基礎資料結構與 STL 1

資料結構是在程式裡儲存資料以方便處理的方法，不同的資料結構有各自的優點與缺點，選擇適當的資料結構可以使得資料的查找、添增或刪除更有效率。

C++ 的**標準模板庫 (Standard template library, STL)** 提供許多十分強大的內建資料結構可以使用，熟悉這些工具既可以減少許多賽場上不必要的精力花費，又可以降低出錯的可能性。

這裏會先介紹最基本的資料結構，包含 ‘stack‘、‘queue‘、陣列的延伸 ‘vector‘、Linked List 和一些比較進階的：‘priority_queue‘、‘map‘、‘set‘等資料結構。前四個基本資料結構建議自己要實作看看，之後大部分都是用 STL 的東西，知道概念之後就可以直接用。

這裏重要的是要熟悉用法和變化題，其中前四個資料結構一定要自己寫出來一遍，知道內部運作。

1.1 標頭檔與 ‘std‘ 名稱空間

STL 的東西都是寫在 ‘std‘名稱空間之下的，因此在使用這些工具之前，都必須加入這兩行程式碼：

```
1 #include <bits/stdc++.h>
2 using namespace std;
```

其中的 ‘<bits/stdc++.h>‘是一個對於競程選手友善的標頭檔，它涵蓋了幾乎所有演算法競賽所需要的工具，以下所有工具都包含在這個標頭檔下，當然你也可以一個一個 ‘include‘。

1.2 動態陣列 (Dynamic arrays)

最基礎的資料結構大概就是陣列了，相信大家也都會使用。

STL 的方便之處是它提供了一個動態型的陣列 ‘vector‘，它可以像一般的陣列一樣操作，並且可以隨時更動陣列大小以配合內部元素數量的增減。

‘vector‘

我們可以用以下方式來宣告 ‘vector‘

```
vector<int> vecA;
// 宣告一個元素型別為int的空vector
vector<int> vecB(10);
// 宣告一個長度為10的vector，初始值為0
vector<int> vecC(10,2);
// 宣告一個長度為10的vector，初始值為2
vector<int> vecD = {1, 2, 3, 4, 5}; // 賦予初始值
vector<int> vecE {1, 2, 3, 4, 5}; // 也可以這樣賦予初值
```

‘< >‘中的資料型別可以替換，代表陣列中元素的型別。

增添與查找

‘vector‘的 ‘push_back‘函數可以將元素新增到 ‘vector‘的最尾端；‘pop_back‘函數則刪除最尾端的元素。

```
vector<int> vec;
vec.push_back(1); // [1]
vec.push_back(4); // [1,4]
vec.pop_back(); // [1];
```

‘vector‘與一般陣列一樣，可以用 ‘operator[]‘進行查找元素。另外也可以透過 STL 內建的 ‘at‘函數與 ‘back‘函數來進行元素的查找。

```
cout << v[0] << endl;
// 印出vector的第0個元素
cout << v.at(0) << endl;
// 也可以這樣
cout << v.back() << endl;
// 印出vector尾端的元素值
```

尋訪 ‘vector‘裡的每個元素可以用更簡短的寫法：

```
for(auto &x:vec){
    cout << x << endl;
}
```

‘size‘函數回傳有多少元素在這個 ‘vector‘裡面

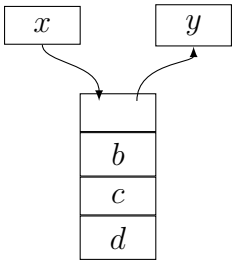
```
for(int i=0;i<vec.size();i++)
    cout << vec[i] << endl;
```

1.3 堆疊 (Stack)

堆疊，顧名思義就是一堆資料疊成一堆，每次都只能從最上面存取資料，否則整個堆疊就會倒塌，這正是所謂先進後出 (first in last out, FILO) 原則。

實作原理

假設我想要疊盤子，將盤子疊在一起，**每次拿到的就是最上面的盤子，放盤子也是最上面**，那就輪到 ‘stack‘出來了！正好就是 ‘stack‘的用武之地。



以上就是 ‘stack‘的示意圖：只能從最上面拿和放（此處原本最上面是 y，可以拿走；或者可以從最上面放 x 進去。）那要怎麼實作呢？只需要用一個陣列即可，那還需要什麼資訊呢？需要維護這個 ‘stack‘的最上方的位置，隨著東西的 ‘push‘、‘pop‘而移動。

習題 1.3.1: stack 練習

請實作一個 ‘stack‘，需要支援兩種操作：

1. ‘PUSH ‘x，代表 ‘stack‘內要放入一個值為 x 的物體
2. ‘POP‘，代表 ‘stack‘內要將最上層的東西拿出來，並輸出其值。如果 ‘stack‘是空的，請輸出 ‘stack is empty¡

```cpp
int Stack[maxn], tot = 0;
void PUSH(int x) {
    Stack[tot++] = x;
}
void POP(){
    if(tot == 0)
        cout << "stack is empty!\n";
    else
        cout << Stack[--tot] << endl;
}
```

‘stack‘

STL 也有內建的 ‘stack‘容器適配器，有新增、刪除、查詢最頂部元素的功能。基本上 STL 中 ‘stack‘能做到的事情 ‘vector‘都能做到，事實上 ‘stack‘就是用 ‘vector‘實作完成的。

容器適配器（container adapter）是利用 STL 原有的容器 (如 ‘vector‘) 去實作的介面。利用容器適配器可以使得程式碼更加簡潔，增加程式可讀性。STL 的 ‘stack‘、‘queue‘與 ‘priority_queue‘都是容器適配器的一種。

以下是宣告一個 ‘stack‘的方法。

```cpp
stack<int> sta; // 宣告一個儲存int型別的stack
```

‘stack‘是放在 ‘<stack>‘標頭檔內，使用前記得引入。

堆疊操作

‘push()‘ 可以將資料放入堆疊的頂部；
‘pop()‘ 將頂端元素刪除；
‘top()‘ 查詢堆疊頂端元素；
‘size()‘ 查詢目前還位於堆疊中的資料數；
‘empty()‘ 回傳堆疊是否為空。

```cpp
sta.push(3); // [3]
sta.push(2); // [3,2]
sta.push(5); // [3,2,5]
```

```
sta.pop(); // [3,2]
cout << sta.size() << endl; // 2
cout << sta.empty() << endl; // 0 (表示非空)
cout << sta.top() << endl; // 2
```

常犯錯誤：在使用 `top()`函數時記得要加括號，不要 CE 在哪裡都不知道。

習題

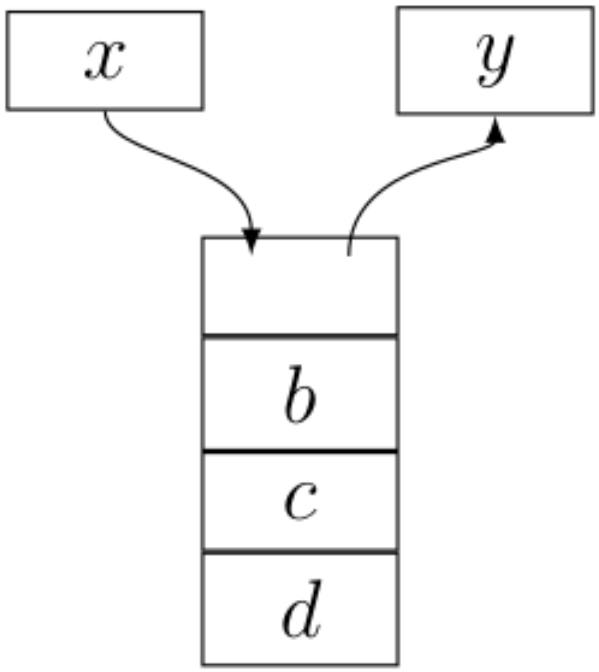
來做題目吧！

習題 1.3.2: Parentheses Balance (ZJ b304, UVa 673)

給你一個由四個字元（`(`、`)`、`{`、`}`）所組成的字串 S，請問 S 是否為一個合法的字串（我們定義字串為合法，若且唯若每一個括弧都可以被匹配到，`()()`合法，而 `{ ( } )`則不合法。），且 $|S| \leq 10^5$。

習題 1.3.3: Rails (TIOJ 1012, UVa 514)

現在有一個火車站，和編號為 $1, 2, 3, \cdots n$ 的火車，身為火車長的 03t 想要將之重新排列。火車的排列如下：



(此處，想要的重新排列是 $5, 4, 3, 2, 1$，而火車可以做的事就是進站，如果 a 號火車進站了，之後又有一個 b 火車進站，則 b 火車得先出站，a 火車才能出站（太窄了 !)，請問 03t 所想要的重新排列是否可能？($n \leq 10^5$)

習題 1.3.4: Cows (TIOJ 1176)

這一題比較難！

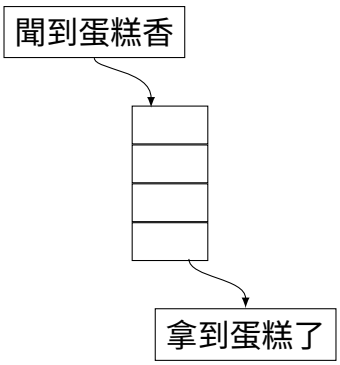
有 N 隻牛排成一條線，每一隻牛都有高度 a_i，這些牛很喜歡看來看去，而第 i 隻牛看得到第 j 隻牛的條件就是中間沒有牛擋住了視線，也就是不存在 $i < k < j$ 使得 $a_k > a_i$。給定 N 和 $a_1, a_2, a_3, \cdots, a_N$，請問每一隻牛可以看得到幾隻牛？($N \leq 10^6$)

1.4 佇列 (Queue)

佇列，顧名思義就是一堆人在排隊，先進去排隊的人先享有辛苦排隊的成果，這是所謂先進先出 (first in first out, FIFO) 原則。

實作原理

想像一家熱門的蛋糕店，有很多人在排隊，而每一個人都不會插隊，那要怎麼模擬呢？當最前面的人拿到了夢寐以求的蛋糕的時候，就會從**最前面離開隊伍**，而當有一個人聞香而來，就會從**最後面加入隊伍**。這就是 ‘queue‘的精神！以下為蛋糕店排隊的模擬圖：



和 ‘stack‘不同的是：一個是拿、放同側，而這個是異側，是 FIFO（First In Last Out）資料結構。實作方法比較難一點，得在陣列維持兩個變數（也就是頭和尾），插入的時候動尾，而離開的時候動頭。請注意：這裏的 Queue 實作方法是循環的，不再和操作數有關，而和元素數量有關。

習題 1.4.1: queue 練習

請實作出一個 ‘queue‘吧！要支援兩個操作：

1. ‘PUSH ‘x，代表 ‘queue‘的最後方要插入一個值為 x 的物體。
2. ‘POP‘，代表 ‘queue‘內要將最前面的東西拿出來，並輸出其值。如果 ‘queue‘是空的，請輸出”‘queue is empty¡”

```
int Queue[maxn], Front = 0, Back = 0;
void PUSH(int x) {
    Queue[Back] = x;
    if(++Back >= maxn) Back -= maxn;
}
void POP(){
    if(Front == Back)
        cout << "queue is empty!\n";
    else{
        cout << Queue[Front] << endl;
        if(++Front >= maxn) Front -= maxn;
```

```
12      }
13  }
```

假設這個佇列最多只能存 N 個元素，當下一個元素要被放進第 $N+1$ 格時，因為有 pop 操作的存在，所以這個佇列不一定是滿的狀態。通常為了有效節省實作佇列時所需的空間，我們都會用循環的方式來實作 queue，意即假設原本存在第 1 格的元素已經被 pop 掉，我們就可以將原本要儲存在 $N+1$ 格的元素儲存在第 1 格，重複利用能用的空間。

‘queue‘

STL 也為佇列設計了一個容器適配器，預設的容器跟 ‘stack‘一樣是 ‘vector‘；它支援從後面插入資料，但是資料是從前面取出。

以下是 ‘queue‘的宣告方法。

```
queue<int> que; // 宣告一個儲存int型別的queue
```

‘queue‘是放在 ‘<queue>‘標頭檔內，使用前必須引入。

佇列操作

‘push()‘ 可以將資料排入佇列的尾端；
‘pop()‘ 將前端元素刪除；
‘front()‘ 查詢佇列前端元素；
‘size()‘ 查詢目前還位於佇列內的資料數；
‘empty()‘ 回傳佇列是否為空。

```
que.push(3); // [3]
que.push(2); // [3,2]
que.push(5); // [3,2,5]
que.pop(); // [2,5]
cout << que.front() << endl; // 2
```

新手常犯錯誤：是 ‘front()‘，不是 ‘top()‘，不要搞錯了。

習題

做習題很重要！

習題 1.4.2: Team queue (UVa 540)

一堆人要排隊去買東西，但是還有一個限制：每一個人可能會是 M 個組中的成員之一（每一個人最多只會加入一個組），所以排隊會有一個新的規則：如果一個人要去排隊了，但是隊伍中有同組的人的話，那這個（有點缺德）的人就會插隊插到自己組的末端。會跟你說隊伍有誰，和一堆進入隊伍／最前面的人離開的指令，請模擬這個情況。(指令數 $\leq 2 \times 10^5$，$M \leq 10^3$)

習題 1.4.3: Throwing cards away I（UVa 10935）

我現在有 n 張牌寫上了 1 到 n 的正整數，一開始由上而下是 $1, 2, 3, \cdots, n$，而我每次會進行這個操作：只要有兩張牌以上，就把第一張牌拿掉，並且將最後一張牌放到最下面。請輸出牌被拿掉的順序？($n \leq 10^5$)

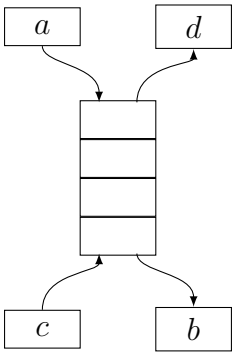
1.5 雙端佇列 (Deque)

在介紹這個資料結構之前，要先決定這個資料結構的唸法。

定理 1.5.1: Deque 的正確唸法

根據 Knuth 的 *The Art of Computer Programming*，Volume 1, Section 2.2.1 "Stacks, Queues, and Deques"：「A deque ("double-ended queue") is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list. A deque is therefore more general than a stack or a queue; **it has some properties in common with a deck of cards, and it is pronounced the same way.**」正確唸法為 [dɛk]，如果念為 [di:kju:] 可能會誤認為 *dequeue*，意思是從 'queue'中移除，造成誤會。

這個東西感覺像是 'queue'和 'stack'的進化版，**可以從頭尾拿，也可以從頭尾放東西**，見以下示意圖：



所以會支援四個操作：兩邊各兩個，放和拿：‘push_back‘、‘push_front‘、‘pop_back‘、‘pop_front‘。那為什麼要用 ‘stack‘和 ‘queue‘，而不直接用 ‘deque‘呢？因為不需要那麼多的功能，還會影響程式的可讀性：如果每次都用 ‘deque‘，但是 ‘stack‘比較好懂，不要每次都註解這個是 ‘stack‘，‘queue‘之類的。那要怎麼實作呢？需要維護頭尾在哪裡，所以從這個方面實作。

習題 1.5.1: deque 練習

來實作 ‘deque‘吧！這一次，需要支援四個操作：

1. ‘POP_FRONT‘ 和 ‘POP_BACK‘，分別代表要從前面和後面拿出東西出來並輸出拿出來的東西的值，如果沒有東西可以拿的話，那就輸出 ‘”deque is empty!”‘
2. ‘PUSH_FRONT‘ x 和 ‘PUSH_BACK‘ x，分別代表要從前面和後面插入一個值為 x 的物體。

   並且保證總共不會超過 N 個操作。

```
int DeQueue[maxn], Front = 0, Back = 0;
void PUSH_FRONT(int x) {
    if(--Front < 0) Front += maxn;
    DeQueue[Front] = x;
}
void PUSH_BACK(int x) {
    if(++Back >= maxn) Back -= maxn;
    DeQueue[Back] = x;
}
void POP_FRONT(){
    if(Front == Back)
        cout << "deque is empty!\n";
    else{
        if(++Front >= maxn) Front -= maxn;
        cout << DeQueue[Front] << endl;
    }
}
void POP_BACK(){
    if(Front == Back)
        cout << "deque is empty!\n";
    else{
        if(--Back < 0) Back += maxn;
```

```
23          cout << DeQueue[Back] << endl;
24      }
25  }
```

‘deque‘

STL 裡面的 ‘deque‘也有雙端佇列的功能，可以在首端或尾端存取資料，宣告時要引入 ‘<deque>‘標頭檔。以下是 ‘deque‘的宣告。

```
1  deque<int> dq; // 宣告一個空的deque
```

雙端佇列操作

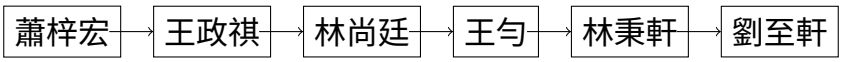
‘push_back()‘ 可以將資料排入雙端佇列的尾端；
‘pop_back()‘ 將雙端佇列尾端元素刪除；
‘push_front()‘ 可以將資料排入雙端佇列的首端；
‘pop_front()‘ 將雙端佇列首端元素刪除；
‘front()‘ 查詢雙端佇列首端元素；
‘back()‘ 查詢雙端佇列尾端元素；
‘size()‘ 查詢目前還位於佇列內的資料數；
‘empty()‘ 回傳佇列是否為空。

```
1  dq.push_back(3); // [3]
2  dq.push_back(1); // [3,1]
3  dq.push_front(2); // [2,3,1]
4  dq.pop_back(); // [2,3]
5  dq.pop_front(); // [3]
6  dq.push_front(1); // [1,3]
7  cout << que.front() << endl; // 1
8  cout << que.back() << endl; // 3
```

能用 ‘stack‘、‘queue‘、一般陣列、或 ‘vector‘解決的問題就盡量避免用 ‘deque‘。

1.6 鏈結串列 (Linked List)

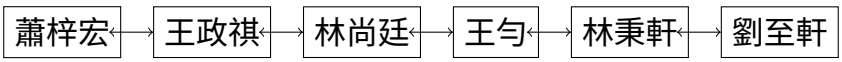
現在假設我們在玩一個遊戲：每一個人都要指一個人，那要怎麼紀錄這種關係呢？將每一個人想成一個東西，裡面包著自己的值和指向的人！這就是 Linked List 的精神。



這裏，蕭梓宏指向王政祺，王政祺指向林尚廷..... 直到劉至軒停止。

雙向串列（Doubly Linked List）

有時候會有人想要知道是誰指向他，就會再維護一個指標指向指向我的那個人，圖形大概長這樣：



作法就是呢：當 A 連到 B 的時候，將 A 指向的人設為 B，而 B 的被指的人設為 A。

查詢和修改和複雜度

如果想要查詢「一個東東指到誰？」的話，顯然這個可以在 $O(1)$ 內做完。那如果想要做的是查詢「我這個人一直指，指到第 k 個人會是誰呢？」那就得花 $O(k)$ 的時間，所以**串列不適合隨機存取**，也就是在只給串列頭的狀況下，查詢第 n 個的值，會變得很慢。

若要隨機存取，通常會搭配陣列來存（也就是呢，維持一個陣列來存）、陣列第 i 值存第 i 個人的值、指向下一個人的陣列值、和被指的人的陣列值。

實作時間

先來個經典問題。

習題 1.6.1: 實作噩夢之 linked list 篇

請實作一個 'linked list'，需要支援四種操作：

1. 'INSERT 'x，在 id $= x$ 的節點之後插入一個新節點 (給予流水號 id)
2. 'DELETE 'x，刪除 id $= x$ 的節點
3. 'PREV/NEXT 'x，查詢 id $= x$ 的節點的前一項/下一項的 id
4. 'TRAVEL'，依序輸出 linked list 內部的內容

對於前三種操作，若找不到 id = x 的節點，則輸出"The node does not exist." 於一行。注意，在此題中，id $= 0$ 的節點為鏈結串列的開頭。

模板題。在實作 linked list 時，我們通常會在最前面加上一個空節點，以利插入操作的實作方便以及一致性。

```
1  struct linked_list {
2      struct node {
3          node *prev, *next;
4          int id;
5          node(): prev(NULL), next(NULL){}
6      };
7      vector<node*> List;
8      // list[x] 儲存指向 id = x 的指標，這就是 linked list 通
          常搭配陣列的地方
9      linked_list(int n): List(n+1,NULL) {
10         List[0] = new node; // 起始空節點
11         List[0]->id = 0;
12     }
13     int prev(int x){ // 查詢前一項的 id
14         if(List[x] && List[x]->prev)
15             return List[x]->prev->id;
16         return -1;
17     }
18     int next(int x){ // 查詢下一項的 id
19         if(List[x] && List[x]->next)
20             return List[x]->next->id;
21         return -1;
22     }
```

```
23      bool Insert(int x, int id) {
24          // 在 id = x 的節點之後插入 id = id 的節點
25          if(!List[x]) return false;
26          node *n = new node;
27          List[id] = n;
28          n->next = List[x]->next;
29          if(n->next) n->next->prev = n;
30          List[x]->next = n;
31          n->prev = List[x];
32          n->id = id;
33          return true;
34      }
35      bool Delete(int x){
36          // 刪除 id = x 的節點
37          if(!List[x]) return false;
38          if(List[x]->next)
39              List[x]->next->prev = List[x]->prev;
40          List[x]->prev->next = List[x]->next;
41          delete List[x];
42          List[x] = NULL;
43          return true;
44      }
45      void travel(){ // 依序尋訪所有節點
46          node *s = List[0]->next; // start
47          while(s){
48              printf("%d ",s->id);
49              s = s->next;
50          }
51          puts("");
52      }
53  };
```

list 的使用時機：當題目要考慮某個 id 的前後項的時候，便可以考慮使用 linked list 解題

習題 1.6.2: 陸行鳥大賽車（NeOJ 21）

輸入說明

第一行包含一個整數 $N(N \leq 10^5)$，代表現在有 N 個玩家，編號 1～N 的玩家目前分別為第 1～N 名 (編號 1 第 1 名、編號 2 第 2 名...)。
第二行包含一個整數 $M(M \leq 50,000)$，代表接下來會依序發生 M 個事件。
接下來的 M 行，每行包含兩個整數 $T_i, X_i(0 \leq T_i \leq 1, 1 \leq X_i \leq N)$，$T_i$ 為 0 的時候，代表編號 X_i 的玩家遭受攻擊，然後離開遊戲；T_i 為 1 的時候，代表編號 X_i 的玩家使用衝刺，無條件超越當前名次比他高一名的玩家。
測試資料保證玩家被淘汰之後不會再出現任何紀錄。

輸出說明

輸出一行，包含 Y 個整數 (Y 是剩餘玩家的數量)，由名次小到大依序輸出，整數兩兩間以空白隔開。

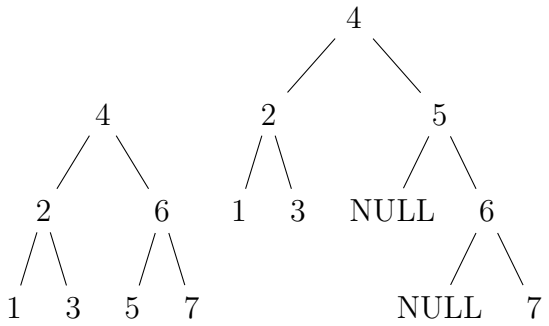
對於初始序列的建構，轉化成按照名次順序插入節點就行了。刪除節點的操作，模板就有直接支援。超越的部分比較麻煩，對於每一筆超越的操作，可以想像成先將前一名的節點刪除，再插入到自己的後面，因為 linked list 對於 id 的插入、刪除操作都是 $O(1)$ ，所以總複雜度 $O(N+M)$。

```
int main(){
    int n, m, operation, id;
    cin >> n >> m;
    link_list List(n);

    // 初始化
    for(int i=0;i<n;i++)
        List.Insert(i,i+1);

    while(m--){
        cin >> operation;
        if(operation == 0) {
            // 刪除節點 id
            cin >> id;
            List.Delete(id);
        } else {
            // 超越前一名
            cin >> id;
            int p = List.prev(id);
```

```
20              if(p!=0){
21                  int pp = List.prev(p);
22                  List.Delete(p);
23                  List.Delete(id);
24                  List.Insert(pp,id);
25                  List.Insert(id,p);
26              }
27          }
28      }
29      List.travel();
30  }
```

1.7 二元搜尋樹 (Binary Search Tree)

這是一個很重要的觀念，要好好學！**這裏會用到一些基本圖論的用語，如果還是不熟的話可以去複習一下**。對於一個**二元樹**（也就是每一個節點都有少於或等於兩個子節點的樹）賦值，使得對於所有節點，**其右節點的子樹中的數字都比這個節點大，而左節點的子樹中的數字都比這個節點小**（稱為二元搜尋樹性質）。什麼意思？且看圖中分解。以下是對於集合 $\{1,2,3,4,5,6,7\}$ 所建的兩棵二元搜尋樹：



對於一棵二元尋樹上的所有節點，其右子樹中的數字都比這個節點大；左子樹中的數字都比這個節點小

為什麼這個重要呢？來看下一個段落！

二元搜尋樹所能做的有趣的好玩的事

這是一個二元樹的節點，每個節點有三個指標分別指向自己的父節點以及左右子節點。每個指向 ‘node‘型別的指標都可以視為一棵二元樹，因此在這個節點

內部實際上是存了三棵樹。

```
struct node {
    node *parent,*lson,*rson;
    int val;
    node(int data):
        parent(NULL), lson(NULL), rson(NULL), val(data){}
};
```

一個指標需要占 8 個 bytes，所以用指標實作的東西都要花蠻多的記憶體，在時間上常數也會比較大。

插入一個元素

假設已經有一個二元搜尋樹了，那要怎麼插入一個新元素？假設這個元素的值為 x，且目前走到的節點的值為 y（一開始當然是從根節點開始走），那該放這個節點的哪裡呢？根據二元搜尋樹的定義，如果 $x < y$，則往左走；反之，則往右走。那如果沒得走了（也就是要走的節點為 NULL），那就將本來應該要走的節點設為 x，就好了。

```
void Insert(node *&x, int val) {
    if(!x) {
        x = new node(val);
        return x->parent = NULL, void();
    }
    if(x->val > val){
        if(x->lson) Insert(x->lson, val);
        else x->lson =
            new node(val), x->lson->parent = x;
    }
    if(x->val < val){
        if(x->rson) Insert(x->rson, val);
        else x->rson =
            new node(val), x->rson->parent = x;
    }
}
```

如何建立二元搜尋樹

為什麼要先講插入再講感覺比較基本的建立？因為呢，對於一個序列 $[a_1, a_2, \cdots a_n]$，將 a_1 設定為根，再對 a_2、a_3、a_n 等，依序插入，插入完了就建立完成！

拜訪這棵樹

中序尋訪是二元樹的拜訪方式之一，利用中序尋訪一個二元搜尋樹可以將資料結構內部的值按照大小排序輸出。因此二元搜尋樹是一個有序 (ordered) 的資料結構。

```
1 void travel(node *x){
2     if(!x) return;
3     travel(x->lson);
4     printf("%d ", x->val);
5     travel(x->rson);
6 }
```

另外還有前序與後序尋訪方式，在其他領域上用得到。

在樹上查詢

在這個二元搜尋樹上，根據值找到節點（或者是查看是否存在，找在樹上最靠近一個值的點之類的），都可以在樹上跑來跑去來搜尋。假設要搜尋的值是 x，然後目前節點的值是 y，那也可以重複以上的動作：$x < y$ 則往左走，反之則往右走，結束條件就是沒得走或找到了。

```
1 node *Find(node *x, int val) {
2     if(!x) return NULL;
3     if(x->val == val) return x;
4     else if(x->val > val) return Find(x->lson, val);
5     else return Find(x->rson, val);
6 }
```

對一個節點 Say Goodbye

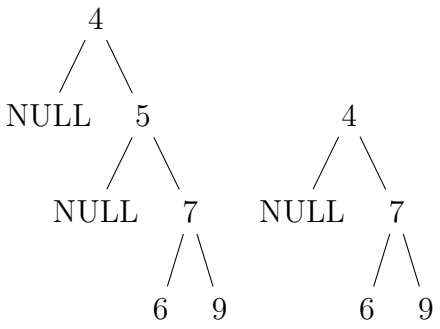
如果想要刪除一個節點，那得分三個 case：依照是不是葉節點（也就是有沒有子節點），可以看出需不需要找替換來遞補原本的節點的位置。如果是葉節點就直接刪掉即可，否則需要找一個替身。那作法：

```
1 bool Delete(node *&root, node *x) {
```

1. 如果是葉節點，則直接刪掉。

```
1 if(!x) return false;
2 if(!x->lson && !x->rson){
3     if(x->parent) (x->parent->val > x->val)?
4         x->parent->lson = NULL:
5         x->parent->rson = NULL;
6     delete x;
7 }
```

2. 如果其中一個子節點為空，則可以直接拉上來



假設要刪除 5 發現他只有一個子樹，那就可以直接把 4 連到 7 就好了（從左圖變成右圖）。

```
1 else if(!x->lson){
2     if(x->parent){
3         (x->parent->val > x->val)?
4         x->parent->lson = x->rson:
5         x->parent->rson = x->rson;
6     } else root = x->rson;
7     x->rson->parent = x->parent;
8     delete x;
9 }
10 else if(!x->rson){
11     if(x->parent){
12         (x->parent->val > x->val)?
13         x->parent->lson = x->lson:
```

```
14          x->parent->rson = x->lson;
15      } else root = x->lson;
16      x->lson->parent = x->parent;
17      delete x;
18  }
```

3. 如果不是，則**需要找一個節點當替身**。

引理 1.7.1: 替身的條件

找到的替身要符合：如果將要刪除的節點設為替身的值，並且將替身刪掉之後，還是會符合二分搜尋樹性質。

看左邊子樹，然後開始往右找到底。例如：如果選了左子樹，那走了之後，開始一直往右子樹走，直到不能再走了，那就將原本的節點的值換成那個點，然後遞迴刪除那個替身。

```
else{
    node *exchange = x->lson;
    while(exchange->rson) exchange = exchange->rson;
    x->val = exchange->val; // copy the data
    Delete(root, exchange);
}
```

最後再加上一行就可以了。

```
    return true;
}
```

刪除一個值

當我們要在二元搜尋樹中刪除一個值時，我們可以先找到它，再把節點刪除。

```
bool Delete_Val(node *&root, int val){
    return Delete(root, Find(root, val));
}
```

體驗二元搜尋樹的美好

整個二元搜尋樹的程式碼在此：https://pastebin.com/ED77CYHV

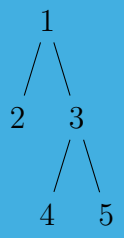
習題 1.7.1: 二元搜尋樹 (TRVBST) (TIOJ 1609)

一個序列依照插入的順序可以排成許多不同的二元搜尋樹，而給你 N 個不同的整數 a_i，依序為插入的順序，請問所構成的二元搜尋樹的中序遍歷為何？($N \leq 10^6$，$-2^{30} \leq a_i \leq 2^{30}$)

二元樹的表示方法：想要輸出一個二元樹，有三種比較常用的方法：

1. 前序：先輸出自己，再照著前序輸出左子樹，再照著前序輸出右子樹
2. 中序：先照著中序輸出左子樹，再輸出自己，再照著中序輸出右子樹
3. 後序：先照著後序輸出左子樹，再照著後序輸出右子樹，再輸出自己

假設用這個樹來當例子：



那前中後序分別為：1 2 3 4 5、2 1 4 3 5、2 4 5 3 1。

做事的代價

現在我們要分析一下以上介紹的運算的複雜度：可以知道，如果這棵樹的高度為 h（最深的節點深度），則最壞的情況就是在插入或搜尋的時候，每次都遇到最下面的節點，則複雜度 $O(h)$。那 h 大概會是多少呢？可以觀察：第一層最多可以有 1 個節點，第二層最多可以有 2 個節點，第三層最多可以有 4 個節點..... 第 k 層最多可以有 2^{k-1} 個節點，總共可以有 $\sum_{k=1}^{h} 2^{k-1} = 2^h - 1$ 個節點，所以

$$n \approx 2^h \implies h \approx \log(n)$$

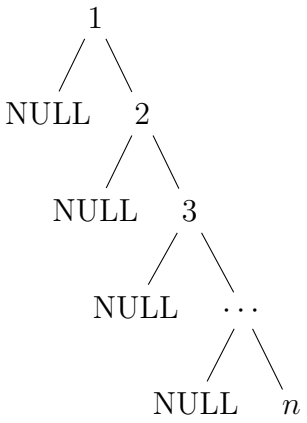
所以如果是一個夠平衡（待會會看不平衡）的二元搜尋樹，則搜尋和插入的複雜度都是 $O(\log n)$。

定理 1.7.1: 搜尋樹的複雜度

給定一個 n 個節點的隨機生成的二元搜尋樹，其搜尋和插入的平均複雜度為 $O(\log n)$

整人搜尋樹

現在我們要對這個序列建立搜尋樹：$[1, 2, 3, 4 \cdots n]$，那結果將會是：



這種情況稱為**退化**。這樣子呢，$h = n$ 而搜尋和插入都變很可悲的 $O(n)$ 了。以後會學到如何避免這個狀況，現在遇到就祈禱題目是隨機產生不會搞你就好了！

平時要多做好事，才不會比賽的時候被測資雷 (X
測資一定會想辦法卡掉這個，所以才需要平衡二元樹 (O

1.8 集合 (Set)

這是 STL 提供的 Binary Search Tree，內部實作是噁心的紅黑樹（Red-Black Tree），是一個實作複雜（所以直接用）但是可以自動平衡避免退化的資料結構。STL 已經幫你實作出來了，用就好了！‘set‘ 裡面的東西不能有重複，插入、查詢、刪除的複雜度都是 $O(\log n)$。看以下程式片段：

```cpp
set<int> se; // 宣告一個元素型別為 int 的集合
se.insert(1); // 插入元素 1
se.insert(2);
se.insert(5);
cout << se.count(1) << endl; // 查找元素，true
cout << se.count(3) << endl; // false
```

```
se.erase(1); // 刪除元素，若集合內不含此元素則回傳 false
cout << *se.find(3) << endl;
// find() 回傳 iterator，若找不到則回傳 se.end()
cout << *se.lower_bound(3) << endl;
cout << *se.upper_bound(3) << endl;
// 因為 set 有序，所以也支援二分搜操作
```

‘multiset‘

STL 也支援可以重複元素的集合，用法與 ‘set‘差不多：

```
multiset<int> se; // 宣告一個元素型別為 int 的 multiset
se.insert(1); // 插入元素 1
se.insert(1); // 可以重複插入
se.insert(5);
cout << se.count(1) << endl; // 計算元素個數，2
cout << se.count(3) << endl; // 0
se.erase(1);
// 一次刪除所有元素 1，若集合內不含此元素則回傳 false
se.erase(find(5)) // 一次刪除一個元素 5
```

當然，‘set‘與 ‘multiset‘也支援 ‘size()‘與 ‘empty()‘，在此不做贅述了。

‘set‘的遍歷

‘set‘是一棵二元搜尋樹，當然可以進行遍歷的操作。STL 的二元搜尋樹沒辦法存取一個子樹的左右節點，不過 STL 提供了迭代器 (iterator)，可以用不同的方式進行遍歷。

```
for(set<int>::iterator it = se.begin(); it != se.end(); it++)
    cout << *it << ' ';
```

或者用 C++11 後提供的 ‘auto‘ 關鍵字進行尋訪

```
for(auto &val: se)
    cout << val << ' ';
```

習題

不要被梗！

習題 1.8.1: 今晚打老虎 (ZJ a091)

奕辰找到了一個神奇的機器，可以做三件事：

1. ‘INSERT ‘ x，也就是把一個數字輸入進去這個機器
2. ‘QUERY MIN‘，也就是查詢目前的最小值
3. ‘QUERY MAX‘，同上，查詢目前的最大值

而且，**只要一個數字被查詢過了，機器就會把他吐出來，代表不在機器中了！**同一時間內不會超過 10^6 個數字，且數字可以被 ‘int‘存下。

習題 1.8.2: 好多燈泡 (TIOJ 1513)

很會找東西的奕辰找到了一張紙條：上面寫了 N 個數字！他在一個很多燈泡和開關的房間裡，每一個燈泡和開關都有編號，被相同編號的開關所控制。一開始每一盞燈都是關的。現在，紙張上寫了數字代表操作順序，第 i 個數字代表要去操作第 i 盞燈（關的變成開的，開的變成關的），已知最後操作了之後只有一盞燈是亮著的，並且直視那盞燈可以獲得神奇魔法！但是奕辰最近忙於看片，沒辦法以一操作，所以請你幫他寫程式，讀入 N 和 N 個指令，輸出哪一個燈泡最後亮著。($N \leq 10^5$)

註：也可以想想 $O(n)$ 解喔 $><$

1.9 映射 (Map)

基本上跟上面的一樣，可是不是存數字而是存兩個值，‘key‘和 ‘value‘（也就是存一個 ‘pair‘，第一個存 ‘key ‘，第二個存 ‘value‘。紅黑樹依照 ‘key‘排列，所以是依照 ‘key ‘搜尋。用處類似一個可以存任何東西，複雜度稍微高一點（同上）的陣列。跟 ‘set‘一樣，‘map‘的 ‘key‘值不能有重複。

```
map<string,int> mapp;
// 宣告一個由 string 映射到 int 的 map
mapp.insert({"apple", 1});
// 插入一個 key 值為 "apple" 的節點
mapp["book"] = 2; // 也可以這樣插入
mapp["apple"] = 3; // 改值
cout << mapp.count("apple") << endl; // true
cout << mapp.count("my girlfriend") << endl; // false
```

```
9
10 for(map<string,int>::iterator it = mapp.begin(); it != mapp.
      end(); it++)
11     cout << it->first << ' ' << it->second << endl;
12
13 for(auto &val: mapp)
14     cout << val.first << ' ' << val.second << endl;
15
16 // apple 3
17 // book 2
```

如果要看一個 ‘map‘裡面的東西，但是不確定它存不存在（像是以下程式）
‘if(mapp[”orange”] == 2) //...‘
那 ‘mapp‘就會先開一個 key 為 ‘orange‘的點，佔空間！所以要先判斷存不存在：
‘if(mapp.count(”orange”) && mapp[”orange”] == 2) //...‘
才不會浪費許多不必要的記憶體。

‘multimap‘

同樣的 STL 也有 ‘multimap‘這種東西，目前我還想不到實際應用，想知道詳細的可以看 cpp reference。

跟著蕭電這樣做

：知道 ‘ multimap ‘的應用的人麻煩私訊我，筆者感激不盡 OwO

習題

‘map‘ 是個好用的東西喔！

習題 1.9.1: Hardwood Species（UVa 10226）

恭喜成為森林管理者！現在你的第一個工作就是計算森林中樹的比例。有 N 棵樹，給你他們的英文名稱（由大小寫英文字母表示），請輸出他們所佔的比例為何。（比例定義為：$\frac{\text{樹出現的次數}}{N}$），（至多 10^4 種樹、10^6 棵樹，請輸出至小數點後四位。）

習題 1.9.2: 連通塊數量 (OJDL)

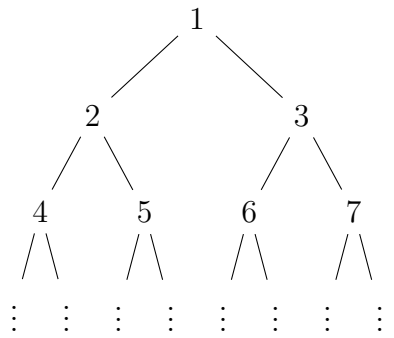
給平面上 N 個點 $(N \leq 10^6)$ 的 x, y 坐標 $(x, y \leq 10^9)$。若兩個點的 x 坐標或 y 坐標只差 1 可以視為同一個連通塊，求連通塊數量。

1.10 堆積 (Heap)

堆積的定義比較不能望文生義：它也是一個二元樹，可是不滿足以上的二元搜尋樹性質，而是符合堆積性質：對於一個 Min-Heap，一個節點的值一定比其子樹的每一個節點的值都小，對於一個 Max-Heap，則是都大。還有一個重要的性質就是：通常實作的時候會把這個樹當成一個**完全二元樹**（也就是除了最後一階可能不填滿其他都滿）的樹，這個好處就是可以用陣列存這個樹。

定理 1.10.1: 用陣列存二元樹

給定一個陣列，可以用陣列來存二元樹，編號就是 BFS 順序（或者白話一點就是橫著編號）：



經觀察可以發現：編號為 i 的節點左節點為 $2i$，右節點為 $2i+1$，而其父節點為 $\lfloor \frac{i}{2} \rfloor$。如果是完全二元樹，就可以發現只需要 $O(n)$ 的空間來存；但是如果退化成鍊可就糟糕了：需要 $O(2^n)$ 的空間存，所以要取捨。這個概念不只在這個 Heap 中會用到，所以得獨立出來講！

顯然，對於一個數字的集合，也會有許多的 Min(Max) Heap 可能。以下所講的 Heap 皆指 Min-Heap，至於 Max-Heap 的實作就只是把不等號反過來就好了。要支援的操作比較多，以下列出來之後我們慢慢揭曉。最後的節點指的是最後一層最右邊的節點。

1. ‘int getMin()‘，代表回傳堆疊根節點的值（也就是最小值），複雜度 $O(1)$。

```
1 int getMin(vector<int> &heap) {
2     if(heap.empty())
```

```
3           printf("Are you kidding?\n");
4       else return heap[0];
5   }
```

2. ‘void insert(int x) ‘ 要加入一個值為 x 的數字進去 Heap 裡面。作法就是先將 x 先丟進去最下面，然後一直往上，只要發現 x 現在的節點比其父節點的值小，就交換他們兩個，並繼續看。

```
1  void Insert(vector<int> &heap, int x) {
2      heap.push_back(x);
3      int loc = heap.size()-1;
4      while(loc){
5          if(heap[loc] < heap[loc>>1])
6              swap(heap[loc], heap[loc>>1]);
7          loc >>= 1;
8      }
9  }
```

3. ‘void deleteRoot(int x) ‘ 刪除根節點。將根換成最後的節點（最後的節點也刪掉），然後開始遞迴往下：如果目前的值比左右子樹的最小值大，則與左右子樹的最小值交換，並且往那個子樹遞迴下去。

```
1  void deleteRoot(vector<int> &heap) {
2      swap(heap[0], heap[heap.size()-1]);
3      heap.pop_back();
4      int pos = 0;
5      while(pos < heap.size()) {
6          int l, r, Min;
7          l = (2*pos >= heap.size()) ?
8              INF : heap[2*pos];
9          r = (2*pos+1 >= heap.size()) ?
10             INF : heap[2*pos+1];
11         if(l == INF && r == INF) break;
12         Min = (l < r) ? 2*pos : 2*pos+1;
13         if(heap[Min] < heap[pos])
14             swap(heap[pos], heap[Min]), pos = Min;
15         else break;
16     }
17 }
```

不難發現，‘insert() ‘和 ‘deleteRoot() ‘的複雜度都是 $O(\log n)$，因為最差就是一直遞迴到底，而因為是完全二元樹，就有 $O(\log n)$ 層。

在轉角與堆積邂逅

這是 code: `https://pastebin.com/zkaMe93Z`

STL 內的 Heap

當然，超棒的 STL 有提供 Heap 可以用，稱為 ‘priority_queue‘，但是不一定是依照以上的實作，可能有出入。如果想要看另外一種複雜的 Heap，可以去查 Fibonacci Heap。

宣告

我們可以用下列方法宣告一個 ‘priority_queue‘

```
priority_queue<int> pq; // 宣告一個整數的 max-heap
```

若要自行定義比較函數 (也就是定義” 小於”)，可以下列方式進行：

```
priority_queue<int, vector<int>, greater<int> > min_pq;
// 把"小於"定義成"大於"，也就是 min-heap
priority_queue<int, vector<int>, comp> custom_pq;
// 自訂義小於比較
```

‘priority_queue‘的比較函數稍微複雜一點，要用 ‘struct‘ 的 ‘operator()‘ 來實作，以下是實作方法：

```
struct comp{
    bool operator()(int a,int b){
        return a%10 < b%10;
    }
};
```

三個操作

STL 內建的 ‘priority_queue‘一樣支援了 heap 該有的操作：

‘push()‘ 可以將資料丟入堆積的內部；

‘pop()‘ 將最大元素刪除；

‘top()‘ 查詢堆積內的最大元素；

```cpp
pq.push(3);
pq.push(2);
cout << pq.top() << endl; // 3
pq.pop();
cout << pq.top() << endl; // 2
```

當然也支援查詢資料結構的大小：
‘size‘ 函數查詢目前還位於堆疊中的資料數；
‘empty‘ 函數回傳堆疊是否為空。

千萬不要對一個空的 ‘priority_queue‘呼叫 top() 函數，不然你會吃個 RE

習題

不，第一題不是 DP 題！

習題 1.10.1: Add All (UVa 10954、ZJ d221)

給你 N 個數字 $a_1, a_2, \cdots, a_N$，請把這些數字加起來。很簡單？但是但是，要加兩個數字 x 和 y 有一個條件：必須付出 $x + y$ 的代價。請問要把這 N 個數字加起來的最小代價為何？($2 \leq N \leq 5000$)

(例：如果 $N = 3$ 且數列為 $[1, 2, 3]$，則最好的作法就是先加 1 和 2，花 3，再把兩個 3 加起來，花費 6，總共花費 $3 + 6 = 9$。

習題 1.10.2: 排隊買飲料 (TIOJ 1999)

有一天，你經過了一家飲料店，發現有 N 個人排隊要買飲料。不過，這家飲料店人手充足，一共有 M 個店員可以服務這些排隊的人潮。為了公平起見，雖然店員很多，但是排隊只排成一列，避免在排很多列的情況下，每列前進的速度會不一樣。

另外，為了服務品質起見，這家店的店員必須遵守兩個規則：必須按照客人排隊順序服務客人，不能先服務排在後面的顧客，而且，如果某個店員服務了某個客人，則該客人點的所有飲料都要由該店員製作，製作完成之後才能服務下一位客人。

你做了一下市場調查，詢問每位排隊的客人要買幾杯飲料。假如所有的店員製作飲料的速度都是每分鐘 1 杯，在遵守這些規則的前提下，請問服務完這些客人至少需要多久？

($N \leq 10^6$、$M \leq 10^4$，輸入完了之後會有 N 個數字，為每個排隊顧客要買多少飲料（最多 1000 杯）

1.11 位元集 (Bitset)

很多時候，會想要存布林陣列，但是你可知道：一個布林值居然佔了八個位元?! 如果題目出機車一點壓記憶體可能就要用到這一招：‘bitset‘。就是一個加強版布林陣列，除了終於讓每一個布林值佔一個位元（也就是空間佔八分之一）之外還加了許多的功能，如反轉，變成字串或數字（‘unsigned long long int‘之類的）的功能！請看以下程式片段。

建構元

```
bitset<10> bs; // 宣告 10-bit 的位元集，初始化為 0
bitset<10> bsA(71); // 將 71 表示為二進制存入
bitset<10> bsB("1010111"); // 也可以這樣初始化
```

運算子

```
cout << bsA << endl; // 輸出
cout << (bsA & bsB) << endl; // AND
cout << (bsA | bsB) << endl; // OR
cout << (bsA ^ bsB) << endl; // XOR
```

```
cout << ~bsA << endl; // NOT
cout << bsA[0] << endl; // 下標運算子
cout << (bsA<<1) << endl; // 左移右移
```

計數技術

```
cout << bsA.size() << endl; // 一開始宣告的bit數
cout << bsA.count() << endl; // 輸出總共有幾個 1
```

型別轉換

```
string s = bsA.to_string();
unsigned int x = bsA.to_ulong();
```

1.12 雜湊表 (Hash table)

先講一下何謂雜湊（Hash）：這個是 Hash Brown，中文叫做薯餅：

可是不是我們要的主題。我們要的是**雜湊（Hash）**！先來看維基百科的定義：

> 「雜湊函式（英語：Hash function）又稱雜湊演算法，是一種從任何一種資料中建立小的數字「指紋」的方法。雜湊函式把訊息或資料壓縮成摘要，使得資料量變小，將資料的格式固定下來。該函式將資料打亂混合，重新建立一個叫做雜湊值（hash values，hash codes，hash sums，或 hashes）的指紋。雜湊值通常用一個短的隨機字母和數字組成的字串來代表。」

簡單來說，就是讓一個很難表示的東西透過某個函數變成一個比較好表示的東西！假設想要對生日來 Hash，那一個生日可能是 2002/11/05，不好處理，那我可能會壓縮成一個字串 ‘”2002/11/05”‘或表示成一個 b 進位的數字 $\pmod p$，b，p 為相異質數之類的，就是為了方便存，而這個函數就叫做 Hash Function。顯然，如果 Hash 出來的東西不一樣，則兩個東西一定不一樣；但是如果 Hash 出來的東西一樣，不代表是一樣的。雖然如此，但是如果好好選 Hash Function，就可以避免碰撞（Collision）了。現在來分析這個可能性：

定理 1.12.1: 雜湊碰撞

假設一個 Hash Function 有 N 個可能值，然後有 M 個東西要被雜湊。每一個都相異的機率為 $\binom{N}{M} \cdot \frac{1}{N^M}$，故有出現碰撞的機率為 $1 - \binom{N}{M} \cdot \frac{1}{N^M}$。這個會隨著 M 而增加，所以在用 Hash 前得先稍微估一下。

這個的實作方法通常是用一個陣列來做，這樣均攤複雜度可以到 $O(1)$ 查詢，$O(1)$ 修改。維持一個長度為 n 的陣列 A，每次要加入一個東西 k 的時候，就把 k 存在 $A[\text{hash}(k) \pmod n]$ 的地方即可。通常也會搭配 Linked List，將 A 的每一個元素視為一個 Linked List 的頭，插入的時候如果遇到已經有東西了就會將 k 放在 $A[\text{hash}(k) \pmod n]$ 的最後頭。刪除也差不多，刪除 k 就到 $A[\text{hash}(k) \pmod n]$ 去看，找到了之後刪除。這樣的好處是平均上來說複雜度非常好，但是如果遇到雷測資就會爆複雜度。

如何選擇雜湊函數

一個好的雜湊函數需要那些特質呢？當然，需要有相當的平均性：不可以輸入一堆東西都出來是同樣的！第二個重要的特質是簡單計算：如果一個雜湊函數需要 $O(n^2)$ 的時間計算，反而會將演算法變慢。通常，喜歡把一個字串表示成一個 p 進位的數字，$a = 1, b = 2 \cdots$ 類推，例如 $\text{hash}(”abc”) = 26^2 \times 1 + 26^1 \times 2 + 26^0 \times 3 = 731$，然後為了防止值太大會模一個大質數，如 10003457 之類的。好一點（或是機車一點）的測資會卡常見的雜湊函數哦！所以可以自己想一想屬於自己的雜湊函數。

‘unordered_set‘

STL 的 hash table 叫做 ‘unordered_set‘，它與 ‘set‘ 一樣，都支援了插入、查詢、刪除的功能，唯一有差別的是因為它的實作原理是 hash ，對於每次操作的複雜度都會是 $O(1)$，但是這樣一來就會沒有二元搜尋樹的良好性質，所以不能按照順序遍歷輸出。

```cpp
unordered_set<int> use; // 宣告一個元素型別為 int 的集合
use.reserve(N); //reserve N個記憶體可以加速程式
use.insert(1); // 插入元素 1
use.insert(2);
use.insert(5);
cout << use.count(1) << endl; // 查找元素，true
cout << use.count(3) << endl; // false
use.erase(1); // 刪除元素，若雜湊表內不含此元素則回傳 false
// 不支援 upper_bound, lower_bound, find 等二分查找功能
for(auto &val: use)
    cout << val << ' ';
// 可以遍歷，但不會照順序
```

‘unordered_map‘

‘unordered_map‘ 的用法與一般的 ‘map‘ 也差不多，實作原理也是個雜湊表。大部分的成員函數以及運算子都與 ‘map‘ 一樣，複雜度 $O(1)$。

```cpp
unordered_map<string,int> umapp;
// 宣告一個由 string 雜湊到 int 的 unordered_map
umapp.insert({"apple", 1});
umapp["book"] = 2;
umapp["apple"] = 3;
cout << umapp.count("apple") << endl; // true
cout << umapp.count("my girlfriend") << endl; // false

for(auto &val: umapp)
    cout << val.first << ' ' << val.second << endl;
// 支援遍歷，一樣是無序的
```

也可以自己寫雜湊！這是 ‘unordered_map‘的定義範例（假設是把 ‘string‘透過雜湊函數變成 ‘unsigned long‘）：

‘unordered_map<‘
‘std::string, //key‘
‘unsigned long, //value‘
‘std::function<unsigned long(std::string)> ‘ ‘//hash: string 變成 unsigned long‘ ‘std::function<bool(std::string, std::string)> //key 的比較函數 ‘
‘> mymap(n, hashing_func, key_equal_fn<std::string>); ‘ ‘//mymap 初始大小，雜湊函數，key 的比較函數 ‘

雜湊表雖然 $O(1)$，但是也有比平衡二元樹慢的時候。當你的鍵值為非常龐大的資料結構時 (例如：超長 string)，在測資隨機生成的條件下，二元搜尋樹的比較函數通常只要比對前幾個字元就能快速查找；雜湊則要花字串長度的線性時間才有辦法計算出一個雜湊值。所以要看輸入資料的類型進行取捨，不要看到無關順序的東西就用 ‘unordered_map‘砸。

實戰演練

這題是 TOI 入營考的題目，時限卡得非常緊，用了常數比較大的 ‘map‘ 一定會 TLE，並且拿到跟暴力 $O(n^4)$ 一樣的分數。對於本題來說，必須考慮使用常數較小的方法 (如排序)，或者直接使用 $O(n^2)$ 的演算法。

習題 1.12.1: 四點共線 (TOI 2019 初選 pA)

給你 n 個點的 x, y 坐標 ($n \leq 3000$，$x_i, y_i \leq 10^4$)，每個點的編號為它們的輸入順序。求編號字典序最小的共線四點，若不存在四點共線則輸出 -1。