ITI 1521. Introduction à l'informatique II

File : applications

by

Marcel Turcotte

Version du 7 mars 2020

Préambule

Préambule

Aperçu

Aperçu

File : applications

Nous nous intéressons à tous les aspects des files en programmation. Nous examinons plusieurs exemples de leur utilisation, notamment le partage de resources et les algorithmes de simulation. Nous explorons le concept de parcours en largeur («*breadth-first-search*»).

Objectif général :

- Cette semaine, vous serez en mesure d'implémenter un algorithme de parcours en largeur.

Préambule

Objectifs d'apprentissage

Objectifs d'apprentissage

- **Justifier** le rôle d'une file dans la résolution d'un problème informatique.
- **Concevoir** un programme informatique nécessitant l'utilisation d'une file.
- **Implémenter** un parcours en largeur.

Lectures :

- `https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur`

Préambule

Plan du module

Plan

Traitement asynchrone

Les applications de type **producteur/consommateur**, **client/serveur** ou **sender/receiver** nécessitent l'utilisation de files si le traitement des données est asynchrone.

Un traitement **asynchrone** signifie que le client et le serveur ne sont pas synchronisés, le serveur n'est pas prêt ou capable de recevoir les données au temps et à la vitesse de l'envoi.

Ce traitement nécessite une file :

- Le **client insère** des données dans la file (**enqueue**) ;
- Le **serveur retire** des données de la file (**dequeue**) au moment opportun.

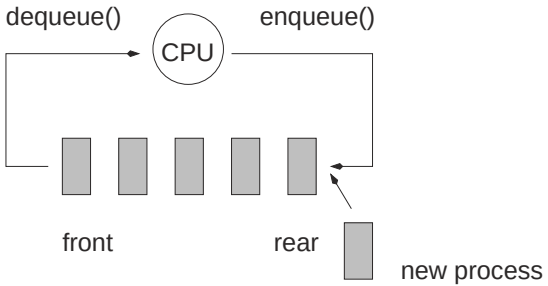
Une telle file est parfois appelée un **tampon** (*buffer*).

Traitement asynchrone

En particulier, les **communications entre processus** (*inter-process communication*) dans un système d'exploitation fonctionnent de la sorte.

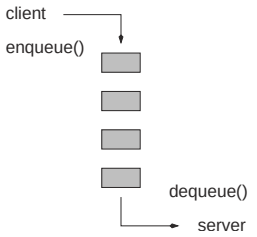
1. gestionnaire d'imprimante (*printer spooler*) ;
2. *buffered i/o* ;
3. l'accès aux disques ;
4. la transmission de messages (paquets) sur un réseau.

Temps partagé



Tous les **systèmes d'exploitation modernes** sont à **temps partagé**. L'une des techniques communes pour le partage du temps s'appelle *round-robin*. Le premier processus en file (dequeue) se voit attribuer une tranche de temps après laquelle son exécution est suspendue et le processus est mis à la fin de la file (enqueue), et on passe au processus suivant.

Communications entre processus



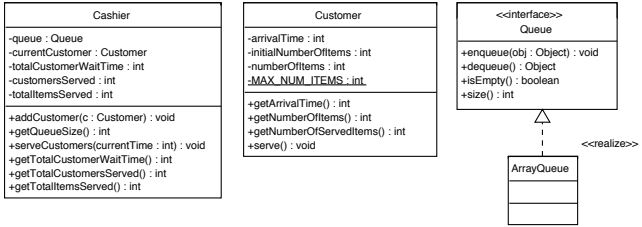

```
while (true) {
  while (! q.isFull()) {
    q.enqueue( ... );
  }
}
```

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

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

Applications

- **Simulations** ;



- **Parcours en largeur**.

Introduction

Qu'affiche la méthode 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);
        }
    }
}
```

À quoi ça sert ?

```
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);
}
```

- Cet algorithme génère **toutes** les chaînes de caractères formées des symboles **0** et **1** en **ordre croissant de longueur** : 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"]
```

À quoi ça sert ?

```
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);
}
```

- Cet algorithme génère **toutes** les chaînes de caractères formées des symboles **L**, **R**, **U** et **D** en **ordre croissant de longueur** : 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"]
```

Labyrinthe

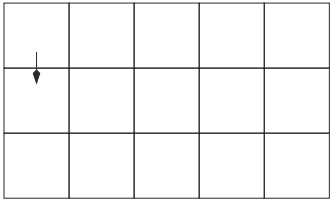
Sémantique

- **Que sont** ces **L**s, **R**s, **U**s et **D**s ?

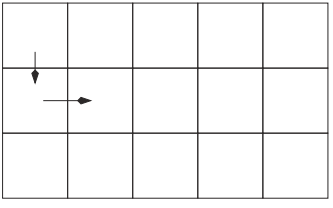
Chaque symbole correspond une **direction** :

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

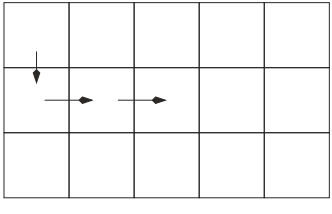
Chaque chaîne correspond à un **chemin** (*path*) dans un espace à 2 dimensions.



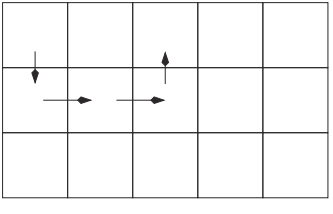
D



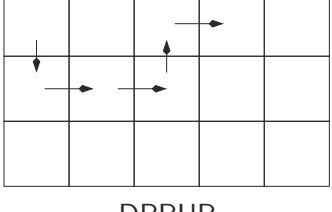
DR



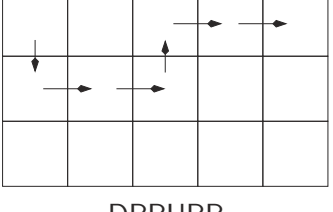
DRR



DRRU



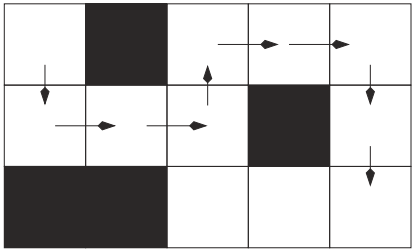
DRRUR



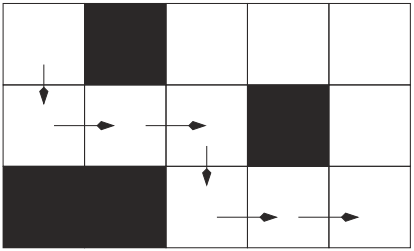
DRRURR

Ajoutons des obstacles

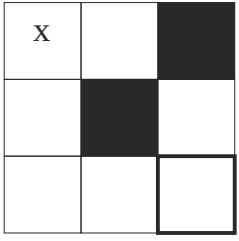
- Supposons maintenant que certaines cases soient **inaccessibles**.

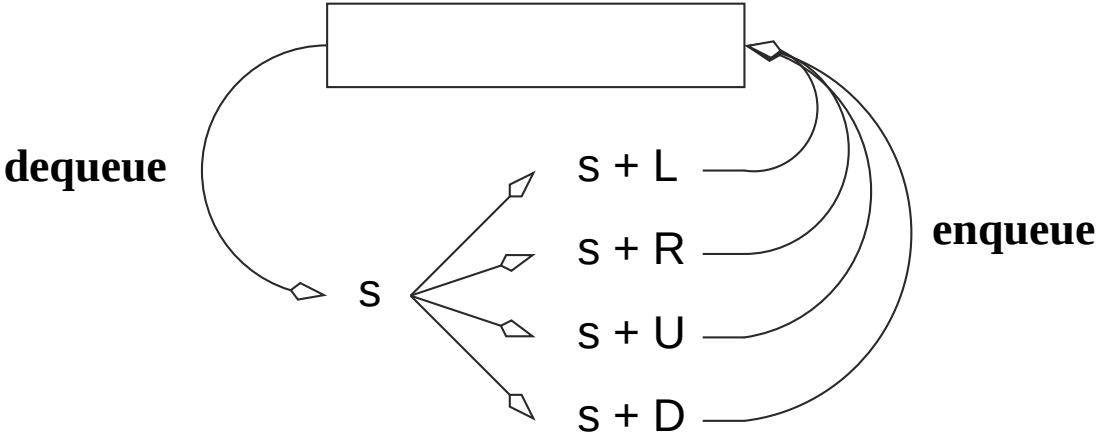


DRRURRDD

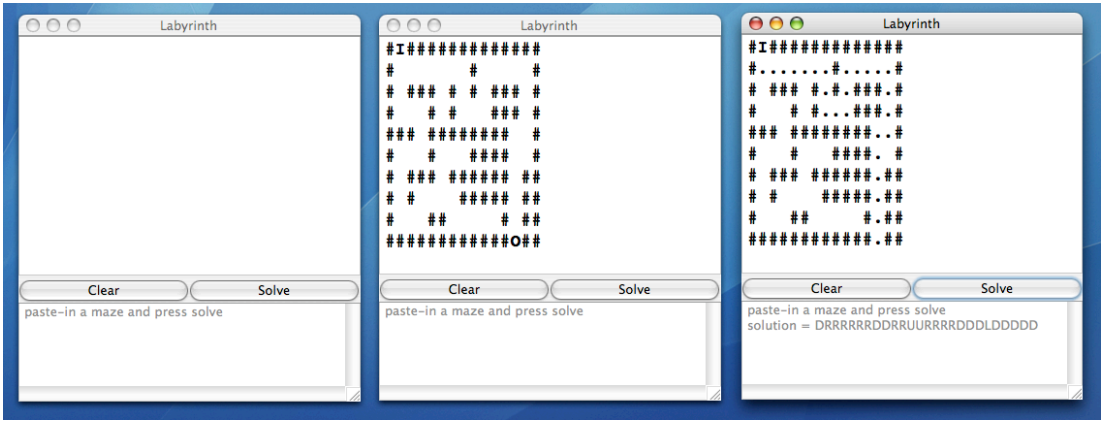


DRRDRR


X


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

Trouver le plus court chemin dans un labyrinthe



Implémentation

Méthodes auxiliaires

- Nous aurons besoin d'une méthode qui vérifie si une solution partielle est valide, **checkPath(String path)**.
- Et aussi une méthode qui nous dit si une solution valide atteint son but, **reachesGoal(String path)**.

Structures de données

Une matrice de caractères, c.-à-d. un **tableau à 2 dimensions**.

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

Une position inaccessible (un mur) est notée par '#', une case vide par ' ', et une position visitée par '+'.

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

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)

- Après chaque déplacement, nous vérifions que la position courante est valide, c.-à-d. à l'intérieur du labyrinthe, n'est pas un mur, et n'a pas été visitée.

```
        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

Parcours en largeur («breadth-first-search»)

- L'algorithme utilisant une **file** s'appelle le «**parcours en largeur**» (**breadth-first search**).
- L'**arbre de recherche** est construit niveau par niveau. Toutes les séquences d'un même niveau (donc toutes les séquences de même longueur) sont traitées avant de procéder avec le niveau suivant.

Parcours en profondeur («depth-first-search»)

- L'algorithme utilisant une **pile** s'appelle le «**parcours en profondeur**» (**depth-first search**).
- L'**abre de recherche** est construit branche par branche. Une séquence est sélectionnée et étendue à répétition jusqu'à ce qu'aucune extension ne soit valide. L'algorithme revient alors en arrière, d'où le surnom d'algorithme de retour-arrière (**backtracking**).

Prologue

Résumé

- Un **parcours en largeur** utilise une **file** afin de trouver le **plus court chemin** dans un espace de recherche.

Prochain module

- **Queue** : applications

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

École de **science informatique** et de génie électrique (SIGE)

Université d'Ottawa