Module 3
STACKS and QUEUES

Introduction to Stacks

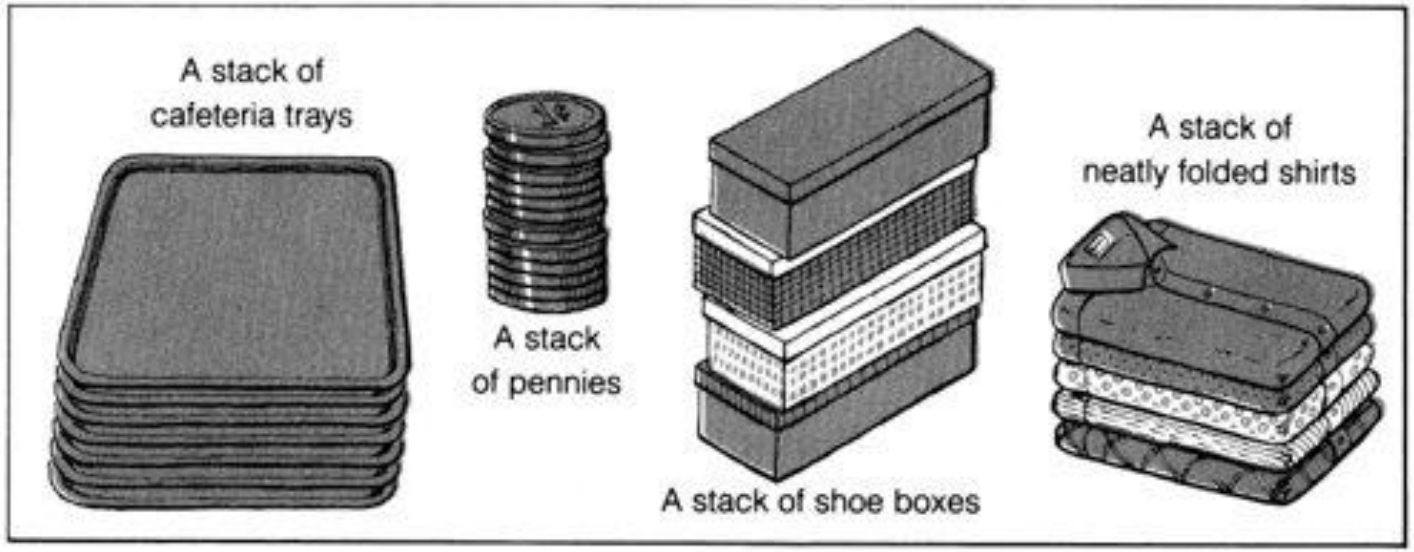
What is a Stack?

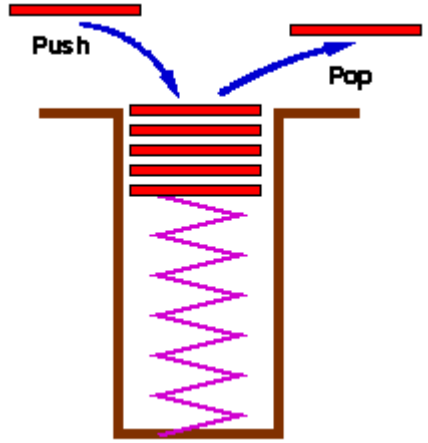
- A linear data structure that follows the Last In, First Out (LIFO) principle.
- The last element added is the first one to be removed.

Key Operations

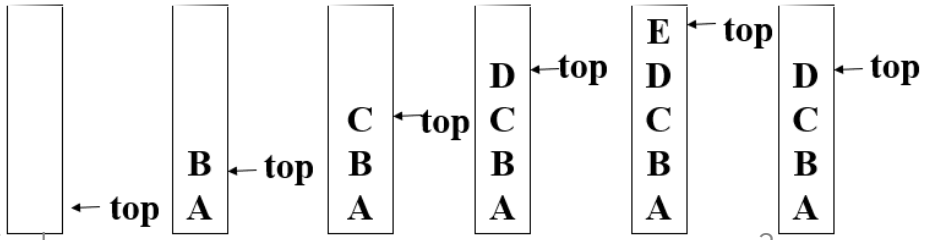
- Push: Add an element to the top of the stack.
- Pop: Remove the top element from the stack.
- Peek/Top: View the top element without removing it.
- isEmpty: Check if the stack is empty.

EXAMPLE AND ILLUSTRATION





Stack: A Last-In-First-Out (LIFO) list



Stack ADT Operations

- **`CreateS(max-stack-size)`** – Create empty stack.
- **`IsFull(stack, max-stack-size)`** – TRUE if full.
- **`Add(stack, item)`** – Push to top.
- **`IsEmpty(stack)`** – TRUE if empty.
- **`Delete(stack)`** – Pop from top.
- **Error Handling**: `stack-full()` and `stack-empty()` functions.

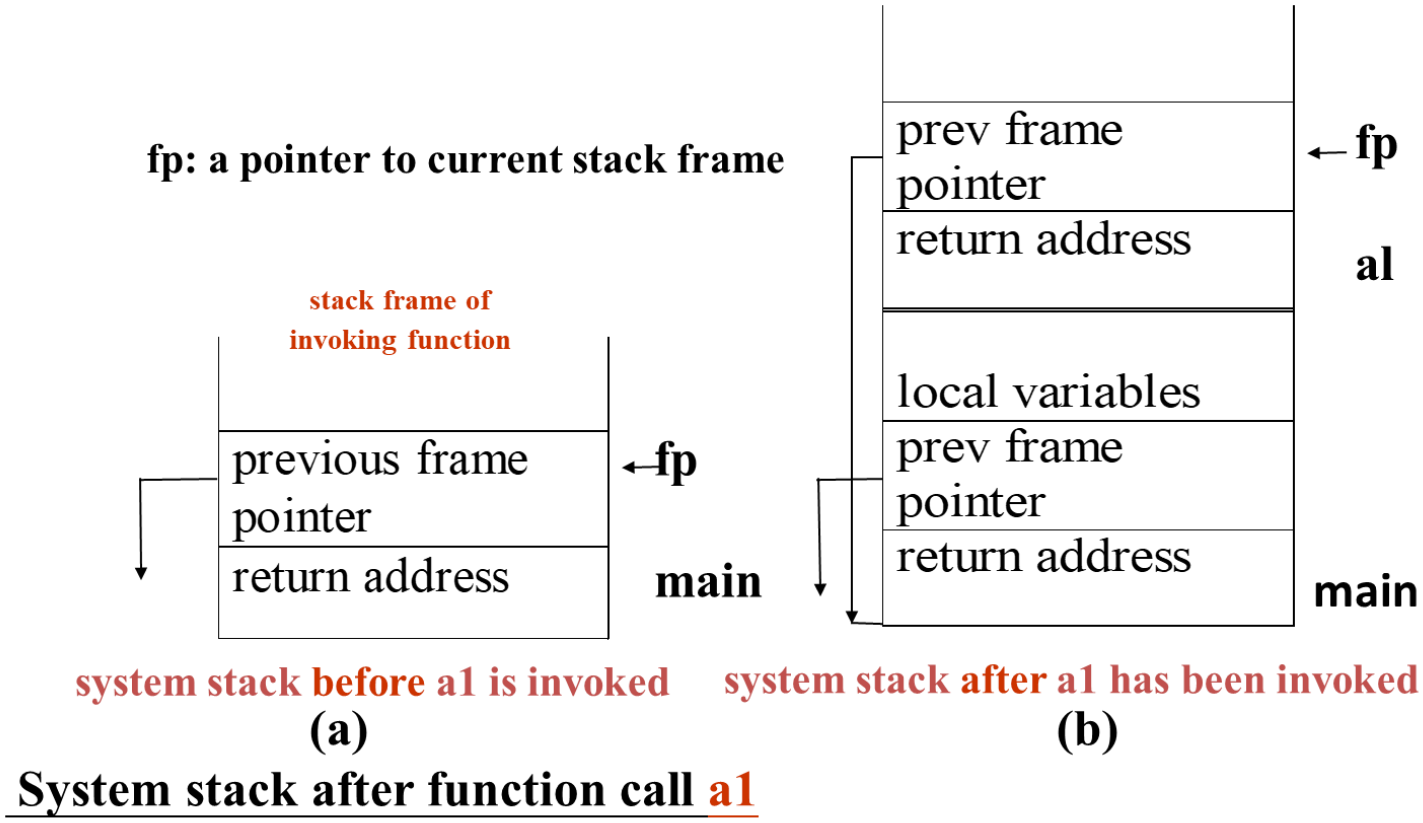
Stacks in Program Execution: Function Calls and Stack Frames

- **Stacks** are integral to program execution.
- **Essential** for managing **function/procedure calls and returns**.
- **Stack frame** enables **recursive calls** without special handling.
- **Function Call** → *Push* a new stack frame.
- **Function Return** → *Pop* the top stack frame.
- **Stack Frame Contents**:
  - Function arguments.
  - Return address (location to resume after call).
  - Local variables (non-static).

Activation Records (Stack Frames) in Function Calls

- When a function is invoked, the program creates an **activation record** (also called a **stack frame**) and places it **on top of the system stack**.
- **Initially**, the activation record contains:
- A **pointer to the previous stack frame** (points to the invoking function's frame).
- The **return address** (location to continue execution after the function returns).
- The **previous frame pointer** links back to the stack frame of the invoking function, maintaining the call chain.

Stack Frame of Function call



Stack Implementation Methods

Array Implementation

- Uses a fixed-size array to store stack elements.
- Simple to implement but has limited capacity.
- `top` variable tracks the current top position.

Linked List Implementation

- Uses dynamically allocated nodes linked together.
- Stack can grow or shrink at runtime (no fixed size limit).
- `top` pointer references the top node in the list.

Implementation of Stack (Array-based)

- Use **1D array** → `stack[MAX-STACK-SIZE]`.
- `top` variable tracks top index.
- Initially `top = -1` (empty).
- **Push Operation**: Increment `top` → store item.
- **Pop Operation**: Retrieve item at `top` → decrement `top`.
- **Checks**:
- `IsEmpty` → `top == -1`.
- `IsFull` → `top == MAX-STACK-SIZE - 1`

Implementation of Stack (Array-based)

```
#define MAX 5
int stack[MAX], top = -1;
// Push operation
void push(int item) {
    if (top == MAX - 1)
        printf("Stack Overflow!\n");
    else
        stack[++top] = item;
        printf("%d pushed to stack.\n", value); }
```

```
// Pop operation
void pop() {
    if (top == -1)
        printf("Stack Underflow!\n");
    else
        printf("Popped: %d\n", stack[top--]);
}
```

Implementation of Stack (Array-based)

```
// Peek function
void peek() {
    if (top == -1) {
        printf("Stack is empty!\n");
    } else {
        printf("Top element is: %d\n", stack[top]);
    }
}
```

```
// isEmpty function
void isEmpty() {
    if (top == -1) {
        printf("Stack is empty.\n");
    } else {
        printf("Stack is not empty.\n");
    }
}
```

Implementation of Stack (Array-based)

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

```
int main() {
    int choice, item;
    do {
        printf("\n1.Push  2.Pop  3.Display  4.Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1: printf("Enter item: ");
                    scanf("%d", &item);
                    push(item);
                    break; ……..
```

Implementation of Stack (Linked List-based)

- **// Node structure**
- struct Node {
- int data;
- struct Node* next;};
- struct Node* top = NULL;

- **// Push operation**
- void push(int value) {
- struct Node* newNode = (structNode*)malloc(sizeof(struct Node));
- if (!newNode) {
- printf("Stack Overflow!\n");
- return; }
- newNode->data = value;
- newNode->next = top;
- top = newNode;
- printf("%d pushed to stack\n", value);}

Implementation of Stack (Linked List-based)

```
// Pop operation
void pop() {
    if (top == NULL) {
        printf("Stack Underflow!\n");
        return;    }
    struct Node* temp = top;
    printf("%d popped from stack\n", top->data);
    top = top->next;
    free(temp); }
```

```
// Peek operation
void peek() {
    if (top == NULL) {
        printf("Stack is empty\n");
        return;    }
    printf("Top element: %d\n", top->data);}
// isEmpty operation
int isEmpty() {
    return (top == NULL); }
```

Implementation of Stack (Linked List-based)

```
// Display stack
void display() {
    struct Node* temp = top;
    if (temp == NULL) {
        printf("Stack is empty\n");
        return;    }
    printf("Stack elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;    }
    printf("\n"); }
```

```
// Main function
int main() {
    int choice, value;
    while (1) {
        printf("\n1. Push\n2. Pop\n3. Peek\n4. isEmpty\n5. Display\n6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:……..
```

Evaluation of Expressions – infix, postfix, prefix and conversions

Applications of Stack

- Balancing Symbols (e.g., parentheses matching)
- Duplicate Parentheses
- Expression Conversion & Evaluation
  - Converting infix to prefix/postfix
  - Evaluating postfix expressions
  - Evaluating prefix expressions

Balancing the parentheses

Ensure that every opening symbol ((, {, [) has a matching closing symbol (), },]), and they are properly nested.

BALANCED EXPRESSION	UNBALANCED EXPRESSION
(a + b)	(a + b
[(c – d) * e]	[(c – d * e]
{ ( ) } [ ]	{ [( ]) }

Checking Balanced Parentheses using Stack

- **Step 1:** Create an empty stack.
- **Step 2:** Scan expression from left to right.
- **Step 3:**
- If symbol is **opening** (`(`, `{`, `[`), push it onto the stack.
- If symbol is **closing** (`)`, `}`, `]`):
  - If stack is empty → Not balanced.
  - Else pop from stack and check if it matches the opening symbol.
- **Step 4:** After scan, if stack is empty → Balanced, else → Not balanced.

Implementation

```
#define MAX 100
struct stack {
    char stck[MAX];
    int top;} s;
```

```
// Push function
void push(char item) {
    if (s.top == MAX - 1) return 0;{
        printf("Stack is Full\n");
    } else {
        s.top++;
        s.stck[s.top] = item; return 1;
    }  }
```

Implementation

```
// Pop function
void pop() {
    if (s.top == -1) {
        printf("Stack is Empty\n");
    } else {
        s.top--;
    } }
```

```
// Function to check matching pairs
int checkPair(char val1, char val2) {
    return ((val1 == '(' && val2 == ')') ||
            (val1 == '[' && val2 == ']') ||
            (val1 == '{' && val2 == '}'));
}
```

Implementation

```
// Function to check if expression is balanced
int checkBalanced(char expr[], int len) {
    for (int i = 0; i < len; i++) {
        if (expr[i] == '(' || expr[i] == '[' || expr[i] == '{') {
            push(expr[i]);
        } else if (expr[i] == ')' || expr[i] == ']' || expr[i] == '}') {
            if (s.top == -1)
                return 0;
        else if (checkPair(s.stck[s.top], expr[i])) {
                pop();
            } else {
                return 0;
            }       }   }
    return (s.top == -1);  }
```

Implementation

```
// Main function
int main() {
    char exp[MAX];
    printf("Enter an expression: ");
    fgets(exp, MAX, stdin);
    exp[strcspn(exp, "\n")] = '\0';
    int len = strlen(exp);
    s.top = -1;
    if (checkBalanced(exp, len))
        printf("Balanced");
    else
        printf("Not Balanced");
    return 0;
}
```

Duplicate Parentheses

- An expression has *duplicate parentheses* if an extra pair of brackets surrounds a valid subexpression unnecessarily.
- In other words, if an expression is enclosed by multiple balanced parentheses without need.
- **Examples**
- `((a+b))` → **Duplicate exists** → **Return True**
- `(a+(b+c))` → **No duplicate** → **Return False**
- Duplicate parentheses reduce readability and do not change meaning of the expression.

Detect Duplicate Parentheses

1. Initialize an empty stack.

2. For each character ch in the expression:

- a) If ch is an operand or operator → Continue.
- b) If ch is an opening parenthesis '(' → Push it onto the stack.
- c) If ch is a closing parenthesis ')':
- i. If the top of the stack is '(' →
- → Duplicate found → Return True.
- ii. Else → Pop elements until '(' is found.
- (This ensures valid matching of parentheses.)

Detect Duplicate Parentheses

3. After scanning all characters:

- a) If the stack is not empty → Unmatched parentheses → Return True.
- b) Else → Return False (No duplicate parentheses).

Implementation

```
#define MAX_SIZE 100
bool isDuplicate(char s[]) {
char Stack[MAX_SIZE];
int top = -1;
int n = strlen(s);
    for (int i = 0; i < n; i++) {
      if (s[i] == ')') {
        // Case 1: Empty brackets ()
        if (top != -1 && Stack[top] == '(') {
            return 1; // Duplicate  }
int elementsInside = 0;
 // Pop until '(' is found
 while (top != -1 && Stack[top] != '(') {
            elementsInside++;
            top--;  }
 // Pop the '(' also
        if (top != -1) {
            top--;    }
```

Implementation

```
// If no valid content inside ()
        if (elementsInside < 1) {
            return true;
        } }
    else {
        // Push character onto stack
        Stack[++top] = s[i];
    }   }
```

```
// If unmatched parentheses remain
   while (top != -1) {
       if (Stack[top] == '(') {
           return true;
       }
       top--;
   }
   return false;
}
```

Implementation

```
int main() {
    char s[MAX_SIZE] = "(((a+(b))+(c+d)))";
    if (isDuplicate(s)) {
        printf("Expression contains duplicate parenthesis.\n");
    } else {
        printf("Expression does not contain duplicate parenthesis.\n");
    }
    return 0;   }
```

Expression Conversion & Evaluation

Infix, postfix and prefix expressions

- **Infix:** *operand1 operator operand2* - Ex: A+B
- **Prefix:** *operator operand1 operand2* - Ex: +AB
- **Postfix:** *operand1 operand2 operator* - Ex: AB+

PRECEDENCE AND ASSOCIATVITY

Token	Operator	Precedence[1]	Associativity
() [] –> .	function call array element struct or union member	17	left-to-right
–– ++	increment, decrement[2]	16	left-to-right
–– ++ ! ~ – + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ –	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += –= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

Problems with Infix

- Hard to parse (ambiguous without rules).
- Requires precedence rules.
- Needs parentheses for clarity.
- Example: `(A + B) * C` vs. `A + (B * C)`.

Alternative Notations

- **Prefix (Polish Notation):** Operator before operands → `+AB`.
- **Postfix (Reverse Polish Notation):** Operator after operands → `AB+`.
- Advantage: No parentheses, no precedence/associativity issues.

Infix, postfix and prefix expressions

Infix Notation	Prefix Notation	Postfix Notation
A + B	+ AB	AB +
(A - C) + B	+ - ACB	AC – B +
A + (B * C)	+ A * BC	ABC *+
(A+B)/(C-D)	/+ AB – CD	AB + CD -/
(A + (B * C))/(C – (D * B))	/+ A * BC – C * DB	ABC * + CDB * - /

Conversion of **Infix to Postfix** Expression

- **Operands** → Send directly to output.
- **Left parenthesis (** → Push onto stack.
- **Right parenthesis)** → Pop & output until (is found.
- **Operators**
- If stack empty / top is (→ Push operator.
- If incoming operator has **higher precedence** → Push it.
- If **lower precedence** → Pop & output until condition satisfied.
- If **same precedence** →
  - **Left** → **Right associativity** → Pop & output top, then push new one.
  - **Right** → **Left associativity** → Push new one.
  - **End of expression** → Pop all operators from stack to output.

INFIX to POSTFIX- Simple expression

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
+	+			0	*a*
b	+			0	*ab*
*	+	*		1	*ab*
c	+	*		1	*abc*
eos				−1	*abc**+

INFIX to POSTFIX- Parenthesized expression

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
*	*			0	*a*
(	*	(		1	*a*
b	*	(		1	*ab*
+	*	(	+	2	*ab*
c	*	(	+	2	*abc*
)	*			0	*abc* +
*	*			0	*abc* +*
d	*			0	*abc* +**d*
eos	*			0	*abc* +**d**

Implementation

- #define MAX_SIZE 100
- **// Function prototypes**
- int IsOperator(char c);
- int IsOperand(char c);
- int precedence(char op);
- int hasHigherOrEqualPrecedence(char op1, char op2);
- void convert(char infix[], char postfix[]);

Implementation

```
// Main function
int main() {
    char infix[MAX_SIZE], postfix[MAX_SIZE];
    int ch;
    do {
        printf("Enter an infix expression: ");
        fgets(infix, MAX_SIZE, stdin);
         infix[strcspn(infix, "\n")] = '\0';
        convert(infix, postfix);
    printf("\nInfix   : %s", infix);
    printf("\nPostfix : %s", postfix);
printf("\n\nDo you want to enter another expression? (1/0): ");
        scanf("%d", &ch);
        getchar(); // clear newline from buffer
    } while (ch == 1);
    return 0;
}
```

Implementation

```
// Check if character is operator
int IsOperator(char c){
    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^'); }

// Check if character is operand (A-Z, a-z, 0-9)
int IsOperand(char c){
    return ((c >= 'A' && c <= 'Z') ||
            (c >= 'a' && c <= 'z') ||
            (c >= '0' && c <= '9')); }
```

```
// Return precedence value
int precedence(char op){
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    if (op == '^') return 3;
    return 0;}
```

Implementation

```
// Check precedence & associativity
int hasHigherOrEqualPrecedence(char op1, char op2) {
    int p1 = precedence(op1);
    int p2 = precedence(op2);
    if (p1 == p2) {
        if (op1 == '^') // '^' is right-associative
            return 0;
        return 1;      // others are left-associative    }
    return (p1 > p2); }
```

Implementation

```
// Convert infix to postfix
void convert(char infix[], char postfix[]) {
    char stack[MAX_SIZE];
    int top = -1, i = 0, j = 0;
    char ch;
// Add '(' to stack and ')' to infix (sentinel trick)
    stack[++top] = '(';
    strcat(infix, ")");
while ((ch = infix[i++]) != '\0') {
        if (ch == ' ')
            continue;   // ignore spaces
else if (ch == '(')
            stack[++top] = ch;
else if (IsOperand(ch))
            postfix[j++] = ch;
```

Implementation

- else if (IsOperator(ch)) {
- while (top != -1 && hasHigherOrEqualPrecedence(stack[top], ch))
- postfix[j++] = stack[top--];
- stack[++top] = ch; }

- else if (ch == ')') {
- while (top != -1 && stack[top] != '(')
- postfix[j++] = stack[top--];
- top--; // remove '('}}
- postfix[j] = '\0';
- }

Practice Infix to Postfix in stack

- **((A*(B+D)/E)-F*(G+H/K)))**
- **(K+L- M*N+(O^P)*W/U/V *T+Q)**

Evaluation of Postfix Expression (using Stack)

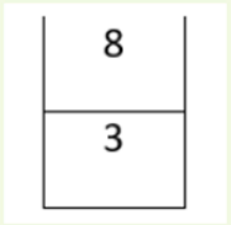
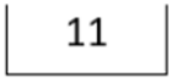
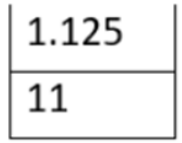
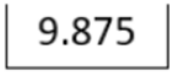
1. Initialize an empty stack
2. For each symbol X in the postfix expression:
- a) If X is an operand → Push X onto stack
- b) If X is an operator:
- i. Pop the top two operands from stack
- ii. Apply the operator (second popped op <operator> first popped op)
- iii. Push the result back onto stack
3. After scanning the expression:
- The value left in the stack is the final result

EVALUATIONOF POSTFIXEXPRESSION: 23*5+

Step	Symbol	Action	Stack
1	2	Push	2
2	3	Push	2 3
3	*	Pop 3 & 2 → 2*3=6 → Push 6	6
4	5	Push	6 5
5	+	Pop 5 & 6 → 6+5=11 → Push 11	11

EVALUATION OF POSTFIX EXPRESSION

Evaluate the postfix expression **3 8 + 9 8 / -**

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

Conversion of **Infix to Prefix** Expression

- 1. Reverse the given infix expression - While reversing, replace '(' with ')' and ')' with '('.
- 2. Initialize an empty stack for operators.
- Initialize an empty string for the output (prefix).
- 3. Scan the reversed infix expression from left to right:
- a) If the symbol is an operand → Append it to output.
- b) If the symbol is '(' → Push it onto stack.
- c) If the symbol is ')' →
- Pop from stack and append to output until '(' is found.
- Discard the '('.

Conversion of **Infix to Prefix** Expression

- d) If the symbol is an operator:
- i. If stack is empty → Push operator.
- ii. If operator has higher precedence than top of stack → Push operator.
- iii. If operator has same precedence →
- - Push operator (for left-associative ops).
- - Special case: If operator is '^' (right-associative), pop stack top and then push '^'.
- iv. If operator has lower precedence than top of stack →
- Pop from stack to output until condition is satisfied,
- then push the operator.

Conversion of **Infix to Prefix** Expression

- 4. After scanning the expression:
- Pop and append all remaining operators from stack to output.
- 5. Reverse the output string.
- This final string is the Prefix expression.

Verification

- **Reverse infix first** (with brackets swapped)
- **Operands → output directly**
- **Operator handling** (higher → push, equal → depends on associativity, lower → pop until safe)
- **Parentheses handling** ( ) pushed, (pops until) is found)
- **End → pop remaining operators**
- **Final reverse output**

Implementation

- #define MAX_SIZE 100
- int IsOperator(char c);
- int IsOperand(char c);
- int Precedence(char c);
- int isRightAssociative(char c);
- void ReverseString(char str[]);
- void ReverseAndSwap(char str[]);
- void Convert(char infix[], char prefix[]);

- int main() {
- char infix[MAX_SIZE], prefix[MAX_SIZE];
- int ch;
- do { printf("Enter an infix expression: ");
- scanf("%s", infix);
- Convert(infix, prefix);
- printf("\nInfix Expression: %s", infix);
- printf("\nPrefix Expression: %s\n", prefix);

Implementation

- printf("\nDo you want to enter another (1/0)? ");
- scanf("%d", &ch);
- } while (ch == 1);
- return 0; }
- int IsOperator(char c) {
- return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
- }

- int IsOperand(char c) {
- return ((c >= 'A' && c <= 'Z') ||
- (c >= 'a' && c <= 'z') ||
- (c >= '0' && c <= '9')); }
- int Precedence(char c) {
- if (c == '+' || c == '-') return 1;
- if (c == '*' || c == '/') return 2;
- if (c == '^') return 3;
- return 0; }

Implementation

```
// Right associativity check (only ^ is right-associative)
int isRightAssociative(char c)
{
    if (c == '^') return 1;
    return 0;
}
```

```
void ReverseString(char str[]) {
    int i, j;
    char temp;
    for (i = 0, j = strlen(str) - 1; i < j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }  }
```

Implementation

```
// Reverse + swap brackets
void ReverseAndSwap(char str[]) {
    ReverseString(str);
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == '(') str[i] = ')';
        else if (str[i] == ')') str[i] = '(';
    }
}
```

```
// ---------- Conversion Function ----------
void Convert(char infix[], char prefix[]) {
    char stack[MAX_SIZE];
    int top = -1;
    int i = 0, j = 0;
    char ch;
// Step 1: Reverse infix and swap brackets
    ReverseAndSwap(infix);
```

Implementation

```
// Step 2: Scan the reversed infix
  while ((ch = infix[i++]) != '\0') {
    if (ch == ' ') continue;
    if (IsOperand(ch)) {
      prefix[j++] = ch;
    } else if (ch == '(') {
      stack[++top] = ch;   }
else if (ch == ')') {
      while (top != -1 && stack[top] != '(') {
        prefix[j++] = stack[top--];  }
        top--; // discard '('   }
    else if (IsOperator(ch)) {
    while (top != -1 &&IsOperator(stack[top])
&&          (Precedence(stack[top]) > Precedence(ch)
||          (Precedence(stack[top]) ==
Precedence(ch) && isRightAssociative(ch))))
{
        prefix[j++] = stack[top--];
 }
        stack[++top] = ch; } }
```

Implementation

```
// Step 3: Pop remaining operators
    while (top != -1) {
        prefix[j++] = stack[top--];
    }
    prefix[j] = '\0';
    // Step 4: Reverse result
    ReverseString(prefix);
}
```

- **Practice step by step**
- **A+B-C*D+(E^F)*G/H/I*J+K**

Evaluation of Prefix Expression

- 1. Initialize an empty stack.
- 2. Scan the prefix expression from **right to left**:
- a) If the symbol is an operand → Push it onto stack.
- b) If the symbol is an operator:
- i. Pop the top two operands from the stack.
- ii. Apply the operator as:
- result = (first popped operand) <operator> (second popped operand)
- iii. Push the result back onto the stack.
- 3. After scanning all symbols:
- The value left in the stack is the final result.

EVALUATION OF PREFIX EXPRESSION

INPUT : **/+33-+47*+123**

Prefix Evaluation:

Symbol	Stack	Action
3	3	Push 3 into stack.
2	3,2	Push 2 into stack.
1	3,2,1	Push 1 into stack.
+	3,3	Pop 1 and 2 from stack and push 1+2=3 into stack.
*	9	Pop 3 and 3 from stack and push 3*3=9 into stack.
7	9,7	Push 7 into stack.
4	9,7,4	Push 4 into stack.
+	9,11	Pop 7 and 4 from stack and push 7*4=11 into stack.
-	2	Pop 9 and 11 from stack and push 11-9=2 into stack.
3	2,3	Push 3 into stack.
3	2,3,3	Push 3 into stack.
+	2,6	Pop 3 and 3 from stack and push 3+3=6 into stack.
/	3	Pop 2 and 6 from stack and push 2/6=3 into stack.

Find the mistake

Implementation

```
#define MAX 100
 ---------- Stack Definition --------
typedef struct {
    int data[MAX];
    int top;
} Stack
---------- Stack Functions ----------
void Push(Stack *s, int value) {
    if (s->top == MAX - 1) {
        printf("Stack Overflow\n");
        exit(1);    }
    s->data[++(s->top)] = value; }

int Pop(Stack *s) {
    if (s->top == -1) {
        printf("Stack Underflow\n");
        exit(1);    }
    return s->data[(s->top)--];  }
```

Implementation

```
// ---------- Prefix Evaluation ----------
int EvaluatePrefix(char expression[]) {
    Stack stack;
    stack.top = -1;
    int i, op1, op2, result;
// Scan expression from right to left
    for (i = strlen(expression) - 1; i >= 0; i--)
{
        char ch = expression[i];
if (ch == ' ')
            continue;
        if (isdigit(ch)) {
            Push(&stack, ch - '0'); // convert char
digit to int
        }
   else {
            op1 = Pop(&stack);
            op2 = Pop(&stack);
```

Implementation

```
switch (ch) {
        case '+': result = op1 + op2; break;
        case '-': result = op1 - op2; break;
        case '*': result = op1 * op2; break;
        case '/': result = op1 / op2; break;
        case '%': result = op1 % op2; break;

 default:
        printf("Invalid operator: %c\n", ch);
        return -1;     }
Push(&stack, result);
     }   }
   return Pop(&stack); // Final result
}
```

Implementation

- // ---------- Main Function ----------
- int main() {
- char expr[MAX];
- printf("Enter a prefix expression: ");
- scanf("%[^\n]s", expr);
- int result = EvaluatePrefix(expr);
- printf("Result = %d\n", result);
- return 0; }

- **Practice**
- CONVERT INFIX TO POSTFIX USING STACK
- a * b + c / d - (e + f) * g / (h - i) + j * k
- APPLY VALUES AND EVALUATE THE PREFIX FOR THE SAME EXPRESSION

Multiple Stacks

N Stacks in a Single Array

Problem:

- Need n stacks simultaneously.
- Maximum size of each stack is unpredictable.
- Sizes vary dynamically.

Multiple Stacks

Naïve Approach:

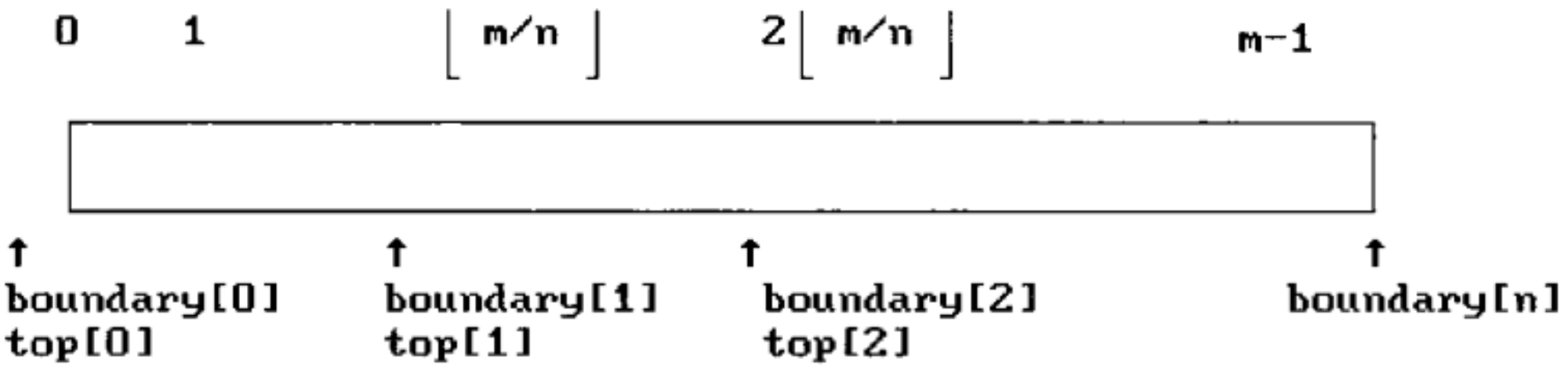
- Divide array into n equal parts.
- Assign each part to one stack.
- Wastes memory if one stack is small and another grows large.

Efficient Approach (Dynamic Sharing):

- Use one array to store all elements of n stacks.
- Use two helper arrays:
    - Top[n] → Stores index of top element of each stack.
    - Next[size] → Works like linked-list, helps track next free/used position.
- Maintain a variable free → points to the beginning of the free list.

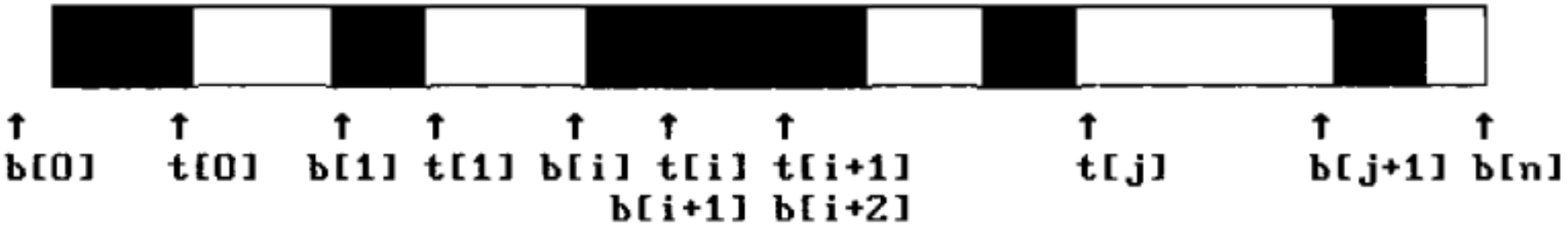
Case size **n = 2**

Initial configuration for n stacks in memory [m]
Case size =n ($\mathbf{n >= 3}$)



All stacks are empty and divided into roughly equal segments.

Configuration when stack i meets stack i + 1, but the memory is not full



b = boundary, t = top

Implementation

- #define MEMORY_SIZE 100
- int n;
- int memory[MEMORY_SIZE];
- int top[10];
- int boundary[10];

- // Function to push an item into ith stack
- void push(int i, int item) {
- if (top[i] == boundary[i+1]) {
- printf("Stack %d is full\n", i);
- } else {
- memory[++top[i]] = item;
- } }

Implementation

```
// Function to pop an item from ith stack
int pop(int i) {
    if (top[i] == boundary[i]) {
        printf("Stack %d is empty\n", i);
        return -1;
    } else {
        return memory[top[i]--];
    }  }
```

```
// Function to display ith stack
void display(int i) {
    if (top[i] == boundary[i]) {
        printf("Stack %d is empty\n", i);
        return;    }
    printf("Stack %d: ", i);
    for (int j = top[i]; j > boundary[i]; j--) {
        printf("%d ", memory[j]);   }
    printf("\n");}
```

Implementation

```
int main() {
    printf("Enter the number of stacks: ");
    scanf("%d", &n);
    // Initialize boundaries and tops
    for (int i = 0; i < n; i++) {
        top[i] = boundary[i] = (MEMORY_SIZE / n) * i - 1;    }
    boundary[n] = MEMORY_SIZE - 1; // Last boundary
```

```
// Test operations
    push(0, 10);    push(0, 20);
    push(1, 30);    push(2, 40);
    push(2, 50);    display(0);
    display(1);    display(2);
    int data1 = pop(0);
    int data2 = pop(1);
printf("\nPopped elements: %d %d\n", data1, data2);
display(0);display(1); return 0; }
```

Try – Multiple Stacks

You have been hired as a software developer for a library management system. One of the requirements is to implement multiple stacks to manage the borrowing and returning of books. Each stack represents a bookshelf, and books are added or removed from the corresponding shelves.

- Write the functions to implement multiple stacks using arrays. Your program should include the following functions:
- Push: Add a book to a specific stack (bookshelf).
- Pop: Remove a book from a specific stack (bookshelf).
- Display: Display all the books in a specific stack (bookshelf).
- Your functions should be able to handle the following scenarios:
- Insertion: Add at least 5 books to different stacks, representing different bookshelves.
- Deletion: Remove a book from a specific stack and display the updated stack.
- Display: Display all the books in a specific stack.

QUEUE DATA STRUCTURES

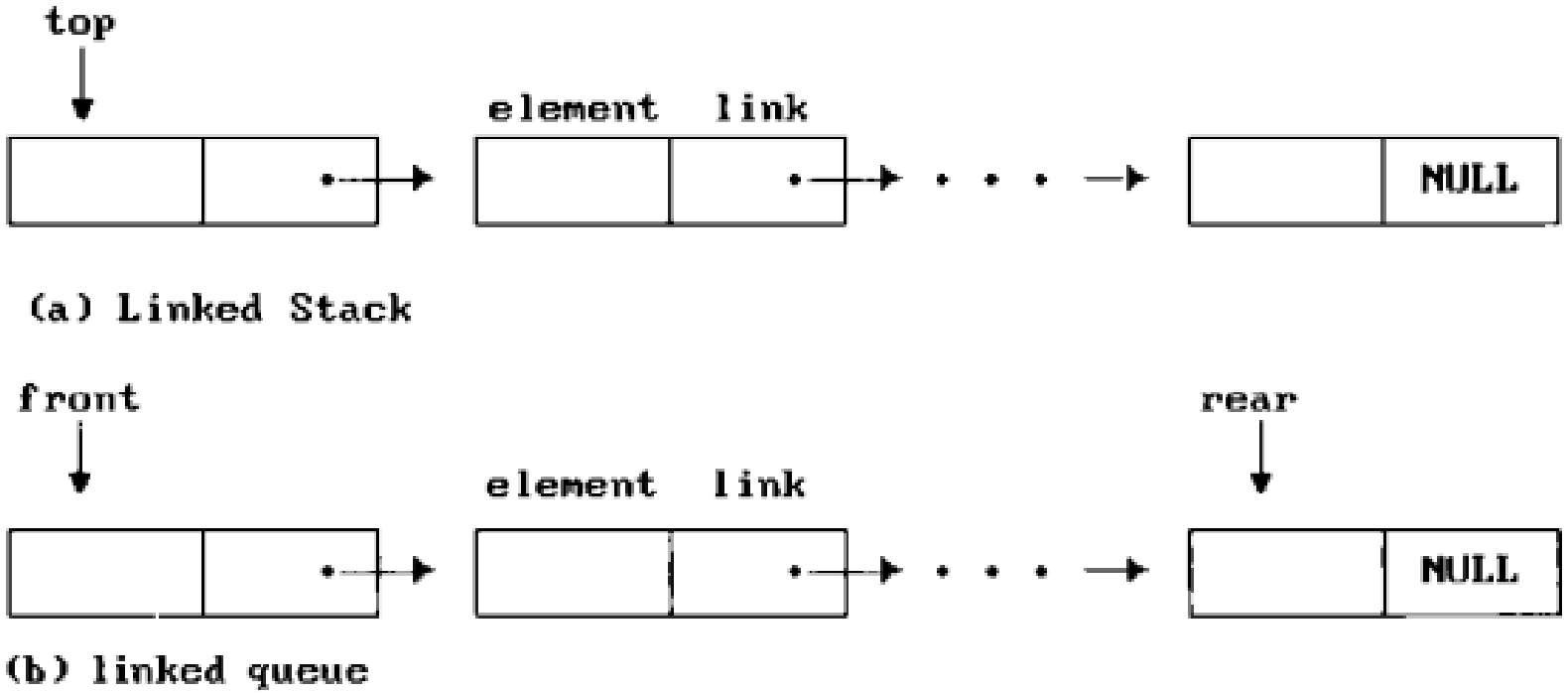
What is Queue? (ADT)

- It is an ordered group of homogeneous items
- Queues have two ends:
  - Elements are added at one end
  - Elements are removed from the other end
- The element added first is also removed first (FIFO: First In, First Out)

Queue Operations

- **enqueue()** → Add (store) an item to the queue
- **dequeue()** → Remove (access) an item from the queue
- **peek()** → Get the front element without removing it
- **isFull()** → Check if the queue is full
- **isEmpty()** → Check if the queue is empty
- **Front pointer** → Used for dequeue/access
- **Rear pointer** → Used for enqueue/store

Linked stack and queue (difference)



Applications of Queue Data Structure

- CPU Scheduling
- Handling of Interrupts in Real-Time Systems
- Graph Algorithms (BFS)
- Call Centers – Request Handling
- Streaming Applications

Queues are used whenever “First Come, First Serve” or order-preserving processing is required.

Types of Queues

Simple Linear Queue (Single Queue)

- Basic queue with one front and one rear

Multiple Queues (Array of Queues)

- Two or more queues implemented together

Circular Queue

- Last position connected back to first position

Double Ended Queue (Deque)

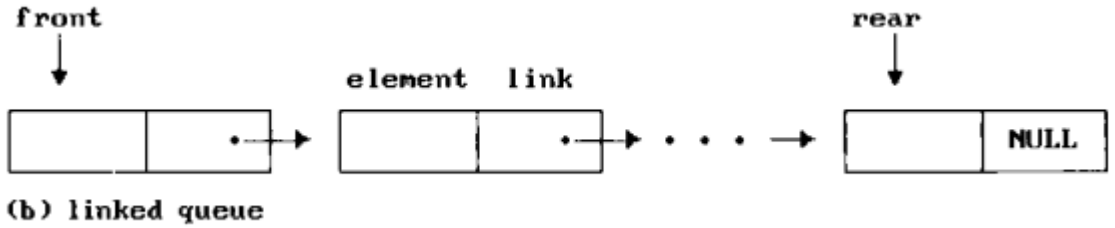
- Insert and delete operations allowed at both ends

Priority Queue

- Elements served based on priority, not just arrival order

Queue Implementation: Array vs Linked List

Inefficient space usage when elements are dequeued.



Linear Queue Implementation

```
#define MAX 5   // Size of the queue
typedef int element;
element queue[MAX];
int front = -1, rear = -1;
// Check if queue is empty
int isEmpty() {
    return (front == -1);}
// Check if queue is full
int isFull() {
    return (rear == MAX - 1);}
```

```
// Enqueue operation
void enqueue(element item) {
    if (isFull()) {
        printf("Queue is full!\n");
        return;    }
    if (isEmpty()) front = 0; // first element
    queue[++rear] = item;
    printf("Enqueued: %d\n", item);}
```

Linear Queue Implementation

```
// Dequeue operation
element dequeue() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
        exit(1);    }
    element item = queue[front];
    if (front == rear) {  front = rear = -1;  }
else {  front++;    }
    return item;}
```

```
// Peek operation
element peek() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
        exit(1);
    }
    return queue[front];
}
```

Linear Queue Implementation

```
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    printf("Front element: %d\n", peek());
    printf("Dequeued: %d\n", dequeue());
    printf("Dequeued: %d\n", dequeue());
    enqueue(40);
    printf("Front element: %d\n", peek());
    return 0;
}
```

Insertion and deletion from a sequential queue

front	*rear*	*Q*[0]	*Q*[1]	*Q*[2]	*Q*[3]	Comments
−1	−1					queue is empty
0	0	J1				Job 1 is added
0	1	J1	J2			Job 2 is added
0	2	J1	J2	J3		Job 3 is added
1	2		J2	J3		Job 1 is deleted
2	2			J3		Job 2 is deleted

Multiple Queues (Array of Queues – Linked Representation)

- #define MAX_QUEUES 3// Number of queues
- typedef int element;
- typedef struct node {
- element item;
- struct node *link; } Node;
- typedef Node* QueuePointer;
- QueuePointer front[MAX_QUEUES], rear[MAX_QUEUES];

- // Initialize all queues
- void initQueues() {
- for (int i = 0; i < MAX_QUEUES; i++) {
- front[i] = rear[i] = NULL;
- }}
- // Check if queue is empty
- int isEmpty(QueuePointer f) {
- return (f == NULL); }

Multiple Queues (Array of Queues – Linked Representation)

```
// Enqueue operation
void enqueue(int q, element item) {
    Node *temp = (Node*) malloc(sizeof(Node));
    if (!temp) { printf("Memory full!\n");
        exit(1);   }
    temp->item = item;
    temp->link = NULL;
    if (front[q]) {
        rear[q]->link = temp;    }
    else {
        front[q] = temp;    }
    rear[q] = temp;
    printf("Enqueued %d in Queue %d\n", item,q);
}
```

Multiple Queues (Array of Queues – Linked Representation)

- // Dequeue operation
- element dequeue(int q) {
- if (isEmpty(front[q])) {
- printf("Queue %d is empty!\n", q);
- exit(1); }
- Node *temp = front[q];
- element item = temp->item;
- front[q] = temp->link;
- free(temp);
- if (!front[q]) rear[q] = NULL; // if queuebecomes empty
- return item;
- }

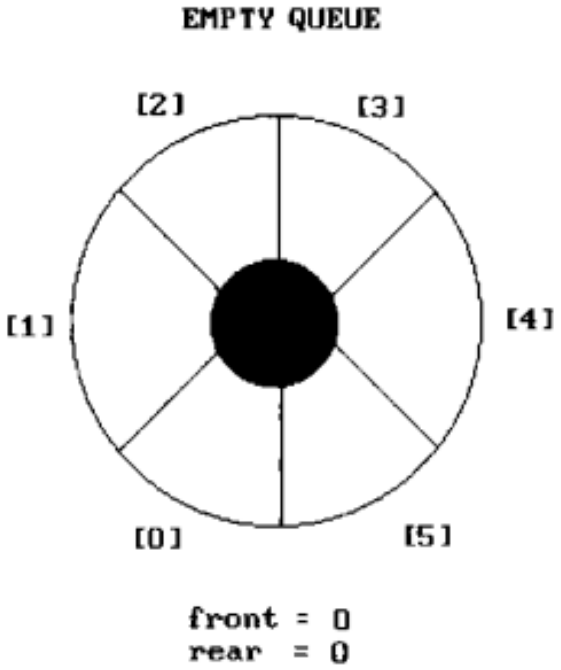
Multiple Queues (Array of Queues – Linked Representation)

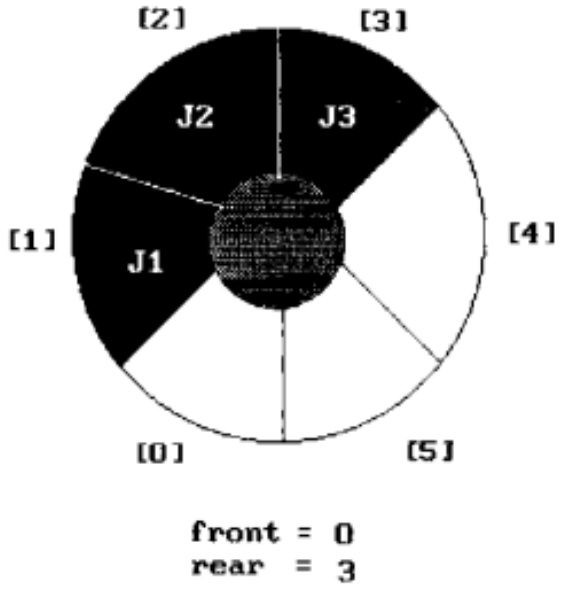
```
// Peek operation
element peek(int q) {
    if (isEmpty(front[q])) {
        printf("Queue %d is empty!\n", q);
        exit(1);  }    return front[q]->item;}
int main() {
    initQueues();
    enqueue(0, 10); enqueue(0, 20);  enqueue(0, 30);
    enqueue(1, 100);
    enqueue(1, 200);
     printf("Front of Queue 0: %d\n", peek(0));
    printf("Dequeued from Queue 0: %d\n", dequeue(0));
    printf("Front of Queue 1: %d\n", peek(1));
    printf("Dequeued from Queue 1: %d\n", dequeue(1))
```

Circular Queue

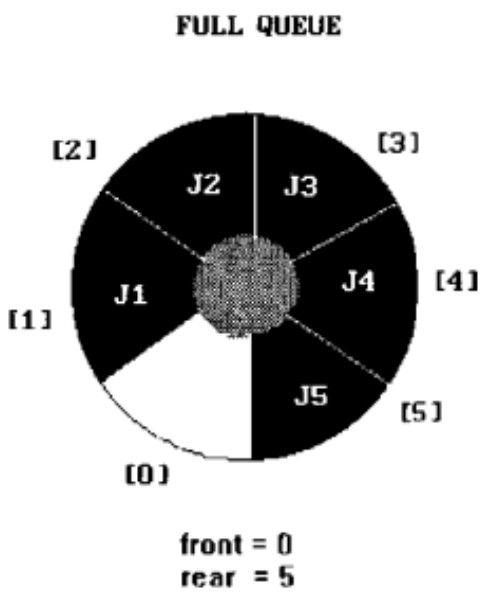
- A Circular Queue is a linear data structure in which the last position is connected back to the first position, forming a circle.
- It solves the limitation of a linear queue, where freed spaces after dequeues cannot be reused.
- Circular Increment Concept:
- When the front or rear reaches the end, it wraps around to the beginning.

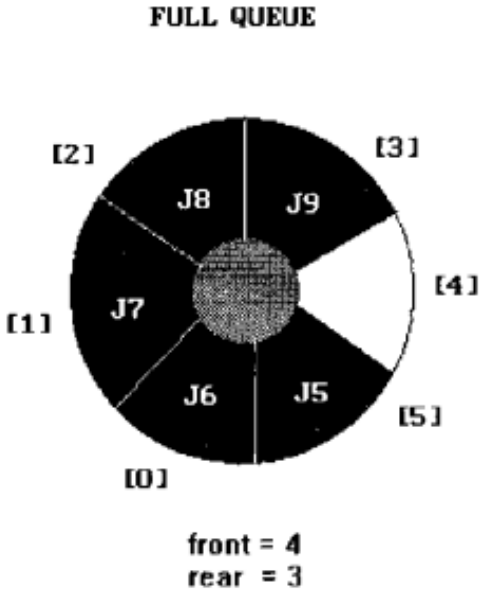
Empty and nonempty circular queues





Full circular queues





Circular Queue Increment Formulas

Enqueue (rear moves forward):

- rear = (rear + 1) % MAX_QUEUE_SIZE;

Dequeue (front moves forward):

- front = (front + 1) % MAX_QUEUE_SIZE;

Check Full Condition:

- isFull() → ((rear + 1) % MAX_QUEUE_SIZE == front)

Check Empty Condition:

- isEmpty() → (front == rear)

Circular Queue - Implementation

```
#define MAX_QUEUE_SIZE 5
typedef int element;
element queue[MAX_QUEUE_SIZE];
int front = 0;   int rear = 0;   int isEmpty() {
    return (front == rear);}
int isFull() {
    return ((rear + 1) % MAX_QUEUE_SIZE == front);
}
```

```
// Enqueue operation
void addq(element item) {
    if (isFull()) {
        printf("Queue is full!\n");
        return;    }
    queue[rear] = item;
    rear = (rear + 1) % MAX_QUEUE_SIZE;
printf("Enqueued: %d\n", item); }
```

Circular Queue - Implementation

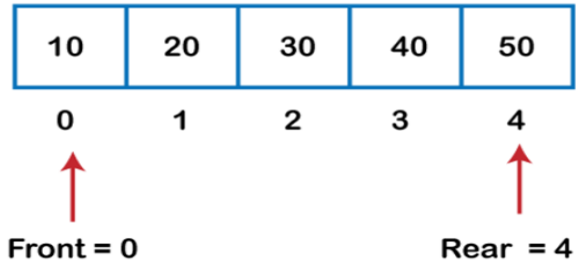
```
// Dequeue operation
element deleteq() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
        exit(1);    }
    element item = queue[front];
    front = (front + 1) % MAX_QUEUE_SIZE;  // Move front circularly
    return item; }
```

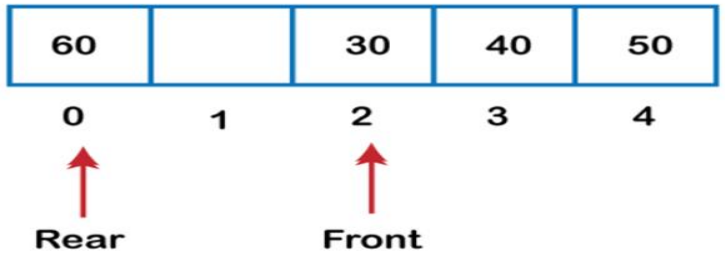
```
// Peek operation
element peek() {
    if (isEmpty()) {
        printf("Queue is empty!\n");
        exit(1);    }
    return queue[front];
}
```

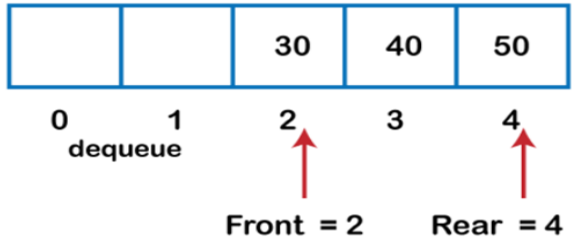
Circular Queue - Implementation

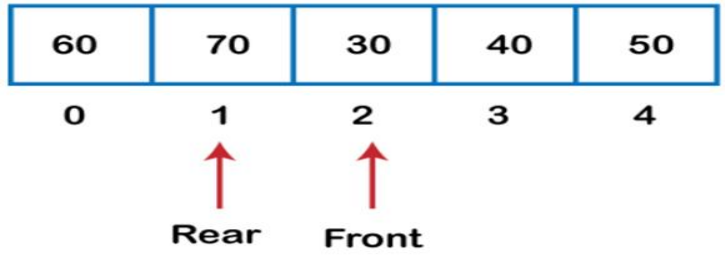
```
int main() {
    addq(10);
    addq(20);
    addq(30);
    addq(40);
    printf("Front element: %d\n", peek());
    printf("Dequeued: %d\n", deleteq());
    printf("Dequeued: %d\n", deleteq());
    addq(50);
    addq(60); // Circular wrap-around
    printf("Front element: %d\n", peek());
     while (!isEmpty()) {
          printf("Dequeued: %d\n", deleteq());
      }
     return 0;
}
```

Circular Queue- Example









Advantage over linear queue:

- No wasted space. After dequeues, circular queue reuses freed positions.

Double ended queue (Deque)

Representation of deque

- **Double Ended Queue (Deque):** is a type of queue in which insertion and deletion of elements can be performed in both ends. It can act as both a queue (FIFO) and a stack (LIFO).
- It does not strictly follow FIFO rule.

Types of Deques

- **Input-Restricted Deque**: Only one end is open for insertion, both ends are open for deletion.

input restricted double ended queue

- **Output-Restricted Deque**: Only one end is open for deletion, both ends are open for insertion.

Output restricted double ended queue

Basic Operations on Deques

- **Insertion**:
- insertFront(x): Insert element x at the front.
- insertRear(x): Insert element x at the rear.
- **Deletion**:
- •deleteFront(): Remove element from the front.
- •deleteRear(): Remove element from the rear.
- **Access**:
- getFront(): Access the front element.
- getRear(): Access the rear element.

Deque- Implementation

```
#define SIZE 10
int deque[SIZE];
int front = -1, rear = -1;
// Insert element at front
void insertFront(int x) {
    if ((front == 0 && rear == SIZE - 1) ||
(front == rear + 1)) {
        printf("Deque is full!\n");
        return;    }
    if (front == -1) {  // Deque is empty
        front = rear = 0;
    } else if (front == 0) {
        front = SIZE - 1;
    } else {
        front = front - 1;    }
    deque[front] = x;
    printf("Inserted %d at front\n", x); }
```

Deque- Implementation

```
// Insert element at rear
void insertRear(int x) {
    if ((front == 0 && rear == SIZE - 1) || (front == rear + 1)) {
        printf("Deque is full!\n");
        return;
    }
    if (front == -1) {  // Deque is empty
        front = rear = 0;
    } else if (rear == SIZE - 1) {
        rear = 0;
    } else {
        rear = rear + 1;
    }
    deque[rear] = x;
    printf("Inserted %d at rear\n", x);
}
```

Deque- Implementation

```
// Delete element from front
void deleteFront() {
    if (front == -1) {
        printf("Deque is empty!\n");
        return;
    }
    printf("Deleted %d from front\n", deque[front]);
    if (front == rear) {  // Only one element
        front = rear = -1;
    } else if (front == SIZE - 1) {
        front = 0;
    } else {
        front = front + 1;
    }
}
```

Deque- Implementation

```
/ Delete element from rear
void deleteRear() {
    if (front == -1) {
        printf("Deque is empty!\n");
        return;
    }
    printf("Deleted %d from rear\n", deque[rear]);
    if (front == rear) {  // Only one element
 front = rear = -1;
    } else if (rear == 0) {
        rear = SIZE - 1;
    } else {   rear = rear - 1;  }}
int main() {
    insertFront(10);
    insertRear(20);
    deleteFront();
    deleteRear();
    return 0; }
```

Applications of Deque

- Deque can be used as both stack and queue, as it supports both operations.
  - Queue behavior: Insert at rear (insertRear) and delete from front (deleteFront).
  - Stack behavior: Insert at front (insertFront) and delete from front (deleteFront) (or from rear).
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Key Differences

Feature	**Linear Queue**	**Circular Queue**	**Deque**
Insertion	Rear only	Rear only	Both front & rear
Deletion	Front only	Front only	Both front & rear
Memory Usage	Wastage possible	Efficient (reuses)	Efficient (reuses + flexible)
Flexibility	Least flexible	Moderate	Most flexible
Example Use Case	Ticket counter line	CPU scheduling	Palindrome check, undo/redo

Priority Queue

An abstract data type where each element is associated with a priority. Elements are served based on priority, not just the order in which they arrive.

- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

- **Real-World Examples**:
  - **Emergency Room**: Treating patients based on the severity of their condition.
  - **Task Scheduling**: Executing tasks based on their importance.

Types of Priority Queues

- **Ascending Priority Queue (Min-Priority Queue)**:
  - Elements with the lowest priority are dequeued first.
- **Descending Priority Queue (Max-Priority Queue)**:
  - Elements with the highest priority are dequeued first.

Array-Based Implementation

- Unsorted Array: Simple insertion, complex deletion.
- Sorted Array: Complex insertion, simple deletion.

Basic Operations:

- Insertion (Enqueue): Add element to array.
- Deletion (Dequeue): Remove the element with the highest/lowest priority.
- Peek: Access the highest/lowest priority element without removing it.

Unsorted Array

- **Insertion in Unsorted Array:**

  Add the new element to the end of the array.

- **Deletion in Unsorted Array:**

  Search the array for the element with the highest (or lowest) priority.

  Remove that element and shift the remaining elements.

Sorted Array

Insertion in Sorted Array:

- **Locate the correct position** in the array to maintain sorted order.
- **Shift elements** to the right to make space for the new element.
- **Insert the new element** at its correct position.

Deletion in Sorted Array:

- Since the array is sorted, the highest (or lowest) priority element is already at the beginning or end of the array.
- **Simply remove the element** from the front (for min-priority) or back (for max-priority) of the array.
- **No need to search** for the element since it's already in place.
- **Shift remaining elements** if needed (only when removing from the front).

Implementation

```
#define MAX 100
int priorityQueue[MAX];
int size = 0;
// Insert element into priority queue
void insert(int element) {
    if (size == MAX) {
        printf("Queue is full!\n");
        return;    }
    priorityQueue[size++] = element; }
```

```
// Delete the maximum element from priority queue
int deleteMax() {
    if (size == 0) {
        printf("Queue is empty!\n");
        return INT_MIN;    }
    int maxIndex = 0;
    for (int i = 1; i < size; i++) {
        if (priorityQueue[i] > priorityQueue[maxIndex])
        {    maxIndex = i;       }    }
```

Implementation

```
int maxElement = priorityQueue[maxIndex];
    // Shift elements to remove the max element
    for (int i = maxIndex; i < size - 1; i++) {
        priorityQueue[i] = priorityQueue[i + 1];
    }
    size--;
    return maxElement; }
```

```
// Display the priority queue
void display() {
    if (size == 0) {
        printf("Queue is empty!\n");
        return;
    }
    for (int i = 0; i < size; i++) {
        printf("%d ", priorityQueue[i]);
    }   printf("\n"); }
```

Implementation

```
int main() {
    insert(4);
    insert(2);
    insert(5);
    insert(3);
    printf("Priority Queue: ");
    display();
    printf("Max Element: %d\n", deleteMax());
    printf("Priority Queue after deletion: ");
    display();
    return 0;
}
```

Try

- Implement Priority Queue Using Linked List
- peekMax() – check the maximum element without removing it.
- peekMin() – check the minimum element without removing it.

Alternative Implementations

- There are more efficient ways to implement priority queues:

- **Heaps**: More efficient for large datasets.
- **Balanced Trees**: Maintain sorted order with better time complexities.