Collisions: Retrieving

6 min

When we retrieve hash map values, we also need to be aware that different keys could point to the same array index leading us to retrieve the wrong value.

To avoid this, we'll search through the linked list at an index until we find a node with a matching key. If we find the node with the correct key, we'll return the value; otherwise, we'll return null.

The .retrieve() method will follow this logic:

store the hashed key in the constant arrayIndex

store the head node of a list in the variable current

while there is a valid node

if the current node's key matches the key

return the current node's value

set current to the next node in the list

return null

Instructions

1. Checkpoint 1 Passed

1.

In .retrieve(), declare a variable whose value can be changed, current. Assign it the head node of the linked list at arrayIndex to current.

We will search for the value we want to retrieve by checking each node in the linked list, starting at the head node.

Hint

Use the LinkedList property that stores the head node.

2. Checkpoint 2 Passed

2.

Create a while loop that we'll use to iterate over each node in the linked list until we either find the value we're looking for or reach the end of the list.

Set the while condition to be the current node of the linked list.

Hint

We stored the current node in the current variable.

3. Checkpoint 3 Passed

3.

Inside of the while loop, add an if statement that checks if the key we received as an argument and the key of the current node are the same.

If both keys are the same, this means we've found the node with the correct value. Return the value stored in the current node.

Hint

Check how nodes are implemented in **Node.js**. Remember that the .key and .value are stored in the .data property of the node.

4. Checkpoint 4 Passed

4.

If the keys don't match, we need to check the next node in the linked list. Outside of the if statement, set current to the next node in the linked list.

Hint

The .next property of Node stores a reference to the next node in the linked list.

5. Checkpoint 5 Passed

5.

If we've looped through the entire linked list without finding the value we want to retrieve, it means the value is not stored in the hash map. After the while loop, return null.

Hint

Each node knows the node following it. Check **Node.js** for the Node property that allows you to access this next node.

HashMap.js

```
const LinkedList = require('./LinkedList');

const Node = require('./Node');

class HashMap {
 constructor(size = 0) {
  this.hashmap = new Array(size)
   .fill(null)
   .map(() => new LinkedList());
 }


 hash(key) {
  let hashCode = 0;
  for (let i = 0; i < key.length; i++) {
   hashCode += hashCode + key.charCodeAt(i);
```

```
  }
  return hashCode % this.hashmap.length;
}

assign(key, value) {
  const arrayIndex = this.hash(key);
  const linkedList = this.hashmap[arrayIndex];
  if (linkedList.head === null) {
    linkedList.addToHead({ key, value });
    return;
  }
  let current = linkedList.head;
  while (current) {
    if (current.data.key === key) {
      current.data = { key, value };
    }
    if (!current.next) {
      current.next = new Node({ key, value });
      break;
    }
    current = current.next;
  }
}

retrieve(key) {
  const arrayIndex = this.hash(key);
  const linkedList = this.hashmap[arrayIndex];
  let current = linkedList.head;

  while (current) {
   if (current.data.key === key) {
    return current.data.value;
```

```
    }
    current = current.next;
   }
   return null;
  }
}

module.exports = HashMap;
```

LinkedList.js

```
const Node = require('./Node');

class LinkedList {
  constructor() {
    this.head = null;
  }

  addToHead(data) {
    const newHead = new Node(data);
    const currentHead = this.head;
    this.head = newHead;
    if (currentHead) {
      this.head.setNextNode(currentHead);
    }
  }

  addToTail(data) {
    let tail = this.head;
    if (!tail) {
      this.head = new Node(data);
    } else {
```

```
    while (tail.getNextNode() !== null) {
      tail = tail.getNextNode();
    }
    tail.setNextNode(new Node(data));
  }
}

removeHead() {
  const removedHead = this.head;
  if (!removedHead) {
    return;
  }
  if (removedHead.next) {
    this.head = removedHead.next;
  }
  return removedHead.data;
}

printList() {
  let currentNode = this.head;
  let output = '<head> ';
  while (currentNode !== null) {
    output += currentNode.data + ' ';
    currentNode = currentNode.next;
  }
  output += `<tail>`;
  console.log(output);
}

findNodeIteratively(data) {
  let currentNode = this.head;
  while (currentNode !== null) {
```

```
    if (currentNode.data === data) {
      return currentNode;
    }
    currentNode = currentNode.next;
  }
  return null;
 }

 findNodeRecursively(data, currentNode = this.head) {
  if (currentNode === null) {
    return null;
  } else if (currentNode.data === data) {
    return currentNode;
  } else {
    return this.findNodeRecursively(data, currentNode.next);
  }
 }

}

module.exports = LinkedList;
```

Node.js

```
class Node {
 constructor(data) {
  this.data = data;
  this.next = null;
 }

 setNextNode(node) {
  if (!(node instanceof Node)) {
```

```
      throw new Error('Next node must be a member of the Node class');
    }
    this.next = node;
  }

  setNext(data) {
    this.next = data;
  }

  getNextNode() {
    return this.next;
  }
}

module.exports = Node;
```

test.js

```
console.log = function() {};
const fs = require('fs');
const { assert, expect } = require('chai');

describe('', function() {
  it('', function() {
    const HashMap = require('../HashMap');
    const hm = new HashMap(3);
    const testValue = 'test';
    expect(hm.retrieve(testValue), `Does \`.retrieve()\` return \`null\` if the value wasn't found?`).to.equal(null);
  });
});
```