CSE250 Week 13: Hash Tables (text ch. 22)

We have seen that a *balanced* binary search tree can provide all of `find`, `insert`, and `remove` in $O(\log n)$ time. It can also support "fingers" (i.e., iterators that need not start at the beginning) and traversal in amortized $O(1)$ time per step in a definite and predictable order---namely, the sorted order of keys. The ordering guarantees that if the data structure is allowed to have different items with the same key (thus functioning as a **bag** or **multiset** or even a **multimap**), then those items will be consecutive in the iteration order---though with some issues to beware of. Plus, problem (3) of the "virtual A7" showed how balanced trees can support indexed access in $O(\log n)$ time.

The "Holy Grail" of associative lookup is to get the three main operations in $O(1)$ time. Various kinds of **Hash Tables** promise this, but bring more caveats. Here is the basic idea:

- The issue with an array A is that if we know in advance the index i of an item, then we can access it in $O(1)$ time as $A(i)$.
- But if we don't know in advance, i may have no relation to the data of an item. If we only have the item, including its key k, this does not help us find it---and `find` takes $O(n)$ time in an unsorted array.
- If we only want to look up items, not care about their **local** relation to other items, then we don't care what the index i actually is.
- Suppose we can associate i uniquely to the key k by a function $i = h(k)$ where h is quick to compute. We need $0 \leq h(k) < m$ where m is the size of the array.
- Then given any `item` we can find it as **`A(h(item.key))`**.

Counting the evaluation $h(k)$ as unit time, this works in $O(1)$ time. Note that we have been comparing keys and calling that unit time per comparison---we can suppose that computing $h(k)$ is just as quick as a comparison *and we only need to do it once*. The function h is called a **hash function**, A the **hash table**, and the technique is called **hashing**. One other note:

- It will not matter if some indices i are unused. All we waste is some memory for making the array larger than perfect for n items---and we will begin with a kind of hashing where the array size m can be less than n.
- Hence, inserting and removing need not involve re-allocating the array.
- "**Holes**" in the array---especially when left by removals---will still be a pain, however.
- Nor can we allow direct updates $A(i) = newItem$ because the new item might not have i as its hash index. We have to work only through the associative-lookup API.

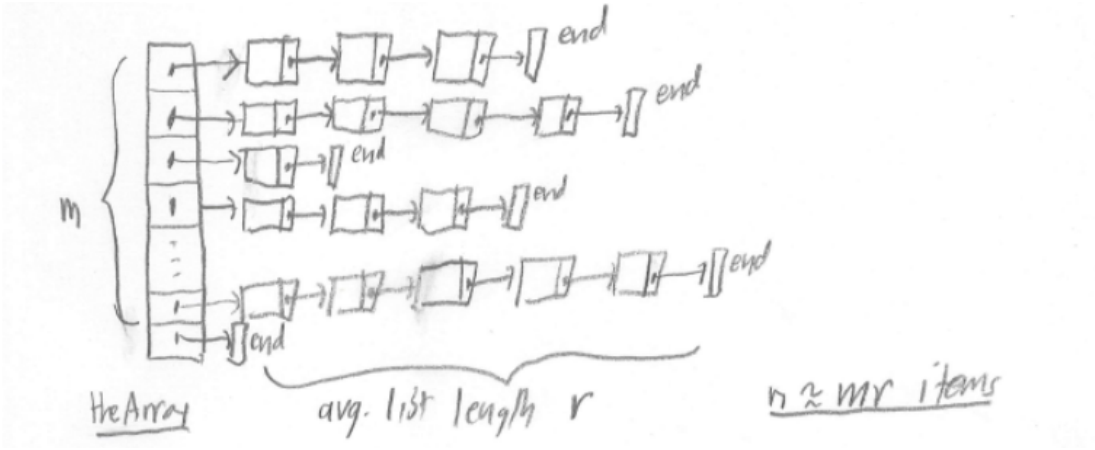
Collisions and Kinds of Hashing

The next pressing issue---before the anticlimax of how to design a hash function, is, what if two different keys k and k' make $h(k) = h(k')$? We can expect such a **hash collision** to occur as soon as

the number n of items becomes about $\sqrt{m}$, owing to the **Birthday Paradox**. We don't want to overwrite one item by another. Two ideas:

1. Make $A(i)$ hold a (linked) list of items having the same hash index i. Then just store such duplicate-hash items in the same list, and use the list's own `find` operation to find them.
2. Use a **probing rule** $p(i,t)$ to find an empty slot i' at which to store the new item directly into the array. Here t counts probes (the text uses j, IMHO confusingly), and we regard the zeroth probe $p(i,0) = i$ as being the original index $h(k') = i$.

Method 1 is called **chaining** and is covered in **section 22.2**. We can recycle the architecture of the "`AIOLI`" class to describe and implement it.



Chaining allows the **load factor** $r = \frac{n}{m}$ to be greater than 1. The text uses a field `fillingFactor` to represent it as a target for method 2 (which is called **open addressing** and covered in **section 22.3**) but does not otherwise define it. We will recycle our notation n, m, r from `AIOLI` (and `BALBOA`) as on Assignment 5. Then r is the average length of one of the linked lists. Provided we choose $m = \Theta(n)$ (indeed, if saving array memory is not an issue, we can take $m = n$ or even $m > n$, especially when we know in advance how big n is going to be), we will get $r = O(1)$ (**usually**). That means the extra time for exhaustive search in a linked list with multiple items will still be $O(1)$. Since adding and removing from a linked list takes $O(1)$ time, we get all three major operations in $O(1)$ time overall.

Open addressing can do the same, but has extra quirks and caveats. Let's table it for later.

Hash Functions

The last implementation detail is how to compute $h(k)$. One niggle of nomenclature is that we ultimately need $0 \leq h(k) < m$, but we can enfore this after-the-fact by taking "$h(k) \bmod m$". Then we can consider hash functions h to have arbitrary integer values and simplify thing by not having to consider "m" to be part of their definition. We can even allow negative integer values, hence not have

to worry about overflows with `Int`, provided we note:

- Scala uses the rule that `i % m` has the same sign as i. So it can be negative.
- We can use `abs(i % m)` but this reflects the modulus around 0.
- A mathematical purist will prefer `Math.floorMod(i,m)`, whose **contract** is to have the same sign as m. So if $m > 0$ its value is always between 0 and $m-1$.

Thus, the hash-table class can do its own enforcement of the class invariant $0 \le h(k) < m$. We can call $h(k) = i_0$ the **primary hash code** of the key k itself, and the actual index $i = i_0 \bmod m$ the **secondary hash index**, but it is usually OK to slur this distinction. This leaves us free to consider arbitrary hash codes and functions for computing them.

Designing hash functions used to be a fun and creative topic---and still is in specialized, performance-intensive applications. Coverage would begin by noting that the simple idea of adding up the alphabet numbers or ASCII codes of the characters in a string, like `h(life) = 12+9+6+5 = 32` (not 42) is a poor one, because it produces too many collisions. The way of doing this in Hebrew called gematria partly does this but in more of a radix manner, using letters as digits, so it spreads out codes more. In gematria, the code of "living" is 18 and the code of "Neron Kesar" is 666. The Arabs had a similar way of using abajad numerals but they switched to a purely-radix idea called Arabic numerals which you may have heard of. In general, you want the has code to involve some multiplications as well as additions, and ...

...well, Java took all the fun out of this by providing a built-in `hashCode` method, and Scala extends this automatically *to any value type*. Except for performance-critical cases, such as chess programs, there is no need to do anything more clever. The topic is anticlimactic.

The further effect of using `hashCode` is that one can hide implementation details from the client. The need of the hash function to depend on how the **client type** (meaning what you use for the generic parameter "`A`") defines its keys used to force its presence to be visible at client scope. I have done this anyway in the "ISR" framework: the line in `Cardbox.scala` to select a hash table forces a "**client container**" (such as `FlowerCardbox` or `SynonymBox`) to pass in different parameters from the single `keyComp` comparator (which worked for all the sorted data structures, but incompletely when it comes to matching whole items).

I could have harmonized the interface across the board by fixing a "`designForSize`" parameter m and requiring the client type to be a value type. The latter not only makes `hashCode` automatic but also guarantees that `equals` can serve as an `itemMatch` function from `keyComp`. *(But then I would not have been so easily able to compose my data structures in a compound manner.)* But it is more useful to point out the differences first before saying how one could hide them.

Implementing Hashing With Chaining

I made the `HashISR` code by editing the previous `AIOLI.scala` file. (It includes `Iter#clone` now.)

```
/** REQ: numSlots > 0; itemMatch should match whole items not just hash key
    Enforces CLASS INV that 0 <= ind < numSlots via Math.floorMod(hash, numSlots)
    Note: as in Java, hashCode can be negative(!) and then % m would give index < 0.
 */
class HashISR[A](numSlots:Int, hashFun: A => Int, itemMatch: (A,A) => Boolean)
                                                          extends ISR[A] { Outer =>
   var theTable: Array[DLLISR[A]] = Array.fill(numSlots)(new DLLISR[A](itemMatch))
   private var _size = 0
```

The iterator class `Iter` is compound: it uses both the hashed array index `ind` and the linked list's own iterator `litr` to locate an item. The text actually does the same, in a way that is IMHO harder to read because the class is anonymous and the array index is (unnecessarily, IMPHO) generalized to be the first of two sub-iterators. Plus it uses tuple syntax because it hard-wires the whole data structure to be a Map with distinct items. Once you have the iterator class, the main operations are quick to code---except for the Scala inner-class handling:

```
   def find(item: A): Iter = {
      //val ind = hashFun(item) % numSlots    //no!!
      val ind =  Math.floorMod(hashFun(item), numSlots)
      val litr = theTable(ind).find(item)
      if (litr.hasNext) {
         return new Iter(ind, litr)
      } else {
         return end
      }
   }

   def insert(item: A): Iter = {
      val ind = Math.floorMod(hashFun(item), numSlots)
      val thisList:DLLISR[A] = theTable(ind)
      val litr = thisList.insert(item, theTable(ind).begin.asInstanceOf[thisList.Iter])
      _size += 1
      return new Iter(ind, litr)
   }

   /** Cannot violate the CLASS INVs, so OK to use freely.
    */
```

```
def remove(loc: Iter): A = {
   assert(loc.hasNext, "Attempt to remove past-end item")
   //needs the component loc.iat to be on a real element
   _size -= 1
   val thisList = theTable(loc.ind)   //val needed for Scala "stability"
   val tmp = thisList.remove(loc.iat.asInstanceOf[thisList.Iter])
   return tmp
}
def remove(item: A): A = {
   val itr = find(item)
   assert(itr.hasNext, s"Attempt to remove non-found item $item in HashISR\n")
   return remove(itr)
}
```

When a "hash bucket" becomes empty after removal(s), we just leave the empty list in its slot. Of course we don't contract the array---even just changing the size m would have chaotic effect on the entries.

Iteration With Chaining

When using the list iterator as a component, we face the problem of advancing it to its own `end` position on an individual list. Unless we are at the end of the last nonempty list in the whole table, we need to step it to the next element---which will be the first element of the next list that is nonempty. This adjustment is done by a private helper method `adjustBin()` that is much like the text's `nextBin()` at the bottom of page 613 going to 614. It is needed even to initialize an iterator, because we don't know whether slot 0 will be empty:

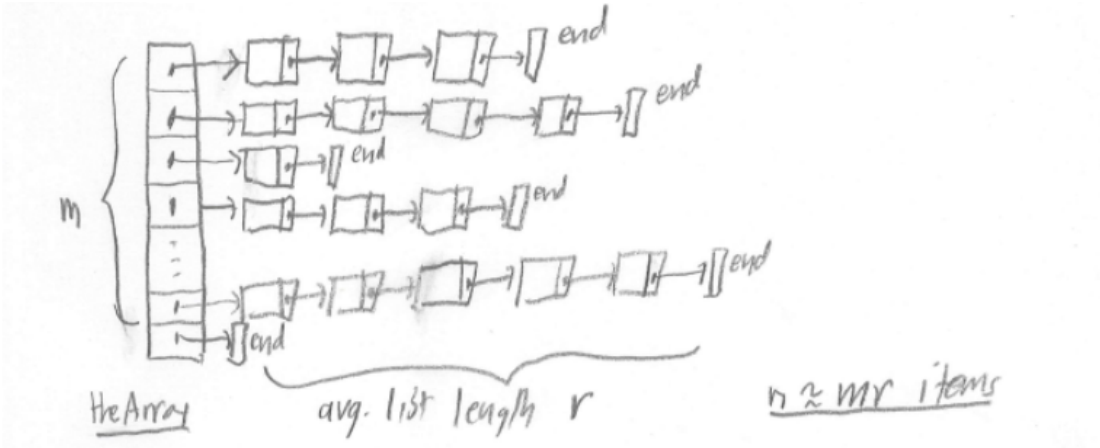
```
def begin: Iter = {
   var itr = new Iter(0, theTable(0).begin)
   itr.adjustBin()   //puts on first element of whole data structure
   return itr        //or end if the whole container is empty
}
def end: Iter = new Iter(numSlots, theTable(numSlots-1).end)
```

The iterator class itself should enforce the invariant that the iterator is never at an "indeterminate position", meaning at the end of a list but not at the end of the data structure overall---which should involve the end position of the table itself, i.e., theTable.length (which we call m).

```
/** Iter adds three methods to standard Scala next() and hasNext for iterators.
    INV1: Iter is attached to the list node it designates, not its "pre"
    INV2: Iter is never at the end position of a chain.
```

```
    INV3: End iterator has ind==numSlots; then we don't care about liat
 */
class Iter(var ind: Int, var liat: DLLISR[A]#Iter) extends Iterator[A] {

   /** Special Scala syntax allows using just parens to return the data item.
    */
   def apply(): A = {
      assert(hasNext, "Attempt to fetch item past end in HashISR\n")
      return liat()   //note re-use of DLLISR#Iter.apply() here
   }
```



The tricky part is stepping over "holes":

```
   private[HashISR] def adjustBin() = {      //like nextBin() in the text
      if (!liat.hasNext) {
         ind += 1
         while (ind < numSlots && theTable(ind).isEmpty) { ind += 1 }
         if (ind < numSlots) {
            liat = theTable(ind).begin
         }
      } //otherwise leaves at ind==numSlots end position, don't care about liat
   }
```

We call it mainly when stepping the iterator:

```
   def next(): A = {
      assert(hasNext, "Attempt to advance past end in HashISR\n")
      //if (liat.hasNext) {
         val ret = liat.next()
         adjustBin()
         return ret
```

```
      //} else {          //should we forgive a violation of INV2 here??
          //adjustBin()
          //return next()  //not infinite loop, but could lead to assertion violation
       //}
    }

    def hasNext: Boolean = liat.hasNext   //absolutely relies on INV2 holding
    //def hasNext: Boolean = { adjustBin(); liat.hasNext }  //"defensive driving"
```

Open Addressing

Chaining can allow n to grow without limit. As the load factor $r = n/m$ grows, performance will degrade, but not in a terrible or sudden manner. Open-address hashing needs $r < 1$ and carries more risks as r approaches 1. The kind of risk depends on the probing rule $p(i,t)$:

- **Linear probing**, defined as $p(i,t) = i + t \bmod m$ (or $= i - t \bmod m$), avoids an infinite loop unless $n = m$, but degrades performance more quickly.
- **Quadratic probing**, most simply implemented as $p(i,t) = i + t^2 \bmod m$, tends to need fewer probes than linear probing, but can have infinite loops unless a **probe count** is used. The text uses a somewhat smarter probe function: $p(i,t) = i + 2t + 3t^2 \bmod m$.
- **Two-level** and **multi-level** hashing uses a second hash function (or more) in case of collisions.
- **Cuckoo hashing** (not in the text) is a two-hash scheme that keeps `find` and `remove` in **guaranteed** $O(1)$ time, at the cost of more disruption when inserting: items with other hash codes may get moved.

Open addressing needs a way to tell whether a slot is filled, truly empty, or "Xed" (the text says "REMOVED"). To illustrate why, here is a jocular but portentous example: Suppose `Doc Smurf` has hash code 6 while `Lego Mario`, `Lego Luigi`, `Lego Bruno`, and `Lego Waldo` all have the same hash index 4 in a table of size 32. This could happen if the hash function made the last j bits depend only on the first 4 chars and the table size m is a multiple of 2^j. (For this reason, the text says to avoid sizes near powers of 2, but this is more a fault of the hash function---and the larger reason for prime sizes IMPHO has to do with quadratic probing. Many chess programs use *only* power-of-2 table sizes.) Inserting them in that order under linear probing gives:

```
0 : _
...
4 : Lego Mario
5 : Lego Luigi
6 : Doc Smurf
7 : Lego Bruno
8 : Lego Waldo
```

```
9 : _
```

Now suppose we removed `Lego Bruno` from the catalog. If we replaced it by _ for empty, then we would "maroon" `Lego Waldo` and subsequently be unable to find Waldo. But if we do

```
0 : _
...
4 : Lego Mario
5 : Lego Luigi
6 : Doc Smurf
7 : Xed
8 : Lego Waldo
9 : _
```

then Waldo is still in touch. To implement this scheme without forcing the table client type to make a Boolean field for "Xed", or forcing the table class to "wrap" the client items in a class that has such a field, we can use `null` to mean empty and a default value `A()` or `A.zombie` to mean "Xed". This is just as much needed with quadratic probing and multi-level hashing, though not with cuckoo hashing. We need an `itemMatch` call between a real item and a zombie to be legal but return false. The code for `find` then becomes (caveat: not pasted from actual code):

```
def find(item: A): Int = {   //returns index of slot found, m for not-found
   val ind = Math.floorMod(hashFun(item), numSlots)
   var t = 0
   var try = probe(ind,t)    // == ind on first use
   while ((!empty(try)) && (!itemMatch(item, A(try)))) {
      t += 1
      try = probe(ind,t)
   }
   if (empty(try)) return theTable.length else return try
}
```

We couldn't include a test for the hash code of the visited items being equal to that of the given item because we needed to keep going past `Doc Smurf` with a different code. Nor do we need to test for "Xed" in a field-specific manner if we assure that item-matching a zombie will return `false`. For the same reason, it is OK to *overwrite* a zombie when inserting:

```
def insert(item: A): Int = {   //returns index of slot found, m for not-found
   val ind = Math.floorMod(hashFun(item), numSlots)
   var t = 0
   var try = probe(ind,t)    // == ind on first use
   while ((!empty(try)) && (!xed(try))) {
```

```
        t += 1
        try = probe(ind,t)
      }
      A(try) = item
      return try    //this is like return new Iter(...) in the ISR convention
   }
```

The code for `remove` can call `find`, but must set the slot of a found item to "xed" (i.e., a zombie) rather than empty it. The main drawback is that the call reduces the "actual" load factor n/m but not the "effective load factor." And a case where quadratic probing goes loopy even with low effective load is:

```
0 : Lego Bruno
1 : _
2 : _
3 : _
4 : Lego Luigi
5 : Lego Mario
6 : _
7 : _
```

Where can we insert `Lego Waldo`? The probe tries

$$
\begin{aligned}
&4,\\
&4+1^2 \,\%\, 8 \;=\; 5,\\
&4+2^2 \,\%\, 8 \;=\; 8 \,\%\, 8 \;=\; 0,\\
&4+3^2 \,\%\, 8 \;=\; 13 \,\%\, 8 \;=\; 5,\\
&4+4^2 \,\%\, 8 \;=\; 20 \,\%\, 8 \;=\; 4,\\
&4+5^2 \,\%\, 8 \;=\; 29 \,\%\, 8 \;=\; 5,\\
&4+6^2 \,\%\, 8 \;=\; 40 \,\%\, 8 \;=\; 0,\\
&4+7^2 \,\%\, 8 \;=\; 53 \,\%\, 8 \;=\; 5,\\
&\ldots
\end{aligned}
$$

all stay blocked by existing elements, so you get an infinite loop unless you spend code steps to check a probe-count limit. This problem---whose theory the mathematician Carl Friedrich Gauss did the most to delineate over 200 years ago---is reduced by making the table size m a prime number. It still can bite hard when the effective load factor gets above **0.85** or so. All these risks said, however, open addressing with well-judged load parameters is the fastest method in general practice.

(Aside, FYI) Hashing in Chess Programs

One special kind of hashing scheme worth mentioning *en passant* is when all data points will be combinations from of a limited set of possible "atomic features" f. Then you assign a fixed primary hash code h_f to each feature as a binary string. The primary hash code of a data point P then becomes

$$h(P) = \bigoplus_{f \in P} h_f ,$$

which means taking the bitwise-XOR of hash codes of all the features present in P. Given a table size m, the secondary hash converts $h(P)$ into a value in the range $0..m-1$. Quite often $m = 2^k$ for some k and the secondary hash just takes k bits off either end or some other way. Assuming the atomic hash codes h_f use "truly random" bits, there is little danger of the "`Lego`" kind.

The basic features in a chess position P are the identity of a piece on a particular square, giving $12 \times 64 = 768$ atomic codes, plus whether White or Black is to move, whether each is still allowed to castle kingside or queenside, and whether *en passant* captures are possible (13 more codes for 781 total). Each code is a 64-bit unsigned long int, as is $h(P)$. If you allocate, say, 512MB to a chess program's hash table, then at 16 bytes per entry (8 to store $h(P)$ itself, and 8 for the position's value along with how recently in the search it was updated), you get 32M = 2^{25} entries. So the secondary (i.e., actual) hash code has 25 bits. This scheme was first used for chess, checkers, and Go by Albert Zobrist in 1969 and is named for him, but a more generic name is **tabulation hashing**.

Since $\sqrt{32M}$ = under 6,000 while chess programs search over a million positions per second even on a single core, the birthday paradox hits instantly and collisions mushroom. Some chess programs do limited probing, but they mostly take the "zeroth option" of open-address hashing: the new value simply overwrites the old. The **minimax process** stops the propagation of almost all collisions. It be like: a bad low value in a max-heap just sinks like a stone out of the picture. With White and Black alternating moves, any extreme value (from a position where one side is a queen ahead, say) that overwrites a normal value gets similarly "minimaxed away" by one or the other. But there have been times when a rogue value from a hash collision has propagated to the root of the search and made a chess program play a blunder. The most famous case is #8 in this list, and I reproduced it in 2011. I've found other positions that propagate hash collisions for multiple chess programs, notably this and this. The zeroth option of not handling collisions is OK in chess because "speed is king and nobody dies."

Iteration and Fingering Gotchas in Hash Tables (not in the text)

Iteration with open addressing is simpler than with chaining: the `next()` method must "step over zombies" as well as holes, but does not have the problem of intermediate `end` positions of lists. Both forms have caveats beyond the immediately obvious one: the iteration order has no relation to sorted order or the order of insertion.

1. When you expand the capacity of an array, the elements generally have to be recopied but their sequence is preserved. When you do this in a hash table, however, the "mod m" values change

in a way that makes elements jump around. So you cannot rely on consistency of the iteration order unless the table size is absolutely fixed.
2. If you have multiple items sharing the same key, they will of course have the same hash code. But they need not be consecutive in the iteration order, even with chaining. Items with different keys giving the same hash code may come between them in the linked list.
3. *Well*, if those duplicate-key items are inserted at consecutive times, then their records *will* be consecutive in the linked list. (The code of `HashISR.scala` reverses the time order because it prepends to the bucket's list.)
4. The code for the task in Assignment 6 works **generically** in any sorted data structure, but not when the `SynonymBox` implementation is changed to a hash table---*unless* we can rely on chaining being used and point 3 not being broken. [Show demo code `SynonymsISR.scala`.]
5. And with open addressing, *fuhgeddaboudit!*
6. With chaining, we could *re-program* the task to skip over different-key items within the one hash bucket, thus searching the whole bucket. The new code is not algorithmically generic, however, because at client level it relies on publicly identifying the bucket within the iteration. (The `Iter` class does not expose the subsidiary `DLLISR#Iter liat` component and its own `end` position to the public.)
7. The re-programmed task *could* work with open addressing, because you could identify the **probe sequence** being searched as terminated by an `empty` slot, and thus treat it as if it were a chained-hash bucket list. The only difference would be the extra cost of stepping over "zombies" and items like `Doc Smurf` with different keys in that sequence. (If you understand how and why this works, then you can say you really understand open-address hashing.)

Despite some leeway in points 6 and 7, the essence is that hash tables are used mainly for strict Sets and Maps. In fact, a map in Perl is called a "hash." However you slice it, the fact that a hash table cannot be meaningfully `sliced` is a limitation on functionality. The implementation of the default `Set` and `Map` in Scala seems to be the same as `HashSet` and `HashMap` on our systems ---and this accounts for `Set` and `Map` being divorced from the chunk of the Scala hierarchy that deals with `Seq`uences. Not even `SortedSet` and `SortedMap` seem to relate them, hmmm.... Those use a balanced binary search tree implementation, our last regular topic.

Footnote: Wikipedia https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree on BSTs versus hash tables:

[S]elf-balancing BSTs have a number of advantages and disadvantages over their main competitor, hash tables. One advantage of self-balancing BSTs is that they allow fast (indeed, asymptotically optimal) enumeration of the items in key order, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple items with the same key. Self-balancing BSTs have better worst-case lookup performance than hash tables (O(log n) compared to O(n)), but have worse average-case performance (O(log n) compared to O(1)).

We will see self-balancing **AVL trees** next.