**Arrays**

**Definition:** In C, an array is a collection of variables of the same type, stored at contiguous memory locations.

**Properties:**

* Fixed size: The size must be defined at compile time.
* Homogeneous elements: All elements must be of the same type.
* Contiguous memory: Elements are stored in consecutive memory locations.

**Operations:**

* **Declaration:** **int arr[10];**
* **Accessing Elements:** **arr[index];**
* **Traversal:** **for(int i = 0; i < n; i++) { printf("%d", arr[i]); }**
* **Insertion:** **arr[index] = value;**
* **Deletion:** Set the element to a sentinel value or shift elements.

**Merits and Demerits:**

* **Merits:**
  + Simple and easy to understand.
  + Fast access time using index.
* **Demerits:**
  + Fixed size can lead to inefficient use of memory.
  + Insertion and deletion are costly operations.

**Applications:**

* Used in situations where the size of the data is known and fixed.
* Suitable for implementing other data structures like stacks and queues.

**Linked Lists**

**Definition:** In C, a linked list is a linear collection of data elements, called nodes, where the linear order is maintained by means of pointers.

**Properties:**

* Dynamic size: Grows and shrinks in size as needed.
* Nodes: Each node contains data and a pointer to the next node.
* Non-contiguous storage: Nodes are stored in different memory locations.

**Operations:**

* **Node Structure:**

struct Node {

int data;

struct Node\* next;

};

* **Insertion:**

void insert(struct Node\*\* head, int newData) {

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

newNode->data = newData;

newNode->next = (\*head);

(\*head) = newNode;

}

* **Deletion:**

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

struct Node\* temp = \*head, \*prev;

if (temp != NULL && temp->data == key) {

\*head = temp->next;

free(temp);

return;

}

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

prev = temp;

temp = temp->next;

}

if (temp == NULL) return;

prev->next = temp->next;

free(temp);

}

* **Traversal:**

void printList(struct Node\* n) {

while (n != NULL) {

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

n = n->next;

}

}

**Merits and Demerits:**

* **Merits:**
  + Dynamic size allows for efficient memory use.
  + Easy insertion and deletion of nodes.
* **Demerits:**
  + Extra memory is required for storing pointers.
  + No direct access to elements; requires traversal from the head.

**Applications:**

* Implementation of other data structures like stacks and queues.
* Efficient manipulation of data when the number of elements is unknown.

**Stacks**

**Definition:** In C, a stack is a linear data structure that follows the Last In, First Out (LIFO) principle.

**Properties:**

* LIFO order: The last element added is the first to be removed.
* Uses a single pointer to the top of the stack.

**Operations:**

* **Stack Structure:**

struct Stack {

int top;

unsigned capacity;

int\* array;

};

* **Create Stack:**

struct Stack\* createStack(unsigned capacity) {

struct Stack\* stack = (struct Stack\*) malloc(sizeof(struct Stack));

stack->capacity = capacity;

stack->top = -1;

stack->array = (int\*) malloc(stack->capacity \* sizeof(int));

return stack;

}

* **Push:**

void push(struct Stack\* stack, int item) {

if (stack->top == stack->capacity - 1) return;

stack->array[++stack->top] = item;

}

* **Pop:**

int pop(struct Stack\* stack) {

if (stack->top == -1) return -1;

return stack->array[stack->top--];

}

**Merits and Demerits:**

* **Merits:**
  + Simple to implement.
  + Efficient for algorithms requiring reverse processing.
* **Demerits:**
  + Limited access, only the top element can be accessed.
  + Can cause stack overflow if not managed properly.

**Applications:**

* Expression evaluation and syntax parsing.
* Backtracking algorithms (e.g., maze solving).
* Function call management in recursion.

**Queues**

**Definition:** In C, a queue is a linear data structure that follows the First In, First Out (FIFO) principle.

**Properties:**

* FIFO order: The first element added is the first to be removed.
* Uses two pointers for front and rear of the queue.

**Operations:**

* **Queue Structure:**

struct Queue {

int front, rear, size;

unsigned capacity;

int\* array;

};

* **Create Queue:**

struct Queue\* createQueue(unsigned capacity) {

struct Queue\* queue = (struct Queue\*) malloc(sizeof(struct Queue));

queue->capacity = capacity;

queue->front = queue->size = 0;

queue->rear = capacity - 1;

queue->array = (int\*) malloc(queue->capacity \* sizeof(int));

return queue;

}

* **Enqueue:**

void enqueue(struct Queue\* queue, int item) {

if (queue->size == queue->capacity) return;

queue->rear = (queue->rear + 1) % queue->capacity;

queue->array[queue->rear] = item;

queue->size = queue->size + 1;

}

* **Dequeue:**

int dequeue(struct Queue\* queue) {

if (queue->size == 0) return -1;

int item = queue->array[queue->front];

queue->front = (queue->front + 1) % queue->capacity;

queue->size = queue->size - 1;

return item;

}

**Merits and Demerits:**

* **Merits:**
  + Simple implementation.
  + Efficient for managing tasks in order.
* **Demerits:**
  + Limited access to front and rear elements.
  + Fixed size can lead to overflow.

**Applications:**

* Task scheduling.
* Breadth-First Search (BFS) in graphs.
* Managing requests in servers

**Trees**

**Definition:** In C, a tree is a hierarchical data structure consisting of nodes, with a root node and child nodes forming a parent-child relationship.

**Properties:**

* Hierarchical structure.
* Each node contains data and pointers to its child nodes.
* Root node with zero or more child nodes.

**Operations:**

* **Node Structure:**

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

* **Insertion:**

struct Node\* newNode(int data) {

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

node->data = data;

node->left = node->right = NULL;

return node;

}

* **Traversal:**

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

**Merits and Demerits:**

* **Merits:**
  + Efficient for hierarchical data storage and retrieval.
  + Reflects hierarchical relationships.
* **Demerits:**
  + Complex implementation and management.
  + Can become unbalanced, leading to inefficient operations.

**Applications:**

* Database indexing.
* File system management.
* Hierarchical data representation (e.g., organization charts).

**Graphs**

**Definition:** In C, a graph is a non-linear data structure consisting of nodes (vertices) and edges connecting pairs of nodes.

**Properties:**

* Consists of vertices and edges.
* Can be directed or undirected.
* Can be weighted or unweighted.

**Operations:**

**Graph Representation (Adjacency List):**

struct AdjListNode {

int dest;

struct AdjListNode\* next;

};

struct AdjList {

struct AdjListNode\* head;

};

struct Graph {

int V;

struct AdjList\* array;

};

* **Create Graph:**

struct Graph\* createGraph(int V) {

struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));

graph->V = V;

graph->array = (struct AdjList\*) malloc(V \* sizeof(struct AdjList));

for (int i = 0; i < V; ++i)

graph->array[i].head = NULL;

return graph;

}

**Merits and Demerits:**

**Merits:**

* Suitable for representing complex relationships
* Flexible and can represent various structures

**Demerits:**

* Can be complex to implement and manage
* Requires significant memory for storage

**Applications:**

* Network routing algorithms
* Social network analysis
* Pathfinding algorithms in maps