Sections marked with a star (★) are not covered in lecture but are instead covered in the online lecture notes. Students are responsible for all material covered in lecture and in the online lecture notes. Material from the online lecture notes will definitely be assessed in the prelim and final exam.

Copyright © 2019 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
1. Graph Concepts
2. Graph Storage
3. Directed Graphs

- Object-oriented adjacency-list-based graph with int weights
- Could extend this to be polymorphic for edge and vertex info
  - use object-oriented programming and dynamic polymorphism
  - use generic programming and static polymorphism
- Could add iterators to improve data encapsulation
- Could use functional programming to inspect/transform graph
- Could use concurrent programming to traverse graph in parallel

```cpp
class GraphInt
{
    public:

    int add_vertex();
    void add_edge( int src_id, int dest_id, int weight );
    Vector<int> get_neighbors( int node_id );
    int get_weight( int src_id, int dest_id );

    Vector<int> dfs( int src_id, int dest_id );
    Vector<int> bfs( int src_id, int dest_id );

    Vector< Vector< Pair<int,int> > > m_graph;
};
```
int GraphInt::add_vertex()
{
    m_graph.push_back( Vector<Pair<int,int>>() );
    return m_graph.size() - 1;
}

void GraphInt::add_edge( int src_id, int dest_id, int weight )
{
    m_graph.at(src_id).push_back( 
        Pair<int,int>(dest_id,weight) );
}

Vector<int> GraphInt::get_neighbors( int node_id )
{
    Vector<int> neighbors;
    for ( auto e : m_graph.at(node_id) )
        neighbors.push_back( e.first );
    return neighbors;
}

int GraphInt::get_weight( int src_id, int dest_id )
{
    for ( auto e : m_graph.at(src_id) )
        if ( e.first == dest_id )
            return e.second;
    assert(false);
}
Draw the conceptual graph and the adjacency list storage resulting from this code sequence:

```java
GraphInt g;

int v0 = g.add_vertex();
int v1 = g.add_vertex();
int v2 = g.add_vertex();
int v3 = g.add_vertex();
int v4 = g.add_vertex();
int v5 = g.add_vertex();
int v6 = g.add_vertex();

g.add_edge( v0, v1, 1 );
g.add_edge( v0, v2, 1 );
g.add_edge( v0, v3, 1 );
g.add_edge( v1, v6, 1 );
g.add_edge( v2, v4, 1 );
g.add_edge( v3, v5, 1 );
g.add_edge( v4, v6, 1 );
g.add_edge( v5, v4, 1 );
```
3.1. Depth-First Search Traversal

```c++
Vector<int> GraphInt::dfs( int src_id, int dest_id )
{
    Stack<Vector<int>> worklist; // pending paths to search
    Set<int> visited; // nodes already visited

    // Initialize worklist with path containing just source node
    Vector<int> p; p.push_back(src_id); worklist.push( p );

    // Keep working until worklist is empty
    while ( worklist.size() != 0 ) {
        // Pop path from _top_ of stack
        auto path = worklist.pop();

        // Check if final vertex in current path is destination
        int vertex = path.at( path.size()-1 );
        if ( vertex == dest_id )
            return path;

        // Check if final vertex has already been visited
        if ( visited.contains( vertex ) ) {
            // Mark vertex as visited
            visited.add( vertex );

            // Iterate through neighbors
            auto neighbors = get_neighbors( vertex );
            for ( int v : neighbors ) {
                // Create temporary new path with neighbor at end
                auto temp = path;
                temp.push_back(v);

                // Push this new path onto _top_ of stack
                worklist.push( temp );
            }
        }
    }

    // Should only get here if we could not find destination
    assert(false);
}
```
Show the current path for each iteration of the \texttt{while} loop

1. Directed Graphs

3.1. Depth-First Search Traversal
3.2. Breadth-First Search Traversal

```cpp
Vector<int> GraphInt::dfs( int src_id, int dest_id )
{
    Queue<Vector<int>> worklist; // pending paths to search
    Set<int> visited; // nodes already visited

    // Initialize worklist with path containing just source node
    Vector<int> p; p.push_back(src_id); worklist.push( p );

    // Keep working until worklist is empty
    while ( worklist.size() != 0 ) {
        auto path = worklist.deq();

        // Check if final vertex in current path is destination
        int vertex = path.at( path.size()-1 );
        if ( vertex == dest_id )
            return path;

        // Check if final vertex has already been visited
        if ( visited.contains( vertex ) ) {
            // Mark vertex as visited
            visited.add( vertex );

            // Iterate through neighbors
            auto neighbors = get_neighbors( vertex );
            for ( int v : neighbors ) {
                // Create temporary new path with neighbor at end
                auto temp = path;
                temp.push_back(v);

                // Enqueue this path on _tail_ of queue
                worklist.enq( temp );
            }
        }
    }

    // Should only get here if we could not find destination
    assert(false);
}
```
Show the current path for each iteration of the while loop