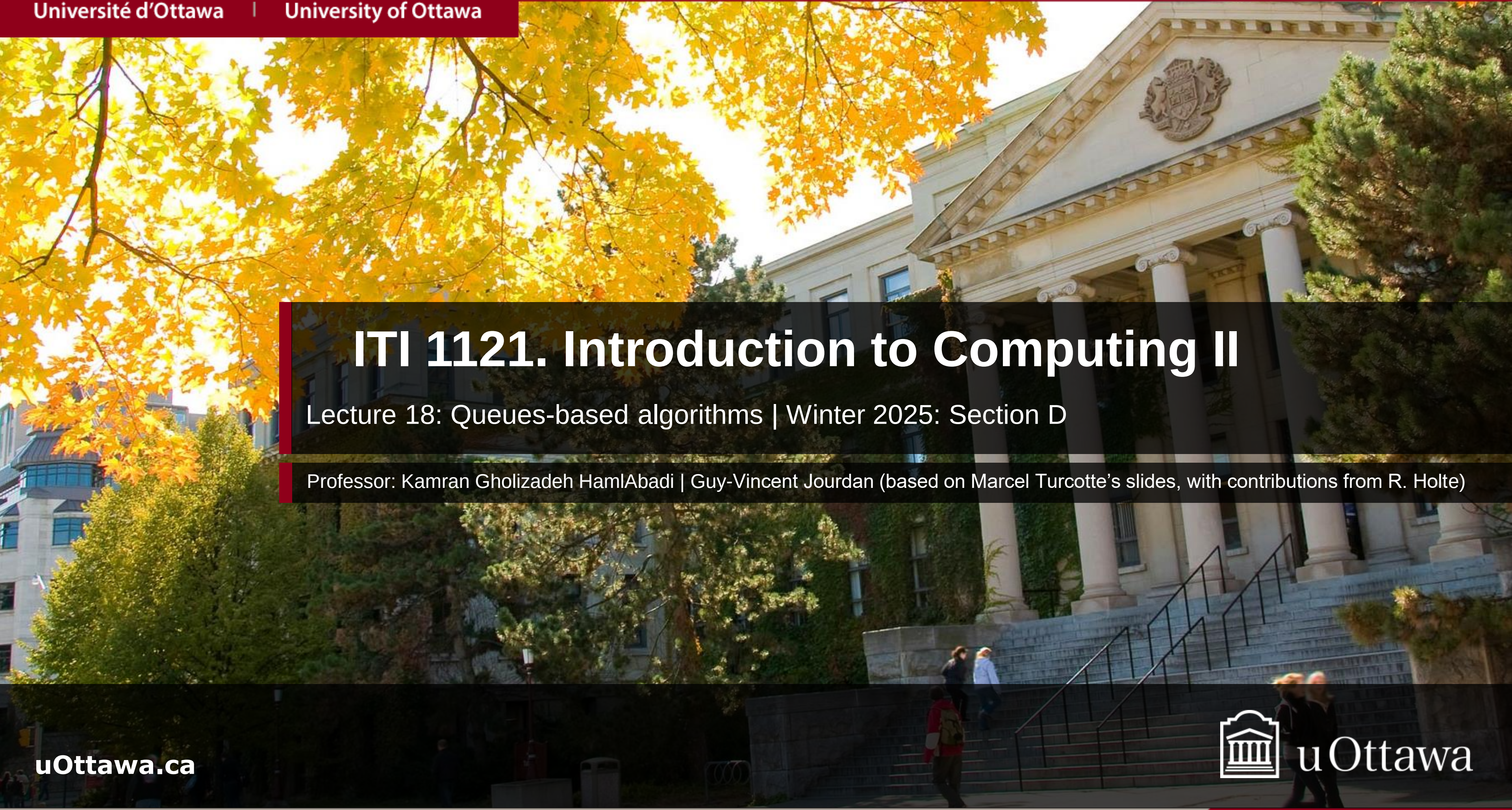
Université d'Ottawa | University of Ottawa



ITI 1121. Introduction to Computing II

Lecture 18: Queues-based algorithms | Winter 2025: Section D

Professor: Kamran Gholizadeh HamlAbadi | Guy-Vincent Jourdan (based on Marcel Turcotte's slides, with contributions from R. Holte)

uOttawa.ca

uOttawa

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

Key Applications

- Used in **producer/consumer**, **client/server**, and **sender/receiver** models.
- Enables **asynchronous data processing** to handle different speeds between sender and receiver.

What is Asynchronous Processing?

- The **client and server operate independently** without requiring real-time synchronization.
- If the server is **not ready or capable** of receiving data, it can process it later.

How It Works?

1. **Client inserts data into a queue** (enqueue).
2. **Server retrieves data from the queue** (dequeue) when it is ready.
3. The queue acts as a **buffer** to manage data flow efficiently.

Benefits

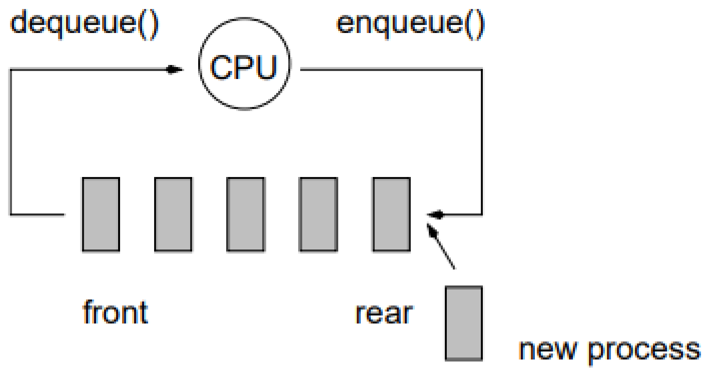
- Prevents **data loss** when the server is unavailable.
- Smoothens **data processing** despite varying speeds.
- Supports **scalability** in distributed systems.

Asynchronous Processes in Operating Systems

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

- **Printer Spooler**: Queues print jobs to be processed sequentially.
- **Buffered I/O**: Temporarily stores data to accommodate differences in processing speeds.
- **Disk Accesses**: Manages read/write operations asynchronously to optimize performance.
- **Network Communication**: Sends and receives data packets independently to prevent delays.

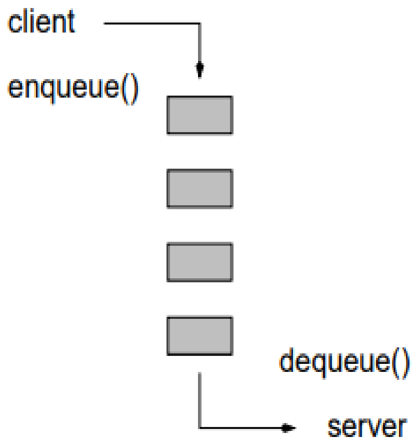
Time-shared applications



- The **CPU** executes multiple processes by switching between them.
- **New** processes are added to the **rear** of the queue (**enqueue**).
- The **CPU** takes the next process from the front of the queue (**dequeue**) for execution.

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



What is IPC?

- A way for processes to **communicate and share data** in an operating system.
- Used in **client-server** applications where one process (client) sends requests, and another process (server) processes them.

How It Works?

1. The **client** adds a message to a queue (**enqueue**).
2. The **server** retrieves messages when it is ready (**dequeue**).
3. The process repeats continuously to handle multiple requests.

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

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

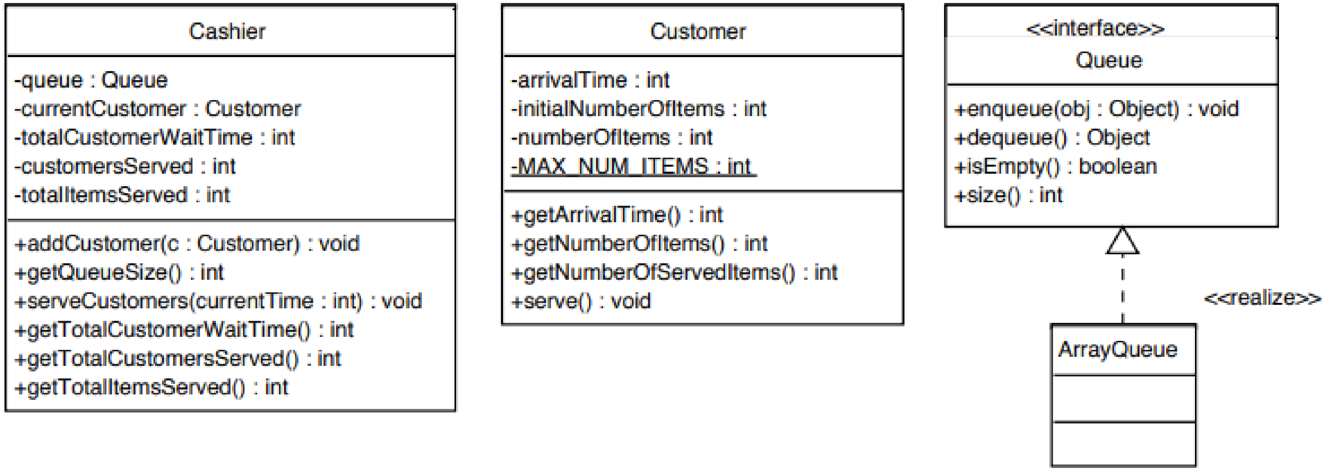
⇒ inter-process communication (IPC), buffered i/o, etc.

Applications (Supermarket Checkout System)

Cashier: Serves customers one by one from the queue.

Customer: Waits in line with a certain number of items.

Queue: Stores customers in order, ensuring First-In-First-Out (FIFO) processing.



Cashier
-queue : Queue
-currentCustomer : Customer
-totalCustomerWaitTime : int
-customersServed : int
-totalItemsServed : int
+addCustomer(c : Customer) : void
+getQueueSize() : int
+serveCustomers(currentTime : int) : void
+getTotalCustomerWaitTime() : int
+getTotalCustomersServed() : int
+getTotalItemsServed() : int
Customer
-arrivalTime : int
-initialNumberOfItems : int
-numberOfItems : int
-MAX_NUM_ITEMS : int
+getArrivalTime() : int
+getNumberOfItems() : int
+getNumberOfServedItems() : int
+serve() : void
<<interface>>
Queue
+enqueue(obj : Object) : void
+dequeue() : Object
+isEmpty() : boolean
+size() : int
<<realize>>
ArrayQueue

Queue-Based Algorithm: Generating Binary Numbers

Algorithm:

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

Steps:

1. **Start with an empty string ("") in the queue.**
2. **Repeat forever:**
   - Remove (dequeue) the front element.
   - Add (enqueue) the element with "0" added at the end.
   - Add (enqueue) the element with "1" added at the end.

How It Works ? This algorithm generates binary numbers in sequence using a queue. 0, 1, 00, 01, 10, 11, 000, 001, . . .
In other words the ensemble of all character strings, S, such that:

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

Step	Action	Queue After Action
1	Start with an empty queue	[]
2	Enqueue 0	[0]
3	Enqueue 1	[0, 1]
4	Dequeue 0	[1]
5	Enqueue 0 + 0 = 00	[1, 00]
6	Enqueue 0 + 1 = 01	[1, 00, 01]
7	Dequeue 1	[00, 01]
8	Enqueue 1 + 0 = 10	[00, 01, 10]
9	Enqueue 1 + 1 = 11	[00, 01, 10, 11]
10	Dequeue 00	[01, 10, 11]
11	Enqueue 00 + 0 = 000	[01, 10, 11, 000]
12	Enqueue 00 + 1 = 001	[01, 10, 11, 000, 001]
13	Dequeue 01	[10, 11, 000, 001]
14	Enqueue 01 + 0 = 010	[10, 11, 000, 001, 010]
15	Enqueue 01 + 1 = 011	[10, 11, 000, 001, 010, 011]

Generalized Queue-Based Algorithm

The generalization to sequences over any finite alphabet is trivial.
In particular, let's consider the following alphabet: $\Sigma = \{L, R, U, D\}$.

Steps:

1. **Start with an empty string ("") in the queue**
2. **Repeat indefinitely:**
   - Remove (dequeue) the front element s.
   - Add (enqueue) new sequences by appending different directions:
     - s + "L" → Left
     - s + "R" → Right
     - s + "U" → Up
     - s + "D" → Down

Step	Action	Queue After Action
1	Start with an empty queue	[]
2	Enqueue ""	[""]
3	Dequeue ""	[]
4	Enqueue L	[L]
5	Enqueue R	[L, R]
6	Enqueue U	[L, R, U]
7	Enqueue D	[L, R, U, D]
8	Dequeue L	[R, U, D]
9	Enqueue LL	[R, U, D, LL]
10	Enqueue LR	[R, U, D, LL, LR]
11	Enqueue LU	[R, U, D, LL, LR, LU]
12	Enqueue LD	[R, U, D, LL, LR, LU, LD]
13	Dequeue R	[U, D, LL, LR, LU, LD]
14	Enqueue RL	[U, D, LL, LR, LU, LD, RL]
15	Enqueue RR	[U, D, LL, LR, LU, LD, RL, RR]
16	Enqueue RU	[U, D, LL, LR, LU, LD, RL, RR, RU]
17	Enqueue RD	[U, D, LL, LR, LU, LD, RL, RR, RU, RD]

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:

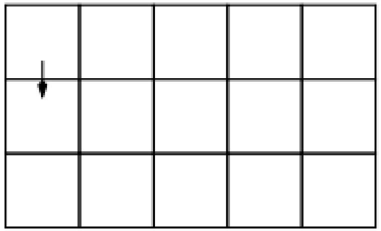
L = left;

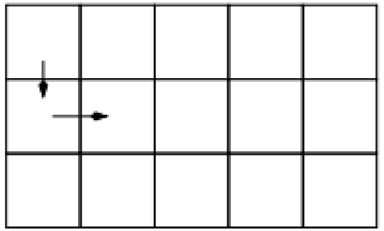
R = right;

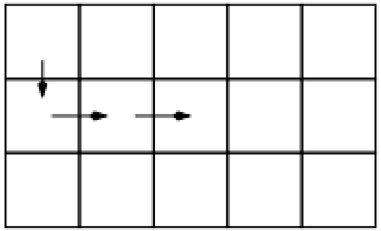
U = up;

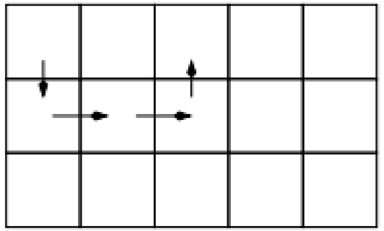
D = down;

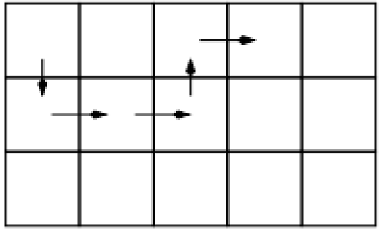
Each character string correspond to a **path** in a two-dimensional plane.

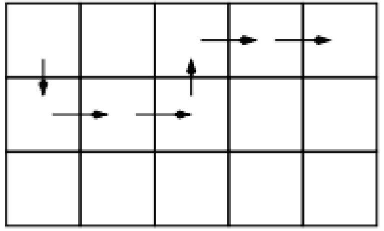

D


DR

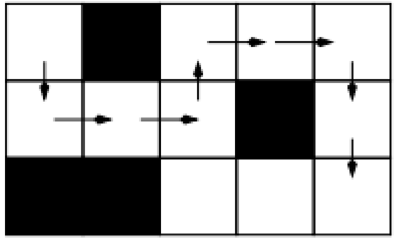

DRR

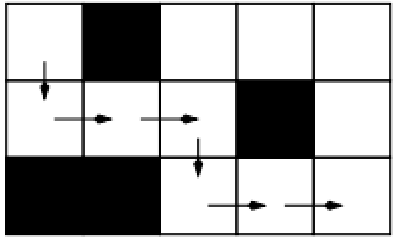

DRRU


DRRUR


DRRURR

Adding obstacles


DRRURRDD


DRRDRR

- Some paths **collide with obstacles** (black squares). The algorithm must be modified to:
  - **Avoid obstacles** (invalid moves).
  - **Ensure paths reach the exit** efficiently.
  - **Discard paths that lead to dead ends**.

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

Auxiliary methods

- **Verifying Path Validity: checkPath(String path):**

This method checks if a given path is valid based on the following conditions:

- The path stays within the grid boundaries.
- The path does not pass through obstacles.
- The path follows valid movement rules.

Example: "DRRU" → This path is checked to ensure it does not collide with obstacles or go out of bounds.

- **Checking Goal Completion: reachesGoal(String path):**
  - This method determines whether the current path leads to the exit point.
  - Does the path reach the target coordinates in the grid?
  - Is this path the shortest or most efficient way to reach the goal?

Example: reachesGoal("DRRU") == true → This means "DRRU" successfully reaches the exit.

Understanding Data Structures in Pathfinding

- **What does this grid represent?**

This grid is a maze representation where:

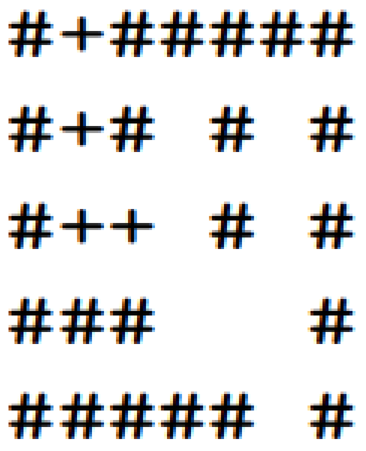
- # represents walls or obstacles (not passable).
- **+** represents a valid path found by an algorithm.
- **Empty** spaces can be traversed.

- **How Does This Relate to Auxiliary Methods?**

1. Checking if a Path is Valid → **checkPath(String path)**
   - The algorithm verifies whether the movement stays within boundaries and does not hit a **# (wall).**
   - If a movement is valid, continue exploring.
   - If a movement hits a wall, discard the path.

Finding the Exit → **reachesGoal(String path)**

1. The algorithm **keeps track of the visited positions** and checks if the **exit is reached**.
2. If the **goal is found**, the path is marked with +.
   - If a path **leads to the exit**, it is part of the solution.
   - If a path is **blocked**, it is discarded.



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

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!

```
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!**

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

```
// Initialize the queue with an empty string
q.enqueue("")

// Continuously generate new sequences
while (true) {
    // Dequeue the front element
    s ← q.dequeue()

    // Iterate through each character in the alphabet
    for each char in alphabet {
        // Enqueue the new string formed by appending char to s
        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.

Labyrinth

- ➢ Our queue-based algorithm implements a state-space search known "breadthfirst-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.

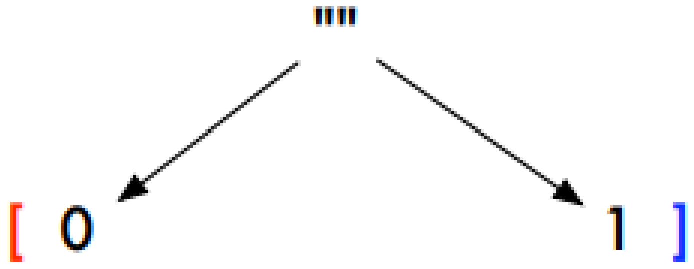
What would occur if no solution exist? How to detected such situation?

Breadth-first-search Algorithm

[""]

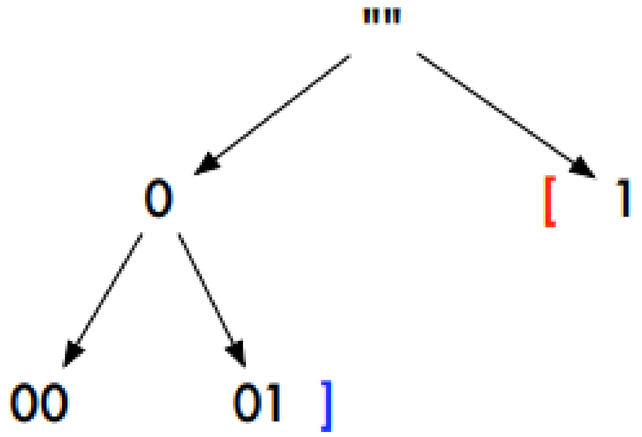
It shows the initial state where the queue contains an empty string [""].

Expanding the First Node



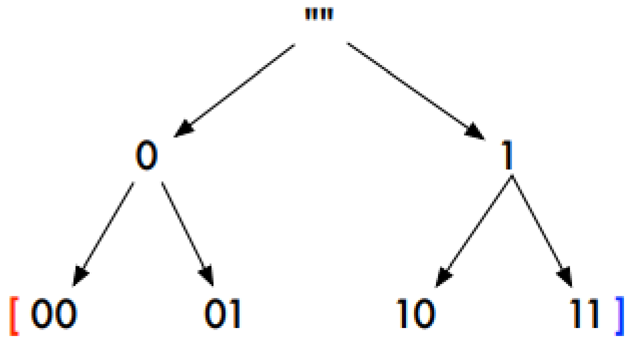
- ➢ The root node "" expands into two possible paths: "0" and "1".
- ➢ These represent the first level of binary string generation.
- ➢ The queue now contains ["0", "1"].

Expanding "0"



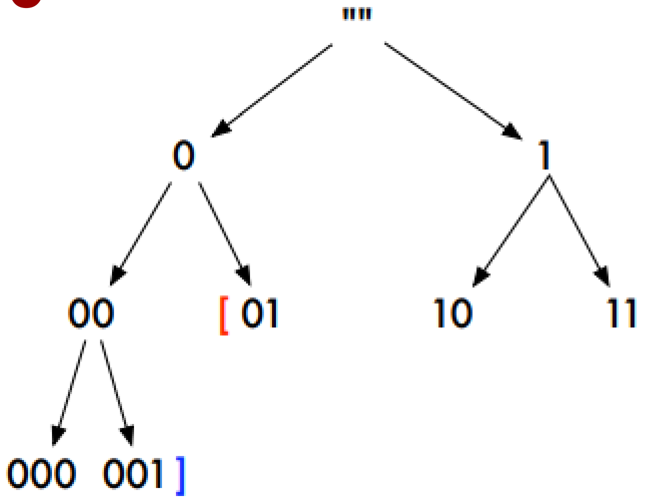
- The path "0" expands into "00" and "01", continuing BFS.
- "1" is still in the queue, waiting for expansion.
- The queue now contains ["1", "00", "01"].

Expanding "1"



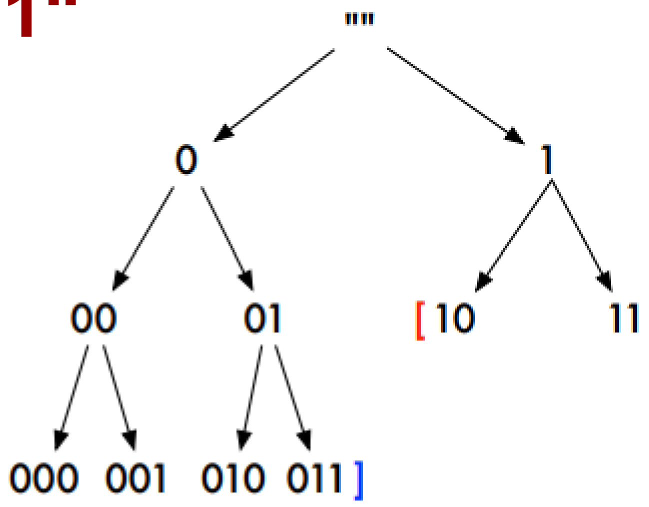
- "1" expands into "10" and "11", covering all paths of length 2.
- The queue now contains ["00", "01", "10", "11"].

Expanding "00"



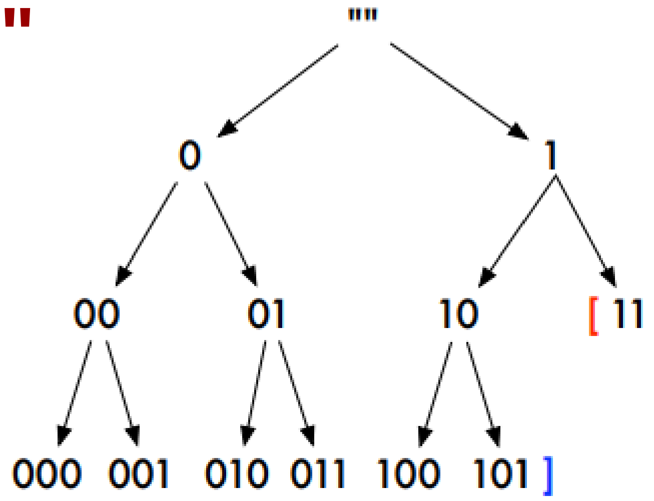
- "00" expands into "000" and "001".
- The queue now contains ["01", "10", "11", "000", "001"].

Expanding "01“



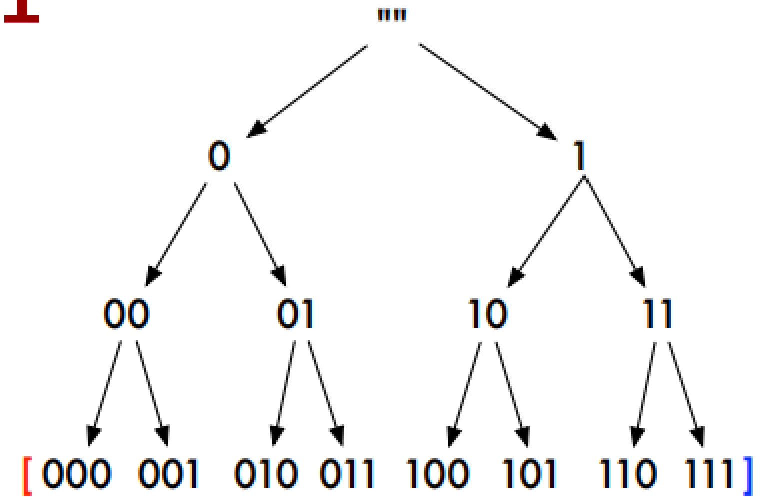
- ➢ "01" expands into "010" and "011".
- ➢ The queue now contains ["10", "11", "000", "001", "010", "011"].

Expanding "10"



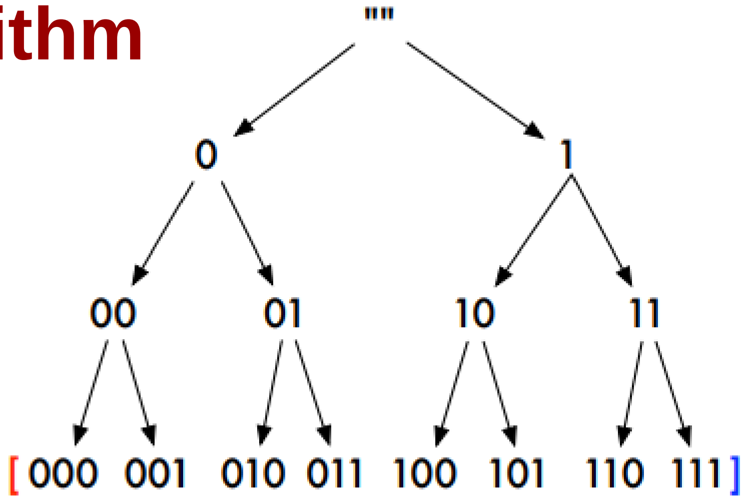
- "10" expands into "100" and "101".
- The queue now contains ["11", "000", "001", "010", "011", "100", "101"].

Expanding "11"



- ➢ "11" expands into "110" and "111", completing all possible paths.
- ➢ The queue now contains ["000", "001", "010", "011", "100", "101", "110", "111"].

Breadth-first-search Algorithm



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

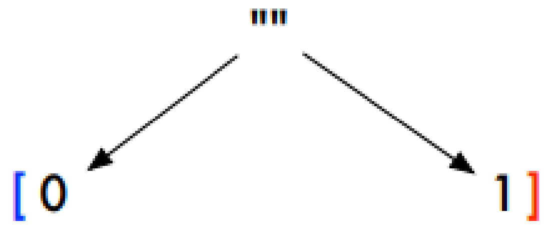
The **search tree** is built layer by layer, all the sequences on the same level (i.e. sequences of the same length) are processed before processing the sequences of the next level.

Depth-first search Algorithm

[""]

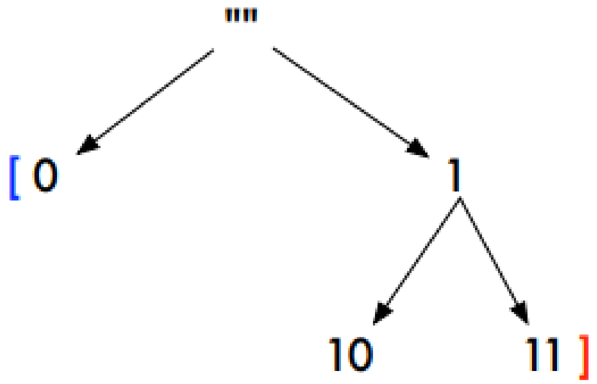
- DFS follows a **stack-based** approach, meaning it **explores one branch fully before backtracking**.
- The queue notation [""] represents that we start with an empty sequence.

Expanding the First Level



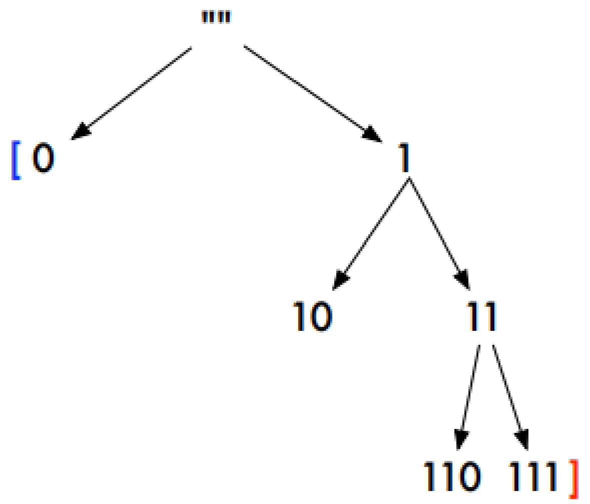
- From "", DFS expands the first branch, adding **"0"** and **"1"** as possible sequences.
- The notation [0] (blue) represents the **next element DFS will process**.
- [1] (red) represents the **element that will be processed after backtracking**.

Exploring "1" First



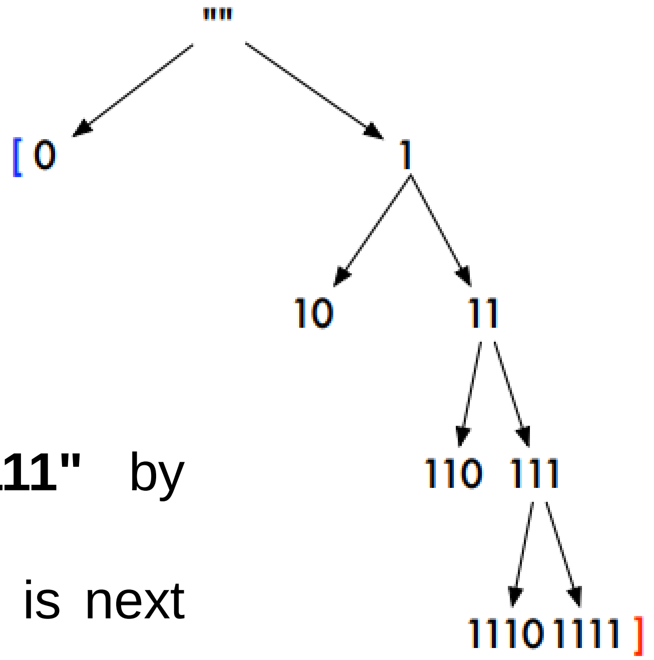
- Unlike BFS (which processes level by level), DFS **fully explores a path before backtracking**.
- DFS selects **"1"** (since it was the last inserted element in the stack).
- It expands **"1"** to "10" and "11".

Expanding "11" First



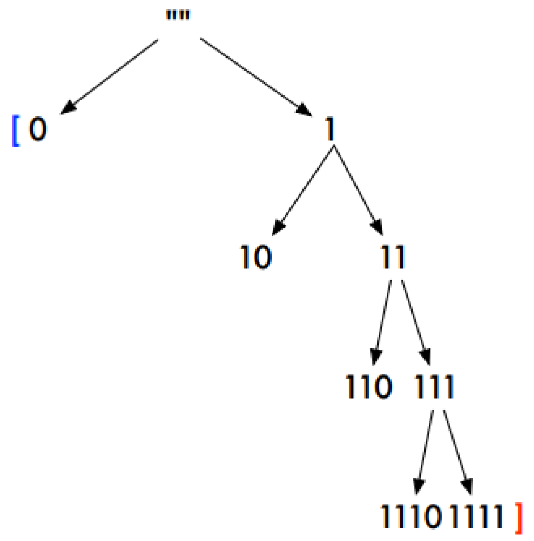
- DFS expands **"11"**, adding "110" and "111".
- The last inserted element **"111"** is the next to be processed.

Expanding "111" First



- DFS continues **expanding "111"** by adding "1110" and "1111".
- The last inserted element "1111" is next to be explored.

Depth-first search Algorithm



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##
```