# CS61B Lecture 31

Friday, April 10, 2020

## Announcements

- Project 3, Gitlet, has been released. It is due on May 1st, with a checkpoint due shortly before. No skeleton code is being provided as part of the project.
    - Several interesting features of Git that you may need to know will be covered in lecture in the next few weeks

## Trie

The worst case cost of a comparison for a `String` is its length, thus a

$$\Theta(M)$$

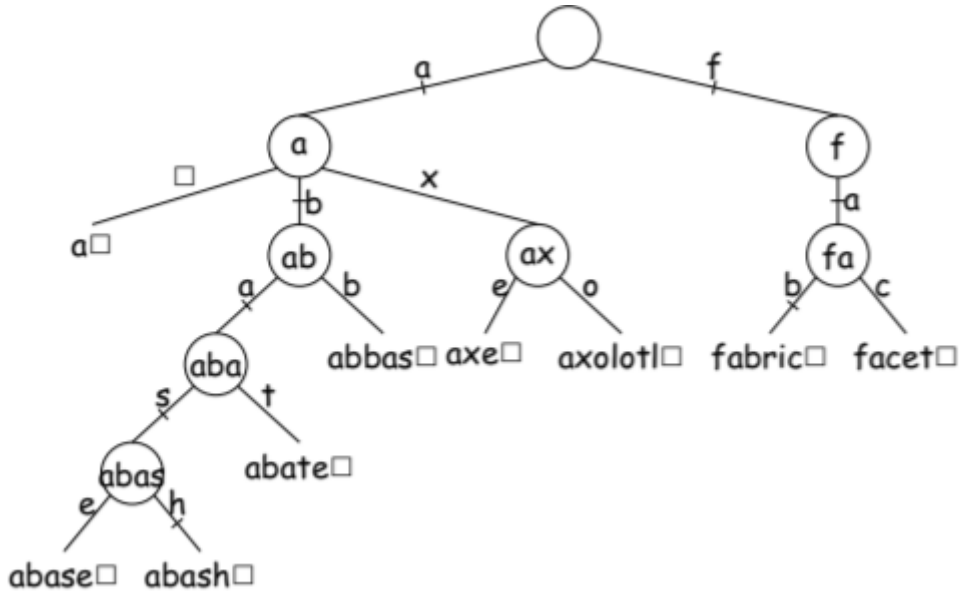
comparison really means

$$\Theta(ML)$$

where ML is the length of the string. To look for key `X`, we keep looking at the same characters of `X` `M` times. Can we do better?

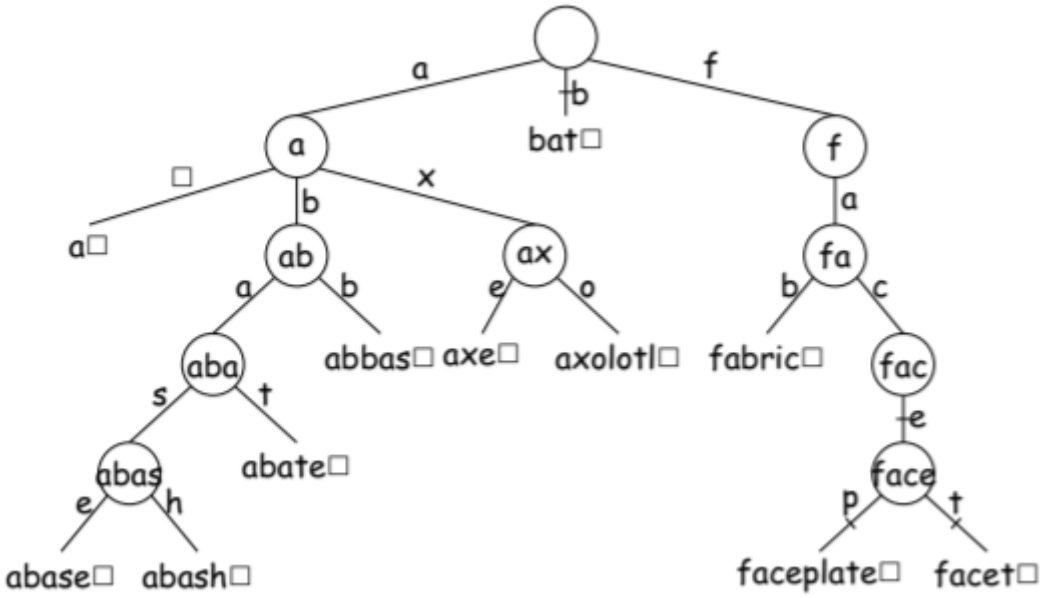
With the set of keys:

```
{a, abase, abash, abate, abbas, axolotl, axe, fabric, facet}
```




The ticked lines show the paths follow for "abash" and "fabric". Each internal node corresponds to a possible prefix.

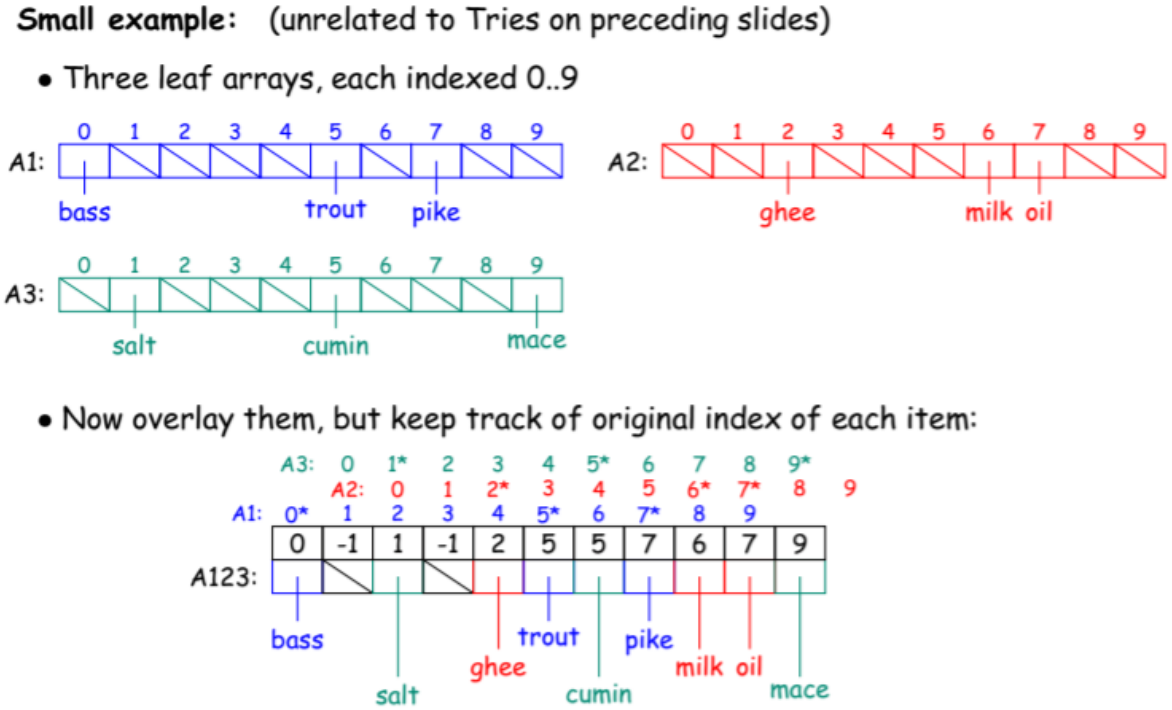
Adding items to a trie is show in the below example, adding "bat" and "faceplate", where the new edges are ticked:




## Scrunching Arrays

This isn't really related to tries (we could have internal nodes indexed by character), but say we have multiple sparsely populated arrays, with the remainder being null values. This is a waste of space.

How can we fix this? Well, we can put arrays on top of each other, using null entries of one array to hold non-null elements of another, using extra markers to tell which entries belong to which array.




Why shouldn't we do this in tries? It's hard to expand our try, the idea is a bit complicated, and this is really more useful for representing large, sparse, fixed tables with many rows and columns.

The number of children in a trie tends to drop drastically when one gets a few levels down from the root, so we should use arrays for the first few levels, which usually have more children, and then linked lists for the lower levels.

# Skip Lists

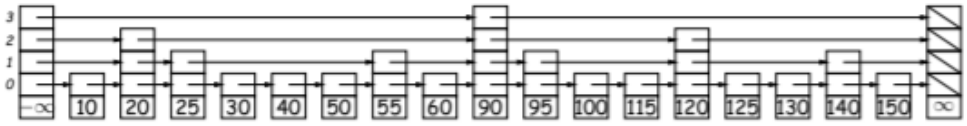
A skip list can be thought of as a kind of `n`-ary search tree in which we choose to put the keys at "random" heights. More often thought of as an ordered list in which one can skip large segments.


To search, we start at the top layer on the left, search until the next step would overshoot, then go down one layer and repeat. When we search for 125 and 127 in the above example, the gray nodes are looked at, and the darker gray nodes are overshoots.

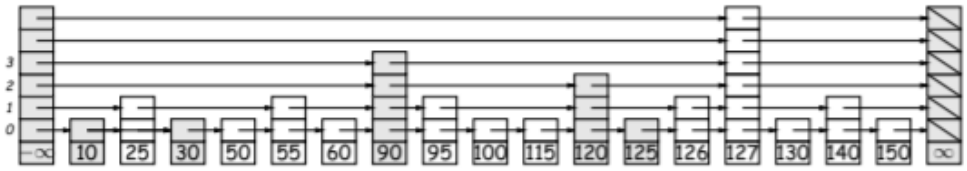
The heights of the nodes were chosen randomly, such that there are about half as many nodes that are `k` or higher as there are that are `k` high. This technique, called **probabilistic balancing**, makes searches fast with high probability.

To add to and delete from skip lists, the procedure is shown below:

- Starting from initial list:




- In any order, we add 126 and 127 (choosing random heights for them), and remove 20 and 40:




- Shaded nodes here have been modified.

# Summary

Balance in search trees allows us to realize

$$\Theta(lgN)$$

performance.

- B-Trees and RBTs give us

$$\Theta(lgN)$$

  performance for searches, insertions and deletions. B-Trees are good for external storage because the large number of nodes minimize the number of I/O operations.
- Tries give

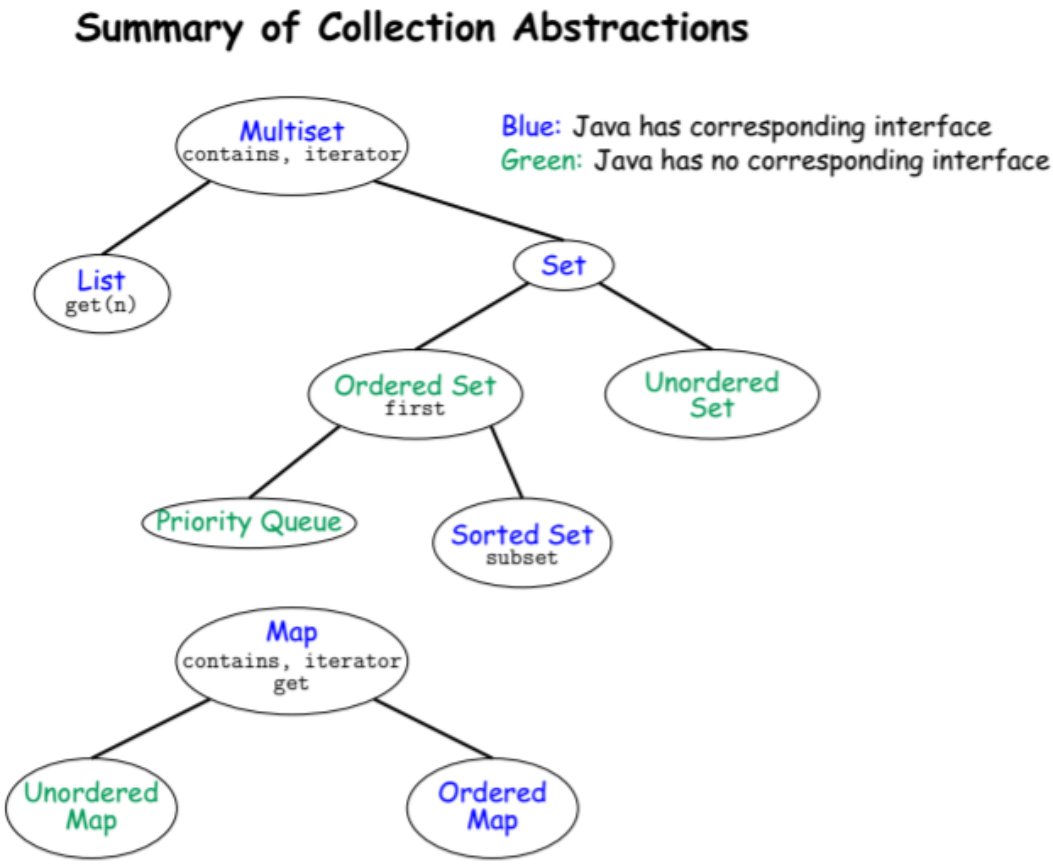
$$\Theta(B)$$

performance for searches, insertions and deletions, where B is the length of the key being processed. However, they are hard to manage space efficiently in.

- Skip lists give us probable

$$\Theta(lgN)$$

performance for searches, insertions and deletions. They are easy to implement, and are presented here for interesting ideas of probabilistic balancing and randomized data structures.



# Corresponding Classes in Java

**Multiset** (Collection)

- List: ArrayList, LinkedList, Stack, ArrayBlockingQueue, ArrayDeque
- Set
  - OrderedSet
    - Priority Queue: PriorityQueue
    - Sorted Set (SortedSet): TreeSet
  - Unordered Set: HashSet

**Map**

- Unordered Map: HashMap
- Ordered Map (SortedMap): TreeMap

e