Data Structure - Module 2 - Stacks

Contents

1. **Stack**:

   A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. In a stack, elements are added and removed from the top of the stack only. The last element added to the stack is the first one to be removed. Stack operations typically include push (addition) and pop (removal).

   1. **Expression Evaluation**: Stacks are used to evaluate arithmetic expressions by converting infix expressions to postfix or prefix notation and then evaluating them.
   2. **Function Call Stack**: Stacks are used to manage function calls in programming languages, storing the return addresses and local variables of each function.
   3. **Undo Mechanism**: Stacks can be used to implement an undo mechanism in text editors or other applications, allowing users to revert their actions sequentially.
2. **Multiple Stacks**:

   Multiple stacks refer to having more than one stack data structure in a program. Each stack operates independently, and each has its own set of elements and its own top pointer (or top index). This is useful when you need to manage multiple collections of data independently.
3. **Recursion**:

   Recursion is a programming technique in which a function calls itself directly or indirectly in order to solve a problem. Recursive functions break down a problem into smaller instances of the same problem until they reach a base case, at which point they return a value without making further recursive calls. Recursion is particularly useful for solving problems that can be divided into similar subproblems
4. **Postfix Expression**:

   A postfix expression, also known as a Reverse Polish Notation (RPN), is a mathematical expression in which each operator follows its operands. In other words, the operator comes after the operands. Postfix notation does not require parentheses to specify the order of operations because it is unambiguous. For example, the infix expression "3 + 4 *2" would be written in postfix notation as "3 4 2* +".
5. **Infix Expression**:

   An infix expression is a standard mathematical expression in which operators are written between the operands. In infix notation, parentheses are used to specify the order of operations. For example, the infix expression "3 + 4 * 2" is written with operators between operands and parentheses to indicate that multiplication should be performed before addition.
6. Auckerman function:

   The Ackermann function, denoted as A(m, n), is a recursive mathematical function that plays a fundamental role in computability theory and complexity analysis. It was introduced by the German mathematician Wilhelm Ackermann in 1928. The function is defined recursively as follows:

   Ackermann Function:

   A(m, n) =

   - n + 1 if m = 0
   - A(m - 1, 1) if m > 0 and n = 0
   - A(m - 1, A(m, n - 1)) if m > 0 and n > 0

   The Ackermann function is known for its rapid growth rate, making it an excellent benchmark for testing the computational power of various algorithms and computing systems. Despite its simple definition, the function grows extremely quickly with small increases in its parameters.

Stack operations

push

```c
#include <stdio.h>
#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

void push(int element) {
    if (top >= MAX_SIZE - 1) {
        printf("Stack Overflow\n");
        return;
    }
    stack[++top] = element;
}

int main() {
    push(10);
    push(20);
    return 0;
}
```

pop

```c
#include <stdio.h>
#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

int pop() {
    if (top < 0) {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--]; }

int main() {
    pop();
    return 0; }
```

Display Elements

```c
#include <stdio.h>
#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

void display() {
    if (top < 0) {
        printf("Stack is empty\n");
        return;
    }
    printf("Stack elements:\n");
    for (int i = top; i >= 0; i--) {
        printf("%d\n", stack[i]);
    }
}

int main() {
    display();
    return 0;
}
```

Top of the stack

```c
#include <stdio.h>
#define MAX_SIZE 100

int stack[MAX_SIZE];
int top = -1;

int top_of_stack() {
    if (top < 0) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack[top];
}
int main() {
    // Let's push some elements onto the stack for demonstration
    stack[++top] = 10;
    stack[++top] = 20;

    // Call top_of_stack and print its result
    int top_element = top_of_stack();
    if (top_element != -1) {
        printf("Top element of the stack: %d\n", top_element);
    }
    return 0;}
```

Multiple Stacks

```c
#include <stdio.h>
#include <stdlib.h>

#define MEMORY_SIZE 100 // size of memory
#define MAX_STACKS 10   // max number of stacks plus 1

typedef int element; // type of elements in the stack

element memory[MEMORY_SIZE]; // global memory declaration
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n; // number of stacks entered by the user

void initializeStacks() {
    int j;

    // divide the array into roughly equal segments
    top[0] = boundary[0] = -1;
    for (j = 1; j < n; j++)
        top[j] = boundary[j] = (MEMORY_SIZE / n) * j;
    boundary[n] = MEMORY_SIZE - 1;
}

void push(int i, element item) {
    if (top[i] == boundary[i + 1]) {
        printf("Stack %d is full. Cannot push.\n", i);
        exit(EXIT_FAILURE);
    }
    memory[++top[i]] = item;
}

element pop(int i) {
    if (top[i] == boundary[i]) {
        printf("Stack %d is empty. Cannot pop.\n", i);
        exit(EXIT_FAILURE);
    }
    return memory[top[i]--];
}

int main() {
    printf("Enter the number of stacks (<= %d): ", MAX_STACKS - 1);
    scanf("%d", &n);

    if (n <= 0 || n >= MAX_STACKS) {
        printf("Invalid number of stacks.\n");
        return 1;
    }

    initializeStacks();

    // Example operations on stacks
    push(0, 10);
    push(0, 20);
    push(1, 30);
    push(2, 40);

    printf("Popped from stack 0: %d\n", pop(0));
    printf("Popped from stack 1: %d\n", pop(1));
    printf("Popped from stack 2: %d\n", pop(2));

    return 0;
}
```

Dynamic allocation for stacks

Write an algorithm to implement a stack using dynamic array whose initial capacity is 1 and array doubling is used to increase the stack's capacity (i.e. dynamically reallocate twice the memory) whenever an element is added to full stack. Implement the operations – push, pop and display.

```c
#include <stdio.h>
#include <stdlib.h>

#define INITIAL_CAPACITY 1

typedef struct {
    int *array;
    int capacity;
    int top; // representing the top of the stack
} Stack;

// Function prototypes
void initializeStack(Stack *stack);
void push(Stack *stack, int item);
int pop(Stack *stack);
void display(Stack *stack);

int main() {
    Stack stack;
    initializeStack(&stack);

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    printf("Stack elements: ");
    display(&stack);

    printf("Popped element: %d\n", pop(&stack));
    printf("Popped element: %d\n", pop(&stack));

    printf("Stack elements after popping: ");
    display(&stack);

    return 0;
}
```

```c
void initializeStack(Stack *stack) {
    stack->array = (int *)malloc(INITIAL_CAPACITY * sizeof(int));
    stack->capacity = INITIAL_CAPACITY;
    stack->top = -1; // Initialize top to -1 as the stack is initially empty
}

void push(Stack *stack, int item) {
    if (stack->top == stack->capacity - 1) {
        // If stack is full, double the capacity
        stack->capacity *= 2;
        stack->array = (int *)realloc(stack->array, stack->capacity * sizeof(int));
    }
    // Increment top and add the element to the top of the stack
    stack->array[++stack->top] = item;
}

int pop(Stack *stack) {
    if (stack->top == -1) {
        // If stack is empty, return an error value
        printf("Error: Stack underflow\n");
        exit(EXIT_FAILURE);
    }
    // Return and remove the element from the top of the stack
    return stack->array[stack->top--];
}

void display(Stack *stack) {
    if (stack->top == -1) {
        printf("Empty stack\n");
        return;
    }
    // Print stack elements from top to bottom
    for (int i = stack->top; i >= 0; i--) {
        printf("%d ", stack->array[i]);
    }
    printf("\n");
}
```

Infix to Postfix

```c
#include <stdio.h>
#include <string.h>

// Function to determine precedence of operators
int precedence(char op) {
    switch(op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        case '(':
        case ')':
            return 0;
        default:
            return -1; // Invalid operator
    }
}

// Function to convert infix expression to postfix
void infix_to_postfix(char infix[], char postfix[]) {
    int top = 0, j = 0;
    char stack[30], symbol;
    stack[top] = '('; // Add '(' to the stack to handle expression starting with a '-'

    for (int i = 0; infix[i] != '\0'; i++) {
        symbol = infix[i];
        switch(symbol) {
            case '(':
                stack[++top] = '(';
                break;
            case ')':
                while (stack[top] != '(')
                    postfix[j++] = stack[top--];
                top--; // Discard '('
                break;
            case '+':
            case '-':
            case '*':
            case '/':
            case '^':
                while (precedence(stack[top]) >= precedence(symbol))
                    postfix[j++] = stack[top--];
                stack[++top] = symbol;
                break;
            default: // Operand
                postfix[j++] = symbol;
        }
    }

    while (stack[top] != '(')
        postfix[j++] = stack[top--];
    postfix[j] = '\0';
}

int main() {
    char infix[20], postfix[20];
    printf("Enter a valid infix expression: ");
    fgets(infix, sizeof(infix), stdin);
    infix_to_postfix(infix, postfix);
    printf("The infix expression is: %s\n", infix);
    printf("The postfix expression is: %s", postfix);
    return 0;
}
```

Evaluation of expression

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <ctype.h> // Added for isdigit()

double compute(char symbol, double op1, double op2) {
    switch(symbol) {
        case '+': return op1 + op2;
        case '-': return op1 - op2; // Corrected '-' case
        case '*': return op1 * op2;
        case '/': return op1 / op2;
        case '$':
        case '^': return pow(op1, op2);
        default: return 0;
    }
}

int main() {
    double s[20], res, op1, op2;
    int top = -1, i; // Initialized top variable
    char postfix[20], symbol;

    printf("\nEnter the postfix expression:\n");
    fgets(postfix, sizeof(postfix), stdin); // Replaced gets() with fgets()

    for(i = 0; i < strlen(postfix); i++) { // Corrected loop condition
        symbol = postfix[i];
        if(isdigit(symbol))
            s[++top] = symbol - '0';
        else {
            op2 = s[top--];
            op1 = s[top--];
            res = compute(symbol, op1, op2);
            s[++top] = res;
        }
    }
    res = s[top--];
    printf("\nThe result is: %f\n", res); // Corrected printf statement

    return 0;
}
```

Tower of hanoi

```c
#include <stdio.h>

// Function to move disks from source to destination using auxiliary tower
void towerOfHanoi(int disks, char source, char destination, char auxiliary) {
    if (disks == 1) {
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }
else
{
    towerOfHanoi(disks - 1, source, auxiliary, destination);

    printf("Move disk %d from %c to %c\n", disks, source, destination);

    towerOfHanoi(disks - 1, auxiliary, destination, source);
}
}
int main() {
    int disks;
    printf("Enter the number of disks: ");
    scanf("%d", &disks);
    printf("Steps to solve the Tower of Hanoi problem with %d disks:\n", disks);
    towerOfHanoi(disks, 'A', 'C', 'B'); // A, B, and C are the names of the towers
    return 0;
}
```

Ackermann Function

```
Ackermann(m, n):
    if m = 0:
        return n + 1
    else if m > 0 and n = 0:
        return Ackermann(m - 1, 1)
    else if m > 0 and n > 0:
        return Ackermann(m - 1, Ackermann(m, n - 1))
```

Data structure - Module 2 - Queue

1. **Queue**:

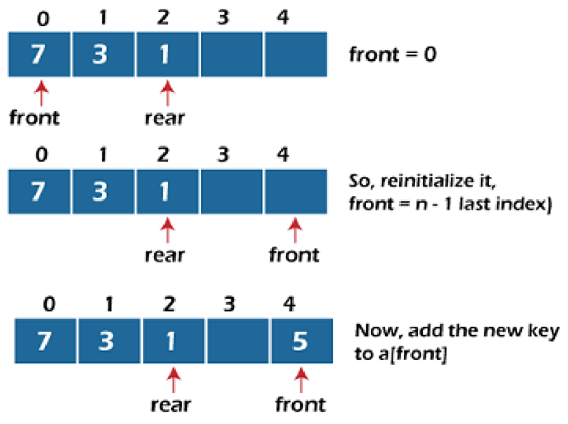
   A queue is a linear data structure that follows the First In, First Out (FIFO) principle. In a queue, elements are added at the rear (enqueue) and removed from the front (dequeue) only. The element that is added first is the one that gets removed first, similar to people waiting in a line. Queue operations include enqueue (insertion) and dequeue (removal).

   1. **Job Scheduling**: Queues are used in job scheduling algorithms, such as First-Come-First-Served (FCFS) and Round Robin, to manage the order of tasks or processes waiting to be executed.
   2. **Breadth-First Search (BFS)**: Queues are used in BFS algorithms to traverse graphs level by level, visiting all nodes at the current level before moving to the next level.
   3. **Printer Spooling**: Queues are used in printer spooling systems to manage print jobs, ensuring that documents are printed in the order they are submitted to the printer queue.
2. **Doubly Ended Queue (Deque)**:

   A doubly ended queue, often abbreviated as deque, is a generalized version of a queue data structure that allows insertion and deletion at both the front and the rear ends. It combines the properties of both stacks and queues. Deques support operations like insertFront, insertRear, deleteFront, deleteRear, and more. This flexibility makes deques useful in situations where elements need to be added or removed from both ends efficiently.

   ***Representation of Deque**:*

   - A deque, or double-ended queue, is a linear list where elements can be added or removed at either end, but not in the middle.
   - It's maintained by a circular array called DEQUE.
   - The deque has two pointers, FRONT and REAR, which point to the two ends of the deque.
   - If FRONT is NULL, it indicates that the deque is empty.



   Variations of Deque:

   1. **Input-Restricted Deque**:
      - Allows insertions at only one end of the list (for example, at the rear).
      - Allows deletions at both ends of the list (from the front and rear).
      - This variation is useful when you need to maintain a queue-like structure but want to prioritize insertions at one end.
   2. **Output-Restricted Deque**:

- Allows deletions at only one end of the list (for example, from the front).
- Allows insertions at both ends of the list (at the front and rear).
- This variation is useful when you need to maintain a stack-like structure but also want to insert elements at the rear.

3. **Priority Queue**:

   A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

   (1) An element of higher priority is processed before any element of lower priority.
   (2) Twoelements with the same priority are processed according to the order in which they were added to the queue.

   `learn algorithms for insertion ,Deletion along with array representation`

Linear Queue Operations

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

#define MAX_SIZE 100

int queue[MAX_SIZE];
int front = 0, rear = -1;

bool is_empty()
{
    return front == -1 && rear == -1;
}

bool is_full()
{
    return rear == MAX_SIZE - 1;
}


void enqueue(int element)
{
    if (is_full()) {
        printf("Queue Overflow\n");
        return;
    }
    if (is_empty()) {
        front =0;
         rear = -1;
    } else {
        rear++;
    }
    queue[rear] = element;

    // Main function
    printf("Enqueued: %d\n", element);
}

int dequeue()
```

```c
{
    if (is_empty()) {
        printf("Queue Underflow\n");
        return -1;
    }
    int element = queue[front];
    if (front == rear) {
        front ==0;
        rear = -1;
    } else {
        front++;
    }

    // Main function
    printf("Dequeued: %d\n", element);
    return element;
}

int peek() {
    if (is_empty()) {
        printf("Queue is empty\n");
        return -1;
    }

    // Main function
    printf("Front element: %d\n", queue[front]);
    return queue[front];
}

void display() {
    if (is_empty()) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display(); // Output: Queue elements: 10 20 30
    peek();    // Output: Front element: 10
    dequeue(); // Output: Dequeued: 10
    display(); // Output: Queue elements: 20 30
    return 0;
}
```

Circular Queue Operations

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

#define MAX_SIZE 100

typedef struct {
    int data[MAX_SIZE];
    int front, rear;
} Queue;

void init_queue(Queue *queue) {
    queue->front = -1;
    queue->rear = -1;
}

bool is_empty(Queue *queue) {
    return queue->front == -1;
}

bool is_full(Queue *queue) {
    return (queue->front == 0 && queue->rear == MAX_SIZE - 1) || (queue->rear == queue->front -
1);
}

void enqueue_front(Queue *queue, int element) {
    if (is_full(queue)) {
        printf("Queue Overflow\n");
        return;
    }
    if (is_empty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->front = (queue->front - 1 + MAX_SIZE) % MAX_SIZE;
    }
    queue->data[queue->front] = element;
}

void enqueue_rear(Queue *queue, int element) {
    if (is_full(queue)) {
        printf("Queue Overflow\n");
        return;
    }
    if (is_empty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % MAX_SIZE;
    }
    queue->data[queue->rear] = element;
}

int dequeue_front(Queue *queue) {
    if (is_empty(queue)) {
        printf("Queue Underflow\n");
        return -1;
    }
    int element = queue->data[queue->front];
```

```c
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
    }
    return element;
}

int dequeue_rear(Queue *queue) {
    if (is_empty(queue)) {
        printf("Queue Underflow\n");
        return -1;
    }
    int element = queue->data[queue->rear];
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->rear = (queue->rear - 1 + MAX_SIZE) % MAX_SIZE;
    }
    return element;
}

void display(Queue *queue) {
    if (is_empty(queue)) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    int i = queue->front;
    do {
        printf("%d ", queue->data[i]);
        i = (i + 1) % MAX_SIZE;
    } while (i != (queue->rear + 1) % MAX_SIZE);
    printf("\n");
}

int main() {
    Queue queue;
    init_queue(&queue);

    enqueue_front(&queue, 10);
    enqueue_front(&queue, 20);
    enqueue_rear(&queue, 30);
    enqueue_rear(&queue, 40);

    printf("After insertion: ");
    display(&queue);

    printf("Deleted from front: %d\n", dequeue_front(&queue));
    printf("Deleted from rear: %d\n", dequeue_rear(&queue));

    printf("After deletion: ");
    display(&queue);

    return 0;
```

```
}
```