CPT-281 - Introduction to Data Structures with C++

Module 13

Sets, Maps, Hash Tables

Dayu Wang

- **Set**
  - → A set is a collection of objects.
  - → Characteristics: membership, unordered, unique
- **Set Operations**
  - → Membership testing
  - → Inserting elements into a set
  - → Removing elements from a set
  - → Union

    $\{1, 2, 5, 7\} \cup \{2, 3, 4, 5\} = \{1, 2, 3, 4, 5, 7\}$
  - → Intersection

    $\{1, 2, 5, 7\} \cap \{2, 3, 4, 5\} = \{2, 5\}$
  - → Difference

    $\{1, 2, 5, 7\} - \{2, 3, 4, 5\} = \{1, 7\}$
  - → Subset

    $\{1, 2, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\}$ = `true`

- **Set in C++ Standard Template Library**

  → **The `set` class in `<set>` library**

  - **A `set` is implemented by balanced binary search tree.**
  - **A `set` does not allow duplicate element keys.**
  - **To iterate over a `set`, you need to use iterators.**

  → **Common class-member functions of `set<Item_Type>`**

Function	Behavior
`size_t size() const;`	Returns the number of elements in the set.
`bool empty() const;`	Tests whether the set is empty.
`pair<iterator, bool> insert(const Item_Type&);`	Inserts an element into the set.
`void erase(iterator);`	Deletes the element at iterator position.
`size_t erase(const Item_Type&);`	Deletes an element.
`void clear();`	Deletes all the elements in the set.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first element in the set.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just after the last element in the set.
`iterator find(const Item_Type&) const;`	Tests whether an element is in the set.

- **Performance of Set**

  → **The `insert()` function**

  - **The function inserts a new key into the set.**
  - **The function returns a pair object containing an iterator and a boolean.**
  - **If insert was successful, the function returns an iterator on the newly inserted key and `true`.**
  - **If failed (the key is already in the set), the function returns the `end()` iterator and `false`.**
  - **Time complexity: $O(\log n)$**

  → **The `find()` function**

  - **The function searches for a key in the set.**
  - **If the key was found, the function returns an iterator on the found key.**
  - **If the key was not found, the function returns the `end()` iterator.**
  - **Time complexity: $O(\log n)$**

- **Unordered Set**

  → **The `unordered_set` class in `<unordered_set>` library**

  - **An `unordered_set` is implemented by hash table (will discuss later).**
  - **An `unordered_set` does not allow duplicate element keys.**
  - **To iterate over an `unordered_set`, you need to use iterators.**

  → **Common class-member functions of `unordered_set<Item_Type>`**

Function	Behavior
`size_t size() const;`	Returns the number of elements in the set.
`bool empty() const;`	Tests whether the set is empty.
`pair<iterator, bool> insert(const Item_Type&);`	Inserts an element into the set.
`iterator erase(iterator);`	Deletes the element at iterator position.
`size_t erase(const Item_Type&);`	Deletes an element.
`void clear();`	Deletes all the elements in the set.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first element in the set. There is **no** ~~guaranteed first element~~ in an unordered set.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just passed all the elements in the set.
`iterator find(const Item_Type&) const;`	Tests whether an element is in the set.

- **Performance of Unordered Set**

  → **The `insert()` function**

  - **The function inserts a new key into the set.**
  - **The function returns a pair object containing an iterator and a boolean.**
  - **If insert was successful, the function returns an iterator on the newly inserted key and `true`.**
  - **If failed (the key is already in the set), the function returns the `end()` iterator and `false`.**
  - **Time complexity: $O(1)$ (average)**

  → **The `find()` function**

  - **The function searches for a key in the set.**
  - **If the key was found, the function returns an iterator on the found key.**
  - **If the key was not found, the function returns the `end()` iterator.**
  - **Time complexity: $O(1)$ (average)**

- **[Sample Question] Write a function that returns all the unique values in a vector of integers.**
  - **The input vector is not sorted in any way.**
  - **Your function can return the unique values in any order.**

→ **We need to use a set to store all the "discovered values".**

Since order does not matter, we can just use `unordered_set`.

```
1  /** Finds all the unique values in a vector.
2      @param vec: input vector (not sorted in any way)
3      @return: a vector containing all the unique values in the input vector
4  */
5  vector<int> unique_values(const vector<int>& vec) {
6      unordered_set<int> us;  // Create an unordered_set to store all the discovered values.
7      vector<int> result;  // Stores the unique values.
8      for (size_t i = 0; i < vec.size(); i++) {
9          if (us.find(vec.at(i)) == us.end()) {
10             result.push_back(vec.at(i));
11             us.insert(vec.at(i));
12         }
13     }
14     return result;
15 }  // Time complexity: O(n) -> Average
```


- **[Sample Question] Set Mismatch**

→ **We have a vector of integers, which originally contains all the numbers from `1` to `n`. Unfortunately, due to some error, one of the numbers in the vector got duplicated to another number in the vector, which results in repetition of one number and loss of another number.**

→ **Find the number that occurs twice and the number that is missing.**

```
1  /** Finds the duplicate number and missing number in a vector.
2      @param vec: a vector containing one duplicate number and one missing number
3      @return: a pair containing the duplicate number and missing number (in that order)
4  */
5  pair<int, int> find_error_nums(const vector<int>& vec) {
6      set<int> s;
7      int d;  // Duplicate value
8      for (size_t i = 0; i < vec.size(); i++) {
9          if (s.find(vec.at(i)) != s.end()) { d = vec.at(i); }
10         else { s.insert(vec.at(i)); }
11     }
12     int n = 1;
13     for (set<int>::iterator it = s.begin(); it != s.end(); it++, n++) {
14         if (*it != n) { return make_pair(d, n); }
15     }
16     return make_pair(d, n);
17 }  // Time complexity: O(n * log(n))
```

- **Maps**

→ The map (also called associative array) is related to the set.

Mathematically, it is a set of ordered pairs whose elements are known as the key and the value.

→ The key is required to be unique, but the value is not necessarily unique.

Example: { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill) }

→ A map can be used to enable efficient storage and retrieval of information in a table.

Type of Item	Key	Value
University student	Student ID	Student name, address, major, GPA
Online store customer	E-mail address	Customer name, shopping cart
Inventory item	Part ID	Description, quantity, price

→ What are the differences between map and indexed collection (array)?

Maps can use user-specified keys to access the information stored in it.

Arrays can only use indices as keys to access the information stored in it.

→ A map is essentially a set of (key, value) pairs.

- **Maps in C++ Standard Template Library**

→ The `map` class in `<map>` library

- A `map` is implemented by balanced binary search tree.
- A `map` does not allow duplicate keys, but allow duplicate values.
- To iterate over a `map`, you need to use iterators.

→ Common class-member functions of `map<Key_Type, Value_Type>`

Function	Behavior
`size_t size() const;`	Returns the number of **entries** (key-value pairs) in the map.
`bool empty() const;`	Tests whether the map is empty.
`pair<iterator, bool> insert(const pair<Key_Type, Value_Type>&);`	Inserts an entry into the map.
`void erase(iterator);`	Deletes the entry at iterator position.
`size_t erase(const Key_Type&);`	Deletes an entry with the given key.
`void clear();`	Deletes all the entries in the map.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first entry in the map.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just after the last entry in the map.
`iterator find(const Key_Type&) const;`	Tests whether a key is in the map.

- **Performance of Map**

→ **The `insert()` function**

- The function inserts a new entry (key-value pair) into the map.
- The function returns a pair object containing an iterator and a boolean.
- If insert was successful, the function returns an iterator on the newly inserted entry and `true`.
- If failed (the key is already in the map), the function returns the `end()` iterator and `false`.
- Time complexity: $O(\log n)$

→ **The `find()` function**

- The function searches for a key (not a value) in the map.
- If the key was found, the function returns an iterator on the found entry (key-value pair).
- If the key was not found, the function returns the `end()` iterator.
- Time complexity: $O(\log n)$

- **Unordered Map**

→ **The `unordered_map` class in `<unordered_map>` library**

- An `unordered_map` is implemented by hash table (will discuss later).
- An `unordered_map` does not allow duplicate keys, but allow duplicate values.
- To iterate over an `unordered_map`, you need to use iterators.

→ **Common class-member functions of `unordered_map<Key_Type, Value_Type>`**

Function	Behavior
`size_t size() const;`	Returns the number of **entries** (key-value pairs) in the map.
`bool empty() const;`	Tests whether the map is empty.
`pair<iterator, bool> insert(const pair<Key_Type, Value_Type>&);`	Inserts an entry into the map.
`iterator erase(iterator);`	Deletes the entry at iterator position.
`size_t erase(const Key_Type&);`	Deletes an entry with the given key.
`void clear();`	Deletes all the entries in the map.
`iterator begin();` `const_iterator begin() const;`	Generates an iterator positioned at the first entry (**no** guarantee) in the map.
`iterator end();` `const_iterator end() const;`	Generates an iterator positioned just passed all the entries in the map.
`iterator find(const Key_Type&) const;`	Tests whether a key is in the map.

- **Performance of Unordered Map**
  - → **The `insert()` function**
    - **The function inserts a new entry (key-value pair) into the unordered map.**
    - **The function returns a pair object containing an iterator and a boolean.**
    - **If insert was successful, the function returns an iterator on the newly inserted entry and `true`.**
    - **If failed (the key is already in the map), the function returns the `end()` iterator and `false`.**
    - **Time complexity: $O(1)$ (average)**
  - → **The `find()` function**
    - **The function searches for a key (not a value) in the unordered map.**
    - **If the key was found, the function returns an iterator on the found entry (key-value pair).**
    - **If the key was not found, the function returns the `end()` iterator.**
    - **Time complexity: $O(1)$ (average)**

- **Member Accessing in Maps and Unordered Maps**
  - → **The `[]` operator in `map` and `unordered_map`**

    ```
    Value_Type& operator[](Key_Type&);
    ```

    - **It returns the value associated with the given key.**
    - **If the given key does not exist in the map, then a new entry is inserted into the map.**
    - **This works for both lvalue and rvalue.**
  - → **The `.at()` class-member function in `map` and `unordered_map`**

    ```
    Value_Type& at(const Key_Type&);
    const Value_Type& at(const Key_Type&) const;
    ```

    - **It returns the value associated with the given key.**
    - **If the given key does not exist in the map, then an exception will be thrown.**
    - **For lvalue, it returns a reference; for rvalue, it returns a `const` reference.**
- **Performance of Member Accessing**
  - → **In maps: $O(\log n)$**
  - → **In unordered maps: $O(1)$ (average)**

- **[Sample Question] Frequency Count**

  → Given a vector of integers, please find the element which has the largest frequency.

  [Example] In vector [4, 0, 9, 2, 9, 1, 2, 2, 4, 3, 2, 9, 0, 0, 1], value 2 appears 4 times, which is the highest frequency.

  → [Solution] Use a map to store the frequency of all the discovered values in the vector.

```
1  int most_frequent_value(const vector<int>& vec) {
2      // Store the frequency of each value into an unordered map.
3      unordered_map<int, unsigned int> m;
4      for (size_t i = 0; i < vec.size(); i++) {
5          if (m.find(vec.at(i)) == m.end()) { m[vec.at(i)] = 1; }
6          else { m[vec.at(i)]++; }
7      }
8      // Find the max value in the unordered map.
9      int result = 0;
10     unsigned int f = 0;
11     for (unordered_map<int, unsigned int>::iterator it = m.begin(); it != m.end(); it++) {
12         if (it->second > f) {
13             result = it->first;
14             f = it->second;
15         }
16     }
17     return result;
18 }  // Time complexity: O(n) -> Average
```


- **Hash Table**

  → The C++ library uses balanced binary search trees to implement sets and maps.

  → This gives access to items in $O(\log n)$.

  → Sets and maps can also be implemented using a data structure known as a hash table.

  - unordered_set
  - unordered_map

  → The idea behind hash tables is to access items directly through their key values.

  → With hash tables, accessing an item on average is $O(1)$.

- **Hash Code and Index Calculation**

  → Hashing ➡ transforming the item's key value into an integer value (hash code), which will be transformed into a table index.

- **Hash Codes for Unicode Characters**

  → There are 2^{16} (65536) Unicode characters. However, for a given file, let's assume that at most 100 different characters actually appear.

  → Rather than use a table with 65536 elements, it would be more sensible to store the items in a smaller table (say 200 elements).

→ We can calculate the array index for character `uni_char` (char type) as

```
int index = (int)uni_char % 200;
```

→ What is the problem with this approach?

Character	Unicode value	Hash Code
Í	205	5
ر	1605	5

→ This situation is known as collision.

→ Unfortunately, when we use hash table, we have to deal with collisions.

• **Functions for Generating Hash Codes**

→ In most applications, keys consist of strings of letters or digits rather than a single character (e.g., social security number).

→ Simple algorithms generate a lot of collisions.

For instance, generating a hash code based on summing the ASCII value of all characters in a string will generate the same index for words that contain the same letters such as `"sign"` and `"sing"`.

→ One algorithm that has shown good results uses the following formula:

$$s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + s_2 \times 31^{n-2} + \cdots + s_{n-1}$$

→ This formula is used by a hash function in the Java `string` class (C++?).

The hash code for `"cat"` is `'c'` × 31^2 + `'a'` × 31 + `'t'` = 98262

The hash code for `"act"` is `'a'` × 31^2 + `'c'` × 31 + `'t'` = 96402

→ An integer returned by this function cannot be unique since there are too many possible strings.

For instance, a string with 10 characters has 62^{10} possibilities to generate the same hash code.

• **Qualities of the string hash function**

→ The probability for two strings have the same hash code is relatively small.

→ Each letter affects the value.

→ The order of each letter affects the value.

→ The values tend to be spread well over the integers.

That is, every hash value in the output range should be generated with roughly the same probability.

- **Good Qualities of a Hash Function**
  - → Spread values evenly.
  - → Cheap to compute.
- **Ways to Resolve Collision**
  - → Open addressing (linear probing)
  - → Chaining
- **Open Addressing (Linear Probing)**
  - → Algorithm - Find an item with a specific hash key.

    Compute `index` as `hash_fcn() % table.size()`

    `if table[index] == NULL`

    item is not in the table.

    `else if table[index]` matches item

    The item is in the table.

    `else`

    Continue to search the table by incrementing the `index` until either the item is found or a `NULL` entry is found.

- **Search Termination**
  - → Ways to obtain proper termination

    Stop when you come back to your starting point.

    Stop after probing N slots, where N is table size.

    Ensure table never full.

    Reallocate when occupancy exceeds threshold.

0	"John"	1
1	"Sam"	2
2	"Pete"	4
3	"Harry"	0
4	"Tom"	0

- **Open Addressing (Linear Probing) Example**

Key	hash_fcn()	hash_fcn() % 5	hash_fcn() % 11
"Tom"	84274	4	3
"John"	2129869	4	5
"Harry"	69496448	3	10
"Sam"	82879	4	5
"Pete"	2484038	3	7

  - → Increasing the table size reduces the probability of a collision.

0	NULL	
1	NULL	
2	NULL	
3	"Tom"	0
4	NULL	
5	"John"	0
6	"Sam"	1
7	"Pete"	0
8	NULL	
9	NULL	
10	"Harry"	0

- **Traversing a Hash Table**

  → You cannot traverse a hash table in a meaningful way.

0	"John"
1	"Sam"
2	"Pete"
3	"Harry"
4	"Tom"

"John", "Sam", "Pete", "Harry", "Tom"

0	NULL
1	NULL
2	NULL
3	"Tom"
4	NULL
5	"John"
6	"Sam"
7	"Pete"
8	NULL
9	NULL
10	"Harry"

"Tom", "John", "Sam", "Pete", "Harry"

- **Deleting an Item Using Open Addressing**

  → We cannot delete an item by setting it to NULL.

  → Instead, we mark it as DELETED (by storing a dummy value).

  → This might lead to search inefficiency particularly if there are many deletions.

  → If the table is almost full and most of the items are deleted, you will have $O(n)$ performance when searching for the few items remaining in the table.

Item Index	Table index	Key
4	0	"John"
	1	NULL
	2	NULL
	3	NULL
4	4	DELETED

"John" is not accessible anymore.

- **Reducing Collisions by Expanding the Table Size**

  → Use a prime number for the size of the table.

  → The probability of a collision is proportional to how full the table is. Hence, when the hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted.

Item Index	Table index	Key
4	0	"John"
4	1	"Sam"
3	2	"Pete"
3	3	"Harry"
4	4	"Tom"

Item Index	Table index	Key
4	0	"John"
4	1	"Sam"
3	2	"Pete"
3	3	"Harry"
4	4	"Tom"
	5	NULL
	6	NULL
	7	NULL
	8	NULL
	9	NULL
	10	NULL

Simply copying the values in the original array to the new array does not work.

→ Rehashing means inserting the items all over again one by one.

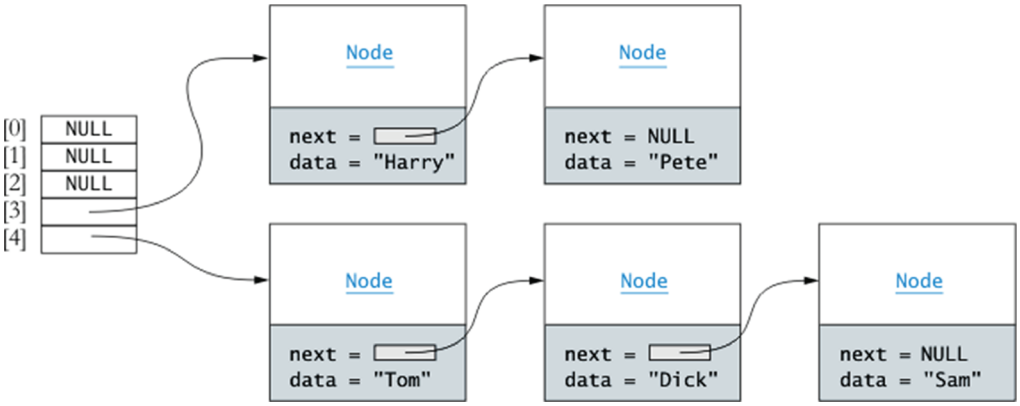
- **Problem with Linear Probing**

  → Linear probing forms clusters of keys in the table causing longer search chains.

  Insert [5, 6, 5, 6, 7].

Index	Value	
0		
1		
2		
3		
4		
5	1st item with code 5	0
6	1st item with code 6	0
7	2nd item with code 5	2
8	2nd item with code 6	2
9	1st item with code 7	2
10		

- **Chaining**
  - → Each table element references a linked list that contains all the items that hash to the same index.



  - → To resolve collisions, you traverse the referenced linked list.
- **Advantages**
  - → Only items with the same hashed index will be examined.
  - → You can store more elements in the table than the table size.
  - → To delete an item, simply remove it from the list.

- **Performance of Hash Tables**
  - → Load factor

    L = [number of filled cells] / [size of the table]

    The lower L, the better performance.
  - → 0 ≤ L < 1 for linear probing tables.
- **Expected Number of Comparisons for Finding an Item in a Hash Table**
  - → Using open addressing with linear probing

$$c = \frac{1}{2}\left(1 + \frac{1}{1-L}\right)$$

  - → Using chaining

$$c = 1 + \frac{L}{2}$$

L	Probes with Linear Probing	Probes with Chaining
0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

- **Hash Table, Ordered Vector, Binary Search Tree**

Operation	Hash Table	Ordered Vector	BST (Unbalanced)
Insertion (on average)	$O(1)$	$O(n)$	$O(\log n)$
Search (on average)	$O(1)$	$O(\log n)$	$O(\log n)$
Insertion (worst)	$O(n)$	$O(n)$	$O(n)$
Search (worst)	$O(n)$	$O(\log n)$	$O(n)$

→ **The overall performance of hash table is better than binary search tree if L < 0.75.**

- **Usage of Hash Table and Binary Search Tree**

Data Structure	Usage
Binary Search Tree	• Quick search, insertion, deletion. • Also good when you want queries (items that start with, greater than, less than, range, etc.). • Good when you want to retrieve a sorted list of items.
Hash Table (Hash maps)	• Fast access if key is known. Fast insertion on average. • Allow you to link objects by means of ids rather than references. • Otherwise, search is not fast.