CS310 – Advanced Algorithms and Data Structures

Spring, 2021

Hash Tables (Section 3.4) for pa2
Class 6: hash functions

- A hash function accepts a key and returns an integer value for it, which is used to place the key (and its value, if any) in the hash table, by computing a bucket number, bucketNo, that is, an integer between 0 and M-1 where the hash table is of size M.
- The notation h(x) is used in S&W for a hash function that directly returns a bucket number for key x. So bucketNo = h(x).
- But note that Java Object.hashCode() method may return an integer of any size, so that value needs to be further processed to determine a bucketNo for HashMap (project 2).
hash functions for HashMap

• Java Object.hashCode() method may return an integer of any size, so it needs to be further processed to specify a bucketNo.

• The JDK HashMap code (project 2) computes a bucket number from this hash by a formula equivalent to bucketNo = hash % M (where M is a power of 2).

• Because M is a power of 2, say $2^{10} = 1024$, this formula just extracts the low 10 bits of hash to specify bucketNo.

• The actual formula is $i = (n-1) \& hash$, where for $n=2^{10}$, $n-1 = 1111111111 \quad (10 \ 1’s)$, which also just extracts the low 10 bits of hash for i, the bucketNo.
Resolving collisions in hash tables

A collision happens when two keys hash to the same bucket, i.e., the same hash-value. What to do?

- Separate chaining. Make the hash table an array of linked lists. *This is what HashMap uses.*
- “Open-addressing” hashing. If the first spot is full, use another spot in the hash table.
  - linear probing: look in the next spot down/wrapped in the array.
  - “Quadratic” probing is also possible, but we won’t cover it. Weiss covers this case.
Separate Chaining: pp. 464-468

- Each hash array element is a (linked) list holding all the keys that hash to that bucket - all the collision participants. pa2: use ArrayList<Node> for the collision list. Node holds a key-value pair.
- This is the simplest way to make a hash table that works decently. Many hash tables work this way, including HashMap in the JDK*
- In some cases separate chaining may be a little slower than linear probing, because it causes memory references that hop around in memory more. This could be important for very large hash tables.
- *In Java 8+, the full HashMaps may switch a bucket from a linked list to a tree structure if the linked list gets too long. This mitigates bad hash function use. pa2: this coding has been removed.
Separate Chaining Data Structure
diagram from pg. 464

For pa2 picture: replace the SSSTs with little ArrayLists, one for each bucket.
• First add: key S, value 0, hash 2, find HT[2] list, empty, add S-0
• 2\textsuperscript{nd} add: key E, val 1, hash 0, find HT[0] list, empty, add E-1
• 4\textsuperscript{th} add: key R, val 3, hash 4, find HT[4], empty, add R-3
• 5\textsuperscript{th} add: key C, val 4, hash 4, push C-4 onto H[4] list, now has 3 elements
• 7\textsuperscript{th} add: key E, val 6, hash 0, replace E’s value with this one, \textsuperscript{note ST behavior}
• This is implementing a ST/Map: each key has only one value
• This is an array of objects, each of which contains a list.
• These are the collision lists of the hash table
• We generally try to keep them shorter than this!
• Each list is typed as SequentialSearchST, meaning it’s meant to be an ST, i.e. a Map: what’s the domain and range?

• Keys are strings, like “A”, values are numbers, like 8, so “A” $\rightarrow$ 8 in the first SSST, etc.

• So we’re making a big ST out of an array of little STs, each implemented by a singly-linked list. We’re using the hash function to assign elements pseudo-randomly to the little STs.

• Pa2: we’re making a big ST out of an array of lists, so we need to iterate through the collision list to locate the key
Implementation of Separate Chaining for ST

• This S&W design of disguising linked lists as little Maps makes for neat-as-a-pin code, but you need to remember that the little st.get has to loop down the list.

• Pa2: we do have to loop down the collision list.

• Well, there is a really ugly part needed to set up the needed hash table array, as we will see
public class SeparateChainingHashST<Key, Value> {
    ... instance variables, constructor
    private int hash(Key key)
    { return (key.hashCode() & 0x7fffffff) % m; }
    public Value get(Key key)
    { return (Value)st[hash(key)].get(key); }
    public void put(Key key, Value val)
    { st[hash(key)].put(key, val); }
    ... }

- Get and put both hash the key, use that hashval to locate the bucket in the hashtable, and thus the bucket’s own little ST/map, ready for get or put.
- This is the neat-as-a-pin code mentioned on the last slide. Pa2: we need to iterate through the collision list to find the key.
S&W Hashing with separate chaining: ugly code, pg. 465
HashMap has this ugliness too

public class SeparateChainingHashST<Key, Value> {
    private int m; // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // --see pg. 375: singly linked list
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[m];

    We can’t directly create an array of generic type in Java. So we create the array of raw type and cast it to the generic type, which causes warnings. We can suppress those warnings.

    Very ugly. Same ugliness shows up in JDK HashMap code. Can’t be avoided if you want the result to be a generic type, as we definitely do.

    Pa2: Has this ugliness too, see next slide.
Hashing with separate chaining: alternatively using array of LinkedList

```java
public class SeparateChainingHashMap<Key, Value> {
    private int m; // HT size
    private List<Map.Entry<Key, Value>>[]> st; // HT
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (List<Map.Entry<Key, Value>>[]) new
          LinkedList[m];
}
```

Again, we directly create an **array of generic type** in Java. So we create the array of raw type and cast it to the generic type, which causes warnings. We can suppress those warnings. Very ugly.

Pa2: we’re using ArrayList, similar.
FYI: Towards understanding Generic types and arrays

- A generic class like `HashSet<String>` doesn’t have its own source code (i.e. code for the `HashSet` of `String` case).

- Instead, it uses the common `HashSet` class (the “raw type”) that accepts elements of all object types via object references.
  - The `HashSet` code can call `element.hashCode()` since `hashCode` is an `Object` method.

- Java arrays are typed in a way so that if `x` is a subtype of `y`, the array of `x` is a subtype of array of `y`, so the array type handles the element type in an inheritance-sensitive way. Ref [Oreilly book](#).

- That means an array element can’t directly be a generic type object, which dissolves into an object reference in use. It doesn’t have a strong-enough type identity.

- But we can create an array of the “raw type”, which yields an array of references, and then use those reference spots with refs to objects created via generics. You do what you have to do.

- The linked chapter shows another way here: use a wildcard generic. Since it doesn’t specify a type, it’s allowed here. ... = new SequentialSearchST<?>[m]
public class SeparateChainingHashST<Key, Value> {
    private int m;  // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (SequentialSearchST<Key, Value>[]) new
    SequentialSearchST[m];
    // Now create the actual lists in the buckets--
    for (int i=0; i < m; i++)
        st[i] = new SequentialSearchST<Key, Value>();
    • Note that the book code has the raw type here, unnecessarily. No
    array type here to cause problems. Works either way.
Hashing with separate chaining: using array of LinkedList

```java
public class SeparateChainingHashMap<Key, Value>
{
    private int m;  // HT size
    private List<Map.Entry<Key, Value>>[] st; // HT
    // ... in constructor, create the hash table:
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (List<Map.Entry<Key,Value>>[]) new
        LinkedList[m];

    With this setup, get and put have more work to do than we saw in the S&W code. After they locate the bucket, they need to search the list in it for the key. If it’s found, access the value. If not, fail on get or add to the list for put.

    This search code is inside the little bucket STs for the S&W code.
```
Resizing the hash table to keep it not too full.

- It is important to keep hash tables less than half full (or 75% full) for good performance.
- This can be quite critical for performance.
- With Java, it’s particularly easy to resize a hash table because you don’t have to carefully dismantle the old one - you can point to a new table and just leave the old table to the garbage collector to clean up.
Here we create a T within a method of T (outside the constructor of course!)
We are using Java’s garbage collection (GC) to clean up the old data structure when we overwrite keys and vals, the master HT variables.
Pa2: this is one way to do it. See “steps” doc.
Performance of Resizing

- Typically, we resize every time the number of entries exceeds 2n, starting at the original half-full point.
- Consider doing a sequence of inserts to an empty hash table of size 64+, prime (M = first prime over 64, that would be 67):
  - insert 1, 2, 3, ..., 31 – still less than half full.
  - insert 32—first resize, to M=128
  - insert 33, 34, 35, ..., 63 –still less than half full (32 = 2^5 ops)
  - insert 64: second resize, to M=256 (64 = 2^6ops)
Performance of Resizing

- In general, during inserts \( n+1 \) to \( 2n \), we do \( n \) simple inserts, each \( O(1) \), plus \( 2n \) relocations into a new hash table, each also \( O(1) \).
  - This totals \( O(n) \) inserts and \( O(n) \) relocations.
  - If we amortize this over the \( n \) inserts, that is, average the total time over the total number of inserts between rehashes, \( n \), we get \( O(1) \) per insert.
  - Naturally there's a "hiccup" when the expansion happens, and this can be unacceptable if it exceeds some threshold time such as human-perceivable time in interactive programs.
  - There are more sophisticated "extensible hashing" methods that don't hiccup so badly.
Intro to inner classes

• Starting with Nodes for linked lists in S&W Sec. 1.3, we have seen private class definitions inside top-level classes.

• Specifically, the “little STs” in buckets of the separate chaining HTs, with code on pg. 375 (online), use an inner class Node

• This kind of class with definition inside another class is called an inner class. There are two kinds of inner classes in Java:
  1. static inner classes, AKA nested classes
  2. non-static inner classes, or plain “inner classes”

• S&W only uses inner classes (not static), and mostly so will we.
• Both kinds of inner classes can be used like ordinary classes.
• They can be hidden from client code with “private class…”.
• They are both used mainly for implementation details of the outer object.
public class Stack<Item> implements Iterable<Item> {
    ...
    private class Node {
        Item item;
        Node next;
    }
    ...

    • This inner class inside Stack.java doesn’t bother with “private Item item;” or “private Node next;” because the whole class is private inside Stack.
    • And the variables of the inner class are always accessible to the outer class’s code anyway.
    • S&W don’t discuss inner classes much, just a little note on pg. 159.
private class Node {
    Item item;
    Node next;
}

• Other code in Stack.java using Nodes:
  public void push(Item item) {
      Node oldfirst = first;
      first = new Node();
      first.item = item;  \(\text{access to item within Node}\)
      first.next = oldfirst;
      n++;
  }
Inner class code’s access to outer class’s variables

- Code in an inner class (but not in a static inner class) can use the *outer class’s* instance variables.
- This is important in implementing iterators.
- See code on pg. 155 inside class Bag<Item>:

```java
private class ListIterator implements Iterator<Item> {
    private Node current = first;
    ...
```

Outer class variable
Note on inner classes (not static)

• An Inner class object is bound to its outer class object.
  • It contains a ref to its outer object.
  • So the outer object can’t be garbage-collected as long as the inner class object is in use.
• Their creation always involves an outer object, and because of the close association, they know their outer object at any time.
• Their code can use the generic type parameter(s) of their outer class (here Item of Bag<Item>, pg. 155)

```java
private class ListIterator implements Iterator<Item>
...
   Item item = ...
```
JDK HashMap implementation

- The Java HashMap uses an array of Nodes, each with key, value, and Node implements Map.Entry.
- The Node for a hash bucket becomes the first Node of a singly-linked collision list, so they each have a next ref.
- The creation of the array has the same generic-type array problem as we saw with SeparateChainingHashST, with @SuppressWarnings(\{"rawtypes","unchecked"\}) to keep from showing a warning to users when the needed cast happens.
- Note: the HashMap constructor can accept an initial size for the hash table, saving on all those resizes in the case you can estimate the final needed HT size:

```java
Map<String> map = HashMap<String>(1000000);
```