Introduction to Computer Science:

data structure

June 2020

Honguk Woo

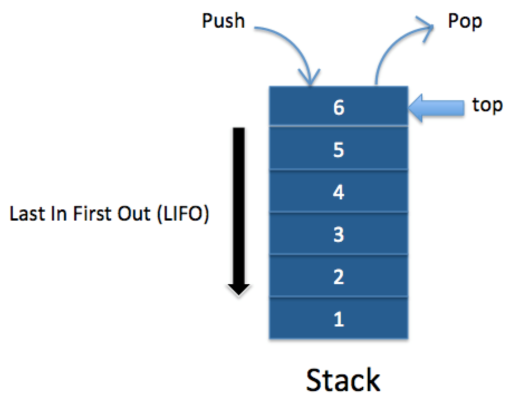
Abstract Data Types

- Views on data
  - Abstract level : sees data objects as groups of objects with similar properties and behaviors
  - Implementation level : sees the properties represented as specific data fields and behaviors represented as methods implemented in code
    - E.g., Array-based implementation : objects in the container are kept in an array
    - Linked-based implementation : objects in the container are not kept physically
- Abstract data type :
  - A data type (or class) for objects whose behavior is defined by a set of value and a set of operations
  - A data type whose properties, data fields and operations, are specified independently of any particular implementation (meaning that specifying “what” but not “how”)
  - ADT : stack, queue, list, graph, tree, ….

Stacks

- **Stack** : an abstract data type in which accesses are made at only one end
  - **LIFO**, which stands for **Last In First Out**
  - The insert is called **Push(),** and the delete is called **Pop()**

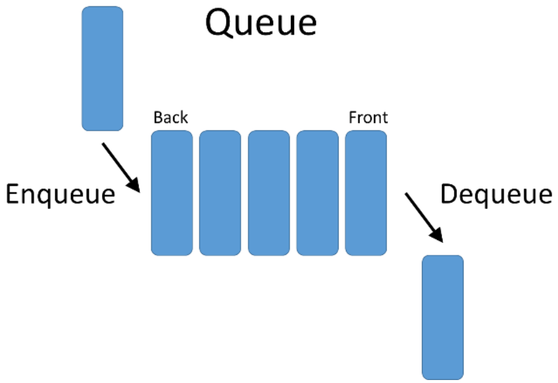
```
WHILE (more data)
    Read value
    Push(myStack, value)


WHILE (NOT IsEmpty(myStack))
    Pop(myStack, value)
    Write value
```



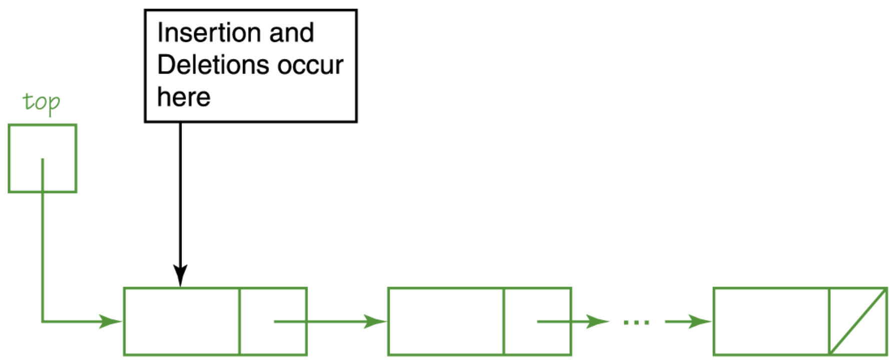
Queues

- **Queue** : an abstract data type in which items are entered at one end and removed from the other end
  - FIFO, for First In First Out
  - No standard queue terminology, but
    - ***Enqueue(), Enque, Enq, Enter,*** and ***Insert*** are used for the insertion operation
    - ***Dequeue(), Deque, Deq, Delete,*** and ***Remove*** are used for the deletion operation.

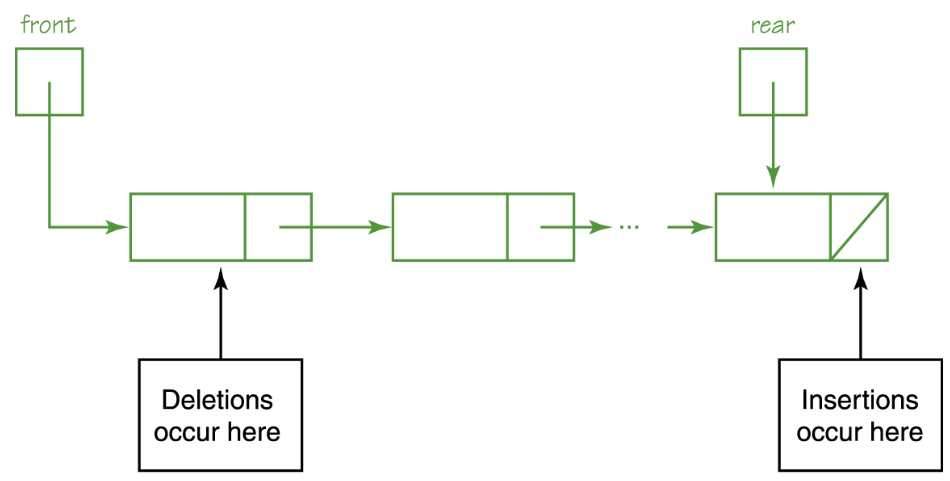
```
WHILE (more data)
    Read value
    Enque(myQueue, value)

WHILE (NOT IsEmpty(myQueue))
    Deque(myQueue, value)
    Write value
```



Stack and Queue in linked list



(a) A linked stack



(b) A linked queue

Lists

- A container of items that can be dynamically added and removed
- The logical operations that can be applied to lists
  - Add item — Put an item into the list
  - Remove item — Remove an item from the list
  - Get next item — Get (look) at the next item
  - more items — Are there more items ?

Array-based Implementations

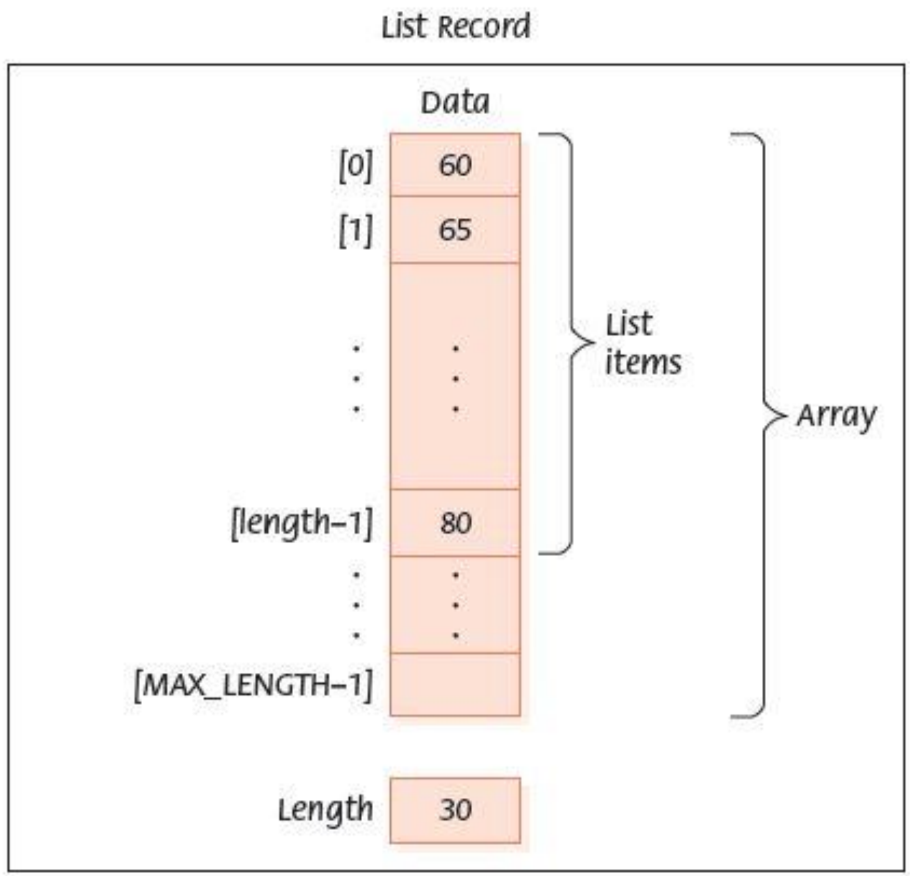


FIGURE 8.3 A sorted list of integers

Linked Implementations

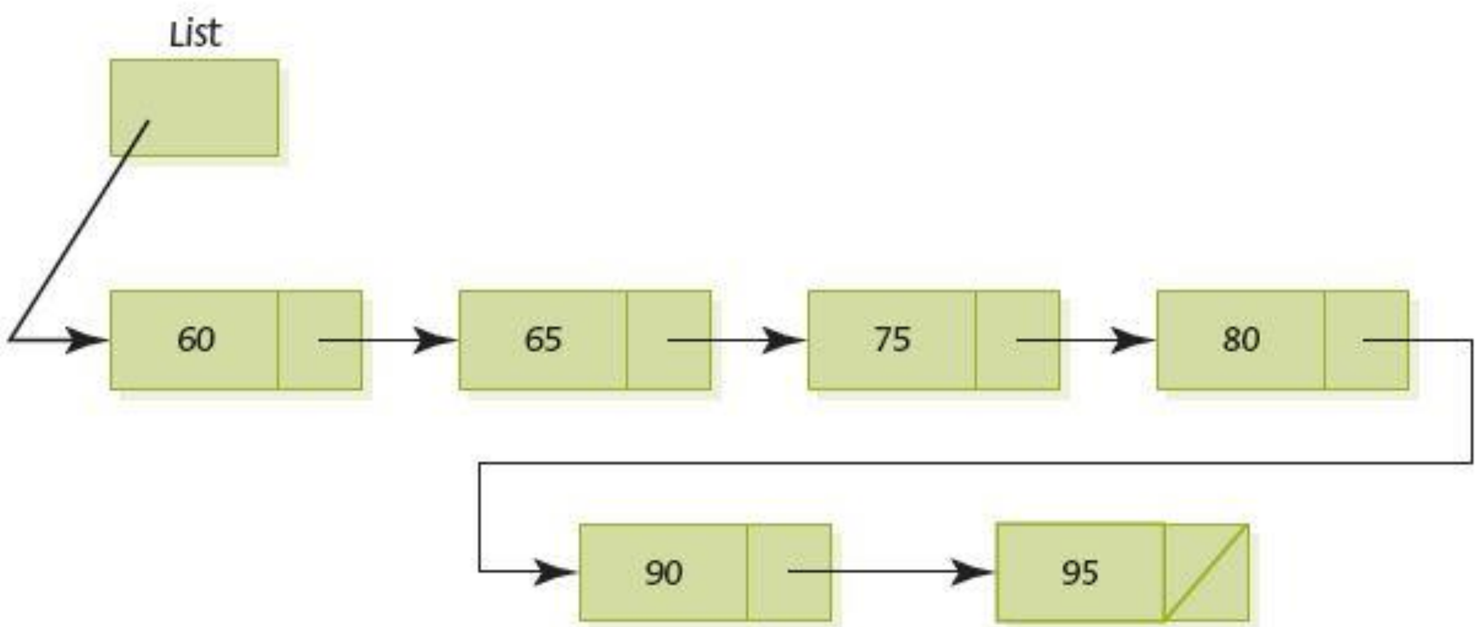


FIGURE 8.4 A sorted linked list

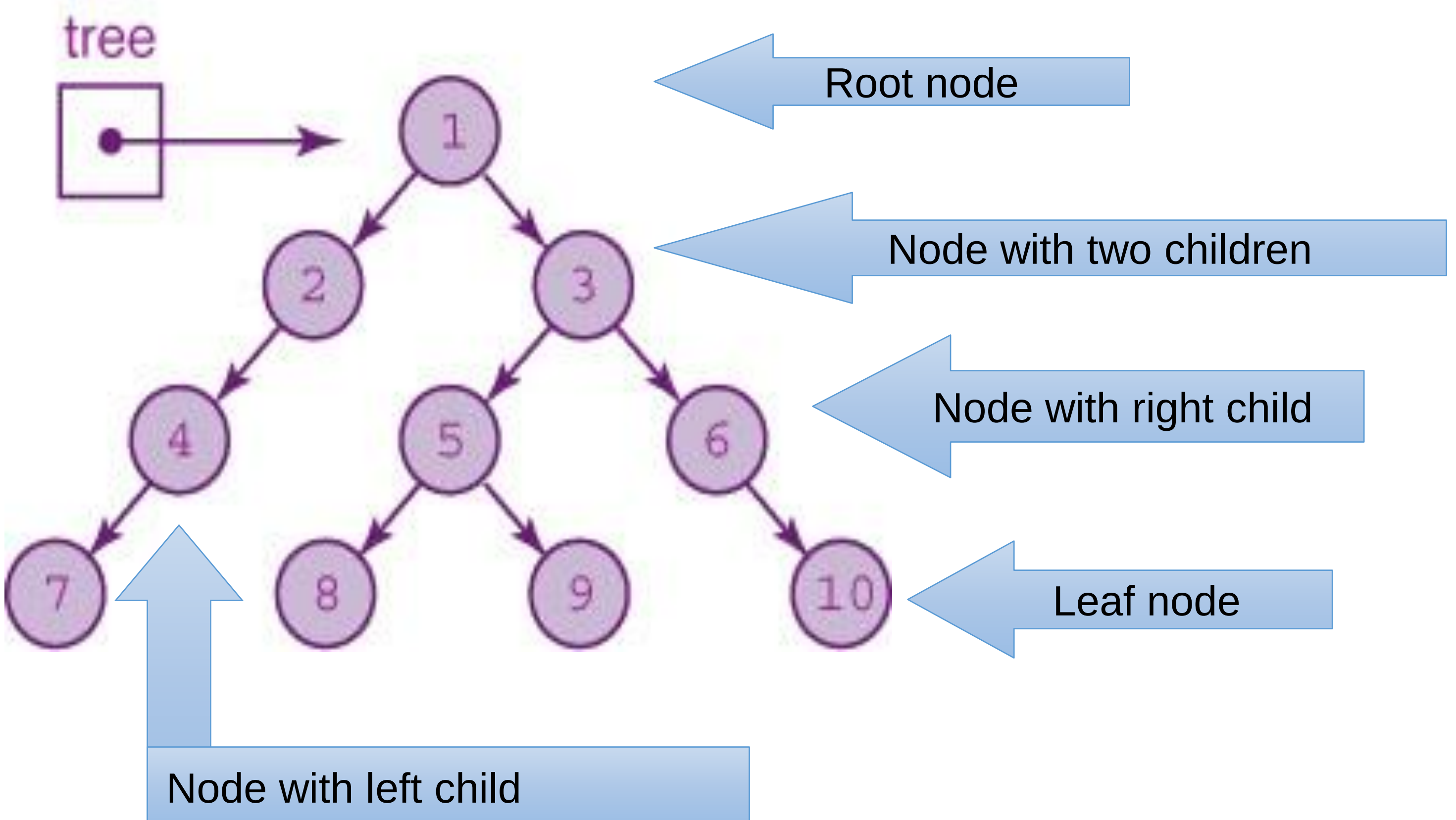
Algorithm for Creating and Print Items in a List

```
WHILE (more data)
       Read value
       Insert(myList, value)
Reset(myList)
Write "Items in the list are "
WHILE (moreItems(myList))
       GetNext(myList, nextItem)
       Write nextItem, ' '
```

Trees

- Structure such as lists, stacks, and queues are linear in nature; only one relationship is being modeled
- More complex relationships require more complex structures : tree, graph
- **Binary tree :** a linked container with a unique starting node called the root, in which each node is capable of having two child nodes, and in which a unique path (series of nodes) exists from the root to every other node

Trees



What is the unique path
to the node containing
5? 9? 7? …

Binary Search Tree : definition

- Binary search tree (BST)
  - A binary tree (shape property) that has the (semantic) property that characterizes the values in a node of a tree:
  - The value in any node is greater than the value in any node in its left subtree and less than the value in any node in its right subtree

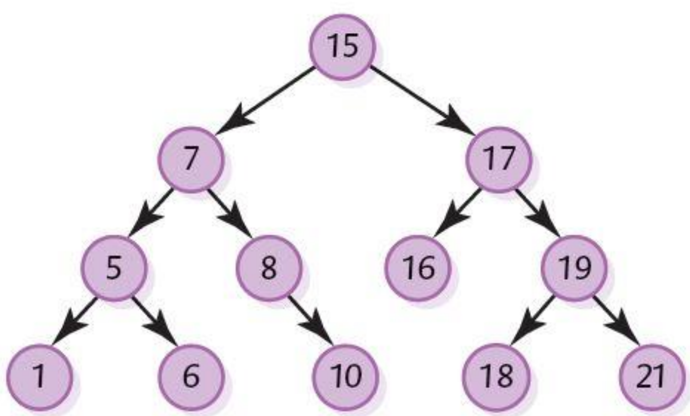
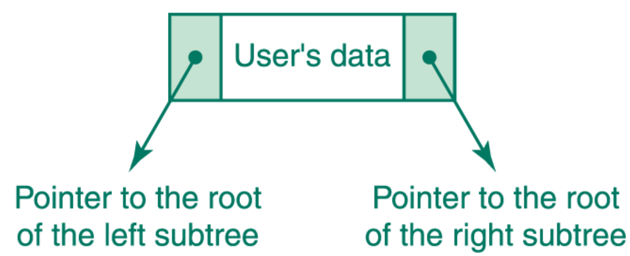


FIGURE 8.7 A binary search tree



Binary Search Tree : search

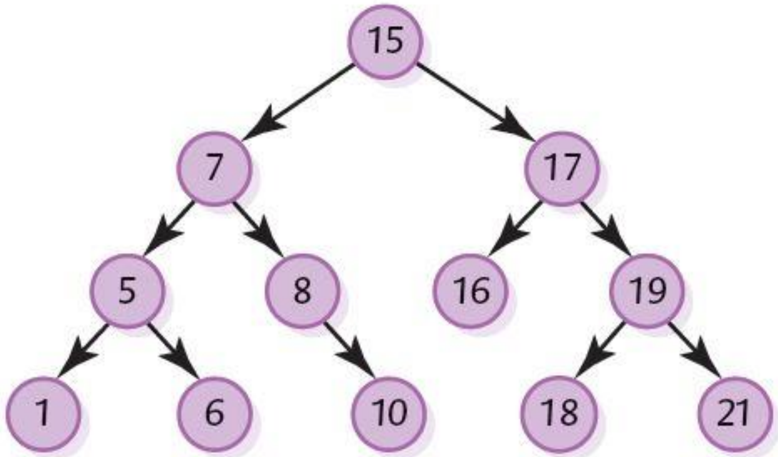


FIGURE 8.7 A binary search tree

```
IsThere(tree, item)

IF (tree is null)
   RETURN FALSE
ELSE
   IF (item equals info(tree))
       RETURN TRUE
   ELSE
       IF (item < info(tree))
           IsThere(left(tree), item)
       ELSE
           IsThere(right(tree), item)
```

Trace the nodes passed as you search for 18, 8, 5, 4, 9, and 15

Binary Search Tree : build

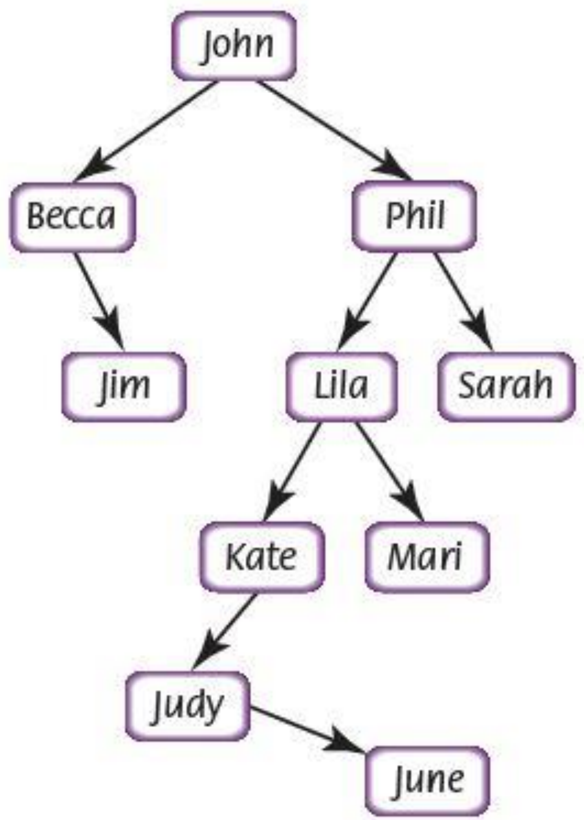


FIGURE 8.9 A binary search tree built from strings

```
Insert(tree, item)

IF (tree is null)
    Put item in tree
ELSE
    IF (item < info(tree))
       Insert (left(tree), item)
    ELSE
       Insert (right(tree), item)
```

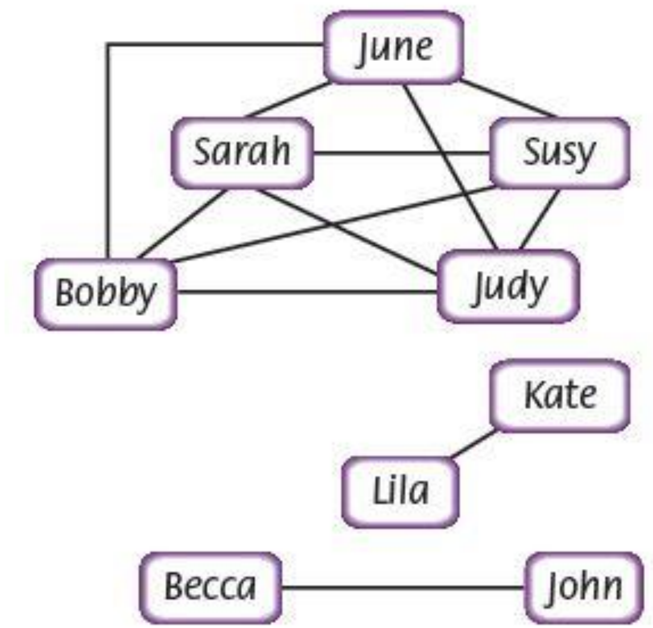
Binary Search Tree : print

```
Print(tree)

If (tree is not null)
   Print (left(tree))
   Write info(tree)
   Print (right(tree))
```

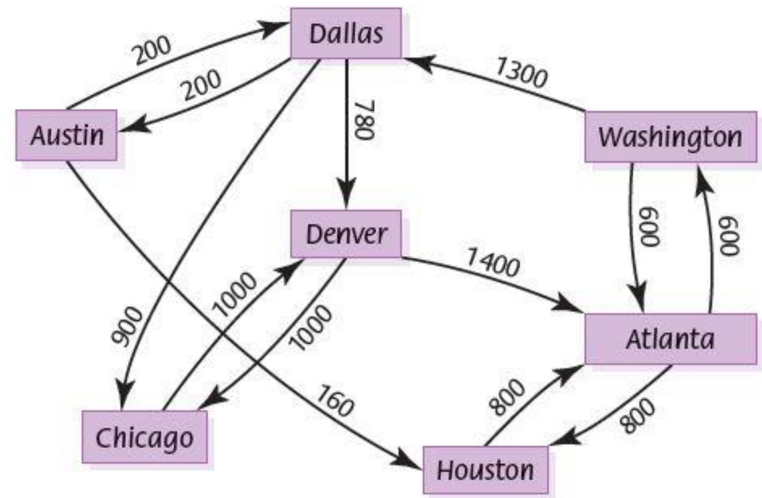
Graph

- **Graph** : A data structure that consists of a set of nodes (called vertices) and a set of edges that relate the nodes to each other
  - A graph **G = (V, E)**
    - V is a set of vertices
    - E is a set of edges
      - $E = (x, y)$ where $x, y \in V$
      - ordered or unordered pairs of vertices from V
  - **Undirected graph**
    - A graph in which the edges have no direction
  - **Directed graph**
    - A graph in which each edge is directed from one vertex to another vertex
- Commonly used for many applications
  - Social graph, map, task, …

Graph : example

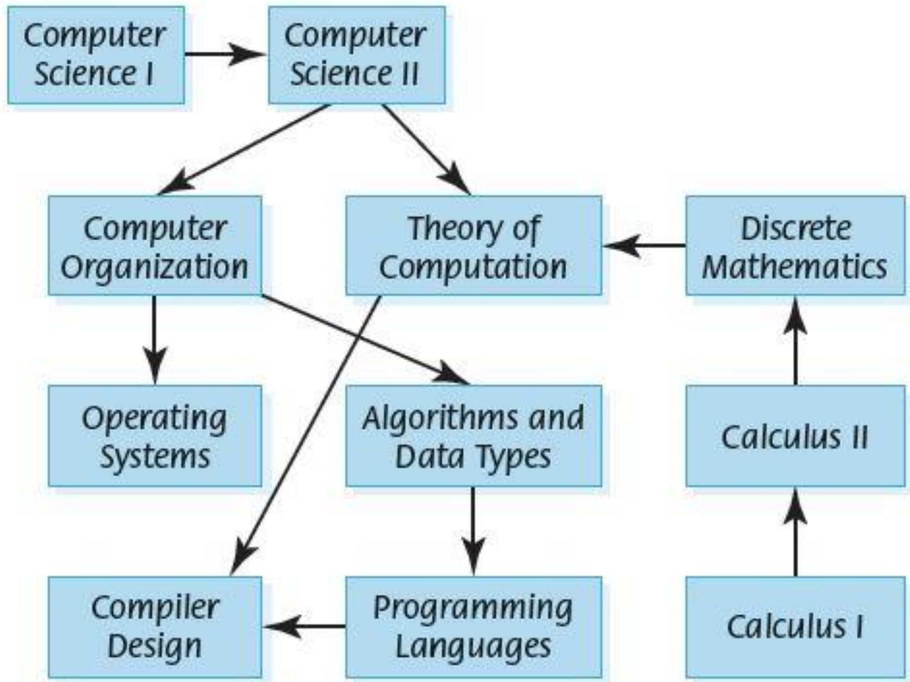


(a) Vertices: People
Edges: Siblings



(b) Vertices: Cities
Edges: Direct flights

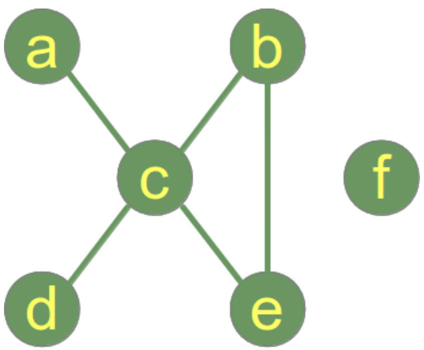
Graph : example



(c) Vertices: Courses
Edges: Prerequisites

FIGURE 8.10 Examples of graphs

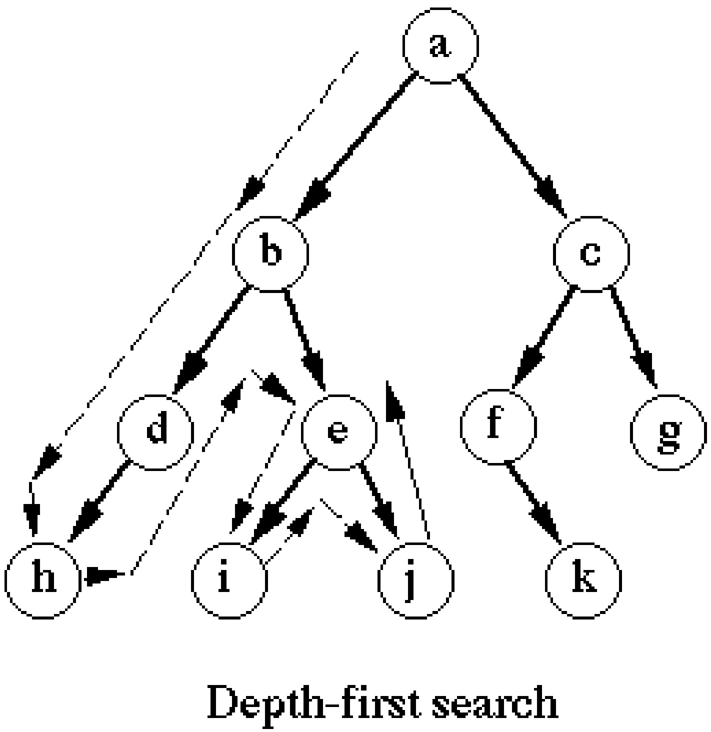
Graph : example

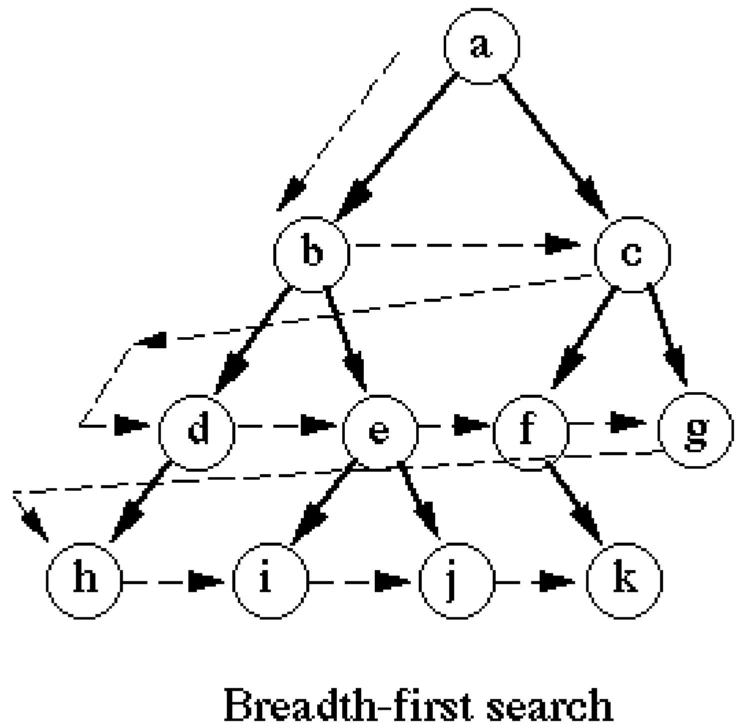


- Adjacency matrix (e.g., in Python dict)

```
graph = { "a" : ["c"],
          "b" : ["c", "e"],
          "c" : ["a", "b", "d", "e"],
          "d" : ["c"],
          "e" : ["c", "b"],
          "f" : [] }
```

Graph traversal

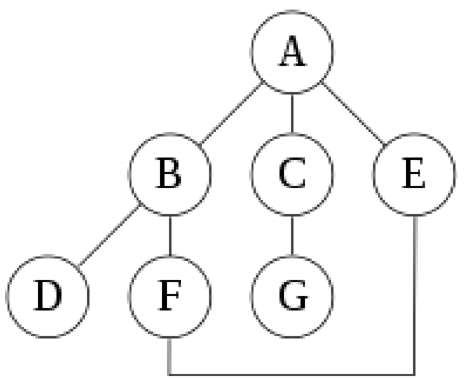




https://kevhuang.com/tree-traversal/

Graph traversal algorithms

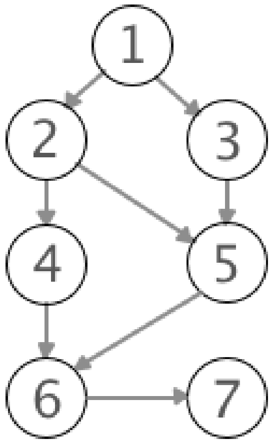
- **A Depth-First Search (DFS) Algorithm**
  - This is called a depth-first search because we start at a given vertex and go to the deepest branch and explore as far down one path before taking alternative choices at earlier branches



A -> B -> D -> F ->E -> C -> G

a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them

DFS with recursion : Python implementation



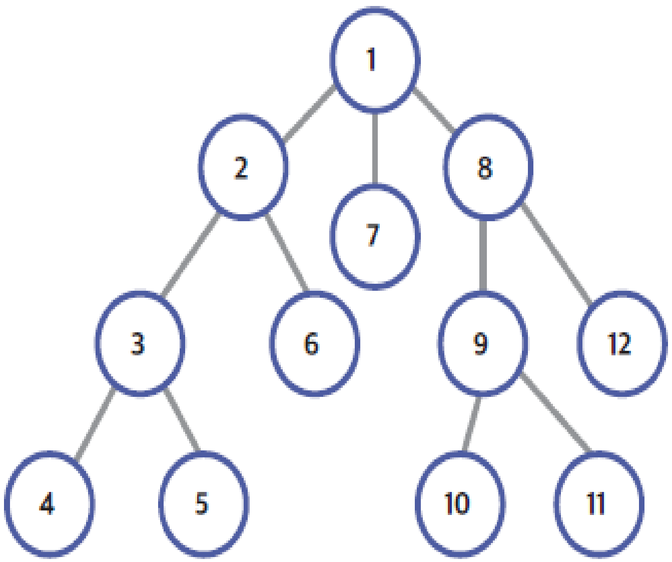

```
def dfs_recursive(graph, vertex, path):
    path += [vertex]
    for neighbor in graph[vertex]:
        if neighbor not in path:
            path = dfs_recursive(graph, neighbor, path)
    return path


adjacency_matrix = {1: [2, 3], 2: [4, 5], 3: [5], 4:
[6], 5: [6], 6: [7], 7: []}
print(dfs_recursive(adjacency_matrix, 1, []))
```

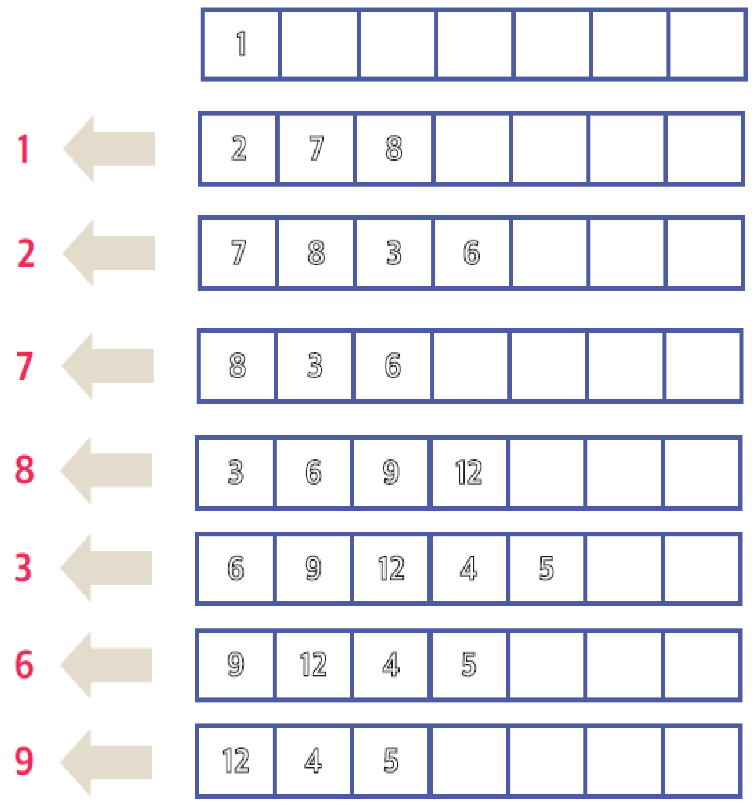
Breadth-First Search

- Breadth-First Search examines all of the vertices adjacent with startVertex before looking at those adjacent with those adjacent to these vertices
  - A Breadth-First Search uses a queue

BFS

- Searching for 9



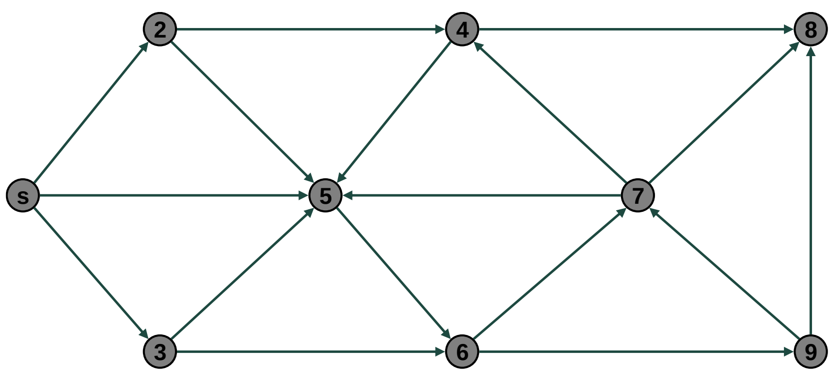


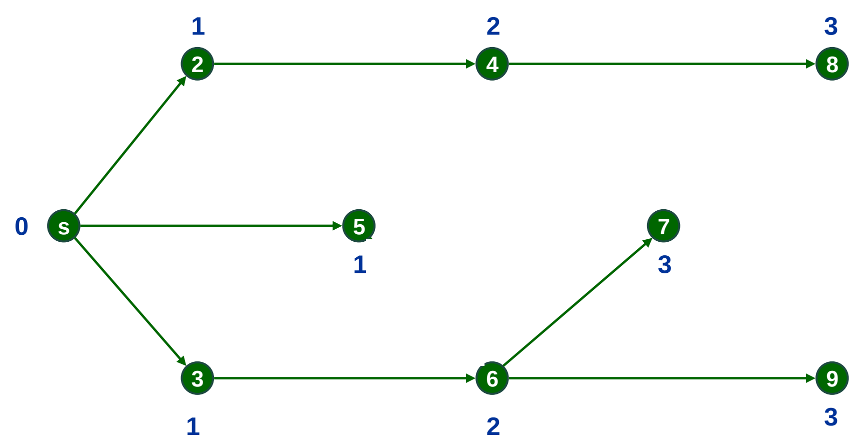
BFS with queue

```
Breadth First Search(startVertex, endVertex)

Set found to FALSE
Enque(myQueue, startVertex)
WHILE (NOT IsEmpty(myQueue) AND NOT found)
        Deque(myQueue, tempVertex)
        IF (tempVertex equals endVertex)
                Write endVertex
                Set found to TRUE
        ELSE IF (tempVertex not visited)
                Write tempVertex
                Enque all unvisited vertexes adjacent with tempVertex
                Mark tempVertex as visited
IF (found)
        Write "Path has been printed"
ELSE
        Write "Path does not exist"
```

Spanning Tree

- Note that tree has no cycle
- Given a graph G = (V, E),
  Spanning tree T = (V, E′) such that
  - E′ ⊂ E
  - **Connecting all vertices of V**

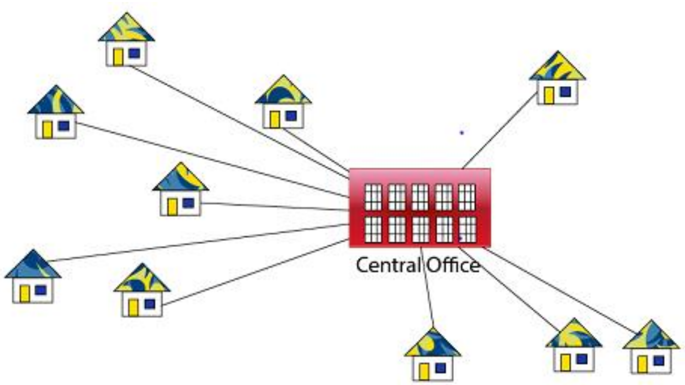


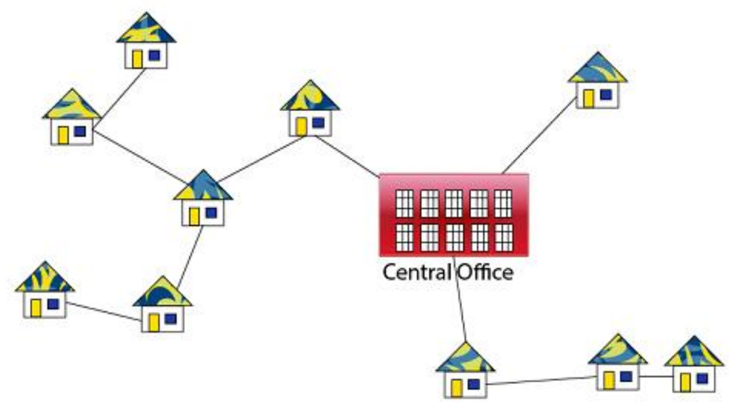


- A spanning tree can be constructed using DFS or BFS

Minimize the total cost of connections

- Suppose we want to connect a set of houses for telephone lines (e.g., cable length = cost)

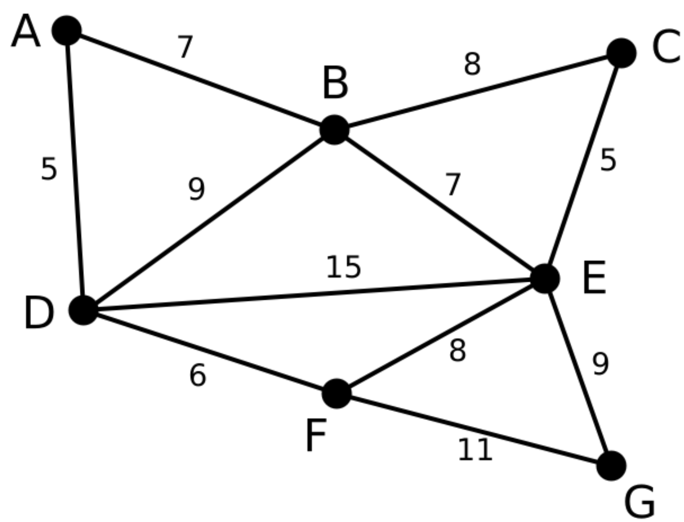




For n vertices,
How many edges ?
How many paths between two vertices ?
Two trees connected by a single edge

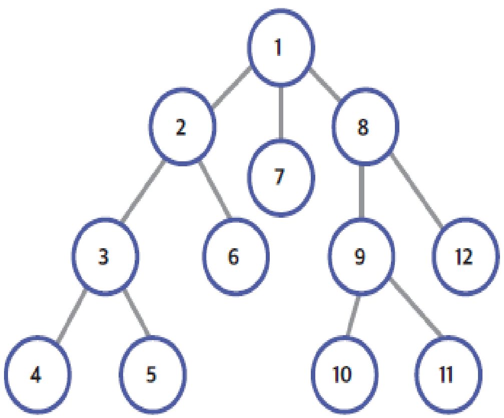
Minimal Spanning Tree (MST)

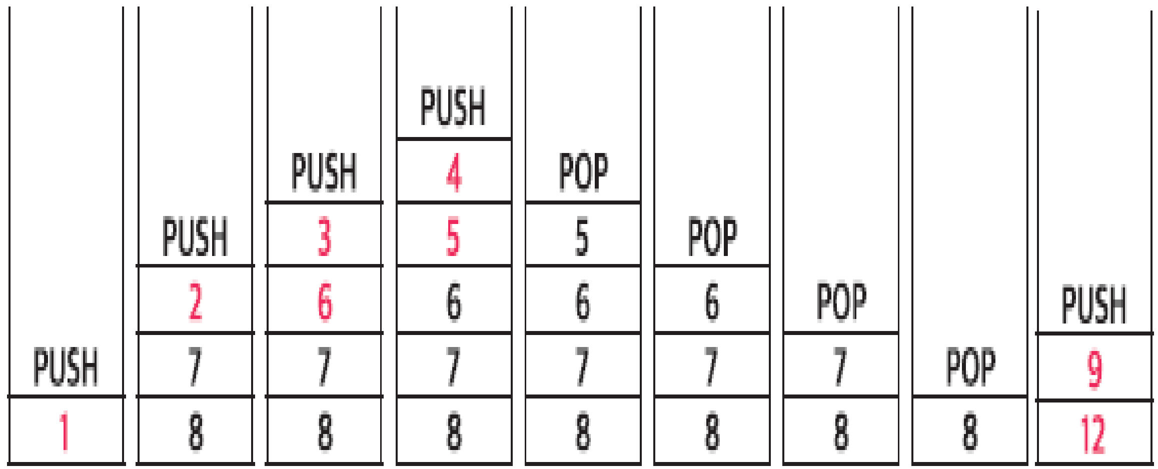
- Spanning tree whose sum of edge weights is minimal
  - The smallest connected graph in terms of edge weights → *How to find it ?*
  - If there is no weight, the number of edges is minimal ?
    - Spanning tree has n-1 edges (for n vertices)



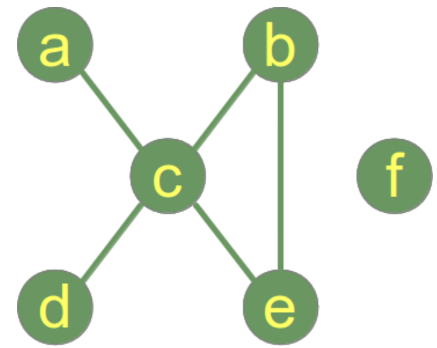
Discussion : DFS with stack

- Searching for 9





DFS with stack : Python implementation




```
def dfs(graph, root, search):
    visited = []
    stack = [root, ]
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.append(node)
            if node == search:
                global found
                found = True
                break
            stack.extend([x for x in graph[node] if x not in visited])
    return visited;


found = False
graph = { "a" : ["c"], "b" : ["c", "e"], "c" : ["a", "b", "d", "e"], "d" : ["c"], "e" : ["c", "b"], "f" : [] }
print(dfs(graph, "a", "b"))
if(found == True):
    print("found")
else:
    print("not found")
```