HW1

1. MaxHeap & MinHeap
2. binary search tree

題目要求:

Max/Min Heap

```
template <class T>
class MaxPQ {
public:
    virtual ~MaxPQ() {}
        // virtual destructor
    virtual bool IsEmpty() const = 0;
        // return true iff the priority queue is empty
    virtual const T& Top() const = 0;
        // return reference to max element
    virtual void Push(const T&) = 0;
        // add an element to the priority queue
    virtual void Pop() = 0;
        // delete element with max priority
};
```

ADT 5.2: A max priority queue

1. Write a C++ abstract class similar to ADT 5.2 for the ADT *MinPQ*, which defines a min priority queue. Now write a C++ class *MinHeap* that derives from this abstract class and implements all the virtual functions of *MinPQ*. The complexity of each function should be the same as that for the corresponding function of *MaxHeap*.

2. Binary Search Tree

(a) Write a program to start with an initially empty binary search tree and make n random insertions. Use a uniform random number generator to obtain the values to be inserted. Measure the height of the resulting binary search tree and divide this height by $\log_2 n$. Do this for $n = 100, 500, 1000, 2000, 3000, \cdots, 10{,}000$. Plot the ratio $height/\log_2 n$ as a function of n. The ratio should be approximately constant (around 2). Verify that this is so.

(b) Write a C++ function to delete the pair with key k from a binary search tree. What is the time complexity of your function?

Max/Min heap

- ADT (max heap 與 min heap ADT 一樣，只有內部功能有差異)

```
template <class T>
class MinHeap{
private:
    T* heap;
    int heapSize;
    int capacity;
public:
    MinHeap(int theCapacity=10);
    ~MinHeap(){ delete[] heap; };
    void Push(const T& e);
    void Pop();
    T& Top() const;
    bool IsEmpty() const { return heapSize==0; }
    int Size() const { return heapSize; }
    void PrintHeap() const;
};
```

輸入:

```
const int arr[7] = {14, 30, 21, 44, 17, 20, 10};
```

輸出:

```
PrintHeap():
10 17 14 44 30 21 20

print heap in order:
10 14 17 20 21 30 44
```

Binary search tree

- ADT

```
template <class K, class E> // key & Element
class Dictionary{
private:
    TreeNode<K, E>* root;
public:
    Dictionary(){
        root = NULL;
    }
    bool IsEmpty() const { return root==NULL; }
    pair<K, E>* Get(const K& k) const;
    void Insert(const pair<K, E>& e);
    TreeNode<K, E>* findMin(TreeNode<K, E>* node);
    TreeNode<K, E>* Delete(TreeNode<K, E> *e, const K& k);
    TreeNode<K, E>* getRoot(){ return root; }
};
template <class K, class E>
pair<K, E>* Dictionary<K, E>::Get(const K& k) const {
    TreeNode<K, E>* current = root;
    while(current){
        if(k>current->data.first) current=current->right;
        else if(k<current->data.first) current = current->
        else return &current->data;
    }
    return 0;
}
```

tree node structure:

```
template <class K, class E>
class TreeNode{
public:
    TreeNode* left;
    TreeNode* right;
    pair<K, E> data;
    TreeNode(pair<K, E> e){
        this->data.first = e.first;
        this->data.second= e.second;
        this->left = NULL;
```

```
        this->right = NULL;
    }
};
```

inorder traverse for show trees

```
template <class K, class E>
int inorderTraverse(TreeNode<K, E>* node, int Level, bool
    int leftLevel=0, rightLevel=0;
    if(node!=NULL){
        leftLevel = inorderTraverse(node->left, Level+1, p
        if(print) cout<<node->data.first<<" "<<node->data.
        rightLevel = inorderTraverse(node->right, Level+1,
    }
    if(leftLevel>=rightLevel&&leftLevel>=Level) return lef
    else if(rightLevel>=leftLevel&&rightLevel>=Level) retu
    return Level;
}
```

a. the ratio height/log2(n) should be *approximately constant (around 2)*

```
void RatioTest(){
    const int range_from = 0, range_to = 10000;
    random_device rd;
    mt19937 generator(rd());
    uniform_int_distribution<int> uniform(range_from, rang
    int arr[10000], n[]={100, 500, 1000, 2000, 3000, 4000,
    for(auto &ni:n){
        Dictionary<int, int> BST;
        for(int i=0; i<ni; i++){
            arr[i] = uniform(generator);
            BST.Insert(make_pair(arr[i], arr[i]));
        }
        double height = (double)inorderTraverse(BST.getRoo
        double ratio = height/(log(ni)/log(2));
        cout<<ratio<<"\n";
```

```
    }
}
```

輸入:

```
uniform(generator); //隨機產生
```

輸出: (說明: 依序輸出n=100~n=10000時，ratio等於多少)

```
1.50515
1.8961
2.20755
2.27982
2.16436
2.25643
2.19732
1.99191
2.03552
2.15953
2.35998
2.10721
```

b. *delete the pair with key k from a binary search tree*

```
void DeleteTest(){
    int arr[]={100, 500, 1000, 2000, 3000, 4000, 5000, 600
    Dictionary<int, int> BST;
    for(auto &ai:arr){
        BST.Insert(make_pair(ai, ai));
    }
    BST.Delete(BST.getRoot(), 5000);
    inorderTraverse(BST.getRoot(), 0, true);
}
```

輸入:

```
int arr[]={100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7
```

動作: 刪去'5000'

輸出: (說明: insert 時，key 跟 element 皆為int 且數值相同)

```
500 500
1000 1000
2000 2000
3000 3000
4000 4000
6000 6000
7000 7000
8000 8000
9000 9000
10000 10000
```

補充: Delete函式時間複雜度: O(log2(n))