DATA STRUCTURE USING C (KCS 301)

By:
Dr. Sunil Kumar
Professor, CSE Dept.
MIET, Meerut

UNIT 2:

Part I: Stacks
Part II: Recursion
Part III: Queues

UNIT 2 – Part I: Stacks

Sunil Kumar, CSE Dept., MIET Meerut

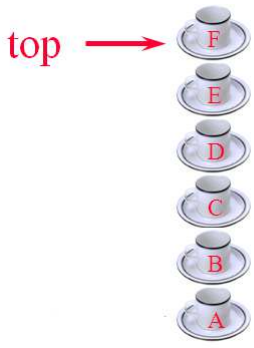
Topics Covered

- Definition of Stack
  - Stack operations: Push & Pop
- Implementation of Stack in C
  - Array Implementation
  - Linked List Implementation
- Applications of Stack:
  - Expression Conversion
    - Infix to Postfix Conversion
    - Infix to Prefix Conversion
  - Expression Evaluation
    - Evaluation of Postfix Expression
    - Evaluation of Prefix Expression

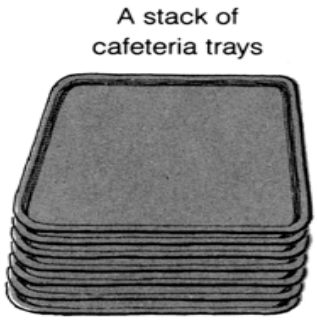
STACKS

- A Stack is a ***linear data structure in which items are added or removed only at one end.***
- Everyday examples of such a data structure are:
  - A Stack of cups
  - A stack of cafeteria trays
  - A stack of coins
- Works on the principle of LIFO
- In particular, the last item to be added to Stack is the first item to be removed
- **STACKS are also called "PILES" AND "PUSH- DOWN"**

5 Sunil Kumar, CSE Dept., MIET Meerut



A stack of cups

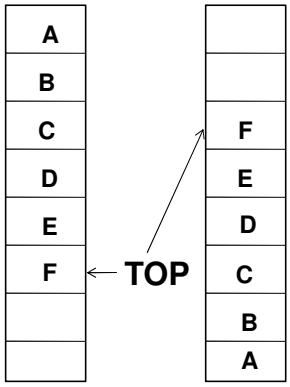


A stack of coins

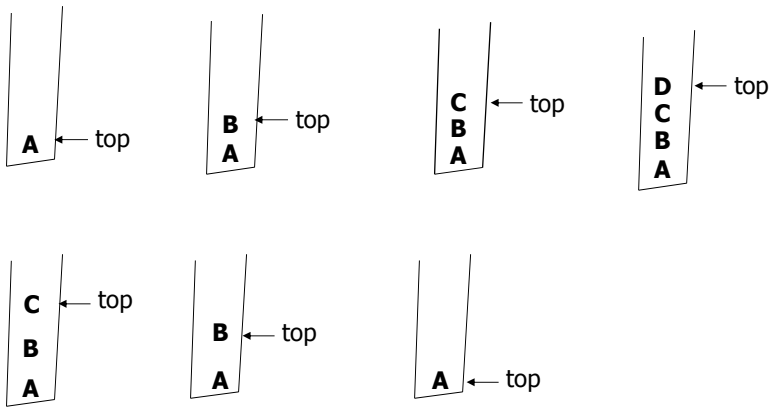
6

Operations On Stack

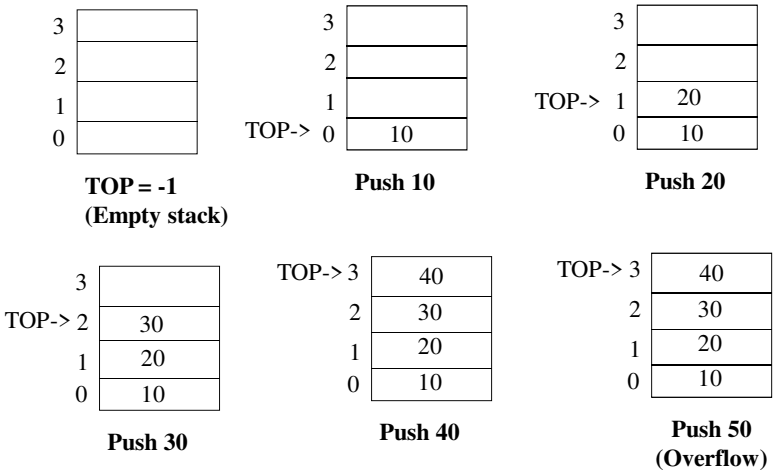
- **PUSH:** is the term to insert an element into a stack
- **POP:** is the term to delete an element from a stack
- **Example:** Suppose the following 6 elements are pushed in order onto an empty stack
- **A, B, C, D, E, F**
- This means:
  - **E cannot be deleted before F is deleted,**
  - **D cannot be deleted before E and F is deleted and so on.**

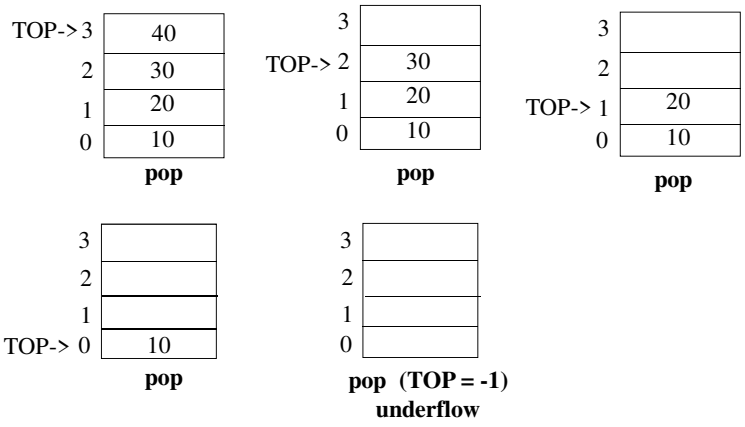


Example: When the elements are **inserted** in the order as **A,B,C,D** then the **size of the stack is 4** and the t**op most element in the stack is D**.
When deletion operation is performed on stack continuously then the order in which the elements are deleted is **D,C,B,A.**



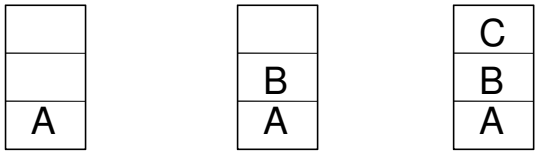
Example : Here Stack Size is 4





POSTPONED DECISIONS

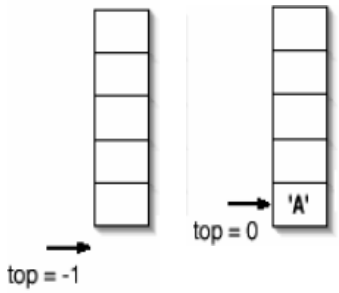
- Stacks are frequently ***used to indicate the order of the processing of data*** when certain steps of the processing must be postponed until other conditions are fulfilled.



Array Implementation of Stack

Algorithm for inserting element into the stack:

Algorithm push( )
1. if top=(SIZE-1)
   then write ('stack overflow')
   else
2. read item or data
3. top←top+1
4. stack[top]← item
5. stop



Array Implementation of Stack...

Algorithm for deleting elements from the stack:

Algorithm pop( )
1. if top=-1
   then write ('stack underflow')
   else
   2. item ←stack[top]
3. top ← top-1
4. stop

Array Implementation of Stack...

Display of Stack:

Algorithm for displaying/printing the contents of stack after push and pop operations.

Algorithm print( )

1. if top = -1
   then write ('stack empty')
2. Repeat for i ← top to 0
   print(stack[i])
3. stop


```
#include<stdio.h>
#include<conio.h>

void push(int);
void pop( );
void display( );
int stack[30], top = -1;
void main()
{
int value, choice;
clrscr( );
while(1)
{
 printf("\n\n***** MENU *****\n");
printf("1.Push\n2.Pop\n3. Display\n4. Exit");
printf("\nEnter your choice: ");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Enter the value to be insert: ");
scanf("%d",&value);
push(value);
break;
case 2: pop( ); break;
case 3: display(); break;
case 4: exit(0);
default: printf("\nWrong choice");
}
}
}
```

```
void push(int value)
{ if(top == SIZE-1)
printf("\nStack is Full!");
else
{
 top++;
stack[top] = value;
printf("\nInsertion success!!!");
}
}
```

```
void pop( )
{
 if(top == -1)
printf("\nStack is Empty!");
else
{
printf("\nDeleted : %d", stack[top]);
top--;
}
}
```



```
void display( )
{
 if(top == -1)
printf("\nStack is Empty!!!");
else
{
int i;
printf(\n"Stack elements are:\n");
for(i=top; i>=0; i--)
printf("%d\n",stack[i]);
}
}
```

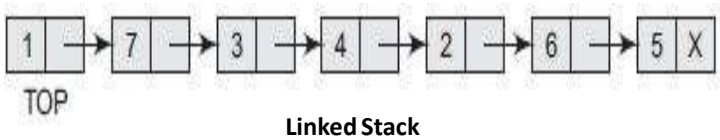
Linked List Implementation of Stack

- Disadvantage of using an array to implement a stack or queue is the **wastage of space**.
- Implementing stacks as a linked lists provides a feasibility on the number of nodes by dynamically growing stacks, as a linked list is a dynamic data structure.
- The stack can grow or shrink as the program demands it.
- A variable **top** always points to top element of the stack.
- ***If top = -1, it specifies stack is empty.***

Linked List Implementation of Stack...

Push Operation:

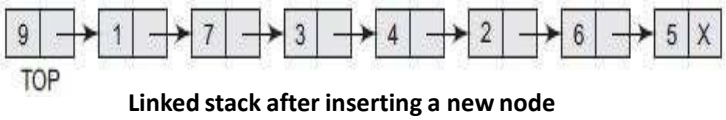
- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.



Linked Stack

Linked List Implementation of Stack...

- To insert an element with value 9, we first check if TOP=-1.
- If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part.
- The new node will then be called TOP.
- If TOP!=-1, then we insert the new node at the beginning of the linked stack and name this new node as TOP.



Linked stack after inserting a new node

20

Algorithm: To Push an element into a linked stack

Step 1: Allocate memory for the new node and name it as NEW

Step 2: SET NEW -> DATA = VALUE

Step 3: IF TOP = NULL
SET NEW -> NEXT = NULL
SET TOP = NEW
ELSE
SET NEW-> NEXT = TOP
SET TOP = NEW
[END OF IF]

Step 4: END

21

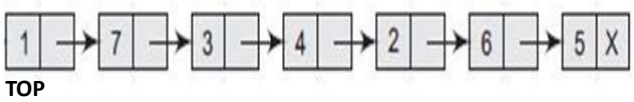
Linked List Implementation of Stack...

Pop Operation:

- The pop operation is used to delete an element into the stack.
- The element is deleted at the topmost position of the stack.
- However, before deleting the value, we must first check if TOP=-1, because if this is the case, then it means that the stack is empty and no more deletions can be done.

Linked List Implementation of Stack...

- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.
- In case TOP!=-1, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes like this.



Linked stack after deleting a node

Algorithm To Pop an element into a linked stack

Step 1: IF TOP= NULL
PRINT "UNDERFLOW"
Goto Step 5
[END OF IF]
ELSE
Step 2: SET PTR = TOP

Step 3: SET TOP = TOP->NEXT

Step 4: FREE PTR

Step 5: END

/* write a c program to implement stack using linked list */

```
#include<stdio.h>          #include<malloc.h>          #include<stdlib.h>
int push( );               int pop( );                int display( );
int choice,i,item;
struct node {
        int data;
        struct node *link;
}*top,*new,*ptr;
main() {                   top=-1;
        printf("\n***Select Menu***\n");
        while(1)  {
                printf("\n1.Push \n2.Pop \n3.Display \n4.Exit");
                printf("\n\nEnter your choice: ");
                scanf("%d",&choice);
                switch(choice)      {
                        case 1:    push( );          break;
                        case 2:    pop( );           break;
                        case 3:    display( );       break;
                        case 4:    exit(0);
                        default: printf("\nWrong choice");
                }/* end of switch */
        }/* end of while */
}/* end of main */
```

```
int push( )
{
    new=malloc(sizeof(struct node));
    printf("\nEnter the value of item: ");
    scanf("%d",&item);
    new->data=item;
    if(top==NULL)
    {
            new->link=NULL;
    }
    else
    {
            new->link=top;
    }
    top=new;
    return;
}/* end of insertion */
```

```
int pop( )
{
        if(top = = NULL)
        {
        printf("\n\nStack is empty");
                return;
        }//if
        else
        {
        printf("\n\nThe deleted element
                is: %d",top->data);
                top=top->link;
        }
        return;
}/* end of pop() */
```



```
int display( )
{
        ptr=top;
        if(top= =NULL)
        {
        printf("\nThe list is empty");
                return;
        }
  printf("\nThe elements in the stack are: ");
        while(ptr!=NULL)
        {
        printf("\n %d",ptr->data);
        ptr=ptr->link;
        }/* end of while */
        return;
}/* end of display*/
```

Applications of Stack

1. Expression Conversion and Evaluation
2. Backtracking
3. Function Call
4. Parenthesis Checking
5. String Reversal
6. Syntax Parsing
7. Memory Management

Applications of Stack...

- Arithmetic Expression Conversion and Evaluation
  - **Infix to Postfix Conversion**
  - **Evaluation of Postfix Expression**
  - **Infix to Prefix Conversion**
  - **Evaluation of Prefix Expression**

Arithmetic Expressions

- Precedence Level
  - **Highest** Exponentiation ( )
  - **Next Highest** Multiplication (*) and Division (/)
  - **Lowest** Addition (+) and subtraction (-)
- Infix Notation

  A + B C – D (G / H) + A
- Polish Notation (**Prefix Notation**)

  + AB - CD (/ GH) + A = + / GHA
- Reverse Polish Notation (**Postfix or Suffix Notation**)

  AB + CD - GH / A +

Arithmetic Expressions Conversion

Note:

- ✓ In infix to postfix conversion, same precedence operators can't remain on to the Stack at the same time, while in infix to prefix conversion, same precedence operators can remain onto the Stack at the same time.

INFIX TO RPN CONVERSION

Algorithm to convert infix expression to RPN:

1. Initialize an empty stack.
2. Repeat the following until the end of the infix expression is reached.
    1. Get next input token (constant, variable, arithmetic operator, left parenthesis, right parenthesis) in the infix expression.
    2. If the token is
        1. A left parenthesis: Push it onto the stack.
        2. A right parenthesis:
            1. Pop and display stack elements until a left parenthesis is on the top of the stack.
            2. Pop the left parenthesis also, but do not display it.
        3. An operator:
            1. While the stack is nonempty and token has lower or equal priority than stack top element, pop and display.
            2. Push token onto the stack.
        4. An operand: Display it.
3. When the end of the infix expression is reached, pop and display stack items until the stack is empty.

(Note: Left parenthesis in the stack has lowest priority)

INFIX TO RPN CONVERSION – DEMO

Expression:

3 * (9 – 2) + 5

↑
S
C
A
N

Output:

1. Scan a token.
   1. 3 is an operand.
   2. Display 3.
2. Scan next token.
   1. * is an operator.
   2. Push * onto stack.
3. Scan next token.
   1. (--- left parenthesis.
   2. Push (onto stack.
4. Scan next token.
   1. 9 is an operand.
   2. Display 9.
5. Scan next token.
   1. – is an operator.
   2. Priority > that of (.
   3. Push – .
6. Scan next token.
   1. 2 is an operand.
   2. Display 2.
7. Scan next token.
   1.) --- right parenthesis.
   2. Pop and push until (is got.

INFIX TO RPN CONVERSION – DEMO

Expression:

Output: **3 9 2 –**

\+ 5

↑
S
C
A
N

8. Scan next token.
   - + is an operator.
   - Priority of + less than that of *
   - Pop * and display.
   - Stack is empty.
   - Push +
9. Scan next token.
   - 5 is an operand.
   - Display.
10. Scan next token.
   - End of expression.
   - Pop all elements and display.

INFIX TO POSTFIX USING STACK

- **EXAMPLE: Convert A * B + C into Postfix Expression**

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B *
5	C	+	A B * C
6			A B * C +

INFIX TO POSTFIX USING STACK...

- **EXAMPLE: A + B * C into Postfix Expression**

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6		+	A B C *
7			A B C * +

INFIX TO POSTFIX USING STACK...

- **EXAMPLE: A * (B + C) into Postfix Expression**

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	)	*	A B C +
8			A B C + *

INFIX TO POSTFIX USING STACK...

- **EXAMPLE: A * B ^ C + D into Postfix Expression**

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

INFIX TO POSTFIX USING STACK...

- EXAMPLE: A * (B + C * D)+ E into Postfix Expression

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
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 +

INFIX TO POSTFIX USING STACK...

- EXAMPLE : CONVERT 2*3/(2-1)+5*3 into Postfix Expression

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	2		2
2	*	*	2
3	3	*	23
4	/	/	23*
5	(	/(	23*
6	2	/(	23*2
7	-	/(-	23*2
8	1	/(-	23*21
9	)	/	23*21-
10	+	+	23*21-/
11	5	+	23*21-/5
12	*	+*	23*21-/53
13	3	+*	23*21-/53
14			23*21-/53*+

40

15

INFIX TO POSTFIX USING STACK...

- EXAMPLE : CONVERT (A+B)^C-(D*E)/F into Postfix Expression

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	(	(	Empty
2	A	(	A
3	+	(+	A
4	B	(+	AB
5	)	Empty	AB+
6	^	^	AB+
7	C	^	AB+C
8	-	-	AB+C^
9	(	-(	AB+C^
10	D	-(	AB+C^D
11	*	-(*	AB+C^D
12	E	-(*	AB+C^DE
13	)	-	AB+C^DE*
14	/	-/	AB+C^DE*
15	F	-/	AB+C^DE*F
16		Empty	AB+C^DE*F/-

41

15

INFIX TO POSTFIX USING STACK...

EXAMPLE : CONVERT A + (B * C - (D / E ↑ F) * G) * H)

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	A	Empty	A
2	+	+	A
3	(	+ (	A
4	B	+ (	A B
5	*	+ (*	A B
6	C	+ (*	A B C
7	-	+ (-	A B C *
8	(	+ (- (	A B C *
9	D	+ (- (	A B C * D
10	/	+ (- (/	A B C * D
11	E	+ (- (/	A B C * D E

42

15

INFIX TO POSTFIX USING STACK...

EXAMPLE : CONVERT A + (B * C - (D / E ↑ F) * G) * H

S.No	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
12	↑	+ (- (/ ↑	A B C * D E
13	F	+ (- (/ ↑	A B C * D E F
14	)	+ (-	A B C * D E F ↑ /
15	*	+ (- *	A B C * D E F ↑ /
16	G	+ (- *	A B C * D E F ↑ / G
17	)	+	A B C * D E F ↑ / G * -
18	*	+ *	A B C * D E F ↑ / G * -
19	H	+ *	A B C * D E F ↑ / G * - H
20		+	A B C * D E F ↑ / G * - H *
21		Empty	A B C * D E F ↑ / G * - H * +

43

15

Convert: (9 - ((3 * 4) + 8) / 4) into Postfix Expression

SNO.	CURRENT SYMBOL	OPERATOR STACK	POSTFIX EXPRESSION
1	(	(	
2	9	(	9
3	–	(-	9
4	(	(-(	9
5	(	(-((	9
6	3	(-((	9 3
7	*	(-((*	9 3
8	4	(-((*	9 3 4
9	)	(-(	9 3 4 *
10	+	(-(+	9 3 4 *
11	8	(-(+	9 3 4 * 8
12	)	(-	9 3 4 * 8 +
13	/	(-/	9 3 4 * 8 +
14	4	(-/	9 3 4 * 8 + 4
15	)		9 3 4 * 8 + 4 / -

44

Infix to Prefix Conversion

- **Given Infix - (A+B)^C-(D*E)/F**
- **Step 1:** Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
- **Step 2:** Obtain the prefix expression of the infix expression.

Infix to Prefix Conversion...

- **Given Infix - (A+B)^C-(D*E)/F**
- **String after reversal –** F/) E*D(^)B+A(
- **String after interchanging right and left parenthesis –** F/ (E*D)^(B+A)

INFIX TO PREFIX USING STACK

- **EXAMPLE: (A+B)^C-(D*E)/F**

S.No	CURRENT SYMBOL	OPERATOR STACK	PREFIX STRING
1	F	Empty	F
2	/	/	F
3	(	/ (	F
4	E	/ (	E F
5	*	/ (*	E F
6	D	/ (*	D E F
7	)	/	* D E F
8	-	-	/ * D E F
9	C	-	C / * D E F
10	^	- ^	C / * D E F
11	(	- ^ (	C / * D E F
12	B	- ^ (	B C / * D E F
13	+	- ^ (+	B C / * D E F
14	A	- ^ (+	A B C / * D E F
15	)	- ^	+ A B C / * D E F
16		-	^ + A B C / * D E F
17		Empty	- ^ + A B C / * D E F

57

14

INFIX TO PREFIX USING STACK

- **EXAMPLE: A * B + C**

S.No	CURRENT SYMBOL	OPERATOR STACK	PREFIX STRING
1	C		C
2	+	+	C
3	B	+	BC
4	*	+ *	BC
5	A	+ *	ABC
6		+	*ABC
			+*ABC

58

8

INFIX TO PREFIX USING STACK

- **EXAMPLE: A + B * C**

S.No	CURRENT SYMBOL	OPERATOR STACK	PREFIX STRING
1	C		C
2	*	*	C
3	B	*	BC
4	+	+	*BC
5	A	+	A*BC
6			+A*BC

59

10

INFIX TO PREFIX USING STACK

- **EXAMPLE: A * (B + C)**

S.No	CURRENT SYMBOL	OPERATOR STACK	PREFIX STRING
1	(	(	Empty
2	C	(	C
3	+	(+	BC
4	B	(+	BC
5	)	(+)	+BC
6	*	*	+ B C
7	A	*	A+ B C
8			*A+ B C

60

11

INFIX TO PREFIX USING STACK

- **EXAMPLE: A - B + C**

S.No	CURRENT SYMBOL	OPERATOR STACK	PREFIX STRING
1	C		C
2	+	+	C
3	B	+	B C
4	-	+ -	B C
5	A	+ -	A B C
6		+	- A B C
7			+-A B C

61

12

INFIX TO PREFIX USING STACK

- **EXAMPLE: A * B ^ C + D**

S.No	CURRENT SYMBOL	OPERATOR STACK	PREFIX STRING
1	D		D
2	+	+	D
3	C	+	C D
4	^	+^	C D
5	B	+^	B C D
6	*	+*	^ B C D
7	A	+*	A ^ B C D
8			+ * A ^ B C D

62

13

INFIX TO PREFIX USING STACK

- **EXAMPLE: A * (B + C* D) + E**

S.No	CURRENT SYMBOL	OPERATOR STACK	PREFIX STRING
1	E		E
2	+	+	E
3	(	+(	E
4	D	+(	D E
5	*	+ (*	D E
6	C	+ (*	C D E
7	+	+ (+	* C D E
8	B	+ (+	B * C D E
9	)	+	+B * C D E
10	*	+ *	+B * C D E
11	A	+	A + B * C D E
12			+ * A + B * C D E

63

14

EVALUATION OF RPN EXPRESSION

Algorithm evaluateRPN(expression)

1. Initialize an empty stack.
2. Do
    1. Get next token (constant, variable, arithmetic operator) in RPN expression.
    2. If token is an operand, push it on the stack.
    3. If token is an operator do the following:
        1. Pop top two values from the stack. (If the stack does not contain two items, report error.)
        2. Apply operator token to these two values.
        3. Push the resulting value onto the stack.
3. Until the end of the expression is encountered.
4. The value of the expression is on the top of the stack (and stack should contain only this value).

EVALUATION OF RPN EXPRESSION – DEMO

Expression:

2 4 * 9 5 + –

↑ SCAN

8

1. Scan a token.
    1. 2 is an operand.
    2. Push 2 onto stack.
2. Scan next token.
    1. 4 is an operand.
    2. Push 4 onto stack
3. Scan next token.
    1. * is an operator.
    2. Pop from stack --- 4.
    3. Pop from stack --- 2.
    4. Apply * on the operands.
    5. Push result 8 onto stack.
4. Scan next token.
    1. 9 is an operand.
    2. Push 9 onto stack.

EVALUATION OF RPN EXPRESSION – DEMO

Expression:

5 + –

↑
S
C
A
N

14

Top →

9
8

1. Scan a token.
   1. 2 is an operand.
   2. Push 2 onto stack.
2. Scan next token.
   1. 4 is an operand.
   2. Push 4 onto stack
3. Scan next token.
   1. * is an operator.
   2. Pop from stack --- 4.
   3. Pop from stack --- 2.
   4. Apply * on the operands.
   5. Push result 6 onto stack.
4. Scan next token.
   1. 9 is an operand.
   2. Push 9 onto stack.
5. Scan next token.
   1. 5 is an operand.
   2. Push 5 onto stack.
6. Scan next token.
   1. + is an operator.
   2. Pop
   3. Pop
   4. Apply +
   5. Push result.
7. Scan next token.
   1. - is an operand.
   2. Pop, pop, apply -, push result.

47

EVALUATION OF RPN EXPRESSION – DEMO

Expression:

–

↑
S
C
A
N

Top →

-6

1. Scan a token.
   1. 2 is an operand.
   2. Push 2 onto stack.
2. Scan next token.
   1. 4 is an operand.
   2. Push 4 onto stack
3. Scan next token.
   1. * is an operator.
   2. Pop from stack --- 4.
   3. Pop from stack --- 2.
   4. Apply * on the operands.
   5. Push result 6 onto stack.
4. Scan next token.
   1. 9 is an operand.
   2. Push 9 onto stack.
5. Scan next token.
   1. 5 is an operand.
   2. Push 5 onto stack.
6. Scan next token.
   1. + is an operator.
   2. Pop
   3. Pop
   4. Apply +
   5. Push result.
7. Scan next token.
   1. - is an operand.
   2. Pop, pop, apply -, push result.

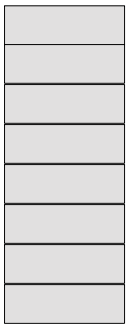
48

EVALUATION OF RPN EXPRESSION – DEMO

Expression:

-6

↑ SCAN



Top →

1. Scan a token.
   1. 2 is an operand.
   2. Push 2 onto stack.
2. Scan next token.
   1. 4 is an operand.
   2. Push 4 onto stack
3. Scan next token.
   1. * is an operator.
   2. Pop from stack --- 4.
   3. Pop from stack --- 2.
   4. Apply * on the operands.
   5. Push result 6 onto stack.
4. Scan next token.
   1. 9 is an operand.
   2. Push 9 onto stack.
5. Scan next token.
   1. 5 is an operand.
   2. Push 5 onto stack.
6. Scan next token.
   1. + is an operator.
   2. Pop
   3. Pop
   4. Apply +
   5. Push result.
7. Scan next token.
   1. - is an operand.
   2. Pop, pop, apply -, push result.
8. Scan next token
   1. End of expression
   2. Pop and display

49

Evaluation of Postfix Expression...

- **Evaluate:** 5 , 6 , 2 , + , * , 12 , 4 , / , -

Symbol scanned	Stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40,12
4	40,12,4
/	40,3
–	37

50

Evaluation of Postfix Expression...

- **Evaluate:** 6 3 2 4 + – *

Symbol scanned	Stack
6	6
3	6, 3
2	6, 3, 2
4	6, 3, 2, 4
+	6, 3, 6
-	6,–3
*	–18

51

Evaluation of Postfix Expression...

- **Evaluate:** 9 3 4 * 8 + 4 / -

Symbol scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

52

Evaluation of Postfix Expression...

- **Evaluate:** 2 3 4 + * 5 *

Symbol scanned	Stack
2	2
3	2, 3
4	2, 3, 4
+	2, 7
*	14
5	14, 5
*	70

Evaluation of Postfix Expression...

- **Evaluate:** 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Symbol scanned	Stack
6	6
2	6 2
3	6 2 3
+	6 5
-	1
3	1 3
8	1 3 8
2	1 3 8 2
/	1 3 4
+	1 7
*	7
2	7 2
↑	49
3	49 3
+	52

Evaluation of Prefix Expression...

- Evaluate: **- * + 4 3 2 5**

Symbol scanned	Stack
5	5
2	5, 2
3	5, 2, 3
4	5, 2, 3, 4
+	5, 2, 7
*	5, 14
–	9

55

Evaluation of Prefix Expression...

- Evaluate: **+ 3 + 4 / 4 20**

Symbol scanned	Stack
20	20
4	20, 4
/	5
4	5, 4
+	9
3	9, 3
+	12

56

Evaluation of Prefix Expression...

- Evaluate: **- * + 4 3 2 15**

Symbol Scanned	Stack
15	15
2	15, 2
3	15, 2, 3
4	15, 2, 3, 4
+	15, 2, 7
*	15, 14
–	–1

57

UNIT 2 - Part II: Recursion

Table of Contents

Iteration Vs Recursion

Iteration	Recursion
It is a process of executing a statement or a set of statements repeatedly, until some specific condition is met.	Recursion is the technique of defining anything in terms of itself.
It uses four clear cut steps: 1. initialization, 2. condition, 3. execution, 4. updation.	There must be an exclusive if statement inside the recursive functions, specifying stopping condition. There are base case and recursive case in recursive function.
Any recursive problem can be solved iteratively.	Not all problems have recursive solution.
It is more efficient in terms of memory utilization and execution speed.	Recursion is generally a worse option to go simple problems. It is less efficient in terms of memory utilization and execution speed.

Principle of Recursion

- **A definition is recursive if it is defined in terms of itself.**
- **Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.**
- The **process is used for repetitive computations** in which each action is stated in terms of a previous result.

Need of Recursion

- Recursion makes solving problems easier by **breaking them into smaller sub problems** thereby making it easier to understand the problem.
- As such, **not all problems can be broken down into smaller sub problems** so that they could be solved recursively.
- The most important advantage is that **recursion makes algorithm and its implementation simple and compact.**
- It increases programmer efficiency.

Example: Factorial Function

- In general, we can express the factorial function as follows:

  n! = n * (n-1)!
- Is this correct? Well… almost.
- The factorial function is only defined for *positive* integers. So we should be a bit more precise:

  n! = 1 (if n is equal to 0 or 1)

  n! = n * (n-1)! (if n is larger than 1)

Factorial Function

The C equivalent of this definition:

```
int fac(int numb){
    if(numb<=1)
        return 1;
    else
        return numb* fac(numb-1);
}
```

✓ *recursion* means that a function calls itself

Factorial Function

- **Assume the number typed is 3, that is, numb=3.**

```
fac(3) :

3 <= 1 ?              No.
fac(3) = 3 * fac(2)
   fac(2) :
      2 <= 1 ?              No.
      fac(2) = 2 * fac(1)
         fac(1) :
            1 <= 1 ?    Yes.
            return 1
      fac(2) = 2 * 1 = 2
      return fac(2)
fac(3) = 3 * 2 = 6
return fac(3)

fac(3)  has the value 6
```

```
int fac(int numb){
   if(numb<=1)
       return 1;
   else
       return numb * fac(numb-1);
}
```

Factorial Function

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. **Compare the recursive solution with the iterative solution:**

Recursive Solution

```
int fac(int numb){
   if(numb<=1)
     return 1;
   else
     return numb*fac(numb-1);
}
```

Iterative Solution

```
int fac(int numb){
  int product=1;
  while(numb>1){
    product *= numb;
    numb--;
  }
  return product;
}
```

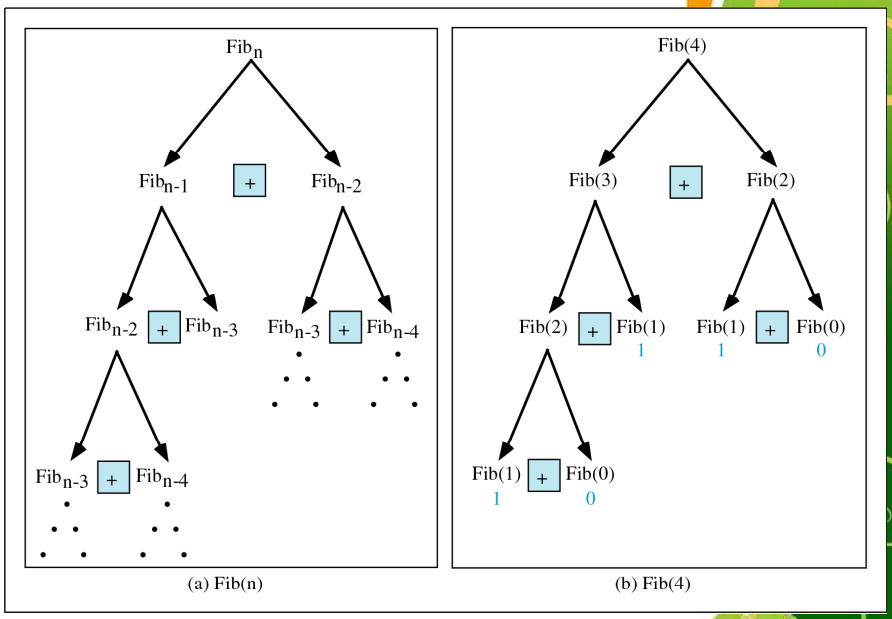
Example: Fibonacci Numbers

- Fibonacci Numbers:
  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`
  where each number is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(number) = F(number-1)+ F(number-2);`

Example: Fibonacci Numbers...

```
//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well

int fib(int number)
{
   if (number = = 0) return 0;
   if (number = = 1) return 1;
   return (fib(number-1) + fib(number-2));
}
```

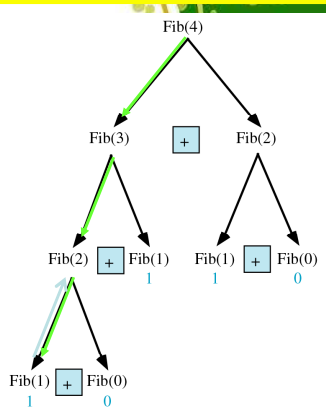


Trace a Fibonacci Number

- Assume the input number is 4, that is, num=4:

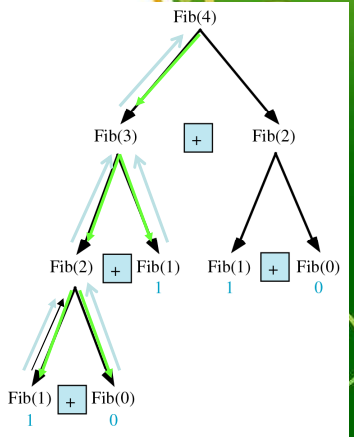
```
fib(4):
    4 == 0 ? No;    4 == 1?  No.
    fib(4) = fib(3) + fib(2)
    fib(3):
        3 == 0 ? No; 3 == 1? No.
        fib(3) = fib(2) + fib(1)
        fib(2):
            2 == 0? No; 2==1? No.
            fib(2) = fib(1)+fib(0)
            fib(1):
                1== 0 ? No; 1 == 1? Yes.
                 fib(1) = 1;
                 return fib(1);
```

```
int fib(int num)
{
  if (num == 0) return 0;
  if (num == 1) return 1;
  return
    (fib(num-1)+fib(num-2));
}
```



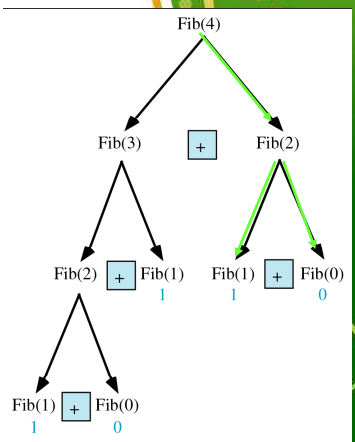
Trace a Fibonacci Number

```
        fib(0):
            0 == 0 ?  Yes.
            fib(0) = 0;
            return fib(0);
    fib(2) = 1 + 0 = 1;
    return fib(2);
fib(3) = 1 + fib(1)
        fib(1):
        1 == 0 ? No; 1 == 1? Yes
        fib(1) = 1;
        return fib(1);
 fib(3) = 1 + 1 = 2;
return fib(3)
```



Trace a Fibonacci Number

```
fib(2):
    2 == 0 ? No; 2 == 1?    No.
    fib(2) = fib(1) + fib(0)
    fib(1):
       1== 0 ? No; 1 == 1?  Yes.
        fib(1) = 1;
        return fib(1);
    fib(0):
        0 == 0 ?   Yes.
        fib(0) = 0;
        return fib(0);
    fib(2) = 1 + 0 = 1;
    return fib(2);
 fib(4) = fib(3) + fib(2)
        = 2 + 1 = 3;
 return fib(4);
```



Example: Fibonacci Number

```
//Calculate Fibonacci numbers iteratively
//much more efficient than recursive solution

int fib(int n)
{
   int f[100];
   f[0] = 0;
   f[1] = 1;
   for (int i=2; i<= n; i++)
       f[i] = f[i-1] + f[i-2];
   return f[n];
}
```


Example: Binary Search

- Search for an element in an array
  - Sequential search
  - Binary search
- Binary search
  - Compare the search element with the middle element of the array
  - If not equal, then apply binary search to half of the array (if not empty) where the search element would be.

Binary Search with Recursion

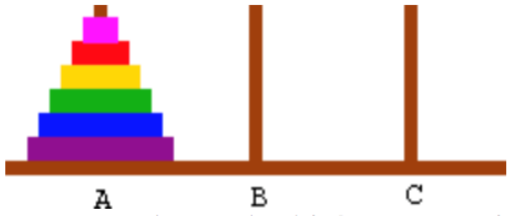
```
// Searches an ordered array of integers using recursion
int bsearchr(int data[],          // input: array
             int first,           // input: lower bound
             int last,            // input: upper bound
             int value            // input: value to find
        )// output: index if found,  otherwise return -1

{
  int middle = (first + last) / 2;
  if (data[middle] == value)
    return middle;
  else if (value < data[middle])
    return bsearchr(data, first, middle-1, value);
  else
    return bsearchr(data, middle+1, last, value);
}
```


Binary Search without Recursion

```
// Searches an ordered array of integers
int bsearch(int data[8];   // input: array
            int size;      // input: array size
            int value;     // input: value to find
           )               // output: if found,return
                           // index; otherwise, return -1
       {
       int first, last, upper;
           first = 0;
           last = size - 1;
       while (true) {
            middle = (first + last) / 2;
            if (data[middle] == value)
                 return middle;
            else if (value < data[middle])
              last = middle - 1;
            else
              first = middle + 1;
      }
```

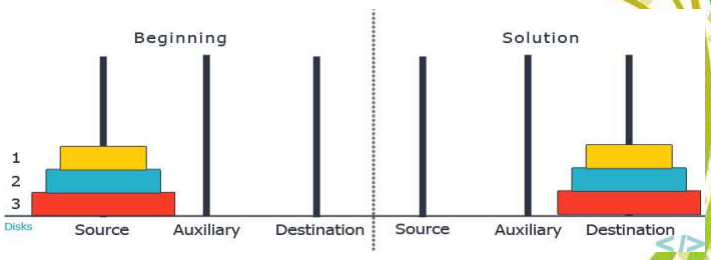
Example: Tower of Hanoi

- Tower of Hanoi is a **mathematical game or puzzle**.
- It consists of **three rods/pegs** and a **number of disks of different sizes** which can slide onto any rod.
- The **puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape**.



Example: Tower of Hanoi…

- The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:
  1. Only one disk can be moved at a time.
  2. No larger disk may be placed on top of a smaller disk.



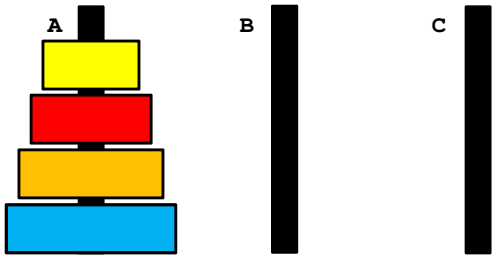
Example: Tower of Hanoi…

- The puzzle can be played with **any number of disks**.
- The **minimum number of moves required to solve a Tower of Hanoi puzzle is**

$$= 2^n - 1$$

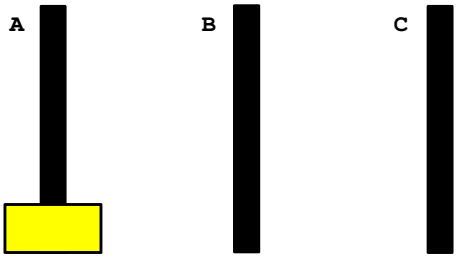
where n is the number of disks.

Towers of Hanoi



Towers of Hanoi

- n = 1

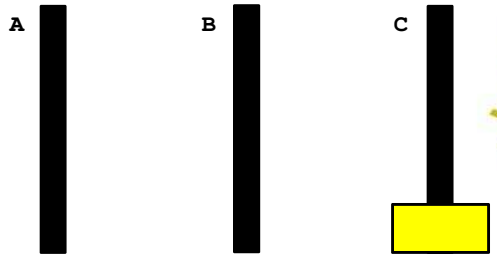


- move disk from A to C

Towers of Hanoi

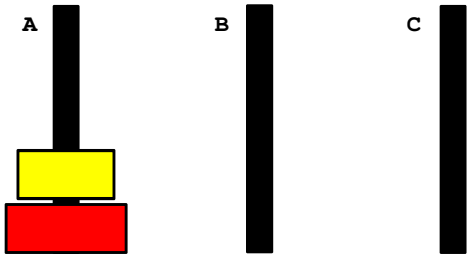
- n = 1

A→C



Towers of Hanoi

- n = 2

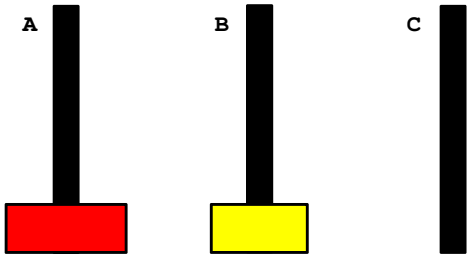


- move disk from A to B

Towers of Hanoi

- n = 2

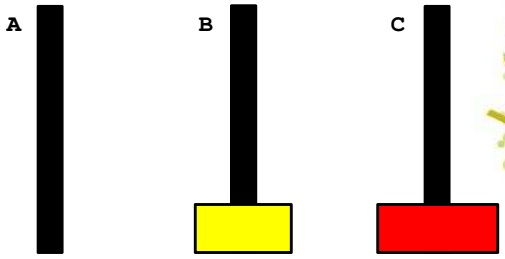
A→B



Towers of Hanoi

- n = 2

A→C

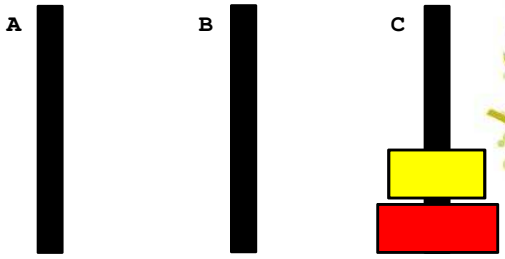


- move disk from B to C

Towers of Hanoi

- n = 2

B→C

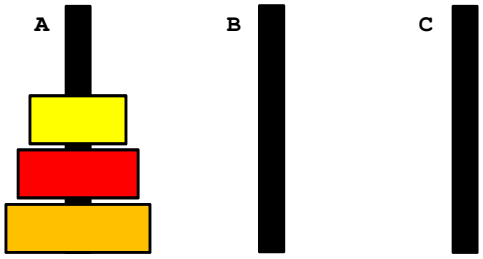


Tower of Hanoi(N= 2)

1. Disk 1 moved from A → B
2. Disk 2 moved from A → C
3. Disk 1 moved from B → C

Towers of Hanoi

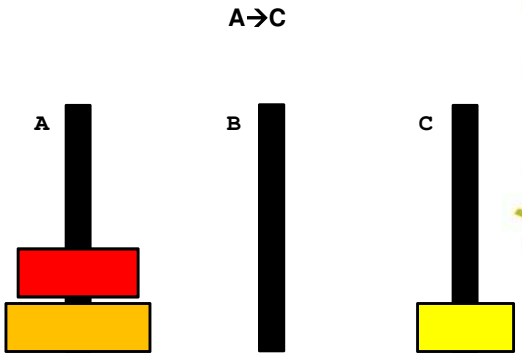
- n = 3



- move disk from A to C

Towers of Hanoi

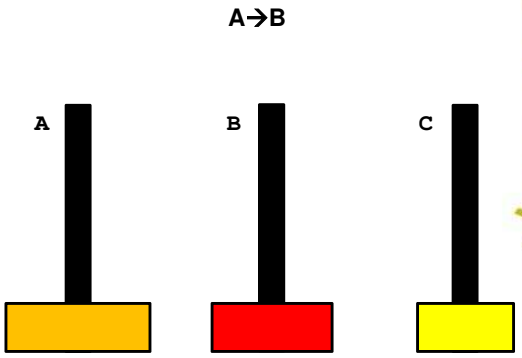
- n = 3



- move disk from A to B

Towers of Hanoi

- n = 3

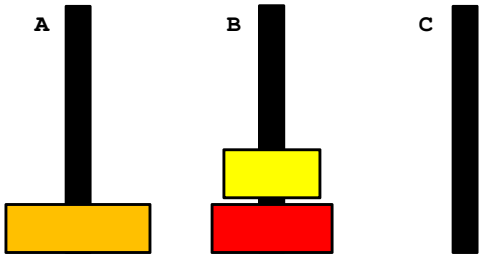


- move disk from C to B

Towers of Hanoi

- n = 3

C→B

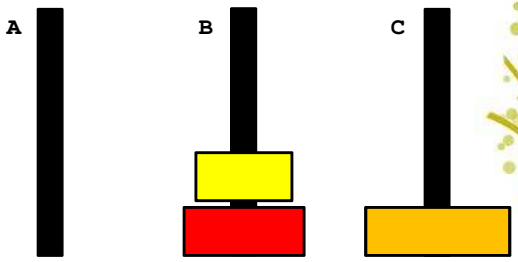


- move disk from A to C

Towers of Hanoi

- n = 3

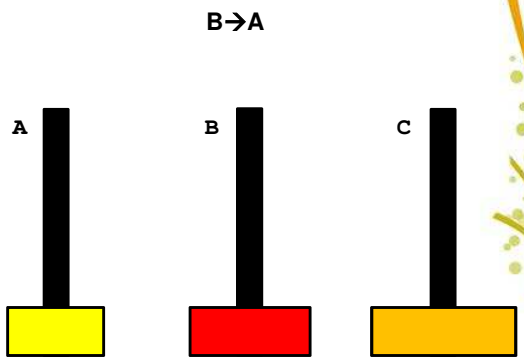
A→C



- move disk from B to A

Towers of Hanoi

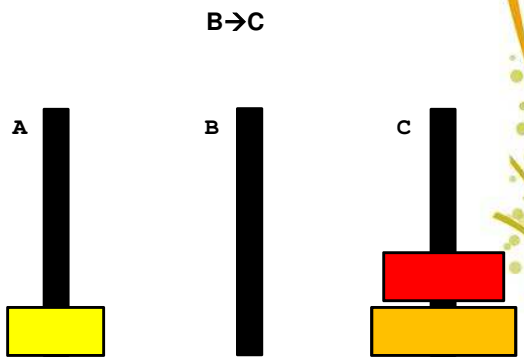
- n = 3



- move disk from B to C

Towers of Hanoi

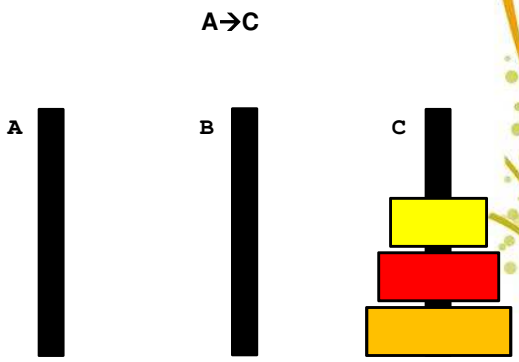
- n = 3



- move disk from A to C

Towers of Hanoi

- n = 3

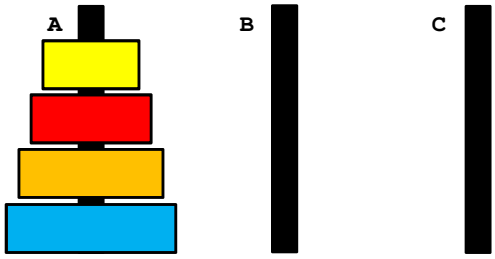


Tower of Hanoi(N= 3)

1. Disk 1 moved from A → C
2. Disk 2 moved from A → B
3. Disk 1 moved from C → B
4. Disk 3 moved from A → C
5. Disk 1 moved from B → A
6. Disk 2 moved from B → C
7. Disk 1 moved from A → C

Towers of Hanoi

- n = 4

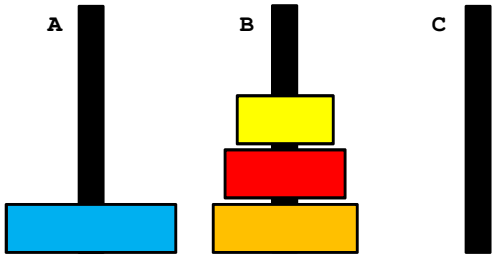


- move (n – 1) disks from A to B using C

Towers of Hanoi

- n = 4

A(N-1) Disks→B

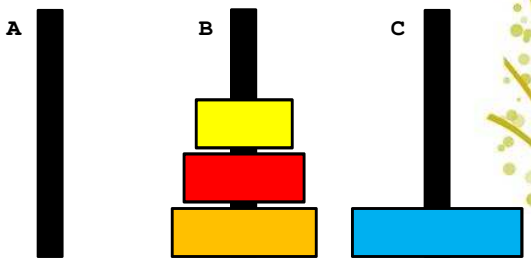


- move disk from A to C

Towers of Hanoi

- n = 4

A→C

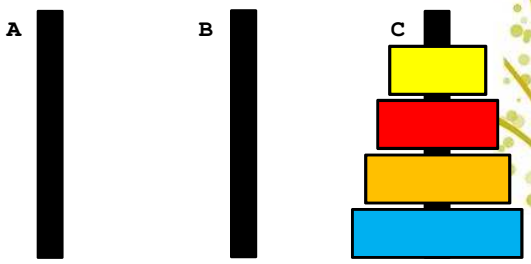


- move (n – 1) disks from B to C using A

Towers of Hanoi

- n = 4

B(N-1) Disks→C



Tower of Hanoi(N= 4)

1. A→ B
2. A→ C
3. B→ C
4. A→ B
5. C→ A
6. C→ B
7. A→ B
8. A→ C
9. B→ C
10. B→ A
11. C→ A
12. B→ C
13. A→ B
14. A→ C
15. B→ C

Towers of Hanoi

- **Base Case *n* = 1**
  1. move disk from A to C
- **Recursive case**
  1. move ($n - 1$) disks from A to B
  2. move 1 disk from A to C
  3. move ($n - 1$) disks from B to C

Algorithm

Let's call the three pegs / rods
BEG(**Source**), AUX(**AUXiliary**) & END(**Destination**).

1) Move the **top N–1 disks** from the **Source to AUXiliary** tower
2) Move the **Nth disk** from **Source to Destination** tower.
3) Move the **N–1 disks** from **AUXiliary tower to Destination** tower.
   Transferring the top N–1 disks from Source to AUXiliary tower can again be thought of as a fresh problem and can be solved in the same manner.

Algorithm

ALGORITHM:

TOH(N, BEG, AUX, END)

1. If N=1 then,
   a. Write BEG→ END
   b. Return.
2. CALL TOH (N-1, **BEG**, END, **AUX**)
3. Write BEG → END
4. CALL TOH (N-1, **AUX**, BEG, **END**)
5. Return.

Tower of Hanoi Program in C

```
#include<stdio.h>
void TOH(int n,char x,char y,char z)
{
if(n>0)
{
TOH(n-1,x,z,y);
printf("\n%c to %c",x,y);
TOH(n-1,y,x,z);
}
}
```

```
int main()
{
int n=3;
TOH(n,'A','B','C');
}
```

Tail Recursion

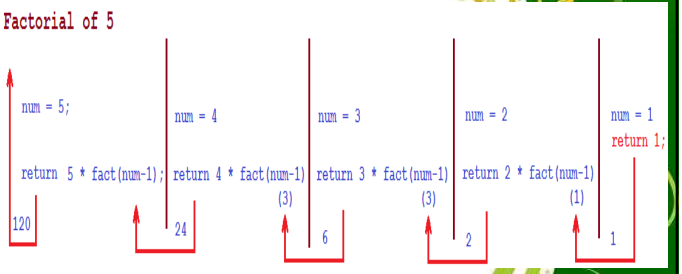
What is tail recursion?

- A recursive function call is **tail recursive when recursive call is the last thing executed by the function.**
- In other words, a recursive function call is said to be **tail recursive if there is nothing to do after the function returns except return its value.**

Contd…

/* non tail recursive example of the factorial function in C*/

```
#include<stdio.h>
void main()
{
int n, c;
printf(“Enter the number: ”);
scanf(“%d”,&n);
c=fact(n);
printf("%d",c);
}
fact(n)
{
if ((n == 0|| n==1)
return 1;
return n * fact(n - 1);
}
```

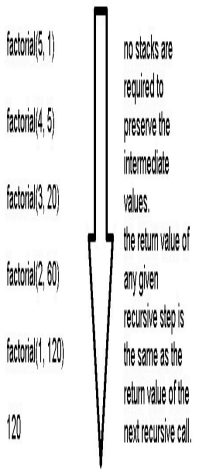


Contd…

// A tail recursive function to calculate factorial

```
#include<stdio.h>
void main( )
{
int n, c;
printf(“Enter the number: ”);
scanf(“%d”,&n);
c=factorial(n, 1);
printf(“Factorial: %d”, c);
}
```

```
int factorial(int n, int f)
{
if (n == 0|| n==1)
{
return f;
}
else
{
f=f*n;
return factorial(n - 1, f);
}
}
```



Contd…

- **Why do we care?**
- The **tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler.**
- **Tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use.**

Removal of Recursion

Removal of Recursion

The function which call itself (In function body) again and again is known as recursive function. This function will call itself as long as the condition is satisfied

This recursion can be removed through Iteration.

Removal of Recursion…

A simple program of factorial through recursion:

```
/*Find the Factorial of any Number*/
#include<stdio.h>
main()
{
int n, value;
printf("Enter the number");
scanf("%d",&n);
if(n<0)
printf("No factorial of negative number");
else
if(n==0)
printf("Factorial of zero is 1");
```



```
else
{
  value=factorial(n); /*function for factorial of number*/
  printf("Factorial of %d= %d",n,value);
}
}
factorial (int k)
{
int fact=1;
if(k>1)
fact=k*factorial(k-1);      /*recursive function call*/
return (fact);
}
```

- Same thing can be replaced with Iteration as

/*Find the factorial of any number*/

```
#include<stdio.h>
main()
{
int n, value;
printf("Enter the number");
scanf("%d",&n);
if(n<0)
printf("No factorial of negative number");
else
if(n==0)
printf("Factorial of zero is 1");
```



```
else
{
  value=factorial(n); /*function for factorial of number*/
  printf("Factorial of %d= %d",n,value);
}
}
int factorial(int no)
{
int i,fact=1;
for(i=no;i>1;i--)
fact=fact*i;
return fact;
}
```

Trade-off between Recursion and Iteration

BASIS FOR COMPARISON	RECURSION	ITERATION
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.
Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.

Trade-off between Recursion and Iteration…

BASIS FOR COMPARISON	RECURSION	ITERATION
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

UNIT 2 - Part III: Queues

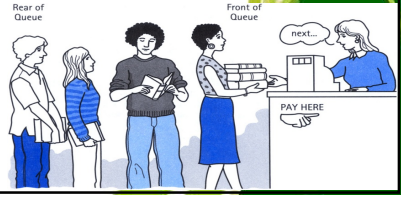
Table of Contents

QUEUE

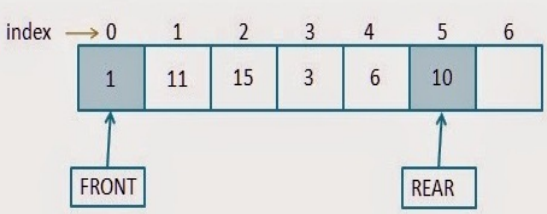
Queue

- Ordered collection of homogeneous elements
- Non-primitive linear data structure.
- A new element is added at one end called **Rear End** and the existing elements are deleted from the other end called **Front End**.
- This mechanism is called First-In-First-Out **(FIFO).**

e.g. People standing in Queue for Movie Ticket



Queue in Data Structure



Operations on Queue

- **add** (enqueue): Add an element to the back
- **remove** (dequeue): Remove the front element.
- **peek ( )**: Examine the element at the front.
- **isEmpty( ):** Check whether queue is empty or not
- **isFull( ) :** Check whether queue is full or not
- **Display( )**: Display the elements of the queue

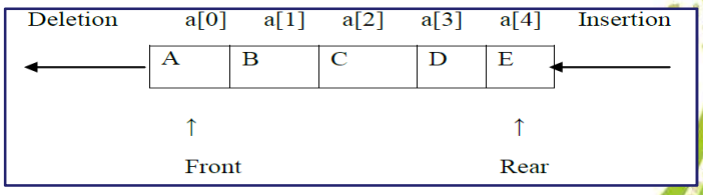
Elements of Queue

- **Front End:**

  This end is used for deleting an element from a queue. Initially front end is set to -1. Front end is incremented by one when a new element has to be deleted from queue.

- **Rear End:**

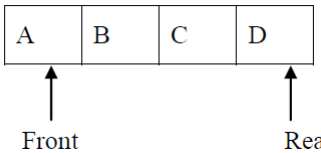
  This end is used for inserting an element in a queue. Initially rear end is set to -1. rear end is incremented by one when a new element has to be inserted in queue.



'Queue Full(Overflow)' Condition

- **Queue Full(Overflow):**
  - Inserting an element in a queue which is already full is known as Queue Full condition **(Rear = Size-1).**
  - When the queue is fully occupied and enqueue() operation is called queue overflow occurs.

- **Example: Queue Full:**
  - Before inserting an element in queue 1st check whether space is available for new element in queue. This can be done by checking position of rear end. Array begins with 0th index position & ends with Size-1 position. If numbers of elements in queue are equal to size of queue i.e. if rear end position is equal to Size-1 then queue is said to be full. **Size of queue = 4**



'Queue Empty(Underflow)' Condition

- **Queue Empty:**
  - Deleting an element from queue which is already empty is known as Queue Empty condition (**Front = Rear = -1**)
  - When the queue is fully empty and dequeue() operation is called queue underflow occurs.

- **Queue Empty:**
  - Before deleting any element from queue check whether there is an element in the queue. If no element is present inside a queue & front & rear is set to -1 then queue is said to be empty.
  - **Size of queue = 4**
  - Front = Rear = -1

Types of Queues

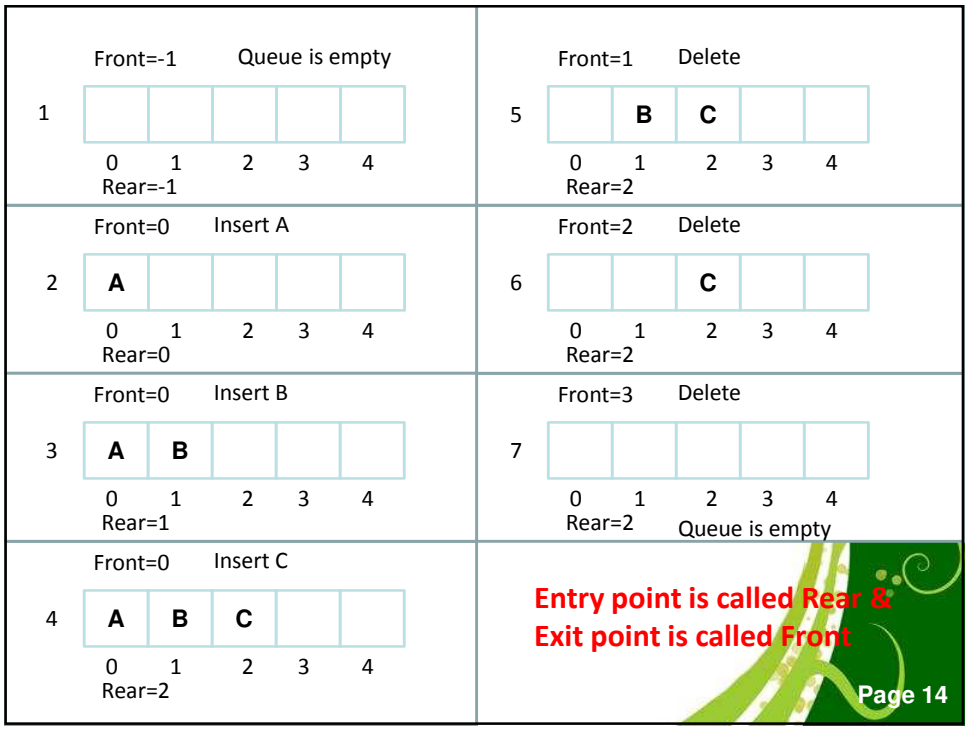
1. **Queue or Linear Queue or Simple Queue**
2. **Circular Queue**
3. **Dequeue (Double Ended Queue)**
4. **Priority Queue**

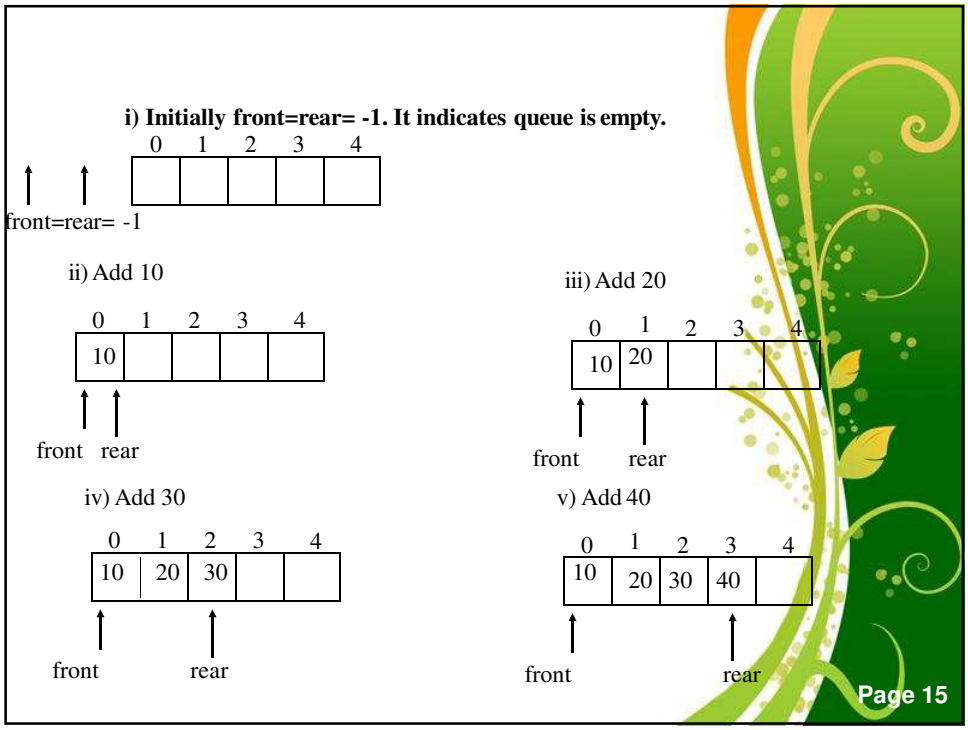
Implementation of Queues

1.Using an array

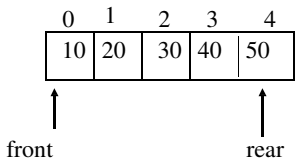
2.Using linked list

Array Implementation of Queue

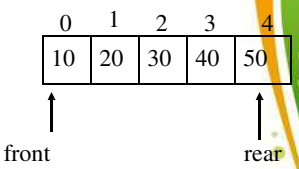

1
Front=-1
Queue is empty
0 1 2 3 4
Rear=-1
2
Front=0
Insert A
A
0 1 2 3 4
Rear=0
3
Front=0
Insert B
A B
0 1 2 3 4
Rear=1
4
Front=0
Insert C
A B C
0 1 2 3 4
Rear=2
5
Front=1
Delete
B C
0 1 2 3 4
Rear=2
6
Front=2
Delete
C
0 1 2 3 4
Rear=2
7
Front=3
Delete
0 1 2 3 4
Rear=2
Queue is empty
Entry point is called Rear &
Exit point is called Front
Page 14


i) Initially front=rear= -1. It indicates queue is empty.
0 1 2 3 4
front=rear= -1
ii) Add 10
0 1 2 3 4
10
front rear
iii) Add 20
0 1 2 3 4
10 20
front rear
iv) Add 30
0 1 2 3 4
10 20 30
front rear
v) Add 40
0 1 2 3 4
10 20 30 40
front rear
Page 15

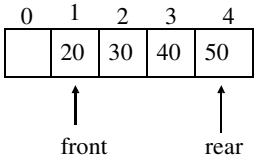
vi) Add 50



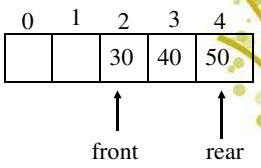
vii) Add 60 (overflow)



viii) delete (10 is removed)



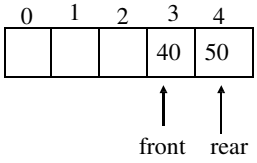
ix) delete (20 is removed)



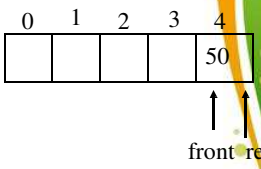
Dr. Sunil Kumar, CSE Dept., MIET Meerut

Page 16

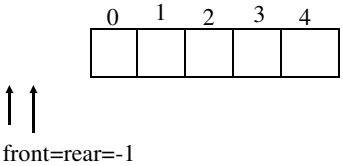
x) delete (30 is removed)



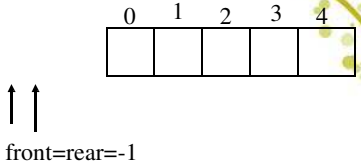
xi) delete (40 is removed)



xii) delete (50 is removed)



ix) delete (underflow)



Dr. Sunil Kumar, CSE Dept., MIET Meerut

Page 17

Implementation of queue using array

Algorithm insert( )

1. If rear = size-1
   then write ('overflow')
2. else
3. Read item
4. rear← rear + 1
5. queue[rear]← item
6. if(front== -1)
7. front= front+1;
8. stop

Algorithm to delete element from the queue

Algorithm delete( )

1. If (front= = -1)
   then write ('queue isempty')
2. else
3. Item ← queue [front]
3. front ← front + 1
4. Stop

Algorithm to display elements from the queue

Algorithm display( )

1. if(front== -1)
   1.1 write ('queue is empty')
2. else
   2.1 repeat for i-> front to rear
   2.2. print queue[i];
3. Stop

Program: implementation of queue using array.

```
# include <stdio.h>
# define size 4
void insertion();
void deletion();
void display();
int front=-1, rear=-1, item, choice, queue[size];
void main()
{
clrscr();
while(1)
{
printf("\n*** MENU ***\n 1. INSERTION\n 2. DELETION\n
3.TRAVERSE\n 4. EXIT\n");
printf("enter your choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:insertion();
break;
case 2:deletion();
break;
case 3:display();  break; case 4:exit(0);
default:printf("*** wrong choice ***\n");}}}
```

```
void insertion()
{
if(rear =size-1)
printf("*** queue is full ***\n");
else
{
printf("Enter item into queue:");
scanf("%d",&item);
rear++;
queue[rear]=item;
if(front==-1)
front++;
}}
```

```
void deletion()
{
if((front==-1)||(front>rear))
printf("*** queue is empty ***\n");
else
{
item=queue[front];
front++;
printf("The deleted item from queue is
%d\n",item);
}
}
```

```
void display()
{
int i;
if(front==-1)
printf("*** queue is empty ***\n");
else
{
printf("\n elements in queue:- ");
for(i=front;i<=rear;i++)
printf("%d",queue[i]);
}}
```

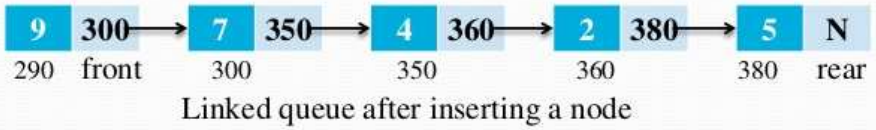

Drawback in queue

- In a queue when the rear pointer reaches to the end of the queue, **insertion would be denied even if room is available at the front.**
- One way to remove this restriction is by using the **circular queue.**

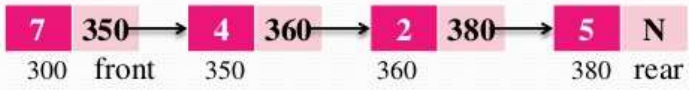
Linked List Implementation of Queue

Linked Representation of Queues

Linked queue



Linked queue after inserting a node



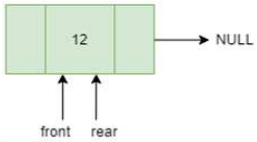
Linked queue after deleting a node

Dr. Sunil Kumar, CSE Dept., MIET Meerut

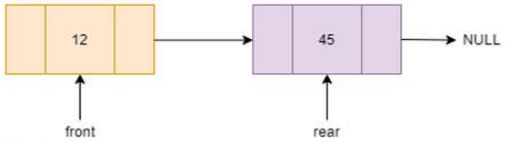
1)

front=NULL
rear=NULL

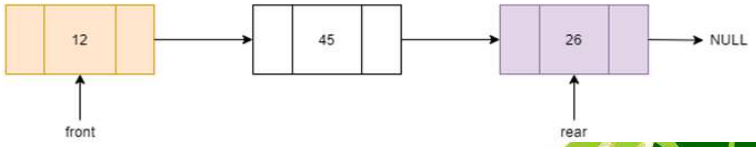
2) **Insert =12**



3) **Insert= 45**

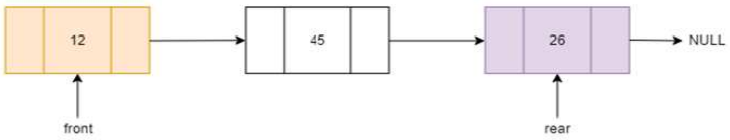


4) **Insert=26**

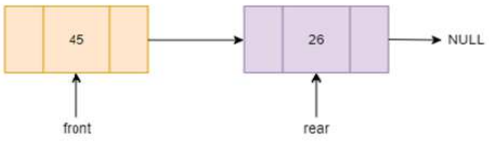


Page 26

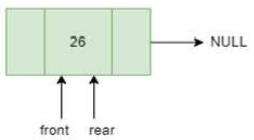
1) **Delete=12**



2) **Delete=45**



3) **Delete=26**



4)

front=NULL
rear=NULL

Page 27

Algorithm to insert elements in the queue

Algorithm_Enqueue

1. newNode -> data = data
2. newNode -> next = NULL
3. if (REAR == NULL)
4. FRONT = REAR = newNode
5. else
6. REAR -> next = newNode
7. REAR = newNode
8. end

Algorithm to delete elements from the queue

Algorithm_Dequeue

1. if(FRONT == NULL)
   print “QUEUE IS EMPTY" and exit.
   else
2. temp = FRONT
3. FRONT = FRONT -> NEXT
4. free(temp)
5. end

```
#include<stdio.h>
#include<conio.h>
struct Node
{
int data;
struct Node *next;
};
struct Node *front = NULL,*rear =
    NULL;
void EnQueue(int);
void DeQueue();
void display();
```

```
int main()
{
int choice, value;
printf("\n*** Queue Implementation
    using Linked List ***\n");
 while(1)
{
  printf("\n****** MENU ******\n");
   printf("1. Insert in Queue\n");
   printf("2. Delete From Queue\n");
   printf("3. Display Queue\n");
   printf("4. Exit\n");
   printf("Enter your choice: ");
   scanf("%d",&choice);
```

Page 30

```
switch(choice)
{
case 1: printf("Insert the value you
    want to enter: ");
scanf("%d", &value);
    EnQueue(value); break;
case 2: DeQueue(); break;
case 3: display(); break;
case 4: exit(0);
default: printf("\nInvalid
    Choice!!\n");
};
}
return 0;
}
```

```
void EnQueue(int value)
{
struct Node *newNode;
newNode = (struct
    Node*)malloc(sizeof(struct
    Node));
newNode -> data = value;
newNode -> next = NULL;
if(front == NULL)
front = rear = newNode;
else
{
rear -> next = newNode;
rear = newNode;
}
}
```

Page 31

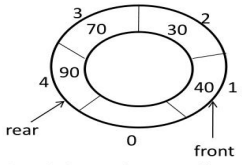
```
void DeQueue()
{
if(front == NULL)
printf("\n Queue is Empty!!!\n");
    else
{
struct Node *temp = front;
front = front -> next;
printf("\n Deleted element is:
    %d\n", temp->data);
free(temp);
}
}
```

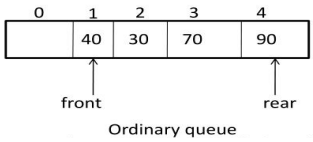
```
void display() {
if(front == NULL)
printf("\n Queue is Empty!!!\n");
    else
{
struct Node *temp = front;
while(temp->next != NULL)
{
printf("%d --> ",temp->data); temp
    = temp -> next;
}
printf("%d \n",temp->data);
}
}
```

Page 32

Overcome disadvantage of Linear Queue

- 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 called '**Ring Buffer**'.
- In a **normal Queue**, we can insert elements until **Rear of queue** becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.





Ordinary queue

Operations on Circular Queue

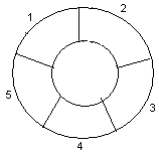
- **enQueue(value):**
  - This function is used to insert an element into the circular queue.
  - In a circular queue, the new element is always inserted at **Rear position**.

deQueue():

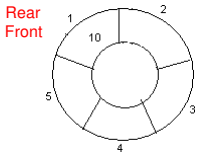
- This function is used to delete an element from the circular queue.
- In a circular queue, the element is always deleted from **Front position**.

Example: Consider the following circular queue with N = 5.

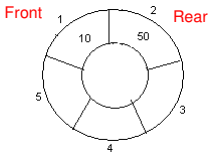
1. Initially, Rear = 0, Front = 0.



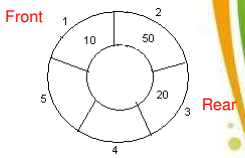
2. Insert 10, Rear = 1, Front = 1.



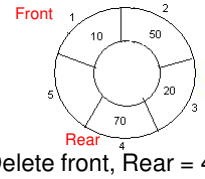
3. Insert 50, Rear = 2, Front = 1.



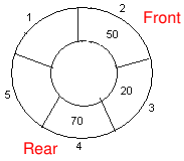
4. Insert 20, Rear = 3, Front = 1.



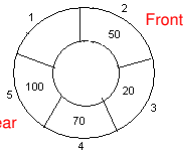
5. Insert 70, Rear = 4, Front = 1.



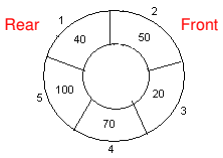
6. Delete front, Rear = 4, Front = 2.



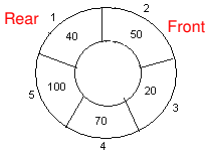
7. Insert 100, Rear = 5, Front = 2.



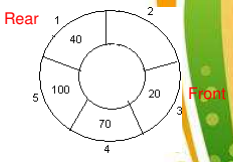
8. Insert 40, Rear = 1, Front = 2.



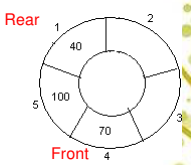
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



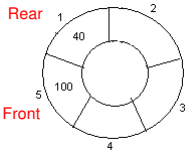
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



Question: Consider the following queue of characters where QUEUE is a **circular array** which is allocated six memory cells

FRONT=2, REAR=4 QUEUE: _ A C D _ _

Describe the queue as following operations take place:

(a) F is added to queue
(b) Two letters are deleted
(c) K, L and M are added
(d) Two letters are deleted
(e) R is added to queue
(f) Two letters are deleted
(g) S is added to queue
(h) Two letters are deleted
(i) One letter is deleted
(j) One letter is deleted

Page 37

Solution:

FRONT=2, REAR=4 QUEUE: _ A C D _ _

(a) FRONT=2, REAR=5 QUEUE: _ A C D F_
(b) FRONT=4, REAR=5 QUEUE: _ _ _ D F _
(c) REAR=2, FRONT=4 QUEUE: L M _ D F K
(d) FRONT=6, REAR=2 QUEUE: L M _ _ _ K
(e) FRONT=6, REAR=3 QUEUE: L M R_ _ K
(f) FRONT=2, REAR=3 QUEUE: _M R _ _ _
(g) REAR=4, FRONT=2 QUEUE: _ M R S _ _
(h) FRONT=4, REAR=4 QUEUE: _ _ _ S _ _
(i) FRONT=REAR=0 [As FRONT=REAR, queue is empty]
(j) Since FRONT=NULL, no deletion can take place. Underflow occurred

Page 38

CIRCULAR QUEUE IMPLEMENTATION

- After Rear reaches the last position, i.e. MAX-1 in order to reuse the vacant positions, we can bring rear back to the 0^{th} position, if it is empty, and continue incrementing Rear in same manner as earlier. Thus Rear will have to be incremented circularly.
- For deletion, Front will also have to be incremented circularly.

Enqueue(Insert) operation on Circular Queue

- **Step 1:** IF (REAR+1)%MAX = FRONT
  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) % MAX
  [END OF IF]
- **Step 3:** SET QUEUE[REAR] = VAL
- **Step 4:** EXIT

Dequeue (Delete) operation on Circular Queue

- **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

PRIORITY QUEUE

- A priority Queue is a collection of elements where **each element is assigned a priority** and the order in which **elements are deleted and processed is determined from the following rules**:

  1) An element of higher priority is processed before any element of lower priority.

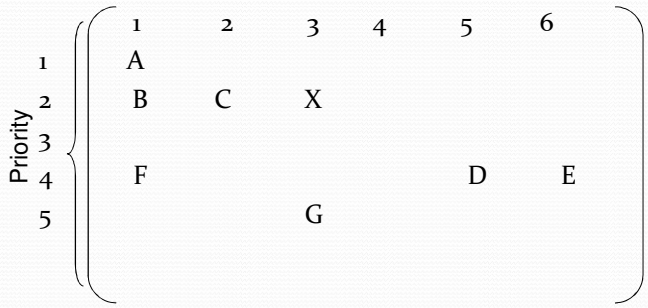
  2) Two elements with the same priority are processed according to the order in which they are added to the queue.

The priority queue implementation

- **The priority queue is again implemented in two way:**
  1. Array/Sequential Representation
  2. Dynamic/Linked Representation

Array Representation of Priority Queue

- One way to maintain a priority queue in memory is to use a separate queue for each level of priority. Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR.
- If each queue is allocated the same amount of space, a two dimensional array QUEUE can be used instead of the linear arrays for representing a priority queue.
- If K represents the row K of the queue, FRONT[K] and REAR[K] are the front and rear indexes of the K^{th} row.



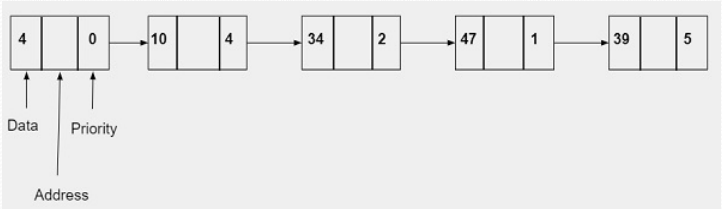
Linked list representation of a priority queue

- Another way to maintain a priority queue in memory is by means of a one-way list. Each node in list will **contain three items of information**: an information field INFO, a priority number PRN and a link field LINK.

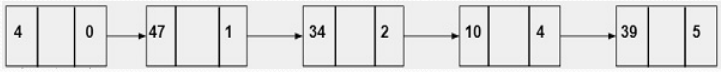
- A node X precedes a node Y in list
  - If X has higher priority than Y
  - Or when both have same priority but X was added to list before Y

Example

- **Input**

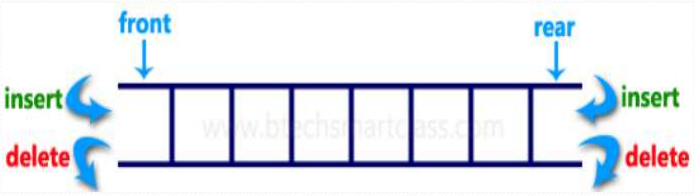


- **output**



Double Ended Queue

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**).
- That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Double Ended Queue

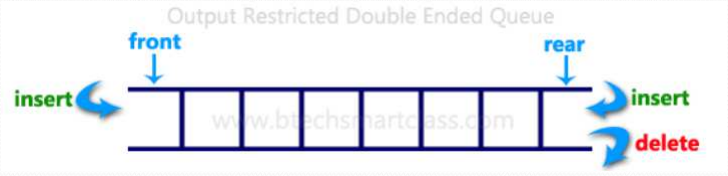
- Double Ended Queue can be represented in TWO ways, those are as follows:
  - **Input Restricted Double Ended Queue**
  - **Output Restricted Double Ended Queue**

Input Restricted Double Ended Queue

- In input restricted double-ended queue, the **insertion operation is performed at only one end and deletion operation is performed at both the ends.**

Output Restricted Double Ended Queue

- In output restricted double ended queue, the **deletion operation is performed at only one end and insertion operation is performed at both the ends**.



Question: Consider the following deque of characters where DEQUE is a circular array which is allocated six memory cells.

LEFT=2, RIGHT=4 DEQUE: _ A,C,D, _ , _

Describe deque while the following operation take place:

(a) F is added to right of deque

(b) Two letters on right are deleted

(c) K,L and M are added to the left of the deque

(d) One letter on left is deleted.

(e) R is added to the left of deque

(f) S is added to right of deque

(g) T is added to the right of deque

Answer:

(a) F is added to right of deque
LFET=2, RIGHT=5 _A C D F _

(b) Two letters on right are deleted
LEFT=2 RIGHT=3 _A C _ _ _

(c) K,L and M are added to the left of the deque
LEFT=5 RIGHT=3 K A C _ M L

(d) One letter on left is deleted.
LEFT=6 RIGHT=3 K A C _ _ L

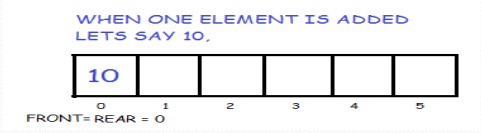
(e) R is added to the left of deque.
LEFT=5 RIGHT= 3 K A C _ R L

(f) S is added to right of deque
LEFT=5 RIGHT= 4 K A C S R L

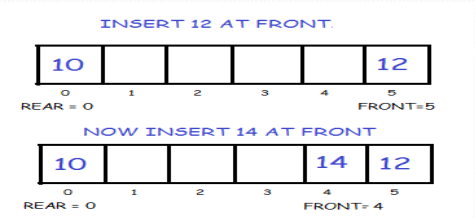
(g) T is added to the right of deque
Since LEFT= RIGHT+1, the array is full and hence T cannot be added to the deque

Insert Elements at Front

- **First we check if the queue is full. If its not full we insert an element at front end by following the given conditions :**
- If the queue is empty then initialize front and rear to 0. Both will point to the first element.



Else we decrement front and insert the element. Since we are using circular array, we have to keep in mind that if front is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.

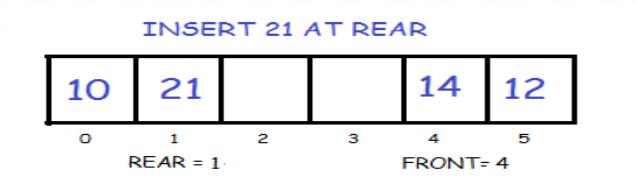


Insert Elements at Front

```
void Dequeue :: push_front(int key)
{
if(full())
{
printf("OVERFLOW\n“);
}
else
{
//If queue is empty
if(front == -1)
front = rear = 0;
//If front points to the first position
    element
else
if(front == 0)
front = SIZE-1;
else
--front;
arr[front] = key;
}
}
```

Insert Elements at Rear

- **Again we check if the queue is full. If its not full we insert an element at back by following the given conditions:**
- If the queue is empty then initialize front and rear to 0. Both will point to the first element.
- Else we increment rear and insert the element. Since we are using circular array, we have to keep in mind that if rear is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.

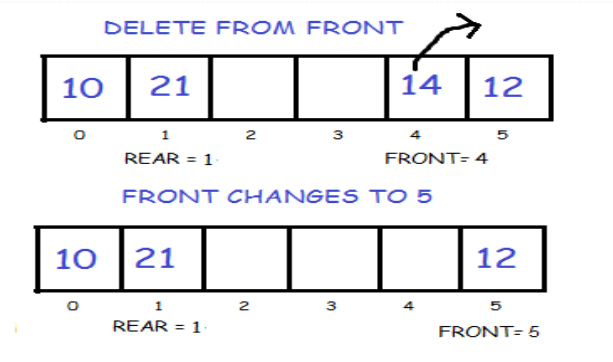


Insert Elements at Rear

```
void Dequeue :: push_back(int key)
{
if(full())
{
printf("OVERFLOW\n");
}
else
{
//If queue is empty
if(front == -1)
front = rear = 0;
//If rear points to the last element
else
if(rear == SIZE-1)
rear = 0;
else
++rear;
arr[rear] = key;
}
}
```

Delete Elements At Front

- **In order to do this, we first check if the queue is empty. If its not then delete the front element by following the given conditions :**
- If only one element is present we once again make front and rear equal to -1.
- Else we increment front. But we have to keep in mind that if front is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.

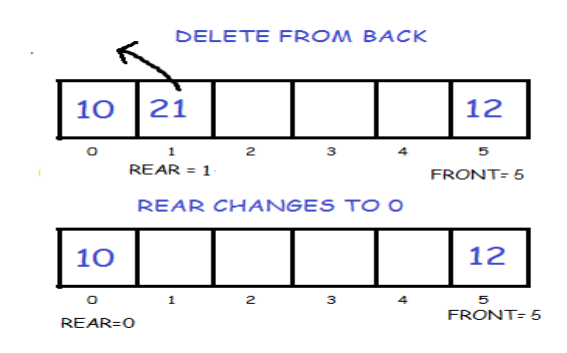


Delete Elements At Front

```
void Dequeue :: pop_front()
{
if(empty())
{
printf("UNDERFLOW\n“);
}
else
{
//If only one element is present
    if(front == rear)
front = rear = -1;
//If front points to the last element
else
if(front == SIZE-1)
front = 0;
else
++front;
}
 }
```

Delete Elements At Rear

- **In order to do this, we again first check if the queue is empty. If its not then we delete the last element by following the given conditions :**
- If only one element is present we make front and rear equal to -1.
- Else we decrement rear. But we have to keep in mind that if rear is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.



Delete Elements At Rear

```
void Dequeue :: pop_back()
{
if(empty())
{
printf( "UNDERFLOW\n)";
}
else
{
//If only one element is present
   if(front == rear)
front = rear = -1;
//If rear points to the first position
    element
else
If(rear == 0)
rear = SIZE-1;
else
--rear;
}
}
```