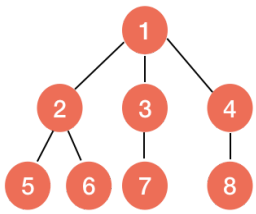
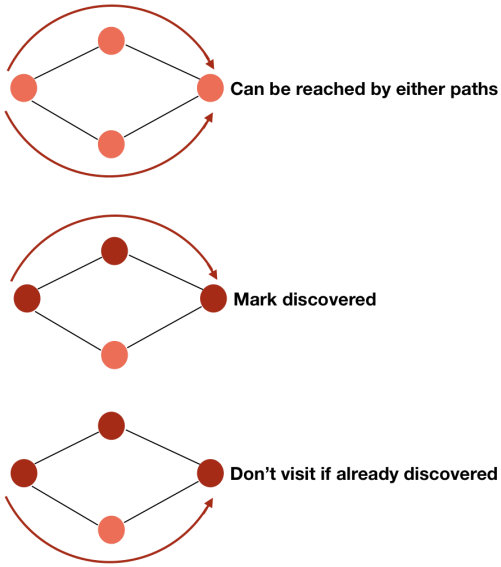
Breadth First Search

Breadth-first search or BFS is a searching technique for graphs in which we first visit all the nodes at the same depth first and then proceed visiting nodes at a deeper depth. For example, in the graph given below, we would first visit the node 1, and then after visiting the nodes 2, 3 and 4, we can proceed to visit any deeper node i.e., nodes 5, 6, 7 and 8.



Let's look at the animation given below to see the working of BFS.

A node can be reached from different nodes using different paths but we need to visit each node only once. So, we mark each node differently into 3 categories - unvisited, discovered and complete.



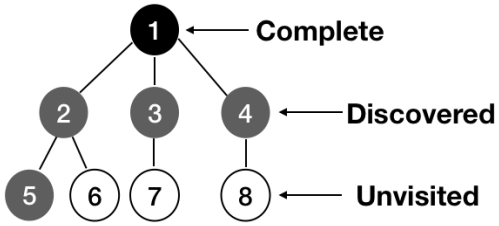
Initially, all nodes are unvisited. After visiting a node for the first time, it becomes discovered.

A node is complete if all of its adjacent nodes have been visited.

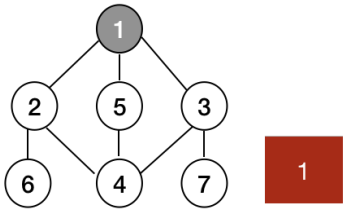
Thus, all the adjacent nodes of a complete node are either discovered or complete.

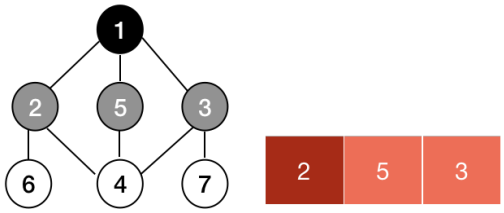
</>
The terms unvisited, discovered and complete are not standard ones. We are using them only to explain the concept in this context.

Generally, three different colors i.e., white, gray and black are used to represent unvisited, discovered and complete respectively.

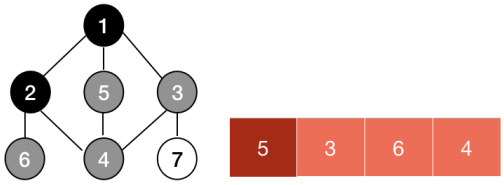


Thus after visiting a node, we first visit all its sibling and then their children. We can use a first in first out (FIFO) queue to achieve this. Starting from the source, we can put all its adjacent nodes in a queue.

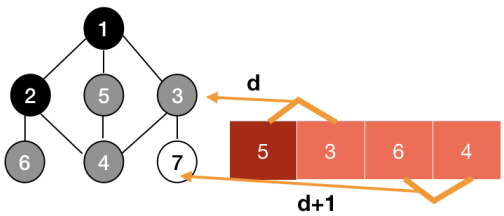




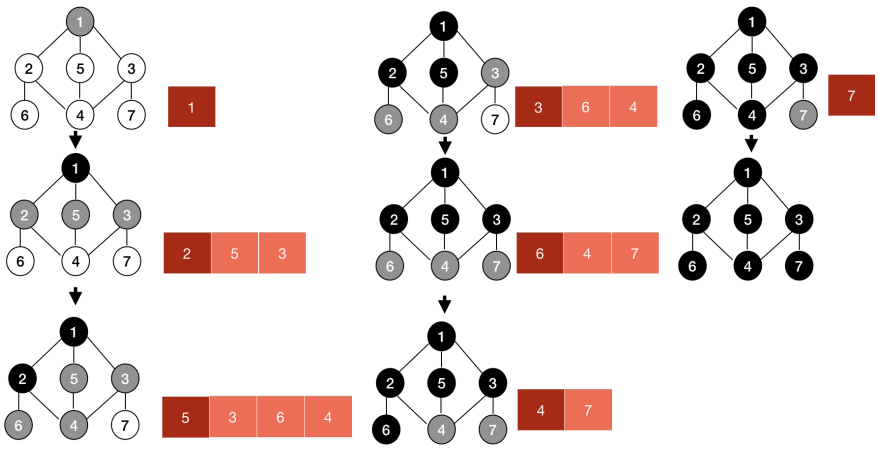
Now, we will visit the first element of the queue and put all its adjacent nodes in the queue.

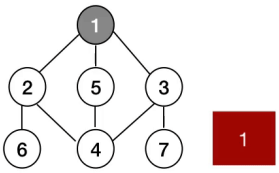


We can see that the queue has all the nodes at the same distance first (say d) and then other nodes at the distance (d+1).



In this way, we can visit the nodes at a smaller distance from the source first before proceeding for the nodes at larger distances and thus, can accomplish BFS.





Code for BFS

We are going to use queue (https://www.codesdope.com/blog/tag/queue/?tag=queue) for the implementation of the BFS.

Our function is going to take the graph (G) and the source node (s) as its input - `BFS(G, s)`.

Our next task is to make all of its nodes unvisited i.e., white. As discussed in the previous chapter, the adjacency-list representation of the graph has list of all its vertices (G.V), so we will iterate over it to make all its element white.

```
for i in G.V
   i.color = white
```

We are going to start from the source, so let's make the source node discovered i.e., gray - `s.color = gray`.

Now, we need to initialize the queue. After discovering a new node, we will enqueue all of its adjacent nodes into this queue. We will dequeue this queue to start visiting a new node.

So, we will start by putting the source node in the queue first.

```
queue.enqueue(s)
```

We will iterate until the queue becomes empty and dequeue the node to get a node to work with.

```
while !q.is_empty
   u = queue.dequeue()
```

Now, we need to enqueue all of the adjacent nodes of the node we got by dequeuing the queue.

We know that we store all the adjacent nodes of a node in an array 'Adj'. So, Adj[u] is the list of all the adjacent nodes of the node u.

```
while !q.is_empty
   u = queue.dequeue()
   for i in Adj[u]
      if i.color == white // only if the node is not yet discovered
         i.color = gray //mark it discovered
         queue.enqueue(i)
   u.color = black // all adjacent nodes of this node are discovered
```

```
BFS(G, s)
  for i in G.V
    i.color = white
  s.color = gray

  queue.enqueue(s)

  while !q.is_empty
    u = queue.dequeue()
    for i in Adj[u]
      if i.color == white // only if the node is not yet discovered
        i.color = gray // mark it discovered
        queue.enqueue(i)
    u.color = black // all adjacent nodes of this node are discovered
```

C Python Java

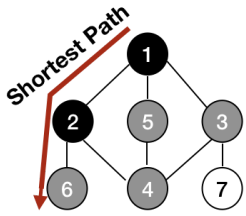
```
#include <stdio.h>
#include <stdlib.h>
enum color{White, Gray, Black};

/*
  Node for linked list of adjacent elements.
  This will contain a pointer for next node.
  It will not contain the real element but the index of
  element of the array containing all the vertices V.
*/
typedef struct list_node {
  int index_of_item;
  struct list_node *next;
}list_node;

/*
  Node to store the real element.
  Contain data and pointer to the
  first element (head) of the adjacency list.
*/
typedef struct node {
  int data;
  enum color colr;
  list_node *head;
}node;

/*
  Graph will contain number of vertices and
  an array containing all the nodes (V).
*/
typedef struct graph{
  int number_of_vertices;
  node heads[]; // array of nodes to store the list of first nodes of each adjacency list
}graph;

node* new_node(int data) {
  node *z;
  z = malloc(sizeof(node));
  z->data = data;
  z->head = NULL;
  z->colr = White;

  return z;
}

list_node* new_list_node(int item_index) {
  list_node *z;
  z = malloc(sizeof(list_node));
  z->index_of_item = item_index;
  z->next = NULL;

  return z;
}

// make a new graph
graph* new_graph(int number_of_vertices) {
  //number_of_vertices*sizeof(node) is the size of the array heads
  graph *g = malloc(sizeof(graph) + (number_of_vertices*sizeof(node)));
  g->number_of_vertices = number_of_vertices;

  //making elements of all head null i.e.,
  //their data -1 and next null
  int i;
  for(i=0; i<number_of_vertices; i++) {
    node *z = new_node(-1); //*z is pointer of node. z stores address of node
    g->heads[i] = *z; //*z is the value at the address z
  }

  return g;
}

// function to add new node to graph
void add_node_to_graph(graph *g, int data) {
  // creating a new node;
  node *z = new_node(data);
  //this node will be added into the heads array of the graph g
  int i;
```

```
  for(i=0; i<g->number_of_vertices; i++) {
    // we will add node when the data in the node is -1
    if (g->heads[i].data < 0) {
      g->heads[i] = *z; //*z is the value at the address z
      break; //node is added
    }
  }
}

// function to check of the node is in the head array of graph or not
int in_graph_head_list(graph *g, int data) {
  int i;
  for(i=0; i<g->number_of_vertices; i++) {
    if(g->heads[i].data == data)
      return 1;
  }
  return 0;
}

// function to add edge
void add_edge(graph *g, int source, int dest) {
  //if source or edge is not in the graph, add it
  if(!in_graph_head_list(g, source)) {
    add_node_to_graph(g, source);
  }
  if(!in_graph_head_list(g, dest)) {
    add_node_to_graph(g, dest);
  }

  int i,j;
  // iterating over heads array to find the source node
  for(i=0; i<g->number_of_vertices; i++) {
    if(g->heads[i].data == source) { //source node found

      int dest_index; //index of destination element in array heads
      // iterating over heads array to find node containg destination element
      for(j=0; j<g->number_of_vertices; j++) {
        if(g->heads[j].data == dest) { //destination found
          dest_index = j;
          break;
        }
      }

      list_node *n = new_list_node(dest_index); // new adjacency list node with destination index
      if (g->heads[i].head == NULL) { // no head, first element in adjaceny list
        g->heads[i].head = n;
      }
      else { // there is head which is pointer by the node in the head array
        list_node *temp;
        temp = g->heads[i].head;

        // iterating over adjaceny list to insert new list_node at last
        while(temp->next != NULL) {
          temp = temp->next;
        }
        temp->next = n;
      }
      break;
    }
  }
}

void print_graph(graph *g) {
  int i;
  for(i=0; i<g->number_of_vertices; i++) {
    list_node *temp;
    temp = g->heads[i].head;
    printf("%d\t",g->heads[i].data);
    while(temp != NULL) {
      printf("%d\t",g->heads[temp->index_of_item].data);
      temp = temp->next;
    }
    printf("\n");
  }
}

typedef struct queue_node {
  node *n;
```

```
  struct queue_node *next;
}queue_node;

struct queue
{
    int count;
    queue_node *front;
    queue_node *rear;
};
typedef struct queue queue;

int is_empty_queue(queue *q)
{
  return !(q->count);
}

void enqueue(queue *q, node *n)
{
  queue_node *new_queue_node;
  new_queue_node = malloc(sizeof(queue_node));
  new_queue_node->n = n;
  new_queue_node->next = NULL;
  if(!is_empty_queue(q))
  {
    q->rear->next = new_queue_node;
    q->rear = new_queue_node;
  }
  else
  {
    q->front = q->rear = new_queue_node;
  }
  q->count++;
}

queue_node* dequeue(queue *q)
{
  queue_node *tmp;
  tmp = q->front;
  q->front = q->front->next;
  q->count--;
  return tmp;
}

queue* make_queue()
{
  queue *q;
  q = malloc(sizeof(queue));
  q->count = 0;
  q->front = NULL;
  q->rear = NULL;
  return q;
}

void print_queue(queue *q) {
  queue_node *tmp;
  tmp = q->front;
  while(tmp != NULL) {
    printf("%d\t",tmp->n->data);
    tmp = tmp->next;
  }
  printf("\n");
}

void bfs(graph *g) {
  //here, we are using first node as source
  node s = g->heads[0];
  int i;
  for(i=0; i<g->number_of_vertices; i++) {
    g->heads[i].colr = White;
  }
  s.colr = Gray;

  queue* q = make_queue();
  enqueue(q, &s);

  while(!is_empty_queue(q)) {
    // print_queue(q);
    queue_node *u = dequeue(q);
```

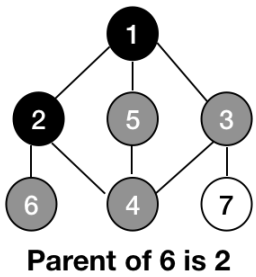
```
    list_node *temp;
    temp = u->n->head;
    while(temp != NULL) {
      if (g->heads[temp->index_of_item].colr == White) {
        g->heads[temp->index_of_item].colr = Gray;
        enqueue(q, &g->heads[temp->index_of_item]);
      }
      temp = temp->next;
    }
    u->n->colr = Black;
    printf("%d\n",u->n->data);
  }
}

int main() {
  graph *g = new_graph(7);
  add_edge(g, 1, 2);
  add_edge(g, 1, 5);
  add_edge(g, 1, 3);
  add_edge(g, 2, 6);
  add_edge(g, 2, 4);
  add_edge(g, 5, 4);
  add_edge(g, 3, 4);
  add_edge(g, 3, 7);
  bfs(g);
  return 0;
}
```

BFS has also an interesting feature that it stores the shortest path of nodes from the source from where we start our search. Thus, if we store the distance at which the nodes are first discovered then that gives us the shortest path of the corresponding nodes from a source.



We can also store a property distance along with its color for each node to accomplish this. Now, suppose we are given a node and we need to print the shortest path from the source s.

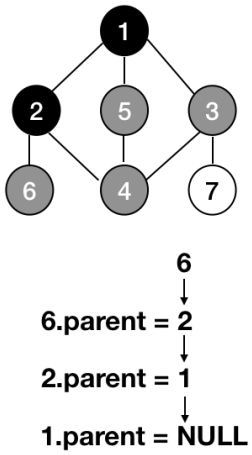
The easiest way of doing this would be to also store the parent of each node i.e., the node from which it was discovered first.



We can easily trace the parent of the node to get back to the source and find the shortest path.

```
BFS-MODIFIED(G, s)
  for i in G.V
    i.color = white
    i.distance  = infinite // initially, we don't have any path to reach
    i.parent = NULL
  s.color = gray
  s.distance = 0 // distance of source is 0
  s.parent = NULL // source has no parent

  queue.enqueue(s)

  while q not empty
    u = queue.dequeue()
    for i in Adj[u]
      if i.color == white // only if the node is not yet discovered
        i.color = gray // mark it discovered
        i.distance = u.distance+1 // distance of the adjacent node is one more than its
        i.parent = u
        queue.enqueue(i)
    u.color = black // all adjacent nodes of this node are discovered
```

Now, we can easily trace the parent of the nodes to print the shortest path.



We will just trace the parent until the source node is reached but if we reach any node whose parent is NULL, then there isn't any path from that node to the source.

```
PRINT-PATH(G, s, n)
  if n == s
    print s
  else if n == NULL
    print "NO path"
  else
    PRINT-PATH(G, s, n.parent)
    print n
```

Analysis of BFS

We have already seen that we are enqueuing and dequeuing each vertex only once and the enqueue and dequeue operations are constant time taking operations ($O(1)$), so the total time taken for enqueue and dequeue is $O(|V|)$. We are also scanning all the adjacency lists of all the node and we have discussed that they are of $\Theta(|E|)$ in number. So, this will take $O(|E|)$ time and thus, the total time for BFS will be $O(|V| + |E|)$.

Depth-first search (/course/algorithms-dfs/) is also a searching technique used with the graphs which we are going to learn in the next chapter.

> “ Books are as useful to a stupid person as a mirror is useful to a blind person. ”
>
> - Chanakya

PREV (/course/algorithms-graphs/) (/course/algorithms-dfs/) NEXT