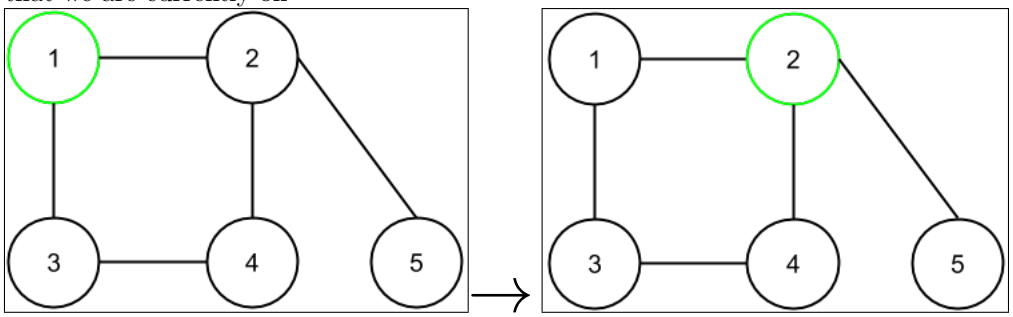
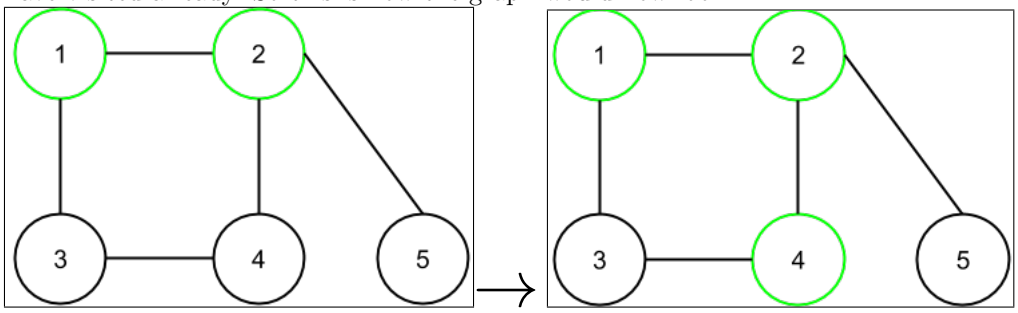
Graph Traversals

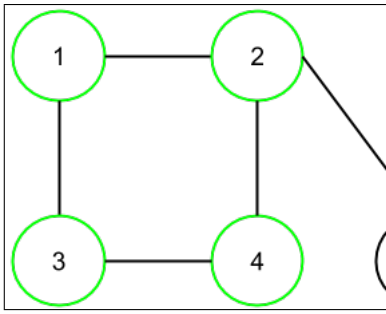
1 Depth-First Search

We will cover two broad ways of searching, ***Depth-First Search*** and ***Breadth-First Search***. Let's start off with Depth-First-Search.
Depth-First Search is a typical recursive strategy that involves exploring the subtrees of children. Let's step through the steps of Depth-First Search and gain some intuition for the process. The green node is the node that we are currently on

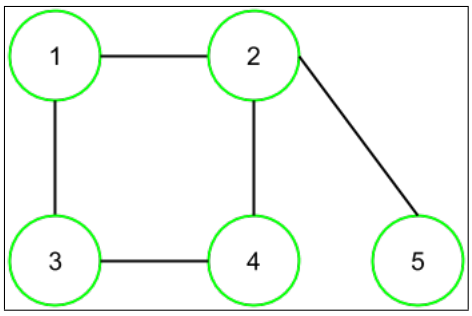


The first two steps are relatively straightforward, we start off at the node 1 and then we travel to its smallest neighbor which is 2. The problem is that smallest neighbor of 2 is 1, which is the node we just came from. We will end up having some sort of infinite loop. To solve this problem, we will keep track of nodes that we have visited already. So this is how the graph would now look.





Now that we have the ability to mark our nodes, we will travel to 2's smallest child which is 4.



These are the final steps to Depth-First Search. We go to 4's smallest neighbor which is 3. After this, we realize that there are no neighbors of 3 that are not marked as a result, we recurse to our parent node 4. There are no unmarked neighbors of 4 so we recurse to its parent, which is 2. 2 does have an unmarked neighbor, 5, so we visit it. 5 has no more neighbors so we go back to 4 and we realize it has no more kids so it goes to 1. 1 no longer has anymore neighbors that are unmarked (since 3 was marked through 4) so our process is done.

Now that an understanding of what Depth-First search is has been established, let's discuss how to properly implement this. The main data structure that will be used is a **Stack**. For our earlier graph, our stack would initially be the element $\{1\}$. After 1 is popped, we place it's children in the stack making the stack $\{2, 3\}$. Following this, 2 is popped which leads to a stack of $\{4, 5, 3\}$. Once 4 is popped our stack looks like $\{3, 5, 3\}$. When the final 3 is the only element in the stack, it is noted that we have already visited the node, since it is marked, so there is no need to visit it again.

As we mentioned earlier, if we go with a naive approach of Depth-First Search, it has the possibility of failing because you may end up getting into an infinite loop. To solve this problem, we suggested the "marking" of a node, but now we will go over how to actually implement it. A nice intuitive way of approaching this problem is having a boolean array that keeps track of if a node has been visited. All elements in the array are initially False, and they are marked as True once they are visited. The code for this traversal can look like the following:

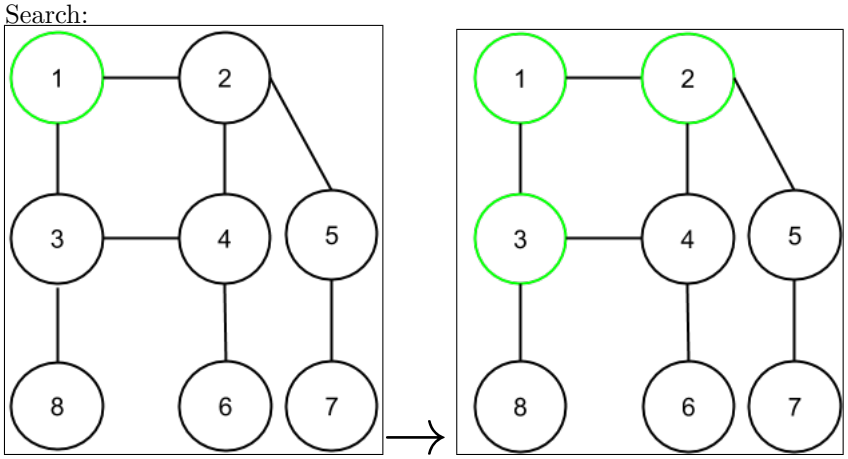
```
1  public class DepthFirstDemo{
2      private boolean[] marked;
3      private int[] edgeTo;
4      private int s;
5      public DepthFirstDemo(Graph G, int s){
6          marked = new boolean[G.V()];
7          edgeTo = new int[G.V()];
8          this.s = s;
9          dfs(G,s);
10         }
11     private void dfs(Graph G, int v) {
12         marked[v] = true;
13         for (int w : G.adj(v)) {
14         if (!marked[w]) {
15             edgeTo[w] = v;
16             dfs(G, w);
17                 }
18         }
19     }
20 }
```

This code runs in $O(V + E)$ time when represented by an Adjacency List. We can derive this by realizing that a vertex may only be visited once (V) and in worst case, you iterate over all the edges (E) which gets us to $O(V + E)$. The amount of memory that this takes up is worst case $O(V)$ or the size of the stack.

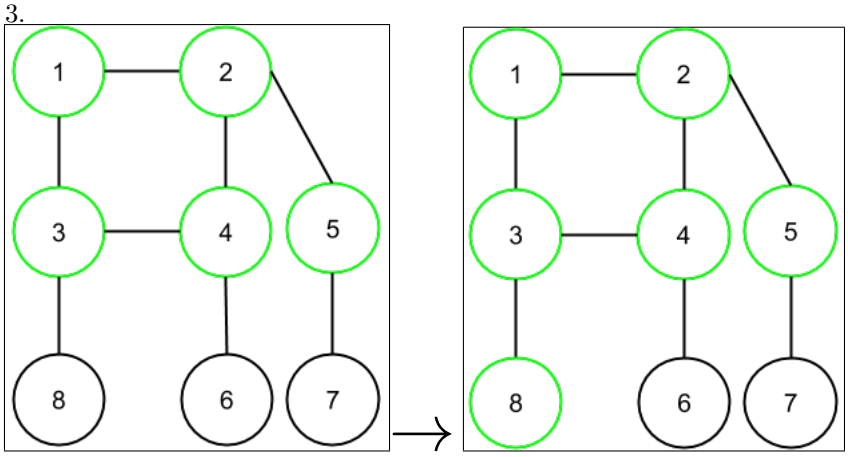
2 Breadth-First Search

To start off this section, let's consider the problem "find the shortest path between 2 vertices" and let's approach it with our current method of Depth-First Search. Running the algorithm, you can see that it fails in certain cases such as the following problem, where we are trying to find the shortest path from 1 to 3:

In this case, using Depth-First Search, we would start from 1 then go to 2 and from there go to 3; however, the shortest path would have been to go directly from 1 to 3. So solve this problem, we can use and algorithm called . Breadth-First Search differs from Depth-First Search as instead of going through one child's sub children, we visit all of a node's neighbors before exploring the neighbor's of those nodes. Let's walk through an example, in it, we will keep a similar way of marking nodes that we did for Depth-First Search:

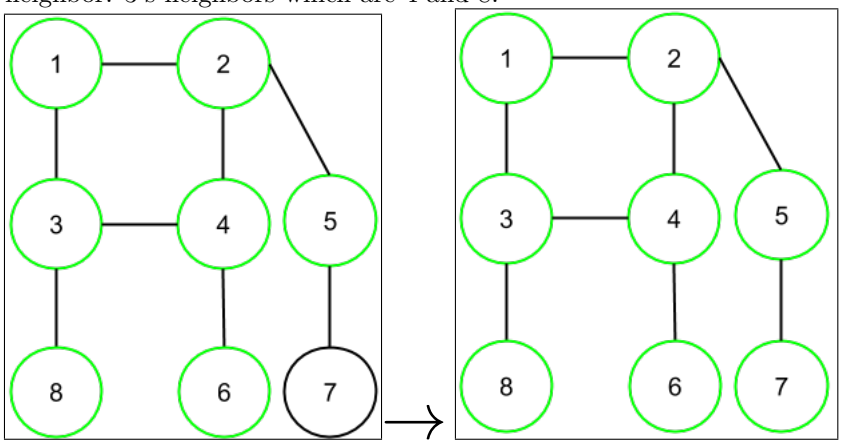


In the initial step, we start off at node 1. We then mark the node and go to its neighbors, which are 2 and 3.



We first go to 2's neighbors, 4 and 5; however, before we visit their neighbors, we go to node 3 and visit its

neighbor. 3's neighbors which are 4 and 8.



These last few steps involve our last level. We go to 4's neighbor, 8, and after that, we go to 5's neighbor 7.

After this, we have finished our Breadth First Search.
Unlike Depth-First Search, we will not use a stack as our underlying data structure for this process. Instead

we use a queue. The queue for the above graphs starts off as $\{1\}$, we then dequeue it and enqeue its neighbors $\{2, 3\}$. We dequeue 2 and enqueue its neighbors which makes the queue $\{3, 4, 5\}$, after this, we enqueue 3's neighbors $\{4, 5, 4, 8\}$. Dequeuing 4, we get $\{5, 4, 8, 6\}$, then after 5 $\{4, 8, 6, 7\}$. We dequeue 4 again; however, since it has already been visited, we do not need to visit it. this leaves us with $\{8, 6, 7\}$. We dequeue each of the following and realize that they do not have any neighbors which leaves us with an empty queue.
We will now have an example implementation of a Breadth-First Search

```
1  public class BreadthFirstDemo {
2      private boolean[] marked;
3      private int[] edgeTo;
4      private final int s;
5
6      public BreadthFirstDemo(Graph G, int s) {
7          marked = new boolean[G.V()];
8          edgeTo = new int[G.V()];
9          this.s = s;
10         bfs(G, s);
11     }
12
13     private void bfs(Graph G, int s) {
14         Queue<Integer> q = new Queue<Integer>();
15         marked[s] = true;
16         q.enqueue(s);
17         while (!q.isEmpty()) {
18             int v = q.dequeue();
19             for (int w : G.adj(v)) {
20                 if (!marked[w]) {
21                     {
22                     edgeTo[w] = v;
23                         marked[w] = true;
24                         q.enqueue(w);
25                         }
26                 }
```

```
27                      }
28                  }
29              }
30  }
```

Breadth-First Search, like Depth-First Search runs in $O(V+E)$ when we use an Adjacency List for the exact same reason.