Algorithms for various data structures: Arrays, theirstorage and retrieval techniques, Stack, Queue,Operations on stack and queue, related algorithms

Algorithms for various data structures define how operations like insertion, deletion, searching, traversal, and sorting are performed efficiently. Let's explore these algorithms in **detail** for widely-used data structures such as arrays, stacks, queues.

1. What Are Arrays?

An **array** is a data structure used to store a collection of elements (values) of the same type. These elements are stored in **contiguous memory locations** and accessed using their **index**.

Key Characteristics:

- Arrays have a **fixed size** (defined at the time of creation).
- Elements in the array are of the same type (e.g., integers, floats, characters).
- Efficient for accessing elements directly using their index.

1. Storage Techniques for Arrays

Arrays are stored in contiguous blocks of memory. Each element in the array can be accessed directly using its **index**, which is calculated as:

Memory Address=Base Address+(Index×Size of each element)\text{Memory Address} = \text{Base Address} + (\text{Index} \times \text{Size of each element})

Storage Example

For an array arr[5] = {10, 20, 30, 40, 50}, if the base address is 1000 and each element occupies 4 bytes:

- Index 0: Address = 1000 + (0 × 4) = 1000
- Index 1: Address = 1000 + (1 × 4) = 1004
- Index 2: Address = 1000 + (2 × 4) = 1008

This direct calculation makes **array access very efficient**.

2. Retrieval Techniques for Arrays

Retrieval involves accessing elements by their index. For example:

- To access the **second element** in the array arr, use arr[1].

3. Common Algorithms for Arrays

1. Traversal

Visit all elements of the array one by one. **Algorithm:**

```
#include <iostream>
using namespace std;
void traverseArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    traverseArray(arr, size);
    return 0;
}
```

2. Searching

Linear Search

Search for an element by traversing the array sequentially. **Algorithm:**

```
#include <iostream>
using namespace std;
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Found at index i
        }
    }
    return -1; // Not found
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 30;

    int result = linearSearch(arr, size, target);
    if (result != -1)
        cout << "Element found at index " << result << endl;
    else
        cout << "Element not found." << endl;
    return 0;
}
```

Binary Search (for sorted arrays)

Divide the search space in half repeatedly to find the target element. **Algorithm:**

```
#include <iostream>
using namespace std;
int binarySearch(int arr[], int size, int target) {
    int low = 0, high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return mid; // Found
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Not found
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 30;
    int result = binarySearch(arr, size, target);
    if (result != -1)
        cout << "Element found at index " << result << endl;
    else
```

```
        cout << "Element not found." << endl;
    return 0;
}
```

3. Insertion

Add a new element at a specific index. Requires shifting elements to the right. **Algorithm:**

```
#include <iostream>
using namespace std;
void insertElement(int arr[], int &size, int position, int element, int capacity) {
    if (size == capacity) {
        cout << "Array is full!" << endl;
        return;
    }
    for (int i = size - 1; i >= position; i--) {
        arr[i + 1] = arr[i];
    }
    arr[position] = element;
    size++;
}
int main() {
    int arr[10] = {10, 20, 30, 40, 50};
    int size = 5;
    int capacity = 10;
    int position = 2, element = 25;

    insertElement(arr, size, position, element, capacity);
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

4. Deletion

Remove an element at a specific index. Requires shifting elements to the left. **Algorithm:**

```
#include <iostream>
using namespace std;
void deleteElement(int arr[], int &size, int position) {
    if (position >= size || position < 0) {
        cout << "Invalid position!" << endl;
        return;
    }
    for (int i = position; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = 5;
    int position = 2;

    deleteElement(arr, size, position);
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

4. Summary

Advantages of Arrays:

- Fast access to elements using their index.
- Suitable for storing fixed-size data.

Limitations:

- Fixed size (cannot grow dynamically).
- Insertion and deletion require shifting elements.

Common Applications:

- Data storage for simple lists or sequences.
- Used as building blocks for more complex data structures like matrices and stacks.

2. What is a Stack?

A **stack** is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle. This means the element that is added last to the stack is the first one to be removed.

Key Characteristics of a Stack:

- **LIFO Principle**: The last element added to the stack is the first one to be removed.
- **Single Point of Access**: Operations like insertion (push) and removal (pop) occur only at the **top** of the stack.
- Stacks are used in applications such as function calls, undo operations, and expression evaluations.

1. Storage and Retrieval Techniques

Storage:

Stacks can be implemented using:

1. **Arrays**: A stack is implemented as a fixed-size array with a variable (top) that keeps track of the position of the last element.
2. **Linked Lists**: A stack is implemented as a dynamic data structure using nodes, where the top pointer points to the most recently added node.

Retrieval:

The **top** of the stack is used to access or retrieve data. The following operations enable stack retrieval:

1. **Pop**: Removes and returns the top element.
2. **Peek**: Returns the top element without removing it.

2. Common Operations and Related Algorithms

1. Push Operation (Insertion)

Add an element to the top of the stack. **Algorithm**:

```
if top == maxSize - 1:
    print "Stack Overflow"
else:
    top = top + 1
    stack[top] = newElement
```

2. Pop Operation (Deletion)

Remove the top element from the stack. **Algorithm**:

```
if top == -1:
    print "Stack Underflow"
else:
    element = stack[top]
    top = top - 1
    return element
```

3. Peek Operation (Access)

Return the element at the top of the stack without removing it. **Algorithm**:

```
if top == -1:
    print "Stack is Empty"
else:
    return stack[top]
```

4. IsEmpty Operation

Check if the stack is empty. **Algorithm**:

```
if top == -1:
    return true
else:
    return false
```

5. IsFull Operation

Check if the stack is full (in case of an array-based implementation). **Algorithm**:

```
if top == maxSize - 1:
    return true
else:
    return false
```

3. Stack Implementation Using C++

Array-Based Stack

C++ Program for Stack Using Arrays

```
#include <iostream>
using namespace std;
class Stack {
private:
    int arr[5];  // Array to store stack elements
    int top;     // Index of the top element
    int maxSize; // Maximum size of the stack
public:
    Stack() {
        top = -1;          // Stack is initially empty
        maxSize = 5;       // Size of the stack
    }
    // Push an element onto the stack
    void push(int value) {
        if (top == maxSize - 1) {
            cout << "Stack Overflow!" << endl;
        } else {
            top++;
            arr[top] = value;
            cout << value << " pushed onto the stack." << endl;
        }
    }
    // Pop an element from the stack
    void pop() {
        if (top == -1) {
            cout << "Stack Underflow!" << endl;
        } else {
            cout << arr[top] << " popped from the stack." << endl;
            top--;
        }
    }

    // Peek (view the top element)
    void peek() {
        if (top == -1) {
            cout << "Stack is Empty!" << endl;
        } else {
            cout << "Top element is: " << arr[top] << endl;
        }
    }
    // Check if the stack is empty
    bool isEmpty() {
        return (top == -1);
    }
    // Display the stack
    void display() {
        if (top == -1) {
            cout << "Stack is Empty!" << endl;
        } else {
            cout << "Stack elements are: ";
            for (int i = 0; i <= top; i++) {
                cout << arr[i] << " ";
            }
            cout << endl;
        }
    }
};
int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.display();

    stack.pop();
    stack.peek();
    stack.display();
    return 0;
}
```

Linked-List-Based Stack

C++ Program for Stack Using Linked List

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};

class Stack {
private:
    Node* top; // Pointer to the top node
public:
    Stack() {
        top = NULL; // Stack is initially empty
    }

    // Push an element onto the stack
    void push(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = top;
        top = newNode;
        cout << value << " pushed onto the stack." << endl;
    }
    // Pop an element from the stack
    void pop() {
        if (top == NULL) {
            cout << "Stack Underflow!" << endl;
        } else {
            Node* temp = top;
            cout << temp->data << " popped from the stack." << endl;
            top = top->next;
            delete temp;
        }
    }
    // Peek (view the top element)
    void peek() {
        if (top == NULL) {
            cout << "Stack is Empty!" << endl;
        } else {
            cout << "Top element is: " << top->data << endl;
        }
    }
    // Check if the stack is empty
    bool isEmpty() {
        return (top == NULL);
    }
    // Display the stack
    void display() {
        if (top == NULL) {
            cout << "Stack is Empty!" << endl;
        } else {
            cout << "Stack elements are: ";
            Node* temp = top;
            while (temp != NULL) {
                cout << temp->data << " ";
                temp = temp->next;
            }
            cout << endl;
        }
    }
};

int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.display();

    stack.pop();
    stack.peek();
    stack.display();
    return 0;
}
```

4. Summary

Advantages of Stack:

1. Easy to implement.
2. Provides controlled access to data.
3. Useful in algorithms like recursion, backtracking, and expression evaluation.

Limitations:

1. Fixed size (if implemented using arrays).
2. Limited to the LIFO principle (not suitable for random access).

3. What is a Queue?

A **Queue** is a linear data structure that operates on the **First-In-First-Out (FIFO)** principle, meaning the element added **first** is the one to be removed **first**.

1. Key Characteristics of a Queue

- **FIFO**: Elements are added at the **rear** and removed from the **front**.
- **Single Access Points**: Insertion happens at the **rear**, and removal happens at the **front**.
- **Order is Important**: Maintains the order of elements based on how they are added.

Types of Queues

1. **Simple Queue**: Follows the FIFO principle.
2. **Circular Queue**: Rear and front are connected, forming a circular structure.
3. **Priority Queue**: Each element has a priority, and elements are dequeued based on their priority.
4. **Double-Ended Queue (Deque)**: Elements can be added or removed from both ends.

2. Storage Techniques

Queues can be implemented using:

1. **Arrays**:
    - Fixed size.
    - Two pointers:
        - **Front**: Tracks the element to be dequeued.
        - **Rear**: Tracks the position for the next enqueue.
    - Example:
    - [10, 20, 30, 40, 50]
    - ↑   ↑
    - Front   Rear
2. **Linked Lists**:
    - Dynamically sized.
    - Uses nodes where each node points to the next.
    - **Front** points to the first node, and **Rear** points to the last node.

3. Retrieval Techniques

- **Peek/Front**: Access the element at the front of the queue without removing it.

- **Dequeue**: Remove and return the element from the front.
- **Traversal**: Visit all elements from front to rear.

4. Common Operations and Related Algorithms

1. Enqueue (Insertion)

Add an element to the rear of the queue. **Algorithm (Array Implementation)**:

```
if rear == maxSize - 1:
    print "Queue Overflow"
else:
    if front == -1:
        front = 0
    rear = rear + 1
    queue[rear] = newElement
```

Algorithm (Linked List Implementation):

Create a new node with the data.

```
If rear == NULL:
    Set both front and rear to the new node.
Else:
    rear.next = newNode
    rear = newNode
```

2. Dequeue (Deletion)

Remove an element from the front of the queue. **Algorithm (Array Implementation)**:

```
if front == -1 or front > rear:
    print "Queue Underflow"
else:
    element = queue[front]
    front = front + 1
```

Algorithm (Linked List Implementation):

```
if front == NULL:
    print "Queue Underflow"
else:
    temp = front
    front = front.next
    if front == NULL:
        rear = NULL
    delete temp
```

3. Peek (Access the Front Element)

Retrieve the value of the front element without removing it. **Algorithm (Array Implementation)**:

```
if front == -1 or front > rear:
    print "Queue is Empty"
else:
    return queue[front]
```

5. Implementation in C++

Queue Using Arrays

```cpp
#include <iostream>
using namespace std;
class Queue {
private:
    int arr[5];
    int front, rear;
    int maxSize;

public:
    Queue() {
        front = -1;
        rear = -1;
        maxSize = 5;
    }
    void enqueue(int value) {
        if (rear == maxSize - 1) {
            cout << "Queue Overflow!" << endl;
        } else {
            if (front == -1) front = 0; // Initialize front
            rear++;
            arr[rear] = value;
            cout << value << " added to the queue." << endl;
        }
    }
    void dequeue() {
        if (front == -1 || front > rear) {
            cout << "Queue Underflow!" << endl;
        } else {
            cout << arr[front] << " removed from the queue." << endl;
            front++;
        }
    }
    void peek() {
        if (front == -1 || front > rear) {
            cout << "Queue is Empty!" << endl;
        } else {
            cout << "Front element is: " << arr[front] << endl;
        }
    }
    void display() {
        if (front == -1 || front > rear) {
            cout << "Queue is Empty!" << endl;
        } else {
            cout << "Queue elements are: ";
            for (int i = front; i <= rear; i++) {
                cout << arr[i] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    Queue q;

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.peek();
    q.display();
    return 0;
}
```

Queue Using Linked Lists

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
class Queue {
private:
    Node* front;
    Node* rear;

public:
    Queue() {
        front = NULL;
        rear = NULL;
    }
    void enqueue(int value) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->next = NULL;
        if (rear == NULL) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        cout << value << " added to the queue." << endl;
    }
    void dequeue() {
        if (front == NULL) {
            cout << "Queue Underflow!" << endl;
        } else {
            Node* temp = front;
            cout << front->data << " removed from the queue." << endl;
            front = front->next;
            if (front == NULL) rear = NULL;
            delete temp;
        }
    }
    void peek() {
        if (front == NULL) {
```

```
        cout << "Queue is Empty!" << endl;
    } else {
        cout << "Front element is: " << front->data << endl;
    }
}

void display() {
    if (front == NULL) {
        cout << "Queue is Empty!" << endl;
    } else {
        Node* temp = front;
        cout << "Queue elements are: ";
        while (temp != NULL) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
}
};

int main() {
    Queue q;

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();

    q.dequeue();
    q.peek();
    q.display();
    return 0;
}
```

6. Summary

Advantages of Queues:

1. Useful in situations where FIFO order is necessary (e.g., task scheduling).
2. Simple to implement using arrays or linked lists.

Limitations:

1. Fixed size in array-based queues.
2. Requires additional logic to handle empty/full states in circular queues.