THE HONG KONG UNIVERSITY OF SCIENCE & TECHNOLOGY
Department of Computer Science & Engineering

**COMP 2012H: Honors OOP and Data Structures**
**Fall 2015**

# Midterm Examination

Date: Saturday, 17 October 2015
Time: 10:30am – 12:00pm
Instructor: Gary Chan
Venue: LT-E

---

- **This is a closed-book examination. However, you are allowed to bring with you a piece of A4-sized paper with notes written, drawn or typed on both sides for use in the examination.**
- **Your answers will be graded according to correctness, efficiency, precision, and clarity.**
- **During the examination, you must put aside your calculators, mobile phones, tablets and all other electronic devices. All mobile phones must be turned off.**
- **This booklet consists of a total of 14 single-sided pages. Please check that all pages are properly printed. You may use the reverse side of the pages for your rough work. If you decide to use the reverse side to continue your work, please indicate so and clearly write down the question number.**

---

**Student name:** _____ **English nickname (if any):** _____

**Student ID:** _____ **ITSC email:** _____@ust.hk

I have not violated the Academic Honor Code in this examination (signature): _____

| Question | Your score | Maximum score |
|---|---|---|
| 1 | | 25 |
| 2 | | 35 |
| Total | | 60 |

1. **A Skiing Problem** *(25 points total)*

Gary enjoys snow-skiing. He likes challenging seemingly unreachable places, or visiting the deepest descend from his starting point in a certain number of steps. In this problem, you will help him planning his ski routes using the programming techniques you learn in COMP2012H.

The skiing area is represented by a 2D array with numbers (integer) in each cell indicating the height (or altitude) of that location, i.e., `map[i][j]` stores the height of `(i, j)` in the map, where `i` is the row index ($i \geq 0$) and `j` is the column index ($j \geq 0$). The heights are always greater than zero. The cell with value "-1" indicates a Danger Zone, which is full of hard rocks, obstacles or trees that one must not enter. Figure 1 shows an example of a 5x5 skiing area.

| row \ col | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 100 | 100 | 200 | 200 | 300 |
| 1 | 200 | 100 | 100 | -1 | 200 |
| 2 | 150 | 100 | 300 | 200 | -1 |
| 3 | 300 | 200 | 100 | 100 | 100 |
| 4 | -1 | -1 | 100 | 200 | 300 |

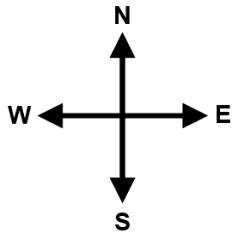


Figure 1: A skiing area and its possible skiing directions.

From any cell in the map, Gary may ski to an adjacent location in one of the four directions, North, South, East or West, with the following three constraints:

- He cannot ski upward (i.e., he can ski only to the cell of the same or lower altitude);
- He cannot enter into any of the Danger Zone; otherwise he will die;
- He cannot go beyond the skiing boundary (i.e., out of the map).

Given our example of Figure 1, Gary **MAY**

- ski from (1,0) towards North;
- ski from (0,1) towards South;

but he **MUST NOT**:

- ski from (1,2) towards North, because (0,2) is higher than (1,2);
- ski from (2,3) towards East, because (2,4) is a Danger Zone;
- ski from (0,0) towards West, because it is out of the map boundary.

In this question, an enumeration `Direction` is defined in the source code as follows:

```
enum Direction{NORTH, SOUTH, EAST, WEST};
```

Furthermore, the maximum number of rows and columns of the map are defined as `MAX_ROW` and `MAX_COL`, respectively.

(a) *(5 points)* Write a function with signature (prototype)

```
bool canSkiTowards(int map[MAX_ROW][MAX_COL], int row, int col,
Direction dir);
```

that returns true if Gary can ski from `(row, col)` towards direction `dir`, and returns false otherwise. You may assume that the location `(row, col)` is always a valid cell on the map (i.e., it is not a Danger Zone nor outside the boundary).

```
bool canSkiTowards(int map[MAX_ROW][MAX_COL], int row, int col,
                   Direction dir)
{
```

(b) *(12 points)* Write a function with signature

```
bool isReachable(int map[MAX_ROW][MAX_COL], int i, int j, int
a, int b);
```

that returns true if Gary can ski from `(i, j)` to `(a, b)`, and returns false otherwise. You may assume that the locations `(i, j)` and `(a, b)` are valid cells in the map.

(Hint: you may need to write a recursive function which `isReachable` calls. The function keeps track of the cells you have tried with the use of another 2D array.)

```
bool isReachable(int map[MAX_ROW][MAX_COL], int i, int j,
                 int a, int b)
{
```

(This page is intentionally left blank for continuation of work.)

(c) *(8 points)* Moving from one cell to its adjacent one is defined as one skiing step. Using the above parts, or otherwise, write a function with signature

```
int lowestInKSteps(int map[MAX_ROW][MAX_COL], int i, int j,
int k);
```

that returns the height of the lowest possible location that he can reach from cell `(i, j)` in `k` skiing steps (i.e., greatest descend in `k` steps). You may assume that the location `(i, j)` is a valid one in the map and $k \geq 0$.

```
int lowestinKSteps(int map[MAX_ROW][MAX_COL], int i, int j,
                   int k)
{
```

2. **Trie: A Data Structure for Dictionary** *(35 points total)*

An English dictionary is a collection of unique English words supporting operations such as insertion, search (lookup), outputting, etc. In this question, you will implement a special data structure called "trie" which supports efficient dictionary operations. For simplicity, we will only consider words composed of the 26 lower-case characters from `'a'` to `'z'`.

A *trie* node structure for dictionary is shown in Figure 2, which is is defined as:

```
#include <cstdio>
#include <fstream>
#include <string>
#include <stdio.h>
#include <iostream>

using namespace std;
const int MAX_NCHAR = 26;   // number of lower-case characters
const int MAX_LENGTH = 30;  // maximum number of characters
                            // in a word excluding '\0'

typedef struct Trie_Node
{
  bool exist;              // whether the word from the root is completed
  struct Trie_Node* next[MAX_NCHAR];
} Trie;
```

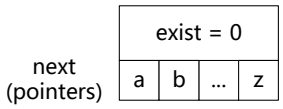


Figure 2: Data structure of a trie node to support efficient dictionary operations.

A trie is a tree structure with a root. Each of the `Trie` nodes has two fields:

- `exist`, which signals the end of a word from the root to the current node.
- `next[MAX_NCHAR]`, whose index corresponds to a character from `'a'` (index 0) to `'z'` (index 25), in increasing order. All its pointers are initialized to `NULL` at node creation.

Trie organizes the words in a tree structure, with each node having a maximum of `MAX_NCHAR` branches. The trie is initialized with a root node where all the pointers are set to `NULL` and `exist = false`. The pointer of a trie node corresponds to the character of a word. Following the pointers, the trie nodes visited from the root to the leave of the tree form the sequential characters of a complete word. On the path, all the leading partial words are also indicated using the `exist` field.

To insert a word, the word is examined character by character from its beginning to the end. For each character encountered, a trie node is created (if not already existed) and pointed by its parent whose pointer index corresponds to the character. As a result, trie nodes are created one by one as the word characters are visited sequentially and inserted into the trie. If the

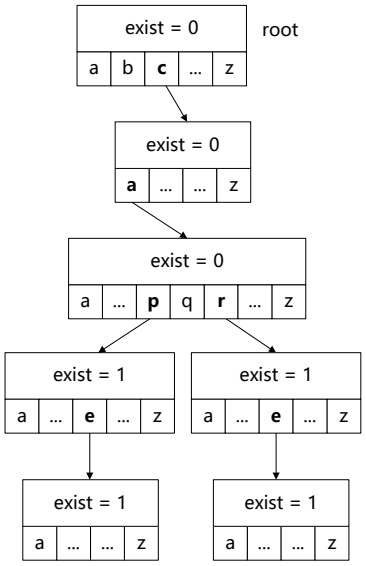


Figure 3: A trie after inserting the words “cap”, “cape”, “car”, and “care” into an empty dictionary.

word has not been completed, we set `exist = false` for the new trie node. Upon the completion of the word, `exist` is then set to be `true`.

We illustrate in Figure 3 the resultant trie with the insertion of four words into an empty dictionary, “cap”, “cape”, “car”, and “care”. The root is initialized with a trie node with `NULL` pointers and `exist = false`. We first consider inserting “cap” into the root node. As the first character is `'c'` and the root pointer of the `'c'` index is `NULL`, a new trie node is created. As the word is not completed yet, `exist = false` in the new node. Then we reach the character `'a'`. A new trie is hence similarly created, pointed by array index 0 (corresponding to character `'a'`) of its parent, with `exisit = false` (as the word has not been completed yet). Then the character `'p'` is encountered in the word. A new trie node is hence created, pointed by the index corresponding to the character `'p'` of the parent. As this completes a word, `exist = true` for the new node. To insert the next word “cape,” we follow the same procedure as above, except that we do not need to create new trie nodes as they have already been created, until “cap” is reached. For the last character in the word, as the pointer corresponding to `'e'` is `NULL`, a new node is created, pointed by the index corresponding to the character `'e'` of its parent. As the word is completed, its `exist` is set to be `true`. Similar steps follow for “car” and “care”, and hence the resultant trie is as shown.

You are given the following function to initialize a new node:

```
Trie* createNode()
{
  Trie *p = new Trie;
  p->exist = false;
  memset(p->next,0,sizeof(p->next));  // setting all pointers to zero
  return p;
}
```

Given the description above, please answer the following questions.

(a) *(3 points)* Write the function with signature (prototype)

```
int index( char ch);
```

which returns the array index of the pointer in a trie node corresponding to the character `ch`.
For example, a call of `index( 'a' )` returns 0, and a call of `index( 'e' )` returns 4.

```
int index( char ch ){
```

(b) *(7 points)* Write a function

```
void Insert(Trie *root, const char* s)
```

which inserts a given word pointed by `s` into a trie with the root node pointed by `root`. Note that `s` points to a null-terminated character array, and the word may or may not have existed in the dictionary.

```
void Insert(Trie *root, const char* s)
{
```

(c) *(7 points)* You are given a text file with a word in each line. The words, not necessarily distinct nor ordered, are to be put into a trie to form a dictionary. Implement the function

```
void Read(char* filename, Trie *root);
```

which opens the file of name `filename`, reads in the words and inserts them into the trie given the root node pointed by `root`. Please simply exit the program upon file error.

```
void Read(char* filename, Trie *root)
{
```

(d) *(6 points)* Implement a function

```
bool Search(Trie *root, const char* s);
```

which searches for the word pointed by `s` in the trie given root pointer `root`. If the word exists, return `true`; otherwise, return `false`.

```
bool Search(Trie* root, const char* s)
{
```

(e) *(12 points)* Implement the function

```
void Print(Trie *root);
```

which prints out all the words in lexicographical order (i.e., according to the order of a dictionary), given a trie whose root node pointed by `root`. As an example, for the dictionary in Figure 3, it should print out

```
cap
cape
car
care
```

Implement your function below. (Hint: You may need to write a a recursive helper function which `Print` calls.)

```
void Print(Trie *root)
{
```

(This page is intentionally left blank for continuation of work.)