latex input: mmd-article-header Title: Data Structures Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Data Structures, Algorithms CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2016 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Union Find

Problem we're trying to solve

- Say we have a larger number of elements of type `E`
- Over time, those elements may become part of a singular set of `E`s
    - But they're not all joining the same set
- We would like to efficiently be able to
    1. Combine two sets together
    2. Ask whether two elements are in the same set
    3. Ask how many distinct sets there are
- For example, we may be trying to find the *connected components* of a graph
- With the algorithm described below, we will not be able to *remove* an item from a set. No one knows a good way to pull that off [Sedgewick].

API it provides

- `UnionFind[E](elems: E*)`
- `union(a:E, b:E): Unit` -- merge set containing `a` with set containing `b`
- `numSets: Int` -- the current number of disjoint sets in this UnionFind
- `sameSet(a:E, b:E): Boolean` -- whether or not `a` and `b` have (even *transitively*) been `union`ed together to be in the same set

How union find it works

Each set is stored as a tree of its consituent elements. One element in the set acts as the root. Which one of the elements is the root is determined by the order in which sets are union'ed together, and is not a reflection of any characteristic of the root element itself. Its children are all in the same set. Those children may each have their own children in the same set, etc.

When we initialize the UnionFind, each element is put in its own set. Specifically, we create a Map from element to parent, and to start out, each element's parent is itself. We also maintain an instance variable which states how many sets there are, and it starts out as $||elements||$.

To perform the `sameSet(a:E, b:E): Boolean` operation, we find out whether `a` and `b` are in the same set by walking up their parent pointers to the root node, and seeing if the respective set-roots are the same node. Most- naively implemented, our "tree" would just be a linked-list, and this operation would take ~(elements in the set). However, how long this really takes depends on how the tree is formatted. E.g. if we could establish an invariant that the tree were to always be a balanced binary tree, then its height would be $O(\log n)$, and that's how long it would take to compare the sets to which `a` and `b` belong. If the tree was kept to have (amortized) constant height, then the set- membership comparison would take (amortized) constant time. In fact, the UnionFind algorithm is able to achieve amortized constant height for these set-trees. The way it does this is in how the `union` operation happens.

As hinted above, the most naive `union(a:E, b:E): Unit` implementation would concatenate to the linked-lists denoting the sets to which `a` and `b` belong. To make this faster, after we've walking from `a` and `b` to their roots, we can update all the nodes in the list to have their `parent` pointers point directly to the root. Then to combine the two sets, we make the smaller set's root node point to the larger set's root node. Now we have a strange-looking tree, but it is amortized constant time to traverse. To make this work, we have to track the size of the sets too. We do this by making changing the Map from element to parent (see above) into a Map from element to (parent, set size).

Suffix Tree (for String Search)

4/27/15

From *The Algorithm Design Manual*

A Trie containing all the suffixes of a given set of strings.

E.g. the suffixes of *ACAC* are

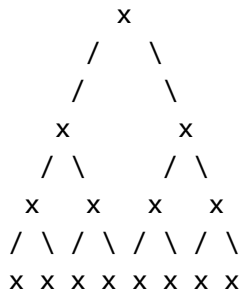
- ACAC
- CAC
- AC
- C

The suffix tree for {*ACAC*, *CACT*} has a root with no value, pointing to a, c, and t nodes, where c and t are noted to be terminal (though may still have children). Then a points to c, which is terminal, but also points to a and t, etc.

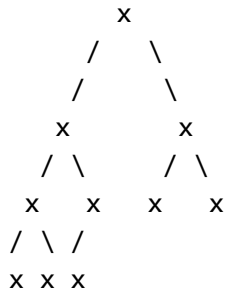
If this runs out space, look up and use a "compressed suffix tree" instead, which always takes up linear space.

Tree

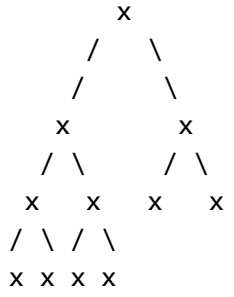
- **Perfect** -- every level is full



- **Complete** -- every level, except possibly the last, is completely filled, and all nodes are as far left as possible



- **Full** -- every node has either two children or zero children



- **Height** -- *distance* from root to deepest leaf
    - So for the above tree examples, the *height* is **3** (*not* 4)

Heap

Overview

- a binary heap is a **complete** binary tree which satisfies the heap ordering property.
- The ordering can be one of two types:
    1. the **min-heap** property: the value of each node is *greater than or equal* to the value of its parent, with the minimum-value element at the root.
    2. the **max-heap** property: same but flipped

- A heap is not a sorted structure but can be regarded as **partially ordered**.
    - There is no particular relationship among nodes on any given level, even among the siblings
- The heap is one maximally efficient implementation of a **priority queue**
- the *parent* of the node at position k in a heap is at position $\frac{k}{2}$
- See implementation with explanation at `~/Dropbox/CSyStuff/PrivateCode/PreDraft_6-27-14`

References

1. Wikipedia
2. Algorithms -- Sedgewick, pg. 316
3. Heapsort Summary Page

Deque -- double-ended queue

Elements can be added/removed/inspected from either the front or the back.

Deque Implementations

- **Doubly-linked-list** -- all required operations are O(1), random access O(n)
- **Growing array** -- *amortized* time is O(1), random access O(1)

Java

1. `LinkedList<T>` -- doubly-linked-list
2. `ArrayDeque<T>` -- growing array

Notes on the implementation java.util.ArrayDeque<E>

- The implementation is based on the use of a "resizable array" instead of e.g. a doubly-linked list
- It is generally faster than `Stack` for stacks and `LinkedList` for queues
- The length of the internal array is always a power of two
- Its size is immediately doubled whenever it becomes full
- Empty elements are guaranteed to always contain `null`
    - This means it must explicitly null-out elements as they are deleted
- The internal array is a field `Object[] elements`
    - Recall, Java does not permit the declaration of a `E[]`
        - However, I believe Scala *would* allow declaring a `Array[E]`
- Fields are maintained to contain the indexes of the `head` and `tail` in `elements`
    - The `head` is the "front/beginning" -- it points to the first element in the array

        - This is where elements are "pushed", "popped", and "removed"
    - The `tail` is the "end" (beautiful friend) -- it points to the element *after* the last one in the array
        - this is where elements are "offered" (i.e. queue-add), and "added"
        - So it is an *invariant* that `element[tail] == null`
    - This means that when adding to the "front", we increment `head` *before adding*, but when adding to the "back", we decrement `tail` *after adding*
- It *is* allowed for `head > tail`, but *iff* it becomes `head == tail`, we `doubleCapacity()` of `elements`
- `elements` is completely *full* iff `head == tail`

Common bits of code

Incrementing head

```
head = (head-1) & (elements.length-1);
```

Since `elements.length` is a power of two, and `head < elements.length`, this will decrement `head`, but if it's already 0, it will wrap around to `elements.length-1`, the last index in the internal array. We couldn't equivalently use the modulo operator instead, because `neg % pos => neg`.

Decrementing tail

```
tail = (tail+1) & (elements.length-1);
```

This is equivalent to `++tail % elements.length`, but probably more efficient for modern processors.

Double capacity

- Create the new array `a` of length `elements.length << 1`
- Copy the elements from `head` of `elements` to the end of `elements` into `a`
- Copy the elements from `0` to `head == tail` into `a` *after* the elements above
- Save `a` as the new `elements` (we don't need to null-out the old `elements` because there are no longer any references to it)
- Update `head = 0` and `tail = oldElements.length`

Delete element at arbitrary index i

- This is *not* an efficient operation, as it inches elements over one slot in the array as necessary
    - It may "inch" the elements forwards *or* backwards, depending on which causes the least overall element motion

- What is the business with `front` and `back`?
  - What this does is
    - `if i >= head => front = i - head`
      - In this case, we will move elements from `head` to `i` over to the *right* by 1
    - `else i < head => front = len - (head-i)`
  - The goal is as follows
    - Let
      - `elements.length = 16`
      - `head = 14`
      - `tail = 8`
      - `i = 2` -- this is the element we want to delete
    - So we have `null`s from `tail until head`
    - Sure, we *could* move `(i+1) until tail` to the *left* by 1
      1. That would require a single arraycopy of 5 elements
    - More efficiently, we could
      1. Move `0 until i` to the *right* by 1
      2. Move `elements[0] = elements[elements.length-1]`
      3. Move `head to (elements.length-2)` to the *right* by 1
    - That would require two arraycopies and an assignment, for a total of (step 1) 2 + (step 2) 1 + (step 3) 1 = 4 elements, which is less than the 5 moves from the simpler above
      - This difference grows as `i` gets smaller and `head` gets bigger

Bloom Filters

i-programmer

The Point

- **Bloom filters *attempt* to tell you if a particular data item is a repeat**
- **False-positives** are ***possible***, **false-negatives** are ***not***

Approximate answers are faster

- As you already knew, you can usually trade space for time; e.g. use more storage, to use less processing time
- What you may not have known, is that quite often, you can also trade certainty for time in such a way that the approximation degrades in a predictable fashion, so that the approximate result is still useful.

Applications

- *Google's BigTable database* uses one to reduce lookups for data rows that haven't been stored.
- The *Squid proxy server* uses one to avoid looking up things that aren't in the cache

History

- Invented in 1970 by Burton *Bloom*

How it works

Executive summary: **Uses multiple different hash functions in-concert**.

Initialization

Initialize a **bit array** `Bloom[i]` to zeros.

Insertion

- Simply compute *k* different hash functions over the item to insert
- For the resulting *k* values, set the bit array at those indices to `1`

Lookup

- Given an item to look up, apply the *k* hash functions and look up the indicated array elements.
- If *any* of them are zero you can be 100% *sure* that you have never encountered the item before.
- *However*, even if *all* of them are one then you can't conclude that you have seen the data item before.
    - Instead, all you can conclude is that it is *likely* that you have encountered the data item before.

Removal

- **It is impossible to remove an item from a Bloom filter.**

Characteristics

- As the bit array fills up the probability of a false positive increases
- There is a formula for calculating the optimal number of hash functions to use for a given size of the bit array and the number of items you plan to store in there

$$k_{opt} = 0.7\frac{m}{n}$$

- There is another formula for calculating the size to use for a given desired probability of error and fixed number of items using the optimal number of hash functions from above

$$m = -2n \cdot \ln p$$

Red Black Tree

Tim Roughgarden's Coursera lecture on it

- A form of *balanced* binary search tree with additional imposed **invariants** to ensure that all the common operations (insert, delete, min, max, pred, succ, search) happen in $O(\log n)$ time.

Invariants

1. Each node is red or black
2. Root is black
3. A red node must have black children
4. Every `root->NULL` path through the tree has same number of black nodes

Theorem: every red-black tree with *n* nodes has

$$height \leq 2 \log_2 (n + 1)$$

Implementations

This is what backs Java's `TreeMap<T>`

B-Tree

Intro

We want to be able to search huge data sets that don't fit in RAM. So we want an index of it so we can search quickly. Both data and index should be stored efficiently in *pages* on disk. But if the entire index doesn't fit on a *page*, we'll need an index for the index (i.e. another level on the tree) [etc. if necessary]. In this scenario, the first access of the data in a page and writing modified pages back to disk is practically all the cost associated with doing *anything* with the data on that page, so we're just trying to minimize those two (read and write) operations.

Invariants / Structure / Properties

Choose an **"order"** $t \geq 2$. It seems everyone defines the 'order' differently, and this is very confusing. I'm going with the definition in *Cormen et al.* because the whole B-Tree is very thoroughly and clearly enunciated there. The Wikipedia page is pretty

good too though, but uses Knuth's (different) definition of 'order'. Cormen calls this version the "minimum degree" of the tree, and surely that is less ambiguous than calling it the 'order'.

1. All leaves have the same depth (viz. the tree's height)
2. Each node has *at most* $2t - 1$ keys
3. Each node has *at least* $t - 1$ keys (except the root)
4. The root has *at least* 1 key
5. Key's are stored in *non-decreasing* order
6. Each internal node has $numKeys + 1$ pointers to child-nodes
7. For an internal node, in between two keys, you find a pointer to another node closer to accessing the data between those two key values

E.g. when $t = 2$

- Each internal node has $t - 1 \leq x \leq 2t - 1$ keys and $t \leq x \leq 2t$ children
    - I.e. $[1, 3]$ keys and $[2, 4]$ children

Implementation

Split full nodes on the way down

One nice trick from Cormen prevents us from getting into a situation where to insert an item we must break up this node, but that will require us to break the parent node, etc. and we feel very anxious about getting it all right. Instead, upon searching for the node to `add` into, we `split` each *full* node on the way down the tree, so that a parent node is *never* full and we can always insert into it by splitting if we have to.

Splitting a node (on add())

```
def split(nonfullParent, fullChild) {
    median = medianElement(child)
    medIdx = insertElement(median, parent)
    newLeft = leftOf(median, child)
    newRight = rightOf(median, child)
    parent.addLink(medIdx, newLeft)
    parent.addLink(medIdx+1, newRight)
}
```

Deleting a key

When we delete a key from an internal node, we have to rearrange the children. When a node gets too small during deletion, we must *pull up* a member of the child below. But now that child might be too small (uh oh, etc.).

I'm thinking I'll just implement the sketch presented in prose in Cormen as an exercise. Maybe I should start by churning it into pseudocode.

Pseudocode

```
def delete(Key k, Node xNode) {
    // cases 1-2
    if (k in xNode) {

        // case 1
        if (xNode is Leaf) {
            remove k from xNode
        }

        // case 2
        else /* xNode is Internal */ {
            Node yChild = getChild( idxOfKey(k) )
            Node zChild = getChild( idxOfKey(k) + 1 )

            // case 2.a
            if (yChild.numKeys ≥ t) {
                find predecessor k' of k and delete it
                    (should req're only a single downward pass)
                keys[idxOfKey(k)] = k'
            }

            // case 2.b
            else if (zChild.numKeys ≥ t) {
                find successor k' of k and delete it
                    (should req're only a single downward pass)
                keys[idxOfKey(k)] = k'
            }

            // case 2.c
            else {
                merge k and zChild into yChild
                remove both k and pointer to zChild from xNode
                free(zChild)
                delete(k, yChild) // recursive call
            }
        }
    }

    // case 3
    else /* k not in xNode */ {
        int idxForDescent = idxOfNodeFor(k)
        Node descNode = getChild(idxForDescent)

        // case 3.a-b
        if (descNode.numKeys == t-1) {

            // case 3.a
            if (descNode has eitherSibling with ≥ t keys) {
                // the goal here is we're making sure that we can,
                // (if necc., [we don't know yet,])
                // remove a <key,node> from descNode
                // so we do something like a "rotation"
```

```
                move a key from xNode into descNode
                move a key from theSibling into xNode
                move the childPointer from theSibling into descNode
            }

            // case 3.b
            else { // merge
                Node mergedNode = combinedFrom(descNode, eitherSibl
                move a key from xNode into mergedNode // becomes me
            }
        }

        delete(k, descNode) // recursive call
    }
}
```

SkipList

- Allows fast search within an ordered sequence of elements
- "Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space." --- inventor William Pugh
    - $\log n$ for contains, insert, and remove
- a "data structure for storing a sorted list of items, using a hierarchy of linked lists that connect increasingly sparse subsequences of the items."
- It's hard to explain but the picture on Wikipedia makes it clear
- So you walk down the highest (sparsest) list until you find that you've skipped your element
    - If you found your element, you're good to go
- Then you go to the most recent element before you skipped over the element, and walk down a lower list
- The last list contains your entire sequence and you'll definitely find your element there

Hash Tables

- Generally it is a "map", i.e. a "partial function" from a key-space `T` to a value-space `U`
    - It is a **map** in the sense that you present it a key, and if that key is known, it returns to you the value that was previously associated with that key
    - It is a **partial function** in the sense that each key maps to at most one value (but may also not map to anything)
- You can convert it to function as a "set" by storing `null` elements as values
- The keys are stored in an *array* of "**buckets**"
- There is a *prehash* function that maps a key to a hash-space (integers)

- To *hash* a key, you compute $prehash(key)$
    - This tells you which bucket-index to insert the key at
- A **hash collision** occurs when you go to insert a key into a bucket, but there is already a *different* key in there that mapped to the same bucket
    - The number of buckets is smaller than the space of `int`s, so collisions occur because of the *pigeonhole principle*
- How *collisions* are resolved is up to your hash table implementation

Open addressing a.k.a. closed hashing a.k.a. (e.g. linear) probing

This is a method of collision resolution, where if you go to put a key into a bucket, but there's already something in there, you keep scooting along to try other buckets until

1. you find that the key is already in there
2. an empty array slot is found (which means the key is *not* already in there)

Linear probing

An *open addressing* method in which the way you "scoot" along to try another bucket is by skipping forward a static number of buckets (e.g. 1).

Hopscotch table

- In a hopscotch table, the "scheme" for inserting elements for which there is already a key is *open addressing* (see above)
- This particular implementation is good for use in a *concurrent* environment
- It is also good for high table load factors (e.g. over 0.9)
- The concept was introduced by M. **Herlihy** et al. in 2008.
- The goal is to assign buckets to a "*neighborhood*" such that buckets within a neighborhood are all within the same *cache line*
- A bucket's neighborhood is its actual bucket index, and then the next $H - 1$ bucket indexes
- When a neighborhood becomes full, the table must be resized
- Each key *must* go into at least the neighborhood of its actual bucket
- Info about which buckets in a neighborhood are actually occupied is stored in a bitmap, to reduce search time through the bucket
- If you're trying to insert an item, and there's already something in the bucket
    1. Run a linear probe to find the first empty slot (doesn't have to be within the right neighborhood)
    2. If the first empty slot index *j is* in the right neighborhood, stick the key into *j*
    3. Otherwise, try to find something from the desireable neighborhood that can be validly moved *into j*, and then stick the new item into the moved item's old slot

4. This process's implementation utilizes the neighborhood bitmaps mentioned above, may need to be performed recursively to finally free up a spot in the desireable neighborhood

Distributed Hash Table (DHT)

1. Same interface as a *hash table* (look up *value* by *key*)
2. Responsibility for maintaining the mapping $keys \rightarrow values$ is *distributed* among the *nodes*
3. We require change in who is participating to cause minimal disruption
4. Useful for web caching, distributed file systems, DNS, IM, multicast, P2P (e.g. BitTorrent's distributed "tracker"), content distribution, and search engines
5. Properties
    1. decentralization/autonomy --- no central coordinator
    2. fault-tolerance --- nodes can continuously join, leave, or fail
    3. scalability --- still functions efficiently with millions of nodes
6. Generally each node must coordinate with $O(\log n)$ other nodes
7. Can be optionally designed for better security against malicious participants, and to allow participants to remain anonymous
8. Handles load balancing, data integrity, and performance

Structure

1. We start with a *keyspace* and a defined *partitioning scheme*
2. To add a new entry
    1. Hash it
    2. Send it to *any* participating node
    3. Keep *forwarding* it until the *single* responsible node is reached
    4. The responsible adds the entry
3. Getting an entry is quite similar
4. Uses **consistent hashing** --- has property that when the table is resized, only $\frac{keys}{slots}$ keys must be remapped.
    1. The hashing techniques make it so that only those members adjacent in the keyspace to a new node have to have they're data sloshed around
5. The **Overlay network** is the set of links connecting nodes
    1. Requires the property that for any key, each node either owns it, or has a link to someone "closer" to it in terms of some defined keyspace distance
    2. There's a tradeoff between the number of links we require each node to have ("degree") and the "route length" queries require

HyperLogLog

- The goal is to efficiently approximate and track set cardinality

- This structure also allows you to *union* two HyperLogLogs and get the approximate cardinality of the union
- It hashes each entry in multiple ways, and only stores the hashed value
- In the expectation, if we added n random elements to a set, the minimum element would have expected value $\frac{1}{n+1}$.
    - So by hashing in a lot of ways, and doing some mathematical statistics, one can look at the minimum value in the case of each of those hashes and use that to estimate the true number of elements in the set.

Count Min Sketch

- The goal is to get approximate counts of the most popular items in a list
- We hash each item in multiple ways, and increment the counter associated with each hash's corresponding bucket
- If some uncommon items hashed to the same bucket sometimes, it probably won't have enough impact to matter
    - But to be extra-careful, we take the bucket which has the minimum count because that means it had the least collisions overal
- Rare items are deleted from the counter, and there is a mechanism for them to be re-added later on with an approximately-correct value if they become popular

References

- https://www.mapr.com/blog/some-important-streaming-algorithms-you-should-know-about