1.pseudo code
因為時間不夠，覺得我應該來不及自己實做出紅黑樹，所以用了 https://www.coders-hub.com/2015/07/red-black-tree-rb-tree-using-c.html 這裡的紅黑樹 source code，做一些修改達成作業要求。結果後來卻發現他的程式有些 bug，所以沒有用到它對顏色做修正的部份，於是這其實只是一棵普通的 BST。

(1) main
宣告 RBtree 物件
while(讀進輸入檔案) 依照每一行輸入對 RBtree 物件做對應的操作

(2) class ID_Value(int b,float a)
原程式碼的 key 是整數，但因為題目有 ID 和 value 都需要比較，所以自己設計了 ID_Value 類別來達成需求。此類別中的成員有 float value 和 int ID。並幫類別設計了>, <, !=, ==, <<運算。針對當 value 相同時,按照 ID 大小升序排列的需求，設計<運算如下。(>同理類推)

```
friend bool operator< (const ID_Value& foo,const ID_Value& bar){
  if (foo.GetValue() != bar.GetValue() ) return (foo.GetValue() < bar.GetValue() );
  else return (foo.GetID() > bar.GetID() );
}
```

(3) I, D 操作
就是 BST 的 insert 和 delete 操作。只是 treenode 的 key 為 ID_Value 類別而非原本的 int。

(4) r 操作
設計 void r(int rank), void Inorder_r(node *current, int rank)和 void sort_r(int rank)功能函式。
利用 BST 的中序 travel，將 travel 的結果放進 sort_list 中達成排名。

```
void RBtree::Inorder_r(node *current, int rank){
    if(已經中止遞迴) return;
    travel 右邊 //因為排名是由大到小，將中序 travel 改成 右->中->左
    if (sort_list 長度< rank ) 將當前 treenode 的 key 放進 sort_list
    else{
      if(已排到需要的 rank，但當前 treenode 的 key 和前一個 value 相等){
        繼續放進 sort_list
      }else 已經排到需要的 rank 了，中止遞迴
    }
    travel 左邊 //因為排名是由大到小，將中序 travel 改成 右->中->左
}

void RBtree::sort_r(int rank){
  清空 sort_list
  呼叫 Inorder_r(this->root, rank) 從 root 開始 travel
}

void RBtree::r(int rank){
  呼叫 sort_r(rank);
  宣告 ans_list 用來放所有 rank 符合的 ID_Value
  將 sort_list 最後一筆放進 ans_list
  while(在 sort_list 由後向前檢查，value 跟最後一筆是一樣的) 都放進 ans_list
  整個 ans_list 是我們要的輸出結果
}
```

(4) R 操作
同 r 操作的 void Inorder_r(node *current, int rank)和 void sort_r(int rank)功能函式。

```
void RBtree::R(int rank){
  呼叫 sort_r(rank);
  宣告 ans_list 用來放所有 rank 符合的 ID_Value
  將 sort_list 最後一筆放進 ans_list
  while(在 sort_list 由後向前檢查，value 跟最後一筆是一樣的) 都放進 ans_list
  ans_list 的最後一筆是我們要的輸出結果
}
```

(5) v 操作
設計 void v(float value), void Inorder_v(node *current, float value)和 void sort_v(float value)功能函式。利用 BST 的中序 travel，將 travel 的結果放進 sort_list 中達成排名。

```
void RBtree::Inorder_r(node *current, int rank){
    if(已經中止遞迴) return;
    travel 右邊 //因為排名是由大到小，將中序 travel 改成 右->中->左
    if (當前 treenode 的 key value>=value) 將當前 treenode 的 key 放進 sort_list
    else 已經排到需要的 value 了，中止遞迴
    travel 左邊
}

void RBtree::sort_v(float value){
  清空 sort_list
  呼叫 Inorder_v(this->root, rank) 從 root 開始 travel
}

void RBtree::v(float value){
  呼叫 sort_v(value);
  sort_list 的最後一筆之排名(sort_list 長度)就是我們要的輸出結果
}
```

(6) V 操作
同 v 操作的 void Inorder_v(node *current, float value)和 void sort_v(float value)功能函式。

```
void RBtree::V(float value){
  呼叫 sort_v(value);
  sort_list 的最後一筆之排名(sort_list 長度)就是我們要的最高結果
  在 sort_list 由後往前找，value 仍相同的最前面一筆之排名就是我們要的最低結果
}
```

(7) K 操作
設計 K(float value, int num), Inorder_K(node *current, float value, int num)和 sort_K(float value, int num)功能函式。利用 BST 的中序 travel，將 travel 的結果放進 sort_list 中達成排名。

```
void RBtree::Inorder_K(node *current, float value, int num){
   if(已經中止遞迴) return;
   travel 右邊 //因為排名是由大到小，將中序 travel 改成 右->中->左
   if (當前 treenode 的 key value>=value){
     將當前 treenode 的 key 放進 sort_list
   }else{
     已經排到需要的 value 了，但繼續多放 num 個進去 sort_list 中
     if(已經多放 num 個進去) 中止遞迴
   }
   travel 左邊
}

void RBtree::sort_K(float value, int num){
  清空 sort_list
  呼叫 Inorder_K(this->root, rank) 從 root 開始 travel
}

void RBtree::K(float value, int num){
  呼叫 sort_K(value,num);
  宣告長度為 num 的 ans_list 用來放結果
  while(ans_list 未放滿){
     //從 sort_list 的倒數第 num 筆開始，向前&向後檢查
     if(前面一筆較接近我們要的 value) 將前面一筆放進 ans_list，並將 index 向前推進
     if(後面一筆較接近我們要的 value) 將後面一筆放進 ans_list，並將 index 向後推進
  }
}
```

2. 時間複雜度分析

(1) I, D 操作

為 BST 的 insert,delete，複雜度期望值 O(lgN)，最壞情形 O(N)。

(2) r, R 操作

```
void RBtree::r(int rank){
  呼叫 sort_r(rank);                                          //Inorder travel，最壞 O(N)
  宣告 ans_list 用來放所有 rank 符合的 ID_Value               //O(1)
  將 sort_list 最後一筆放進 ans_list
  while(由後向前檢查，value 跟最後一筆是一樣的) 都放進 ans_list    //最壞 O(N)
  整個 ans_list 是我們要的輸出結果
}
```

其他 v，V，K 操作的複雜度也類似。最壞時間複雜度皆為 O(N)。