

Data Structures

Lecture 11: Graphs (cont.)

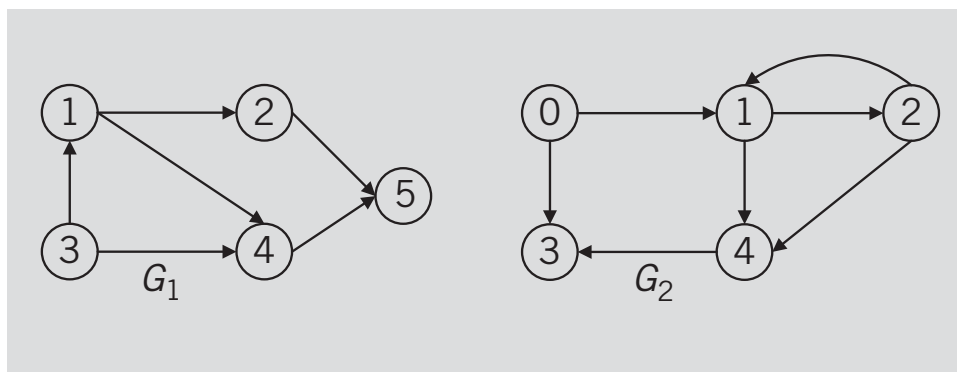
Nopadon Juneam
Department of Computer Science
Kasetsart university
(10/9/63)

Outlines

- Graph representations: adjacency matrix & adjacency list (review)
- Basic operations on graphs (detailed)
- More operations on graphs
- Graph representations: pros & cons

Adjacency Matrix

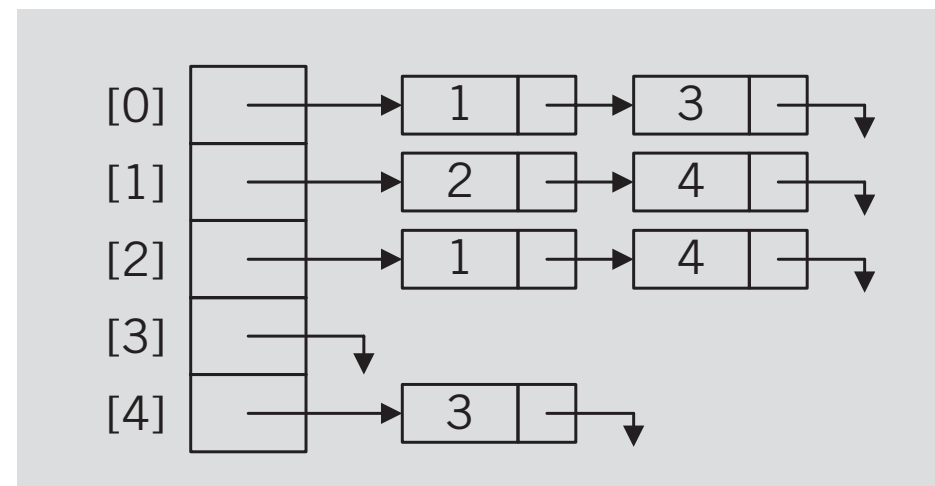
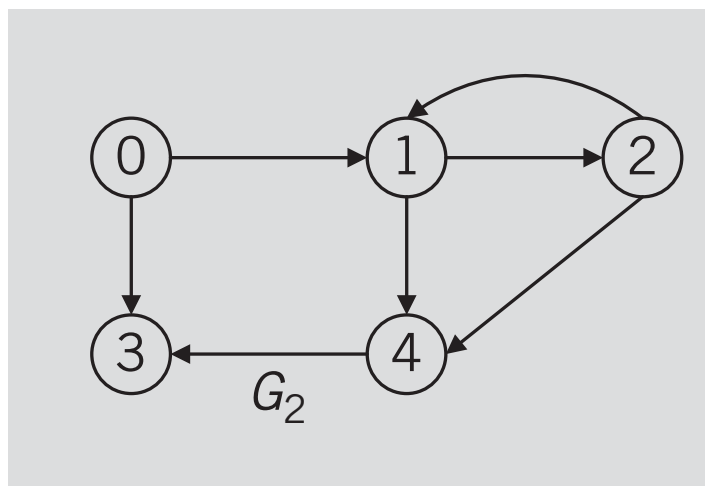
- The ***adjacency-matrix representation*** is a 2D array of size $n \times n$, where n is the number of vertices in a graph.
- The (i, j) -th entry of the array is 1 if there is an edge from vertex i to vertex j ; otherwise, the (i, j) -th entry is 0.



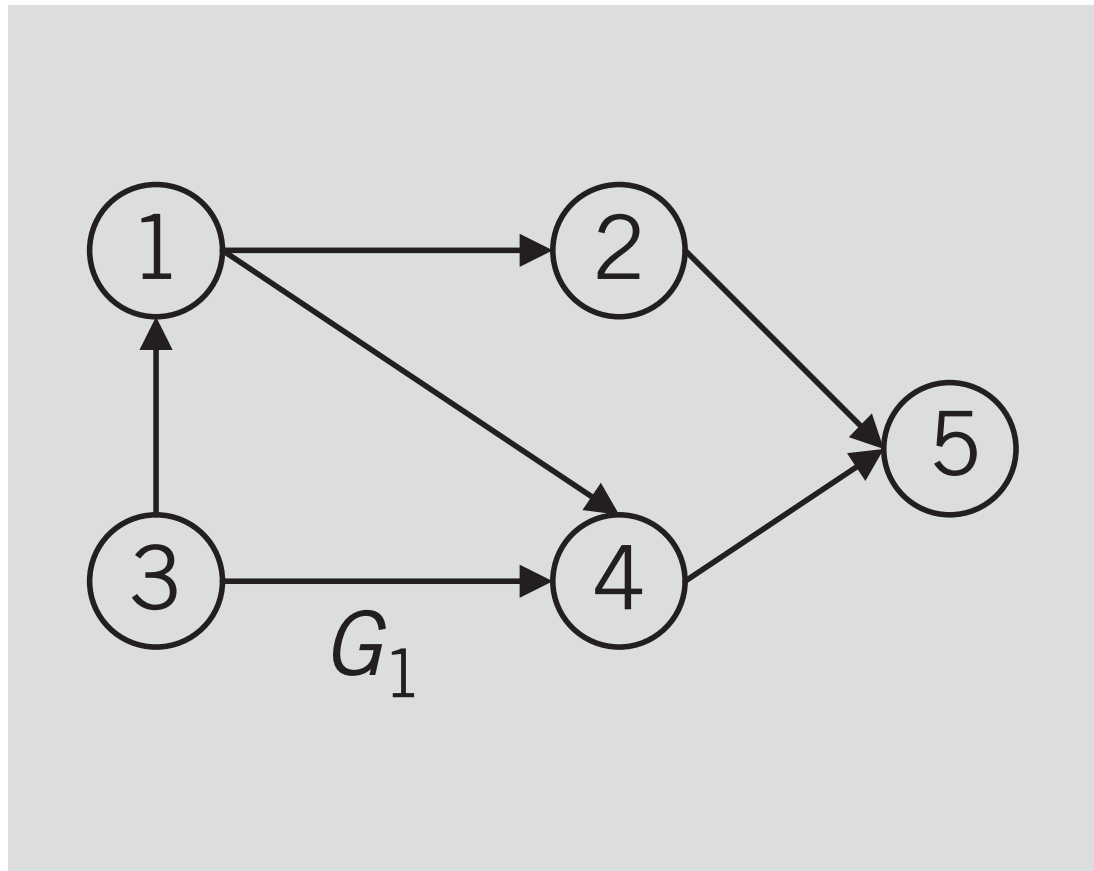
$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency List

- The ***adjacency-list representation*** of a graph G consists of an array Adj of n lists, where n is the number of vertices in a graph; one list for each vertex in V .
- For each u in V , the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge (u, v) in E . In other words, $Adj[u]$ consists of all the vertices adjacent to u .



Programming Exercise



- Let's try to create the above graph.

Basic Operations on Graphs

- Basic operations commonly performed on a graph:
 - `createGraph(n)`: create the empty graph with n isolated vertices;
 - `addEdge(G, u, v)`: add the edge from vertex u to vertex v in the graph G ;
 - `printGraph(G)`: print the graph G ;
 - `deleteGraph(G)`: delete the graph G ;

Operation: createGraph

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

```
// Create the graph using adjacency-matrix representation
int** createGraph(const int n) {
    // Return 2D array of size n*n
    int** adjMatrix = malloc(sizeof(int*)*n);
    for (int i=0; i<n; i++) {
        adjMatrix[i] = malloc(sizeof(int)*n);
        for (int j=0; j<n; j++)
            adjMatrix[i][j] = 0;
    }
    return adjMatrix;
}
```

```
// Create the graph using adjacency-list representation
struct Node** createGraph(int n){
    // Return array of n lists (vectors)
    struct Node** adjList = malloc(sizeof(struct Node*)*n);
    for(int i=0; i<n; i++)
        adjList[i] = NULL;
    return adjList;
}
```

```
// Implementation of singly linked list
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 ;

```
// Add the edge to the graph using
// adjacency-matrix representation
void addEdge(int** adjMatrix, int u,
int v) {
    adjMatrix[u][v] = 1;
}
```

```
// Add the edge to the graph using adjacency-
// list representation
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;
    }
}
```


Operation: printGraph

- printGraph(G): print the graph G;

```
// Print the adjacency-matrix representation of the graph
void printGraph(int** adjMatrix, int n)
{
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++)
            printf("%d ", adjMatrix[i][j]);
        printf("\n");
    }
}
```

```
// Print the adjacency list representation of the graph
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");
    }
}
```

Operation: deleteGraph

- deleteGraph(G): delete the graph G;

```
// Delete the adjacency-matrix representation of the graph
void deleteGraph(int** adjMatrix, int n){
    for (int i=0; i<n; i++) {
        free(adjMatrix[i]);
    }
    free(adjMatrix);
}
```

```
// Delete the adjacency list representation of the graph
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);
}
```

More Operations on Graphs

- `removeEdge(G, u, v)`: remove the existing edge from vertex u to vertex v .
- `addVertex(G, u)`: add the new vertex u to the graph G ;
- `removeVertex(G, u)`: remove the existing vertex u from the graph G ;
- `isAdjacent(G, u, v)`: check whether vertices u and v are adjacent in G ;
- `inDegree(G, u)`: return the *in-degree* of vertex u in G ;
- `outDegree(G, u)`: return the *out-degree* of vertex u in G ;

Operation: removeEdge

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

```
// Remove the existing edge from
// the graph using adjacency-matrix
// representation
void removeEdge(int** adjMatrix,
int u, int v) {
    adjMatrix[u][v] = 0;
}
```

```
// Remove the existing edge from the graph using
// adjacency-list representation
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);
    }
}
```

Operation: addVertex

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

```
// Add the new vertex to the graph
// using adjacency-matrix
// representation
int** addVertex(int** adjMatrix, int *n,
int u) {
    int** adjMatrix_new =
createGraph(u+1);
    for (int i=0; i<*n; i++) {
        for (int j=0; j<*n; j++)
            if(adjMatrix[i][j] == 1)
                adjMatrix_new[i][j] = 1;
    }
    deleteGraph(adjMatrix, *n);
    *n = u+1;
    return adjMatrix_new;
}
```

```
// Add the new vertex to the graph
// graph using adjacency-list
// representation
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;
}
```

Operation: removeVertex

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

```
// Remove the existing vertex from the graph
// using adjacency-matrix representation
int** removeVertex(int** adjMatrix, int* n, int u)
{
    for (int i=0; i<*n; i++)
        for (int j=u; j<*n-1; j++)
            adjMatrix[i][j] = adjMatrix[i][j+1];

    for (int i=0; i<*n; i++)
        for (int j=u; j<*n-1; j++)
            adjMatrix[j][i] = adjMatrix[j+1][i];

    int** adjMatrix_new = createGraph(*n-1);
    for (int i=0; i<*n-1; i++)
        for (int j=0; j<*n-1; j++)
            adjMatrix_new[i][j] = adjMatrix[i][j];
    deleteGraph(adjMatrix, *n);
    (*n)--;
    return adjMatrix_new;
}
```

```
// Remove the existing vertex
// from the graph using
// adjacency-list representation
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)--;
}
```

Operation: isAdjacent

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

```
// Check whether the two vertices
// are adjacent using adjacency-
// matrix representation
int isAdjacent(int** adjMatrix, int u,
int v) {
    if(adjMatrix[u][v] == 1)
        return 1;
    else
        return 0;
}
```

```
// Check whether the two vertices
// are adjacent using adjacency-
// list representation
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;
}
```

Operation: inDegree

- $\text{inDegree}(G, u)$: return the *in-degree* of vertex u in G ;

```
// Report the in-degree of the
// vertex using adjacency-
// matrix representation
int inDegree(int** adjMatrix, const
int n, int u) {
    int in_deg = 0;
    for(int i=0; i<n; i++) {
        if(adjMatrix[i][u])
            in_deg++;
    }
    return in_deg;
}
```

```
// Report the in-degree of the
// vertex using adjacency-
// list representation
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;
}
```


Operation: outDegree

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

```
// Report the out-degree of the
// vertex using adjacency-
// matrix representation
int outDegree(int** adjMatrix, const
int n, int u) {
    int out_deg = 0;
    for(int i=0; i<n; i++) {
        if(adjMatrix[u][i])
            out_deg++;
    }
    return out_deg;
}
```

```
// Report the out-degree of the
// vertex using adjacency-
// list representation
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;
}
```

Graph Representations: Complexity of Operations

Operations	Adjacency matrix	Adjacency list
createGraph (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	$O(V)$	$O(V)$
<i>Remarks</i>	Slow to add/remove vertices as matrix must be resized/copied	Slow to remove edges because it needs to iterate all the adjacent vertices