CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: DATA STRUCTURES – PART I

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Explain what a data structure is, and what it means to design a data structure
- Indicate when a data structure is needed, and specify the data structure that matches the situation
- Describe standard data structures such as *stacks*, *queues*, *singly/doubly linked lists*, and discuss and compare/contrast different implementations of them
- Define graphs, graph representations, and basic graph concepts
- Define trees, tree representations, and basic related concepts

OUTLINE

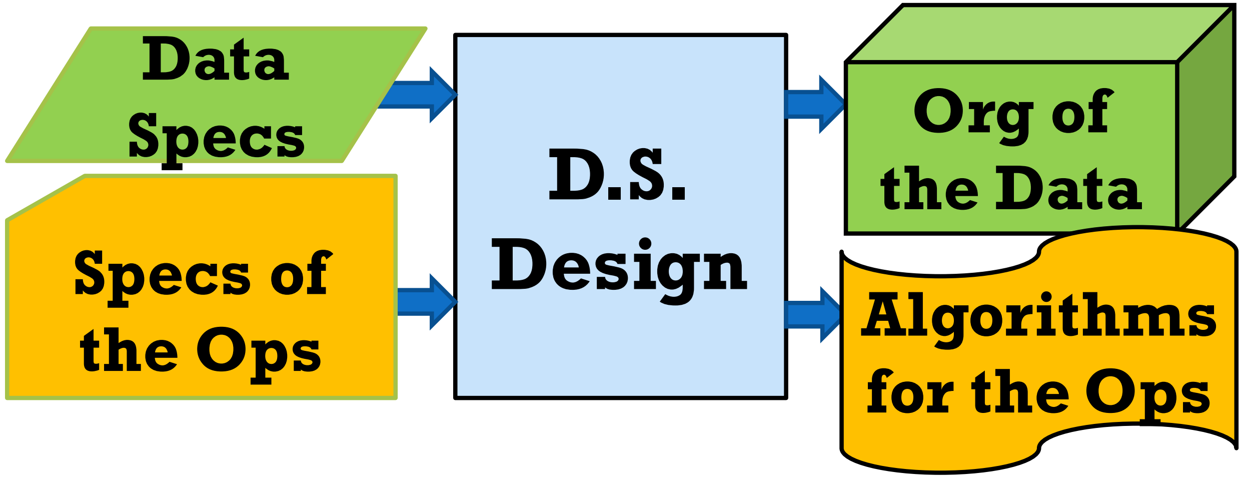
- Preliminaries
    - Definition of data structures
    - Design of data structure
    - When a data structure is needed in the context of algorithm design
- Overview of Stacks and Queues
- Records and Pointers
- Linked Lists
- Graphs
- Trees

WHAT IS A DATA STRUCTURE

- A data structure is
  - An <u>organization</u> of a data set
  - Several <u>operations / actions</u> to be performed on the data set
- Examples of data organizations:
  - Arrays, tables (multidemensional arrays), lists, trees, etc
- Examples of operations/actions:
  - Insert (i.e., add) a new data element
  - Find a data element in the data set
  - Delete an existing element
  - Find the minimum, or the maximum, or the most recently added element

THE DESIGN PROCESS OF A DATA STRUCTURE

- Any design process is an input-output process
  - Input: the specifications ("specs") of what is to be designed
  - Output: A design that fulfills the specs
- Data structure design process
  - Input:
    - Specs of the data set: data type, and possibly data set size
    - Specifications of the operations
  - Output:
    - An organization of the data set
    - An algorithm for each operation



A NOTE ABOUT DATA STRUCTURE DEFINITION AND DESIGN

- The specifications of some data structures include (partially or fully) the actual organization of the data

- Examples: linked lists, binary trees, heaps, etc.

- In such cases, we'll call (in this course) such data structures "data structures with *built-in* organization".

WHEN DO WE NEED A DATA STRUCTURE?

-- IN THE CONTEXT OF ALGORITHM DESIGN --

- A brainstorming exercise

WHEN DO WE NEED A DATA STRUCTURE?

-- IN THE CONTEXT OF ALGORITHM DESIGN --

- If, in an algorithm, one or more operations
  - Are called upon to execute many times on some set of data

- At that point, it may lead to more efficiency (better speed) if
  - That set of data and those operations are implemented by an efficient data structure

STACKS
-- DEFINITION --

- A stack S is a data structure where
  - The data is of any kind (int, float/double, char, strings, etc.)
  - The operations
    - **Push**(S,a): inserts a new data element a into the structure
    - **Pop**(S): deletes (and returns) the most recently added data element
    - **Top**(S): returns (but does not delete) the most recently added data element
- The stack is a data structure that implements the "last in, first out" (LIFO) policy (aka, "last come, first serve")

STACKS
-- IMPLEMENTATION--

Data organization:

- **Array S[1:n], and an index k to the next empty slot, or**
- **A linked list, with a pointer to the most recently added record**

Time: O(1), which means constant

```
Procedure Push(S,a)
begin
        S[k]=a;
        k++;    // what if k==n+1?
end
```

Time: O(1)

```
Function Pop(S)
begin
        if (k==1) then
                return null;
        end if
        k--;
        return(S[k]);
end
```

Time: O(1)

```
Function Top(S)
begin
        if (k==1) then
                return null;
        else
                return(S[k-1]);
        end if
end
```

QUEUES

-- DEFINITION --

- A queue Q is a data structure where
  - The data is of any kind (int, float/double, char, strings, etc.)
  - The operations
    - **enqueue**(Q,a): inserts a new data element a into the structure
    - **dequeue** (Q): deletes (and returns) the oldest (i.e., least recently) added data element
- The queue is a data structure that implements the "first in, first out" (FIFO) policy (aka, "first come, first serve")

QUEUES
-- IMPLEMENTATION--

Data organization:

Initially: tail=n; head=?

- Array S[1:n], an index ***head*** pointing to the oldest element in the queue, and an index ***tail*** pointing to the next empty slot where a new element will be added , OR
- A linked list, with two pointers ***head*** and ***tail***

```
Procedure enqueue(Q,a)
begin
        Q[tail]=a;
        tail--;
end
```

Time: O(1)

If tail==0, what happens?

```
Function dequeue(Q)
begin
        head--;
        return (Q[head+1]);
end
```

Time: O(1)

If head==0, what happens?

- **The array implementation has a lot of issues: Name them?**
- **Better implementations:**
  - Circular arrays;
  - Even better: Linked lists (Why?)

RECORDS AND POINTERS
-- NEED TO EXPAND OUR SYNTAX--

- As we saw in stacks and queues, arrays are not always adequate for implementing data structures
    - Data structures are *dynamic*: they grow and shrink at execution time
    - Whereas arrays are *static* in size: their size is determined at *compile time*
    - Arrays are too simple to represent complicated data organizations such as arbitrary trees
- Therefore, we need to expand the syntax and language structure to allow for
    - Dynamic data structures that grow and shrink at execution without memory limitations
    - Allocation (and release) of memory (of user-defined data type) dynamically as needed
    - Addressing schemes for dynamically allocated memory
- Records and pointers are an important way for meetings those goals

RECORDS AND POINTERS

-- RECORD DEFINITION --

- A record is an aggregate of several elements called *fields* (or *members*), where
  - each field is a variable of a standard type or of a record type
- Syntax (in our pseudo language), and an example of rec def+decl

Syntax for defining a record type:

```
record name
begin
        field declaration;
        …….
        field declaration;
        field declaration;
end
```

Example:

```
record employee
begin
        char name[1:30];
        int SSN;
        char address[1:100];
        float salary;
end
```

```
employee x;
// creates memory for an actual employee record

employee y[1:10];
// creates an array of 10 employee records
```

- In object-oriented programming (OOP), a record corresponds to a *class*

RECORDS AND POINTERS
-- POINTERS --

- A pointers is an address
  - Like a home address, specifying the location of a house
  - But in computers, it is simply an index (integer) to a memory location
- Note the important difference between an object and its address

Object	Address
A house: An actual physical structure with rooms, kitchen, doors, land, etc.	A couple of lines: 123 Main Street Washington, DC 20052
Record: employee x; // occupies a huge chunk of memory	An index: Single integer index of the first byte of x in memory

RECORDS AND POINTERS

-- POINTERS VISUALIZATION --

RECORDS AND POINTERS

-- CONTRAST B/W REC DEF, REC DECL, AND POINTERS --

<table>
<tr><th>Type Definition</th><th>Declaration</th><th>Pointer</th></tr>
<tr><td><pre>
record employee
begin
      char name[1:30];
      int SSN;
      char address[1:100];
      float salary;
end
</pre></td><td>employee x;

// occupies a big chunk of memory</td><td>Address of x;

// Single <u>integer</u> index of the first byte of x in memory</td></tr>
<tr><td>Record definition is like a Blueprint</td><td>A physical object matching the blue print</td><td>The index of the of the 1st byte of the physical record in memory</td></tr>
<tr><td> </td><td></td><td>123 Main Street
Washington, DC 20052
USA</td></tr>
</table>

ACCESS TO RECORDS AND THEIR FIELDS

- Syntax: if X is a record and F is a field in X, we access F using the dot syntax
  - X.F accesses the field F of record X
- Example:

```
employee x;  // allocates empty memory for x
x.name="John Smith";      // fill in field "name" of x
x.SSN=123456789;          // fill in field "SSN" of x
x.salary=50000;           // fill in field "salary' of x
x.address="123 main St\n DC, USA";
```

- Examples:
  - Overwrite the SSN field of x with value 124555678: x.SSN = 124555678;
  - Change the salary value (of $100,000) to x: x.salary=100000;
  - Give x a raise of $5000: x.salary = x.salary+5000;
  - Read the SSN of x and assign that value to a new var: int S=x.SSN;

USING ADDRESSES OF RECORDS

- Suppose x is a declared employee record
- To get the address of x:
  1. Declare a variable p of type "employee address", such as any of:
     - **employee *** p; **employeePointer** p; **employeePtr** p; **employeeAdr** p;
  2. Assign to p the address of x
     - p=&x;
     - But it is OK to write
       - p=x; or p=address(x); or anything that says that p is the address of x
- Accessing the fields using the record address (use the arrow or dot syntax)
  - p -> salary; but still OK to write p.salary;

CREATING RECORDS DYNAMICALLY

-- WHY --

- In many algorithmic situations
  - Data are added and deleted during execution
  - The number of additions/deletions vary from execution to execution (i.e., from input to input)
  - The maximum size of the data (structures) may not be known ahead of time
  - For example, we may not know how big a queue or a stack will grow
- Therefore, we need a mechanism that
  - Allows us to create (reserve) memory dynamically (i.e., during execution), such as allocation of new records of a certain pre-defined type, as needed during execution
  - Manipulate (i.e., read and write) the dynamically created records

CREATING RECORDS DYNAMICALLY
-- HOW--

- Use **new** to create a record of a predefined type, returning the address of the allocated memory
  - Syntax: recordPtr p=**new**(predefined-record-type)
  - Example: employeePtr p=new(employee);
  - Details:
    - **new** is a call to he OS to find+reserve a free chunk of memory large enough to hold a full record
    - Once done, the OS returns to the calling the address of the allocated record
      - Ex: After "employeePtr p=new(employee);" is done, p has the address of the allocated record
    - The allocated record is empty
    - You can now fill in the individual fields with data of your own

Example:

```
employeePtr p = new (employee);
p.name="John Smith";
p.SSN=312959876;
p.salary=200000;
p.address="345 Maple Street\n Ontario";
p.next=null;    // assumed null by default
```

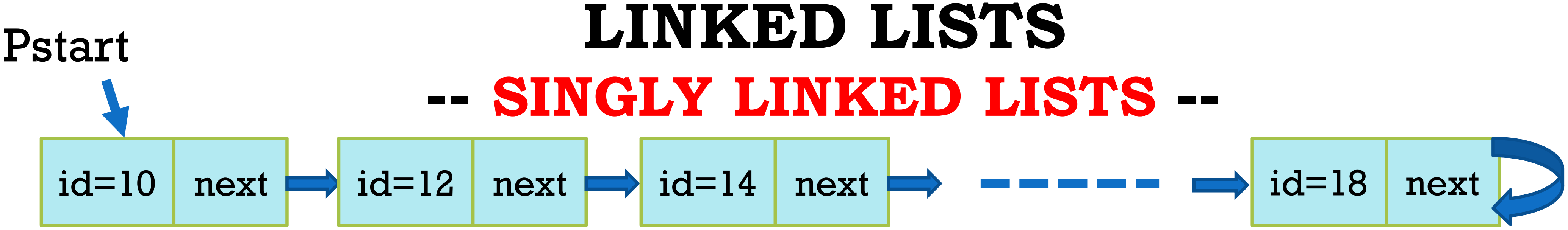
SELF-REFERENTIAL RECORDS

- A self-referential record is a record where at least one of the fields is of type “pointer to a record of the same type”
- Example: modify the employee record so it has

```
Define a record type employee:
record employee
begin
      char name[1:30];
      int SSN;
      char address[1:100];
      float salary;
      employeePtr next; // a new field
end
```

LINKED LISTS

-- SINGLY LINKED LISTS --



- A singly linked list is data structure (with built-in organization) where:
  - The organization is a sequence of self-referential records
  - Every record has a field that points to the next record in the sequence
    - Records usually hold data field(s), as pre-defined by the user
  - A pointer *Pstart* that points to the first record is part of the data structure
  - Optionally, a pointer *Pend* that points to the end-record of the list is included
  - The operations are (typically): **insert**(…), **find**(…), **delete**(…)
- Useful in many situations (where navigation can be done in one direction), including the implementation of unlimited stacks and queues

LINKED LISTS

-- SPECS OF SINGLY LINKED LIST OPERATIONS --

- **Find**(L, key): finds a record in the list, whose uniquely identifying "key" field has the specified value, and returns the address of that record if found, null otherwise
- **Find**(L, int k): returns the address of the kth record in the list (or null if k is larger than the size of list)
- **Insert**(L, a): dynamically creates a new record, adds the data 'a" to the data field(s) of the record, and put the record at the end (or start) of the list
- **Insert**(L, a,k): like above, except the new record is inserted as the kth record in the list (if k is non-negative value ≤ the size of the list)
- **Delete**(L, p), **Delete**(L,key): delete the record whose address is p or whose key (uniquely identifying value) is the specified value, and "smooth out" the gap

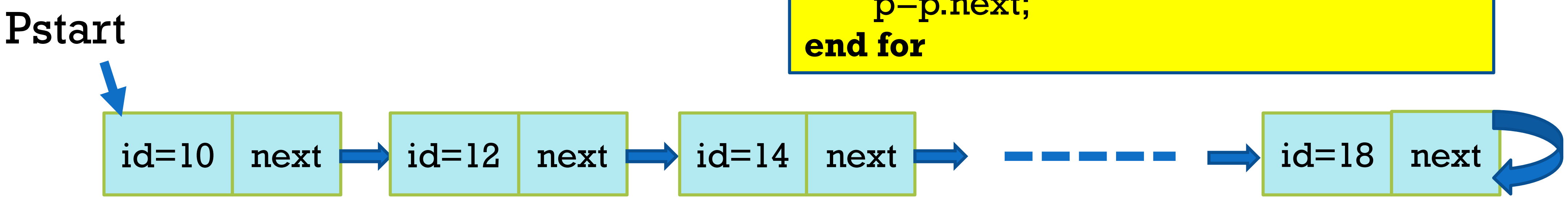
LINKED LISTS

-- AN EXAMPLE OF SINGLY LINKED LISTS --

- Create a new (simple) record type (call it simpleRec), and then create a list:

```
//Define a new record type:
record simpleRec
begin
    int id;
    simpleRecPtr next;
end
```

```
// create and populate a list of 10 records
simpleRecPtr  Pstart = new(simpleRec);
Pstart.id=10;
simpleRecPtr p = Pstart ;
for n=1 to 9 do
    simpleRecPtr q=new(simpleRec);
    q.id=10+2*n;
    p.next=q;
    p=p.next;
end for
```



LINKED LISTS

-- IMPLEMENTATION OF SINGLY LINKED LIST OPS --

Time: O(|L|)

```
Function Find(L,a)
Begin
    Pointer p=Pstart of L;
    While(p!=null && p.key!=a) do
        p=p.next;
    End while
    Return p;
End Find
```

Time: O(1)

```
Procedure Insert(L,a)   // inserts at the start
Begin
    Pointer p=new(record-type);
    p.data=a;
    p.next=Pstart of L;
    Pstart = p;
    // the newly created record is the new start
End Insert
```

Time: O(|L|)

```
Procedure Delete(L,a)
Begin
    Pointer p=Pstart of list L;
    Pointer q; // one step behind p
    While(p!=null && p.key!=a) do
        q=p;      p=p.next;
    End while
    // if record is not found
    If (p==null) then return; end if
```

```
    // continue Delete (L,a) here
    If (p==Pstart) then     // 1st record to be deleted
        Pstart=Pstart.next;
    Else
        q.next=p.next;  // bypass record p
    End if
    release(p);
    // optional, frees memory of deleted record
End
```

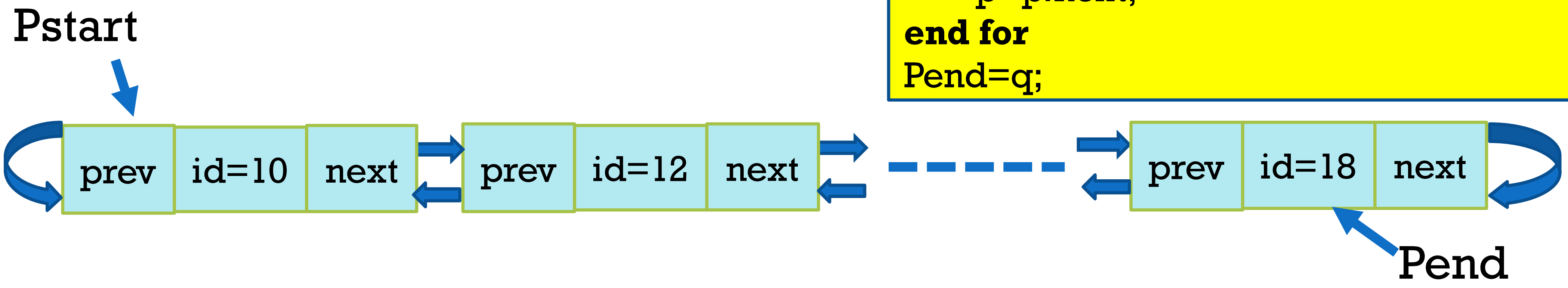
LINKED LISTS

-- DOUBLY LINKED LISTS --

- Like singly linked lists except that each record has a field that a field that points to the next record and another field that points to the previous record

```
//Define a new record type:
record simpleRec
begin
      int id;
      simpleRecPtr next;
      simpleRecPtr prev;
end
```

```
// create abd polulate a list of 10 records
simpleRecPtr  Pstart = new(simpleRec);
Pstart.id=10;
simpleRecPtr p = Pstart ,   q;
for n=1 to 9 do
      simpleRecPtr q = new(simpleRec);
      q.id=10+2*n;
      p.next = q;    q.prev = p;
      p=p.next;
end for
Pend=q;
```



CHECK YOUR UNDERSTANDING: QUIZ

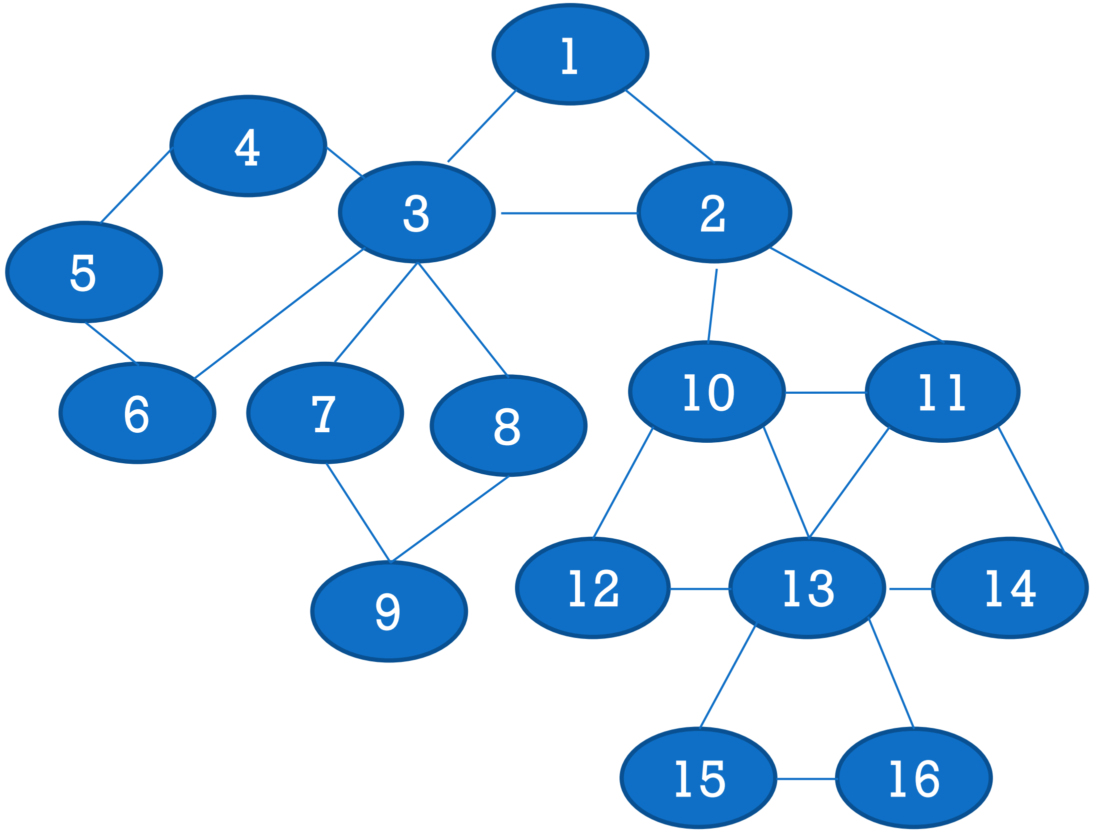
- Time to search in a doubly linked list: a. $O(|L|^2)$, b. $O(|L|)$, c. $O(1)$
- Time to delete in a doubly linked list: a. $O(|L|^2)$, b. $O(|L|)$, c. $O(1)$
- Space complexity of a singly linked list: a. $O(|L|)$, b. $O(1)$, c. $O(\log |L|)$
- Both singly linked lists and doubly linked lists have the same Big-O space complexity: a. YES, b. NO
- Array implementation of stacks and queues can run out of memory (assuming you computer has infinite RAM):: a. YES, b. NO
- Linked-list implementation of stacks and queues can run out of memory (assuming your computer has "infinite" RAM): a. YES, b. NO

GRAPHS

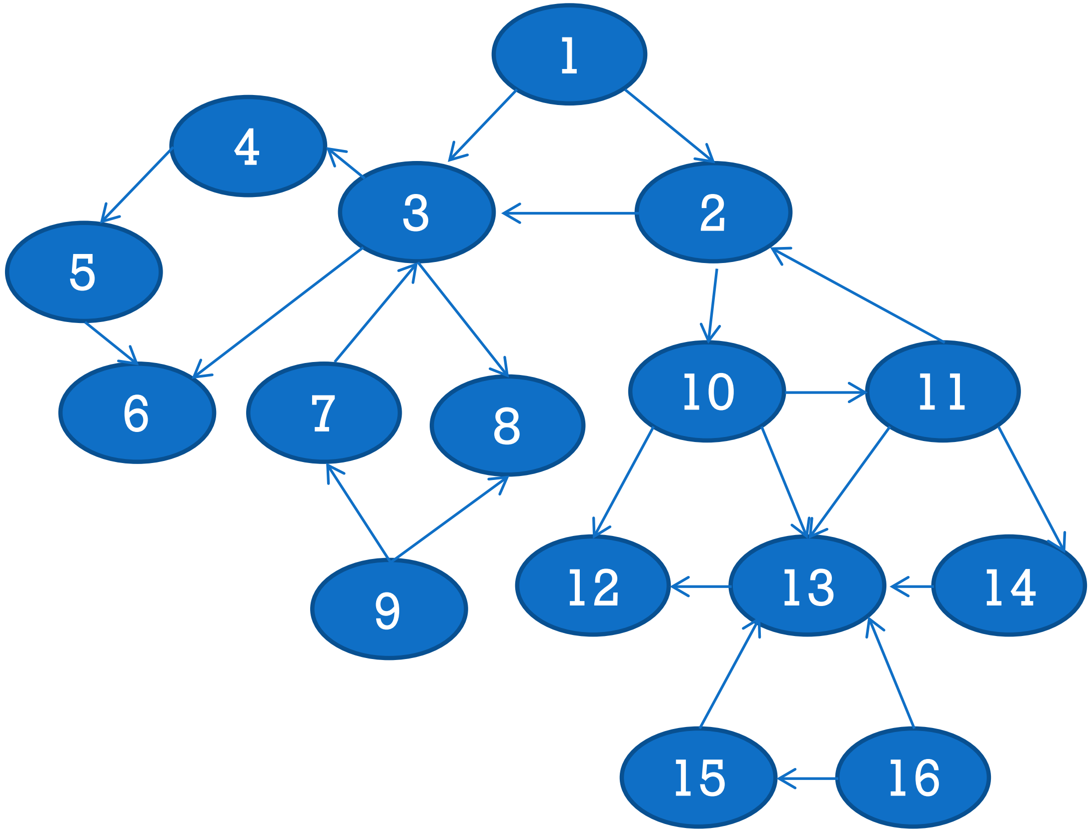
- Definition: A graph G=(V,E) consists of a finite set V, whose elements are called nodes, and a set E, which is a subset of V x V. The elements of E are called edges.
- Directed vs. undirected graphs:
  - If the directions of the edges are of significance, that is, (x,y) is different from (y,x), the graph is called a ***directed graph*** (or ***digraph***).
  - Otherwise, the graph is called ***undirected***
- Weighted (di)graph: It is a (di)graph where every edge has a associated with it a number called its weight.

GRAPH EXAMPLE

- V={1,2,3,…,16},E={(1,2) , (1,3) , (2,3), (3,4), (4,5), (5,6) , (6,3), …}



Undirected



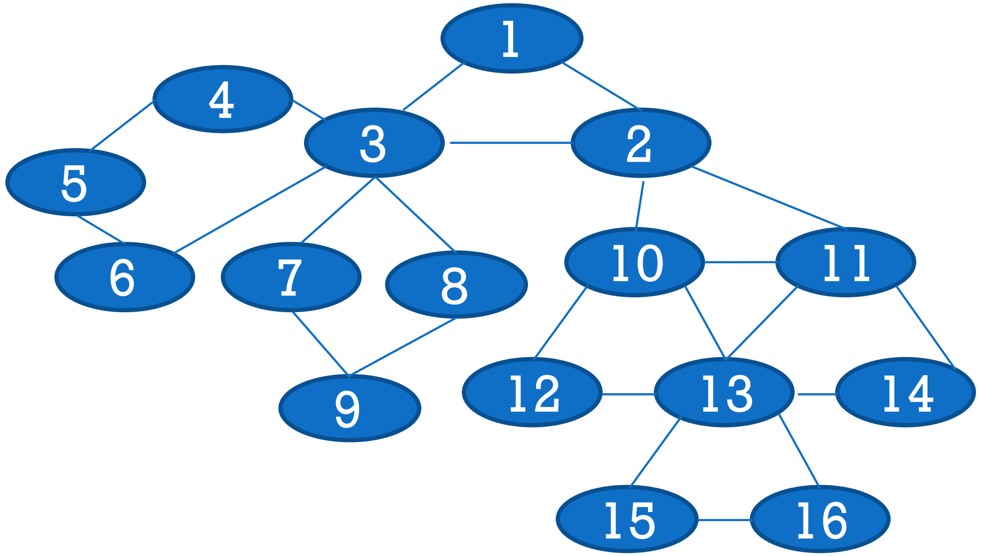
Directed

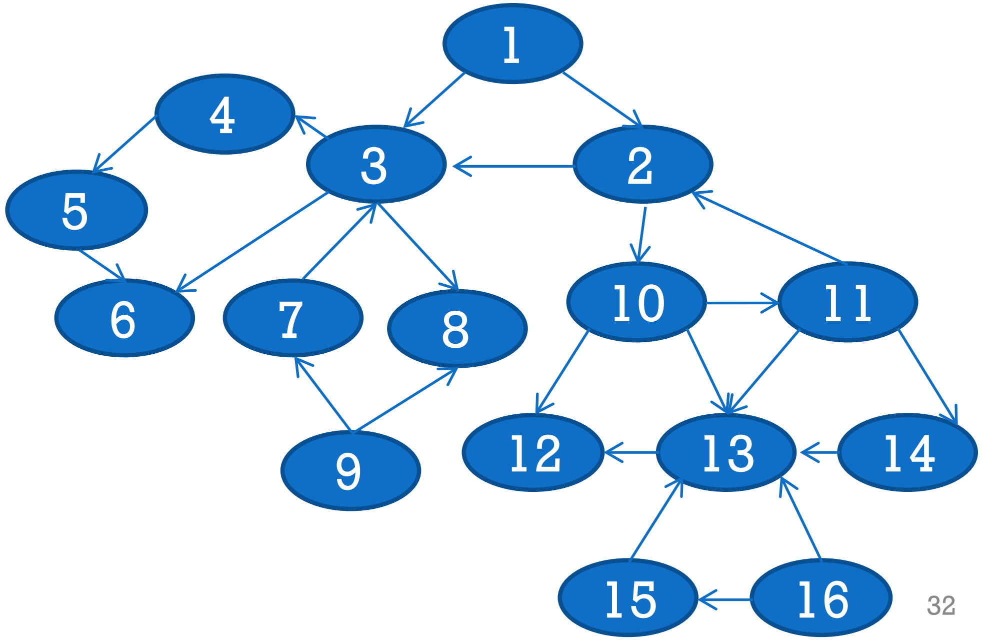
WHAT CAN GRAPHS REPRESENT

- A graph can represent a map
  - The nodes are points of interest (countries, cities, homes, etc.)
  - The edges are transportation lines (e.g., roads, train tracks, etc.)
  - If undirected, the edges are two-way streets; if directed, 1-way St.
- A graph can represent a computer/communication network
  - The nodes are computers/switches
  - The edges are communication lines
  - If undirected, the links are bidirectional; id directed, unidirectional
- A graph can represent a type of relation between entities
  - The nodes are entities/objects/concepts
  - The edges are designated relations between the entities (e.g., friend of, spouse of, boss of, acquaintance of, generalization of, special case of, etc.)

GRAPHS CONCEPTS

- **Adjacency**:
  - If (x,y) is an edge, then x is said to be ***adjacent to*** y, and y is ***adjacent from*** x.
  - In undirected graphs, if (x,y) is an edge, we just say that x and y are adjacent (or x is adjacent to y, or y is adjacent to x). Also, we say that x is the neighbor of y.





GRAPHS CONCEPTS

-- ADJACENCY, INCIDENCE, DEGREE --

Undirected Graphs	Directed Graphs (Digraphs)
Adjacency: If (x,y) is an edge, then x is said to be ***adjacent to*** y, and y is ***adjacent from*** x. We can also say that x and y are adjacent, and x and y are neighbors	**Adjacency**: If (x,y) is an edge, then x is said to be ***adjacent to*** y, and y is ***adjacent from*** x.
Degree: The degree of a node x is the number of neighbors of x.	**Indegree:** the indegree (fan-in) of a node x is the number of nodes adjacent to x, i.e., the number of edges coming to x **Outdegree**: the outdegree (fan-out) of x is the number of node adjacent from x, i.e., number of edges leaving x.
Incidence: An edge e=(x,y) is said to be ***incident to*** y and ***incident from*** x	

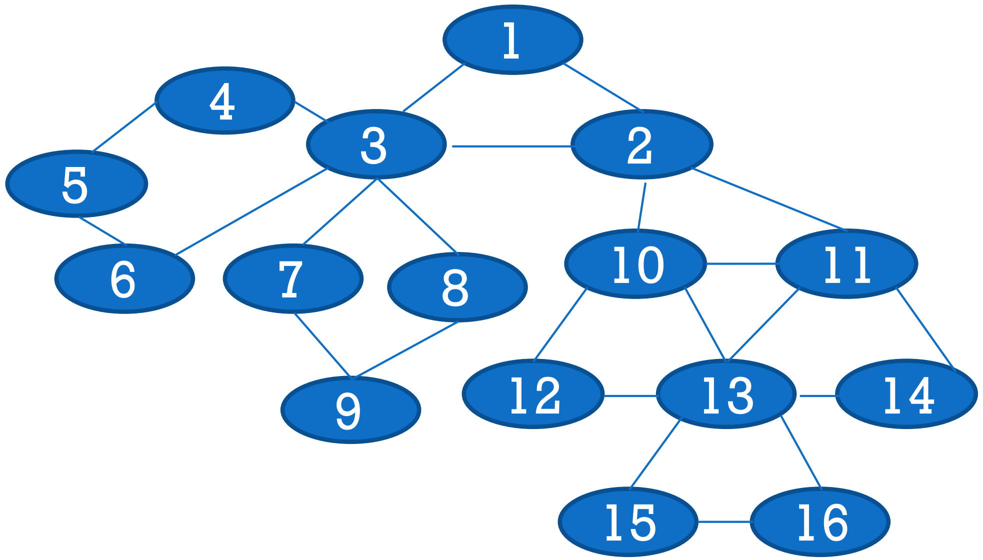
GRAPHS CONCEPTS

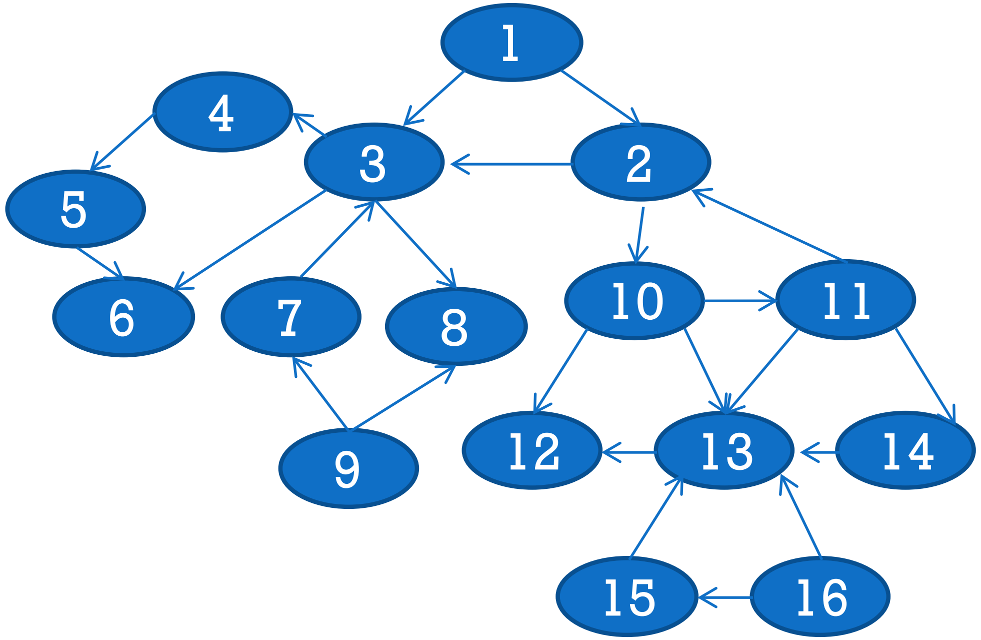
-- PATHS, CYCLES, DISTANCE --

<table>
<tr><th>Undirected Graphs</th><th>Directed Graphs (Digraphs)</th></tr>
<tr><td colspan="2">Path: A path from a node x to a node y is a sequence of nodes $x, x_1, x_2, \dots, x_n, y$ such that x is adjacent to x_1, x_1 is adjacent to x_2, …, and x_n is adjacent to y.
Note: A path can go through a node multiple times.
A simple path: It is a path where no node repeats.</td></tr>
<tr><td colspan="2">Path length: The length of a path is the number of its edges (not number of nodes), if the graph is unweighted. If the graph is weighted, the length of a path is the sum of the weights of its edges.</td></tr>
<tr><td colspan="2">Distance: the <i><u>distance</u></i> from a node x to a node y in a (di)graph is the length of the <u>shortest</u> path from x to y.</td></tr>
<tr><td colspan="2">Cycle: A cycle is a path that begins and ends at the same node.</td></tr>
</table>

GRAPHS CONCEPTS

-- PATHS, CYCLES, DISTANCE --





GRAPHS CONCEPTS
-- CONNECTIVITY--

Undirected Graphs	Directed Graphs (Digraphs)
Connected: a graph is connected if for every pair of nodes there is at least one path between them.	**Strongly Connected**: a digraph is strongly connected if there is at least one path from every node x to every node y
Disconnected: a graph is disconnected if it is not connected	**Weakly connected**: a digraph is weakly connected if the underlying undirected graph (derived by ignoring the edge directions) is connected.
Connected component (of a graph G**)**: It is any maximal connected subgraph of G. A **subgraph** of G is a subset of nodes with all their edges inherited from G. **Maximal**: If any other node from G is added (along with its incident edges) to the subgraph, the latter becomes disconnected	**Strongly connected component**: It is any maximal strongly connected subgraph of G

GRAPH REPRESENTATIONS

- There are two standard representations of (di)graphs
  - Adjacency matrix
  - Adjacency lists
- Let G=(V,E) be a (di)graph where V={1,2,…,n}
- Adjacency Matrix: An $n \times n$ matrix $A[1{:}n, 1{:}n]$ where

$$A[i,j] = \begin{cases} 1 \text{ if } (i,j) \in E, \text{that is, } (i,j) \text{ is an edge} \\ 0 \text{ if } (i,j) \notin E, \text{that is, } (i,j) \text{ is not an edge} \end{cases}$$

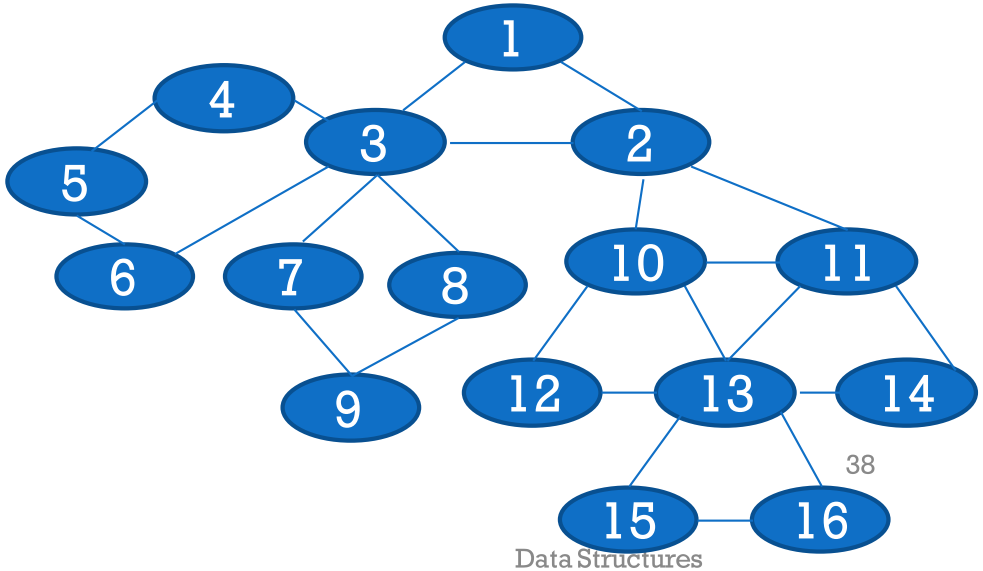
- If the (di)graph is weighted, the matrix becomes weight matrix $W[1{:}n, 1{:}n]$:

$$W[i,j] = \begin{cases} \text{weight of edge } (i,j) \text{ if } (i,j) \text{ is an edge} \\ \infty \text{ if } (i,j) \text{ is not an edge} \end{cases}$$

GRAPH REPRESENTATIONS

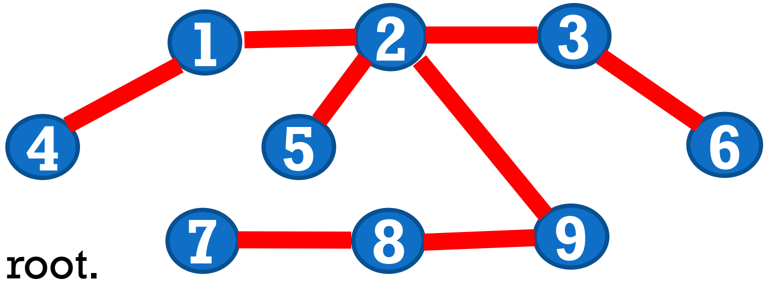
-- ADJACENCY LISTS --

- Let G=(V,E) be a (di)graph where V={1,2,…,n}
- Adjacency Lists representation:
  - An array $A[1:n]$ of pointers to linked lists, where
  - $\forall$ nodes i, $A[i]$ is pointer to the start of a linked list containing all the nodes adjacent from node i
  - The order of the records (one record per node) in each list is arbitrary
  - One convenient order: increasing order

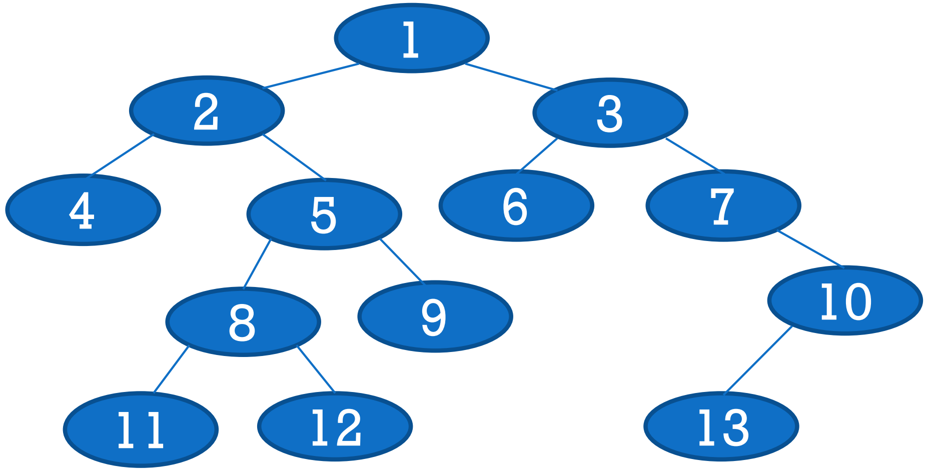


TREES

- **Definition:** A ***tree*** is an undirected, connected, acyclic graph
  - In a tree, there is exactly one simple path between every pair of nodes



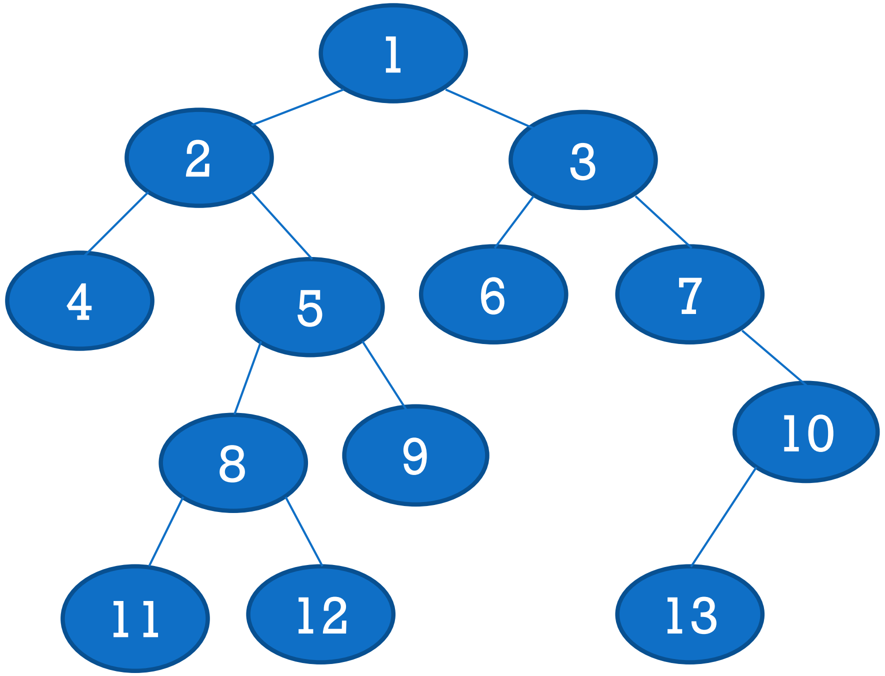
- **Definition**: A ***rooted tree*** is a tree where one node is designated as root.



- **Hierarchical layout of a rooted tree**:
  - By holding a rooted tree at its root, and letting the other nodes descend from it, we get a hierarchical structure.
  - Note that there is exactly one path from the root to any node

TREE CONCEPTS

-- PARENTS, CHILDREN, ANCESTORS, DESCENDANTS --

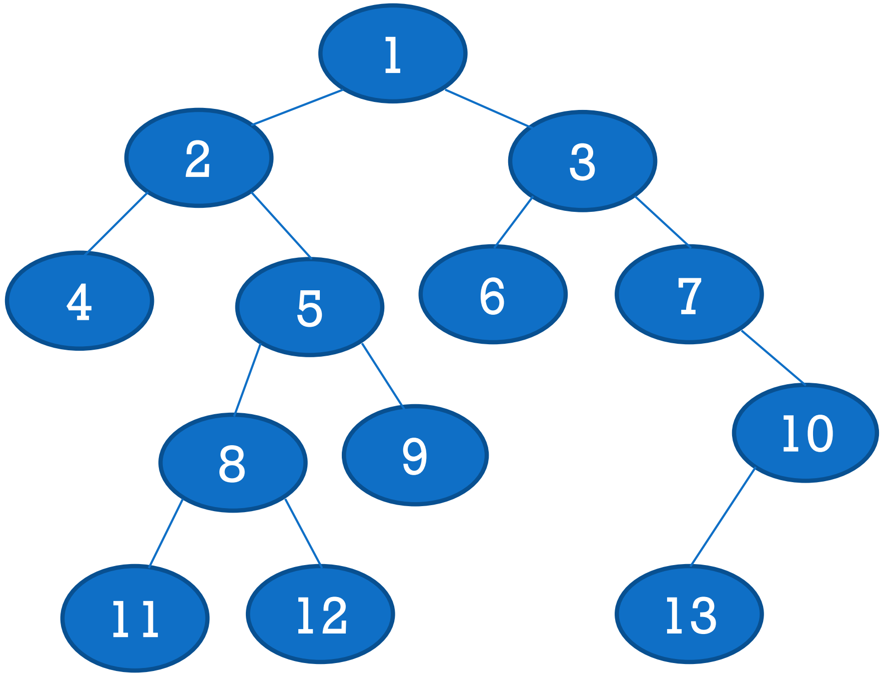


- Let T be a rooted tree rooted at node r
- Every node x (other than r) hangs down from a single node, called ***parent*** of x
- The nodes adjacent to x and hang down from it are called ***children*** of x
- A ***leaf*** is a node that has no children.
- An ***internal node*** is any non-leaf
- The ***ancestors*** of x are the nodes on the path from x to r, including x and r
- The ***proper ancestors*** of x are all the ancestors of x except x itself
- The ***descendants*** of x are: x, its x, their children, and so on all the way down
- The ***proper descendants*** of x are all the descendants of x except x itself

TREE CONCEPTS

-- SUBTREES, DEPTH, HEIGHT --

- Let T be a rooted tree rooted at node r
- ***Subtree***: the subtree rooted at x is the tree consisting of x and all its descendants

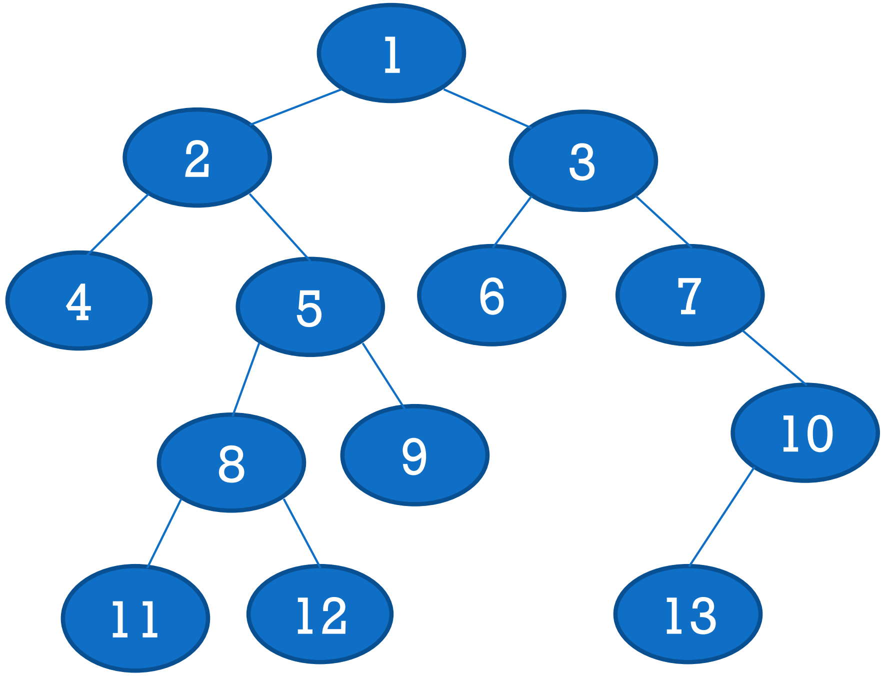


- The ***depth*** of node x is the distance from the root r to node x
- The ***height*** of x is the distance from x to the farthest descendant of x
- The ***height*** (or ***depth***) of the tree is the height of the root

TREE CONCEPTS

-- LEVELS --

- Let T be a rooted tree rooted at node r, in a top-down hierarchical layout
- The nodes clearly partition into levels:
  - The top level contains just the root, and it is labeled level 0
  - The next level, labeled level 1, contains the children of the root
  - The level after that, labeled level 2, contains the grandchildren of the root
  - and so on.



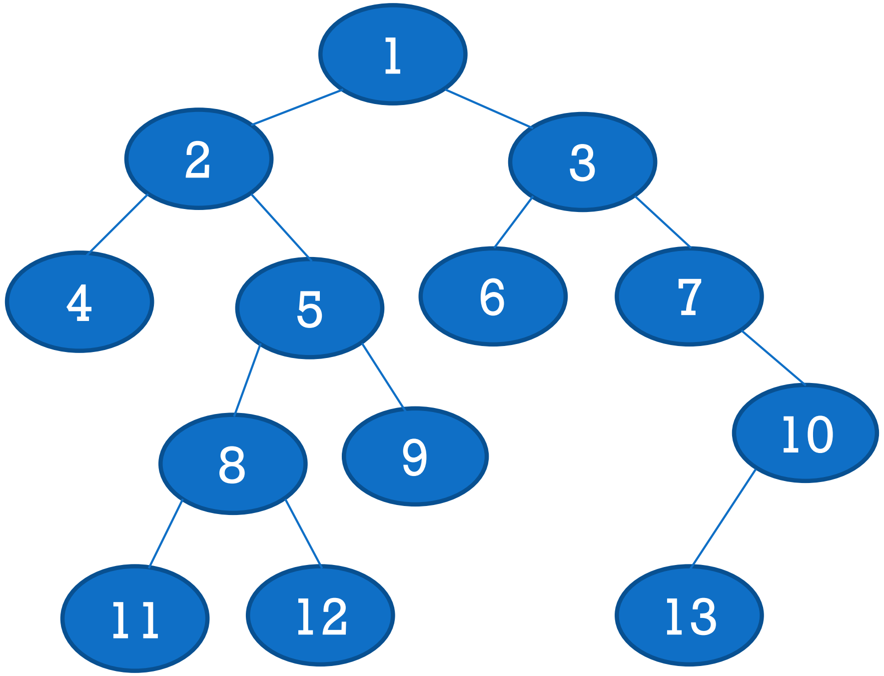
- Observation: The ***depth*** of node x is the label of its level
- The ***height*** (or ***depth***) of the tree is the label of its lowest level

CHECK YOUR UNDERSTANDING: QUIZ

- In a rooted tree, the depth of x = height of x: Yes or No?
- In a rooted tree, the depth of the tree = height of x the tree: Yes or No?
- In a rooted tree, <u>not</u> laid out in a top-down hierarchy:
  - A leaf is: (a) Any node of degree 1; (b) the root if it is of degree 0, and any node of degree 1 other than the root; (c) Any node of degree of degree 0
  - Depth of x is: (a) distance from root to x, (b) index of x, (c) distance from x is a leaf descendant of x
  - Height of x is: (a) distance from root to x, (b) distance from x is a closest leaf reachable from of x, (c) distance from x is a farthest leaf reachable from x
  - Depth of the tree is: (a) the radius of the graph from the root (i.e., largest distance from r to any node), (b) the distance from r to the closest leaf

BINARY TREES

- **Definition:** A binary tree is a rooted tree where every node has at most two children
- For convenience, the children of each node are designated as ***left child*** and ***right child***
- A node can have 2 children, a left child only, a right child only, on none



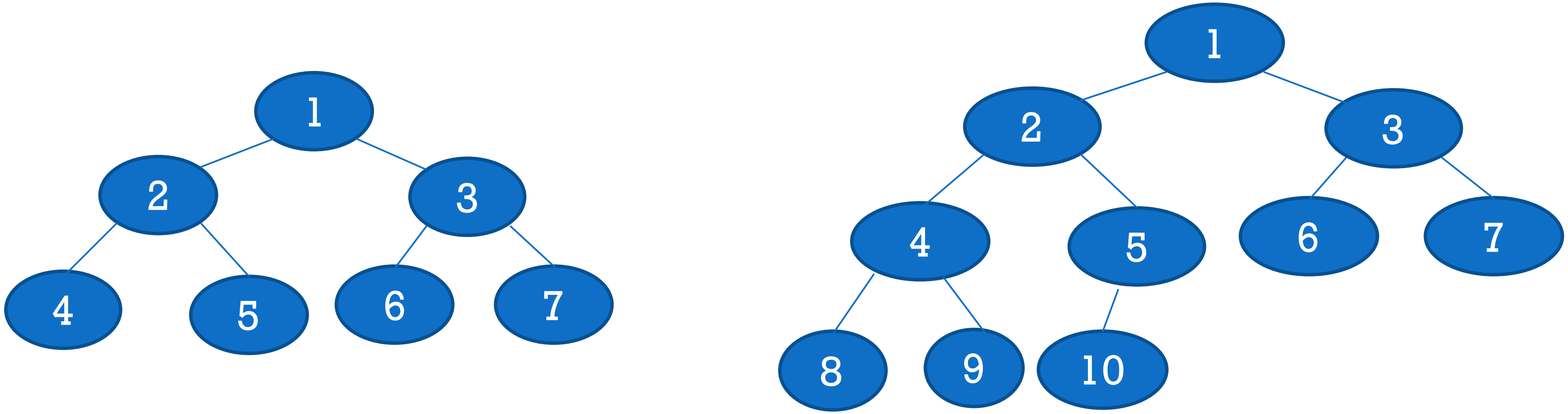
- Representation: Every node can be represented as a record of at least three fields
  - **Data** (could be one or more fields storing data)
  - **Left** (a pointer pointing to the left child, or null if there is no left child)
  - **Right** (a pointer pointing to the right child , or null if there is no right child)
  - **Parent** (Optional, pointing to the parent node)

BINARY TREES
-- SPECIAL CASES --

- **Definition**: A ***perfect binary tree*** is a binary tree where every non-leaf has two children and all the leaves are at the same level
- **Exercise**: Show that the number of nodes in level i of a perfect binary tree is 2^i. Show also that a perfect binary tree of height h has $2^{h+1} - 1$ nodes
- The **canonical labeling of a perfect binary tree**: It is a labeling of the nodes from top to bottom, left to right, starting with the root being labeled 1
- **Definition**: An ***almost complete binary tree*** of n nodes is the binary tree consisting of the first n nodes of a perfect binary tree

BINARY TREES

-- SPECIAL CASES --



BINARY TREES

-- PERFECT/ALMOST COMPLETE BINARY TREES--

- **Observation**: In a canonically labeled perfect/almost complete binary tree:
  - The labels of the children of node i are 2i and 2i+1
  - The label of the parent of i is $\lfloor \frac{i}{2} \rfloor$ (integer division of i by 2).
- **Observations about almost complete binary trees**:
  - If the bottom level is removed, the tree becomes a perfect binary tree.
  - The nodes in the bottom level are packed to the left end of the tree without any "holes"
- Array implementation of **almost complete binary tree (**of n nodes): An array A[1:n] where A[i] stores the data of node labeled i.

K-ARY TREES
-- EXTENSIONS OF BINARY TREES --

- **Definition**: Given a positive integer k>1, a ***k-ary tree*** is a rooted tree where every nose has at most k childen
  - Special case: When k=3, the tree is called a ***ternary tree***
  - Caution: A 4-ary tree is not called quad-tree. Rather a *quadtree* is a tree where every internal node has **exactly** (rather than *at most*) 4 children.
- **Definition**: A perfect k-ary tree is a k-ary tree where internal node has exactly k children, and all the leaves are at the same (bottom) level
- Exercises:
  1. Think of a definition of almost complete k-ary trees
  2. Think of canonical labeling of perfect /almost complete k-ary trees
  3. Think if and how perfect / almost complete k-ary trees can be implemented with arrays