Insertion Sort (Normal Array)

```
int main() {
   int size;
   cout << "\n ----- insertion sort ----- \n";
   cout << "\nHow many elements to sort:  ";  cin >> size;
   int* dataArray = new int[size];   // claim the array
   // sort random integers
   for (int k = 0; k < size; k++)  dataArray[k] = rand();
   cout << "\nInserttion sort of Integer data initially in random
order\n\n";
   for (int k = 0; k < size; k++) {
     printf("%d, ", dataArray[k]); }
   printf("\n");
   // Insertion Sort
   for (int i = 1; i < size; i++) {
      int tmp = dataArray[i];
      int j = i - 1;
      while (j >= 0 && tmp > dataArray[j]) { //DESC
      // while (j >= 0 && dataArray[j] > tmp) ASC
         dataArray[j + 1] = dataArray[j];
         j = j--;
      }
      dataArray[j + 1] = tmp;
   }
   cout << "First five sorted elts:  ";
   for (int i = 0; i < 5; i++)                    8,23,0,0
      cout << dataArray[i] << ", ";               0,8,23,0
   cout << endl;                                  0,0,8,23
   cout << "Last five sorted elts:  ";
   for (int i = 5; i > 0; i--)
      cout << dataArray[size - i] << ", ";
   cout << endl;
   return 0;
} //Time: O(n^2) Space: O(1)
```

Quick Sort

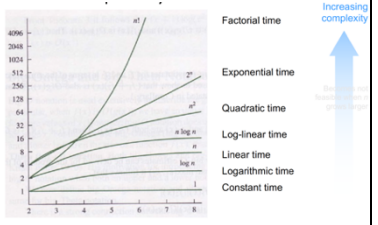
```
void quicksort(int data[], int p, int r) {
// p: start, r: end index                          qs(data,0,4)
   int pivot, low, high, q;                        qs(data,0,1)
   if (p >= r) { return; } //base case             qs(data,0,0)
   pivot = p; //set first element as pivot         qs(data,3,4)
   low = p + 1;                                    qs(data,3,3)
   high = r;
   while (low < high) {
      while(data[low] <= data[pivot] && low < r) { low++; }
      while(data[high] > data[pivot] && high > p) { high--; }
      if (low < high) { swap(data[low], data[high]); }
   }
   if (data[pivot] > data[high]) { //swap pivot with high
      swap(data[pivot], data[high]);
   }
   q = high;
   quicksort(data, p, q-1);
   quicksort(data, q+1, r);
}
//Time (AVG): O(nlog n) Time(Worst): O(n^2)

Best case: random {1, 3, 2, 4, 7, 9, 6, 8}
Worst case: ascending or descending {1, 2, 3, 4, 5}
```

Slowest growth

$2/N$
37
$\log(N)$
$\sqrt{N}$
N
$N \log N$, $N \log(N^2)$
$N \log^2(N)$
$N^{1.5}$
N^2
$N^2 \log(N)$
N^3
2^N
$N!$

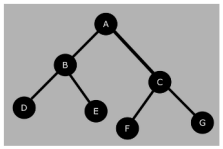
Note that $N \log(N^2)$ is $2N \log N$

Fastest growth

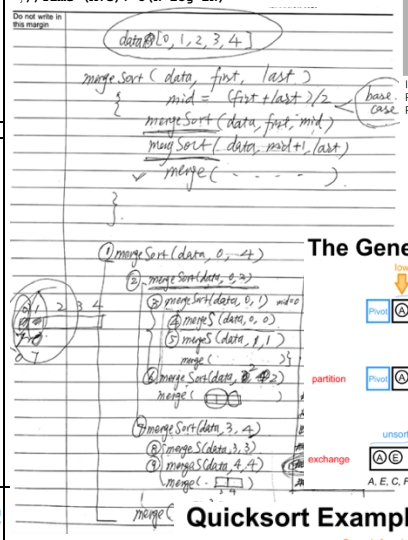


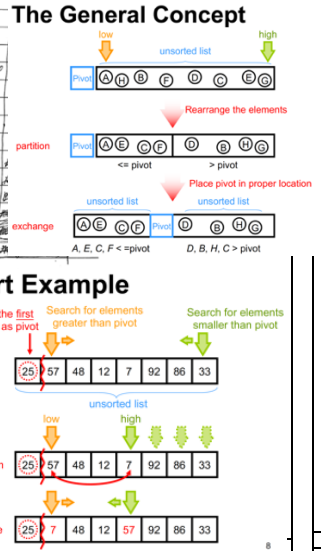
Merge Sort

```
void merge(int data[], int size, int firstloc, int midloc, int
lastloc) {
   int* tmp = new int[size];
   int i = firstloc, j = midloc + 1, k = 0;

   while (i <= midloc && j <= lastloc) {
      if (data[i] > data[j]) { tmp[k++] = data[i++]; } //DESC
      // if (data[i] <= data[j]) { tmp[k++] = data[i++]; } ACS
      else { tmp[k++] = data[j++]; }                          ms(data,0,5)
   }                                                          ms(data,0,2)
   while (i <= midloc) { tmp[k++] = data[i++]; }              ms(data,0,1)
   while (j <= lastloc) { tmp[k++] = data[j++]; }             ms(data,2,2)
   i = 0;                                                     ms(data,3,5)
   while (i < k) { data[firstloc + i] = tmp[i++]; }           ms(data,3,4)
}                                                             ms(data,4,5)
                                                              ms(data,5,5)
void mergesort(int data[], int size, int firstloc, int lastloc) {
   int midloc = (firstloc + lastloc) / 2;
   if (firstloc >= lastloc) { return; }
   mergesort(data, size, firstloc, midloc);
   mergesort(data, size, midloc + 1, lastloc);
   merge(data, size, firstloc, midloc, lastloc);
}//Time (AVG): O(N log 2N)
```



In-order: D,B,E,A,F,C,G
Pre-order: A,B,D,E,C,F,G
Post-order: D,E,B,F,G,C,A



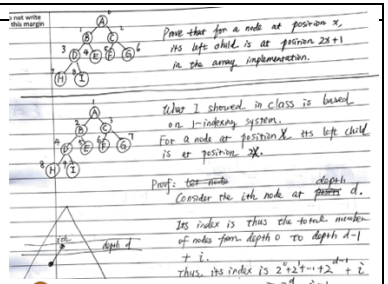


Terminologies of Tree

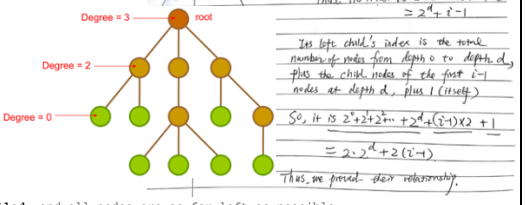
Properties of tree data structure:
- There is **one and only one path between every pair of vertices in a tree**
- A tree with **n vertices has exactly (n-1) edges**
- A graph is a tree if and only if it is **minimally connected**
- Any connected **graph with n vertices and (n-1) edges is a tree**

Root: The first node from where the tree originates is called as a root node
- In any tree, there must be only one root node
- We can never have multiple root nodes in a tree data structure

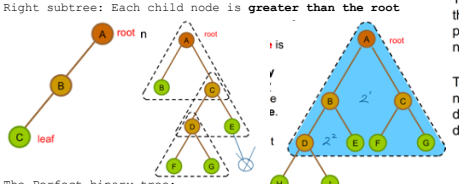
Leaf nodes: The node which **does not have any child** is called a leaf node
Ancestor(祖先): A node that is **connected to all lower-level nodes**
(The nodes along the simple path from the root to node x are the ancestors of x)
Descendants(子孫): The **connected lower-level nodes are "descendants" of the ancestor node**
(The descendants of a node x are the nodes in the subtrees of x)
Parent nodes: The node which **has a branch from it to any other node**
Child nodes: The node which **is a descendant of some node** is called a child node



Level: The number of **edges along the unique path between it and the root node**
Depth: The **length of the unique path from the root to the node**
Height: The number of **edges from the node to the deepest leaf (from 0)**
Degree: The **number of subtrees of a node**
Number of nodes: Count of all nodes
Path: The **sequence of Nodes and Edges from one node to another node**



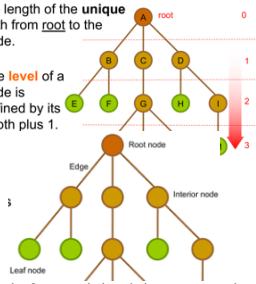
Skewed tree: All nodes are **either on the left-hand side or right-hand side**
Full binary tree: Every node in the tree has **either 0 or 2 children**
Complete binary tree: Every level, **except possibly the last, is completely filled**, and all nodes are as far left as possible
Perfect binary tree: All **interior nodes have two children, and all leaves have the same depth or same level**
Left subtree: Each child node is **smaller than the root**
Right subtree: Each child node is **greater than the root**



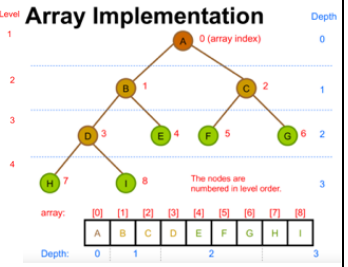
The Perfect binary tree:
The maximum height: **n-1**
The minimum height: **int(log2(n))**
The maximum number of nodes: **2^(h+1)-1**
The minimum number of nodes: **h+1**

The **depth** of a node is the length of the **unique** path from **root** to the node.

The **level** of a node is defined by its depth plus 1.



Array Implementation



Finding the parent node of node x in the array implementation:
This depends on how you save the root node. If you save the root node from position 1 in array A. The position of the parent node is int((the position of the x node/2). If you store the root at A[0], The position of the parent node is int((the position of the x node -1)/2).

How about x's left child and right child in the array?
This depends on how you save the root node. If you save the root node from position 1 in array A, the node at x's left child is at 2x, and x's right child is at 2x+1. If you store the root at A[0], x's left child is 2x+1, and x's right child is at 2x+2.

Binary search tree (BST)'s definition: Each node has at most 2 children

Searching on a BST:
Start the search from the root node, if the searched node is smaller than the current root node, go to the left subtree. If the searched node is larger than the current root node, go to the right subtree.
Running time: **Best: O(log n) Average: O(n) Worst: O(n)**

Inserting a node in a BST:
Create a new node and assign a value to it, if root node=NULL, insert the new node. If the new node is larger than the root node, move to the right subtree. If the new node is smaller than the root node, move to the left subtree.

Deleting a node in a BST:
1. Leaf node: Delete it and then reset its parent's reference
2. If the node has one child: Before deletion, adjust the pointer of the parent to point to the grandson, then delete it
3. If the node has two Childs: Replace the deleted node with its inorder predecessor (biggest node in left subtree) or inorder successor (smallest node in right subtree)

Linked List

```
class Node {
public:
    int data; Node* next;
};
Node* mergeSortedList(Node* list1, Node* list2) {
    Node* newList = NULL;
    Node* end = NULL;
    if (list1->data <= list2->data) {
        newList = list1;  list1 = list1->next;}
    else {
        newList = list2;  list2 = list2->next;}
    end = newList;
    while (list1 != NULL && list2 != NULL) {
        if (list1->data <= list2->data) {
            end->next = list1;  list1 = list1->next;}
        else {
            end->next = list2;  list2 = list2->next;}
        end = end->next;
    }
    if (list1 != NULL) { end->next = list1; }
    if (list2 != NULL) { end->next = list2; }
    return newList;
}
int getlen(Node* list) {
    int len = 0;
    Node* curr = list;
    while (curr != NULL) {
        len++;
        curr = curr->next;}
    return len;
}
```

```
int countNode (treeNode* tree) {
  if tree == NULL { return 0; }
  if (tree->left == NULL && tree->right != NULL)
      return 1 + countNode(tree->right);
  if (tree->left != NULL && tree->right == NULL)
      return 1 + countNode(tree->left);
  return countNode(tree->left) + countNode(tree->right);
}
```

```
Node* getmid(Node* list) {
    Node* mid = list;
    Node* end = list->next;
    while (mid->next != NULL && (end != NULL && end->next != NULL)) {
        mid = mid->next;
        end = end->next->next;}
    return mid;
}
Node* mergeSort(Node* list) {
    if (list == NULL || list->next == NULL) { return list; }
    Node* mid = new Node();
    mid = getmid(list);
    Node* list2 = mid->next;
    mid->next = NULL;
    Node* newList = mergeSortedList(mergeSort(list), mergeSort(list2));
    return newList;
}
void insert(Node** list_ref, int newdata) {
    Node* newNode = new Node();
    newNode->data = newdata;
    newNode->next = (*list_ref);
    (*list_ref) = newNode;
}
void printList(Node* node) {
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}
```

```
int oneChildNode = 0;
for(int i=0; i<n; i++) {
  int l=2*i+1;
  int r=2*i+2;
  bool leftempty = true;
  bool rightempty = true;
  if(l<n && A[l]!=-1) leftempty=false;
  if(R<n && A[R]!=-1) rightempty=false;
  if(leftempty &&!rightempty || !leftempty &&rightemtpty)
    oneChildNode++;
}
return oneChildNode;
```

Heap

```
Remove Time(heapify(H,1))
Def Heap_Extraction(H,key){
  if heap_size(H)<=0:
    throw Error "heap_underflow"
  max = H[1]
  H[1] = H[Heapsize(H)]//move last element to root
  Heap_size(H)-=1      //Heap_size minus 1
  Heapify(H,1)
  return max
}
```

```
void swap(int *a, int *b){
  int tmp = *a;
  *a = *b;  *b = tmp;
}
void buildHeap(int arr[], int N){
  int startIdx = (N / 2) - 1;
  for (int i = startIdx; i >= 0; i--) { heapify(arr, N, i); }
}
void heapify(int arr[], int N, int i){
  int largest = i;    int l = 2 * i + 1;  int r = 2 * i + 2;
  if (l < N && arr[l] > arr[largest]) { largest = l; }
  if (r < N && arr[r] > arr[largest]) { largest = r; }
  if (largest != i) {
  swap(&arr[i], &arr[largest]);  heapify(arr, N, largest);  }
}
void heapSort(int arr[], int N) {
  for (int i = N / 2 - 1; i >= 0; i--) {
    heapify(arr, N, i); }
  for (int i = N - 1; i > 0; i--) {
    swap(arr[0], arr[i]);  heapify(arr, i, 0); }
}
```

```
removeMax (92,80,50,70,30,8,40,6,15,12)
H=(80,70,50,15,30,8,40,6,12)
```

```
Def Heap_sort(H) { //O(NlogN)
  Build Heap(H)    #O(N)
  for i in length(H) downto 2
    swap(H[1],H[i])
    Heap_size(H)-=1
    Heapify(H,1)
```

```
Build heap: 3 4 1 9 2 8 0
Heapify(H,3): 3 4 8 9 2 1 0
Heapify(H,2): 3 9 8 4 2 1 0
Heapify(H,1): 9 4 8 3 2 1 0
```

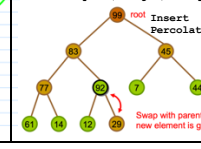
```
Insert 95 into heap
H=(100,90,40,50,55,20,19,80)
H=(100,90,40,50,55,20,19,50)
H=(100,90,40,90,55,20,19,50)
H=(100,95,40,90,55,20,19,50)
```

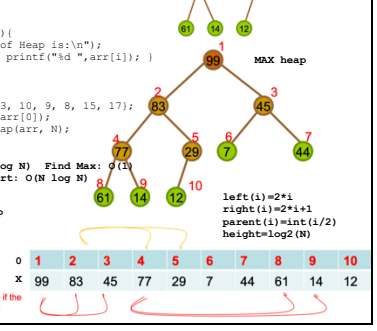
```
Insert(min heap)
H[]=(2,3,7,4,9,8,15,13,10, )
H[]=(2,3,7,4,9,8,15,13,10,9)
H[]=(2,3,7,4,3,8,15,13,10,9)
H[]=(1,2,7,4,3,8,15,13,10,9)
```

```
Insert (O(log(n))
Def Heap_insert(H,key){
  Heap_size = Heap_size(H)+1 //increase array size
  i = Heap_size(H)  //i is the index of new element
  while (i > 1 and H[parent(i)] < key //Max heap{
                 // H[parent(i)] > key //min heap
    H[i] = H[parent(i)]
    i = parent(i) )    //parent(i)=int(i/2)
  H[i] = key }
(the implementation of this step depends on what data type
you use, e.g. array, linked list, or vector)
```

```
void deleteRoot(int arr[], int& n){
 int lastElement = arr[n - 1];
 arr[0] = lastElement;
 n = n - 1;
 heapify(arr, n, 0);
}
void insertNode(int arr[], int& n, int Key){
 n = n + 1;
 arr[n - 1] = Key;
 heapify(arr, n, n - 1);
}
void printHeap(int arr[], int N){
 printf("Array representation of Heap is:\n");
 for (int i = 0; i < N; ++i) { printf("%d ",arr[i]); }
 printf("\n");
}
int main(){
 int arr[] = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17};
 int N = sizeof(arr) / sizeof(arr[0]);
 buildHeap(arr, N);    printHeap(arr, N);
 return 0;
}
```

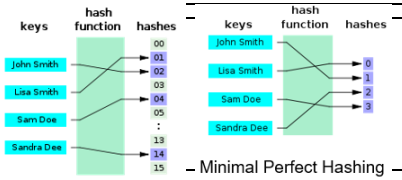
//Insert: O(log N) Delete: O(log N) Find Max: O(1)
Build Heap: O(N log N) Heap Sort: O(N log N)





Hash

Hash: any function that can be **used to map data of arbitrary size to fixed-size values**
Collision: situation that occurs **when two distinct pieces of data have the same hash value**
Search(T, k) = O(1) Insertion(T, k) = O(1)
Deletion(T, k) = O(1)
Good Hashing: Low Cost, Easy and fast to compute,
Variable Range, Able to transform words, symbols into numbers,
Uniformity,
Distributes the keys evenly
Minimize the chance of collisions



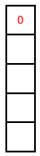
Perfect Hashing

Minimal Perfect Hashing

Opening addressing

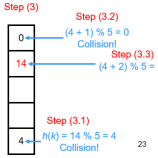
Linear probing(顺序找空位)

h(k) = k % n
h(k) != null, try
(h(k) + 1) % n, if != null, try
(h(k) + 2) % n……
Step (1)



Quadratic probing

h(k, i) = (h(k) + i^2) % n
h(k) = k % n
(h(k) + 1^2) % n //Jump 1 slot
(h(k) + 2^2) % n //Jump 3 slot
(h(k) + 3^2) % n //Jump 5 slot



Double hashing

h1(k) = k % n, h2(k) = 1 + k % (n+1)
h(k, i) = (h1(k) + i * h2(k)) % n
h(k, 1) = (h1(k) + h2(k)) % n
h(k, 2) = (h1(k) + 2 * h2(k)) % n
h(k, 3) = (h1(k) + 3 * h2(k)) % n

Advantage: **Jump interval is decided using a second independent hash function.**

Disadvantage: **Costs more time to compute a new hash value.**

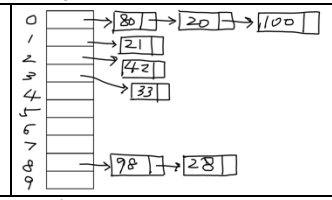
Advantage: **Since open addressing do not use pointers, it can be easier to serialize | Less memory requires for small record size**

Disadvantage: **needs more computation to avoid clustering | Once the table becomes full, hash functions fail to terminate**

Closing addressing / Chaining (Collision resolution)

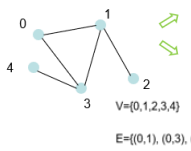
Chaining buckets
Insert to list: O(1)
Search in list: O(|T(hx)|)
Delete in list: O(|T(hx)|)
n be the number of keys, m be the size of T
α = n/m = average number of keys per slot
Unsuccessful search: **θ(1+α)**
Expected search time = **θ(1) if α = O(1)**
Advantage: **easy to implement effectively**
Easily delete a value from the table
It uses less memory if the record is large compared to the open addressing

Disadvantage: **Difficult to serialize data from the table**



Graph

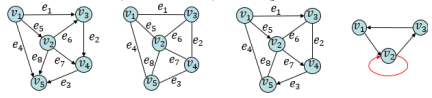
A graph is a data structure that consist a sets of vertices (called nodes) and edges.
A Graph is a tuple: $G=(V,E)$
A set of vertices (nodes): $V=\{v1,v2,v3,\dots\}$
A set of edges: $E=\{e\_1,e\_2,e\_3\dots\}$



Path: {0, 1, 2, 1, 3, 4}

Simple path: {2, 1, 0, 3, 4}

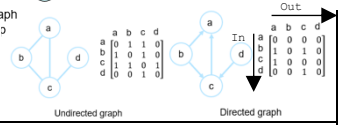
Complete Graph



Hand-shaking lemma

Directed graph: $\sum_{v\in V} in\text{-}degree(v) = \sum_{v\in V} out\text{-}degree(v) = |E|$

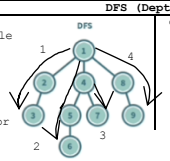
Undirected graph: $\sum_{v\in V} degree(v) = 2|E|$



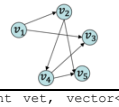
V	In-degree	Out-degree	Degree
v_1	0	3	3
v_2	1	3	4
v_3	2	1	3
v_4	2	1	3
v_5	3	0	3
all	8	8	16

DFS (Depth-first Search)

1. Start from a vertex
2. Travel to new vertex going as deep as possible
3. If stuck, go back and repeat the process




```
dfs(v)
 for each x ∈ V
  v.visited = false
  S ← (v)  //S is a stack implemented using vector
  D ← ()  //list for output
  while S not empty
    v ∈ S.pop_back() // v is the S' last element
    if v is not visited
      D.push_back(v)
      v.visited = true.  //indicate that we have visited v
    for each x  s.t.(v,x) ∈ E  and  ! x.visited
      S.push_back(x)  //newly pushed item will be visited next
    return D
```

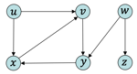


$dfs(v_1)$: v_1, v_2, v_4, v_3, v_5

$dfs(v_2)$: ?

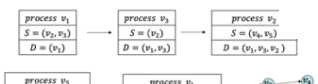
$dfs(v_3)$: v_3

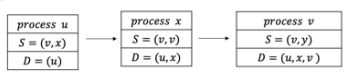
```
dfs(v)
  output (v)
  v.visited←true
  for each x such that (v,x) ∈ E /
    if ! x.visited
      dfs (x)
```

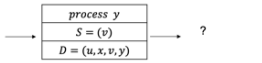


$dfs(u)$
$S=(u)$
$D=()$

$dfs(v_1)$
$S=(v_1)$
$D=()$







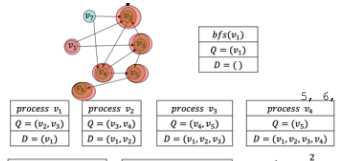

```
void RecDFS(int vet, vector<vector<int>> mat, vector<bool> &visited){ //starting from node 0
    cout << "Node " << vet << " visited" << endl;
    visited[vet] = 1;
    for (int i=0;i<mat.size();i++){
        for(int j=0;j<mat.size();j++){
            if (mat[i][j] == 1 and !visited[j])
                return RecDFS(j, mat, visited);
        }
    }
}
void printMat(vector<vector<int>> mat){
    cout << "inputed Mat:" << endl;
    for(int i = 0;i<mat.size();i++){
        for(int j=0;j<mat[i].size();j++){
            cout << mat[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl << endl;
}
```

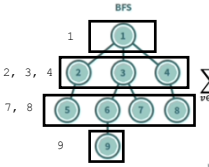
```
void nonRecDFS(int vet, vector<vector<int>> mat){
    vector<bool> visited;
    for (int i=0;i<mat.size();i++) visited.push_back(0); //initialization
    vector<int> S;
    S.push_back(vet);
    while(!S.empty()){
        int element = S.back();
        S.pop_back();
        visited[element] = 1;
        cout << "Node " << element << " visited" << endl;
        for(int j=0;j<mat.size();j++){
            if (mat[element][j] == 1 and !visited[j])
            S.push_back(j);
        }
    }
}
int main(){
    vector<vector<int>> Mat = readMat();    printMat(Mat);
    int nodeNum = Mat.size();    vector<bool> visited;
    for (int i=0;i<nodeNum;i++) visited.push_back(0); //initialization
    cout << "recursive DFS" << endl;   RecDFS(0, Mat, visited);
    cout << "non-recursive DFS" << endl; nonRecDFS(0, Mat);
    return 0;
}
```

BFS (Breadth-first search)

1. Start from a vertex
2. Travel to new vertex in the first layer
   (which is connected with the start vertex)
3. If there is no new vertex in the first layer,
   go to nodes in the second layer
   (which is connected with the nodes in the first laver)

BFS (Running time: **O(|V|+|E|)**)





Hand-shaking Lemma

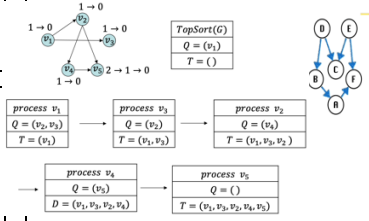
$\sum_{v\in V} outdegree(v) = |E|$

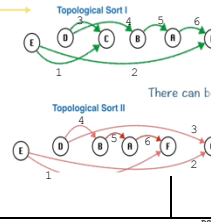
$O(|V|+|E|)$

```
bfs(v0)
 for each x ∈ V                          |V|
    x.state ← white  //initially all nodes are not touched
 v0.d ← 0 //distance from the source node v0
 v0.p ← nil //parent node in BFS tree     O(1)
 Q ← (v0)  //queue, implemented using a vector
 D ← ()   //list for our output
 while Q not empty
    v ∈ Q.pop_front //v is Q[0], then remove Q[0] from Q
    D.push_back(v) //for the output
    v.state ← red  // visited, safe to remove
    for each x  s.t.(v,x) ∈ E  and  x.state == white
       Q.push_back(x)
       x.state ← green
       x.d ← v.d + 1
       x.p ← v
 return D
```

Topological Sort

Topological sort or topological ordering: a directed graph is **a linear ordering of its vertices s. t. for every directed edge (u, v)** from vertex u to vertex v, u comes before v in the ordering






```
TopSort(G: Graph)
 Q ← () //queue for indegree-zero vertices
 T ← () //list for our output
 for each v ∈ V in G
    v.indegree = Compute In Degree(v) //Adjacency table   |V|
    if v.indegree == 0  then
       Q.push_back(v)
 while Q not empty  do
    v ← Q.pop_back()
    T.push_back(v)       //for the output                 |E|
    for each x  s.t.(v,x) ∈ E  do
       x.indegree − −
       if x.indegree == 0  then
          Q.push_back(x)
 if T.size() < |V|
    Error("Graph has a cycle")
 return T
O(|V|+|E|)
```