Hash Table Lab

"I've been doing a lot of learning from mistakes, first and foremost, and building off that."

After finishing each part of the lab, copy your entire project and work on the copy for the next part!

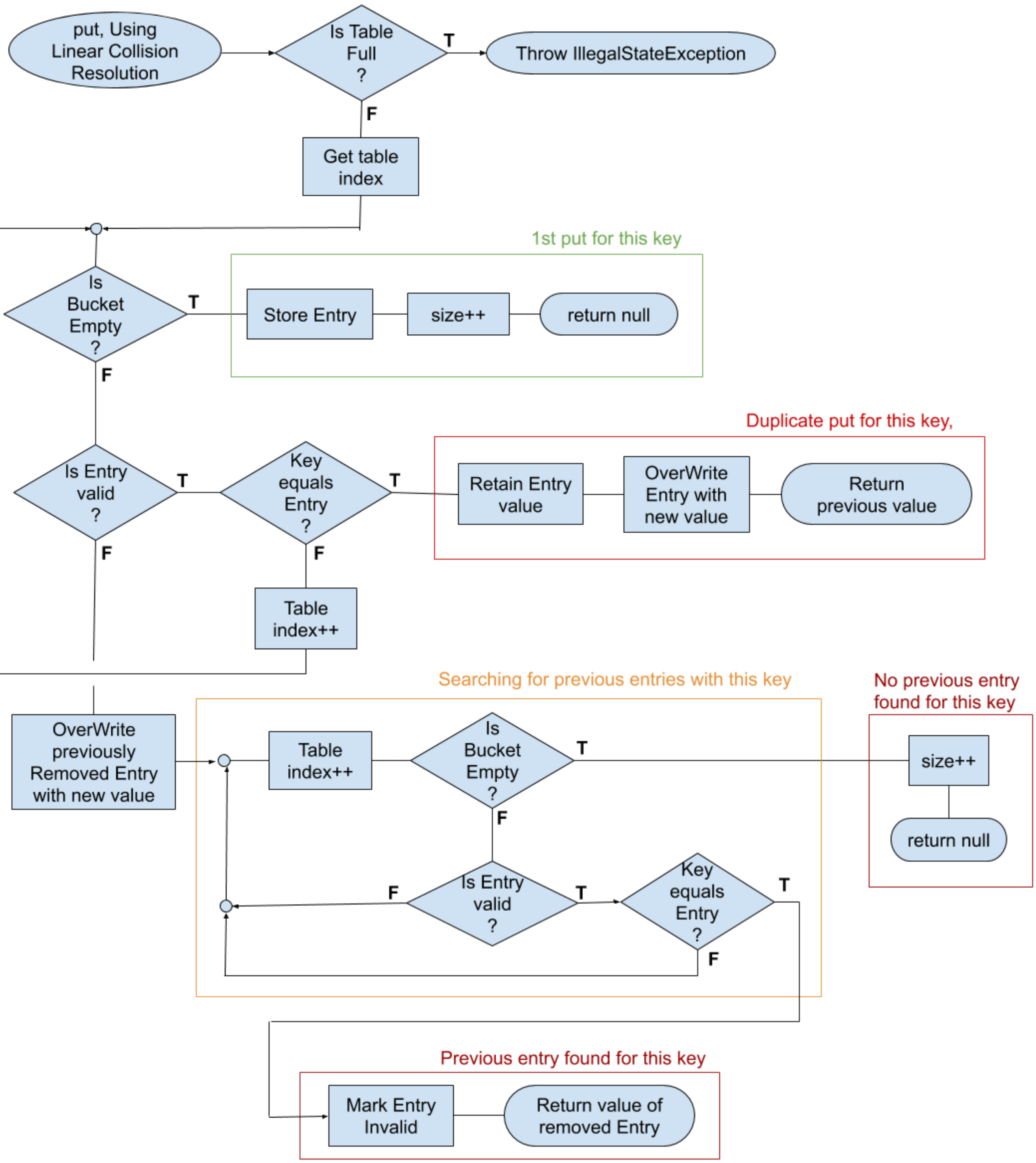
Part 2: Modify the *HashTable* class, implementing linear probing to handle collisions.

- Add a *size* field to *HashTable*.
  - Initialize to 0
  - Increment for *put*s and decrement for *remove*s.
- Add a *removed* field to *Entry* to indicate an unused bucket.
- Implement the *remove* method:
  - If the *Entry* exists, leave the *Entry* in place & mark the *removed* field as *true*.
    - Be sure to use *equals* (on the key object) to verify you've found the correct object.
  - Add linear probing when collisions occur:
    - Search until object is found or empty bucket encountered
    - Skip *removed* objects.
  - Return the previously stored value if the key is valid; otherwise, return *null* if the key was not found.
  - Decrement *size* if the *remove* succeeded.
- Modify the *get* method:
  - If the *Entry* exists, return the *value*.
    - Use *equals* (on the key object) to verify you've found the correct object.
  - If the *key* hashes to a different value, a collision occurred:
    - Use linear probing to find the object
    - Search until the object is found (verify with *equals*) or an empty bucket is encountered
    - Skip *removed* objects.
  - Return the stored value if the *key* is valid; otherwise, return *null* if the key was not found.

Continues on next page…

- Modify the *put* method (**see next page** for flowchart):
    - Check *size* to be sure space is available.
    - If the hashed location is empty, store the value, increment *size,* & return *null*
    - If a collision occurs, the key may have been used before:
        - Is the object already stored in the table?
            - Use *equals* (on the key object) to verify
            - If duplicate, overwrite the location & return the previously stored value (don't increment *size*)
        - Not at the hashed location? Use linear probing to find an empty table location:
            - Check for duplicate keys at each location
            - If an empty location is encountered:
                - Save the new object
                - Return *null*
                - Increment *size*
    - If, while searching for an unused table location, a *removed* location is encountered:
        - Save the new object in place of the *removed* Entry
        - Continue searching for a duplicate key:
            - Until an empty bucket is encountered:
                - Increment *size*
                - Return *null*
            - Or a duplicate is found:
                - Mark it *removed*
                - Return the previously stored value at the duplicate location

- Modify the *toString* method to print "dummy" for deleted locations.
- Verify that your modified HashTable works correctly:
    - The inputs from part 1 (no collisions), should produce the same result as before.
    - Verify that inserting items with collisions and removing items works. As a minimum, you should check the supplied test cases.


put, Using Linear Collision Resolution
Is Table Full ?
T
Throw IllegalStateException
F
Get table index
1st put for this key
Is Bucket Empty ?
T
Store Entry
size++
return null
F
Duplicate put for this key,
Is Entry valid ?
T
Key equals Entry ?
T
Retain Entry value
OverWrite Entry with new value
Return previous value
F
F
Table index++
Searching for previous entries with this key
No previous entry found for this key
OverWrite previously Removed Entry with new value
Table index++
Is Bucket Empty ?
T
size++
return null
F
F
Is Entry valid ?
T
Key equals Entry ?
T
F
Previous entry found for this key
Mark Entry Invalid
Return value of removed Entry