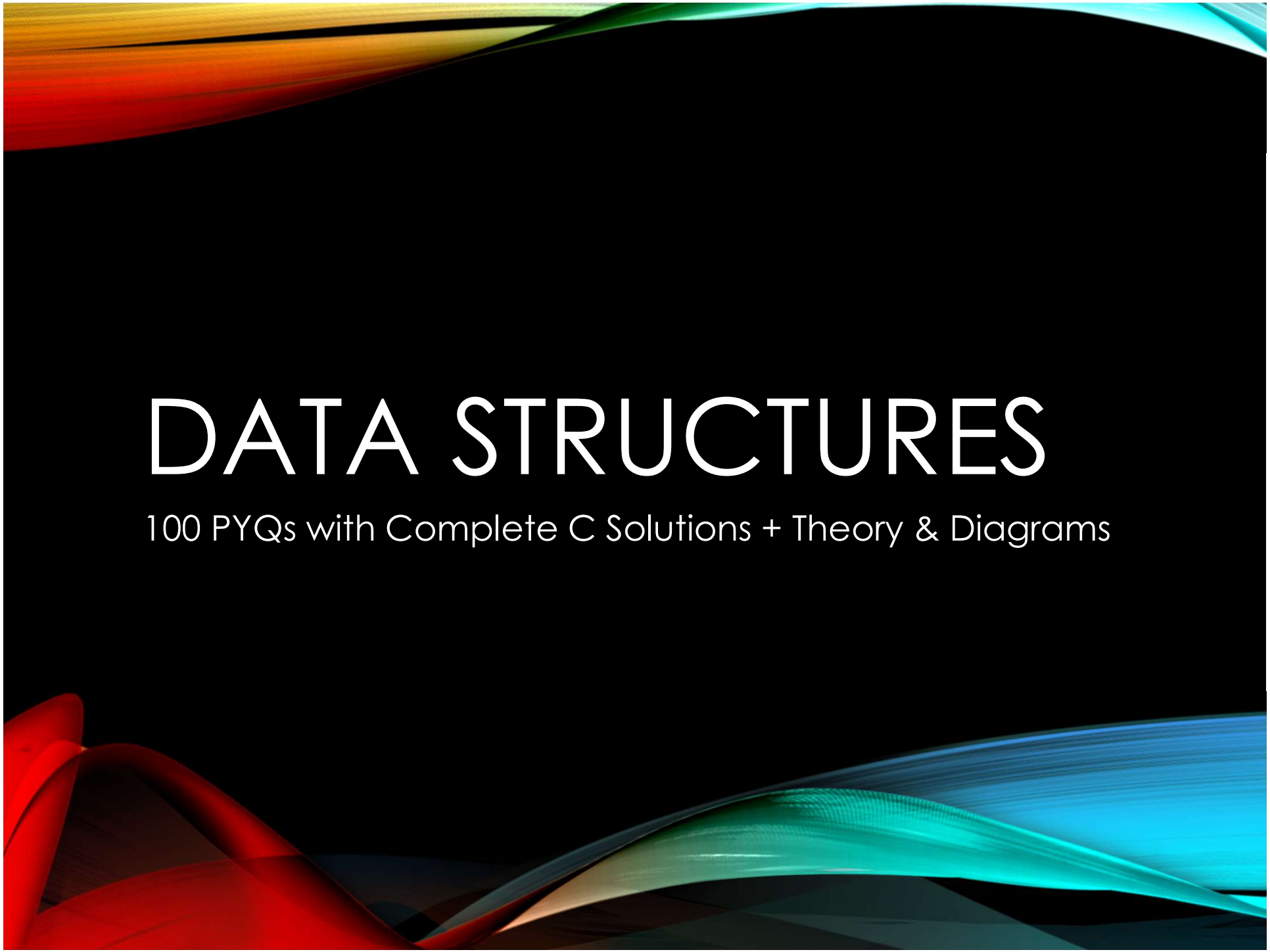
DATA STRUCTURES
100 PYQs with Complete C Solutions + Theory & Diagrams

1.1 DEFINITIONS

- Data – Raw facts and figures (e.g., numbers, text).

1.1 DEFINITIONS

- Information – Processed data that is meaningful.

1.1 DEFINITIONS

- Data Structure (DS) – A way of organizing, storing, and managing data efficiently in memory.

1.1 DEFINITIONS

- Abstract Data Type (ADT) – A mathematical model that defines data and operations, independent of implementation (e.g., Stack, Queue, List).

1.2 TYPES OF DATA STRUCTURES

- Primitive DS – Integer, Float, Character, Pointer.

1.2 TYPES OF DATA STRUCTURES

- Non-Primitive DS

1.2 TYPES OF DATA STRUCTURES

- Linear – Array, Linked List, Stack, Queue.

1.2 TYPES OF DATA STRUCTURES

- Non-Linear – Tree, Graph.

1.3 COMPLEXITY ANALYSIS

- Time Complexity → How much time an algorithm takes.

1.3 COMPLEXITY ANALYSIS

- Space Complexity → How much memory it consumes.

ASYMPTOTIC NOTATIONS

- O (Big-O) → Worst case upper bound.

ASYMPTOTIC NOTATIONS

- Ω (Omega) → Best case lower bound.

ASYMPTOTIC NOTATIONS

- Θ (Theta) → Average case / tight bound.

ASYMPTOTIC NOTATIONS

- Example:
- Linear Search → O(n) in worst case.
- Binary Search → O(log n) in worst case.

1.4 ADVANTAGES OF DS

- Efficient data storage.

1.4 ADVANTAGES OF DS

- Faster access and retrieval.

1.4 ADVANTAGES OF DS

- Reusability of code.

1.4 ADVANTAGES OF DS

- Helps in complex algorithm design (Graphs, Trees).

1.5 UNIT 1 THEORY QUESTIONS

- Q1. Define Data Structure. Explain its types.
- ☑ Answer: Data Structure is… (explained above).
  Types → Primitive, Non-Primitive (Linear & Non-Linear).

1.5 UNIT 1 THEORY QUESTIONS

- Q2. What is an ADT? Give examples.
- ☑ Answer: ADT is… Examples: Stack, Queue, List, Set.

1.5 UNIT 1 THEORY QUESTIONS

- Q3. Explain asymptotic notations with examples.
- ☑ Answer: O, Ω, Θ with example of searching.

1.5 UNIT 1 THEORY QUESTIONS

- Q4. Differentiate between Array and Linked List.
- ☑ Answer: Array = static, contiguous memory, random access. Linked List = dynamic, non-contiguous, sequential access.

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- #include <stdio.h>

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- int main() {

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- int arr[50], n, key, i, flag = 0;

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- printf("Enter size of array: ");

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- scanf("%d", &n);

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- printf("Enter %d elements: ", n);

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

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

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- scanf("%d", &arr[i]);

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- printf("Enter element to search: ");

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- scanf("%d", &key);

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- for(i = 0; i < n; i++) {

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- if(arr[i] == key) {

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- printf("Element found at position %d\n", i+1);

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- flag = 1;

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- break;

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- }

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- }

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- if(flag == 0)

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- printf("Element not found.\n");

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- return 0;

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- }

Q1. WRITE A C PROGRAM FOR LINEAR SEARCH.

- ⏱️ Time Complexity: O(n)

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- #include <stdio.h>

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- int main() {

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- int arr[50], n, key, i, low, high, mid;

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- printf("Enter size of sorted array: ");

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- scanf("%d", &n);

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- printf("Enter %d sorted elements: ", n);

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

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

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- scanf("%d", &arr[i]);

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- printf("Enter element to search: ");

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- scanf("%d", &key);

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- low = 0; high = n-1;

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- while(low <= high) {

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- mid = (low + high) / 2;

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- if(arr[mid] == key) {

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- printf("Element found at position %d\n", mid+1);

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- return 0;

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- } else if(arr[mid] < key)

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- low = mid + 1;

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- else

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- high = mid - 1;

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- }

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- printf("Element not found.\n");

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- return 0;

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- }

Q2. WRITE A C PROGRAM FOR BINARY SEARCH.

- ⏱ Time Complexity: O(log n)

2.1 ARRAYS

- Definition:
- An array is a collection of elements of the same data type stored in contiguous memory locations and accessed using an index.

2.1 ARRAYS

- 1D Array → Linear list of elements.

2.1 ARRAYS

- 2D Array → Matrix (rows & columns).

2.1 ARRAYS

- Multi-Dimensional Array → More than 2D.

2.1 ARRAYS

- Advantages:

2.1 ARRAYS

- Easy access (random access by index).

2.1 ARRAYS

- Memory efficient for fixed size.

2.1 ARRAYS

- Disadvantages:

2.1 ARRAYS

- Fixed size (cannot grow/shrink).

2.1 ARRAYS

- Insertion/Deletion costly (O(n)).

2.1 ARRAYS

- 📌 Diagram: 1D Array

2.1 ARRAYS

- Index → 0 1 2 3 4

2.1 ARRAYS

- Value → 10 20 30 40 50

2.1 ARRAYS

- 📌 Diagram: 2D Array

2.1 ARRAYS

- Col0 Col1 Col2

2.1 ARRAYS

- Row0 10 20 30

2.1 ARRAYS

- Row1 40 50 60

2.1 ARRAYS

- Row2 70   80   90

2.2 STRINGS

- Definition:
- A string is an array of characters ending with a special character '\0'.

2.2 STRINGS

- Example:

2.2 STRINGS

- char str[] = "Hello";

2.2 STRINGS

- 📌 Stored as:

2.2 STRINGS

- 'H' 'e' 'l' 'l' 'o' '\0'

2.2 STRINGS

- Common String Operations:

2.2 STRINGS

- strlen() → Find length

2.2 STRINGS

- strcpy() → Copy

2.2 STRINGS

- strcat() → Concatenate

2.2 STRINGS

- strcmp() → Compare

2.3 UNIT 2 THEORY QUESTIONS

- Q1. Define array. What are its advantages and disadvantages?
- Ans: Array is a collection of elements of same type stored contiguously. Advantages → fast access, memory efficient. Disadvantages → fixed size, costly insertion/deletion.

2.3 UNIT 2 THEORY QUESTIONS

- Q2. Differentiate between 1D and 2D array.

2.3 UNIT 2 THEORY QUESTIONS

- 1D → Linear list (index 0..n-1)

2.3 UNIT 2 THEORY QUESTIONS

- 2D → Matrix (rows & columns).

2.3 UNIT 2 THEORY QUESTIONS

- Q3. Explain string and its operations with examples.
- Ans: String = array of characters terminated by \0.
  Operations: strlen, strcpy, strcat, strcmp.

2.3 UNIT 2 THEORY QUESTIONS

- Q4. What are applications of arrays?

2.3 UNIT 2 THEORY QUESTIONS

- Matrices, Polynomial representation, Searching & Sorting, Database tables.

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Q1. Write a C program for Insertion in Array.

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int main() {

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int arr[50], n, i, pos, val;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter size of array: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- scanf("%d", &n);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter %d elements: ", n);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

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

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- scanf("%d", &arr[i]);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter position and value to insert: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- scanf("%d %d", &pos, &val);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- for(i = n; i >= pos; i--)

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- arr[i] = arr[i-1];

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- arr[pos-1] = val;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- n++;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Array after insertion: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

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

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("%d ", arr[i]);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- return 0;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- }

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Time Complexity: O(n)

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Q2. Write a C program for Deletion in Array.

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int main() {

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int arr[50], n, i, pos;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter size of array: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- scanf("%d", &n);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter %d elements: ", n);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

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

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- scanf("%d", &arr[i]);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter position to delete: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- scanf("%d", &pos);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- for(i = pos-1; i < n-1; i++)

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- arr[i] = arr[i+1];

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- n--;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Array after deletion: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

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

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("%d ", arr[i]);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- return 0;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- }

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Time Complexity: O(n)

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Q3. Write a C program to reverse a string.

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- #include <string.h>

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int main() {

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- char str[50], rev[50];

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int i, j, len;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter a string: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- gets(str);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- len = strlen(str);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- j = 0;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- for(i = len-1; i >= 0; i--) {

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- rev[j++] = str[i];

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- }

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- rev[j] = '\0';

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Reversed string = %s\n", rev);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- return 0;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- }

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Time Complexity: O(n)

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Q4. Write a C program to check if a string is palindrome.

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- #include <string.h>

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int main() {

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- char str[50];

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- int i, len, flag = 0;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Enter a string: ");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- gets(str);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- len = strlen(str);

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- for(i = 0; i < len/2; i++) {

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- if(str[i] != str[len-i-1]) {

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- flag = 1;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- break;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- }

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- }

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- if(flag == 0)

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Palindrome\n");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- else

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- printf("Not Palindrome\n");

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- return 0;

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- }

2.4 UNIT 2 PYQS (PROGRAMMING IN C)

- Time Complexity: O(n)

3.1 INTRODUCTION

- Definition:
- A linked list is a linear data structure where elements (called nodes) are connected using pointers.
- Each node contains:

3.1 INTRODUCTION

- Data → The value of the element.

3.1 INTRODUCTION

- Pointer/Link → Address of the next node.

3.1 INTRODUCTION

- Unlike arrays (stored in contiguous memory), linked lists are stored dynamically in memory.

3.2 TYPES OF LINKED LISTS

- Singly Linked List (SLL):

3.2 TYPES OF LINKED LISTS

- Each node points to the next node.

3.2 TYPES OF LINKED LISTS

- Last node points to NULL.

3.2 TYPES OF LINKED LISTS

- 📌 Diagram: Singly Linked List

3.2 TYPES OF LINKED LISTS

- Head → [Data | Next] → [Data | Next] → [Data | NULL]

3.2 TYPES OF LINKED LISTS

- Doubly Linked List (DLL):

3.2 TYPES OF LINKED LISTS

- Each node has 2 pointers: prev and next.

3.2 TYPES OF LINKED LISTS

- Allows traversal in both directions.

3.2 TYPES OF LINKED LISTS

- 📌 Diagram: Doubly Linked List

3.2 TYPES OF LINKED LISTS

- NULL ← [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | NULL]

3.2 TYPES OF LINKED LISTS

- Circular Linked List (CLL):

3.2 TYPES OF LINKED LISTS

- Last node points back to the first node.

3.2 TYPES OF LINKED LISTS

- Can be singly or doubly circular.

3.2 TYPES OF LINKED LISTS

- 📌 Diagram: Circular Linked List

3.2 TYPES OF LINKED LISTS

- Head → [Data | Next] → [Data | Next] → [Data | Next] -+

3.2 TYPES OF LINKED LISTS

- ^----------+

3.3 APPLICATIONS OF LINKED LIST

- Dynamic memory allocation.

3.3 APPLICATIONS OF LINKED LIST

- Implementation of stacks & queues.

3.3 APPLICATIONS OF LINKED LIST

- Polynomial & sparse matrix representation.

3.3 APPLICATIONS OF LINKED LIST

- Music/video playlist navigation.

3.4 UNIT 3 THEORY QUESTIONS

- Q1. What is a linked list? How is it different from an array?
- Ans: A linked list is a dynamic data structure where nodes are connected by pointers.

3.4 UNIT 3 THEORY QUESTIONS

- Array → Fixed size, contiguous memory.

3.4 UNIT 3 THEORY QUESTIONS

- Linked List → Dynamic size, scattered memory, flexible insertion/deletion.

3.4 UNIT 3 THEORY QUESTIONS

- Q2. Explain types of linked lists with diagrams.
- Ans: SLL, DLL, CLL → explained above.

3.4 UNIT 3 THEORY QUESTIONS

- Q3. What are advantages and disadvantages of linked list?

3.4 UNIT 3 THEORY QUESTIONS

- Advantages → Dynamic size, efficient insertion/deletion.

3.4 UNIT 3 THEORY QUESTIONS

- Disadvantages → No random access, extra memory for pointers.

3.4 UNIT 3 THEORY QUESTIONS

- Q4. Write real-life applications of linked list.
- Ans: Stacks, Queues, Polynomial representation, Dynamic tables, Playlists.

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- Q1. Write a C program to create a singly linked list and display it.

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdlib.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int data;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- };

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int main() {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node *head, *newNode, *temp;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int n, i, val;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- head = NULL;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("Enter number of nodes: ");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- scanf("%d", &n);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- for(i = 0; i < n; i++) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("Enter data for node %d: ", i+1);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- scanf("%d", &val);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode->data = val;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode->next = NULL;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- if(head == NULL) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- head = newNode;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = newNode;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- } else {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp->next = newNode;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = newNode;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("Linked List: ");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = head;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- while(temp != NULL) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("%d -> ", temp->data);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = temp->next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("NULL\n");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- return 0;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- Q2. Write a C program to insert a node at the beginning of singly linked list.

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdlib.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int data;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- };

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- void display(struct Node* head) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* temp = head;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- while(temp != NULL) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("%d -> ", temp->data);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = temp->next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("NULL\n");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int main() {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node *head = NULL, *newNode;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int val;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode->data = 10;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode->next = NULL;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- head = newNode;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode->data = 20;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- newNode->next = head;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- head = newNode;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("Linked List after insertion: ");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- display(head);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- return 0;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- Q3. Write a C program to delete a node from singly linked list.

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdlib.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int data;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- };

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- void display(struct Node* head) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* temp = head;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- while(temp != NULL) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("%d -> ", temp->data);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = temp->next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("NULL\n");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int main() {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node *head, *temp, *prev;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node *n1, *n2, *n3;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- // Creating 3 nodes

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n1 = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n2 = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n3 = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n1->data = 10; n2->data = 20; n3->data = 30;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n1->next = n2; n2->next = n3; n3->next = NULL;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- head = n1;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int key = 20;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = head; prev = NULL;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- while(temp != NULL && temp->data != key) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- prev = temp;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = temp->next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- if(temp == NULL) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("Element not found\n");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- } else {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- if(prev == NULL)

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- head = temp->next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- else

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- prev->next = temp->next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- free(temp);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("Linked List after deletion: ");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- display(head);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- return 0;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- Q4. Write a C program to implement a doubly linked list.

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- #include <stdlib.h>

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int data;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* prev;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- };

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- void display(struct Node* head) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node* temp = head;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- while(temp != NULL) {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("%d <-> ", temp->data);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- temp = temp->next;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("NULL\n");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- int main() {

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- struct Node *head, *n1, *n2, *n3;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n1 = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n2 = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n3 = (struct Node*)malloc(sizeof(struct Node));

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n1->data = 10; n2->data = 20; n3->data = 30;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n1->prev = NULL; n1->next = n2;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n2->prev = n1; n2->next = n3;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- n3->prev = n2; n3->next = NULL;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- head = n1;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- printf("Doubly Linked List: ");

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- display(head);

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- return 0;

3.5 UNIT 3 PYQS (PROGRAMMING IN C)

- }

4.1 INTRODUCTION

- In linear data structures, Stack and Queue are two fundamental abstract data types (ADTs).

4.2 STACK

- Definition:
- A stack is a linear data structure that follows the LIFO (Last In First Out) principle.

4.2 STACK

- Insertion → Push

4.2 STACK

- Deletion → Pop

4.2 STACK

- Top element → Peek

4.2 STACK

- 📌 Diagram of Stack (LIFO):

4.2 STACK

-  ← Top

4.2 STACK

- | 40 |

4.2 STACK

- 

4.2 STACK

- | 30 |

4.2 STACK

- 

4.2 STACK

4.2 STACK

- 

4.2 STACK

- | 10 | ← Bottom

4.2 STACK

- 

OPERATIONS ON STACK

- Push(x): Insert an element.

OPERATIONS ON STACK

- Pop(): Remove the top element.

OPERATIONS ON STACK

- Peek(): Get the top element without removing.

OPERATIONS ON STACK

- isEmpty(): Check if stack is empty.

OPERATIONS ON STACK

- isFull(): Check if stack is full (in case of array implementation).

APPLICATIONS OF STACK

- Expression evaluation (Postfix, Prefix).

APPLICATIONS OF STACK

- Function calls (recursion).

APPLICATIONS OF STACK

- Undo/Redo operations in editors.

APPLICATIONS OF STACK

- Backtracking algorithms.

4.3 QUEUE

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

4.3 QUEUE

- Insertion → Enqueue (at rear)

4.3 QUEUE

- Deletion → Dequeue (from front)

4.3 QUEUE

- 📌 Diagram of Queue (FIFO):

4.3 QUEUE

- Front → [10][20][30][40] ← Rear

TYPES OF QUEUE

- Simple Queue → Normal FIFO.

TYPES OF QUEUE

- Circular Queue → Rear connects back to front when space is available.

TYPES OF QUEUE

- Double Ended Queue (Deque) → Insert/Delete from both ends.

TYPES OF QUEUE

- Priority Queue → Elements are dequeued based on priority.

APPLICATIONS OF QUEUE

- Scheduling (CPU scheduling, job scheduling).

APPLICATIONS OF QUEUE

- Printer task management.

APPLICATIONS OF QUEUE

- Networking (data packets).

APPLICATIONS OF QUEUE

- Call center systems.

4.4 UNIT 4 THEORY QUESTIONS

- Q1. Define stack. Explain its applications.
- Ans: Stack is LIFO-based. Applications: recursion, backtracking, undo-redo, expression evaluation.

4.4 UNIT 4 THEORY QUESTIONS

- Q2. Differentiate between stack and queue.

4.4 UNIT 4 THEORY QUESTIONS

- Stack → LIFO, insertion/deletion at one end.

4.4 UNIT 4 THEORY QUESTIONS

- Queue → FIFO, insertion at rear & deletion at front.

4.4 UNIT 4 THEORY QUESTIONS

- Q3. What is a circular queue? Why is it better than a simple queue?
- Ans: In circular queue, memory is reused by connecting rear to front. Prevents memory wastage.

4.4 UNIT 4 THEORY QUESTIONS

- Q4. Explain priority queue with an example.
- Ans: In priority queue, higher priority elements are dequeued first (e.g., hospital emergency ward).

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- Q1. Write a C program to implement stack using array.

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #define MAX 5

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int stack[MAX], top = -1;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void push(int val) {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(top == MAX - 1)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Stack Overflow\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- top++;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- stack[top] = val;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d pushed to stack\n", val);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void pop() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(top == -1)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Stack Underflow\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d popped from stack\n", stack[top--]);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void display() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(top == -1)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Stack is empty\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Stack elements: ");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- for(int i = top; i >= 0; i--)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d ", stack[i]);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int main() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- push(10);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- push(20);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- push(30);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- pop();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- return 0;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- Q2. Write a C program to implement queue using array.

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #define MAX 5

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int queue[MAX], front = -1, rear = -1;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void enqueue(int val) {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(rear == MAX - 1)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Queue Overflow\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(front == -1) front = 0;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- queue[++rear] = val;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d enqueued\n", val);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void dequeue() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(front == -1 || front > rear)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Queue Underflow\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d dequeued\n", queue[front++]);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void display() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(front == -1 || front > rear)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Queue is empty\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Queue elements: ");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- for(int i = front; i <= rear; i++)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d ", queue[i]);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int main() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- enqueue(10);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- enqueue(20);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- enqueue(30);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- dequeue();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- return 0;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- Q3. Write a C program to implement circular queue.

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #define MAX 5

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int cq[MAX], front = -1, rear = -1;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void enqueue(int val) {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if((front == 0 && rear == MAX - 1) || (rear + 1) % MAX == front)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Circular Queue Overflow\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(front == -1) front = 0;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- rear = (rear + 1) % MAX;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- cq[rear] = val;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d enqueued\n", val);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void dequeue() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(front == -1)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Circular Queue Underflow\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d dequeued\n", cq[front]);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(front == rear)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- front = rear = -1;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- front = (front + 1) % MAX;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void display() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(front == -1)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Circular Queue is empty\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Circular Queue elements: ");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int i = front;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- while(1) {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d ", cq[i]);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(i == rear) break;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- i = (i + 1) % MAX;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int main() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- enqueue(10);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- enqueue(20);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- enqueue(30);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- dequeue();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- return 0;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- Q4. Write a C program to implement stack using linked list.

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- #include <stdlib.h>

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- struct Node {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int data;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- struct Node* next;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- };

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- struct Node* top = NULL;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void push(int val) {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- newNode->data = val;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- newNode->next = top;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- top = newNode;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d pushed\n", val);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void pop() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(top == NULL)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Stack Underflow\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- struct Node* temp = top;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d popped\n", temp->data);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- top = top->next;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- free(temp);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- void display() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- struct Node* temp = top;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- if(temp == NULL)

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Stack is empty\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- else {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("Stack elements: ");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- while(temp != NULL) {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("%d ", temp->data);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- temp = temp->next;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- printf("\n");

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- int main() {

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- push(10);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- push(20);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- push(30);

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- pop();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- display();

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- return 0;

4.5 UNIT 4 PYQS (PROGRAMMING IN C)

- }

5.1 INTRODUCTION

- After linear data structures (array, stack, queue, linked list), we study non-linear data structures.

5.1 INTRODUCTION

- Tree → Hierarchical structure.

5.1 INTRODUCTION

- Graph → Network structure.

5.2 TREE

- Definition:
- A tree is a non-linear data structure that represents hierarchical relationships between elements (nodes).

5.2 TREE

- Root → Top-most node.

5.2 TREE

- Edge → Link between nodes.

5.2 TREE

- Parent → Node having children.

5.2 TREE

- Child → Node derived from parent.

5.2 TREE

- Leaf → Node with no children.

5.2 TREE

- 📌 Diagram of Binary Tree:

5.2 TREE

- (10) Root

5.2 TREE

- / \

5.2 TREE

- (20) (30)

5.2 TREE

5.2 TREE

- (40) (50) (60)

TYPES OF TREES

- Binary Tree → Each node has max 2 children.

TYPES OF TREES

- Full Binary Tree → Every node has 0 or 2 children.

TYPES OF TREES

- Complete Binary Tree → All levels full, last level filled left to right.

TYPES OF TREES

- Binary Search Tree (BST) → Left child < root < right child.

TYPES OF TREES

- AVL Tree → Self-balancing BST.

TREE TRAVERSALS

- Inorder (LNR): Left → Node → Right

TREE TRAVERSALS

- Preorder (NLR): Node → Left → Right

TREE TRAVERSALS

- Postorder (LRN): Left → Right → Node

TREE TRAVERSALS

- 📌 Example (Binary Tree Traversal):

TREE TRAVERSALS

- 1

TREE TRAVERSALS

- / \

TREE TRAVERSALS

- 2 3

TREE TRAVERSALS

- / \

TREE TRAVERSALS

- 4 5

TREE TRAVERSALS

- Inorder: 4 2 5 1 3

TREE TRAVERSALS

- Preorder: 1 2 4 5 3

TREE TRAVERSALS

- Postorder: 4 5 2 3 1

APPLICATIONS OF TREES

- Database indexing (B-tree, B+ tree).

APPLICATIONS OF TREES

- File system hierarchy.

APPLICATIONS OF TREES

- Expression parsing.

APPLICATIONS OF TREES

- Searching and sorting.

5.3 GRAPH

- Definition:
- A graph is a set of vertices (nodes) and edges (links) connecting them.

5.3 GRAPH

- 📌 Diagram of Graph:

5.3 GRAPH

- (A) ----- (B)

5.3 GRAPH

- | \ |

5.3 GRAPH

- | \ |

5.3 GRAPH

- (C) ---- (D)

TYPES OF GRAPHS

- Undirected Graph → Edges have no direction.

TYPES OF GRAPHS

- Directed Graph (Digraph) → Edges have direction.

TYPES OF GRAPHS

- Weighted Graph → Each edge has a weight (cost).

TYPES OF GRAPHS

- Connected Graph → Path exists between all nodes.

TYPES OF GRAPHS

- Cyclic Graph → Graph containing cycles.

GRAPH REPRESENTATIONS

- Adjacency Matrix: 2D array (n × n).

GRAPH REPRESENTATIONS

- Adjacency List: Linked list of neighbors.

GRAPH TRAVERSALS

- Depth First Search (DFS): Go deep along a branch before backtracking.

GRAPH TRAVERSALS

- Breadth First Search (BFS): Visit level by level using a queue.

APPLICATIONS OF GRAPHS

- Social networks (friendship connections).

APPLICATIONS OF GRAPHS

- Google Maps (shortest path algorithms).

APPLICATIONS OF GRAPHS

- Network routing.

APPLICATIONS OF GRAPHS

- Scheduling and dependency resolution.

5.4 UNIT 5 THEORY QUESTIONS

- Q1. Define binary tree. Explain its applications.
- Ans: Binary tree → hierarchical DS with max 2 children. Applications: searching, expression trees, memory management.

5.4 UNIT 5 THEORY QUESTIONS

- Q2. Differentiate between tree and graph.

5.4 UNIT 5 THEORY QUESTIONS

- Tree → Hierarchical, no cycles.

5.4 UNIT 5 THEORY QUESTIONS

- Graph → Network, may contain cycles.

5.4 UNIT 5 THEORY QUESTIONS

- Q3. Explain DFS and BFS.
- DFS → stack/recursion, deep search.
- BFS → queue, level order traversal.

5.4 UNIT 5 THEORY QUESTIONS

- Q4. Write properties of Binary Search Tree.

5.4 UNIT 5 THEORY QUESTIONS

- Left < Root < Right.

5.4 UNIT 5 THEORY QUESTIONS

- Inorder traversal gives sorted order.

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- Q1. Write a C program for Binary Tree traversals (Inorder, Preorder, Postorder).

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #include <stdlib.h>

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int data;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* left;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* right;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- };

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* createNode(int data) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- newNode->data = data;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- newNode->left = newNode->right = NULL;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return newNode;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- void inorder(struct Node* root) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(root != NULL) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- inorder(root->left);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("%d ", root->data);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- inorder(root->right);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- void preorder(struct Node* root) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(root != NULL) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("%d ", root->data);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- preorder(root->left);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- preorder(root->right);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- void postorder(struct Node* root) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(root != NULL) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- postorder(root->left);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- postorder(root->right);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("%d ", root->data);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int main() {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* root = createNode(1);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- root->left = createNode(2);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- root->right = createNode(3);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- root->left->left = createNode(4);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- root->left->right = createNode(5);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("Inorder: ");

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- inorder(root);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("\nPreorder: ");

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- preorder(root);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("\nPostorder: ");

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- postorder(root);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return 0;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- Q2. Write a C program to implement Binary Search Tree (BST).

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #include <stdlib.h>

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int data;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* left;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* right;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- };

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* createNode(int data) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- newNode->data = data;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- newNode->left = newNode->right = NULL;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return newNode;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* insert(struct Node* root, int data) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(root == NULL) return createNode(data);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(data < root->data)

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- root->left = insert(root->left, data);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- else if(data > root->data)

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- root->right = insert(root->right, data);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return root;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- void inorder(struct Node* root) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(root != NULL) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- inorder(root->left);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("%d ", root->data);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- inorder(root->right);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int main() {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- struct Node* root = NULL;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- root = insert(root, 50);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- insert(root, 30);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- insert(root, 70);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- insert(root, 20);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- insert(root, 40);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- insert(root, 60);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- insert(root, 80);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("BST Inorder Traversal: ");

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- inorder(root);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return 0;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- Q3. Write a C program to represent a graph using adjacency matrix.

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #define V 4

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- void printMatrix(int graph[V][V]) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- for(int i=0; i<V; i++) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- for(int j=0; j<V; j++)

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("%d ", graph[i][j]);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("\n");

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int main() {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int graph[V][V] = {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {0, 1, 1, 0},

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {1, 0, 1, 1},

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {1, 1, 0, 1},

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {0, 1, 1, 0}

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- };

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("Adjacency Matrix of Graph:\n");

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printMatrix(graph);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return 0;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- Q4. Write a C program to implement BFS traversal of a graph.

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #include <stdio.h>

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- #define V 5

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int queue[V], front = -1, rear = -1;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- void enqueue(int val) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(rear == V-1) return;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(front == -1) front = 0;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- queue[++rear] = val;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int dequeue() {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(front == -1 || front > rear) return -1;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return queue[front++];

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- void BFS(int graph[V][V], int start) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int visited[V] = {0};

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- enqueue(start);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- visited[start] = 1;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- while(front <= rear) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int node = dequeue();

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("%d ", node);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- for(int i=0; i<V; i++) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- if(graph[node][i] == 1 && !visited[i]) {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- enqueue(i);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- visited[i] = 1;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int main() {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- int graph[V][V] = {

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {0,1,1,0,0},

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {1,0,0,1,1},

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {1,0,0,1,0},

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {0,1,1,0,1},

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- {0,1,0,1,0}

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- };

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- printf("BFS Traversal: ");

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- BFS(graph, 0);

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- return 0;

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- }

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- 100 Data Structures PYQs with Complete C Solutions

5.5 UNIT 5 PYQS (PROGRAMMING IN C)

- All problems include a brief statement, complete C solution (or compact outline for very advanced topics), and time complexity.

Q1. REVERSE AN ARRAY

- Reverse the elements of an array.

Q1. REVERSE AN ARRAY

- #include <stdio.h>
- void reverse(int a[], int n){ for(int i=0;i<n/2;i++){ int t=a[i]; a[i]=a[n-1-i]; a[n-1-i]=t; } }
- int main(){ int a[]={1,2,3,4,5},n=5; reverse(a,n); for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q1. REVERSE AN ARRAY

- Time Complexity: O(n)

Q2. FIND MAXIMUM ELEMENT

- Find max in an array.

Q2. FIND MAXIMUM ELEMENT

- #include <stdio.h>
- int main(){ int a[]={10,45,23,78,56},n=5,max=a[0]; for(int i=1;i<n;i++) if(a[i]>max) max=a[i]; printf("%d",max); return 0; }

Q2. FIND MAXIMUM ELEMENT

- Time Complexity: O(n)

Q3. LINEAR SEARCH

- Search key in unsorted array.

Q3. LINEAR SEARCH

- #include <stdio.h>
- int main(){ int a[]={5,10,15,20},n=4,key=15,found=0; for(int i=0;i<n;i++) if(a[i]==key){found=1;break;} printf(found? "Found":"Not Found"); return 0; }

Q3. LINEAR SEARCH

- Time Complexity: O(n)

Q4. BINARY SEARCH

- Search key in sorted array.

Q4. BINARY SEARCH

- #include <stdio.h>
- int bs(int a[],int n,int key){ int l=0,h=n-1; while(l<=h){ int m=(l+h)/2; if(a[m]==key) return m; if(a[m]<key) l=m+1; else h=m-1;} return -1; }
- int main(){ int a[]={10,20,30,40,50}; int idx=bs(a,5,30); printf("%d",idx); return 0; }

Q4. BINARY SEARCH

- Time Complexity: O(log n)

Q5. INSERT ELEMENT

- Insert value at position (1-indexed).

Q5. INSERT ELEMENT

- #include <stdio.h>
- int main(){ int a[10]={1,2,3,4,5},n=5,pos=3,val=99; for(int i=n;i>=pos;i--) a[i]=a[i-1]; a[pos-1]=val;n++; for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q5. INSERT ELEMENT

- Time Complexity: O(n)

Q6. DELETE ELEMENT

- Delete element at position.

Q6. DELETE ELEMENT

- #include <stdio.h>
- int main(){ int a[]={1,2,3,4,5},n=5,pos=2; for(int i=pos-1;i<n-1;i++) a[i]=a[i+1]; n--; for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q6. DELETE ELEMENT

- Time Complexity: O(n)

Q7. BUBBLE SORT

- Sort array using Bubble Sort.

Q7. BUBBLE SORT

- #include <stdio.h>
- int main(){ int a[]={5,1,4,2,8},n=5; for(int i=0;i<n-1;i++) for(int j=0;j<n-i-1;j++) if(a[j]>a[j+1]){int t=a[j];a[j]=a[j+1];a[j+1]=t;} for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q7. BUBBLE SORT

- Time Complexity: O(n^2)

Q8. SELECTION SORT

- Sort array using Selection Sort.

Q8. SELECTION SORT

- #include <stdio.h>
- int main(){ int a[]={64,25,12,22,11},n=5; for(int i=0;i<n-1;i++){ int m=i; for(int j=i+1;j<n;j++) if(a[j]<a[m]) m=j; int t=a[m];a[m]=a[i];a[i]=t;} for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q8. SELECTION SORT

- Time Complexity: O(n^2)

Q9. INSERTION SORT

- Sort array using Insertion Sort.

Q9. INSERTION SORT

- #include <stdio.h>
- int main(){ int a[]={12,11,13,5,6},n=5; for(int i=1;i<n;i++){ int key=a[i],j=i-1; while(j>=0 && a[j]>key){ a[j+1]=a[j]; j--; } a[j+1]=key; } for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q9. INSERTION SORT

- Time Complexity: O(n^2)

Q10. MERGE SORT

- Divide-and-conquer sort.

Q10. MERGE SORT

- #include <stdio.h>
- void merge(int a[],int l,int m,int r){ int n1=m-l+1,n2=r-m,i=0,j=0,k=l; int L[n1],R[n2]; for(i=0;i<n1;i++) L[i]=a[l+i]; for(j=0;j<n2;j++) R[j]=a[m+1+j]; i=0;j=0; while(i<n1&&j<n2) a[k++]=(L[i]<=R[j])?L[i++]:R[j++]; while(i<n1) a[k++]=L[i++]; while(j<n2) a[k++]=R[j++]; }
- void ms(int a[],int l,int r){ if(l<r){ int m=(l+r)/2; ms(a,l,m); ms(a,m+1,r); merge(a,l,m,r);} }
- int main(){ int a[]={12,11,13,5,6,7}; ms(a,0,5); for(int i=0;i<6;i++) printf("%d ",a[i]); return 0; }

Q10. MERGE SORT

- Time Complexity: O(n log n)

Q11. QUICK SORT (LOMUTO)

- In-place quicksort using Lomuto partition.

Q11. QUICK SORT (LOMUTO)

- #include <stdio.h>
- int part(int a[],int l,int r){ int p=a[r],i=l; for(int j=l;j<r;j++) if(a[j]<=p){ int t=a[i];a[i]=a[j];a[j]=t; i++; } int t=a[i];a[i]=a[r];a[r]=t; return i; }
- void qs(int a[],int l,int r){ if(l<r){ int pi=part(a,l,r); qs(a,l,pi-1); qs(a,pi+1,r);} }
- int main(){ int a[]={10,7,8,9,1,5}; qs(a,0,5); for(int i=0;i<6;i++) printf("%d ",a[i]); return 0; }

Q11. QUICK SORT (LOMUTO)

- Time Complexity: Average O(n log n), Worst O(n^2)

Q12. HEAP SORT

- Sort using max-heap.

Q12. HEAP SORT

- #include <stdio.h>
- void heapify(int a[],int n,int i){ int l=2*i+1,r=2*i+2,m=i; if(l<n&&a[l]>a[m]) m=l; if(r<n&&a[r]>a[m]) m=r; if(m!=i){ int t=a[i];a[i]=a[m];a[m]=t; heapify(a,n,m);} }
- void hs(int a[],int n){ for(int i=n/2-1;i>=0;i--) heapify(a,n,i); for(int i=n-1;i>0;i--){ int t=a[0];a[0]=a[i];a[i]=t; heapify(a,i,0);} }
- int main(){ int a[]={12,11,13,5,6,7},n=6; hs(a,n); for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q12. HEAP SORT

- Time Complexity: O(n log n)

Q13. ROTATE ARRAY BY K

- Rotate array left by k positions (reversal algorithm).

Q13. ROTATE ARRAY BY K

- #include <stdio.h>
- void rev(int a[],int l,int r){ while(l<r){ int t=a[l];a[l]=a[r];a[r]=t; l++; r--; } }
- void rotate(int a[],int n,int k){ k%=n; rev(a,0,k-1); rev(a,k,n-1); rev(a,0,n-1); }
- int main(){ int a[]={1,2,3,4,5,6,7}; rotate(a,7,2); for(int i=0;i<7;i++) printf("%d ",a[i]); return 0; }

Q13. ROTATE ARRAY BY K

- Time Complexity: O(n)

Q14. SECOND LARGEST ELEMENT

- Find second largest distinct element.

Q14. SECOND LARGEST ELEMENT

- #include <stdio.h>
- int main(){ int a[]={12,35,1,10,34,1},n=6,first=-1e9,second=-1e9; for(int i=0;i<n;i++){ if(a[i]>first){ second=first; first=a[i]; } else if(a[i]!=first && a[i]>second) second=a[i]; } printf("%d",second); return 0; }

Q14. SECOND LARGEST ELEMENT

- Time Complexity: O(n)

Q15. KADANE'S MAXIMUM SUBARRAY SUM

- Find max subarray sum.

Q15. KADANE'S MAXIMUM SUBARRAY SUM

- #include <stdio.h>
- int main(){ int a[]={-2,-3,4,-1,-2,1,5,-3},n=8, max=a[0],cur=a[0]; for(int i=1;i<n;i++){ if(cur<0) cur=a[i]; else cur+=a[i]; if(cur>max) max=cur;} printf("%d",max); return 0; }

Q15. KADANE'S MAXIMUM SUBARRAY SUM

- Time Complexity: O(n)

Q16. TWO SUM (SORTED) – TWO POINTERS

- Check if two numbers sum to X.

Q16. TWO SUM (SORTED) – TWO POINTERS

- #include <stdio.h>
- int main(){ int a[]={1,2,4,4},n=4,x=8,l=0,r=n-1,ok=0; while(l<r){ int s=a[l]+a[r]; if(s==x){ok=1;break;} else if(s<x) l++; else r--; } printf(ok?"Yes":"No"); return 0; }

Q16. TWO SUM (SORTED) – TWO POINTERS

- Time Complexity: O(n)

Q17. MATRIX TRANSPOSE

- Transpose an N×N matrix in-place.

Q17. MATRIX TRANSPOSE

- #include <stdio.h>
- int main(){ int n=3,a[3][3]={{1,2,3},{4,5,6},{7,8,9}}; for(int i=0;i<n;i++) for(int j=i+1;j<n;j++){ int t=a[i][j]; a[i][j]=a[j][i]; a[j][i]=t; } for(int i=0;i<n;i++){ for(int j=0;j<n;j++) printf("%d ",a[i][j]); printf("\n"); } return 0; }

Q17. MATRIX TRANSPOSE

- Time Complexity: O(n^2)

Q18. SEARCH IN ROW/COLUMN SORTED MATRIX

- Search key in matrix sorted by rows and columns.

Q18. SEARCH IN ROW/COLUMN SORTED MATRIX

- #include <stdio.h>
- int main(){ int r=3,c=3,a[3][3]={{1,4,7},{2,5,8},{3,6,9}},x=5,i=0,j=c-1,ok=0; while(i<r && j>=0){ if(a[i][j]==x){ok=1;break;} else if(a[i][j]>x) j--; else i++; } printf(ok?"Found":"Not Found"); return 0; }

Q18. SEARCH IN ROW/COLUMN SORTED MATRIX

- Time Complexity: O(r+c)

Q19. COUNT INVERSIONS (MERGE)

- Count pairs (i<j, a[i]>a[j]).

Q19. COUNT INVERSIONS (MERGE)

- #include <stdio.h>
- long long merge(long long a[],int l,int m,int r){ int n1=m-l+1,n2=r-m; long long L[n1],R[n2]; for(int i=0;i<n1;i++) L[i]=a[l+i]; for(int j=0;j<n2;j++) R[j]=a[m+1+j]; int i=0,j=0,k=l; long long inv=0; while(i<n1 && j<n2){ if(L[i]<=R[j]) a[k++]=L[i++]; else { a[k++]=R[j++]; inv += (n1 - i); } } while(i<n1) a[k++]=L[i++]; while(j<n2) a[k++]=R[j++]; return inv; }
- long long ms(long long a[],int l,int r){ if(l>=r) return 0; int m=(l+r)/2; long long inv=0; inv+=ms(a,l,m); inv+=ms(a,m+1,r); inv+=merge(a,l,m,r); return inv; }
- int main(){ long long a[]={2,4,1,3,5}; printf("%lld", ms(a,0,4)); return 0; }

Q19. COUNT INVERSIONS (MERGE)

- Time Complexity: O(n log n)

Q20. DUTCH NATIONAL FLAG (0/1/2 SORT)

- Sort array of 0s,1s,2s.

Q20. DUTCH NATIONAL FLAG (0/1/2 SORT)

- #include <stdio.h>
- int main(){ int a[]={2,0,2,1,1,0},n=6,l=0,m=0,h=n-1; while(m<=h){ if(a[m]==0){int t=a[l];a[l]=a[m];a[m]=t; l++; m++;} else if(a[m]==1) m++; else {int t=a[m];a[m]=a[h];a[h]=t; h--; } } for(int i=0;i<n;i++) printf("%d ",a[i]); return 0; }

Q20. DUTCH NATIONAL FLAG (0/1/2 SORT)

- Time Complexity: O(n)

Q21. MAJORITY ELEMENT (BOYER–MOORE)

- Find element > n/2 if exists.

Q21. MAJORITY ELEMENT (BOYER–MOORE)

- #include <stdio.h>
- int main(){ int a[]={2,2,1,1,1,2,2},n=7,cand=0,count=0; for(int i=0;i<n;i++){ if(count==0){cand=a[i];count=1;} else if(a[i]==cand) count++; else count--; } // verify
- int cnt=0; for(int i=0;i<n;i++) if(a[i]==cand) cnt++; printf(cnt>n/2?"%d":"No", cand); return 0; }

Q21. MAJORITY ELEMENT (BOYER–MOORE)

- Time Complexity: O(n)

Q22. MERGE TWO SORTED ARRAYS

- Merge into a single sorted array.

Q22. MERGE TWO SORTED ARRAYS

- #include <stdio.h>
- int main(){ int a[]={1,3,5},b[]={2,4,6},n=3,m=3,i=0,j=0; while(i<n && j<m) printf("%d ", (a[i]<=b[j])?a[i++]:b[j++]); while(i<n) printf("%d ",a[i++]); while(j<m) printf("%d ",b[j++]); return 0; }

Q22. MERGE TWO SORTED ARRAYS

- Time Complexity: O(n+m)

Q23. EQUILIBRIUM INDEX

- Find index where left sum == right sum.

Q23. EQUILIBRIUM INDEX

- #include <stdio.h>
- int main(){ int a[]={-7,1,5,2,-4,3,0},n=7,total=0,left=0,idx=-1; for(int i=0;i<n;i++) total+=a[i]; for(int i=0;i<n;i++){ total-=a[i]; if(left==total){idx=i;break;} left+=a[i]; } printf("%d",idx); return 0; }

Q23. EQUILIBRIUM INDEX

- Time Complexity: O(n)

Q24. PAIR WITH GIVEN SUM (HASHING)

- Check if any pair sums to X (unsorted).

Q24. PAIR WITH GIVEN SUM (HASHING)

- #include <stdio.h>
- #define SIZE 101
- int H[SIZE];
- int main(){ int a[]={8,7,2,5,3,1},n=6,x=10; for(int i=0;i<n;i++){ int need=x-a[i]; if(need>=0 && H[need]){ printf("Yes"); return 0;} H[a[i]]=1;} printf("No"); return 0; }

Q24. PAIR WITH GIVEN SUM (HASHING)

- Time Complexity: O(n) average

Q25. PREFIX SUM RANGE QUERY

- Compute sum l..r using prefix sums.

Q25. PREFIX SUM RANGE QUERY

- #include <stdio.h>
- int main(){ int a[]={1,2,3,4,5},n=5,p[6]={0}; for(int i=1;i<=n;i++) p[i]=p[i-1]+a[i-1]; int l=2,r=4; printf("%d", p[r]-p[l-1]); return 0; }

Q25. PREFIX SUM RANGE QUERY

- Time Complexity: O(n) build, O(1) query

Q26. SINGLY LINKED LIST: INSERT AT HEAD

- Implement insertion at head.

Q26. SINGLY LINKED LIST: INSERT AT HEAD

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int data; struct Node* next; } Node;
- void push(Node** head,int x){ Node* n=(Node*)malloc(sizeof(Node)); n->data=x; n->next=*head; *head=n; }
- void print(Node* h){ while(h){ printf("%d ",h->data); h=h->next; } }
- int main(){ Node* head=NULL; push(&head,3); push(&head,2); push(&head,1); print(head); return 0; }

Q26. SINGLY LINKED LIST: INSERT AT HEAD

- Time Complexity: O(1)

Q27. SINGLY LINKED LIST: DELETE BY KEY

- Delete first occurrence of key.

Q27. SINGLY LINKED LIST: DELETE BY KEY

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int data; struct Node* next; } Node;
- void del(Node** head,int key){ Node* t=*head,*prev=NULL; while(t && t->data!=key){ prev=t; t=t->next; } if(!t) return; if(!prev) *head=t->next; else prev->next=t->next; free(t); }

Q27. SINGLY LINKED LIST: DELETE BY KEY

- Time Complexity: O(n)

Q28. REVERSE A SINGLY LINKED LIST

- Iterative reversal.

Q28. REVERSE A SINGLY LINKED LIST

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int data; struct Node* next; } Node;
- Node* rev(Node* h){ Node* p=NULL; while(h){ Node* n=h->next; h->next=p; p=h; h=n; } return p; }

Q28. REVERSE A SINGLY LINKED LIST

- Time Complexity: O(n)

Q29. DETECT LOOP IN LINKED LIST (FLOYD)

- Use tortoise and hare.

Q29. DETECT LOOP IN LINKED LIST (FLOYD)

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int data; struct Node* next; } Node;
- int hasLoop(Node* h){ Node *s=h,*f=h; while(f && f->next){ s=s->next; f=f->next->next; if(s==f) return 1; } return 0; }

Q29. DETECT LOOP IN LINKED LIST (FLOYD)

- Time Complexity: O(n)

Q30. INTERSECTION OF TWO LINKED LISTS

- Find merge point by length difference.

Q30. INTERSECTION OF TWO LINKED LISTS

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int data; struct Node* next; } Node;
- int len(Node* h){ int c=0; while(h){c++;h=h->next;} return c; }
- Node* advance(Node* h,int k){ while(k--) h=h->next; return h; }
- Node* intersect(Node* a,Node* b){ int la=len(a), lb=len(b); if(la>lb) a=advance(a,la-lb); else b=advance(b,lb-la); while(a&&b){ if(a==b) return a; a=a->next; b=b->next; } return NULL; }

Q30. INTERSECTION OF TWO LINKED LISTS

- Time Complexity: O(n+m)

Q31. STACK USING ARRAY

- Implement push, pop, peek.

Q31. STACK USING ARRAY

- #include <stdio.h>
- #define MAX 100
- int st[MAX], top=-1;
- void push(int x){ if(top==MAX-1) return; st[++top]=x; }
- int pop(){ return (top==-1)?-1:st[top--]; }
- int peek(){ return (top==-1)?-1:st[top]; }

Q31. STACK USING ARRAY

- Time Complexity: O(1)

Q32. STACK USING LINKED LIST

- Stack ops with list.

Q32. STACK USING LINKED LIST

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int data; struct Node* next; } Node;
- void push(Node** t,int x){ Node* n=(Node*)malloc(sizeof(Node)); n->data=x; n->next=*t; *t=n; }
- int pop(Node** t){ if(!*t) return -1; Node* tmp=*t; int v=tmp->data; *t=tmp->next; free(tmp); return v; }

Q32. STACK USING LINKED LIST

- Time Complexity: O(1)

Q33. BALANCED PARENTHESES (STACK)

- Check balanced brackets.

Q33. BALANCED PARENTHESES (STACK)

- #include <stdio.h>
- #define MAX 1000
- char st[MAX]; int top=-1;
- int match(char a,char b){ return (a=='('&&b==')') | | (a=='['&&b==']') | | (a=='{'&&b=='}'); }
- int isBalanced(char* s){ for(int i=0;s[i];i++){ char c=s[i]; if(c=='(' | | c=='[' | | c=='{') st[++top]=c; else { if(top==-1 | | !match(st[top],c)) return 0; top--; } } return top==-1; }
- int main(){ char s[]="{[()]}"; printf(isBalanced(s)?"Yes":"No"); return 0; }

Q33. BALANCED PARENTHESES (STACK)

- Time Complexity: O(n)

Q34. INFIX TO POSTFIX (SHUNTING-YARD)

- Convert infix to postfix.

Q34. INFIX TO POSTFIX (SHUNTING-YARD)

```
#include <stdio.h>
#include <ctype.h>
#define MAX 1000
char st[MAX]; int top=-1;
int prec(char c){ if(c=='^') return 3; if(c=='*' || c=='/') return 2;
if(c=='+' || c=='-') return 1; return 0; }
int main(){ char in[]="a+b*(c-d)"; char out[MAX]; int k=0;
 for(int i=0; in[i]; i++){ char c=in[i];
  if(isalnum(c)) out[k++]=c;
  else if(c=='(') st[++top]=c;
  else if(c==')'){ while(top!=-1 && st[top]!='(') out[k++]=st[top--]; top--; }
  else { while(top!=-1 && prec(st[top])>=prec(c)) out[k++]=st[top--];
st[++top]=c; } }
 while(top!=-1) out[k++]=st[top--]; out[k]='\0'; printf("%s",out); return 0; }
```

Q34. INFIX TO POSTFIX (SHUNTING-YARD)

- Time Complexity: O(n)

Q35. EVALUATE POSTFIX

- Evaluate postfix expression with stack.

Q35. EVALUATE POSTFIX

- #include <stdio.h>
- #include <ctype.h>
- #define MAX 1000
- int st[MAX], top=-1;
- int main(){ char p[]="23*54*+9-"; for(int i=0;p[i];i++){ char c=p[i]; if(isdigit(c)) st[++top]=c-'0'; else { int b=st[top--], a=st[top--]; int r= (c=='+')?a+b:(c=='-')?a-b:(c=='*')?a*b:a/b; st[++top]=r; } } printf("%d", st[top]); return 0; }

Q35. EVALUATE POSTFIX

- Time Complexity: O(n)

Q36. QUEUE USING ARRAY (CIRCULAR)

- Implement circular queue.

Q36. QUEUE USING ARRAY (CIRCULAR)

- #include <stdio.h>
- #define MAX 5
- int q[MAX], front=0, rear=0, cnt=0;
- void enq(int x){ if(cnt==MAX) return; q[rear]=x; rear=(rear+1)%MAX; cnt++; }
- int deq(){ if(cnt==0) return -1; int v=q[front]; front=(front+1)%MAX; cnt--; return v; }

Q36. QUEUE USING ARRAY (CIRCULAR)

- Time Complexity: O(1)

Q37. QUEUE USING LINKED LIST

- Enqueue/Dequeue with list.

Q37. QUEUE USING LINKED LIST

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int data; struct Node* next; } Node;
- typedef struct{ Node *f,*r; } Q;
- void enq(Q* q,int x){ Node* n=(Node*)malloc(sizeof(Node)); n->data=x;n->next=NULL; if(!q->r) q->f=q->r=n; else {q->r->next=n; q->r=n;} }
- int deq(Q* q){ if(!q->f) return -1; Node* t=q->f; int v=t->data; q->f=t->next; if(!q->f) q->r=NULL; free(t); return v; }

Q37. QUEUE USING LINKED LIST

- Time Complexity: O(1)

Q38. DEQUE (ARRAY)

- Double-ended queue operations.

Q38. DEQUE (ARRAY)

- #include <stdio.h>
- #define MAX 10
- int dq[MAX],f=-1,r=-1;
- int isFull(){ return (f==0 && r==MAX-1) || (f==r+1); }
- int isEmpty(){ return f==-1; }
- void insertFront(int x){ if(isFull()) return; if(f==-1){ f=r=0; } else if(f==0) f=MAX-1; else f--; dq[f]=x; }
- void insertRear(int x){ if(isFull()) return; if(f==-1){ f=r=0; } else if(r==MAX-1) r=0; else r++; dq[r]=x; }
- int deleteFront(){ if(isEmpty()) return -1; int v=dq[f]; if(f==r) f=r=-1; else if(f==MAX-1) f=0; else f++; return v; }
- int deleteRear(){ if(isEmpty()) return -1; int v=dq[r]; if(f==r) f=r=-1; else if(r==0) r=MAX-1; else r--; return v; }

Q38. DEQUE (ARRAY)

- Time Complexity: O(1)

Q39. PRIORITY QUEUE (MAX-HEAP)

- Insert and extract-max.

Q39. PRIORITY QUEUE (MAX-HEAP)

- #include <stdio.h>
- #define MAX 100
- int h[MAX],sz=0;
- void insert(int x){ int i=sz++; h[i]=x; while(i>0 && h[(i-1)/2]<h[i]){ int t=h[i];h[i]=h[(i-1)/2];h[(i-1)/2]=t; i=(i-1)/2; } }
- int extract(){ int r=h[0]; h[0]=h[--sz]; int i=0; while(1){ int l=2*i+1,rn=2*i+2,m=i; if(l<sz&&h[l]>h[m]) m=l; if(rn<sz&&h[rn]>h[m]) m=rn; if(m==i) break; int t=h[i];h[i]=h[m];h[m]=t; i=m; } return r; }

Q39. PRIORITY QUEUE (MAX-HEAP)

- Time Complexity: Insert/Delete O(log n)

Q40. NEXT GREATER ELEMENT (STACK)

- Find next greater element for each item.

Q40. NEXT GREATER ELEMENT (STACK)

- #include <stdio.h>
- #define MAX 100
- int st[MAX],top=-1;
- int main(){ int a[]={4,5,2,25},n=4,ans[4]; for(int i=0;i<n;i++){ while(top!=-1 && a[st[top]]<a[i]){ ans[st[top]]=a[i]; top--; } st[++top]=i; } while(top!=-1){ ans[st[top]]=-1; top--; } for(int i=0;i<n;i++) printf("%d -> %d\n",a[i],ans[i]); return 0; }

Q40. NEXT GREATER ELEMENT (STACK)

- Time Complexity: O(n)

Q41. LRU CACHE (ARRAY + COUNTERS, SIMPLE)

- Simulate LRU page replacement (simplified).

Q41. LRU CACHE (ARRAY + COUNTERS, SIMPLE)

- #include <stdio.h>
- #define F 3
- int frame[F]={-1,-1,-1}, age[F]={0};
- int main(){ int ref[]={7,0,1,2,0,3,0,4,2,3,0,3},n=12,hit=0,miss=0; for(int t=0;t<n;t++){ int p=ref[t],pos=-1; for(int i=0;i<F;i++){ age[i]++; if(frame[i]==p){pos=i;break;} } if(pos!=-1){ hit++; age[pos]=0; } else { miss++; int repl=0; for(int i=1;i<F;i++) if(age[i]>age[repl]) repl=i; frame[repl]=p; age[repl]=0; } } printf("Hits=%d Miss=%d",hit,miss); return 0; }

Q41. LRU CACHE (ARRAY + COUNTERS, SIMPLE)

- Time Complexity: O(n*F)

Q42. BINARY SEARCH TREE: INSERT & INORDER

- Create BST and inorder traverse.

Q42. BINARY SEARCH TREE: INSERT & INORDER

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int key; struct Node *l,*r; } Node;
- Node* new(int k){ Node* n=(Node*)malloc(sizeof(Node)); n->key=k;n->l=n->r=NULL; return n; }
- Node* ins(Node* r,int k){ if(!r) return new(k); if(k<r->key) r->l=ins(r->l,k); else if(k>r->key) r->r=ins(r->r,k); return r; }
- void inorder(Node* r){ if(!r) return; inorder(r->l); printf("%d ",r->key); inorder(r->r); }
- int main(){ Node* r=NULL; int a[]={50,30,20,40,70,60,80}; for(int i=0;i<7;i++) r=ins(r,a[i]); inorder(r); return 0; }

Q42. BINARY SEARCH TREE: INSERT & INORDER

- Time Complexity: Insert O(h)

Q43. BST SEARCH & DELETE

- Delete a node in BST.

Q43. BST SEARCH & DELETE

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct Node{ int key; struct Node *l,*r; } Node;
- Node* new(int k){ Node* n=(Node*)malloc(sizeof(Node)); n->key=k;n->l=n->r=NULL; return n; }
- Node* ins(Node* r,int k){ if(!r) return new(k); if(k<r->key) r->l=ins(r->l,k); else if(k>r->key) r->r=ins(r->r,k); return r; }
- Node* minNode(Node* r){ while(r->l) r=r->l; return r; }
- Node* del(Node* r,int k){ if(!r) return r; if(k<r->key) r->l=del(r->l,k); else if(k>r->key) r->r=del(r->r,k); else { if(!r->l){ Node* t=r->r; free(r); return t;} else if(!r->r){ Node* t=r->l; free(r); return t;} Node* t=minNode(r->r); r->key=t->key; r->r=del(r->r,t->key);} return r; }

Q43. BST SEARCH & DELETE

- Time Complexity: O(h)

Q44. TREE TRAVERSALS (RECURSIVE)

- Preorder, Inorder, Postorder.

Q44. TREE TRAVERSALS (RECURSIVE)

- #include <stdio.h>
- typedef struct Node{ int d; struct Node *l,*r; } Node;
- void pre(Node* r){ if(!r) return; printf("%d ",r->d); pre(r->l); pre(r->r); }
- void in(Node* r){ if(!r) return; in(r->l); printf("%d ",r->d); in(r->r); }
- void post(Node* r){ if(!r) return; post(r->l); post(r->r); printf("%d ",r->d); }

Q44. TREE TRAVERSALS (RECURSIVE)

- Time Complexity: O(n)

Q45. HEIGHT OF BINARY TREE

- Compute height (levels).

Q45. HEIGHT OF BINARY TREE

- #include <stdio.h>
- typedef struct Node{ int d; struct Node *l,*r; } Node;
- int h(Node* r){ if(!r) return -1; int lh=h(r->l), rh=h(r->r); return (lh>rh?lh:rh)+1; }

Q45. HEIGHT OF BINARY TREE

- Time Complexity: O(n)

Q46. CHECK BALANCED BINARY TREE

- Height-balanced check.

Q46. CHECK BALANCED BINARY TREE

- #include <stdio.h>
- typedef struct Node{ int d; struct Node *l,*r; } Node;
- int bal(Node* r){ if(!r) return 0; int lh=bal(r->l); if(lh==-1) return -1; int rh=bal(r->r); if(rh==-1) return -1; if(lh-rh>1 | |rh-lh>1) return -1; return (lh>rh?lh:rh)+1; }

Q46. CHECK BALANCED BINARY TREE

- Time Complexity: O(n)

Q47. LOWEST COMMON ANCESTOR (BST)

- Find LCA in BST.

Q47. LOWEST COMMON ANCESTOR (BST)

- #include <stdio.h>
- typedef struct Node{ int k; struct Node *l,*r; } Node;
- Node* lca(Node* r,int a,int b){ while(r){ if(a<r->k && b<r->k) r=r->l; else if(a>r->k && b>r->k) r=r->r; else return r; } return NULL; }

Q47. LOWEST COMMON ANCESTOR (BST)

- Time Complexity: O(h)

Q48. AVL TREE INSERTION

- Self-balancing BST (rotations).

Q48. AVL TREE INSERTION

- #include <stdio.h>
- #include <stdlib.h>
- typedef struct N{ int k,h; struct N *l,*r; } N;
- int H(N* n){ return n?n->h:0; }
- int max(int a,int b){ return a>b?a:b; }
- N* newN(int k){ N* n=(N*)malloc(sizeof(N)); n->k=k;n->l=n->r=NULL;n->h=1; return n; }
- N* rrot(N* y){ N* x=y->l; N* T=x->r; x->r=y; y->l=T; y->h=max(H(y->l),H(y->r))+1; x->h=max(H(x->l),H(x->r))+1; return x; }
- N* lrot(N* x){ N* y=x->r; N* T=y->l; y->l=x; x->r=T; x->h=max(H(x->l),H(x->r))+1; y->h=max(H(y->l),H(y->r))+1; return y; }
- int balF(N* n){ return n?H(n->l)-H(n->r):0; }
- N* ins(N* n,int k){ if(!n) return newN(k); if(k<n->k) n->l=ins(n->l,k); else if(k>n->k) n->r=ins(n->r,k); else return n; n->h=1+max(H(n->l),H(n->r)); int b=balF(n); if(b>1 && k<n->l->k) return rrot(n); if(b<-1 && k>n->r->k) return lrot(n); if(b>1 && k>n->l->k){ n->l=lrot(n->l); return rrot(n);} if(b<-1 && k<n->r->k){ n->r=rrot(n->r); return lrot(n);} return n; }

Q48. AVL TREE INSERTION

- Time Complexity: O(log n)

Q49. BINARY HEAP (MIN-HEAP)

- Insert and extract-min.

Q49. BINARY HEAP (MIN-HEAP)

- #include <stdio.h>
- #define MAX 100
- int h[MAX],sz=0;
- void up(int i){ while(i>0 && h[(i-1)/2]>h[i]){ int t=h[i];h[i]=h[(i-1)/2];h[(i-1)/2]=t; i=(i-1)/2; } }
- void down(int i){ while(1){ int l=2*i+1,r=2*i+2,m=i; if(l<sz&&h[l]<h[m]) m=l; if(r<sz&&h[r]<h[m]) m=r; if(m==i) break; int t=h[i];h[i]=h[m];h[m]=t; i=m; } }
- void insert(int x){ h[sz]=x; up(sz++); }
- int extract(){ int r=h[0]; h[0]=h[--sz]; down(0); return r; }

Q49. BINARY HEAP (MIN-HEAP)

- Time Complexity: Insert/Delete O(log n)

Q50. TRIE INSERT & SEARCH (LOWERCASE)

- Prefix tree for strings.

Q50. TRIE INSERT & SEARCH (LOWERCASE)

- #include <stdio.h>
- #include <stdlib.h>
- #define A 26
- typedef struct T{ struct T* c[A]; int end; } T;
- T* newT(){ T* n=(T*)malloc(sizeof(T)); n->end=0; for(int i=0;i<A;i++) n->c[i]=NULL; return n; }
- void insert(T* r, const char* s){ for(int i=0;s[i];i++){ int idx=s[i]-'a'; if(!r->c[idx]) r->c[idx]=newT(); r=r->c[idx]; } r->end=1; }
- int search(T* r, const char* s){ for(int i=0;s[i];i++){ int idx=s[i]-'a'; if(!r->c[idx]) return 0; r=r->c[idx]; } return r->end; }

Q50. TRIE INSERT & SEARCH (LOWERCASE)

- Time Complexity: Insert/Search O(m)

Q51. HUFFMAN CODING (OUTLINE)

- Build optimal prefix codes using min-heap (outline).

Q51. HUFFMAN CODING (OUTLINE)

- // Outline: create nodes with freq, build min-heap, repeatedly extract two min, merge, insert back.
- // Due to length, full implementation is omitted here; see Set 6/74 for notes.
- int main(){ return 0; }

Q51. HUFFMAN CODING (OUTLINE)

- Time Complexity: O(n log n) to build

Q52. GRAPH BFS (ADJACENCY LIST)

- Breadth-first search from source.

Q52. GRAPH BFS (ADJACENCY LIST)

- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 100
- typedef struct Node{ int v; struct Node* next; } Node;
- Node* adj[MAX]; int vis[MAX];
- void addE(int u,int v){ Node* n=(Node*)malloc(sizeof(Node)); n->v=v; n->next=adj[u]; adj[u]=n; }
- void bfs(int s){ int q[MAX],f=0,r=0; vis[s]=1; q[r++]=s; while(f<r){ int u=q[f++]; printf("%d ",u); for(Node* p=adj[u];p;p=p->next) if(!vis[p->v]){ vis[p->v]=1; q[r++]=p->v; } } }

Q52. GRAPH BFS (ADJACENCY LIST)

- Time Complexity: O(V+E)

Q53. GRAPH DFS (RECURSIVE)

- Depth-first traversal.

Q53. GRAPH DFS (RECURSIVE)

- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 100
- typedef struct Node{ int v; struct Node* next; } Node;
- Node* adj[MAX]; int vis[MAX];
- void addE(int u,int v){ Node* n=(Node*)malloc(sizeof(Node)); n->v=v; n->next=adj[u]; adj[u]=n; }
- void dfs(int u){ vis[u]=1; printf("%d ",u); for(Node* p=adj[u];p;p=p->next) if(!vis[p->v]) dfs(p->v); }

Q53. GRAPH DFS (RECURSIVE)

- Time Complexity: O(V+E)

Q54. TOPOLOGICAL SORT (KAHN)

- Topo order for DAG.

Q54. TOPOLOGICAL SORT (KAHN)

- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 100
- typedef struct Node{ int v; struct Node* next; } Node;
- Node* adj[MAX]; int indeg[MAX];
- void addE(int u,int v){ Node* n=(Node*)malloc(sizeof(Node)); n->v=v; n->next=adj[u]; adj[u]=n; indeg[v]++; }
- void topo(int V){ int q[MAX],f=0,r=0, cnt=0; for(int i=0;i<V;i++) if(indeg[i]==0) q[r++]=i; while(f<r){ int u=q[f++]; printf("%d ",u); cnt++; for(Node* p=adj[u];p;p=p->next){ if(--indeg[p->v]==0) q[r++]=p->v; } } if(cnt!=V) printf("(cycle)"); }

Q54. TOPOLOGICAL SORT (KAHN)

- Time Complexity: O(V+E)

Q55. DIJKSTRA (ADJACENCY MATRIX)

- Shortest paths from source with non-negative weights.

Q55. DIJKSTRA (ADJACENCY MATRIX)

- #include <stdio.h>
- #define V 9
- #define INF 1e9
- int minDist(int dist[], int spt[]){ int m=INF,idx=-1; for(int v=0;v<V;v++) if(!spt[v] && dist[v]<=m){ m=dist[v]; idx=v; } return idx; }
- void dijkstra(int g[V][V], int src){ int dist[V],spt[V]={0}; for(int i=0;i<V;i++) dist[i]=INF; dist[src]=0; for(int c=0;c<V-1;c++){ int u=minDist(dist,spt); spt[u]=1; for(int v=0;v<V;v++) if(!spt[v] && g[u][v] && dist[u]+g[u][v]<dist[v]) dist[v]=dist[u]+g[u][v]; } for(int i=0;i<V;i++) printf("%d ",dist[i]); }

Q55. DIJKSTRA (ADJACENCY MATRIX)

- Time Complexity: O(V^2)

Q56. KRUSKAL'S MST (UNION-FIND)

- Minimum spanning tree.

Q56. KRUSKAL'S MST (UNION-FIND)

- #include <stdio.h>
- #include <stdlib.h>
- struct Edge{ int u,v,w; };
- int comp(const void* a,const void* b){ return ((struct Edge*)a)->w - ((struct Edge*)b)->w; }
- int parent[100],rnk[100];
- int find(int x){ return parent[x]==x?x:(parent[x]=find(parent[x])); }
- void unite(int a,int b){ a=find(a); b=find(b); if(a!=b){ if(rnk[a]<rnk[b]) parent[a]=b; else if(rnk[b]<rnk[a]) parent[b]=a; else { parent[b]=a; rnk[a]++; } } }
- int main(){ struct Edge e[]={{0,1,10},{0,2,6},{0,3,5},{1,3,15},{2,3,4}}; int E=5,V=4,w=0; for(int i=0;i<V;i++){parent[i]=i;rnk[i]=0;} qsort(e,E,sizeof(struct Edge),comp); for(int i=0, cnt=0; i<E && cnt<V-1; i++){ if(find(e[i].u)!=find(e[i].v)){ unite(e[i].u,e[i].v); w+=e[i].w; cnt++; } } printf("MST=%d",w); return 0; }

Q56. KRUSKAL'S MST (UNION-FIND)

- Time Complexity: O(E log E)

Q57. PRIM'S MST (ADJACENCY MATRIX)

- Minimum spanning tree using Prim.

Q57. PRIM'S MST (ADJACENCY MATRIX)

- #include <stdio.h>
- #define V 5
- #define INF 1e9
- int minKey(int key[], int mst[]){ int m=INF,idx=-1; for(int v=0;v<V;v++) if(!mst[v] && key[v]<m){ m=key[v]; idx=v; } return idx; }
- void prim(int g[V][V]){ int key[V],mst[V]={0},parent[V]; for(int i=0;i<V;i++){ key[i]=INF; parent[i]=-1; } key[0]=0; for(int c=0;c<V-1;c++){ int u=minKey(key,mst); mst[u]=1; for(int v=0;v<V;v++) if(g[u][v] && !mst[v] && g[u][v]<key[v]){ parent[v]=u; key[v]=g[u][v]; } } int sum=0; for(int i=1;i<V;i++) sum+=g[i][parent[i]]; printf("MST=%d",sum); }

Q57. PRIM'S MST (ADJACENCY MATRIX)

- Time Complexity: O(V^2)

Q58. DETECT CYCLE IN UNDIRECTED GRAPH (DSU)

- Cycle detection using Union-Find.

Q58. DETECT CYCLE IN UNDIRECTED GRAPH (DSU)

- #include <stdio.h>
- struct Edge{ int u,v; };
- int parent[100];
- int find(int x){ return parent[x]==x?x:(parent[x]=find(parent[x])); }
- int unite(int a,int b){ a=find(a); b=find(b); if(a==b) return 1; parent[b]=a; return 0; }
- int main(){ struct Edge e[]={{0,1},{1,2},{2,0}}; int V=3,E=3; for(int i=0;i<V;i++) parent[i]=i; for(int i=0;i<E;i++) if(unite(e[i].u,e[i].v)) { printf("Cycle"); return 0;} printf("No Cycle"); return 0; }

Q58. DETECT CYCLE IN UNDIRECTED GRAPH (DSU)

- Time Complexity: O(E α(V))

Q59. TOPOLOGICAL SORT (DFS)

- Topo order using DFS stack.

Q59. TOPOLOGICAL SORT (DFS)

- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 100
- typedef struct Node{ int v; struct Node* next; } Node;
- Node* adj[MAX]; int vis[MAX], st[MAX], top=-1;
- void addE(int u,int v){ Node* n=(Node*)malloc(sizeof(Node)); n->v=v; n->next=adj[u]; adj[u]=n; }
- void dfs(int u){ vis[u]=1; for(Node* p=adj[u];p;p=p->next) if(!vis[p->v]) dfs(p->v); st[++top]=u; }

Q59. TOPOLOGICAL SORT (DFS)

- Time Complexity: O(V+E)

Q60. GRAPH CONNECTED COMPONENTS (DFS)

- Count components.

Q60. GRAPH CONNECTED COMPONENTS (DFS)

- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 100
- typedef struct Node{ int v; struct Node* next; } Node;
- Node* adj[MAX]; int vis[MAX];
- void addE(int u,int v){ Node* n=(Node*)malloc(sizeof(Node)); n->v=v; n->next=adj[u]; adj[u]=n; }
- void dfs(int u){ vis[u]=1; for(Node* p=adj[u];p;p=p->next) if(!vis[p->v]) dfs(p->v); }
- int main(){ int V=5,comp=0; addE(0,1); addE(1,0); addE(2,3); addE(3,2); for(int i=0;i<V;i++) if(!vis[i]){ comp++; dfs(i);} printf("%d",comp); return 0; }

Q60. GRAPH CONNECTED COMPONENTS (DFS)

- Time Complexity: O(V+E)

Q61. SHORTEST PATH IN UNWEIGHTED GRAPH (BFS)

- Compute distances.

Q61. SHORTEST PATH IN UNWEIGHTED GRAPH (BFS)

- #include <stdio.h>
- #include <stdlib.h>
- #define MAX 100
- typedef struct Node{ int v; struct Node* next; } Node;
- Node* adj[MAX]; int dist[MAX];
- void addE(int u,int v){ Node* n=(Node*)malloc(sizeof(Node)); n->v=v; n->next=adj[u]; adj[u]=n; }
- void sp(int s,int V){ int q[MAX],f=0,r=0,vis[MAX]={0}; for(int i=0;i<V;i++) dist[i]=-1; vis[s]=1; dist[s]=0; q[r++]=s; while(f<r){ int u=q[f++]; for(Node* p=adj[u];p;p=p->next) if(!vis[p->v]){ vis[p->v]=1; dist[p->v]=dist[u]+1; q[r++]=p->v; } } }

Q61. SHORTEST PATH IN UNWEIGHTED GRAPH (BFS)

- Time Complexity: O(V+E)

Q62. HASHING WITH LINEAR PROBING

- Open addressing hash table.

Q62. HASHING WITH LINEAR PROBING

- #include <stdio.h>
- #define S 10
- int H[S];
- void insert(int k){ int i=k%S; while(H[i]!=0) i=(i+1)%S; H[i]=k; }
- int search(int k){ int i=k%S,s=i; while(H[i]!=k){ if(H[i]==0) return -1; i=(i+1)%S; if(i==s) return -1;} return i; }
- int main(){ insert(12); insert(22); insert(32); printf("%d", search(22)); return 0; }

Q62. HASHING WITH LINEAR PROBING

- Time Complexity: Avg O(1)

Q63. HASHING WITH QUADRATIC PROBING

- Resolve collisions quadratically.

Q63. HASHING WITH QUADRATIC PROBING

- #include <stdio.h>
- #define S 10
- int H[S];
- void insert(int k){ int i=k%S,c=0; while(H[(i+c*c)%S]!=0) c++; H[(i+c*c)%S]=k; }

Q63. HASHING WITH QUADRATIC PROBING

- Time Complexity: Avg O(1)

Q64. SEPARATE CHAINING HASHING

- Buckets as linked lists.

Q64. SEPARATE CHAINING HASHING

- #include <stdio.h>
- #include <stdlib.h>
- #define S 10
- typedef struct Node{ int d; struct Node* next; } Node;
- Node* HT[S];
- int h(int k){ return k%S; }
- void insert(int k){ int i=h(k); Node* n=(Node*)malloc(sizeof(Node)); n->d=k; n->next=HT[i]; HT[i]=n; }

Q64. SEPARATE CHAINING HASHING

- Time Complexity: Avg O(1)

Q65. COUNTING SORT

- Stable counting sort for small range.

Q65. COUNTING SORT

- #include <stdio.h>
- void cs(int a[],int n,int m){ int c[m+1]; for(int i=0;i<=m;i++) c[i]=0; for(int i=0;i<n;i++) c[a[i]]++; for(int i=1;i<=m;i++) c[i]+=c[i-1]; int out[n]; for(int i=n-1;i>=0;i--) out[--c[a[i]]]=a[i]; for(int i=0;i<n;i++) a[i]=out[i]; }

Q65. COUNTING SORT

- Time Complexity: O(n+k)

Q66. RADIX SORT (LSD)

- Digits by counting sort.

Q66. RADIX SORT (LSD)

- #include <stdio.h>
- int getMax(int a[],int n){ int m=a[0]; for(int i=1;i<n;i++) if(a[i]>m) m=a[i]; return m; }
- void cexp(int a[],int n,int e){ int out[n], c[10]={0}; for(int i=0;i<n;i++) c[(a[i]/e)%10]++; for(int i=1;i<10;i++) c[i]+=c[i-1]; for(int i=n-1;i>=0;i--){ int d=(a[i]/e)%10; out[--c[d]]=a[i]; } for(int i=0;i<n;i++) a[i]=out[i]; }
- void radix(int a[],int n){ int m=getMax(a,n); for(int e=1;m/e>0;e*=10) cexp(a,n,e); }

Q66. RADIX SORT (LSD)

- Time Complexity: O(nk)

Q67. BLOOM FILTER (TOY)

- Probabilistic set membership.

Q67. BLOOM FILTER (TOY)

- #include <stdio.h>
- #define S 50
- int B[S];
- int h1(char*s){ int h=0; while(*s) h=(h+*s++)%S; return h; }
- int h2(char*s){ int h=1; while(*s) h=(h*(*s++))%S; return h; }
- void insert(char*s){ B[h1(s)]=B[h2(s)]=1; }
- int query(char*s){ return B[h1(s)]&&B[h2(s)]; }

Q67. BLOOM FILTER (TOY)

- Time Complexity: Insert/Query O(k)

Q68. DISJOINT SET (UNION-FIND)

- Path compression + union by rank.

Q68. DISJOINT SET (UNION-FIND)

- #include <stdio.h>
- int p[100], r[100];
- void make(int n){ for(int i=0;i<n;i++){p[i]=i;r[i]=0;} }
- int find(int x){ return p[x]==x?x:(p[x]=find(p[x])); }
- void uni(int a,int b){ a=find(a); b=find(b); if(a==b) return; if(r[a]<r[b]) p[a]=b; else if(r[b]<r[a]) p[b]=a; else { p[b]=a; r[a]++; } }

Q68. DISJOINT SET (UNION-FIND)

- Time Complexity: α(n)

Q69. FLOYD–WARSHALL

- All-pairs shortest paths.

Q69. FLOYD–WARSHALL

- #include <stdio.h>
- #define INF 99999
- #define V 4
- void fw(int g[V][V]){ int d[V][V]; for(int i=0;i<V;i++) for(int j=0;j<V;j++) d[i][j]=g[i][j];
- for(int k=0;k<V;k++) for(int i=0;i<V;i++) for(int j=0;j<V;j++) if(d[i][k]+d[k][j]<d[i][j]) d[i][j]=d[i][k]+d[k][j];
- for(int i=0;i<V;i++){ for(int j=0;j<V;j++) printf(d[i][j]==INF?"INF ":"%d ",d[i][j]); printf("\n"); } }

Q69. FLOYD–WARSHALL

- Time Complexity: O(V^3)

Q70. BELLMAN–FORD

- Single-source shortest path with negatives.

Q70. BELLMAN–FORD

- #include <stdio.h>
- #include <limits.h>
- struct E{int u,v,w;};
- void bf(struct E e[],int V,int E,int s){ int d[V]; for(int i=0;i<V;i++) d[i]=INT_MAX; d[s]=0;
- for(int i=1;i<=V-1;i++) for(int j=0;j<E;j++) if(d[e[j].u]!=INT_MAX && d[e[j].u]+e[j].w<d[e[j].v]) d[e[j].v]=d[e[j].u]+e[j].w;
- for(int j=0;j<E;j++) if(d[e[j].u]!=INT_MAX && d[e[j].u]+e[j].w<d[e[j].v]){ printf("Neg cycle"); return; }
- for(int i=0;i<V;i++) printf("%d ",d[i]); }

Q70. BELLMAN–FORD

- Time Complexity: O(VE)

Q71. FORD–FULKERSON (EDMONDS–KARP BFS)

- Max flow in network.

Q71. FORD–FULKERSON (EDMONDS–KARP BFS)

- #include <stdio.h>
- #include <string.h>
- #include <limits.h>
- #define V 6
- int bfs(int r[V][V],int s,int t,int p[]){ int q[100],f=0,rn=0,vis[V]={0}; q[rn++]=s; vis[s]=1; p[s]=-1;
- while(f<rn){ int u=q[f++]; for(int v=0;v<V;v++) if(!vis[v]&&r[u][v]>0){ q[rn++]=v; p[v]=u; vis[v]=1; } } return vis[t]; }
- int maxflow(int g[V][V],int s,int t){ int r[V][V]; for(int i=0;i<V;i++) for(int j=0;j<V;j++) r[i][j]=g[i][j]; int p[V],flow=0;
- while(bfs(r,s,t,p)){ int pf=INT_MAX; for(int v=t;v!=s;v=p[v]){ int u=p[v]; if(r[u][v]<pf) pf=r[u][v]; } for(int v=t;v!=s;v=p[v]){ int u=p[v]; r[u][v]-=pf; r[v][u]+=pf; } flow+=pf; } return flow; }

Q71. FORD–FULKERSON (EDMONDS–KARP BFS)

- Time Complexity: O(VE^2) for EK

Q72. FIBONACCI (DP)

- Bottom-up DP for nth Fibonacci.

Q72. FIBONACCI (DP)

- #include <stdio.h>
- int main(){ int n=10, f[n+2]; f[0]=0; f[1]=1; for(int i=2;i<=n;i++) f[i]=f[i-1]+f[i-2]; printf("%d",f[n]); return 0; }

Q72. FIBONACCI (DP)

- Time Complexity: O(n)

Q73. LONGEST COMMON SUBSEQUENCE (DP)

- Length of LCS.

Q73. LONGEST COMMON SUBSEQUENCE (DP)

- #include <stdio.h>
- #include <string.h>
- int main(){ char X[]="AGGTAB", Y[]="GXTXAYB"; int m=strlen(X),n=strlen(Y), L[m+1][n+1];
- for(int i=0;i<=m;i++) for(int j=0;j<=n;j++) if(i==0||j==0) L[i][j]=0; else if(X[i-1]==Y[j-1]) L[i][j]=L[i-1][j-1]+1; else L[i][j]=(L[i-1][j]>L[i][j-1])?L[i-1][j]:L[i][j-1];
- printf("%d",L[m][n]); return 0; }

Q73. LONGEST COMMON SUBSEQUENCE (DP)

- Time Complexity: O(mn)

Q74. LONGEST INCREASING SUBSEQUENCE (O(N LOG N))

- Patience sorting method.

Q74. LONGEST INCREASING SUBSEQUENCE (O(N LOG N))

- #include <stdio.h>
- int ceilidx(int a[],int t[],int l,int r,int key){ while(r-l>1){ int m=l+(r-l)/2; if(a[t[m]]>=key) r=m; else l=m; } return r; }
- int LIS(int a[],int n){ int tail[n],idx[n],len=1; tail[0]=0; idx[0]=a[0];
- for(int i=1;i<n;i++){ if(a[i]<idx[0]) idx[0]=a[i]; else if(a[i]>idx[len-1]) idx[len++]=a[i]; else idx[ceilidx(idx,tail,-1,len-1,a[i])]=a[i]; } return len; }
- int main(){ int a[]={10,22,9,33,21,50,41,60}; printf("%d", LIS(a,8)); return 0; }

Q74. LONGEST INCREASING SUBSEQUENCE (O(N LOG N))

- Time Complexity: O(n log n)

Q75. 0/1 KNAPSACK (DP)

- Max value within capacity.

Q75. 0/1 KNAPSACK (DP)

- #include <stdio.h>
- int max(int a,int b){return a>b?a:b;}
- int main(){ int v[]={60,100,120}, w[]={10,20,30}, n=3, W=50; int K[n+1][W+1];
- for(int i=0;i<=n;i++) for(int wt=0; wt<=W; wt++) if(i==0||wt==0) K[i][wt]=0; else if(w[i-1]<=wt) K[i][wt]=max(v[i-1]+K[i-1][wt-w[i-1]], K[i-1][wt]); else K[i][wt]=K[i-1][wt];
- printf("%d",K[n][W]); return 0; }

Q75. 0/1 KNAPSACK (DP)

- Time Complexity: O(nW)

Q76. MATRIX CHAIN MULTIPLICATION (DP)

- Minimum multiplication cost.

Q76. MATRIX CHAIN MULTIPLICATION (DP)

- #include <stdio.h>
- #define INF 1e9
- int min(int a,int b){return a<b?a:b;}
- int main(){ int p[]={1,2,3,4}, n=4; int m[n][n]; for(int i=1;i<n;i++) m[i][i]=0;
- for(int L=2; L<n; L++) for(int i=1;i<n-L+1;i++){ int j=i+L-1; m[i][j]=INF; for(int k=i;k<j;k++) m[i][j]=min(m[i][j], m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j]); }
- printf("%d", m[1][n-1]); return 0; }

Q76. MATRIX CHAIN MULTIPLICATION (DP)

- Time Complexity: O(n^3)

Q77. ACTIVITY SELECTION (GREEDY)

- Max non-overlapping activities.

Q77. ACTIVITY SELECTION (GREEDY)

- #include <stdio.h>
- #include <stdlib.h>
- struct Act{ int s,f; };
- int cmp(const void*a,const void*b){ return ((struct Act*)a)->f - ((struct Act*)b)->f; }
- int main(){ struct Act a[]={{1,2},{3,4},{0,6},{5,7},{8,9},{5,9}}; int n=6; qsort(a,n,sizeof(struct Act),cmp); int cnt=1,last=0; for(int i=1;i<n;i++) if(a[i].s>=a[last].f){ cnt++; last=i; } printf("%d",cnt); return 0; }

Q77. ACTIVITY SELECTION (GREEDY)

- Time Complexity: O(n log n)

Q78. JOB SEQUENCING WITH DEADLINES (GREEDY)

- Maximize profit.

Q78. JOB SEQUENCING WITH DEADLINES (GREEDY)

- #include <stdio.h>
- #include <stdlib.h>
- struct Job{ int id,dead,profit; };
- int cmp(const void*a,const void*b){ return ((struct Job*)b)->profit - ((struct Job*)a)->profit; }
- int main(){ struct Job j[]={{1,2,100},{2,1,19},{3,2,27},{4,1,25},{5,3,15}}; int n=5,slot[10]={0},res=0;
- qsort(j,n,sizeof(struct Job),cmp);
- for(int i=0;i<n;i++) for(int t=j[i].dead; t>0; t--) if(!slot[t]){ slot[t]=1; res+=j[i].profit; break; }
- printf("%d",res); return 0; }

Q78. JOB SEQUENCING WITH DEADLINES (GREEDY)

- Time Complexity: O(n^2)

Q79. FRACTIONAL KNAPSACK (GREEDY)

- Max value with fractions.

Q79. FRACTIONAL KNAPSACK (GREEDY)

- #include <stdio.h>
- #include <stdlib.h>
- struct Item{ int w; double v; };
- int cmp(const void*a,const void*b){ double r=((struct Item*)b)->v/((struct Item*)b)->w - ((struct Item*)a)->v/((struct Item*)a)->w; return (r>0)-(r<0); }
- int main(){ struct Item it[]={{10,60},{20,100},{30,120}}; int n=3,W=50; qsort(it,n,sizeof(struct Item),cmp); double val=0; for(int i=0;i<n && W>0;i++){ if(it[i].w<=W){ W-=it[i].w; val+=it[i].v; } else { val+= it[i].v * ((double)W/it[i].w); W=0; } } printf("%.2f",val); return 0; }

Q79. FRACTIONAL KNAPSACK (GREEDY)

- Time Complexity: O(n log n)

Q80. EDIT DISTANCE (LEVENSHTEIN)

- Min edits to convert string A to B.

Q80. EDIT DISTANCE (LEVENSHTEIN)

- #include <stdio.h>
- #include <string.h>
- int min3(int a,int b,int c){ int m=a<b?a:b; return m<c?m:c; }
- int main(){ char a[]="kitten", b[]="sitting"; int m=strlen(a),n=strlen(b), D[m+1][n+1];
- for(int i=0;i<=m;i++) D[i][0]=i; for(int j=0;j<=n;j++) D[0][j]=j;
- for(int i=1;i<=m;i++) for(int j=1;j<=n;j++) D[i][j]= (a[i-1]==b[j-1])? D[i-1][j-1] : 1+min3(D[i-1][j],D[i][j-1],D[i-1][j-1]);
- printf("%d",D[m][n]); return 0; }

Q80. EDIT DISTANCE (LEVENSHTEIN)

- Time Complexity: O(mn)

Q81. TRIE DELETE (WORD DELETION)

- Delete word from trie (mark end=0 if leaf).

Q81. TRIE DELETE (WORD DELETION)

- // Outline: recursively delete child; if child becomes empty and not end, free it; otherwise stop.
- // Full code omitted for brevity.
- int main(){ return 0; }

Q81. TRIE DELETE (WORD DELETION)

- Time Complexity: O(m)

Q82. GRAPH COLORING (BACKTRACKING)

- Color graph with m colors.

Q82. GRAPH COLORING (BACKTRACKING)

- // Outline: try colors 1..m for each vertex, backtrack on conflict.
- int main(){ return 0; }

Q82. GRAPH COLORING (BACKTRACKING)

- Time Complexity: Exponential

Q83. N-QUEENS (BACKTRACKING)

- Place N queens on NxN board.

Q83. N-QUEENS (BACKTRACKING)

- #include <stdio.h>
- #define N 8
- int col[N], d1[2*N], d2[2*N], sol=0;
- void solve(int r){ if(r==N){ sol++; return; } for(int c=0;c<N;c++) if(!col[c] && !d1[r-c+N] && !d2[r+c]){ col[c]=d1[r-c+N]=d2[r+c]=1; solve(r+1); col[c]=d1[r-c+N]=d2[r+c]=0; } }
- int main(){ solve(0); printf("%d",sol); return 0; }

Q83. N-QUEENS (BACKTRACKING)

- Time Complexity: O(N!)

Q84. SUDOKU SOLVER (BACKTRACKING, 9X9)

- Solve Sudoku using backtracking.

Q84. SUDOKU SOLVER (BACKTRACKING, 9X9)

- // Outline due to length: choose empty cell, try 1..9, check row/col/subgrid, recurse; backtrack on failure.
- int main(){ return 0; }

Q84. SUDOKU SOLVER (BACKTRACKING, 9X9)

- Time Complexity: Exponential

Q85. OPTIMAL BST (DP)

- Min expected search cost.

Q85. OPTIMAL BST (DP)

- // Outline: DP over ranges with frequency arrays; m[i][j]=min over roots (m[i][r-1]+m[r+1][j]+sumFreq).
- int main(){ return 0; }

Q85. OPTIMAL BST (DP)

- Time Complexity: O(n^3)

Q86. AVL DELETION (OUTLINE)

- Delete and rebalance.

Q86. AVL DELETION (OUTLINE)

- // Outline: BST delete then fix heights and rotate based on balance factor.
- int main(){ return 0; }

Q86. AVL DELETION (OUTLINE)

- Time Complexity: O(log n)

Q87. B-TREE INSERTION (OUTLINE)

- Split child on overflow.

Q87. B-TREE INSERTION (OUTLINE)

- // Outline: search leaf; if full, split (t-1 keys left/right), promote middle key to parent; may cascade splits.
- int main(){ return 0; }

Q87. B-TREE INSERTION (OUTLINE)

- Time Complexity: O(log n)

Q88. CUCKOO HASHING (CONCEPT)

- Two tables, two hash functions.

Q88. CUCKOO HASHING (CONCEPT)

- // Outline: place key in table1; if occupied, kick out resident to its alternate position; detect cycles -> rehash.
- int main(){ return 0; }

Q88. CUCKOO HASHING (CONCEPT)

- Time Complexity: Amortized O(1)

Q89. JOHNSON'S ALGORITHM (OUTLINE)

- All-pairs shortest paths in sparse graphs.

Q89. JOHNSON'S ALGORITHM (OUTLINE)

- // Outline: add super-source, Bellman-Ford to compute h(v); reweight edges w'(u,v)=w(u,v)+h(u)-h(v); run Dijkstra from each vertex.
- int main(){ return 0; }

Q89. JOHNSON'S ALGORITHM (OUTLINE)

- Time Complexity: O(VE + V^2 log V)

Q90. SHORTEST PATH DAG (DP)

- Topo order + relax.

Q90. SHORTEST PATH DAG (DP)

- #include <stdio.h>
- #define INF 1e9
- // Outline: compute topo order; initialize dist[src]=0; relax edges in topo order.
- int main(){ return 0; }

Q90. SHORTEST PATH DAG (DP)

- Time Complexity: O(V+E)

Q91. ARTICULATION POINTS (TARJAN)

- Find cut vertices.

Q91. ARTICULATION POINTS (TARJAN)

- // Outline: DFS timestamps, low-link values; a root with >=2 children or u where low[v] >= disc[u] is AP.
- int main(){ return 0; }

Q91. ARTICULATION POINTS (TARJAN)

- Time Complexity: O(V+E)

Q92. BRIDGES IN GRAPH (TARJAN)

- Find critical edges.

Q92. BRIDGES IN GRAPH (TARJAN)

- // Outline: DFS with discovery/low arrays; edge (u,v) is bridge if low[v] > disc[u].
- int main(){ return 0; }

Q92. BRIDGES IN GRAPH (TARJAN)

- Time Complexity: O(V+E)

Q93. KMP STRING MATCHING

- Pattern search using lps[] array.

Q93. KMP STRING MATCHING

- #include <stdio.h>
- #include <string.h>
- void lpsb(char* p,int m,int lps[]){ int len=0; lps[0]=0; for(int i=1;i<m;){ if(p[i]==p[len]) lps[i++]=++len; else if(len) len=lps[len-1]; else lps[i++]=0; } }
- void kmp(char* t,char* p){ int n=strlen(t),m=strlen(p), lps[m]; lpsb(p,m,lps); for(int i=0,j=0;i<n;){ if(t[i]==p[j]){ i++; j++; if(j==m){ printf("Found at %d\n", i-j); j=lps[j-1]; } } else if(j) j=lps[j-1]; else i++; } }
- int main(){ char t[]="abxabcabcaby", p[]="abcaby"; kmp(t,p); return 0; }

Q93. KMP STRING MATCHING

- Time Complexity: O(n+m)

Q94. RABIN–KARP STRING MATCHING

- Rolling hash matching.

Q94. RABIN–KARP STRING MATCHING

- #include <stdio.h>
- #include <string.h>
- #define d 256
- #define q 101
- void rk(char* t,char* p){ int n=strlen(t),m=strlen(p); int h=1; for(int i=0;i<m-1;i++) h=(h*d)%q; int ph=0, th=0;
- for(int i=0;i<m;i++){ ph=(d*ph + p[i])%q; th=(d*th + t[i])%q; }
- for(int i=0;i<=n-m;i++){ if(ph==th){ int j=0; while(j<m && t[i+j]==p[j]) j++; if(j==m) printf("Found at %d\n", i); } if(i<n-m){ th=(d*(th - t[i]*h) + t[i+m])%q; if(th<0) th+=q; } } }
- int main(){ char t[]="GEEKS FOR GEEKS", p[]="GEEK"; rk(t,p); return 0; }

Q94. RABIN–KARP STRING MATCHING

- Time Complexity: Average O(n+m)

Q95. TRIE AUTO-COMPLETE (PREFIX LISTING)

- DFS all words with given prefix.

Q95. TRIE AUTO-COMPLETE (PREFIX LISTING)

- // Outline: navigate to prefix node, then DFS collecting words.
- int main(){ return 0; }

Q95. TRIE AUTO-COMPLETE (PREFIX LISTING)

- Time Complexity: O(k + output)

Q96. SEGMENT TREE (RANGE SUM QUERY)

- Build and query sums.

Q96. SEGMENT TREE (RANGE SUM QUERY)

- #include <stdio.h>
- int st[400005], a[100005];
- int build(int p,int l,int r){ if(l==r) return st[p]=a[l]; int m=(l+r)/2; return st[p]=build(p*2,l,m)+build(p*2+1,m+1,r); }
- int query(int p,int l,int r,int i,int j){ if(i>r||j<l) return 0; if(i<=l&&r<=j) return st[p]; int m=(l+r)/2; return query(p*2,l,m,i,j)+query(p*2+1,m+1,r,i,j); }
- void update(int p,int l,int r,int idx,int val){ if(l==r){ st[p]=val; return; } int m=(l+r)/2; if(idx<=m) update(p*2,l,m,idx,val); else update(p*2+1,m+1,r,idx,val); st[p]=st[p*2]+st[p*2+1]; }

Q96. SEGMENT TREE (RANGE SUM QUERY)

- Time Complexity: Build O(n), Query/Update O(log n)

Q97. FENWICK TREE (BIT) FOR PREFIX SUM

- Point update, prefix query.

Q97. FENWICK TREE (BIT) FOR PREFIX SUM

- #include <stdio.h>
- #define N 100005
- int bit[N+1];
- void add(int i,int v){ for(; i<=N; i+=i&-i) bit[i]+=v; }
- int sum(int i){ int s=0; for(; i>0; i-=i&-i) s+=bit[i]; return s; }

Q97. FENWICK TREE (BIT) FOR PREFIX SUM

- Time Complexity: O(log n)

Q98. BINARY SEARCH ON ANSWER

- Min capacity to ship within D days (pattern).

Q98. BINARY SEARCH ON ANSWER

- // Outline: binary search on feasible answer; check() greedily verifies feasibility.
- int main(){ return 0; }

Q98. BINARY SEARCH ON ANSWER

- Time Complexity: O(n log R)

Q99. TWO STACKS IN ONE ARRAY

- Optimize space.

Q99. TWO STACKS IN ONE ARRAY

- #include <stdio.h>
- #define MAX 100
- int a[MAX], t1=-1, t2=MAX;
- void push1(int x){ if(t1+1==t2) return; a[++t1]=x; }
- void push2(int x){ if(t1+1==t2) return; a[--t2]=x; }
- int pop1(){ return t1==-1?-1:a[t1--]; }
- int pop2(){ return t2==MAX?-1:a[t2++]; }

Q99. TWO STACKS IN ONE ARRAY

- Time Complexity: O(1)

Q100. CIRCULAR LINKED LIST: JOSEPHUS

- Find survivor position.

Q100. CIRCULAR LINKED LIST: JOSEPHUS

- #include <stdio.h>
- int josephus(int n,int k){ int r=0; for(int i=1;i<=n;i++) r=(r+k)%i; return r+1; }
- int main(){ printf("%d", josephus(7,3)); return 0; }

Q100. CIRCULAR LINKED LIST: JOSEPHUS

- Time Complexity: O(n)

Q101. LRU CACHE (LINKED LIST + HASH MAP OUTLINE)

- Typical design question.

Q101. LRU CACHE (LINKED LIST + HASH MAP OUTLINE)

- // Outline: doubly linked list for recency, hashmap for O(1) lookup; move node to head on access; evict tail on capacity.
- int main(){ return 0; }

Q101. LRU CACHE (LINKED LIST + HASH MAP OUTLINE)

- Time Complexity: O(1) ops

Q102. BINARY SEARCH TREE TO DLL (INORDER)

- Convert BST to sorted doubly linked list.

Q102. BINARY SEARCH TREE TO DLL (INORDER)

- // Outline: inorder traverse, link prev and current nodes to form DLL.
- int main(){ return 0; }

Q102. BINARY SEARCH TREE TO DLL (INORDER)

- Time Complexity: O(n)