基礎資料結構

baluteshih

2019 年 9 月 18 日

1 何謂資料結構？

顧名思義，資料結構就是拿來儲存資料的一種「結構」。舉個例子，最常見的就是各位應該要會的陣列，陣列的功用這裡就不多說了。一個典型的資料結構通常都會支援以下三種操作：

1. push 將資料放進資料結構內。
2. pop 將資料從資料結構內拿出。
3. query 詢問使用者需要的資料，由資料結構維護。

資料結構是一種工具，在程式設計上我們根據不同用途來選擇不同的資料結構，並考慮其時間、空間，甚至是 coding 複雜度。
由於他是一種工具，儘管某些題目純需要資料結構就能 AC，但其實它們常常需要搭配不同的演算法來解題。

2 標準模板庫 (Standard Template Library)

許多資料結構在 C++ 的函式庫裡面都找得到，而這種方便的工具就被我們稱為「標準模板庫」(StandardTemplateLibrary)，也就是所謂的「STL」，使用的時候一律都要加上 std::(當然你也可以 using namespace std;)。
不過自己實作過一遍資料結構絕對是好的，這樣有時候在真正遇到 STL 無法解決的問題時，又或是遇到不懂 STL 的運作方式時，才可以真正深入探討問題，甚至自己馬上實作出來，避免掉不必要的麻煩。所以在往後的教學中，除非遇上太難的資料結構，否則筆者會先提供該資料結構的實作方法，才會再提供相關的 STL 資訊。
但 STL 畢竟是函式庫，這種方便的工具將會被大量的程式使用，為了滿足多數的使用者，許多 STL 常常會犧牲一些時間或者是空間複雜度來滿足需求，而這樣的動作有時候反而

會導致常數過大，使得有些題目不自己實作一個出來一樣會 TLE，聽完這樣的敘述後，應該不會覺得先教寫法是多餘的了吧 (X。

講完一整串廢話了，那就進入下一個階段吧。

2.1 型別模板

在使用 STL 之前有個概念必須要知道。
當我們宣告一個陣列的時候，通常都是以型態 (T) ＋名稱＋大小來宣告，但 STL 本身就是一個資料結構，舉最常用的 vector 當例子 (下面會提到)，如果宣告一個 vector 的陣列，你得到的就真的是一個 vector 的陣列 (可以看成二維陣列)，而不是你想要的東西。
那要怎麼告訴 STL 儲存你要的型態呢？C++ 在這裡衍生出了模板（template）的概念，在宣告一個 STL(C) 時，我們會以 C<T> name 來宣告一個 STL。
EX: vector<int> v;
而模板裡裝的東西有時候甚至不只一個，舉 map 當例子，map 的模板就需要兩個型態來宣告。
EX: map<int,int> m;
當然，也有些 STL 需要的模板雖然定義是要多個，有時候也可以不用寫滿，但這必須要在允許的情況下操作。

2.2 迭代器

在 C 裡面，一般的變數我們可以用 &name 來取得變數的記憶體位址，而若要存取位址，我們便需要相對應的指標型態變數來存取位址。但這在 STL 上會出現許多問題，因為宣告 STL 得到的東西並不是單單一個變數，可能是一個 class，於是 C++STL 便提供了迭代器（Iterator）來完成等價的工作。
你可以把迭代器當成一種指標來用，而且它們其實是類似的，你可以在一個迭代器 x 上加上星號 *，由 *x 讀取到你想要的資料。不同 STL 模板的迭代器會有不同的用法，有些可以隨機存取 (甚至直接當作一般指標在用)，有些只能做 ++、- -這類的運算，有些甚至連- -都辦不到。
若你需要宣告一個迭代器，必須在把模板打完後在後面加上::iterator。
EX: map<int,int>::iterator x;
不用急著熟悉不同 STL 模板的迭代器，用久了自然也就熟了。

順帶一提，C++ 在許多容器中都有提供.begin() 和.end() 這兩個基本的迭代器，其中.end() 是指向最後一項的後一項，也就是不存在的記憶體，這讓遍歷容器顯得方便許多。
再補充一個，通常提供.begin() 和.end() 的容器也會提供.rbegin(),.rend()，意義上是把元素反過來看的同樣工具。

3 資料結構們

讓我們開始正題吧！

3.1 堆疊 (stack)

3.1.1 簡介

堆疊是什麼？各位可以想像自己在疊盤子，而資料就是一個個的盤子，然而你應該不行從下面抽出盤子對吧？(笑
這種 First in last out 的性質正是 Stack 在維護的事情，而在實作部分我們可以使用陣列或是 linked list(下面會提到) 實作。

3.1.2 實作

Algorithm 1: Stack

```
1  int stack[MAXN],top=-1;
2  void push(int data){
3      stack[++top]=data;
4  }
5  void pop(){
6      if(top==-1) return;
7      top--;
8  }
9  int query(){
10     assert(top>-1);
11     return stack[top];
12 }
```

以上三個基本操作有看懂嗎？
簡單來說就是拿一個變數 top 來維護現在資料的最尾端，-1 代表 stack 裡面是空的，其他操作就很簡單了。
stack 在往後的演算法當中將會是使用頻繁的一項資料結構，請務必要學起來 (?

3.1.3 STL

以下假設變數名為 s：

1. 標頭檔：<stack>
2. 建構式：stack<T> s
3. s.size()：回傳 stack 內容物的個數，複雜度 $O(1)$。

4. s.empty()：回傳 stack 是否是空的，複雜度 $O(1)$。
5. s.top()：回傳 stack 最尾端的元素，複雜度 $O(1)$。
6. s.push(T a)：在 stack 尾端加入元素 a，複雜度 $O(1)$。
7. s.pop()：刪除 stack 最尾端的元素，複雜度 $O(1)$。

3.1.4 習題

1. (ZJ b923) 實作 stack
2. (ZJ c123) stack 的應用
3. (ZJ d016) 後序運算式的使用
4. (ZJ a565) stack 的應用
5. (ZJ a813) stack 的經典應用

3.2 佇列 (queue)

3.2.1 簡介

相信英文好的人說不定已經猜到佇列主要是在維護什麼性質了。
是人應該都排過隊，沒有人喜歡被插隊 (理論上)，而佇列就是一個保守的隊伍。
First in First out，這正是佇列主要在維護的性質，實作部份一樣可以使用陣列或是 linked list 實作。

3.2.2 實作

Algorithm 2: Queue

```
1  int queue[MAXN],l=0,r=-1;
2  void push(int data){
3      queue[++r]=data;
4  }
5  void pop(){
6      if(r<l) return;
7      l++;
8  }
9  int query(){
10     assert(r>=l);
11     return queue[l];
12 }
```

queue 的部分我們會拿一個 l 代表排頭，r 代表排尾，於是當 r<l 的時候，就代表著 queue 裡面沒有任何資料。
queue 一樣是使用頻繁的資料結構之一，沒學好這兩個最好不要再前進了 (?
不過各位可能會發現一個很神奇的問題：如果我不斷地 push、再 pop，那 queue 不就會讀到不該讀的記憶體了？
為了解決這個問題我們會使用環狀佇列，或是使用 linked list，這裡打算提到的是前者。

Algorithm 3: Circular Queue

```
1  int queue[MAXN],l=0,r=-1;
2  void push(int data){
3      queue[(++r)%MAXN]=data;
4  }
5  void pop(){
6      if(r<l) return;
7      l++;
8  }
9  int query(){
10     assert(r>=l);
11     return queue[l%MAXN];
12 }
```

我們在所有存取陣列的數值上加上模運算，雖然常數大了些，不過 coding 複雜度將會大大地減少許多。
不過它其實還是會在環狀繞回來撞到前面資料的時候出現問題，但是如果出現這個問題，你覺得還能用環狀 queue 嗎 (笑。

3.2.3 STL

以下假設變數名為 q：

1. 標頭檔：<queue>
2. 建構式：queue<T> s
3. q.size(),q.empty()：同 stack。
4. q.front()：回傳 queue 最前面的元素，複雜度 $O(1)$。
5. q.push(T a)：在 queue 尾端加入元素 a，複雜度 $O(1)$。
6. q.pop()：刪除 queue 最前面的元素，複雜度 $O(1)$。

3.2.4 習題

1. (UVA 10935) 實作 queue

2. (ZJ c249) 穩定婚姻問題
3. (ZJ c223) queue 的演算法優化

3.3 鏈結串列 (Linked List)

3.3.1 簡介

前面已經多次提到 linked list 這個名詞了，不過千萬不要覺得它什麼都辦得到，它會讓你覺得很煩，但其實同時也是一項實用的資料結構。
串列的理念就是為了解決許多麻煩的操作，有多麻煩？
也許各位在撰寫 C++ 有時會想要中途插入一個值在陣列中間，又或是打算把整個陣列搬運到另一個陣列上。
而在不了解 linked list 的情況下，可能就會有人直接把陣列來個乾坤大挪移。這在時間複雜度上是極糟的，操作一次即需要 $O(n)$，那操作 n 次不就需要 $O(n^2)$！
linked list 能做到的，就是把這樣的過程壓縮成 $O(1)$，進而大大降低時間複雜度。
另外 linked list 在實作時，通常會搭配動態宣告的記憶體，同時也為程式運作上省下了不少記憶體。

3.3.2 實作

Algorithm 4: a node of LinkedList

```
1 struct node{
2     int data;
3     node *next=nullptr;
4 };
5 node *head;
```

上面我們可以看到，我們宣告一個 head 指標，這是拿來記錄一個串列的頭，而串列的尾巴即是 null。
於是我們只需要另外開一個陣列，紀錄每個 node 的記憶體位址，就可以針對指定的 node 進行 O(1) 的插入點動作了！
然而這樣的缺點就是 linked list 無法做到所謂的「隨機存取」，除非有指定的編號方式，否則要像陣列一樣讀取 array[x] 的這種操作在 linked list 上是辦不到的。
至於其他方便的操作將會在後面的練習題一一看到，屆時就可以體會 linked list 的強大以及恐怖之處了 (不要問，你會怕。

不過，當你開心地在實作 linked list 時，你有時可能會發現一個神奇的問題：
我要如何讀到現在這個節點的前一個節點呢？
這個問題我們會使用雙向串列來解決，也就是同時擁有兩個指標來指向前後。

Algorithm 5: a node of Doubly LinkedList

```
1 struct node{
2     int data;
3     node *back=nullptr,*next=nullptr;
4 };
```

最後的應用就交給各位自行學習了。

3.3.3 STL

list 的 STL 是雙向串列，對於 list 中的任何一項，都可以 $O(1)$ 知道它的前一項和後一項。但若是要存取第 i 項時的複雜度是 $O(i)$。以下假設變數名為 l：

1. 標頭檔：<list>
2. 建構式：list<T> l
3. l.size(),l.empty()：同 stack
4. l.push_front(T a),l.push_back(T a)：加入一個元素 a 在 list 的前面或後面，複雜度 $O(1)$
5. l.pop_front(),l.pop_back()：從 list 的前面或後面刪除一個元素，複雜度 $O(1)$
6. l.insert(iterator it,size_type n,T a)：在 it 指的那項的前面插入 n 個 a 並回傳指向 a 的迭代器。複雜度 $O(n)$。
7. l.erase(iterator first,iterator last)：把 [first,last) 指到的東西全部刪掉，回傳 last。複雜度與砍掉的數量呈線性關係，如果沒有指定 last，那會自動視為只刪除 first 那項。
8. l.splice(iterator it,list& x,iterator first,iterator last)：first 和 last 是 x 的迭代器。此函式會把 [first,last) 指到的東西從 x 中剪下並加到 it 所指的那項的前面。x 會因為這項函式而改變。若未指定 last，那只會將 first 所指的東西移到 it 前方。複雜度與轉移個數呈線性關係。

3.3.4 習題

1. (ZJ b938) 實作 linked list
2. (ZJ d718) linked list 的實際應用
3. (TIOJ 1225) 黑板上有 n 個數字寫成一排，每次選擇兩個相鄰的數字，把比較小的那個數字擦掉 (如果兩個數字一樣大，那麼擦掉任何一個都可以。) 然而，這些步驟需要花費留下來的數字大小，試問最小總花費。
4. (TIOJ 1930) 題敘略。

3.4 動態 array

3.4.1 簡介

有時候我們無法預估陣列所需要宣告的長度，又必須得宣告陣列，為了避免 MLE，將陣列動態宣告將會是好選擇。

3.4.2 實作

作法稍顯複雜，有興趣的讀者可以自行上網搜尋接下來提到的 vector 的實作方法，關鍵算法是倍增法。

3.4.3 STL

以下假設變數名為 v：

1. 標頭檔：<vector>
2. 建構式：vector<T> v
3. v[i]：回傳 v 的第 i 個元素，複雜度 $O(1)$。
4. v.size(),v.empty()：同 stack。
5. v.push_back(T a),v.pop_back()：同 list。
6. v.back()：回傳 vector 最尾端的元素，複雜度 $O(1)$。
7. v.resize(size_type a,const T& b)：強制將 v 的長度變為 a，若比原本長，則後面加 b 直到長度為 a，若比原本短則將多出的部分捨去，若無指定 b 將會預設為 T 的預設值。
8. u(比較運算子)v：回傳比較 u,v 字典序的結果。複雜度通常是 $O(max(sizes, sizet))$。

其實還有另一個名叫 string 的 STL，他的原身是 vector<char>，但由於太常用了所以被獨立了出來並做了優化，也可以輸出入。以下假設變數名為 s：

1. 標頭檔：<string>
2. 建構式：string s
3. s=t：讓 s 變得跟 t 一樣，複雜度不明，但通常是 $O(sizes + sizet)$。
4. s+=t：在 s 的尾端加上 t，複雜度通常是 $O(sizes + sizet)$。
5. s.c_str()：本函式會回傳跟 s 一樣的 C 式字串。複雜度 $O(1)$。
6. cin > > s：輸入字串至 s，直到讀到不可見字元。

7. cout < < s：輸出字串 s。
8. getline(cin,s,char c)：輸入字串至 s，直到讀到字元 c。未指定 c 時，c 是換行符號。

由於是出自 vector，所以 vector 的操作他也都有。

3.4.4 習題

1. (ZJ a011) getline 的應用
2. (ZJ d098) 有個支援 string 的東西叫做 stringstream，用法有點複雜，可以利用這題或配合網路資源學起來，頗有幫助。

3.5 雙向佇列

3.5.1 簡介

有時候你可能有了 queue 還不滿足，可能想要在隊伍的前方也能加入元素。
所以 deque 就誕生了。

3.5.2 實作

可自行在 queue 上稍作修改。

3.5.3 STL

deque 的操作和 vector 大同小異，以下假設變數名為 dq：

1. 標頭檔：<deque>
2. 建構式：deque<T> dq
3. dq.push_front(T a),dq.push_back(T a)：加入一個元素 a 在 deque 的前面或後面，複雜度 $O(1)$

其他 vector 有的東西他也都有。

題外話，deque 的常數很大，但很神奇的是 stack 和 queue 的預設容器全都取自他，也就是他們會直接宣告一個 deque 來當作 stack 或 queue 給使用者用。
當然容器是可以修改的，你可以在宣告 stack 的時候寫成 stack<T,list<T> >，這樣容器就會變成 list 了。

3.5.4 習題

1. (TIOJ 1618) dqeue 的應用
2. (TIOJ 1566) dqeue 的應用

3.6 pair

3.6.1 簡介

把兩個變數綁起來的方便工具，支援字典序比較。

3.6.2 STL

以下變數假設變數名為 p：

1. 標頭檔：<utility>
2. 建構式：pair<S,T> p，S 跟 T 可為不同型態。
3. p.first：回傳 p 的第一個值。
4. p.second：回傳 p 的第二個值。
5. make_pair(S a,T b)：回傳一個 (a,b) 的 pair。

3.7 tuple

3.7.1 簡介

如果兩個元素不滿足又不想自己刻 struct 的話就可以用 tuple，他支援三個以上的元素。
同樣支援字典序比較。

3.7.2 STL

以下變數假設變數名為 t：

1. 標頭檔：<tuple>
2. 建構式：tuple<T1,T2...> t，Ti 的型態可兩兩相異，只要你寫得出來就能無限往後加。
3. get<i>(t)：回傳 t 的第 i 個值。
4. make_tuple(T1 a1,T2 a2,...)：回傳一個 (a1,a2,...) 的 tuple。

3.8 優先佇列 (Priority Queue)

3.8.1 簡介

Priority Queue 主要是在維護一群數字的最大值，並支援插入數字、刪除最大值及詢問最大值的功能。
這個最大值可以根據你喜歡的比較方式重新定義，只要你有辦法比較任意兩元素的大小就

可以了。
Priority Queue 是個非常好用的工具，還請務必要熟悉他。

3.8.2 實作

Priority Queue 有非常多種實作方法，這裡將示範二元堆積 (Binary Heap) 的實作方法。

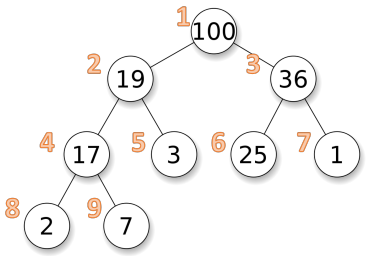

Figure 1: Heap 的普遍長相

如圖所示，Heap 是一顆完全二元樹 (Complete Binary Tree)(之後會更詳細的向大家解釋。)
除了完全二元樹的條件外，Heap 還得滿足以下兩種條件：

1. Heap 根節點上的值總是大於等於他左右子節點的值。
2. Heap 的左右子樹也是一棵 Heap。

這是一種遞迴的定義方法，可說是非常簡潔。
那要如何維護這樣的條件呢？我們可以直接看看 code。

Algorithm 6: Binary Heap

```
1  int heap[MAXN],top=0;
2  void push(int data){
3      heap[++top]=data;
4      for(int i=top;i>1;)
5          if(heap[i]>heap[i/2])
6              swap(heap[i],heap[i/2]),i/=2;
7          else break;
8  }
9  void pop(){
10     heap[1]=heap[top--];
11     for(int i=1;i*2<=top;)
12         if(heap[i]<heap[i*2])
13             swap(heap[i],heap[i*2]),i=i*2;
14         else if(i*2<top && heap[i]<heap[i*2+1])
15             swap(heap[i],heap[i*2+1]),i=i*2+1;
16         else break;
17 }
18
```

```
19 int query(){
20     assert(top>0);
21     return heap[1];
22 }
```

正如同圖片上所編碼的，我們令 1 號節點是根節點，這樣子必定能滿足節點 i 的左子節點是 $2i$，右子節點是 $2i+1$
加入元素的時候就放在最尾端，並不斷從下換上去；
刪除元素的時候，就把最尾端的元素換上來，然後不斷換下去；
回傳最大值就不用說了吧 XD？

Heap 的深度是很好預測的，由於每一層都強制要比上一層多一倍的數量的關係，所以深度會是 $O(\log size)$。
那與深度呈線性關係的加入和刪除自然也就是 $O(\log size)$ 了。

3.8.3 STL

priority_queue 有三個型別參數 T、Con 和 Cmp。T 是內容物的型別，Con 是所採用的容器 (預設為 vector)，Cmp 是比大小的依據。Cmp 的預設值是 less<T>，此時的 priority_queue 是最大堆，若改成 greater<T>，則 priority_queue 為最小堆，又或者是你可以自己定義。以下假設變數名為 pq：

1. 標頭檔：<queue>
2. 建構式：priority_queue<T,Con,Cmp> pq
3. 建構式：priority_queue<T,Con,Cmp> pq(iterator first,iterator last)：會有一個 priotiry_queue 內含 [first,last) 所指到的東西，複雜度與元素個數呈線性。
4. pq.size(),pq.empty()：同 stack。
5. pq.push(T a)：加入元素 a，複雜度 $O(\log size)$。
6. pq.top()：回傳最大值，複雜度 $O(1)$。
7. pq.pop()：刪除最大值，複雜度 $O(\log size)$。

3.8.4 習題

1. (ZJ b606) 給你 n 個數字，已知把兩個數字相加的代價是兩個數字的總和，試問把所有數字加起來的最小代價。
2. (104 全國賽 pA/TIOJ 1911) 請你維護一個同時能加入元素、回傳 + 刪除最大值的資料結構。
3. (TIOJ 1231) 題敘略。

3.9 平衡樹 (Balanced Search Tree)

3.9.1 簡介

當你想快速查找在一堆元素中：

1. 是否存在元素 x？
2. 大於 x 的最小值為何？
3. 小於 x 的最大值為何？

則平衡樹會是你的好選擇。
由於實作過於困難，這裡將不提及實作，直接切入 STL。

3.9.2 STL

第一個要介紹的是 set，他可以 $O(\log size)$ 插入、刪除元素和辦到以上事情，並保證不存在重複元素，因此我們會稱元素的值為鍵值 (Key)。以下假設變數名為 s：

1. 標頭檔：<set>
2. 建構式：set<K> s
3. s.size(),s.empty(),s.clear()：同 vector。
4. s.insert(K k)：加入元素 k，複雜度 $O(\log size)$。
5. s.erase(iterator first,iterator last)：刪除 [first,last)，若沒有指定 last 則只刪除 first，複雜度與 $O(\log size)$ 加上元素個數有關係。
6. s.erase(K k)：刪除鍵值 k，複雜度 $O(\log size)$。
7. s.find(K k)：回傳指向鍵值 k 的迭代器，若不存在則回傳 s.end()，複雜度 $O(\log size)$。
8. s.lower_bound(K k)：回傳指向第一個鍵值大於等於 k 的迭代器。複雜度 $O(\log size)$。
9. s.upper_bound(K k)：回傳指向第一個鍵值大於 k 的迭代器。複雜度 $O(\log size)$。

再來是增加了一些功能的 map，map 有兩個型別參數 K 和 T，K 是鍵值的型別，T 是對應的值的型別。
map 中每一個元素其實是 pair<K,T>，所以迭代器指向的東西是一個 pair。以下假設變數名為 m：

1. 標頭檔：<map>
2. 建構式：map<K,T> m

3. m.size(),m.empty(),m.clear(),m.erase(iterator first,iterator last),m.erase(K k), m.find(K k),m.lower_bound(K k),m.upper_bound(K k)：同 set。
4. m[k]：存取鍵值 k 對應的值，若 k 沒有對應的值，會插入一個元素，使 k 對應到預設值並回傳之。複雜度 $O(\log size)$。
5. m.insert(pair<K,T> k)：若沒有鍵值為 k.first 的值，插入一個鍵值為 k.first 的值對應到 k.second，並回傳一個 pair，first 是指向剛插入的元素的迭代器、second 是 true；若已經有了，回傳一個 pair，first 是指向鍵值為 k.first 的元素的迭代器，second 是 false。複雜度 $O(\log size)$。

這兩個最基本工具看似非常方便，但卻存在著某些缺點。

1. 常數過大 ($O(\log size)$)。
2. 無法存取重複元素。

當然這兩個問題是有辦法被分開解決的，以下簡介這兩項工具。

1. 常數過大 ($O(\log size)$)。
   改成 unordered_set,unordered_map，即可降低常數 (可說是少掉一個 log)。
   差異：喪失掉 lower_bound 及 upper_bound 的功能，迭代器只能做 ++ 運算，遍歷的時候也不會依據大小遍歷 (原本會)。
   需注意：標頭檔為 <unordered_set> 和 <unordered_map>
2. 無法存取重複元素。
   改成 multiset,multimap，即可加入重複元素。
   差異：map 喪失掉下標功能 (m[k])
   需注意：有一個函式 equal_range(K k)，會回傳一個 iterator 的 pair，第一項代表 lower_bound(k)，第二項代表 upper_bound(k)。這兩項迭代器之間的項就是那些鍵值是 k 的項；也有另一個函式 count(K k)，會回傳鍵值 k 的元素有幾個。set 跟 map 也有這兩個函式，但其實沒什麼用。
   另外就是，erase(K k) 會把鍵值是 k 的全部刪光，所以如果只要刪一個的話必須改成 s.erase(s.find(k))

其實好像可以 unordered_multiset 的樣子，不過筆者不知道這到底要幹嘛就是了。

3.9.3 習題

1. (ZJ d518) map+string。
2. (104 全國賽 pA/TIOJ 1911) 可以用 multiset 輕鬆寫掉這題，不過常數稍微大了一些。

3. (TIOJ 1161) 有 n 種技能，第 i 種技能有權重 s_i 及 a_i，每次你可以花代價 1 來增加你的 s 或 a(初始皆為 0)，你能學到技能 i 若且唯若 $s \geqq s_i$ 且 $a \geqq a_i$，試問你要花多少代價才能學到至少 k 種技能？

4. (TIOJ 1221) 有 n 種菜，k 個鍋子，且有 p 道菜依序得炒，每次炒菜使用的炒菜鍋，沒有被用過，或者上一道菜與這一道菜種類不同，炒菜前就必須洗鍋子，試問最少必須洗幾次鍋子？

3.10 Bitset

3.10.1 簡介

由於 bool 陣列的大量使用和記憶體缺陷 (一個 bool 用到的記憶體其實跟 char 一樣大)，於是他被拿了出來並做了極好的優化。

3.10.2 STL

以下變數假設變數名為 b：

1. 標頭檔：<bitset>
2. 建構式：bitset<N> b(a)：用 a 初始化一個長度為 N(不可為變數) 的 bitset。這裡 a 可以是 unsignedlong、string 或 C 式字串。如果沒有指定 a，或者如果 b 有一些地方沒被 a 初始化，那些地方預設為 0。
3. b.count()：回傳 b 有幾個位元是 1。複雜度 $O(N)$。
4. b(位元運算)：不管是一元還是二元的位元運算都可以。如果是兩個 bitset 的二元位元運算，兩個 bitset 的長度需一致。複雜度 $O(N)$。
5. b[a]：存取第 a 位，複雜度 $O(1)$。
6. b.set()：將所有位元設成 1。複雜度 $O(N)$。
7. b.reset()：將所有位元設成 0。複雜度 $O(N)$。
8. b.flip()：將所有位元的 0、1 互換。複雜度 $O(N)$。
9. b.to_string()：回傳一個字串和 b 的內容一樣。複雜度 $O(N)$。
10. b.to_ulong()：回傳一個 unsigned long 和 b 的內容一樣 (在沒有溢位的範圍內)。複雜度 $O(N)$。
11. b.to_ullong()：回傳一個 unsigned long long 和 b 的內容一樣 (在沒有溢位的範圍內)。複雜度 $O(N)$。

bitset 不是容器，而且它也沒有迭代器。通常而言，如果要估計常數的話，相較於直接使用陣列，空間是 1/8、count 約是 1/6、位元運算約是 1/30。當然這些都不是絕對的。
要注意的是，上述的複雜度沒有明文規定，不過通常是如此。

4 小結

介紹完了許多 STL 和資料結構，還請各位要多加練習才能熟悉這些東西。
如果覺得用法很多很雜也可以掃過一遍就先用用看，看想用什麼再回來查。
然而筆者這邊的資訊也不是齊全的，大多數只是列出一些比較常用的東西而已。
~~(我絕對不會說裡面有很多都是從建中講義抓來的。)~~
總之如果還想了解更多，請各位多多善加利用以下兩個網站：

http://www.cplusplus.com/
https://en.cppreference.com/w/