CSE 3318

Week of 07/08/2024

Instructor : Donna French

OLQ

Coding Assignment 4 will be using Quick Sort.

I HIGHLY recommend that you go ahead and start on the Coding Assignment and get the Quick Sort code work on a small hardcoded array. Add the print statements as shown here in the slides.

Practice by giving yourself an array and figuring out the prints. Then, run your program and check your answer.

Coding Assignment 4

Make a copy of Coding Assignment 3. Remove the Merge Sort and Insertion Sort code.

Add a Quick Sort function to your code.

You will run your code on the same files from Coding Assignment 3.

There will be other things to do but get this working to practice for the OLQ.

Coding Assignment 4

You will be making a copy of Coding Assignment 3.

Replace the MergeSort and Insertion Sort with Quick Sort.

The Quick Sort we did in class used the rightmost element as the pivot.

You will be using conditional compile to make your program run 3 different ways.

Coding Assignment 4

QSR

Run Quick Sort with rightmost element as pivot.

QSM

Run Quick Sort with middle element as pivot

QSRND

Run Quick Sort with a random pivot element

Coding Assignment 4

```
#ifdef QSM
int middle = (high+low)/2;
swap(&A[middle], &A[high]);
#elif QSRND
int random = (rand() % (high-low+1)) + low;
swap(&A[random], &A[high]);
#endif
int pivot = A[high];
```

Coding Assignment 4

```
int partition (int A[], int low, int high)
{
    int i, j = 0;
    #ifdef QSM
    int middle = (high+low)/2;
    swap(&A[middle], &A[high]);
    #elif QSRND
    int random = (rand() % (high-low+1)) + low;
    swap(&A[random], &A[high]);
    #endif
    int pivot = A[high];
```

Coding Assignment 4

	PC tics			Omega tics		
n	QSR	QSM	QSRND	QSR	QSM	QSRND
1,024						
10,000						
50,000						
100,000						
500,000						
1,000,000						
2,000,000						

Coding Assignment 4

	PC tics - 100,000					Omega tics - 100,000				
	T1 Random	T2 1/4 ordered 3/4 random	T3 1/2 ordered 1/2 random	T4 3/4 ordered 1/4 random	T5 Ordered	T1 Random	T2 1/4 ordered 3/4 random	T3 1/2 ordered 1/2 random	T4 3/4 ordered 1/4 random	T5 Ordered
QSR										
QSM										
QSRND										

OLQ

- The Quick Sort code from lecture will be provided in the quiz.
- You will be provided with a copy of this table for reference.

i	j	pivot	low	high

OLQ

You will be given an array to work with (for example)

{9, 7, 5, 12, 11}

Assume that a function to print the entire array is included the line after each call to function `swap()` inside `partition()`.

This call will be in the code provided with the quiz itself and will be in the OLQ Study Guide.

```
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
         printArray(A);
      }
   }
   swap(&A[i + 1], &A[high]);
   printArray(A);

   return (i + 1);
}
```

OLQ

OLQ

For your given array, you will need to write the output of every call to this function - what does the array look like after each call to function `swap()`?

For example, if your given array is

```
{9,7,5,12,11}
```

then your answer will be

loop swap 9 9	{9,7,5,12,11}
loop swap 7 7	{9,7,5,12,11}
loop swap 5 5	{9,7,5,12,11}
final swap 12 11	{9,7,5,11,12}
final swap 9 5	{5,7,9,11,12}
loop swap 7 7	{5,7,9,11,12}
final swap 9 9	{5,7,9,11,12}

{9, 7, 5, 12, 11}

OLQ

loop swap 9 9	{9,7,5,12,11}
loop swap 7 7	{9,7,5,12,11}
loop swap 5 5	{9,7,5,12,11}
final swap 12 11	{9,7,5,11,12}
final swap 9 5	{5,7,9,11,12}
loop swap 7 7	{5,7,9,11,12}
final swap 9 9	{5,7,9,11,12}

```
i = low -1

for (j = low -> j < high)
{
  if A[j] < pivot
    move i
    swap A[i] A[j]
}
swap A[i+1] with A[high]
```

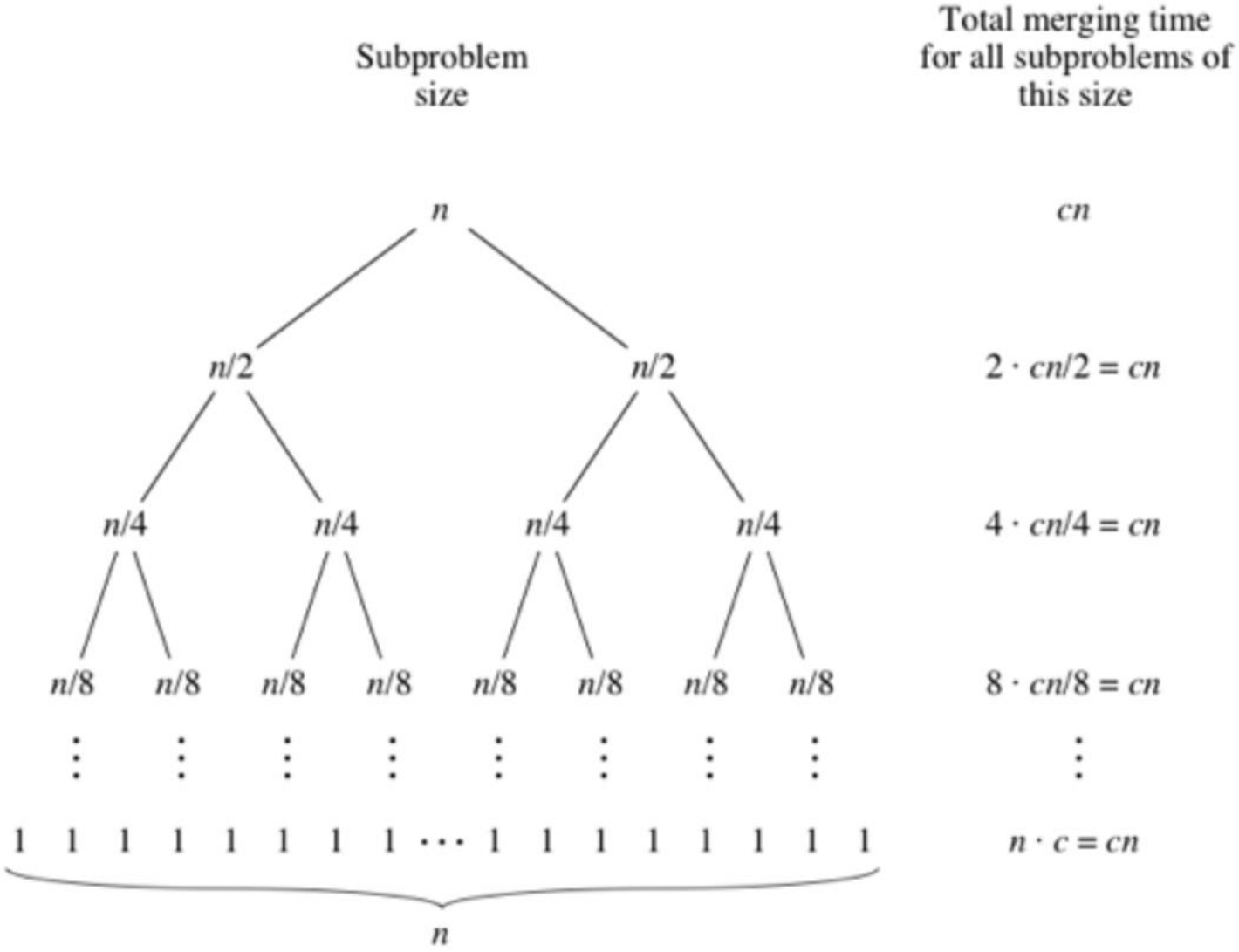
OLQ

{7,12,5,9,11}

loop swap 7 7	{7,12,5,9,11}
loop swap 12 5	{7,5,12,9,11}
loop swap 12 9	{7,5,9,12,11}
final swap 12 11	{7,5,9,11,12}
loop swap 7 7	{7,5,9,11,12}
loop swap 5 5	{7,5,9,11,12}
final swap 9 9	{7,5,9,11,12}
final swap 7 5	{5,7,9,11,12}

```
i = low -1

for (j = low -> j < high)
{
  if A[j] < pivot
    move i
    swap A[i] A[j]
}
swap A[i+1] with A[high]
```

Analysis of Merge Sort



The total time merging in Merge Sort is the sum of the merging times for all the levels.

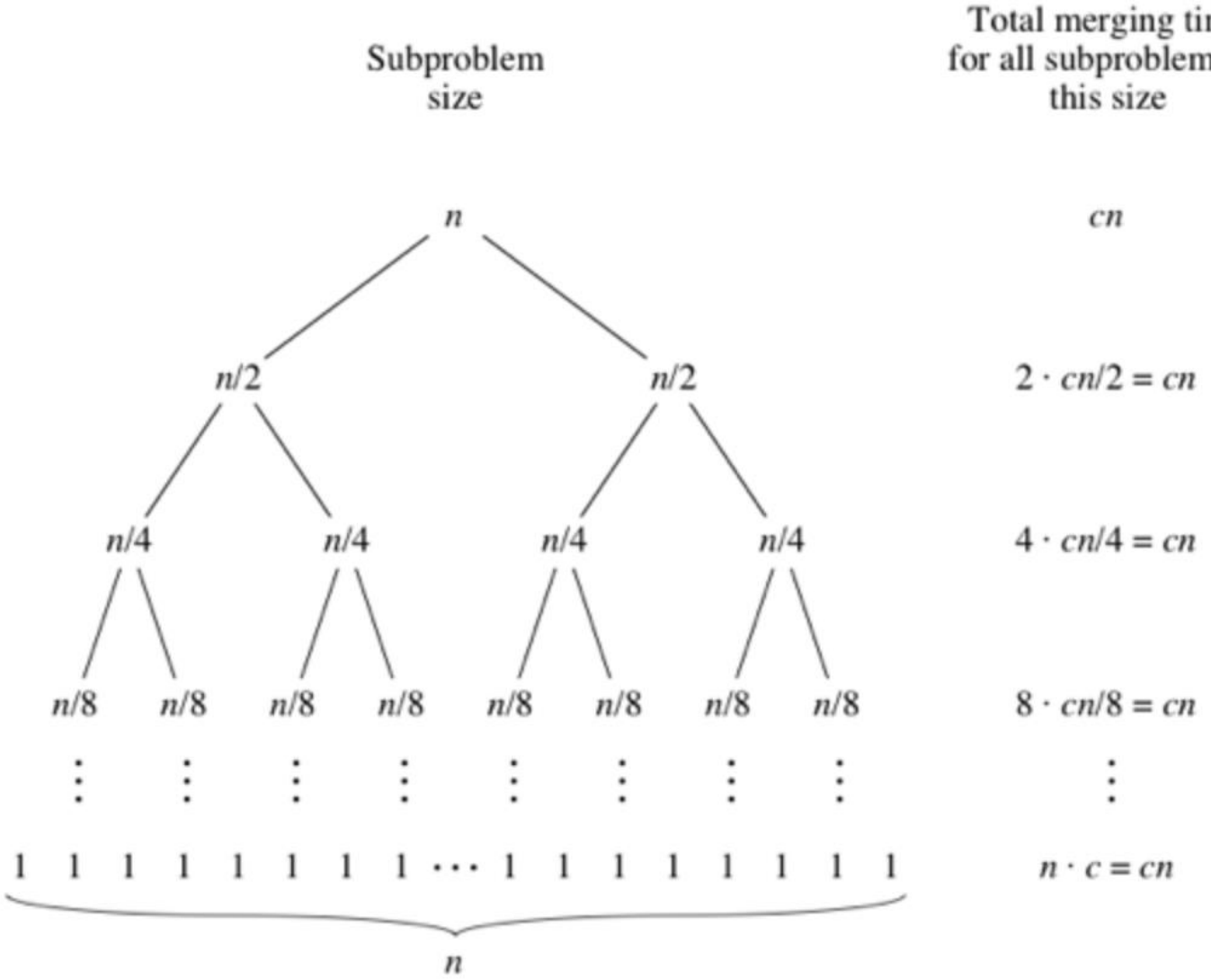
If there are Z levels in the tree, then the total merging time is

Z * *cn*

So what is Z?

Does this pattern look familiar?

$cn\log_2 n$ Analysis of Merge Sort



n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6

Formatted Input and Output

`sscanf()` and `sprintf()`

```
sscanf(buffer, control_string, args, …)

sprintf(buffer, control_string, args, …)
```

`buffer`	buffer in memory
`control_string`	conversion specifier
`args`	argument to conversion specifier

Formatted Input and Output

`sscanf()` **and** `sprintf()`

```
sprintf(buffer, "%s %s has student id %s ",
        first_name, last_name, id);

sscanf(buffer, "%s %s %*s %*s %*s %s", a, b, c);
```

```
char buffer[100] = {};
char first_name[50] = {};
char last_name[50] = {};
char id[10] = {};
char a[50] = {};
char b[50] = {};
char c[10] = {};
```

```
Enter first name Fred

Enter last name Flintstone

Enter id 1000000001
```

```
printf("Enter first name ");
scanf("%s", &first_name);
printf("\nEnter last name ");
scanf("%s", &last_name);
printf("\nEnter id ");
scanf("%s", &id);
```


sprintfDemo.c

```
Breakpoint 2, main () at sprintfDemo.c:23
23                sprintf(buffer, "%s %s has student id %s ",
24                        first_name, last_name, id);
(gdb) p first_name
$1 = "Fred", '\000' <repeats 36 times>"\377, \265\360\000\000\000\000\000",
<incomplete sequence \302>
(gdb) p last_name
$2 =
"Flintstone\000\000\000\000\000\000\000\347\377\377\001\000\000\000\340\366\252\252\25
2*\000\000`\347\252*\000\000\020\350\377\377\377\177\000\000\000", <incomplete sequence
\340>
(gdb) p id
$3 = "1000000001"
(gdb) step


(gdb) p buffer
$4 = "Fred Flintstone has student id 1000000001
\000\000\000\000\000\000\001\000\000\000\000\000\303\000\307\006@", '\000' <repeats 13
times>"\300, \313!\311>\000\000\000\220\006@", '\000' <repeats 13 times>"\220,
\350\377\377"
```


sprintfDemo.c

`%*s` tells `sscanf()` to skip the characters between whitespaces.

```
Fred Flintstone has student id 1000000001

27              sscanf(buffer, "%s %s %*s %*s %*s %s", a, b, c);
(gdb) p a
$5 = "\000\000\000\000\000\000\000\000\002\223\000\311>", '\000' <repeats 11
times>"\340, \366\252\252\252*\000\000\001", '\000' <repeats 15 times>, "\001"
(gdb) p b
$6 = "\000\000\000\000\001\000\000\000\227\a\000\000\001", '\000' <repeats 11 times>,
"`藪\252*\000\000\340\347\377\377\377\177\000\000\220\347\377\377\377\177\000\000.N"
(gdb) p c
$7 = "@\374@\311>\000\000\000\250\002"
(gdb) step
28              printf("First name = %s\nLast Name = %s\nID = %s\n\n", a, b, c);
(gdb) p a
$8 = "Fred\000\000\000\000\002\223\000\311>", '\000' <repeats 11 times>"\340,
\366\252\252\252*\000\000\001", '\000' <repeats 15 times>, "\001"
(gdb) p b
$9 = "Flintstone\000\000\001", '\000' <repeats 11 times>, "`藪
\252*\000\000\340\347\377\377\377\177\000\000\220\347\377\377\377\177\000\000.N"
(gdb) p c
$10 = "1000000001"
```

Adjacency Matrix

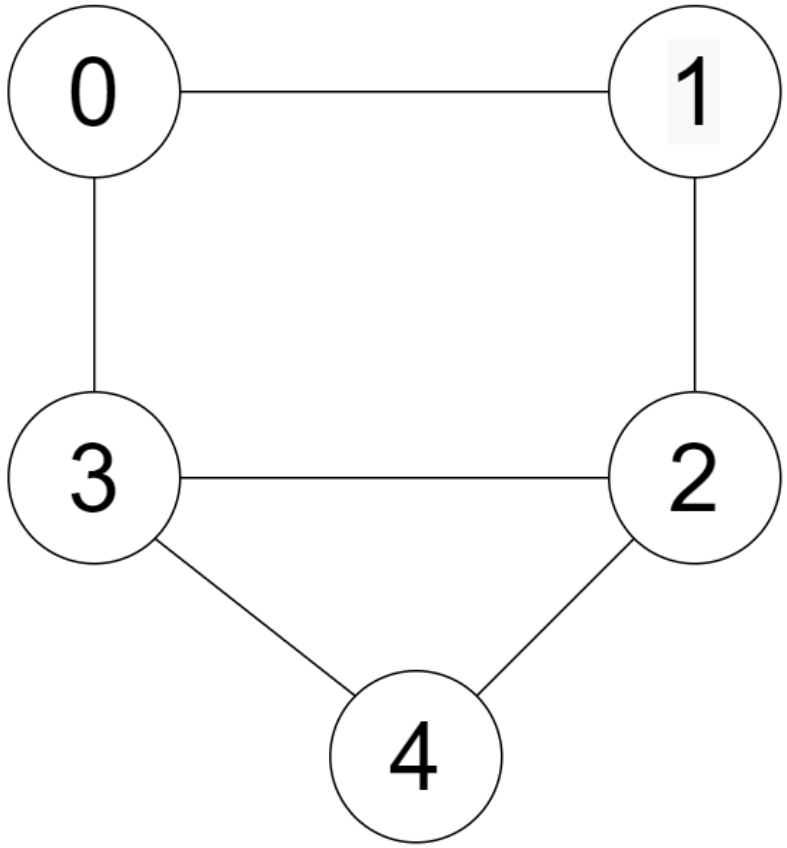
To create an adjacency matrix, we need a list of edges.

This list could be entered at prompts or read from a file.

Using the input list, each edge could be marked with a 1 in the matrix to indicate an edge.

Once the matrix is complete, a vertex's adjacent vertices can be found.

Adjacency Matrix



List the edges for this undirected graph

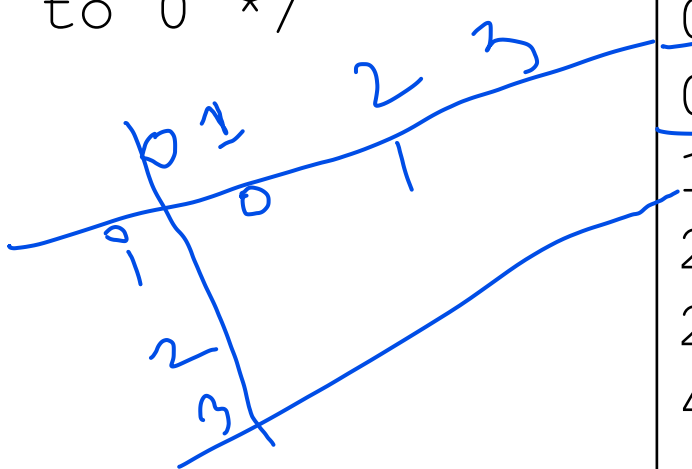
```
0 1
0 3
1 2
2 3
2 4
4 3
```

Adjacency Matrix

```
int main(void)
{
      int AdjMatrix[MAX][MAX];

      CreateAdjacencyMatrix(AdjMatrix);

      PrintAdjacencyMatrix(AdjMatrix);

      FindAdjacentVertex(AdjMatrix);

      return 0;
}
```

Adjacency Matrix

```
void CreateAdjacencyMatrix(int AdjMatrix[][MAX])
{
  int start = 0, end = 0;
  int i = 0, j = 0;
  char buffer[100] = {};
  FILE *FH = fopen("EdgeList.txt", "r+");
  if (FH == NULL)
    exit(0);

  /* initialize adjacency matrix to 0 */
  for(i = 0; i < MAX; i++)
    for(j = 0; j < MAX; j++)
      AdjMatrix[i][j] = 0;
```



EdgeList.txt

```
0 1
0 3
1 2
2 3
2 4
4 3
```

Side Note : Initializing a 2D Array

```
/* initialize adjacency matrix to 0 */
for(i = 0; i < MAX; i++)
  for(j = 0; j < MAX; j++)
    AdjMatrix[i][j] = 0;
```

Inside the function to create the matrix and can be run anytime needed for any value.

vs

```
/* initialize adjacency matrix to 0 */
int AdjMatrix[MAX][MAX] = {0};
```

Can ONLY be done when the array is declared.

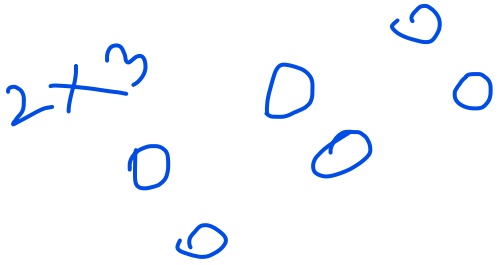
vs

```
/* initialize adjacency matrix to -1 */
int AdjMatrix[MAX][MAX] = {-1};
```

Does NOT set all elements of the array to -1.

Side Note :Initializing a 2D Array

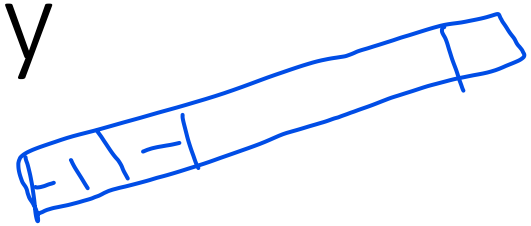
There is another way…



```
memset(AdjMatrix, -1, MAX*MAX*sizeof(int));
```

There are a couple of things you need to understand about using this method to initialize an array.

Side Note :Initializing a 2D Array



```
memset(AdjMatrix, -1, MAX*MAX*sizeof(int));
```

Why use `MAX*MAX*sizeof(int)` instead of `sizeof(AdjMatrix)` or 25?

```
memset(AdjMatrix, -1, sizeof(AdjMatrix));
```

```
GraphAM.c: In function ‘CreateAdjacencyMatrix’:
GraphAM.c:47:33: warning: ‘sizeof’ on array function
parameter ‘AdjMatrix’ will return size of ‘int (*)[5]’
[-Wsizeof-array-argument]
   47 |     memset(AdjMatrix, -1, sizeof(AdjMatrix));
```

Side Note :Initializing a 2D Array

```
memset(AdjMatrix, -1, sizeof(AdjMatrix));


GraphAM.c: In function ‘CreateAdjacencyMatrix’:
GraphAM.c:47:33: warning: ‘sizeof’ on array function parameter ‘AdjMatrix’ will return size of ‘int (*)[5]’ [-Wsizeof-array-argument]
   47 |     memset(AdjMatrix, -1, sizeof(AdjMatrix));
```

But it's just a WARNING and warnings don't matter - right???

Since we still got an executable, let's run it and see if JUST a warning matters……

Side Note :Initializing a 2D Array

```
48        memset(AdjMatrix, -1, sizeof(AdjMatrix));
```

```
(gdb) p AdjMatrix
$1 = (int (*)[5]) 0x7fffffffe230
```

```
(gdb) p *AdjMatrix
$2 = {-1, -1, 0, 0, 0}
```

```
(gdb) p sizeof(AdjMatrix)
$4 = 8
```

```
(gdb) p *AdjMatrix@5
$3 = {{-1, -1, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}}
```

Side Note :Initializing a 2D Array

```
48        memset(AdjMatrix, -1, 25);  //25 is MAX*MAX


(gdb) p *AdjMatrix@5
$1 = {{-1, -1, -1, -1, -1}, {-1, 255, 0, 0, 0}, {0, 0, 0,
0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}}


(gdb) p/x  *AdjMatrix@5
$2 = {{0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff}, {0xffffffff, 0xff, 0x0, 0x0, 0x0}, {0x0, 0x0,
0x0, 0x0, 0x0}, {0x0, 0x0, 0x0, 0x0, 0x0}, {0x0, 0x0,
    0x0, 0x0, 0x0}}
```

Side Note :Initializing a 2D Array

```
48      memset(AdjMatrix, -1, MAX*MAX*sizeof(int));


(gdb) p *AdjMatrix@5
$1 = {{-1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1}, {-1,
-1, -1, -1, -1}, {-1, -1, -1, -1, -1}, {-1, -1, -1, -1,
-1}}
(gdb) p *AdjMatrix@6
$2 = {{-1, -1, -1, -1, -1}, {-1, -1, -1, -1, -1}, {-1,
-1, -1, -1, -1}, {-1, -1, -1, -1, -1}, {-1, -1, -1, -1,
-1}, {32767, -314081792, 1294833030, 0, 0}}
```

Adjacency Matrix

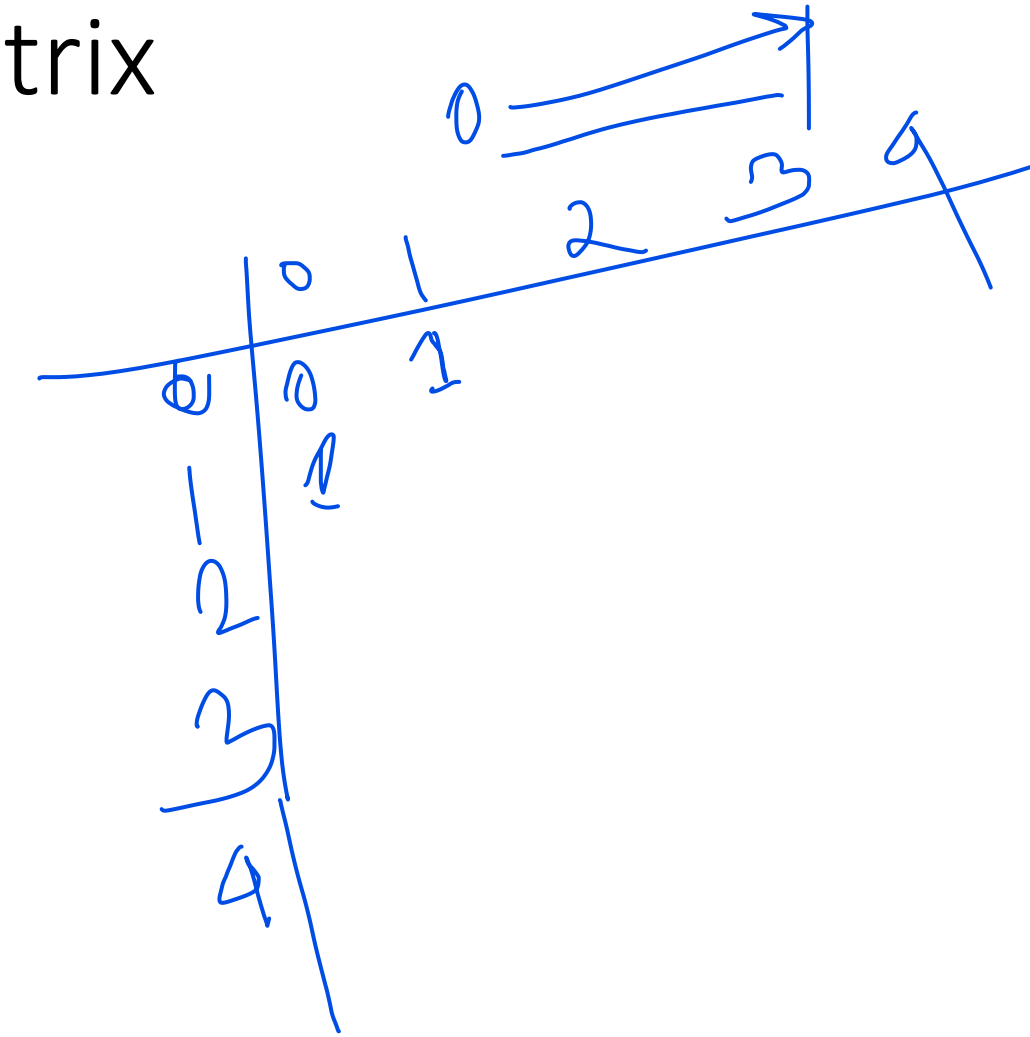
```
while (fgets(buffer, sizeof(buffer)-1, FH))
{
 sscanf(buffer, "%d %d", &start, &end);
  AdjMatrix[start][end] = 1;
  #ifdef UNDIRECTED
  AdjMatrix[end][start] = 1;
  #endif
}

 fclose(FH);
}
```

EdgeList.txt

```
0 1
0 3
1 2
2 3
2 4
4 3
```

Adjacency Matrix

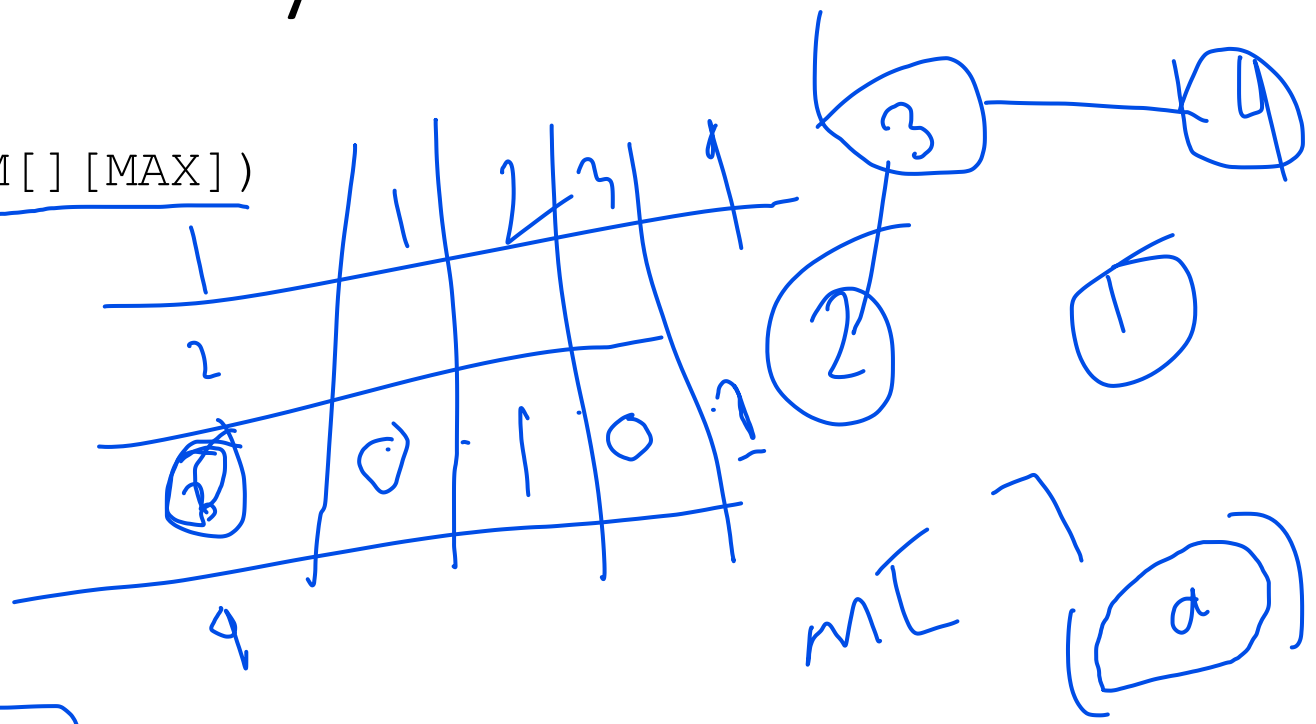
```
AdjMatrix[start][end] = 1;
#ifdef UNDIRECTED
AdjMatrix[end][start] = 1;
#endif
```



```
gcc GraphAM.c -g

gcc GraphAM.c -g -D UNDIRECTED
```

Adjacency Matrix

```
void FindAdjacentVertex(int AM[][MAX])
{
   int SearchVertex = 0;
   int i = 0;

   printf("Enter a vertex ");
   scanf("%d", &SearchVertex);

   for (i = 0; i < MAX; i++)
   {
      if (AM[SearchVertex][i])
         printf("Vertex %d is adjacent to vertex %d\n", SearchVertex, i);
   }
}
```



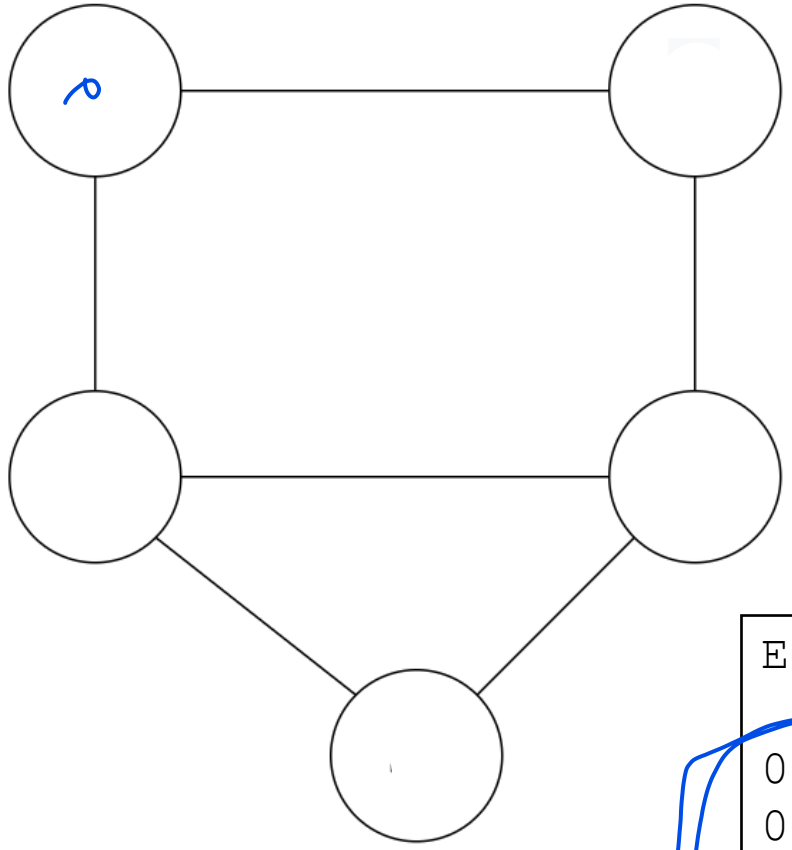
```
for (i = 0; i < MAX; i++)
{
   if (AM[SearchVertex][i])
      printf("Vertex
}
```

Adjacency Matrix

```
gcc GraphAM.c -g -D UNDIRECTED
```

		✓		

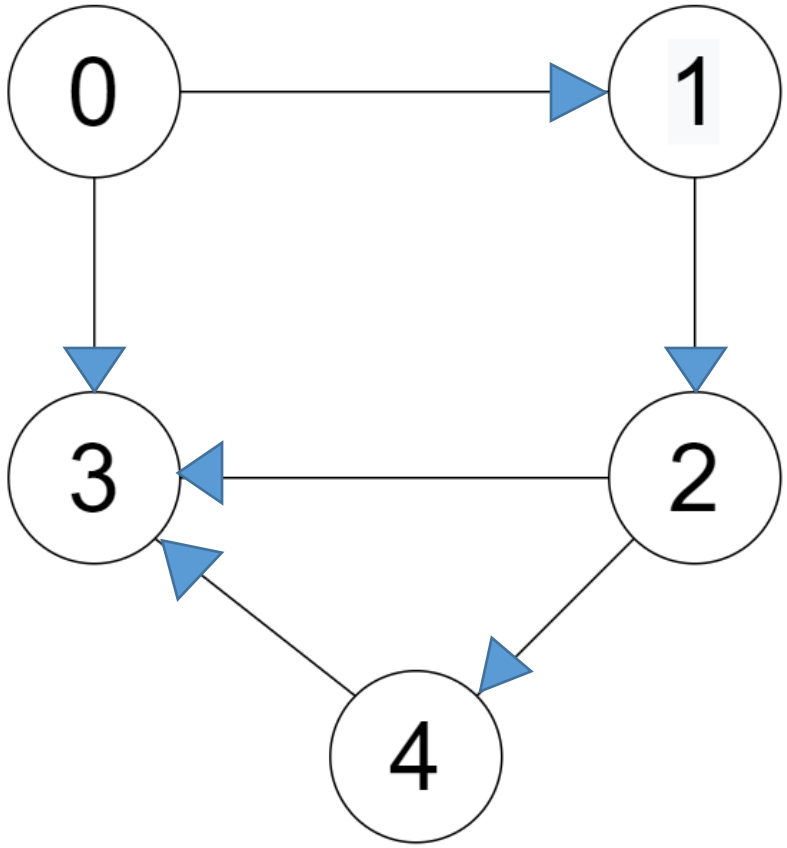
```
01010
10100
01011
10101
00110
Enter a vertex 3
Vertex 3 is adjacent to vertex 0
Vertex 3 is adjacent to vertex 2
Vertex 3 is adjacent to vertex 4
```



EdgeList.txt

```
0 1
0 3
1 2
2 3
2 4
4 3
```

Adjacency Matrix



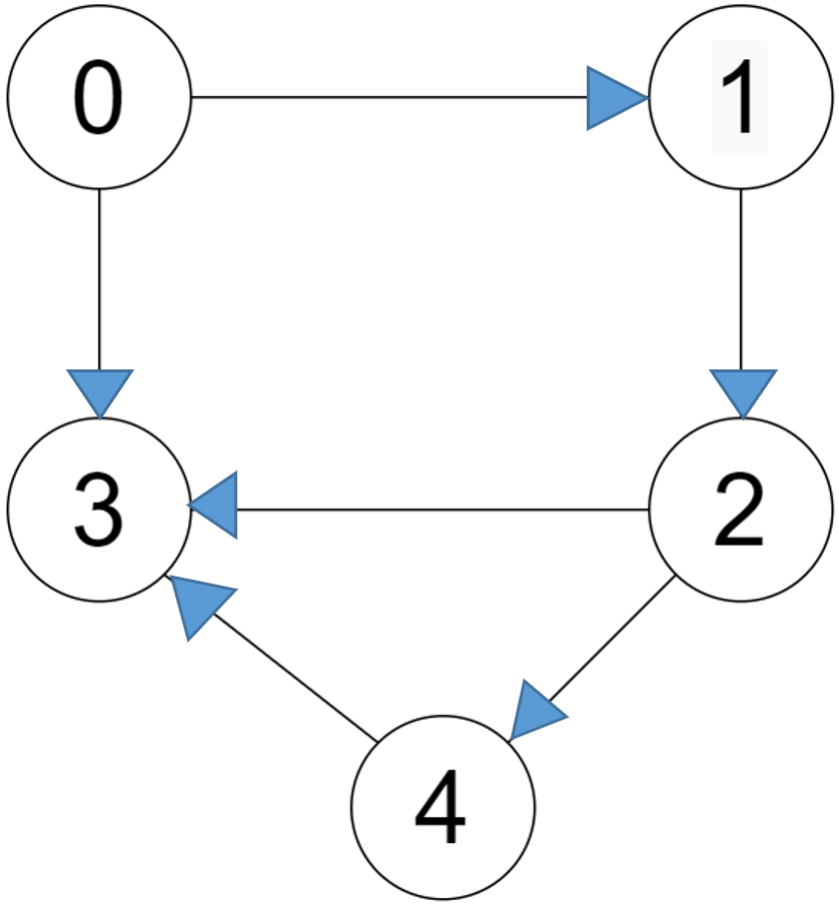

```
0 1
0 3
1 2
2 3
2 4
4 3
```

If this input file was for a directed graph, what would the graph look like based on the file?

```
for (i = 0; i < MAX; i++)
{
   if (AM[SearchVertex][i])
      printf("Vertex
}
```

Adjacency Matrix

```
gcc GraphAM.c -g

01010
00100
00011
00000
00010
Enter a vertex 3

Enter a vertex 2
Vertex 2 is adjacent to vertex 3
Vertex 2 is adjacent to vertex 4
```

Graph vs Tree

A graph is collection of two sets, *V* and *E*, where V is a finite non-empty set of vertices and E is a finite non-empty set of edges.

Vertices can also be called the nodes in the graph.

Two adjacent vertices are joined by edges.

A graph is denoted as G = {*V*, *E*}.

Graph vs Tree

A tree is a finite set of one or more nodes such that –

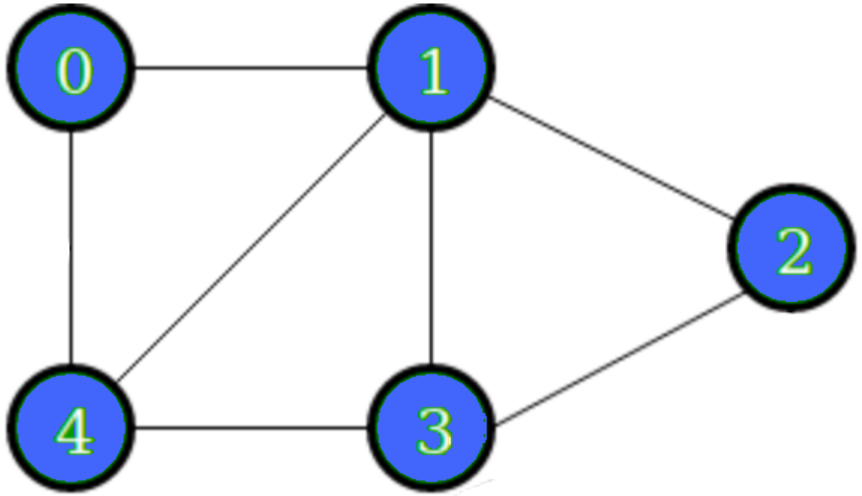
There is a specially designated node called root.

The remaining nodes are partitioned into n >= 0 disjoint sets T_1, T_2, T_3, …, T_n

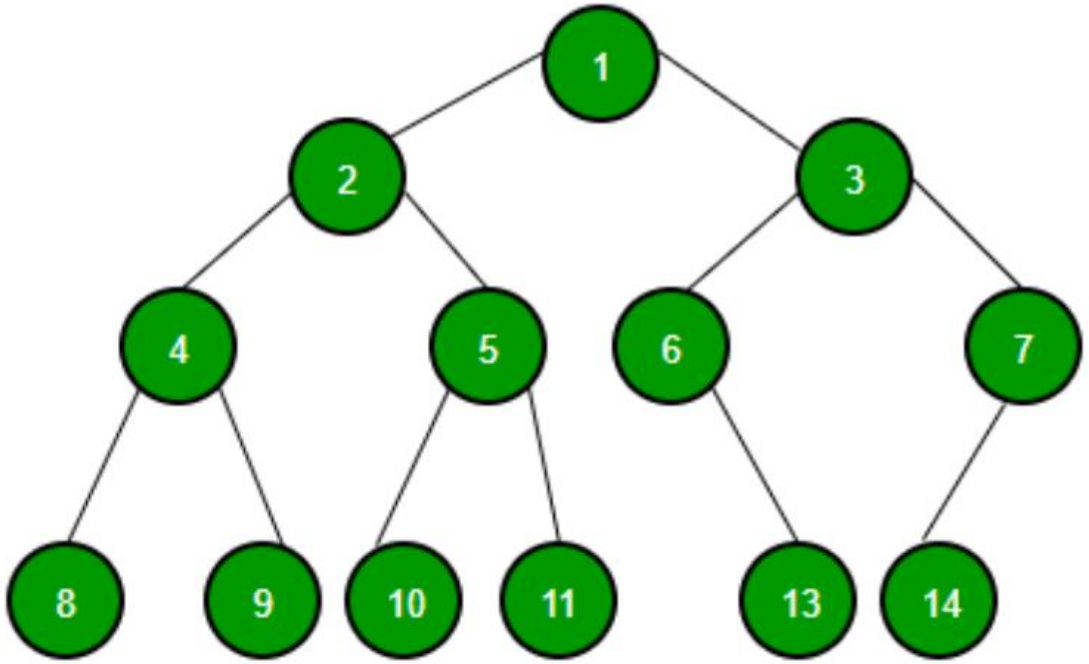
where T_1, T_2, T_3, …, T_n are called the subtrees of the root.

Graph vs Tree

Graph



Tree

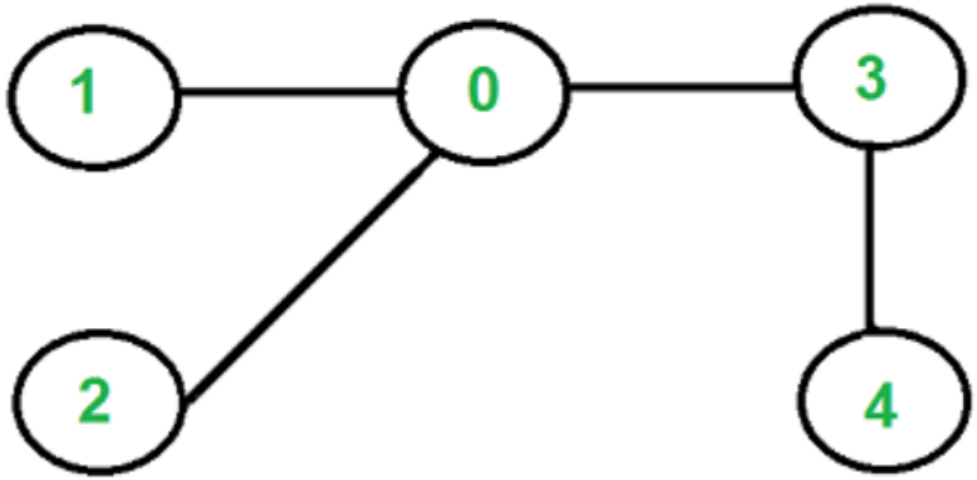


Graph vs Tree

Graph	Tree

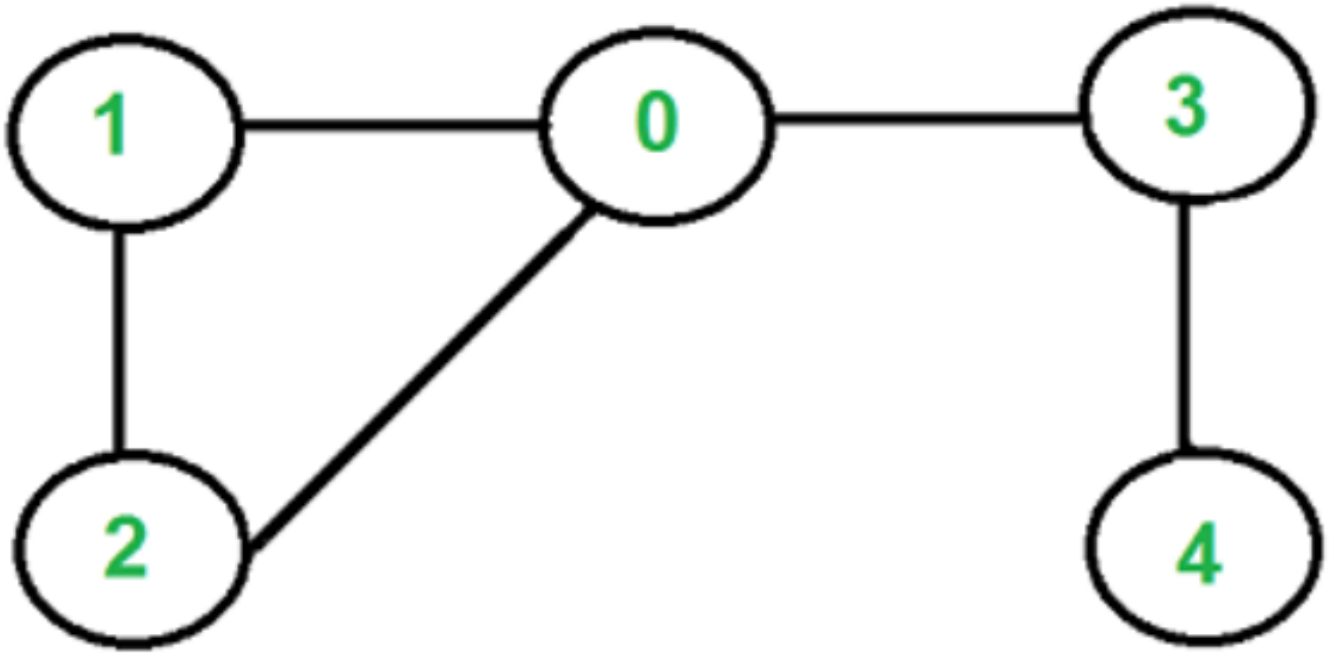
Graph vs Tree

Graph or tree or both?



Graph vs Tree

Graph or tree or both?



Graph vs Tree

Trees are a special case of graph.

Trees are minimally connected graph.

Every tree can be considered a graph, but every graph cannot be considered a tree.

Cycles are not allowed in trees but are allowed in graphs.

Graph vs Tree

If you can get from vertex A to vertex B by traveling over a sequence of edges, then we say that there is a **path** between them.

If there is a **path** between every pair of vertices, then we say the graph is **connected**. This does not mean that every vertex has an edge connecting it to every other vertex.

A connected graph with no cycles is called a **tree**. A tree is a minimally connected graph.

A **cycle** is a path with no repeated edges that repeats a vertex more than once.

In a **free** tree, no vertex/node is specified as the root.

In a **rooted** tree, you pick a vertex as a root and let all the other vertices hang below it.

Graph Traversal

The process of visiting and exploring a graph for processing is called graph traversal.

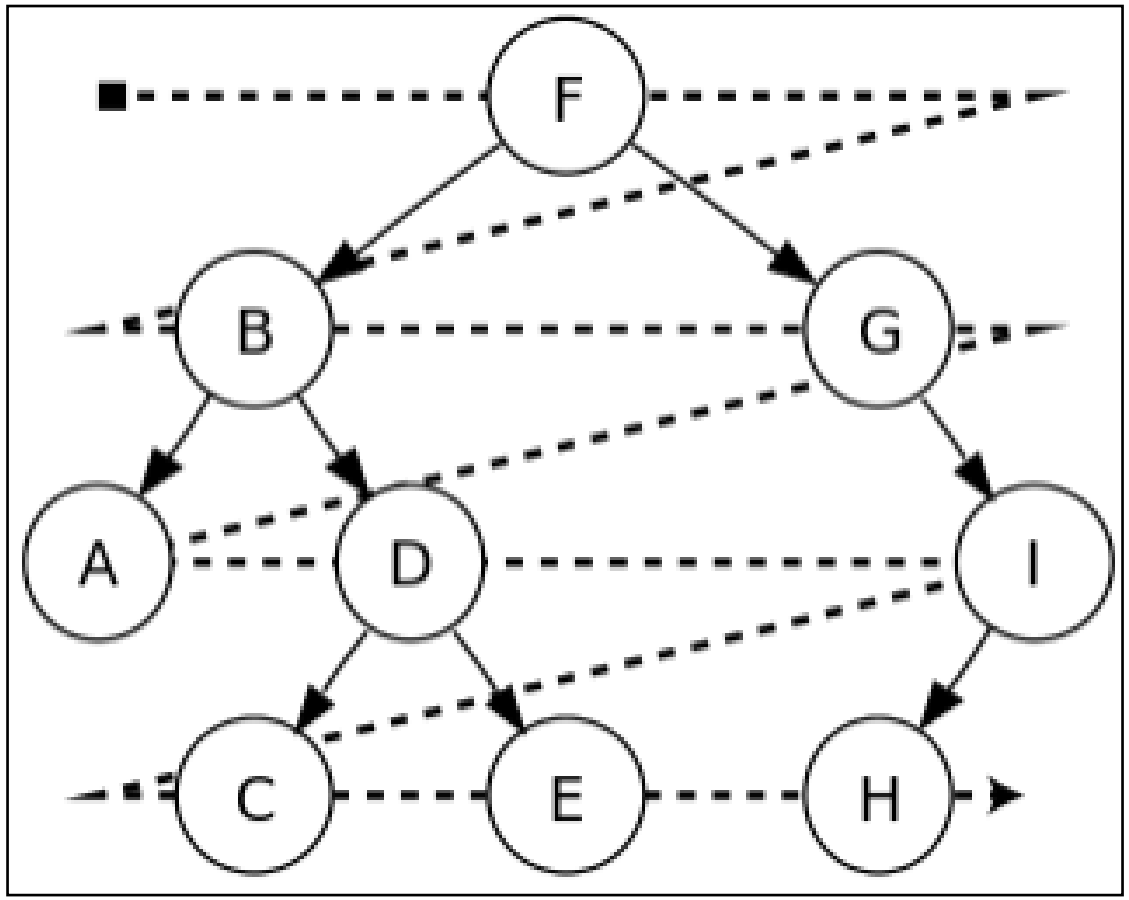
To be more specific it is all about visiting and exploring each vertex and edge in a graph such that all the vertices are explored exactly once.

There are several graph traversal techniques such as Breadth-First Search and Depth First Search.

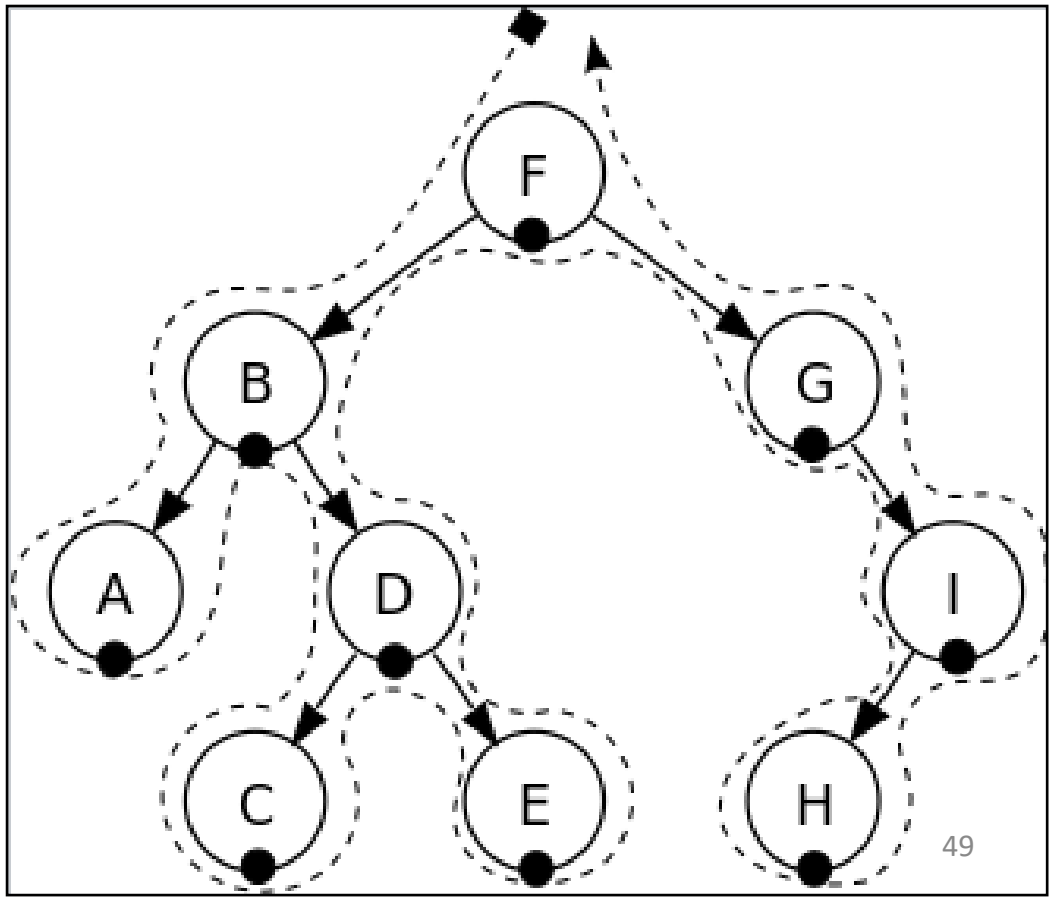
The challenge is to use a graph traversal technique that is most suitable for solving a particular problem.

Breadth-first vs Depth-first Traversal

Breadth-first



Depth-first



Depth-first Traversals

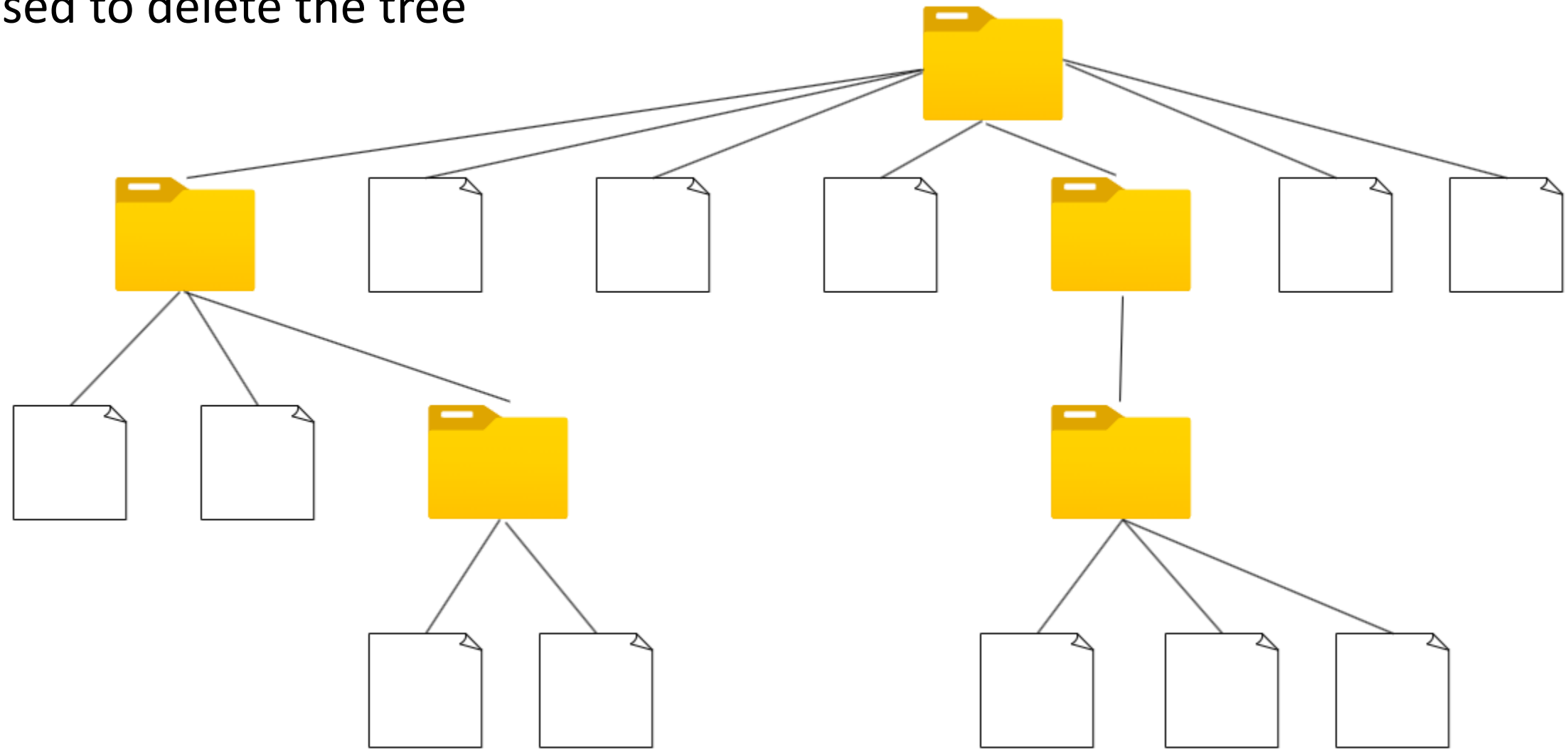
- Inorder Traversal
  - Gives us the nodes in increasing order
- Preorder Traversal
  - Parent nodes are visited before any of its child nodes
  - Used to create a copy of the tree
  - File systems use it to track your movement through directories
- Postorder Traversal
  - Used to delete the tree
  - File systems use it to delete folders and the files under them

Preorder Traversal

Parent nodes are visited before any of its child nodes - used to make a copy

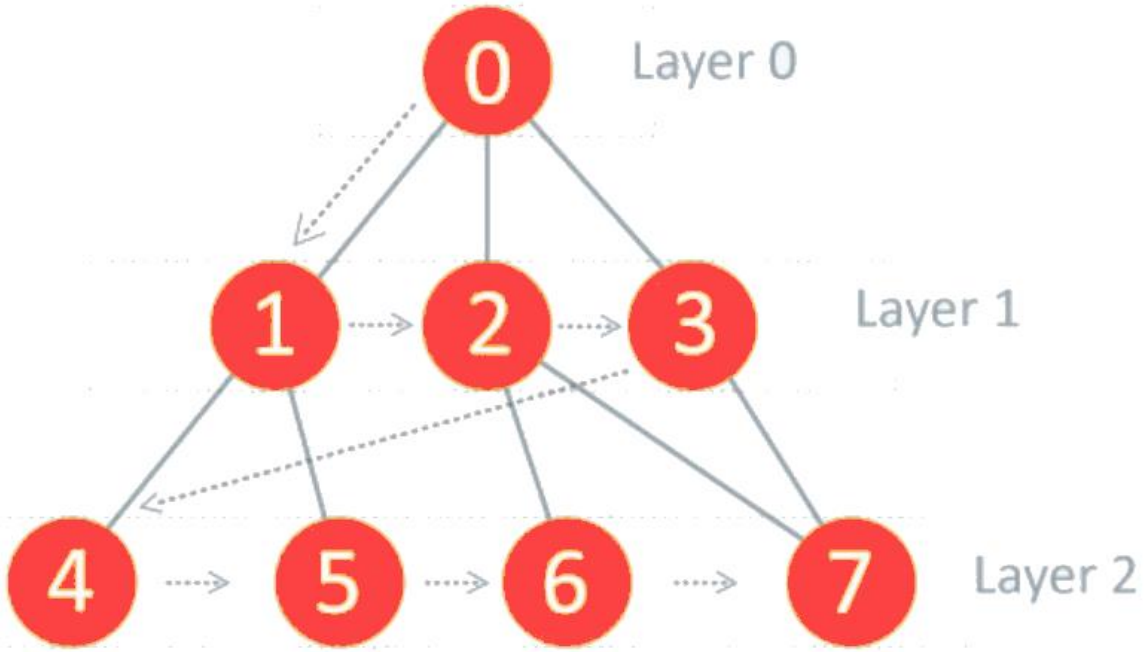
Postorder Traversal

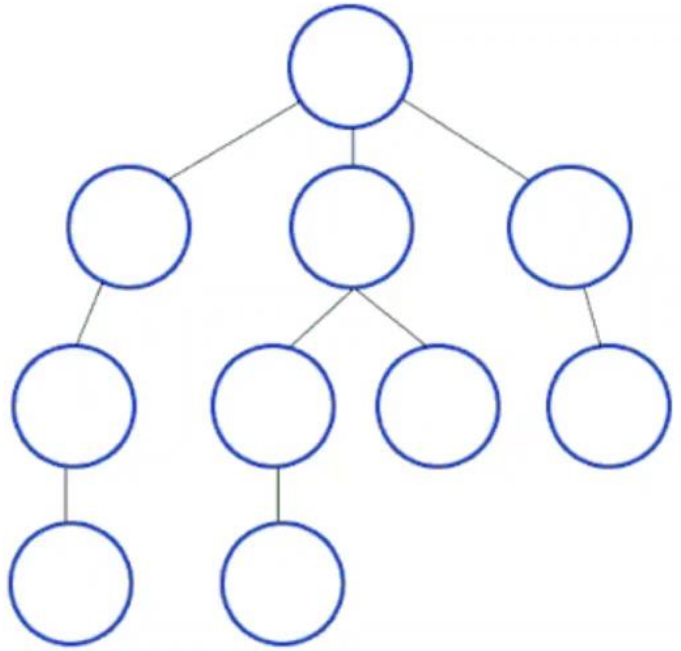
Used to delete the tree



Breadth-first Search

Breadth-First Search algorithm is a graph traversing technique, where you select a random initial vertex and start traversing the graph layer-wise in such a way that all the vertices and their respective adjacent vertices are visited and explored.





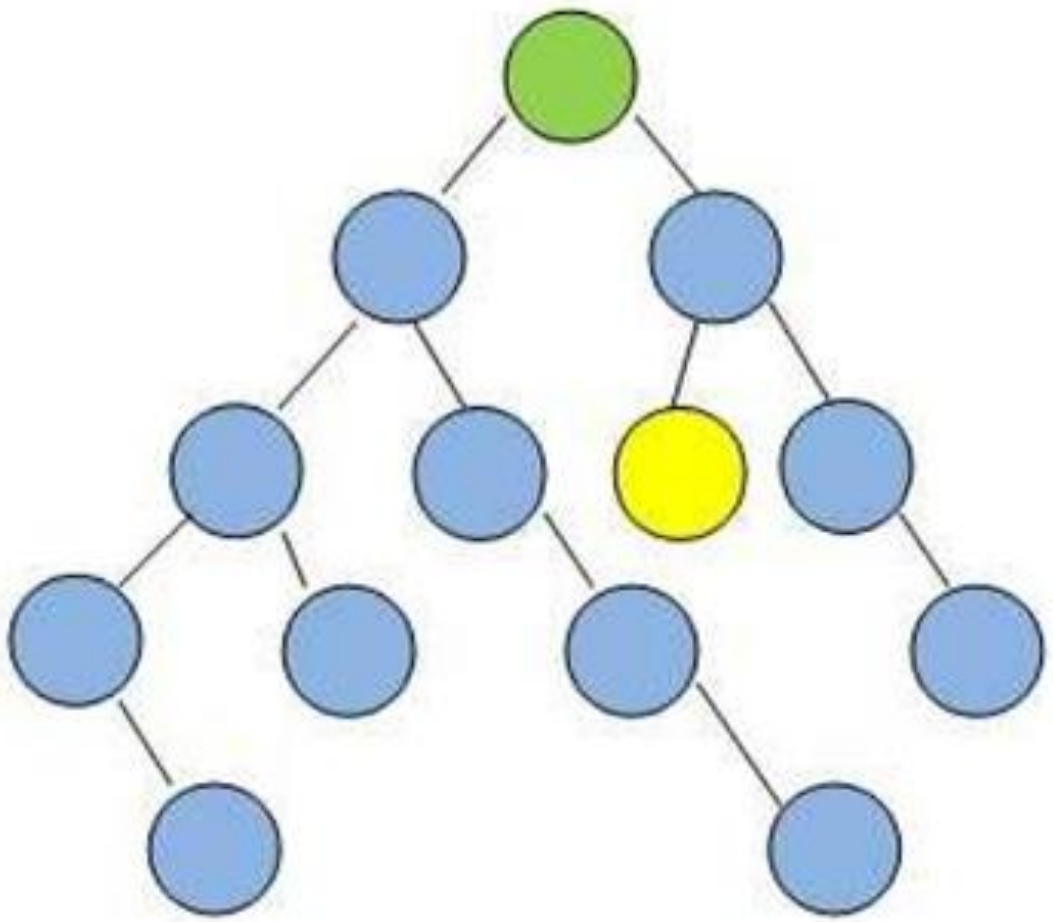
Breadth-first Search

Breadth-first Search can find the shortest path from a source vertex to any other vertex in a graph.

Shortest, in this context, means the least number of edges.

We are going to start with undirected graphs.

Assume that if there is ever a choice amongst multiple nodes, both the BFS and DFS algorithms will choose the left-most node first.



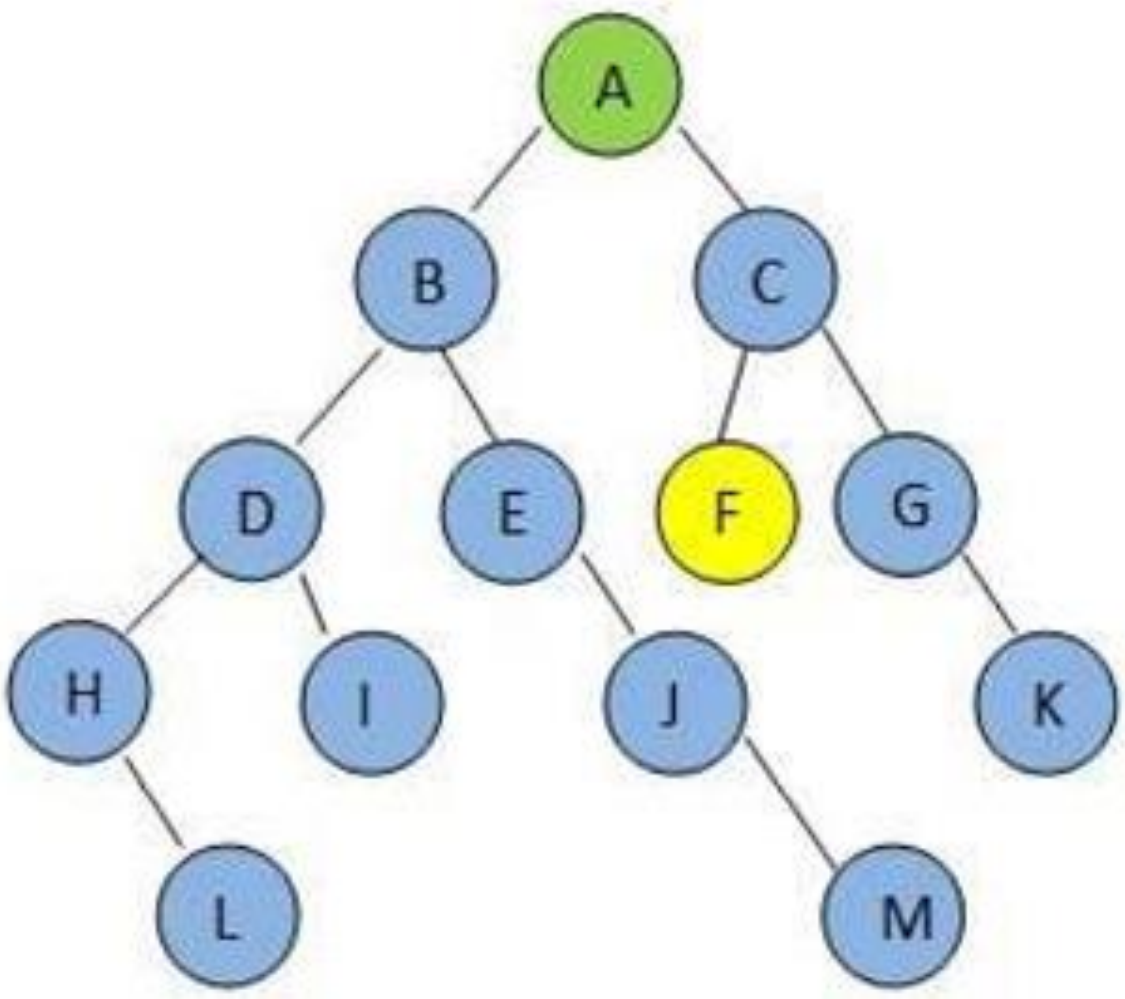
A BFS

B DFS

C Neither BFS nor DFS will ever encounter the goal node in this graph

D BFS and DFS encounter same number of nodes before encounter the goal node

Starting from the green node at the top, which algorithm will visit the least number of nodes before visiting the yellow goal node?

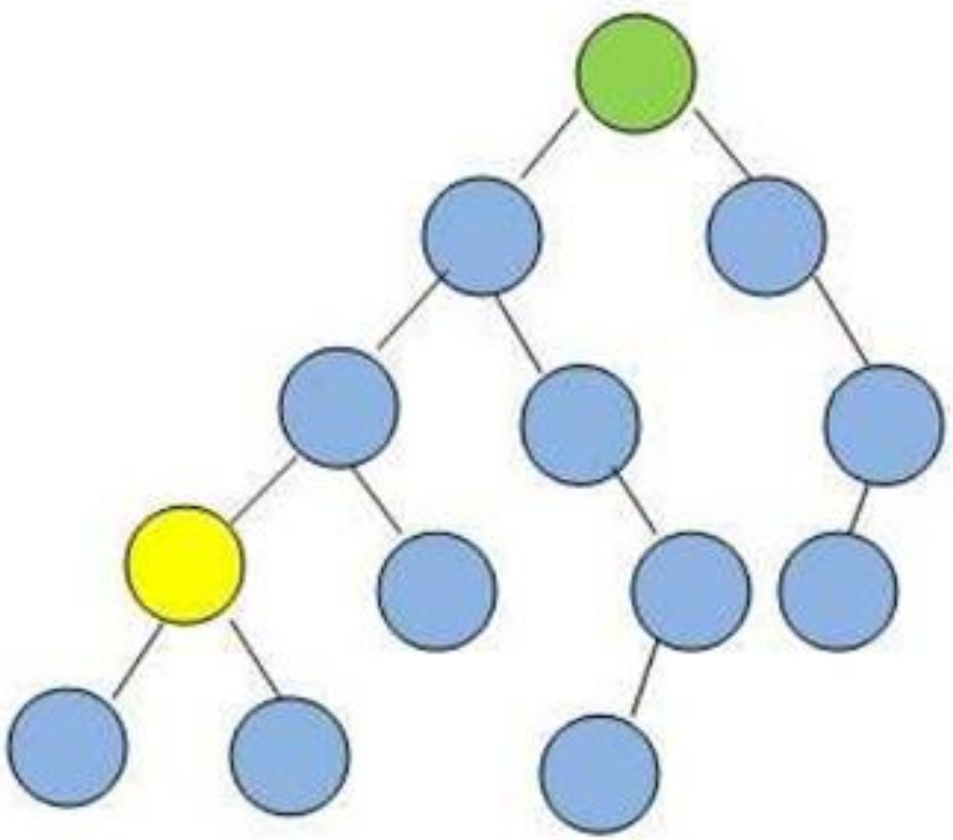


Before encountering goal node F

BFS algorithm encounters nodes: ABCDE

DFS algorithm encounters nodes: ABDHLIEJMC

Assume that if there is ever a choice amongst multiple nodes, both the BFS and DFS algorithms will choose the left-most node first.



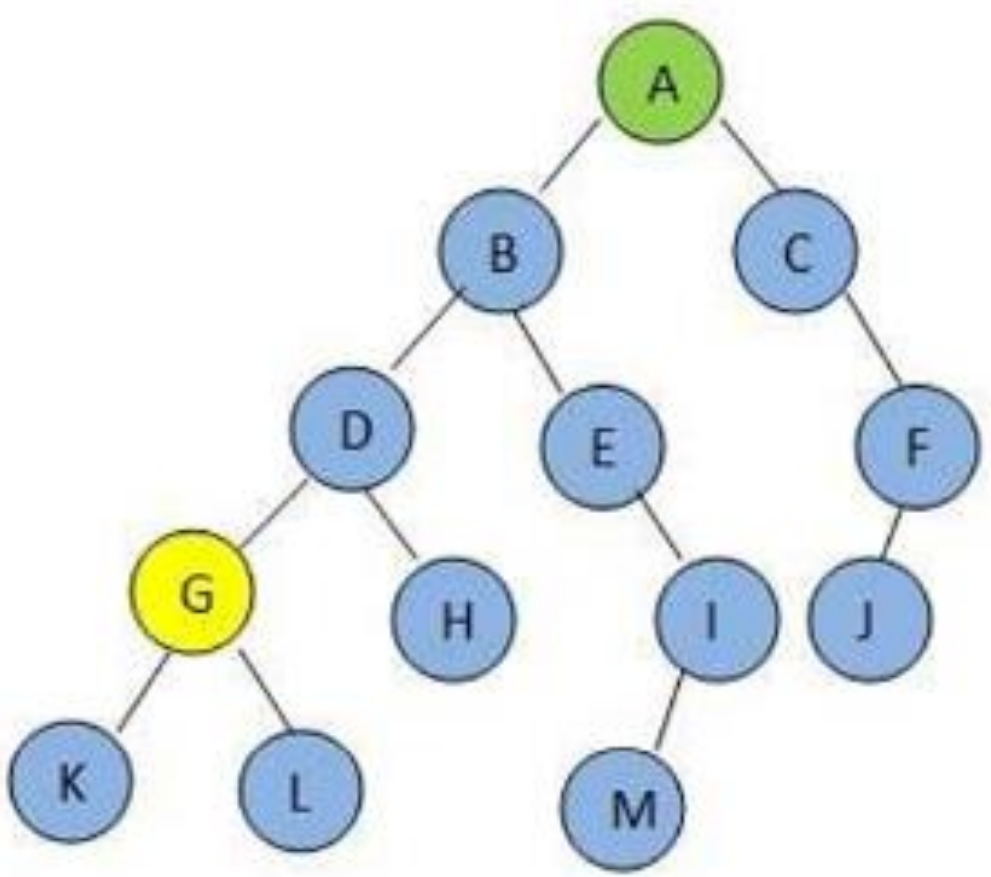
A BFS

B DFS

C Neither BFS nor DFS will ever encounter the goal node in this graph

D BFS and DFS encounter same number of nodes before encounter the goal node

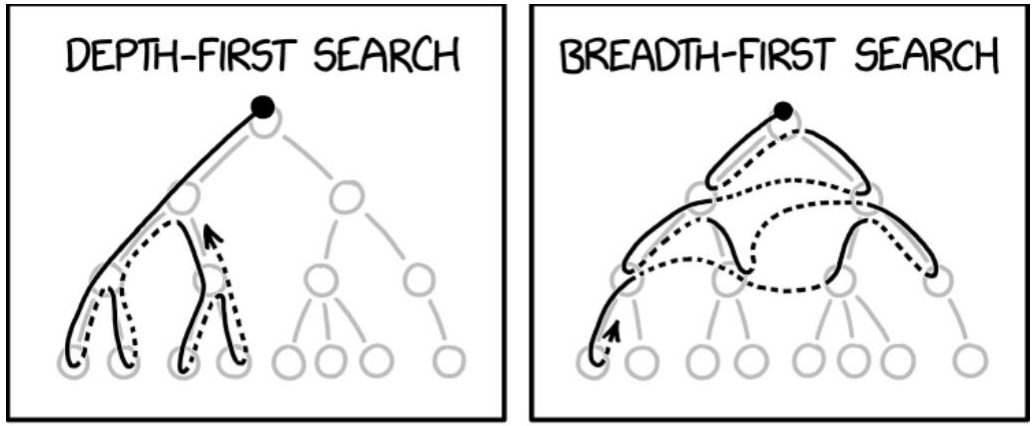
Starting from the green node at the top, which algorithm will visit the least number of nodes before visiting the yellow goal node?



Before encountering goal node G

BFS algorithm encounters nodes: ABCDEF

DFS algorithm encounters nodes: ABD


DEPTH-FIRST SEARCH
BREADTH-FIRST SEARCH

For every 2 balloons you pop, one new balloon is added.

So, if there are 20 balloons, after you pop the first 2, then there will be 19 balloons left.

After you pop the next 2, there are 18 balloons.

How many balloons do you pop before the board is empty?

A: $O(1)$

B: $O(n)$

C: $O(\log_2 n)$

D: $O(n^2)$

When there is 1 balloon, you need to pop 1 balloon.

When there are 2 balloons, you need to pop 3 balloons.

When there are 3 balloons, you need to pop 5 balloons.

When there are 4 balloons, you need to pop 7 balloons.

See a pattern?

When there are 5 balloons, you need to pop 2n-1 balloons

2n - 1 balloons need to be popped to clear the board.

A: $O(1)$

B: $O(n)$

C: $O(\log_2 n)$

D: $O(n^2)$

2n - 1 balloons n
popped to clear

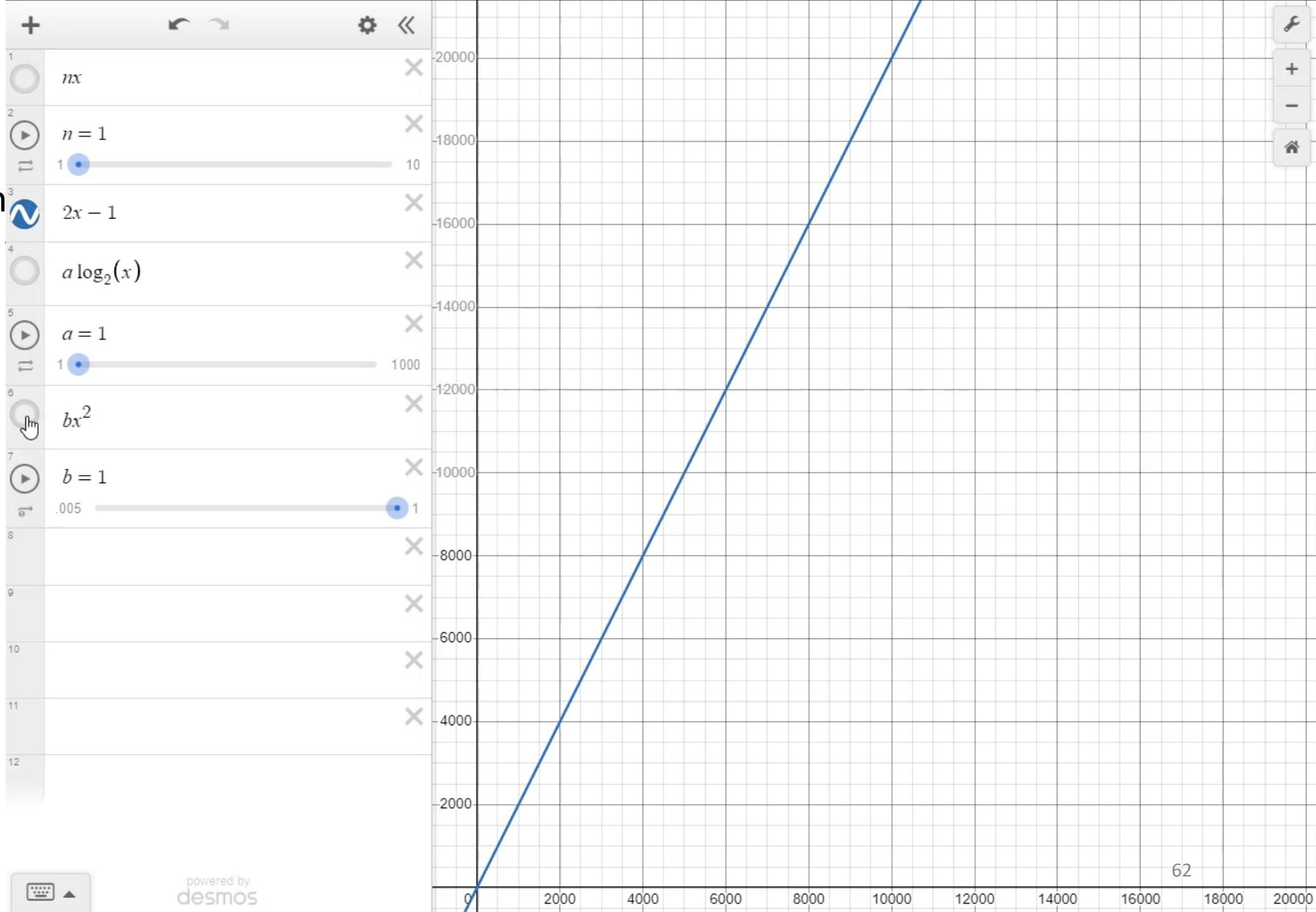
2n - 1

A: O(1)

B: O(n)

C: O($\log_2 n$)

D: O(n^2)



By the time you have popped the first n balloons, n-1 new balloons have been added to the board.

After popping those n-1 balloons, there are n-2 new balloons added to the board.

After popping those n-2 balloons , there are n-3 new balloons on the board.

This same pattern continues until no new balloons are added to the board.

How many total balloons do you pop?

By the time you have popped the first n balloons, n-1 new balloons have been added to the board.

After popping those n-1 balloons, there are n-2 new balloons added to the board.

After popping those n-2 balloons , there are n-3 new balloons on the board.

This same pattern continues until no new balloons are added to the board.

How many total balloons do you pop?

n
+
n-1
+
n-2
+
n-3
+
n-4
+
…
n-(n-1)

20 +
19 +
18 +
17 +
16 +
15 +
14 +
13 +
12 +
11 +
10 +
9 +
8 +
7 +
6 +
5 +
4 +
3 +
2 +
1

$$\frac{n^2 + n}{2}$$

$$\frac{20^2 + 20}{2}$$

210

By the time you have popped the first n balloons, n-1 new balloons have been added to the board.

After popping those n-1 balloons, there are n-2 new balloons added to the board.

After popping those n-2 balloons , there are n-3 new balloons on the board.

This same pattern continues until no new balloons are added to the board.

How many total balloons do you pop?

A: O(1)

B: O(n)

C: $O(\log_2 n)$

D: $O(n^2)$

Side Note : Array Queues

A **queue** is an abstract data structure that follows the

First-In-First-Out (FIFO)

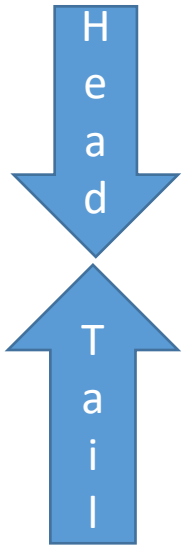
methodology.

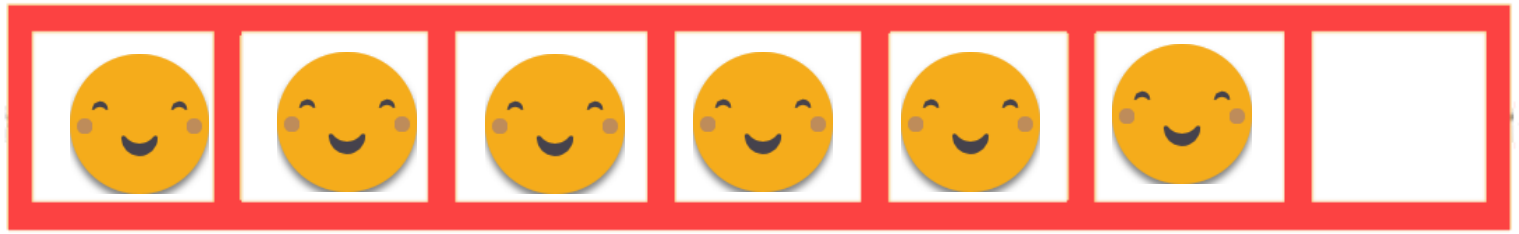
Data inserted first (FI) will be accessed first (FO).

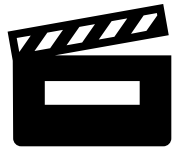
Data is added (enqueue) at one end (tail)

Data is removed (dequeue) at the other end (head).

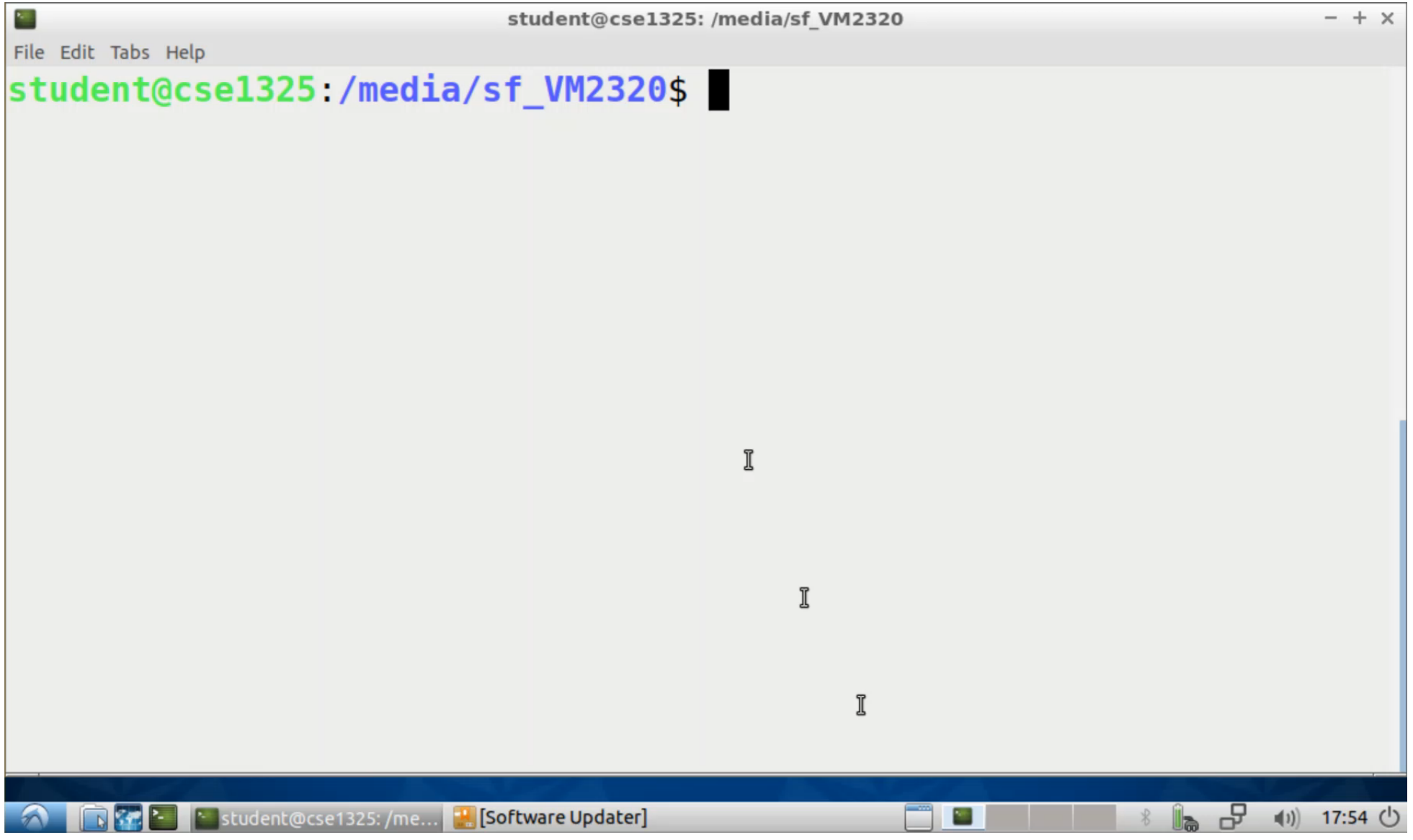
Side Note : Array Queues







Side Note : Array Queues



Side Note : Array Queues

```
65  int main(void)
66  {
67      int QueueArray[MAX];
68      int tail = -1;
69      int head = -1;
```

```
enqueue(QueueArray, &head, &tail);
dequeue(QueueArray, &head, &tail);
```

Side Note : Array Queues

```
12  void enqueue(int QueueArray[], int *head, int *tail)
13  {
14      int enitem = 0;
15
16      if (*head > *tail)
17      {
18          *head = -1;
19          *tail = -1;
20      }
21
22      if (*tail == MAX - 1)
23          printf("Queue Overflow \n");
24      else
25      {
26          if (*head == -1)  /*If queue is initially empty */
27              *head = 0;
28          printf("\nEnter element to enqueue : ");
29          scanf("%d", &enitem);
30          (*tail)++;
31          QueueArray[*tail] = enitem;
32          display(QueueArray, *head, *tail);
33      }
34  }
```

Side Note : Array Queues

```
12  void enqueue(int QueueArray[], int *head, int *tail)
13  {
14      int enitem = 0;
15
16      if (*head > *tail)
17      {
18          *head = -1;
19          *tail = -1;
20      }
```

Side Note : Array Queues

```
22      if (*tail == MAX - 1)
23          printf("Queue Overflow \n");
24      else
25      {
26          if (*head == -1)  /*If queue is initially empty */
27              *head = 0;
28          printf("\nEnter element to enqueue : ");
29          scanf("%d", &enitem);
30          (*tail)++;
31          QueueArray[*tail] = enitem;
32          display(QueueArray, *head, *tail);
33      }
34  }
```

Side Note : Array Queues

```
36  void dequeue(int QueueArray[], int *head, int *tail)
37  {
38      if (*head == -1 || *head > *tail)
39      {
40          printf("\n\nQueue is empty\n\n");
41      }
42      else
43      {
44          printf("\n\nDequeue %d\n", QueueArray[*head]);
45          (*head)++;
46          display(QueueArray, *head, *tail);
47      }
48  }
```

Side Note : Array Queues

```
50  void display(int QueueArray[], int head, int tail)
51  {
52      int i = 0;
53
54      if (head == -1)
55          printf("\n\nQueue is empty\n\n");
56      else
57      {
58          printf("\n\nQueue : ");
59          for (i = head; i <= tail; i++)
60              printf("%d ", QueueArray[i]);
61          printf("\n\n");
62      }
63  }
```

Breadth-first Search

Vertex 0 is the starting vertex.

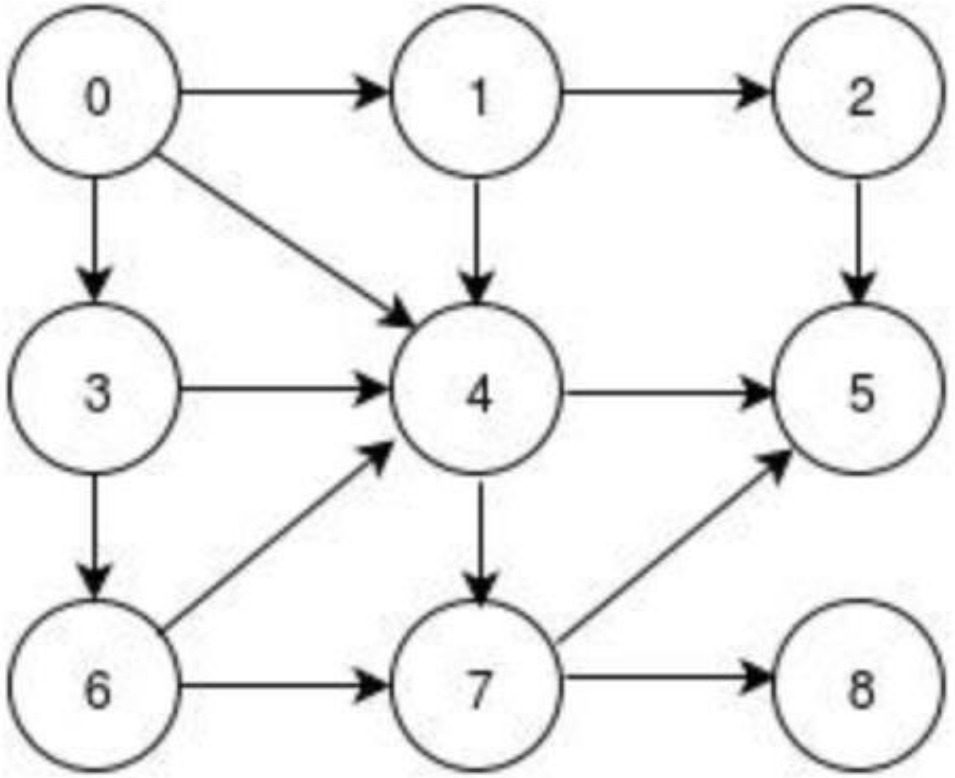
We will visit all vertices adjacent to vertex 0

1, 4, 3

We can visit these three vertices in any order.

A traversal could be any of these combinations :

0 1 3 4	0 1 4 3	0 3 1 4
0 3 4 1	0 4 3 1	0 4 1 3



Breadth-first Search

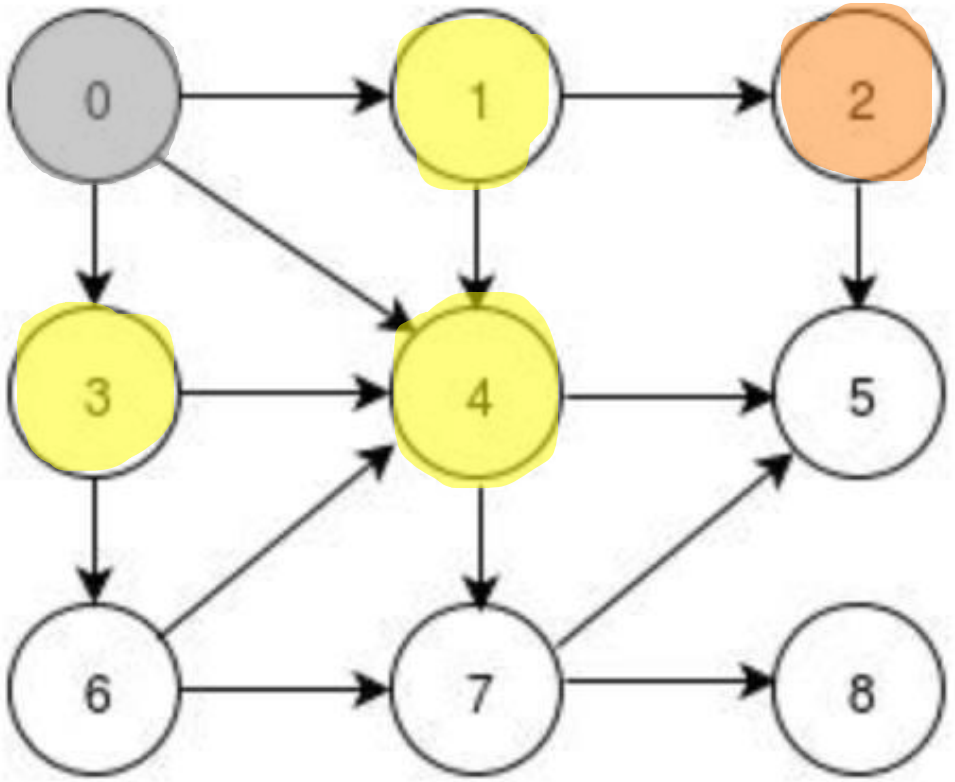
We mark 0, 1, 4 and 3 as "visited".

Now we visit all the vertices adjacent to 1

2, 4 but 4 has already been visited so we skip it

Our traversal is now any of these combinations :

0 1 3 4 2	0 1 4 3 2	0 3 1 4 2
0 3 4 1 2	0 4 3 1 2	0 4 1 3 2



Breadth-first Search

Using traversal

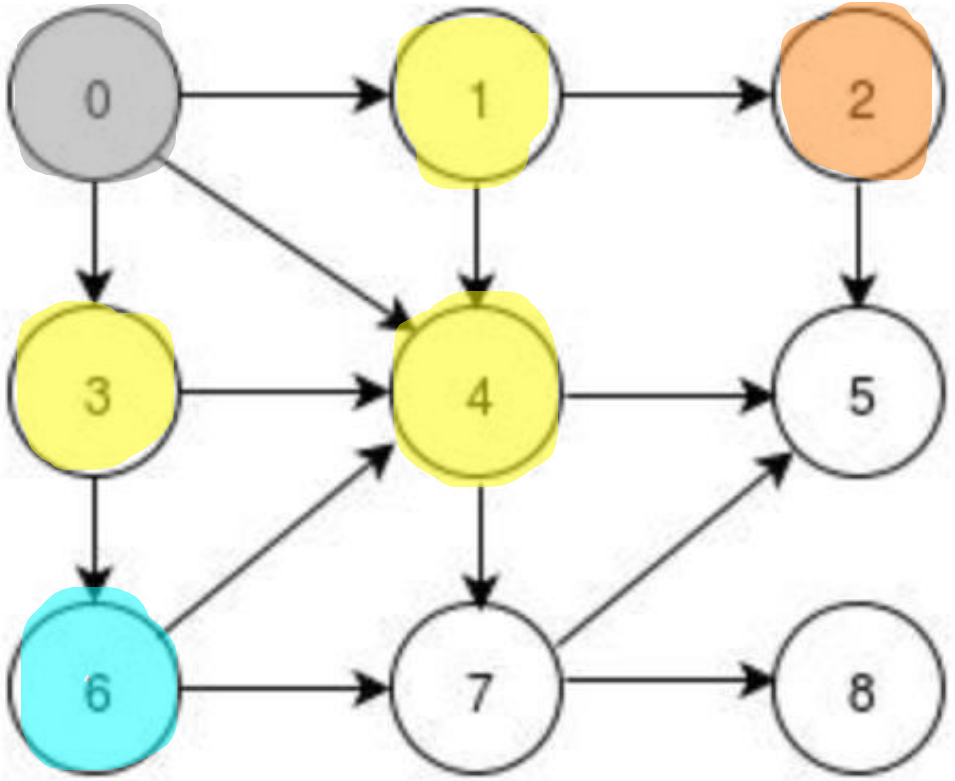
0 1 3 4 2

we will visit all the vertices adjacent to 3

6, 4 but 4 has already been visited so we skip it

Now our traversal is

0 1 3 4 2 6



Breadth-first Search

Using traversal

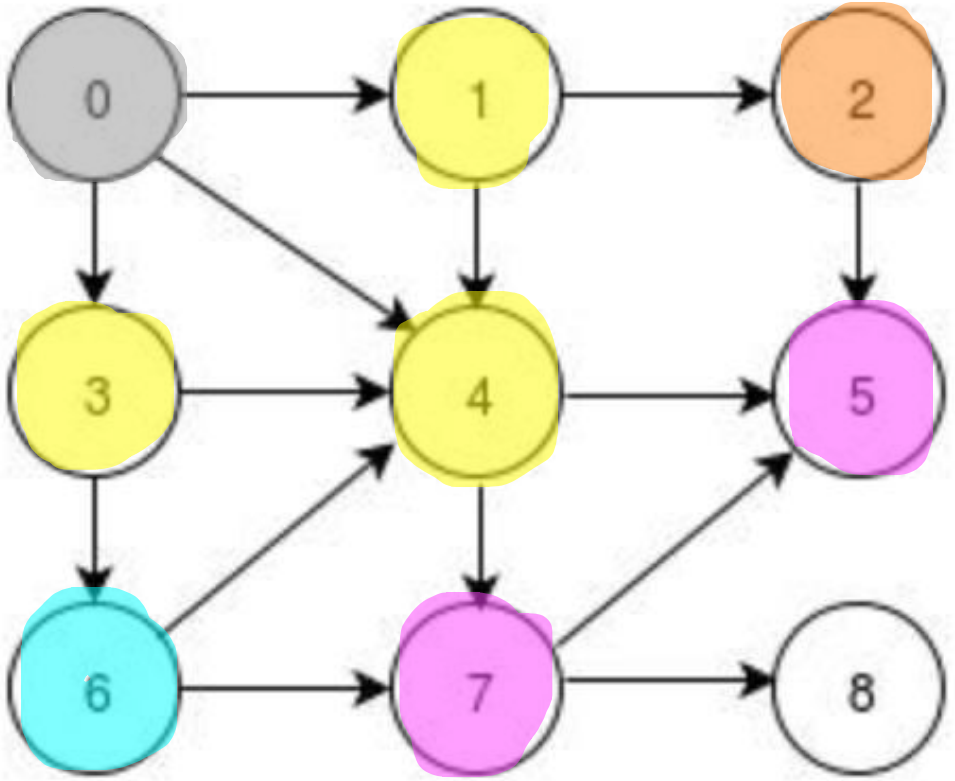
0 1 3 4 2 6

we will visit all the vertices adjacent to 4

5, 7

Now our traversal is

0 1 3 4 2 6 5 7



Breadth-first Search

Using traversal

0 1 3 4 2 6 5 7

we will visit all the vertices adjacent to 2

5 but 5 has already been visited

Now we visit all vertices adjacent to 6

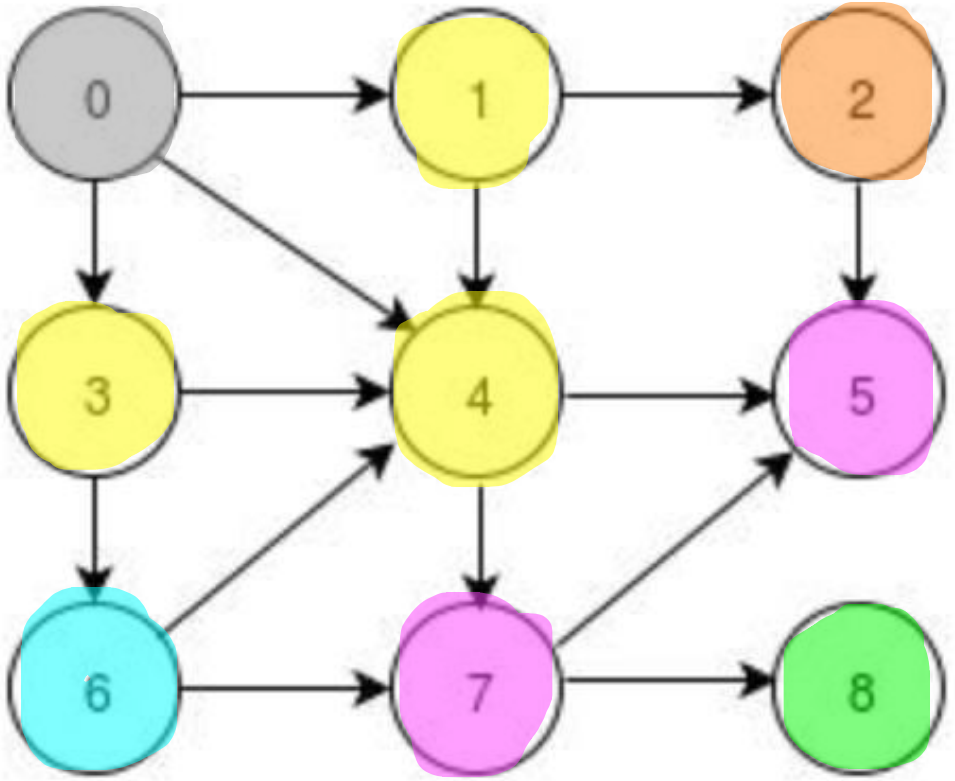
4, 7 – both have already been visited

Now we visit all vertices adjacent to 5

There are none.

Now we visit all vertices adjacent to 7

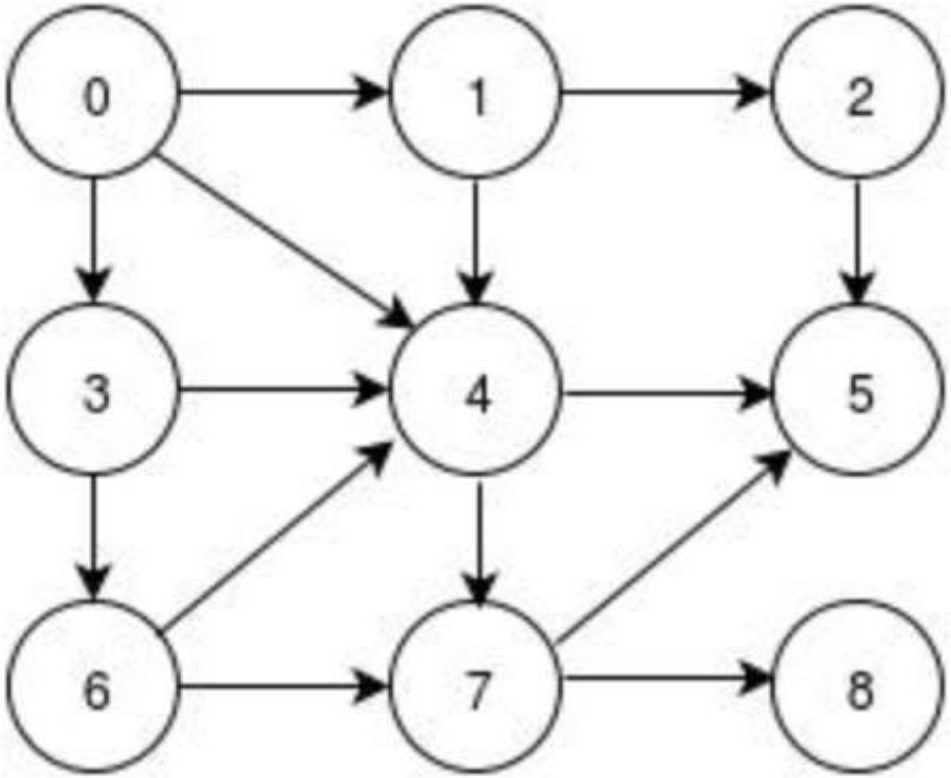
8 has not been visited so we add it to our traversal



0 1 3 4 2 6 5 7 8

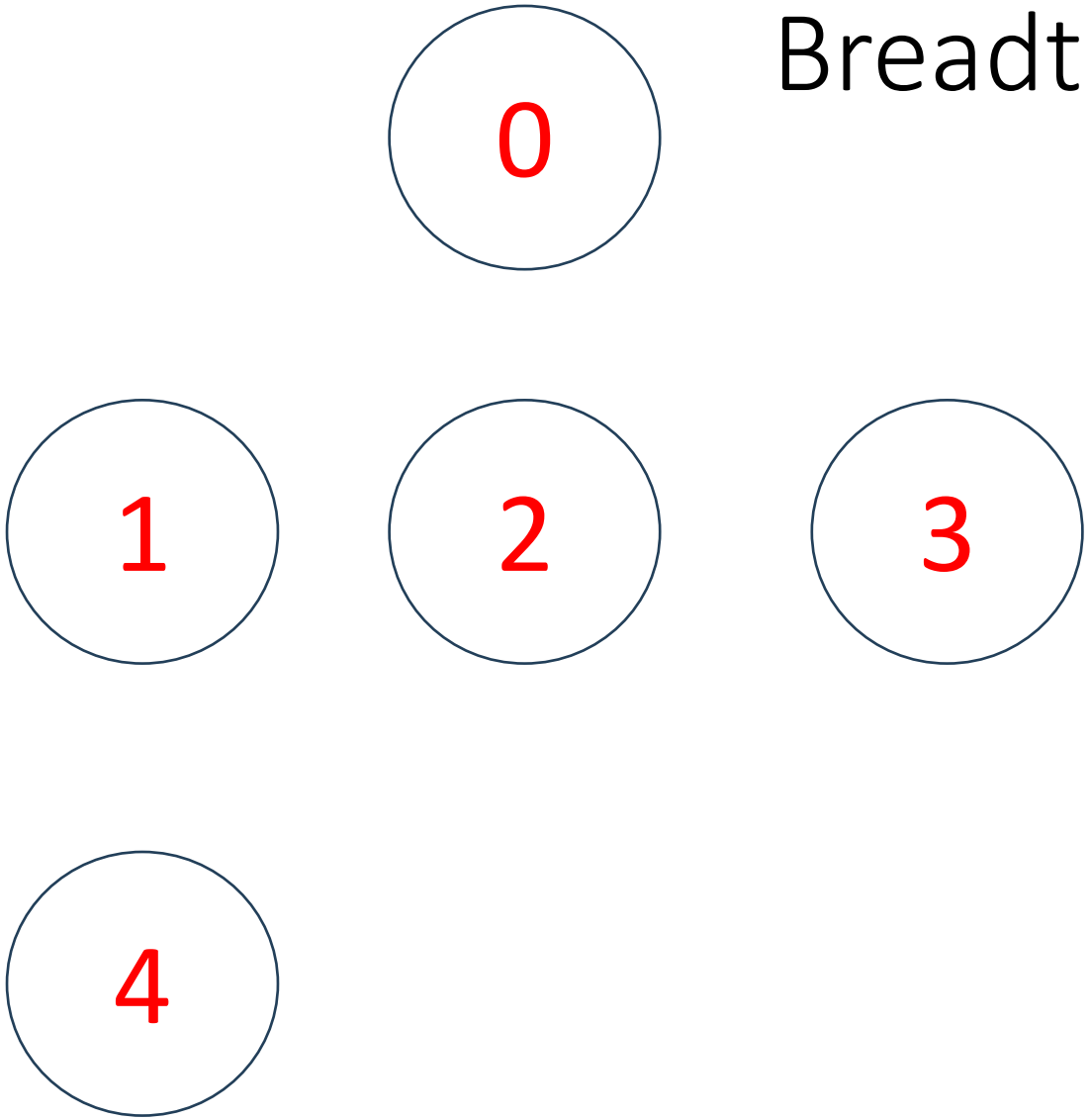
Breadth-first Search

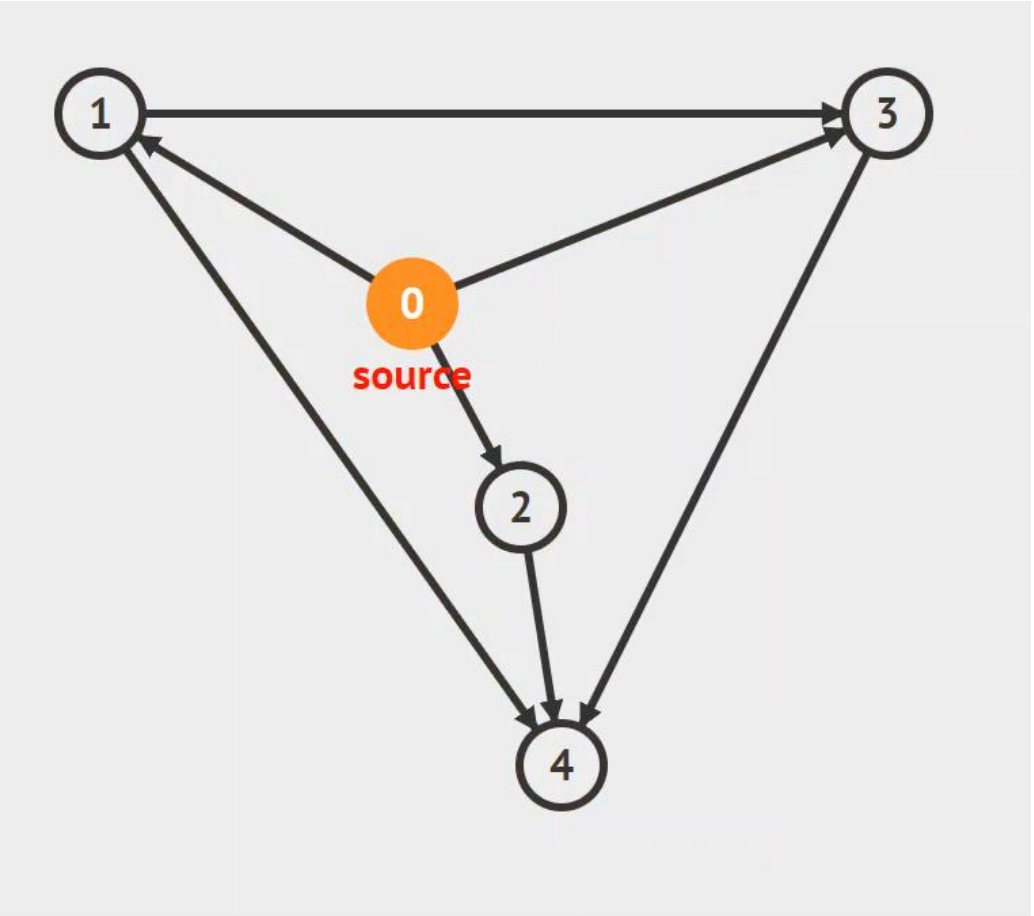
0 1 3 4 2 6 5 7 8



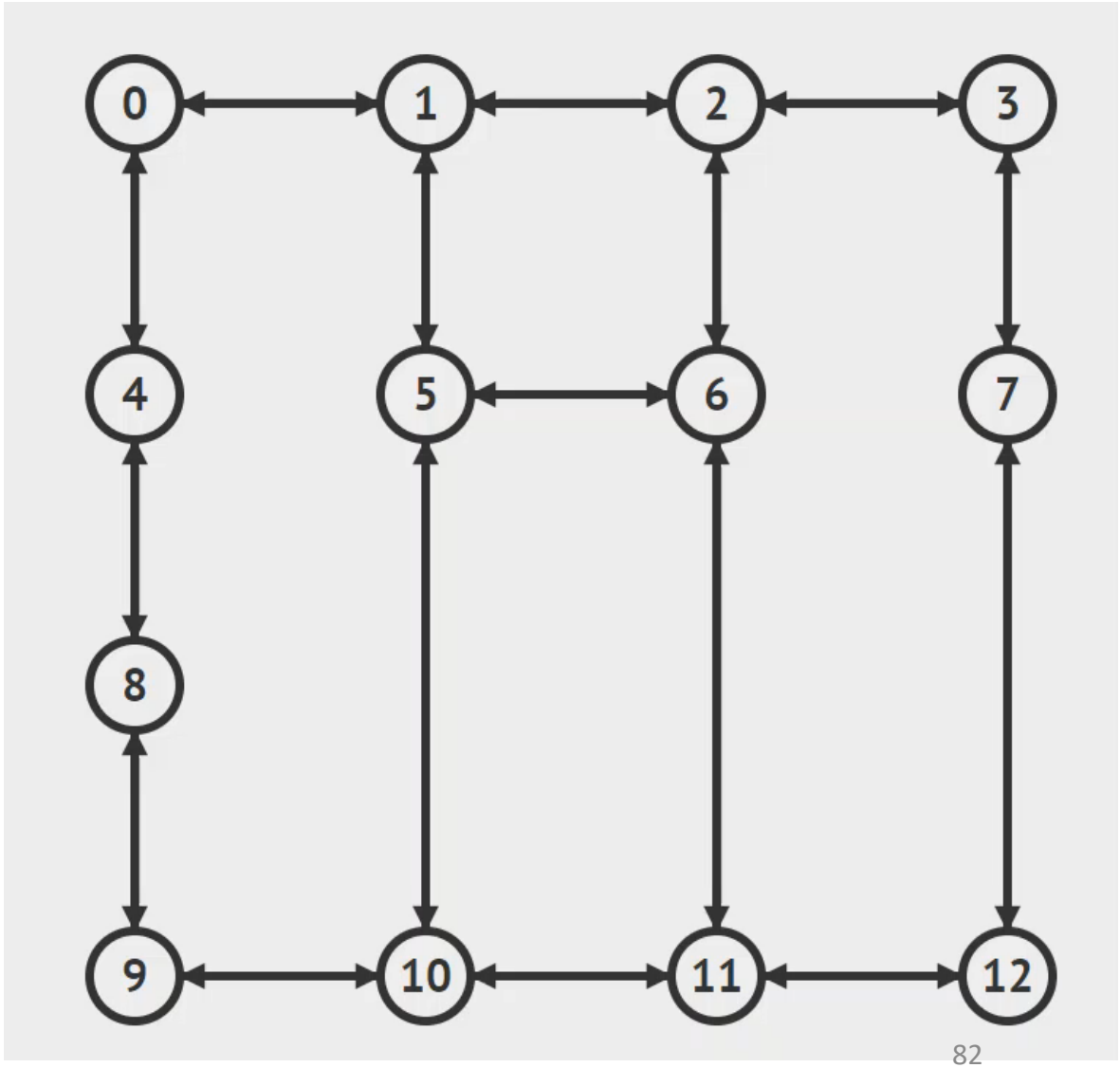
There’s no unique traversal and it can be different based on the order of the successors.

Breadth-first Search





Breadth-first Search



Breadth-first Search

Crawlers in Search Engines

Breadth-First Search is one of the main algorithms used for indexing web pages.

The algorithm starts traversing from the source page and follows all the links associated with the page.

Each web page is a node in a graph.

Breadth-first Search

GPS Navigation systems

Breadth-First Search is one of the best algorithms used to find neighboring locations by using the GPS.

Find the Shortest Path for an unweighted graph:

When it comes to an unweighted graph, calculating the shortest path is quite simple since the idea behind the shortest path is to choose a path with the least number of edges. Breadth-First Search can allow this by traversing a minimum number of nodes starting from the source node.

Breadth-first Search

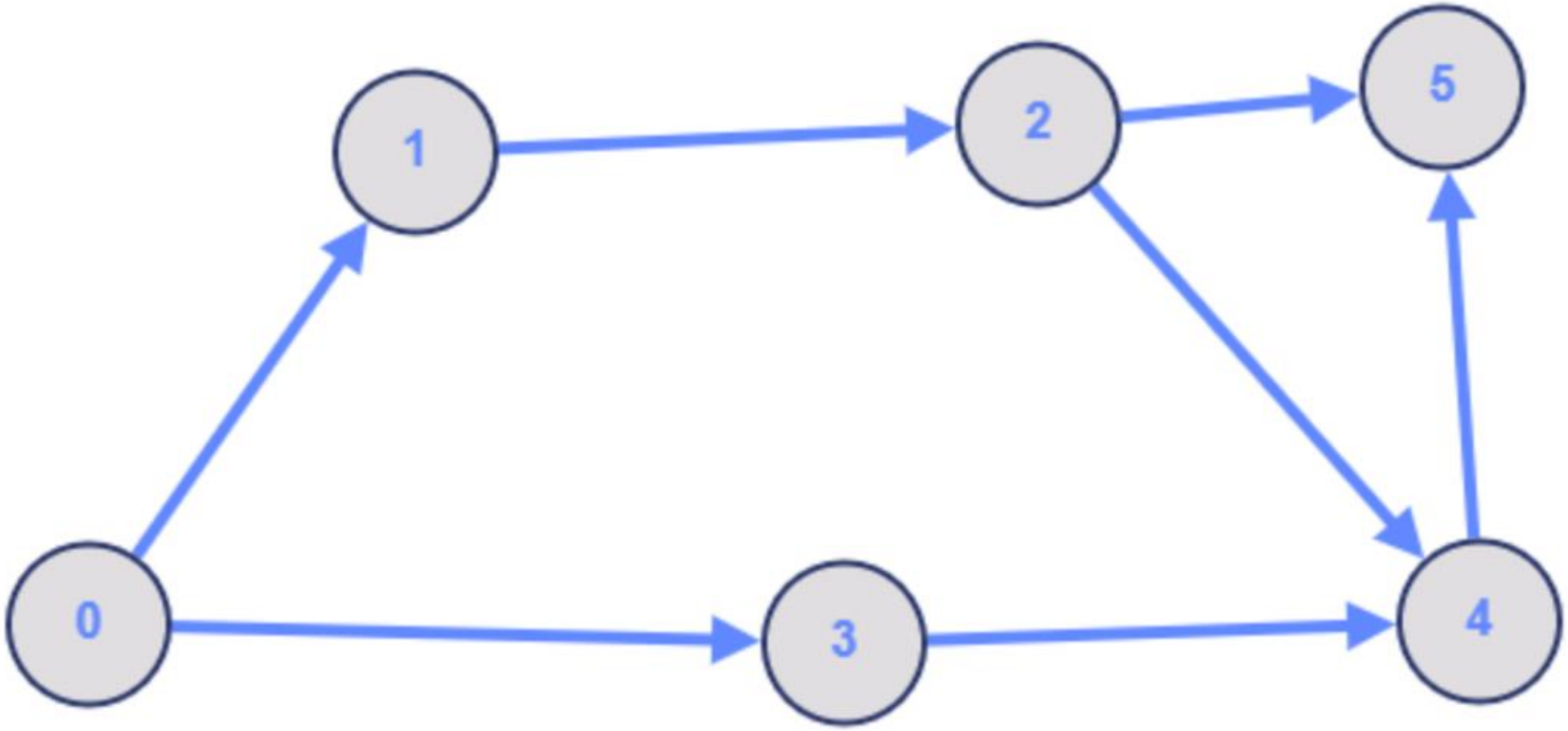
Broadcasting

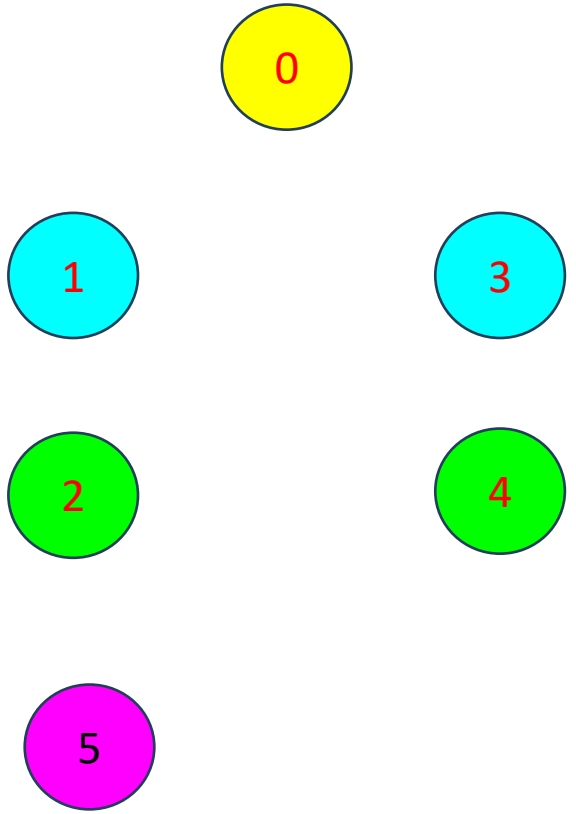
Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. One of the most commonly used traversal methods is Breadth-First Search. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

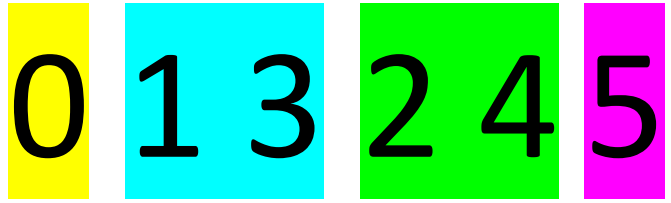
Peer to Peer Networking

Breadth-First Search can be used as a traversal method to find all the neighboring nodes in a Peer to Peer Network. For example, BitTorrent uses Breadth-First Search for peer to peer communication.

Breadth-first Search



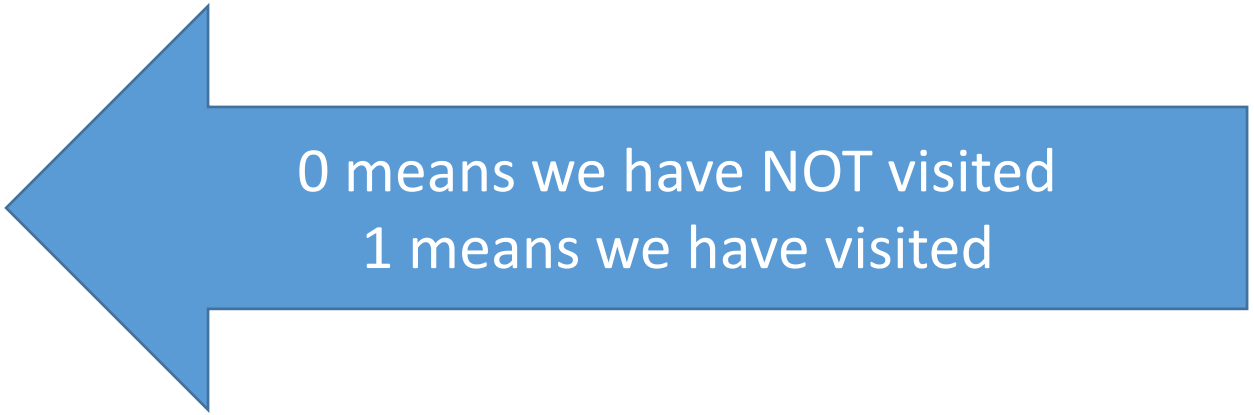




Breadth-first Search

We need a structure to hold our vertex information. So far, we have known a vertex by a name or label and we need to know if we have visited it.

```
typedef struct
{
    char label;
    int visited;
}
Vertex;
```



Breadth-first Search

We need two pieces of information to do our BFS

1. We need to be able to determine neighbors

   Adjacency Matrix

2. We need to keep track of which vertices have been visited

   Array that can hold vertices

Breadth-first Search

Adjacency Matrix

```
int AdjMatrix[MAX][MAX];
```

Array to hold vertices

```
Vertex *VertexArray[MAX];
```

Note that this is an array of **pointers** to `Vertex`. We will be dynamically allocating each vertex as we need it and storing the pointer to its memory in the array.

Breadth-first Search

We need to initialize the Adjacency Matrix first.

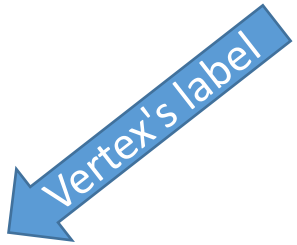
All elements of the 2D array need to be set to 0 to indicate that edges are not neighbors.

```
for(i = 0; i < MAX; i++)
{
    for(j = 0; j < MAX; j++)
    {
        AdjMatrix[i][j] = 0;
    }
}
```

```
int AdjMatrix[MAX][MAX] = {};
```

Breadth-first Search

Now we need to add the vertices to the `VertexArray`



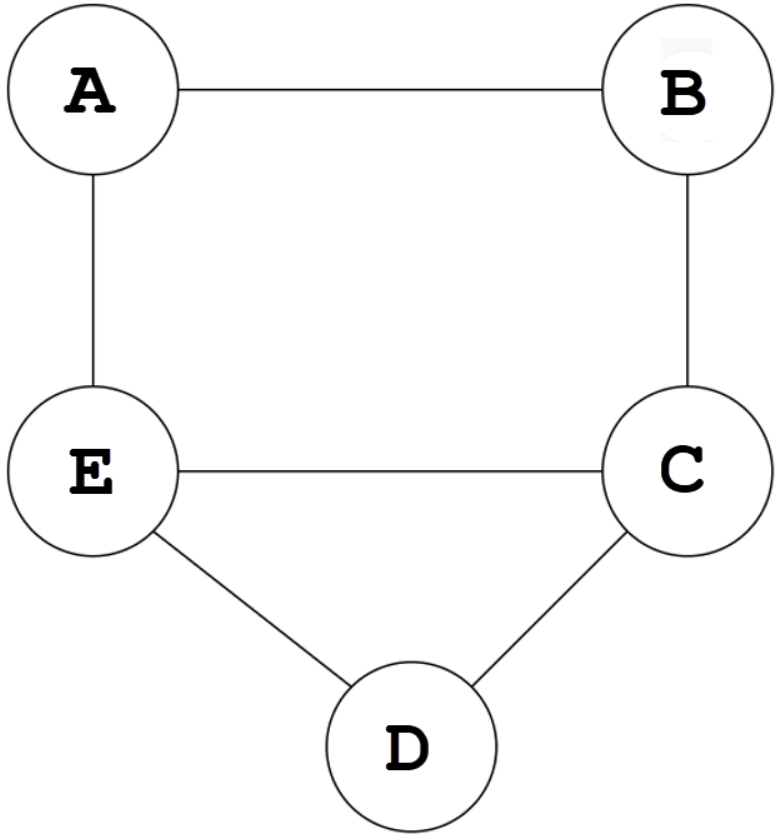

```
addVertex('A', VertexArray, &VertexCount);
```

```
void addVertex(char label, Vertex *VertexArray[], int *VertexCount)
{
      Vertex *NewVertex = malloc(sizeof(Vertex));
      NewVertex->label = label;
      NewVertex->visited = 0;
      VertexArray[(*VertexCount)++] = NewVertex;
}
```

Breadth-first Search

Now we call `addVertex()` for every vertex in our graph.

```
addVertex('A', VertexArray, &VertexCount);   // 0
addVertex('B', VertexArray, &VertexCount);   // 1
addVertex('C', VertexArray, &VertexCount);   // 2
addVertex('D', VertexArray, &VertexCount);   // 3
addVertex('E', VertexArray, &VertexCount);   // 4
```




```
(gdb) p *VertexArray[0]
$5 = {label = 65 'A', visited = 0}
(gdb) p *VertexArray[1]
$6 = {label = 66 'B', visited = 0}
(gdb) p *VertexArray[2]
$7 = {label = 67 'C', visited = 0}
(gdb) p *VertexArray[3]
$8 = {label = 68 'D', visited = 0}
(gdb) p *VertexArray[4]
$9 = {label = 69 'E', visited = 0}
```

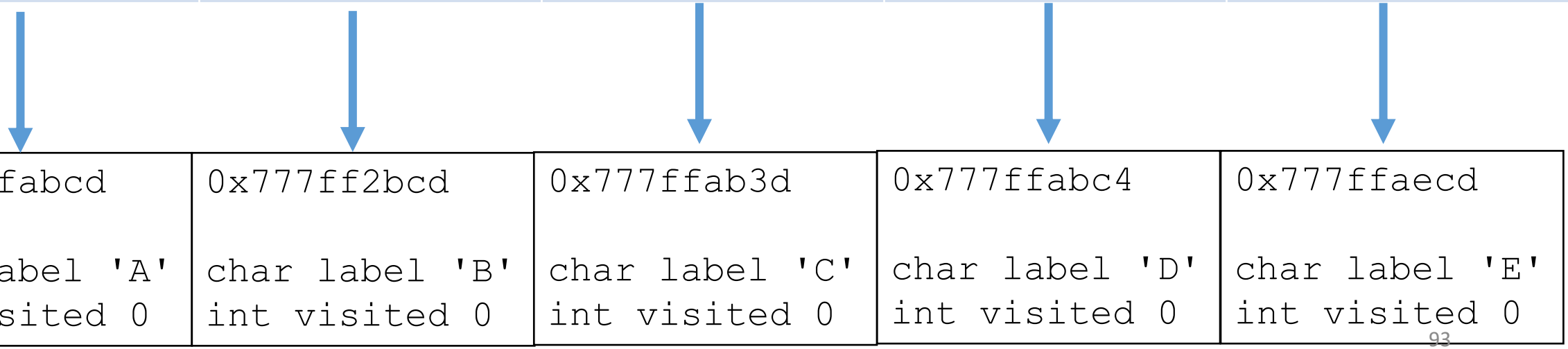
	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0

Breadth-first Search

Remember that `VertexArray` is an array of pointers to structs of type `Vertex`.

	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0

0	1	2	3	4
0x777ffabcd	0x777ff2bcd	0x777ffab3d	0x777ffabc4	0x777ffaecd



Breadth-first Search

Now we need to add edges to the Adjacency Matrix.

```
addEdge(0, 1, AdjMatrix);

void addEdge(int start, int end, int AdjMatrix[][MAX])
{
     AdjMatrix[start][end] = 1;
     #ifdef UNDIRECTED
     AdjMatrix[end][start] = 1;
     #endif
}
```

Breadth-first Search

Now we call `addEdge()` for every edge in our graph.

This is an undirected graph so we would compile our program as

```
gcc BFSG.c –D UNDIRECTED
```

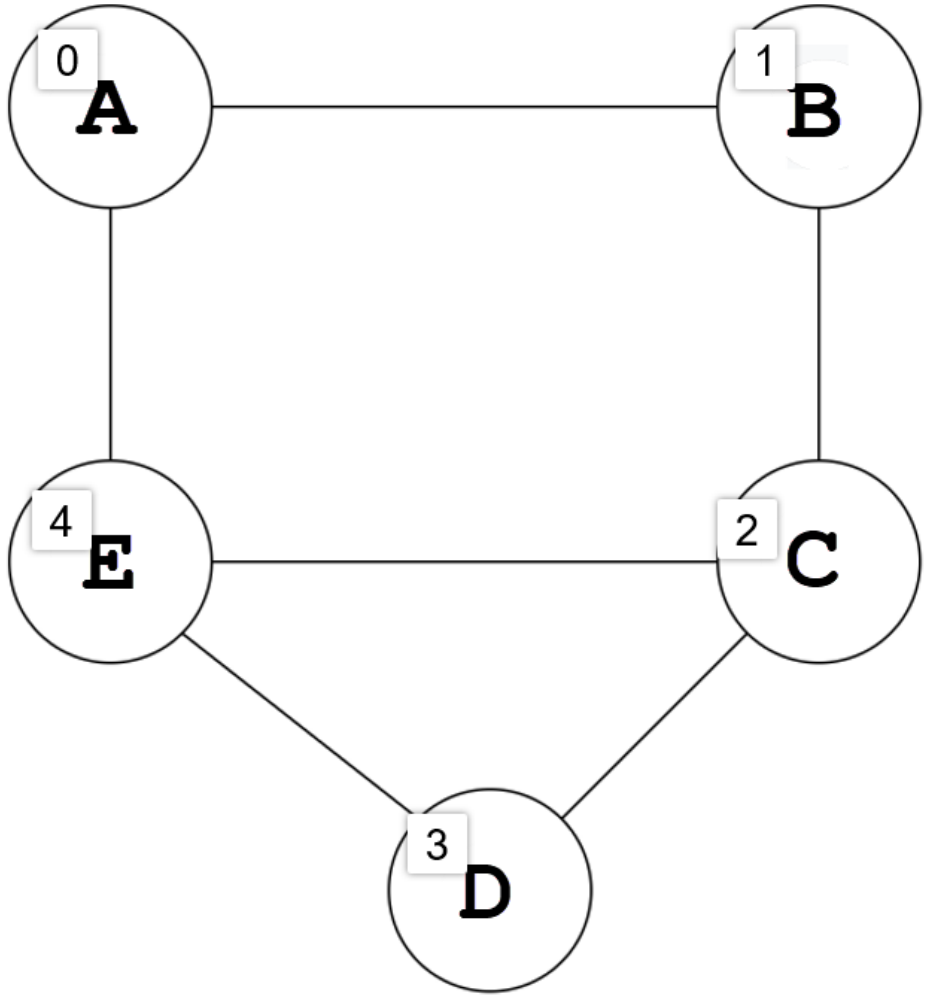
so that this code

```
AdjMatrix[start][end] = 1;
#ifdef UNDIRECTED
AdjMatrix[end][start] = 1;
#endif
```

would look like this

```
AdjMatrix[start][end] = 1;
AdjMatrix[end][start] = 1;
```

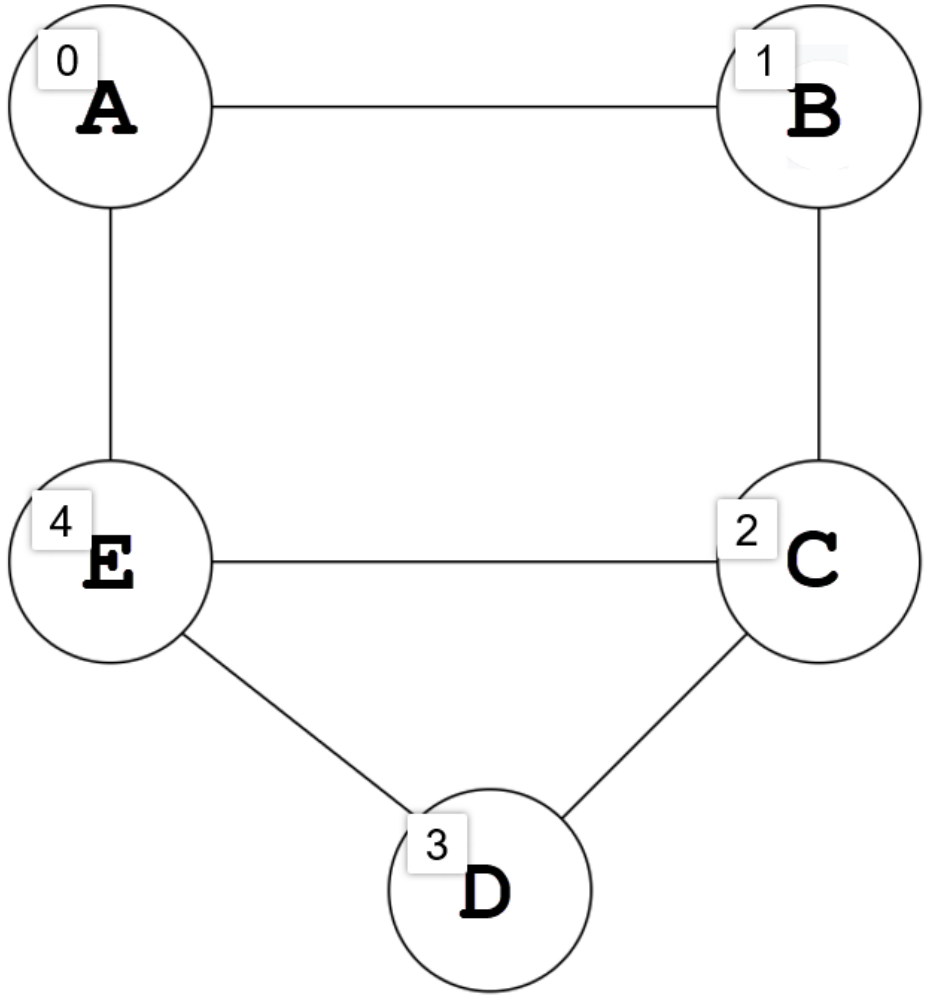
to the compiler.



Breadth-first Search

Now we call `addEdge()` for every edge in our graph.

What are the edges in this graph based on the array indices assigned to each vertex?



0,1
1,2
2,4
4,0
2,3
3,4

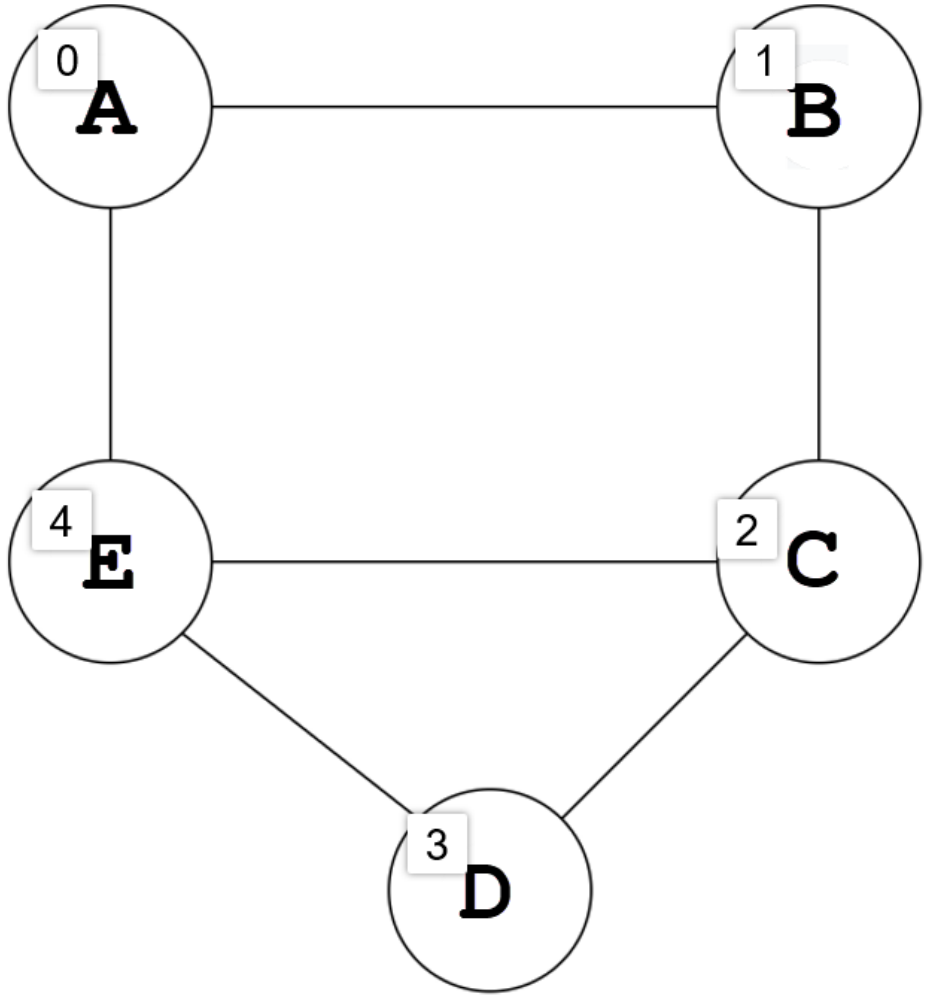
Do we need the opposite pairs?
1,0 or 2,1 or 4,2 or 0,4 or 3,2 or 4,3

```
addEdge(0, 1, AdjMatrix);
addEdge(1, 2, AdjMatrix);
addEdge(2, 4, AdjMatrix);
addEdge(4, 0, AdjMatrix);
addEdge(2, 3, AdjMatrix);
addEdge(3, 4, AdjMatrix);
```

Breadth-first Search

```
addEdge(0, 1, AdjMatrix);
addEdge(1, 2, AdjMatrix);
addEdge(2, 4, AdjMatrix);
addEdge(4, 0, AdjMatrix);
addEdge(2, 3, AdjMatrix);
addEdge(3, 4, AdjMatrix);
```

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0



Breadth-first Search

We have our vertex array and adjacency matrix set up. Now we can call the Breadth First Search function.

```
BreadthFirstSearch(VertexArray, VertexCount, AdjMatrix);
```

We need to create a few items now…

```
int queue[MAX] = {};
int head = -1;
int tail = -1;
int queueItemCount = 0;
int CurrentVertexIndex = 0;
```

We are using an array for our queue. It is fast (less overhead from adding and deleting vertices) and is sufficient for what we are doing.

Breadth-first Search

Our traversal will start at Vertex A

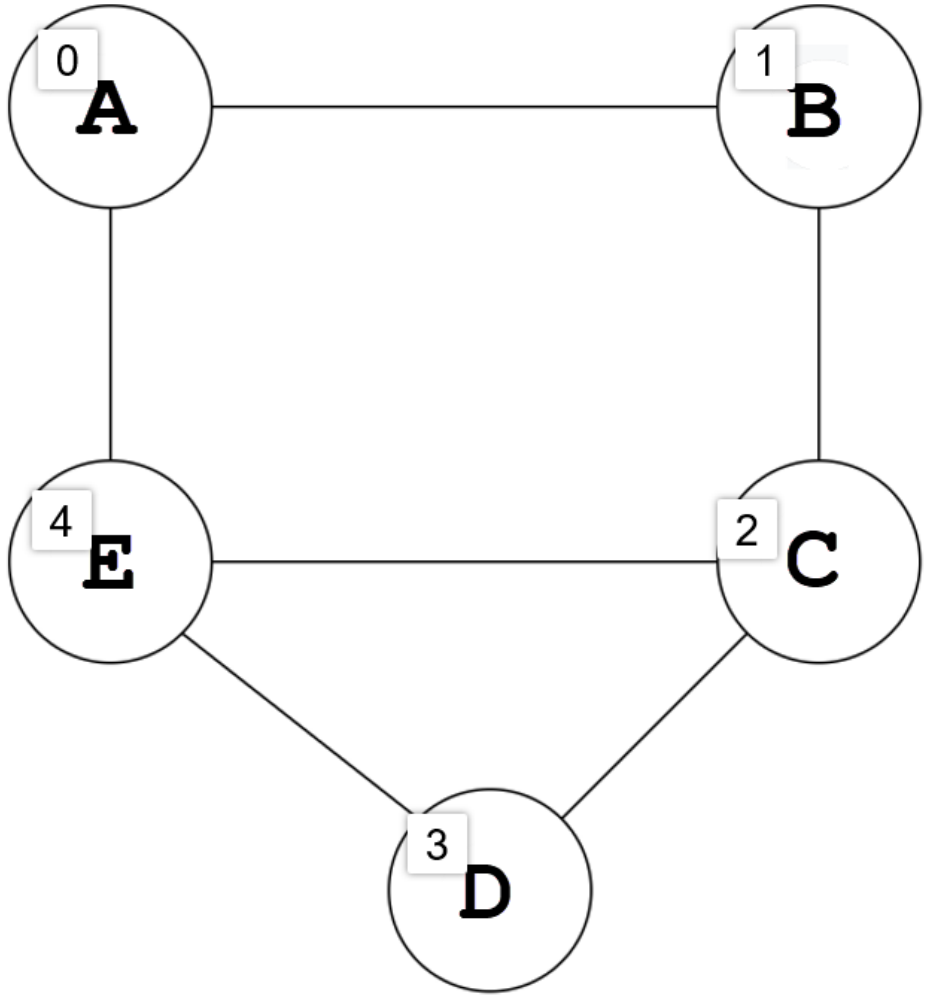
Why?

Because we said so…

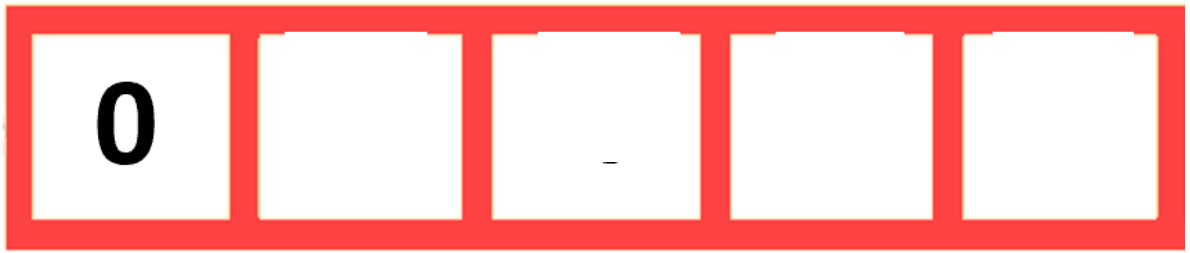
When we set up the adjacency matrix and the vertex array, we decided to put Vertex A at index 0 and we are starting with index 0.

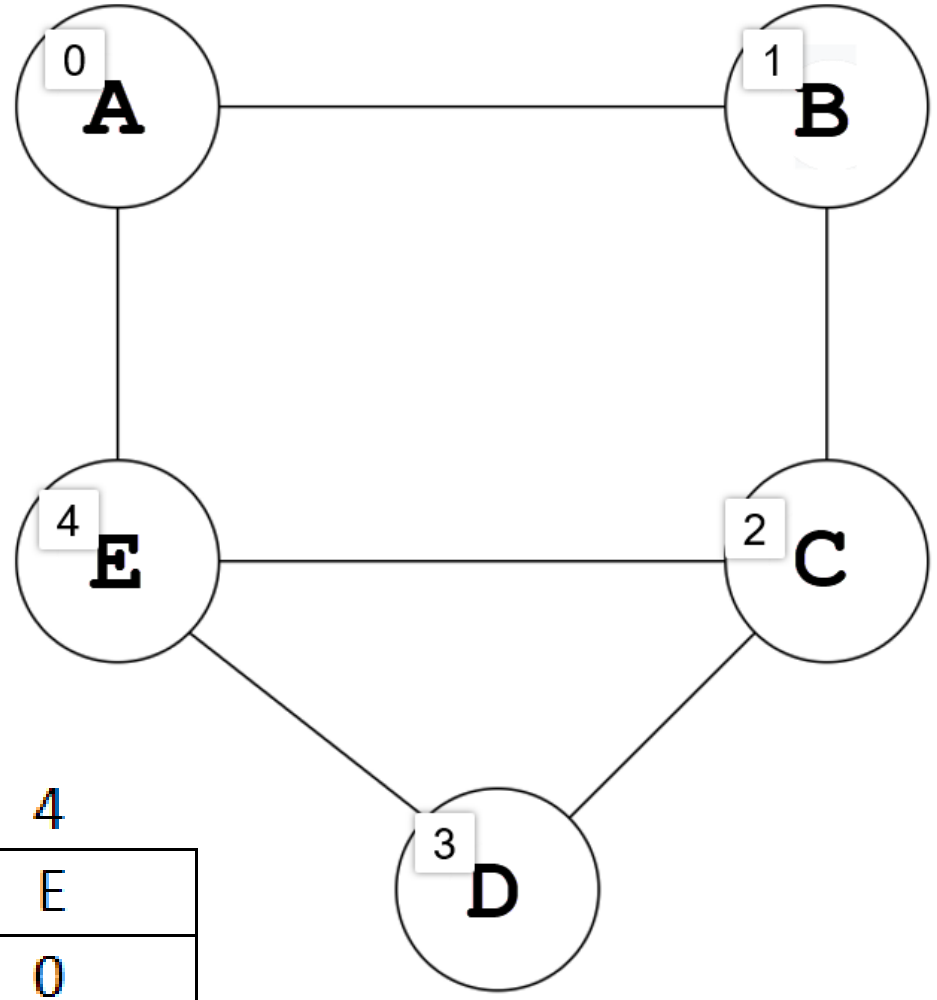
The starting vertex could be altered as needed.

Starting at index 0 is not REQUIRED.



Breadth-first Search





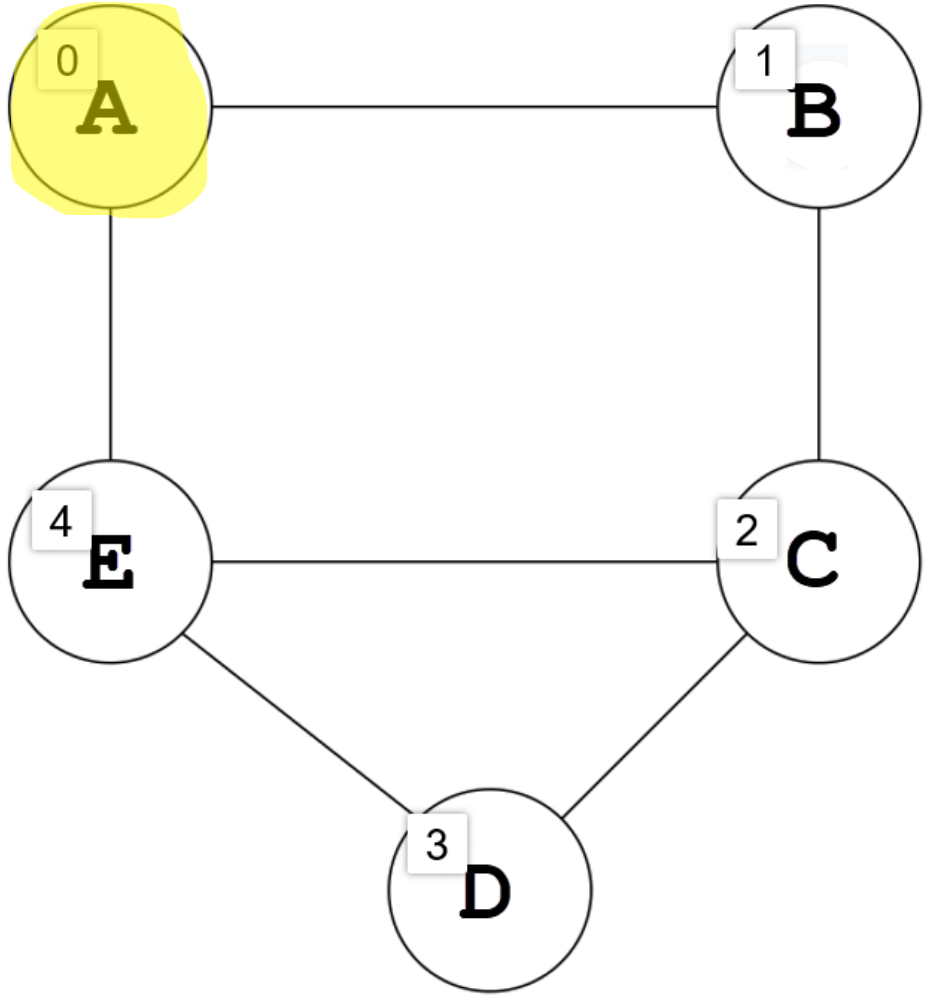
	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	0	0	0	0

Breadth-first Search

Our traversal will start at Vertex A

```
VertexArray[0]->visited = 1;
```

	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	0	0	0	0



Remember that `VertexArray` is an array of pointers to structs of type `Vertex`.

We are marking our starting vertex as having been visited.

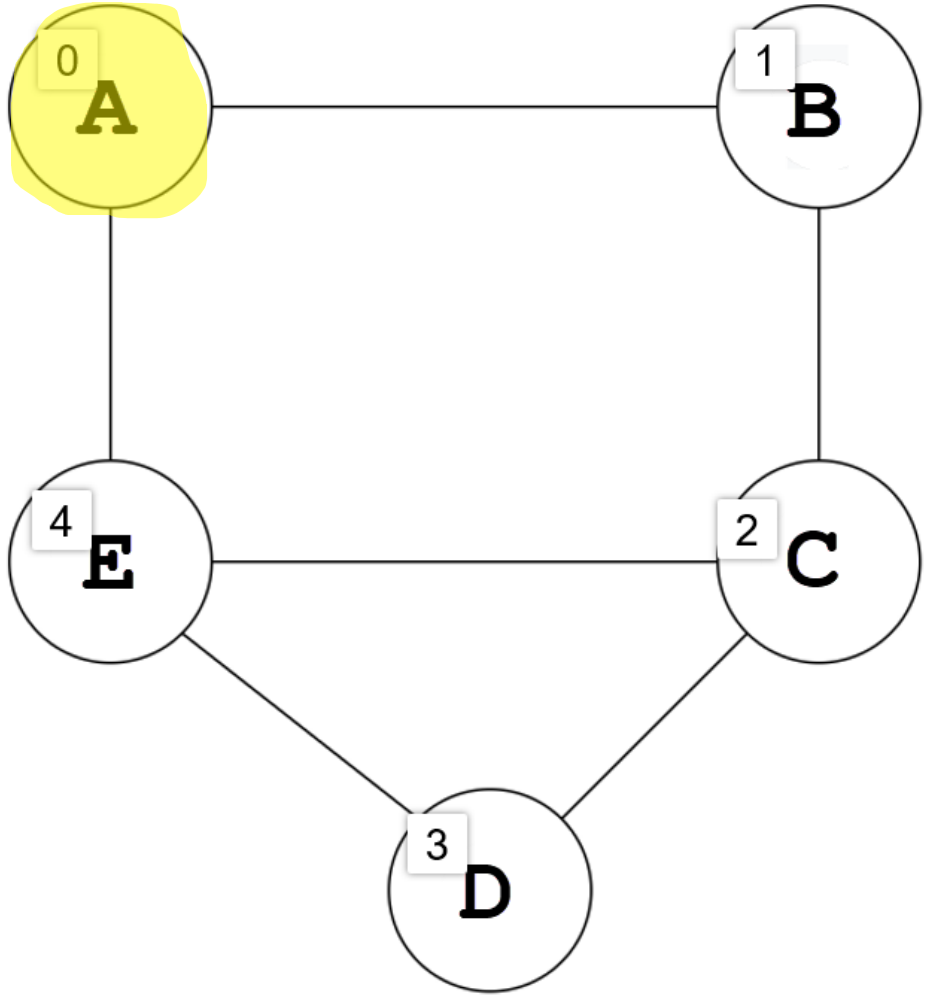
Breadth-first Search

Now that we have visited Vertex A, we add it to our queue.

```
enqueue(queue, &head, &tail, 0);
queueItemCount++;
```

`head` and `tail` are both -1 (they were initialized to -1) and we are putting Vertex A's index of 0 into our queue.

We increment `queueItemCount` to track how many items are active in our queue.

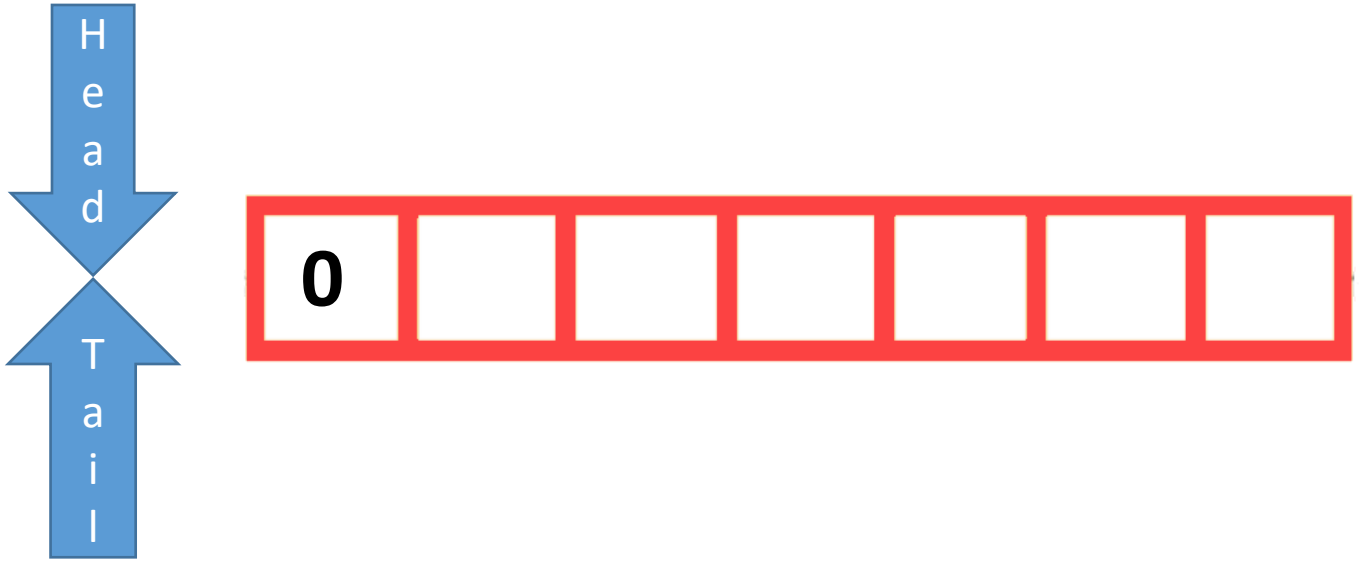


Breadth-first Search

```
37  void enqueue(int QueueArray[], int *head, int *tail, int value)
38  {
39
40      if (*tail == MAX - 1)
41          printf("Queue Overflow \n");
42      else
43      {
44          if (*head == -1)  /*If queue is initially empty */
45              *head = 0;
46          (*tail)++;
47          QueueArray[*tail] = value;
48      }
49  }
```

Breadth-first Search

```
if (*tail == MAX - 1)
   printf("Queue Overflow \n");
else
{

   if (*head == -1)
      *head = 0;
   (*tail)++;
   QueueArray[*tail] = value;
}
```

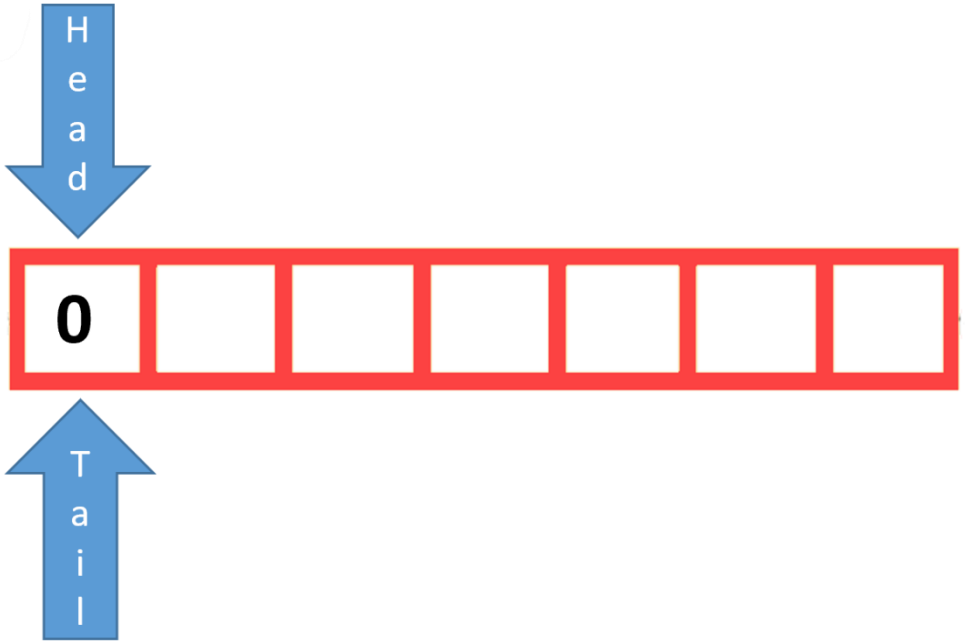


Breadth-first Search

```
95      while(queueItemCount)
96      {
97          CurrentVertexIndex = dequeue(queue, &head, &tail);
98
99          queueItemCount--;
100
101         for (i = 0; i < VertexCount; i++)
102         {
103             if (AdjMatrix[CurrentVertexIndex][i] == 1)
104             {
105                 if (VertexArray[i]->visited == 0)
106                 {
107                     enqueue(queue, &head, &tail, i);
108                     queueItemCount++;
109                     VertexArray[i]->visited = 1;
110                 }
111             }
112         }
113     }
```

Breadth-first Search

```
while(queueItemCount)
{
      CurrentVertexIndex = dequeue(queue, &head, &tail);
      queueItemCount--;
```

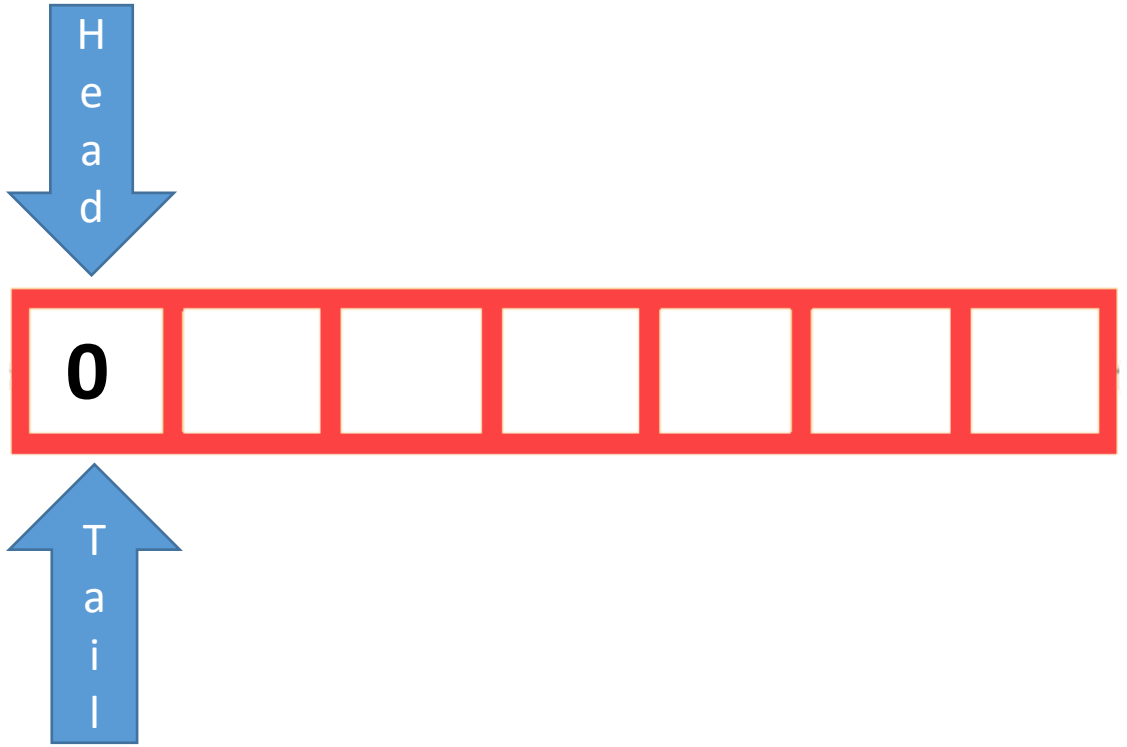


While there are unused items in the queue, dequeue the head and place that value in `CurrentVertexIndex` and decrement the number of items in the queue.

This is the vertex we will use to check for adjacent/neighbor vertices.

Breadth-first Search

```
int dequeue(int QueueArray[], int *head, int *tail)
{
   int IndexAtHead = 0;

   if (*head == -1 || *head > *tail)
   {
      printf("\n\nQueue is empty\n\n");
      IndexAtHead = -1;
   }
   else
   {
      IndexAtHead = QueueArray[*head];
      (*head)++;
   }

   return IndexAtHead;
}
```

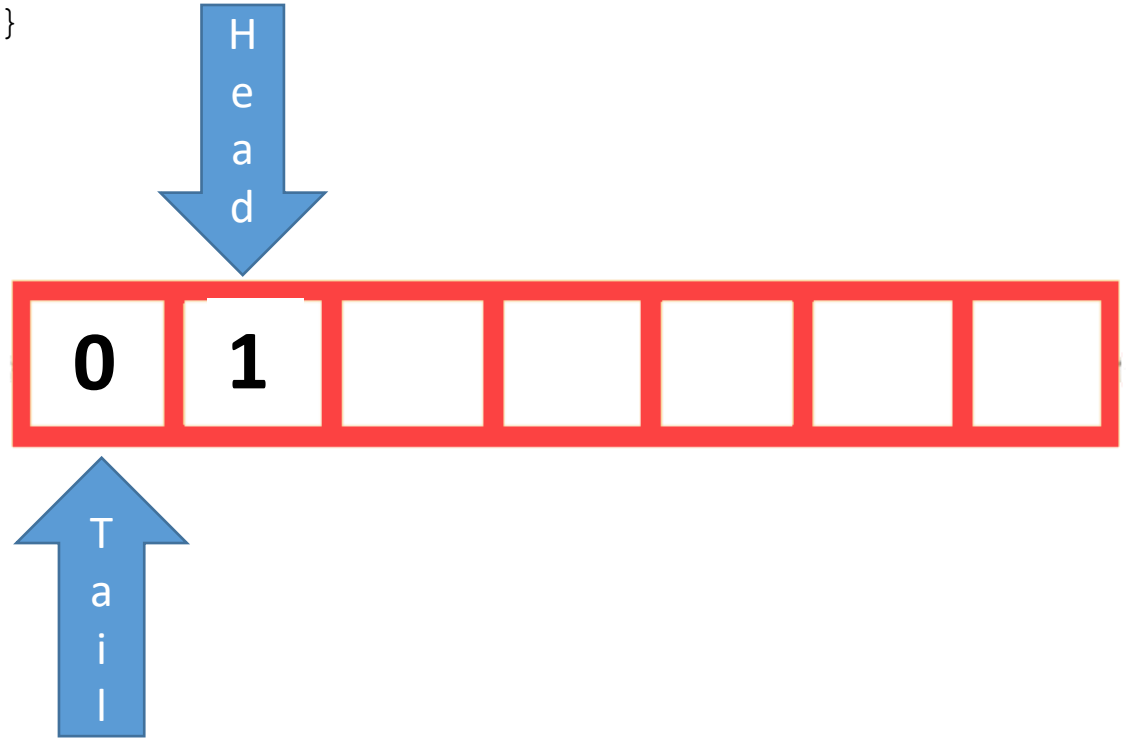


Breadth-first Search

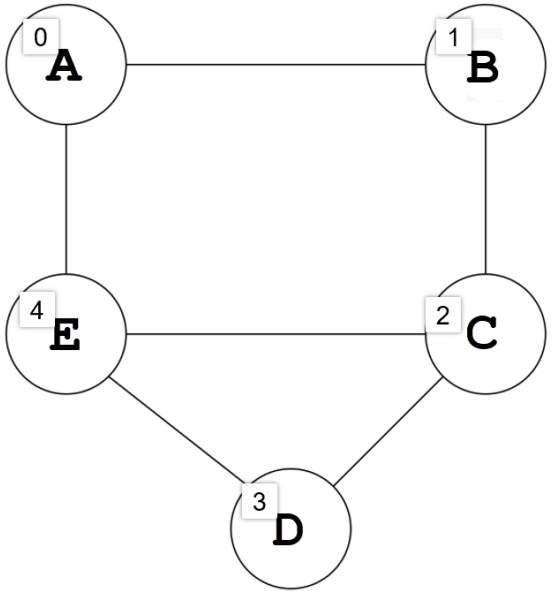
```
for (i = 0; i < VertexCount; i++)
{
   if (AdjMatrix[CurrentVertexIndex][i] == 1)
   {
      if (VertexArray[i]->visited == 0)
      {
         enqueue(queue, &head, &tail, i);
         queueItemCount++;
         VertexArray[i]->visited = 1;
      }
   }
}
```

Breadth-first Search

```
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}
```



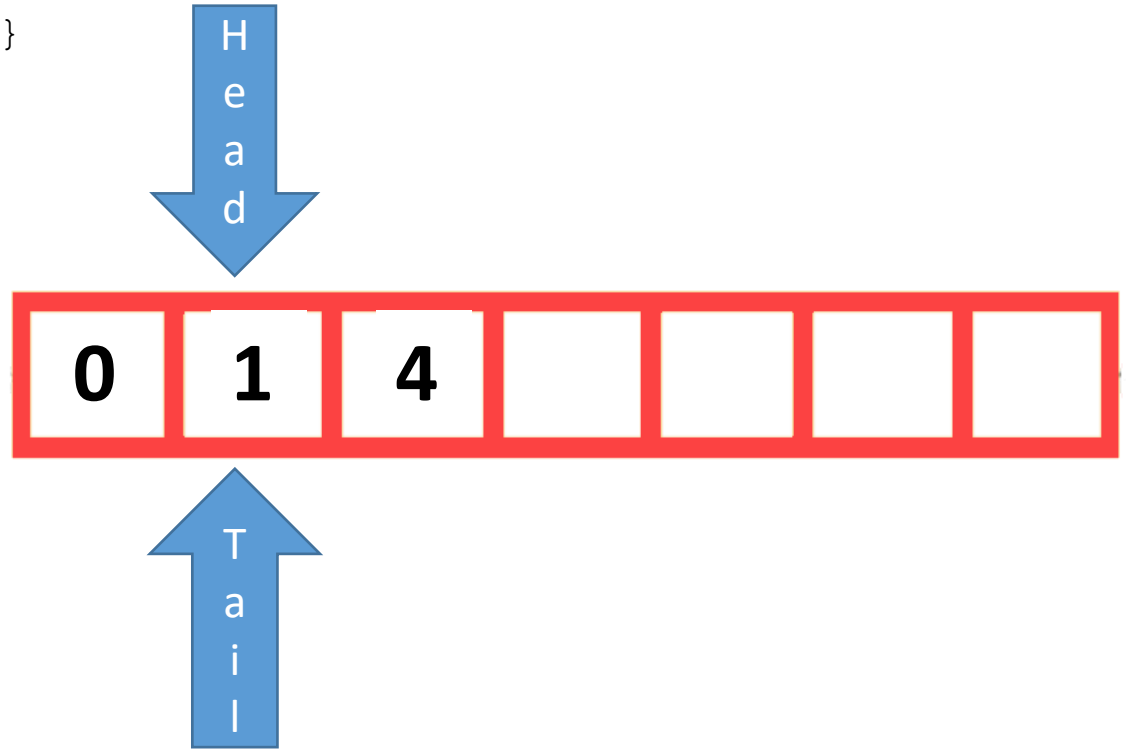
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0



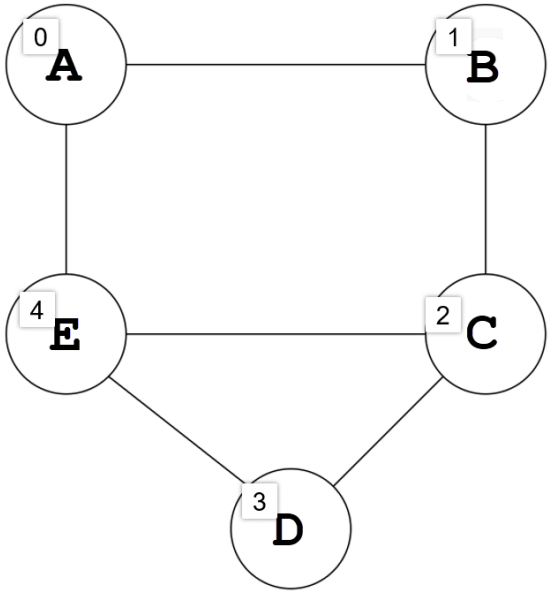
	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	0	0	0	0

Breadth-first Search

```
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}
```



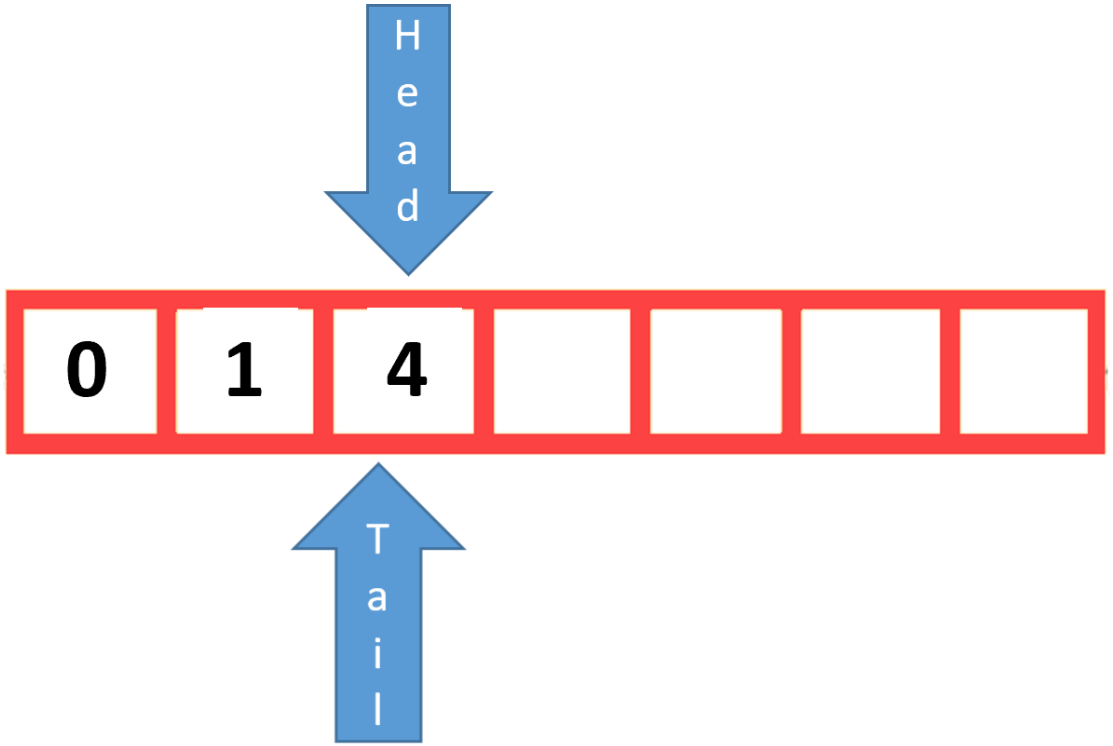
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0



	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	0	0	0

Breadth-first Search

```
while(queueItemCount)
{
     CurrentVertexIndex = dequeue(queue, &head, &tail);
     queueItemCount--;
```

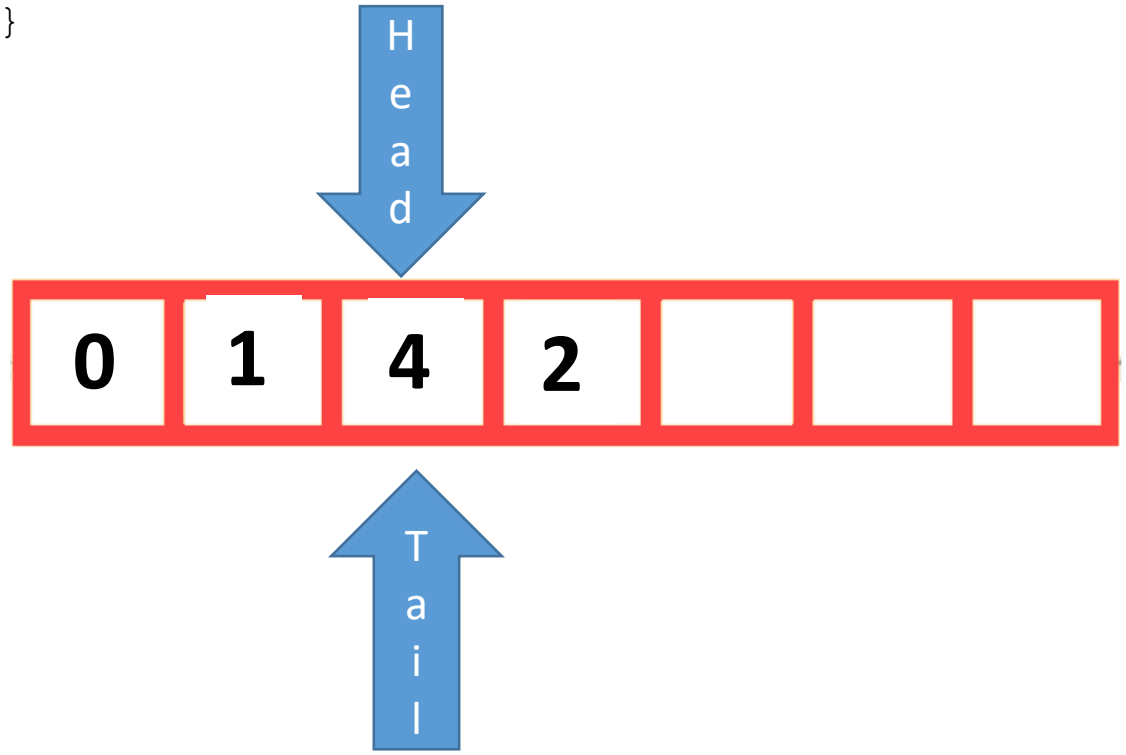


While there are unused items in the queue, dequeue the head and place that value in `CurrentVertexIndex` and decrement the number of items in the queue.

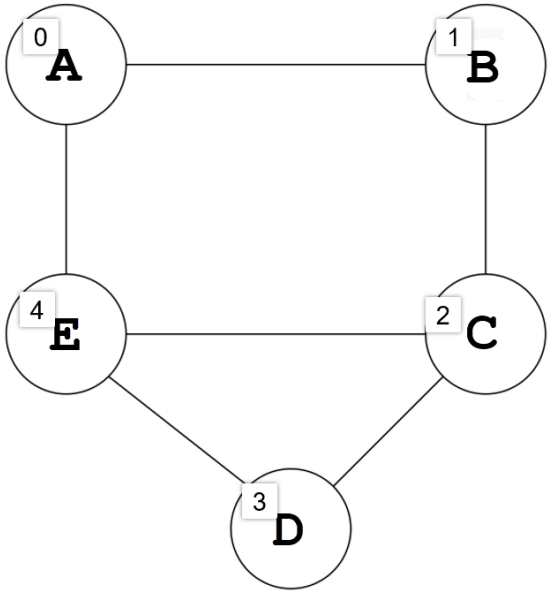
We are now going to check for neighbors of the vertex at index 1 (Vertex B).

Breadth-first Search

```
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}
```



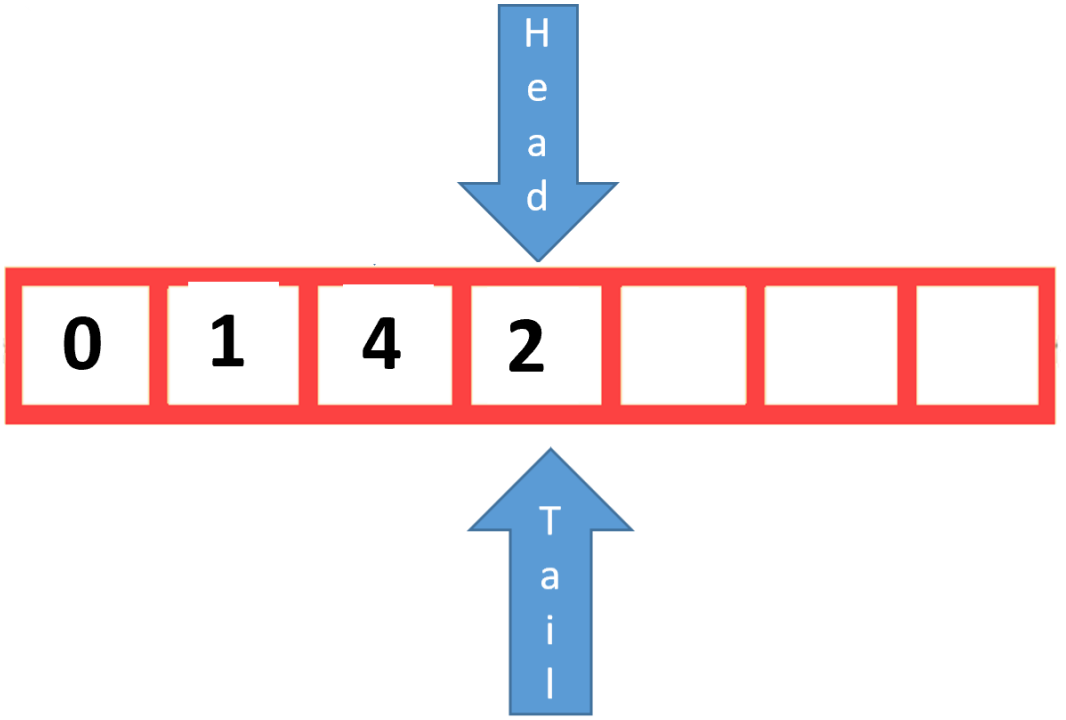
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0



	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	0	0	1

Breadth-first Search

```
while(queueItemCount)
{
      CurrentVertexIndex = dequeue(queue, &head, &tail);
      queueItemCount--;
```

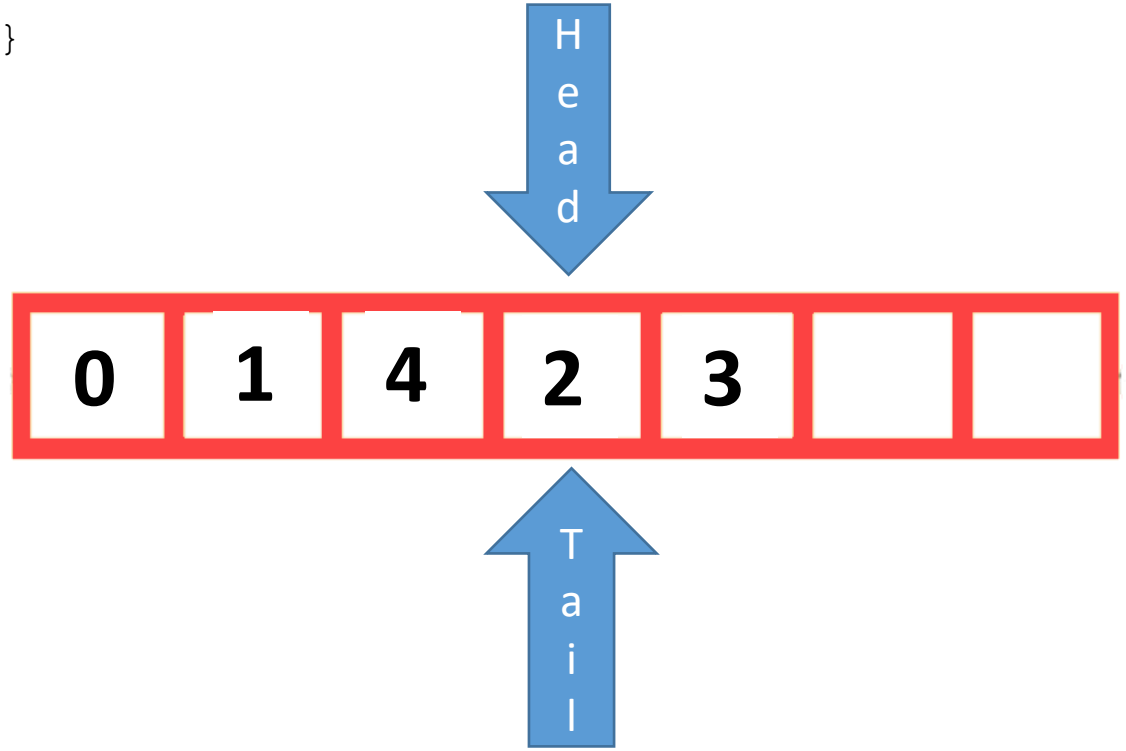


While there are unused items in the queue, dequeue the head and place that value in `CurrentVertexIndex` and decrement the number of items in the queue.

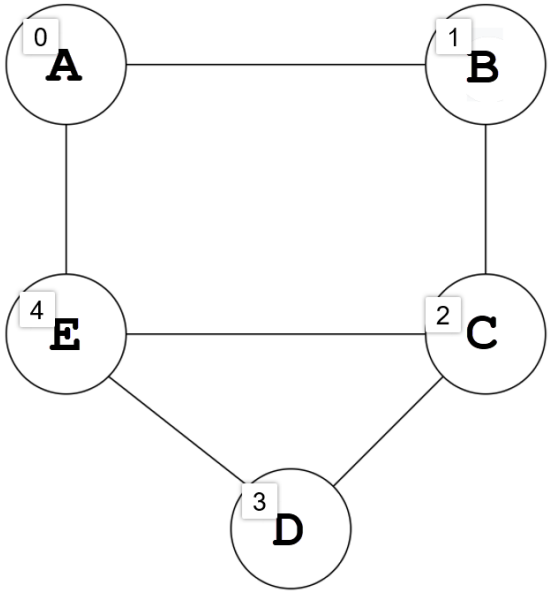
We are now going to check for neighbors of the vertex at index 4 (Vertex E).

Breadth-first Search

```
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}
```



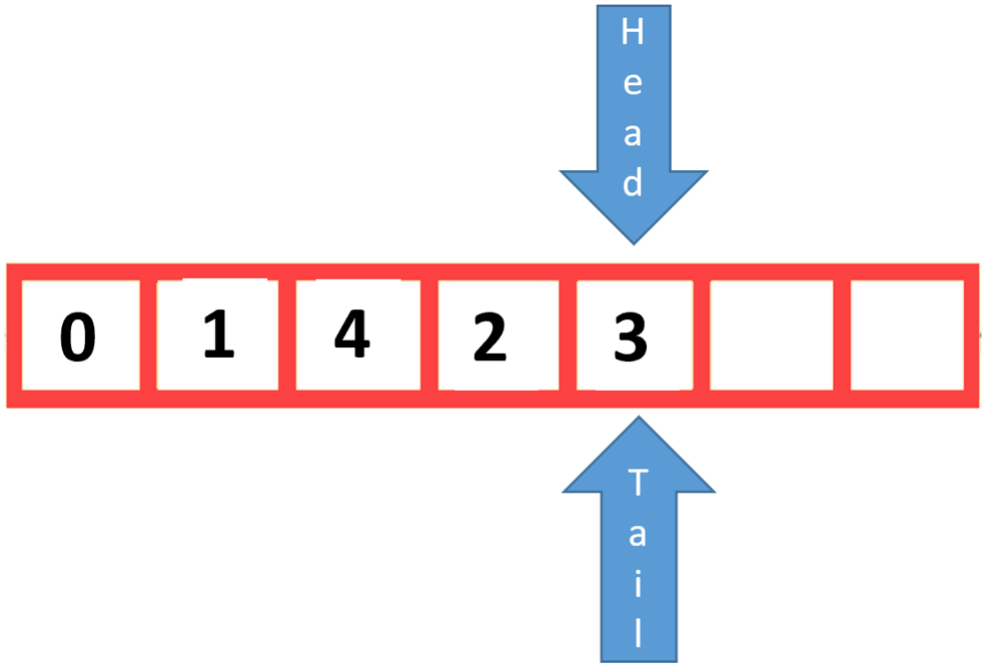
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0



	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	0	1

Breadth-first Search

```
while(queueItemCount)
{
     CurrentVertexIndex = dequeue(queue, &head, &tail);
     queueItemCount--;
```

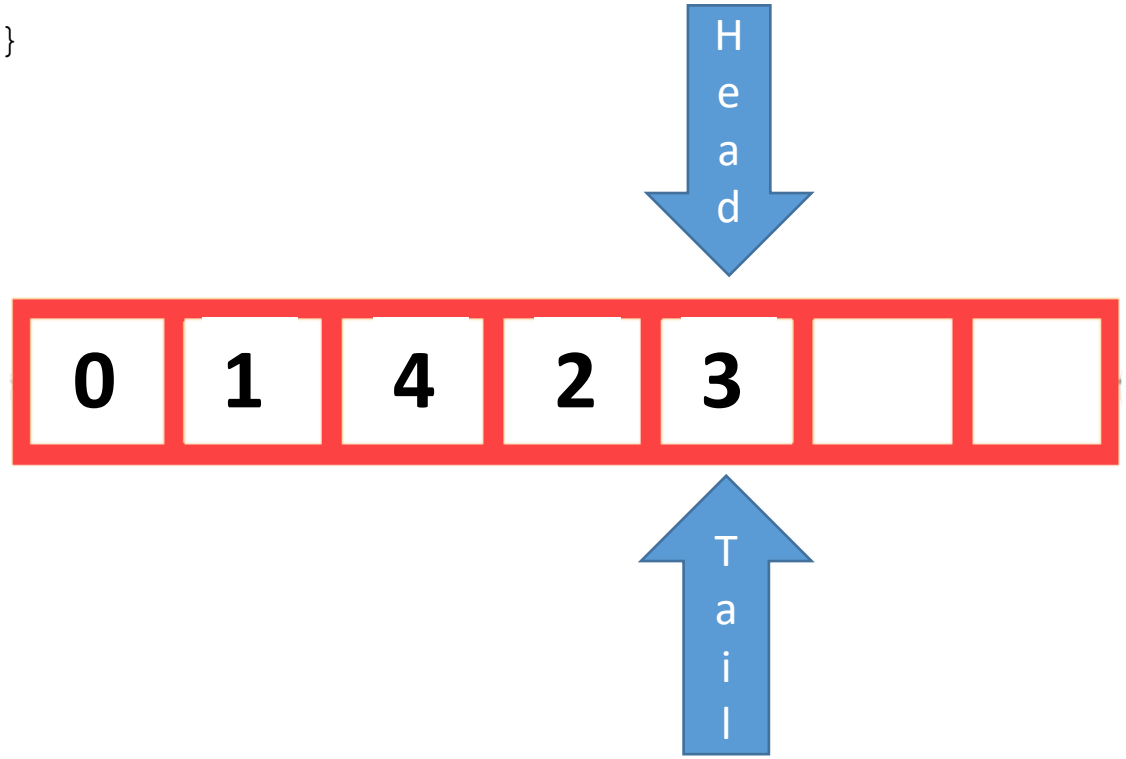


While there are unused items in the queue, dequeue the head and place that value in `CurrentVertexIndex` and decrement the number of items in the queue.

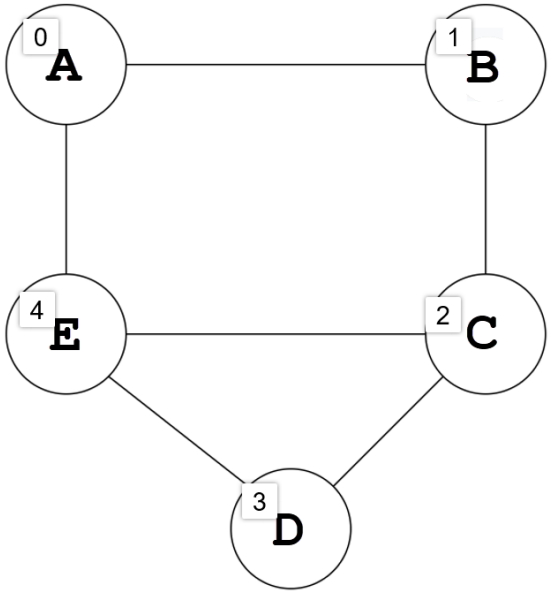
We are now going to check for neighbors of the vertex at index 2 (Vertex C).

Breadth-first Search

```
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}
```



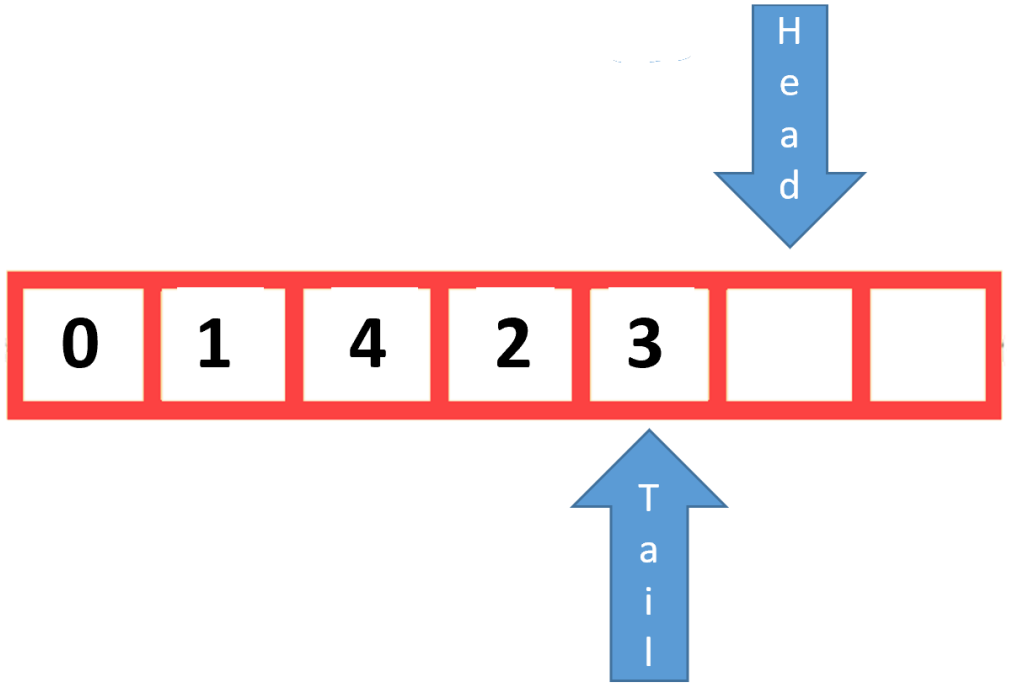
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0



	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	1	1

Breadth-first Search

```
while(queueItemCount)
{
      CurrentVertexIndex = dequeue(queue, &head, &tail);
      queueItemCount--;
```

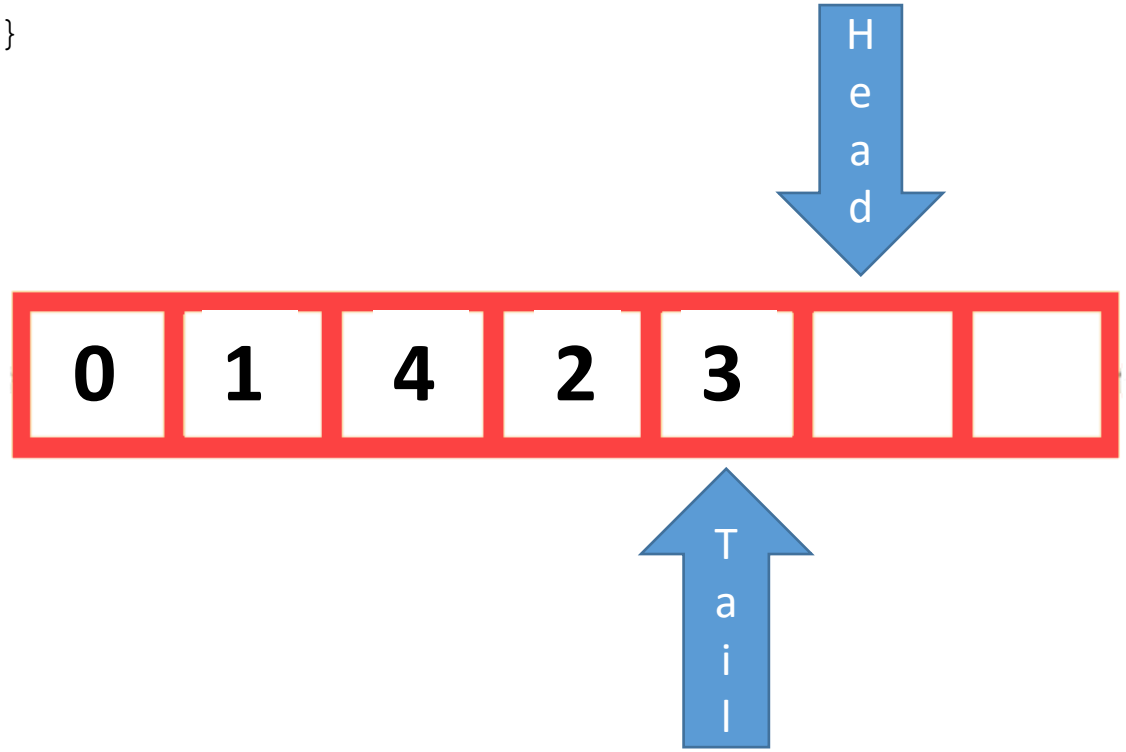


While there are unused items in the queue, dequeue the head and place that value in `CurrentVertexIndex` and decrement the number of items in the queue.

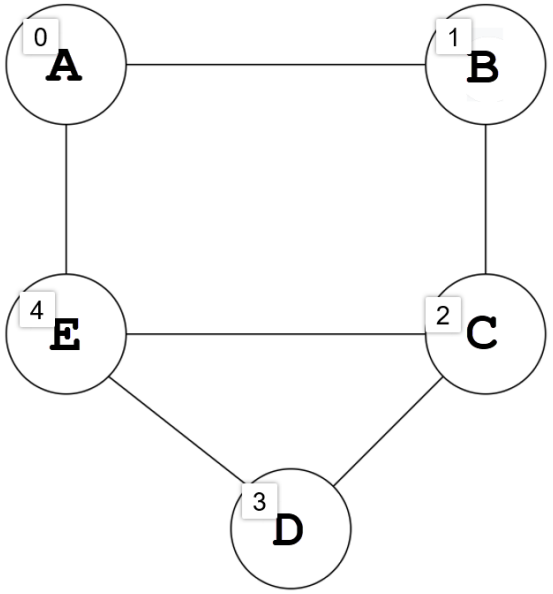
We are now going to check for neighbors of the vertex at index 3 (Vertex D).

Breadth-first Search

```
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}
```



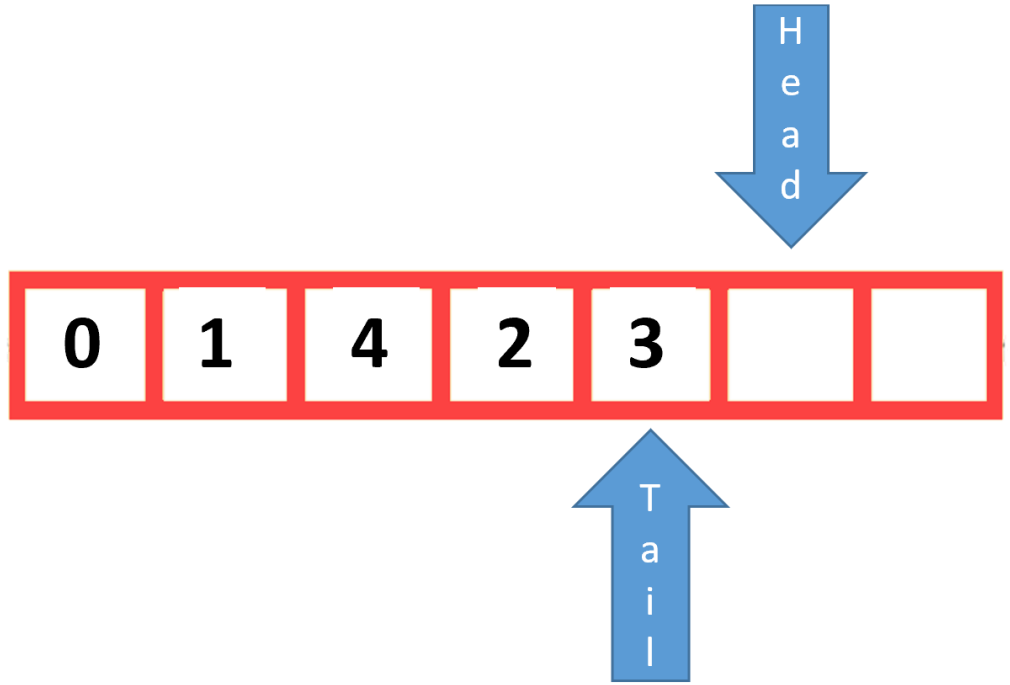
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0



	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	1	1

Breadth-first Search

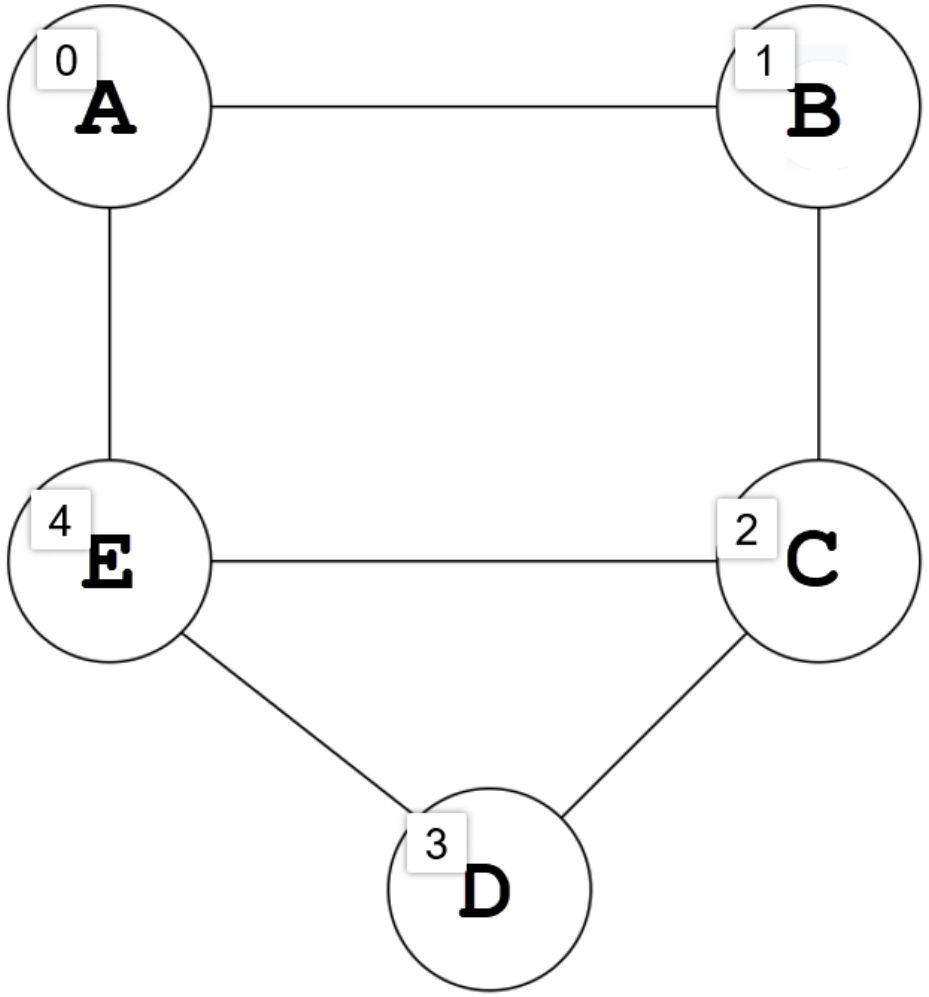
```
while(queueItemCount)
{
    CurrentVertexIndex = dequeue(queue, &head, &tail);
    queueItemCount--;
```



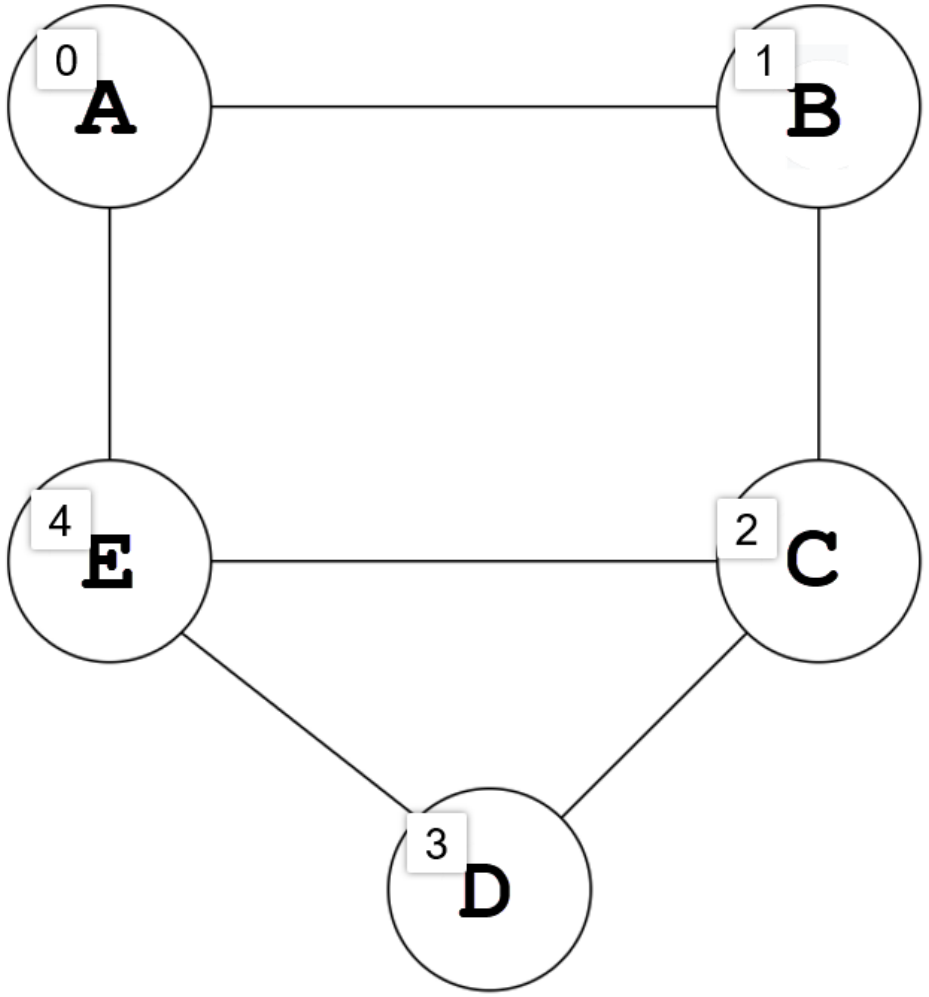
`queueItemCount` is now 0. The while loop will stop and the program will end.

We have traversed every vertex.

Breadth-first Search



Breadth-first Search



Breadth-first Search

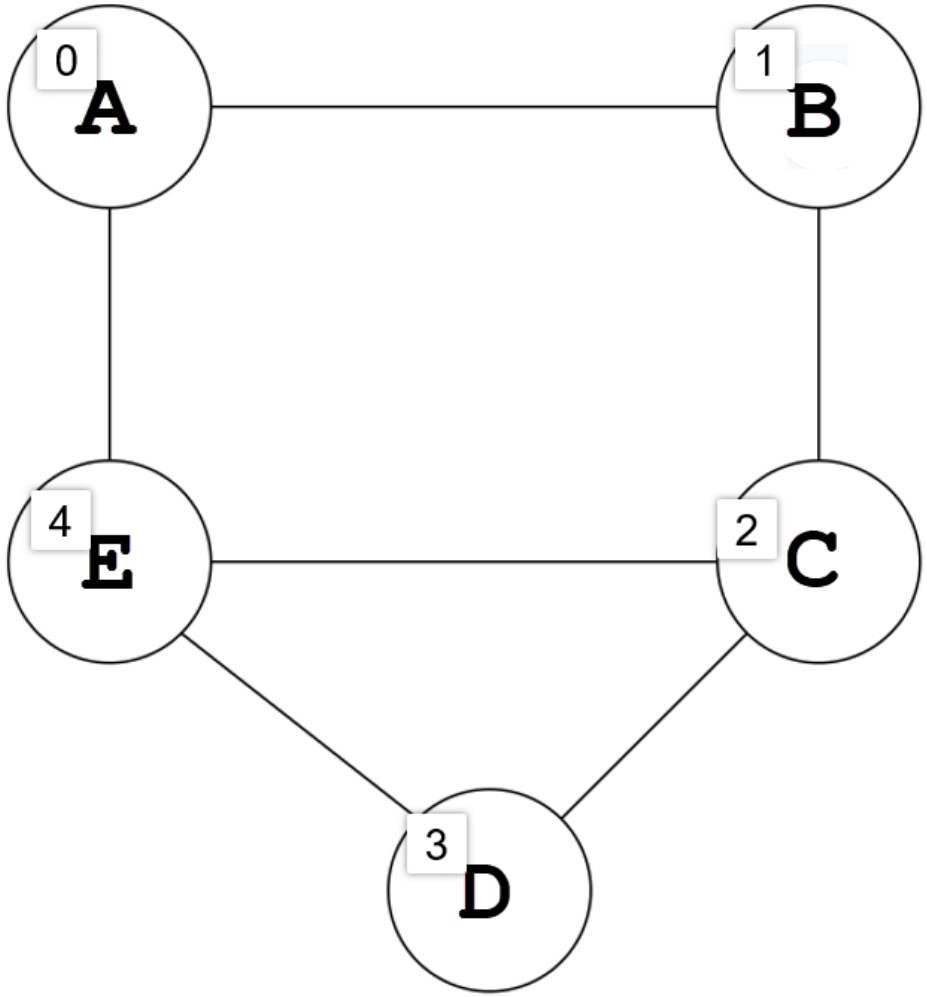
How long does breadth-first search take for a graph with vertex set *V* and edge set *E*?

It depends…..

Depends on what???

What data structure did the code use for figuring out neighbors…

Breadth-first Search



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0

0	1	4	
1	0	2	
2	1	3	4
3	2	4	
4	0	2	3

How many edges are in this graph?

6 or 12

Breadth-first Search

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	0	0
2	0	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0

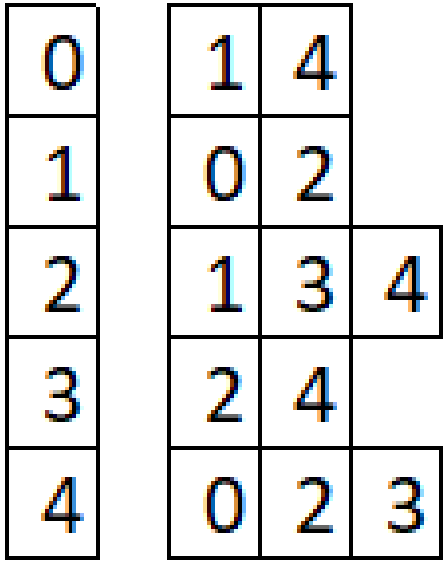
BFS using an adjacency matrix

Every vertex is visited once.

To find a vertex's edges, we have to look at the entire row which has a width of V

$O(V^2)$

Breadth-first Search



BFS using an adjacency list

Every vertex is visited once.

When a vertex is visited, we process every edge incident on that vertex

So we visit every vertex once and every directed edge once.

O(V+E)

How many vertices?

How many edges?

Breadth-first Search

The BFS algorithm is particularly useful for one thing – finding the shortest path on unweighted graphs.

BFS can find the shortest path for an **unweighted** graph.

We will use a different algorithm to find the shortest path on a weighted graph.

Shortest does not mean unique – there can be multiple shortest paths.

Breadth-first Search

Shortest does not mean unique – there can be multiple shortest paths.

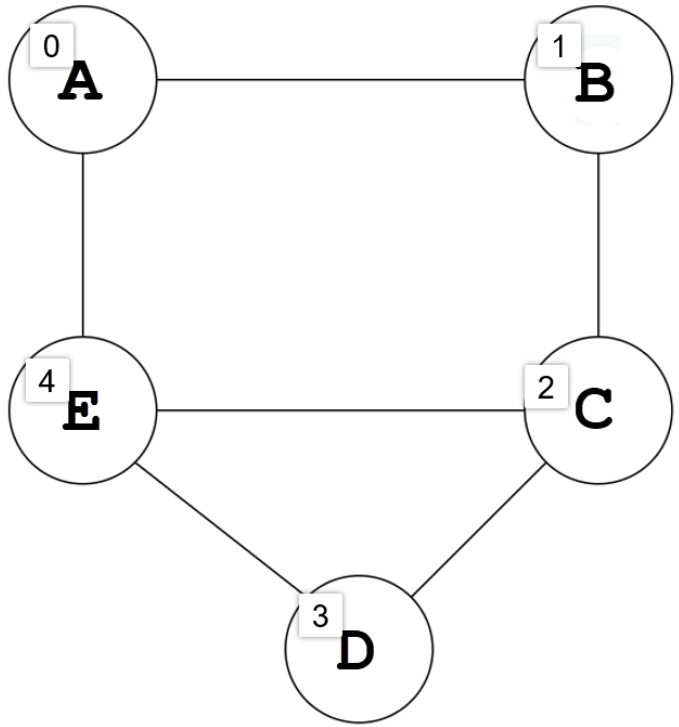
Which shortest path will be chosen will depend on the order of the vertices and how they are processed during the BFS.

Since all edges are equal in an unweighted graph, all shortest paths are valid but only one will be chosen by the code.

Breadth-first Search

The first step is to add more information to our vertex structure.

```
typedef struct
{
    char label;
    int distance;
    int previous;
    int visited;
}
Vertex;
```

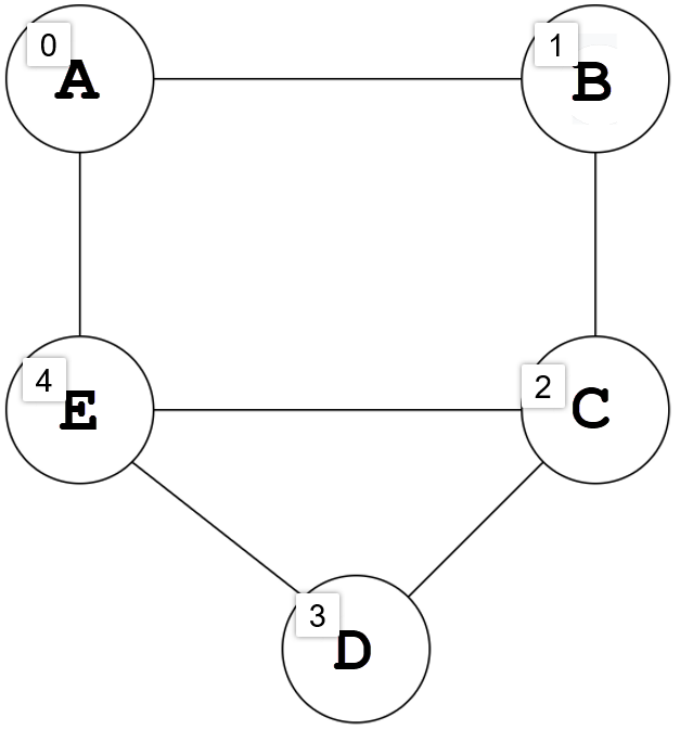


Breadth-first Search

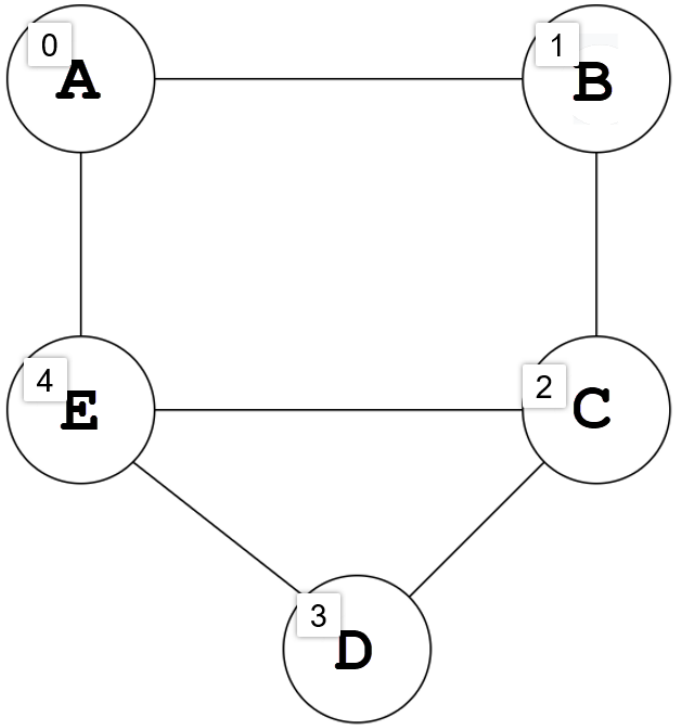
When doing a BFS, we pick one vertex to be our starting point. That vertex is the vertex we put into the vertex array at cell 0.

The new `distance` attribute in our vertex is recording how away from the starting vertex that vertex is.

Vertex B is 1 away from Vertex A (our starting vertex) so `distance` in Vertex B would be set to 1.



Breadth-first Search



So how would we fill in `distance` in the Vertex Array?

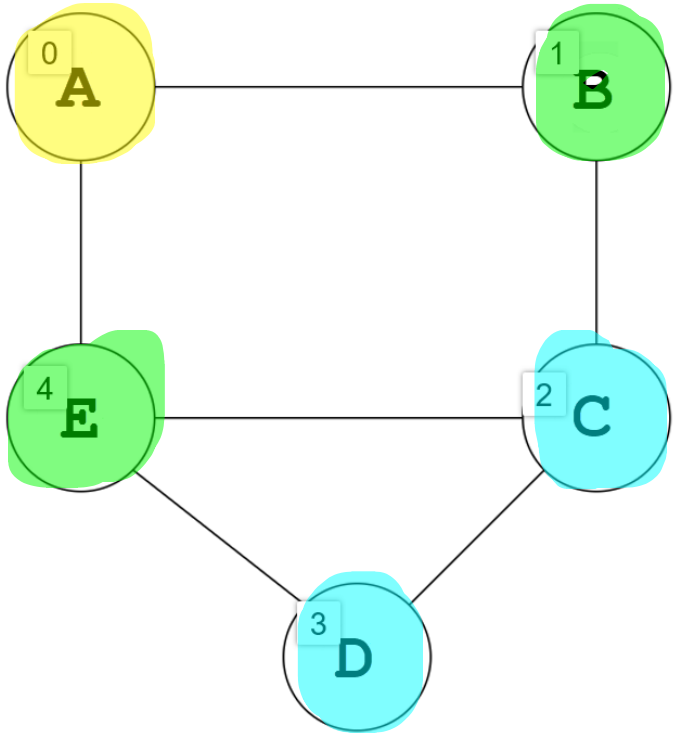
We initialize `distance` to -1 to show a distance has either not been calculated or that there is no edge connecting that vertex to the starting vertex.

	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	-1	-1	-1	-1	-1

Breadth-first Search

Notice that the distance of each vertex from the source/starting vertex corresponds to the levels we create when doing a BFS.

A | B|E | C|D



A^0

B^1 E^4

C^2 D^3

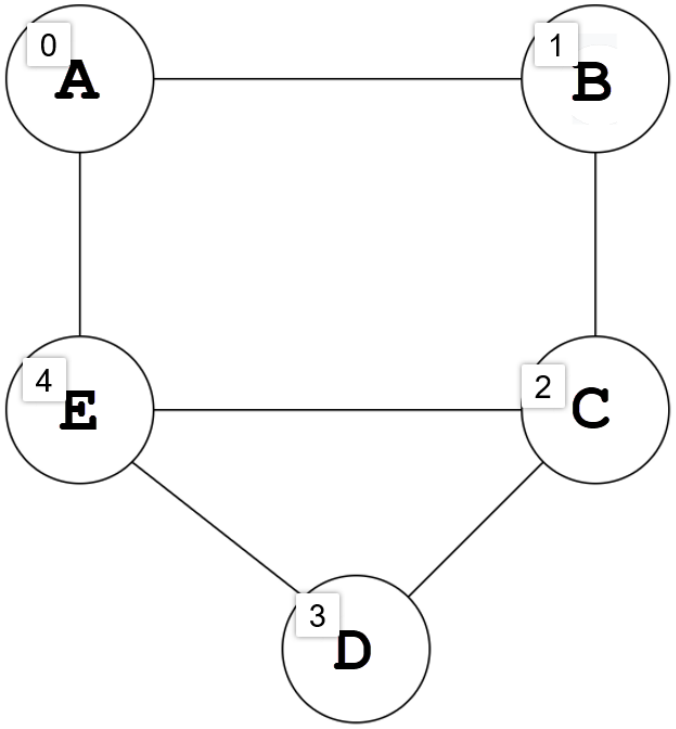
	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance					

Breadth-first Search

Now we want to fill in our other new attribute, `previous`.

The `previous` member of the `Vertex` struct contains the index of the vertex we just passed through on our way to the current vertex.

For example, to get to vertex B, we started at vertex A so we would put vertex A's index value into vertex B's `previous`.

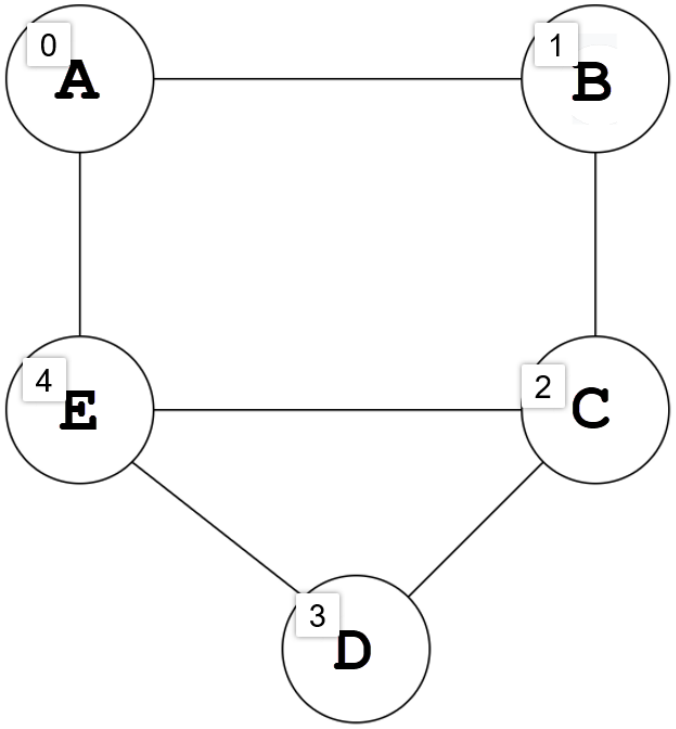


Breadth-first Search

So how would we fill in `previous` in the Vertex Array?

We initialize `previous` to -1 to show that this attribute has not been set or that the vertex is not connected to another vertex.

	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	0	1	2	2	1
Previous	-1	-1	-1	-1	-1



Breadth-first Search

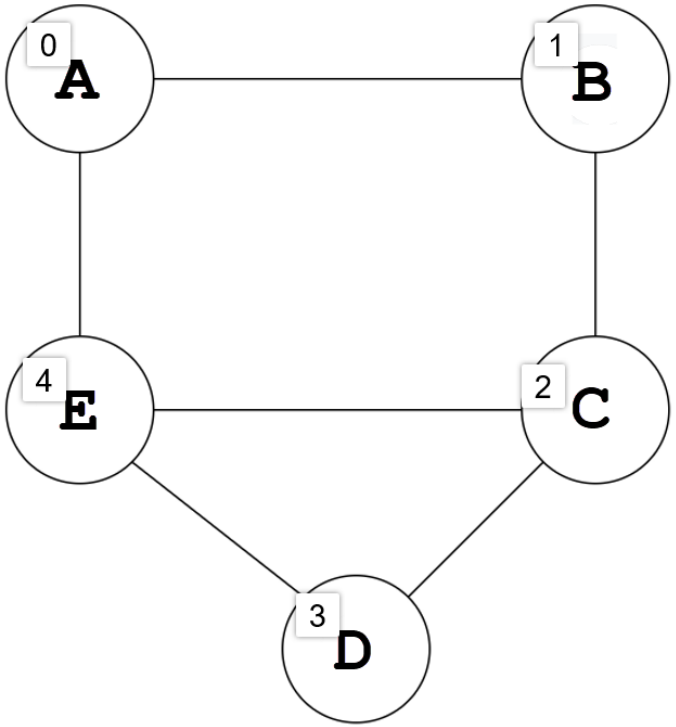
As we do our BFS, we will fill in `previous`.

We start with A.

From A, we go to B and E.

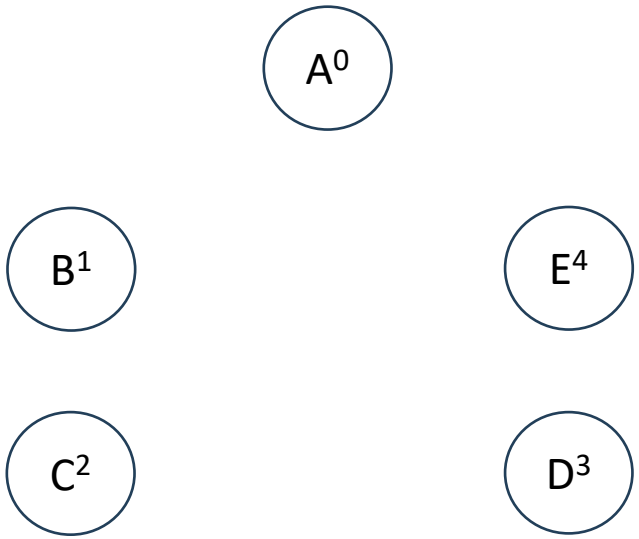
We would set `previous` in B and E to A.

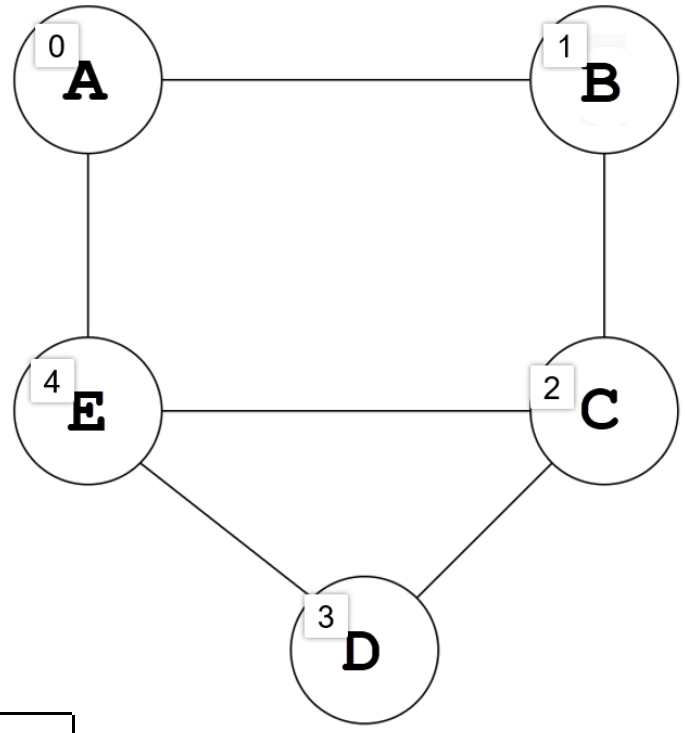
	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	0	1	2	2	1
Previous	-1	-1	-1	-1	-1



Breadth-first Search

A B|E C|D



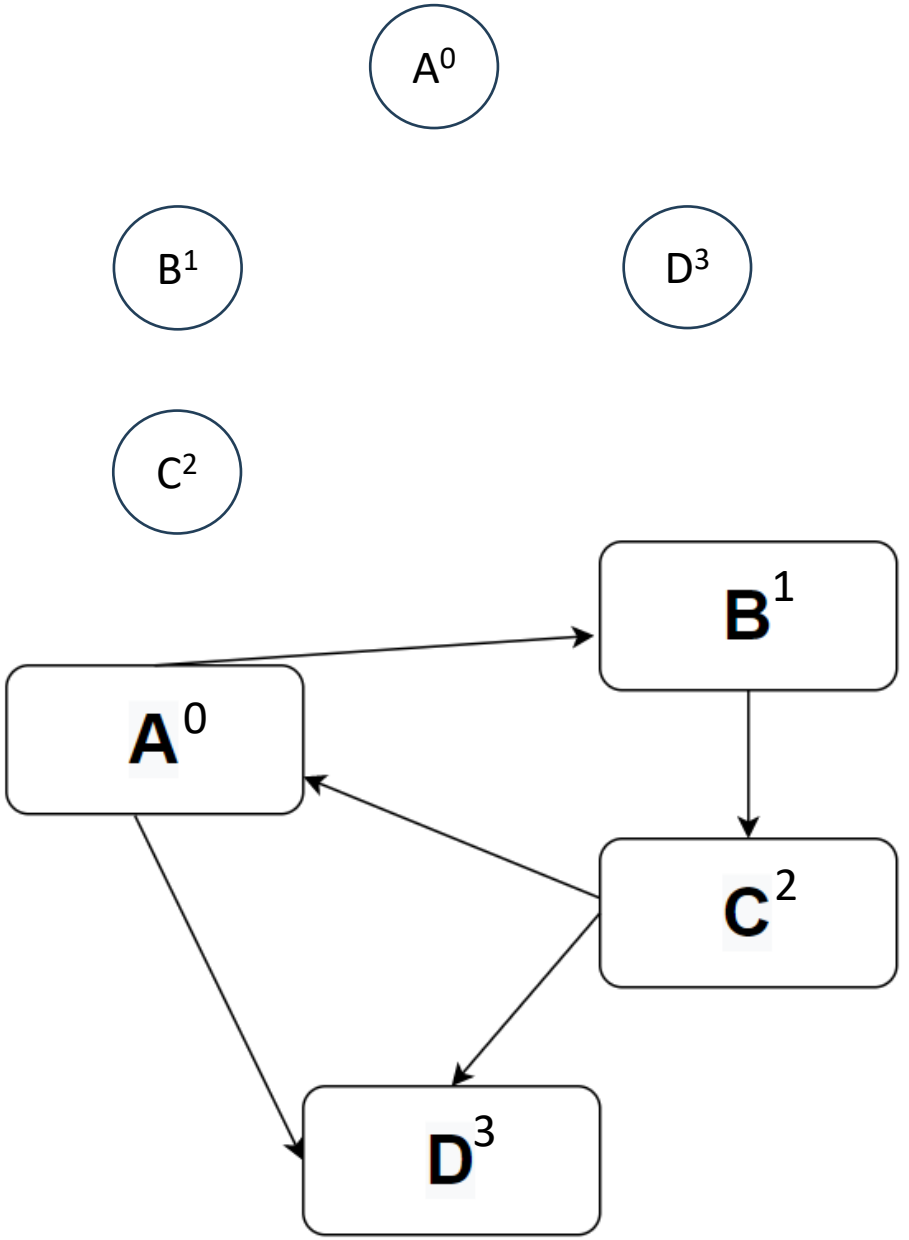


	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	-1	1	2	2	1
Previous	-1	-1	-1	-1	-1

Breadth-first Search

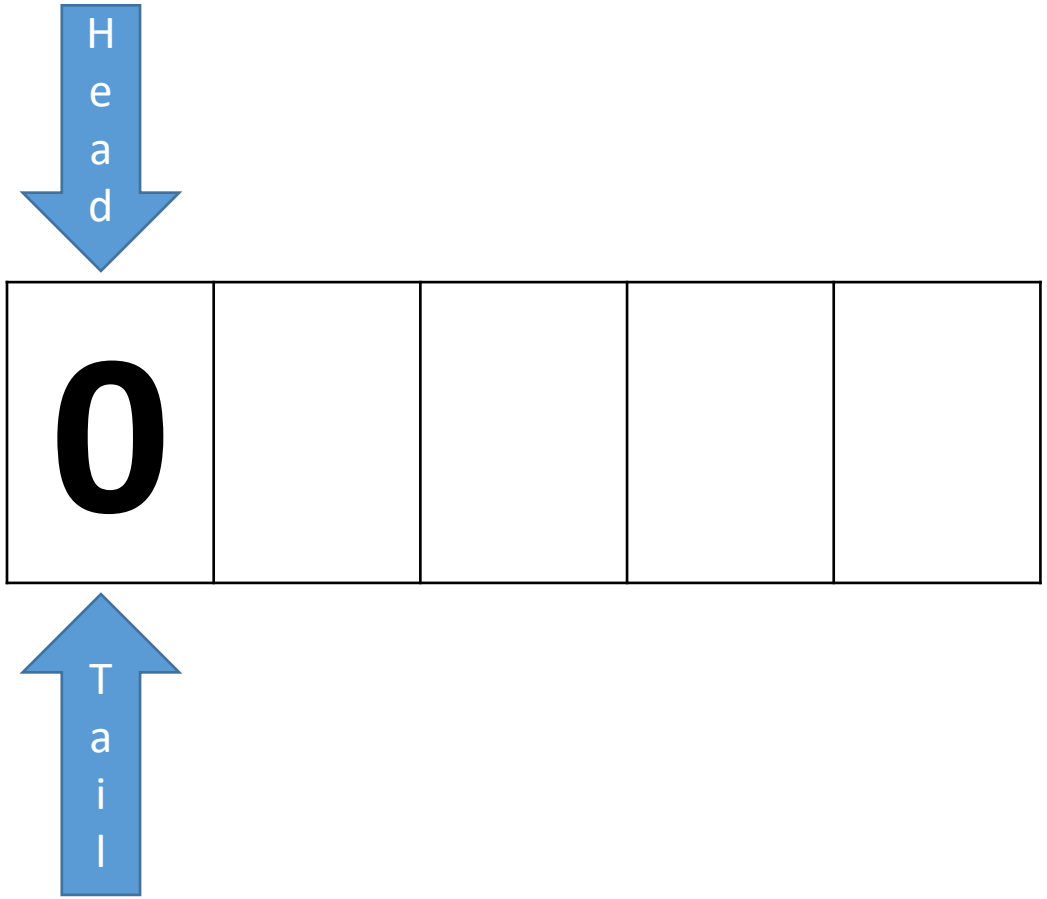
	0	1	2	3
Label	A	B	C	D
Visited	0	0	0	0
Distance	-1	-1	-1	-1
Previous	-1	-1	-1	-1

	0	1	2	3
Label	A	B	C	D
Visited	1	1	1	1
Distance	0	1	2	1
Previous	-1	0	1	0



Breadth-first Search

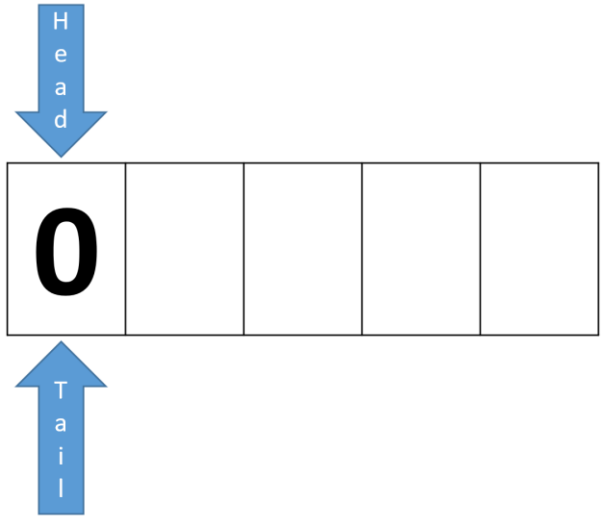
```
void BreadthFirstSearch(Vertex *VertexArray[], int VertexCount, int AdjMatrix[][MAX])
{
        int tail = -1;
        int head = -1;
        int i = 0;
        int queueItemCount = 0;
        int queue[MAX] = {};
        int CurrentVertexIndex = 0;

        VertexArray[0]->visited = 1;
        VertexArray[0]->previous = -1;
        VertexArray[0]->distance = 0;

        enqueue(queue, &head, &tail, 0);
        queueItemCount++;
```



queueItemCount

1

Breadth-first Search

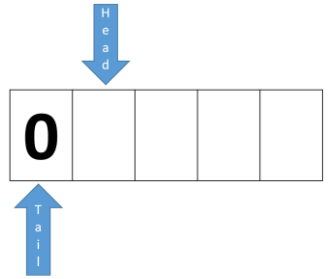


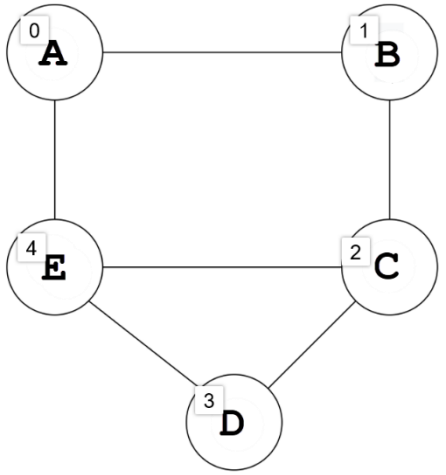
queueItemCount
1

```
while(queueItemCount)
{
    CurrentVertexIndex = dequeue(queue, &head, &tail);
    queueItemCount--;

    for (i = 0; i < VertexCount; i++)
    {
        if (AdjMatrix[CurrentVertexIndex][i] == 1)    /* Found a neighbor */
        {
            if (VertexArray[i]->visited == 0)  // have we visited already?
            {
                enqueue(queue, &head, &tail, i);
                queueItemCount++;
                VertexArray[i]->visited = 1;
                VertexArray[i]->distance = VertexArray[CurrentVertexIndex]->distance + 1;
                VertexArray[i]->previous = CurrentVertexIndex;
            }
        }
    }
}
```

CurrentVertexIndex =
i =

Breadth-first Search

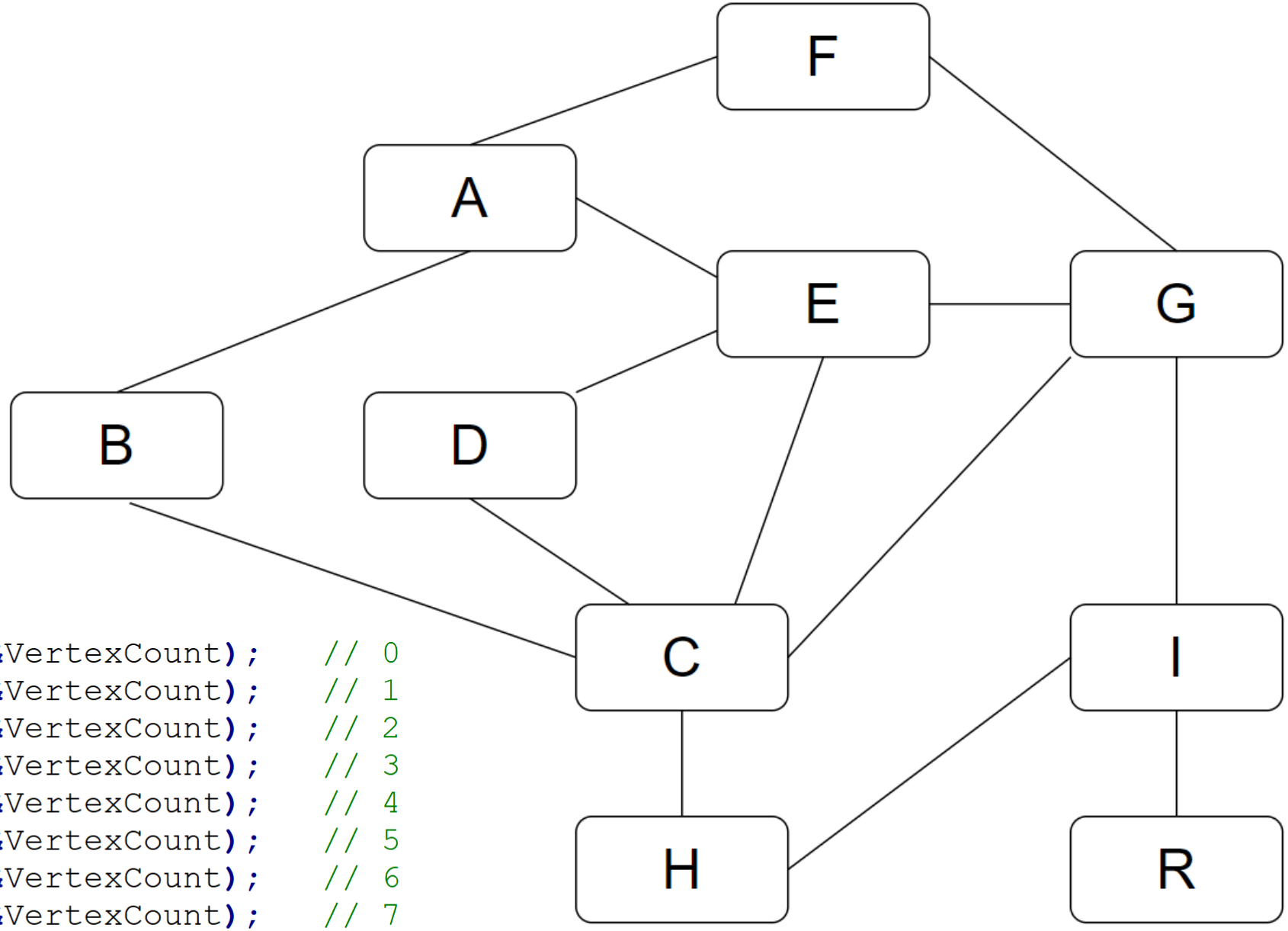





```
// Visited and neighbor?
enqueue(queue, &head, &tail, i);
queueItemCount++;
VertexArray[i]->visited = 1;
VertexArray[i]->distance = VertexArray[CurrentVertexIndex]->distance + 1;
VertexArray[i]->previous = CurrentVertexIndex;
```

	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	1	1
Distance	0	1	2	2	1
Previous	-1	0	1	4	0

```
addEdge(0, 1, AdjMatrix);
addEdge(1, 2, AdjMatrix);
addEdge(2, 3, AdjMatrix);
addEdge(3, 4, AdjMatrix);
addEdge(4, 5, AdjMatrix);
addEdge(5, 6, AdjMatrix);
addEdge(6, 0, AdjMatrix);
addEdge(8, 3, AdjMatrix);
addEdge(7, 8, AdjMatrix);
addEdge(3, 6, AdjMatrix);
addEdge(6, 7, AdjMatrix);
addEdge(6, 8, AdjMatrix);
addEdge(1, 8, AdjMatrix);
addEdge(4, 9, AdjMatrix);
```

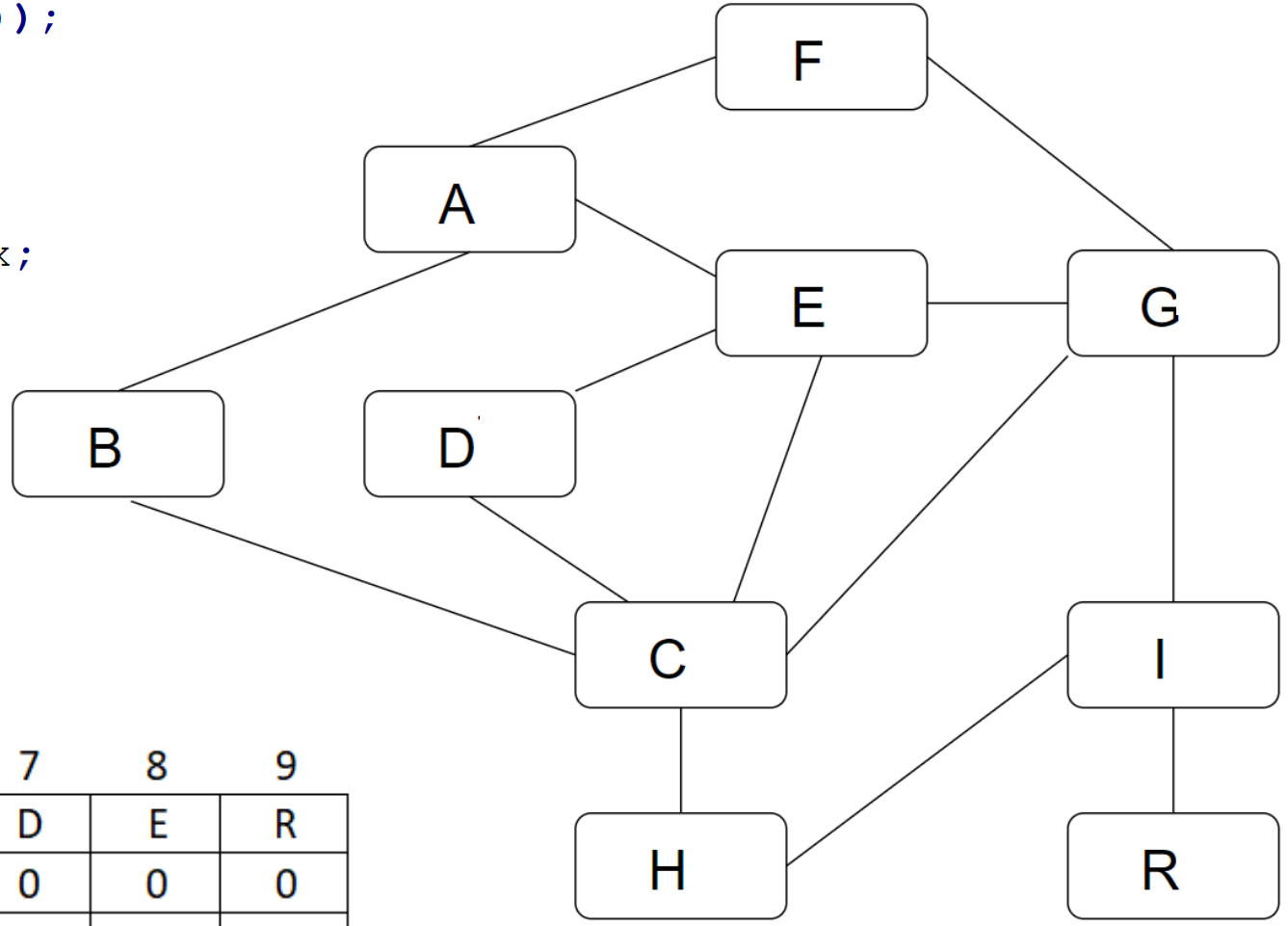



```
addVertex('B', VertexArray, &VertexCount);    // 0
addVertex('A', VertexArray, &VertexCount);    // 1
addVertex('F', VertexArray, &VertexCount);    // 2
addVertex('G', VertexArray, &VertexCount);    // 3
addVertex('I', VertexArray, &VertexCount);    // 4
addVertex('H', VertexArray, &VertexCount);    // 5
addVertex('C', VertexArray, &VertexCount);    // 6
addVertex('D', VertexArray, &VertexCount);    // 7
addVertex('E', VertexArray, &VertexCount);    // 8
addVertex('R', VertexArray, &VertexCount);    // 9
```

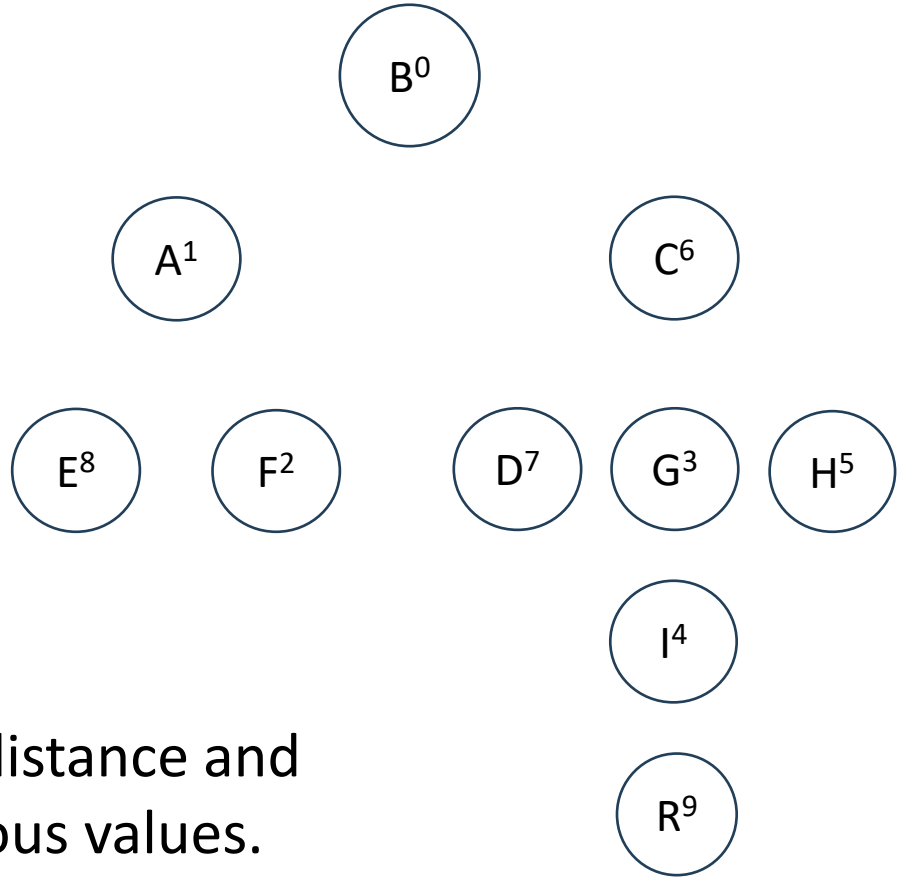
```
void addVertex(char label, Vertex *VertexArray[], int *VertexCount)
{
    Vertex *NewVertex = malloc(sizeof(Vertex));
    NewVertex->label = label;
    NewVertex->visited = 0;
    NewVertex->previous = -1;
    NewVertex->distance = -1;
    VertexArray[(*VertexCount)++] = NewVertex;
}
```

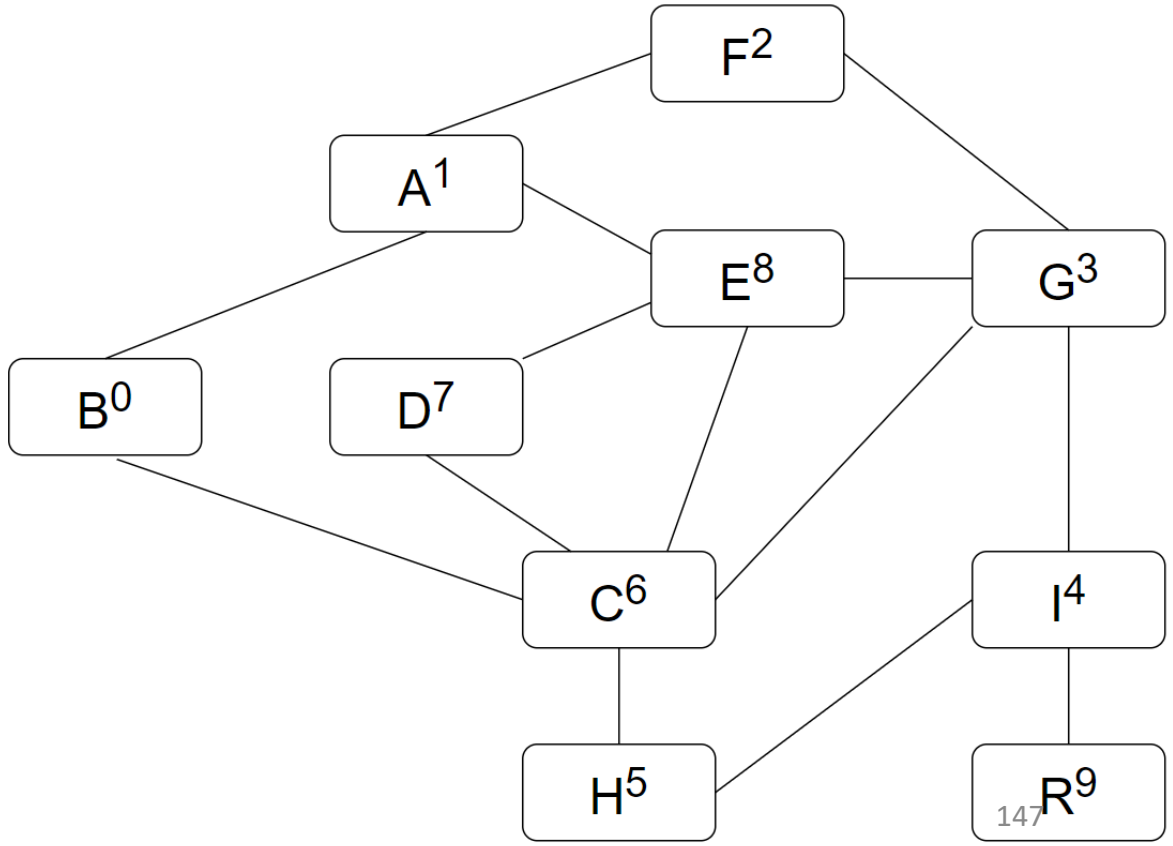
```
addVertex('B', VertexArray, &VertexCount);   // 0
addVertex('A', VertexArray, &VertexCount);   // 1
addVertex('F', VertexArray, &VertexCount);   // 2
addVertex('G', VertexArray, &VertexCount);   // 3
addVertex('I', VertexArray, &VertexCount);   // 4
addVertex('H', VertexArray, &VertexCount);   // 5
addVertex('C', VertexArray, &VertexCount);   // 6
addVertex('D', VertexArray, &VertexCount);   // 7
addVertex('E', VertexArray, &VertexCount);   // 8
addVertex('R', VertexArray, &VertexCount);   // 9
```

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	0	0	0	0	0	0	0	0	0	0
Distance	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Previous	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1



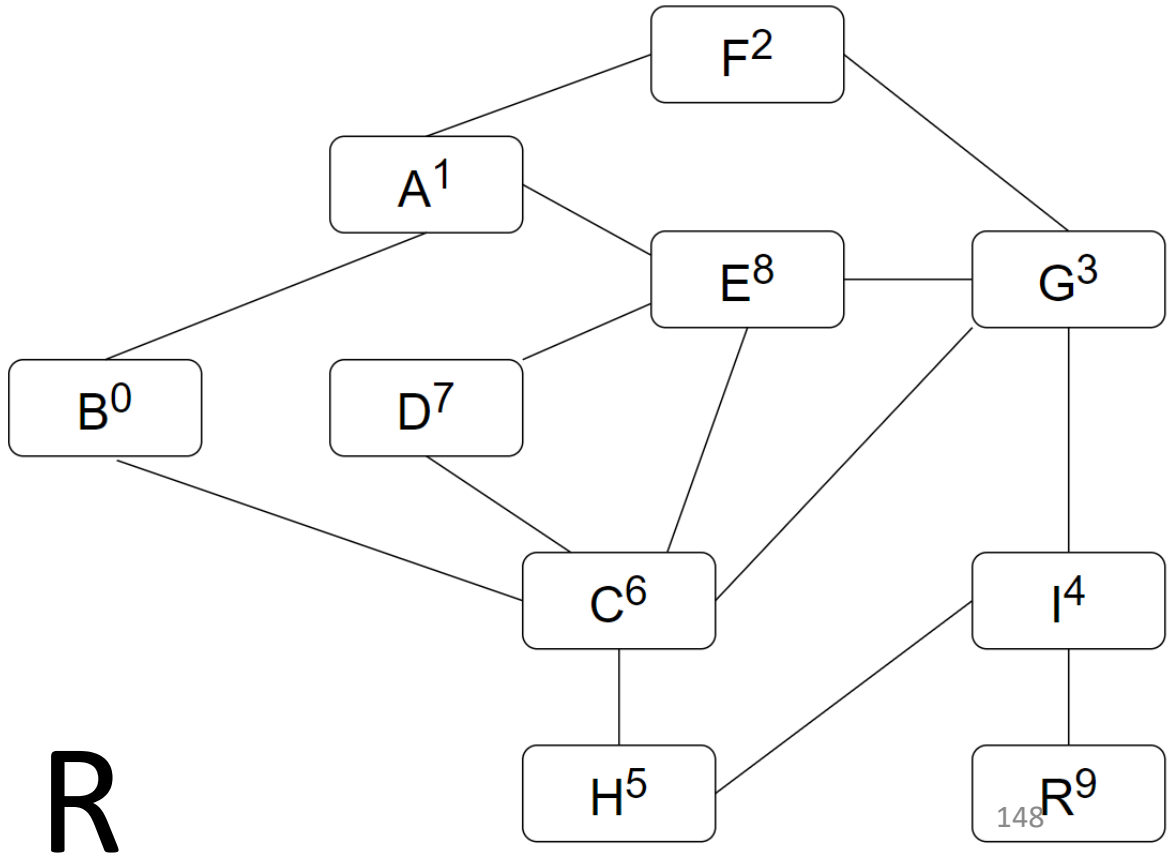
	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	0	0	0	0	0	0	0	0	0	0
Distance	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Previous	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1





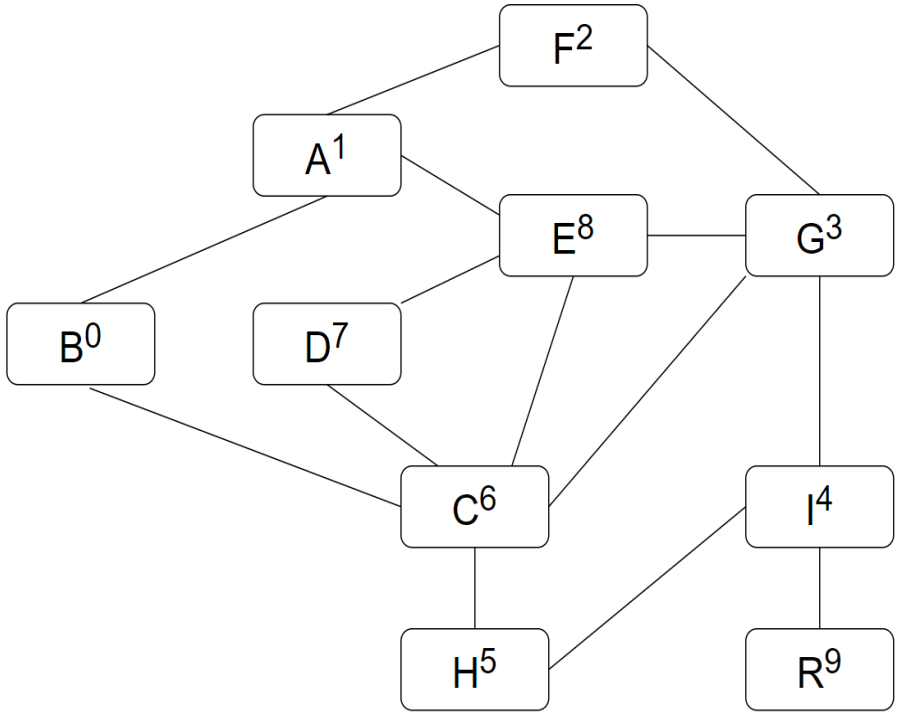
Find distance and previous values.

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4



B A|C EF|DGH I R

```
Source is B.  What is the destination? A
BA
Source is B.  What is the destination? C
BC
Source is B.  What is the destination? D
BCD
Source is B.  What is the destination? E
BAE
Source is B.  What is the destination? F
BAF
Source is B.  What is the destination? G
BCG
Source is B.  What is the destination? H
BCH
Source is B.  What is the destination? I
BCGI
Source is B.  What is the destination? R
BCGIR
```



	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

B A|C EF|DGH I R

```
Source is B.  What is the destination? W
Destination W is not in graph

Source is B.  What is the destination? A
BA
```

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```
printf("Source is %c.  What is the destination? ", VertexArray[0]->label);
scanf(" %c", &dest);

destindex = 0;
while (destindex < VertexCount && dest != VertexArray[destindex]->label)
{
   destindex++;
}

if (destindex == VertexCount)
   printf("Destination %c is not in graph\n", dest);
else
{

}
```

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```
char path[10] = {};

int pathindex = -1;

int destindex = -1;

int previndex = -1;
```

```
pathindex = VertexArray[destindex]->distance;
previndex = VertexArray[destindex]->previous;
path[pathindex] = VertexArray[destindex]->label;

while (pathindex > 0)
{
      pathindex--;
      path[pathindex] = VertexArray[previndex]->label;
      previndex = VertexArray[previndex]->previous;
}
printf("%s\n", path);
```

dest = 'A'

BA

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```
char path[10] = {};

int pathindex = -1;

int destindex = -1;

int previndex = -1;
```

```
pathindex = VertexArray[destindex]->distance;
previndex = VertexArray[destindex]->previous;
path[pathindex] = VertexArray[destindex]->label;

while (pathindex > 0)
{
        pathindex--;
        path[pathindex] = VertexArray[previndex]->label;
        previndex = VertexArray[previndex]->previous;
}
printf("%s\n", path);
```

dest = 'G'

destindex = 3

BCG

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```
char path[10] = {};

int pathindex = -1;

int destindex = -1;

int previndex = -1;
```

```
pathindex = VertexArray[destindex]->distance;
previndex = VertexArray[destindex]->previous;
path[pathindex] = VertexArray[destindex]->label;

while (pathindex > 0)
{
      pathindex--;
      path[pathindex] = VertexArray[previndex]->label;
      previndex = VertexArray[previndex]->previous;
}
printf("%s\n", path);
```

dest = 'R'

destindex = 9

BCGIR

Breadth-first Search

Using the Breadth-first Search technique to traverse a graph can give us the shortest path between two vertices when we keep track of the distance from the source vertex to every vertex and each vertex's previous vertex.

This shortest path is not unique, but it is the shortest.

This technique only works for unweighted graphs.