CSCI 241

Lecture 17

Hash Functions, Hash Tables, Hash Sets, Hashtags

CSCI 241

Lecture 17

Hash Functions, Hash Tables, Hash Sets~~, Hashtags~~

Happenings

Monday, 11/5

- Tech Talk: Vitech HighJump – 5 pm in CF 125

Tuesday, 11/6

- Peer Lecture Series: C Language Workshop – 4 pm in CF 420
- ACM Presents: Susheel Gopalan from Microsoft– 5 pm in CF 316

Wednesday, 11/7

- CS Alumni Happy Hour – 3:30 pm at Sam's Tavern in Seattle
- New Club: IEEE Student Chapter Meeting – 5:30 pm in ET 321

Thursday, 11/8

- Group Advising to Declare the CS Premajor – 3 pm in CF 420
- STEM Mix it Up – 5 pm in the MAC Gym

Announcements

- A1 is almost done - grades should be out tonight
- Midterm - aiming for middle of this week

Goals

- Know the purpose, definition, and properties **hash functions**.
- Know how to use a hash function to implement a hash table.
- Know how to use modular arithmetic to construct a basic hash function on integers.
- Know how to implement a HashSet using chaining for collision resolution.
- Know the definition of load factor in hash table.

Reminder: The **Set** ADT

- A **Set** maintains a collection of **unique** things.
- Java has this ADT built in as an interface:

  `java.util.Set`
- Some methods from `java.util.Set:`
  - `boolean` **`add`**`(Object ob)`
  - `boolean` **`contains`**`(Object ob)`
  - `boolean` **`remove`**`(Object ob)`

Reminder: The **Set** ADT

- A **Set** maintains a collection of **unique** things.
- Java has this ADT built in as an interface:

  `java.util.Set<T>`
- Some methods from `java.util.Set:`
  - `boolean` **`add`**`(T ob)`
  - `boolean` **`contains`**`(T ob)`
  - `boolean` **`remove`**`(T ob)`

Hashing: Motivation

- Consider implementations of the Set ADT:

	`add`	`contains`	`remove`
Unsorted Array or Linked List	O(1)	O(n)	O(n)
Sorted Linked List	O(n)	O(n)	O(n)
Sorted Array	O(n)	O(log n)	O(n)
AVL Tree	O(log n)	O(log n)	O(log n)
Magical Array	**O(1)**	**O(1)**	**O(1)**

How would you implement a Set that can only contain the digits 0..10?

Remember Radix Sort?

[07, 19, 61, 11, 14, 54, 01, 08]

0	1	2	3	4	5	6	7	8	9

Bukkits on 1's place

insert(4)

boolean[] A:

0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

insert(4)

boolean[] A:

0	F
1	F
2	F
3	F
4	T
5	F
6	F
7	F
8	F
9	F

insert(4)

insert(7)

boolean[] A:

0	F
1	F
2	F
3	F
4	T
5	F
6	F
7	F
8	F
9	F

insert(4)

insert(7)

boolean[] A:

0	F
1	F
2	F
3	F
4	T
5	F
6	F
7	T
8	F
9	F

insert(4)

insert(7)

insert(4)

boolean[] A:

0	F
1	F
2	F
3	F
4	T
5	F
6	F
7	T
8	F
9	F

insert(4)

insert(7)

insert(4)

boolean[] A:

0	F
1	F
2	F
3	F
4	T
5	F
6	F
7	T
8	F
9	F

```
insert(i):
 A[i] = true

contains(i):
 return A[i]

remove(i):
 A[i] = false
```

Direct-Address Table

insert(4)

insert(7)

insert(4)

```
insert(i):
 A[i] = true

contains(i):
 return A[i]

remove(i):
 A[i] = false
```

boolean[] A:

0	F
1	F
2	F
3	F
4	T
5	F
6	F
7	T
8	F
9	F

Direct-Address Table

- This was easy because the (ADT) Set contents came from a small, fixed space of possible values (0..10).

- Hash functions are the magic that lets us do this with *any* space of values.

Reminder: The Modulus Operator

a % b gives the remainder when dividing a by b:

12 % 8 => 4

24 % 10 => 4

4 % 10 => 4

28 % 14 => 0

Hash Tables with Integers

How can we determine an index for **any** integer in a **fixed-sized** array?

- Modular arithmetic:
  store value k in the k % 10 bucket
  - (14 % 10) => 4
  - (10 % 10) => 0
  - (1 % 10) => 1
  - (11 % 10) => 1

uh oh…

boolean[] A:

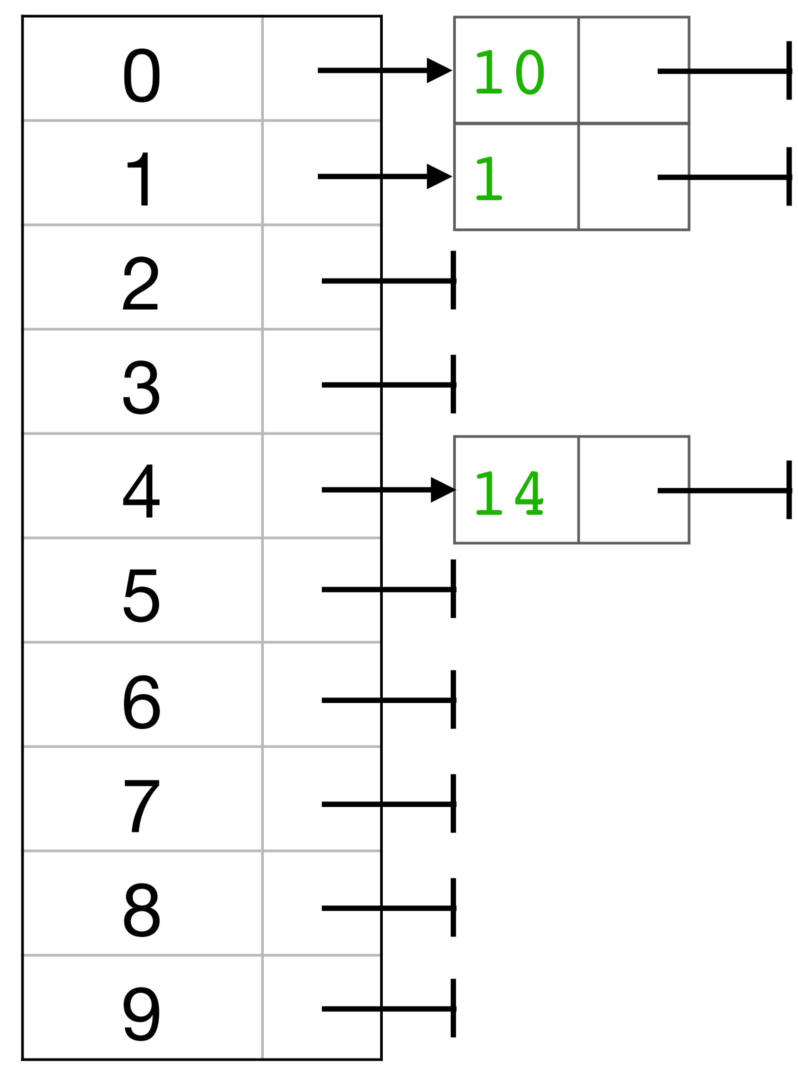
0	T
1	T
2	F
3	F
4	T
5	F
6	F
7	F
8	F
9	F

Hash Tables with Integers: Collisions

- Modular arithmetic:
  store value k in the k % 10 bucket

  - (14 % 10) => 4
  - (10 % 10) => 0
  - (1 % 10) => 1
  - (11 % 10) => 1

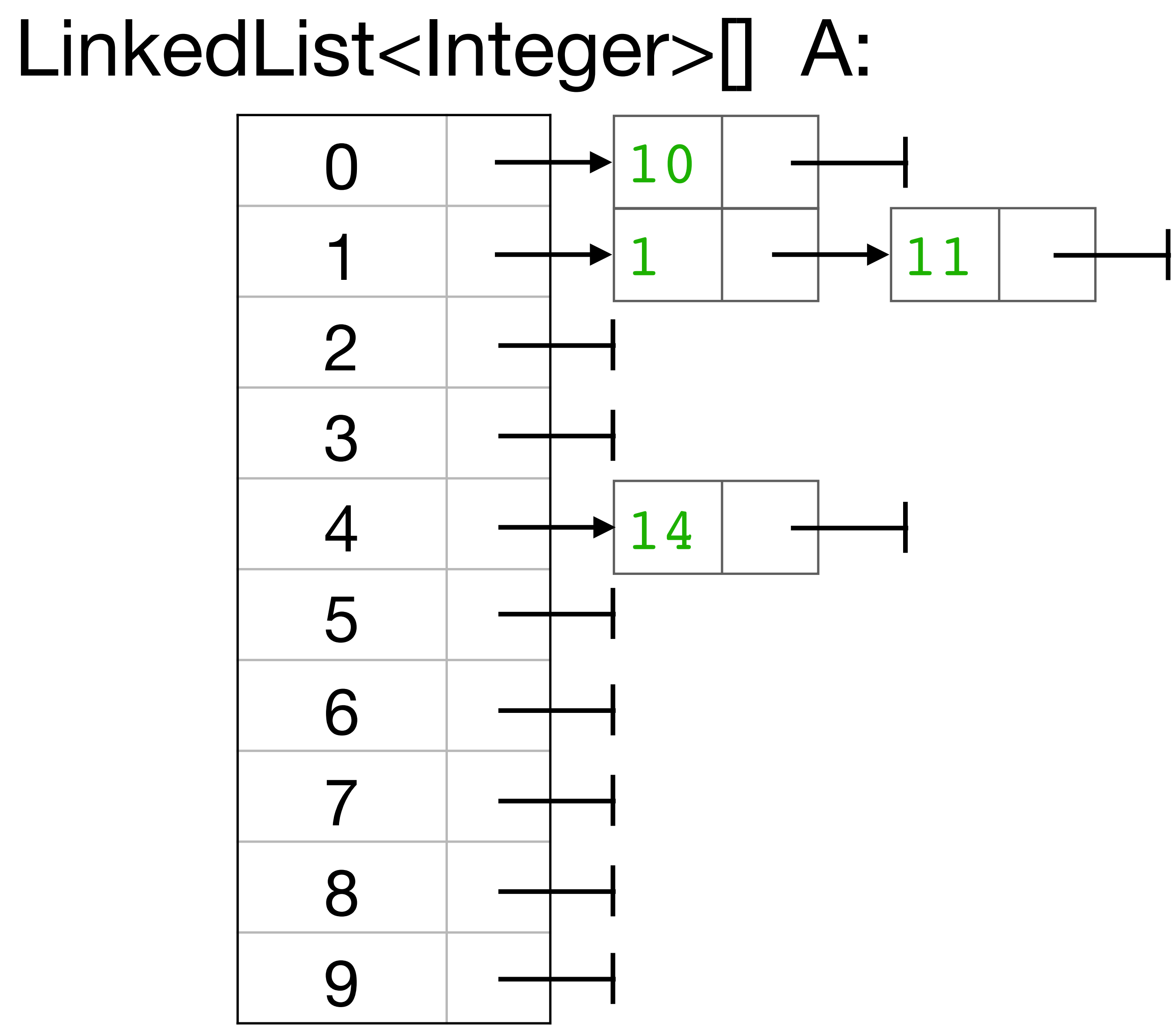
LinkedList<Integer>[] A:



Hash Tables with Integers: Collisions

- Modular arithmetic:
  store value k in the k % 10 bucket

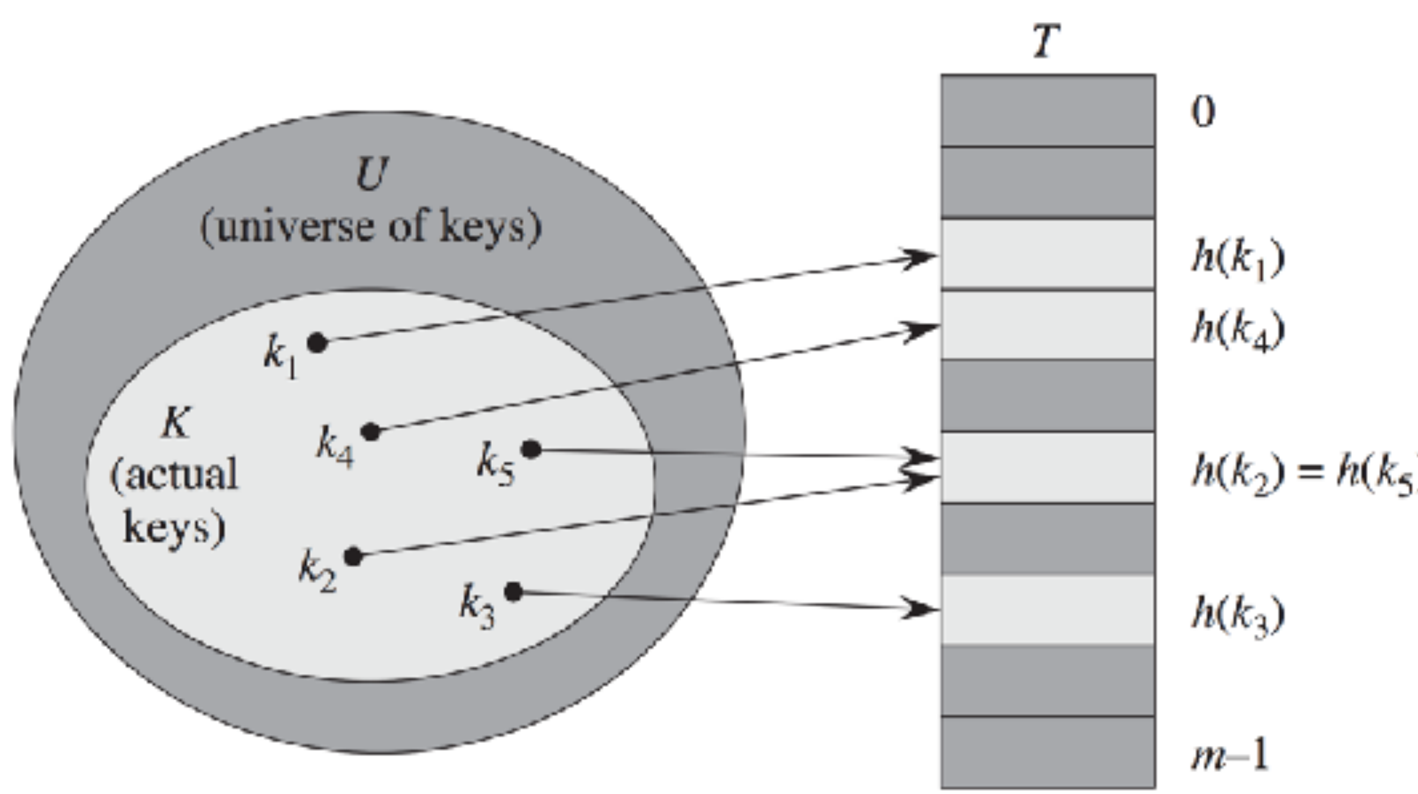
  - (14 % 10) => 4
  - (10 % 10) => 0
  - (1 % 10) => 1
  - (11 % 10) => 1



Hash Functions: Formally

A **Hash Function** takes an object (String, integer, Person, …) and returns a non-negative integer index into a **fixed-size** array.

h: U —> { 0, 1, 2, … m-1}



Hash Functions: Properties

If *h* is a hash function, then:

- h is **deterministic** and **fast to compute**:
  for some fixed key k, *h*(k) always returns the same value and is efficiently computable (usually O(1))

- Equal objects hash to equal values:
  h(i) == *h*(j) if `i.equals(j)`

- Collisions are possible:
  If `!i.equals(j)` it **is possible** that *h*(i) == *h*(j)

  or: h(i) == h(j) does not imply `i.equals(j)`

Hash Functions on General Objects

- Not just limited to integers.
- java.util.Object has a hashCode method:
  - any object can be hashed!
- Hash functions on non-integers:
  - possible because all things are represented in binary and can be reinterpreted as numbers
  - more on this later

HashSet<T>

```
/** insert value into the set. return false if the
value was already in the set, true otherwise */
boolean insert(T value) {
  int h = hash(value)
  search the list at A[h] for value
  if found:
     return false
  else:
     insert value into A[h] and return true
}

/** return true if value is in the set,
  * false otherwise */
boolean contains(T value) { … }

/** insert value into the set. return true if the
  * value was in the set, false otherwise */
boolean remove(T value) { … }
```

HashSet<T>: What's the runtime?

```
/** insert value into the set. return false if the
value was already in the set, true otherwise */
boolean insert(T value) {
  int h = hash(value)
  search the list at A[h] for value
  if found:
     return false
  else:
     insert value into A[h] and return true
}
```

HashSet<T>: What's the runtime?

```
/** insert value into the set. return false if the
value was already in the set, true otherwise */
boolean insert(T value) {
  int h = hash(value)                          O(1)
  search the list at A[h] for value            O(length of list)
  if found:
     return false                              O(1)
  else:
     insert value into A[h] and return true    O(1)
}
```

HashSet<T>: What’s the runtime?

All operations require searching a single bucket and doing some other stuff that runs in O(1).

```
/** return true if value is in the set,
  * false otherwise */
boolean contains(T value) { … }

/** remove value from the set. return true if the
  * value was in the set, false otherwise */
boolean remove(T value) { … }
```

Hash Functions: Desirable Properties

We would ***like*** our hash functions distribute values evenly among buckets.

It's hard to guarantee this without knowing keys ahead of time, but usually easy in practice using heuristics.

Hash Functions: Desirable Properties

A universally terrible hash function: $h(k) = 0$

Hash function quality often depends on the keys.
e.g., if keys are WWU CSCI course numbers:

- h(k) = k % 100 (1's place)
  - bad because many collisions (141, 241, 301, …)

- h(k) = k / 100 (100's place)
  - bad because this will only use buckets 0..6

One weird tip: make the table size prime so divisibility patterns in keys don't result in patterns in hash buckets.

Hash Tables: Load Factor

How full is your hash table?

Load factor λ = # entries in set / size of the array

With a perfectly-behaved hash function, average bucket size is λ, so average-case runtime is $O(\lambda)$.