Heap

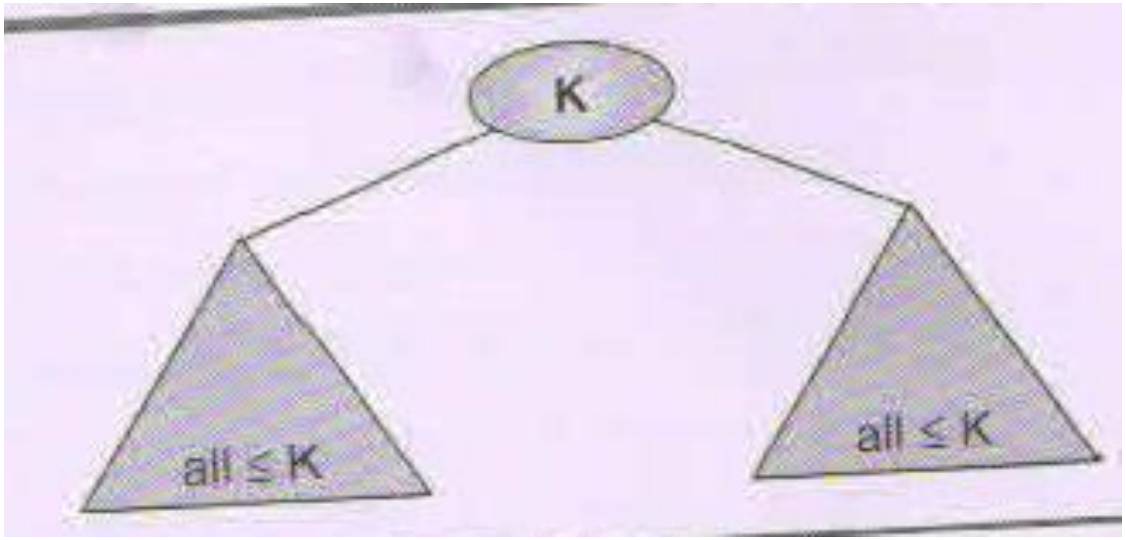
Dr. N. L. Gavankar

Definition

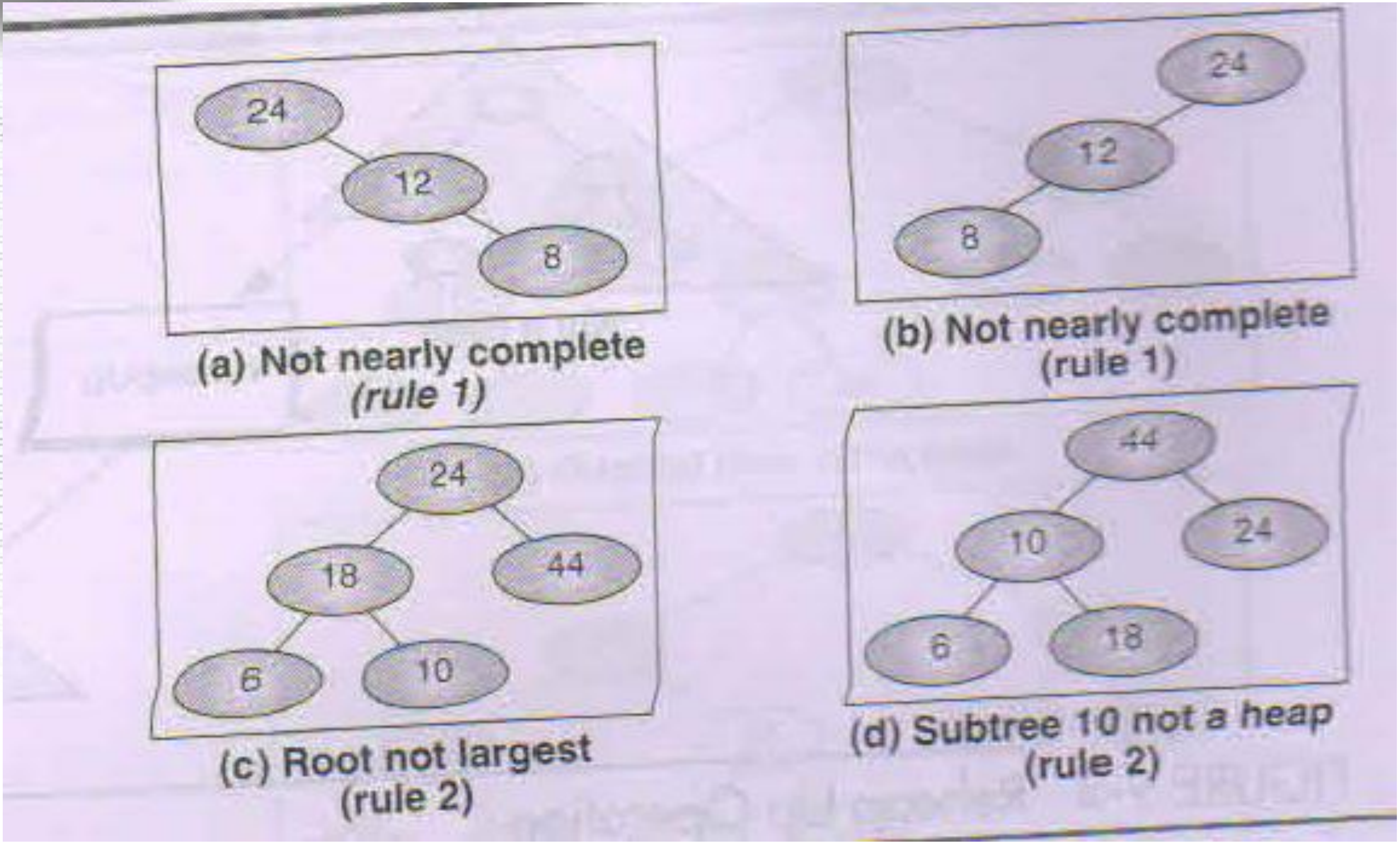
A heap, is a binary tree structure with the following properties:

1. The tree is complete or nearly complete.
2. The key value of each node is greater than or equal to the key value in each of its descendents.

- Max
- Min



Heap



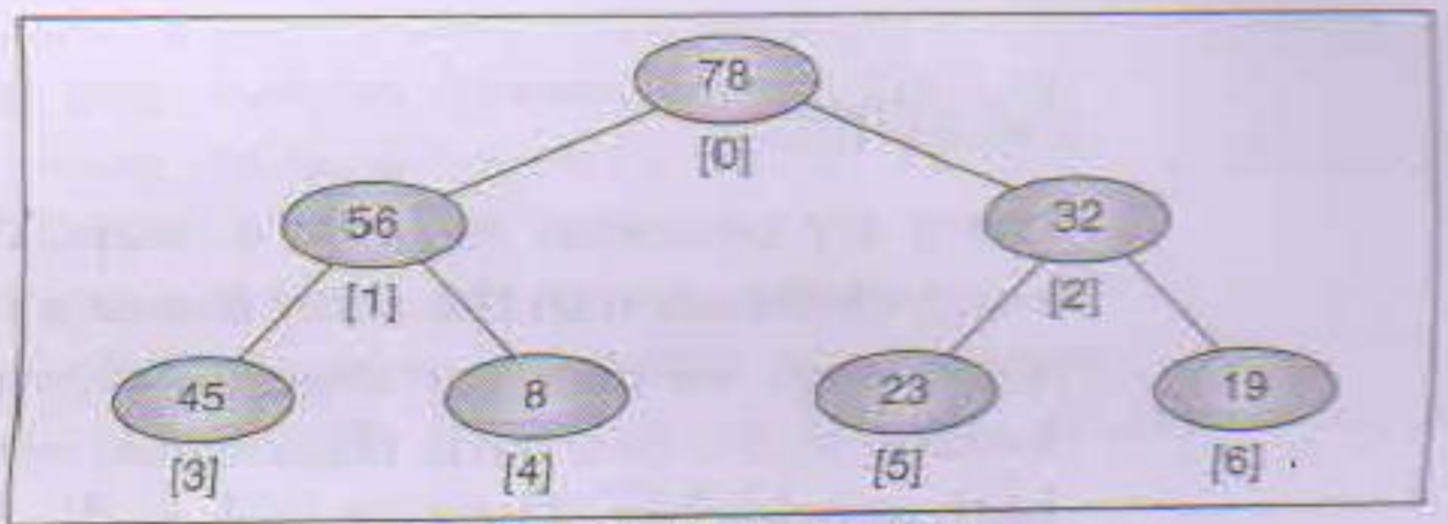
(a) Not nearly complete
(rule 1)

(b) Not nearly complete
(rule 1)

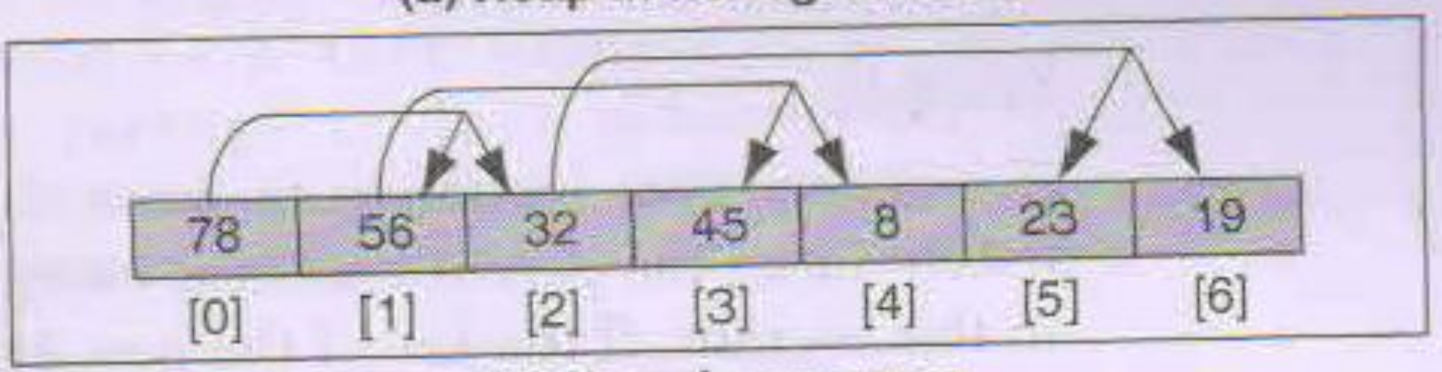
(c) Root not largest
(rule 2)

(d) Subtree 10 not a heap
(rule 2)

Heap Implementation



(a) Heap in its logical form



(b) Heap in an array

Heap Implementation

1. For a node located at index i, its children are found at:
   a. Left child: $2i + 1$
   b. Right child: $2i + 2$
2. The parent of a node located at index i is located at $\lfloor (i - 1) / 2 \rfloor$
3. Given the index for a left child, j, its right sibling, if any, is J+1
   Conversely, given the index for a right child, k, its left sibling, wh[...]
   exist, is found at $k - 1$.
4. Given the size, n, of a complete heap, the location of the first leaf is $\lfloor (n / 2) \rfloor$.
   Given the location of the first leaf element, the location of the last nonleaf element is one less.

Heap – Insert & Delete

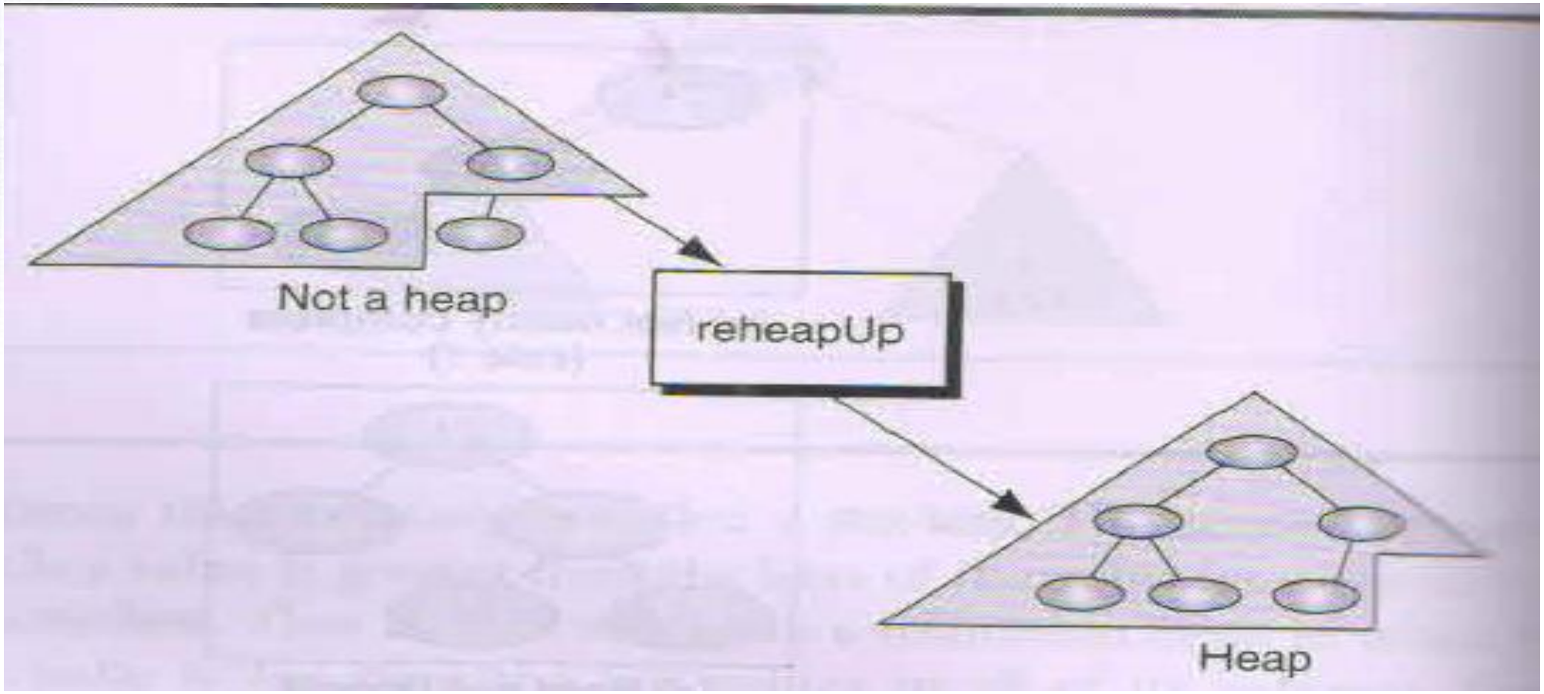
- To implement insert and Delete operations

1. Reheap up

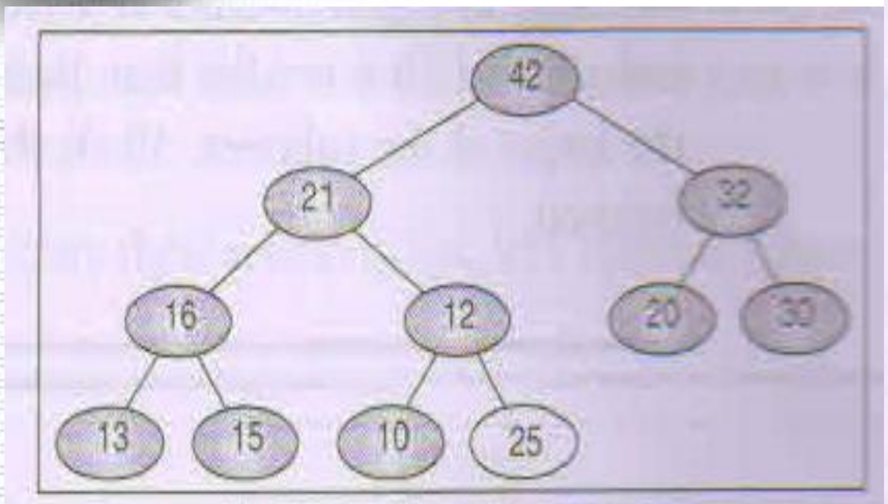
2. Reheap down

Reheap up

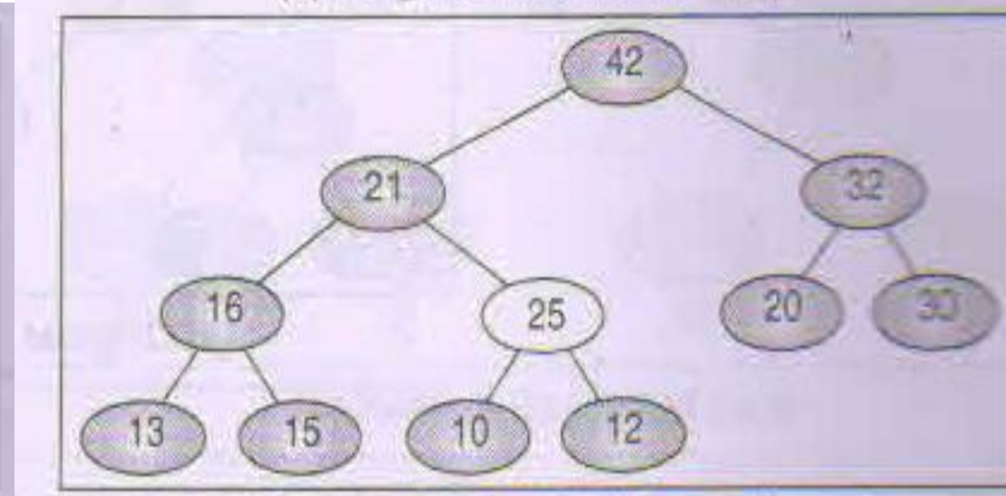
- The Reheap operation reorders a “broken” heap by floating the last element up the tree until it is in its correct location in the heap



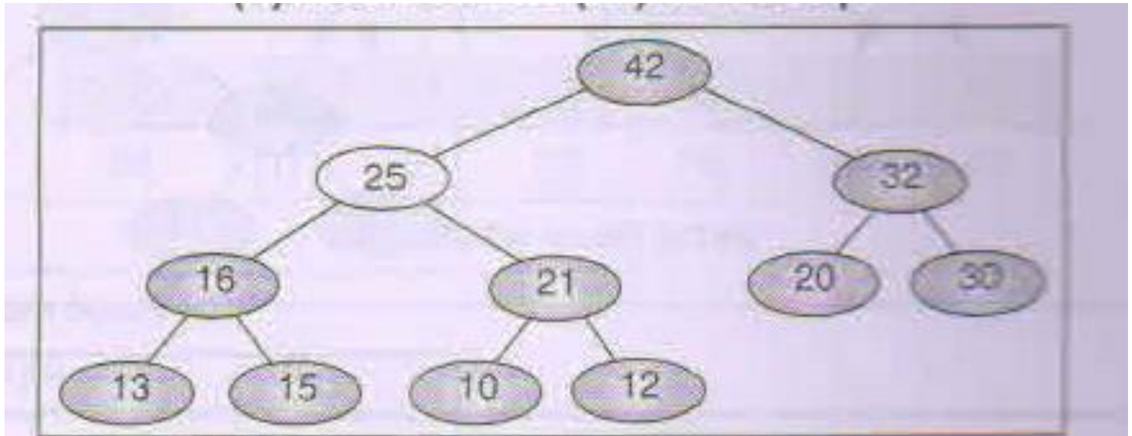
Reheap up



(a) Original tree: not a heap



(b) Last element (25) moved up



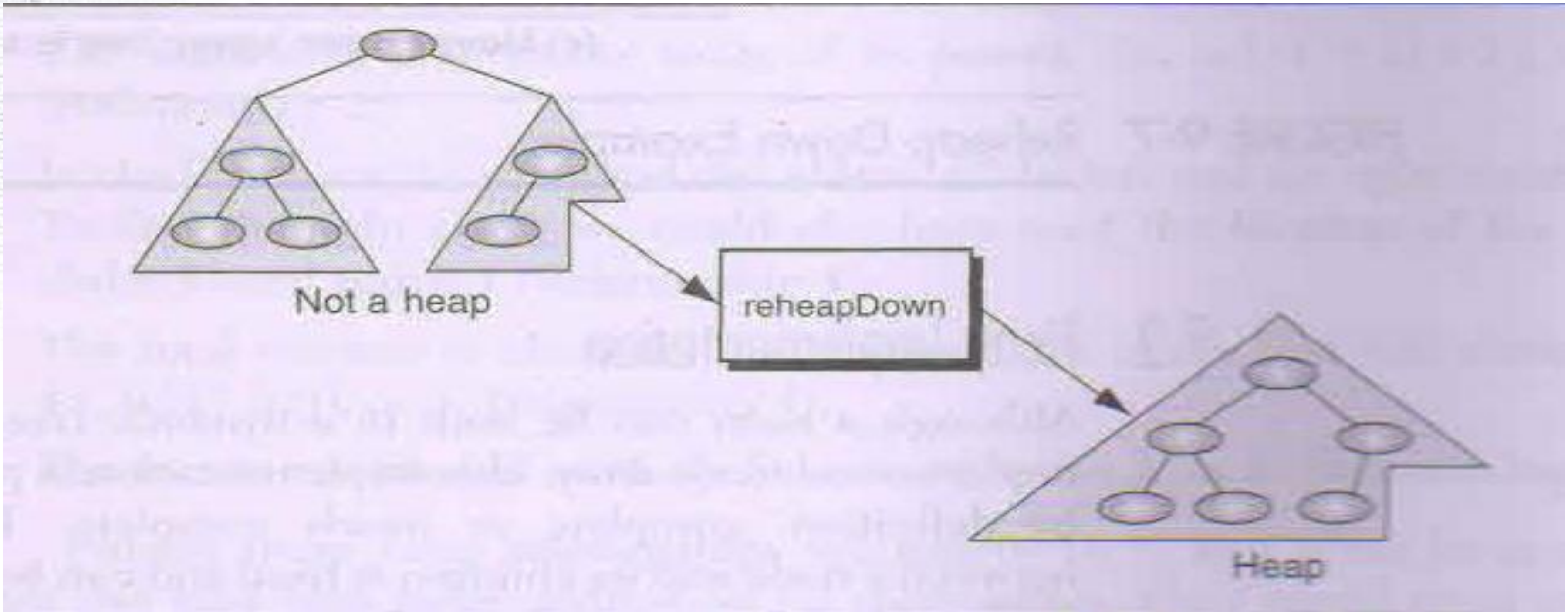
(c) Moved up again: tree is a heap

Reheap up Algorithm

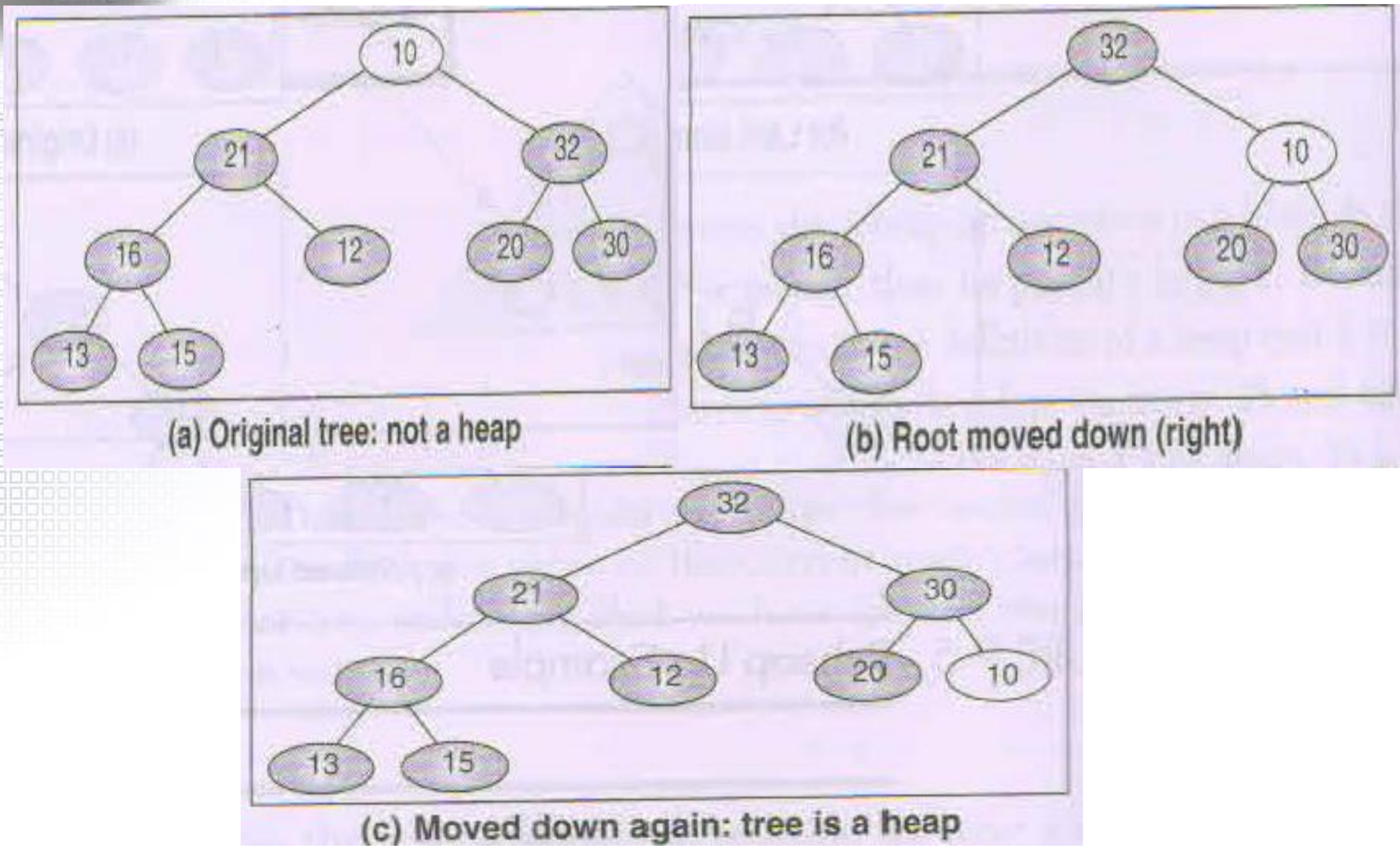
```
Algorithm reheapUp (heap, newNode)
Reestablishes heap by moving data in child up to its
correct location in the heap array.
   Pre  heap is array containing an invalid heap
        newNode is index location to new data in heap
   Post heap has been reordered
 1 if (newNode not the root)
   1  set parent to parent of newNode
   2  if (newNode key > parent key)
      1  exchange newNode and parent)
      2  reheapUp (heap, parent)
   3  end if
 2 end if
end reheapUp
```

Reheap down

- Reheap down reorders a broken heap by pushing the root down the tree until it is in its correct position in the heap.



Reheap down


10
21
32
16
12
20
30
13
15
(a) Original tree: not a heap
32
21
10
16
12
20
30
13
15
(b) Root moved down (right)
32
21
30
16
12
20
10
13
15
(c) Moved down again: tree is a heap

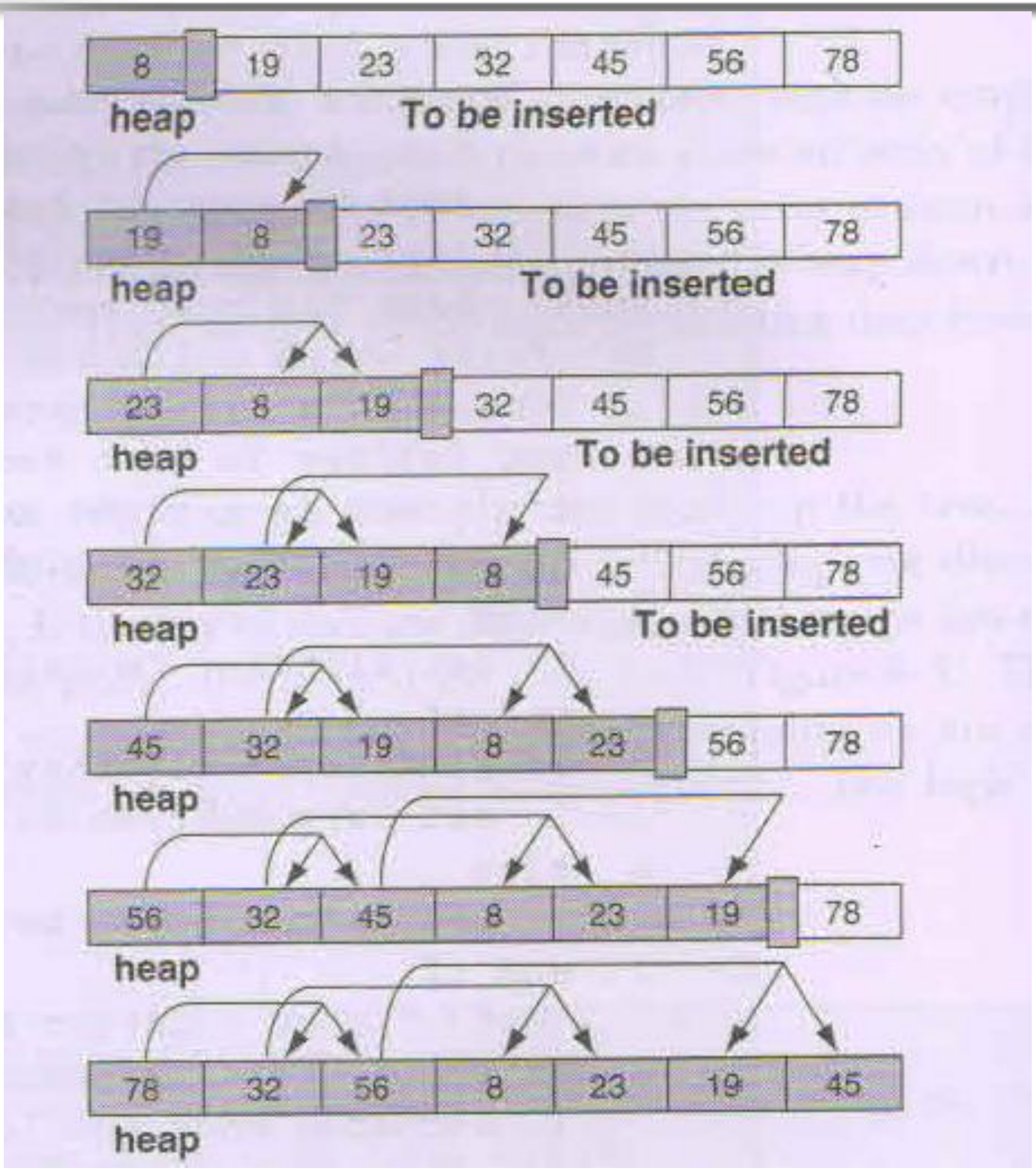
Reheap down Algorithm

```
Algorithm reheapDown (heap, root, last)
Reestablishes heap by moving data in root down to its
correct location in the heap.
   Pre    heap is an array of data
          root is root of heap or subheap
          last is an index to the last element in heap
   Post   heap has been restored
   Determine which child has larger key
```

Reheap down Algorithm

```
1 if (there is a left subtree)
   1 set leftKey to left subtree key
   2 if (there is a right subtree)
      1 set rightKey to right subtree key
   3 else
      1 set rightKey to null key
   4 end if
   5 if (leftKey > rightKey)
      1 set largeSubtree to left subtree
   6 else
      1 set largeSubtree to right subtree
   7 end if
      Test if root > larger subtree
   8 if (root key < largeSubtree key)
      1 exchange root and largeSubtree
      2 reheapDown (heap, largeSubtree, last)
   9 end if
2 end if
end reheapDown
```

Build a Heap

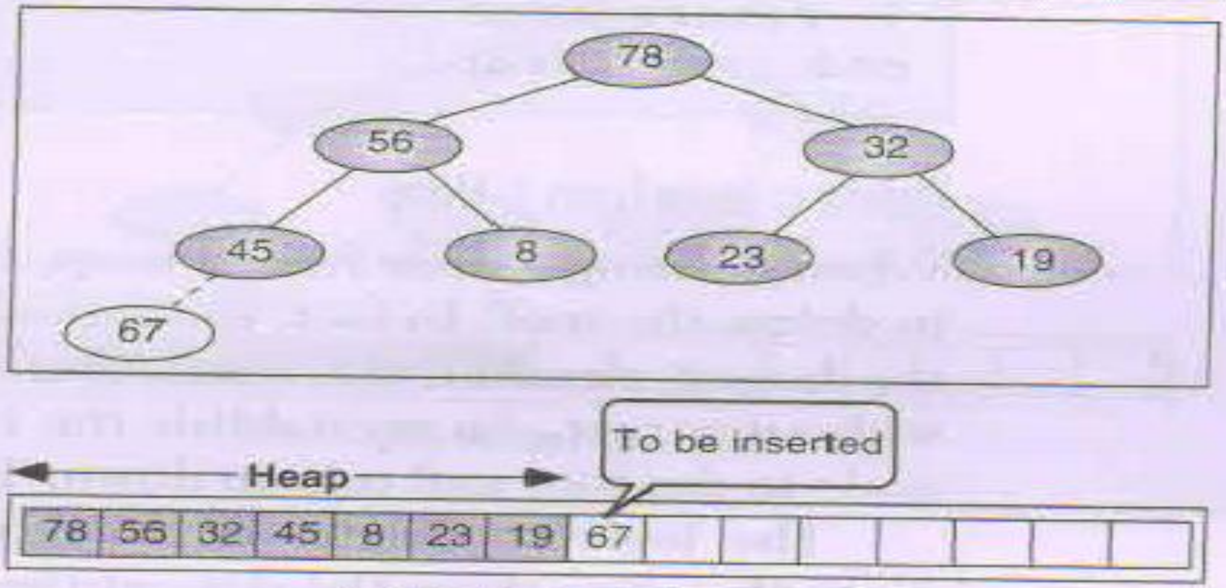


8 | 19 | 23 | 32 | 45 | 56 | 78
heap
To be inserted
19 | 8 | 23 | 32 | 45 | 56 | 78
heap
To be inserted
23 | 8 | 19 | 32 | 45 | 56 | 78
heap
To be inserted
32 | 23 | 19 | 8 | 45 | 56 | 78
heap
To be inserted
45 | 32 | 19 | 8 | 23 | 56 | 78
heap
56 | 32 | 45 | 8 | 23 | 19 | 78
heap
78 | 32 | 56 | 8 | 23 | 19 | 45
heap

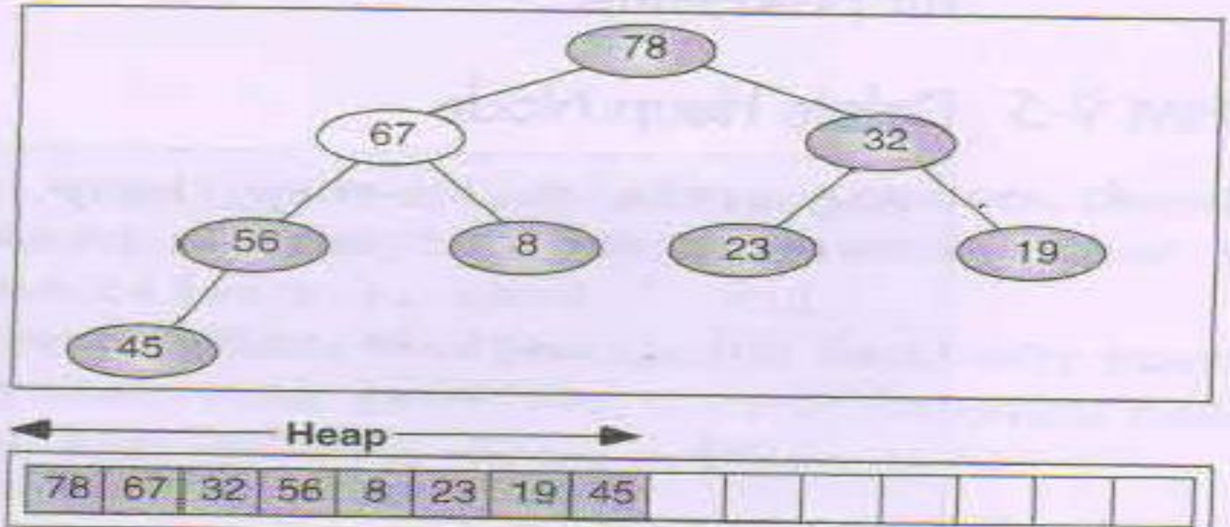
Build a Heap

```
Algorithm buildHeap (heap, size)
Given an array, rearrange data so that they form a heap.
   Pre    heap is array containing data in nonheap order
          size is number of elements in array
   Post   array is now a heap
1 set walker to 1
2 loop (walker < size)
   1  reheapUp(heap, walker)
   2  increment walker
3 end loop
end buildHeap
```

Insert a node into a Heap



(a) Before reheap up

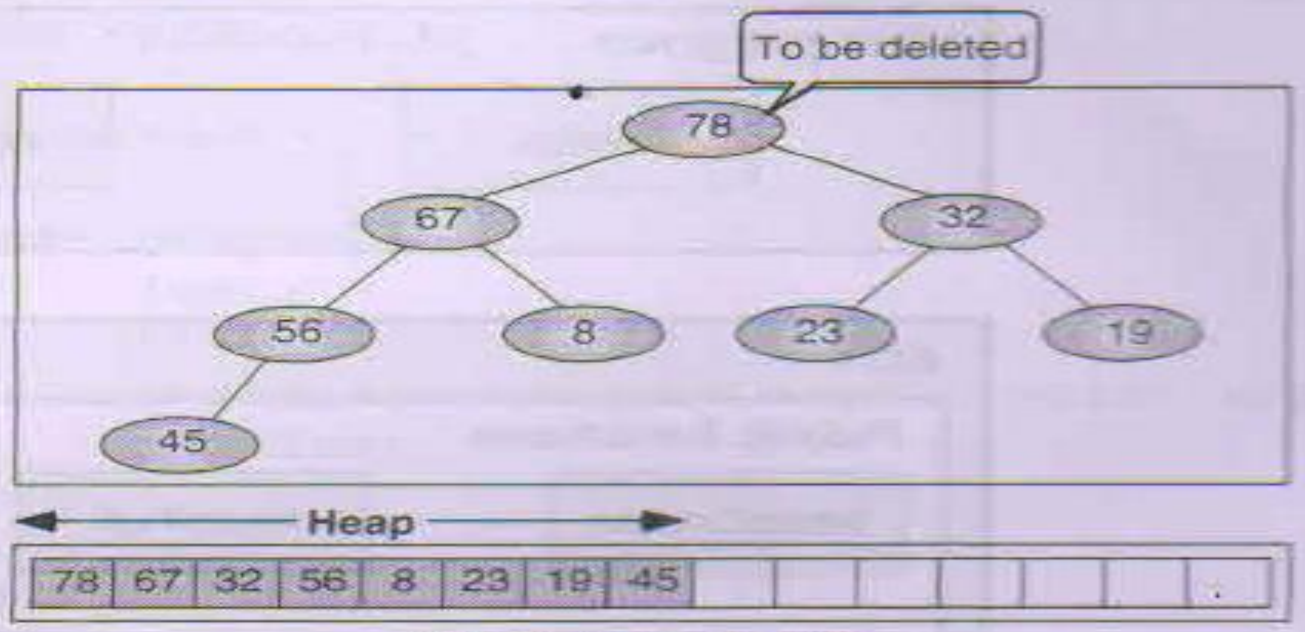


(b) After reheap up

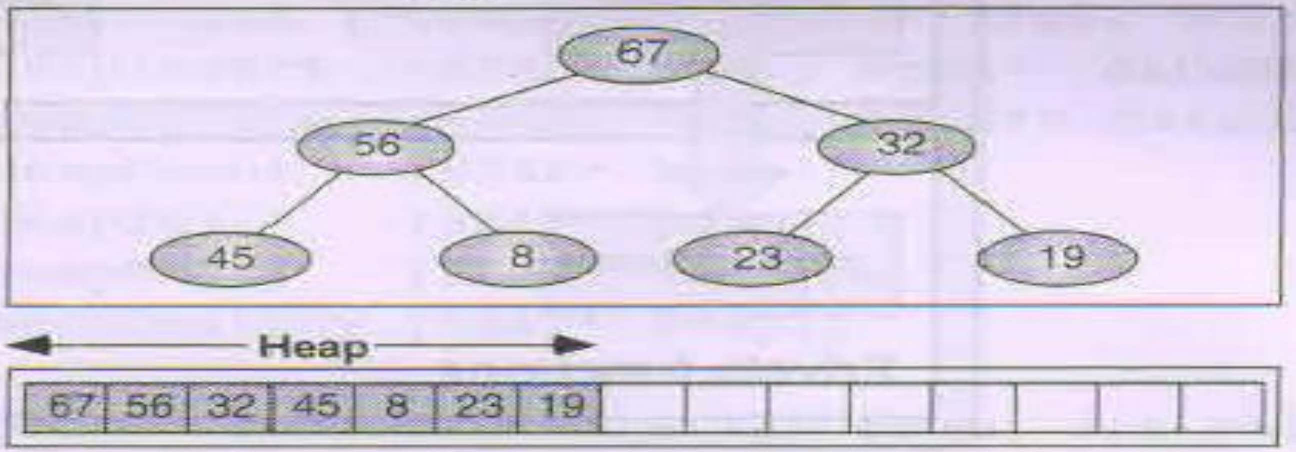
Insert a node into a Heap

```
Algorithm insertHeap (heap, last, data)
Inserts data into heap.
   Pre      heap is a valid heap structure
            last is reference parameter to last node
            data contains data to be inserted
   Post     data have been inserted into heap
   Return true if successful; false if array full
1 if (heap full)
   1  return false
2 end if
3 increment last
4 move data to last node
5 reheapUp (heap, last)
6 return true
end insertHeap
```

Delete a node into a Heap



(a) Before delete



(b) After delete

Delete a node into a Heap

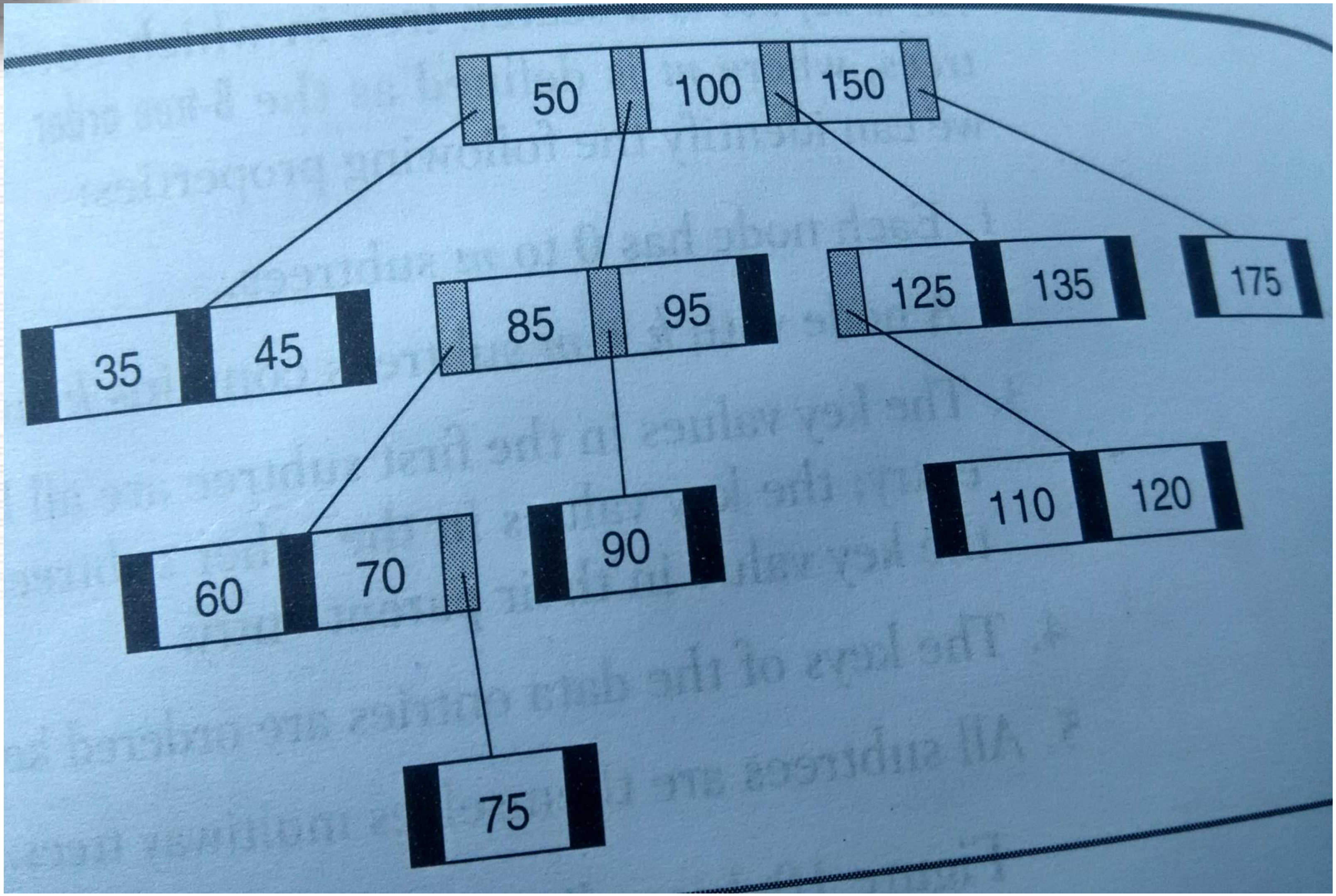
```
Algorithm deleteHeap (heap, last, dataOut)
Deletes root of heap and passes data back to caller.
   Pre    heap is a valid heap structure
          last is reference parameter to last node in
          dataOut is reference parameter for output
   Post   root deleted and heap rebuilt
          root data placed in dataOut
   Return true if successful; false if array empty
 1 if (heap empty)
   1 return false
 2 end if
 3 set dataOut to root data
 4 move last data to root
 5 decrement last
 6 reheapDown (heap, 0, last)
 7 return true
end deleteHeap
```

Multiway Trees

An *m-way tree* is a search tree in which each node can have from 0 to m subtrees, where m is defined as the **B-tree order**. Given a nonempty multiway tree, we can identify the following properties:

1. Each node has 0 to m subtrees.
2. A node with $k < m$ subtrees contains k subtrees and $k - 1$ data entries.
3. The key values in the first subtree are all less than the key value in the first entry; the key values in the other subtrees are all greater than or equal to the key value in their parent entry.
4. The keys of the data entries are ordered $key_1 \leq key_2 \leq \ldots \leq key_k$.
5. All subtrees are themselves multiway trees.

Multiway Trees

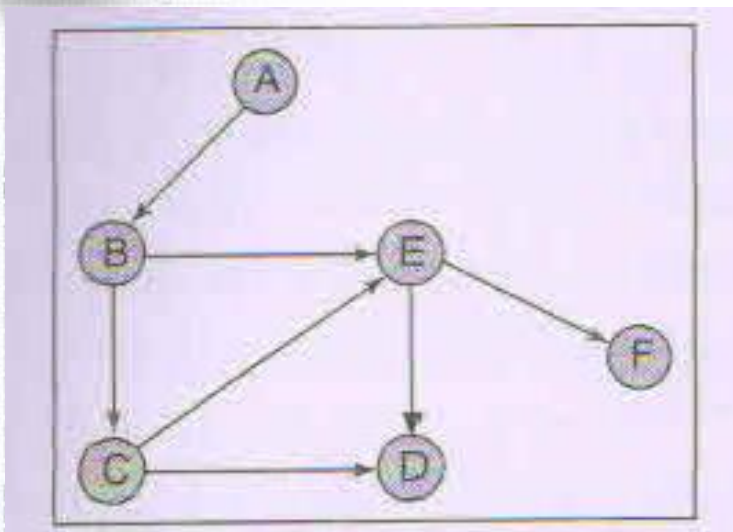


Graphs

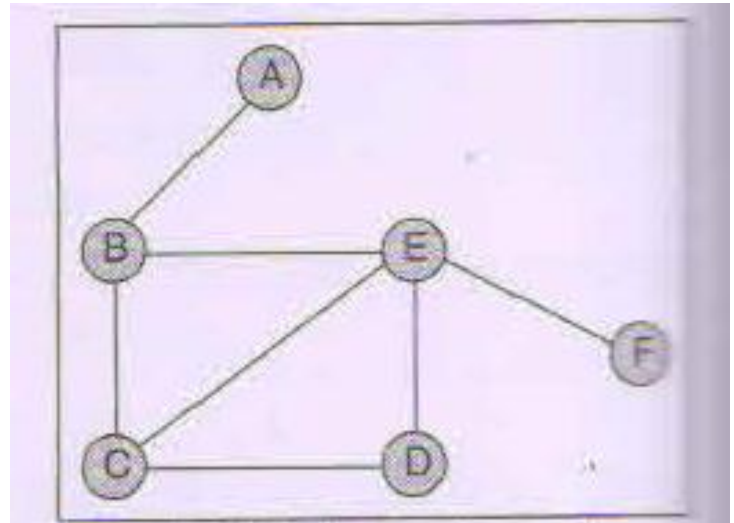
- A Graph is collection of nodes, called Vertices, and a collection of segments, called lines, connecting pair of vertices.

1. Directed
2. Undirected

Graphs

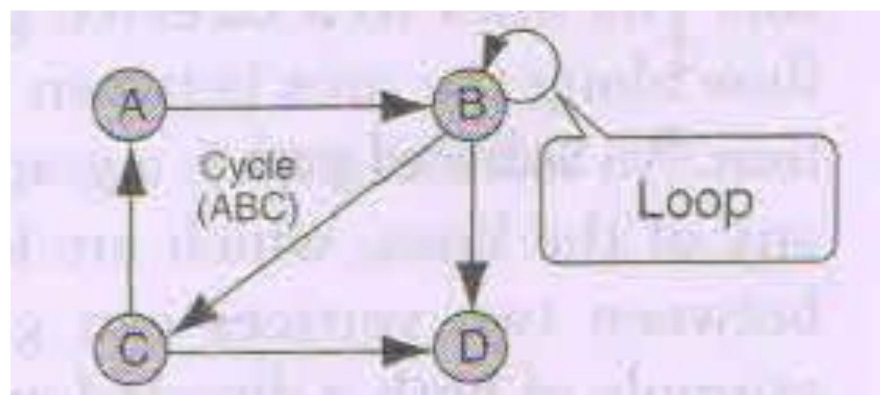


(a) Directed graph

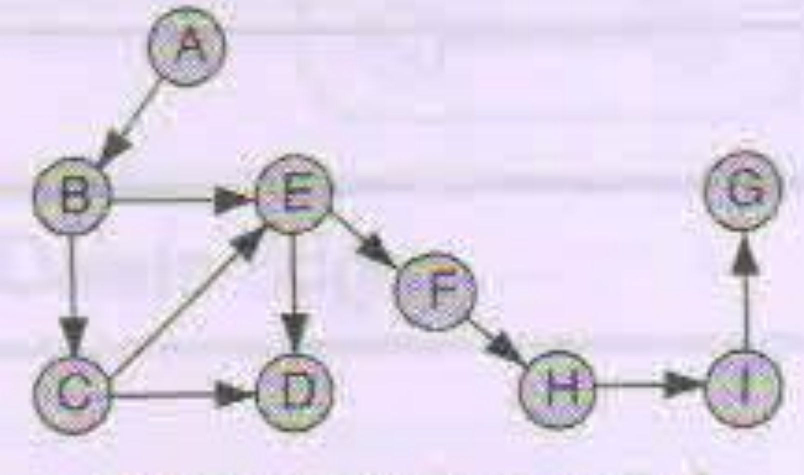


(b) Undirected graph

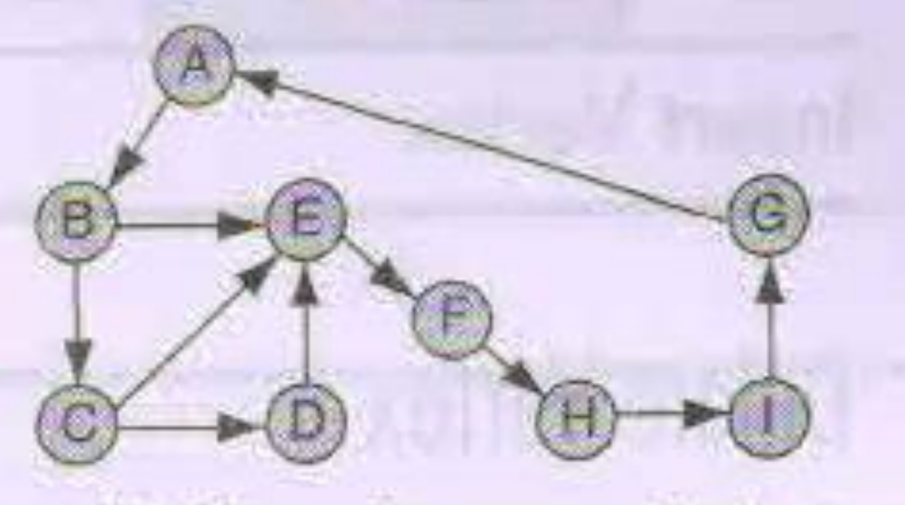
- Path
- Adjacent vertices
- Cycle (At least 3 vertices)
- Loop
- Connected vertices and Graph



Graphs



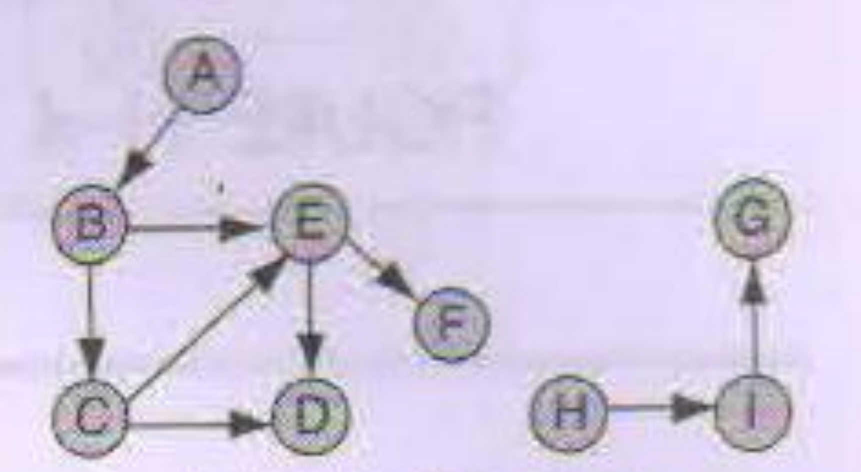
(a) Weakly connected



(b) Strongly connected

Degree

- In degree
- Out Degree

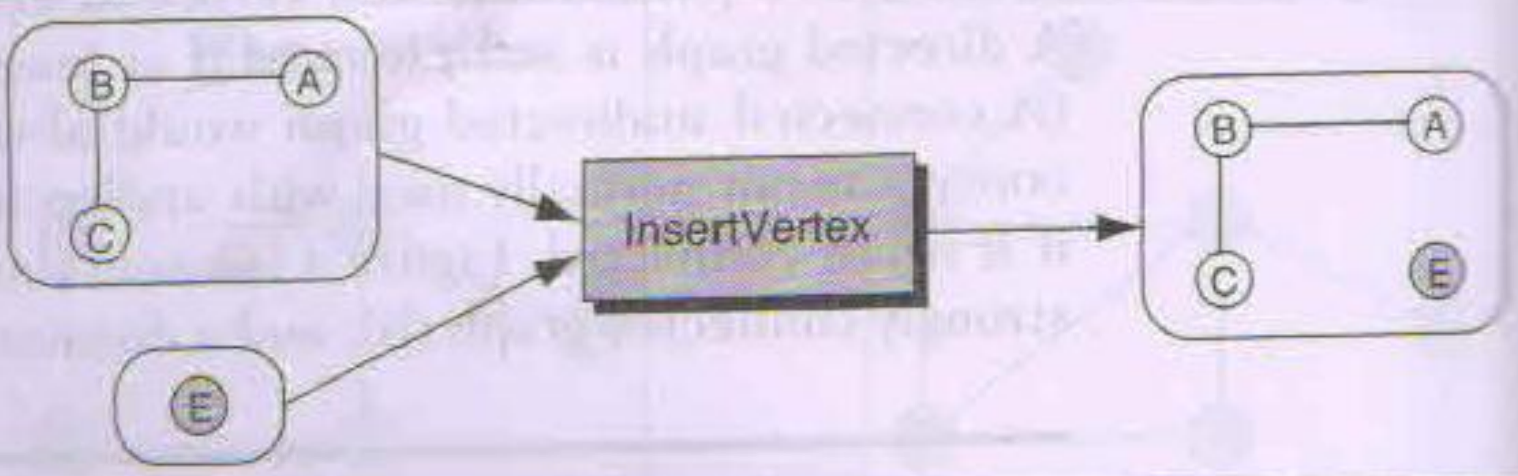


(c) Disjoint graph

Graph Operations

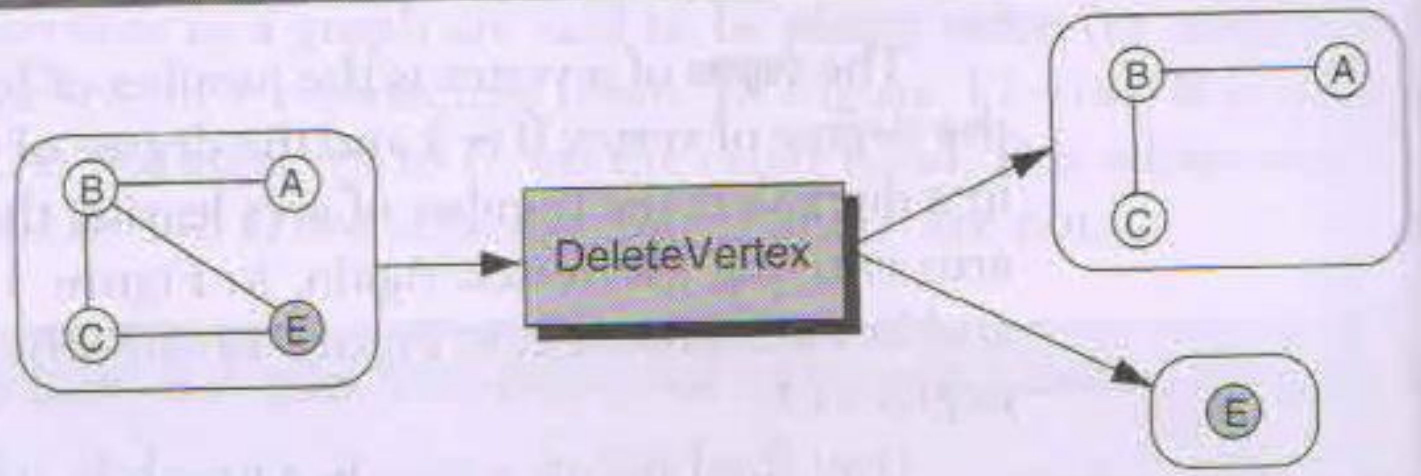
- Insert a Vertex
- Delete a Vertex
- Add an Edge
- Delete an Edge
- Find a Vertex
- Traverse a graph

Graph Operations



Insert Vertex

Graph Operations

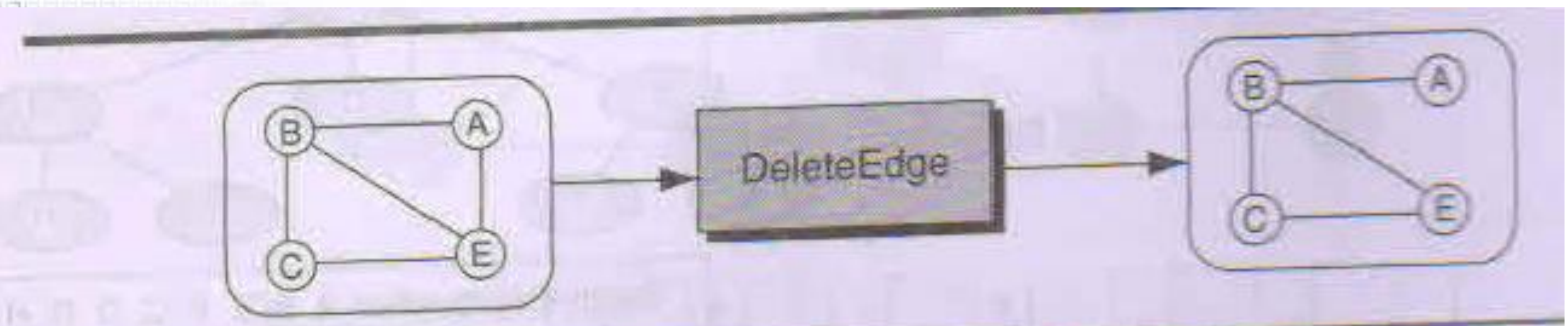


Delete Vertex

Graph Operations

Add Edge

Graph Operations

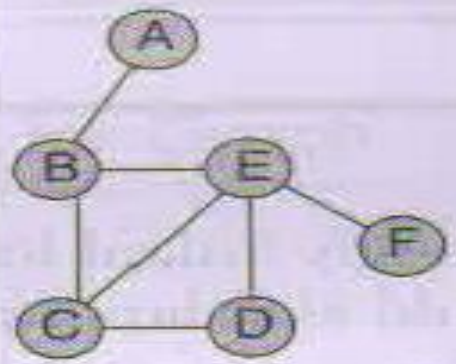


Delete Edge

Graph Operations

Find Vertex

Graph Storage Structure

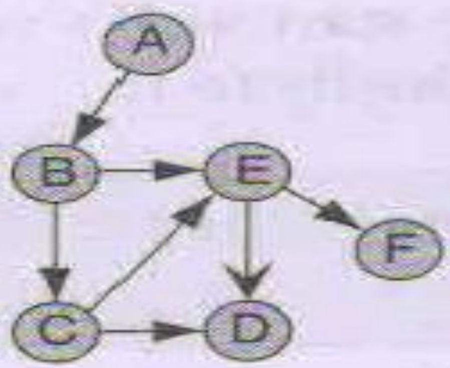


Vertex vector
A
B
C
D
E
F

Adjacency matrix

	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

(a) Adjacency matrix for nondirected graph



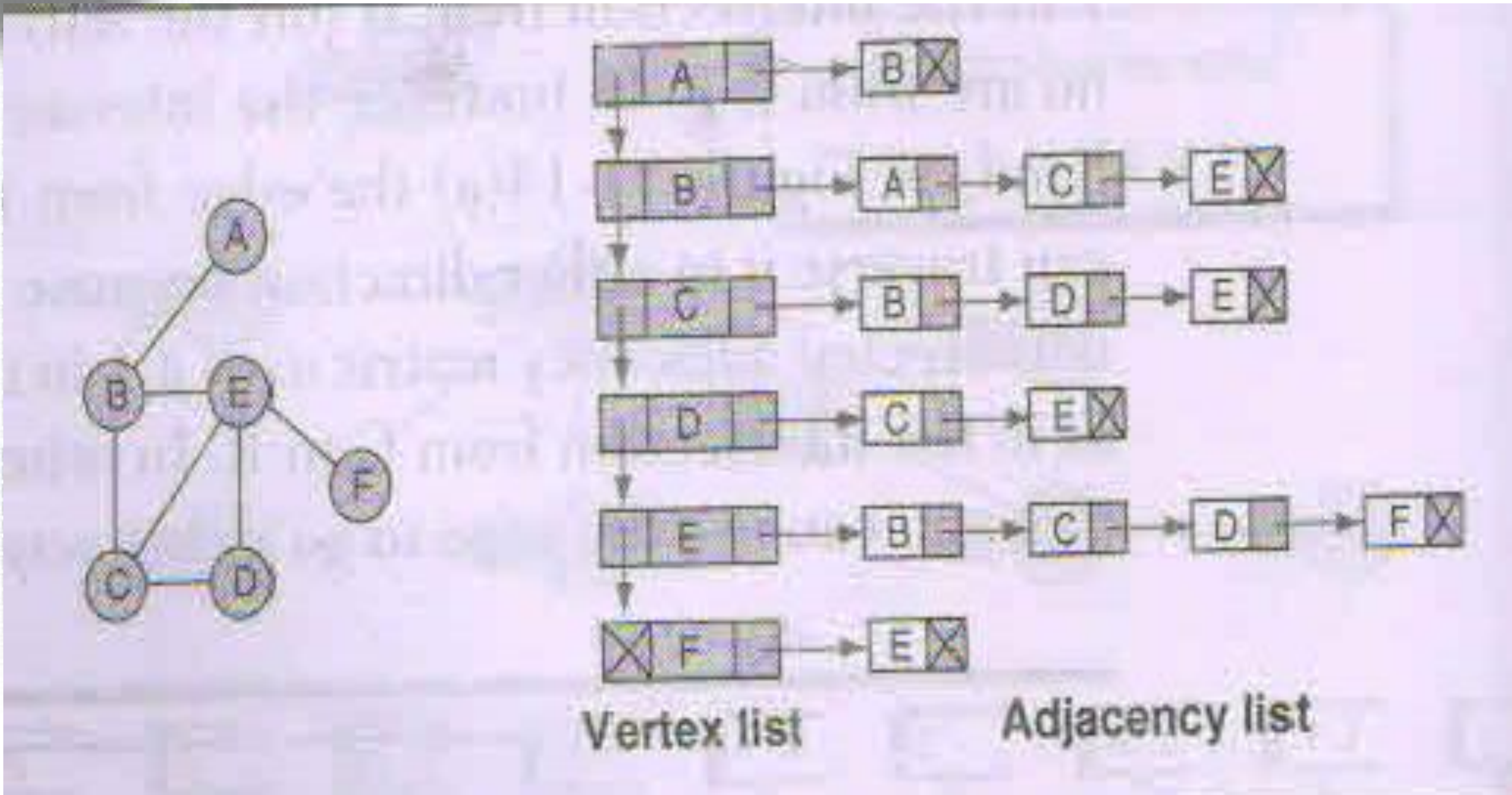
Vertex vector
A
B
C
D
E
F

Adjacency matrix

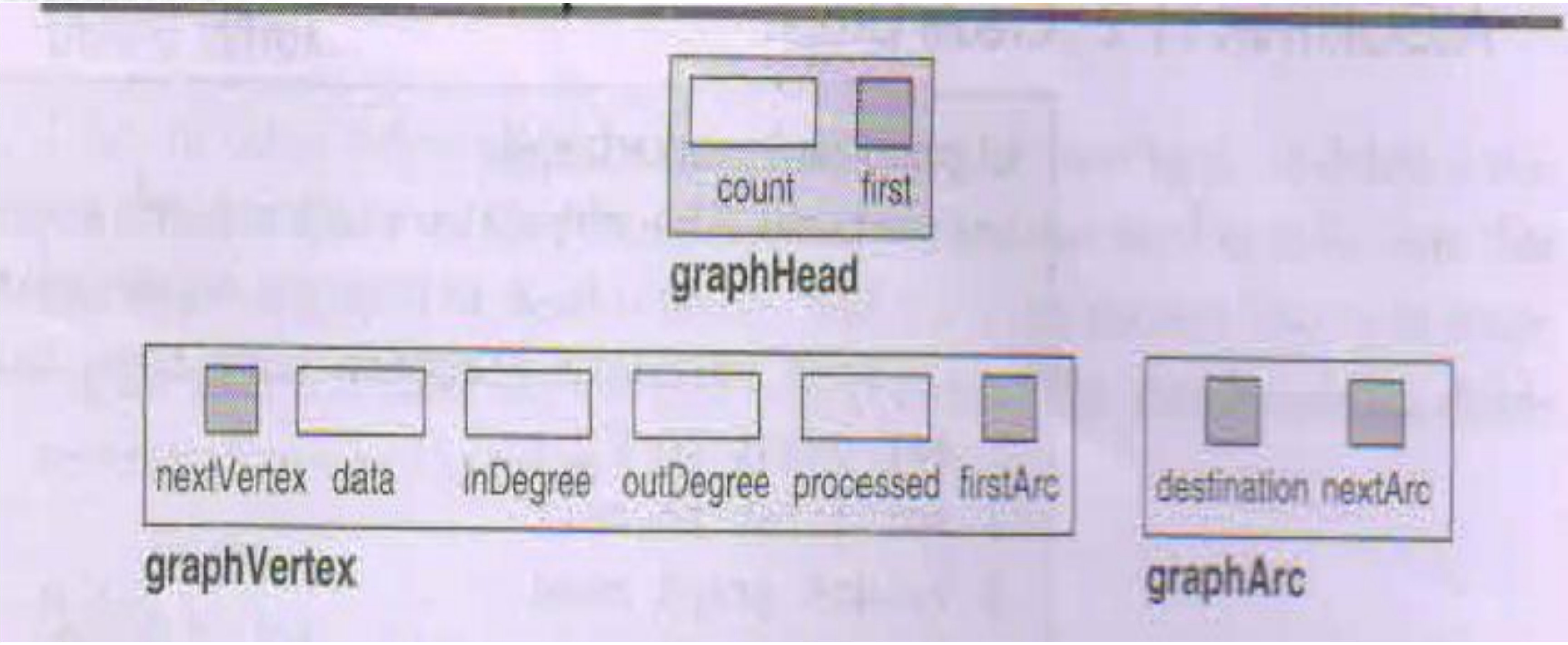
	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

(b) Adjacency matrix for directed graph

Graph – Adjacency List



Graph Data Structure



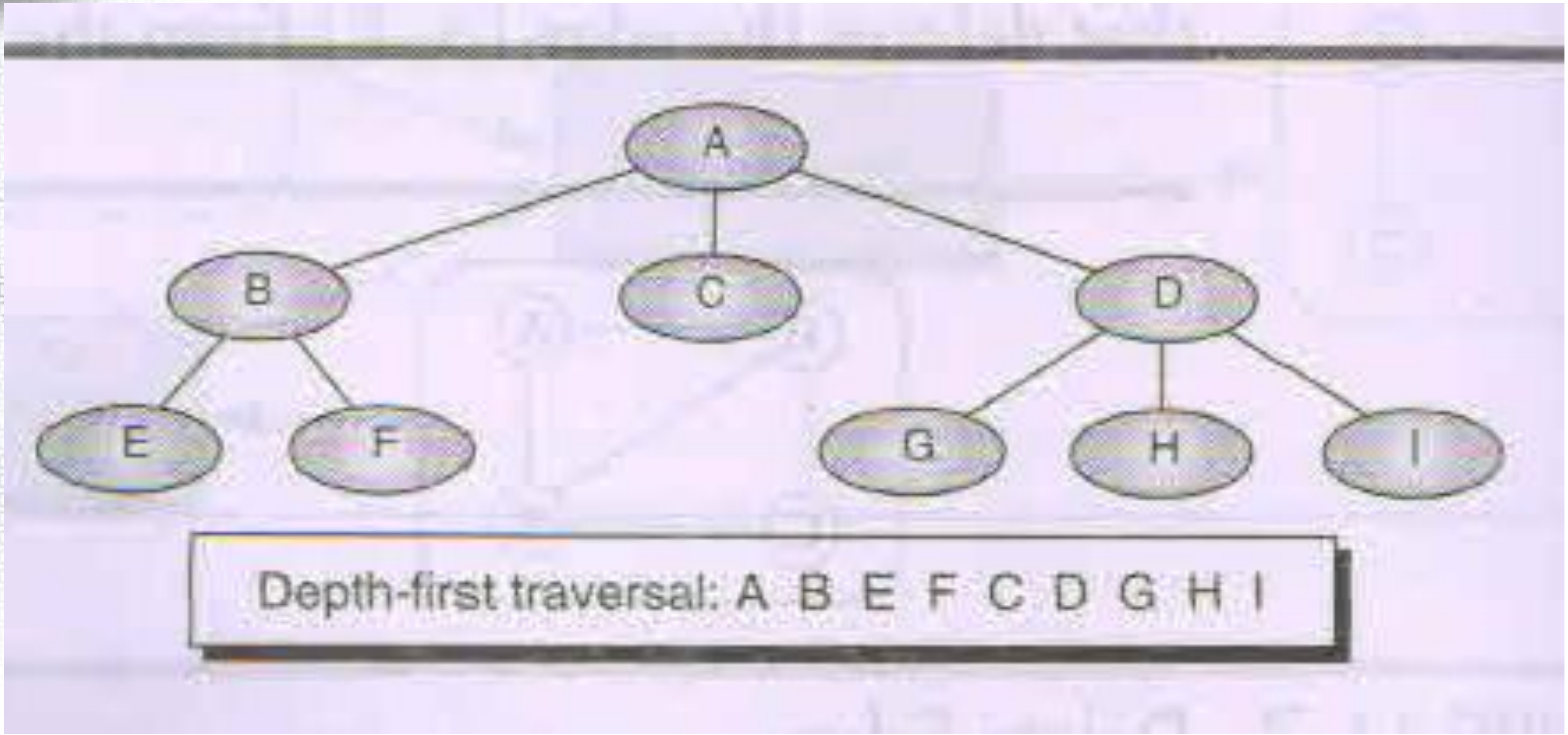
Graph Data Structure

```
graphHead
    count
    first
end graphHead

graphVertex
    nextVertex
    data
    inDegree
    outDegree
    processed
    firstArc
end graphVertex

graphArc
    destination
    nextArc
end graphArc
```

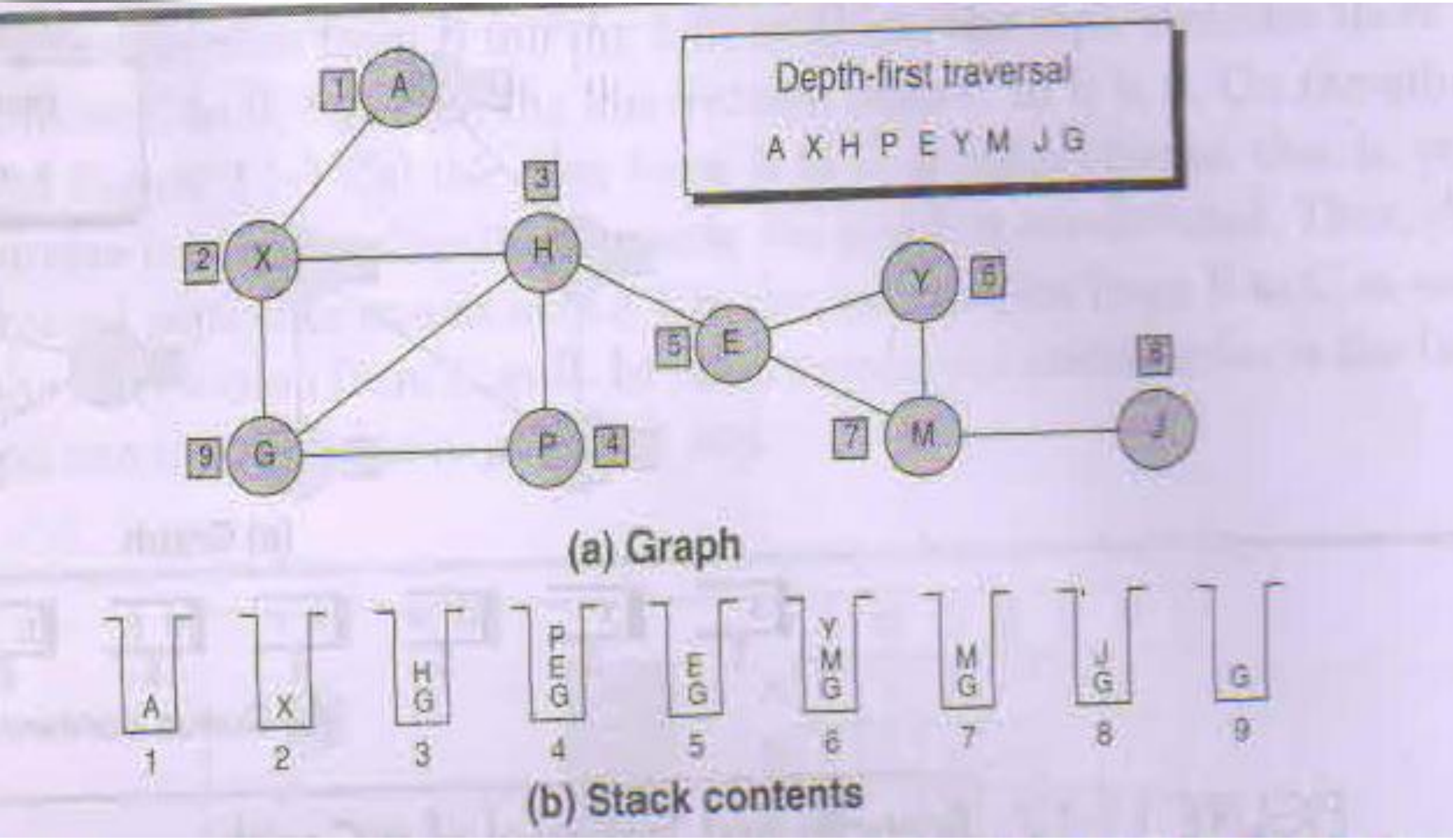
Graph Traversal Techniques

1. Depth-first Search Traversal (DFS)
2. Breadth-first Search Traversal (BFS)

Depth-first Search Traversal (DFS)



Depth-first Search Traversal (DFS)



(a) Graph

(b) Stack contents

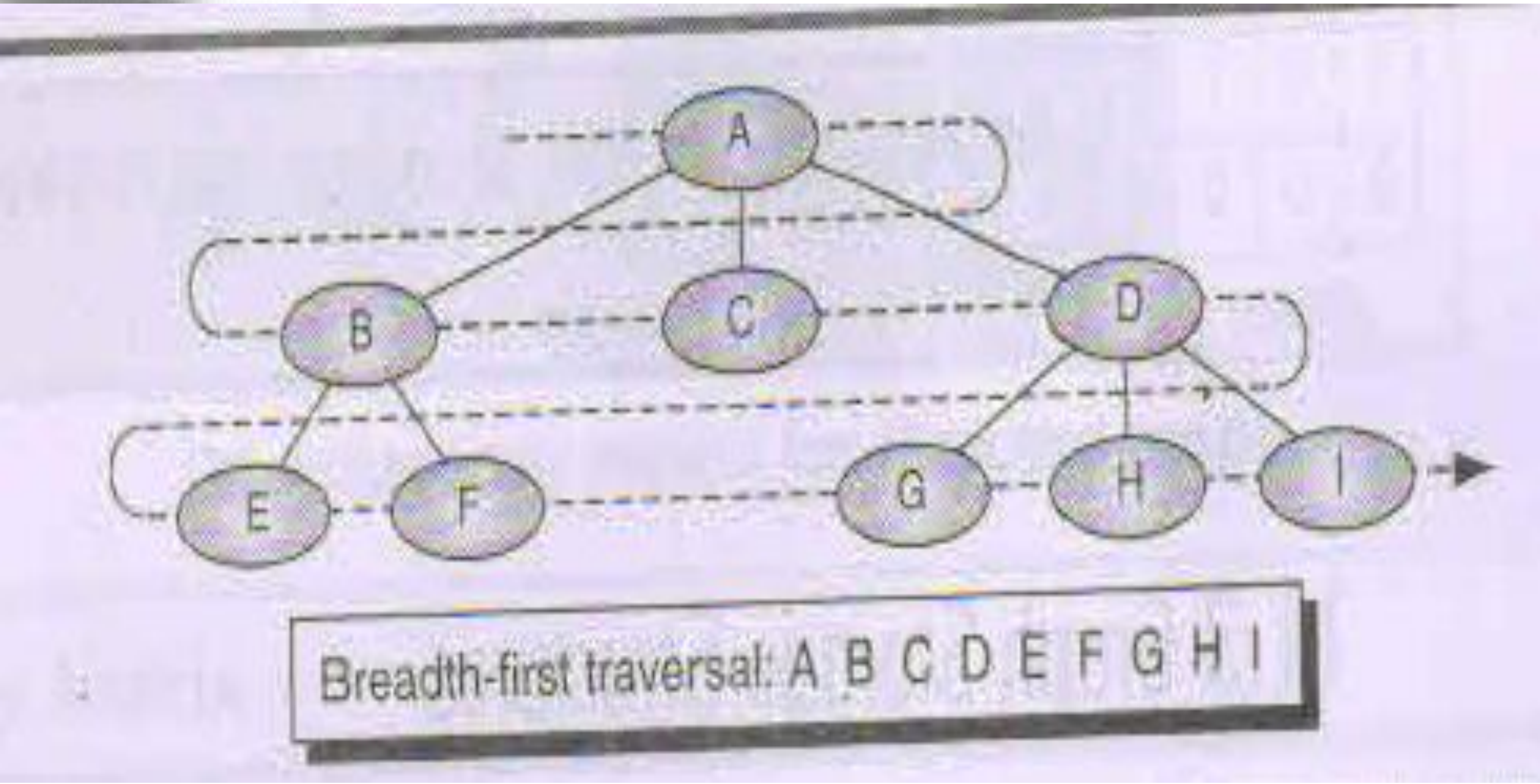
Depth-first Search Traversal (DFS)

This algorithm executes a depth-first search on a graph G beginning at a starting node A.

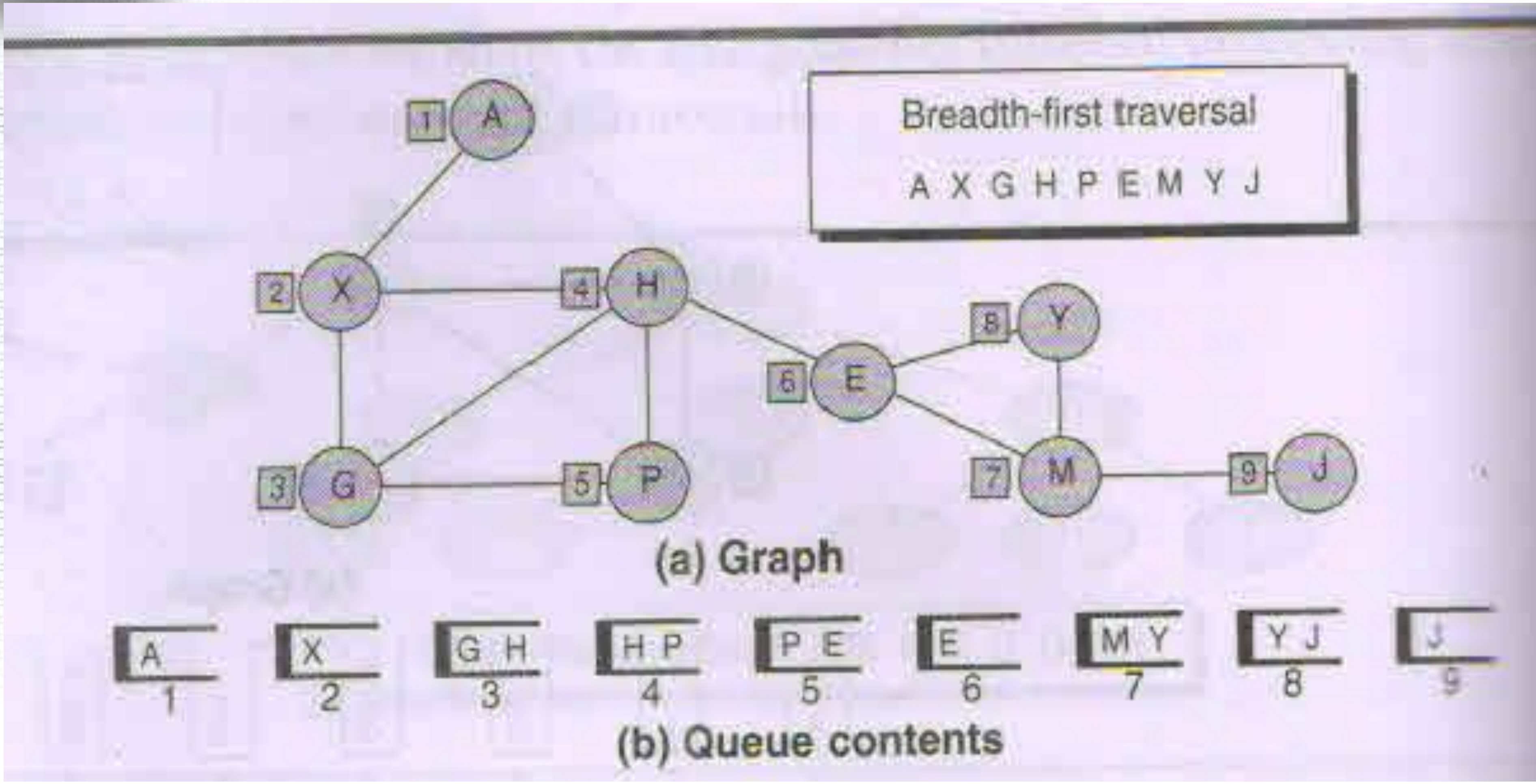
1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A onto STACK and change its status to the waiting state (STATUS = 2).
3. Repeat Steps 4 and 5 until STACK is empty.
4. Pop the top node N of STACK. Process N and change its status to the processed state (STATUS = 3).
5. Push onto STACK all the neighbors of N that are still in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

   [End of Step 3 loop.]
6. Exit.

Breadth-first Search Traversal (DFS)



Breadth-first Search Traversal (DFS)



(a) Graph

(b) Queue contents

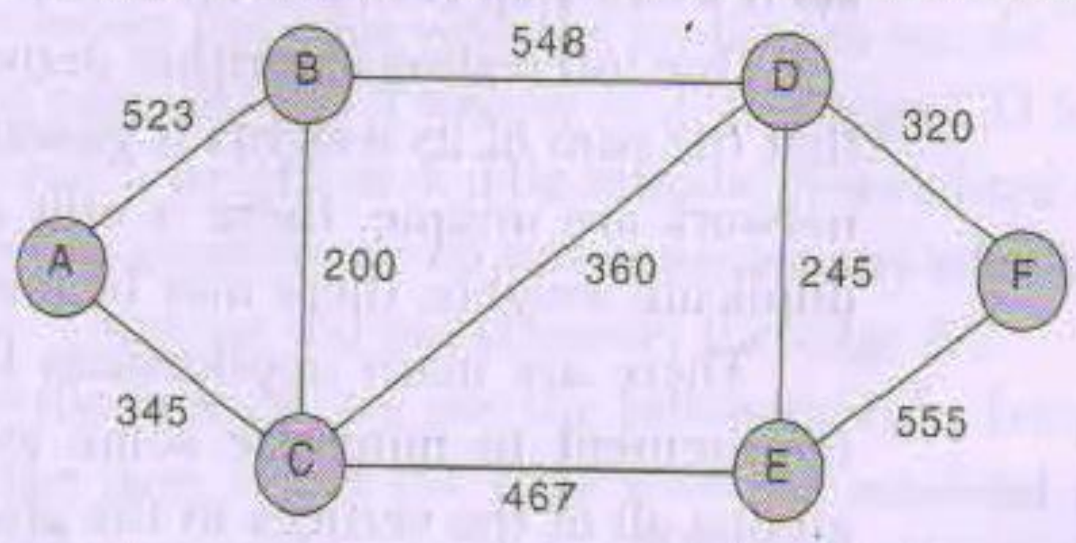
Breadth-first Search Traversal (DFS)

This algorithm executes a breadth-first search on a graph *G* beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).
3. Repeat Steps 4 and 5 until QUEUE is empty:
4. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).
5. Add to the rear of QUEUE all the neighbors of N that are in the steady state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

   [End of Step 3 loop.]
6. Exit.

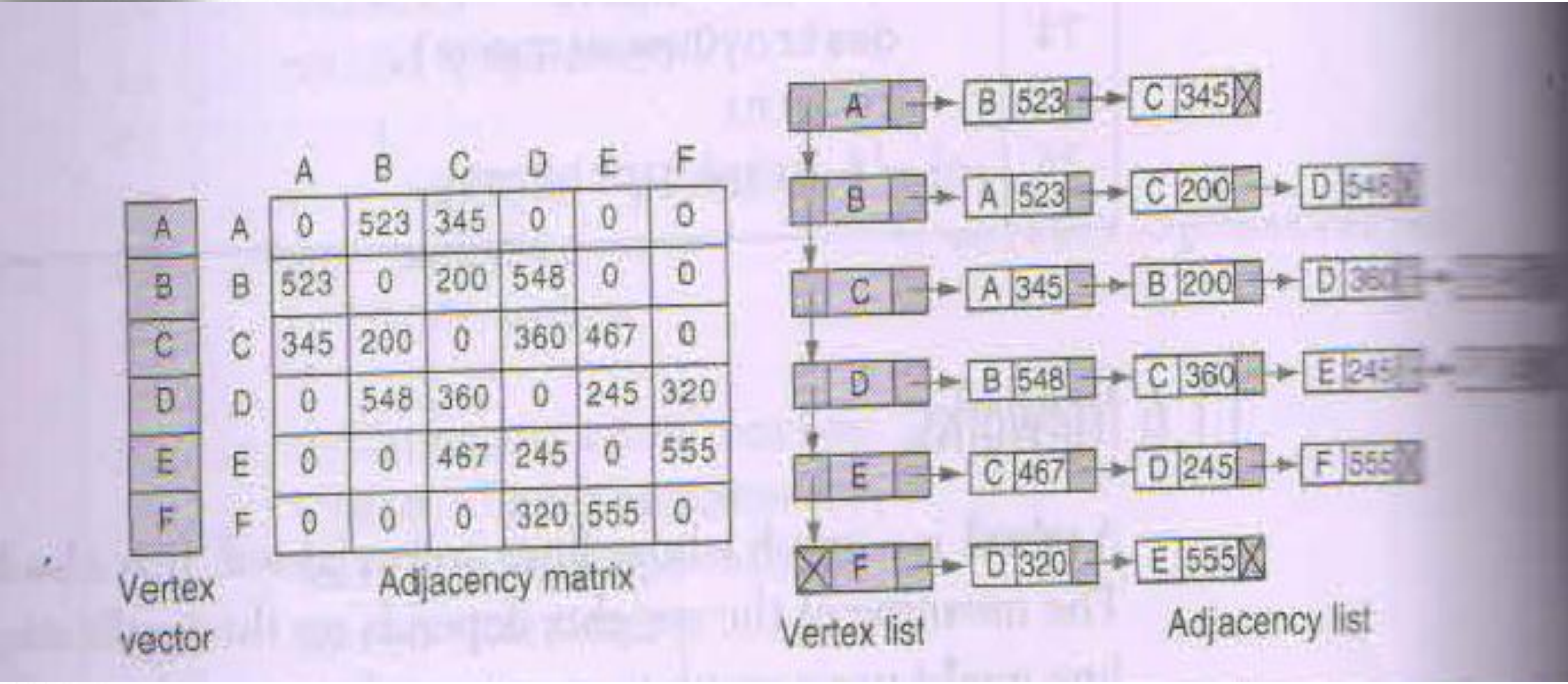
Graph- A Network



City Network

- **Weighted Graph**

Graph- A Network



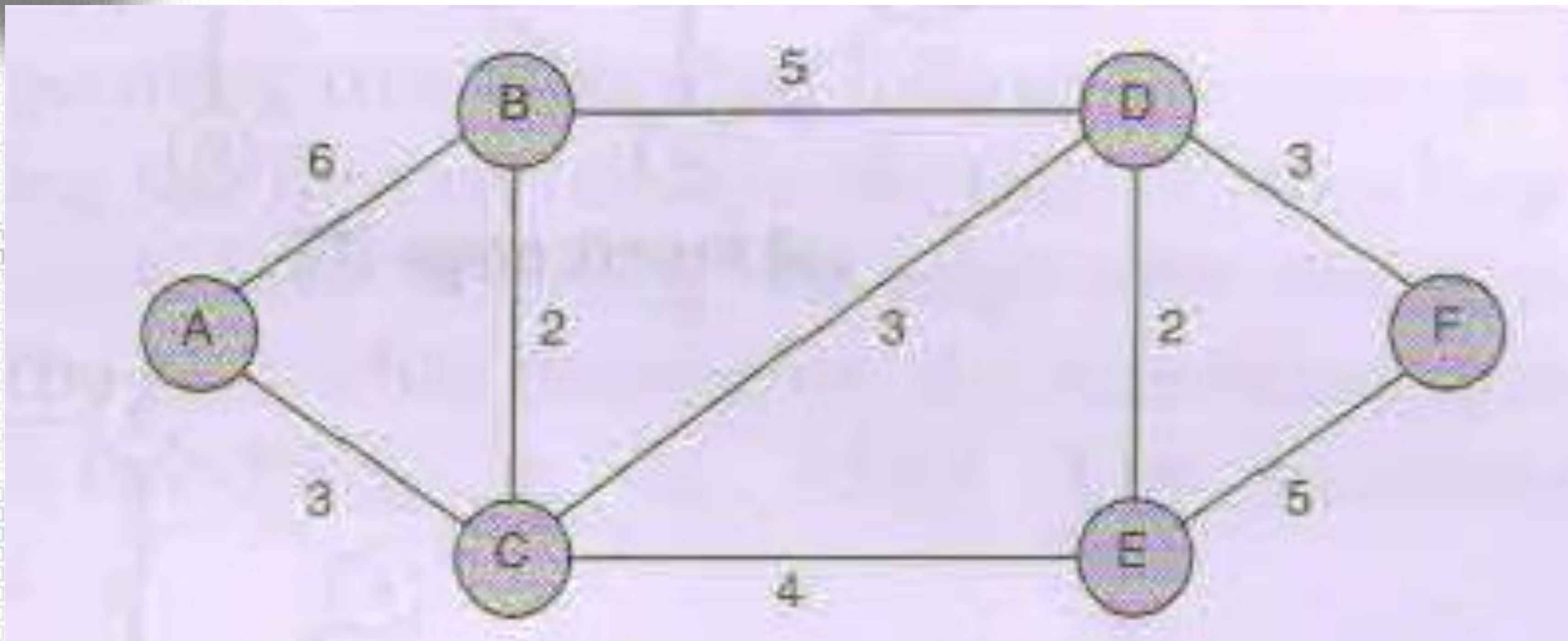
Two Applications of Network

- Spanning Tree
- Shortest Path

Graph- Spanning Tree

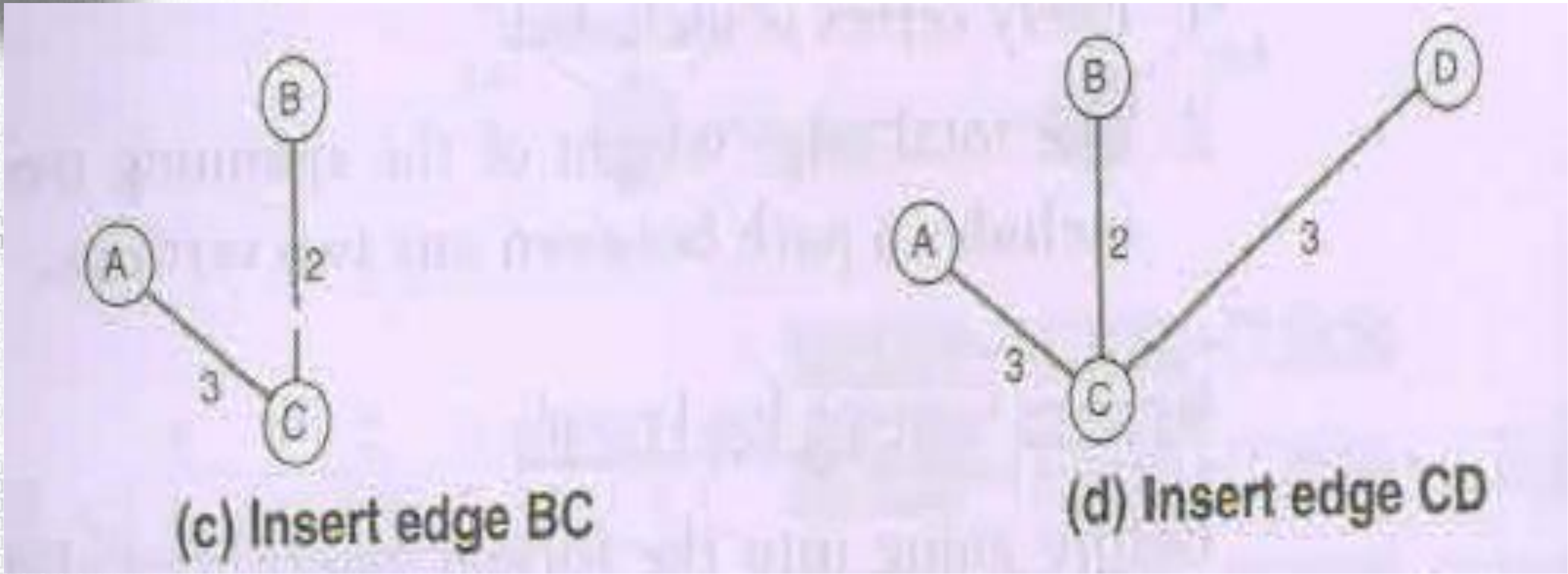
- **A spanning tree contains all of the vertices in a graph.**
- **A minimum spanning tree is a spanning tree in which the total weight of the lines should be the minimum of all possible trees in the graph.**

Graph- Minimum Spanning Tree

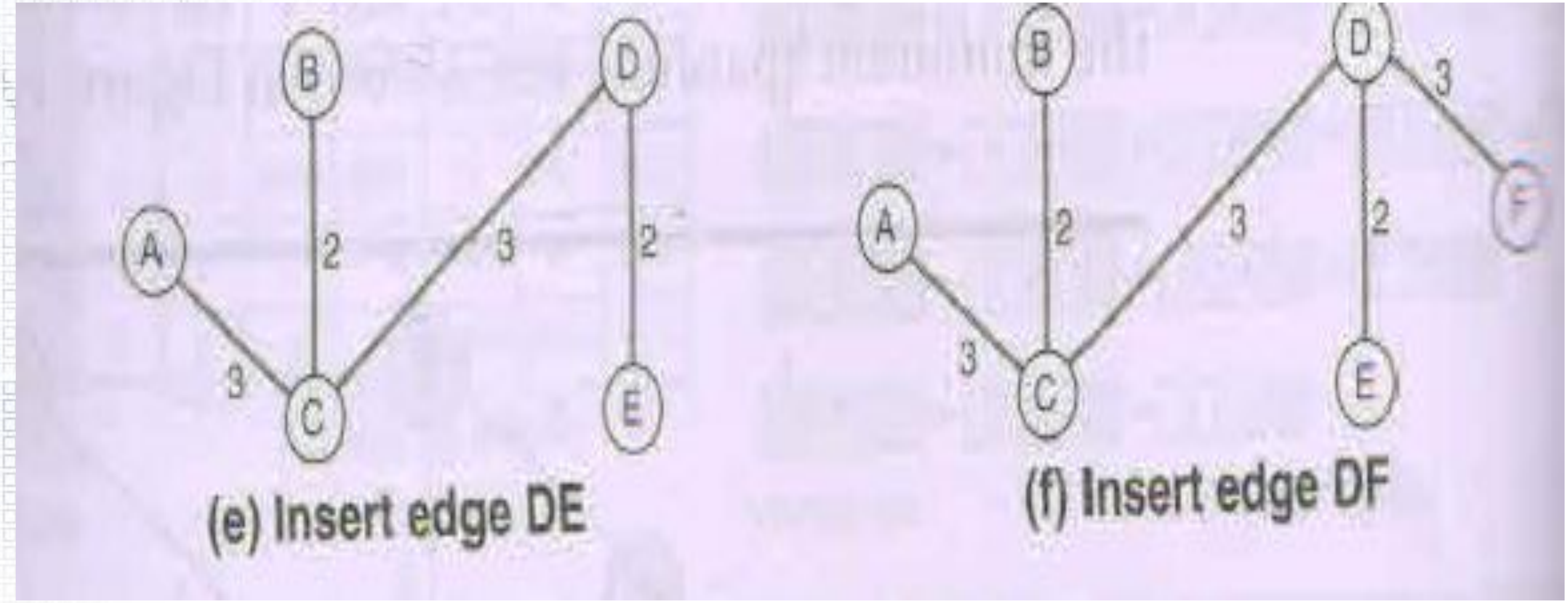


Graph- Minimum Spanning Tree

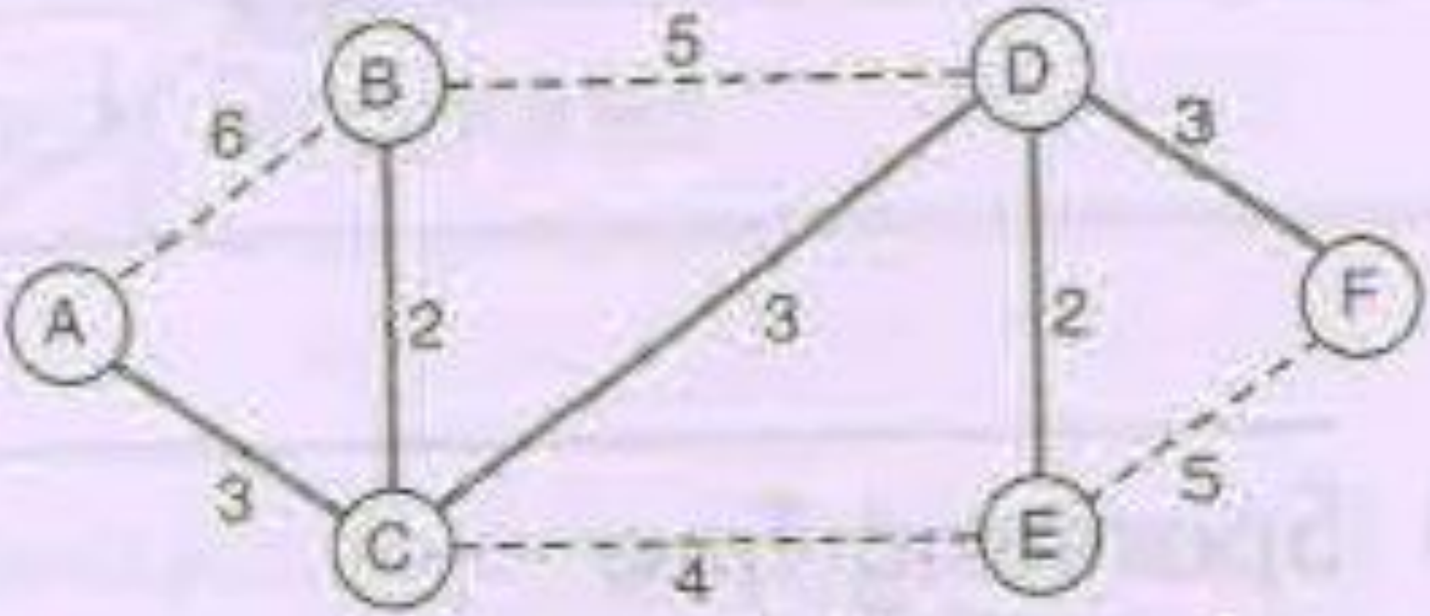
Graph- Minimum Spanning Tree



Graph- Minimum Spanning Tree



Graph- Minimum Spanning Tree



(g) Final tree in the graph

Searching- Importance

- One of the most common and time-consuming operations in computer science

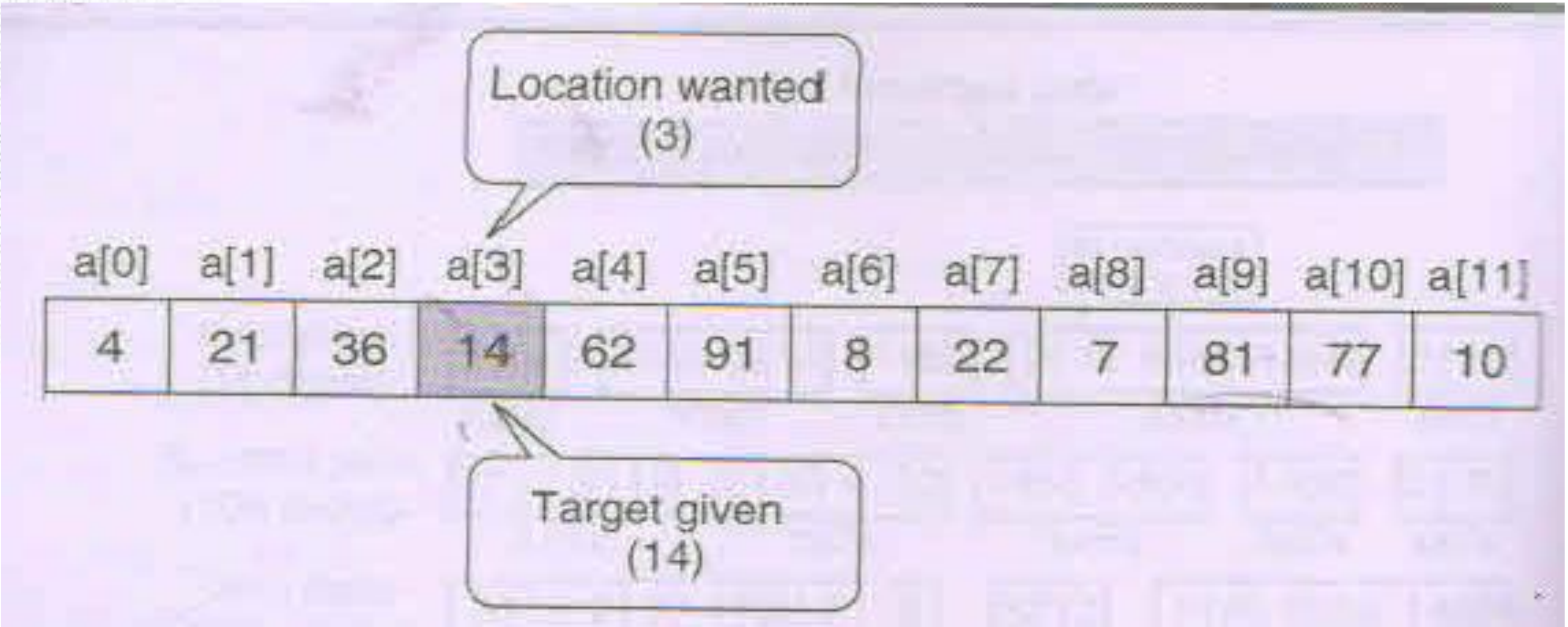
- Search is process of finding a value in a list of values. In other words, Searching is the process of locating given value position in a list of values.

- Searching time should be minimum

Searching- Sequential Search [O(n)]

- When the list is not ordered

- Generally used for small list or lists that are not searched often.



Sequential Search Algorithm

```
Algorithm seqSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
   Pre     list must contain at least one element
           last is index to last element in the list
           target contains the data to be located
           locn is address of index in calling algorithm
   Post    if found: index stored in locn & found true
           if not found: last stored in locn & found false
   Return found true or false
```

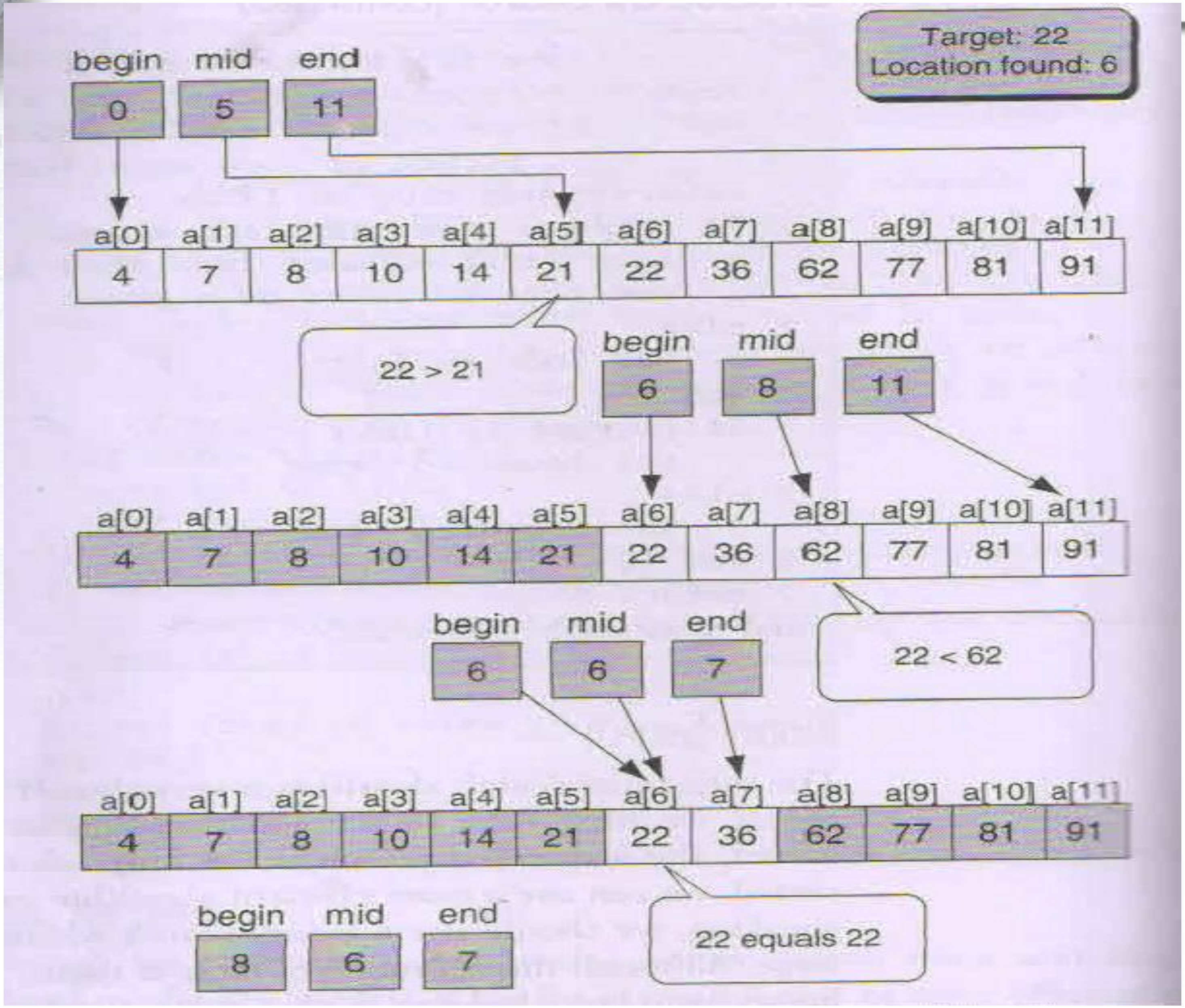
Sequential Search Algorithm

```
1 set looker to 0
2 loop (looker < last AND target not equal list[looker
  1  increment looker
3 end loop
4 set locn to looker
5 if (target equal list[looker])
  1  set found to true
6 else
  1  set found to false
7 end if
8 return found
end seqSearch
```

Searching- Binary Search [O(log(n))]

- When the list is ordered
- Generally used for long list (Greater than 16)
- Mid = $\lfloor$ low + high / 2 $\rfloor$

Searching- Binary Search [O(log(n))]



Searching- Binary Search [O(log(n))]

```
Algorithm binarySearch (list, last, target, locn)
Search an ordered list using Binary Search
   Pre    list is ordered; it must have at least 1 value
          last is index to the largest element in the list
          target is the value of element being sought
          locn is address of index in calling algorithm
   Post   FOUND: locn assigned index to target element
                 found set true
        NOT FOUND: locn = element below or above target
                 found set false
   Return found true or false
```

Searching- Binary Search [O(log(n))]

```
1 set begin to 0
2 set end to last
3 loop (begin <= end)
   1 set mid to (begin + end) / 2
   2 if (target > list[mid])
         Look in upper half
      1 set begin to (mid + 1)
   3 else if (target < list[mid])
         Look in lower half
      1 set end to mid - 1
   4 else
         Found: force exit
      1 set begin to (end + 1)
   5 end if
4 end loop
```

Searching- Binary Search [O(log(n))]

```
4 end loop
5 set locn to mid
6 if (target equal list [mid])
   1  set found to true
7 else
   1  set found to false
8 end if
9 return found
end binarySearch
```

Searching- Fibonacci Search [O(log(n))]

- Given a sorted array arr[] of size n and an element x to be searched in it. Return index of x if it is present in array else return -1.

- Fibonacci Numbers are recursively defined as F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1.

- First few Fibonacci Numbers are

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci Search [O(log(n))]

Similarities with Binary Search:

- Works for sorted arrays
- A Divide and Conquer Algorithm.
- Has O(log n) time complexity.

Fibonacci Search [O(log(n))]

Differences with Binary Search:

- Fibonacci Search divides given array in unequal parts
- Binary Search uses division operator to divide range. Fibonacci Search doesn’t use /, but uses + and -. The division operator may be costly on some CPUs.
- Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful.

Fibonacci Search [O(log(n))]

Example

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	-	-

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Smallest Fibonacci number greater than or equal to 11 is 13. In which, fibMm2 = 5, fibMm1 = 8, and fibM = 13.

Fibonacci Search [O(log(n))]

fibMm2	*fibMm1*	*fibM*	*offset*	i=min(offset+fibL n)	*arr[i]*	*Consequence*
5	8	13	0	5	45	Move one down, reset offset
3	5	8	5	8	82	Move one down, reset offset
2	3	5	8	10	90	Move two down
1	1	2	8	9	85	Return i

Fibonacci Search [O(log(n))]

Example

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	-	-

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci Search [O(log(n))]

Example

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	-	-

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci Search [O(log(n))]

Example

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	-	-

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci Search [O(log(n))]

Example

i	1	2	3	4	5	6	7	8	9	10	11	12	13
ar[i]	10	22	35	40	45	50	80	82	85	90	100	-	-

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Fibonacci Search- Algorithm

Observations:

Below observation is used for range elimination

F(n - 2) approx (1/3) * F(n) and

F(n - 1) approx (2/3) * F(n).

and hence for the O(log(n)) complexity.

Fibonacci Search- Algorithm

Let the searched element be x.

The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of given array. Let the found Fibonacci number be fib (m'th Fibonacci number). We use (m-2)'th Fibonacci number as the index (If it is a valid index). Let (m-2)'th Fibonacci Number be i, we compare arr[i] with x, if x is same, we return i. Else if x is greater, we recur for subarray after i, else we recur for subarray before i.

Fibonacci Search- Algorithm

Below is the complete algorithm

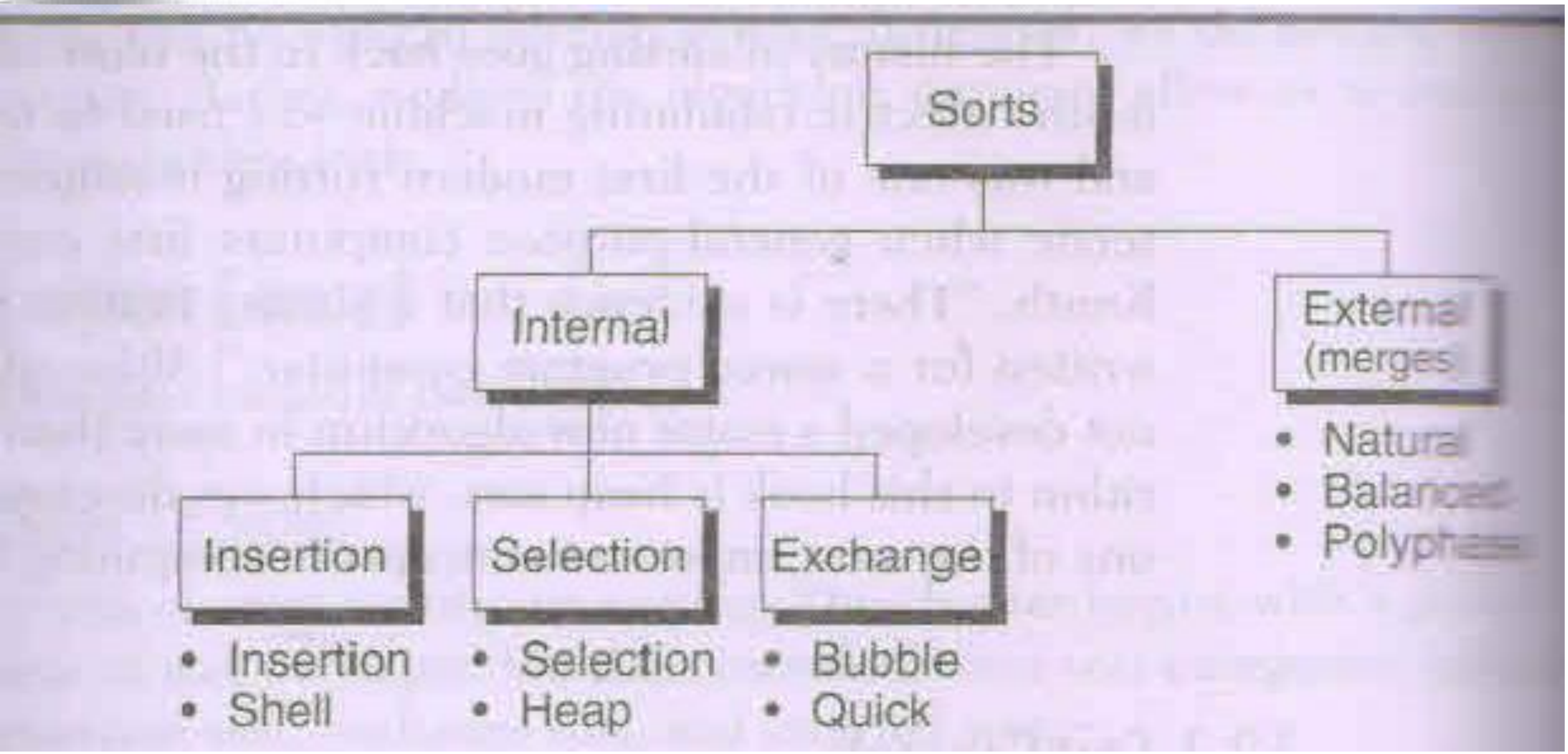
Let arr[0..n-1] be the input array and element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m'th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].

Fibonacci Search- Algorithm

2. While the array has elements to be inspected:
   i. Compare x with the last element of the range covered by fibMm2
   ii. **If** x matches, return index
   iii. **Else If** x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
   iv. **Else** x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

Sorting


Sorts
Internal
Insertion
Selection
Exchange
Insertion
Shell
Selection
Heap
Bubble
Quick

Sorting

365	blue
212	green
876	white
212	yellow
119	purple
737	green
212	blue
443	red
567	yellow

(a) Unsorted data

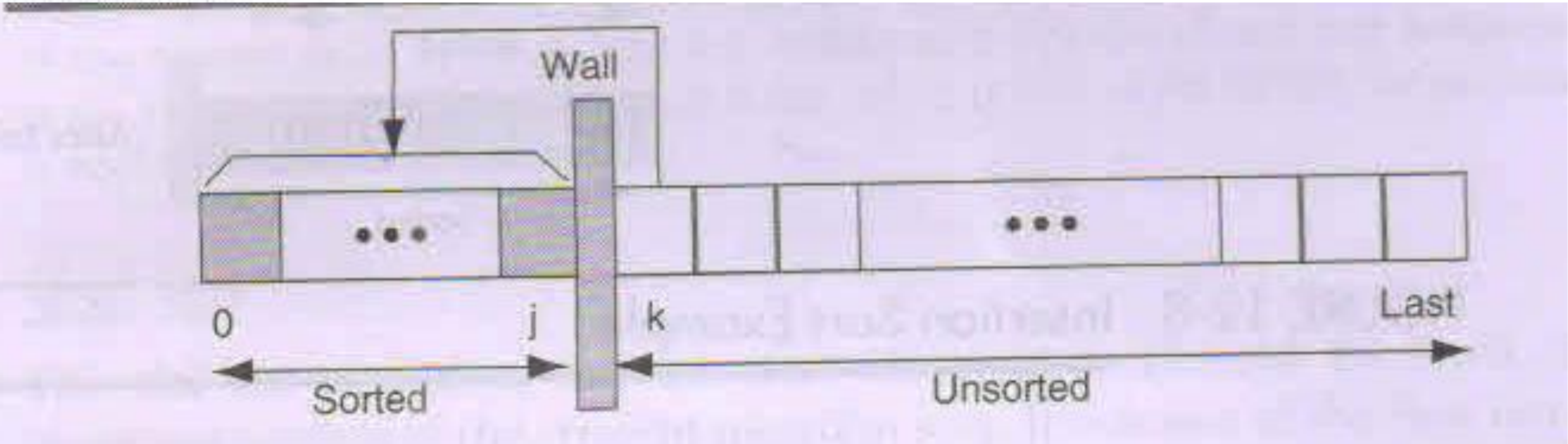
119	purple
212	green
212	yellow
212	blue
365	blue
443	red
567	yellow
737	green
876	white

(b) Stable sort

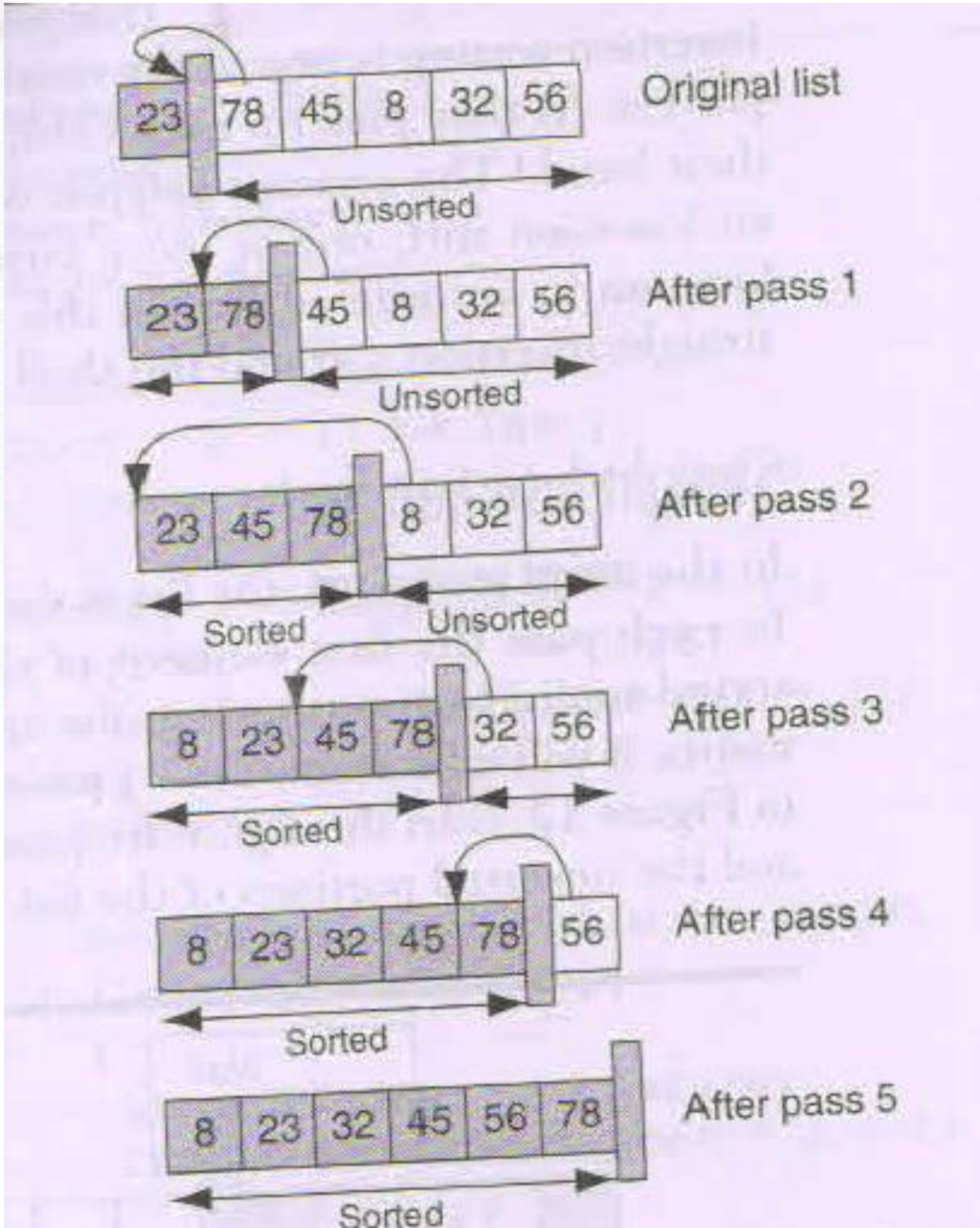
119	purple
212	blue
212	green
212	yellow
365	blue
443	red
567	yellow
737	green
876	white

(c) Unstable sort

Insertion Sort



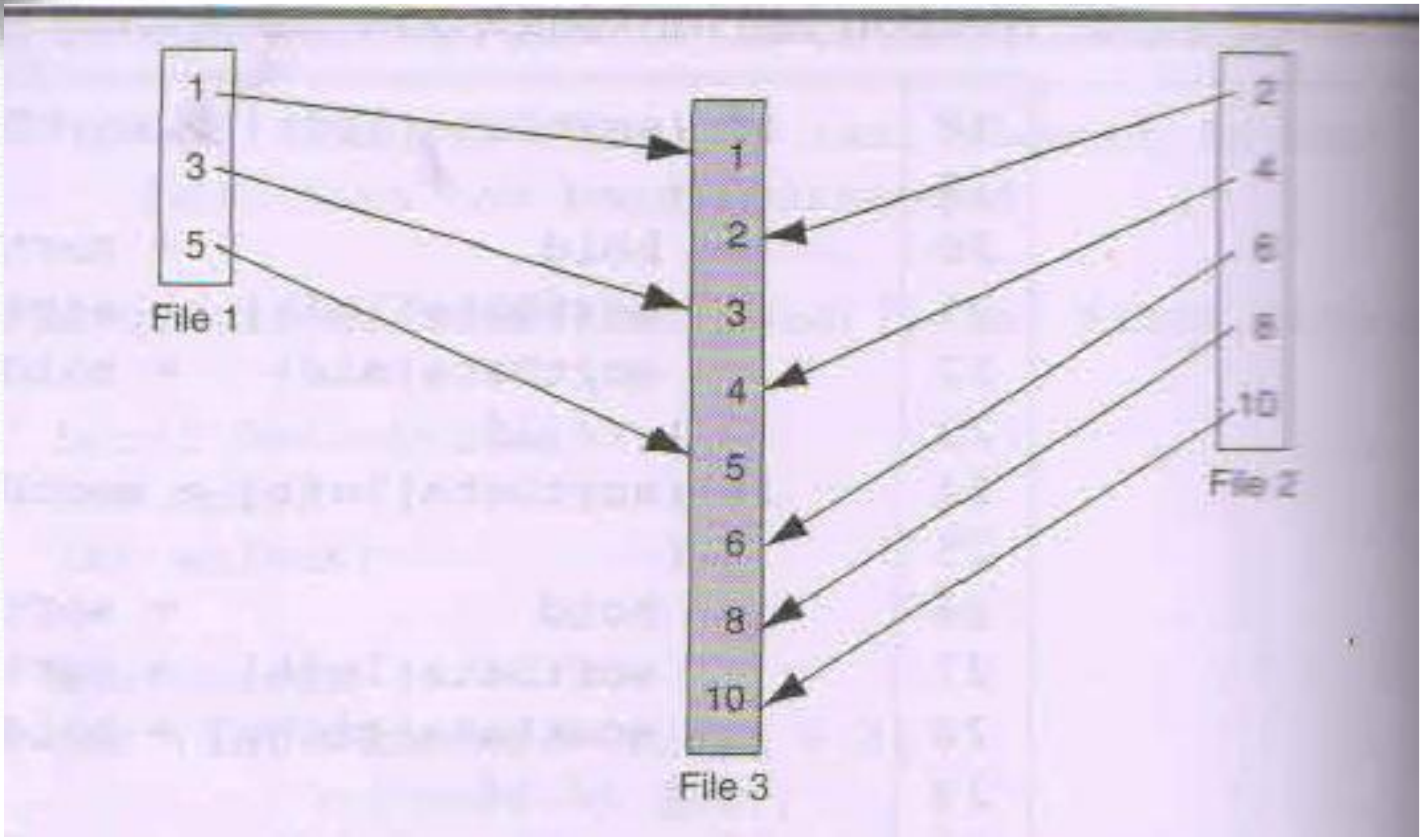
Insertion Sort



Insertion Sort

```
Algorithm insertionSort (list, last)
Sort list array using insertion sort. The array is
divided into sorted and unsorted lists. With each pass, the
first element in the unsorted list is inserted into the
sorted list.
   Pre  list must contain at least one element
        last is an index to last element in the list
   Post list has been rearranged
 1 set current to 1
 2 loop (until last element sorted)
   1 move current element to hold
   2 set walker to current - 1
   3 loop (walker >= 0 AND hold key < walker key)
      1 move walker element right one element
      2 decrement walker
   4 end loop
   5 move hold to walker + 1 element
   6 increment current
 3 end loop
end insertionSort
```

Merging two Sorted Files



Merging two Sorted Files

```
Algorithm mergeFiles
Merge two sorted files into one file.
   Pre     input files are sorted
   Post    input files sequentially combined in out
1 open files
2 read (file1 into record1)
3 read (file2 into record2)
4 loop (not end file1 OR not end file2)
   1  if (record1 key <= record2 key)
      1  write  (record1 to file3)
      2  read   (file1 into record1)
      3  if (end of file1)
         1  set record1 key to infinity
      4  end if
   2  else
      1  write  (record2 to file3)
      2  read   (file2 into record2)
      3  if (end of file2)
         1  set record2 key to infinity
      4  end if
   3  end if
```

Merge Sort [O(n log n)

Suppose the array A contains 14 elements as follows:

66, 33, 40, 22, 55, 88, 60, 11, 80, 20, 50, 44, 77, 30

Each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows:

Pass 1. Merge each pair of elements to obtain the following list of sorted pairs:

33, 66   22, 40   55, 88   11, 60   20, 80   44, 50   30, 70

Pass 2. Merge each pair of pairs to obtain the following list of sorted quadruplets:

22, 33, 40, 66   11, 55, 60, 88   20, 44, 50, 80   30, 77

Merge Sort [O(n log n)]

Pass 3. Merge each pair of sorted quadruplets to obtain the following two sorted subarrays:

11, 22, 33, 40, 55, 60, 66, 88   20, 30, 44, 50, 77, 80

Pass 4. Merge the two sorted subarrays to obtain the single sorted array

11, 20, 22, 30, 33, 40, 44, 50, 55, 60, 66, 77, 80, 88

The original array A is now sorted.

Radix Sort

Suppose 9 cards are punched as follows:

348, 143, 361, 423, 538, 128, 321, 543, 366

Worst Case $O(n^2)$
Best Case $O(n \log n)$

Radix Sort

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

Radix Sort

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass

Radix Sort

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass

Heap Sort