Gebze Technical University

CSE-222

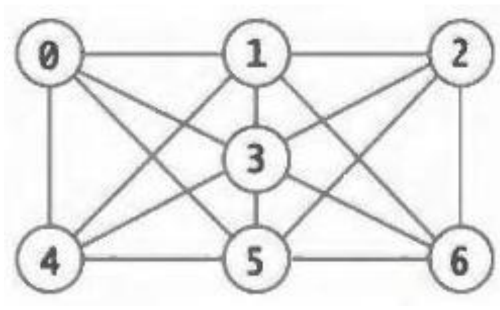
#HW8 - Question1

Visualization and Traversing of Graph Data Structure

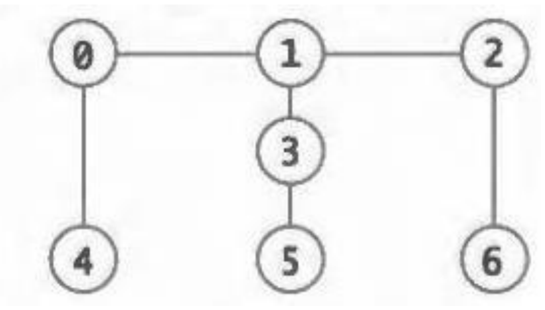
Nevzat Seferoglu

171044024

Problem Definition :



A



B

- Represent the graphs above using adjacency lists. Draw the corresponding data structure.
- Represent the graphs above using an adjacency matrix. Draw the corresponding data structure.
- For each graph above, what are the IVI=n, the IEI=m, and the density? Which representation is better for each graph? Explain your answers.
- Draw DFS tree starting from vertex 2 and traversing the vertices adjacent to a vertex in descending order (largest to smallest).
- Draw BFS tree starting from vertex 2 and traversing the vertices adjacent to a vertex in descending order (largest to smallest).

Show your work step by step and make your explanation. Write your solution by latex, word or handwriting (scan or take picture) and upload it as a single *StudentNumber.pdf* file.

Adjacent List :

In this implementation , each vertex of the graph is used as an index of list array which each node represents an edge of vertex which is already defined as index of array of lists.

*Visualization of Graph 'A' with Adjacent List Implementation:

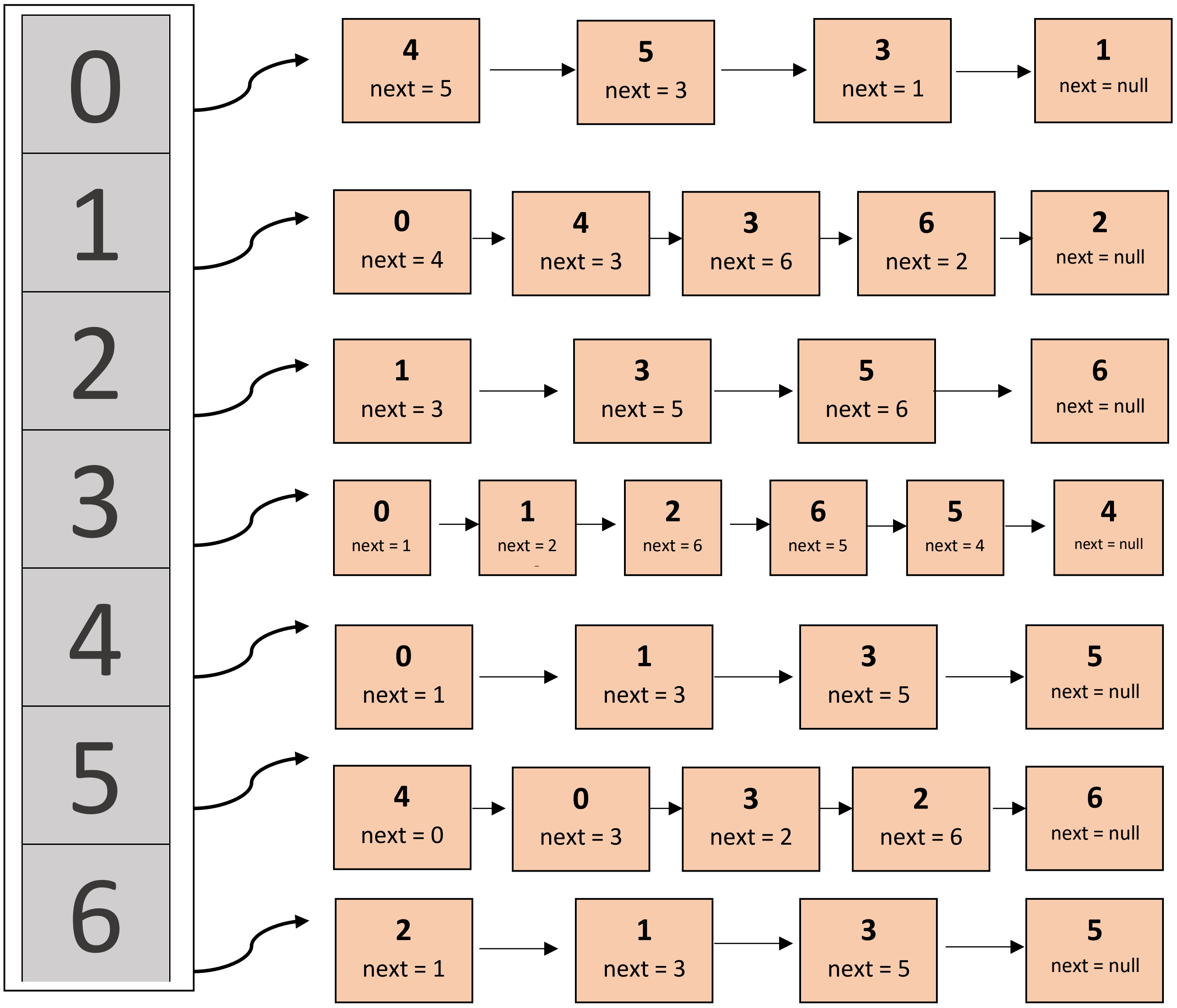
As first steps ,

*On the other hand, we also need to allocate array with total size of vertex on the graph.

*As last , we need to indicate the type of graph.

*Total Size : **7**

*Type of Graph : **Undirected Graph**



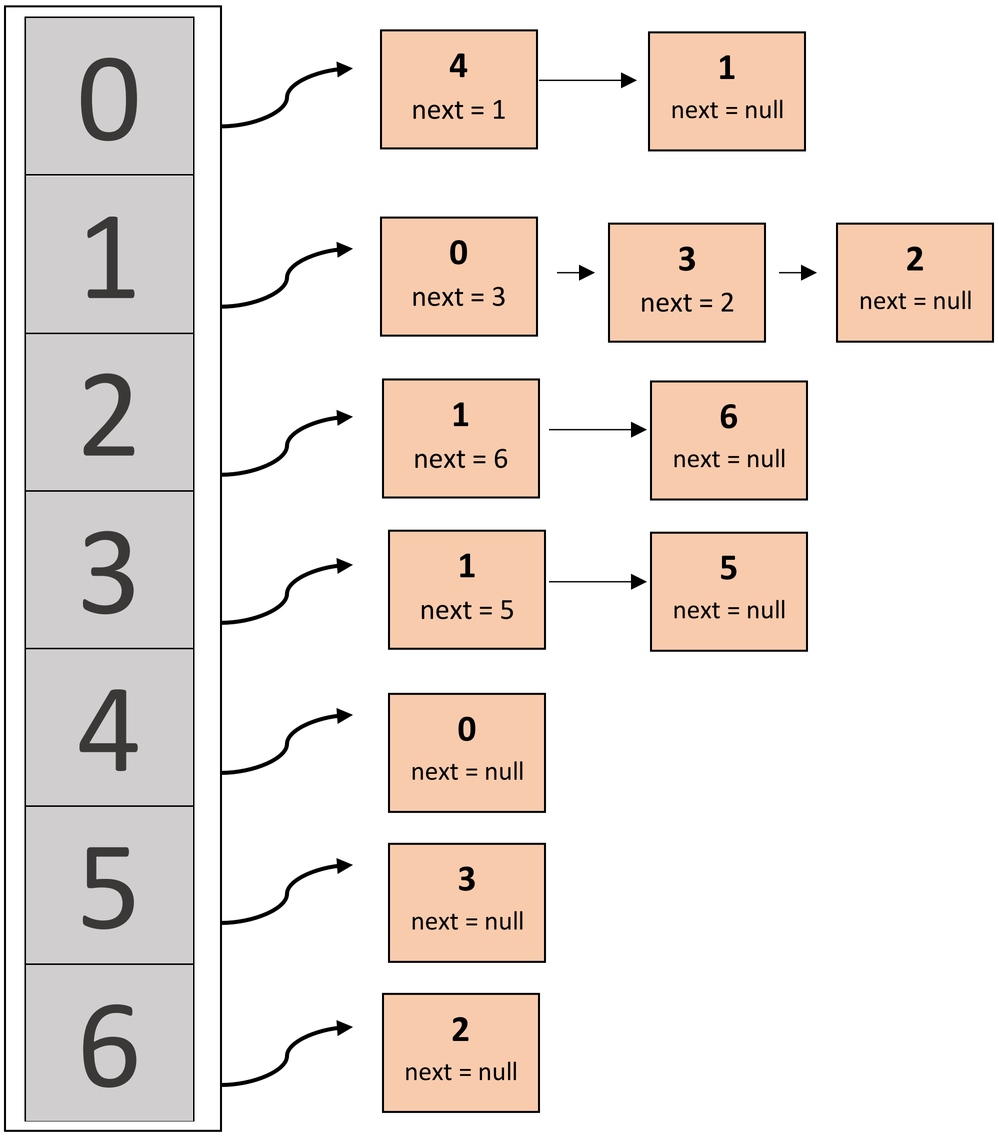
***Visualization of Graph ‘B’ with Adjacent List Implementation:**

As first steps ,

* We need to allocate array with total size of vertex on the graph.
* We also need to indicate the type of graph.

*Total Size : **7**

*Type of Graph : **Undirected Graph**



Adjacent Matrix :

In this implementation , each vertex of the graph is used as an index of two-dimensional array which each **(i , j)** represents weight of the weighted graph. If the graph is not weighted graph , we can put some distinct value on this cell. The book that we use for this course suggests putting **infinite value of double primitive type**. Thus , we can represent not connected vertexes with **INF** and connected vertex with **another distinct number**.

*Visualization of Graph 'A' with Adjacent Matrix Implementation:

As first steps ,

* We need to allocate a two-dimensional array with total size of vertex on the graph for both dimension.

* We need to indicate the type of graph.

* We need to designate a distinct value to represent connection.

*Total Size : **7**
*Type of Graph : **Undirected Graph**
*Connected Vertex Value = **1.0**
*Unconnected Vertex Value = **INF**

CELL	0	1	2	3	4	5	6
0	INF	1.0	INF	1.0	1.0	1.0	INF
1	1.0	INF	1.0	1.0	1.0	INF	1.0
2	INF	1.0	INF	1.0	INF	1.0	1.0
3	1.0	1.0	1.0	INF	1.0	1.0	1.0
4	1.0	1.0	INF	1.0	INF	1.0	INF
5	1.0	INF	1.0	1.0	1.0	INF	1.0
6	INF	1.0	1.0	1.0	INF	1.0	INF

*Visualization of Graph ‘A’ with Adjacent Matrix Implementation:

As first steps ,

* We need to allocate a two-dimensional array with total size of vertex on the graph for both dimension.
* We need to indicate the type of graph.
* We need to designate a distinct value to represent connection.

*Total Size : **7**
*Type of Graph : **Undirected Graph**
*Connected Vertex Value = **1.0**
*Unconnected Vertex Value = **INF**

CELL	0	1	2	3	4	5	6
0	INF	1.0	INF	INF	1.0	INF	INF
1	1.0	INF	1.0	1.0	INF	INF	INF
2	INF	1.0	INF	INF	INF	INF	1.0
3	INF	1.0	INF	INF	INF	1.0	INF
4	1.0	INF	INF	INF	INF	INF	INF
5	INF	INF	INF	1.0	INF	INF	INF
6	INF	INF	1.0	INF	INF	INF	INF

*Analyzing and Selecting appropriate implementation for graph ‘A’ and ‘B’:

Both implementation has unique advantageous for different kind of graphs. There is a parameter to determine which one is efficient in terms of memory allocation and which one is efficient for some operation on the graph. This parameter is **density** of the graph.

According to calculated density we can determine which implementation is more efficient than the other. We can divide the graph into two parts according to their density.

Dense Graph :

This type of graph has a total number of edge (**|E|**) which so close to the square of total number vertex (**$|V|^2$**) but more less. Therefore, we can assume that $|E| = O\,(|V|^2)$.

Sparse Graph :

This type of graph has a total number of edge (**|E|**) which is much less than the square of total number vertex (**$|V|^2$**). Therefore, we can assume that $|E| = O\,(|V|)$.

As a result , if we use adjacency matrix for sparse graph , it will not be efficient because there will be a lot **INF** which represents non-connectivity but if we use adjacency list , there would not be **INF** or that kind of stuff. We can conclude that **adjacency matrix is more efficient for dense graph** , on the other hand **adjacency list is more efficient for sparse graph** because there will not be unnecessary memory allocation.

***Density :**

We can calculate density with this formula : $\boldsymbol{D} = \frac{|E|}{|V|^2}$

Analyzing of graph ‘A’ :

$|V| = 6$
$|E| = 16$
$D = \frac{16}{36} = 0.4$

Analyzing of graph ‘B :

$|V| = 6$
$|E| = 6$
$D = \frac{6}{36} = 0.16$

According to calculated density , we cannot truly say which implementation is better. Because density is much less than the square of total number of vertex for both graph. Therefore , we can analyze both implementation in a sense of **storage efficiency**.

Storage Efficiency :

Storage is allocated for **all** vertex combinations in **adjacency matrix.** We can represent this allocation with **$|V|^2$**. If the graph is sparse (not many edges), there will be a lot of wasted space in the adjacency matrix. In an adjacency list, only the adjacent edges are stored. On the other hand, in an adjacency list, each edge is represented by a reference to an **Edge** object containing data about the **source**, **destination**, and **weight**. There is also a reference to the **next edge** in the list. In a matrix representation, only the weight associated with an edge is stored. Therefore, each element in an adjacency list requires approximately **four times** the storage of an element in an adjacency matrix.

*Based on this, we can conclude that the break-even point in terms of storage efficiency occurs when approximately 25 percent of the adjacency matrix is filled with meaningful data. That is, the adjacency list uses less (more) storage when less than (more than) 25 percent of the adjacency matrix would be filled.

Percentage of fullness for graph in adjacency matrix 'A' = $\frac{2*|E|}{|V|^2} = \frac{(2*16)}{36} = 0.8$ which is much more than 0.25. **Therefore, for the graph 'A' adjacency matrix will be much more efficient than the adjacency list.**

Percentage of fullness for graph in adjacency matrix 'B' = $\frac{2*|E|}{|V|^2} = \frac{(2*6)}{36} = 0.3$ which is much than 0.25. **Therefore, for the graph 'B' adjacency matrix will be much more efficient than the adjacency list.**

Draw DFS tree starting from vertex 2 and traversing the vertices adjacent to a vertex in descending order (largest to smallest).

Graph ‘A’ :

Discovery = { 2 }

Finish Order = { }

$$\begin{tikzcd} 2 \end{tikzcd}$$

Discovery = { 2 , 6 }

Finish Order = { }

$$\begin{tikzcd} 2 \arrow[r] & 6 \end{tikzcd}$$

Discovery = { 2 , 6 , 5 }

Finish Order = { }

$$\begin{tikzcd} 2 \arrow[r] & 6 \arrow[r] & 5 \end{tikzcd}$$

Discovery = { 2 , 6 , 5 , 4 }

Finish Order = { }

$$\begin{tikzcd} 2 \arrow[r] & 6 \arrow[r] & 5 \arrow[r] & 4 \end{tikzcd}$$

Discovery = { 2 , 6 , 5 , 4 ,3 }

Finish Order = { }

$$\begin{tikzcd} 2 \arrow[r] & 6 \arrow[r] & 5 \arrow[r] & 4 \arrow[r] & 3 \end{tikzcd}$$

Discovery = { 2 , 6 , 5 , 4 ,3 , 1 }

Finish Order = { }

$$\begin{tikzcd} 2 \arrow[r] & 6 \arrow[r] & 5 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 1 \end{tikzcd}$$

Discovery = { 2 , 6 , 5 , 4 ,3 , 1 , 0 }

Finish Order = { }

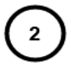
$$\begin{tikzcd} 2 \arrow[r] & 6 \arrow[r] & 5 \arrow[r] & 4 \arrow[r] & 3 \arrow[r] & 1 \arrow[r] & 0 \end{tikzcd}$$

Discovery = { 2 , 6 , 5 , 4 ,3 , 1 , 0 }

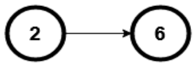
Finish Order = { 0 , 1, 3 , 4 , 5 , 6 }

Graph ‘B’ :

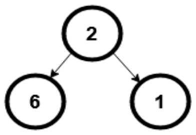
Discovery = { 2 }
Finish Order = { }



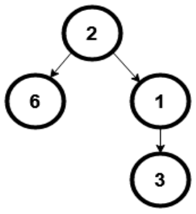
Discovery = { 2 , 6 }
Finish Order = {}



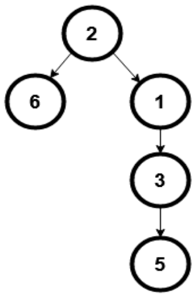
Discovery = { 2 ,6 , 1 }
Finish Order = { 6 }



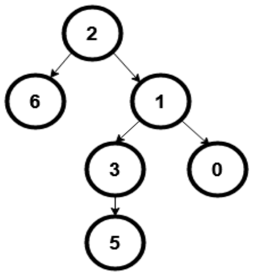
Discovery = { 2 , 6 , 1 , 3 }
Finish Order = { 6 }



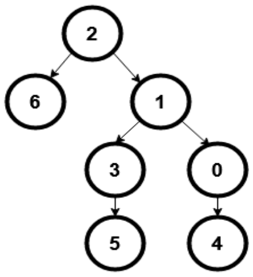
Discovery = { 2 ,6 , 1 , 3 , 5 }
Finish Order = { 6 , 5 , 3 }



Discovery = { 2 ,6 , 1 , 3 , 5 , 0 }
Finish Order = { 6 , 5 , 3 }



Discovery = { 2 ,6 , 1 , 3 , 5 , 0 , 4 }
Finish Order = { 6 , 5 , 3 , 4 ,0 }

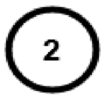


Discovery = { 2 ,6 , 1 , 3 , 5 , 0 , 4 }
Finish Order = { 6 , 5 , 3 ,4 ,0 , 1 , 2 }

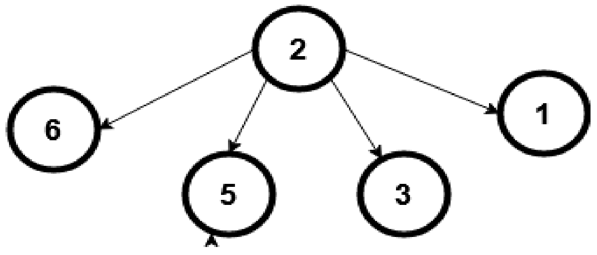
Draw BFS tree starting from vertex 2 and traversing the vertices adjacent to a vertex in descending order (largest to smallest).

Graph ‘A’ :

Qeueu = { 2 }
Visited = { }

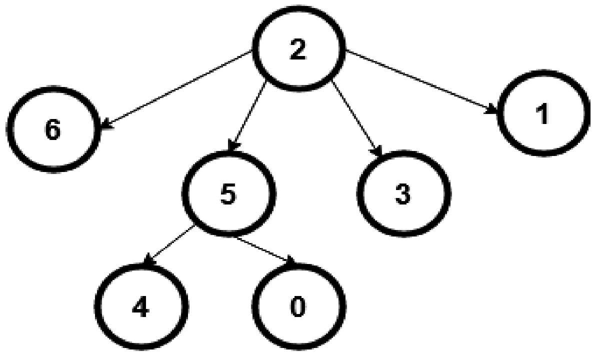


Qeueu = { 6,5,3,1 }
Visited = { 2 }



Qeueu = { 5,3,1 }
Visited = { 2,6 }

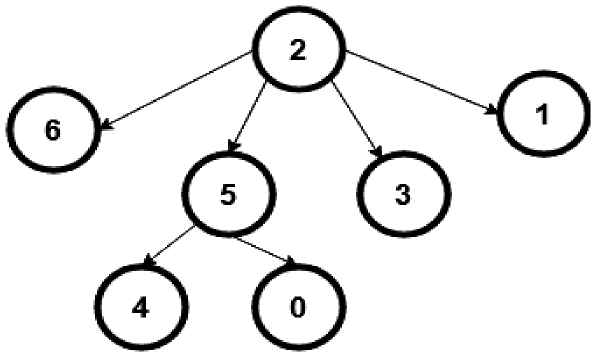
Qeueu = { 3 ,1 ,0,4 }
Visited = {2 ,6 ,5 }



Qeueu = { 1 ,0,4 }
Visited = {2 ,6 ,5 ,3 }

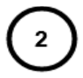
Qeueu = { 0,4 }
Visited = {2 ,6 ,5 ,3 ,1 }

Qeueu = { 4 }
Visited = {2 ,6 ,5 ,3 ,1 , 0 }

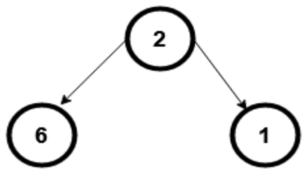


Graph ‘B’:

Qeueu = { 2 }
Visited = { }

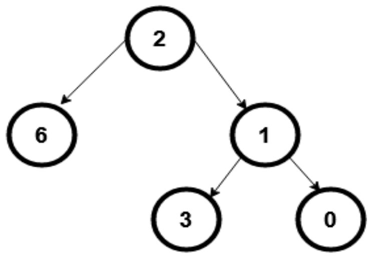


Qeueu = { 6,1}
Visited = { 2 }

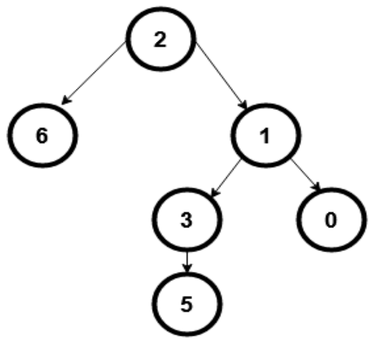


Qeueu = { 1 }
Visited = { 2,6 }

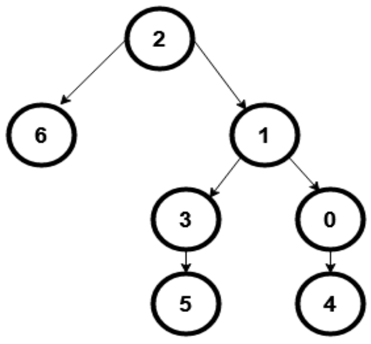
Qeueu = { 3 ,0 }
Visited = {2 ,6 ,1 }



Qeueu = { 0,5 }
Visited = {2 ,6 ,1 , 3 }



Qeueu = { 5,4 }
Visited = {2 ,6 ,1 ,3 ,0 }



Qeueu = { 4 }
Visited = {2 ,6 ,1,3,0,5 }