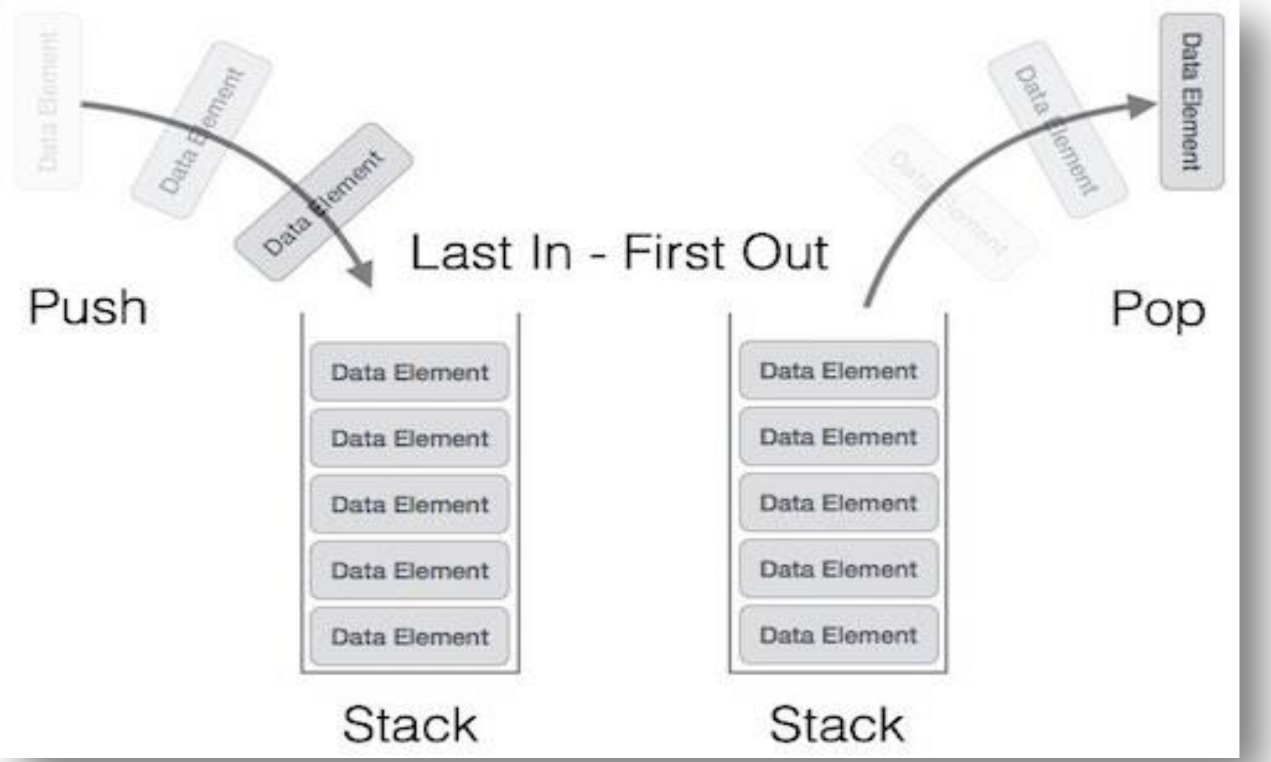
Stack

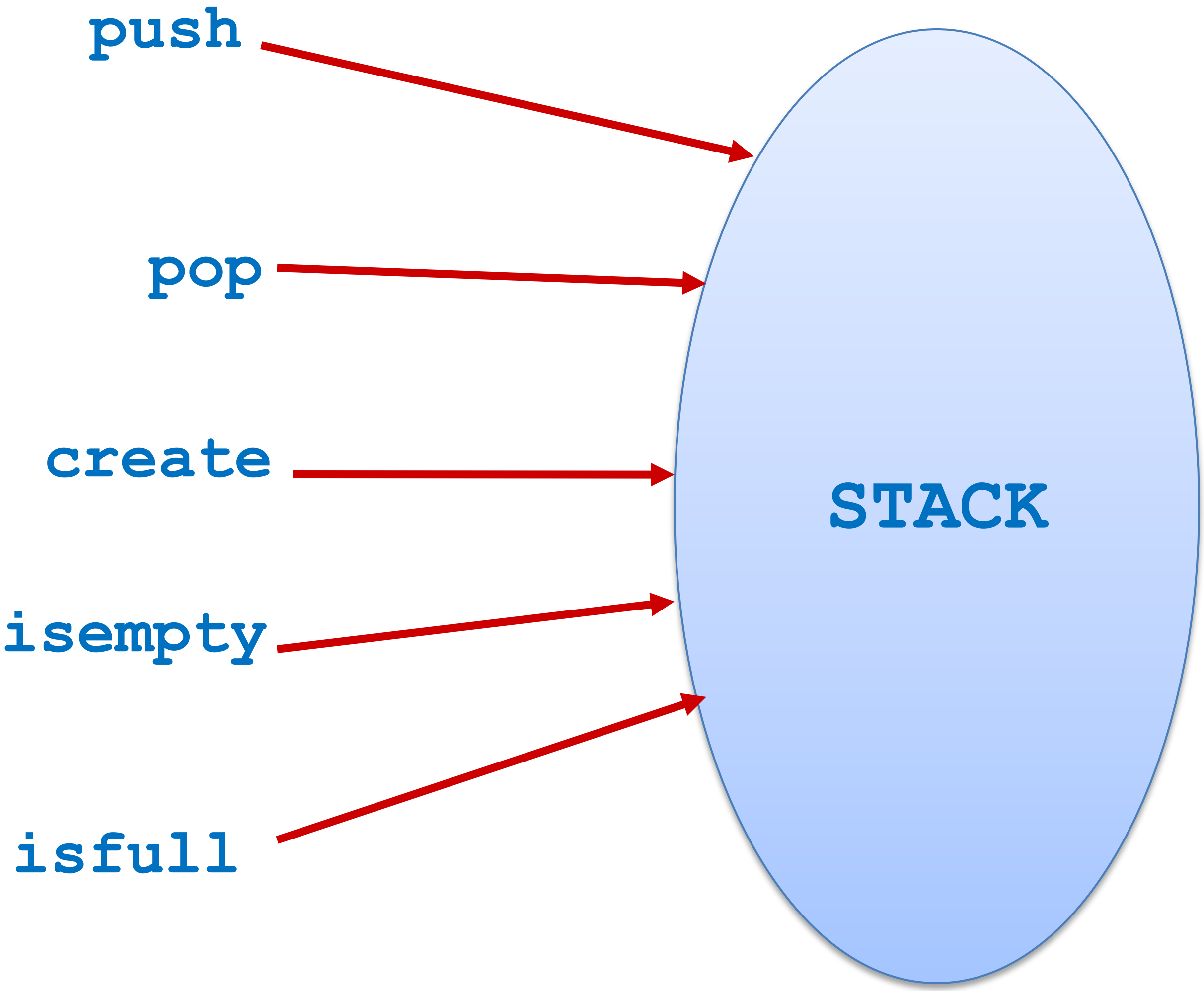
Basic Idea

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

Stack Representation



- Can be implemented by means of Array, Structure, Pointers and Linked List.
- Stack can either be a fixed size or dynamic.



push

pop

create

isempty

isfull

STACK

STACK: Last-In-First-Out (LIFO)

- `void push (stack *s, int element);`

  /* Insert an element in the stack */

- `int pop (stack *s);`

  /* Remove and return the top element */

- `void create (stack  *s);`

  /* Create a new stack */

- `int isempty (stack *s);`

  /* Check if stack is empty */

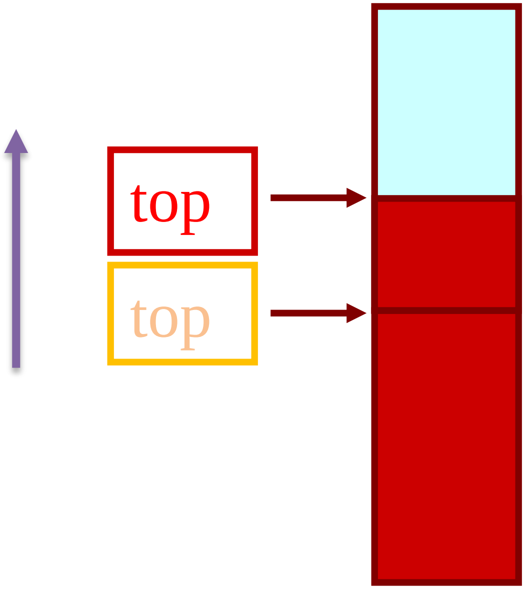
- `int isfull (stack *s);`

  /* Check if stack is full */

Assumption: stack contains integer elements!

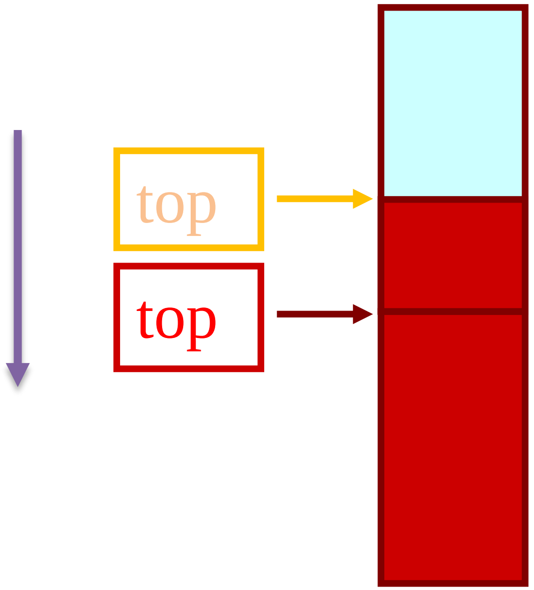
Stack using Array

Push using Stack



PUSH

Pop using Stack



POP

Applications of Stacks

- Direct applications:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Validate XML

- Indirect applications:
  - Auxiliary data structure for algorithms
  - Component of other data structures

Stacks

- Access is allowed only at one point of the structure, normally termed the *top* of the stack
  - access to the most recently added item only
- Operations are limited:
  - push (add item to stack)
  - pop (remove top item from stack)
  - top (get top item without removing it)
  - clear
  - isEmpty
  - size?
- Described as a "Last In First Out" (LIFO) data structure

Stack Operations

Assume a simple stack for integers.

```
Stack s = new Stack();
s.push(12);
s.push(4);
s.push( s.top() + 2 );
s.pop()
s.push( s.top() );
//what are contents of stack?
```

Stacks – Linked List

```
.h>
#include <stdlib.h>
// Define node structure
struct Node {
    int data;
    struct Node* next;
};
int main() {
    struct Node* top = NULL;
    struct Node* temp;
    int choice, value;
    while (1) {
        printf("\n--- Stack Menu (Linked List) ---\n");
        printf("1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                // PUSH operation
                printf("Enter value to push: ");
                scanf("%d", &value);
                temp = (struct Node*)malloc(sizeof(struct Node));
                if (temp == NULL) {
                    printf("Memory allocation failed!\n");
                    break;
                }
                temp->data = value;
                temp->next = top;
                top = temp;
                printf("%d pushed onto stack.\n", value);
                break;
```

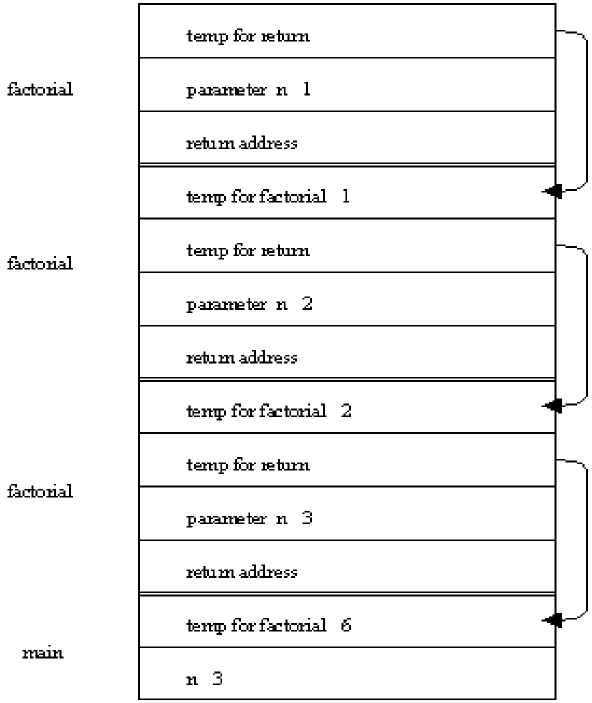
```
2:
        // POP operation
        if (top == NULL) {
            printf("Stack
Underflow! Nothing to pop.\n");
        } else {
            temp = top;
            printf("Popped
element: %d\n", top->data);
            top = top->next;
            free(temp);
        }
        break;
```

```
case 3:
        // PEEK operation
        if (top == NULL) {
            printf("Stack is
empty.\n");
        } else {
            printf("Top element:
%d\n", top->data);
        }
        break;
```

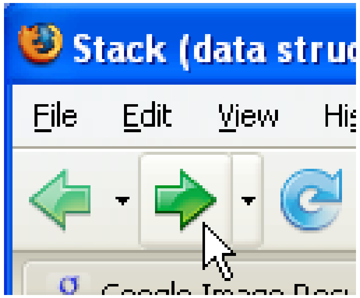
```
                // DISPLAY operation
                if (top == NULL) {
                    printf("Stack is empty.\n");
                } else {
                    printf("Stack elements:\n");
                    temp = top;
                    while (temp != NULL) {
                        printf("%d\n", temp->data);
                        temp = temp->next;
                    }
                }
                break;
            case 5:
                // EXIT
                printf("Exiting...\n");
                // Free remaining stack memory
                while (top != NULL) {
                    temp = top;
                    top = top->next;
                    free(temp);
                }
                return 0;
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }
    return 0;
}
```

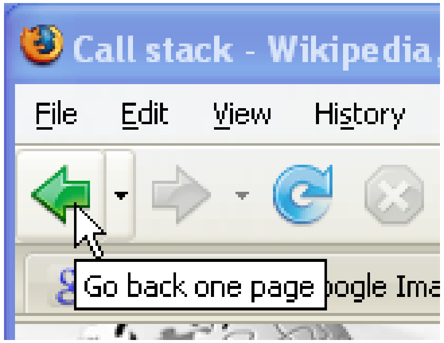
Problems that Use Stacks

- The runtime stack used by a process (running program) to keep track of methods in progress
- Search problems
- Undo, redo, back, forward



	temp for return
factorial	parameter n 1
	return address
	temp for factorial 1
	temp for return
factorial	parameter n 2
	return address
	temp for factorial 2
	temp for return
factorial	parameter n 3
	return address
	temp for factorial 6
main	n 3





Mathematical Calculations

What is 3 + 2 * 4?   2 * 4 + 3?   3 * 2 + 4?

The precedence of operators affects the order of operations. A mathematical expression cannot simply be evaluated left to right.

A challenge when evaluating a program.

Lexical analysis is the process of interpreting a program.

Involves Tokenization

What about 1 - 2 - 4 ^ 5 * 3 * 6 / 7 ^ 2 ^ 3

Steps to Convert Infix to Postfix Using a Stack

Initialize an empty stack for operators and an empty list for the output.

Scan the infix expression from left to right.

For each character in the infix expression:

If the character is an operand, add it to the output list.

If the character is an operator, pop from the stack to the output list until the stack is empty or the operator at the top of the stack has less precedence than the current operator. Then push the current operator onto the stack.

If the character is a left parenthesis ('('), push it onto the stack.

If the character is a right parenthesis (')'), pop from the stack to the output list until a left parenthesis is encountered. Remove the left parenthesis from the stack.

Pop all the operators from the stack and add them to the output list.

Infix and Postfix Expressions

- The way we are use to writing expressions is kno as infix notation
- Postfix expression does not
- require any precedence rules
- 3 2 * 1 + is postfix of 3 * 2 + 1
- evaluate the following postfix expressions and wr out a corresponding infix expression:

2 3 2 4 * + *

1 2 - 3 2 ^ 3 * 6 / +

1 2 3 4 ^ * +

2 5 ^ 1 -

uation of Postfix Expressions

- Easy to do with a stack
- given a proper postfix expression:
  - get the next token
  - if it is an operand push it onto the stack
  - else if it is an operator
    - pop the stack for the right hand operand
    - pop the stack for the left hand operand
    - apply the operator to the two operands
    - push the result onto the stack
  - when the expression has been exhausted the result is the top (and only element) of the stack

Infix to Postfix

- Convert the following equations from infix to postfix:

  2 ^ 3 ^ 3 + 5 * 1

  11 + 2 - 1 * 3 / 3 + 2 ^ 2 / 3

  Problems:

  Negative numbers?

  parentheses in expression

Infix to Postfix Conversion

- Requires operator precedence parsing algorithm
  - parse v. To determine the syntactic structure of a sentence or other utterance

Operands: add to expression

Close parenthesis: pop stack symbols until an open parenthesis appears

Operators:

  Have an on stack and off stack precedence

  Pop all stack symbols until a symbol of lower precedence appears. Then push the operator

End of input: Pop all remaining stack symbols and add to the expression

Infix and Postfix Notations

- Infix: operators placed between operands:

  A+B*C

- Postfix: operands appear before their operators:-

  ABC*+

- There are no precedence rules to learn in postfix notation, and parentheses are never needed

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

A + B * C → (A + (B * C)) → (A + (B C *)) → A B C * +

A + B * C + D → ((A + (B * C)) + D) → ((A + (B C*))+ D) → ((A B C *+) + D) → A B C * + D +

Infix to postfix conversion

- Use a stack for processing operators (push and pop operations).
- Scan the sequence of operators and operands from left to right and perform one of the following:
  - output the operand,
  - push an operator of higher precedence,
  - pop an operator and output, till the stack top contains operator of a lower precedence and push the present operator.

The algorithm steps

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

- No Two Operators of same priority can stay together in stack
- Lowest priority cannot be placed before highest priority

K+L-M*N+(O^P)*W/U/V*T+Q

Reversal of a String

Reversing a string using a stack involves utilizing the Last-In-First-Out (LIFO) property of stacks.

Push Operation: Each character of the string is pushed onto the stack. Since the stack operates in a LIFO manner, the first character of the string ends up at the bottom of the stack, and the last character is at the top.

Pop Operation: Each character is then popped from the stack. As a result, the characters are retrieved in reverse order, since the last character pushed (which is the last character of the original string) is the first to be popped.

Introduction to Analysis and Algorithms

Overview of Key Concepts and Notations

1. Introduction to Analysis

Analysis of Algorithms is the study of the performance of algorithms, focusing on time and space requirements.

It helps in comparing different algorithms for the same task, particularly for large inputs.

Two main types:

Empirical analysis: Based on implementation and running on test cases.

Theoretical analysis: Based on mathematical estimation.

2. What is an Algorithm?

An algorithm is a finite sequence of well-defined instructions used to solve a problem.

Examples: Searching, sorting, multiplication, etc.

Expressed using:

- Pseudocode
- Flowcharts
- Programming code

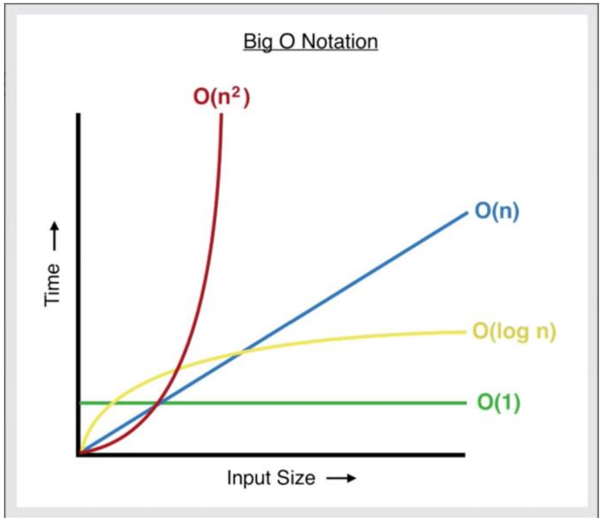
3. Characteristics of Algorithms

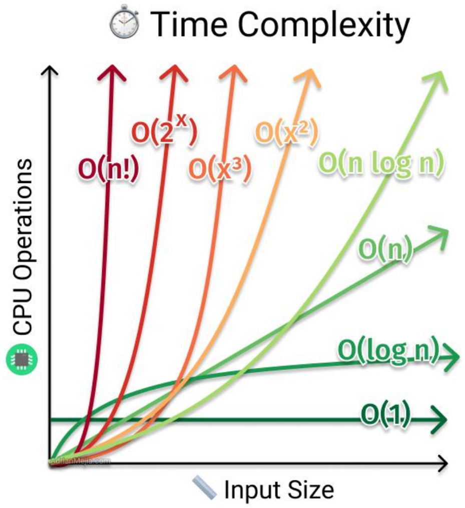
- Input: Zero or more inputs.
- Output: At least one output.
- Finiteness: Must terminate.
- Definiteness: Clear instructions.
- Effectiveness: Basic, feasible operations.

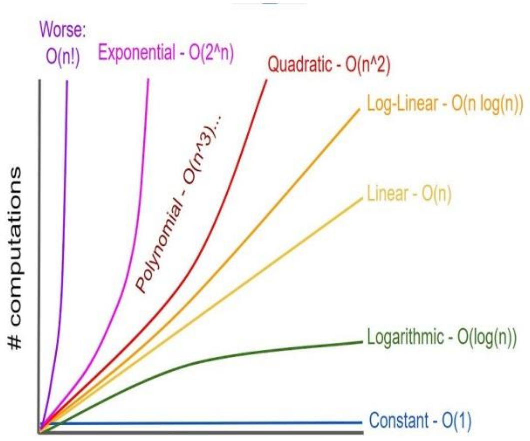
4. Time and Space Complexities

- Time: Number of primitive operations.
- Space: Memory usage during execution.
- Examples:
- - Linear Search: O(n)
- - Binary Search: O(log n)

Time and Space Complexities







Space Complexities

Definition:
Space complexity measures the amount of memory an algorithm requires to run as a function of the input size.
Goal:
To understand how the memory usage of an algorithm increases as the input data grows larger.
Example:
If an algorithm creates a copy of the input array, its space complexity is O(n), as the memory usage grows linearly with the size of the input array. If an algorithm only uses a fixed amount of extra memory regardless of the input size, its space complexity is O(1).
Common Notations:
Like time complexity, space complexity is also often expressed using Big O notation.

Time Complexities

Definition:

Time complexity measures the amount of time an algorithm takes to run as a function of the size of its input.

Goal:

To understand how the execution time of an algorithm increases as the input data grows larger.

Example:

Consider searching for a specific element in a list. A linear search (checking each element one by one) has a time complexity of O(n), meaning the time it takes grows linearly with the number of elements (n) in the list. In contrast, a binary search (assuming the list is sorted) has a time complexity of O(log n), which is significantly faster for large lists.

Common Notations:

Time complexity is often expressed using Big O notation, such as O(1), O(log n), O(n), O(n log n), O(n^2).

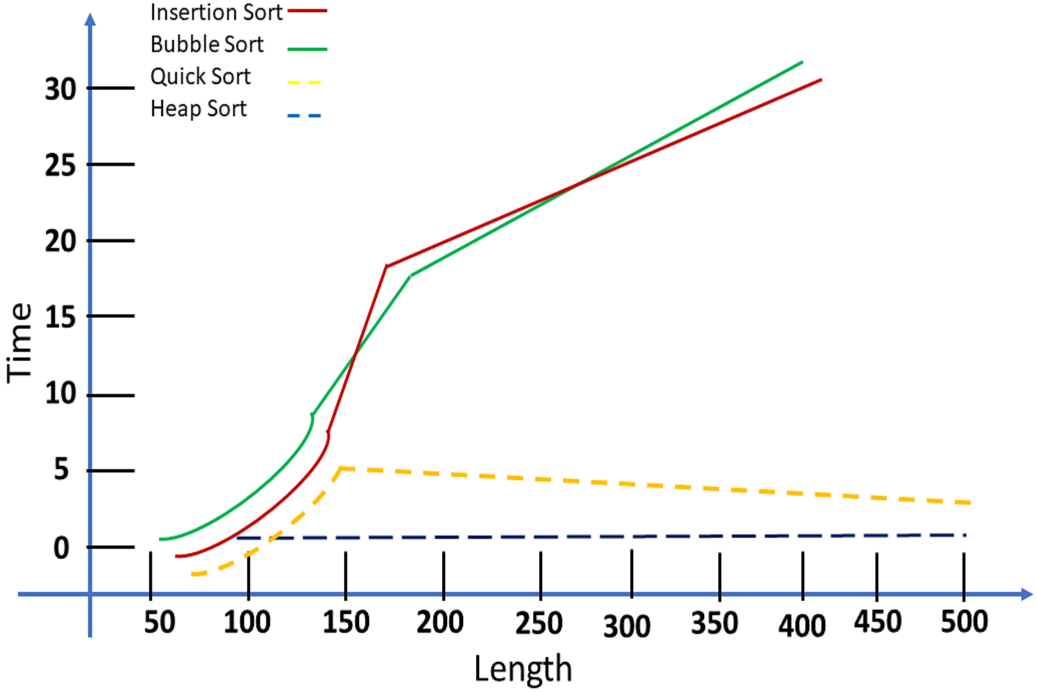
5. Order of Growth Functions

- O(1): Constant - Array access
- O(log n): Logarithmic - Binary Search
- O(n): Linear - Linear Search
- O(n log n): Linearithmic - Merge Sort
- O(n^2): Quadratic - Bubble Sort
- O(2^n): Exponential - Recursive Fibonacci
- O(n!): Factorial - TSP

6. Asymptotic Notations

- Big-O (O): Upper bound - Worst case
- Omega (Ω): Lower bound - Best case
- Theta (Θ): Tight bound - Average case
- Example: $T(n) = 3n^2 + 2n + 1$ → $O(n^2)$

Big O notation	Theta notation(θ)	Omega notation(Ω)
Represents the worst-case time complexity i.e. the upper bound.	Represents the average-case time complexity.	Represents the best-case time complexity i.e. the lower bound.



Methods for Calculating Space Complexity

S(n) denote the algorithm's space complexity.

an integer occupies 4 bytes of memory.

As a result, the number of allocated bytes would be the space complexity.
Line 1 allocates memory space for two integers, resulting in S(n) = 4 bytes multiplied by 2 = 8 bytes.
Line 2 represents a loop.
Lines 3 and 4 assign a value to an already existing variable.
As a result, there is no need to set aside any space.
The return statement in line 6 will allocate one more memory case.
As a result, S(n)= 4 times 2 + 4 = 12 bytes.
Because the array is used in the algorithm to allocate n cases of integers, the final space complexity will be f S(n) = n + 12 = O (n).

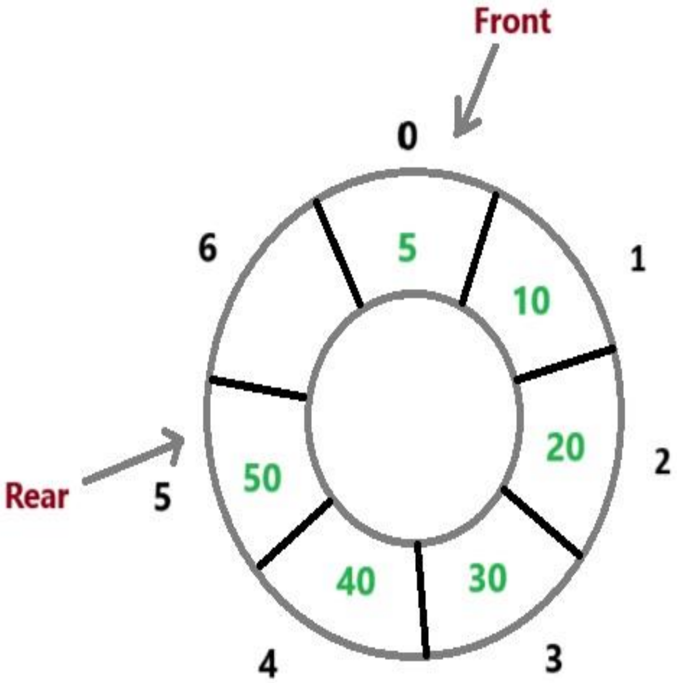
```
1.int mul, i
2.While i < = n do
3.   mul <- mul * array[i]
4.   i <- i + 1
5.end while
6.return mul
```

```
1. mul <- 1
2. i <- 1
3. While i <= n do
 4   mul = mul * 1
 5    i = i + 1
6 End while
```

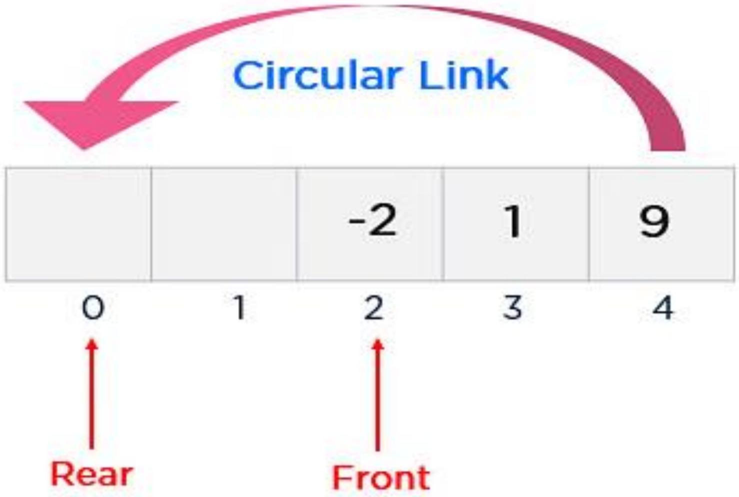
Methods for Calculating Time Complexity

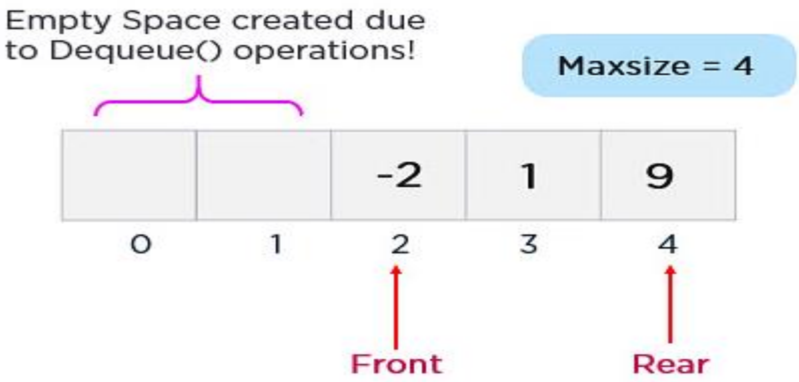
Let T(n) be a function of the algorithm's time complexity.
Lines 1 and 2 have a time complexity of O. (1).
Line 3 represents a loop.
As a result, you must repeat lines 4 and 5 (n -1) times.
As a result, the time complexity of lines 4 and 5 is O. (n).

Finally, adding the time complexity of all the lines yields the overall time complexity of the multiple function fT(n) = O(n).
The iterative method gets its name because it calculates an iterative algorithm's time complexity by parsing it line by line and adding the complexity..



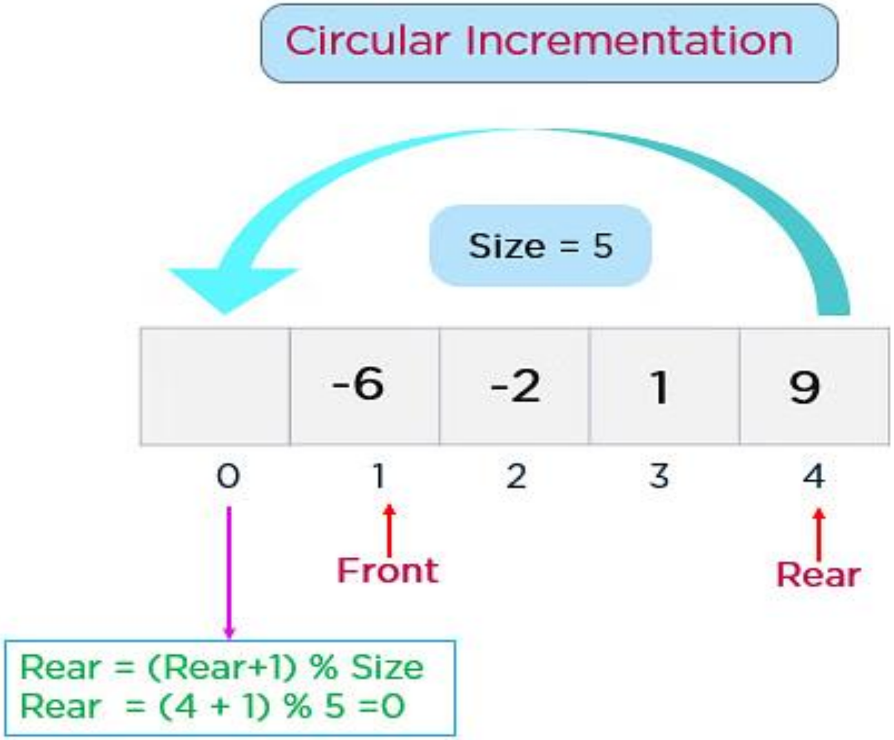
Circular link allows rear pointer to reach at the beginning of a queue.





Rear + 1 = 4 + 1 = 5 (Overflow Error)

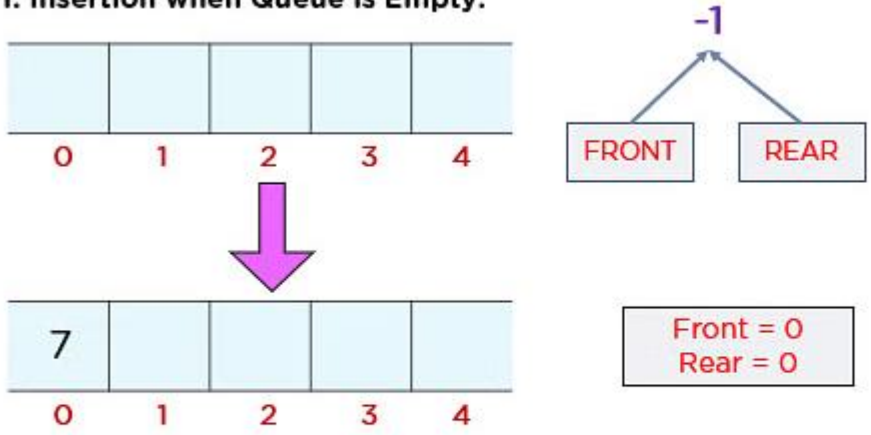
Rear = (Rear + 1)% MaxSize = 0 (Reached loc. 0 / Beginning of queue)



Rear = (Rear+1) % Size
Rear = (4 + 1) % 5 =0

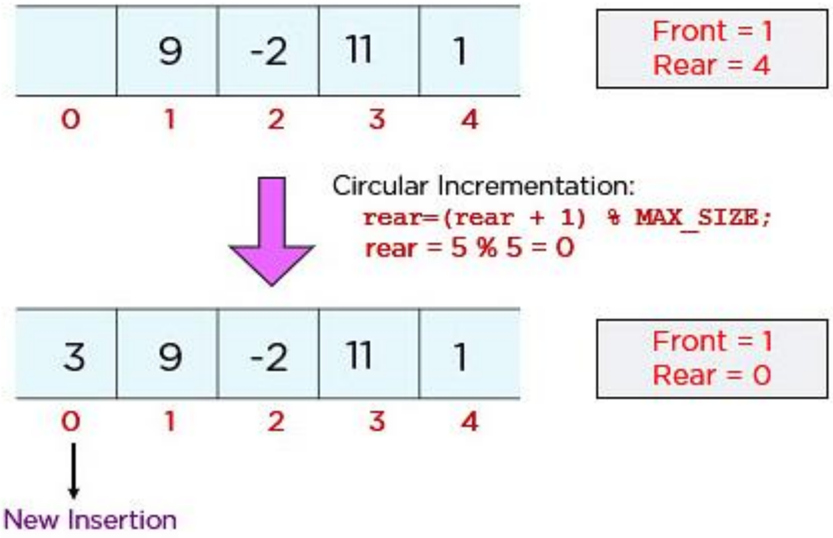
- Step 1: Check if the queue is full (Rear + 1 % Maxsize = Front)
- Step 2: If the queue is full, there will be an Overflow error
- Step 3: Check if the queue is empty, and set both Front and Rear to 0
- Step 4: If Rear = Maxsize - 1 & Front != 0 (rear pointer is at the end of the queue and front is not at 0th index), then set Rear = 0
- Step 5: Otherwise, set Rear = (Rear + 1) % Maxsize
- Step 6: Insert the element into the queue (Queue[Rear] = x)
- Step 7: Exit

1. Insertion when Queue is Empty:



Front = 0
Rear = 0

2. Insertion when queue is completely filled but there is space at the beginning of the queue:

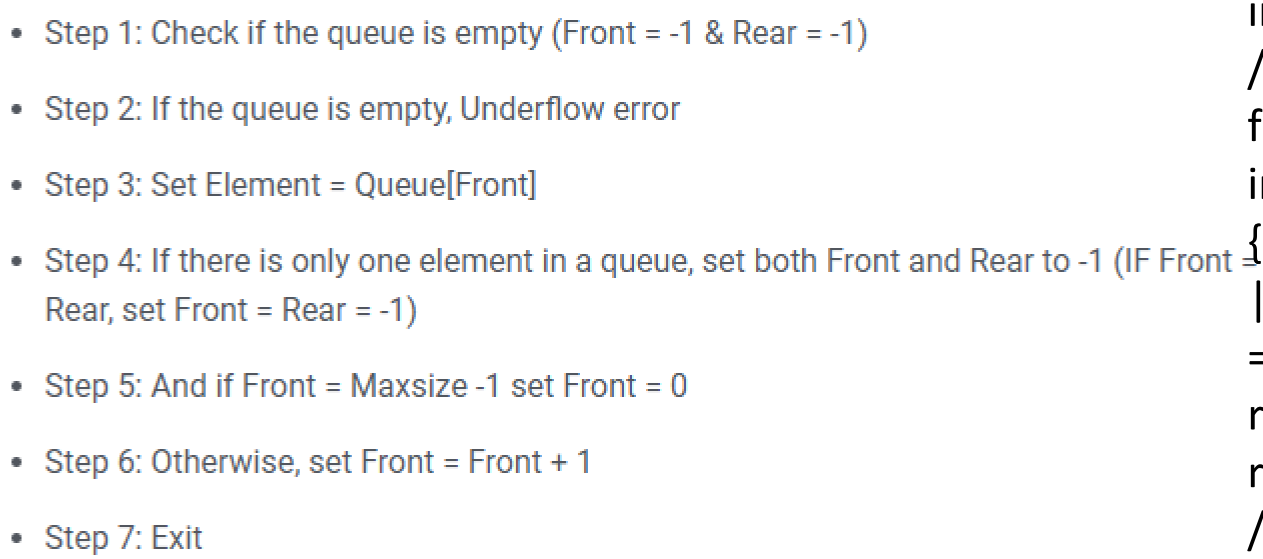


Front = 1
Rear = 4

Circular Incrementation:

```
rear=(rear + 1) % MAX_SIZE;
rear = 5 % 5 = 0
```

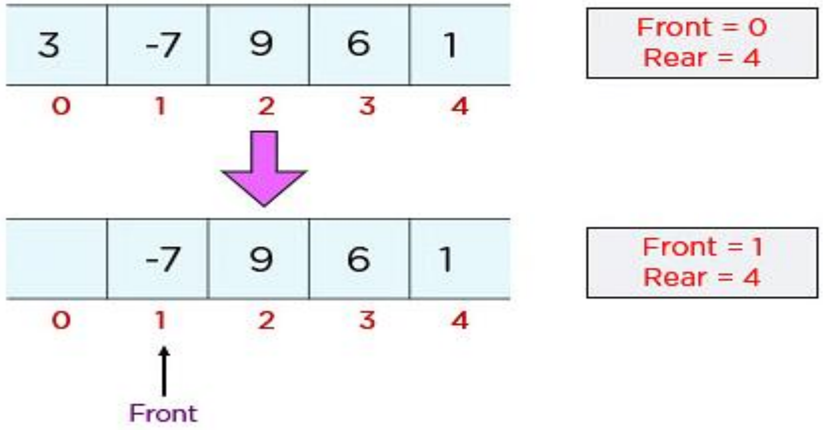
Front = 1
Rear = 0



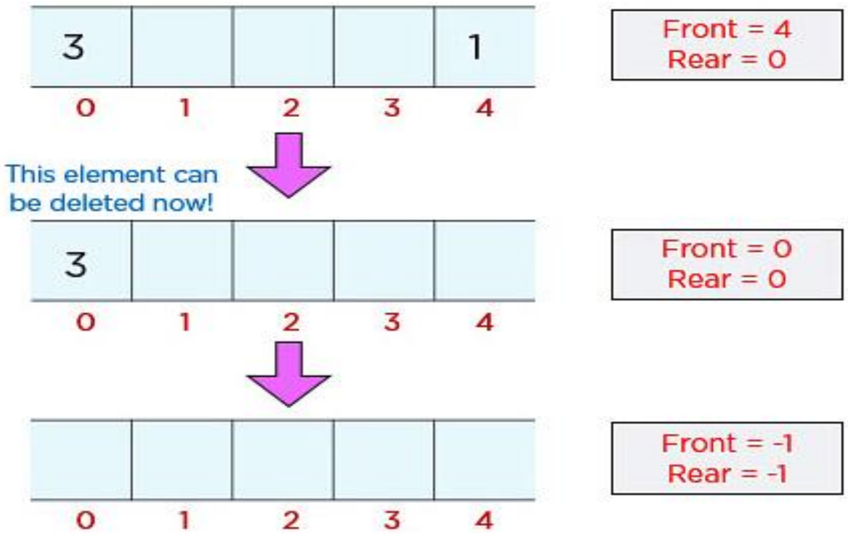
- Step 1: Check if the queue is empty (Front = -1 & Rear = -1)
- Step 2: If the queue is empty, Underflow error
- Step 3: Set Element = Queue[Front]
- Step 4: If there is only one element in a queue, set both Front and Rear to -1 (IF Front = Rear, set Front = Rear = -1)
- Step 5: And if Front = Maxsize -1 set Front = 0
- Step 6: Otherwise, set Front = Front + 1
- Step 7: Exit

```
#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; }
```

1. Deletion when rear at the end of queue and front at the beginning of the queue



2. Deletion when front reached at end of queue but there is element rear is at beginning of queue



Condition	Code Expression
Queue is Empty	front == -1
Queue is Full	(rear + 1) % SIZE == front
Enqueue	rear = (rear + 1) % SIZE; queue[rear] = item;
Dequeue	front = (front + 1) % SIZE;

```
#include <stdio.h>
#define SIZE 5
int main()
{
    int queue[SIZE];
    int front = -1, rear = -1;
    int choice, item, i;
    while (1)
            {
        printf("\n--- Circular Queue Menu ---\n");
        printf("1. Enqueue (Insert)\n");
        printf("2. Dequeue (Delete)\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
            {
```

```
case 1:
if ((front == 0 && rear == SIZE - 1) || (rear + 1)
% SIZE == front)
{
          printf("Queue is Full Overflow)\n");
      }
else {
          printf("Enter the item to insert: ");
          scanf("%d", &item);
          if (front == -1) {
             front = rear = 0;
          } else {
             rear = (rear + 1) % SIZE;
          }
          queue[rear] = item;
          printf("Item inserted: %d\n", item);
      }
      break;
```

```
case 2:
         if (front == -1)
         {
           printf("Queue is Empty Underflow)\n");
        }
         else
         {
           printf("Item deleted: %d\n", queue[front]);
           if (front == rear)
         {
              front = rear = -1;
           }
         else
         {
              front = (front + 1) % SIZE;
           }
        }
        break;
```

```
case 3: if (front == -1) {
          printf("Queue is Empty\n");
        } else {
          printf("Queue elements: ");
          i = front;
          while (1) {
            printf("%d ", queue[i]);
            if (i == rear)
              break;
            i = (i + 1) % SIZE;
          }
          printf("\n");
        }
        break;
```

```
case 4:
         printf("Exiting program.\n");
        return 0;

      default:
        printf("Invalid choice. Try again.\n");
    }
  }

  return 0;
}
```

```
#include <stdio.h>
#include <string.h>
#define SIZE 100

int main() {
    char str[SIZE], stack[SIZE];
    int top = -1;
    int i;

    printf("Enter a string: ");
    gets(str);  // For simplicity; use fgets in real code

    // Push all characters onto stack
    for (i = 0; str[i] != '\0'; i++) {
        top++;
        stack[top] = str[i];
    }
    // Pop characters and overwrite original string
    for (i = 0; top != -1; i++) {
        str[i] = stack[top];
        top--;
    }
    str[i] = '\0';

    printf("Reversed string: %s\n", str);

    return 0;
}
```

Reversal of a String

Initialize a Stack: Create an empty stack to hold the characters of the string.

Push Characters: Traverse the string character by character and push each character onto the stack.

Pop Characters: Pop each character from the stack and append it to the reversed string.

Checking Validity of Expressions with Nested Parentheses

Validating an expression with nested parentheses involves ensuring that each opening parenthesis has a corresponding closing parenthesis and that they are properly nested. This can be efficiently checked using a stack:

- **Push Operation**: Every time an opening parenthesis (is encountered, it is pushed onto the stack.
- **Pop Operation**: Every time a closing parenthesis) is encountered, it is checked against the stack. If the stack is empty (no corresponding opening parenthesis), the expression is invalid. If the stack is not empty, the top element (an opening parenthesis) is popped.
- After processing the entire expression, if the stack is empty, all opening parentheses had matching closing parentheses and the expression is valid. If the stack is not empty, it indicates unmatched opening parentheses.

Checking Validity of Expressions with Nested Parentheses

Initialize a Stack: Create an empty stack to hold the opening parentheses.

Traverse the Expression:

- **Push (**: When an opening parenthesis is encountered, push it onto the stack.
- **Pop)**: When a closing parenthesis is encountered, check the stack:
    - If the stack is empty, the expression is invalid.
    - If the stack is not empty, pop the top element.

Final Check: After traversing the expression, if the stack is empty, the expression is valid. Otherwise, it is invalid.

Queue

A queue is a **linear data structure** that follows the ***First-In-First-Out (FIFO)*** principle.

In a queue, the element that **is added first will be removed first**, much like a real-life queue (e.g., a line of people waiting at a bank).

Queue

Queue as an Abstract Data Type (ADT)

As an Abstract Data Type, a queue is defined by the following properties and operations, regardless of the specific implementation:

FIFO Order: Elements are processed in the order they were added.

Operations:

- **Enqueue**: Add an element to the rear of the queue.
- **Dequeue**: Remove an element from the front of the queue.
- **Front (or Peek)**: Access the front element without removing it.
- **IsEmpty**: Check if the queue is empty.
- **IsFull**: Check if the queue is full (if the implementation has a fixed size).

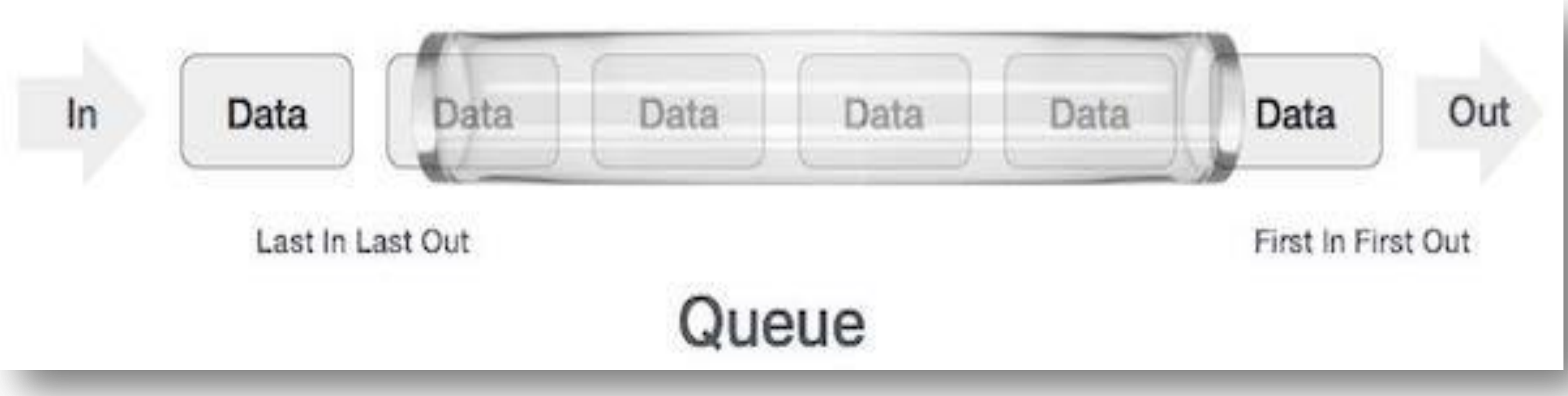
Basic Idea

- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to

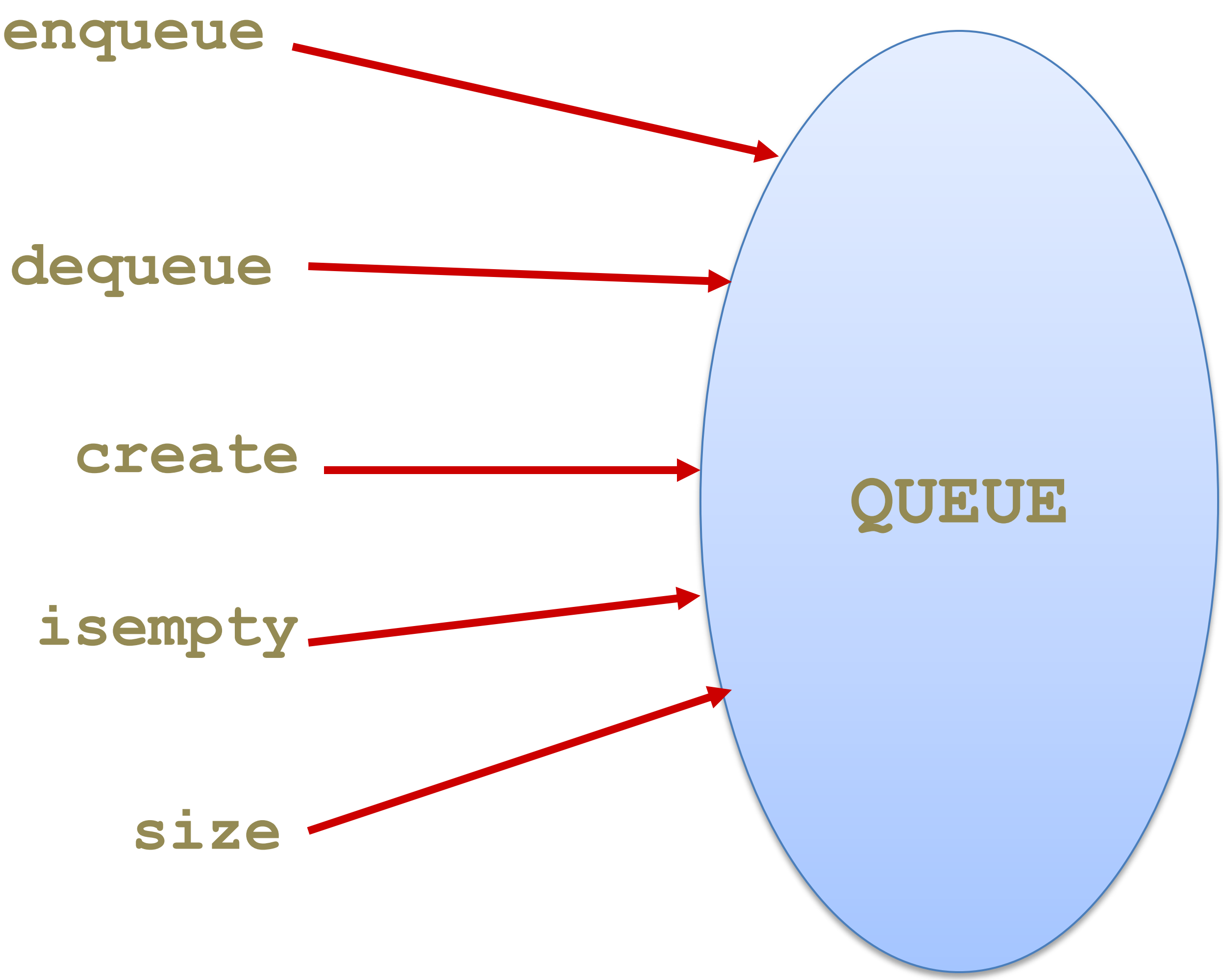
to

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

Queue Representation



- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.



enqueue

dequeue

create

isempty

size

QUEUE

QUEUE: First-In-First-Out (LIFO)

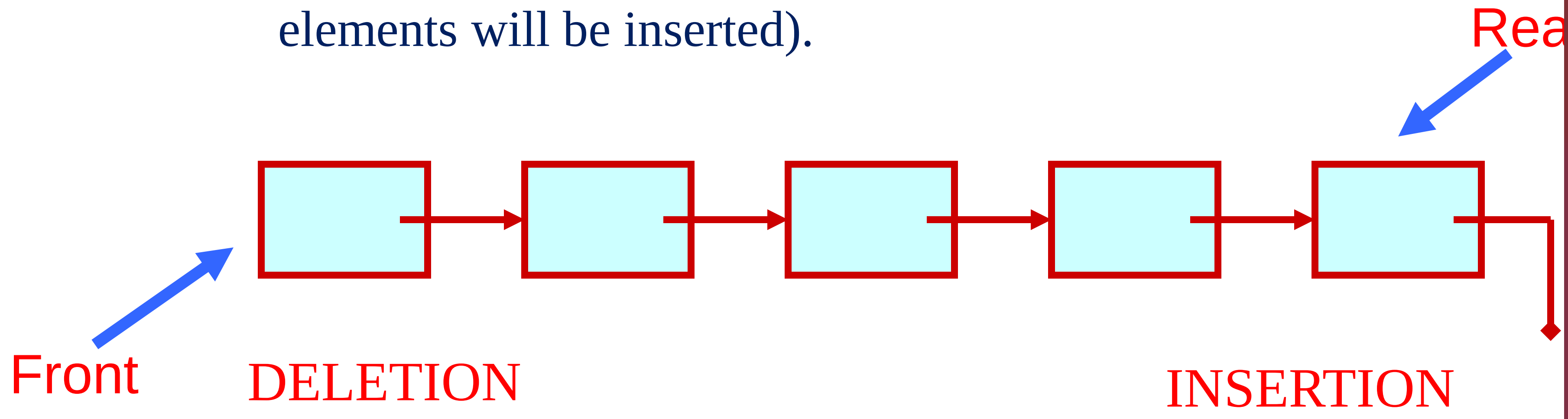
```
void enqueue (queue *q, int element);
                    /* Insert an element in the queue */
int dequeue (queue *q);
                    /* Remove an element from the queue */
queue *create();
                    /* Create a new queue */
int isempty (queue *q);
                    /* Check if queue is empty */
int size (queue *q);
                    /* Return the no. of elements in queue */
```

Assumption: queue contains integer elements!

Queue using Linked List

Basic Idea

- Basic idea:
  - Create a linked list to which items would be added to one end and deleted from the other end.
  - Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted).
    - Another pointing to the end of the list (point where new elements will be inserted).



Rea

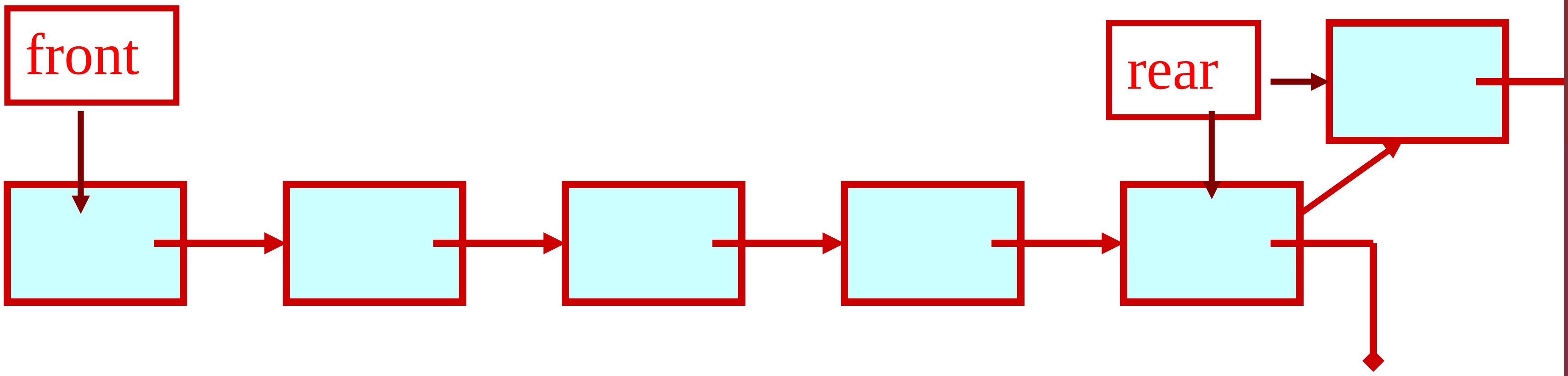
Front

DELETION

INSERTION

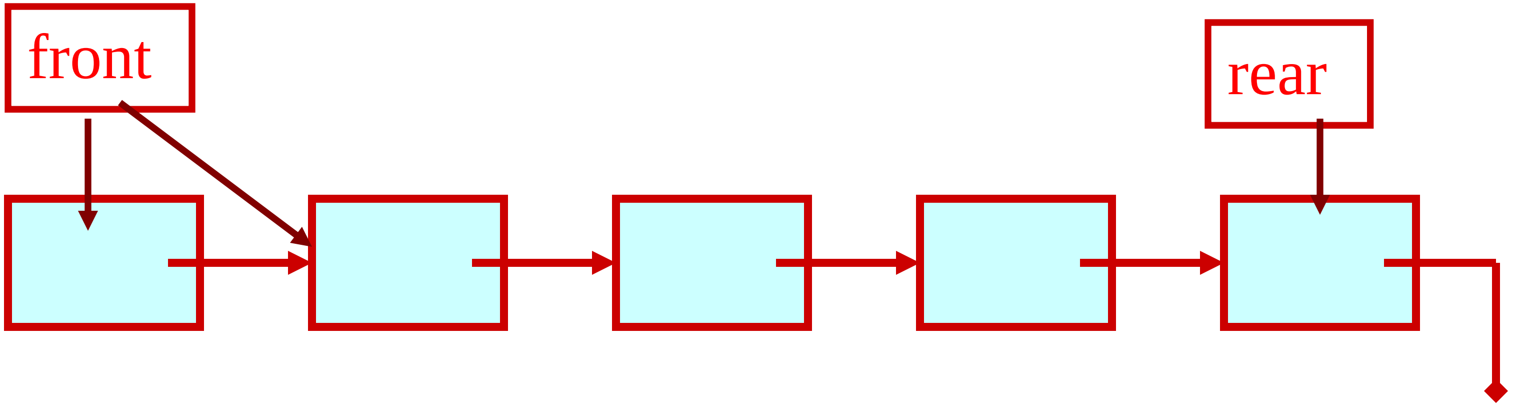
Queue: Linked List Structure

ENQUEUE



Queue: Linked List Structure

DEQUEUE



Example :Queue using Linked List

```
struct qnode
{
  int val;
   struct qnode *next;
};

struct queue
{
   struct qnode *qfront, *qrear;
};
typedef struct queue QUEUE;
```

```
void enqueue (QUEUE *q,int element)
{
   struct qnode *q1;
   q1=(struct qnode *)malloc(sizeof(struct qnode));
   q1->val= element;
   q1->next=q->qfront;
   q->qfront=q1;
}
```

Example :Queue using Linked List

```
int size (queue *q)
{
   queue *q1;
   int count=0;
   q1=q;
   while(q1!=NULL)
   {
     q1=q1->next;
     count++;
   }
return count;
}
```

```
int dequeue (queue *q)
{
   int val;
   queue *q1,*prev;
   q1=q;
   while(q1->next!=NULL)
   {
      prev=q1;
      q1=q1->next;
   }
   val=q1->val;
   prev->next=NULL;
   free(q1);
return (val);
}
```

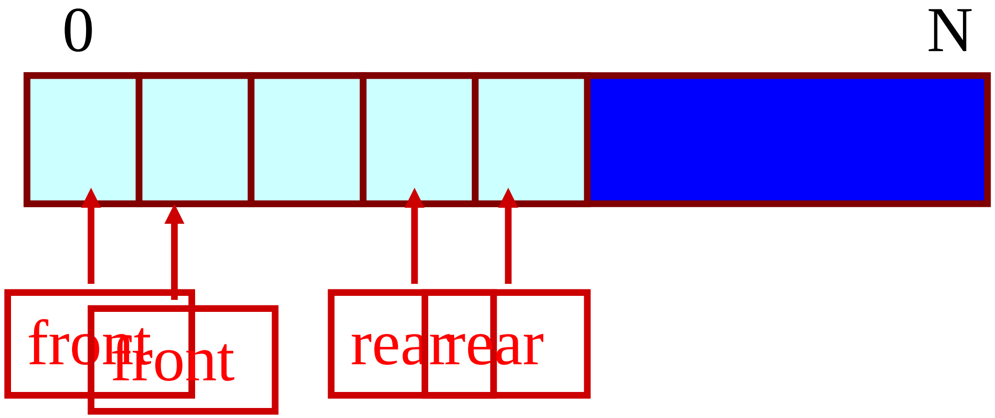
```
int peek (queue *q)
{
   queue *q1;
   q1=q;
   while(q1->next!=NULL)
      q1=q1->next;
return (q1->val);
}
```

Problem With Array Implementation

- The size of the queue depends on the number and order of enqueue and dequeue.
- It may be situation where memory is available but enqueue is not possible.

ENQUEUE DEQUEUE

Effective queuing storage area of array gets reduced.



Use of circular array indexing

Applications of Queues

- Direct applications:-
  - Waiting lists
  - Access to shared resources (e.g., printer)
  - Multiprogramming

- Indirect applications:-
  - Auxiliary data structure for algorithms
  - Component of other data structures