

**Symbiosis Institute of Technology**

**Faculty of Engineering**

**CSE- Academic Year 2023-24**

**Data Structures – Lab Batch 2022-26**

|  |  |
| --- | --- |
|  | **Lab Assignment No:- 1,2,3** |
|  |  |
| **Name of Student** | Kshitij Gurbuxani |
| **PRN No.** | 22070122097 |
| **Batch** | 22-26 |
| **Class** | CS B |
| **Academic Year &**  **Semester** | 23-24 Semester 1 |
| **Date of Submission** | 28-08-2023 |
|  |  |
| **Title of Assignment:** | 1. Implement following searching algorithm: Linear search with multiple occurrences 2. Implement following searching algorithms in menu:    1. Binary search with iteration    2. Binary search with recursion |
| **Theory:** | 1. Prepare table for following searching and sorting algorithms for their best case, average case and worst case time complexities. Linear search, binary search, bubble sort, Insertion sort, selection sort, merge sort, quick sort.  | **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | | --- | --- | --- | --- | | Linear Search | O(1) | O(n) | O(n) | | Binary Search | O(1) | O(log n) | O(log n) | | Bubble Sort | O(n) | O(n^2) | O(n^2) | | Insertion Sort | O(n) | O(n^2) | O(n^2) | | Selection Sort | O(n^2) | O(n^2) | O(n^2) | | Merge Sort | O(n log n) | O(n log n) | O(n log n) | | Quick Sort | O(n log n) | O(n log n) | O(n^2) |  1. Discuss on Best case and Worst case time complexities of Linear search, binary search, bubble sort, Insertion sort, selection sort, merge sort, quick sort.   **Linear Search:**  Best Case: O(1)  The best case occurs when the element being searched for is found in the very first comparison.  Worst Case: O(n)  The worst case happens when the element being searched for is either at the end of the list, or it's not present at all. In such cases, you need to examine every element.  **Binary Search:**  Best Case: O(1)  The best case occurs when the element being searched for is found at the middle of the sorted array.  Worst Case: O(log n)  The worst case happens when the element is not present, or it's at one of the extreme ends. In each comparison, the search space is reduced by half.  **Bubble Sort:**  Best Case: O(n)  The best case occurs when the input array is already sorted. In this case, the algorithm performs only one pass through the array to confirm that no swaps were needed.  Worst Case: O(n^2)  The worst case happens when the input array is sorted in reverse order. In each pass, it needs to swap elements throughout the entire array.  **Insertion Sort:**  Best Case: O(n)  The best case occurs when the input array is already sorted. In this case, the algorithm can simply iterate through the array without making any swaps.  Worst Case: O(n^2)  The worst case occurs when the input array is sorted in reverse order. In each step, it needs to compare and shift elements, leading to a quadratic time complexity.  **Selection Sort:**  Best Case: O(n^2)  The best case is the same as the worst case for selection sort, as it always needs to search for the minimum element, even if the array is partially sorted.  Worst Case: O(n^2)  The worst case occurs when the input array is sorted in reverse order. In each pass, it needs to search for the minimum element and swap it with the first unsorted element.  **Merge Sort:**  Best Case: O(n log n)  The best case is the same as the average and worst cases for merge sort because it always divides the array into halves and then merges them. It consistently maintains a time complexity of O(n log n).  **Quick Sort:**  Best Case: O(n log n)  The best case occurs when the pivot choice consistently divides the list into nearly equal halves.  Worst Case: O(n^2)  The worst case occurs when the pivot choice is poor and consistently divides the list into highly imbalanced sublists. In this case, it degrades to a quadratic time complexity. |
| **Source**  **Code/Algorithm/**  **Flow Chart:** | **Code for program A (Linear Search with multiple occurrences):** |

|  |  |
| --- | --- |
|  | **Output for Program A:**  **Code for Program B ( Binary Search Menu Driven) :** |
| **Output Screenshots (if applicable)** | **Output for Program B:**    **Code for Bubble Sort Iterative:**    **Output:**    **Code for Bubble Sort Recursive:**    **Output:**    **Code for Iterative Selection Sort:**    **Output:**      **Code for Recursive Selection Sort:**    **Output:**    **Code for Iterative Insertion Sort:**  **Output for above program:**    **Code for Recursive Insertion Sort:**    **Output for above program:**    **Code for Iterative Merge Sort:**  #include <stdio.h>  void merge(int array[], int start, int middle, int end)  {      int a, b, c;      int n1 = middle - start + 1;      int n2 = end - middle;      // Create temporary arrays to store the values after splitting occurs.      int LEFT[n1], RIGHT[n2];      for (a = 0; a < n1; a++)      {          LEFT[a] = array[start + a];      }      for (b = 0; b < n2; b++) {          RIGHT[b] = array[middle + 1 + b];      }      // Merge the temporary array back into one array.      a = 0; // Index of first array that is created.      b = 0; // Index of second array that is created.      c = start; // Index of merged array when recombination occurs      while (a < n1 && b < n2)      {          if (LEFT[a] <= RIGHT[b])          {              array[c] = LEFT[a];              a++;          }          else          {              array[c] = RIGHT[b];              b++;          }          c++;      }      // Copy the remaining elements of LEFT[], if any      while (a < n1)      {          array[c] = LEFT[a];          a++;          c++;      }      // Copy the remaining elements of RIGHT[], if any      while (b < n2)      {          array[c] = RIGHT[b];          b++;          c++;      }  }  void mergeSortIter(int array[], int num)  {      int currentSize; // Current size of arrays that will be merged.      int start; // Starting index of left array that will be merged.      // Merge the distributed arrays from bottom to top.      for (currentSize = 1; currentSize <= num - 1; currentSize = 2 \* currentSize)      {          for (start = 0; start < num - 1; start += 2 \* currentSize)          {              int middle = start + currentSize - 1;              int end = (start + 2 \* currentSize - 1 < num - 1) ? start + 2 \* currentSize - 1 : num - 1;              merge(array, start, middle, end);          }      }  }  int main()  {      int array[] = {12, 11, 13, 5, 6, 7};      int num = sizeof(array) / sizeof(array[0]);      mergeSortIter(array, num);      printf("Sorted array after merge sort technique is implemented: ");      for (int a = 0; a < num; a++)      {          printf("%d ", array[a]);      }      return 0;  }  **Output for above code:**    **Recursive Merge sort:**  #include <stdio.h>  void merge(int array[], int start, int middle, int end)  {      int a, b, c;      int n1 = middle - start+ 1;      int n2 = end - middle;      // Create temporary arrays to store the values after splitting takes place.      int LEFT[n1], RIGHT[n2];      // Copy data to temporary arrays LEFT[] and RIGHT[]      for (a = 0; a < n1; a++)      {          LEFT[a] = array[start+ a];      }      for (b = 0; b < n2; b++)      {          RIGHT[b] = array[middle + 1 + b];      }      // Merge the temporary arrays back into the main array.      a = 0; // Index of first array which is created.      b = 0; // Index of second array which is created.      c = start; // Index of merged array after recombination takes place.      while (a < n1 && b < n2)      {          if (LEFT[a] <= RIGHT[b])          {              array[c] = LEFT[a];              a++;          }          else          {              array[c] = RIGHT[b];              b++;          }          c++;      }      // Copy the remaining elements of LEFT[], if any are leftover      while (a < n1)      {          array[c] = LEFT[a];          a++;          c++;      }      // Copy the remaining elements of RIGHT[], if any are left over      while (b < n2)      {          array[c] = RIGHT[b];          b++;          c++;      }  }  void recursiveMergeSort(int array[], int start, int end)  {      if (start< end)      {          int middle = start+ (end - start) / 2;          // Recursive sorting of left and right arrays          recursiveMergeSort(array, start, middle);          recursiveMergeSort(array, middle + 1, end);          // Calling the merge function to merge the sorted subarrays          merge(array, start, middle, end);      }  }  int main()  {      int array[] = {12, 11, 13, 5, 6, 7};      int n = sizeof(array) / sizeof(array[0]);      recursiveMergeSort(array, 0, n - 1);      printf("Sorted array: ");      for (int a = 0; a < n; a++)      {          printf("%d ", array[a]);      }      return 0;  }  **Output for above code:**      **Code for Iterative and recursive Quick Sort:**  #include <stdio.h>  void printArray(int \*A, int n)  {      for (int i = 0; i < n; i++)      {          printf("%d ", A[i]);      }      printf("\n");  }  int partition(int A[], int low, int high)  {      int pivot = A[low];      int i = low + 1;      int j = high;      int temp;      do      {          while (A[i] <= pivot)          {              i++;          }          while (A[j] > pivot)          {              j--;          }          if (i < j)          {              temp = A[i];              A[i] = A[j];              A[j] = temp;          }      }      while (i < j);      // Swap A[low] and A[j]      temp = A[low];      A[low] = A[j];      A[j] = temp;      return j;  }  void quickSort(int A[], int low, int high)  {      int partitionIndex; // Index of pivot after partition      if (low < high)      {          partitionIndex = partition(A, low, high);          quickSort(A, low, partitionIndex - 1);  // sort left subarray          quickSort(A, partitionIndex + 1, high); // sort right subarray      }  }  int main()  {      int A[] = {9, 4, 4, 8, 7, 5, 6};      int n = 9;      n =7;      printArray(A, n);      quickSort(A, 0, n - 1);      printArray(A, n);      return 0;  }  **Output for above program:** |
| **Conclusion** | Thus we have studied different sorting algorithms and their time complexities. |