Data Structure

(Chapter 1.4)

Data Structures

- Often… the way you organize the 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 Structure
  - Array
  - Linked list
  - Stack
  - Queue
- Set
- Dictionary (Map)
- Tree
- Graph

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

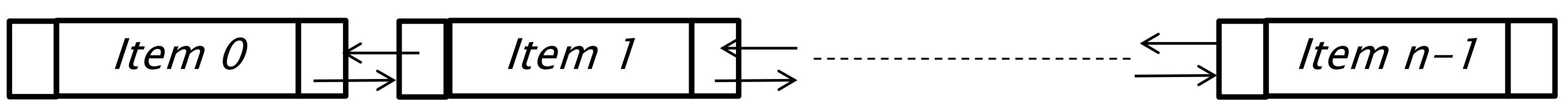
Fundamental Data Structures

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

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 list provides following two advantages over arrays
  - Dynamic size
  - Ease of insertion/deletion
- Linked lists have following drawbacks:
  - Random access is not allowed.

Linked Lists in java

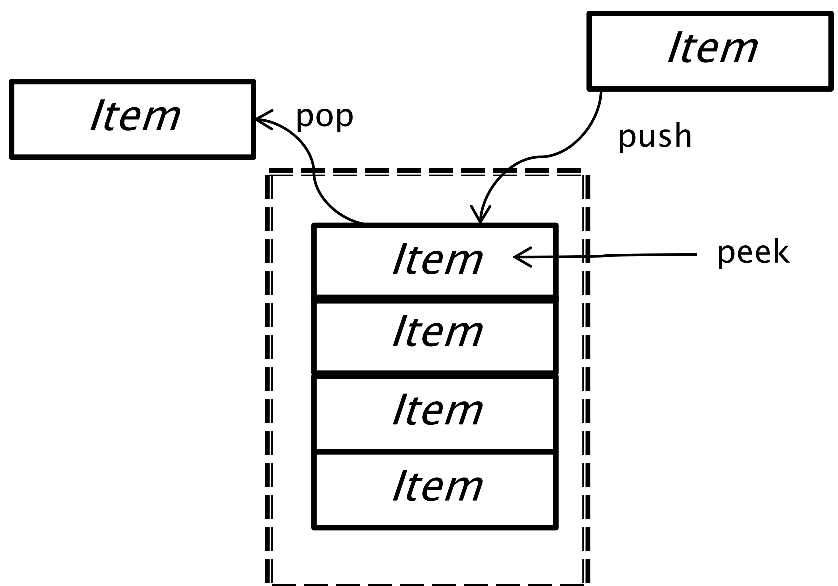
```java
import java.util.*;

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);

   }
}
```

Fundamental Data Structures

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

Stack

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



Stack

- the insert operation is called <u>Push</u>
- the delete operation is called <u>Pop</u>

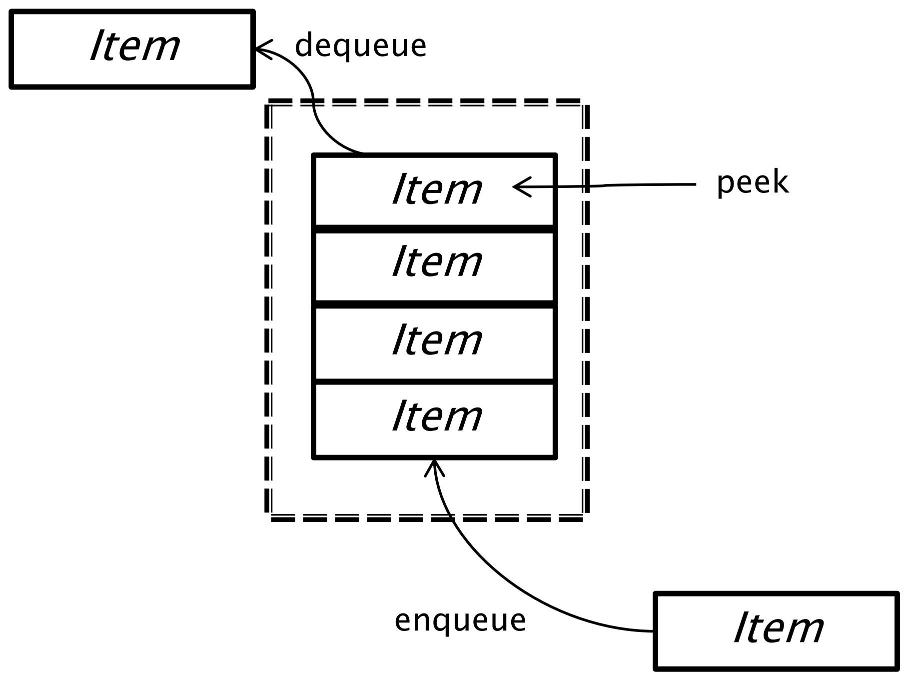
- Example:
  - Analysis of languages (e.g. properly nested brackets)
  - Properly nested: (())
  - Wrongly nested: (()

Stack

```
CheckBalancedParenthesis(exp)
n ← length(exp)
Create a stack s
for i ← 0 to n-1 do
    if (exp[i] is '(' ) do
        Push (exp[i])
    else if (exp[i] is ')' )
        if(s is empty or dose not pair with exp[i])
        return False
    else
        pop()
If (s is empty)
    return true
else
    return false
```

Queues

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



Fundamental Data Structures

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

Set

- a Set is just like a Set in math, ie: set = { 1, 2, 3, 4 }
- the key thing to remember:
  - *sets cannot contain duplicate items*
- all we can really do with a set is:
  - **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

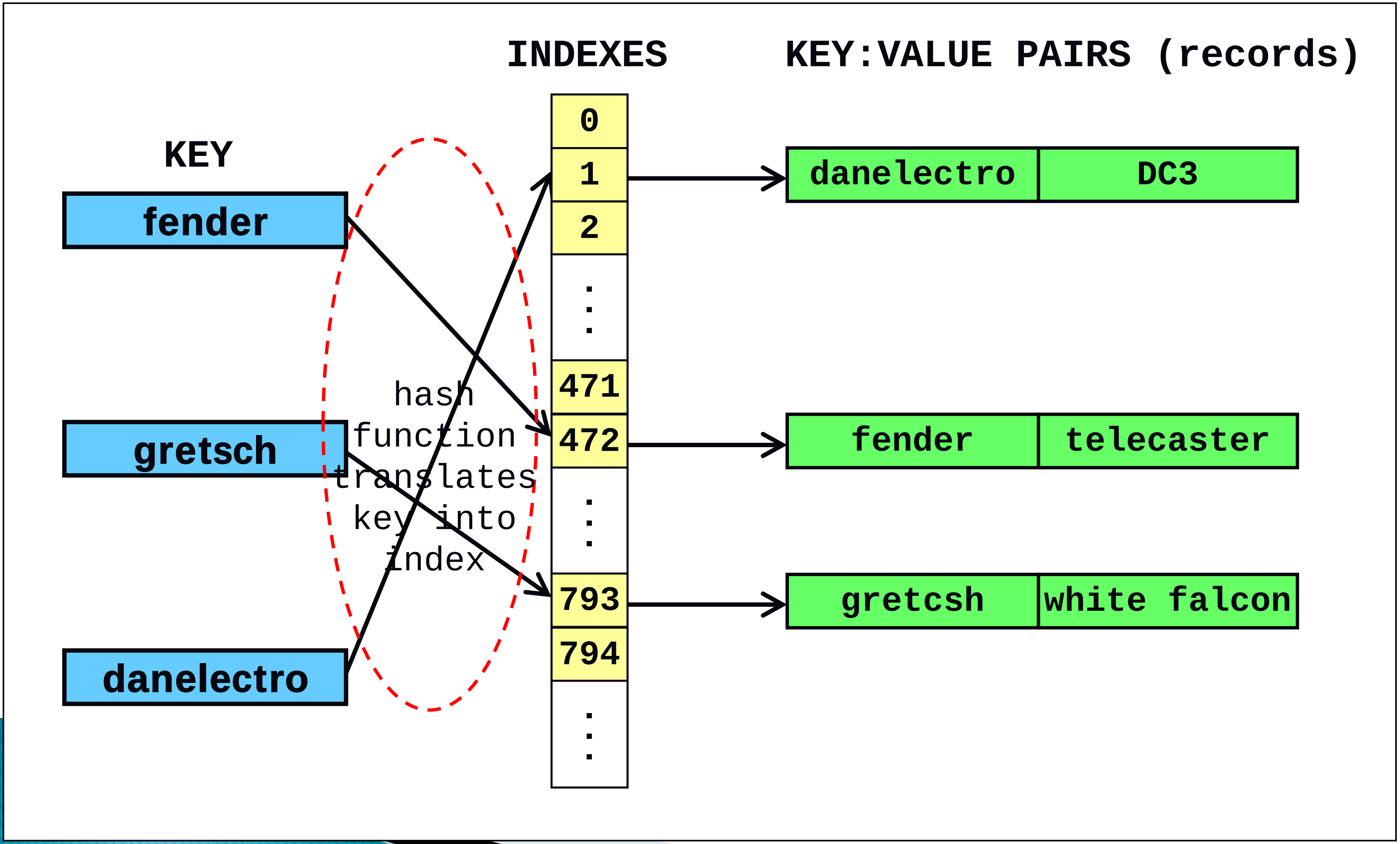
- there are a few different ways to implement Set
  - HashSet:
    - *HashSet* is the fastest implementation, *but it is unordered*
  - TreeSet
    - *TreeSet* is slower, *but maintains a sorted order*

Fundamental Data Structures

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

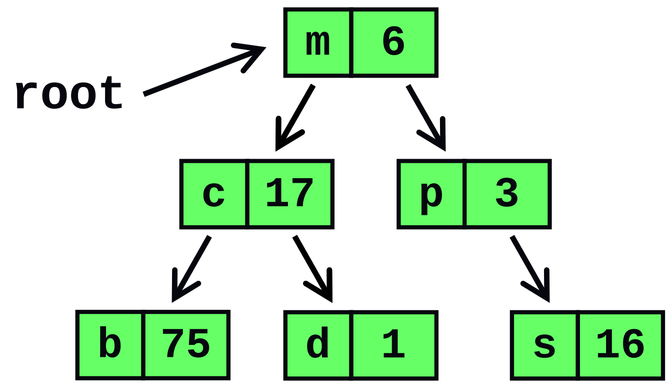
Map (as a hash table)

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



Map (as a balanced tree)

- tree implementations, using red-black trees

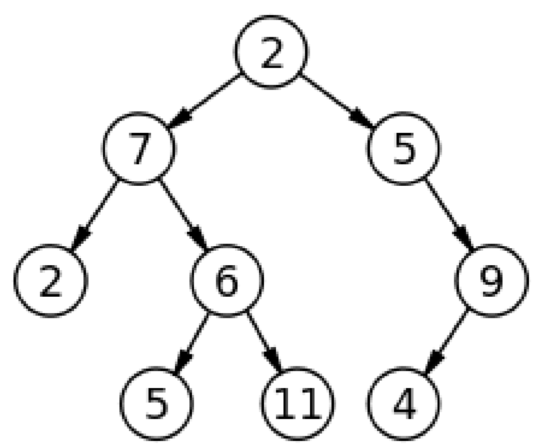


Fundamental Data Structures

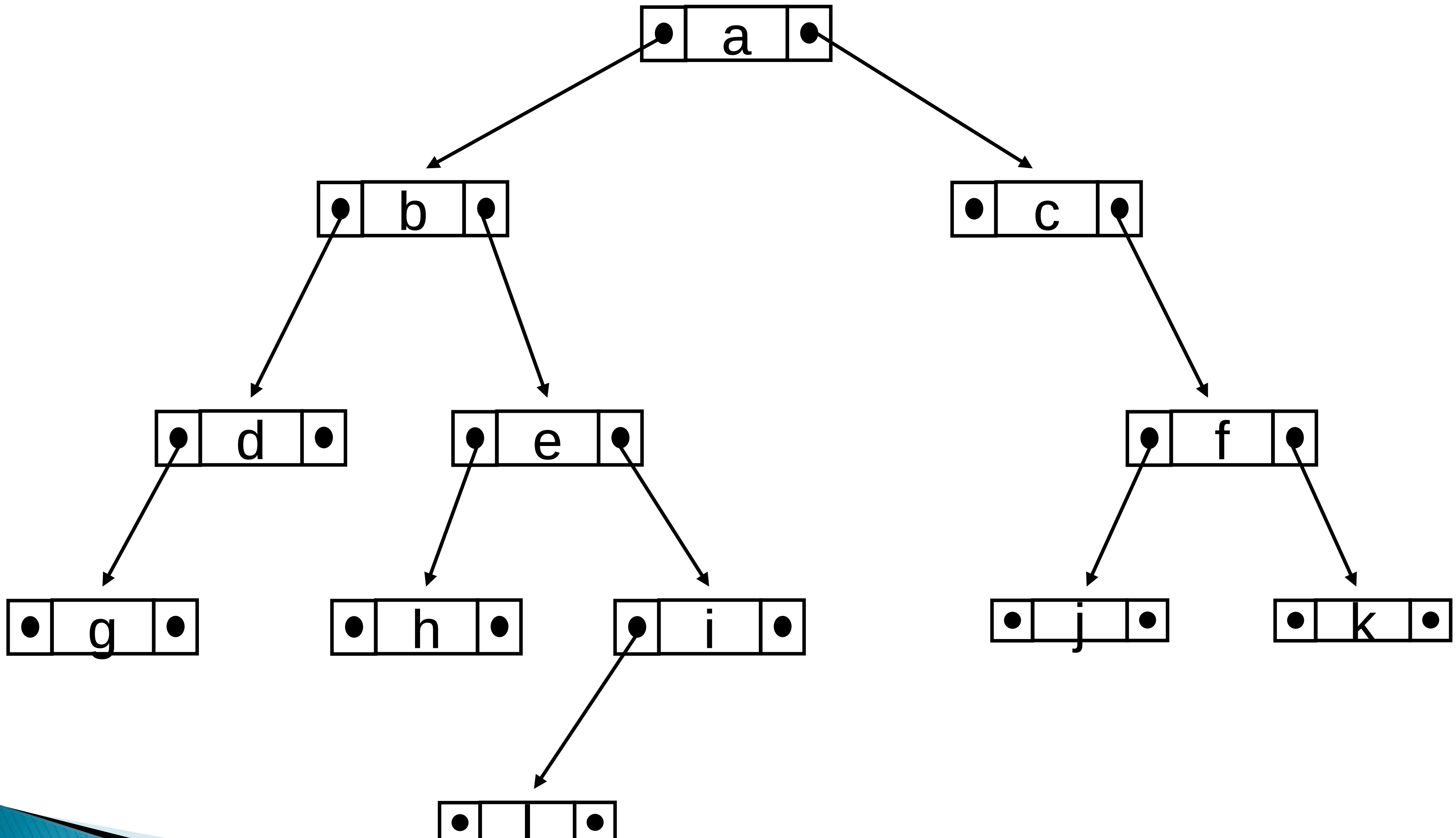
- Linear Data Structure
  - Array
  - Linked list
  - Stack
  - Queue
- Set
- Dictionary (Map)
- Tree
- Graph

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

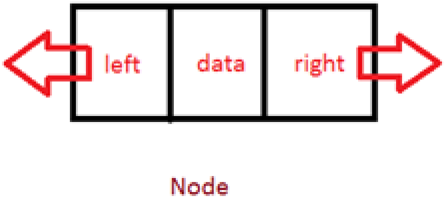


Binary tree implementation



Tree node implementation

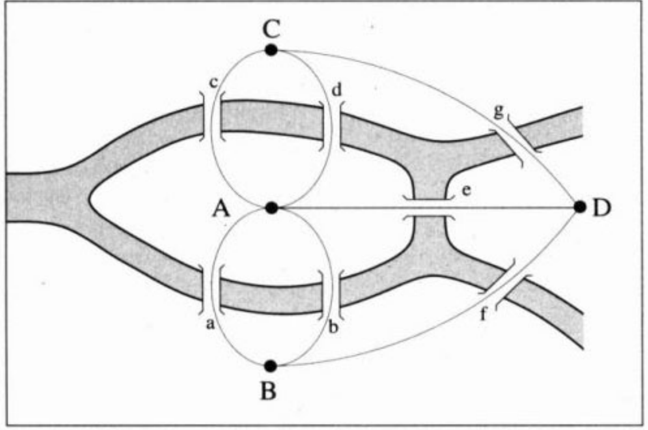
```
public class Node {
        public int data;
        public Node left;
        public Node right;

        public Node(int d) {
                data = d;
        }
```



Fundamental Data Structures

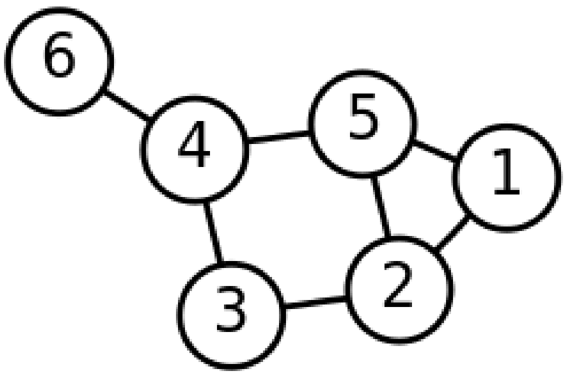
- Linear Data Structure
  - Array
  - Linked list
  - Stack
  - Queue
- Set
- Dictionary (Map)
- Tree
- Graph

Graphs

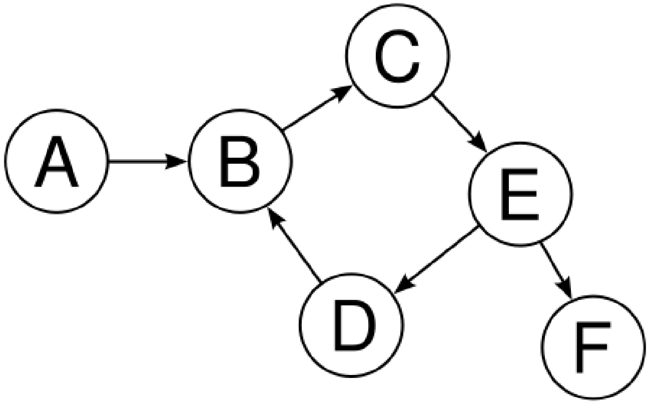


Motivation: Real world connections

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



Undirected



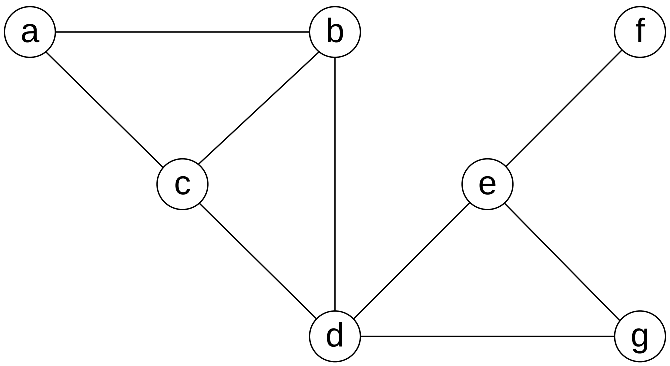
Directed

Representing Graphs

1. Adjacency matrix
2. Adjacency lists

Representation: Adjacency Matrix

- For this graph:

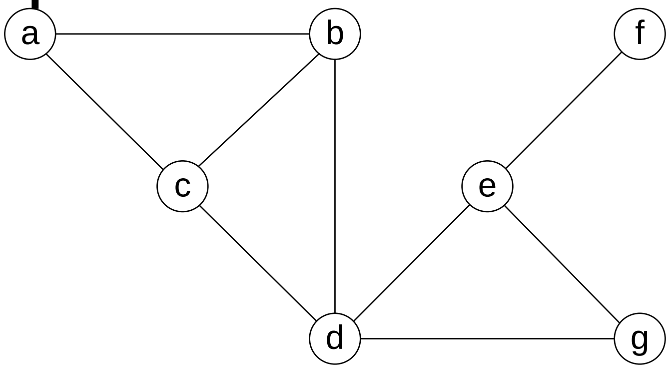


- Adjacency matrix is the following:

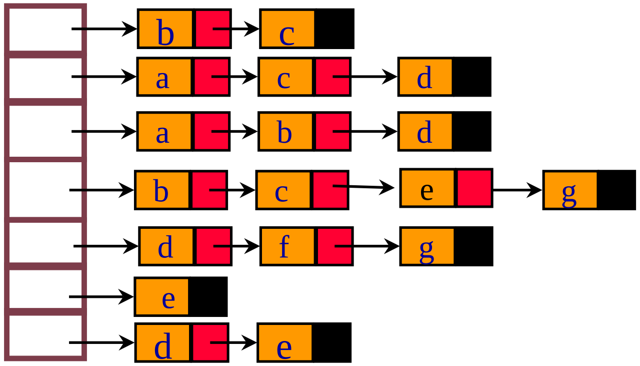
	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	1	0	0	0
d	0	1	1	0	1	0	1
e	0	0	0	1	0	1	1
f	0	0	0	0	1	0	0
g	0	0	0	1	1	0	0

Representation: Adjacency List

- For this 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

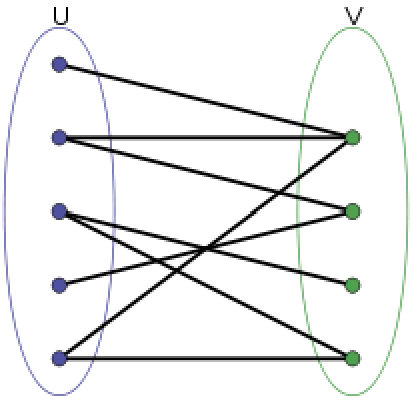
1. Adjacency matrix
   - Or Weight Matrix for weighted graphs

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

Properties

- *Connected graph*
  - A graph where there is a path connecting each pair of vertices
- *Bi-partite graph*
  - Vertices can be divided into two separate sets u and v, so that all edges go from set u to set v



Properties

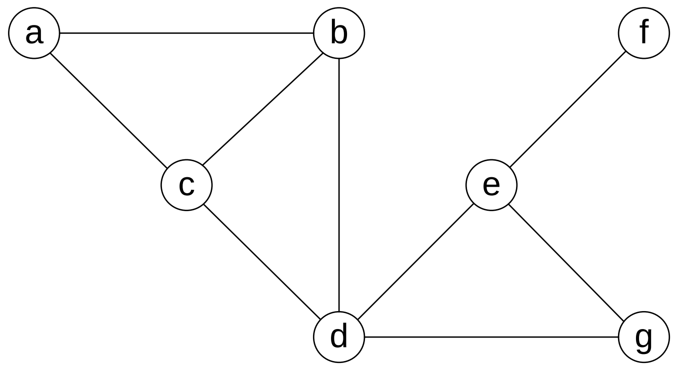
- *Cyclic graph*
  - A graph containing at least one cycle
  - (must have 3 vertices)
- *Acyclic graph*
  - A graph containing no cycles

Graph Algorithm

(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
- "Re-draws" graph in a tree-like fashion

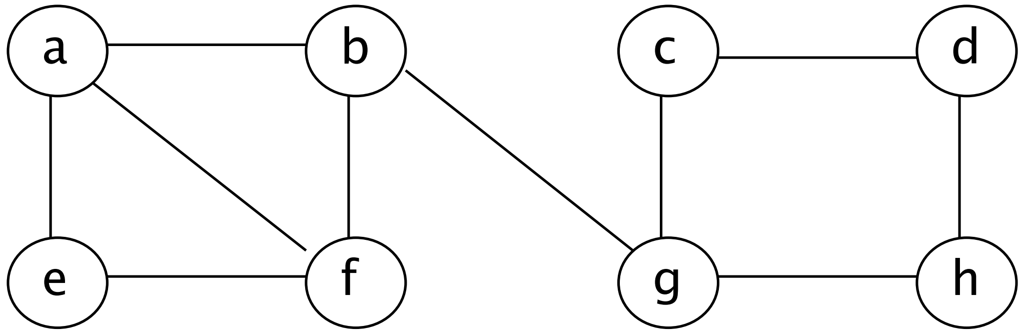
DFS

Algorithm:

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

- the stmt "visit node v" should be replaced by whatever you are doing
- the output is typically a "*DFS Tree*", which is a tree containing all the edges that are used to visit node
- edges that are in G, but not in the DFS Tree are called "*back edges*"

DFS Example (using the algo)



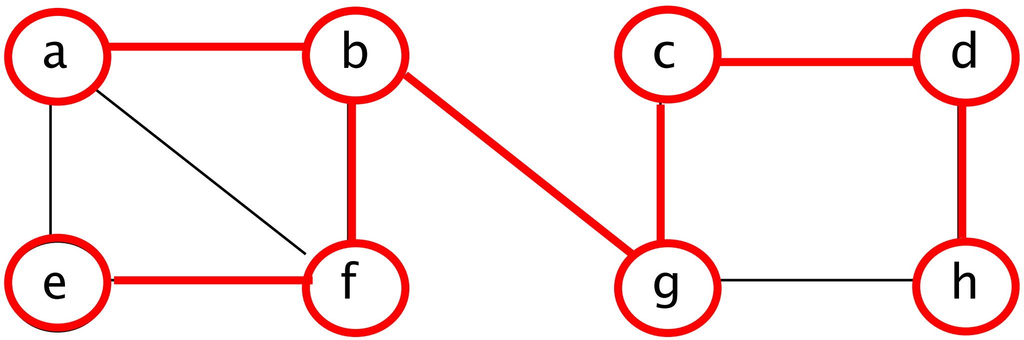
Notes: To trace the operation algorithm we use a stack.

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

When v becomes a dead-end (ie: 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 adacent 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

Breadth-first search (BFS)

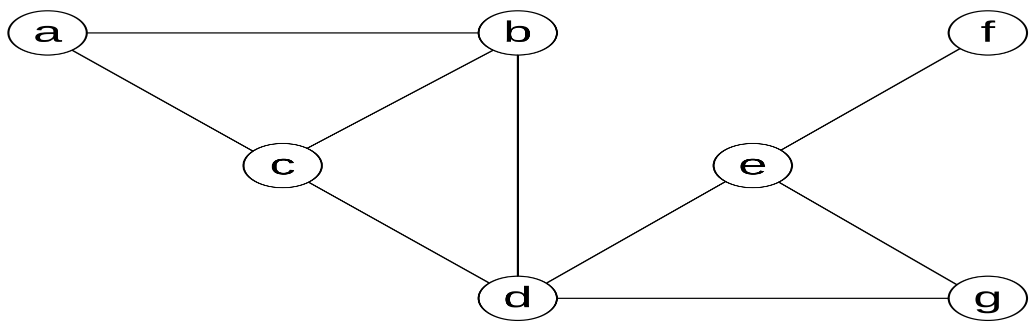
- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- “Redraws” graph in tree-like fashion (with tree edges and cross edges for undirected graph)

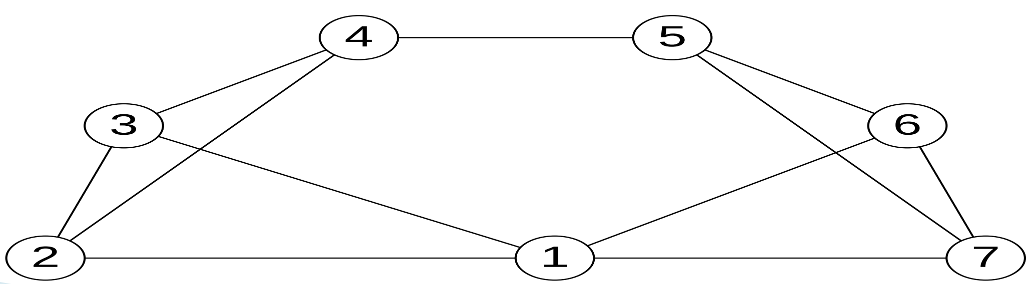
Breadth First Search

Informally:

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

Examples:



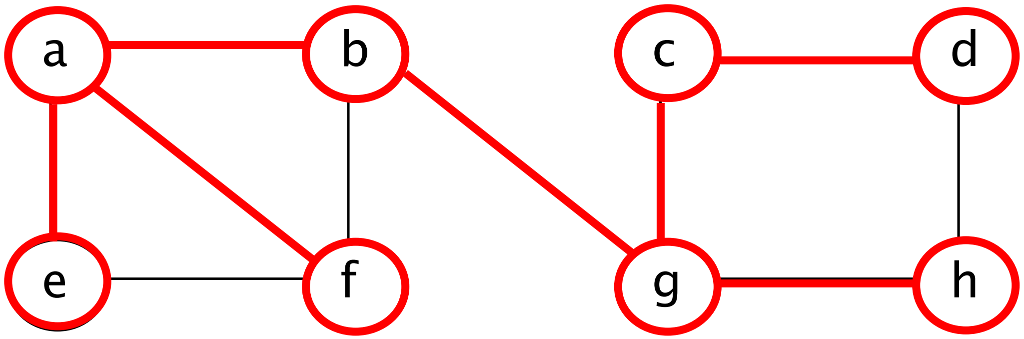


BFS Algorithm

```
BFS(G):
   init 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
   add v to Q
   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
         remove Q.head from Q
```

- the stmt "visit node v|w" can be replaced by whatever you are doing
- 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)

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

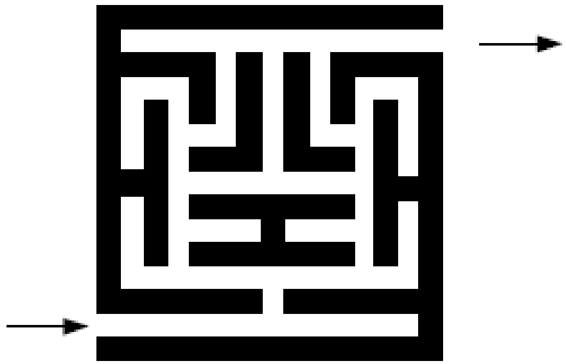
- Consider some examples…

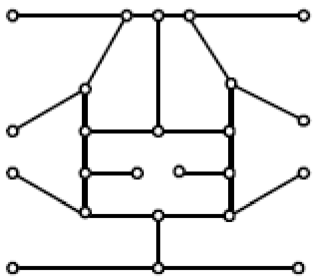
Problem 1: Spanning Tree

- 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

- Explain how you can 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

Try it/ homework

1. Chapter 1.4, page37, question 1,3,9
2. Chapter 3.5, page 128, questions 1,2,4,10

QUIZ Announcement

- There will be a quiz in the lab next week.

- It will be 5 questions, on D2L
  - It will take 10-20 minutes
  - Followed by a lab activity