
C
CodeIT

Lesson 1 Objectives:

To gain an understanding of:

1. What are and why we need the **hash table** data structure
2. How to **analyse and implement** hash tables
3. How to **minimise the disadvantages** of hash tables and handle hash collisions

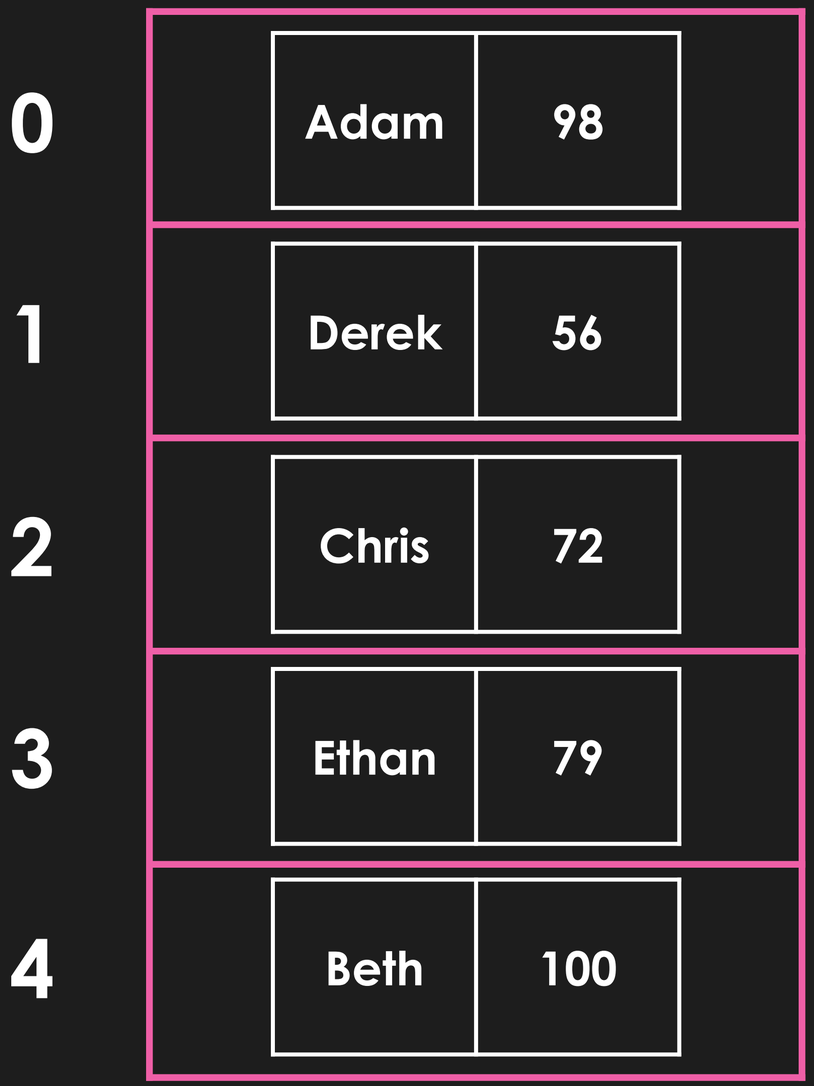
EXAM SCORES

username	test score
Adam	98
Derek	56
Chris	72
Ethan	79
Beth	100
Dylan	89

Imagine we had a sample from a list of students and their test scores like so

We want to make it such that each student is able to quickly find their name in a huge list, with up to 1000 students!

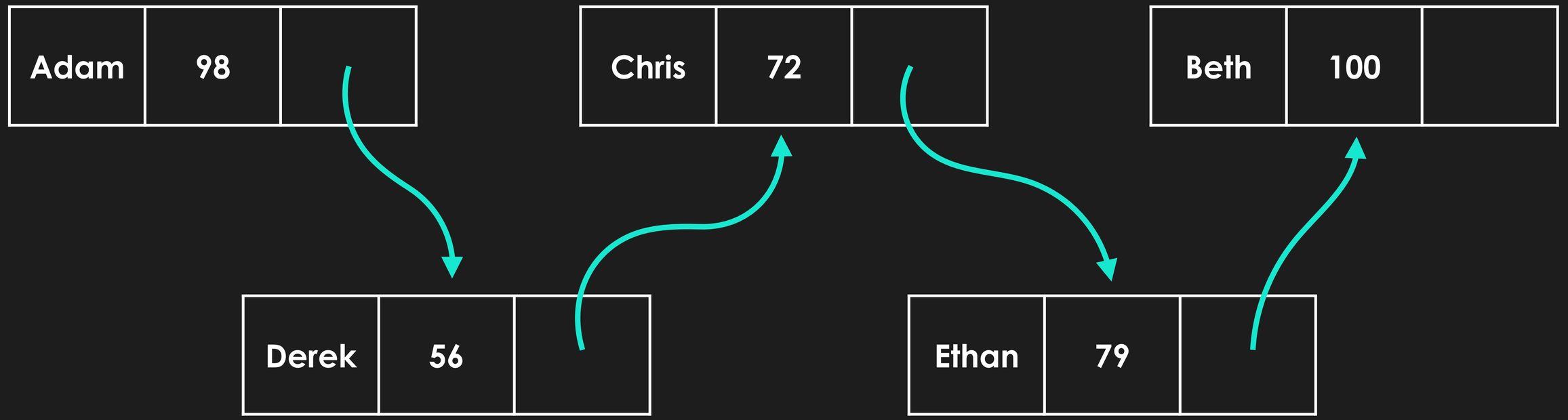
Example 1: Data in 2D Lists



SEARCH: O(N)

INSERT: O(N)

Example 2: Linked Lists

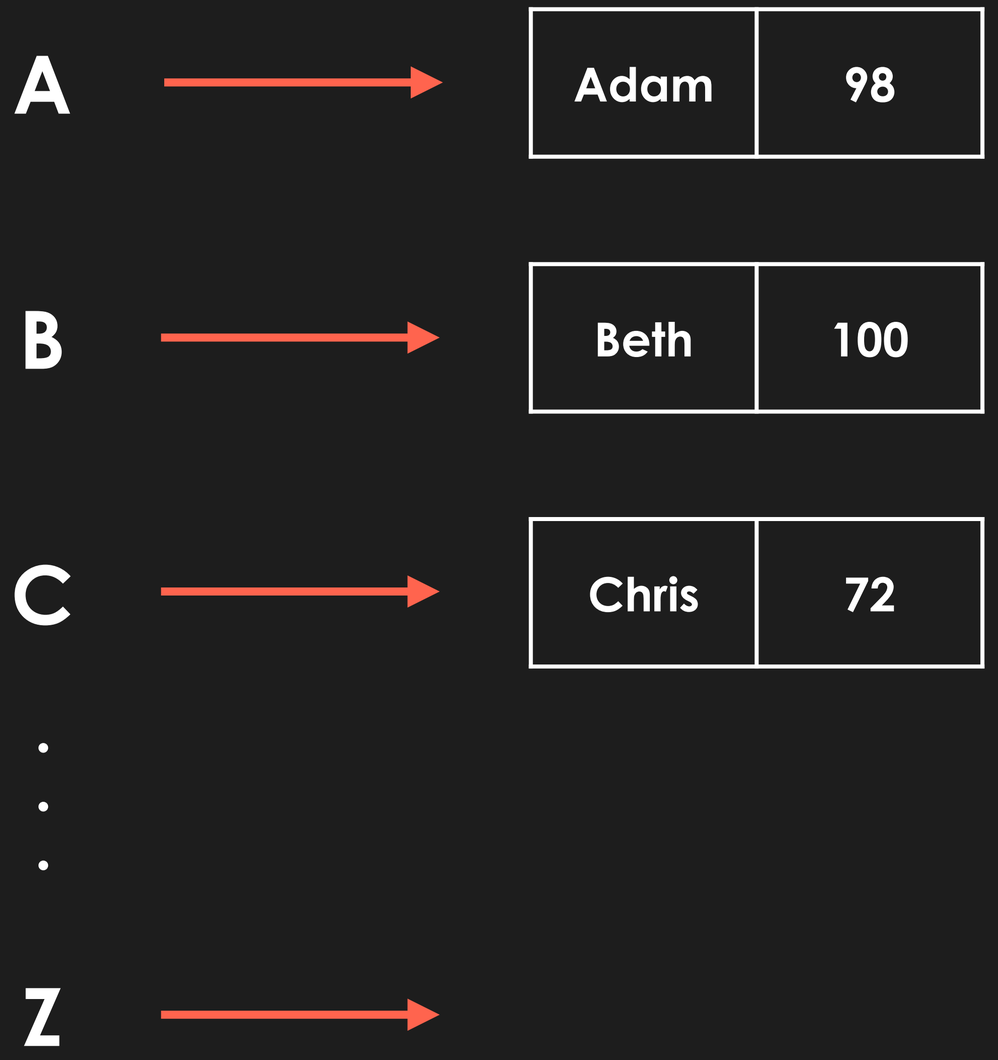


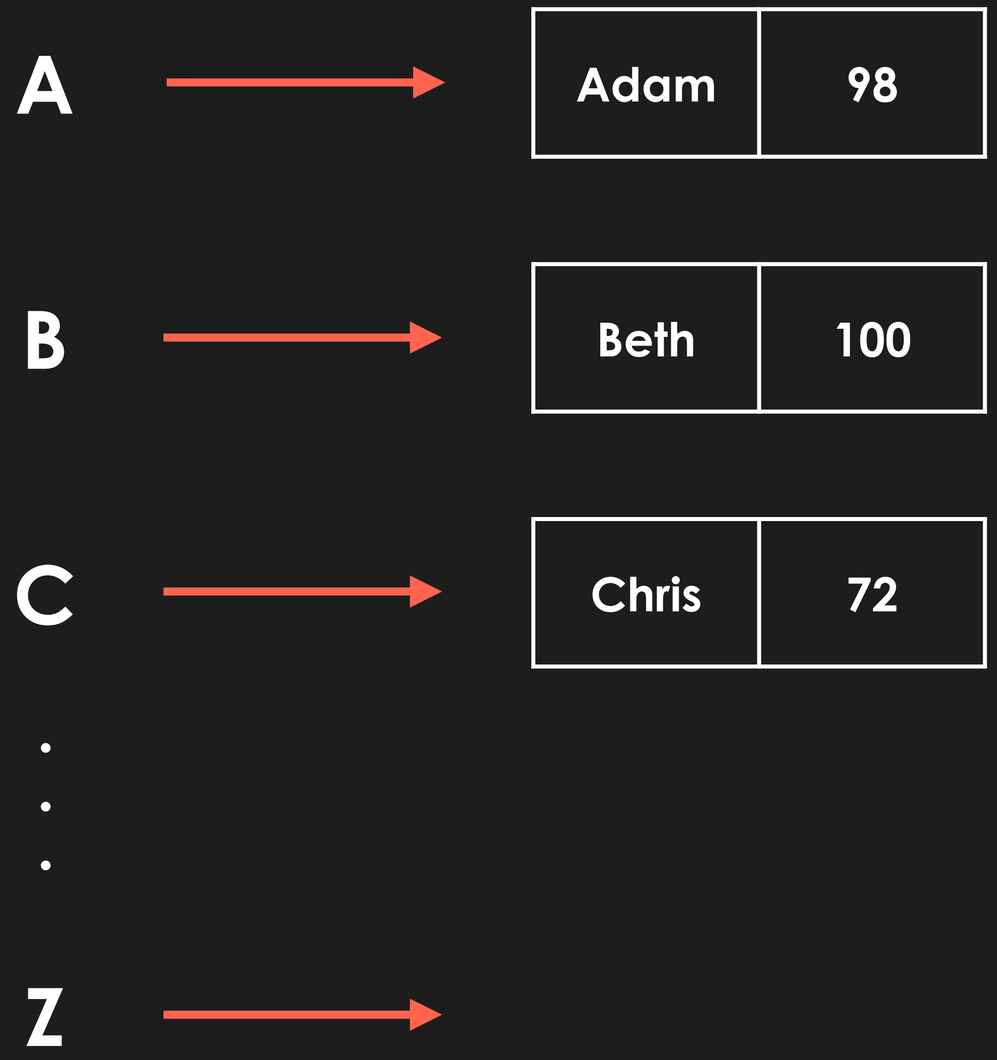
SEARCH: O(N)

INSERT: O(1)

Both cases we have seen are linear, unordered structures, and if a student were to search for his name in such a list, it might be quite difficult! Can we do better?

What if we created a list specifically for each letter in the alphabet, and then populated each list with a student whose letter started with the same alphabet


A
Adam
98
B
Beth
100
C
Chris
72
Z



Now, students can simply look at the first letter of their name, and then easily find their score

Let's see how we can implement this in code!

main

```
scores = [None] * 26
insertStudent(scores, "Adam", 98)
insertStudent(scores, "Derek", 56)
insertStudent(scores, "Chris", 72)
insertStudent(scores, "Ethan", 79)
```

First, we create a list with **26 indices (one for each alphabet)**, and then call the **insertStudent** function for each student to insert into this list

insertStudent

```
def insertStudent(scores, studentName, studentScore):
    index = convertToIndex(studentName)
    scores[index] = [studentName, studentScore]
```

Our `insertStudent` looks like so:

- It takes in our scores list, `studentName` and `studentScore`
- It then converts the `studentName` to the appropriate `index` and stores it in scores as a 2D list!

insertStudent

```
def convertToIndex(studentName):
    firstLetter = studentName[0].lower()
    index = ord(firstLetter) - 97
    return index
```

Our `convertToIndex` looks like so:

- It takes in our `studentName`
- It uses the ASCII encoding of the studentName and converts it to the appropriate index for it's first letter!

Now, we need a way to access students' score via their name. Let's create a function for that!

getScore

```
def getScore(scores, studentName):
    index = convertToIndex(studentName)
    return scores[index]
```

Our `getScore` simply:

- Uses the convertToIndex function to get the correct index for our studentName
- Accesses that index and returns the value in our scores list

Result

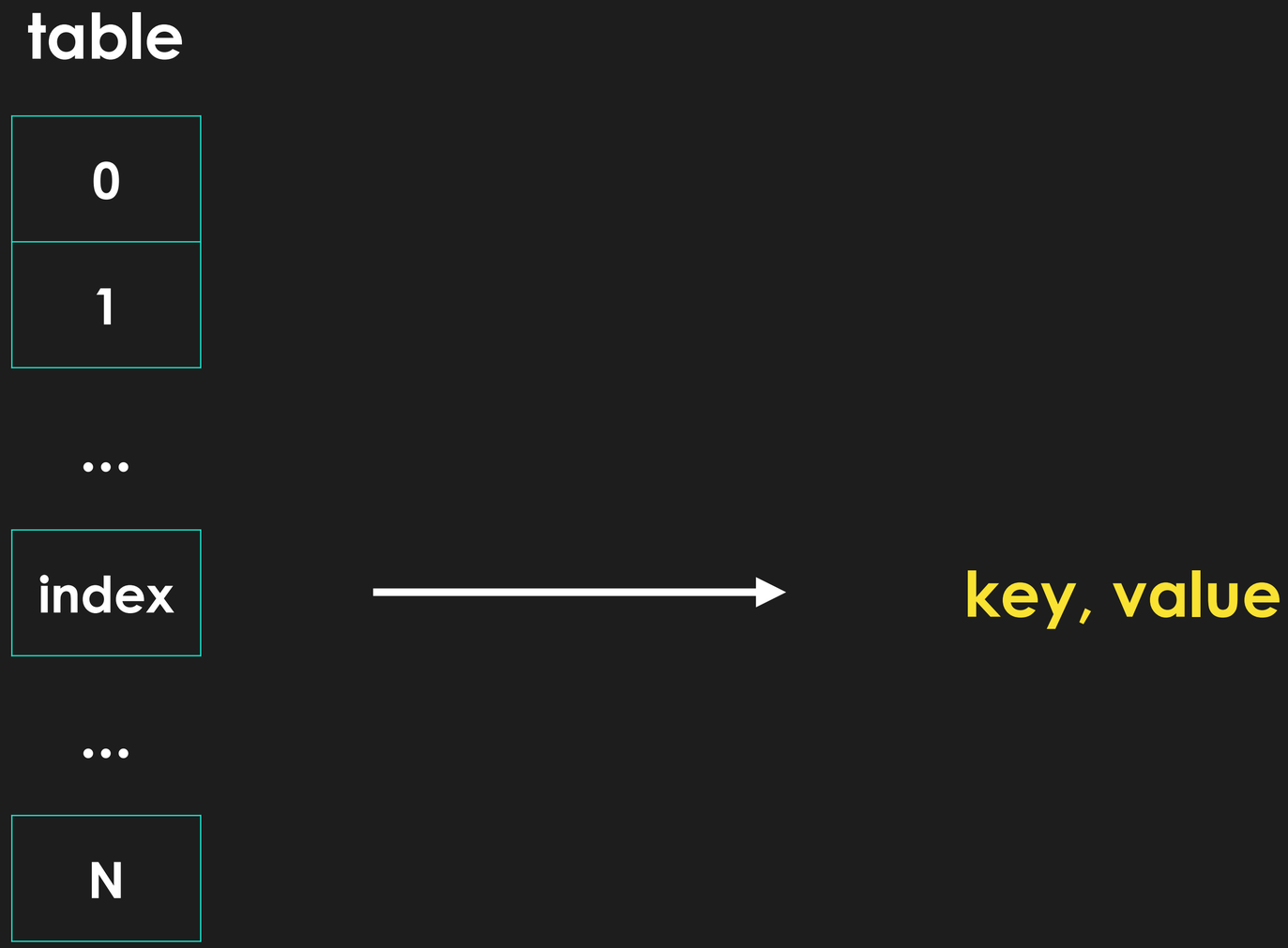
```
print(getScore(scores, "Ethan"))
```

```
['Ethan', 79]
```

What we have essentially done is created a
Hash Table

Hash Tables

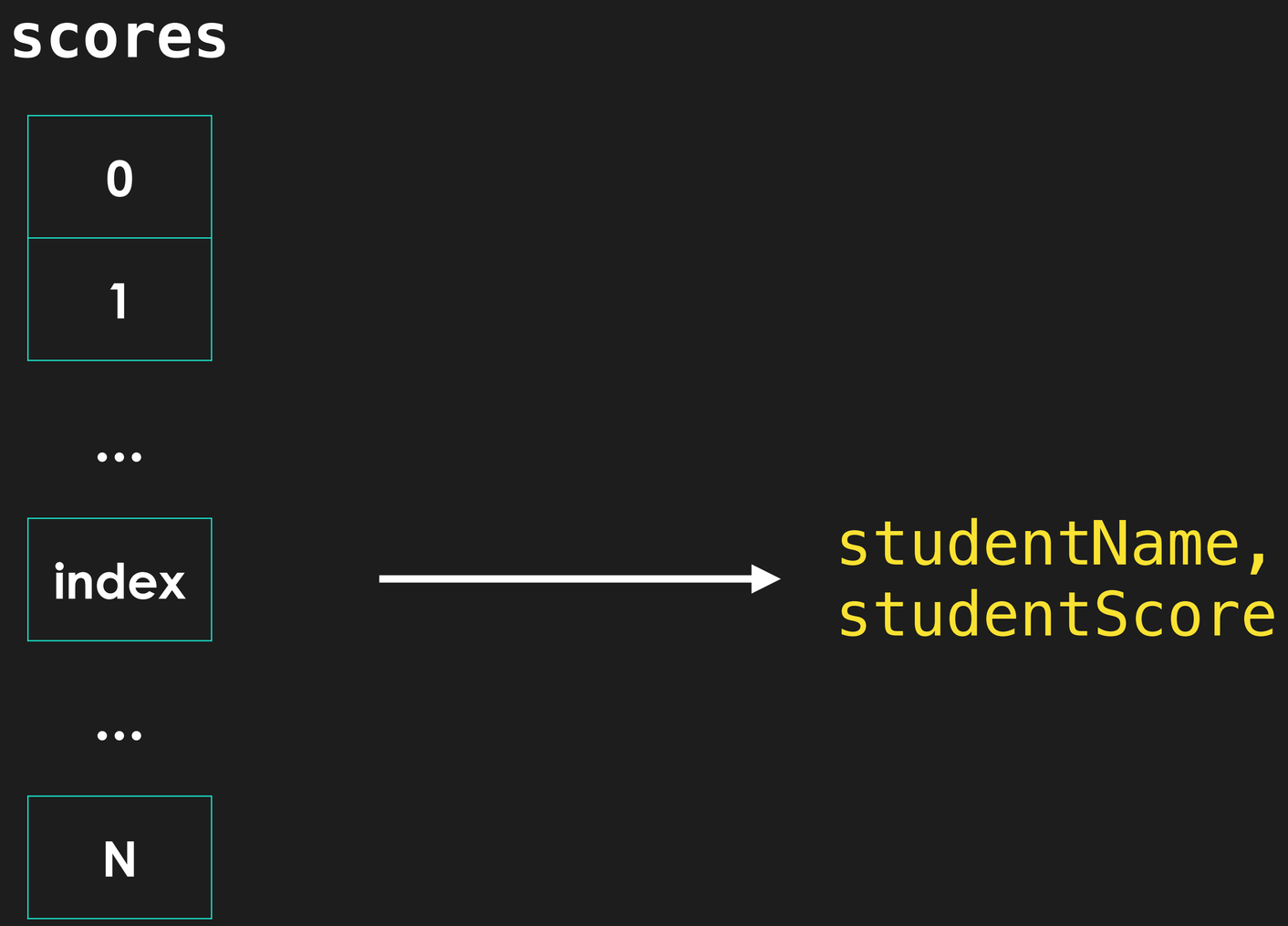
Hash Tables



Hash Tables in the context of scores

Hash Table

Hash Table: Search



Implementation of HashTable

HashItem

```
class HashItem:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

We first define a **HashItem** with attributes of key & value to make things easier for our HashTable

HashTable

```
class HashTable:

    def __init__(self):
        self.table = [None] * 26

    def hash(key):
        firstLetter = key[0].lower()
        index = ord(firstLetter) - 97
        return index
```

We then create a **HashTable** class with:

- a simple size 26 list as it's table, one for each alphabet
- a hash method that converts a key to it's appropriate index

HashTable

```
    def insert(self, key, value):
        index = hash(key)
        self.table[index] = HashItem(key, value)

    def search(self, key, value):
        index = hash(key)
        return self.table[index]
```

We can now define our respective insert and search functions!

Lab Session 1

Implement **hash_table.py**

Use **hash_test.py** to test

Collision in HashTables

Collision in HashTables

By right, if our hash function distributes all items uniformly, such that each index in our table only has one item, then the time complexity for search would be **O(1)**

However, in a dynamic data set that is ever changing, this is impossible because you are unable to account for **unseen keys** that have yet to enter!

Note:

- A hash function that can generate no collisions is called a "perfect hash function", but may not be uniform

- If the hash function is 1-to-1 and has no wasted space then the hash function is said to be "minimally perfect"

Consider the following:

```
scores.insert("Derek", 56)
scores.insert("Dylan", 89)
```

Consider the following:

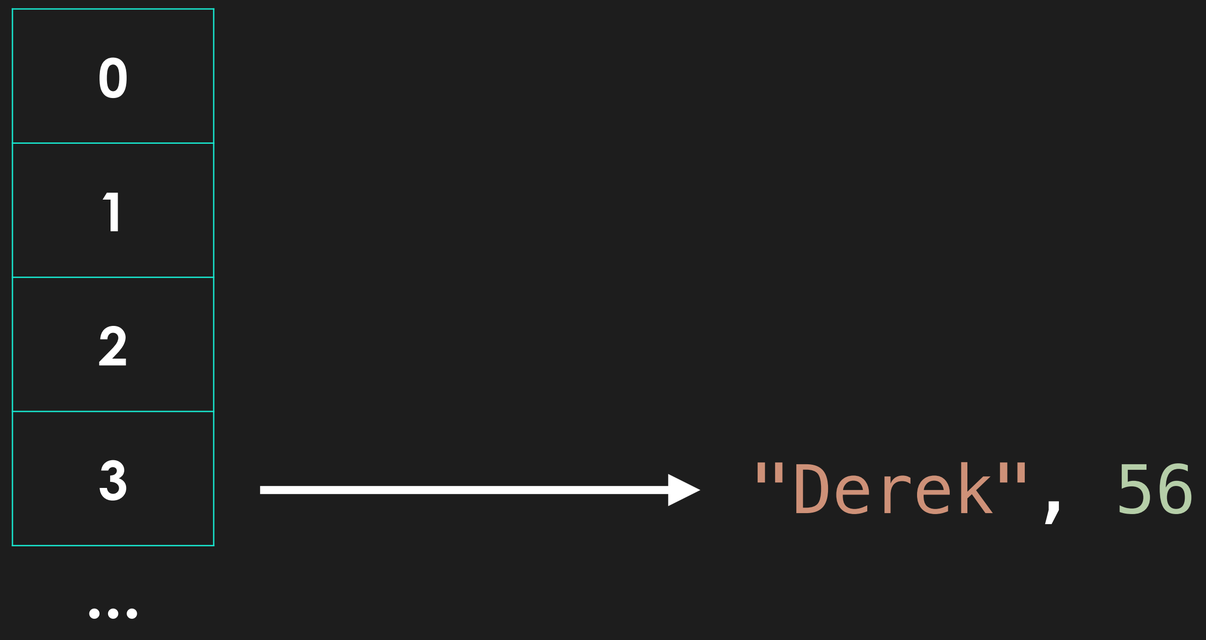
```
scores.insert("Derek", 56)
scores.insert("Dylan", 89)
```

Both Derek & Dylan are going to have the same hash index. As such, this will cause what we call a collision!

Based on previous example

```
scores.insert("Derek", 56)
```

```
scores
```



Based on previous example

```
scores.insert("Dylan", 89)
```

```
scores
```

Note: In general, we try to avoid collision through creating complex hash functions that generate unique keys.

Consider the following two hash functions

Hash Function 1

```
def hashToIndex1(key):
    firstLetter = key[0].lower()
    index = ord(firstLetter) - 97
    return index
```

Hash Function 2

```
def hashToIndex2(key):
    index = 0
    for i in key:
        index += ord(i)
    return index % 100
```

Hash Function 2 seems much more avoidant of collisions. However, the problem is that no matter how complex the hash function, collisions can still occur:

```
hashToIndex2("JO") -> 53

hashToIndex2("IP") -> 53
```

Note: Additionally, we can also create more space for keys in our hash table. However, this may come at a cost of memory space!

Solutions to Hash Table collisions

Solution 1: Chaining

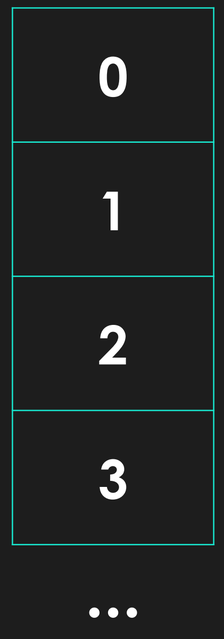
Solution 1: Chaining

Each key in our hash table will be assigned a linked list

Example of Hash Table with chaining

```
insert: "Adam", 98
```

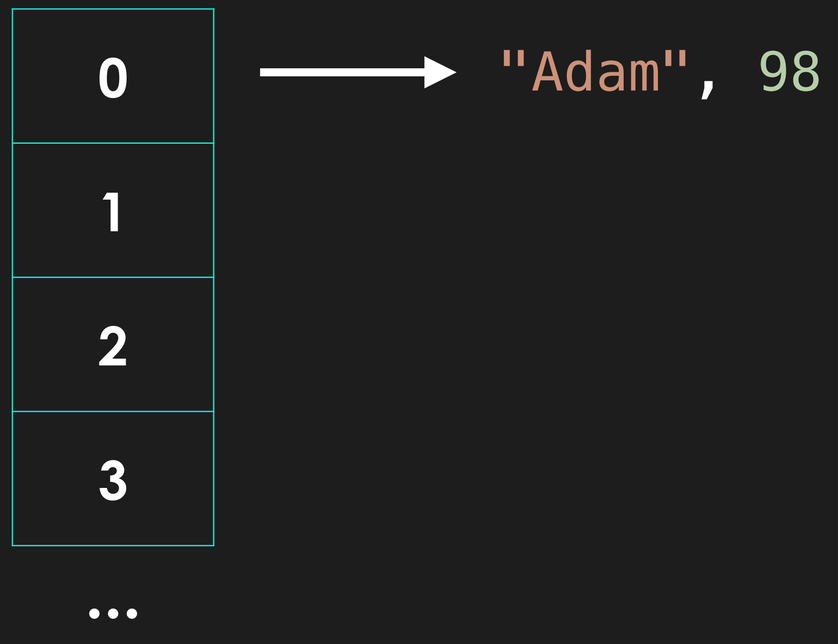
scores



Example of Hash Table with chaining

```
insert: "Adam", 98
```

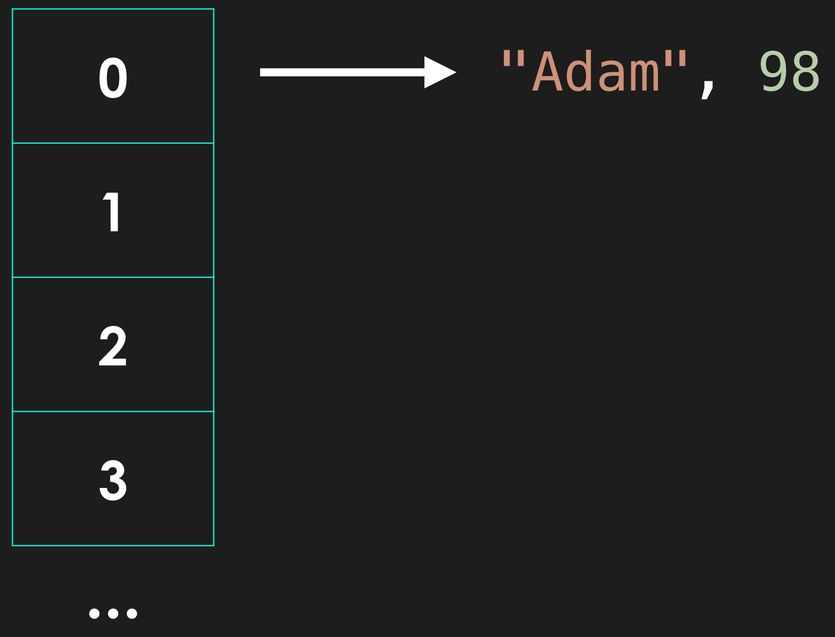
```
scores
```



Example of Hash Table with chaining

```
insert: "Alice", 93
```

scores



Example of Hash Table with chaining

```
insert: "Alice", 93
```

scores

Example of Hash Table with chaining

```
insert: "Derek", 56
```

scores

Example of Hash Table with chaining

```
insert: "Derek", 56
```

scores

Example of Hash Table with chaining

```
insert: "Beth", 100
```

scores

Example of Hash Table with chaining

```
insert: "Beth", 100
```

scores

Example of Hash Table with chaining

```
insert: "Dylan", 89
```

scores

Example of Hash Table with chaining

```
insert: "Dylan", 89
```

scores

Example of Hash Table with chaining

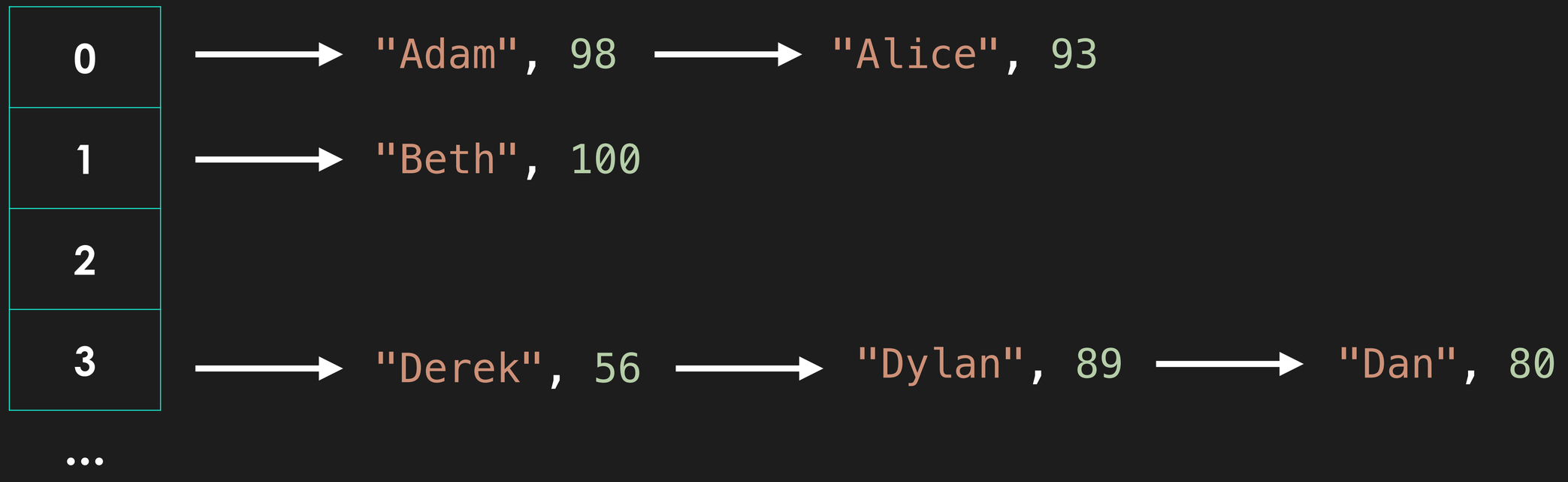
```
insert: "Dan", 80
```

scores

Example of Hash Table with chaining

```
insert: "Dan", 80
```

scores



Implementation

```
class HashTable:

    def __init__(self):
        self.table = [[] for i in range(26)]

    def hashToIndex(self, key):
        firstLetter = key[0].lower()
        index = ord(firstLetter) - 97
        return index
```

- Each item in our table is now a list rather than None

- We use python lists rather than actual linked lists for simplicity. However, the chaining method relies on linked lists due to constant time insertion and deletion

```
    def insert(self, key, value):
        index = self.hashToIndex(key)
        self.table[index].append(HashItem(key, value))

    def getHashItem(self, key):
        index = self.hashToIndex(key)
        keyList = self.table[index]
        for item in keyList:
            if item.key == key:
                return item
```

- Our insert method now appends to our list at the key's index

- When we search for a key now, we must iterate through that key's list to search for the correct key. This is so we don't retrieve a wrong hash item

Time Complexity of Hash Table with chaining

Time Complexity of Hash Table with chaining

Average Case:

- **Insertion: O(1)**
- **Search: O(1)**

What if our keys looked like so:

username	test score
Derek	98
Dylan	56
Daniel	72
Dan	79
Diego	100

Our table will then become a linked list!

Time Complexity of Hash Table with chaining

Average Case:

- **Insertion: O(1)**
- **Search: O(1)**

Worst Case:

- **Insertion: O(1)**
- **Search: O(N)**

Solution 2: Linear Probing (Open Addressing)

Solution 2: Linear Probing (Open Addressing)

If we encounter a key that is already taken, we move it over to the next slot. If all slots taken, we resize our HashTable

Consider this HashFunction

```
def hashToIndex(key, size):
    firstLetter = key[0].lower()
    index = ord(firstLetter) - 97
    return index % size
```

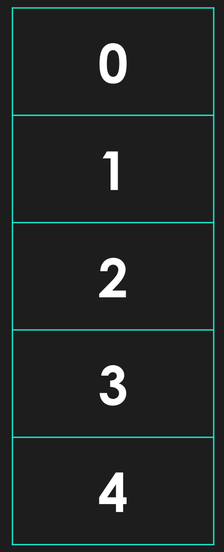
Once again, our hash function converts our key based on its **first letter** to a number between 0 - 25. However, this time, it **mods the result** to get a number within the **bounds of size**

Let's start our HashTable at **size 5** initially

Example of linear probing

```
insert: "Adam", 98
```

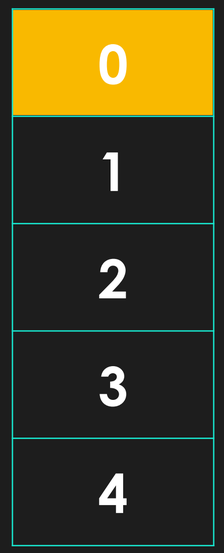
scores



Example of linear probing

```
insert: "Adam", 98
```

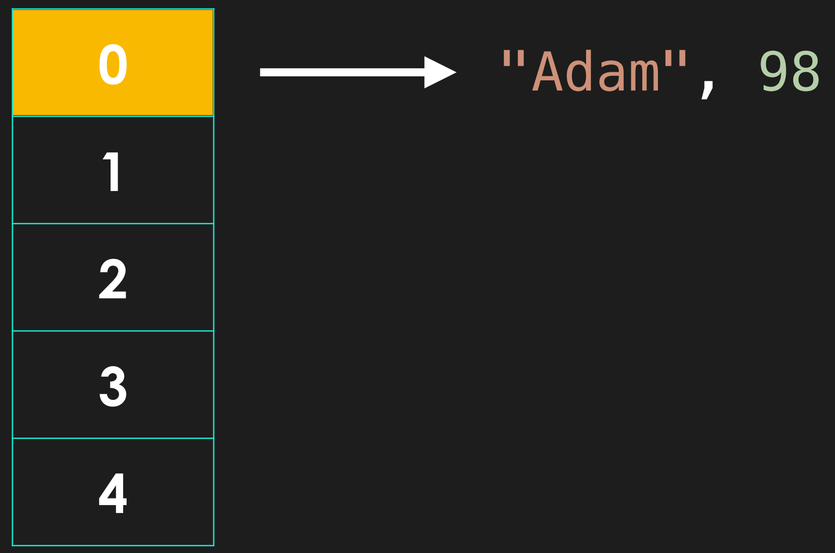
scores



Example of linear probing

```
insert: "Adam", 98
```

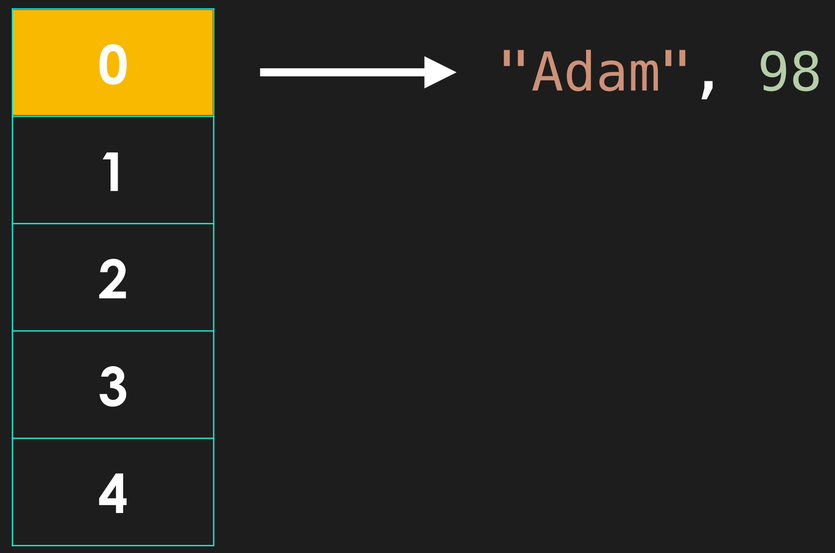
```
scores
```



Example of linear probing

```
insert: "Alice", 93
```

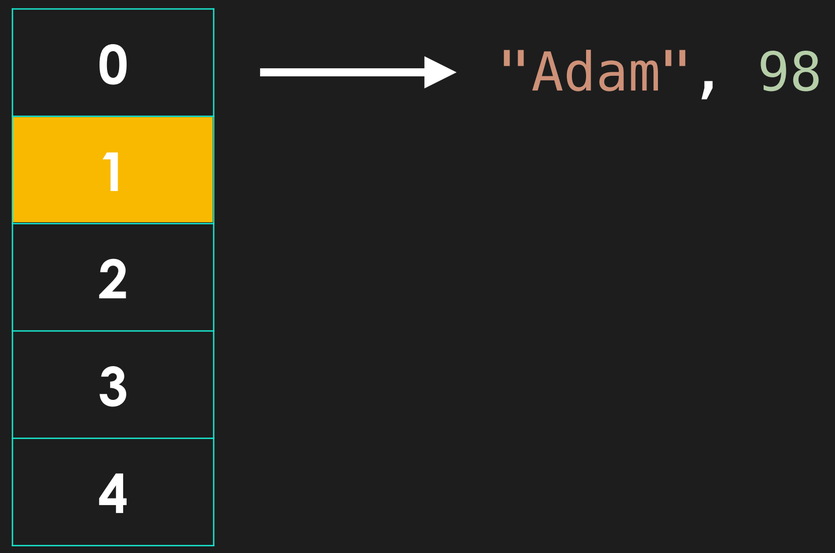
```
scores
```



Example of linear probing

```
insert: "Alice", 93
```

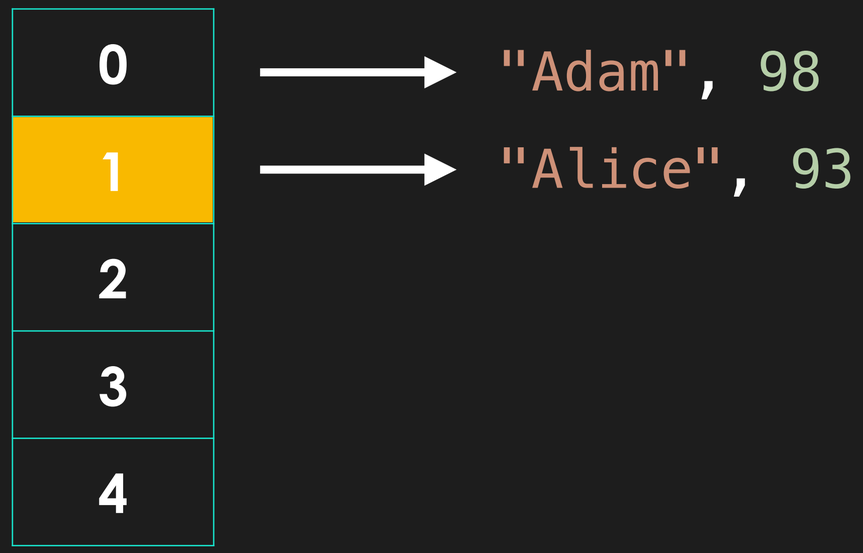
scores



Example of linear probing

```
insert: "Alice", 93
```

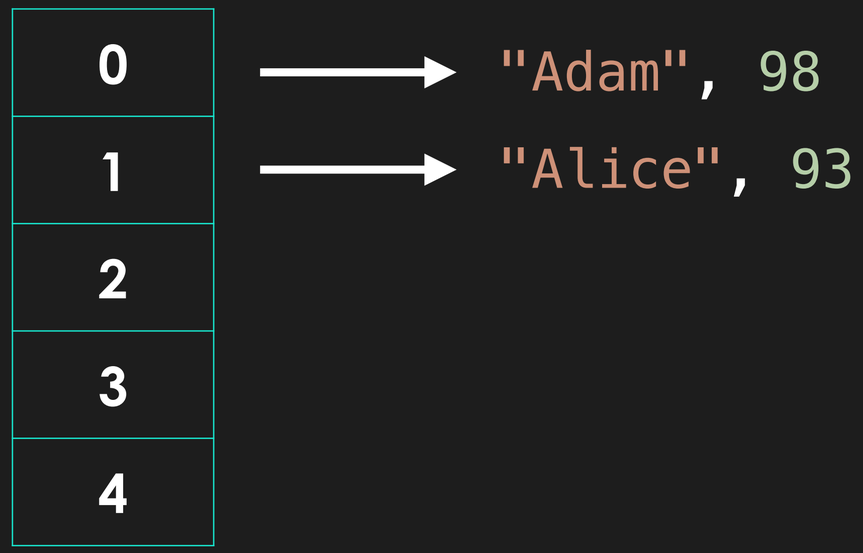
```
scores
```



Example of linear probing

```
insert: "Beth", 100
```

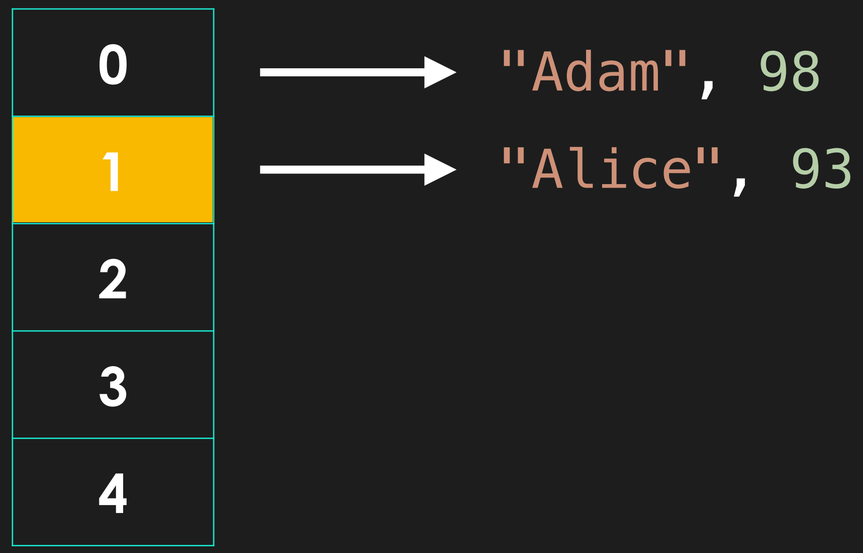
scores



Example of linear probing

```
insert: "Beth", 100
```

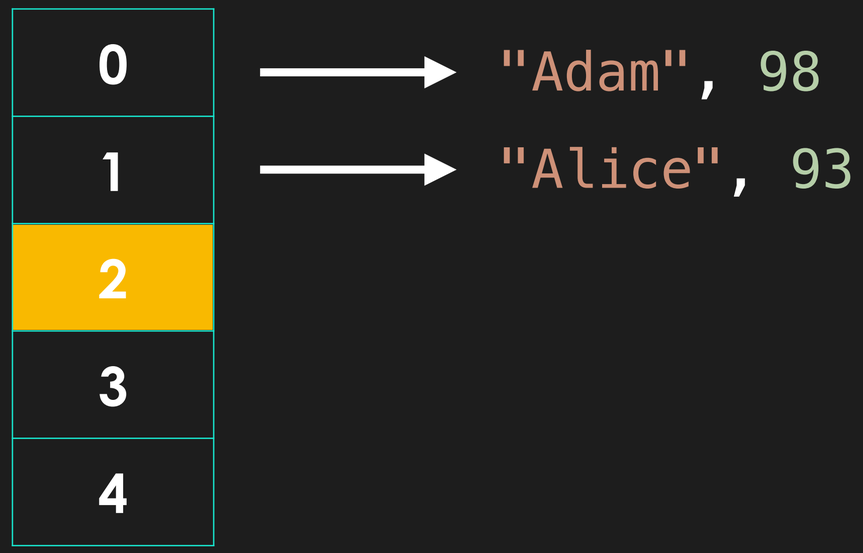
```
scores
```



Example of linear probing

```
insert: "Beth", 100
```

scores



Example of linear probing

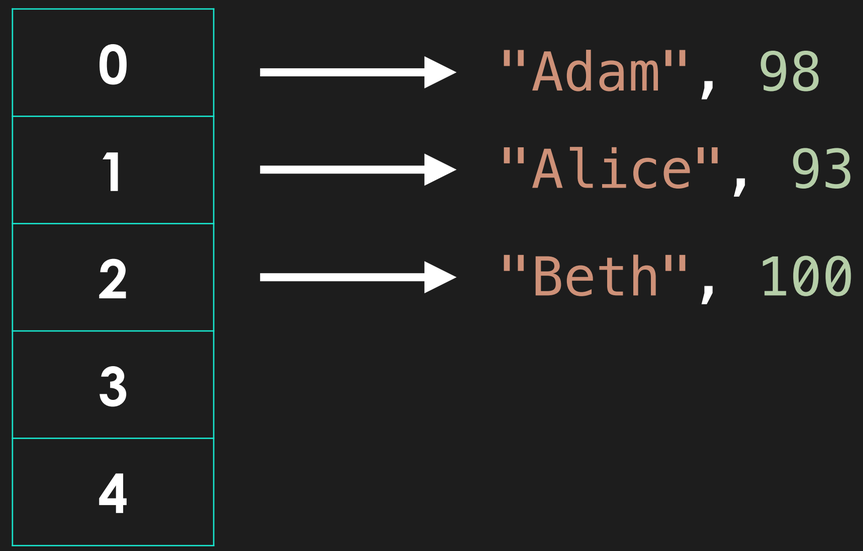
```
insert: "Beth", 100
```

scores

Example of linear probing

```
insert: "Esther", 89
```

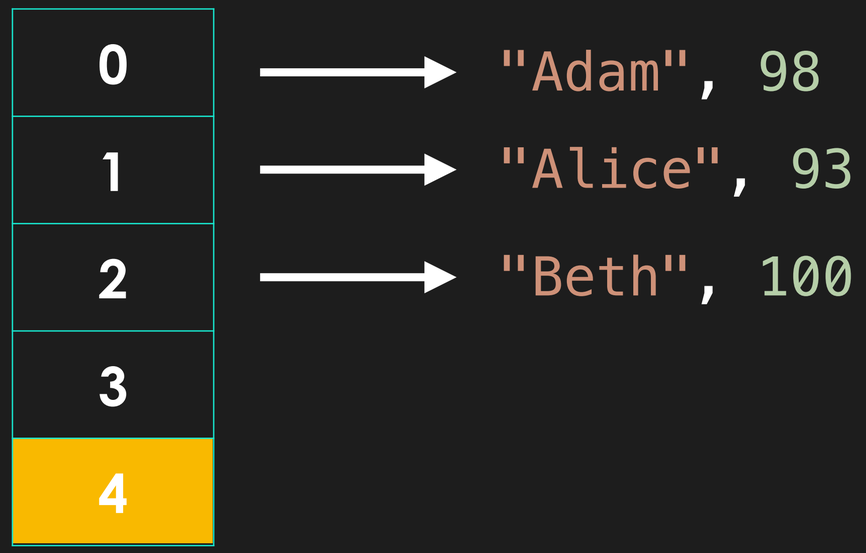
scores



Example of linear probing

```
insert: "Esther", 89
```

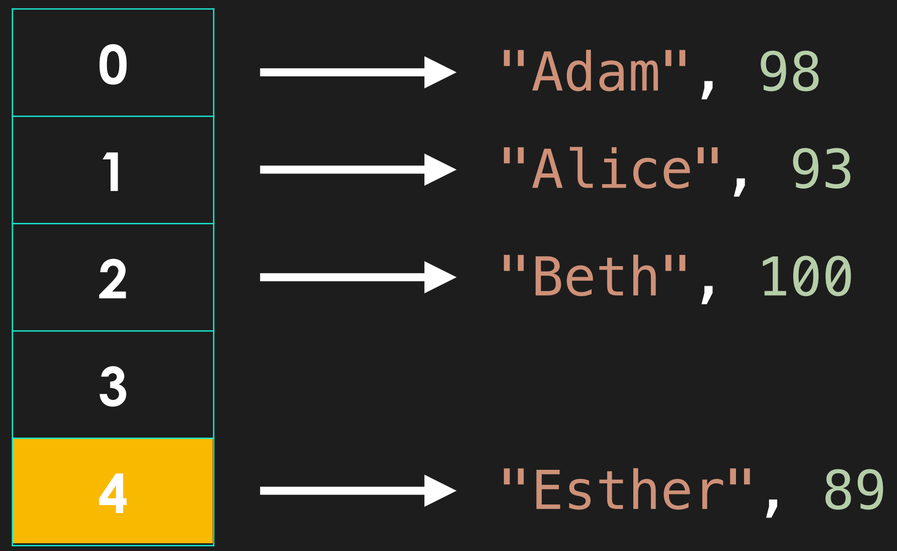
scores



Example of linear probing

```
insert: "Esther", 89
```

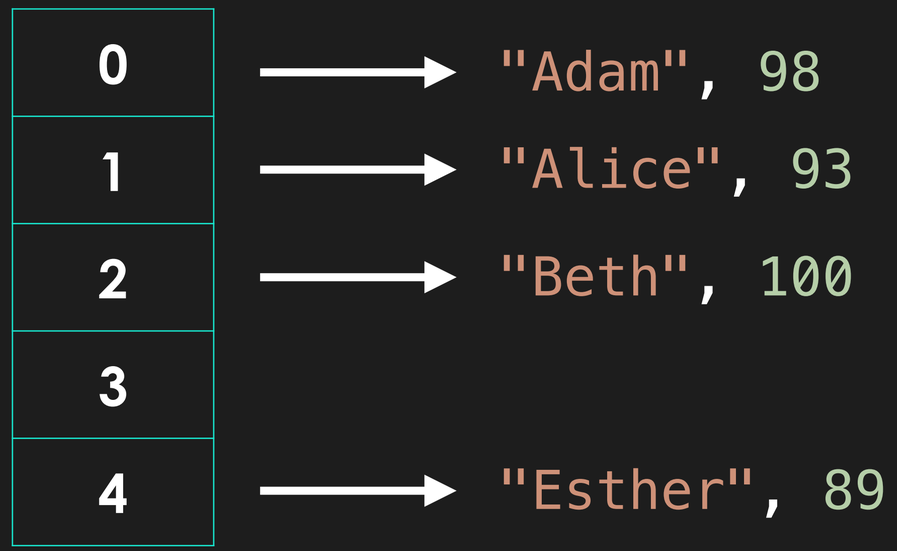
scores



Example of linear probing

```
insert: "Ethan", 72
```

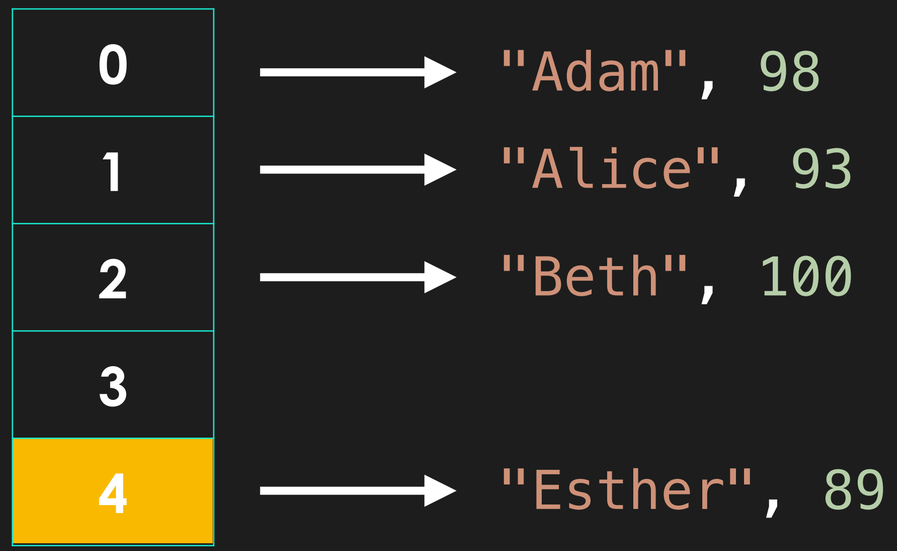
scores



Example of linear probing

```
insert: "Ethan", 72
```

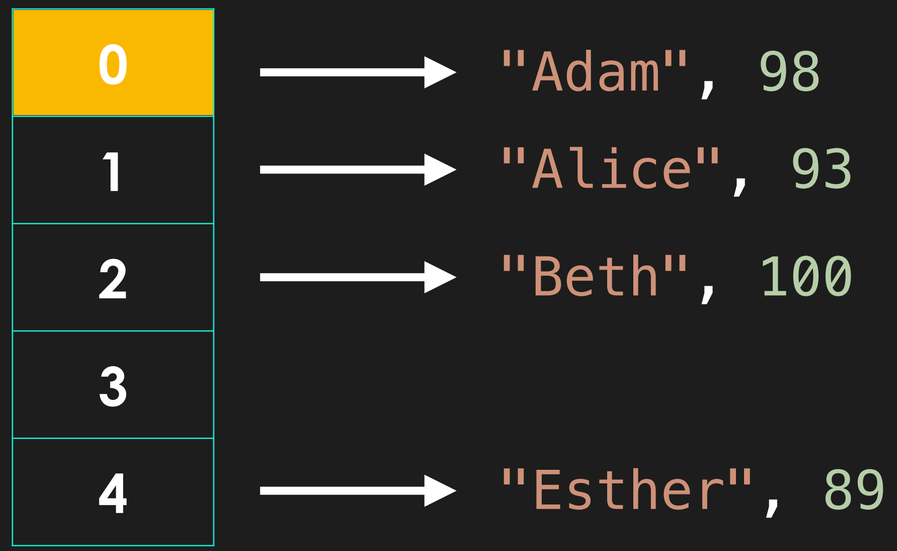
scores



Example of linear probing

```
insert: "Ethan", 72
```

scores



Example of linear probing

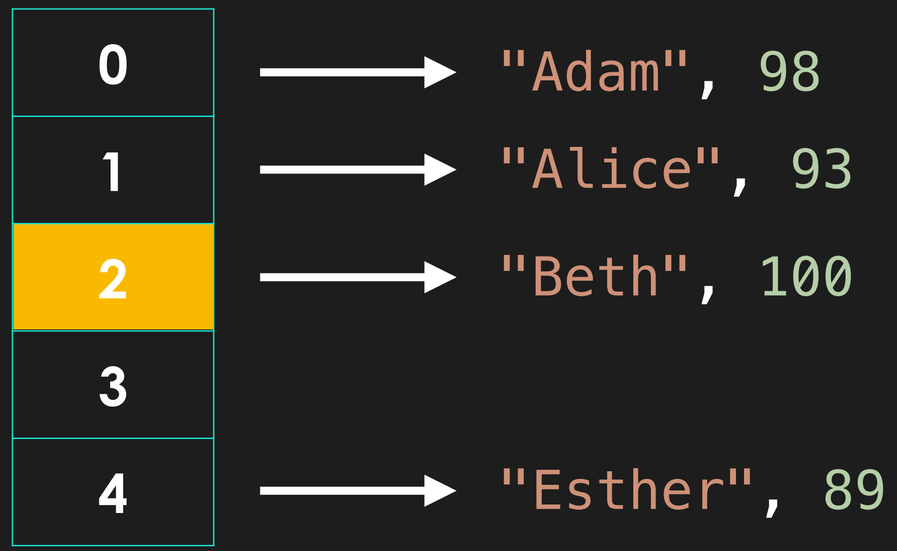
```
insert: "Ethan", 72
```

scores

Example of linear probing

```
insert: "Ethan", 72
```

scores



Example of linear probing

```
insert: "Ethan", 72
```

scores

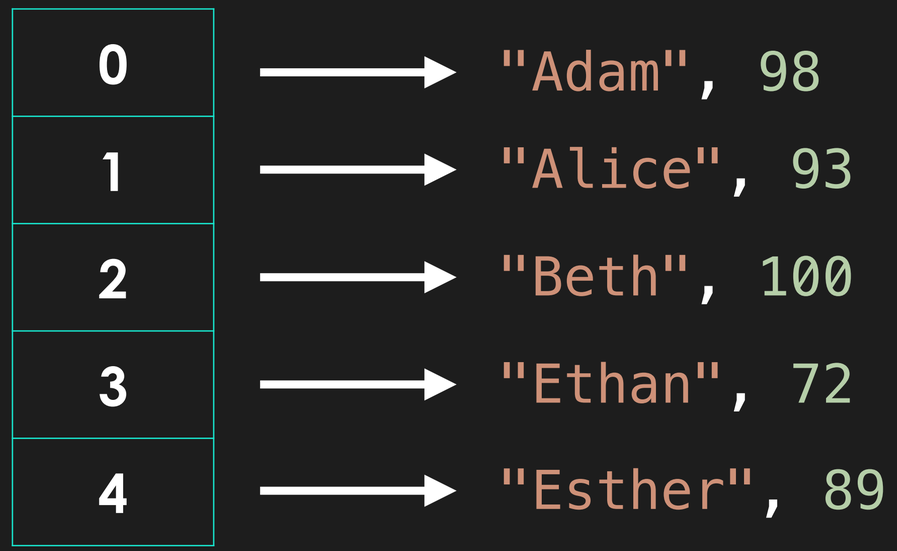
Example of linear probing

```
insert: "Ethan", 72
```

scores

Example of linear probing

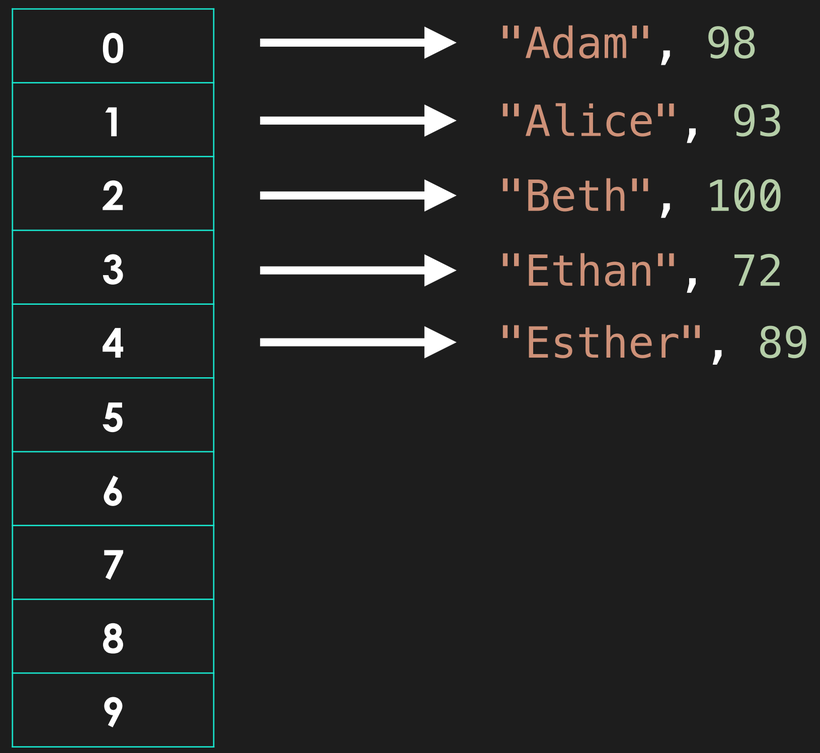
scores



No more space!

Example of linear probing

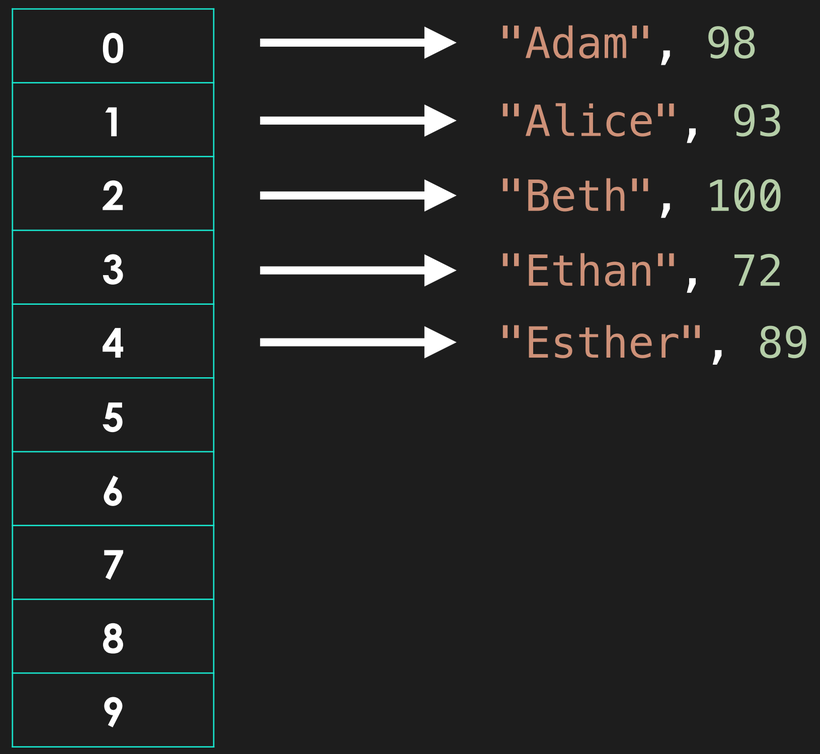
scores



Resize!

Example of linear probing

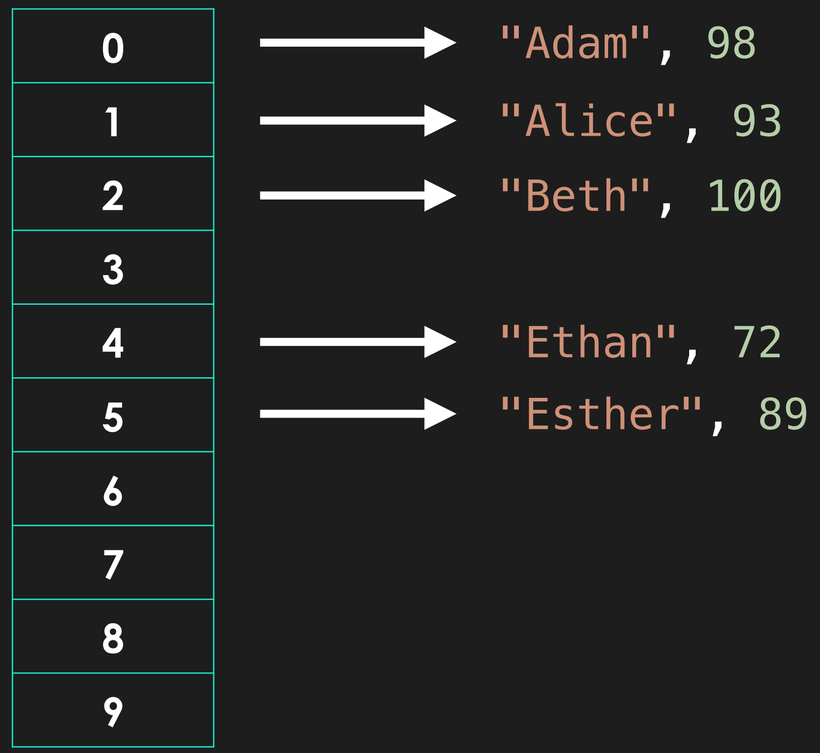
scores



At this point, we also have to **rehash** (reinsert) all our keys because our **table capacity has changed**. Notice how if we search for Ethan, we won't be able to reach him because after Esther there is a blank space!

Example of linear probing

scores

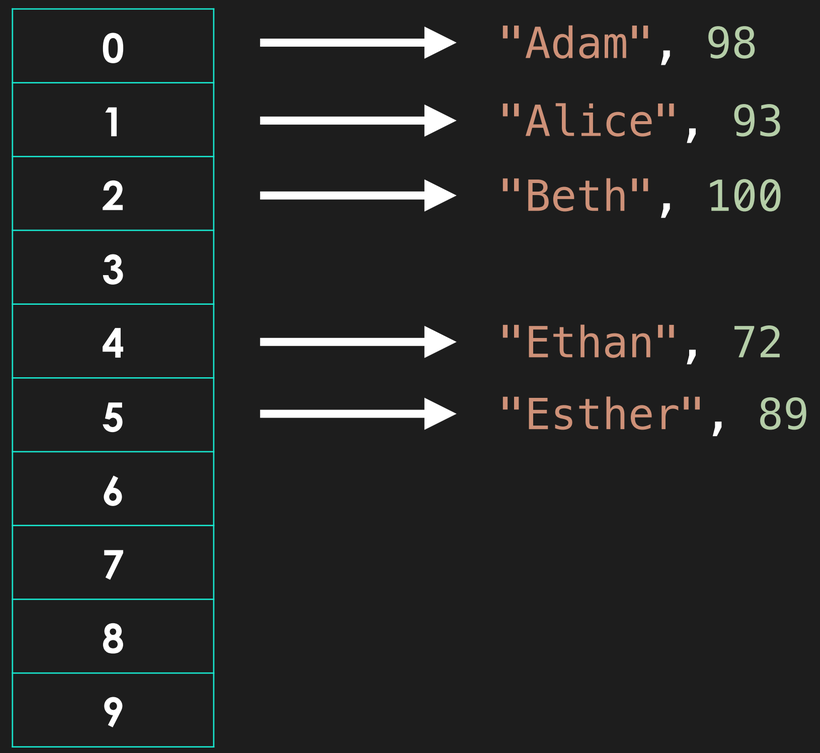


At this point, we also have to **rehash** (reinsert) all our keys because our **table capacity has changed**. Notice how if we search for Ethan, we won't be able to reach him because after Esther there is a blank space!

Example of linear probing

```
insert: "Kevin", 56
```

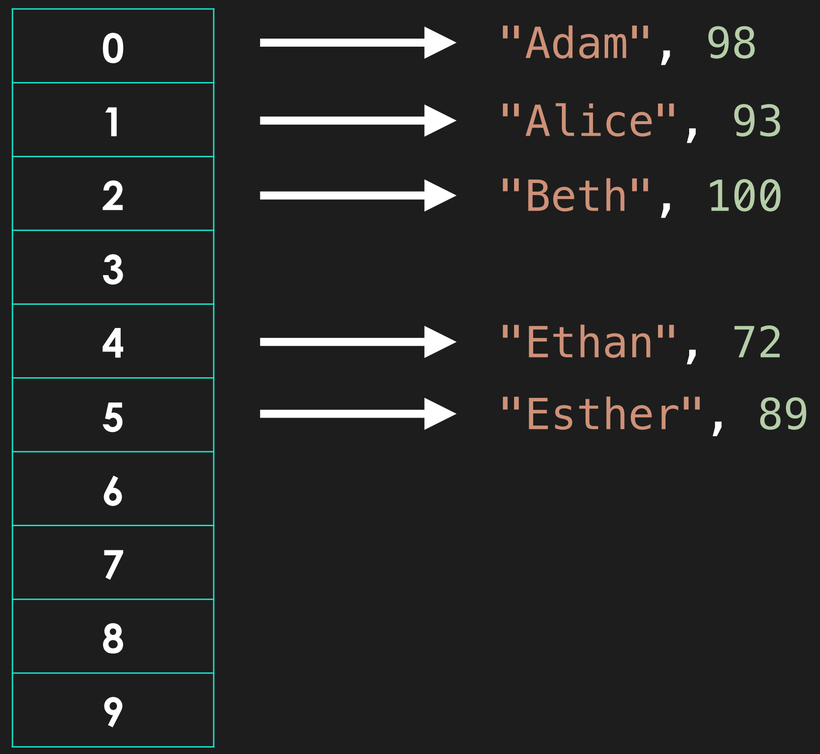
scores



Example of linear probing

```
insert: "Kevin", 56
```

```
scores
```



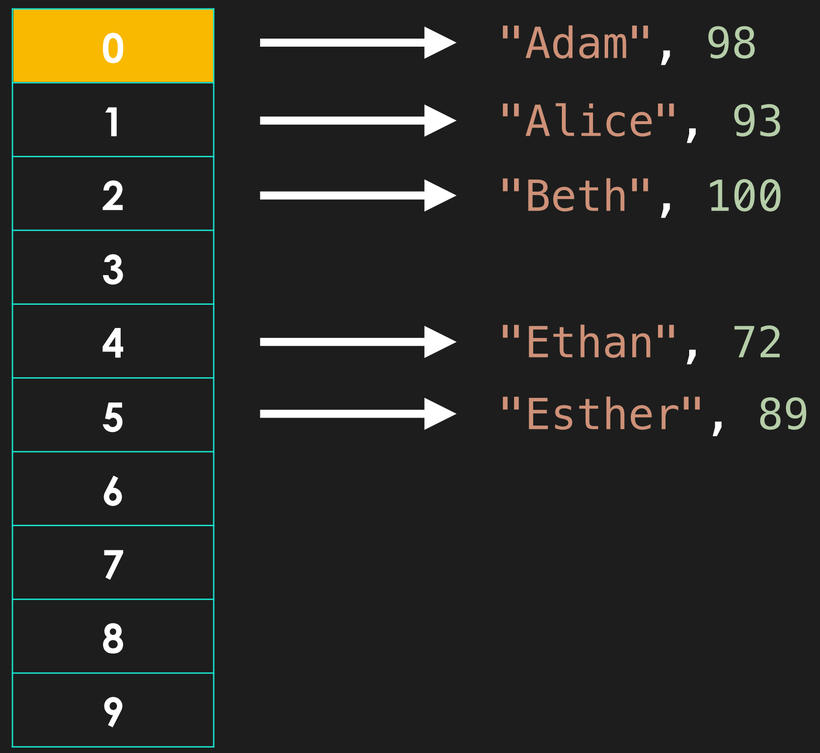
For our tables, our hash function should always use a modulo to ensure hash indices remain within the capacity of the table.

For instance, **"K"** should give **hash index 10.** However, if we mod this by 10, we get **0**.

Example of linear probing

```
insert: "Kevin", 56
```

```
scores
```



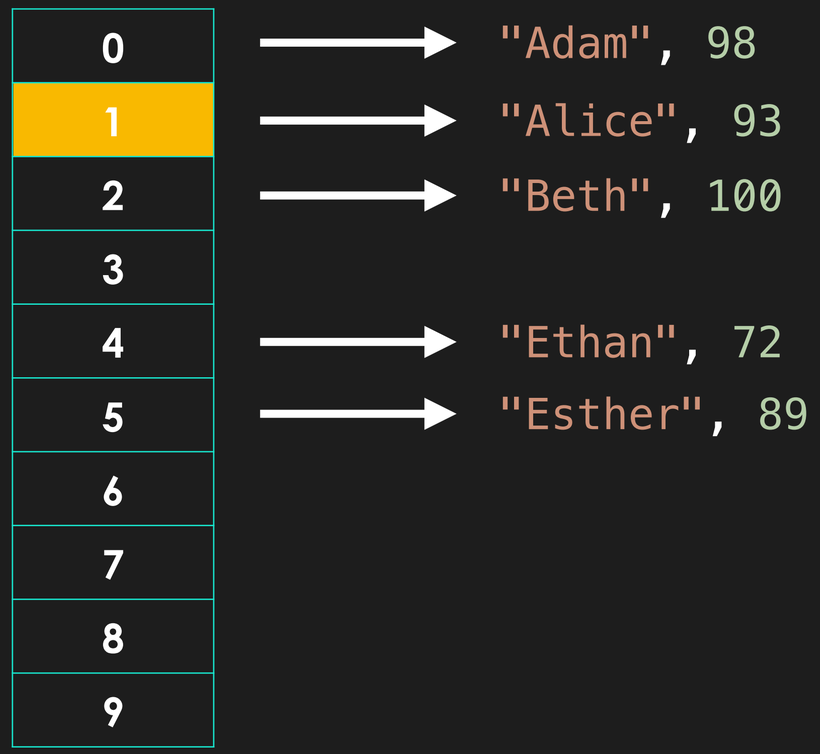
For our tables, our hash function should always use a modulo to ensure hash indices remain within the capacity of the table.

For instance, **"K"** should give **hash index 10.** However, if we mod this by 10, we get **0**.

Example of linear probing

```
insert: "Kevin", 56
```

```
scores
```



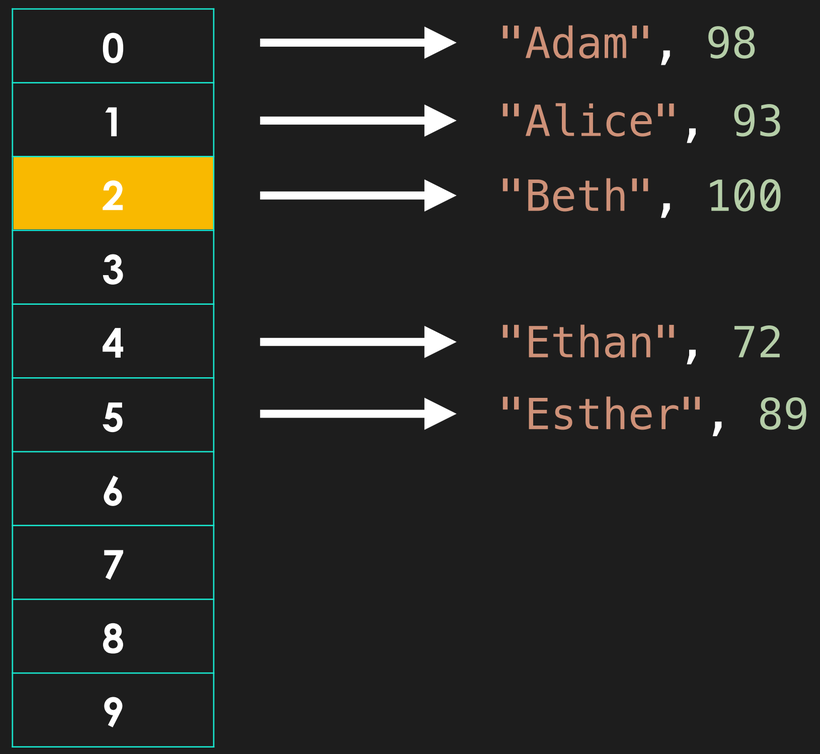
For our tables, our hash function should always use a modulo to ensure hash indices remain within the capacity of the table.

For instance, **"K"** should give **hash index 10.** However, if we mod this by 10, we get **0**.

Example of linear probing

```
insert: "Kevin", 56
```

```
scores
```



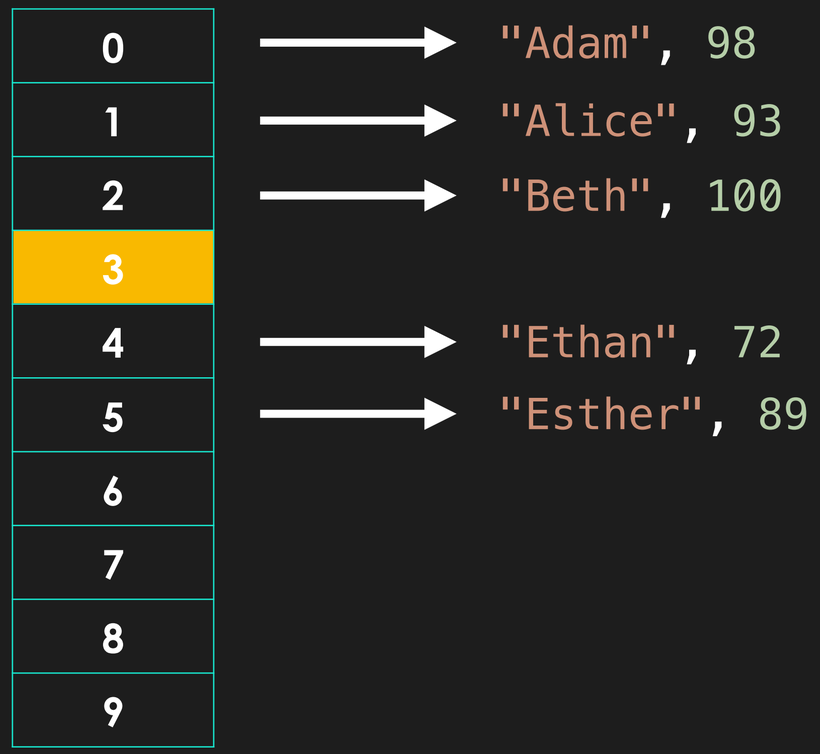
For our tables, our hash function should always use a modulo to ensure hash indices remain within the capacity of the table.

For instance, **"K"** should give **hash index 10.** However, if we mod this by 10, we get **0**.

Example of linear probing

```
insert: "Kevin", 56
```

```
scores
```



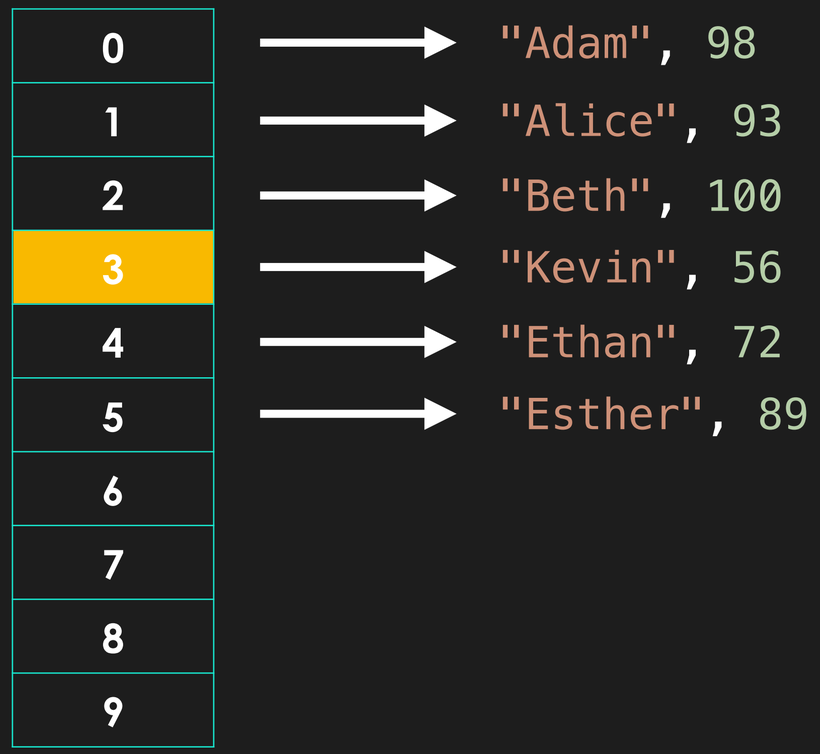
For our tables, our hash function should always use a modulo to ensure hash indices remain within the capacity of the table.

For instance, **"K"** should give **hash index 10.** However, if we mod this by 10, we get **0**.

Example of linear probing

```
insert: "Kevin", 56
```

```
scores
```



For our tables, our hash function should always use a modulo to ensure hash indices remain within the capacity of the table.

For instance, **"K"** should give **hash index 10.** However, if we mod this by 10, we get **0**.

Note: When using the hash tables, we try to avoid resizes as much as possible because rehashing is an expensive operation, and thus it is important to choose a good starting capacity and hash function to avoid resizing and rehashing

LP Hash Table Hash

```
def hashToIndex(self, key, capacity):
    firstLetter = key[0].lower()
    index = ord(firstLetter) - 97
    return index % capacity
```

LP Hash Table Insert

```
def insert(self, key, value):
    if (self.size == len(self.table)):
        self.resizeAndRehash()

    index = self.hashToIndex(key, len(self.table))

    while self.table[index] != None:
        index += 1
        index = index % len(self.table)

    self.table[index] = HashItem(key, value)
    self.size += 1
```

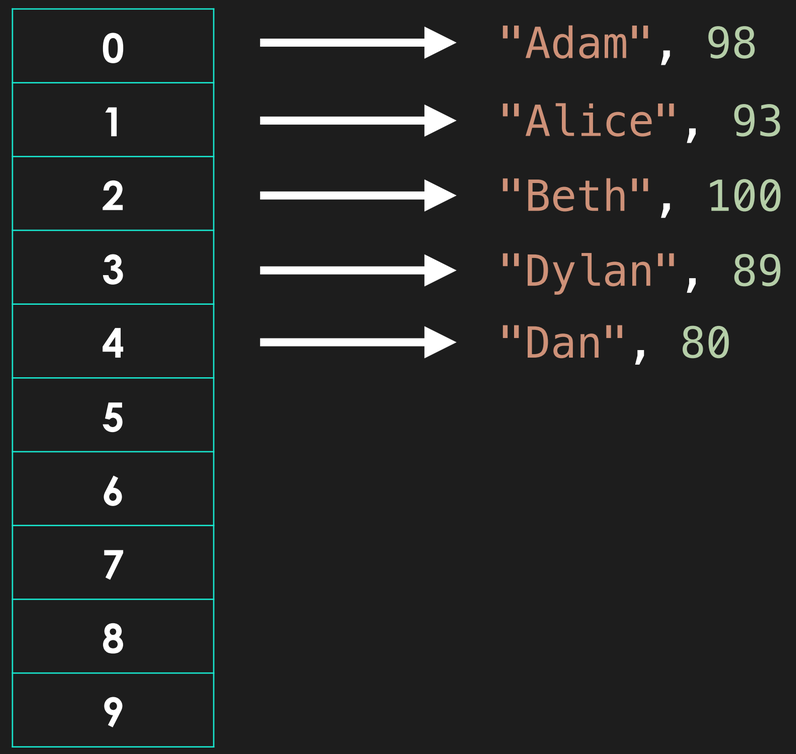
LP Hash Table Resize

```
def resizeAndRehash(self):
    currentCap = len(self.table)
    newTable = LPHashTable(currentCap * 2)

    for i in range(currentCap):
        newTable.insert(self.table[i].key, self.table[i].value)

    self.table = newTable.table
    return
```

Searching in LP Hash Table

Linear Probing Search

scores



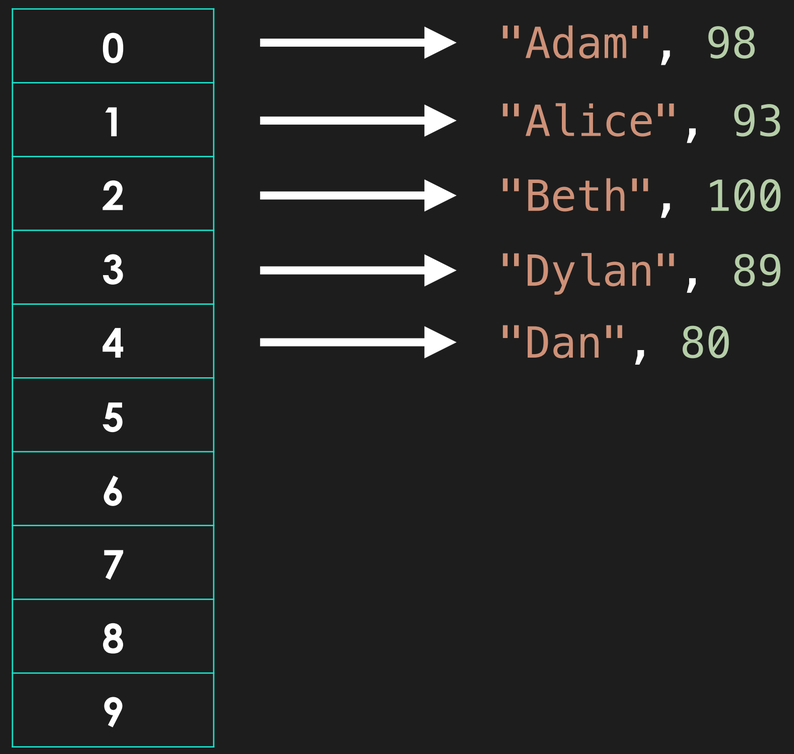
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

```
search: "Dan"
```

scores



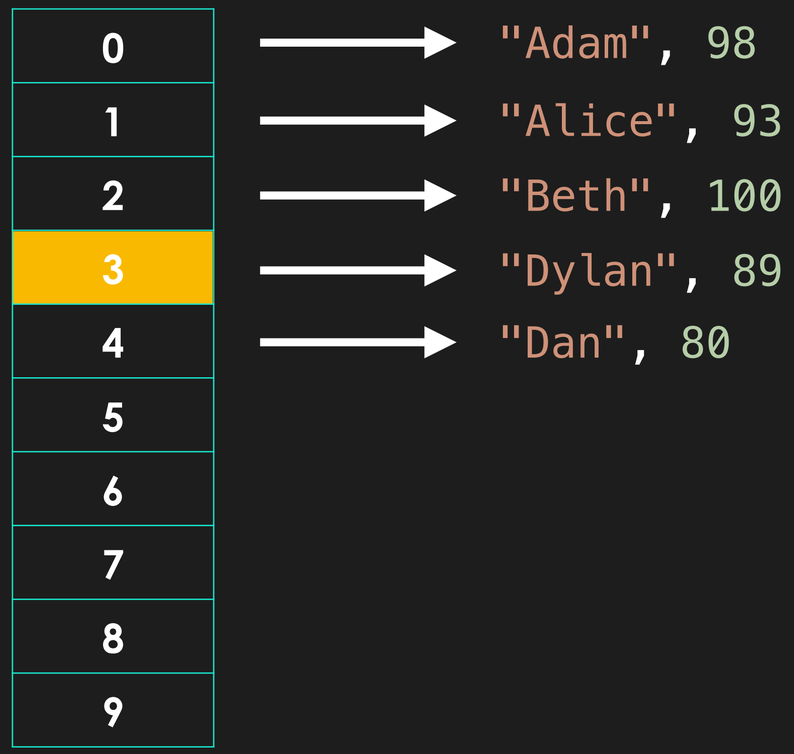
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

```
search: "Dan"
```

```
scores
```



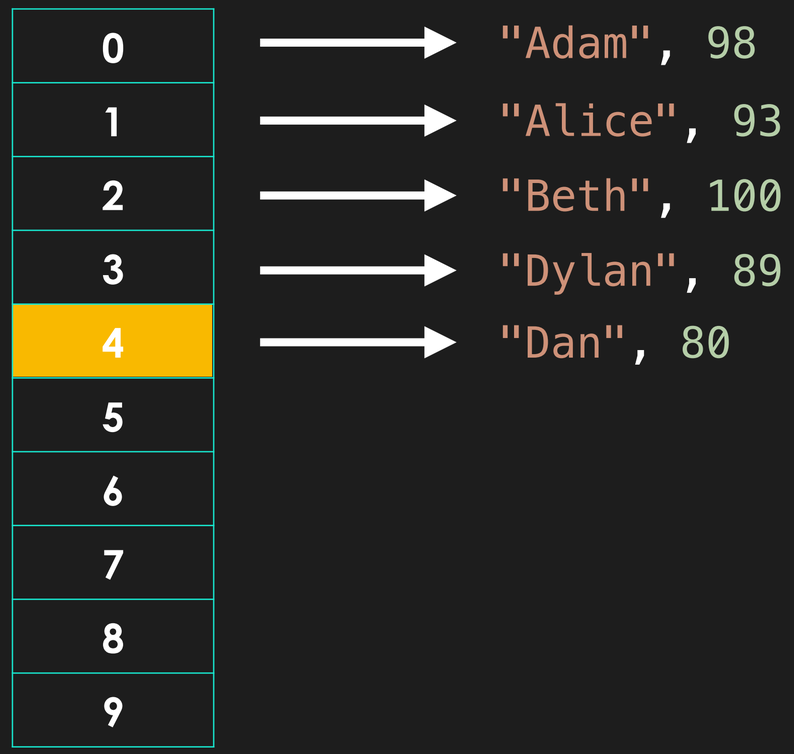
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

```
search: "Dan"
```

```
scores
```



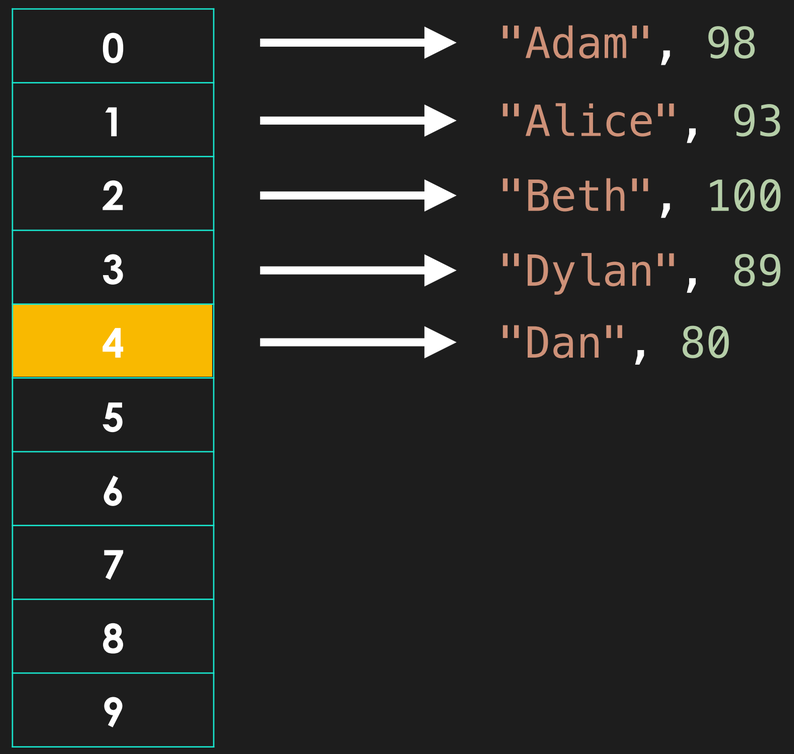
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

`search: "Dan"` **`value: 89!`**

`scores`



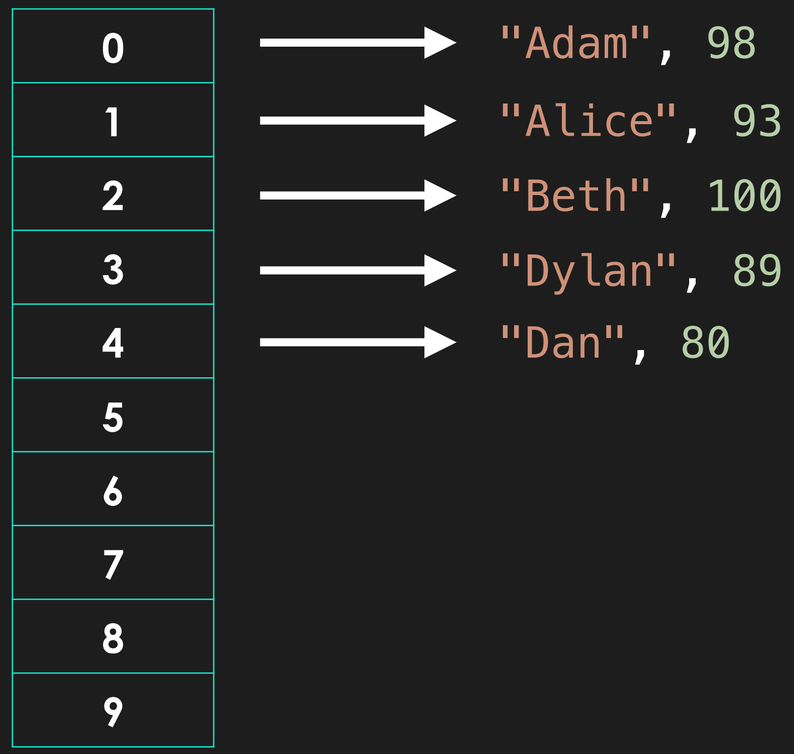
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

```
search: "Darren"
```

```
scores
```



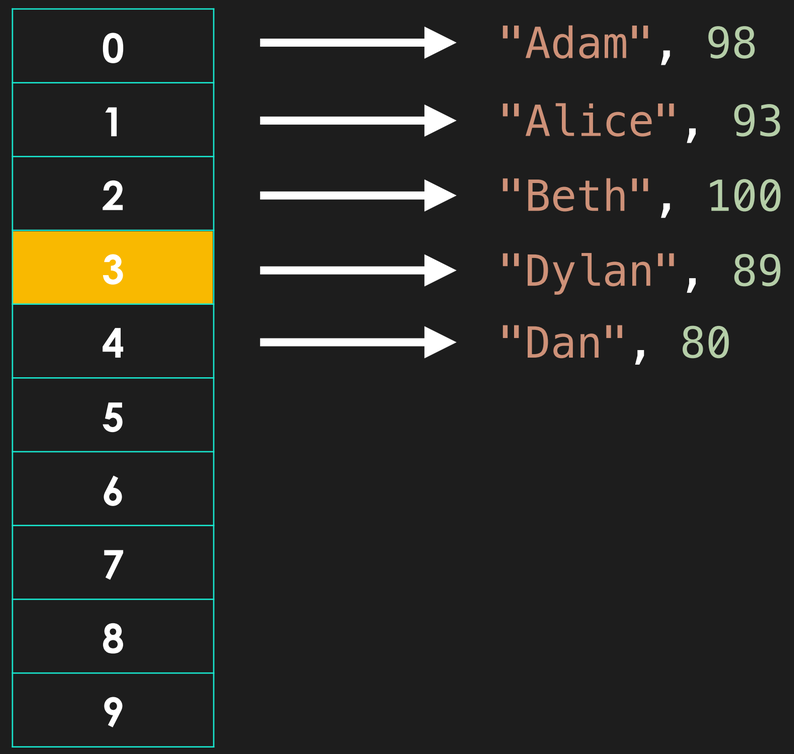
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

```
search: "Darren"
```

```
scores
```



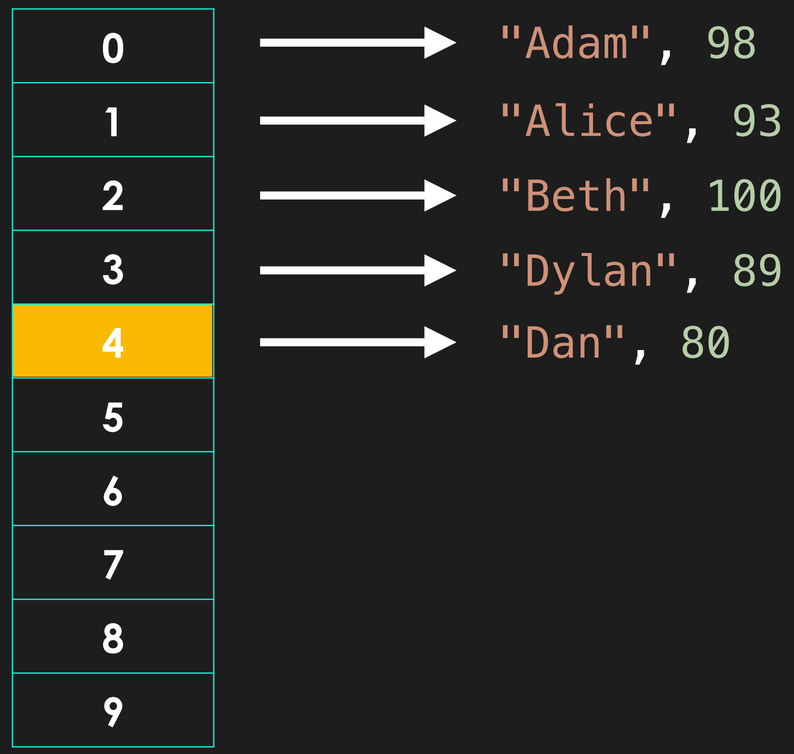
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

```
search: "Darren"
```

```
scores
```



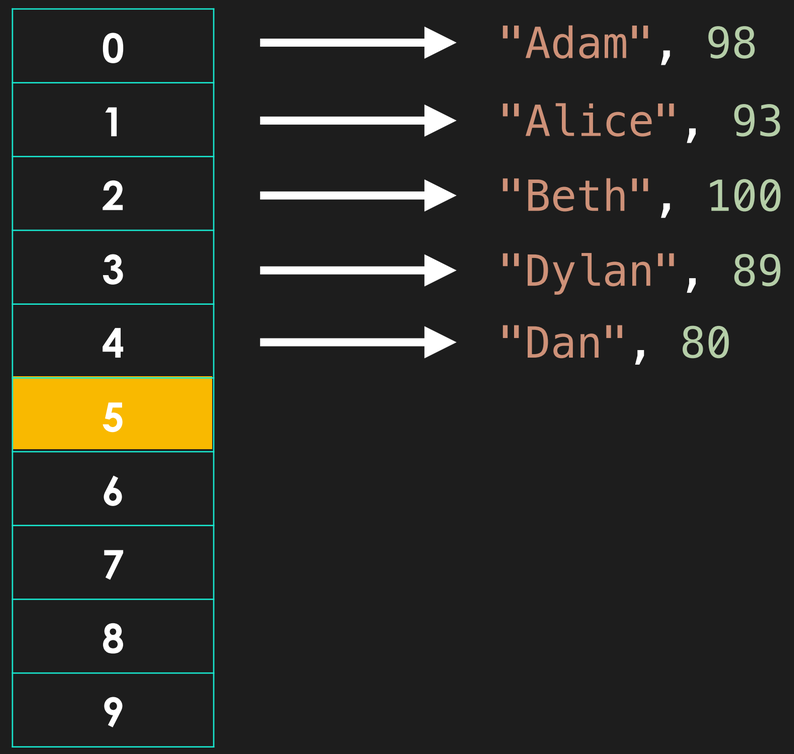
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

```
search: "Darren"
```

```
scores
```



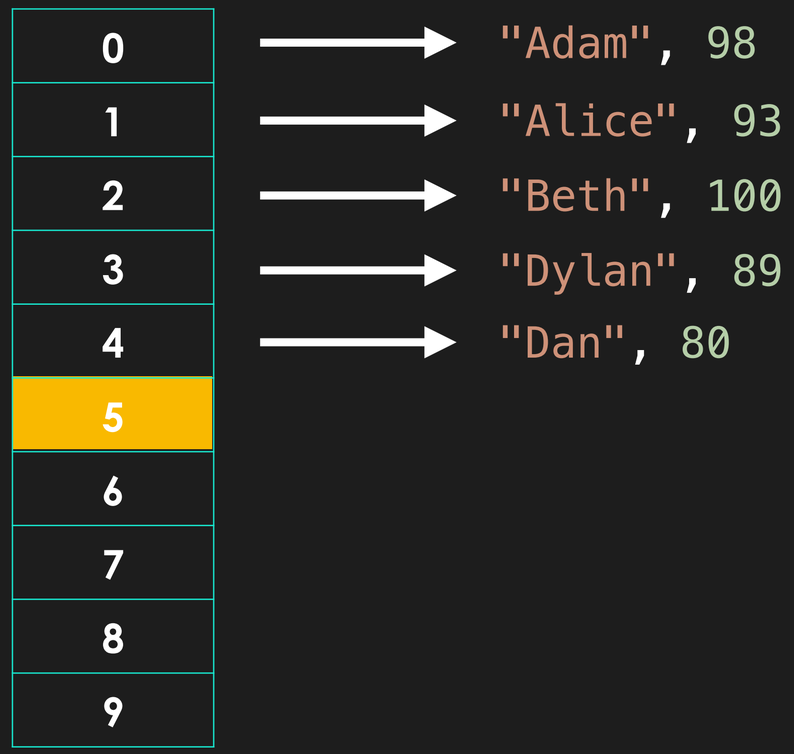
Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

Linear Probing Search

`search: "Darren"` **`value: None`**

`scores`



Searching

1. Get hash index
2. If keys don't match, increment index until either a **null index** is encountered or all **indices have been checked**
3. Return value if key is found

LP Hash Table Search

```
def search(self, key):
        index = self.hashToIndex(key, len(self.table))
        start = index

        if self.table[index] == None:
            return None

        while (self.table[index].key != key):
            index += 1
            index %= len(self.table)

            if index == start or self.table[index] == None:
                return None
            elif index == start:
                return None

        return self.table[index]
```

Deleting from LP Hash Table

Deleting from LP Hash Table

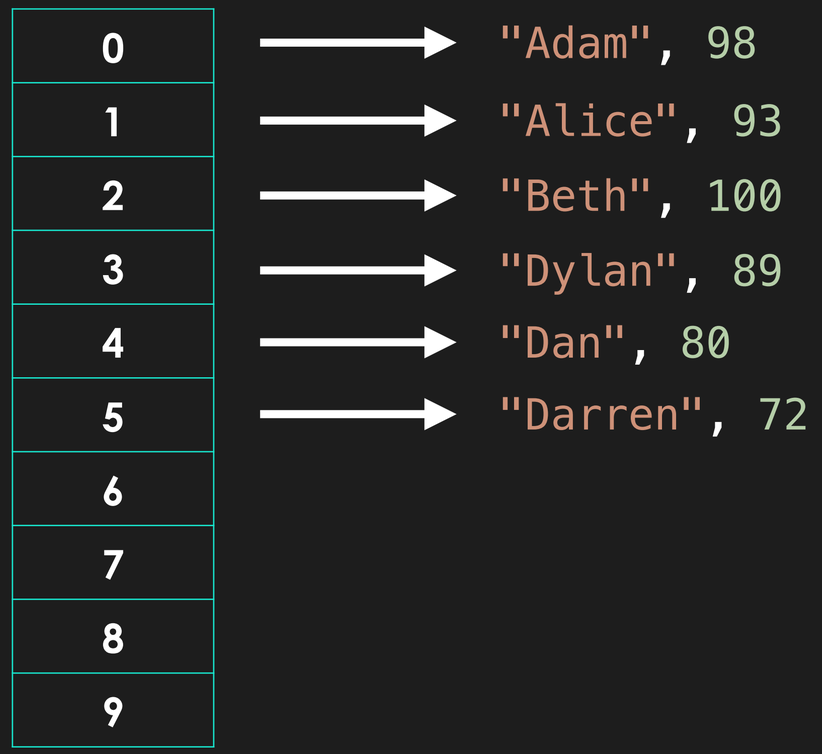
LP Hash Table deletion is a little different from previous variants. For chaining, we can easily remove the key from it's list after searching for it. In Linear Probing, however, we can't just delete a key at a particular index.

Linear Probing Delete

Consider the following situation:

```
delete: "Dan"
```

scores

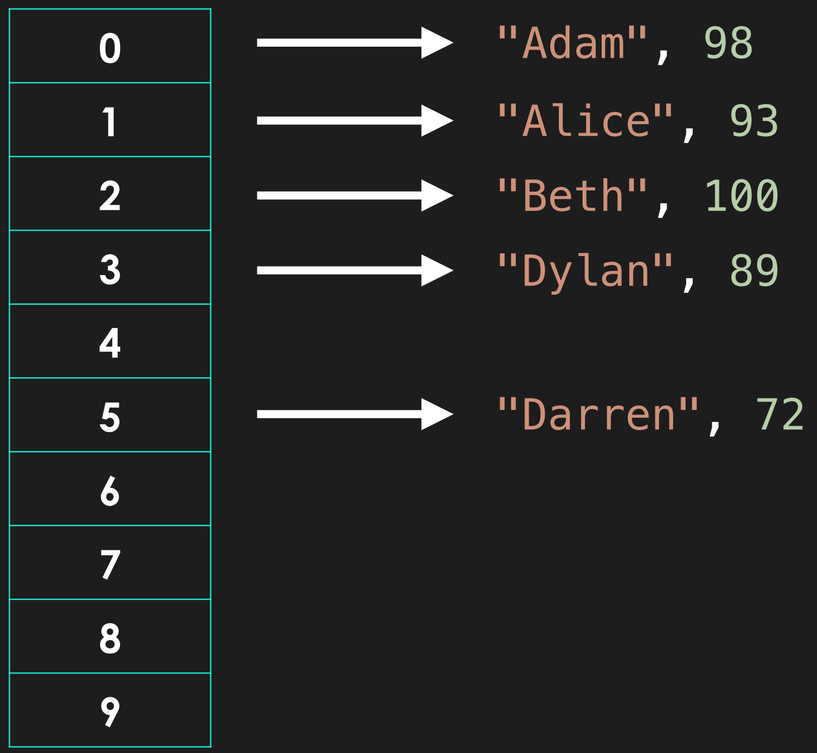


Linear Probing Delete

Consider the following situation:

```
delete: "Dan"
```

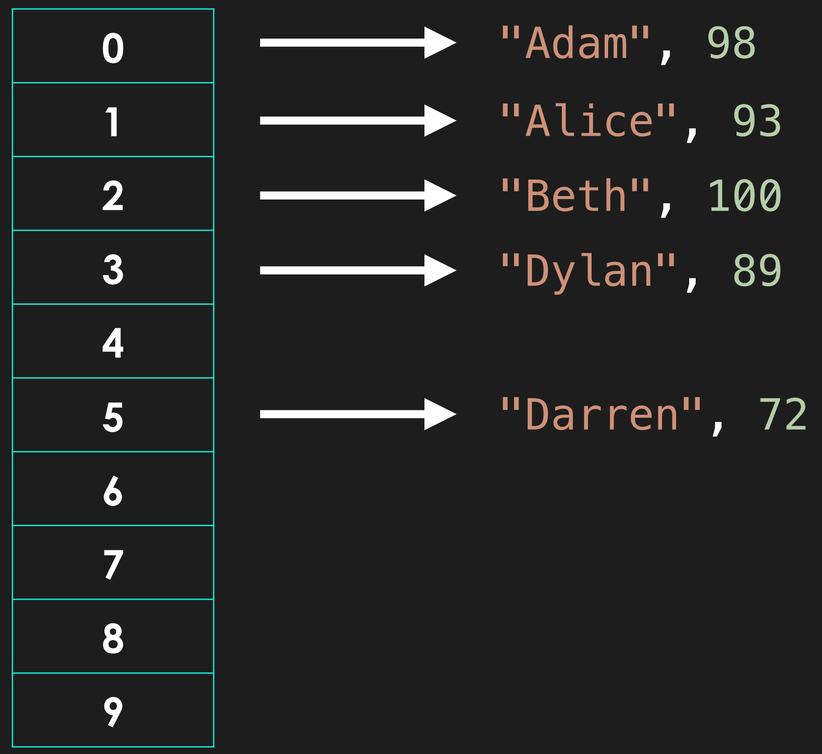
scores



Linear Probing Delete

Consider the following situation:

```
delete: "Dan"

scores
```



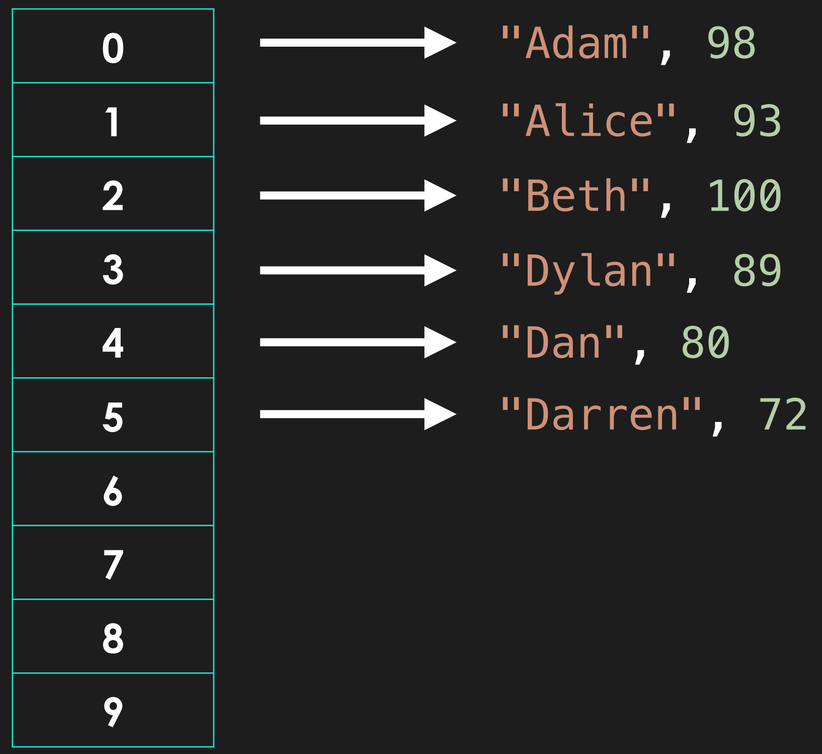
If we just deleted Dan like that, we would have lost access to "Darren"!

Linear Probing Delete

Consider the following situation:

```
delete: "Dan"
```

scores



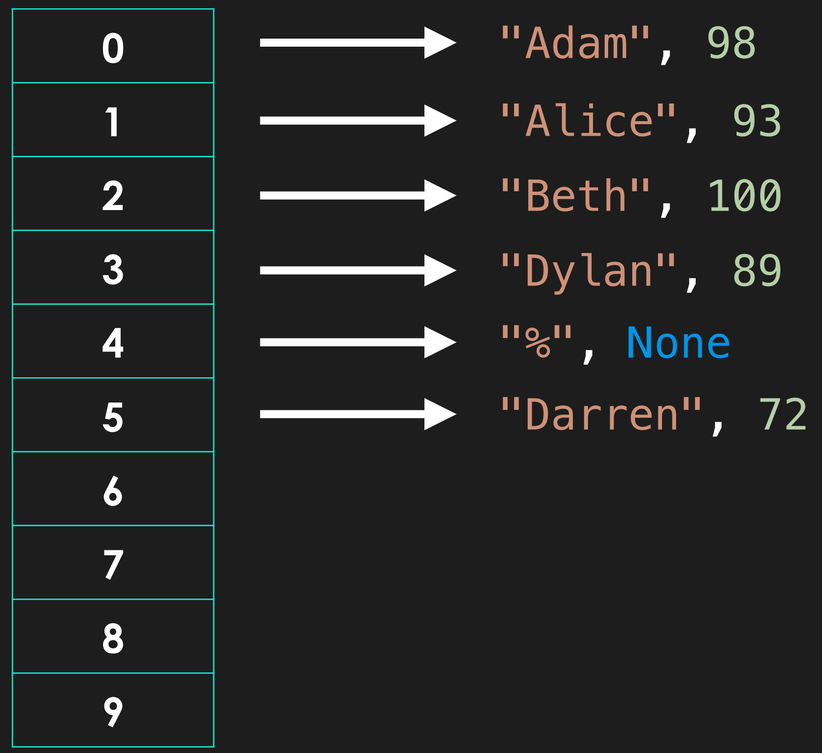
Instead we can simply put what is known as a wild card in deleted spaces. This is a key that can be used to identify deleted spaces!

Linear Probing Delete

Consider the following situation:

```
delete: "Dan"
```

scores



Instead we can simply put what is known as a wild card in deleted spaces. This is a key that can be used to identify deleted spaces!

LP Hash Table Delete

```
def delete(self, key):
    index = self.hashToIndex(key, len(self.table))
    start = index

    if self.table[start] == None:
        return None

    while (self.table[index].key != key):
        index += 1
        index %= len(self.table)

        if index == start:
            print("key doesn't exist, delete failed")
            return

        elif not self.table[index]:
            print("key doesn't exist, delete failed")
            return


    self.table[index] = HashItem("%", None)
    self.size -= 1
```

In fact, now that we have such a delete method, we need to alter our insert method slightly too, such that when a wildcard value is encountered, we can insert there!

Time Complexity of Hash Table with linear probing

Time Complexity of Hash Table with linear probing

Average Case:

- **Insertion: O(1)**
- **Search: O(1)**

Time Complexity of Hash Table with linear probing

Average Case:

- **Insertion: O(1)**
- **Search: O(1)**

Worst Case:

- **Insertion: O(N)**
- **Search: O(N)**

Chaining vs Linear Probing

CHAINING	LINEAR PROBING
Slower as accessing linked list means accessing random areas of memory	Faster due to locality of reference. Arrays are contiguous blocks of memory
Doesn't experience clustering	If hash function is weak, there will be clustering (i.e. many keys have the same hash result and form a run in some portion of the array)
Deletion is easy	Deletion is troublesome and after many deletions rehash is required
Memory consumption is high	Memory consumption is not as high

Applications of Hash Tables

Applications of Hash Tables

A. **Database Indexing:** Hash Tables is one method to implement database indexing, allowing users fast search for a key

B. **Maps / Dictionaries:** The python dictionary makes use of the hash table data structure. In fact, in other languages, you might hear of the term map, which is often implemented through hash tables

C. **Sets:** Sets only allow for unique keys. Using hash tables, an efficient implementation of sets is possible

Hash Tables vs Binary Seach Trees

Hash Tables and Search Trees are often the go to data structures when trying to achieve efficient search operations. However, it is important when to know which to use:

Hash Tables vs Binary Seach Trees

A. **Hash Tables:** Hash Tables are good when you know the size of the data because then you can adjust your hash function to minimise collisions. Effectively, this makes search **O(1)**, faster than the logN search trees!

B. **Binary Search Trees:** Search Trees are useful when the data is constantly changing and therefore your structure might need to be dynamic. This is where BSTs really shine because with hash tables, you would have to resize a lot which is not only time, but memory inefficient. **Additionally**, in BSTs, **ordering is maintained**, which means operations such as rank can be easy to compute

Lab Session 2

Implement **linear_probing.py**

Use **lp_test.py** to test