**Roll No…………….. Total No. of Pages:……**

**ST-2 (SET-I)**

**6th SEMESTER 2023-24**

**CS192- Advanced Data Structures**

**Time allowed: 90 Minutes Max. Marks: 40**

**General Instructions:**

* **Follow the instructions given in each section.**
* **Make sure that you attempt the questions in order.**

**SECTION-A (10\*1 mark=10 marks)**

***(All questions are compulsory)***

1. In a recursive function, what is the process of calling the function again within itself called?
   1. Nesting
   2. Repetition
   3. Looping
   4. **Self-invocation**
2. Which of the following is a requirement for a recursive function to work correctly?
   1. The function must have multiple base cases.
   2. The function must call itself at least twice.
   3. The function must have a loop inside.
   4. **The function must have a termination condition.**
3. In dynamic programming, recursion with memoization is often used to optimize problems involving grids. What is memoization?
   1. Keeping a record of function calls and their parameters
   2. Using matrices instead of grids for better performance
   3. **Storing computed values to avoid redundant calculations**
   4. Measuring memory usage during recursion
4. Which type of recursion involves a function calling itself multiple times for smaller sub-problems?
   1. Tail recursion
   2. Linear recursion
   3. Binary recursion
   4. **Tree recursion**
5. The "N-Queens Problem" is a classic example often used to illustrate:
   1. Divide and conquer
   2. Greedy algorithms
   3. Dynamic programming
   4. **Backtracking**
6. What is the main idea behind the "subset sum problem" that makes it a backtracking candidate?
   1. **It involves finding all possible subsets of a set**
   2. It requires sorting the elements of the set
   3. It can be solved using a single loop
   4. It has an exponential number of possible solutions
7. What is the time complexity to delete a node from the end of a singly linked list?
   1. O(1)
   2. **O(n)**
   3. O(log n)
   4. O(n^2)
8. Which type of linked list allows traversal in both directions?
   1. Singly linked list
   2. **Doubly linked list**
   3. Circular linked list
   4. Array-based list
9. Which type of polymorphism is achieved through function overloading?
   1. **Compile-time polymorphism**
   2. Runtime polymorphism
   3. Static polymorphism
   4. Dynamic polymorphism
10. Which keyword is used to implement method overriding in C++?
    1. extends
    2. **virtual**
    3. override
    4. over

**SECTION-B (5\*2 mark=10 marks)**

***(All questions are compulsory)***

1. struct Node {

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

void printList(Node\* head) {

while (head != nullptr) {

cout << head->data << " ";

head = head->next;

}

}

int main() {

Node\* head = new Node(1);

head->next = new Node(2);

head->next->next = new Node(3);

printList(head);

return 0;

}

* 1. **1 2 3**
  2. 3 2 1
  3. 1
  4. Compilation Error

1. class A {

public:

virtual void display() { cout << "A"; }

};

class B : public A {

public:

void display() { cout << "B"; }

};

int main() {

A obj;

A \*ptr = new B;

obj.display();

ptr->display();

return 0;

}

1. **AB**
2. BA
3. AA
4. BB
5. #include <iostream>

using namespace std;

void printPattern(int n) {

if (n <= 0)

return;

cout << n << " ";

printPattern(n - 2);

cout << n << " ";

}

int main() {

printPattern(6);

return 0;

}

a) 6 4 2

b) 6 4 2 4 2 6

**c) 6 4 2 2 4 6**

d) 2 4 6

1. #include <iostream>

using namespace std;

void generateSubsets(string str, int index, string current) {

if (index == str.length()) {

cout << "{" << current << "}" << " ";

return;

}

generateSubsets(str, index + 1, current);

generateSubsets(str, index + 1, current + str[index]);

}

int main() {

generateSubsets("abc", 0, "");

return 0;

}

1. {} {a} {b} {c} {ab} {ac} {bc} {abc}
2. **{} {c} {b} {bc} {a} {ac} {ab} {abc}**
3. {abc} {ab} {ac} {bc} {a} {b} {c} {}
4. {} {a} {b} {c} {ab} {ac} {bc} {abc}
5. Consider the recursive function that calculates the nth term of the sequence: 1, 3, 9, 27, ... where each term is 3 times the previous term. The function is defined as:

int sequenceTerm(int n) {

if (n == 0)

return 1;

return 3 \* sequenceTerm(n - 1);

}

What is the time complexity of this function?

1. O(1)
2. O(log n)
3. O(n)
4. **O(3^n)**

**SECTION-C(Coding Question) (2x5 marks=5 marks)**

Q16) Write a recursive function to find the sum of digits of a positive integer.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | 742 | 98 | 352 |
| **Output** | Sum of digits is: 13 | Sum of digits is: 17 | Sum of digits is: 10 |

Solution :

**#include <iostream>**

**// Recursive function to calculate sum of digits**

**int sumOfDigits(int n) {**

**// Base case: If the number is 0, return 0 as there are no digits to add.**

**if (n == 0)**

**return 0;**

**// Recursive step: Return the last digit (n % 10) and add it to the sum of digits**

**// obtained by calling the function with the remaining digits (n / 10).**

**return n % 10 + sumOfDigits(n / 10);**

**}**

**int main() {**

**int n;**

**std::cout << "Enter a positive integer: ";**

**std::cin >> n;**

**// Call the sumOfDigits function with the input positive integer.**

**int sum = sumOfDigits(n);**

**// Output the sum of digits.**

**std::cout << "Sum of digits is: " << sum << std::endl;**

**return 0;**

**}**

Q17) Implement a singly linked list and delete a node at a given position from the list.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | 8 9 12 4 2 1  Position= 2 | 34 19 9 7  Position= 3 | 1 2 3 4 5 6  Position= 4 |
| **Output** | Linked List after deleting nodes: 8 9 4 2 1 | Linked List after deleting nodes: 34 19 9 | Linked List after deleting nodes: 1 2 3 4 6 |

Solution :

**#include <iostream>**

**using namespace std;**

**// Node structure for singly linked list**

**struct Node {**

**int data;**

**Node\* next;**

**};**

**// Function to insert an element at the end of the linked list**

**Node\* insertAtEnd(Node\* head, int data) {**

**Node\* newNode = new Node;**

**newNode->data = data;**

**newNode->next = nullptr;**

**if (head == nullptr) {**

**return newNode;**

**}**

**Node\* current = head;**

**while (current->next != nullptr) {**

**current = current->next;**

**}**

**current->next = newNode;**

**return head;**

**}**

**// Function to display the linked list**

**void displayLinkedList(Node\* head) {**

**Node\* current = head;**

**while (current != nullptr) {**

**cout << current->data << " ";**

**current = current->next;**

**}**

**}**

**// Function to delete a node at a given position from the linked list**

**Node\* deleteNodeAtPosition(Node\* head, int position) {**

**if (head == nullptr) {**

**return nullptr;**

**}**

**if (position == 0) {**

**Node\* temp = head;**

**head = head->next;**

**delete temp;**

**return head;**

**}**

**Node\* current = head;**

**Node\* prev = nullptr;**

**int currentPosition = 0;**

**while (current != nullptr && currentPosition < position) {**

**prev = current;**

**current = current->next;**

**currentPosition++;**

**}**

**if (current == nullptr) {**

**return head; // Node at the given position not found**

**}**

**prev->next = current->next;**

**delete current;**

**return head;**

**}**

**int main() {**

**Node\* head = nullptr;**

**int data;**

**// Create the linked list with user input**

**cout << "Enter elements for linked list (enter -1 to stop):\n";**

**while (true) {**

**cin >> data;**

**if (data == -1)**

**break;**

**head = insertAtEnd(head, data);**

**}**

**// Display the linked list before deleting nodes**

**cout << "Linked List before deleting nodes: ";**

**displayLinkedList(head);**

**// Prompt the user to enter the position of the node to be deleted**

**int position;**

**cout << "\nEnter the position of the node to be deleted (0-indexed): ";**

**cin >> position;**

**// Delete the node at the given position from the linked list**

**head = deleteNodeAtPosition(head, position);**

**// Display the linked list after deleting nodes**

**cout << "Linked List after deleting nodes: ";**

**displayLinkedList(head);**

**// Free memory by deleting nodes**

**while (head != nullptr) {**

**Node\* temp = head;**

**head = head->next;**

**delete temp;**

**}**

**return 0;**

**}**

**SECTION-D (Coding Question)(1x10 mark=10 mark)**

Q18) Given a partially filled 9×9 Sudoku grid, solve the puzzle and fill the empty cells.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | Enter the Sudoku grid (0 for empty cells):  3 0 6 5 0 8 4 0 0  5 2 0 0 0 0 0 0 0  0 8 7 0 0 0 0 3 1  0 0 3 0 1 0 0 8 0  9 0 0 8 6 3 0 0 5  0 5 0 0 9 0 6 0 0  1 3 0 0 0 0 2 5 0  0 0 0 0 0 0 0 7 4  0 0 5 2 0 6 3 0 0 | Enter the Sudoku grid (0 for empty cells):  3 1 6 5 7 8 4 9 2  5 2 9 1 3 4 7 6 8  4 8 7 6 2 9 5 3 1  2 6 3 0 1 5 9 8 7  9 7 4 8 6 0 1 2 5  8 5 1 7 9 2 6 4 3  1 3 8 0 4 7 2 0 6  6 9 2 3 5 1 8 7 4  7 4 5 0 8 6 3 1 0 | Enter the Sudoku grid (0 for empty cells):  0 1 6 5 7 8 4 9 2  5 2 9 1 3 4 7 6 8  4 8 7 6 2 9 5 3 1  2 6 3 0 1 5 9 8 7  9 7 4 8 6 0 1 2 5  8 5 1 7 9 2 6 4 3  1 3 8 0 4 7 2 0 6  6 9 2 3 5 1 8 7 4  7 4 5 0 8 6 3 1 9 |
| **Output** | Sudoku Solved:  3 1 6 5 7 8 4 9 2  5 2 9 1 3 4 7 6 8  4 8 7 6 2 9 5 3 1  2 6 3 4 1 5 9 8 7  9 7 4 8 6 3 1 2 5  8 5 1 7 9 2 6 4 3  1 3 8 9 4 7 2 5 6  6 9 2 3 5 1 8 7 4  7 4 5 2 8 6 3 1 9 | Sudoku Solved:  3 1 6 5 7 8 4 9 2  5 2 9 1 3 4 7 6 8  4 8 7 6 2 9 5 3 1  2 6 3 4 1 5 9 8 7  9 7 4 8 6 3 1 2 5  8 5 1 7 9 2 6 4 3  1 3 8 9 4 7 2 5 6  6 9 2 3 5 1 8 7 4  7 4 5 2 8 6 3 1 9 | Sudoku Solved:  3 1 6 5 7 8 4 9 2  5 2 9 1 3 4 7 6 8  4 8 7 6 2 9 5 3 1  2 6 3 4 1 5 9 8 7  9 7 4 8 6 3 1 2 5  8 5 1 7 9 2 6 4 3  1 3 8 9 4 7 2 5 6  6 9 2 3 5 1 8 7 4  7 4 5 2 8 6 3 1 9 |

Solution :

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**const int N = 9;**

**bool findUnassignedLocation(vector<vector<int>>& grid, int& row, int& col) {**

**for (row = 0; row < N; row++) {**

**for (col = 0; col < N; col++) {**

**if (grid[row][col] == 0)**

**return true;**

**}**

**}**

**return false;**

**}**

**bool isSafeInRow(vector<vector<int>>& grid, int row, int num) {**

**for (int col = 0; col < N; col++) {**

**if (grid[row][col] == num)**

**return false;**

**}**

**return true;**

**}**

**bool isSafeInColumn(vector<vector<int>>& grid, int col, int num) {**

**for (int row = 0; row < N; row++) {**

**if (grid[row][col] == num)**

**return false;**

**}**

**return true;**

**}**

**bool isSafeInBox(vector<vector<int>>& grid, int boxStartRow, int boxStartCol, int num) {**

**for (int row = 0; row < 3; row++) {**

**for (int col = 0; col < 3; col++) {**

**if (grid[row + boxStartRow][col + boxStartCol] == num)**

**return false;**

**}**

**}**

**return true;**

**}**

**bool isSafe(vector<vector<int>>& grid, int row, int col, int num) {**

**return isSafeInRow(grid, row, num) &&**

**isSafeInColumn(grid, col, num) &&**

**isSafeInBox(grid, row - row % 3, col - col % 3, num);**

**}**

**bool solveSudoku(vector<vector<int>>& grid) {**

**int row, col;**

**if (!findUnassignedLocation(grid, row, col))**

**return true; // Sudoku solved successfully**

**for (int num = 1; num <= 9; num++) {**

**if (isSafe(grid, row, col, num)) {**

**grid[row][col] = num;**

**if (solveSudoku(grid))**

**return true;**

**grid[row][col] = 0; // Unassign if the current choice didn't lead to a solution**

**}**

**}**

**return false; // Backtrack**

**}**

**void printGrid(const vector<vector<int>>& grid) {**

**for (int row = 0; row < N; row++) {**

**for (int col = 0; col < N; col++) {**

**cout << grid[row][col] << " ";**

**}**

**cout << endl;**

**}**

**}**

**int main() {**

**vector<vector<int>> grid(N, vector<int>(N));**

**cout << "Enter the Sudoku grid (0 for empty cells):\n";**

**for (int row = 0; row < N; row++) {**

**for (int col = 0; col < N; col++) {**

**cin >> grid[row][col];**

**}**

**}**

**if (solveSudoku(grid)) {**

**cout << "Sudoku Solved:\n";**

**printGrid(grid);**

**} else {**

**cout << "No solution exists.";**

**}**

**return 0;**

**}**