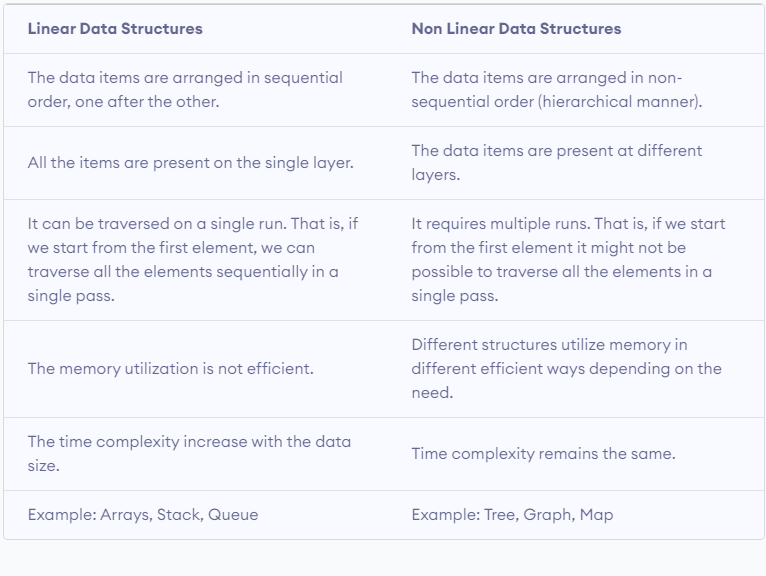
Data Structure & Algorithm

Data Structure: Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently. A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways.

Algorithm: An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations. According to its formal definition, an algorithm is a finite set of instructions carried out in a specific order to perform a particular task. In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output.

Types of Data Structure: The two basic data structure types are linear data structures, non linear data structure. Linear:1.Array, 2.Stack, 3.Queue, 4.Linked List. Non-Linear: 1.Graph, 2.Tree.



Linear data structures

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement.

Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

There are mainly three asymptotic notations:

Big-O notation

Omega notation

Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

O(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Ω(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }

Master Theorem

The master method is a formula for solving recurrence relations of the form:

T(n) = aT(n/b) + f(n),

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed

to have the same size.

f(n) = cost of the work done outside the recursive call,

which includes the cost of dividing the problem and

cost of merging the solutions

Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.

\*\*\*If a ≥ 1 and b > 1 are constants and f(n) is an asymptotically positive function, then the time complexity of a recursive relation is given by

T(n) = aT(n/b) + f(n)

where, T(n) has the following asymptotic bounds:

1. If f(n) = O(nlogb a-ϵ), then T(n) = Θ(nlogb a).

2. If f(n) = Θ(nlogb a), then T(n) = Θ(nlogb a \* log n).

3. If f(n) = Ω(nlogb a+ϵ), then T(n) = Θ(f(n)).

ϵ > 0 is a constant.

Stack Data Structure

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. In simpler terms, the last element added to the stack is the first one to be removed. Think of it as a stack of plates - you can only add or remove plates from the top.

Basic Operations of Stack

There are some basic operations that allow us to perform different actions on a stack.

Push: Add an element to the top of a stack

Pop: Remove an element from the top of a stack

IsEmpty: Check if the stack is empty

IsFull: Check if the stack is full

Peek: Get the value of the top element without removing it

Working of Stack Data Structure

The operations work as follows:

A pointer called TOP is used to keep track of the top element in the stack.

When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.

On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.

On popping an element, we return the element pointed to by TOP and reduce its value.

Before pushing, we check if the stack is already full

Before popping, we check if the stack is already empty



// Stack implementation in C++

#include <stdlib.h>

#include <iostream>

using namespace std;

#define MAX 10

int size = 0;

// Creating a stack

struct stack {

int items[MAX];

int top;

};

typedef struct stack st;

void createEmptyStack(st \*s) {

s->top = -1;

}

// Check if the stack is full

int isfull(st \*s) {

if (s->top == MAX - 1)

return 1;

else

return 0;

}

// Check if the stack is empty

int isempty(st \*s) {

if (s->top == -1)

return 1;

else

return 0;

}

// Add elements into stack

void push(st \*s, int newitem) {

if (isfull(s)) {

cout << "STACK FULL";

} else {

s->top++;

s->items[s->top] = newitem;

}

size++;

}

// Remove element from stack

void pop(st \*s) {

if (isempty(s)) {

cout << "\n STACK EMPTY \n";

} else {

cout << "Item popped= " << s->items[s->top];

s->top--;

}

size--;

cout << endl;

}

// Print elements of stack

void printStack(st \*s) {

printf("Stack: ");

for (int i = 0; i < size; i++) {

cout << s->items[i] << " ";

}

cout << endl;

}

// Driver code

int main() {

int ch;

st \*s = (st \*)malloc(sizeof(st));

createEmptyStack(s);

push(s, 1);

push(s, 2);

push(s, 3);

push(s, 4);

printStack(s);

pop(s);

cout << "\nAfter popping out\n";

printStack(s);

}

Applications of Stack Data Structure

Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

To reverse a word - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.

In compilers - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 \* (7 - 9) by converting the expression to prefix or postfix form.

In browsers - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

Queue Data Structure

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. In simpler terms, the first element added to the queue is the first one to be removed. Think of it as a line of people waiting for a service - the person who joins the line first gets serviced first.

Operations on a Queue:

Enqueue: Add an element to the back of the queue.

Dequeue: Remove the element from the front of the queue.

Front: Retrieve the front element of the queue without removing it.

isEmpty: Check if the queue is empty.

Size: Get the number of elements in the queue.

Implementing a Queue in C++:

#include <iostream>

// Define maximum size of the queue

#define MAX\_SIZE 100

// Define queue class

class Queue {

private:

int arr[MAX\_SIZE];

int front, rear;

public:

// Constructor

Queue() {

front = -1;

rear = -1;

}

// Enqueue operation

void enqueue(int val) {

if (rear >= MAX\_SIZE - 1) {

std::cout << "Queue Overflow! Cannot enqueue element " << val << std::endl;

return;

}

if (isEmpty()) {

front = 0;

}

arr[++rear] = val;

std::cout << "Enqueued " << val << " into the queue." << std::endl;

}

// Dequeue operation

void dequeue() {

if (isEmpty()) {

std::cout << "Queue Underflow! Cannot dequeue element." << std::endl;

return;

}

int val = arr[front++];

std::cout << "Dequeued " << val << " from the queue." << std::endl;

if (front > rear) {

front = -1;

rear = -1;

}

}

// Peek operation

int peek() {

if (isEmpty()) {

std::cout << "Queue is empty. Peek operation cannot be performed." << std::endl;

return -1; // Return a default value indicating error

}

return arr[front];

}

// Check if queue is empty

bool isEmpty() {

return (front == -1 && rear == -1);

}

// Check if queue is full

bool isFull() {

return (rear >= MAX\_SIZE - 1);

}

};

int main() {

// Create a queue object

Queue myQueue;

// Enqueue some elements into the queue

myQueue.enqueue(5);

myQueue.enqueue(10);

myQueue.enqueue(15);

// Peek at the front element

std::cout << "Front element: " << myQueue.peek() << std::endl;

// Dequeue some elements from the queue

myQueue.dequeue();

myQueue.dequeue();

// Check if the queue is empty

if (myQueue.isEmpty()) {

std::cout << "Queue is empty." << std::endl;

} else {

std::cout << "Queue is not empty." << std::endl;

}

return 0;

}

There are four different types of queues:

Simple Queue

Circular Queue

Priority Queue

Double Ended Queue