MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

BACHELOR OF COMPUTER APPLICATION (BCA)

CSB-1112 (Problem Solving Using C)

Block 4: Data Structures and Basic Algorithms in C

Unit 15: Searching and Sorting Algorithms

Structure	
15.0	Introduction
15.1	Objectives
15.2	Searching Algorithms
15.3	Sorting
15.4	Summary
15.5	Questions
15.6	References

15.0 Introduction

In the realm of computer science, efficient data management is crucial, and two of the most fundamental operations for achieving this are searching and sorting. Searching algorithms allow us to locate specific elements within a dataset, which is vital for applications ranging from simple data retrieval to complex database queries. Sorting algorithms, on the other hand, arrange data in a particular order, significantly improving the performance of other operations like searching, merging, and data presentation. Understanding these algorithms is essential for optimizing the performance of software applications and enhancing their overall efficiency.

Searching algorithms can be broadly classified into two categories: linear searching and binary searching. Linear searching involves scanning each element of the dataset sequentially until the desired element is found, making it straightforward but inefficient for large datasets. Binary searching, however, requires the dataset to be sorted and operates by repeatedly dividing the search interval in half, which makes it significantly faster than linear searching for large datasets. Both methods have their unique use cases and understanding their implementation and performance characteristics is fundamental for effective data handling.

Sorting algorithms come in various forms, each with its own advantages and trade-offs. Simple algorithms like Bubble Sort and Selection Sort are easy to understand and implement but are not suitable for large datasets due to their high time complexity. More advanced algorithms like Quick Sort and Merge Sort offer better performance with average-case time complexities of O(n log n). Heap Sort combines the advantages of both with consistent performance. Mastering these algorithms not only involves learning their mechanics but also understanding when to use each one based on the specific requirements of the application.

15.1 Objectives

By the end of this chapter, readers will be able to:

- Understand Linear and Binary Search
- Implement Basic Sorting Algorithms
- Master Advanced Sorting Algorithms
- Analyze Algorithm Performance
- Choose Appropriate Algorithms

15.2 Searching Algorithms

Searching is a method of locating desired data items that have been stored in a data structure. In addition to hash tables and search trees, data structures can also be arrays, linked lists, and other types of storage. It is common for the right search algorithm to be determined by the data structure being searched.

Depending on how they conduct searches, search algorithms can be categorized.

These are

- Linear searching
- Binary searching

15.2.1 Linear searching

Linear searching the most natural way to search is via Linear Search, which is straightforward but occasionally has bad speed. Using this strategy, the search starts by looking through each item in the list until the needed record is located. The list's components can be arranged in any sequence. i.e., separated or not.

Comparing the list's first element with the target element is how we start our search. The element's position is returned and the search is terminated if it matches. If not, we will compare and go on to the next part. The target element is thus compared to every element until a match is made. When the match fails and there are no more elements to compare, we return position -1, indicating that the target element is not present in the list.

Time Complexity: O(n) - where n is the number of elements in the list.

Best Case: O(1) - The target value is at the first position.

Worst Case: O(n) - The target value is at the last position or not present at all.

Usage: Useful for small lists or unsorted lists where performance is not critical.

For example consider the following list of elements. 55 95 75 85 11 25 65 45, Suppose we want to search for element 11(i.e. Target element = 11). We first compare the target element with first element in list i.e. 55. Since both are not matching we move on the next elements in the list and compare. Finally we will find the match after 5 comparisons at position 4 starting from position 0. Linear search can be implemented in two ways. i) Non recursive ii) Recursive

Algorithm for Linear search

Linear_Search (A[ ], N, val , pos)

Step 1 : Set pos = -1 and k = 0

Step 2 : Repeat while k < N

Begin

Step 3 : if A[k] = val

Set pos = k

print pos

Goto step 5

End while

Step 4 : print “Value is not present”

Step 5 : Exit

Non recursive C program for Linear search

```
#include <stdio.h>
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i; // Return the index where the target is found
        }
    }
    return -1; // Return -1 if the target is not found
}
int main() {
    int arr[] = {2, 4, 6, 8, 10, 12};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 8;

    int result = linearSearch(arr, n, target);

    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }

    return 0;
}
```

15.2.2 Binary Searching

With a run-time complexity of O (log n), binary search is a quick search algorithm. The divide and conquer strategy is used by this search algorithm. Binary search compares the collection's middle item to find a specific item. The item's index is returned if a match is found. The item is searched in the sub-array to the left of the middle item if the middle item exceeds the item. If not, the item is looked for in the sub-array to the middle item's right. Until the subarray's size drops to zero, this operation is carried out on it as well.

Sorting the list of objects in either ascending or descending order is necessary before using binary searching.

- **Time Complexity:** O(log n) - where n is the number of elements in the list.
- **Best Case:** O(1) - The target value is at the middle of the interval.
- **Worst Case:** O(log n) - The target value is at one of the extreme ends or not present.
- **Usage:** Suitable for large sorted lists due to its logarithmic complexity.

Algorithm:

Binary_Search (A [ ], U_bound, VAL)

Step 1 : set BEG = 0 , END = U_bound , POS = -1

Step 2 : Repeat while (BEG <= END)

Step 3 : set MID = (BEG + END) / 2

Step 4 : if A [MID] == VAL then

POS = MID

print VAL " is available at ", POS

GoTo Step 6

End if

if A [MID] > VAL then

set END = MID – 1

Else

set BEG = MID + 1

End if

End while

Step 5 : if POS = -1 then

print VAL " is not present "

End if

Step 6 : EXIT

Non recursive C program for binary search

```
#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; // Calculate the mid index
        if (arr[mid] == target) {
            return mid; // Return the index if target is found
        } else if (arr[mid] < target) {
            low = mid + 1; // If target is greater, search the right half
        } else {
            high = mid - 1; // If target is smaller, search the left half
        }
    }
    return -1; // Return -1 if target is not found
}
int main() {
    int arr[] = {2, 4, 6, 8, 10, 12};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 8;
    int result = binarySearch(arr, n, target);
    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }
    return 0;
}
```

15.3 SORTING

Arranging the elements in a list either in ascending or descending order. Various sorting algorithms are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort

15.3.1 Bubble sort

One type of exchange sort is the bubble sort. This method involves critical element shifting and frequent comparisons between the elements. An often used sorting algorithm is bubble sort. It is simple to comprehend but requires a lot of time—more comparisons are needed to sort a list. This kind involves switching and comparing two sequential parts. As a result, every element in the array is examined one by one. It's not the same as the selected sort. When two records are discovered to be out of order, they are quickly switched rather than first applying swapping and then searching for the minimum element.

- **Time Complexity:** $O(n^2)$ - where n is the number of elements in the list.
- **Best Case:** O(n) - The list is already sorted.
- **Worst Case:** $O(n^2)$ - The list is sorted in reverse order.
- **Usage:** Simple to implement but inefficient for large lists.

ALGORITHM:

Bubble_Sort (A [ ] , N)

Step 1: Start

Step 2: Take an array of n elements

Step 3: for i=0,.....n-2

Step 4: for j=i+1,…….n-1

Step 5: if arr[j]>arr[j+1] then

Interchange arr[j] and arr[j+1]

End of if

Step 6: Print the sorted array arr

Step 7:Stop

Program for Bubble sort

```
#include <stdio.h>
void bubbleSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            // Swap arr[j] and arr[j + 1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}
int main() {
```

```
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Array before sorting: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    bubbleSort(arr, n);
    printf("\nArray after sorting: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

15.3.2 Selection sort (Select the smallest and Exchange)

The first item is ranked first after being compared to the other n-1 items. Whichever item is lowest overall is placed first.Following that, the second item on the list is taken and compared to the remaining (n-2) items; if an item on the (n-2) items has a value lower than the second item's, it is exchanged (swapped) with the second item on the list, and so on.

- **Time Complexity:** $O(n^2)$ - where n is the number of elements in the list.
- **Best Case:** $O(n^2)$ - Due to the lack of a break condition in the algorithm.
- **Worst Case:** $O(n^2)$ - Every element needs to be compared with every other element.
- **Usage:** Useful for small lists or when memory usage is a concern.

Algorithm:

```
Selection_Sort ( A [ ] , N )
Step 1 :start
Step 2: Repeat For K = 0 to N – 2
Begin
Step 3: Set POS = K
Step 4
: Repeat for J = K + 1 to N – 1
Begin
If A[ J ] < A [ POS ]
Set POS = J
End For
Step 5: Swap A [ K ] with A [ POS ]
```

End For

Step 6: stop

Selection sort Program

```
#include <stdio.h>
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    // One by one move boundary of unsorted subarray
    for (i = 0; i < n - 1; i++) {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Array before sorting: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    selectionSort(arr, n);
    printf("\nArray after sorting: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
```

```
}
```

15.3.3 Insertion Sort

Insertion sort: It builds a sorted output list by repeatedly eating one input element. Insertion sort takes one element out of the input data for each iteration, locates its proper place in the sorted list, and inserts it there. It continues until there are no more input components.

- **Time Complexity:** $O(n^2)$ - where n is the number of elements in the list.
- **Best Case:** O(n) - The list is already sorted.
- **Worst Case:** $O(n^2)$ - The list is sorted in reverse order.
- **Usage:** Efficient for small or nearly sorted lists.

ALGORITHM:

Step 1: start

Step 2: for i ← 1 to length(A)

Step 3: j ← i

Step 4: while j > 0 and A[j-1] > A[j]

Step 5: swap A[j] and A[j-1]

Step 6: j ← j - 1

Step 7: end while

Step 8: end for

Step9: stop

Insertion Sort Programming

```
#include <stdio.h>
void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
```

```
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Array before sorting: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    insertionSort(arr, n);
    printf("\nArray after sorting: \n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

15.3.4 Quick Sort

Divide and conquer is the method used in Quick Sort. Tony Hoare developed it in 1959. The low elements and the high elements are the first two smaller sub-arrays created by quick sort.

Recursively sorting the sub-arrays is thus possible with Quick sort.

- **Time Complexity:** O(n log n) on average.
- **Best Case:** O(n log n) - The pivot element divides the list into two equal halves.
- **Worst Case:** O(n^2) - The pivot element is the smallest or largest, resulting in unbalanced partitions.
- **Usage:** Highly efficient for large lists but not stable. Performance can be improved with a good pivot selection strategy.

ALGORITHM:

Step 1: Pick an element, called a pivot, from the array.

Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Quick Sort Program

```
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
```

```
    *a = *b;
    *b = temp;
}
// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Select the rightmost element as pivot
    int i = (low - 1);      // Index of smaller element
    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than or equal to 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);
}
// Function to implement quicksort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index, arr[pi] is now at right place
        int pi = partition(arr, low, high);
        // Separately sort elements before partition and after 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++)
        printf("%d ", arr[i]);
    printf("\n");
}
// Driver program to test above functions
```

```
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

15.3.5 Merge Sort

Based on the divide and conquer strategy, merge sort is a sorting method. Merge sort divides the unsorted list into N sublists, each containing one element, since a list with just one element is regarded as sorted. After that, it continuously merges these sublists to create new sorted sublists until finally just one sorted list is created. Merge Sort has an O(n log n) time complexity, which makes it very quick.

- **Time Complexity:** O(n log n) - where n is the number of elements in the list.
- **Best Case:** O(n log n) - Consistent performance.
- **Worst Case:** O(n log n) - Consistent performance.
- **Usage:** Suitable for large lists, especially when stable sorting is required.

Algorithm

Conceptually, merge sort works as follows:

1. Divide the unsorted list into two sub lists of about half the size.
2. Divide each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned.
3. Merge the two sub lists back into one sorted list.

Merge Sort Program

```
#include <stdio.h>
// Function to merge two subarrays of arr[]
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    // Create temporary arrays
    int L[n1], R[n2];
```

```
    // Copy data to temporary arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    // Merge the temporary arrays back into arr[l..r]
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Main function that sorts arr[l..r] using merge()
void mergeSort(int arr[], int l, int r) {
```

```
    if (l < r) {
        // Find the middle point
        int m = l + (r - l) / 2;
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
// Function to print an array
void printArray(int A[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}
// Driver code
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    printf("Given array is \n");
    printArray(arr, arr_size);
    mergeSort(arr, 0, arr_size - 1);
    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}
```

15.3.6 Heap sort

With the characteristic that a parent is always greater than or equal to either of its offspring (if they exist), it is a fully binary tree. Priority queue is used to sort the heap once it has been first produced (max or min) using a binary tree.

- **Time Complexity:** O(n log n) - where n is the number of elements in the list.
- **Best Case:** O(n log n) - Consistent performance.
- **Worst Case:** O(n log n) - Consistent performance.
- **Usage:** Efficient and stable but requires additional space for the heap structure.

Algorithms

- Start with just one element. One element will always satisfy heap property.
- Insert next elements and make this heap.
- Repeat step b, until all elements are included in the heap.

Steps of Sorting:

- Exchange the root and last element in the heap.
- Make this heap again, but this time do not include the last node.
- Repeat steps a and b until there is no element left.

Heap Sort Program

```
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
// Function to heapify a subtree rooted with node i which is an index in arr[]
void heapify(int arr[], int n, int i) {
   int largest = i;  // Initialize largest as root
   int left = 2 * i + 1;  // Left child
   int right = 2 * i + 2; // Right child
   // If left child is larger than root
   if (left < n && arr[left] > arr[largest])
      largest = left;
   // If right child is larger than largest so far
   if (right < n && arr[right] > arr[largest])
      largest = right;
   // If largest is not root
   if (largest != i) {
      swap(&arr[i], &arr[largest]);
      // Recursively heapify the affected sub-tree
      heapify(arr, n, largest);
   }
}
```

```
// Main function to do heap sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(&arr[0], &arr[i]);
        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}
// Driver code
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array:\n");
    printArray(arr, n);
    heapSort(arr, n);
    printf("Sorted array:\n");
    printArray(arr, n);
    return 0;
}
```

15.4 Summary

In this chapter, we explored the fundamental concepts of searching and sorting algorithms, essential tools for efficient data management in computer science. We began with searching algorithms, examining both linear and binary search methods. Linear search involves scanning each element in a dataset sequentially,

making it straightforward but inefficient for large datasets. Binary search, which requires the dataset to be sorted, operates by repeatedly dividing the search interval in half, offering a much faster alternative for large datasets.

Next, we delved into sorting algorithms, starting with basic techniques such as Bubble Sort, Selection Sort, and Insertion Sort. These algorithms are simple to understand and implement but are generally inefficient for large datasets due to their higher time complexities. We then moved on to more advanced sorting algorithms like Quick Sort, Merge Sort, and Heap Sort. Quick Sort uses a divide-and-conquer approach, Merge Sort employs a recursive merging process, and Heap Sort utilizes a heap data structure, all providing more efficient sorting for larger datasets.

Throughout the chapter, we emphasized the importance of understanding the time and space complexities of each algorithm to make informed choices about which to use in different scenarios. By mastering these algorithms, one can significantly improve the performance of data handling and processing tasks, ensuring that applications run efficiently and effectively.

15.5 Questions

1. **What is the main difference between linear search and binary search?**

Ans.1 The main difference is that linear search scans each element sequentially, making it straightforward but inefficient for large datasets, with a time complexity of O(n). Binary search, on the other hand, requires the dataset to be sorted and repeatedly divides the search interval in half, making it much faster for large datasets, with a time complexity of O(log n).

2. **When would you prefer to use linear search over binary search?**

Ans.2 Linear search is preferred when the dataset is small or unsorted, and when the overhead of sorting the data before performing a binary search is not justified. It is also useful when the data is being received in real-time and needs to be searched immediately.

3. **What is the key advantage of Quick Sort compared to other sorting algorithms?**

Ans.3 The key advantage of Quick Sort is its average-case time complexity of O(n log n), which makes it very efficient for large datasets. It is also an in-place sort (it doesn't require additional storage), and its divide-and-conquer approach allows for easy parallelization. However, its worst-case time complexity is $O(n^2)$, which can be mitigated by using good pivot selection strategies.

4. **How does Merge Sort work and why is it considered stable?**

Ans.4 Merge Sort is a divide-and-conquer algorithm that splits the array into halves, recursively sorts each half, and then merges the sorted halves to produce the final sorted array. It is considered stable because it preserves the relative order of equal elements. The time complexity of Merge Sort is consistently O(n log n) for all cases.

5. **What are the primary factors to consider when choosing a sorting algorithm for a particular application?**

Ans.5 The primary factors to consider include the size of the dataset, the time complexity (average, worst, and best cases), space complexity, stability (whether the algorithm maintains the relative order of equal elements), and whether the dataset is already partially sorted. Additionally, the specific use case and any constraints on memory usage or processing power should be taken into account.

15.6 References

- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.

***