**Assignment for Sorting and Data Structures:-**

1. **Given an unsorted array of integers, implement Selection Sort and return the sorted array.**

**Input: arr = [64, 25, 12, 22, 11]**

**Output: [11, 12, 22, 25, 64]**

#include <stdio.h>

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

for (int current = 0; current < n - 1; current++) {

int minIndex = current;

for (int next = current + 1; next < n; next++) {

if (arr[next] < arr[minIndex]) {

minIndex = next;

}

}

int temp = arr[current];

arr[current] = arr[minIndex];

arr[minIndex] = temp;

}

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

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

selectionSort(arr, n);

printf("Sorted array: ");

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

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

}

return 0;

}

1. **Write a function that uses Bubble Sort to sort an array of strings in alphabetical order.**

**Input: arr = ["banana", "apple", "cherry"]**

**Output: ["apple", "banana", "cherry"]**

#include <stdio.h>

#include <string.h>

void bubbleSort(char arr[][20], int n) {

for (int pass = 0; pass < n - 1; pass++) {

for (int current = 0; current < n - pass - 1; current++) {

if (strcmp(arr[current], arr[current + 1]) > 0) {

char temp[20];

strcpy(temp, arr[current]);

strcpy(arr[current], arr[current + 1]);

strcpy(arr[current + 1], temp);

}

}

}

}

int main() {

char arr[][20] = {"banana", "apple", "cherry"};

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

bubbleSort(arr, n);

printf("Sorted array: ");

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

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

}

return 0;

}

1. **Implement Quick Sort and return the number of swaps performed during the sorting process.**

**Input: arr = [10, 7, 8, 9, 1, 5]**

**Output: X (where X is the number of swaps)**

#include <stdio.h>

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

int pivot = arr[high];

int smallerIndex = low - 1;

for (int current = low; current < high; current++) {

if (arr[current] <= pivot) {

smallerIndex++;

int temp = arr[smallerIndex];

arr[smallerIndex] = arr[current];

arr[current] = temp;

}

}

int temp = arr[smallerIndex + 1];

arr[smallerIndex + 1] = arr[high];

arr[high] = temp;

return smallerIndex + 1;

}

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

if (low < high) {

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

quickSort(arr, low, pivotIndex - 1);

quickSort(arr, pivotIndex + 1, high);

}

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

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

quickSort(arr, 0, n - 1);

printf("Sorted array: ");

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

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

}

return 0;

}

1. **Given an array of integers, use Insertion Sort to sort the array in descending order and return it.**

**Input: arr = [12, 11, 13, 5, 6]**

**Output: [13, 12, 11, 6, 5]**

#include <stdio.h>

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

for (int current = 1; current < n; current++) {

int key = arr[current];

int prev = current - 1;

while (prev >= 0 && arr[prev] < key) {

arr[prev + 1] = arr[prev];

prev--;

}

arr[prev + 1] = key;

}

}

int main() {

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

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

insertionSortDescending(arr, n);

printf("Sorted array in descending order: ");

for (int i = n - 1; i >= 0; i--) {

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

}

return 0;

}

1. **Implement Merge Sort and count the number of inversions in the array (an inversion is a pair of indices such that i < j and arr[i] > arr[j]).**

**Input: arr = [2, 3, 8, 6, 1]**

**Output: 5 (the number of inversions)**

#include <stdio.h>

void merge(int arr[], int low, int mid, int high, int\* inversions) {

int n1 = mid - low + 1;

int n2 = high - mid;

int left[n1], right[n2];

for (int i = 0; i < n1; i++)

left[i] = arr[low + i];

for (int j = 0; j < n2; j++)

right[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = low;

while (i < n1 && j < n2) {

if (left[i] <= right[j]) {

arr[k] = left[i];

i++;

} else {

arr[k] = right[j];

j++;

(\*inversions) += n1 - i;

}

k++;

}

while (i < n1) {

arr[k] = left[i];

i++;

k++;

}

while (j < n2) {

arr[k] = right[j];

j++;

k++;

}

}

void mergeSort(int arr[], int low, int high, int\* inversions) {

if (low < high) {

int mid = low + (high - low) / 2;

mergeSort(arr, low, mid, inversions);

mergeSort(arr, mid + 1, high, inversions);

merge(arr, low, mid, high, inversions);

}

}

int main() {

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

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

int inversions = 0;

mergeSort(arr, 0, n - 1, &inversions);

printf("Inversions: %d\n", inversions);

return 0;

}

**SEARCHING ALGORITHMS**

1. **Linear Search:**

**Write a function that performs a Linear Search on an array of integers to find the index of a target value. Return -1 if not found.**

**Input: arr = [3, 5, 2, 4, 9], target = 4**

**Output: 3 (the index of the target)**

#include <stdio.h>

int linearSearch(int arr[], int n, int targetValue) {

for (int current = 0; current < n; current++) {

if (arr[current] == targetValue) {

return current;

}

}

return -1;

}

int main() {

int arr[] = {3, 5, 2, 4, 9};

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

int targetValue = 4;

int index = linearSearch(arr, n, targetValue);

printf("Index of %d: %d\n", targetValue, index);

return 0;

}

1. **Binary Search:**

**Given a sorted array and a target value, implement Binary Search and return the index of the target. Return -1 if not found.**

**Input: arr = [1, 2, 3, 4, 5], target = 3**

**Output: 2 (the index of the target)**

#include <stdio.h>

int binarySearch(int arr[], int n, int target) {

int low = 0, high = n - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == target)

return mid;

else if (arr[mid] < target)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main() {

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

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

int target = 3;

int index = binarySearch(arr, n, target);

printf("Index of %d: %d\n", target, index);

return 0;

}

**DATA STRUCTURES**

1. **Queue:**

**Implement a Queue using two stacks. Include methods for enqueue, dequeue, and check if the queue is empty.**

**Input: Perform a series of operations like enqueueing 1, 2, 3 and then dequeueing.**

**Output: After operations, return the elements in the queue.**

#include <stdio.h>  
#include <stdlib.h>  
  
#define MAX\_SIZE 100  
  
// Structure for a stack  
struct Stack {  
 int top;  
 int array[MAX\_SIZE];  
};  
  
// Function to initialize a stack  
void initialize(struct Stack\* stack) {  
 stack->top = -1;  
}  
  
// Function to check if a stack is empty  
int isEmpty(struct Stack\* stack) {  
 return stack->top == -1;  
}  
  
// Function to push an element onto a stack  
void push(struct Stack\* stack, int item) {  
 if (stack->top == MAX\_SIZE - 1) {  
 printf("Stack Overflow\n");  
 return;  
 }  
 stack->array[++stack->top] = item;  
}  
  
// Function to pop an element from a stack  
int pop(struct Stack\* stack) {  
 if (isEmpty(stack)) {  
 printf("Stack Underflow\n");  
 return -1;  
 }  
 return stack->array[stack->top--];  
}  
  
// Function to enqueue an element (push to stack1)  
void enqueue(struct Stack\* stack1, int x) {  
 push(stack1, x);  
}  
  
// Function to dequeue an element (pop from stack2, or transfer from stack1 to stack2 if needed)  
int dequeue(struct Stack\* stack1, struct Stack\* stack2) {  
 if (isEmpty(stack2)) {  
 while (!isEmpty(stack1)) {  
 push(stack2, pop(stack1));  
 }  
 }  
 if (isEmpty(stack2)) {  
 printf("Queue is empty\n");  
 return -1;  
 }  
 return pop(stack2);  
}  
  
int main() {  
 struct Stack stack1, stack2;  
 initialize(&stack1);  
 initialize(&stack2);  
  
 enqueue(&stack1, 1);  
 enqueue(&stack1, 2);  
 enqueue(&stack1, 3);  
  
 printf("%d\n", dequeue(&stack1, &stack2));  
 printf("%d\n", dequeue(&stack1, &stack2));  
 printf("%d\n", isEmpty(&stack1) && isEmpty(&stack2));  
 printf("%d\n", dequeue(&stack1, &stack2));  
 printf("%d\n", isEmpty(&stack1) && isEmpty(&stack2));  
 printf("%d\n", dequeue(&stack1, &stack2));  
  
 return 0;  
}