112703003 資訊二 黃柏淵 DS_HW3

第一大題：Function 的程式碼與解釋

1. Binary Search Tree

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

// 生成僅包含該數字的bst，並返回其root
TreeNode* CreateBST(int num) {
    return new TreeNode(num);
}

// 將數字插入bst，返回更新後的 root
TreeNode* InsertBST(int num, TreeNode* root) {
    if (root == nullptr) {
        return new TreeNode(num); // 如果 root 為空，生成新節點
    }
    if (num < root->val) {
        root->left = InsertBST(num, root->left); // 插入到左子樹
    } else if (num > root->val) {
        root->right = InsertBST(num, root->right); // 插入到右子樹
    }
    return root; // 返回更新後的根節點
}
```

```
// 列印bst的所有節點
void PrintBST(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    PrintBST(root->left);              // in-order左子樹
    cout << root->val << " ";
    PrintBST(root->right);             // in-order右子樹
}

// 計算bst的高度
int HeightBST(TreeNode* root) {
    if (root == nullptr) {
        return 0; // 空樹高度為0
    }
    int leftHeight = HeightBST(root->left);
    int rightHeight = HeightBST(root->right);
    return max(leftHeight, rightHeight) + 1; // 返回左右子樹高度的最大值+1
}
```

定義二元搜尋樹節點的結構體（TreeNode），每個節點有三個屬性：

- val: 節點的數值。
- left: 指向左子節點的指標，預設為 nullptr。
- right: 指向右子節點的指標，預設為 nullptr。

構造函數：

- 當建立一個 TreeNode 時，初始化 val 為指定數值，並將 left 和 right 設定為空指標（nullptr）

CreateBST 函數：

- 使用 new TreeNode(num) 創建一個節點，其中 num 是節點的數值，返回這個節點作為樹的根。

InsertBST 函數：

- 如果 root 為空（nullptr），表示當前子樹是空的，直接創建並返回新的節點。
- 如果 num < root->val，表示 num 應插入到當前節點的左子樹：
  i. 遞迴調用 InsertBST，將 num 插入到左子樹中，並更新 root->left。
- 如果 num > root->val，表示 num 應插入到當前節點的右子樹：
  i. 遞迴調用 InsertXXX，將 num 插入到右子樹中，並更新 root->right。
- 返回更新後的 root。

PrintBST 函數：

- 以 In-order Traversal 列印樹中所有節點的數值。
- 如果當前節點為空，直接返回。
- 遞迴訪問並列印左子樹的節點。
- 列印當前節點的數值。
- 遞迴訪問並列印右子樹的節點。

HeightBST 函數：

- 如果 root 為空，返回高度 0（空樹的高度為零）。
- 分別計算左子樹和右子樹的高度：
  i. 遞迴調用 HeightXXX 獲取左、右子樹的高度。
- 返回左右子樹高度的最大值，加 1（因為要算上當前節點本身）。

2. AVL Tree

```cpp
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    int height;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr), height(1) {}
};

// 取得節點高度
int GetHeight(TreeNode* node) {
    return node ? node->height : 0;
}

// 更新節點高度
void UpdateHeight(TreeNode* node) {
    if (node) {
        node->height = 1 + max(GetHeight(node->left), GetHeight(node->right));
    }
}
```

```cpp
// 計算平衡因子
int GetBalance(TreeNode* node) {
    return node ? GetHeight(node->left) - GetHeight(node->right) : 0;
}

// 右旋操作
TreeNode* RightRotate(TreeNode* y) {
    TreeNode* x = y->left;
    TreeNode* T2 = x->right;

    // 旋轉
    x->right = y;
    y->left = T2;

    // 更新高度
    UpdateHeight(y);
    UpdateHeight(x);

    return x;
}
```

```cpp
// 左旋操作
TreeNode* LeftRotate(TreeNode* x) {
    TreeNode* y = x->right;
    TreeNode* T2 = y->left;

    // 旋轉
    y->left = x;
    x->right = T2;

    // 更新高度
    UpdateHeight(x);
    UpdateHeight(y);

    return y;
}

// 建立僅包含一個數字的 AVL Tree
TreeNode* CreateAVL(int num) {
    return new TreeNode(num);
}
```

```cpp
// 插入節點並保持 AVL 平衡
TreeNode* InsertAVL(int num, TreeNode* root) {
    if (root == nullptr) {
        return new TreeNode(num); // 新節點
    }

    // 遞迴插入節點
    if (num < root->val) {
        root->left = InsertAVL(num, root->left);
    } else if (num > root->val) {
        root->right = InsertAVL(num, root->right);
    } else {
        return root; // 不允許重複節點
    }

    // 更新高度
    UpdateHeight(root);

    // 檢查平衡因子
    int balance = GetBalance(root);

    // 左重：右旋
    if (balance > 1 && num < root->left->val) {
        return RightRotate(root);
    }

    // 右重：左旋
    if (balance < -1 && num > root->right->val) {
        return LeftRotate(root);
    }

    // 左右重：左旋後右旋
    if (balance > 1 && num > root->left->val) {
        root->left = LeftRotate(root->left);
        return RightRotate(root);
    }
```

```cpp
    // 右左重：右旋後左旋
    if (balance < -1 && num < root->right->val) {
        root->right = RightRotate(root->right);
        return LeftRotate(root);
    }

    return root;
}

// 印出 AVL Tree (in-order)
void PrintAVL(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    PrintAVL(root->left);
    cout << root->val << " ";
    PrintAVL(root->right);
}

// 計算 AVL Tree 的高度
int HeightAVL(TreeNode* root) {
    return GetHeight(root);
}
```

TreeNode 結構體：

- val：節點的數值。
- left：指向左子節點的指標。
- right：指向右子節點的指標。
- height：該節點的高度，初始化為 1（單一節點的高度）。
- 構造函數：用指定的數值初始化節點，left 和 right 預設為空指標，height 設為 1。

GetHeight 函數：

- 返回節點的高度。如果節點為 nullptr，返回高度 0。

UpdateHeight 函數：

- 每次節點有更新（插入、旋轉）後，都需要重新計算節點高度。
- 更新節點的高度。
- 節點高度等於其左右子樹最大高度加 1（包含自己）。

GetBalance 函數：

- AVL Tree 保持平衡因子在 [-1, 1] 範圍內，若超出範圍則需要調整。
- 計算節點的平衡因子（Balance Factor）。
- 平衡因子定義為左子樹高度減右子樹高度。

RightRotate 函數：(用於處理「左重」（平衡因子大於 1）的情況。)

- 將 x（左子節點）提升為新根節點。
- 原根節點 y 成為 x 的右子節點。
- x 的右子樹 T2 被掛接到 y 的左子節點。
- 更新 x 和 y 的高度。
- 返回旋轉後的新根節點 x

LeftRotate 函數：(用於處理「右重」（平衡因子小於 -1）的情況。)

- 將 y（右子節點）提升為新根節點。
- 原根節點 x 成為 y 的左子節點。
- y 的左子樹 T2 被掛接到 x 的右子節點。
- 更新 x 和 y 的高度。
- 返回旋轉後的新根節點 y。

CreateAVL 函數：

- 創建一個僅包含數值 num 的 AVL Tree 節點。

InsertAVL 函數：

- 如果當前樹為空，直接創建新節點。
- 根據 num 值的大小，遞迴插入到左或右子樹。
- 更新節點高度。
- 檢查平衡因子並採取相應的旋轉操作：
  - i. 左重（>1）：
    1. 單右旋。
    2. 或左旋後右旋（左右重）。
  - ii. 右重（<-1）：
    1. 單左旋。
    2. 或右旋後左旋（右左重）。
- 返回平衡後的根節點。

PrintAVL 函數：

- In-order Traversal 列印 AVL Tree 的節點值。

HeightAVL 函數：

- 返回整棵 AVL Tree 的高度。

3. Treap

```cpp
// 節點結構
struct TreeNode {
    int val;         // 節點value
    double priority; // 節點priority (min-heap)
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x, double p) : val(x), priority(p), left(nullptr), right(nullptr) {}
};

// 右旋操作
TreeNode* RightRotate(TreeNode* y) {
    TreeNode* x = y->left;
    TreeNode* T2 = x->right;

    // 執行旋轉
    x->right = y;
    y->left = T2;

    return x;
}
```

```cpp
// 左旋操作
TreeNode* LeftRotate(TreeNode* x) {
    TreeNode* y = x->right;
    TreeNode* T2 = y->left;

    // 執行旋轉
    y->left = x;
    x->right = T2;

    return y;
}

// 建立僅包含一個數字的 Treap 節點
TreeNode* CreateTreap(int num, double priority) {
    return new TreeNode(num, priority);
}
```

```cpp
// 維持 BST 和 min-heap 性質
TreeNode* InsertTreap(int num, double priority, TreeNode* root) {
    if (root == nullptr) {
        return CreateTreap(num, priority); // 建立新節點
    }

    // 按 BST 性質插入節點
    if (num < root->val) {
        root->left = InsertTreap(num, priority, root->left);

        // 維持 min-heap 性質
        if (root->left->priority < root->priority) {
            root = RightRotate(root);
        }
    } else if (num > root->val) {
        root->right = InsertTreap(num, priority, root->right);

        // 維持 min-heap 性質
        if (root->right->priority < root->priority) {
            root = LeftRotate(root);
        }
    }

    return root;
}
```

```cpp
// 列印 Treap (in-order)
void PrintTreap(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    PrintTreap(root->left);
    cout << "(" << root->val << ", " << root->priority << ") ";
    PrintTreap(root->right);
}

// 計算 Treap 的高度
int HeightTreap(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    int leftHeight = HeightTreap(root->left);
    int rightHeight = HeightTreap(root->right);
    return max(leftHeight, rightHeight) + 1;
}
```

TreeNode 結構體：

- val：節點的值，按 BST 性質 排序。
- priority：節點的優先級，按 min-heap 性質 排序。
- left 和 right：分別指向左子節點和右子節點。
- 構造函數：初始化節點的值和優先級，左右子節點預設為空指標（nullptr）。

RightRotate 函數：(用於修復左子節點的優先級小於根節點的情況)

- 將 x（y->left）提升為新根節點。
- 原根節點 y 成為 x 的右子節點。
- x 的右子樹 T2 被移動到 y 的左子樹。
- 返回旋轉後的新根節點 x。

LeftRotate 函數：(用於修復右子節點的優先級小於根節點的情況)

- 將 y（x->right）提升為新根節點。
- 原根節點 x 成為 y 的左子節點。
- y 的左子樹 T2 被移動到 x 的右子樹。
- 返回旋轉後的新根節點 y。

CreateTreap 函數：

- 建立一個僅包含指定值和優先級的 Treap 節點。

InsertTreap 函數：

- 插入一個具有值 num 和優先級 priority 的節點，並保持 Treap 的 BST 和 min-heap 性質。
- 如果樹為空，建立新節點作為根。
- 如果 num 小於當前節點的值，將其插入到左子樹：
    i. 如果插入後左子節點的優先級小於當前節點的優先級，執行右旋操作。
- 如果 num 大於當前節點的值，將其插入到右子樹：
    i. 如果插入後右子節點的優先級小於當前節點的優先級，執行左旋操作。
- 返回更新後的根節點。

PrintTreap 函數：

- 以 In-order Traversal 的方式列印 Treap 節點（值和優先級）。

HeightTreap 函數：

- 計算 Treap 的高度。
- 如果節點為空，返回高度 0。
- 遞迴計算左子樹和右子樹的高度。
- 返回兩者的最大值加 1（包含當前節點）。

4. SkipList

```cpp
// 定義節點結構
struct SkipListNode {
    int val;
    vector<SkipListNode*> forward; // 向前的指針陣列

    SkipListNode(int x, int level) : val(x), forward(level, nullptr) {}
};

// 定義 SkipList 結構
struct SkipList {
    int maxLevel;            // 最大層數
    SkipListNode* head;      // SkipList 的頭節點

    SkipList(int maxLevel) {
        this->maxLevel = maxLevel;
        head = new SkipListNode(-1, maxLevel); // 建立虛擬頭節點
    }
};
```

```cpp
// 決定節點層數的函數 (基於銅板結果)
int GetLevelFromCoinToss(const string& coinToss) {
    int level = 1; // 至少一層
    for (char c : coinToss) {
        if (c == 'H') {
            level++;
        } else {
            break; // 遇到 T 結束計算
        }
    }
    return level;
}

// 創建一個節點
SkipList* CreateSkipList(int maxLevel) {
    return new SkipList(maxLevel);
}
```

```cpp
void InsertSkipList(int val, const string& coinToss, SkipList* list) {

    vector<SkipListNode*> update(list->maxLevel, nullptr);
    SkipListNode* curr = list->head;

    // 找到每層的插入位置
    for (int i = list->maxLevel - 1; i >= 0; i--) {
        while (curr->forward[i] && curr->forward[i]->val < val) {
            curr = curr->forward[i];
        }
        update[i] = curr;
    }

    // 建立新節點
    SkipListNode* newNode = new SkipListNode(val, level);

    // 更新指針
    for (int i = 0; i < level; i++) {
        newNode->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = newNode;
    }
}

// 列印 SkipList
void PrintSkipList(SkipList* list) {
    cout << "SkipList:" << endl;
    for (int i = list->maxLevel - 1; i >= 0; i--) {
        SkipListNode* curr = list->head->forward[i];
        cout << "Level " << i + 1 << ": ";
        while (curr) {
            cout << curr->val << " ";
            curr = curr->forward[i];
        }
        cout << endl;
    }
}
```

```cpp
// 計算 SkipList 的高度
int HeightSkipList(SkipList* list) {
    int height = 0;
    SkipListNode* curr = list->head;
    while (curr->forward[height]) {
        height++;
    }
    return height;
}
```

SkipListNode 結構體：

- val：節點的值。
- forward：向前的指針陣列，表示該節點在每一層的連接關係。
- 構造函數：初始化節點的值 x 和層數 level，將 forward 初始化為大小為 level 的空指針向量。

SkipList 結構：

- maxLevel：跳表的最大層數，用於限制每個節點最多能參與的層數。
- head：虛擬頭節點，方便操作（例如插入和查找）。

- 構造函數：初始化跳表的最大層數 maxLevel。建立一個值為 -1 的虛擬頭節點，並初始化其指針向量。

GetLevelFromCoinToss 函數：

- 通過模擬擲銅板決定新節點的層數，H（正面）增加層數，T（反面）停止

CreateSkipList 函數：

- 創建一個新的跳表

InsertSkipList 函數：

- 插入值為 val 的節點到跳表，並隨機分配其層數。
- 使用 GetLevelFromCoinToss 決定新節點的層數，並限制其最大值為 list->maxLevel。
- 準備一個 update 向量，用於記錄每層中應指向新節點的節點。
- 從最高層開始，按順序找到每層的插入位置（按 BST 性質尋找）。
- 建立新節點，更新各層中的連接關係：新節點指向原來的下一節點。更新節點指向新節點。

PrintSkipList 函數：

- 從最高層開始，依次遍歷每一層。
- 列印每層的節點值。

HeightSkipList 函數：

- 從第 0 層開始，逐層檢查是否存在節點。
- 若該層有節點，計算高度 +1，直到遇到空層為止。

第二大題：測試程式碼正確性

1. Binary Search Tree

```cpp
int main() {
    TreeNode* root = CreateBST(3);      // 建立根節點
    root = InsertBST(2, root);          // 插入數字
    root = InsertBST(1, root);
    root = InsertBST(5, root);
    root = InsertBST(4, root);

    cout << "BST 中的所有節點：" << endl;
    PrintBST(root);
    cout << endl;

    cout << "BST 的高度：" << HeightBST(root) << endl;

    return 0;
}
```

```
BST 中的所有節點：
1 2 3 4 5
BST 的高度：3
```

依序輸入 3,2,1,5,4 之後，bst 的結構會長得如圖所示，而我的程式碼計算樹的總高度是左右子樹高度的最大值加 1，因此樹高為 3。

```
      3
     / \
    2   5
   /   /
  1   4
```

2. AVL Tree

```cpp
int main() {
    TreeNode* root = nullptr;

    root = InsertAVL(1, root);
    root = InsertAVL(2, root);
    root = InsertAVL(3, root);
    root = InsertAVL(4, root);
    root = InsertAVL(5, root);
    root = InsertAVL(6, root);
    root = InsertAVL(7, root);

    cout << "AVL Tree 中的所有節點：" << endl;
    PrintAVL(root);
    cout << endl;

    cout << "AVL Tree 的高度：" << HeightAVL(root) << endl;

    return 0;
}
```

```
AVL Tree 中的所有節點：
1 2 3 4 5 6 7
AVL Tree 的高度：3
```

依序輸入 1,2,3,4,5,6,7 之後，avl_tree 的結構會長得如圖所示。而程式碼計算樹的總高度是左右子樹高度的最大值加 1，因此樹高為 3。

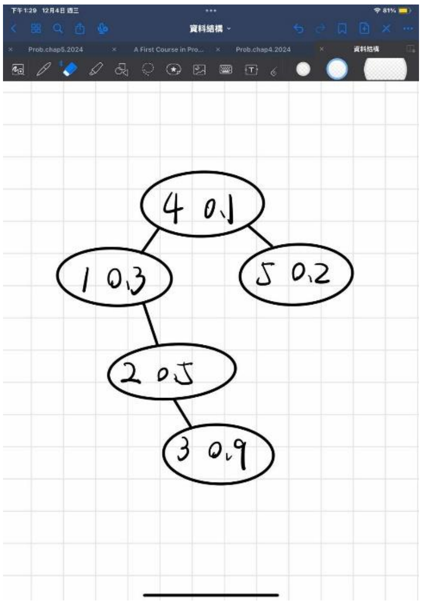
```
     4
    / \
   2   6
  / \ / \
 1  3 5  7
```

3. Treap

```cpp
int main() {
    TreeNode* root = nullptr;

    // 依序插入節點 (value, priority)
    root = InsertTreap(3, 0.9, root);
    root = InsertTreap(2, 0.5, root);
    root = InsertTreap(1, 0.3, root);
    root = InsertTreap(5, 0.2, root);
    root = InsertTreap(4, 0.1, root);

    cout << "Treap 中的所有節點：" << endl;
    PrintTreap(root);
    cout << endl;

    cout << "Treap 的高度：" << HeightTreap(root) << endl;

    return 0;
}
```

```
Treap 中的所有節點：
(1, 0.3) (2, 0.5) (3, 0.9) (4, 0.1) (5, 0.2)
Treap 的高度：4
```

而程式碼計算樹的總高度是左右子樹高度的最大值加 1，因此樹高為 4。



4. SkipList

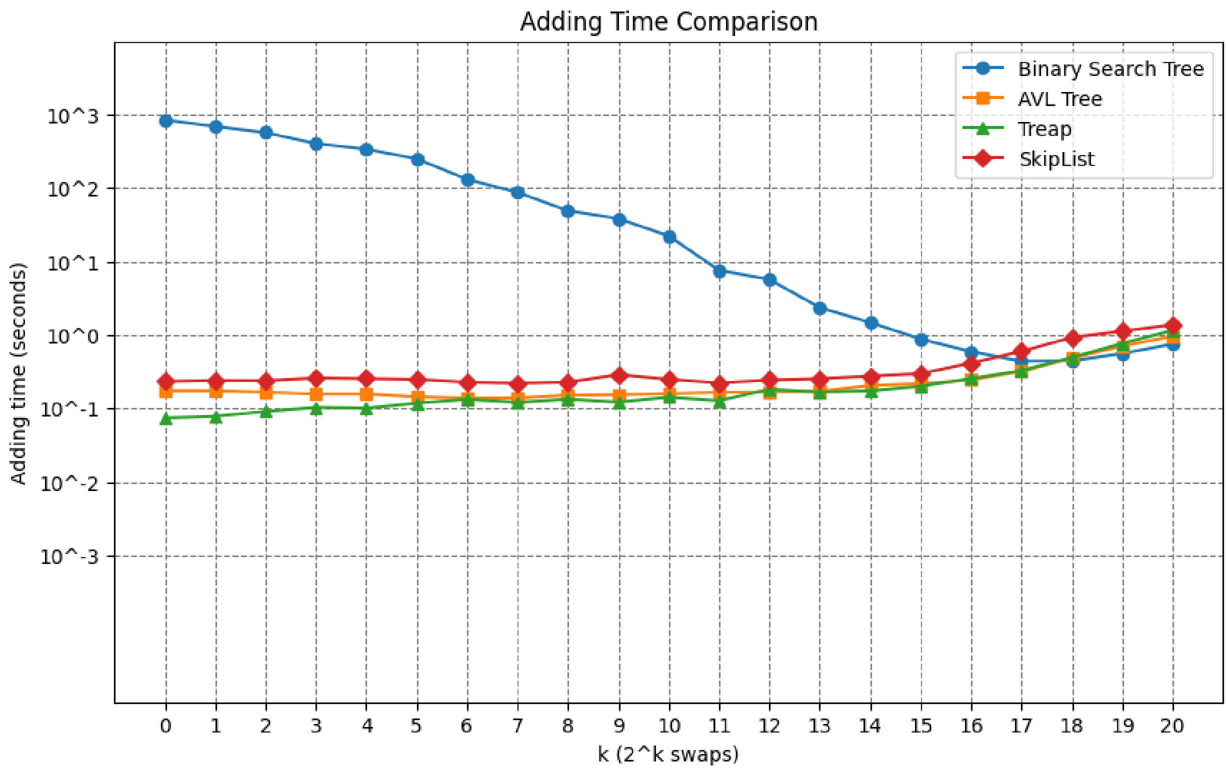
```
int main() {
    int maxLevel = 5; // 假設最大層數為 5
    SkipList* list = CreateSkipList(maxLevel);

    // 依序插入節點 (值, 銅板結果)
    InsertSkipList(1, "HHT", list);
    InsertSkipList(2, "T", list);
    InsertSkipList(3, "HT", list);
    InsertSkipList(4, "HHHT", list);
    InsertSkipList(5, "T", list);
    InsertSkipList(6, "T", list);
    InsertSkipList(7, "HHT", list);

    // 列印 SkipList
    PrintSkipList(list);

    // 列印高度
    cout << "SkipList 的最大高度：" << HeightSkipList(list) << endl;

    return 0;
}
```

```
SkipList:
Level 5:
Level 4: 4
Level 3: 1 4 7
Level 2: 1 3 4 7
Level 1: 1 2 3 4 5 6 7
SkipList 的最大高度：4
```

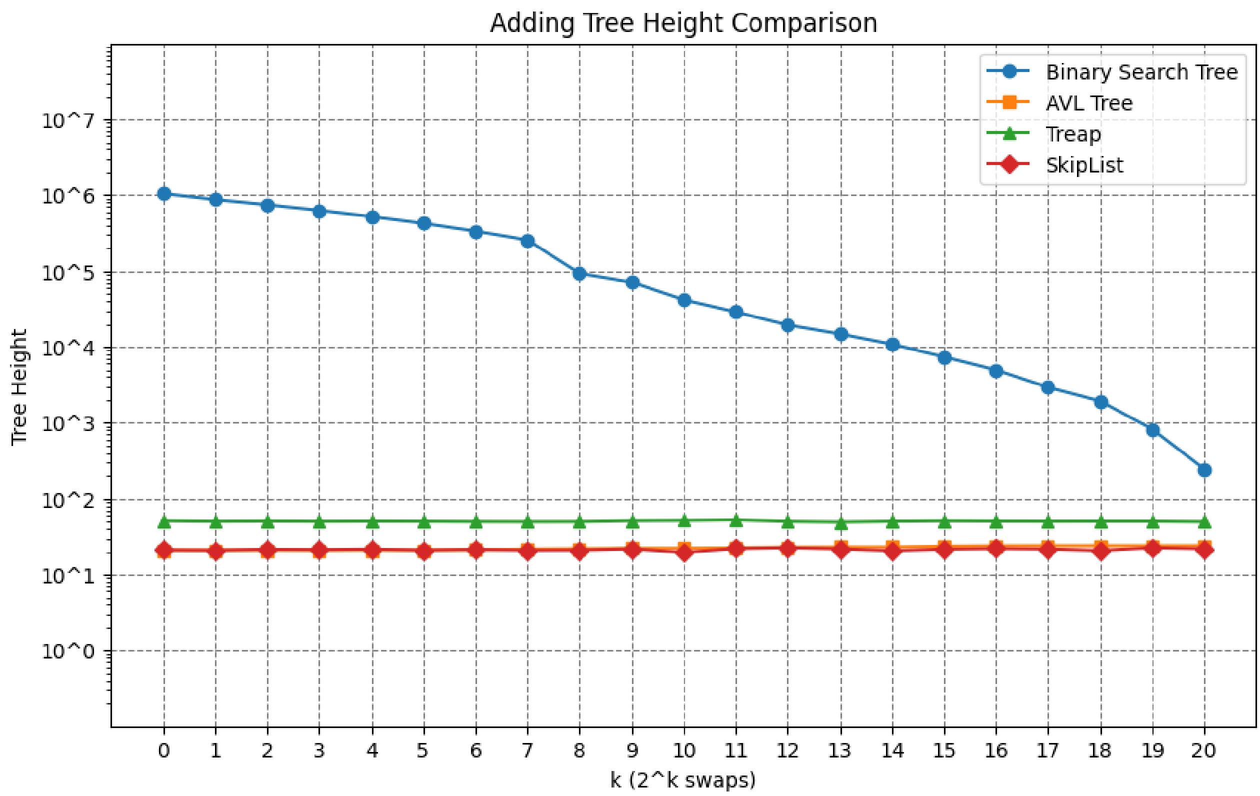
SkipList 出來的圖如上，從 level1 到 level4，因此最大高度為 4。

第三大題：實驗圖

圖一：



圖二：



第四大題：解釋實驗圖

圖一：

Binary Search Tree: 一開始的時間最久，而會隨著 swap 的次數增加，使得時間減少。當 k 很大時，數據排列幾乎是完全隨機的。隨機排列的數據插入時，整體接近於均勻分佈，形成的樹結構相對平衡。插入操作的平均高度相對較低，每個節點的插入時間幾乎是對數級別 O(log n)，所以建樹時間較短。當 k 減少時，隨機交換次數不足以打亂數據，使得數據排列逐漸接近初始的遞增順序。接近有序的數據插入時，會導致樹變得不平衡，最糟糕的情況是形成 Linked List。插入操作的時間從 O(log n) 增長到 O(n)，建樹時間開始顯著增加。當 k 非常小時，數據排列幾乎是完全有序的（遞增或接近遞增）。此時，插入順序導致每次新節點都成為前一個節點的右子節點，最終形成一個 Linked List。樹的高度接近節點數量 n，每次插入的時間為 O(n)，整體建樹時間呈現近似二次增長。

AVL Tree: 插入一個節點時，尋找適當位置插入，時間複雜度為 O(h)，其中 h 是樹高。在插入後檢查每個祖先節點的平衡因子，最多進行 O(log n) 次旋轉。因此單次插入時間複雜度：O(log n)。插入 n 個節點需要重複上述操作 n 次，時間複雜度為： O(n log n)。隨機交換的時間複雜度：交換 k 次的總時間為 O(k)。總時間複雜度是：T(n, k) = O(n log n) + O(k)。
而時間則呈現增長，因為每次交換涉及兩個數組元素的隨機存取和交換，總交換成本為 O(k)。當 k 較小時，數組的排列接近有序，AVL 樹在插入過程中需要的旋轉次數較少，插入時間較短。當 k 越大時，交換的時間開銷越高。

Treap: 結合了 BST 和 Heap 的特性：插入過程需要遍歷樹的高度 O(h)。Heap 性質：在插入過程中，如果違反 Heap 性質，可能需要執行旋轉操作。平均情況下，Treap 是一棵隨機平衡的樹，高度為 O(log n)，因此單次插入的時間複雜度為 O(log n)。插入 n 個節點的總時間為 O(n log n)。因為 k 次隨機交換的時間複雜度為 O(k)。因此每次測試的總時間由插入操作和隨機交換組成： T(n, k) = O(n log n) + O(k)。Treap 的時間呈現增長，因為 k 次隨機交換的成本是線性增長的，隨著 k 增大，交換操作耗費的時間顯著增加，直接影響總時間。當 k 較小時，數據接近有序，Treap 插入的路徑較短，旋轉操作較少，插入時間相對較低。

SkipList: 插入一個節點，首先需要從最高層向下尋找插入位置（類似於搜索）。每層最多進行一次比較，平均情況下 Skip List 的高度為 O(log n)，因此單次插入的時間複雜度為 O(log n)。插入 n 個節點需要重複上述操作 n 次，因此插入所有節點的總時間複雜度為 O(n log n)。k 次隨機交換的時間複雜度為

O(k)，其中每次交換操作的成本為 O(1)。總時間包括隨機交換和插入操作的成本為 T(n, k) = O(k) + O(n log n)。當 k 較小，此時總時間幾乎完全由插入操作的 O(n log n) 主導，而插入時間的變化主要受隨機層數分佈的微小波動影響，因此時間趨勢較為平穩。當 k 增大，雖然插入操作的成本仍然主導，但隨機交換操作的額外時間導致了總時間的緩慢增長。因此 SkipList 這條線呈現成長趨勢。

圖二：(log_2 是指 log 以 2 為底, log_(1/p)是指 log 以 1/p 為底)

<u>Binary Search Tree</u>: 一開始 swap 次數是 2^0 時，樹高是 2 的 20 次方，當 swap 次數增加時，樹高會開始遞減。當 k 大時，數據經過大量隨機交換後，接近完全隨機排列。隨機排列的數據插入時，節點的分佈較均勻，樹的結構接近於「平衡樹」。在接近平衡的情況下，平均高度為 O(log n)。因此，當 k 大時，平均樹高較低。當 k 開始減少時，隨機交換次數減少，數據的隨機性降低，排列開始部分保留初始的順序性。由於數據不夠隨機，部分區域的插入順序接近遞增或遞減，導致這些區域的樹結構變得不平衡（即一些分支變得更深）。不平衡樹的高度會增加，接近於線性增長（最糟情況是 O(n)），因此此區域的樹高明顯上升。當 k 較小時，隨機交換次數很少，多數數據仍接近初始遞增排列。數據接近有序排列時，插入順序會導致退化，形成近乎鏈表的結構（例如，所有節點都是右子節點或左子節點）。退化的高度接近節點數 n，為最差情況的 O(n)。因此，當 k 越小，平均樹高逐漸逼近完全有序時的最大值（例如 k=20k = 2^0k=20 時，高度為 1,048,576）。

<u>AVL Tree</u>: 在圖二中呈現的高度在 21 到 24，在 k 變大時，高度略微上升。理論高度上限與節點數 n 呈對數關係 H_max ≈ 1.44*log_2(n) − 0.33。對於 n = 2 ^ 20，計算得出理論高度約為 H_max ≈ 1.44 * 20 − 0.33 ≈ 28.47。但在實驗中，隨機插入的數據會生成接近平衡的 AVL 樹，因此實際高度比理論最大高度低。而當 k 增大時，陣列被打亂得越徹底，插入順序越隨機。對於更隨機的插入順序，儘管 AVL 樹會自動平衡，但隨機插入可能使局部子樹的結構變得稍微不均勻，導致整體高度略微增加。

<u>Treap</u>: 對樹的平均高度的影響，結果顯示高度大致穩定在 49-52 之間，略有波動。在鍵值和優先級隨機的情況下，Treap 的高度與隨機 bst 相似，預期高度為 H_avg ~ 4.311 * log_2(n)，對於 n = 2 ^ 20，高度理論值為 H_avg ~ 4.311 * 20 ≈ 86.22，但程式中的優先級由 rand() 生成，且數組經過有限次隨機交換，因此實際高度會較低。雖然優先級是隨機生成的，但對於 n = 2 ^ 20 的大量數據，優先級分佈接近均勻，樹的結構因此保持穩定。無論 k 值為何，優先級的隨機性主導了樹的結構，使得插入順序對最終高度影響有限。即使鍵值交換次數不同，Treap 的旋轉操作會自動調整樹的平衡，因此高度基本保持穩定。

<u>SkipList</u>: 高度取決於最大層數 maxLevel 和插入時的概率 p。每個節點的層數 l 是隨機生成的，滿足 P(層數 = l) = [p ^ (l − 1)] * (1 − p)，預期高度 H 與節點數 n 的對數成正比，公式為： H_avg ~ log_(1 / p)(n)，若 p = 0.5，則高度滿足 H_avg ~ log_2(n)，對於 n = 2 ^ 20，理論高度約為 H_avg ~ 20。跳表的層數由隨機分佈決定，而非插入順序，因此即使數組經過隨機交換，對層數的分佈影響不大。但由於每次測試中隨機生成層數會帶來波動，實際高度略高於理論值，穩定在 21 左右。

第五大題：遇到的問題

1. Bst 在 swap 次數少的時候跑太慢，時間跑不出來，所以後來改成從 swap 次數多的開始記錄時間。
2. 就是 treap 在 swap 的時候，理論值的樹高為甚麼會跟實際差那麼多？
3. 高度那張(圖二)，覺得 avl tree 跟 skip list 高度重疊有點怪怪的。