Sorting

Onee of the most common and time consuming tasks in computer science is the

retrieval of target information from huge data, which needs searching. Searching

is the process of fi nding the location of the target among a list of objects. The two basic

search techniques are the following:

1. S equential search

2. B inary se arch

There are certain ways of organizing data, which make the search process more effi -

cient. If the data is kept in a proper order, it is much easier to search. Sorting is a process

of organizing data in a certain order to help retrieve it more effi ciently.

Searching

The proce ss of locating target data is known as *searching*. Consider a situation wh ere you

are trying to get the phone number of your friend from a telephone directory. The telephone

directory can be thought of as a table or a fi le, which is a collection of records. Each

record has one or more fi elds such as name, address, and telephone number. The fi elds,

which are used to distinguish records, are known as *keys*. While searching, we are asked

to fi nd the record which contains information along with the target key. When we think of

a telephone directory, the search is usually by name. However, when we try to locate the

record corresponding to a given telephone number, the key will be the telephone number.

Search techniques

Depending on the way data is scanned for searching a particular record, the search techniques

are categorized as follows:

1. Sequential search

2. Binary search

3. Fibonacci search

4. Index sequential search

5. Hashed search

The performance of a searching algorithm can be computed by counting the number of

comparisons to find a given value.

Sequential Search

The easiest search technique is a sequential search. This is a technique that must be used

when records are stored without any consideration given to order.

Hence, a sequential search begins with the first available record and proceeds to the

next available record repeatedly until we find the target key or conclude that it is not

found. Sequential search is also called as *linear search*.

int SeqSearch (int A[max], int key, int n)

{

int i, fl ag = 0, position;

for(i = 0; i < n; i++)

{

if(key == A[i])

{

position = i;

fl ag = 1;

break;

}

}

if(fl ag == 1) // if found return position

return(position);

else // return −1 if not found

return(−1);

}

Let us compute the amount of time the sequential search needs to search for a target

data. For this, we must compute the number of times the comparisons of keys is done. In

general, for any search algorithm, the computational complexity is computed by considering

the number of comparisons made.

The number of comparisons depends on where the target data is stored in the search list.

If the target data is placed at the fi rst location, we get it in just one comparison. Two comparisons

are needed if the target data is in the second location. Similarly, *i* comparisons are

required if the target data is at the *i*th location and *n* comparisons, if it is at the *n*th location.



**Ordered list search** When elements are ordered, binary search (discussed in Section 9.2.2)

is preferred. However, when data is ordered and is of smaller size, sequential search with a

small change is preferred to binary search. In addition, when the data is ordered but stored

in a data structure such as a linked list, modified sequential search is preferred. While

searching an ordered list, we need not continue the search till the end of list to know that the

target element is not in the list. While searching in an ascending ordered list, whenever an

element that is greater than or equal to the target is encountered, the search stops.

Binary Search

As discussed, sequential search is not suitable for larger lists. It requires *n* comparisons in

the worst case. We have a better method when the data is sorted. Let us consider a typical

game played by kids. You are asked to guess the number thought of by your friend in the

range of 1 to 100. You are to guess by asking him a minimum number of questions. Of

course, you are not allowed to ask him the number itself. The easiest approach is to start

asking him, ‘Is it 1?’ In case the answer is ‘No’, then ask, ‘Is it 2?’ Continue this process

in the ascending order of integers till you get the answer as ‘Yes’.

What if the number your friend has in mind is 99? Obviously, this approach is not

an efficient one. The solution to this problem is to ask him a question, ‘Is it 50?’ If no,

another question to be asked is, ‘is it greater than 50?’ If the answer is ‘Yes’, then the

range to be searched is 51 to 100, which is half of the previous range. If the answer is

‘No’, the range is 1 to 49, which is again half of the original. You may continue doing so

till you guess the number. Surely, the second approach reduces the total number of questions

asked on an average.

This method is called *binary search*, as we have divided the list to be searched every time

into two lists and the search is done in only one of the lists. Consider that the list is sorted in

ascending order. In binary search algorithm, to search for a particular element, it is first

compared with the element at the middle position, and if it is found, the search is successful,

else if the middle position value is greater than the target, the search will continue in the first

half of the list; otherwise, the target will be searched in the second half of the list. The same

process is repeated for one of the halves of the list till the list is reduced to size one.

Algorithm 9.3 depicts the logic behind this type of search.

Let n be size of the list

Let target be the element to be searched

Let flag = 0, low = 0, high = n1

2. if low high, then

middle = (low + high)/2

else goto step (5)

3. if(key[middle] = target)

Position = middle, fl ag = 1

Goto step (5)

else if(key[middle] > target) then

high = middle − 1

else

low = middle + 1

4. Goto step(2)

5. if fl ag = 1

report as target element found at location ‘position’

else

report that element is not found in the list

6. stop

Sorting

One of the fundamental problems in computer science is ordering a list of items. There

are plenty of solutions to this problem, commonly known as *sorting algorithms*. Some

sorting algorithms are simple and iterative, such as the bubble sort. Others such as the

quick sort are extremely complicated but produce lightning-fast results.

*Sorting* is the operation of arranging the records of a table according to the key value

of each record, or it can be defined as the process of converting an unordered set of elements

to an ordered set.

Types of Sorting

Sorting algorithms are divided into two categories: internal and external sorts.

If all the records to be sorted are kept internally in the main memory, they can be

sorted using an internal sort. However, if there are a large number of records to be

sorted, they must be kept in external files on auxiliary storage.

*Internal Sorting*

Any sort algorithm that uses main memory exclusively during the sorting is called as an

*internal sort algorithm*. This assumes high-speed and random access to all data members.

The various internal sorting

techniques are the following:

1. Bubble sort

2. Insertion sort

3. Selection sort

4. Quick sort

5. Heap sort

6. Shell sort

7. Bucket sort

8. Radix sort

9. File sort

10. Merge sort

General Sort Concepts

*Sort Order*

Data can be ordered either in ascending or in descending order. The order in which the

data is organized, either ascending or descending, is called *sort order*. For example, the

percentages of marks obtained by students in the examination are organized in descending

order to decide ranks, whereas the names in the telephone directory are organized

alphabetically in ascending order.

*Sort Stability*

A sorting method is said to be stable if at the end of the method, identical elements occur

in the same relative order as in the original unsorted set. While sorting, we must take care

of the special case—when two or more of the records have the same key, it is important

to preserve the order of records in this case of duplicate keys.





Bubble Sort

The bubble sort is the oldest and the simplest sort in use. Unfortunately, it is also the slowest.

The bubble sort works by comparing each item in the list with the item next to it and

swapping them if required.

illustrates a bubble sort using an array of size 7.



void bubblesort(int A[max], int n)

{

int i, j,temp;

for(i = 1; i < n; i++) // number of passes

{

for(j = 0; j < n − i; j++) // j varies from 0 to

// n − i

{

if( A[j] > A[j + 1] ) // compare two successive

// numbers

{

temp = A[j]; // swap A[j] with A[j + 1]

A[j] = A[j + 1];

A[j + 1] = temp;

}

}

}

}

For descending order of sorting, only the comparison condition should be changed in

Program Code 9.7.

if(A[j] < A[j + 1] ) // change as <

Insertion Sort

The insertion sort works just like its name suggests—it inserts each item into its proper

place in the final list. The simplest implementation of this requires two list structures: the

source list and the list into which the sorted items are inserted.

Let us consider a list *L* {3, 6, 9, 14}. Given this sorted list, we need to insert a new

element 5 in it. The commonly used process would involve the following steps:

1. Compare the new element 5 and the last element 14

2. Shift 14 right to get 3, 6, 9, ,14

3. Shift 9 right to get 3, 6, ,9, 14

4. Shift 6 right to get 3, ,6, 9, 14

5. Insert 5 to get 3, 5, 6, 9, 14

These steps could be coded as the following piece of code:

// insert t into a[0:i − 1]

int j;

// let X be the element to be inserted

// shift elements from the last member to right by one position

// till you get a smaller one

for(j = i − 1; j >= 0 && X < a[j]; j−−)

a[j + 1] = a[j];

// Insert t at j + 1 location

a[j + 1] = X;

These steps when done for each element of the list are to be sorted by considering

another list and starting with one element in it. The steps for inserting an element in the

sorted list can then be repeatedly used to yield the sorted list. Let us consider the following

list of numbers: *L* {7, 3, 5, 6, 1}. The following steps are required to sort this list.

1. Start with 7 and insert 3 > 3, 7

2. Insert 5 > 3, 5, 7

3. Insert 6 > 3, 5, 6, 7

4. Insert 1 > 1, 3, 5, 6, 7

The piece of code needed to do this will look like

for(int i = 1; i < n; i++)

{ // insert a[i] into a[0:i − 1]

// code to insert comes here

}

After adding the code for insertion we have already built, the resultant code

will be

for(int i = 1; i < n; i++)

{ // insert a[i] into a[0:i − 1]

int t = a[i];

int j;

for(j = i − 1; j >= 0 && t < a[j]; j−−)

a[j + 1] = a[j];

a[j + 1] = t;

}