ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 10, 2013

Abstract

- Queues-based algorithms
  - Asynchronous processes
  - Simulations
  - State-space search

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Summary

Queues are implemented with arrays or linked elements.

Summary

Queues are implemented with arrays or linked elements.

In the case of circular arrays, **modulo** arithmetic is used when incrementing the index so that the queue wraps around the array when it reaches the end.

```
index = ( index + 1 ) % MAX_QUEUE_SIZE;
```

Summary

Queues are implemented with arrays or linked elements.

In the case of circular arrays, **modulo** arithmetic is used when incrementing the index so that the queue wraps around the array when it reaches the end.

```
index = ( index + 1 ) % MAX_QUEUE_SIZE;
```

One must be careful to detect the case when the array gets full and avoid **overriding** any elements as well as distinguishing between the full and empty queues.

Summary

Queues are implemented with arrays or linked elements.

In the case of circular arrays, **modulo** arithmetic is used when incrementing the index so that the queue wraps around the array when it reaches the end.

```
index = ( index + 1 ) % MAX_QUEUE_SIZE;
```

One must be careful to detect the case when the array gets full and avoid **overriding** any elements as well as distinguishing between the full and empty queues.

Several implementations are possible: using **sentinel values**, destroying the array, using a **boolean value** to indicate if the queue is full/empty or to maintain a **count** of the number of elements in the queue. Of course, the details of the implementation of each method will vary with the implementation.

Summary

Queues are implemented with arrays or linked elements.

In the case of circular arrays, **modulo** arithmetic is used when incrementing the index so that the queue wraps around the array when it reaches the end.

```
index = ( index + 1 ) % MAX_QUEUE_SIZE;
```

One must be careful to detect the case when the array gets full and avoid **overriding** any elements as well as distinguishing between the full and empty queues.

Several implementations are possible: using **sentinel values**, destroying the array, using a **boolean value** to indicate if the queue is full/empty or to maintain a **count** of the number of elements in the queue. Of course, the details of the implementation of each method will vary with the implementation.

The operation **dequeue()** is sometimes called **serve()**; because queues are often used in the context of client/server applications.

Asynchronous processes

Applications such as **producer/consumer**, **client/server** or **sender/receiver** necessitate using a queue for asynchronous processing of the data.

Asynchronous processes

Applications such as **producer/consumer**, **client/server** or **sender/receiver** necessitate using a queue for asynchronous processing of the data.

Asynchronous processing means that the client and the server are not synchronized, which would occur if the server is not ready or capable of receiving the data at that time or speed.

Asynchronous processes

Applications such as **producer/consumer**, **client/server** or **sender/receiver** necessitate using a queue for asynchronous processing of the data.

Asynchronous processing means that the client and the server are not synchronized, which would occur if the server is not ready or capable of receiving the data at that time or speed.

Asynchronous processes:

- The client insert data into a queue (enqueue);
- The server removes data from the queue (dequeue) whenever it's ready for processing.

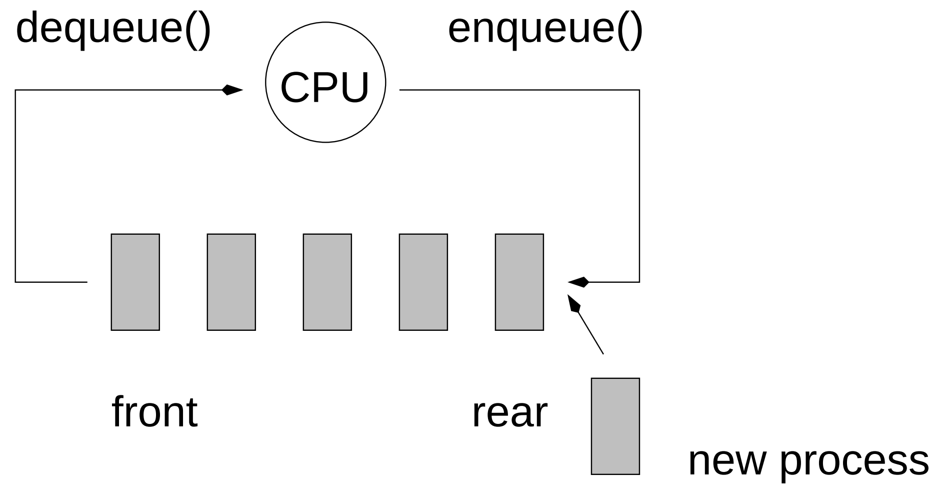
In this context, the queue is sometimes called a **buffer**.

Asynchronous processes

In particular, *inter-process communications* in operating systems work like this.

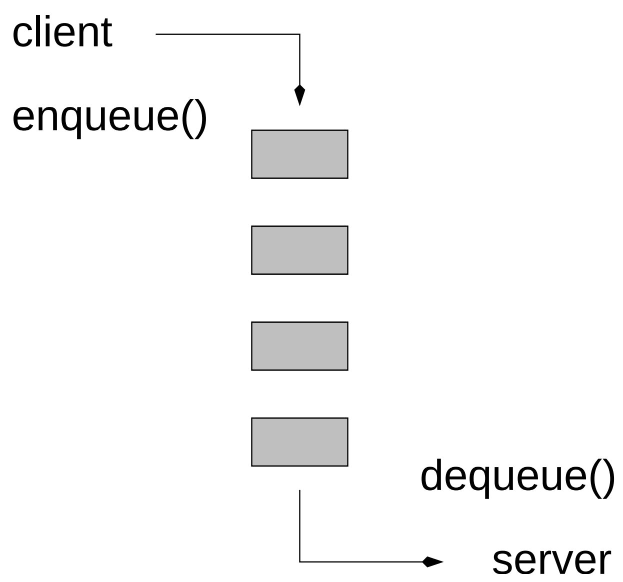
1. *printer spooler*;
2. *buffered i/o*;
3. disk accesses;
4. sending messages (packets) across the network.

Time-shared applications

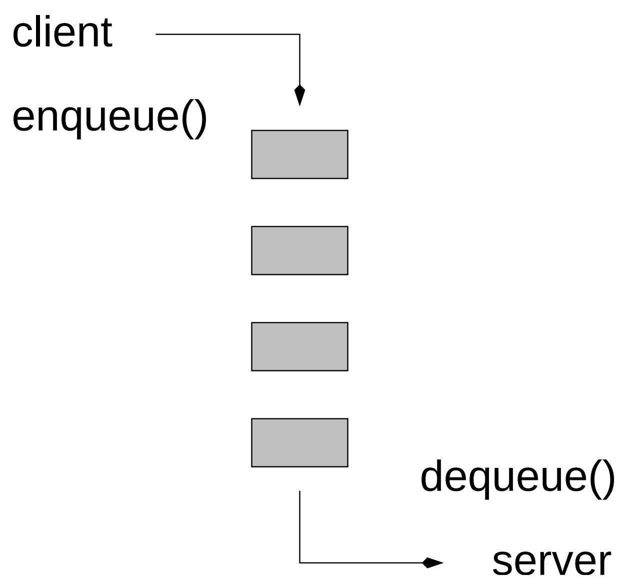


All modern operating systems operate in time-shared mode. One of the frequent techniques to share time is called *round-robin*. The first process in the queue is allocated a slice of time (dequeue) after which it is suspended and put at the end of the queue (enqueue), time is allocated for the next process.

Inter-process communications

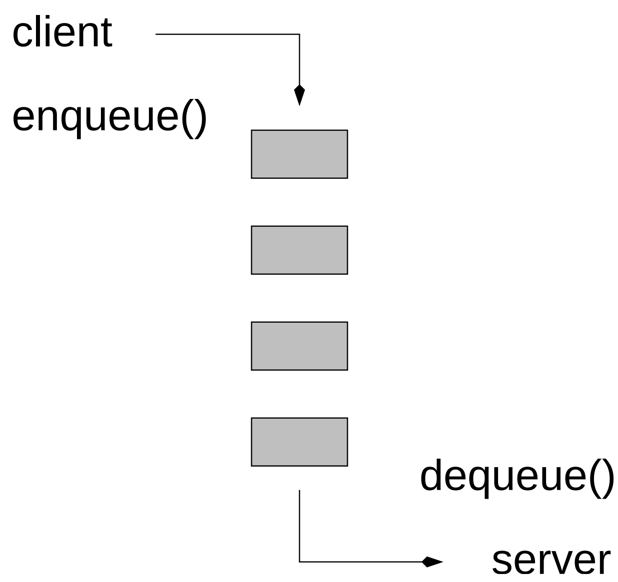


Inter-process communications




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

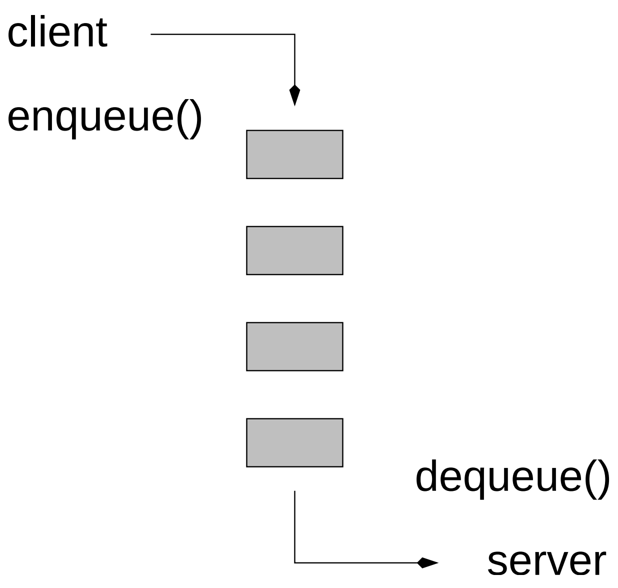
Inter-process communications




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

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

Inter-process communications




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

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

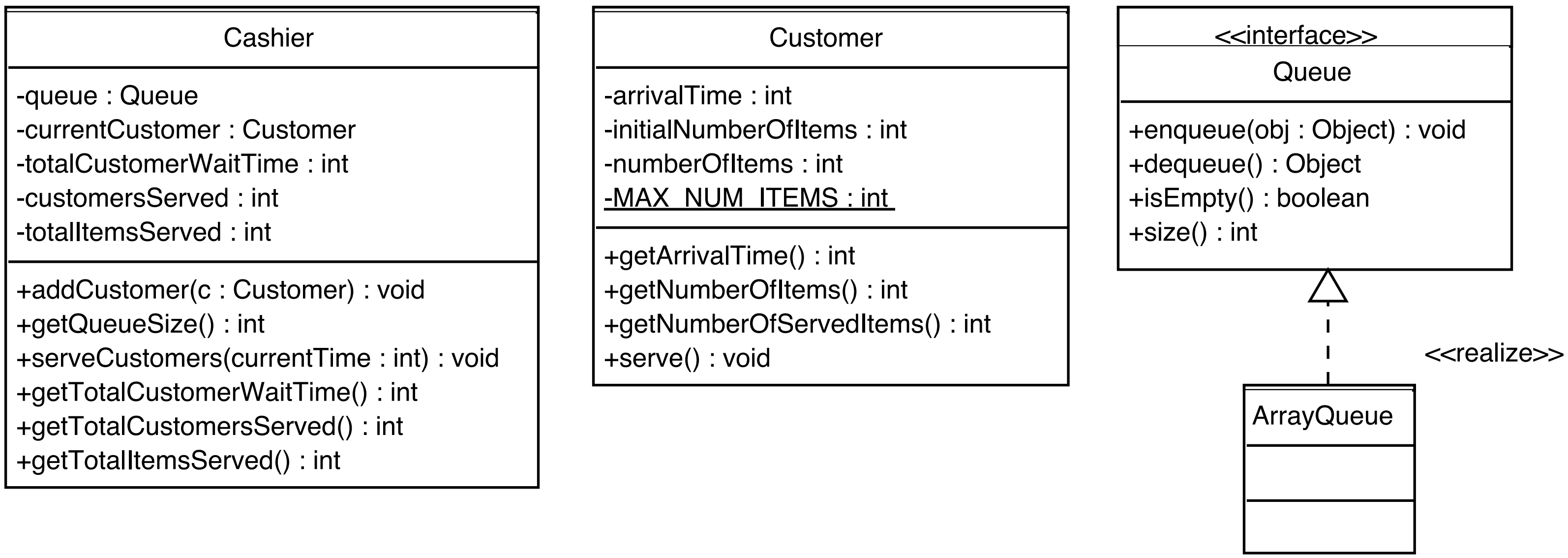
⇒ *inter-process communication* (IPC), *buffered i/o*, etc.

Applications

1. Simulations (clients arrival in a supermarket);

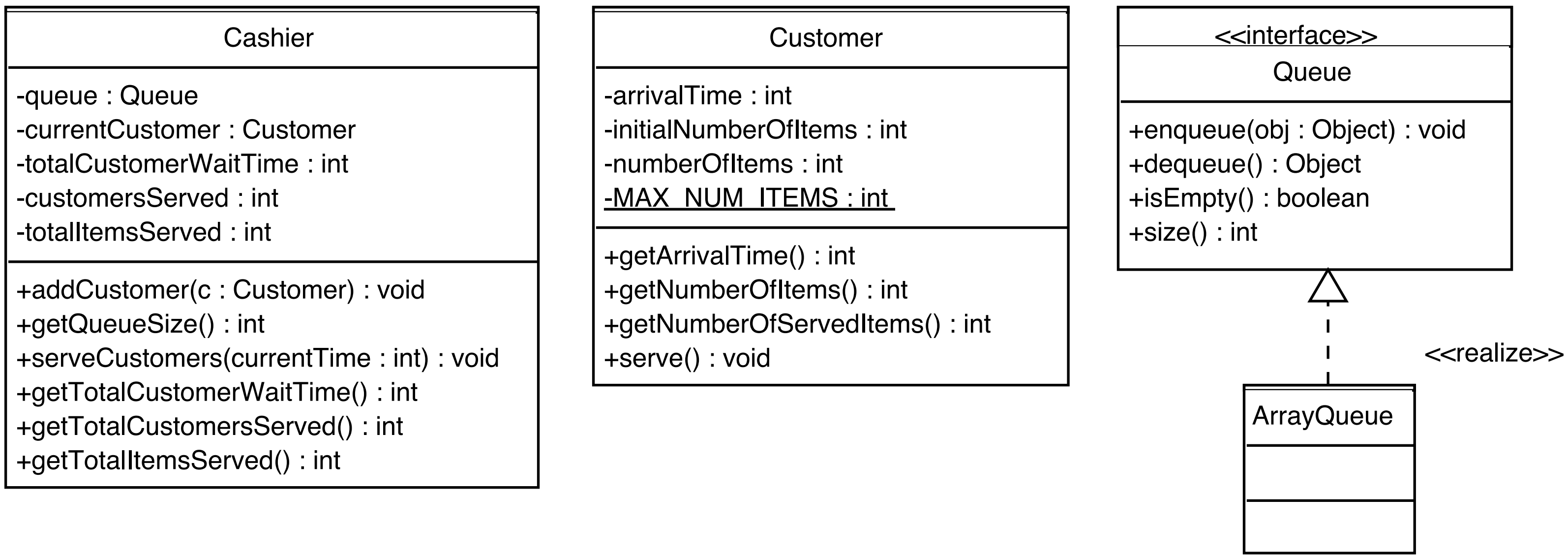
Applications

1. Simulations (clients arrival in a supermarket);



Applications

1. Simulations (clients arrival in a supermarket);



2. Queue-based algorithms (breadth-first search, labyrinth).

Queue-based algorithms

Algorithm:

1. enqueue “”
2. while true
   (a) s ← dequeue
   (b) enqueue “s + 0”
   (c) enqueue “s + 1”

What does the above algorithm do?

Queue-based algorithms

Algorithm:

1. enqueue “”
2. while true
   (a) s $\leftarrow$ dequeue
   (b) enqueue “s + 0”
   (c) enqueue “s + 1”

What does the above algorithm do?

It generates **all** the possible sequences of 0s and 1s **in increasing order of length**, 0, 1, 00, 01, 10, 11, 000, 001, . . .

Queue-based algorithms

Algorithm:

1. enqueue “”
2. while true
   (a) s $\leftarrow$ dequeue
   (b) enqueue “s + 0”
   (c) enqueue “s + 1”

What does the above algorithm do?

It generates **all** the possible sequences of 0s and 1s **in increasing order of length**, 0, 1, 00, 01, 10, 11, 000, 001, . . .

In other words the ensemble of all character strings, S, such that:

$$S \equiv [s \leftarrow \{0, 1, s' + 0, s' + 1\}; s' \in S]$$

```
new queue            []
enqueue(0)           [0]
enqueue(1)           [0,1]
s = 0 = dequeue()    [1]
enqueue(s+0 = 0+0)   [1,00]
enqueue(s+1 = 0+1)   [1,00,01]
s = 1 = dequeue()    [00,01]
enqueue(s+0 = 1+0)   [00,01,10]
enqueue(s+1 = 1+1)   [00,01,10,11]
s = 00 = dequeue()   [01,10,11]
enqueue(s+0 = 00+0)  [01,10,11,000]
enqueue(s+1 = 00+1)  [01,10,11,000,001]
s = 01 = dequeue()   [10,11,000,001]
enqueue(s+0 = 01+0)  [10,11,000,001,010]
enqueue(s+1 = 01+1)  [10,11,000,001,010,011]
s = 10 = dequeue()   [11,000,001,010,011]
enqueue(s+0 = 10+0)  [11,000,001,010,011,100]
enqueue(s+1 = 10+1)  [11,000,001,010,011,100,101]
```

Generalization

The generalization to sequences over any finite alphabet is trivial.

In particular, let's consider the following alphabet: $\Sigma = \{L, R, U, D\}$.

1. enqueue ""
2. while true
   (a) s $\leftarrow$ dequeue
   (b) enqueue "s + L"
   (c) enqueue "s + R"
   (d) enqueue "s + U"
   (e) enqueue "s + D"

```
new queue             []
enqueue("")           [""]
s = "" = dequeue()    []
enqueue(s+L = L)      [L]
enqueue(s+R = R)      [L,R]
enqueue(s+U = U)      [L,R,U]
enqueue(s+D = D)      [L,R,U,D]
s = L = dequeue()     [R,U,D]
enqueue(s+L = L+L)    [R,U,D,LL]
enqueue(s+R = L+R)    [R,U,D,LL,LR]
enqueue(s+U = L+U)    [R,U,D,LL,LR,LU]
enqueue(s+D = L+D)    [R,U,D,LL,LR,LU,LD]
s = R = dequeue()     [U,D,LL,LR,LU,LD]
enqueue(s+L = R+L)    [U,D,LL,LR,LU,LD,RL]
enqueue(s+R = R+R)    [U,D,LL,LR,LU,LD,RL,RR]
enqueue(s+U = R+U)    [U,D,LL,LR,LU,LD,RL,RR,RU]
enqueue(s+D = R+D)    [U,D,LL,LR,LU,LD,RL,RR,RU,RD]
s = U = dequeue()     [D,LL,LR,LU,LD,RL,RR,RU,RD]
enqueue(s+L = U+L)    [D,LL,LR,LU,LD,RL,RR,RU,RD,UL]
enqueue(s+R = U+R)    [D,LL,LR,LU,LD,RL,RR,RU,RD,UL,UR]
```

Let's give a meaning to those strings

What are those Ls, Rs, Us and Ds?

Let's give a meaning to those strings

What are those Ls, Rs, Us and Ds?

Let's say that each symbol of this alphabet corresponds to a direction:

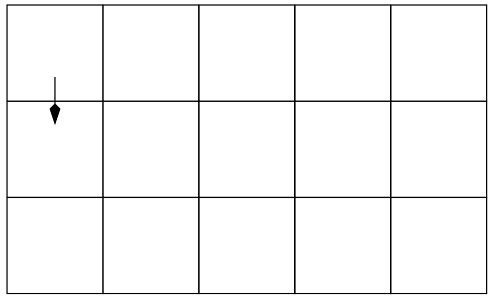
L = *left*;

R = *right*;

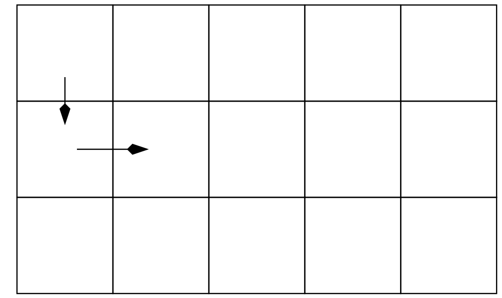
U = *up*;

D = *down*;

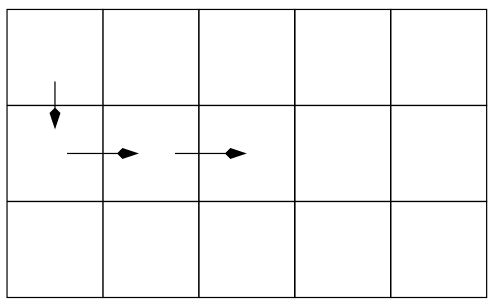
Each character string correspond to a **path** in a two dimensional plane.



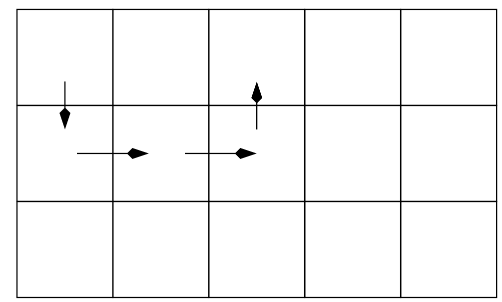
D



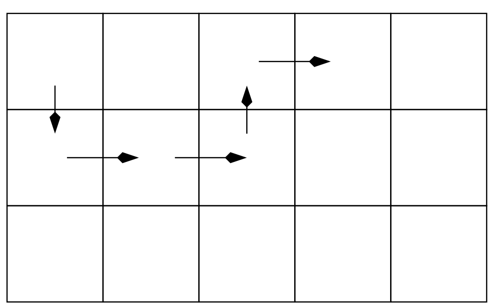
DR



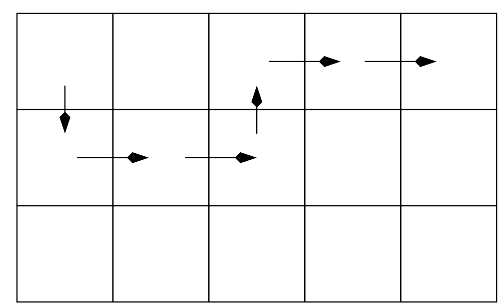
DRR



DRRU

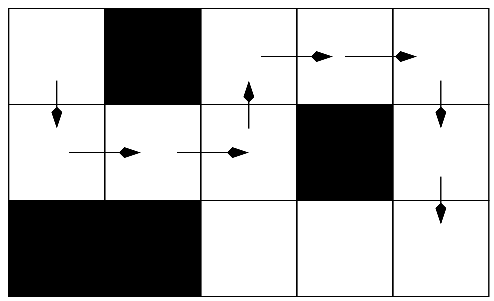


DRRUR

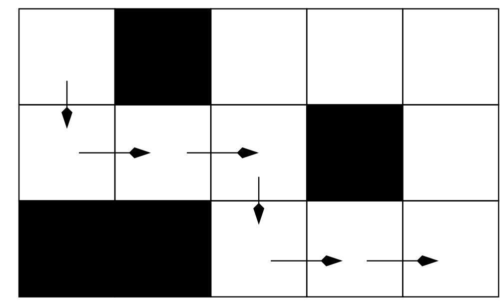


DRRURR

Adding obstacles



DRRURRDD



DRRDRR

⇒ What are the necessary modifications to our string generating algorithm so that it only generates valid paths? and finds the exit?

Auxiliary methods

Verifying that a path is valid: **checkPath(String path)**.

Has the exit been found: **reachesGoal(String path)**.

Data structures

A two dimensional array of characters:

char[][] maze;

A position is not accessible (wall) if it contains a '#', a cell is empty if ' ' and visited if '+'.

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

checkpath(String path)

```
private boolean checkPath( String path ) {

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

    int row, col;

    row = 0; // let’s assume that the entrance is found at (0,0)
    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 maze, not inside 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!

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;
}
```

Labyrinth

A queue-based algorithm to find a path through a labyrinth.

This algorithm has the property that it is guaranteed to find the shortest path if it exists!

Our queue-based algorithm to solve the maze problem is similar to our algorithm to generate all stings, in increasing order of length, over a finite-size alphabet.

```
q.enqueue( "" )
while ( true )
  s <- q.dequeue()
  for each char in the alphabet
      q.enqueue( s + char )
```

Our queue-based algorithm to solve the maze problem is similar to our algorithm to generate all stings, in increasing order of length, over a finite-size alphabet.

```
q.enqueue( "" )
while ( true )
  s <- q.dequeue()
  for each char in the alphabet
      q.enqueue( s + char )
```

⇒ The main difference being that the elements are filtered before being put into the queue — i.e. only valid prefixes are added to the rear of the queue.

Remarks

Our queue-based algorithm implements a state-space search known "breadth-first-search".

Remarks

Our queue-based algorithm implements a state-space search known "breadth-first-search".

Could this algorithm be using a stack?

Remarks

Our queue-based algorithm implements a state-space search known "breadth-first-search".

Could this algorithm be using a stack? Discuss the implications.

Remarks

Our queue-based algorithm implements a state-space search known "breadth-first-search".

Could this algorithm be using a stack? Discuss the implications.

The stack-based algorithm implements a "depth-first-search".

Remarks

Our queue-based algorithm implements a state-space search known "breadth-first-search".

Could this algorithm be using a stack? Discuss the implications.

The stack-based algorithm implements a "depth-first-search".

Why are these algorithms called "breadth-first-search" and "depth-first-search" respectively?

Remarks

Our queue-based algorithm implements a state-space search known "breadth-first-search".

Could this algorithm be using a stack? Discuss the implications.

The stack-based algorithm implements a "depth-first-search".

Why are these algorithms called "breadth-first-search" and "depth-first-search" respectively?

A variant of these algorithms is called beam-search and consists in limiting the number of solutions kept in the queue.

Remarks

Our queue-based algorithm implements a state-space search known "breadth-first-search".

Could this algorithm be using a stack? Discuss the implications.

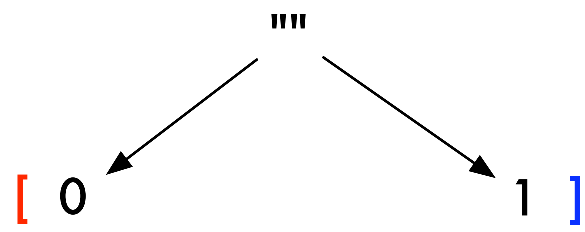
The stack-based algorithm implements a "depth-first-search".

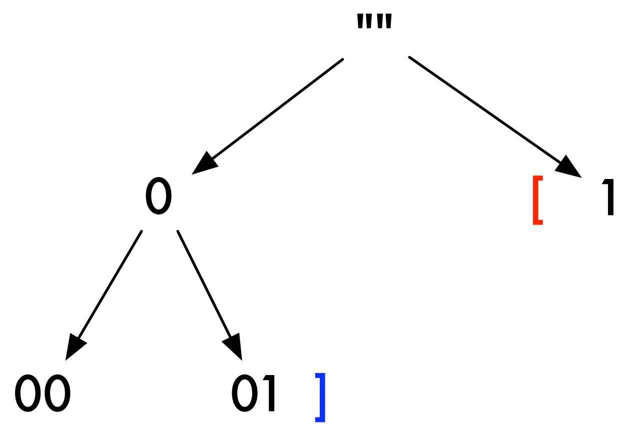
Why are these algorithms called "breadth-first-search" and "depth-first-search" respectively?

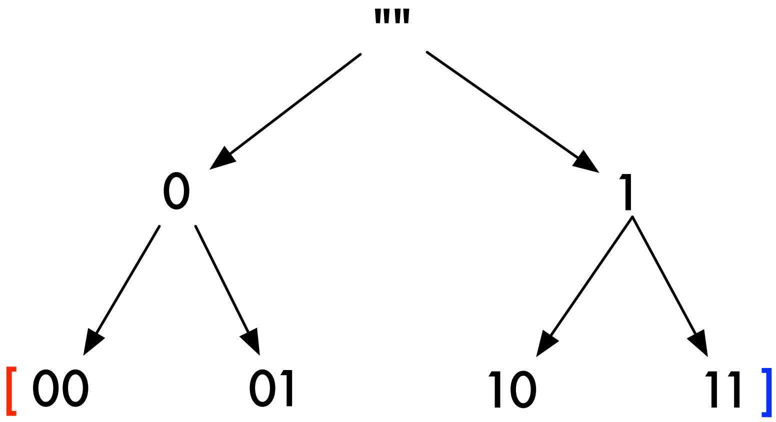
A variant of these algorithms is called beam-search and consists in limiting the number of solutions kept in the queue.

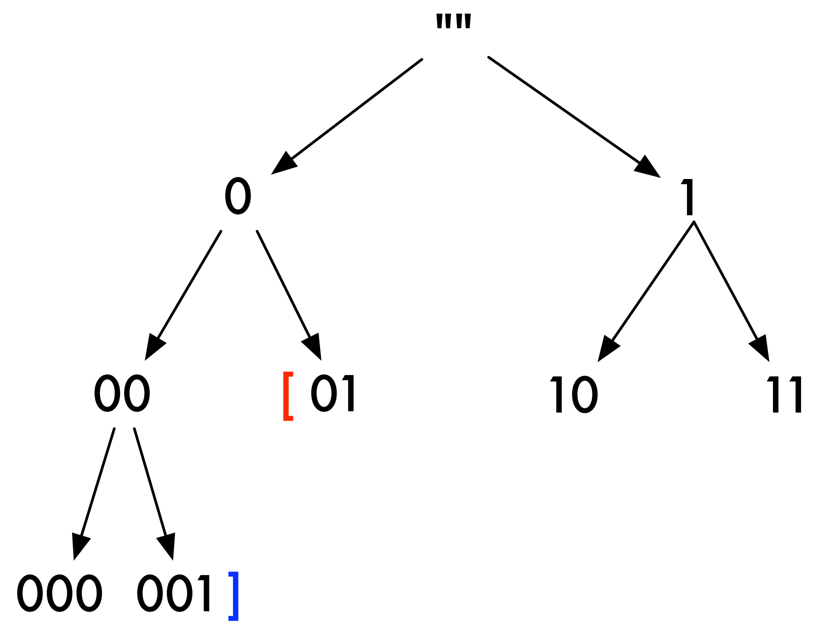
What would occur if no solution exist? How to detected such sitution?

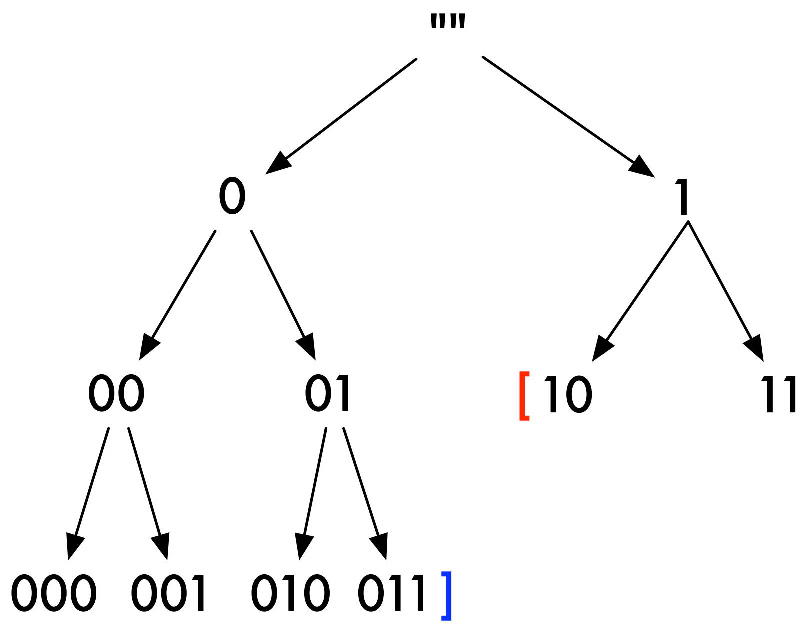
[""]

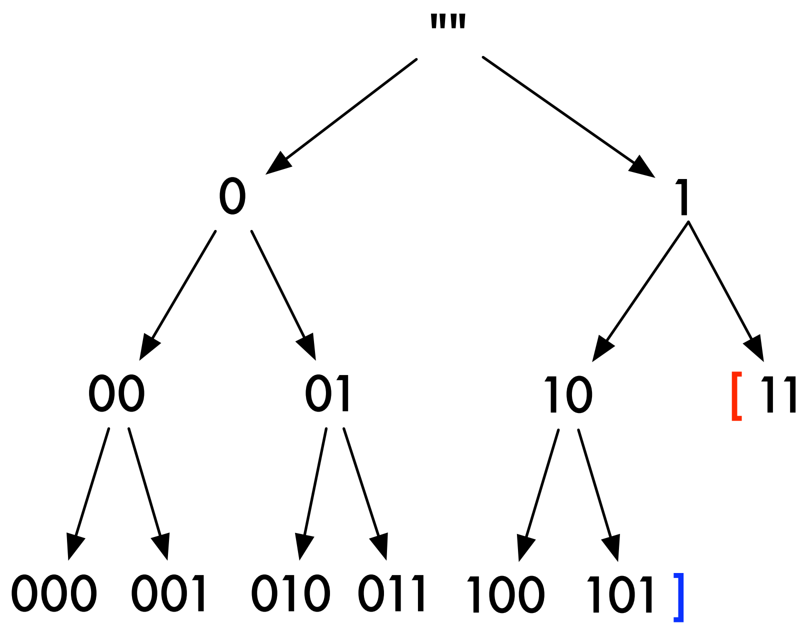

""
[0
1]

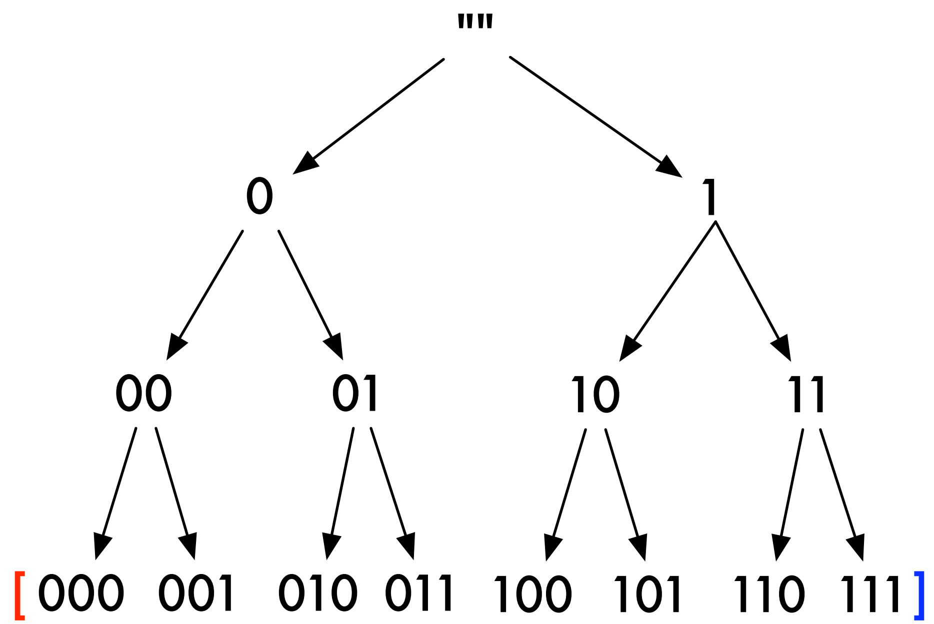

""
0
[
1
00
01
]

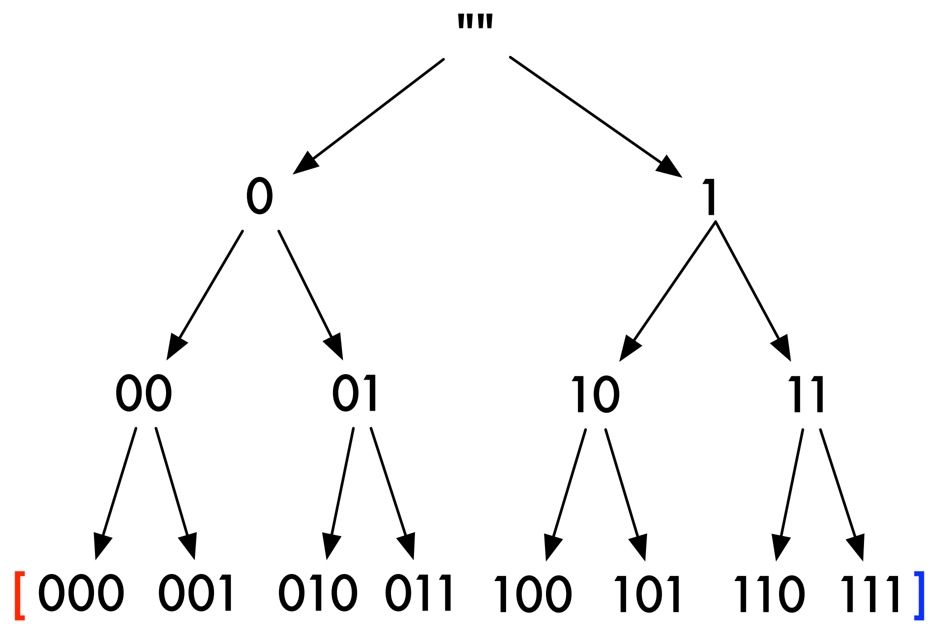

""
0
1
[00
01
10
11]


""
0
1
00
[01
10
11
000
001]


""
0
1
00
01
[10
11
000
001
010
011]


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

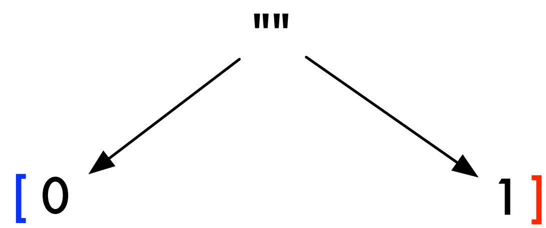

""
0
1
00
01
10
11
[000
001
010
011
100
101
110
111]

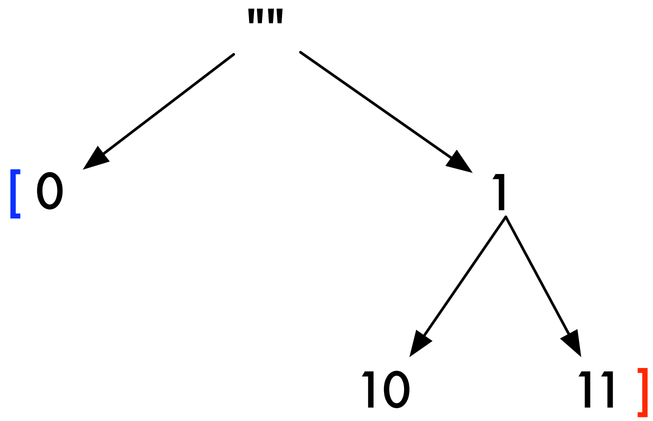


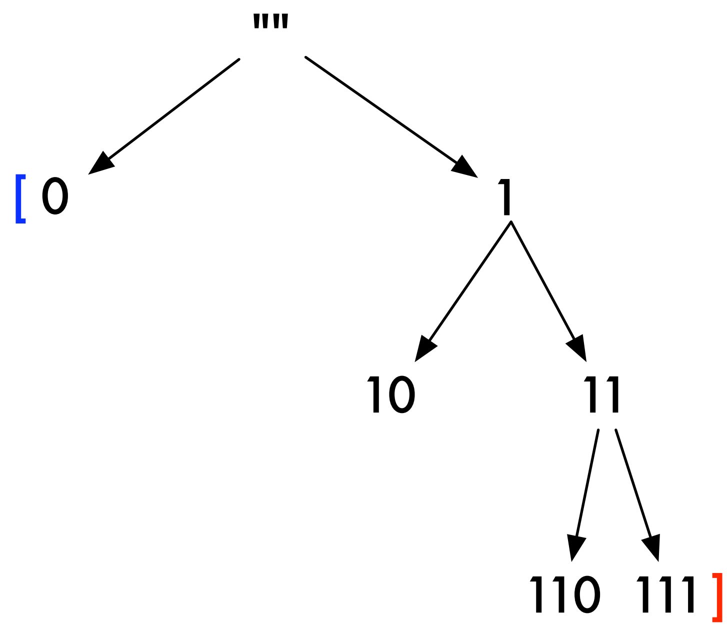
The **queue**-based implementation of the search is called "**breadth-first search**".

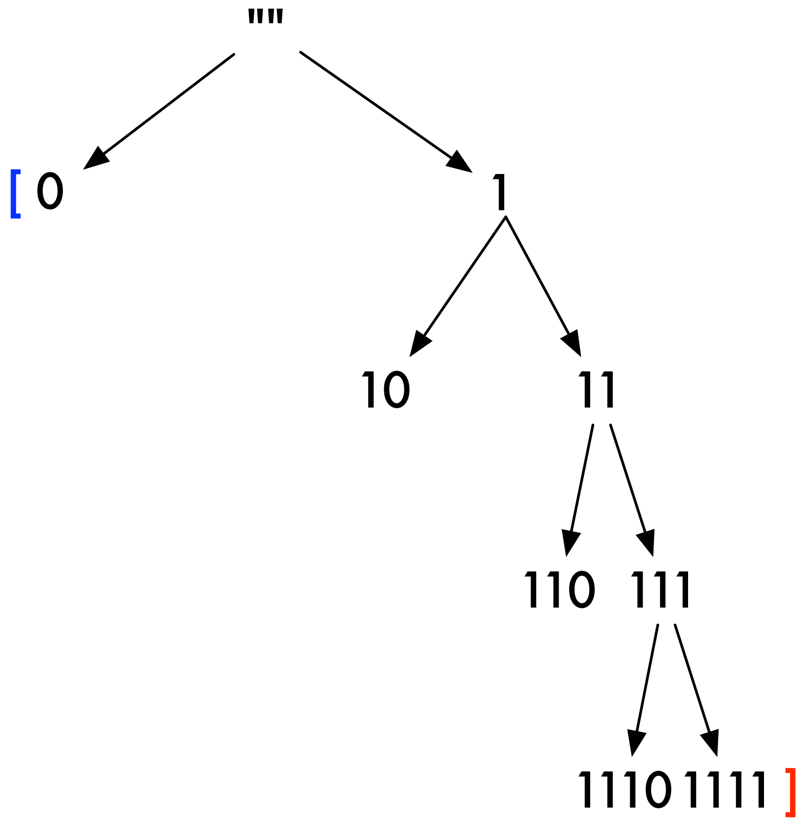
The **search tree** is built layer by layer, all the sequences on the same level (i.e. sequences of the same length) are processed before processing the sequences of the next level.

[""]

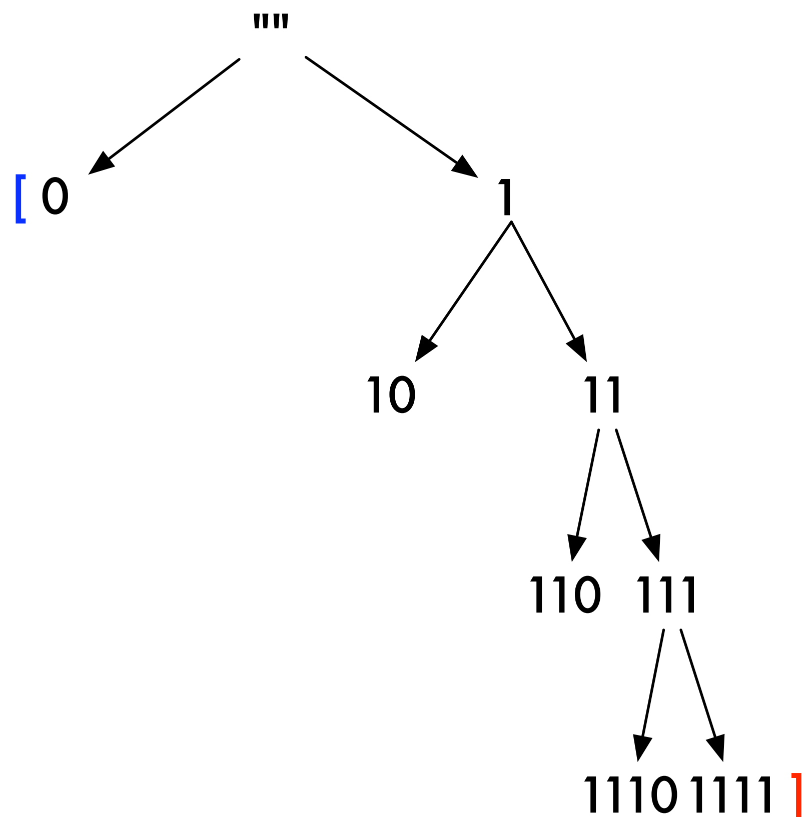

""
[0
1]


""
[0
1
10
11]


""
[0
1
10
11
110
111]



""
[0
1
10
11
110
111
1110
1111]



The **stack**-based implementation of the search is called "**depth-first search**".

The **search tree** is built branch by branch, a sequence is selected and repeatedly expanded until a dead-end occurs. The algorithm then backtracks to the next sequence onto the stack. Hence the surname **backtracking algorithm**.

```
#I#############
#    ####     #
## # #    ### #
# ##   ####   #
#    # ## # ###
## ### ## #  ##
#  ###     ####
## ######### ##
#            ##
############O##
```