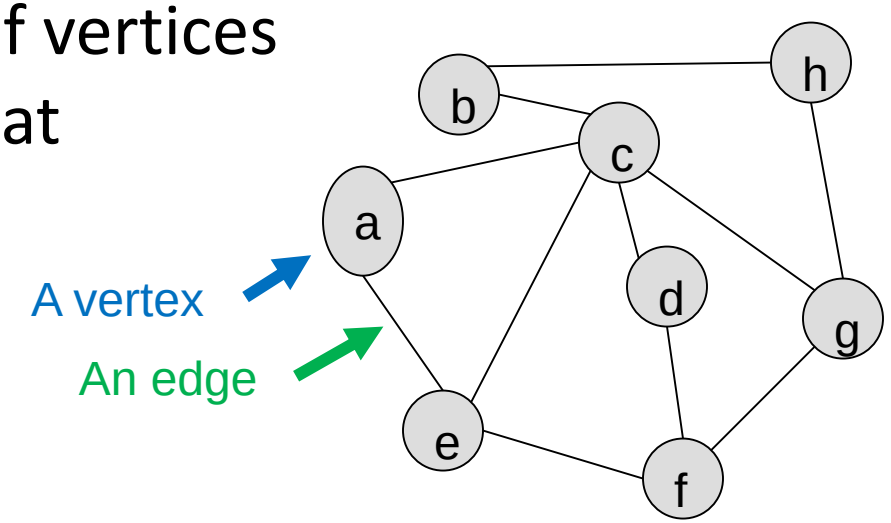
CSCI 104
Graph Representation and Traversals

Mark Redekopp
David Kempe
Sandra Batista

GRAPH REPRESENTATIONS

Graph Notation

- A **graph** is a collection of vertices (or nodes) and edges that connect vertices



V
a
b
c
d
e
f
g
h

|V|=n=8

E
(a,c)
(a,e)
(b,h)
(b,c)
(c,e)
(c,d)
(c,g)
(d,f)
(e,f)
(f,g)
(g,h)

|E|=m=11

- Let **V** be the set of vertices
- Let **E** be the set of edges
- Let **|V| or n** refer to the number of vertices
- Let **|E| or m** refer to the number of edges

Graphs in the Real World

- Social networks
- Computer networks / Internet
- Path planning
- Interaction diagrams
- Bioinformatics

Basic Graph Representation

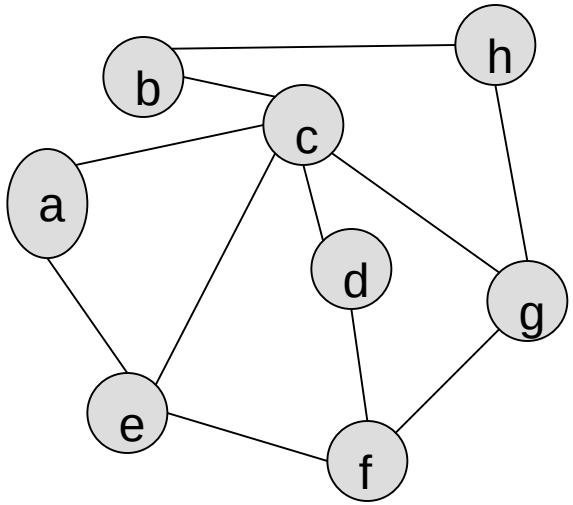
- Can simply store edges in a list
  - Unsorted
  - Sorted

V
a
b
c
d
e
f
g
h

|V|=n=8

E
(a,c)
(a,e)
(b,h)
(b,c)
(c,e)
(c,d)
(c,g)
(d,f)
(e,f)
(f,g)
(g,h)

|E|=m=11



Graph ADT

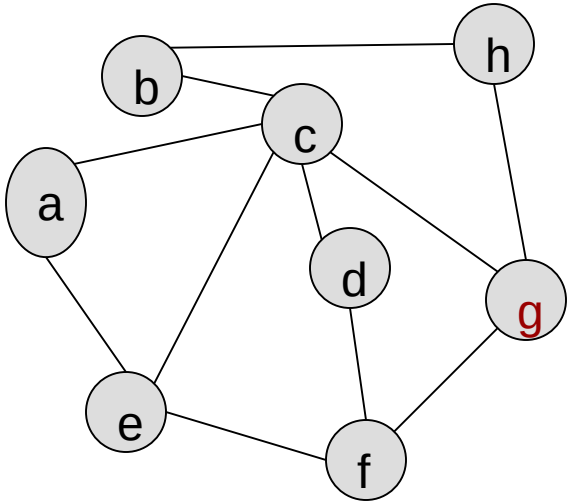
- What operations would you want to perform on a graph?
- `addVertex() : Vertex`
- `addEdge(v1, v2)`
- `getAdjacencies(v1) : List<Vertices>`
  - `Returns any vertex with an edge from v1 to itself`
- `removeVertex(v)`
- `removeEdge(v1, v2)`
- `edgeExists(v1, v2) : bool`

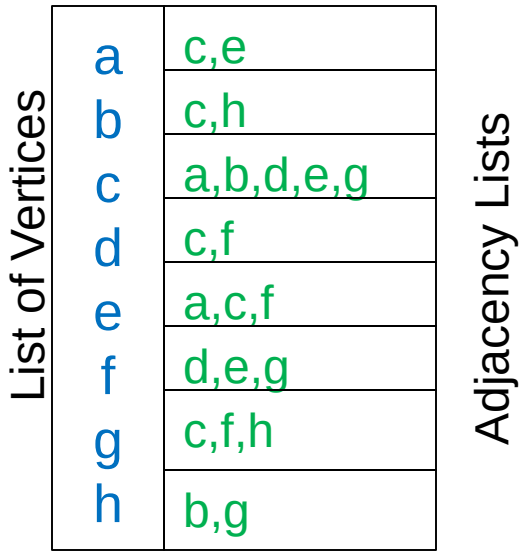
```
#include<iostream>
using namespace std;

template <typename V, typename E>
class Graph{


};
```

Perfect for templating the data associated with a vertex and edge as V and E

More Common Graph Representations

- Graphs are really just a list of lists
  - List of vertices each having their own list of adjacent vertices
- Alternatively, sometimes graphs are also represented with an adjacency matrix
  - Entry at (i,j) = 1 if there is an edge between vertex i and j, 0 otherwise





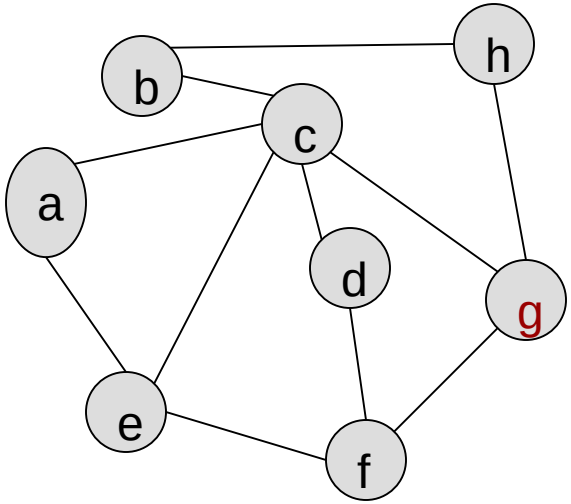
How would you express this using the ADTs you've learned?

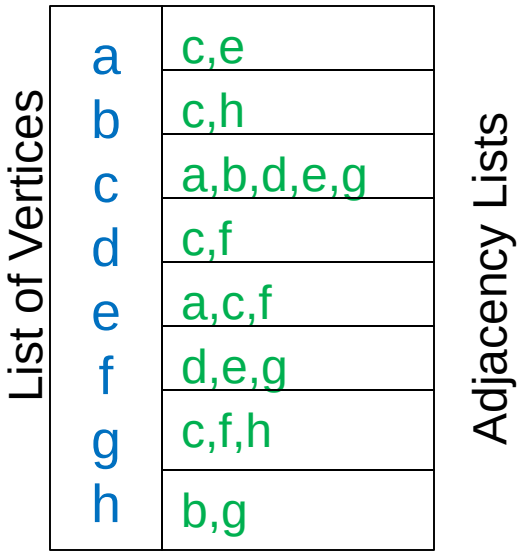
	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Let |V| = n = # of vertices and
  |E| = m = # of edges
- Adjacency List Representation
  - O(_____) memory storage
  - Existence of an edge requires O(_____) time
- Adjacency Matrix Representation
  - O(_____) storage
  - Existence of an edge requires O(_____) lookup





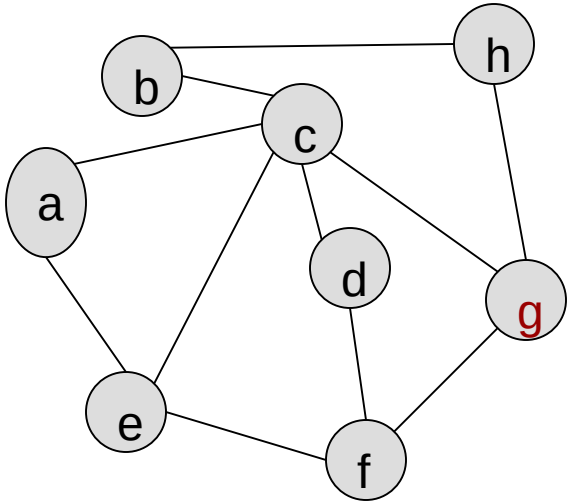
How would you express this using the ADTs you've learned?

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Let |V| = n = # of vertices and |E| = m = # of edges
- Adjacency List Representation
  - O(|V| + |E|) memory storage
  - Define **degree** to be the number of edges incident on a vertex (deg(a) = 2, deg(c) = 5, etc.
  - Existence of an edge requires searching the adjacency list in O(deg(v))
- Adjacency Matrix Representation
  - $O(|V|^2)$ storage
  - Existence of an edge requires O(1) lookup (e.g. matrix[i][j] == 1)



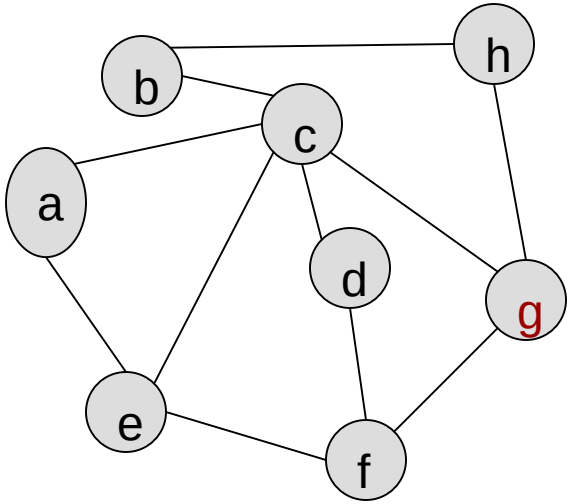
List of Vertices	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

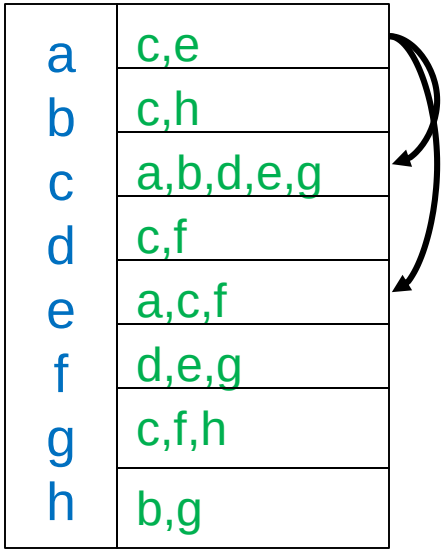
	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a…
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b



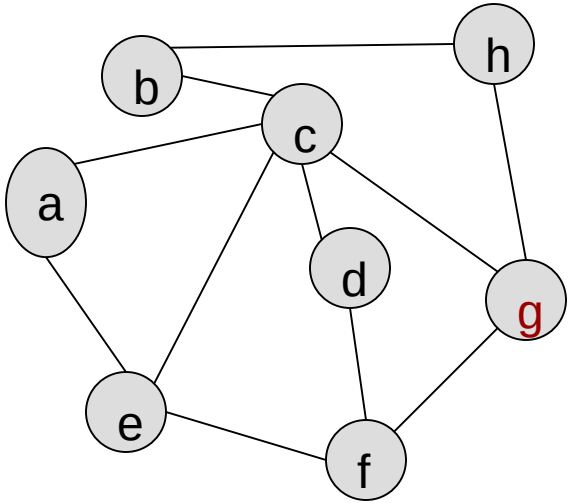


	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

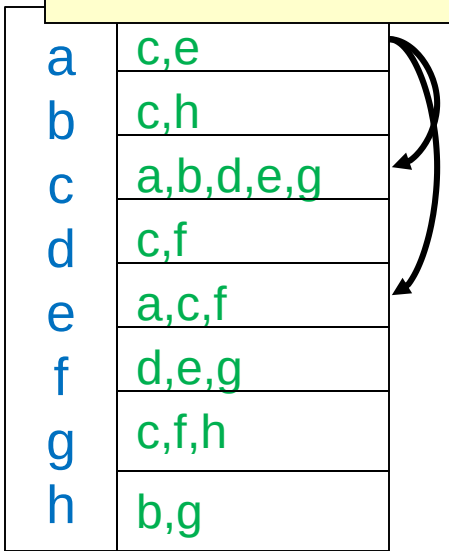
Adjacency Matrix Representation

Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a…
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b




```
int sum = 0;
for(int i=0; i < n; i++){
  sum += adj[src][i]*adj[i][dst];
}
if(sum > 0) // two-hop path exists
```

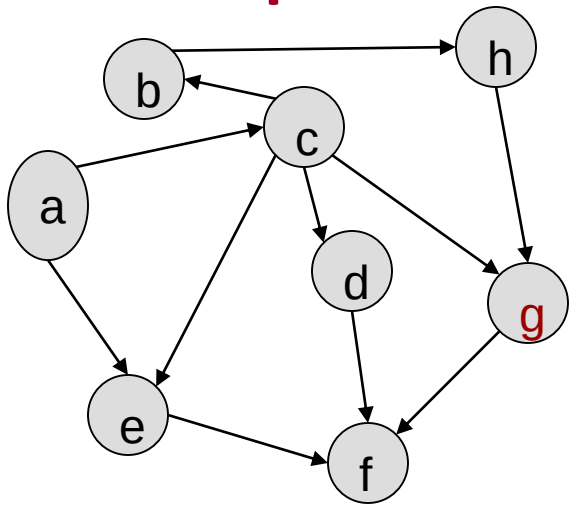


	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Directed vs. Undirected Graphs

- In the previous graphs, edges were **undirected** (meaning edges are 'bidirectional' or 'reflexive')
  - An edge (u,v) implies (v,u)
- In **directed** graphs, links are unidirectional
  - An edge (u,v) does not imply (v,u)
  - For Edge (u,v): the **source** is u, **target** is v
- For adjacency list form, you may need 2 lists per vertex for both predecessors and successors



List of Vertices	Adjacency Lists
a	c,e
b	h
c	b,d,e,g
d	f
e	f
f	
g	f
h	g

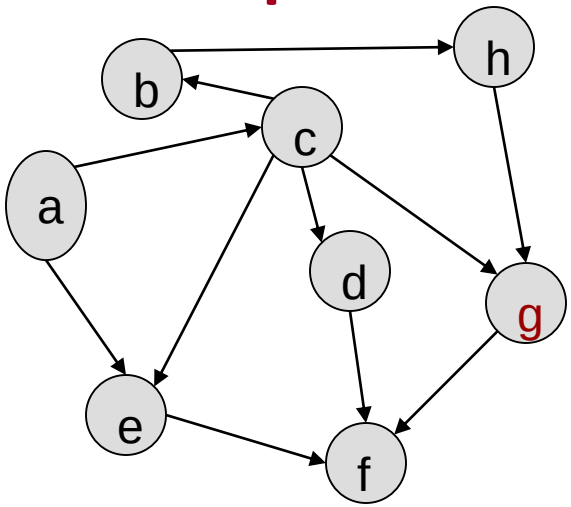
Target

Source	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	0	0	0	0	0	1
c	0	1	0	1	1	0	1	0
d	0	0	0	0	0	1	0	0
e	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	1	0	0
h	0	0	0	0	0	0	1	0

Adjacency Matrix Representation

Directed vs. Undirected Graphs

- In directed graph with edge (src,tgt) we define
  - Successor(src) = tgt
  - Predecessor(tgt) = src
- Using an adjacency list representation *may* warrant two lists predecessors and successors



List of Vertices

	Succs (Outgoing)	Preds (Incoming)
a	c,e	
b	h	c
c	b,d,e,g	a
d	f	c
e	f	a,c
f		d, e, g
g	f	c,h
h	g	b

Source \ Target

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	0	0	0	0	0	1
c	0	1	0	1	1	0	1	0
d	0	0	0	0	0	1	0	0
e	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	1	0	0
h	0	0	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Runtime, |V| = n, |E| =m

Operation vs Implementation for Edges	Add edge	Delete Edge	Test Edge	Enumerate edges for single vertex
Unsorted array or Linked List				
Sorted array				
Adjacency List				
Adjacency Matrix				

Graph Runtime, |V| = n, |E| =m

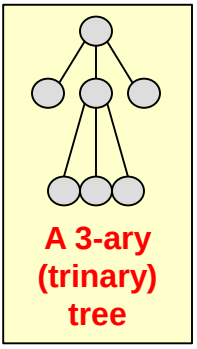
Operation vs Implementation for Edges	Add edge	Delete Edge	Test Edge	Enumerate edges for single vertex
Unsorted array or Linked List	Θ(1)	Θ(m)	Θ(m)	Θ(m)
Sorted array	Θ(m)	Θ(m)	Θ(log m) [if binary search used]	Θ(log m)+Θ(deg(v)) [if binary search used]
Adjacency List	Time to find List for a given vertex + Θ(1)	Time to find List for a given vertex + Θ(deg(v))	Time to find List for a given vertex + Θ(deg(v))	Time to find List for a given vertex + Θ(deg(v))
Adjacency Matrix	Θ(1)	Θ(1)	Θ(1)	Θ(v)

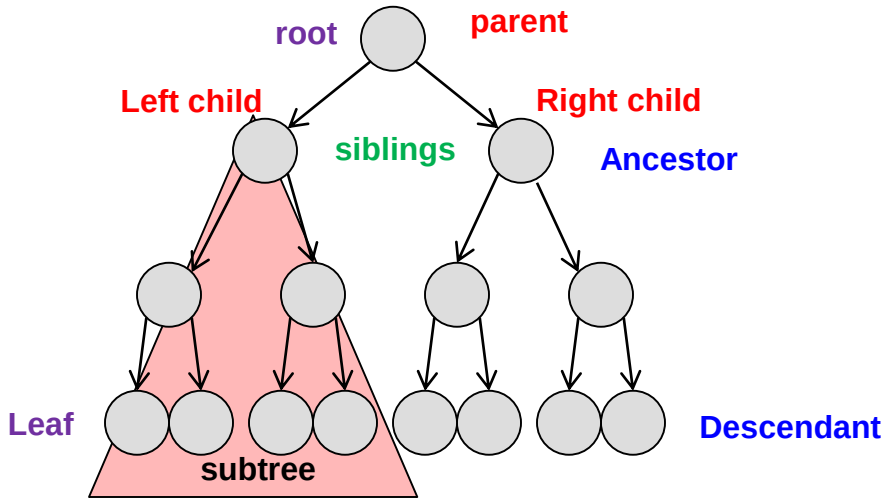
A graph with restrictions

TREES

Tree Definitions – Part 1

- **Definition**: A connected, acyclic (no cycles) graph with:
  - A root node, r, that has 0 or more subtrees
  - Exactly one path between any two nodes
- In general:
  - Nodes have exactly one parent (except for the root which has none) and 0 or more children
- d-ary tree
  - Tree where each node has at most d children
  - Binary tree = d-ary Tree with d=2



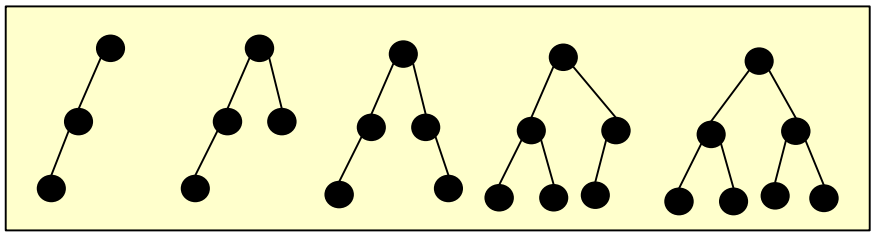


Terms:

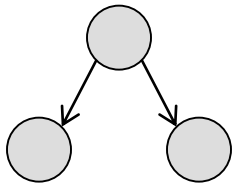
- **Parent(i)**: Node directly above node i
- **Child(i)**: Node directly below node i
- **Siblings**: Children of the same parent
- **Root:** Only node with no parent
- **Leaf:** Node with 0 children
- **Height:** Number of nodes on longest path from root to any leaf
- **Subtree(n)**: Tree rooted at node n
- **Ancestor(n)**: Any node on the path from n to the root
- **Descendant(n):** Any node in the subtree rooted at n

Tree Definitions – Part 2

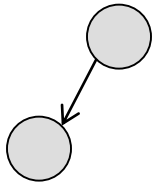
- Tree height: maximum # of nodes on a path from root to any leaf
- **Full** d-ary tree, T, where
  - Every vertex has 0 or d children and all leaf nodes are at the same level (i.e. adding 1 more node requires increasing the height of the tree)
- **Complete** d-ary tree
  - **Top h-1 levels are full** AND **bottom level is filled left-to-right**
  - Each level is filled left-to-right and a new level is not started until the previous one is complete
- **Balanced** d-ary tree
  - Tree where, for EVERY node, the subtrees for each child differ in height by at most 1



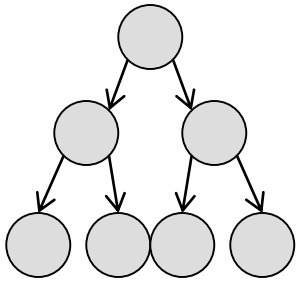
DAPS, 6th Ed. Figure 15-8



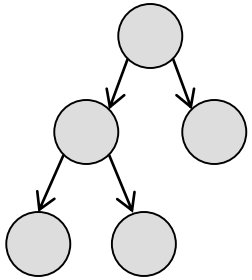
Full



Complete, but not full



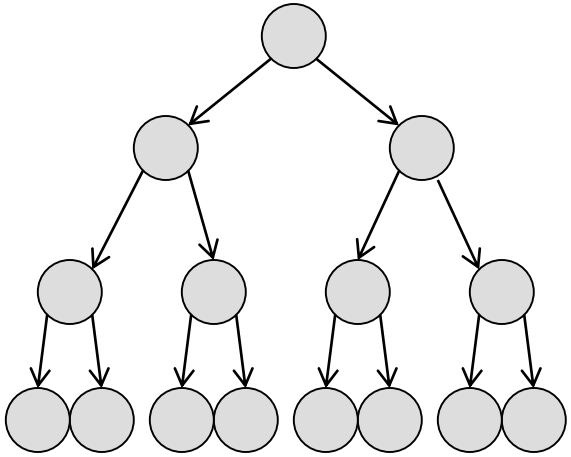
Full



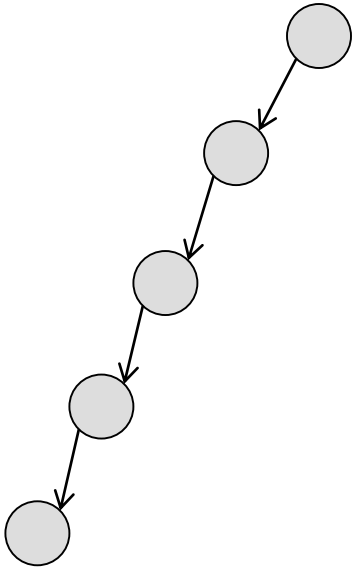
Complete, but not full

Tree Height

- A full or complete binary tree of n nodes has height, h=$\lceil log_2(n+1) \rceil$
  - This implies the minimum height of any tree with n nodes is $\lceil log_2(n+1) \rceil$
- The maximum height of a tree with n nodes is, ___



15 nodes => height $\log_2(16) = 4$



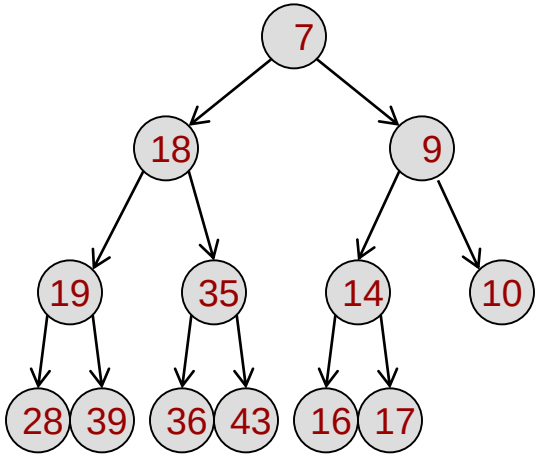
5 nodes => height = __

Array-based and Link-based

TREE IMPLEMENTATIONS

Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let's say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = _____
  - Left_child(i) = _____
  - Right_child(i) = _____



0	1	2	3	4	5	6	7	8	9	10	11	12	13
em	7	18	9	19	35	14	10	28	39	36	43	16	17

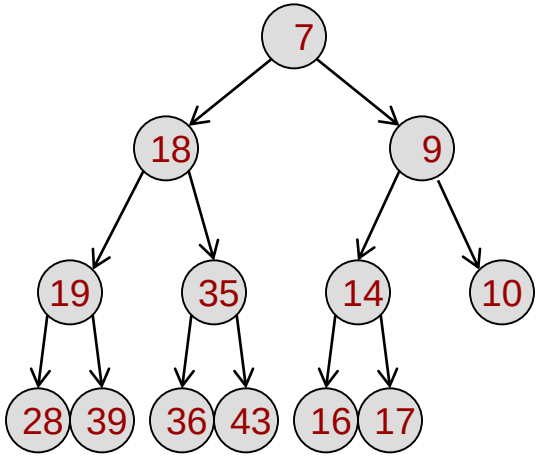
parent(5) = _____

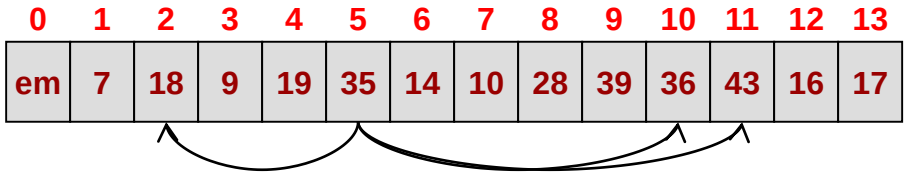
Left_child(5) = _____

Right_child(5) = _____

Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let's say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding node i's parent, left, and right child?
  - Parent(i) = i/2
  - Left_child(i) = 2*i
  - Right_child(i) = 2*i + 1



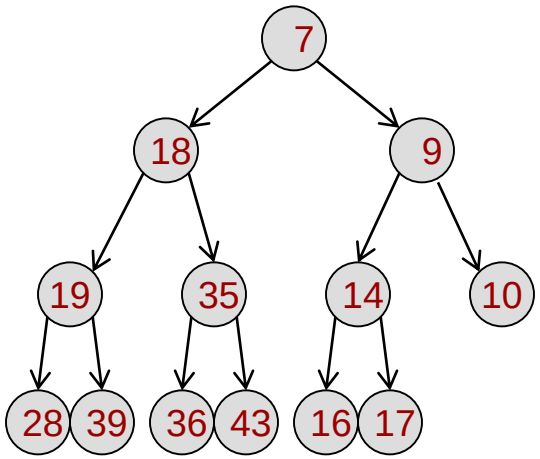


parent(5) = 5/2 = 2
Left_child(5) = 2*5 = 10
Right_child(5) = 2*5+1 = 11

Non-complete binary trees require much more bookeeping to store in arrays…usually link-based approaches are preferred

0-Based Indexing

- Now let's assume we start the root at index 0 of the array
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = _____
  - Left_child(i) = _____
  - Right_child(i) = _____

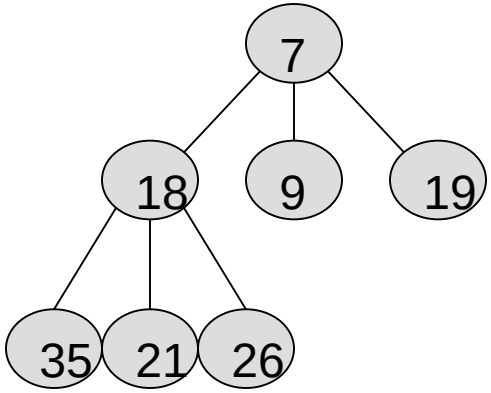


0	1	2	3	4	5	6	7	8	9	10	11	12
7	18	9	19	35	14	10	28	39	36	43	16	17

parent(5) = _____
Left_child(5) = _____
Right_child(5) = _____

D-ary Array-based Implementations

- Arrays can be used to store d-ary **complete** trees
  - Adjust the formulas derived for binary trees in previous slides in terms of **d**



A 3-ary (trinary) tree

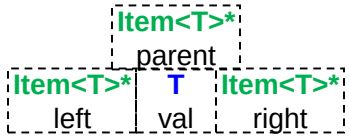
0	1	2	3	4	5	6
7	18	9	19	35	21	26

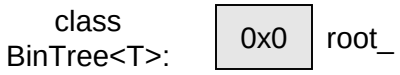
Link-Based Approaches

- For an arbitrary (**non-complete**) d-ary tree we need to use pointer-based structures
  - Much like a linked list but now with two pointers per Item
- Use NULL pointers to indicate no child
- Dynamically allocate and free items when you add/remove them

```
#include<iostream>
using namespace std;

template <typename T>
struct Item {
  T val;
  Item<T>* left,right;
  Item<T>* parent;
};
// Bin. Search Tree
template <typename T>
class BinTree
{
 public:
 BinTree();
 ~BinTree();
 void add(const T& v);
 ...
 private:
 Item<T>* root_;
};
```

Item<T> blueprint:





Link-Based Approaches

- Add(5)
- Add(6)
- Add(7)