

CSE 2320

Week of 07/13/2020

Instructor : Donna French

Adjacency Matrix

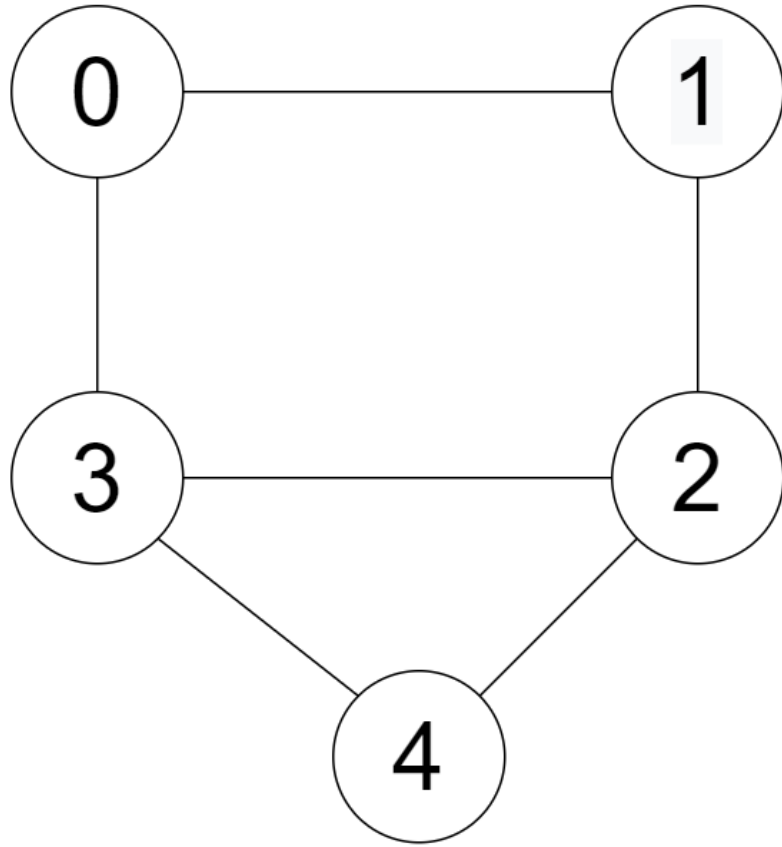
To create an adjacency matrix, we need a list of edges.

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

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

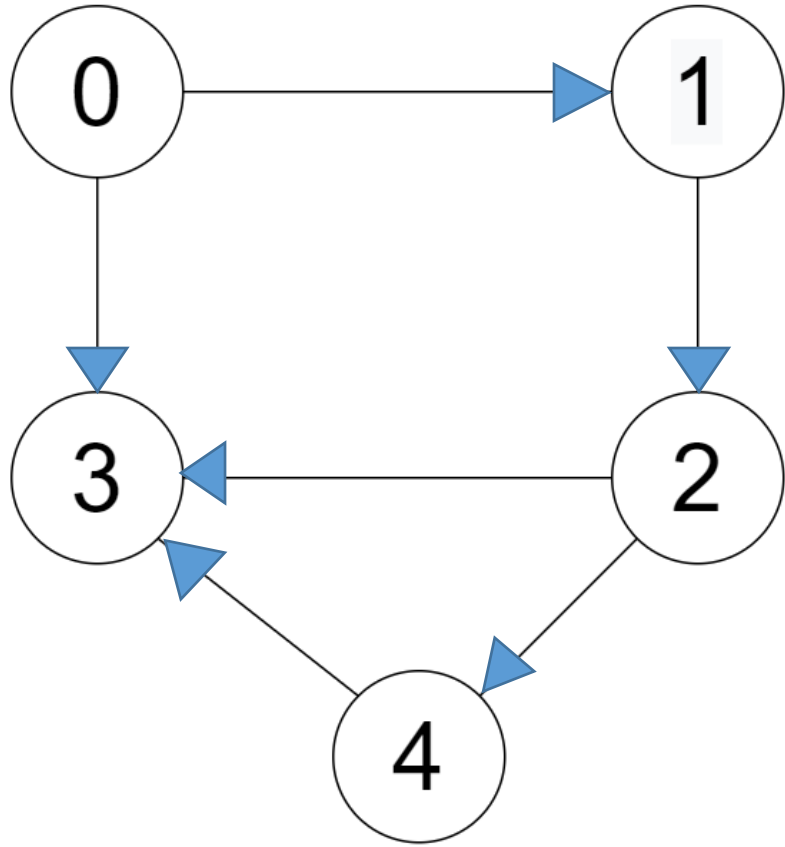
Adjacency Matrix

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

```
gcc GraphAM.c -g
```

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

Adjacency Matrix



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?

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  
}
```

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

```
01010
```

```
10100
```

```
01011
```

```
10101
```

```
00110
```

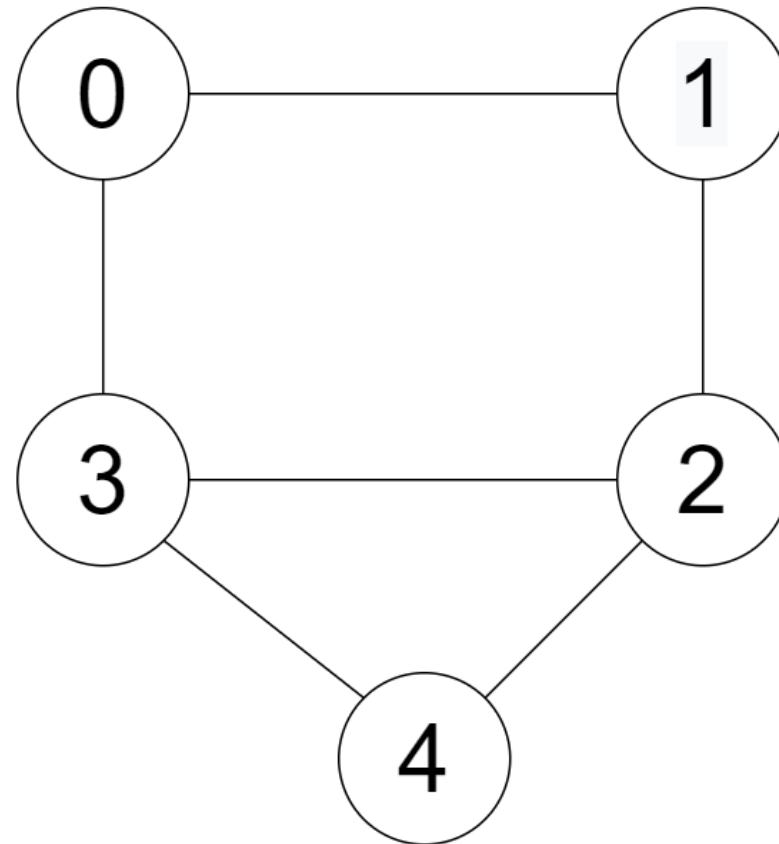
```
Enter a vertex 3
```

```
Vertex 3 is adjacent to vertex 0
```

```
Vertex 3 is adjacent to vertex 2
```

```
Vertex 3 is adjacent to vertex 4
```

Adjacency Matrix



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

```
gcc GraphAM.c -g
```

```
01010
```

```
00100
```

```
00011
```

```
00000
```

```
00010
```

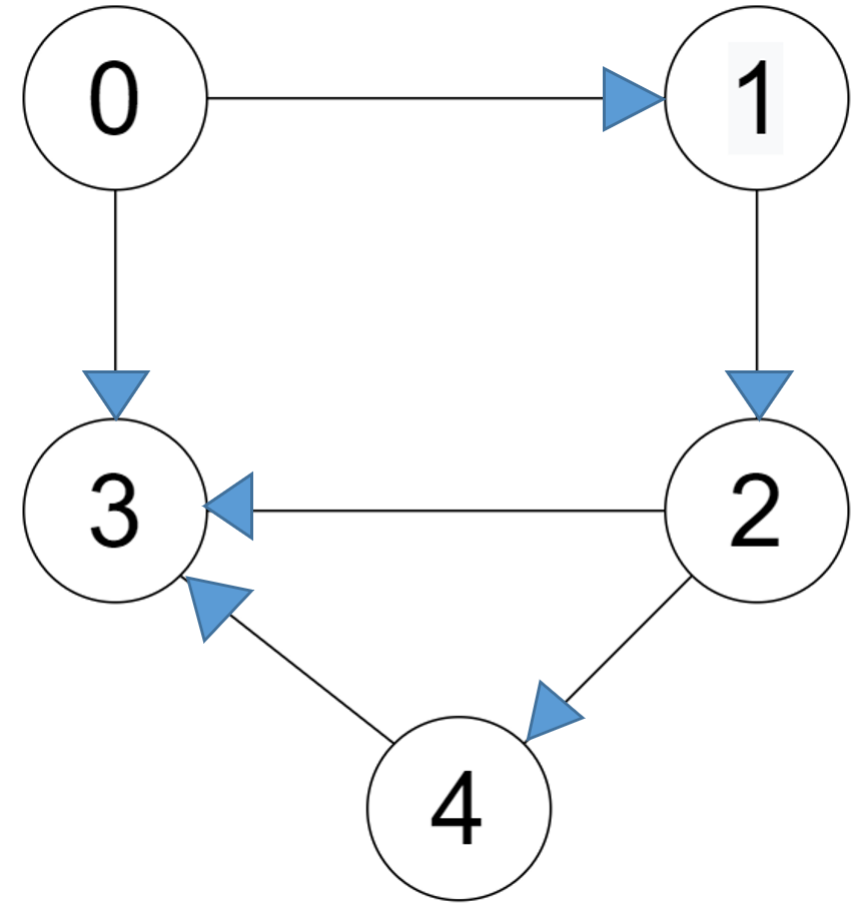
```
Enter a vertex 3
```

```
Enter a vertex 2
```

```
Vertex 2 is adjacent to vertex 3
```

```
Vertex 2 is adjacent to vertex 4
```

Adjacency Matrix



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 –

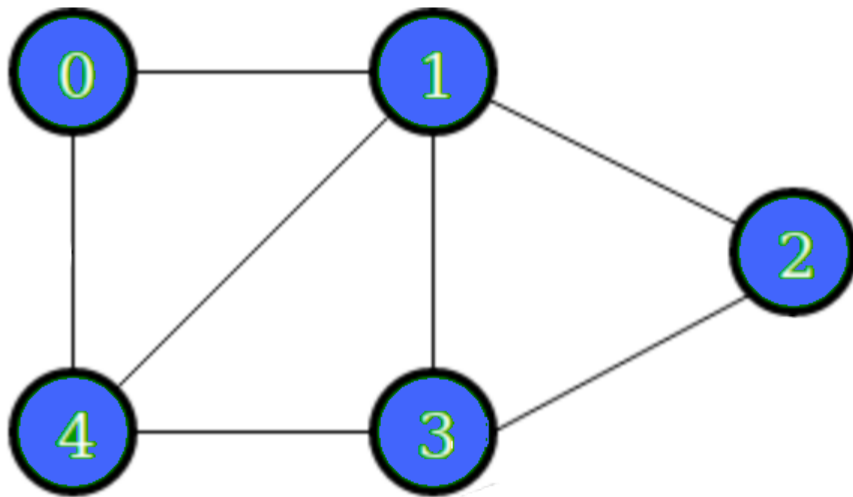
There is a specially designated node called root.

The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, T_2, T_3, \dots, T_n$

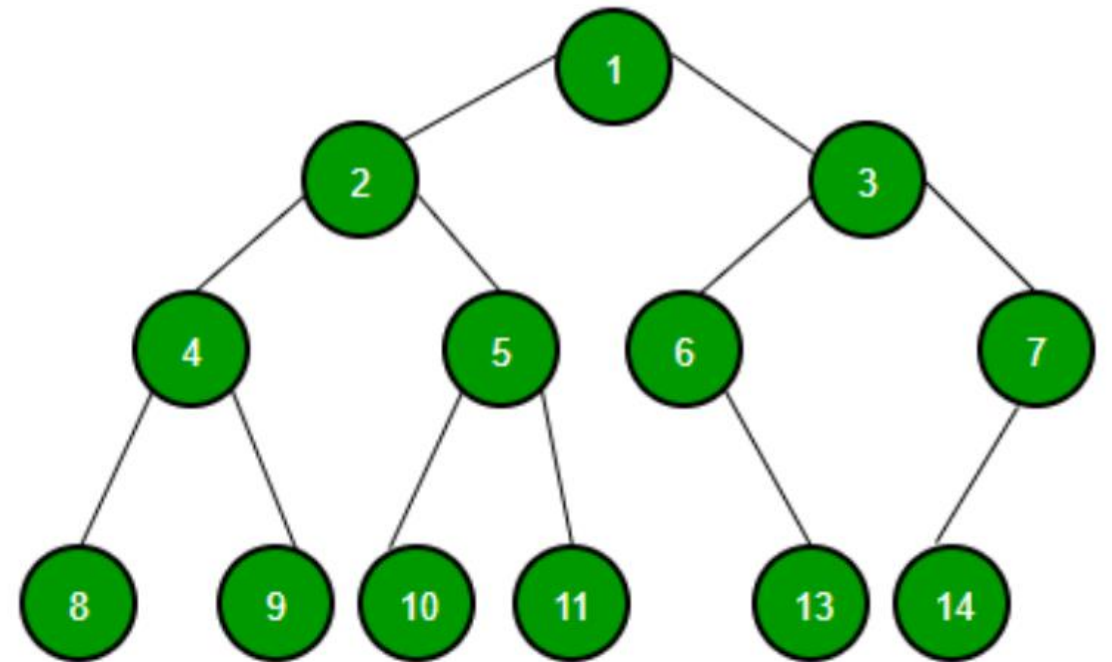
where $T_1, T_2, T_3, \dots, T_n$ are called the subtrees of the root.

Graph vs Tree

Graph



Tree

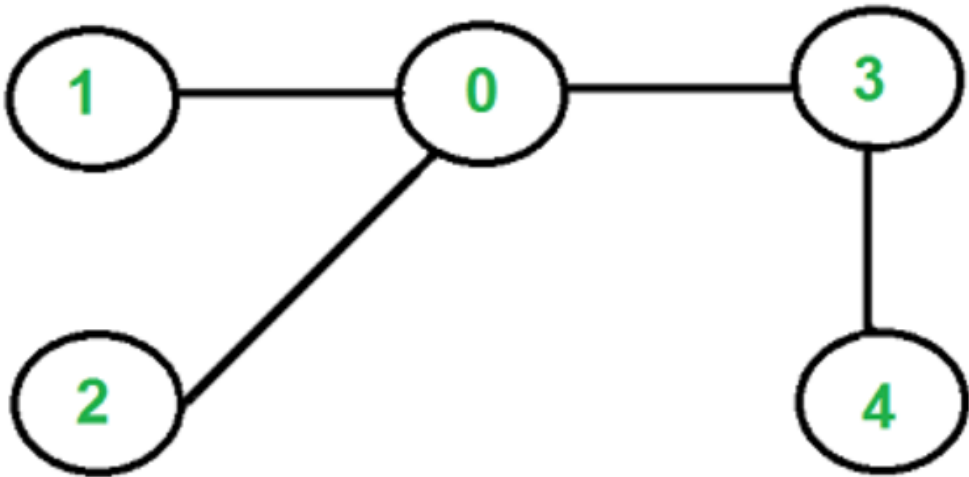


Graph vs Tree

Graph	Tree
Non-linear data structure	Non-linear data structure
Collection of vertices and edges	Collection of nodes and edges
Each vertex can have any number of edges	General trees can have any number of child nodes. Binary trees have at most two child nodes.
No unique vertex called root	Unique node called root
A cycle can be formed	No cycles are allowed

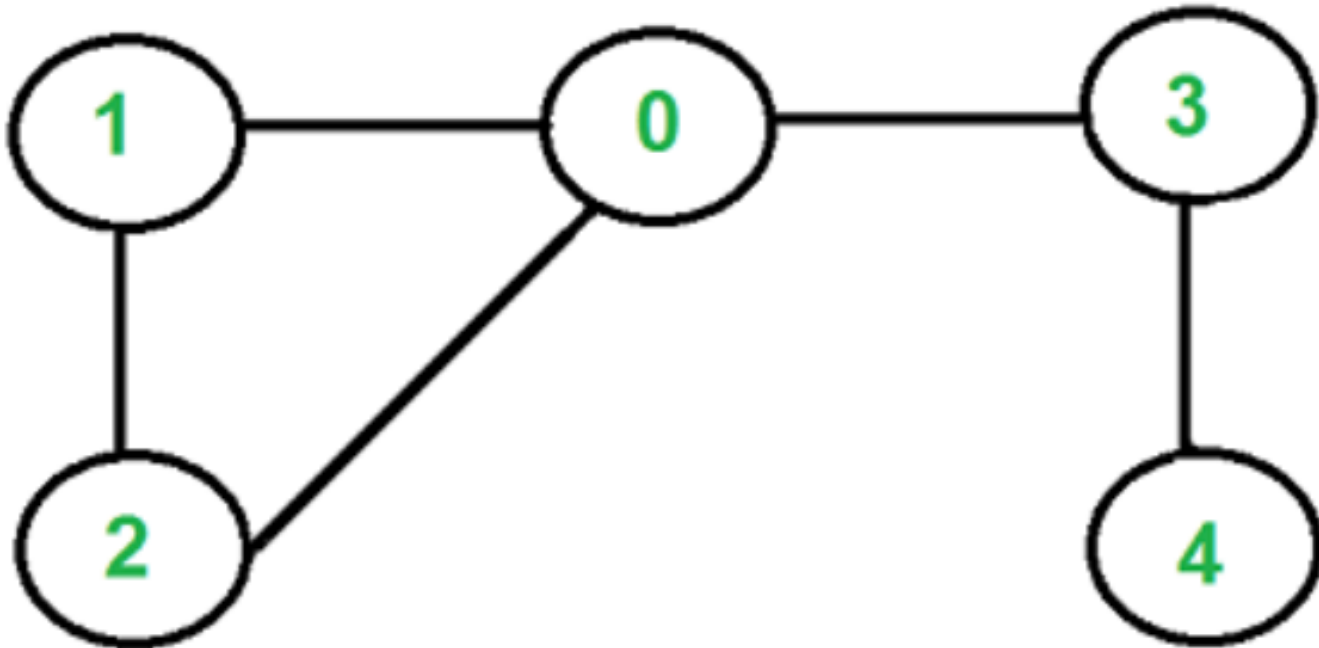
Graph vs Tree

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.

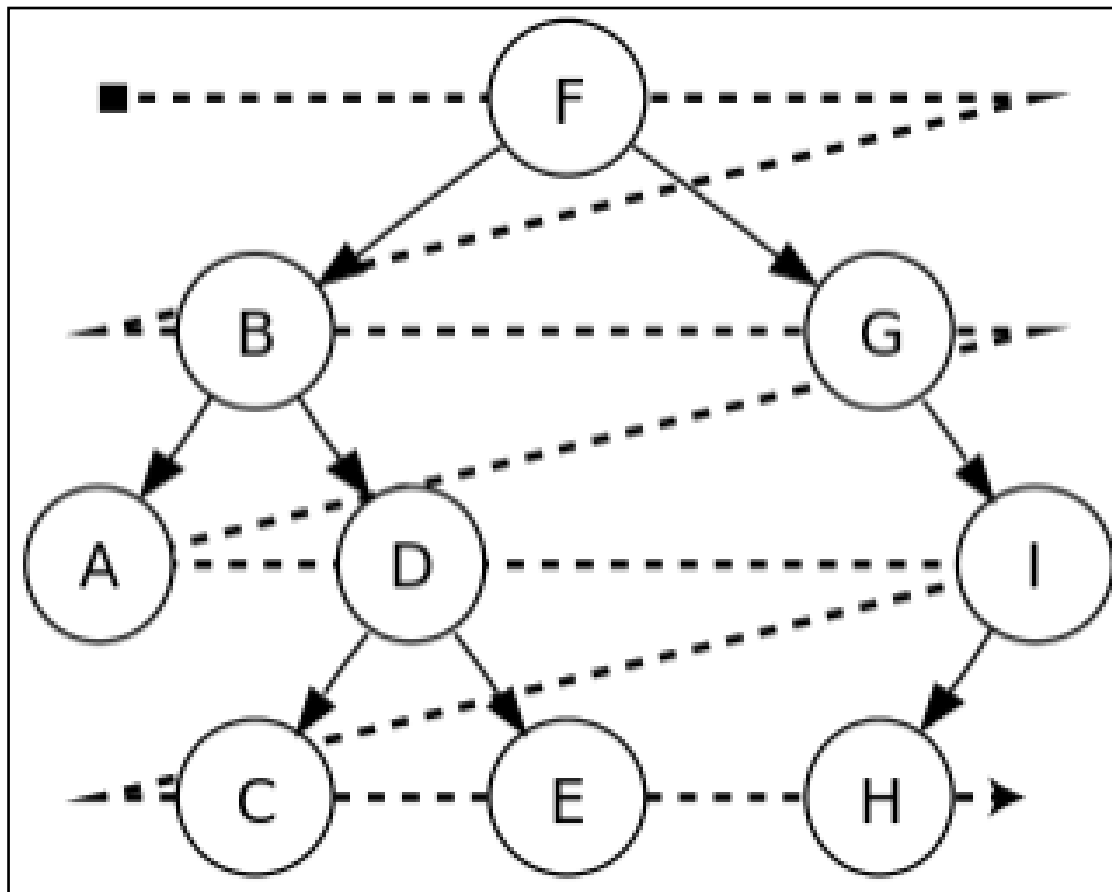
To be more specific it is all about visiting and exploring each vertex and edge in a graph such that all the vertices are explored exactly once.

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

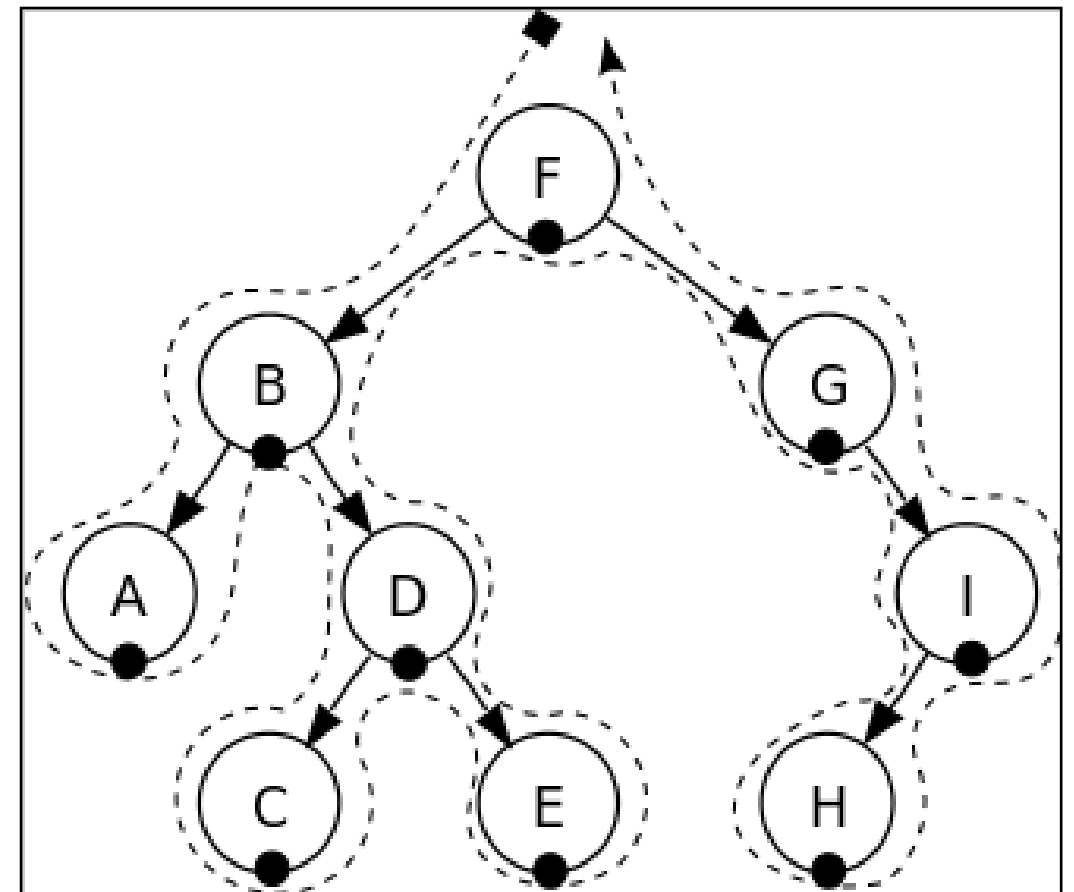
The challenge is to use a graph traversal technique that is most suitable for solving a particular problem.

Breadth-first vs Depth-first Traversal

Breadth-first



Depth-first

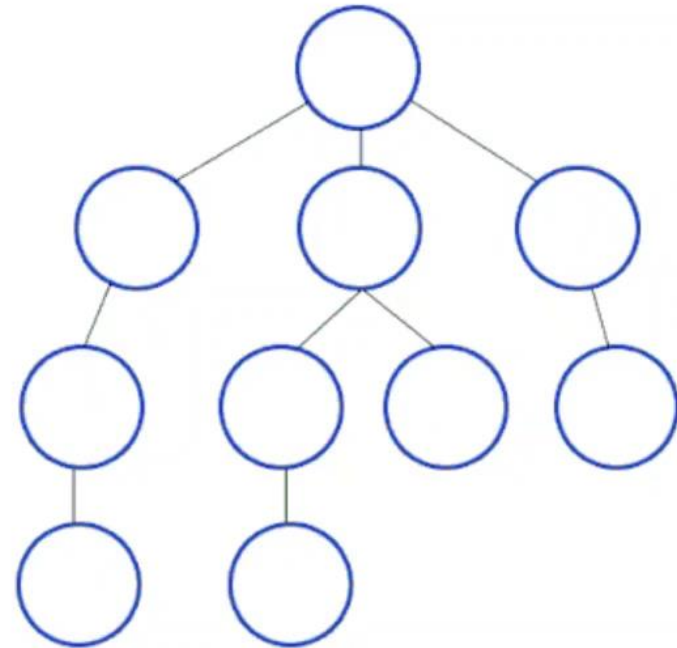
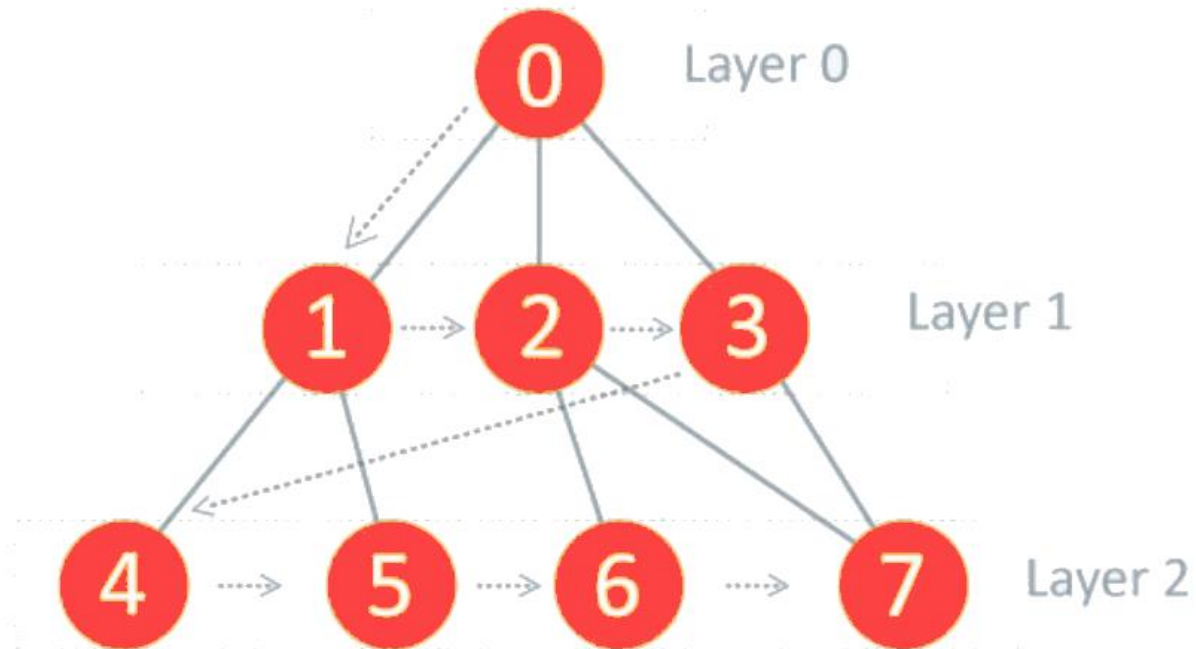


Depth-first Traversals

- 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

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.

Side Note : Array Queues

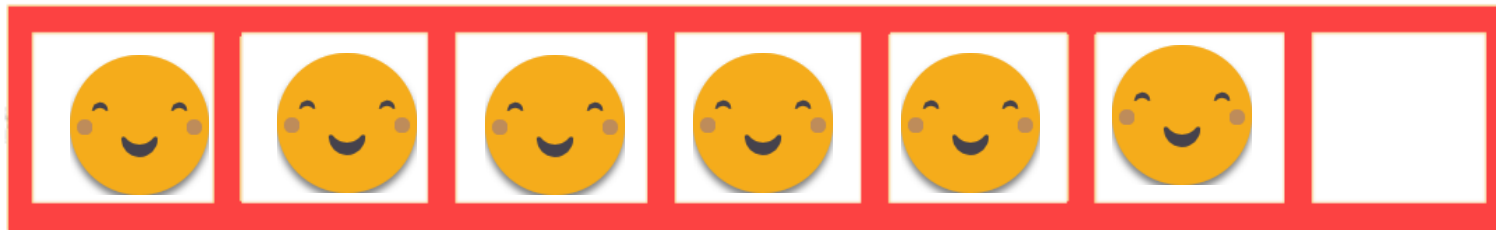
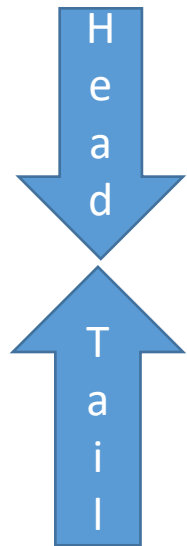
A **queue** is an abstract data structure that follows the First-In-First-Out (FIFO) methodology.

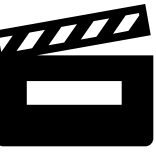
Data inserted first (FI) will be accessed first (FO).

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

A screenshot of a Linux terminal window. The title bar reads 'student@cse1325: /media/sf_VM2320'. The menu bar shows 'File Edit Tabs Help'. The prompt is 'student@cse1325: /media/sf_VM2320\$' followed by a black cursor. The terminal area is mostly empty with three small 'I' shaped cursor icons. The bottom taskbar shows the Ubuntu logo, several application icons, a '[Software Updater]' notification, and system status icons including battery, network, and volume, along with the time '17:54' and a power button icon.

```
student@cse1325: /media/sf_VM2320$
```

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     }
21
```

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 }
```

Conditional Compile

```
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(void)
6  {
7      char DayOfWeek[10];
8
9      #ifdef Monday
10         strcpy(DayOfWeek, "Monday");
11     #elif Wednesday
12         strcpy(DayOfWeek, "Wednesday");
13     #else
14         strcpy(DayOfWeek, "Friday");
15     #endif
16
17     printf("Today is %s\n\n", DayOfWeek);
18
19     return 0;
20 }
```

3 different ways to compile this code

`gcc CondComp.c`

`gcc CondComp.c -D Monday`

`gcc CondComp.c -D Wednesday`

Conditional Compile

```
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(void)
6  {
7      char DayOfWeek[10];
8
9      #ifdef Monday
10         strcpy(DayOfWeek, "Monday");
11     #elif Wednesday
12         strcpy(DayOfWeek, "Wednesday");
13     #else
14         strcpy(DayOfWeek, "Friday");
15     #endif
16
17     printf("Today is %s\n\n", DayOfWeek);
18
19     return 0;
20 }
```

```
gcc CondComp.c -E
```

```
int main(void)
{
    char DayOfWeek[10];

    strcpy(DayOfWeek, "Friday");

    printf("Today is %s\n\n", DayOfWeek);

    return 0;
}
```

Conditional Compile

```
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(void)
6  {
7      char DayOfWeek[10];
8
9      #ifdef Monday
10         strcpy(DayOfWeek, "Monday");
11     #elif Wednesday
12         strcpy(DayOfWeek, "Wednesday");
13     #else
14         strcpy(DayOfWeek, "Friday");
15     #endif
16
17     printf("Today is %s\n\n", DayOfWeek);
18
19     return 0;
20 }
```

```
gcc CondComp.c -E -D Monday
```

```
int main(void)
{
    char DayOfWeek[10];

    strcpy(DayOfWeek, "Monday");

    printf("Today is %s\n\n", DayOfWeek);

    return 0;
}
```

Conditional Compile

```
2  #include <stdio.h>
3  #include <string.h>
4
5  int main(void)
6  {
7      char DayOfWeek[10];
8
9      #ifdef Monday
10         strcpy(DayOfWeek, "Monday");
11     #elif Wednesday
12         strcpy(DayOfWeek, "Wednesday");
13     #else
14         strcpy(DayOfWeek, "Friday");
15     #endif
16
17     printf("Today is %s\n\n", DayOfWeek);
18
19     return 0;
20 }
```

```
gcc CondComp.c -E -D Wednesday
```

```
int main(void)
{
    char DayOfWeek[10];

    strcpy(DayOfWeek, "Wednesday");

    printf("Today is %s\n\n", DayOfWeek);

    return 0;
}
```

Breadth-first Search

Vertex 0 is the starting vertex.

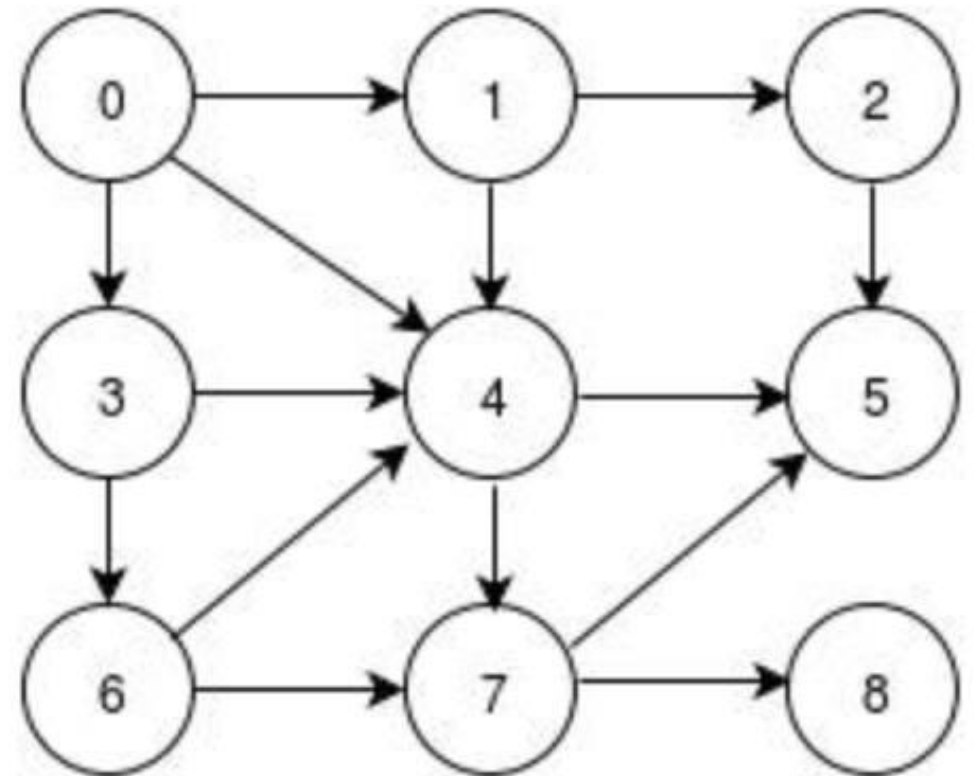
We will visit all vertices adjacent to vertex 0

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

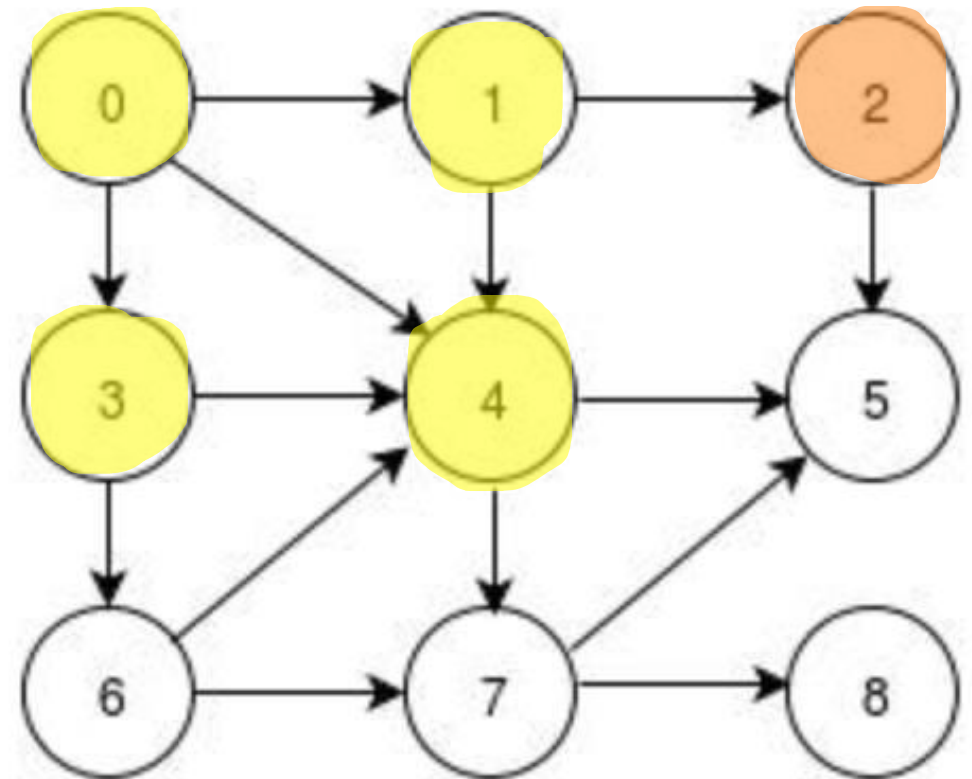
We mark 0, 1, 4 and 3 as "visited".

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

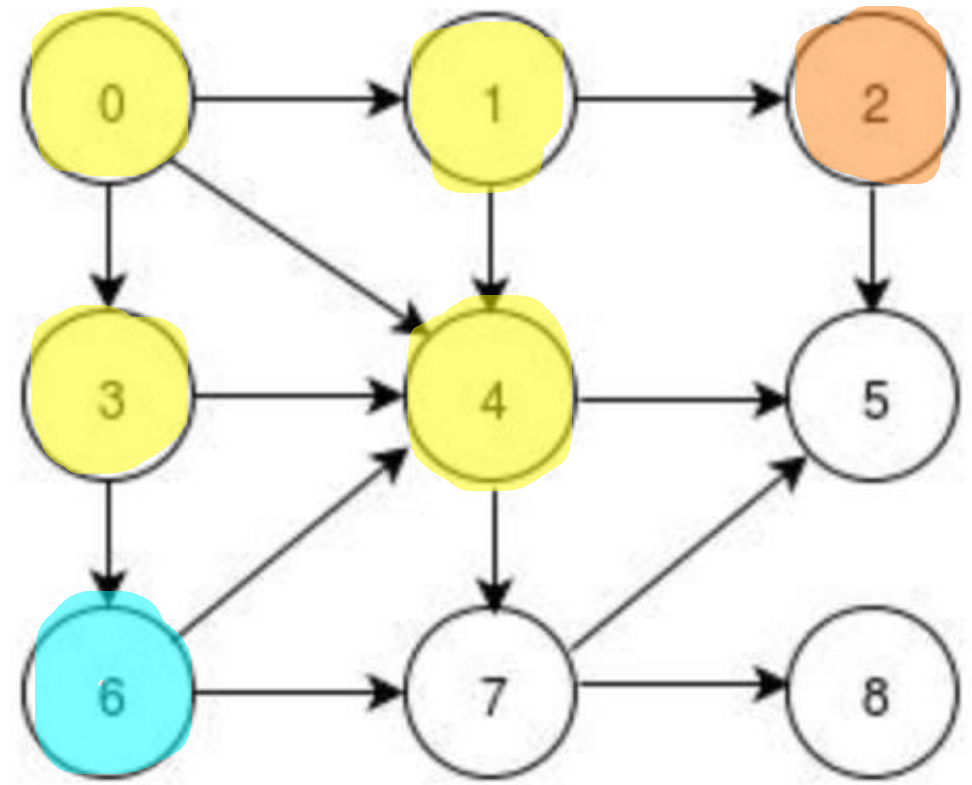
0 1 3 4 2

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

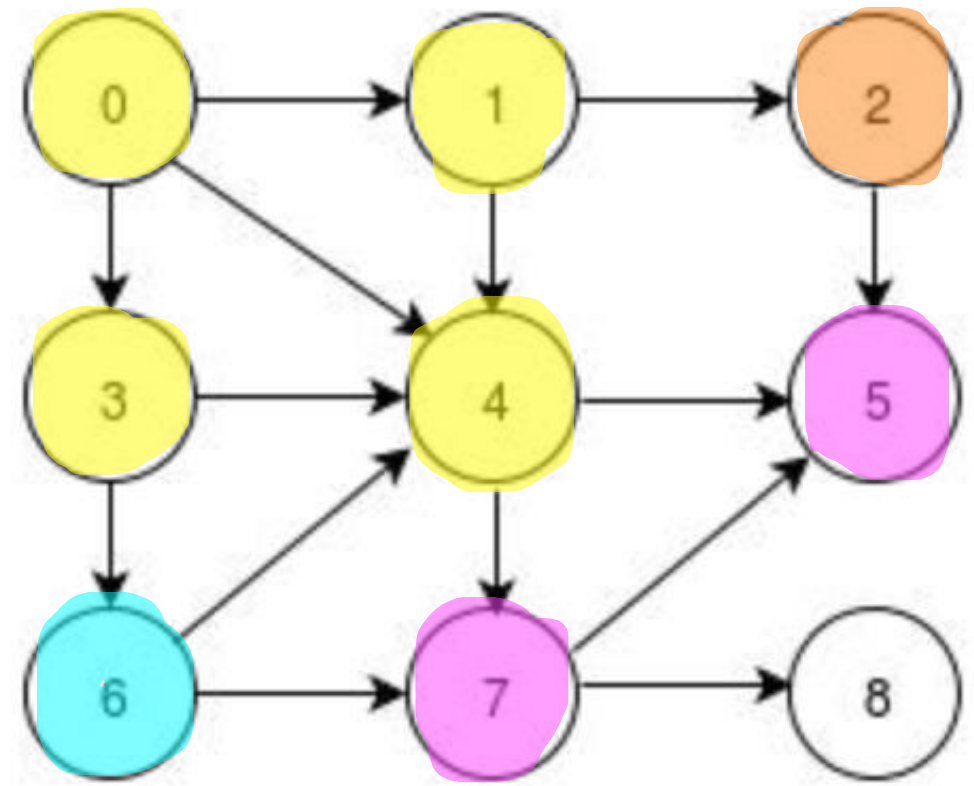
0 1 3 4 2 6

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

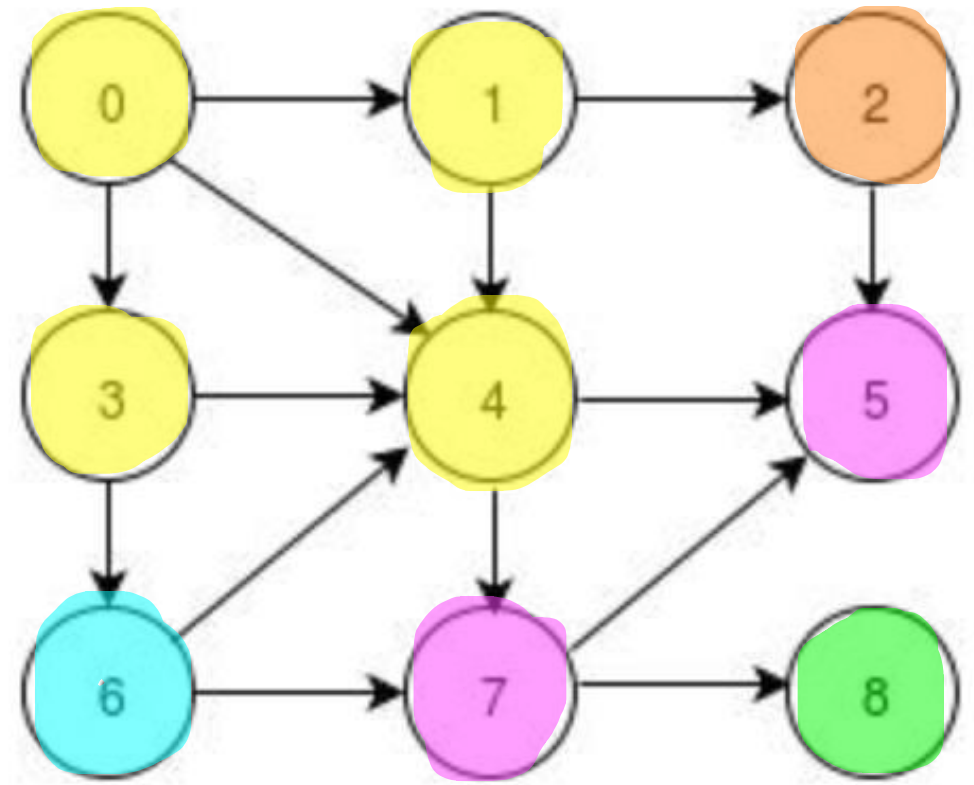
0 1 3 4 2 6 5 7

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.

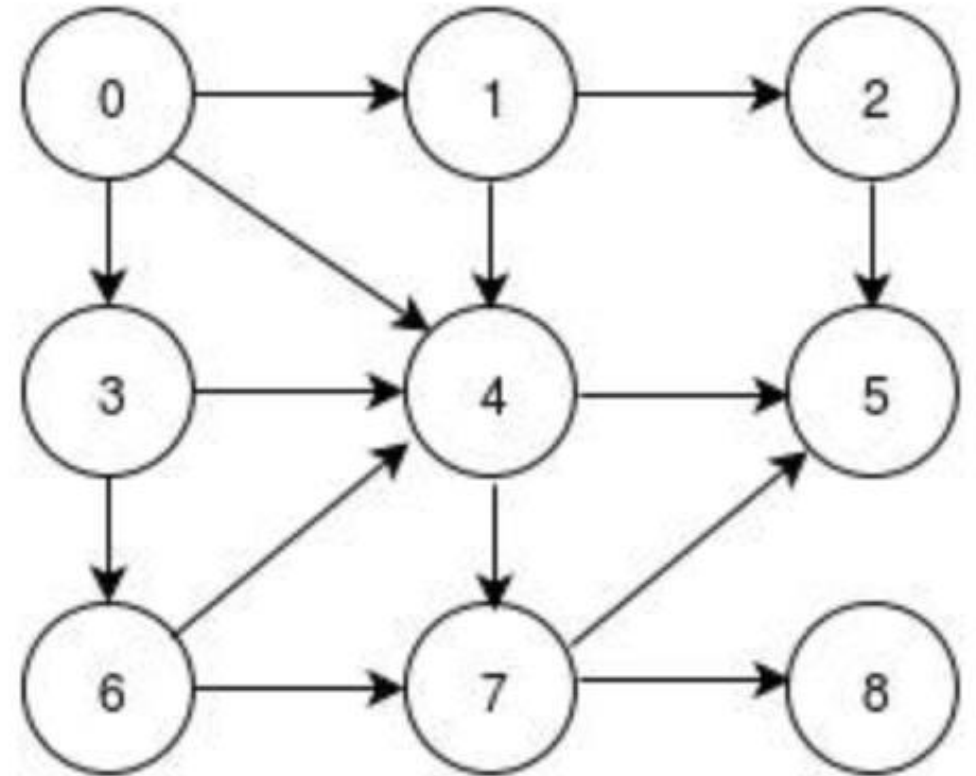
Now we visit all vertices adjacent to 7
8 has not been visited so we add it to our traversal



0 1 3 4 2 6 5 7 8

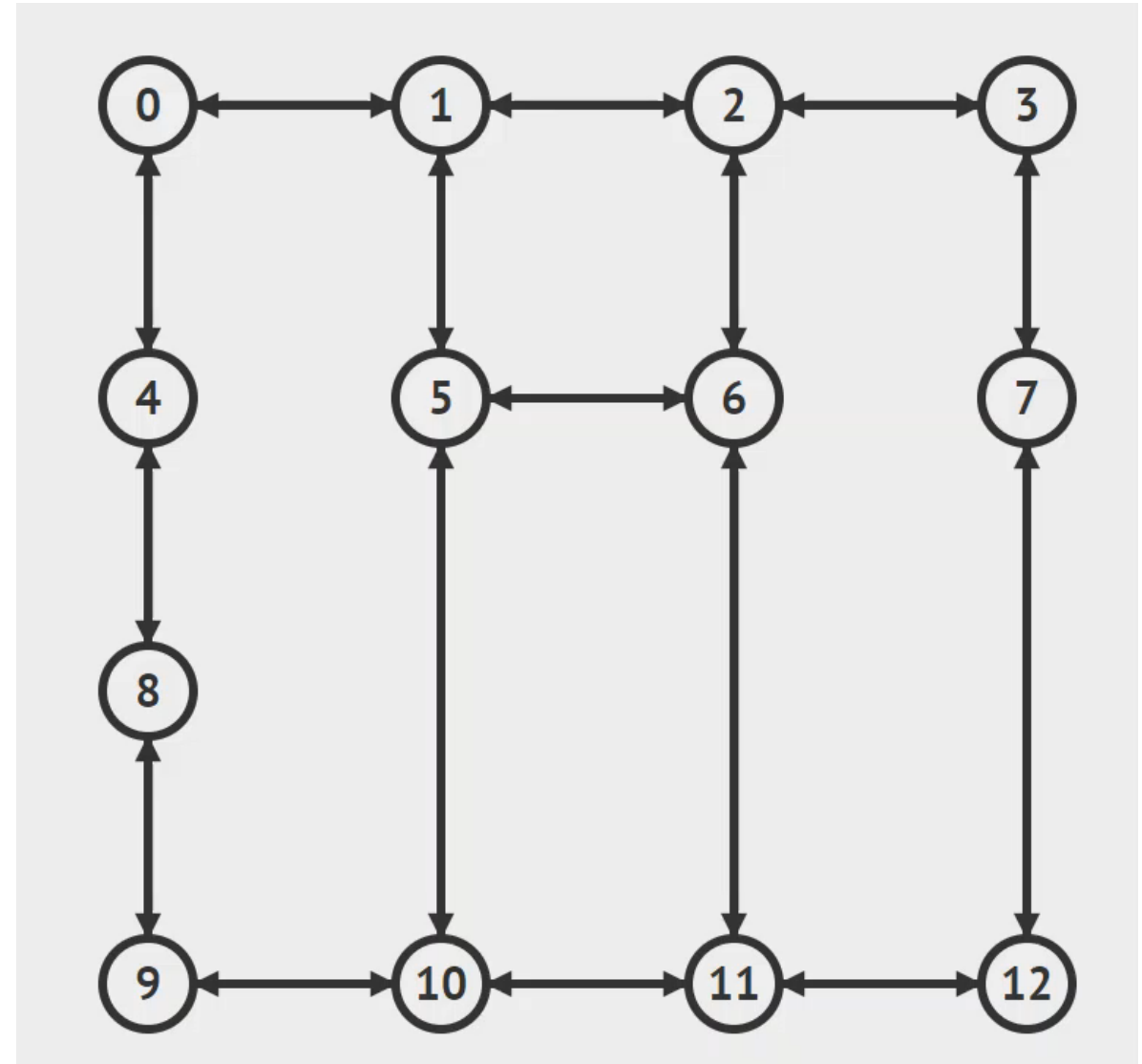
Breadth-first Search

0 1 3 4 2 6 5 7 8

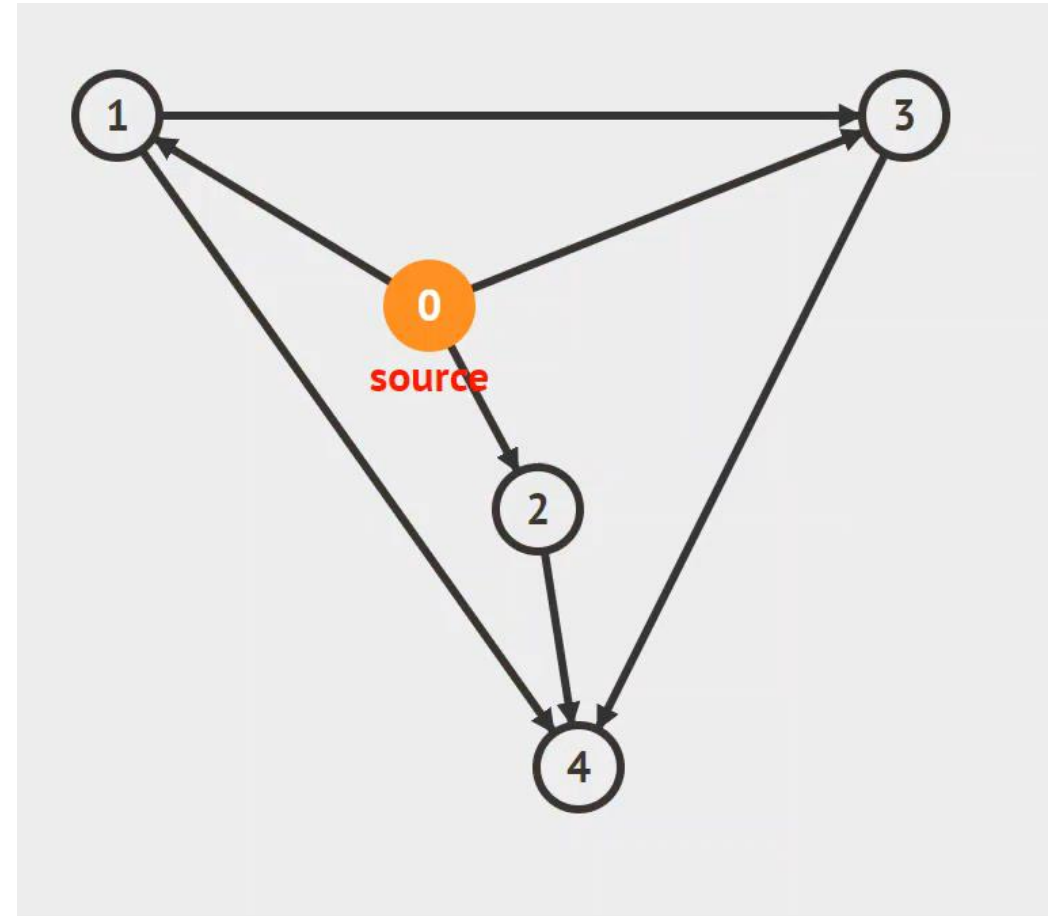


There's no unique traversal and it can be different based on the order of the successors.

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.

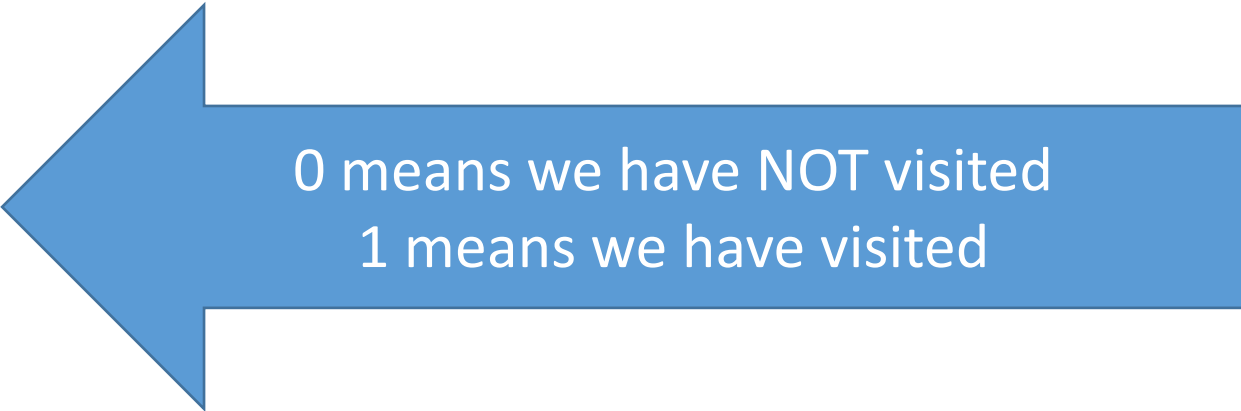
Peer to Peer Networking

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

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;
```



0 means we have NOT visited
1 means we have visited

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;  
    }  
}
```

Breadth-first Search

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



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

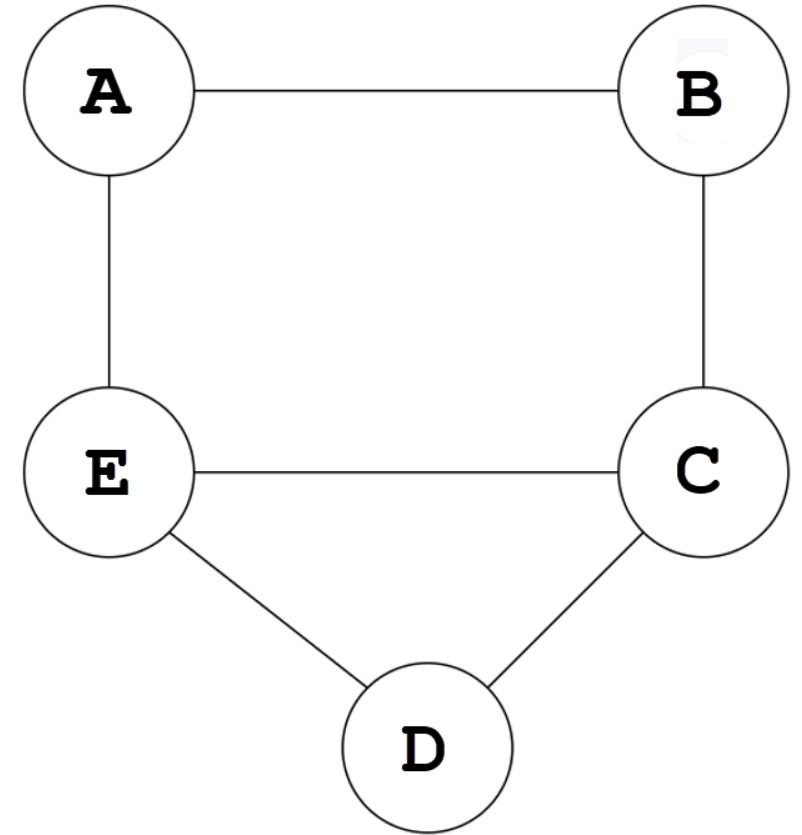
```
void addVertex(char label, Vertex *VertexArray[], int *VertexCount)
{
    Vertex *NewVertex = malloc(sizeof(Vertex));
    NewVertex->label = label;
    NewVertex->visited = 0;
    VertexArray[(*VertexCount)++] = NewVertex;
}
```

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

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 BFG.c -D UNDIRECTED
```

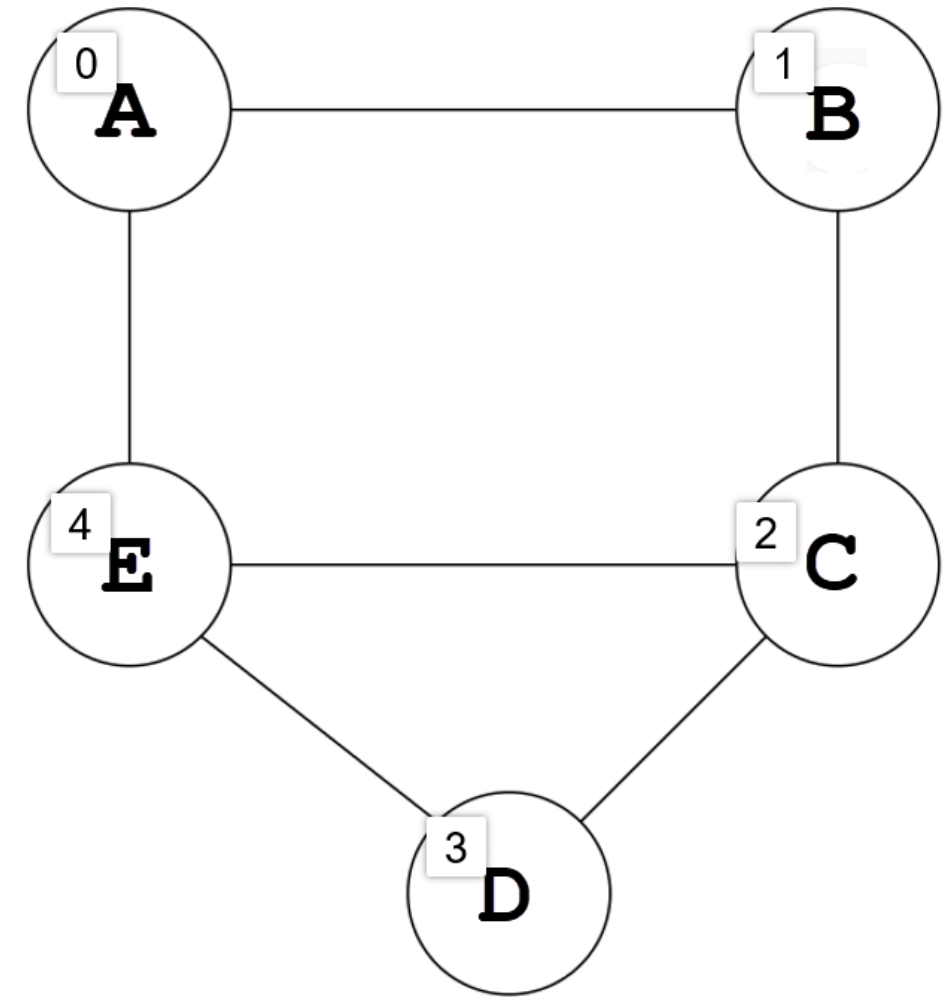
so that this code

```
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

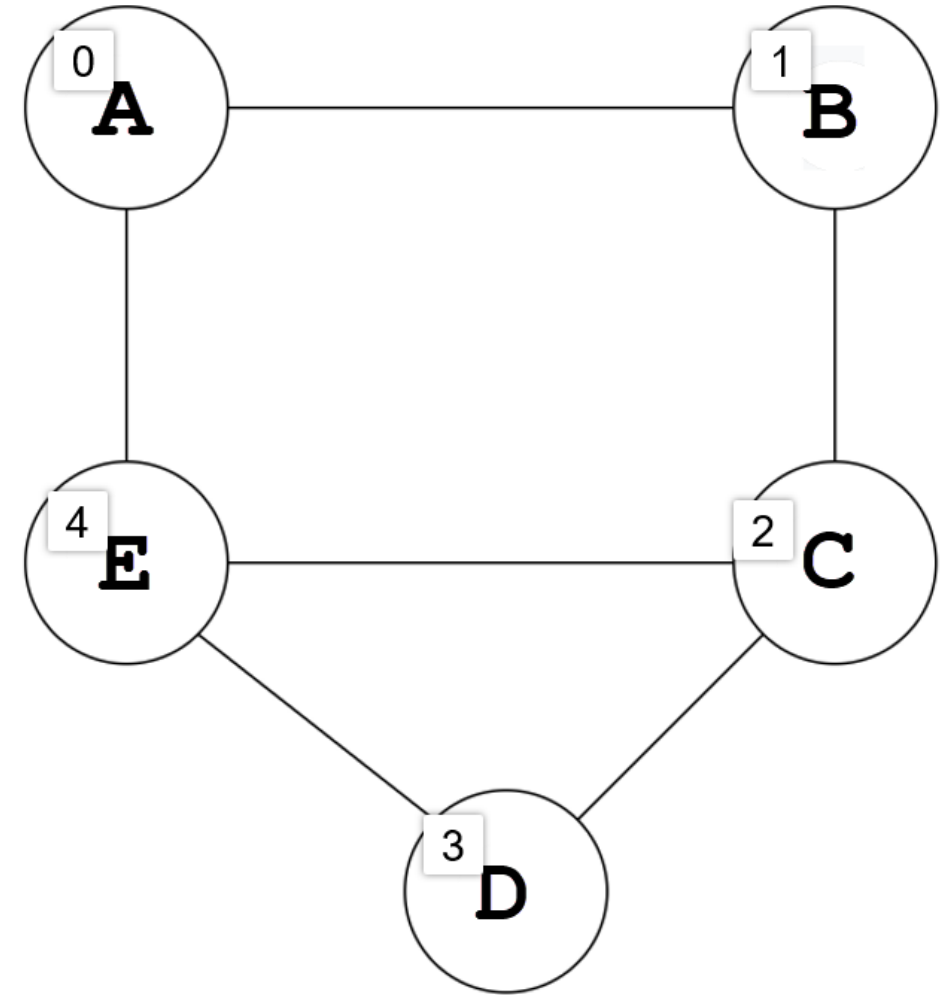
Now we call `addEdge()` for every edge in our graph.

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

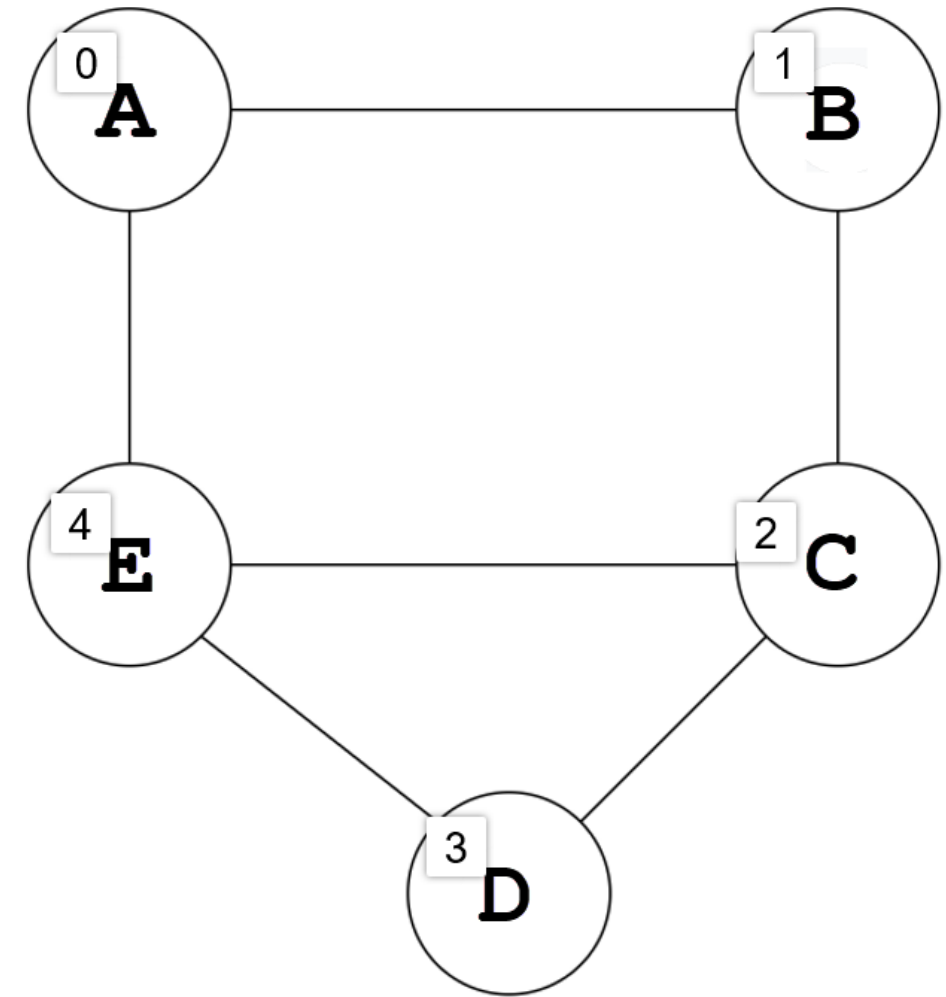
```
addEdge(0, 1, AdjMatrix);  
addEdge(1, 2, AdjMatrix);  
addEdge(2, 4, AdjMatrix);  
addEdge(4, 0, AdjMatrix);  
addEdge(2, 3, AdjMatrix);  
addEdge(3, 4, AdjMatrix);
```



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



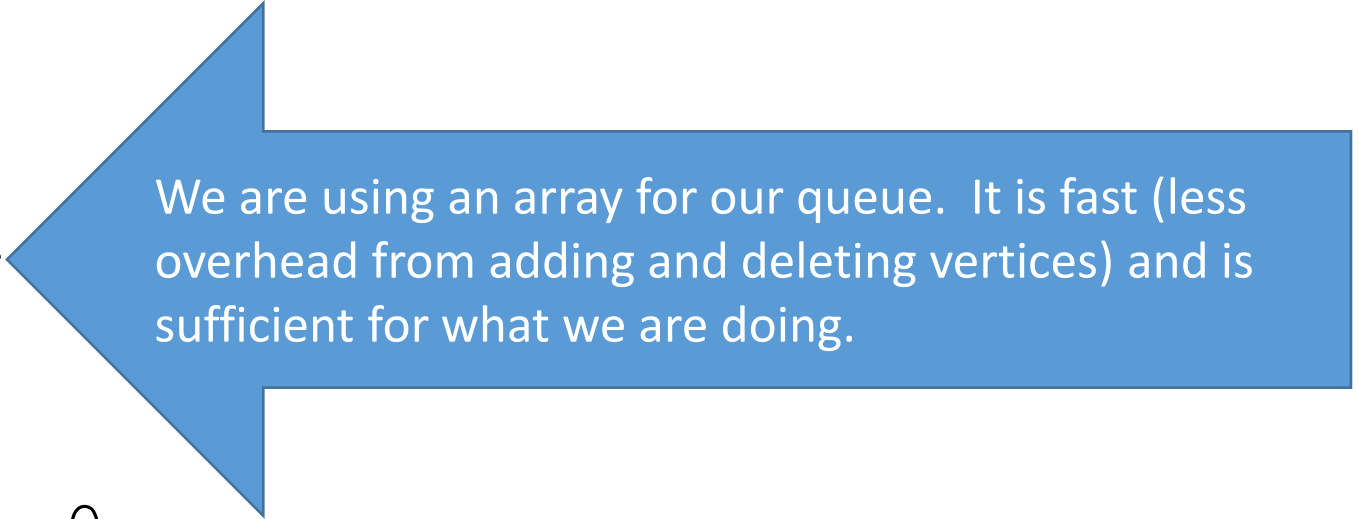
Breadth-first Search

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

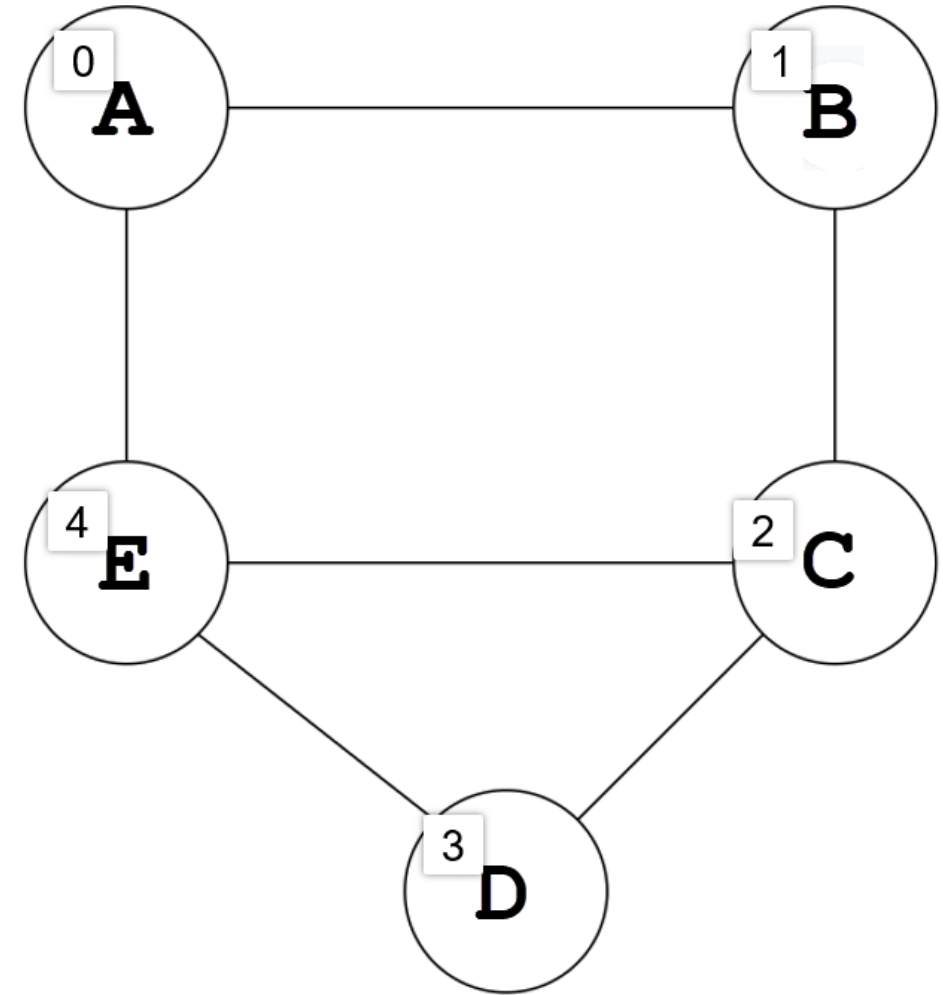
Why?

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

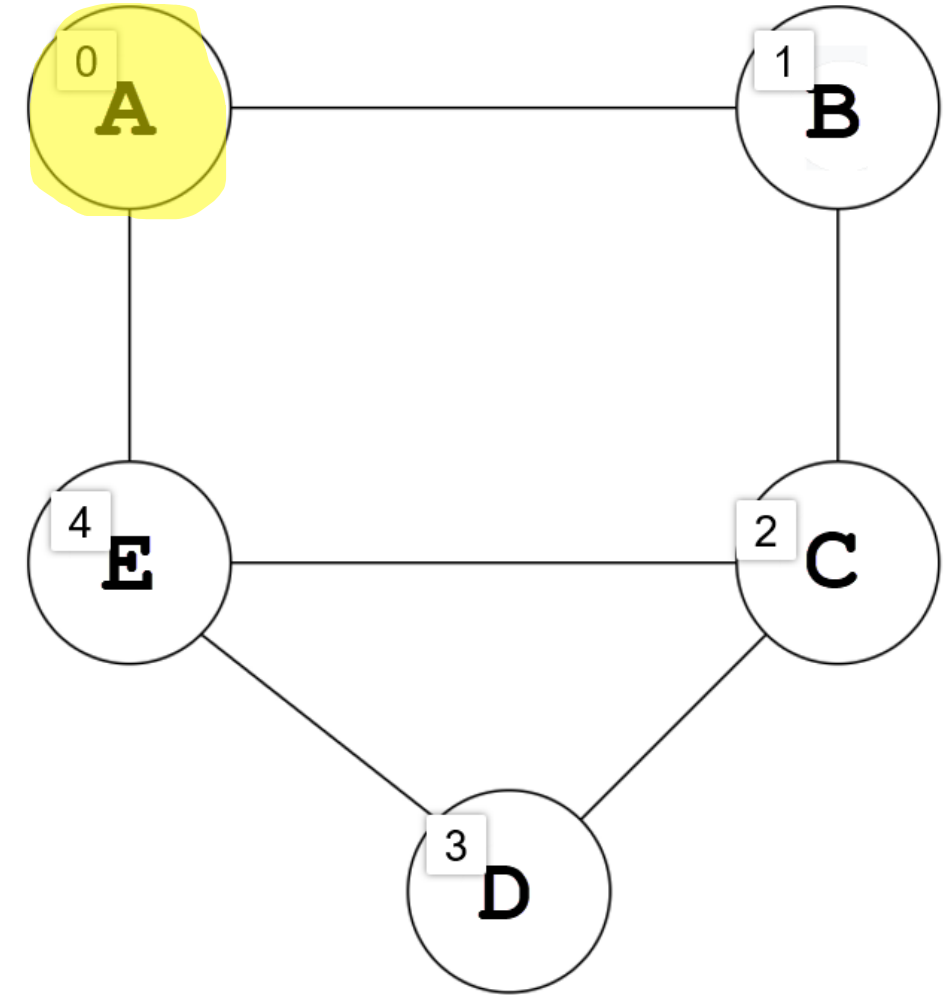
Our traversal will start at Vertex A

```
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.



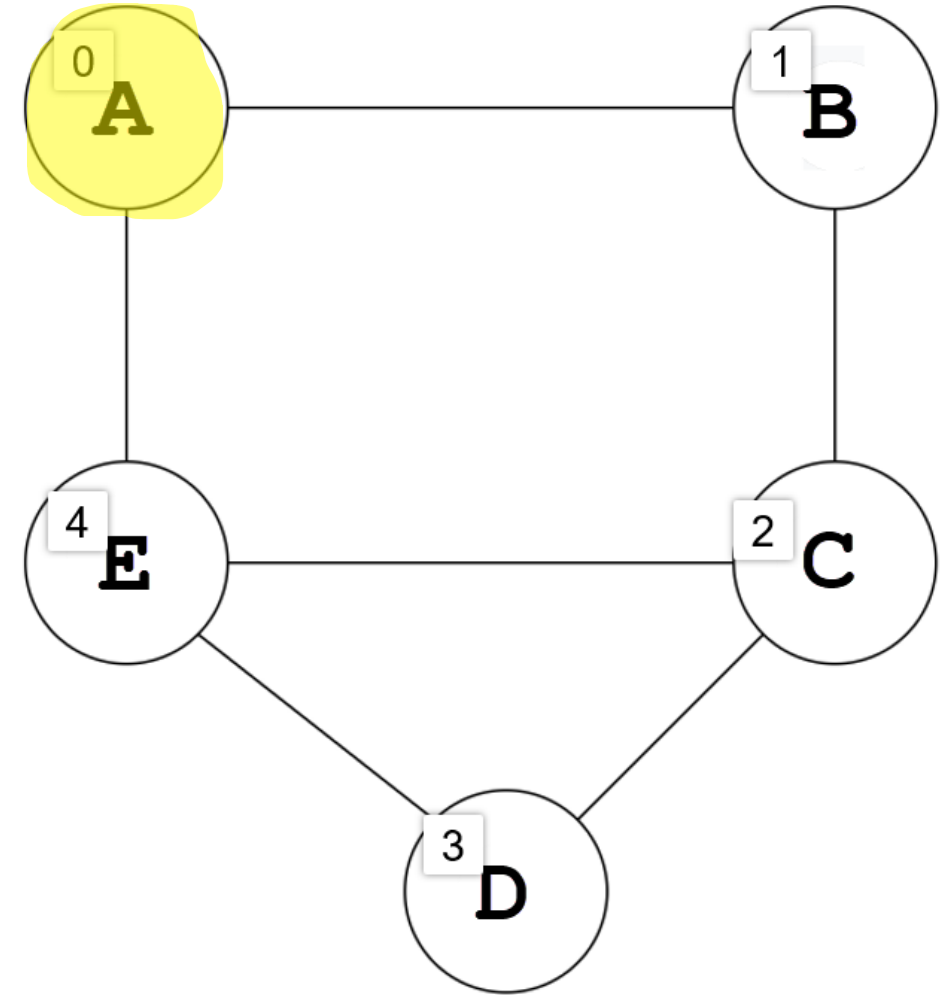
Breadth-first Search

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.

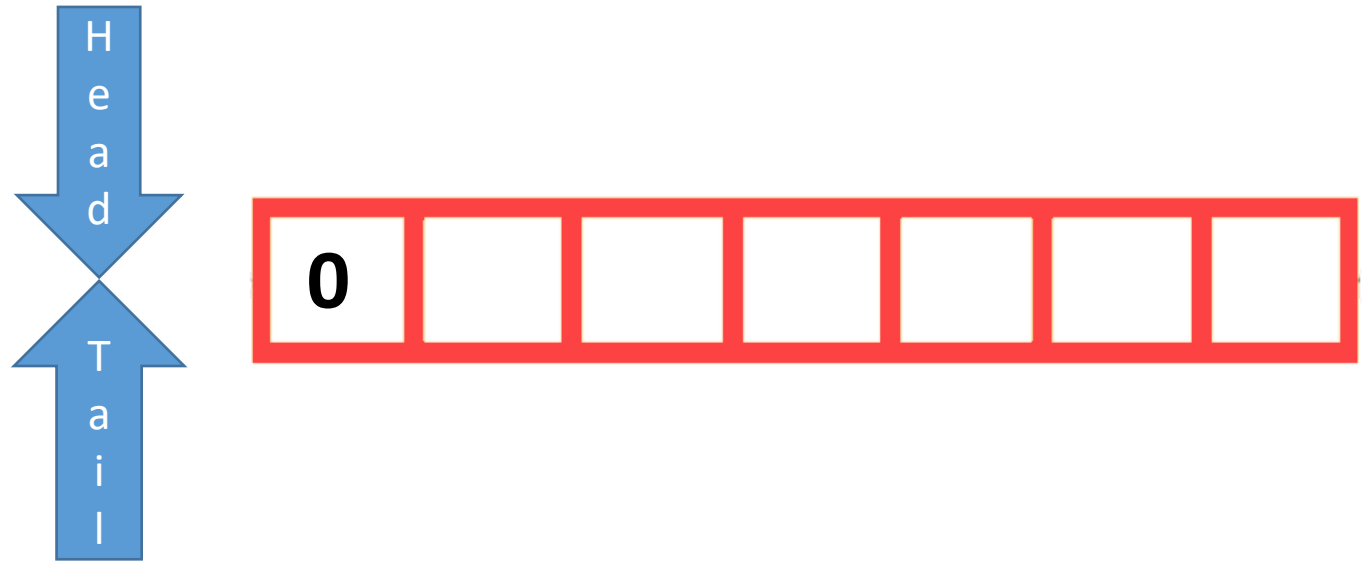


Breadth-first Search

```
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;
}
```



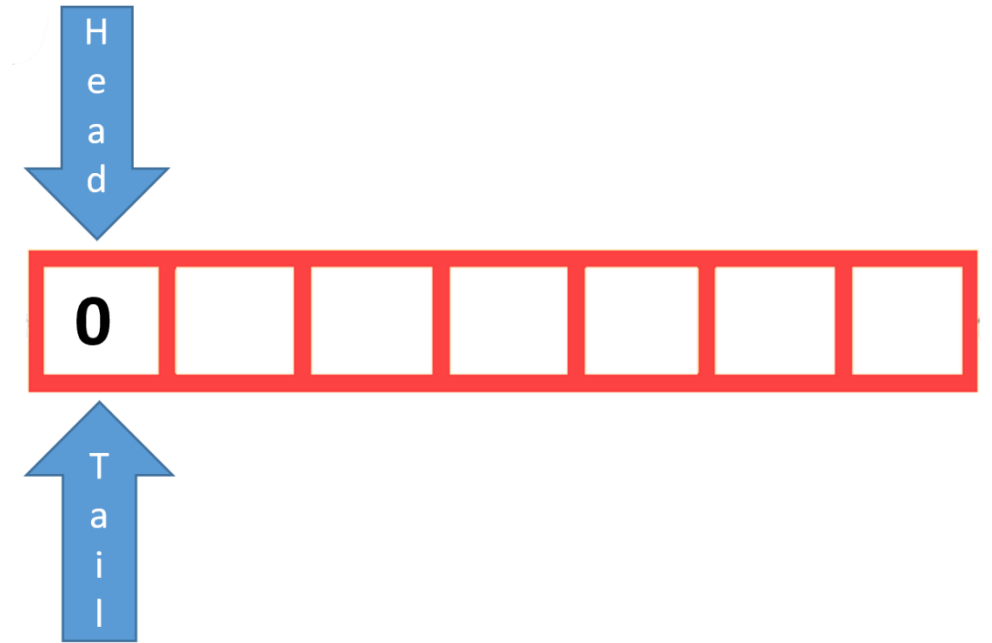
Breadth-first Search

```
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.



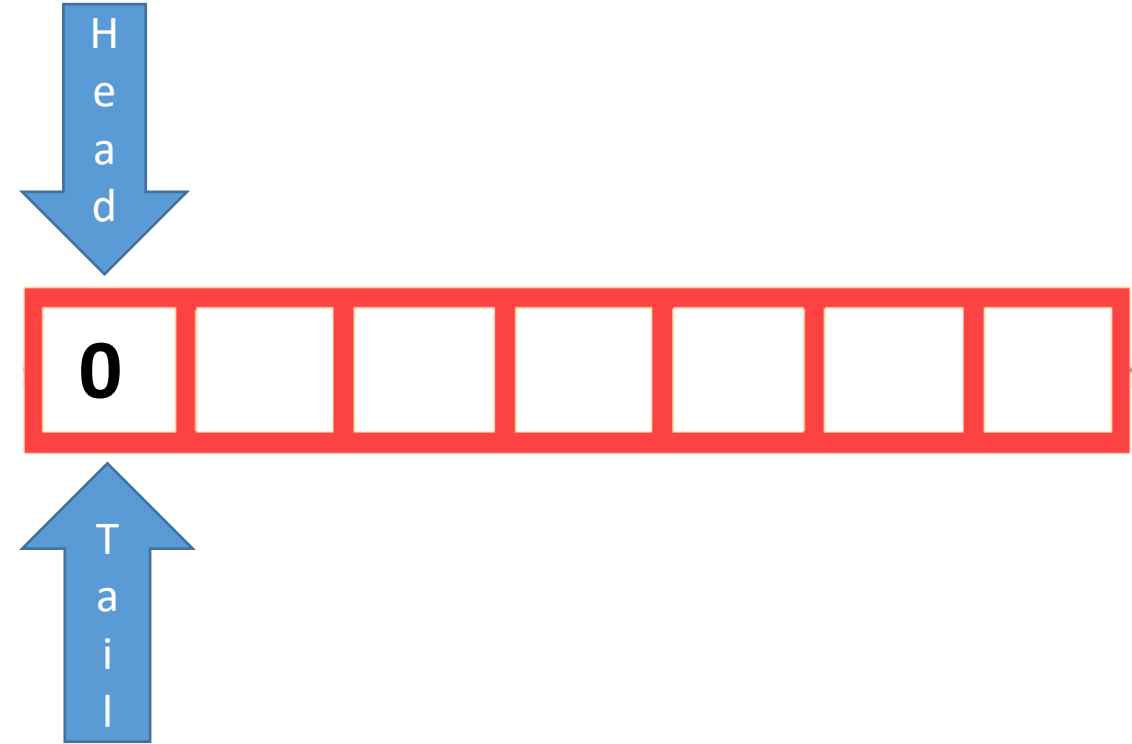
This is the vertex we will use to check for adjacent/neighbor vertices.

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;
}
```



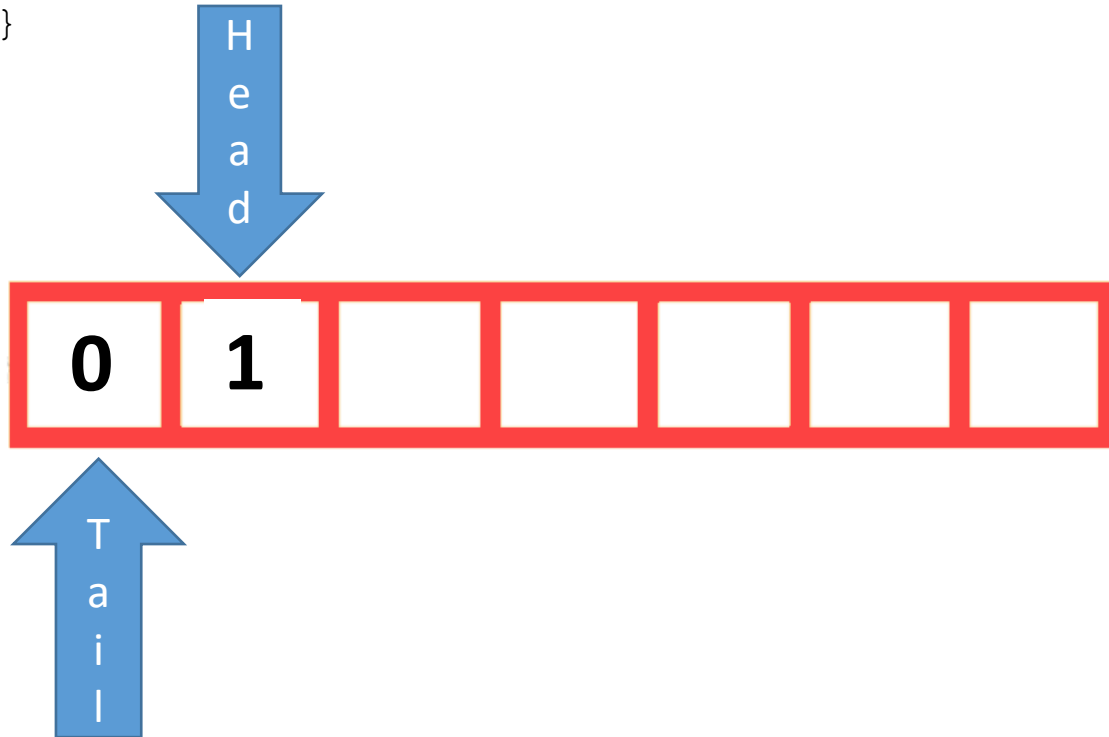
Breadth-first Search

```
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;
        }
    }
}
```

```

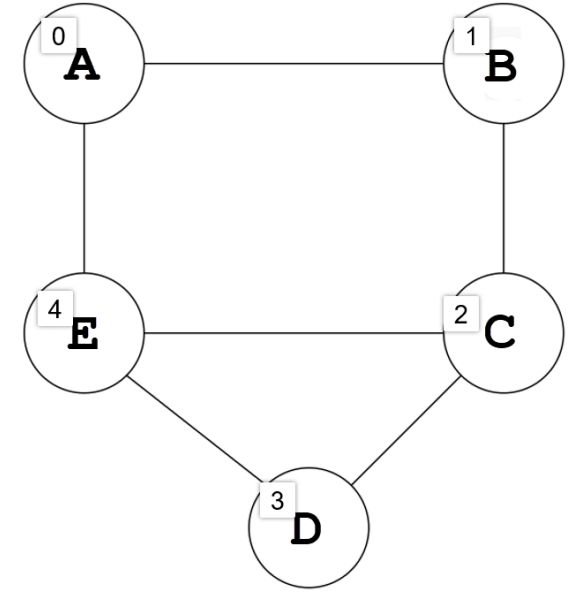
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}

```



Breadth-first Search

	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

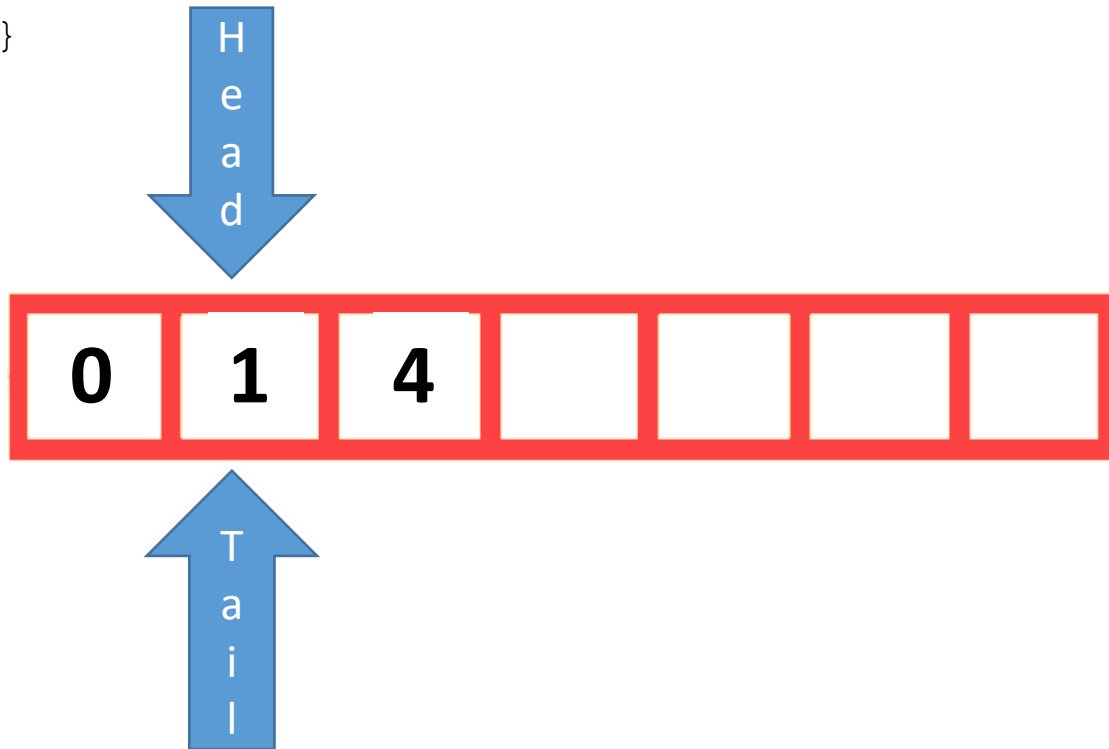


	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	0	0	0	0

```

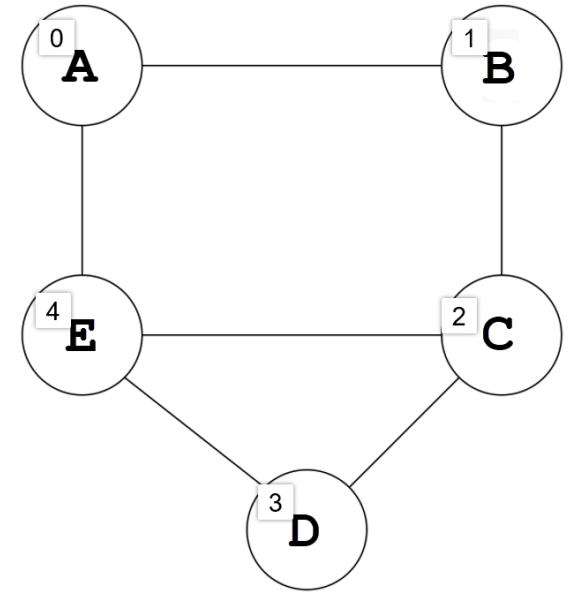
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}

```



Breadth-first Search

	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

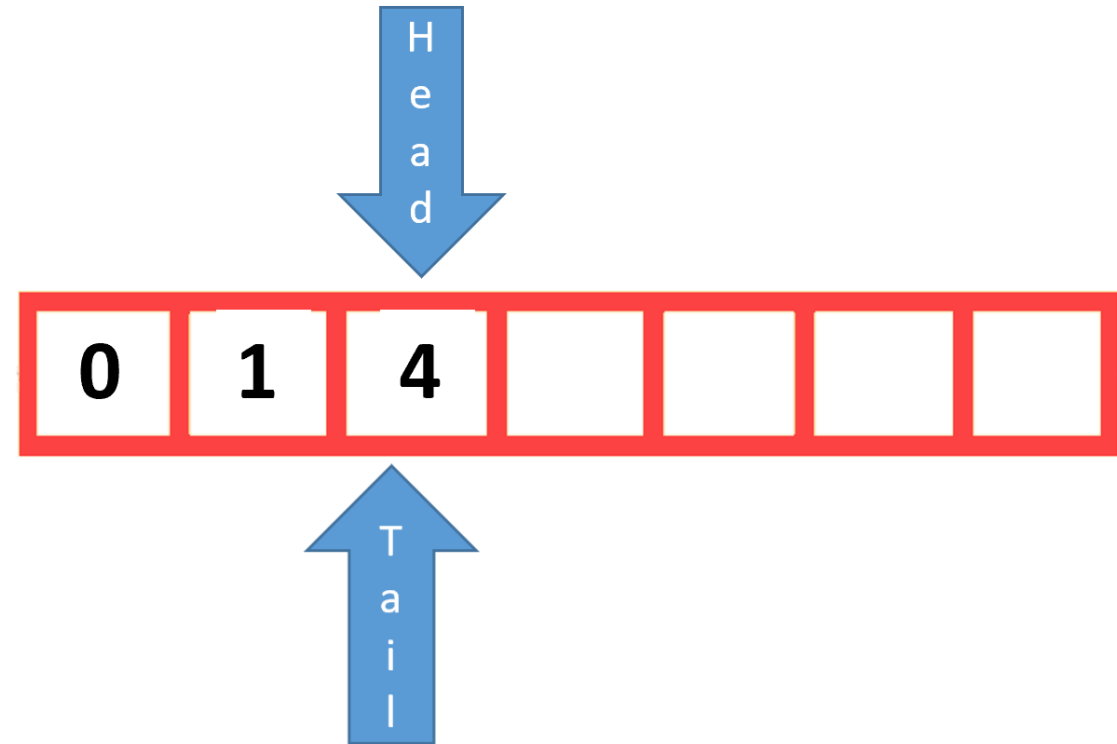


	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	0	0	0

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.

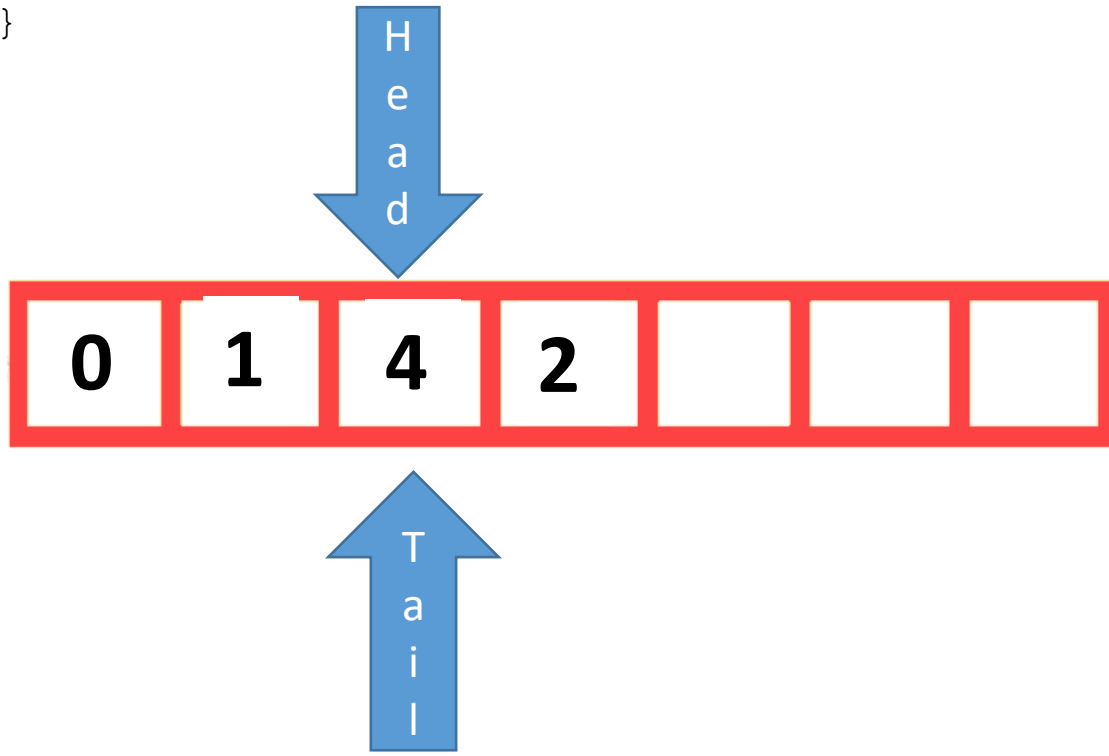


We are now going to check for neighbors of the vertex at index 1 (Vertex B).

```

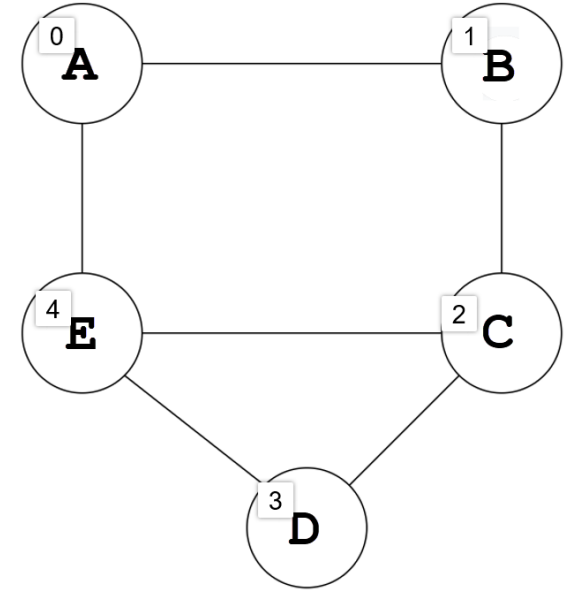
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}

```



Breadth-first Search

	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

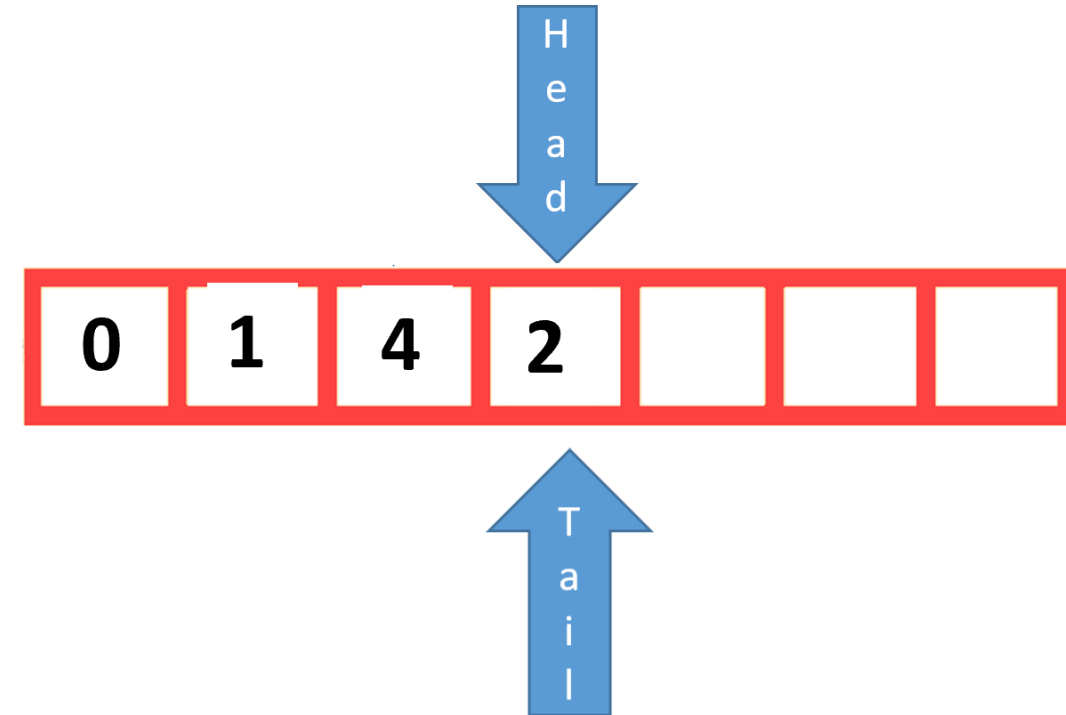


	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	0	0	1

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.

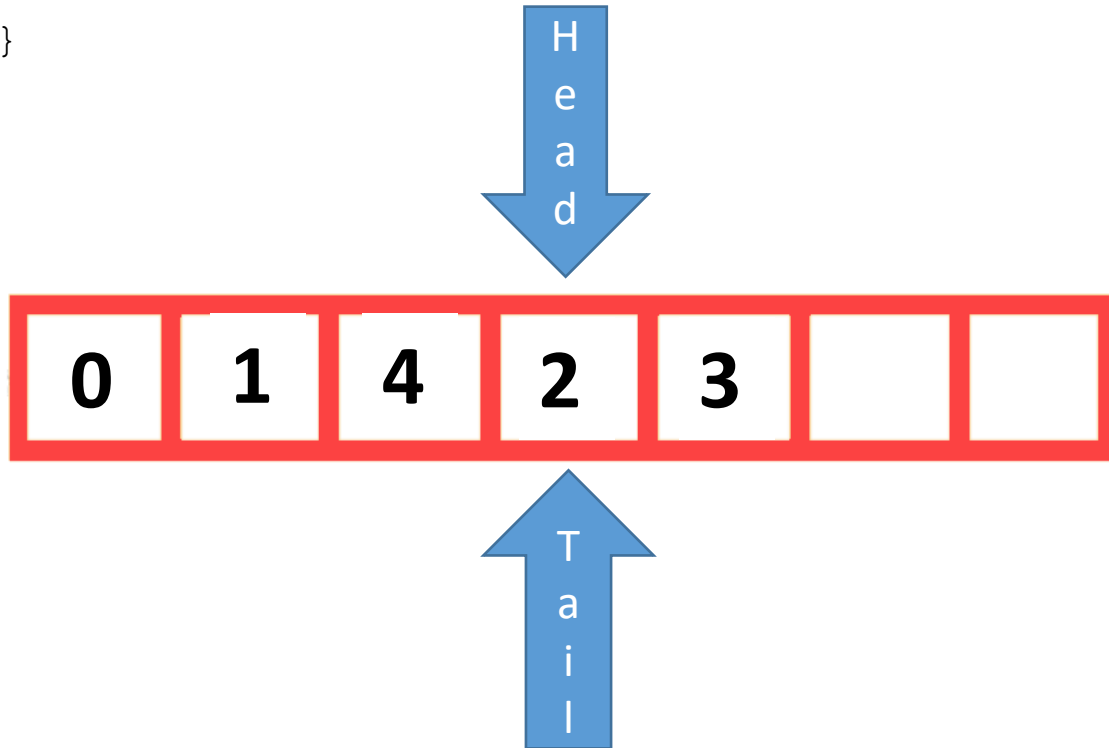


We are now going to check for neighbors of the vertex at index 4 (Vertex E).

```

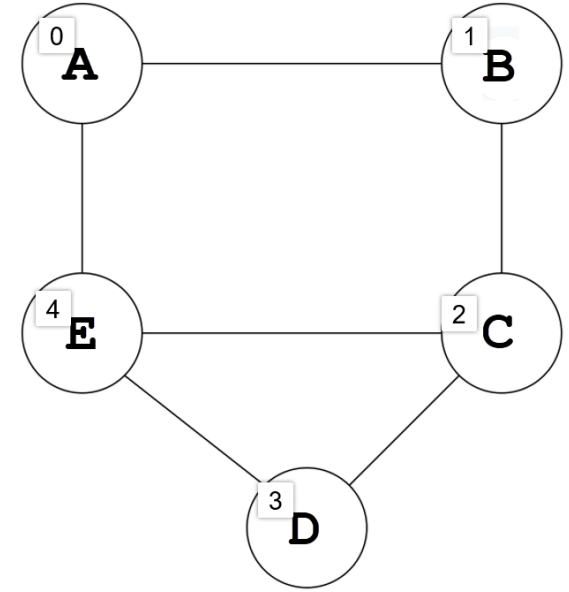
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}

```



Breadth-first Search

	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

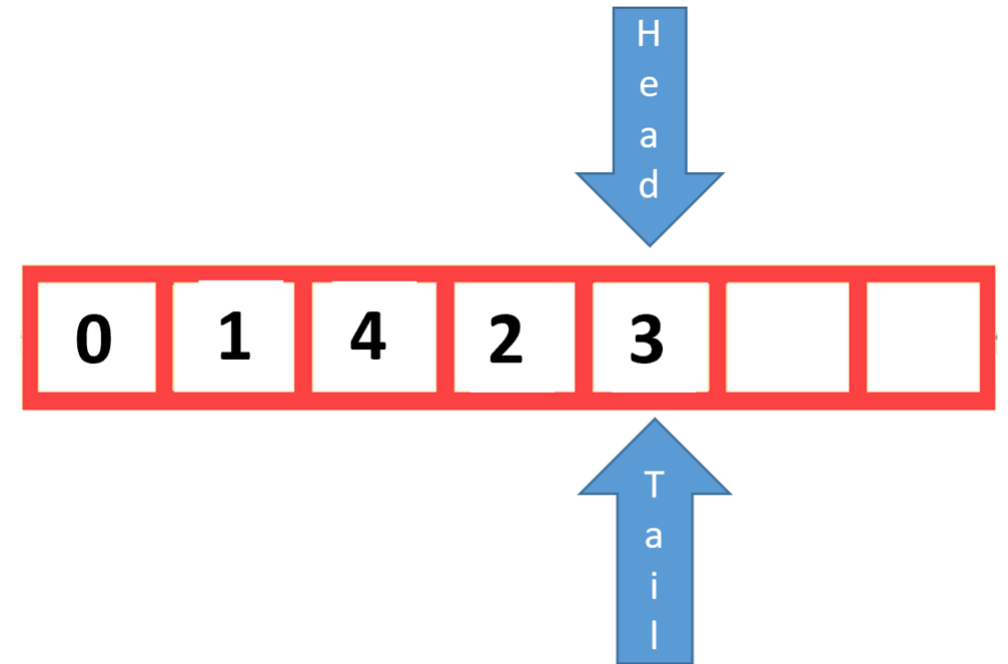


	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	0	1

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.

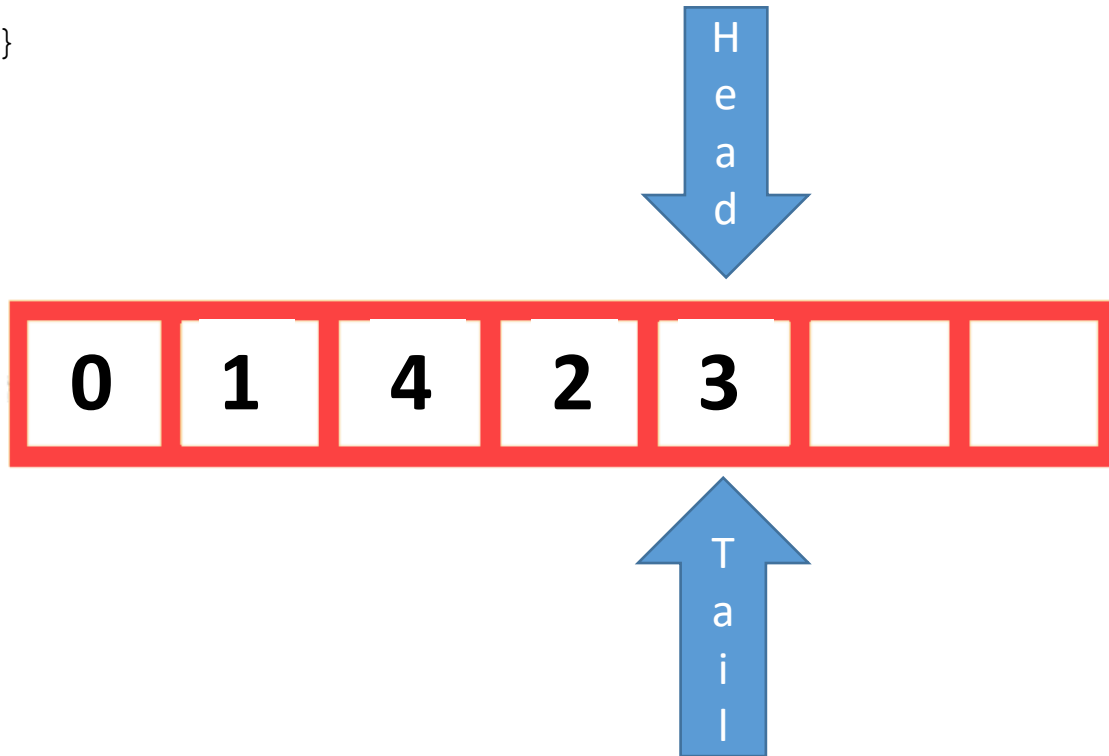


We are now going to check for neighbors of the vertex at index 2 (Vertex C).

```

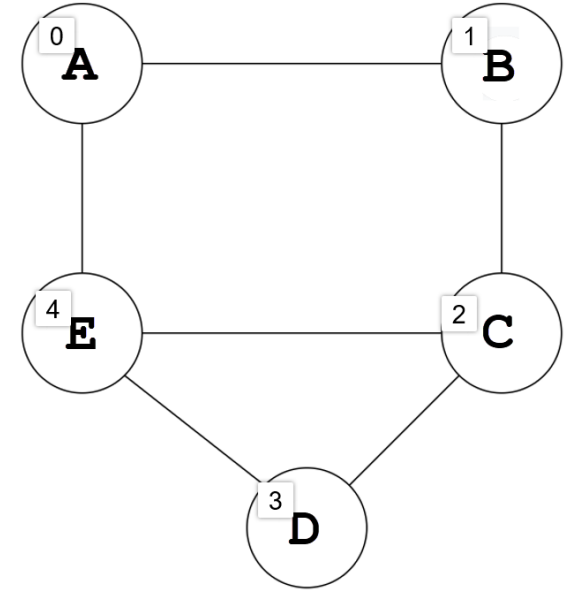
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}

```



Breadth-first Search

	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

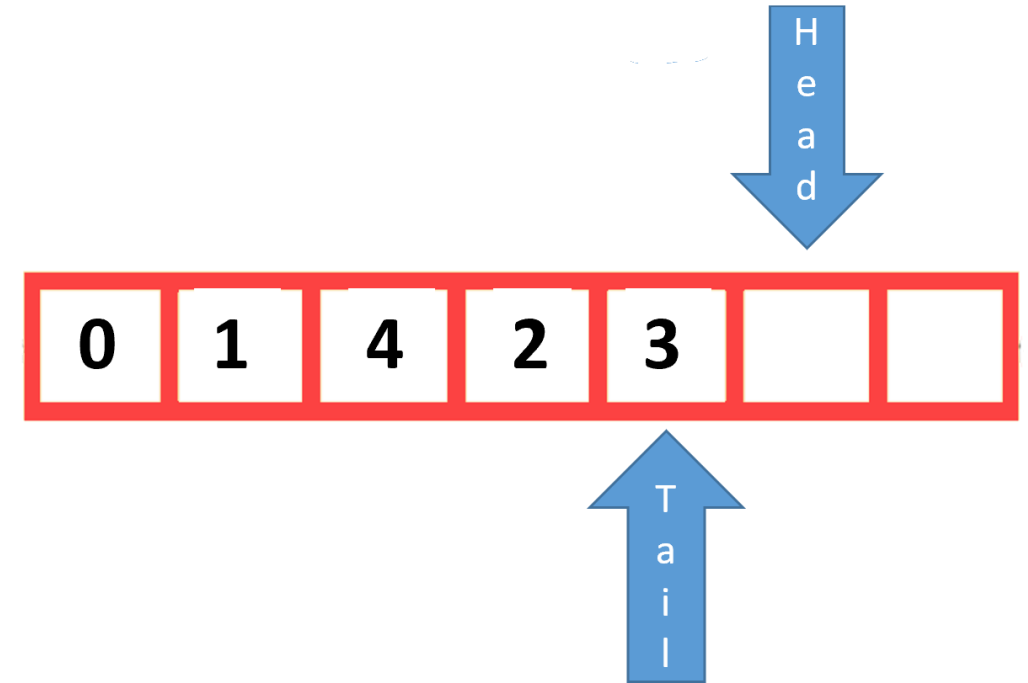


	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	1	1

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.

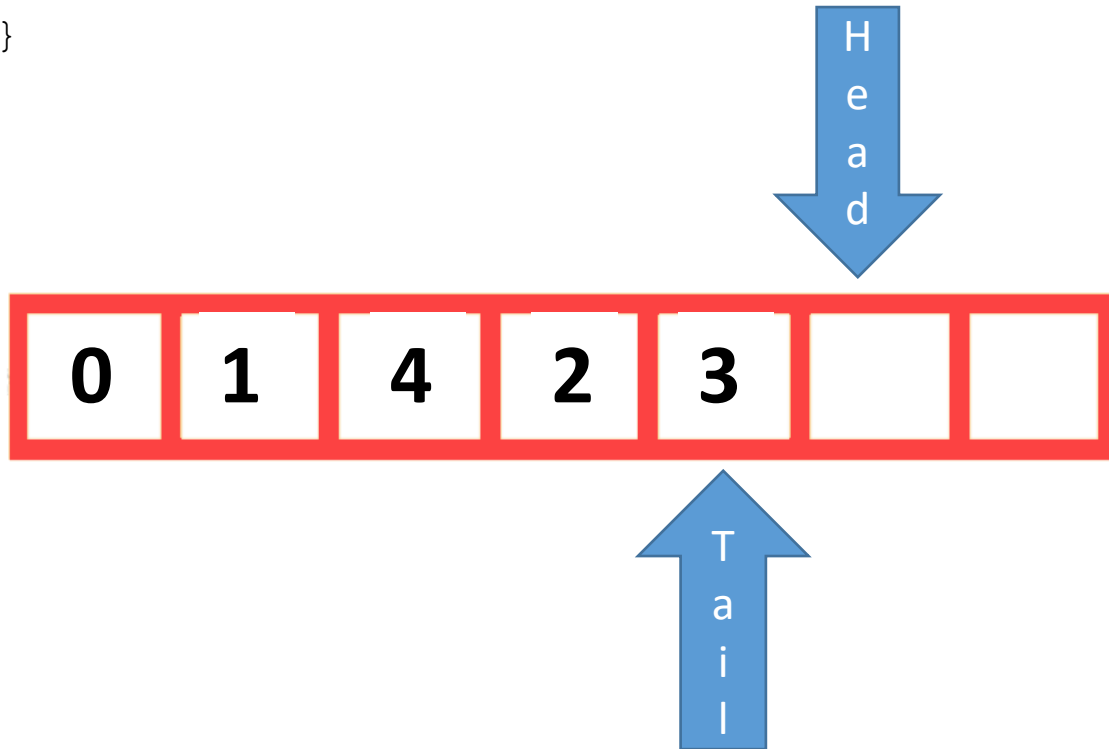


We are now going to check for neighbors of the vertex at index 3 (Vertex D).

```

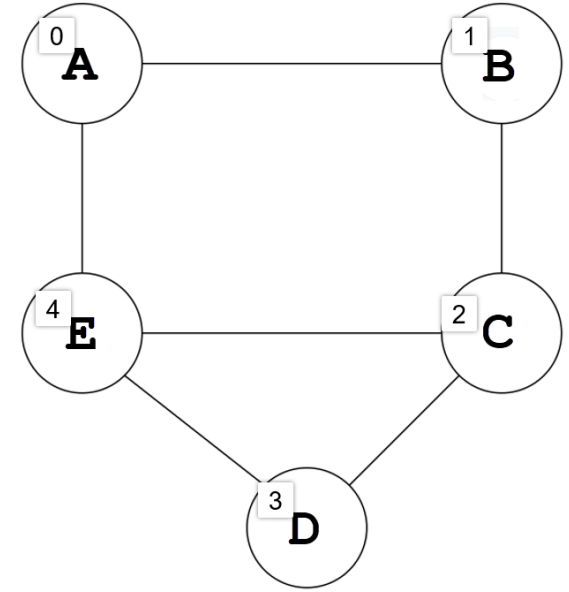
for (i = 0; i < VertexCount; i++)
{
    if (AdjMatrix[CurrentVertexIndex][i] == 1)
    {
        if (VertexArray[i]->visited == 0)
        {
            enqueue(queue, &head, &tail, i);
            queueItemCount++;
            VertexArray[i]->visited = 1;
        }
    }
}

```



Breadth-first Search

	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



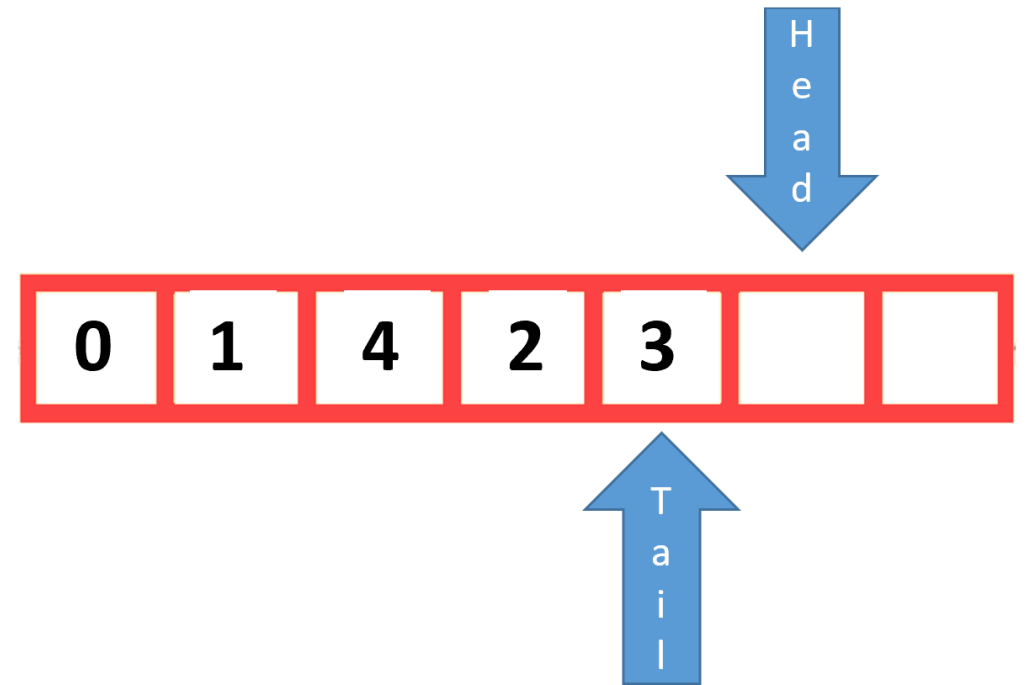
	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	1	1

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

