**DAY - 2 SORTING and SEARCHING**

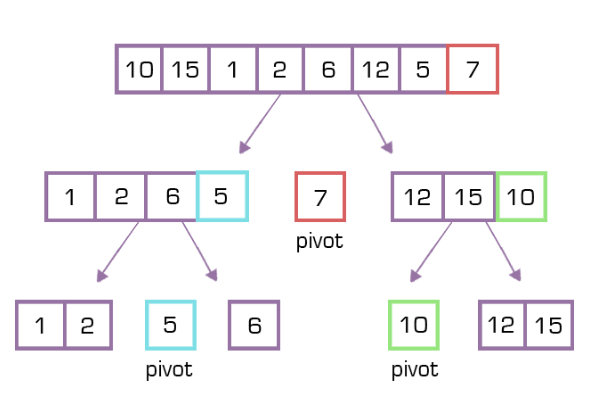
**Problem 1 - Explain how *QuickSort* works**

Solution - **Quick Sort** is a divide and conquer, comparison, in-place algorithm.

* Efficient implementations of Quick sort are **not a stable** sort.
* When implemented well, it can be about two or three times faster than its main competitors, merge sort and heap sort.
* **Quick sort** determines a  **pivot**, which is a somewhat arbitrary **(typically the *last (rightmost) element* is used as an estimate (guess) of the median).** element in the collection.
* Using the pivot point, quicksort partitions (or divides) the larger unsorted collection into two, smaller lists.

**Pseudo Code -**

1. Choose a pivot element: In QuickSort, we select a pivot element from the array. The pivot can be any element, but for simplicity, we often choose the last element in the array as the pivot.
2. Partitioning: Rearrange the elements in the array such that all elements smaller than the pivot are placed before it, and all elements greater than the pivot are placed after it. The pivot element will be in its final sorted position. This process is called partitioning.
3. Recursively sort sub-arrays: After partitioning, the array is divided into two sub-arrays. One sub-array consists of elements smaller than the pivot, and the other consists of elements greater than the pivot. Recursively apply the same steps to these sub-arrays.
4. Combine the sorted sub-arrays: After the recursive sorting, the sub-arrays are sorted individually. The final step is to combine these sub-arrays to get the fully sorted array.



#include <iostream>

using namespace std;

// Function to swap two elements

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partition function for QuickSort

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

int pivot = arr[high]; **// Choose the rightmost element as pivot**

int i = (low - 1); **// Index of smaller element**

for (int j = low; j <= high - 1; j++) {

**// If the current element is smaller than or equal to the pivot**

if (arr[j] <= pivot) {

i++; // Increment index of smaller element

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

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

**// Recursive function to implement QuickSort**

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

if (low < high) {

// pi is the partitioning index

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

// Sort the elements on the left of the partition and the right of the partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// **Function to print an array**

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

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

cout << arr[i] << " ";

}

cout << endl;

}

**// Test the QuickSort algorithm**

int main() {

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

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

cout << "Original array: ";

printArray(arr, n);

quickSort(arr, 0, n - 1);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

This implementation uses the Lomuto partition scheme, which chooses the rightmost element as the pivot. It recursively partitions the array around the pivot and sorts the resulting subarrays. The **swap** function is used to swap two elements in the array.

In the **main** function, we create an array, print the original array, sort it using the **quickSort** function, and finally print the sorted array.

When you run the program, the output will be:

Original array: 64 25 12 22 11

Sorted array: 11 12 22 25 64

**Time complexity** of Quick Sort is ***O*(*n log n*).**

In the worst case, it makes ***O*(*n*2)**.

**PROBLEM 2 – Search in a Sorted and Rotated Array using ubiquitous binary search.**

The Ubiquitous Binary Search is a technique that extends the traditional binary search algorithm to solve a variety of problems beyond searching for a specific element in a sorted array. It's called "ubiquitous" because it can be applied to different scenarios and is a powerful tool in algorithmic problem-solving.

However, the Ubiquitous Binary Search extends this concept by adapting the binary search approach to solve different problems efficiently.

Here are a few examples of how the Ubiquitous Binary Search can be used:

Finding the Lower Bound:

Finding the Upper Bound:

Searching in a Sorted and Rotated Array:

Bitonic Array Search: A bitonic array is an array that first increases and then decreases in terms of its elements. The Ubiquitous Binary Search can be utilized to find the maximum element in a bitonic array or search for a target element efficiently.

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**// Function to perform the search in a sorted and rotated array**

**int searchRotatedArray(vector<int>& nums, int target) {**

**int left = 0;**

**int right = nums.size() - 1;**

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

**// If the middle element is the target, return its index**

**if (nums[mid] == target) {**

**return mid;**

**}**

**// If the left half is sorted**

**if (nums[left] <= nums[mid]) {**

**// If the target is within the sorted left half**

**if (target >= nums[left] && target < nums[mid]) {**

**right = mid - 1; // Update the right pointer**

**} else {**

**left = mid + 1; // Update the left pointer**

**}**

**}**

**// If the right half is sorted**

**else {**

**// If the target is within the sorted right half**

**if (target > nums[mid] && target <= nums[right]) {**

**left = mid + 1; // Update the left pointer**

**} else {**

**right = mid - 1; // Update the right pointer**

**}**

**}**

**}**

**// If the target is not found, return -1**

**return -1;**

**}**

**int main() {**

**vector<int> nums = {4, 5, 6, 7, 0, 1, 2};**

**int target = 0;**

**int result = searchRotatedArray(nums, target);**

**if (result == -1) {**

**cout << "Target element not found." << endl;**

**} else {**

**cout << "Target element found at index " << result << "." << endl;**

**}**

**return 0;**

**}**

Explanation:

1. The searchRotatedArray function takes a reference to a vector nums and the target element target. It returns the index of the target element if found, or -1 otherwise.
2. We initialize the left and right pointers to the start and end of the array, respectively.
3. We use a while loop to perform the binary search until the left pointer is less than or equal to the right pointer.
4. Inside the loop, we calculate the middle index mid using the formula (left + right) / 2 to avoid integer overflow.
5. We check if the middle element nums[mid] is equal to the target. If so, we return the index mid.
6. Next, we check if the left half of the array is sorted by comparing nums[left] with nums[mid]. If it is, we further check if the target is within the range of the sorted left half. If yes, we update the right pointer to mid - 1 to search in the left half; otherwise, we update the left pointer to mid + 1.
7. If the left half is not sorted, it means the right half must be sorted. In this case, we check if the target is within the range of the sorted right half. If yes, we update the left pointer to mid + 1 to search in the right half; otherwise, we update the right pointer to mid - 1.
8. If the target is not found

**Problem 3 - Given an unsorted list or array, the problem is to find the kth smallest or largest element using Heap sort using C++ with explanation**

**Solution -**

Here's the step-by-step process to find the kth smallest element using Heap sort:

1. First, we need to build a binary min heap from the given array. A binary heap is a complete binary tree where each node is smaller than its children (in the case of a min heap). We can build the heap by repeatedly calling a function called heapify, which maintains the heap property by comparing the current node with its children and swapping if necessary.

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

{

int smallest = i; // Initialize the smallest as the root

int left = 2 \* i + 1; // Left child index

int right = 2 \* i + 2; // Right child index

// If left child is smaller than the root

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

smallest = left;

// If right child is smaller than the smallest so far

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

smallest = right;

// If the smallest is not the root

if (smallest != i) {

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

heapify(arr, n, smallest);

}

}

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

// Build the heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

}

1. **After building the heap, the smallest element will be at the root of the heap (index 0). We can remove the smallest element k-1 times from the heap to find the kth smallest element.**

int findKthSmallest(int arr[], int n, int k)

{

buildHeap(arr, n);

// Remove the smallest element k-1 times

for (int i = 0; i < k - 1; i++) {

// Replace the root with the last element

swap(arr[0], arr[n - 1]);

n--; // Reduce the heap size

heapify(arr, n, 0); // Heapify the root node

}

// Return the kth smallest element

return arr[0];

}

1. **Finally, we can call the findKthSmallest function with the unsorted array and the value of k to find the kth smallest element. For example:**

int main() {

int arr[] = {7, 10, 4, 3, 20, 15};

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

int k = 3;

int kthSmallest = findKthSmallest(arr, n, k);

cout << "The " << k << "th smallest element is: " << kthSmallest << endl;

return 0;

}

**Output:**

|  |
| --- |
| **The 3rd smallest element is: 7** |
|  |

**Problem 4 - Sort a linked list using Insertion Sort**

**Solution**

#include <iostream>

// Definition of a node in the linked list

struct Node {

int data;

Node\* next;

};

// Function to insert a new node at the beginning of the linked list

void insert(Node\*\* head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = (\*head);

(\*head) = newNode;

}

// Function to display the linked list

void displayList(Node\* head) {

while (head != nullptr) {

std::cout << head->data << " ";

head = head->next;

}

std::cout << std::endl;

}

// Function to perform insertion sort on a linked list

void insertionSort(Node\*\* head) {

Node\* sorted = nullptr; // Initialize sorted linked list

Node\* current = \*head; // Traverse the given linked list

while (current != nullptr) {

Node\* nextNode = current->next;

// Find the insertion point in the sorted list

Node\* temp = sorted;

if (temp == nullptr || temp->data > current->data) {

current->next = sorted;

sorted = current;

} else {

while (temp->next != nullptr && temp->next->data < current->data) {

temp = temp->next;

}

current->next = temp->next;

temp->next = current;

}

current = nextNode;

}

\*head = sorted; // Update the head to point to the sorted list

}

int main() {

Node\* head = nullptr;

// Example usage

insert(&head, 5);

insert(&head, 10);

insert(&head, -3);

insert(&head, 8);

insert(&head, 1);

std::cout << "Linked list before sorting: ";

displayList(head);

insertionSort(&head);

std::cout << "Linked list after sorting: ";

displayList(head);

return 0;

}

Explanation of the steps:

1. The linked list is represented using a struct called **Node**, which contains an integer **data** and a pointer **next** to the next node in the list.
2. The **insert** function is used to insert a new node at the beginning of the linked list. It takes a pointer to the head of the list and the data for the new node.
3. The **displayList** function is used to print the elements of the linked list.
4. The **insertionSort** function performs the insertion sort algorithm on the linked list. It takes a pointer to the head of the list.
5. In the **insertionSort** function, a new empty linked list called **sorted** is initialized.
6. The current node is initially set to the head of the list.
7. We traverse the given linked list by moving the **current** pointer.
8. For each node, we detach it from the original list by assigning **nextNode** as the next node after the current node.
9. We find the appropriate position to insert the current node in the **sorted** list.
10. If the **sorted** list is empty or the first node has a greater value than the current node, we insert the current node at the beginning of the **sorted** list.
11. Otherwise, we traverse the **sorted** list to find the insertion point. We stop when we reach the end of the **sorted** list or find a node with a greater value than

**Problem 5 – To demonstrate the superiority of interpolation search over binary search, let's consider an example where we have a sorted array of integers and we want to search for a specific element. We will compare the performance of interpolation search and binary search on this array.**

Interpolation search is an improved searching algorithm over binary search, especially when the data is uniformly distributed. It performs better in scenarios where the data is sorted and evenly distributed.

Interpolation Search:

1. Interpolation search is an improved search algorithm over binary search, especially when the data is uniformly distributed.
2. It works by estimating the position of the target value in the sorted array based on the values of the first and last elements.
3. Interpolation search calculates the probable position of the target by using a formula that takes into account the range of values and their distribution in the array.
4. It then narrows down the search range by comparing the estimated position with the target value.
5. If the estimated value matches the target, the search is successful.
6. Otherwise, it adjusts the search range based on whether the estimated value is greater or smaller than the target.
7. The process continues until the target value is found or the search range becomes empty.

Binary Search:

1. Binary search is a commonly used search algorithm for sorted arrays.
2. It works by repeatedly dividing the search range in half.
3. It compares the middle element of the current range with the target value.
4. If the middle element matches the target, the search is successful.
5. If the middle element is greater than the target, the search continues in the left half of the current range.
6. If the middle element is smaller than the target, the search continues in the right half of the current range.
7. The process repeats until the target value is found or the search range becomes empty.

Now, let's compare the performance of interpolation search and binary search.

Performance Comparison:

1. Interpolation search generally performs better than binary search when the data is uniformly distributed because it estimates the position of the target more accurately.
2. In such cases, interpolation search can converge on the target position more quickly, resulting in faster search times.
3. However, if the data is not uniformly distributed or contains repeated elements, interpolation search may perform worse than binary search.
4. Binary search always takes a constant amount of time to divide the search range in half, making it more predictable in terms of performance.
5. On the other hand, interpolation search's performance depends on the distribution of the data, which can lead to variations in search times.

In conclusion, interpolation search can provide superior performance over binary search when the data is uniformly distributed. However, it's important to consider the characteristics of the data and choose the appropriate search algorithm accordingly.

#include <iostream>

using namespace std;

// Interpolation Search

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

int low = 0;

int high = n - 1;

while (low <= high && target >= arr[low] && target <= arr[high]) {

if (low == high) {

if (arr[low] == target)

return low;

return -1;

}

// Calculating the position to probe

int pos = low + ((target - arr[low]) \* (high - low)) / (arr[high] - arr[low]);

if (arr[pos] == target)

return pos;

if (arr[pos] < target)

low = pos + 1;

else

high = pos - 1;

}

return -1;

}

// Binary Search

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

int low = 0;

int high = n - 1;

while (low <= high) {

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

if (arr[mid] == target)

return mid;

if (arr[mid] < target)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

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

int target = 7;

// Interpolation Search

int interpolationIndex = interpolationSearch(arr, n, target);

if (interpolationIndex != -1)

cout << "Element found at index " << interpolationIndex << " using Interpolation Search.\n";

else

cout << "Element not found using Interpolation Search.\n";

// Binary Search

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

if (binaryIndex != -1)

cout << "Element found at index " << binaryIndex << " using Binary Search.\n";

else

cout << "Element not found using Binary Search.\n";

return 0;

}

Explanation of the steps:

1. The **interpolationSearch** function performs interpolation search on the sorted array. It takes three parameters: the array **arr**, the size of the array **n**, and the target element to search for **target**.
2. In the **interpolationSearch** function, we initialize **low** as the starting index of the array (0) and **high** as the ending index of the array (n - 1).
3. The **while** loop continues until **low** is less than or equal to **high** and the target element is within the range of the subarray defined by **low** and **high**.
4. Inside the **while** loop, we calculate the position to probe using interpolation formula:

|  |
| --- |
| int pos = low + ((target - arr[low]) \* (high - low)) / (arr[high] - arr[low]); |

1. If the value at the calculated position **pos** is equal to the target element, we return the index **pos** as the element has been found.
2. If the value at **pos** is less than the target element, we update **low** to **pos + 1** to search

**Problem 6** - Develop a sorting algorithm that can handle arrays of floating-point numbers and sorts them in descending order.

SOLUTION -

One commonly used sorting algorithm that can handle arrays of floating-point numbers and sort them in descending order is the Bubble Sort algorithm. Here's an implementation in C++:

#include <iostream>

void bubbleSort(float arr[], int size) {

for (int i = 0; i < size - 1; ++i) {

for (int j = 0; j < size - i - 1; ++j) {

if (arr[j] < arr[j + 1]) {

// Swap arr[j] and arr[j + 1]

float temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

}

int main() {

float arr[] = {4.5, 2.1, 9.8, 1.6, 5.3};

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

bubbleSort(arr, size);

std::cout << "Sorted array in descending order: ";

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

std::cout << arr[i] << " ";

}

std::cout << std::endl;

return 0;

}

Explanation of the steps in the Bubble Sort algorithm:

1. The **bubbleSort** function takes an array **arr** and its size as parameters.
2. The outer loop iterates **size - 1** times because after each iteration, the largest element is placed at its correct position, so we don't need to compare it again.
3. The inner loop compares adjacent elements and swaps them if they are in the wrong order. In this case, we compare **arr[j]** with **arr[j + 1]**.
4. If the elements are out of order, we swap them by using a temporary variable **temp**.
5. After each iteration of the inner loop, the largest element "bubbles" to the end of the array.
6. Finally, in the **main** function, we create an example array **arr** and obtain its size.
7. We call the **bubbleSort** function to sort the array in descending order.
8. Finally, we print the sorted array using a loop.

In this implementation, the Bubble Sort algorithm is used to sort the array in descending order. It repeatedly compares adjacent elements and swaps them if they are in the wrong order, resulting in the largest elements "bubbling" to the top. The algorithm continues this process until the entire array is sorted.