Fundamental Data Structures

(Chapter 1.4)

Data Structures

- Often … the way you organize data affects the performance of your algorithm

- A *data structure* is a particular way of storing and organizing data
  - Part of algorithm design is choosing the right data structure

Fundamental Data Structures

- Linear Data Structures
  - Array
  - Linked list
  - Stack
  - Queue
- Set
- Dictionary (Map)
- Tree
- Graph

Kahoot!

Arrays

- A sequence of *n* items of the same type, accessed by an index

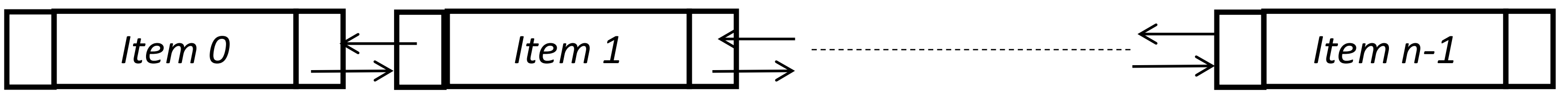
Item[0]	*Item[1]*	…	*Item[n-1]*

- The good:
  - Each item accessed in same constant time
- The bad:
  - Size is fixed
  - Insertion / deletion in an array is time consuming – all the elements following the inserted element must be shifted appropriately

Linked Lists

- (singly) A sequence of zero or more elements called *nodes,* consisting of data and a pointer

- (doubly) Pointers in each direction



Linked Lists

- Linked lists provide two key advantages over arrays
  - Dynamic size
  - Ease of insertion/deletion

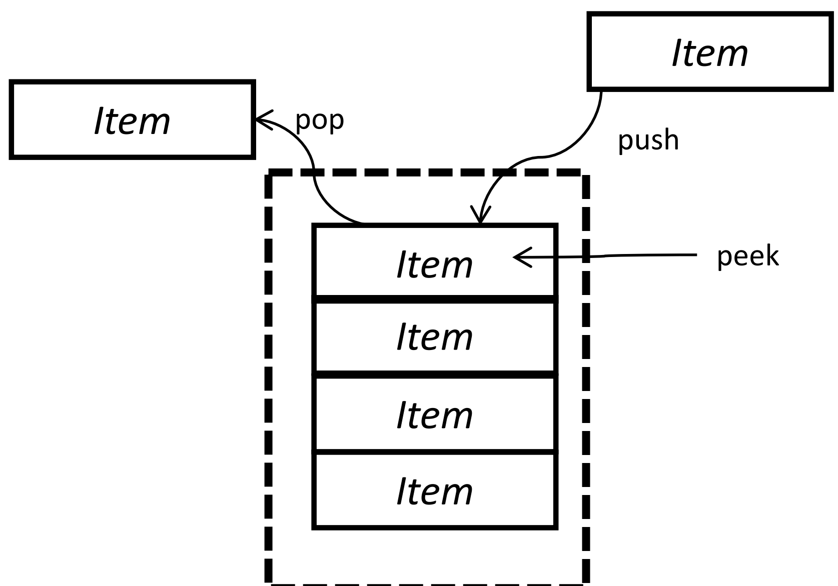
- Linked lists have some drawbacks:
  - Random access is not allowed

Linked Lists in java

```java
import java.util.*;

public class LinkedListDemo {

   public static void main(String args[]) {
      // create a linked list
      LinkedList ll = new LinkedList();

      // add elements to the linked list
      ll.add("A");
      ll.add("B");
      ll.add("C");
      ll.addLast("Z");
      ll.addFirst("s");
      ll.add(1, "k");


      // remove elements from the linked list
      ll.remove(2);

   }
}
```

Stack

- Like a stack of plates
- Last-in-first-out (LIFO)



Operations on a stack

- Insert operation is called <u>Push</u>
- Delete operation is called <u>Pop</u>
- Examining the top item is <u>Peek</u>

- Example application:
  - Analysis of languages (e.g. properly nested brackets)
  - Properly nested: (())
  - Wrongly nested: (()

Stack

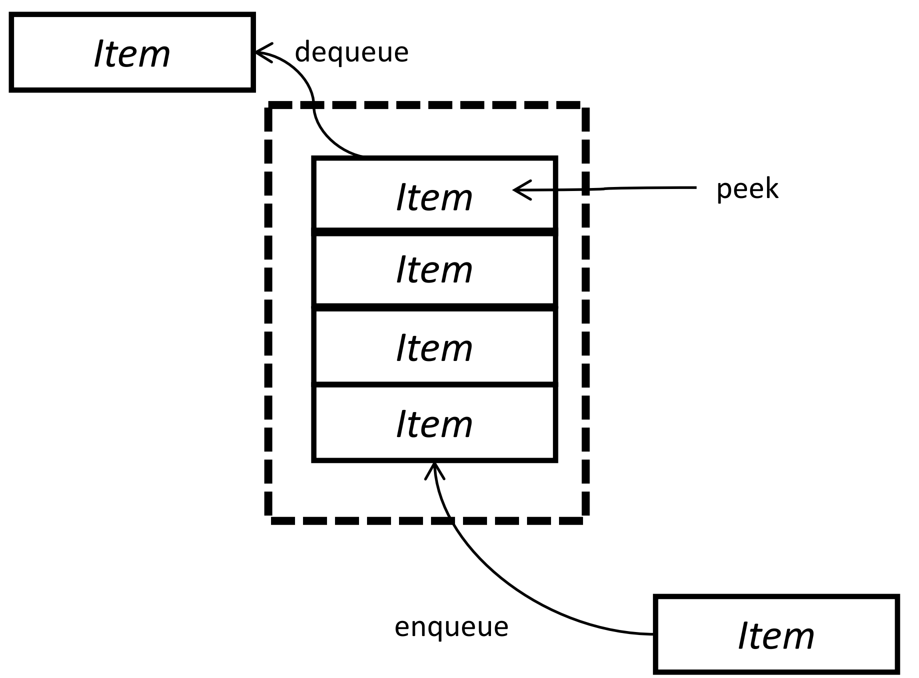
```
CheckBalancedParenthesis(expr)
n ← length(expr)
Create a stack s
for i ← 0 to n-1 do
   if (expr[i] is '(' ) do
      s.Push(expr[i])
   else if (expr[i] is ')' )
      if (s is empty) or
            (s.Peek() does not pair with expr[i])
         return False
      else
         s.Pop()
if (s is empty)
   return True
else
   return False
```

Abstract Data Type

- Often a data structure is closely associated with a set of available operations
- Data structure + operations = *abstract data type*
  - From an OOP perspective, think about members (methods) of a class
- Example from before: priority queue
  - Underlying implementation was a heap
  - Operations were Insert and deleteMax
- Example: operations on stacks:
  - Push, pop, peek

Queues

- Like a line-up
- First-in-first-out (FIFO)



Operations on a queue

- Adding to the queue is Enqueue
- Removing from the queue is Dequeue
- The top/front element is the Head (sometimes there is a "Peek" method)

Set

- A Set is just like a set in math, i.e: set = { 1, 2, 3, 4 }
- The key thing to remember:
  - ***Sets cannot contain duplicate items***
- Operations on a Set:
  - **Add things into it**
  - **Take things out of it**
  - **Check if it contains something**
  - **Iterate over the Set (examine each item, one-by-one)**

Set in Java

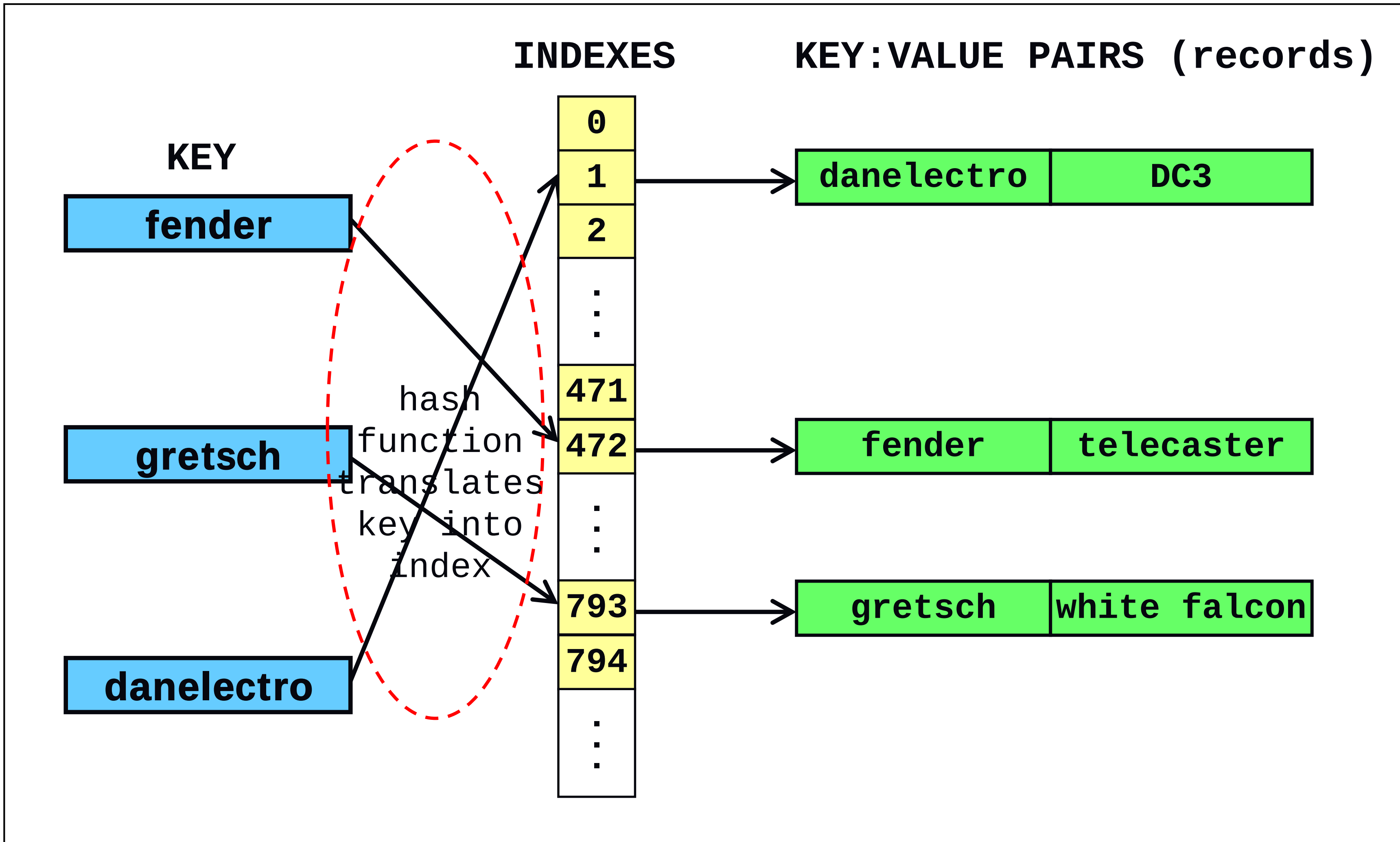
- There are a few different ways to implement Set

  - HashSet:
    - *HashSet* is the fastest implementation, *but it is unordered*

  - TreeSet
    - *TreeSet* is slower, *but maintains a sorted order*

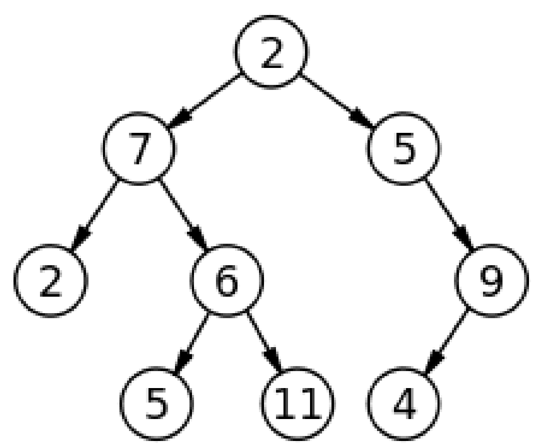
Map (as a hash table)

- A ***Map*** is a lookup table that takes a ***key*** and returns a ***value***
  - the most common implementation is as a hashtable (hashmap)

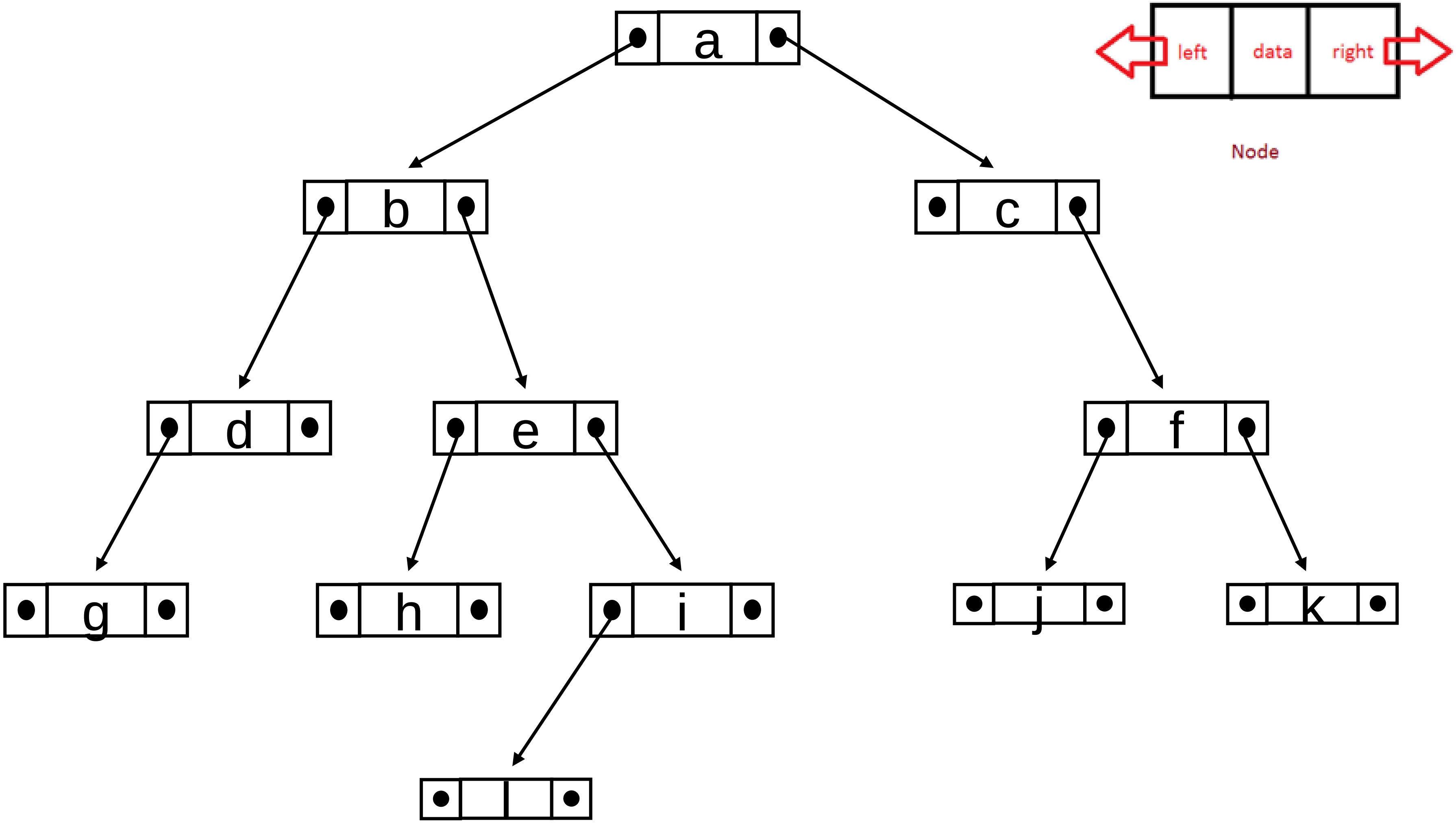


Trees

- A connected, acyclic graph
  - Usually we think of trees as having a *root*
- Representing data in a tree can speed up your algorithms in many natural problems

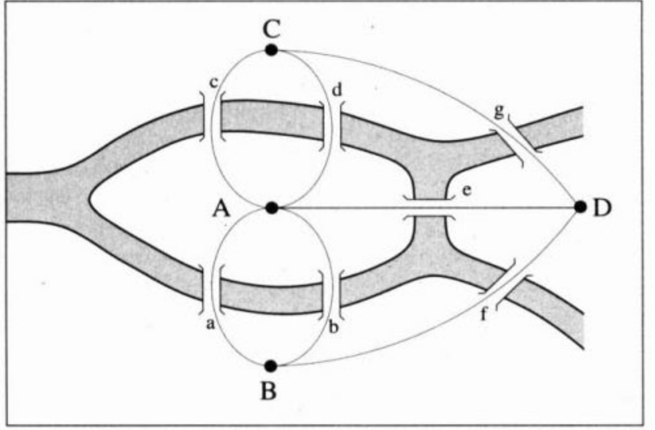


Binary tree implementation

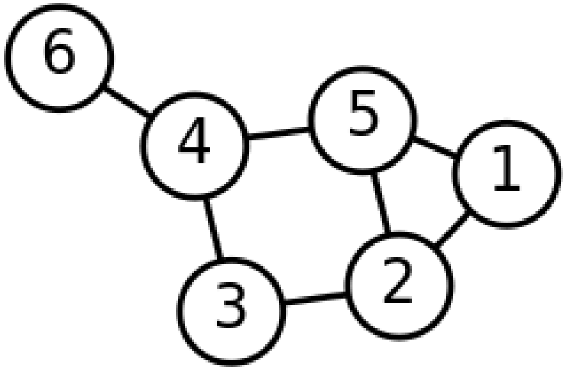


Graphs

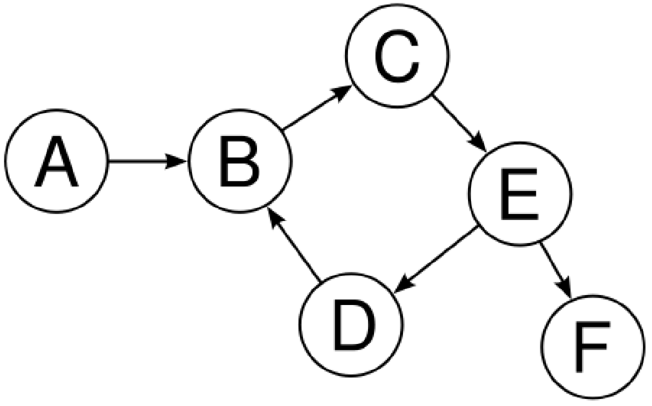
- G = (V, E)
  - V is a set of *vertices*
  - E is a set of *edges*



Motivation: Real world connections



Undirected



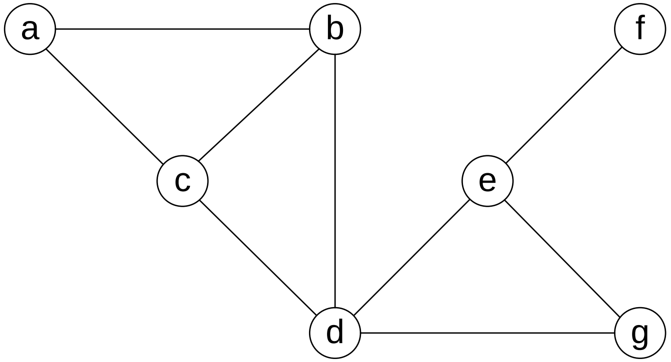
Directed

Representing Graphs

1. Adjacency matrix
2. Adjacency lists

Representation: Adjacency Matrix

- For this graph:

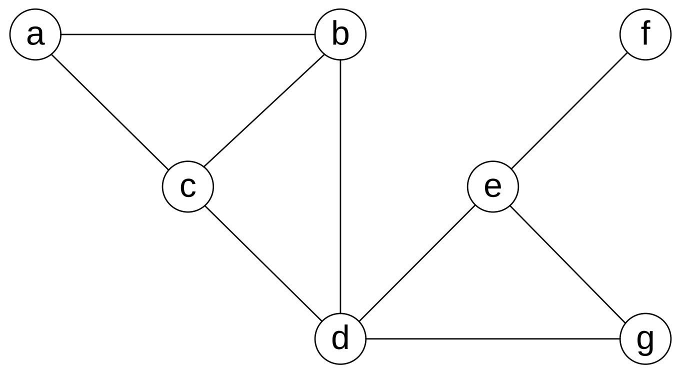


- Adjacency matrix is the following:

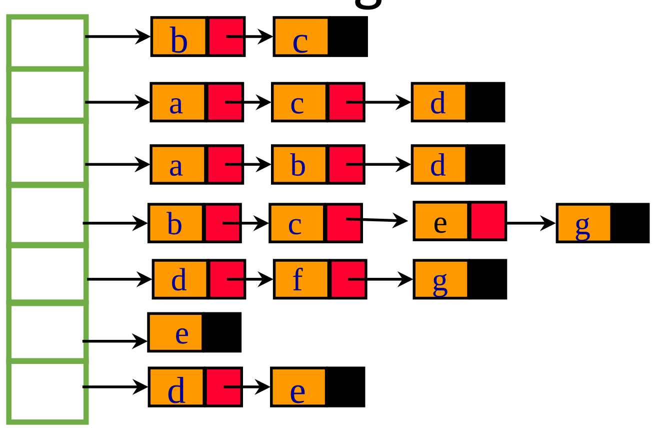
	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	1	0	0	0
d	0	1	1	0	1	0	1
e	0	0	0	1	0	1	1
f	0	0	0	0	1	0	0
g	0	0	0	1	1	0	0

Representation: Adjacency List

- For the same graph:



- Adjacency list is the following:
  - *For vertex a*:
  - *For vertex b*:
  - *For vertex c*:
  - *For vertex d*:
  - *For vertex e*:
  - *For vertex f*:
  - *For vertex g*:



Representing Graphs

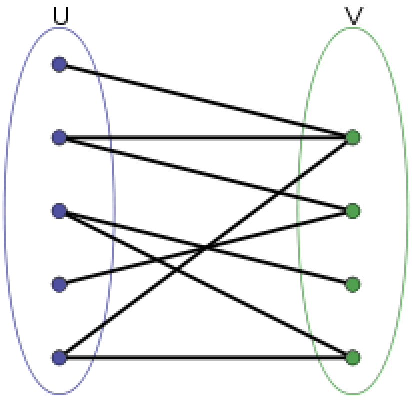
1. Adjacency matrix
   - Or Weight Matrix for weighted graphs

2. Adjacency lists
   - A list of vertices connected to each vertex

- Which one to use?
  - Depends on the nature of the graph (sparse or not)
  - Depends on the algorithm

Some special graphs

- *Connected graph*
  - A graph where there is a path connecting every two vertices
- *Bipartite graph*
  - Vertices can be partitioned into two separate sets u and v, so that all edges go from set u to set v



Some special graphs

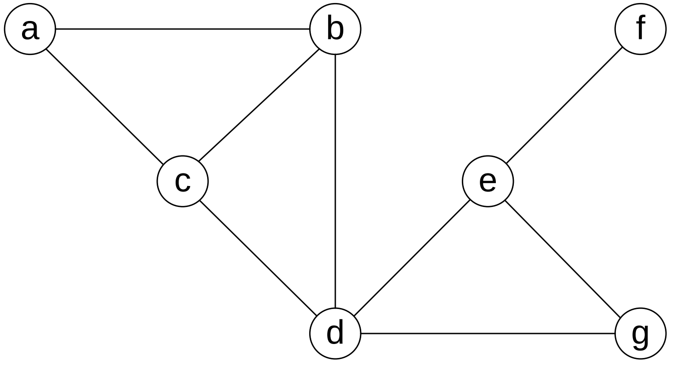
- *Cyclic graph*
  - A graph containing at least one cycle
- *Acyclic graph*
  - A graph containing no cycles
- *Tree*
  - *Any connected + acyclic graph*

Graph Algorithms

(Chapter 3.5)

Graph Traversal

- Many real-world problems require processing of each vertex (or edge) in a graph
  - e.g. Routing a message on a network



Graph Traversal Algorithms

- Graph traversal algorithms give a method for *systematically processing* all vertices

Idea: "visit" all the vertices, one at a time,
marking them as we visit them

- Two approaches:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

Depth-First Search (DFS)

- Visits all vertices by always *moving away* from the last vertex visited (if possible)
  - Backtracks if there are no more adjacent vertices
- Implementation often uses a stack of vertices being processed
- Follows a tree-like route throughout the graph

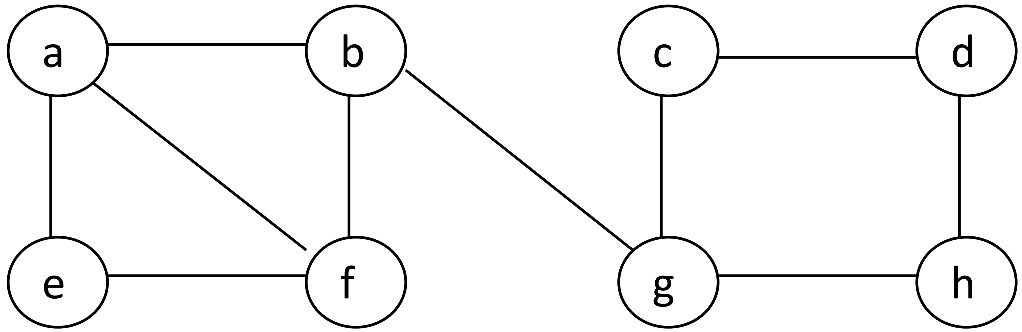
DFS

Algorithm:

```
Depth First Search(Graph G):
  initialize all visited values to false
  for each vertex v in V    // where G = {V,E}
     if v has not been visited
        dfs(Vertex v)
dfs(Vertex v)
  visit node v
  for each vertex w in V adjacent to v
     if w has not been visited
        dfs(w)
```

- “Visit node v” means doing whatever you need to do at each node
- The output is typically a “***DFS Tree***”, which is a tree containing all the edges that were used to visit nodes
- Edges that are in G, but not in the DFS Tree are called “***back edges***”

DFS Example (using the algo)



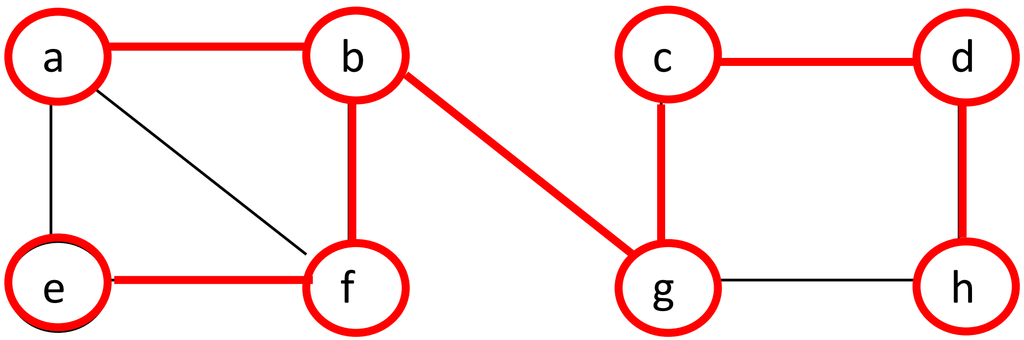
Notes: To trace the operation algorithm we use a stack.

When we make a recursive call (e.g. (dfs(v)), we push v onto the stack.

When v becomes a dead-end (i.e. no more adjacent unvisited neighbors) it is popped off the stack.

Typically we break ties for next unvisited neighbor by using alphabetical order.

DFS Example (using the algo)



DFS: a b f e g c d h

Uses of DFS

DFS is commonly used to:

- find a spanning tree
- find a path from v to u (ie: get out of a maze)
- find a cycle
- find all connected components
  - searching state-space of problems for solution (AI)

Efficiency of DFS

- The basic operation is:

```
for each vertex w in V adjacent to v
    if w has not been visited
        dfs(w)
```

- We can see that this operation will be performed once for each vertex that occurs in the underlying graph structure
  - therefore the #basic ops depends on the size of the structure used to implement the graph

- Basically we need to visit each element of the data structure exactly once. So the efficiency must be:
  - $O(|V|^2)$ for adjacency matrix
  - $O(|V|+|E|)$ for adjacency lists

Which is worse?

- $O(|V|^2)$
- $O(|V|+|E|)$

Breadth-first search (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex

- Instead of a stack, BFS uses a queue

- Similar to level-by-level tree traversal

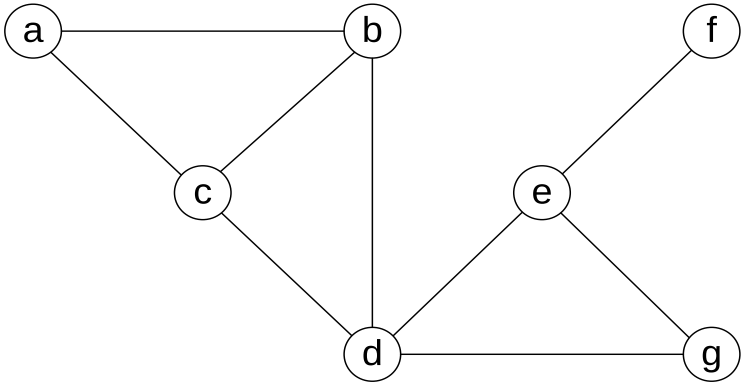
- Also visits a tree-like route throughout the graph, but perhaps a different tree than DFS

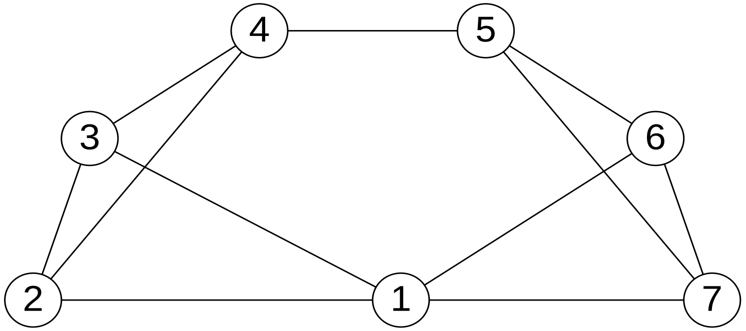
Breadth First Search

Informally:

- for each vertex v in V
- visit all vertices adjacent to v
- when all those vertices have been visited, visit all vertices 2 hops away
- continue in this way until all have been visited

Examples:





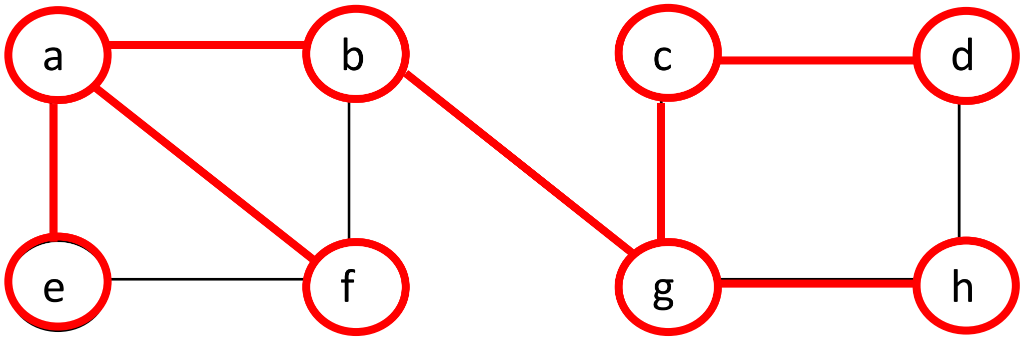
BFS Algorithm

```
BFS(G):
   initialize all visited flags to false
   for each v in V
      if v has not been visited
         bfs(v)
bfs(v)
   visit node v
   initialize a queue Q
   Q.enqueue(v)
   while Q is not empty
      for each w adjacent to Q.head
             if w has not been visited
         visit node w
         add w to Q
      Q.dequeue()
```

- Use a queue (FIFO) to determine which vertex to visit next

- Edges that are in G, but not in the resulting BFS tree are called ***cross-edges***

BFS Example (using the algo)



BFS: a b e f g c h d

Notes on BFS

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - adjacency matrices: $O(|V|^2)$
  - adjacency lists: $O(|V|+|E|)$

- Yields single ordering of vertices (order added/deleted from queue is the same)
  - Whereas with DFS, the order that vertices get *pushed* onto the stack may be different from the order they get *popped*

BFS Applications

- Really the same as DFS
- But… with some judgment… there are applications where BFS seems better:
  - Finding all connected components in a graph
  - Traversing all nodes within one connected component
  - Finding the shortest path (number hops) between two connected vertices

Problems

- In many problems… we need to traverse a graph
- Either DFS or BFS will work
  - But one is better

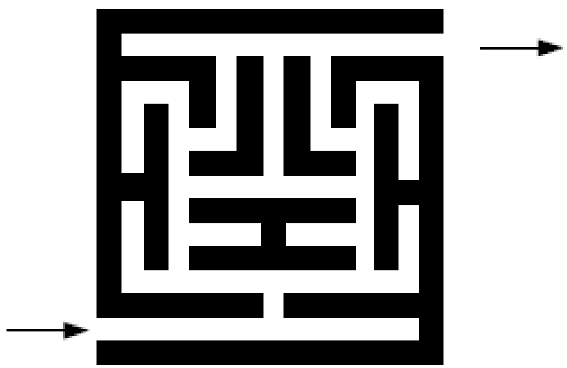
- Consider some examples…

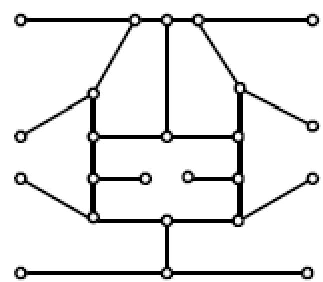
Problem 1: Spanning Tree

- Given a connected graph G, use BFS or DFS to construct a spanning tree of G.
  - use BFS so that we get “shorter” paths between vertices
  - this is a straight-up application of BFS, just build a new graph (the spanner) as we go

Problem 2: Maze Solving

- Model the following maze as a graph. Use DFS to find a path through the maze
  - use DFS because its tree is constructed by moving along existing edges (in contrast, BFS keeps back-tracking to the parent node, so you would have to walk further)





Problem 3: Shortest Path

- Use BFS to find the shortest path between two connected vertices, u and w
  - use BFS because it will find a shortest path (DFS will find "a path" – not always the shortest one)

Step 1: run bfs(u) to create a spanning tree T rooted at u (all paths from in T, starting at root, are shortest)

Step 2: extract the path from T

- use DFS on T, to find any path (as in the previous problem),

Problem 4: Determine Connectivity

- Can you use BFS or DFS to determine if a graph is connected?
  - either will work
  - modify the first loop so that it calls dfs|bfs on any vertex. If there are any unvisited vertices when it returns, the graph is not connected

Practice problems

1. Chapter 1.4, page 37, questions 1,3,9
2. Chapter 3.5, page 128, questions 1,2,4,10