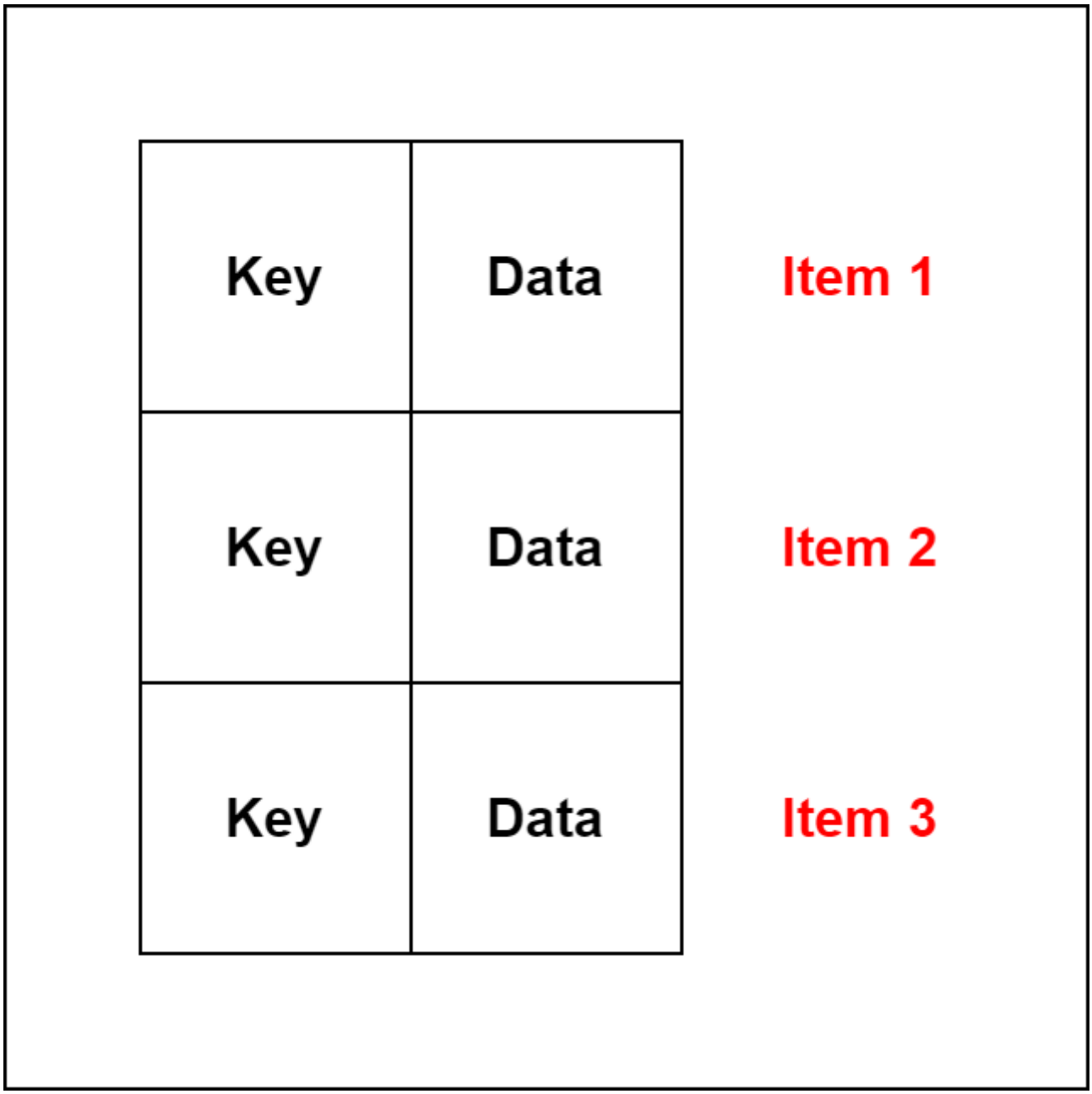
Alright, let's explore the fascinating world of hash tables! Imagine you have a special container where you want to keep things organized. These things come in pairs - each with a unique number and some important data.

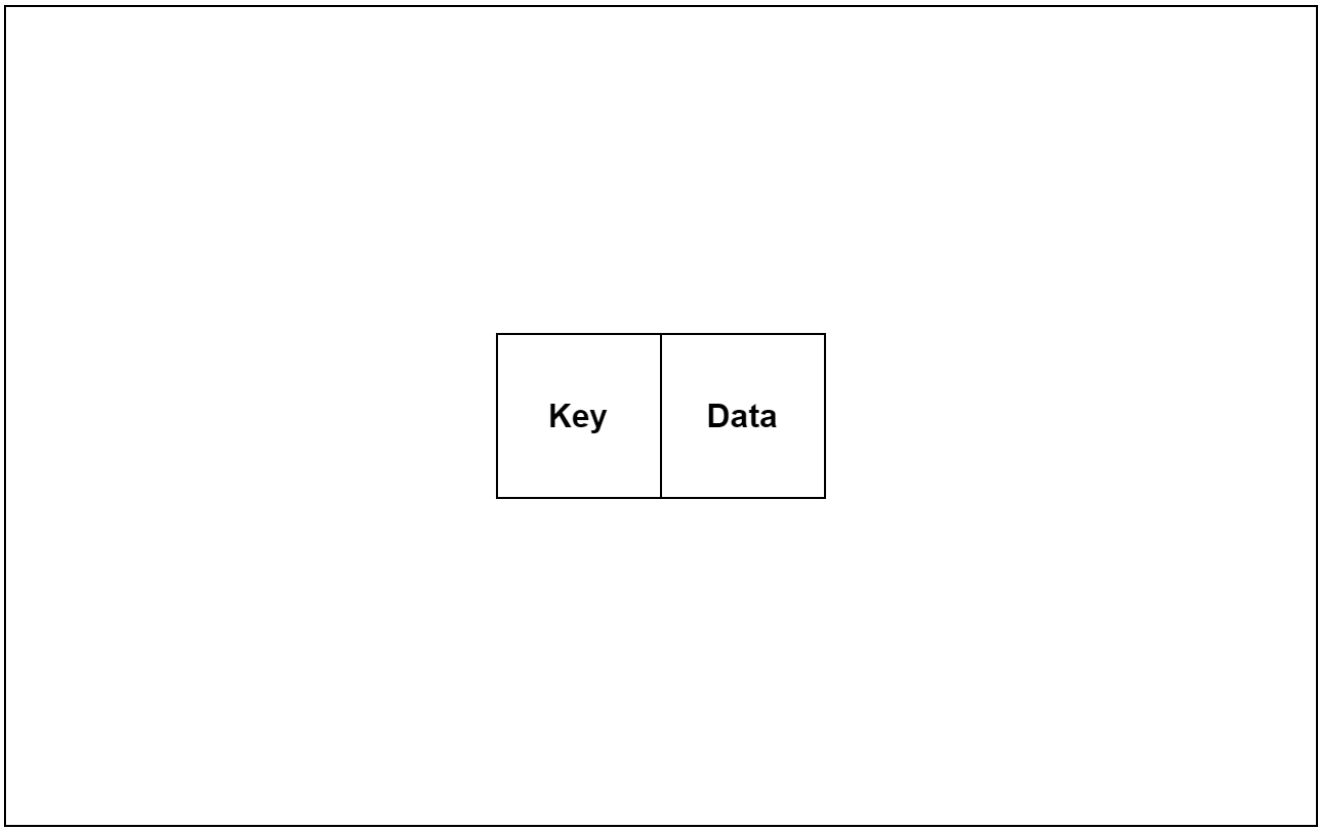
Think of this unique number as a special key, like a secret code that only that thing knows. And the data that goes with it is like the treasure that's hidden and protected by that secret code.

So, in our magical hash table, each item has a key that's like its ID card, and a value that's like its hidden treasure. And the beauty of this is that when you want to find that treasure later, you don't need to search the whole container. You just tell the table the key, and it quickly gives you the treasure associated with it. It's like having a special map that takes you straight to your treasure chest!



So, in simple terms, a hash table is like a super-efficient organizer that pairs up unique IDs (keys) with valuable items (values), making it easy to find what you're looking for without any fuss.

- **Key**- unique integer that is used for indexing the values
- **Value** - data that are associated with keys.

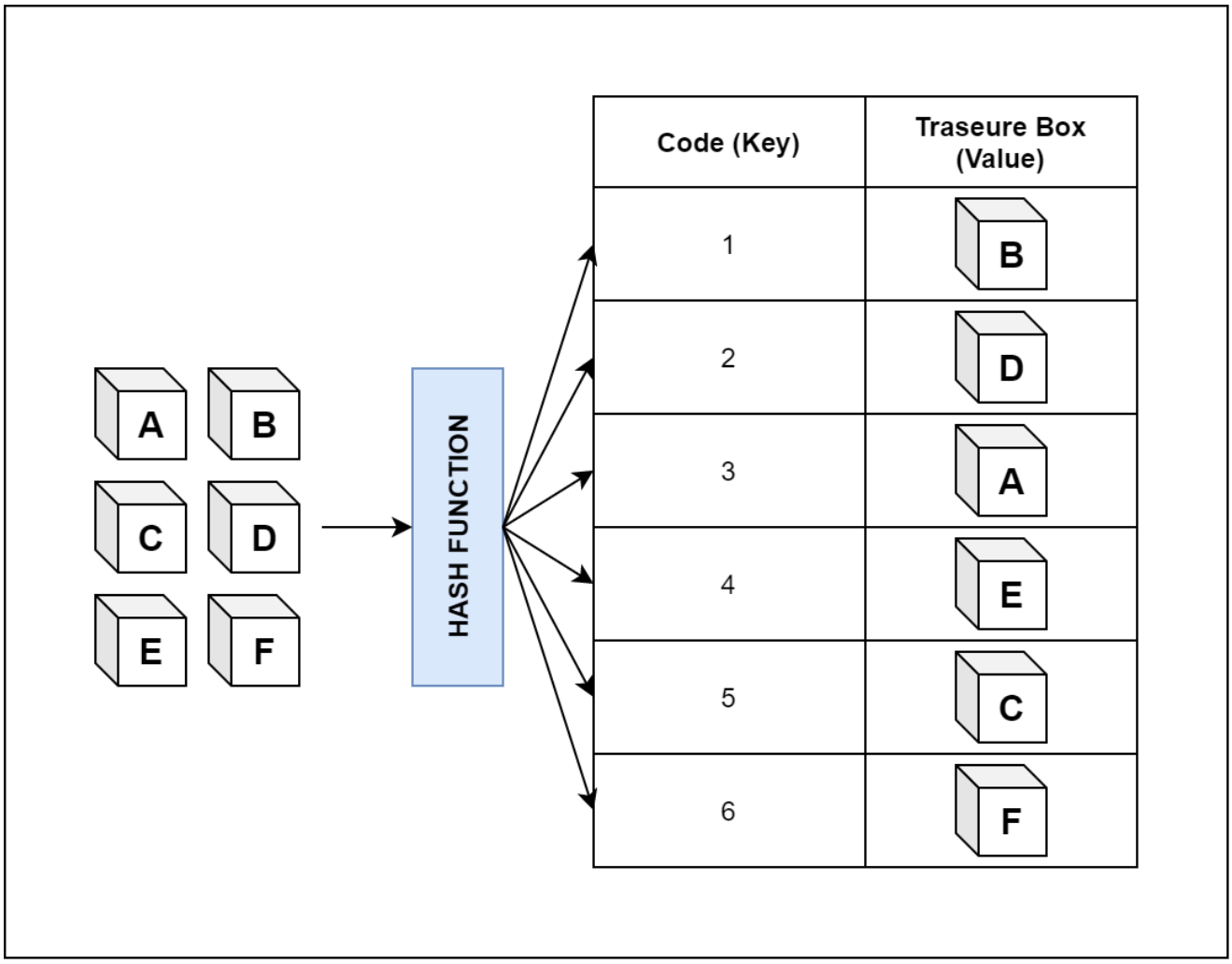


Hashing (Hash Function)

Alright, let's delve into the concept of hashing and hash functions! Imagine we have a special table where we want to arrange things in a smart way. We know that each thing comes with a unique code, like a secret symbol that only that thing knows. And we also have important stuff linked with each of these codes.

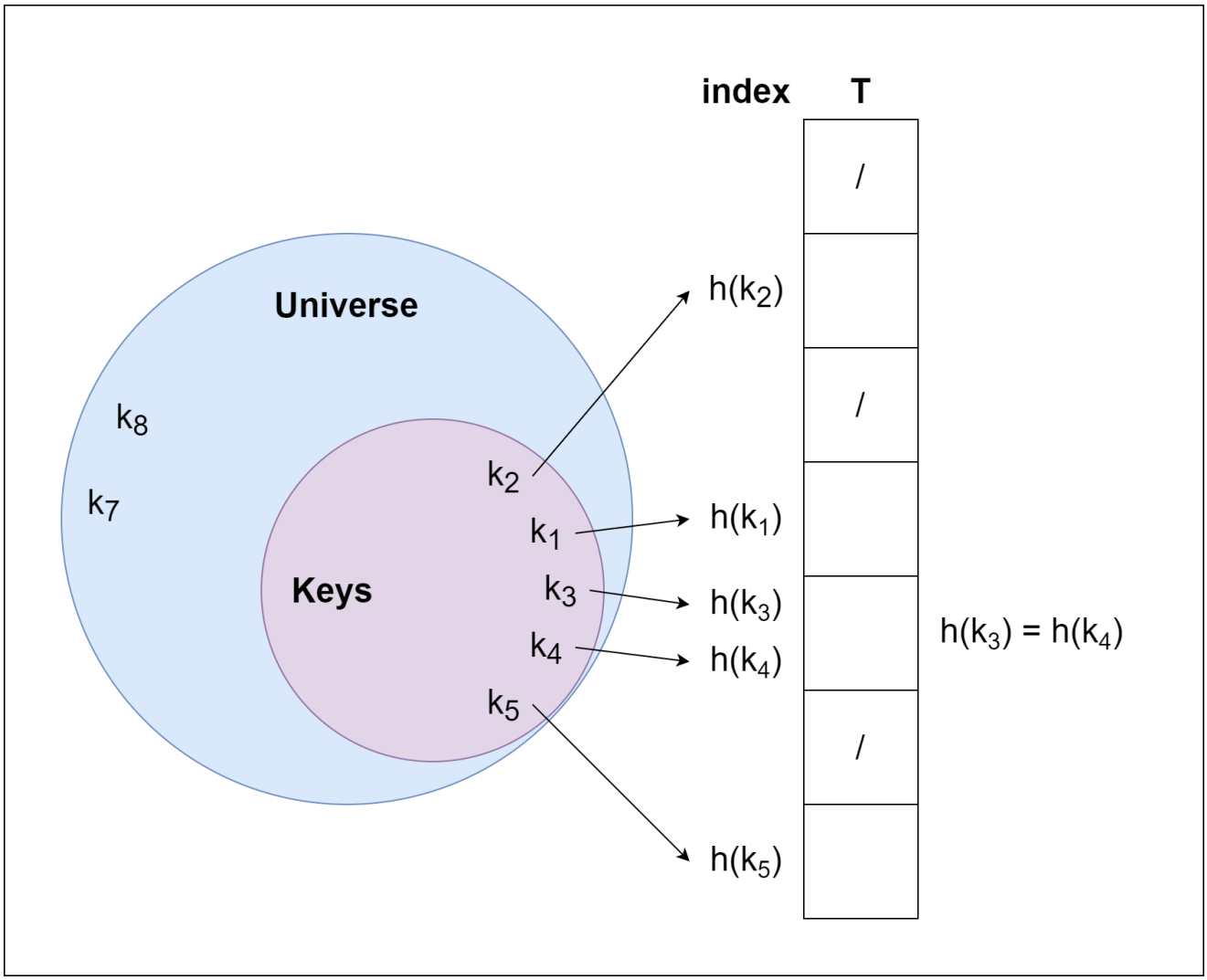
So, when we want to store something in our table, we use this unique code as a guide. It's like a treasure map that tells us exactly where to put our item. We call this process "hashing."

Now, let's say we have a key, which is like the code for a certain thing, and we have a special function called a "hash function." Think of this function like a magical tool that takes the key as input and produces a new number. This new number tells us exactly where in the table to store our item.



In other words, if we have a key "k" and we apply the hash function "h(x)" to it, the result "h(k)" is like the address where we'll keep the item linked with that key. It's like the hash function is guiding us to the perfect spot in our special table.

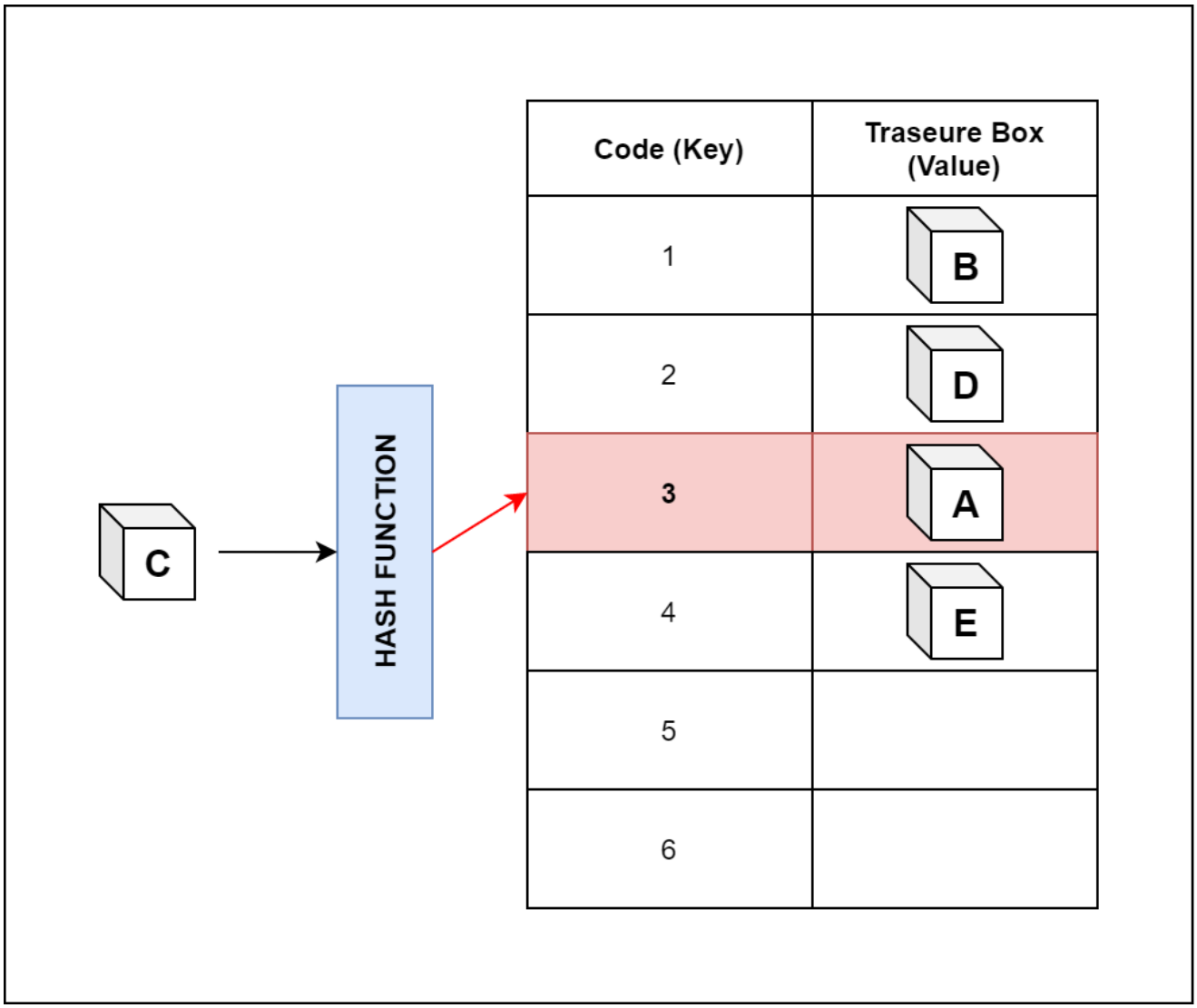
So, hashing is this clever way of using keys and a hash function to quickly find the right place to store our stuff in our organized table.



Hash Collision

Okay, let's talk about something important when it comes to hashing - hash collision. Imagine you have this amazing guide that tells you where to put things in your special table. But sometimes, two things might have the same guide that leads them to the same spot in the table. This situation, where different things end up in the same place, is called a "hash collision."

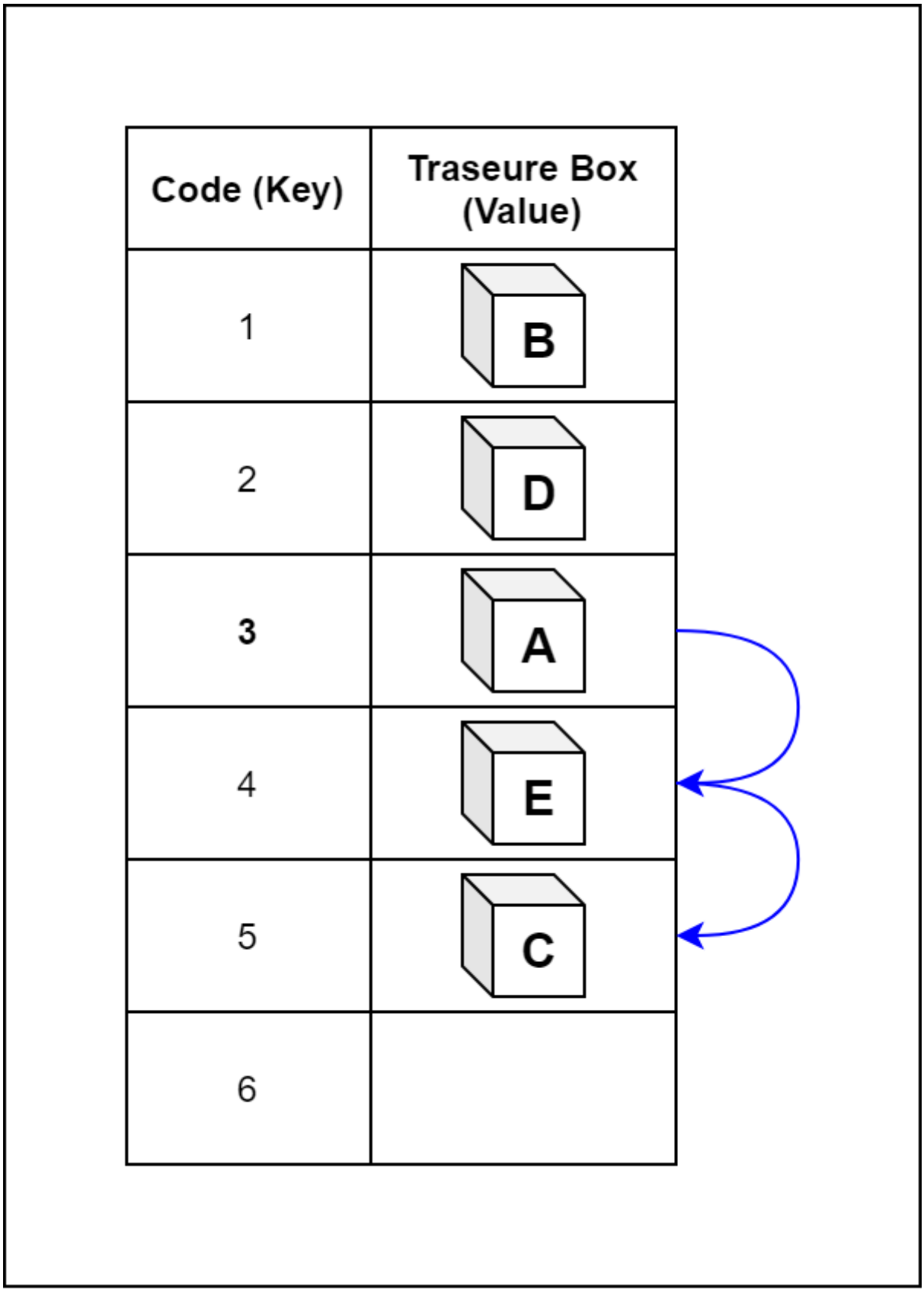
Now, when we have a hash collision, we're faced with a bit of a puzzle. We need to decide what to do with these things that want to share the same spot. We have a couple of tricks up our sleeve to solve this problem.



One way is to use "chaining." Think of it like a chain of connected boxes. When things collide, we create a chain of these boxes at that spot in the table. Each box holds one of the things that wants to be in that same place. So, when we need to find something, we follow the chain until we get to the right box.

Code (Key)	Traseure Box (Value)
1	B
2	D
3	A → C
4	E
5	
6	

Another approach is called "open addressing." This is like a game of musical chairs. When something collides, we keep looking for the next available spot until we find one. It's like moving to the next chair if your favorite one is taken. There are different ways to do this, like "linear probing," "quadratic probing," and even "double hashing."

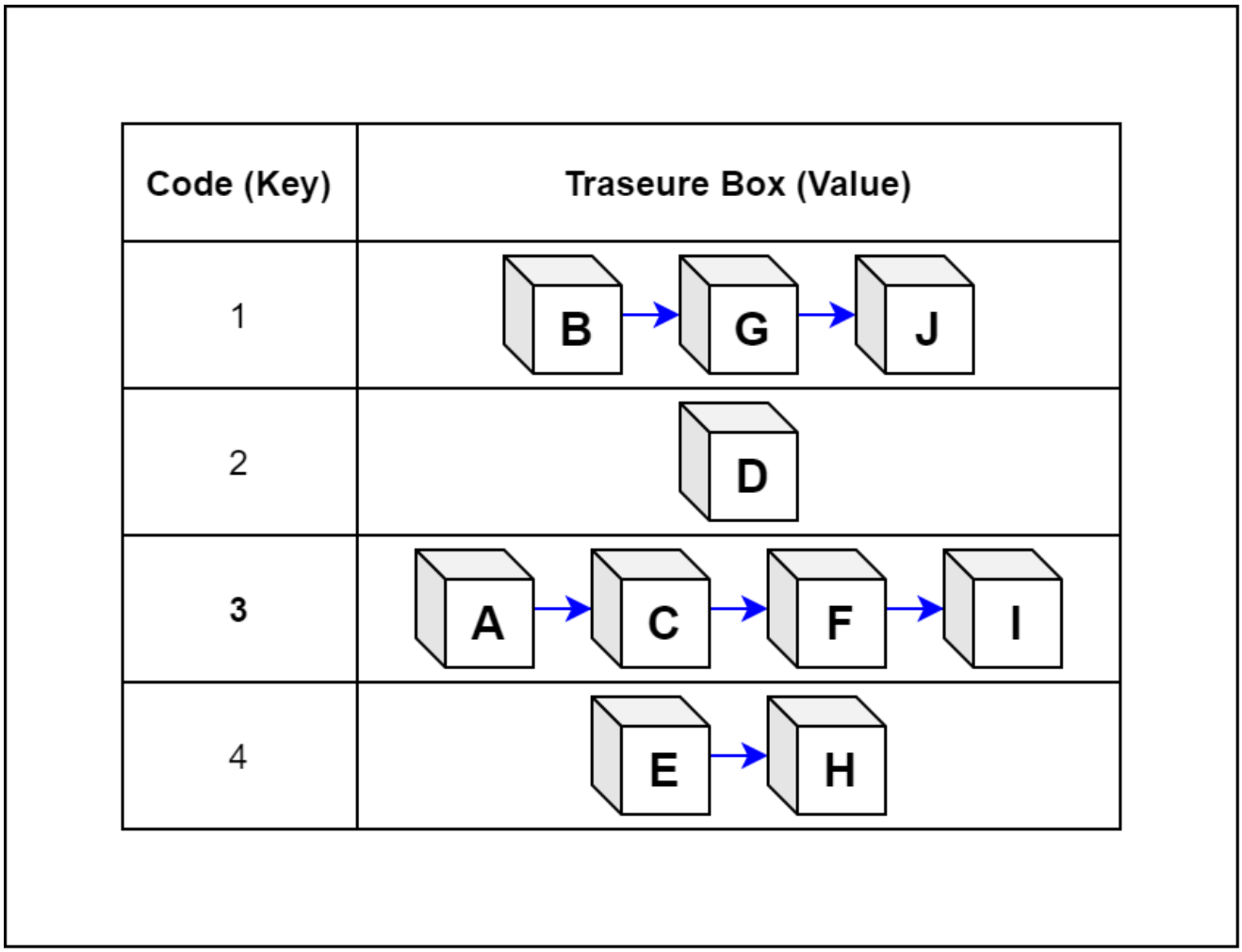


1. Collision resolution by chaining

Alright, let's dive into the first way to tackle hash collisions, which is like giving each of our colliding items a cozy little corner.

Imagine our hash function accidentally points two items to the same spot in our table. Now, instead of making them fight for that spot, we're going to make a small list right there. Think of it as a sort of "shared space" for these items. We call this method "chaining."

Each spot in our table becomes like a tiny apartment complex. If more than one thing wants to move in, we just put them in a list, sort of like stacking up boxes. Now, each box holds one of our items, and it also knows where the next box is. It's like a chain of boxes.



So, when we need to find something, we just follow this chain of boxes. If we know which box we're looking for, we can quickly find it in its little stack. This makes things super efficient, even if a few items ended up in the same spot.

Here's the secret code we can use to do this:

- When we want to find something, we just look in the box at the front of the stack.
- If we want to add something new, we just put it in front of the stack.
- If we want to remove something, we just empty the box at the front of the stack.

Pseudocode for operations:

```
chainedHashSearch(T, k)
  return T[h(k)]
chainedHashInsert(T, x)
  T[h(x.key)] = x //insert at the head
chainedHashDelete(T, x)
  T[h(x.key)] = NIL
```

This chaining technique makes sure that our items always find a comfy home, even if they have to share a space sometimes.

---

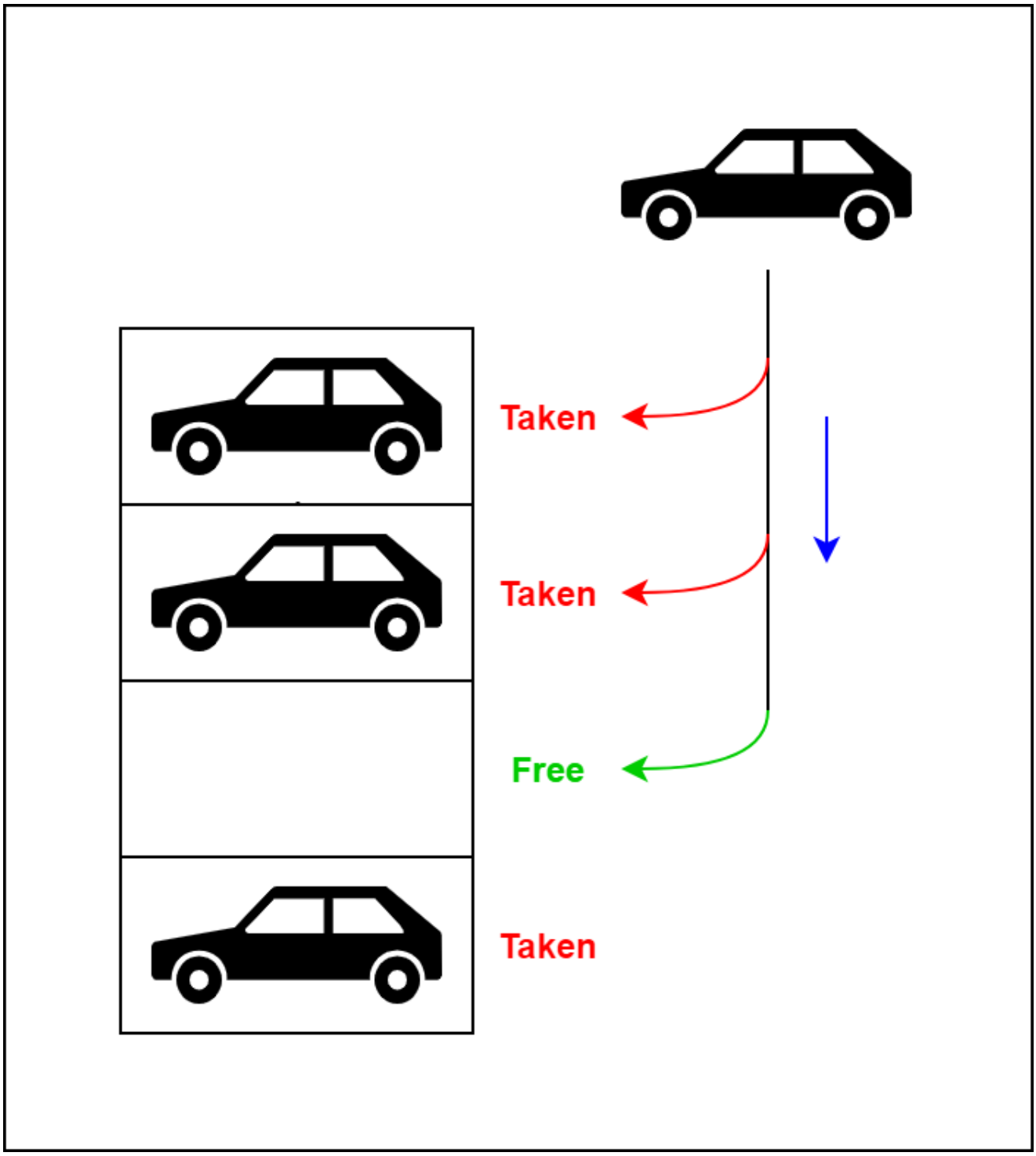
2. Open Addressing

Alright, let's switch gears and explore another approach to tackle hash collisions. This one is a bit different from chaining and involves a bit of a "finders, keepers" attitude.

Imagine we have our hash table, but this time, if our hash function says a spot is taken, we're not going to put the new item there. Instead, we'll keep looking for the next available spot until we find an empty one.

So, if one spot is occupied, we start moving forward, checking each slot until we find an empty one. This way, we're still sticking to our table, just moving along it until we find a free space. That's why they call it "open addressing."

Think of it like looking for a parking spot in a crowded parking lot. You drive along, checking each spot, until you find an empty one where you can park your car.



Now, there are a couple of ways to do this. You can simply keep moving forward and filling up the next empty spot you find. Or, you can follow a certain pattern to decide where to check next, like moving one step, two steps, or more.

So, to sum it up, open addressing is like trying to find an empty spot in a busy lot – you keep going until you find one. And in our case, when we need to find something, we use the same method to locate it.

Different techniques used in open addressing are:

1. Linear Probing
2. Quadratic Probing
3. Double Hashing

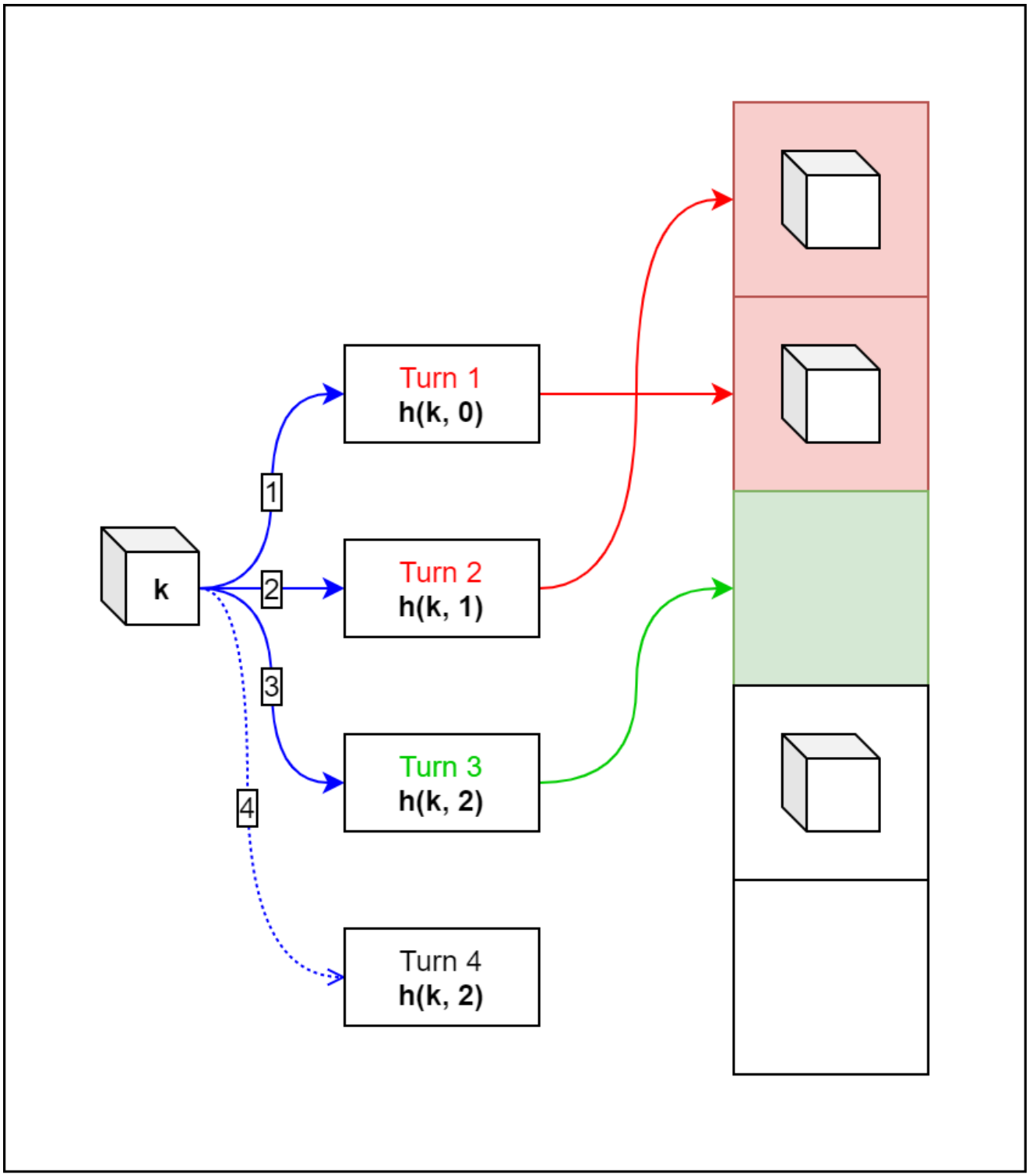
i. Linear Probing

let's dive into another approach called "linear probing." It's a bit like trying to find a parking spot again, but this time with a slightly different strategy.

Imagine our hash table is like a row of parking spots. When we want to park a car (or in this case, an item), we calculate its hash and try to put it in a spot. But, oh no, someone else's car is already in that spot! What do we do? Well, in linear probing, we just move to the next spot and try again.

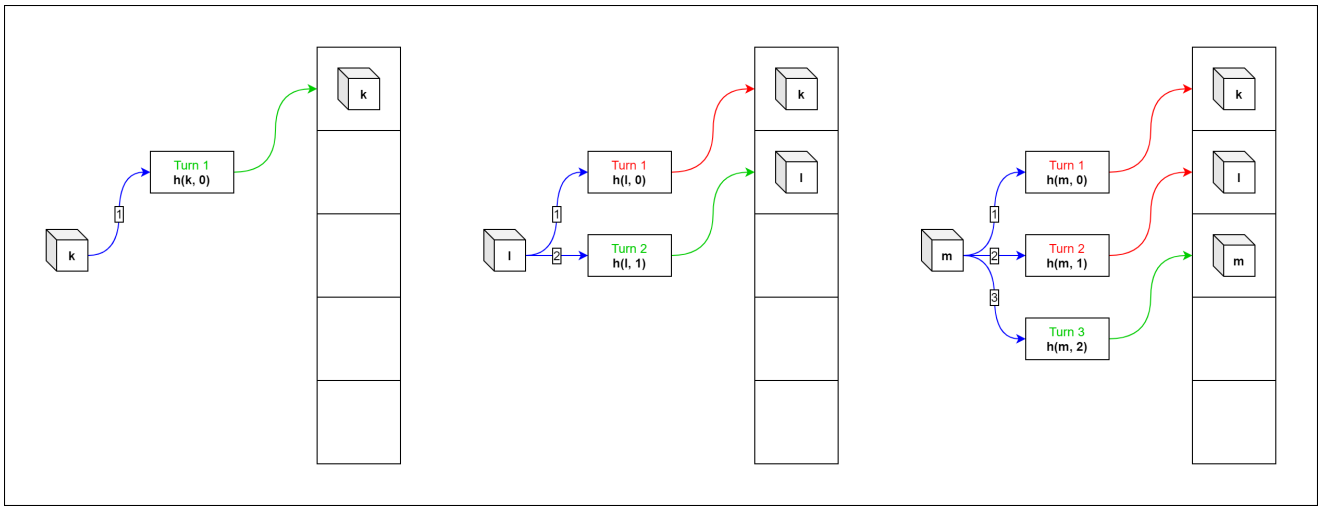
The formula `h(k, i) = (h'(k) + i) mod m` tells us exactly where to check next. Here, `i` is a counter that starts from 0 and goes up by 1 each time we need to move to a new spot. `h'(k)` represents another hash function we use to calculate a new index.

So, if the spot we calculated first is taken (that's `h(k, 0)`), we move to the next one (`h(k, 1)`), and then the next one after that (`h(k, 2)`), and so on. We keep going in this straight-line pattern until we find an empty spot.



But, here's the catch: Linear probing can sometimes cause a bunch of cars (or items) to be parked next to each other. Imagine if you had a row of cars in a parking lot, and everyone tried to park in the first empty spot they found. That would create a big line of cars, right? That's the cluster problem with linear probing – it can make a bunch of items sit next to each other, making it a bit slower to find an empty spot later.

So, while linear probing is quick to search and insert in many cases, it can sometimes lead to these clusters, which can slow things down a bit.



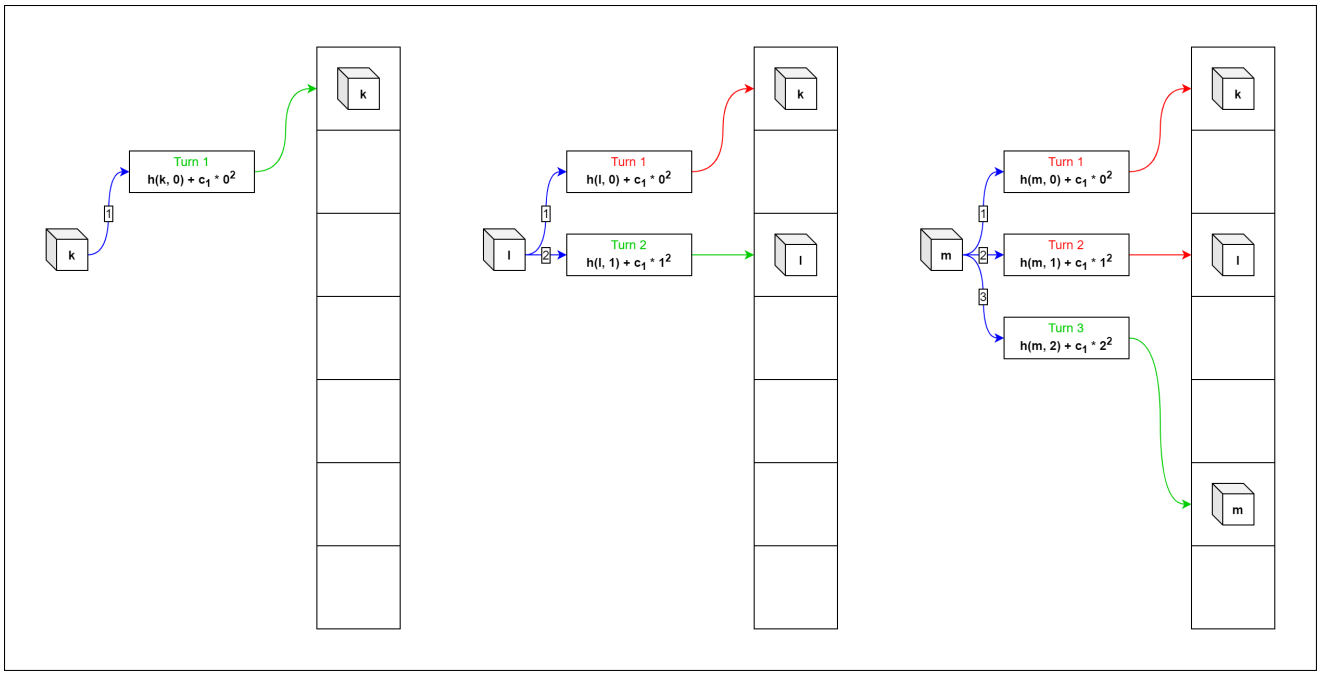
ii. Quadratic Probing

Alright, let's explore another strategy called "quadratic probing." Imagine we're trying to find a parking spot again, but this time with a different trick up our sleeves.

In quadratic probing, we still want to park our car (or item) in a certain spot in the parking lot (our hash table), just like before. But instead of moving to the next spot right away, we're going to space things out a bit.

Here's the formula we use to figure out where to check next: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$. I know it looks complicated, but don't worry – I'll break it down for you.

- $h'(k)$ is our trusty hash function.
- i is our counter, just like before, starting from 0 and increasing by 1 each time.
- c_1 and c_2 are positive numbers that help us space things out differently compared to linear probing.



So, let's say we're trying to park in a spot, but it's already taken (that's $h(k, 0)$). Instead of moving directly to the next spot, like we did in linear probing, we'll space things out a bit. We'll move to a

spot that's c_1 steps away ($h(k, 1)$), and then, if needed, to a spot that's c_2 steps away ($h(k, 2)$), and then $c_2 * 2$ steps away ($h(k, 3)$), and so on.

This spacing helps prevent those annoying clusters that can happen with linear probing. It's like saying, "Let's not all crowd in the same place!" We give the parking lot a bit more room to breathe.

So, quadratic probing is a bit like a more organized way of searching for a parking spot – we still move in a pattern, but we're not rushing to the next available space. We're giving the parking lot some space to avoid those annoying traffic jams of parked items.

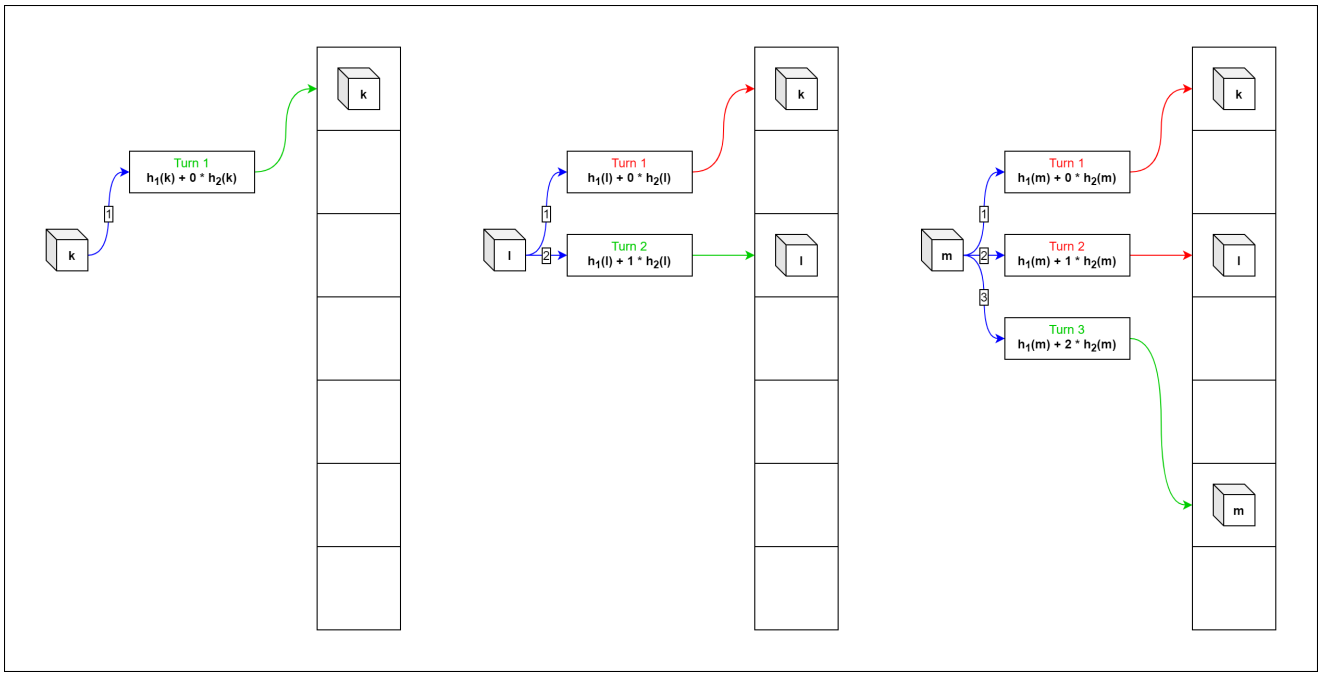
iii. Double hashing

Now, let's delve into a technique called "double hashing." Imagine you're at the parking lot again, but this time, you have not just one, but two clever strategies to find a parking spot.

In double hashing, we're going to apply a bit of a tag team approach. When our primary hash function doesn't lead us to a free spot, we'll call in the second hash function to help us out.

Here's the trick we'll use: $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Don't worry, I'll break it down for you.

- $h_1(k)$ is our main hash function that gives us the initial spot.
- i is our trusty counter again, starting from 0 and increasing by 1 each time.
- $h_2(k)$ is our second hash function that kicks in when we need help finding a spot.



So, let's say our primary hash function led us to a spot, but it's already occupied (that's $h(k, 0)$). Instead of just moving to the next spot right away, we call in our second hash function to guide us. The second hash function tells us how far to move from the initial spot – $ih_2(k)$ steps away.

And just like that, we've got a backup plan. If our primary strategy doesn't work, we use the second hash function to adjust our path. It's like having a buddy who gives you directions when you're a bit lost.

So, double hashing is a bit like teamwork in finding a parking spot. Our first hash function does its best, and if things get tricky, our second hash function steps in to save the day. With this approach, we're more likely to find a parking spot efficiently, even when the parking lot gets crowded.

Good Hash Functions

Now, let's talk about what makes a hash function "good." Just like how some tools are better at solving specific problems, some hash functions are better at preventing collisions – those tricky situations when two pieces of data end up in the same parking spot.

A good hash function might not be like a superhero that stops all collisions in their tracks, but it definitely knows how to make them happen less often. It's like having a magical sorting hat that tries its best to put students in their right houses at Hogwarts.

Imagine you have a magical spell – your hash function – that you cast on each data item. This spell should scatter the items across the available parking spots in such a way that they don't clump together too much. Just like throwing a handful of glitter into the air, you want the pieces to spread out and land in different places.

So, even though a good hash function can't completely prevent collisions, it's skilled at reducing them. It's like having a recipe that makes sure your chocolate chips are evenly distributed in your cookie dough.

In the world of hashing, a good hash function is a bit like a master chef who knows how to sprinkle just the right amount of seasoning to make the dish perfect. It might not make everything perfect, but it sure makes things a whole lot better!

Here, we will look into different methods to find a good hash function:

1. Division Method

Alright, let's dive into the first method of hashing called the "Division Method." This method uses a simple trick to find the parking spot for our data in the hash table.

Imagine we have a key, let's call it "k," and a hash table with "m" available spots. To figure out where our data belongs, we use a special formula: `h(k) = k mod m`.

Think of it like this – we're dividing the key "k" by the size of the hash table "m," and the remainder tells us where the data should go. It's kind of like sharing a pizza among friends – you divide the slices and see which one is left over.

For example, if we have a hash table with 10 spots and the key "k" is 112, we calculate `h(k) = 112 mod 10`, which equals 2. So, the data with the key 112 goes into the spot number 2 of our hash table.

But there's a little twist – the size of the hash table "m" shouldn't be a power of 2 like 2, 4, 8, and so on. This is because when we calculate "k mod m," we usually end up with the lower bits of "k," which might not be evenly distributed.

So, if "m" is 22 and "k" is 17, our formula becomes `h(k) = 17 mod 22 = 10001 mod 100 = 01`. Similarly, if "m" is 23, we get `h(k) = 17 mod 23 = 10001 mod 100 = 001`.

In cases where "m" is 2 to the power of "p," like 2, 4, 8, and so on, the formula simply takes the lower "p" bits of "k" as the hash value. It's like picking out only a specific part of your phone number.

```
if m = 22, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 01
if m = 23, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 001
if m = 24, k = 17, then h(k) = 17 mod 22 = 10001 mod 100 = 0001
if m = 2p, then h(k) = p lower bits of m
```

So, the division method gives us a quick way to find a parking spot for our data in the hash table based on the remainder of the division. Just remember, for this method, avoid using hash table

sizes that are powers of 2 to ensure a more even distribution of data.

2. Multiplication Method

Alright, let's continue our journey through hashing by exploring the "Multiplication Method." This method involves a bit of mathematical magic to find the right spot for our data in the hash table.

The formula we use here is: `h(k) = ⌊m(kA mod 1)⌋`.

Now, what's all this about? Let's break it down step by step.

First, we have our key "k" and a constant "A." This "A" can be any number between 0 and 1. But there's a special value that works pretty well, and it's about (√5-1)/2. Just think of "A" as a seasoning that we'll sprinkle on our key.

Now, we want to find a slot in our hash table. We're going to do some math: "kA mod 1." This might look a bit strange, but all it's doing is taking the fractional part of "kA." Just like when we divide a pizza slice in half and take the part that's left – that's the fractional part.

Now comes the "⌊ ⌋" symbol, which means we're taking the floor value. Imagine you have a floating point number, like 3.8. The floor value is like the biggest whole number that's less than or equal to 3.8 – and that's 3.

So, putting it all together, the formula is saying: take the fractional part of "kA," multiply it by "m" (which is the size of our hash table), and then round it down to the nearest whole number. This gives us the perfect spot for our data in the hash table.

Why does this work? Well, it's a bit like doing a spin with a compass to find your way – the "kA mod 1" helps spread the values out, and the floor operation ensures we get a nice whole number index.

So, that's the multiplication method – a bit of math and a sprinkle of a constant "A" to find the right place for our data in the hash table. It's like following a treasure map with a few clever twists!

3. Universal Hashing

Alright, let's dive into "Universal Hashing." It's a cool concept that involves a bit of randomness to make our hash functions even more versatile.

So, imagine you're playing a game where you need to decide how to arrange your toys on a shelf. Universal hashing is like rolling a dice to figure out how to arrange them. You choose a hash function at random, without even knowing what the keys (or toys) are. This randomness helps spread things out and prevents any particular pattern from forming.

Now, why would we want to do this? Well, think about it. If we always use the same hash function, and our data happens to have a specific pattern, we might end up with some slots getting crowded while others stay empty. That's not very efficient.

But with universal hashing, it's like changing the rules of the game every time you play. This way, your toys (or keys) are scattered around the shelf (or hash table) in a more unpredictable way.

So, to sum it up, universal hashing is like adding a sprinkle of randomness to our hash functions. It helps keep things balanced and fair, just like rolling a dice to decide where your toys go on the shelf. And that's how we keep our hash tables in great shape!

C++ Example

```
// Implementing hash table in C++

#include <iostream>
#include <list>
using namespace std;

class HashTable
{
  int capacity;
  list<int> *table;

public:
  HashTable(int V);
  void insertItem(int key, int data);
  void deleteItem(int key);

  int checkPrime(int n)
  {
  int i;
  if (n == 1 || n == 0)
  {
    return 0;
  }
  for (i = 2; i < n / 2; i++)
  {
    if (n % i == 0)
    {
    return 0;
    }
  }
  return 1;
  }
  int getPrime(int n)
  {
  if (n % 2 == 0)
  {
    n++;
  }
  while (!checkPrime(n))
  {
    n += 2;
  }
  return n;
```

```
  }

  int hashFunction(int key)
  {
  return (key % capacity);
  }
  void displayHash();
};
HashTable::HashTable(int c)
{
  int size = getPrime(c);
  this->capacity = size;
  table = new list<int>[capacity];
}
void HashTable::insertItem(int key, int data)
{
  int index = hashFunction(key);
  table[index].push_back(data);
}

void HashTable::deleteItem(int key)
{
  int index = hashFunction(key);

  list<int>::iterator i;
  for (i = table[index].begin();
   i != table[index].end(); i++)
  {
  if (*i == key)
    break;
  }

  if (i != table[index].end())
  table[index].erase(i);
}

void HashTable::displayHash()
{
  for (int i = 0; i < capacity; i++)
  {
  cout << "table[" << i << "]";
  for (auto x : table[i])
    cout << " --> " << x;
  cout << endl;
  }
}

int main()
{
  int key[] = {231, 321, 212, 321, 433, 262};
  int data[] = {123, 432, 523, 43, 423, 111};
  int size = sizeof(key) / sizeof(key[0]);
```

```
  HashTable h(size);

  for (int i = 0; i < size; i++)
  h.insertItem(key[i], data[i]);

  h.deleteItem(12);
  h.displayHash();
}
```

Applications of Hash Table

Let's explore the various ways we use hash tables in the real world. Hash tables are like magical tools that help us organize and find things super quickly.

- First off, imagine you have a huge library, and you want to find a specific book without spending hours searching. Hash tables come to the rescue! They're like librarian wizards that can instantly tell you where the book is located. This is why we use hash tables when we need to find things super fast, like looking up a word in a dictionary.

- Now, let's talk about secrets. Hash tables are like secret vaults. When you want to keep something secure, like your password or credit card information, hash tables help encrypt it. It's like turning your information into a secret code that only the right key can unlock. So, in the world of computers, hash tables are used in cryptography to protect sensitive data.

- Lastly, think about organizing your stuff. Imagine you have a collection of stamps, and you want to find a specific one without flipping through every page. Hash tables work wonders here too! They help us index and organize data efficiently. Just like how you'd use an index in the back of a book to find a chapter quickly.

So, hash tables are like your personal organizers, fast finders, and secret keepers in the digital world. They help us find, secure, and sort data in the blink of an eye!