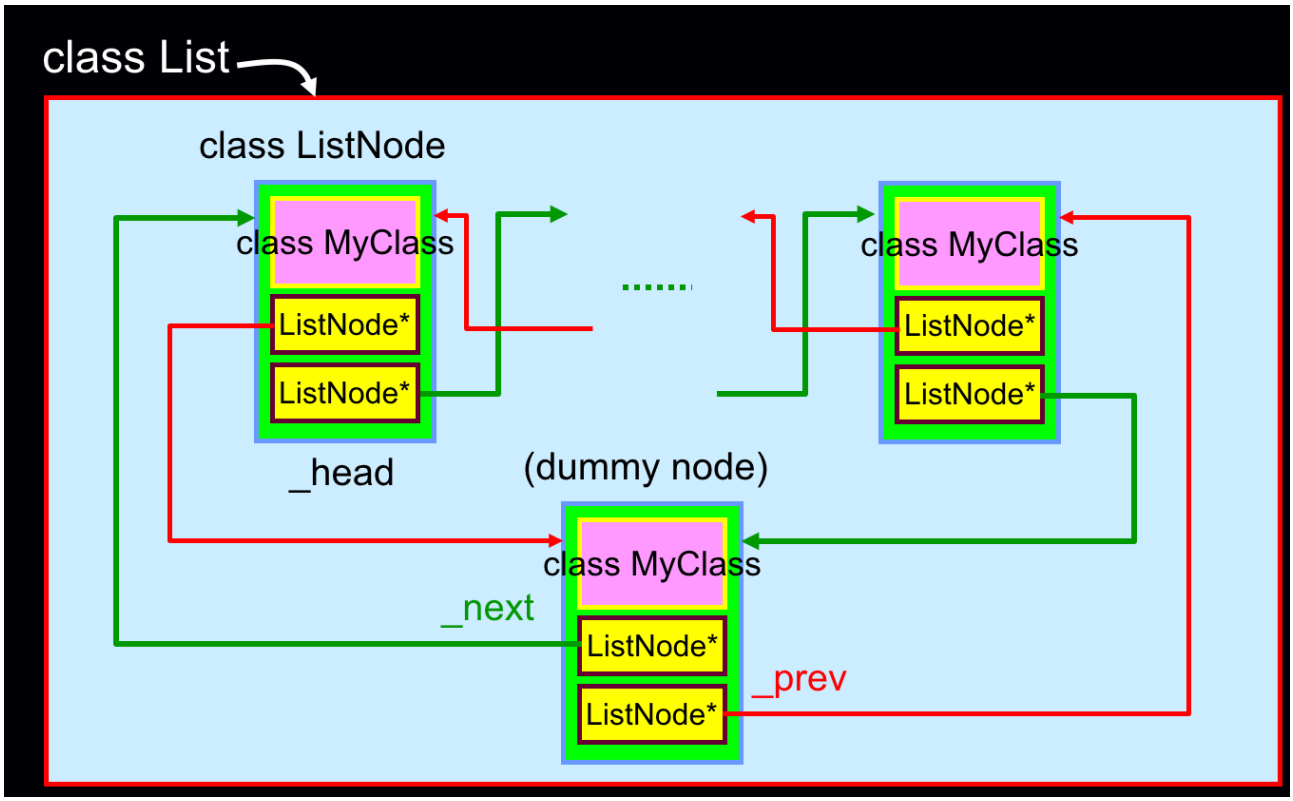
DSnP HW5 Performance Study Report B03901022　卓伯鴻
一、Performance of Data Structure

(1)array
1.組成在array這種資料結構的操作上，因為可以使用原有的陣列資料格式來儲存資料，所以在指令操作上比較方便(可以直接操作arr[i]，中括號裡的數值i來讀取存在個單元中的資料)
2.優點：在search資料這個部分只需要花O(1)的速度)
3.缺點：有一開始開空間給array時的問題(無法確認需要多少空間來儲存)，因次需要做動態調整，而在這邊也需要準確的歸還(delete)之前所佔用的儲存位置，以免造成memory leak。

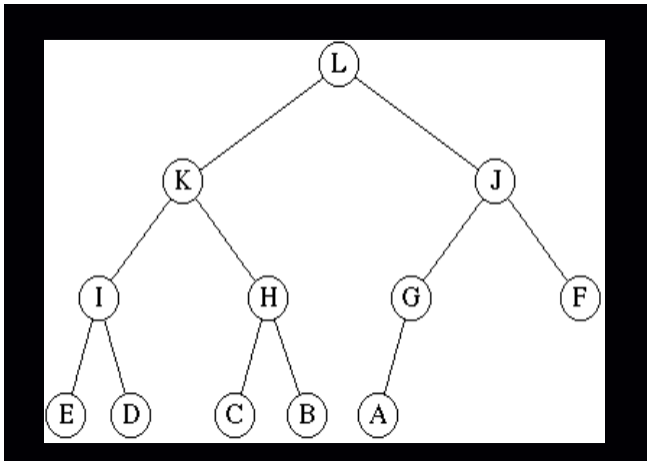
(2)Double-linked List(DList)



1.組成：DList由很多個DListNode構成，每一個DListNode裡面都有存兩個pointer一個是他前一個node的位置叫做_prev(紅色系列)，另外一個是存放下一個node的位置叫做_next(綠色系列)，而每個Node裡面也都有存放自己的資料。而在DList這個資料結構裡有一個dummy node，這個node會連到第一個Node(dummy node的_next是第一個node)，他的_prev則是最後一個_node，因此在跑iterator(一個node一個node查詢的時候)，他的終止條件可以填在dummy node上，也就是他最後一個node的下一個。
2.優點：建構好DList與DListNode，在新增資料上相當的方便(只需要新增一個node再更改兩個node的連結)。在做sort的時候，如果每個Node裡面存的資料量很大，可以透過更改連結(_prev&_next)的方式，來變換這串DList的順序。
3.缺點：在搜尋單筆資料的時間上需要O(n)的時間。

(3)Binary Search Tree(BST)
1.組成：BST由很多個BSTNode組成，如同DListNode一樣，但是每一個BSTNode中有三個儲存其他位置的pointer：其一是_right，這個pointer是儲存一個在他右邊(比他大)的Node，另一是_left，這個pointer是儲存一個在他左邊(比他小)的Node，最後一個則是_parent，是儲存一個上層的Node(下

左圖中位於他上方的Node)。而我把一開始建構這個BST的dummy node放在整個tree的最右下方的_right的地方，也就是存有最大儲存值的node的_right。

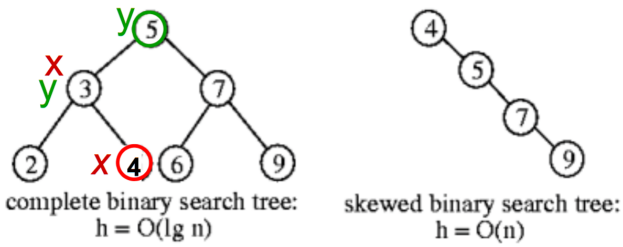


successor：是恰好比他大的下一個值，所以找的方法是如果他的_right不是空的就找他_right的subtree中最小的元素，如果他的_right是空的話，就找他最小的ancestor中的node，而這個node的_left也剛好是這個node。這個值也會是iterator中的下一個(iterator++)
precessor：是恰好比他小的上一個值，找的方法剛好與successor相反(把_right改成_left)，也就是在iterator中找到的上一個值(iterator—)

```
Tree-Successor(x)
1. if x.right ≠ NIL
2.     return Tree-Minimum(x.right)
3. y = x.p
4. while y ≠ NIL and x == y.right
5.     x = y
6.     y = y.p
7. return y
```

y is the lowest ancestor of *x*, whose left child is also an ancestor of *x*

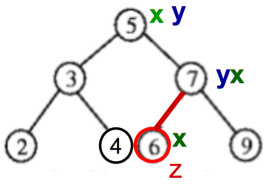
```
Tree-Minimum(x)
1. while x.left ≠ NIL
2.     x = x.left
3. return x
```



(左圖為pseudo code 右圖為圖例)

insert：新增一個新的Node，把它推入正確的位置(從整個tree最上方的node開始找，比他大就往右走，比他小就往左走，走到有空位為止)，下圖左為pseudo code和示意圖。

```
Tree-Insert(T, z)
1.  y = NIL
2.  x = T.root
3.  while x ≠ NIL
4.        y = x
5.        if z.key < x.key
6.              x = x.left
7.        else x = x.right
8.  z.p = y
9.  if y == NIL
10.       T.root = z  // T is empty
11. elseif z.key < y.key
12.             y.left = z
13. else y.right = z
```



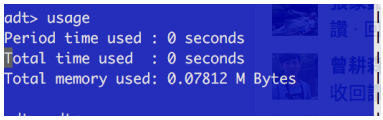

```
Transplant(T, u, v)
1.  if u.p == NIL
2.      T.root = v
3.  elseif u == u.p.left
4.      u.p.left = v
5.  else u.p.right = v
6.  if v ≠ NIL
7.      v.p = u.p
```

erase：刪去一個node。首先先考慮刪去一個Node要如何補上Node的空缺，於是希望把比他小的那個node接上去，或是下面有空位的話就找他的successor接上去，而無論何種狀況都需要一個把NodeU接到NodeV上的方法，也就是上圖右的pseudo code在做的事。而在delete的過程中總共有四種case，把這四種case分開處理的話，寫成pseudo code就是下圖了。

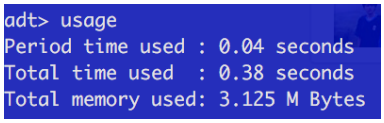
```
Tree-Delete(T, z)
1.  if z.left == NIL   // case A
2.      Transplant(T, z, z.right)
3.  elseif z.right == NIL   // case B
4.      Transplant(T, z, z.left)
5.  else y = Tree-Minimum(z.right)
6.          if y.p ≠ z   // case D
7.              Transplant(T, y, y.right)
8.              y.right = z.right
9.              y.right.p = y
10.         Transplant(T, z, y) // case C
11.         y.left = z.left
12.         y.left.p = y
```

2.優點：建立好BST之後就可以利用O(logn)的時間把想找的資料找到，而且每次推入新的資料都不需要重新排序(都是排序好的)。
3.缺點：在丟入新的資料跟刪掉一個資料的時候，會動到相當多的連結，操作起來略顯麻煩，而且也要考慮這些動作是否會干擾到dummy node的連結。

二、Performance Comparing
(1)實驗：吾人把作業檔案中的do1,do2,do3,do4分開來對三個資料結構做測試，並比較它們的性能。
(2)結果：
1.array:

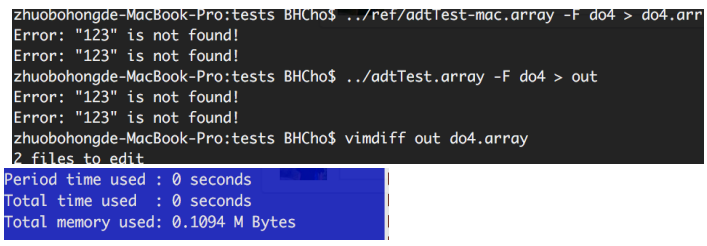


do1



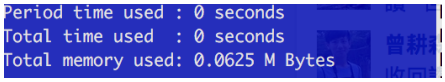
do2

do3

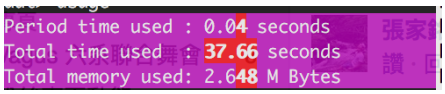


do4

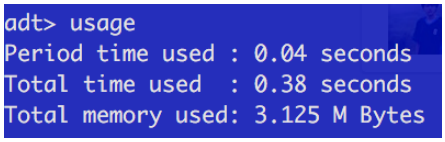
2.DList



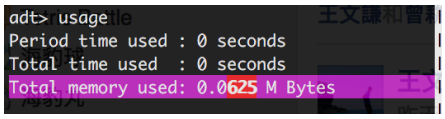
do1



do2

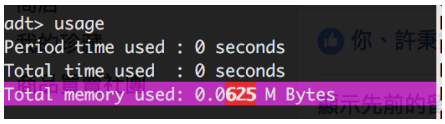


do3



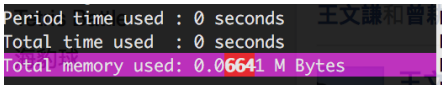
do4

3.BST

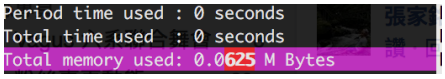


do1

do2



do3



do4

(3)討論與比較：

在每個資料量比較小的狀況下，使用array作為儲存的模式將會是一個好選擇。