Done by Mokhtar M. Hasan, Lecturer at the Computer Science Department, College of Science for Women, to fulfill the course subject of Advanced Algorithm material.

Advanced Algorithms, Third Class, Computer Science Department, CSW, 2013-2014

Textbooks:

1- Introduction to Algorithms (Third Edition) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivert, and Clifford Stein, The MIT Press, 2009.

2- Algorithms (Fourth Edition) by Robert Sedgewick, and Kevin Wayne, Pearson Education, 2011.

3- Algorithms and Data Structures, by Alfred Strohmeier, 2000.

4- Sorting and Searching Techniques, Unit II.

5- Sorting and Searching Algorithms: A Cookbook, by Thomas Niemann, Portland, Oregon.

Definition of Algorithm: The term *algorithm* is used in computer science to describe a finite, deterministic, and effective problem-solving method suitable for implementation as a computer program.

Specificatins:

- independent of the particular programming language being used.
- equally appropriate for many computers and many programming languages.
- It is the method, rather than the computer program itself.
- specifies the steps that we can take to solve the problem.

We can define an algorithm by describing a procedure for solving a problem in a natural language, or by writing a computer program that implements the procedure.

Example: Greatest Common Divisor for two any given two numbers, called Euclid's algorithm, as follows:

English-language description

Compute the greatest common divisor of two nonnegative integers *p* and *q* as follows: If *q* is 0, the answer is *p*. If not, divide *p* by *q* and take the remainder *r*. The answer is the greatest common divisor of *q* and *r*.

VB.net-language description

```
public function gcd(p as integer,
           q as integer) as integer
   if q = 0 then
      return p
   else
      dim r as integer = p % q;
      return gcd(q, r)
   end if
end function
```

Most algorithms of interest involve organizing the data involved in the computation. Such organization leads to data structures, which also are central objects of study in computer science. Algorithms and data structures go hand in hand.

Simple algorithms can give rise to complicated data structures and, conversely, complicated algorithms can use simple data structures.

When we use a computer to help us solve a problem, we typically are faced with a number of possible approaches. For small problems, it hardly matters which approach we use, as long as we have one that correctly solves the problem. For huge problems (or applications where we need to solve huge numbers of small problems), however, we quickly become motivated to devise methods that use time and space efficiently.

When developing a huge or complex computer program, a great deal of effort must go into understanding and defining the problem to be solved, managing its complexity, and decomposing it into smaller subtasks that can be implemented easily.

Analyzing an Algorithm: This term has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process. So, the choice of which algorithm that will be adopted perhaps involving sophisticated mathematical analysis as shown hereinbefore.

Sorting Algorithms

Sorting is the process of rearranging a sequence of objects so as to put them in some logical order.
Our primary concern is algorithms for rearranging arrays of items where each item contains a key. The objective of the sorting algorithm is to rearrange the items such that their keys are ordered according to some well-defined ordering rule (usually numerical or alphabetical order). We want to rearrange the array so that each entry's key is no smaller than the key in each entry with a lower index and no larger than the key in each entry with a larger index in case of ascending order and vice versa in case of descending order.

Certification:
Does the sort implementation always put the array in order, no matter what the initial order? As a conservative practice, practically, we include the statement assert isSorted(a); in our test client to certify that array entries are in order after the sort.

1- Selection Sort

One of the simplest sorting algorithms works as follows: First, find the smallest item in the array and exchange it with the first entry (itself if the first entry is already the smallest). Then, find the next smallest item and exchange it with the second entry. Continue in this way until the entire array is sorted. This method is called selection sort because it works by repeatedly selecting the smallest remaining item, the aforementioned speech can be formalized as the following algorithm:

Pseudo Code:
Consider you have Start and End indices which represent the boundaries of a given array.

```
• Find the index Small;
• Exchange the values located at Start and Small;
• Advance Start
```

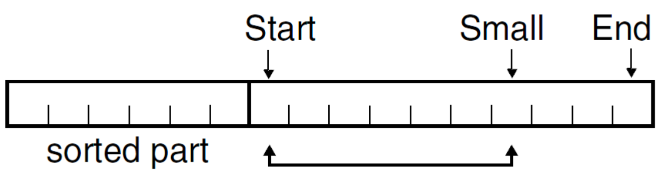
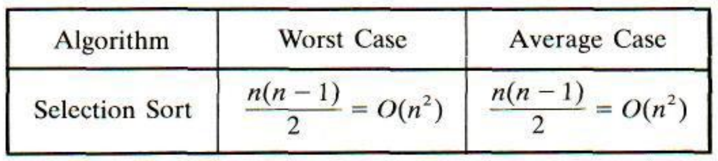


Figure: shows the mechanisms of the Selection Sort Algorithm.

Complexity of the Selection Sort Algorithm:



Algorithm	Worst Case	Average Case
Selection Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{2} = O(n^2)$

- Selection sort uses $N^2/2$ compares and N exchanges to sort an array of length N.
- Negative: $O(n^2)$ comparisons are needed, independently of the initial order, even if the elements are already sorted.
- Positive: Never more than O(n) moves are needed.
- Conclusion: It’s a good technique for elements heavy to move, but easy to compare.

Example:
Suppose an array A contains 8 elements as follows:
77, 33, 44, 11, 88, 22, 66, 55

Elements	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7
77	*11*	*11*	*11*	*11*	*11*	*11*	*11*
33	33	*22*	*22*	*22*	*22*	*22*	*22*
44	44	44	*33*	*33*	*33*	*33*	*33*
11	77	77	77	*44*	*44*	*44*	*44*
88	88	88	88	88	*55*	*55*	*55*
22	22	33	44	77	77	*66*	*66*
66	66	66	66	66	66	77	*77*
55	55	55	55	55	88	88	88
	changed	changed	changed	changed	changed	changed	Not changed

Note: italic values in the table are in right order and never compared again.

Sorted data: 11, 22, 33, 44, 55, 66, 77, 88

2- Insertion Sort

It is the process of inserting each element into its proper position. This sorting algorithm is frequently used when n is small.

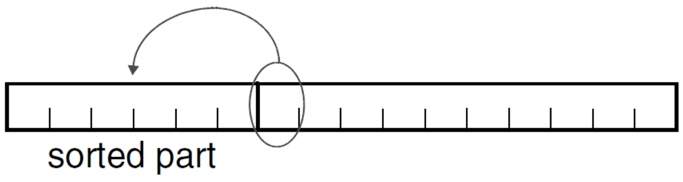
As in selection sort, the items to the left of the current index are in sorted order during the sort, but they are not in their final position, as they may have to be moved to make room for smaller items encountered later. The array is, however, fully sorted when the index reaches the right end, in other words:

The insertion sort algorithm scans A from A[l] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[l], A[2], . . . , A[K-1].

Unlike that of selection sort, the running time of insertion sort depends on the initial order of the items in the input. For example, if the array is large and its entries are already in order (or nearly in order), then insertion sort is much, much faster than if the entries are randomly ordered or in reverse order.

In simple words:

Insert an element in a sorted sequence keeping the sequence sorted.



Complexity of the Insertion Sort Algorithm:

Algorithm	Worst Case	Average Case
Insertion Sort	$\frac{n(n-1)}{2} = O(n^2)$	$\frac{n(n-1)}{4} = O(n^2)$

	Comparisons	Exchanges
Best Case	N-1	0
Average	$N^2/4$	$N^2/4$
Worst Case	$N^2/2$	$N^2/2$

Example:

Suppose an array A contains 8 elements as follows:
77, 33, 44, 11, 88, 22, 66, 55

Elements	**Pass 1**	**Pass 2**	**Pass 3**	**Pass 4**	**Pass 5**	**Pass 6**	**Pass 7**
77	*33*	*33*	*11*	*11*	*11*	*11*	*11*
33	*77*	*44*	*33*	*33*	*22*	*22*	*22*
44	44	*77*	*44*	*44*	*33*	*33*	*33*
11	11	11	*77*	*77*	*44*	*44*	*44*
88	88	88	88	*88*	*77*	*66*	*55*
22	22	22	22	22	*88*	*77*	*66*
66	66	66	66	66	66	*88*	*77*
55	55	55	55	55	55	55	88
	changed	changed	changed	changed	changed	changed	changed

Note: italic values in the table are in right order and never compared again.

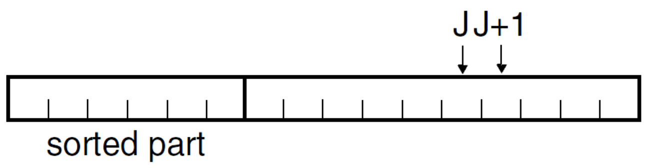
Sorted data: 11, 22, 33, 44, 55, 66, 77, 88

3-Bubble sort

In this sorting algorithm, multiple swapping take place in one iteration. Smaller elements move or 'bubble' up to the top of the list. In this method ,we compare the adjacent members of the list to be sorted , if the item on top is greater than the item immediately below it, they are swapped.

Pseudo Code:

• *walk through the unsorted part from the end;*
• *exchange adjacent elements if not in order;*
• *increase the sorted part, decrease the unsorted part by one element.*



Complexity of the Bubble Sort Algorithm:
The total number of comparisons in Bubble sort are: = (N-1)+(N-2)….+2+1
=(N-1)*N/2=O(N^2), and uses O(n^2) comparisons on average.

	Comparisons	Exchanges
Best Case	N-1	0
Average	(1/2)*n*(n-1)	(1/2)*n*(n-1)
Worst Case	(1/2)*n*(n-1)	(1/2)*n*(n-1)

Example:
Suppose an array A contains 8 elements as follows:
77, 33, 44, 11, 88, 22, 66, 55

Elements	**Pass 1**	**Pass 2**	**Pass 3**	**Pass 4**	**Pass 5**	**Pass 6**	**Pass 7**
77	*11*	*11*	*11*	*11*	*11*	*11*	*11*
33	77	*22*	*22*	*22*	*22*	*22*	*22*
44	33	77	*33*	*33*	*33*	*33*	*33*
11	44	33	77	*44*	*44*	*44*	*44*
88	22	44	44	77	*55*	*55*	*55*
22	88	55	55	55	77	*66*	*66*
66	55	88	66	66	66	77	*77*
55	66	66	88	88	88	88	88

Note: italic values in the table are in right order and never compared again.

Sorted data: 11, 22, 33, 44, 55, 66, 77, 88

3-Quick Sort

This is the most widely used internal sorting algorithm. It is based on divide-and-conquer type, i.e. Divide the problem into sub-problems, until solved sub problems are found, it is worth to mention that this algorithm is recursive one.

Sorting Table (Start..End):
• Partition Table (Start..End), and call J the location of partitioning;
• Sort Table (Start..J-1);
• Sort Table (J+1..End).

Complexity of the Bubble Sort Algorithm:
The worst case is when the sequence is already sorted, and execution time is of order O(n^2), and the best case is when the sequence is divided exactly at its mid-position, so time is O(nlogn).

However, since there is a recursive behavior in this sorting algorithm, so we need additional storage space than before, hence, the storage space for best case and average case is O(log n), and for worst case is proportional to n since the algorithm calls itself n-1 times.

Example:
Suppose A is the following list of 12 numbers:

(44) 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

The quick sort algorithm finds the final position of one of the numbers; in this illustration, we use the first number, 44. This is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

(22) 33, 11, 55, 77, 90, 40, 60, 99, (44) 88, 66

(Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.)
Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list

22, 33, 11, (44) 77, 90, 40, 60, 99, (55), 88, 66 .

(Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.)
Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list

22, 33, 11, (40) 77, 90, (44) 60, 99, 55, 88, 66

(Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22, 33, 11, 40, (44) 90, (77) 60, 99, 55, 88, 66

(Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

$\underbrace{22, 33, 11, 40,}_{\text{First sublist}}$ (44) $\underbrace{90, 77, 60, 99, 55, 88, 66}_{\text{second sublist}}$

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.
The above reduction step is repeated with each sublist containing 2 or more elements.

Note: in the above example, the pivot element is the first element; we can choose the middle element as the pivot element.

So, the entire the solution will be:

Initial index i=0, j=11, pivot=44, index of pivot=0

Data	Pivot element	i index	j index
(44), 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66	44	0	9
22, 33, 11, 55, 77, 90, 40, 60, 99, (44), 88, 66	44	3	9
22, 33, 11, (44), 77, 90, 40, 60, 99, 55, 88, 66	44	3	6
22, 33, 11, 40, 77, 90, (44), 60, 99, 55, 88, 66	44	3	6
22, 33, 11, 40, **(44)**, 90, 77, 60, 99, 55, 88, 66	44	4	6
Stop, divide, (0–3), and (5-11)	44 is set, index 4	4	4
(22), 33, 11, 40	22	0	3
11, 33, (22), 40	22	0	2
11, (22), 33, 40	22	1	2
Stop, divide, 0 and (2-3)	22 is set, index 1	1	1
11, no call is needed since it's a single value	11 is set, index 0	0	0
(33), 40	33	0	1
Stop, divide, 1 (index)	33 is set, index 0	0	0
40, no call is needed for single value	40 is set, index 1	0	0
(90), 77, 60, 99, 55, 88, 66	90	0	6
66, 77, 60, 99, 55, 88, (90)	90	0	6
66, 77, 60, (90), 55, 88, 99	90	3	6
66, 77, 60, 88, 55, (90), 99	90	5	6
Stop, divide (0-4), and 6	90 is set, index 5	5	5
(66), 77, 60, 88, 55	66	0	4
55, 77, 60, 88, (66)	66	0	4
55, (66), 60, 88, 77	66	1	4
55, 60, (66), 88, 77	66	1	2
Stop, divide (0-1), (3-4)	66 is set, index 2	2	2
(55), 60	55	0	1
Stop, divide 1	55 is set, index 0	0	0
Stop, 66 single value	60 is set, index 1		
(88), 77	88	0	1
77, (88)	88	1	1
Stop, divide 0	88 is set, index 1		
Stop, 77 is single value	77 is set, index 0		

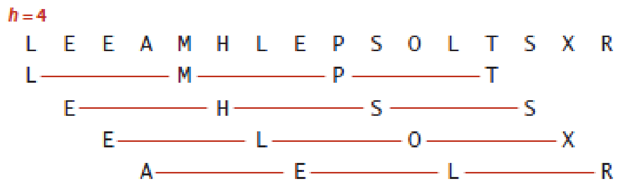
Now, to collect the sorted data, going backward and place the outcoming indices as appear.

Sorted data: 11, 22, 33, 40, 44, 55, 60, 66, 77, 88, 90

4- Shell Sort

To exhibit the value of knowing properties of elementary sorts, we next consider a fast algorithm based on insertion sort. Insertion sort is slow for large unordered arrays because the only exchanges it does involve adjacent entries, so items can move through the array only one place at a time.

The idea is to rearrange the array to give it the property that taking every h th entry (starting anywhere) yields a sorted subsequence. Such an array is said to be h-sorted.



An *h*-sorted sequence is *h* interleaved sorted subsequences

However, h can take the following values h= {1, 4, 13, 40, 121, 364, 1093, … }.
So, these values are subject to the following formula: h = 3*h + 1, under the condition that h starts at the largest increment less than N/3 and decreasing to 1, h uses the sequence of decreasing values ½(3^k-1) where k is h position in h array.

Note that for 100000 random Doubles; Shell is 600 times faster than Insertion sort.

Example: consider the following dada, sort is using shell sort.

16	4	3	13	5	6	8	9	10	11	12	17	15	18	19	7	1	2	14	20

And consider h= {1, 2, 4, 7}.

Sol:
Firstly, we have to decide the h values as follows:
N=20.

We can divide this into three smaller slices:

16	4	3	13	5	6	8
9	10	11	12	17	15	18
19	7	1	2	14	20	

Reassembling the array we now have:

9	4	1	2	5	6	8	16	7	3	12	14	15	18	19	10	11	13	17	20

Slicing this into five we get:

9	4	1	2
5	6	8	16
7	3	12	14
15	18	19	10
11	13	17	20

Again we sort each column to give:

5	3	1	2
7	4	8	10
9	6	12	14
11	13	17	16
15	18	19	20

Logically reassembling, we now have the dataset:

5	3	1	2	7	4	8	10	9	6	12	14	11	13	17	16	15	18	19	20

We can now slice the array into 10:

5	3
1	2
7	4
8	10
9	6
12	14
11	13
17	16
15	18
19	20

Sorting the columns gives us:

1	2
5	3
7	4
8	6
9	10
11	13

12	14
15	16
17	18
19	20

Reassembling we get:

1	2	5	3	7	4	8	6	9	10	11	13	12	14	15	16	17	18	19	20

The majority of the elements are now near to where they should be, and the last pass of the algorithm is a conventional insertion sort, that gives.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

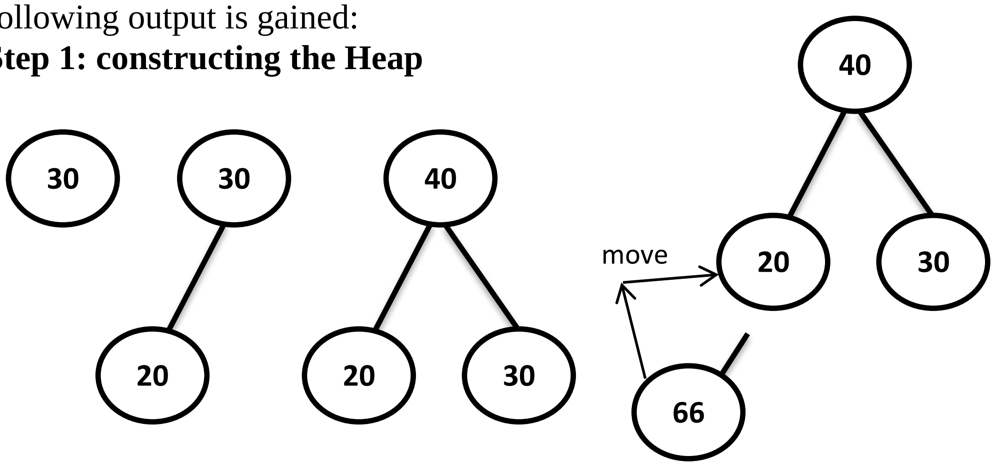
5- Heap Sort

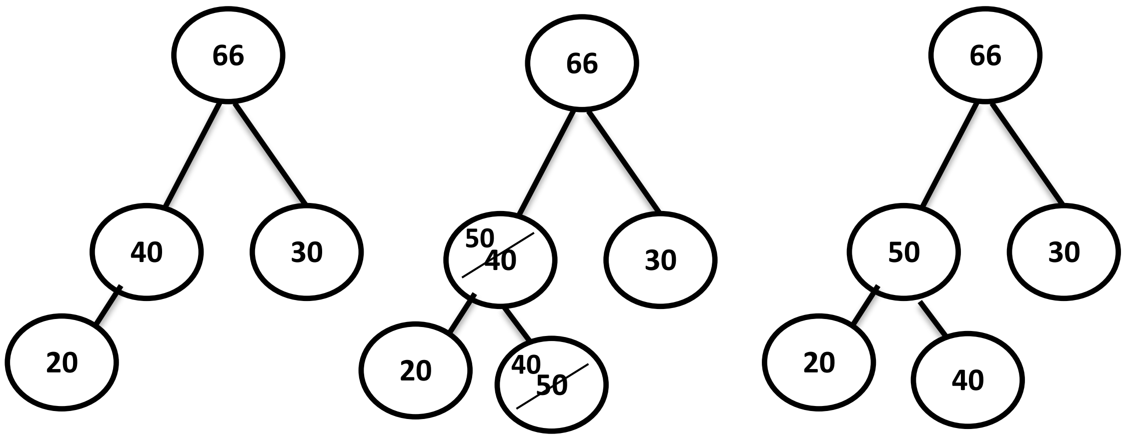
A heap is a complete binary tree, in which each node satisfies the heap condition.
Heap condition: The key of each node is greater than or equal to the key in its children. Thus the root node will have the largest key value (max heap).
MaxHeap: Suppose H is a complete binary tree with n elements. Then H is called a heap or maxheap, if the value at N is greater than or equal to the value at any of the children of N.
MinHeap: The value at N is less than or equal to the value at any of the children of N.

The operations on a heap
(i) New node is inserted into a Heap
(ii) Deleting the Root of a Heap, in which reheaping is required.

Example: consider the following data, sort them using max heap.
30, 20, 40, 66, 50

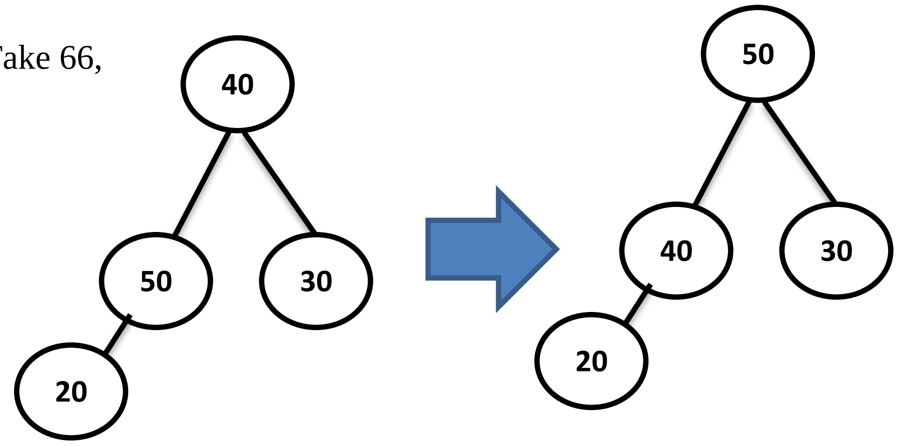
Sol:
Now, we are building the binary tree with considering the heap condition, the following output is gained:
Step 1: constructing the Heap



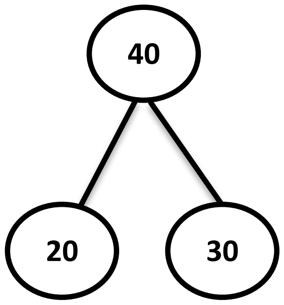


Step 2: Destructing the Heap

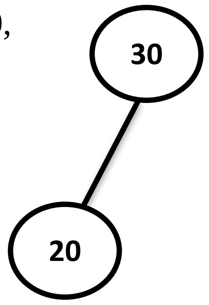
Take 66,



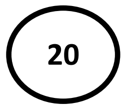
Take 50,



Take 40,



Take 30



Take 20

Then, the sorted data is: 66, 50, 40, 30, 20

6- Radix Sort

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

Complexity of Radix Sort

Worst case, O(n^2), best case O(n log n). In other words, radix sort performs well only when the number s of digits in the representation of the Ai's is small.

Another drawback of radix sort is that one may need d*n (d is the number of digits, n is input array size) memory locations. This comes from the fact that all the items may be "sent to the same pocket" during a given pass. This drawback may be minimized by using linked lists rather than arrays to store the items during a given pass. However, one will still require 2*n memory locations.

Example:

Sort the following numbers: 348, 143, 361, 423, 538, 128, 321, 543, 366.

Sol:

(a) In the first pass, the units digits are sorted into pockets. (The pockets are pictured upside down, so 348 is at the bottom of pocket 8.) The cards are collected pocket by pocket, from pocket 9 to pocket 0. (Note that 361 will now be at the bottom of the pile and 128 at the top of the pile.) The cards are now reinput to the sorter.

(b) In the second pass, the tens digits are sorted into pockets. Again the cards are collected pocket by pocket and reinput to the sorter.

(c) In the third and final pass, the hundreds digits are sorted into pockets.

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass.

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass.

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass.

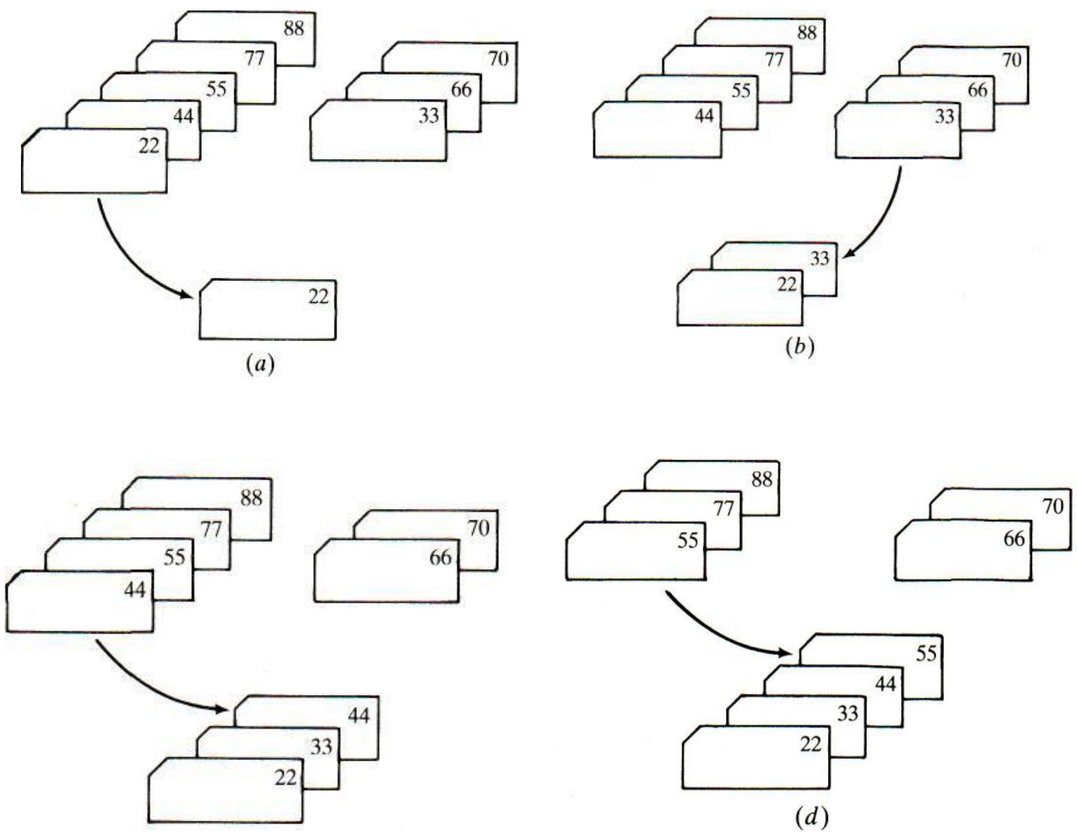
Now, collecting the sorted data column by column, we get:

128, 143, 321, 348, 361, 366, 423, 538, 543.

7- Merge Sort

Combing the two lists is called as merging. For example A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with n = r + s elements is called merging. After combing the two lists the elements are sorted by using the following merging algorithm:

Suppose one is given two sorted lists of data. The lists are merged as in following figure, that is, at each step, the two front data are compared and the smaller one is placed in the combined list. When one of the lists is empty, all of the remaining data in the other deck are put at the end of the combined list.

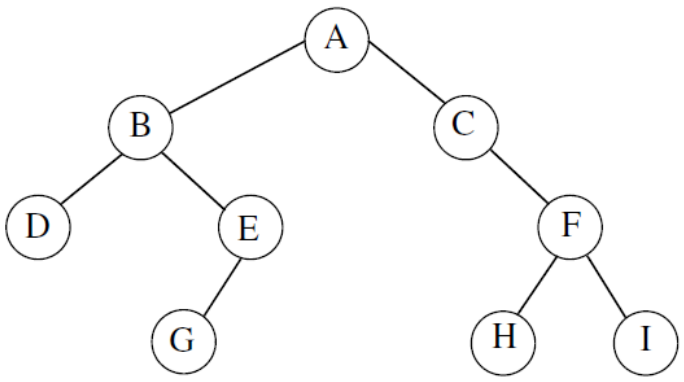


Complexity of the algorithm

The total computing time = O(n log2 n). The disadvantages of using mergesort is that it requires two arrays of the same size and type for the merge phase.

8- Binary Tree Sort

A binary tree is a finite set E, that is empty, or contains an element r and whose other elements are partitioned in two binary trees, called left and right subtrees. r is called the root (racine) of the tree. The elements are called the nodes of the tree. A node without a successor (a tree whose left and right subtrees are empty) is called a leaf.



Example of binary tree.

Traversal of a Binary Tree

1. Preorder or depth-first order

(i) visit the root (ii)traverse the left subtree (iii)traverse the right subtree

2. Inorder or symmetric order

(i) traverse the left subtree (ii)visit the root (iii)traverse the right subtree

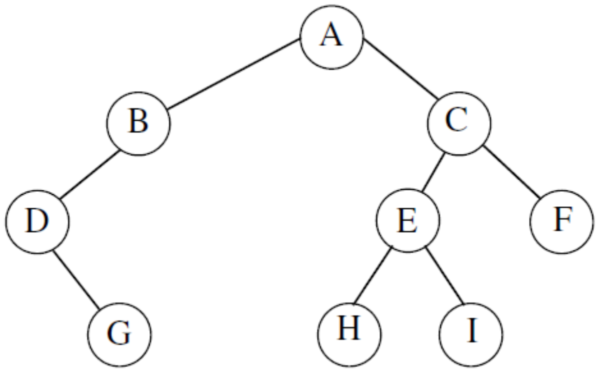
3. Postorder

(i) traverse the left subtree (ii)traverse the right subtree (iii)visit the root

4. Level-order or breadth-first order

Visit all the nodes at the same level, starting with level 0

Example: traverse the following tree with all types of tree traversals.



Sol:

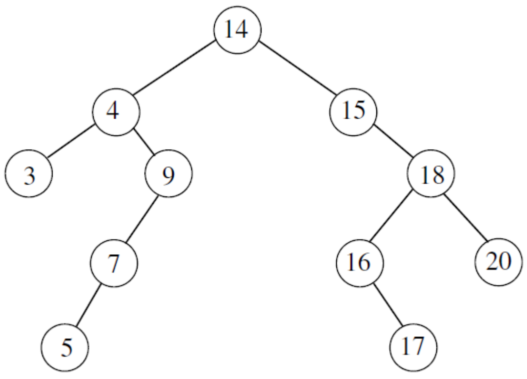
Preorder:	A B D G C E H I F
Inorder:	D G B A H E I C F
Postorder:	G D B H I E F C A
By level:	A B C D E F G H I

Search Tree

A search tree is a special case of a binary tree. Each node contains a key and should be a dominant condition that existed all the way during the construction of this tree, that is, consider that P is the current node, L is the left child and R is the right child, then, this condition should hold:

$$\text{Key}(L) < \text{key}(P) \leq \text{key}(R)$$

As the following example:



Inorder: 3 4 5 7 9 14 15 16 17 18 20

Application: Sorting
Input: 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17
Processing: Build the tree
Result: Traverse in inorder

Example: consider the following data, sort them using binary search tree,
Data= 10, 5, 23, 65, 8, 2, 9, 65, 43, 90, 3, 5

Note: To be solved by students during the lecture.

Previous Sorting Methods (1 to 8) are called internal sort which means the data should resides in the memory to be sorted and all the indices of the aeeay A(i) should be available as requested, but, however, in case of the memory does not hold all the data at the same time, we resort to the external sort which are:

1- Sorting with Disk
2- Sorting with Tapes

External Sort

One method for sorting a file is to load the file into memory, sort the data in memory, then write the results. When the file cannot be loaded into memory due to resource limitations, an external sort applicable. We will implement an external sort using replacement selection to establish initial runs, followed by a polyphase merge sort to merge the runs into one sorted file.

1- Sorting with Disks

We will first illustrate merge sort using disks. The following example illustrate the concept of sorting with disks The file F containing 600 records is to be sorted. The main memory is capable of sorting of 1000 records at a time. The input file F is stored on one disk and we have in addition another scratch disk. The block length of the input file is 500 records.
We see that the file could be treated as 6 sets of 1000 records each. Each set is sorted and stored on the scratch disk as a run. These 5 runs will then be merged as follows.
Allocate 3 blocks of memory each capable of holding 500 records. Two of these buffers B1 and B2 will be treated as input buffers and the third B3 as the output buffer. We have now the following:

1) 6 run R1, R2, R3, R4, R5, R6 on the scratch disk.
2) 3 buffers B1,B2 and B3

- Read 500 records from R1 into B1.
- Read 500 records from R2 into B2.
- Merge B1 and B2 and write into B3.

- When B3 is full- write it out to the disk as run R11.
- Similarly merge R3 and R4 to get run R12.
- Merge R5 and R6 to get run R13.

Thus, from 6 runs of size 1000 each, we have now 3 runs of size 2000 each. The steps are repeated for steps R11 and R12 to get a run of size 4000.
This run is merged with R13 to get a single sorted run of size 6000.

2- Sorting with Tapes
Sorting with tapes is essentially similar to the merge sort used for sorting with disks. The differences arise due to the sequential access restriction of tapes. This makes the selection time prior to data transmission an important factor, unlike seek time and latency time. Thus is sorting with tapes we will be more concerned with arrangement of blocks and runs on the tape so s to reduce the selection or across time.

Example
A file of 6000 records is to be sorted. It is stored on a tape and the block length is 500. The main memory can sort unto a 1000 records at a time. We have in addition 4 search tapes T1-T4.

Search Algorithms

Searching refers to the operation of finding the location of a given item in a collection of items. The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise, the main objective of the algorithm is minimize the number of comparisons during search in order to find ITEM.

1- Sequential Search
This is the most natural searching method. The most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one.

```
Algorithm
        SEQUENTIAL SEARCH
        INPUT :  List of Size N.  Target Value T
        OUTPUT : Position of T in the list-1
        BEGIN
           Set FOUND = false
           Set I := 0
            While (I <= N) and (FOUND is false)
                IF List[i] ==t THEN
                        FOUND = true
                ELSE
                        I = I+1
            IF FOUND==false THEN
                    T is not present in the List
        END
```

Example: consider the following data, find the item 40 and calculate the number of comparisons needed. Data= 43, 54, 2, 4, 65, 76, 40, 23, 40, 24, 56.

2- Binary Search

Suppose DATA is an array which is sorted in increasing numerical order. Then there is an extremely efficient searching algorithm, called binary search, which can be used to find the location LOC of a given ITEM of information in DATA.

The binary search algorithm applied to our array DATA works as follows. During each stage of our algorithm, our search for ITEM is reduced to a segment of elements of DATA: DATA[BEG], DATA[BEG + 1], DATA[BEG + 2], ... DATA[END].

Note that the variable BEG and END denote the beginning and end locations of the segment respectively. The algorithm compares ITEM with the middle element DATA[MID] of the segment, where MID is obtained by MID = INT((BEG + END) / 2).

If DATA[MID] = ITEM, then the search is successful and we set LOC: = MID. Otherwise a new segment of DATA is obtained as follows:

(a) If ITEM < DATA[MID], then ITEM can appear only in the left half of the segment: DATA[BEG],DATA[BEG + 1],..... ,DATA[MID - 1]
So we reset END := MID - 1 and begin searching again.

(b) If ITEM > DATA[MID], then ITEM can appear only in the right half of the segment: DATA[MID + 1], DATA[MID + 2],....,DATA[END]
So we reset BEG := MID + 1 and begin searching again.
Initially, we begin with the entire array DATA; i.e. we begin with BEG = 1 and END = n, If ITEM is not in DATA, then eventually we obtain END< BEG

This condition signals that the search is unsuccessful, and in this case we assign LOC=NULL. Here NULL is a value that lies outside the set of indices of DATA. We now formally state the binary search algorithm.

Example: Let DATA be the following sorted 13-element array:
DATA: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99
We apply the binary search to DATA for different values of ITEM.
(a) Suppose ITEM = 40. The search for ITEM in the array DATA is pictured in where the values of DATA[BEG] and DATA[END] in each stage of the algorithm are indicated by parenthesis and the value of DATA[MID] by a bold. Specifically, BEG, END and MID will have the following successive values:

(1) Initially, BEG = 1 and END 13. Hence,
MID = INT[(1 + 13) / 2] = 7 and so DATA[MID] = 55

(2) Since 40 < 55, END = MID – 1 = 6. Hence,
MID = INT[(1 + 6) / 2] = 3 and so DATA[MID] = 30

(3) Since 40 > 30, BEG = MID + 1 = 4. Hence,
MID = INT[(4 + 6) / 2] = 5 and so DATA[MID] = 40

The search is successful and LOC = MID = 5.
(1) (11), 22, 30, 33, 40, 44, **55**, 60, 66, 77, 80, 88, (99)
(2) (11), 22, **30**, 33, 40, (44), 55, 60, 66, 77, 80, 88, 99
(3) 11, 22. 30, (33), **40**, (44), 55, 60, 66, 77, 80, 88, 99 [Successful]

Complexity of the Binary Search Algorithm
The complexity is measured by the number of comparisons f(n) to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size in half. Hence we require at most f(n) comparisons to locate ITEM where $f(n) = [\log_2 n] + 1$.

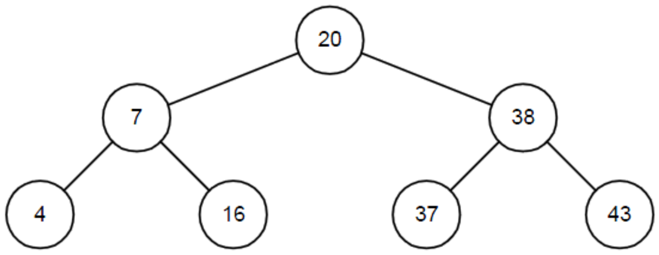
That is the running time for the worst case is approximately equal to $\log_2 n$. The running time for the average case is approximately equal to the running time for the worst case.

Limitations of the Binary Search Algorithm

The algorithm requires two conditions:
(1) The list must be sorted and
(2) One must have direct access to the middle element in any sublist.

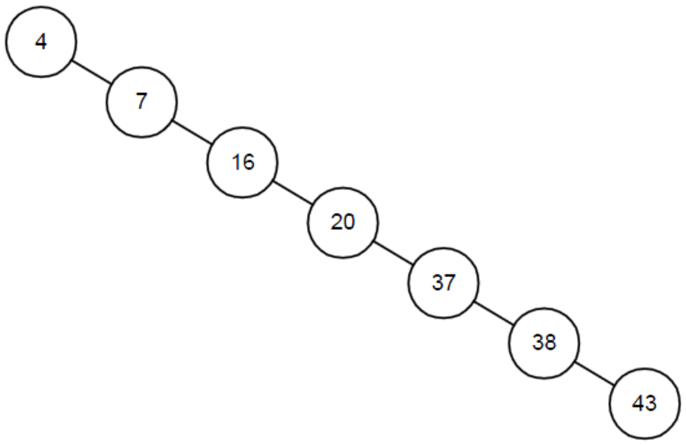
3- Binary Search Tree

This method is very effective, as each iteration reduced the number of items to search by one-half.



A Binary Search Tree

To search a tree for a given value, we start at the root and work down. For example, to search for 16 in above figure, we first note that 16 < 20 and we traverse to the left child. The second comparison finds that 16 > 7, so we traverse to the right child. On the third comparison, **we succeed**.



An Unbalanced Binary Search Tree

Each comparison results in reducing the number of items to inspect by one-half. In this respect, the algorithm is similar to a binary search on an array. However, this is true only if the tree is balanced. The above figure shows another tree containing the same values. While it is a binary search tree, its behavior is more like that of a linked list, with search time increasing proportional to the number of elements stored.
So, note that binary search tree differs from binary search which is applied using array.

Example: calculate the number of comparisons for finding item 43 in the tree before the last one.

Example: using binary search tree, find the item 59 in the following data,
Data: 43, 54, 89, 21, 3, 90, 54, 32, 3, 59, 89, 78, 45.

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)