CS 304 Lecture 10

Priority queues, heaps, and graphs

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

Northeastern Illinois University

Chicago, Illinois, 60625

22 November 2016

Priority queues

- A **priority queue** is an abstract data type with an interesting accessing protocol - only the highest-priority element can be accessed.
- Priority queues are useful for any application that involves processing items by priority.
- Methods of priority queues include:

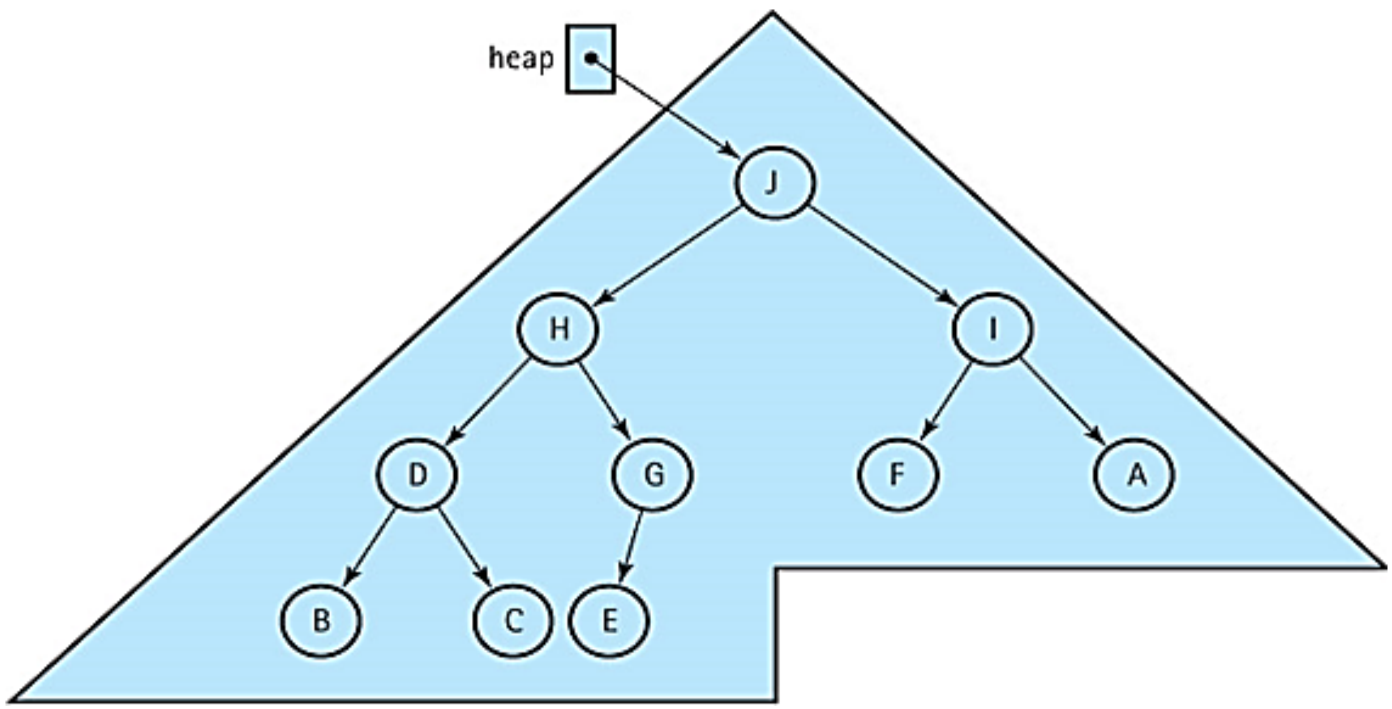
```
boolean isEmpty();// Checks if this priority queue is empty
boolean isFull(); // Checks if this priority queue is full
void enqueue(T element);
// Throws PriQOverflowException if this priority queue is
// full; otherwise, adds element to this priority queue.
T dequeue();
// Throws PriQUnderflowException if this priority queue is
// empty; otherwise, removes element with highest priority
// from this priority queue and returns it.
```

Priority queues

- There are many ways to implement a priority queue:
  - **An unsorted list** - Dequeuing would require searching through the entire list.
  - **An array-based sorted list** - Enqueuing is expensive.
  - **A reference-based sorted list** - Enqueuing again is $O(N)$.
  - **A binary search tree** - On average, $O(\log_2 N)$ steps for both `enqueue` and `dequeue`.
  - **A heap** - guarantees $O(\log_2 N)$ steps, even in the worst case.

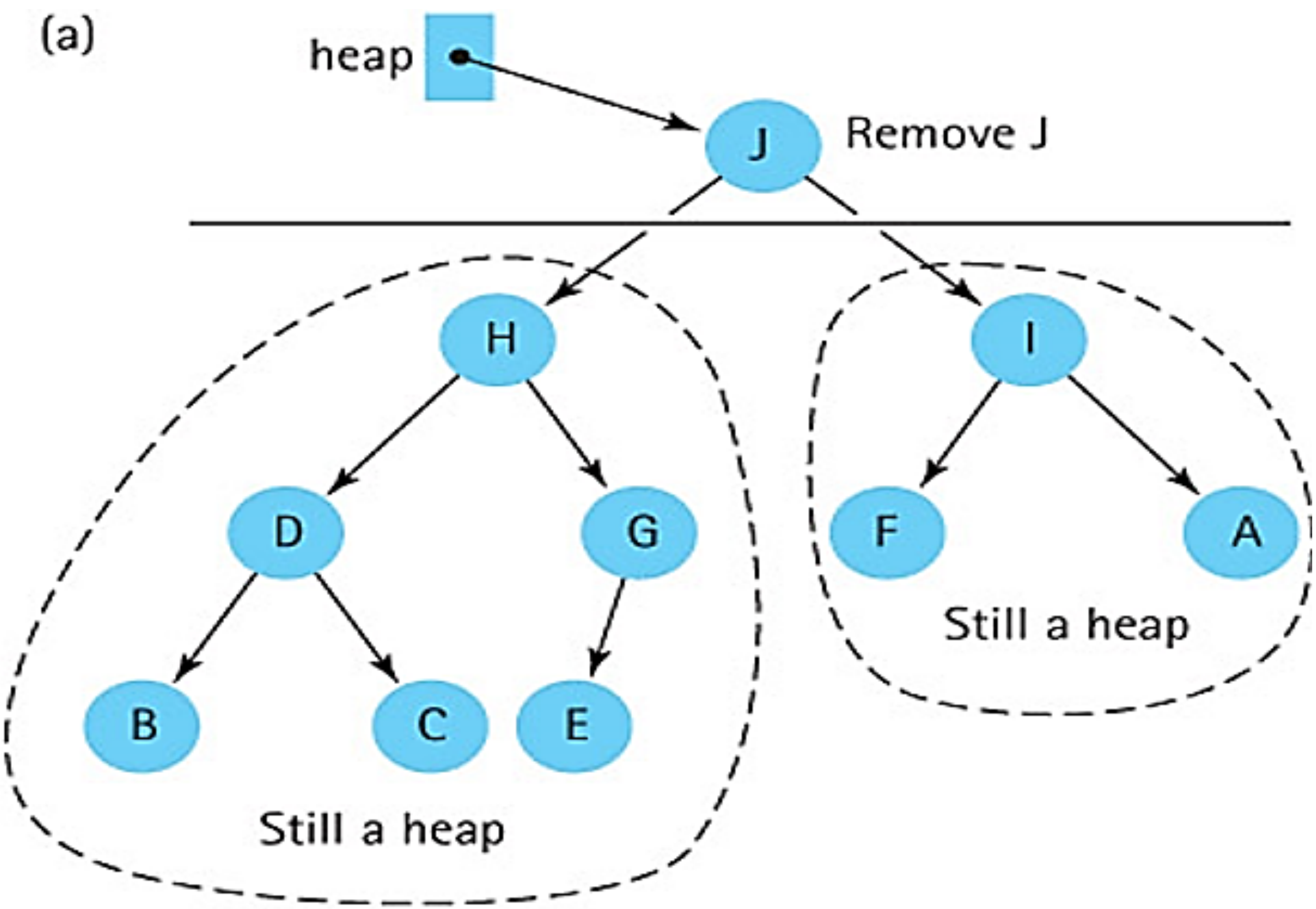
Heaps

- **Heap** - An implementation of a *priority queue* based on a complete binary tree which satisfies two properties:
  - The shape property: the tree must be a complete binary tree.
  - The order property: for every node in the tree, the value stored in that node is greater than or equal to the value in each of its children.



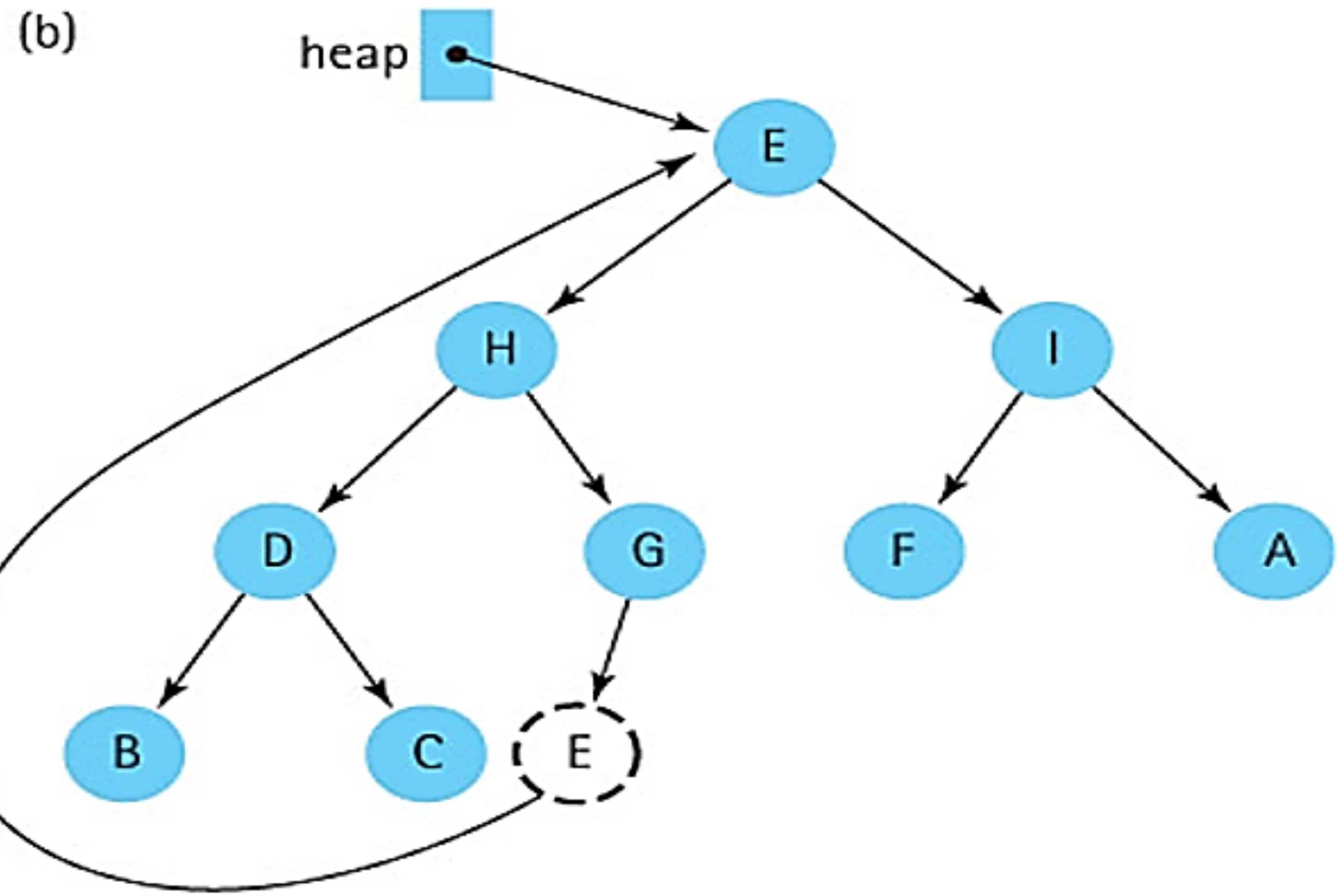
Heaps

- The **`dequeue`** operation



Heaps

- The **`dequeue`** operation



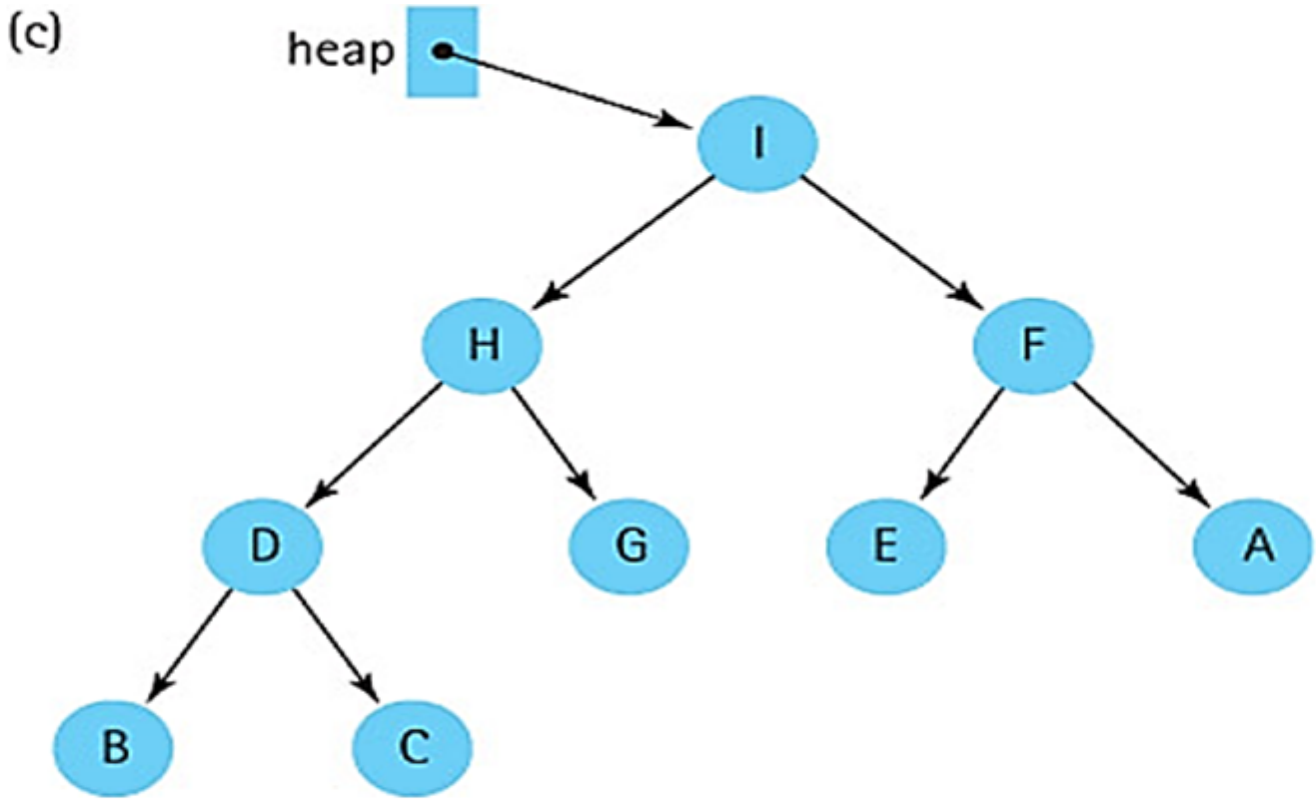
Heaps

- The **`dequeue`** operation

<u>`reheapDown (element)`</u>

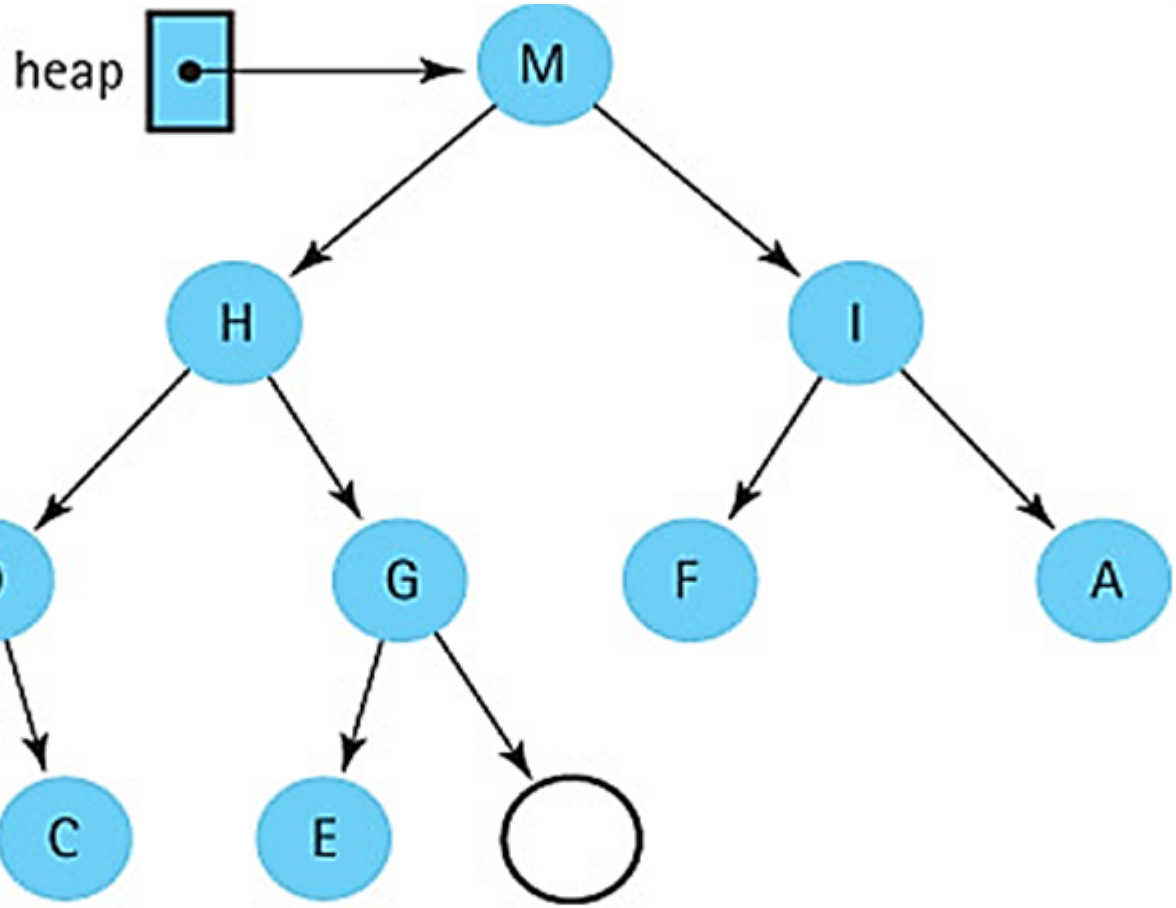
`Effect: Adds element to the heap.`

`Precondition: The root of the tree is empty.`



Heaps

- The **`enqueue`** operation



(a) Add K

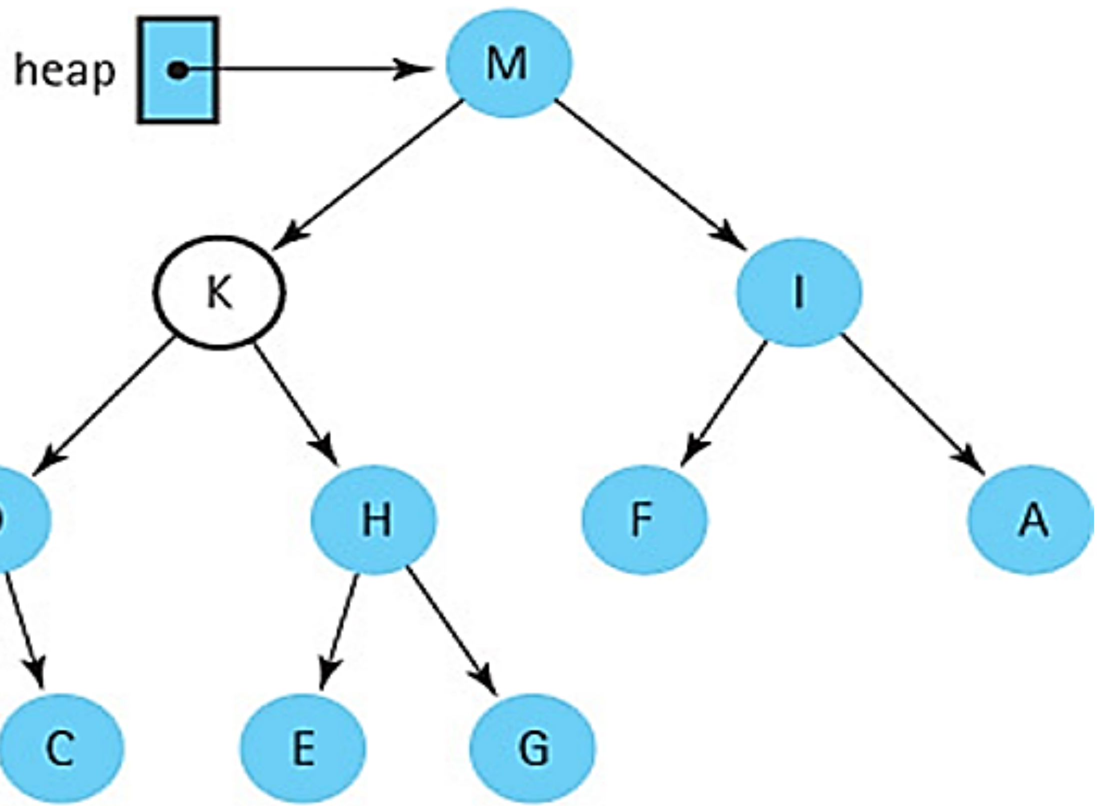
Heaps

- The **enqueue** operation

<u>reheapUp (element)</u>

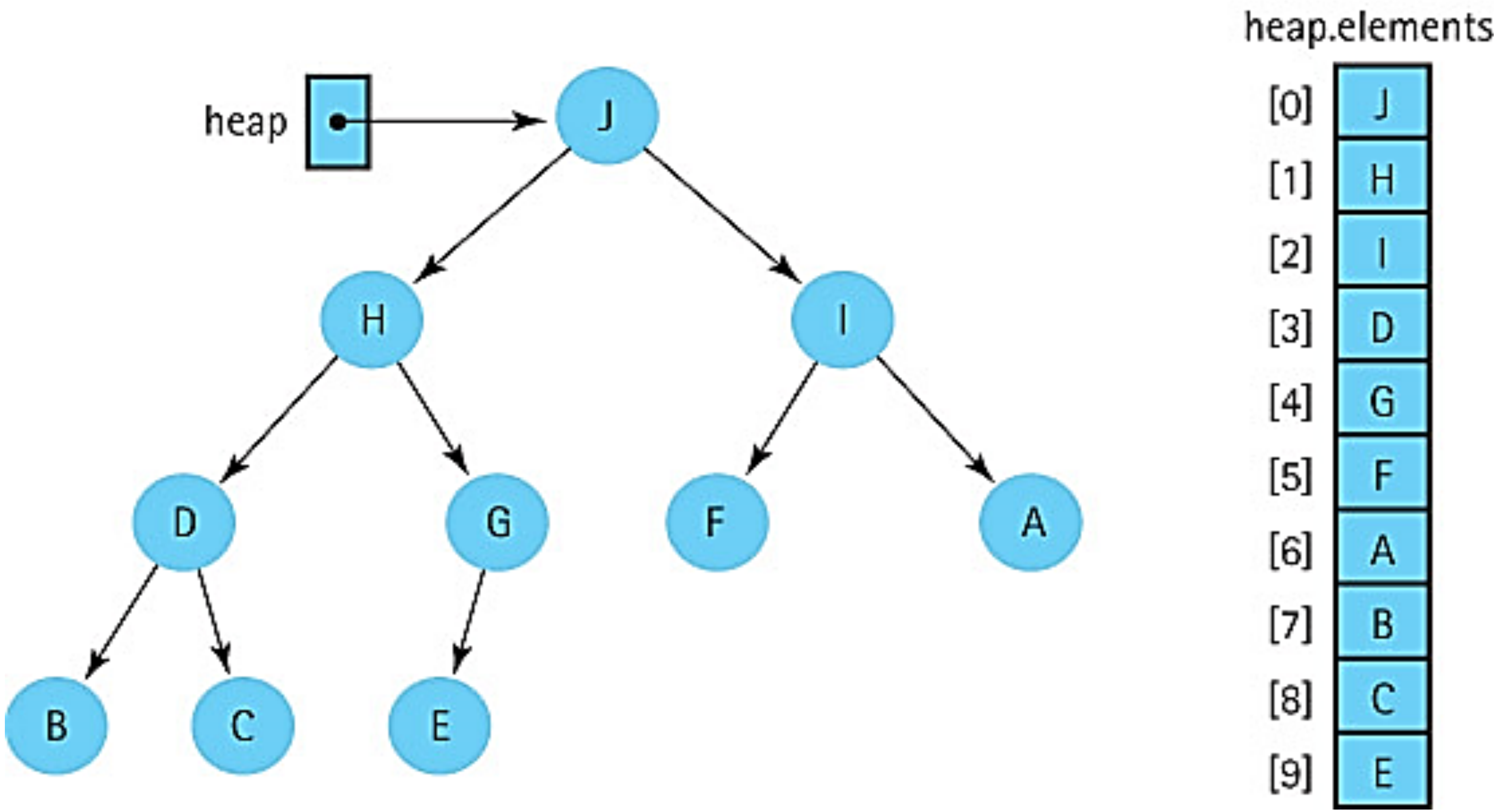
Effect: Adds element to the heap.

Precondition: The last index position of the tree is empty.



(b) reheapUp

Heap implementation

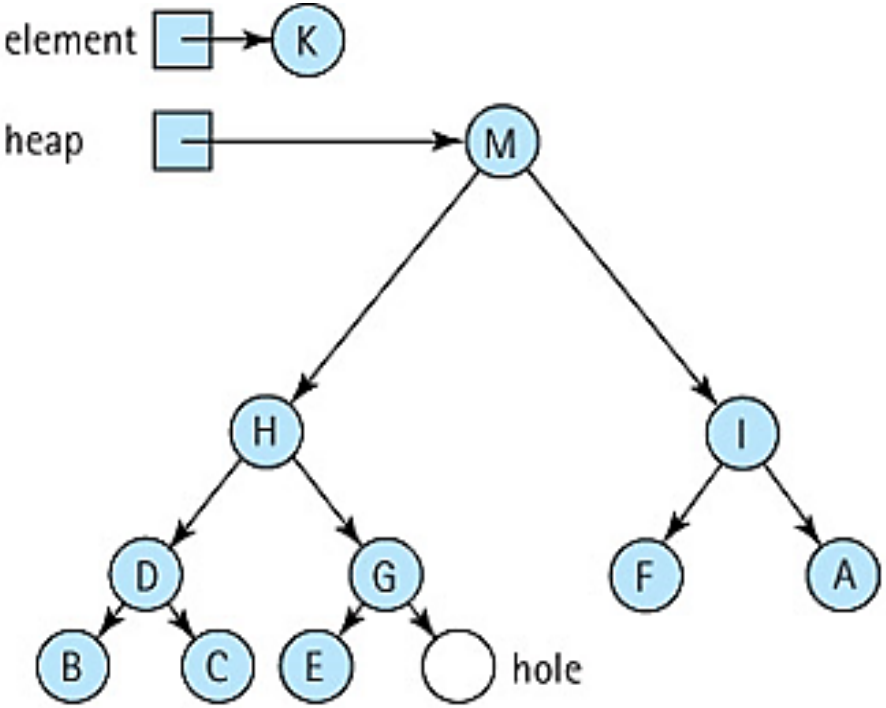


The **enqueue** method

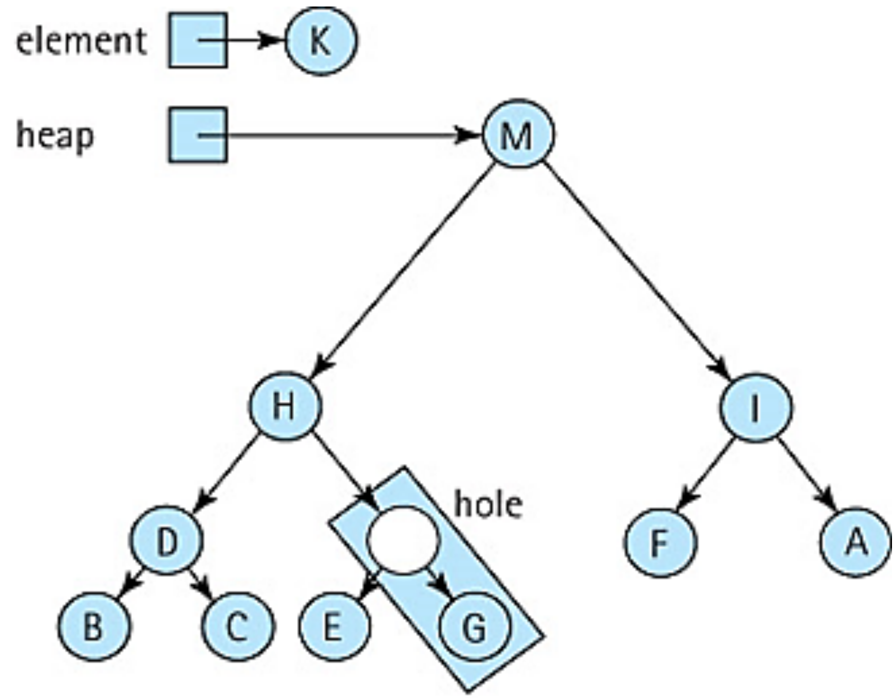
```
public void enqueue(T element) throws PriQOverflowException
// Throws PriQOverflowException if this priority queue is full;
// otherwise, adds element to this priority queue.
{
  if (lastIndex == maxIndex)
    throw new PriQOverflowException("Priority queue is full");
  else
  {
    lastIndex++;
    elements.add(lastIndex, element);
    reheapUp(element);
  }
}
```

The `enqueue` method

The `reheapUp` algorithm



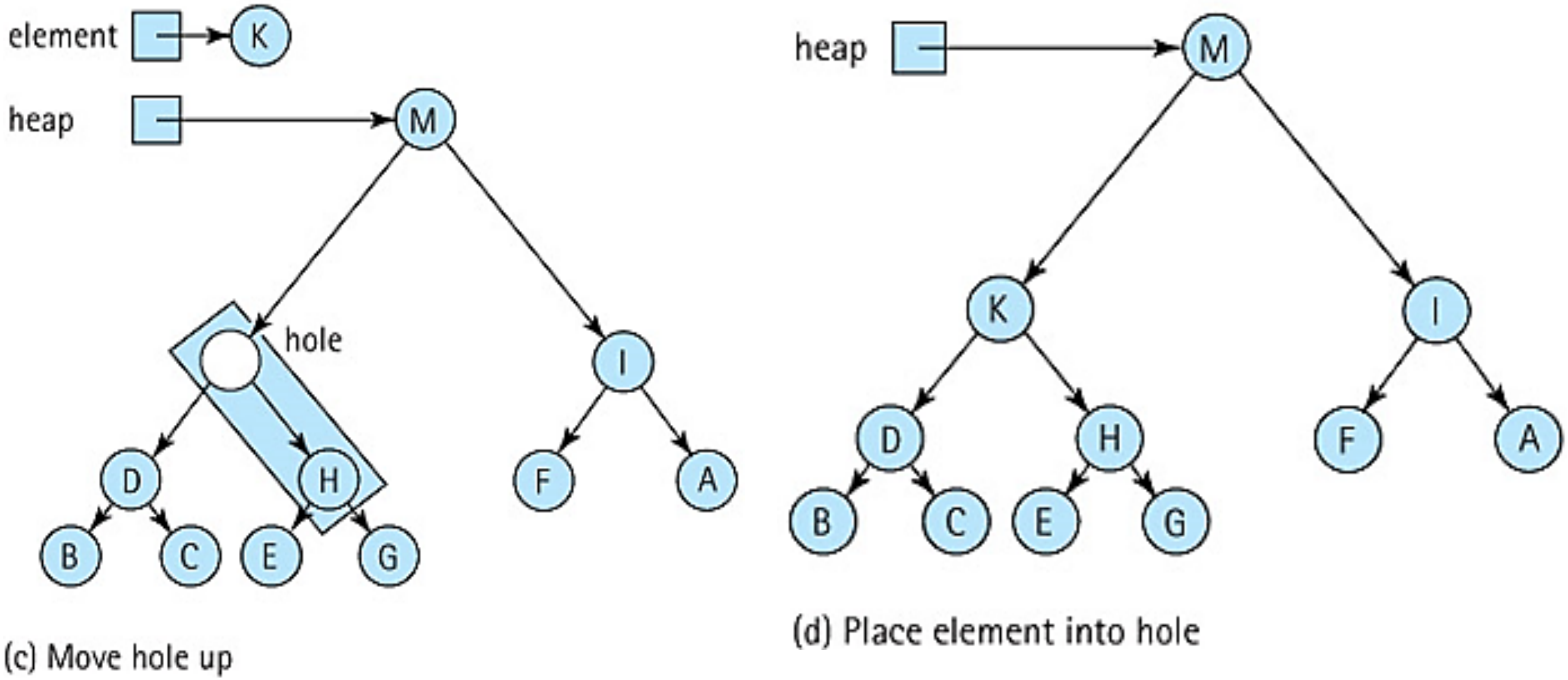
(a) Add K



(b) Move hole up

The `enqueue` method

The `reheapUp` algorithm



The `enqueue` method

- In a binary tree, the following relationships hold for an element at position **`index`**:
  - If the element is not the root, its parent is at position **`(index - 1) / 2`**.
  - If the element has a left child, the child is at position **`(index * 2) + 1`**.
  - If the element has a right child, the child is at position **`(index * 2) + 2`**.

The **enqueue** method

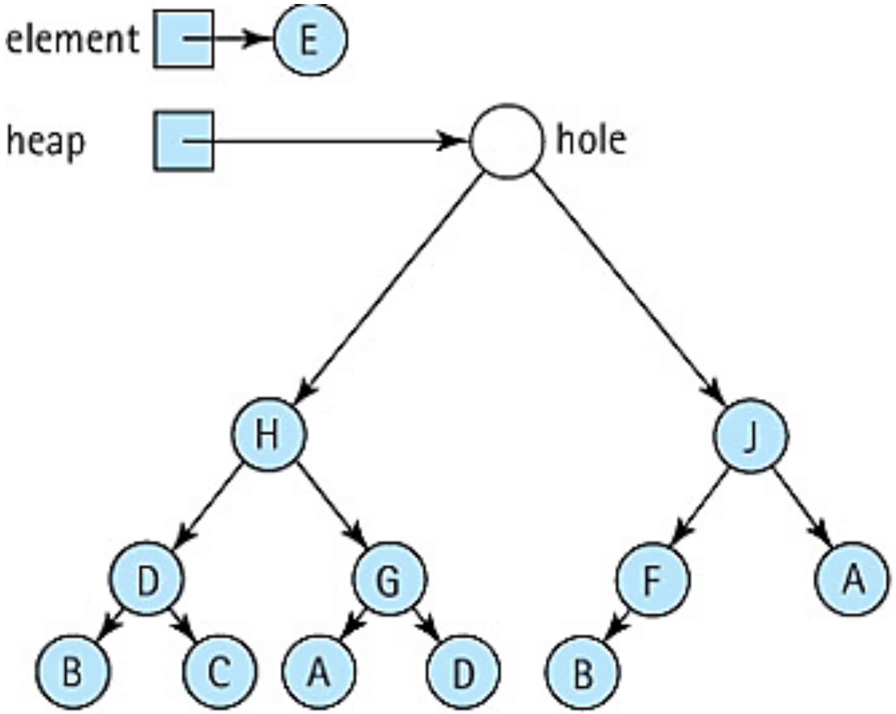
```
private void reheapUp(T element)
// Current lastIndex position is empty.
// Inserts element into the tree and ensures shape and order
// properties.
{
  int hole = lastIndex;

  while ((hole > 0) // hole is not root
         &&
      (element.compareTo(elements.get((hole - 1) / 2]) > 0))
      // element > hole's parent
  {
    elements.set(hole, elements.get((hole - 1) / 2));
    // move hole's parent down
    hole = (hole - 1) / 2;
    // move hole up
  }

  elements.set(hole, element); // place element into final hole
}
```

The **dequeue** method

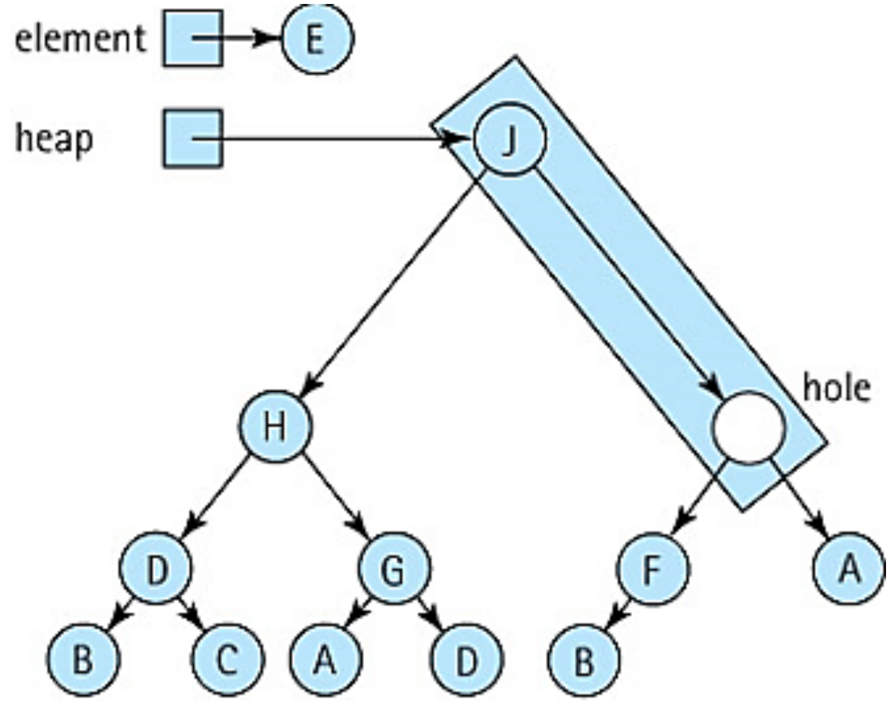
```
public T dequeue() throws PriQUnderflowException
// Throws PriQUnderflowException if this priority queue is empty;
// otherwise, removes element with highest priority from this
// priority queue and returns it.
{
  T hold;      // element to be dequeued and returned
  T toMove;    // element to move down heap
  if (lastIndex == -1)
    throw new PriQUnderflowException("Priority queue is empty");
  else
  {
    hold = elements.get(0); // remember element to be returned
    toMove = elements.remove(lastIndex);// element to reheap down
    lastIndex--;            // decrease priority queue size
    if (lastIndex != -1)
       reheapDown(toMove);  // restore heap properties
    return hold;            // return largest element
  }
}
```

The **`dequeue`** method

The **`reheapDown`** algorithm



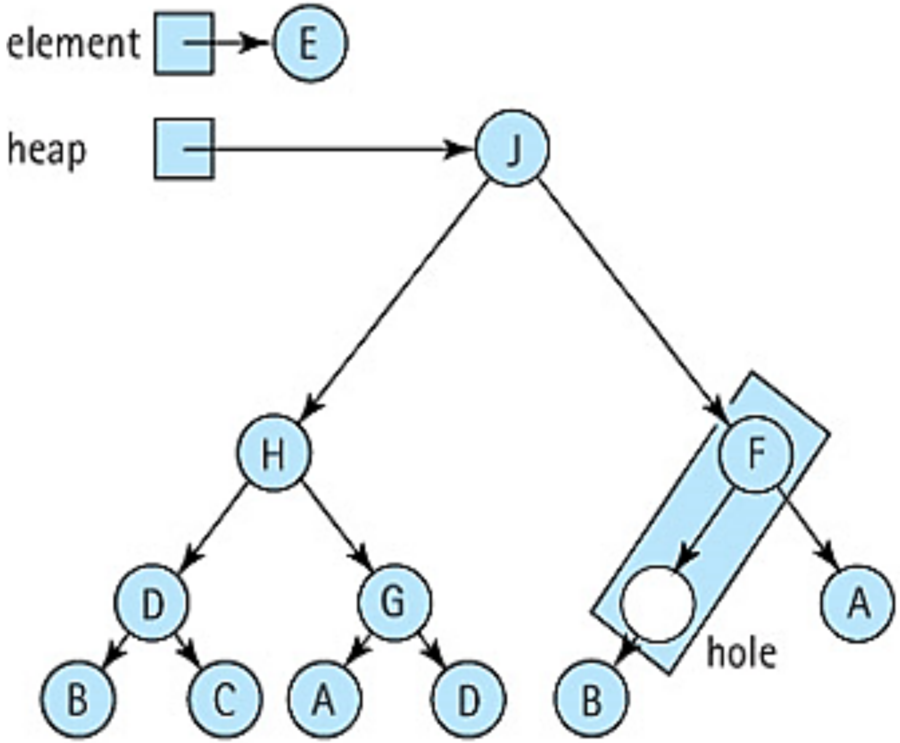
(a) reheapDown (E);



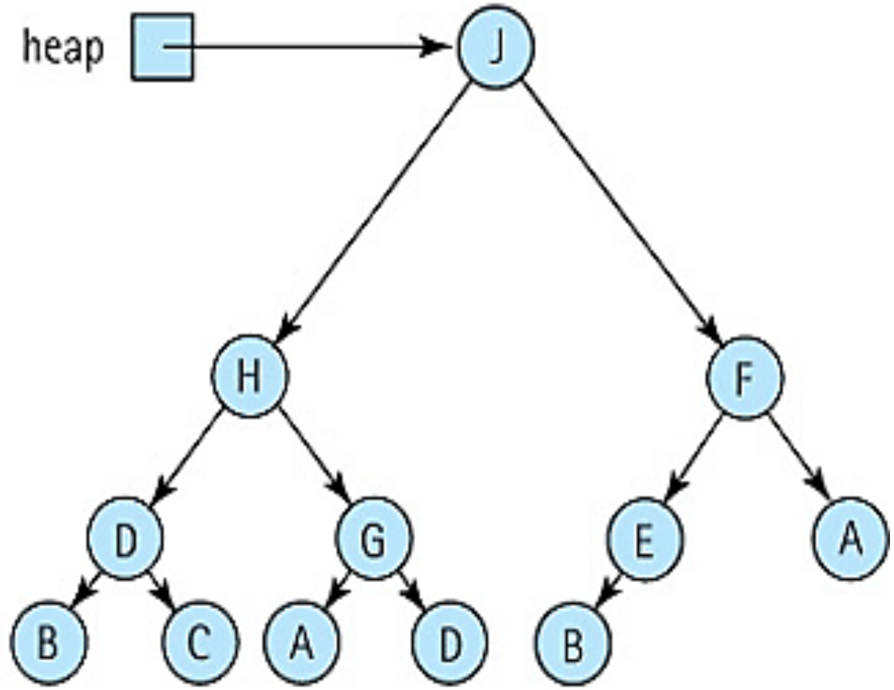
(b) Move hole down

The `dequeue` method

The `reheapDown` algorithm



(c) Move hole down

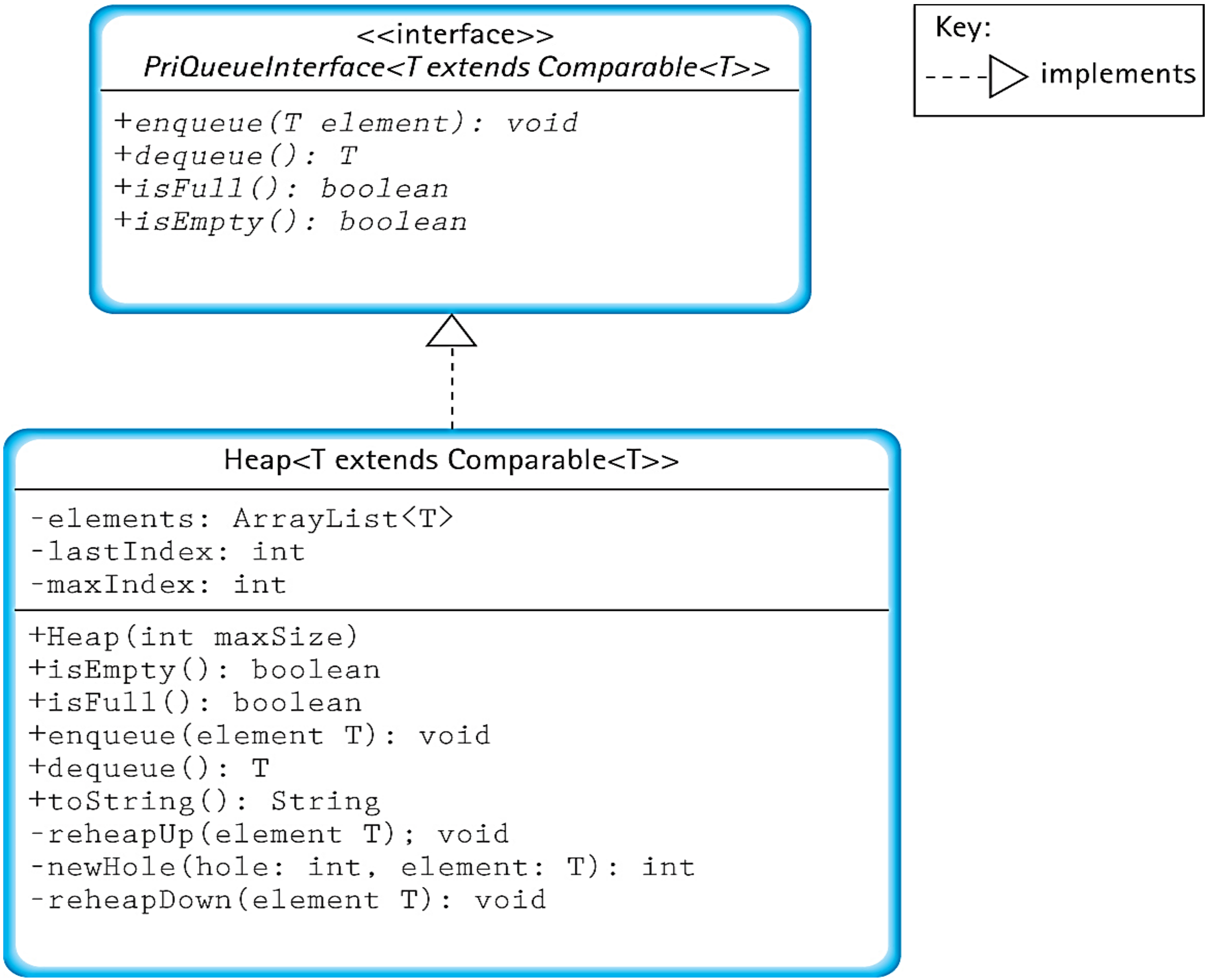


(d) Fill in final hole

The **dequeue** method

```
private void reheapDown(T element)
// Current root position is "empty";
// Inserts element into the tree and ensures shape and order
// properties.
{
  int hole = 0;        // current index of hole
  int newhole;         // index where hole should move to
  newhole = newHole(hole, element);   // find next hole
  while (newhole != hole)
  {
    elements.set(hole, elements.get(newhole)); // move element up
    hole = newhole;                            // move hole down
    newhole = newHole(hole, element);          // find next hole
  }
  elements.set(hole, element); // fill in the final hole
}
```

Heap implementation



Heaps vs other representations of priority queues

	enqueue	dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$

Heap sort

- We have learned several sorting algorithms: selection sort, bubble sort, insertion sort, merge sort and quick sort. Due to the properties of heaps, we can also use heaps to sort elements.
- The general approach of the heap sort is as follows:
  - Take the root (maximum) element off the heap, and put it into its place.
  - `reheap` the remaining elements. (This puts the next-largest element into the root position.)
  - Repeat until there are no more elements.
- For this to work we must first arrange the original array into a heap.

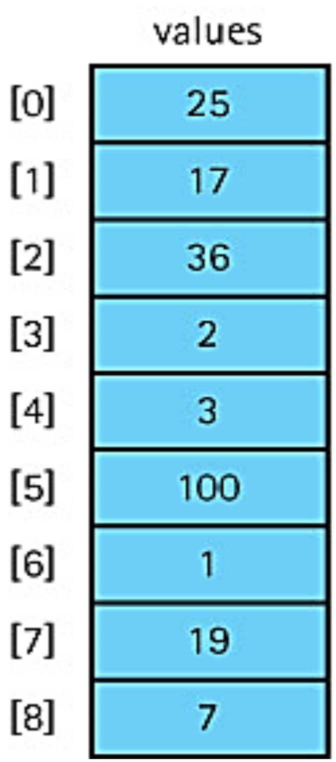
Heap sort

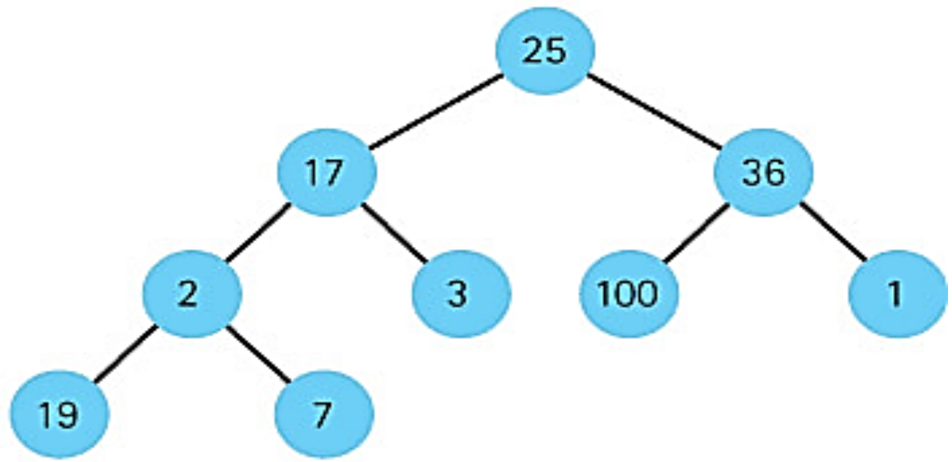
- Building a heap

```
buildHeap
for index going from first nonleaf node up to the root node
     reheapDown(values[index], index, SIZE - 1)
```

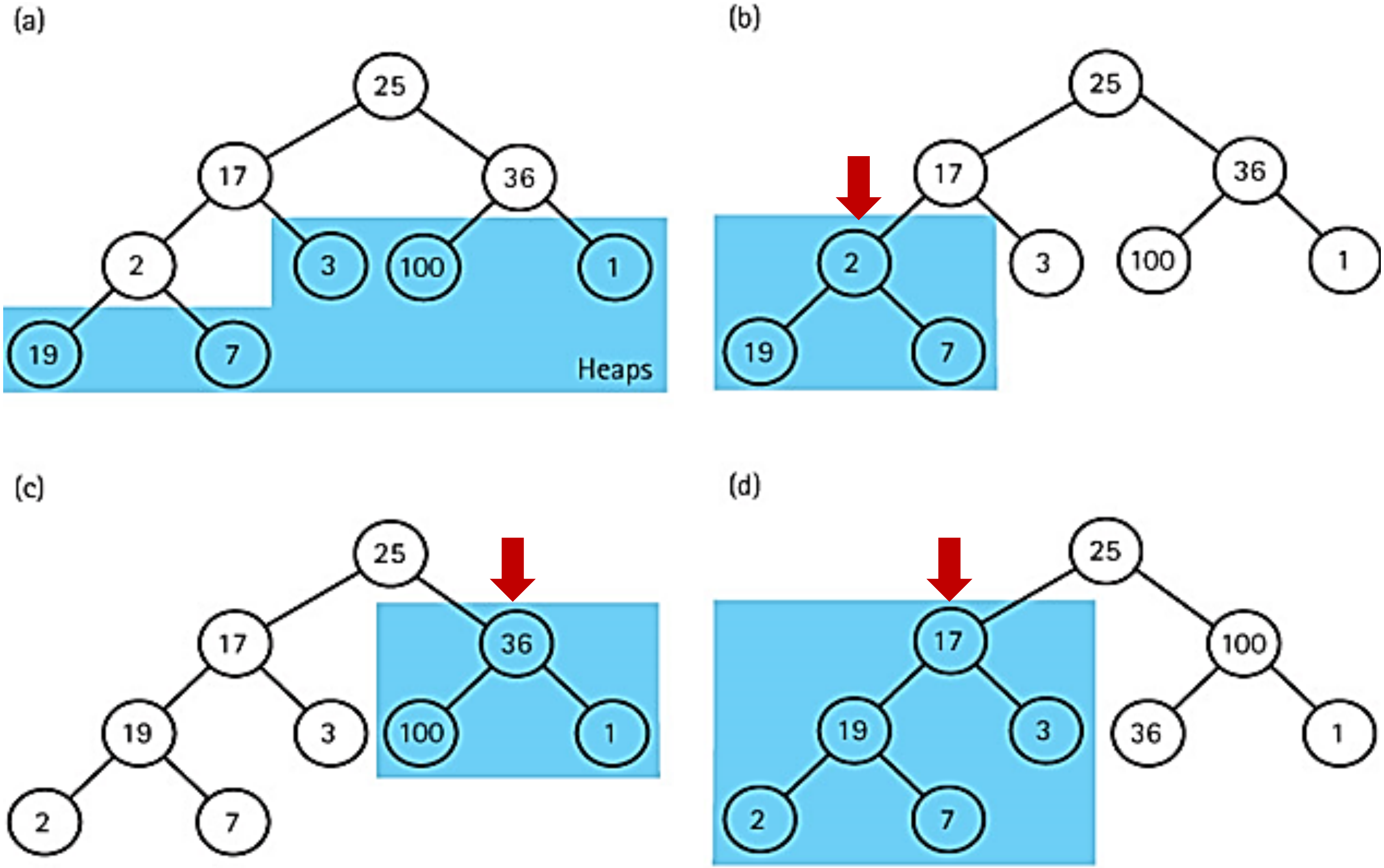
the end index of the heap

the index of the root of the subtree that is to be made into a heap





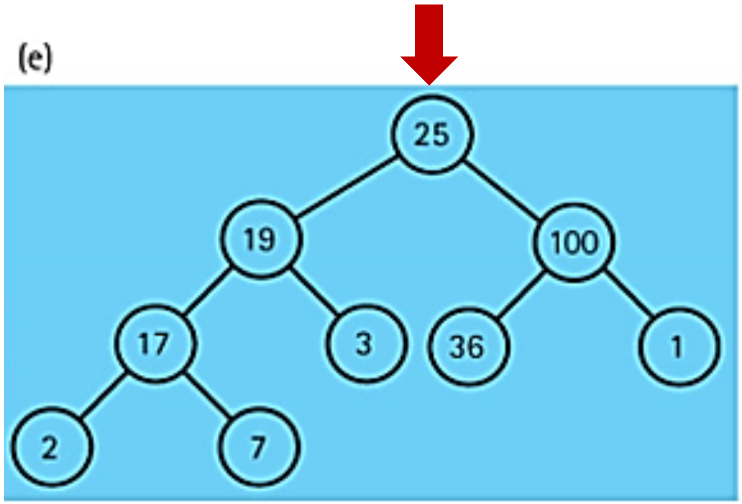
Heap sort



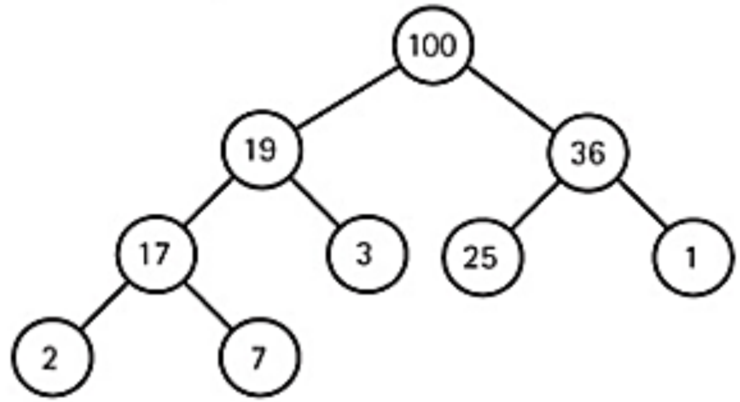
(a)
25
17
36
2
3
100
1
19
7
Heaps
(b)
25
17
36
2
3
100
1
19
7
(c)
25
17
36
19
3
100
1
2
7
(d)
25
17
100
19
3
36
1
2
7

Heap sort

(e)



(f) Tree now represents a heap



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Original values	25	17	36	2	3	100	1	19	7
After reheapDown index = 3	25	17	36	19	3	100	1	2	7
After index = 2	25	17	100	19	3	36	1	2	7
After index = 1	25	19	100	17	3	36	1	2	7
After index = 0	100	19	36	17	3	25	1	2	7

Tree is a heap.

Heap sort

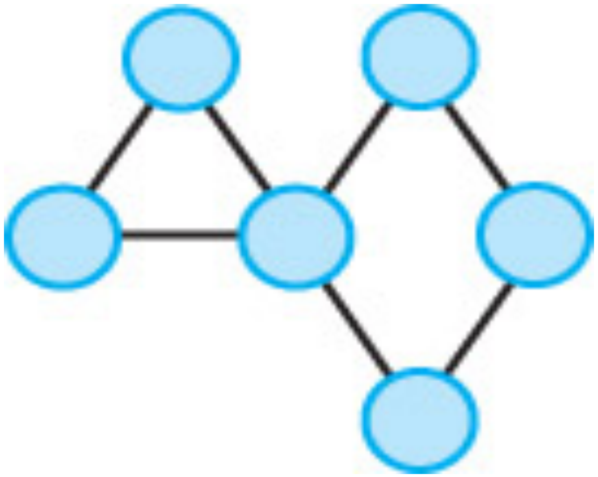
```
Sort Nodes
for index going from last node up to next-to-root node
    Swap data in root node with values[index]
    reheapDown(values[0], 0, index-1)

static void heapSort()
// Post: The elements in the array values are sorted by key
{
  int index;
  // Convert the array of values into a heap
  for (index = SIZE / 2 - 1; index >= 0; index--)
    reheapDown(values[index], index, SIZE - 1);

  // Sort the array
  for (index = SIZE - 1; index >=1; index--)
  {
    swap(0, index);
    reheapDown(values[0], 0, index - 1);
  }
}
```

Heap sort

- The time complexity of heap sort is $O(N\log_2 N)$.
  - For small arrays, **`heapSort`** is not very efficient because of all the "overhead".
  - For large arrays, however, **`heapSort`** is very efficient.
- Unlike quick sort, heap sort's efficiency is not affected by the initial order of the elements.
- Heap sort is also efficient in terms of space – it only requires constant extra space.
- Heap sort is an elegant, fast, robust, space efficient algorithm!

Graphs

- **Graph** - A data structure that consists of a set of nodes and a set of edges that relate the nodes to each other.
- **Vertex** - A node in a graph.
- **Edge (arc)** - A pair of vertices representing a connection between two nodes in a graph.
- **Undirected graph** - A graph in which the edges have no direction.
- **Directed graph (digraph)** - A graph in which each edge is directed from one vertex to another (or the same) vertex.

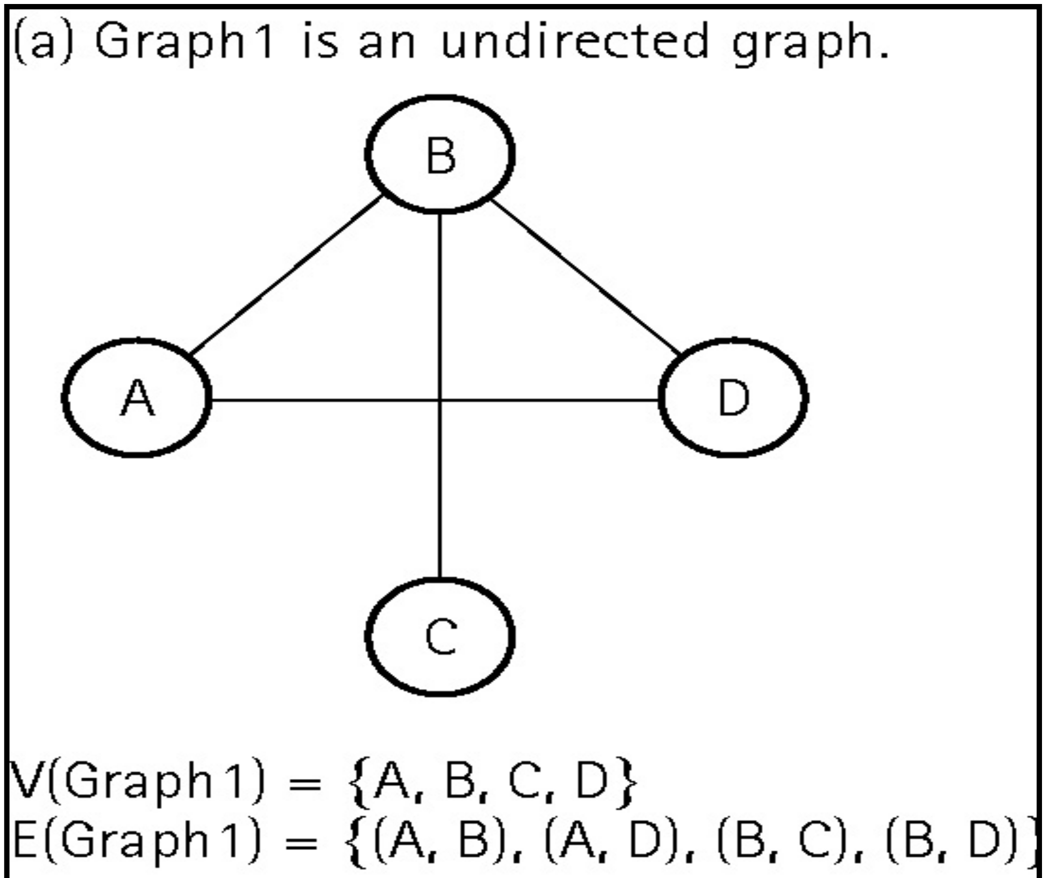


Graphs

A graph G is defined as follows:

$G = (V, E)$ where $V(G)$ is a finite, nonempty set of vertices;

$E(G)$ is a set of edges (written as pairs of vertices).



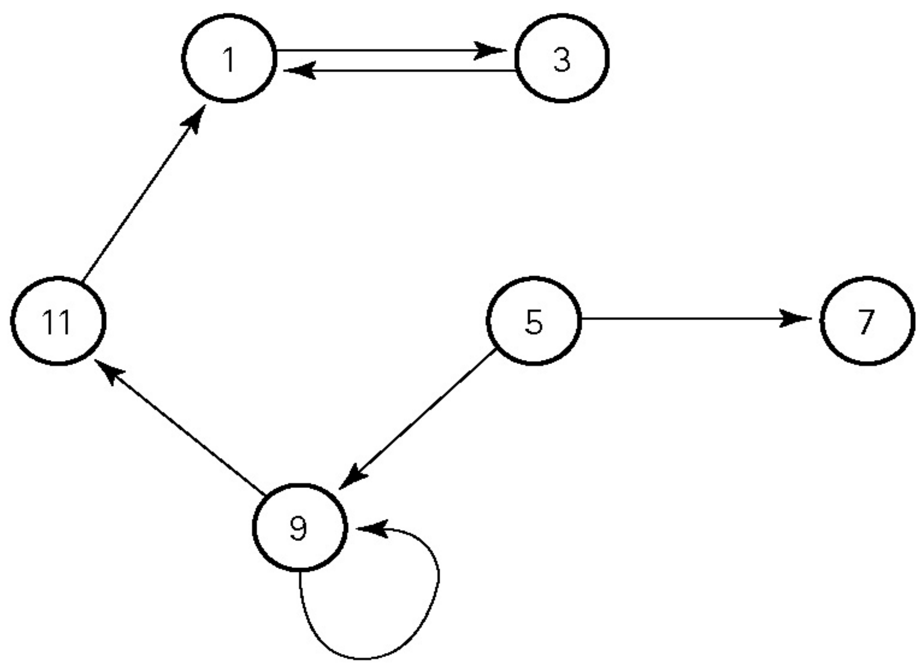
Graphs

A graph G is defined as follows:

$G = (V, E)$ where $V(G)$ is a finite, nonempty set of vertices;

$E(G)$ is a set of edges (written as pairs of vertices).

(b) Graph2 is a directed graph.



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

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

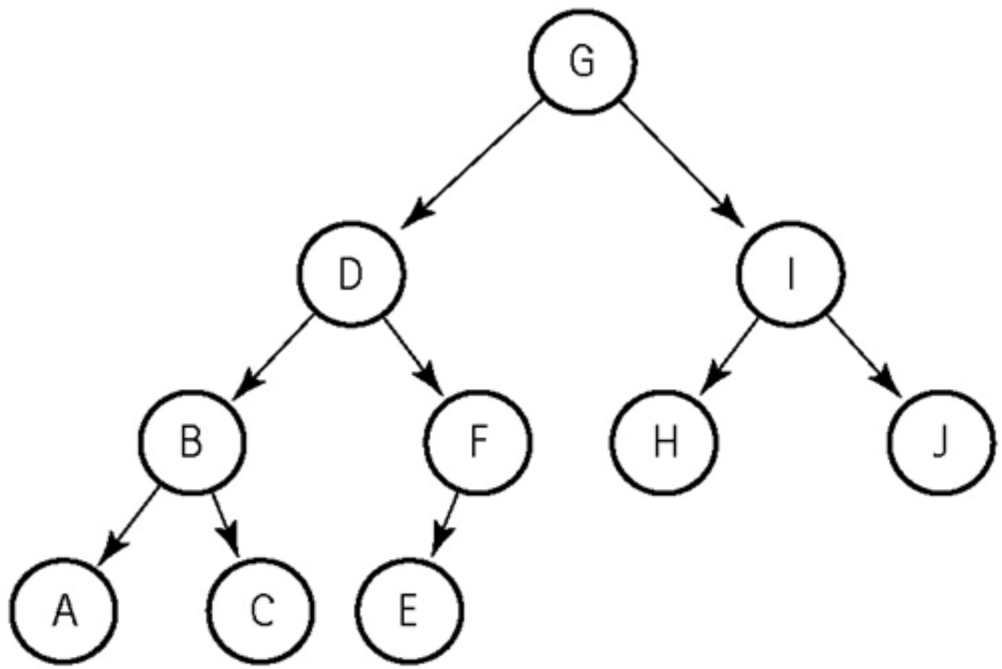
Graphs

A graph G is defined as follows:

$G = (V, E)$ where $V(G)$ is a finite, nonempty set of vertices;

$E(G)$ is a set of edges (written as pairs of vertices).

(c) Graph3 is a directed graph.



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

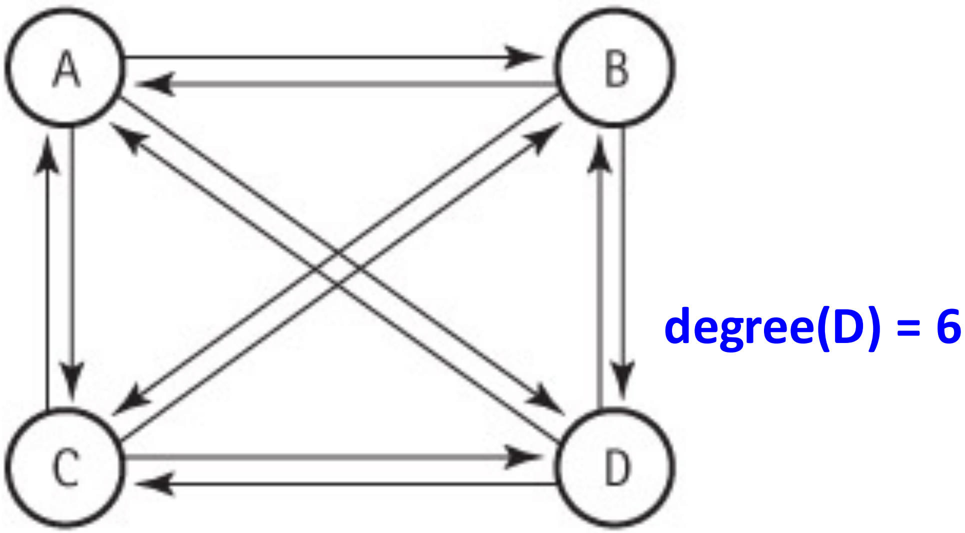
E(Graph3) = {(G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

Graphs

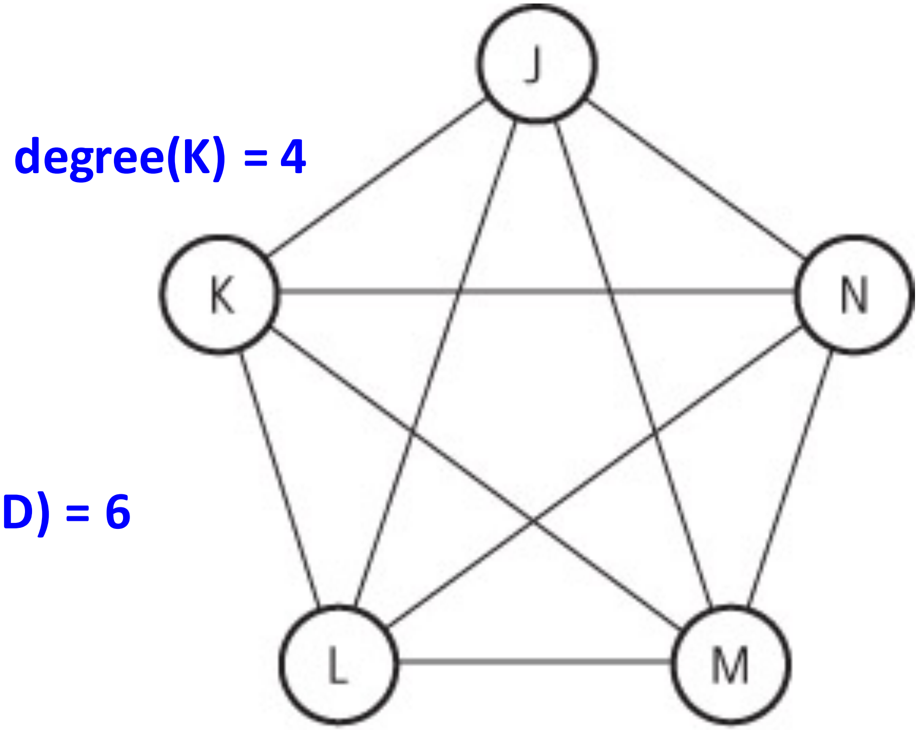
- **Adjacent vertices** - Two vertices in a graph that are connected by an edge.
- **Vertex degree** - The number of edges connected to this vertex.
- **Path**: A sequence of vertices that connects two nodes in a graph.
- **Complete graph** - A graph in which every vertex is directly connected to every other vertex.
- **Weighted graph** - A graph in which each edge carries a value.

Graphs

Complete graphs



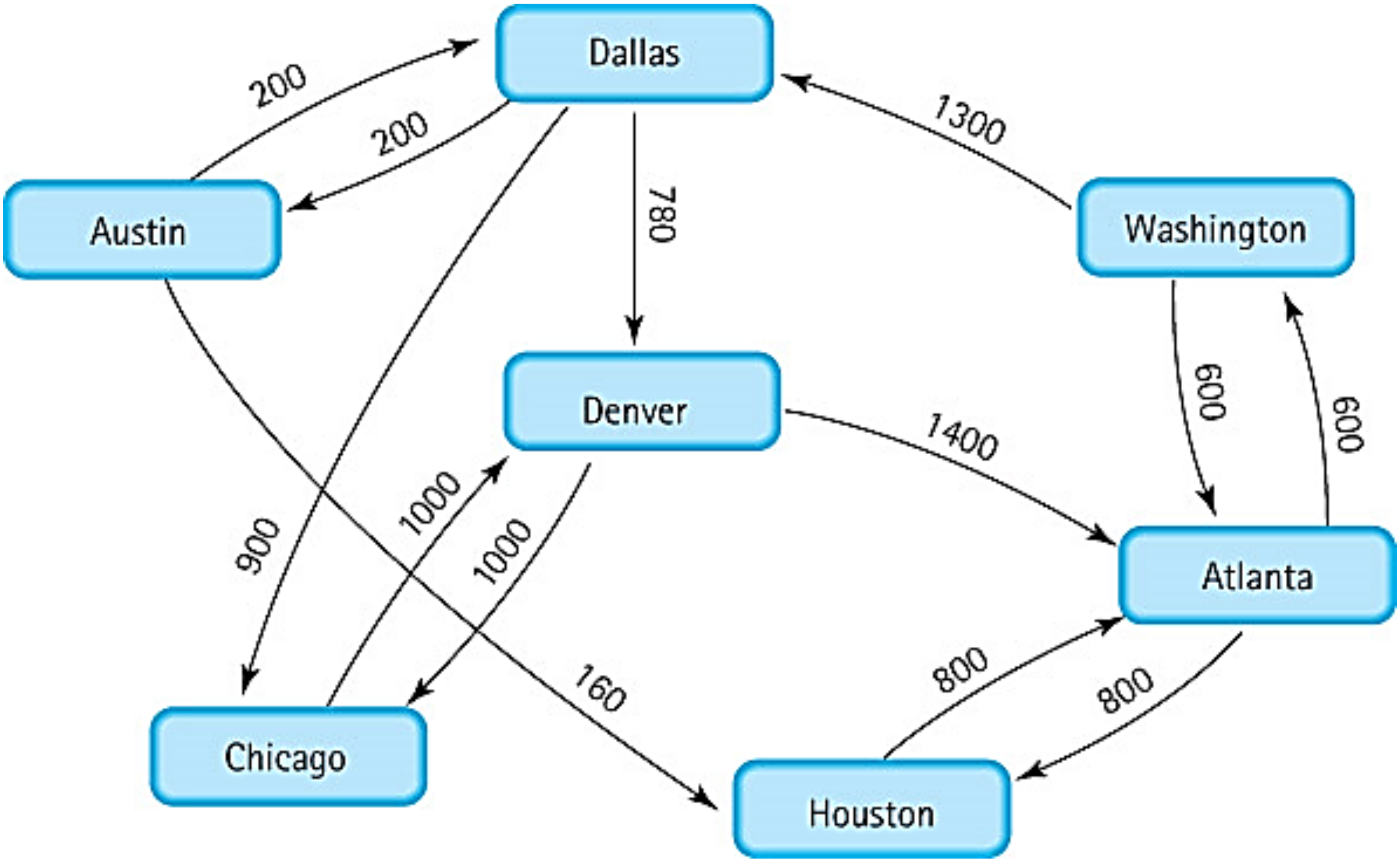
(a) Complete directed graph.



(b) Complete undirected graph.

Graphs

A weighted graph



Graphs

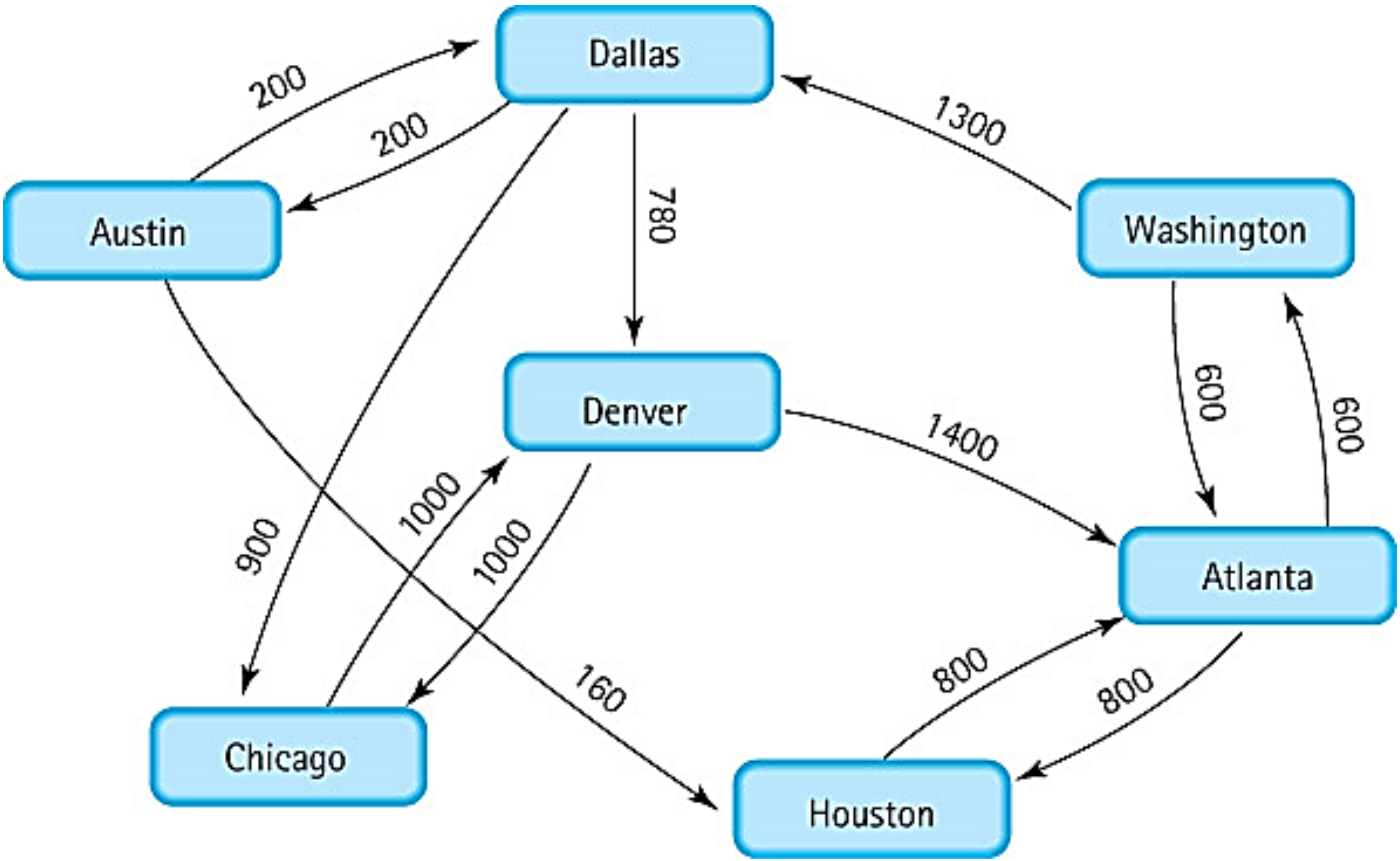
- Methods of graphs include:

```
isEmpty(), isFull(), addVertex(T vertex) , hasVertex(T vertex)
addEdge(T fromVertex, T toVertex, int weight)
weightIs(T fromVertex, T toVertex)

UnboundedQueueInterface<T> getToVertices(T vertex);
// Returns a queue of the vertices that are adjacent from
// vertex.

clearMarks(), markVertex(T vertex), isMarked(T vertex)

T getUnmarked();
// Returns an unmarked vertex if any exist; otherwise, returns
// null.
```

Array-based implementation for graphs

- **Adjacency matrix** - For a graph with N nodes, an N by N table that shows the existence (and weights) of all edges in the graph.
- With this approach a graph consists of
  - an integer variable `numVertices`,
  - a one-dimensional array `vertices`,
  - a two-dimensional array `edges` (the adjacency matrix).

Array-based implementation for graphs



Array-based implementation for graphs

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)

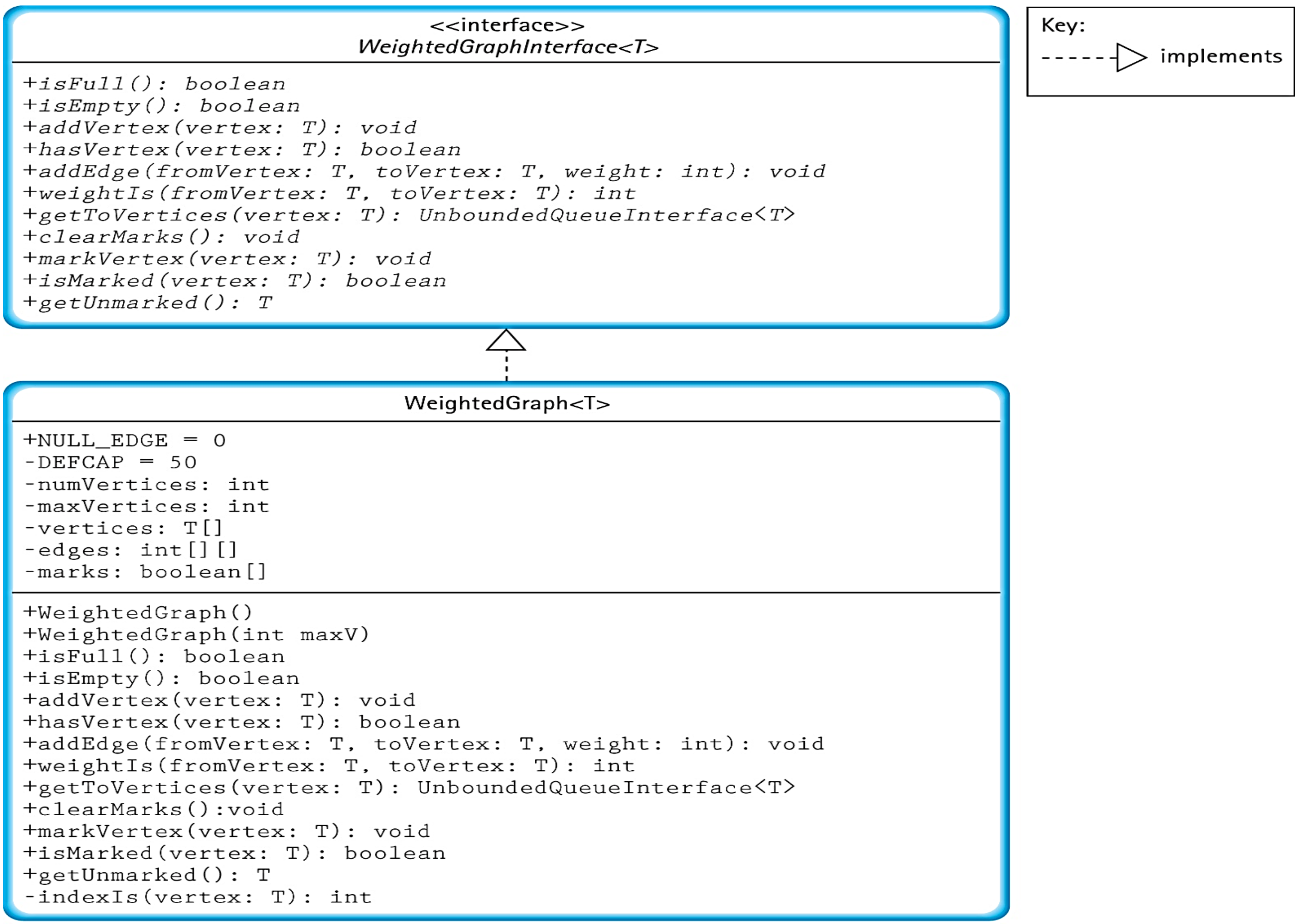
Array-based implementation for graphs

```
public class WeightedGraph<T> implements
                                  WeightedGraphInterface<T>
{
  public static final int NULL_EDGE = 0;
  private static final int DEFCAP = 50;  // default capacity
  private int numVertices;
  private int maxVertices;
  private T[] vertices;
  private int[][] edges;
  private boolean[] marks;  // marks[i] is mark for vertices[i]

  public WeightedGraph()
  // Instantiates a graph with capacity DEFCAP vertices.
  {
    numVertices = 0;
    maxVertices = DEFCAP;
    vertices = (T[]) new Object[DEFCAP];
    marks = new boolean[DEFCAP];
    edges = new int[DEFCAP][DEFCAP];
  }
```

Array-based implementation for graphs

```
public WeightedGraph(int maxV)
// Instantiates a graph with capacity maxV.
{
  numVertices = 0;
  maxVertices = maxV;
  vertices = (T[]) new Object[maxV];
  marks = new boolean[maxV];
  edges = new int[maxV][maxV];
}

public void addVertex(T vertex)
// Adds vertex to this graph.
{
  vertices[numVertices] = vertex;
  for (int index = 0; index < numVertices; index++)
  {
    edges[numVertices][index] = NULL_EDGE;
    edges[index][numVertices] = NULL_EDGE;
  }
  numVertices++;
}
```

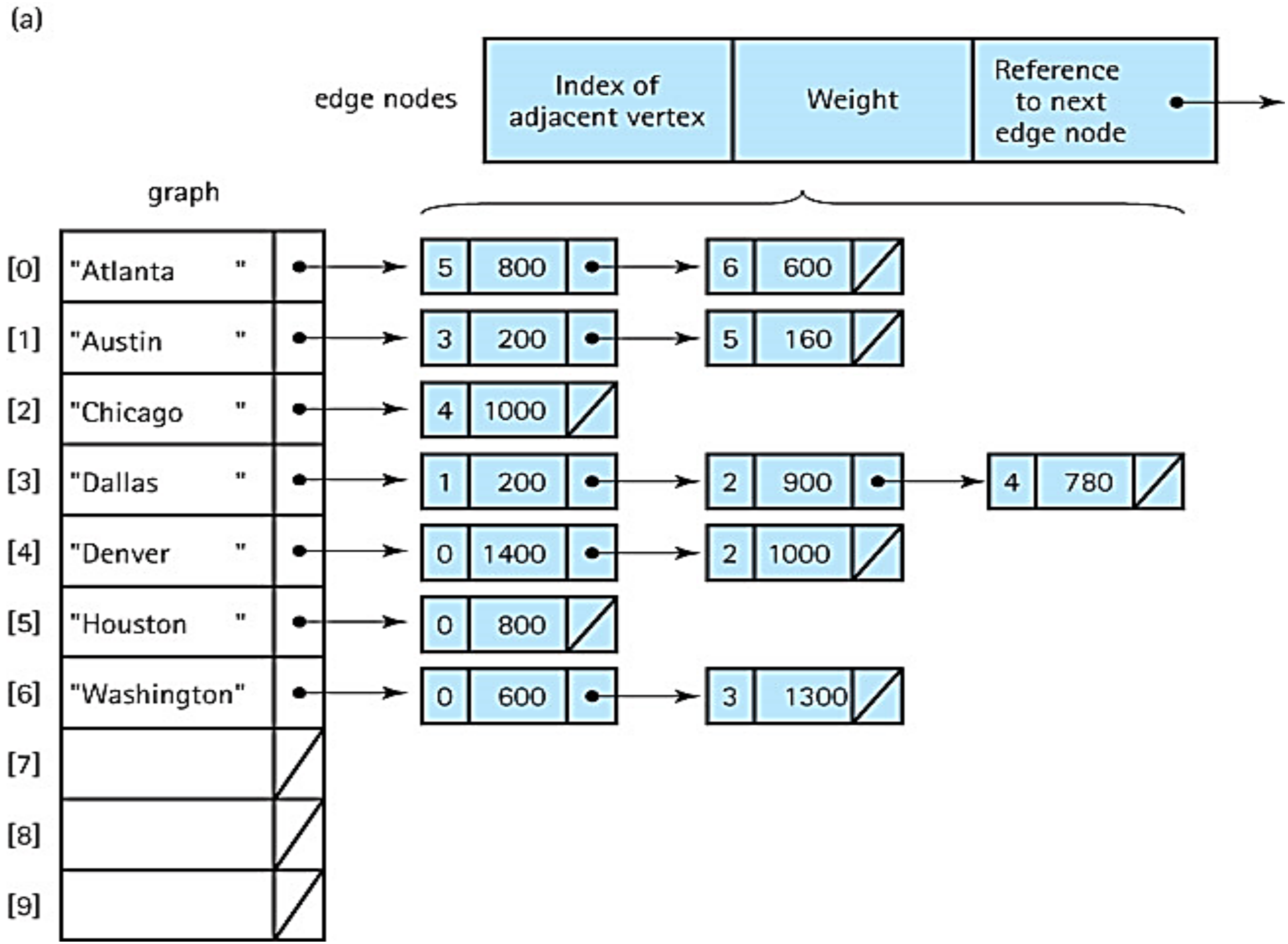
Array-based implementation for graphs



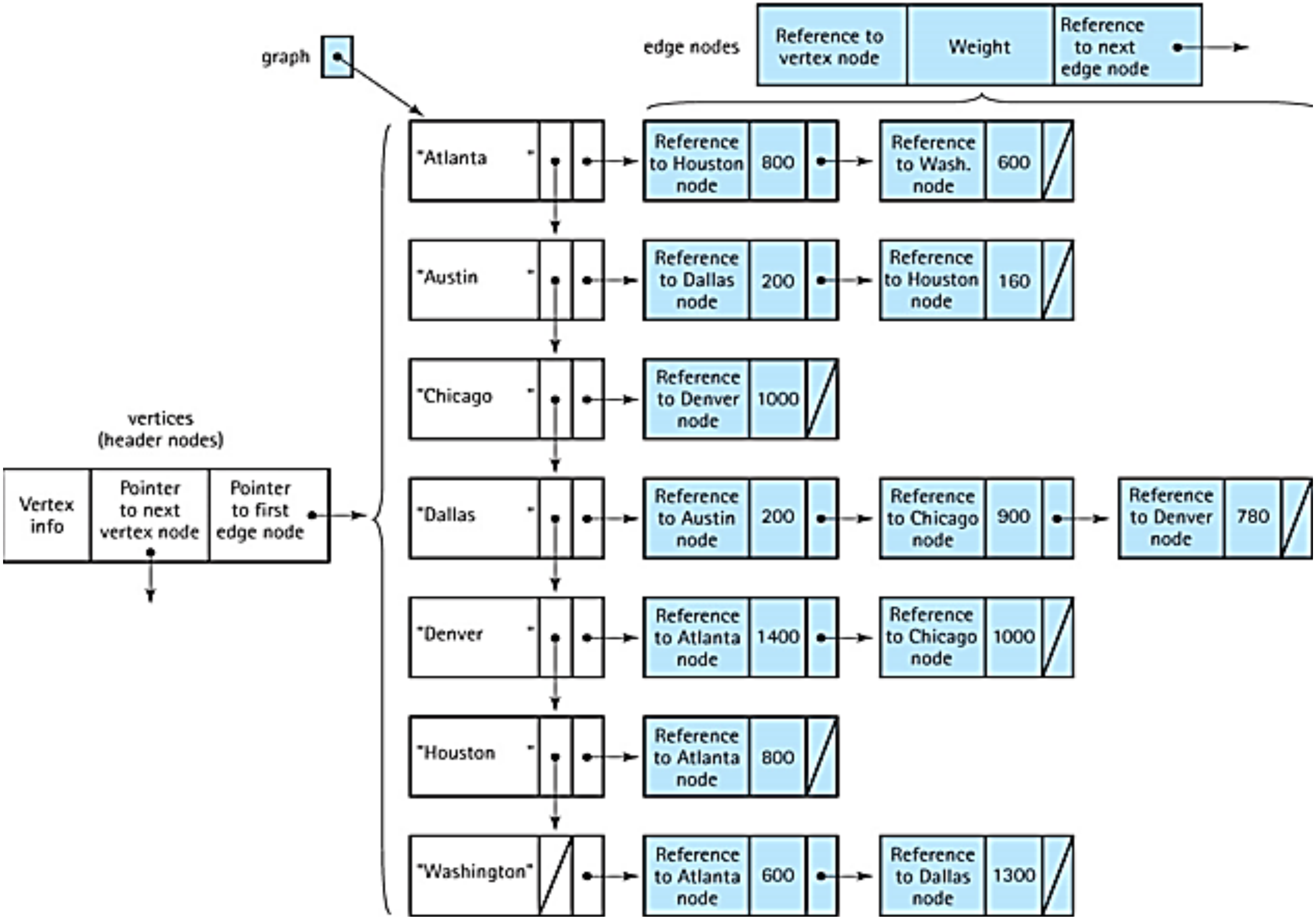
Linked-based implementation for graphs

- **Adjacency list** - A linked list that identifies all the vertices to which a particular vertex is connected; each vertex has its own adjacency list.
- We look at two alternate approaches:
  - Using an array of vertices that each contains a reference to a linked list of nodes;
  - Using a linked list of vertices that each contains a reference to a linked list of nodes.

Linked-based implementation for graphs



Linked-based implementation for graphs

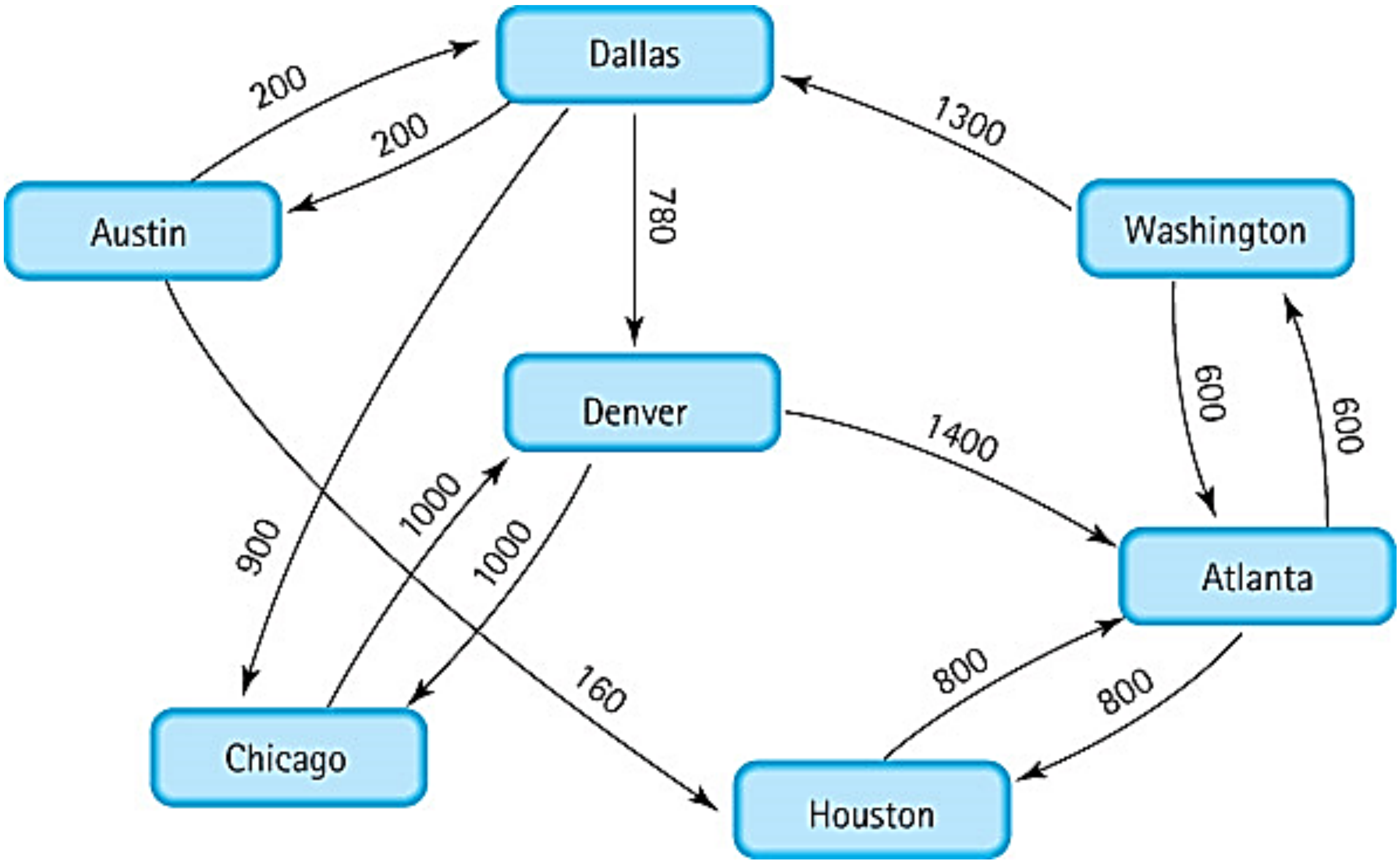


Graph traversal

- There are two types of graph traversal:
    - **Depth-first strategy** - The traversal goes down a branch to its deepest point and moves up. This type of traversal is also called Depth First Search (DFS).
    - **Breadth-first strategy** - The traversal visits each vertex on level 0 (the root), then each vertex on level 1, then each vertex on level 2, and so on. This type of traversal is also called Breadth First Search (BFS).

Graph traversal

- Can we get from Austin to Washington?



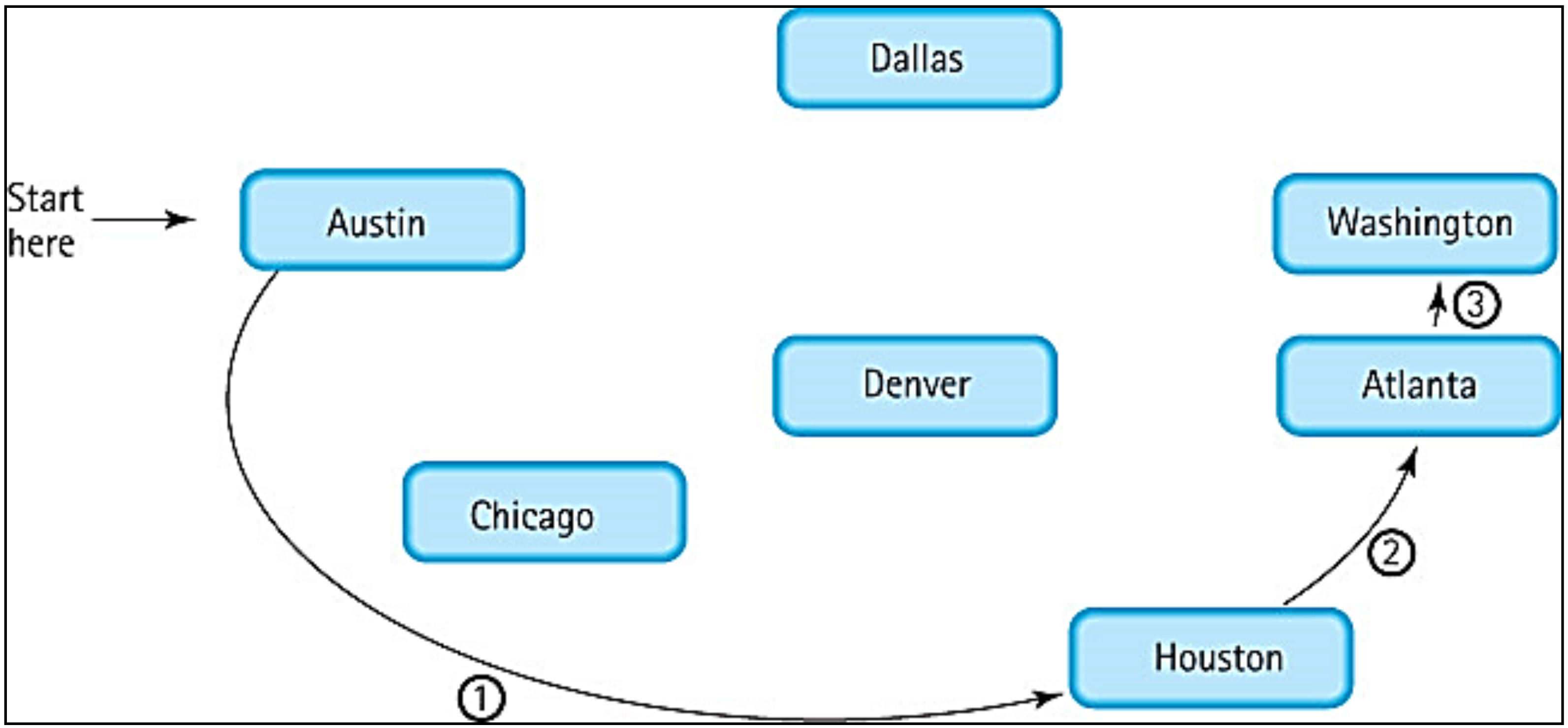
Graph traversal

- Depth first search

```
DFS(startVertex)
stack.push(startVertex)
while (!stack.isEmpty())
   vertex = stack.top()
   if (vertex is not visited)
       visit vertex
   if (all adjacent vertices are visited)
      stack.pop();
   else
      push an unvisited adjacent vertex onto stack
```

Graph traversal

- Depth first search



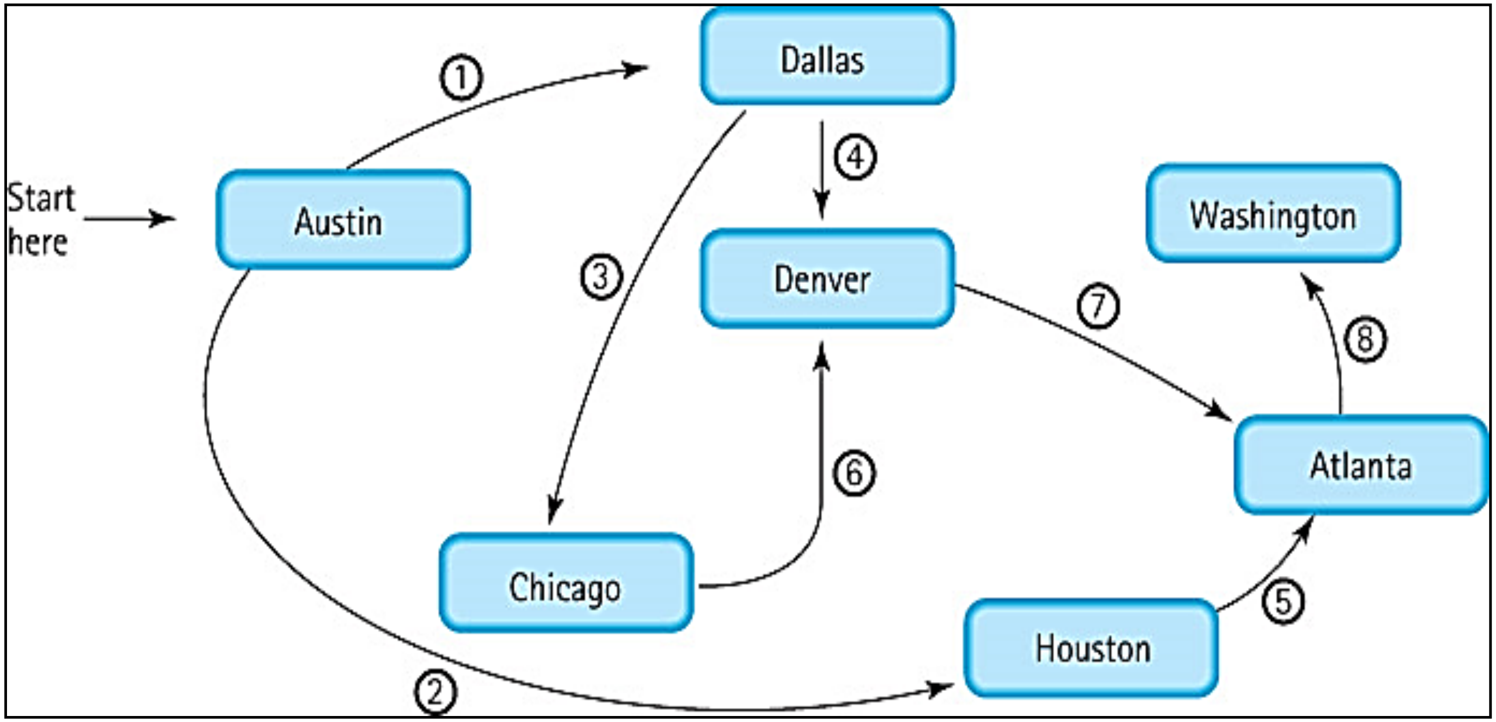
Graph traversal

- Breadth first search

```
BFS(startVertex)
vertex = startVertex
visit vertex
queue.enqueue(vertex)
while (!queue.isEmpty())
    vertex = queue.dequeue()
    visit all unvisited adjacent vertices and enqueue
    them onto queue
```

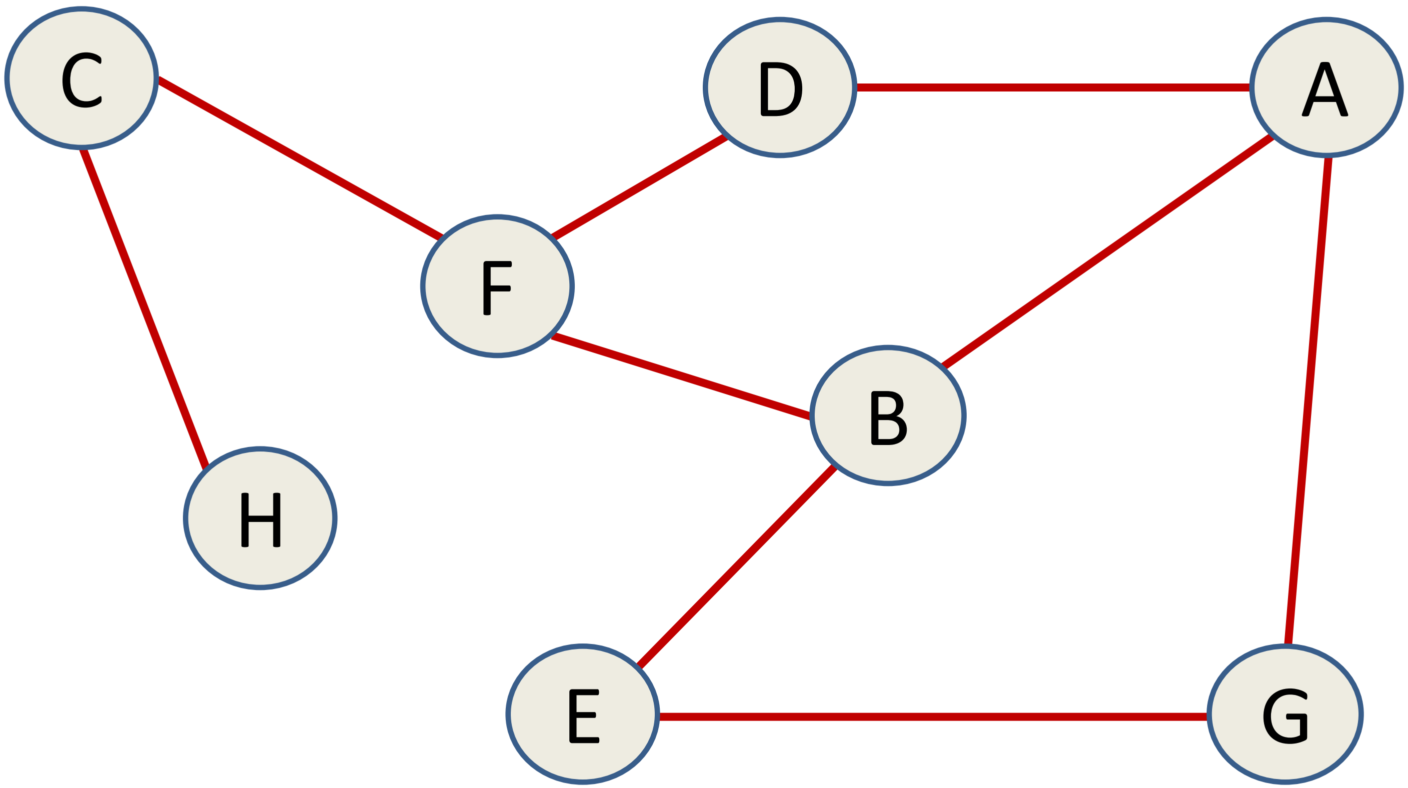
Graph traversal

- Breadth first search



Graph traversal

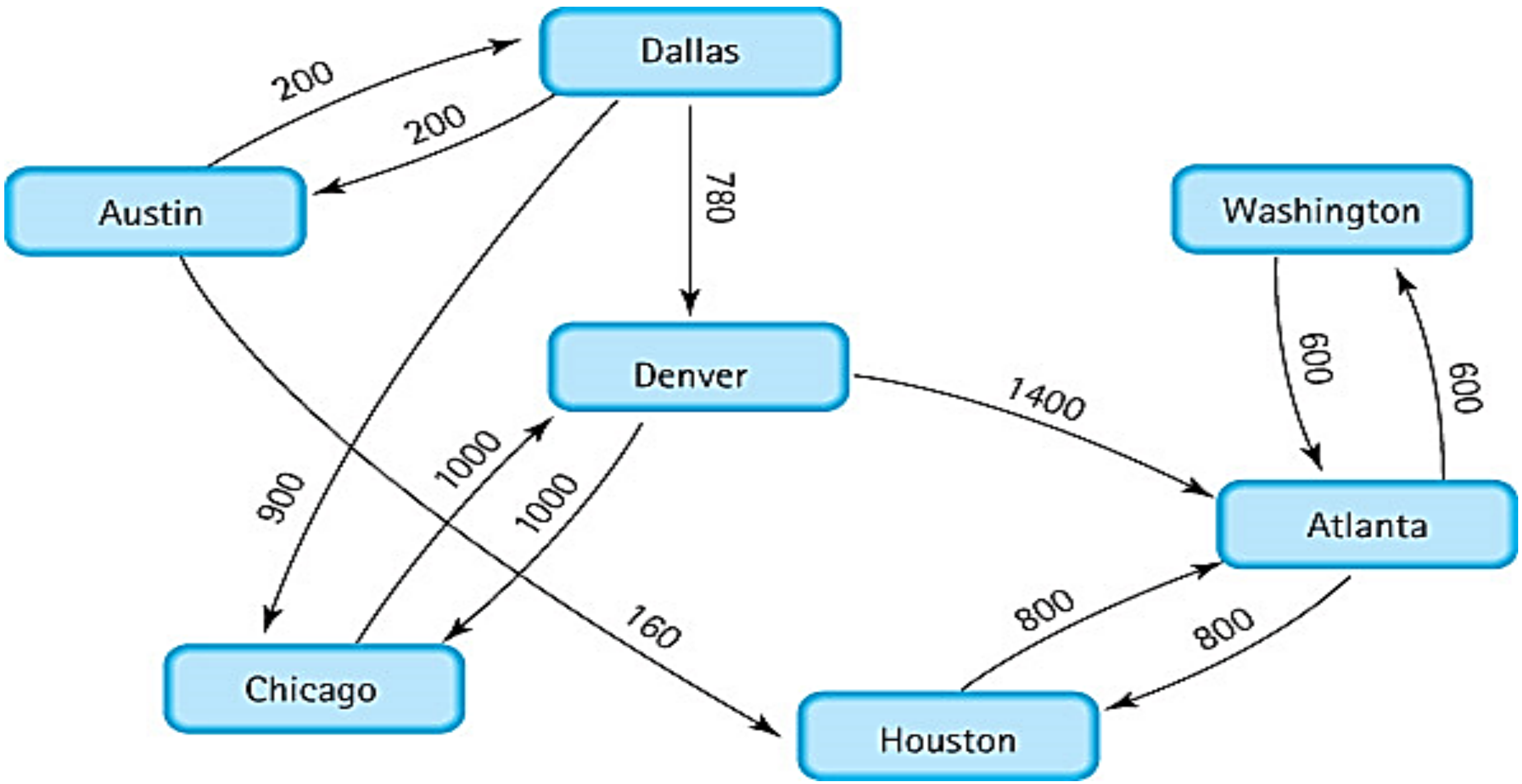
- Now let's do some practices!



- What would the DFS traversal print out? **A B E G F C H D**
- What would the BFS traversal print out? **A B D G E F C H**

Unreachable vertices

- With this new graph we cannot fly from Washington to Austin, Chicago, Dallas, or Denver.



Action items

- Read book chapter 9.