STACKS, QUEUES, AND GRAPHS (oh my!)

You're already familiar with arrays, and with adding an item to the last free spot in an array, or removing an item from the last filled spot. You remember the annoyance of adding or removing Items from the front of the array, of course, too; all that annoying shifting to fill in the gap. You also know how to add and remove Items from the head or the tail of a linked list. This background means that you will find it easy to learn about the data structure called a stack.

Before we get into the details, let me point out that both C++ and Java give you abstract data types for both arrays and linked lists. These are template classes (note the <>). We'll cover the Java ones here, but you can always look up the C++ ones by googling 'C++ standard template library'. Here's a Java linked list being declared:

```
List<String> list1 = new ArrayList<String>();
```

Notice that the general type List is the same, here, but the implementation details are unique to the right hand side (ArrayList):

You add new items to ArrayLists with the add() method:

```
list1.add("taco");
list1.add("tamale");
```

You can change items already in the list with the set() method:

```
list1.set(0, "torta");       // "taco" becomes "torta"
```

And you can get things from them with the get method:

```
list1.get(1);                // gives back "tamale"
```

Here's a Java array:

```
Vector<Double> array1 = new Vector<String>();
```

In Java, these basic classes of list and array are the basis for other classes that add new public functions to them. In Java, we describe adding new (public) functions to a basic class as extending the interface of the basic class. The first such class we'll discuss is the Stack. Java has a built in Stack class:

```
Stack<Integer> s = new Stack<Integer>();
```

Stacks store Items in last-in first-out (LIFO) order. In practice, this means that you can add Items to the stack, in an operation called push:

```
s.push(4);
```

Each push places the new Item in the 'top' position on the stack.

```
s.push(6); s.push(8); s.push(2);        // 2 is on top
```

You can inspect the stack by asking if it is empty or not.

You can also inspect the 'top' Item on the stack, but you're not really supposed to look at the Items below it.

```
System.out.println(s.peek());           // print the 2
```

You can remove Items from the stack, in a pop() operation. Popping removes the top Item and (in our case) also returns the Item that has been popped off.

```
Integer poppedGuy = s.pop();            // pops the 2
```

What's good about this sort of data structure? Mostly, it gives your program a way to 'remember' data it's seen before and then process it at a later time. A simple example of stacks at work is when your compiler makes sure all your {} brackets match up. The rule is:

```
Make a stack to hold characters;
Read through the .cxx file, letter by letter, looking for '{' and '}'.

When you see a '{', push it on the stack;
When you see a '}',
    If the stack isn't empty,
        pop a '{' off the stack
    Else,
        a '{' is missing or a '}' is extra
When done, if the stack is empty, the '{' and '}' all matched
Else, a '}' is missing or a '{' is extra
```

You can sort of see what it's remembering here: it has to match the very first bracket { with the very last bracket }, because these brackets match inside-to-outside, like this: { { { } { } } { } }.

Stacks can be implemented using a partially filled array (but the order matters here!); if so, we keep track of the number of filled slots and push and pop Items at the end of the array (rather than the front) to avoid shifting elements. You can also make a stack that uses a linked list; in this case, it's easier to keep pushing and popping at the head node of the list (unless you maintain double links and a tail pointer, which makes adding and removing from the tail easy).

Your local memory (which is also called the stack) stores local variables as a stack; each new variable gets pushed on until it goes out of scope (hits its closing }), and then pops off, freeing up a spot for new variables to live.

Stacks are usually considered in common with queues, which store Items in first-in, first out (FIFO) order. It's easy to think of queues as being like the line at the grocery store; you'd be mad if you got in line first and then everyone behind you was allowed to butt in front of you, pay, and leave. Queues (being the opposite of stacks) make sure that the first Item that gets added to them is also the first one that leaves. This provides us with another way to remember preexisting data we've seen in the program, just in the opposite order that a stack remembers it.

Queues are easy to build as a linked list (with a tail pointer), because they let us delete Items from both ends. But this raises a big problem for array-based queues. We have to allow the Items to shift in the array, and that raises the problem of shifting them. In this case, we'd probably use a 'circular' array, which lets the Items slide around (and wrap from the end to the front). There, your implementation has to remember the first slot that is filled and the last slot that is empty, and you need some helper functions to get you slot 0 when you ask for the slot after slot[capacity-1] and to tell when the queue is full. We won't implement a circular queue this semester, but it's a handy structure to use if you know in advance that your queue will never grow beyond a certain size.

If you notice how similar stacks and queues are, you might wonder if you can use one structure for both—and you can, as long as you are careful. There's a structure called a deque—for a double ended queue, which can add or remove Items from either end. Your last graph homework will involve using this deque; add and remove at the same end, and it acts like a stack; add and remove at opposite ends, and it will act like a queue.

```
Deque<Integer> d = new Deque<Integer>();
d.addFirst(7);
d.removeLast();
d.addLast(8);
d.removeFirst();
```

Maps are a data structure that will come in handy soon as well. All the data structures you've seen so far (arrays, lists, and trees) store single Items. But maps store pairs of Items, which don't necessarily have the same types. The first Item is usually a key (a data type that is somehow comparable), and the second one is usually the data you want to store. For instance, CU stores a map for each of you students with your school record as the data, and your student ID number as the key. Python's dictionary structure {} is a map, as well. Java's got a class for Maps:

```
Map<String, String> mapDefinitions = new HashMap<String, String>();
```

Adding items to a map works like this. We put the key first and the value second:

```
mapDefinitions.put("shallow copy", "2 objects that share the exact same dog");
mapDefinitions.put("deep copy", "2 objects with independent data");
```

Later, if you forget what a shallow copy does, you can print (System.out.println() is Java's version of cout << … << endl, so you know):

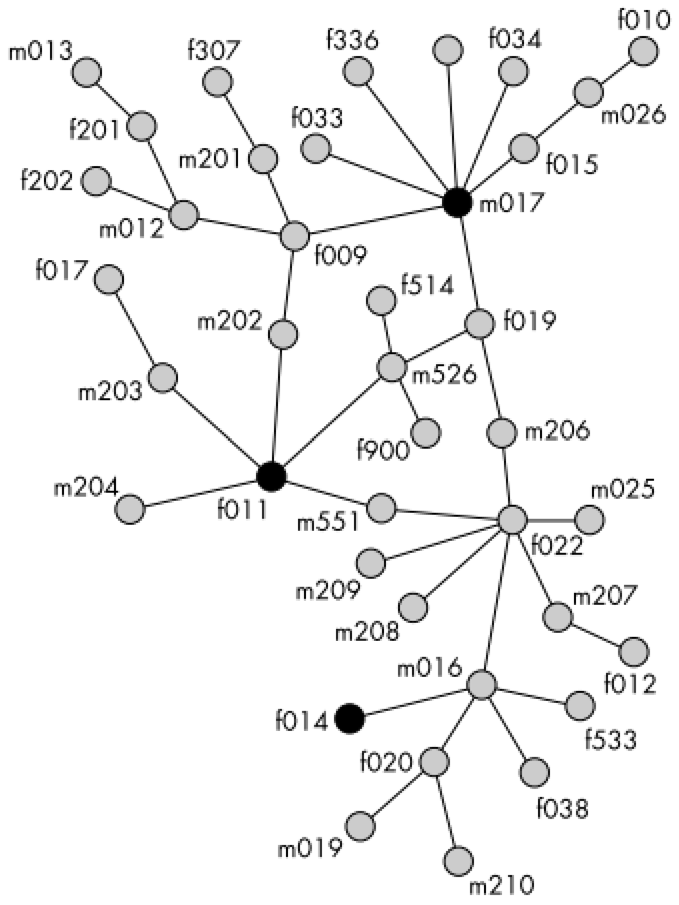
```
System.out.println(mapDefinitions.get("shallow copy"));
```

You can make a Map tell you all the keys it has:

```
mapDefinitions.keySet())          // gives back a List of keys ("deep copy" & "shallow copy" here)
```

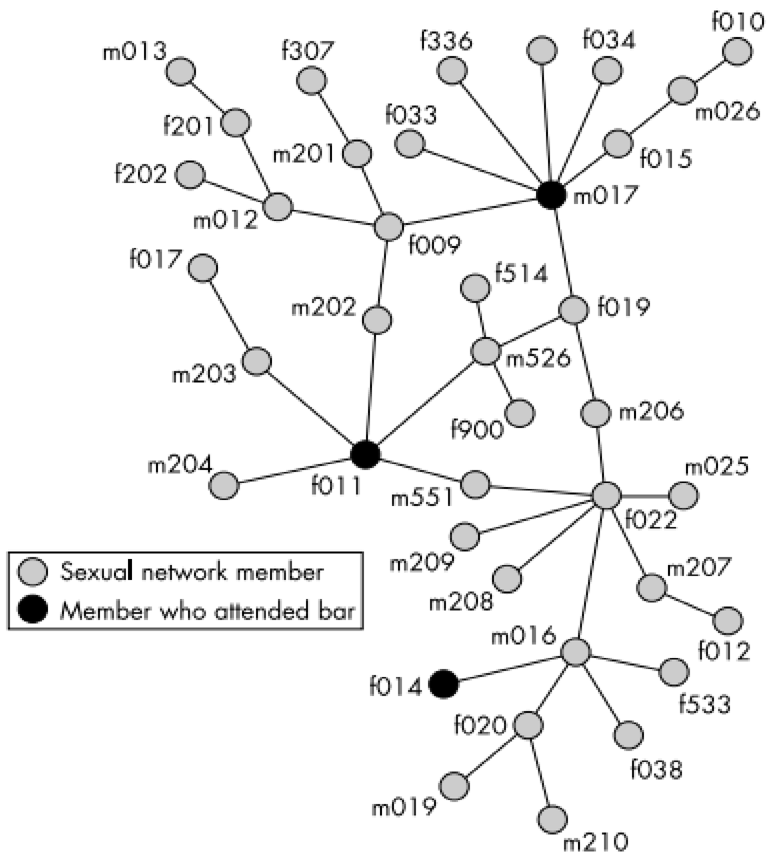
I found this useful for the depth first and breadth first routines.

GRAPHS: keep track of relations between individuals, as below:



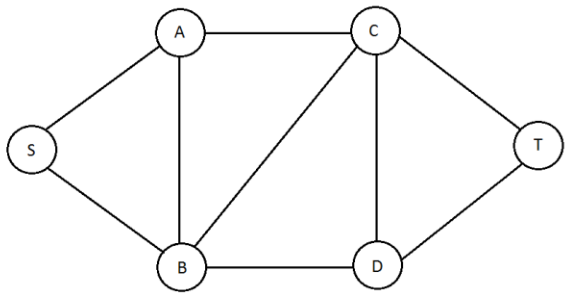
Even when those relationships are not so nice…

Sexual network analysis of a gonorrhoea outbreak



DEPTH FIRST SEARCH OF A GRAPH

Here's a graph along with its edge vector representation; for simplicity we assign all edges a cost of 1 for now. I'm using an adjacency matrix here (so there are zeros for missing edges). Your Graph.java class uses an adjacency list, but this matrix is easier to draw or print out.



[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

We want a method that automatically explores the graph if that's feasible.

To prime the search, we take our starting node S and add it to a particular kind of vector, called toExplore. We also initialize another vector of boolean values, called visited, to all-false except for S. Since we're starting the search there, we can consider S as visited.

[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

vertices

[0]	true
[1]	false
[2]	false
[3]	false
[4]	false
[5]	false

visited

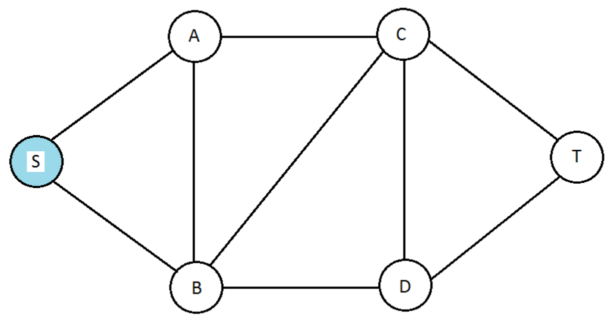
[0]	S

yetToExplore

Now, we repeat these steps:

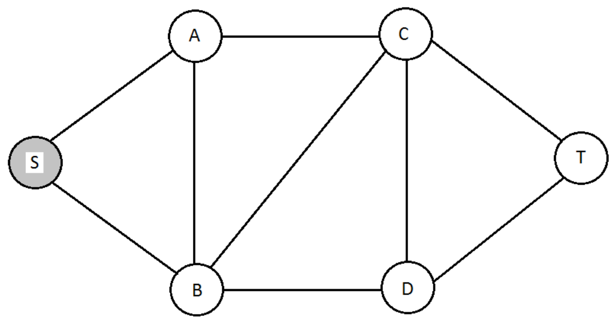
```
1)      While the stack is not empty,
2)              We pop v, the top vertex in yetToExplore, out of the stack
3)              For each neighboring vertex that v has an edge to,
3a)                     If we have not visited this neighbor,
3b)                             Then add it to the top of yetToExplore and mark its visited as true
```

First time though, the vertex we added to yetToExplore is S. The stack is not empty, so we are not done exploring. So we look at the places we can go from S.



	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

From S, we can get to A and B; once S is off yetToExplore, and A and B are on it, we'll have this.



[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

vertices

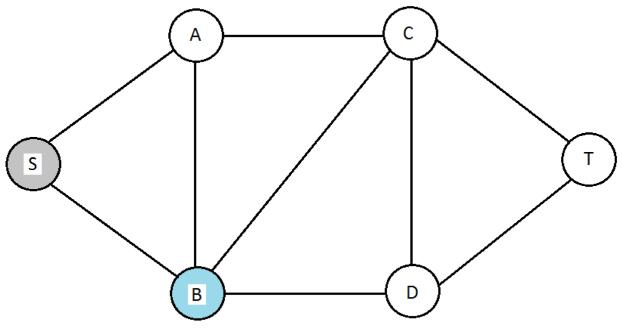
[0]	true
[1]	true
[2]	true
[3]	false
[4]	false
[5]	false

visited

[0]	A
[1]	B

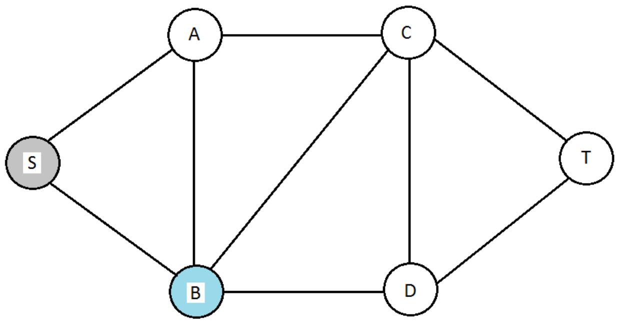
yetToExplore

In the next round, we'll take B off yetToExplore. Then, we look at B's neighbors:



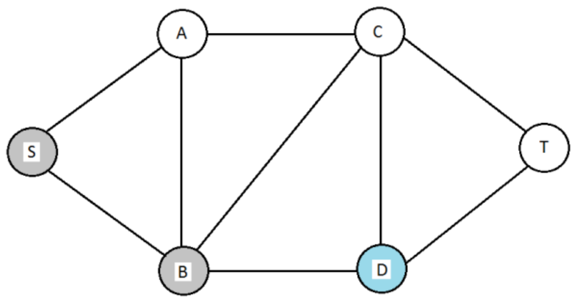
	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

From B, we can get to S and A, but we've already visited those. The unvisited neighbors we still have are C and D, so they go in the yetToExplore vector on top of A.



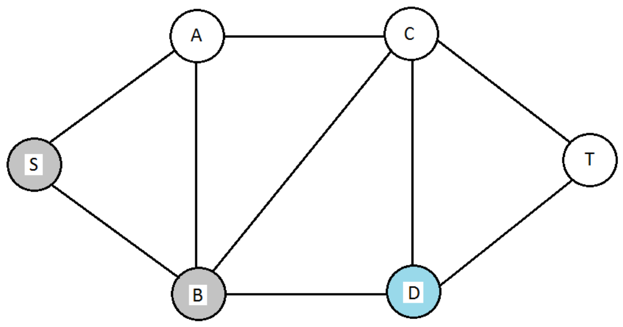
	vertices		visited		yetToExplore
[0]	S	[0]	true	[0]	A
[1]	A	[1]	true	[1]	C
[2]	B	[2]	true	[2]	D
[3]	C	[3]	true		
[4]	D	[4]	true		
[5]	T	[5]	false		

Now in the next round, we'll take D off yetToExplore. Now, we look at D's neighbors:



	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

The only unvisited vertex left is T, so it goes into yetToExplore and gets marked as visited.

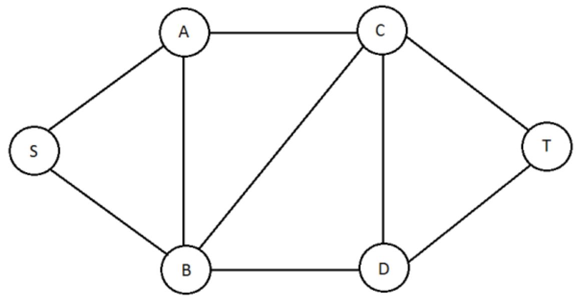


	vertices		visited		yetToExplore
[0]	S	[0]	true	[0]	A
[1]	A	[1]	true	[1]	C
[2]	B	[2]	true	[2]	T
[3]	C	[3]	true		
[4]	D	[4]	true		
[5]	T	[5]	true		

In the final rounds, we pop T off the yetToExplore stack. No further vertices are reachable from T, so nothing gets pushed onto the stack. We then pop C off the yetToExplore stack, and discover no new places we can go from C. Last, we pop A off the stack; once more, no new vertices are reachable from A. So we're done.

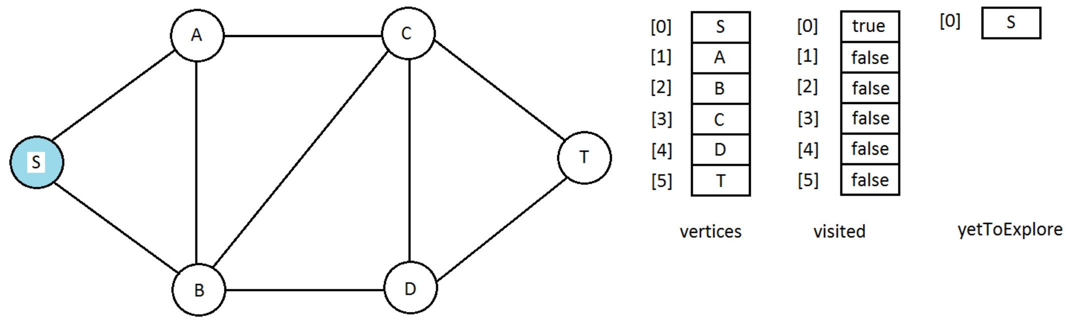
The good thing about depth first search is that it will always explore the graph completely. The bad thing about it is that it will not care about finding vertices near the starting vertex before vertices that are farther away; in fact, depth first search is kind of biased towards finding more distant vertices first, due to that stack.

A second search algorithm, breadth first search, is more efficient at exploring the graph in order from the starting vertex. This search algorithm is almost the same, except that *we add to one end of yetToExplore and remove from the other*. This biases the search toward finding vertices close to the starting one first. We begin with the same graph and the same edges:

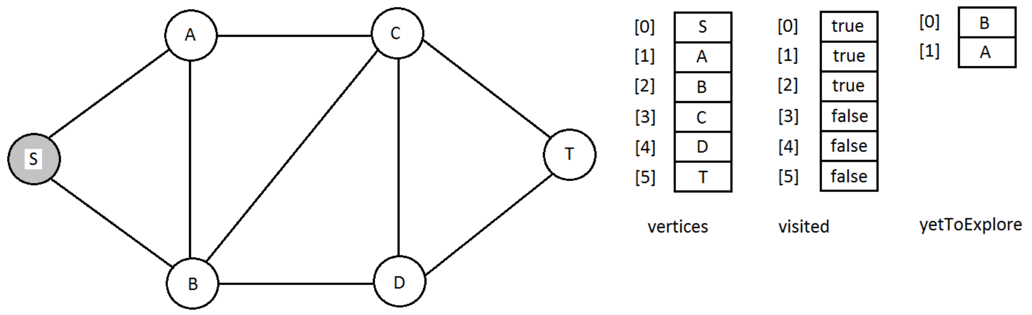


	S	A	B	C	D	T
S	0	1	1	0	0	0
A	1	0	1	1	0	0
B	1	1	0	1	1	0
C	0	1	1	0	1	1
D	0	0	1	1	0	1
T	0	0	0	1	1	0

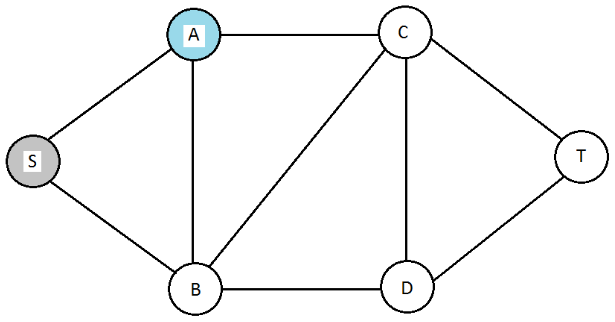
We even start out the same way, with S added to yetToExplore and marked as visited:



When we remove S from the end of the vector, we push its neighbors onto the front of yetToExplore, as before:



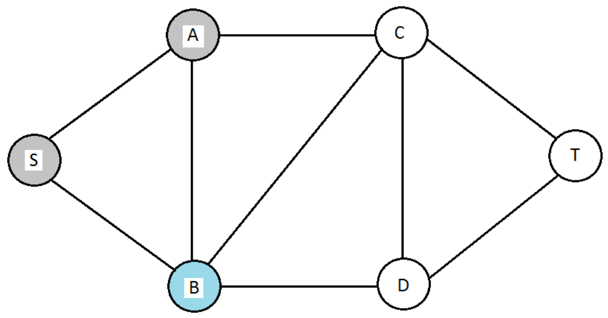
Now, the next time through, we'll inspect A. A has only one unvisited neighbor, C, so C goes in the front of yetToExplore and gets marked as visited:



	vertices	visited
[0]	S	true
[1]	A	true
[2]	B	true
[3]	C	true
[4]	D	false
[5]	T	false

	yetToExplore
[0]	C
[1]	B

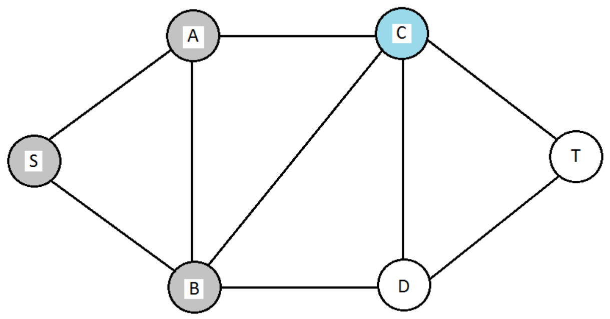
Then, we'll take B off and inspect it. B has one unvisited neighbor, D, so D goes onto the front of yetToExplore:



	vertices	visited
[0]	S	true
[1]	A	true
[2]	B	true
[3]	C	true
[4]	D	true
[5]	T	false

	yetToExplore
[0]	D
[1]	C

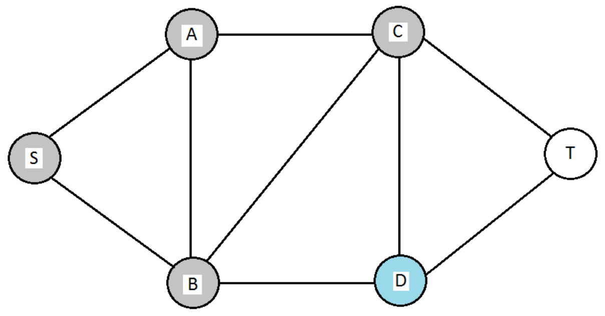
Now, we take C off the end of yetToExplore. From C, we can reach the unvisited vertex T, so we mark T as visited, and add it to the front of yetToExplore:



	vertices	visited
[0]	S	true
[1]	A	true
[2]	B	true
[3]	C	true
[4]	D	true
[5]	T	true

	yetToExplore
[0]	T
[1]	D

When we remove D from the end, nothing gets added to yetToExplore, because D has no unvisited neighbors:



	vertices
[0]	S
[1]	A
[2]	B
[3]	C
[4]	D
[5]	T

	visited
[0]	true
[1]	true
[2]	true
[3]	true
[4]	true
[5]	true

	yetToExplore
[0]	T

Finally, we'll remove the T from yetToExplore, and after that, the stack will be empty. Again, this means we can stop.

Unlike depth first search, breadth first search always finds vertices close to the starting path first, in terms of hops from one vertex to another. It may take longer to do that, though. Getting an optimal solution (the shortest path) is usually more computationally expensive than just getting any solution (any path at all), so this is not surprising.