**Introduction to Data Structures**

* **Definition**: Explanation of what data structures are.
* **Importance**: Why data structures are crucial in programming and problem-solving.
* **Types**: Classification into linear and non-linear data structures.

**1. Arrays**

* **Definition**: A collection of elements identified by index or key.
* **Properties**:
  + Fixed size
  + Homogeneous elements
  + Direct access to elements
* **Advantages**:
  + Easy to use and implement
  + Fast access time
* **Disadvantages**:
  + Fixed size
  + Inefficient insertions/deletions
* **Applications**:
  + Storing data in a linear form
  + Used in algorithms like sorting and searching

**Example Program**:

#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**

* **Definition**: A collection of nodes where each node contains data and a reference to the next node.
* **Types**: Singly, Doubly, Circular.
* **Properties**:
  + Dynamic size
  + Sequential access
* **Advantages**:
  + Dynamic size
  + Efficient insertions/deletions
* **Disadvantages**:
  + Sequential access (slower access time)
  + Extra memory for pointers
* **Applications**:
  + Implementing stacks, queues
  + Managing dynamic memory
* **Example Program**:

#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**

* **Definition**: A collection of elements with Last In First Out (LIFO) access.
* **Properties**:
  + LIFO principle
  + Dynamic size
* **Advantages**:
  + Simple implementation
  + Useful for reversing data, backtracking
* **Disadvantages**:
  + Limited access (only top element accessible)
* **Applications**:
  + Expression evaluation and conversion
  + Backtracking algorithms
* **Example Program**:

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

struct Stack {

int top;

int items[MAX];

};

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

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

printf("Stack overflow\n");

return;

}

s->items[++s->top] = item;

}

int pop(struct Stack\* s) {

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

printf("Stack underflow\n");

return -1;

}

return s->items[s->top--];

}

int main() {

struct Stack s;

s.top = -1;

push(&s, 1);

push(&s, 2);

printf("%d popped from stack\n", pop(&s));

return 0;

}

**4. Queues**

* **Definition**: A collection of elements with First In First Out (FIFO) access.
* **Types**: Simple Queue, Circular Queue, Priority Queue, Deque.
* **Properties**:
  + FIFO principle
  + Dynamic size
* **Advantages**:
  + Simple implementation
  + Useful for scheduling, buffering
* **Disadvantages**:
  + Limited access (only front and rear elements accessible)
* **Applications**:
  + Task scheduling
  + Buffering in data streams
* **Example Program**:

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

struct Queue {

int front, rear, size;

int items[MAX];

};

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

if (q->size == MAX) {

printf("Queue overflow\n");

return;

}

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

q->items[q->rear] = item;

q->size++;

}

int dequeue(struct Queue\* q) {

if (q->size == 0) {

printf("Queue underflow\n");

return -1;

}

int item = q->items[q->front];

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

q->size--;

return item;

}

int main() {

struct Queue q;

q.front = 0;

q.rear = -1;

q.size = 0;

enqueue(&q, 1);

enqueue(&q, 2);

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

return 0;

}

**5.Trees**

* **Definition**: A hierarchical structure with nodes connected by edges.
* **Types**: Binary Tree, Binary Search Tree, AVL Tree, Heap, B-Tree.
* **Properties**:
  + Hierarchical structure
  + Dynamic size
* **Advantages**:
  + Efficient searching, insertion, and deletion
  + Represents hierarchical data
* **Disadvantages**:
  + Complex implementation
  + Requires balancing (in some types)
* **Applications**:
  + Database indexing
  + Hierarchical data representation
* **Example Program (Binary Search Tree)**:

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left, \* 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;

}

**7. Graphs**

* **Definition**: A collection of vertices and edges.
* **Types**: Directed, Undirected, Weighted, Unweighted.
* **Properties**:
  + Non-linear structure
  + Dynamic size
* **Advantages**:
  + Models complex relationships
  + Flexible representation
* **Disadvantages**:
  + Complex implementation
  + Requires more memory
* **Applications**:
  + Network routing
  + Social networks
* **Example Program (Adjacency Matrix)**:

#include <stdio.h>

#define V 5

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, 0, 1},

{1, 0, 1, 0, 0},

{0, 1, 0, 1, 0},

{0, 0, 1, 0, 1},

{1, 0, 0, 1, 0}

};

printGraph(graph);

return 0;

}

**7. Heaps**

* **Definition**: A special tree-based data structure that satisfies the heap property.
* **Types**: Min-Heap, Max-Heap.
* **Properties**:
  + Complete binary tree
  + Heap property (min-heap or max-heap)
* **Advantages**:
  + Efficient priority queue operations
  + Useful for sorting
* **Disadvantages**:
  + Complex implementation
  + Requires rebalancing
* **Applications**:
  + Priority queues
  + Heap sort
* **Example Program (Max-Heap)**:

#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;

}

**8. Hash Tables**

* **Definition**: A data structure that stores key-value pairs for efficient lookup.
* **Properties**:
  + Uses hash functions
  + Handles collisions (chaining or open addressing)
* **Advantages**:
  + Fast lookups
  + Efficient space utilization
* **Disadvantages**:
  + Complex implementation
  + Poor performance with many collisions
* **Applications**:
  + Database indexing
  + Caching
* **Example Program**:

#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");

}

}

**9. Tries**

* **Definition**: A tree-like data structure used for storing a dynamic set of strings.
* **Properties**:
  + Nodes represent characters
  + Used for efficient searching
* **Advantages**:
  + Fast search time
  + Predictable performance
* **Disadvantages**:
  + Requires more memory
  + Complex implementation
* **Applications**:
  + Autocomplete
  + Spell checkers
* **Example Program**:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ALPHABET\_SIZE 26

struct TrieNode {

struct TrieNode\* children[ALPHABET\_SIZE];

int isEndOfWord;

};

struct TrieNode\* getNode() {

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

node->isEndOfWord = 0;

for (int i = 0; i < ALPHABET\_SIZE; i++) {

node->children[i] = NULL;

}

return node;

}

void insert(struct TrieNode\* root, const char\* key) {

struct TrieNode\* pCrawl = root;

for (int level = 0; level < strlen(key); level++) {

int index = key[level] - 'a';

if (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

pCrawl->isEndOfWord = 1;

}

int search(struct TrieNode\* root, const char\* key) {

struct TrieNode\* pCrawl = root;

for (int level = 0; level < strlen(key); level++) {

int index = key[level] - 'a';

if (!pCrawl->children[index])

return 0;

pCrawl = pCrawl->children[index];

}

return (pCrawl != NULL && pCrawl->isEndOfWord);

}

int main() {

char keys[][8] = {"the", "a", "there", "answer", "any", "by", "bye", "their"};

struct TrieNode\* root = getNode();

for (int i = 0; i < 8; i++) {

insert(root, keys[i]);

}

printf("%s --- %s\n", "the", search(root, "the") ? "Present" : "Not present");

printf("%s --- %s\n", "these", search(root, "these") ? "Present" : "Not present");

printf("%s --- %s\n", "their", search(root, "their") ? "Present" : "Not present");

printf("%s --- %s\n", "thaw", search(root, "thaw") ? "Present" : "Not present");

return 0;

}

**10. B-Trees**

* **Definition**: A self-balancing tree data structure that maintains sorted data and allows searches, insertions, deletions in logarithmic time.
* **Properties**:
  + Multi-way tree
  + Balanced tree
* **Advantages**:
  + Efficient for large data
  + Reduces disk read operations
* **Disadvantages**:
  + Complex implementation
  + Requires more memory
* **Applications**:
  + Databases
  + File systems
* **Example Program**:

Due to the complexity, a full B-Tree implementation is lengthy. Here's a simplified representation:

#include <stdio.h>

#include <stdlib.h>

#define T 3

struct BTreeNode {

int \*keys;

int t;

struct BTreeNode \*\*C;

int n;

int leaf;

};

struct BTreeNode \*createNode(int t, int leaf) {

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

node->t = t;

node->leaf = leaf;

node->keys = (int \*)malloc(2 \* t \* sizeof(int));

node->C = (struct BTreeNode \*\*)malloc(2 \* t \* sizeof(struct BTreeNode \*));

node->n = 0;

return node;

}

void traverse(struct BTreeNode \*root) {

if (root != NULL) {

for (int i = 0; i < root->n; i++) {

if (!root->leaf)

traverse(root->C[i]);

printf(" %d", root->keys[i]);

}

if (!root->leaf)

traverse(root->C[root->n]);

}

}

int main() {

struct BTreeNode \*root = createNode(T, 1);

// For a full implementation, add functions to insert, search, etc.

printf("B-Tree created with order %d\n", T);

traverse(root);

return 0;

}