Data Structure and Programming HW5 Report

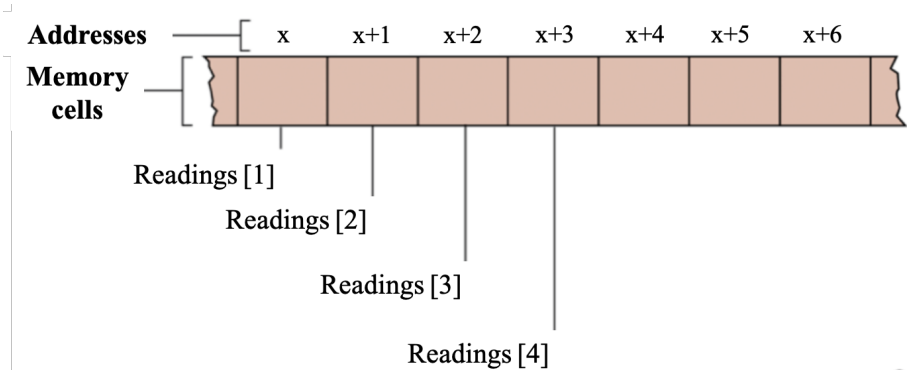
Name: 呂承樺 Student ID: B06901011

一、 資料結構的實做

A. 簡述關於這三種ADT的資料結構與操作上之不同。

1. Array

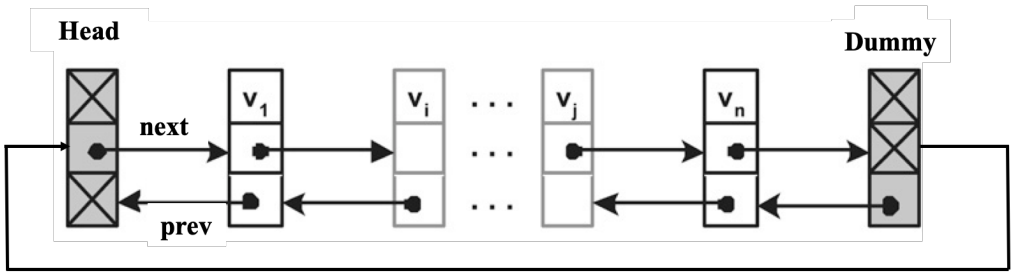
Array的記憶體位置在電腦中是一整段的，Array的每個元素都連續的被排列在其中，所以他們的記憶體位址是連續的（如下圖一），因著這個特性，使得Array在insert和delete的時候可以減少多餘的運算，而有助於加速工作的執行。



圖一

2. Doubly Linked List

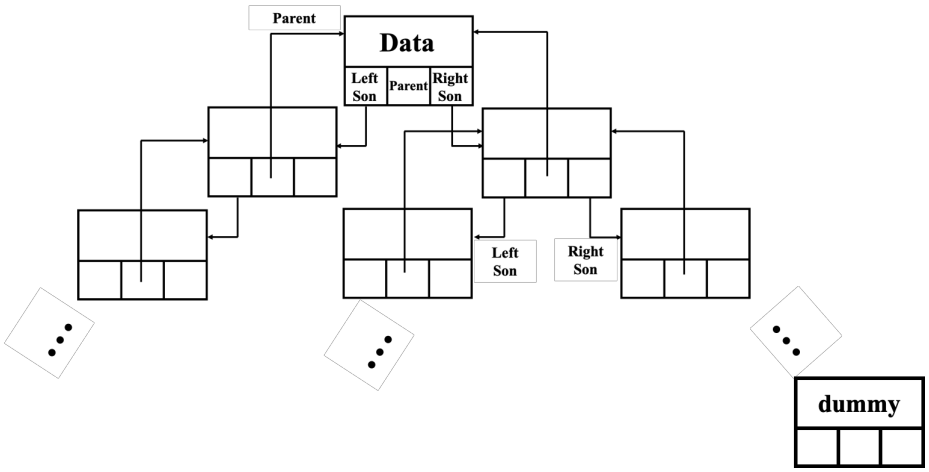
Doubly linked list裡面，每個元素除了儲存data，還多了兩個指標儲存前一個元素以及後一個元素的位址（如圖二），因此在list中任意挑選一個node為起點，都可以很容易地得到它的前一個以及後一個元素的資料；doubly linked list和array最大的差異在於doubly linked list的元素，它們彼此的記憶體位址不見得是連續的，而是透過指標將其串在一起，因此會需要一些額外運算的時間。另外，本次實作中，在最後一個元素後面新增一個dummy node，使得dummy node的下一個元素指向head，使得整個list成為一個圓環式的結構。



圖二

3. Binary Search Tree

一個樹狀的資料結構，其中每個元素除了儲存data之外，還存有三個pointer，分別存取此元素的left son、right son和parent的記憶體位址(如圖三)；此外，每個左子樹所存的元素裡的data都小於root的data，且每個右子樹所存的元素裡的data都大於root的data，因著這樣的特性，在insert新的元素的時候，因為會被視為某一元素的left son或是right son，進而變成樹裡的leaf，所以會非常的快速，但是在delete的時候，則有比較多的狀況，會執行得比較久。



圖三

B. 你如何去使用程式去實現上述的概念？

1. Array

(a) Iterator (見下圖四)

```
const T& operator * () const { return (*_node); }
T& operator * () { return (*_node); }
iterator& operator ++ () { _node++; return *this; }
iterator operator ++ (int) {iterator tmp = *this; ++(*this); return tmp; }
iterator& operator -- () { _node--; return *this; }
iterator operator -- (int) {iterator tmp = *this; --(*this); return tmp; }
iterator operator + (int i) const { iterator tmp = *this;  tmp += i; return tmp; }
iterator& operator += (int i) { _node += i; return *this; }
iterator& operator = (const iterator& i) { _node = i._node; return (*this); }
bool operator != (const iterator& i) const { if(_node == i._node) return false; return true; }
bool operator == (const iterator& i) const { if(_node == i._node) return true; return false; }
```

圖四

「 * 」：直接取_node的指標即可。
「 ++ 」:往後移動一格，因為array記憶體的連續性，所以直接_node++。
「 -- 」:往前移動一格，因為array記憶體的連續性，所以直接_node--。
「 + i 」：直接往後移動i格。
「 = 」：對_node做assign。
「 !=/== 」: 判斷_node是否相同。

(b) Member functions

begin()：回傳array開頭的pointer，也就是_data。

end()：回傳整個array的最後一個元素的下一個，所以是_data+_size。
empty()：判斷_size是否等於零，是回傳true，否則回傳false。
size()：直接回傳_size。
push_back(x)：新增新的node在整個陣列的最後，判斷_capacity是否有足夠空間存入新的_node，若無，則新增一_capacity為原本兩倍的陣列，並把本來的資料copy過去。
pop_front()：移除第一個node，也就是erase(begin())。
pop_back()：移除最後一個node，也就是erase(end())。
erase(pos)：把要刪除的那格用array的最後一個node蓋掉。
erase(x)：先找到要刪除的node，再拿array的最後一個node把它蓋掉。
find(x)：搜尋整個array尋找第一個_data是x的_node。
clear()：把整個array的記憶體位址還給電腦，必將_size設定回0。

2. Doubly Linked List

(a) Iterator（見下圖五）

```
const T& operator * () const { return _node->_data; }
T& operator * () { return _node->_data; }
iterator& operator ++ () { _node = _node -> _next; return *(this); }
iterator operator ++ (int) { iterator tmp = *this; ++(*this); return tmp; }
iterator& operator -- () { _node = _node -> _prev; return *(this); }
iterator operator -- (int) { iterator tmp = *this; --(*this); return tmp; }
iterator& operator = (const iterator& i) { _node = i._node; return *(this); }
bool operator != (const iterator& i) const { if(_node != i._node) return true; return false; }
bool operator == (const iterator& i) const { if(_node == i._node) return true; return false; }
```

圖五

「＊」：直接回傳_node的_data。
「++」：將iterator指向原來的_node的下一個元素。
「--」：將iterator指向原來的_node的前一個元素。
「=」：對_node做assign。
「!=/==」: 判斷_node是否相同。

(b) Member functions

begin()：回傳list開頭的iterator，也就是_head。
end()：回傳dummy node的iterator，也就是_head->_prev。
empty()：如果begin()指向dummy node，表示list為空。
size()：從begin()開始不斷平移iterator，平移到end()的次數即為list大小。
push_back(x)：新增新的node在dummy node和原本的最後一個node之間，並且重新link彼此前後的關係。
pop_front()：移除第一個node，重新link各個node之間的前後關係，並將_head的指標指向新的_head。
pop_back()：若非空list，移除最後一個node，也就是erase(end())。
erase(pos)：若非空list，則將需刪除的node的前一個node與需刪除的node的後一個node重新link起來，並將該node記憶體位址還給電腦。
erase(x)：先找到要刪除的node的iterator，再執行erase(pos)。
find(x)：搜尋整個list尋找第一個_data是x的_node的位址。
clear()：把整個list的記憶體位址還給電腦。
sort()：最基本的比較方法，時間複雜度為$O(n^2)$。

3. Binary Search Tree

(a) Iterator

「＊」：直接回傳_node的_data。
「++」：要把iterator指向原本node的successor，較複雜的原因是node的successor不見得跟原本的node有link的關係，所以必須從_data的大小找起。先判斷如果_node有right son，則尋找以right son為root的子樹的最小值；若無，則必須從_node的parent找起。（見下圖六）

```
iterator& operator ++ () {
    if (_node->RSon != 0){
        _node = _node->RSon;
        while(_node->LSon != 0) _node = _node->LSon;
    }
    else{
        if(_node->parent->LSon == _node) _node = _node->parent;
        else {
            _node = _node->parent;
            while(_node->parent->LSon != _node) _node = _node->parent;
            _node = _node->parent;
        }
    }
    return (*this);
}
```

圖六

「--」: 要把iterator指向原本node的pre-successor，其概念其實跟「++」很像，只是方向相反而已。（見下圖七）

```
iterator& operator -- () {
    if (_node->LSon != 0){
        _node = _node->LSon;
        while(_node->RSon != 0) _node = _node->RSon;
    }
    else{
        if(_node->parent->RSon == _node) _node = _node->parent;
        else{
            _node = _node->parent;
            while(_node->parent->RSon != _node) _node = _node->parent;
            _node = _node->parent;
        }
    }
    return (*this);
}
```

圖七

「=」：對_node做assign。
「!=/==」: 判斷_node是否相同。

(b) Member Functions

begin()：回傳整個BST裡面最小的值的位址。
end()：回傳dummy node的iterator，也就是iterator(_tail)。
empty()：判斷_size是否為零。
size()：直接回傳_size的大小。

pop_front()：移除第一個node，所以直接erase(begin())。
pop_back()：若非空BST，移除最後一個node，並把link重新接上。
erase(pos)：BST的erase非常複雜，因為若要砍掉樹上的任何一個branch，都將牽動到整棵樹的長相與走向，所以我的寫法分很多種case討論，以下一一說明。
在必須刪除的node裡面，可以簡易的區分成3種，node為leaf、node只有一個son以及node有兩個son的狀況去討論。在經過分析後，得到了統一的結論。首先，如果要刪除的node是_tail則回傳false。再來將BST只有root的狀態獨立出來寫，也就是直接把整個BST設回預設。若是BST不只有root的話，我額外令了兩個iterator來追蹤我想要刪除的點的son或是parent，方便更改它們的連接方式，或有些情況下，直接把node裡的data做更動即可。最複雜的點是若是son碰到_tail情況下，link的關係會很麻煩，所以我把這幾種特殊狀況獨立出來寫。
erase(x)：其實就是在找到要刪除的x的iterator後，再執行erase(iterator)即可。
find(x)：搜尋整個BST尋找第一個_data是x的_node的位址。
insert(x)：也是將case分成三種：BST為空、BST只有root、BST不只有root的情況分別討論。在第三種情況，我也額外多令了兩個node用來追蹤我要插入新的node的parent或是son，方便將整個BST串起來。
clear()：把整個BST的記憶體位址還給電腦，但是並不刪除dummy node。

C. 請說明為何你要使用這樣的實做方式，有何優缺點。

1. Array

我覺得我的作法應該是最簡單、最直觀的寫法，可能code裡面有很多不必要獨立出來討論的case，但是為了保險起見，很多地方會獨立出來寫，必免不必要的error，優點應該就是保守、比較容易讀懂；缺點的話應該是程式比較冗長，不夠精簡，可能甚至因此而消耗比較多記憶體及執行時間。

2. Doubly Linked List

寫法很直觀，基本上就只是把Doubly Linked List的定義轉換成程式語言而已，但是在sort的部分（見下圖八），程式看似簡單的幾行，但是時間複雜度卻是不折不扣的$O(n^2)$，只要list裡面的node超過幾萬個，基本上執行sort就需要花上好一段時間。本來有打算改成比較快的方式，但是BST實在消耗太多腦力，所以最後就不打算更改了，所以如果程式執行的有點久，請助教或是教授多一點耐心，它真的只是跑得比較慢而已，並沒有進入無限迴圈。

```
void sort() const {
    for (iterator first = begin(); first != end(); first++){
        for (iterator sec = first._node->_next; sec != end(); sec++){
            if ((*first) > (*sec))  swap(first._node->_data, sec._node->_data);
        }
    }
}
```

圖八

3. Binary Search Tree

寫法相較起來沒有很容易讀懂，因為我有試著把所有BST可能出現的case都畫出來，發現有大概十幾到二十種，如果每種case都分開討論的話，整個程式會非常冗，而且很難debug，所以我把所有的case在紙上做了進一步比較與分析，才寫成大概30行內可以跑完的erase和insert。比較大的缺點是因為保險起見，我在insert和erase的地方都會新令兩個node或是兩個iterator來確保我存取的是我要的地方或node，所以可能會花掉比較多的記憶體空間。

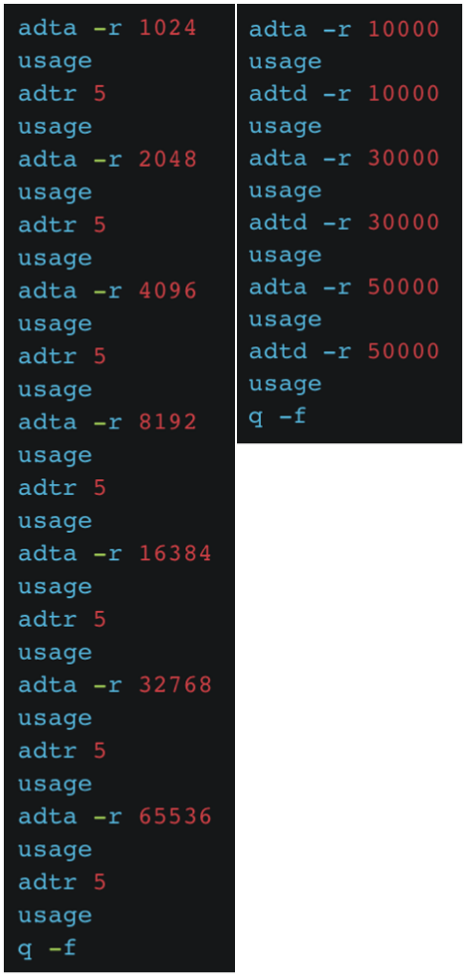
比較特別的地方是我把root的parent令成_tail，如此一來，在整個BST中所有的node都會有parent，有點類似Doubly Linked List的概念，把整個Tree串成一個環狀，方便很多的搜尋。

二、 實驗比較

A. 實驗設計

我寫了三個測試檔案，其中Test1主要測試執行insert (push_back)以及clear的速度（見側圖九左），而Test2主要測試insert和erase的速度（見側圖九右），Test3主要測試比較不好的狀態下，自己的code與reference做比較。

其中Test1數值的選定是因為它們是2^{10}~2^{16}，希望這樣取值可以對應時間複雜度O(log(n))是線性的。Test2的取值則是比較隨意。



圖九

（左：Test1、右：Test2）

B. 實驗預期

1. Test1

Insert要花的時間應該是Array ≅ Dlist > BST，因為insert的時間複雜度分別是O(n)、O(n)、O(log(n))，但是reset的時間應該都會蠻短的，可能到最後幾個得時候才會出現秒數，而且隨著數值的增加，估計Array和Dlist應該會執行很久。

2. Test2

Erase要花費的時間應該是Array > Dlist ≅ BST，因為erase的時間複雜度是分別是O(n)、O(1)、O(1)。

3. Test3

生成aaa到zzz的add list和delete list，比較三種不同的排列方式的執行速度，並與reference做比較。

C. 結果比較與討論

1. Test1

Test1 (unit : sec)	Array	Dlist	BST
adta -r 1024	0	0.01	0
adtr 5	0.01	0	0
adta -r 2048	0	0.05	0
adtr 5	0.02	0	0
adta -r 4096	0	0.19	0
adtr 5	0.12	0	0
adta -r 8192	0	0.78	0.01
adtr 5	0.45	0	0
adta -r 16384	0	3.03	0.01
adtr 5	1.61	0.01	0.01
adta -r 32768	0.01	12.32	0.02
adtr 5	7.03	0.01	0.01
adta -r 65536	0.02	49.03	0.05
adtr 5	21.1	0.01	0.02

Insert的時間大約不如預期，結果呈現Dlist > Array ≅ BST，最後面，Dlist跑超級久的，可能是寫法的問題。至於Reset的成果大約符合預期，當數值偏大的時候Array跑的時間會比較久。

2. Test2

Test2 (unit : sec)	Array	Dlist	BST
adta -r 10000	0	1.17	0
adtd -r 10000	0	0.14	0.65
adta -r 30000	0.01	10.37	0.01
adtd -r 30000	0	1.23	6.69
adta -r 50000	0.02	28.73	0.03
adtd -r 50000	0	3.49	20.82

Erase執行的時間跟預期不太相同，結果呈現BST > Dlist > Array，而insert的時間與test1一致，呈現Dlist > Array ≅ BST，可見Dlist的push_back時間複雜度比較大。而在erase裡面，BST會花較久的原因我認為是在尋找要刪除的node已經花了不少時間，若再加上又搜尋successor和pre-successor又花去大半時間，所以在執行起來會比較久。

3. Test3

Test3 (unit : sec)	Array	Dlist	BST	Array reference	Dlist reference	BST reference
adta -s aaa~zzz	0.58	4.01	2.36	0.57	1.26	2.05
adtd -s zzz~aaa	6.26	6.39	6.69	0.56	0.56	1.97

因為排序方式的不同，insert的時間呈現Dlist > BST > Array，但是相較reference的BST > Dlist > Array，可見很有可能是因為這三種排序中我最先寫Dlist，尚未很熟悉整個概念，所以寫法比較複雜與冗長，導致執行速度比較緩慢。而delete的部分雖然都跟reference有很多倍的差距，但是基本上可以看出BST需要的時間最久。