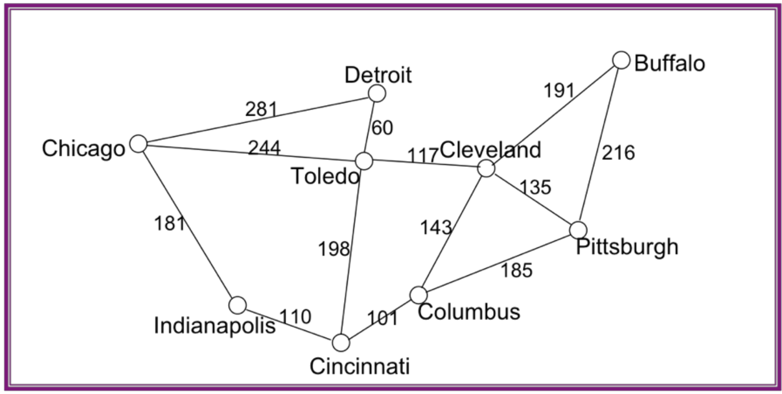
Graph Traversal

EECS 233

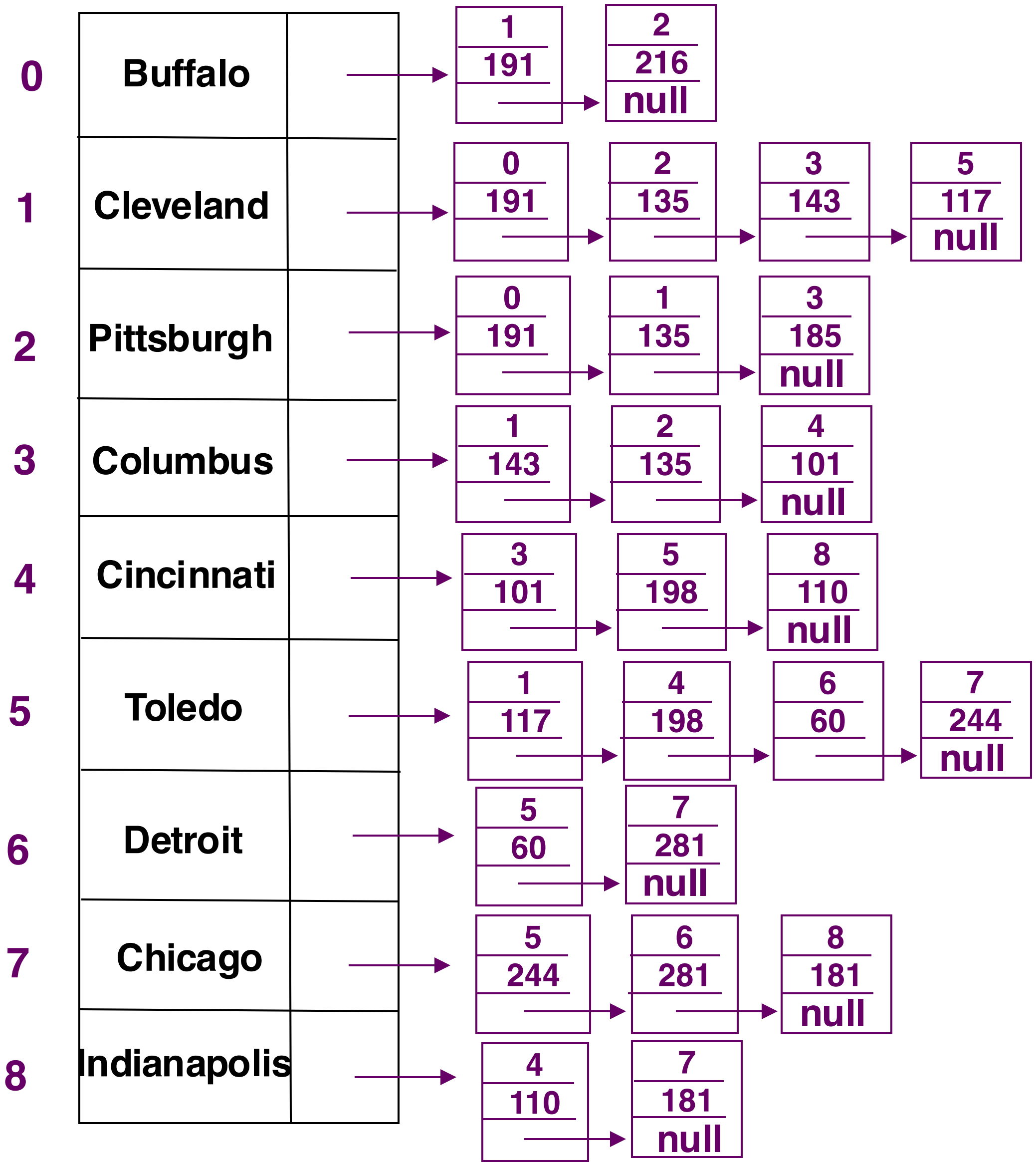
Graph Traversals

- Traversing a graph involves starting at some vertex and visiting all of the vertices that can be reached from that vertex.
  - visiting a vertex = processing its data in some way
  - if the graph is connected, all of the vertices will be visited



- Example:
  - A Web crawler (vertices = pages, edges = links)

- We will consider two types of traversals:
  - **depth-first**:
    - Visit the starting vertex
    - Proceed as far as possible along a given path (via a neighbor) before “backtracking” and going along the next path
  - **breadth-first**:
    - visit the starting vertex
    - visit all of its neighbors
    - visit all unvisited vertices 2 edges away
    - visit all unvisited vertices 3 edges away, etc.

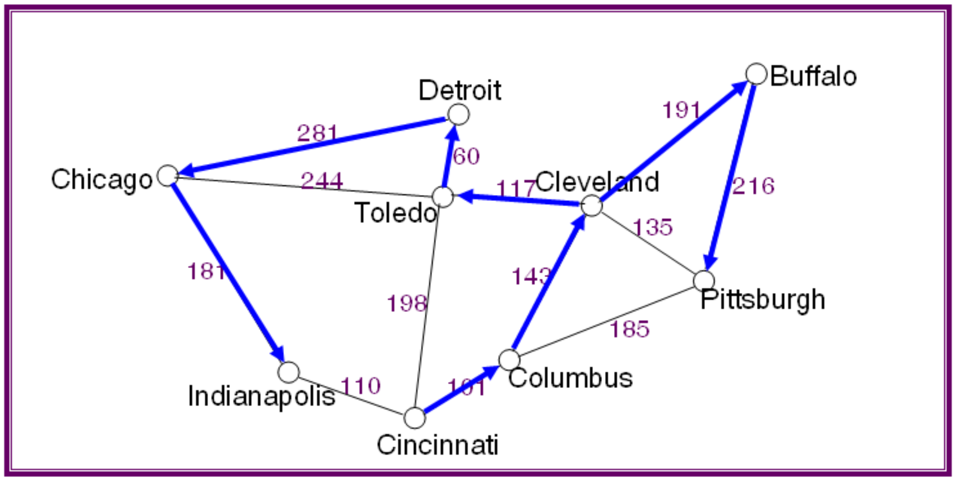

0 Buffalo
1 191
2 216 null
1 Cleveland
0 191
2 135
3 143
5 117 null
2 Pittsburgh
0 191
1 135
3 185 null
3 Columbus
1 143
2 135
4 101 null
4 Cincinnati
3 101
5 198
8 110 null
5 Toledo
1 117
4 198
6 60
7 244 null
6 Detroit
5 60
7 281 null
7 Chicago
5 244
6 281
8 181 null
8 Indianapolis
4 110
7 181 null

Depth-first Traversal

- Visit a vertex, then make recursive calls on all of its yet-to-be-visited neighbors:

depthFirstTrav(v)
    myDepthFirstTrav(v, NULL)

myDepthFirstTrav(node, parent)
    visit node and mark it as visited
    node.parent = parent
    for each vertex w in node's neighbors
        if (w has not been visited)
            myDepthFirstTrav(w, node)




```
public class Graph {
        class Vertex {
                private String id;
                private linkedList <Edge> edges;
                    // adjacency list
                private boolean encountered;
                private boolean done;
                private Vertex parent;
                private double cost;
                …
        }
        class Edge {
                private int endNode;
                private double cost;
                …
        }
        private Vertex[] vertices;
        private int numVertices;
        private int maxNum;
        …
}
```

Depth-first Traversal

- Visit a vertex, then make recursive calls on all of its yet-to-be-visited neighbors:

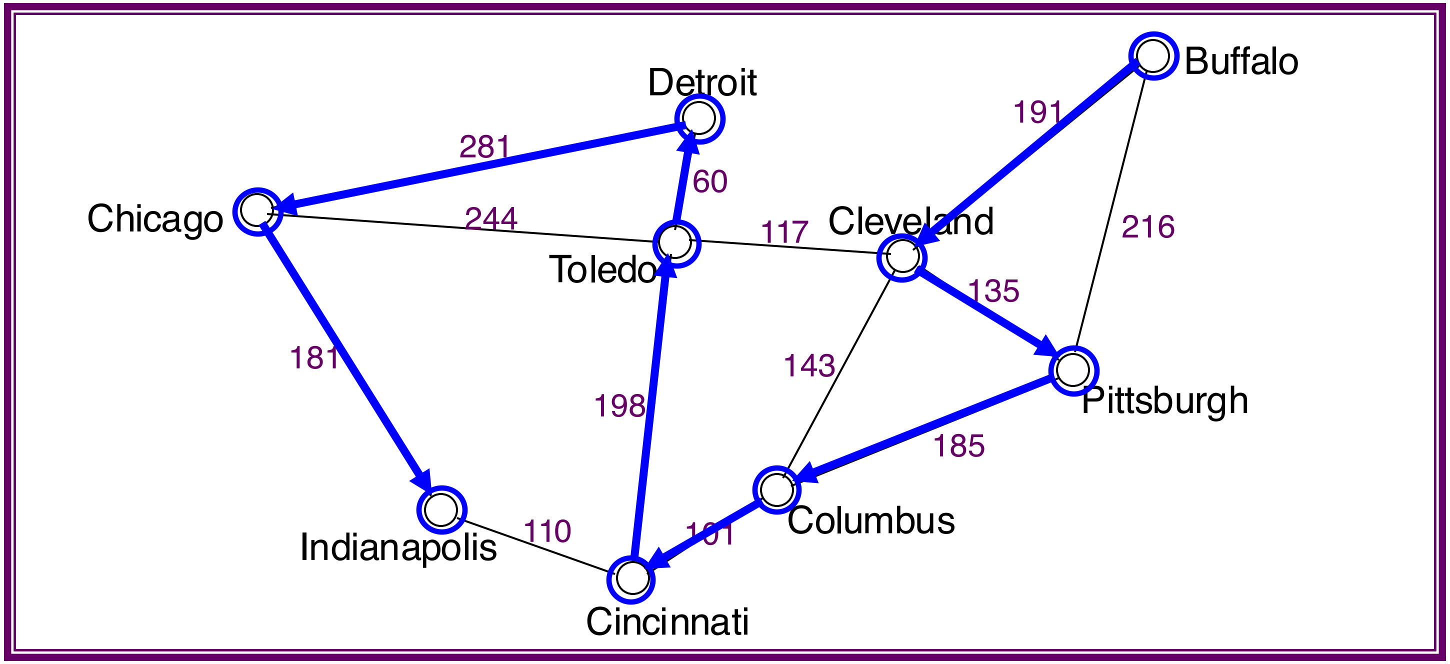
```
myDepthFirstTrav(v, parent)
        visit v and mark it as visited
        v.parent = parent
        for each vertex w in v's neighbors
                if (w has not been visited)
                        myDepthFirstTrav(w, v)
```

- Implementation:

```
public void myDepthFirstTrav(int i, int parent) {
        System.output.println(vertices[i].id);
        vertices[i].encountered = true;
        vertices[i].parent = parent;
        Iterator<Edge> edgeItr = edges.iterator();
        while (edgeItr.hasNext()) {
                Edge curEdge = edgeItr.next()
                j = curEdge.endNode;
                if (vertices[j].encountered == false)
                        myDepthFirstTrav(j, i);
        }
}
```

```
public class Graph {
        class Vertex {
                private String id;
                private linkedList <Edge> edges;
                   // adjacency list
                private boolean encountered;
                private boolean done;
                private Vertex parent;
                private double cost;
                …
        }
        class Edge {
                private int endNode;
                private double cost;
                …
        }
        private Vertex[] vertices;
        private int numVertices;
        private int maxNum;
        …
}
```

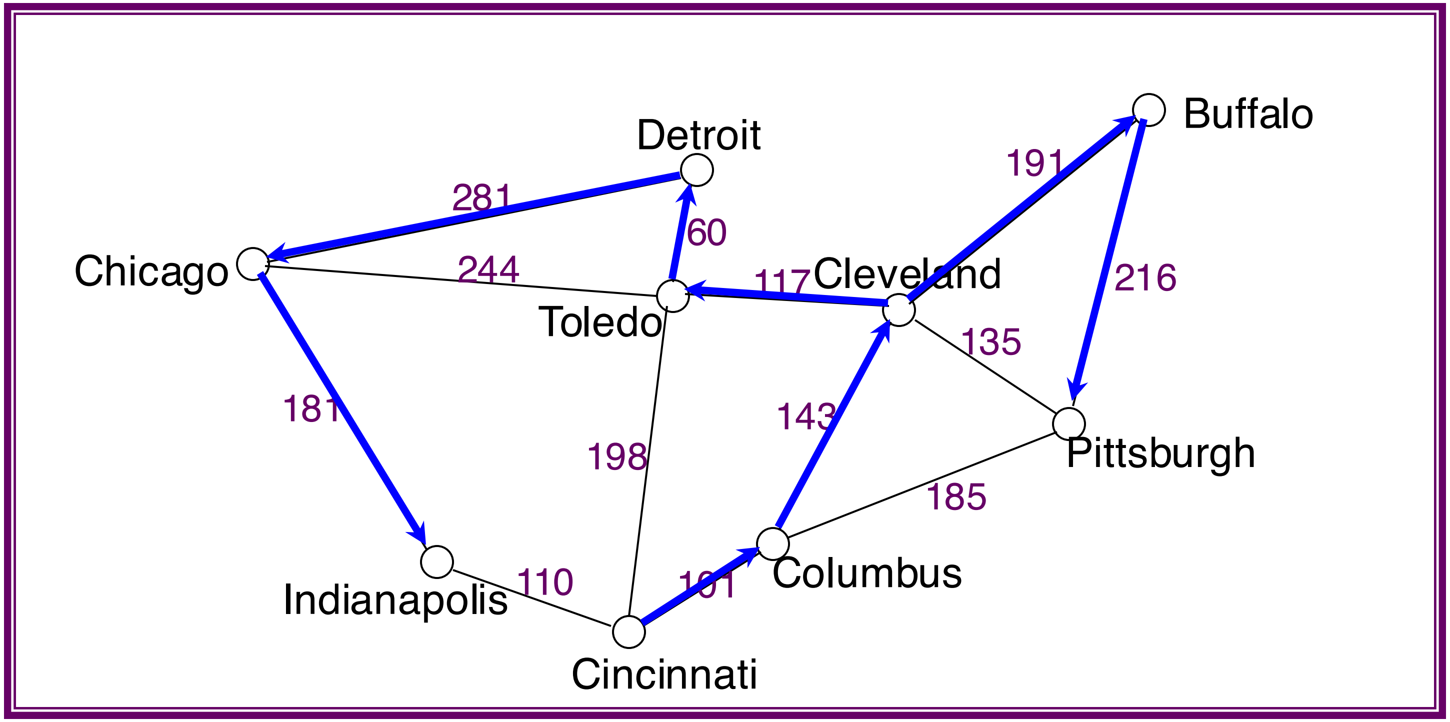
Example: From Buffalo



myDepthFirstTrav("Buffalo", null)
    visit Buffalo; Buffalo.parent = NULL
    w = "Cleveland"
    myDepthFirstTrav("Cleveland", "Buffalo")
        visit Cleveland; Cleveland.parent = Buffalo
        w = "Pittsburgh"
        myDepthFirstTrav("Pittsburgh", "Cleveland")
            visit Pittsburg; Pittsburg.parent = Cleveland
            w = "Buffalo"; Buffalo has been visited
            w = "Columbus"
            myDepthFirstTrave("Columbus", "Pittsburgh")
                visit Columbus; Columbus.parent = Pittsburg;
                w = "Cincinnati"
                myDepthFirstTrav("Cincinnati", "Columbus")
                …

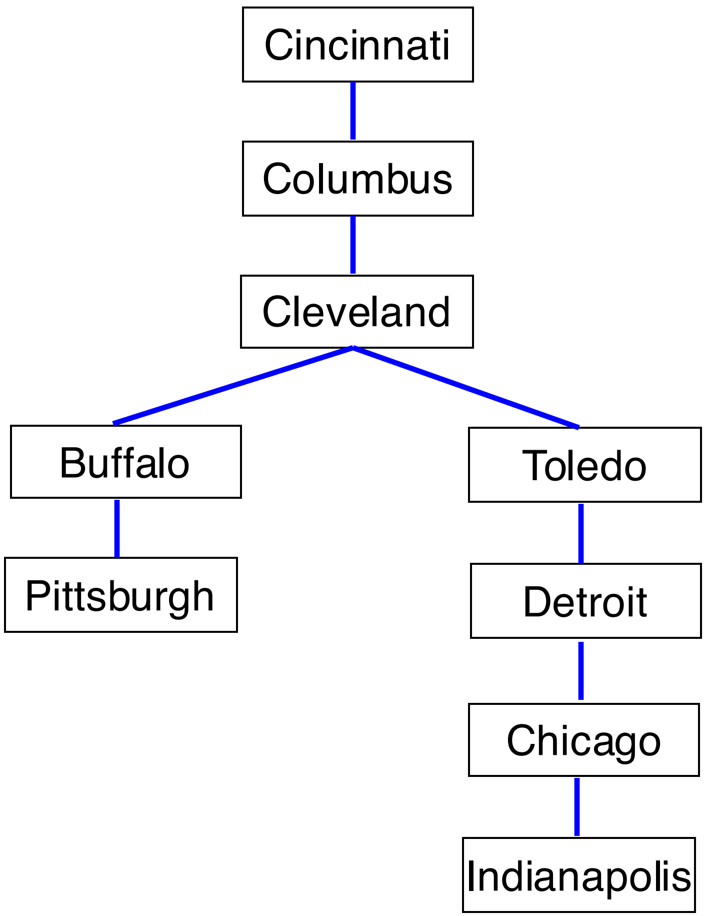
myDeapthFirstTrav(v, parent)
    visit v and mark it as visited
    v.parent = parent
    for each vertex w in v's neighbors
        if (w has not been visited)
            myDeapthFirstTrav(w, v)

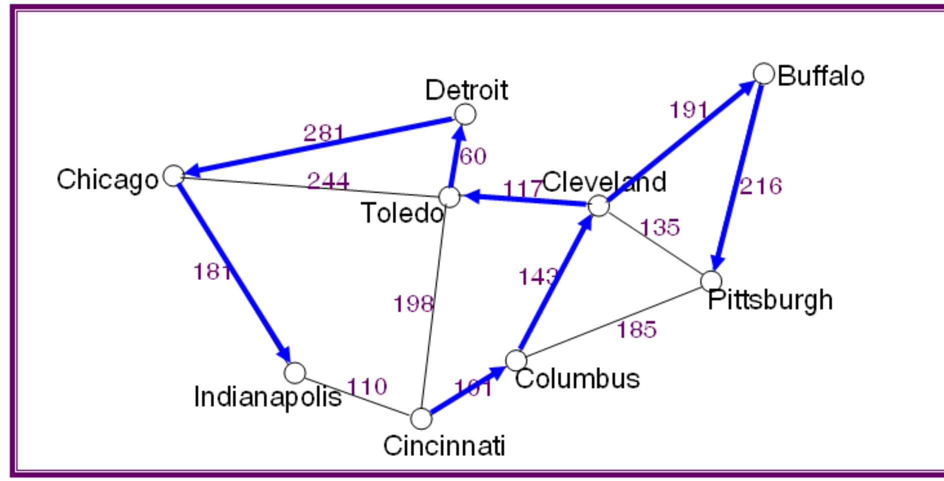
Example: From Cincinnati



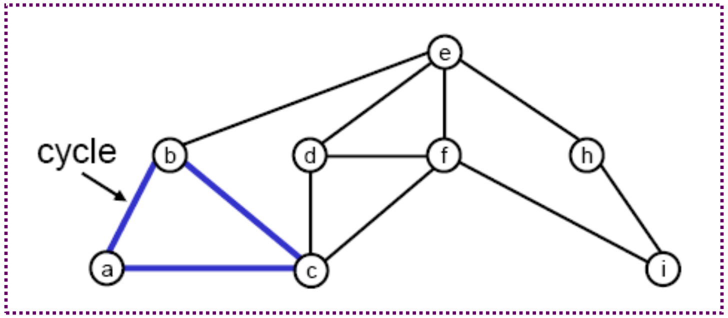
Order: Cincinnati, Columbus, Cleveland, Buffalo, Pittsburg, Toledo, Detroit, Chicago, Indianapolis

Depth-First Spanning Trees





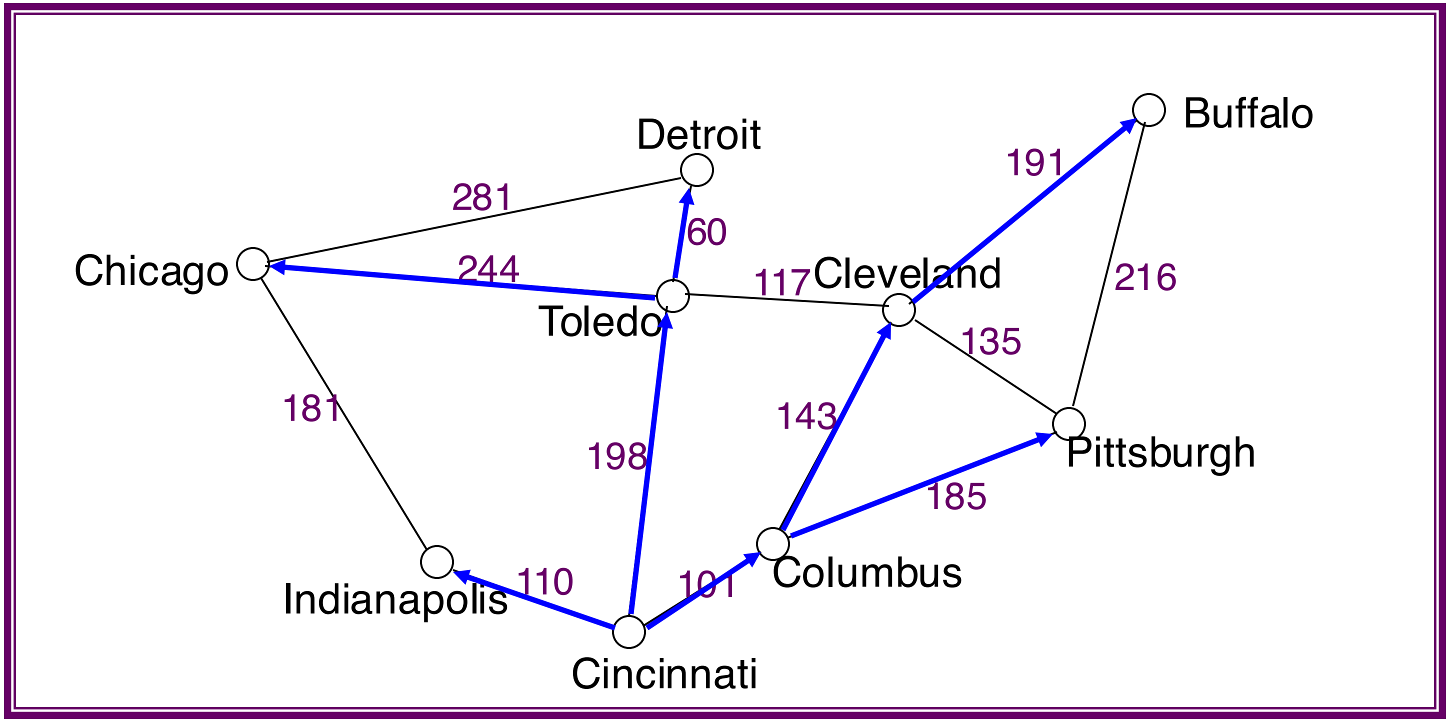
An Application: Checking for Cycles



To discover a cycle:

- perform a depth-first traversal
  - when considering the non-parent neighbors of a current vertex, if we discover one that is already marked as visited, there is a cycle
- If we discover no cycles during the course of the traversal, the graph is acyclic.

Breadth-First Traversal



Order: Cincinnati, Columbus, Toledo, Indianapolis, Pittsburgh, Cleveland, Detroit, Chicago, Buffalo

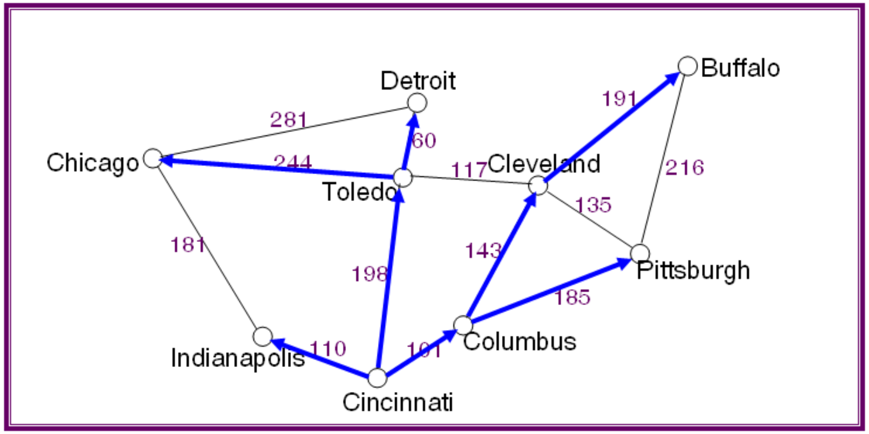
- Starting from Cincinnati, what would be the order of visits?

- **breadth-first**:
  - visit a vertex
  - visit all of its neighbors
  - visit all unvisited vertices 2 edges away
  - visit all unvisited vertices 3 edges away, etc.

Breadth-First Traversal Method (pseudo code)

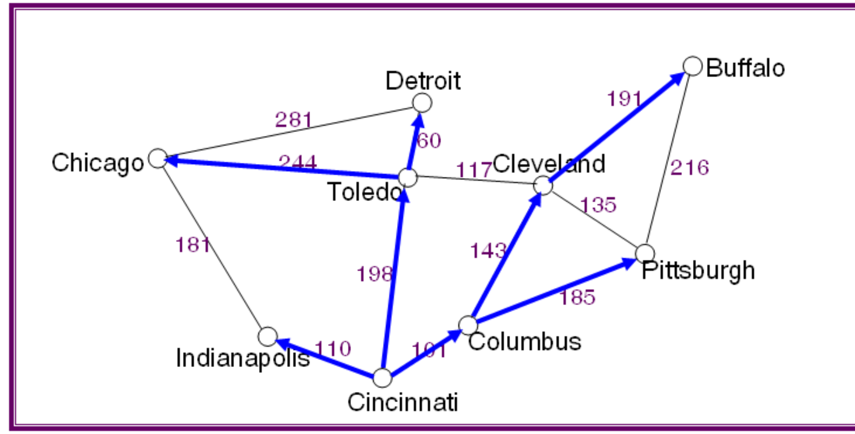
- Use a queue, as we did for level-order tree traversal:

```
bfTrav(origin)

        origin.parent = null
        create a new queue q
        q.insert(origin)

        while (!q.isEmpty())
                v = q.remove()
                visit v

                for each vertex w in v's neighbors
                        if w is not encountered
                            mark w as encountered
                            w.parent = v;
                            q.insert(w);
```



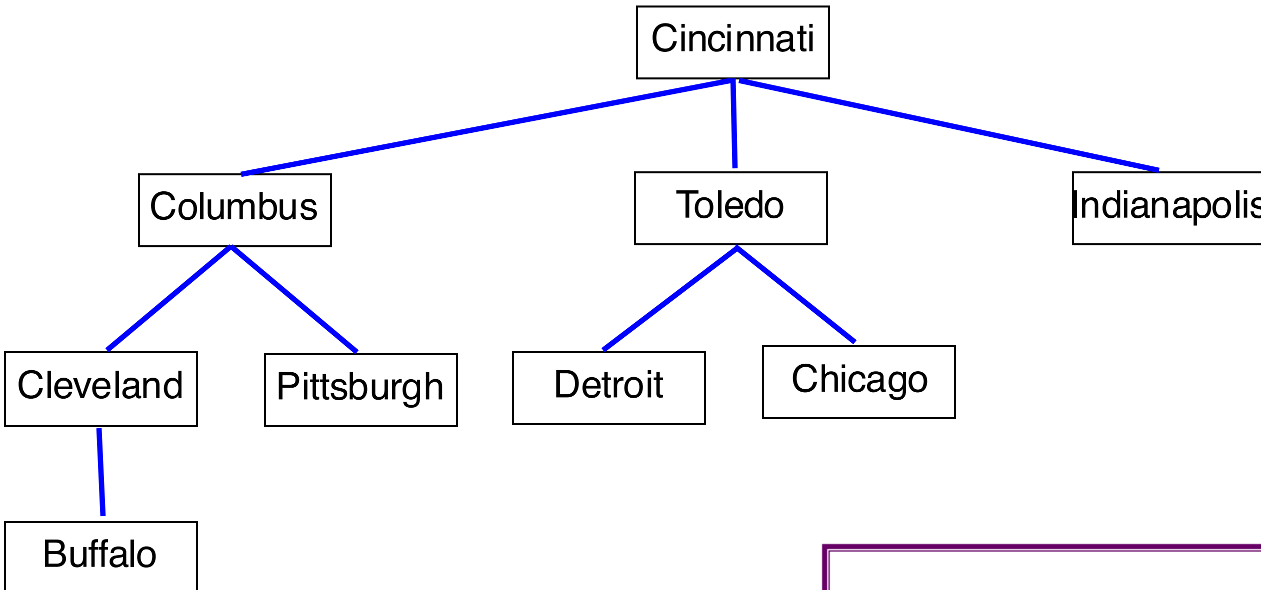
Example: From Cincinnati

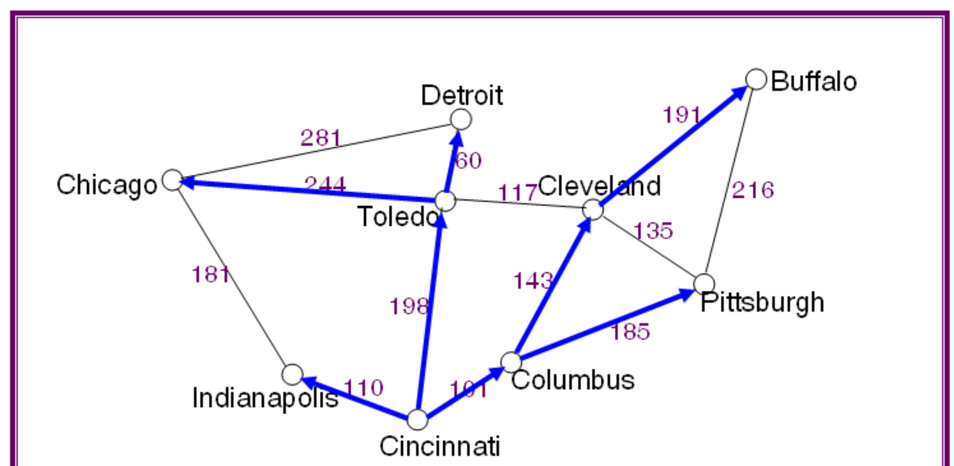
- Tracing the queue operations
  insert(“Cincinnati”);
  remove(“Cincinnati”); // and print it
  insert(“Columbus”);
  insert(“Toledo”);
  insert(“Indianapolis”);
  remove(“Columbus”); // and print it
  insert(“Cleveland”);
  insert(“Pittsburgh”);
  remove(“Toledo”); // and print it
  insert(“Detroit’);
  insert(“Chicago”);
  remove(“Indianapolis”); // and print it
  remove(“Cleveland”); // and print it
  insert(“Buffalo”);
  remove(“Pittsburgh”); // and print it
  remove(“Detroit”); // and print it
  remove(“Chicago”); // and print it
  remove(“Buffalo”); // and print it



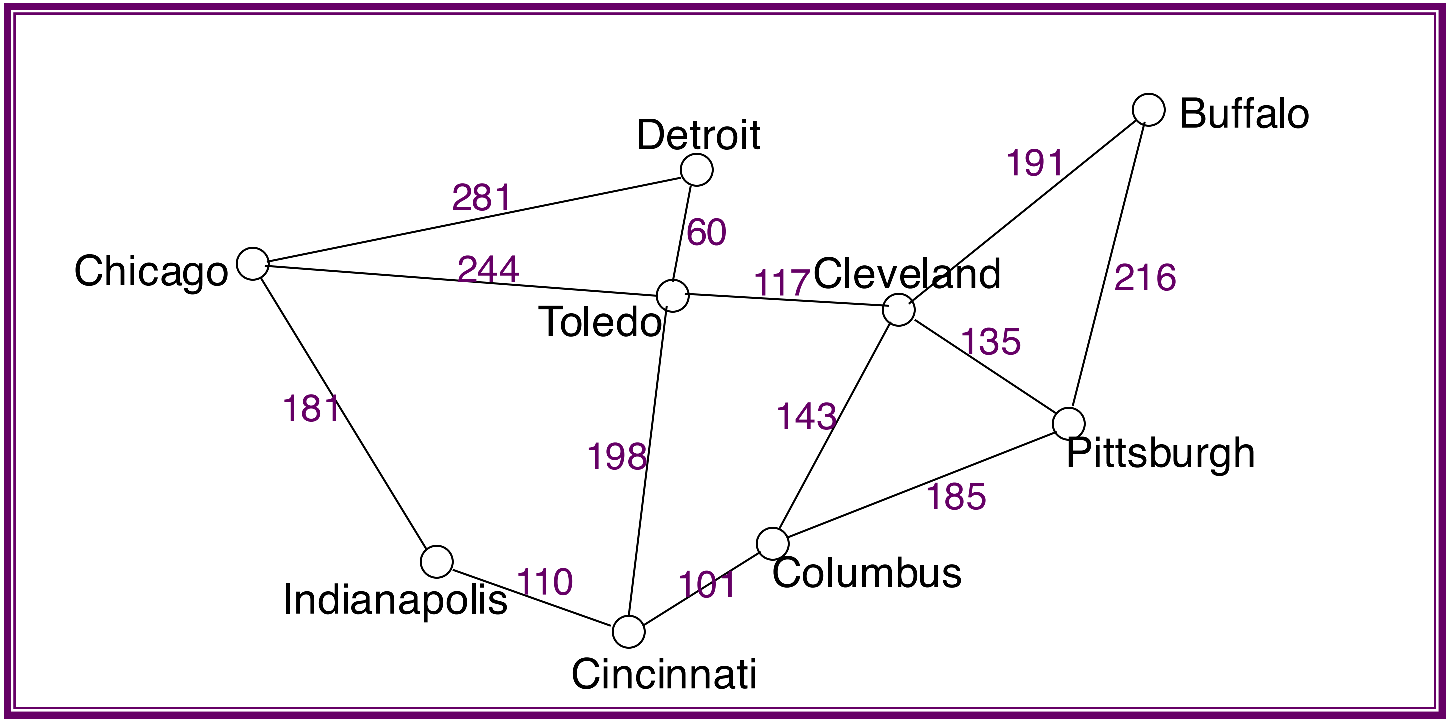

```
bfTrav(origin)

    origin.parent = null
    create a new queue q
    q.insert(origin)

    while (!q.isEmpty())
        v = q.remove()
        visit v

        for each vertex w in v's neighbors
            if w is not encountered
                mark w as encountered
                w.parent = v;
                q.insert(w);
```

Breadth-First Spanning Trees





Example: From Cleveland



Running Time of Graph Traversal

- Let V = number of vertices in the graph, and E = number of edges
- Assume we use an adjacency list as the data structure
- A traversal requires $O(V + E)$ steps.
  - visit each vertex once
  - traverse each vertex's adjacency list at most once
    - the total length of the adjacency lists is $2E = O(E)$
  - for a dense graph, $E \rightarrow V^2$, so the worst-case bound is $O(V^2)$
  - for a sparse graph, $O(V + E) << O(V^2)$