

# CS 225 Spring 2019 :: TA Lecture Notes

## 4/10 Graphs II

By Wenjie

---

### Graph ADT

- **Data:** all vertices, all edges, and the structure to maintain relations between vertices and edges.
- **Functions:**
  - `insert vertex/edge,`
  - `remove vertex/edge,`
  - `find incident edges,`
  - `check if two vertices are adjacent,` and
  - In case of directed graph find origin/destination.

### Graph implementation 1: Edge List

- **Vertex collection:** Use a hash table (find/remove/insert will be  $O(1)$ ).
- **Edge collections:** Use a linked list (hash table is not good because we have many collisions (no random distribution, violates SUHA) )
- **Running time:**
  - `Insert vertex` → we are using hash table where insert takes  $O(1)$  time.
  - `Remove vertex` → removing from hash table takes  $O(1)$ , but we need to remove incident edges which means we need to loop over edges list. We have  $m$  edges so it will take  $O(m)$
  - `areAdjacent` → again, we need to loop over the edge list which takes  $O(m)$  time.
  - `InsertEdge` → add edge to edge list by adding to the front so it takes  $O(1)$
  - `incidentEdges` →  $O(m)$ .
  - The running times seem linear however, we know that the relationship between number of nodes and the number of edges could be  $n^2$ ; which means  $O(m)$  could in fact be  $O(n^2)$

### Graph implementation 2: Adjacency Matrix

- Maintain a hash table of vertices and a list of edges.
- Add an  $n \times n$  matrix → store a pointer to the edge in edge list for every index in the matrix where the two vertices are adjacent.

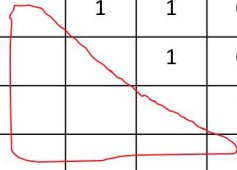
# CS 225 Spring 2019 :: TA Lecture Notes

## 4/10 Graphs II

By Wenjie

---

	u	v	w	z
u		1	1	0
v			1	0
w				1
z				



- Number one denotes a pointer to the edge in the edge list when two nodes are adjacent.
- We do not care about the bottom triangle (in red) if the graph is not directed

- **Insert data into the matrix**
  - If the table is full, we need to double the size in both dimensions which takes  $O(n^2)$  time.
  - We have to expand every  $n$  inserts, thereby on average resizing is  $O(n)$  amortized  $\rightarrow O^*(n)$ . Overall, insertion takes  $O^*(n)$ .
- **Remove a vertex**
  - Remove from hash table  $\rightarrow O(1)$ .
  - Remove instance edges: Loop over all rows and columns of that vertex which takes  $O(n)$ 
    - We will also have an awkward gap in the middle of the matrix after removing a row and a column. So we swap with the empty row/column with the last row/column.
  - Total running time is  $O(n) + O(1) = O(n)$ .
- incidentEdges  $\rightarrow$  we need to loop over row/column which takes  $O(n)$ .
- InsertEdge  $\rightarrow O(1)$
- Find/check adjacent vertices takes  $O(1) \rightarrow$  just a table lookup.
- **Space** complexity is  $O(n^2)$ .