Monday, March 7

Regular Expressions (regex/regexp): series of chars that defines a search pattern

- Used by databases, websites, vim, and command-line like grep/awk
- Lexical analysis in a compiler

Basic syntax

- char the char itself
- . matches any char
- char+ one or more occurrences of char
- char* zero or more occurrences of char
- char? one or zero occurrences of char
- [chars] any char in brackets
- [a-z] any lower case a-z
- (chars) group of chars
- {num} num of times char matched

Theory of finite automata: the theory behind regex

Types of automata: see slide

- How are automata theory and regular expressions connected?
    - Finite automata: abstract machines recognize patterns such as in strings
    - Regular expressions: generate patterns of strings, an algebraic formula

Deterministic finite automata (DFA)

- Math model of a machine that accepts particular set of words over some alphabet
- Set of words is called the language
- Given string will always produce the same result
- A DFA quintuple
    - E- input alphabet
    - S- finite nonempty set of states
    - S0- start state
    - D- state transition function
    - F- set of final states
- Letters and strings
    - a exists in E, call a a letter
    - x exists in E*, call x a string
    - Lambda exists in E0, empty string
    - ax is a string
- Example: regex “int” input int
    - See slides

Nondeterministic Finite Automata (NFA)

- Quintuple same as DFA
- State transitions are not uniquely determined by current input
- Can be written as a DFA, exponentially large
- Must guess in the beginning, backtrack if you go down wrong path
- Backtracking
    - Expensive recursive operation
    - Must exmplore all paths (worst case)

        - Num of paths can grow exponentially
    - Simulate multiple states
        - State transition function map to reachable states
        - Remember all the states that can be reached for some string
        - Worst case it will be in every state
        - Linear time
        - Algorithms matter

The regex package

Wednesday, March 9

Hashing and Bloom Filters

Hash: take key (string) and make a number from it (that number is uniformly distributed - each thing equally probable)

Use data structures

- Arrays and linked lists
    - Linear search: lookup in unsorted array take O(n)
    - Binary seach: lookups in sorted array take O(logn)
    - Linked list lookup takes O(n), insert time is O(1)
    - Balanced tree seach insert delete is all O(logn)
- Direct address table is a key-value store where the key is the index to the value
    - Random access array: search in O(1)

Hashing

- O(1) search time on avg O(n) in worst case when they collide
- Trading space for time
- Really big HT is called sparse \

Hash Tables

- Unordered key-value pairs
- Offers efficient lookup, insert, delete

Hash functions

- Hash key to degenerate indices for each value
- Good hash function
    - Uses all input data
    - Fast computation
    - Minimizes hash collisions (uniform dist and similar strings have very diff hash values)
    - Half bits have to change??
- Types
    - Division based
    - Multiplication nased
    - String-valued keys

Division based

- Not the best

String value keys

- char->int->int%table size

- Can’t just add/multiply
- Not easy to find a good one

Load factor
- Lambda
- As it goes to zero probability of collision goes to zero
- Num slots filled out of the whole thing ex. 6/20

Birthday paradox
- Sharing a birthday is a hash collision

Hash collisions
- Hash of two keys is the same
- Is num of keys is greater than num of hash values, then collisions are unavoidable

Tackling collisions

Linear probing
- Move sequentially in slots till you find an empty one
- Try to store at location where index = (i+1)%SIZE then (i+2)%SIZE
- Go until the value itself is found or there is an empy slot
- Not great for deleting

Quadratic probing
- Square the seq of hash values when looking for next
- (i*i)%SIZE then ((i+1)*(i+1))%SIZE
- HT must have size prime number
- Never more than half full

Double hashing
- Multiple hash functions
- Load factor lambda = n / size T

Chaining with linked lists
- Each slot is a pointer to singly linked list of key-values with same hash value
- Lookup searches list to find matching key
- Load factor lambda xxxx
- Time complexities xxxxxxxx

Open addressing vs chaining
- OA is computationally intense
- Dynamic: chaining never fills up
- Chaining less sensitive to load factor and hash funtion
- Cache performance
    - Chaining: bad bc linked lists
    - OA, good bc directly in same table
- Space
    - Chaining: uses extra space for linkes and parts of HT never used
    - OA: slot can be used even without input mapped to it

Find keys
- Keep count to see if at end
- Check keys for value

Insert key

- Find empty spot

Bloom Filters

- Data struct to tell whether an element is in a set
- Hash table structured like a bit array
- Add and search no delete
- If the BF is set to 0 then you know for sure it is not there, if it is 1 it could be there (could have been set by a diff value with that same hash)
    - Possibly set or def not set
- Fast and mem efficient
- Filter query

False positives

- The bit is 1 but it wasn't because of the value you are checking rn it was from something before
- There is math for the probability of this

Time and space complexity

- Xxxxxxx

Cryptographic hash

- Large thing reduce with hash and then encrypt decrypt
- Message integrity
- Digital signatures
- Following properties
    - Deterministic: same message has same hash
    - Quick to computer
    - Not rly possible to find two diff messages with same hash
    - Avalanche effect: change message one bit, hash changes half the bits

Friday, March 11

Multithreading

Processes and threads

- Process
    - code, data and stack
    - Program state: cpu registers, program counter, stack pointer
    - Only one can run on a single cpu core at any given time (multi-core cpus czn be multiple processes)
- Process contains one or more threads
    - Threads within a process run concurrently
    - Threads share mem with each other
- Items in each
    - Process
        - Address space
        - Open files
        - Child processes
        - Signals and handlers
        - Accounting info

        - Global variables
    - Thread
        - Program counter
        - Register values
        - Stack
        - Stack pointer
        - Local variables

Address space

- Program executes code, instructions have addresses
- Programs access data, each byte of data has an address
- Program thinks it is the only one working (abstraction)
- Address space is the region where it executes

Single vs multi-threaded address spaces

- Threads enhance processes not replace them

Why use threads

- Faster to create or destroy - no separate address space
- Allow single application to do lots of things = keep working during io wait
- Context switching
    - Each thread gets to issue its own ios
    - More ios can be outstanding/pending
- Sharing mem
    - Threads share mem unlike processes → easier to share data

Basic posix thread api

-