Hash Tables

Professor Frank Kreimendahl

School of Computing and Data Science
Wentworth Institute of Technology

November 9, 2022

Hash Table
Hash Table Description
Definition
Example
Hash Codes
HashCode
Collisions
Open Addressing
Chaining
KWHashMap

Hash Table

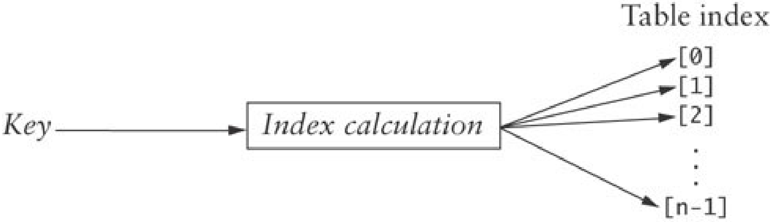
Hash Table Description

- **Hash tables** are implementations of data storage with useful properties:
  - Used to implement sets
  - Used to implement maps
  - Hash tables store keys (and maybe values)
  - These keys (and associated values) are directly accessible
  - Similar to an index in an array, there's only one location an entry might be in a hash table

Hash Table
Hash Table Description
Definition
Example
Hash Codes
HashCode
Collisions
Open Addressing
Chaining
KWHashMap

Definition

- A **Hash table** consists of two necessary parts:
  - An array to hold values – the table
  - A **hash function** which translates a key to an integer value called a **hash code**
  - The integer value is used as an array index – Java arrays only ever use `int`s as an index



Example Hash Function

- Consider an array of size 20, and characters for keys
- An example hash function could be: convert the character to ASCII, and then `mod` the result by 20
- With this function, all the resulting indices are between $0-19$, and each character shows up at a predictable location
- For example, $A = 65$, so its index is $65\%20 = 5$. $a = 97$, so its index is $97\%20 = 17$
- We can store/retrieve these characters by looking directly at their associated index, no searching needed
- Is there an issue with this?

Hash Table
- Hash Table Description
- Definition
- Example
- Hash Codes
- HashCode
- Collisions

Open Addressing

Chaining

KWHashMap

Hash Codes

- Usually keys are strings of letters/numbers
- The number of possible keys is much larger than the table
- Different keys can generate the same hash code, causing a *collision*
- A good hash function distributes all of the keys evenly across possible indices
- Researchers have written better hash functions already – we typically use those rather than create our own

Hash Table
Hash Table Description
Definition
Example
Hash Codes
HashCode
Collisions
Open Addressing
Chaining
KWHashMap

Java `HashCode` Method

- Java's string hash function is called with the `.HashCode()` method
- Both the individual characters and their position in the string have an effect on the hash code
- string s has the hash code $s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + \cdots + s_{n-1}$
- Example: `"Cat".HashCode() =` $\text{‘C’} \times 31^2 + \text{‘a’} \times 31 + \text{‘t’} = 67510$
- 31 is chosen as a multiplier because it is prime, which gives good distribution properties usually

Hash Table
Hash Table Description
Definition
Example
Hash Codes
HashCode
Collisions
Open Addressing
Chaining
KWHashMap

Collisions

- `.HashCode()` distributes hash codes evenly, so one index isn't more likely, given a range of keys
- The probability of a collision is based on how full the table is
- There is **always** a non-zero chance of a collision
- We will look at two ways to handle collisions without losing information

Hash Table
Open Addressing
Open Addressing
Issues
Example
Deletion
Resizing
Probing
Probing
Chaining
KWHashMap

Open Addressing

Hash Table
Open Addressing
Open Addressing
Issues
Example
Deletion
Resizing
Probing
Probing
Chaining
KWHashMap

Open Addressing

- Open addressing can be used to find/add items to a hash table without collision issues
- If there is a collision inserting a key, use *linear probing* to find other possible spots for the key:
  - Increment the index by 1 until there is a null element
  - Store the key there
- If there is a collision searching for a key, follow the same steps:
  - Increment the index by 1 until the key is found or there is a null element

Issues

- What happens if you reach the end of the array?
  - Treat the array like a circular array
  - Set the index to 0 and then start incrementing again
- What happens if the array is full?
  - We will search for a null spot forever
  - Instead, detect an end condition: when we get back to our starting spot
- Avoid a full table by resizing after a certain *load factor*

Example

Key	"Tom"	"Harry"	"Sam"	"Pete"
`hashCode()`	84274	69496448	82879	2484038
`hashCode() % 5`	4	3	4	3

Inserting Tom:

				Tom

Example

Key	"Tom"	"Harry"	"Sam"	"Pete"
`hashCode()`	84274	69496448	82879	2484038
`hashCode() % 5`	4	3	4	3

Inserting Harry:

			Harry	Tom

Example

Key	"Tom"	"Harry"	"Sam"	"Pete"
`hashCode()`	84274	69496448	82879	2484038
`hashCode() % 5`	4	3	4	3

Inserting Sam:

Sam			Harry	Tom

Example

Key	"Tom"	"Harry"	"Sam"	"Pete"
`hashCode()`	84274	69496448	82879	2484038
`hashCode() % 5`	4	3	4	3

Inserting Pete:

Sam	Pete		Harry	Tom

Hash Table
Open Addressing
Open Addressing
Issues
Example
Deletion
Resizing
Probing
Probing
Chaining
KWHashMap

Deletion

- We can't just set an index to null to delete that item with open addressing
- What if there had been a collision?
  - Set index to a dummy node – space is available for insertion but you should continue searching for find operations
- The dummy node can be replaced with a new key if that key is not in the table

Hash Table
Open Addressing
Open Addressing
Issues
Example
Deletion
Resizing
Probing
Probing
Chaining
KWHashMap

Resizing

- More collisions means more steps for each insert/find/delete operation
- Move all elements to a larger table so there are fewer collisions:
  - Create a larger array (ideally with a prime number of elements)
  - Insert all of the elements in the current array into the new one
  - Note that this requires *rehashing* – the indices might change with a different table size
  - Do not copy dummy values

Resizing Example

Key	"Tom"	"Harry"	"Sam"	"Pete"
`hashCode()`	84274	69496448	82879	2484038
`hashCode() % 5`	4	3	4	3
`hashCode() % 11`	3	10	5	7

Sam	Pete		Harry	Tom

Reinsert keys at new indices:

			Tom		Sam		Pete		Harry	

Hash Table
Open Addressing
Open Addressing
Issues
Example
Deletion
Resizing
Probing
Probing
Chaining
KWHashMap

Probing

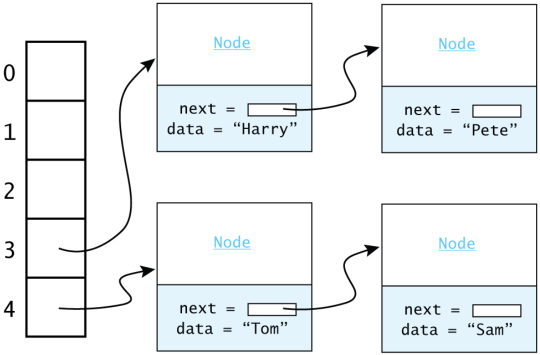
- Linear probing leads to clusters of values in adjacent indices, which is inefficient
- *Quadratic* probing changes the increments when there is a collision
- Use square increments: $+1^2, +2^2, +3^2 \ldots$, using a circular array
- This spreads out colliding keys
- Issue: This sequence doesn't reach every index
- Solution: If the array has a prime size and enough free space, it will succeed

Hash Table
Open Addressing
Chaining
Chaining
Advantages
Performance
KWHashMap

Chaining

Chaining Description

- *Chaining* is an alternative solution to hash code collisions
- Instead of each element in the table holding a value, each element holds a linked list
- These linked lists are called *buckets*



Advantages

- You only need to examine keys in a single bucket
- You can store more unique keys than the hash table size
- Insertion is simple – if the key is not in its bucket, insert it at the beginning of the list
- Deletion is simple – remove the key from the linked list

Performance

- For both open addressing and chaining hash tables, the load factor measures the number of non-null elements divided by table size
- The load factor determines how quickly we can insert/find/delete because it determines the chance of a collision
- Open addressing works more slowly than chaining when load factors are high
- Open addressing doesn't require linked lists so it is more memory-efficient
- When load factor is low, a hash table is as efficient as accessing values in an array, which is as efficient as possible

Hash Table
Open Addressing
Chaining
KWHashMap
Interface
Entry Class
get operation
put operation
remove operation
Data Fields

KWHashMap

KWHashMap Interface

Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this table contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key`. Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this table if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping.
`int size()`	Returns the size of the table.

`Entry` Class

Data Field	Attribute
`private K key`	The key.
`private V value`	The value.
Constructor	**Behavior**
`public Entry(K key, V value)`	Constructs an `Entry` with the given values.
Method	**Behavior**
`public K getKey()`	Retrieves the key.
`public V getValue()`	Retrieves the value.
`public V setValue(V val)`	Sets the value.

Class to hold key-value pairs for entries in a hashtable

get operation

GET(key)

```
index ← key.hashCode() % table.length
if index is negative then
    index += table.length
if table[index] is null then
    return null
for all e in list at table[index] do
    if e.key matches key then
        return e.value
return null
```

put operation

```
PUT(key, value)
index ← key.hashCode() % table.length
if index is negative then
    index += table.length
if table[index] is null then
    table[index] ← new linked list
Search list for key
if key in table then
    set new value of entry
    return old value of entry
else
    Insert new key/value pair into list
    Increment numKeys
    return null
```

remove operation

REMOVE(key)

```
index ← key.hashCode() % table.length
if index is negative then
    index += table.length
if table[index] is null then
    return null
Search list for key
if key in table then
    Remove entry from list
    Decrement numKeys
    return value
return null
```

Data fields

```
import java.util.*;
public class HashtableChain<K, V>
             implements KWHashMap<K, V> {
  // Insert inner class Entry<K, V> here.
  /** The table */
  private LinkedList<Entry<K, V>>[] table;
  /** The number of keys */
  private int numKeys;
  /** The capacity */
  private static final int CAPACITY = 101;
  /** The maximum load factor */
  private static final double LOAD_THRESHOLD = 3.0;
  public HashtableChain() {
    table = new LinkedList[CAPACITY];
  }
  ...
}
```