Homework 3: Dictionaries

The Most General ADT

Preliminaries

- **Two Data Structures in this homework:** Association List & HashTables
- **Association List**
  - Our implementation of Association List is going to assume that someone could insert duplicating keys!
    - If that's the case, we need to reassign the key's value if its already in the Association List
- **HashTable**
  - Our implementation is going to use separate chaining
    - We're going to hash our keys so we know where to insert them into our HashTable
      - If two things get hashed to the same index, then we build a list!
    - Therefore, each "bucket" is its own list!

More on Assoication List and HashTable specifics in later slides

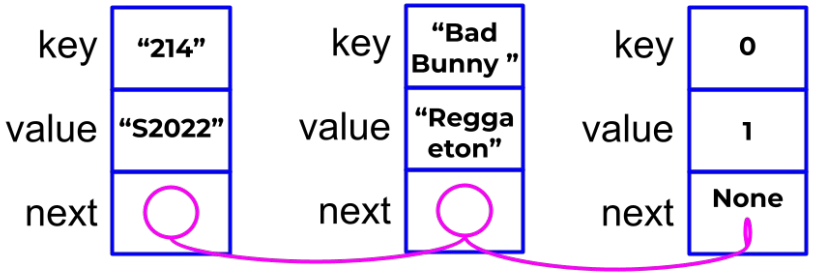
HW3 is when things start to ramp up but you still got this!

Goals for this assignment

- Association List and HashTable Class
  - May seem like they're two completely different things but they are closely related
    - Possible to abstract most of Hash Table operations with Association List
- Avoiding reusing code
  - In this assignment, you may find yourself repeating the same logic/code blocks
    - Make this into a helper function!
    - Otherwise, it's bad practice & much harder to debug if you run into errors
    - Makes you code look "cleaner" and **much** more organized
- Testing "unpredictable" code
  - Some mistakes/errors may not be as obvious as you're testing, but there's ways to work around it

Content Review: Association List

- Association List are essentially Linked-List but with extra steps
  - You've already implemented Linked-List in HW2, a lot of that logic applies here!
- **Insertion:** go through the entire Linked-List and search for duplicates (for HW3), ***then*** insert -> O(1)
  - If you know duplicates aren't a worry, insert at the front
- Time complexities are O(n) for both lookup and insertion. Not bad but not good, can be better (see HashTables)
  - But maybe they make sense for your use-case (i.e smaller dictionaries)
    - Trade-offs to each data structure! Pick what's best for your problem
    - Also easier to implement!



The AssociationList Class Methods

- **Given:** _head, _length & the cons struct
  - Given to you for a reason, make use of them!
  - If you find yourself struggling with making the Linked-List, look back to HW2
    - Ask course staff for assistance ASAP!
- **My Adivce:**
  - This is the first assignment where I **strongly recommend** that you write helper functions. -> better practice + cleaner code
    - Lots of repeating logic/code
  - A lot of this Class is going to involve iterating over a Linked-List (we'll talk about this in next slide) and doing something at some condition

Your Job:
def len(self) -> nat?
def mem?(self, key: K) -> bool?
def get(self, key: K) -> V
def put(self, key: K, value: V) ->NoneC
def del(self, key: K) -> NoneC

Bolded def -> going to talk about these

mem?, get, put, and del

- All of these involve iterating over the Linked-List so lets go over the logic for that (will see this again in later homework assignments!)

Iterating over a Linked-List

1. Assign current head to sometype of variable (lets say this var is x)
2. Iterate while x is not at the end of the Linked-List
3. If {some condition}, **return** {your desired output}
4. Else, reassign x to the next element in the Linked-List (go back to step 3)

My hint for step 3: *you just want the data field of a key:value pair*

mem?(self, key: K) -> bool?

- Return True/False if when iterating over the Linked-List you find your desired key

get(self, key: K) -> V

- **Special Case:** If key is nonexistent, error
- Return value from a desire key:value pair

put(self, key: K, value: V) ->NoneC

- **Special Case:** If Key is already in the Linked-List, update it's value
- Otherwise, Add the key:value pair to the Linked-List
  - Cons!

del(self, key: K) -> NoneC

- If the head is the key you want to delete, easy, just reassign the head!
- If that's not care, iterate over the Linked-List until you reach your desire key:value pair.
  - Recall pop/dequeue in HW2

Expected Behavior for Assoication List

let a = AssociationList()

a.mem?(“ethan”) -> False

a.get(“214”) -> error

a.put(“214”, “S2022”) -> “214”: “S2022” key:value pair is added to the Linked-List

a.put(10, 100) -> 10:100 Key:value pair is added to the Linked-List

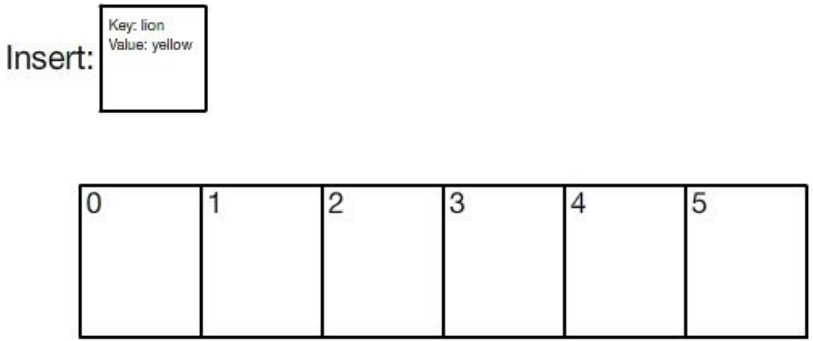
a.put(10,1) -> 10:100 Key:Value is updated to 10:1

a.get(10) -> 1

a.del(“214”) -> “214”: “S2022” key:value pair is removed from the Linked-List

Content Review: HashTables via separate chaining

- We can think of our HashTable as an array where each “bucket” (index) is going to be its own list
  - Each bucket is going to correlate to a specific index
  - If two keys get hashed to the same bucket, we add it to the list!
- How do we know what should go where?
  - Hashing function
    - **In DSSL2 -> make_sbox_hash**
    - **In HW3 -> _hash**
      - Can call _hash on a key and use modulo (%) with the HashTables length
- Expected O(1) insertion & lookup but could need linear search to find a specific key -> O(n)
- Normally, HashTables are made with dynamic arrays (grow in size, and rehashing) to maintain a desired **load factor**
  - We maintain a load factor to try to keep O(1) time complexity



Don’t worry about this in HW3 but this means that when you intalize your HashTable array, it needs to be right length!!

GIF Sourced from https://brilliant.org/wiki/hash-tables/

The HashTable Class

- **Given:** _hash, _size, _data
  - Use these! Especially _hash, going to needed when you need to figure out where each element you put goes (see previous slide)
- **Testing**
  - Testing can be difficult because the hash function we use is **unpredictable**
    - Make use of first_char_hasher, but don't use them in your actual HashTable
      - It's predictable -> easier to debug, but that predictability is bad for our use-case
  - Your HashTable needs to handle **collisions** correctly! Make sure you're forcing collisions to occur in your test
- **My Advice:**
  - Don't make HashTables harder than they already are. It's possible to **reuse** a lot of your code from before with a couple of more nuances added to it.
  - Again, if you find yourself repeating the same logic/code, make this into a **helper function!**
  - Probably going to write a couple of helper functions for this -> easier to debug

Your Job: Same as association list (see slide 5)

mem?, get, put, and del

- **My Adivce:** Each of these is going to first involve getting the hash code of the desired key
  - If there is a collision at this hash code (index), then you should iterate through the Linked-List at this index (see slide 6 for the general logic)
  - Lots of similar logic from Association List
  - Possible to abstract all of these functions

Getting the Hash Code in HW3

1. Call _hash on the key
2. Use modulo, % with the HashTables length

HashCode won't be the same each time, so be weary when testing! (see slide 8)

mem?(self, key: K) -> bool?

- Get the hash code of this key
- If this index is empty -> False
  - But maybe this index is a LL, need to iterate to see

get(self, key: K) -> V

- **Special Case:** If key is nonexistent -> error
- Get the hash code of this key
- Return the value of this key
  - Again, may need to iterate through a LL

put(self, key: K, value: V) ->NoneC

- **Special Case:** If Key is already in the HashTable, update it's value
- Get the hash code of this key
- If collision, add this element to the LL

del(self, key: K) -> NoneC

- Get the hash code of this key
- **My advice (from my experience):** Check if the key is a member of the dictionary
  - If it is, delete the key:value pair from this index/LL
    - See slide 6 for deleting a key:value pair from a LL

LL = Linked-List

Expected Behavior for HashTables

let ht = HashTable()

ht.mem?(“ethan”) -> False

ht.get(“214”) -> error

ht.put(“214”, “S2022”) -> “214”: “S2022” key:value pair is added to the HashTable at index 2

ht.put(10, 100) -> “214”: “S2022” key:value pair is added to the HashTable at index 2, “214”:”S2022” is already there, add this and make a Linked-List

ht.put(10,1) -> 10:100 Key:Value is found at index 2’s LL, updated to 10:1 at index 2

ht.get(10) -> 1

ht.del(“214”) -> “214”: “S2022” key:value pair is removed from the Linked-List at index 2

That’s all for HW3!