ADT Performance Study Report

B04705036 電機三 許天鈞

一、 Implementation

1. Doubly linked-list (Dlist)、Array、BST 三種 ADT 資料結構與操作上的不同

Dlist 的基本單位為 list node，每個 node 裡除了有主要儲存的資料(_data)外，也有兩個 pointers (_prev, _next)分別指向該 node 的前後鄰居，相互串接成為一種可雙向 access 的列表結構，而 dummy node(_head)的存在則可以告訴我們某 traverse 方向的 list 已到達尾端，traverse 便可以終止。因為他有_prev 和_next 兩指標來存取 neighboring nodes，Dlist 的操作複雜度幾乎都是 O(1) (push_front/back, pop_front/back, empty, insert)，只有 find 和 size 是 O(n)。

Array 是指 dynamic array，它生成時會視目前_capacity，一次性要一塊連續性的記憶體，然後再一格一格分配來存_data，array 大小(_size)會根據目前有多少_data 在 array 裡來 update，當 array 滿了(_size = capacity)，原本的 array 會被 delete，capacity 會 double，然後再重新要一塊容量為新_capacity，更大的 array 來存資料，因為 array 不斷地被 new 和 delete，所以是動態的概念，同時也比 static array 有效率。因為 array 可以透過(pointer+pos)來 iterate through 全部_data，可被 random access，如果不考慮 order 的話，幾乎全部的操作複雜度都可以降低到 O(1) (push_front/back, pop_front/back, size, empty, insert, erase)，只有 find 仍至少需要 O(logn)，所以整理來說表現應會比 Dlist 好。

BST 是一個 decision tree，它會在每個 tree node 比較資料大小，如果目前資料大於等於該 node 資料，則往左繼續走，若否，則往右，所以 BST 的基本單位為 tree node，node 裡則有要儲存的資料(_data)和三個指標，分別指向_parent、_left、 _node 來記錄它的來源和左右串接元素。由上述可知，一個串接好的 BST 會是 sort 好的，但相關操作如 find、insert、delete 的複雜度則會等於 O(height(T))，與 BST 的 height 有關。

2. 程式實做
   - Dlist

     主要有三個 Class: Class DListNode、Class DList 和 DList 裡的 Class iterator 來協助 traverse 或存取節點資料的操作(如++、--、!=、==、=、*)。

     DListNode 裡只有三個 data member，分別是用來儲存資料的 _data 和分別指向前後節點的指標：_prev 和_next。

     DList 裡則有一 data member，是用來當作 Dlist 結尾的 dummy node: _head，和多個 member function，為主要的資料操作實踐，以下將概要說明各函式。

     bool empty()

     {如果 Dlist 裡只有 dummy node(_head->_prev == head->next)，則 return true} -> **O(1)**

     size_t size()

     {iterate though 全部 node，用一個計數器算 size 大小} -> O(n)

     void push_back(const T&)

     {new 一個新的 DListNode，然後透過 pointer 操作將其接到 DList 尾端} -> **O(1)**

     void pop_front/back()

     {透過 pointer 操作將新的結構接好，再將 DList 內第一個/最後一個 DListNode delete 掉} -> **O(1)**

     bool erase(iterator / T&)

     {先在 DList 內找要刪除的 DListNode，如果找不到則 return false，再操作 pointer 將新的結構接好，然後再 delete 掉該 DListNode，return true} -> **O(n)**

     void clear()

     {將 DList 裡全部的 DListNode 一個一個 pop 掉} -> **O(n)**

     void sort()

{用雙層 for 來 iterate through 全部 DListNode 並進行前後比較，如果後比前小則 swap，來達成 DList ascending 的排序} -> **O(n^2)**

- Array

  包含兩個 Class: Class Array 和協助 traverse 或存取_data[i]等操作(如++、--、!=、==、=、*)的 Class iterator (在 Class Array 裡)。

  Array 裡有三個 data member，分別為用來指向資料存放位置的 pointer _data、紀錄 Array 大小的_size 和表示 Array 容量的 _capacity，以及多個 member function 進行主要的資料操作實踐，和一個自定義的 helper function，以下將概要說明各函式。

  bool empty()

  {如果 Array 裡元素為零(_data = 0)，則 return true} -> **O(1)**

  (const) operator [] (size_t i) (const)

  {overload 中括號來讀取_data[i]的資料內容 } -> **O(1)**

  size_t size()

  {return 目前的_size 值 } -> **O(1)**

  void push_back(const T&)

  {- 若 Array 未滿 (_size < _capacity)，則指定_data[_size]為欲加入元素

  - 若 Array 已滿(_size = _capacity)，則呼叫 helper function **expand()**: 將_capacity*=2，回 local function 後 delete 原有 Array，再 new 一新尺寸增大的 Array，最後才進行_data[_size]的插入 } -> **O(1)**

  void pop_front()

  {將首位元素以最後一位元素取代(_data[0] = _data[size-1]*)* ，然後 _size-1 更新 Array 大小} -> **O(1)**

  void pop_back()

  {_size-1 更新 Array 大小} -> **O(1)**

bool erase(iterator / T&)
{先在 Array 內找要刪除的元素位置，如果找不到則 return false，找到後直接以末位元素取代(_data[index] = _data[size-1])，_size-1 更新 Array 大小，return true} -> **O(n)**

void clear()
{將 Array 裡全部的元素 一個一個 pop 掉} -> **O(n)**

void sort()
{呼叫 stl 內建的 sort function } -> **O(nlogn)**

- BST
- 主要有三個 Class: Class BSTreeNode、Class BSTtree 和 BSTree 裡的 Class iterator 來協助 traverse 或存取節點資料的操作(如++、--、!=、==、=、*)，此外，另有 **imax()**和 **imin()**兩 helper function 來判斷 BSTree 的最大值和最小值。
- BSTreeNode 裡有五個 data member，分別是用來儲存資料的 _data、分別指向左、右和來源節點的三指標：_left、_right、_parent，以及判斷是否 traverse 過該點的布林值 visited (default = false)。
- BSTree 裡則僅有兩 data member，分別是_root 和紀錄 BSTree 大小的_size，另外有多個 member function，為主要的資料操作實踐，和四輔助的 helper function，以下將概要說明各函式。

bool empty()
{如果 BStree 裡元素為零(_root = 0)，則 return true} -> **O(1)**

size_t size()
{return 目前的_size 值 } -> **O(1)**

void insert(const T& x)
{- 若 BSTree 為空 (_root = 0)，則 new 一個 BSTreeNode(x) 並 assign 給_root
- 若 BSTree 不為空，則進行 x 與各節點的比較，若 x<= node-

>_data，則繼續往左走，若 x > node->_data，則往右走，直到走到 BSTree 底端，則在該位置 new 一個 BSTreeNode(x)，然後把它跟 parent 接好，_size+1 更新 BSTree 大小 } -> **O(height(T))**

void pop_front/back()
{先判斷 0 child 或 1 child，再進行相應 pointer 變化，最後再 delete 掉該 node，_size-1 更新 BSTree 大小} -> **O(1)**

bool erase(iterator / T&)
{先在 BSTree 內找要刪除的 node 位置，如果找不到則 return false，找到後呼叫 del(node)進行刪除，return true} -> **O(height(T))**

void clear()
{將 BSTree 裡全部的元素 一個一個 pop 掉} -> **O(n)**

Helper functions:
BSTreeNode max(BSTreeNode*)*
{return 從該 node 開始往右的最大值} -> O(height(T))

BSTreeNode min(BSTreeNode*)*
{return 從該 node 開始往左的最小值} -> O(height(T))

BSTreeNode max(BSTreeNode*)*
{return 該 node 右 subtree 的最小值 } -> O(height(T))

void del(BSTreeNode)*
{判斷該 node 是 0 child、1child 還是 2children，再進行相對應的 pointer 操作，把該 node isolate 出來然後 delete，_size-1 更新 BSTree 大小 } -> O(1)

二、 Comparison

1. 實驗設計
   - **Task 1: insert n data (1 by 1)**
   - **Task 2: insert n data (1 by 1) and remove all**
   - **Task 3: alternatively insertion and deletion**

- **Task 4: sort the data**

2. 實驗預期
   - **Task 1:**
     dlist (O(1)) ~= array (O(1)) < bst (O(height(T)))
   - **Task 2:**
     dlist (O(1)+O(n)) ~= array (O(1)+O(n)) < bst(O(height(T))+O(n))
   - **Task 3:**
     Case erase:
     bst (O(height(T))) < dlist (O(1)+O(n)) ~= array (O(1)+O(n))
     Case pop_front/pop_back:
     dlist (O(1)) ~= array (O(1)) < bst(O(height(T)))
   - **Task 4:**
     bst (O(1)) < array (O(nlogn)) < dlist(O(n^2))

3. 結果比較與討論
   - **Task 1:**
     (small case): adta –r 30

	dlist	**array**	**bst**
Time (s)	0	0	0
Memory useage (M bytes)	0.1406	0.1484	0.1406

     (greater case): adta –r 100000

	dlist	**array**	**bst**
Time (s)	0.01	0.03	0.1
Memory useage (M bytes)	4.793	8.18	6.973

     Small case 因為新增資料量少所以三種 adt 新增效率差不多，但 greater case 就可看出三者的效率差別，所耗時間跟預估的差不多 dlist ~= array < bst，memory usage 則是 array 使用量最大。

   - **Task 2:**
     (small case): adta –r 30 + adtd -a

	dlist	array	bst
Time (s)	0	0	0
Memory useage (M bytes)	0.1289	0.1562	0.1523

(greater case): adta –r 100000 + adtd -a

	dlist	array	bst
Time (s)	0.02	0.03	0.11
Memory useage (M bytes)	4.797	8.152	6.348

Small case 因為處理資料量少所以三種 adt 新增再 clear() 效率差不多，但 greater case 就可看出三者的效率差別，所耗時間跟預估的差不多 dlist ~= array < bst，memory usage 則是 array 使用量最大。

- **Task 3:**

Case erase: (adta –r 1000 + adtd –r 500)*6

	dlist	array	bst
Time	0.02	0	0.24
Memory useage	0.3516	0.4414	0.6836

所耗時間跟預估的有點不一樣 array < dlist < bst，memory usage 則是 bst 使用量最大，如果再把 case 處理數量增大三者的效率差距會更大(bst 和 dlist 都會變超級慢)，array 在 earse 上的表現是效率最好的。

Case pop_front/back: (adta –r 10000 + adtd –f 5000)*6

	dlist	array	bst
Time	0.01	0.02	0.06
Memory useage	2.52	4.191	2.5

所耗時間跟預估的差不多 dlist ~= array < bst，memory usage 則是 array 使用量最大。

- **Task 4:** adta –r 10000 + adts

-	**dlist**	**array**	**bst**
Time	3.03	0.01	0
Memory useage	0.6172	1.172	0.7773

所耗時間跟預估的差不多 bst < array < dlist，memory usage 則是 array 使用量最大，如果再把 case 處理數量增大三者的效率差距會更大(dlist 會很慢)，bst 因為生成 tree 的時候就已經 sort 好了，所以是效率最好的。