Home work #3 B10815057 廖聖郝

1. String Quick Sort

```
quick_sort(sequence.begin(), sequence.end());//quick sort
```

丟入 iterator 的 begin、end，然後排序

```
template<typename iterator,typename compare = std::less<typename
std::iterator_traits<iterator>::value_type> >
//compare是比較大小的funciton object，若沒有傳入就用預設的
void quick_sort(iterator begin, iterator end,compare comp = compare()) {
	//因為iterator的end是指到最後一個element的下個位置，所以要先把end指回最後一個
element，才不會存取違規
	if (distance(begin, end) <= 1) return;//防呆，如果begin end寫顛倒，就不做
	_quick_sort(begin, end - 1, comp);//把end指回最後一個element，呼叫真正的quick sort
}
```

Quick sort 主演算法:把 piviot 放到正確位置後，左右 2 半各自遞迴 quick sort

```
template<typename iterator, typename compare = std::less<typename
std::iterator_traits<iterator>::value_type> >
//compare是比較大小的funciton object，若沒有傳入就用預設的
void _quick_sort(iterator begin, iterator end, compare comp = compare()) {
//quick sort主演算法
	iterator loc;//piviot
	partition(begin, end, loc, comp);
	//把piviot(begin)，放到對的位置(左邊都小於，右邊都大於)，並更新piviot位置到loc
	if (begin != loc) _quick_sort(begin, loc - 1, comp);
	//遞迴呼叫，但為了避免iterator(loc - 1)指到begin之前，要做防呆
	if (end != loc) _quick_sort(loc + 1, end, comp);
	//遞迴呼叫，但為了避免iterator(loc + 1)指到end之後，要做防呆
}
```

Partition:把 piviot 放到正確位置(左邊都小於，右邊都大於)

```
template<typename iterator, typename compare = std::less<typename
std::iterator_traits<iterator>::value_type> >
//compare是比較大小的funciton object，若沒有傳入就用預設的
void partition(iterator begin, iterator end, iterator& loc, compare comp = compare())
{//把piviot放到對的位置(左邊都小於，右邊都大於)
     iterator left = begin, right = end;//起始點、終點
     loc = begin;//piviot設成起始點
     while (true) {
          while (comp(*loc,*right) && (loc != right)) {
          //從終點(right)往左比較，直到遇到比自己小的，就停下
               --right;//往左走
          }
          if (loc == right) {//如果自己是最小的，代表piviot的位置已經是正確的，結束函式
               return;
          }
          else {
          //如果不是最小的，就把piviot的位置換到新的位置，但還不是正確位置，繼續做
               swap(*loc, *right);//交換
               loc = right;//piviot指到新的位置
          }
          while (!comp(*loc, *left) && (loc != left)) {
          //從起點(left)往右比較，直到遇到比自己大的，就停下
               ++left;//往右走
          }
          if (loc == left) {//如果自己是最大的，代表piviot的位置已經是正確的，結束函式
               return;
          }
          else{//如果不是最大的，就把piviot的位置換到新的位置，但還不是正確位置，繼續做
               swap(*loc, *left);//交換
               loc = left;//piviot指到新的位置
          }
     }
}
```

2. Priority Queue using Binary Heap

節點 class 與類型 enum

```
template<typename element,typename order>
//element是heap內的資料型別(int)，order是優先權的資料型別(char)
class heap_node {//heap node的class
public:
	heap_node() {}
	heap_node(element i, order p, heap_node<element, order>* f) : data(i), priority(p),father(f) {}
	template<typename, typename> friend class heap;
private:
	element data;//資料
	order priority;//優先權
	heap_node<element, order>* father = nullptr;//父節點指標
	heap_node<element, order>* left = nullptr;//右邊小孩指標
	heap_node<element, order>* right = nullptr;//左邊小孩指標
};
enum class heap_type : bool {//heap的類型
	MIN,
	MAX
};
```

Heap class，insert funciton 提供一個介面，真正的實作放在另一個 insert 裡

```
template<typename element, typename order>
//element是heap內的資料型別(int)，order是優先權的資料型別(char)
class heap {//heap的class
public:
	heap(){}
	heap(heap_type mm):min_max(mm){}
	void insert(element in,order pr) {//插入新資料
		insert(&root, root, in, pr, 0);//呼叫真正的insert function
		++number_of_node;//節點數+1
	}
```

真正實作的 insert funciton，使用二元樹的 index 確認是否為目標節點(complete binary tree 最後一節點)，往下遞迴直到找到目標節點指標，創建新節點後往上遞迴比較交換到正確位置，以符合 heap 的規則

```
    void insert(heap_node<element, order>** now, heap_node<element, order>* father,
element in, order pr,int index) {
//now是現在節點的指標的指標，配置記憶體時才可以正確連接到，index用於確認是否為目標節點
        if (index == number_of_node) {
        //若現在節點的index與節點數相等，代表目前正位於新節點的位置
            *now = new heap_node<element, order>(in, pr, father);//配置新節點
            rise(*now);
            //新node的值可能並不符合heap的規則，所以要把該值往上遞迴交換到正確位置
            return;
        }
        if (*now != nullptr) {
        //不是leaf node的小孩指標且不為目標節點，就遞迴下去
            insert(&(*now)->left, *now, in, pr, index * 2 + 1);
            //左小孩的index為自己*2+1
            insert(&(*now)->right, *now, in, pr, index * 2 + 2);
            //右小孩的index為自己*2+2
        }
    }
```

Heap 的刪除都是 root node，但並不會釋放 root node 的記憶體，而是將最後一個 node 的值替代到 root node，並刪除最後一個 node，替代後還要往下遞迴交換到正確位置，以符合 heap 的規則

```
    void erase() {//刪除資料(root node)
        heap_node<element, order>* last = find_last(root,0);
        //找到complete binary tree最後一個node
        if (last == root) {//如果是root node，刪除就好
            delete root;
            root = nullptr;
            return;
        }
        root->priority = last->priority;
        //把root node的資料替代為complete binary tree最後一個node的資料
        root->data = last->data;
        //把root node的優先權替代為complete binary tree最後一個node的優先權
        if (last->father->left == last) {//如果最後一個node是父節點的左小孩
            last->father->left = nullptr;//父節點的左小孩設為空
        }
        else {//如果最後一個node是父節點的右小孩
            last->father->right = nullptr;//父節點的右小孩設為空
        }
        delete last;//刪除最後一個node
        fall(root);
        //新root node的值可能並不符合heap的規則，所以要把該值往下遞迴交換到正確位置
    }
```

find_last 回傳最後一個 node(complete binary tree 的最後)

```
heap_node<element, order>* find_last(heap_node<element, order>* now,int index) {
//找到complete binary tree最後一個node
	if (now == nullptr) return  nullptr;
	//走到leaf node的小孩指標就回傳null，表示這一分支遞迴沒有解
	if (index == number_of_node - 1) {
	//若該節點index與節點數-1相等，代表該節點就是complete binary tree最後一個node
		return now;//回傳
	}
	heap_node<element, order>* L_result = find_last(now->left, index * 2 + 1);
	//往左遞迴，左小孩的index為自己*2+1，結果暫存到L_result
	heap_node<element, order>* R_result = find_last(now->right, index * 2 + 2);
	//往右遞迴，右小孩的index為自己*2+2，結果暫存到R_result
	if (L_result) return L_result;//若左邊結果有解，回傳
	else if (R_result) return R_result;//若右邊結果有解，回傳
	else return nullptr;//都沒有解，回傳此分支無解
}
```

heap 的輸出函式，重複的輸出 root node 然後 erase，直到沒有節點為止

```
void erase_output() {
//heap的輸出是把root node印出，然後erase，替代成新的root node，繼續印出root node，直到沒有節點為止
	while (number_of_node != 0) {
		cout << root->data;//輸出root node的值
		erase();//erase root node
		--number_of_node;//節點數-1
	}
}
```

往上走的遞迴函式

```
void rise(heap_node<element, order>* now) {//把現在節點的值往上遞迴交換到正確位置
	if (now == nullptr) return;//走到leaf node的小孩指標就停止
	if (now->father == nullptr) return;//最多走到root node就停
	if (//比較該節點與父節點的優先權決定是否需要交換節點，MIN與MAX類型的比較都涵蓋了
		(     (min_max == heap_type::MIN) && (now->father->priority > now->priority)  ) ||
		(     (min_max == heap_type::MAX) && (now->father->priority < now->priority)  ) ){
		swap(now->father->priority, now->priority);//交換優先權
		swap(now->father->data, now->data);//交換資料
		rise(now->father);//該值還是有可能不符合heap的規則，所以繼續遞迴往上走
	}
}
```

往下走的遞迴函式

```
void fall(heap_node<element, order>* now) {//把現在節點的值往下遞迴交換到正確位置
	if (now == nullptr) return;//走到leaf node的小孩指標就停止
	heap_node<element, order>* compare;
	//往下走不像往上走只有一條路，而是可能有2條路，所以用指標存放可能會走的路
	if (now->left == nullptr || now->right == nullptr) {//若有小孩指標為null
		compare = now->left != nullptr ? now->left : now->right;
		//走不是null的那邊(2個都null就為null)
	}
	else {//若2個小孩都存在，往根據heap類型的方向走
		if (min_max == heap_type::MIN) compare = now->left->priority < now->right->priority ?
now->left : now->right;
		//MIN類型，往優先權小的方向走
		else compare = now->left->priority > now->right->priority ? now->left : now->right;
		//MAX類型，往優先權大的方向走
	}

	if (compare == nullptr) return;//2個小孩都不存在(null)，直接結束
	if (//比較該節點與選擇子節點的優先權決定是否需要交換節點，MIN與MAX類型的比較都涵蓋了
		((min_max == heap_type::MIN) && (compare->priority < now->priority)) ||
		((min_max == heap_type::MAX) && (compare->priority > now->priority)) ){

		swap(compare->priority, now->priority);//交換優先權
		swap(compare->data, now->data);//交換資料
		fall(compare);//該值還是有可能不符合heap的規則，所以繼續遞迴往下走
	}
}
```

main 函式

```
int main() {
	string line;
	heap<char,int> all(heap_type::MIN);//創建heap，設類型為MIN
	char data,split;//暫存字元
	int pr;//暫存整數
	while (!cin.eof()) {
		cin >> data>> pr >> split;//輸入資料
		all.insert(data, pr);//插入到heap
	}
	all.erase_output();//輸出整個heap
	return 0;
}
```

3. Source code

GIST IMGUR 圖片

3-1:

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

template<typename iterator, typename compare = std::less<typename std::iterator_traits<iterator>::value_type> >  <T> 提供適用於 IntelliSe
//compare是比較大小的funciton object，若沒有傳入就用預設的
void partition(iterator begin, iterator end, iterator& loc, compare comp = compare()) {//把piviot放到對的位置(左邊都小於，右邊都大於)
    iterator left = begin, right = end;//起始點、終點
    loc = begin;//piviot設成起始點
    while (true) {
        while (comp(*loc,*right) && (loc != right)) {//從終點(right)往左比較，直到遇到比自己小的，就停下
            --right;//往左走
        }
        if (loc == right) {//如果自己是最小的，代表piviot的位置已經是正確的，結束函式
            return;
        }
        else {//如果不是最小的，就把piviot的位置換到新的位置，但還不是正確位置，繼續做
            swap(*loc, *right);//交換
            loc = right;//piviot指到新的位置
        }
        while (!comp(*loc, *left) && (loc != left)) {//從起點(left)往右比較，直到遇到比自己大的，就停下
            ++left;//往右走
        }
        if (loc == left) {//如果自己是最大的，代表piviot的位置已經是正確的，結束函式
            return;
        }
        else{//如果不是最大的，就把piviot的位置換到新的位置，但還不是正確位置，繼續做
            swap(*loc, *left);//交換
            loc = left;//piviot指到新的位置
        }
    }
}
template<typename iterator, typename compare = std::less<typename std::iterator_traits<iterator>::value_type> >
//compare是比較大小的funciton object，若沒有傳入就用預設的
void _quick_sort(iterator begin, iterator end, compare comp = compare()) {//quick sort主演算法
    iterator loc;//piviot
    partition(begin, end, loc, comp);//把piviot(begin)，放到對的位置(左邊都小於，右邊都大於)，並更新piviot位置到loc
    if (begin != loc) _quick_sort(begin, loc - 1, comp);//遞迴呼叫，但為了避免iterator(loc - 1)指到begin之前，要做防呆
    if (end != loc) _quick_sort(loc + 1, end, comp);//遞迴呼叫，但為了避免iterator(loc + 1)指到end之後，要做防呆
}

template<typename iterator,typename compare = std::less<typename std::iterator_traits<iterator>::value_type> >
//compare是比較大小的funciton object，若沒有傳入就用預設的
void quick_sort(iterator begin, iterator end,compare comp = compare()) {
    //因為iterator的end是指到最後一個element的下個位置，所以要先把end指回最後一個element，才不會存取違規
    if (distance(begin, end) <= 1) return;//防呆，如果begin end寫顛倒，就不做
    _quick_sort(begin, end - 1, comp);//把end指回最後一個element，呼叫真正的quick sort
}


int main() {
    vector<string> sequence;
    string tmp;
    while (cin >> tmp) {//讀取字串
        sequence.push_back(tmp);//加到sequence
    }
    quick_sort(sequence.begin(), sequence.end());//quick sort
    cout << sequence[0];//第一個輸出前面不加空白
    for (int i = 1;i < sequence.size();i++) {
        cout << ' ' << sequence[i];//後面的輸出前面都要加空白
    }
    return 0;
}
```

3-2:

```cpp
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
template<typename element,typename order>//element是heap內的資料型別(int)，order是優先權的資料型別(char)
class heap_node {//heap node的class
public:
    heap_node() {}
    heap_node(element i, order p, heap_node<element, order>* f) : data(i), priority(p),father(f) {}
    template<typename, typename> friend class heap;
private:
    element data;//資料
    order priority;//優先權
    heap_node<element, order>* father = nullptr;//父節點指標
    heap_node<element, order>* left = nullptr;//右邊小孩指標
    heap_node<element, order>* right = nullptr;//左邊小孩指標
};
enum class heap_type : bool {//heap的類型
    MIN,
    MAX
};
```

```cpp
template<typename element, typename order>//element是heap內的資料型別(int)，order是優先權的資料型別(char)
class heap {//heap的class
public:
    heap(){}
    heap(heap_type mm):min_max(mm){}
    void insert(element in,order pr) {//插入新資料
        insert(&root, root, in, pr, 0);//呼叫真正的insert function
        ++number_of_node;//節點數+1
    }
    void erase() {//刪除資料(root node)
        heap_node<element, order>* last = find_last(root,0);//找到complete binary tree最後一個node
        if (last == root) {//如果是root node，刪除就好
            delete root;
            root = nullptr;
            return;
        }
        root->priority = last->priority;//把root node的資料替代為complete binary tree最後一個node的資料
        root->data = last->data;//把root node的優先權替代為complete binary tree最後一個node的優先權
        if (last->father->left == last) {//如果最後一個node是父節點的左小孩
            last->father->left = nullptr;//父節點的左小孩設為空
        }
        else {//如果最後一個node是父節點的右小孩
            last->father->right = nullptr;//父節點的右小孩設為空
        }
        delete last;//刪除最後一個node
        fall(root);//新root node的值可能並不符合heap的規則，所以要把該值往下遞迴交換到正確位置
    }
    void erase_output() {
    //heap的輸出是把root node印出，然後erase，替代成新的root node，繼續印出root node，直到沒有節點為止
        while (number_of_node != 0) {
            cout << root->data;//輸出root node的值
            erase();//erase root node
            --number_of_node;//節點數-1
        }
    }
```

```
private:
    const heap_type min_max = heap_type::MIN;//heap類型
    heap_node<element, order>* root = nullptr;//root node
    int number_of_node = 0;//節點數
    void insert(heap_node<element, order>** now, heap_node<element, order>* father, element in, order pr,int index) {
        //now是現在節點的指標的指標，配置記憶體時才可以正確連接到，index用於確認是否為目標節點
        if (index == number_of_node) {//若現在節點的index與節點數相等，代表目前正位於新節點的位置
            *now = new heap_node<element, order>(in, pr, father);//配置新節點
            rise(*now);//新node的值可能並不符合heap的規則，所以要把該值往上遞迴交換到正確位置
            return;
        }
        if (*now != nullptr) {//不是leaf node的小孩指標且不為目標節點，就遞迴下去
            insert(&(*now)->left, *now, in, pr, index * 2 + 1);//左小孩的index為自己*2+1
            insert(&(*now)->right, *now, in, pr, index * 2 + 2);//右小孩的index為自己*2+2
        }
    }
    void rise(heap_node<element, order>* now) {//把現在節點的值往上遞迴交換到正確位置
        if (now == nullptr) return;//走到leaf node的小孩指標就停止
        if (now->father == nullptr) return;//最多走到root node就停
        if (//比較該節點與父節點的優先權決定是否需要交換節點，MIN與MAX類型的比較都涵蓋了
            (   (min_max == heap_type::MIN) && (now->father->priority > now->priority)  ) ||
            (   (min_max == heap_type::MAX) && (now->father->priority < now->priority)  ) ){

            swap(now->father->priority, now->priority);//交換優先權
            swap(now->father->data, now->data);//交換資料
            rise(now->father);//該值還是有可能不符合heap的規則，所以繼續遞迴往上走
        }
    }
void fall(heap_node<element, order>* now) {//把現在節點的值往下遞迴交換到正確位置
    if (now == nullptr) return;//走到leaf node的小孩指標就停止
    heap_node<element, order>* compare;
    //往下走不像往上走只有一條路，而是可能有2條路，所以用指標存放可能會走的路
    if (now->left == nullptr || now->right == nullptr) {//若有小孩指標為null
        compare = now->left != nullptr ? now->left : now->right;//走不是null的那邊(2個都null就為null)
    }
    else {//若2個小孩都存在，往根據heap類型的方向走
        if (min_max == heap_type::MIN) compare = now->left->priority < now->right->priority ? now->left : now->right;
        //MIN類型，往優先權小的方向走
        else compare = now->left->priority > now->right->priority ? now->left : now->right;
        //MAX類型，往優先權大的方向走
    }

    if (compare == nullptr) return;//2個小孩都不存在(null)，直接結束
    if (//比較該節點與選擇子節點的優先權決定是否需要交換節點，MIN與MAX類型的比較都涵蓋了
        ((min_max == heap_type::MIN) && (compare->priority < now->priority)) ||
        ((min_max == heap_type::MAX) && (compare->priority > now->priority)) ){

        swap(compare->priority, now->priority);//交換優先權
        swap(compare->data, now->data);//交換資料
        fall(compare);//該值還是有可能不符合heap的規則，所以繼續遞迴往下走
    }
}
heap_node<element, order>* find_last(heap_node<element, order>* now,int index) {//找到complete binary tree最後一個node
    if (now == nullptr) return  nullptr;//走到leaf node的小孩指標就回傳null，表示這一分支遞迴沒有解
    if (index == number_of_node - 1) {//若該節點index與節點數-1相等，代表該節點就是complete binary tree最後一個node
        return now;//回傳
    }
    heap_node<element, order>* L_result = find_last(now->left, index * 2 + 1);//往左遞迴，左小孩的index為自己*2+1，結果暫存到L_result
    heap_node<element, order>* R_result = find_last(now->right, index * 2 + 2);//往右遞迴，右小孩的index為自己*2+2，結果暫存到R_result
    if (L_result) return L_result;//若左邊結果有解，回傳
    else if (R_result) return R_result;//若右邊結果有解，回傳
    else return nullptr;//都沒有解，回傳此分支無解
}
int main() {
    string line;
    heap<char,int> all(heap_type::MIN);//創建heap，設類型為MIN
    char data,split;//暫存字元
    int pr;//暫存整數
    while (!cin.eof()) {
        cin >> data>> pr >> split;//輸入資料
        all.insert(data, pr);//插入到heap
    }
    all.erase_output();//輸出整個heap
    return 0;
}
```