Chapter 2
Stack and Queue

DECLARATION OF ARRAYS

- An array must be **declared before being used.**
- Declaring an array means specifying the following:
  - **Data type**—the kind of values it can store, for example, int, char, float, double.
  - **Name**—to identify the array.
  - **Size**—the maximum number of values that the array can hold. Arrays are declared using the following syntax: type name[size]

Syntax:

```
type name[size];
```

Example:

```
int Marks[10];
```

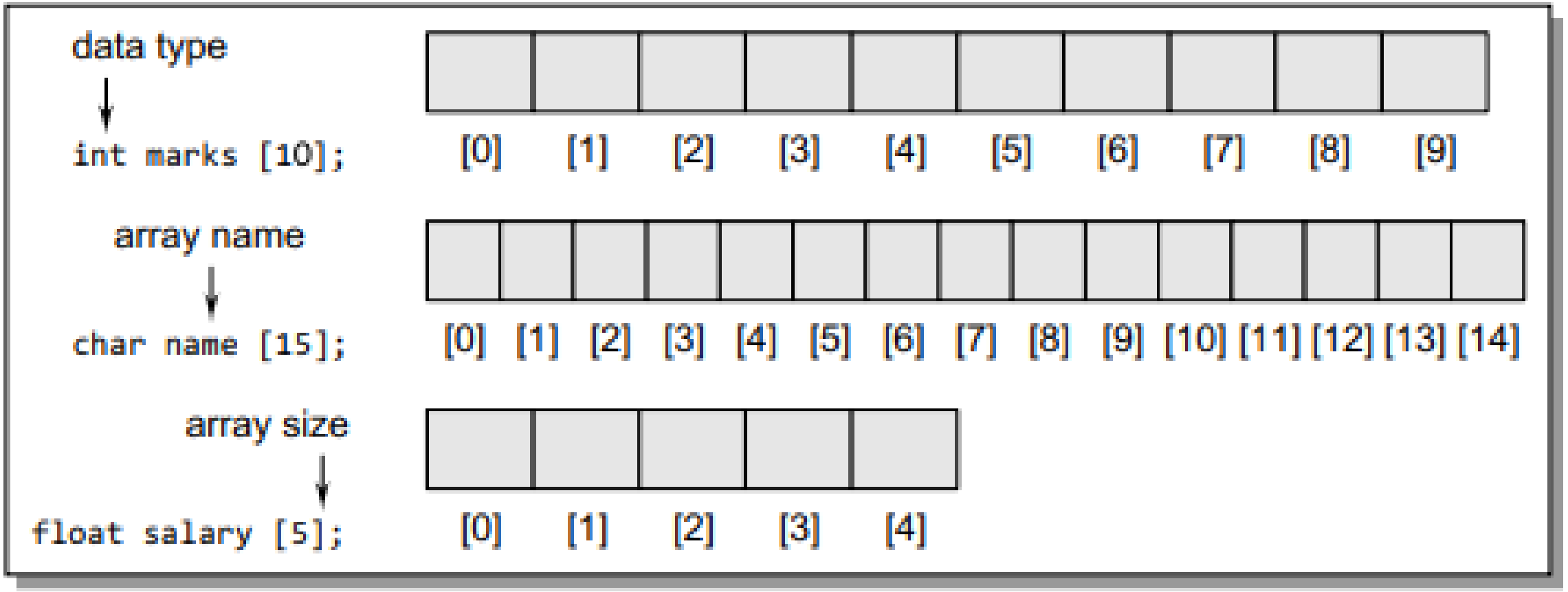


Figure 3.3 Declaring arrays of different data types and sizes

Accessing the elements of an array

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
        marks[i] = -1;
```

Figure 3.4 Code to initialize each element of the array to −1

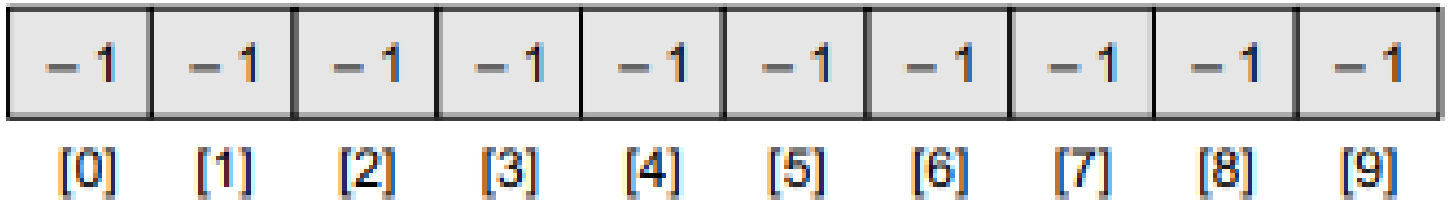


Figure 3.5 Array `marks` after executing the code given in Fig. 3.4

Calculating the Address of Array Elements

Example 3.1 Given an array `int marks[]={99,67,78,56,88,90,34,85}`, calculate the address of `marks[4]` if the `base address = 1000`.

Solution

99	67	78	56	**88**	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	**marks[4]**	marks[5]	marks[6]	marks[7]
1000	1002	1004	1006	**1008**	1010	1012	1014

We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

```
marks[4] = 1000 + 2(4 - 0)
         = 1000 + 2(4) = 1008
```

Calculating the Length of an Array

Example 3.2 Let `Age[5]` be an array of integers such that

`Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7`

Show the memory representation of the array and calculate its length.

Solution

The memory representation of the array `Age[5]` is given as below.

`Length = upper_bound - lower_bound + 1`

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length = 4 - 0 + 1 = 5`

Initializing Arrays during Declaration

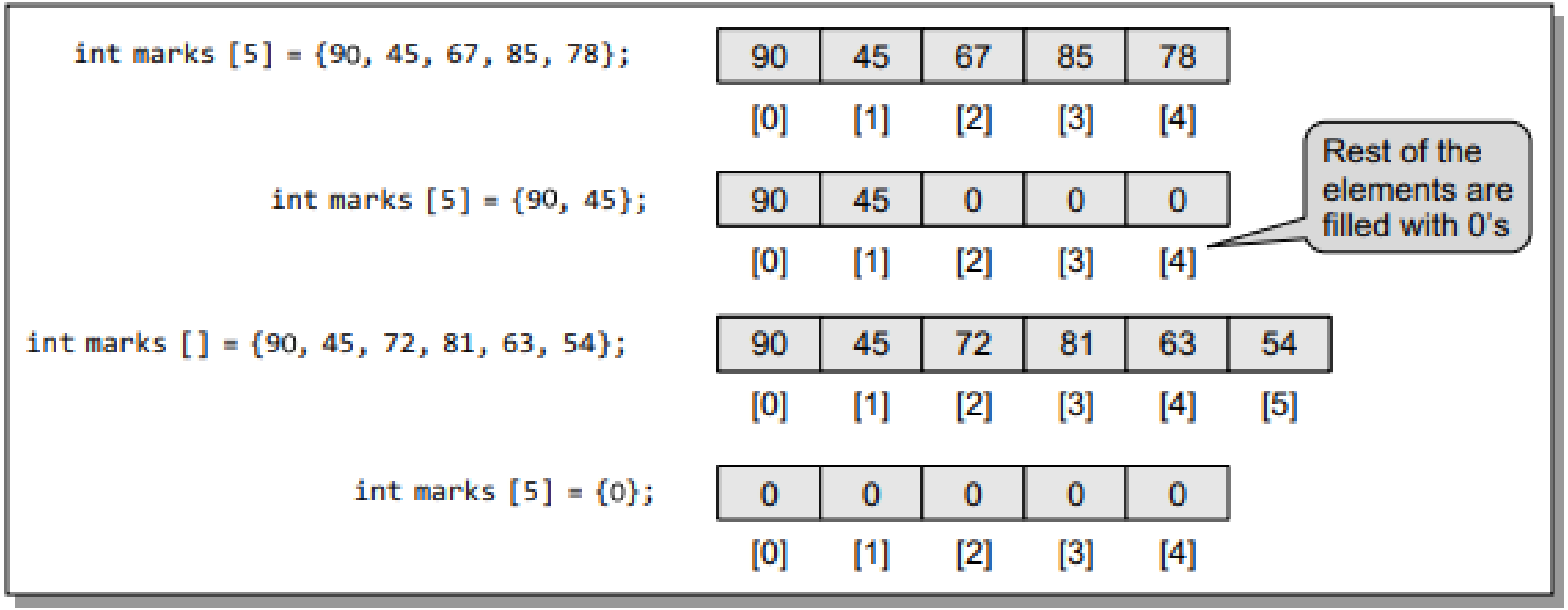


Figure 3.8 Initialization of array elements

OPERATIONS ON ARRAYS

- There are a number of operations that can be preformed on arrays. These operations include:
  1. Traversing an array
  2. Inserting an element in an array
  3. Searching an element in an array
  4. Deleting an element from an array
  5. Merging two arrays
  6. Sorting an array in ascending or descending order

Traversing an Array

Programming Examples

1. Write a program to read and display *n* numbers using an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, arr[20];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        for(i=0;i<n;i++)
        {
                printf("\n arr[%d] = ", i);
                scanf("%d",&arr[i]);
        }
        printf("\n The array elements are ");
        for(i=0;i<n;i++)
                printf("\t %d", arr[i]);
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The array elements are     1     2     3     4     5
```

- Write a program to find the mean of n numbers using arrays
- Write a program to find the mean of n numbers using arrays

Pointers

- C pointer is the derived data type that is used to **store the address of another variable** and can also be used to **access and manipulate** the variable's data stored at that location.

- The pointers are considered as **derived data types.**

Pointer Declaration

- **Syntax**
  - type *var-name;

- **Example of Valid Pointer Variable Declarations**
  - int *ip; /* pointer to an integer */
  - double *dp; /* pointer to a double */
  - float *fp; /* pointer to a float */
  - char *ch /* pointer to a character */

Pointer Initialization

- **Syntax**
  - pointer_variable = &variable;

- **Example**
  - int x = 10;
  - int *ptr = &x;

Referencing and Dereferencing Pointers

- **The & Operator** −
  - It is also known as the "Address-of operator".
  - It is used for Referencing which means taking the address of an existing variable (using **&**) to set a pointer variable.

- **The * Operator** −
  - It is also known as the "dereference operator".
  - **Dereferencing** a pointer is carried out using the *** operator** to get the value from the memory address that is pointed by the pointer.

Access and Manipulate Values using Pointer

```c
#include <stdio.h>

int main() {
  int x = 10;

  // Pointer declaration and initialization
  int * ptr = & x;

  // Printing the current value
  printf("Value of x = %d\n", * ptr);

  // Changing the value
  * ptr = 20;

  // Printing the updated value
  printf("Value of x = %d\n", * ptr);

  return 0;
}
```

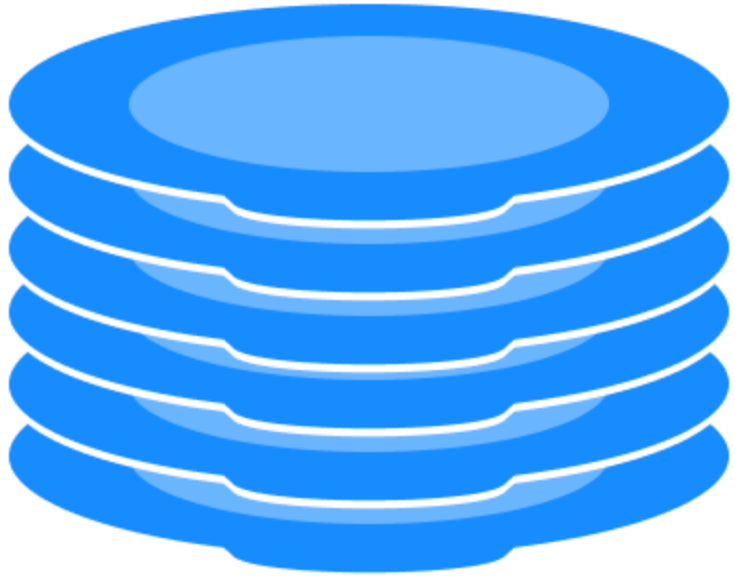
Value of x = 10
Value of x = 20

```c
#include <stdio.h>

int main() {
  int x = 10;
  float y = 1.3f;
  char z = 'p';

  // Pointer declaration and initialization
  int * ptr_x = & x;
  float * ptr_y = & y;
  char * ptr_z = & z;

  // Printing the values
  printf("Value of x = %d\n", * ptr_x);
  printf("Value of y = %f\n", * ptr_y);
  printf("Value of z = %c\n", * ptr_z);

  return 0;
}
```

Value of x = 10
Value of y = 1.300000
Value of z = p

Stack ADT

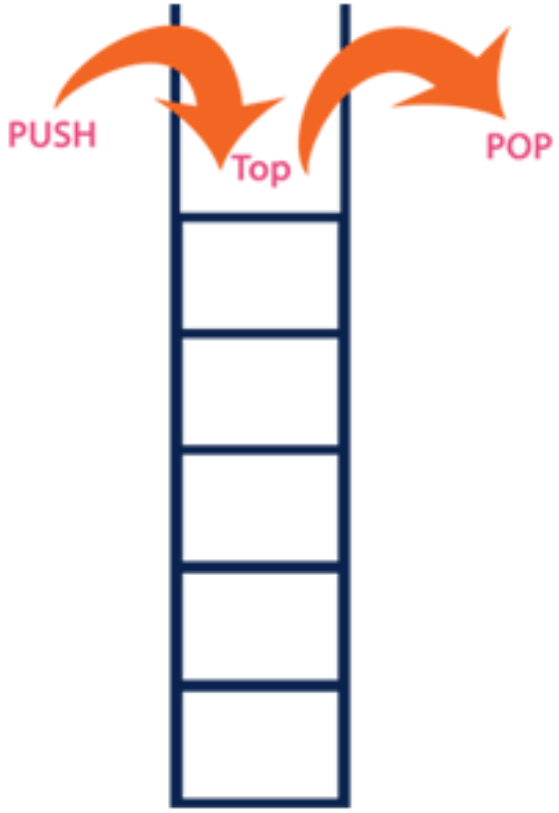
- **What is a Stack?**
  - Stack is a linear data structure in which the insertion and deletion operations are performed at only one end.
  - In a stack, adding and removing of elements are performed at a single position which is known as "**top**".
  - That means, a new element is added at top of the stack and an element is removed from the top of the stack.
  - In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



What is a Stack?

- In a stack, the insertion operation is performed using a function called "**push**" and deletion operation is performed using a function called "**pop**".

  In the figure, PUSH and POP operations are performed at a top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)



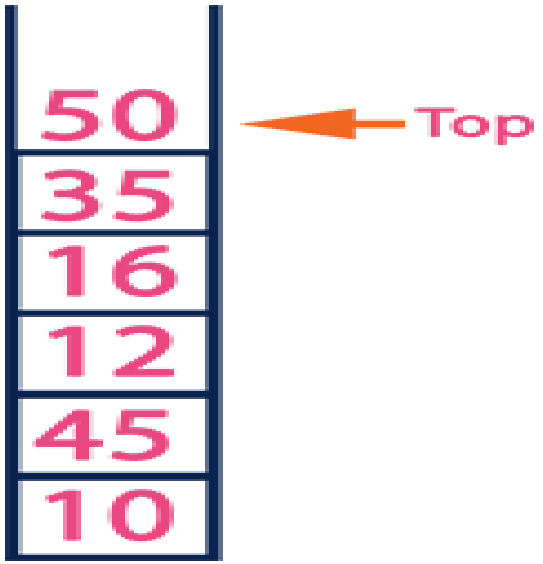
A stack data structure can be defined as

1. Stack is a linear data structure in which the operations are performed based on **LIFO principle**.

2. "A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Example

- If we want to create a stack by inserting 10,45,12,16,35 and 50.
- Then 10 becomes the bottom-most element and 50 is the topmost element.
- The last inserted element 50 is at Top of the stack as shown in the image below...



Stack data structure can be implemented in two ways.

They are as follows:

1. **Using Array**
2. **Using Linked List**

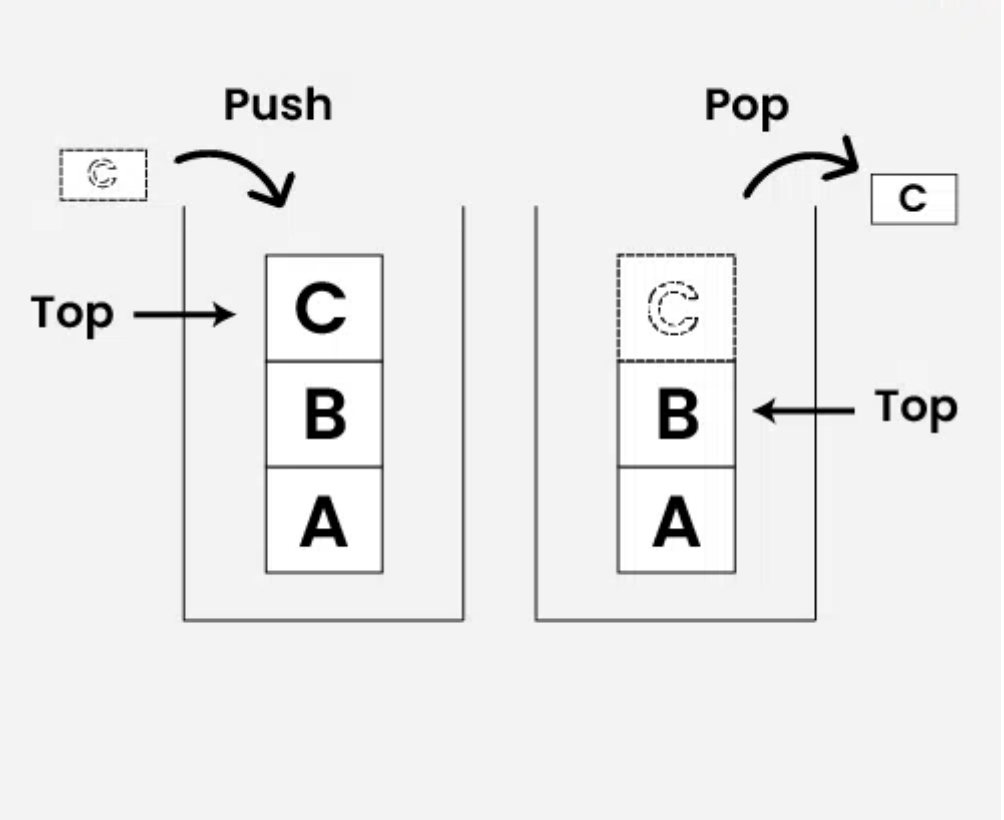
- When a stack is implemented using an **ARRAY**, that **stack can organize an only limited number of elements.**
- When a stack is implemented using a **LINKED LIST**, that **stack can organize an unlimited number of elements.**

Stack Using Array

- A stack data structure can be implemented using a **one-dimensional array.**
- But stack implemented using array **stores only a fixed number** of data values.
- Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**.
- Initially, the **top is set to -1**.
- Whenever we want to **insert** a value into the stack, **increment the top value by one** and then insert.
- Whenever we want to **delete** a value from the stack, then delete the top value and **decrement the top value by one**.

Operations on a Stack

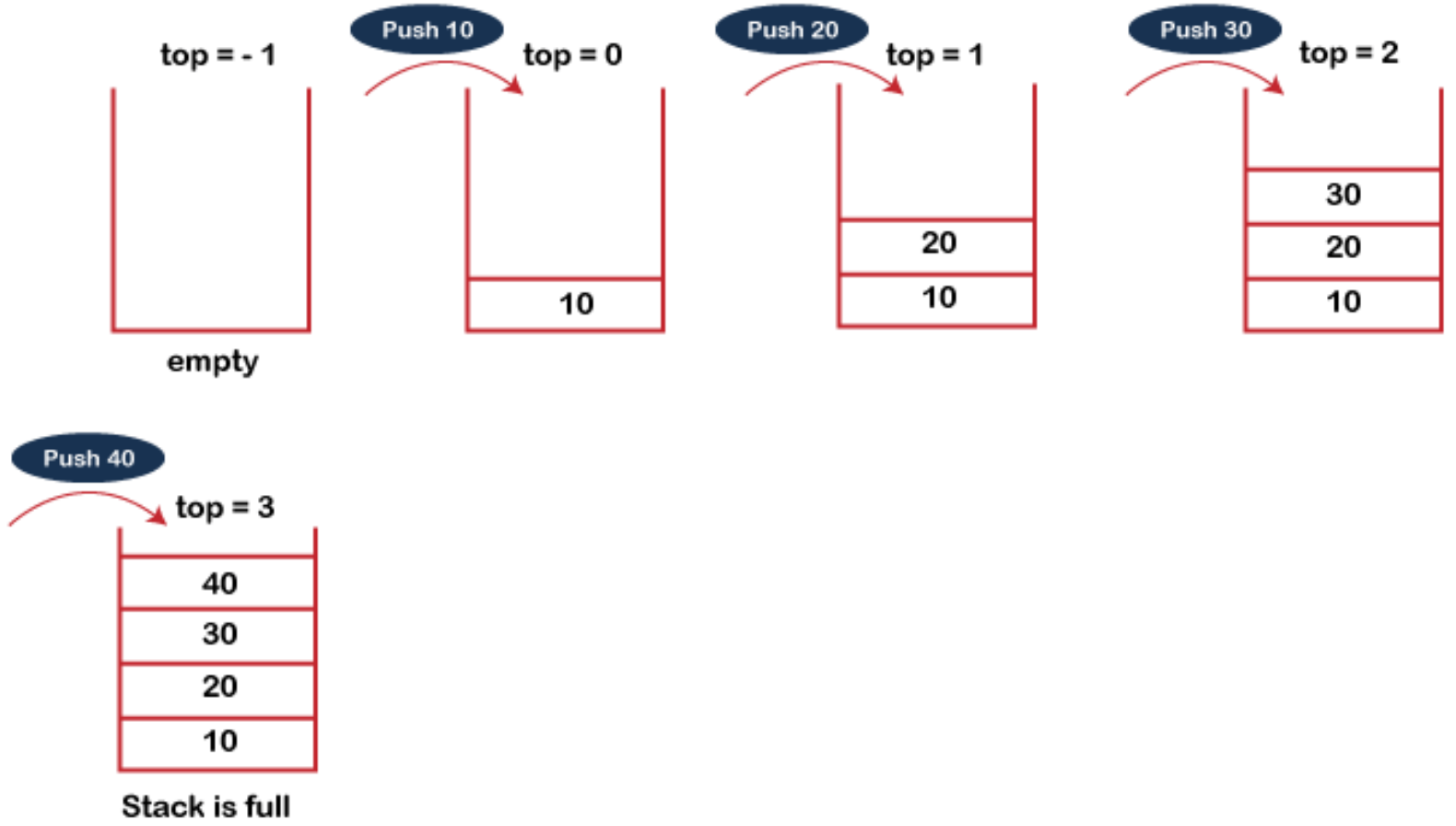
- The following operations are performed on the stack...
  1. **Push (To insert an element on to the stack)**
  2. **Pop (To delete an element from the stack)**
  3. **Display (To display elements of the stack)**



push(value) - Inserting value into the stack

- Adding an element into the top of the stack is referred to as push operation.
- Push operation involves following two steps.
  1. Increment the variable **Top** so that it can now refer to the next memory location.
  2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.
- **Stack is overflown** when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

push(value) - Inserting value into the stack



Algorithm:

- begin
- **if** top = n then stack full
- top = top + 1
- stack (top) : = item;
- end

```
void push (int val,int n) //n is size of the stack
{
   if (top == n )
   printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
   }
}
```

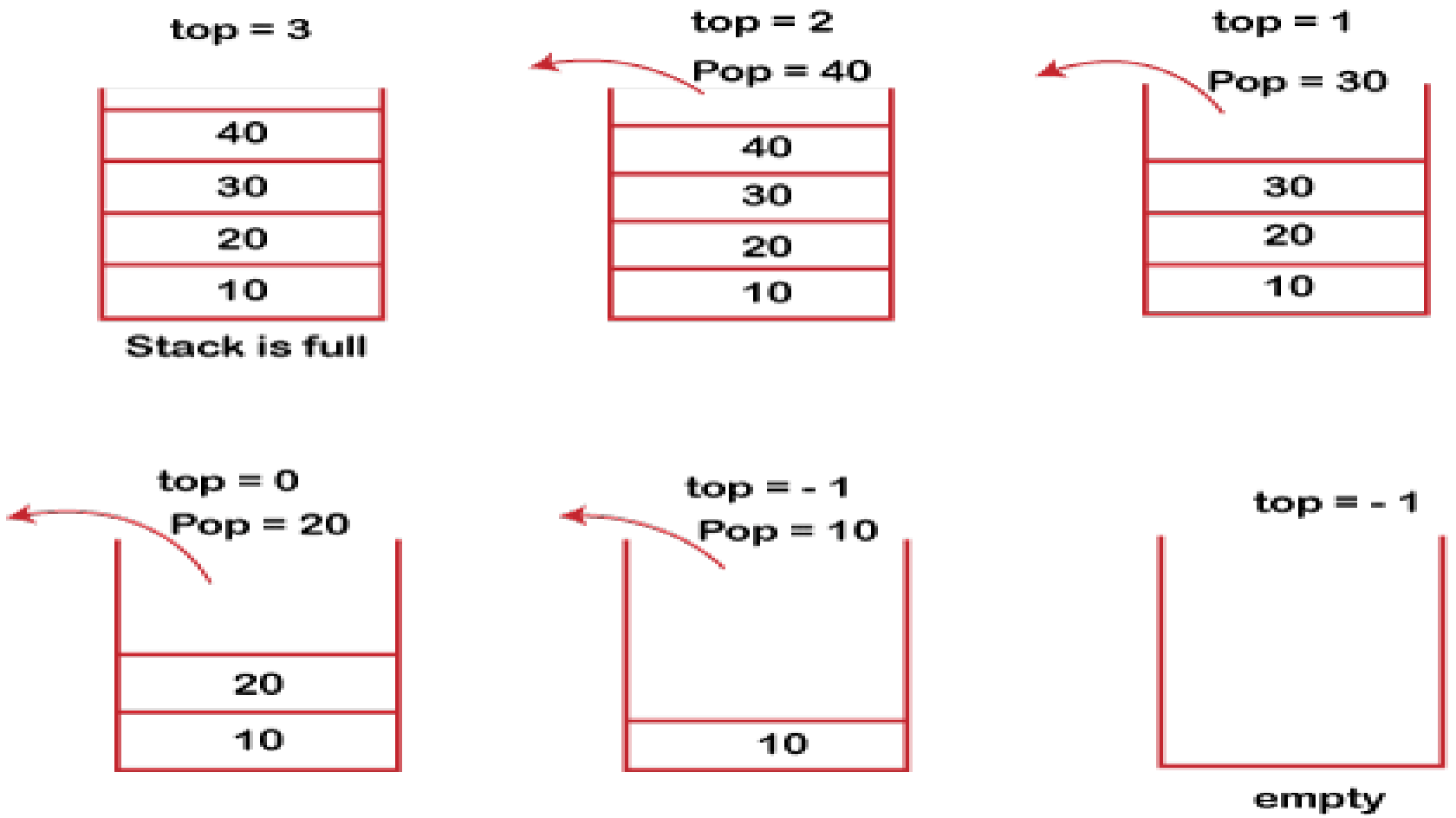
Pop Operation in Stack:
(Removes an item from the stack.)

- The items are popped in the reversed order in which they are pushed.
- If the stack is empty, then it is said to be an **Underflow condition.**

- Pop operation involves following three steps :

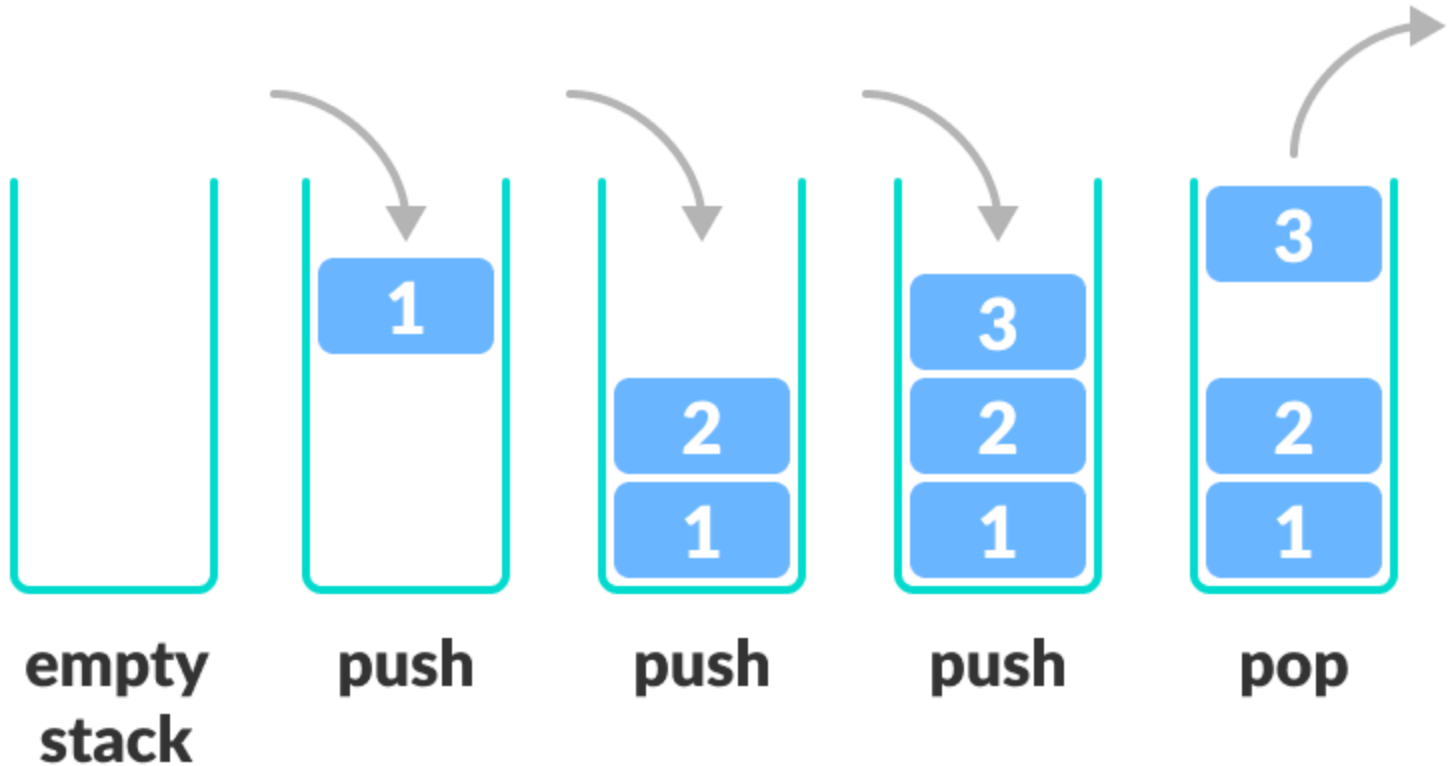
1. Before popping the element from the stack, we check if the stack is empty .
2. If the stack is empty (top == -1), then Stack Underflows and we cannot remove any element from the stack.
3. Otherwise, we store the value at top, decrement the value of top by 1 (top = top – 1) and return the stored top value.

Pop Operation in Stack:
(Removes an item from the stack.)



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
1
3
2
1
3
2
1
empty stack
push
push
push
pop

```c
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;

/* Check if the stack is empty */
int isempty()
{
    if(top == -1)
        return 1;
    else
        return 0;
}

/* Check if the stack is full */
int isfull()
{
    if(top == MAXSIZE)
        return 1;
    else
        return 0;
}
```

```c
/* Function to return the topmost element in the stack */
int peek()
{
    return stack[top];
}

/* Function to delete from the stack */
int pop()
{
    int data;
    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    } else {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}

/* Function to insert into the stack */
int push(int data)
{
    if(!isfull()) {
        top = top + 1;
        stack[top] = data;
    } else {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

```c
/* Main function */
int main()
{
    push(44);
    push(10);
    push(62);
    push(123);
    push(15);
    printf("Element at top of the stack: %d\n" ,peek());
    printf("Elements: \n");

    // print stack data
    while(!isempty()) {
        int data = pop();
        printf("%d\n",data);
    }
    printf("Stack full: %s\n" , isfull()?"true":"false");
    printf("Stack empty: %s\n" , isempty()?"true":"false");
    return 0;
}
```

```c
#include <stdio.h>
#define MAXSIZE 5

struct stack
{
    int stk[MAXSIZE];
    int top;
};
typedef struct stack STACK;
STACK s;

void push(void);
int  pop(void);
void display(void);

void main ()
{
    int choice;
    int option = 1;
    s.top = -1;

    printf ("STACK OPERATION\n");
    while (option)
    {
        printf ("----------------------------------------\n");
        printf ("      1    -->    PUSH              \n");
        printf ("      2    -->    POP               \n");
        printf ("      3    -->    DISPLAY           \n");
        printf ("      4    -->    EXIT              \n");
        printf ("----------------------------------------\n");

        printf ("Enter your choice\n");
        scanf    ("%d", &choice);
        switch (choice)
        {
        case 1:
            push();
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            return;

        fflush (stdin);
        printf ("Do you want to continue(Type 0 or 1)?\n");
        scanf    ("%d", &option);
    }
}
/*  Function to add an element to the stack */
void push ()
{
    int num;
    if (s.top == (MAXSIZE - 1))
    {
        printf ("Stack is Full\n");
        return;
    }
    else
    {
        printf ("Enter the element to be pushed\n");
        scanf ("%d", &num);
        s.top = s.top + 1;
        s.stk[s.top] = num;
    }
    return;
}

/*  Function to delete an element from the stack */
int pop ()
{
    int num;
    if (s.top == - 1)
    {
        printf ("Stack is Empty\n");
        return (s.top);
    }
    else
    {
        num = s.stk[s.top];
        printf ("poped element is = %dn", s.stk[s.top]);
        s.top = s.top - 1;
    }
    return(num);
}
/*  Function to display the status of the stack */
void display ()
{
    int i;
    if (s.top == -1)
    {

    }
    else
    {
        printf ("\n The status of the stack is \n");
        for (i = s.top; i >= 0; i--)
        {
            printf ("%d\n", s.stk[i]);
        }
    }
    printf ("\n");
}
```

Program with Structure

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

- The parenthesis is represented by the brackets shown below:

  ( )

  Where, (→ Opening bracket

) → Closing bracket

- These parentheses are used to represent the mathematical representation.
- The balanced parenthesis means that when the opening parenthesis is equal to the closing parenthesis, then it is a balanced parenthesis.

- **Example 1:** **(2+5) * 4**
- In the above expression, there is one opening and one closing parenthesis means that both opening and closing brackets are equal; therefore, the above expression is a balanced parenthesis.

- **Example 2:** **2 * ((4/2) + 5)**
- The above expression has two opening and two closing brackets which means that the above expression is a balanced parenthesis.

- **Example 3:** **2 * ((4/2) + 5**
- The above expression has two opening brackets and one closing bracket, which means that both opening and closing brackets are not equal; therefore, the above expression is unbalanced.

Algorithm to check balanced parenthesis

Step 1: Set x equal to 0.

Step 2: Scan the expression from left to right.

For each opening bracket "(", increment x by 1.

For each closing bracket ")", decrement x by 1.

This step will continue scanning until x<0.

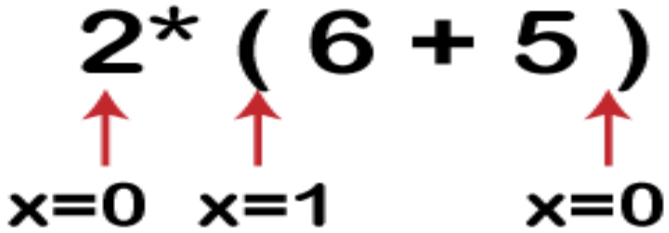
Step 3: If x is equal to 0, then

"Expression is balanced."

` Else

"Expression is unbalanced."

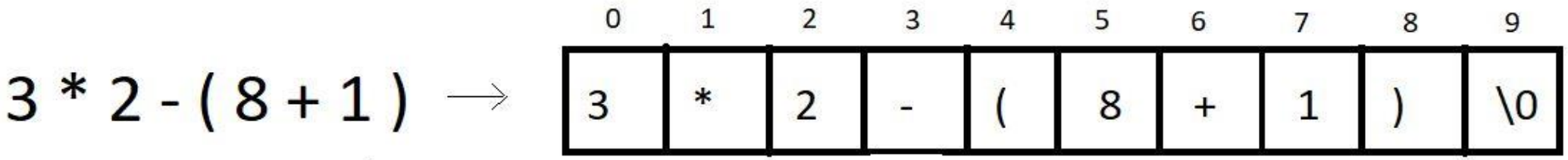
Suppose expression is 2 * (6 + 5)



- Solution: First, the x variable is initialized by 0. The scanning starts from the variable '2', when it encounters '(' then the 'x' variable gets incremented by 1 and when the x reaches to the last symbol of the expression, i.e., ')' then the 'x' variable gets decremented by 1 and it's final value becomes 0. We have learnt in the above algorithm that if x is equal to 0 means the expression is balanced; therefore, the above expression is a balanced expression.

Use of stacks to match these parentheses. Let's see how:

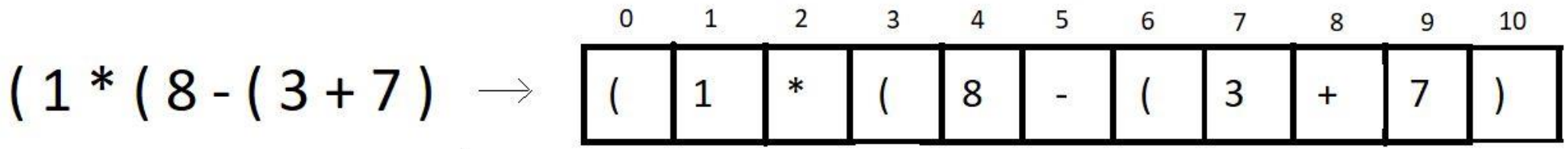
1. Assume the expression given to you as a character array.

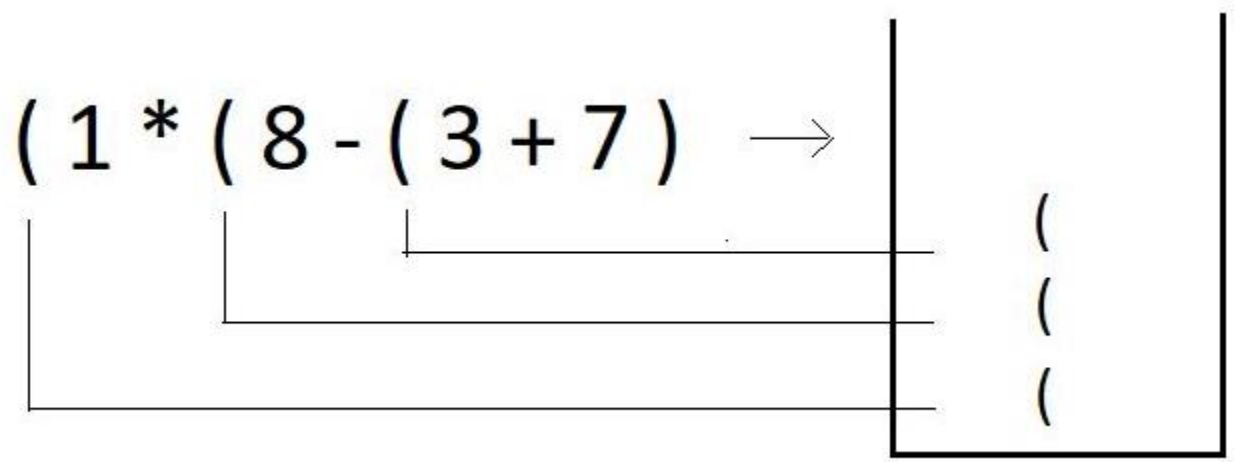


2. Iterate through the character array and ignore everything you find other than the opening and the closing parenthesis. Every time you find an opening parenthesis, push it inside a character stack. And every time you find a closing parenthesis, pop from the stack, in which you pushed the opening bracket.

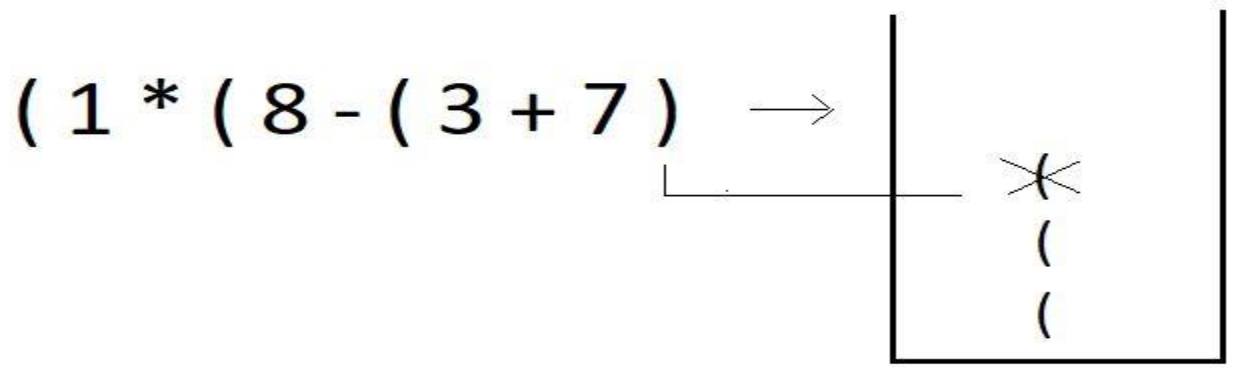
checking if the above expression has balanced parentheses or not.

- **Step 1:** Iterate through the char array, and push the opening brackets at positions 0, 3, 6 inside the stack.





Step 2: Try popping an opening bracket from the stack when you encounter a closing bracket in the expression.



Step 3: Since we reached the EOE and there are still two parentheses left in the stack, we declare this expression of parentheses **unbalanced**.

STACKS – Infix to Postfix Conversion

- https://mucertification.com/topic/2-1-f-stacks-infix-to-postfix-conversion/

Precedence of Operators

precedence of operators determines the order in which operators are evaluated in expressions.

Here is a typical precedence order for common operators in many programming languages, from highest to lowest:

1. **Parentheses**: ()
2. **Unary operators**: +, **-** (unary plus and minus), ! (logical NOT)
3. **Exponents**: ^
4. **Multiplication and Division**: *, /, %
5. **Addition and Subtraction**: +, **-** (binary)
6. **Relational operators**: <, <=, >, >=
7. **Equality operators**: ==, !=
8. **Logical AND**: &&
9. **Logical OR**: ||

Associativity

- **Left to Right**: For most binary operators like
  - +, -, *, /, <, <=, >, >=, ==, !=, &&, ||
- **Right to Left**: For unary operators and exponentiation, such as
  - +, - (unary), ! , ^

Examples

- Unary minus has higher precedence than multiplication:
  - -A * B is equivalent to (-A) * B
- Multiplication and division have the same precedence and are evaluated left to right:
  - A * B / C is equivalent to (A * B) / C
- Addition and subtraction have the same precedence and are evaluated left to right:
  - A + B - C is equivalent to (A + B) - C

Rules for the conversion from infix to postfix expression

- Print the operand as they arrive.
- If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
- If the incoming symbol is '(', push it on to the stack.
- If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
- If the incoming symbol has **higher precedence** than the top of the stack, **push it on the stack**.
- If the incoming symbol has **lower precedence** than the top of the stack, **pop and print the top of the stack.** Then **test** the incoming operator **against the new top of the stack**.
- If the incoming operator has the **same precedence** with the top of the stack then **use the associativity rules.** If the associativity is from **left to right then pop** and print the top of the stack then push the incoming operator. If the associativity is from **right to left then push** the incoming operator.
- At the end of the expression, pop and print all the operators of the stack.

infix to postfix expression

Example 1

Infix Expression: A + B

Steps:

1. Read A and output it.
2. Read + and push it to the stack.
3. Read B and output it.
4. Pop + from the stack and output it.

Postfix Expression: AB+

Example 2

Infix Expression: A + B * C

Steps:

1. Read A and output it.
2. Read + and push it to the stack.
3. Read B and output it.
4. Read * and push it to the stack (since * has higher precedence than +).
5. Read C and output it.
6. Pop * from the stack and output it.
7. Pop + from the stack and output it.

Postfix Expression: ABC*+

Example 3

Infix Expression: (A + B) * C

Steps:

1. Read (and push it to the stack.
2. Read A and output it.
3. Read + and push it to the stack.
4. Read B and output it.
5. Read) and pop + from the stack and output it. Discard (.
6. Read * and push it to the stack.
7. Read C and output it.
8. Pop * from the stack and output it.

Postfix Expression: AB+C*

Example 4

Infix Expression: A + B * C + D

Steps:

1. Read A and output it.
2. Read + and push it to the stack.
3. Read B and output it.
4. Read * and push it to the stack.
5. Read C and output it.
6. Pop * from the stack and output it.
7. Read + and push it to the stack (pop + from the stack and output it).
8. Read D and output it.
9. Pop + from the stack and output it.

Postfix Expression: ABC*+D+

Example 5

Infix Expression: A * (B + C) * D

Steps:

1. Read A and output it.
2. Read * and push it to the stack.
3. Read (and push it to the stack.
4. Read B and output it.
5. Read + and push it to the stack.
6. Read C and output it.
7. Read) and pop + from the stack and output it. Discard (.
8. Pop * from the stack and output it.
9. Read * and push it to the stack.
10. Read D and output it.
11. Pop * from the stack and output it.

Postfix Expression: ABC+*D*

Infix Expression: A + B * (C ^ D - E)

Steps:

1.Read A and output it.
2.Read + and push it to the stack.
3.Read B and output it.
4.Read * and push it to the stack.
5.Read (and push it to the stack.
6.Read C and output it.
7.Read ^ and push it to the stack.
8.Read D and output it.
9.Read - and push it to the stack (pop ^ from the stack and output it).
10.Read E and output it.
11.Read) and pop - from the stack and output it. Discard (.
12.Pop * from the stack and output it.
13.Pop + from the stack and output it.

Postfix Expression: ABCD^E-*+

Infix Expression: A * (B + C * D) - E

Steps:

1.Read A and output it.
2.Read * and push it to the stack.
3.Read (and push it to the stack.
4.Read B and output it.
5.Read + and push it to the stack.
6.Read C and output it.
7.Read * and push it to the stack.
8.Read D and output it.
9.Pop * from the stack and output it.
10.Pop + from the stack and output it.
11.Read) and discard it.
12.Pop * from the stack and output it.
13.Read - and push it to the stack.
14.Read E and output it.
15.Pop - from the stack and output it.

Postfix Expression: ABCD*+*E-

Example 8

Infix Expression: (A + B) * (C - D) / E

Steps:

1.Read (and push it to the stack.
2.Read A and output it.
3.Read + and push it to the stack.
4.Read B and output it.
5.Read) and pop + from the stack and output it. Discard (.
6.Read * and push it to the stack.
7.Read (and push it to the stack.
8.Read C and output it.
9.Read - and push it to the stack.
10.Read D and output it.
11.Read) and pop - from the stack and output it. Discard (.
12.Pop * from the stack and output it.
13.Read / and push it to the stack.
14.Read E and output it.
15.Pop / from the stack and output it.

Postfix Expression: AB+CD-*E/

Infix Expression: A + (B * C - (D / E ^ F) * G) * H

Steps:

1.Read A and output it.
2.Read + and push it to the stack.
3.Read (and push it to the stack.
4.Read B and output it.
5.Read * and push it to the stack.
6.Read C and output it.
7.Read - and push it to the stack (pop * from the stack and output it).
8.Read (and push it to the stack.
9.Read D and output it.
10.Read / and push it to the stack.
11.Read E and output it.
12.Read ^ and push it to the stack.
13.Read F and output it.
14.Pop ^ from the stack and output it.
15.Pop / from the stack and output it.
16.Read) and discard it.
17.Read * and push it to the stack.
18.Read G and output it.
19.Pop * from the stack and output it.
20.Pop - from the stack and output it.
21.Pop * from the stack and output it.
22.Read H and output it.
23.Pop * from the stack and output it.
24.Pop + from the stack and output it.

Postfix Expression: ABC*DEF^/G*-H*+

Infix Expression: A * B + C * D - E / F

Steps:

1.Read A and output it.
2.Read * and push it to the stack.
3.Read B and output it.
4.Pop * from the stack and output it.
5.Read + and push it to the stack.
6.Read C and output it.
7.Read * and push it to the stack.
8.Read D and output it.
9.Pop * from the stack and output it.
10.Pop + from the stack and output it.
11.Read - and push it to the stack.
12.Read E and output it.
13.Read / and push it to the stack.
14.Read F and output it.
15.Pop / from the stack and output it.
16.Pop - from the stack and output it.

Postfix Expression: AB*CD*+EF/-

- https://prepinsta.com/data-structures/infix-to-postfix-conversion/

Evaluating a postfix expression:

Take the above converted postfix notation. First let us get the result from the infix notation:

3 + 8 - 9 / 8

= 3 + 8 - 1.125 = 11 - 1.125 = 9.875

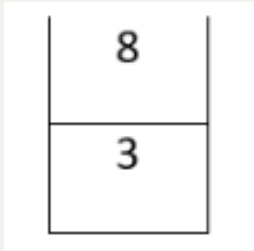
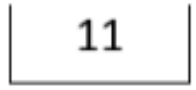
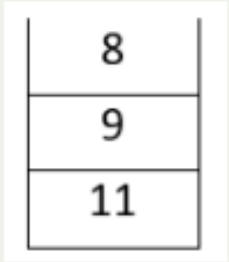
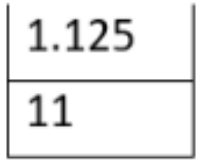
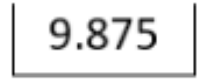
Now, let us take the postfix expression we got from the previous conversion:

3 8 + 9 8 / -

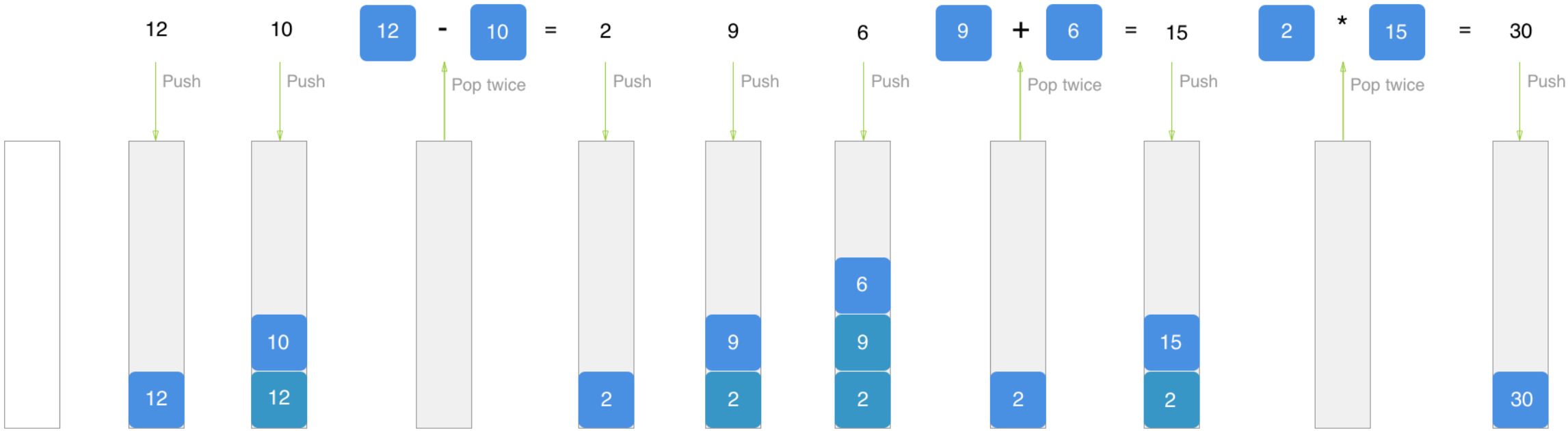
- Read the expression from left to right, initialize a stack with the same length as the expression.
- If the element encountered is an operand, push it into the stack.
- If the element encountered is an operator, pop two operands a and b from the stack, apply the operator (b operator a) and push the result back into the stack.

In the above example: **3 8 + 9 8 / -**

4. In the above example: **3 8 + 9 8 / -**

3 8	
+	3 + 8 = 11 
9 8	
/	9/8 = 1.125 
-	11 – 1.125 = 9.875 

Example : 12 10 - 9 6 + *



- ***Input**: str = “2 3 1 * + 9 -“*
  ***Output**: -4*
- ***Input**: str = “100 200 + 2 / 5 * 7 +”*
  ***Output**: 757*

Recursion

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.
- Every recursive solution has two major cases. They are
  - Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - Recursive case, in which first the problem at hand is divided into simpler sub-parts.
- Second the function calls itself but with sub-parts of the problem obtained in the first step.
- Third, the result is obtained by combining the solutions of simpler sub-parts.
- Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems.
- In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

Queues

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.

Example of Queues

- People **moving on an escalator**. The people who got on the escalator first will be the first one to step out of it.
- People **waiting for a bus**. The first person standing in the line will be the first one to get into the bus.
- People **standing outside the ticketing window** of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- L**uggage kept on conveyor belts**. The bag which was placed first will be the first to come out at the other end.
- **Cars lined at a toll bridge**. The first car to reach the bridge will be the first to leave.

Operations on Queues

- In Fig. 8.1, FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.

- The queue after addition would be as shown in Fig. 8.2.

Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.

- If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue. The queue after deletion will be as shown in

Fig. 8.3. Here, FRONT = 1 and REAR = 6.

Figure 8.1 Queue

Figure 8.2 Queue after insertion of a new element

Figure 8.3 Queue after deletion of an element

Algorithm to insert an element in a queue

Step 1: IF REAR = MAX-1
Write OVERFLOW
Goto step 4
[END OF IF]
Step 2: IF FRONT=-1 and REAR=-1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR+1
[END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT

Algorithm to delete an element from a queue

Step 1: IF FRONT = -1 OR FRONT > REAR

Write UNDERFLOW

ELSE

SET VAL = QUEUE[FRONT]

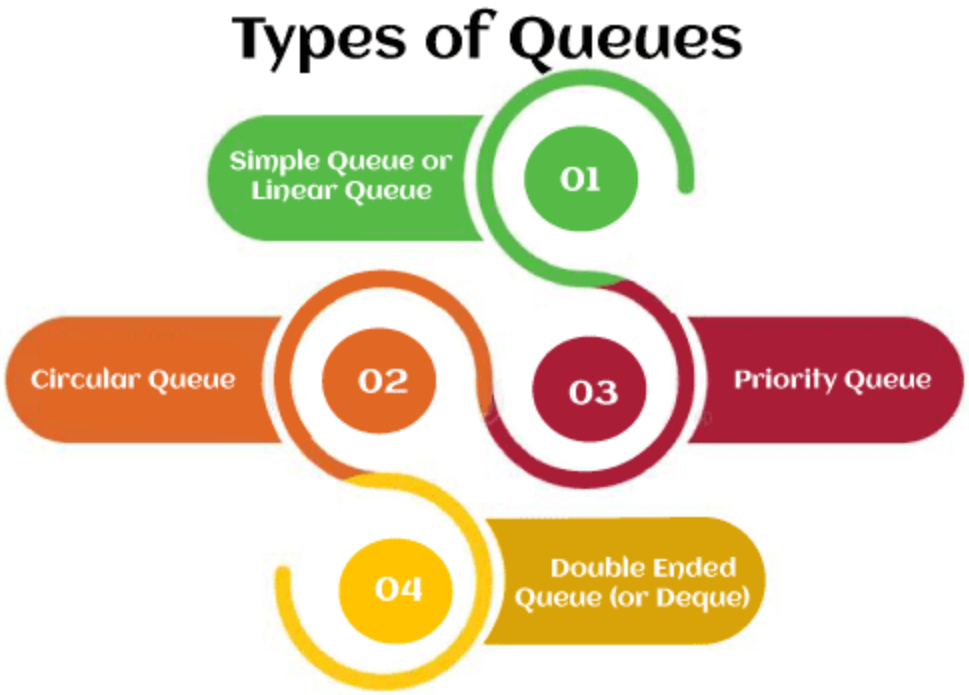
SET FRONT = FRONT+1

[END OF IF]

Step 2: EXIT

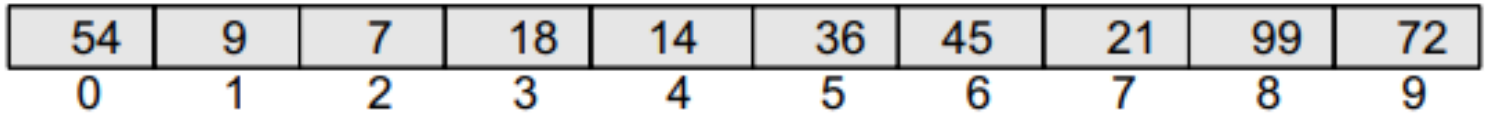
TYPES OF QUEUES

- A queue data structure can be classified into the following types:
  1. Simple Queue
  2. Circular Queue
  3. Priority Queue
  4. Deque

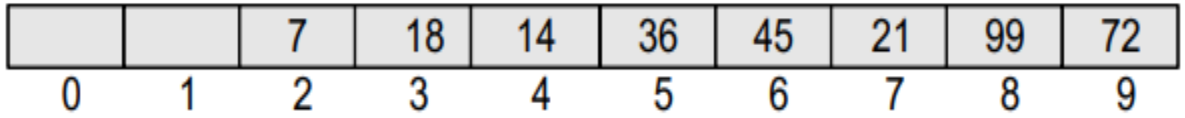


Linear Queues

- In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.
- Look at the queue shown in Fig.



- Figure Linear queue Here, FRONT = 0 and REAR = 9.
- Now, if you want to insert another value, it will not be possible because the queue is completely full.
- There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made.
- The queue will then be given as shown in Fig.



- Figure of Queue after two successive deletions Here, front = 2 and REAR = 9. Suppose we want to insert a new element in the queue shown in Fig.
- Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true. This is a major drawback of a linear queue

Circular queue

- In Circular Queue, all the nodes are represented as circular.
- It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- It is also known as Ring Buffer, as all the ends are connected to another end.
- The representation of circular queue is shown in the below image.

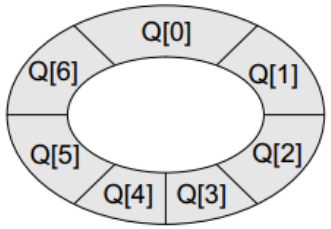


Figure 8.15 Circular queue

- The drawback that occurs in a linear queue is overcome by using the circular queue.
- If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.
- The main advantage of using the circular queue is better memory utilization.

Applications of Circular Queue

- **The circular Queue can be used in the following scenarios:**
  - **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
  - **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
  - **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every j interval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Operations on Circular Queue

- The following are the operations that can be performed on a circular queue:
  - **Front:** It is used to get the front element from the Queue.
  - **Rear:** It is used to get the rear element from the Queue.
  - **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
  - **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Circular Queues insertion

- For insertion, we now have to check for the following three conditions:
  - **If front = 0 and rear = MAX – 1,** then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.
  - **If rear != MAX – 1,** then rear will be incremented and the value will be inserted s illustrated in Fig. 8.17.
  - **If front != 0 and rear = MAX – 1,** then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown in Fig. 8.18.

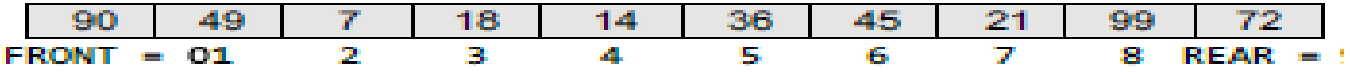


Figure 8.16 Full queue

Figure 8.17 Queue with vacant locations

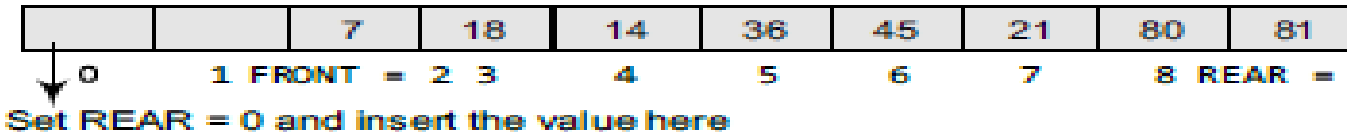


Figure 8.18 Inserting an element in a circular queue

Inserting an element in a circular queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW”
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Figure 8.19 Algorithm to insert an element in a circular queue

Delete an element

- Look at Fig. 8.20. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and front = rear, then after deleting the element at the front he queue becomes empty and so front and rear are set to –1. This is illustrated in Fig. 8.21.
- If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0.

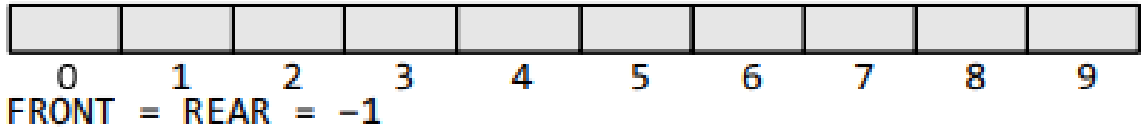


Figure 8.20 Empty queue

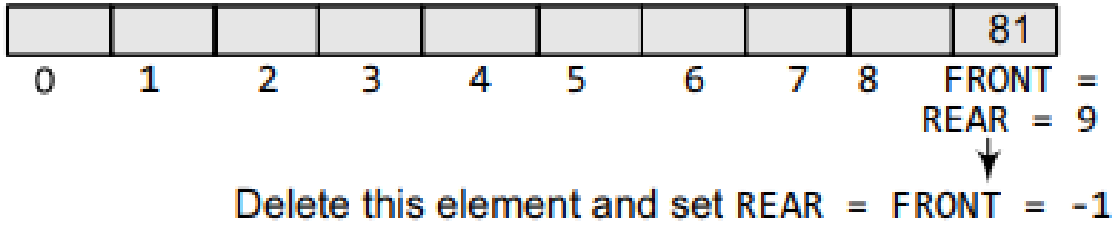


Figure 8.21 Queue with a single element

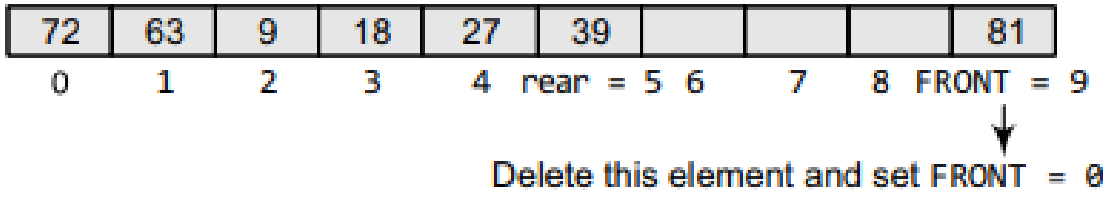


Figure 8.22 Queue where FRONT = MAX–1 before deletion

Algorithm to delete an element from a circular queue

Let us look at Fig. 8.23 which shows the algorithm to delete an element from a circular queue.

In Step 1, we check for the underflow condition.

In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL.

In Step 3, we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue.

The value of FRONT is then updated based on the outcome of these checks.

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
        ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

Figure 8.23 Algorithm to delete an element from a circular queue

```c
// Circular Queue implementation in C

#include <stdio.h>

#define SIZE 5

int items[SIZE];
int front = -1, rear = -1;

// Check if the queue is full
int isFull() {
  if ((front == rear + 1) || (front == 0 && rear == SIZE - 1)) return 1;
  return 0;
}

// Check if the queue is empty
int isEmpty() {
  if (front == -1) return 1;
  return 0;
}
```

```c
// Adding an element
void enQueue(int element) {
  if (isFull())
    printf("\n Queue is full!! \n");
  else {
    if (front == -1) front = 0;
    rear = (rear + 1) % SIZE;
    items[rear] = element;
    printf("\n Inserted -> %d", element);
  }
}
```

```c
// Removing an element
int deQueue() {
  int element;
  if (isEmpty()) {
    printf("\n Queue is empty !! \n");
    return (-1);
  } else {
    element = items[front];
    if (front == rear) {
      front = -1;
      rear = -1;
    }
    // Q has only one element, so we reset the
    // queue after dequeing it. ?
    else {
      front = (front + 1) % SIZE;
    }
    printf("\n Deleted element -> %d \n", element);
    return (element);
  }
}
```

```c
// Display the queue
void display() {
  int i;
  if (isEmpty())
    printf(" \n Empty Queue\n");
  else {
    printf("\n Front -> %d ", front);
    printf("\n Items -> ");
    for (i = front; i != rear; i = (i + 1) % SIZE) {
      printf("%d ", items[i]);
    }
    printf("%d ", items[i]);
    printf("\n Rear -> %d \n", rear);
  }
}
```

```c
int main() {
  // Fails because front = -1
  deQueue();

  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);

  // Fails to enqueue because front == 0 && rear == SIZE - 1
  enQueue(6);

  display();
  deQueue();

  display();

  enQueue(7);
  display();

  // Fails to enqueue because front == rear + 1
  enQueue(8);

  return 0;
}
```

Deques

- Deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end.
- The deque stands for Double Ended Queue.
- Deque is a linear data structure where the insertion and deletion operations are performed from both ends.
- We can say that deque is a generalized version of the queue.
- Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule.
- The representation of a deque is given as follows -

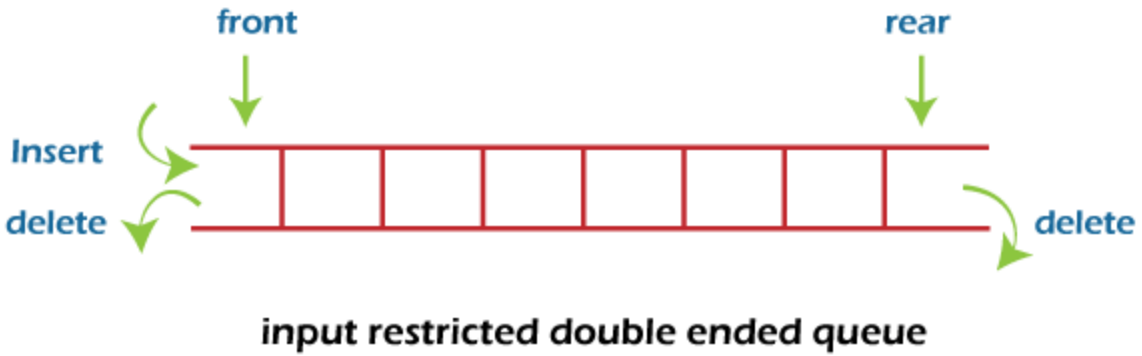
Representation of deque

Types of deque

- There are two types of deque -
  - Input restricted queue
  - Output restricted queue

Input restricted Queue

- In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

- In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

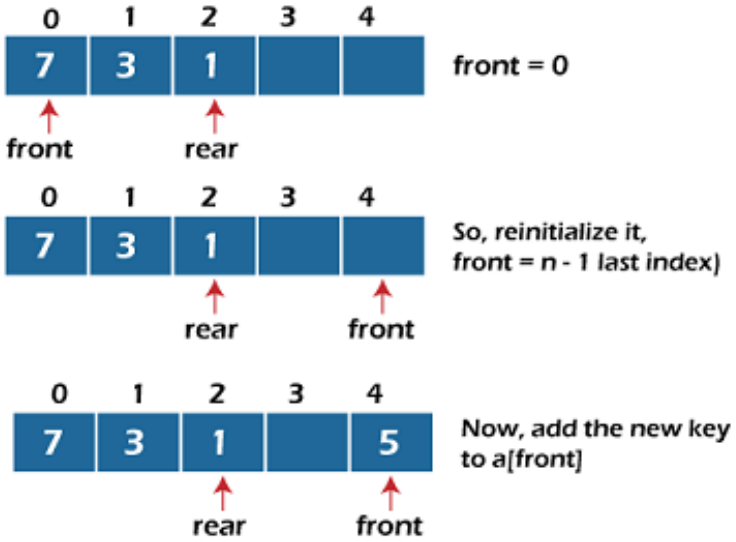
Output restricted double ended queue

Operations performed on deque

- There are the following operations that can be applied on a deque -
  - Insertion at front
  - Insertion at rear
  - Deletion at front
  - Deletion at rear
  - Get the front item from the deque
  - Get the rear item from the deque
  - Check whether the deque is full or not
  - Checks whether the deque is empty or not

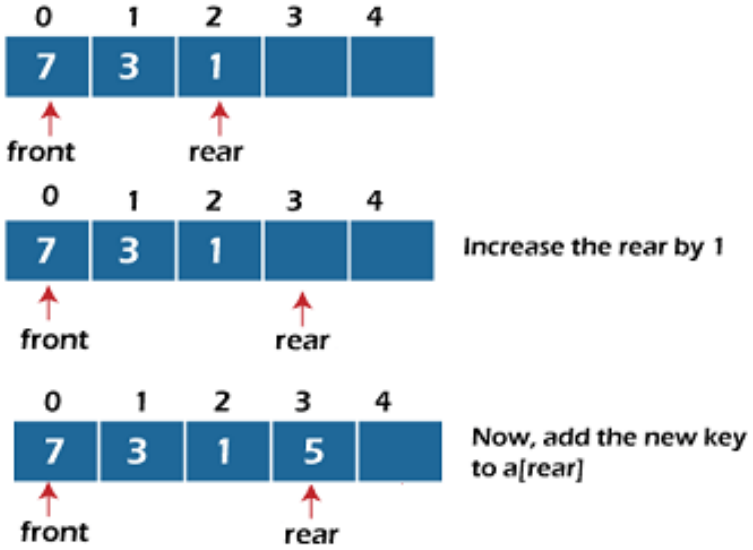
Insertion at the front end

- In this operation, the element is inserted from the front end of the queue.
- Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -
  - If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
  - Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



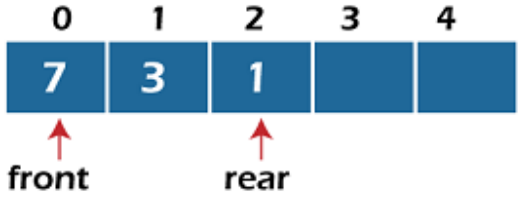
Insertion at the rear end

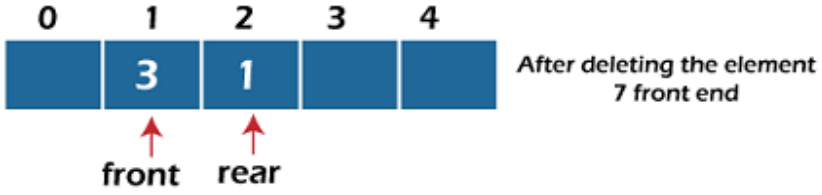
- In this operation, the element is inserted from the rear end of the queue.
- Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions –
  - If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
  - Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

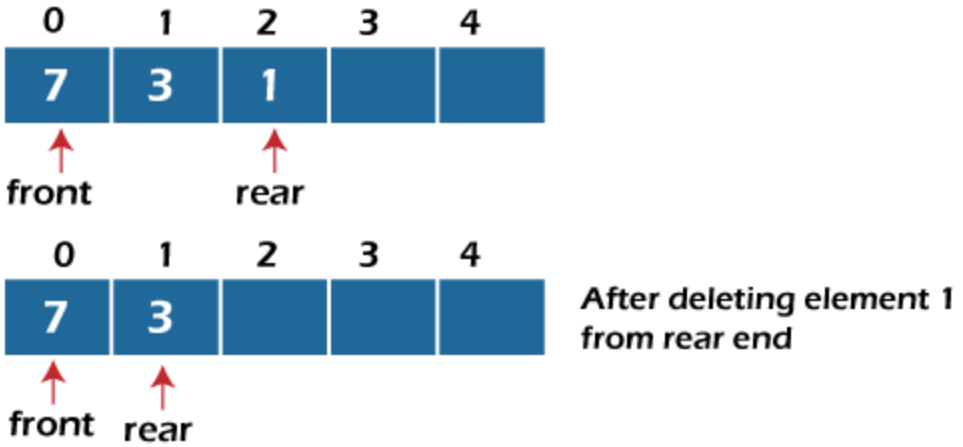
- In this operation, the element is deleted from the front end of the queue.
- Before implementing the operation, we first have to check whether the queue is empty or not.
- If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions –
  - If the deque has only one element, set rear = -1 and front = -1.
  - Else if front is at end (that means front = size - 1), set front = 0.
  - Else increment the front by 1, (i.e., front = front + 1).





Deletion at the rear end

- In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.
  - If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.
  - If the deque has only one element, set rear = -1 and front = -1.
  - If rear = 0 (rear is at front), then set rear = n - 1.
  - Else, decrement the rear by 1 (or, rear = rear -1).



- **Check empty**
- This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.
- **Check full**
- This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.
- The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

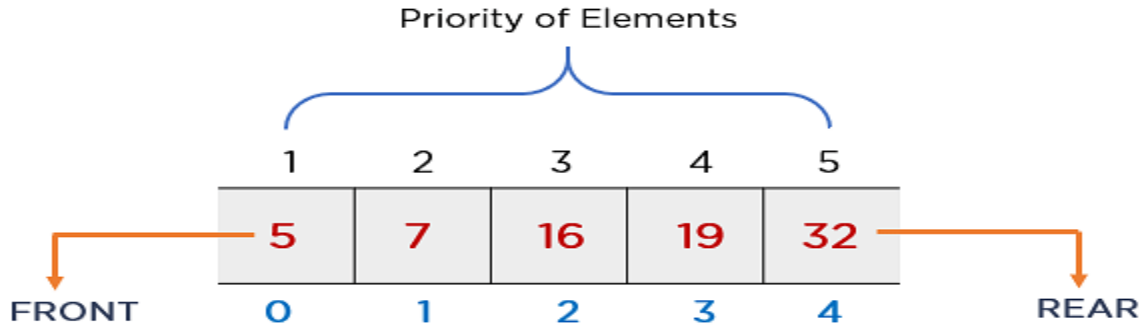
- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Priority Queues

- A priority queue is a data structure in which each element is assigned a priority.

- The priority of the element will be used to determine the order in which the elements will be processed.

- A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first.



Characteristics of a Priority queue

- A priority queue is an extension of a queue that contains the following characteristics:
  - Every element in a priority queue has some priority associated with it.
  - An element with the higher priority will be deleted before the deletion of the lesser priority.
  - If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

For example,

- If there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.

- However, CPU time is not the only factor that determines the priority, rather it is just one among several factors.

- Another factor is the importance of one process over another.

- In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

Types of Priority Queue:

- **Min Priority Queue**: Minimum number of value gets the highest priority and lowest number of element gets the highest priority.

- **Max Priority Queue**: Maximum number value gets the highest priority and minimum number of value gets the minimum priority.

Priority Queue Approaches

- Priority Queue can be implemented in two ways:

- **Using ordered Array:** In ordered array enqueue operation takes O(n) time complexity because it enters elements in sorted order in queue. And deletion takes O(1) time complexity.

- **Using unordered Array:** In unordered array deletion takes O(n) time complexity because it search for the element in Queue for the deletion and enqueue takes o(1) time complexity.

Implementation of a Priority Queue

- There are two ways to implement a priority queue.

- We can either use a **sorted list** to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority

or

- we can use an **unsorted list** so that insertions are always done at the end of the list.

- Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed.

The basic operations of a queue include:

1. Enqueue: Add an element to the back of the queue.
2. Dequeue: Remove the element at the front of the queue.
3. Peek: Return the element at the front of the queue without removing it.
4. Size: Return the number of elements in the queue.
5. isEmpty: Check if the queue is empty.

Some common applications of Queue data structure :

1. **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
2. **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
3. **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
4. **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
5. **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
6. **Traffic Management**: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.
7. **Operating systems:** Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.
8. **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
9. **Printer queues :**In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
10. **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
11. **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.