PCCST303DATA STRUCTURES AND ALGORITHMS

MODULE 1

SYLLABUS

Basic Concepts of Data Structures Definitions; Data Abstraction; Performance Analysis - Time & Space Complexity, Asymptotic Notations;

Polynomial representation using Arrays, Sparse matrix (Tuple representation); Stacks and Queues - Stacks, Multi-Stacks, Queues, Circular Queues, Double Ended Queues; Evaluation of Expressions- Infix to Postfix, Evaluating Postfix Expressions.

SYSTEM LIFE CYCLE (SLC)

- Good programmers regard large scale computer programs as systems that contain many complex interacting parts. (Systems: Large Scale Computer Programs.)
- As systems, these programs undergo a development process called System life cycle.(SLC : Development Process of Programs)

  Different Phases of System Life Cycle

  1. Requirements
  2. Analysis
  3. Design
  4. Refinement and coding
  5. Verification

1. Requirement Phase:

- All programming projects begin with a **set of specifications** that defines the purpose of that program.
- Requirements describe the information that the programmers are given (**input**) and the results (**output**) that must be produced.
- Frequently the initial specifications are defined vaguely and we must develop rigorous input and output descriptions that include all cases.

2. Analysis Phase

- In this phase the problem is break down into manageable pieces.
- There are two approaches to analysis:-**bottom up and top down**.

- Bottom up approach is an older, **unstructured** strategy that places an early emphasis on coding fine points. Since the programmer does not have a master plan for the project, the resulting program frequently has many **loosely connected, error ridden segments**.
- Top down approach is a structured approach divide the program into manageable segments.
- This phase generates **diagrams** that are used to design the system.
- Several alternate solutions to the programming problem are developed and compared during this phase

3. Design Phase

- This phase continues the work done in the analysis phase.
- The designer approaches the system from the perspectives of both **data objects** that the program needs and the **operations** performed on them.
- The first perspective leads to the **creation of abstract data types** while the second requires the **specification of algorithms** and a consideration of algorithm design strategies.

  Ex: Designing a scheduling system for university

  Data objects: Students, courses, professors etc

  Operations: insert, remove search etc

  ie. We might add a course to the list of university courses, search for the courses taught by some professor etc.
- Since abstract data types and algorithm specifications are language independent.
- We must specify the information required for each data object and ignore coding details.

  Ex: Student object should include name, phone number, social security number etc.

4. Refinement and Coding Phase

- In this phase we choose representations for data objects and write algorithms for each operation on them.
- Data objects representation can determine the efficiency of the algorithm related to it. So we should write algorithms that are independent of data objects first.
- Frequently we realize that we could have created a much better system. (May be we realize that one of our alternate design is superior than this). If our original design is good, it can absorb changes easily.

5. Verification Phase

- This phase consists of
  - developing correctness proofs for the program
  - Testing the program with a variety of input data.
  - Removing errors.

Correctness of Proofs

- Programs can be proven correct using proofs.(like mathematics theorem)
- Proofs are very time consuming and difficult to develop for large projects.
- Scheduling constraints prevent the development of complete sets of proofs for a larger system.
- However, selecting algorithm that have been proven correct can reduce the number of errors.

Testing

- Testing can be done only after coding.
- Testing requires working code and set of test data.
- Test data should be chosen carefully so that it includes all possible scenarios.
- Good test data should verify that every piece of code runs correctly.
- For example if our program contains a *switch* statement, our test data should be chosen so that we can check each *case* within *switch* statement.

Error Removal

- If done properly, the correctness of proofs and system test will indicate erroneous code.
- Removal of errors depends on the design and code.
- While debugging large undocumented program written in 'spaghetti' code, each corrected error possibly generates several new errors.
- Debugging a well documented program that is divided into autonomous units that interact through parameters is far easier. This especially true if each unit is tested separately and then integrated into system.

ALGORITHMS

Definition: An ***algorithm*** is a finite set of instructions to accomplish a particular task. In addition, all algorithms must satisfy the following criteria:

(1) ***Input***. There are zero or more quantities that are externally supplied.

(2) ***Output***. At least one quantity is produced.

(3) ***Definiteness***. Each instruction is clear and unambiguous.

(4) ***Finiteness***. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) ***Effectiveness***. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

We can describe algorithm in many ways

1. We can use a natural language like English
2. Graphical Representation called flow chart, but they work well only if the algorithm is small and simple.

Translating a Problem into an Algorithm

Example [Selection sort]: Suppose we must devise an algorithm that sorts a collection of n > 1 elements of arbitrary type. A simple solution is given by the following

[Selection Sort: In each pass of the selection sort, the smallest element is selected from the unsorted list and exchanged with the elements at the beginning of the unsorted list]

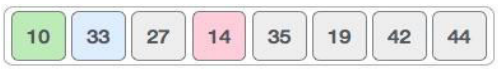
Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −

- From those elements that are currently unsorted, find the smallest and place it next in the sorted list
- We assume that the elements are stored in an array 'list', such that the i^{th} integer is stored in the i^{th} Position list[i], 0 <= i <n
- Algorithm 1.1 is our first attempt to deriving a solution

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

1.1 Selection sort algorithm

- We are written this partially in C and partially in English
- To turn the program 1.1 into a real C program, two clearly defined sub tasks are remain: **finding the smallest integer** and **interchanging it with list[i].**

- We can solve this by using a function

```
void swap(int *x, int *y)
/* both parameters are pointers to ints */
   int temp = *x;   /* declares temp as an int and assigns
                       to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

1.2 Swap Function

- To swap their values one could call swap(&a, &b)
- We can solve the first subtask by assuming that the minimum is the list[i]. Checking list[i] with list[i+1], list[i+2]……,list[n-1]. Whenever we find a smaller number we make it as the minimum. We reach list[n-1] we are finished.

```
#include <stdio.h>
int main()
{
        int a[100], n, i, j, position, swap;
        printf("Enter number of elements");
        scanf("%d", &n);
        printf("Enter %d Numbersn", n);
        for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
        for(i = 0; i < n - 1; i++)
        {
            position=i;
            for(j = i + 1; j < n; j++)
            {
                    if(a[position] > a[j])
                    position=j;
            }
            if(position != i)
            {
                    swap=a[i];
                    a[i]=a[position];
                    a[position]=swap;
            }
        }
        printf("Sorted Array:n");
        for(i = 0; i < n; i++)
```

```
        printf("%dn", a[i]);
        return 0;
}
```

- **Correctness Proof**

Theorem 1.1 Algorithm SelectionSort(a, n) correctly sorts a set of $n \geq 1$ elements; the result remains in $a[1:n]$ such that $a[1] \leq a[2] \leq \cdots \leq a[n]$.

Proof: We first note that for any i, say $i = q$, following the execution of lines 6 to 9, it is the case that $a[q] \leq a[r]$, $q < r \leq n$. Also observe that when i becomes greater than q, $a[1:q]$ is unchanged. Hence, following the last execution of these lines (that is, $i = n$), we have $a[1] \leq a[2] \leq \cdots \leq a[n]$.

We observe at this point that the upper limit of the **for** loop in line 4 can be changed to $n-1$ without damaging the correctness of the algorithm. □

Example 1.2 [*Binary search*]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, *list* [0] $\leq$ *list* [1] $\leq \cdots \leq$ *list* [n−1]. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, i, such that *list*[i] = *searchnum*. If *searchnum* is not present, we should return −1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, *left* = 0 and *right* = n−1. Let *middle* = (*left*+*right*)/2 be the middle position in the list. If we compare *list* [*middle*] with *searchnum*, we obtain one of three results:

(1) **searchnum < list[middle]**. In this case, if *searchnum* is present, it must be in the positions between 0 and *middle* − 1. Therefore, we set *right* to *middle* − 1.

(2) **searchnum = list[middle]**. In this case, we return *middle*.

(3) **searchnum > list[middle]**. In this case, if *searchnum* is present, it must be in the positions between *middle* + 1 and n − 1. So, we set *left* to *middle* + 1.

If *searchnum* has not been found and there are still integers to check, we recalculate *middle* and continue the search. Program 1.5 implements this searching strategy. The algorithm contains two subtasks: (1) determining if there are any integers left to check, and (2) comparing *searchnum* to *list*[*middle*].

```
while (there are more integers to check ) {
   middle = (left + right) / 2;
   if (searchnum < list[middle])
      right = middle - 1;
   else if (searchnum == list[middle])
         return middle;
      else left = middle + 1;
}
```

Program 1.5: Searching a sorted list

We can handle the comparisons through either a function or a macro. In either case, we must specify values to signify less than, equal, or greater than. We will use the strategy followed in C's library functions:

- We return a negative number (−1) if the first number is less than the second.
- We return a 0 if the two numbers are equal.
- We return a positive number (1) if the first number is greater than the second.

Although we present both a function (Program 1.6) and a macro, we will use the macro throughout the text since it works with any data type.

```
int compare(int x, int y)
{/* compare x and y, return -1 for less than, 0 for equal,
    1 for greater */
   if (x < y) return -1;
   else if (x == y) return 0;
        else return 1;
}
```

Program 1.6: Comparison of two integers

The macro version is:

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <=  . . .  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

Recursive Algorithm

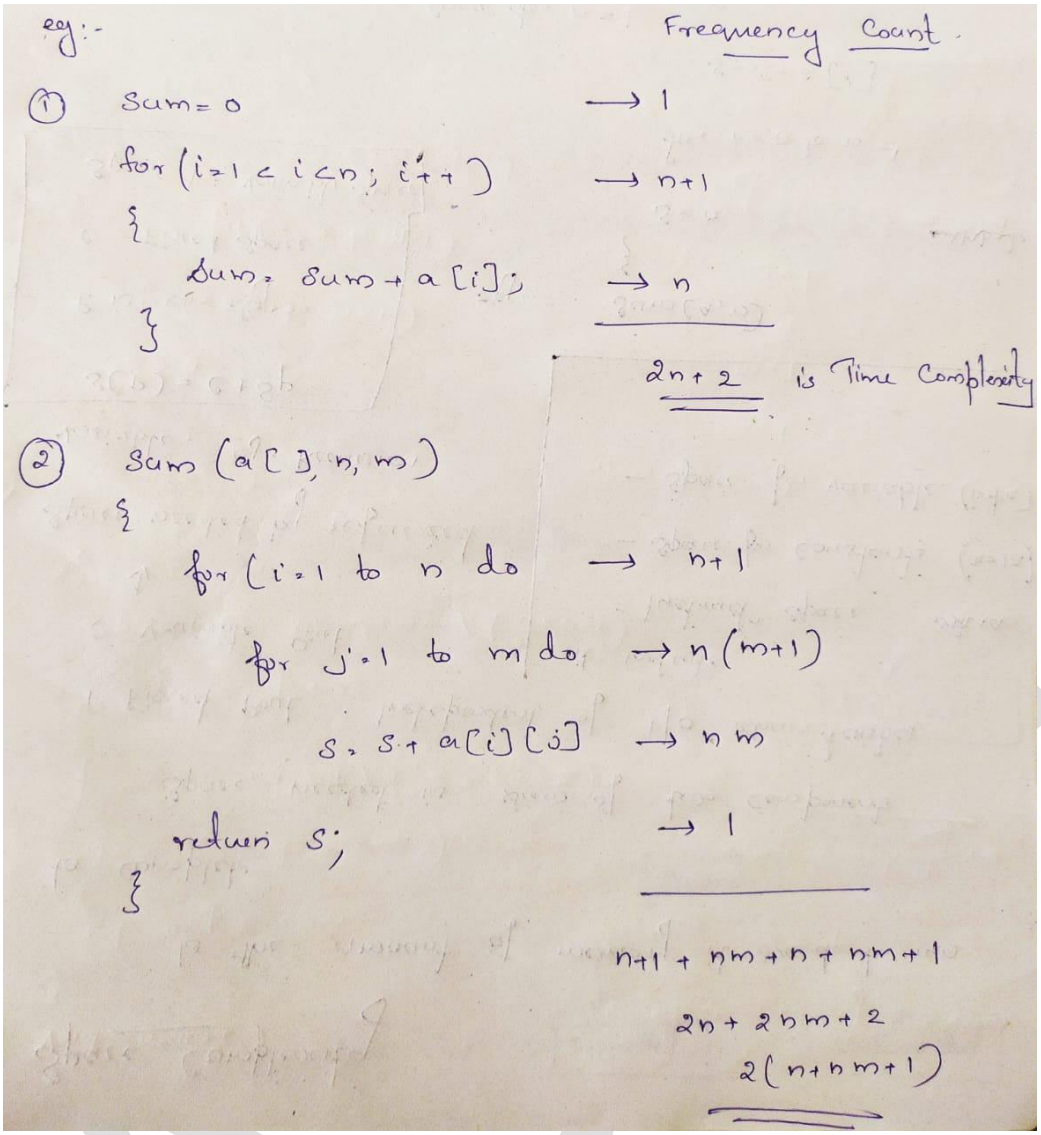
- An algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is direct recursive.
- Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A.
- These recursive mechanisms are extremely powerful, but even more importantly; many times they can express an otherwise complex process very clearly.

```
nt binsearch(int list[], int searchnum, int left,
                                        int right)
/* search list[0] <= list[1] <= ... <= list[n-1] for
   searchnum. Return its position if found. Otherwise
   return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
```

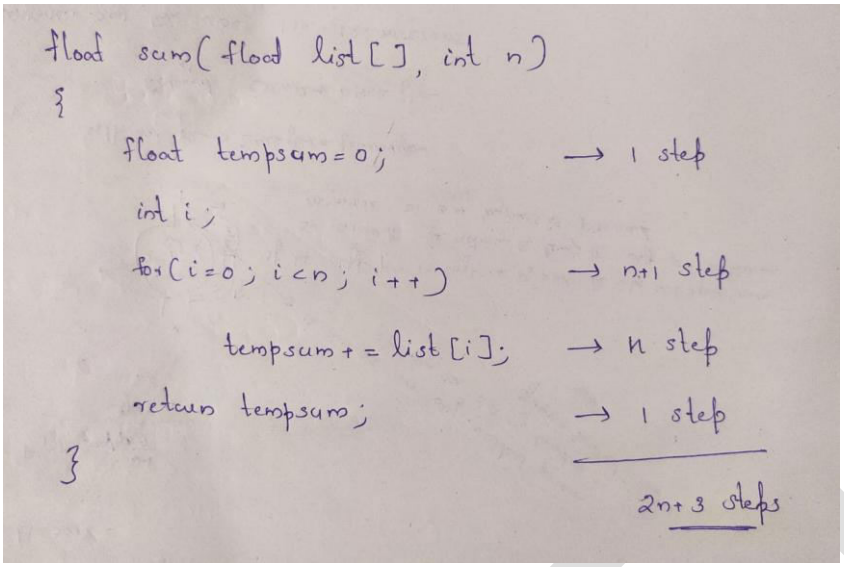
rogram 1.8: Recursive implementation of binary search

➢

- Examples:



3. Iterative function for summing a list of numbers

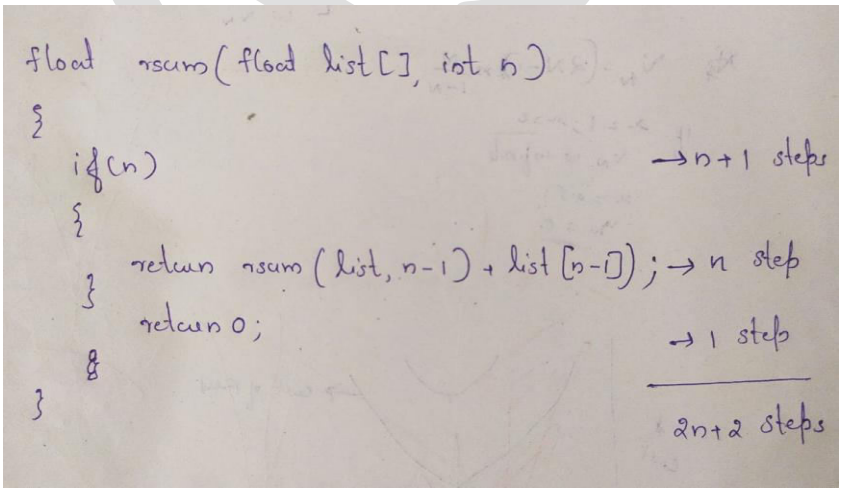


Tabular Method

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

s/e =steps/execution

4. Recursive summing of a list of numbers



Tabular Method

Statement	s/e	Frequency	Total steps
float rsum(float list[ ], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

- When we analyze an algorithm it depends on the input data, there are three cases :
  a. **Best case:** The best case is the minimum number of steps that can be executed for the given parameters.
  b. **Average case:** The average case is the average number of steps executed on instances with the given parameters.
  c. **Worst case**: In the worst case, is the maximum number of steps that can be executed for the given parameters

ASYMPTOTIC NOTATION

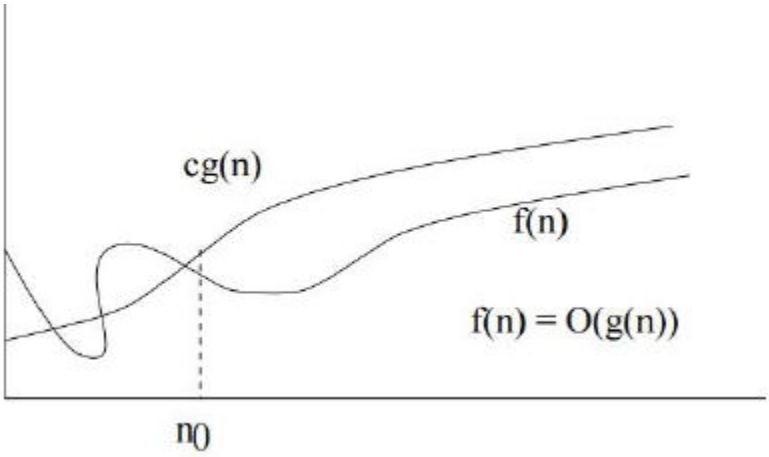
- Complexity of an algorithm is usually a function of n.
- Behavior of this function is usually expressed in terms of one or more standard functions.
- Expressing the complexity function with reference to other known functions is called **asymptotic complexity**.
- Three basic notations are used to express the asymptotic complexity

1. **Big – Oh notation O :** Upper bound of the algorithm
2. **Big – Omega notation Ω :** Lower bound of the algorithm
3. **Big – Theta notation Θ :** Average bound of the algorithm

1. Big – Oh notation O

- Formal method of expressing the upper bound of an algorithm's running time.
- i.e. it is a measure of longest amount of time it could possibly take for an algorithm to complete.
- It is used to represent the **worst case** complexity.

- **f(n) = O(g(n))** if and only if there are two positive constants **c** and **n0** such that **f(n) ≤ c g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is big-O of g(n)"**.



- Examples:

1. Derive the Big – Oh notation for f(n) = 2n + 3

Ans:

2n + 3 <= 2n+3n

2n+3 <= 5n for all n>=1

Here c = 5

g(n) = n

so, f(n) = O(n)

T(n)	Complexity
$5n^3+200n^2+15$	$O(n^3)$
$3n^2+2^{300}$	$O(n^2)$
$5\log_2 n+15 \log n$	O(log n)
$2\log n^3$	O(log n)

$4n+\log n$	$O(n)$
2^{64}	$O(1)$
$\text{Log } n^{10}+2\sqrt{n}$	$O(\sqrt{n})$
2^n+n^{1000}	$O(2^n)$

Eg. 1: $3n+2$

$f(n)=3n+2 \leq 4n$ for all $n \geq 2$

$\leq 4*n$

$\leq O(n)$

Eg2:- $10n2 +4n+2$

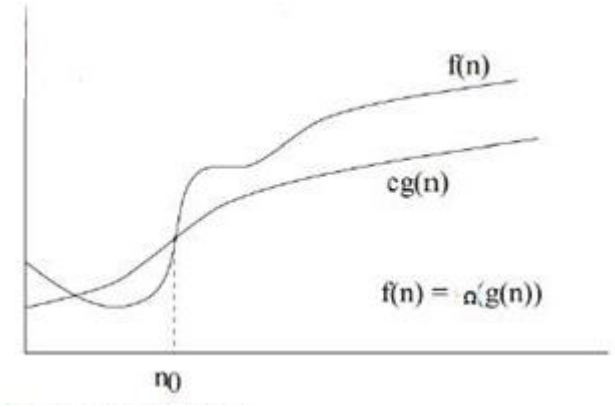
$f(n)= 10n2 +4n+2 \leq 11n2$ for all $n \geq 5$

How did we find n>=5
We will check for values of n=1,2,3,4
and only after n become 5 and higher
this will hold true

$= O(n2)$

2. Big – Omega notation Ω

- **f(n) = Ω(g(n))** if and only if there are two positive constants **c** and **n0** such that **f(n) ≥ c g(n)** for all **n ≥ n0.**
- Then we say that **"f(n) is omega of g(n)"**.



- Examples:

  Derive the Big – Omega notation for f(n) = 2n + 3

  Ans:

  2n + 3 >= 1n for all n>=1

  Here c = 1

  g(n) = n

  so, f(n) = Ω (n)

3. Big – Theta notation Θ

- **f(n) = Θ(g(n))** if and only if there are three positive constants c1, c2 and n0 such that **c1 g(n) ≤ f(n) ≤ c2 g(n)** for all **n ≥ n0** .
- Then we say that **"f(n) is theta of g(n)"**.
- Examples:

  Derive the Big – Theta notation for f(n) = 2n + 3

  Ans:

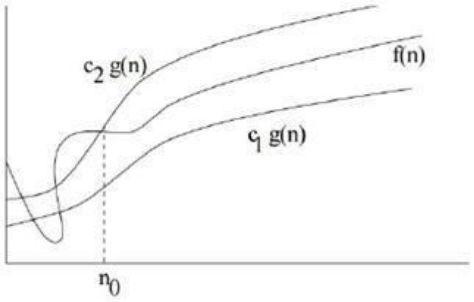
  1n <= 2n + 3 <= 5n for all n>=1

  Here c1 = 1

  C2 = 5

  g1(n) and g2(n) = n

  so, f(n) = **Θ** (n)



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof: •

When $n \geq 1$, $n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$

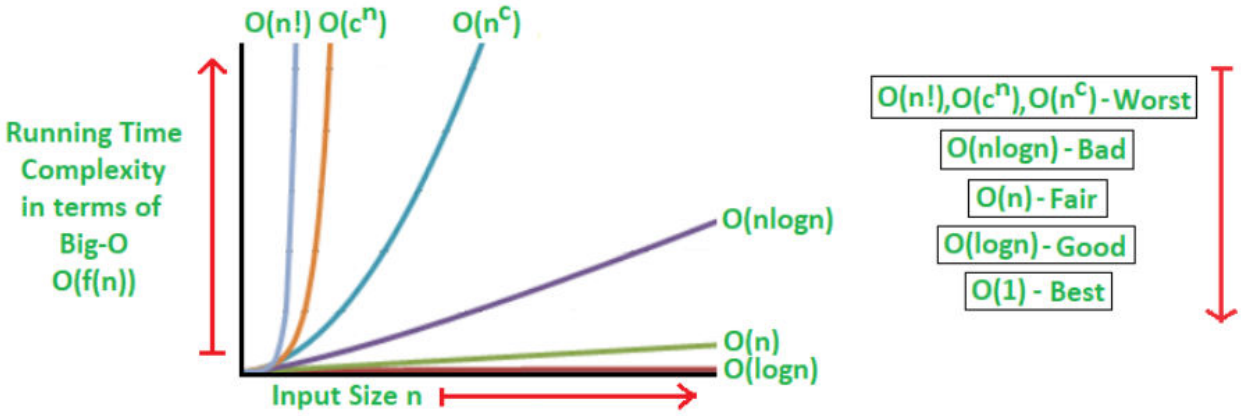
- When $n \geq 0$, $n^2 \leq n^2 + 5n + 7$
- Thus, when $n \geq 1$

  $1n^2 \leq n^2 + 5n + 7 \leq 13n^2$

  Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, $c1 = 1$, and $c2 = 13$.)

Comparison of different Algorithm

Algorithm	Best case	Average case	Worst case
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Binary search	O(1)	O(log n)	O(log n)
Linear search	O(1)	O(n)	O(n)



Examples

1. $f(n) = 2n^2 + 3n + 4$

$\Rightarrow 2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$

$2n^2 + 3n + 4 \leq 9n^2 \quad n \geq 1$

$\downarrow \quad \downarrow$

$c \quad g(n)$

$f(n) = O(g(n))$

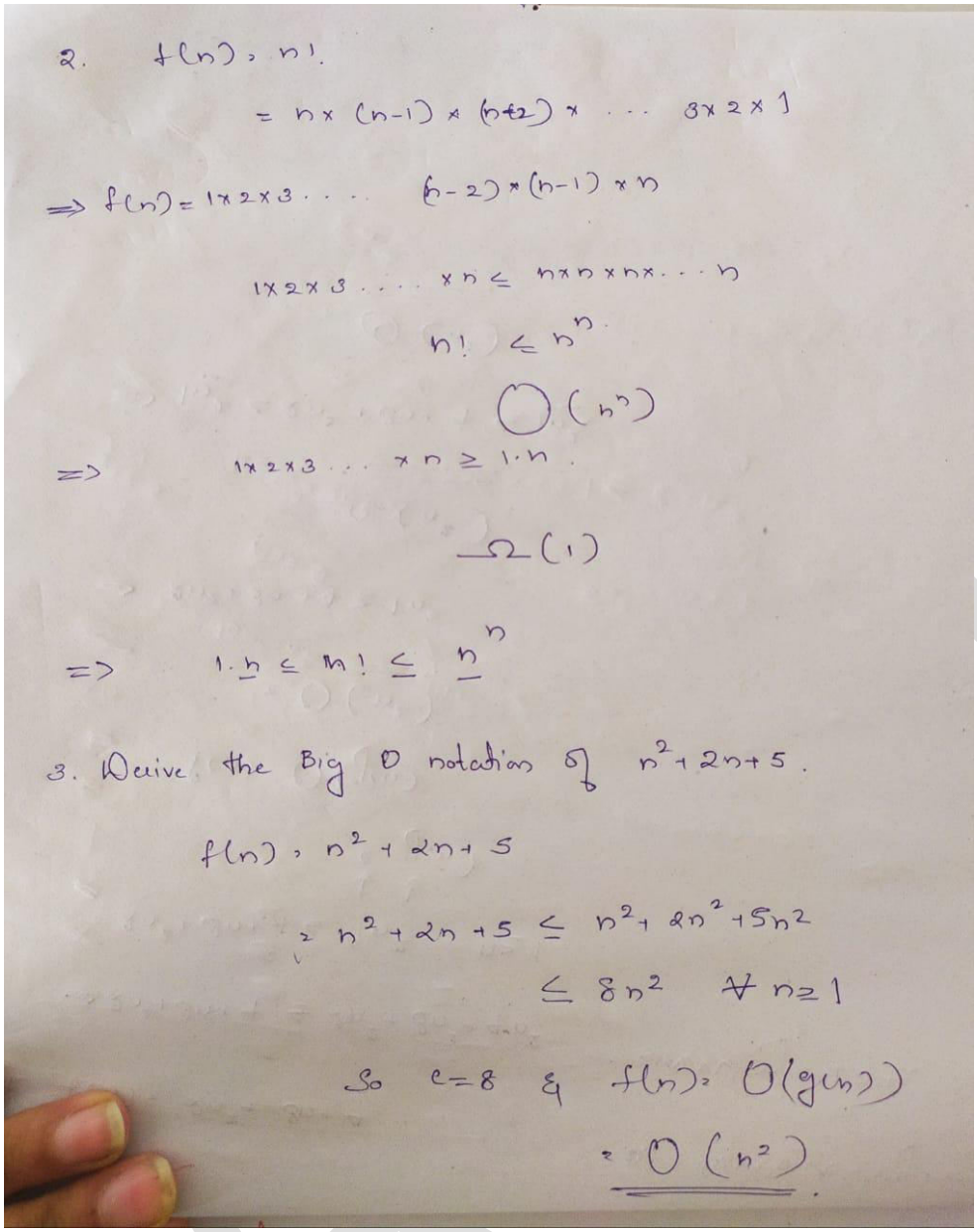
$= O(n^2)$

$\Rightarrow 2n^2 + 3n + 4 \geq 1n^2$

$\Omega(n^2)$

$\Rightarrow 1n^2 \leq 2n^2 + 3n + 4 \leq 9n^2$

$\Theta(n^2)$



TIME COMPLEXITY OF LINEAR SEARCH

- Any algorithm is analyzed based on the unit of computation it performs. For linear search, we need to count the number of comparisons performed, but each comparison may or may not search the desired item.

Best Case	Worst Case	Average Case
1	n	n / 2

TIME COMPLEXITY OF BINARY SEARCH

- In Binary search algorithm, the target key is examined in a sorted sequence and this algorithm starts searching with the middle item of the sorted sequence.
    a. If the middle item is the target value, then the search item is found and it returns True.
    b. If the target **item < middle** item, then search for the target value in the first half of the list.
    c. If the target **item > middle** item, then search for the target value in the second half of the list.
- In binary search as the list is ordered, so we can eliminate half of the values in the list in each iteration.
- Consider an example, suppose we want to search 10 in a sorted array of elements, then we first determine 15 the middle element of the array. As the middle item contains 18, which is greater than the target value 10, so can discard the second half of the list and repeat the process to first half of the array. This process is repeated until the desired target item is located in the list. If the item is found then it returns True, otherwise False.
- In Binary Search, each comparison eliminates about half of the items from the list. Consider a list with n items, then about ***n/2*** items will be eliminated after first comparison. After second comparison, ***n/4*** items of the list will be eliminated. If this process is repeated for several times, then there will be just one item left in the list. The number of comparisons required to reach to this point is $n/2^i = 1$. If we solve for i, then it gives us $i = log_2 n$. The maximum number is comparison is logarithmic in nature, hence the time complexity of binary search is ***O(log n).***

Best Case	Worst Case	Average Case
1	O(log n)	O(log n)

Compute time complexity of linear search algorithm using frequency count method

```
Algorithm LinearSearch(A[1..n], n, key)
{
 step 1 : loc=-1                                   1
 step 2 : repeat step 3 for i = 1 to n             n+1
 step 3:    if ( A[i]= key)                        n
 step 4:       loc = i                             1
 step 5:       break                               1
            end if
          end repeat
  step 6:  if loc == -1 then                       1
  step 7:     print "Element not found"
  step 8: else                                     1
  step 9:     print " Element found at " loc
  step 10:  stop
}                                                  2n +6
```

What is the purpose of calculating frequency count? Compute the frequency count of the following code fragment.

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        printf("%d",a[i][j]);
```

```
for(i=0;i<n;i++)

    for(j=0;j<n;j++)

        printf("%d",a[i][j]);
```

Derive the Big O notation for $f(n) = n^2+2n+5$.

$f(n) = n^2+2n+5$. let $g(n) = n^2$, $f(n) = Og(n)$ iff. $f(n) <= c*g(n)$

$n^2+2n+5 <= c* n^2$ $n^2+2n+5 - c* n^2 =0$

→

$(1-c) n^2+2n+5=0$

let n =1

(1-c)+2+5 =0

-c +8 =0 => c = 8.

$n^2+2n+5 = O(n^2)$, for all the values of c >=8 and n0 >=1

Polynomial representation using Arrays

A polynomial p(x) is the expression in variable x which is in the form

$ax^n + bx^{n-1} + \ldots. + jx + k$

where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial. An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

1. one is the coefficient
2. other is the exponent

Eg, $10x^2 + 26x$,
here 10 and 26 are coefficients and 2, 1 is its exponential value.
The sign of each coefficient and exponent is stored within the coefficient and the exponent itself Additional terms having equal exponent is possible one. The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent.

Array Representation of Polynomial
Arrays can be used to represent and manipulate polynomials in a single variable.
For eg: A(x)=3x2+2x-4
B(x)=x^8-10x^5-3x^3+1
The polynomial A(x) has 3 terms →$3x^2$, 2x and -4
The coefficients of A(x) are →3, 2 and -4
The exponents are → 2, 1 and 0
A term of a polynomial can be represented as a (coefficient, exponent) pair
For eg: (3,2) represents $3x^2$
A term whose coefficient is non-zero is called a nonzero term. The degree of a polynomial is the largest exponent from among the nonzero terms. Arrange the terms in decreasing order of exponent. This simplifies many of the operations. Use a 2D array to represent coefficient and exponent pair of a polynomial.
Eg: A[][]=$3x^2$+2x-4

3	2
2	1
-4	0

Eg: B[][]=$5x^5$+$3x^4$-$2x^3$+9

5	5
3	4
-2	3
9	0

Result[][]=A[][]+B[][]

5	5
3	4
-2	3
3	2

2	1
5	0

Algorithm POLY_ADDITION
Input: Two polynomial POLY1 and POLY2 with size of the row ptr1 and ptr2.
Output: Sum of two polynomial RESULT with row size, rptr
Data Structure: Polynomial is implemented using array.

```
Steps:
i=0,j=0, rptr=0
ptr1=row size of(POLY1), ptr2= row size of(POLY2)
While(i<ptr1 AND j<ptr2)
    if( POLY1[i][1]=POLY2[j]][1]) then
        RESULT[rptr][0]=POLY1[i][0]+POLY2[j][0]
        RESULT[rptr][1]=POLY1[i][1]
         i=i+1,
         j=j+1,
         rptr=rptr+1
ElseIf( POLY1[i][1]>POLY2[j][1])

  RESULT[rptr][0]=POLY1[i][0]

  RESULT[rptr][1]=POLY1[i][1]

  i=i+1,rptr=rptr+1

Else

  RESULT[rptr][0]=POLY2[j][0]

  RESULT[rptr][1]=POLY2[j][1]

j=j+1,rptr=rptr+1

EndIf

EndWhile

While(i<ptr1)

  RESULT[rptr][0]=POLY1[i][0]

  RESULT[rptr][1]=POLY1[i][1]

  i=i+1,rptr=rptr+1

EndWhile

While(j<ptr2)

  RESULT[rptr][0]=POLY2[j][0]
```

```
    RESULT[rptr][1]=POLY2[j][1]
    j=j+1,rptr=rptr+1
  EndWhile
  Stop
```

The "POLY_ADDITION" algorithm combines two sparse polynomials represented as matrices (POLY1 and POLY2) and outputs the sum in another sparse matrix (RESULT). It uses three pointers to traverse the rows of the input matrices, comparing the exponents of corresponding terms. When the exponents match, it adds the coefficients and stores the result in the output matrix. The algorithm efficiently handles sparse polynomials by avoiding unnecessary zero terms in the result and ensures a compact representation of the sum.

SPARSE MATRIX

It is a special array that contains more number of zero values than the non-zero values for their elements
Eg:

No of zero elements =6
No. of non zero elements = 3
Therefore, it's a sparse matrix

0	1	7
0	0	0
0	0	2

A sparse matrix =2D sparse array. A matrix is said to be a sparse matrix if most of its elements are zero.
A dense matrix is a matrix that is not sparse. The density of a matrix is the percentage of entries that are non-zero. If most of the elements are zero then the occurrence of zero elements in a large array is both a computational and storage inconvenience
Alternative Representations

1. Array representation
2. Dynamic representation

ARRAY Representation (Tuple matrix)

All non-zero elements are stored in another array of triplet

- No of rows in the new array = No. of non – zero elements + 1
- No. of columns in the new array = 3

Triplet contains

- row number of the non-zero element
- column number of the non-zero element
- Value of non-zero element

Triplet can be represented by

- <Row, Col, Element>

Example: Sparse Matrix

No of zero elements =10

No. of non zero elements = 2

Therefore, it's a sparse matrix

Tuple matrix

(0,0) – No of rows in sparse matrix

(0,1) – No of columns in sparse matrix

(0,2) – No of non-zero elements in sparse matrix

1	0	0	0
0	0	0	5
0	0	0	0

3	4	2
0	0	1
1	3	5

Sparse Matrix to tuple matrix

k=1, i=0,j=0

0	0	0
0	0	0
6	0	0
0	0	2

4	3	2
2	0	6
3	2	2

k=1, i=0,j=0, count=0

TUPLE[0][0]=r

TUPLE[0][1]=c

TUPLE[0][2]=count

1. While(i<r)
    1. While(j<c)
        1. If (A[i][j]!=0)
            1. TUPLE[k][0]=i
            2. TUPLE[k][1]=j

3. TUPLE[k][2]=A[i][j]

4. k=k+1, count = count +1

2. EndIf

2. j=j+1

3. EndWhile

2. i=i+1

3. EndWhile

4. TUPLE[0][0]=r

5. TUPLE[0][1]=c

6. TUPLE[0][2]=count

7. End

Sparse Matrix addition

0	**4**	**0**	**0**
0	**0**	**7**	**0**
6	**0**	**0**	**0**

Convert to Sparse Matrix

3	4	3
0	1	4
1	2	7
2	0	6

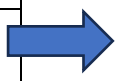
0	**0**	**0**	**3**
1	**0**	**0**	**0**
2	**0**	**0**	**0**

3	4	3
0	1	4
1	2	7

3	4	3
0	3	3
1	0	1
2	0	2

2	0	6

3	4	3
0	3	3
1	0	1
2	0	2



3	4	5
0	1	4
0	3	3
1	0	1
1	2	7
2	0	8

Sparse Matrix addition – case 1

Row value of tuple 1 is less than tuple 2 and there are elements in both tuple matrix. In this case we copy the row in tuple 1 to the sum matrix.

3	4	3
0	1	4
1	2	7
2	0	6

3	4	3
1	0	3
1	2	1
2	0	2

3	4	4
0	1	4
1	0	3
1	2	8
2	0	8

If ((**TUPLE1 [i][0] < TUPLE2 [j][0]**))
SUM [ptr][0] =TUPLE1 [i][0]
SUM [ptr][1] =TUPLE1 [i][1]
SUM [ptr][2] =TUPLE1 [i][2]
i=i+1,
ptr=ptr+1,
elem=elem+1

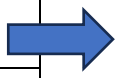
Sparse Matrix addition – case 2

Row value of tuple 2 is less than tuple 1 and there are elements in both tuple matrix. In this case we copy the row in tuple 2 to the sum matrix.

3	4	3
1	0	4
1	2	7

2	0	6

3	4	3
0	1	3
1	2	1
2	0	2



3	4	4
0	1	3
1	0	4
1	2	8
2	0	8

If ((**TUPLE1 [i][0] > TUPLE2 [j][0]**))
SUM [ptr][0] =TUPLE2 [j][0]
SUM [ptr][1] =TUPLE2 [j][1]
SUM [ptr][2] =TUPLE2 [j][2]
j=j+1,
ptr=ptr+1,
elem=elem+1

Sparse Matrix addition – case 3

Row value and column value of both tuple is equal and there are elements in both tuple matrix. In this case we find sum of the values and add to the sum matrix.

3	4	3
0	1	4
1	2	7
2	0	6

3	4	3
0	1	3
1	0	1
2	0	2

3	4	4
0	1	7
1	0	1
1	2	7
2	0	8

If((**TUPLE1 [i][0] = TUPLE2 [j][0]**) AND (**TUPLE1 [i][1] =TUPLE2 [j][1]**))
SUM [ptr][0] =TUPLE1 [i][0]
SUM [ptr][1] =TUPLE1 [i][1]
SUM [ptr][2] =TUPLE1 [i][2] +TUPLE2 [j][2]
Ptr=ptr+1,
i=i+1,
j=j+1,
elem=elem+1

Sparse Matrix addition – case 4

Row value and column value of both tuple is equal and column value of first matrix is smaller than the second one so we copy the first one entirely and then increase the counter of tuple matrix one only

3	4	3
0	1	4
1	0	7
2	0	6

3	4	3
0	1	3
1	2	1
2	0	2

3	4	4
0	1	7
1	0	7
1	2	7
2	0	8

If ((**TUPLE1 [i][0] = TUPLE2 [j][0]**) && (**TUPLE1 [i][1] <TUPLE2 [j][1]**))
SUM [ptr][0] =TUPLE1 [i][0]
SUM [ptr][1] =TUPLE1 [i][1]
SUM [ptr][2] =TUPLE1 [i][2]
i=i+1,
ptr=ptr+1,
elem=elem+1

Sparse Matrix addition – case 4

Row value of both tuple is equal and column value of first matrix is smaller than the second one so we copy the first one entirely and then increase the counter of tuple matrix one only

3	4	3
0	1	4
1	0	7
2	0	6

3	4	3
0	1	3
1	2	1
2	0	2

3	4	4
0	1	7
1	0	7
1	2	7
2	0	8

If ((**TUPLE1 [i][0] = TUPLE2 [j][0]**) && (**TUPLE1 [i][1] <TUPLE2 [j][1]**))
SUM [ptr][0] =TUPLE1 [i][0]
SUM [ptr][1] =TUPLE1 [i][1]
SUM [ptr][2] =TUPLE1 [i][2]
i=i+1,
ptr=ptr+1,
elem=elem+1

Sparse Matrix addition – case 5

Row value of both tuple is equal and column value of second matrix is smaller than the second one so we copy the second one entirely and then increase the counter of tuple matrix two only

3	4	3
0	1	4
1	2	7
2	0	6

3	4	3
0	1	3
1	0	1
2	0	2

3	4	4
0	1	7
1	0	1
1	2	7
2	0	8

If ((**TUPLE1 [i][0] = TUPLE2 [j][0]**) && (**TUPLE1 [i][1] >TUPLE2 [j][1]**))
SUM [ptr][0] =TUPLE2 [j][0]
SUM [ptr][1] =TUPLE2[j][1]
SUM [ptr][2] =TUPLE2[j][2]
j=j+1,
ptr=ptr+1,
elem=elem+1

Sparse Matrix addition – case 6a

After completing an iteration there are elements left in tuple 1 we have to copy it fully to sum matrix

3	4	3
1	0	4
1	1	7
1	2	6

3	4	3
0	0	3
0	1	1
0	2	2

3	4	6
0	0	3
0	1	1
0	2	2
1	0	4
1	1	7
1	2	6

While(i<n1)
SUM[ptr][0]=TUPLE1[i][0]
SUM[ptr][1]=TUPLE1[i][1]
SUM[ptr][2]=TUPLE1[i][2]
i=i+1,
ptr=ptr+1,
elem=elem+1
EndWhile

Sparse Matrix addition – case 6b

After completing an iteration there are elements left in tuple2 we have to copy it fully to sum matrix

3	4	3
0	0	4
0	1	7
0	2	6

3	4	3
1	0	3
1	1	1
1	2	2

3	4	6
0	0	4
0	1	7
0	2	6
1	0	3
1	1	1
1	2	2

```
While(i<n1)
        SUM[ptr][0]=TUPLE1[i][0]
        SUM[ptr][1]=TUPLE1[i][1]
        SUM[ptr][2]=TUPLE1[i][2]
        i=i+1,
        ptr=ptr+1,
        elem=elem+1
EndWhile
```

Stack

Stack is a Linear Data structure. It is an ordered collection of homogeneous data elements where insertion and deletion takes place at one end only. Eg: shunting of trains in a railway yard, Plates on a tray, Stack of books. It follows a Last In First Out order. It can be implemented in two ways

- **Static implementation** -using array. It is a very simple technique, but it is not flexible. The size of the stack has to be declared during program design and after that size cannot be varied
- **Dynamic implementation** –using Linked List. It uses pointers to implement stack. It is more efficient
  - TOP>=SIZE-1 – overflow, stack is full
  - Top= -1 –underflow, stack is empty

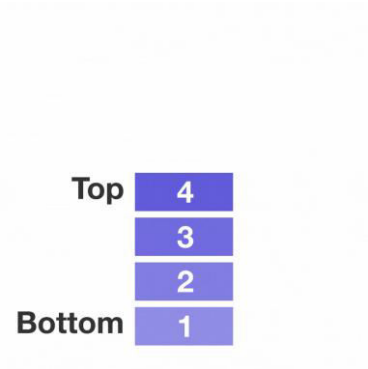
Operations of Stack

1. PUSH
2. POP

Insertion and deletion is at TOP of the stack

An element in a stack : ITEM

The maximum no of elements a stack can accommodate : SIZE



PUSH

Insert an item to stack

Algorithm Push_Array

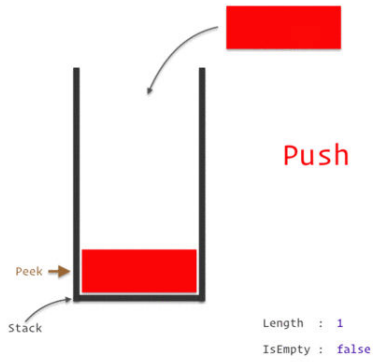
• Input: The new item ITEM to be pushed.

• Output: A stack with newly pushed ITEM at the TOP position

• DS: An array A with TOP as the pointer.

Steps:

1. If TOP>=SIZE-1 then
    a. Print “Stack full”
2. Else
    a. TOP=TOP+1
    b. A[TOP]=ITEM
3. EndIF
4. Stop



POP

Delete an item to stack

Algorithm Pop_Array

• Input: A stack with elements.

• Output: Remove an ITEM from the TOP of stack if it is not empty

• DS: An array A with TOP as the pointer.

Steps

1. If TOP < 0 then
    a. Print " Stack is empty"
2. Else
    a. ITEM=A[TOP]
    b. TOP=TOP-1
3. EndIf
4. Stop

Stack -Status

Algorithm Status_Array

• Input: A stack with elements.
• Output: State whether it is empty or full, available free space and item at TOP
• DS: An array A with TOP as the pointer.

Steps:

1. If TOP<0 then
    a. Print " Stack is empty"
2. Else
    a. If TOP>=SIZE-1 then
        i. Print " Stack is full"
    b. Else
        i. Print "Element at the TOP is", A[TOP]
        ii. Free=(SIZE-1-TOP)/SIZE*100
        iii. Print "Percentage of free stack is",free
    c. Endif
3. EndIf
4. Stop.

Applications of stack

1. String Reversal
2. Evaluation of Arithmetic expression
    1. Infix to postfix conversion
    2. Postfix evaluation
3. Activation Record Management
4. Multiple Stack
5. Tower Hanoi

Evaluation of Arithmetic expression

There are three type of notation for an arithmetic expression. Three types of notations are:

1. Infix notation: operator is written between the operands (A+B)
2. Prefix notation: operator is written before the operands, also called polish notation (+AB)
3. Postfix notation: operator is written after operands, also known as suffix or reverse Polish notation (AB+)

Advantages of using postfix notation:

Human beings are quite used to work with infix notation, but infix is much complex and required to remember set of rules. (E.g. precedence and associativity). Computer have to scan left to right several times to evaluate infix expression. Postfix is much easy to work, and no need for operator precedence and other rules. Computer can evaluate an expression in a single scan

Infix to Postfix

Rules to be remembered during infix to postfix conversion:

1. Parenthesize the expression starting from left to right.
2. During parenthesize the expression, operands associated with operator having higher Precedence are first parenthesized.
3. The sub expression which has been converted into postfix is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Order of Precedence (highest to lowest)

- Exponentiation
- Multiplication/division *, /
- Addition/subtraction +, -
- For operators of same precedence, the left-to-right rule applies:
- A+B+C means (A+B)+C.
- For exponentiation, the right-to-left rule applies:
- A B C means A (B C)

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A B * C – D + E/F	A B C*D – E F/+

You should formulate the conversion algorithm using the following six rules:

1. Scan the input string (infix notation) from left to right. One pass is sufficient.
2. If the next symbol scanned is an operand, it may be immediately appended to the postfix string.
3. If the next symbol is an operator,
   i. Pop and append to the postfix string every operator on the stack that
      a) is above the most recently scanned left parenthesis, and
      b) has precedence higher than or is a right-associative operator of equal precedence to that of the new operator symbol.
   ii. Push the new operator onto the stack.
4. When a left parenthesis is seen, it must be pushed onto the stack.
5. When a right parenthesis is seen, all operators down to the most recently scanned left parenthesis must be popped and appended to the postfix string. Furthermore, this pair of parentheses must be discarded.
6. When the infix string is completely scanned, the stack may still contain some operators. [Why are there no parentheses on the stack at this point?] All the remaining operators should be popped and appended to the postfix string.

ALGORITHM

1. Scan all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is an operand, then immediately append it to the Postfix Expression.
3. If the reading symbol is left parenthesis '(', then Push it onto the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.
5. If the reading symbol is an operator (+, –, *, /), then Push it onto the Stack. However, first, pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix. If an open parenthesis is there on top of the stack then push the operator into the stack.
6. If the input is over, pop all the remaining symbols from the stack and append them to the postfix.

Question. Convert A+B*C/D-E to postfix

Next Symbol	Postfic String	Stack	Rule
A	A		2
+	A	+	3
B	A B	+	2
*	A B	+ *	3
C	A B C	+ *	2
/	A B C *	+/	3
D	A B C * D	+ /	2
-	A B C * D / +	-	3
E	A B C * D / + E	-	2
	A B C * D / + E -		6

Postfix to Infix Conversion

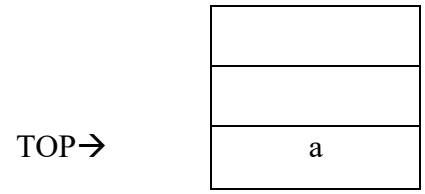
Postfix to infix conversion utilizes a stack to transform expressions from postfix notation to infix. Operand elements are pushed onto the stack, and when operators are encountered, the necessary operands are popped to form infix expressions. This process continues until the entire postfix expression is processed, resulting in the final infix expression obtained from the stack.

Let's take an example a b c / - a d / e - *

Steps

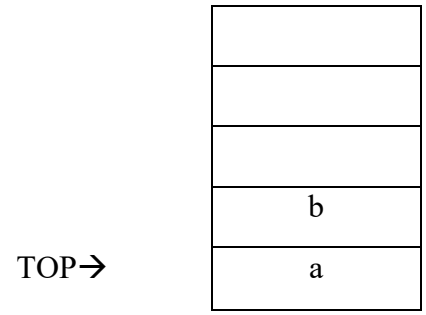
1. 'a' is an operand push it into stack

   a b c / - a d / e - *



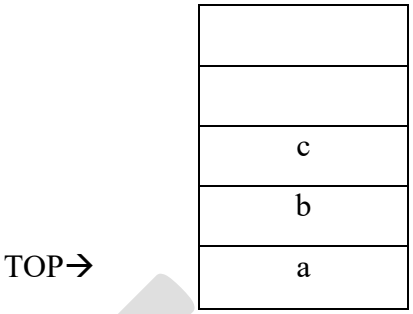
2. 'b' is an operand push it into stack

   a b c / - a d / e - *

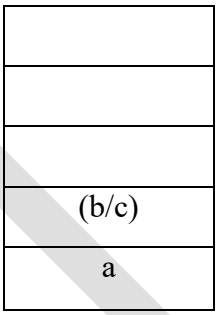


3. ‘c’ is an operand push it into stack

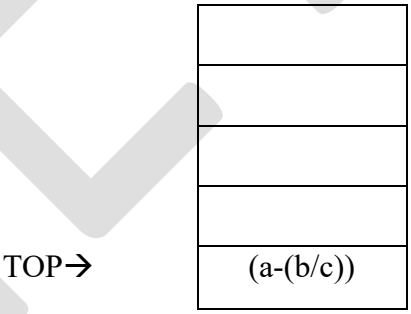
   a b c / - a d / e - *



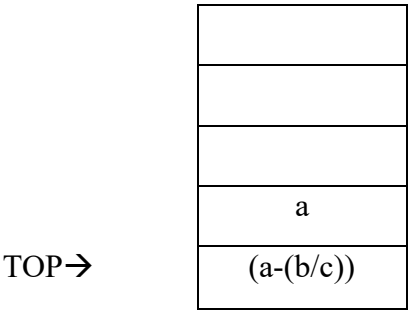
4. ‘/’ is an operator. Pop two operand from stack
   a b c / - a d / e - *
   Operand1=c
   Operand 2=b
   Operator= /
   Push back result after evaluating= (b/c)



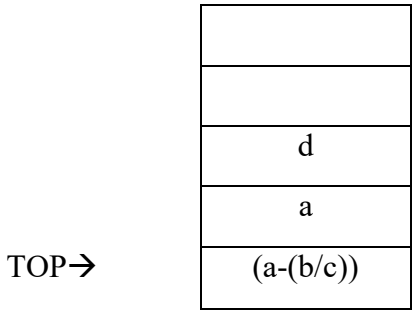
5. ‘-’ is an operator. Pop two operand from stack
   a b c / - a d / e - *
   Operand1=(b/c)
   Operand 2=a
   Operator= -
   Push back result after evaluating= (a-(b/c))



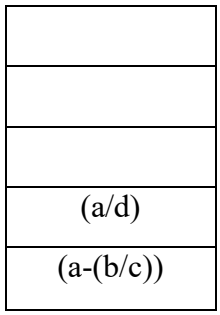
6. ‘a’ is an operand push it into stack
   a b c / - **a** d / e - *



7. ‘d’ is an operand push it into stack
   a b c / - **a d** / e - *

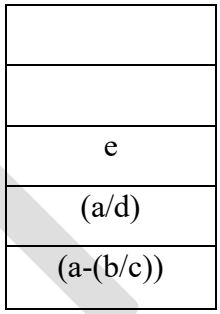


8. '/' is an operator. Pop two operand from stack
   a b c / - a d / e - *
   Operand1=d
   Operand 2=a
   Operator= /
   Push back result after evaluating= (a/d)

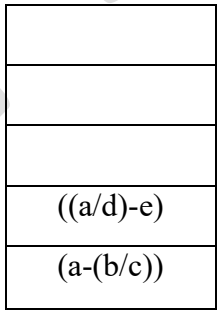


9. 'e' is an operand push it into stack

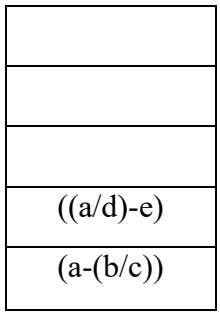
   a b c / - **a d / e - ***



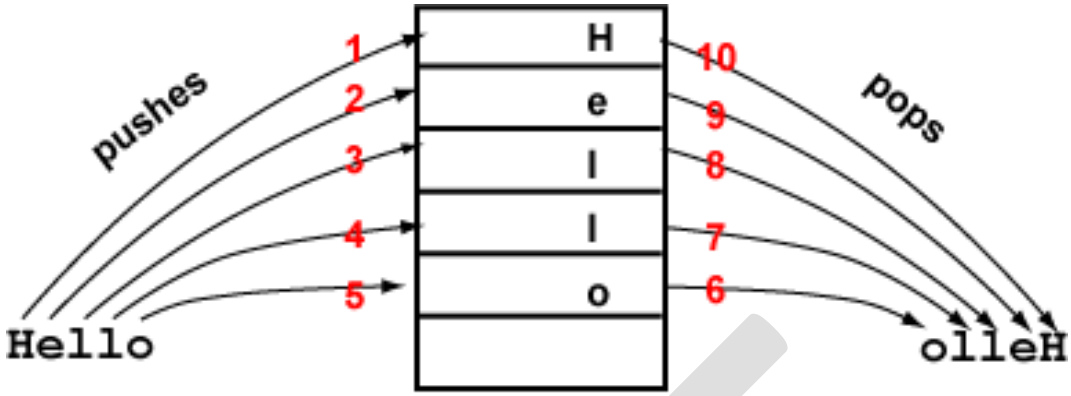
10. '-' is an operator. Pop two operand from stack
    a b c / - a d / e - *
    Operand1=e
    Operand 2=(a/d)
    Operator= -
    Push back result after evaluating= ((a/d)-e)



11. '*' is an operator. Pop two operand from stack
    a b c / - a d / e - *
    Operand1=((a/d)-e)
    Operand 2=(a-(b/c))
    Operator= *
    Push back result after evaluating=
    ((a-(b/c)) *((a/d)-e))



String Reversal



Push the string to the stack and then pop it back and print it to display in reverse order

```
Algorithm ReverseStringWithStack(inputString):
    stack = empty stack  // Initialize an empty stack data structure

    // Step 1: Push each character onto the stack
    for each character in inputString:
        // Step 1.1: Push the current character onto the stack
        stack.push(character)
        // Step 1.2: Move to the next character in the inputString

    reversedString = ""  // Initialize an empty string for the reversed result

    // Step 2: Pop each character from the stack and append it to the reversedString
    while stack is not empty:
        // Step 2.1: Pop the top character from the stack
        poppedCharacter = stack.pop()
        // Step 2.2: Append the popped character to the reversedString
        reversedString += poppedCharacter
        // Step 2.3: Repeat until the stack is empty

    return reversedString
```

Queue

A queue is an ordered list, a linear structure, insertions take place at one end, the rear, and deletions take place at the other end, the front.

Two operations on the queue are

- Insertion (ENQUEUE): Take place at the end called REAR
- Deletion (DEQUEUE): Take place at other end called FRONT

Restrictions on queue

- the first element which is inserted into the queue will be the first one to be removed.
- queues are known as First In First Out (FIFO) lists.

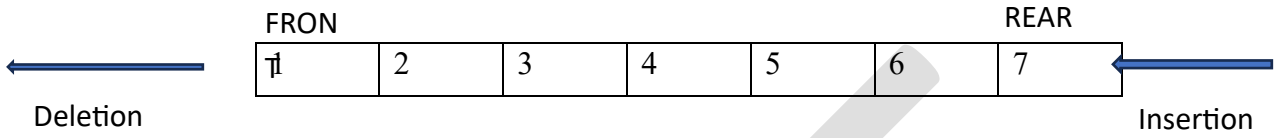
Representation of Queues

Two ways to represent a queue in memory

- Using an Array
- Using Linked List

Queue using Array

One dimensional array, say Q[0 … n-1] can be used to represent a Queue. Two pointers, FRONT and REAR indicate two end of the queue. Insertion to the REAR end and deletion from FRONT end.



Insertion(ENQUEUE)

Initially the queue will be initialized as front= -1 and rear = -1. Before inserting check whether the queue is full or not. If not full, then insert the element to (REAR+1). Make sure that the front always points to the first element by incrementing the front pointer when the first element is inserted.

Algorithm ENQUEUE (ITEM)

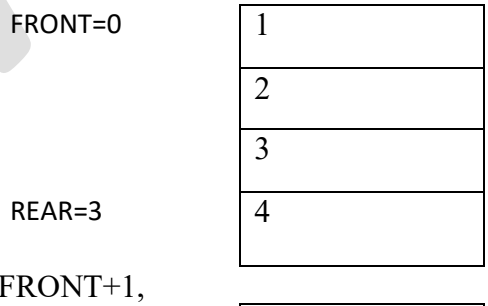
Input: ITEM has to be inserted in to the REAR end of the queue.
Output: Queue is enriched with new element ITEM.
Data Structure: Queue is implemented using array.
Steps:

1. If(REAR=N-1) then
    a. print " Queue is full"
2. Else
    a. If(REAR=-1 AND FRONT=-1) //Queue is empty
        i. Set FRONT=0
    b. EndIf
    c. REAR=REAR+1
    d. Q[REAR]=ITEM
3. EndIf
4. Stop

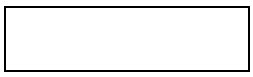
Deletion(DEQUEUE)

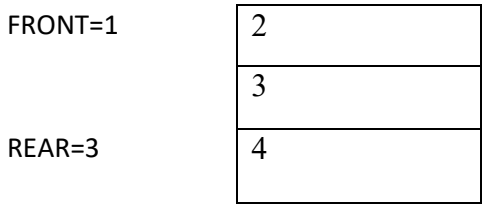
Before deleting, check whether the queue is empty or not. If not empty, then delete the element. Make sure that the FRONT and REAR always points to -1 by decrementing the pointers when the last element is deleted.
Check whether queue is empty.
Rear>=Front and Front!=-1

NOT EMPTY

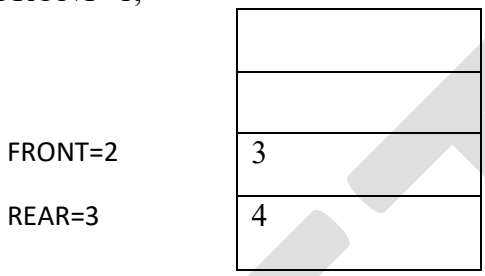


DEQUEUE, Make FRONT =FRONT+1,

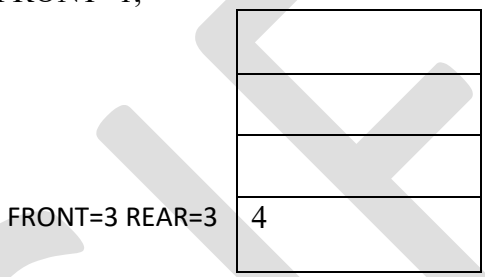




DEQUEUE, Make FRONT =FRONT+1,

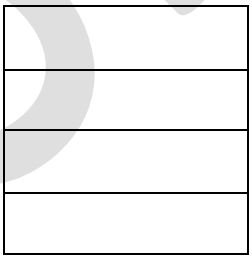


DEQUEUE, Make FRONT =FRONT+1,



DEQUEUE, Make FRONT =FRONT+1,

FRONT=-1, REAR=-1



Algorithm DEQUEUE (ITEM)

Input: ITEM has to be inserted in to the REAR end of the queue.
Output: Queue is enriched with new element ITEM.
Data Structure: Queue is implemented using array.
Steps:

1. If(FRONT=-1)
    a. Print " Queue is empty"
2. Else
    a. ITEM=Q[FRONT]
    b. If(FRONT=REAR)
        i. REAR=-1
        ii. FRONT=-1
    c. Else

i. FRONT=FRONT+1
d. EndIf
3. EndIf
4. Stop

Limitation

Disadvantages of the above implementation is as we delete element from the queue, the queue moves down array. So, the storage space in the beginning is discarded and never used again.

Solution

Keep FRONT always at the zero index position. To maintain front at zero index position, every delete operation would require shifting of all succeeding element in the array by one position

Advantages:

- It enables us to utilize all the empty position in an array i.e. no wastage of space.

Disadvantages:

- Every delete operation requires shift all the succeeding elements in the queue by one position . If the queue is lengthy, this can be very time consuming

Different Queue Structure

1. Circular Queue
2. DEQue
3. Priority Queue

Circular Queue

A circular queue is a data structure that follows the First-In-First-Out (FIFO) principle, but with a circular arrangement of elements in a fixed-size array. It avoids wasting space by reusing freed space from dequeued elements. Circular queues have front and rear pointers that wrap around the array, allowing continuous insertion and deletion operations without the need to shift elements.

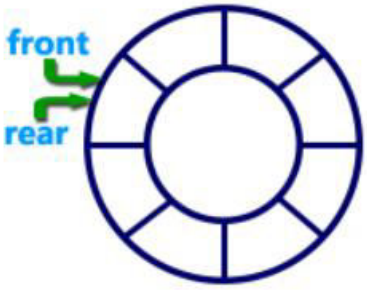
QUEUE IS FULL

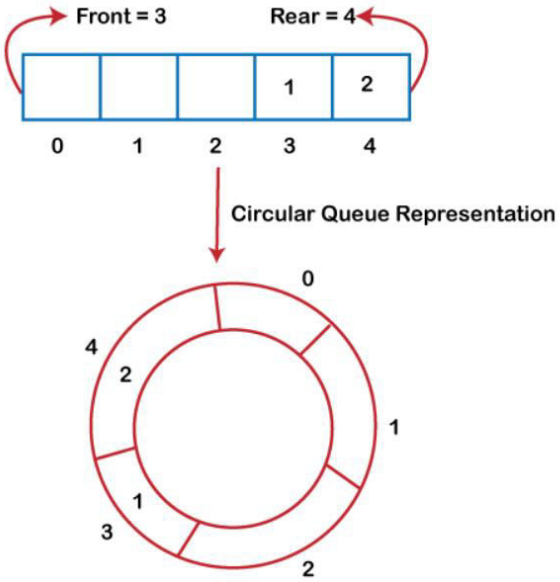
25	30	51	60	80
FRONT				REAR

QUEUE IS FULL (Even after three elements are deleted

			60	80
			FRONT	REAR

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also known as a Ring Buffer.





Physically circular array same as ordinary array, say Q[0..N-1], but Q[0] comes in between Q[1] and Q[N-1] . Both pointers will move in same direction.
Example: If the current pointer is at position i, then the next location will be (i+1) MOD SIZE.

- i.e, if size =8
- Current position = 1 then next position = 2
- Current position = 7 then next position = (7+1) MOD 8 =0

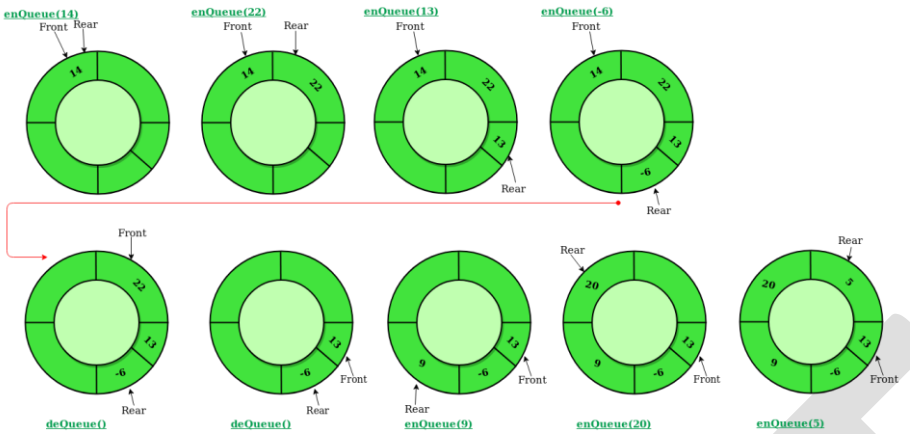
States of the Circular Queue
Circular Queue is empty

- FRONT=-1
- REAR=-1

Circular Queue is full

- FRONT=(REAR+1) MOD SIZE
- i.e 0=(3+1) mod 4 → Queue FULL

Insertion(C-ENQUEUE)
Initially the queue will be initialized as front= -1 and rear = -1. Before inserting check whether the queue is full or not. If not full, then insert the element to (REAR+1) MOD SIZE. Make sure that the front always points to the first element by incrementing the front pointer when the first element is inserted.



Algorithm C-ENQUEUE(ITEM)

Input: An element ITEM to be inserted into the circular queue

Output: Circular queue with the ITEM , if not full

Data Structure: CQ is implemented by using array

Steps:

1. next=(REAR+1) mod SIZE
2. If next=FRONT
    a. Print" Queue is full"
3. Else
    a. REAR=next
    b. Q[REAR]=ITEM
    c. If FRONT=-1 then
        i. FRONT=0
    d. EndIf
4. EndIf
5. Stop

Deletion (C-DEQUEUE)

Algorithm DEQUEUE()

Input: A Queue with n elements

Output: The deleted element is ITEM if the Queue is not empty.

Data Structure: CQ is implemented using array.

Steps:

1. If FRONT=-1 then
    a. Print "Queue is Empty"
2. Else
    a. ITEM=Q[FRONT]
    b. If FRONT=REAR
        i. FRONT=-1
        ii. REAR=-1
    c. Else
        i. FRONT=(FRONT+1) mod SIZE
    d. EndIf
3. EndIf

4. Stop

Consider a circular queue of size=4. Let' see how these operations works in a circular queue.

1. ENCQUEUE(1)

Initially stack is empty

F=-1, R=-1

ENCQUEUE(1)

1			

F=0, R=0

2. ENCQUEUE(2)

ENCQUEUE(2)

1	2		

F=0 R=1

3. ENCQUEUE(3)

ENCQUEUE(3)

1	2	3	

F=0 R=2

4. ENCQUEUE(4)

ENQUEUE(4)

1	2	3	4

F=0 R=3

5. DECQUEUE

DECQUEUE

	2	3	4

F=1 R=3

6. ENCQUEUE(5)

ENCQUEUE(5)= REAR=(REAR+1)%SIZE=(3+1)%4=0

5	2	3	4

R=0 F=1

7. DECQUEUE

DECQUEUE, FRONT=FRONT+1%SIZE= (1+1)%4=2

5		3	4

R=0 F=2

8. ENCQUEUE(6)

ENCQUEUE(6) REAR=(REAR+1)%SIZE=(0+1)%4=1

5	6	3	4

R=1 F=2

9. DECQUEUE

DECQUEUE FRONT=FRONT+1%SIZE= (2+1)%4=3

5	6		4

R=1 F=3

10. DECQUEUE

DECQUEUE FRONT=FRONT+1%SIZE= (3+1)%4=0

5	6		

F=0 R=1

11. DECQUEUE

DECQUEUE FRONT=FRONT+1%SIZE= (0+1)%4=1

	6		

F=1, R=1

12. DECQUEUE

DECQUEUE FRONT=REAR SO SET FRONT=-1, REAR=-1

F=-1, R=-1

Double Ended Queue (Deque)

A Double-Ended Queue (Deque) is a versatile data structure that allows insertion and deletion of elements from both ends. It supports operations such as enqueue and dequeue at both the front and rear, making it suitable for scenarios requiring efficient insertion and removal at both ends. Deques find applications in algorithms like breadth-first search and in scenarios where elements need to be efficiently added or removed from both the front and rear. This is a hybrid linear structure provides all the capabilities of stacks and queues in a single data structure. It does not require the LIFO and FIFO orderings that are enforced by stack and queue.

Insertion→ ←Insertion

1	2	3

Deletion← →Deletion

Representation of deque

Deque operations

The following are the operations of a Deque:

1. addFront(item) or Push_DQ(ITEM) adds a new item to the front of the deque.
2. addRear(item) or Inject(ITEM) adds a new item to the rear of the deque.
3. removeFront() or Pop_DQ() removes the front item from the deque
4. removeRear() or Eject() removes the rear item from the deque.

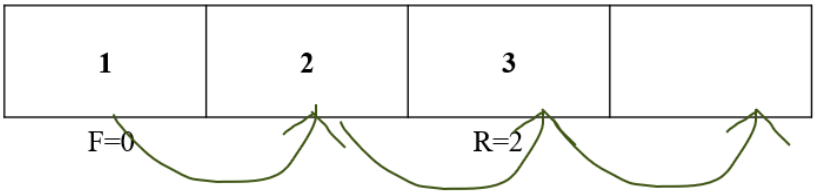
Push_DQ(ITEM) or addFront(int item)

Algorithm:
Input: deque with two pointer front and rear
Output: One item is inserted at front of dequeue
Data Structure: Deque is implemented by using Array
Steps:

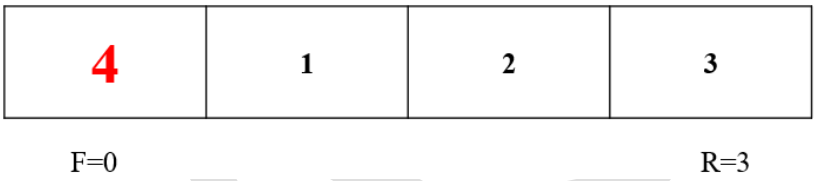
1. if(front==0 and rear=n-1)
    a. Print " Queue is full..'
    b. Exit
2. If(front=-1)
    a. Rear=0
    b. Front=0
    c. Q[front]=item
3. Else if(front>0)
    a. front=front-1

   b. Q[front]=item
4. Else
   a. i=rear
   b. While(i>=front)
      i. Q[i+1]=Q[i]
      ii. i=i-1
   c. End while
   d. Q[front]=item
   e. Rear++
5. End if

The "Push_DQ" or "addFront" algorithm inserts an item at the front of a deque (double-ended queue) implemented using an array. It first checks if the deque is full, and if so, it prints a message and exits. If the deque is not full, it handles cases where the front pointer is at the beginning or greater than 0. In the latter case, it shifts elements to make space for the new item. Finally, it inserts the item at the front and updates the rear pointer accordingly.



i=2(rear)
Q[i+1]=Q[i]// shifting one position till i become 0 results to



Inject(ITEM) or addRear(item)
Algorithm:
Input: deque with two pointer front and rear
Output: One item is inserted at rear of dequeue
Data Structure: Deque is implemented by using Array
Steps:

1. if(front==0 and rear=n-1)
   a. Print " Queue is full.."
   b. Exit
2. If(front=-1)
   a. Rear=0
   b. Front=0
   c. Q[front]=item
3. Else if(rear<n-1)
   a. Rear++

   b. Q[rear]=item
4. Else
   a. i=front
   b. While(i<=rear)
      i. Q[i-1]=Q[i]
      ii. i=i+1\
   c. End while
   d. Q[rear]=item
   e. Front=front-1
5. End if

The "Inject" or "addRear" algorithm inserts an item at the rear of a deque (double-ended queue) implemented using an array. It first checks if the deque is full, printing a message and exiting if so. Then, it handles cases where the deque is initially empty or if there is space at the rear for the new item. In the latter case, it directly adds the item at the rear. If the deque is full, it shifts elements to the right to create space for the new item, updates the rear pointer, and adjusts the front pointer accordingly.

Pop_DQ() or removeFront()

Algorithm:
Input: deque with two pointer front and rear
Output: One item is deleted at front of dequeue
Data Structure: Deque is implemented by using Array
Steps:

1. if(front==-1 and rear=-1)
   a. Print " Queue is empty.."
   b. Exit
2. Item=Q[front]
3. If(front==rear)
   a. Rear=-1
   b. Front=-1
4. Else
   a. Front++
5. End if
6. Return item

The "Pop_DQ" or "removeFront" algorithm deletes an item from the front of a deque (double-ended queue) implemented using an array. It first checks if the deque is empty, printing a message and exiting if so. Then, it retrieves the item at the front of the deque. If the deque has only one element, it updates both the front and rear pointers to -1. Otherwise, it increments the front pointer to remove the item. The algorithm then returns the deleted item.

Eject() or removeRear()

Algorithm:
Input: deque with two pointer front and rear
Output: One item is deleted at rear of dequeue
Data Structure: Deque is implemented by using Array
Steps:

1. if(front==-1 and rear=-1)
   a. Print " Queue is empty.."

    b. Exit
2. Item=Q[rear]
3. if(front==rear)
    a. Rear=-1
    b. Front=-1
4. Else
    a. Rear—
5. End if
6. Return item

The "Eject" or "removeRear" algorithm deletes an item from the rear of a deque (double-ended queue) implemented using an array. It first checks if the deque is empty, printing a message and exiting if so. Then, it retrieves the item at the rear of the deque. If the deque has only one element, it updates both the front and rear pointers to -1. Otherwise, it decrements the rear pointer to remove the item. The algorithm then returns the deleted item.

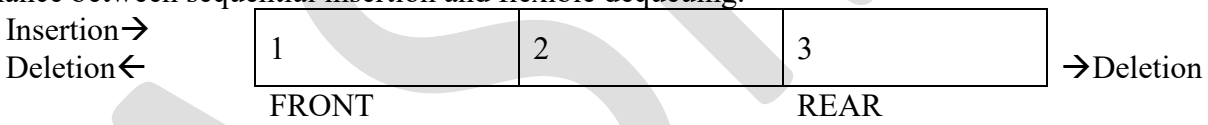
Types of deque

There are two types of deque -

1. Input restricted queue
2. Output restricted queue

Input restricted Queue

In an input-restricted queue, elements can only be inserted at one end (typically the rear), while deletion operations are allowed from both ends. This design is useful when maintaining a specific order of insertion is essential, but elements need to be dequeued from either end based on specific requirements. It provides a balance between sequential insertion and flexible dequeuing.



Output restricted Queue

An output-restricted queue allows deletion operations at one end (usually the front) while supporting insertions from both ends. This design is beneficial when the priority lies in preserving the order of deletions, yet elements can be dynamically inserted from either end as needed. It strikes a balance between sequential deletions and versatile insertions.