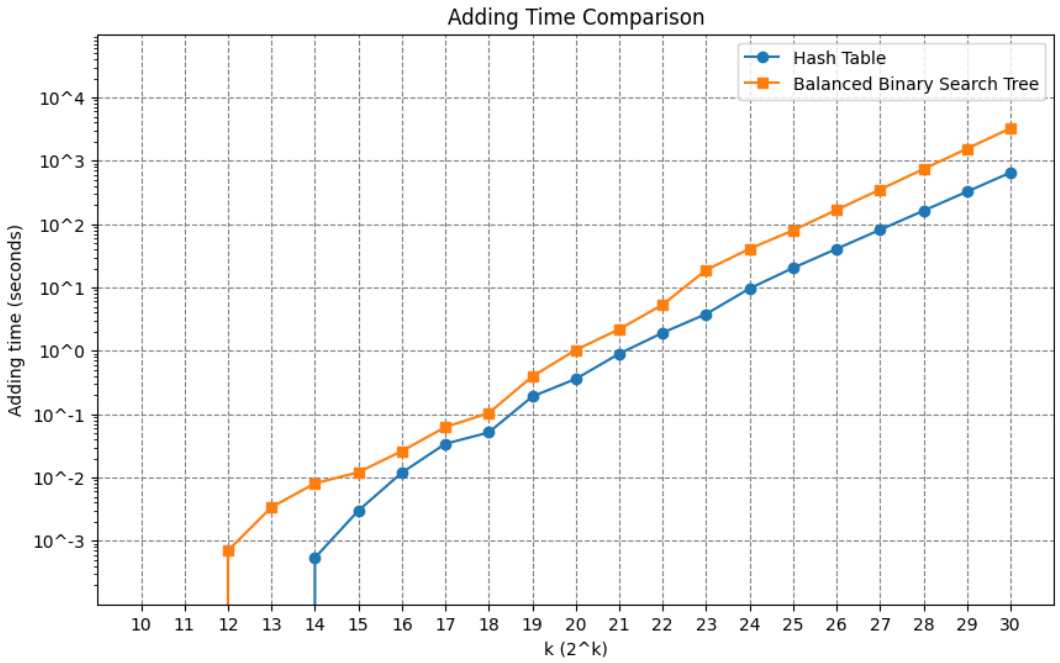
112703003 資訊二 黃柏淵 DS_HW4

第一題：新增資料所需時間



估計：

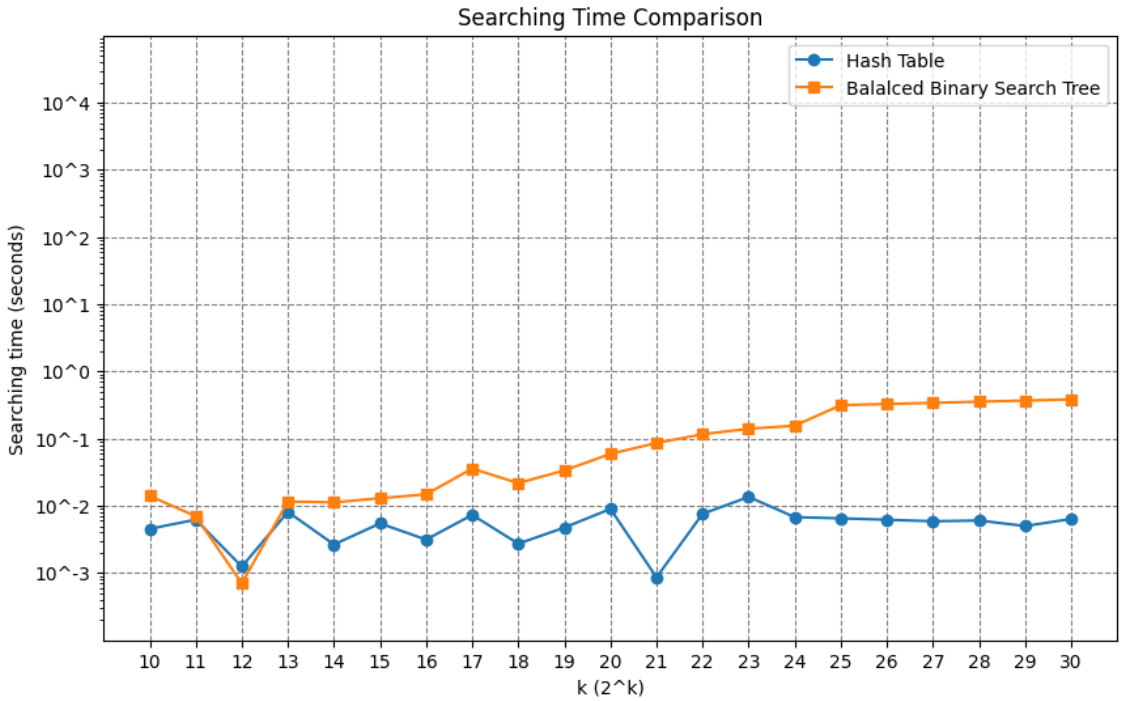
1. Hash Table：k 從 25 次方後使用估計。由於 n = 2^k，當 k 增加 1 時，n 會變成原來的 2 倍。假設上一次的數據量是 n，執行時間是 T(n)。這一次的數據量是 2n，執行時間是 T(2n)，時間複雜度是 O(n) : T(2n) / T(n) 應該約等於 2。所以我用的估計時間是呈現 2 倍。
2. Balanced Binary Search Tree： k 從 25 次方後使用估計。因為總複雜度是 O(n log n)，當 n 變為 2n 時，T(2n) = (2n) * log(2n)，T(n) = n log(n)，T(2n)/T(n) ≈ 2 * (1 + 1/log(n))。當 n 很大時（假設 k = 25），log(n) ≈ 25，理論增長比例 ≈ 2*(1 + 1/25) ≈ 2.1 倍。所以我用的估計時間是呈現 2.1 倍。

實驗圖解釋：

1. Hash Table：k 從 12 開始，數據成長趨勢大約呈現 2 倍。在 unordered_map 中，單一插入操作的平均時間複雜度是 O(1)，所以整體的平均時間複雜度是 O(n) ，也就是 O(2^k)，在最壞情況下（大量 hash collision），複雜度會退化到 O(n^2)，程式中包含了生成隨機數的時間，這部分是 O(n)。因此總時間複雜度在平均情況：O(n) = O(2^k)，最壞情況：O(n^2) = O(2^2k)，可以觀察到實際執行時間應該大約是隨著 k 呈指數增長（2^k），這符合平均情況的理論複雜度 O(2^k)。

2. Balanced Binary Search Tree：每次插入操作的複雜度是 O(log n)，要插入 n 個元素，所以總複雜度是 O(n log n)。當 n 變為 2n 時：T(2n) = (2n)log(2n)，T(n) = n log(n)，T(2n) / T(n) = (2n log(2n)) / (n log(n)) ≈ 2 * (1 + 1/log(n))。當 n 很大時（假設 k = 25）：log(n) ≈ 25。理論增長比例 ≈ 2(1 + 1/25) ≈ 2.1 倍。與 Hash Table 的固定 2 倍增長相比，Balanced BST 的增長比例略高於 2 倍，所花的時間多於 Hash Table(折線上面那條)，而成長比例則呈現約 2.1 倍。

第二題：資料結構搜尋資料所需時間



估計：log2(x) = log 以 2 為底的 x

1. Hash Table：在 k = 26 的時候，程式嘗試分配動態記憶體失敗，被終端機直接終止執行，因此使用估計。因為平均搜尋時間複雜度是 O(1)，即與數據規模 n 無關，所以(k = 26 ~ 30)取現有數據中最慢時間跟最快的時間當作上下界，並在上下界中隨機產生 5 筆數據當作搜尋時間。

2. Balanced Binary Search Tree：在 k = 26 的時候，程式嘗試分配動態記憶體失敗，被終端機直接終止執行，因此使用估計。平均搜尋時間複雜度是 O(log n)，搜尋時間會隨著 k 緩慢增加，並呈現接近對數增長的趨勢。估計時間基於先前的搜尋時間趨勢及時間複雜度推得，我們知道 T(k = 25) 的實際搜尋時間，假設為 t_{25}，當 k=26 時，理論搜尋時間應為 T(k = 26) ≈ T(k = 25) * [log2(2^25) / log2(2^26)] = T(k = 25) * (26/25)，搜尋時間將增加約 1.04 倍，因此 k 從 26 到 30 的搜尋時間為根據前一

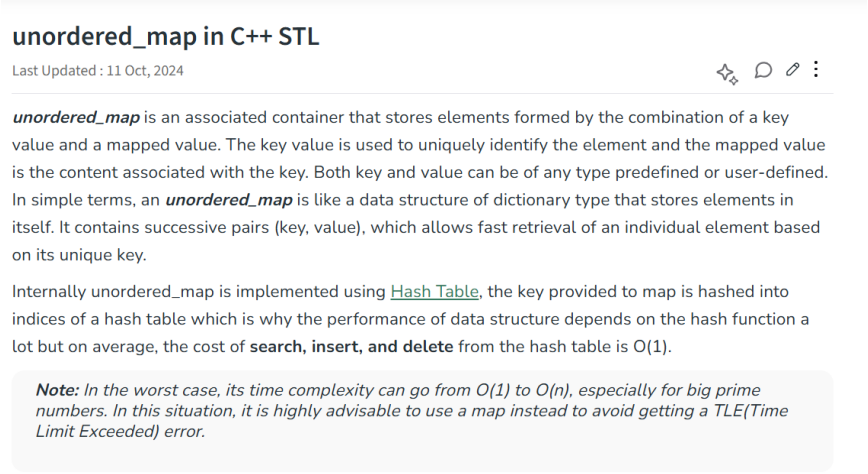
筆數據的時間乘上 1.04 而得。

實驗圖解釋：

1. Hash Table：如圖所示，搜尋時間並無呈現明顯變動，因為 hash_table.reserve(n); 預先分配了足夠的 buckets，確保減少 hash 碰撞的機率以及避免在插入過程中需要 rehash。不論數據規模(n)多大，搜尋操作基本上都是先計算 hash 值 O(1)，然後找到對應的 bucket O(1)，再 bucket 中搜尋 O(1)（因為 reserve(n) 使得碰撞很少），就算 k 再大，但因為 bucket 數量也足夠多，每個 bucket 的平均長度仍然維持在很小的數字，所以搜尋時間基本保持不變。

2. Balanced Binary Search Tree：如圖所示，Balanced BST 所花的搜尋時間高於 Hash Table，且當 k 增加時，搜尋時間呈現成長趨勢，因為 BST (std::map) 的搜尋時間複雜度是 O(log n)，當 n = 2^k 時，每次搜尋需要大約 k 次比較，所以當 k 增加時，每次搜尋需要的比較次數也會增加。BST 的節點在記憶體中是分散的，每次搜尋時需要跳轉到不同的記憶體位置，這會造成較多的 cache miss。而 hash table 因為 bucket array 是連續的，cache 效率較高。

第三題：Hash Table 標準函式庫來源及使用說明



unordered_map in C++ STL

Last Updated : 11 Oct, 2024

unordered_map is an associated container that stores elements formed by the combination of a key value and a mapped value. The key value is used to uniquely identify the element and the mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined. In simple terms, an ***unordered_map*** is like a data structure of dictionary type that stores elements in itself. It contains successive pairs (key, value), which allows fast retrieval of an individual element based on its unique key.

Internally unordered_map is implemented using Hash Table, the key provided to map is hashed into indices of a hash table which is why the performance of data structure depends on the hash function a lot but on average, the cost of **search, insert, and delete** from the hash table is O(1).

Note: *In the worst case, its time complexity can go from O(1) to O(n), especially for big prime numbers. In this situation, it is highly advisable to use a map instead to avoid getting a TLE(Time Limit Exceeded) error.*

資料來源：https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/

std::hash

Defined in header <bitset>
Defined in header <coroutine>
Defined in header <chrono> (since C++26)
Defined in header <filesystem>
Defined in header <functional>
Defined in header <memory>
Defined in header <optional>
Defined in header <stacktrace>
Defined in header <string>
Defined in header <string_view>
Defined in header <system_error>
Defined in header <thread>
Defined in header <typeindex>
Defined in header <variant>
Defined in header <vector>

```
template< class Key >
struct hash;
```
(since C++11)

The unordered associative containers `std::unordered_set`, `std::unordered_multiset`, `std::unordered_map`, `std::unordered_multimap` use specializations of the template `std::hash` as the default hash function.

資料來源：https://en.cppreference.com/w/cpp/utility/hash

Adding comparison test 使用證明：

```
    // 測試 unordered_map (hash table)
    unordered_map<int, int> hash_table;
    auto start = chrono::high_resolution_clock::now();

    for(int i = 0; i < n; i++) {
        hash_table[numbers[i]] = i;
    }

    auto end = chrono::high_resolution_clock::now();
    return chrono::duration<double>(end - start).count();
}
```

使用了 C++ 標準模板庫的 std::unordered_map，實現 Hash Table。儲鍵值對（Key-Value Pair）在這裡的型別為 int 和 int(鍵：用來查找和存取資料、值：與鍵相關聯的資料)。預設使用標準 Hash Function（std::hash）。預設使用鏈結法（Chaining）處理衝突。numbers[i] 是鍵，i 是值。std::unordered_map 計算鍵的哈希值，將其映射到內部的 buckets（桶）。如果該 bucket 內已經有其他元素（發生衝突），則使用鏈結法將新元素加入 bucket 的 Linked List。

Search comparison test 使用證明：

```
    // 建立 hash table 並記錄其狀態
    unordered_map<int, int> hash_table;

    // 設置足夠大的 bucket 數來減少 rehash
    hash_table.reserve(n);

    for(int i = 0; i < n; i++) {
        hash_table[numbers[i]] = i;
    }
```

std::unordered_map 是基於 Hash Table 實現的容器。它使用一個 Hash Function（預設為 std::hash）將鍵轉換為整數（Hash Value），並根據此 Hash Value 決

定鍵應該存放的 bucket（桶）。Hash Table 內部維護了一組 buckets，每個 bucket 用於存放 Hash Value 相同的鍵值對。當不同的鍵具有相同的 Hash Value 時，這些鍵會以鏈結法（Chaining）存儲在同一個 bucket 中。

運作過程：

1. 插入鍵值對時，std::unordered_map 使用 Hash Function 計算鍵的 Hash Value，並將其映射到對應的 bucket。
2. 如果該 bucket 已經包含元素，則通過鏈結法將新鍵值對添加到 bucket 中。
3. 為了提升插入性能，可以使用 reserve(n) 預先分配足夠的 buckets，以減少插入過程中的 rehash 操作（當 buckets 數量不足時，容器會自動擴容並重新分配所有元素，這是一個高成本操作）。

第四題：Balanced BST 標準函式庫來源及使用說明

C++ std::map 用法與範例

本篇將介紹如何使用 C++ std map 以及用法，C++ std::map 是一個關聯式容器，關聯式容器把鍵值和一個元素連繫起來，並使用該鍵值來尋找元素、插入元素和刪除元素等操作。

map 是有排序關聯式容器，即 map 容器中所有的元素都會根據元素對應的鍵值來排序，而鍵值 key 是唯一值，並不會出現同樣的鍵值 key，也就是說假設已經有一個 鍵值 key 存在 map 裡，當同樣的鍵值 key 再 insert 資料時，新的資料就會覆蓋掉原本 key 的資料。

map 的實作方式通常是用紅黑樹(red-black tree)實作的，這樣它可以保證可以在 O(log n) 時間內完成搜尋、插入、刪除，n為元素的數目。

資料來源： https://shengyu7697.github.io/std-map/

使用證明：

```
26      // 測試 map (BST)
27      map<int, int> bst;
28      auto start = chrono::high_resolution_clock::now();
29
30      for(int i = 0; i < n; i++) {
31          bst[numbers[i]] = i;
32      }
```

```
26      // 建立 BST
27      map<int, int> bst;
28      for(int i = 0; i < n; i++) {
29          bst[numbers[i]] = i;
30      }
```

使用了 std::map<int, int> 作為平衡二元搜尋樹 (BST, Binary Search Tree) 的實現。std::map 使用紅黑樹來存儲鍵值對 (Key-Value Pairs)。鍵：numbers[i]，即隨機生成的數字。值：i，即對應數字的索引。每次插入鍵值對 numbers[i] = i，先計算鍵 numbers[i] 在樹中的插入位置。如果鍵不存在於樹中，將其插入適當位置。插入後，紅黑樹自動調整以保持平衡。

第五題：實驗程式碼（含新增與搜尋的程式碼範例）與使用說明

1. Hash Table
   A. 新增範例

```
    unordered_map<string, int> hash_table;

    // 插入元素到 Hash Table
    hash_table["ntu"] = 112;
    hash_table["nycu"] = 113;
    hash_table["nthu"] = 114;
    hash_table["nccu"] = 119;

    // 顯示插入結果
    cout << "Hash Table Elements:" << endl;
    for (const auto& pair : hash_table) {
        cout << pair.first << " -> " << pair.second << endl;
    }
```

使用[]和賦值操作插入鍵值對。如果鍵不存在，會自動創建；如果已存在，則更新值。使用範圍 for 循環遍歷。每個元素是一個鍵值對(pair)，其中：pair.first: 鍵(string)、pair.second: 值(int)。

   B. 搜尋範例

```
    // 搜尋元素
    string key_to_search = "nccu";
    if (hash_table.find(key_to_search) != hash_table.end()) {
        cout << "Found: " << key_to_search << " -> " << hash_table[key_to_search] << endl;
    } else {
        cout << "Key '" << key_to_search << "' not found in the hash table." << endl;
    }

    // 搜尋另一個元素
    key_to_search = "ncku";
    if (hash_table.find(key_to_search) != hash_table.end()) {
        cout << "Found: " << key_to_search << " -> " << hash_table[key_to_search] << endl;
    } else {
        cout << "Key '" << key_to_search << "' not found in the hash table." << endl;
    }
```

使用 find() 方法搜尋，返回迭代器，若找到則指向元素，若未找到則返回 hash_table.end()，比較返回的迭代器是否為 end()，來判斷元素是否存在，如果找到，使用 hash_table[key_to_search] 獲取對應值。

C. 新增、搜尋範例結果

```
Hash Table Elements:
nccu -> 119
nthu -> 114
nycu -> 113
ntu -> 112
Found: nccu -> 119
Key 'ncku' not found in the hash table.
```

D. 圖一(新增)

```
// 用於生成隨機數的函數
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<int64_t> dis(1, (1LL << 30));

// 測試特定大小的插入操作
double test_insertion(int k) {
    int n = 1 << k;  // n = 2^k

    // 準備隨機數據
    vector<int> numbers(n);
    for(int i = 0; i < n; i++) {
        numbers[i] = dis(gen);
    }

    // 測試 unordered_map (hash table)
    unordered_map<int, int> hash_table;
    auto start = chrono::high_resolution_clock::now();

    for(int i = 0; i < n; i++) {
        hash_table[numbers[i]] = i;
    }

    auto end = chrono::high_resolution_clock::now();
    return chrono::duration<double>(end - start).count();
}
```

首先設置隨機數生成器： 使用 random_device 產生真隨機種子，餵給 mt19937 隨機數生成器。設定產生範圍在 1 到 2^30 之間的均勻分布隨機整數。

test_insertion 函數接收參數 k，決定要測試的數據量為 2^k 個。例如 k=10 時，會測試 1024 個數據的插入時間。

函數會先生成指定數量的隨機數存入 numbers 向量。接著創建雜湊表並開始計時，將隨機數當作 key、索引位置當作 value 插入。最後計算並返回整個插入過程所需的時間（以秒為單位）。

E. 圖二(搜尋)

```cpp
const int SEARCH_COUNT = 100000;

random_device rd;
mt19937 gen(rd());

double test_search(int k) {
    int n = 1 << k;
    uniform_int_distribution<int64_t> dis(1, (1LL << 30));

    // 準備插入的數據
    vector<int> numbers(n);
    for(int i = 0; i < n; i++) {
        numbers[i] = dis(gen);
    }

    // 建立 hash table 並記錄其狀態
    unordered_map<int, int> hash_table;

    // 設置足夠大的 bucket 數來減少 rehash
    hash_table.reserve(n);

    for(int i = 0; i < n; i++) {
        hash_table[numbers[i]] = i;
    }

    // 準備搜尋的數據
    vector<int> search_numbers(SEARCH_COUNT);
    for(int i = 0; i < SEARCH_COUNT; i++) {
        search_numbers[i] = dis(gen);
    }

    // 輸出 hash table 的狀態
    cout << "Bucket count: " << hash_table.bucket_count() << endl;
    cout << "Load factor: " << hash_table.load_factor() << endl;
    cout << "Max load factor: " << hash_table.max_load_factor() << endl;
```

```cpp
    // 進行搜尋並計時
    auto start = chrono::high_resolution_clock::now();

    int found_count = 0;
    for(int i = 0; i < SEARCH_COUNT; i++) {
        if(hash_table.find(search_numbers[i]) != hash_table.end()) {
            found_count++;
        }
    }

    auto end = chrono::high_resolution_clock::now();
    double time = chrono::duration<double>(end - start).count();

    cout << "Found: " << found_count << " items" << endl;

    return time;
}
```

初始設置：

- 固定進行 100000 次搜尋測試，使用 mt19937 生成高品質隨機數，生成範圍：1 到 2^30。

初始數據準備：

- 生成 2^k 個隨機數存入 numbers 向量，數量由參數 k 決定，例如 k=10 時生成 1024 個數據。

雜湊表建立：

- 使用 reserve() 預先分配空間，避免在插入過程中觸發 rehash，將 numbers 中的隨機數依序插入，建立 (隨機數->索引) 的映射。

搜尋數據準備：

- 生成 100000 個新的隨機數作為搜尋目標，存入 search_numbers 向量雜湊表狀態檢查：bucket_count：目前的桶數量、load_factor：現有元素數量/桶數量、max_load_factor：觸發 rehash 的載入因子閾值。

搜尋測試：

- 對每個 search_numbers 中的數字執行 find()，計數成功找到的數量（found_count），使用 chrono 精確計時整個搜尋過程。

結果輸出：

- 找到的元素數量，返回搜尋操作耗時（秒）。

2. Balanced Binary Search Tree

A. 新增範例

```
8   // 創建一個 std::map 作為 Balanced Binary Search Tree
9   map<string, int> bst;
10
11  // 插入元素到 BST
12  bst["ntu"] = 112;
13  bst["nycu"] = 113;
14  bst["nthu"] = 114;
15  bst["nccu"] = 119;
16
17  // 顯示 BST 元素
18  cout << "BST Elements (In Order):" << endl;
19  for (const auto& pair : bst) {
20      cout << pair.first << " -> " << pair.second << endl;
21  }
```

map 新增資料時，首先比較新 key 與根節點大小，若小於根節點，往左子樹尋找位置，若大於根節點，往右子樹尋找位置，重複此過程直到找到適當位置，插入新節點並調整紅黑樹平衡。

B. 搜尋範例

```
    // 搜尋元素
    string key_to_search = "nccu";
    auto it = bst.find(key_to_search);
    if (it != bst.end()) {
        cout << "Found: " << it->first << " -> " << it->second << endl;
    } else {
        cout << "Key '" << key_to_search << "' not found in the BST." << endl;
    }

    // 搜尋另一個元素
    key_to_search = "ncku";
    it = bst.find(key_to_search);
    if (it != bst.end()) {
        cout << "Found: " << it->first << " -> " << it->second << endl;
    } else {
        cout << "Key '" << key_to_search << "' not found in the BST." << endl;
    }
```

使用 find() 方法搜尋，從根節點開始比較，小於當前節點往左搜尋，大於當前節點往右搜尋，直到找到或到達葉節點。

代碼使用迭代器判斷結果，若找到，迭代器指向該元素，若未找到，返回 bst.end()。

C. 新增、搜尋範例結果

```
BST Elements (In Order):
nccu -> 119
nthu -> 114
ntu -> 112
nycu -> 113
Found: nccu -> 119
Key 'ncku' not found in the BST.
```

D. 圖一(新增)

```cpp
// 用於生成隨機數的函數
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<int64_t> dis(1, (1LL << 30));

// 測試特定大小的插入操作
double test_insertion(int k) {
    int n = 1 << k;  // n = 2^k

    // 準備隨機數據
    vector<int> numbers(n);
    for(int i = 0; i < n; i++) {
        numbers[i] = dis(gen);
    }

    // 測試 map (BST)
    map<int, int> bst;
    auto start = chrono::high_resolution_clock::now();

    for(int i = 0; i < n; i++) {
        bst[numbers[i]] = i;
    }

    auto end = chrono::high_resolution_clock::now();
    return chrono::duration<double>(end - start).count();
}
```

生成 2^k 個隨機數，使用 mt19937 生成器，範圍在 1 到 2^30 之間，存入 numbers 向量。

插入測試：創建 map 物件，記錄開始時間，將隨機數作為 key、索引作為 value 插入，每次插入需要 O(log n) 時間維護樹的平衡，計算並返回總插入時間（秒）。

E. 圖二(搜尋)

```cpp
const int SEARCH_COUNT = 100000;  // 搜尋次數為十萬次

random_device rd;
mt19937 gen(rd());

double test_search(int k) {
    int n = 1 << k;
    uniform_int_distribution<int64_t> dis(1, (1LL << 30));

    // 準備插入的數據
    vector<int> numbers(n);
    for(int i = 0; i < n; i++) {
        numbers[i] = dis(gen);
    }

    // 建立 BST
    map<int, int> bst;
    for(int i = 0; i < n; i++) {
        bst[numbers[i]] = i;
    }
```

```cpp
    // 準備搜尋的數據
    vector<int> search_numbers(SEARCH_COUNT);
    for(int i = 0; i < SEARCH_COUNT; i++) {
        search_numbers[i] = dis(gen);
    }

    // 進行搜尋並計時
    auto start = chrono::high_resolution_clock::now();

    int found_count = 0;
    for(int i = 0; i < SEARCH_COUNT; i++) {
        if(bst.find(search_numbers[i]) != bst.end()) {
            found_count++;
        }
    }

    auto end = chrono::high_resolution_clock::now();
    double time = chrono::duration<double>(end - start).count();

    // 輸出找到的數量，用於驗證
    cout << "Found: " << found_count << " items" << endl;

    return time;
}
```

參數設定：k 決定初始數據量：2^k 個數據，固定執行 100000 次搜尋測試，使用隨機數生成器 mt19937。數值範圍：1 到 2^30。

數據準備階段：

- 初始數據：生成 2^k 個隨機數存入 numbers 向量。
- 搜尋數據：生成 100000 個獨立的隨機數存入 search_numbers 向量。
- 這兩組數據互相獨立，模擬實際搜尋場景。

BST 建立：將 numbers 中的隨機數依序插入 map，每個節點儲存 (隨機數->索引) 的映射，map 會自動維護樹的平衡。

搜尋測試：對 100000 個測試數據執行 find()，每次搜尋從根節點開始，依大小比較決定往左或右子樹，時間複雜度：O(log n)，n 為樹的節點數，計數成功找到的元素數量。

時間計算：使用 chrono 高精度計時，只計算搜尋階段的時間，返回總耗時（秒）。

第六大題：心得、疑問、與遇到的困難

1. 想了解例如時做作業似的部分，會比較推薦用 map 還是 set 去實踐？
2. Balanced BST 的搜尋時間的推估，我能找出大概會成長多少比例，然後都透過乘上前一筆數據而得嗎？
3. 如果推得時間會差不多維持在穩定，除了找上下界，然後在界內隨機選取數據之外，還有其他的方法獲得數據嗎？