**### Data Structures in C**

**#### 1. Introduction to Data Structures**

- \*\*Definition\*\*: A data structure is a way of organizing and storing data so that it can be accessed and worked with efficiently.

- \*\*Importance\*\*: Understanding data structures is crucial for efficient algorithm design and problem-solving.

**#### 2. Basic Concepts in C**

- \*\*Pointers\*\*: Essential for dynamic data structures.

- \*\*Dynamic Memory Allocation\*\*: Using `malloc`, `calloc`, `realloc`, and `free` to manage memory.

**#### 3. Linear Data Structures**

**##### Arrays**

- \*\*Definition\*\*: A collection of elements identified by index or key.

- \*\*Operations\*\*: Traversal, insertion, deletion, and searching.

- \*\*Example\*\*:

```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;

}

```

**##### Linked Lists**

- \*\*Singly Linked List\*\*:

- \*\*Node Definition\*\*:

```c

struct Node {

int data;

struct Node\* next;

};

```

- **\*\*Basic Operations\*\*:**

- **\*\*Insertion:**

void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = (\*head\_ref);

(\*head\_ref) = new\_node;

}

```

- \*\*Traversal\*\*:

```c

void printList(struct Node\* node) {

while (node != NULL) {

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

node = node->next;

}

printf("NULL\n");

}

```

**- \*\*Example\*\*:**

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 1);

insertAtBeginning(&head, 2);

printList(head);

return 0;

}

```

**- \*\*Doubly Linked List\*\*:**

**- \*\*Node Definition\*\*:**

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

**- \*\*Example\*\*:**

void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = (\*head\_ref);

new\_node->prev = NULL;

if ((\*head\_ref) != NULL)

(\*head\_ref)->prev = new\_node;

(\*head\_ref) = new\_node;

}

void printList(struct Node\* node) {

struct Node\* last;

printf("Traversal in forward direction:\n");

while (node != NULL) {

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

last = node;

node = node->next;

}

printf("NULL\n");

printf("Traversal in reverse direction:\n");

while (last != NULL) {

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

last = last->prev;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

insertAtBeginning(&head, 1);

insertAtBeginning(&head, 2);

printList(head);

return 0;

}

**##### Stacks**

- \*\*Definition\*\*: A linear data structure that follows LIFO (Last In, First Out) principle.

**- \*\*Operations\*\*:** Push, pop, peek.

**- \*\*Array-based Implementation\*\***

#include <stdio.h>

#define MAX 1000

struct Stack {

int top;

int arr[MAX];

};

**PUSH**

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

if (stack->top == MAX - 1) {

printf("Stack overflow\n");

return;

}

stack->arr[++stack->top] = data;

}

int pop(struct Stack\* stack) {

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

printf("Stack underflow\n");

return -1;

}

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

}

**POP**

int main() {

struct Stack stack;

stack.top = -1;

push(&stack, 10);

push(&stack, 20);

printf("Popped element: %d\n", pop(&stack));

return 0;

}

```

**##### Queues**

**- \*\*Definition\*\*:** A linear data structure that follows FIFO (First In, First Out) principle.

**- \*\*Operations\*\*:** Enqueue, dequeue, front, rear.

**- \*\*Array-based Implementation\*\***

#include <stdio.h>

#define MAX 1000

struct Queue {

int front, rear, size;

int arr[MAX];

};

**ENQUEUE**

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

if (queue->rear == MAX - 1) {

printf("Queue overflow\n");

return;

}

queue->arr[++queue->rear] = data;

if (queue->front == -1) {

queue->front = 0;

}

}

**DEQUEUE**

int dequeue(struct Queue\* queue) {

if (queue->front == -1 || queue->front > queue->rear) {

printf("Queue underflow\n");

return -1;

}

return queue->arr[queue->front++];

}

int main() {

struct Queue queue;

queue.front = queue.rear = -1;

enqueue(&queue, 10);

enqueue(&queue, 20);

printf("Dequeued element: %d\n", dequeue(&queue));

return 0;

}

```

**#### 4. Non-Linear Data Structures**

**##### Trees**

**- \*\*Binary Tree\*\*:**

**- \*\*Node Definition\*\*:**

```c

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

```

**- \*\*Insertion\*\*:**

```c

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\*\*:**

```c

void inOrder(struct Node\* root) {

if (root != NULL) {

inOrder(root->left);

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

inOrder(root->right);

}

}

```

**##### Graphs**

**- \*\*Adjacency Matrix\*\*:**

#include <stdio.h>

#define V 4

void printGraph(int graph[V][V]) {

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

for (int j = 0; j < V; j++) {

printf("%d ", graph[i][j]);

}

printf("\n");

}

}

int main() {

int graph[V][V] = {

{0, 1, 0, 1},

{1, 0, 1, 0},

{0, 1, 0, 1},

{1, 0, 1, 0}

};

printGraph(graph);

return 0;

}

```

**- \*\*Adjacency List\*\*:**

#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;

}

```

**#### 5. Advanced Data Structures**

##### Hash Tables

- \*\*Definition\*\*: A data structure that maps keys to values using a hash function.

- \*\*Example\*\*:

```c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct HashItem {

char\* key;

int value;

};

struct HashTable {

struct HashItem\*\* items;

int size;

};

unsigned int hashFunction(char\* key, int size) {

unsigned int hash = 0;

for (int i = 0; key[i]; i++) {

hash = (hash << 5) + key[i];

}

return hash % size;

}

struct HashTable\* createHashTable(int size) {

struct HashTable\* table = (struct HashTable\*)malloc(sizeof(struct HashTable));

table->size = size;

table->items = (struct HashItem\*\*)calloc(size, sizeof(struct HashItem\*));

return table;

}

void insert(struct HashTable\* table, char\* key, int value) {

unsigned int index = hashFunction(key, table->size);

struct HashItem\* item = (struct HashItem\*)malloc(sizeof(struct HashItem));

item->key = strdup(key);

item->value = value;

table->items[index] = item;

}

int search(struct HashTable\* table, char\* key) {

unsigned int index = hashFunction(key, table->size);

struct HashItem\* item = table->items[index];

if (item == NULL) return -1;

return item->value;

}

int main() {

struct HashTable\* table = createHashTable(10);

insert(table, "key1", 1);

insert(table, "key2", 2);

printf("Value for 'key1': %d\n", search(table, "key1"));

printf("Value for 'key2': %d\n", search(table, "key2"));

return 0;

}

```

##### Heaps

- \*\*Definition\*\*: A specialized tree-based structure that satisfies the heap property.

- \*\*Example (Min-Heap)\*\*:

```c

#include <stdio.h>

#include <stdlib.h>

void swap(int\* x, int\* y) {

int temp = \*x;

\*x = \*y;

\*y = temp;

}

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

int smallest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] < arr[smallest])

smallest = left;

if (right < n && arr[right] < arr[smallest])

smallest = right;

if (smallest != i) {

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

heapify(arr, n, smallest);

}

}

void insert(int arr[], int\* n, int key) {

\*n += 1;

int i = \*n - 1;

arr[i] = key;

while (i != 0 && arr[(i - 1) / 2] > arr[i]) {

swap(&arr[i], &arr[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

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

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

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

printf("\n");

}

int main() {

int arr[10];

int n = 0;

insert(arr, &n, 3);

insert(arr, &n, 2);

insert(arr, &n, 1);

printArray(arr, n);

return 0;

}

```

**#### 6. Algorithm Analysis**

- \*\*Big O Notation\*\*: Explain the concept of algorithm complexity.

- \*\*Time and Space Complexity\*\*: Analyze the complexity of various data structure operations.

**#### 7. Conclusion**

- Recap key points.

- Highlight the importance of choosing the right data structure for a given problem.

This outline and content should help you create a comprehensive presentation on data structures with C. Feel free to adapt and expand upon this information as needed for your presentation.