Programming & Data Structures

Module -II

Programme: B. Tech-CSE,

Course: SKE309

Faculty: Dr. S. K. Dubey

Index

- Linked List
- Tree
- Graph

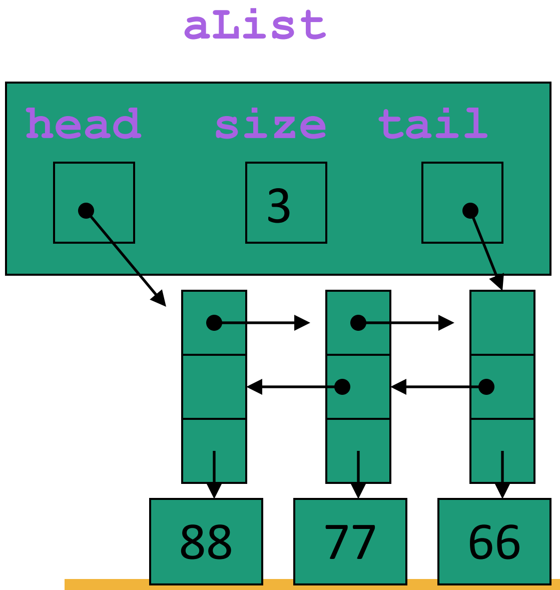
`ArrayList` Drawback

- Problems arise from using an array
  - values can be added only at back of `ArrayList`
  - to insert a value and "shift" others after it requires extensive copying of values
  - similarly, deleting a value requires shifting
- We need a slightly different structure to allow simple insertions and deletions
  - the `LinkedList` class will accomplish this

The `LinkedList` Class

- Given

```
LinkedList alist = new LinkedList();
. . .
aList.add(new(integer(88));
aList.add(new(integer(77));
aList.add(new(integer(66));
```



Resulting object shown at left

Variations on Linked Lists

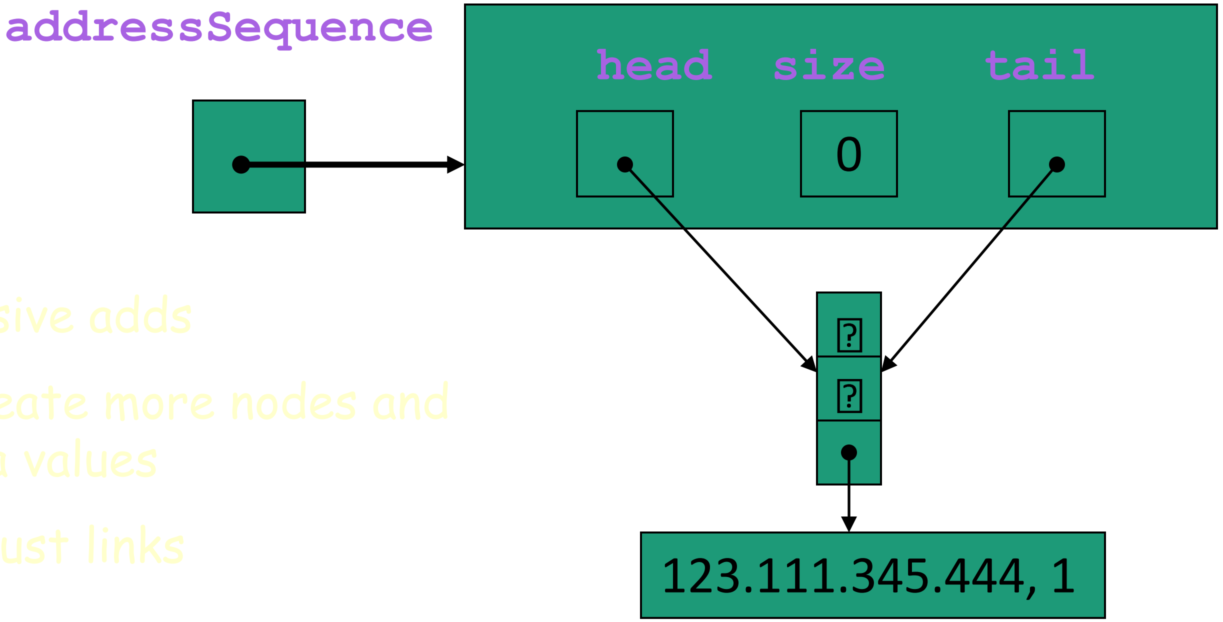
- Lists can be linked doubly as shown
- Lists can also be linked in one direction only
  - attribute would not need link to tail
  - node needs forward link and pointer to data only
  - last item in list has link set to null
- Lists can be circularly linked
  - last node has link to first node

Using a `LinkedList`

- Solve the IP address counter to use `LinkedList`
- Note source code, Figure 12.3
  - receives text file via `args[0]`
  - reads IP addresses from file
  - prints listing of distinct IP addresses and number of times found in file

Adding to the Linked List

- Results of command for first add
  `addressSequence.add(anAddressCounter);`



- Successive adds
  - create more nodes and data values
  - adjust links

Accessing Values in a Linked List

- Must use the `.get` method

```
((AddressCounter)
 addresssSequence.get(index)).incrementCount();
```

- A `LinkedList` has no array with an index to access an element
- get method must …
  - begin at `head` node
  - iterate through `index` nodes to find match
  - return reference of object in that node
- Command then does cast and `incrementCount()`

Accessing Values in a Linked List

- To print successive values for the output

```
for (int i = 0; i < addressSequence.size(); i++)
System.out.println(addressSequence.get(i));
```

`get(i)` starts at first node, iterates `i` times to reach desired node

`size` method determines limit of loop counter

- Note that each `get(i)` must pass over the same first `i-1` nodes previously accessed
- This is inefficient

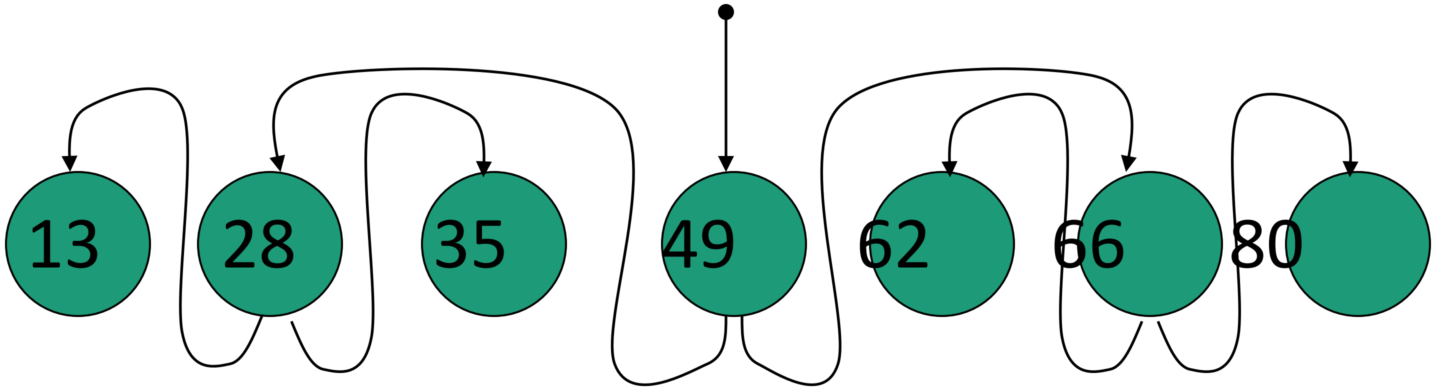
Inserting Nodes Anywhere in a Linked List

- Recall problem with **`ArrayList`**
  - can add only at end of the list
  - linked list has capability to insert nodes anywhere
- We can say
  `addressSequence.add(n, new anAddressCounter);`
  Which will …
  - build a new node
  - update **`head`** and **`tail`** links if required
  - update node handle links to place new node to be n^{th} item in the list
  - allocates memory for the data item

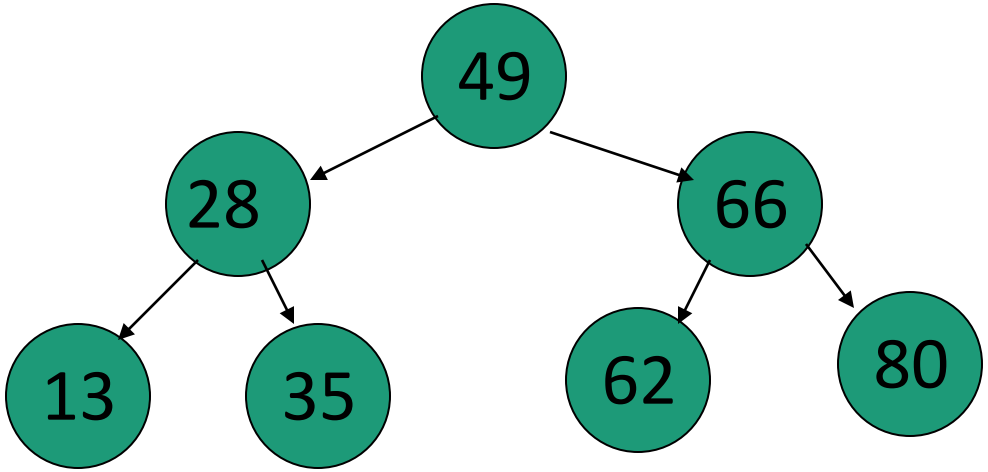
An Introduction to Trees

- We seek a way to organized a linked structure so that …
  - elements can be searched more quickly than in a linearly linked structure
  - also provide for easy insertion/deletion
  - permit access in less than O(n) time
- Recall binary search strategy
  - look in middle of list
  - keep looking in middle of subset above or below current location in list
  - until target value found

Visualize Binary Search

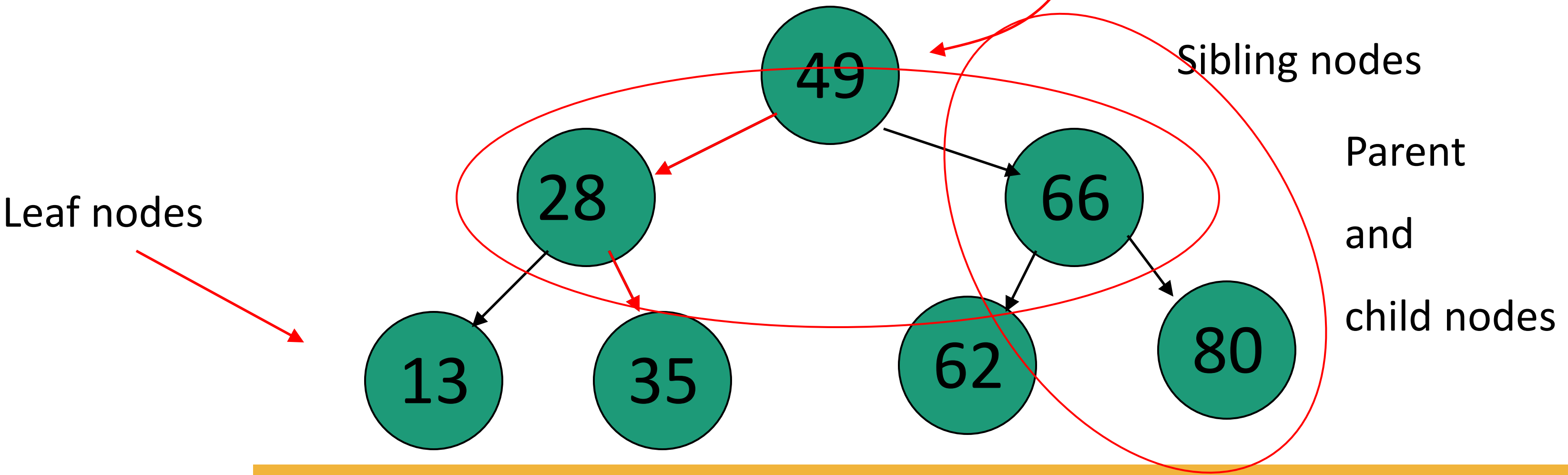


Drawn as a *binary tree*



Tree Terminology

- A tree consists of:
  - finite collection of nodes
  - non empty tree has a root node
  - root node has no incoming links
  - every other node in the tree can be reached from the root by unique sequence of links



Applications of Trees

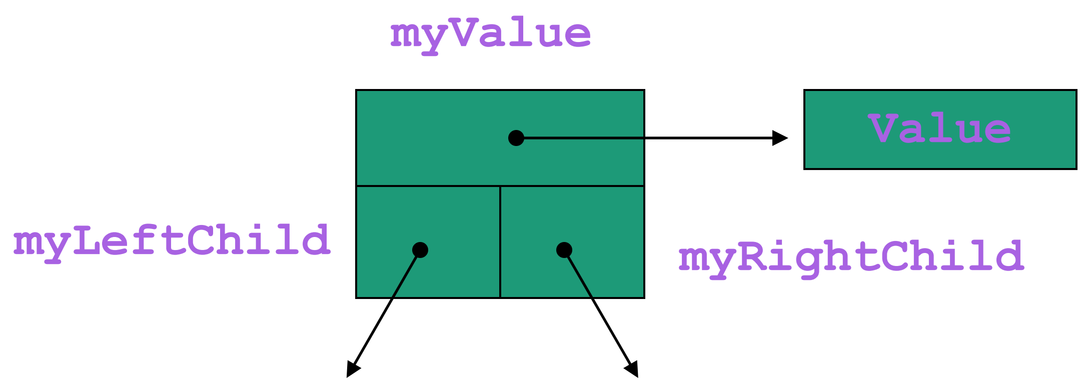
- Genealogical tree
  - pictures a person's descendants and ancestors
- Game trees
  - shows configurations possible in a game such as the Towers of Hanoi problem
- Parse trees
  - used by compiler to check syntax and meaning of expressions such as 2 * (3 + 4)

Examples of Binary Trees

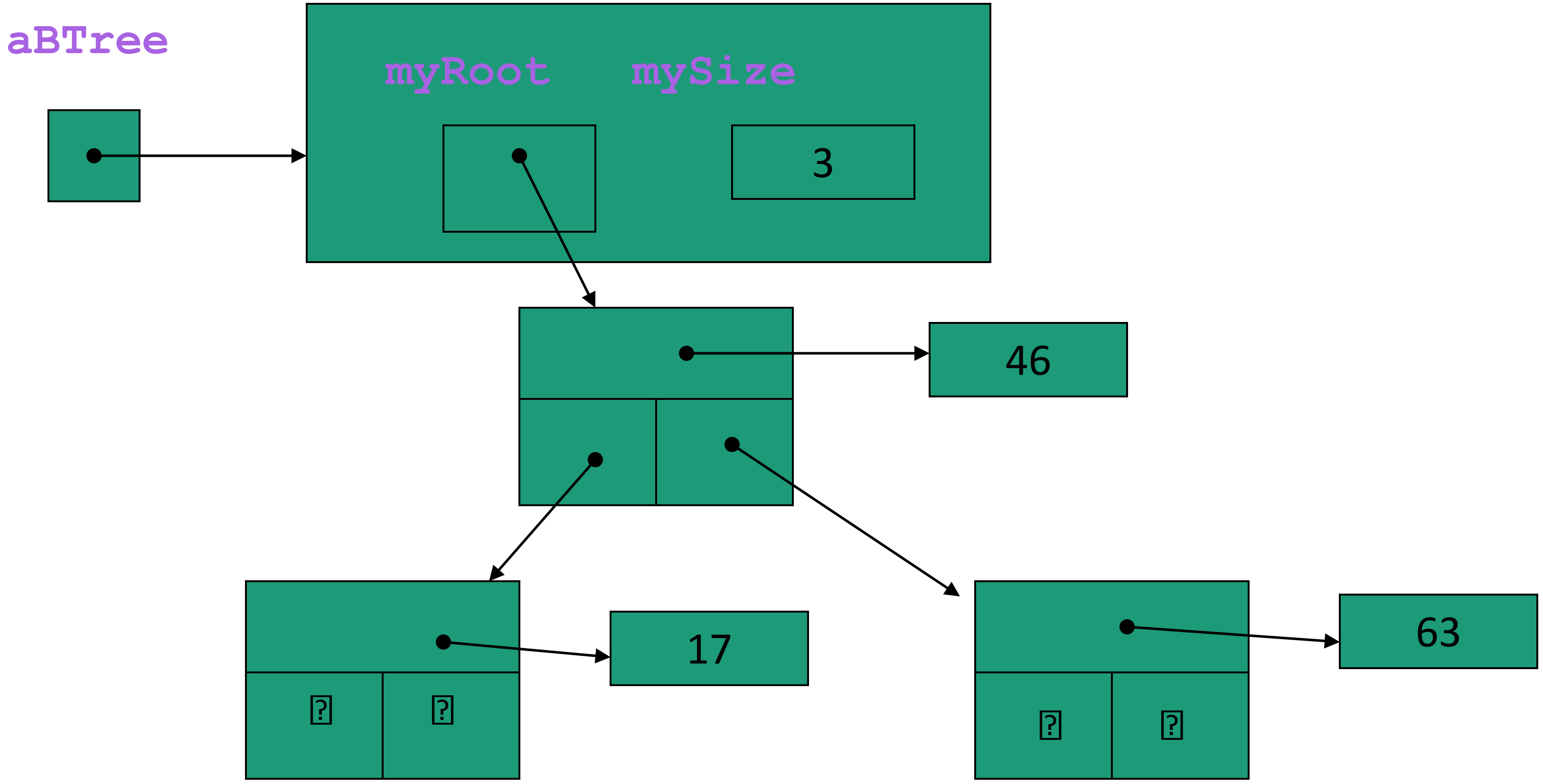
- Each node has at most two children
- Useful in modeling processes where a test has only two possible outcomes
  - true or false
  - coin toss, heads or tails
- Each unique path can be described by the sequence of outcomes
- Can be applied to decision trees in expert systems of artificial intelligence

Implementing Binary Trees

- Binary tree represented by multiply linked structure
  - each node has two links and a handle to the data
  - one link to left child, other to the right



Visualizing a BinaryTree



Binary Search Trees

Search Algorithm

1. Initialize a handle **currentNode** to the node containing the root
2. Repeatedly do the following:

```
If target_item < currentNode.myValue
        set currentNode = currentNode.leftChild
If target_item > currentNode.myValue
        set currentNode = currentNode.rightChild
Else
        terminate repetition because target_item has been found
```

Tree Traversals

- A traversal is moving through the binary tree, visiting each node exactly once
  - for now order not important
- Traverse Algorithm

1. Visit the root and process its contents
2. Traverse the left subtree
   1. visit its root, process
   2. traverse left sub-sub tree
   3. traverse right sub-sub tree
3. Traverse the right subtree
   1. …

Tree Traversal is Recursive

If the binary tree is empty then
  do nothing

Else
  L: Traverse the left subtree
  N: Visit the root
  R: Traverse the right subtree

The "anchor"

The inductive step

Traversal Order

Three possibilities for inductive step …

- Left subtree, Node, Right subtree
  the <u>inorder</u> traversal

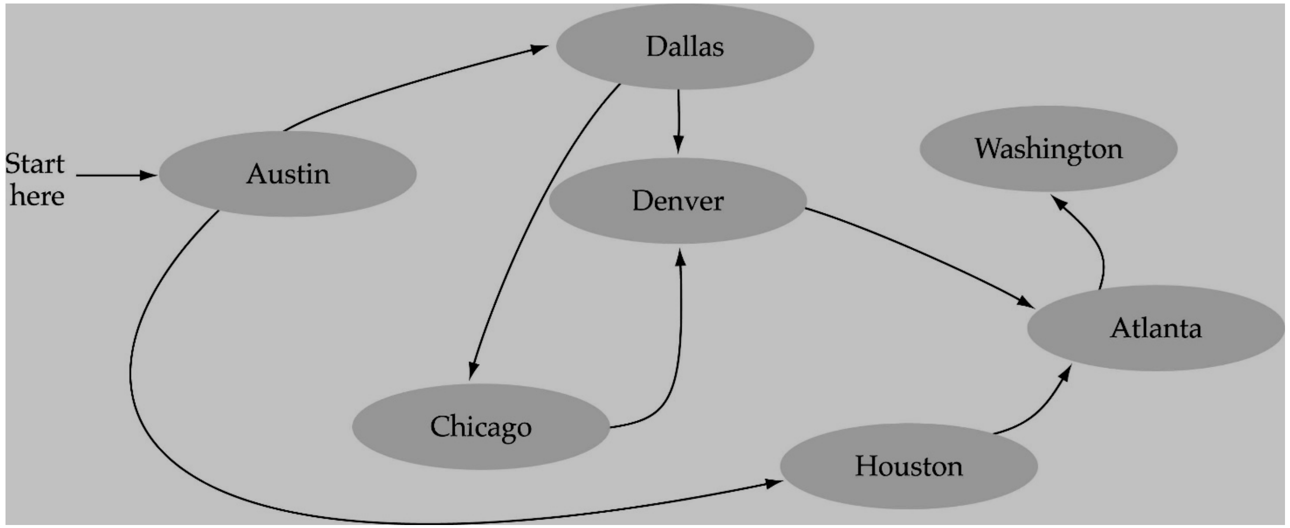
- Node, Left subtree, Right subtree
  the <u>preorder</u> traversal

- Left subtree, Right subtree, Node
  the <u>postorder</u> traversal

Graphs

What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices



Formal definition of graphs

- A graph G is defined as follows:

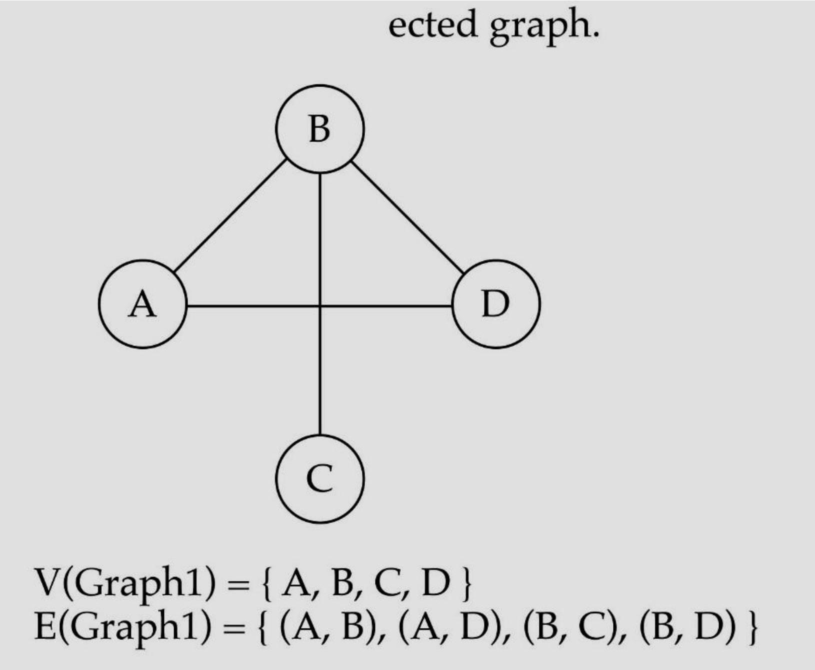
$$G=(V,E)$$

V(G): a finite, nonempty set of vertices

E(G): a set of edges (pairs of vertices)

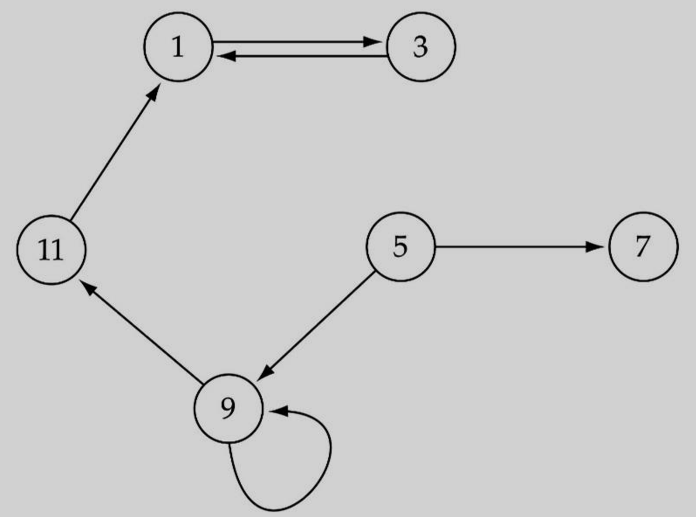
Directed vs. undirected graphs

- When the edges in a graph have no direction, the graph is called *undirected*



Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)



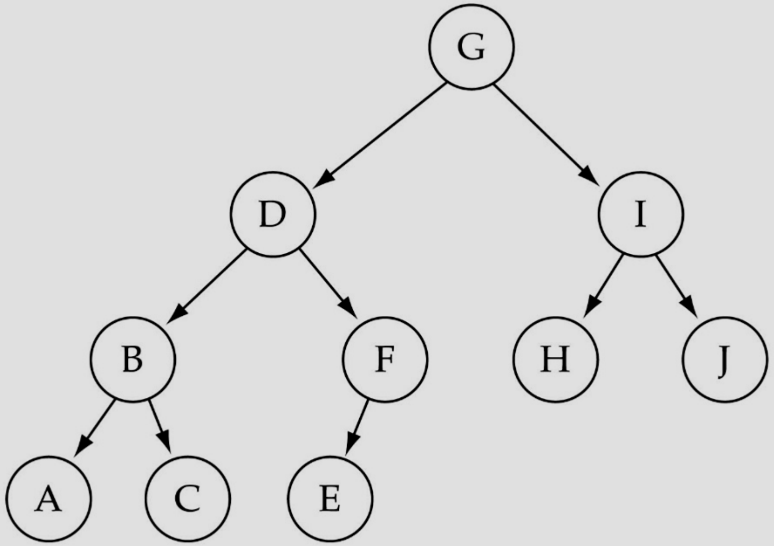
(b) Graph2 is a directed graph.

V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7) 1), (9, 9), (11, 1) }

Trees vs graphs

- Trees are special cases of graphs!!

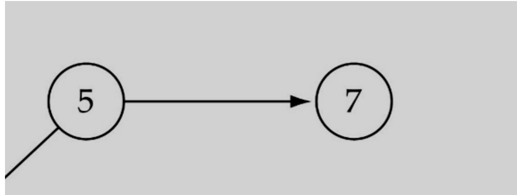
(c) Graph3 is a directed graph.



V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge

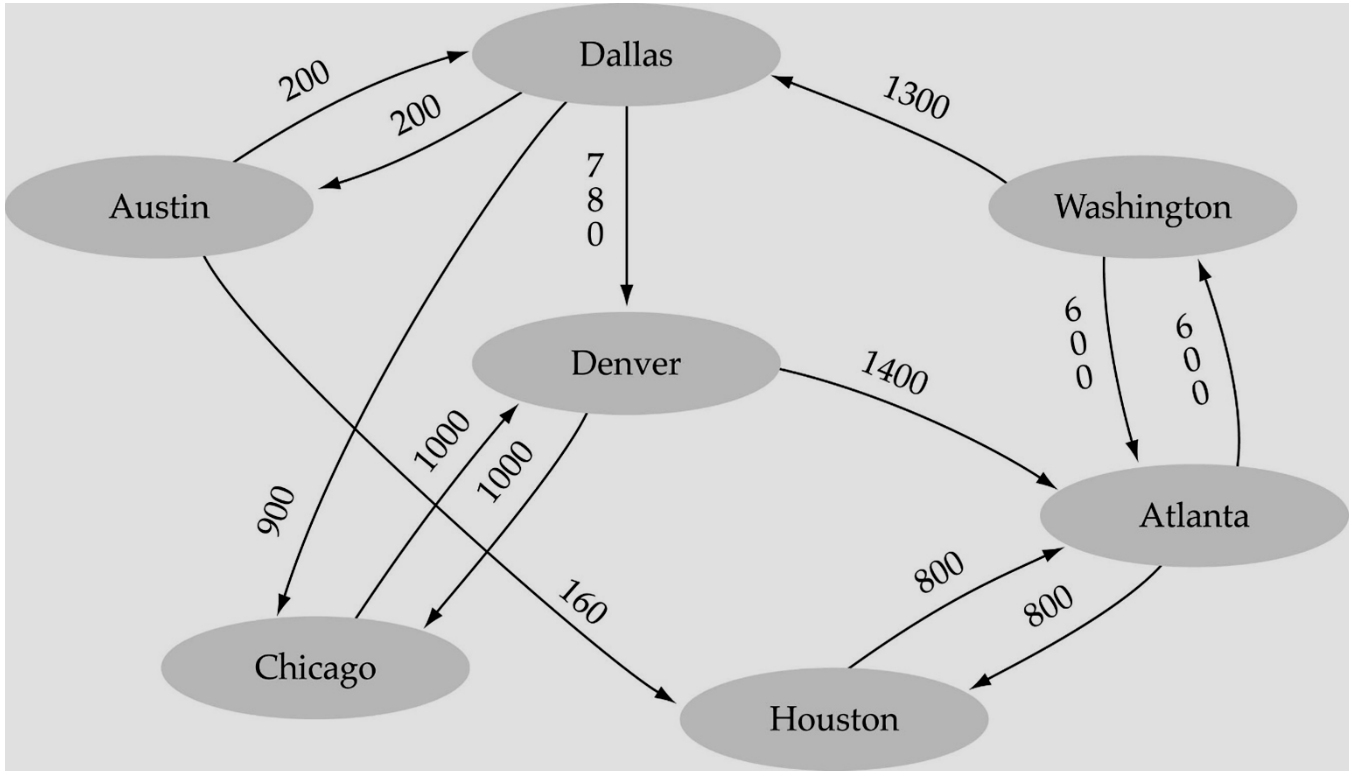


to
from

- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

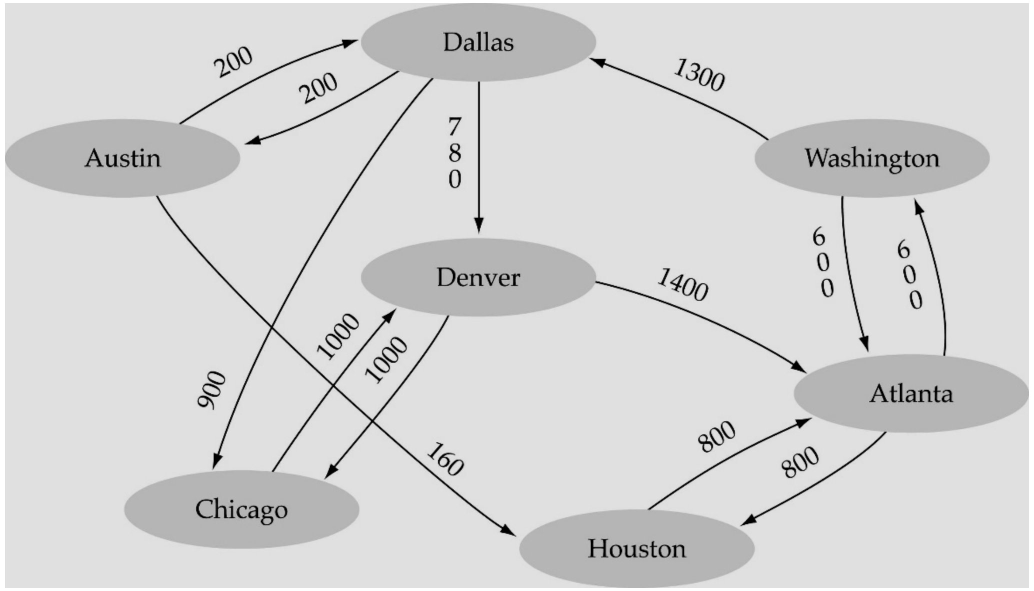
Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value



Graph implementation

- Array-based implementation
  - A 1D array is used to represent the vertices
  - A 2D array (adjacency matrix) is used to represent the edges



graph

.numVertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

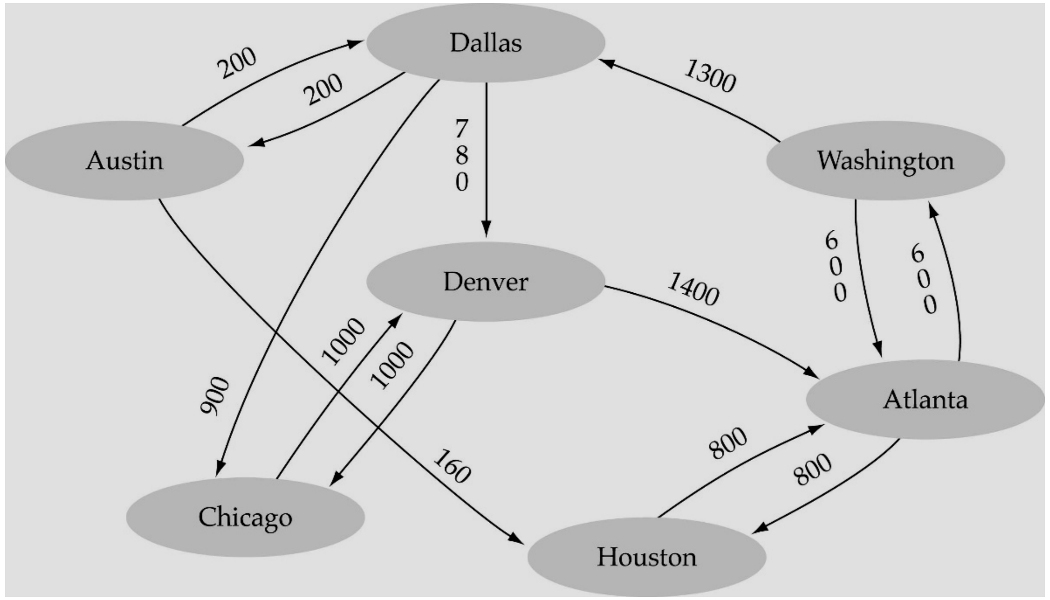
.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

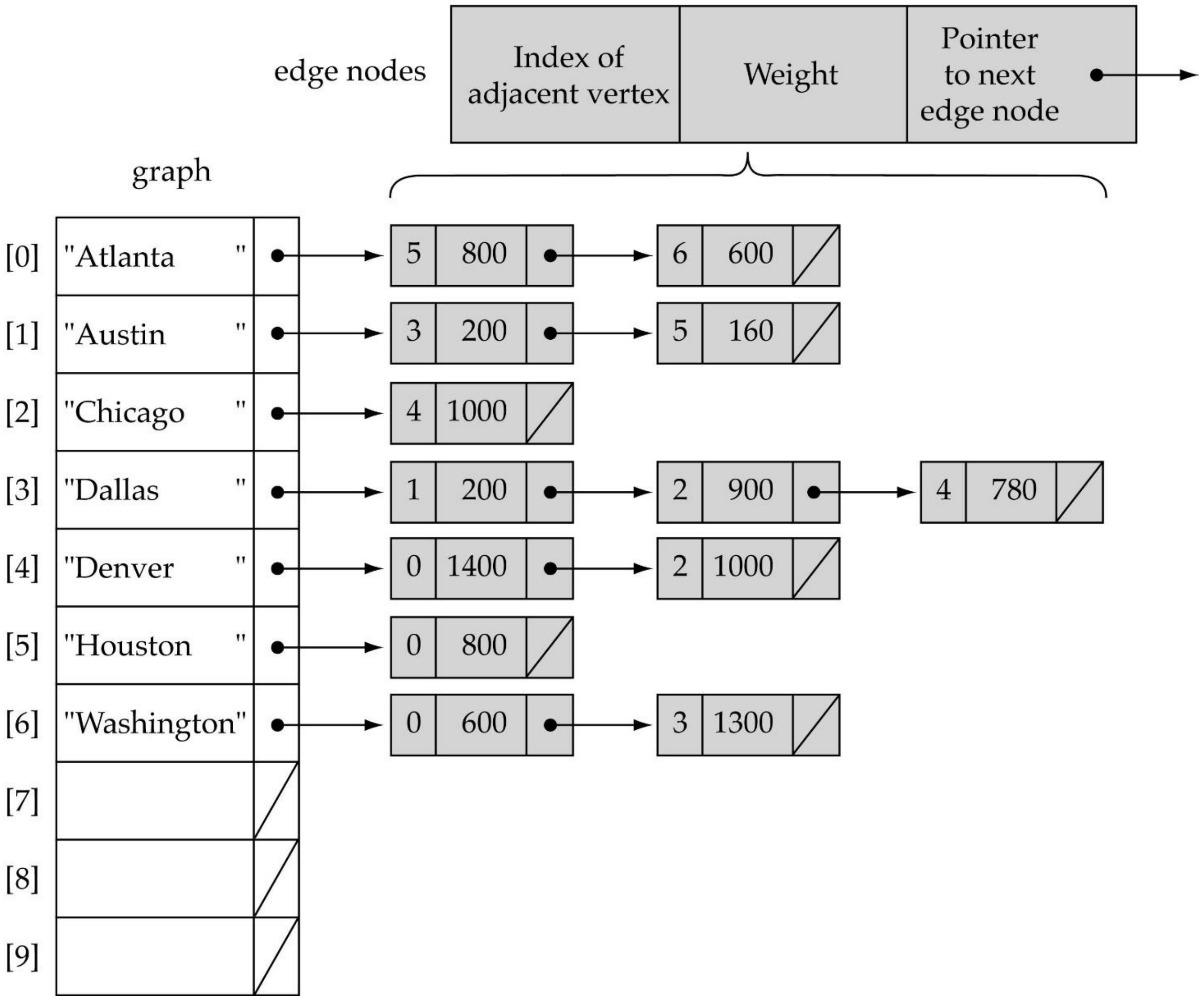
(Array positions marked '•' are undefined)

Graph implementation (cont.)

- Linked-list implementation
  - A 1D array is used to represent the vertices
  - A list is used for each vertex *v* which contains the vertices which are adjacent from *v* (adjacency list)



(a)


edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node
graph
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
5 800
6 600
3 200
5 160
4 1000
1 200
2 900
4 780
0 1400
2 1000
0 800
0 600
3 1300

Adjacency matrix vs. adjacency list representation

- **Adjacency matrix**
  - Good for dense graphs --$|E|$~$O(|V|^2)$
  - Memory requirements: $O(|V| + |E|) = O(|V|^2)$
  - Connectivity between two vertices can be tested quickly
- **Adjacency list**
  - Good for sparse graphs -- $|E|$~$O(|V|)$
  - Memory requirements: $O(|V| + |E|)=O(|V|)$
  - Vertices adjacent to another vertex can be found quickly

Graph searching

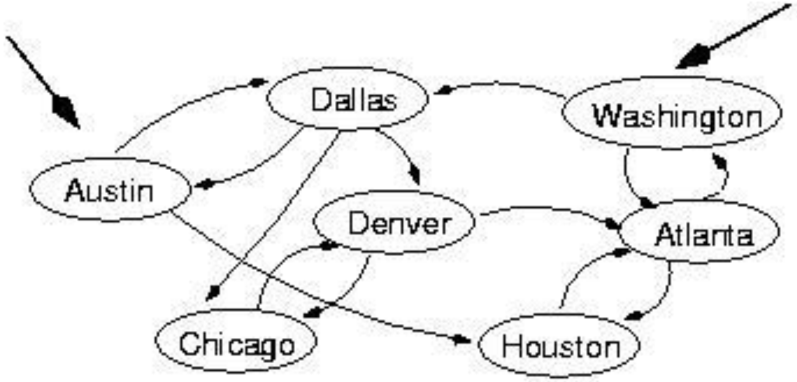
- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)
- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)

Depth-First-Search (DFS)

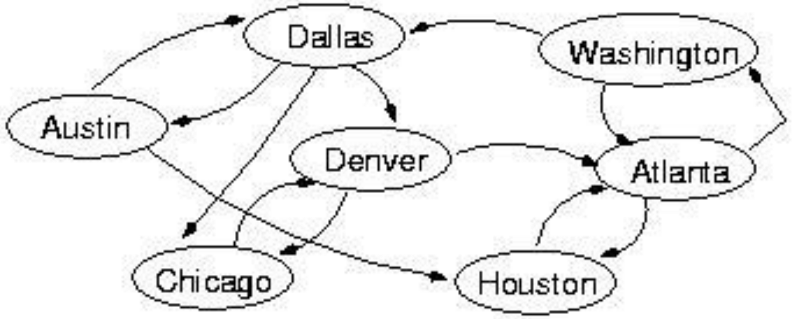
- What is the idea behind DFS?
  - Travel as far as you can down a path
  - Back up *as little as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)
- DFS can be implemented efficiently using a *stack*

Depth-First-Search (DFS) *(cont.)*

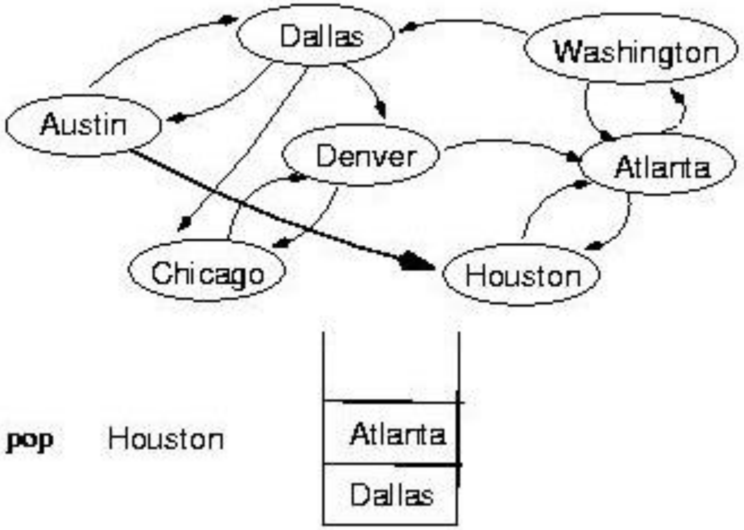
```
Set found to false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Push all adjacent vertices onto stack
WHILE !stack.IsEmpty() AND !found

IF(!found)
  Write "Path does not exist"
```



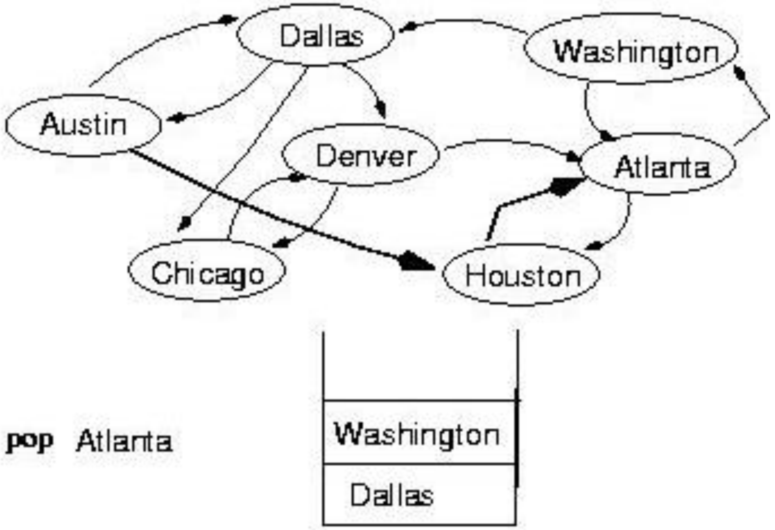
(initialization)



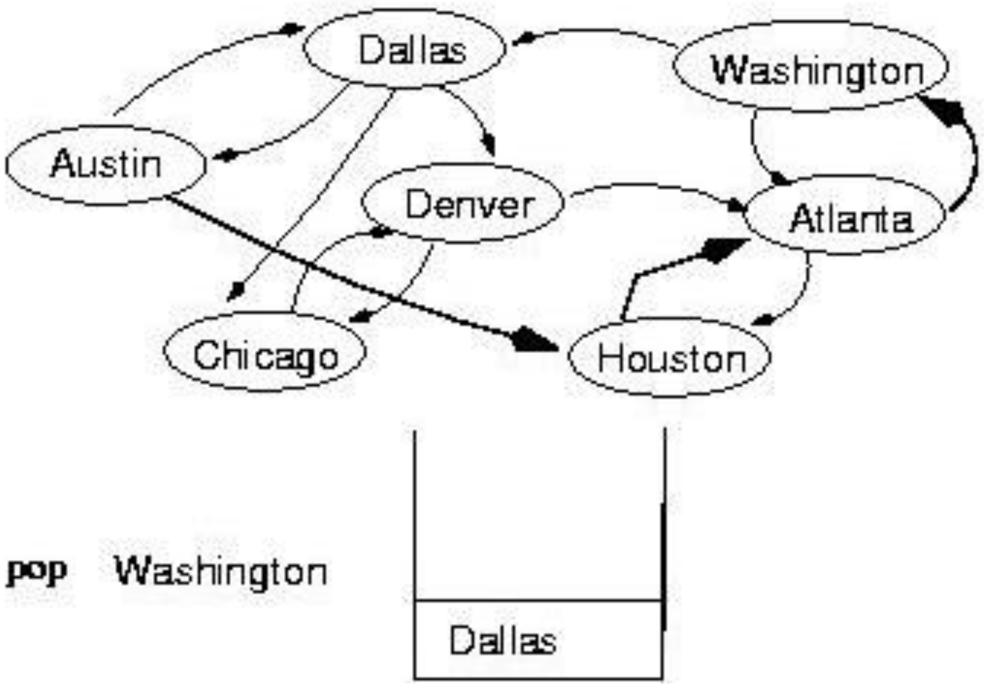
pop Austin



pop Houston



pop Atlanta


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
pop Washington
Dallas

Breadth-First-Searching (BFS)

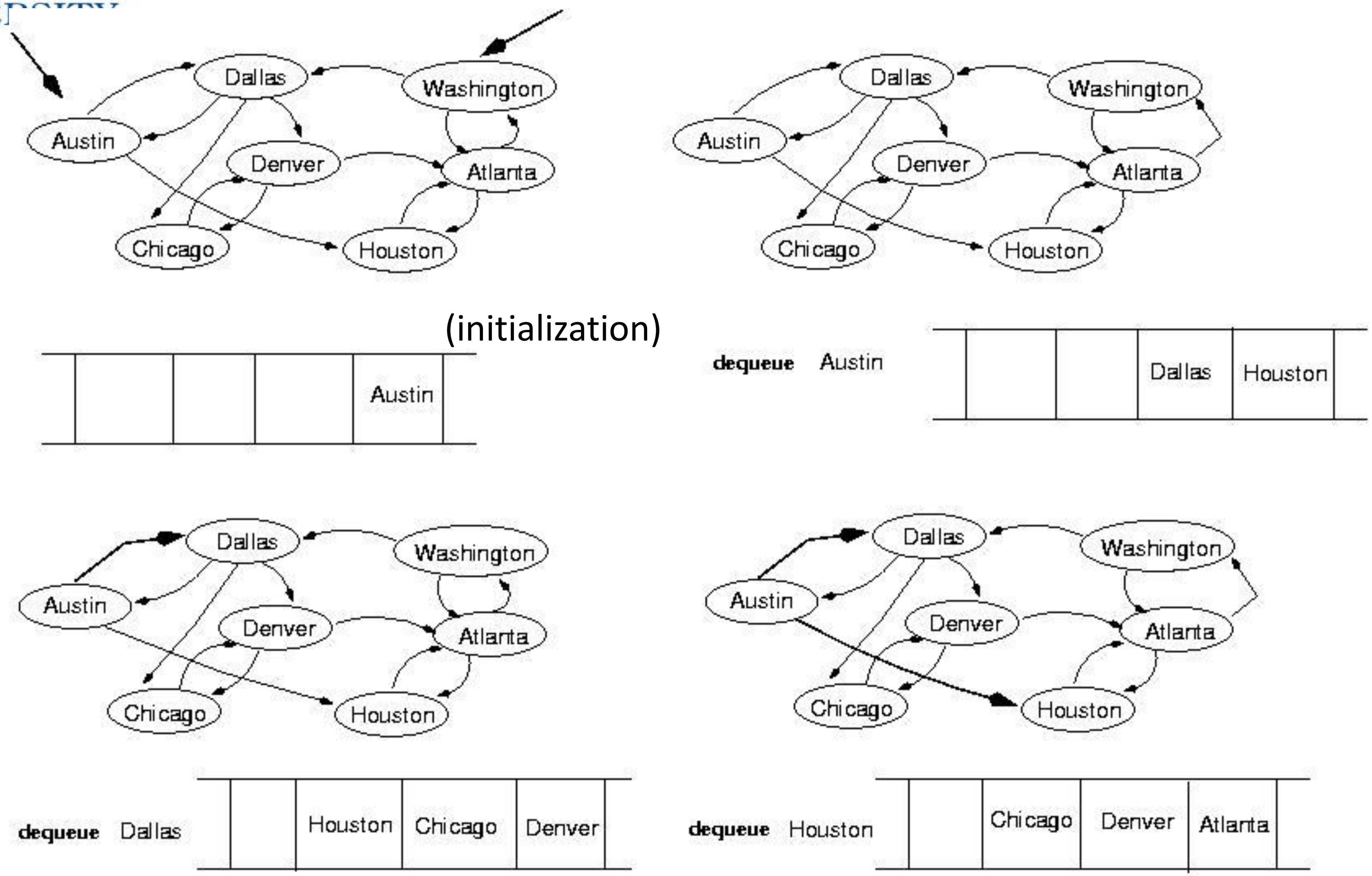
- What is the idea behind BFS?
  - Look at all possible paths at the same depth before you go at a deeper level
  - Back up *as far as possible* when you reach a "dead end" (i.e., next vertex has been "marked" or there is no next vertex)

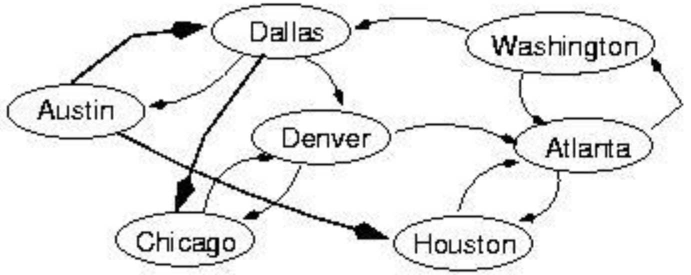
Breadth-First-Searching (BFS) (cont.)

- BFS can be implemented efficiently using a *queue*

```
Set found to false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
```

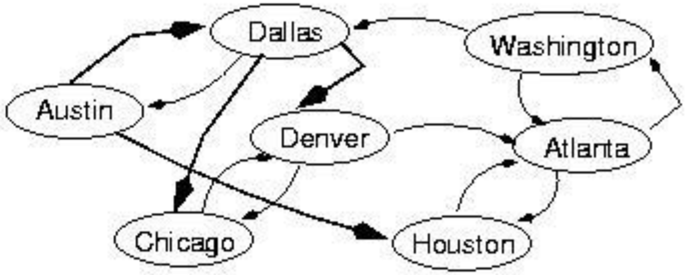
- Should we mark a vertex when it is enqueued or when it is dequeued ?


Dallas
Washington
Austin
Denver
Atlanta
Chicago
Houston
(initialization)
Austin
dequeue Austin
Dallas
Houston
dequeue Dallas
Houston
Chicago
Denver
dequeue Houston
Chicago
Denver
Atlanta



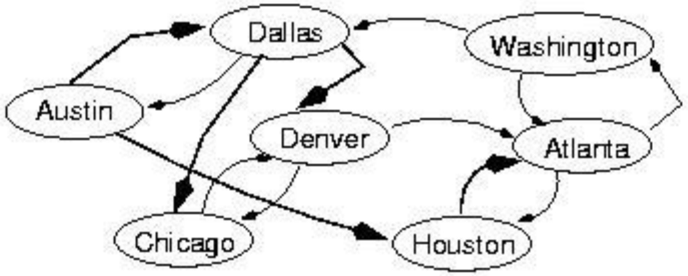
dequeue Chicago

	Denver	Atlanta	Denver



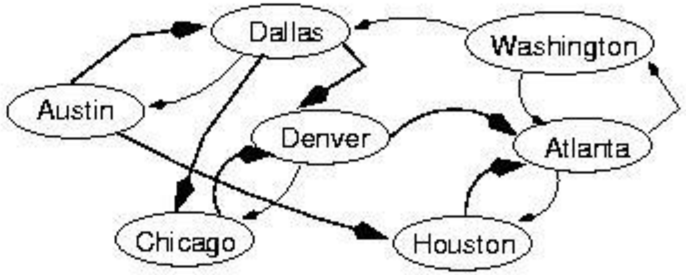
dequeue Denver

	Atlanta	Denver	Atlanta



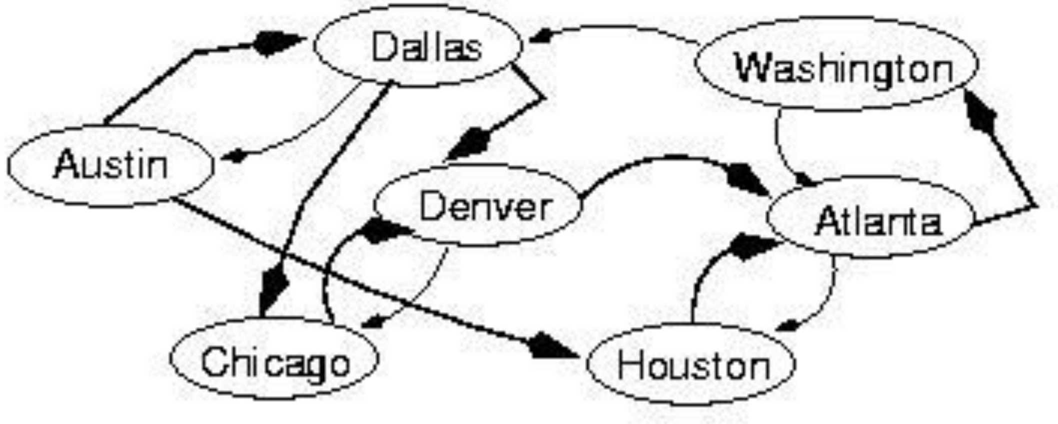
dequeue Atlanta

	Denver	Atlanta	Washington

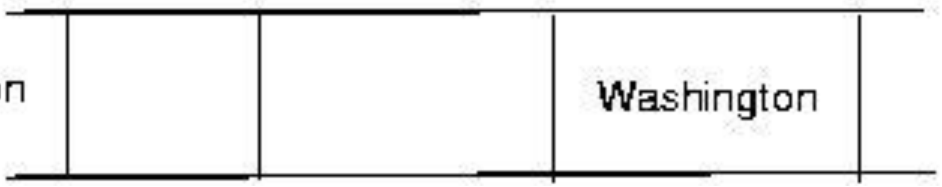


dequeue Denver,
next: Atlanta

	Washington	Washington



dequeue Washington



Single-source shortest-path problem

- There are multiple paths from a source vertex to a destination vertex
- *Shortest path*: the path whose total weight (i.e., sum of edge weights) is minimum
- Examples:
  - Austin->Houston->Atlanta->Washington: 1560 miles
  - Austin->Dallas->Denver->Atlanta->Washington: 2980 miles

Single-source shortest-path problem (cont.)

- Common algorithms: *Dijkstra's* algorithm, *Bellman-Ford* algorithm
- BFS can be used to solve the shortest graph problem when the graph is **weightless** or all the weights are the same

(mark vertices before Enqueue)

Thank You