Homework #6

RELEASE DATE: 05/26/2015

DUE DATE: 06/09/2015, **16:20** in CSIE R102/R104 and on github

As directed below, you need to submit your code to the designated place on the course website.

Any form of cheating, lying or plagiarism will not be tolerated. Students can get zero scores and/or get negative scores and/or fail the class and/or be kicked out of school and/or receive other punishments for those kinds of misconducts.

Discussions on course materials and homework solutions are encouraged. But you should write the final solutions alone and understand them fully. Books, notes, and Internet resources can be consulted, but not copied from.

Since everyone needs to write the final solutions alone, there is absolutely no need to lend your homework solutions and/or source codes to your classmates at any time. In order to maximize the level of fairness in this class, lending and borrowing homework solutions are both regarded as dishonest behaviors and will be punished according to the honesty policy.

Both English and Traditional Chinese are allowed for writing any part of your homework (if the compiler recognizes Traditional Chinese, of course). We do not accept any other languages. As for coding, either C or C++ or a mixture of them is allowed.

This homework set comes with 200 points and 20 bonus points. In general, every homework set of ours would come with a full credit of 200 points.

6.1 Skip List, Binary Search Tree

(1) (20%) Do Exercise C-9.15 of the textbook.

(2) (20%) Do Exercise R-10.5 of the textbook.

(3) (20%) Do Exercise C-10.12 of the textbook.

6.2 Balanced Binary Search Trees

(1) (25%) `libavl` (`http://adtinfo.org/`) is a useful library for binary search trees. For instance, the following short code constructs an AVL tree of 16 integers and print it out.

```
#include <stdio.h>
#include <stdlib.h>
#include "avl.h"
void preorder_integer_avl(const struct avl_node *node){
  if (node == NULL)
    return;
  printf ("%d ", *((int *)node->avl_data));
  if (node->avl_link[0] != NULL || node->avl_link[1] != NULL){
      putchar('(');
      preorder_integer_avl(node->avl_link[0]);
      putchar(',');
      putchar(' ');
      preorder_integer_avl(node->avl_link[1]);
      putchar(')');
  }
}

int int_compare(const void *pa, const void *pb, void *param)
{
  int a = *(const int *)pa;
  int b = *(const int *)pb;

  if (a < b) return -1;
  else if (a > b) return +1;
  else return 0;
}

int main(){
  struct avl_table *tree;
  tree = avl_create(int_compare, NULL, NULL);

  int i;
  for(i=0;i<16;i++){
    int* element = (int*)malloc(sizeof(int));
    *element = i;
    void **p = avl_probe(tree, element);
  }

  preorder_integer_avl(tree->avl_root);
  puts("");
  return 0;
}
```

Note that the manual of `libavl` may be difficult to read; the code above comes from modifying the `avl-test.c` in `libavl`. When the code is compiled with `avl.c`, it correctly outputs an AVL tree (pre-order).

```
7 (3 (1 (0 , 2 ), 5 (4 , 6 )), 11 (9 (8 , 10 ), 13 (12 , 14 (, 15 ))))
```

Write a program `hw6_2` that reads 32 strings (of length at most 128 that can be compared lexicographically) line by line (each line containing one string) from `stdin` and inserts them to the AVL tree (`avl.c`), height-bounded binary search tree (`bst.c`), and Red-Black tree (`rb.c`). Please output the resulting trees (pre-order) to `stdout` with the format similar to the output above in three lines. **Your output order should be AVL tree, height-bounded binary search tree, and Red-Black tree.**

You are encouraged but not required to insert the following 32 keywords in C

```
auto, break, case, char, const, continue, default, do, double, else, enum,
extern, float, for, goto, if, int, long, register, return, short, signed,
sizeof, static, struct, switch, typedef, union, unsigned, void, volatile,
while
```

to see what the trees are like.

6.3 Disjoint Set

Next, we introduce you with a useful data structure called the disjoint set that can be implemented with a forest of trees, and guide you to think about extending "disjoint-set forest" with binary search trees to provide even more sophisticated functionality. The disjoint sets is also called partition, and is introduced in Section 11.4.3 of the textbook. Let's first start with a story.

Suppose that you have n friends, and each friend initially owns a computer game. To make it simple, the friends are numbered from 1 to n, and the game initially-owned by the i-th friend is numbered i. In other words, the set of all games $\{1, 2, \ldots, n\}$ is initially partitioned to n disjoint subsets (owned-by-friend) $\{1\}, \{2\}, \ldots, \{n\}$.

There are two kinds of incidents in our story. In the first incident, as time goes by, some friends find the computer games on hand tedious, and want to play other games. If the current owner u of the i-th computer game finds it tedious, and gets a message on Facebook that the j-th computer game is interesting, two possible results will take place:

- If u also owns the j-th computer game, u will simply switch to play her/his j-th game. No other actions are needed.
- If u doesn't own the j-th computer game, u will visit the person who owns the j-th game, say v, and borrow all v's computer games. Note that after this incident, v will no longer own any computer game. That is, v will go study hard for the DSA homeworks (**and will not be eligible for borrowing games from other friends**).

Another incident arises when you visit your friend who owns the i-th game one day. During the visit, you find that your friend owns many games that are of interest to you. You check your pocket and find s dollars. You then ask your friend about the price of each game she/he owns, with the hope of calculating the maximum number of games you can buy (from the store nearby, not from your friend). To simplify this problem, **you will not actually go buy those from the store nearby nor your friend**.

In this very last problem of the semester, you are asked to simulate the two scenarios with efficient data structures and algorithms. Before you switch to the next page, we encourage everyone to stop and **think about how to solve the problem**. We have sub-problems on the next page to help you conquer the problem.

Input Format

Please read the input from `stdin`. The first line contains two integers, n and m, separated by a space. n denotes the number of your friends, and m denotes the number of incidents. It is guaranteed that $1 \le n \le 10^5$ and $1 \le m \le 2 \times 10^5$. The second line contains n integers, $p_1, p_2, \ldots, p_n$, with p_i and p_{i+1} separated by a space, where p_i denotes the price of the i-th computer game. It is guaranteed that $1 \le p_i \le 10^8$. Finally, each of the following m lines contains three integers that represents an incident. The two kinds of incidents are denoted as follows.

- `1` $[i]$ $[j]$: The current owner of the i-th computer game finds it tedious, and gets a message on Facebook that the j-th computer game is interesting. It is guaranteed that $1 \le i, j \le n$ and $i \ne j$. There is no need to output anything. You only need to take the action that changes the disjoint sets if needed.

- 2 [i] [s]: You visit your friend who owns the i-th game, with s dollars in your pocket. You then ask your friend about the price of each game she/he owns, with the hope of calculating the maximum number of games you can buy. You should output two integers, u and k, separated by a space, to `stdout`. Here u denotes the id of the friend who owns the i-th game, and k indicates the maximum number of games (within u's collection) you can buy with s dollars. It is guaranteed that $1 \le i \le n$ and $1 \le s \le 10^{12}$.

Sample Input

```
4 10
2 1 4 7
2 3 3
1 3 2
2 3 3
1 1 3
2 2 4
1 3 2
2 2 4
2 4 7
1 2 4
2 4 7
```

Sample Output

```
3 0
3 1
1 2
1 2
4 1
1 3
```

(1) (25%) Read Section 11.4.3 of the textbook while paying a special attention to "A Tree-Based Partition Implementation" (hereby called the disjoint-set forest). Then, consider one heuristic that merges the two trees by **depth** instead of size, **without doing union-by-size nor path-compression**. That is, the shorter tree is always merged as a child of the taller tree. Prove that the disjoint-set forest with this heuristic yields a worst-case running time for `find` and `union` within $O(\log n)$. (*Hint: Prove that each `union` operation keeps trees within depth $O(\log n)$ by showing that within the forest, any tree of height h always contains at least 2^h nodes.*)

(2) (25%) Suppose that you only need to output u rather than u and k for this problem. Write down the pseudo-code of an efficient algorithm based on the disjoint forest. *(Hint: In addition to the disjoint forest, you may need an array to keep track of the "owner" of each tree)*

(3) (25%) Suppose that the prices of your friend u's games are stored in a balanced BST as keys, and you have access to the size and the sum of all keys of any subtree of the BST in an $O(1)$ time, write down the pseudo-code of an efficient algorithm for outputting k for the particular u. *(Hint: What does BST stand for? :-))*

(4) (Bonus 20%) So now we know that merging (union) can be done efficiently by inserting the smaller tree into the larger one, locating u can be done by careful bookkeeping, and that a special balanced BST helps locating k. The only difficulty is how to merge two balanced BSTs. Given two balanced BSTs T_1 and T_2, with the size of them being n_1 and n_2 respectively, one can easily come up with a naïve algorithm to obtain the union of T_1 and T_2 by inserting all the elements of T_1 into T_2, and the time complexity is $O(n_1 \log(n_1 + n_2))$. If we take the same heuristic as (1) and always insert the elements of the smaller BST into the bigger one, prove that processing all incidents of the first kind takes $O(n(\log n)^2)$ time.

(5) (40%) Write a program `hw6_3` to solve the problem efficiently. You can use any data structure and algorithm you want as long as your program outputs the answer correctly within the given

time limit (**2 seconds** per each test case). For instance, in addition to the disjoint-set forest, other data structures like treap or copy-on-write segment tree may help. If you want to implement the disjoint-set forest, TA Yen-Chieh provides a modified AVL-tree library based on `libavl`.

Notes about the Modified AVL-tree Library `avl_ntudsa.h/avl_ntudsa.c`

Each node of the tree contains

(1) **int** *avl_data*: the key of AVL-tree.

(2) **int** *avl_cnt*: for counting duplicate keys. For instance, if there is one 1126 in AVL-tree, *avl_cnt* will change from 1 to 2 if you insert another 1126.

(3) **int** *avl_cnode*[2]: the total number of nodes in left subtree (*avl_cnode*[0]) and right subtree (*avl_cnode*[1]).

(4) **long long int** *avl_sum*[2]: the sum of keys in left subtree (*avl_sum*[0]) and right subtree (*avl_sum*[1]).

Besides, some of the functions are removed. The rest is same as `avl.h/avl.c`.

Submission File

Please submit your written part of the homework on all problems together to R102/R104 before the deadline. For the coding part, please follow the same guide of hw2 and submit through github (while tagging your submission as `hw6`). Please **DO NOT PUT BINARY FILES** in your repository.

Your `hw6` directory of the repository should contain the following items:

- `hw6_2.c` or `hw6_2.cpp` or any other non-`libavl` code that you write
- `hw6_3.c` or `hw6_3.cpp` or any other non-`libavl` code that you write
- an optional `Makefile.inc` if needed, which guides how `hw6_2.o` and `hw6_3.o` can be generated
- an optional `README`, anything you want the TAs to read before grading your code

The TAs will use the `Makefile` provided in the repository (and `libavl` code) to test your code. Please make sure that your code can be compiled with the `Makefile` on CSIE R217 linux machines.
Please make sure that your code can be compiled and run with the Makefile on CSIE R217 linux machines. Otherwise your program "fails" its most basic test and can result in ZERO!

Note that this is the last homework of this semester and is therefore your last chance of using the MEDALs, if you still have any!