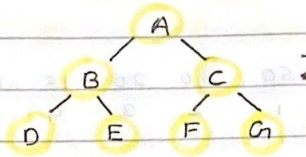
Algorithms & Data Structures

- Heap and Heap Sort

• Binary tree representation of array.

(is filled out level by level)



⇌

A	B	C	D	E	F	G
1	2	3	4	5	6	7

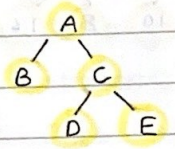
If node is at index - i

its left child is at - 2 × i

its right child is at - 2 × i + 1

its parent is at - $[i/2]$

If child of a parent is missing →



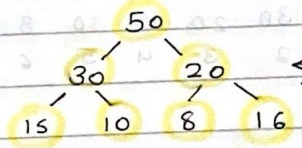
⇌

A	B	C	–	–	D	E
1	2	3	4	5	6	7

• Heap

↳ Heap is a complete binary tree. Two types of heaps –

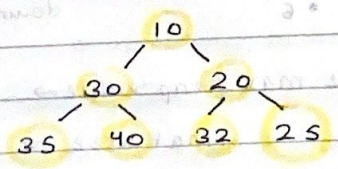
• Max Heap – Root node has a greater value than all its children. It has to be a complete binary tree



⇌

H	50	30	20	15	10	8	16
	1	2	3	4	5	6	7

• Min Heap – Root node has a smaller value than all its children. It also has to be a complete binary tree.



⇌

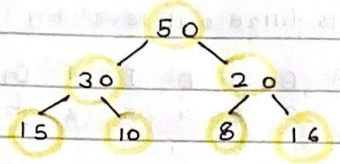
H	10	30	20	35	40	32	25
	1	2	3	4	5	6	7

↳ Insert function for heap

Insert looks different for max heap and for min heap!

→ Insert for Max Heap

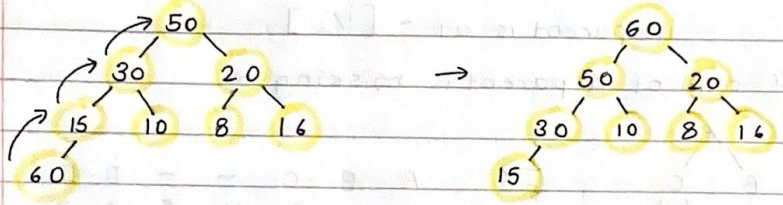
Suppose I want to insert 60 in the following binary tree.→



H	50	30	20	15	10	8	16	60
	1	2	3	4	5	6	7	8

The new heap would look as follows →

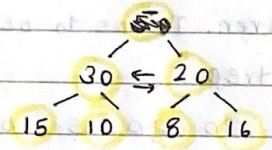
! The direction of adjustment is upwards.



The time taken to insert an element can be O(1) to O(log N)

↳ Delete Function for Max Heap

Suppose I want to delete 50 in the following binary tree →



H	~~50~~	30	20	15	10	8	16
	1	2	3	4	5	6	7

To maintain structure of a complete binary tree, the last element is moved first →

H	16	30	20	15	10	8
	1	2	3	4	5	6

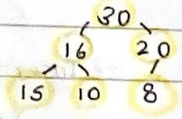
! The direction of adjustment is downwards

Now we need to maintain the max heap value →

30 > 20, thus →

H	30	16	20	15	10	8
	1	2	3	4	5	6

Final tree →



The time taken to ~~insert~~ delete an element is O(log N) → depends on the height

After deletion - heap size went from 7 to 6

H	30	16	20	15	10	8	50
	1	2	3	4	5	6	7

The 50 can be stored in the empty space (! but not in the heap). If this continues and the elements continue to be deleted, it would be in order such that

H Empty | 8 10 15 16 20 30 50

which is a sorted array!

This is called Heap Sort

Heap Sort

Steps for Heap sort →

1. For a given set of elements, create a heap (by inserting elements 1 by 1.
2. Once heap is formed - delete the elements → which will form a sorted array.

The time complexity is ~~O(log n)~~. $O(n \log_2 n)$

Heapify

The process of creating a heap data structure from a binary tree.

- Start from the first non-leaf node or the last element, which is set as the largest. The children are compared and if the ~~leftChildIndex~~ parent is lesser than children, the current element is swapped.

The time complexity is $O(n)$ - which is much lesser than inserting in heap which is $O(n \log n)$.

Priority Queue

A special type of queue in which element is associated with a priority value - and elements are served on the basis of their priority.

We use heaps as the data structure to implement priority queues.

It can be the biggest number or the smallest number that is given most priority.

Heap operations in Pseudocode

```
Key          // a type, usually int for types of values in heap
int N        // number of elements in array or heap
Key a[]      // heap array of size N containing items of type Key.
```

! The heap array a[] and N will be encapsulated inside heap object.

```
insert (Key x)
    a[++N] = x
    siftUp (N)
// siftUp from position k - the key or node value at position k
// may be greater than that of its parent at k/2.
// k is a position in the heap array h.
siftUp (int k)
    v = a[k]
    a[0] = ∞
    while (v > a[k/2]):
        a[k] := a[k/2]
        k = k/2
    a[k] = v
// key of node at position k may be less than that of its
// children and may need to be moved down some levels
// key is a position in the heap array h.
siftDown (int k)
    v = a[k]
    while (2k ≤ N)   // * while node at pos k has a left child  (* node)
        j = 2k
        if (j < N ∧ a[j] < a[j+1])
            ++j
        if (v ≥ a[j]) break
        a[k] = a[j]
        k = j
    a[k] = v
```

```
Key remove()
    a[0] = a[1]
    a[1] = a[N-1]
    siftDown(1)
    return a[0]
```

Recursive Version of Heapify

```
Max-Heapify (A, i)
  l = left(i)
  r = right(i)
  if l ≤ A.heap-size and A[l] > A[i]
      largest = l
  Else largest = i
  if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
  if largest ≠ i
    exchange A[i] with A[largest]
    Max-Heapify (A, largest)
```

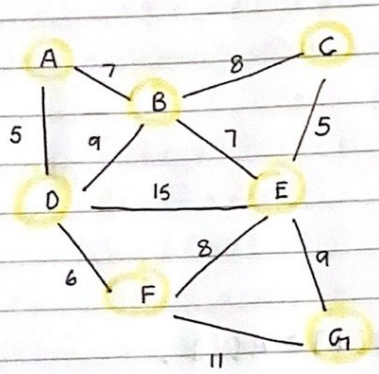
! To note →

- k = heap position
- a[k] = index in array [] for graphs
- dist[a[k]] = priority in heap node at position k in heap array.

Graphs on next page →

- Graphs

- Can represent number of problems such as length or cost of optical fibre between network points, road distances, etc.
- Graph is stored as follows in computer memory →



Text file for this graph ↴

This is storing graph in secondary storage

7	11	
1	2	7
1	4	5
2	3	8
2	4	9
2	5	7
3	5	5
4	5	15
4	6	6
5	6	8
5	7	9
6	7	11

The first row tells us there as 7 vertices and 11 edges. The rows after consist of the edge between the nodes and the weight at the end. For example Edge of A to B has weight 7.

- To store this graph in primary storage - we can use adjacency lists or adjacency matrixes.
  - ↳ We say adjacency because it records which vertex is next or connected to any other vertex.

Adjacency Matrix

-> A square matrix of N X N size where N is the number of nodes in the graph and it is used to represent the connections between the edges of a graph.

-> adj [u, v] represents the weight of the edges between vertices u and v

→ Example of Adjacency Matrix for given graph →

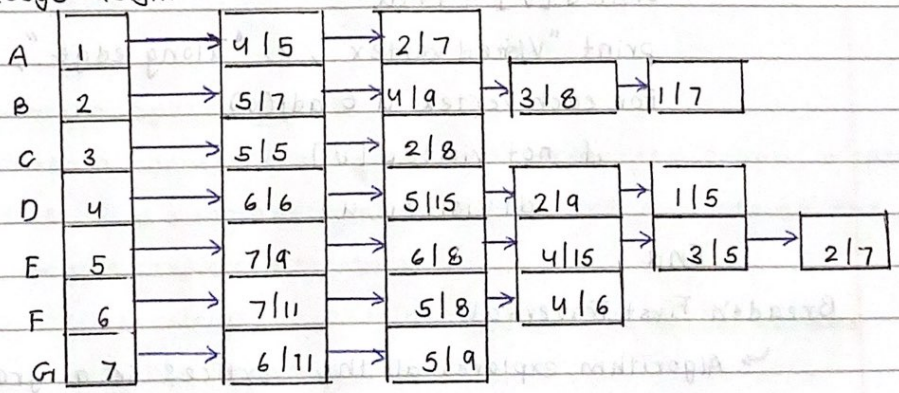
	A 1	B 2	C 3	D 4	E 5	F 6	G 7
A 1	0	7	0	5	0	0	0
B 2	7	0	8	9	7	0	0
C 3	0	8	0	0	5	0	0
D 4	5	9	0	0	15	6	0
E 5	0	7	5	15	0	8	9
F 6	0	0	0	6	8	0	11
G 7	0	0	0	0	9	11	0

- Adjacency matrix is inefficient for sparse graphs !

Adjacency Lists

→ A data structure used to represent a graph where each node in the graph stores a list of its neighbouring vertices.
↳ in a bidirectional graph, each edge is represented by two linked list nodes.

→ Example of Adjacency List for given graph →
(edge weight is also recorded)



Graph Traversal

- Graph is a complex data structure and requires specific algorithms to process each edge and node.
- The traversal algorithms are as follows →

Depth First Traversal

↳ Algorithm starts at the root node (selecting some arbritary node as the root node in the case of graph) and explores as far as possible along each branch before backtracking.

↳ Time complexity = O(V+E) where V is number of edges and E is the number of edges in the graph.

↳ To avoid processing a node more than once, we use a boolean visited array.

↳ Pseudocode for DFS → (uses recursion)

```
Graph:: DF(vertex s)
Begin
    id = 0
    for v = 1 to V
        visited[v] = 0
    dfVisit(0,s)
End

Graph:: dfVisit(vertex prev, vertex v)
Begin
    visited[v] = ++id
    print "Visited vertex ", v, "along edge ", prev, "-", v
    For each vertex u ∈ adj(v)
        if not visited[u]
            dfVisit(v,u)
End
```

Breadth First Traversal

↳ Algorithm explores all the vertices in a graph at the current depth level before moving on to the vertices at the next depth level. Starts at a specified vertex and visits all its neighbours.

↳ Time Complexity = O(V+E)

↳ To avoid processing a node more than once, we use a boolean visited array.

↳ We use queues for this algorithm.

↳ Pseudocode for BFS →

```
Graph :: BF(Vertex s)
 Begin
   Queue q
   id = 0
   for v = 1 to V
      visited[v] = 0
   q.enQueue(s)
   while (not q.isEmpty())
      v = q.deQueue()
       if (notvisited[v])
           visited[v] = ++id
           for each vertex u ∈ adj(v)
            if (notvisited[u])
                q.enQueue(u)
             end for
       end if
   end while
 End
```

Minimum Spanning Tree

- Given a connected, weighted and undirected graph, a spanning tree is a subgraph that is a tree and connects all the ~~graphs~~ vertices together.
  - ↳ The weight is obtained by adding all the edge weights in the tree.
- A minimum spanning tree (MST) is a spanning tree with weight less than or equal equal to the weight of the spanning tree.
  - ↳ The one with the lowest total cost.
  - ↳ There can be multiple ~~sp~~ MSTs depending on which vertice is chosen.

Prim's Algorithm MST

- A greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph.
  - ↳ means a subset of the edges that forms a tree including every vertex where the total weight of all the edges in the tree is minimized.
- Steps for Prims →
  1. Input - A non-empty connected weighted graph with vertices V and edges E.
  2. Initialise - $V_{new} = \{x\}$ where x is an arbitrary node from V, $E_{new} = \{\}$
  3. Repeat - until $V_{new} = V$:
     - ↳ Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if multiple edges with same weight, any of them can be picked)
  4. Output - V_{new} and E_{new} describe a minimal spanning tree
- Data structures for Prims (Adjacency lists) →
  - → Node structure - graph edges in the linked lists
  - → Array of linked lists to store graph in memory
    Node[] adj;
  - → Array int dist[] to record current distance of a vertex from MST.
  - → Array int parent[] to store the MST.
  - → Heap H to find the next vertex nearest to the MST.
  - → Array int hPos[] which records the position of any vertex with the heap array a[]. Thus → vertex v is in position hPos[v] in a[].
- Time Complexity of Prims → ~~$O(E \log_2 V)$~~ $O(E * \log(E))$ where E is the number of edges and V is number of vertexes.

! This algorithm relies on priority queue

! Choice of starting node can affect MST output

↳ Pseudocode for Prims MST

```
Prims-Lists (vertex s)
Begin
    // G = (V, E)
    For each v ∈ V
        dist[v] = ∞
        parent[v] = 0        // treat 0 as special null vertex
        hPos[v] = 0          // indicates that v ∉ heap
    h = new Heap (|V|, hPos, dist)   // priority queue (heap) is empty.
    h.insert (s)                     // s will be the root of MST

    while (not h.isEmpty())          // should repeat |V|-1 times
        v = h.remove()               // add v to the MST
        dist[v] = -dist[v]           // marks v as now in MST
        for each u ∈ adj(v)          // examine each neighbour u of v
            if weight(v,u) < dist[u]
                dist[u] = weight(v,u)
                parent[u] = v
                if u ≠ h
                    h.insert(u)
                else
                    h.siftUp(hPos[u])
                end if
            end if
        end for
    end while
    return parent
End
```

! parent[] stores the mst

! siftUp and siftDown are used to adjust hPos[].

! Only make sure to include vertices that are not already in MST

Kruskal's Algorithm MST

- A greedy algorithm that sorts all the edges of the given graph in increasing order. Then it adds the edges and nodes from decreasing order and keeps adding till no cycle is formed.
- Steps for Kruskals →
  1. Sort all the edges in non-decreasing order of their weight.
  2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle forms - don't use the edge. - uses Union Find
  3. Repeat step 2 until there are (V-1) edges in the spanning tree.
- Data Structures for Kruskals →
  - Union Find Algorithm → It implements find and union operations on a disjoint set data structure. It finds the root parent of an element and determines whether if two elements are in the same set or not. If two elements are at different sets - merge the smaller set to the larger set.
    ↳ at the end we get connected components in a graph.
  - Heaps are also used.
- Time Complexity for Kruskals
  → Algorithm is run in O(Elog V) time
  → The performance with Prims algorithm is similar

! Has to include every vertex in the graph

! Kruskals treats every node as an independent tree and connects one with another.

Kruskal's Pseudocode →

```
MST_Kruskal()
begin
    //input is a simple connected graph represented by array of edges edge[]
    // output is list of edges T in MST
    // create a partition for the set of vertices
    for each vertex v ∈ V
        C_v = {v}
    //create a minHeap h from array of edges E
    h = new Heap(E)
    //let T be an initially empty tree
    T = ∅
    while size(T) < n-1
        (u,v,wgt) = h.removemin()
        C_v = findSet(v)
        C_u = findSet(u)
        if C_u ≠ C_v
            union(C_u, C_v)
            T = T ∪ {(u,v,wgt)}
    return T
end
```

! Pseudocode checks if u and v vertices belong to different sets (partitions) — to check we find the representative root of the sets, if they are different edge is added to MST.
↳ this ensures no cycles are formed.

This is a very efficient way to make sure the MST is found without forming cycles. Also makes sure the same edge is not including again.

Dijkstra's Algorithm

- A popular algorithm for solving many single source shortest problems having non-negative edge weights in the graphs.
  - ↳ It maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest distance. The distance is updated if a smaller vertex is found. This process continues until the destination vertex is reached (all vertices have been visited.
- Steps for Dijkstra's →
  1. Mark the source node with a current distance of 0 and the rest with infinity.
  2. Set the non-visited node with the smallest current distance as the current node.
  3. For each neighbour, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0 → 1.
     - ↳ If it is smaller than the current distance of Node, it will be set as the current distance of N.
  4. Mark the current node 1 as visited
  5. Go to step 2 if any nodes are unvisited.
- Data structures for Dijkstra's →
  - → Node structure for all the graph edges
  - → Array of linked list to store graph in memory - Node** adj;
  - → Int array dist[] to record current distance of a graph vertex from the starting vertex s.
  - → Int array parent[] to store SPT
  - → Priority queue to find the next nearest vertex to s.
    - ↳ It will contain an internal heap array h[].
  - → Int array hPos[] which records position of any vertex with heap array h[].
    - ↳ vertex v is in position hPos[v] in h[]

↳ Pseudocode for Dijkstra's

```
Begin
  // G = (V, E)
  for each v ∈ V
      dist[v] = ∞
      parent[v] = null   // have a special null vertex
      hPos[v] = 0        //indicates that v ∉ heap
  pq = new Heap          // priority queue(heap) initially empty
  v = s                  // s will be the root of the SPT
  dist[s] = 0

  while v ≠ null         // loop should repeat |V|-1 times
      for each u ∈ adj(v)    // examine each neighbour u of v
          d = wgt(v,u)
          if dist[v] + d < dist[u]
              dist[u] = dist[v] + d
              if u ∉ pq                  // After changing dist[u]
                  pq.insert(u)           // either insert it or siftUp
              else                       // in the heap and record
                  pq.siftUp(hPos[u])     // its new parent vertex.
              parent[u] = v
          end if
      end for
      v = pq.remove()
  while end
  return parent
End
```

→ Time complexity

↳ $O((V+E)\log_2 V)$ for Dijkstra's Algorithm.

- Sorting Algorithms -

Quick Sort

↳ A sorting algorithm based on Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the picked pivot in its correct position in the sorted array.

↳ Pseudocode for quicksort →

```
quicksort (array) {
    if (array.length > 1) {
        choose a pivot ~~(usually halfway through array)~~
        while (items left in array) {
            if (item < pivot)
                put item into subarray 1
            else
                put item into subarray 2
        }
        quicksort (subarray 1)
        quicksort (subarray 2)
    }
```

↳ Time complexity = $O(N \log n)$

↳ efficient on large sets, requires less memory to function.

Bubble Sort

↳ Simplest sorting algorithm - works by repeatedly swapping the adjacent elements if they are in the wrong order.

↳ Not suitable for large data sets

↳ Time complexity = $O(N^2)$

↳ It is a very easy algorithm but it limits the efficiency and can take a lot of time to sort an array.

↳ Pseudocode for Bubble Sort →

```
bubbleSort (int a[], int n)
Begin
    for i=1 to n-1      // n-1 repeats
        sorted = true
        for j = 0 to n-1-j
            if a[j] > a[j+1]
                temp = a[j]
                a[j] = a[j+1]
                a[j+1] = temp
                sorted = false
        end for
        if sorted = true
            break from i loop
    end for
End
```

Comb Sort

↳ A similar sorting algorithm to bubble sort but it improves upon by comparing elements that are farther apart. - It works by repeatedly comparing and swapping elements with a gap between them, which starts large and decreases with every iteration.

↳ The gap size is chosen by a shrink factor which determines how quickly gap decreases to 1 → in which case the algorithm does a final pass using bubble sort.

↳ The shrink factor is most efficient when its 1.3.

↳ Time complexity = $O(n^2)$

→ Pseudocode on next page

```
combSort (input[])
    gap = input.size  // initialize the gap size
    shrink = 1.3      // set the gap shrink factor
    sorted = false
    loop while sorted = false
        //update gap value for next comb pass
        gap = floor(gap/shrink)
        if gap > 1
            sorted = false
        else
            gap = 1
            sorted true
        end if
        i = 0
        loop while i + gap < input.size
            if input[i] > input[i+gap]
                swap(input[i], input[i+gap])
                sorted = false
            end if
            i = i + 1
        end loop
    end loop
end
```

Merge Sort

↳ Algorithm divides the array and does the work of recombining the subparts afterwards — whereas quick sort does the work of partioning first and then divides into 2 smaller quick sorts.

↳ Time complexity ≈ $O(2N\log_2 N)$

↳ twice as slow as quick sort

↳ Pseudocode for Merge Sort

```
// a is an array of size n
mergeSort (a, n)
    int b = new int[n]
    mSort (a, b, 0, n-1)
// a and b are arrays, l is the left index and r is the right index
mSort (a, b, l, r)
    if r > l
        m = (r+l)/2
        mSort (a, b, l, m)
        mSort (a, b, m+1, r)
        // copy lhs subarray of a to b
        for i = m+1 to l+1
            b[i-1] = a[i-1]
        // copy rhs subarray of a to b in reverse order
        // b[r] = a[m+1], b[r-1] = a[m+2], etc....
        for j = m to r-1
            b[r+m-j] = a[j+1]

        // merge 2 sorted subarrays in b into one sorted []
        for k = l to k = r
            if b[i] < b[j]
                a[k] = b[i++]
            else
                a[k] = b[j--]
```

↳ merge sort is more stable - even if the data varies.