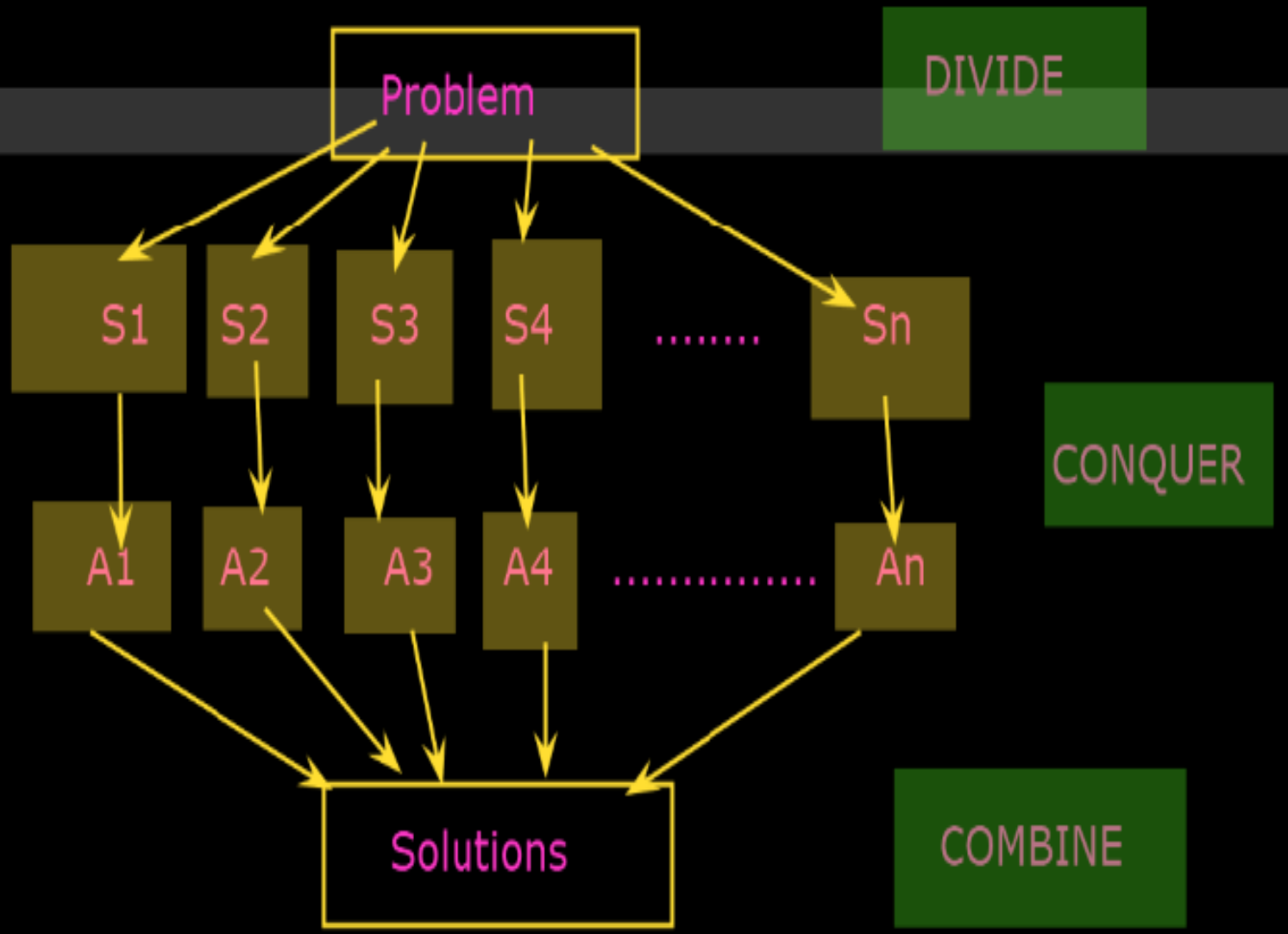
Algorithms & Data Structure

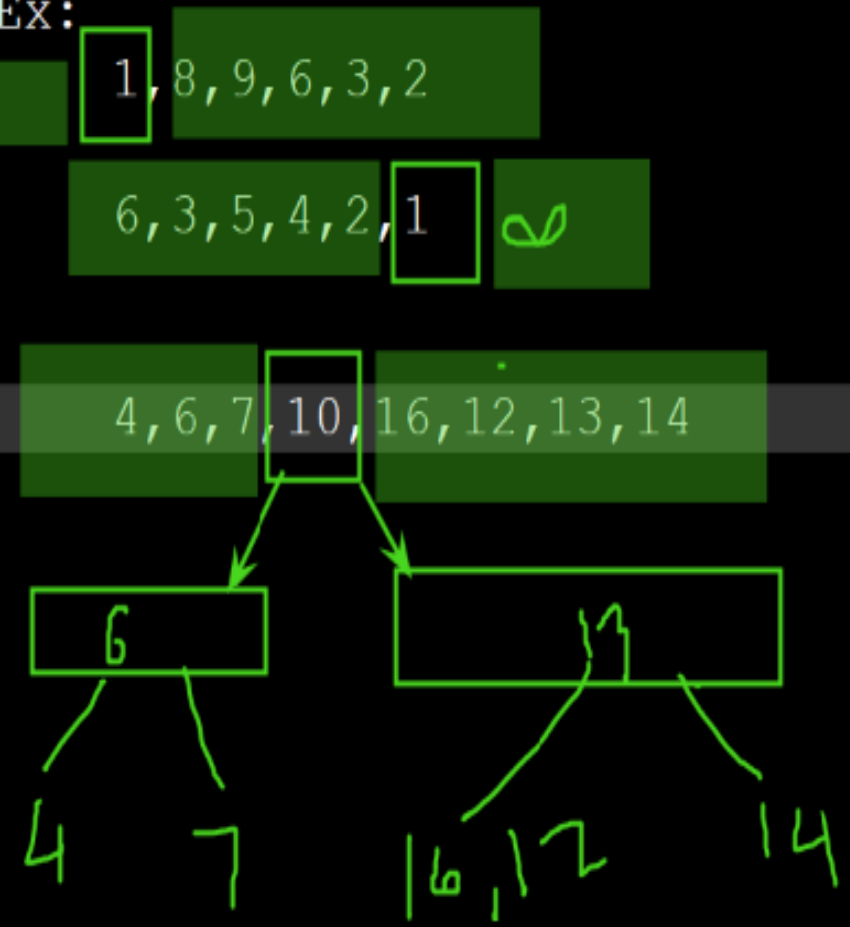
Kiran Waghmare

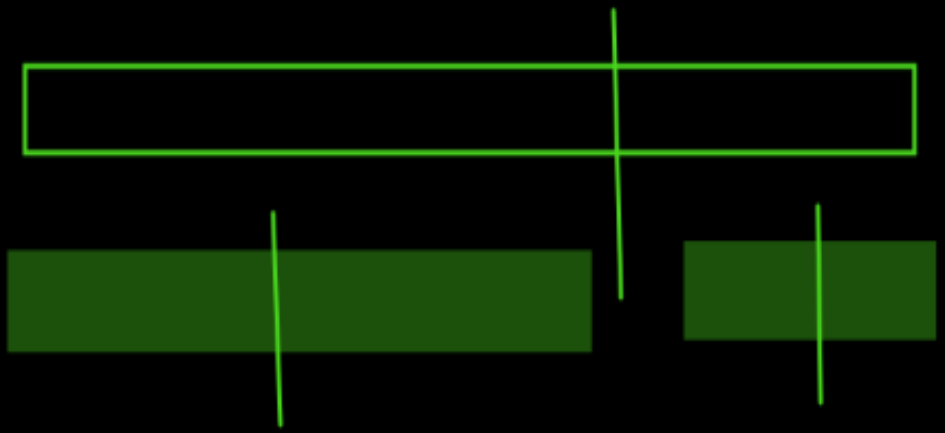
Divide and Conquer:



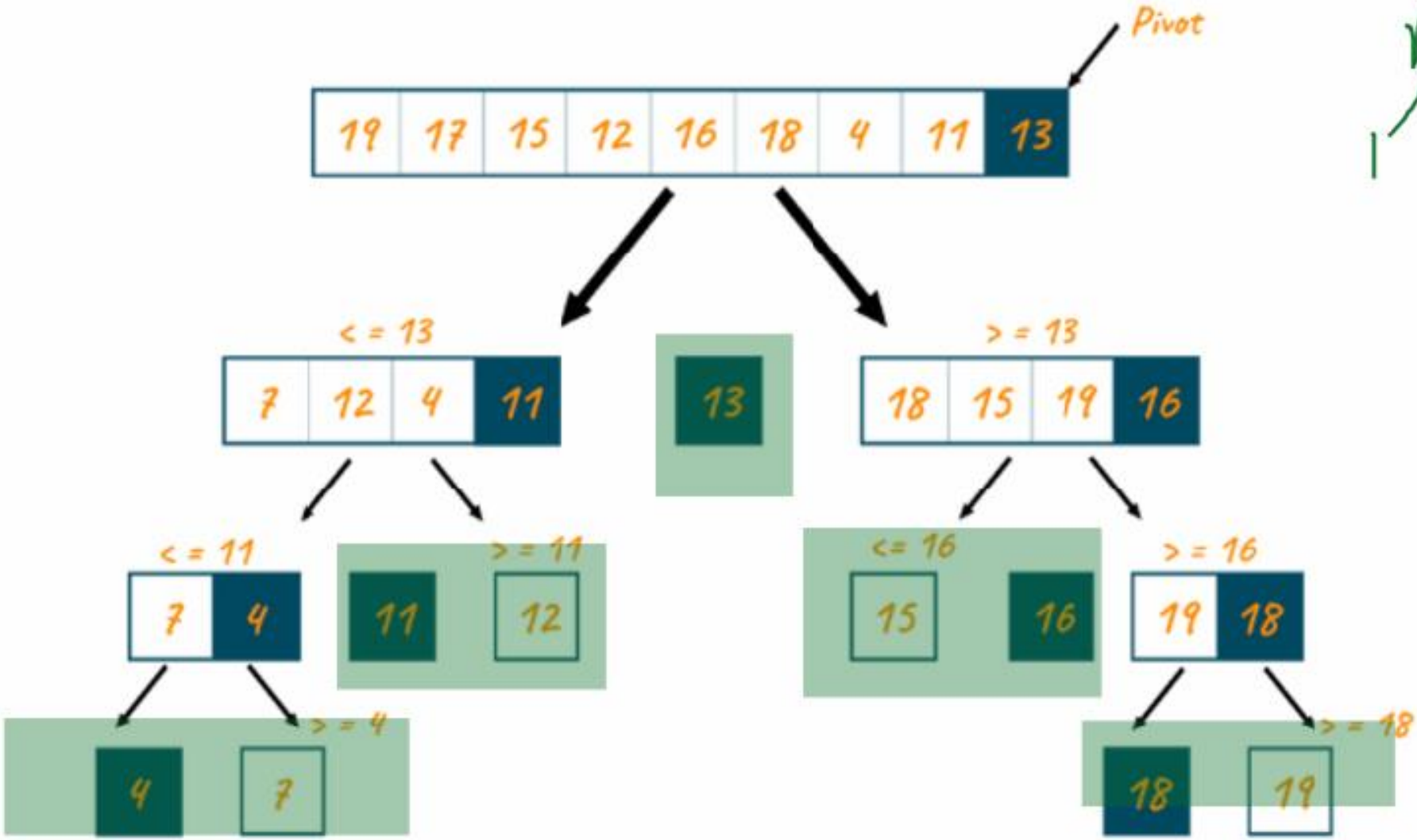
Quick sort:
pivot element:

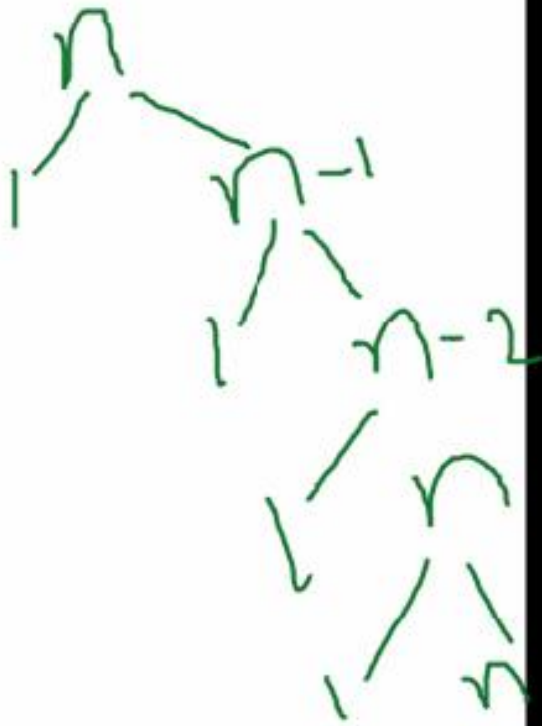
Ex:





Quick Sort Algorithm





Searching in Arrays

- **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.
  E.g. We have linear array A as below:

1	2	3	4	5
15	50	35	20	25

Suppose item to be searched is 20. We will start from beginning and will compare 20 with each element. This process will continue until element is found or array is finished. Here:

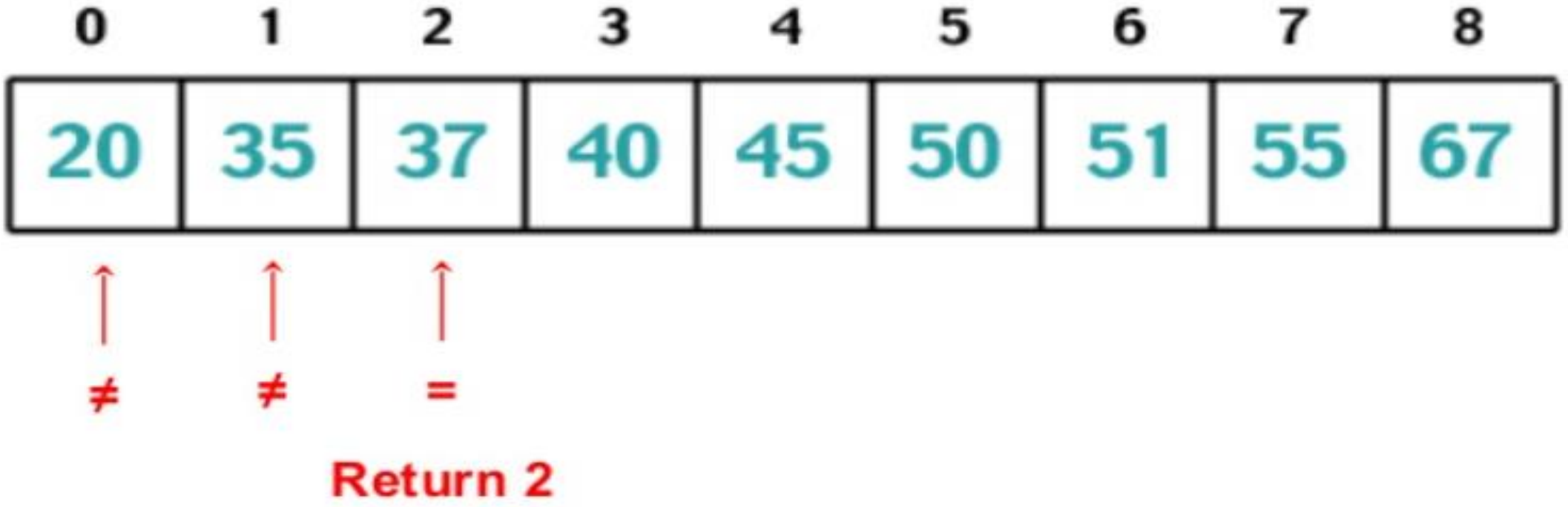
1) Compare 20 with 15
   20 # 15, go to next element.
2) Compare 20 with 50
   20 # 50, go to next element.

3) Compare 20 with 35
   20 #35, go to next element.

4) Compare 20 with 20
   20 = 20, so 20 is found and its location is 4.

Linear Search

- Find 37?



Linear Search

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Program 3

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

Examples :

Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}
x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
110, 100, 130, 170}
x = 175;
Output : -1
Element x is not present in arr[].

The time complexity of the above algorithm is O(n).

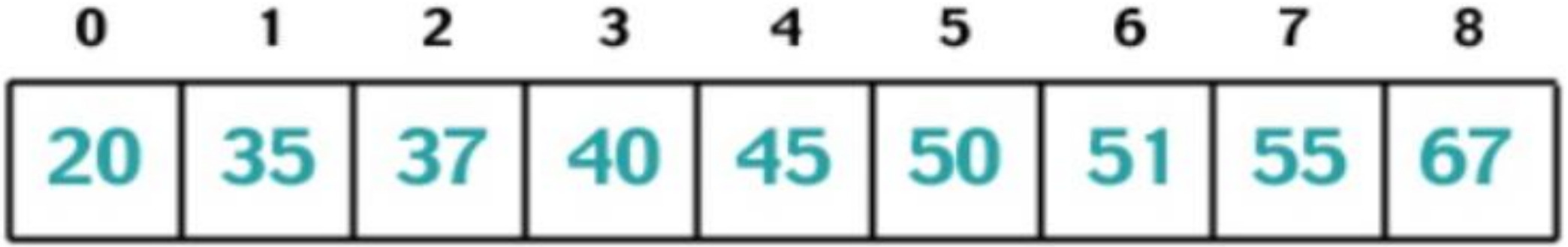
Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster-searching comparison to Linear search.

Improve Linear Search Worst-Case Complexity

if element Found at last O(n) to O(1)
if element Not found O(n) to O(n/2)

Binary Search

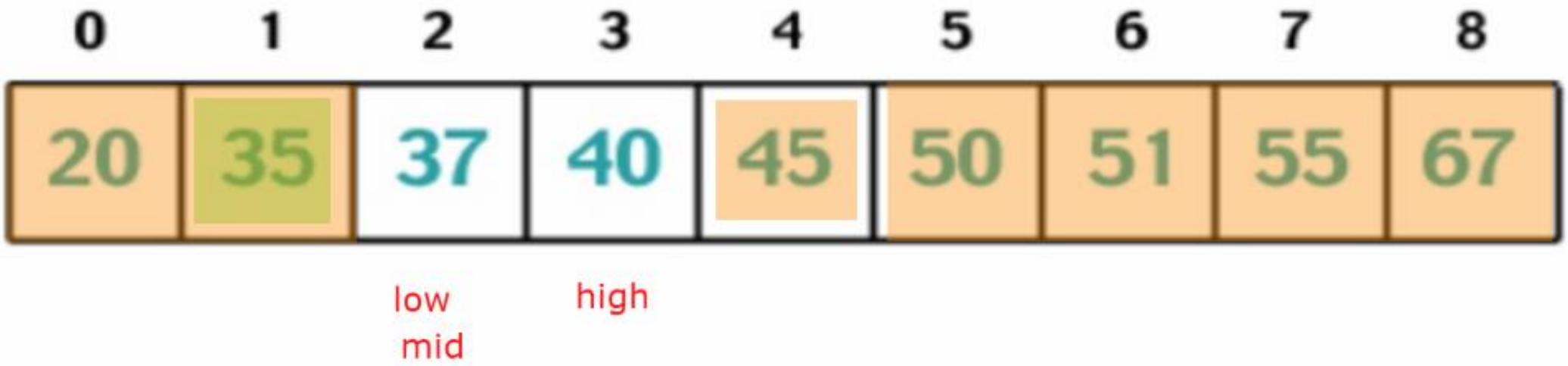
- Find 37?
  1. Sort Array.



Binary Search

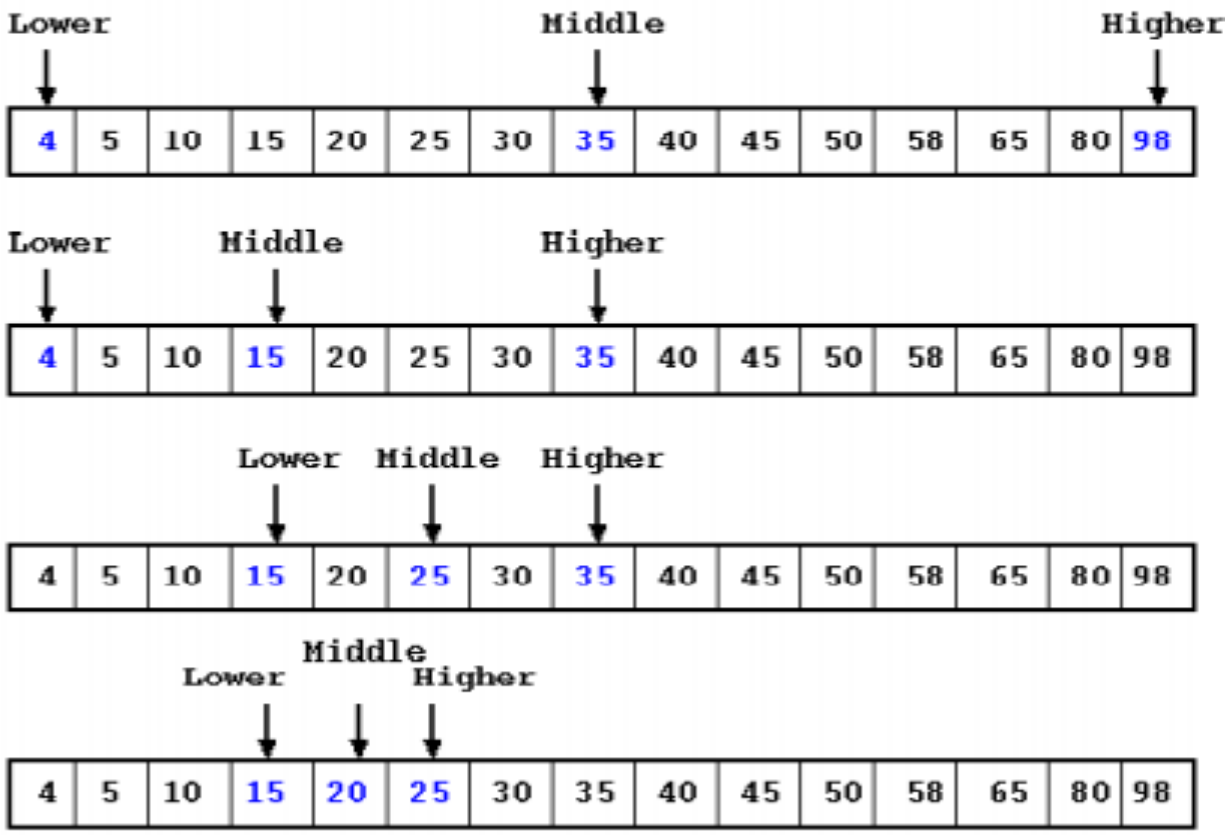
low	high	mid
0	8	4 → 1
0	3	1 → 2
2	3	2 → 3

- Find 37?
  1. Sort Array.



Binary Search

- The binary search algorithm can be used with only sorted list of elements.
- Binary Search first divides a large array into two smaller sub-arrays and then recursively operate the sub-arrays.
- Binary Search basically reduces the search space to half at each step



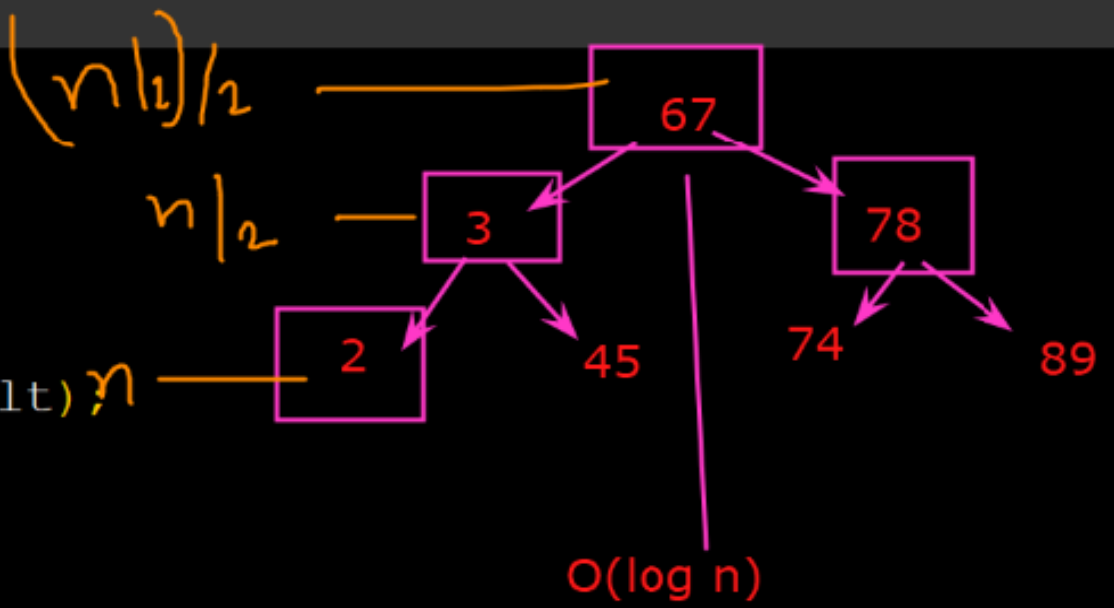
Binary Search

- **Example**: Consider the following elements stored in an array and we are searching for the element 67. The trace of the algorithm is given below.

	BEG & END	MID = (BEG+END)/2	Compare	Location
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEG MID END	BEG = 0 END = 4	MID = (0+4)/2 MID = 2	67>39 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 39 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEGMID END	BEG = 3 END = 4	MID = (3+4)/2 MID = 3	67>47 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 47 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 BEGMID END	BEG = 4 END = 4	MID = (4+4)/2 MID = 4	67>57 (Does not match)	LOC = -1
The search element i.e. 67 is greater than the element in the middle position i.e. 57 then continues the search to the right portion of the middle element.				
A[0] A[1] A[2] A[3] A[4] 12 23 39 47 57 ENDBEG	BEG = 5 END = 4	Since the condition (BEG <= END) is false the comparison ends		
We get the output as **67 Not Found**				

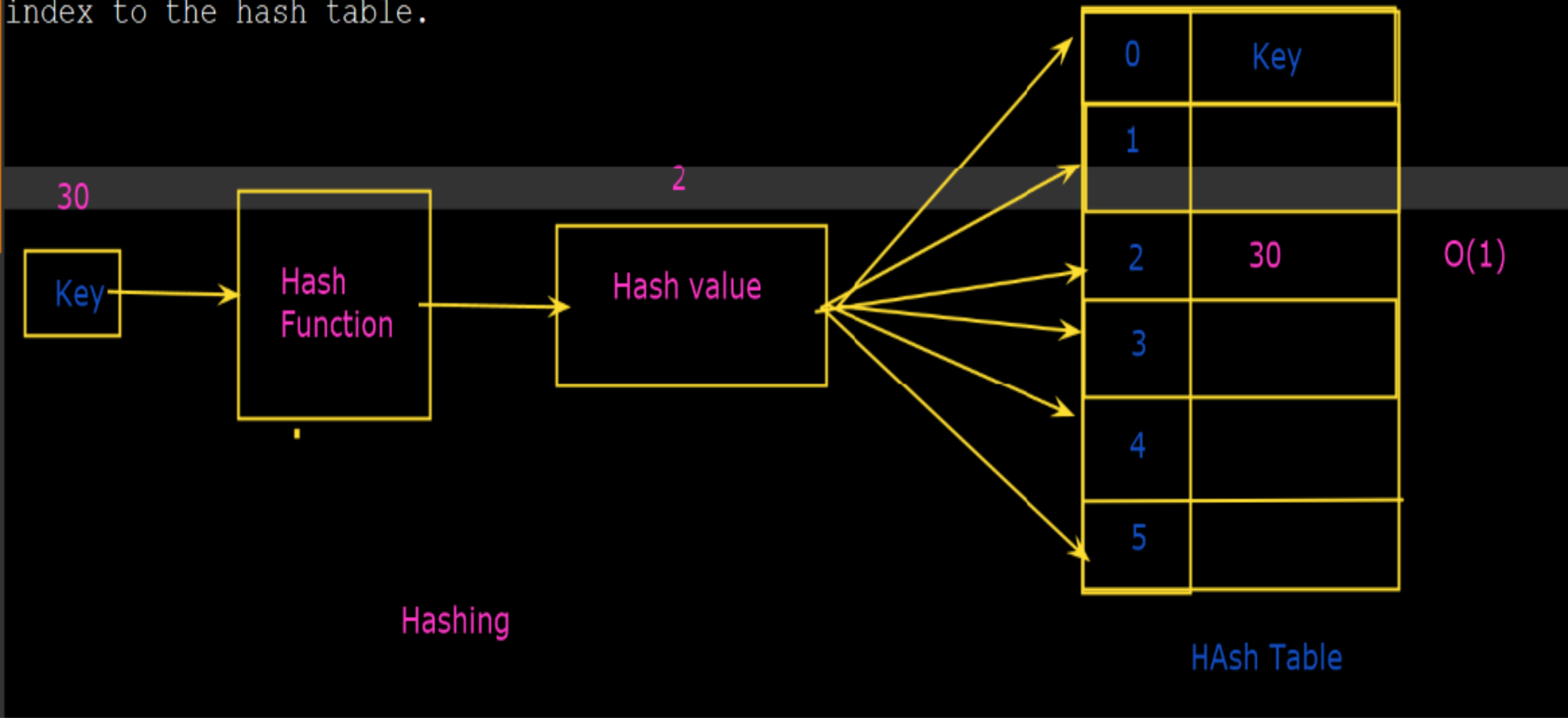
```java
public static void main(String args[]){

    int arr[]={2,31,45,67,74,78,89};
    int x =2;
    int n = arr.length;
    int result = search(arr, x,0,n-1);
    if(result == -1)
        System.out.println("Not found!");
    else
        System.out.println("Found! "+result);
}
```



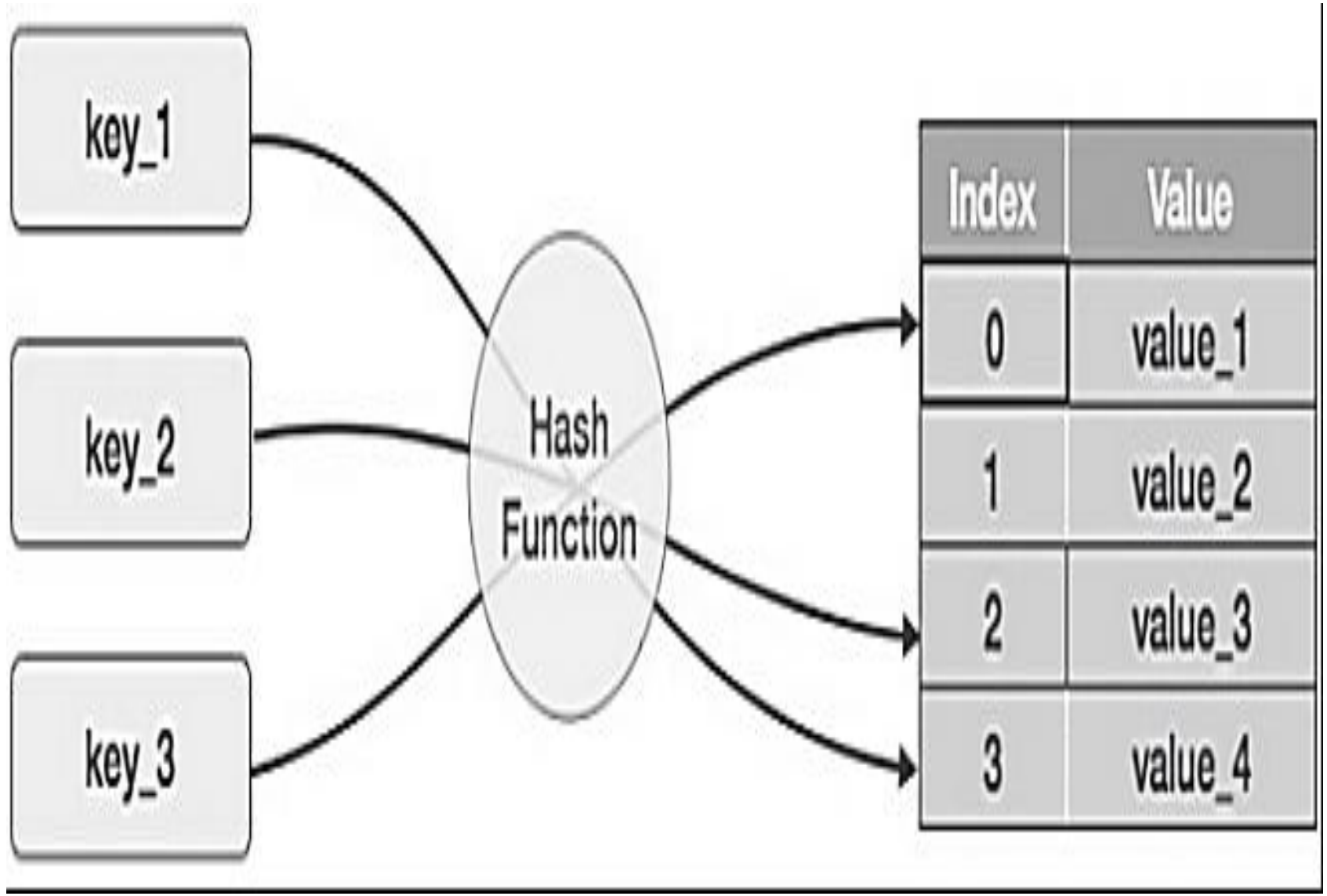
Root node: <comp Min (Best case) O(1)
Leaf node: > comp Max (Worst case) O(log n)
Height of tree---> time complexity

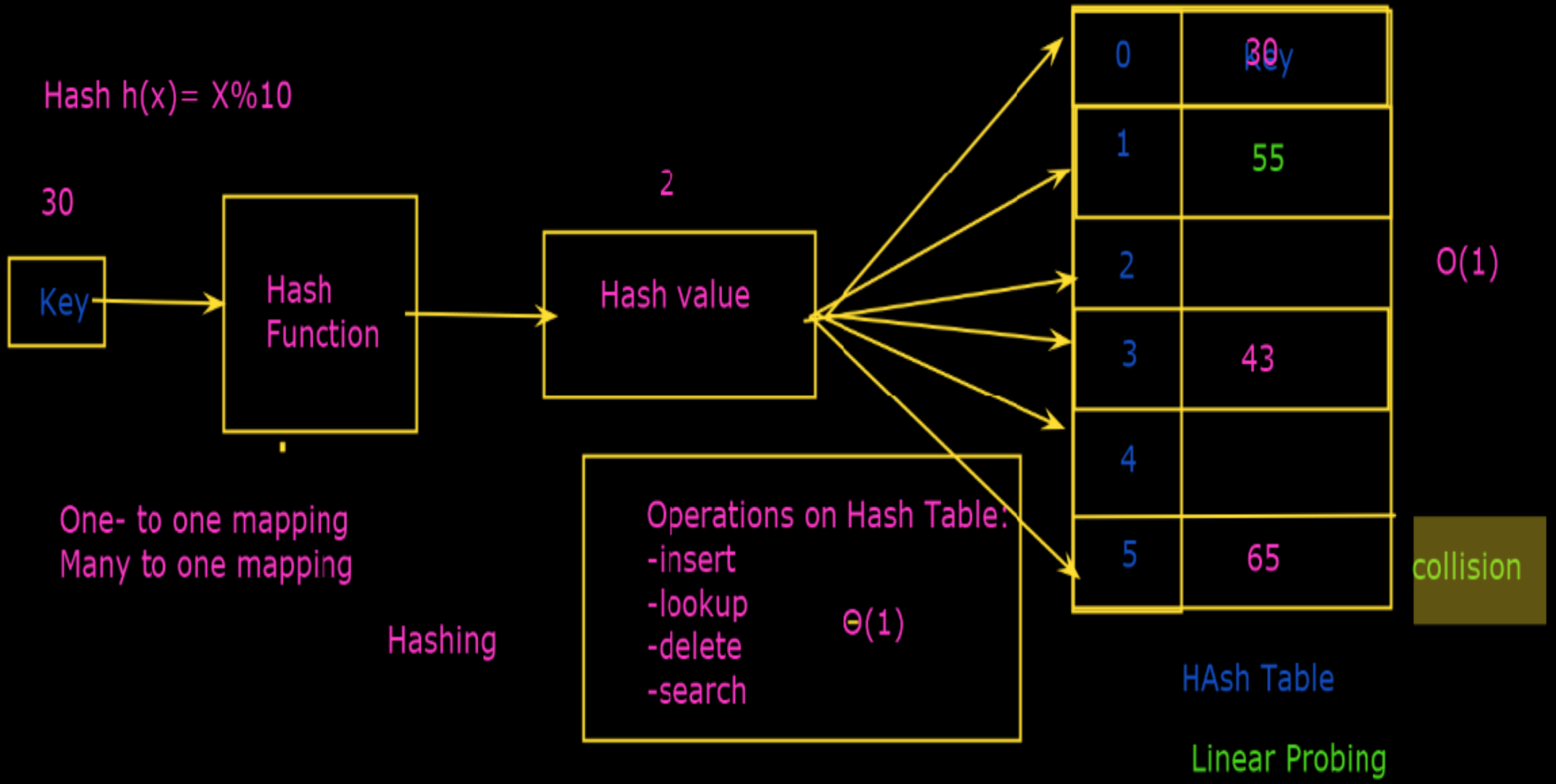
-The hash function receives hash key, which returns the index of an element in an array called as hash table.
-A perfect hash function maps each search key into a different integer suitable as an index to the hash table.



Hash Table

- A **hash table** is a data structure that stores elements and allows insertions, lookups, and deletions to be performed in O(1) time.
- A **hash table** is an alternative method for representing a dictionary
- In a hash table, a **hash function** is used to map keys into positions in a table. This act is called **hashing**
- Hash Table Operations
  - **Search**: compute f(k) and see if a pair exists
  - **Insert**: compute f(k) and place it in that position
  - **Delete**: compute f(k) and delete the pair in that position
- In ideal situation, hash table search, insert or delete takes $\Theta(1)$


key_1
key_2
key_3
Hash Function
Index
Value
0
value_1
1
value_2
2
value_3
3
value_4


Hash h(x)= X%10
30
Key
Hash Function
2
Hash value
0
30
1
55
2
3
43
4
5
65
O(1)
collision
One- to one mapping
Many to one mapping
Hashing
Operations on Hash Table:
-insert
-lookup
-delete
-search
Θ(1)
HAsh Table
Linear Probing

E:\Data Structures\Hanif_Search\Searching\ Unit28_Hashing1.ppt

Some Applications of Hash Tables (1/2)

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.
- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

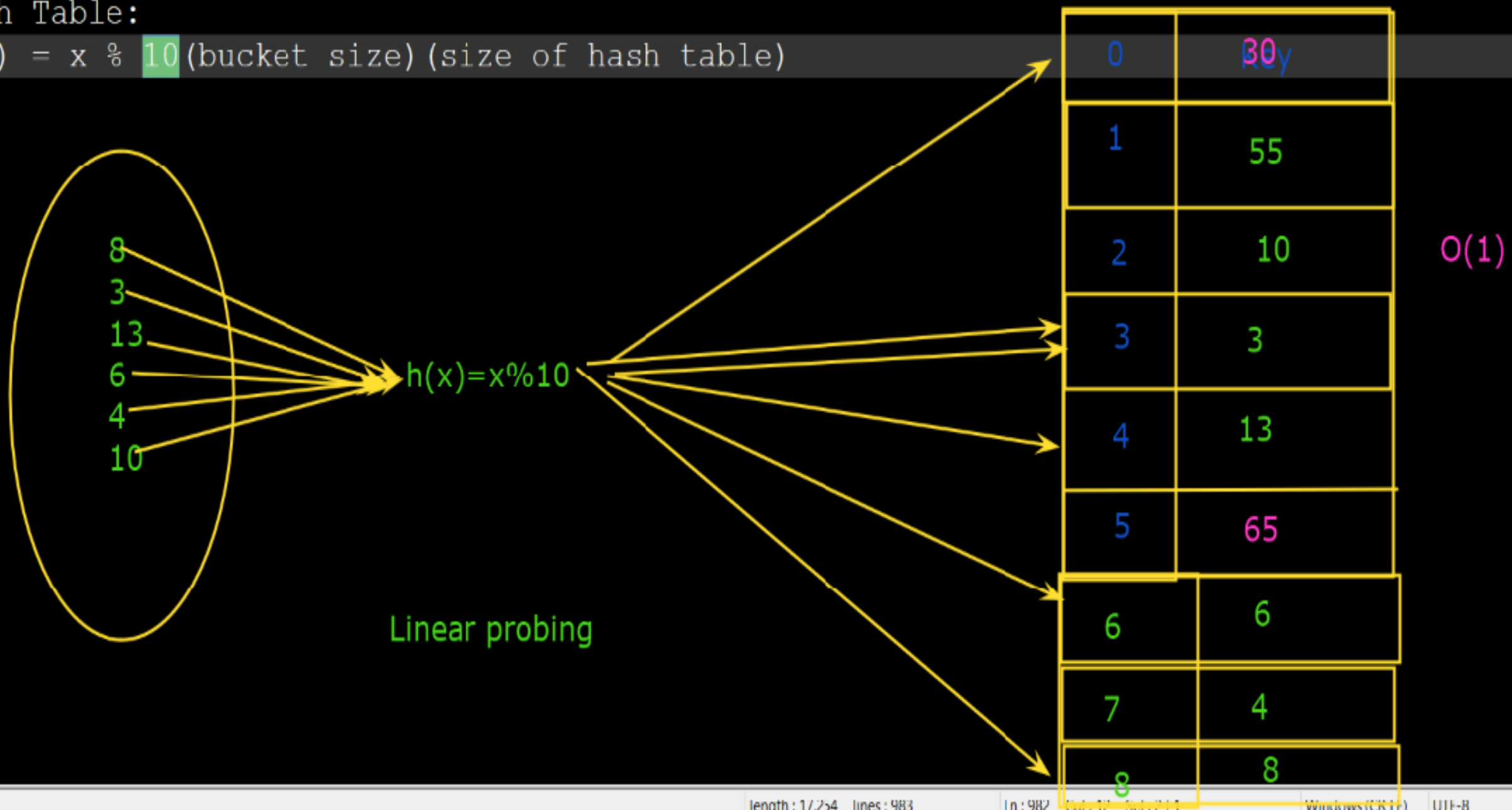
Clip slide

Some Applications of Hash Tables (2/2)

- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.
- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.
- **Browser Cashes**: Hash tables are used to implement browser cashes.

Hash Table:

h(x) = x % 10 (bucket size) (size of hash table)



Example 1: Illustrating Hashing (1/2)

- **Use the function `f(r) = r.id % 13` to load the following records into an array of size 13.**

Zaid	1.73	985926
Musab	1.60	970876
Muneeb	1.58	980962
Adel	1.80	986074
Adnan	1.73	970728
Yousuf	1.66	994593
Husain	1.70	996321

Example 1: Illustrating Hashing (2/2)

Name	ID	h(r) = id % 13
Zaid	985926	6
Musab	970876	10
Muneeb	980962	8
Adel	986074	11
Adnan	970728	5
Yousuf	994593	2
Husain	996321	1

0	1	2	3	4	5	6	7	8	9	10	11	12
	Husain	Yousuf			Louai	Zaid		Muneeb		Musab	Adel	

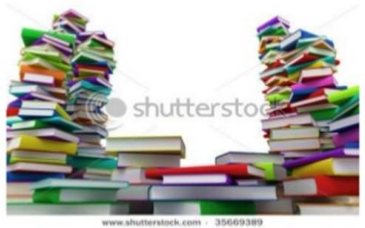
Hash table

Type	Unordered associative array
Invented	1953

Time complexity in big O notation

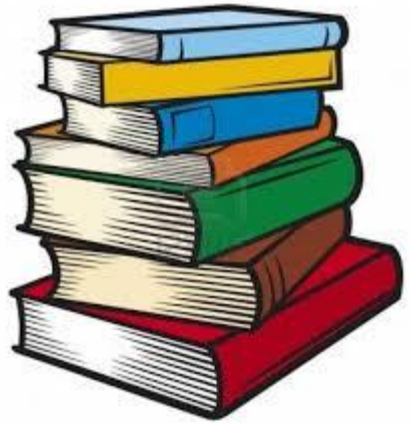
Algorithm	Average	Worst case
Space	O(*n*)[1]	O(*n*)
Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)

Examples of stack



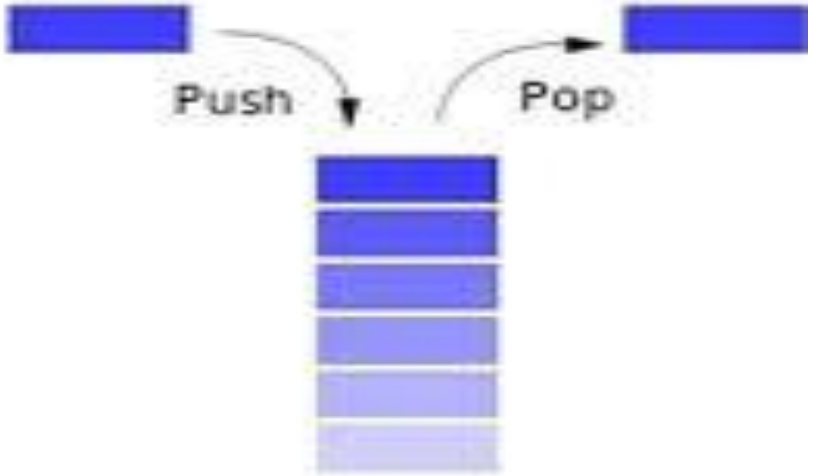
Stacks

Kiran Waghmare



Stack of books

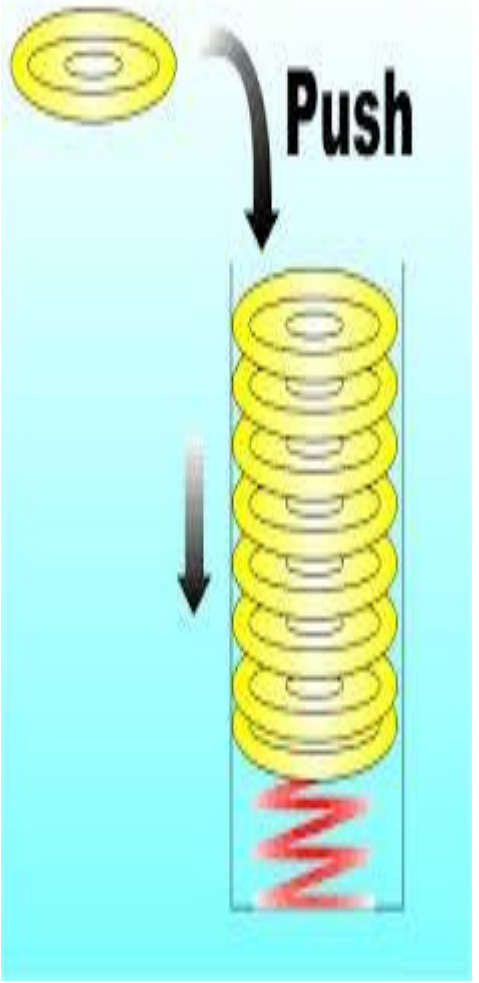
Stack of Coins



Memory stack

Stack

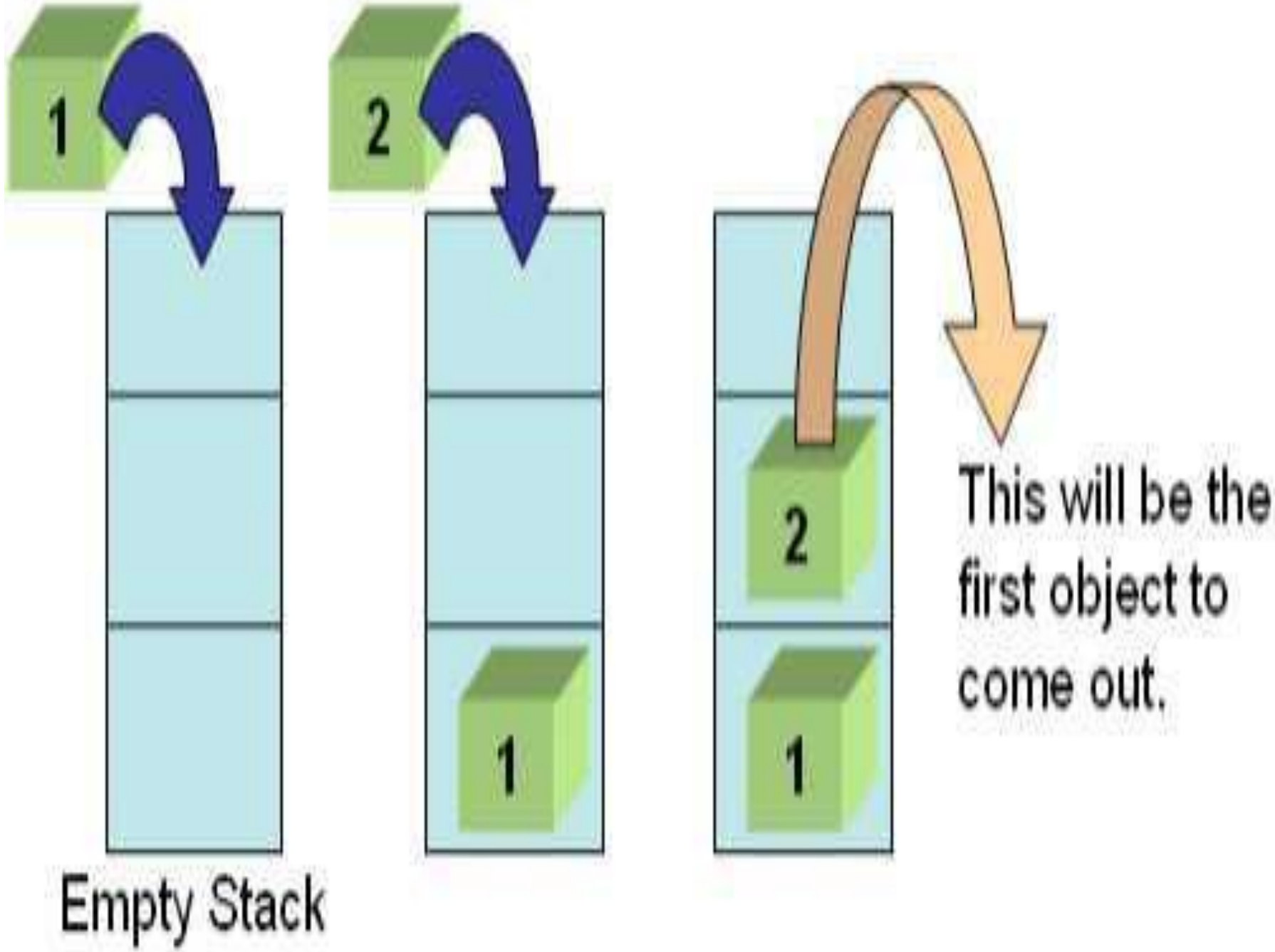
- Stack is an ordered list of similar data type.
- Stack is a LIFO structure. (Last in First out).
- push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

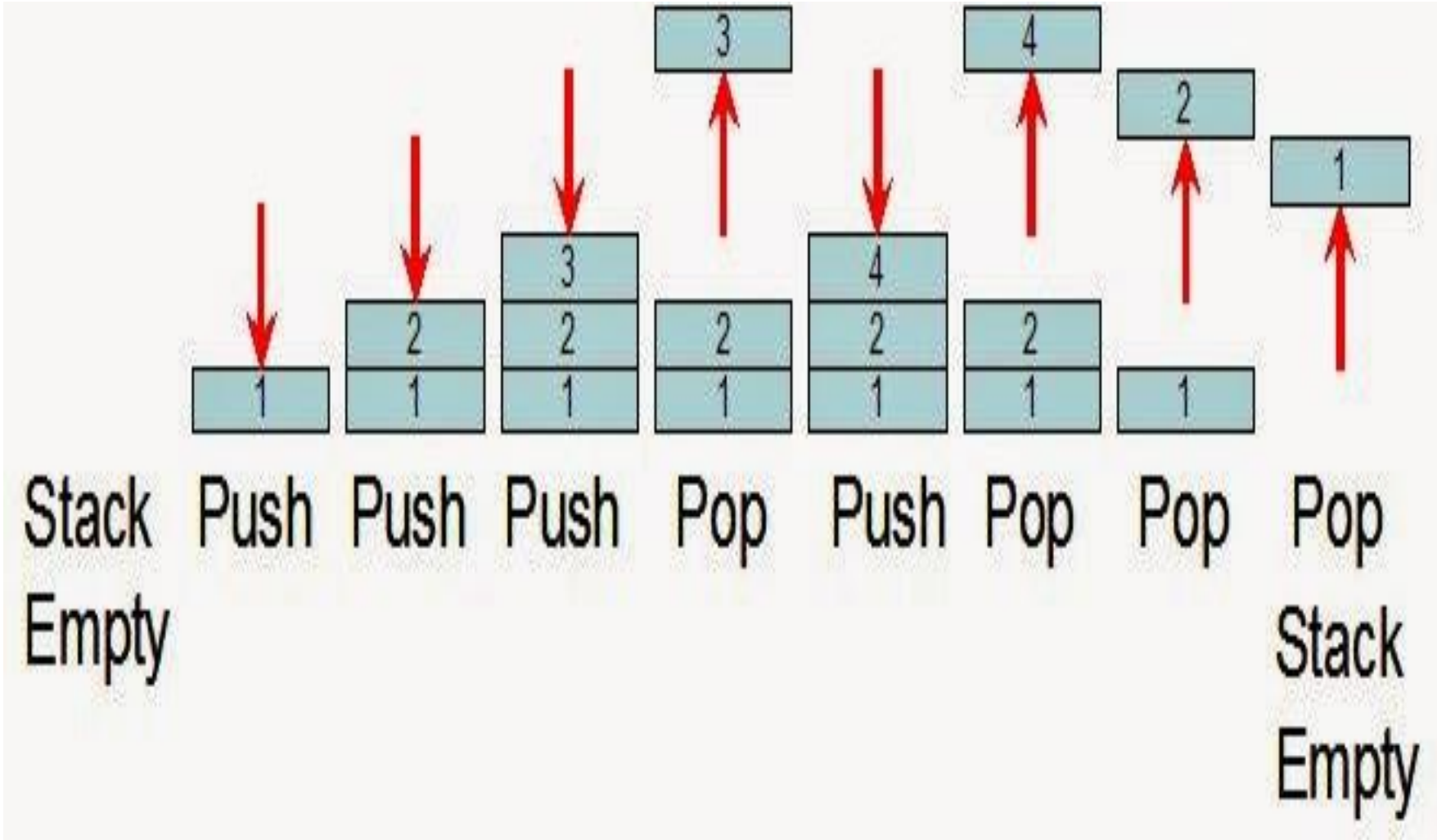


Stack:
-data structure
-ordered list of similar data types.
-LIFO sturcture(Last in First out)
-TOS (TOP) : Top of stack
-Operations:
    -Push (insert)
    -Pop (delete)
    -isempty()
    -isFull()

Standard Stack Operations

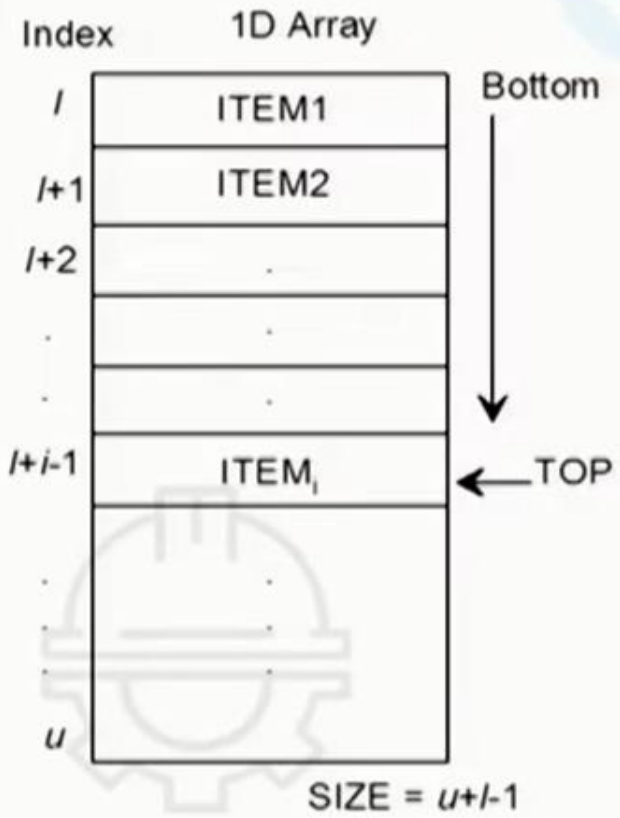
- **The following are some common operations implemented on the stack:**
- **push():**
  - When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():**
  - When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():**
  - It determines whether the stack is empty or not.
- **isFull():**
  - It determines whether the stack is full or not.'
- **peek():**
  - It returns the element at the given position.
- **count():**
  - It returns the total number of elements available in a stack.
- **change():**
  - It changes the element at the given position.
- **display():**
  - It prints all the elements available in the stack.


1
2
2
1
2
1
This will be the first object to come out.
Empty Stack

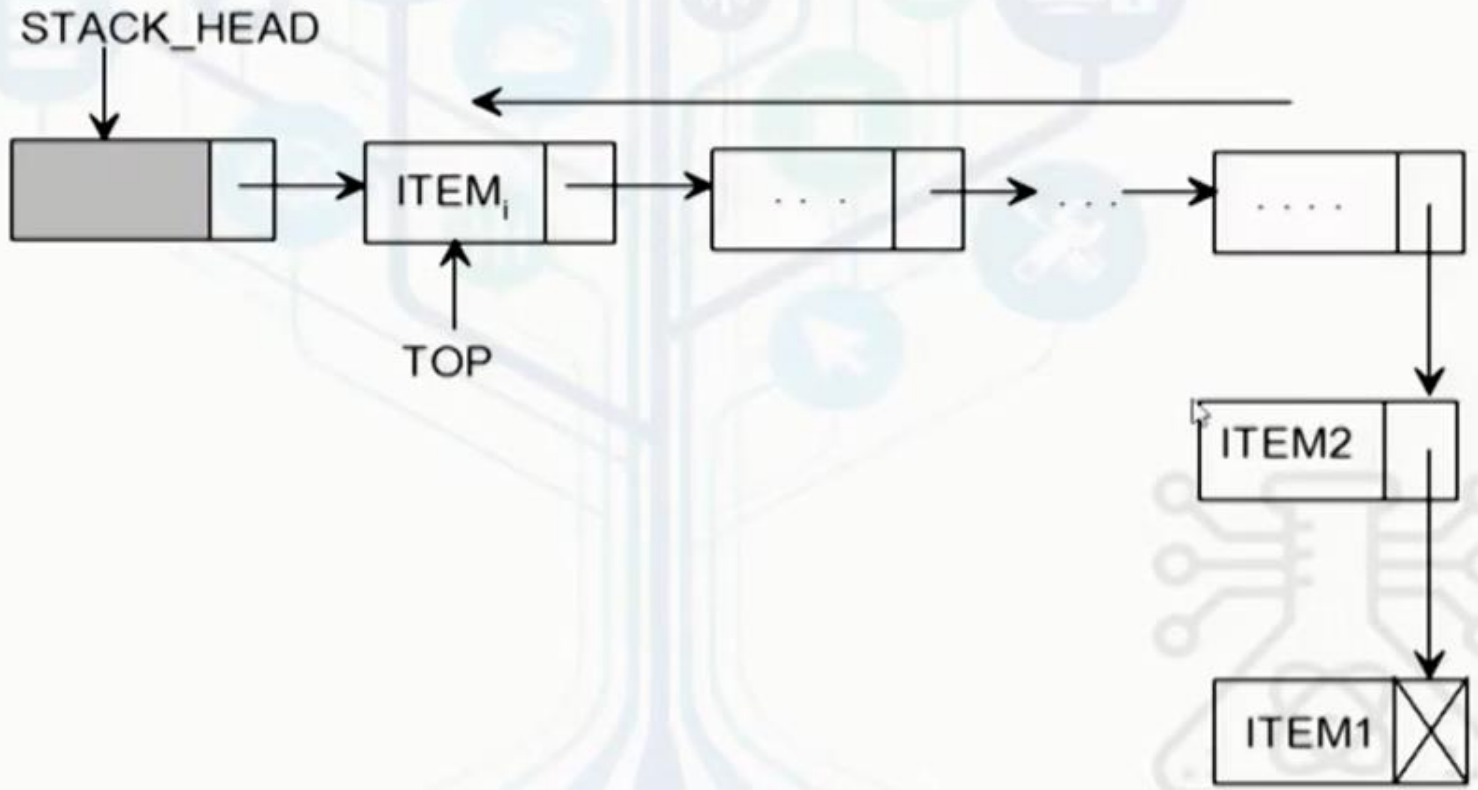

Stack Empty
Push
Push
Push
Pop
Push
Pop
Pop
Pop
Stack Empty
1
2
3
4

Memory representations

Array representation



Linked list representation



Operations on stack

Push — To insert an item into the stack

Pop — To remove an item from a stack

Status — To know the present state of a stack

- if stack is empty,
- size of the stack,
- the element at top

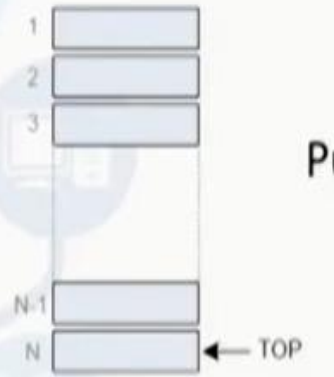
Operation: Push with array

We have assumed that the array index varies from 1 to SIZE and TOP points the location of the current top-most item in the stack. The following algorithm defines the insertion of an item into a stack represented using an array A.

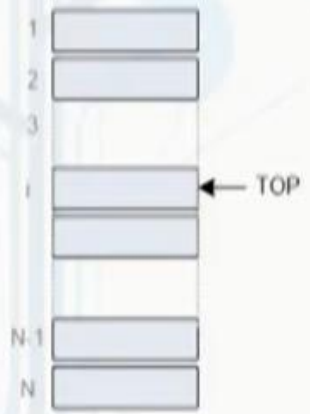
Input: The new item ITEM to be pushed onto it.
Output: A stack with a newly pushed ITEM at the TOP position.
Data structure: An array *A* with TOP as the pointer.

Steps:

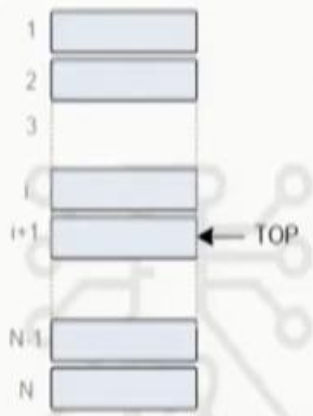
1. **If** TOP ≥ SIZE **then**
2. + **Print** "Stack is full"
3. **Else**
4. TOP = TOP + 1
5. A[TOP] = ITEM
6. **EndIf**
7. **Stop**



Push operation is failed



Before push()



After push()

Insertion in Stack

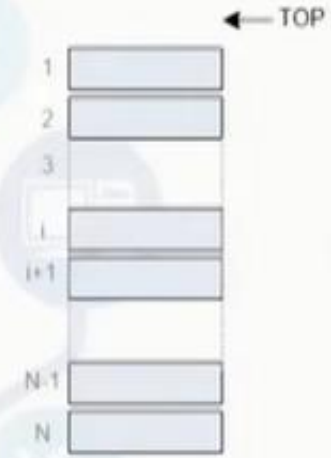
Operation: Pop with array

The following algorithm defines the deletion of an item from a stack represented using an array A.

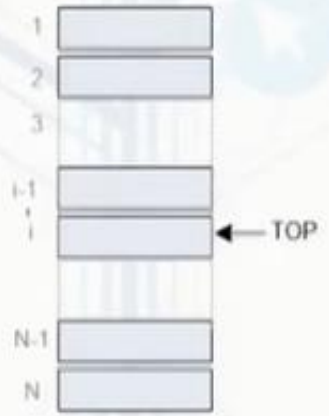
Input: A stack with elements.
Output: Removes an ITEM from the top of the stack if it is not empty.
Data structure: An array A with TOP as the pointer.

Steps:

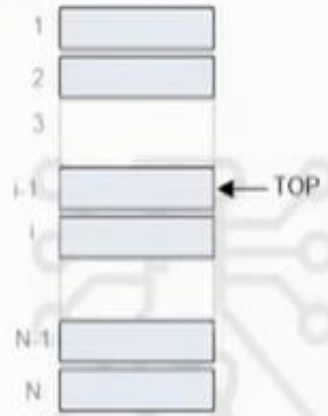
1. **If** TOP < 1 **then**
2. **Print** "Stack is empty"
3. **Else**
4. ITEM = A[TOP]
5. TOP = TOP – 1
6. **EndIf**
7. **Stop**



Pop operation is failed



Before Pop()



After Pop()

Deletion in Stack

Applications of Stack

- **The following are the applications of the stack:**

- **Balancing of symbols:**
  - Stack is used for balancing a symbol. For example, we have the following program:
  - As we know, each program has an opening and closing braces;
  - when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack.
  - Therefore, the net value comes out to be zero.
  - If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:**
  - Stack is also used for reversing a string.
  - For example, we want to reverse a "javaTpoint" string, so we can achieve this with the help of a stack.
  - First, we push all the characters of the string in a stack until we reach the null character.
  - After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

Applications of Stack:

---

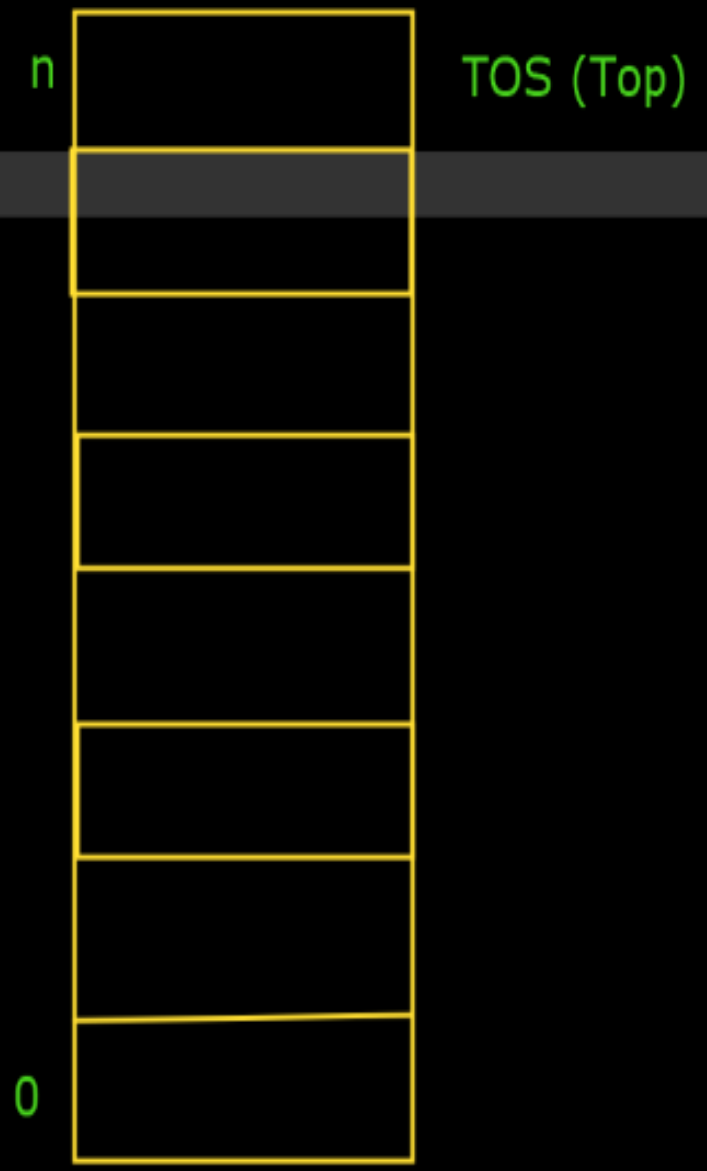
1. Balancing of symbols:

Input: (()()) (( ))) ([{)}}

(())) ((((

())(

)((



Applications of Stack:

1. Balancing of symbols:

2. String reversal

```
reverse(StringBuffer str)
{
    int n = str.length();

    for(int i=0;i<n;i++)
    {
        push(str.charAt(i));
    }

    for(int i=0;i<n;i++)
    {
        char ch = pop();
        str.setCharAt(i, ch);
    }
}
```

- **UNDO/REDO:**
  - It can also be used for performing UNDO/REDO operations.
  - For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc.
  - So, there are three states, a, ab, and abc, which are stored in a stack.
  - There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  - If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:**
  - The recursion means that the function is calling itself again.
  - To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):**
  - This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:**
  - Suppose we have to create a path to solve a maze problem.
  - If we are moving in a particular path, and we realize that we come on the wrong way.
  - In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

- **Expression conversion:**
  - Stack can also be used for expression conversion.
  - This is one of the most important applications of stack.
  - The list of the expression conversion is given below:
    - Infix to prefix
    - Infix to postfix
    - Prefix to infix
    - Prefix to postfix
    - Postfix to infix
- **Memory management:**
  - The stack manages the memory.
  - The memory is assigned in the contiguous memory blocks.
  - The memory is known as stack memory as all the variables are assigned in a function call stack memory.
  - The memory size assigned to the program is known to the compiler.

  When the function is created, all its variables are assigned in the stack memory.

  When the function completed its execution, all the variables assigned in the stack are released.

Check for Balanced Brackets in an expression (well-formedness) using Stack

- Given an expression string exp, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in the given expression.

- Example:

- Input: exp = “[()]{}{[()()]()}”
- Output: Balanced
- Explanation: all the brackets are well-formed

- Input: exp = “[(])”
- Output: Not Balanced
- Explanation: 1 and 4 brackets are not balanced because
- there is a closing ‘]’ before the closing ‘(‘

Polish Notations

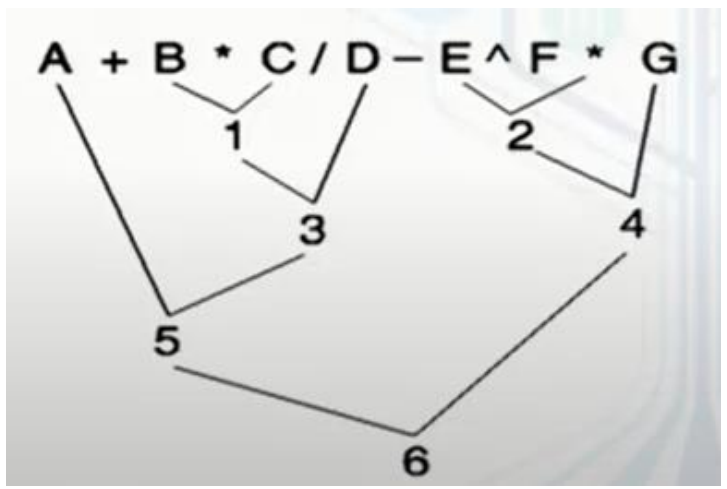
1. **Infix Notation : A+B**
2. **Prefix Notation: +AB**
3. **Postfix Notation : AB+**

- **Operator Precedence:**
  1. **BODMAS Rule**
  2. **Brackets, Exponential, (* / %), (+ -)**
- **Rules: Infix to Postfix Conversion**
  1. Parenthesize the expression starting from left to right.
  2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in above expression B* C is parenthesized first before A+B.
  3. The sub-expression (part of expression) which has been converted into postfix is to be treated as single operand.
  4. Once the expression is converted to postfix from remove the parenthesis.

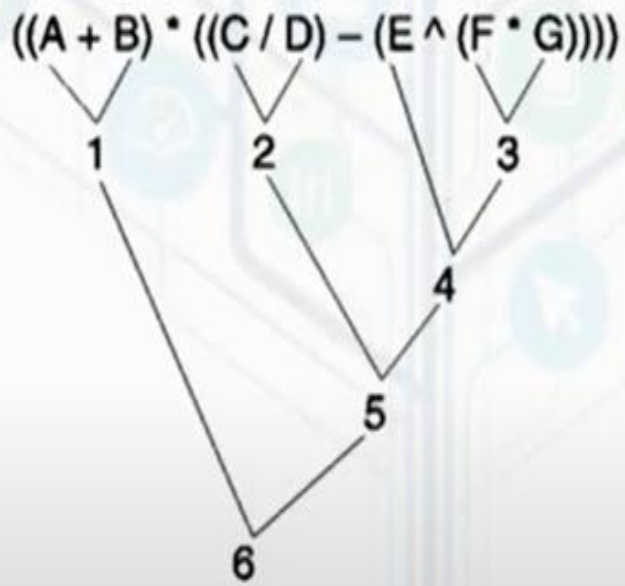
A + B * C / D – E ^ F * G

Precedence and associativity of operators

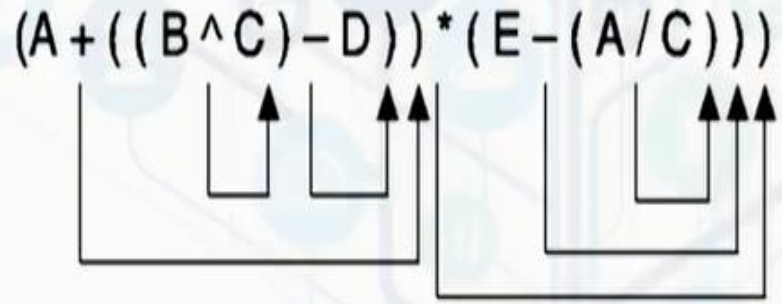
Operators	*Precedence*	*Associativity*
– (unary), +(unary), NOT	6	–
^ (exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), – (subtraction)	4	Left to right
<, <=, +, < >, >=	3	Left to right
AND	2	Left to right
OR, XOR	1	Left to right



((A + B) * ((C/D) – (E ^ (F * G))))



((A + ((B ^ C) – D)) * (E – (A/C)))



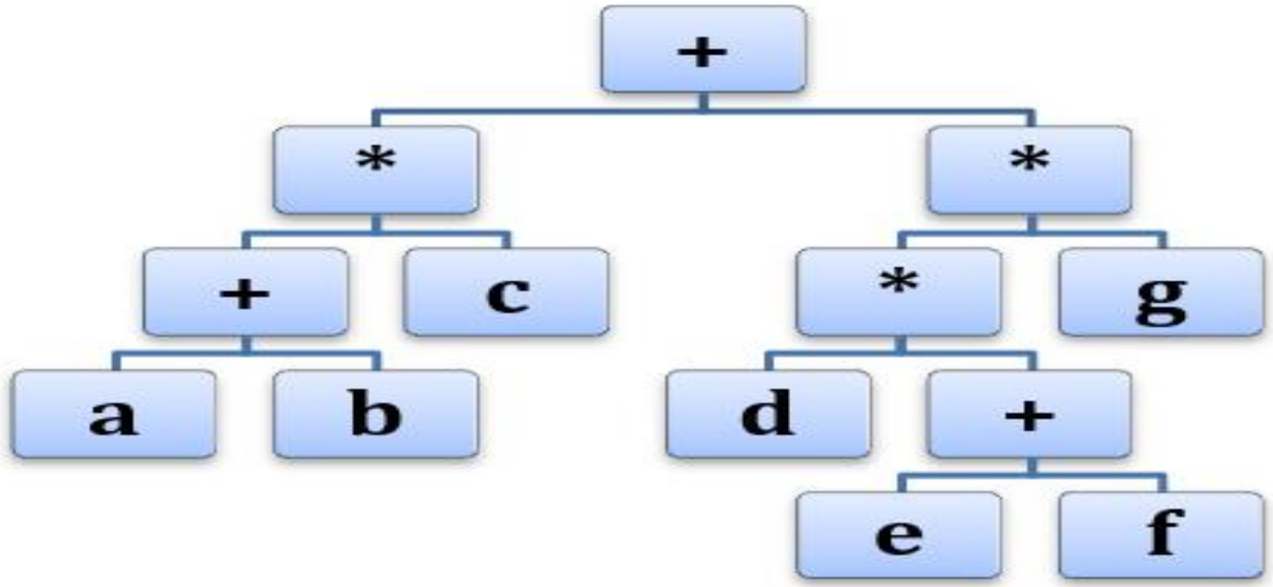
A B C ^ D – + E A C / – *

Application: Evaluation of a postfix expression

Steps:

1. Append a special delimiter '#' at the end of the expression
2. item = *E*.**ReadSymbol**( ) // Read the first symbol from *E*
3. **While** (item ≠ '#') **do**
4. **If** (item = operand) **then**
5. **PUSH**(item) // Operand is the first push into the stack
6. **Else**
7. op = item // The item is an operator
8. *y* = **POP**( ) // The right-most operand of the current operator
9. *x* = **POP**( ) // The left-most operand of the current operator
10. *t* = *x* op *y* // Perform the operation with operator 'op' and operands *x*, *y*
11. **PUSH**(*t*) // Push the result into stack
12. **EndIf**
13. item = *E*.**ReadSymbol**( ) // Read the next item from *E*
14. **EndWhile**
15. value = **POP**( ) // Get the value of the expression
16. **Return**(value)
17. **Stop**

Construct a binary expression tree for the following statement :
(a + b * c) + ((d * e + f) * g)



Balance Parenthesis

- Given an expression string exp, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in exp.
- **Example**:
- ***Input**: exp = “[(])”*
  ***Output**: Not Balanced*

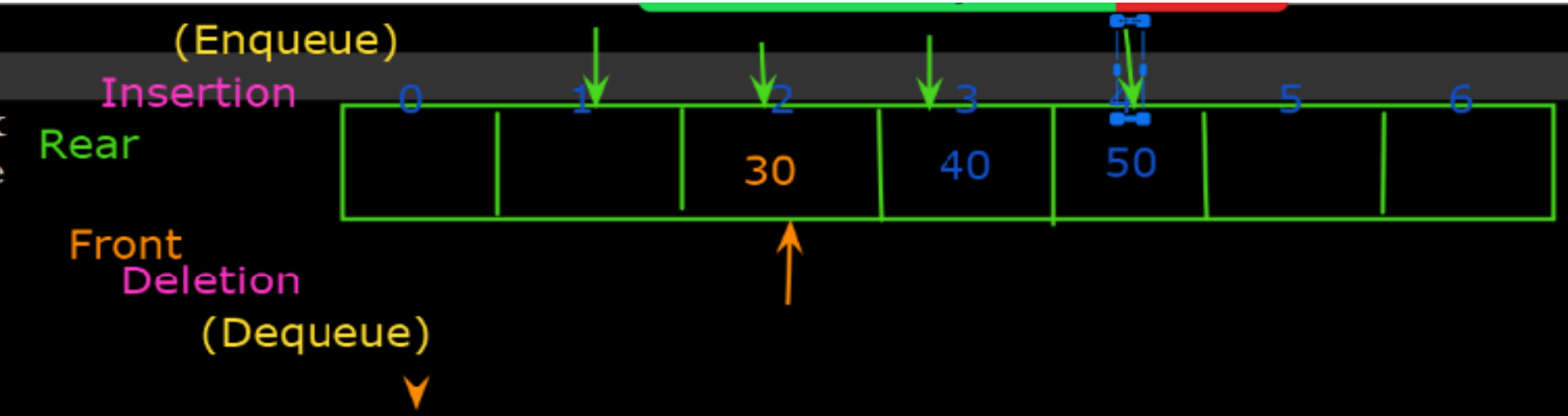
- ***Input**: exp = “[()]{}{[()()]()}”*
  ***Output**: Balanced*

Agenda:

-Stack

-Queue

Queue:



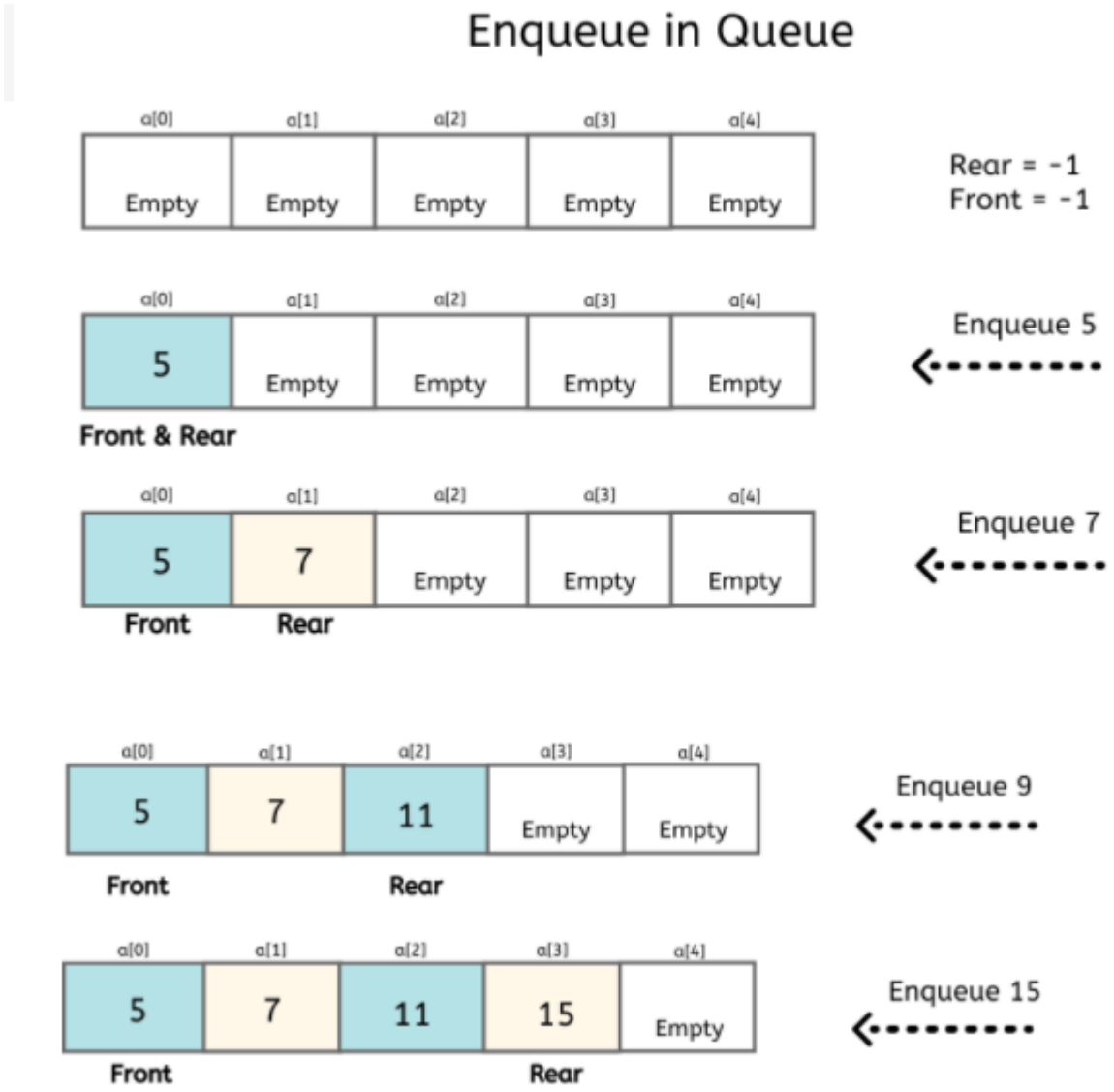
Queue

- Ordered collection of homogeneous elements.
- Non-primitive linear data structure.
- A new element is added at one end called rear end and the existing elements are deleted from the other end called front end.
- This mechanism is called First-In-First-Out (FIFO)
- Total no. of elements in queue = rear-front+1

1. Enqueue()

When we requirea to add an element to the Queue we perform Enqueue() operation.

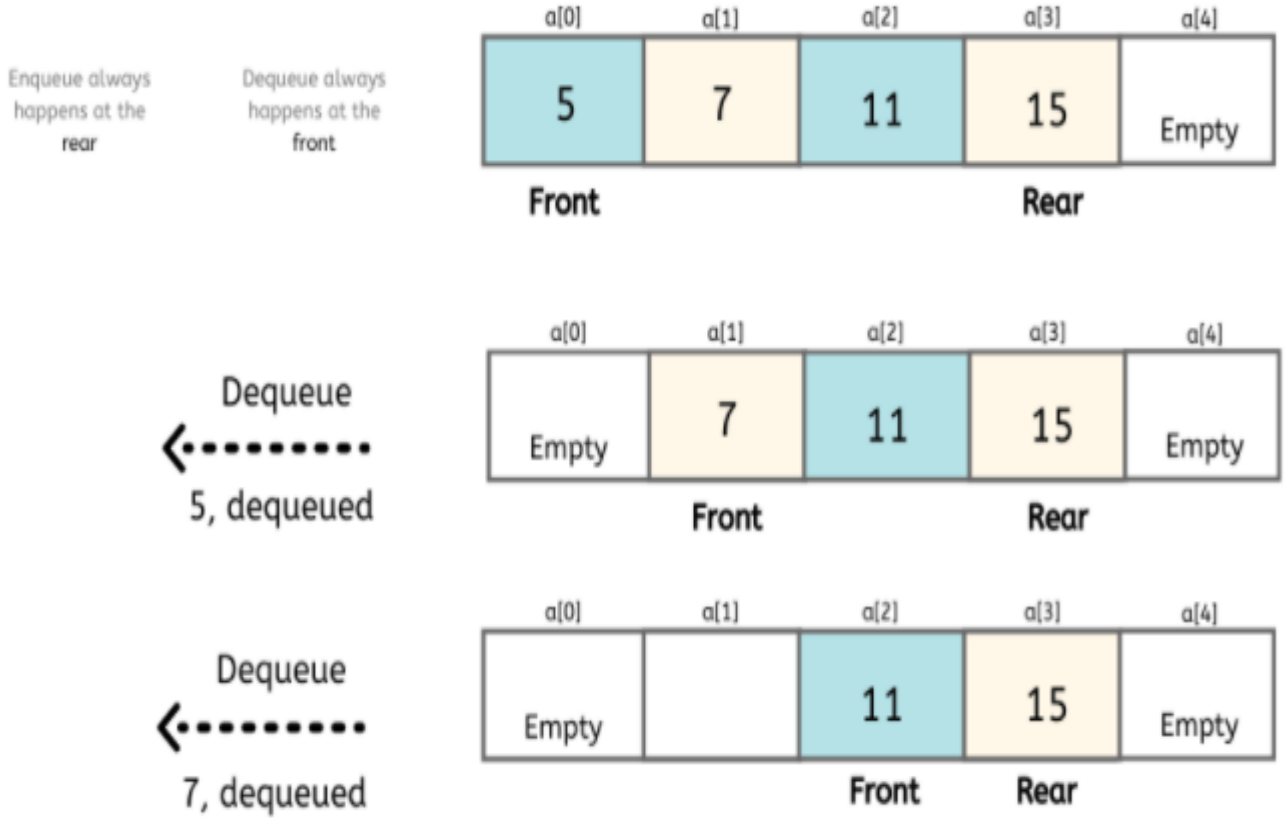
Push() operation is synonymous of insertion/addition in a data structure.



2. Dequeue()

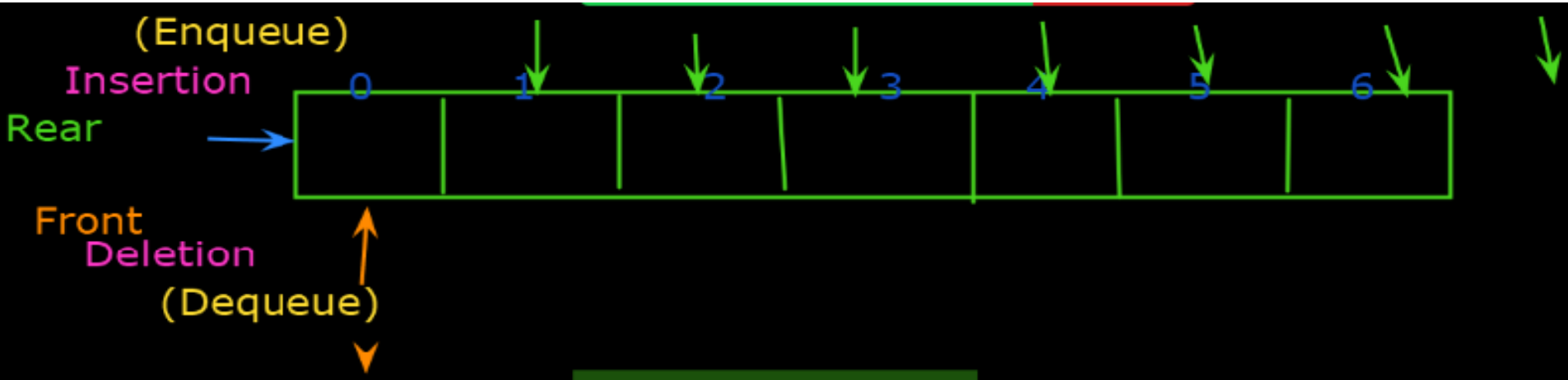
When we require to delete/remove an element to the Queue we perform Dequeue() operation.

Dequeue() operation is synonymous of deletion/removal in a data structure.



Agenda:

-Stack
-Queue



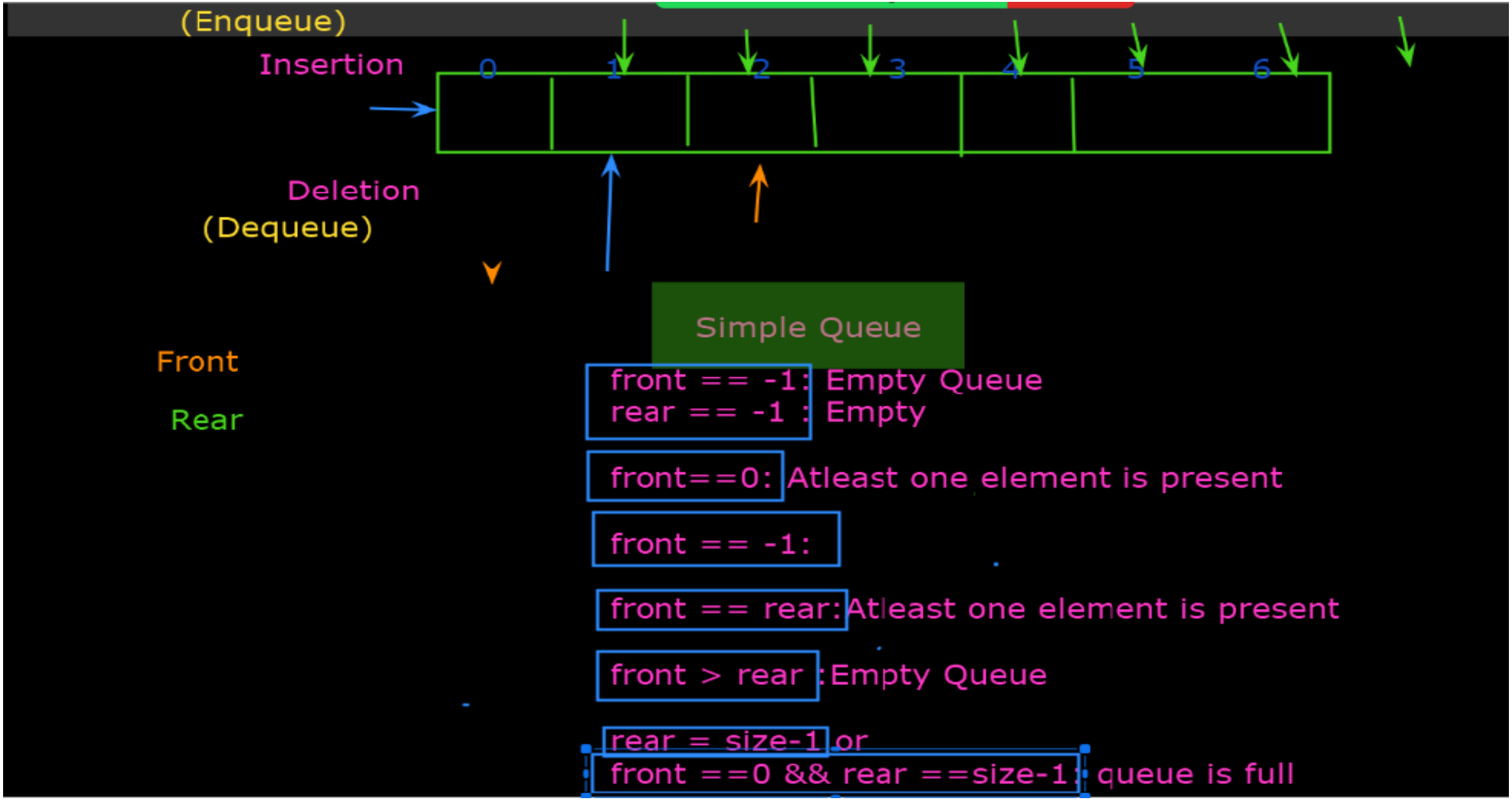
Queue:

Simple Queue

-data structure
-Queue follows the rule : FIFO (First In First Out)
-Insertion operation is called as Enqueue
-Deletion operation is called as Dequeue
-Useing 2 pointers:
    -Rear: Inserting an element in queue
    -Front: Deleting an element in queue
    -Rear = -1
    -Front = -1/ 0

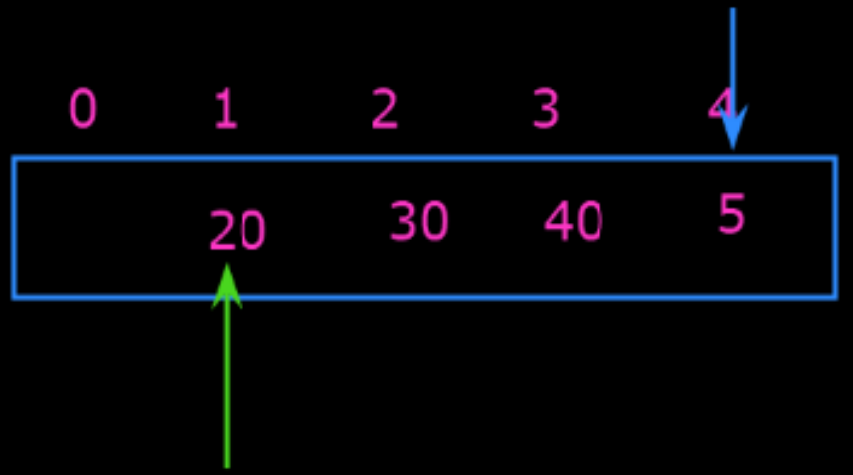
Operations on th Queue:

-Enqueue
-Dequeue


(Enqueue)
Insertion
0
1
2
3
4
5
6
Deletion
(Dequeue)
Simple Queue
Front
Rear
front == -1: Empty Queue
rear == -1 : Empty
front==0: Atleast one element is present
front == -1:
front == rear: Atleast one element is present
front > rear :Empty Queue
rear = size-1 or
front ==0 && rear ==size-1: queue is full

```
    {
        for(int i=front;i<=rear;i++)
        {
            System.out.println(Q[i]+" ");
        }
    }
    //print the values of rear & front

    public static void main(String args[])
    {
    }
}
```



Time complexity:

Enqueue: O(1)
Dequeue: O(1)

Simple Queue

A simple queue are the general queue that we use on perform insertion and deletion on FIFO basis i.e. the new element is inserted at the rear of the queue and an element is deleted from the front of the queue.

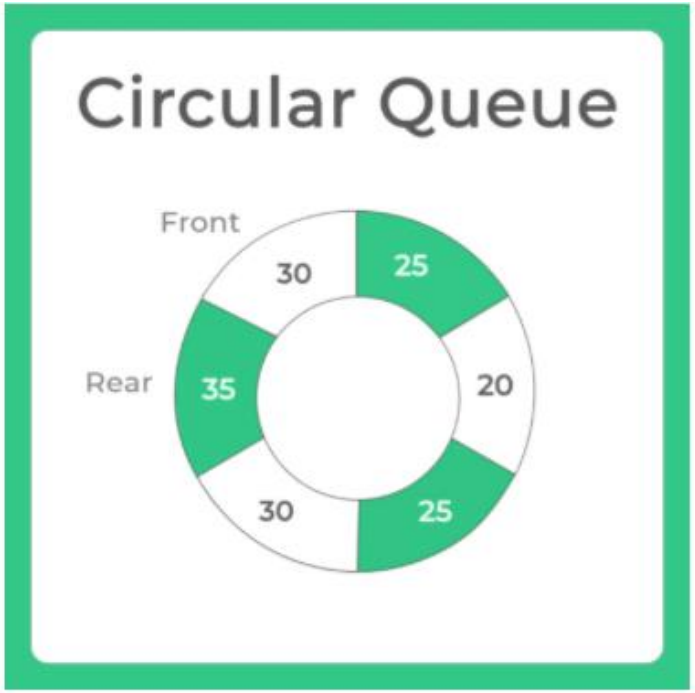
Applications of Simple Queue

The applications of simple queue are:-

- CPU scheduling
- Disk Scheduling
- Synchronization between two process.

Circular Queue

Circular queue is a type of queue in which all nodes are treated as circular such that the first node follows the last node. It is also called ring buffer. In this type of queue operations are performed on first in first out basis i.e the element that has inserted first will be one that will be deleted first.



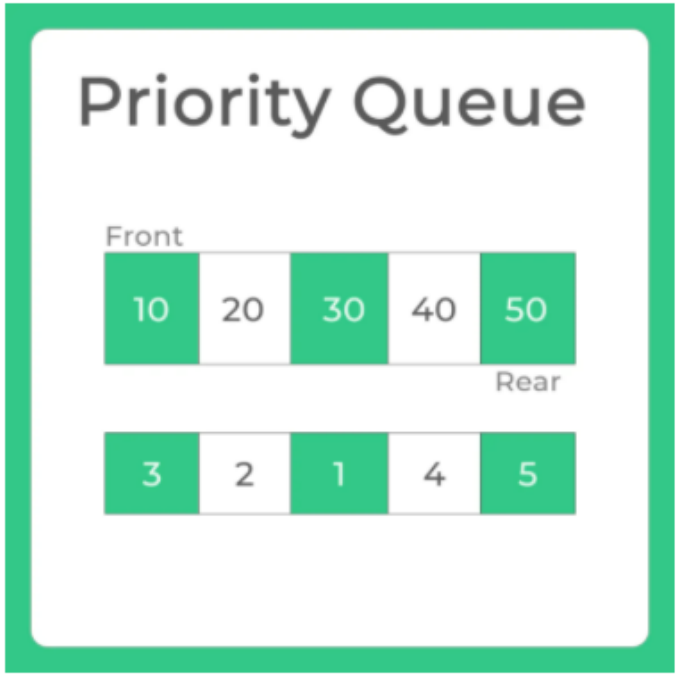
Applications of Circular Queue

The applications of circular queue are:-

- CPU scheduling
- Memory management
- Traffic Management

Priority Queue

Priority Queue is a special type of queue in which elements are treated according to their priority. Insertion of an element take place at the rear of the queue but the deletion or removal of an element take place according to the priority of the element. Element with the highest priority is removed first and element with the lowest priority is removed last.

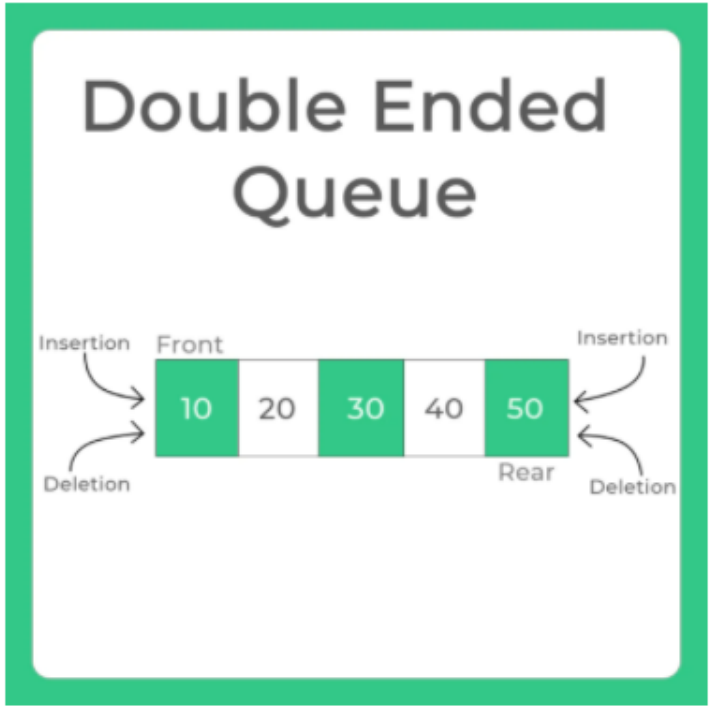


Applications of Priority Queue

The applications of priority queue are:-

- Dijkstra's shortest path algorithm
- Data compression in huffman codes.
- Load balancing and interrupt handling in operating system.
- Sorting heap.

Double Ended Queue



Double ended queue are also known as deque. In this type of queue insertion and deletion of an element can take place at both the ends. Further deque is divided into two types:-

- Input Restricted Deque :- In this, input is blocked at a single end but allows deletion at both the ends.
- Output Restricted Deque :- In this, output is blocked at a single end but allows insertion at both the ends.

Applications of Double Ended Queue

The applications of double ended queue are:-

- To execute undo and redo operation.
- For implementing stacks.
- Storing the history of web browsers.

Thanks