Problem session 2

Dibran Dokter 1047390

September 12, 2019

1

1.1

```
n = 0
x = L.head
while x.next != NIL
    n+=1
    x = x.next
v = ⌊n/2)⌋
x = head
n = 0
while x.next != NIL
    if n == v
        return x.ky
    n++
    x = x.next
```

1.2

1)

```
isEmpty()
 return no_elements == 0

push(element)
 setLength(stack, no_elements+1)
 stack[no_elements] = element
 no_elements += 1

pop()
 if isEmpty(stack)
     error "Stack empty"
 elem = stack[no_elements]
```

```
  setLength(stack, no_elements-1)
  return elem

top()
  if isEmpty(stack)
      error "Stack empty"
  return stack[no_elements]

display()
  n = 0
  while n < no_elements
      writeLn(stack[n])
```

2)

createStack() = $\mathcal{O}(1)$
isEmpty() = $\mathcal{O}(1)$
push(element) = $\mathcal{O}(1)$
pop() = $\mathcal{O}(1)$
top() = $\mathcal{O}(1)$
display() = $\mathcal{O}(n)$

3)

"AAA"
"AAA"
"BBB"
"CCC"
"CCC"
"AAA"
"BBB"

4)

```
empty()
  setLength(stack,0)
  no_element = 0
```

1.3

when we push n elements after m the time for m is :
$t(m) = y + (n \cdot x) + (n \cdot y) + (n \cdot x) + (n \cdot y) = n \cdot 2XY + y$
because we need the first y between the end of the first push and the next one, then we have (n-1) pushes and pauses to push in the rest of the n elements after

that we start popping the rest of the elements until we get to the pop of the element m

1.4

Q:	1
π:	1
d:	0
Q:	2 3 4
π:	1 1 1
d:	1 1 1
Q:	3 4
π:	1 1
d:	1 1
Q:	4 5 6
π:	1 3 3
d:	1 2 2
Q:	5 6
π:	3 3
d:	2 2
Q:	6
π:	3
d:	2

1.5

To solve this we first run BFS and then we find the vertices which have an outgoing edge with s.
If they have an outgoing edge with s and are in the BFS list as connected to s we have found a cycle.
Then we find the shortest cycle using the distance and return the cycle by getting the predecessors from the vertex.

```
findShortestCycle(G, s)
 bfsList = BFS(G, s)
 outGoingVertex = []
 foreach v in G
     if(v.neighbor == s)
         outGoingVertex.push(v)
 shortestVertex = s
 minDistance = ∞
 foreach v in outGoingVertex
     if v in bfsList                 if v.distance < minDistance
             shortestVertex = v
 if shortestVertex == s
   return False
 shortestCycle = []
 shortestCycle.push(shortestVertex)
 vertex = shortestVertex
 while vertex.distance > 0
     shortestCycle.push(vertex.predecessor)
     vertex = vertex.predecessor
 return (True, shortestCycle)
```

The algorithm first runs BFS to find the paths to s. Then we find the vertices which go to s.

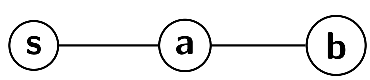
After that we find the vertices which have both an outgoing connection to s and an incoming connection by checking if the vertex is in the BFS list.

If there is a path from a vertex connected to s back to s we have found a cycle. Otherwise there does not exist a cycle.

When we have found the cycle we add the found vertex and loop through the predecessors in the BSF list to create a list of vertices which form the cycle.

1.6

G has to be a undirected and connected graph.
To solve this we need to see that when a vertex has a neighbor that has another neighbor we can create a cycle.
This will not be a simple cycle since the vertices are not distinct.



The cycle here is : [s, A, B, A, s].

This leads to the algorithm:

```
isCyclic(G, s)
 foreach v ∈ adj[s]
     if adj[v] != NIL
         return True
```

else return False