Searching and Sorting

Searching refers to finding the position of a value in a collection of values. Sorting refers to arranging data om a certain order.

Introduction To Searching

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array. However if that value is not present in the array, the searching process displays an appropriate message and in this case is said to be unsucessful. Two popular methods for searching the array elements: **linar search** and **binary search**.

Linear Search

Linear search, also called a **sequential search**, is a very simple method used for searching an array for a particular value. It works by comparing the values to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements.

Complexity of Linear Search Algorithm

Linear search executes in $O(n)$ time where n is the number of elements in the array.

Binary Search

In this algorithm, `BEG` and `END` are the beginning and ending positions of the segments that we are looking to search for the element. `MID` is calculated as $\frac{\texttt{(BEG + END)}}{2}$. Initially, `BEG = lower_bound` and `END = upper_bound`. The algorithm is terminated wher `A[MID] = VAL`. When the algorithhm ends, we will set `POS = MID`. `POS` is the position at which the value is present in the array.

If `VAL` is not equal to `A[MID]`, then the values of `BEG`, `END`, and `MID` will be changed depending on whether `VAL`is smaller or greater than `A[MID]`.

(a) If `VAL` $<$ `A[MID]`, then `VAL` will be presenrt in the left segment of the array. So, the value of `END` will be changed as `END = MID - 1`.

(b) If `VAL` $>$ `A[MID]`, then `VAL` will be present in the right segment of the array. So, the value of `BEG` will be changed as `BEG = MID + 1`

Finally, if `VAL` is not present in the array, then eventually, `END` will be less than `BEG`. When ths happens, the algorithm will terminate and the search is unsuccessful.

Complexity of Binary Search Algorithm

The complexity of the binary search algorithm can be expressed as $f(n)$, where n is the number of elements in the array. The complexity of the algorithm is calculated depending on the number of comparisons that are made. In the binary search algorithm, we can see that with each comparison, the size of the segment where search has to be made is reduced to half. Thus, we can say that, in order to locate a particular value in the array, the total number of comparisons that will be made is given as

$$2^{f(n)} > n \quad \text{or} \quad f(n) = \log_2 n$$

Interpolation Search

Interpolation search, also known as extrapolation seazrch, is a searching technique that finds a specified value in a sorted array. In each step of interpolation search, the remaining search space for the value to be found is calculated. The calculation is done based on the values at the bounds of the search space and the value to be searched. The value found at this estimated position is the compared with the value being searched for. If the two values are equal, then the search is complete.

In case the values are not equal then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position. Thus, we see that interpolation search is similiar to binary search. However, the important difference between the two techniques is that binary search always selets the middle value of the remaining search space. It discards half of the values based on the comparison between the value found at the estimated position and the value to be searched. In interpolation search, interpolation is used to find an item near the one being search for, and then linear search is used to find the exact item.

Complexity of Interpolation Search Algorithm

When n elements of a list to be sorted are uniformly distributed (average case), interpolation search makes about $\log(\log n)$ comparisons. In the worst case, that is when the elements increase exponentialy, the algorithm can make up to $O(n)$ comparisons.

Jump Search

When we have an already sorted listm then the other efficient algorithm to search for a value is jump search or block search. In jump search, it is not necessary to scan all the elements in the list to find the desired value. We just check an element and if i is less than the desired value, then some of the elements following it are skipped by jumping ahead. Adter moving a little forward again, the element is checked. If the element is greater than the desired value, then we have a boundary and we are sure that the desired value lies between the previously checked element and the currently checked element. However, if the checked element is less than the element begin searched for, then we again make a small jump and repeat the process.

Adnatage of Jump Search over Linear Search

Suppose we have a sorted list of 1000 elements where the elements have values $0, 1, 2, 3, 4, \ldots, 999$, then sequential seach will find the value 674 in exactly 674 iterations, But with jump search, the same value can be found in 44 iterations. Hence, jump search performs far better than a linear search on a sorted list of elements.

Binary search is easy to implement and has a complexity of $O(\log n)$, but in case of a list having a large number of elements, jumping to the middle of the list to make comparisons is not a good idea because if the value being searched is at the beginning of the list then one (or even more) large step(s) in the backward direction would have to be taken. In such cases, jump search performs better as we have to move a little backwards only once. Hence, when jumping back is slower than jumping foward, the jump search algorithm always performs better.

How to Choose the Step Length?

For the jump search algorithm to work efficiently, we must define a fixed size for the step. If the step size is 1, then algorithm is the same as linear search. In order to find an appropriate step size, we must first try to figure out the relation between the size of the list (n) and the size of the step (k). Usually, k is calculated as $\sqrt{n}$

Further Optimization of Jump Search

In real-world applications, lists can be very large. In such large list searching the value from the beginning of the list is not a good idea. A better option is to start to search from the k-th element.

Complexity of Jump Search Algorithm

Jump search works by jumping through the array with a step size (optimially chosen to be $\sqrt{n}$) to find the interval of the value. Once this interval is identified, the value is searched using the linear search technique. Therefore, the complexity of the jump search algorithm is $O(\sqrt{n})$.

Introduction To Sorting

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if `A` is an array, then the elements of `A` are arranged in a sorted order (ascending order) in such a way that `A[o]` $<$ `A[2]` $< \ldots <$ `A[N]`.

A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be numerical order, lexicograhical order, or any user-defined order. There are two types of sorting:

- **Internal Sorting** which deals with sorting the data stored in the computer's memory
- **External Sorting** which deals with sorting the data stored in files. External sorting is applied when there is a voluminous data that cannot be stored in the memory.

Sorting on Multiple Keys

Many times, it is desired to sort arrays of records using multiple keys. This situation usually occurs when a single key is not sufficient to uniquely identify a record.

Practical Considerations for Internal Sorting

When analyzing the performance of different sorting algorithms, the practical considerations would be the following:

- Number of sort key comparisons that will be performed
- Number of times the records in the list will be moved
- Best case performance
- Worst Case performance
- Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done.

Bubble Sort

Bubble sort sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order). In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

Technique

(a) In Pass 1, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-2]` is compared with `A[N-1]`. Pass 1 involves $n-1$ comparisons and places the biggest element at the highest index of the array.

(b) In Pass 2, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`, and so on. Finally, `A[N-3]` is compared with `A[N-2]`. Pass 2 involves $n-2$ comparisions and places the second biggest element at the highest index of the array.

(c) In Pass 3, `A[0]` and `A[1]` are compared, then `A[1]` is compared with `A[2]`, `A[2]` is compared with `A[3]`,,and so on. Finally `A[N-4]` is compared with `A[N-3]`. Pass 3 involves $n-3$ comparisions and places the third biggest element at the third highest index of the array.

(d) In Pass $n-1$, `A[0]` and `A[1]` are compared, so that `A[0]` $<$ `A[1]`. After this step, all thr elements of the array are arranged in ascending order.

Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort there are $N-1$ passes in total. In the first pass $N-1$ comparisons are made to place the highest element in its correct position. Then in Pass 2, there are $N-2$ comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$$f(n) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1$$
$$f(n) = \frac{n(n-1)}{2}$$
$$f(n) = \frac{n^2}{2} + O(n) = O(n^2)$$

Therefore, the complexity of bubble sort algorithm is $O(n^2)$. Where n is the total number of elements in the array.

Bubble Sort Optimization

Consider a case when the array is already sorted. In this case no swampping is done but we still have to continue with all $n - 1$ passes. Once we detected that the array is sorted, the algorithm must not be executed further. In order to stop the execution of further passes after the array is sorted, we can have a variable flag which is set to `TRUE` before each pass and is made `FALSE` when a swapping is performed. The coded for the optimized bubble sort can be given as:

```
void bubble_sort(int *arr, int n) {
     int i, j, temp, flag = 0;
     for(i = 0; i < n; i++) {
        for(j = 0; j < n - i - 1; j++) {
           if(arr[j] > arr[j + 1]) s{
             flag = 1;
             temp = arr[j + 1];
             arr[j + 1] = arr[j];
             arr[j] = temp;
           }
        }
        if(flag == 0)    //array is sorted
           return;
     }
}
```

Complexity of Optimizd Bubble Sort Algorithm

In the best case, when the array is already sorted, the optimized bubble sort will take $O(n)$ time. In the worst case, when all the passes are performed, the algorithm will perform slower than the original algorithm. In the average case, the performance will see an improvement. The original bubble sort algorithm takes $O(n^2)$ in all the cases.

Insertion Sort

Insertion sort is a simple sorting algorithm in which the sorted array (or list) is built one element at a time. The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value unti it is in its correct place.

Technique

Insertion sort words as follows:

- The array of values to be sorted is divided into two sets. On ethat stores sorted values and another that contains unsorted values

- The sorting algorithm will proceed until there are no elements in the unsorted set.

- Suppose there are n elements in the array. Initially, the element with index 0 (assuming `LB = 0`) is in the sorted set. Rest of the elements are in the unsorted set.

- The first element of the unsorted partition has array index 1 (if `LB = 0`).

- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

Initially, `A[0]` is the only element in the sorted set. In Pass 1, A[1] will be placed either before or after `A[0]`, so that the array A is sorted. In Pass 2, `A[2]` will be placed either before `A[0]`, in between `A[0]` and `A[1]`, or after `A[1]`. In Pass 3, `A[3]` will be placed in its proper place. In Pass $N-1$, $A[N-1]$ will be placed in its proper place to keep the array sorted.

To insert an element `A[K]` in a sorted list `A[0], A[1],...,` $A[K-1]$, we need to compare `A[K]` with $A[k-1]$, then with $A[K-2]$, $A[K-3]$, and so on until we meet an element $A[J]$ such that $A[J] \leq A[K]$. In order to insert $A[K]$ in its correct position, we need to move elements $A[K-1], A[K-2], \ldots, A[J]$ by one position and then $A[K]$ is inserted at the $(j+1)^{\text{th}}$ location.

Complexity of Insertion Sort

The best case occurs when the array is already sorted. In this case, the running time of the algorithm has a linear running time $O(n)$. This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of teh array.

The worst case of the insertion sort occurs when the array is sorted in the reverse order. In the worst case, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. Therefore, in the worst case , insertion sort has a quadratic running time $O(n^2)$.

In the average case, the insertion sort algorithm will have to make at least $\frac{(K-1)}{2}$ comparisons. Thus, the average case also has a quadratic running time $O(n^2)$.

Advantages of Insertion Sort

The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.

- It can be efficiently implemented on data sets that are already substantially sorted.

- It performs better than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40% faster than the selection sort.

- It requires less memory space (only $O(1)$ of additional memory space).

- It is said to be online, as it can sort a list and recieve new elements.

Selection Sort

Selection sort is a sorting algorithm that has a quadratic running time complexity of $O(n^2)$, making it inefficient to be used on large lists. Although selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations. Selection sort is generally used for sorting files with very large objects (records) and small keys.

Technique

Consider an array `ARR` with N elements. Selection sort works as follows:

First find the smallest value in the array and place it in the first position. Then, find the second smallest value in the array and place it in the second position. Repeat this procedure until the entire array is sorted. Therefore,

- In Pass 1, find the position `POS` of the smallest value in the array and then swap `ARR[POS]` and `ARR[0]`. Thus, `ARR[0]` is sorted.

- In Pass 2, find the position of the smallest value in sub-array of $N-1$ elements. Swap `ARR[POS]` with `ARR[1]`. Now, `ARR[0]` and `ARR[1]` is sorted.

- In Pass $N-1$, find the position of the smaller of the elements $ARR[N-2]$ and $ARR[N-1]$. Swap `ARR[POS]` and $ARR[N-2]$ so that $ARR[0], ARR[1], \ldots, ARR[N-1]$ is sorted.

Complexity of Selection Sort

Selection sort is a sorting algorithm that is independent of the original order of the elements in the array. In Pass 1, selecting the element with the smallest value calls for scanning all n elements; thus, $n-1$ comparisons are required in the first pass. Then, the smallest value is swapped with the element in the first position. In Pass 2, selecting the second smallest value requires scanning the remaining $n-1$ elements and so on. Therefore,

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}=O(n^2)\ \textbf{comparisons}$$

Advantages of Selection Sort

- it is simple and easy to implement.

- It can be used for small data sets.

- It is 60% more efficient than bubble sort.

However, in the case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Merge Sort

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.

Divide means partitioning the n-element array to be sorted into two sub-arrays of $\frac{n}{2}$ elements. If A is an array containing zero or one element, then it is already sorted. However, if there are more elements in the array, divide A into two sub-arrays, A_1 and A_2, each containing about half of the elements of A.

Conquer means sorting the two sub-arrays recursively using merge sort.

Combine means merging the two sorted sub-arrays of size $\frac{n}{2}$ to produce the sorted array of n elements.

Merge sort algorithm focuses on two main concepts to improve its performance (running time):

- A smaller list takes fewer steps and thus less time to sort than large list.
- As number of steps is relatively less, thus less time is needed to create a sorted list from two sorted list rather than creating it using two unsorted lists.

The basic steps of a merge sort algorithm are as follows:

- If the array is of length 0 or 1, then it is already sorted.
- Otherwise, divide the unsorted array into two sub-arrays of about half the size.
- Use merge sort algorithm recursively to sort each sub-array.
- Merge the two sub-arrays to form a single sorted list.

The running time of merge sort in the average case and the worst case can be given as $O(n \log n)$. Although merge sort has an optimal time complexity, it needs an additional space of $O(n)$ for the temporary array `TEMP`.

Quick Sort

Quick sort makes $O(n \log n)$ comparisons in the average case to sort an array of n elements. However, in the worst case, it has a quadratic running time given as $O(n^2)$. Basically, the quick sortalgorithm is faster than other $O(n \log n)$ algorithms, because its efficient implementation can minimize the probability of requiring quadratic time. Quick sort is also known as partition sort.

Like merge sort, this algorithm works by using divide-and-conquer strategy to divide a single unsorted array into two smaller sub-arrays.

The quick sort algorithm works works as follows:

1. Select an element pivot from the array elements.

2. Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way). After such a partitioning, the pivot is placed in its final position. This is called the **partition** operation.

3. Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)

Like merge sort, the **base case** of the recursion occurs when the array has zero or one element because in that case the array is already sorted. After the iteration, one element (pivot) is always in its final position. Hence, with every iteration, there is one less element to be sorted in the array.

Thus, the main task is to find the pivot element, which will partition the array into two halves.

Technique

Quick sort works as follows:

1. Set the index of the first element in the array `loc` and `left` variables. Also, set the index of the last element of the array to the `right variable`.

That is, `loc = 1, left = 0`, and `right =` $n - 1$ (where n in is the number of elements in the array)

2. Start from the element pointed by right and scan the array from right to left, comparing each element on the way with the element pointed by the variable `loc`.

That is, `a[loc]` should be less than `a[right]`.

(a) If that is the case, then continue comparing until `right` becomes equal to `loc`. Once `right = loc`, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have a `a[loc]` $>$ `a[right]`, then interchange the two values and jump tp Step 3.

(c) Set `loc = right`

3. Start from the element pointed by `left` and scan the array from left to right, comparing each element on the way with the element pointed by loc.

That is, `a[loc]` should be greater than `a[left]`.

(a) If that is the case, then continue comparing until `left` becomes equal to `loc`. Once `left = loc`, it means the pivot has been placed in its correct position.

(b) However, if at any point, we have `a[loc]` $<$ `a[left]`, then interchange the two values and jump tp Step 2.

(c) Set `loc = left`.

Complexity of Quick Sort

In the average case, the running time of quick sort can be given as $O(n \log n)$. The partitioning of the array simply loops over the elements of the array once uses $O(n)$ time.

In the best case, every time we partition the array, we divide the list into two nearly equal pieces. That is, the recursive call processes the sub-array of half the size. At the most, only $\log n$ nested calls can be made before we reach a sub-array of size 1. It means the depth of the call tree is $O(\log n)$. And because at each level, there can only be $O(n)$, the resultant time is given as $O(n \log n)$. And because at each level, there can only be $O(n)$, the resultant time is given as $O(n \log n)$ time.

Practically, the efficiency of quick sort depends on the element which is chosen as the pivot. Its worst-case efficiency is given as $O(n^2)$. The worst case occurs when the array is already sorted and left-most element is chosen as the pivot.

However, many implementations randomly chooses the pivot element. The randomized version of the quick sort algorithm always has an algorithmic complexity of $O(n \log n)$.

Pros and Cons of Quick Sort

It is faster than other algorithms such as bubble sort, selection sort, and insertion sort. Quick sort can be used to sort arrays of small size, medium size, or large size. On the flip side, quick sort is complex and massively recursive.

Radix Sort

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the **radix** is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. The words are first sorted according to the first letter of the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B and so on.

During the second pass, names are grouped to the second letter. After the second pass, names are sorted on the first two letters. This process is continued till the n^{th} pass, where n is the length of the name with maximum numbers of letters.

After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names begining with A. In the second pass, collect the names from the second bucket, and so on.

When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the number, we have ten buckets, each for one digit $(0, 1, \ldots, 9)$ and the number of passes will depend on the length of the number having maximum number of digits.

Complexity of Radix Sort

To calculate the complexity of radix sort algorithm, assume there are n numbers that have to be sorted and k is te number of digits in the largest number. In this case, the radix sort algorithm takes $O(kn)$ times to execute. When radix sort is applied on a data set of finite size (very small set of numbers), then the algorithm runs in $O(n)$ asymptotic time.

Pros and Cons of Radix Sort Radix sort is a very simple algorithm. When programmered properly, radix sort is one of the fastest sorting algorithms for numbers or strings of letters.

There are trade-offs for radix sort that can make it s less preferable as compared to other sorting algorithms. Radix sort takes more space than other sorting algorithms. Radix sort takes more spcae than other sorting algorithms. Besides the array of numbers, we need 10 buckes to sort number, 26 buckets to sort strings containing only characters, and at least 40 buckets to sort a string containing alphanumeric characters.

Another drawback of radix sort is that the algorithm is dependent on digits or letters. This feature compromises with the flexibility to sort input of any data type. For every different data type, the algorithm has to be rewritten. Even id the sorting order changes, the algorithm has to be rewritten. Thus, radix sort takes more time to write and writing general purpose radix sort algorithms that can handle all kinds of data is not a trivial tasks.

Heap Sort

Using the basic concepts of heaps, we apply them to write efficient algorithm called heap sort that has a running time complexity of $O(n \log n)$

Given an array `ARR` with n elements, the heap sort algorithm can be used to sort `ARR` in two phases:

- In Phase 1, build a heap H using the elements of `ARR`.
- In Phase 2, repeatedly delete the root element of the heap formed in Phase 1.

In a max heap, we know that the largest value in H is always present at the root node. So in Phase 2, when the root element is deleted, we are actually collecting the elements if `ARR` in decreasing order.

Complexity of Heap Sort

Heap sort uses two heap operations: `insertions` and `root deletion`. Each element extracted from the root is placed in the last empty location of the array.

In Phase 1, when we build a heap, the number of comparisons to find the right location of the new element in H cannot exceed the depth of H. Since H is a complete tree, its depth cannot exceed m, where m is the number of elements in heap H.

Thus, the total number of comparisons $g(n)$ to insert n elements of `ARR` in H is bounded as:

$$g(n) \leq n \log n$$

hence, the running time of the first phase of the heap sort algorithms is $O(n \log n)$,

In Phase 2, we habe H which is a compelte tree with m elements having left and right sub-trees as heaps. Assuming L to be the root of the tree, **reheaping** the tree would need 4 comparisons to move L one step down the tree H. Since the depth of H cannot exceed $O(\log m)$, reheaping the tree will require a maximum of $4 \log m$ comparisons to find the right location of L in H.

Since n elements will be deleted from heap H, reheaping will be done n times. Therefore, the number of comparisons to delete n elements is bounded as:

$$h(n) \leq 4n \log n$$

Hence, the running time of the second phase of the heap sort algorithm is $O(n \log n)$.

Each phase requires time proportional to $O(n \log n)$. Therefore, the running time to sort an array of n elements in the wost case is proportional to $O(n \log n)$.

Therefore, we can conclude that heap sort is a simple, fast, and stable sorting algorithm that can be used to sort large sets of data efficiently.

Shell Sort

Shell short, is a sort algorithm that is a generalization of insertion sort. While discussing insertion sort, we have observered two things:

- First, insertion sort works well when the input is "almost sorted".
- Second, insertion sort is quite inefficient to use as it moves values just one position at a time.

Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position. In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes. However, the last step of Shell sort is a plain insertion sort. But by the time we reach the last step, the elements are already "almost sorted", and hence it provides good performance.

Technique

To visuaize the way in which shell sort works, perform the following steps:

•bf Step 1: Arrange the elements of the array in the form of a table and sort the columns (using insertion sort).

•**Step 2:** Repeat Step 1, each time with smaller number of longer columns in such a way that at the end, there is only one collumn of data to be sorted.

Note that we are only visualizing the elements being arranged in a table, the algorithm does its sorting in-place.

Tree Sort

A tree is a sorting algorithm that sorts numbers by making use of the properties of binary search tree. The algorithm builds a binary search tree using the numbers to be sorted and then does an in-order traversal so that the numbers are retrieved in a sorted order.

Complexity of Tree Sort Algorithm

Best Case

- Inserting a number in a binary search tree takes $O(\log n)$ time.
- So, the complete binary search tree with n numbers is built in $O(n \log n)$ time.
- A binary tree is traversed in $O(n)$ time.
- Total time required $= O(n \log n) + O(n) = O(n \log n)$

Worst Case

- Occurs with an unbalanced binary search, i.e., when the numbers are already sorted.
- Binary search tree with n numbers is builit in $O(n^2)$ time.
- A binary tre is traversed in $O(n)$ time.
- Total time required $= O(n^2) + O(n) = O(n^2)$.
- The worst case can be improved by using a self-balancing binary search tree.

External SOrting

External sorting is a sorting technique that can handle massive amounts of data. It is usually applied when the data being sorted does not fit into the main memory (RAM) and, thereforem a slower memory (usually disk or even magnetic tape) needs to be used.

Generalized External Merge Sort Algorithm

If the amount of data to be sorted exceeds the available memory by a factor of K, then K chuncks (also known as K run lists) of data are created. These K chunks are sorted and then a K-way merge is performed. If the amount of RAM available is given as X, then there will be K input bufers and 1 output buffer.

If the ratio of data to be sorted and available RAM is particularly large, multi-pass sorting is used. We can first merge only the first half of the sorted chunks, then the other half, and finally merge the two sorted chuncks. The exact nuber of passes depends on the following factors:

- Size of the data sorted when compared with the available RAM
- Physical characteristics of the magnetic disk such as transfer rate, seek time, etc.

Applications of External Sorting

External sorting is used to update a master file from a transaction file.

It is also used in database applications for performing operations like **Projection** and **Join**. Projection means selecting a subset of fields and join means joining two files on a common field to create a new file whose fields are the union of the fields of the two files. External sorting is also used to remove duplicate records.

Points to Remember

• Searching refers to finding the position of a value in a collection of values. Some of the popular searching techniques are linear search, binary search, interpolation search, and jump search.

• Linear search works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

• Binary Search works efficiently with a sorted list. In this algorithm, the value to be searched is compared with the middle element of the array segment.

• In each step of interpolation search, the search space for the value to be found is calculated. The calculation is done based on the values to be searched.

• Jump search is used with sorted lists. We first check an element and if it is less than the desired value, then a block of elements is skipped by jumping ahead, and the element is greater than the desired value, then we have a boundary and we are sure that the desired value lies between the previously checked element and the currently checked element.

• Internal sorting deals with sorting the data stored in the memory, whereas external sorting deals with sorting the data stored in files.

• In bubble sorting, consecutive adjacdent pairs of elements in the array are compared with each other.

• Insertion sort works by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in the correct place.

• Selection sort works by finding the smallest value and placing it in the first position. It then finds the second smallest value and places it in the second position. This procedure is repeated until the whole array is sorted.

• Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm. Divide means partitioning the n-element array to be sorted into two sub-arrays of $\frac{n}{2}$ elements in each sub-array. Conquer means sorting the two sub-arrays recursively using merge sort. Combine means merging the two sorted sub-arrays of size $\frac{n}{2}$ each to produce a sorted array of n elements. The running time of merge sort in average case and worst case can be given as $O(n \log n)$

• Quick sort works by using a divide-and-conquer strayegy. It selects a pivot element and rearranges the elements in such a way that all elements less than pivot appear before it and all elements greater than pivot appear after it.

• Radix sort is a linear sorting algorithm that uses the concept of sorting names in alphabetical order.

• Heap sort sorts an array in two phases. In the first phase, it builds a heap of the given array. In the second phase, the root elements is deleted repeatedly and inserted into an array.

• Shell sort is considered an improvement over insertion sort, as it compares elements separated by a gap of several positions.

• A tree sort is a sorting algorithm that sorts numbers by making use of the properties of binary search tree. The algorithm first builds a binary search tree using the numbers to be sorted and then does an in-order traversal so that the numbers are retrieved in a sorted order.