CS2040C Semester 1 2018/2019

Data Structures and Algorithms

Tutorial 04 - More on Linear Data Structures and Midterm Test Preparation

For Week 6 (Week Starting 17 September 2018)

Document is last modified on: September 14, 2018

1 Introduction and Objective

This tutorial marks the end of the first $\frac{1}{3}$ of CS2040C: Basic C++, basic analysis of algorithms (worst case time complexity only), various sorting algorithms, and various linear Data Structures (DSes).

In week 6, there will be a 15% midterm test covering the first $\frac{1}{3}$ of CS2040C. A few questions in this tutorial is dedicated to discuss questions that have appeared in past relevant midterm test (and thus will not appear again this semester).

For students that wish to have additional practice questions, you may download Past Year Midterm Papers from `https://www.comp.nus.edu.sg/~stevenha/cs2040c.html`. Do note that CS2040-2017-18-S4 is conducted in Java. CS2040C-2017-18-S1's midterm scope includes basic features of Priority Queue. CS2040C-2017-18-S2's midterm paper has the same scope as this semester's midterm test and you are strongly recommended to try this.

2 Tutorial 04 Questions

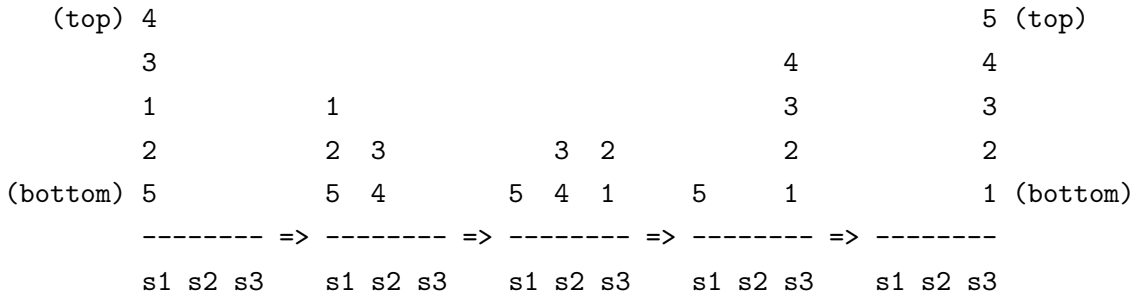
Q1.) Stacking Integers

You are given three stacks: $s1$, $s2$, $s3$ where $s2$ and $s3$ are initially empty and $s1$ contains n ($1 \leq n \leq 100\,000$) distinct positive integers $\in [1..n]$ but **in arbitrary order**.

Your job is simply to determine if it is possible to make $s3$ contains n positive integers in decreasing (sorted) order from top of stack to bottom, but you are constrained as follows:

1. You can only transfer integers from a **lower numbered** stack to a **higher numbered** stack.
2. You can use $s2$ as a temporary data structure (if necessary).
3. You are **NOT** allowed to use *any other* data structure in your solution.
4. You are allowed to use one (or two) more temporary integer variable(s) (if necessary).

For example, if $s1$ contains {5 (bottom), 2, 1, 3, 4 (top)} initially, then the answer is 'possible':



Two sample I/Os are shown below. Convince yourself that the answer for second sample is 'impossible':

Sample Input 1	→	Sample Output 1
5 2 1 3 4		true

Sample Input 2	→	Sample Output 2
1 5 2 4 3		false

Sample Input 3	→	Sample Output 3
5 4 3 2 1		true

Devise an algorithm to complete your job with the lowest worse-case time complexity. Analyse the time complexity of your code.

To solve this problem, a few observations need to be made.

1. You cannot move any integers out from $s3$. Hence, integers that are transferred into $s3$ must be in **increasing order**.
2. It is possible to calculate which is the next integer to be transferred into $s3$. For example, if $s3$ contains (bottom) 1, 2, 3 (top), the next integer to be transferred to $s3$ is 4.
3. There are only 3 possible moves: $s1$ to $s3$, $s2$ to $s3$ and $s1$ to $s2$.

4. For the first two moves, we can only use them if the integer to be transferred matches the computed next integer for $s3$. Otherwise, we can only use the last move to transfer integers from $s1$ to $s2$.

We will now present a solution outline.

1. Set $next\_num = 1$. $next\_num$ stores the next number that is to be transferred to $s3$.
2. If top of stack $s1$ equals to $next\_num$, transfer it to $s3$. Increment $next\_num$ by 1.
3. Else if top of stack $s2$ equals to $next\_num$, transfer it to $s3$. Increment $next\_num$ by 1.
4. Else if $s1$ is not empty, transfer the top integer in $s1$ to $s2$.
5. If $s1$ is empty, terminate the algorithm and return impossible.
6. If the algorithm runs to completion, the output should be true (possible).

Q2). Bracket Matching (Easier?)

In lecture, you have seen the Bracket Matching problem where we are interested in checking whether all brackets are matched correctly. The brackets are matched like this: (with), [with] and { with }. In this tutorial, we are also interested in solving the Bracket Matching problem. However, we will limit the brackets to only (and). Can you devise a *shorter* (in code length) and *simpler* algorithm than the one presented in lecture to solve this *easier* problem?

If we apply the stack based algorithm discussed in lecture, the stack would only contain a single type of bracket. Hence, we can replace the stack with an integer counter variable to denote how many *unopened* open brackets there are. This will simplify the code to a simple increment, decrement and checking of the integer counter variable.

Midterm Test Preparation

Q3.) Generals at War

Two countries A and B are at war. You are the army general of country A and has a strong belief in the effectiveness of a tight battle line formation. You have a troop of N ($1 < N < 1\,000\,000$) soldiers (indexed from 0 to N-1, initially all are alive of course) that you have put in one long and tight battle line: {0, 1, 2, ..., N-1}, where soldier 0 has no one on his left and soldier $N-1$ has no one on his right. The value N is given in the first line of the input. Then you send this troop to fight country B's army.

Over the course of the war, there are G ($0 < G \leq N$) groups of soldiers who are killed, which is described in G casualty reports. The value G is given in the second line of the input. As your troop forms a tight battle line, every time a group of soldiers is killed, their indices happen to be contiguous. That is, a casualty report is simply a pair of two integers (L, R) ($0 \leq L \leq R \leq N$-1) that describes that your soldiers with index from $\{L, L+1, ..., R-1, R\}$ are all killed simultaneously :(. As you have highlighted the importance of maintaining tight battle line, the still-living soldier on the left of soldier with index L (if exists, as L can be index 0) and the still-living soldier on the right of soldier with index R (if exists, as R can be index N-1) must quickly 'close the gap' and march forward again. These G casualty reports are valid reports, i.e. once a soldier has perished, they will never be mentioned again in future casualty reports. The G casualty reports arrive to you chronologically.

Now your job is to design a C++ program that you can use to immediately order soldier $L-1$ (if exist, or report -1 otherwise) and $R+1$ (if exist, or report -1 otherwise) to close the gap to maintain tight battle line. For example, if given $N = 10$ (soldiers) in the first line, $G = 3$ (casualty reports) in the second line, and three pairs of casualty reports: (4, 5), (0, 3), (7, 7) in the next three lines, your program must *quickly* outputs (3, 6), (-1, 6), and (6, 8), respectively in three lines, i.e.:

Input		Output
10		3 6
3		-1 6
4 5	→	6 8
0 3		
7 7		

To help you understand the sample test case, here are the states of your troop for that test case:

```
0123456789 // initial state, you have 10 soldiers, with 3 casualty reports
01236789   // after 1st report (soldiers 4 and 5 perish), 3 and 6 close gap
6789       // after 2nd report (soldiers 0, 1, 2, and 3 perish), 6 is the leftmost
689        // after 3rd report (soldiers 7 perish), 6 and 8 close gap
```

About 70 percent of the actual CS1020E students of S1 AY 2016/17 (158 students) leave this question blank... This is really the question to differentiate the top 75-100 percentile for the coveted A grades (A-/A/A+)... And although posed like a 'Doubly Linked List' style question, the ultimate answer

does NOT use DLL as we lost the ability to quickly, in O(1), identify the index of soldier **L** and **R** in the surviving list if we use DLL. That's why we have to use the 'simulated DLL' using **ln** (left neighbor) and **rn** (right neighbor) arrays.

```
#include <iostream>
using namespace std;

#define MAX_N 1000010 // give a slightly bigger size, just in case

// need to put this big array at heap, otherwise stackoverflow...
// yes, we do NOT need (and cannot use) Doubly Linked List for this question
int i, G, N, L, R, ln[MAX_N], rn[MAX_N];

int main() {
  cin >> N;
  cin >> G;
  // initially, this is the army formation
  ln[0] = -1; // no left neighbor of soldier 0
  for (i = 1; i < N  ; i++) ln[i] = i-1; // this is left neighbor of soldier i
  for (i = 1; i < N-1; i++) rn[i] = i+1; // this is right neighbor of soldier i
  rn[N-1] = -1; // no right neighbor of soldier N-1
  while (G--) { // for each casualty report, or use for (int j = 0; j < G; j++) {
    cin >> L >> R; // guaranteed to be valid and 0 <= L <= R <= N-1
    ln[rn[R]] = ln[L]; // this is my new left
    rn[ln[L]] = rn[R]; // this is my new right
    // for this problem, there won't be a case where all soldiers die
    cout << ln[rn[R]] << " " << rn[ln[L]] << endl; // these two have to close gap
  }
  return 0;
}
```

Q4). Birthday Reminder Problem

You are provided with an array of N people, each being a *Person* object of the following class structure:

```
class Person {
public:
    string name;
    int day;      // DD: 1 to 31
    int month;    // MM: 1 to 12
    int year;     // YYYY: 0 - 3000
};
```

The *name* property is the name of the person, whereas the *day*, *month* and *year* represents the birthday of the person. It is guaranteed that all N people have distinct and valid birthdates.

Your task is to write a *custom comparator* for STL sort to sort the array of N people. They are to be sorted by ascending birth **months** (MM), then tiebreak by ascending birth **dates** (DD) and then by ascending **age**.

```
bool compareBirthdays(Person a, Person b) {
    // returns true if Person a should come before Person b
    // returns false if Person b should come before Person a









}
```

Model answer is as below:

```
bool compareBirthdays(Person a, Person b) {
    if (a.month != b.month)
        return a.month < b.month; // month asc

    if (a.day != b.day)
        return a.day < b.day;  // day asc

    return a.year > b.year;  // year desc
}
```

You can stay back and ask the tutor about any other questions that you may have to prepare you for the more challenging Midterm Test on Wednesday, 3 October 2018.