DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
CHANDIGARH UNIVERSITY
Discover. Learn. Empower.

Experiment 1

Student Name: Shreyansh Singh **UID: 22BCS12333**

Branch: CSE **Section/Group: 634-A**

Semester: 5 **Date of Performance: 19/7/24**

Subject Name: DAA Lab **Subject Code: 22CSH-311**

1. **Aim:**

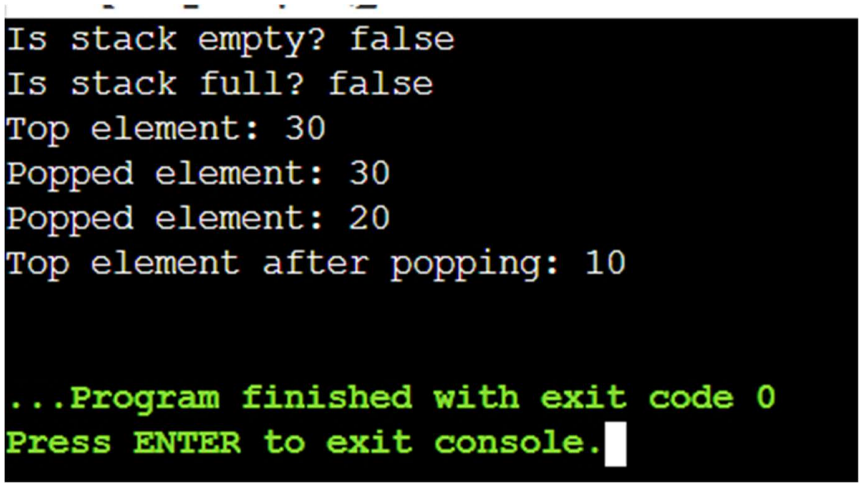
   Analyze if the stack is empty or full, and if elements are present, return the top element in the stack using templates. Also, perform push and pop operations on the stack.

2. **Objective:**
   - Demonstrate the use of the predefined stack container from the C++ Standard Library for basic stack operations.
   - Perform and verify stack operations such as pushing, popping, checking if the stack is empty, and accessing the top element.
   - Provide a clear example of stack manipulation, including checking the stack's status before and after operations

3. **Implementation/Code:**

```java
import java.util.Stack;
import java.util.EmptyStackException;

public class Main
{
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        // Perform push operations
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Check if stack is empty
        System.out.println("Is stack empty? " + stack.isEmpty());

        // Check if stack is full (inbuilt stack does not have a capacity limit, so always false)
        System.out.println("Is stack full? " + isFull(stack));

        // Get the top element
        System.out.println("Top element: " + stack.peek());

        // Perform pop operations
        System.out.println("Popped element: " + stack.pop());
        System.out.println("Popped element: " + stack.pop());

        // Check the top element after popping
        System.out.println("Top element after popping: " + stack.peek());
    }
    public static boolean isFull(Stack<?> stack) {
        return false;
    }
}
```

4. **Output**


Is stack empty? false
Is stack full? false
Top element: 30
Popped element: 30
Popped element: 20
Top element after popping: 10
...Program finished with exit code 0
Press ENTER to exit console.

5. Time Complexity (Applicable for subject DAA)

- Push Operation (s.push(element)): O(1)
- Pop Operation (s.pop()): O(1)
- Check if Empty (s.empty()): O(1)
- Access Top Element (s.top()): O(1)

Time complexity is always constant for stack operations

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment 2

Student Name: Shreyansh Singh **UID: 22BCS12333**
Branch: BE-CSE **Section/Group: 634-A**
Semester: 5th **Date of Performance:26/07/24**
Subject Name: DAA **Subject Code: 22CSH-311**

1. **Aim:** Code implement power function in O(logn) time complexity.

2. **Objective:** To implement power function in O(logn) time complexity.

3. **Procedure/Algorithm: Pseudocode:**

1. Take Two long integers, x (base) and n (exponent).
2. If n is 0, return 1.
3. If n is even:
   Calculate half_pow by recursively calling power(x, n / 2).
   Return half_pow * half_pow.
4. If n is odd:
   Calculate half_pow by recursively calling power(x, n / 2).
   Return half_pow * half_pow * x.
5. In the main function:
6. Initialize x and n.
7. Call the power function with x and n.
8. Print the result as x^n = result.

4. Script and Output: Script:

```java
public class Main
{
    public static long power(long x, long n) {
        if (n == 0) {
            return 1;
        } else if (n % 2 == 0) {
            long halfPow = power(x, n / 2);
            return halfPow * halfPow;
        } else {
            long halfPow = power(x, n / 2);
            return halfPow * halfPow * x;
        }
    }

    public static void main(String[] args) {
        long x = 2;
        long n = 10;
        long result = power(x, n);
        System.out.println(x + "^" + n + " = " + result);
    }
}
```

5. Output:

```
2^10 = 1024


...Program finished with exit code 0
Press ENTER to exit console.
```

6. Learning Outcome:

- **Master Recursion:** Understand the principles of recursive functions and their base and recursive cases.

- **Efficient Exponentiation:** Learn the exponentiation by squaring method to optimize power calculations.
- **Conditional Logic:** Gain skills in using conditional statements to handle different scenarios within a function.
- **Java Implementation:** Implement mathematical algorithms in Java, learning syntax and structure.
- **Algorithm Optimization:** Develop skills in designing and optimizing algorithms for better performance.

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment 3

Student Name: Shreyansh Singh **UID: 22BCS12333**
Branch: BE-CSE **Section/Group: 634-A**
Semester: 5th **Date of Performance:02/08/24**
Subject Name: DAA **Subject Code: 22CSH-311**

1. **Aim:** Code to find frequency of elements in a given array in O(n) time complexity.

2. **Objective:**
   a. Calculate the frequency of each element in a given array, where n is the length of the array.
   b. Hash map and its implementation.

3. **Algorithm:**

   a. **Create a HashMap:** Initialize an empty HashMap to store each element of the array as the key and its frequency as the value.

   b. **Traverse the Array:** Loop through each element in the array. For each element:
      i. If the element is already in the HashMap, increase its count by 1.
      ii. If the element is not in the HashMap, add it with a count of 1.

   c. **Output the Frequencies:** After the loop, iterate through the HashMap and print each element and its corresponding frequency.

4. Code:

```java
import java.util.HashMap;

public class Main {

    public static void findFrequency(int[] arr) {
        HashMap<Integer, Integer> freqMap = new HashMap<>();

        // Traverse the array and store the frequency of each element
        for (int num : arr) {
            freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
        }

        // Output the frequency
        for (int key : freqMap.keySet()) {
            System.out.println("Element: " + key + " -> Frequency: " + freqMap.get(key));
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 1, 2, 3};
        findFrequency(arr);
    }
}
```

5. Output:

```
Element: 1 -> Frequency: 2
Element: 2 -> Frequency: 3
Element: 3 -> Frequency: 2
Element: 4 -> Frequency: 1
```

6. Learning Outcome:

a. Understand HashMap Usage: Learn how to use the HashMap in Java to store and retrieve data efficiently.

b. Implement Frequency Counting: Gain the ability to count the frequency of elements in an array using a single traversal, achieving O(n) time complexity.

7. Time & Space Complexity

a. Time Complexity**:** O (n)
b. Space Complexity: O (1)

Experiment 3

Student Name: Zatch **UID:**

Branch: BE-CSE **Section/Group:**

Semester: 5th **Date of Performance:**

Subject Name: DAA Lab **Subject Code: 22CHS-311**

1. **Aim:** Apply the concept of Linked list and write code to Insert and Delete an element at the beginning and at end in Doubly and Circular Linked List.

2. **Objective:** The objective of this experiment is to implement and demonstrate the basic operations—insertion and deletion—at both the beginning and end of Doubly Linked Lists and Circular Linked Lists, showcasing how these structures can be efficiently managed and manipulated.

3. **Algorithm:**

Algorithm for Doubly Linked List (DLL)

Insertion at the Beginning:

1. **Create a New Node**: Create a new node with the given data.
2. **Check if List is Empty**:
   - If the list is empty, set both head and tail to point to the new node.
   - Otherwise, proceed to the next step.
3. **Link New Node**:
   - Set the new node's next to point to the current head.
   - Set the current head's prev to point to the new node.
4. **Update Head**: Set the new node as the head.

Insertion at the End:

1. **Create a New Node**: Create a new node with the given data.
2. **Check if List is Empty**:
   - If the list is empty, set both head and tail to point to the new node.
   - Otherwise, proceed to the next step.
3. **Link New Node**:
   - Set the current tail's next to point to the new node.
   - Set the new node's prev to point to the current tail.
4. **Update Tail**: Set the new node as the tail.

Deletion from the Beginning:

1. **Check if List is Empty**: If the list is empty, do nothing.

2. **Check if Single Node**:
    - If the list has only one node, delete the node and set both head and tail to nullptr.
    - Otherwise, proceed to the next step.
3. **Delete Node**:
    - Store the current head in a temporary pointer.
    - Move head to the next node.
    - Set the new head's prev to nullptr.
    - Delete the temporary node.

Deletion from the End:

1. **Check if List is Empty**: If the list is empty, do nothing.
2. **Check if Single Node**:
    - If the list has only one node, delete the node and set both head and tail to nullptr.
    - Otherwise, proceed to the next step.
3. **Delete Node**:
    - Store the current tail in a temporary pointer.
    - Move tail to the previous node.
    - Set the new tail's next to nullptr.
    - Delete the temporary node.

Algorithm for Circular Linked List (CLL)

Insertion at the Beginning:

1. **Create a New Node**: Create a new node with the given data.
2. **Check if List is Empty**:
    - If the list is empty, set the head to the new node and point next to itself.
    - Otherwise, proceed to the next step.
3. **Find Last Node**:
    - Traverse the list to find the last node (node whose next points to head).
4. **Link New Node**:
    - Set the new node's next to the current head.
    - Set the last node's next to point to the new node.
5. **Update Head**: Set the new node as the head.

Insertion at the End:

1. **Create a New Node**: Create a new node with the given data.
2. **Check if List is Empty**:
    - If the list is empty, set the head to the new node and point next to itself.
    - Otherwise, proceed to the next step.
3. **Find Last Node**:
    - Traverse the list to find the last node (node whose next points to head).
4. **Link New Node**:
    - Set the last node's next to the new node.
    - Set the new node's next to point to head.

Deletion from the Beginning:

1. **Check if List is Empty**: If the list is empty, do nothing.
2. **Check if Single Node**:
   - If the list has only one node, delete the node and set head to nullptr.
   - Otherwise, proceed to the next step.
3. **Find Last Node**:
   - Traverse the list to find the last node (node whose next points to head).
4. **Delete Node**:
   - Store the current head in a temporary pointer.
   - Move head to the next node.
   - Set the last node's next to point to the new head.
   - Delete the temporary node.

Deletion from the End:

1. **Check if List is Empty**: If the list is empty, do nothing.
2. **Check if Single Node**:
   - If the list has only one node, delete the node and set head to nullptr.
   - Otherwise, proceed to the next step.
3. **Find Second Last Node**:
   - Traverse the list to find the second last node (node whose next points to the last node).
4. **Delete Node**:
   - Store the last node in a temporary pointer.
   - Set the second last node's next to point to head.
   - Delete the temporary node.

4. Implementation/Code:

Doubly Linked List:

```
#include <iostream>
using namespace std;

struct Node {
   int data;
   Node* prev;
   Node* next;

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

class DoublyLinkedList {
public:
   Node* head;
   Node* tail;
```

```
DoublyLinkedList() : head(nullptr), tail(nullptr) {}

// Insert at the beginning
void insertAtBeginning(int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = tail = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    cout << "Inserted " << val << " at the beginning." << endl;
}

// Insert at the end
void insertAtEnd(int val) {
    Node* newNode = new Node(val);
    if (!tail) {
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    cout << "Inserted " << val << " at the end." << endl;
}

// Delete from the beginning
void deleteFromBeginning() {
    if (!head) return;
    Node* temp = head;
    if (head == tail) {
        head = tail = nullptr;
    } else {
        head = head->next;
        head->prev = nullptr;
    }
    cout << "Deleted " << temp->data << " from the beginning." << endl;
    delete temp;
}

// Delete from the end
void deleteFromEnd() {
    if (!tail) return;
```

```
        Node* temp = tail;
        if (head == tail) {
            head = tail = nullptr;
        } else {
            tail = tail->prev;
            tail->next = nullptr;
        }
        cout << "Deleted " << temp->data << " from the end." << endl;
        delete temp;
    }

    void printList() {
        cout << "Current List: ";
        Node* current = head;
        while (current) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

int main() {
    DoublyLinkedList dll;

    dll.insertAtBeginning(3);
    dll.insertAtEnd(4);
    dll.insertAtBeginning(2);
    dll.insertAtEnd(5);
    dll.printList();  // Output: 2 3 4 5

    dll.deleteFromBeginning();
    dll.printList();  // Output: 3 4 5

    dll.deleteFromEnd();
    dll.printList();  // Output: 3 4

    return 0;
    }
```

Circular Linked List:

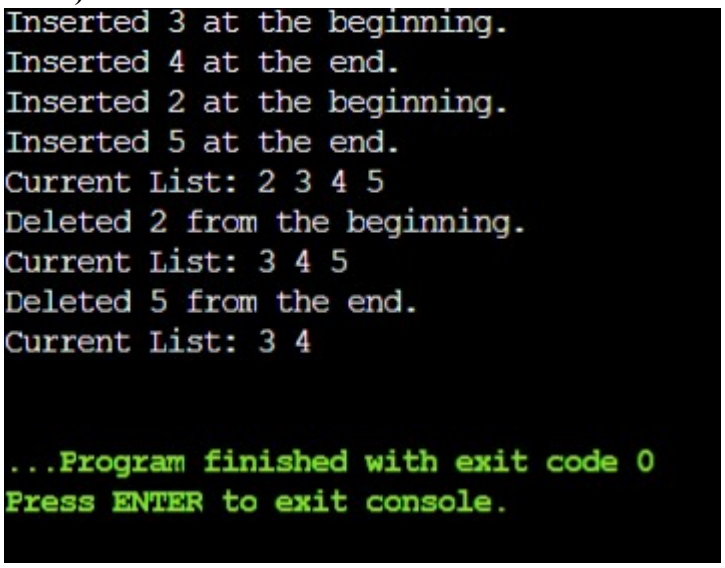
```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

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

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() : head(nullptr) {}

    // Insert at the beginning
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
            head->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            newNode->next = head;
            temp->next = newNode;
            head = newNode;
        }
        cout << "Inserted " << val << " at the beginning." << endl;
    }

    // Insert at the end
    void insertAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
            head->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
```

```
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
    }
    cout << "Inserted " << val << " at the end." << endl;
}

// Delete from the beginning
void deleteFromBeginning() {
    if (!head) return;
    if (head->next == head) {
        cout << "Deleted " << head->data << " from the beginning." << endl;
        delete head;
        head = nullptr;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        Node* delNode = head;
        temp->next = head->next;
        head = head->next;
        cout << "Deleted " << delNode->data << " from the beginning." << endl;
        delete delNode;
    }
}

// Delete from the end
void deleteFromEnd() {
    if (!head) return;
    if (head->next == head) {
        cout << "Deleted " << head->data << " from the end." << endl;
        delete head;
        head = nullptr;
    } else {
        Node* temp = head;
        while (temp->next->next != head) {
            temp = temp->next;
        }
        Node* delNode = temp->next;
        temp->next = head;
        cout << "Deleted " << delNode->data << " from the end." << endl;
        delete delNode;
    }
```

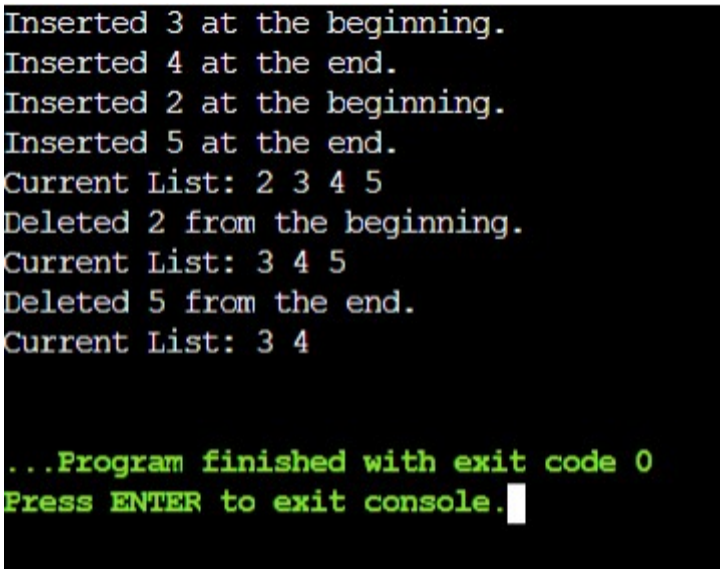
```
    }

    void printList() {
        if (!head) {
            cout << "The list is empty." << endl;
            return;
        }
        cout << "Current List: ";
        Node* current = head;
        do {
            cout << current->data << " ";
            current = current->next;
        } while (current != head);
        cout << endl;
    }
};

int main() {
    CircularLinkedList cll;

    cll.insertAtBeginning(3);
    cll.insertAtEnd(4);
    cll.insertAtBeginning(2);
    cll.insertAtEnd(5);
    cll.printList();  // Output: 2 3 4 5

    cll.deleteFromBeginning();
    cll.printList();  // Output: 3 4 5

    cll.deleteFromEnd();
    cll.printList();  // Output: 3 4

    return 0;
    }
```

5. **Output:**

(Doubly Linked List)



(Circular Linked List)



6. **Time Complexity: O(1) For insertion and deletion**
   O(n) For display
7. **Space Complexity: O(n)**

8. **Learning Outcomes:**
   a. Understand to manage nodes with insertion and deletion at both ends.
   b. Understand time and space complexities.
   c. Understand to apply linked list to solve real world problems.

Experiment 5

Student Name: Zatch **UID:**
Branch: CSE **Section/Group:**
Semester: 5th **Date of Performance:**
Subject Name: DAA LAB **Subject Code: 22CSH-311**

1. **Aim:**

   Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted. The elements can be read from a file or can be generated using the random number generator.

2. **Objective:**

   To sort a list of elements using the Quick sort algorithm and measure the time complexity for different list sizes.

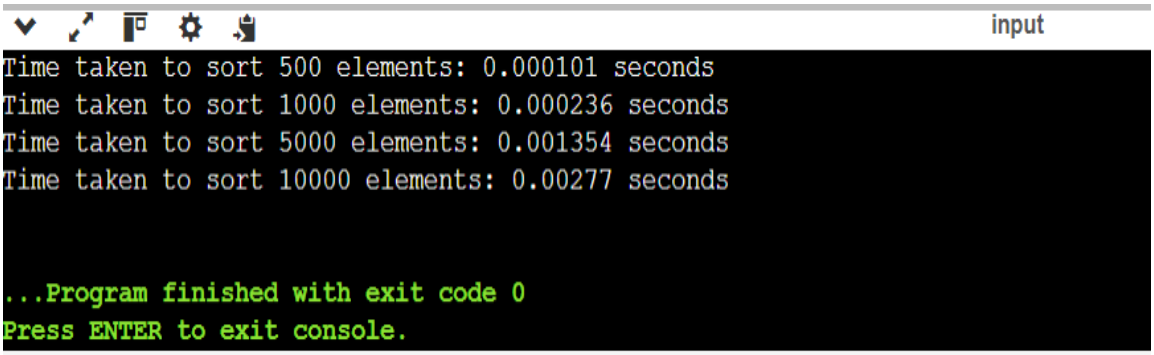
3. **Implementation/Code:**

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

using namespace std;
void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivot = arr[high];
        int i = low - 1;
```

```
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr[i], arr[j]);
            }
        }
        swap(arr[i + 1], arr[high]);
        int pi = i + 1;
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

double measureTime(int n) {
    vector<int> arr(n);
    for (int& num : arr) {
        num = rand() % 10000; // Random numbers
    }

    clock_t start = clock();
    quickSort(arr, 0, n - 1);
    clock_t end = clock();

    return double(end - start) / CLOCKS_PER_SEC;
}
```

```
int main() {
    srand(time(0)); // Seed for random number generation

    int sizes[] = {100, 500, 1000, 5000, 10000}; // Different sizes to test
    for (int n : sizes) {
        double timeTaken = measureTime(n);
        std::cout << "Time taken to sort " << n << " elements: " << timeTaken
<< " seconds" << std::endl;
    }

    return 0;
}
```

4. Output



5. Time Complexity

Average Case: O(nlogn)
Worst Case: O(n^2)

6. Learning Outcome

- Learn the principles and steps involved in the Quick sort algorithm, including the partitioning process and recursive sorting. Understand how the algorithm sorts an array by dividing it into smaller sub-arrays.
- Gain insight into the time and space complexity of Quick sort. Learn to differentiate between the average-case, worst-case, and best-case scenarios and understand the factors influencing Quick sort's performance.
- Compare Quick sort's efficiency with other sorting algorithms like Merge sort, Heap sort, and Bubble sort. Understand the trade-offs between Quick sort’s average-case performance and its worst-case behavior.
- Develop skills in implementing Quick sort in a programming language (e.g., C++). Learn about pivot selection strategies and how they affect the algorithm’s performance. Gain experience in optimizing and debugging sorting algorithms.
- Learn how to measure and analyze the performance of Quick sort. Understand how to use timing functions to evaluate sorting efficiency and how different input sizes impact the sorting time. Develop the ability to interpret performance results and apply this knowledge to real-world scenarios.

Experiment 6

Student Name: Shreyansh Singh **UID: 22BCS12333**

Branch: CSE **Section/Group: 634-A**

Semester: 5 **Date of Performance: 27/9/2024**

Subject Name: DAA Lab **Subject Code: 22CSH-311**

1. Aim:

To implement subset-sum problem using Dynamic Programming

2. Objective

To determine if there exists a subset within a given set of integers that sums up to specified target value. It does by generating possible subsets, calculating their sums, and checking if any subset matches target sum.

3. Algorithm

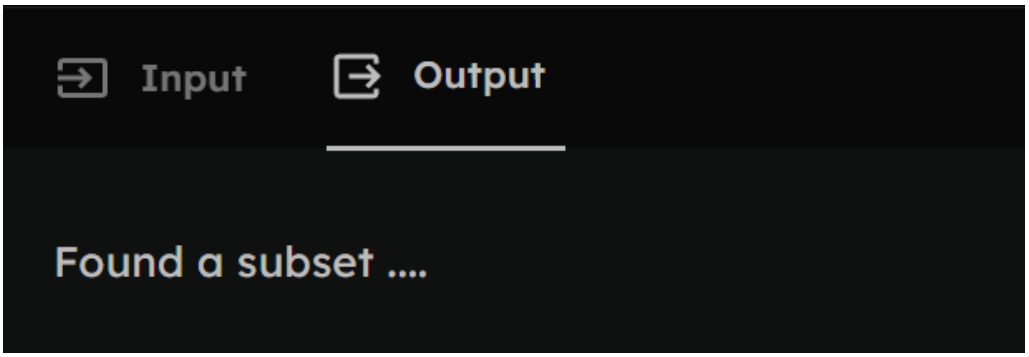
- Initialize the array and target sum.
- Iterate through all possible subsets using bit manipulation.
- For each subset, calculate the sum by checking the bits.
- If the sum of the current subset equals the target, return true.
- Continue checking other subsets if not found.
- If no subset matches the target sum after all iterations, return false.
- Output the result based on the function's return value.

4. Implemetation/Code

```
#include <bits/stdc++.h>
using namespace std;

bool checkSubset(int subset[], int size, int totalsum) {
    int subset_sum = 0;
    for (int i = 0; i < size; ++i) {
        subset_sum += subset[i];
    }
    return subset_sum == totalsum;
}

void generateSubsets(int arr[], int n, int r, int totalsum, bool& found) {
    int subset[n];
    int subset_size = 0;

    for (int i = 0; i < n; ++i) {
        if (r & (1 << i)) {
            subset[subset_size++] = arr[i];
        }
    }

    if (checkSubset(subset, subset_size, totalsum)) {
        found = true;
    }
}

bool subsetsum_DP(int arr[], int n, int totalsum) {
    bool found = false;
    for (int r = 1; r < (1 << n); ++r) {
        generateSubsets(arr, n, r, totalsum, found);
        if (found) {
            return true;
        }
    }
    return false;
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i) {
        cout << arr[i] << " ";
```

```
  }
    cout << endl;
}

int main() {
    int setvalue[] = {7, 200, 10, 18, 26, 23};
    int value = 10;
    int n = sizeof(setvalue) / sizeof(setvalue[0]);

    printArray(setvalue, n);

    if (subsetsum_DP(setvalue, n, value)) {
        cout << "Found a subset" << endl;
    } else {
        cout << "Nope, not in the value" << endl;
    }

    return 0;
}
```

5. Output



6. Time Complexity : O(2^N *N)

7. Space Complexity : O(1)

8. Learning Outcomes:-

1. Understand how to generate all subsets of a set using bit manipulation.
2. Learn how to check for a specific condition (subset sum) in combinatorial problems.
3. Gain experience in handling subset problems using arrays and bitwise operations.

Experiment 7

Student Name: NEMESIS **UID: 22BAI**

Branch: CSE **Section/Group:**

Semester: 5 **Date of Performance:**

Subject Name: DAA Lab **Subject Code: 22CSH-311**

1. **Aim:**
   Develop a program and analyze complexity to implement 0-1 Knapsack using Dynamic Programming.
2. **Objective**

   To implement 0-1 Knapsack using Dynamic Programming.
3. **Algorithm**
   - Initialize a 2D table for storing results.
   - For each item, update the table with the maximum profit for each weight.
   - Consider both including and excluding the item.
   - Repeat for all items and capacities.
   - The final cell in the table contains the maximum profit.

4. **Implemetation/Code**

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 1000;
int knapsack(int W, int wt[], int profit[], int n) {
int dp[MAXN + 1][MAXN + 1] = {0};
for (int i = 1; i <= n; ++i) {
   for (int w = 1; w <= W; ++w) {
```

```
if (wt[i - 1] <= w)
dp[i][w] = max(dp[i - 1][w], profit[i - 1] + dp[i - 1][w - wt[i - 1]]);
else
  dp[i][w] = dp[i - 1][w];
    }
}
    return dp[n][W];
}
int main() {
    int n, W;
    cout << "ENTER NUMBER OF ITEMS AND CAPACITY: ";
    cin >> n >> W;
    if (n <= 0 || W <= 0) {
        cout << "Number of items and capacity must be positive." << endl;
        return 1;
    }
    if (n > MAXN || W > MAXN) {
        cout << "Number of items or capacity exceeds maximum allowed size." << endl;
        return 1;
    }
    int profit[MAXN], wt[MAXN];
    cout << "ENTER PROFITS AND WEIGHTS: ";
    for (int i = 0; i < n; ++i) {
        cin >> profit[i] >> wt[i];
    }
    cout << "MAXIMUM KNAPSACK VALUE: " << knapsack(W, wt, profit, n) <<
endl;
    return 0;
}
```

Output

```
ENTER NUMBER OF ITEMS AND CAPACITY: 3 4
ENTER PROFITS AND WEIGHTS: 1 4
2 5
3 1
MAXIMUM KNAPSACK VALUE: 3
```

5. **Time Complexity** : O(N*W)

6. **Space Complexity :** O(N*W)

7. **Learning Outcomes:-**

1. Learn dynamic programming for optimization problems.
2. Understand how to solve the knapsack problem efficiently.
3. Gain insight into space and time trade-offs.

Experiment 8

Student Name: Zatch **UID:**

Branch: BE-CSE **Section/Group:**

Semester: 5th **Date of Performance:**

Subject Name: DAA Lab **Subject Code: 22CHS-311**

1. **Aim:** Develop a program and analyze complexity to find shortest paths in a graph with positive edge weights using Dijkstra's algorithm.

2. **Objective:** To implement and analyze the efficiency of Dijkstra's algorithm for finding the shortest paths in a graph with positive edge weights. The goal is to evaluate the algorithm's time complexity based on the data structure used for the priority queue (e.g., binary heap or Fibonacci heap).

3. **Algorithm:**

   Step1:- Initialize:

   - Create a distance array dist[] and set all values to infinity (∞) except for the source node, which is set to 0.
   - Use a priority queue (min-heap) to store nodes with their current known shortest distance.

   Step 2:- Push Source:

   - Insert the source node into the priority queue with a distance of 0.

   Step 3:- While Queue is Not Empty:

   - Extract the node u with the minimum distance from the priority queue.
   - For each neighboring node v of u, calculate the tentative distance to v through u.
   - If this distance is smaller than the current distance in dist[v], update dist[v] and insert v into the priority queue with the new distance.

   Step 4:- Repeat:

   - Continue this process until the priority queue is empty.

   Step 5:- Result:

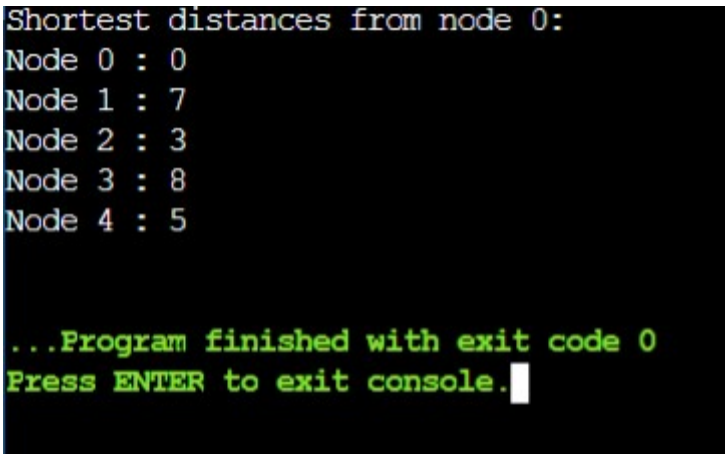
- At the end, dist[] contains the shortest distance from the source node to all other nodes.

4. **Implementation/Code:**

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>

using namespace std;

typedef pair<int, int> PII; // Pair to store (distance, vertex)

// Dijkstra's algorithm to find the shortest paths from a source node
vector<int> dijkstra(int source, const vector<vector<PII>>& graph, int V) {
    priority_queue<PII, vector<PII>, greater<PII>> pq; // Min-heap priority
     queue
    vector<int> dist(V, INT_MAX); // Initialize distances to infinity

    dist[source] = 0; // Distance to the source is 0
    pq.push({0, source}); // Push the source into the priority queue

    while (!pq.empty()) {
        int u = pq.top().second; // Get the vertex with the smallest distance
        pq.pop();

        // Traverse all adjacent vertices of u
        for (const auto& neighbor : graph[u]) {
            int v = neighbor.second;
            int weight = neighbor.first;

            // Relaxation step
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v}); // Push updated distance to the priority queue
            }
        }
    }

    return dist; // Return the shortest distance to all vertices
}
```

```
int main() {
    int V = 5; // Number of vertices
    vector<vector<PII>> graph(V);

    // Adding edges (weight, vertex)
    graph[0].push_back({10, 1});
    graph[0].push_back({3, 2});
    graph[1].push_back({1, 3});
    graph[2].push_back({4, 1});
    graph[2].push_back({8, 3});
    graph[2].push_back({2, 4});
    graph[3].push_back({7, 4});

    int source = 0; // Set the source node

    // Run Dijkstra's algorithm
    vector<int> distances = dijkstra(source, graph, V);

    // Output the shortest distances from the source node
    cout << "Shortest distances from node " << source << ":\n";
    for (int i = 0; i < V; ++i) {
        cout << "Node " << i << " : " << distances[i] << "\n";
    }

    return 0;
}
```

5. **Output:**



6. **Time Complexity:**

   Using Binary Heap: O((V + E) log V), where V is the number of vertices and E is the number of edges.

   Using Fibonacci Heap: O(E + V log V).

   Space Complexity:

   The overall **space complexity** of Dijkstra's algorithm is **O(V + E)**, where **V** is the number of vertices and **E** is the number of edges.

7. **Learning Outcomes:**

   1. **Understanding of Dijkstra's Algorithm**: You will learn how to implement and apply Dijkstra's algorithm to find the shortest paths in graphs with positive edge weights, utilizing a priority queue for efficiency.
   2. **Time and Space Complexity Analysis**: You will gain insights into analyzing the time complexity (O((V + E) log V) using a binary heap) and space complexity (O(V + E)) of the algorithm.
   3. **Graph Representation**: You will understand how to represent graphs efficiently using adjacency lists and handle shortest path problems with real-world applications.

Experiment 9

Student Name: Zatch **UID:**

Branch: BE-CSE **Section/Group:**

Semester: 5th **Date of Performance:**

Subject Name: DAA Lab **Subject Code: 22CHS-311**

1. **Aim:** Develop a program and analyze complexity to find all occurrences of a pattern P in a given strings.

2. **Objective:** To develop a program to find all occurrences of a pattern P in a string S, and analyze the time complexity based on different algorithms (e.g., brute force, KMP, or Boyer-Moore). Compare performance for varying pattern lengths and input sizes.

3. **Algorithm:**

Step 1: Build LPS Array (Longest Prefix Suffix)

- Initialize an array lps[] of size m (length of P).
- Set lps[0] = 0 and iterate through P to fill the rest of the lps[] values, where lps[i] represents the longest proper prefix which is also a suffix for the substring P[0…i].
- Time complexity: O(m).

Step 2: Start Matching P in S

- Set two pointers, i = 0 for S and j = 0 for P.
- Traverse through the string S. If P[j] == S[i], increment both i and j.
- If there is a mismatch, use lps[] to update j (i.e., set j = lps[j-1]), without resetting i.

Step 3: Record Occurrence

- If j reaches the length of P (i.e., j == m), a full pattern match is found. Record the index i - j as the occurrence.
- Then, reset j = lps[j-1] to continue searching.

Step 4: End the Search

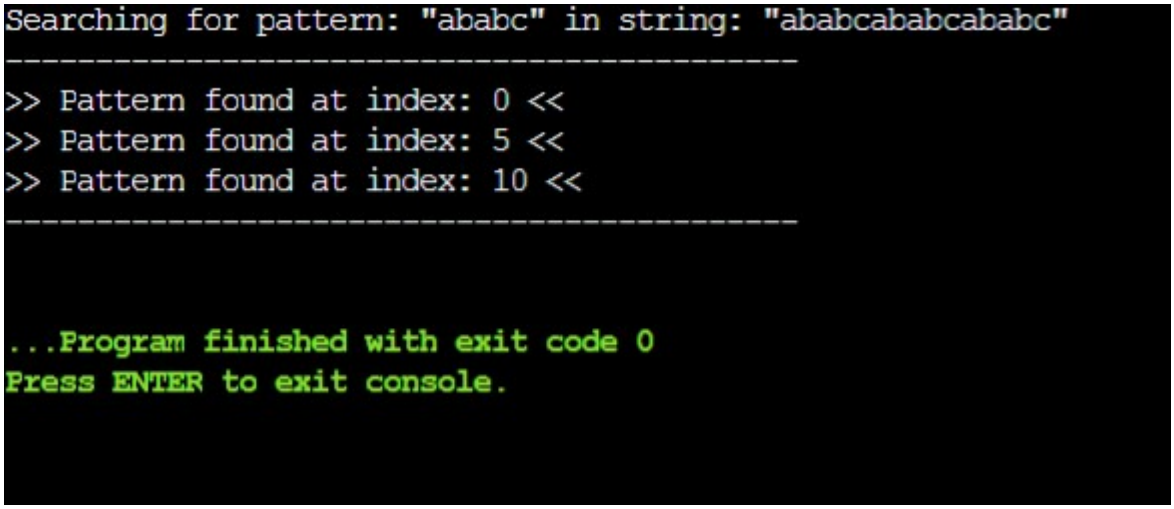
- Continue until i traverses the entire string S.

4. **Implementation/Code:**

```
#include <iostream>
#include <vector>
using namespace std;

// Function to build the LPS (Longest Prefix Suffix) array
vector<int> buildLPS(const string& P) {
    int m = P.length();
    vector<int> lps(m, 0);
    int len = 0;  // length of previous longest prefix suffix
    int i = 1;

    while (i < m) {
        if (P[i] == P[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];  // use previous lps
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}

// KMP search function to find all occurrences of P in S
void KMPsearch(const string& S, const string& P) {
    int n = S.length();
    int m = P.length();
    vector<int> lps = buildLPS(P);

    int i = 0;  // index for S
    int j = 0;  // index for P

    bool found = false;  // flag to check if any pattern is found

    cout << "Searching for pattern: \"" << P << "\" in string: \"" << S << "\"\n";
    cout << "----------------------------------------\n";
```

```
    while (i < n) {
        if (P[j] == S[i]) {
            i++;
            j++;
        }

        if (j == m) {
            found = true;
            cout << ">> Pattern found at index: " << (i - j) << " <<\n";
            j = lps[j - 1];  // get the index from LPS array
        } else if (i < n && P[j] != S[i]) {
            if (j != 0) {
                j = lps[j - 1];  // use the LPS array
            } else {
                i++;
            }
        }
    }

    if (!found) {
        cout << ">> No occurrences of the pattern found. <<\n";
    }

    cout << "-----------------------------------------\n";
}

int main() {
    string S = "ababcababcababc";
    string P = "ababc";
    KMPsearch(S, P);
    return 0;
     }
```

5. **Output:**



6. **Time Complexity:** The overall **time complexity** is: **O(n + m)**
   Space Complexity: The overall **space complexity** is: **O(m)**.

7. **Learning Outcomes:**
   1. **Efficient String Matching:**
      You will understand how the KMP algorithm efficiently finds all occurrences of a pattern P in a string S, avoiding redundant comparisons.
   2. **LPS Array Utility:**
      You will learn how to construct and utilize the Longest Prefix Suffix (LPS) array to skip unnecessary comparisons, optimizing the search process.
   3. **Time and Space Trade-offs:**
      You will gain insight into analyzing algorithms for both time and space complexity, understanding why KMP achieves linear time complexity **O(n + m)** with only **O(m)** space overhead.