Sets and Tries

Kris Micinski
Set (ADT)

- add(set, element)
- contains(set, element)
- union(set, set)
- intersection(set, set)

Many possible implementations of sets!

- Hash tables
- Balanced BSTs
- Lists
Sets as Hash Tables

• Exercise: How would you perform:
  • Insert
  • Lookup
  • Union
  • Intersection

• What is runtime (big-O) of each, assuming good hash function, m buckets, and at most n values in each set
Sets as Hash Tables

- Exercise: How would you perform:
  - Insert
  - Lookup
  - Union
  - Intersection

  Assume you wanted a persistent version of each operation, how do runtimes change?

- What is runtime (big-O) of each, assuming good hash function, $m$ buckets, and at most $n$ values in each set
Upshot: hash tables decent at many common set operations

(But better implementations exist)
Sets as BSTs

(Imperative version..)

- Exercise: How would you perform:
  - Insert
  - Lookup
  - Union
  - Intersection

- What is runtime (big-O) of each? Assume n elements in set
Sets as BSTs

• Exercise: How would you perform:

  • Insert
  • Lookup
  • Union
  • Intersection

Assume you wanted a persistent version of each operation, how do runtimes change?

• What is runtime (big-O) of each? Assume \( n \) elements in set
Upshot: BSTs give us better persistent hash behavior

(But still better implementations exist!)
Assume $m$ buckets

On average, $k$ links per bucket

Observation: if you have $m \times k$ items in table, lookup takes $\sim k$ time
Assume $m$ buckets

On average, $k$ links per bucket

Question

If you want to lower per-item lookup time, what do you do?
Observation

More buckets = faster lookup

(To a point… Then you bottom out)
Exercise

• Load ~500k words into dictionary

• Then, perform ~500k lookups
  
  • Not good benchmark of real-world use b/c uniform dist

• Expectation: bigger hash table = lower lookup time

• What real-world problem does this solve..?
# Load `words.txt` into a hash table of size s
def loadIntoTable(s):
    print("loading words into table...")
    hashTable = HashTable(s)
    with open("words.txt", "r") as ins:
        for line in ins:
            words.append(line)
            hashTable.insert(line,True)
    return hashTable

# Look up each word in the hash table
def lookupWords(table):
    print("looking up all words in dictionary")
    for word in words:
        table.lookup(word)
# Load `words.txt` into a hash table of size s

def loadIntoTable(s):
    print("loading words into table...")
    hashTable = HashTable(s)
    with open("words.txt", "r") as ins:
        for line in ins:
            words.append(line)
            hashTable.insert(line,True)
    return hashTable

# Look up each word in the hash table

def lookupWords(table):
    print("looking up all words in dictionary")
    for word in words:
        table.lookup(word)

Vary s by 10k, 20k, ...
At first, not enough buckets to compensate

Test of `lookupWords`

Later, not much added benefit for extra buckets..
Question

Why not just always use 100k buckets?
**Observation:** An optimal hash table requires knowing a priori the number of elements stored in it

If we’re using less of the hash table than we need, we’re wasting a lot of memory just on the buckets
Enter the trie…

A trie is a **suffix tree** that *compactly* represents sets of strings
Let’s say we want to represent the following set…

\{“Ali”, “Aly”, “Al”\}
Let’s say we want to represent the following set…

\{“ali”, “aly”, “al”\}
Trie shares **common prefixes**. In this example, **blue** nodes indicate that the element is in the set

Get value by traversing down spine…

```plaintext
{“ali”, “aly”, “al”}
```
Trie shares **common prefixes**. In this example, blue nodes indicate that the element is in the set

Get value by traversing down spine…

```
{“ali”, “aly”, “al”}
```
Trie shares common prefixes. In this example, blue nodes indicate that the element is in the set.

Get value by traversing down spine...

Red node = no data at that node

{“ali”, “aly”, “al”}
Draw Example Trie For...

- {“b”, “ba”, “bac”}
- {“alice”, “alicia”, “alejandro”}
- {“alice”, “bob”}
Building Tries

• Let’s say we want to build tries for strings in the English lowercase alphabet only

• I.e., 26 characters

• Obvious problems with this we will fix later (e.g., “José”)
Building Tries

- Insight: represent trie as Node with 26 children buckets

Represent as array of size 26

Lookup-next constant time via random access
class Trie:
    def __init__(self, buckets):
        self.content = False
        self.contents = [None] * buckets
        self.buckets = buckets

    def bucket(self, chr):
        return ord(chr) - ord('a')
As is common in data structures, I’ve just shown one example formulation here, other equivalent ones exist.
Trie Lookup

Let’s lookup aly

- Start at root
- Go to “a” bucket
Trie Lookup

Let's lookup aly

- Start at root
- Go to “a” bucket
- Keep going to l
Trie Lookup

Let’s lookup aly

- Start at root
- Go to “a” bucket
- Keep going to l
- Keep going to y
Trie Lookup

Let’s lookup aly

- Start at root
- Go to “a” bucket
- Keep going to l
- Keep going to y
- Return color == blue
What is running time of lookup?
Trie Lookup

What is running time of lookup?

$O(\text{len(key)})$
Exercise: Write pseudo-code for `lookup`
def lookupHelper(self,string,i,m):
    if (i >= m):
        return self.content
    else:
        bucket = self.contents[self.bucket(string[i])]
        return bucket != None
        and bucket.lookupHelper(string,i+1,m)

def lookup(self,string):
    return self.lookupHelper(string,0,len(string))

Can write slightly-more-optimized version of this with loops…
Exercise: Write pseudo-code for `insert`
```python
def insertHelper(self, string, i, m):
    if (i >= m):
        # Set this bucket to True
        self.content = True
    else:
        if (self.contents[self.bucket(string[i])] == None):
            self.contents[self.bucket(string[i])] = Trie(self.buckets)

        self.contents[self.bucket(string[i])].insertHelper(string, i+1, m)

def insert(self, string):
    self.insertHelper(string, 0, len(string))
```

Construct new child trie if one doesn’t exist!
Question

What would persistent insert look like for tries..?
Binary Tries

- **Insight:** one simple lexicographic order is binary numbers

What set of binary strings does this trie represent?
**Question:** in general, are binary tries a good idea?

Assuming random binary strings: better / worse than binary tree?

Assuming binary strings w/ common prefix?
Because we can treat \textbf{any} alphabet as the binary alphabet with the necessary transformations, binary tries are \textit{always} an option!
Kris speaks extemporaneously about the cache...
Binary Tries: Crummy Cache?

Your computer “caches” recently-used memory

Every time your computer needs to touch memory it hasn’t seen recently it’s slower

Upshot: binary trie causes potentially-lots of memory access
Tries vs. BSTs

- Trie leverages prefix-ordering—e.g., lexicographic
  - E.g., for dictionaries, $O(\text{len(word)})$ rather than $O(\log(\text{words}))$
  - Also, in practice, most words are small, so this is even better!
- Downside: not always useful prefix ordering
  - E.g., storing floating-point numbers in a trie is naïve
- Tries do use more memory when implemented naively!
Bait + Switch

• First motivated tries by saying they saved memory—but tries still allocate lots of potentially-empty buckets

• What gives!?
  • Are tries more “compact” (memory efficient) vs hash tables?
  • Under what circumstances?
Going into the Future

• Tries: useful representing sets w/ dense common-prefixes

• Tries are good adaptive data structures: “resize” automatically
  • (Hash Tbls don’t do this! “Fill up” with data, becomes slow)

• As we implemented them now, not amazing performance:
  • But, basis for many other optimized implementations
• 10000 race participants are assigned numbers from 1 to 10000, but some drop out the day of the race. I would use a (trie / hash table)

• 2000 500-character strings come in over the network and need to be remembered, there is no apparent structure to their contents (trie / hash table)
Sets vs. Maps

- Most of these data structures can be used to store sets or maps (i.e., key/value associations)

- Trivial impl of set from map: just make key True

- To implement map from set (often): add place for value

\{al |-> 23, ali |-> 48, aly |-> 119\}
Sets vs. Maps

Can often get “cheaper” implementation by playing low-level tricks if we know it’s just a set versus a map

(E.g., bitmaps for efficiency)
Deciding Between Them

- **BST**: elements not prefix-oriented, want adaptive memory usage, okay with imperative data structure
  - Self-balancing trees (necessary for $O(\log(\text{elems}))$ behavior) like R/B and AVL are imperative

- **Trie**: prefix-oriented data, want persistence, ok w/ more overhead than BST

- **Hash table**: You know roughly how much data you have (to avoid resizing), keys unordered, don’t need persistence
Still not fast as it could be!

Come back next time for **bloom filters**

(Blazing-fast probabilistic set)
Exercises (for exam)

- Think about how you would implement union / intersection
  - For hash tables
  - For BSTs
  - For Tries

- Which one is best in principle? Which do you suspect would be best in practice?