AVL (Height-Balanced) Trees

• AVL tree (height-balanced tree)
  – Resulting binary search is nearly balanced
• Perfectly balanced binary tree
  – Heights of left and right subtrees of the root: equal
  – Left and right subtrees of the root are perfectly balanced binary trees

FIGURE 11-12 Perfectly balanced binary tree
AVL (Height-Balanced) Trees (cont’d.)

• An AVL tree (or height-balanced tree) is a binary search tree such that
  – The heights of the left and right subtrees of the root differ by at most one
  – The left and right subtrees of the root are AVL trees

FIGURE 11-13 AVL and non-AVL trees
**Proposition:** Let $T$ be an AVL tree and $x$ be a node in $T$. Then $|x_h - x_l| \leq 1$, where $|x_h - x_l|$ denotes the absolute value of $x_h - x_l$.

Let $x$ be a node in the AVL tree $T$.

1. If $x_l > x_h$, we say that $x$ is left high. In this case, $x_l = x_h + 1$.
2. If $x_l = x_h$, we say that $x$ is equal high.
3. If $x_h > x_l$, we say that $x$ is right high. In this case, $x_h = x_l + 1$.

**Definition:** The balance factor of $x$, written $bf(x)$, is defined by $bf(x) = x_h - x_l$.

Let $x$ be a node in the AVL tree $T$. Then,

1. If $x$ is left high, $bf(x) = -1$.
2. If $x$ is equal high, $bf(x) = 0$.
3. If $x$ is right high, $bf(x) = 1$.

**Definition:** Let $x$ be a node in a binary tree. We say that the node $x$ violates the balance criteria if $|x_h - x_l| > 1$, that is, the heights of the left and right subtrees of $x$ differ by more than 1.
Binary Search Trees (cont’d.)

- A binary search tree, $T$, is either empty or the following is true:
  - $T$ has a special node called the root node
  - $T$ has two sets of nodes, $L_T$ and $R_T$, called the left subtree and right subtree of $T$, respectively
  - The key in the root node is larger than every key in the left subtree and smaller than every key in the right subtree
  - $L_T$ and $R_T$ are binary search trees
AVL (Height-Balanced) Trees (cont’d.)

- Definition of a node in the AVL tree

```cpp
template<class elemType>
struct AVLNNode
{
    elemType info;
    int bfactor;  //balance factor
    AVLNode<elemType> *llink;
    AVLNode<elemType> *rlink;
};
```
AVL (Height-Balanced) Trees (cont’d.)

• AVL binary search tree search algorithm
  – Same as for a binary search tree
  – Other operations on AVL trees
    • Implemented exactly the same way as binary trees
    – Item insertion and deletion operations on AVL trees
      • Somewhat different from binary search trees operations
Insertion

• First search the tree and find the place where the new item is to be inserted
  – Can search using algorithm similar to search algorithm designed for binary search trees
  – If the item is already in tree
    • Search ends at a nonempty subtree
    • Duplicates are not allowed
  – If item is not in AVL tree
    • Search ends at an empty subtree; insert the item there
• After inserting new item in the tree
  – Resulting tree might not be an AVL tree
Insertion (cont’d.)

**FIGURE 11-14** AVL tree before and after inserting 90

**FIGURE 11-15** AVL tree before and after inserting 75
Insertion (cont’d.)

FIGURE 11-16 AVL tree before and after inserting 95

FIGURE 11-17 AVL tree before and after inserting 88
AVL Tree Rotations

- Rotating tree: reconstruction procedure
- Left rotation and right rotation
- Suppose that the rotation occurs at node x
  - Left rotation: certain nodes from the right subtree of x move to its left subtree; the root of the right subtree of x becomes the new root of the reconstructed subtree
  - Right rotation at x: certain nodes from the left subtree of x move to its right subtree; the root of the left subtree of x becomes the new root of the reconstructed subtree
FIGURE 11-18 Right rotation at $b$

FIGURE 11-19 Left rotation at $a$
FIGURE 11-20 Double rotation: First rotate left at a and then right at c

FIGURE 11-21 Left rotation at a followed by a right rotation at c
AVL Tree Rotations (cont’d.)

FIGURE 11-22 Double rotation: First rotate right at c, then rotate left at a
template <class elemT>
void rotateToLeft(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;  //pointer to the root of the
    //right subtree of root
    if (root == NULL)
        cerr << "Error in the tree" << endl;
    else if (root->rlink == NULL)
        cerr << "Error in the tree:"
         << " No right subtree to rotate."
            << endl;
    else
        {
            p = root->rlink;
            root->rlink = p->llink;  //the left subtree of p becomes
            //the right subtree of root
            p->llink = root;
            root = p;  //make p the new root node
        }
}  //rotateLeft

template <class elemT>
void rotateToRight(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;  //pointer to the root of
    //the left subtree of root
    if (root == NULL)
        cerr << "Error in the tree" << endl;
    else if (root->llink == NULL)
        cerr << "Error in the tree:"
         << " No left subtree to rotate."
            << endl;
    else
        {
            p = root->llink;
            root->llink = p->rlink;  //the right subtree of p becomes
            //the left subtree of root
            p->rlink = root;
            root = p;  //make p the new root node
        }
}  //end rotateRight
template <class elemT>
void balanceFromLeft(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *w;

    p = root->llink;    // p points to the left subtree of root

    switch (p->bfactor)
    {
    case -1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToRight(root);
        break;

    case 0:
        cerr << "Error: Cannot balance from the left." << endl;
        break;

    case 1:
        w = p->rlink;
        switch (w->bfactor)  // adjust the balance factors
        {
        case -1:
            root->bfactor = 1;
            p->bfactor = 0;
            break;

        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;

        case 1:
            root->bfactor = 0;
            p->bfactor = -1;
        } // end switch
        w->bfactor = 0;
        rotateToLeft(p);
        root->llink = p;
        rotateToRight(root);
    } // end switch;
} // end balanceFromLeft
template <class elemT>
void balanceFromRight(AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *w;

    p = root->rlink;  // p points to the left subtree of root

    switch (p->bfactor)
    {
    case -1:
        w = p->llink;
        switch (w->bfactor) // adjust the balance factors
        {
        case -1:
            root->bfactor = 0;
            p->bfactor = 1;
            break;

        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;

        case 1:
            root->bfactor = -1;
            p->bfactor = 0;
        } // end switch

        w->bfactor = 0;
        rotateToRight(p);
        root->rlink = p;
        rotateToLeft(root);
        break;

    case 0:
        cerr << "Error: Cannot balance from the left." << endl;
        break;

    case 1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToLeft(root);
    } // end switch
} // end balanceFromRight
FIGURE 11-23 Item insertion into an initially empty AVL tree
Data Structures Using C++ 2E

Chapter 12
Graphs
Introduction (cont’d.)

**FIGURE 12-1** The Königsberg bridge problem

**FIGURE 12-2** Graph representation of the Königsberg bridge problem
Graph Definitions and Notations

• Borrow definitions, terminology from set theory
• Subset
  – Set $Y$ is a subset of $X$: $Y \subseteq X$
    • If every element of $Y$ is also an element of $X$
• Intersection of sets $A$ and $B$: $A \cap B$
  – Set of all elements that are in $A$ and $B$
• Union of sets $A$ and $B$: $A \cup B$
  – Set of all elements in $A$ or in $B$
• Cartesian product: $A \times B$
  – Set of all ordered pairs of elements of $A$ and $B$
Graph Definitions and Notations (cont’d.)

- **Graph G pair**
  - \( G = (V, E) \), where \( V \) is a finite nonempty set
    - Called the set of vertices of \( G \), and \( E \subseteq V \times V \)
  - Elements of \( E \)
    - Pairs of elements of \( V \)

- **\( E \): set of edges of \( G \)**
  - \( G \) called trivial if it has only one vertex

- **Directed graph (digraph)**
  - Elements in set of edges of graph \( G \): ordered

- **Undirected graph**: not ordered
**FIGURE 12-3** Various undirected graphs

![Undirected Graphs](image)

**FIGURE 12-4** Various directed graphs

![Directed Graphs](image)

\[
V(G_1) = \{1, 2, 3, 4, 5\} \\
E(G_1) = \{(1, 2), (1, 4), (2, 5), (3, 1), (3, 4), (4, 5)\}
\]

\[
V(G_2) = \{0, 1, 2, 3, 4\} \\
E(G_2) = \{(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)\}
\]

\[
V(G_3) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \\
E(G_3) = \{(0, 1), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (4, 3), (5, 6), (6, 8), (7, 3), (7, 8), (8, 10), (9, 4), (9, 7), (9, 10)\}
\]
Graph Definitions and Notations (cont’d.)

• Graph $H$ called subgraph of $G$
  – If $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$
  – Every vertex of $H$: vertex of $G$
  – Every edge in $H$: edge in $G$

• Graph shown pictorially
  – Vertices drawn as circles
    • Label inside circle represents vertex

• Undirected graph: edges drawn using lines

• Directed graph: edges drawn using arrows
Graph Definitions and Notations (cont’d.)

• Let $u$ and $v$ be two vertices in $G$
  – $u$ and $v$ adjacent
    • If edge from one to the other exists: $(u, v) \in E$

• Loop
  – Edge incident on a single vertex

• $e_1$ and $e_2$ called parallel edges
  – If two edges $e_1$ and $e_2$ associate with same pair of vertices \{u, v\}

• Simple graph
  – No loops, no parallel edges
Graph Definitions and Notations (cont’d.)

• Let \( e = (u, v) \) be an edge in \( G \)
  – Edge \( e \) is incident on the vertices \( u \) and \( v \)
  – Degree of \( u \) written \( \text{deg}(u) \) or \( d(u) \)
    • Number of edges incident with \( u \)

• Each loop on vertex \( u \)
  – Contributes two to the degree of \( u \)

• \( u \) is called an even (odd) degree vertex
  – If the degree of \( u \) is even (odd)
Graph Definitions and Notations (cont’d.)

• Path from \( u \) to \( v \)
  – If sequence of vertices \( u_1, u_2, \ldots, u_n \) exists
    • Such that \( u = u_1, \ u_n = v \) and \((u_i, u_{i+1})\) is an edge for all \( i = 1, 2, \ldots, n - 1 \)

• Vertices \( u \) and \( v \) called connected
  – If path from \( u \) to \( v \) exists

• Simple path
  – All vertices distinct (except possibly first, last)

• Cycle in \( G \)
  – Simple path in which first and last vertices are the same
Graph Definitions and Notations (cont’d.)

- **G** is connected
  - If path from any vertex to any other vertex exists

- **Component of G**
  - Maximal subset of connected vertices

- Let **G** be a directed graph and let **u** and **v** be two vertices in **G**
  - If edge from **u** to **v** exists: \((u, v) \in E\)
    - **u** is adjacent to **v**
    - **v** is adjacent from **u**
Graph Definitions and Notations (cont’d.)

• Definitions of paths and cycles in $G$
  – Similar to those for undirected graphs

• $G$ is strongly connected
  – If any two vertices in $G$ are connected
Graph Representation

• Graphs represented in computer memory
  – Two common ways
    • Adjacency matrices
    • Adjacency lists
Adjacency Matrices

• Let $G$ be a graph with $n$ vertices where $n > 0$
• Let $V(G) = \{v_1, v_2, \ldots, v_n\}$
  – Adjacency matrix

$$A_G(i,j) = \begin{cases} 
1 & \text{if } (v_i, v_j) \in E(G) \\
0 & \text{otherwise}
\end{cases}$$

$$A_{G_1} = \begin{bmatrix} 
0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}, \\
A_{G_2} = \begin{bmatrix} 
0 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{bmatrix}.$$
Adjacency Lists

• Given:
  – Graph $G$ with $n$ vertices, where $n > 0$
  – $V(G) = \{v_1, v_2, ..., v_n\}$

• For each vertex $v$: linked list exists
  – Linked list node contains vertex $u$: $(v, u) \in E(G)$

• Use array $A$, of size $n$, such that $A[i]$
  – Reference variable pointing to first linked list node containing vertices to which $v_i$ adjacent

• Each node has two components: vertex, link
  – Component vertex
    • Contains index of vertex adjacent to vertex $i$
Adjacency Lists (cont’d.)

- Example 12-4

**FIGURE 12-5** Adjacency list of graph G2 of Figure 12-4

**FIGURE 12-6** Adjacency list of graph G3 of Figure 12-4
Operations on Graphs

• Commonly performed operations
  – Create graph
    • Store graph in computer memory using a particular graph representation
  – Clear graph
    • Makes graph empty
  – Determine if graph is empty
  – Traverse graph
  – Print graph
Operations on Graphs (cont’d.)

• Graph representation in computer memory
  – Depends on specific application
• Use linked list representation of graphs
  – For each vertex $v$
    • Vertices adjacent to $v$ (directed graph: called immediate successors)
    • Stored in the linked list associated with $v$
• Managing data in a linked list
  – Use class unorderedLinkedList
• Labeling graph vertices
  – Depends on specific application
Graphs as ADTs

• See code on pages 692-693
  – Defines a graph as an ADT
  – Class specifying basic operations to implement a graph

• Definitions of the functions of the class graphType

```cpp
bool graphType::isEmpty() const
{
    return (gSize == 0);
}
```
Graphs as ADTs (cont’d.)

• **Function** `createGraph`  
  – Implementation  
    • Depends on how data input into the program  
    – See code on page 694

• **Function** `clearGraph`  
  – Empties the graph  
    • Deallocates storage occupied by each linked list  
    • Sets number of vertices to zero  
  – See code on page 695
Graph Traversals

• Processing a graph
  – Requires ability to traverse the graph

• Traversing a graph
  – Similar to traversing a binary tree
    • A bit more complicated

• Two most common graph traversal algorithms
  – Depth first traversal
  – Breadth first traversal
Depth First Traversal

• Similar to binary tree preorder traversal
• General algorithm

```plaintext
for each vertex, v, in the graph
  if v is not visited
    start the depth first traversal at v
```

**FIGURE 12-7** Directed graph $G_3$
Depth First Traversal (cont’d.)

• General algorithm for depth first traversal at a given node \( v \)
  – Recursive algorithm

1. mark node \( v \) as visited
2. visit the node
3. for each vertex \( u \) adjacent to \( v \)
   if \( u \) is not visited
   start the depth first traversal at \( u \)
Depth First Traversal (cont’d.)

- Function `dft` implements algorithm

```cpp
void graphType::dft(int v, bool visited[])
{
    visited[v] = true;
    cout << " " << v << " "; //visit the vertex

    linkedListIterator<int> graphIt;

    //for each vertex adjacent to v
    for (graphIt = graph[v].begin(); graphIt != graph[v].end(); ++graphIt)
    {
        int w = *graphIt;
        if (!visited[w])
            dft(w, visited);
    } //end while
} //end dft
```
Depth First Traversal (cont’d.)

• **Function** `depthFirstTraversal`  
  – Implements depth first traversal of the graph

```cpp
void graphType::depthFirstTraversal()
{
    bool *visited; // pointer to create the array to keep
                    // track of the visited vertices
    visited = new bool[gSize];

    for (int index = 0; index < gSize; index++)
        visited[index] = false;

    // For each vertex that is not visited, do a depth
    // first traversal
    for (int index = 0; index < gSize; index++)
        if (!visited[index])
            dft(index, visited);

    delete [] visited;
} // end depthFirstTraversal
```
Depth First Traversal (cont’d.)

- **Function** `depthFirstTraversal`
  - Performs a depth first traversal of entire graph
- **Function** `dftAtVertex`
  - Performs a depth first traversal at a given vertex

```cpp
void graphType::dftAtVertex(int vertex)
{
    bool *visited;

    visited = new bool[gSize];

    for (int index = 0; index < gSize; index++)
        visited[index] = false;

    dft(vertex, visited);

    delete [] visited;
} // end dftAtVertex
```
Breadth First Traversal

• Similar to traversing binary tree level-by-level
  – Nodes at each level
    • Visited from left to right
  – All nodes at any level $i$
    • Visited before visiting nodes at level $i+1$
Breadth First Traversal (cont’d.)

• General search algorithm
  – Breadth first search algorithm with a queue

  1. for each vertex \( v \) in the graph
     if \( v \) is not visited
        add \( v \) to the queue //start the breadth first search at \( v \)
  2. Mark \( v \) as visited
  3. while the queue is not empty
     3.1. Remove vertex \( u \) from the queue
     3.2. Retrieve the vertices adjacent to \( u \)
     3.3. for each vertex \( w \) that is adjacent to \( u \)
        if \( w \) is not visited
           3.3.1. Add \( w \) to the queue
           3.3.2. Mark \( w \) as visited
void graphType::breadthFirstTraversal()
{
    linkedQueueType<int> queue;

    bool *visited;
    visited = new bool[gSize];

    for (int ind = 0; ind < gSize; ind++)
        visited[ind] = false;  //initialize the array
        //visited to false

    linkedListIterator<int> graphIt;

    for (int index = 0; index < gSize; index++)
        if (!visited[index])
        {
            queue.addQueue(index);
            visited[index] = true;
            cout << " " << index << " ";

            while (!queue.isEmptyQueue())
            {
                int u = queue.front();
                queue.deleteQueue();

                for (graphIt = graph[u].begin();
                    graphIt != graph[u].end(); ++graphIt)
                {
                    int w = *graphIt;
                    if (!visited[w])
                    {
                        queue.addQueue(w);
                        visited[w] = true;
                        cout << " " << w << " ";
                    }
                }
            }  //end while
        }
    delete [] visited;
}  //end breadthFirstTraversal
Shortest Path Algorithm

• Weight of the graph
  – Nonnegative real number assigned to the edges connecting to vertices

• Weighted graphs
  – When a graph uses the weight to represent the distance between two places

• Weight of the path $P$
  – Given $G$ as a weighted graph with vertices $u$ and $v$ in $G$ and $P$ as a path in $G$ from $u$ to $v$
    • Sum of the weights of all the edges on the path

• Shortest path: path with the smallest weight
Shortest Path Algorithm (cont’d.)

• Shortest path algorithm space (greedy algorithm)
• See code on page 700

- \texttt{class weightedGraphType}
  • Extend definition of \texttt{class graphType}
  • Adds function \texttt{createWeightedGraph} to create graph and weight matrix associated with the graph

Let $G$ be a graph with $n$ vertices, where $n \geq 0$. Let $V(G) = \{v_1, v_2, \ldots, v_n\}$. Let $W$ be a two-dimensional $n \times n$ matrix such that

$$W(i,j) = \begin{cases} w_{ij} & \text{if } (v_i, v_j) \text{ is an edge in } G \text{ and } w_{ij} \text{ is the weight of the edge } (v_i, v_j) \\ \infty & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$
Shortest Path

• General algorithm
  – Initialize array \textit{smallestWeight}
    \begin{align*}
    \text{smallestWeight}[u] &= \text{weights}[\text{vertex}, u] \\
    \text{Set} \quad \text{smallestWeight}[\text{vertex}] &= \text{zero} \\
    \text{Find vertex } v \text{ closest to vertex where shortest path is not determined} \\
    \text{Mark } v \text{ as the (next) vertex for which the smallest weight is found}
    \end{align*}
Shortest Path (cont’d.)

• General algorithm (cont’d.)
  – For each vertex \( w \) in \( G \), such that the shortest path from vertex to \( w \) has not been determined and an edge \((v, w)\) exists
    • If weight of the path to \( w \) via \( v \) smaller than its current weight
    • Update weight of \( w \) to the weight of \( v \) + weight of edge \((v, w)\)
Shortest Path (cont’d.)

**FIGURE 12-8** Weighted graph $G$

**FIGURE 12-9** Graph after Steps 1 and 2 execute
Shortest Path (cont’d.)

FIGURE 12-10 Graph after the first iteration of Steps 3 to 5

FIGURE 12-11 Graph after the second iteration of Steps 3 to 5
Shortest Path (cont’d.)

**FIGURE 12-12** Graph after the third iteration of Steps 3 to 5

**FIGURE 12-13** Graph after the fourth iteration of Steps 3 through 5
Shortest Path (cont’d.)

• See code on pages 704-705
  – C++ function `shortestPath` implements previous algorithm
    • Records only the weight of the shortest path from the source to a vertex

• Review the definitions of the function `printShortestDistance` and the constructor and destructor on pages 705-706
Minimum Spanning Tree

• Airline connections of a company
  – Between seven cities

![Diagram of Minimum Spanning Tree](image)

**FIGURE 12-14** Airline connections between cities and the cost factor of maintaining the connections
Minimum Spanning Tree (cont’d.)

• Due to financial hardship
  – Company must shut down maximum number of connections
  • Still be able to fly (maybe not directly) from one city to another

FIGURE 12-15 Possible solutions to the graph of Figure 12-14
Minimum Spanning Tree (cont’d.)

• Free tree $T$
  – Simple graph
  – If $u$ and $v$ are two vertices in $T$
    • Unique path from $u$ to $v$ exists

• Rooted tree
  – Tree with particular vertex designated as a root
Minimum Spanning Tree (cont’d.)

• Weighted tree $T$
  – Weight assigned to edges in $T$
  – Weight denoted by $W(T)$: sum of weights of all the edges in $T$

• Spanning tree $T$ of graph $G$
  – $T$ is a subgraph of $G$ such that $V(T) = V(G)$
Minimum Spanning Tree (cont’d.)

• Theorem 12-1
  – A graph $G$ has a spanning tree if and only if $G$ is connected
  – From this theorem, it follows that to determine a spanning tree of a graph
    • Graph must be connected
• Minimum (minimal) spanning tree of $G$
  – Spanning tree with the minimum weight
Minimum Spanning Tree (cont’d.)

• Two well-known algorithms for finding a minimum spanning tree of a graph
  – Prim’s algorithm
    • Builds the tree iteratively by adding edges until a minimum spanning tree obtained
  – Kruskal’s algorithm
Minimum Spanning Tree (cont’d.)

• General form of Prim’s algorithm

1. Set $V(T) = \{\text{source}\}$
2. Set $E(T) = \text{empty}$
3. for $i = 1$ to $n$
   3.1. $\text{minWeight} = \text{infinity}$;
   3.2. for $j = 1$ to $n$
       if $v_j$ is in $V(T)$
           for $k = 1$ to $n$
               if $v_k$ is not in $T$ and $\text{weight}[v_j, v_k] < \text{minWeight}$
                   
                   \[
                   \begin{align*}
                   \text{endVertex} &= v_k; \\
                   \text{edge} &= (v_j, v_k); \\
                   \text{minWeight} &= \text{weight}[v_j, v_k]; \\
                   \end{align*}
                   \]
           end
       end
   end
3.3. $V(T) = V(T) \cup \{\text{endVertex}\}$;
3.4. $E(T) = E(T) \cup \{\text{edge}\}$;

FIGURE 12-16 Weighted graph $G$
Minimum Spanning Tree (cont’d.)

• See code on page 710
  - class msTreeType defines spanning tree as an ADT

• See code on page 712
  - C++ function minimumSpanning implementing Prim’s algorithm
  - Prim’s algorithm given in this section: $O(n^3)$
    • Possible to design Prim’s algorithm order $O(n^2)$

• See function printTreeAndWeight code
• See constructor and destructor code
FIGURE 12-17 Graph $G$, $V(T)$, $E(T)$, and $N$ after Steps 1 and 2 execute
Topological Order

• Topological ordering of $V(G)$
  – Linear ordering $v_{i1}, v_{i2}, \ldots, v_{in}$ of the vertices such that
    • If $v_{ij}$ is a predecessor of $v_{ik}$, $j \neq k$, $1 \leq j \leq n$, $1 \leq k \leq n$
    • Then $v_{ij}$ precedes $v_{ik}$, that is, $j < k$ in this linear ordering

• Algorithm topological order
  – Outputs directed graph vertices in topological order
  – Assume graph has no cycles
    • There exists a vertex $v$ in $G$ such that $v$ has no successor
    • There exists a vertex $u$ in $G$ such that $u$ has no predecessor
Topological Order (cont’d.)

• Topological sort algorithm
  – Implemented with the depth first traversal or the breadth first traversal

• Extend `class graphType` definition (using inheritance)
  – Implement breadth first topological ordering algorithm
    • Called `class topologicalOrderType`
  – See code on pages 714-715
    • Illustrating class including functions to implement the topological ordering algorithm
Breadth First Topological Ordering

• General algorithm

1. Create the array $\text{predCount}$ and initialize it so that $\text{predCount}[i]$ is the number of predecessors of the vertex $v_i$.
2. Initialize the queue, say $\text{queue}$, to all those vertices $v_k$ so that $\text{predCount}[k]$ is 0. (Clearly, $\text{queue}$ is not empty because the graph has no cycles.)
3. while the $\text{queue}$ is not empty
   3.1. Remove the front element, $u$, of the $\text{queue}$.
   3.2. Put $u$ in the next available position, say $\text{topologicalOrder[topIndex]}$, and increment $\text{topIndex}$.
   3.3. For all the immediate successors $w$ of $u$,
      3.3.1. Decrement the predecessor count of $w$ by 1.
      3.3.2. if the predecessor count of $w$ is 0, add $w$ to $\text{queue}$. 
Breadth First Topological Ordering (cont’d.)

- Breadth First Topological order
  - 0 9 1 7 2 5 4 6 3 8 10

**FIGURE 12-18** Arrays `predCount`, `topologicalOrder`, and `queue` after Steps 1 and 2 execute

**FIGURE 12-19** Arrays `predCount`, `topologicalOrder`, and `queue` after the first iteration of Step 3
FIGURE 12-20 Arrays predCount, topologicalOrder, and queue after the second iteration of Step 3

FIGURE 12-21 Arrays predCount, topologicalOrder, and queue after the third iteration of Step 3
Breadth First Topological Ordering (cont’d.)

- See code on pages 718-719
  - Function implementing breadth first topological ordering algorithm

![FIGURE 12-22 Arrays predCount, topologicalOrder, and queue after Step 3 executes](image)
Euler Circuits

• Euler’s solution to Königsberg bridge problem
  – Reduces problem to finding circuit in the graph

• Circuit
  – Path of nonzero length
    • From a vertex $u$ to $u$ with no repeated edges

• Euler circuit
  – Circuit in a graph including all the edges of the graph

• Eulerian graph $G$
  – If either $G$ is a trivial graph or $G$ has an Euler circuit
Euler Circuits (cont’d.)

• Graph of Figure 12-24: Euler circuit

FIGURE 12-23 A graph with all vertices of odd degree

FIGURE 12-24 A graph with all vertices of even degree
Euler Circuits (cont’d.)

• Theorem 12-2
  – If a connected graph $G$ is Eulerian, then every vertex of $G$ has even degree

• Theorem 12-3
  – Let $G$ be a connected graph such that every vertex of $G$ is of even degree; then, $G$ has an Euler circuit

**FIGURE 12-25** Graph of the Königsberg bridge problem with two additional bridges
Euler Circuits (cont’d.)

• Fleury’s Algorithm

Step 1. Choose a vertex $v$ as the starting vertex for the circuit and choose an edge $e$ with $v$ as one of the end vertices.

Step 2. If the other end vertex $u$ of the edge $e$ is also $v$, go to Step 3. Otherwise, choose an edge $e_1$ different from $e$ with $u$ as one of the end vertices. If the other vertex $u_1$ of $e_1$ is $v$, go to Step 3; otherwise, choose an edge $e_2$ different from $e$ and $e_1$ with $u_1$ as one of the end vertices and repeat Step 2.

Step 3. If the circuit $T_1$ obtained in Step 2 contains all the edges, then stop. Otherwise, choose an edge $e_j$ different from the edges of $T_1$ such that one of the end vertices of $e_j$, say, $w$ is a member of the circuit $T_1$.

Step 4. Construct a circuit $T_2$ with starting vertex $w$, as in Steps 1 and 2, such that all the edges of $T_2$ are different from the edges in the circuit $T_1$.

Step 5. Construct the circuit $T_3$ by inserting the circuit $T_2$ at $w$ of the circuit $T_1$. Now go to Step 3 and repeat Step 3 with the circuit $T_3$. 
Euler Circuits (cont’d.)

• Fleury’s Algorithm (cont’d.)

FIGURE 12-26 A graph with all vertices of even degree
Summary

• Many types of graphs
  – Directed, undirected, subgraph, weighted
• Graph theory borrows set theory notation
• Graph representation in memory
  – Adjacency matrices, adjacency lists
• Graph traversal
  – Depth first, breadth first
• Shortest path algorithm
• Prim’s algorithm
• Euler circuit