ITI 1121. Introduction to Computing II

Queue: applications

by

Marcel Turcotte

Version March 7, 2020

Preamble

Preamble

Overview

Overview

Queue: applications

We are interested in all aspects of queues in programming. We examine several examples of their use, including resource sharing and simulation algorithms. We explore the concept of **breadth-first-search**.

General objective:

- This week, you will be able to implement a breadth-first-search algorithm.

Preamble

Learning objectives

Learning objectives

- **Justify** the role of a queue in solving a computer problem.
- **Design** a computer program requiring the use of a queue.
- **Implement** a breadth-first-search.

Readings:

- `https://en.wikipedia.org/wiki/Breadth-first_search`

Preamble

Plan

Plan

Asynchronous processing

Applications such as **producer/consumer**, **client/server** or **sender/receiver** require the use of queues if the data processing is asynchronous.

Asynchronous processing means that the client and server are not synchronized, the server is not ready or able to receive the data at the time and speed of sending.

This treatment requires a queue:

- The **client** inserts data into the queue (**enqueue**);
- The **server** removes data from the queue at the appropriate time (**dequeue**).

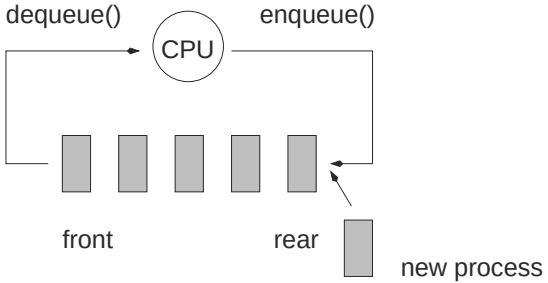
Such a queue is sometimes called a **buffer**.

Asynchronous processing

In particular, the **inter-process communications** in an operating system work like this.

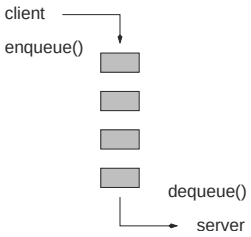
1. Printer spooler.;
2. Buffered i/o;
3. access to disks;
4. the transmission of messages (packets) over a network.

Timeshare



All modern **operating systems** are time-shared. One of the common techniques for time-sharing is called **round-robin**. The first process in the queue (dequeue) is assigned a time slot after which its execution is suspended and the process is put at the end of the queue (enqueue), and we move on to the next process.

Communications between processes



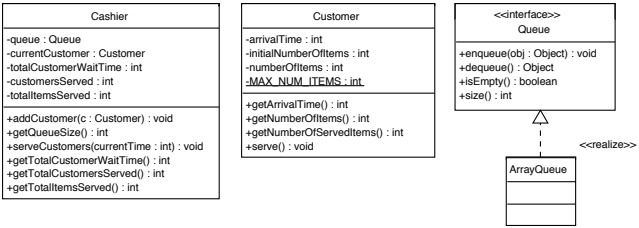

```
while (true) {
  while (! q.isFull()) {
    q.enqueue( ... );
  }
}
```

```
while (true) {
  while (! q.empty()) {
    process(q.dequeue())
  }
}
```

$\Rightarrow$ *inter-process communication* (IPC), *buffered i/o*, etc.

Applications

- **Simulations**;



- **Breadth-first-search**.

Introduction

What displays the method Search.run()?

```
public class Search {
    public static void run() {

        Queue<String> q;
        q = new LinkedQueue<String>();
        q.enqueue("");

        while (true) {
            String s;
            s = q.dequeue();

            q.enqueue(s + "0");
            q.enqueue(s + "1");

            System.out.println(q);
        }
    }
}
```

What's that for?

```
Queue<String> q;
q = new LinkedQueue<String>();
q.enqueue("");

while (true) {
    String s;
    s = q.dequeue();

    q.enqueue(s + "0");
    q.enqueue(s + "1");

    System.out.println(q);
}
```

- This algorithm generates **all** strings consisting of the symbols **0** and **1** in ascending order of length: 0, 1, 00, 01, 10, 11, 000, 001, ...

```
[""]
["0"]
["0","1"]
["1"]
["1","00"]
["1","00","01"]
["00","01"]
["00","01","10"]
["00","01",’10","11"]
["01","10","11"]
["01","10","11","000"]
["01","10","11","000","001"]
["10","11","000","001"]
["10","11","000","001","010"]
["10","11","000","001","010","011"]
["11","000","001","010","011"]
["11","000","001","010","011","100"]
["11","000","001","010","011","100","101"]
```

What does it do?

```
Queue<String> q;
q = new LinkedQueue<String>();
q.enqueue("");

while (true) {
    String s;
    s = q.dequeue();

    q.enqueue(s + "L");
    q.enqueue(s + "R");
    q.enqueue(s + "U");
    q.enqueue(s + "D");

    System.out.println(q);
}
```

- This algorithm generates **all** strings formed by the symbols **L**, **R**, **U** and **D** in **increasing order of length**: L, R, U, D, LL, LR, LU, LD, ...

```
[]
[""]
[]
["L"]
["L","R"]
["L","R","U"]
["L","R","U","D"]
["R","U","D"]
["R","U","D","LL"]
["R","U","D","LL","LR"]
["R","U","D","LL","LR","LU"]
["R","U","D","LL","LR","LU","LD"]
["U","D","LL","LR","LU","LD"]
["U","D","LL","LR","LU","LD","RL"]
["U","D","LL","LR","LU","LD","RL","RR"]
["U","D","LL","LR","LU","LD","RL","RR","RU"]
["U","D","LL","LR","LU","LD","RL","RR","RU","RD"]
["D","LL","LR","LU","LD","RL","RR","RU","RD"]
["D","LL","LR","LU","LD","RL","RR","RU","RD","UL"]
```

Labyrinth

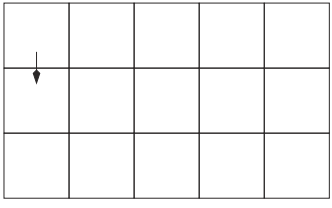
Semantics

- **What are** these **L**s, **R**s, **U**s and **D**s?

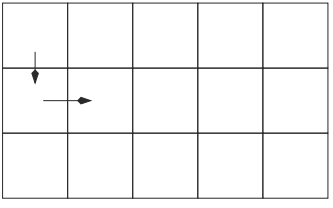
Each symbol corresponds to a **direction**:

L = *left*;
R = *right*;
U = *up*;
D = *down*.

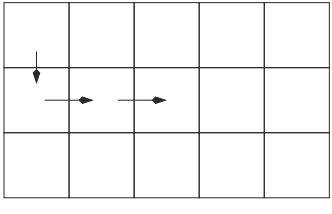
Each **string** corresponds to a **path** in two-dimensional space.



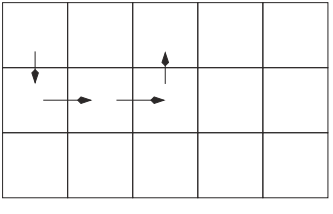
D



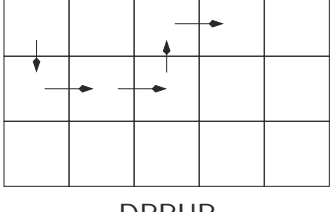
DR



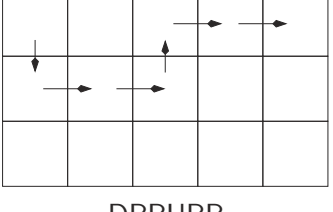
DRR



DRRU



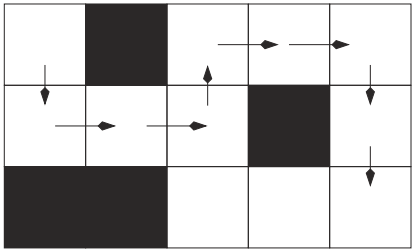
DRRUR



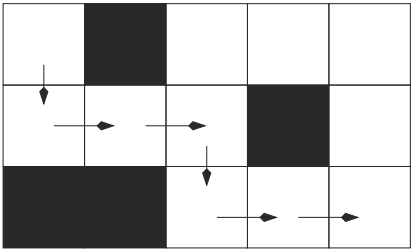
DRRURR

Let's add some obstacles.

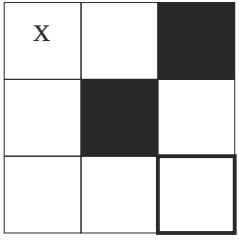
- Now let's assume that some of the cells are **unaccessible**.

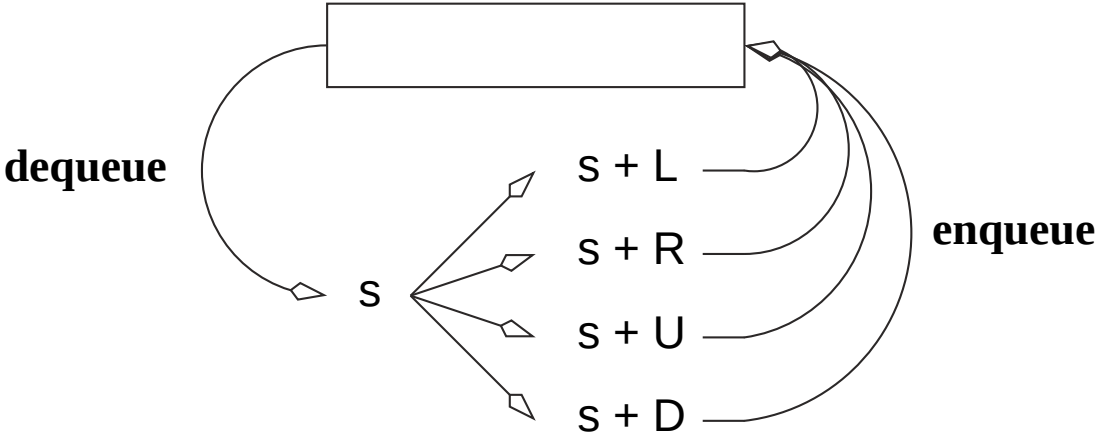


DRRURRDD

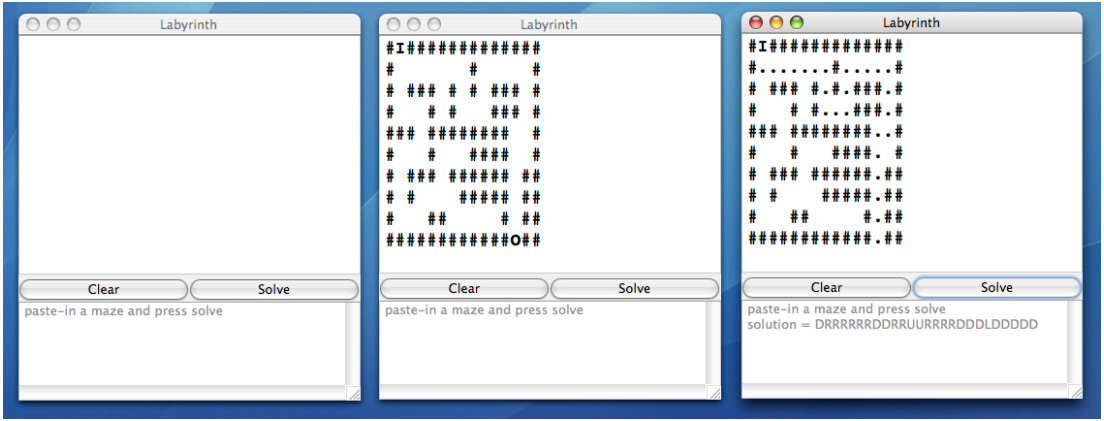


DRRDRR


X


dequeue
s
s + L
s + R
s + U
s + D
enqueue

Finding the shortest path in a maze



Implementation

Helper methods

- We'll need a method that checks if a partial solution is valid, **checkPath(String path)**.
- As well as a method that tells us if a valid solution achieves its goal, **reachesGoal(String path)**.

Data structures

A matrix of characters, i.e. a **2-dimensional array**.

```
char [][] maze;
```

An inaccessible position (a wall) is denoted by '#', an empty cell by ' ', and a visited position by '+'.

```
#+#####
#+# # #
#++ # #
###   #
##### #
```

checkpath(String path)

```
private boolean checkPath(String path) {

    boolean[][] visited;
    visited = new boolean[MAX_ROW][MAX_COL];

    int row, col;

    row = 0;
    col = 0;

    int pos=0;

    boolean valid = true;
```

checkpath(String path)

```
...
while (valid && pos < path.length()) {
    char direction = path.charAt(pos++);
    switch (direction) {
    case LEFT:
        col--;
        break;
    case RIGHT:
        col++;
        break;
    case UP:
        row--;
        break;
    case DOWN:
        row++;
        break;
    default:
        valid = false;
    }
    ...
```

checkpath(String path)

- After each move, we check that the current position is valid, i.e. inside the labyrinth, is not a wall, and has not been visited.

```
        if ((row >= 0) && (row < MAX_ROW) &&
            (col >= 0) && (col < MAX_COL)) {
            if (visited[row][col] || grid[row][col]==WALL) {
                valid = false;
            } else {
                visited[ row ][ col ] = true;
            }
        } else {
            valid = false;
        }
    } // end of while loop

    return valid;
}
```

«Are we done yet!»

```
    private boolean reachesGoal(String path) {
        int row = 0;
        int col = 0;
        for (int pos=0; pos < path.length(); pos++) {
            char direction = path.charAt(pos);
            switch (direction) {
            case LEFT:  col--; break;
            case RIGHT: col++; break;
            case UP:    row--; break;
            case DOWN:  row++; break;
            }
        }
        return grid[row][col] == OUT;
    }
```

Conclusions

Breadth-first-search

- The algorithm using a **queue** is called **breadth-first search**.
- The **search tree** is built level by level. All the sequences of the same level (thus all the sequences of the same length) are processed before proceeding with the next level.

Depth-first-search

- The algorithm using a **stack** is called the **depth-first search**.
- The **search tree** is built branch by branch. A sequence is selected and repeatedly extended until no extension is valid. The algorithm then backtracks, hence the nickname **backtracking**.

Prologue

Summary

- A **breadth-first-search** uses a **queue** to find the **shortest path** in a search space.

Next module

- **Queue** : array-based implementation

References I


E. B. Koffman and Wolfgang P. A. T.
***Data Structures: Abstraction and Design Using Java*.**
John Wiley & Sons, 3e edition, 2016.

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa