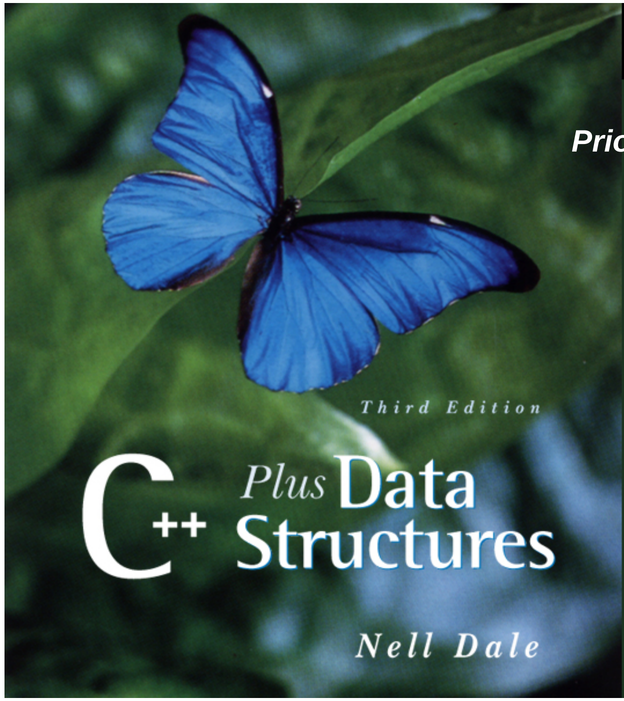


Chapter

9

Priority Queues, Heaps, and Graphs

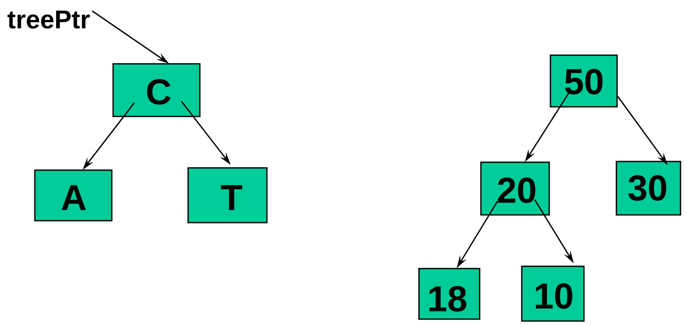
What is a Heap?

A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

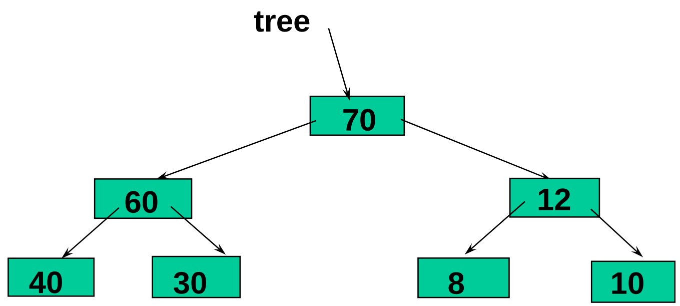
– **SHAPE property: Its shape must be a complete binary tree.**

– **ORDER property: For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.**

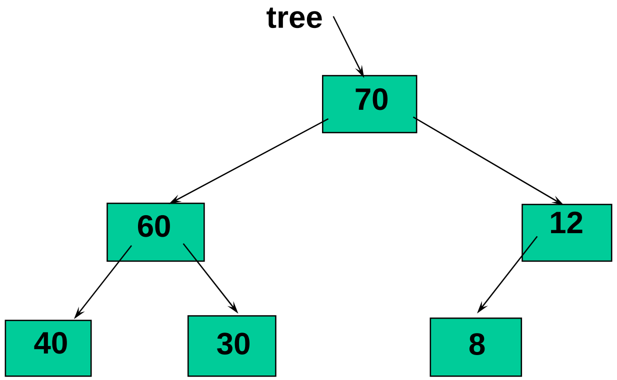
Are these Both Heaps?



Is this a Heap?

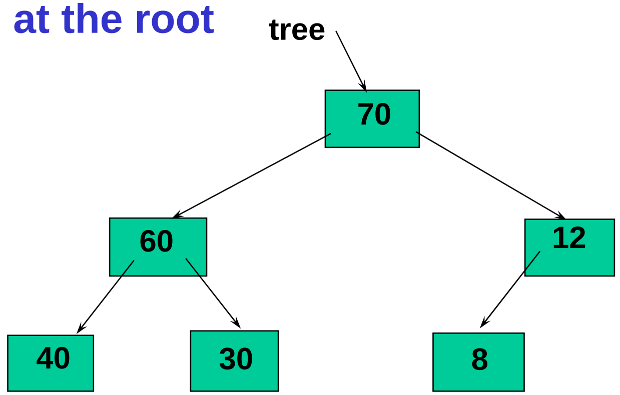


Where is the Largest Element in a Heap Always Found?



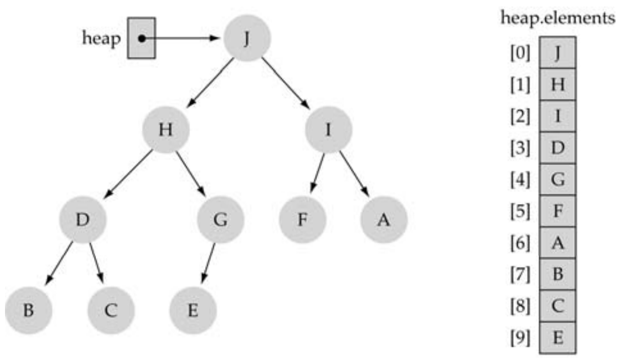
Where is the Largest Element in a Heap Always Found?

From ***ORDER property***, the largest value of the heap is always stored at the root

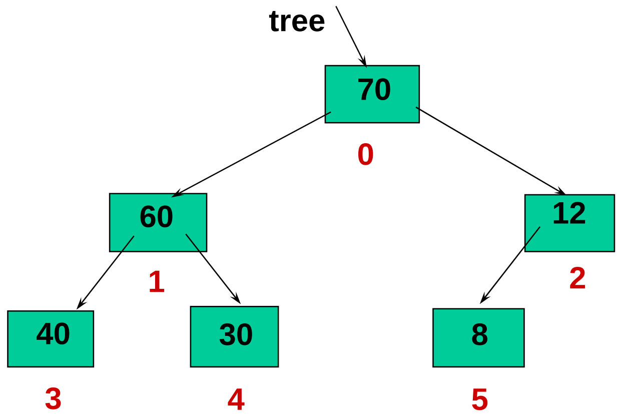


Heap implementation using array representation

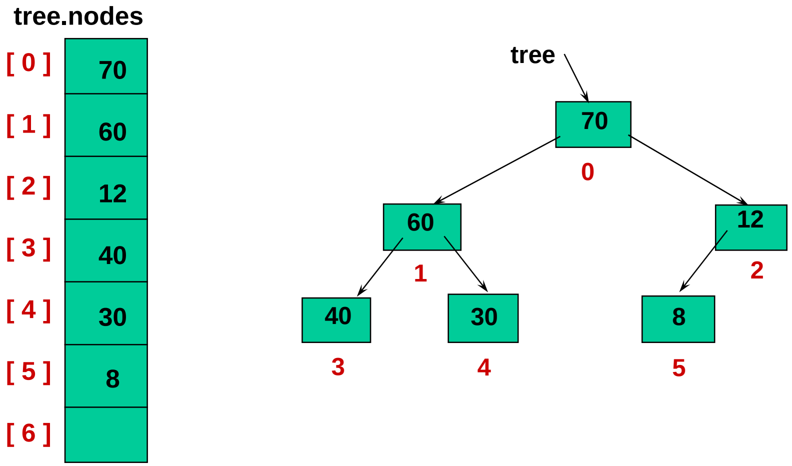
- A heap is a complete binary tree, so it is easy to be implemented using an array representation



We Can Number the Nodes Left to Right by Level This Way

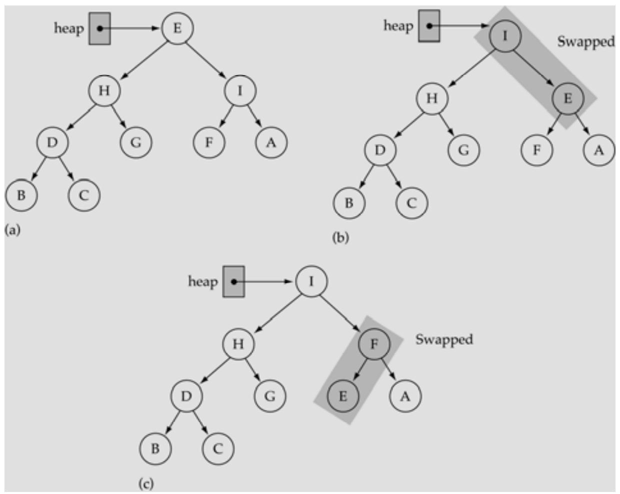


And use the Numbers as Array Indexes to Store the Trees



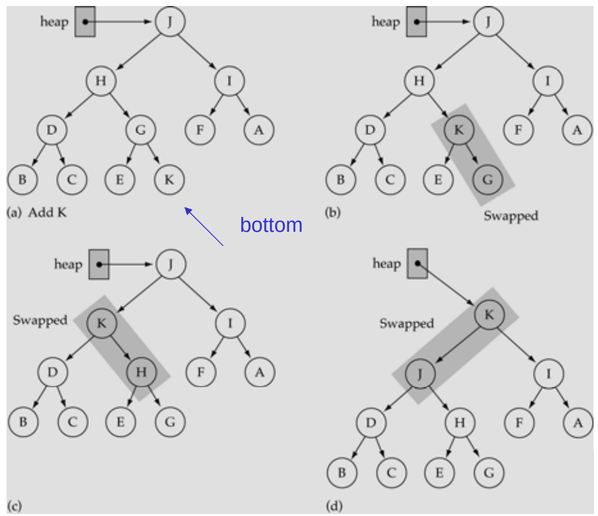

```
//  HEAP SPECIFICATION

//  Assumes  ItemType  is either a built-in simple data
// type or a class with overloaded relational operators.

template< class  ItemType >
struct  HeapType
{
   void   ReheapDown ( int  root ,  int  bottom ) ;
   void   ReheapUp ( int  root,  int  bottom ) ;

   ItemType* elements; //ARRAY to be allocated dynamically

   int  numElements ;
};
```


The ReheapDown function (used by deleteItem)



Assumption:
heap property is violated at the root of the tree

The ReheapUp function (used by insertItem)



Assumption:
heap property is violated at the rightmost node at the last level of the tree

ReheapDown

```
//  IMPLEMENTATION  OF RECURSIVE HEAP MEMBER FUNCTIONS

template< class  ItemType >
void   HeapType<ItemType>::ReheapDown ( int root, int  bottom )

// Pre:  root is the index of the node that may violate the
// heap order property
// Post: Heap order property is restored between root and bottom

{
      int  maxChild ;
      int  rightChild ;
      int  leftChild ;

      leftChild  =  root * 2 + 1 ;
      rightChild  =  root * 2 + 2 ;
```

rightmost node in the last level

ReheapDown (cont)

```
  if  ( leftChild  <=  bottom ) // Is there leftChild?
  {
     if ( leftChild  == bottom )  // only one child
        maxChild  =  leftChld ;
     else // two children
       {
         if (elements [ leftChild ] <= elements [ rightChild ] )
              maxChild  =  rightChild ;
         else
              maxChild  =  leftChild ;
       }
     if  ( elements [ root ] < elements [ maxChild ] )
       {
          Swap ( elements [ root ] , elements [ maxChild ] ) ;
          ReheapDown ( maxChild, bottom ) ;
       }
   }
}
```

ReheapUp

```
//  IMPLEMENTATION          continued

template< class  ItemType >
void   HeapType<ItemType>::ReheapUp ( int  root,  int  bottom )

//  Pre:  bottom is the index of the node that may violate the heap
//  order property.  The order property is satisfied from root to
//  next-to-last node.
//  Post:  Heap order property is restored between root and bottom

{
    int  parent ;

    if  ( bottom  > root ) // tree is not empty
    {
       parent = ( bottom - 1 ) / 2;
       if ( elements [ parent ]  <  elements [ bottom ] )
       {
          Swap ( elements [ parent ], elements [ bottom ] ) ;
          ReheapUp ( root, parent ) ;
       }
    }
}
```

rightmost node in the last level

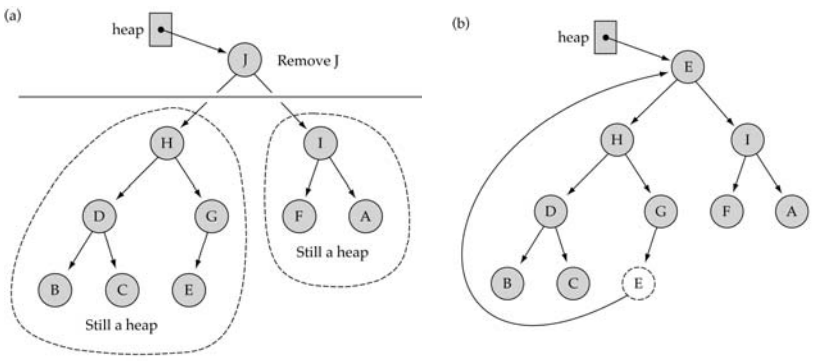
Removing the largest element from the heap

(1) Copy the bottom rightmost element to the root

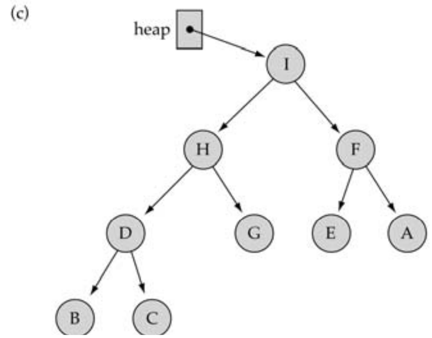
(2) Delete the bottom rightmost node

(3) Fix the heap property by calling *ReheapDown*

Removing the largest element from the heap (cont.)



Removing the largest element from the heap (cont.)

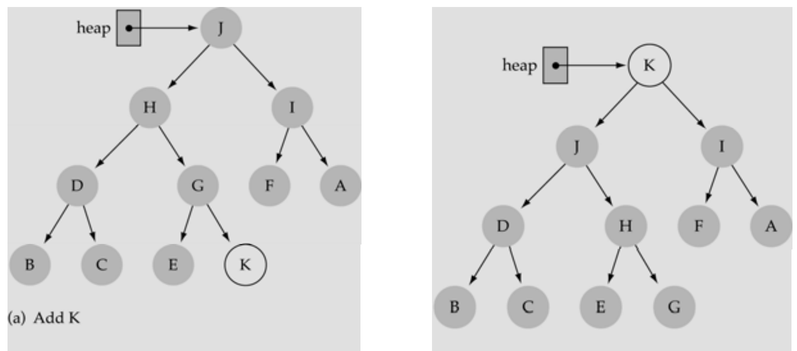


Inserting a new element into the heap

(1) Insert the new element in the next bottom leftmost place

(2) Fix the heap property by calling *ReheapUp*

Inserting a new element into the heap (cont.)



(a) Add K

(b) ReheapUp

Priority Queue

A priority queue is an ADT with the property that only the highest-priority element can be accessed at any time.

ADT Priority Queue Operations

Transformers
- **MakeEmpty**
- **Enqueue**
- **Dequeue**

change state

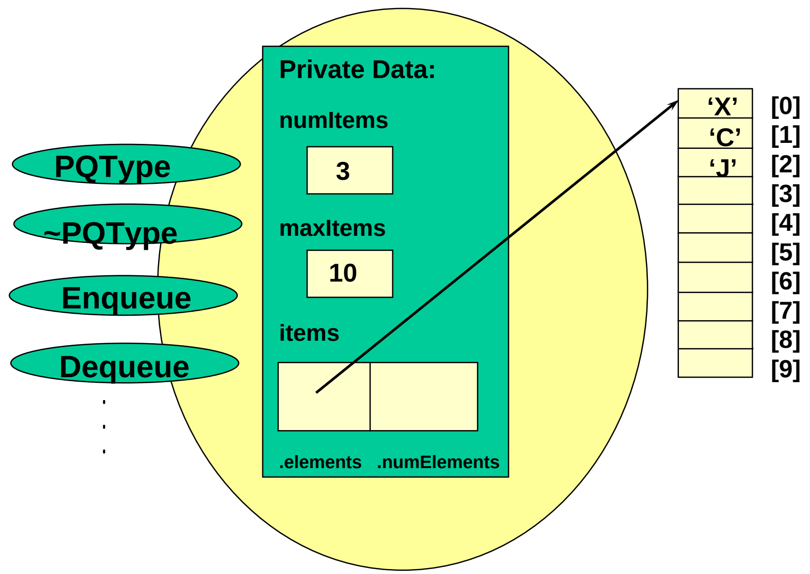
Observers
- **IsEmpty**
- **IsFull**

observe state

Implementation Level

- 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 0(N)
  - **A Binary Search Tree-** On average, $0(\log_2 N)$ steps for both enqueue and dequeue
  - **A Heap**- guarantees $0(\log_2 N)$ steps, even in the worst case

class PQType<char>



Class PQType Declaration

```
class FullPQ(){};
class EmptyPQ(){};
template<class ItemType>
class PQType
{
public:
  PQType(int);
  ~PQType();
  void MakeEmpty();
  bool IsEmpty() const;
  bool IsFull() const;
  void Enqueue(ItemType newItem);
  void Dequeue(ItemType& item);
private:
  int length;
  HeapType<ItemType> items;
  int maxItems;
};
```


Class PQType Function Definitions

```
template<class ItemType>
PQType<ItemType>::PQType(int max)
{
  maxItems = max;
  items.elements = new ItemType[max];
  length = 0;
}
template<class ItemType>
void PQType<ItemType>::MakeEmpty()
{
  length = 0;
}
template<class ItemType>
PQType<ItemType>::~PQType()
{
  delete [] items.elements;
}
```


Class PQType Function Definitions

Dequeue
Set item to root element from queue
Move last leaf element into root position
Decrement length
items.ReheapDown(0, length-1)

Enqueue
Increment length
Put newItem in next available position
  items.ReheapUp(0, length-1)

Code for Dequeue

```
template<class ItemType>
void PQType<ItemType>::Dequeue(ItemType& item)
{
  if (length == 0)
    throw EmptyPQ();
  else
  {
    item = items.elements[0];
    items.elements[0] = items.elements[length-1];
    length--;
    items.ReheapDown(0, length-1);
  }
}
```

Code for Enqueue

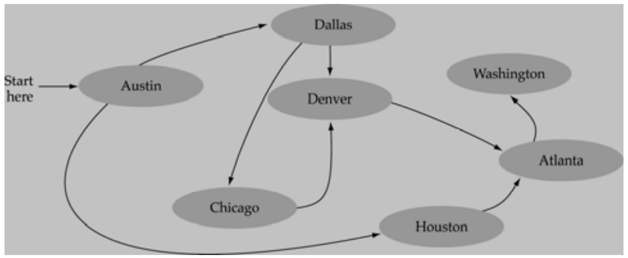
```
template<class ItemType>
void PQType<ItemType>::Enqueue(ItemType newItem)
{
  if (length == maxItems)
    throw FullPQ();
  else
  {
    length++;
    items.elements[length-1] = newItem;
    items.ReheapUp(0, length-1);
  }
}
```


Comparison of Priority Queue Implementations

	Enqueue	*Dequeue*
Heap	**O(log$_2$*N*)**	**O(log$_2$*N*)**
Linked List	**O(*N*)**	**O(*N*)**
Binary Search Tree		
Balanced	**O(log$_2$*N*)**	**O(log$_2$*N*)**
Skewed	**O(*N*)**	**O(*N*)**

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)

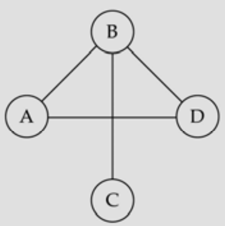
  V(G): a finite, nonempty set of vertices

  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*

ected graph.

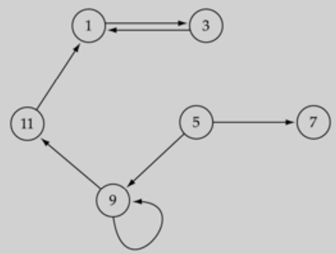


V(Graph1) = { A, B, C, D }
E(Graph1) = { (A, B), (A, D), (B, C), (B, D) }

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.



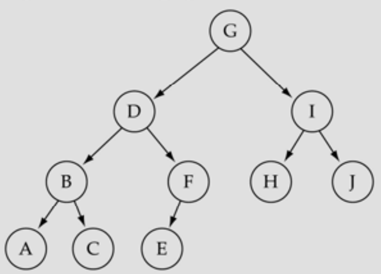
V(Graph2) = { 1, 3, 5, 7, 9, 11 }
E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7)), (9, 9), (11, 1) }

Warning: if the graph is directed, the order of the vertices in each edge is important !!

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

5 is adjacent to 7
7 is adjacent from 5

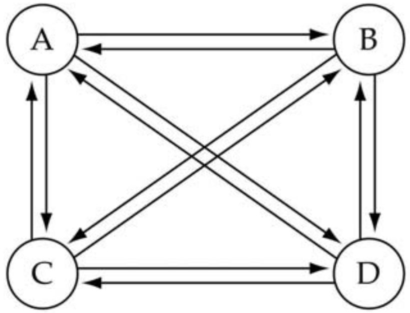
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$O(N^2)$



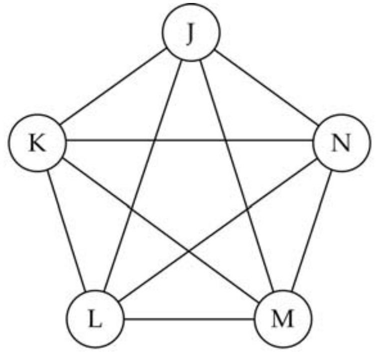
(a) Complete directed graph.

Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

*N * (N-1) / 2*

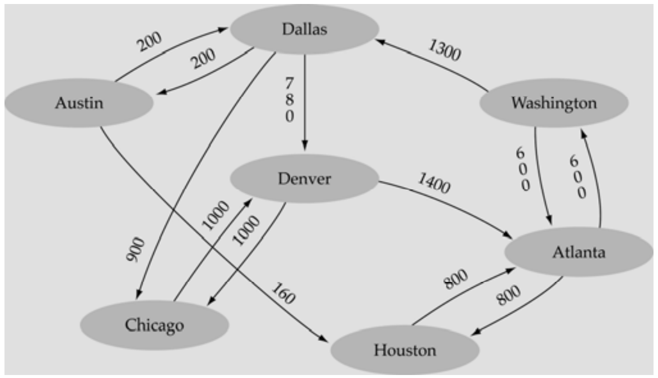
$O(N^2)$



(b) Complete undirected graph.

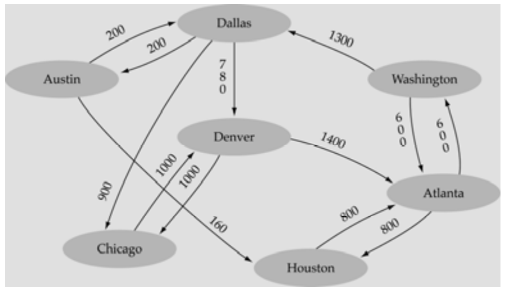
Graph terminology (cont.)

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



Graph implementation

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



Array-based implementation

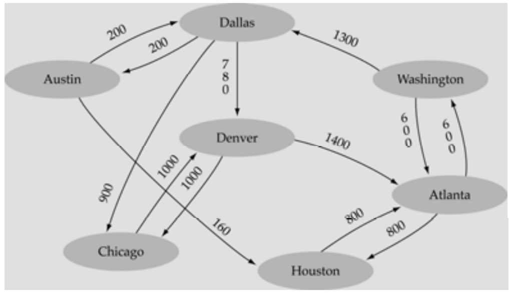
graph
.numVertices 7
.vertices .edges

	.vertices	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	"Atlanta "	0	0	0	0	0	800	600	•	•	•
[1]	"Austin "	0	0	0	200	0	160	0	•	•	•
[2]	"Chicago "	0	0	0	0	1000	0	0	•	•	•
[3]	"Dallas "	0	200	900	0	780	0	0	•	•	•
[4]	"Denver "	1400	0	1000	0	0	0	0	•	•	•
[5]	"Houston "	800	0	0	0	0	0	0	•	•	•
[6]	"Washington"	600	0	0	1300	0	0	0	•	•	•
[7]		•	•	•	•	•	•	•	•	•	•
[8]		•	•	•	•	•	•	•	•	•	•
[9]		•	•	•	•	•	•	•	•	•	•

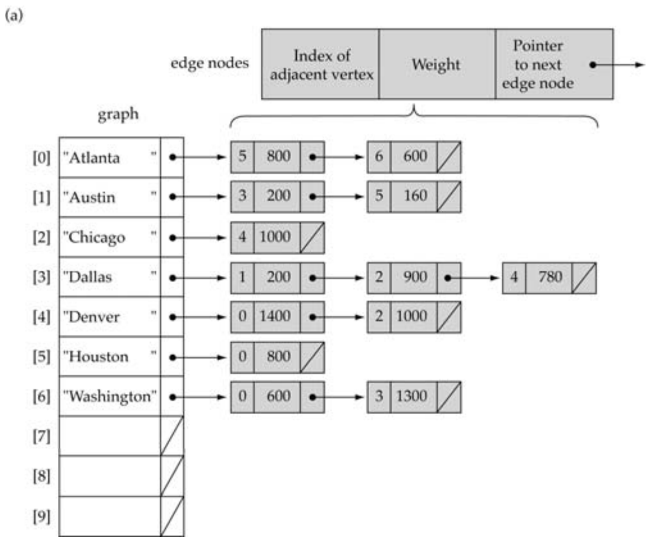
(Array positions marked '•' are undefined)

Graph implementation (cont.)

- Adjacency List: 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*



Linked-list implementation



Adjacency matrix vs. adjacency list representation

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

Graph specification based on adjacency matrix representation

```
const int NULL_EDGE = 0;

template<class VertexType>
class GraphType {
  public:
    GraphType(int);
    ~GraphType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void AddVertex(VertexType);
    void AddEdge(VertexType, VertexType, int);
    int WeightIs(VertexType, VertexType);
    void GetToVertices(VertexType,
  QueType<VertexType>&);
    void ClearMarks();
    void MarkVertex(VertexType);
    bool IsMarked(VertexType) const;
```

```
private:
   int numVertices;
   int maxVertices;
   VertexType* vertices;
   int **edges;
   bool* marks;
};
```

(continues)


```
template<class VertexType>
GraphType<VertexType>::GraphType(int maxV)
{
 numVertices = 0;
 maxVertices = maxV;
 vertices = new VertexType[maxV];
 edges = new int[maxV];
 for(int i = 0; i < maxV; i++)
   edges[i] = new int[maxV];
 marks = new bool[maxV];
}

template<class VertexType>
GraphType<VertexType>::~GraphType()
{
 delete [] vertices;
 for(int i = 0; i < maxVertices; i++)
   delete [] edges[i];
 delete [] edges;
 delete [] marks;
}
```

(continues)


```
void GraphType<VertexType>::AddVertex(VertexType vertex)
{
 vertices[numVertices] = vertex;

 for(int index = 0; index < numVertices; index++) {
   edges[numVertices][index] = NULL_EDGE;
   edges[index][numVertices] = NULL_EDGE;
 }

 numVertices++;
}

template<class VertexType>
void GraphType<VertexType>::AddEdge(VertexType fromVertex,
                     VertexType toVertex, int weight)
{
 int row;
 int column;

 row = IndexIs(vertices, fromVertex);
 col = IndexIs(vertices, toVertex);
 edges[row][col] = weight;
}
```

(continues)


```
template<class VertexType>
int GraphType<VertexType>::WeightIs(VertexType fromVertex,
                          VertexType toVertex)
{
 int row;
 int column;

 row = IndexIs(vertices, fromVertex);
 col = IndexIs(vertices, toVertex);
 return edges[row][col];
}
```

Graph searching

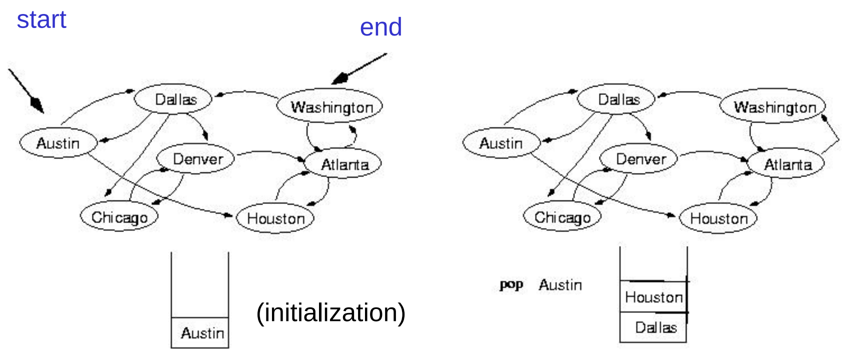
- *Problem*: find a path between two nodes of the graph (e.g., Austin and Washington)
- *Methods*: Depth-First-Search (DFS) or Breadth-First-Search (BFS)

Depth-First-Search (DFS)

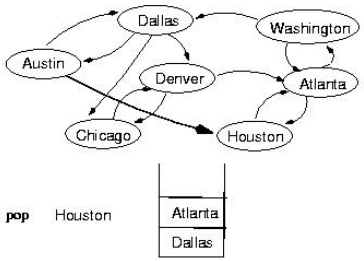
- What is the idea behind DFS?
  - Visit all nodes in a branch to its deepest point before moving up
  - Travel as far as you can down a path
- DFS can be implemented efficiently using a *stack*

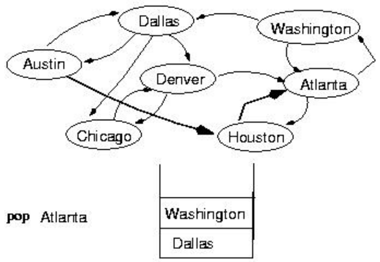
Depth-First-Search (DFS) *(cont.)*

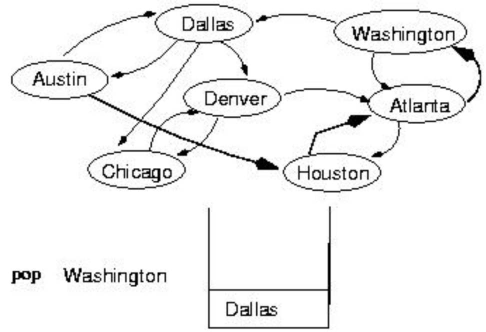
```
Set found to false
stack.Push(startVertex)
DO
  stack.Pop(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Push all adjacent vertices onto stack
WHILE !stack.IsEmpty() AND !found

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

(initialization)








```
template <class ItemType>
void DepthFirstSearch(GraphType<VertexType> graph,
  VertexType startVertex, VertexType endVertex)
{
 StackType<VertexType> stack;
 QueType<VertexType> vertexQ;

 bool found = false;
 VertexType vertex;
 VertexType item;

 graph.ClearMarks();
 stack.Push(startVertex);
 do {
   stack.Pop(vertex);
   if(vertex == endVertex)
     found = true;
```

(continues)


```
  else {
      if(!graph.IsMarked(vertex)) {
        graph.MarkVertex(vertex);
        graph.GetToVertices(vertex, vertexQ);

        while(!vertexQ.IsEmpty()) {
          vertexQ.Dequeue(item);
          if(!graph.IsMarked(item))
            stack.Push(item);
        }
      }
    } while(!stack.IsEmpty() && !found);

    if(!found)
      cout << "Path not found" << endl;
}
```

(continues)

```
template<class VertexType>
void
  GraphType<VertexType>::GetToVertices(VertexType
  vertex,

  QueTye<VertexType>& adjvertexQ)
{
 int fromIndex;
 int toIndex;

 fromIndex = IndexIs(vertices, vertex);
 for(toIndex = 0; toIndex < numVertices;
  toIndex++)
   if(edges[fromIndex][toIndex] != NULL_EDGE)
     adjvertexQ.Enqueue(vertices[toIndex]);
}
```


Breadth-First-Searching (BFS)

- What is the idea behind BFS?
  - Visit all the nodes on one level before going to the next level
  - Look at all possible paths at the same depth before you go at a deeper level

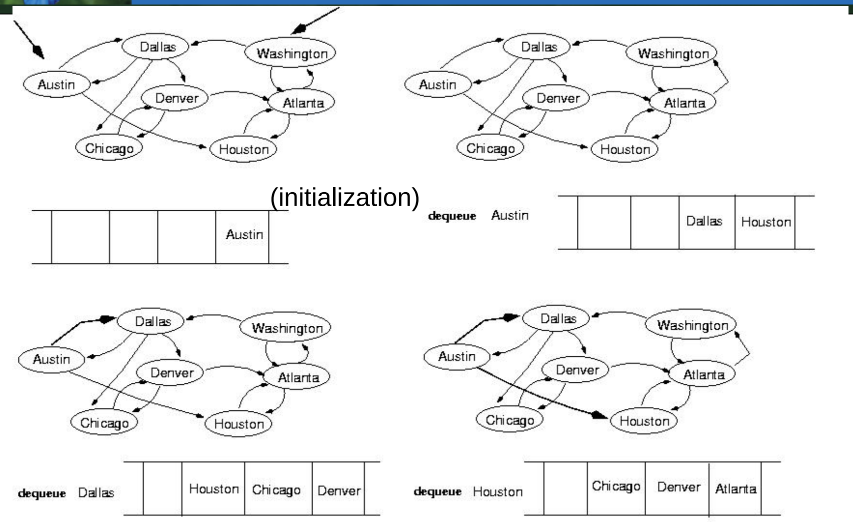
Breadth-First-Searching (BFS) (cont.)

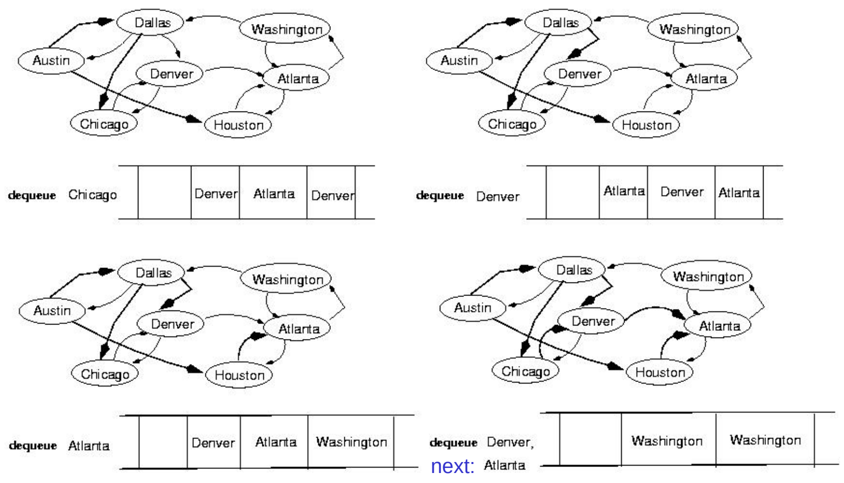
- BFS can be implemented efficiently using a *queue*

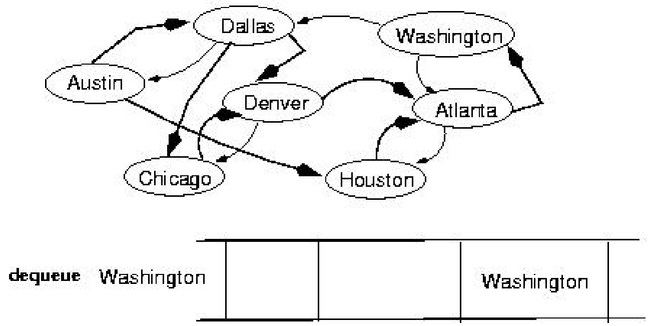
```
Set found to false
queue.Enqueue(startVertex)
DO
  queue.Dequeue(vertex)
  IF vertex == endVertex
    Set found to true
  ELSE
    Enqueue all adjacent vertices onto queue
WHILE !queue.IsEmpty() AND !found
IF(!found)
  Write "Path does not exist"
```

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

start end








```
template<class VertexType>
void BreadthFirtsSearch(GraphType<VertexType>
  graph, VertexType startVertex, VertexType
  endVertex);
{
 QueType<VertexType> queue;
 QueType<VertexType> vertexQ;//

 bool found = false;
 VertexType vertex;
 VertexType item;

 graph.ClearMarks();
 queue.Enqueue(startVertex);
 do {
   queue.Dequeue(vertex);
   if(vertex == endVertex)
     found = true;
```

(continues)


```
   else {
     if(!graph.IsMarked(vertex)) {
       graph.MarkVertex(vertex);
       graph.GetToVertices(vertex, vertexQ);

       while(!vertxQ.IsEmpty()) {
         vertexQ.Dequeue(item);
         if(!graph.IsMarked(item))
           queue.Enqueue(item);
       }
     }
   }
 } while (!queue.IsEmpty() && !found);

 if(!found)
   cout << "Path not found" << endl;
}
```

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)

ADT Set Definitions

Base type: The type of the items in the set
Cardinality**:** The number of items in a set
Cardinality of the base type**:** The number of items in the base type
Union of two sets: A set made up of all the items in either sets
Intersection of two sets**:** A set made up of all the items in both sets
Difference of two sets**:** A set made up of all the items in the first set that are not in the second set

Beware: At the Logical Level

- Sets can not contain duplicates. Storing an item that is already in the set does not change the set.
- If an item is not in a set, deleting that item from the set does not change the set.
- Sets are not ordered.

Implementing Sets

Explicit implementation (Bit vector)
Each item in the base type has a representation in each instance of a set. The representation is either true (item is in the set) or false (item is not in the set).

Space is proportional to the cardinality of the base type.

Algorithms use Boolean operations.

Implementing Sets (cont.)

Implicit implementation (List)
The items in an instance of a set are on a list that represents the set. Those items that are not on the list are not in the set.

Space is proportional to the cardinality of the set instance.

Algorithms use ADT List operations.

Explain:

Although sets are not ordered, why is the SortedList ADT a better choice as the implementation structure for the implicit representation?