Data Structures & Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 12

Hash Table
Dynamic Length

Course Roadmap

Part 2: Non-Linear Data Structures

Lecture 8: Binary Tree

Lecture 9: Binary Search Tree

Lecture 10: Self Balancing Binary Search Tree

Lecture 11: Binary Heap Tree

Lecture 12: Hash Table

Lecture 13: Graph

Lecture 14: STL in C++ (Non-Linear Data Structures)

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Rehashing

2- Rehashing Separate Chaining

3- Rehashing Double Hashing

4- Time Complexity & Space Complexity


Let's
START UP

Lecture Agenda

Section 1: Introduction to Rehashing

Section 2: Rehashing Separate Chaining

Section 3: Rehashing Double Hashing

Section 4: Time Complexity & Space Complexity

Introduction to Rehashing

- As the name suggests, rehashing means hashing again. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.
- Increases the size of the hash table when load factor becomes too high (defined by a cutoff). Anticipating that prob(collisions) would become higher. Typically expand the table to twice its size (but still prime) Need to reinsert all existing elements into new hash table. We make rehash: When load factor reaches some threshold (e.g, $\lambda \geq 0.5$)

- Why rehashing?

- Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of O(1). Hence, rehash must be done, increasing the size of the bucketArray so as to reduce the load factor and the time complexity.

Introduction to Rehashing

- Rehashing can be done as follows:

➢ For each addition of a new entry to the map, check the load factor. If it's greater than its pre-defined value (or default value of 0.75 if not given), then Rehash. For Rehash, make a new array of double the previous size and make it the new bucketarray. Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

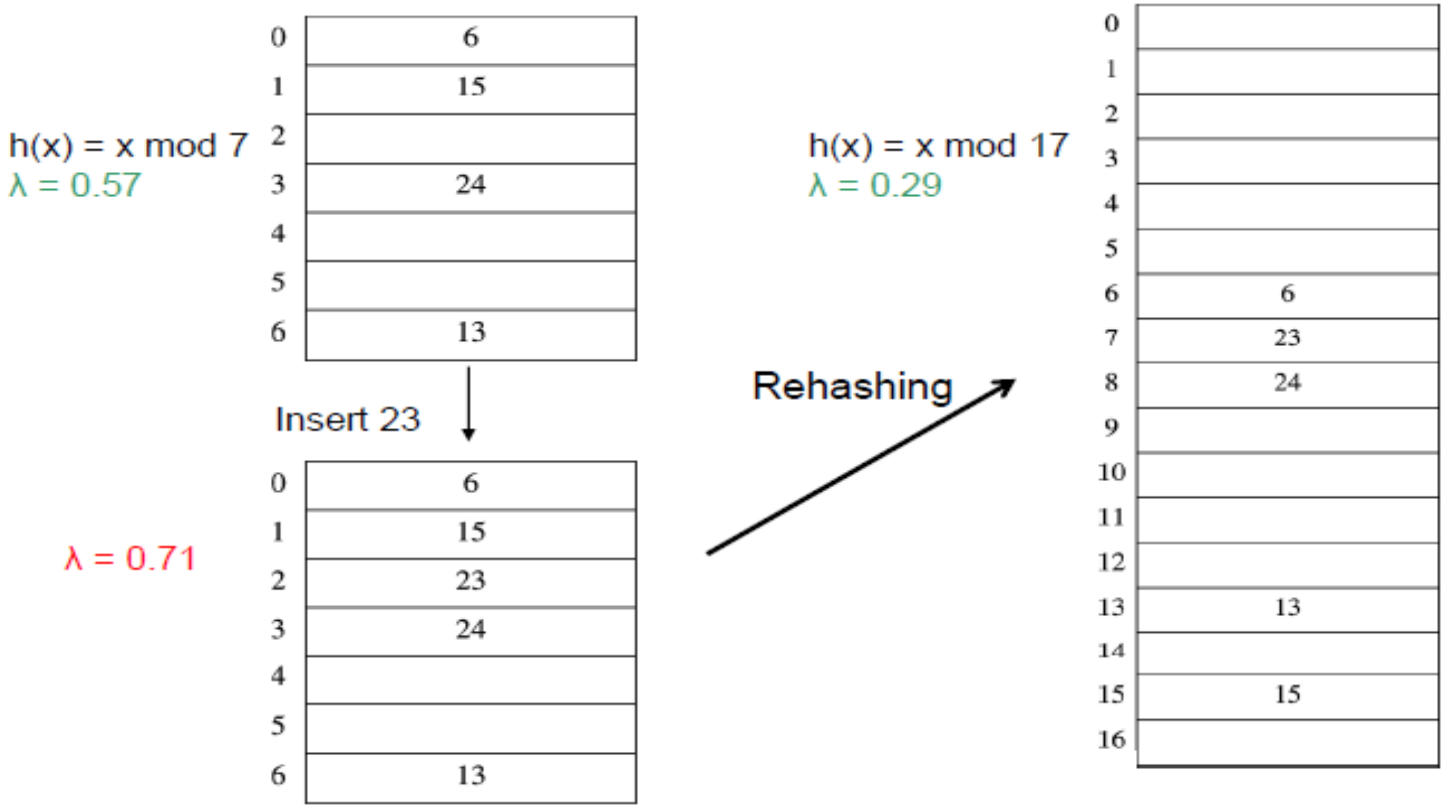
➢ When an insert is made such that the number of entries in a hash table exceeds the product of the load factor and the current capacity then the hash table will need to be rehashed. Rehashing includes increasing the size of the underlying data structure and mapping existing items to new bucket locations. In some implementations, if the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

➢ To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table - followed by a rehash - when items are deleted. From the point of space-time tradeoffs, this operation is similar to the deallocation in dynamic arrays.

Introduction to Rehashing

- A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold r_max. Then a new larger table is allocated, each entry is removed from the old table, and inserted into the new table. When all entries have been removed from the old table then the old table is returned to the free storage pool. Likewise, when the load factor falls below a second threshold r_min, all entries are moved to a new smaller table.
- For hash tables that shrink and grow frequently, the resizing downward can be skipped entirely. In this case, the table size is proportional to the maximum number of entries that ever were in the hash table at one time, rather than the current number. The disadvantage is that memory usage will be higher, and thus cache behavior may be worse. For best control, a "shrink-to-fit" operation can be provided that does this only on request.
- If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries n and of the number m of operations performed. For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If m elements are inserted into that table, the total number of extra re-insertions that occur in all dynamic resizings of the table is at most m - 1. In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

Introduction to Rehashing



Introduction to Rehashing

- Alternatives to all-at-once rehashing

➢ Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to perform the resizing gradually.

➢ Disk-based hash tables almost always use some alternative to all-at-once rehashing, since the cost of rebuilding the entire table on disk would be too high.

- Incremental resizing

- One alternative to enlarging the table all at once is to perform the rehashing gradually:
- During the resize, allocate the new hash table, but keep the old table unchanged.
- In each lookup or delete operation, check both tables.
- Perform insertion operations only in the new table.
- At each insertion also move r elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

Lecture Agenda

- ✓ Section 1: Introduction to Rehashing
- **Section 2: Rehashing Separate Chaining**
- Section 3: Rehashing Double Hashing
- Section 4: Time Complexity & Space Complexity

Rehashing Separate Chaining (Open Hashing)

- Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.
- The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries N is much higher than the number of slots.
- For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number N of entries in the table.
- Rehashing Separate Chaining: after each insertion operation and deletion operation check if the laod factor need to update the size of the hash table to rehash it.

Separate Chaining (Open Hashing)

- The idea is to keep a list of all elements that hash to the same value. The array elements are pointers to the first nodes of the lists. A new item is inserted to the front of the list.

- Advantages:

- Better space utilization for large items.
- Simple collision handling: searching linked list.
- Overflow: we can store more items than the hash table size.
- Deletion is quick and easy: deletion from the linked list.

- Disadvantages:

- Cache performance of chaining is not good as keys are stored using linked list.
- cache performance as everything is stored in same table.
- Wastage of Space (Some Parts of hash table are never used)
- If the chain becomes long, then search time can become O(n) in worst case.
- Uses extra space for links.

Separate Chaining Algorithms

- **Initialization:**

All entries are set to NULL

- **Search:**

Locate the cell using hash function.

sequential search on the linked list in that cell.

- **Insertion:**

Locate the cell using hash function.

(If the item does not exist) insert it as the first item in the list.

Check the current load factor to determine if the hash table need to be updated.

- **Deletion:**

Locate the cell using hash function.

(If the item exists) Delete the item from the linked list.

Check the current load factor to determine if the hash table need to be updated.

Rehashing (Separate Chaining)

- Initialize a global hash table with dynamic length

```cpp
#include <bits/stdc++.h>
using namespace std;

// A single linked list node
struct node {
    string data;
    node* next;
};
// Initialize a hash table with dynamic length
int n;
int capacity;
// Initialize a global array pointer for heads
node** head;

// Initialize the load_factor
const float load_factor = 0.5;
```

Rehashing-Separate-Chaining.cpp

Hash Function - Rehashing (Separate Chaining)

- These functions calculates the hash value of the given key with type string

```
// This function calculates the hash value of the given key with type string
int hash_function_str_complex(string key) {
    int res = 0;
    for (int i = 0 ; i < key.size() ; i++)
        res = (res + 37LL * key[i]) % capacity;
    return res;
}
```

- In the Main function:

```
capacity = 2;
head = new node*[capacity];
for (int i = 0 ; i < capacity ; i++)
    head[i] = NULL;
```

Rehashing-Separate-Chaining.cpp

Insertion Operation - Rehashing (Separate Chaining)

```
// This function inserts a node at the begin of the linked list
node* insert_begin(node* curr_head, string key) {
    // allocate new node and put it's data
    node* new_node = new node();
    new_node->data = key;
    // check if the linked list is empty
    if (curr_head == NULL) {
        curr_head = new_node;
    }
    // otherwise insert the new node in the begin of the linked list
    else {
        // set next of the new node to be the head
        new_node->next = curr_head;
        // set the new node as a head
        curr_head = new_node;
    }
    return curr_head;
}
```

Rehashing-Separate-Chaining.cpp

Reinsert Method - Rehashing (Separate Chaining)

```
// This function reinserts the elements in the hash table
void reinsert_items(node** temp, int old_capacity) {
    // loop on all items in the hash table to reinsert them again
    for (int i = 0 ; i < old_capacity ; i++) {
        // check if the current item empty
        if (head[i] == NULL)
            continue;
        // loop till insert all nodes
        node* curr = head[i];
        while (curr != NULL) {
            // calculate the hash value of the given key
            int idx = hash_function_str_complex(curr->data);
            // insert the new element
            temp[idx] = insert_begin(temp[idx], curr->data);
            curr = curr->next;
        }
    }
}
```

Rehashing-Separate-Chaining.cpp

Delete Linked List - Rehashing (Separate Chaining)

```cpp
// This function deletes all nodes in the linked list in iterative way
void delete_linked_list(node* curr_head) {
    node* curr = curr_head;
    while (curr != NULL) {
        curr_head = curr_head->next;
        delete(curr);
        curr = curr_head;
    }
}
```

Rehashing-Separate-Chaining.cpp

Reserve Method - Rehashing (Separate Chaining)

```
// This function updates the capacity of the hash table
void reserve(int new_capacity) {
    // Initialize a new hash table with the new capacity
    node** temp = new node*[new_capacity];
    for (int i = 0 ; i < new_capacity ; i++)
        temp[i] = NULL;
    // store the old capacity to use it in reinserting the items
    int old_capacity = capacity;
    // set the current capacity of the hash table to be the new capacity
    capacity = new_capacity;
    // copy the elements in the current hash table to the new hash table
    reinsert_items(temp, old_capacity);
    // delete the old hash table
    for (int i = 0 ; i < old_capacity ; i++)
        delete_linked_list(head[i]);
    delete head;
    // set the temp hash table with new capacity to be the hash table
    head = temp;
}
```

Rehashing-Separate-Chaining.cpp

Next Prime Method - Rehashing (Separate Chaining)

```cpp
// This function find the next prime of the given number
int get_next_prime(int x) {
    // loop for maximum 150 numbers after x to find a prime one
    for (int i = x + 1 ; i < x + 150 ; i++) {
        bool is_prime = true;
        // loop to check if the current number is prime
        for (int j = 2 ; j * j <= i ; j++) {
            if (i % j == 0)
                is_prime = false;
        }
        // if the current number is prime return it
        if (is_prime)
            return i;
    }
    // return the given number
    return x;
}
```

Rehashing-Separate-Chaining.cpp

Search Operation - Rehashing (Separate Chaining)

```
// This function searches for a node in the linked list
bool search_node(node* curr_head, string key) {
    // iterate on the nodes till reach the last node in the linked list
    node* curr = curr_head;
    while (curr != NULL) {
        // check if the given key exists in the linked list
        if (curr->data == key)
            return true;
        curr = curr->next;
    }
    return false;
}
// This function searches for an element in the hash table
bool search_item(string key) {
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // return whether the current position in the hash table contain the given key
    return search_node(head[idx], key);
}
```

Rehashing-Separate-Chaining.cpp

Insertion Operation - Rehashing (Separate Chaining)

```
// This function inserts an element in the hash table
void insert_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == true)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // insert the new element
    head[idx] = insert_begin(head[idx], key);
    // update the size of the hash table
    n = n + 1;
    // check if we need to update the capacity of the hash table
    if (1.0 * n / capacity >= load_factor)
        reserve(get_next_prime(2 * capacity));
}
```

Rehashing-Separate-Chaining.cpp

Deletion Operation - Rehashing (Separate Chaining)

```
// This function require a node to delete the node after it in the linked list
node* delete_node(node* curr_head, string key) {
    // check if the linked list is empty
    if (curr_head == NULL)
        return curr_head;
    // check if the first node in the list is the deleted node
    if (curr_head->data == key) {
        // get the node which it will be deleted
        node* temp_node = curr_head;
        // shift the head to be the next node
        curr_head = curr_head->next;
        // delete the temp node
        delete(temp_node);
    }
    // otherwise loop till reach the deleted node
    else {
```

Rehashing-Separate-Chaining.cpp

Deletion Operation - Rehashing (Separate Chaining)

```
    // otherwise loop till reach the deleted node
    else {
        // get the deleted node and the prev node of it in the linked list
        node* curr = curr_head;
        node* prev = NULL;
        while (curr != NULL && curr->data != key) {
            prev = curr;
            curr = curr->next;
        }
        // jump the deleted node
        prev->next = curr->next;
        // delete the node which selected
        delete(curr);
    }
    return curr_head;
}
```

Rehashing-Separate-Chaining.cpp

Deletion Operation - Rehashing (Separate Chaining)

```
// This function deletes an element at the given index in the hash table
void delete_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == false)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // delete the given key from it's list
    head[idx] = delete_node(head[idx], key);
    // update the size of the hash table
    n = n - 1;
    // check if we need to update the capacity of the min-heap
    if (n < capacity / 2)
        reserve(get_next_prime(capacity / 2));
}
```

Rehashing-Separate-Chaining.cpp

Traversal Operation - Rehashing (Separate Chaining)

```
// This function prints the contents of the linked list
void print_linked_list(node* curr_head) {
    // print the data nodes starting from head till reach the last node
    node* curr = curr_head;
    while (curr != NULL) {
        cout << curr->data << "\n";
        curr = curr->next;
    }
}
// This function prints the contents of the hash table
void print_hash_table() {
    // loop to print the elements in the hash table
    for (int i = 0 ; i < capacity ; i++) {
        if (head[i] == NULL)
            continue;
        cout << "index " << i << ":\n";
        print_linked_list(head[i]);
    }
}
```

Rehashing-Separate-Chaining.cpp

Functionality Testing - Rehashing (Separate Chaining)

- In the Main function:

```
cout << "Hash Table capacity: " << capacity << '\n';
cout << "Hash Table size: " << n << '\n';
cout << "-------------------------------------------------\n";
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 75 ; j++) {
        insert_item(item);
        item += i;
    }
    cout << "Hash Table capacity: " << capacity << '\n';
    cout << "Hash Table size: " << n << '\n';
    cout << "-------------------------------------------------\n";
}
```

Rehashing-Separate-Chaining.cpp

Functionality Testing - Rehashing (Separate Chaining)

- In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 75 ; j++) {
        if (search_item(item) == false)
            cout << item << " not found\n";
        item += i;
    }
}

//print_hash_table();
cout << "-------------------------------------------------------\n";
```

Rehashing-Separate-Chaining.cpp

Functionality Testing - Rehashing (Separate Chaining)

- In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 75 ; j++) {
        if (j < 10) {
            item += i;
            continue;
        }
        delete_item(item);
        item += i;
    }
    cout << "Hash Table capacity: " << capacity << '\n';
    cout << "Hash Table size: " << n << '\n';
    cout << "-----------------------------------------------------\n";
}
```

Rehashing-Separate-Chaining.cpp

Functionality Testing - Rehashing (Separate Chaining)

- In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 10 ; j++) {
        if (search_item(item) == false)
            cout << item << " not found\n";
        item += i;
    }
}

//print_hash_table();
cout << "-------------------------------------------------\n";
```

Rehashing-Separate-Chaining.cpp

Lecture Agenda

- ✔ Section 1: Introduction to Rehashing
- ✔ Section 2: Rehashing Separate Chaining
- Section 3: Rehashing Double Hashing
- Section 4: Time Complexity & Space Complexity

Rehashing (Double Hashing) (Close Hashing)

- In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.
- The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes. When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.
- Separate chaining has a disadvantage of using lists. Requires the implementation of a second data Structure. In an open addressing hashing system, all the data go inside the table. Generally the load factor should be below 0.5 If a collision occurs, alternative cells are tried until an empty cell is found, Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Open Addressing Algorithms

- **Initialization:**

All entries are set to NULL

- **Search:** Locate the cell using hash function.

Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

- **Insertion:** Locate the cell using hash function.

Keep probing until an empty slot or a deleted slot is found. Once an empty slot is found, insert k.

Check the current load factor to determine if the hash table need to be updated.

- **Deletion:** Locate the cell using hash function.

Keep probing until k is found. Once the item is found, mark it as a deleted slot.

Check the current load factor to determine if the hash table need to be updated.

- If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".
- Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.
- Cells h0(x), h1(x), h2(x), ... are tried in succession where hi(x) = (hash(x) + f(i)) mod TableSize.

Rehashing (Double Hashing)

- Initialize a global hash table with dynamic length

```cpp
#include <bits/stdc++.h>
using namespace std;

// Initialize a hash table with dynamic length
int n;
int capacity;
int prime_capacity;
string* arr;

// Initialize the load_factor
const float load_factor = 0.5;
```

- In the Main function:

```cpp
capacity = 3;
prime_capacity = 2;
arr = new string[capacity];
```

Rehashing-Double-Hashing.cpp

Hash Function - Rehashing (Double Hashing)

- These functions calculates the hash value of the given key with type string

```cpp
// This function calculates the hash value of the given key with type string
int hash_function_str_complex(string key) {
    int res = 0;
    for (int i = 0 ; i < key.size() ; i++)
        res = (res + 37LL * key[i]) % capacity;
    return res;
}

// This function calculates the hash value of the given hash value
int hash_function_secondary(int hash_idx) {
    return prime_capacity - hash_idx % prime_capacity;
}
```

Rehashing-Double-Hashing.cpp

Reinsert Method - Rehashing (Double Hashing)

```
// This function reinserts the elements in the hash table
void reinsert_items(string* temp, int old_capacity) {
    // loop on all items in the hash table to reinsert them again
    for (int i = 0 ; i < old_capacity ; i++) {
        // check if the current item empty
        if (arr[i] == "" || arr[i] == "#")
            continue;
        // calculate the hash value of the given key
        int idx = hash_function_str_complex(arr[i]);
        // calculate the second hash value of the given hash value
        int step = hash_function_secondary(idx);
        // loop till find an empty position
        int j = 0;
        while (temp[(idx + j*step) % capacity] != "")
            j++;
        // insert the new element
        temp[(idx + j*step) % capacity] = arr[i];
    }
}
```

Rehashing-Double-Hashing.cpp

Reserve Method - Rehashing (Double Hashing)

```
// This function updates the capacity of the hash table
void reserve(int new_capacity) {
    // Initialize a new hash table with the new capacity
    string* temp = new string[new_capacity];
    // store the old capacity to use it in reinserting the items
    int old_capacity = capacity;
    // set the current capacity of the hash table to be the new capacity
    capacity = new_capacity;
    // copy the elements in the current hash table to the new hash table
    reinsert_items(temp, old_capacity);
    // delete the old hash table
    delete[] arr;
    // set the temp hash table with new capacity to be the hash table
    arr = temp;
}
```

Rehashing-Double-Hashing.cpp

Next Prime Method - Rehashing (Double Hashing)

```cpp
// This function find the next prime of the given number
int get_next_prime(int x) {
    // loop for maximum 150 numbers after x to find a prime one
    for (int i = x + 1 ; i < x + 150 ; i++) {
        bool is_prime = true;
        // loop to check if the current number is prime
        for (int j = 2 ; j * j <= i ; j++) {
            if (i % j == 0)
                is_prime = false;
        }
        // if the current number is prime return it
        if (is_prime)
            return i;
    }
    // return the given number
    return x;
}
```

Rehashing-Double-Hashing.cpp

Search Operation - Rehashing (Double Hashing)

```
// This function searches for an element in the hash table
bool search_item(string key) {
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // calculate the second hash value of the given hash value
    int step = hash_function_secondary(idx);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i*step) % capacity] != key &&
           arr[(idx + i*step) % capacity] != "")
        i++;
    // return whether the current position in the hash table
    // contain the given key
    return arr[(idx + i*step) % capacity] == key;
}
```

Rehashing-Double-Hashing.cpp

Insertion Operation - Rehashing (Double Hashing)

```
// This function inserts an element in the hash table
void insert_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == true)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // calculate the second hash value of the given hash value
    int step = hash_function_secondary(idx);
    // loop till find an empty position
    int i = 0;
    while (arr[(idx + i*step) % capacity] != "" &&
           arr[(idx + i*step) % capacity] != "#")
        i++;
    // insert the new element
    arr[(idx + i*step) % capacity] = key;
    // update the size of the hash table
    n = n + 1;
    // check if we need to update the capacity of the hash table
    if (1.0 * n / capacity >= load_factor) {
        prime_capacity = get_next_prime(2 * capacity);
        reserve(get_next_prime(prime_capacity));
    }
}
```

Rehashing-Double-Hashing.cpp

Deletion Operation - Rehashing (Double Hashing)

```
// This function deletes an element at the given index in the hash table
void delete_item(string key) {
    // check if the given key exists or not
    if (search_item(key) == false)
        return;
    // calculate the hash value of the given key
    int idx = hash_function_str_complex(key);
    // calculate the second hash value of the given hash value
    int step = hash_function_secondary(idx);
    // loop till find the given key
    int i = 0;
    while (arr[(idx + i*step) % capacity] != key)
        i++;
    // set the position of the deleted element as a hash sign
    arr[(idx + i*step) % capacity] = "#";
    // update the size of the hash table
    n = n - 1;
    // check if we need to update the capacity of the min-heap
    if (n < capacity / 2) {
        prime_capacity = get_next_prime(capacity / 2);
        reserve(get_next_prime(prime_capacity));
    }
}
```

Rehashing-Double-Hashing.cpp

Traversal Operation - Rehashing (Double Hashing)

```
// This function prints the contents of the hash table
void print_hash_table() {
    // loop to print the elements in the hash table
    for (int i = 0 ; i < capacity ; i++) {
        if (arr[i] == "" || arr[i] == "#")
            continue;
        cout << "index " << i << ": " << arr[i] << '\n';
    }
}
```

Rehashing-Double-Hashing.cpp

Functionality Testing - Rehashing (Double Hashing)

- In the Main function:

```
cout << "Hash Table capacity: " << capacity << '\n';
cout << "Hash Table prime_capacity: " << prime_capacity << '\n';
cout << "Hash Table size: " << n << '\n';
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 75 ; j++) {
        insert_item(item);
        item += i;
    }
    cout << "Hash Table capacity: " << capacity << '\n';
    cout << "Hash Table prime_capacity: " << prime_capacity << '\n';
    cout << "Hash Table size: " << n << '\n';
    cout << "------------------------------------------------\n";
}
```

Rehashing-Double-Hashing.cpp

Functionality Testing - Rehashing (Double Hashing)

- In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 75 ; j++) {
        if (search_item(item) == false)
            cout << item << " not found\n";
        item += i;
    }
}

// print_hash_table();
cout << "--------------------------------------------------\n";
```

Rehashing-Double-Hashing.cpp

Functionality Testing - Rehashing (Double Hashing)

- In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 75 ; j++) {
        if (j < 10) {
            item += i;
            continue;
        }
        delete_item(item);
        item += i;
    }
    cout << "Hash Table capacity: " << capacity << '\n';
    cout << "Hash Table prime_capacity: " << prime_capacity << '\n';
    cout << "Hash Table size: " << n << '\n';
    cout << "----------------------------------------------------\n";
}
```

Rehashing-Double-Hashing.cpp

Functionality Testing - Rehashing (Double Hashing)

- In the Main function:

```
for (char i = 'a' ; i <= 'm' ; i++) {
    string item = "";
    item += i;
    item += i;
    item += i;
    for (int j = 0 ; j < 10 ; j++) {
        if (search_item(item) == false)
            cout << item << " not found\n";
        item += i;
    }
}

// print_hash_table();
cout << "--------------------------------------------------\n";
```

Rehashing-Double-Hashing.cpp

Lecture Agenda

- ✔ Section 1: Introduction to Rehashing
- ✔ Section 2: Rehashing Separate Chaining
- ✔ Section 3: Rehashing Double Hashing
- Section 4: Time Complexity & Space Complexity

Time Complexity & Space Complexity

- Load factor λ of a hash table T is defined as follows:
  - N = number of elements in T (current size)
  - M = size of hash table T (total size)
  - λ = N/M (load factor) λ is the average length of chain
  - Unsuccessful search time: O(λ), Successful search time: O(λ/2)
- For the first step, time taken depends on the K and the hash function. For example, if the key is a string "abcd", then it's hash function may depend on the length of the string. But for very large values of n, the number of entries into the map, length of the keys is almost negligible in comparison to n so hash computation can be considered to take place in constant time, i.e, O(1).
- For the second step, traversal of the list of K-V pairs present at that index needs to be done. For this, the worst case may be that all the n entries are at the same index. So, time complexity would be O(n). But, enough research has been done to make hash functions uniformly distribute the keys in the array so this almost never happens.
- So, on an average, if there are n entries and b is the size of the array there would be n/b entries on each index. This value n/b is called the load factor that represents the load that is there on our map. This Load Factor needs to be kept low, so that number of entries at one index is less and so is the complexity almost constant, i.e., O(1).

Time Complexity & Space Complexity

- Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

LOAD FACTOR	SUCCESSFUL			UNSUCCESSFUL		
	LINEAR	i + bkey	DOUBLE	LINEAR	i + bkey	DOUBLE
25%	1.17	1.16	1.15	1.39	1.37	1.33
50%	1.50	1.44	1.39	2.50	2.19	2.00
75%	2.50	2.01	1.85	8.50	4.64	4.00
90%	5.50	2.85	2.56	50.50	11.40	10.00
95%	10.50	3.52	3.15	200.50	22.04	20.00

Separate Chaining vs. Open Addressing

Separate Chaining

1) Chaining is Simpler to implement.
2) In chaining, Hash table never fills up, we can always add more elements to chain.
3) Chaining is Less sensitive to the hash function or load factors.
4) Chaining is mostly used when it is unknown how many and how frequently keys

may be inserted or deleted.

5) Cache performance of chaining is not good as keys are stored using linked list.
6) Wastage of Space (Some Parts of hash table in chaining are never used).
7) Chaining uses extra space for links.

Open Addressing

1) Open Addressing requires more computation.
2) In open addressing, table may become full.
3) Open addressing requires extra care for to avoid clustering and load factor.
4) Open addressing is used when the frequency and number of keys is known.
5) Open addressing provides better cache performance as everything is stored in same table.
6) In Open addressing, a slot can be used even if an input doesn't map to it.
7) No links in Open addressing.

Lecture Agenda

- ✔ Section 1: Introduction to Rehashing
- ✔ Section 2: Rehashing Separate Chaining
- ✔ Section 3: Rehashing Double Hashing
- ✔ Section 4: Time Complexity & Space Complexity

Practice

Practice

1- Find if an array is subset of another array
2- Union and Intersection of two linked lists
3- Check for pair in array with sum as number x
4- Minimum delete operations to make all elements of array same
5- Minimum operation to make all elements equal in array
6- Count maximum points on same line
7- Check if a given array contains duplicate elements within k distance from each other
8- Find duplicates in a given array when elements are not limited to a range
9- Find top k (or most frequent) numbers in a stream
10- Find most frequent element in an array
11- Find smallest subarray with all occurrences of a most frequent element
12- First element occurring k times in an array
13- Given an array of pairs, find all symmetric pairs in it
14- Find the only repetitive element between 1 to n-1
15- Find any one of the multiple repeating elements in read only array

Practice

16- Find top three repeated in array
17- Group multiple occurrence of array elements ordered by first occurrence
18- Check if two given sets are disjoint
19- Sum of non-overlapping sum of two sets
20- Find elements which are present in first array and not in second
21- Check if two arrays are equal or not
22- Find pair with given sum and maximum shortest distance from end
23- Find pair with given product
24- Find missing elements of a range
25- Find k-th missing element in increasing sequence which is not present
26- Find pair with greatest product in array
27- Minimum number of subsets with distinct elements
28- Remove minimum number of elements such that no common element exist in both array
29- Count items common to both the lists but with different prices
30- Minimum index sum for common elements of two lists

Practice

31- Find pairs with given sum such that elements of pair are in different rows
32- Common elements in all rows of a given matrix
33- Find distinct elements common to all rows of a matrix
34- Find all permuted rows of a given row in a matrix
35- Change the array into a permutation of numbers from 1 to n
36- Count pairs from two sorted arrays whose sum is equal to a given value x
37- Count pairs from two linked lists whose sum is equal to a given value
38- Count quadruples from four sorted arrays whose sum is equal to a given value x
39- Calculate number of subarrays having sum exactly equal to k
40- Count pairs whose products exist in array
41- Find all pairs whose sum is x in two unsorted arrays
42- Cumulative frequency of count of each element in an unsorted array
43- Numbers with prime frequencies greater than or equal to k
44- Find pairs in array whose sums already exist in array
45- Find all pairs (a, b) in an array such that a % b = k

Practice

46- Convert an array to reduced form
47- Return maximum occurring character in an input string
48- Second most repeated word in a sequence
49- Smallest element repeated exactly ‘k’ times (not limited to small range)
50- Find k numbers with most occurrences in the given array
51- Find the first repeating element in an array of integers
52- Find sum of non-repeating (distinct) elements in an array
53- Find non-repeating element
54- Find k-th distinct (or non-repeating) element in an array
55- Print all distinct elements of a given integer array
56- Find only integer with positive value in positive negative value in array
57- Pairs of positive negative values in an array
58- Count pairs with given sum
59- Group words with same set of characters
60- Maximum distance between two occurrences of same element in array

Assignment

Implement STL Unordered Set

- Unordered sets are containers that store unique elements in no particular order, and which allow for fast retrieval of individual elements based on their value.
- In an unordered_set, the value of an element is at the same time its key, that identifies it uniquely. Keys are immutable, therefore, the elements in an unordered_set cannot be modified once in the container - they can be inserted and removed, though.
- Internally, the elements in the unordered_set are not sorted in any particular order, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their values (with a constant average time complexity on average).
- unordered_set containers are faster than set containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.
- An unordered_set is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized.
- The unordered_set can contain key of any type - predefined or user-defined data structure but when we define key of type user define the type, we need to specify our comparison function according to which keys will be compared.

Implement STL Unordered Set

- Sets vs. Unordered Sets: Set is an ordered sequence of unique keys whereas unordered_set is a set in which key can be stored in any order, so unordered. Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal).
- Methods on Unordered Sets: For unordered_set many functions are defined among which most users are the size and empty for capacity, find for searching a key, insert and erase for modification. The Unordered_set allows only unique keys, for duplicate keys unordered_multiset should be used.

More Info: cplusplus.com/reference/unordered_set/unordered_set/
More Info: en.cppreference.com/w/cpp/container/unordered_set
More Info: geeksforgeeks.org/unordered_set-in-cpp-stl/
More Info: geeksforgeeks.org/set-in-cpp-stl/

Implement STL Unordered Set

- Member functions: (constructor) Construct vector (public member function)
  (destructor) Vector destructor (public member function)
  (operator=) Assign content (public member function)
- Iterators: (begin) Return iterator to beginning (public member function)
  (end) Return iterator to end (public member function)
  (cbegin) Return const_iterator to beginning (public member function)
  (cend) Return const_iterator to end (public member function)
- Capacity: (empty) Test whether vector is empty (public member function)
  (size) Return size (public member function)
  (max_size) Return maximum size (public member function)
- Operations: (find) Get iterator to element (public member function)
  (count) Count elements with a specific value (public member function)
  (equal_range) Get range of equal elements (public member function)

Implement STL Unordered Set

- Modifiers: (insert) Insert element (public member function)
  (erase) Erase elements (public member function)
  (swap) Swap content (public member function)
  (clear) Clear content (public member function)
- Buckets: (bucket_count) Return number of buckets (public member function)
  (max_bucket_count) Return maximum number of buckets (public member function)
  (bucket_size) Return bucket size (public member type)
  (bucket) Locate element's bucket (public member function)
- Hash policy: (load_factor) Return load factor (public member function)
  (max_load_factor) Get or set maximum load factor (public member function)
  (rehash) Set number of buckets (public member function)
  (reserve) Request a capacity change (public member function)

More Info: cplusplus.com/reference/unordered_set/unordered_set/
More Info: en.cppreference.com/w/cpp/container/unordered_set

Implement STL Unordered Map

- Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.
- In an unordered_map, the key value is generally used to uniquely identify the element, while the mapped value is an object with the content associated to this key. Types of key and mapped value may differ.
- Internally, the elements in the unordered_map are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).
- unordered_map containers are faster than map containers to access individual elements by their key, although they are generally less efficient for range iteration through a subset of their elements.
- Unordered maps implement the direct access operator (operator[]) which allows for direct access of the mapped value using its key value as argument.

Implement STL Unordered Map

- unordered_map is an associated container that stores elements formed by combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.
- Internally unordered_map is implemented using Hash Table, the key provided to map are hashed into indices of hash table.
- unordered_map vs. unordered_set: In unordered_set, we have only key, no value, these are mainly used to see presence/absence in a set.
- unordered_map vs. map: map (like set) is an ordered sequence of unique keys whereas in unordered_map key can be stored in any order, so unordered. Map is implemented as balanced tree structure that is why it is possible to maintain an order between the elements (by specific tree traversal).

More Info: cplusplus.com/reference/unordered_map/unordered_map/
More Info: en.cppreference.com/w/cpp/container/unordered_map
More Info: geeksforgeeks.org/unordered_map-in-cpp-stl/
More Info: geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/

Implement STL Unordered Map

- Member functions: (constructor) Construct vector (public member function)
  (destructor) Vector destructor (public member function)
  (operator=) Assign content (public member function)
- Iterators: (begin) Return iterator to beginning (public member function)
  (end) Return iterator to end (public member function)
  (cbegin) Return const_iterator to beginning (public member function)
  (cend) Return const_iterator to end (public member function)
- Capacity: (empty) Test whether vector is empty (public member function)
  (size) Return size (public member function)
  (max_size) Return maximum size (public member function)
- Modifiers: (insert) Insert element (public member function)
  (erase) Erase elements (public member function)
  (swap) Swap content (public member function)
  (clear) Clear content (public member function)

Implement STL Unordered Map

- Element access: (operator[]) Access element (public member function)
  (at) Access element (public member function)
- Operations: (find) Get iterator to element (public member function)
  (count) Count elements with a specific value (public member function)
  (equal_range) Get range of equal elements (public member function)
- Buckets: (bucket_count) Return number of buckets (public member function)
  (max_bucket_count) Return maximum number of buckets (public member function)
  (bucket_size) Return bucket size (public member type)
  (bucket) Locate element's bucket (public member function)

Implement STL Unordered Map

- Hash policy:
  - (load_factor) Return load factor (public member function)
  - (max_load_factor) Get or set maximum load factor (public member function)
  - (rehash) Set number of buckets (public member function)
  - (reserve) Request a capacity change (public member function)

More Info: cplusplus.com/reference/unordered_map/unordered_map/
More Info: en.cppreference.com/w/cpp/container/unordered_map


DO
MORE.