Imperial College London

EE1-08/EE2-18 Soft. Eng. I – Algorithms & Data Structures

Coursework Assignment 1 – Spring 2019

Sahbi Ben Ismail (s.ben-ismail@imperial.ac.uk)

An Amazing Maze

This assignment deals with the representation of mazes as data structures, and basic algorithms to solve them.

Consider the maze displayed in Figure 1 below. The circles, called nodes, can be thought of as rooms. The lines, called edges, connect one room to another. Note that each node has at most four edges in the maze, corresponding to the four directions North, South, East, and West.

Solving a maze consists in reaching the `finish` node (`L` here), starting from the `start` node (`A` here).

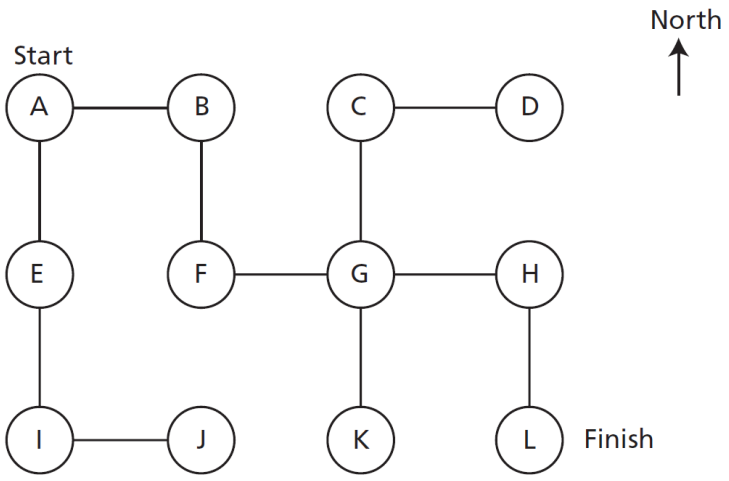

Figure 1: An example of maze.

Consider the following definition for a corresponding data structure in C++:

```
struct node {
  char id;
  node* north;
  node* south;
  node* east;
  node* west;
};

typedef node* nodeptr;
```

Write an implementation for the following functions:

```
void build_basic_maze(nodeptr& start, nodeptr& finish);
// builds the basic maze displayed in Figure 1

nodeptr traverse_maze(const nodeptr& start, const std::string& ↩
    path);
// traverses the maze using a predefined path

void solve_interactively(const nodeptr& start, const nodeptr& ↩
    finish, std::string& path);
// allows the user to solve the maze interactively

nodeptr random_walk(const nodeptr& start, const nodeptr& finish↩
    , std::string& path);
// simulates a random walk in the maze

void solve_queue(const nodeptr& start, const nodeptr& finish, ↩
    std::string& path);
// an improvement of the random walk: solves the maze using a ↩
    queue to store the unvisited neighbours of the current node

void solve_stack(const nodeptr& start, const nodeptr& finish, ↩
    std::string& path);
// an improvement of the random walk: solves the maze using a ↩
    stack to store the unvisited neighbours of the current node
```

Maze building

The function `build_basic_maze` dynamically allocates the memory for all the nodes of the maze, and stores the start and the finish in the input/output

arguments `start` and `finish`.

For example, we could have this code in the `main`:

```
nodeptr start = NULL;
nodeptr finish = NULL;

build_basic_maze(start, finish);
nodeptr current = start;

std::cout << "start = " << start->id << std::endl;
// displays start = A

std::cout << "finish = " << finish->id << std::endl;
// displays finish = L
```

A path in the maze

The input for function `traverse_maze` are a pointer to the start of the maze, and a string of characters representing a path. The function traverses the maze according to the path. The function returns a pointer to the last visited node.

For example, we could have this code in the `main`:

```
std::string path = "ESEES";
// East, then South, then East, then East, then South.
nodeptr stop = traverse_maze(start, path);
// starting from node A, stop should point to the node L

path = "SSE";
// South, then South, then East.
nodeptr stop = traverse_maze(start, path);
// starting from node A, stop should point to the node J
```

Maze interactive solving (by the user)

The function `solve_interactively` allows the user to solve the maze by entering a serie of directions (one character each step: 'N', 'S', 'E', or 'W'). The function finishes when the user reaches the finish node. The path used by the user is stored in the input/output argument `path`.

For example, we could have this interaction scenario in the terminal:

```
You are in room A of the maze.
You can go: (S)outh, (E)ast, or (Q)uit.
E

You are in room B of the maze.
You can go: (S)outh, (W)est, or (Q)uit.
S

You are in room F of the maze.
You can go: (N)orth, (E)ast, or (Q)uit.
E

You are in room G of the maze.
You can go: (N)orth, (S)outh, (E)ast, (W)est, or (Q)uit.
N

You are in room C of the maze.
You can go: (S)outh, (E)ast, or (Q)uit.
E

You are in room D of the maze.
You can go: (W)est, or (Q)uit.
W

You are in room C of the maze.
You can go: (S)outh, (E)ast, or (Q)uit.
S

You are in room G of the maze.
You can go: (N)orth, (S)outh, (E)ast, (W)est, or (Q)uit.
E

You are in room H of the maze.
You can go: (S)outh, (W)est, or (Q)uit.
S

Congratulations, you reached the finish!
```

Random walk in the maze

The function `random_walk` simulates a random walk in the maze. From the current node, the function makes a random choice on the direction to take, and moves in that direction. The path used by the function is stored in the input/output argument `path`.

Note that this function might produce an infinite loop by revisiting the same nodes again and again: $A \to B \to A \to B \to A \to B \to A \ \ldots$ for example. Make an appropriate choice to stop the random walk if it takes too many steps to solve the maze.

Maze solving using a queue

An improvement of the random walk may consist in memorizing the unvisited neighbours of the current node, and try to visit them one by one in order to reach the finish. The function `solve_queue` uses a queue data structure to store the unvisited neighbours. The path used by the function is stored in the input/output argument `path`.

You can use the `C++` built-in data structure queue (`#include` <queue>).

Make the current node point to the start of the maze. Take the following actions, then repeat:

- If the current node points to the finish, stop.
- Mark the current node as visited.
- Add all unvisited neighbours to the north, south, east, and west to a queue.
- Remove the next element from the queue and make it the current node.

In which order will the nodes of the sample maze be visited?

Notes:

- Your algorithm should work with any pair of start and finish nodes, not just nodes A and L.
- Your algorithm should also work if there are loops, such as a connection between nodes E and F.

Maze solving using a stack

Similarily to `solve_queue`, the function `solve_stack` uses a stack data structure to store the unvisited neighbours. The path used by the function is stored in the input/output argument `path`.

You can use the `C++` built-in data structure stack (`#include` <stack>).

In which order will the nodes of the sample maze be visited?

Appendix: Queues and Stacks

The figures 2 and 3 below illustrate the two important routines for a queue and a stack: `push` (add an element) and `pop` (remove an element).

A queue supports a **FIFO** (first in, first out) data access: the data items are removed from the queue in the same order that they were added to the queue.

A stack supports a **LIFO** (last in, first out) data access: the data items are removed from the stack in the reverse of the order in which the data is stored.

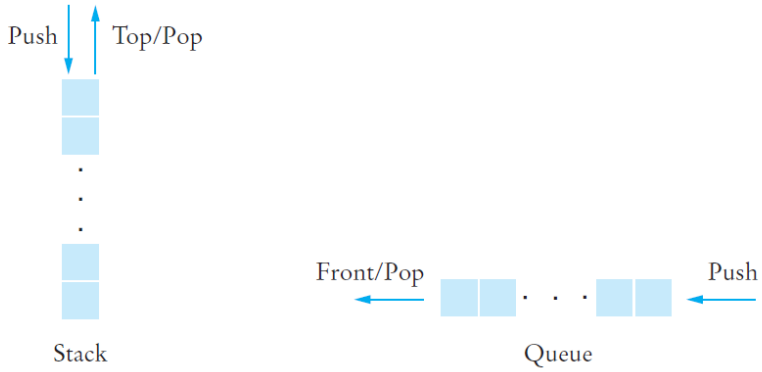

Figure 2: Behaviour of a stack and a queue, the principle of push/pop.

(a) Stack (b) Queue

Figure 3: Behaviour of a stack and a queue, an illustration of push/pop.