Lecture Notes
Data Structures and Algorithms

Dictionaries and Hash Tables

The Dictionary ADT - Introduction

The Dictionary abstract data type (also known as a map or associative array) is a collection of keys and values that allows you to store and retrieve values based on their keys. Some common operations that you can perform with a dictionary include:

- Inserting a key-value pair
- Updating the value for a given key
- Removing a key-value pair
- Looking up the value for a given key

Hash tables are a common way to implement dictionaries because they allow for efficient insertion, deletion, and lookup of values based on keys. In a hash table, the keys are hashed to indices in an array, and the values are stored in the corresponding array elements. The hash function maps keys to indices in a way that tries to evenly distribute the keys across the array, which helps to reduce the number of collisions (when two or more keys hash to the same index).

Here are two examples of how you might use dictionaries implemented as hash tables in Java:

Suppose we are given a list of students and their grades on a test. We want to write a program that keeps track of the grades and can answer the following queries:

- What is the average grade for the class?
- What is the highest grade?
- What is the lowest grade?

To solve this problem, we could use a dictionary ADT to store the grades as key-value pairs, where the key is the student's name and the value is the student's grade. Here is an example of how we could use a dictionary ADT in Java to solve this problem:

```java
// Declare the dictionary ADT to store the student grades.
Dictionary<String, Integer> grades = new Hashtable<>();

// Add the student grades to the dictionary.
grades.put("Alice", 87);
grades.put("Bob", 72);
grades.put("Charlie", 95);
grades.put("Dave", 65);
grades.put("Eve", 100);

// Calculate the average grade for the class.
int sum = 0;
for (String name : grades.keys()) {
  sum += grades.get(name);
}
double average = (double) sum / grades.size();

// Print the average grade.
System.out.println("The average grade is " + average);

// Find the highest and lowest grades.
int highest = Integer.MIN_VALUE;
int lowest = Integer.MAX_VALUE;
for (String name : grades.keys()) {
  int grade = grades.get(name);
  if (grade > highest) {
    highest = grade;
  }
  if (grade < lowest) {
    lowest = grade;
  }
}

// Print the highest and lowest grades.
System.out.println("The highest grade is " + highest);
System.out.println("The lowest grade is " + lowest);
```

In this example, we use a hash table (implemented as a **HashMap**) to count the number of times each word appears in a file. The keys are the words and the values are the counts. We use the **put** method to insert or update the count for each word, and the **get** method to retrieve the count for a given word.

```
import java.util.Map;
import java.util.HashMap;

public class ContactList {
  private Map<String, String> contacts;

  public ContactList() {
```

```java
    contacts = new HashMap<>();
  }

  public void addContact(String name, String phoneNumber) {
    contacts.put(name, phoneNumber);
  }

  public String getPhoneNumber(String name) {
    return contacts.get(name);
  }

  public void removeContact(String name) {
    contacts.remove(name);
  }
}
```

In this example, we use a hash table (also implemented as a **HashMap**) to store a list of contacts, where the keys are the names of the contacts and the values are their phone numbers. The **addContact**, **getPhoneNumber**, and **removeContact** methods allow us to insert, retrieve, and remove entries from the contact list.

Hash Tables - Introduction

Usage

- A hash table is a data structure that stores key-value pairs and allows for efficient insertion, deletion, and lookup of the values based on the keys.
- You might use a hash table to implement a cache for frequently accessed data, such as the results of database queries.
- You might use a hash table to implement a spelling checker, where the keys are words and the values are whether or not the word is spelled correctly.
- You might use a hash table to store data for a social network, where the keys are user IDs and the values are the user's profile information.

How hash tables work

- A hash table consists of an array of "*buckets*" and a hash function.
- The hash function maps keys to indices in the array. For example, the hash function might take a string key and return the sum of the ASCII values of the characters in the string modulo the number of buckets.
- When you want to add a key-value pair to the hash table, you use the hash function to determine which bucket the pair should go in.
- When you want to look up the value for a given key, you use the hash function to determine which bucket the key-value pair is in, and then search that bucket for the key.

An example implementation

```
public class HashTable<K, V> {
    // Inner class for representing entries in the hash table
    private static class Entry<K, V> {
      // The key-value pair
      private final K key;
      private V value;
      // Reference to the next entry in the chain (if there is a
collision)
      private Entry<K, V> next;
      public Entry(K key, V value, Entry<K, V> next) {
        this.key = key;
        this.value = value;
        this.next = next;
      }
    }
    // The size of the array (i.e., the number of buckets)
    private final int size;
    // The array of entries (buckets)
    private final Entry<K, V>[] entries;
    @SuppressWarnings("unchecked")
    public HashTable(int size) {
      // Initialize the array of entries
      this.size = size;
      this.entries = new Entry[size];
    }
    public void put(K key, V value) {
      // Determine which bucket the key-value pair should go in
      int index = key.hashCode() % size;
      // Add the key-value pair to the front of the linked list for
that bucket
      entries[index] = new Entry<>(key, value, entries[index]);
    }
    public V get(K key) {
      // Determine which bucket the key should be in
      int index = key.hashCode() % size;
      // Search the linked list for the key
      for (Entry<K, V> entry = entries[index]; entry != null; entry =
entry.next) {
        if (entry.key.equals(key)) {
          // Return the value if the key is found
          return entry.value;
        }
      }
      // Return null if the key is not found
      return null;
    }
  }
```

- **Entry** is a private nested class that represents a key-value pair in the hash table. It has fields for the key, the value, and a reference to the next entry in the chain (if there is a collision).
- **size** is the size of the array (i.e., the number of buckets) in the hash table.
- **entries** is the array of entries in the hash table. Each element of the array represents a bucket, which can contain a linked list of entries that have the same hash code (if there are collisions).
- The **HashTable** constructor takes an integer size as an argument and initializes the size and entries fields.

- The **put** method takes a key and a value as arguments and adds the key-value pair to the hash table. It uses the **hashCode** method of the key and the **%** operator to determine which bucket the key-value pair should go in. If there is already an entry with the same key in the bucket, it updates the value for that entry. Otherwise, it adds a new entry to the front of the linked list for that bucket.
- The **get** method takes a key as an argument and returns the value for that key, if it exists in the hash table. It uses the **hashCode** method of the key and the **%** operator to determine which bucket the key should be in, and then searches the linked list for the key. If it finds the key, it returns the value; otherwise, it returns **null**.

All elements of this code are explained during the lecture. Don’t worry if you don’t understand parts of the code yet.

Hashing

1. Hashing and how it is used in a hash table.
    - Hashing is the process of using a hash function to map keys to indices in an array, called the hash table.
    - The hash function takes a key as input and returns an integer (the hash code) that is used to determine which bucket the key-value pair should go in.
    - The hash function should try to evenly distribute the keys across the array of buckets, also known as the universe (**N**).
2. How the hash function maps keys to indices in the hash table.
    - The hash function applies some transformation to the key to produce an integer (the hash code).
    - The hash code is then reduced to an index in the array of buckets using the modulo operator (**%**). For example, if the hash table has **N** buckets, the index for a given hash code **h** would be **h % N**.
3. Some examples of hash functions.
    - Here are a few examples of simple hash functions:
        - For a string key, you could add up the ASCII values of the characters in the string and return the result modulo **N** to map the hash code to an index in the array of buckets.
        - For an integer key, you could return the key itself modulo **N** to map the hash code to an index in the array of buckets.
        - For a key that consists of multiple fields (such as a person's name and age), you could combine the hash codes of the fields using bitwise operations (e.g., **hashCode = name.hashCode() ^ age**) modulo **N** to map the hash code to an index in the array of buckets.
    - It is important to note that these are just simple examples, and in practice you should choose a hash function that is appropriate for the type of keys you are using and that minimizes the number of collisions.
4. The importance of choosing a good hash function.
    - A good hash function should produce a good distribution of hash codes so that the keys are evenly distributed across the buckets in the universe.
    - If the hash function is not good, there may be too many collisions, which can lead to poor performance (e.g., long chains in the buckets or too many probes in open addressing).

- It is generally a good idea to choose a hash function that is fast to compute and that makes good use of the entropy (randomness) in the keys.

Here are some more detailed examples of the hash functions mentioned before:

Example 1: For a string key **s**, the hash function could be defined as follows:

```
int hash(String s, int N) {
    int h = 0;
    for (int i = 0; i < s.length(); i++) {
      h += s.charAt(i);
    }
    return h % N;
  }
```

This hash function adds up the ASCII values of the characters in the string and returns the result, modulo **N** to map the hash code to an index in the array of buckets.

Example 2: For an integer key **k**, the hash function could be defined as follows:

```
  int hash(int k, int N) {
    return k % N;
  }
```

This hash function returns the key itself, modulo **N** to map the hash code to an index in the array of buckets.

Example 3: For a key that consists of multiple fields (such as a person's name and age), the hash function could be defined as follows:

```
  int hash(String name, int age, int N) {
    return (name.hashCode() ^ age) % N;
  }
```

This hash function combines the hash codes of the fields using a bitwise XOR operation, and then returns the result modulo **N** to map the hash code to an index in the array of buckets.

Example 4: An example of a **BAD** hash function:

```
public int hash(String key) {
    int sum = 0;
    for (int i = 0; i < key.length(); i++) {
        sum += key.charAt(i);
    }
    return sum % N;
}
```

This hash function calculates the sum of the ASCII values of the characters in the key and then uses the modulus operator to map the sum to a bucket in the hash table. While this may seem like a reasonable approach at first glance, it has a number of problems:

- The distribution of the keys is likely to be very poor, as the sum of the ASCII values of the characters may not be evenly distributed among the possible values.
- The hash function does not take into account the size of the universe (number of buckets) or the properties of the keys, which can significantly affect the quality of the hash function.
- The time complexity of the hash function is O(n), as it has to iterate through all the characters in the key to calculate the sum.

Collisions

1. What is a collision is in the context of a hash table.
    - A collision occurs when two or more keys hash to the same index in the array.
    - This means that the hash function maps more than one key to the same bucket in the hash table.
2. The consequences of collisions and how they can affect the performance of the hash table.
    - If there are too many collisions, the hash table may become inefficient because it will have to store multiple key-value pairs in the same bucket. This can lead to long chains of key-value pairs in the buckets, or to too many probes in open addressing.
    - There are a few different ways to handle collisions, but they all involve adding some extra steps when inserting or retrieving key-value pairs from the hash table. These extra steps can increase the time complexity of the hash table operations, which can negatively impact the performance of the hash table.
3. How the probability of a collision depends on the number of keys and the size of the universe (the number of buckets **N**), as well as the quality of the hash function.
    - The probability of a collision is the likelihood that two or more keys will hash to the same index in the array. It is generally a good idea to choose a hash function that has a low probability of collisions.
    - The probability of a collision depends on the number of keys and the size of the universe (**N**), as well as the quality of the hash function.
    - For example, if the universe (**N**) is much smaller than the number of keys, it is more likely that there will be collisions because there are not enough buckets to go around. On the other hand, if the universe is much larger than the number of keys, the probability of a collision will be much lower because there are more buckets available.
    - The quality of the hash function also plays a role in the probability of a collision. A good hash function should evenly distribute the keys across the array of buckets, while a poor hash function may not do this as well.
    - A more formal definition of probability of collisions in a hash table using mathematical notation:

      Let K be the number of keys and N be the number of buckets (the size of the universe) in the hash table. The probability of a collision is the probability that two or more keys will hash to the same index in the array. This can be expressed as:

      P(collision) = 1 - P(no collision)

      Where P(no collision) is the probability that all K keys will hash to different indices in the array. This probability can be calculated as follows:

      P(no collision) = (N / N) * ((N-1) / N) * ... * ((N-K+1) / N) = (N-K+1) / N^K

Substituting this expression into the equation for P(collision) gives:

P(collision) = 1 - (N-K+1) / N^K

This equation can be used to calculate the probability of a collision in a hash table given the number of keys K and the size of the universe (the number of buckets) N.

Example 1: Suppose we have a hash table with 10 buckets (universe **N** = 10) and the following keys and hash codes:

- Key "cat" has a hash code of 3.
- Key "dog" has a hash code of 3.
- Key "bird" has a hash code of 7.

In this example, the keys "cat" and "dog" both hash to the same index (3) in the array, so there is a collision.

Example 2: Suppose we have a hash table with 5 buckets (universe **N** = 5) and the following keys and hash codes:

- Key 12 has a hash code of 2.
- Key 27 has a hash code of 7.
- Key 18 has a hash code of 3.

In this example, there are no collisions because all of the keys hash to different indices in the array.

Example 3: Suppose we have a hash table with 8 buckets (universe **N** = 8) and the following keys and hash codes:

- Key "apple" has a hash code of 1.
- Key "banana" has a hash code of 9.
- Key "cherry" has a hash code of 17.

In this example, the keys "banana" and "cherry" both hash to the same index (1) in the array, so there is a collision.

Collision Resolution in Hash Tables

1. Begin by explaining what collision resolution is and why it is necessary in a hash table.
    - Collision resolution is the process of handling collisions when they occur in a hash table.
    - Collisions can occur when two or more keys hash to the same index in the array.
    - Collision resolution is necessary because it allows the hash table to store multiple key-value pairs in the same bucket without losing information.

2. Discuss the two main methods of collision resolution: separate chaining and open addressing.

    - Separate chaining is a method of collision resolution in which each bucket in the hash table is a separate data structure (e.g., a linked list) that stores the key-value pairs that hash to that bucket.

    - Open addressing is a method of collision resolution in which the key-value pairs are stored directly in the array of buckets. When a collision occurs, the hash table uses a probing sequence to find the next available bucket to store the key-value pair.

3. Separate chaining:
    - In separate chaining, each bucket in the hash table is a separate data structure (such as a linked list) that stores all the key-value pairs that hash to that bucket.
    - To insert a key-value pair into the hash table, the key is hashed to determine the index of the bucket where the pair should be stored. If the bucket is empty, a new linked list is created and the key-value pair is added to it. If the bucket is not empty, the key-value pair is added to the end of the linked list.
    - To search for a key in the hash table, the key is hashed to determine the index of the bucket where the key may be stored. The linked list at that index is then searched for the key. If the key is found, the corresponding value is returned. If the key is not found, null is returned.
    - To delete a key-value pair from the hash table, the key is hashed to determine the index of the bucket where the pair may be stored. The linked list at that index is then searched for the key. If the key is found, the key-value pair is removed from the linked list. If the key is not found, the operation has no effect.

```
// Declare the hash table with an array of linked lists as the buckets.
LinkedList<Entry<K, V>>[] buckets = new LinkedList[N];

// Insert a key-value pair into the hash table.
public void put(K key, V value) {
  int index = hash(key);
  if (buckets[index] == null) {
    buckets[index] = new LinkedList<>();
  }
  buckets[index].add(new Entry<>(key, value));
}

// Retrieve a value from the hash table using a key.
public V get(K key) {
  int index = hash(key);
  if (buckets[index] == null) {
    return null;
  }
  for (Entry<K, V> entry : buckets[index]) {
    if (entry.key.equals(key)) {
      return entry.value;
    }
  }
  return null;
}
```

- **LinkedList<Entry<K, V>>[] buckets**: This line declares an array of linked lists to be used as the buckets in the hash table. Each bucket is a linked list of **Entry** objects, which represent the key-value pairs in the hash table.
- **int index = hash(key)**: This line calculates the index in the array where the key-value pair should be stored. The **hash** function maps the key to an integer value, which is then used to determine the index in the array.
- **if (buckets[index] == null)**: This line checks if there is already a linked list at the index in the array. If there is no linked list, it creates a new one to store the key-value pair.
- **buckets[index].add(new Entry<>(key, value))**: This line adds the key-value pair to the linked list at the index in the array. The **Entry** object stores the key and value as fields.
- **int index = hash(key)**: This line calculates the index in the array where the key-value pair should be retrieved from.
- **if (buckets[index] == null)**: This line checks if there is a linked list at the index in the array. If there is no linked list, it returns **null** to indicate that the key is not in the hash table.
- **for (Entry<K, V> entry : buckets[index])**: This line iterates through the linked list at the index in the array. The **entry** variable represents each **Entry** object in the list.
- **if (entry.key.equals(key))**: This line checks if the key of the **Entry** object matches the key being searched for. If it does, it returns the value of the **Entry** object.

4. Open addressing

- In open addressing, all key-value pairs are stored in the hash table itself, using an array as the underlying data structure.
- To insert a key-value pair into the hash table, the key is hashed to determine the index of the bucket where the pair should be stored. If the bucket is empty, the key-value pair is stored there. If the bucket is not empty, the algorithm uses a probe sequence (such as linear probing or quadratic probing) to find the next empty bucket to store the key-value pair.

- To search for a key in the hash table, the key is hashed to determine the index of the bucket where the key may be stored. The algorithm then follows the probe sequence from that index until it either finds the key or encounters an empty bucket. If the key is found, the corresponding value is returned. If the key is not found, null is returned.
- To delete a key-value pair from the hash table, the key is hashed to determine the index of the bucket where the pair may be stored. The algorithm then follows the probe sequence from that index until it either finds the key or encounters an empty bucket. If the key is found, the key-value pair is marked as deleted (e.g., by using a special "deleted" value or a flag in the Entry object). The key-value pair is not actually removed from the array, but the algorithm will not search through deleted elements during future operations.
- One issue with open addressing is the problem of clashing, which occurs when two keys that are not equal but have the same hash value are stored in the same bucket. This can cause the probe sequence to loop indefinitely and never find an empty bucket. To avoid this, a good hash function should aim to evenly distribute the keys among the buckets and minimize the number of collisions.
- Uses less memory than separate chaining, as all key-value pairs are stored in the same array

```java
// Declare the hash table as an array of entries.
Entry<K, V>[] table = new Entry[N];

// Insert a key-value pair into the hash table.
public void put(K key, V value) {
  int index = hash(key);
  while (table[index] != null) {
    if (table[index].key.equals(key)) {
      table[index].value = value;
      return;
    }
    index = (index + 1) % N;
  }
  table[index] = new Entry<>(key, value);
}

// Retrieve a value from the hash table using a key.
public V get(K key) {
  int index = hash(key);
  while (table[index] != null) {
    if (table[index].key.equals(key)) {
      return table[index].value;
    }
    index = (index + 1) % N;
  }
  return null;
}
```

- **Entry<K, V>[] table**: This line declares an array of **Entry** objects to be used as the hash table. Each **Entry** object represents a key-value pair in the hash table.
- **int index = hash(key)**: This line calculates the index in the array where the key-value pair should be inserted. The **hash** function maps the key to an integer value, which is then used to determine the index in the array.
- **while (table[index] != null)**: This line is the beginning of a loop that continues until an empty bucket is found to store the key-value pair.
- **if (table[index].key.equals(key))**: This line checks if the key of the **Entry** object at the current index in the array matches the key being inserted. If it does, the value of the **Entry** object is updated and the loop terminates.
- **index = (index + 1) % N**: This line calculates the next index in the array to check using open addressing. The **index** is incremented by 1 and then modulo **N** is taken to ensure that the index stays within the bounds of the array.
- **table[index] = new Entry<>(key, value)**: This line stores the key-value pair in the empty bucket (at the **index**) found by the loop. The **Entry** object stores the key and value as fields.
- **int index = hash(key)**: This line calculates the index in the array where the key-value pair should be retrieved from.
- **while (table[index] != null)**: This line is the beginning of a loop that continues until the key is found or an empty bucket is encountered.
- **if (table[index].key.equals(key))**: This line checks if the key of the **Entry** object at the current index in the array matches the key being searched for. If it does, it returns the value of the **Entry** object and the loop terminates.
- **index = (index + 1) % N**: This line calculates the next index in the array to check using open addressing. The **index** is incremented by 1 and then modulo **N** is taken to ensure that the index stays within the bounds of the array.
- **return null**: If the loop completes without finding the key, this line is reached and **null** is returned to indicate that the key is not in the hash table.

Open Addressing and "Probing"

Linear probing:

Linear probing is a probe sequence where the index of the next bucket to check is determined by incrementing the current index by 1. For example:

```
int index = hash(key);
while (table[index] != null) {
    if (table[index].key.equals(key)) {
        // key found, return value
    }
    index = (index + 1) % N;
}
```

Benefits:

- Simple and easy to implement
- Avoids primary clustering (clumps of occupied buckets next to each other)

Downsides:

- Can suffer from secondary clustering (clumps of empty buckets next to each other)
- Can take longer to find an empty bucket or the key being searched for if the hash table is filled

Quadratic probing:

Quadratic probing is a probe sequence where the index of the next bucket to check is determined by incrementing the current index by the square of an increment value. For example:

```
int index = hash(key);
int i = 1;
while (table[index] != null) {
    if (table[index].key.equals(key)) {
        // key found, return value
    }
    index = (index + (i * i)) % N;
    i++;
}
```

Benefits:

- Avoids both primary and secondary clustering

Downsides:

- Can take longer to find an empty bucket or the key being searched for if the hash table is filled
- More complex to implement

Qualitative Measures

There are several ways to measure the quality of a hash table, both qualitatively and quantitatively. Qualitatively, we can measure the performance of a hash table based on the following factors:

1. **Load Factor**: The load factor of a hash table is the ratio of the number of items stored in the table to the number of buckets in the table. A load factor close to 1 indicates a high level of utilization, which can lead to a high number of collisions and poor performance. A load factor close to 0 indicates a low level of utilization, which can lead to wasted space.
2. **Collision Resolution**: The method used to resolve collisions can greatly affect the performance of a hash table. Separate chaining is generally considered to be more efficient than open addressing, as it allows for constant-time insertions and lookups. However, it also requires more memory to store the linked lists.
3. **Hash Function**: The quality of the hash function used can greatly affect the performance of a hash table. A good hash function will distribute the items evenly across the buckets, resulting in a low number of collisions. A poor hash function will result in a high number of collisions, leading to poor performance.
4. **Resizing**: The ability to resize a hash table is important for maintaining good performance as the number of items stored in the table grows. A table that can be resized dynamically can maintain a low load factor and good performance, while a table that cannot be resized will eventually become inefficient as the load factor approaches 1.

Hash Tables: Complexity

- The time complexity of hash table operations (insertion, deletion, and search) depends on the number of key-value pairs stored in the table, the size of the table (i.e., the number of buckets), and the collision resolution strategy used.
- In the worst case, when all keys hash to the same bucket, separate chaining has a time complexity of O(n) for all hash table operations. This is because the linked list in the single bucket will grow as more key-value pairs are inserted, and the entire list must be searched to find a specific key.
- In the average case, when the keys are evenly distributed among the buckets, separate chaining has a time complexity of O(1) for insertion and search, and O(n/m) for deletion, where n is the number of key-value pairs and m is the number of buckets. This is because each key-value pair is stored in a separate bucket, and only the bucket containing the key needs to be searched to find it.
- In the worst case, when all keys hash to the same bucket, open addressing has a time complexity of O(n) for insertion and search, and O(1) for deletion. This is because the bucket containing the key will be full, and all key-value pairs must be searched to find the key or an empty bucket to insert it.
- In the average case, when the keys are evenly distributed among the buckets, open addressing has a time complexity of O(1) for insertion and search, and O(1) for deletion, assuming that a good hash function is used to evenly distribute the keys among the buckets.