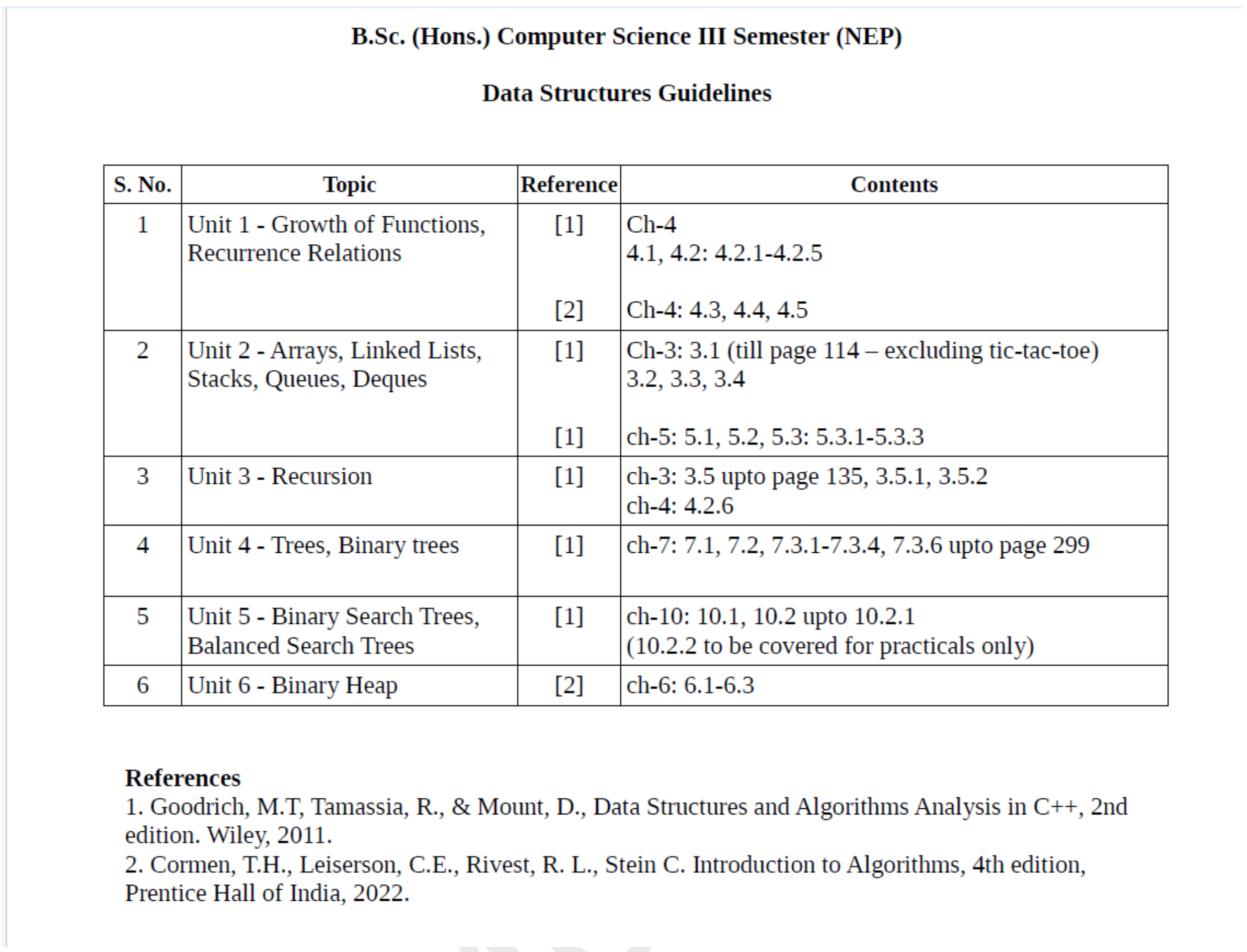


B.Sc. (Hons.) Computer Science III Semester (NEP)

Data Structures Guidelines

S. No.	Topic	Reference	Contents
1	Unit 1 - Growth of Functions, Recurrence Relations	[1] [2]	Ch-4 4.1, 4.2: 4.2.1-4.2.5 Ch-4: 4.3, 4.4, 4.5
2	Unit 2 - Arrays, Linked Lists, Stacks, Queues, Deques	[1] [1]	Ch-3: 3.1 (till page 114 – excluding tic-tac-toe) 3.2, 3.3, 3.4 ch-5: 5.1, 5.2, 5.3: 5.3.1-5.3.3
3	Unit 3 - Recursion	[1]	ch-3: 3.5 upto page 135, 3.5.1, 3.5.2 ch-4: 4.2.6
4	Unit 4 - Trees, Binary trees	[1]	ch-7: 7.1, 7.2, 7.3.1-7.3.4, 7.3.6 upto page 299
5	Unit 5 - Binary Search Trees, Balanced Search Trees	[1]	ch-10: 10.1, 10.2 upto 10.2.1 (10.2.2 to be covered for practicals only)
6	Unit 6 - Binary Heap	[2]	ch-6: 6.1-6.3

References
1. Goodrich, M.T, Tamassia, R., & Mount, D., Data Structures and Algorithms Analysis in C++, 2nd edition. Wiley, 2011.
2. Cormen, T.H., Leiserson, C.E., Rivest, R. L., Stein C. Introduction to Algorithms, 4th edition, Prentice Hall of India, 2022.

Array and its operation

1. Single-Dimensional Arrays
int arr[10];
2. Multi-Dimensional Arrays
int mat[3][4];
3. Dynamic Array using pointers

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

int main() {
    int n;
    cout << "Enter size of array: ";
    cin >> n;

    // dynamically allocate memory
    int* arr = new int[n];

    // input elements
    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    // print elements
    cout << "Array elements: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // resize array manually (create bigger array)
    int newSize = n + 2;
    int* newArr = new int[newSize];
    for (int i = 0; i < n; i++) {
        newArr[i] = arr[i];
    }
    newArr[n] = 99;      // extra values
    newArr[n + 1] = 100;

    cout << "Resized Array: ";
    for (int i = 0; i < newSize; i++) {
        cout << newArr[i] << " ";
    }
    cout << endl;

    // free memory
    delete[] arr;
    delete[] newArr;

    return 0;
}
```

Linear Search in Array, LL
Binary Search in Array

```c
#include <stdio.h>

int binarySearch(int arr[], int n, int x) {
    int low = 0;
    int high = n-1;
    while (low <= high) {
        int mid = low + (high - low) / 2;

        // Check if x is present at mid
        if (arr[mid] == x)
            return mid;

        // If x greater, ignore left half
        if (arr[mid] < x)
            low = mid + 1;

        // If x is smaller, ignore right half
        else
            high = mid - 1;
    }

    // If we reach here, then element was not present
    return -1;
}
```

Array Sorting(Insertion Sort)

```cpp
// C++ program for implementation of Insertion Sort
#include <iostream>
using namespace std;

/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver method
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Two D Array , students, marks of some papers
Days, Hours of Study
Dynamic two D Array

LL and its operations, CRUD LL

Find Length of LL
Find Nth Element

Recursion

Print 1 to n
Print n to 1
Factorial
Reverse of a String
Fabbonaci Series

Palindrome
Reverse
LL for String
Generic LL

Doubly Link List

Structure of a Doubly Linked List

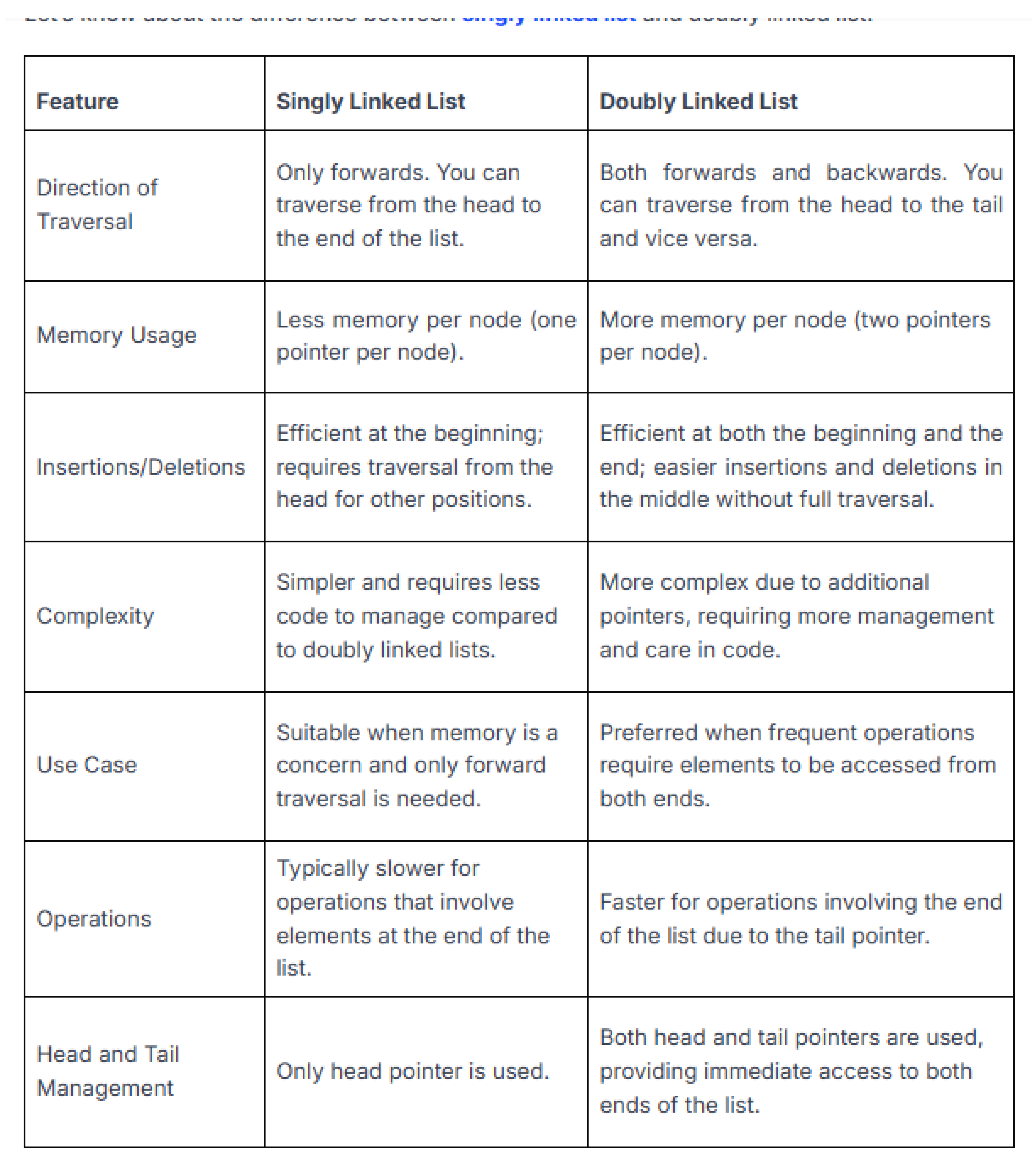
Node:

Each node in a doubly linked list includes:

- Data: The actual information held within the node, which could be numbers, strings, or any other data type.
- Next Pointer: A reference to the next node in the list, which helps in traversing the list forward.
- Prev Pointer: A reference to the previous node in the list, which facilitates backward traversal.

How It Works?

- Head and Tail: The list has two special nodes, the head and the tail. The head points to the first element of the list, while the tail points to the last element. These pointers enhance operations such as adding or removing elements at the ends of the list.
- Traversal: You can start from the head to move forward through the list or from the tail to move backward, making it versatile for various operations that require scanning the list in either direction.
- Insertions and Deletions: Adding or removing nodes can be done efficiently at any point in the list. When inserting or deleting a node, pointers from adjacent nodes are updated to maintain the list’s integrity without needing to shift elements as in contiguous data structures like **array**



Let's know about the difference between singly linked list and doubly linked list.

Feature	Singly Linked List	Doubly Linked List
Direction of Traversal	Only forwards. You can traverse from the head to the end of the list.	Both forwards and backwards. You can traverse from the head to the tail and vice versa.
Memory Usage	Less memory per node (one pointer per node).	More memory per node (two pointers per node).
Insertions/Deletions	Efficient at the beginning; requires traversal from the head for other positions.	Efficient at both the beginning and the end; easier insertions and deletions in the middle without full traversal.
Complexity	Simpler and requires less code to manage compared to doubly linked lists.	More complex due to additional pointers, requiring more management and care in code.
Use Case	Suitable when memory is a concern and only forward traversal is needed.	Preferred when frequent operations require elements to be accessed from both ends.
Operations	Typically slower for operations that involve elements at the end of the list.	Faster for operations involving the end of the list due to the tail pointer.
Head and Tail Management	Only head pointer is used.	Both head and tail pointers are used, providing immediate access to both ends of the list.

DLL Code without using sentinel and keep no. of nodes.

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

// Node class for Doubly Linked List
class Node {
```

```
public:
    int data;
    Node* prev;
    Node* next;

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

// Doubly Linked List class
class DoublyLinkedList {
private:
    Node* head;
    Node* tail;
    int n; // size of list

public:
    // Constructor
    DoublyLinkedList() : head(nullptr), tail(nullptr), n(0) {}

    // Destructor
    ~DoublyLinkedList() {
        while (head != nullptr) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
    }

    // Insert at head
    void insertHead(int val) {
        Node* newNode = new Node(val);
        if (head == nullptr) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        n++;
    }

    // Insert at tail
    void insertTail(int val) {
        Node* newNode = new Node(val);
        if (tail == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        n++;
    }

    // Retrieve element at position (0-based index)
    int retrieve(int pos) {
        if (pos < 0 || pos >= n) {
            cout << "Invalid position\n";
            return -1;
        }
        Node* curr = head;
        for (int i = 0; i < pos; i++) {
            curr = curr->next;
        }
        return curr->data;
    }

    // Update element at position
    void update(int pos, int val) {
        if (pos < 0 || pos >= n) {
            cout << "Invalid position\n";
            return;
        }
        Node* curr = head;
        for (int i = 0; i < pos; i++) {
            curr = curr->next;
        }
        curr->data = val;
    }

    // Delete from head
    void deleteHead() {
        if (head == nullptr) {
            cout << "List is empty\n";
            return;
        }
        Node* temp = head;
        if (head == tail) {
            head = tail = nullptr;
        } else {
            head = head->next;
            head->prev = nullptr;
        }
        delete temp;
        n--;
    }

    // Delete from tail
    void deleteTail() {
        if (tail == nullptr) {
            cout << "List is empty\n";
            return;
        }
        Node* temp = tail;
        if (head == tail) {
            head = tail = nullptr;
        } else {
            tail = tail->prev;
            tail->next = nullptr;
        }
        delete temp;
        n--;
    }

    // Delete from position
    void deleteAt(int pos) {
        if (pos < 0 || pos >= n) {
            cout << "Invalid position\n";
            return;
        }
        if (pos == 0) {
            deleteHead();
            return;
        }
        if (pos == n - 1) {
            deleteTail();
            return;
        }
        Node* curr = head;
        for (int i = 0; i < pos; i++) {
            curr = curr->next;
        }
```

```
        curr->prev->next = curr->next;
        curr->next->prev = curr->prev;
        delete curr;
        n--;
    }

    // Search for a value (returns index, or -1 if not found)
    int search(int val) {
        Node* curr = head;
        int index = 0;
        while (curr != nullptr) {
            if (curr->data == val) return index;
            curr = curr->next;
            index++;
        }
        return -1;
    }

    // Display list
    void display() {
        Node* curr = head;
        cout << "List: ";
        while (curr != nullptr) {
            cout << curr->data << " ";
            curr = curr->next;
        }
        cout << endl;
    }
};

// Main function to test
int main() {
    DoublyLinkedList dll;

    dll.insertHead(10);
    dll.insertHead(20);
    dll.insertTail(30);
    dll.insertTail(40);
    dll.display();

    cout << "Retrieve at index 2: " << dll.retrieve(2) << endl;

    dll.update(1, 25);
    dll.display();
    dll.deleteHead();
    dll.display();

    dll.deleteTail();
    dll.display();

    dll.deleteAt(1);
    dll.display();

    int pos = dll.search(30);
    if (pos != -1)
        cout << "Element 30 found at index " << pos << endl;
    else
        cout << "Element not found\n";

    return 0;
}
```

DLL with Sentinel node.

Here's the code written exactly in the book's style:

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

// Node of a doubly linked list
typedef int Elem;   // element type (can be changed as needed)

class DNode {
private:
    Elem elem;        // element value
    DNode* prev;      // previous node in list
    DNode* next;      // next node in list
    friend class DLinkedList; // allow DLinkedList access
};

class DLinkedList {
public:
    DLinkedList();               // constructor
    ~DLinkedList();              // destructor
    bool empty() const;          // is list empty?
    const Elem& front() const;    // get front element
    const Elem& back() const;     // get back element
    void addFront(const Elem& e); // add to front
    void addBack(const Elem& e);  // add to back
    void removeFront();          // remove from front
    void removeBack();           // remove from back
    void display() const;        // utility to print list

private:  // data members
    DNode* header;   // list sentinels
    DNode* trailer;

protected: // local utilities
    void add(DNode* v, const Elem& e); // insert new node before v
    void remove(DNode* v);            // remove node v
};

// Constructor
DLinkedList::DLinkedList() {
    header = new DNode;
    trailer = new DNode;
    header->next = trailer;
    trailer->prev = header;
}

// Destructor
DLinkedList::~DLinkedList() {
    while (!empty()) removeFront();
    delete header;
    delete trailer;
}

// Check empty
bool DLinkedList::empty() const {
    return (header->next == trailer);
}

// Return front element
const Elem& DLinkedList::front() const {
    return header->next->elem;
}

// Return back element
```

```
const Elem& DLinkedList::back() const {
    return trailer->prev->elem;
}

// Add before a given node
void DLinkedList::add(DNode* v, const Elem& e) {
    DNode* u = new DNode;
    u->elem = e;
    u->next = v;
    u->prev = v->prev;
    v->prev->next = u;
    v->prev = u;
}

// Remove a given node
void DLinkedList::remove(DNode* v) {
    DNode* u = v->prev;
    DNode* w = v->next;
    u->next = w;
    w->prev = u;
    delete v;
}

// Insert at front
void DLinkedList::addFront(const Elem& e) {
    add(header->next, e);
}

// Insert at back
void DLinkedList::addBack(const Elem& e) {
    add(trailer, e);
}

// Remove front node
void DLinkedList::removeFront() {
    if (!empty())
        remove(header->next);
}

// Remove back node
void DLinkedList::removeBack() {
    if (!empty())
        remove(trailer->prev);
}
```

```cpp
// Utility: display list
void DLinkedList::display() const {
    DNode* curr = header->next;
    cout << "List: ";
    while (curr != trailer) {
        cout << curr->elem << " ";
        curr = curr->next;
    }
    cout << endl;
}

// Main test
int main() {
    DLinkedList dll;

    dll.addFront(10);
    dll.addFront(20);
    dll.addBack(30);
    dll.addBack(40);

    dll.display();

    cout << "Front: " << dll.front() << endl;
    cout << "Back: " << dll.back() << endl;

    dll.removeFront();
    dll.display();

    dll.removeBack();
    dll.display();

    return 0;
}
```

Output:

```
List: 20 10 30 40
Front: 20
Back: 40
List: 10 30 40
List: 10 30
```

Circular LL

https://www.wscubetech.com/resources/dsa/circular-linked-list-data-structure

What is a Circular Linked List?

A circular linked list is a type of data structure where each node points to the next node, and the last node points back to the first, forming a circle.
This setup allows for an endless cycle of node traversal, which can be particularly useful in applications where the end of the list naturally reconnects to the beginning.
For example, in computer applications that manage resources in a loop or in creating playlists where the last song leads back to the first song, a circular linked list offers a seamless way to cycle through items without the need for complex logic to manage the end of the list.

```
#include <iostream>

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

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() : head(nullptr) {}

    void append(int data) {
        Node* newNode = new Node{data, nullptr};
        if (!head) {
            head = newNode;
            head->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }
    }

    void printList() {
        if (head) {
            Node* temp = head;
            do {
```

```
            std::cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        std::cout << std::endl;
    }
  }
};

int main() {
    CircularLinkedList cll;
    cll.append(1);
    cll.append(2);
    cll.append(3);
    cll.printList();  // Output: 1 2 3
    return 0;
}
```

Stack

Applications of Stacks:

- **Function calls:** Stacks are used to keep track of the return addresses of function calls, allowing the program to return to the correct location after a function has finished executing.
- **Recursion:** Stacks are used to store the local variables and return addresses of recursive function calls, allowing the program to keep track of the current state of the recursion.
- **Expression evaluation:** Stacks are used to evaluate expressions in postfix notation (Reverse Polish Notation).
- **Syntax parsing:** Stacks are used to check the validity of syntax in programming languages and other formal languages.
- **Memory management:** Stacks are used to allocate and manage memory in some operating systems and programming languages.

Advantages of Stacks:

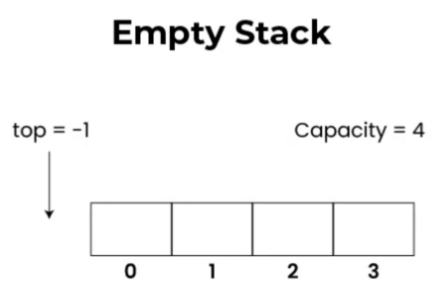
- **Simplicity:** Stacks are a simple and easy-to-understand data structure, making them suitable for a wide range of applications.

- **Efficiency:** Push and pop operations on a stack can be performed in constant time **(O(1))**, providing efficient access to data.
- **Last-in, First-out (LIFO):** Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed. This behavior is useful in many scenarios, such as function calls and expression evaluation.
- **Limited memory usage:** Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

Disadvantages of Stacks:

- **Limited access:** Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
- **Potential for overflow:** If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
- **Not suitable for random access:** Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
- **Limited capacity:** Stacks have a fixed capacity, which can be a limitation if the number of elements that need to be stored is unknown or highly variable.

Stack using Array



```
// C++ program to create a stack with
// given capacity
```

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

class Stack {
    int top, cap;
    int *a;

public:
    Stack(int cap) {
        this->cap = cap;
        top = -1;
        a = new int[cap];
    }

    ~Stack() {
        delete[] a;
    }

    bool push(int x) {
        if (top >= cap - 1) {
            cout << "Stack Overflow\n";
            return false;
        }
        a[++top] = x;
        return true;
    }

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

    int peek() {
        if (top < 0) {
            cout << "Stack is Empty\n";
            return 0;
        }
        return a[top];
    }

    bool isEmpty() {
        return top < 0;
    }
};

int main() {
    Stack s(5);
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " popped from stack\n";

    cout << "Top element is: " << s.peek() << endl;

    cout << "Elements present in stack: ";
    while (!s.isEmpty()) {
        cout << s.peek() << " ";
        s.pop();
    }

    return 0;
}
```

Stack - Linked List Implementation

#TODO

Infix Expressions

Infix expressions are mathematical expressions where the operator is placed between its operands. This is the most common mathematical notation used by humans. For example, the expression "2 + 3" is an infix expression

Parentheses are used in infix notation to specify the order in which operations should be performed. For example, in the expression "(2 + 3) * 4"

Evaluating Infix Expressions

Evaluating infix expressions requires additional processing to handle the order of operations and parentheses. First convert the **infix expression** to **postfix notation**. This can be done using a stack or a recursive algorithm. Then evaluate the postfix expression.

Advantages of Infix Expressions

- More natural and easier to read and understand for humans.

- Widely used and supported by most programming languages and calculators.

Disadvantages Infix Expressions

- Requires parentheses to specify the order of operations.
- Can be difficult to parse and evaluate efficiently.

Prefix Expressions (Polish Notation)

Prefix expressions are also known as **Polish notation**, are a mathematical notation where the operator precedes its operands.

For example, the infix expression "a + b" would be written as "+ a b" in prefix notation.

Advantages of Prefix Expressions

- No need for parentheses, as the operator always precedes its operands.
- Easier to parse and evaluate using a stack-based algorithm.
- Can be more efficient in certain situations, such as when dealing with expressions that have a large number of nested parentheses.

Disadvantages of Prefix Expressions

- Can be difficult to read and understand for humans.
- Not as commonly used as infix notation.

Postfix Expressions (Reverse Polish Notation)

Postfix expressions are also known as **Reverse Polish Notation (RPN)**, are a mathematical notation where the **operator follows its operands**

For example, the infix expression "5 + 2" would be written as "5 2 +" in postfix notation.

Advantages of Postfix Notation

- Also eliminates the need for parentheses.
- Easier to read and understand for humans.
- More commonly used than prefix notation.

Disadvantages of Postfix Expressions

- Requires a stack-based algorithm for evaluation.
- Can be less efficient than prefix notation in certain situations.

spect	Infix Notation	Prefix Notation (Polish Notation)	Postfix Notation (Reverse Polish Notation)
Readability	Human-readable	Less human-readable, requires familiarity	Less human-readable, requires familiarity
Operator Placement	Between operands	Before operands	After operands
Parentheses Requirement	Often required	Not required	Not required
Operator Precedence Tracking	Required, parentheses determine precedence	Not required, operators have fixed precedence	Not required, operators have fixed precedence
Evaluation Method	Left-to-right	Right-to-left	Left-to-right
Ambiguity Handling	May require explicit use of parentheses	Ambiguity rare, straightforward evaluation	Ambiguity rare, straightforward evaluation
Unary Operator Handling	Requires careful placement	Simplified handling due to explicit notation	Simplified handling due to explicit notation
Computer Efficiency	Less efficient due to precedence tracking and parentheses handling	More efficient due to fixed precedence and absence of parentheses	More efficient due to fixed precedence and absence of parentheses
Usage	Common in everyday arithmetic and mathematical notation	Common in computer science and programming languages	Common in computer science and programming languages

Infix → Postfix conversion (using **Stacks**).

This is a classic problem solved using the **Shunting Yard Algorithm** (Edsger Dijkstra).

Rules Recap:

1. Operands (A, B, C, ... or numbers) go **directly** to output.
2. Operators (+, -, *, /, ^) are pushed to a **stack**, respecting precedence.
3. If operator precedence is lower or equal than the operator on the stack, pop from stack to output.
4. Left parenthesis (→ push to stack.
5. Right parenthesis) → pop until a (is found.
6. At end → pop remaining operators.

A+B*C
ABC*+

(A+B)*C
AB+C*

A+B*C-D/E
ABC*+DE/-

(A+B)*(C+D)
AB+CD+*

(A+B*C)/(D-E)
ABC*+DE-/

A^B^C
ABC^^

Input: A+(B*C-(D/E^F)*G)*H
Output: ABC*DEF^/G*-H*+

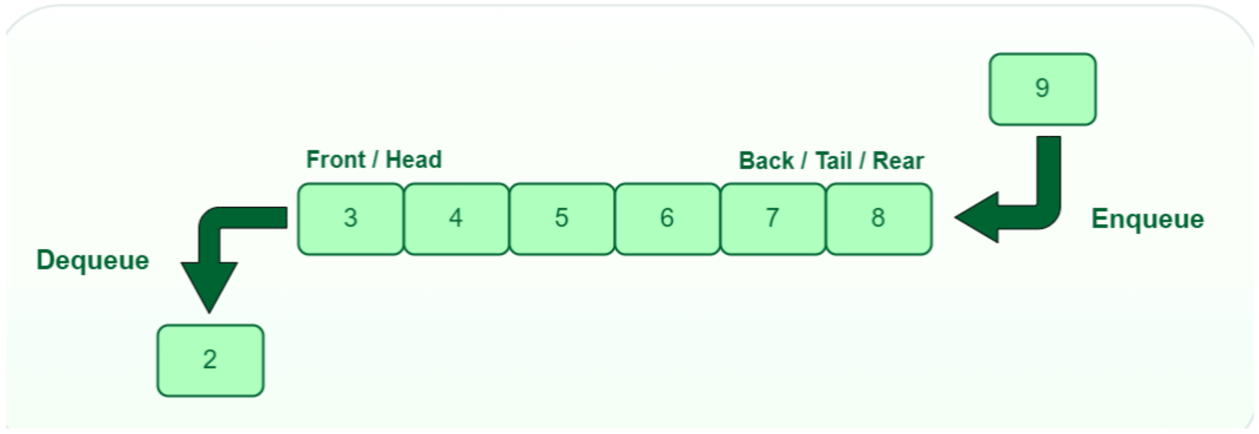
Input: s = "A*(B+C)/D"
Output: ABC+*D/

Queue

A **Queue Data Structure** is a fundamental concept in computer science used for storing and managing data in a specific order.

- It follows the principle of "**First in, First out**" **(FIFO)**, where the first element added to the queue is the first one to be removed.
- It is used as a buffer in computer systems where we have speed mismatch between two devices that communicate with each other. For example, CPU and keyboard and two devices in a network
- Queue is also used in Operating System algorithms like CPU Scheduling and Memory Management, and many standard algorithms like Breadth First Search of Graph, Level Order Traversal of a Tree.

When an element is inserted in a queue, then the operation is known as **Enqueue** and when an element is deleted from the queue, then the operation is known as **Dequeue.**



Primary Queue Operations:

- void enqueue(int Element): When this operation is performed, an element is inserted in the queue at the end i.e. at the rear end. (Where T is Generic i.e we can define Queue of any type of data structure.) This operation takes constant time i.e O(1).
- int dequeue(): When this operation is performed, an element is removed from the front end and is returned. This operation takes constant time i.e O(1).

Auxiliary Queue Operations:

- int front(): This operation will return the element at the front without removing it and it takes O(1) time.
- int rear(): This operation will return the element at the rear without removing it, Its Time Complexity is O(1).
- int isEmpty(): This operation indicates whether the queue is empty or not. This Operation is also done in O(1).

- int size(): This operation will return the size of the queue i.e. the total number of elements present in the queue and it's time complexity is O(1).

Types of Queues:

- Simple Queue: Simple queue also known as a linear queue is the most basic version of a queue. Here, insertion of an element i.e. the Enqueue operation takes place at the rear end and removal of an element i.e. the Dequeue operation takes place at the front end.
- Circular Queue: This is mainly an efficient array implementation of Simple Queue. In a circular queue, the element of the queue act as a circular ring. The working of a circular queue is similar to the linear queue except for the fact that the last element is connected to the first element. Its advantage is that the memory is utilized in a better way. This is because if there is an empty space i.e. if no element is present at a certain position in the queue, then an element can be easily added at that position.
- Priority Queue: This queue is a special type of queue. Its specialty is that it arranges the elements in a queue based on some priority. The priority can be something where the element with the highest value has the priority so it creates a queue with decreasing order of values. The priority can also be such that the

element with the lowest value gets the highest priority so in turn it creates a queue with increasing order of values.

- Dequeue: Dequeue is also known as Double Ended Queue. As the name suggests double ended, it means that an element can be inserted or removed from both the ends of the queue unlike the other queues in which it can be done only from one end. Because of this property it may not obey the First In First Out property.

Implementation of Queue:

- Sequential allocation: A queue can be implemented using an array. It can organize a limited number of elements.
- Linked list allocation: A queue can be implemented using a linked list. It can organize an unlimited number of elements.

Applications of Queue:

- Multi programming: Multi programming means when multiple programs are running in the main memory. It is essential to organize these multiple programs and these multiple programs are organized as queues.
- Network: In a network, a queue is used in devices such as a router or a switch. Another application of a queue is a mail queue which is a directory that stores data and controls files for mail messages.

- Job Scheduling: The computer has a task to execute a particular number of jobs that are scheduled to be executed one after another. These jobs are assigned to the processor one by one which is organized using a queue.
- Shared resources: Queues are used as waiting lists for a single shared resource.

<u>Real-time application of Queue:</u>

- Working as a buffer between a slow and a fast device. For example keyboard and CPU, and two devices on the network.
- ATM Booth Line
- Ticket Counter Line
- CPU task scheduling
- Waiting time of each customer at call centers.

<u>Advantages of Queue:</u>

- A large amount of data can be managed efficiently with ease.
- Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
- Queues are useful when a particular service is used by multiple consumers.
- Queues are fast in speed for data inter-process communication.

- Queues can be used in the implementation of other data structures.

<u>Disadvantages of Queue:</u>

- The operations such as insertion and deletion of elements from the middle are time consuming.
- In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
- Searching for an element takes O(N) time.
- Maximum size of a queue must be defined prior in case of array implementation.

1. Queue using Array

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

#define MAX 5

class Queue {
private:
    int arr[MAX];
    int front, rear;

public:
    Queue() {
        front = -1;
        rear = -1;
    }

    bool isFull() {
        return (rear == MAX - 1);
    }

    bool isEmpty() {
```

```cpp
        return (front == -1 || front > rear);
    }

    void enqueue(int val) {
        if (isFull()) {
            cout << "Queue Overflow!\n";
            return;
        }
        if (front == -1) front = 0;
        arr[++rear] = val;
        cout << val << " enqueued.\n";
    }

    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow!\n";
            return;
        }
        cout << arr[front] << " dequeued.\n";
        front++;
    }

    void peek() {
        if (isEmpty()) {
            cout << "Queue is empty!\n";
            return;
        }
        cout << "Front element: " << arr[front] << endl;
    }

    void display() {
        if (isEmpty()) {
            cout << "Queue is empty!\n";
            return;
        }
        cout << "Queue: ";
        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.display();
    q.peek();
    return 0;
}
```

2. Circular Queue (Better Array Implementation)

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

#define MAX 5

class CircularQueue {
private:
    int arr[MAX];
    int front, rear, count;

public:
    CircularQueue() {
        front = rear = 0;
        count = 0;
    }

    bool isFull() {
        return (count == MAX);
    }

    bool isEmpty() {
        return (count == 0);
    }

    void enqueue(int val) {
        if (isFull()) {
            cout << "Queue Overflow!\n";
            return;
        }
        arr[rear] = val;
        rear = (rear + 1) % MAX;
        count++;
        cout << val << " enqueued.\n";
    }

    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow!\n";
            return;
        }
        cout << arr[front] << " dequeued.\n";
        front = (front + 1) % MAX;
        count--;
    }

    void peek() {
        if (isEmpty()) {
            cout << "Queue is empty!\n";
            return;
        }
        cout << "Front element: " << arr[front] << endl;
    }

    void display() {
        if (isEmpty()) {
            cout << "Queue is empty!\n";
            return;
        }
        cout << "Queue: ";
        for (int i = 0; i < count; i++) {
            cout << arr[(front + i) % MAX] << " ";
        }
        cout << endl;
    }
};

int main() {
    CircularQueue cq;
    cq.enqueue(10);
    cq.enqueue(20);
    cq.enqueue(30);
    cq.enqueue(40);
    cq.enqueue(50);
    cq.display();
    cq.dequeue();
    cq.dequeue();
    cq.display();
    cq.enqueue(60);
    cq.display();
    cq.peek();
    return 0;
}
```

3. Queue using Linked List

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

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

class Queue {
private:
    Node* front;
    Node* rear;

public:
    Queue() : front(nullptr), rear(nullptr) {}

    bool isEmpty() {
        return (front == nullptr);
    }

    void enqueue(int val) {
        Node* newNode = new Node(val);
        if (rear == nullptr) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            rear = newNode;
        }
        cout << val << " enqueued.\n";
    }

    void dequeue() {
        if (isEmpty()) {
            cout << "Queue Underflow!\n";
            return;
        }
        Node* temp = front;
        cout << front->data << " dequeued.\n";
        front = front->next;
        if (front == nullptr) rear = nullptr;
        delete temp;
    }

    void peek() {
        if (isEmpty()) {
            cout << "Queue is empty!\n";
            return;
        }
        cout << "Front element: " << front->data << endl;
    }

    void display() {
        if (isEmpty()) {
            cout << "Queue is empty!\n";
            return;
        }
        cout << "Queue: ";
        Node* curr = front;
        while (curr != nullptr) {
            cout << curr->data << " ";
            curr = curr->next;
        }
        cout << endl;
    }
};

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();
    q.dequeue();
    q.display();
    q.peek();
    return 0;
}
```

Prefix → Postfix Conversion

Algorithm (using stack)

1. Read the prefix expression from right to left.
2. If the symbol is an operand, push it onto the stack.
3. If the symbol is an operator:
    - Pop two operands from the stack.
    - Concatenate them in postfix order: `operand1 operand2 operator`.

    ○ Push this new string back onto the stack.

4. At the end, the stack will have the postfix expression.

Example
Prefix: *+AB-CD
Postfix: AB+CD-*

Prefix: -+A*BCD
Postfix = ABC*+D-

Prefix: / *AB +CD
Postfix = AB*CD+/

Prefix: *-A/BC-/AKL
Postfix = ABC/-AK/L-*

Prefix: +*AB^CD
Postfix = AB*CD^+

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>
using namespace std;

string prefixToPostfix(string prefix) {
    stack<string> st;

    // scan from right to left
    for (int i = prefix.size() - 1; i >= 0; i--) {
        char symbol = prefix[i];

        if (isalnum(symbol)) {
            // operand → push as string
            st.push(string(1, symbol));
        } else {
            // operator → pop two operands
            string op1 = st.top(); st.pop();
            string op2 = st.top(); st.pop();

            // form postfix expression: op1 op2 operator
            string expr = op1 + op2 + symbol;
            st.push(expr);
        }
    }

    // final postfix expression
    return st.top();
}

int main() {
    string prefix = "*+AB-CD";
    cout << "Prefix: " << prefix << endl;
    cout << "Postfix: " << prefixToPostfix(prefix) << endl;
    return 0;
}
```