國立臺南大學資訊工程學系

資工三「演算法」課程

第四次作業

**題目: Dictionary**

|  |  |  |
| --- | --- | --- |
| 班級 | ： | 資工三 |
| 姓名 | ： | 陳博琳 |
| 學號 | ： | S10859007 |

老師：陳宗禧

中華民國 110年12月11日

# 目錄

1. **簡介及問題描述……….……………..…………………………………………1**
   1. **簡介…………….…………………………………………………………………………2**
   2. **問題………………………….……………………………………………………………4**
2. **理論分析….………………………..………………………………………8**
3. **演算法則….…………………………..……………………………………10**
4. **程式設計環境架構.………………………..…………………………………12**
5. **程式.…………………………………………..………………………………14**
6. **執行結果、討論與心得.………………………..……………………………18**

參考文獻………………………………………………………….…………………22

**(一) 簡介及問題描述**

利用 Binary Search Tree (BST), Treap (TH), 2-3 Tree (23T) and Hashing 理論驗證與實驗分析該問題!

1. 簡介

以上述三個方法實作 dictionary，而 dictionary 的單字應該沒有標點符號，且大小寫視為 相同，且相同的單字應該只存在一份。

2. 問題

實作與分析下列演算法，從檔案讀入，完成 dictionary，再讀入另外的檔案查詢單字是否 存在 dictionary 中，並有新增，搜尋，修改的功能，此部分由使用者輸入。

* + 1. Binary Search Tree (BST),
    2. Treap (TH)
    3. 2-3 Tree (23T)
    4. Hashing

**(二) 理論分析**

首先第一個Basic Prime Test…

1. Binary Search Tree (BST)

1. Insert：是先 new 一個 root 的 node，接著使用迴圈，由 x 先探詢往下的路，若插入的 element 小於當前的 node，則往左子樹走(x=x->left)，若大於當前的 node，則往右子樹 走(x=x->right)，當往下到達 leaf node 的時候，x 會指向 NULL，而這個就是迴圈的中 止條件，而 y 會在每次成功進入迴圈時更新 y=x，代表 y 為 x 的 parent。當迴圈結束 後，需要將真正的 node 接上 y，因此將將 y 設給 insert\_node 的 parent (insert\_node- >parent=y)，接著需要知道 insert\_node 是接在 parent y 的右小孩或左小孩，因此再用 if 判斷他們的大小，若 y->element>insert\_node->element，則放在 y 的左小孩，反之放 在 y 的右小孩。

2. Search：搜尋是利用迴圈判斷當前的 node 是否是要找的資料，倘若不是，則比較其 element 大小決定項右子樹或左子樹走，判斷依據與之前相同，若當前的指標指向 NULL(代表已經到 leaf)，且沒有找到目標時結束

3. Delete：刪除較為麻煩，因為要先找到左子樹有最大值的node來替代要被刪除的node， 且會有三種狀況：(1) 被刪除的 node 沒有 child。(2) 被刪除的 node 有 1 個 child(左或 右)。(3) 被刪除的 node 有 2 個 child。(1)較簡單，只要將該 node delete 掉即可，而(2) (3)我們會調整成同一類型，.將所有的 child 集中到其中一邊，接著把他的 parent 設為 要的刪除的 node 其 parent。

1. Treap

1. Insert：插入的方法和二元搜尋樹類似，都是比較後往左或往右走，但需要保持 min heap 的關係，因此做完 insert 後要比較它和 child 的 priority，若 parent 的 priority 大於 child 的 priority，則需做 rotate，如果是 root 大於右 child，則要做 leftrotate，把較小的 priorty 放上來， 反之，若 root 大於左 child，則要做 rightrotate。

2. Search： 與 BST 的搜尋法類似。

3. Delete：(1) 假如 root==NULL ，回傳 null。 (2) 假如 element 大於當前的 data 則呼叫 -5- 遞迴向右走，反之則向左走。 (3) 若刪除的 root 沒有左 child，則把右小孩替代要被刪除的位 置，反之亦同。 (4) 最後需要比較 priority 的值，再考慮是否要 rotate。

1. 2-3 Tree (23T)

1.insert: 針對值進行大小比較，將值安插至對應的Node中，看看Node中的值的數量是否超過2個，若超過，則將中間值往上拉，加入或形成一個parent node，若Node所連結的child node超過3個，進行分割。

2.search:如果node為2檢查search的數是大於兩者，介於兩者之間或小於兩者，若小於兩者到left child，若大於兩者right child，若界在兩者之間middle child，如果node為1與bst方法相同

3.delete: 不在葉子(Non-leaf)下，就拿左子樹最大和右子樹最小去補，在葉子下就直接對葉子做處理，當要刪除的節點「裡頭元素不夠了」會形成空洞，這時我們需要透過以下的方式，從下而上把洞給補起來，如果左鄰右舍還有剩，就「向左鄰右舍」借 這就是Rotation，如果左鄰右舍沒得借了，那就把鄰居「併」起來，聯合起來「向樓上」借這就是Combination 如果樓上借完之後不夠了，會形成一個洞，這個流程，一路補起來

1. Hashing

insert: 先利用Hash Function取得Table的index。接著，只要在每一個slot的list之front加入item，即可保證在O(1)O(1)的時間複雜度完成。

Search:先利用Hash Function取得Table的index，再利用Linked list的traversal尋找item。

Delete:先利用Hash Function取得Table的index，再利用Linked list的traversal尋找欲刪除的item。

**(三) 演算法則**

1. Binary Search Tree (BST)

|  |
| --- |
| class BST(object): |
|  | def \_\_init\_\_(self, root:Node): |
|  | self.root = root |
|  |  |
|  | def delete\_node(self, node:Node): |
|  | node = self.get\_node\_by\_key(node.key) |
|  | parent\_node = node.parent\_node |
|  | # not have sub-tree |
|  | if not node.left\_node and not node.right\_node: |
|  | if parent\_node: |
|  | if parent\_node.left\_node == node: |
|  | parent\_node.left\_node = None |
|  | else: |
|  | parent\_node.right\_node = None |
|  | else: |
|  | # the node is root, update the root |
|  | self.root = None |
|  | return |
|  |  |
|  | # only have right sub-tree |
|  | if not node.left\_node and node.right\_node: |
|  | if parent\_node: |
|  | if parent\_node.left\_node == node: |
|  | parent\_node.left\_node = node.right\_node |
|  | else: |
|  | parent\_node.right\_node = node.right\_node |
|  | else: |
|  | # the node is root, update the root |
|  | self.root = node.right\_node |
|  | return |
|  |  |
|  | # have left sub-tree |
|  | if node.left\_node: |
|  | left\_bst = BST(node.left\_node) |
|  | largest\_node\_in\_left = left\_bst.get\_largest\_node() |
|  |  |
|  | # check if the root of left sub-tree is the largest node in left sub-tree |
|  | if largest\_node\_in\_left == node.left\_node: |
|  | largest\_node\_in\_left.right\_node = node.right\_node |
|  | largest\_node\_in\_left.parent\_node = node.parent\_node |
|  | if not parent\_node: |
|  | # the node is root, update the root |
|  | self.root = largest\_node\_in\_left |
|  | return |
|  |  |
|  | # check the largest node in left sub-tree have its left sub-tree or not |
|  | if largest\_node\_in\_left.left\_node: |
|  | largest\_node\_in\_left.parent\_node.right\_node = largest\_node\_in\_left.left\_node |
|  | else: |
|  | largest\_node\_in\_left.parent\_node.right\_node = None |
|  |  |
|  | # check the deleted node is root or not |
|  | if parent\_node: |
|  | if parent\_node.left\_node == node: |
|  | parent\_node.left\_node = largest\_node\_in\_left |
|  | else: |
|  | parent\_node.right\_node = largest\_node\_in\_left |
|  | else: |
|  | # the node is root, update the root |
|  | self.root = largest\_node\_in\_left |
|  |  |
|  | largest\_node\_in\_left.right\_node = node.right\_node |
|  | largest\_node\_in\_left.left\_node = node.left\_node |
|  |  |
|  | return |

演算法時間複雜度(time complexity)

Insert：O(n), 依序一個一個插入

Search：O(logn), 每次都是往左子樹或往右子樹

Delete：O(logn), 每次都是往左子樹或往右子樹演算法

空間複雜度(space complexity)

Insert：O(*n*), insert前在陣列中，insert後會佔n大小的空間

Search: O(1), 不需要額外的空間儲存

Delete：O(1), 不需要額外的空間儲存

2. Treap

**typedef struct item \* pitem;**

**struct item {**

**int prior, value, cnt;**

**bool rev;**

**pitem l, r;**

**};**

**int cnt (pitem it) {**

**return it ? it->cnt : 0;**

**}**

**void upd\_cnt (pitem it) {**

**if (it)**

**it->cnt = cnt(it->l) + cnt(it->r) + 1;**

**}**

**void push (pitem it) {**

**if (it && it->rev) {**

**it->rev = false;**

**swap (it->l, it->r);**

**if (it->l) it->l->rev ^= true;**

**if (it->r) it->r->rev ^= true;**

**}**

**}**

**void merge (pitem & t, pitem l, pitem r) {**

**push (l);**

**push (r);**

**if (!l || !r)**

**t = l ? l : r;**

**else if (l->prior > r->prior)**

**merge (l->r, l->r, r), t = l;**

**else**

**merge (r->l, l, r->l), t = r;**

**upd\_cnt (t);**

**}**

**void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {**

**if (!t)**

**return void( l = r = 0 );**

**push (t);**

**int cur\_key = add + cnt(t->l);**

**if (key <= cur\_key)**

**split (t->l, l, t->l, key, add), r = t;**

**else**

**split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;**

**upd\_cnt (t);**

**}**

**void reverse (pitem t, int l, int r) {**

**pitem t1, t2, t3;**

**split (t, t1, t2, l);**

**split (t2, t2, t3, r-l+1);**

**t2->rev ^= true;**

**merge (t, t1, t2);**

**merge (t, t, t3);**

**}**

**void output (pitem t) {**

**if (!t) return;**

**push (t);**

**output (t->l);**

**printf ("%d ", t->value);**

**output (t->r);**

演算法時間複雜度(time complexity)

Insert：O(logn), 依序一個一個插入

Search：O(logn), 每次都是往左子樹或往右子樹

Delete：O(logn), 每次都是往左子樹或往右子樹演算法

空間複雜度(space complexity)

Insert：O(*n*), insert前在陣列中，insert後會佔n大小的空間

Search: O(1), 不需要額外的空間儲存

Delete：O(1), 不需要額外的空間儲存

3. 2-3 Tree (23T)

class Node23

{

static Node23\* const Unused;

int data[2];

Node23\* child[3];

Node23\* parent;

public:

// New nodes are always 2-nodes

Node23(int d, Node23\* p=nullptr, Node23\* a=nullptr, Node23\* b=nullptr)

: data{d, 0}, child{a, b}, parent{p} { child[2] = Unused; }

// For manual tree-building we need to make 3-nodes

Node23(int d0, int d1, Node23\* p=nullptr,

Node23\* a=nullptr, Node23\* b=nullptr, Node23\* c=nullptr)

: data{d0, d1}, child{a, b, c}, parent{p} { }

~Node23() {

delete child[0]; // delete is okay with nullptr

delete child[1];

if (is3node()) delete child[2]; // but not with our special value

}

// "active" pointer is neither nullptr nor Unused.

static bool active(Node23\* node) { return node && node != Unused; }

bool is3node() const { return child[2] != Unused; }

bool is\_leaf() const { return child[0] == nullptr; }

bool expand(int d) {

if (is3node())

return false; // will become a 4-node; let caller handle it

if (d < data[0]) {

data[1] = data[0];

data[0] = d;

}

else

data[1] = d;

child[2] = nullptr; // mark as 3-node (overwriting Unused value)

return true;

}

void print(bool top = true) const {

if (child[0]) child[0]->print(false);

std::cout << data[0] << ' ';

if (child[1]) child[1]->print(false);

if (is3node()) {

std::cout << data[1] << ' ';

if (active(child[2])) child[2]->print(false);

}

if (top) std::cout << '\n';

}

void print\_structure(int depth = 0) const {

if (is3node()) {

if (active(child[2])) child[2]->print\_structure(depth + 1);

std::cout << std::setw(depth \* 4) << "" << data[1] << '\n';

}

if (child[1]) child[1]->print\_structure(depth + 1);

std::cout << std::setw(depth \* 4) << "" << data[0] << '\n';

if (child[0]) child[0]->print\_structure(depth + 1);

}

};

Node23\* const Node23::Unused = reinterpret\_cast<Node23\*>(1);

int main() {

// 10

// 3 , 5 15

// 1 4 6 12 18

// Manually create a tree (with null parent pointers for now)

auto a = new Node23(10,

nullptr,

new Node23(3, 5,

nullptr,

new Node23(1),

new Node23(4),

new Node23(6)),

new Node23(15,

nullptr,

new Node23(12),

new Node23(18)));

a->print();

// The structure printout shows the tree sideways.

// Tilt your head to the left to view it. :)

a->print\_structure();

delete a;

}

演算法時間複雜度(time complexity)

Insert：O(logn)

Search：O(logn), 每次都是往左子樹或往右子樹

Delete：O(logn), 每次都是往左子樹或往右子樹演算法

空間複雜度(space complexity)

Insert：O(*n*), insert前在陣列中，insert後會佔n大小的空間

Search: O(1), 不需要額外的空間儲存

Delete：O(1), 不需要額外的空間儲存

1. Hashing

// CPP program to implement hashing with chaining

#include<bits/stdc++.h>

using namespace std;

class Hash

{

    int BUCKET;    // No. of buckets

    // Pointer to an array containing buckets

    list<int> \*table;

public:

    Hash(int V);  // Constructor

    // inserts a key into hash table

    void insertItem(int x);

    // deletes a key from hash table

    void deleteItem(int key);

    // hash function to map values to key

    int hashFunction(int x) {

        return (x % BUCKET);

    }

    void displayHash();

};

Hash::Hash(int b)

{

    this->BUCKET = b;

    table = new list<int>[BUCKET];

}

void Hash::insertItem(int key)

{

    int index = hashFunction(key);

    table[index].push\_back(key);

}

void Hash::deleteItem(int key)

{

  // get the hash index of key

  int index = hashFunction(key);

  // find the key in (index)th list

  list <int> :: iterator i;

  for (i = table[index].begin();

           i != table[index].end(); i++) {

    if (\*i == key)

      break;

  }

  // if key is found in hash table, remove it

  if (i != table[index].end())

    table[index].erase(i);

}

// function to display hash table

void Hash::displayHash() {

  for (int i = 0; i < BUCKET; i++) {

    cout << i;

    for (auto x : table[i])

      cout << " --> " << x;

    cout << endl;

  }

}

// Driver program

int main()

{

  // array that contains keys to be mapped

  int a[] = {15, 11, 27, 8, 12};

  int n = sizeof(a)/sizeof(a[0]);

  // insert the keys into the hash table

  Hash h(7);   // 7 is count of buckets in

               // hash table

  for (int i = 0; i < n; i++)

    h.insertItem(a[i]);

  // delete 12 from hash table

  h.deleteItem(12);

  // display the Hash table

  h.displayHash();

  return 0;

}

演算法時間複雜度(time complexity)

Worst case 所有的東西放進同一個slot

Insert：O(n)

Search：O(n)

Delete：O(n)

空間複雜度(space complexity)

Insert：O(*n*), insert前在陣列中，insert後會佔n大小的空間

Search: O(1), 不需要額外的空間儲存

Delete：O(1), 不需要額外的空間儲存

**(四) 程式設計環境架構**

程式設計語言、工具、環境與電腦硬體等規格說明…

1. 程式語言

C in MS Windows

2. 程式開發工具

Visual C++ 2019

3. 電腦硬體



**(五) 程式 (含source code, input code, and output code)**

程式含source code, input code, and output code等…

1. 主程式

C or C++ program

請見cpp檔

2. Input Code Format

Three of examples for input use are in below….

Dictionary

(1)testfile1 (445 word)

(2)testfile2 (268 word)

Find file

1. 完整句子存在dictionary中
2. 重複單字
3. 並非完整句子存在dictionary中

3. Output Code Format

Three of examples for output use are in below….

Dictionary testfile1

(1) 

(2) 

(3) 

(4) 

(5) (function exp)

(6) (function add all)

Dictionary testfile2

(1) 

(2) 

(3) 

(4) 

(5) (function exp)

(6) (function add all)

用dictionary testfile1

find file11

1. 
2. 
3. 
4. 
5. (function exp)
6. (function add all)

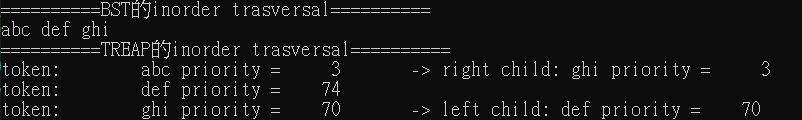
Find file12

1. 
2. 
3. 
4. 
5. (function exp)
6. (function add all)

Find file13

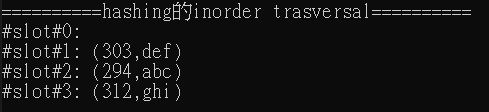
1. 
2. 
3. 
4. 
5. (function exp)
6. (function add all)

插入前

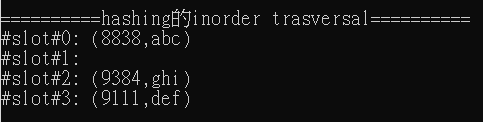
(1)

(2) 

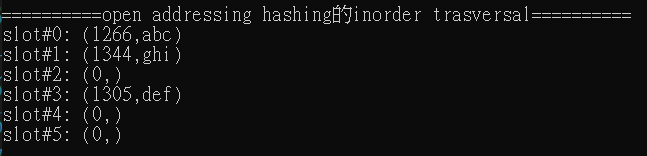
(3) Function1



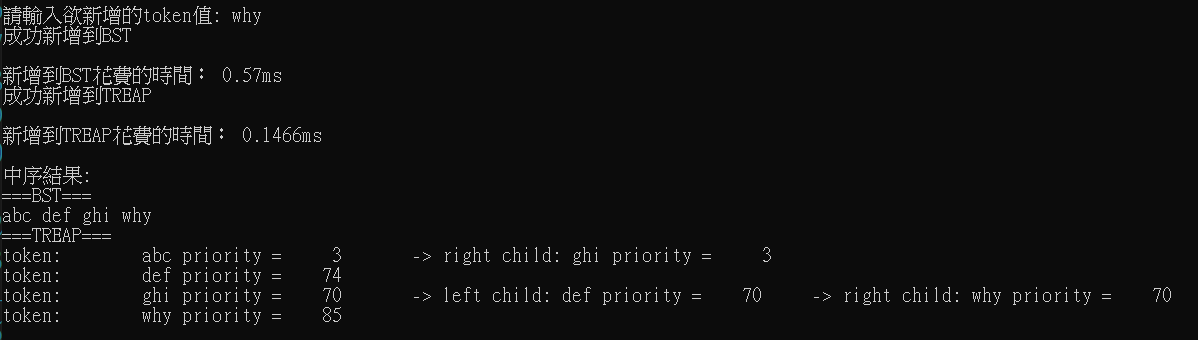
1. function2

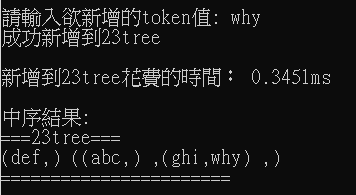


(5)



插入後

(1)

(2) 

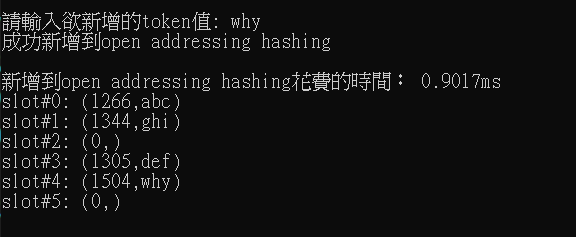
(3)exp



(4)add all

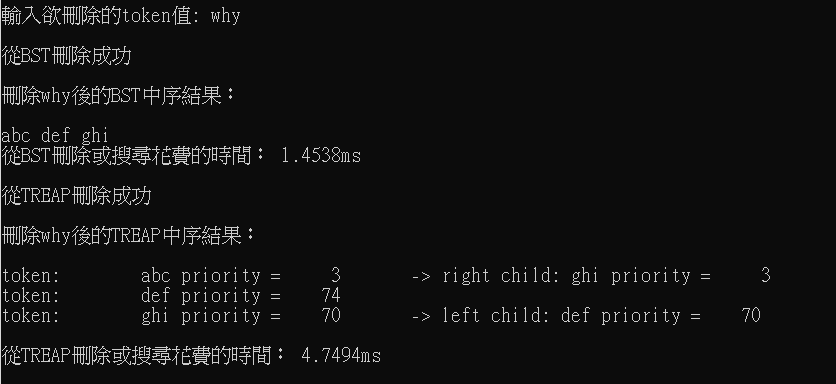


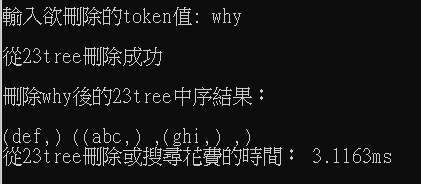
(5)



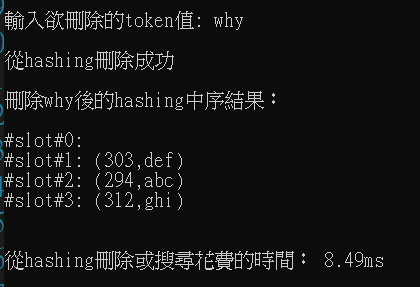
刪除後

(1)

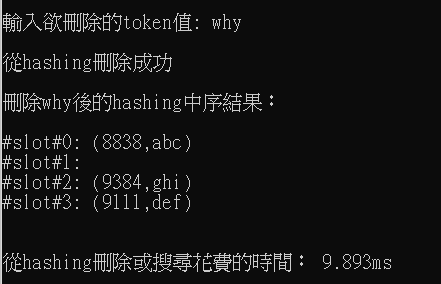


(2) 

(3)exp



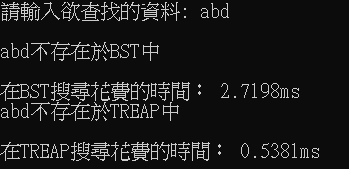
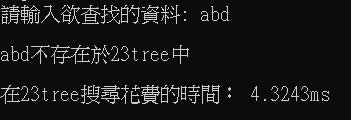
(4)add all

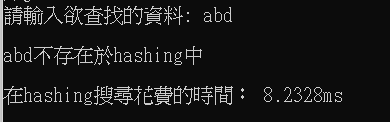


(5)

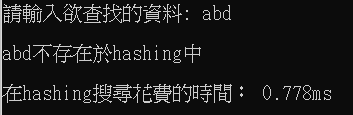


搜尋

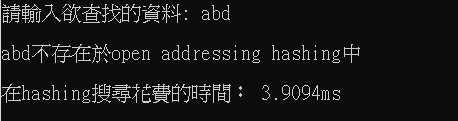
1. 
2. 
3. exp



1. add all



(5)



**(五) 執行結果、討論與心得**

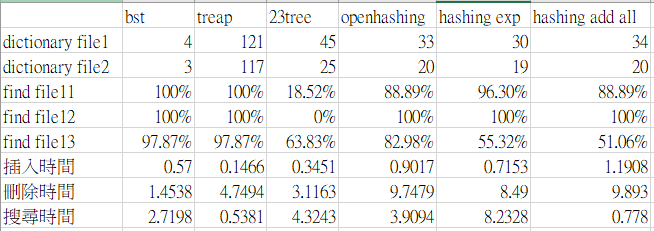
執行結果與討論 (執行時間、problem *n*的大小等問題討論)等…

1. 執行結果

請參閱以上output

2. 討論

執行時間、問題大小等問題討論! 利用 MS Excel 畫出問題大小與執行時間的關係!



這裡列出所有比較表格

Collison 發生時採用openhashing 處理先將起始大小設為2當發生collison時增加存放空間的大小

3. 心得

這邊的openhashing是進行exp function，程式碼有多add all的function因為時間複雜度並無太大差異所以就沒有再繼續比較

**參考文獻**

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms," Third Edition, The MIT Press, 2009.
2. R.C.T. Lee, S.S. Tseng, R.C. Chang, and Y.T.Tsai, "Introduction to the Design and Analysis of Algorithms," McGraw-Hill, 2005.
3. Anany V. Levitin, "Introduction to the Design and Analysis of Algorithms," 3rd Edition, Addison Wesley, 2012.
4. Richard Neapolitan and Kumarss Naimipour, "Foundations of Algorithms," Fourth Edition, Jones and Bartlett Publishers, 2010.
5. <https://www.hackerearth.com/practice/math/number-theory/primality-tests/tutorial/>
6. …