Hash Tables

Victor Milenkovic

Department of Computer Science
University of Miami

CSC220 Programming II – Spring 2024

Implementations of Map

Implementations of Map

The goal is to implement a Map,

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

We have done six implementations:

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

We have done six implementations:

- Unsorted array: add O(1), find O(n), remove O(1).

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

We have done six implementations:

- Unsorted array: add $O(1)$, find $O(n)$, remove $O(1)$.
- Sorted array: add $O(n)$, find $O(\log n)$, remove $O(n)$.

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

We have done six implementations:

- Unsorted array: add O(1), find O(n), remove O(1).
- Sorted array: add O(n), find O(log n), remove O(n).
- Sorted linked list: add, find, remove, all O(n).

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

We have done six implementations:

- Unsorted array: add O(1), find O(n), remove O(1).
- Sorted array: add O(n), find O(log n), remove O(n).
- Sorted linked list: add, find, remove, all O(n).
- Unbalanced binary search tree: add, find, remove, all O(log n) expected time for input in random order but all O(n) worst case time on ordered input.

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

We have done six implementations:

- Unsorted array: add O(1), find O(n), remove O(1).
- Sorted array: add O(n), find O(log n), remove O(n).
- Sorted linked list: add, find, remove, all O(n).
- Unbalanced binary search tree: add, find, remove, all O(log n) expected time for input in random order but all O(n) worst case time on ordered input.
- Skip List (SkipMap): add, find, and remove in O(log n) expected time on any input.

Implementations of Map

The goal is to implement a Map,

- which is the formal name for what a phone directory does:
- give it a name and it gives you a number or address or whatever.
- Of course, we know that the proper terms are key and value.

We have done six implementations:

- Unsorted array: add O(1), find O(n), remove O(1).
- Sorted array: add O(n), find O(log n), remove O(n).
- Sorted linked list: add, find, remove, all O(n).
- Unbalanced binary search tree: add, find, remove, all O(log n) expected time for input in random order but all O(n) worst case time on ordered input.
- Skip List (SkipMap): add, find, and remove in O(log n) expected time on any input.
- 2-3 Tree (BTree): add, find, and remove in O(log n) worst case time.

Even Faster!

Those last ones were a bit complicated.

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,
- and it's pretty simple too!

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,
- and it's pretty simple too!

The trick is to store each key at essentially a random location in the array.

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,
- and it's pretty simple too!

The trick is to store each key at essentially a random location in the array. Since they are not in alphabetical order, it is not necessary to move O(n) entries when adding or removing "Aaron".

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,
- and it's pretty simple too!

The trick is to store each key at essentially a random location in the array. Since they are not in alphabetical order, it is not necessary to move O(n) entries when adding or removing "Aaron".
Why did I bother with binary search trees, skip lists, and 2-3 trees?

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,
- and it's pretty simple too!

The trick is to store each key at essentially a random location in the array. Since they are not in alphabetical order, it is not necessary to move O(n) entries when adding or removing "Aaron".
Why did I bother with binary search trees, skip lists, and 2-3 trees?

- Sometimes it is important that the entries be in alphabetical or numerical order.

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,
- and it's pretty simple too!

The trick is to store each key at essentially a random location in the array. Since they are not in alphabetical order, it is not necessary to move O(n) entries when adding or removing "Aaron".

Why did I bother with binary search trees, skip lists, and 2-3 trees?

- Sometimes it is important that the entries be in alphabetical or numerical order.
- When you look up a name, it is nice to be able to go forward or back a few names in case you misspelled it.

Even Faster!

Those last ones were a bit complicated.

- You would think it would be impossible to do better or at least very complicated.
- BUT it is possible to do add, find, and remove in O(1) expected time,
- and it's pretty simple too!

The trick is to store each key at essentially a random location in the array. Since they are not in alphabetical order, it is not necessary to move O(n) entries when adding or removing "Aaron".
Why did I bother with binary search trees, skip lists, and 2-3 trees?

- Sometimes it is important that the entries be in alphabetical or numerical order.
- When you look up a name, it is nice to be able to go forward or back a few names in case you misspelled it.
- If you want Milenkovic in a hash table, you better not look for Milenkovich because it will be far far away.

Google

This may not seem like a big deal, but think about how Google finds new web pages.

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.
- A (B+) tree will do this. A hash table won't.

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.
- A (B+) tree will do this. A hash table won't.
- Even though a tree has multiple levels, if you are working in a tiny subtree of the tree, as in this example, your running time is more like O(1),

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.
- A (B+) tree will do this. A hash table won't.
- Even though a tree has multiple levels, if you are working in a tiny subtree of the tree, as in this example, your running time is more like O(1),
- with a small constant factor because it's all in one place on one hard disk.

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.
- A (B+) tree will do this. A hash table won't.
- Even though a tree has multiple levels, if you are working in a tiny subtree of the tree, as in this example, your running time is more like O(1),
- with a small constant factor because it's all in one place on one hard disk.

If you used a huge hash table,

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.
- A (B+) tree will do this. A hash table won't.
- Even though a tree has multiple levels, if you are working in a tiny subtree of the tree, as in this example, your running time is more like O(1),
- with a small constant factor because it's all in one place on one hard disk.

If you used a huge hash table,

- http://www.cs.miami.edu/ vjm/csc220/prog01/lab.txt

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.
- A (B+) tree will do this. A hash table won't.
- Even though a tree has multiple levels, if you are working in a tiny subtree of the tree, as in this example, your running time is more like O(1),
- with a small constant factor because it's all in one place on one hard disk.

If you used a huge hash table,

- http://www.cs.miami.edu/ vjm/csc220/prog01/lab.txt
- would be on a completely different hard disk (server, US state) than

Google

This may not seem like a big deal, but think about how Google finds new web pages.

- As soon as it finds one new web age, probably the home page of a new company, it will following links from it to all the other pages for that same company.
- These are all in the same domain with very similar URLs.
  - www.cs.miami.edu/home/vjm/prog03/Fib.java
  - www.cs.miami.edu/home/vjm/prog06/SkipMap.java
- It makes a lot of sense to store similar URLs next to each other on a hard disk so you can load and save them all quickly.
- A (B+) tree will do this. A hash table won't.
- Even though a tree has multiple levels, if you are working in a tiny subtree of the tree, as in this example, your running time is more like O(1),
- with a small constant factor because it's all in one place on one hard disk.

If you used a huge hash table,

- http://www.cs.miami.edu/ vjm/csc220/prog01/lab.txt
- would be on a completely different hard disk (server, US state) than
- http://www.cs.miami.edu/ vjm/csc220/prog02/lab.txt.

hashCode

We will store the entries in an array.

hashCode

We will store the entries in an array.

- What “random” index should we use for "Milenkovic"?

hashCode

We will store the entries in an array.

- What “random” index should we use for "Milenkovic"?
- Built in classes have a hashCode method.

hashCode

We will store the entries in an array.

- What “random” index should we use for "Milenkovic"?
- Built in classes have a hashCode method.
- We will look at the one for String:

```
public int hashCode()
Returns a hash code for this string. The hash code for a
String object is computed as
 s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

 using int arithmetic, where s[i] is the ith character of the
 string, n is the length of the string, and ^ indicates
 exponentiation. (The hash value of the empty string is zero.)
```

hashCode

We will store the entries in an array.

- What “random” index should we use for "Milenkovic"?
- Built in classes have a hashCode method.
- We will look at the one for String:

```
public int hashCode()
Returns a hash code for this string. The hash code for a
String object is computed as
 s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

 using int arithmetic, where s[i] is the ith character of the
 string, n is the length of the string, and ^ indicates
 exponentiation. (The hash value of the empty string is zero.)
```

WHAT???

Letters are Numbers

Remember that everything is bits.

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + "  " + n);
}
```

I get:

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108
- e 101

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108
- e 101
- n 110

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108
- e 101
- n 110
- k 107

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108
- e 101
- n 110
- k 107
- o 111

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108
- e 101
- n 110
- k 107
- o 111
- v 118

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108
- e 101
- n 110
- k 107
- o 111
- v 118
- i 105

Letters are Numbers

Remember that everything is bits.
So the letter 'M' is really just a small integer.
If I run:

```
String name = "Milenkovic";
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  int n = c;
  System.out.println(c + " " + n);
}
```

I get:

- M 77
- i 105
- l 108
- e 101
- n 110
- k 107
- o 111
- v 118
- i 105
- c 99

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is
- 'c' * 31 * 31 + 'a' * 31 + 't'

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is
- 'c' * 31 * 31 + 'a' * 31 + 't'
- Start with zero. Multiply by 31 and add 'c':

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is
- 'c' * 31 * 31 + 'a' * 31 + 't'
- Start with zero. Multiply by 31 and add 'c':
- 'c'

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is
- 'c' * 31 * 31 + 'a' * 31 + 't'
- Start with zero. Multiply by 31 and add 'c':
- 'c'
- Multiply by 31 and add 'a':

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is
- 'c' * 31 * 31 + 'a' * 31 + 't'
- Start with zero. Multiply by 31 and add 'c':
- 'c'
- Multiply by 31 and add 'a':
- 'c' * 31 + 'a'

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is
- 'c' * 31 * 31 + 'a' * 31 + 't'
- Start with zero. Multiply by 31 and add 'c':
- 'c'
- Multiply by 31 and add 'a':
- 'c' * 31 + 'a'
- Multiply by 31 and add 't':

Hash Code

Here is my hash code:

```
System.out.println("Hash code of " + name +
                   " is " + name.hashCode());
```

Hash code of Milenkovic is -1110834957
It's negative? How can that be?

- Let's start with a simple example. The hash code of "cat" is
- 'c' * 31 * 31 + 'a' * 31 + 't'
- Start with zero. Multiply by 31 and add 'c':
- 'c'
- Multiply by 31 and add 'a':
- 'c' * 31 + 'a'
- Multiply by 31 and add 't':
- 'c' * 31 * 31 + 'a' * 31 + 't'

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360
- code = 2398261

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360
- code = 2398261
- code = 74346201

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360
- code = 2398261
- code = 74346201
- code = -1990234958

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360
- code = 2398261
- code = 74346201
- code = -1990234958
- code = -1567741443

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360
- code = 2398261
- code = 74346201
- code = -1990234958
- code = -1567741443
- code = -1355344359

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360
- code = 2398261
- code = 74346201
- code = -1990234958
- code = -1567741443
- code = -1355344359
- code = 933997936

Hash Code

Let's do the same trick with "Milenkovic" and print out each step:

```
int code = 0;
for (int i = 0; i < name.length(); i++) {
  char c = name.charAt(i);
  code = 31 * code + c;
  System.out.println("code = " + code);
}
```

- code = 77
- code = 2492
- code = 77360
- code = 2398261
- code = 74346201
- code = -1990234958
- code = -1567741443
- code = -1355344359
- code = 933997936
- code = -1110834957

Negative Numbers??

By the time we get to adding the ’k’,

Negative Numbers??

By the time we get to adding the ’k’,

- multiplying the code by 31 makes it bigger than 2 billion,

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or
- $-2^{31} = -2147483648$

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or
- $-2^{31} = -2147483648$
- and back up to zero.

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or
- $-2^{31} = -2147483648$
- and back up to zero.
- For example, what is $11111111111111111111111111111111_2$?

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or
- $-2^{31} = -2147483648$
- and back up to zero.
- For example, what is $11111111111111111111111111111111_2$?
- It's the sum of two other numbers on this slide.

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or
- $-2^{31} = -2147483648$
- and back up to zero.
- For example, what is $11111111111111111111111111111111_2$?
- It's the sum of two other numbers on this slide.
- And if you add 1, and the overflow "goes away", you get $00000000000000000000000000000000_2$.

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or
- $-2^{31} = -2147483648$
- and back up to zero.
- For example, what is $11111111111111111111111111111111_2$?
- It's the sum of two other numbers on this slide.
- And if you add 1, and the overflow "goes away", you get $00000000000000000000000000000000_2$.
- So $11111111111111111111111111111111_2$ is -1.

Negative Numbers??

By the time we get to adding the 'k',

- multiplying the code by 31 makes it bigger than 2 billion,
- which is the limit of a 32 bit integer.
- Instead of causing an error it just overflows to the the negative.

The largest positive int is $01111111111111111111111111111111_2$ or

- $2^{31} - 1 = 2147483647$
- Add 1 and you get $10000000000000000000000000000000_2$ or
- $-2^{31} = -2147483648$
- and back up to zero.
- For example, what is $11111111111111111111111111111111_2$?
- It's the sum of two other numbers on this slide.
- And if you add 1, and the overflow "goes away", you get $00000000000000000000000000000000_2$.
- So $11111111111111111111111111111111_2$ is -1.

An int can only hold integers in the range from -2147483648 to 2147483647.

Hash Code

Why not just add up the characters? Why the powers of 31?

Hash Code

Why not just add up the characters? Why the powers of 31?

- If we just added up the characters,

Hash Code

Why not just add up the characters? Why the powers of 31?

- If we just added up the characters,
- then "dear" and "read" and "dare" would be at the same hash code.

Hash Code

Why not just add up the characters? Why the powers of 31?

- If we just added up the characters,
- then "dear" and "read" and "dare" would be at the same hash code.
- But it is worse than that.

Hash Code

Why not just add up the characters? Why the powers of 31?

- If we just added up the characters,
- then "dear" and "read" and "dare" would be at the same hash code.
- But it is worse than that.
- So would "case". Do you see why?

Hash Code

Why not just add up the characters? Why the powers of 31?

- If we just added up the characters,
- then "dear" and "read" and "dare" would be at the same hash code.
- But it is worse than that.
- So would "case". Do you see why?
- 'c' + 's' = 'd' + 'r'

Hash Code

Why not just add up the characters? Why the powers of 31?

- If we just added up the characters,
- then "dear" and "read" and "dare" would be at the same hash code.
- But it is worse than that.
- So would "case". Do you see why?
- 'c' + 's' = 'd' + 'r'
- 'c' + 'a' + 's' + 'e' = 'd' + 'a' + 'r' + 'e'

Hash Code

Why not just add up the characters? Why the powers of 31?

- If we just added up the characters,
- then "dear" and "read" and "dare" would be at the same hash code.
- But it is worse than that.
- So would "case". Do you see why?
- 'c' + 's' = 'd' + 'r'
- 'c' + 'a' + 's' + 'e' = 'd' + 'a' + 'r' + 'e'
- Using powers of 31 makes these all different.

Hash Index

We start with "Milenkovic" and we get a seemingly random 32 bit integer.

Hash Index

We start with "Milenkovic" and we get a seemingly random 32 bit integer.

- We can't use it as an index into an array.

Hash Index

We start with "Milenkovic" and we get a seemingly random 32 bit integer.

- We can't use it as an index into an array.
- It's negative and way too big.

Hash Index

We start with "Milenkovic" and we get a seemingly random 32 bit integer.

- We can't use it as an index into an array.
- It's negative and way too big.
- How can we make it into an index into an array of size m?

Hash Index

We start with "Milenkovic" and we get a seemingly random 32 bit integer.

- We can't use it as an index into an array.
- It's negative and way too big.
- How can we make it into an index into an array of size m?

```
int hashIndex (String name, int m) {
  int code = name.hashCode();
  int index = code % m;
  if (index < 0)
    index += m;
  return index;
}
```

Hash Index

In English:

Hash Index

In English:

- First, mod it by m (divide by m and take the remainder).

Hash Index

In English:

- First, mod it by m (divide by m and take the remainder).
- The remainder will be in the range -(m-1) to m-1.

Hash Index

In English:

- First, mod it by m (divide by m and take the remainder).
- The remainder will be in the range -(m-1) to m-1.
- If it is in the range from -(m-1) to -1, add m to put it into the the range 1 to m-1.

Hash Index

In English:

- First, mod it by m (divide by m and take the remainder).
- The remainder will be in the range -(m-1) to m-1.
- If it is in the range from -(m-1) to -1, add m to put it into the the range 1 to m-1.
- In other words, switch to the positive remainder.

Example:

Hash Index

In English:

- First, mod it by m (divide by m and take the remainder).
- The remainder will be in the range -(m-1) to m-1.
- If it is in the range from -(m-1) to -1, add m to put it into the the range 1 to m-1.
- In other words, switch to the positive remainder.

Example:

- -19 divided by 7 is -2 remainder -5, right?

Hash Index

In English:

- First, mod it by m (divide by m and take the remainder).
- The remainder will be in the range -(m-1) to m-1.
- If it is in the range from -(m-1) to -1, add m to put it into the the range 1 to m-1.
- In other words, switch to the positive remainder.

Example:

- -19 divided by 7 is -2 remainder -5, right?
- But it is also -3 remainder 2. Check -3*7+2=-19.

Collision

So we can store "Milenkovic" in an array of length m

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

- That's called a collision.

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

- That's called a collision.
- Collisions are inevitable because of the birthday paradox.

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

- That's called a collision.
- Collisions are inevitable because of the birthday paradox.
- It takes only $\sqrt{m}$ entries before a collision becomes likely.

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

- That's called a collision.
- Collisions are inevitable because of the birthday paradox.
- It takes only $\sqrt{m}$ entries before a collision becomes likely.
- So if there are more than $\sqrt{365} = 19.1$ people in the room today, it is likely two of use have the same birthday.

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

- That's called a collision.
- Collisions are inevitable because of the birthday paradox.
- It takes only $\sqrt{m}$ entries before a collision becomes likely.
- So if there are more than $\sqrt{365} = 19.1$ people in the room today, it is likely two of use have the same birthday.

There are two ways to deal with collisions:

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

- That's called a collision.
- Collisions are inevitable because of the birthday paradox.
- It takes only $\sqrt{m}$ entries before a collision becomes likely.
- So if there are more than $\sqrt{365} = 19.1$ people in the room today, it is likely two of use have the same birthday.

There are two ways to deal with collisions:

- Separate Chaining

Collision

So we can store "Milenkovic" in an array of length m

- at index hashIndex("Milenkovic", m).
- When we want to find it, we just call hashIndex again.
- Problem solved!
- Or is it?

Of course the problem is that two names could hash to the same index.

- That's called a collision.
- Collisions are inevitable because of the birthday paradox.
- It takes only $\sqrt{m}$ entries before a collision becomes likely.
- So if there are more than $\sqrt{365} = 19.1$ people in the room today, it is likely two of use have the same birthday.

There are two ways to deal with collisions:

- Separate Chaining
- Open Addressing

Separate Chaining

Separate Chaining using linked lists.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as n<m, the expected time is O(1).

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as n<m, the expected time is O(1).

What if it gets full?

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as n<m, the expected time is O(1).

What if it gets full?

- If n=m, it doesn't run out of space but as n gets larger and larger,

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as n<m, the expected time is O(1).

What if it gets full?

- If n=m, it doesn't run out of space but as n gets larger and larger,
- it gets slower and slower.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as n<m, the expected time is O(1).

What if it gets full?

- If n=m, it doesn't run out of space but as n gets larger and larger,
- it gets slower and slower.
- If n=2m, the expected length of a list is n/m = 2.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as n<m, the expected time is O(1).

What if it gets full?

- If n=m, it doesn't run out of space but as n gets larger and larger,
- it gets slower and slower.
- If n=2m, the expected length of a list is n/m = 2.

Instead, just allocate an array twice as big and put every entry in the new hash table.

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as $n < m$, the expected time is O(1).

What if it gets full?

- If n=m, it doesn't run out of space but as n gets larger and larger,
- it gets slower and slower.
- If n=2m, the expected length of a list is n/m = 2.

Instead, just allocate an array twice as big and put every entry in the new hash table.

- Don't just move the lists!!

Separate Chaining

Separate Chaining using linked lists.

- Each entry in the array is the head of a linked list of nodes.
- To find an entry, go to its hash index and search the linked list.

This is O(n) worst case if everyone hashed to the same index,

- but this is extremely unlikely.
- On average, the length of a linked list is n/m.
- So as long as $n<m$, the expected time is O(1).

What if it gets full?

- If n=m, it doesn't run out of space but as n gets larger and larger,
- it gets slower and slower.
- If n=2m, the expected length of a list is n/m = 2.

Instead, just allocate an array twice as big and put every entry in the new hash table.

- Don't just move the lists!!
- Entries in the same list in the first table will be in different lists in the second table.

Open Addressing

Open addressing stores colliding entries at another location in the array.

Open Addressing

Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space

Open Addressing

Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space
- but which is over-subscribed.

Open Addressing

Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space
- but which is over-subscribed.
- The spaces are numbered and everyone is assigned a number (their hash index).

Open Addressing

Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space
- but which is over-subscribed.
- The spaces are numbered and everyone is assigned a number (their hash index).
- No more than m/2 show up on any particular day,

Open Addressing

Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space
- but which is over-subscribed.
- The spaces are numbered and everyone is assigned a number (their hash index).
- No more than m/2 show up on any particular day,
- but collisions are still certain.

Open Addressing

Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space
- but which is over-subscribed.
- The spaces are numbered and everyone is assigned a number (their hash index).
- No more than m/2 show up on any particular day,
- but collisions are still certain.

So if you arrive and your space is filled,

Open Addressing

Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space
- but which is over-subscribed.
- The spaces are numbered and everyone is assigned a number (their hash index).
- No more than m/2 show up on any particular day,
- but collisions are still certain.

So if you arrive and your space is filled,

- you are supposed to keep driving until you find an empty one and park there.

Open Addressing

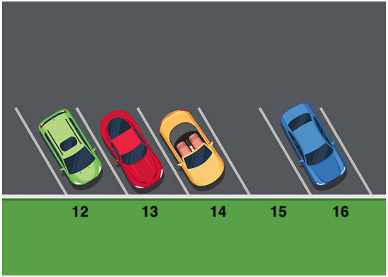
Open addressing stores colliding entries at another location in the array.

- Sort of like a parking lot with assigned space
- but which is over-subscribed.
- The spaces are numbered and everyone is assigned a number (their hash index).
- No more than m/2 show up on any particular day,
- but collisions are still certain.

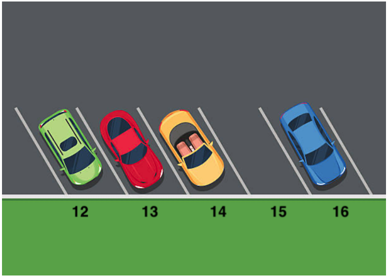
So if you arrive and your space is filled,

- you are supposed to keep driving until you find an empty one and park there.
- If you reach the end of the lot, you go back to the beginning.

Add

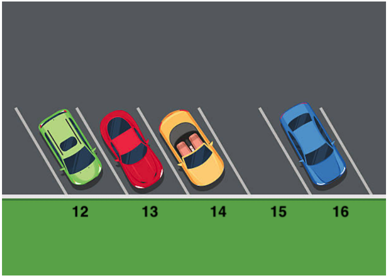


Add



Here is how to add.

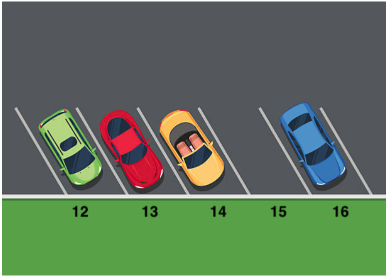
Add



Here is how to add.

- Suppose my assigned number is 12.

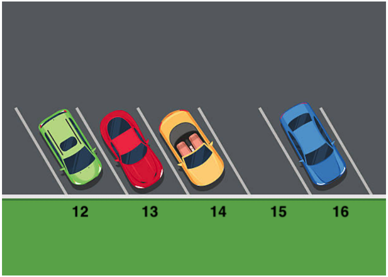
Add



Here is how to add.

- Suppose my assigned number is 12.
- But 12 is full.

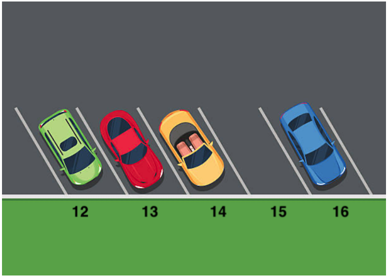
Add



Here is how to add.

- Suppose my assigned number is 12.
- But 12 is full.
- So are 13 and 14.

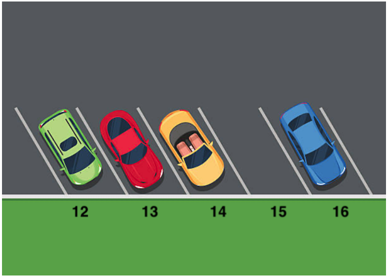
Add



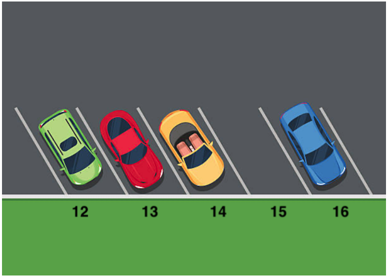
Here is how to add.

- Suppose my assigned number is 12.
- But 12 is full.
- So are 13 and 14.
- So I park in 15.

Find

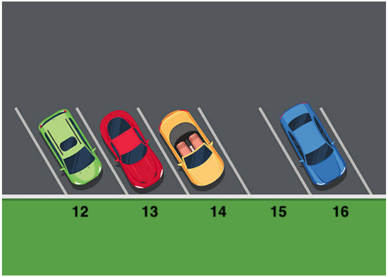


Find



Here is how to find Victor.

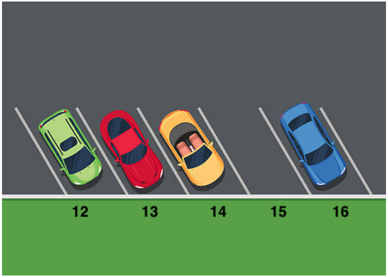
Find



Here is how to find Victor.

- You know my assigned number (hash index of Victor) is 12.

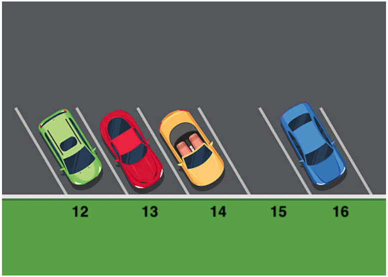
Find



Here is how to find Victor.

- You know my assigned number (hash index of Victor) is 12.
- But I am not in 12 (assume everyone has their name on their license plate).

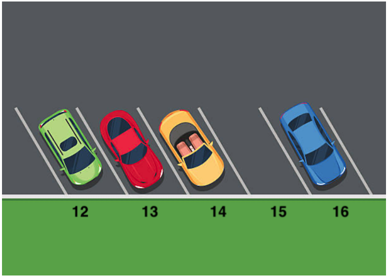
Find



Here is how to find Victor.

- You know my assigned number (hash index of Victor) is 12.
- But I am not in 12 (assume everyone has their name on their license plate).
- So you keep looking at 13, 14, 15.

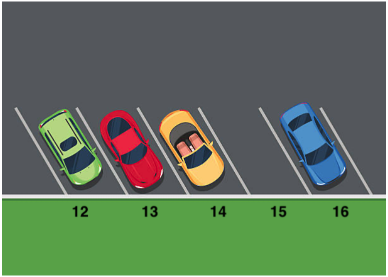
Find



Here is how to find Victor.

- You know my assigned number (hash index of Victor) is 12.
- But I am not in 12 (assume everyone has their name on their license plate).
- So you keep looking at 13, 14, 15.
- If I am there, you find me in 15.

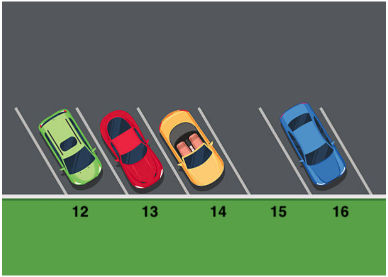
Find



Here is how to find Victor.

- You know my assigned number (hash index of Victor) is 12.
- But I am not in 12 (assume everyone has their name on their license plate).
- So you keep looking at 13, 14, 15.
- If I am there, you find me in 15.
- If I am not there (stayed home) you get to an empty space and stop.

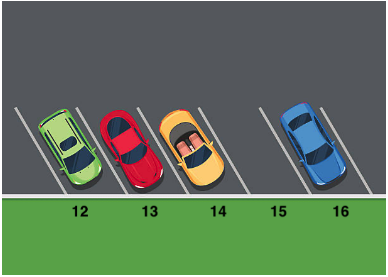
Find



Here is how to find Victor.

- You know my assigned number (hash index of Victor) is 12.
- But I am not in 12 (assume everyone has their name on their license plate).
- So you keep looking at 13, 14, 15.
- If I am there, you find me in 15.
- If I am not there (stayed home) you get to an empty space and stop.
- You know if you see an empty space that I am not there because I am supposed to park in the first empty space I see.

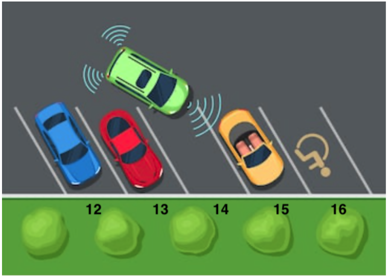
Find



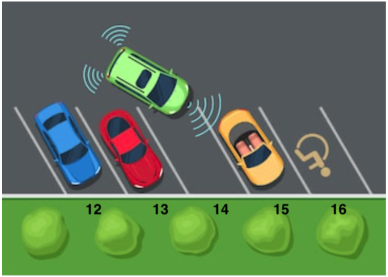
Here is how to find Victor.

- You know my assigned number (hash index of Victor) is 12.
- But I am not in 12 (assume everyone has their name on their license plate).
- So you keep looking at 13, 14, 15.
- If I am there, you find me in 15.
- If I am not there (stayed home) you get to an empty space and stop.
- You know if you see an empty space that I am not there because I am supposed to park in the first empty space I see.
- So if I am there, I must be before the first empty space you see.

Problem with Find

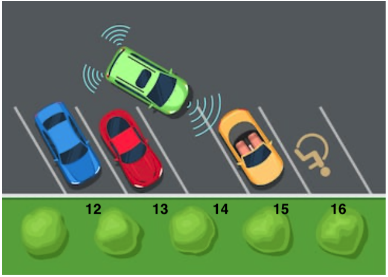


Problem with Find



There is a problem with find.

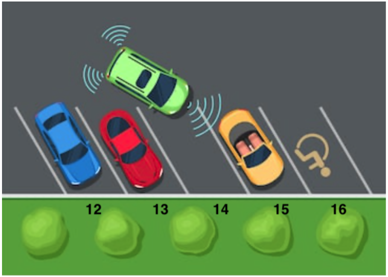
Problem with Find



There is a problem with find.

- What if someone was in a spot that I skipped, in this case 14.

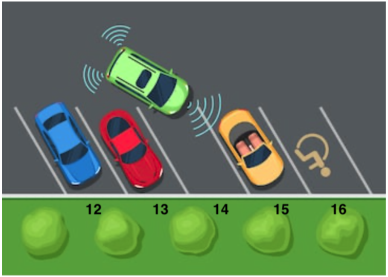
Problem with Find



There is a problem with find.

- What if someone was in a spot that I skipped, in this case 14.
- And left before you came looking for me.

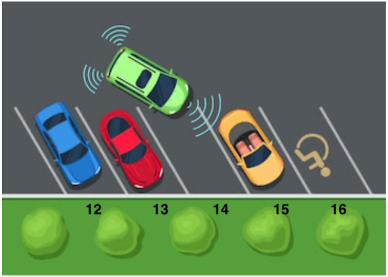
Problem with Find



There is a problem with find.

- What if someone was in a spot that I skipped, in this case 14.
- And left before you came looking for me.
- You will see an empty space and stop looking for me.

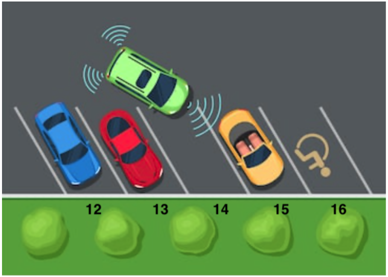
Problem with Find



There is a problem with find.

- What if someone was in a spot that I skipped, in this case 14.
- And left before you came looking for me.
- You will see an empty space and stop looking for me.
- Even though I am in the next space (15).

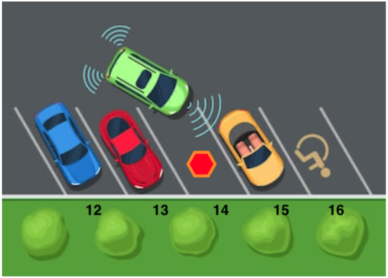
Problem with Find



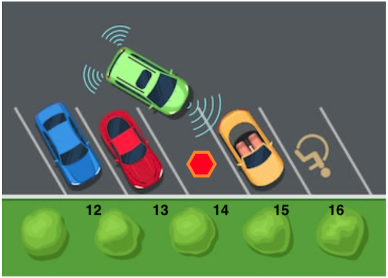
There is a problem with find.

- What if someone was in a spot that I skipped, in this case 14.
- And left before you came looking for me.
- You will see an empty space and stop looking for me.
- Even though I am in the next space (15).
- How do we fix this?

Marking DELETED spaces

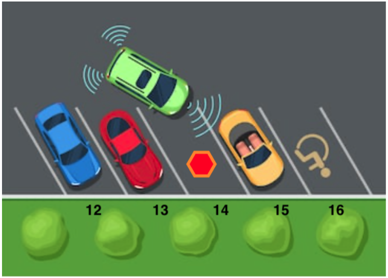


Marking DELETED spaces



The fix is to mark DELETED spaces.

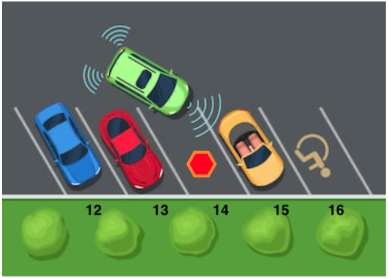
Marking DELETED spaces



The fix is to mark DELETED spaces.

- If you leave, put a traffic cone in the space.

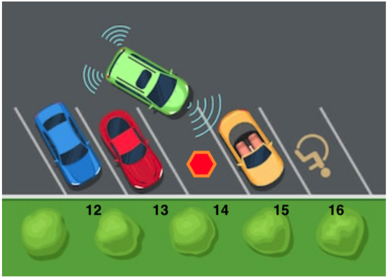
Marking DELETED spaces



The fix is to mark DELETED spaces.

- If you leave, put a traffic cone in the space.
- If you are trying to find me and see a traffic cone, keep looking.

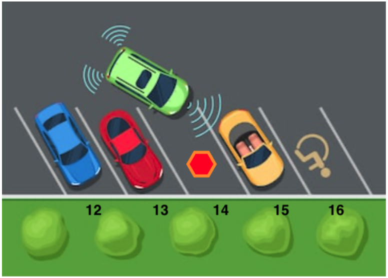
Marking DELETED spaces



The fix is to mark DELETED spaces.

- If you leave, put a traffic cone in the space.
- If you are trying to find me and see a traffic cone, keep looking.
- DO NOT treat it as an empty space.

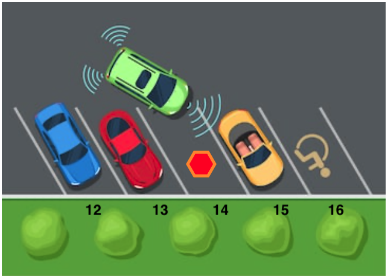
Marking DELETED spaces



The fix is to mark DELETED spaces.

- If you leave, put a traffic cone in the space.
- If you are trying to find me and see a traffic cone, keep looking.
- DO NOT treat it as an empty space.
- However, when I am parking, I DO treat it as an empty space.

Marking DELETED spaces



The fix is to mark DELETED spaces.

- If you leave, put a traffic cone in the space.
- If you are trying to find me and see a traffic cone, keep looking.
- DO NOT treat it as an empty space.
- However, when I am parking, I DO treat it as an empty space.
- I park in the first space that is empty or has a traffic cone.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.
- The new array length should be 4 times the number (n) of cars.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.
- The new array length should be 4 times the number (n) of cars.
- You have to take everyone out of the old array and park them in the new array.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.
- The new array length should be 4 times the number (n) of cars.
- You have to take everyone out of the old array and park them in the new array.
- Even if the new array is the same length, you can't just move cars to the same index.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.
- The new array length should be 4 times the number (n) of cars.
- You have to take everyone out of the old array and park them in the new array.
- Even if the new array is the same length, you can't just move cars to the same index.

And you can't let the lot get too full.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.
- The new array length should be 4 times the number (n) of cars.
- You have to take everyone out of the old array and park them in the new array.
- Even if the new array is the same length, you can't just move cars to the same index.

And you can't let the lot get too full.

- If there are m-1 cars, what's the expected time to add?

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.
- The new array length should be 4 times the number (n) of cars.
- You have to take everyone out of the old array and park them in the new array.
- Even if the new array is the same length, you can't just move cars to the same index.

And you can't let the lot get too full.

- If there are m-1 cars, what's the expected time to add?
- O(m) right? So this is not like separate chaining.

Open Addressing rehashing rule

Eventually, the lot might get full of traffic cones,

- which will make it O(m) to find anyone.
- because you can't stop searching when you see a traffic cone.
- So even if there is always at most m/2 cars parked (m is array length),
- you need to rehash the whole thing when the number of cars PLUS traffic cones is over m/2.
- The new array length should be 4 times the number (n) of cars.
- You have to take everyone out of the old array and park them in the new array.
- Even if the new array is the same length, you can't just move cars to the same index.

And you can't let the lot get too full.

- If there are m-1 cars, what's the expected time to add?
- O(m) right? So this is not like separate chaining.
- You have to keep n <= m/2.

Summary

Summary

The hash code of an item is an int,

Summary

The hash code of an item is an int,

- possibly negative,

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.
- add, find, remove are n/m expected time.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.
- add, find, remove are n/m expected time.
- Need to reallocate and rehash when n > m.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.
- add, find, remove are n/m expected time.
- Need to reallocate and rehash when n > m.

Open Addressing

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.
- add, find, remove are n/m expected time.
- Need to reallocate and rehash when n > m.

Open Addressing

- More suitable for real-time programming or operating systems.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.
- add, find, remove are n/m expected time.
- Need to reallocate and rehash when n > m.

Open Addressing

- More suitable for real-time programming or operating systems.
- Item cycles through the array until it finds an empty spot.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.
- add, find, remove are n/m expected time.
- Need to reallocate and rehash when n > m.

Open Addressing

- More suitable for real-time programming or operating systems.
- Item cycles through the array until it finds an empty spot.
- Removed items need to leave a “traffic cone”.

Summary

The hash code of an item is an int,

- possibly negative,
- usually unique (for up to $\sqrt{2^{32}} = 2^{16} = 65536$ entries).
- Don't use ^ or pow when calculating the hash code of a String.

The hash index of an item is an int

- in the range 0 to m-1.
- Mod the hash code by m,
- add m if negative.
- Not unique ($\sqrt{m}$ is a lot less than m)!

A hash table is an array of length m.

- Item with hash index i is stored at index i,
- unless there is an item there already.

Separate Chaining

- stores a linked list at each index.
- add, find, remove are n/m expected time.
- Need to reallocate and rehash when n > m.

Open Addressing

- More suitable for real-time programming or operating systems.
- Item cycles through the array until it finds an empty spot.
- Removed items need to leave a "traffic cone".
- Needs to rehash and possibly reallocate when less than m/2 spaces are empty.

Extra Credit

What does expected time mean for hash tables?

Extra Credit

What does expected time mean for hash tables?

- Unlike SkipMap, there *is* a bad input for a hash table.

Extra Credit

What does expected time mean for hash tables?

- Unlike SkipMap, there *is* a bad input for a hash table.
- Unlike BST, this input is rather hard to find.

Extra Credit

What does expected time mean for hash tables?

- Unlike SkipMap, there *is* a bad input for a hash table.
- Unlike BST, this input is rather hard to find.
- This is the basis for BitCoin!

Extra Credit

What does expected time mean for hash tables?

- Unlike SkipMap, there *is* a bad input for a hash table.
- Unlike BST, this input is rather hard to find.
- This is the basis for BitCoin!
- You will learn a technique *Universal Hashing* which makes the expected time $\mathrm{O}(n)$ for all inputs.

Extra credit exercise:

Extra Credit

What does expected time mean for hash tables?

- Unlike SkipMap, there *is* a bad input for a hash table.
- Unlike BST, this input is rather hard to find.
- This is the basis for BitCoin!
- You will learn a technique *Universal Hashing* which makes the expected time $\mathrm{O}(n)$ for all inputs.

Extra credit exercise:

- Find 100 words that hash to the same index in a table of length 100.

Extra Credit

What does expected time mean for hash tables?

- Unlike SkipMap, there *is* a bad input for a hash table.
- Unlike BST, this input is rather hard to find.
- This is the basis for BitCoin!
- You will learn a technique *Universal Hashing* which makes the expected time $\mathrm{O}(n)$ for all inputs.

Extra credit exercise:

- Find 100 words that hash to the same index in a table of length 100.
- Email your solution to vjm@miami.edu by Tuesday morning.

Extra Credit

What does expected time mean for hash tables?

- Unlike SkipMap, there *is* a bad input for a hash table.
- Unlike BST, this input is rather hard to find.
- This is the basis for BitCoin!
- You will learn a technique *Universal Hashing* which makes the expected time $\mathrm{O}(n)$ for all inputs.

Extra credit exercise:

- Find 100 words that hash to the same index in a table of length 100.
- Email your solution to vjm@miami.edu by Tuesday morning.
- What should it be worth?