YEAH A7

- By Trip && Zheng

Logistics

- This assignment is due on Friday, March 5th at class time. Feel free to work in pairs!

Logistics

- This assignment is due on Friday, March 5th at class time. Feel free to work in pairs!
- In this assignment, you'll be implementing classes for 2 hash tables: a Linear Probing Hash Table, and a Robin Hood Hash Table!

Logistics

- This assignment is due on Friday, March 5th at class time. Feel free to work in pairs!
- In this assignment, you'll be implementing classes for 2 hash tables: a Linear Probing Hash Table, and a Robin Hood Hash Table!
- We **highly** recommend going through Keith's lecture slides for detailed overviews of each.

Overview: Enums

- An **Enumerated Type**, or an **`enum`**, is a way to represent a variety of **subtypes** under a single umbrella type.

Overview: Enums

- An **Enumerated Type,** or an **`enum`**, is a way to represent a variety of **subtypes** under a single umbrella type.
- For example, if you wanted to create a **`car`** class, a great way to classify the car maker would be an enum! Here's an example:

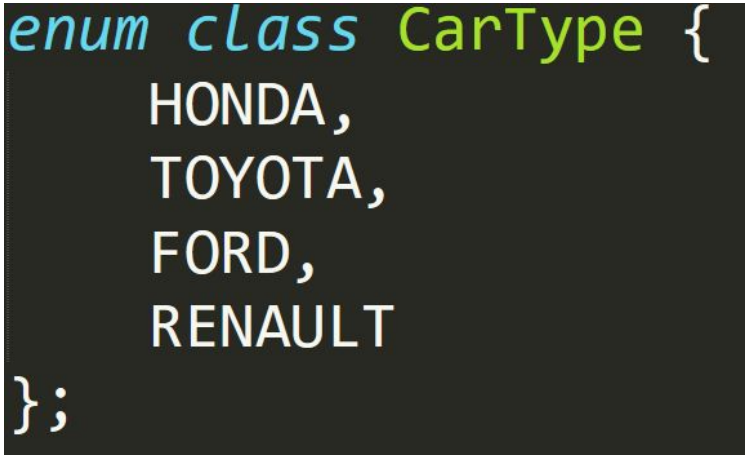
Overview: Enums

- An **Enumerated Type**, or an **`enum`**, is a way to represent a variety of **subtypes** under a single umbrella type.
- For example, if you wanted to create a **`car`** class, a great way to classify the car maker would be an enum! Here's an example:

```
enum class CarType {
    HONDA,
    TOYOTA,
    FORD,
    RENAULT
};
```

Overview: Enums

- An **Enumerated Type**, or an **`enum`**, is a way to represent a variety of **subtypes** under a single umbrella type.
- For example, if you wanted to create a **`car`** class, a great way to classify the car maker would be an enum! Here's an example:
- And example usages:
  - `CarType c1 = CarType::HONDA`
  - `CarType c2 = CarType::FORD`
  - `if (c1 == c2) // do something!`



Overview: Enums

- An **Enumerated Type**, or an **`enum`**, is a way to represent a variety of **subtypes** under a single umbrella type.
- For example, if you wanted to create a **`car`** class, a great way to classify the car maker would be an enum! Here's an example:
- And example usages:
  - `CarType c1 = CarType::HONDA`
  - `CarType c2 = CarType::FORD`
  - `if (c1 == c2) // do something!`
- Any questions?

```
enum class CarType {
    HONDA,
    TOYOTA,
    FORD,
    RENAULT
};
```

Linear Probing

Linear Probing

- In this first part, your job is to implement the following class:

```
class LinearProbingHashTable {
public:
    LinearProbingHashTable(HashFunction<std::string> hashFn);
    ~LinearProbingHashTable();

    bool contains(const std::string& key) const;

    bool insert(const std::string& key);
    bool remove(const std::string& key);

    bool isEmpty() const;
    int  size() const;

    void printDebugInfo();
```

Linear Probing

- Here's the private section:

```
private:
    enum class SlotType {
        TOMBSTONE, EMPTY, FILLED
    };

    struct Slot {
        std::string value;
        SlotType type;
    };

    Slot* elems;

    /* The rest is up to you to decide; see below */
};
```

Linear Probing

- Here's the private section:

```
private:
    enum class SlotType {
        TOMBSTONE, EMPTY, FILLED
    };

    struct Slot {
        std::string value;
        SlotType type;
    };

    Slot* elems;

    /* The rest is up to you to decide; see below */
};
```

Remember these? These **`enums`** signify whether a slot is empty, full, or a tombstone!

Linear Probing

- Here's the private section:

```
private:
    enum class SlotType {
        TOMBSTONE, EMPTY, FILLED
    };

    struct Slot {
        std::string value;
        SlotType type;
    };

    Slot* elems;

    /* The rest is up to you to decide; see below */
};
```

Remember these? These **`enums`** signify whether a slot is empty, full, or a tombstone!

This is a **`Slot`**, which is a single entry in the hash table!

Linear Probing

- Here's how to use the HashFunction type:

```
void foo(HashFunction<string> hashFn) {
    HashFunction<string> copy = hashFn // You can resassign HashFunctions!
    string str = "hello";
    int hashCode = copy(str); // Use the HashFunction like a normal function!
    cout << hashCode << endl;
}
```

Linear Probing: Insert

- To insert into the Hash Table, begin by trying to insert at the hashed value of the element. Try the next Slot if the current Slot is FILLED. Insert your element if the slot if EMPTY or TOMBSTONE.

```
LinearProbingHashTable::insert("elem2");
-----------------------------------------
int index = myHashFunction("elem2"); // index = 0
```

{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

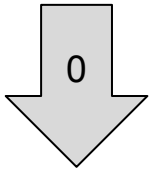
Linear Probing: Insert

- To insert into the Hash Table, begin by trying to insert at the hashed value of the element. Try the next Slot if the current Slot is FILLED. Insert your element if the slot if EMPTY or TOMBSTONE.

```
LinearProbingHashTable::insert("elem2");
-----------------------------------------
int index = myHashFunction("elem2"); // index = 0
```

This Slot is FILLED.

Keep moving!

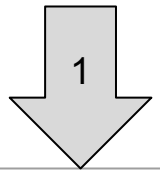


{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

Linear Probing: Insert

- To insert into the Hash Table, begin by trying to insert at the hashed value of the element. Try the next Slot if the current Slot is FILLED. Insert your element if the slot if EMPTY or TOMBSTONE.

```
LinearProbingHashTable::insert("elem2");
-----------------------------------------
int index = myHashFunction("elem2"); // index = 0
```



{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

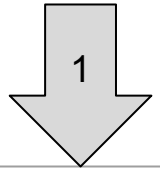
Linear Probing: Insert

- To insert into the Hash Table, begin by trying to insert at the hashed value of the element. Try the next Slot if the current Slot is FILLED. Insert your element if the slot if EMPTY or TOMBSTONE.

```
LinearProbingHashTable::insert("elem2");
-----------------------------------------
int index = myHashFunction("elem2"); // index = 0
```

This Slot a TOMBSTONE.

We can fill it! Return true!



{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

Linear Probing: Insert

- Some edge cases:
  - Be sure to return `false` if the table is full!
  - Also return `false` if the element being inserted **already exists** in the table!

Linear Probing: Contains

- To determine whether an element exists in the Hash Table, linearly scan through the array from the hash position until you either find the element (return true!) or find an empty space (return false!) You should **not** stop at tombstones.

{“elem1”, SlotType::FILLED }	{“elem2”, SlotType::TOMBSTONE }	{“elem3”, SlotType::FILLED }	{“”, SlotType::EMPTY }	{“”, SlotType::EMPTY }
0	1	2	3	4

Linear Probing: Contains

- To determine whether an element exists in the Hash Table, linearly scan through the array from the hash position until you either find the element (return true!) or find an empty space (return false!) You should **not** stop at tombstones.

```
LinearProbingHashTable::contains("elem3");
-------------------------------------------
int index = myHashFunction("elem3"); // index = 0
```

{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

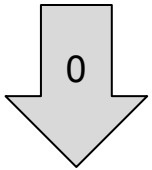
Linear Probing: Contains

- To determine whether an element exists in the Hash Table, linearly scan through the array from the hash position until you either find the element (return true!) or find an empty space (return false!) You should **not** stop at tombstones.

```
LinearProbingHashTable::contains("elem3");
-------------------------------------------
int index = myHashFunction("elem3"); // index = 0
```

The element at index 0 is filled, but the value is not elem3!

Keep moving!

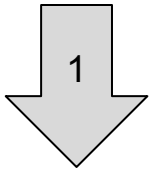


{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

Linear Probing: Contains

- To determine whether an element exists in the Hash Table, linearly scan through the array from the hash position until you either find the element (return true!) or find an empty space (return false!) You should **not** stop at tombstones.

```
LinearProbingHashTable::contains("elem3");
-------------------------------------------
int index = myHashFunction("elem3"); // index = 0
```



{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

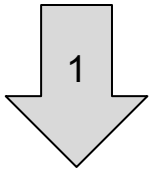
Linear Probing: Contains

- To determine whether an element exists in the Hash Table, linearly scan through the array from the hash position until you either find the element (return true!) or find an empty space (return false!) You should **not** stop at tombstones.

```
LinearProbingHashTable::contains("elem3");
-------------------------------------------
int index = myHashFunction("elem3"); // index = 0
```

This elem is a tombstone!

Keep moving!

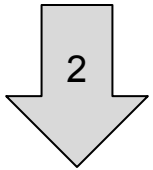


{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

Linear Probing: Contains

- To determine whether an element exists in the Hash Table, linearly scan through the array from the hash position until you either find the element (return true!) or find an empty space (return false!) You should **not** stop at tombstones.

```
LinearProbingHashTable::contains("elem3");
-------------------------------------------
int index = myHashFunction("elem3"); // index = 0
```



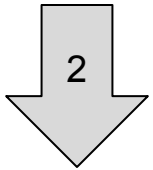
{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

Linear Probing: Contains

- To determine whether an element exists in the Hash Table, linearly scan through the array from the hash position until you either find the element (return true!) or find an empty space (return false!) You should **not** stop at tombstones.

```
LinearProbingHashTable::contains("elem3");
-------------------------------------------
int index = myHashFunction("elem3"); // index = 0
```

This is our element!
Return true!



{"elem1", SlotType::FILLED }	{"elem2", SlotType::TOMBSTONE }	{"elem3", SlotType::FILLED }	{"", SlotType::EMPTY }	{"", SlotType::EMPTY }
0	1	2	3	4

Linear Probing: Contains

- Some edge cases:
  - Be sure to return `false` if the table is empty!
  - That's it :)

Linear Probing: Remove

- Remove is just like **`contains`**, except when you locate the element, set its SlotType field to TOMBSTONE.

Linear Probing: Remove

- Remove is just like **`contains`**, except when you locate the element, set its SlotType field to TOMBSTONE.
- Be sure to return **`false`** if the table is empty, or if you can't find the element!

Linear Probing: Remove

- Remove is just like **`contains`**, except when you locate the element, set its SlotType field to TOMBSTONE.
- Be sure to return **`false`** if the table is empty, or if you can't find the element!
- Large brain hint: Is there a way you can **consolidate** logic between **`contains()`** and **`remove()`**?

Linear Probing

- Final notes about this problem:
  - You shouldn't use **recursion** in this project. Doing so will cause stack overflow problems for our larger test cases.

Linear Probing

- Final notes about this problem:
  - You shouldn't use **recursion** in this project. Doing so will cause stack overflow problems for our larger test cases.
  - Don't mess with the `string` values of slots, unless you're inserting a new element. If you're `removing` a slot, just set its SlotType to TOMBSTONE.

Linear Probing

- Final notes about this problem:
  - You shouldn't use **recursion** in this project. Doing so will cause stack overflow problems for our larger test cases.
  - Don't mess with the `string` values of slots, unless you're inserting a new element. If you're `removing` a slot, just set its SlotType to TOMBSTONE.
  - In a similar vein, don't read the string value of TOMBSTONE slots! Take a second to think why this is important.

Linear Probing

- Final notes about this problem:
  - You shouldn't use **recursion** in this project. Doing so will cause stack overflow problems for our larger test cases.
  - Don't mess with the **`string`** values of slots, unless you're inserting a new element. If you're **`removing`** a slot, just set its SlotType to TOMBSTONE.
  - In a similar vein, don't read the string value of TOMBSTONE slots! Take a second to think why this is important.
  - We highly recommend implementing **printDebugInfo()** like you did in A6.

Linear Probing

- Final notes about this problem:
  - You shouldn't use **recursion** in this project. Doing so will cause stack overflow problems for our larger test cases.
  - Don't mess with the **`string`** values of slots, unless you're inserting a new element. If you're **`removing`** a slot, just set its SlotType to TOMBSTONE.
  - In a similar vein, don't read the string value of TOMBSTONE slots! Take a second to think why this is important.
  - We highly recommend implementing **printDebugInfo()** like you did in A6.
  - Please don't use **any** auxiliary data structures in this assignment. A Slot * is all you need.

Linear Probing

Any last questions?

Robinhood Hashing

Robinhood Hashing

- What are some drawbacks of LP?

Robinhood Hashing

- What are some drawbacks of LP?
  - Some elements are much further than others.

Robinhood Hashing

- What are some drawbacks of LP?
  - Some elements are much further than others.
  - Tombstones can drastically slow down searches.

Robinhood Hashing

- What are some drawbacks of LP?
  - Some elements are much further than others.
  - Tombstones can drastically slow down searches.
- The next class you are implementing attempt to alleviate these problems!

Robinhood Hashing

- What are some drawbacks of LP?
  - Some elements are much further than others.
  - Tombstones can drastically slow down searches.
- The next class you are implementing attempt to alleviate these problems!

Robinhood Hashing

- What are some drawbacks of LP?
  - Some elements are much further than others.
  - Tombstones can drastically slow down searches.
- The next class you are implementing attempt to alleviate these problems!

Robinhood Hashing

- The public functions are exactly the same:

```
class RobinHoodHashTable {
public:
    RobinHoodHashTable(HashFunction<std::string> hashFn);
    ~RobinHoodHashTable();

    bool contains(const std::string& key) const;

    bool insert(const std::string& key);
    bool remove(const std::string& key);

    bool isEmpty() const;
    int  size() const;

    void printDebugInfo();
```

Robinhood Hashing

- Some changes to the private members:

```
private:
    static const int EMPTY_SLOT = /* something */;

    struct Slot {
        std::string value;
        int distance;
    };

    Slot* elems;
```

Robinhood Hashing

- Some changes to the private members:

```
private:
    static const int EMPTY_SLOT = /* something */;

    struct Slot {
        std::string value;
        int distance;
    };

    Slot* elems;
```

No tombstones (rip).

Robinhood Hashing

- Some changes to the private members:

```
private:
    static const int EMPTY_SLOT = /* something */;

    struct Slot {
        std::string value;
        int distance;
    };

    Slot* elems;
```

No tombstones (rip).

We additionally store **distance**.

Robinhood Hashing

- Some changes to the private members:

```
private:
    static const int EMPTY_SLOT = /* something */;

    struct Slot {
        std::string value;
        int distance;
    };

    Slot* elems;
```

No tombstones (rip).

We additionally store **distance**. **Distance** measures how many slots to the RIGHT of the original slot (wraps around) the element is.

Robinhood Hashing

- What changed?
- Insert, contain, and remove.

Robinhood Hashing: Insert

- LP: If no duplicates and has space, find the next open slot (tombstone or empty) and insert the element there.

Robinhood Hashing: Insert

- LP: If no duplicates and has space, find the next open slot (tombstone or empty) and insert the element there.
- Robinhood: If no duplicates and has space, find the next empty slot OR a slot filled by an element with less distance than you.

Robinhood Hashing: Insert

- LP: If no duplicates and has space, find the next open slot (tombstone or empty) and insert the element there.
- Robinhood: If no duplicates and has space, find the next empty slot OR a slot filled by an element with less distance than you.
    - Remember to find a home for that element using the same rule :).

Robinhood Hashing: Contains

- LP: Go through the table and look for it. Stop when you find the element or an open slot.

Robinhood Hashing: Contains

- LP: Go through the table and look for it. Stop when you find the element or an open slot.
- Robinhood: Go through the table and look for it. Stop when you find the element or an open slot **OR** an element that is closer to home than you should be.

Robinhood Hashing: Remove

- LP: Look for it using the same set of rules as **`contains`**. If found, replace with a tombstone and call it a day.

Robinhood Hashing: Remove

- LP: Look for it using the same set of rules as **`contains`**. If found, replace with a tombstone and call it a day.
- Robinhood: Look for it using the same set of rules as **`contains`**. If found, delete the element, and shift the element to the right of it.

Robinhood Hashing: Remove

- LP: Look for it using the same set of rules as **`contains`**. If found, replace with a tombstone and call it a day.
- Robinhood: Look for it using the same set of rules as **`contains`**. If found, delete the element, and shift the element to the right of it.
    - Repeat this shift until you have found an empty slot or an element situating in its native slot.

Robinhood Hashing: Remove

- LP: Look for it using the same set of rules as **`contains`**. If found, replace with a tombstone and call it a day.
- Robinhood: Look for it using the same set of rules as **`contains`**. If found, delete the element, and shift the element to the right of it.
  - Repeat this shift until you have found an empty slot or an element situating in its native slot.
  - This is called **backwards-shift deletion**.

Robinhood Hashing: Implementation Thoughts

- Most of the tips from LP applies!

Linear Probing

- Final notes about this problem:
  - You shouldn't use **recursion** in this project. Doing so will cause stack overflow problems for our larger test cases.
  - Don't mess with the **`string`** values of slots, unless you're inserting a new element. If you're **`removing`** a slot, just set its SlotType to TOMBSTONE.
  - In a similar vein, don't read the string value of TOMBSTONE slots! Take a second to think why this is important.
  - We highly recommend implementing **printDebugInfo()** like you did in A6.
  - Please don't use **any** auxiliary data structures in this assignment. A Slot * is all you need.

~~Linear Probing~~ Robinhood Hashing

- Final notes about this problem:
  - You shouldn't use **recursion** in this project. Doing so will cause stack overflow problems for our larger test cases.
  - Don't mess with the `string` values of slots, unless you're inserting a new element. If you're `removing` a slot, just set its SlotType to TOMBSTONE.
  - ~~In a similar vein, don't read the string value of TOMBSTONE slots! Take a second to think why this is important.~~
  - We highly recommend implementing **printDebugInfo()** like you did in A6.
  - Please don't use **any** auxiliary data structures in this assignment. A Slot * is all you need.

Robinhood Hashing: Implementation Thoughts

- Most of the tips from LP applies!
- Do not kick elements out when distances are the same.

Robinhood Hashing: Implementation Thoughts

- Most of the tips from LP applies!
- Do not kick elements out when distances are the same.
- Be super careful about your backwards-shift deletion. Use the interactive interface to make sure that you are doing it correctly!

Robinhood Hashing: Implementation Thoughts

- Most of the tips from LP applies!
- Do not kick elements out when distances are the same.
- Be super careful about your backwards-shift deletion. Use the interactive interface to make sure that you are doing it correctly!
- There are a lot of optimizations you need to make.

Robinhood Hashing: Implementation Thoughts

- Most of the tips from LP applies!
- Do not kick elements out when distances are the same.
- Be super careful about your backwards-shift deletion. Use the interactive interface to make sure that you are doing it correctly!
- There are a lot of optimizations you need to make.
- Start early :).

Robinhood Hashing

Questions?