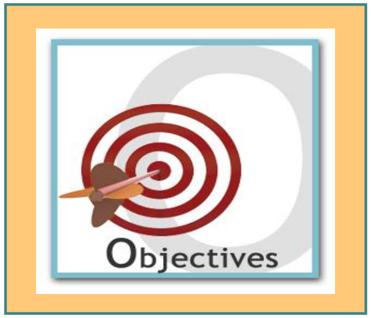
Data Structures Using C

Sorting and Searching Algorithms

Lesson Objectives

- **To understand and compare different**
  - **Sorting techniques:**
    - Bubble Sort
    - Quick Sort
    - Insertion Sort
  - **Searching techniques:**
    - Sequential Search
    - Binary Search



Sorting Techniques

- **Sorting is arranging elements in an ascending or descending order.**

- **Sorting comprises the following techniques:**
- **Bubble Sort**
  - Quick Sort
  - Insertion sort
  - Merge Sort
  - Count Sort

- **We will see Bubble sort, Quick sort and insertion sort in detail**

Bubble Sort - Algorithm

- **Logic for Bubble Sort Algorithm**
  - Compare adjacent elements (n) and (n+1), starting with n=1.
    - If the first is greater than the second, swap them.
  - Repeat this for each pair of adjacent elements, starting with the “first two elements”, and ending with the “last two elements”.
  - Repeat this for each pair of adjacent elements, starting with the “first two elements”, and ending with the “last two elements”.
    - At any point, the last element should be the largest.
  - Repeat the steps for all elements except the last one.
  - Keep repeating for one fewer element each time, until you have no more pairs to compare.

Simple Bubble Sort - Algorithm

- **Simple Bubble Sort:**

```
void bubbleSort(int numbers[], int array_size)
{
	int i, j, temp;
	for (i = (array_size - 1); i >= 0; i--)		{
	for (j = 1; j <= i; j++) {
			if (numbers[j-1] > numbers[j])		{
				/*swapping two numbers*/
				temp = numbers[j-1];
				numbers[j-1] = numbers[j];
				numbers[j] = temp; }
			}
		}
	}
```

Features

- **If the array contains n elements, then to sort them you require n-1 passes.**
- **If the data is mostly sorted, then we can do it faster.**
  - If there are no swaps in a particular iteration of the INNER loop, we can stop.
  - No swap indicates that array is already sorted.
- **It is called as improved bubble sort.**

Improved Bubble Sort - Algorithm

- **Improved Bubble Sort:**

```
void bubbleSort(int numbers[], int array_size)
{
    int i, j, temp, flag=0;
    for (i = (array_size - 1); i >= 0; i--)
    {
        for (j = 1; j <= i; j++)         {
            if (numbers[j-1] > numbers[j])        {
                /*swapping two numbers*/
                flag=1; /* flag is set to one*/
                temp = numbers[j-1];
                numbers[j-1] = numbers[j];
                numbers[j] = temp;
            }
        }/* end of inner for loop*/
```

Improved Bubble Sort - Algorithm (Contd...)

- **Improved Bubble Sort:**

```
/*If flag is 0 means no swapping is done hence break the loop */
If (flag==0)
    break;
else
   flag = 0; /*Reinitialize the flag for next iteration*/
} /* end of outer for loop*/
 } /*End of function*/
```

Quick Sort - Algorithm

- **Quick sort uses the principle of "divide-and-conquer".**
- **It requires only n log(n) time to sort n items.**
- **It is recursive. If the language does not support recursion, the implementation becomes extremely complicated.**

Quick Sort - Algorithm (Contd…)

- **Quick sort works by partitioning a given array**

  A[low . . high] in two non-empty sub arrays:
  - A[low . . q], and A[q+1 . . high]
    - Every value in A[low . . q] is less than or equal to every value in A[q+1 . . high].

- **Subsequently, the two subarrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array, and index "q" is computed as a part of the partitioning procedure.**

Quick Sort - Algorithm (Contd…)

- **Consider an array of N elements.**
- **Use the first element of the array as “Pivot value”.**
  - Partition the array so that all elements larger than the Pivot value are on the right side.
  - Keep all elements smaller than the Pivot value on the left side.
  - Keep the Pivot value in the right place.

Quick Sort - Algorithm Using Recursion

```
void sort(int elements[], int left, int right)
{
    int pivot, l, r;
    l = left;
    r = right;
    pivot = elements[left];
    while (left < right)
    {
        while ((elements[right] >= pivot) && (left < right))
            right--;
        if (left != right)
        {
            elements[left] = elements[right];
            left++;
        }
```

Quick Sort - Algorithm Using Recursion (Contd…)

```
while ((elements[left] <= pivot) && (left < right))
                    left++;
if (left != right)                    {
                    elements[right] = elements[left];
                    right--;     }              }
elements[left] = pivot;
pivot = left;
left = l;
right = r;
if (left < pivot)
      sort(elements, left, pivot - 1);
if (right > pivot)
      sort(elements, pivot + 1, right);
}
```

Example

- **Example on Quick sort algorithm:**
  (Demo using Quicksortalgorithm.txt)

Pivot	*left*	*right*	*Step*	*array*
43	0	6		43 76 22 45 53 10 66
43	1	6		43 76 22 45 53 10 66
43	1	5	66>43 right--	43 76 22 45 53 10 66
43	2	4	Swap 10 and 76	43 10 22 45 53 76 66

Example (Contd…)

Pivot	*left*	*right*	*Step*	*array*
43	2	4		43 10 22 45 53 76 66
43	2	2	Since 53 and 45 > pivot	43 10 22 45 53 76 66
43	3	2	Since 22 < pivot	43 10 22 45 53 76 66
43	3	2	Since left >right swap pivot and the number at right	22 10 43 45 53 76 66

Example (Contd…)

- **Observe that left part of the array contains all numbers less than (<) the pivot and right part contains values greater than (>) the pivot.**

- **Now, the same steps will be separately applied on both parts to get a sorted array.**

Insertion Sort - Description

- **Implemented by inserting a particular element at the appropriate position**
- **While inserting the element we need to find the position to insert the element.**
- **All other elements will be shifted one location on right to make space for new element and then the element will be inserted at the position**
- **This is normally done in place (i.e by using single array)**

Insertion Sort - Description (Contd…)

- **Let a0, ..., an-1 be the sequence to be sorted. At the beginning and after each iteration of the algorithm the sequence consists of two parts: the first part a0, ..., ai-1 is already sorted, the second part ai, ..., an-1 is still unsorted (i   0, ..., n).**

- **At the beginning the sorted part consists of element a0 only, at the end it consists of all elements a0, ..., an-1.**

Insertion Sort - Description (Contd…)

- **In order to insert element ai into the sorted part, it is compared with ai-1, ai-2 etc. When an element ak with ak<= ai is found, ai is inserted after it. If no such element is found, then ai is inserted at the beginning of the sequence.**

- **After inserting element ai the length of the sorted part has increased by one. In the next iteration, ai+1 is inserted into the sorted part and so on.**

Insertion Sort - Example

- **Example: Consider the following array**
- **5 7 0 3 4 2 6 1.**
- **On the left side the sorted part of the sequence is shown as underline. For each iteration, the number of positions the inserted element has moved is shown in brackets.**
- **5 7 0 3 4 2 6 1 (0) – only a[0] is in sorted part**
- **5 7 0 3 4 2 6 1 (0) – array is sorted till a[1]**

Insertion Sort - Example (Contd…)

- 0 5 7 3 4 2 6 1 (2) - 0 will be inserted at a[0] location
- 0 3 5 7 4 2 6 1 (2) - 3 will be inserted at a[2] position
- 0 3 4 5 7 2 6 1 (2) - 4 will be inserted at a[2] position
- 0 2 3 4 5 7 6 1 (4) - 2 will be inserted at a[2] position
- 0 2 3 4 5 6 7 1 (1) - 6 will get inserted at a[4] position
- 0 1 2 3 4 5 6 7 (6) - 1 will be inserted at a[1] position

Insertion Sort - Features

- **Less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort.**

- **Advantages**
  - simple implementation
  - efficient for (quite) small data sets
  - efficient for data sets that are already substantially sorted: the time complexity is $O(n + d)$, where d is the number of inversions

Insertion Sort - Algorithm

```
void insertionSort(int numbers[], int array_size)
{
        int i, j, index;
        for (i=1; i < array_size; i++)      {
            index = numbers[i];
            j = i;
            while ((j > 0) && (numbers[j-1] > index))
            {
                numbers[j] = numbers[j-1];
                j = j - 1;
            }
            numbers[j] = index;
        }
}
```


IGATE
Speed.Agility.Imagination

Simple Insertion Sort

- **How do you keep your cards sorted in a card game?**
  - You insert each new card in its proper place!
  - For example: a[0]=30
    - To add 15 in array a.

      shift 30 one location right, and then add 15 so that the numbers will be in a sorted order

      15 30
    - To add 40 at the end because it is > 30

      15 30 40

Searching Techniques

- **Searching is looking for an element in a set of elements.**
- **For Example**
  - Searching a word in a dictionary which consists of sorted words.
  - Searching for Employee Details With Employee Number in an Employee Directory
- **Searching comprises of following algorithms**
  - Sequential Search or Linear Search
  - Binary search

Sequential Search

- **Sequential search is also called as "Linear Search ".**
  - It is the simplest searching technique if number of elements are less.
  - It is useful when data is unsorted.
  - It operates by checking every element of a list one at a time in sequence until a match is found.
    - Best case: find the value at first position
    - Worst case: find the value at last position
    - Average case: find the value at the middle

Sequential Search - Example

- **Example on Sequential search:**
  - Consider an array as shown below:
    - 14 15 23 10 7 9
  - To search whether the number 23 exists in the array or not:
    - Start comparing from the first element one by one till we find the number or we reach the last element in the array.
    - We should stop searching once we get the number at 2nd position and return the position.
  - Suppose that a data set has N items "Sequential Search" requires N/2 comparisons on an average.

Binary Search

- **Binary Search is useful for searching sorted data.**

- **It is faster than sequential search.**

- **It reduces the span of searching the value.**

- **Steps involved in Binary Search:**
  1. Compare the value at middle, otherwise divide the data into two parts at the middle.
  2. If the value to be searched is less than (<) the value at middle, then search in the fist half otherwise in the next half.

- **Best case - The value is at the middle position.**

Binary Search

- **Example on Binary search:**
  - Consider an array as shown below:
    - 10 12 13 14 18 20 25 27 30 35 40 45 47
  - To search whether the number 18 exists in the array or not:
    - Compare 18 with middle element(ie.25). Otherwise
      - Divide the array: Because x <25, we need to search

        10 12 13 14 18 20.
      - Compare Middle Element in sub array otherwise divide and obtain the element

Binary Search

- **Note that, while using Binary Search:**
  - For N = 1000, you require maximum 10 comparisons.
  - For N = 1 million, you require maximum 20 comparisons.
  - The condition you have to fulfill is that you need to keep the data sorted on the relevant field.
- **Suppose a data set has N items, “Binary Search” requires log2(N) comparisons.**
- **Develop the Algorithm for Binary Search Using C.**

Binary Search - Sample Code

- **One possible solution:**
  Comment: returns index of matching array element

```
binarySearch(int  array[], int  value,  int left, int  right)
{
        int mid;
        if (right < left)
                return  -1;  //-1 represents Element Not Found
        mid = floor ( (right - left) / 2) + left;
        if (array [mid] == value)
                return mid;
```

Binary Search - Sample Code (Contd…)

```
if (value < array [mid] )
        return binarySearch(array, value, left, mid-1);
else
        return binarySearch(array, value, mid+1, right);
}
```

Comparison - Example

- **How do you look up a telephone number in the directory?**
  - You look at the name, say “Pramod Patil”, open the directory at random, and compare the first character of the first name on that page.
  - If the first character is less the “P”, you continue the search in the second half.
  - If the first character is greater than “P”, you continue the search in the first half.

Comparison - Example (Contd…)

- **If the first character is "P", you continue the search using the second character until you find the name you started with.**

- **What is the pre-requisite, to succeed with this kind of search?**
  - **Answer**: The directory has to be sorted in an ascending order of names!

Comparison - Example (Contd…)

- **Now, hypothesize that you have got hold of a phone number. You need to find out who it belongs to!**

- **Assuming you have only the directory to refer to, how will you locate the owner of this number?**
  - **Answer:** You need to do a “Sequential Search” on the Phone Numbers!!

Sequential Search Versus Binary Search

Array Size	Number of Comparisons by Sequential Search	Number of Comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4, 294, 967, 296	4, 294, 967, 296	33

IGATE
Speed.Agility.Imagination

Discussion

- **In the following scenario, which algorithm will you use for searching?**
  - You want to purchase a birthday gift of photo frame for your friend, so you walk down to the local photo store to examine their collection and find a suitable frame.

Discussion

- **Consider the following scenario**
  - A company stores their vendor details in a sorted linked list. If you want to search a vendor detail with vendor name among 100 elements, which algorithm will you use for searching.

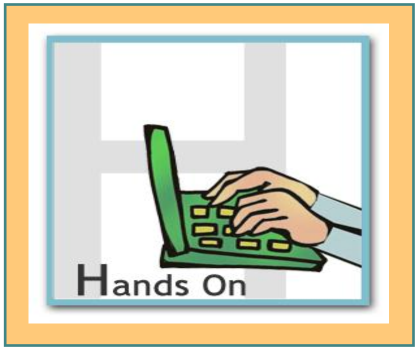
IGATE
Speed.Agility.Imagination

Discussion

- **Consider the following scenario**
  - A company stores their vendor details in a sorted linked list. If you want to search a vendor detail with vendor name among 100 elements, which algorithm will you use for searching.

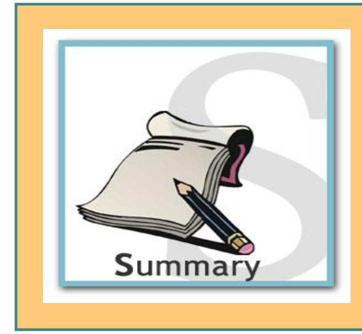
Lab

- **Lab 9**



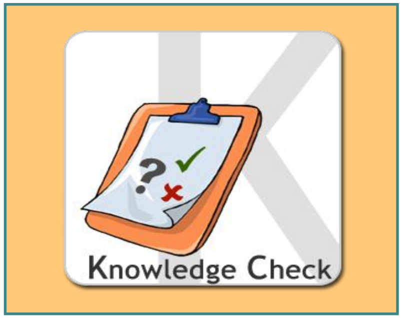
Summary

- **"Sequential Search" requires N/2 comparisons on an average.**
- **"Binary Search" requires log2(N) comparisons.**
- **There are different sorting techniques like:**
  - Bubble sort
  - Quick sort
  - Insertion sort



Review Question

- **Question 1: Which of the following sorting techniques uses recursion:**
  - Option 1: Bubble sort
  - Option 2: Quick Sort
  - Option 3: Insertion sort



- **Question 2: Arranging elements in an ascending or descending order is called as ___.**

Review Question: Match the Following

1.	Bubble sort
2.	Sequential search
3.	Binary search
4.	Quick sort
5.	Insertion sort

A.	Best case is finding element at the fist position
B.	Require to use nested loops
C.	Recursive method
D.	Find position before inserting element
E.	Best case is finding the element at the middle
F.	collision