4CCS1DST – Data Structures

Lecture 4:

Solutions to exercises

Exercise 1

How would you modify the Parentheses Matching Algorithm if you wanted as output the pairs of positions of matched parentheses.

For example, for the input

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
(	[	(	a	+	b	)	*	c	+	d	*	e	]	/	{	(	f	+	g	)	-	h	}	)

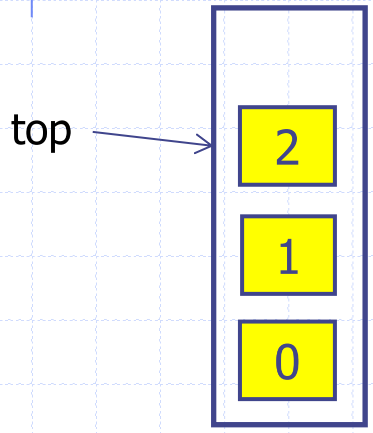
the output should be:

(2,6)   (1,13)   (16,20)   (15,23)   (0,24)

Exercise 1 (cont)

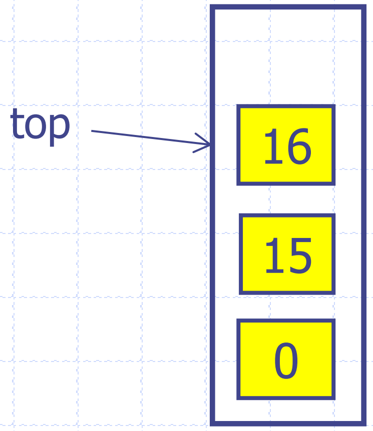
Answer

The Parentheses Matching Algorithm (given on slide 15, Lecture 4) puts opening parentheses on the stack. To identify the positions of the parentheses in each matching pair, keep on the stack the positions of the encountered opening parentheses (not the parentheses themselves). In the given example, after the first three opening parentheses are read, the stack is:



When the algorithm encounters a closing parenthesis at position 6, it takes from the top of the stack the position 2 and checks if the opening parenthesis at this position matches the closing parenthesis at the current position 6. The parentheses at positions 2 and 6 match so the algorithm prints the pair of positions (2,6) and continues.

The stack content when the current position is 19:



Exercise 1 (cont)

The modified Parentheses Matching Algorithm:

```
Let $S$ be an empty stack
for $i = 0$ to $n$-1 do
    if $X[i]$ is an opening grouping symbol then
            $S$.push( $i$ )                    { push on the stack the position of this opening symbol }
    else if $X[i]$ is a closing grouping symbol then
            if $S$.isEmpty() then
                    return false   { nothing to match with }
            $j \leftarrow$ S.pop()
            if $X[j]$ does not match the type of $X[i]$ then
                    return false   { wrong type }
            print "($j$, $i$)"                 { positions of matching grouping symbols }

if $S$.isEmpty() then
    return true { every symbol matched }
else return false { some symbols were never matched }
```

Exercise 2

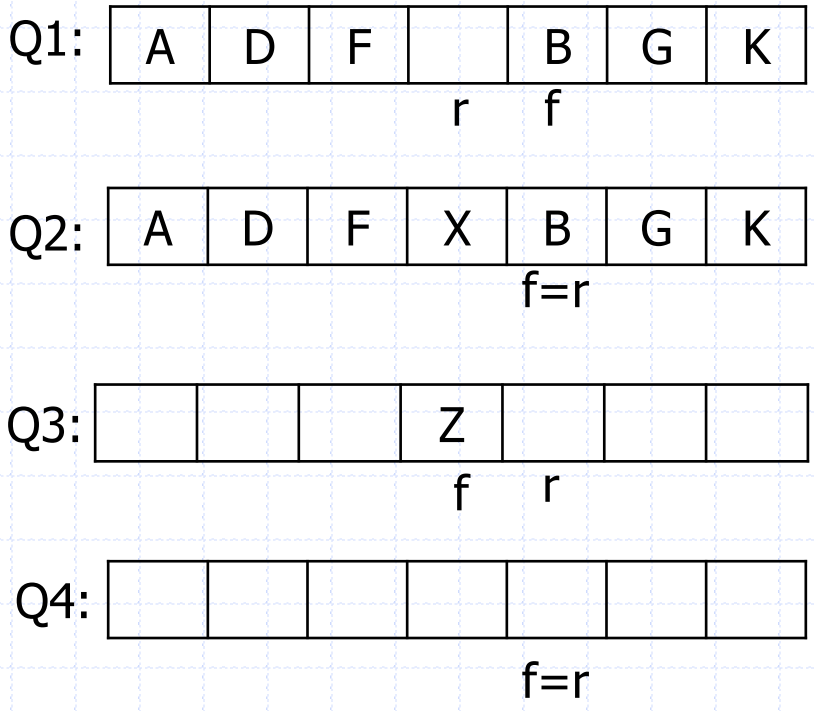
In the array-based implementation of Queue, "array location ***r*** is kept empty." Consequently, when the queue is considered full, there is still one empty location in the array.

Could we put another element in that final empty location, and declare that the queue is full only if the size of the queue is equal to the size (length) of the array? What would be a problem with and how that problem could be fixed?

<u>Answer</u>

If only one empty location is left in the array, then the array is considered full. If we tried to put another element in the last remaining empty location, we would have a problem illustrated by the following example.

Q1 stores queue (B, G, K, A, D, F). If we added another element X, we would get Q2. Now consider Q3, which stores the one-element queue (Z). If we remove this element, we get the empty queue represented in Q4. How could we distinguish between the 7-element queue Q2 and the empty queue Q4? Note that we only rely on the indices f and r.



Exercise 2 (cont)

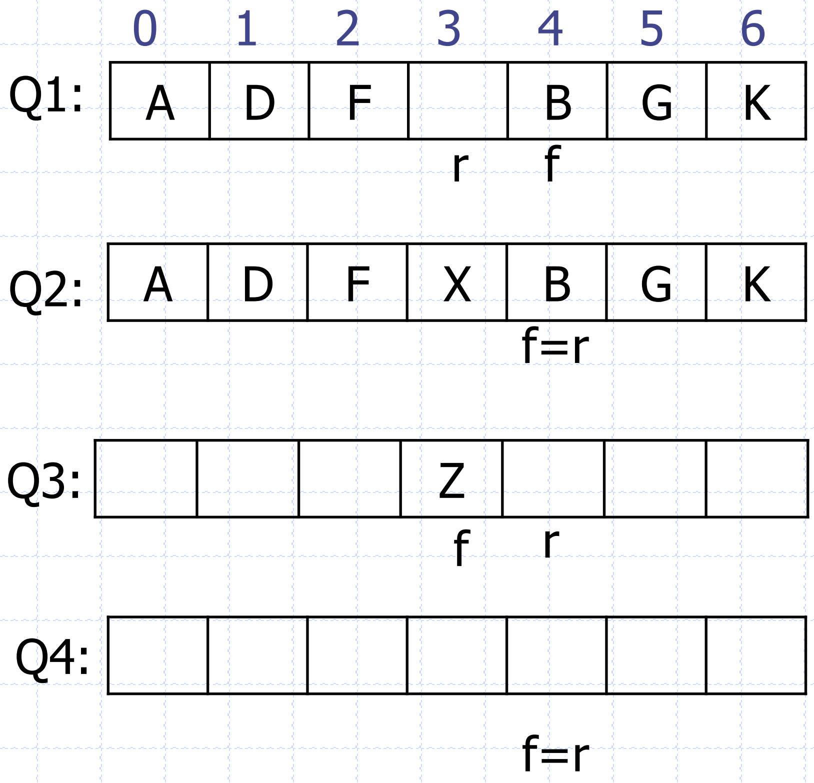
If we do want to use all locations in the array, then we have to maintain some additional information about the current state of the queue. For example, it would be sufficient to maintain, in addition to variables f and r, a boolean variable "full" indicating if the queue is full. For the four queues from the previous page, the variables f, r and full would be:

Q1: f = 4, r = 3, full = false;

Q2: f = 4, r = 4, full = true;

Q3: f = 3, r = 4, full = false;

Q4: f = 4, r = 4, full = false.



Now we know that Q2 is a queue with 7 elements (and for each valid index i, we know where is the element with this index), and we know that Q4 is the empty queue.

The implementation of the queue operations would be more complicated and we decided that those complication would outweigh the gain.

Exercise 3

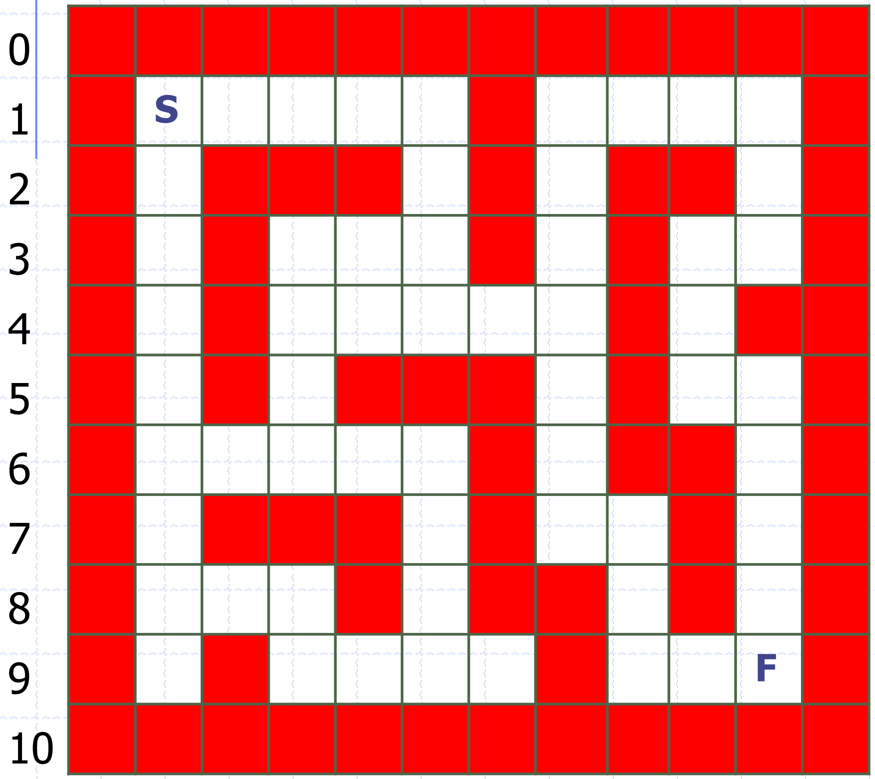
An example of a maze is shown on the next slide. Consider the following algorithm for exploration of such a maze to find out if the Finish location is reachable from the Start location. We can go from one location to a neighbouring location (two neighbouring locations share one side).

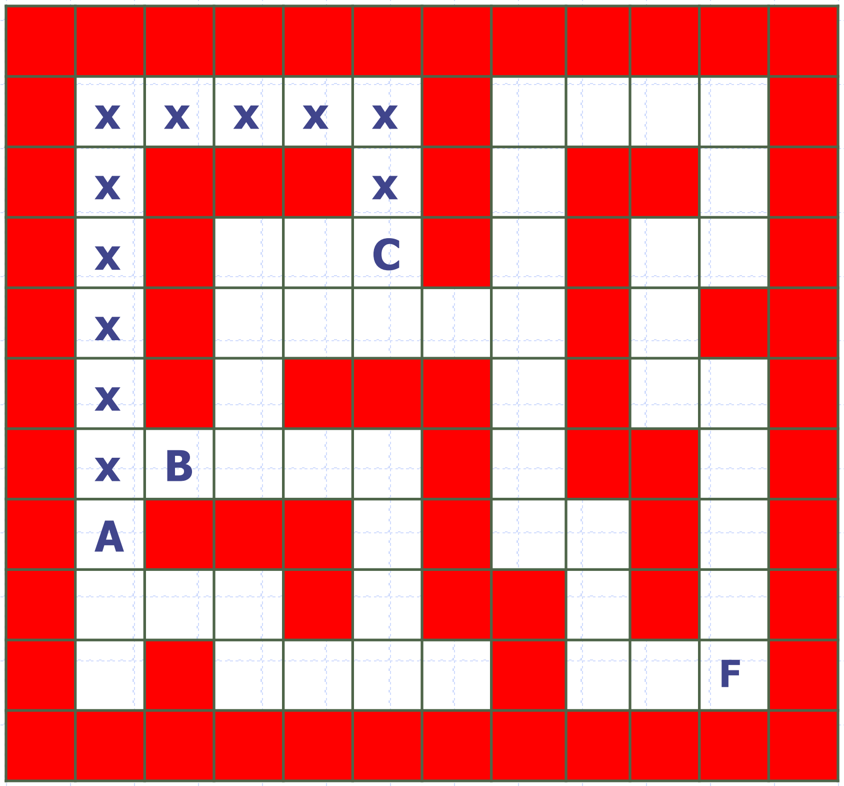
Algorithm:

- Each location is either unknown, discovered or explored.
- Initially Start is discovered and all other locations are unknown.
- All discovered locations are kept in a Queue.
- While Queue is not empty:
    - take (remove) a (discovered) location from Queue,
    - mark this location as explored,
    - mark all its unknown neighbouring locations as discovered and add them to Queue.
- Stop when Finish is discovered (Finish is reachable from Start) or when Queue becomes empty (Finish is not reachable).

Exercise 3 (cont.)

Show the status of each location (unknown, discovered, or explored) when Finish is discovered. Assume the neighbouring locations of the current location are considered in the order: S, E, N, W. The right diagram shows an intermediate state: the explored locations are marked with x; Queue is (A,B,C).





Exercise 3 (cont.)

Answer

Let’s assume that the neighbouring locations of a location z are considered in the following order: S, E, N, W.

The queue at the beginning of an iteration:

1: {(1,1)} - initially only the start location is in the queue;

2: {(2,1), (1,2)} - the neighbouring locations of the start location;

3: {(1,2), (3,1)} - location (2,1) has been removed from the queue and its newly discovered neighbouring location (3,1) has been added to the queue;

4: {(3,1), (1,3)} - location (1,2) has been removed from the queue and its newly discovered neighbouring location (1,3) has been added to the queue;

….

10: {(6,1), (2,5)} - the queue at the beginning of iteration 10;

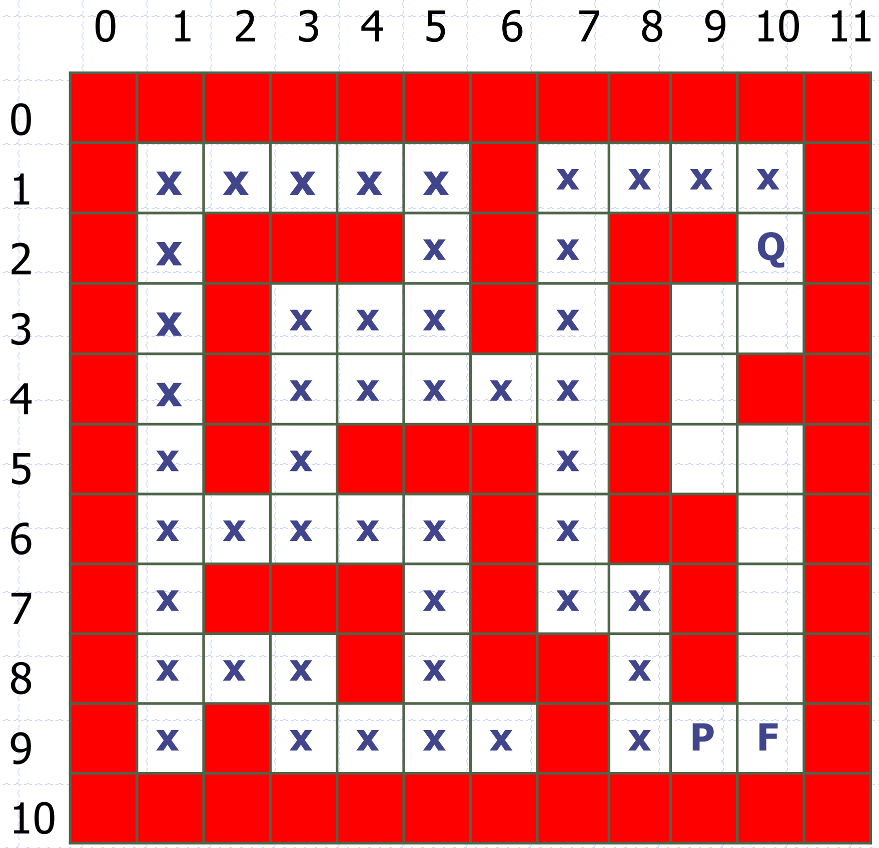
11: {(2,5), (7,1), (6,2)} - location (6,1) has been removed from the queue and its two newly discovered neighbouring locations (7,1) and (6,2) have been added to the queue;

12: {(7,1) (6,2), (3,5)} - location (2,5) has been removed from the queue and the newly discovered location (3,5) has been added to the queue;

We are now in the state as in the right diagram of the previous slide. The exploration continues until the finish location F is reached.

Exercise 3 (cont.)

The left diagram below shows the point during the computation when the Finish location is about to be discovered. Explored locations are marked with x, the current Queue is (P,Q), that is, {(9,9), (2,10)}.



	0	1	2	3	4	5	6	7	8	9	10	11
0	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
1	🟥	x	x	x	x	x	🟥	x	x	x	x	🟥
2	🟥	x	🟥	🟥	🟥	x	🟥	x	🟥	🟥	Q	🟥
3	🟥	x	🟥	x	x	x	🟥	x	🟥			🟥
4	🟥	x	🟥	x	x	x	x	x	🟥		🟥	🟥
5	🟥	x	🟥	x	🟥	🟥	🟥	x	🟥			🟥
6	🟥	x	x	x	x	x	🟥	x	🟥	🟥		🟥
7	🟥	x	🟥	🟥	🟥	x	🟥	x	x	🟥		🟥
8	🟥	x	x	x	🟥	x	🟥	🟥	x	🟥		🟥
9	🟥	x	🟥	x	x	x	x	🟥	x	P	F	🟥
10	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥

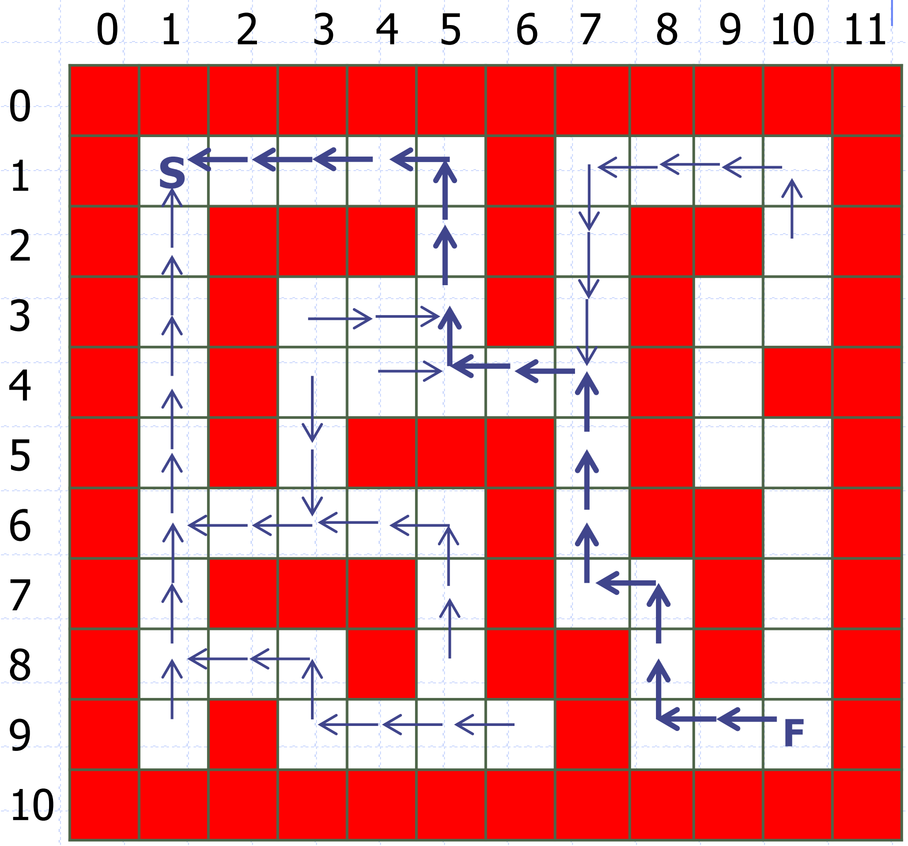
Exercise 3 (cont.)

How could we extend this algorithm to output a path from the location Start to the location Finish, if Finish is reachable from Start?

For each discovered location, remember from which neighbouring location this location was discovered. The arrows in this diagram show where the locations were discovered from. For example, the location (2,1) was discovered from location (1,1), the Finish location (9,10) was discovered from location (9,9), etc.

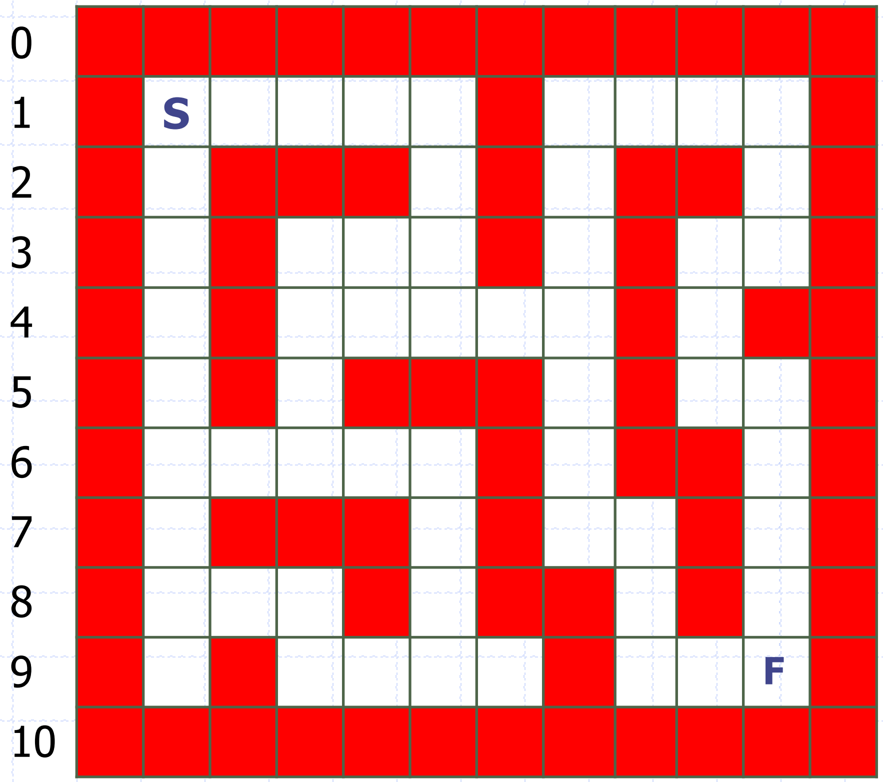
We can obtained a path from S to F, in the backward direction, by tracing the arrows from F back to S. This path is highlighted on the diagram. It is a shortest paths from S to F.

This algorithm always finds a shortest path from S to F, unless there is no path from S to F.



Exercise 3 (cont.)

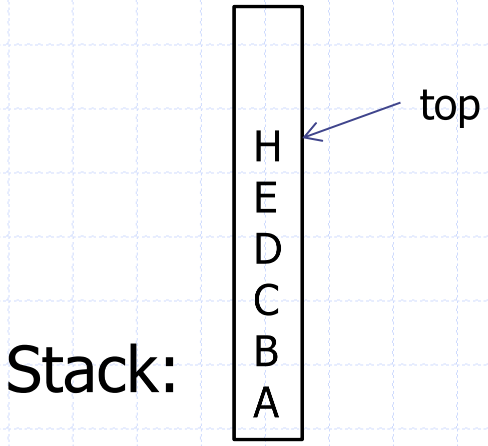
What would happen, if we used Stack instead of Queue in this algorithm? Trace the computation showing the status of the locations (unknown, discovered, or explored) and the contents of the Stack. Assume the neighbouring locations of the current location are considered in the order: S, E, N, W.

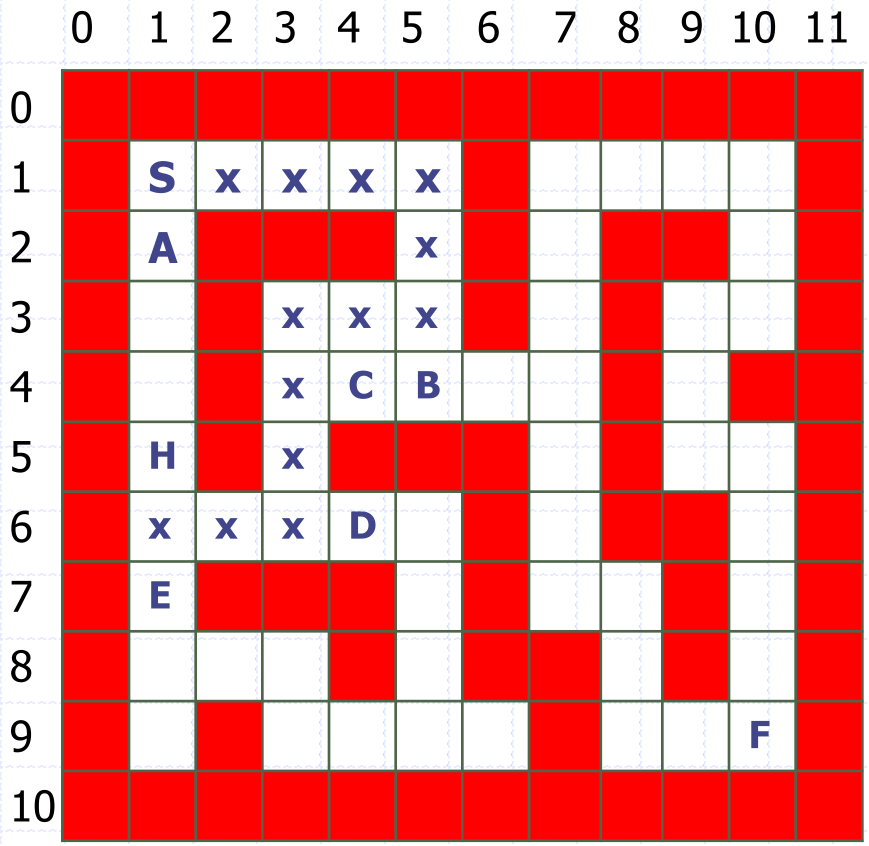


Exercise 3 (cont.)

We would also correctly identified whether F is reachable from S, if we used Stack instead of Queue.

An intermediate state of the computation with Stack is shown on the diagrams. A path from S to F will be found, but it might not be a shortest path.


Stack:
H
E
D
C
B
A
top


0 1 2 3 4 5 6 7 8 9 10 11
0 1 2 3 4 5 6 7 8 9 10
S
A
B
C
D
E
H
F
x