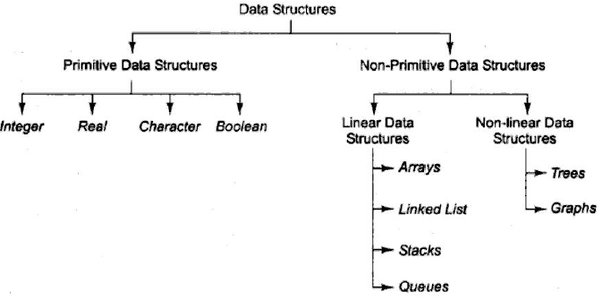
**Short Question 2 marks**

1. *Define Linear and Non-Linear Data Structure*

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. The types of linear data structures are Array, Queue, Stack, Linked List.

A non-linear data structure is also another type of data structure in which the data elements are not arranged in a contiguous manner. **Trees** and **Graphs** are the types of non-linear data structure.

1. Mention different types of Data Structure.



1. Write a C program to create a node in singly Linked List

#include <stdio.h>

#include <stdlib.h>

*// Define the structure for a node*

struct Node {

int data*; // Data of the node*

struct Node\* next; *// Pointer to the next node*

};

*// Function to create a new node*

struct Node\* createNode(int value) {

*// Allocate memory for a new node*

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

*// Check if memory allocation was successful*

if (newNode == NULL) {

printf("Memory allocation failed\n");

exit(1); *// Terminate the program*

}

*// Assign data to the new node*

newNode->data = value;

newNode->next = NULL; *// The new node is initially not linked to any other node*

return newNode;

}

int main() {

*// Create a new node with data value 42*

struct Node\* newNode = createNode(42);

*// Display the data of the new node*

printf("Data of the new node: %d\n", newNode->data);

*// Don't forget to free the allocated memory when done*

free(newNode);

return 0;

}

1. Explain LIFO and FIFO in Data Structure using an example

LIFO (Last In First Out) is a principle used in data structures where the last element added is the first one to be removed.

Example: Consider a stack with elements 3, 7, and 2. If you push 4 onto the stack, it becomes [4, 3, 7, 2]. When you pop, the top element (4) is removed.

FIFO (First In First Out) is a principle where the first element added is the first one to be removed. It is similar to a queue, where people wait in a line, and the person who arrived first is the first to be served.

Example: A queue with elements 5, 3, and 8. If you enqueue 2, it becomes [5, 3, 8, 2]. When you dequeue, the first element (5) is removed.

1. Explain the advantages of Double linked list over single linked List.

the advantages of doubly linked list over singly linked list:

* **Bidirectional traversal:** Doubly linked lists allow traversal in both forward and reverse directions due to the presence of two pointers (previous and next) in each node.
* **Efficient node deletion:** Deleting a node from a doubly linked list is more efficient than in a singly linked list. Since each node has a pointer to its previous node, removing a node does not require traversing the entire list to find the previous node, as it does in singly linked lists.
* **Ease of insertion/deletion at both ends:** Doubly linked lists enable quick insertion and deletion at both the beginning and end of the list.
* **Enhanced flexibility:** The presence of two pointers in each node provides greater flexibility in manipulating the list.
* **Backtracking in algorithms:** In certain algorithms or data structures, backtracking is essential. Doubly linked lists allow efficient backtracking as they provide easy access to the previous nodes.

1. What is the maximum number of nodes in a binary tree of height k?

The maximum number of nodes in a binary tree of height *k* can be calculated using the formula:

Maximum Nodes=2*k*+1−1

This formula is derived from the fact that at each level *i* of a binary tree, there can be a maximum of 2*i* nodes. The height *k* represents the number of levels in the tree.

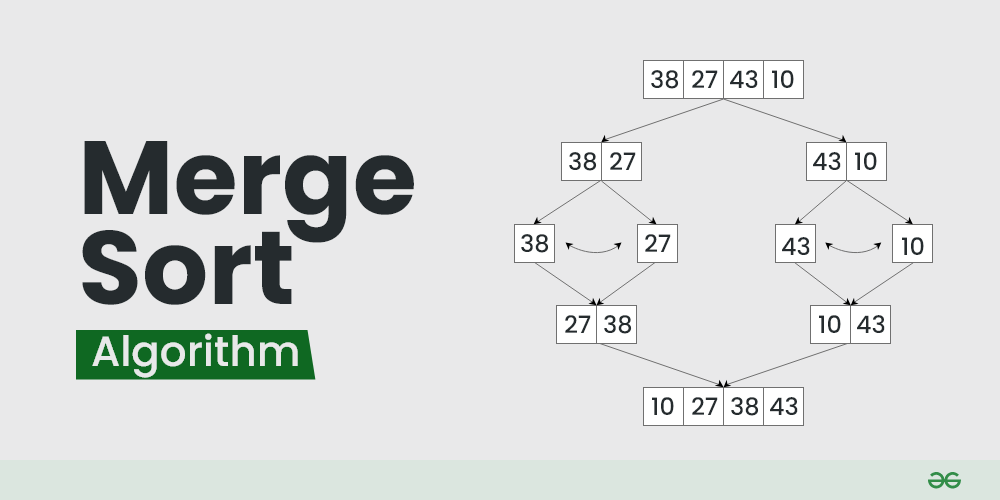
1. What is Divide and Conquer method? Explain the mechanism of Merge Sort.

**Divide and Conquer Method :** Divide and Conquer is a problem-solving strategy where a problem is broken into smaller subproblems that are easier to solve. The solutions to the subproblems are then combined to solve the original problem. It involves three steps: divide the problem into smaller subproblems, conquer the subproblems by solving them recursively, and combine the solutions of the subproblems to solve the original problem.

**Mechanism of Merge Sort :** Merge Sort is a sorting algorithm that employs the Divide and Conquer method. It works as follows:

1. **Divide:** The unsorted list is divided into *n* sublists, each containing one element.
2. **Conquer:** Adjacent sublists are recursively sorted. This is done by repeatedly dividing each sublist into halves until each sublist contains only one element.
3. **Combine:** The sorted sublists are merged to produce new sorted sublists until a single sorted list is obtained.

Let us consider an array arr={38,27,43,10}



1. What happens to stack when top=maxsize-1?

When the top of the stack reaches the maximum index value (i.e., **top = maxsize - 1**), the stack is considered full, and attempting to push more elements onto the stack will result in a stack overflow.

1. What operations are used in stack and queue to insert and delete elements respectively?

In stack push and pop is used to insert and delete elements resp.

In queue enqueue and dequeue is used to insert and delete elements resp.

1. Write the postfix expression for (A+B) \*(C-D).

the postfix expression is: *AB*+*CD*−∗

1. Define Data Structure and mention uses of Data Structure

A data structure is a way of organizing and storing data to perform operations efficiently. It defines the relationships and the rules for manipulating the data, ensuring optimal access, storage, and retrieval.

Uses of data Structure:

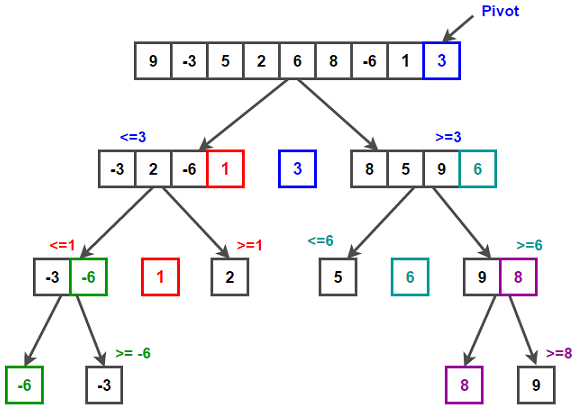
* **Efficient Data Retrieval:** Data structures facilitate quick access and retrieval of information, enabling efficient search operations.
* **Optimized Memory Utilization:** Well-designed data structures minimize space wastage, ensuring optimal memory utilization and improved program performance.
* **Algorithm Design:** Data structures form the foundation for designing and implementing efficient algorithms, essential for tasks like sorting, searching, and graph traversal.
* **Compiler Design:** In compiler construction, data structures such as symbol tables and parsing trees are utilized for efficient language translation and compilation.
* **File Systems:** Data structures, including indexes and file allocation tables, are employed in file systems to manage and access data stored on disk drives efficiently.
* **Networking:** Data structures are used in network representations, routing tables, and protocols, contributing to efficient data transmission and routing.

1. What is Divide and Conquer method? Explain the mechanism of Quick Sort.

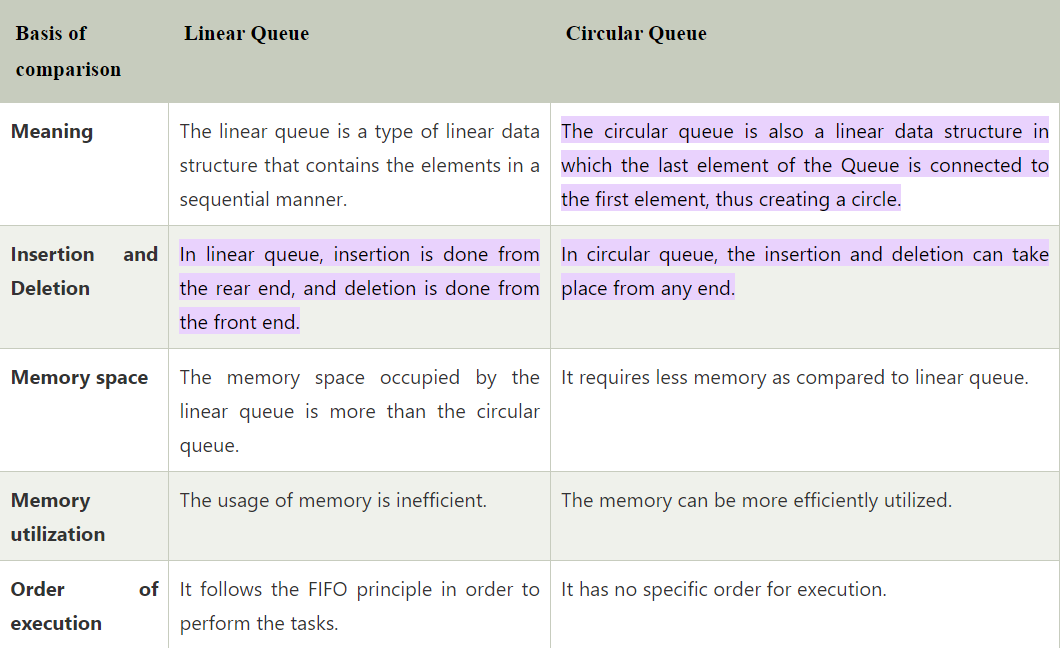
Mechanism of Quick Sort:

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array



1. Write the difference between linear queue and circular queue.



1. Give the underflow and overflow condition in stack.

Underflow happens when we try to pop an item from an empty stack. Overflow happens when we try to push more items on a stack than it can hold.

**5 marks questions**

* 1. What are the disadvantages of Linear Queue? Explain with an example.

Disadvantages of Queue:

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

Example: Consider a linear queue represented as an array: [5, 8, 3, 2, 7].

If three elements are dequeued, the queue becomes: [\_, \_, \_, 2, 7].

The space at the front is wasted and remains unused, leading to memory inefficiency

* 1. Explain the difference between complete Binary tree and Full binary tree with example

|  |  |
| --- | --- |
| **Binary Tree** | **Full Binary Tree** |
| A complete binary tree is a binary tree where all levels are completely filled, except possibly for the last level, which is filled from left to right. | A full binary tree is a binary tree in which every node has either 0 or 2 children. Every node is either a leaf or has two children. |
| binary trees are filled from left to right on all levels, ensuring nodes are as left as possible in the last level. | Full binary trees do not concern themselves with the left-to-right filling of levels; they only focus on whether a node has 0 or 2 children. |
| Example: 1  / \  2 3  / \ /  4 5 6 | Example: 1  / \  2 3  / \   * 1. 5 |

* 1. What is array? Explain one -dimensional and multi -dimensional array with an example

**Array:** An array is a collection of elements of the same data type that are stored in contiguous memory locations. Each element in an array is identified by an index or a key.

**One-Dimensional Array:** A one-dimensional array is a linear collection of elements, where each element has a unique index or position.

**Example:** **int numbers[5] = {10, 20, 30, 40, 50};**

**Multi-Dimensional Array:** A multi-dimensional array is an array with more than one dimension. It can be thought of as an array of arrays.

**Example:**

**int matrix[3][3] = {**

**{1, 2, 3},**

**{4,5,6},**

**{7, 8, 9}**

**};**

* 1. What are the disadvantages of array over Queue with an example.

**Disadvantages of Array Over Queue** :

* *Fixed Size Limitation :* Arrays have a fixed size, which must be declared at the beginning. This limitation becomes problematic when the size of the data set is dynamic or unknown in advance.
* *Memory Wastage :* Arrays allocate memory for a predetermined number of elements, regardless of the actual number of elements present. This can result in memory wastage, especially when the array is not fully utilized.
* *Inefficient Dequeue Operation*: In a queue implemented using an array, the dequeue operation can be inefficient. When elements are dequeued from the front, the remaining elements must be shifted to fill the gap. This process is time-consuming, especially for large arrays.
* *Example*:

#include <stdio.h>

#define MAX\_SIZE 5

int queue[MAX\_SIZE];

int front = -1;

int rear = -1;

*// Enqueue operation*

void enqueue(int item) {

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

printf("Queue is full. Cannot enqueue %d.\n", item);

return;

}

if (front == -1) {

*// Queue is empty, set front to* 0

front = 0;

}

*// Increment rear and enqueue the item*

rear++;

queue[rear] = item;

}

*// Dequeue operation*

void dequeue() {

if (front == -1) {

printf("Queue is empty. Cannot dequeue.\n");

return;

}

*// Dequeue the item and increment front*

int dequeuedItem = queue[front];

front++;

*// If front surpasses rear, reset front and rear to -1*

if (front > rear) {

front = rear = -1;

}

printf("Dequeued: %d\n", dequeuedItem);

}

*// Display the elements in the queue*

void display() {

if (front == -1) {

printf("Queue is empty.\n");

return;

}

printf("Queue: ");

for (int i = front; i <= rear; i++) {

printf("%d ", queue[i]);

}

printf("\n");

}

int main() {

enqueue(10);

enqueue(20);

enqueue(30);

display();

dequeue();

display();

enqueue(40);

enqueue(50);

enqueue(60); // Attempting to enqueue when the queue is full

display();

return 0;

}

* 1. Convert the given infix operation to postfix expression: ((A + B) – C \* (D / E)) + F

((A + B) - C \* (D / E)) + F

(AB+-C\*DE/)+F

(AB+-CDE/\*)+F

AB+CDE/\*-+F

A B + C D E / \* - F +

* 1. Convert the given infix operation to prefix expression: (A – B/C) \* (A/K-L)

(A - B/C) \* (A/K - L)

(A-/BC)\*(A/-KL)

(-A/BC)\*(/A-KL)

\*-A/BC/A-KL

\*-A / B C / A - K L

* 1. What are the advantages of Circular Queue? Explain with an example.
* **Efficient space utilization** – Circular queues make the most of available space by reusing it once items are removed, avoiding wasted memory areas.
* **No need to shift elements** – When removing or adding items, there’s no need to move other elements around, which simplifies the process.
* **Constant time operations** – Operations like insertion and deletion take a consistent amount of time, making performance predictable and stable.
* **Reusable memory slots** – As items are processed, the space they occupied becomes immediately available for new items, which optimizes memory use.
* **Suitable for cyclic processes** – They are ideal for situations where processes repeat in a cycle, such as scheduling tasks in an operating system.
* **Example:** Consider a scenario where a printer is receiving print jobs. A circular queue can be employed to manage these print jobs efficiently:

#include <stdio.h>

#define MAX\_SIZE 5

int circularQueue[MAX\_SIZE];

int front = -1;

int rear = -1;

*// Function to check if the circular queue is empty*

int isEmpty() {

return front == -1;

}

*// Function to check if the circular queue is full*

int isFull() {

return (front == 0 && rear == MAX\_SIZE - 1) || (front == rear + 1);

}

*// Function to enqueue an element into the circular queue*

void enqueue(int item) {

if (isFull()) {

printf("Circular Queue is full. Cannot enqueue %d.\n", item);

return;

}

if (isEmpty()) {

front = 0;

}

rear = (rear + 1) % MAX\_SIZE;

circularQueue[rear] = item;

printf("Enqueued: %d\n", item);

}

*// Function to dequeue an element from the circular queue*

void dequeue() {

if (isEmpty()) {

printf("Circular Queue is empty. Cannot dequeue.\n");

return;

}

int dequeuedItem = circularQueue[front];

printf("Dequeued: %d\n", dequeuedItem);

if (front == rear) {

*// Last element is dequeued, reset the queue*

front = rear = -1;

} else {

front = (front + 1) % MAX\_SIZE;

}

}

*// Function to display the elements in the circular queue*

void display() {

if (isEmpty()) {

printf("Circular Queue is empty.\n");

return;

}

printf("Circular Queue: ");

int i = front;

do {

printf("%d ", circularQueue[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

enqueue(10);

enqueue(20);

enqueue(30);

display();

dequeue();

display();

enqueue(40);

enqueue(50);

enqueue(60); *// Attempting to enqueue when the queue is full*

display();

return 0;

}

* 1. Explain the child node, parent node, root node and leaf node with an example.

**Child node**: The node which is the descendant(node which follows another node) of any node on a tree is known as a child node.

**Or**

A child node is a node that has a parent node. It is connected below its parent node in the tree structure.

**Parent node:** the node which is predecessor of another node is known as parent node.

**Or**

A parent node is a node that has one or more child nodes. It is situated above its child nodes in the hierarchy.

**Root node:** A node that is the first or topmost node in a tree is called a root node. In every tree, there is always one root node, which is the only node that has never previously been connected to another node

**Or**

The root node is the topmost node in a tree hierarchy. It has no parent, as it is the starting point of the tree.

**Leaf node:** node that does not have any child node is called a leaf node.

**Or**

A leaf node is a node that has no children. It is situated at the end of a branch in the tree structure.

Example:

A (Root)

/ \

B C (Parents)

/ \ \

D E F (Children)

/

G (Leaf)

* 1. What is Stack? Explain the uses of stack with operations of stack.

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. In a stack, elements are added or removed from the same end, known as the top. The operations on a stack are performed at the top of the stack.

Uses of Stack:

* **Function calls and recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function’s execution.
* **Undo/Redo operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.
* **Expression evaluation**: Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack’s top elements.
* **Browser history:** Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.
* **Backtracking Algorithms**: The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack

***Operations on a Stack****:*

*Push:*

*Adds an element to the top of the stack.*

*Pop:*

*Removes and returns the element from the top of the stack.*

*Peek (or Top):*

*Returns the element at the top of the stack without removing it.*

*isEmpty:*

*Checks if the stack is empty.*

*isFull:*

*Checks if the stack is full (applicable in fixed-size implementations).*

*Size:*

*Returns the number of elements currently in the stack*.

**10 marks questions**

* + 1. Write a C program to sort the following: 6, 5,3,1,8,7,2,4 using quick sort and explain the run time complexity of the above

#include <stdio.h>

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return i + 1;

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

void printArray(int arr[], int size) {

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

int arr[] = {6, 5, 3, 1, 8, 7, 2, 4};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

printArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

*The worst-case time complexity of the Quick Sort algorithm is O(n^2), but on average, it is O(n log n)*

* + 1. Write an algorithm to insert and delete an element in a stack.

**Algorithm to insert an element in stack:**

Let Stack[size] is an array , TOP=topmost element and

ITEM=element to be pushed into stack

PUSH(STACK, TOP, SIZE, ITEM)

Step1: if(TOP=SIZE=1), then

Display OVERFLOW

Exit

Step2: Else

TOP=TOP+1

STACK[TOP]=ITEM

[end of if]

Step3: exit

**Algorithm to delete an element in stack:**

Step1: Let ITEM

Step2: if(TOP<0) then

Display UNDERFLOW

Exit

Step3: Else

ITEM=STACK[TOP]

Display “popped item is ITEM”

TOP=TOP-1

[end of if]

Step4: Exit

* + 1. Explain Best- case, average- case and worst-case complexity taking an example of your own

**Best-Case Complexity:** The best-case complexity represents the minimum amount of resources (time or space) required by an algorithm for the most favorable input. In the context of searching for an element in an array, the best-case scenario occurs when the target element is found at the very beginning of the array. The time complexity in this case is O(1), indicating constant time, as the algorithm needs to perform only one comparison to find the element.

**Average-Case Complexity:** The average-case complexity represents the expected amount of resources required by an algorithm when given inputs of average difficulty. For the linear search algorithm in an unsorted array, the average-case scenario involves searching for an element at a random position within the array. On average, you would need to search through half of the array before finding the target element or determining that it's not present. The average-case time complexity for a linear search in an unsorted array is O(n/2), which simplifies to O(n) in Big O notation.

**Worst-Case Complexity:** The worst-case complexity represents the maximum amount of resources required by an algorithm for the least favorable input. In the worst-case scenario for a linear search, the target element is either at the very end of the array or not present at all, requiring the algorithm to traverse the entire array. The worst-case time complexity for a linear search in an unsorted array is O(n), where n is the size of the array.

* + 1. Write a C Program to insert, delete and traverse in a single linked list.

#include <stdio.h>

#include <stdlib.h>

*// Node structure for the linked list*

struct Node {

int data;

struct Node\* next;

};

*// Function to insert a new node at the beginning of the list*

void insertAtBeginning(struct Node\*\* head, int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

if (newNode == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

newNode->data = value;

newNode->next = \*head;

\*head = newNode;

printf("Inserted %d at the beginning\n", value);

}

*// Function to delete a node with a given value from the list*

void deleteNode(struct Node\*\* head, int value) {

struct Node\* temp = \*head;

struct Node\* prev = NULL;

*// Find the node to be deleted*

while (temp != NULL && temp->data != value) {

prev = temp;

temp = temp->next;

}

*// If the node is not present*

if (temp == NULL) {

printf("%d not found in the list\n", value);

return;

}

*// Remove the node*

if (prev == NULL) {

\*head = temp->next;

} else {

prev->next = temp->next;

}

printf("Deleted %d from the list\n", value);

free(temp);

}

*// Function to traverse and print the elements of the list*

void traverseList(struct Node\* head) {

printf("List elements: ");

while (head != NULL) {

printf("%d ", head->data);

head = head->next;

}

printf("\n");

}

*// Function to free the memory occupied by the list*

void freeList(struct Node\*\* head) {

struct Node\* current = \*head;

struct Node\* next;

while (current != NULL) {

next = current->next;

free(current);

current = next;

}

\*head = NULL;

}

int main() {

struct Node\* head = NULL;

*// Inserting elements at the beginning*

insertAtBeginning(&head, 3);

insertAtBeginning(&head, 7);

insertAtBeginning(&head, 11);

*// Traversing and printing the list*

traverseList(head);

*// Deleting a node*

deleteNode(&head, 7);

*// Traversing and printing the list after deletion*

traverseList(head);

*// Freeing the memory occupied by the list*

freeList(&head);

return 0;

}

* + 1. Write a C Program to search a number ‘8’ from the following: 6, 5,3,1,8,7,2,4

#include <stdio.h>

*// Function to search for a number in an array*

int searchNumber(int arr[], int size, int target) {

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

if (arr[i] == target) {

return i; *// Return the index if found*

}

}

return -1*; // Return -1 if the number is not found*

}

int main() {

*// Given array*

int array[] = {6, 5, 3, 1, 8, 7, 2, 4};

int size = sizeof(array) / sizeof(array[0]);

*// Number to search*

int target = 8;

*// Search for the number*

int index = searchNumber(array, size, target);

*// Display the result*

if (index != -1) {

printf("Number %d found at index %d\n", target, index);

} else {

printf("Number %d not found in the array\n", target);

}

return 0;

}

* + 1. Write a C program to sort the following: 4,8,6,5,3,1,7,2 using quick sort and explain the run time complexity of the above

#include <stdio.h>

*// Function to swap two elements in an array*

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

*// Function to partition the array and return the pivot index*

int partition(int arr[], int low, int 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]);

return i + 1;

}

*// Function to implement quicksort*

void quicksort(int arr[], int low, int high) {

if (low < high) {

int pivotIndex = partition(arr, low, high);

*// Recursively sort the sub-arrays*

quicksort(arr, low, pivotIndex - 1);

quicksort(arr, pivotIndex + 1, high);

}

}

*// Function to print the array*

void printArray(int arr[], int size) {

printf("Sorted array: ");

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

printf("%d ", arr[i]);

}

printf("\n");

}

int main() {

*// Given array*

int array[] = {4, 8, 6, 5, 3, 1, 7, 2};

int size = sizeof(array) / sizeof(array[0]);

*// Sorting using quicksort*

quicksort(array, 0, size - 1);

*// Printing the sorted array*

printArray(array, size);

return 0;

}

The average-case time complexity of quicksort is O(n log n). This makes it a very efficient sorting algorithm for large datasets. However, it's important to note that the worst-case time complexity is O(n^2), which occurs when the chosen pivot element consistently partitions the array in an unbalanced way,

* + 1. Write an algorithm to insert and delete an element in a Queue

**Algorithm to insert element in a Queue (enqueue)**

QINSERT(QUEUE[MAXSIZE], ITEM)

Step1: if REAR=MAXSIZE-1 then

Display OVERFLOW

Exit

Step2: Else if FRONT=1 or REAR=-1, then

FRONT=0

REAR=0

Step3: else

REAR=REAR+1

[end of if]

Step4: QUEUE[REAR]=ITEM

Step5: exit

**Algorithm to delete element in a Queue(dequeue)**

QDELETE(QUEUE[MAXSIZE])

Step1: Let ITEM

Step2: if FRONT=1 or REAR=-1 then

Display UNDERFLOW

[end of if]

Step3: ITEM=QUEUE[FRONT]

Step4: display “delete item is” ITEM

Step5: else if FRONT=REAR then

[when queue has only one element]

FRONT=-1

REAR=-1

Step6: else

FRONT=FRONT+1

Step7: exit

* + 1. Write a C Program to insert, delete and traverse in a Linear Queue.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 5

*// Structure to represent a linear queue*

struct Queue {

int front, rear;

int data[MAX\_SIZE];

};

*// Function to initialize a queue*

void initializeQueue(struct Queue \*q) {

q->front = -1;

q->rear = -1;

}

*// Function to check if the queue is empty*

int isEmpty(struct Queue \*q) {

return q->front == -1;

}

*// Function to check if the queue is full*

int isFull(struct Queue \*q) {

return (q->rear == MAX\_SIZE - 1 && q->front == 0) || (q->rear == q->front - 1);

}

*// Function to insert (enqueue) an element into the queue*

void enqueue(struct Queue \*q, int value) {

if (isFull(q)) {

printf("Queue Overflow\n");

return;

}

if (isEmpty(q)) {

q->front = 0;

}

q->rear = (q->rear + 1) % MAX\_SIZE;

q->data[q->rear] = value;

printf("Enqueued: %d\n", value);

}

*// Function to delete (dequeue) an element from the queue*

int dequeue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue Underflow\n");

return -1;

}

int value = q->data[q->front];

if (q->front == q->rear) {

*// Reset the queue when the last element is dequeued*

initializeQueue(q);

} else {

q->front = (q->front + 1) % MAX\_SIZE;

}

printf("Dequeued: %d\n", value);

return value;

}

*// Function to traverse and print the elements of the queue*

void traverseQueue(struct Queue \*q) {

if (isEmpty(q)) {

printf("Queue is empty\n");

return;

}

printf("Queue elements: ");

int i = q->front;

do {

printf("%d ", q->data[i]);

i = (i + 1) % MAX\_SIZE;

} while (i != (q->rear + 1) % MAX\_SIZE);

printf("\n");

}

int main() {

struct Queue myQueue;

initializeQueue(&myQueue);

enqueue(&myQueue, 3);

enqueue(&myQueue, 7);

enqueue(&myQueue, 11);

traverseQueue(&myQueue);

dequeue(&myQueue);

traverseQueue(&myQueue);

enqueue(&myQueue, 5);

enqueue(&myQueue, 9);

traverseQueue(&myQueue);

enqueue(&myQueue, 13); *// Trying to enqueue when the queue is full*

return 0;

}

* + 1. Write a C Program to search a number ‘2’ from the following: 6, 5,3,1,8,7,2,4

#include <stdio.h>

*// Function to search for a number in an array*

int searchNumber(int arr[], int size, int target) {

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

if (arr[i] == target) {

return i; *// Return the index if found*

}

}

return -1*; // Return -1 if the number is not found*

}

int main() {

*// Given array*

int array[] = {6, 5, 3, 1, 8, 7, 2, 4};

int size = sizeof(array) / sizeof(array[0]);

*// Number to search*

int target = 2;

*// Search for the number*

int index = searchNumber(array, size, target);

*// Display the result*

if (index != -1) {

printf("Number %d found at index %d\n", target, index);

} else {

printf("Number %d not found in the array\n", target);

}

return 0;

}