NOTES

Subject Name: Data Structures & Analysis of Algorithms **Subject Code**: KCA-205 **Semester**: II

UNIT-3

Insertion Sort, Selection Sort, Bubble Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time: Counting Sort and Bucket Sort.

Terminology used with Graph, Data Structure for Graph Representations: Adjacency Matrices, Adjacency List, Adjacency. Graph Traversal: Depth First Search and Breadth First Search, Connected Component.

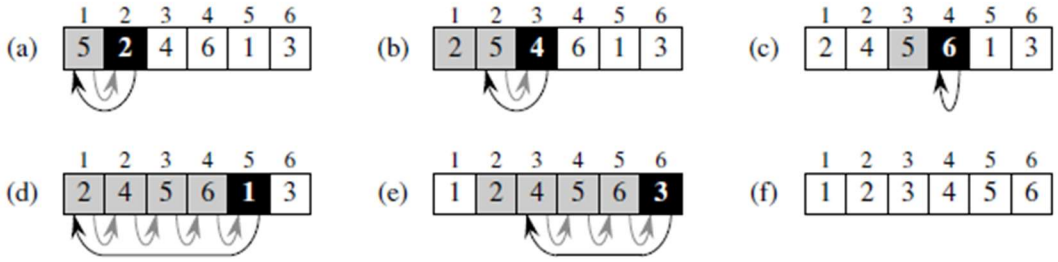
INSERTION-SORT*(A)*

The pseudo code for the algorithm is give below.

Pseudo code:

```
INSERTION-SORT(A)
1       for j ← 2 to length[A]
2               do key ← A[ j ]
3               ✂Insert A[ j ] into the sorted sequence A[1 . . j − 1].
4               i ← j − 1
5               while i > 0 and A[i ] > key
6                       do A[i + 1] ← A[i ]
7                          i ←i − 1
8               A[i + 1] ← key
```

Example



Bubble Sort

- Bubble sort is similar to selection sort in the sense that it repeatedly finds the largest/smallest value in the unprocessed portion of the array and puts it back.
- However, finding the largest value is not done by selection this time.
- We "bubble" up the largest value instead.
- Compares adjacent items and exchanges them if they are out of order.
- Comprises of several passes.
- In one pass, the largest value has been "bubbled" to its proper position.
- In second pass, the last value does not need to be compared.
- Traverse a collection of elements
    - Move from the front to the end
    - "Bubble" the largest value to the end using pair-wise comparisons and swapping

Bubble sort algorithm

```
void bubbleSort (int S[ ],  int length) {
    bool isSorted = false;
    while(!isSorted)
    {
            isSorted = true;
            for(int i = 0; i<length; i++)
    {
```

```
                if(S[i] > S[i+1])
                {
                    int temp = S[i];
                    S[i] = S[i+1];
                    S[i+1] = temp;
                    isSorted = false;
                }
            }
            length--;
        }
}
```

Selection Sort

- **Selection sort** is a sorting algorithm which works as follows:
  - Find the minimum value in the list
  - Swap it with the value in the first position
  - Repeat the steps above for remainder of the list (starting at the second position)

Algorithm

```
void selectionSort(int numbers[], int array_size) {
  int i, j, T, min, count;
  for (i = 0; i < array_size; i++) {
    min = i;
    for (j = i + 1; j < array_size; j++) {
      if (numbers[j] < numbers[min]) {
        min = j; }
    }
```

```
    T = numbers[min];
    numbers[min] = numbers[i];
    numbers[i] = T;
  }
}
```

Time Complexity of Selection Sort

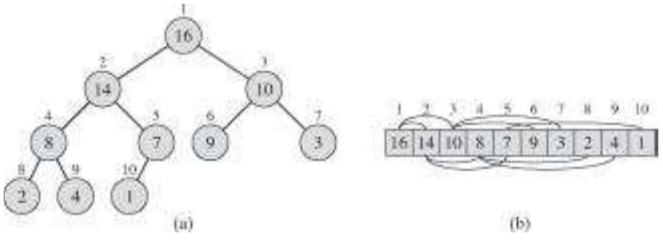
- The total number of comparisons is :-

  (N-1)+(N-2)+(N-3)+…… +1= N(N-1)/2

- We can ignore the constant 1/2
- We can express N(N-1) as $N^2 – N$
- We can ignore N as well since N^2 grows more rapidly than N, making our algorithm $O(N^2)$

Heap Sort

The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. The root of the tree is A[1], and given the index *i* of a node, we can easily compute the indices of its parent, left child, and right child:



PARENT (i) => ***return*** [i /2]
LEFT (i) => ***return*** 2*i*
RIGHT (i) => ***return*** 2i+ 1
On most computers, the LEFT procedure can compute 2i in one instruction by simply shifting the binary representation of i left by one bit position.

Similarly, the RIGHT procedure can quickly compute **2i + 1** by shifting the binary representation of **i** left by one bit position and then adding in a 1 as the low-order bit. The PARENT procedure can compute ***[i/2]*** by shifting **i** right one bit position. Good implementations of heapsort often implement these procedures as "macros" or "inline" procedures.

There are two kinds of binary heaps: **max-heaps** and **min-heaps**.

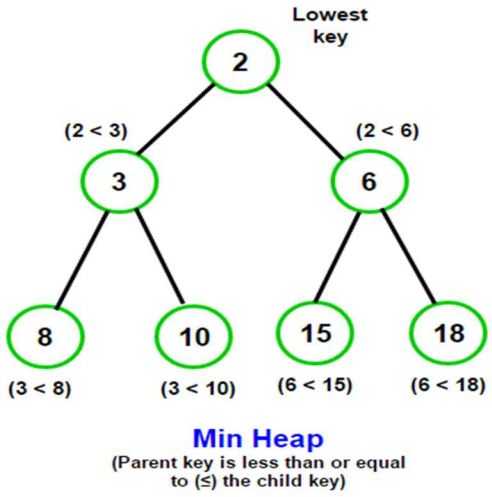
max-heap

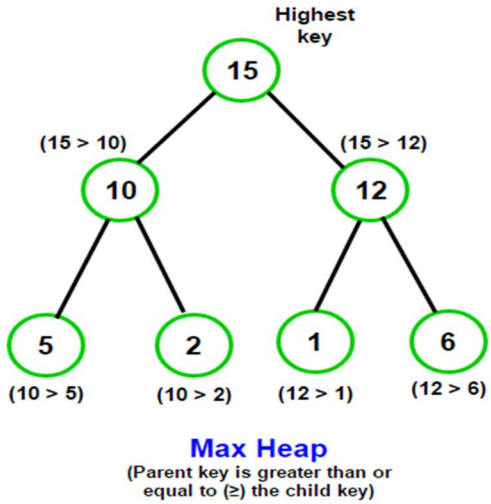
- In a ***max-heap,*** the ***max-heap property*** is that for every node ***i*** other than the root, **A[PARENT(i)] >= A[i] ,** that is, the value of a node is at most the value of its parent.

Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

min-heap

- A ***min-heap*** is organized in the opposite way; the ***min-heap property*** is that for every node **i** other than the root, **A[PARENT(i)<=A[i] ,** The smallest element in a min-heap is at the root.





Important points to note

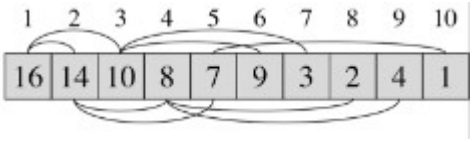
- The height of a node in a heap is the number of edges on the longest simple downward
- path from the node to a leaf and
- The height of the heap is the height of its root.
- Height of a heap of n elements which is based on a complete binary tree is **O(log n).**

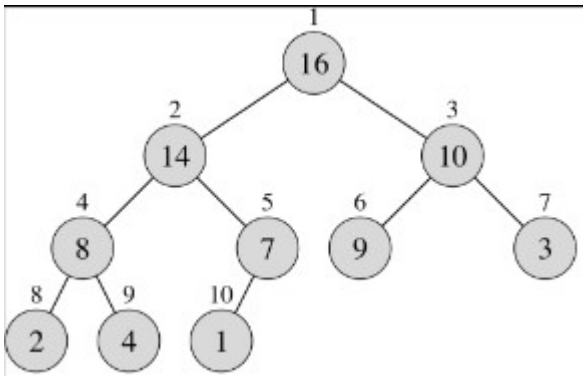
Maintaining the heap property

MAX-HEAPIFY lets the value at A[i] "float down" in the max-heap so that the subtree rooted at **index i** obeys the max-heap property.

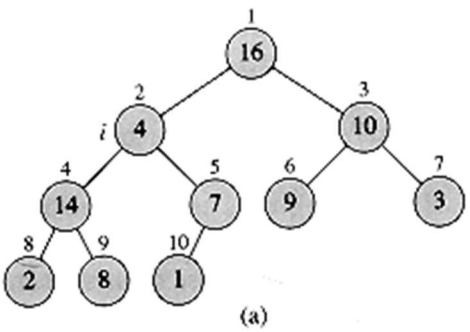
```
MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4     then largest ← l
5     else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7     then largest ← r
8  if largest ≠ i
9     then exchange A[i] ↔ A[largest]
10         MAX-HEAPIFY(A, largest)
```

At each step, the largest of the elements A[i], A[LEFT(i)], and A[RIGHT(i)] is determined, and its index is stored in *largest.* If A[i] is largest, then the subtree rooted at node i is already a max-heap and the procedure terminates. Otherwise, one of the two children has the largest element, and A[i] is swapped with A[largest], which causes node i and its children to satisfy the max-heap property. The node indexed by *largest,* however, now has the original value A[i], and thus the subtree rooted at *largest* might violate the max-heap property. Consequently,
we call MAX-HEAPIFY recursively on that subtree.

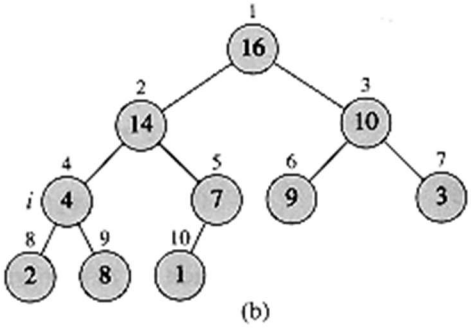




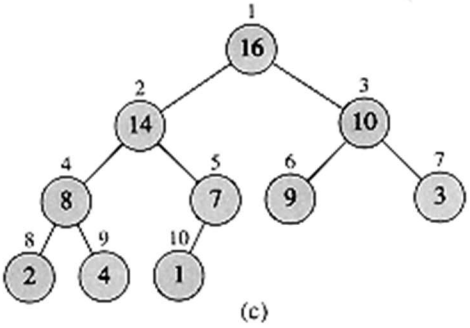
The action of MAX-HEAPIFY (A, 2), where *heap-size* = 10.
(a) The initial configuration, with A [2] at node i = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in
(b) by exchanging A [2] with A[4], which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY (A,4) now has i = 4. After swapping A[4] with A[9].
(c), node 4 is fixed up, and the recursive call MAX-HEAPIFY(A, 9) yields no further change to the data structure.



(a)



(b)



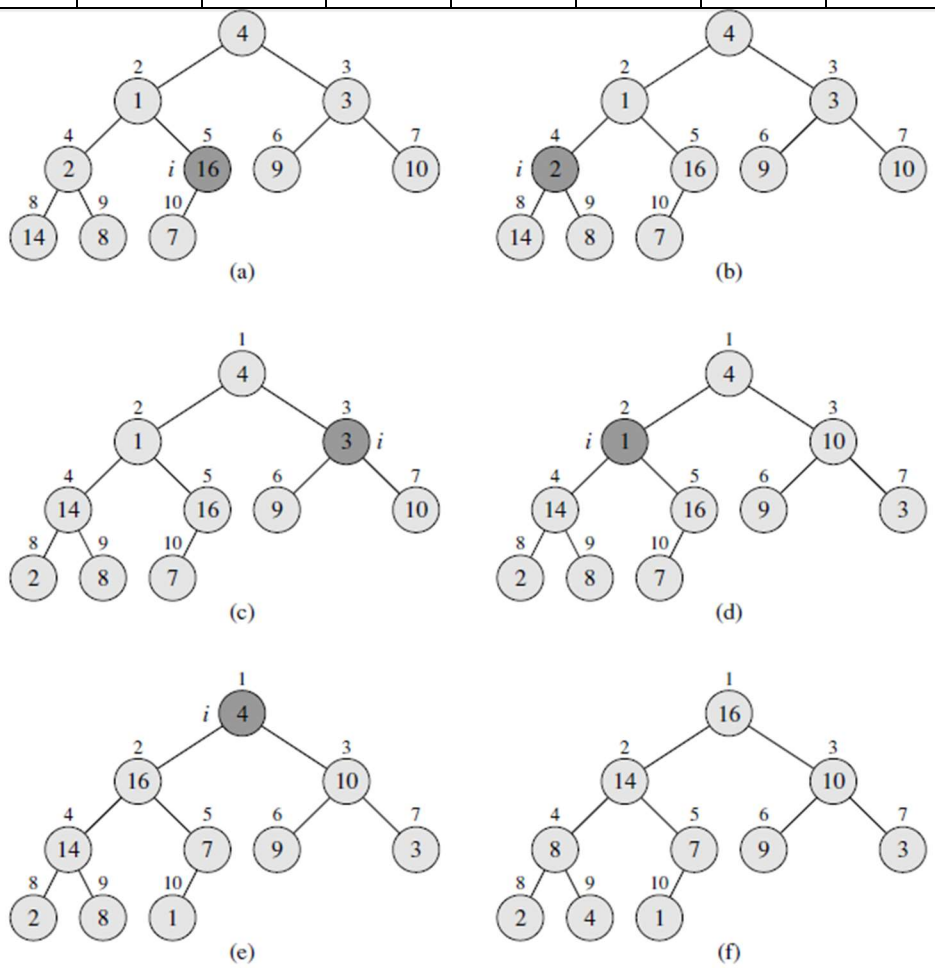
(c)

The running time of MAX-HEAPIFY by the recurrence can be described as
T (n) < = T (2n/3) + O (1)
The solution to this recurrence is **T(n)=O(log n)**

Building a heap

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

4	1	3	2	16	9	10	14	8	7



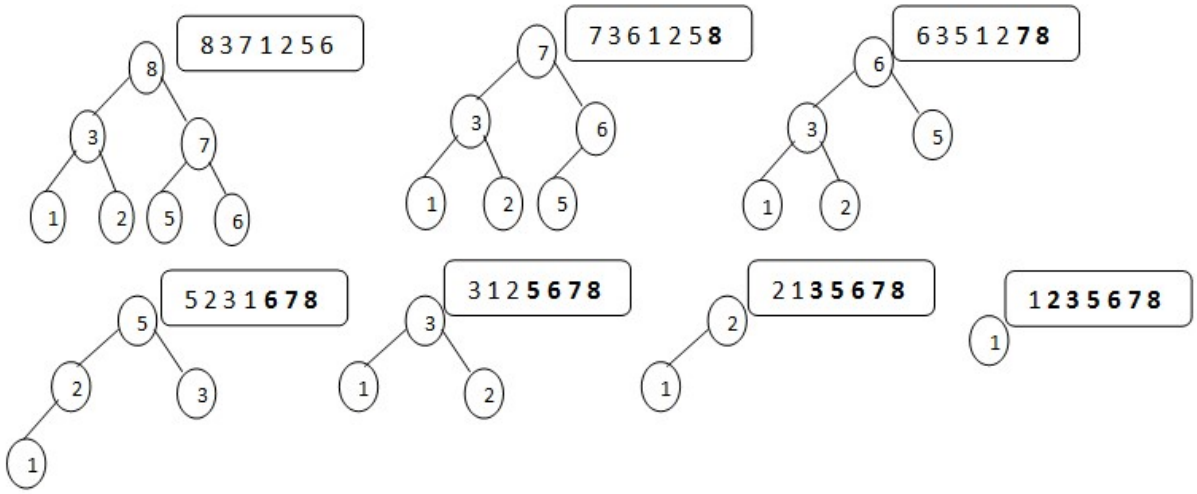
We can derive a tighter bound by observing that the time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small. Our tighter analysis relies on the properties that an n-element heap has height [log n] and at most [n/2h+1] nodes of any height h.

The total cost of BUILD-MAX-HEAP as being bounded is T(n)=O(n)

The Heap Sort Algorithm

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3      do exchange A[1] ↔ A[i]
4         heap-size[A] ← heap-size[A] − 1
5         MAX-HEAPIFY(A, 1)
```

Example:- The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)



The HEAPSORT procedure takes time **O(n log n)**, since the call to BUILD-MAX- HEAP takes time **O(n)** and each of the **n - 1** calls to MAX-HEAPIFY takes time **O(log n).**

Review of Sorting

So far we have seen a number of algorithms for sorting a list of numbers in ascending order. Recall that an ***in-place*** **sorting algorithm is one that uses no additional array storage** (however, we allow Quicksort to be called in-place even though they need a stack of size *O*(log *n*) for keeping track of the recursion). **A sorting algorithm is *stable* if duplicate elements remain in the same relative position after sorting.**

Slow Algorithms: Include Bubble Sort, Insertion Sort, and Selection Sort. These are all simple $\Theta(n^2)$ in-place sorting algorithms. Bubble Sort and Insertion Sort can be implemented as stable algorithms, but Selection Sort cannot (without significant modifications).
Mergesort: Mergesort is a stable $\Theta(n \log n)$ sorting algorithm. The downside is that Merge Sort Is the only algorithm of the three that requires additional array storage, implying that it is not an in-place algorithm.
Quicksort: Widely regarded as the *fastest* of the fast algorithms. This algorithm is $O(n \log n)$ in The *expected case*, and $O(n^2)$ in the worst case. The probability that the algorithm takes asymptotically longer (assuming that the pivot is chosen randomly) is extremely small for large *n*.It is an(almost) in-place sorting algorithm but is not stable.
Heapsort: Heapsort is based on a nice data structure, called a *heap*, which is a fast priority queue. Elements can be inserted into a heap in $O(\log n)$ time, and the largest item can be extracted in $O(\log n)$ time. (It is also easy to set up a heap for extracting the smallest item.) If you only want to extract the *k* largest values, a heap can allow you to do this is $O(n + k \log n)$ time. It is an in-place algorithm, but it is not stable.

Lower Bounds for Comparison-Based Sorting:
Can we sort faster than $O(n \log n)$ time?
We will give an argument that if the sorting algorithm is based solely on making comparisons between the keys in the array, then it is impossible to sort more efficiently than $(n \log n)$ time.
Such an algorithm is called a *comparison-based* sorting algorithm, and includes all of the algorithms given above .Virtually all known general purpose sorting algorithms are based on making comparisons, so this is not a very restrictive assumption. This does not preclude the possibility of a sorting algorithm whose actions are determined by other types of operations, for example, consulting the individual bits of numbers, performing arithmetic operations, indexing into an array based on arithmetic operations on keys. We will show that any *comparison-based* sorting algorithm for a input sequence $a1; a2; : : : ; an$ must make at least $(n \log n)$ comparisons in the worst-case.

Decision Tree Argument: In order to prove lower bounds, we need an abstract way of modeling "any possible" comparison-based sorting algorithm, we model such algorithms in terms of an abstract model called a *decision tree*. In a *comparison-based* sorting algorithm only comparisons between the keys are used to determine the action of the algorithm.

let us assume that $n = 3$, and let's build a decision tree for Selection Sort. Recall that the algorithm consists of two phases. The first finds the smallest element of the entire list, and swaps it with the first element. The second finds the smaller of the remaining two items, and swaps it with the

second element. Here is the decision tree (in outline form). The first comparison is between *a*1 and *a*2. The possible results are:

*a*1 <= *a*2**:** Then *a*1 is the current minimum. Next we compare *a*1 with *a*3 whose results might be either:

*a*1 <=*a*3**:** Then we know that *a*1 is the minimum overall, and the elements remain in their original positions. Then we pass to phase 2 and compare *a*2 with *a*3. The possible results are:

*a*2 <=*a*3**:** Final output is ***a*1; *a*2; *a*3.**

*a*2 > *a*3**:** These two are swapped and the final output is ***a*1; *a*3; *a*2.**

*a*1 > *a*3**:** Then we know that *a*3 is the minimum is the overall minimum, and it is swapped with *a*1. The we pass to phase 2 and compare *a*2 with *a*1 (which is now in the third position of the array) yielding either:

*a*2 <=*a*1**:** Final output is ***a*3; *a*2; *a*1.**

*a*2 > *a*1**:** These two are swapped and the final output is ***a*3; *a*1; *a*2.**

*a*1 > *a*2**:** Then *a*2 is the current minimum. Next we compare *a*2 with *a*3 whose results might be either:

*a*2 <=*a*3**:** Then we know that *a*2 is the minimum overall. We swap *a*2 with *a*1, and then pass to phase 2, and compare the remaining items *a*1 and *a*3. The possible results are:

*a*1 <=*a*3**:** Final output is ***a*2; *a*1; *a*3.**

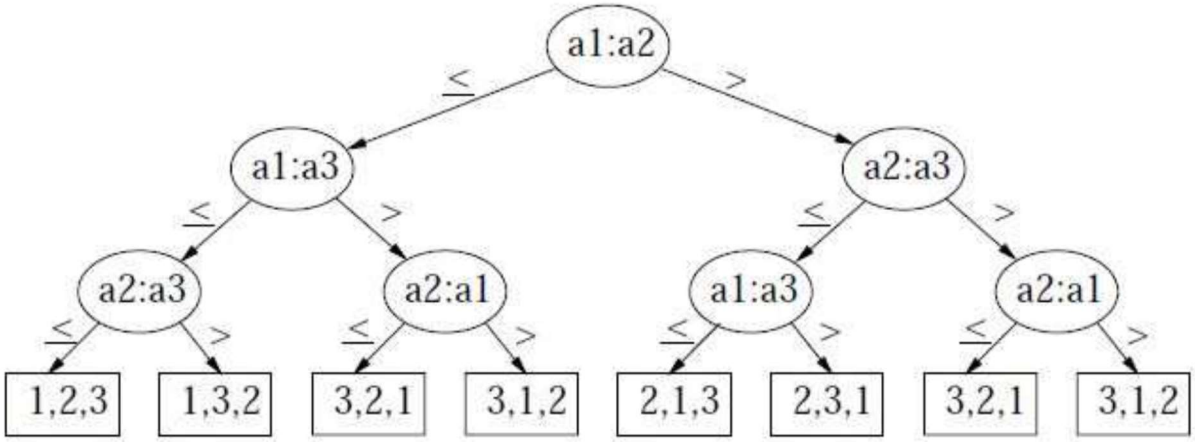
*a*1 > *a*3**:** These two are swapped and the final output is ***a*2; *a*3; *a*1.**

*a*2 > *a*3**:** Then we know that *a*3 is the minimum is the overall minimum, and it is swapped with *a*1. We pass to phase 2 and compare *a*2 with *a*1 (which is now in the third position of the array) yielding either:

*a*2<= *a*1**:** Final output is ***a*3; *a*2; *a*1.**

*a*2 > *a*1**:** These two are swapped and the final output is ***a*3; *a*1; *a*2.**

The final decision tree is shown below.



Decision Tree for Selection Sort on 3 keys.

Using Decision Trees for Analyzing Sorting: Consider any sorting algorithm. Let $T(n)$ be the maximum number of comparisons that this algorithm makes on any input of size n. Notice that the running time of the algorithm must be at least as large as $T(n)$, since we are not counting data movement or other computations at all. The algorithm defines a decision tree. Observe that the height of the decision tree is exactly equal to $T(n)$, because any path from the root to a leaf corresponds to a sequence of comparisons made by the algorithm.

A lower bound for the worst case

The length of the longest path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Theorem: Any comparison-based sorting algorithm has worst-case running time (n log n).

From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of **height *h*** with ***l* reachable leaves** corresponding to a **comparison sort on *n* elements**. Because each of the ***n*!** permutations of the input appears as some leaf, we have ***n*! ≤ *l*.** Since a binary tree of height ***h*** has no more than 2^h leaves, we have

$n! \le l \le 2^h$,

which, by taking logarithms, implies

$h \ge \lg(n!)$ (since the lg function is monotonically increasing)

$= \Omega(n \lg n)$ (by Stirling formula)

Counting sort

Counting sort assumes which each one of the n input elements is an integer within the range 1 to k, for some integer k. Whenever k = O(n), the sort runs within linear time.

Fundamental concept: determine, for every input element x, the number of elements less than x. This enables one to determine x's position in the sorted array.

The code makes use of the following arrays:

- **A[1..n] is the input array, with length[A] = n.**
- **B[1..n] holds the sorted output.**
- **C[1..k] provides temporary storage.**

```
COUNTING-SORT(A, B, k)
 1  for i ← 0 to k
 2       do C[i] ← 0
 3  for j ← 1 to length[A]
 4       do C[A[j]] ← C[A[j]] + 1
 5  ▷ C[i] now contains the number of elements equal to i.
 6  for i ← 1 to k
 7       do C[i] ← C[i] + C[i − 1]
 8  ▷ C[i] now contains the number of elements less than or equal to i.
 9  for j ← length[A] downto 1
10       do B[C[A[j]]] ← A[j]
11          C[A[j]] ← C[A[j]] − 1
```

Analysis of Counting Sort

The **for** loop of lines 1–2 takes time ***Θ(k),*** the **for** loop of lines 3–4 takes time ***Θ(n)***, the **for** loop of lines 6–7 takes time ***Θ(k),*** and the **for** loop of lines 9–11 takes time ***Θ(n).*** Thus, the overall time is ***Θ(k+n).*** In practice, we usually use counting sort when we have ***k = O(n),*** in which case **the running time is *Θ(n).*** Counting sort beats the lower bound of *Ω(n* lg *n)* proved in previous lecture because **it is not a comparison sort.**

Example for counting sort:

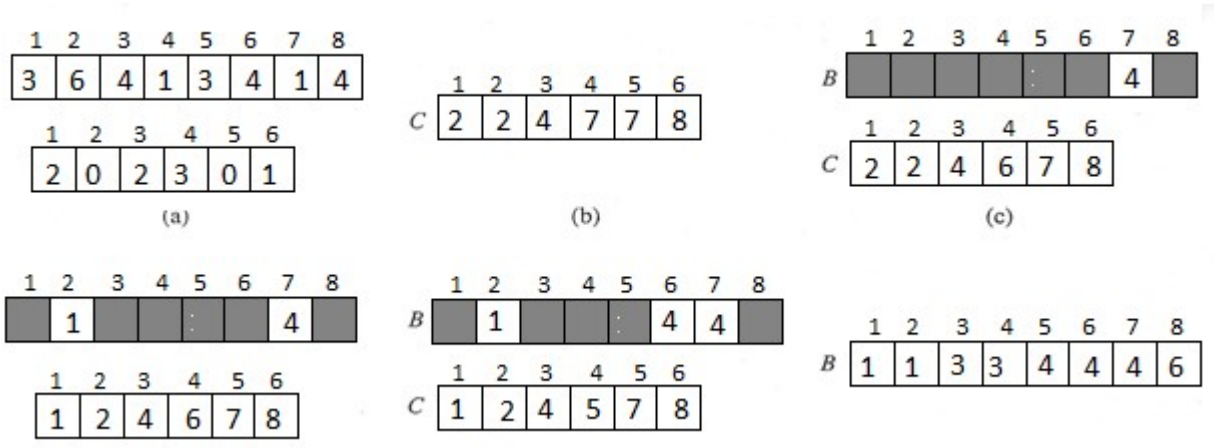


Fig : The operation of COUNTING_SORT on an input array A[1..8], where each element of A is positive integer no larger than k = 6.

- The array A and the auxiliary array C after line 4.
- The array C after line 7
- The output array B and the auxiliary array C after one, two and three iterations of the loop in line 9-11 respectively. Only the lightly shaded elements of array B have been filled in.
- The final sorted output array B.

Example-2

Input Data

0	4	2	2	0	0	1	1	0	1	0	2	4	2

Count Array

0	1	2	3	4
5	3	4	0	2

Sorted Data

0	0	0	0	0	1	1	1	2	2	2	2	4	4

Why Counting sort is Stable Sort

A sorting algorithm is said to be **stable** if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

The Counting sort is a stable sort i.e., multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array.

Suppose that the for-loop in line 9 of the Counting sort is rewritten:

9 for *j* ← 1 to *n*

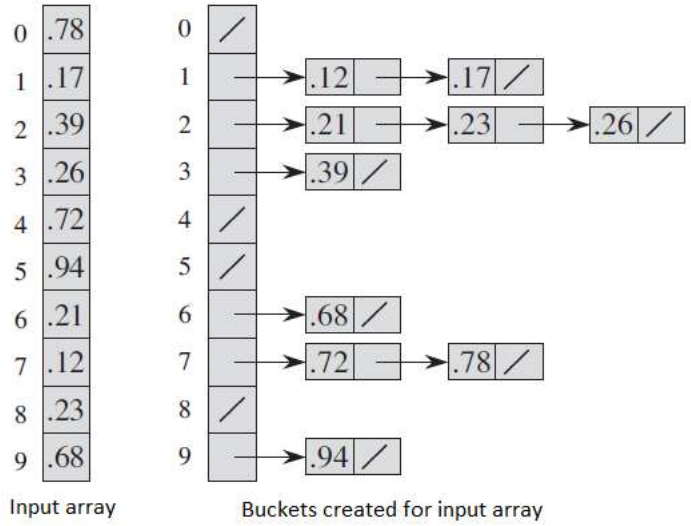
then the stability no longer holds. Notice that the correctness of argument in the CLR does not depend on the order in which array $A[1 . . n]$ is processed. The algorithm is correct no matter what order is used. In particular, the modified algorithm still places the elements with value k in position $c[k - 1] + 1$ through $c[k]$, but in reverse order of their appearance in $A[1 . . n]$

Bucket Sort

The idea of bucket sort is to divide the interval [0, 1) into ***n*** **equal-sized** subintervals, or ***buckets***, and then distribute the ***n*** input numbers into the buckets. Since the inputs are uniformly distributed over [0, 1), we don't expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each. Our code for bucket sort assumes that the input is an n-element array A and that each element $A[i]$ in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array $B[0 \,..\, n-1]$ of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists.

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3        do insert A[i] into list B[⌊n A[i]⌋]
4  for i ← 0 to n − 1
5        do sort list B[i] with insertion sort
6  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Example -1



The operation of BUCKET-SORT. **(a)** The input array $A[1 \,..\, 10]$. **(b)** The array $B[0 \,..\, 9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket ***i*** holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

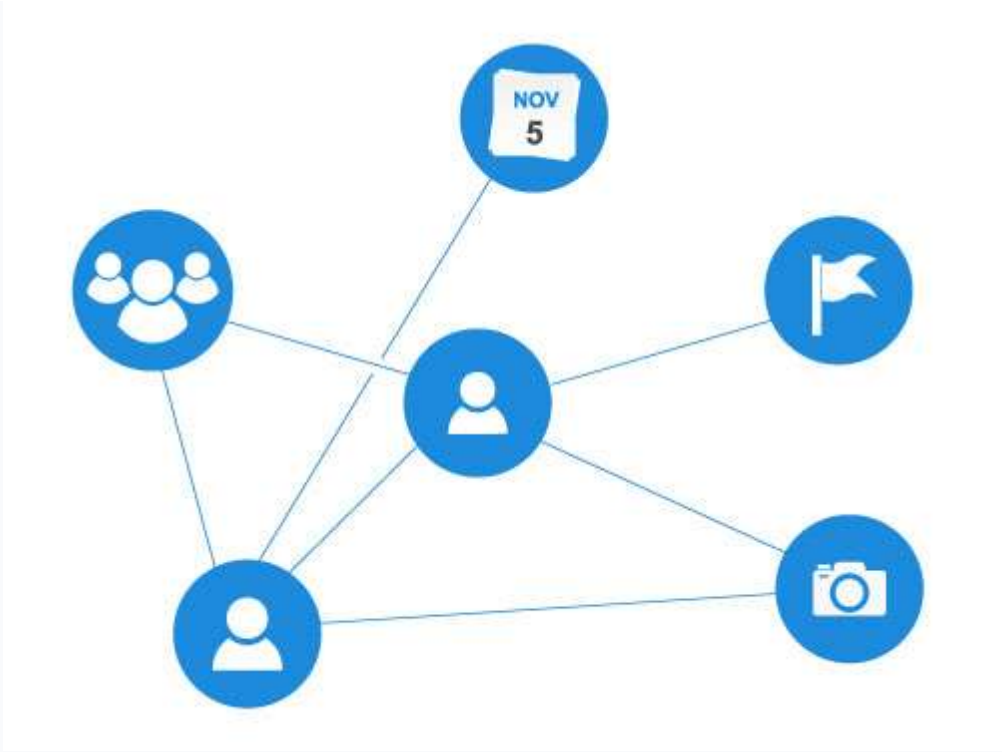
To analyze the running time,
Observe that all lines except line 5 take *O(n)* time in the worst case. It remains to balance the total time taken by the *n* calls to insertion sort in line 5.
To analyze the cost of the calls to insertion sort, let *ni* be the random variable denoting the number of elements placed in bucket *B*[*i*]. **Since insertion sort runs in quadratic time.** But we are expecting that very less number of elements will be lie into each bucket so insertion sort will take constant time
We conclude that the **expected time for bucket sort is Θ*(n)*.** Thus, the entire bucket sort algorithm runs in linear expected time.

Graph Data Structure

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this by means of an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

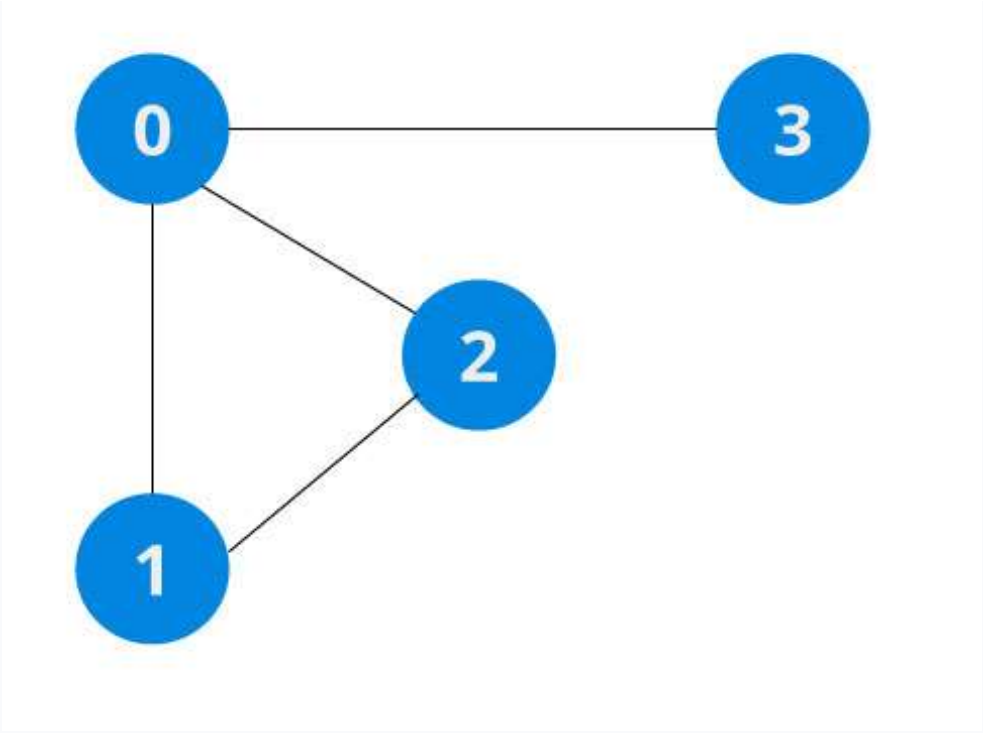
Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page etc., a new edge is created for that relationship.



All of facebook is then, a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V,E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)



In the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

Some Terminology of graph

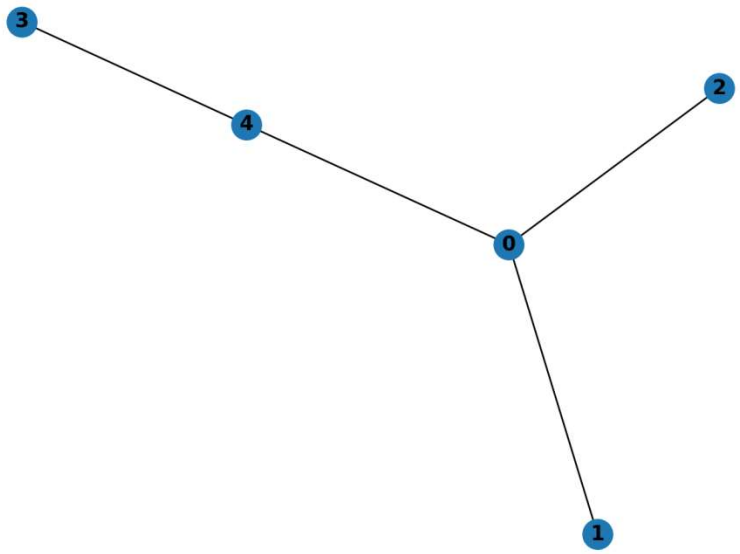
- A **Graph** $G(V, E)$ is a data structure that is defined by a set of Vertices (V) and a set of Edges (E).
- **Vertex** (v) or node is an indivisible point, represented by the lettered components on the example graph below
- An **Edge** (vu) connects vertex v and vertex u together.
- The **Degree** $d(v)$ of vertex v, is the count of edges connected to it.

- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessary mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

- **Space Complexity** is shown as *Θ(G)* and represents how much memory is needed to hold a given graph
- **Adjacency Complexity** shown by *O(G)* is how long it takes to find all the adjacent vertices to a give vertex *v*.

Representing Graph Data Structures

When you're representing a graph structure *G*, the vertices, *V*, are very straight forward to store since they are a set and can be represented directly as such. For instance, for a graph of vertices:



V = {0,1,2,3,4}

Graph Representation

Things get a little more interesting when you start storing the Edges, *E*. Here there are two common structures that you can use represent and navigate the edge set:

- Adjacency Matrix
- Adjacency List

We're going to take a look at a simple graph and step through each representation of it. We will assess each one according to its **Space Complexity** and **Adjacency Complexity.**

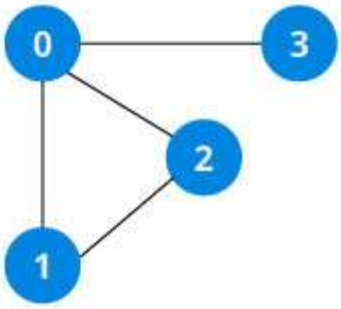
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element `a[i][j]` is 1, it represents that there is an edge connecting vertex i and vertex j.

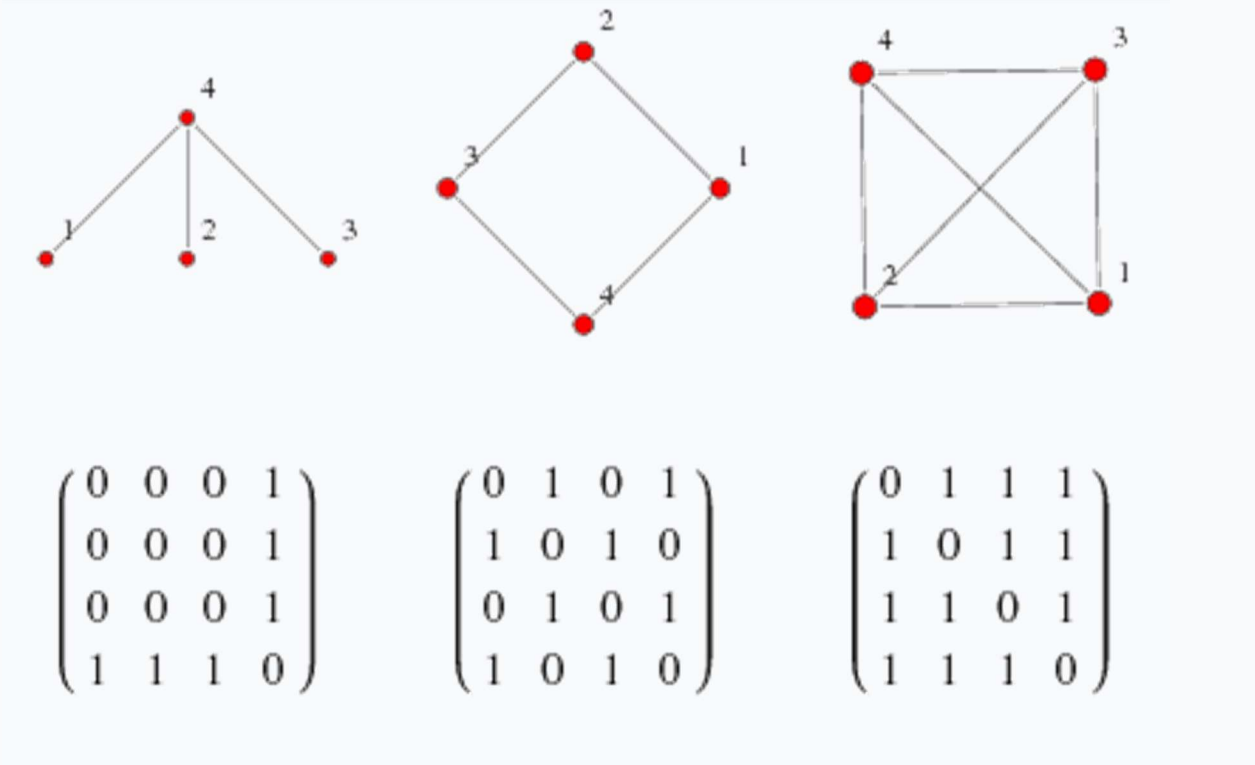
The adjacency matrix for the graph we created above is



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.



Pros

- Highly interpretable. It's symmetrical unless its a directed graph and you can neatly store edge values in each matrix entry.
- Decent **adjacency complexity** of $O(G) = |V|$. In order to find all vertices adjacent to v, we need to scan their whole row with the adjacency matrix.

Cons

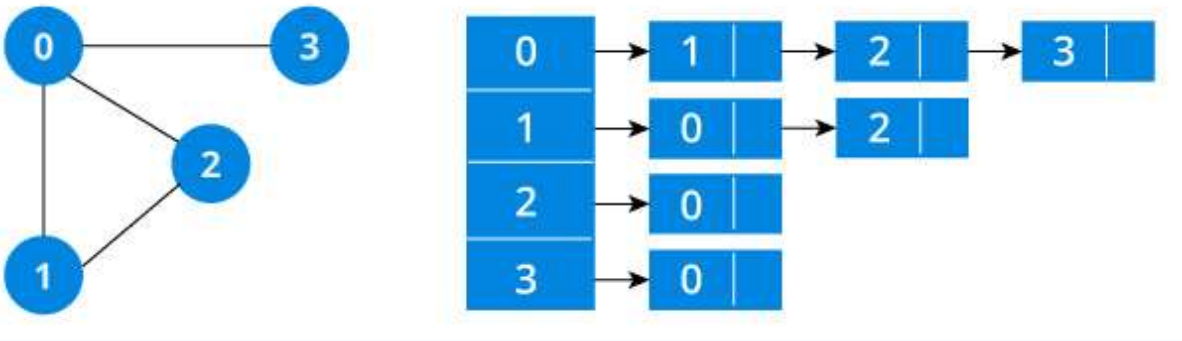
- Terrible **space complexity** of $\Theta(G) = |V|^2$. Here we are storing every possible vertex permutation of length 2, including each vertex paired with itself. This is more than double the maximum vertex set of possible combinations, ($|V|$ *choose 2)*

2. Adjacency List

An adjacency list represents a graph as an array of linked list.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Pros:

- It's like a dictionary!
- **Space complexity** of $\Theta(G) = |V| + 2|E|$ We have a list for every vertex and in total these lists will store $2|E|$ elements since each edge will appear in both vertex lists.
- Great **adjacency complexity.** For a given vertex v, $O(G)$ is equal to $d(v)$ the Degree of v. When looking for all adjacent neighbors this is in fact the best possible value here.

Optimal Representation

Choosing the optimal data structure to represent a given graph G is actually dependent on the density of edges within G. This can be roughly summarized as follows.

- If $|E| \approx |V|$ ie there are about as many edges as there are vertices then G is considered **Sparse** and an adjacency list is preferred.
- If $|E| \approx (|V|$ *choose 2)* ie is close to the maximum number of edges in G then it is considered **Dense** and the adjacency matrix is preferred.

Graph Operations

The most common graph operations are:

- *Check if element is present in graph*
- *Graph Traversal*
- *Add elements(vertex, edges) to graph*

Finding path from one vertex to another

Depth First Search (DFS) Algorithm

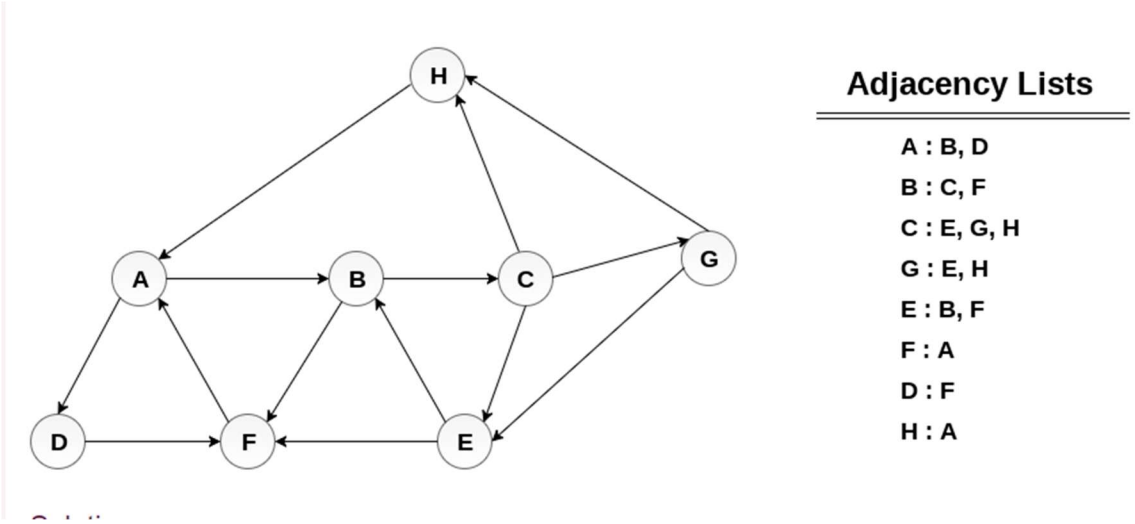
Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)

- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]
- **Step 6:** EXIT



Push H onto the stack

1. STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

1. Print H
2. STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

1. Print A
2. Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

1. Print D
2. Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

1. Print F
2. Stack : B

Pop the top of the stack i.e. B and push all the neighbours

1. Print B
2. Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

1. Print C
2. Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

1. Print G
2. Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

1. Print E
2. Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

1. H → A → D → F → B → C → G → E

Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

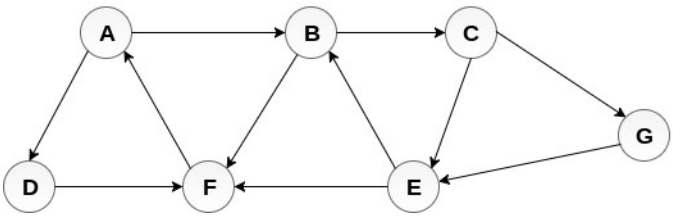
The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state)
  for each node in G
- **Step 2:** Enqueue the starting node A
  and set its STATUS = 2
  (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until
  QUEUE is empty
- **Step 4:** Dequeue a node N. Process it
  and set its STATUS = 3
  (processed state).
- **Step 5:** Enqueue all the neighbours of
  N that are in the ready state
  (whose STATUS = 1) and set
  their STATUS = 2
  (waiting state)
  [END OF LOOP]
- **Step 6:** EXIT

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

Lets start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.

1. QUEUE1 = {A}
2. QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

1. QUEUE1 = {B, D}
2. QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

1. QUEUE1 = {C, F}
2. QUEUE2 = { A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are

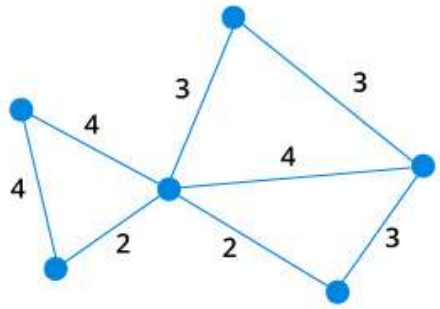
1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

8. Remove G from QUEUE1, all of G's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited

3. QUEUE1 = {}
4. QUEUE2 = {A, B, D, C, F, E,G}

1

Start with a weighted graph

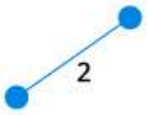


2

Choose a vertex

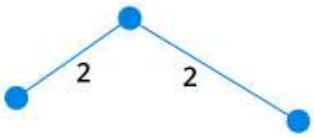
3

Choose the shortest edge from this vertex and add it



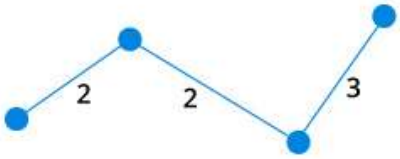
4

Choose the nearest vertex not yet in the solution



5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



6

Repeat until you have a spanning tree