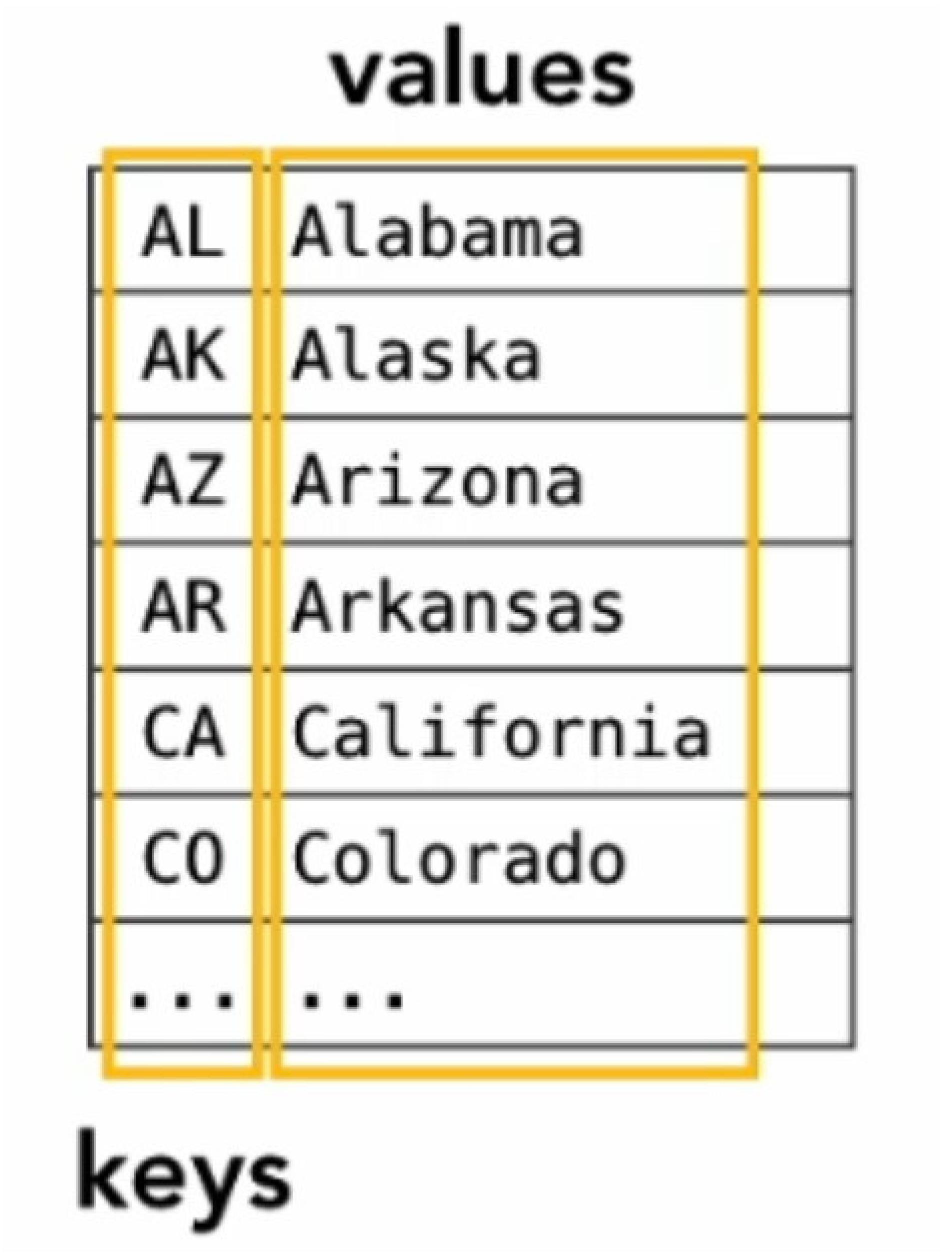
Symbol table

A symbol table is a Abstract Data Type for key-value pairs that supports two operations: insert (put) a new pair into the table and search for (get) the value associated with a given key. The primary purpose of a symbol table is to associate a value with a key. The main operations of a symbol table are put, get, delete, contains & isEmpty.

Also known as maps, dictionaries, or associative arrays.


values
keys

AL	Alabama
AK	Alaska
AZ	Arizona
AR	Arkansas
CA	California
CO	Colorado
...	...

API for a generic basic symbol table

public class ST<Key, Value>	
ST()	*create a symbol table*
void put(Key key, Value val)	*put key-value pair into the table (remove key from table if value is null)*
Value get(Key key)	*value paired with key (null if key is absent)*
void delete(Key key)	*remove key (and its value) from table*
boolean contains(Key key)	*is there a value paired with key?*
boolean isEmpty()	*is the table empty?*
int size()	*number of key-value pairs in the table*
Iterable<Key> keys()	*all the keys in the table*

Design choices:

A best practice is to make Key types immutable, because consistency cannot otherwise be guaranteed

- *Duplicate keys:*
  - Only one value is associated with each key (no duplicate keys in a table).
  - When a client puts a key-value pair into a table already containing that key (and an associated value), the new value replaces the old one.

  These conventions define the associative array abstraction, where you can think of a symbol table as being just like an array, where keys are indices and values are array entries. In a conventional array, keys are integer indices that we use to quickly access array values; in an associative array (symbol table), keys are of arbitrary type, but we can still use them to quickly access values. Some programming languages (not Java) provide special support that allows programmers to use code such as `st[key]` for `st.get(key)` and `st[key] = val` for `st.put(key, val)` where key and val are objects of arbitrary type.
- *Null keys:*
  Keys must not be null. As with many mechanisms in Java, use of a null key results in an exception at runtime
- *Null values:*
  We also adopt the convention that no key can be associated with the value null. get() should return null for keys not in the table.
  This convention has intended consequences:

First, we can test whether or not the symbol table has a value associated with a given key by testing whether get() returns null.
Second, we can use the operation of calling put() with null as its second (value) argument to implement deletion.

- *Deletion:*
  Deletion in symbol tables generally involves one of two strategies: lazy deletion, where we associate keys in the table with null, then perhaps remove all such keys at some later time; and eager deletion, where we remove the key from the table immediately (one application is to ensure that no key in the table is associated with null.).

Below are various possible implementations of Symbol Table

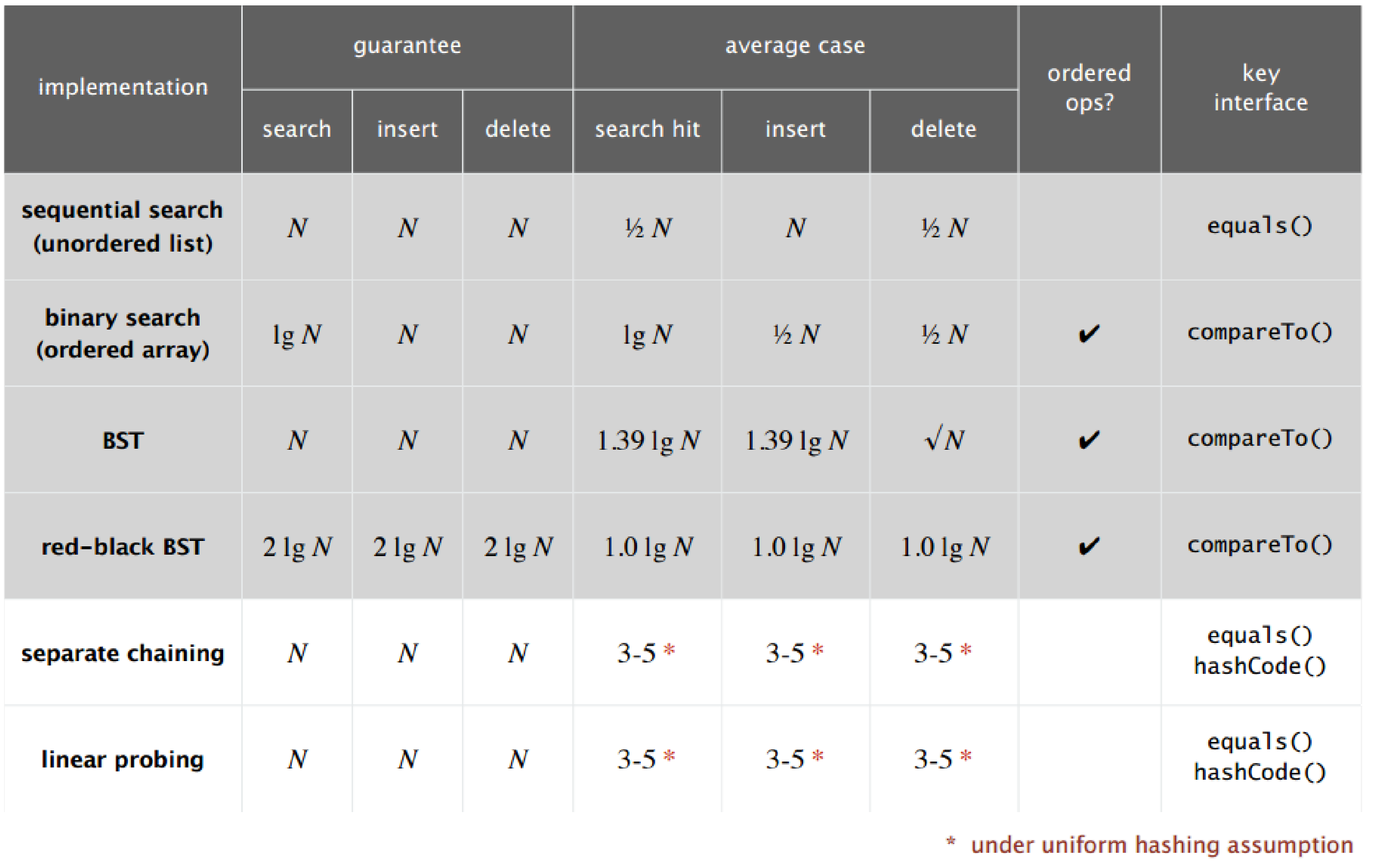
- Hashing based implementation (Hash Table) { notes link }
- Ordered ST implementation { notes link }
  - Using Array
  - Using LinkedList (keys lies in array in sorted order, while values lies in Linked-List)
  - Binary search tree based implementation
  - Balanced BST based implementation
    > > It is an extension of binary search trees implementation and takes O(logn) in worst case for search, insert and delete operations.
  - Ternary search implementation
    > > This is one of the important methods used for implementing dictionaries.
- Unordered ST implementation { notes link }
  - Using Array
  - Using LinkedList

Java Includes

```
Red-Black BSTs: TreeMap, TreeSet.
 – TreeMap is more space-efficient.
 – TreeMap only works with Comparable objects.

Hash Tables: HashMap, IdentityHashMap.
 – HashMap will be more efficient in general, so use it whenever you don’t care about the order of the keys.
 – HashMap is more time-efficient.
 – HashMap only works with objects with a suitable hashCode() implementation.
 – In HashMap, when there is a collision, a single bucket stores a list of entries.
```

Performance Comparison:



implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

* under uniform hashing assumption

References:

- Blog on ST
- Princeton ST Notes