GIT Department of Computer Engineering

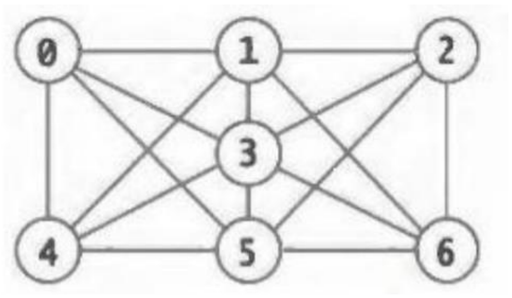
CSE 222/505 – Spring 2020

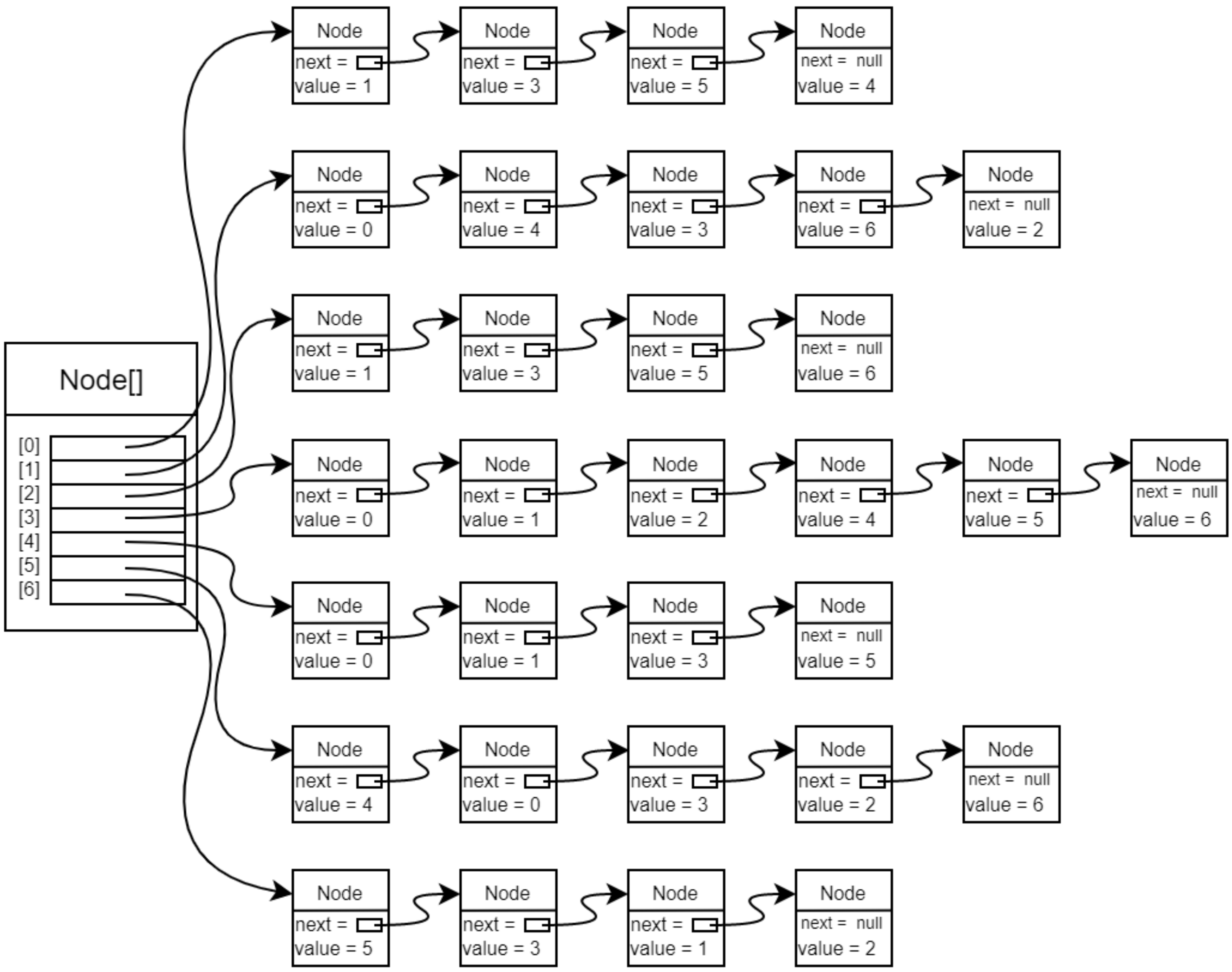
Homework #08 Part 1 Report

Abdullah ÇELİK

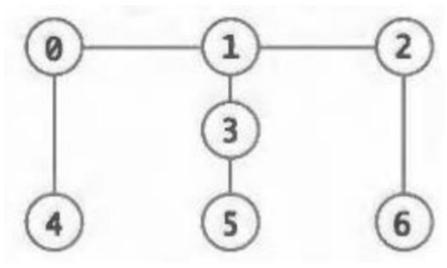
171044002

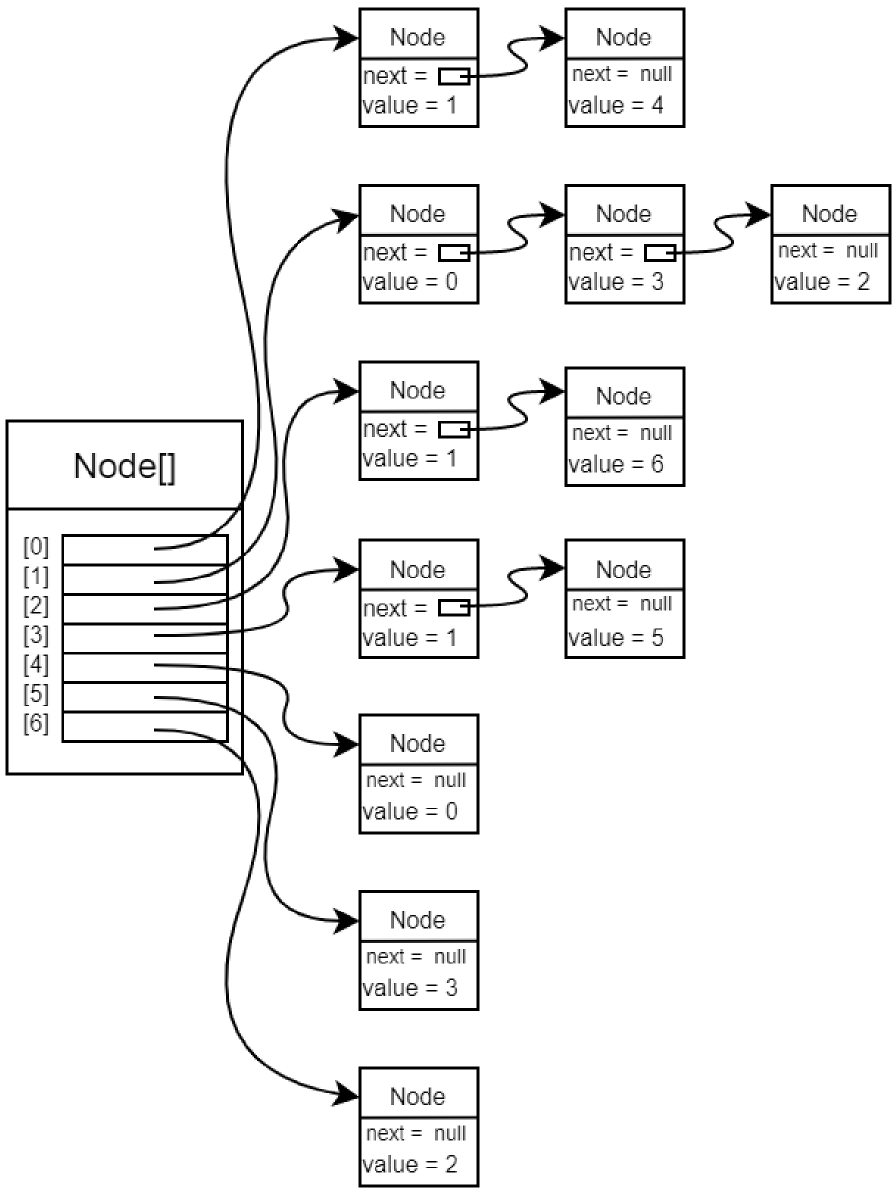
Representing the following graph using adjacency lists



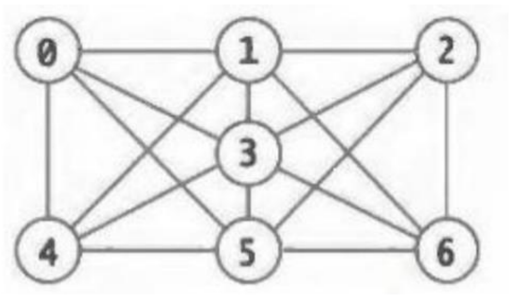


Representing the following graph using adjacency lists





Representing the following graph using adjacency matrix

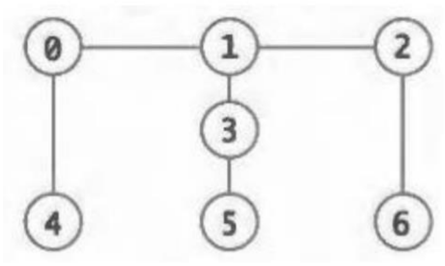


Column

Row

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]		1.0		1.0	1.0	1.0	
[1]	1.0		1.0	1.0	1.0		1.0
[2]		1.0		1.0		1.0	1.0
[3]	1.0	1.0	1.0		1.0	1.0	1.0
[4]	1.0	1.0		1.0		1.0	
[5]	1.0		1.0	1.0	1.0		1.0
[6]		1.0	1.0	1.0		1.0	

Representing the following graph using adjacency matrix



Column

Row

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
[0]		1.0			1.0		
[1]	1.0		1.0	1.0			
[2]		1.0					1.0
[3]		1.0				1.0	
[4]	1.0						
[5]				1.0			
[6]			1.0				

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.

$|V|$ = n = 6 and $|E|$ = m = 32 in first graph. The density of the graph is the ratio of $|E|$ to $|V|^2$. The density is 0,89. The density is quite high.

Some graph algorithms are of the form:

```
1-  for each vertex u in some subset of the vertices
2-      for each vertex u in some subset of the vertices
3-          if(u,v) is an edge
4-              Do something with edge (u,v).
```

For an adjancecy matrix representation, Step 3 tests a matrix value and is $\Theta(1)$, so the overall algorithm is $\Theta(|V|^2)$. However, for an adjacency list representation, Step 3 searches a list and is $\Theta(|E_u|)$, so the combination of Steps 2 and 3 is $\Theta(|E|)$ and the overall algorithm is $\Theta(|V||E|)$. For a dense graph, the adjacency matrix gives the best performance for this type of algorithm, and for a sparse graph, the performance is the same for both representation. And in graphs with high density, adjacency matrix takes less mermory than adjacency list(next,value). Therefore, it is also advantageous as memory.

$|V|$ = n = 6 and $|E|$ = m = 12 in second graph. The density of the graph is the ratio of $|E|$ to $|V|^2$. The density is 0,34. So graph is a sparse graph.

Many graph algorithms are of the form:

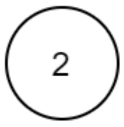
```
1-  **for** each vertex u in the graph
2-      **for** each vertex v adjacent to u
3-              Do something with edge (u,v).
```

For an adjacency list representation, Step 1 is $\Theta(|V|)$ and Step 2 is $\Theta(|E_u|)$. Thus , the combination of Steps 1 and 2 will represent examining each edge in the graph, giving $\Theta(|E|)$. For an adjacecncy matrix representation, Step 2 is also $\Theta(|V|)$, and thus the overall algorithm is $\Theta(|V|^2)$. Thus, for a sparse graph, the adjacency list gives better performance fort his type of algorithm, whereas for a dense graph, the performance is the sam efor either representation.

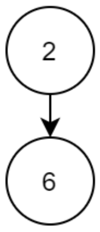
Draw DFS tree starting from vertex 2 and traversing the vertices adjacent to a vertex in descending order (largest to smallest).

For first graph,

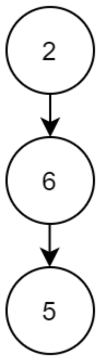
Start vertex 2 add it to tree



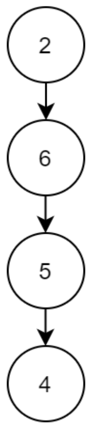
Recursion call to 2's largest and unvisited adjacent and add it to the tree.



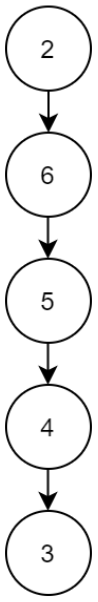
Recursion call to 6's largest and unvisited adjacent and add it to the tree.



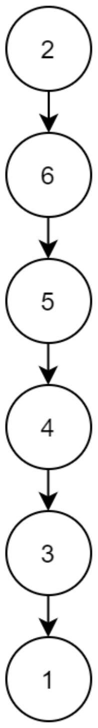
Recursion call to 5's largest and unvisited adjacent and add it to the tree.



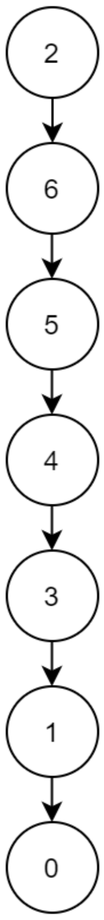
Recursion call to 4's largest and unvisited adjacent and add it to the tree.



Recursion call to 3's largest and unvisited adjacent and add it to the tree.



Recursion call to 1's largest and unvisited adjacent and add it to the tree.



Recursion call to 0's largest and unvisited adjacent and add it to the tree. All adjacent of 0 were visited. Return to the previous call.

Recursion call to 1's largest and unvisited adjacent and add it to the tree. All adjacent of 1 were visited. Return to the previous call.

Recursion call to 3's largest and unvisited adjacent and add it to the tree. All adjacent of 3 were visited. Return to the previous call.

Recursion call to 4's largest and unvisited adjacent and add it to the tree. All adjacent of 4 were visited. Return to the previous call.

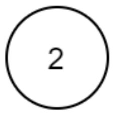
Recursion call to 5's largest and unvisited adjacent and add it to the tree. All adjacent of 5 were visited. Return to the previous call.

Recursion call to 6's largest and unvisited adjacent and add it to the tree. All adjacent of 6 were visited. Return to the previous call.

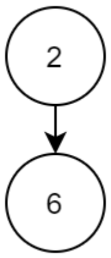
Recursion call to 2's largest and unvisited adjacent and add it to the tree. All adjacent of 2 were visited. The recursion is finished.

For second graph,

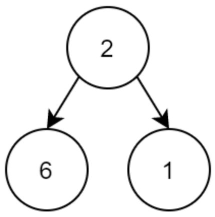
Start vertex 2 add it to tree



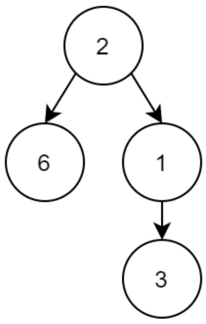
Recursion call to 2's largest and unvisited adjacent and add it to the tree.



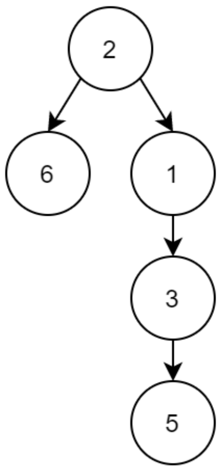
Recursion call to 6's largest and unvisited adjacent and add it to the tree. All adjacent of 6 were visited. Return to the previous call. Recursion call to 2's largest and unvisited adjacent and add it to the tree.



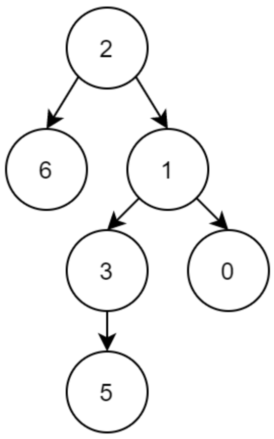
Recursion call to 1's largest and unvisited adjacent and add it to the tree.



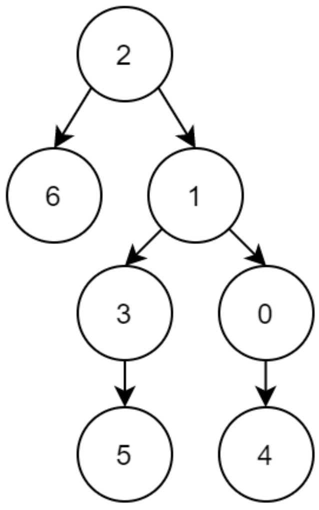
Recursion call to 3's largest and unvisited adjacent and add it to the tree.



Recursion call to 5's largest and unvisited adjacent and add it to the tree. All adjacent of 5 were visited. Return to the previous call. Recursion call to 3's largest and unvisited adjacent and add it to the tree. All adjacent of 3 were visited. Return to the previous call. Recursion call to 1's largest and unvisited adjacent and add it to the tree.



Recursion call to 0's largest and unvisited adjacent and add it to the tree.



Recursion call to 4's largest and unvisited adjacent and add it to the tree. All adjacent of 4 were visited. Return to the previous call.

Recursion call to 0's largest and unvisited adjacent and add it to the tree. All adjacent of 0 were visited. Return to the previous call.

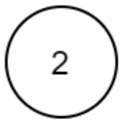
Recursion call to 1's largest and unvisited adjacent and add it to the tree. All adjacent of 1 were visited. Return to the previous call.

Recursion call to 2's largest and unvisited adjacent and add it to the tree. All adjacent of 2 were visited. Recursion is finished.

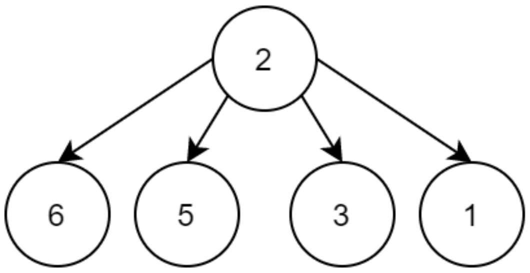
Draw BFS tree starting from vertex 2 and traversing the vertices adjacent to a vertex in descending order (largest to smallest).

For first graph,

Start vertex 2 add it to tree

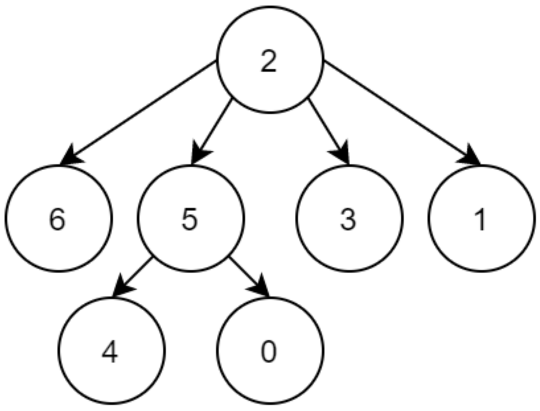


Add 2’s adjacent as descending order to tree.



Add 6’s unvisited adjacent as descending order to tree. All were visited.

Add 5’s adjacent as descending order to tree.



Add 3’s unvisited adjacent as descending order to tree. All were visited.

Add 1’s unvisited adjacent as descending order to tree. All were visited.

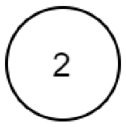
Add 4’s unvisited adjacent as descending order to tree. All were visited.

Add 0’s unvisited adjacent as descending order to tree. All were visited.

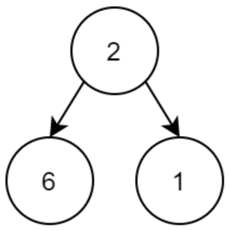
Breadth First is over.

For second graph,

Start vertex 2 add it to tree

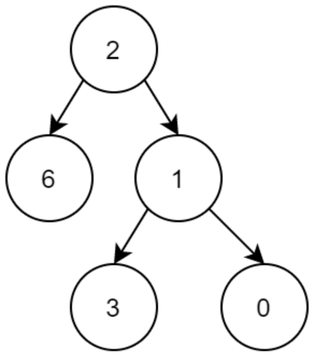


Add 2’s adjacent as descending order to tree.

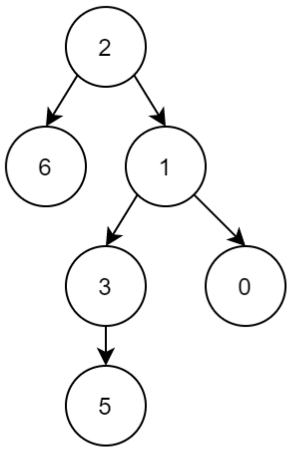


Add 6’s unvisited adjacent as descending order to tree. All were visited.

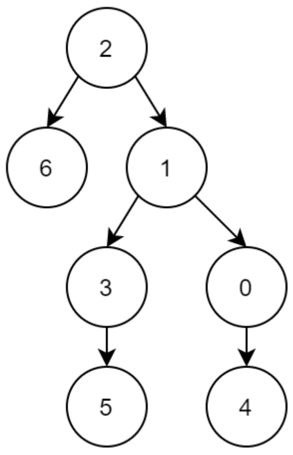
Add 1’s unvisited adjacent as descending order to tree.



Add 3’s unvisited adjacent as descending order to tree.



Add 0’s unvisited adjacent as descending order to tree.


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

Add 5’s unvisited adjacent as descending order to tree. All were visited.

Add 4’s unvisited adjacent as descending order to tree. All were visited.

Breadth First is over.