Lecture 11:
Hash Tables

Logistics

- Point Solutions App
  - Session ID: **database**
- Mid-term Exam on **Oct 3**

Recap

- Database Configuration
- Indexing in C++

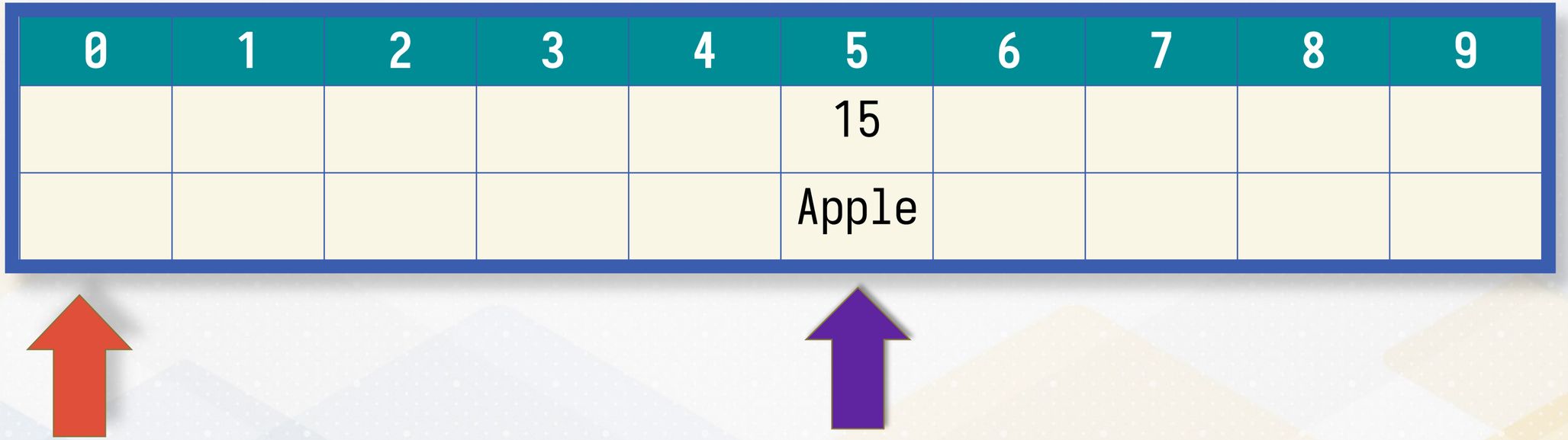
Lecture Overview

- Hash Tables
- Hash Function
- Deletion and Position Tracking
- Quadratic Probing
- Double Hashing

Hash Table

Hash Table

Think of a **hash table** like a **cabinet of drawers**.
Each drawer can hold a piece of paper with the **key** and **associated value**.



DIRECT ACCESS USING HASH FUNCTION

Inserting Key-Value Pair

Insert (15, Apple)

0	1	2	3	4	5	6	7	8	9
					15				
					Apple				

HASH(KEY) = KEY % 10 = 15 % 10 = 5 (since 15 = 10 * 1 + 5)

Inserting Key-Value Pair

Insert (26, Grape)

0	1	2	3	4	5	6	7	8	9
					15	26			
					Apple	Grape			

HASH(KEY) = KEY % 10 = 26 % 10 = 6 (since 26 = 10 * 2 + 6)

Inserting Key-Value Pair with Collision

Insert (56, Kiwi)

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		

HASH(KEY) = KEY % 10 = 56 % 10 = 6

Finding Key-Value Pair

Find value associated with key 15 = Apple

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		

HASH(KEY) = KEY % 10 = 15 % 10 = 5

Finding Key-Value Pair

Find value associated with key 56 = Kiwi

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		

HASH(KEY) = KEY % 10 = 56 % 10 = 6

Finding Key-Value Pair

Find value associated with key 86 = KEY NOT FOUND

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		

HASH(KEY) = KEY % 10 = 86 % 10 = 6

Hash Function

Hash Function

Hash function maps keys to slots in the hash table

```
size_t hashFunction(int key) {
  return key % totalSlots; // Our simple formula
}
```

```
HASH(KEY) = KEY % TABLE_SIZE
```

Collision Handling

Linear Probing — Move to the next available slot until we find one empty.

0	1	2	3	4	5	6	7	8	9
80	11				15	26	56	18	79
Lime	Date				Apple	Grape	Kiwi	Fig	Pear

If slot 8 is taken, check slot 9, then 0, 1, and so on.

"Linear" Probing

Formula to find the index of the next slot I after K “probes" is “linear" with respect to K.

```
INDEX I = (HASH(KEY) + K) % TABLE_SIZE

HASH(KEY) = KEY % TABLE_SIZE
```

HashEntry Struct

HashEntry Struct	Represents a piece of paper containing **key** and **value**.

```
struct HashEntry {
  int key, value;
  HashEntry(int k, int v) : key(k), value(v) {}
};
```

HashIndex

HashTable is simply an array of drawers. Each drawer can either hold a **HashEntry** or be **empty** (indicating no value).

```
class HashIndex {
  std::vector<std::optional<HashEntry>> hashTable;

  static const size_t capacity = 100; // Hard-coded capacity
  HashIndex() { hashTable.resize(capacity); }
};
```

Inserting Key-Value Pair

insertOrUpdate	Uses linear probing to find the next available slot for insertion or locate an existing key for updating.

```
void insertOrUpdate(int key, int value) {
  size_t index = hashFunction(key);
  do {
    if (!hashTable[index]) {
      hashTable[index] = HashEntry{key, value, true}; // Insert new entry
      break;
    } else if (hashTable[index]->key == key) {
      hashTable[index]->value += value; // Update existing entry
      break;
    }
    index = (index + 1) % capacity; // Linear probing to next slot
  } while (index != originalIndex); // Returned to starting point
}
```

Finding Key-Value Pair

getValue	The retrieval process employs linear probing as well to navigate through potential collision sequences.

```
int getValue(int key) const {
  size_t index = hashFunction(key);
  do {
    if (hashTable[index] && hashTable[index]->key == key) {
      return hashTable[index]->value; // Key found
    }
    index = (index + 1) % capacity; // Continue probing
  } while (index != originalIndex);
  return -1; // Key not found
}
```

Operation Complexity

N = Number of key-value pairs in the hash table.

	NO/FEW COLLISIONS	LOTS OF COLLISIONS
INSERT	O(1)	O(N)
FIND	O(1)	O(N)

Clustering Problem with High Collision Rates

Bad Hash Function: HASH(KEY) = 5

0	1	2	3	4	5	6	7	8	9
					15	26	56		
					Apple	Grape	Kiwi		

HASH(KEY) = 5

Deletion & Position Tracking

Finding key 32

Find value associated with key 32 = Fig

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				

HASH(KEY) = KEY % 10 = 32 % 10 = 2

Deletion

Delete key 13

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				

HASH(KEY) = KEY % 10 = 13 % 10 = 3

Deletion

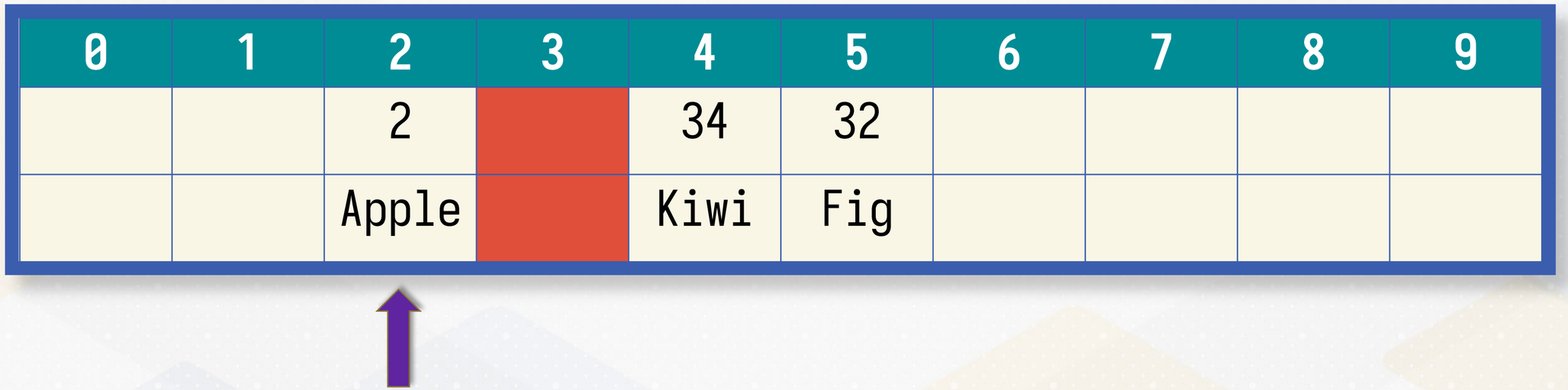
Delete key 13

0	1	2	3	4	5	6	7	8	9
		2		34	32				
		Apple		Kiwi	Fig				

HASH(KEY) = KEY % 10 = 13 % 10 = 3

Finding key 32 After Deletion

Find value associated with key 32 = Not Found



HASH(KEY) = KEY % 10 = 32 % 10 = 2

Solution #1: Rehashing All Keys

Insert all keys into another empty hash table.

0	1	2	3	4	5	6	7	8	9
		2	32	34					
		Apple	Fig	Kiwi					

HASH(KEY) = KEY % 10

Solution #2: Lazy Deletion

Mark a hash entry as deleted without removing it.

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				
1	1	1	0	1	1	1	1	1	1

Finding key 32 After Deletion

Find value associated with key 32 = Fig

0	1	2	3	4	5	6	7	8	9
		2	13	34	32				
		Apple	Grape	Kiwi	Fig				
1	1	1	0	1	1	1	1	1	1

HASH(KEY) = KEY % 10 = 32 % 10 = 2

Lazy Deletion using Exists Flag

```
struct HashEntry {
  int key;
  int value;
  bool exists; // Indicates if the entry is active or deleted

  // Updated constructor includes position
  HashEntry(int k, int v, int pos) : key(k), value(v), exists(true) {}
};
```

Vector to Static-Sized Array

```
class HashIndex {
private:
  static const size_t capacity = 100; // Hard-coded capacity
  HashEntry hashTable[capacity];      // Static-sized array
};
```

Position Tracking

Position is used for tracking the location of each entry within the hash table.

```
struct HashEntry {
  int key;
  int value;
  int position; // Tracks the final position in the array
  bool exists;  // Indicates if the entry is active


  // Updated constructor includes position
  HashEntry(int k, int v, int pos)
      : key(k), value(v), position(pos), exists(true) {}
};
```

Position Tracking

```
void insertOrUpdate(int key, int value) {
  size_t originalIndex = hashFunction(key);
  bool inserted = false;
  do {
    if (!hashTable[index].exists) {
      hashTable[index] = HashEntry(key, value, true);
      hashTable[index].position = index;
      inserted = true;
      break;
    }
    ...
  }
}
```

Limitations of Linear Probing

Clustering increases likelihood of collision; searches become inefficient

0	1	2	3	4	5	6	7	8	9
		2	12	22	32				
		Apple	Grape	Kiwi	Fig				

HASH(KEY) = KEY % 10

Quadratic Probing

Quadratic Probing

Formula to find the index of the next slot I after K “probes” is “quadratic" with respect to K.

$$\text{INDEX I} = (\text{HASH(KEY)} + \text{K}^2) \% \text{ TABLE\_SIZE}$$

$$\text{HASH(KEY)} = \text{KEY} \% \text{ TABLE\_SIZE}$$

Inserting 2

Insert (2, Apple)

0	1	2	3	4	5	6	7	8	9
		2							
		Apple							

HASH(2) = 2; K = 0; INDEX I = (2 + 0^2) % 10 = 2

Inserting 12

Insert (12, Grape)

0	1	2	3	4	5	6	7	8	9
		2	12						
		Apple	Grape						

HASH(12) = 2; K = 1; INDEX I = (2 + 1^2) % 10 = 3

Inserting 22

Insert (22, Kiwi)

0	1	2	3	4	5	6	7	8	9
		2	12			22			
		Apple	Grape			Kiwi			

HASH(22) = 2; K = 2; INDEX I = (2 + 2^2) % 10 = 16 % 10 = 6

Inserting 32

Insert (32, Fig)

0	1	2	3	4	5	6	7	8	9
	32	2	12			22			
	Fig	Apple	Grape			Kiwi			

HASH(32) = 2; K = 3; INDEX I = (2 + 3^2) % 10 = 11 % 10 = 1

Quadratic Probing

Probe sequence follows a quadratic formula.

```
void insertOrUpdate(int key, int value) {
  size_t originalIndex = hashFunction(key);
  bool inserted = false;
  int i = 0; // Attempt counter
  do {
    if (!hashTable[index].exists) {
      ...
    } else if (hashTable[index].key == key) {
      ...
    }
    i++;
    index = (originalIndex + i * i) % capacity; // Quadratic probing
  } while (index != originalIndex && !inserted);
}
```

Benefits of Quadratic Probing

Better spread of keys; higher search efficiency

0	1	2	3	4	5	6	7	8	9
	32	2	12			22			
	Fig	Apple	Grape			Kiwi			

Limitation of Quadratic Probing

Secondary clustering of all keys hashed to 2.

0	1	2	3	4	5	6	7	8	9
	32	2	12			22			
	Fig	Apple	Grape			Kiwi			

Double Hashing

Double Hashing

Double hashing	Uses a secondary hash function to calculate the probe step, offering a unique probe sequence for each key.

```
INDEX I = (HASH1(KEY) + K * HASH2(KEY)) % TABLE_SIZE

HASH1(KEY) = KEY % TABLE_SIZE

HASH2(KEY) = 1 + KEY % (TABLE_SIZE - 1)
```

Inserting 2

Insert (2, Apple)

0	1	2	3	4	5	6	7	8	9
		2							
		Apple							

HASH1(2) = 2; HASH2(2) = 1 + (2 % 9) = 1 + 2 = 3
K = 0; INDEX I = (2 + 0 * 3) % 10 = 2

Inserting 12

Insert (12, Grape)

0	1	2	3	4	5	6	7	8	9
		2				12			
		Apple				Grape			

HASH1(12) = 2; HASH2(12) = 1 + (12 % 9) = 1 + 3 = 4
K = 1; INDEX I = (2 + 1 * 4) % 10 = 6 % 10 = 6

Inserting 22

Insert (22, Kiwi)

0	1	2	3	4	5	6	7	8	9
		2				12	22		
		Apple				Grape	Kiwi		

HASH1(22) = 2; HASH2(22) = 1 + (22 % 9) = 1 + 4 = 5
K = 1; INDEX I = (2 + 1 * 5) % 10 = 7 % 10 = 7

Inserting 32

Insert (32, Fig)

0	1	2	3	4	5	6	7	8	9
		2				12	22	32	
		Apple				Grape	Kiwi	Fig	

HASH1(32) = 2; HASH2(32) = 1 + (32 % 9) = 1 + 5 = 6
K = 1; INDEX I = (2 + 1 * 6) % 10 = 8 % 10 = 8

Benefit of Double Hashing

0	1	2	3	4	5	6	7	8	9
		2				12	22	32	
		Apple				Grape	Kiwi	Fig	

Conclusion

- Hash Tables
- Hash Function
- Deletion and Position Tracking
- Quadratic Probing
- Double Hashing