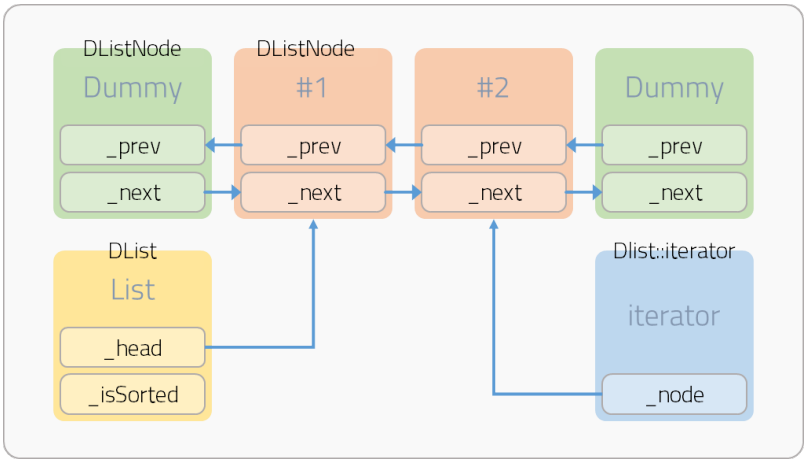
# HW5

姓名: 王國豪　　學號: b07901032

1. Implementation

A. Doubly Linked List




i. Structure

DList 的原理是讓每一個 node 都以 prev 和 next 來存取前後的 node。這裡實作上不儲存 end，而是使用了一個 dummy node 來當作 end 並接續第一個 node 形成一個環。在 size=0 時 head 會指向這個 dummy；當有資料的時候，head 會指向第一個 node 也就是 dummy 的 next。

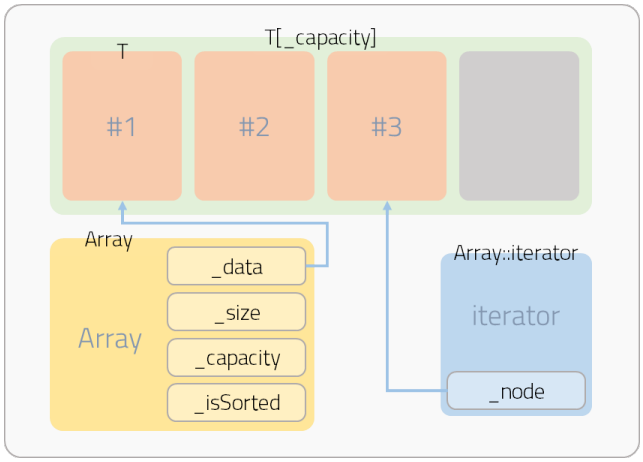
ii. iterator find();

用 iterator 跑過 begin 到 end 來比對。

iii. void sort() const;

在 dlist 中的 sort，我選擇用 selection sort 與 std::swap 搭配 iterator 實作。Selection sort 所需額外空間是 O(1)，比較固定 n(n-1)/2 次，而交換的次數最壞是 n-1 次，是在逆序的情況時發生。

B. Array(Vector)




i. Structure

實作上是先向系統要一塊大小為 capacity*sizeof(T)的記憶體，當 append 的時候放進這一塊記憶體中。如果 capacity 不夠用了，就要一塊兩倍大的，將東西搬(copy)到新的位置，再 free 掉原本佔有的記憶體。

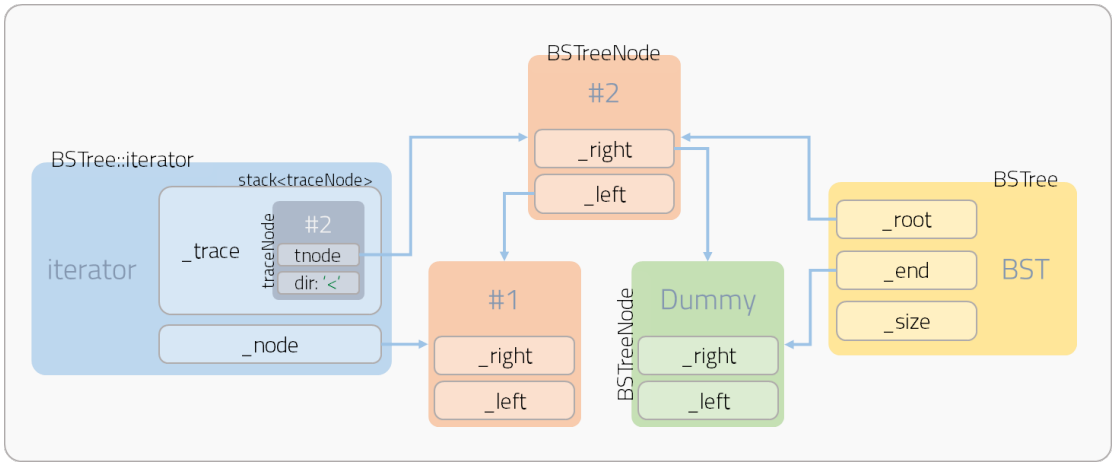
ii. copy vs move

c++11 裡有個功能叫做 move，它可以將一個東西變成 rvalue reference(簡單解釋: 就像是在 a = 2+3，5 就是那個 rvalue，在這個 copy 給 a 之後就消失了)。若我們的 testObj 是 move_constructible 與 move_assignable 的話，那他就可以透過 a = move(new T())來將在 new T()位置的內容物直接 assign 給 a，在這個操作過後 new T()的位置上就不會有東西。因為量大，若是在 expand 時若使用 move 的話，可以省下不少記憶體與效能。

iii. iterator find();

若沒有 sort 過，就用 iterator 跑過 begin 到 end 一一比對。
若有 sort 過，就先以 exponential search 跑過，找到一個小一點的區間，再以 binary search 跑過這個區間來搜尋。

C. Binary Search Tree




i. Structure

樹本體的實作上開了一個 dummy 當作 end，如此一來在 end 的判斷上快速許多，缺點是在 insert 與 erase 時都要確保 end 不會改變或者離開這棵樹，原本只有 4 個 case 的 delete 就因為 dummy 的關係多了兩個 case。

iterator 的實作上開了一個 stack 儲存從 root 跑到 node 的路徑與方向，如此一來每個 node 就不用儲存 parent。不過在++或--時要控制 trace 的內容，在 begin、end 或 find 時要將 trace 建立完整再回傳這個 iterator。相對若是每個 node 儲存 parent，iterator 只要儲存指向該 node 的 pointer 就好。

ii. `void bsfindcandy(BSTreeNode<T>*& node, const T& x, iterator& it);`

這是同時給 find 與 insert 使用的 recursive search 函式，主要目的是對於一個新增的 data 找到 insert 的位置的 iterator。

iii. void print() const;


這是 verbose print 的長相，可以 define PF 與 LEAF 來調整位移量與葉子的顯示與否。~~~><(((°>°是一隻 dummy fish。

2. Experiment

(A) add/delete<front|back|random>/sort 5000 times

| REF | Dlist | | Array | | BST | |
|---|---|---|---|---|---|---|
| | time | memory | time | memory | time | memory |
| adta –r 5000 | 0 | 0.5547 | 0.0025 | 0.7617 | 0.005 | 0.5586 |
| adtd –f 5000 | 0 | 0.5586 | 0 | 0.7617 | 0 | 0.5703 |
| adtd –b 5000 | 0 | 0.5586 | 0 | 0.7617 | 0 | 0.5703 |
| adtd –r 5000 | 0.03 | 0.5586 | 0 | 0.7617 | 0.09 | 0.582 |
| adts | 0.25 | 0.5586 | 0 | 0.7617 | 0 | 0.5625 |

| Self | Dlist | | Array | | BST | |
|---|---|---|---|---|---|---|
| | time | memory | time | memory | time | memory |
| adta –r 5000 | 0 | 0.5508 | 0 | 0.7578 | 0 | 0.5703 |
| adtd –f 5000 | 0.06 | 0.5586 | 0 | 0.7578 | 0 | 0.582 |
| adtd –b 5000 | 0.06 | 0.5586 | 0 | 0.7578 | 0 | 0.5859 |
| adtd –r 5000 | 0.01 | 0.5586 | 0 | 0.7578 | 0.1 | 0.582 |
| adts | 0.09 | 0.5547 | 0 | 0.7578 | 0 | 0.5703 |

(B) add/delete<front|back|random>/sort 50000 times

| REF | Dlist | | Array | | BST | |
|---|---|---|---|---|---|---|
| | time | memory | time | memory | time | memory |
| adta –r 50000 | 0.005 | 3.305 | 0.0075 | 3.395 | 0.02 | 3.309 |
| adtd –f 50000 | 0 | 3.305 | 0 | 3.395 | 0 | 3.316 |
| adtd –b 50000 | 0 | 3.305 | 0.01 | 3.395 | 0 | 3.316 |
| adtd –r 50000 | 3.18 | 3.305 | 0.02 | 3.395 | 13.76 | 3.332 |
| adts | 20.23 | 3.305 | 0.02 | 3.395 | 0 | 3.309 |

| Self | Dlist | | Array | | BST | |
|---|---|---|---|---|---|---|
| | time | memory | time | memory | time | memory |
| adta –r 50000 | 0.0025 | 3.301 | 0.005 | 3.391 | 0.0425 | 3.316 |
| adtd –f 50000 | 5.51 | 3.305 | 0 | 3.391 | 0.03 | 3.328 |
| adtd –b 50000 | 5.54 | 3.305 | 0 | 3.391 | 0.03 | 3.332 |
| adtd –r 50000 | 4 | 3.305 | 0 | 3.391 | 21.47 | 3.328 |
| adts | 10.67 | 3.301 | 0.01 | 3.391 | 0 | 3.316 |

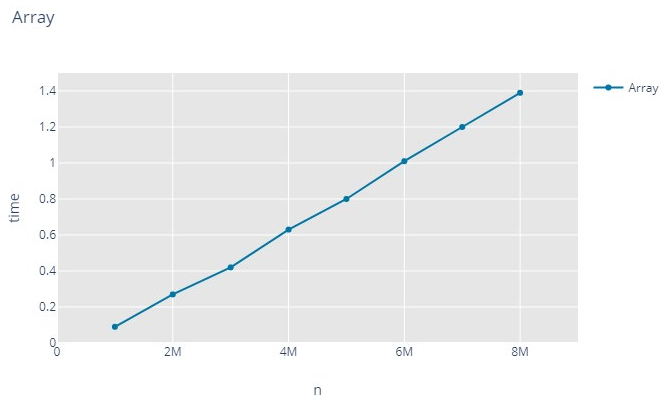
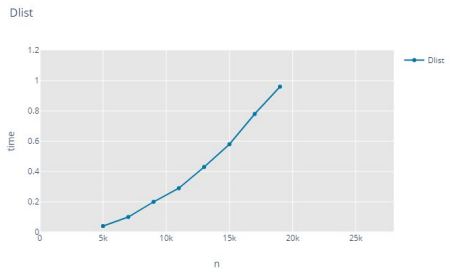
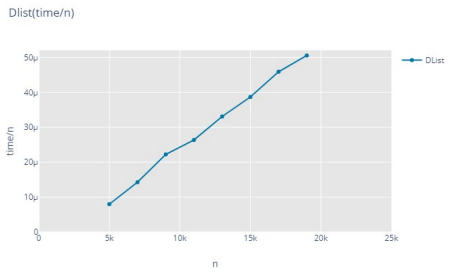
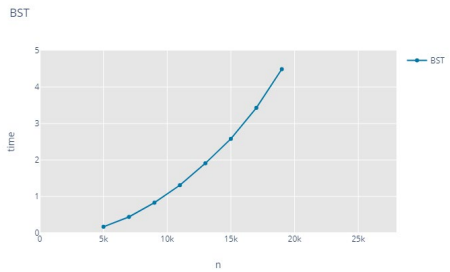
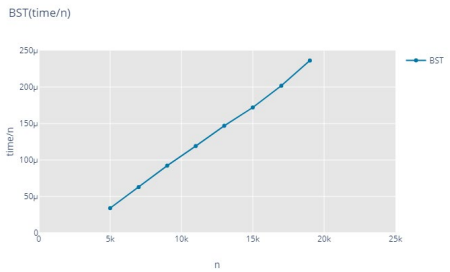
- Ref vs Self 分析(上表):

  首先我們可以看到一個顯著的差異在 delete 的效能，我的效能皆比 ref code 差上許多，我想是因為在 erase 中的條件判斷叫了太多 function 的關係。

```
 1|bool erase(iterator pos) {
 2|    if(empty() || pos == end()) return false;
 3|    else if(pos == begin()) pop_front();
 4|    else if(pos == --end()) pop_back();
 5|    else{
 6|        pos._node->_prev->_next = pos._node->_next;
 7|        pos._node->_next->_prev = pos._node->_prev;
 8|        delete pos._node;
 9|    }
10|    return true;
11|}
```

  在 sort 上面，我想是因為選用 selection sort 的關係，而 ref 使用 bubble，會有許多的交換次數，因此有顯著的效能提升。

- Delete vs n 分析

| Array |
| --- |
|  |
| Dlist |
|  |
| Dlist(time/n) |
|  |
| BST |
|  |
| BST(time/n) |
|  |

| Analysis | Array | Dlist | BST |
| --- | --- | --- | --- |
| Time Complexity | O(1) | O(n) | O(n) |

因為在 dlist 與 bst 的 delete 中，getPos 是以 iterator++的方式進行，因此 time 先除去 n 來做分析，推導出上表的結論。