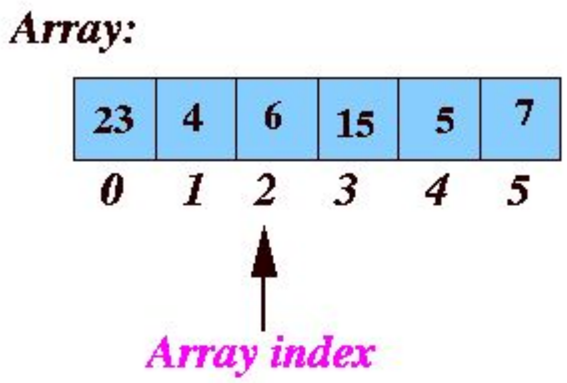
- Classification of Data Structure
- Array Data Structure
- Representation Linear Array in Memory
- Representation of Two Dimensional Array in Memory
- Representation of Multidimensional Array in Memory
- Operations on Array Data Structure
    - Traversing Linear Array
    - Insertion Operation
        - Time complexity Analysis
            - Best Case Analysis
            - Worst Case Analysis
    - Deletion Operation
        - Time complexity Analysis
            - Best Case Analysis
            - Worst Case Analysis
    - Binary Search Algorithm
        - Time complexity Analysis
            - Best Case Analysis
            - Worst Case Analysis

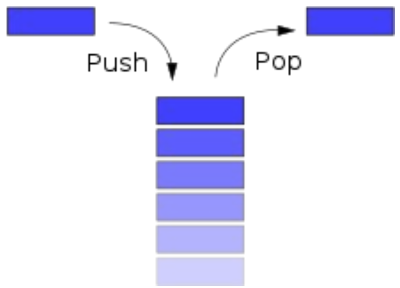
Classification of Data Structure:

Data Structure

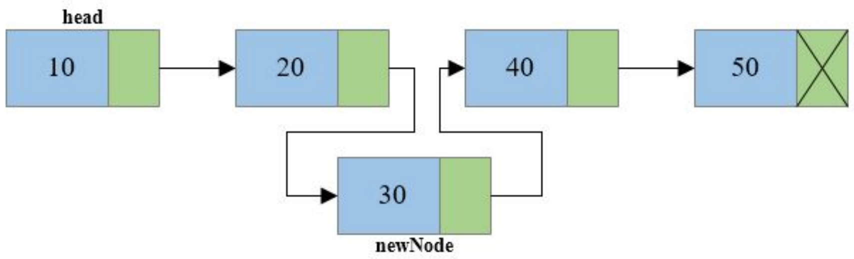
Linear Data Structure

Non-linear Data Structure

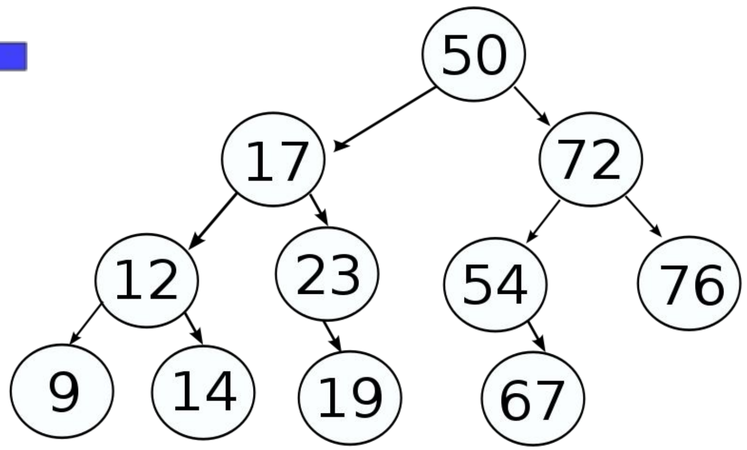




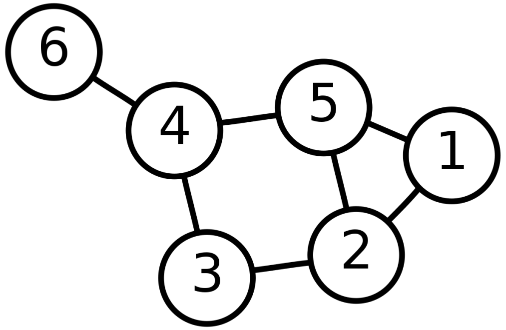
Stack



Link List



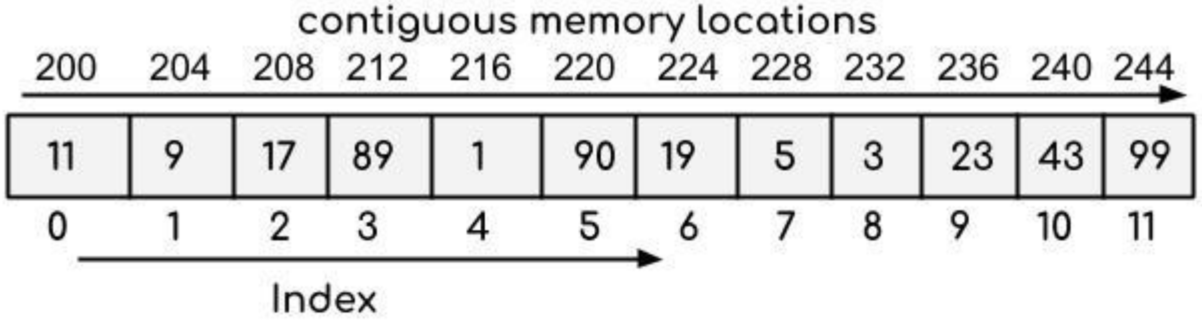
Tree



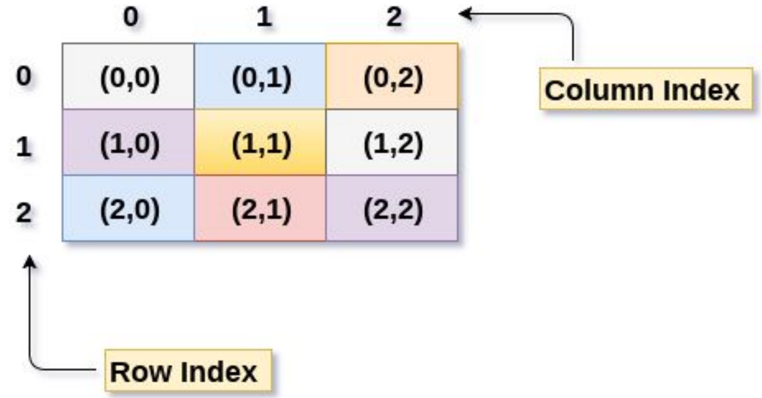
Graph

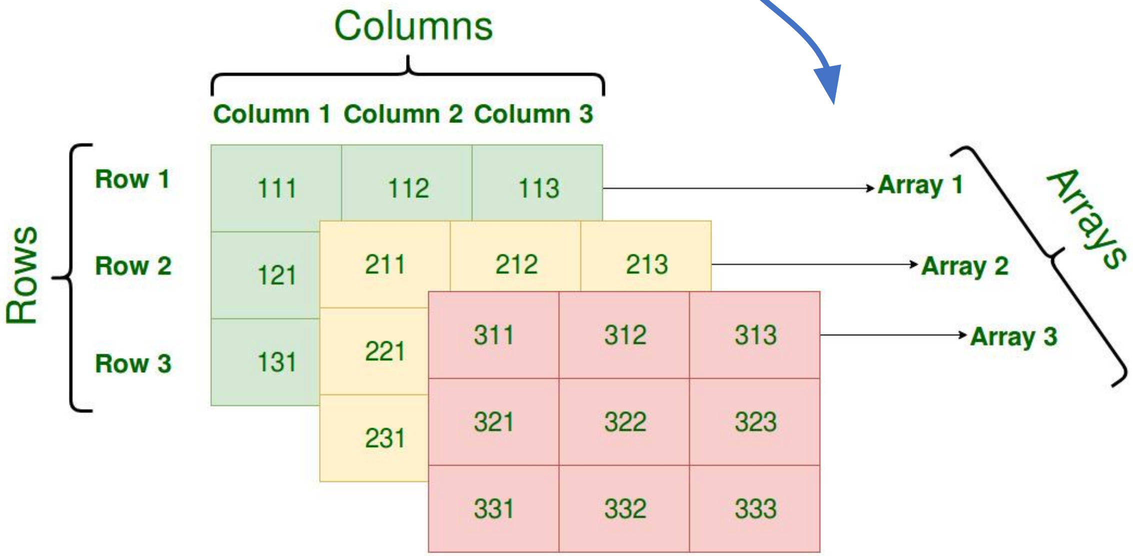
Array Data Structure:

- Linear Array :

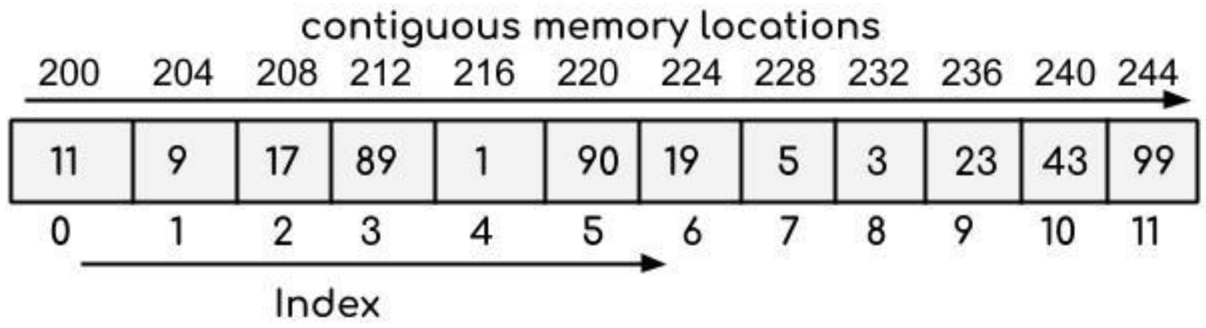


- Two Dimensional Array:
- Three Dimensional Array:





Representation of Linear Array in Memory:



Loc(A[4])=200+4x(4-0)=216 $\Rightarrow Loc(A[k]) = Base(A) + w * (k - LB)$

where $Base(A)$ is the base address of the array A, w is the size of the data type of the array element and LB is the Lower Bound (LB) of the array

Find $Loc(A[7])$?

Representation of Two-Dimensional Array in Memory:

- In C language, $float\ A[5][3]$

	0	1	2
0	A[0][0]	A[0][1]	A[0][2]
1	A[1][0]	A[1][1]	A[1][2]
2	A[2][0]	A[2][1]	A[2][2]
3	A[3][0]	A[3][1]	A[3][2]
4	A[4][0]	A[4][1]	A[4][2]

Row-Major Order

2000	A[0][0]	Row 0
2004	A[0][1]	
2008	A[0][2]	
2012	A[1][0]	Row 1
2016	A[1][1]	
2020	A[1][2]	
2024	A[2][0]	Row 2
2028	A[2][1]	
2032	A[2][2]	
2036	A[3][0]	Row 3
2040	A[3][1]	
2044	A[3][2]	
2048	A[4][0]	Row 4
2052	A[4][1]	
2056	A[4][2]	

Column-Major Order

2000	A[0][0]	Column 0
2004	A[1][0]	
2008	A[2][0]	
2012	A[3][0]	
2016	A[4][0]	
2020	A[0][1]	Column 1
2024	A[1][1]	
2028	A[2][1]	
2032	A[3][1]	
2036	A[4][1]	
2040	A[0][2]	Column 2
2044	A[1][2]	
2048	A[2][2]	
2052	A[3][2]	
2056	A[4][2]	

Representation of Two-Dimensional Array in Memory: (Cont..)

2000	A[0][0]	Row 0
2004	A[0][1]	
2008	A[0][2]	
2012	A[1][0]	Row 1
2016	A[1][1]	
2020	A[1][2]	
2024	A[2][0]	Row 2
2028	A[2][1]	
2032	A[2][2]	
2036	A[3][0]	Row 3
2040	A[3][1]	
2044	A[3][2]	
2048	A[4][0]	Row 4
2052	A[4][1]	
2056	A[4][2]	

(a) Row-major Order

$float\ A[5][3]$

Find $Loc(A[3][1]) = ?$

Row-major Order:

$$Loc(A[3][1]) = 2000 + 4 * [(3-0) * 3 + (1-0)] = 2040$$

In general,

$$datatype\ A[d_1][d_2];$$

$$Loc(A[k_1][k_2] = Base(A) + w * [(E_1 * d_2 + E_2)]$$

$where\ d_i\ is\ the$ length of i^{th} dimension of the array A ,
$E_i = k_i - LB_i$ where E_i is called the effective index of i^{th} dimension and LB_i is the lower bound of i^{th} dimension ,
w is the size of the datatype of the array A

Column-major Order:

$$Loc(A[3][1]) = 2000 + 4 * [(1-0) * 5 + (3-0)] = 2032$$

In general,

$$datatype\ A[d_1][d_2];$$

$$Loc(A[k_1][k_2] = Base(A) + w * [(E_2 * d_1 + E_1)]$$

2000	A[0][0]	Column 0
2004	A[1][0]	
2008	A[2][0]	
2012	A[3][0]	
2016	A[4][0]	
2020	A[0][1]	Column 1
2024	A[1][1]	
2028	A[2][1]	
2032	A[3][1]	
2036	A[4][1]	
2040	A[0][2]	Column 2
2044	A[1][2]	
2048	A[2][2]	
2052	A[3][2]	
2056	A[4][2]	

(b) Column-major Order

Representation of Multi-Dimensional Array in Memory:

Row-major Order:

In general for *two-dimensional* array, ***datatype*** $\boldsymbol{A[d_1][d_2]}$;

$$Loc(A[k_1][k_2] = Base(A) + w * [(E_1 * d_2 + E_2)]$$

In general for ***n-dimensional*** array, ***datatype*** $\boldsymbol{A[d_1][d_2] \ldots . [d_n]}$

$$Loc(A[k_1][k_2] \ldots . [k_n]) = Base(A) + w * [(\ldots . ((E_1 * d_2 + E_2) * d_3 + E_3) \ldots) * d_n + E_n]$$

$where\ d_i\ is\ the$ length of i^{th}dimension of the array A, $E_i = k_i - LB_i$ where E_i is called the effective index of i^{th} dimension and LB_i is the lower bound of i^{th}dimension ,w is the size of the datatype of the array A

Column-major Order:

In general for *two-dimensional* array, ***datatype*** $\boldsymbol{A[d_1][d_2]}$;

$$Loc(A[k_1][k_2] = Base(A) + w * [(E_2 * d_1 + E_1)]$$

In general for ***n-dimensional*** array, ***datatype*** $\boldsymbol{A[d_1][d_2] \ldots . [d_n]}$

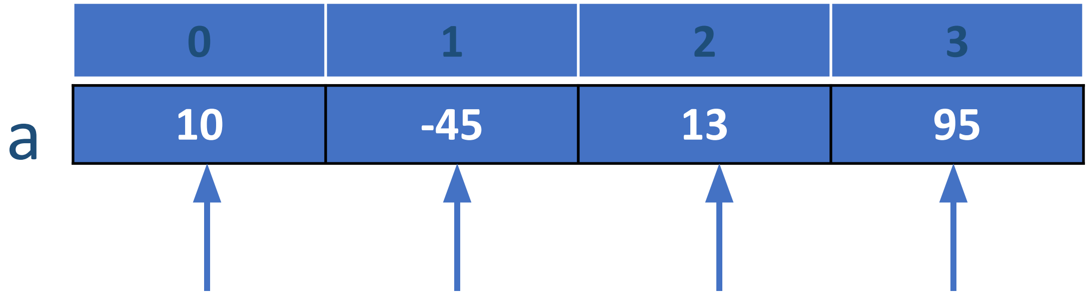
$$Loc(A[k_1][k_2] \ldots . [k_n]) = Base(A) + w * [(\ldots . ((E_n * d_{n-1} + E_{n-1}) * d_{n-2} + E_{n-2}) \ldots + E_2) * d_1 + E_1]$$

$where\ d_i\ is\ the$ length of i^{th}dimension of the array A, $E_i = k_i - LB_i$ where E_i is called the effective index of i^{th} dimension and LB_i is the lower bound of i^{th}dimension ,w is the size of the datatype of the array A

Operations on Array Data Structure: Traversing Linear Array

Traversing a linear array means accessing and processing each element of the array exactly once.

```
void display (int n)
{
 int i;
  for (i=0 ; i<n ; i++)
     printf("%d", a[i]);
 }
/* Assumption:: here the array
a is declared globally */
```



Output: 10 -45 13 95

Time complexity : $f(n) = O(n)$

Insertion Operation on Array Data Structure

Let *int a[MAX];*

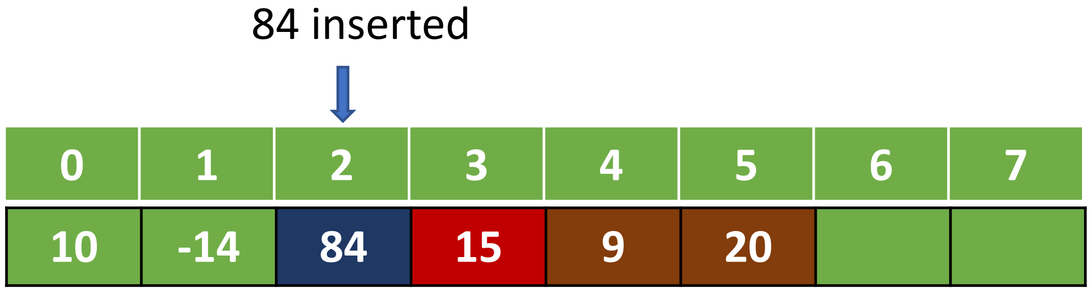
MAX : the size of the array a

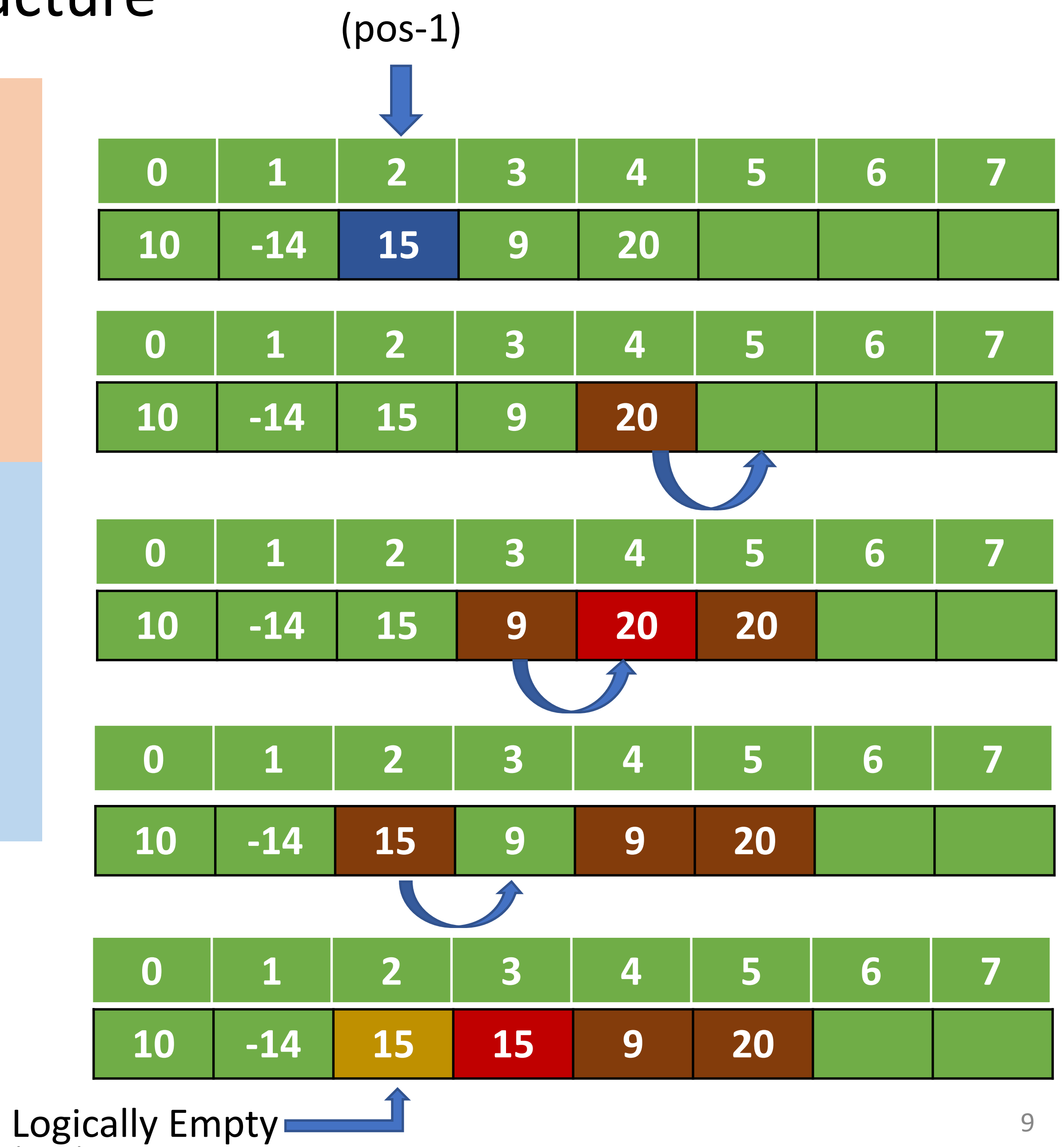
n : The number elements present in the array a

Suppose, we want to insert an element x=84 at a given position pos=3 of the given array a.

Here, MAX=8, n=5

//Shifting from right to left starting from (n-1) to (pos-1)
for (i=n-1 ; i >=(pos-1) ; i--)
  a[i+1]=a[i];
// a[pos-1] is logically empty now
 a[pos-1]=x; **//the element *x* is inserted**
n=n+1; **// the number of element n is updated**





Insertion Operation on Array Data Structure: (Cont…)

1. *In case of insertion operation, we first check the overflow condition*
2. *The position **pos** given by the user must be checked its validity that means whether the position is valid position or not (valid position range:* $1 <= pos <= (n+1)$ *)*

```
void insert (int x, int pos)
{
  int  i;
  if (n==MAX_SIZE)  // MAX_SIZE is the size of the array
   printf("\n Overflow");
  else
   {
    if ((pos>=1) && (pos<=(n+1)))  // valid position
      {
       //Shifting from right to left starting from (n-1) to (pos-1)
       for ( i=n-1 ; i >=(pos-1) ; i-- )
         a[i+1]=a[i];
      // a[pos-1] is logically empty now
      a[pos-1]=x; //the element x is inserted
      n=n+1; // the number of element n is updated
      }
    else
      printf("\n Invalid position");
   }
}
```

Time complexity Analysis:

Best Case Analysis:

0	1	2	3	4	5	6	7
10	-14	15	9	20			

Here n=5

Best case Situation: If we want to insert at the (n+1)th position, then a[n]=x and n=n+1

Hence, $f(n) = O(1)$ [Constant time]

Worst Case Analysis: If we inset at the first position, then

0	1	2	3	4	5	6	7
10	-14	15	9	20			

n no. of shifting to be done

Hence, $f(n) = O(n)$

Deletion Operation on Array Data Structure:

Let *int a[MAX];*

MAX : the size of the array a

n : The number elements present in the array a

Suppose, we want to deletion an element at a given position pos=3 of the given array a.

Here, MAX=8, n=6

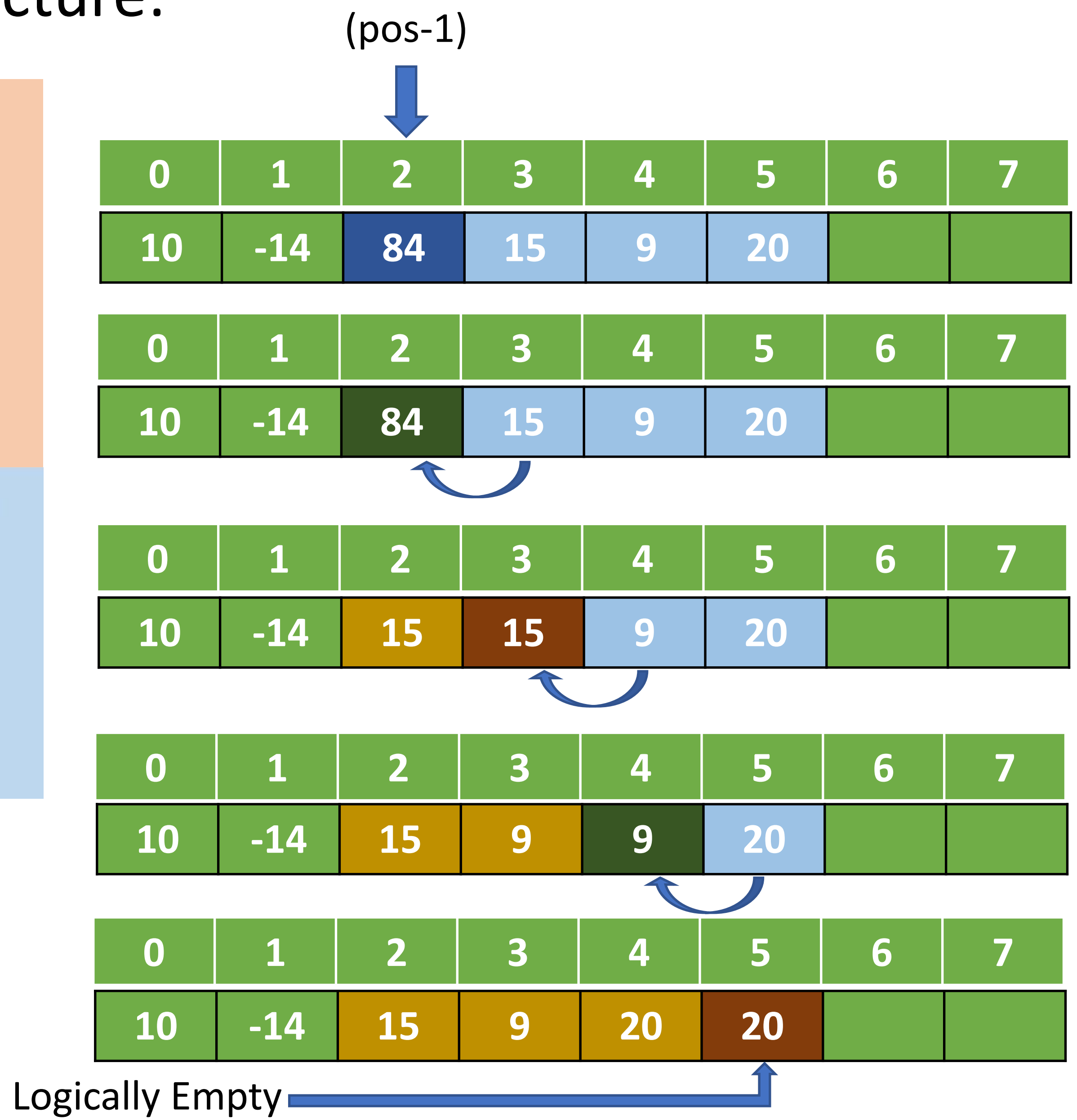
//Shifting from left to right starting from pos to $(n-1)$

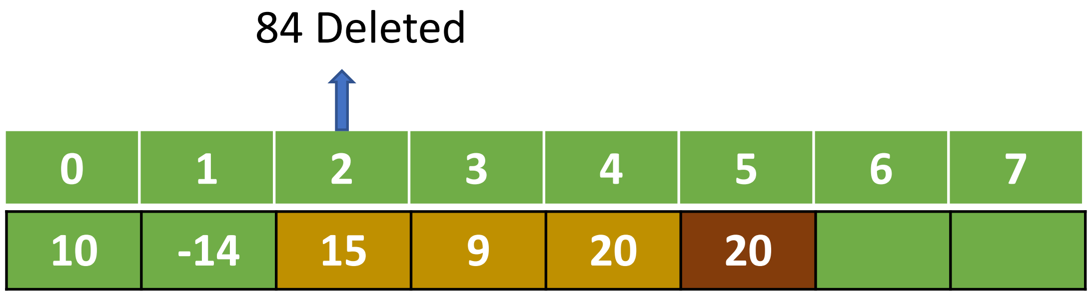
for (i=pos ; i <=(n-1) ; i++)

a[i]=a[i+1];

// deleted the element from pos

n=n-1; **// the number of element n is updated**





Deletion Operation on Array Data Structure: (Cont…)

1. *In case of deletion operation, we first check the underflow condition*
2. *The position **pos** given by the user must be checked its validity that means whether the position is valid position or not (valid position range:* $1 <= pos <= n$*))*

```
void deletion ( int pos)
{
  int  i;
  if (n==0)  // No Element is present in the given array
    printf("\n Underflow");
  else
   {
    if ((pos>=1) && (pos<=n)))  // valid position
       {
         //Shifting from left to right starting from pos to (n − 1)
         for ( i=pos ; i <=(n-1) ; i++ )
           a[i]=a[i+1];
         // a[n-1] is logically empty now
          n=n-1; // the number of element n is updated
       }
    else
      printf("\n Invalid position");
   }
}
```

Time complexity Analysis:

Best Case Analysis:

0	1	2	3	4	5	6	7
10	-14	15	9	20			

Here n=5

Best case Situation: If we want to delete at the nth position, then n=n-1

Hence, $f(n) = O(1)$ [Constant time]

Worst Case Analysis: If we want to delete at the first position,

0	1	2	3	4	5	6	7
10	-14	15	9	20			

$(n-1)$ no. of shifting to be done

Hence, $f(n) = O(n-1) = O(n)$

Binary Search Algorithm:

Limitations:

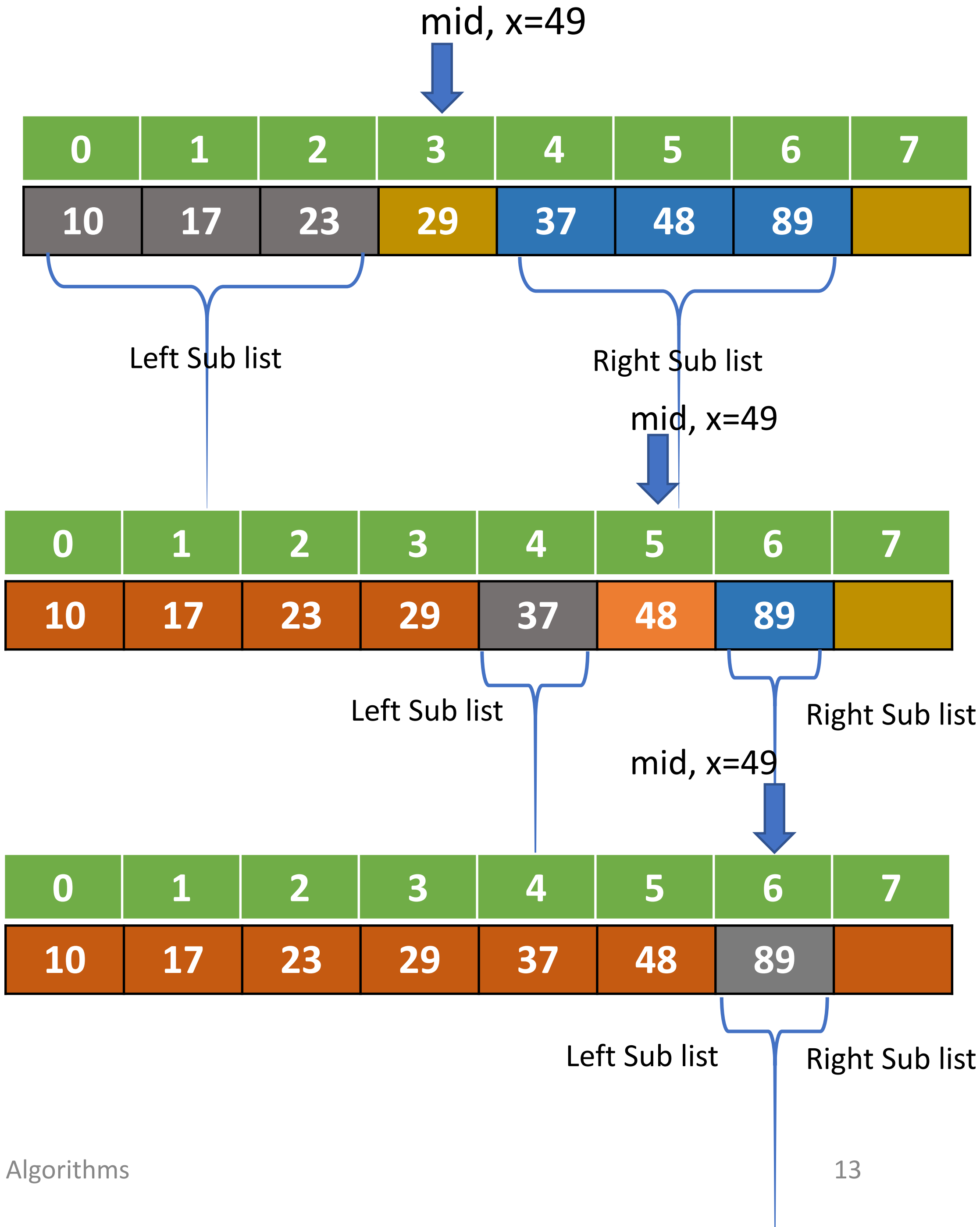
1) *The list must be sorted*

2) *The data structure, containing the list of sorted elements, must have direct access to any element without accessing any other elements*

n=7, x=49, Initially, beg=0, end=(n-1)=6,

```
mid=(beg+end)/2;
If (x==a[mid])
      printf("\n Search is successful ");
else
 If(x>a[mid])
    beg=mid+1;
    else
    end=mid-1;
```

beg	end	mid
0	6	3
4	6	5
6	6	6
6	5	STOP



Binary Search Algorithm: (Cont..)

```
int binary_search(int x)
{
 int beg = 0, end = n − 1, mid;
 while(beg <= end)
  {
   mid = (beg + end)/2;
   if(x == a[mid])
     return(mid);
   if(x < a[mid])
     end = mid − 1;
   else
     beg = mid + 1;
  }
 return(−1);
}

//Iterative Method
```

```
int b_search(int x, int beg, int end)
{
   if(beg ≤ end)
   {
     mid = (beg + end)/2;
     if(x == a[mid])
       return(mid);
     if(x < a[mid])
       {
        end = mid − 1;
        b_search(x, beg, end);
       }
      else
       {
        beg = mid + 1;
        b_search(x, beg, end);
       }
   }
   else
    return(−1);
}
```

Worst Case Time Complexity:

$T(n) = c + T(n/2)$ and $T(1) = 1$

$T(n) = c + T\left(\frac{n}{2}\right)$

$T\left(\frac{n}{2}\right) = c + T\left(\frac{n}{2^2}\right)$

$T(n) = c + T\left(\frac{n}{2}\right)$

$= c + c + T\left(\frac{n}{2^2}\right)$

$= 2c + T\left(\frac{n}{2^2}\right)$

$T\left(\frac{n}{2^2}\right) = c + T\left(\frac{n}{2^3}\right)$

$T(n) = 3c + T\left(\frac{n}{2^3}\right)$

$T(n) = \mathrm{k}c + T\left(\frac{n}{2^k}\right)$

Let $T\left(\frac{n}{2^k}\right) = T(1)$

Therefore, $\frac{n}{2^k} = 1$

$\Rightarrow 2^k = n \Rightarrow k = \log_2 n$

$T(n) = \mathrm{k}c + T\left(\frac{n}{2^k}\right)$

$= c\log_2 n + T(1) = O(\log_2 n)$

Best Case Time Complexity:

$T(n) = O(1)$