CLASSIFICATION OF DATA STRUCTURES

Introduction.

Data structures :- Organising or structuring of data in such a way that operations on that data can be performed in an efficient & efficient way. (or)

A data structure is a systematic method of organizing, storing, and managing data in a computer so that it can be accessed, processed, and modified efficiently.

Data structures :-

(Data organization + set of operations + Efficient constraints).

→ Data organization :- It refers to how data elements are arranged in memory.

→ operations refer to the set of actions that can be performed on the data.

→ performace constraints involve time and space efficiency of these operations.

Need for data structures.

1. Data base management systems
2. Operating systems
3. Artificial Intelligence
4. Computer networks
5. file systems
6. compiler design
7. Efficient algorithm implementation
8. Optimal time / space usage
9. Real world problem modeling
10. foundation for advanced computing.

→ Without proper data structures, handling large volumes of data efficiently would be extremely difficult.

Ex :- Retriving the students according to their roll numbers.
Efficient way is placing according to their roll numbers.

Abstract Data types (ADTs)

An Abstract Data types (ADT) is a mathematical model that defines data and the operations that can be performed on the data, without specifying the implementation details.

→ It mainly focuses on what operations are performed not how they are implemented

for example, a stack ADT may define the following operations.

- push()
- pop()
- peek() | top()
- isEmpty()

The stack can be implemented using

(1) Array implementation

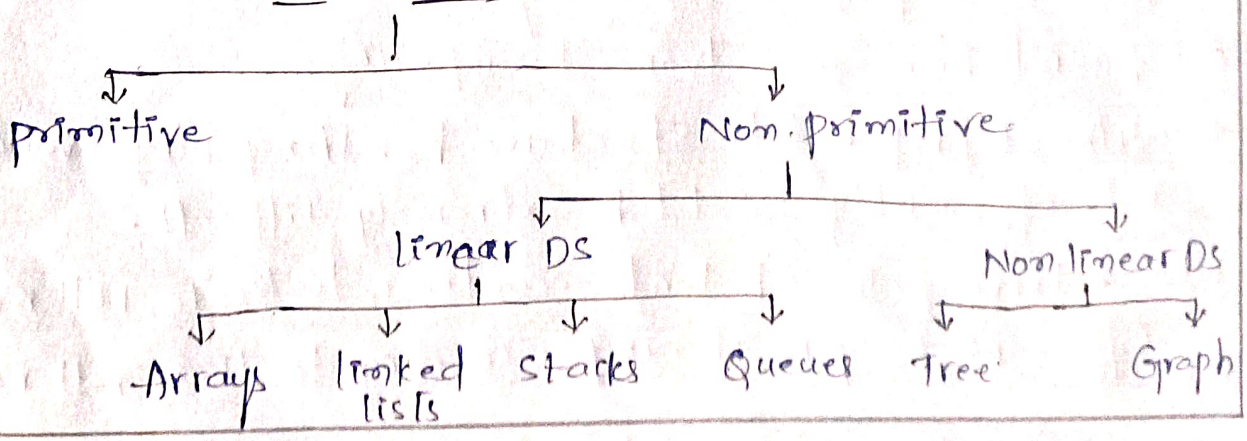
(2) Linked list implementation.

Both implementations satisfy the same ADT specifications.

Advantages.

1. Encapsulation (Data Security)
2. Abstraction
3. Modularity
4. Data Structural independence
5. Reusability.

Classification of DS



Primitive data structures

→ primitive data structures as basic data types provided directly by the programming language.

→ They store single value and are used as building blocks for more complex data structures.

Type	Description	operations	Memory
Int	interger values	+, -, *, /, %	fixed
float	Real numbers	Arithmetic	Fixed
Char	Single character	Assignment	fixed
bool	True / False	logical	fixed.

Characteristics

1. They are atomic in nature, meaning they cannot be divided into smaller components.
2. They occupy fixed memory size.
3. Memory allocation is handled by the complier or programming language.
4. They support basic arithmatic and logical operations.

Non primitive data structures.

1. They store collections of multiple data elements
2. These structures are user defined and allow efficient management of large data sets.

Non primitive structures further divided into 2 types.

→ Linear data structures
→ Non linear data structures.

1) Linear Data Structures.

In linear data structures, elements are arranged sequentially order. Each element is connected to its previous and next element. But their memory allocations ary may or may not sequential.

The traversal of elements occur in a single sequentral path.

- Arrays
- Linked lists
- Stacks
- Queues

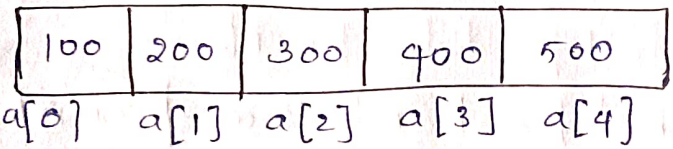
Characteristics

1. Data elements are arranged in sequential order
2. Each element has a unique predecessor and successor. (Except first and last).
3. Traversal can be performed in single pass.
4. They are easier to implement compared to non linear structures.

(i) Array data structures

An array is a static linear data structures consisting of fixed size homogeneous (same type) element stored in contiguous memory allocations. Each element is accessed using a zero-based index.

Memory representation



Index formula.

address(arr[i]) = base + i x sizeof(type)

Key Characteristics

→ Fixed size declared at compile-time
→ Contiguous allocation (no gaps)
→ Random access via index
→ Cache-friendly (spatial locality)
→ Homogeneous elements only

Operations on arrays

(1) Insertion:

- Insertion means adding a new element at a specified position.
- To insert an element, existing elements must be shifted to the right.

Worst case: Insert at beginning (shift all n elements)

Worst Case Complexity O(n).

Algorithm: (Insert at position k)

Insert (arr, n, item, k)

1. for i = n-1 down to k
2. arr[i+1] = arr[i]
3. arr[k] = item
4. n = n+1

Mined representation.

Initial = [10, 20, 30, 40] n=4

insert 25 at k=2

Step 1: [10, 20, 30, 40]

Step 2: [10, 20, 25, 30, 40] n=5

Time Complexity O(n-k) position dependent

(2) Deletion

Deletion removes an element from a particular position.

→ Deletion requires shifting all elements from deletion point +1 to end left by one position. Worst Case - Delete from beginning.

Algorithm. (Delete at position k)

Delete (arr, n, k)

1. for (i=k to n-2)
2. arr[i] = arr[i+1]
3. n = n-1

Visual trace

Initial : [10, 20, 30, 40] n=4

Delete at k=1 [20]

Step 1 : [10, 30, 40, 40]

Step 2 : [10, 30, 40] n=3.

Time Complexity : O(n-k), O(n) worst case

Traversal Operation

Definition : Visiting each element exactly once in sequential order.

Theory :- Array traverser leverages random access property. Start from index 0 to n-1 accessing each element in constant time O(1) for element, total O(n).

Algorithm :-

Traversal (arr, n)

1. for (i=0 to n-1)
2. print arr[i]
3. End for.

C - implementation

```
void traverse (int arr[ ], int n)
for (int i=0, i<n, i++)
{ printf("arr[%d] = %d \n", i, arr[i]);
 }
}
```

Example trace :-

text

Array [10, 20, 30, 40] n=4

i=0; arr[0]=10

i=1; arr[1]=20

i=2 ; all[2] = 30

i=3; all[3] = 40

Time : O(n)

Searching operation :

Finding index of given element

Types :

1. Linear Search : O(n) – unsorted
2. Binary Search : O(log(n)) – sorted array.

Linear Search algorithm :-

Search (arr, n, key)

1. for i=0 to n-1
2. If arr[i] == key
3. Return
4. Return -1 // Not found

Binary Search (sorted array)

Binary-Search (arr, n, key)

1. low = 0, high = n-1
2. While low < high
3. mid = low + (high - low)/2
4. If arr[mid] == key return mid
5. Else if arr[mid] > key
6. high = mid - 1
7. Else low = mid + 1
8. Return -1.

Update operation :

Replace element at specific index

Theory: Simplest operation – direct assignment O(1) time.

Algorithm :

update (arr, index, new-value)

1. arr[index] = new-value

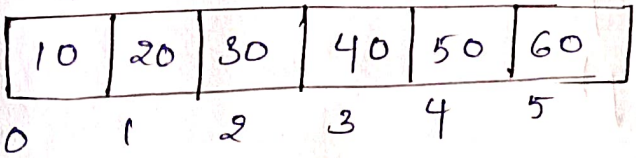
Example:
Initial: [10, 20, 30, 40]
update arr[1] = 25
Result: [10, 25, 30, 40] O(1)

3. Complexity Summary table

operation	Best Case	Average	worst case	space
Insert	O(1) end	O(n)	O(n) start	O(1)
Delete	O(1) end	O(n)	O(n) start	O(1)
Traversal	O(n)	O(n)	O(n)	O(1)
Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
update	O(1)	O(1)	O(1)	O(1)

Array representation - Mixed format

Array visual representation.



n = 6
indices.

insert 25 at pos = 2
Step by step shifting
[10] [20] [30] [40] [50] [60]

Algorithm prelimaries

Algorithm Definition
It is a step by step process to solve a problem.
Time Complexity: The amount of time required to run an algorithm (CPU time).
Space Complexity: The amount of space is required to run an algorithm (Memory space.

Calculation of time complexity.

→ frequency count method

The time complexity for addition of two numbers.

Algorithm step	count	Frequency	Exc
1. Algorithm add	0	1	0
2. start	0	1	0
3. Read 2 number into a,b	1	1	1
4. Add a,b show the result in c	1	1	1
5. print c	1	1	1
6. stop	0	1	0

Total no. of steps = 3

If we know the no. of constat steps in a algorithm then we indicate it as O(1).

Advantages & Limitations

Advantages

- Direct random access
- simple Implementation
- Efficient memory access due to contiguous storage.
- Sustainable for numerical compuations.
- Cache efficient
- predicatable performance.

Limitations

- fixed size (static)
- Insert / Delete O(n) shifting
- Wasted space if underutilized
- Memory fragmentation.
- possible memory wastage.

Applications

perfect for look up tables, matrices, Implementating other DS (stake, queue).

(2) Linked list data structures

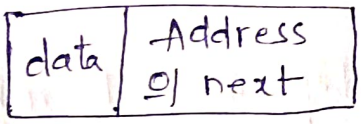
Definition: Linked lists are dynamic and linear data structure which arranges the elements as collection of nodes containing data and pointer to next node. Elements stored in non contiguous memory locations connected via pointers.

Linked lists are 2 types

1. Single linked list
2. Double linked list
3. Circular linked list

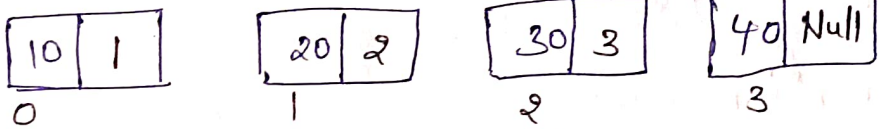
(1) Single linked list

single linked list consisting of two parts i.e;



Memory representation.

logical view:



Physical:

Node structure:

```
Struct Node
{ int data;
  Struct Node * next;
};
```

Key characteristics of linked lists

→ Dynamic size (runtime allocation)
→ Non contiguous Sattered memory
→ Sequential access only O(n)
→ O(1) insert / delete at head
→ No wasted space.

operation on linked lists

1) Traversal operation

Definition: Visiting each node sequentially from head to NULL

→ It take single pass to traverse through a linked list

→ Starts from head, follow the next pointer until NULL terminator. O(n) linear time.

Algorithm.

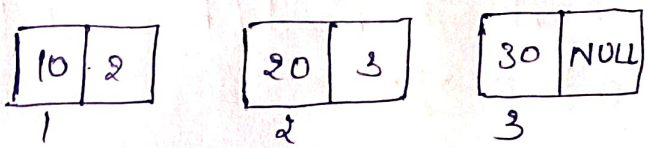
Traversal (head)

1. Current = head
2. While (current != NULL)
3. print current → data
4. Current = Current → next

C Implementation

```
Void traverse (struct Node *head)
{ Struct Node* Current = head;
   While (Current != NULL)
   { printf ("%d", current → data);
     Current = Current → next;
   }
}
```

Example



Current = 10 → 20 → 30 → NULL O(n).

Insert at Head position

Adding new node at head position.

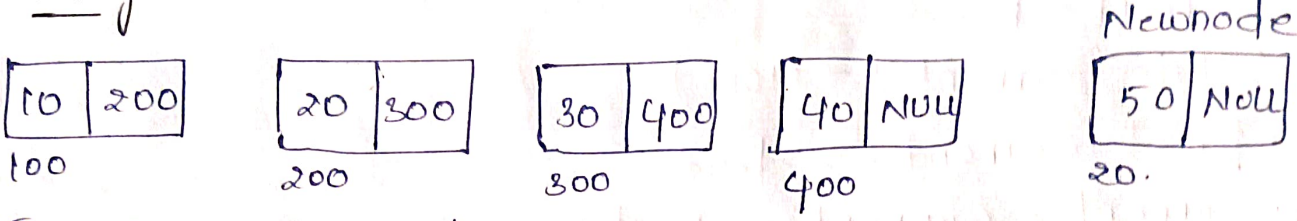
→ creating node, point to current head, updata head O(1) Constant time.

Algorithm.

insert-head (head, data)

1. newNode = malloc (Size of (Node))
2. newNode → data = data
3. newNode → next = head
4. head = newNode
5. Return head.

Tracing



→ Insert 50 at head position.

50 → 10 → 20 → 30 → 40 → NULL.

Insertion at K position

Insertion a node a K^{th} position.

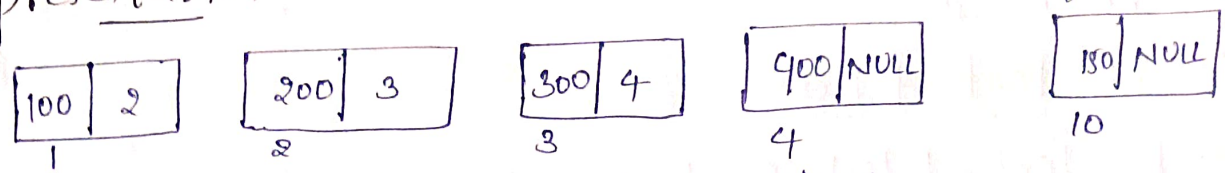
First we have to traverse to (K-1)th node, K^{th} node. insert between (K-1) and K. O(K) time.

Algorithm.

insert-at-position (head, data, K)

1. newNode → data = data
   newNode → next = NULL
2. If K == 0 Return insert-at-head (head, data)
3. p = head ; for (i = 0 to K-2);
4. p = p → next
5. q = q → next
6. p → next = newnode
   newnode → next = q
7. Return head.

Representation



→ newNode have to Insert at 2 position

then p = 100, q = 200

100 → 150 → 200 → 300 → 400 → NULL

Deletions

(i) Delete at head position

→ Remove the first node

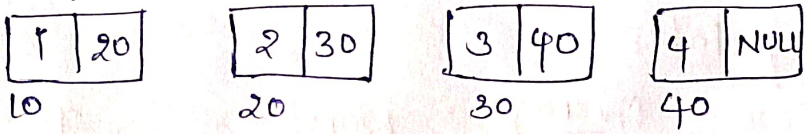
Save head, move head to head → next, free old head O(1)

Algorithm

delete_head (head)

1. if head == NULL, Return NULL
2. temp = head;
3. head = head → next
4. free (temp)
5. Return head;

Tracing



After deletion : 2 → 3 → 4 → NULL

(ii) Deletion at k^{th} position

Algorithm

Delete_position (head, k)

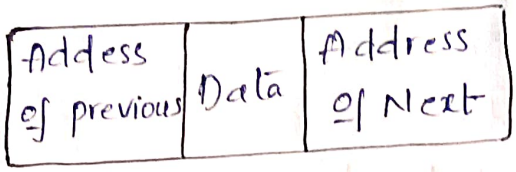
1. if k = 0, Return delete_head (head
2. p = head : for i = 0 to k-2 p = p → next
3. q = p → next
4. p → next = q → next

5. free(q)
6. return head

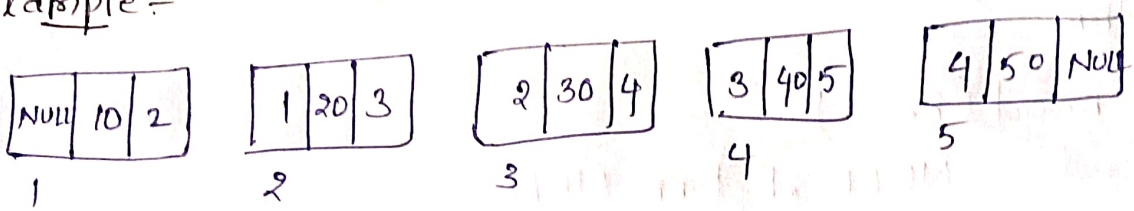
(2) Double linked list

Double linked list is a linear data structures which arranges the elements as collection of nodes that are linked with each other.

→ Here node consisting of 3 parts

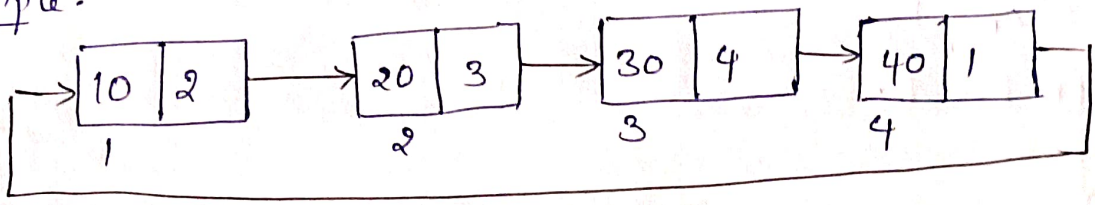


example :-



Circular linked list

Example :



Complexity Summary table

operation	Best case	Average	Worst case	space
Traversal	O(n)	O(n)	O(n)	O(1)
Insert head	O(1)	O(1)	O(1)	O(1)
Insert tail	O(n)	O(n)	O(n)	O(1)
Delete head	O(1)	O(1)	O(1)	O(1)
Search	O(1)	O(n)	O(n)	O(1)

Advantages & Limitations

Advantages

1. Dynamic sizing.

2. O(1) head insert / delete
3. No memory wastage
4. Easy memory management

Limitations

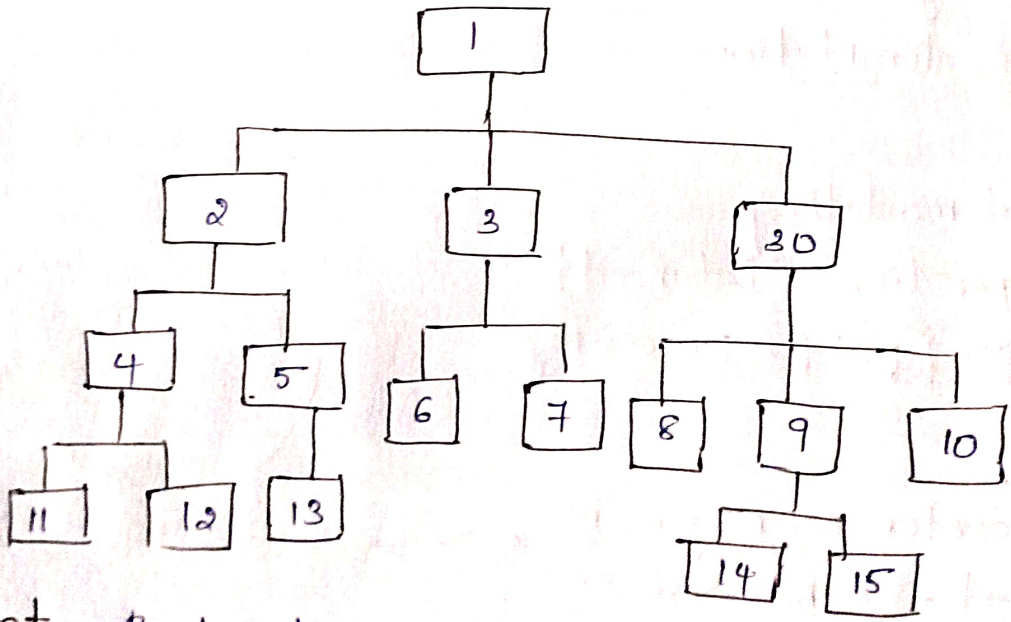
1. No random access O(n)
2. Extra pointer storage overhead
3. Cache inefficient
4. Complex implementation.

Non Linear Data Structures

(1) Trees

→ Tree is a non linear data structure which arranges the given data or elements in hierarchial order.
→ It identifies the parent-child relation ships among the items.
→ There is an item specially designated as root of tree.
→ All the remaining elements are organized into of tree.
→ All the remaining elements are organized into disjoint subsets called sub trees.

Consider an example



Root - Root is higher item of the tree.
from example "1" is the root of the tree.

leaf node :- leaf node refers for does not having further any nodes.

Here - 11, 12, 13, 14, 15 are the leafs.

parent :- which have the child

The parents are 1, 2, 3, 30, 4, 5, 6, 7, 8, 9, 10

Ancestors :- 1 is ancestor for 2, 3, 30

1, 2, 3, 30 are ancestors for 4, 5, 6, 7, 8, 9, 10

Descendants :- for example 9 descendants are 14, 15

Node structures

```
struct TreeNode
{ int data;
  struct TreeNode *left, *right;
};
```

Key characteristics

→ Hierachical structures

→ Root has indegree = 0

→ leaves have out degree = 0

→ Height = longest root to leaf path.

Advantages & limitations

Advantages.

- Hierarchial modeling
- O(log n) operations (balanced)
- prefix / infix / post-fix processing.

Limitations

- Complex pointer management
- Unbalanced → O(n) worst case
- Extra space for pointers.

Applications

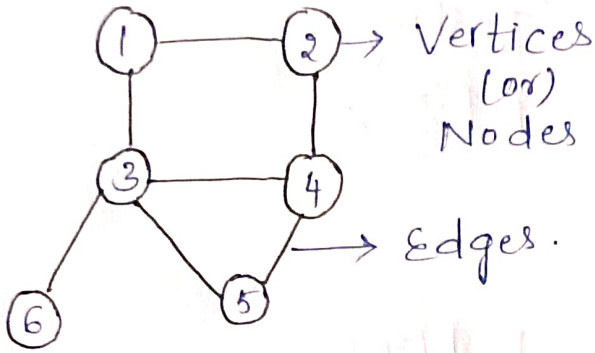
File systems, databases, expression trees, decision making.

(2) Graphs

→ Graph is a non linear data structure which arranges the given element in inter connected way.

→ The elements called nodes inter connected through edges.

Example:-



Key characteristics

- Arbitary Connections
- Directed / undirected
- Weighted / unweighted
- Cyclic / Acyclic.

Advantages & Limitations

Advantages

→ Model real world networks

→ Flexible Connectivity

→ Multiple path algorithms.

Limitations.

→ High space complexity

→ Complex algorithms.

→ Scalability issues.

ADT implementation Stratergies

(1) Stacks

1. Stack is a linear data structure
2. The elements are arranged in LIFO manner.

LIFO – Last In First Out

3. The operation on the stack only done on the top top=-1

Ex: Arrangement of plates.

→ If we want take a plate then we pick top plate which placed last

→ The process of insertion is 'pushing'.

→ The process of deletion is 'poping'.

Insertion (push)

top = -1 push (10)

top = 0 [10] push (20)

top = 1 [20] [10] push (30)

top = 2 [30] [20] [10]

Deletion (pop)

top = 2 [30] [20] [10] → pop → top 2 deleted → [20] [10] top = 1

top = 1 [20] [10] → pop → top 1 deleted → [10] top = 0

top = 0 [10] → deleted → top = -1

pop → top 0

(2) Queue

1. Queue is a linear data structure
2. The elements are arranged in FIFO manner.

FIFO - first in first out

(i) Where the elements are inserted is called rare.

(ii) Where the elements deleted is front.

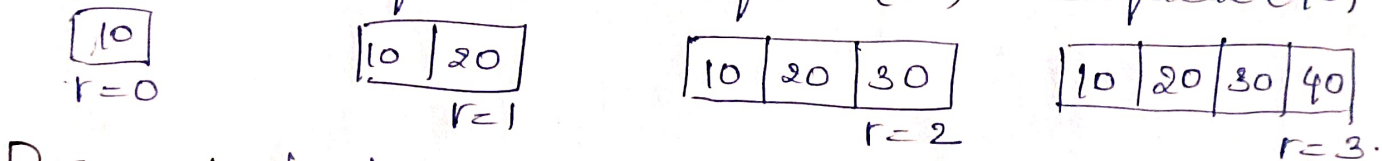
rare = -1, front = 0.

→ The deletion of elements in the Queue is called Enqueue.

→ The deletion of elements in the Queue is called Dequeue.

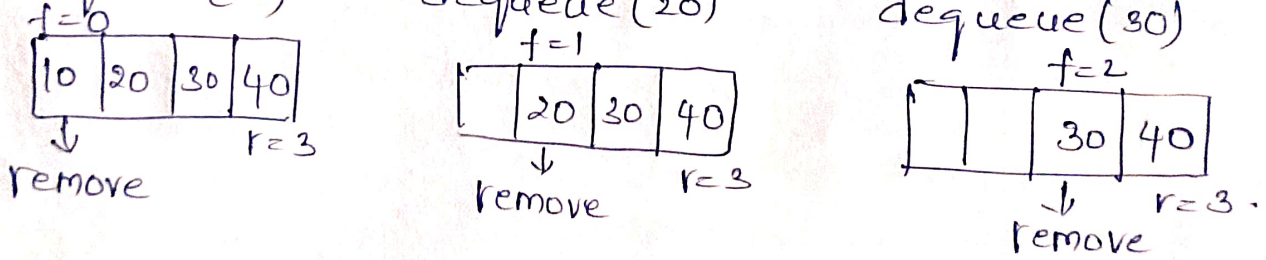
Enqueue:- rare = -1

Enqueue(10) Enqueue(20) Enqueue(30) Enqueue(40)



Dequeue:- front = 0.

dequeue(10) dequeue(20) dequeue(30)



Searching: Searching is a process of finding whether the given Particular key element is present in the collection of elements or not. There are two types of searching techniques. They are.

1. LINEAR SEARCH.
2. BINARY SEARCH.

1. LINEAR SEARCH:

Linear Search is a searching technique, also known as sequential search, is the simplest searching algorithm that sequentially checks each element of an array or list until the target is found or the end is reached.

The time complexity of linear Search is O(n) in the worst case and average case, because in the worst case, it may have to check all elements. In the best case, when the element is found at first position, the time complexity is O(1). It is suitable for small data sets but becomes inefficient for large data sets.

Algorithm process:

i) Initialize index i=0.
ii) compare array[i] with target key.
iii) If equal, return i (Position found).
iv) If N-1, increment i and repeat from step 2.
v) If end reached without match, return -1 (Not found)

Algorithm: Linear_Search (A, n, key)

```
Algorithm Linear_Search(A,n,key)
1. Begin.
2. i←0  // Initialize index to0.
3. while i<n Do // Continue while within bounds.
4. If A[i] = key then
5. Return i // Found: return Index.
6. End if
```

7. i ← i+1 // Move to next element
8. End while
9. Return -1 // Not found.
10. End

Explanation of Algorithm process:

1. Start the algorithm and initialize the index variable i=0. (begin from the first element of the array).
2. check the condition i<n to make sure the index is within the array size.
3. Compare the current ~~siz~~ element A[i] with the given key element.
4. If A[i] is equal to key, return i (the position where the element is found) and stop the algorithm.
5. If not equal, increase the index by 1 (i = i+1 / i++) and repeat the process.
6. If loop ends (all elements checked) and the key is not found, return -1 to indicate that element is not present in the array.

Detailed Process step-by-step:

Step 1: Set index counter i=0 (Start from first element).
Step 2: check if i<n (array bounds valid).
Step 3: compare A[i] with target key.
Step 4: If equal, return current index i. (Successful search).
Step 5: If not equal, increment by 1.
Step 6: Repeat from step 2, until match found or end reached.
Step 7: If loop ends (i≥n), return -1. (unsuccessful Search).

Solved Example : 1 [ELEMENT FOUND]

Array A: {12, 45, 23, 51, 19, 8, 96} (n = 7)

key = 23

Step	i	A[i]	A[i] = key?	Action
1.	0	12	12 ≠ 23	i = 1
2.	1	45	45 ≠ 23	i = 2
3.	2	23	23 = 23	Return 2.

Explanation: Step 1:

Set index i = 0

Check A[0] = 12

Compare 12 with 23 [12 ≠ 23] → not equal.

So, move to next index (i = 1).

Step 2:

Now i = 1

Check A[1] = 45

Compare 45 with 23 [45 ≠ 23] → not equal.

So, move to next index (i = 2).

Step 3:

Now i = 2

Check A[2] = 23

Compare 23 with 23 [23 = 23] → equal.

Element found at index 2.

So, the algorithm returns 2 and stop the process.

Result : Element 23 found at index 2 after 3 comparisions.

Solved Example : 2 [ELEMENT NOT FOUND].

Array A: {10, 20, 30, 40, 50} (n = 5).

key = 25

Step	i	A[i]	A[i]=key?	Action
1.	0	10	10≠25	i=1
2.	1	20	20≠25	i=2
3.	2	30	30≠25	i=3
4.	3	40	40≠25	i=4
5.	4	50	50≠25	i=5
6.	5	-	i≥n	Return -1.

Explanation:

Step 1:

Set index i=0.

Check A[0]=10.

Compare 10 with 25 [10≠25] → not equal.

So, move to next index (i=1).

Step 2:

Set index i=1

check A[1]=20.

Compare 20 with 25 [20≠25] →not equal.

So, move to next index (i=2).

Step 3:

Now index i=2

Check A[2]=30.

Compare 30 with 25 [30≠25]→not equal.

So, move to next index (i=3).

Step 4:

Set index i=3.

Check A[3]=40.

Compare 40 with 25 [40≠25]→not equal.

So, move to next index (i=4).

Step 5:

Now index i=4.

Check A[4]=50.

Compare 50 with 25 [50≠25] →not equal.

So, move to next index (i=5).

Step 6:

Now i=5

Since i≥n (5≥5), the loop stops.

Element is not found.

Result: Element 25 not found after 5 comparisions.

Solved Example 3: [BEST CASE]

Array A: {15, 22, 37, 48} (n=4).

key = 15.

Step	i	A[i]	A[i]=key?	Action
1	0	15	15=15	Return 0.

Explanation:

Step 1:

Set index i=0

check A[0]=15

compare 15 with 15 [15=15] → Equal.

Element found.

Result: Element 15 found at index 0 after 1 comparision.

Complexity Analysis:

Best Case: O(1) → Element at position 0.

Average Case: O(N/2) ≈ O(N)

Worst Case: O(N) → Element at end or absent.

Space complexity: O(1).

Advantages and Disadvantages:

Advantages:

1. Simple to implement.
2. Works on unsorted arrays.
3. No preprocessing required.

Disadvantages:

1. Very slow for large arrays.
2. Always scan sequentially.

Applications: Small arrays, nearly sorted data, linked lists.

Recursive Linear Search:

It converts iterative linear search into recursive form by using function calls instead of loops. In this method, the function checks one element at a time and then calls itself to check the next element in the array.

The process continues until the element is found or the base case is reached. If the element matches the key, it returns the index. If the end of the array is reached without a match, the function returns -1 indicating the element is not found [unsuccessful Search].

Algorithm:

Recursive-LinearSearch(arr, n, key, index)

Algorithm Recursive-Linear Search (arr, n, key, index)

1. Begin.
2. If index ≥ n then // Base case 1: End reached.
3. Return -1
4. End if
5. If arr[index] = key then // Base case 2: Element found.
6. Return index.
7. End if
8. Return Recursive-Linear Search (arr, n, key, index+1)
   // Recursive case.
9. End.

Detailed Process step-by-step:

1. Base case 1 (index ≥ n): Array bounds exceeded → return -1 (not found).

2. Base case 2 (arr[index] = key): Match found → returns current index.

3. Recursive case: No match → call same function with index + 1

4. Result propagates back through call stack to main caller.

In the algorithm:

arr → It is a array that contains the list of elements.

n → size (number of elements)

key → Element we want to search.

index → current position (usually starts from 0).

index ≥ n → Base case 1

Base case → stopping condition.

arr[index] = key → Base case 2.

Return index → Returns position 'i' [Successful Search].

Return -1 → Unsuccessful Search [key is not found].

Return Recursive_Linear Search (arr, n, key, index + 1) → Recursive call (function calls itself).

Solved Example: [ELEMENT FOUND]

Array A: {12, 45, 23, 51, 19, 8, 96} (n = 7)

key = 23.

Initial call: Recursive_Linear Search (A, 7, 23, 0).

Call level	index	arr[index]	step Executed	Action
call 1	0	12	Step 2: 0<7 ✓ Step 6: 12≠23	Recursive call (index=1)
call 2	1	45	Step 2: 1<7 ✓ Step 6: 45≠23	Recursive call (index=2)
call 3	2	23	Step 2: 2<7 ✓ Step 6: 23=23	Return 2
Return to call 2	2	-	Recieves 2	Return 2
Return to call 1	2	-	Recieves 2	Return 2

Explanation:

1. Call 1 (index=0)

   check : index ≥ 7 ? → 0 ≥ 7 → NO.

   Compare A[0]=12 with 23 → Not equal.

   Action: Make recursive call with index=1.

   The function starts at index 0 and compares 12 with 23; since they are not equal it makes a recursive call with index 1.

2. Call 2: (index=1)

   check: index ≥ 7 ? → 1 ≥ 7 → NO

   Compare A[1]=45 with 23 → Not equal.

   Action: Make recursive call with index=2.

   The function checks index 2 and compares 23 with 45; since they are not equal it makes a recursive call with index 2.1.

3. Call 3: (index=2)

   Check: index ≥ 7 ? → 2 ≥ 7 → NO

   Compare A[2]=23 with 23 → Equal.

Element found

~~Reut~~ Return 2. [index]

The function checks index 2 and compares 23 with 23; Since they are equal the element is found and index 2 is returned.

Returning Back:

call 2 receives 2 → returns 2.

Call 2 receives the value 2 from call 3 and returns 2 to the previous call.

Call 1 receives 2 → returns 2.

call 1 receives the value 2 and return 2 as the final answer.

Result: Element found at index 2 → Successful Search.

Solved Example 2: [ELEMENT NOT FOUND]

Array A: {10, 20, 30, 40, 50} 60, 80

key = 25 (n=5)

Call level	index	arr[index]	Step executed	Action
call 1	0	10	10 ≠ 25	call index = 1
call 2	1	20	20 ≠ 25	call index = 2
call 3	2	30	30 ≠ 25	call index = 3
call 4	3	40	40 ≠ 25	call index = 4
call 5	4	50	50 ≠ 25	call index = 5
call 6	5	–	Step 2: 5 ≥ 5	Return -1.

Explanation:

1. Call 1: (index = 0)

check: index ≥ n → 0 ≥ 5 → NO.

Compare A[0] = 10 with 25 → Not Equal.

Recursive call → index = 1.

The function starts at index 0 and checks 10; since it is not 25, it moves to next index.

2. Call 2: (index = 1)

check: index ≥ n → 0 ≱ 1 ≥ 5 → NO.

Compare A[1] = 20 with 25 → Not Equal.

Recursive call → index = 2.

The function starts at index 1 and compares 20 with 25, since they are not equal, it moves forward.

3. Call 3: (index = 2)

check: index ≥ n → 2 ≥ 5 → NO.

Compare A[2] = 30 with 25 → Not Equal.

Recursive call → index = 3

The function starts at index 2 and compares 30 with 25, since they are not equal, it moves forward.

4. Call 4: (index = 3)

check: index ≥ n → 3 ≥ 5 → NO.

Compare A[3] = 40 with 25 → Not Equal.

Recursive call → index = 4.

The function starts at index 3 and compares 40 with 25, since they are not equal, it moves forward.

5. Call 5 : (index = 4)

Check : index ≥ n → 4 ≥ 5 → NO

Compare A[4] = 50 with 25 → Not equal

Recursive call → index = 5.

The function checks index 4 and compares 50 with 25, since they are not equal it moves forward.

6. Call 6 : (index = 5)

Check : index ≥ n → 5 ≥ 5 → YES (Base Case)

Compare : End of array reached. So, Return -1

The index becomes 5 which is equal to n, so the base case is satisfied and the function returns -1, indicating the element is not found. [unsuccessful Search].

Result : 25 not found after 6 recursive calls. So, return -1.

Solved Example 3 : [BEST CASE (FIRST POSITION)].

Array A : {15, 22, 37, 48} (n = 4)

key = 15.

Call level	index	arr[index]	step executed	Action
Call 1	0	15	step 6 : 15 = 15	Return 0

Explanation :

1. Call 1 : (index = 0)

Check : index ≥ n → 0 ≥ 5 → NO

compare A[0] = 15 with 15 → Equal.

Return 0 [index].

Result : 15 is found at index 0. So, return 2. [Element found].

Complexity Analysis

Time Complexity : O(N) → Worst Case.

Space Complexity : O(N) due to recursive stack.

Best Case : O(1) → first element match.

Average Case : O(N/2) ≈ O(N).

Comparsion : Recursive Vs Iterative

Aspect	Iterative	Recursive
Time	O(N)	O(N)
Space	O(1)	O(N)
Code	Loop	Function calls.
Stack	No risk.	Stack Overflow risk.

2. BINARY SEARCH:

Iterative method

Binary Search works only on a sorted array. First, it finds the middle element of the array and compares it with the key element. If both are equal, the element is found. If the key is smaller than the middle element, the Search continues in the left half of the array. If the key is larger, the search continues in the right half. This Process of dividing the array into two ~~e~~halves [This Process of dividing the array int] continue until the element is found or there are no elements left to Search.

Time complexity of Binary search is O(log n) because in each step the search space is reduced to half. This makes binary search much faster than linear search for large datasets. In the best case, when the element is found in the first comparision, the time complexity is O(1).

Algorithm - Step by step:

Algorithm for BinarySearch.

Find Target in Sorted array.

1. Initialize low=0, high=n-1.
2. while low<=high
3. calculate mid = low+(high-low)/2
4. If array[mid]==target, return mid.
5. else if array[mid]>target, set high=mid-1
6. else set low=mid+1.
7. If loop ends, return -1 (not found).

PSEUDO CODE:

```
int binarysearch(int array[],int n,int target)
{
Step 1: Initialize bounds.
int low=0,high=n-1;
Step 2: while search space exists
        while(low<=high).
{
 Step 3: calculate mid
 int mid = low+(high-low)/2;
 Step 4: Target found
 if(array[mid]==target)
 return mid;
 Step 5: Search left half
 else if(array[mid]>target)
```

```
        high=mid-1;
        Step 6: Search right half
        else
        low=mid+1;
    }
        Step 7: Not found
        return -1;
}
```

Explanation:

1. The main function call:

- Low = The very first index (0).
- High = The very last index (n-1).
- key = The number you are searching for.

2. The execution steps (The logic):

Every time, the function runs it follows these 3 steps:

- Step 1: check the base case ("The exit Rule").

If low > high, it means search area has vanished and the number isn't there.

Result: Return -1 (Not found).

- Step 2: Calculate the mid Value.

mid = (low+high)/2 (or) low+(high-low)/2.

3. Three possible Decisions:

- If A[mid] == key: You found it! Return index.
- If A[mid] < key: The middle number is small. The key must be on right.

Action: Recurse right

(New range: high = mid+1).

- If A[mid] > key: The middle number is big. The key must be on left.

Action: Recurse left.

(New range: low = mid - 1).

Examples [Recursive]:

Sorted array {10, 20, 30, 40, 50, 60, 70}

(n = 7)

key = 50

Initial Range: Index 0 to 6.

Call Tree Visualization:

Call 1: [0, 6]

Mid = (low + high)/2 = (0 + 6)/2 = 6/2 = 3.

Mid = 3.

a[3] = 40.

Compare with key 50 → 50 > 40.

So, the key must be in right half. Now low = mid + 1

= 3 + 1

low = 4

{10, 20, 30, 40, 50, 60, 70}

low = 0 (Mid, key) high = 6.

a[mid] = a[3]

50 > 40.

{10, 20, 30} Discard left side

low = 4 {50, 60, 70} high = 6. Take right half

Call 2: [4, 6]

Mid = (low + high)/2 = (4 + 6)/2 = 10/2 = 5.

Mid = 5

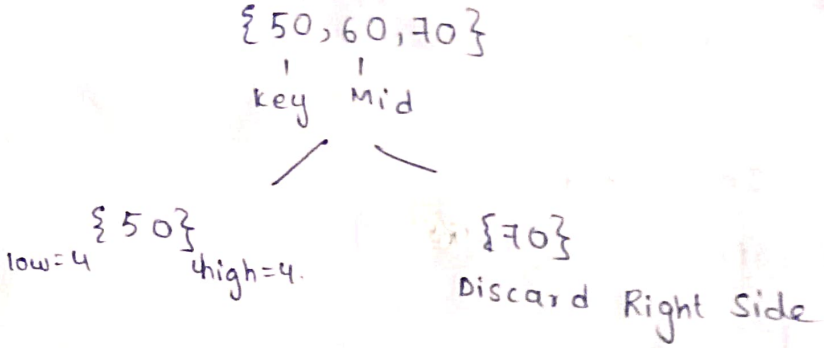
a[5]=60

Compare with key 50 → 50<60

So, the key must be in left half. Now

high=mid-1=5-1=4

high=4.



Call 3: [4,4]

Mid=(low+high)/2 = (4+4)/2 = 8/2 = 4.

Mid=4.

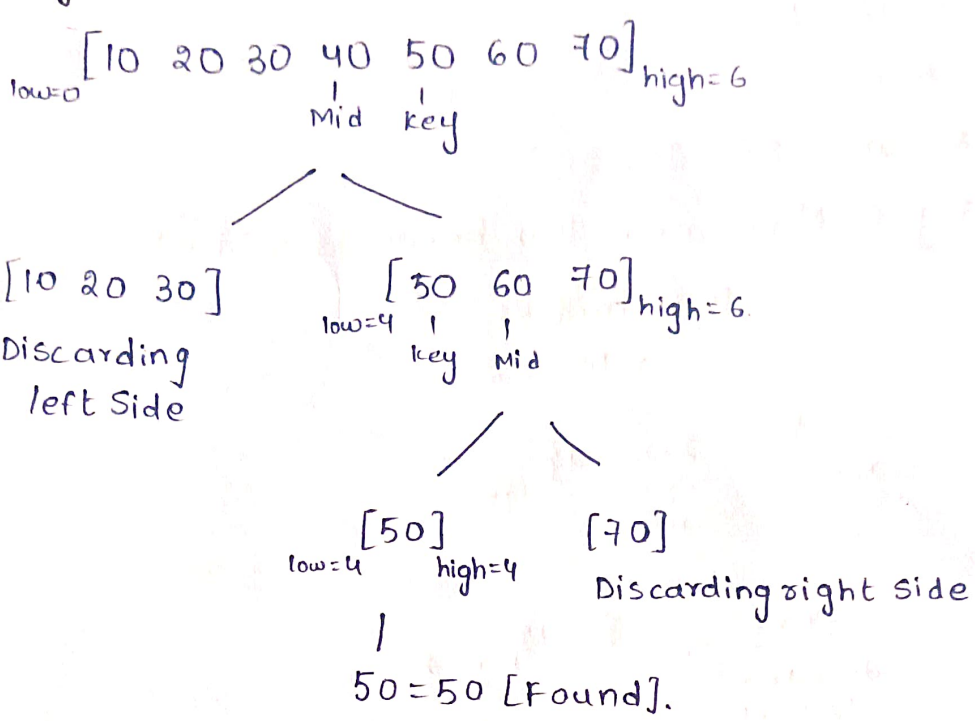
a[4]=50.

Compare with key 50 → 50=50.

Returns index i.e returns 4.

Call stack	low	high	Range	mid	A[mid]	Decision	Result
Call 1	0	9	[0-6]	3	40	40<50	call 2
call 2	4	6	[4-6]	5	60	60>50	call 3
call 3	4	4	[4-4]	4	50	50=50	Return 5
Return	5	-	-	-	-	-	final: 5

final diagram flow:



Result: 50 is found at index 4. So, returns "4."

Example for unsuccessful Search:

A = {10, 20, 30, 40, 50, 60, 70} (n = 7)

Key = 25.

Call 1:

Initially low = 0, high = n-1 = 6

Mid = (low+high)/2 = (0+6)/2 = 3.

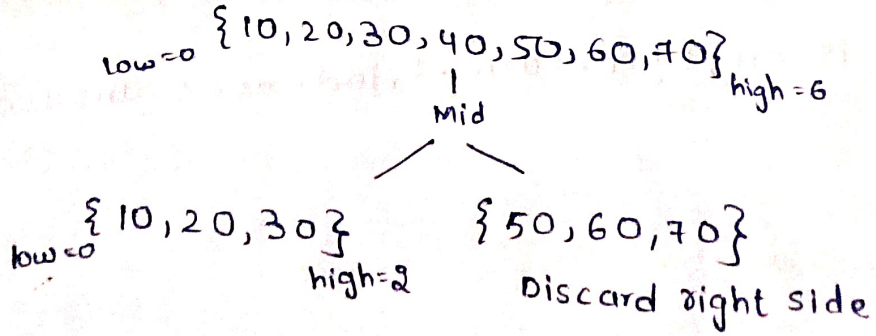
Value at A[3] = 40

Compare 40 with 25 [40 > 25].

Recurse left. Now new high becomes high = mid - 1

= 3 - 1

= 2



Call 2: low=0, high=2

Mid = (0+2)/2 = 1

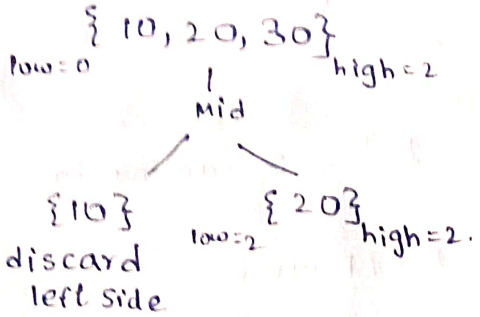
Value at A[1] = 20.

Compare 20 with 25 (20<25)

Recurse right. Now new low becomes mid+1

low = mid + 1

= 1+1 = 2.



Call 3: low=2, high=2

Mid = (2+2)/2 = 4/2 = 2.

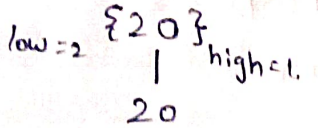
Value at A[2] = 30.

Compare 30 with 25 (30>25)

Recurse left. Now new high becomes mid-1

high = mid - 1

= 2-1 = 1.



Call 4:

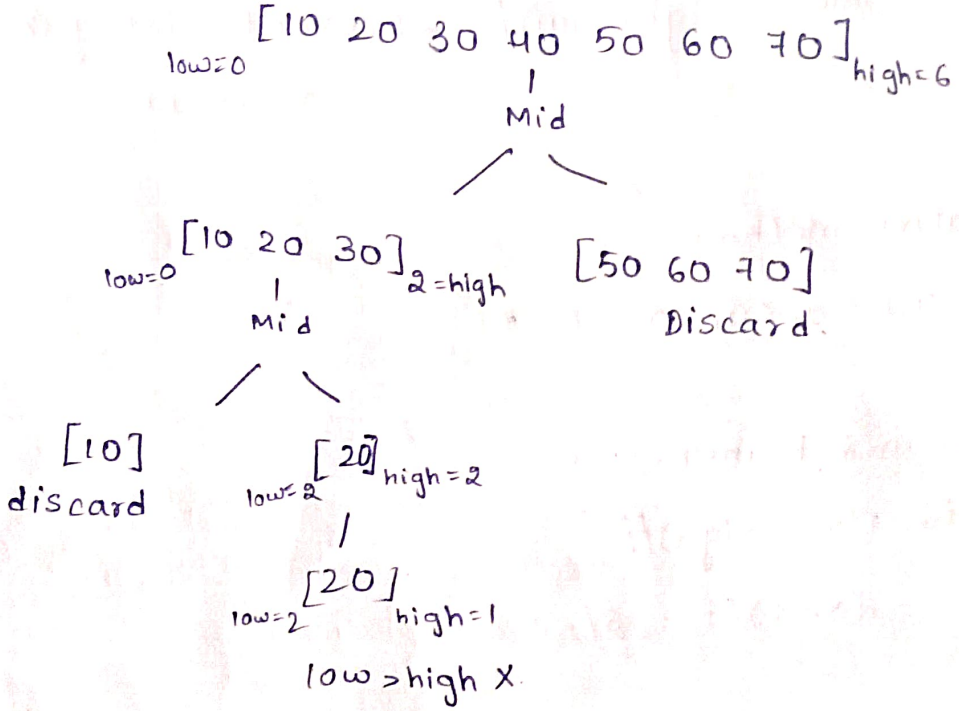
low = 2, high = 1

Here low > high (2 > 1)

So the range is empty. Return -1, indicates unsuccessful search.

Call stack	low	high	Range	mid	A[mid]	Decision	Result
Call 1	0	6	[0-6]	3	40	40>25	call 2
call 2	0	2	[0-2]	1	20	20<25	call 3
Call 3	2	2	[2-2]	2	30	30>25	call 4
Call 4	2	1	-	-	-	-	-

Final diagram flow:



Unsuccessful Search. Returns -1.

Time and space complexity:

Time complexity - O(log n) - halves search space each step.

Space complexity: O(1) only uses constant extra variables.

Key points:

- Uses loop, O(1) space – Iterative
- Uses call stacks, O(log N) Space – Recursive
- O(log N) time, requires sorted array – Both
- Mid calculation : (low + high) / 2

  (or)

  low + (high - low) / 2. – prevents integer overflow.

Examples [Iterative method :

1. A = { 10, 20, 30, 40, 50, 60, 70 } (n = 7)

   key = 50.

   Initial → low = 0 and high = n - 1 = 6.

   calculate mid = (low + high) / 2

   = (0 + 6) / 2 = 6 / 2 = 3.

Iteration 1:

Mid = 3.

A[3] = 40.

Since 40 < 50, the key must be in the right half.

Set low = mid + 1

= 3 + 1

= 4.

New search Space = $\{50, 60, 70\}_{h=6}^{l=4}$

Iteration 2: Now low = 4, high = 6

Mid = (4 + 6) / 2 = 10 / 2 = 5.

A[5] = 60

Since 60 > 50, the key must be in left half

Set high = mid - 1

= 5 - 1

= 4

New Search Space = {50}$_{l=4}$ $_{h=4}$

Iteration 3 Now low = 4, high = 4

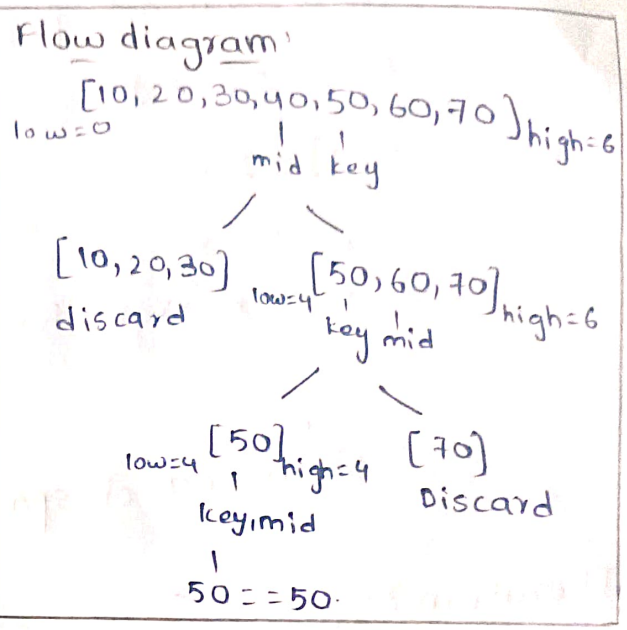
Mid = (4+4)/2 = 8/2 = 4

A[4] = 50

50 == 50.

Since 50 is found at index 5, it returns 5.

Flow diagram:



Iteration	low	high	mid	A[mid]	Search space	Decision	Action
Initially	0	6	-	-	[0 ---- 6]	-	-
1	0	6	3	40	10, 20, 30, 40, 50, 60, 70	40 < 50	low = 4
2	4	6	5	60	50, 60, 70	60 > 50	high = 4
3	4	4	4	50	50	50 = 50	Return 5

2. Element not found example:

A = {10, 20, 30, 40, 50, 60, 70} (n = 7)

key = 25

Iteration 1: Initially low = 0, high = n - 1 = 6

A[3] = 40

25 < 40

So search in left half. So, ~~low~~ high = mid - 1

= 3 - 1

= 2

New search space = {10, 20, 30}
low = 0, high = 2

Iteration 2:

Now low = 0, high = 2

Mid = 2/2 = 1.

A[1] = 20

25 > 20.

Search in right half. So, low = mid + 1

= 1 + 1

= 2.

New search space {30}

low = 2 high = 2.

Iteration 3:

Now low = 2, high = 2

mid = (2+2)/2 = 4/2 = 2.

A[2] = 30

25 < 30

Search range becomes empty

high = mid - 1

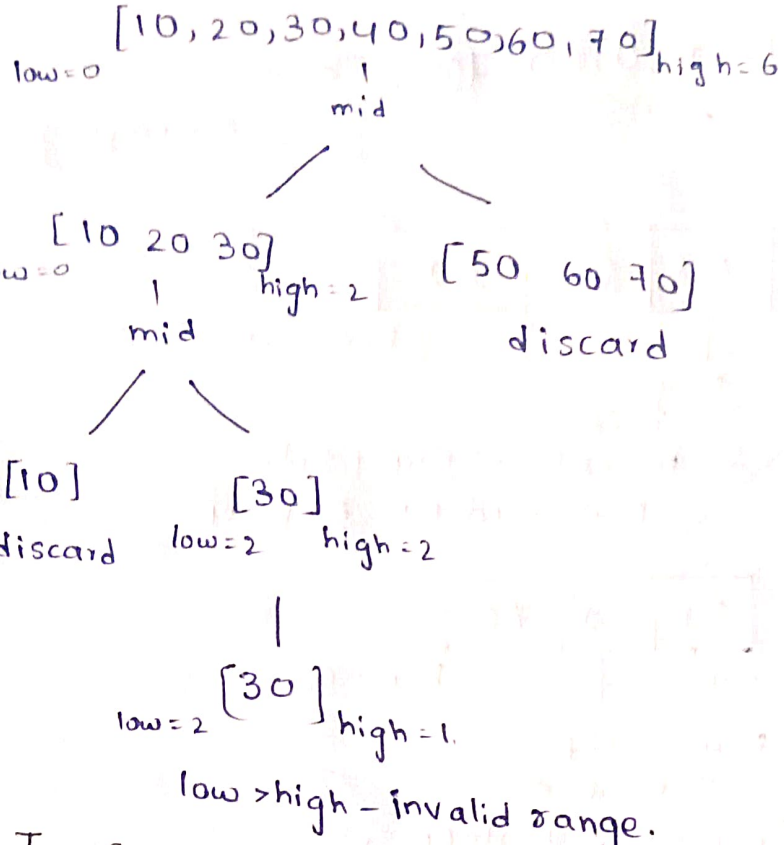
= 2 - 1 = 1.

low = 2, high = 1 [low > high]

Result: Unsuccessful Search → Returns -1.

Iteration	comparision	Meaning	New range
1	40 > 25	key is smaller than 40	Search left
2	20 < 25	key is greater than 20.	search right
3	30 > 25	key is smaller than 30	Search left → empty
End	low > high	Search range invalid	Return -1.

Flow diagram



low > high - invalid range.

In Recursive

Recursive version divides problem into smaller sub problems using function calls.

Main function call:

Initial: Recursive-Binary-search(array, low=0, high=n-1, key)

1. Check base case: If low > high → Return -1.
2. Compute mid = (low+high)/2.
3. Three possible decisions:

A. A[mid] == key → Return mid (FOUND)

B. A[mid] < key → Recurse right

Recursive-Binary-search (array, mid+1, high, key)

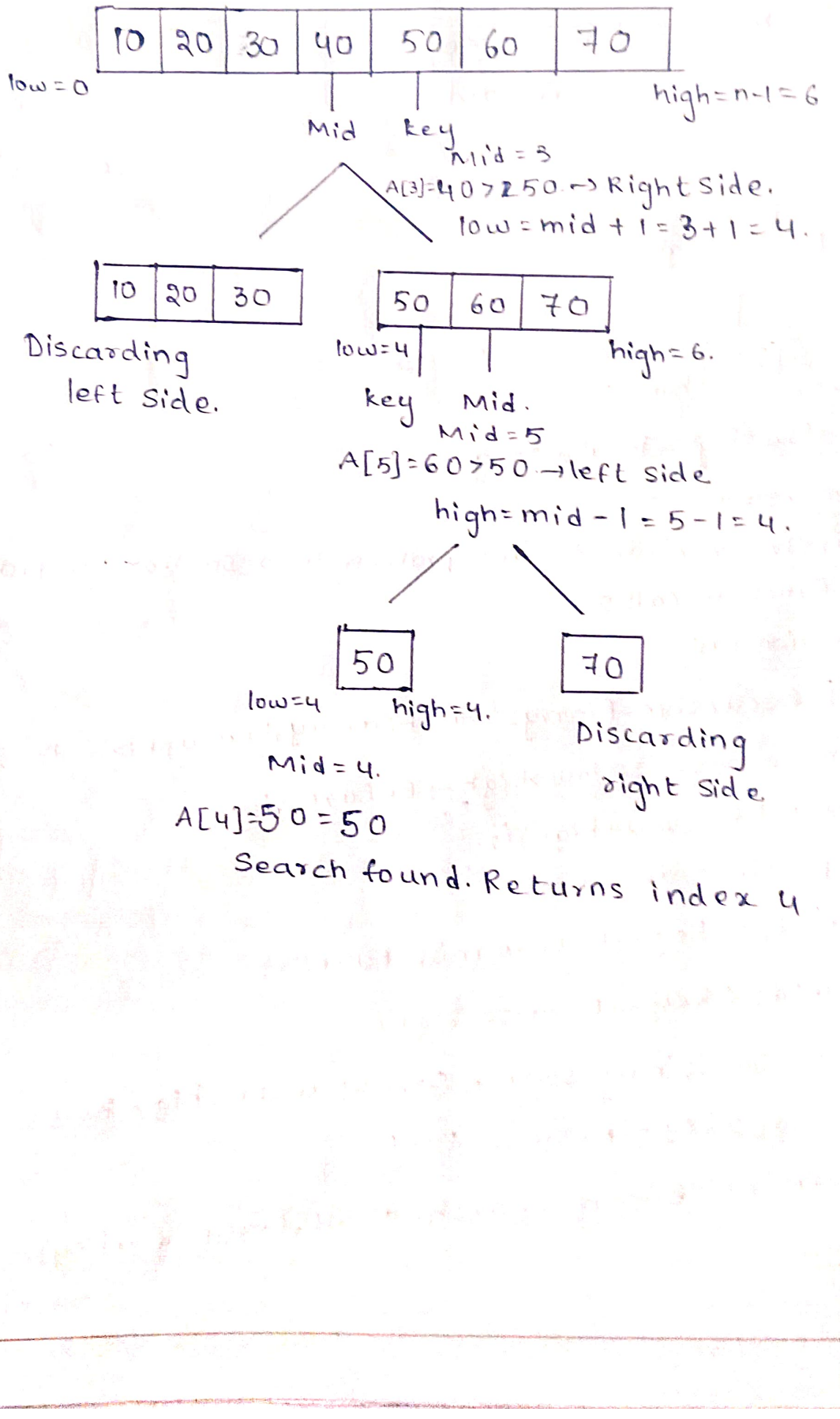
C. A[mid] > key → Recurse left

Recursive- Binary-search(array, low, mid-1, key).

Flow diagrams

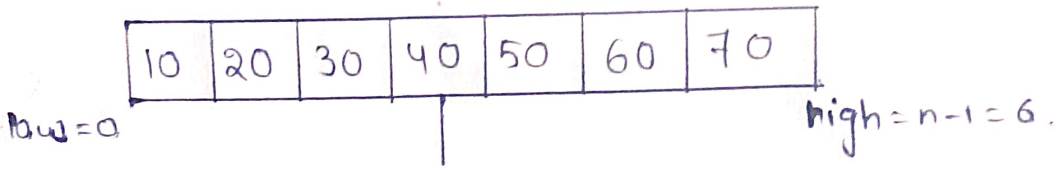
* For example A = {10, 20, 30, 40, 50, 60, 70} [Element found example]
(n=7)

key = 50.



For example A = {10, 20, 30, 40, 50, 60, 70} [Element not found example].

Key = 25 (n = 7).

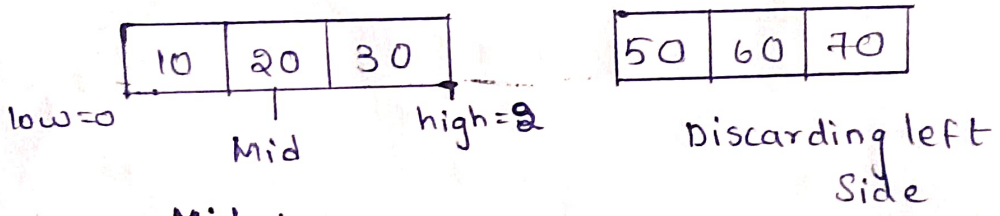


Mid = 3

A[4] = 40 > 25 → left side.

high = mid - 1

= 3 - 1 = 2



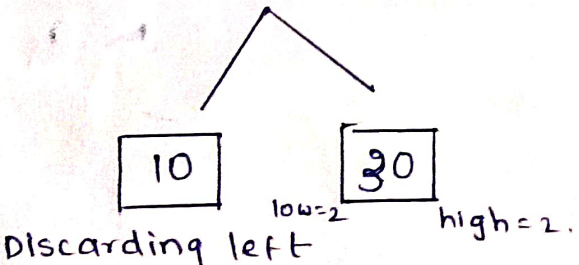
Mid = 1.

A[1] = 20 < 25 → right side

low = mid + 1

= 1 + 1

= 2.



Mid = 2.

A[2] = 30 > 25 → left side [No elements].

high = mid - 1 = 2 - 1 = 1.

low = 2, high = 1

low > high ✗

Returns -1 indicating unsuccessful Search.

SORTINGS

The Process of arranging given elements either in ascending order (or) desending order.

They are 3 types:

i) Bubble Sort

ii) Insertion Sort

iii) Selection Sort.

Bubble Sort:-

It will Compare the sucessive elements and swap if necessary through all iterations.

Bubble Sort operates in multiple passes over the array. In the first it compares every pair of adjacent elements from the start to the end -1 position, if the left element greater than the right, it swaps them, Pushing the largest element to the very end. The second pass repeats this but stops one position short (since the last element is now sorted), bubbling the second-largest to its place and soon -each pass reduced the unsorted region by one. Optimization skips remaining passes run for n elements. Swaps are $O(n^2)$ in worst reverse cases, but stable (preserves relative order of equals) and in-place (no extra space beyond variables).

Algorithm:-

```
Procedure bubble sort (A: list of sortable items)
  n = length (A)  //n -> Size (number of elements)
 repeat
Swapped = false //A -> It is an array the contains the list of elements
for i = 0 to n-1 do
if (A[i] > A[i+1] then // i -> iteration from 0 to n-1
  Swap (A[i], A[i+1])
    Swapped = true
  endif
 endfor
n = n-1 // largest element now at end until not swaped.
end procedure
```

Algorithm explanation:-

1. Start from the first elements
2. Compare first two elements
3. If the first element is greater than the second, swap them.

4. Move to next pair and repeat the comparison.

5. Continue until the end of the list.

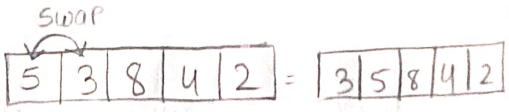
6. Repeat the whole process until no swaps are needed.

Step by step trace:

Trace bubble sort an array [5,3,8,4,2] (ascending order). marks sorted portion after each pass.

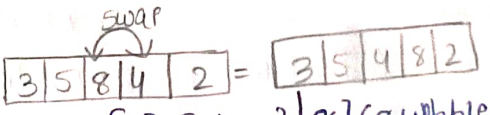
Intial: [5,3,8,4,2]

Pass1 (i=1 to 4):



i=1: 5>3? yes → swap → [3,5,8,4,2]

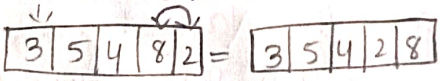
i=2: 5<8? no → [3,5,8,4,2] → 3 5 8 4 2



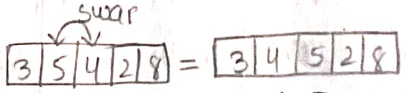
i=3: 8>4? yes → swap → [3,5,4,8,2] →

i=4: 8>2? yes → swap → [3,5,4,2,8] | End Pass1: [3,5,4,2|8] (8 bubbled)

Pass2 (i=1 to 3):



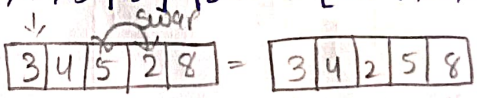
i=1: 3<5? no → [3,5,4,2|8] → 3 5 4 2 8



i=2: 5>4? yes → swap → [3,4,5,2|8] →

i=3: 5>2? yes → swap → [3,4,2,5|8] | End: [3,4,2,5|8] (5 bubbled)

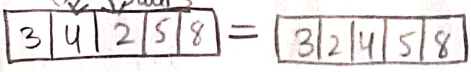
Pass3 (i=1 to 2):



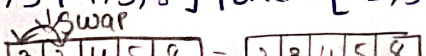
i=1: 3<4? no → [3,4,2,5|8]

i=2: 4>2? yes → swap → [3,2,4,5|8] | End: [3,2,4|5,8] (4 bubbled)

Pass4 (i=1 to 1):



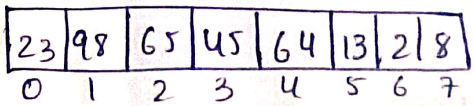
i=1: 3>2? yes → swap → [2,3|4,5,8] | End: [2,3|4,5,8] (3 bubbled)



Pass5: No swaps needed (already sorted). Final: [2,3,4,5,8].

Explanation:-

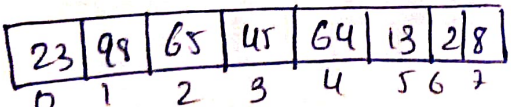
Iteration 1:- we can consider an array



Step1:- a(0) = 23 a(1) = 98

23 > 98 (False)

Swap is not required



Step2: a[1] = 98 a[2] = 65

98 > 65 (True)

Swap is required

Now swap 98 and 65 and write an array

23	65	98	45	64	13	2	8
0	1	2	3	4	5	6	7

Step3:-

a[2]=98 a[3]=45

98>45(TRUE)

Swap is required

Now swap 98 and 45 and rewrite the array

23	65	45	98	64	13	2	8
0	1	2	3	4	5	6	7

Step4:-

a[3]=98 a[4]=64

98>64 (TRUE)

Swap is required

Now swap 98 and 64 and rewrite the array:

23	65	45	64	98	13	2	8
0	1	2	3	4	5	6	7

Step5:-

a[4]=98 a[5]=13

98>13(TRUE)

Swap is required

Now Swap 98 and 13 and rewrite the array:

23	65	45	64	13	98	2	8
0	1	2	3	4	5	6	7

Step6:-

a[5]=98 a[6]=2

98>2(TRUE)

Swap is required.

Now swap 98 and 2 and rewrite the array:

23	65	45	64	13	2	98	8
0	1	2	3	4	5	6	7

Step7:-

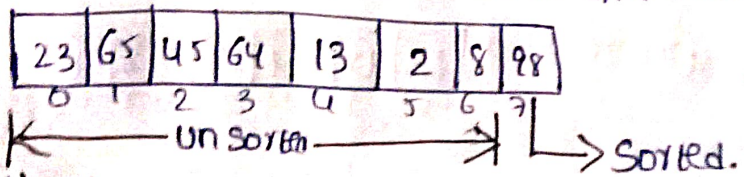
a[6]=98 a[7]=8

98>8(TRUE)

Swap is required.

Now swap 98 and 8 and rewrite the array:

23	65	45	64	13	2	8	98
0	1	2	3	4	5	6	7



Iteration2:-

Step1:

a[0]=23 a[1]=65

23>65(False)
No swap is required

23	65	45	64	13	2	8	98
0	1	2	3	4	5	6	7

Step2:-
a[1]=65 a[2]=45
65>45 (TRUE)
swap is required
Now swap 65 and 45 and rewrite the array:

23	45	65	64	13	2	8	98
0	1	2	3	4	5	6	7

Step3:
a[2]=65 a[3]=64
65>64(TRUE), Swap is required
Now swap 65 and 64 and re write the array:

23	45	64	65	13	2	8	98
0	1	2	3	4	5	6	7

Step4:-
a[3]=65 a[4]=13
65>13(TRUE), swap is required
Now swap 65 and 13 and re write the array:

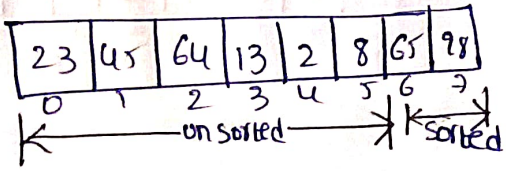
23	45	64	13	65	2	8	98
0	1	2	3	4	5	6	7

Step5:-
a[4]=65 a[5]=2
65>2 (TRUE), Swap is required
Now swap 65 and 2 and rewrite the array:

23	45	64	13	2	65	8	98
0	1	2	3	4	5	6	7

Step6:
a[5]=65 a[6]=8
65>8 (TRUE), swap is required
Now swap 65 and 8 and re write the array:

23	45	64	13	2	8	65	98
0	1	2	3	4	5	6	7



Iteration 3:-
Step1:-
a[0]=23 a[1]=45
23>45(F), swap is not required

23	45	64	13	2	8	65	98
0	1	2	3	4	5	6	7

Step2:-
a[1]=45 a[2]=64
45>64(TRUE), swap is not required
Now swap u

23	45	64	13	2	8	65	98
0	1	2	3	4	5	6	7

Step3:
a[2] = 64 a[3] = 13
64 > 13 (True), swap is required
Now swap 64 and 13 and rewrite the array:

23	45	13	64	2	8	65	98
0	1	2	3	4	5	6	7

Step4:
a[3] = 64 a[4] = 2
64 > 2 (True), swap is required
Now swap 64 and 2 and rewrite the array:

23	45	3	2	64	8	65	98
0	1	2	3	4	5	6	7

Step5:
a[4] = 64 a[5] = 8
64 > 8 (True), swap is required
Now swap 64 and 8 and rewrite the array

23	45	13	2	8	64	65	98
0	1	2	3	4	5	6	7

← unsorted → | sorted

Iteration 4:
Step1:
a[0] = 23 a[1] = 45
23 > 45 (F) No swap is required

23	45	13	2	8	64	65	98
0	1	2	3	4	5	6	7

Step2:
a[1] = 45 a[2] = 13
45 > 13 (True), swap is required
Now swap 45 and 13 and rewrite the array:

23	13	45	2	8	64	65	98
0	1	2	3	4	5	6	7

Step3:
a[2] = 45 a[3] = 2
45 > 2 (T)
swap is required, Now swap 45 and 2 & rewrite the array

23	13	2	45	8	64	65	98
0	1	2	3	4	5	6	7

Step4:
a[3] = 45 a[4] = 8
45 > 8 (True), swap is required
Now swap 45 and 8 and rewrite the array:

23	13	2	8	45	64	65	98
0	1	2	3	4	5	6	7

← unsorted | sorted →

Iteration 5:
Step1:
a[0] = 23 a[1] = 13

23 > 13 (TRUE), swap is required

Now swap 23 and 13 and rewrite the array:

13	23	2	8	45	64	65	98
0	1	2	3	4	5	6	7

Step2:-

a[1] = 23 a[2] = 2

23 > 2 (T), swap is required

Now swap 23 and 2 and rewrite the array:

13	2	23	8	45	64	65	98
0	1	2	3	4	5	6	7

Step3:-

a[2] = 23 a[3] = 8

23 > 8 (TRUE), swap is required

Now swap 23 and 8 & rewrite the array:

13	2	8	23	45	64	65	98
0	1	2	3	4	5	6	7

Unsorted → Sorted →

Iteration 6:-

Step1:

a[0] = 13 a[1] = 2

13 > 2 (TRUE)

Swap is required, Now swap 13 and 2

2	13	8	23	45	64	65	98
0	1	2	3	4	5	6	7

Step2:-

a[1] = 13 a[2] = 8

13 > 8 (TRUE), swap is required

Now swap 13 and 8 and rewrite the array

2	8	13	23	45	64	65	98
0	1	2	3	4	5	6	7

Time Complexity:-

Best Case O(n): Already sorted array requires just one pass with no swaps, early termination via flag checks all n-1 adjacent pairs once.

Case	Comparison	Swaps
Best	n-1	0
Average	$\sim n^2/2$	$\sim n^2/2$
Worst	$n(n-1)/2$	$n(n-1)/2$

Space Complexity O(1):-

In place algorithm (only swap variables needed).

ii) Insertion Sort:-

Insertion Sort builds a sorted Prefix incrementally by taking each unsorted element and inserting in Correct Position within the already-sorted left Position. Foreach index i from 1 to n-1, it Store A[i] as the key then shifts all elements in the sorted Prefix (0 to i-1) that are greater than key one position right ward until finding the correct insertion point (or) reaching index 0. This mimics organizing cards in hand - each new card finds its place through minimal shifts.

Algorithm:-

```
Void insertion Sort(Comparable a[]
{
  for (int i=1; i<n; i++) //i -> iteration from 1 to n
  {
    comparable tmp = a[p]
    int j;
  for (j=i; j>0 && tmp.Compare to (a[j-1])<0; j--)
      a[j] = a[j-1]; //j ->
      a[j] = tmp;
    }
}
```

Algorithm explanation:-

1. Assume the first element is already sorted.
2. Pick the next element (the key) from unsorted section.
3. Compare the key with elements in the sorted portion, moving right to left
4. Shift larger elements to the right to make space.
5. Insert the key into its correct Position
6. Repeat until the entire array is sorted.

Step-by-step trace (Array: [64, 25, 12, 22, 11] → ascending)

Initial: [64, 25, 12, 22, 11]

i = 1 (P = 1): key = 25; 64 > 25 → Shift → [25, 64, 12, 22, 11]

i = 2 (P = 2): key = 12; 64 > 12 → Shift → [25, 12, 64, 22, 11] → 25 > 12 → Shift → [12, 25, 64, 22, 11]

i = 3 (P = 3): key = 22; 64 > 22 → Shift → [12, 22, 25, 11, 64] → 25 > 11 → Shift → [12, 22, 11, 25, 64] → 22 > 11 → Shift → Final: [11, 12, 22, 25, 64]

Explanation:-

We can consider an array

16	8	250	2	18	90	68	52	1
0	1	2	3	4	5	6	7	8

we should take 1st number should be sorted remaining number is unsorted

16
0

sorted

8	250	2	18	90	68	52	1
1	2	3	4	5	6	7	8

← unsorted →

Iteration 1:-

a(1) will be inserted in the correct position of sorted array compare a(1) with a(0)

a(0) > a(1) = (16) > 8

the a(0) will be shifted to right.

8	16
0	1

sorted

250	2	18	90	68	52	1
2	3	4	5	6	7	8

← unsorted →

Iteration 2:-

a[2] will be inserted in the correct position of sorted array compare a[2] with a[1], a[0]

16 > 250 (F), 8 > 250 (F)

8	16	250
sorted	2	

2	18	90	68	52	1
3	4	5	6	7	8

unsorted

Iteration 3:-

a[3] will be inserted in the correct position of sorted array compare a[3] with a[2], a[1], a[0]

i) Compare a[2] with a[3]

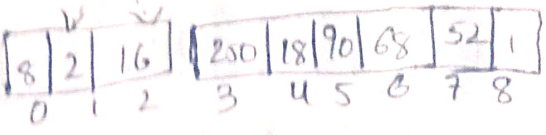
250 > 2 (TRUE) shift to right side

8	16	2
0	1	2

250	18	90	68	52	1
3	4	5	6	7	8

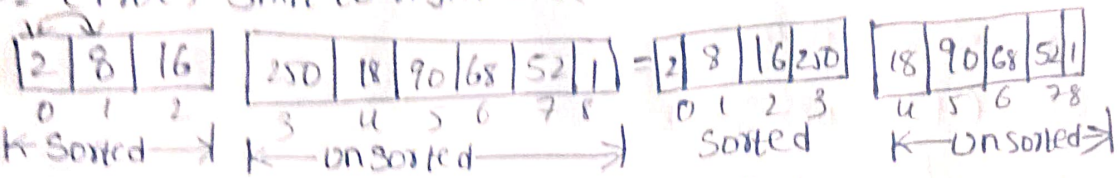
ii) Now check a(1) and a(2)

16 > 2 (true) shift to right side



iii) Compare a(0) with a(3)

8 > 2 (true) shift to right side

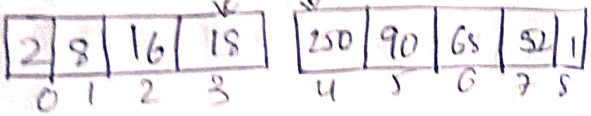


Iteration4:-

Now a[4] will be inserted in correct position of sorted array compare a[4] with a[3] a[2], a[1], a[0].

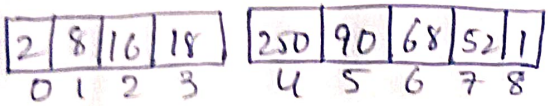
i) Now Compare a(3) with a[4]

250 > 18 (T) move to right side



ii) Compare a[2] with a[3]

16 > 18 (F)

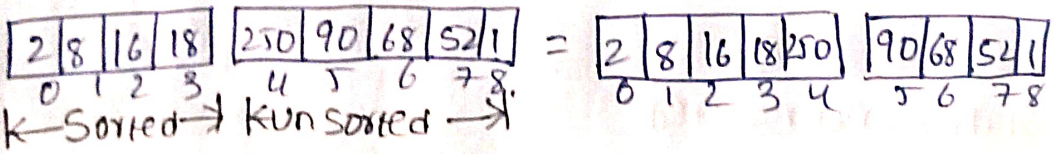


iii) Compare a[1] with a[3]

8 > 18 (False)

iv) Compare a[0] with a[3]

2 > 18 (False)

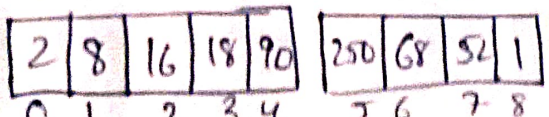


Iteration5:-

Now a[5] will be inserted in correct position of sorted array Compare a[5] with a[4], a[3], a[2], a[1], a[0].

i) Now Compare a[4] with a[5]

250 > 90 (true) move to right side



ii) Now compare a[3] with a[4]

18<90 it remains same

2	8	16	18	90	250	68	52	1
0	1	2	3	4	5	6	7	8

iii) Compare a[2] with a[3]

16<18 it remains same

iv) Compare a[1] with a[2]

8<16 (it remains same)

v) Compare a[0] with a[1]

2<8 (it remains same)

2	8	16	18	90	250	68	52	1
0	1	2	3	4	5	6	7	8

← sorted → | unsorted →

Iteration6:-

Now a[6] will be inserted in the correct position of sorted array. Compare a[5], a[4], a[3] and a[2], a[1], a[0].

i) Compare a[5] with a[6]

250>68 Shift to right side

2	8	16	18	90	68	250	52	1
0	1	2	3	4	5	6	7	8

ii) Compare a[4] with a[5]

90>68 (T) Shift to right side

2	8	16	18	68	90	250	52	1
0	1	2	3	4	5	6	7	8

iii) Compare a[3] with a[4]

18<68 (F) Shift is not required

iv) Compare a[2] with a[3]

16<18 (T) Shift is not required

v) Compare a[1] with a[2]

8<16 (T) Shift is not required

vi) Compare a[0] with a[1]

2<8 (T) Shift is not required

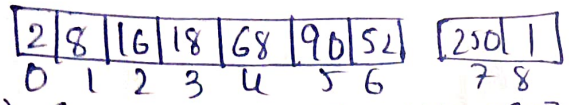
2	8	16	18	68	90	250	52	1
0	1	2	3	4	5	6	7	8

Iteration 7:-

Now a[7] will be inserted in the correct position of sorted array. Compare a[6], a[5], a[4], a[3], a[2], a[1], a[0].

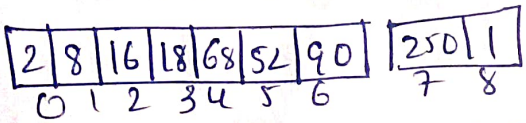
i) Compare a[6] with a[7]

250 > 52 (T) shift to right side



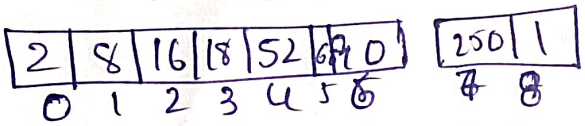
ii) Compare a[5] with a[6]

90 > 52 (T) Shift to right side



iii) Compare a[4] with a[5]

68 > 52 (T) Shift to right side



iv) Compare a[3] with a[4]

18 < 52 (F) Shift is not required

v) Compare a[2] with a[3]

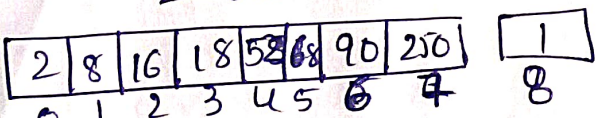
16 < 18 (F) Shift is not required

vi) Compare a[1] with a[2]

8 < 16 (F) Shift is not required.

vii) Compare a[0] with a[1]

2 < 8 (T) Shift is not required

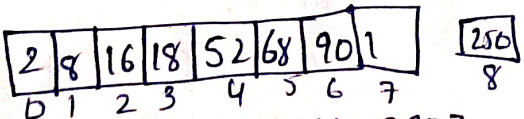


Iteration 8:-

Now a[8] will be inserted in correct position of sorted array. Compare a[7], a[6], a[5], a[4], a[3], a[2], a[1], a[0].

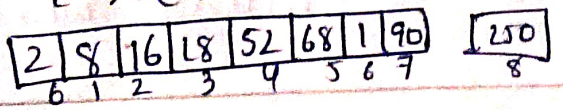
i) Compare a[7] with a[8]

250 > 1 (T) Shift to right side



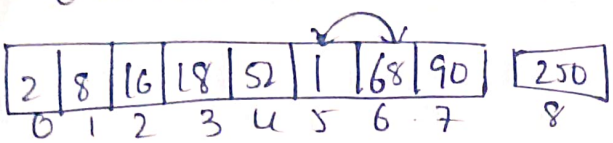
ii) Compare a[6] with a[7]

90 > 1 (T) Shift to right side



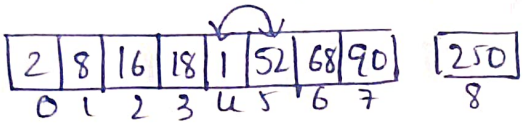
iii) Compare a[5] with a[6]

68 > 1 (T) Shift to right side



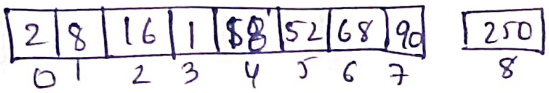
iv) Compare a[4] with a[5]

52 > 1 (T) Shift to right side



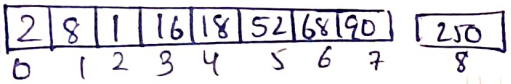
v) Compare a[3] with a[4]

18 > 1 (T) Shift to right Side



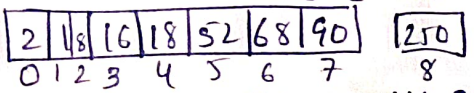
vi) Compare a[2] with a[3]

16 > 1 (T) Shift to right Side



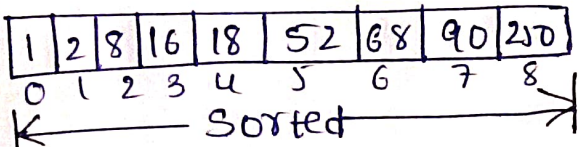
vii) Compare a[1] with a[2]

8 > 1 (T) Shift to right Side



vii) Compare a[0] with a[1]

2 > 1 (T) Shift to right Side



Time complexity:-

Best Case O(n):- Already sorted makes only n-1 comparisons with zero shifts - each key ≥ previous element.

Average Case $O(n^2)$: Random data shifts ~n/2 elements per insertion an average → ~$n^2/4$ shifts + $n^2/2$ comparisons total.

Worst Case $O(n^2)$:

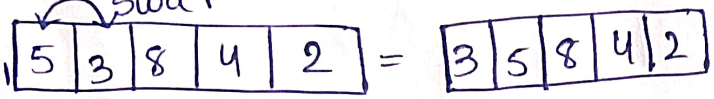
Reverse sorted requires maximum shifts: exactly $n(n-1)/2$ comparisons + $n(n-1)/2$ shifts

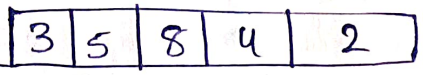
Case	Comparisions	Shifts	Big-O
Best	n-1	0	O(n)
Avg	$\sim n^2/2$	$\sim n^2/4$	$O(n^2)$
worst	$n(n-1)/2$	$n(n-1)/2$	$O(n^2)$

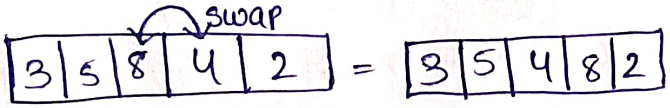
Space O(1) :- O(1) in-Place . Stable (Preserves relative order of equals) Excellent for small / nearly-sorted data (n<50 or streaming input).

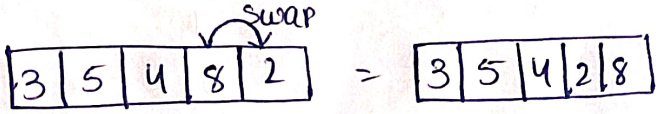
Bubble sort Step by Step trace :-

Pass 1 :- (i = 1 to 4)

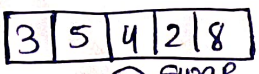
1) i=1 

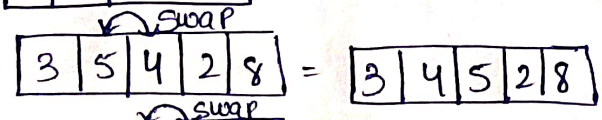
2) i=2 

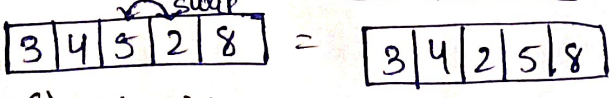
3) i=3 

4) i=4 

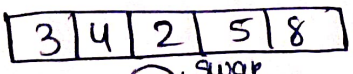
Pass 2 :- (i = 1 to 3)

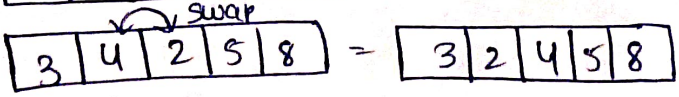
1) i=1 

2) i=2 

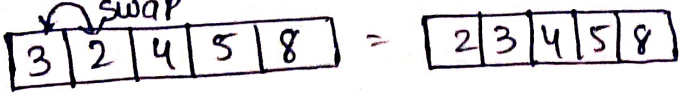
3) i=3 

Pass 3 :- (i = 1 to 2) :-

1) i=1 

2) i=2 

pass 4 :- (i = 1 to 1)

1) i=1 

Selection sort:

Selection sort Systematically builds a sorted prefix by repeatedly selecting the minimum element from the unsorted suffix and placing it at the current position. For i from 0 to n-2, it scans the entire unsorted subarray (i to n-1) to find the smallest element's index, then swaps it with A[i]. Each pass permanently fixes one more element in the sorted prefix, performing exactly one swap per outer iteration regardless of data arrangement. Unlike bubble sort's adjacent swaps, selection sort minimizes swaps (n-1 maximum) but always scans full unsorted regions, making comparisions the bottleneck at $O(n^2)$.

Algorithm:

```
Void selectionSort (comparable a[], int n)
{
 for (int i=0; i<n-1; i++) //Loop through each element
 {                          Starting from index
  int min = i; //Assume the current index is
  for (int j=i+1; j<n; j++)   the minimum
    if (a[j].compareTo (a[min]) <0)
      min=j;    //compare current element with
                  current minum
    swap(a,i,min); //swap the smallest element
  }               with the first unsorted element
}
```

Algorithm explanation:

1. Assume the first element is the minimum.
2. Compare it with all remaining elements.
3. Find the smallest element in the unsorted part
4. Swap it with the first unsorted element.
5. Move the boundary of sorted array one step forward
6. Repeat until the array is sorted.

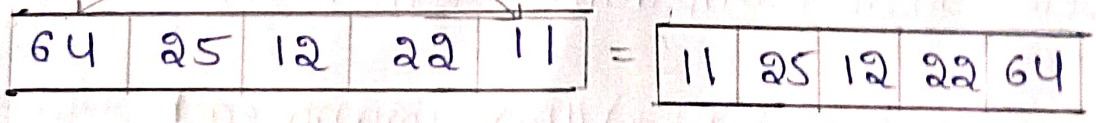
Step by step trace:

Trace insertion sort an array [64, 25, 12, 22, 11]

Initial: ~~[5p3, 8p4,~~ [64, 25, 12, 22, 11]

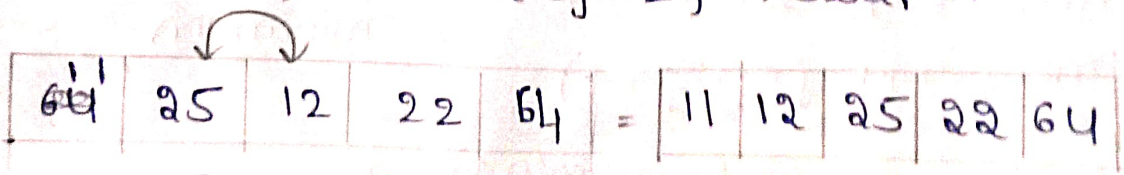
Pass 1 (i=0):

min = 11 (j=4) → swap A



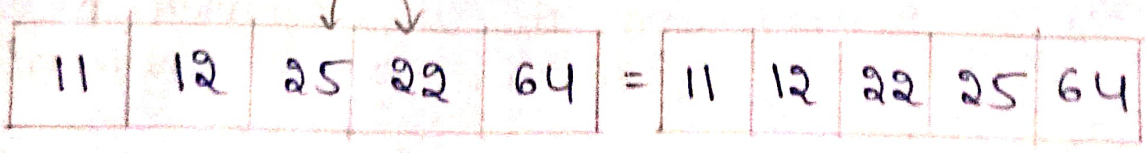
Pass 2 (i=1):

min scan → min = 12 (j=2) → swap A.



Pass 3 (i=2):

min scan → min = 22 (j=3) → swap A.



Pass 4 (i = 3):

min scan → min = 25 (j = 3) → no swap

11	12	22	25	64

Explanation:

We can consider an array

18	60	2	15	48	1	5	100	56

Iteration 1:

choose a[min] from the array

a[min] = a[5] = 1

Swap a[5] with a[0]

1	60	2	15	48	18	5	100	56

Iteration 2:

choose a[min] from a[1] to a[8]

a[min] = a[2]

Swap a[2] with a[1]

1	2	60	15	54	18	5	100	56

Iteration 3:

choose a[min] from a[2] to a[8]

a[min] = a[6]

Swap a[6] with a[2]

1	2	5	15	48	18	60	100	56

Iteration 4:

choose a[min] from a[3] to a[8]

a[min] = a[3]

No swap ~~a[3]~~

1	2	5	15	48	18	60	100	56

Iteration 5:

choose a[min] from a[4] to a[8]

a[min] = a[5]

swap a[5] with a[4]

1	2	5	15	18	48	60	100	56

Iteration 6:

choose a[min] from a[5] to a[8]

a[min] = a[5]

NO swap.

Iteration 7:

choose a[min] from a[6] to a[8]

a[min] = a[8]

swap a[8] with a[6]

1	2	5	15	18	48	56	100	60

Iteration 8:

choose a[min] from a[7] to a[8]

a[min] = a[8]

swap a[8] with a[7]

1	2	5	15	18	48	56	60	100

Time complexity:

Best case - $O(n^2)$:

Already sorted requires full scans of unsorted portions each pass - always exactly $\frac{n(n-1)}{2}$ comparisons.

Average case - $O(n^2)$

Random elements → checks all remaining elements each time.

worst case - $O(n^2)$:

Reverse sorted performs identical full scans to best case: precisely $n(n-1)/2$ comparisons every execution.

case	comparisions	swaps	Big-O
Best	$n(n-1)/2$	$\leq n-1$	$O(n^2)$
Avg	$\sim n^2/2$	$\leq n-1$	$O(n^2)$
worst	$n(n-1)/2$	$\leq n-1$	$O(n^2)$

Space complexity:

$O(1)$ in-place Not stable (may swap equal elements) optimal for swap-minimization but rarely practical beyond $n<100$.