Introduction to Algorithms
Data Structures

CSE 680
Prof. Roger Crawfis

Overview

- Review basic abstract data structures
  - Sets
  - Lists
  - Trees
  - Graphs
- Review basic concrete data structures
  - Linked-List and variants
  - Trees and variants
- Examine key properties
- Discuss usage for solving important problems (search, sort, selection).

Sets and Multisets

- Common operations
  - Fixed Sets
    - Contains (search)
    - Is empty
    - Size
    - Enumerate
  - Dynamic Sets add:
    - Add
    - Remove
- Other operations (not so common)
  - Intersection
  - Union
  - Sub-set
  - Note, these can, and probably should, be implemented statically (outside of the class).

Set – Language Support

- .NET **Framework** Support (C#,VB,C++,…)
  - IEnumerable interface
  - ICollection interface
- Java **Framework** Support
  - Collection
  - Set
- STD library (C++)
  - Set and Multiset classes and their iterators.

List

- Common Queries
  - Enumerate
  - Number of items in the list
  - Return element at index *i*.
  - Search for an item in the list (contains)
- Common Commands
  - Add element
  - Set element at index *i*.
  - Remove element?
  - Insert before index *i*?

List – Language Support

- Arrays – fixed size.
- .NET **Framework** Support (C#,VB,C++,…)
  - IList interface
  - List<T> class
- Java **Framework** Support
  - List interface
  - ArrayList<T> and Vector<T> classes
- STD library (C++)
  - std::vector<T> class.

Concrete Implementations

- Set
  - What might you use to implement a concrete set?
  - What are the pro's and con's of each approach?
- List
  - Other than arrays, could you implement a list with any other data structure?

Rooted Trees

- A ***tree*** is a collection of *nodes* and *directed edges*, satisfying the following properties:
  - There is one specially designated node called the ***root***, which has no edges pointing to it.
  - Every node except the *root* has exactly one edge pointing to it.
  - There is a **unique** path (of nodes and edges) from the *root* to each node.

Basic Tree Concepts

- Node – user-defined data structure that that contains pointers to data and pointers to other nodes:
  - **Root** – Node from which all other nodes descend
  - **Parent** – has child nodes arranged in subtrees.
  - **Child** – nodes in a tree have 0 or more children.
  - **Leaf** – node without descendants
  - **Degree** – number of direct children a tree/subtree has.

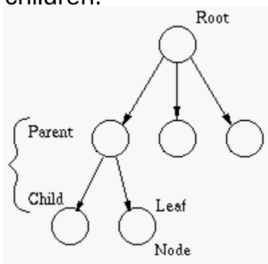
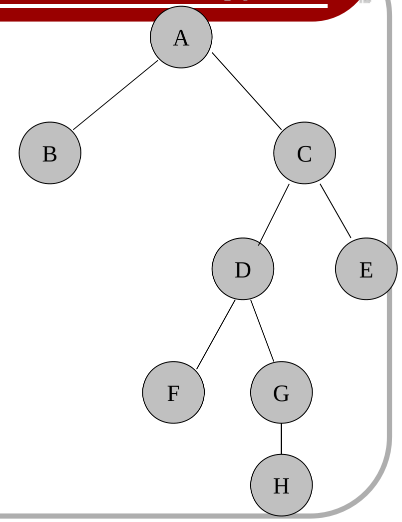


Figure: Tree data structure

Height and Level of a Tree

- **Height** – # of edges on the *longest* path from the root to a leaf.
- **Level** – Root is at level 0, its direct children are at level 1, etc.
- Recursive definition for height:

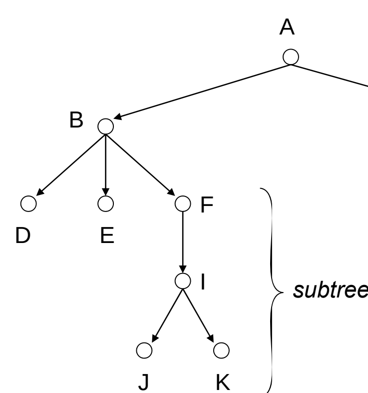
1+ max(height(T_L), height(T_R))



Rooted Trees

- If an edge goes from node *a* to node *b*, then *a* is called the ***parent*** of *b*, and *b* is called a ***child*** of *a*.
- Children of the same parent are called ***siblings***.
- If there is a path from *a* to *b*, then *a* is called an ***ancestor*** of *b*, and *b* is called a ***descendent*** of *a*.
- A node with all of its descendants is called a ***subtree***.
- If a node has no children, then it is called a ***leaf*** of the tree.
- If a node has no parent (there will be exactly one of these), then it is the ***root*** of the tree.

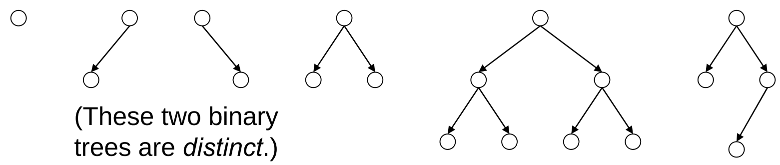
Rooted Trees: *Example*



- A is the *root*
- D, E, G, H, J & K are *leaves*
- B is the *parent* of D, E & F
- D, E & F are *siblings* and *children* of B
- I, J & K are *descendants* of B
- A & B are *ancestors* of I

Binary Trees

- Intuitively, a ***binary tree*** is a *tree* in which each node has no more than two children.

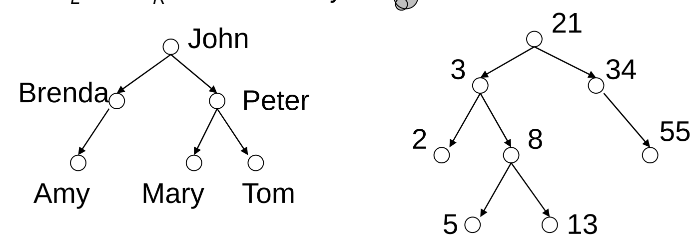


(These two binary trees are *distinct*.)

Binary Search Trees

- A ***binary search tree*** is ... each node, *n*, has a *value* s... properties:
  - *n*'s value is > all valu...
  - *n*'s value is < all values in ... T_R, and
  - T_L and T_R are both *binary sea... trees*.

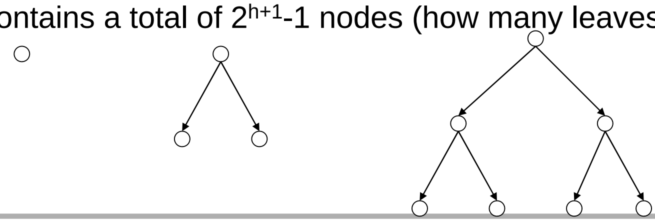
In other words, can we put non-hierarchical data into a tree. We will study Binary Search Trees later.



Binary Trees

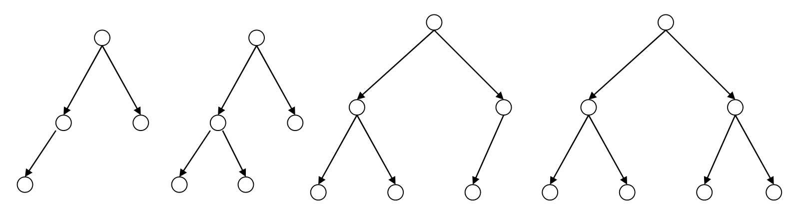
This term is ambiguous, some indicate that each node is either full or empty.

- A binary tree is *full* if it has no missing nodes.
  - It is either empty.
  - Otherwise, the root's subtrees are *full* binary trees of height h – 1.
- If not empty, each node has 2 children, except the nodes at level *h* which have no children.
- Contains a total of 2^{h+1}-1 nodes (how many leaves?)



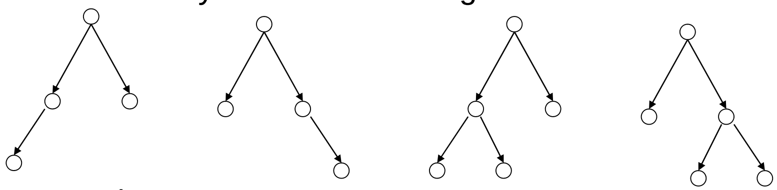
Binary Trees

- A binary tree of height *h* is ***complete*** if it is *full* down to level *h – 1*, and level *h* is filled from left to right.
  - All nodes at level h – 2 and above have 2 children each,
  - If a node at level h – 1 has children, all nodes to its left at the same level have 2 children each, and
  - If a node at level h – 1 has 1 child, it is a left child.



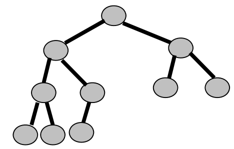
Binary Trees

- A binary tree is ***balanced*** if the difference in height between any node's left and right subtree is ≤ 1.

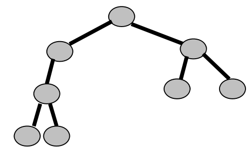


- Note that:
  - A full binary tree is also complete.
  - A complete binary tree is not always full.
  - Full and complete binary trees are also balanced.
  - Balanced binary trees are not always full or complete.

Complete & Balanced Trees



Complete and Balanced



Not Balanced, Why?

Binary Tree: *Pointer-Based Representation*

```
struct  TreeNode;              // Binary Tree nodes are struct's
typedef  string  TreeItemType; // items in TreeNodes are string's
class  BinaryTree
{
private:
    TreeNode  *root;           // pointer to root of Binary Tree
};
struct  TreeNode               // node in a Binary Tree:
{                              //  place in Implementation file
  TreeItemType  item;
  TreeNode  *leftChild;              // pointer to TreeNode's left
   child
  TreeNode  *rightChild;       // pointer to TreeNode's right child
};
```

Binary Tree: *Table-Based Representation*

Basic Idea:

- Instead of using *pointers* to the left and right child of a node, use *indices* into an array of nodes representing the binary tree.
- Also, use variable *free* as an index to the first position in the array that is available for a new entry. Use either the *left* or *right child* indices to indicate additional, available positions.
- Together, the list of available positions in the array is called the ***free list***.

Binary Tree: *Table-Based Representation*

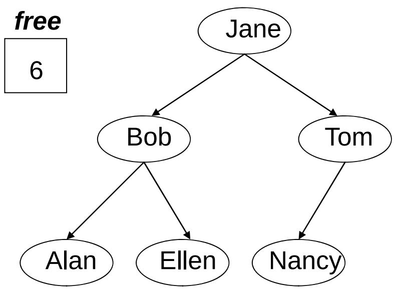
Index	Item	Left Child	Right Child
0	Jane	1	2
1	Bob	3	4
2	Tom	5	-1
3	Alan	-1	-1
4	Ellen	-1	-1
5	Nancy	-1	-1
6	?	-1	7
7	?	-1	8
8	?	-1	9
9	. . .	. . .	. . .

root

0

free

6



Binary Tree: *Table-Based Representation*

Index	Item	Left Child	Right Child
0	Jane	1	2
1	Bob	3	4
2	Tom	5	-1
3	Alan	-1	-1
4	Ellen	-1	-1
5	Nancy	**6**	-1
6	**Mary**	-1	**-1**
7	?	-1	8
8	?	-1	9
9	. . .	. . .	. . .

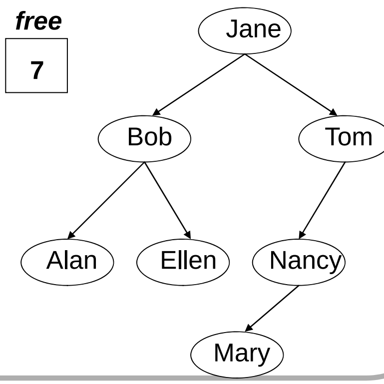
root

0

free

7

** Mary Added under Nancy.*



Binary Tree: *Table-Based Representation*

Index	Item	Left Child	Right Child
0	Jane	1	2
1	Bob	3	**-1**
2	Tom	5	-1
3	Alan	-1	-1
4	**?**	-1	**7**
5	Nancy	6	-1
6	Mary	-1	-1
7	?	-1	8
8	?	-1	9
9	. . .	. . .	. . .

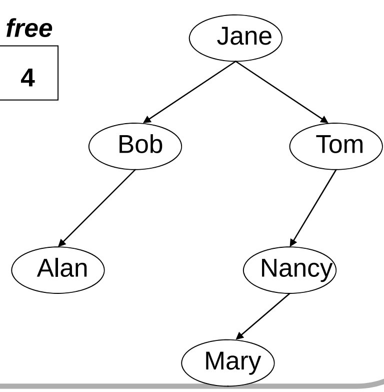
root

0

free

4

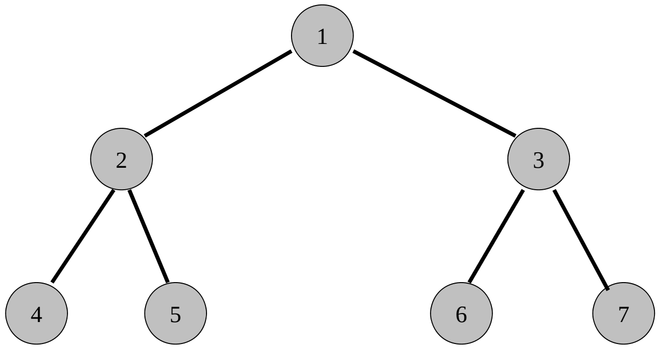
** Ellen deleted.*



Binary Tree: *Table-Based Representation*

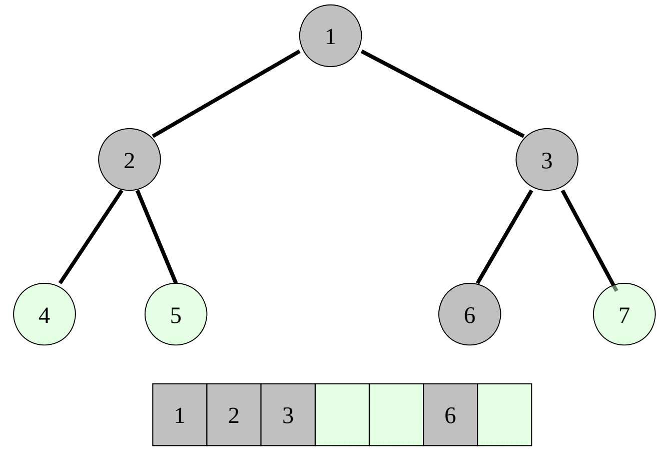
```
const int MaxNodes = 100;      // maximum size of a Binary Tree
typedef string TreeItemType;   // items in TreeNodes are string's
struct TreeNode                // node in a Binary Tree
{
   TreeItemType item;
   int leftChild;              // index of TreeNode's left child
   int rightChild;             // index of TreeNode's right child
};
class BinaryTree
{
private:
      TreeNode node[MaxNodes];
      int root;             // index of root of Binary Tree
      int free;             // index of free list, linked by rightChild
};
```

Level Ordering

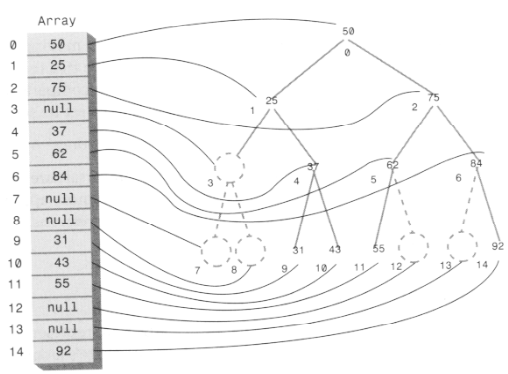


Let i, $1 \le i \le n$, be the number assigned to an element of a complete binary tree.

Array-Based Representation



Array-Based Representation

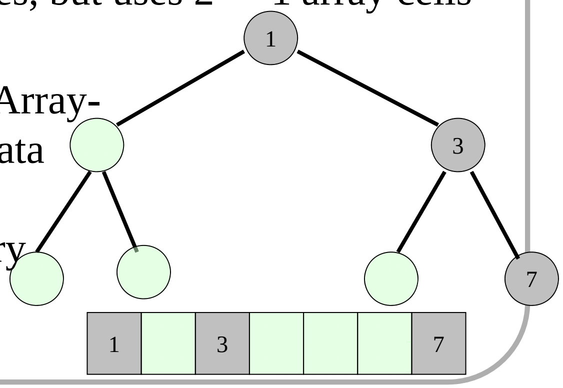


Array-Based Representation

- Array-based representations allow for efficient traversal. Consider the node at index *i.*
  - Left Child is at index *2i+1.*
  - Right Child is at index *2i+2.*
  - Parent is at ***floor****((i-1)/2).*

Array-Based Representation

- **Drawback** of array-based trees: Example has only 3 nodes, but uses $2^{h+1}-1$ array cells to store it
- Generally use Array-based only if data set exhibits **complete** binary tree behavior



Traversing a Binary Tree

- Depth-first Traversal
  - Preorder
  - Inorder
  - Postorder
- Breadth-First Traversal
  - Level order

Preorder Traversal

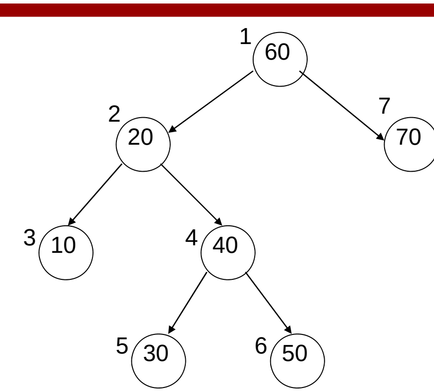
Basic Idea:

1) ***Visit*** the *root*.

2) Recursively invoke ***preorder*** on the *left subtree*.

3) Recursively invoke ***preorder*** on the *right subtree*.

Preorder Traversal



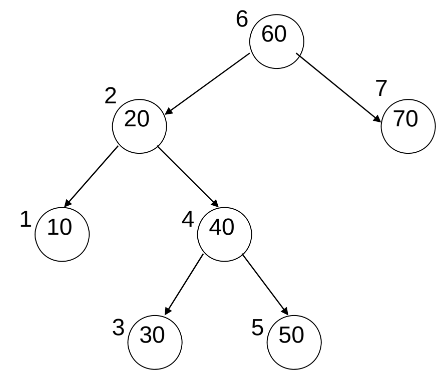
Preorder Result: 60, 20, 10, 40, 30, 50, 70

Inorder Traversal

Basic Idea:

1) Recursively invoke ***inorder*** on the *left subtree*.
2) ***Visit*** the *root*.
3) Recursively invoke ***inorder*** on the *right subtree*.

Inorder Traversal



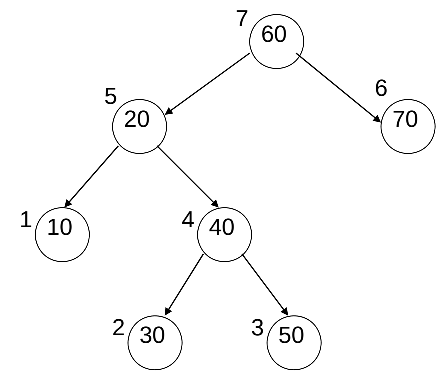
Inorder Result: 10, 20, 30, 40, 50, 60, 70

Postorder Traversal

Basic Idea:

1) Recursively invoke ***postorder*** on the *left subtree*.
2) Recursively invoke ***postorder*** on the *right subtree*.
3) ***Visit*** the *root*.

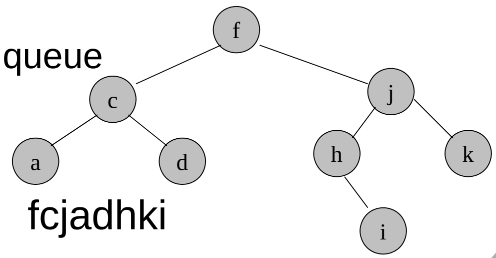
Postorder Traversal



Postorder Result: 10, 30, 50, 40, 20, 70, 60

Level order traversal

- Visit the tree in left-to-right, by level, order:
  - Visit the root node and put its children in a queue (left to right).
  - Dequeue, visit, and put dequeued node's children into the queue.
- Repeat until the queue is empty.



fcjadhki

Pointer-Based, *Preorder* Traversal in C++

```
// FunctionType is a pointer to a function with argument
//  (TreeItemType &) that returns void.
   typedef  void  (*FunctionType) (TreeItemType  &treeItem);

// Public member function
void  BinaryTree::preorderTraverse( FunctionType  visit )
{
     preorder( root,  visit );
}
```

Pointer-Based, *Preorder* Traversal in C++

```
// Private member function
void  BinaryTree::preorder( TreeNode  *treePtr,
    FunctionType  visit )
{
   if( treePtr != NULL )
   {
      visit( treePtr -> item );
      preorder( treePtr -> leftChild,  visit );
      preorder( treePtr -> rightChild,  visit );
   }
}
```

Pointer-Based, *Preorder* Traversal in C++

Suppose that we define the function

```
   void  printItem( TreeItemType  &treeItem )
   {  cout << treeItem << endl;
   }
```

Then,

```
   // create myTree
      BinaryTree   myTree;
   // load data into myTree
      . . .
   // print TreeItems encountered in preorder traversal
   of myTree
      myTree.preorderTraverse( &printItem );
```

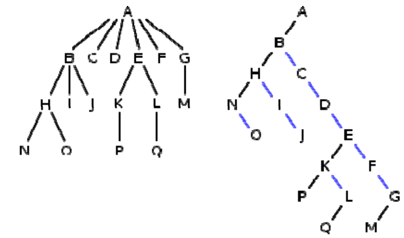
Nonrecursive Traversal of a Binary Tree

Basic Idea for a Nonrecursive, *Inorder* Traversal:

1) Push a pointer to the *root* of the binary tree onto a stack.
2) Follow *leftChild* pointers, pushing each one onto the stack, until a *NULL leftChild* pointer is found.
3) Process (visit) the item in this node.
4) Get the node's *rightChild* pointer:
   - If it is **not** *NULL*, then push it onto the stack, and return to step 2 with the *leftChild* pointer of this *rightChild*.
   - If it is *NULL*, then pop a node pointer from the stack, and return to step 3. If the stack is empty (so nothing could be popped), then stop — the traversal is done.

N-ary Trees

- We can encode an n-ary tree as a binary tree, by have a list of linked-list of children. Hence still two pointers, one to the first child and one to the next sibling.
- Kinda rotates the tree.



Other Binary Tree Properties

- The number of edges in a tree is *n-1.*
- The number of nodes *n* in a full binary tree is: $n = 2^{h+1} - 1$ where *h* is the height of the tree.
- The number of nodes *n* in a complete binary tree is:
  - minimum: $n = 2^h$
  - maximum: $n = 2^{h+1} - 1$ where *h* is the height of the tree.
- The number of nodes *n* in a full or perfect binary tree is:
  - $n = 2L - 1$ where *L* is the number of leaf nodes in the tree.
- The number of leaf nodes *n* in a full or perfect binary tree is:
  - $n = 2^h$ where *h* is the height of the tree.
- The number of leaf nodes in a Complete Binary Tree with *n* nodes is *UpperBound*(*n* / 2).
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.

Graphs

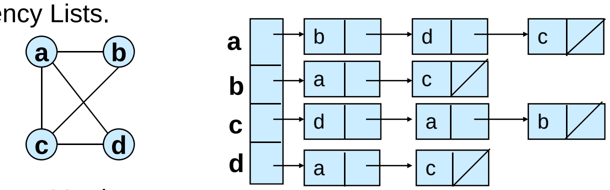
- *Graph G = (V, E)*
  - *V* = set of vertices
  - *E* = set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbf{R}$.
  - Dense: $|E| \approx |V|^2$.
  - Sparse: $|E| << |V|^2$.
- $|E| = O(|V|^2)$

Graphs

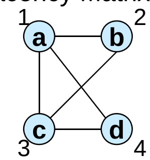
- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - Not necessarily so if G is directed.
- If G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a tree.
- Other definitions in Appendix B (B.4 and B.5) as needed.

Representation of Graphs

- Two standard ways.
  - Adjacency Lists.



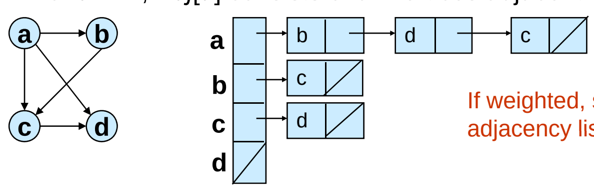
  - Adjacency Matrix.



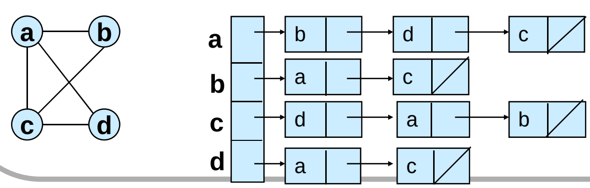
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Adjacency Lists

- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u.



If weighted, store weights also in adjacency lists.



Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is

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

    No. of edges leaving v
  - Total storage: $\Theta(|V| + |E|)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is

    $\sum_{v \in V} \text{degree}(v) = 2|E|$

    No. of edges incident on v. Edge (u,v) is incident on vertices u and v.
  - Total storage: $\Theta(|V| + |E|)$

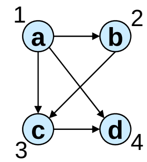
Pros and Cons: adj list

- Pros
  - Space-efficient, when a graph is sparse.
  - Can be modified to support many graph variants.
- Cons
  - Determining if an edge $(u, v) \in G$ is not efficient.
    - Have to search in u's adjacency list. $\Theta(\text{degree}(u))$ time.
    - $\Theta(V)$ in the worst case.

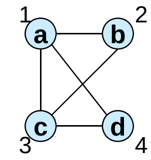
Adjacency Matrix

- $|V| \times |V|$ matrix A.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- A is then given by:

$$A[i,j] = a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$



	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^T$ for undirected graphs.

Space and Time

- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph.

Some graph operations

	adjacency matrix	adjacency lists
insertEdge	O(1)	O(e)
isEdge	O(1)	O(e)
#successors?	O(V)	O(e)
#predecessors?	O(V)	O(E)

C# Interfaces

```csharp
using System;
using System.Collections.Generic;
using System.Security.Permissions;
[assembly: CLSCompliant(true)]
namespace OhioState.Collections.Graph {
    /// <summary>
    /// IEdge provides a standard interface to specify an edge and any
    /// data associated with an edge within a graph.
    /// </summary>
    /// <typeparam name="N">The type of the nodes in the
       graph.</typeparam>
    /// <typeparam name="E">The type of the data on an
       edge.</typeparam>
    public interface IEdge<N,E>    {
        /// <summary>
        /// Get the Node label that this edge emanates from.
        /// </summary>
        N From { get; }
        /// <summary>
        /// Get the Node label that this edge terminates at.
        /// </summary>
        N To { get; }
        /// <summary>
        /// Get the edge label for this edge.
        /// </summary>
        E Value { get; }
    }
```

```csharp
/// <summary>
/// The Graph interface
/// </summary>
/// <typeparam name="N">The type associated at each node.
   Called a node or node label</typeparam>
/// <typeparam name="E">The type associated at each edge. Also
   called the edge label.</typeparam>
public interface IGraph<N,E>    {
    /// <summary>
    /// Iterator for the nodes in the graoh.
    /// </summary>
    IEnumerable<N> Nodes { get; }
    /// <summary>
    /// Iterator for the children or neighbors of the specified node.
    /// </summary>
    /// <param name="node">The node.</param>
    /// <returns>An enumerator of nodes.</returns>
    IEnumerable<N> Neighbors(N node);
    /// <summary>
    /// Iterator over the parents or immediate ancestors of a node.
    /// </summary>
    /// <remarks>May not be supported by all graphs.</remarks>
    /// <param name="node">The node.</param>
    /// <returns>An enumerator of nodes.</returns>
    IEnumerable<N> Parents(N node);
```

C# Interfaces

```csharp
/// <summary>
/// Iterator over the emanating edges from a node.
/// </summary>
/// <param name="node">The node.</param>
/// <returns>An enumerator of nodes.</returns>
IEnumerable<IEdge<N, E>> OutEdges(N node);
/// <summary>
/// Iterator over the in-coming edges of a node.
/// </summary>
/// <remarks>May not be supported by all graphs.</remarks>
/// <param name="node">The node.</param>
/// <returns>An enumerator of edges.</returns>
IEnumerable<IEdge<N, E>> InEdges(N node);
/// <summary>
/// Iterator for the edges in the graph, yielding IEdge's
/// </summary>
IEnumerable<IEdge<N, E>> Edges { get; }
/// <summary>
/// Tests whether an edge exists between two nodes.
/// </summary>
/// <param name="fromNode">The node that the edge
 emanates from.</param>
/// <param name="toNode">The node that the edge terminates
 at.</param>
/// <returns>True if the edge exists in the graph. False
 otherwise.</returns>
bool ContainsEdge(N fromNode, N toNode);
/// <summary>
/// Gets the label on an edge.
```

```csharp
/// </summary>
/// <param name="fromNode">The node that the edge
 emanates from.</param>
/// <param name="toNode">The node that the edge terminates
 at.</param>
/// <returns>The edge.</returns>
E GetEdgeLabel(N fromNode, N toNode);
/// <summary>
/// Exception safe routine to get the label on an edge.
/// </summary>
/// <param name="fromNode">The node that the edge
 emanates from.</param>
/// <param name="toNode">The node that the edge terminates
 at.</param>
/// <param name="edge">The resulting edge if the method was
 successful. A default
/// value for the type if the edge could not be found.</param>
/// <returns>True if the edge was found. False
 otherwise.</returns>
bool TryGetEdge(N fromNode, N toNode, out E edge);
    }
}
```

C# Interfaces

```csharp
using System;
  namespace OhioState.Collections.Graph {
    /// <summary>
    /// Graph interface for graphs with finite size.
    /// </summary>
    /// <typeparam name="N">The type associated at each node. Called a node or node
       label</typeparam>
    /// <typeparam name="E">The type associated at each edge. Also called the edge label.</typeparam>
    /// <seealso cref="IGraph{N, E}"/>
    public interface IFiniteGraph<N, E> : IGraph<N, E>    {
        /// <summary>
        /// Get the number of edges in the graph.
        /// </summary>
        int NumberOfEdges { get; }
        /// <summary>
        /// Get the number of nodes in the graph.
        /// </summary>
        int NumberOfNodes { get; }
    }
}
```