**Data structures in C**

**Arrays**

**Description**: A collection of elements identified by index or key.

**Operations**: Access (O(1)), Insertion (O(n)), Deletion (O(n)).

**Use Cases**: Implementing other data structures, storing data that is accessed sequentially.

**Example code for Arrays using C:**

#include <stdio.h>

Int main() {

Int arr[5] = {1, 2, 3, 4, 5};

For (int I = 0; I < 5; i++) {

Printf(“%d “, arr[i]);

}

Return 0;

}

**2. Linked Lists**

**Description**: A linear collection of elements, where each element points to the next.

**Types**:

Singly Linked List: Each node points to the next node.

**Doubly Linked List**: Each node points to both the next and previous nodes.

**Circular Linked List**: The last node points back to the first node.

**Operations**: Insertion (O(1) at head), Deletion (O(1) if node is known), Access (O(n)).

**Use Cases**: Dynamic memory allocation, implementing stacks and queues.

**Example prog:**

#include <stdio.h>

#include <stdlib.h>

Struct Node {

Int data;

Struct Node\* next;

};

Void printList(struct Node\* n) {

While (n != NULL) {

Printf(“%d “, n->data);

N = n->next;

}

}

Int main() {

Struct Node\* head = NULL;

Struct Node\* second = NULL;

Struct Node\* third = NULL;

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

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

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

Head->data = 1;

Head->next = second;

Second->data = 2;

Second->next = third;

Third->data = 3;

Third->next = NULL;

printList(head);

return 0;

}

**3. Stacks**

**Description**: A LIFO (Last In, First Out) structure.

**Operations**: Push (O(1)), Pop (O(1)), Peek (O(1)).

**Use Cases:** Backtracking, function call management (recursion), undo mechanisms.

**Example prog**:

#include <stdio.h>

#include <stdlib.h>

#define MAX 1000

Struct Stack {

Int top;

Int arr[MAX];

};

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

If (stack->top == MAX – 1) {

Printf(“Stack overflow\n”);

Return;

}

Stack->arr[++stack->top] = item;

}

Int pop(struct Stack\* stack) {

If (stack->top == -1) {

Printf(“Stack underflow\n”);

Return -1;

}

Return stack->arr[stack->top--];

}

Int main() {

Struct Stack stack;

Stack.top = -1;

Push(&stack, 10);

Push(&stack, 20);

Push(&stack, 30);

Printf(“%d popped from stack\n”, pop(&stack));

Return 0;

}

**4. Queues**

**Description**: A FIFO (First In, First Out) structure.

**Types**:

**Simple Queue**: Basic FIFO queue.

**Circular Queue**: The last position is connected back to the first position.

**Priority Queue:** Elements are dequeued based on priority.

**Deque:** Double-ended queue allowing insertion and deletion at both ends.

**Operations**: Enqueue (O(1)), Dequeue (O(1)), Peek (O(1)).

**Use Cases**: Task scheduling, handling asynchronous data (e.g., IO buffers).

**Example prog**:

#include <stdio.h>

#include <stdlib.h>

#define MAX 1000

struct Queue {

int front, rear, size;

int arr[MAX];

};

struct Queue\* createQueue() {

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

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

queue->rear = MAX - 1;

return queue;

}

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

if (queue->size == MAX) {

printf("Queue overflow\n");

return;

}

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

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

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

}

int dequeue(struct Queue\* queue) {

if (queue->size == 0) {

printf("Queue underflow\n");

return -1;

}

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

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

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

return item;

}

int main() {

struct Queue\* queue = createQueue();

enqueue(queue, 10);

enqueue(queue, 20);

enqueue(queue, 30);

printf("%d dequeued from queue\n", dequeue(queue));

return 0;

}

**5. Trees**

**Description**: A hierarchical structure with a root node and child nodes.

**Types**:

**Binary Tree**: Each node has at most two children.

**Binary Search Tree (BST)**: A binary tree where left child < parent < right child.

**AVL Tree**: A self-balancing binary search tree.

**B-tree**: A balanced tree suitable for disk storage.

**Operations**: Insertion (O(log n)), Deletion (O(log n)), Search (O(log n)).

**Use Cases**: Database indexing, file systems, syntax trees.

**Example prog**:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

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

node->data = data;

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

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

return node;

}

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

inorder(root);

return 0;

}

**6. Heaps**

**Description:** A special tree-based structure that satisfies the heap property.

**Types**:

**Max-Heap**: Parent nodes are greater than or equal to child nodes.

**Min-Heap**: Parent nodes are less than or equal to child nodes.

**Operations**: Insertion (O(log n)), Deletion (O(log n)), Peek (O(1)).

**Use Cases**: Priority queues, heap sort, algorithm optimization (e.g., Dijkstra’s algorithm).

**Example prog**:

#include <stdio.h>

Void heapify(int arr[], int n, int i) {

Int largest = I;

Int left = 2 \* I + 1;

Int right = 2 \* I + 2;

If (left < n && arr[left] > arr[largest])

Largest = left;

If (right < n && arr[right] > arr[largest])

Largest = right;

If (largest != i) {

Int temp = arr[i];

Arr[i] = arr[largest];

Arr[largest] = temp;

Heapify(arr, n, largest);

}

}

Void heapSort(int arr[], int n) {

For (int I = n / 2 – 1; I >= 0; i--)

Heapify(arr, n, i);

For (int I = n – 1; I > 0; i--) {

Int temp = arr[0];

Arr[0] = arr[i];

Arr[i] = temp;

Heapify(arr, I, 0);

}

}

Void printArray(int arr[], int n) {

For (int I = 0; I < n; ++i)

Printf(“%d “, arr[i]);

Printf(“\n”);

}

Int main() {

Int arr[] = {12, 11, 13, 5, 6, 7};

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

heapSort(arr, n);

printf(“Sorted array is \n”);

printArray(arr, n);

return 0;

}

**7. Graphs**

**Description**: A set of nodes (vertices) connected by edges.

**Types**:

**Directed Graph (Digraph)**: Edges have a direction.

**Undirected Graph**: Edges have no direction.

**Weighted Graph**: Edges have weights.

**Operations**: Traversal (O(V + E)), Shortest path (Dijkstra’s, O(V^2) or O(E + V log V) with priority queue).

**Use Cases**: Network routing, social networks, dependency resolution.

**Example prog**:

#include <stdio.h>

#include <stdlib.h>

Struct Node {

Int dest;

Struct Node\* next;

};

Struct AdjList {

Struct Node\* head;

};

Struct Graph {

Int V;

Struct AdjList\* array;

};

Struct Node\* newNode(int dest) {

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

Node->dest = dest;

Node->next = NULL;

Return node;

}

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;

}

Void addEdge(struct Graph\* graph, int src, int dest) {

Struct Node\* node = newNode(dest);

Node->next = graph->array[src].head;

Graph->array[src].head = node;

Node = newNode(src);

Node->next = graph->array[dest].head;

Graph->array[dest].head = node;

}

Void printGraph(struct Graph\* graph) {

For (int v = 0; v < graph->V; ++v) {

Struct Node\* pCrawl = graph->array[v].head;

Printf(“\n Adjacency list of vertex %d\n head “, v);

While (pCrawl) {

Printf(“-> %d”, pCrawl->dest);

pCrawl = pCrawl->next;

}

Printf(“\n”);

}

}

Int main() {

Int V = 5;

Struct Graph\* graph = createGraph(V);

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

printGraph(graph);

return 0;

}

**8. Hash Tables**

**Description**: A structure that maps keys to values using a hash function.

**Operations**: Insertion (O(1)), Deletion (O(1)), Search (O(1)).

**Use Cases:** Implementing associative arrays, database indexing.

**Example prog**:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define SIZE 10

Struct DataItem {

Int data;

Int key;

};

Struct DataItem\* hashArray[SIZE];

Struct DataItem\* dummyItem;

Struct DataItem\* item;

Int hashCode(int key) {

Return key % SIZE;

}

Struct DataItem\* search(int key) {

Int hashIndex = hashCode(key);

While (hashArray[hashIndex] != NULL) {

If (hashArray[hashIndex]->key == key)

Return hashArray[hashIndex];

++hashIndex;

hashIndex %= SIZE;

}

Return NULL;

}

Void insert(int key, int data) {

Struct DataItem\* item = (struct DataItem\*) malloc(sizeof(struct DataItem));

Item->data = data;

Item->key = key;

Int hashIndex = hashCode(key);

While (hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {

++hashIndex;

hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}

Struct DataItem\* delete(struct DataItem\* item) {

Int key = item->key;

Int hashIndex = hashCode(key);

While (hashArray[hashIndex] != NULL) {

If (hashArray[hashIndex]->key == key) {

Struct DataItem\* temp = hashArray[hashIndex];

hashArray[hashIndex] = dummyItem;

return temp;

}

++hashIndex;

hashIndex %= SIZE;

}

Return NULL;

}

Void display() {

For (int I = 0; I < SIZE; i++) {

If (hashArray[i] != NULL)

Printf(“ (%d,%d)”, hashArray[i]->key, hashArray[i]->data);

Else

Printf(“ ~~ “);

}

Printf(“\n”);

}

Int main() {

dummyItem = (struct DataItem\*) malloc(sizeof(struct DataItem));

dummyItem->data = -1;

dummyItem->key = -1;

insert(1, 20);

insert(2, 70);

insert(42, 80);

insert(4, 25);

insert(12, 44);

insert(14, 32);

insert(17, 11);

insert(13, 78);

insert(37, 97);

display();

item = search(37);

if (item != NULL) {

printf(“Element found: %d\n”, item->data);

} else {

Printf(“Element not found\n”);

}

Delete(item);

Item = search(37);

If (item != NULL) {

Printf(“Element found: %d\n”, item->data);

} else {

Printf(“Element not found\n”);

}

}