# Data Structures

## Lecture 13.1: Graph (cont.)

Nopadon Juneam
Department of Computer Science
Kasetsart university

# Outlines

- Conclusions about graph representations
- Adjacency list implementation in C++ using STL

# Graph Representations: Complexity of Operations

| Operations | Adjacency matrix | Adjacency list |
|---|---|---|
| `createGraph`<br>(space to store graph) | $O(\|V\|^2)$ | $O(\|V\|)$ |
| `addEdge` | $O(1)$ | $O(\|V\|)$ |
| `addVertex` | $O(\|V\|^2)$ | $O(\|V\|)$ |
| `removeVertex` | $O(\|V\|^2)$ | $O(\|E\|)$ |
| `removeEdge` | $O(1)$ | $O(\|V\|)$ |
| `isAdjacent` | $O(1)$ | $O(\|V\|)$ |
| `inDegree` | $O(\|V\|)$ | $O(\|V\|+\|E\|)$ |
| `outDegree` | $O(\|V\|)$ | $O(\|V\|)$ |
| *Remarks* | Slow to add/remove vertices as matrix must be resized/copied | Slow to remove edges because it needs to iterate all the adjacent vertices |

# Graph Representations: Pros & Cons

- Remarks:

  - ***Adjacency matrix:*** Slow to add/remove vertices because matrix must be resized/copied

  - **Adjacency list**: Slow to remove edges because it needs to iterate through all the adjacent vertices

- *Conclusions:* Adjacency list is generally preferred if the graph is *sparse,* i.e., when $|E| \approx |V|$; Adjacency matrix is preferred if the graph is *dense,* i.e., when $|E| \approx |V|^2$

# Basic Graph Operations Using Adjacency-List Representation (1)

```
// A simple adjacency-list representation of graph using STL
#include <iostream>
#include <vector>
using namespace std;
// Function to create a graph with n vertices
vector<int>* createGraph(int n)
{
    // Return array of n lists (vectors)
    return new vector<int>[n];
}
// Function to add a directed edge into the graph
void addEdge(vector<int>* adjList, int u, int v)
{
    adjList[u].push_back(v);
}
// Function to print the adjacency-list representation of graph
void printGraph(vector<int>* adjList, int V)
{
    for (int v = 0; v < V; ++v)
    {
        cout << "[" << v << "] head ";
        for(int i=0; i < adjList[v].size(); i++)
            cout << "-> " << adjList[v].at(i);
        cout << endl;
    }
    cout << endl;
}
```

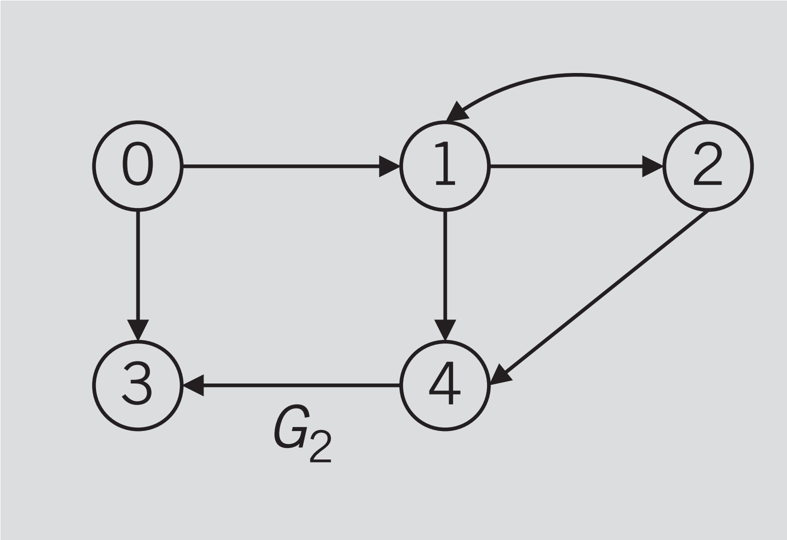
# Basic Graph Operations Using Adjacency-List Representation (2)

```
// Driver code
int main()
{
    int n = 5;

    vector<int>* adjList = createGraph(n);

    //Vertex numbers should be from 0 to 4.
    addEdge(adjList, 0, 1);
    addEdge(adjList, 0, 3);
    addEdge(adjList, 1, 2);
    addEdge(adjList, 1, 4);
    addEdge(adjList, 2, 1);
    addEdge(adjList, 2, 4);
    addEdge(adjList, 4, 3);

    printGraph(adjList, n);
    return 0;
}
```




```
Lasalle:codes dmodify$ ./
a.out
[0] head -> 1-> 3
[1] head -> 2-> 4
[2] head -> 1-> 4
[3] head
[4] head -> 3
```

# Operation: createGraph

- `createGraph(n)`: create the empty graph with *n* isolated vertices

```
struct Node** createGraph(int n)
{
    struct Node** adjList =
malloc(sizeof(struct Node*)*n);
    for(int i=0; i<n; i++) {
        adjList[i] = NULL;
    }
    return adjList;
}
```

```
#define NMAX 500

vector<int>* createGraph()
{
    return new vector<int>[NMAX];
}
```

```
struct Node
{
    int adj_vertex;
    struct Node* next;

};
```

# Operation: addEdge

- `addEdge(G,u,v)`: add the edge from vertex *u* to vertex *v* in the graph *G*

```
void addEdge(struct Node** adjList, int u,
int v)
{
    struct Node* node = adjList[u];
    if(node == NULL) {
    node = malloc(sizeof(struct Node));
    node->adj_vertex = v;
    node->next = NULL;
    adjList[u] = node;
    } else {
    while(node->next != NULL)
        node =  node->next;
    struct Node* new_node =
malloc(sizeof(struct Node));
    new_node->adj_vertex = v;
    new_node->next = NULL;
    node->next = new_node;
    }
}
```

```
void addEdge(vector<int>*
adjList, int u, int v)
{
    adjList[u].push_back(v);
}
```

# Operation: printGraph

- `printGraph`(G): print the graph *G*

```
void printGraph(struct Node** adjList, int n)
{
    for (int u = 0; u < n; u++)
    {
    printf("[%d] head: ", u);

    struct Node* node = adjList[u];

        while(node) {
        printf("-> %d ", node->adj_vertex);
        node = node->next;
        }

        printf("-> NULL \n");
    }
    printf("\n");
}
```

```
void printGraph(vector<int>* adjList,
int V)
{
    for (int v = 0; v < V; ++v)
    {
        cout << "[" << v << "] head ";
        for(int i=0; i <
            adjList[v].size(); i++)
            cout << "-> " <<
                adjList[v].at(i);
        cout << "\n";
    }
}
```

# Operation: removeEdge

- `removeEdge(G, u, v)`: remove the existing edge from vertex *u* to vertex *v*

```
void removeEdge(struct Node** adjList,
int u, int v)
{
    struct Node* node = adjList[u];
    if(node->adj_vertex == v) {
        adjList[u] = node->next;
        free(node);
    } else {
        struct Node* prev_node = node;
        node = node->next;
        while(node->adj_vertex != v) {
            prev_node = node;
            node = node->next;
        }
        prev_node->next = node->next;
        free(node);
    }
}
```

```
void removeEdge(vector<int>* adjList, int
u, int v) {
    for(int i=0; i<adjList[u].size(); i++)
        if(adjList[u].at(i) == v) {
        adjList[u].erase(
            adjList[u].begin()+i);
            return;
        }
}
```

# Operation: addVertex

- `addVertex(G, u)`: add the new vertex *u* to the graph *G*

```
struct Node** addVertex(struct Node**
adjList, int *n, int u) {
    struct Node** new_adjList =
createGraph(u+1);
    for (int i=0; i<*n; i++) {
        new_adjList[i] = adjList[i];
        adjList[i] = NULL;
    }
    deleteGraph(adjList, *n);
    *n = u+1;
    return new_adjList;
}
```

```
void addVertex(int &n, int u)
{
    n = u+1;
}
```

```
void deleteGraph(struct Node** adjList, int n) {
    for (int u=0; u<n; u++) {
        struct Node* node = adjList[u];
        while(node != NULL) {
            struct Node* next_node = node->next;
            free(node);
            node = next_node;
        }
    }
    free(adjList);
}
```

# Operation: removeVertex

- `removeVertex(G, u)`: remove the existing vertex *u* from the graph *G*

```
void removeVertex(struct Node** adjList, int *n,
int u) {

    for(int v=0; v<*n; v++) {
        if(isAdjacent(adjList, v, u) == 1)
            removeEdge(adjList, u, v);
    }
    struct Node* node = adjList[u];
    while(node != NULL) {
        struct Node* next_node = node->next;
        free(node);
        node = next_node;
    }
    adjList[u] = NULL;
    if(u < *n-1)
        return;
    (*n)--;
}
```

```
void removeVertex(vector<int>*
adjList,int &n, int u)
{
    if(u < n-1)
        return;
    for(int v=0; v<n; v++)
        removeEdge(adjList, v, u);
    adjList[u].clear();
    n--;
}
```

# Operation: isAdjacent

- `isAdjacent(G, u, v)`: check whether vertices *u* and *v* are adjacent in *G*

```
int isAdjacent(struct Node**
adjList, int u, int v) {
    struct Node* node = adjList[u];
    int ret = 0;
    while(node != NULL) {
        if(node->adj_vertex == v)
            ret = 1;
        node = node->next;
    }
    return ret;
}
```

```
bool isAdjacent(vector<int>* adjList, int
u, int v)
{
    for(int i=0; i<adjList[u].size(); i++)
        if(adjList[u].at(i) == v)
            return true;
    return false;
}
```

# Operation: inDegree

- `inDegree(G, u)`: return the *in-degree* of vertex *u* in G

```
int inDegree(struct Node** adjList, int n,
int u) {
    int in_deg = 0;

    for(int i=0; i<n; i++) {
        struct Node* node = adjList[i];
        while(node) {
            if(node->adj_vertex == u)
                in_deg++;
            node = node->next;
        }
    }
    return in_deg;
}
```

```
int inDegree(vector<int>* adjList, const
int n,  int u) {
    int count = 0;
    for(int v=0; v<n; v++)
        for(int i=0; i<adjList[v].size();
i++)
            if(adjList[v].at(i) == u)
                count++;
    return count;
}
```

# Operation: outDegree

- `outDegree(G, u)`: return the *out-degree* of vertex *u* in G

```
int outDegree(struct Node**
adjList, int n, int u) {
    struct Node* node = adjList[u];
    int out_deg = 0;
    while(node != NULL) {
        out_deg++;
        node = node->next;
    }
    return out_deg;
}
```

```
int outDegree(vector<int>* adjList, int u)
{
    return adjList[u].size();
}
```