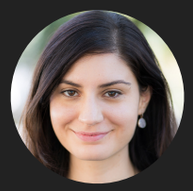
SEARCHING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

21: Hash tables



Alexandra Papoutsaki
she/her/hers

Lecture 21: Hash tables

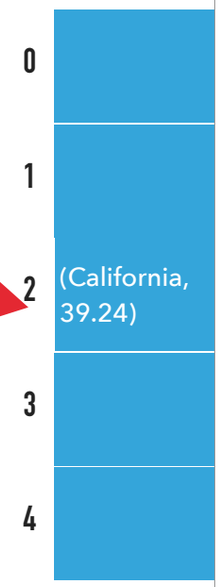
- Hash functions
- Separate chaining
- Open addressing

Some slides adopted from Algorithms 4th Edition or COS226

- Everything we've seen so far about dictionaries has assumed that the keys and values can be of any type, that's why we have used `Key` and `Value` generic types. Here's an interesting alternative idea:
- If we knew that keys were always integers and that these integers were small enough, we could use an array to implement a dictionary.
- Let's say that we want to build a dictionary for key-value pairs, where the keys are small ints; the value type does not matter. Let's assume that we will build the dictionary in an array called `dict` whose size will be the maximum key we expect. For a given key `i`, we would interpret the key as the i-th index in the array `dict`. And the associated value would be saved in the `dict[i]` entry of the array. That would make search, insertion, and deletion constant! We just have to go to the appropriate index and we're done.
- In this lecture, we will consider **hashing**, an extension of this simple method so that we can generalize this technique to keys of any type, not just small integers.

Basic plan for implementing dictionaries using hashing

- Goal: Build a key-indexed array (table or hash table or hash map) to model dictionaries for efficient ($O(1)$ search).
- Hash function: Method to transform key into an array index
  - Also known as the (hash value).
    - `hash("California") = 2`
    - `hash("Texas") = 2 ???`
- Issues:
  - Computing the hash function.
  - Method for checking whether two keys are equal.
  - How to handle collisions when two keys hash to same index.



- Our goal will be to build key-indexed arrays to model dictionaries and do it in such a way that searching a key will be extremely efficient, that is it can be done in constant time!
- Our solution should transform keys, regardless of their type, to array indices in which we will store them along with their associated values. You will often hear me calling this special type of array as table or hash table or hash map. In standard Java libraries, there is a difference between `HashMap` and `HashTable` classes, but in general you can consider all these terms interchangeable: hash tables or hash maps are key-indexed arrays that model dictionaries or symbol tables which hold (key,value) pairs or associations and allow us to search for keys and their associated values in constant time! :) When you want to use Java's built-in implementation of a hashmap, use the `java.util.HashMap` class.
- The first part of building dictionaries that use hashing is to compute what is called a *hash function*.
- A hash function will be a method that will transform a key into an array index or as it is known a *hash value*.
- For example, if we had a dictionary whose keys correspond to US states and the values to their population, we would pass to the hash function our String, let's say "California", and it would give us back an index to the array. In this case, 2. We can now save in index 2 the key along with its associated value("California", 39.24). Now if you want to search for the value associated with "California" you just have to calculate again the hash value (2) and you get back ("California", 39.24) in constant time. If you want to update the existing value, e.g., because the population has changed, you just need to find the appropriate index and substitute its contents with the new value.
- There are a couple of challenges we will face when trying to hash our keys to their hash values.
- First of all, we have to come up with a hash function in the first place which is not an obvious thing to do.
- Dictionaries that are implemented using hashing do not support ordered operations. We map keys to indices but we don't necessarily map smaller keys to smaller array indices. Instead of using `Comparable` keys and the `compareTo` method as we did with BSTs and 2-3/LLRB trees, we are only interested in asking whether two keys are equal. That means that our second challenge is to come up with a technique that will allow us to ask whether two keys are equal.
- The third problem will arise during collisions. Collisions happen when two different keys both hash to the same array index. E.g., if we hash "Texas" to 2, we get a

collision: index 2 has already been occupied by (“California”, 39.24).

Thus, the second part of hashing, after we have calculated the hash value of a key, is to resolve any potential collisions.

Computing hash function

- Ideal scenario: Take any key and uniformly “scramble” it to produce a table/dictionary index.
- Requirements:
  - Consistent - equal keys must produce the same hash value.
  - Efficient - quick computation of hash value.
  - Uniform distribution - every index is equally likely for each key.
- Although thoroughly researched, still problematic in practical applications.
- Examples: Dictionary where keys are social security numbers.
  - Bad: if we choose the first three digits (geographical region and time).
  - Better: if we choose the last three digits.
  - Best: use all data.
- Practical challenge: Need different approach for each key type.

- Now let’s start thinking about the hash function which given any key should tell us what its hash value or array index should be. This means that ideally we seek a function that can easily calculate the hash value. Remember, the whole point of seeing yet another implementation of dictionaries is to accomplish O(1) instead of O(logn) key search!
- The hash functions should also be consistent. If we give it two equal keys it should give us back the same hash value.
- The hashing function should also calculate indices in a uniform fashion: every array index should be equally likely for each key. This will ensure that we avoid collisions as much as we can.
- Mathematicians and computer scientists have researched this last problem in a lot of detail but hashing is still a very much active research area.
- Let’s make these abstract ideas about hashing functions a bit more concrete. Let’s assume that we want to build a dictionary whose keys are social security numbers and its values full names and we want to come up with a hashing function that given a social security number gives us back an array index where we can search, update, or delete it along with its associated value.
- It would probably be a bad idea to use the first three digits of the social security number as a hash function because so many social security numbers will start with the same 3 digits therefore we’ll have a ton of collisions.
- We’d have a better chance using the last three digits.
- But actually, in most cases, we want to find a way to use all of the available data, that is all of the digits of the SSN.
- The real practical challenge with hashing is developing a hash function for every key type. For standard types like Integers, Strings, Doubles and so forth, we can count on the Java developers to implement good hash functions. But if we're going to be allowing dictionaries to hold keys of our own types of data we're going to have to worry about these things in order to get a hash function that's effective and leads to fast search of our keys.

Hashing in Java

- All Java classes inherit a method `hashCode()`, which returns an integer.
- Requirement: If `x.equals(y)` then it should be `x.hashCode()==y.hashCode()`.
- Ideally (but not necessarily): If `!x.equals(y)` then it should be `x.hashCode()!=y.hashCode()`.
- Default implementation: Memory address of x.
  - Need to override *both* `equals()` and `hashCode()` for custom types.
  - Already done for us for `Integer`, `Double`, etc.

- Hashmaps are so widely used for systems programming and applications that some conventions for hashing are built into Java. In particular, all Java classes inherit from the class `Object` a method called `hashCode` which returns a 32-bit int value. Notice, this is not the hash value as we don't have a known hash table size yet. We will eventually have to take that integer and map it to our hash table.
- Here's an interesting requirement for the `hashCode` function: if a key `x` and a key `y` are equal, then their hashCode values should be equal! Conversely, if the `hashCode` values are different, then we know that the keys are not equal.
- Symmetrically, if two keys are not equal, then we'd like it to be that the `hashCode` function gives us two different integers. This is not always the case.
- In Java, the default implementation for hashing an object is returning its memory address of the object. This is not good enough: what if for example we have two different objects with the same content. Since we know they are equal, we would expect their `hashCode` values to also be equal but using the memory address would mean they are not.
- This is why whenever we make our own classes and want to allow for objects of our class to be used as keys in dictionaries, we have to override both the `equals` and `hashCode` method so that they are consistent.

- Java has already done it for standard types like `Integer, Double, String`, etc.

Equality test in Java

- Requirement: For any objects `x`, `y`, and `z`.
  - Reflexive: `x.equals(x)` is `true`.
  - Symmetric: `x.equals(y)` iff `y.equals(x)`.
  - Transitive: if `x.equals(y)` and `y.equals(z)` then `x.equals(z)`.
  - Non-null: if `x.equals(null)` is `false`.
- If you don't override it, the default implementation checks whether `x` and `y` refer to the same object in memory.

- We just said that to be able to implement hash maps that hold keys of our custom types we need to override the equals and hashCode method in our custom classes. Let's start with looking into how we can override the `equals` method by setting a few requirements. Let's assume we have three objects `x`, `y`, and `z`.
- The first requirement for the equality test is that it has to be reflexive. If you ask whether `x` is equal to `x` you should get back that this is true.
- The second requirement for equality testing is that it has to be symmetric. If `x` is equal to `y` then `y` has to be equal to `x`.
- The third requirement, transitive, says that if `x` is equal to `y` and `y` is equal to `z`, then `x` should be equal to `z`.
- We also have a fourth requirement that says that if we compare `x` to `null` we should get false.
- We have seen in the past that the default implementation of the equals method is to check whether `x` and `y` reference the same object in memory.
- Let's see now how we can override this default implementation to incorporate the four requirements of equality tests that we just saw.

Java implementations of `equals()` for user-defined types

```
public class Date {
      private int month;
      private int day;
      private int year;
      …
      public boolean equals(Object y) {
          if (y == this) return true;
          if (y == null) return false;
          if (y.getClass() != this.getClass()) return false;
          Date that = (Date) y;
          return (this.day == that.day &&
                  this.month == that.month &&
                  this.year == that.year);
      }
}
```

- We'll get to see how to override `equals` through an example of a class `Date` that models, well… a date. A date consists of three instance variables of type `int`: a `month`, a `day`, and a `year`.
- If you look into the signature of `equals` you will notice that it returns a `boolean` and takes as a parameter an object `y` of type `Object`. This was a decision made by Java developers so that no matter what the name of your class, since it will always be a descendant of the class `Object`, we would have a consistent signature. Be sure to not alter the signature when implementing the equals (and later the hashCode) method.
- The first `if`-statement actually optimizes our equality test by asking whether `this` object and the `y` object that is passed both refer to the same memory location. If they do, then we can immediately return `true` and be done.
- If not, we will ask whether `y` is `null`. Remember the non-null requirement. `x.equals(null)` should return `false`.
- The third `if`-statement asks whether the name of the class that `y` belongs to is the same with the name of this class (`Date`). Since we allow any type of object, it would totally be legit syntax to write `banana.equals(apple)` but we probably can quickly say that bananas are not apples and they should not be tested for equality.
- Note, you might find implementations of the `equals` method where `instanceof` is used instead of `getClass()`. In that case you'd also accept subtypes.
- Now that we have checked that indeed they are of the same class, we need to write a line that to us looks superfluous but it is needed from the compiler: we need to cast the object `y` to the specific class it belongs to, in this case `Date`. Let's name the casted `y` reference as `that` (see what we did there? :))
- Now, we're at last ready to answer if `this` is equal to `that`. They are equal if all their instance variables are equal. How do we know if they are equal? Since they are primitives, Java is responsible in telling us whether two ints are equal when we use the ==.

General equality test recipe in Java

- Optimization for reference equality.
  - `if (y == this) return true;`
- Check against null.
  - `if (y == null) return false;`
- Check that two objects are of the same type.
  - `if (y.getClass() != this.getClass()) return false;`
- Cast them.
  - `Date that = (Date) y;`
- Compare each significant field.
  - `return (this.day == that.day && this.month == that.month && this.year == that.year);`
  - If a field is a primitive type, use `==`.
  - If a field is an object, use `equals()`.
  - If field is an array of primitives, use `Arrays.equals()`.
  - If field is an area of objects, use `Arrays.deepEquals()`.

- All the things we just discussed for `Date` would generalize for any class you might design.
- Always override `equals` and do the following:
- Optimize for reference equality.
- Check against `null`.
- Check that they are of the same type and cast the passed argument.
- Compare each significant field (variables).
- If the fields are primitives, you can compare them with the ==
- If they are objects, then they should be responsible to tell you if they're equal through their own `equals` method.
- If the fields are arrays of primitives or of objects, then you can use the `equals` and `deepEquals` methods from the `Arrays` class.
- TADA!

Java implementations of hashCode()

```
public final class Integer {
      private final int value;
      …
      public int hashCode() {
            return (value);
      }
}
```

```
public final class Boolean {
      private final boolean value;
      …
      public int hashCode() {
            if(value)    return 1231;
            else return 1237;
      }
}
```

- Now that we've seen how to override the `equals` method, it's time to see how to override the `hashCode` method. But before we see a generic recipe for custom types, I wanted you to get a glimpse of how Java does it for the wrapper classes `Integer` and `Boolean`.
- The `hashCode` value of an `Integer` object is the number it represents. Brilliant? :D
- The `hashCode` value of a `Boolean` object is 1231 if true and 1237 if false.
- Why 1231 and 1237 you may ask?1231 and 1237 are just two (sufficiently large) arbitrary prime numbers. Any other two large prime numbers would do fine. We won't get into the math behind this choice but large prime numbers are always preferred because they minimize collisions.
- You can read in the recommended textbook more examples of how Java implements the `hashCode` method for floating points, arrays, and strings.

Java implementations of hashCode() for user-defined types

```
public class Date {
     private int month;
     private int day;
     private int year;
     …
     public int hashCode() {
         int hash = 1;
         hash = 31*hash + ((Integer) month).hashCode();
         hash = 31*hash + ((Integer) day).hashCode();
         hash = 31*hash + ((Integer) year).hashCode();
         return hash;
         //could be also written as
         //return Objects.hash(month, day, year);
     }
}
```

31x+y rule

- Let's see now how we can implement the `hashCode` method for custom types using again the class `Date` as an example.
- We will use the 31x+y rule in the following way. We start with the `hashCode` value at 1.
- We repeatedly, multiply the `hashCode` value with 31 and add to it the `hashCode` value of each of the fields.
- The use of a small prime integer (31) ensures that the bits of all the fields play a role in the creation of the final `hashCode` value.

General hash code recipe in Java

- Combine each significant field using the 31x+y rule.
- Shortcut 1: use `Objects.hash()` for all fields (except arrays).
- Shortcut 2: use `Arrays.hashCode()` for primitive arrays.
- Shortcut 3: use `Arrays.deepHashCode()` for object arrays.

In general, you can use the 31x+y rule in your own classes or you can use Java's newer shortcuts. `Objects.hash(field1, field2, …)` for all types of fields and again a special case for arrays either of primitives or objects.

Modular hashing

- Hash code: an int between -2^{31} and $2^{31} - 1$
- Hash value: an int between 0 and $m - 1$, where m is the hash table size (typically a prime number or power of 2).
- The class that implements the dictionary of size m should implement a hash function. Examples:
  - ```
    private int hash (Key key){
        return key.hashCode() % m;
    }
    ```
    - Bug! Might map to negative number.
  - ```
    private int hash (Key key){
        return Math.abs(key.hashCode()) % m;
    }
    ```
    - Very unlikely bug. For a hash code of -2^{31}, `Math.abs` will return a negative number!
  - ```
    private int hash (Key key){
        return (key.hashCode() & 0x7fffffff) % m;
    }
    ```
    - Correct.

- We said the the whole point of using hashing was to map a key to an index in the table.
- But so far, what we've seen is that the `hashCode` returns a 32-bit integer, including negative numbers!
- What we should do is take this `hashCode` and map it to a number between `0` and `m-1`, where `m` will be the hash table size - typically `m` is a prime number of power of two. This technique is called modular hashing.
- Therefore, if we were to build a class to implement a hash table, we would need a new method, let's call it `hash`, that given a key, it calls its `hashCode` method and then maps the returned `hashCode` value between `0` to `m-1`. This can be done with the modulo operator `%`.
- The first problem we'll encounter is that a negative `hashCode` value will give us an index out of bounds error.
- We could take the absolute value of it using the `Math.abs` method and then apply modulo `m` operation, but we could still run into a very unlikely bug. For the smallest possible integer that Java can hold, -2^{31}, the absolute value that the method Math.abs will return is a negative number!!!
- The trick is to use this convoluted-looking bitwise-and operation that will always result to a positive number between 0 and m-1.
- Don't worry, you don't have to memorize it! Java has built-in classes for hash tables so you don't need to reinvent the wheel here. Similarly to how we've seen how to implement array lists but in practice we always use the `java.util.ArrayList` class, here too, we see the basic idea of implementing a hashmap but in practice we will be using the `java.util.HashMap`, too.
    ```

Uniform hashing assumption

- Uniform hashing assumption: Each key is equally likely to hash to an integer between 0 and $m - 1$.
- Mathematical model: balls & bins. Toss n balls uniformly at random into m bins.
- Bad news: Expect two balls in the same bin after ~$\sqrt{(\pi m/2)}$ tosses.
  - Birthday problem: In a random group of 23 or more people, more likely than not that two people will share the same birthday.
- Good news: load balancing
  - When $n >> m$, the number of balls in each bin is "likely close" to n/m.

- In an ideal world, our assumption is that each key is equally likely to be hashed to an integer between 0 and `m-1`. And that allows us to model the situation with a so-called "Bins and Balls" model that directly relates the study of hash functions to classical probability theory.
- In the "Bins and Balls" problem, you have `m` bins which correspond to the size of our hash table and `n` balls which correspond to how many keys we need to hash to our table.
- We get these n balls and throw them uniformly at random into the m bins. Unfortunately, it won't take many tosses until two balls fall in the same bin. Why do we care if they fall in the same bin? Because then we have a collision!
- This is similar to a classic combinatorics problem you might have heard, the birthday problem. In a group of >=23 people, it is more likely than not that we get two people who share the same birthday.
- Luckily there are some good news on load balancing. If we have a lot of keys compared to the size of the table (or a lot of balls compared to bins), then under the uniform hashing assumption, each bin will have about n/m balls or each index will have about n/m keys hashed to it.

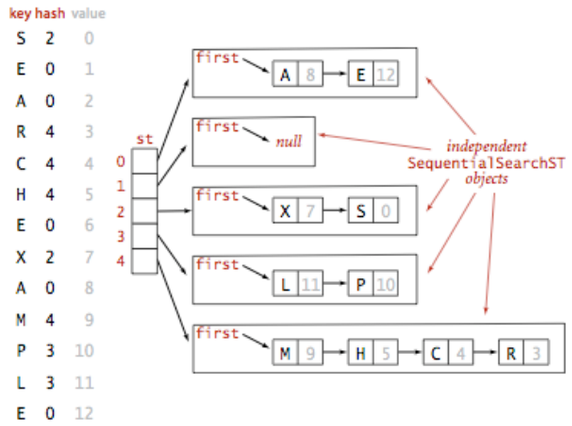
Lecture 21: Hash tables

- Hash functions
- Separate chaining
- Open addressing

- Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2,450 keys are hashed into an array with 1,000,000 , even with a perfectly uniform random distribution, according to the birthday problem there is approximately a 95% chance of at least two of the keys being hashed to the same array index. Therefore, almost all hash table implementations have some collision resolution strategy to handle such events.
- We will see two different techniques that handle collisions: separate chaining and linear probing.

Separate/External Chaining (Closed Addressing)

- Use an array of $m < n$ distinct linked lists (chains)
  [H.P. Luhn, IBM 1953].
  - Hash: Map key to integer i between 0 and $m - 1$.
  - Insert: Put at front of i-th chain (if not already there).
  - Search: Need to only search the i-th chain.



Hashing with separate chaining for standard indexing client

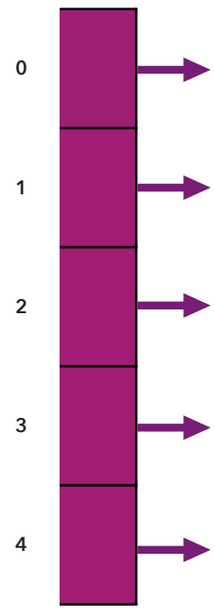
Separate or external chaining or closed addressing is a very simple technique designed in the 1950s. Instead of having an array that every index can hold a unique item, we attach to each index a linked list of all the keys (and their associated values) that have hashed to it.

Separate Chaining Example

- Let's assume we implement a dictionary using hashing and separate chaining for collisions.
- The size of the table is 5, that is $m = 5$.
- We will hash the keys S, E, A, R, C, H, E, X, A, M, P, L, E where I will provide you with their hash values.
- Every time we hash a key, we go to the chain attached to that index and traverse the linked list.
  - If we find a node with the same key we want to insert, we just update its corresponding value.
  - If no node contains our key, we insert the key-value pair at the head of the chain.

- Let's assume that we implement a dictionary using hashing and handling collisions with separate chaining.
- Arbitrarily, I have chosen the hash table size to be 5, that is m=5.
- We will hash the keys S, E, A, R, C, H, E, X, A, M, P, L, E where I will provide you with their hash values.
- Every time we hash a key, we go to the chain attached to that index and traverse the linked list.
- If we find a node with the same key we want to insert, we just update its corresponding value.
- If no node contains our key, we insert the key-value pair at the head of the chain.
- Let's see this in action.

Separate Chaining Example

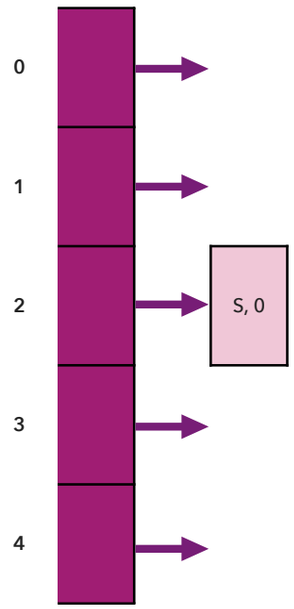


Next step: Insert (S, 0)

- We start with an empty hash table of size 5, where every index points to an empty linked list.

Separate Chaining Example

Key	Hash	Value
S	2	0

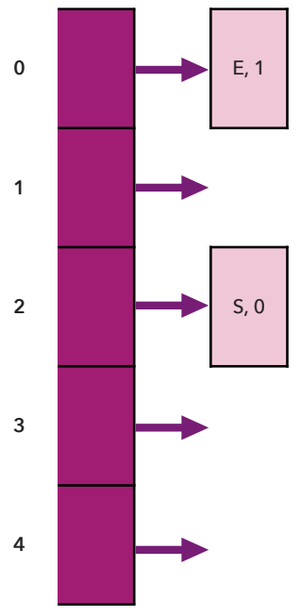


Next step: Insert (E, 1)

- We hash the key S to index 2. The 2-index has an empty linked list so we just put the (S, 0) at the head of the list. 0 is the associated value for key S.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1

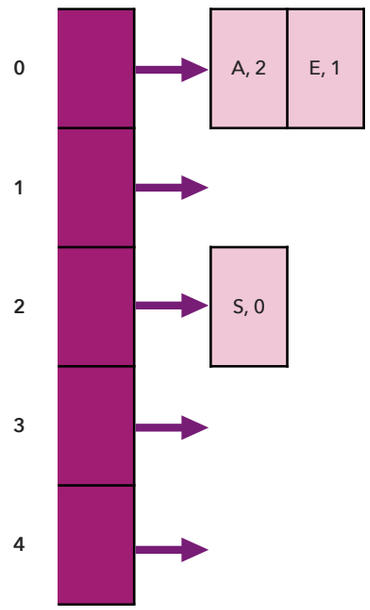


Next step: Insert (A, 2)

- We hash the key E to index 0. The 0-index has an empty linked list so we just put the (E, 1) at the head of the list. 1 is the associated value for key E.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2

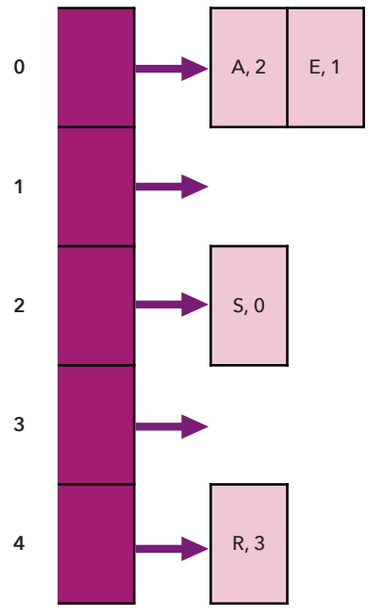


Next step: Insert (R, 3)

- We hash the key A to index 0. The 0-index has a linked list with one node (E,0). Since A is not equal to E, we make a new node (A, 2) and add it to the head of the list attached to the 0-index.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3

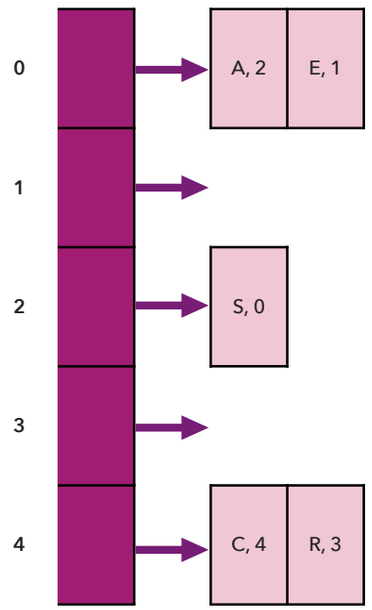


Next step: Insert (C, 4)

- We hash the key R to index 4. The 4-index has an empty linked list so we just put the (R, 3) at the head of the list. 3 is the associated value for key R.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4

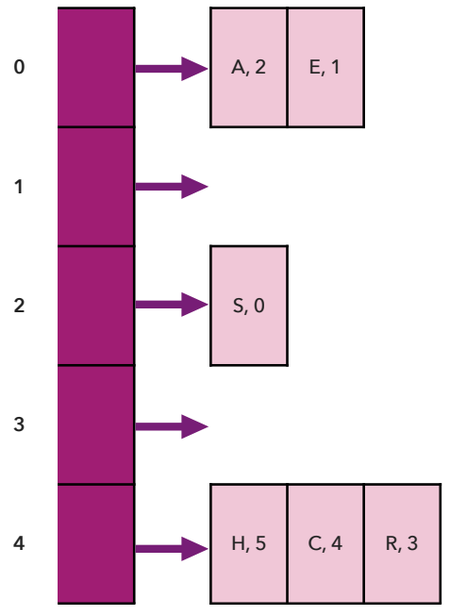


Next step: Insert (H, 5)

- We hash the key C to index 4. The 4-index has a linked list with one node (R,3). Since C is not equal to R, we make a new node (C, 4) and add it to the head of the list attached to the 4-index.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5

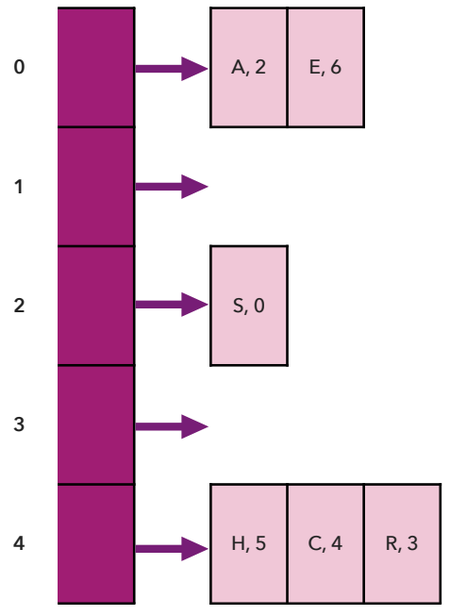


Next step: Insert (E, 6)

- We hash the key H to index 4. The 4-index has a linked list with two nodes, ((C,4), (R,3)). Since H is not equal to either C or R, we make a new node (H, 5) and add it to the head of the list attached to the 4-index.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6

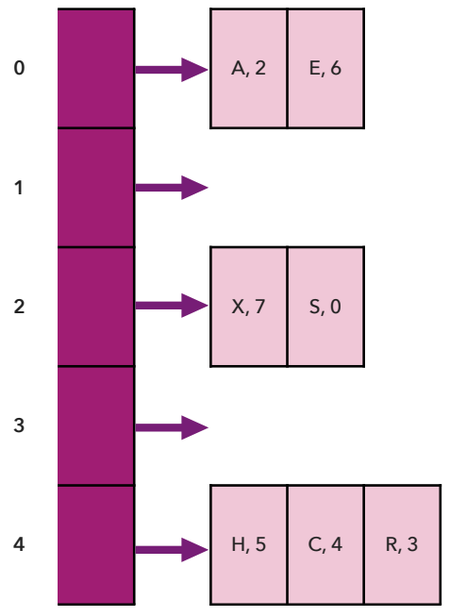


Next step: Insert (X, 7)

- We hash the key E to index 0. The 0-index has a linked list with two nodes, ((A,2), (E,1)). Going over the keys, we find that E is equal to the second node's key. We just update its value from 1 to 6. The 0-index linked list now holds ((A,2), (E,6)).

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7

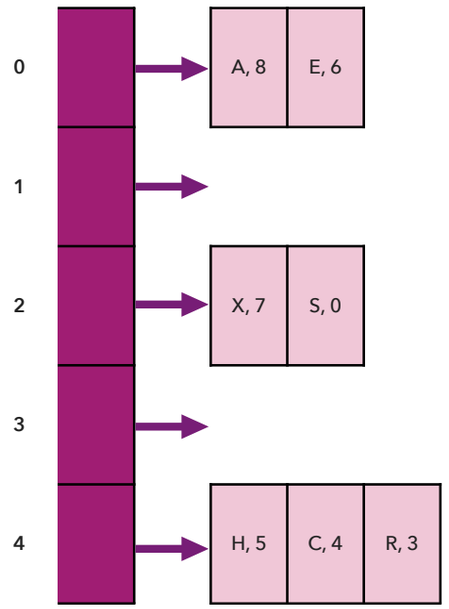


Next step: Insert (A, 8)

- We hash the key X to index 2. The 2-index has a linked list with one nodes, (S,0). Since X is not equal to S, we make a new node (X, 7) and add it to the head of the list attached to the 2-index.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8

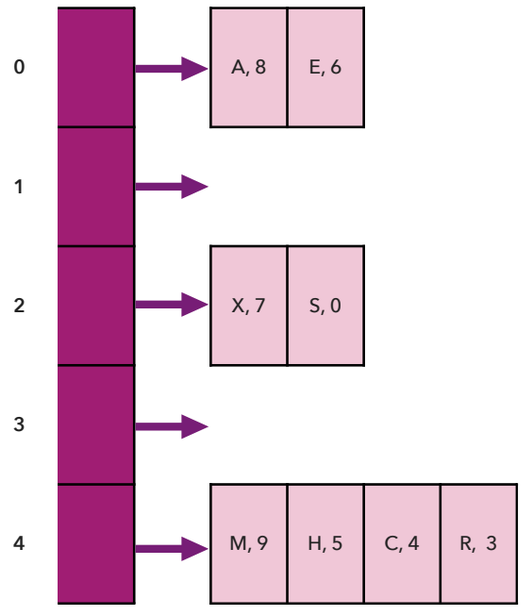


Next step: Insert (M, 9)

- We hash the key A to index 0. The 0-index has a linked list with two nodes, (A,2), (E,6)). Going over the keys, we find that A is equal to the first node's key. We just update its value from 2 to 8. The 0-index linked list now holds ((A,8), (E,6)).

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9

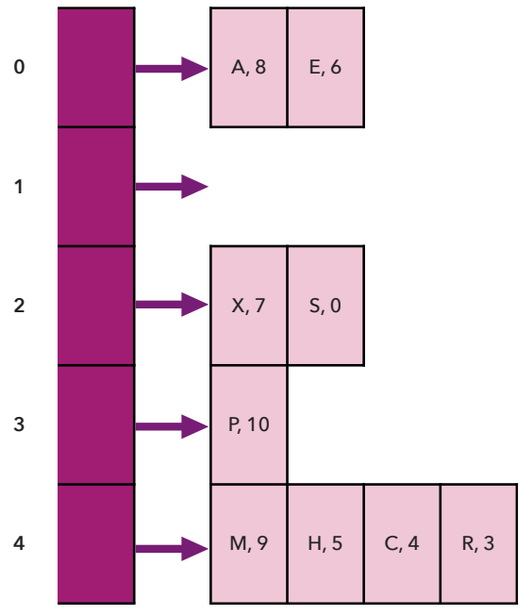


Next step: Insert (P, 10)

- We hash the key M to index 4. The 4-index has a linked list with three nodes, ((H,5),(C,4), (R,3)). Since M is not equal to either H, C or R, we make a new node (M,9) and add it to the head of the list attached to the 4-index.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10

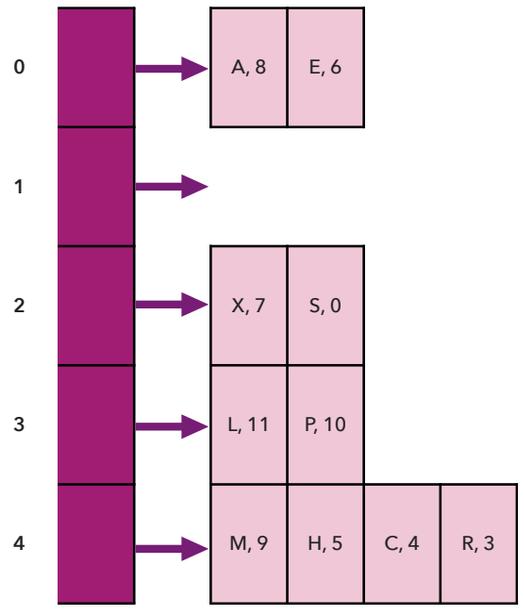


Next step: Insert (L, 11)

- We hash the key P to index 3. The 3-index has an empty linked list so we just put the (P, 10) at the head of the list. 10 is the associated value for key P.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11

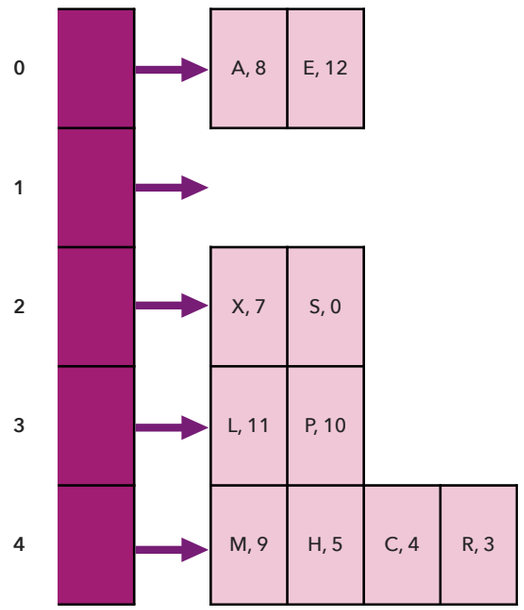


Next step: Insert (E, 12)

- We hash the key L to index 3. The 3-index has a linked list with one node, (P,10). Since L is not equal to either P, we make a new node (L,11) and add it to the head of the list attached to the 3-index.

Separate Chaining Example

Key	Hash	Value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12



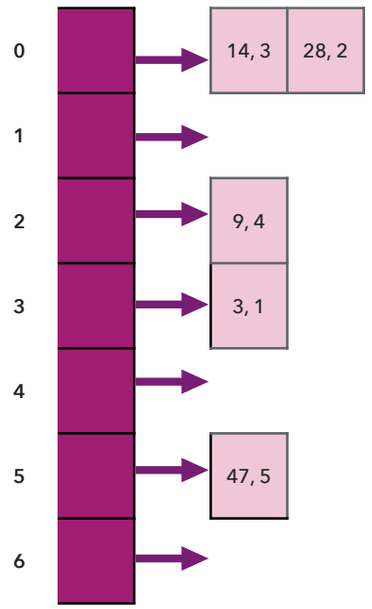
- We hash the key E to index 0. The 0-index has a linked list with two nodes, ((A,8), (E,6)). Going over the keys, we find that E is equal to the second node's key. We just update its value from 6 to 12. The 0-index linked list now holds ((A,8), (E,12)).
- And we're done!

Practice Time

- Assume a dictionary implemented using hashing and separate chaining for handling collisions.
- Let $m = 7$ be the hash table size.
- For simplicity, we will assume that keys are integers and that the hash value for each key k is calculated as $h(k) = k \,\%\, m$.
- Insert the key-value pairs (47, 0), (3, 1), (28, 2), (14, 3), (9,4), (47,5) and show the resulting dictionary.

Answer

Key	Hash	Value
47	5	0
3	3	1
28	0	2
14	0	3
9	2	4
47	5	5



Dictionary with separate chaining implementation

```
public class SeparateChainingLiteHashST<Key, Value> {

    private int m = 128;  // hash table size
    private Node[] st = new Node[m];
    // array of linked-list dictionaries/s.
    //Node is inner class that holds keys and values of type Object

    public Value get(Key key) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next;)
            if (key.equals(x.key)) return (Value) x.val;
        return null;
    }

    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next;)
            if (key.equals(x.key)) {
                x.val = val;
                return;
        }
        st[i] = new Node(key, val, st[i];
    }
```

This is how an implementation of hash tables using separate chaining to handle collisions could look in Java.

Analysis of Separate Chaining

- Under uniform hashing assumption, if n keys to hash in a table with size m, the length of each chain is ~n/m.
- Consequence: Number of probes (calls to either `equals()` or `hashCode()`) for search/insert is proportional to n/m (m times faster than sequential search in a single chain).
  - m too large -> too many empty chains.
  - m too small -> chains too long.
  - Typical choice: m~$1/4n$ -> constant time per operation.

- Let’s see some basic facts about the runtime complexity of handling collisions with separate chaining.
- Under the uniform hashing assumption, the length of each chain/linked list is about `n/m`, where `n` is the number of hashed keys, and `m` the hash table size.
- As a consequence, the number of probes (that is calls either to equals() or hashCode()) for search and insert is proportional to n/m (that is m times faster than sequential search in a simple array of capacity n).
- This has a couple of implications: If the hash table size is too large then we have too many empty chains and waste space for nothing. If it is too small, then the linked lists become too long and we waste time searching through them.
- A typical choice is to keep the hash table around a quarter of the hashed keys. This warranties an almost constant time complexity both for search and insertion.

Resizing in a separate-chaining hash table

- Goal: Average length of chain n/m = constant lookup.
  - Double hash table size when $n/m \geq 8$.
  - Halve hash table size when $n/m \leq 2$.
  - Need to rehash all keys when resizing (hashCode value for key does not change, but hash value changes as it depends on table size).

- Our goal with hash table is to have constant lookup (search).
- Things when n/m is larger than 4 become slow and we'd need a larger hashtable.
- A rule of thumb is to double the hash table size when n/m >=8 and half it when it is underutilized that is n/m <=2.
- Notice that for a new `m` we'd need to re-hash all of our keys.

Parting thoughts about separate-chaining

- Deletion: Easy! Hash key, find its chain, search for a node that contains it and remove it.
- Ordered operations: not supported. Instead, look into (balanced) BSTs.
- Fastest and most widely used dictionary implementation for applications where key order is not important.

- Here are some parting thoughts about
- Deletion is quite straightforward. Just hash the key, find the chain it should part of, search for a node that contains that key and remove the node from the linked list.
- The whole point of hashing is to uniformly disperse keys so any order in the keys is lost. If you need to quickly find the maximum or minimum key, find keys, etc, then hashing is **not.**
- Hashing with separate chaining is easy to implement and probably the fastest and most widely used dictionary implementation for applications where key order is not important.

Lecture 21: Hash tables

- Hash functions
- Separate chaining
- Open addressing

Next, we consider an alternative scheme for collision resolution: linear probing.

Linear Probing

- Belongs in the open addressing family.
- Alternate approach to handle collisions when $m > n$.
- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot and put it there.
- If the array is full, the search would not terminate.

- Linear probing is another approach to implementing hashing that handles collisions that belongs in the general family of open addressing techniques.
- The idea is that when the hash table size m is greater than the number of keys n (m>n), then we can take advantage of any open indices and place there keys that when hashed create collisions.
- We will accomplish this by maintaining the keys and values in. Two parallel arrays.
- After we hash a key, if we find the index that corresponds to the hash value to be occupied, we will search for next available empty slot and place our key there (and the value in the same index in the parallel array).
- If the array is full, the search would not terminate. This is why it is important that m>n.

Linear Probing

- Hash: Map key to integer i between 0 and $m - 1$.
- Insert: Put at index i if free. If not, try $i + 1$, $i + 2$, etc.
- Search: Search table index i. If occupied but no match, try $i + 1$, $i + 2$, etc
  - If you find a gap then you know that it does not exist.
- Table size m **must** be greater than the number of key-value pairs n.

- There are more advanced open addressing that find open slots, but linear probing is truly the simplest. If we has a key to index i and i is taken, then we try i+1, i+2, … (wrapping around with the % operator). This applies both to search and insertion.
- For search, if we find a gap, then we know that the key we're looking for cannot exist.
- Again, all this can happen only if m>n.

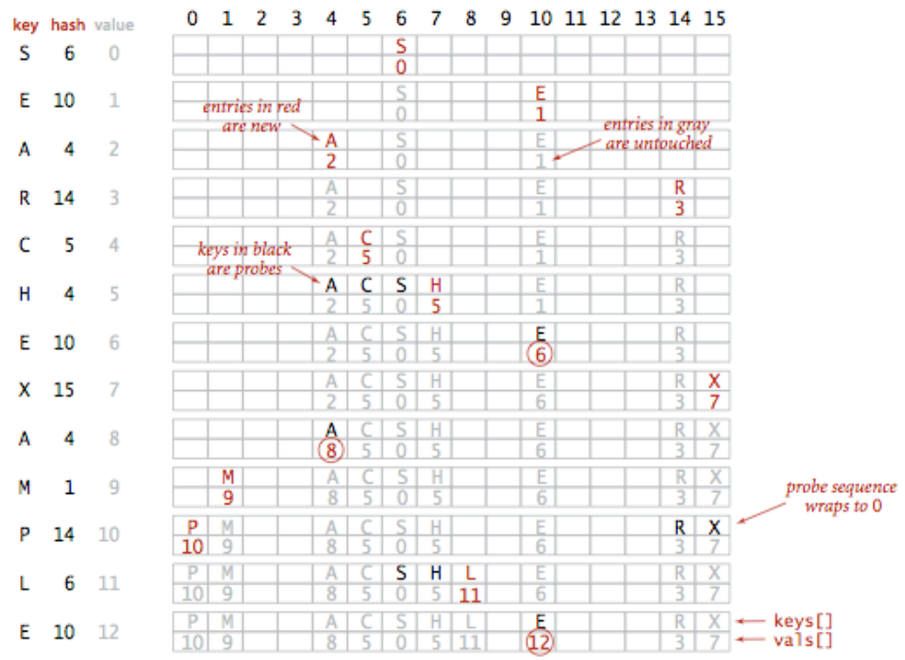
Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

3.4 LINEAR PROBING DEMO

Let’s see linear probing in action through this video.

Linear Probing Example



Trace of linear-probing ST implementation for standard indexing client

You can also see how linear probing works in this figure. You might have noticed that after a while most keys will cluster in a specific area of the hashtable. This problem is called **primary clustering.**

Practice time

- Assume a dictionary implemented using hashing and linear probing for handling collisions.
- Let $m = 7$ be the hash table size.
- For simplicity, we will assume that keys are integers and that the hash value for each key k is calculated as $h(k) = k \,\%\, m$.
- Insert the key-value pairs (47, 0), (3, 1), (28, 2), (14, 3), (9,4), (47,5) and show the resulting dictionary.

- Let’s practice now with the same sequence of key-value pairs in a dictionary implemented using hashing and linear probing for handling collisions

Answer

Key	Hash	Value
47	5	0
3	3	1
28	0	2
14	0	3
9	2	4
47	5	5

Keys	28	14	9	3		47	
Values	2	3	4	1		5	
Indices	0	1	2	3	4	5	6

- The resulting array can be seen at the bottom. Notice that since hashing 14 led to a collision, we inserted the key (and its associated value) in the next available position, at index 1.

Dictionary with linear probing implementation

```
public class LinearProbingHashST<Key, Value> {

    private int m = 32768;  // hash table size
    private Value[] Vals = (Value[]) new Object[m];
    private Key[] Vals = (Key[]) new Object[m];

    public Value get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i+1) % m;)
            if (key.equals(keys[i])) return vals[i];
        return null;
    }

    public void put(Key key, Value val) {
        int i;
        for (int i = hash(key); keys[i] != null; i = (i+1) % m;)
            if (key.equals(keys[i])){
                break;
        }
        keys[i] = key;
        vals[i] = val;
    }
```

- The code for implementing a hash table with linear probing is quite simple.
- We hold two parallel arrays for the keys and values.
- The search starts at the index that the key is hashed and keeps looking to the right (wrapping around %m). If it finds a gap that means that the search fail and it returns null.
- Insertion works similarly. We start at the hash value. If it is occupied, we keep going to the right. If we find the key we break. Either way we will have stopped either at a gap where we insert a new key and value or at an index with the same key where we just update the value.

Primary clustering

- Cluster: a contiguous block of keys.
- Observation: new keys likely to hash in middle of big clusters.

As we saw before, linear probing suffers of the problem of primary clustering which leads to slowdowns.

Analysis of Linear Probing

- Proposition: Under uniform hashing assumption, the average number of probes in a linear-probing hash table of size m that contains $n = \alpha m$ keys is at most
  - $1/2(1 + \frac{1}{1-a})$ for search hits and
  - $1/2(1 + \frac{1}{(1-a)^2})$ for search misses and insertions.
  - [Knuth 1963]
- Parameters:
  - m too large -> too many empty array entries.
  - m too small -> search time becomes too long.
  - Typical choice for load factor: $\alpha = n/m \sim 1/2$ -> constant time per operation.

- The analysis of the average number of probes (calls to `equals` or `hashCode`) is quite complicated and depends on the load factor of the hashtable, that is a=n/m.
- What you need to remember is that when m is too large then we waste a lot of space
- When m is too small we have a lot of collisions and search time becomes too long, that is it degrades to linear.
- A good choice for the load factor is to keep it at 0.5. Then we end up with an average constant time for insertion and search.

Resizing in a linear probing hash table

- Goal: Fullness of array (load factor) $n/m \leq 1/2$.
  - Double hash table size when $n/m \geq 1/2$.
  - Halve hash table size when $n/m \leq 1/8$.
  - Need to rehash all keys when resizing (hash code does not change, but hash value changes as it depends on table size).
  - Deletion not straightforward.

- Similarly to separate chaining, we might need to change the size of our hashtable.
- In general, we want to keep the load factor <0.5. If greater, then double and rehash. If <1/8 then halve and rehash.
- Deletion in open addressing techniques is quite complicated. The easiest way to implement delete is to find and remove the key–value pair and then to reinsert all of the key–value pairs in the same cluster that appear after the deleted key–value pair. If the hash table doesn't get too full, the expected number of key–value pairs to reinsert will be a small constant. An alternative is to flag the deleted linear-probing table entry so that it is skipped over during a search but is used for an insertion. If there are too many flagged entries, create a new hash table and rehash all key–value pairs.

Quadratic Probing

- Another open addressing technique that aims to reduce primary clustering by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.
- Modify the probe sequence so that $h(k, i) = (h(k) + c_1 i + c_2 i^2) \,\%\, m, c_2 \neq 0$, where i is the i-th time we have had a collision for the given index.
  - When $c_2 = 0$, then quadratic probing degrades to linear probing.

- Under the open addressing techniques, we have one more technique that tries to resolve collisions while aiming to avoid primary clustering issues. This technique is known as quadratic probing.
- If h(k) is the base hash value calculated for a key k and we have a collision, then we add successive values of a polynomial until we find an open slot.
- We won’t be getting in this technique and its analysis in depth but I want you to know how it works.

Quadratic probing - Example

- $h(k) = k \,\%\, m$ and $h(k, i) = (h(k) + i^2) \,\%\, m$.
- Assume $m = 13$, and key-value pairs to insert: (17,0), (33,1), (18,2), (20,3), (44,4), (11,5), (19,6), (7,7).

	0	1	2	3	4	5	6	7	8	9	10	11	12	
(17,0)					17									
(33,1)					17			33						
(18,2)					17	18		33						
(20,3)					17	18		33	20					Collision!
(44,4)					17	18	44	33	20					Collision!
(11,5)					17	18	44	33	20			11		
(19,6)					17	18	44	33	20		19	11		Collision!
(7,7)				7	17	18	44	33	20		19	11		Collision!

Let’s see what happens under quadratic probing given the above hashing functions.

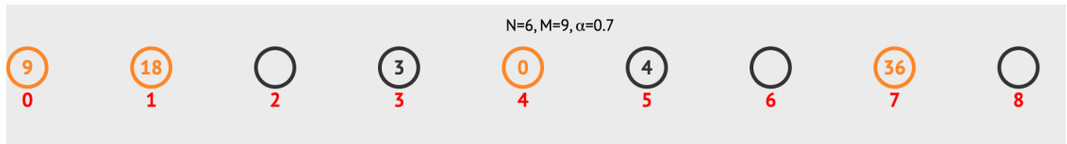
PRACTICE TIME

- $h(k) = k \,\%\, m$ and $h(k, i) = (h(k) + i^2) \,\%\, m$.
- Assume $m = 9$, and key-value pairs to insert: (3,0), (9,1), (18,2), (0,3), (4,4), (36,5).

Let’s see what happens under quadratic probing given the above hashing functions.

ANSWER

- $h(k) = k \,\%\, m$ and $h(k, i) = (h(k) + i^2) \,\%\, m$.
- Assume $m = 9$, and key-value pairs to insert: (3,0), (9,1), (18,2), (0,3), (4,4), (36,5).



Let’s see what happens under quadratic probing given the above hashing functions.

Summary for dictionary operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$
2-3 search tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
LLRB tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Separate chaining	n	n	n	1	1	1
Open addressing	n	n	n	1	1	1

- To summarize, if you want to implement a dictionary you have a few choices.
- BSTs are easy and offer ordered operations but can degrade to linear height.
- 2-3 search trees are a bit more complicated but stay balanced and allow for logarithmic search and insertion. They also support ordered operations.
- Hashtables either with separate chaining or open addressing collision handling offer constant search, insertion, and deletion on average and linear at the worst case. They do not support ordered operations.
- In general, separate chaining is preferred to open addressing. In practice, hash tables handle collisions with more advanced techniques these days, such as double hashing or cuckoo hashing.

Hash tables vs balanced search trees

- Hash tables:
  - Simpler to code.
  - No effective alternative of unordered keys.
  - Faster for simple keys (a few arithmetic operations versus $\log n$ compares).
- Balanced search trees:
  - Stronger performance guarantee.
  - Support for ordered dictionary operations.
  - Easier to implement `compareTo()` than `hashCode()`.
- Java includes both:
  - Balanced search trees: `java.util.TreeMap`, `java.util.TreeSet`.
  - Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

- This slide summarizes the differences between dictionaries implemented with hash tables versus with balanced search trees such as 2-3 search trees.
- Java includes standard implementations both for balanced BSTs and hash tables.
- `HashMap` handles collisions with separate chaining while `IdentityHashMap` with linear probing.
- As with everything in CS62, it all boils down to understanding the requirements of your application. If you require ordered operations then go for BSTs. Same if you want guaranteed logarithmic performance. If this is not important, hash tables will be much faster.

Lecture 21: Hash tables

- Hash functions
- Separate chaining
- Open addressing

- And with that, we have successfully covered the course unit on searching. We started with implementing dictionaries with BSTs, continued with balanced BSTs (2-3 search trees), and we concluded with the fastest of them all, hash tables.
- In the future, keep hash tables in mind during interviews, they tend to be favored as data structures because of their average constant lookup time.

Readings:

- Recommended Textbook: Chapter 3.4 (Pages 458-477)
- Website:
  - https://algs4.cs.princeton.edu/34hash/
- Visualization:
  - https://visualgo.net/en/hashtable

- All the information that we covered can be found in more detail on the course textbook and its website. Don't forget to practice the visualization website.

Problem 1

- Insert the keys E, A, S, Y, Q, U, T, I, O, N in that order into an initially empty table of m=5 lists, using separate chaining. Use the hash function 11*k%m to transform the 4th letter of the English alphabet into a table index.

ANSWER 1

- Insert the keys E, A, S, Y, Q, U, T, I, O, N in that order into an initially empty table of m=5 lists, using separate chaining. Use the hash function 11*k%m to transform the 4th letter of the English alphabet into a table index.
- 0 -> E -> Y -> T -> O
- 1 -> A -> U
- 2 -> Q
- 3 -> null
- 4 -> S -> I -> N