CIS 313

week of Jan 28

fourth week of the term
hash tables

• chapter 11
• we want to manage a dynamic set $K (|K|=n)$ where each element has a key in universe $U = \{0,1,...,m-1\}$
• support efficient operations SEARCH, INSERT and DELETE (i.e., in $O(1)$)
• if $m$ is small, an array $T[0,...,m-1]$ would suffice
• each slot in $T$ corresponds to a key in the universe
• if the set doesn’t contain key $k$, then $T[k] = \text{NIL}$. 
hash tables

• if $|U|$ is large, an array of size $|U|$ might be impractical/impossible
• idea: the number of keys actually used $n$ might be much smaller than $|U|$
• we can thus reduce the storage requirement while still achieving the efficiency
• hash table: store $n$ items of $K$ in a table $T$ of size $m$ ($m \ll |U|$)
• hash function $h$ determines where to put an item ($h: U \rightarrow \{0,1,\ldots,m-1\}$)
• issues
  • what to do when two items hash to same location (collision)
  • how to choose good hash function $h$ (minimize collisions)
  • how to choose table size $m$
  • dynamically increase table size
    • important in databases but not addressed here
collision resolution

• what to do with two items x and y that hash to same location?
• $h(x.\text{key}) = h(y.\text{key})$

• open addressing
  • look at other locations in the table
  • table might overflow
  • more complicated

• closed addressing
  • all items that hash to location t stay there in some structure
  • bucket, linked list, ...
chaining

- first: simple version of chaining
- table T with m slots, each containing a linked list
- hash function h maps keys to \{0, 1, ..., m-1\}
- INSERT(T, x): put x in a node at the head of T[h(x.key)]
- SEARCH(T,k): search for an item with key k in the list T[h(k)]
- DELETE(T,x): delete x from the list T[h(x.key)] (done in O(1) with doubly linked list)
- load factor: \( \alpha = n/m \), where n is the number of items in the set.
- simple uniform hashing (ideal): search time is \( 1 + \Theta(\alpha) \) (average-case)
- also called *closed addressing* (since item stored at that location)
choosing a hash function

• let k be the key and T a table of size m
• want h(k) to distribute keys uniformly across locations \{0,1,...,m-1\} (i.e., approximate the simple uniform hashing)
• division method: \( h(k) = k \mod m \)
  • choice of table size m important
  • if m=2^p, then only low order bits of k matter (poor choice)
  • if k not distributed well, then h(k) prone to be biased
  • best if m a prime
multiplication method

• pick constant $A$ with $0 < A < 1$

• $h(k) = \lfloor m \cdot ((k \cdot A) \mod 1) \rfloor$ (here “mod 1” means fractional part of real number)

• Knuth suggests $A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339 \ldots$

• nice example on p 264 of text
universal hashing

- problem with fixed hash function: all keys might hash to same slot
- universal hashing: family of hash functions $\mathcal{H}$, maps key universe $U$ onto $\{0, 1, ..., m-1\}$
- remark: no single input will always exhibit worst-case behavior (good average-case performance)
- want for any $k, l \in U$ that the number of $h \in \mathcal{H}$ such that $h(k) = h(l)$ is at most $\frac{|\mathcal{H}|}{m}$ (universal hashing)
- idea is to pick an $h \in \mathcal{H}$ randomly if possible
- intuitively if keys $k \in U$ not distributed well a random $h \in \mathcal{H}$ will still distribute the locations well and excess avoid collision
- example family: $\mathcal{H}$ will depend on fixed $p, m$
  - $m$ is table size, $p>m$ is a prime so that all keys $k<p$
  - choose $a, b$ with $0<a<p$, $b<p$ (randomly)
  - $h(k) = ((ak+b) \mod p) \mod m$
  - proof that $\mathcal{H}$ is universal in text, depending on basic number theory (nice proof)
back to collision resolution: open addressing

- instead of using lists in chaining, all elements are stored in the hash table, so no storage requirement for points, saving spaces to reduce collisions
- for key k = x.key, if location T[h(k)] is full (via collision), need to put x in a different location
- look in a sequence of locations depending on k. This is called the probe sequence
- using the hash function h<k,i> to determine the slot to probe at time i on key k
- look in locations h<k,0>, h<k,1>, h<k,2>, ... until find empty slot in which to place x
- requirement: for every key k, (h<k,0>, h<k,1>, ..., h<k,m-1>) be a permutation of (0,1,...,m-1) so every position of the hash table is considered eventually
strategies for probe sequences

• simplest (and worst): *linear probing*
  • \( h_{<k,i>} = (h(k) + i) \mod m \)
  • that is, if \( h(k) \) is full, look in locations \( h(k)+1, h(k)+2, h(k)+3, \ldots \)
  • problem: primary clustering (slots are clustered in long lines)

• quadratic probing
  • pick constants \( c, d \)
  • \( h_{<k,i>} = (h(k) + c*i + d*i^2) \mod m \)
  • \( c, d, m \) need to be chosen carefully so that \( h_{<k,i>} \) can probe entire table
  • problem: secondary clustering (milder than primary clustering)

• double hashing (the current best one)
  • use two hash functions \( h_1, h_2 \)
  • \( h_{<k,i>} = (h_1(k) + i*h_2(k)) \mod m \)
  • need \( m \) and \( h_2(k) \) to be relatively prime
other uses of hash functions

• database indexing
  • need extendible hash tables as many insertions happen
  • not good for range queries (“find all values between a and b”)
  • B-tree indexes more popular

• cryptographically secure hashing
  • password files
  • multi-party communication
  • hash functions very different looking

• Bloom filters, count-min sketch
count-min sketch

- problem: count events in a data-stream, many possible events ($n$ large), want number of occurrences of each event
- conventional data structure too large
- count-min sketch is probabilistic structure, uses sub-linear space
- idea: table size of $w$ columns and $d$ rows
- each row $j$ associates with hash function $h_j$ mapping to $\{0,1,...,w-1\}$
- when event $e$ occurs, increment location $[j, h_j(e)]$
- estimate of number of occurrences of $a$ is the min of all locations $[j, h_j(e)]$
count-min sketch (cont’d)

Algorithm 1 Count-Min Sketch

insert(x):
    for $i = 1$ to $d$ do
        $M[i, h_i(x)] \leftarrow M[i, h_i(x)] + 1$
    end for

query(x):
    $c = \min \{M[i, h_i(x)] \text{ for all } 1 \leq i \leq d\}$

return $c$
properties of count-min

- use $O(1)$ in both time and space (no new memory allocation when (many) events are added
- good for parallelization
- never underestimate the numbers of occurred events
binary search trees

• chapter 12
• we will look at
  • definitions
  • properties
  • operations: insert, delete, search
  • traversals: inorder, postorder, preorder, level order
  • worst case behavior
  • average case behavior
• then move onto self-balancing BSTs: red-black, 2-3, 2-3-4, ...
various trees

• free tree
• rooted tree
• ordered tree
• binary tree
• binary search tree
  • (search property) let x be a node in a BST. If y is a node in the left subtree of x, then y.key <= x.key. If y is in the right subtree of x, then y.key >= x.key
assorted facts and definitions

• any tree with n nodes has n-1 edges
• a binary tree with left/right pointers and n nodes has n+1 null pointers
• a full binary tree with n internal nodes has n+1 external nodes
• full binary tree: all nodes have either 2 children (the internal nodes) or 0 children (external)
• a binary tree of n nodes has height at least \( \lg n \) and at most n-1
• height = distance of node from bottom, depth = distance from top
facts, defs cont’d

• internal path length (I): sum of the depths of all the nodes
• external path length (E): sum of the depths of the nulls (externals)
• fact: E=I+2n  (nice exercise)
• I corresponds to successful search in BST, average search time is 1+ I/n
• E corresponds to unsuccessful search, average failed search time is E/(n+1)
• worst case tree: skew tree (every node has just one child)
sample BST
BST operations

• **find(x)**
• **insert(x):** find a null and put it there
• **successor(x)**
  • successor(10)=11, successor(15)=17
  • algorithm?
    • if x has right child, go right once, then left until end
    • otherwise, follow parent links until “right” turn

• **delete(x): how?**
  • if 0 children, remove
  • if 1 child, splice out
  • if 2 children, replace with successor value, then remove successor node
walks

- inorder
  - 1 3 4 5 7 8 9 10 11 12 13 15 17 18 20 23

- preorder
  - 12 10 5 3 1 4 8 7 9 11 17 13 15 20 18 23

- postorder
  - 1 4 3 7 9 8 5 11 10 15 13 18 23 20 17 12
randomly built BST

• we have n values and will insert them one-by-one into a BST
• what will that BST look like?
• there are n! permutations of the input
  • we assume each one equally likely
• how many BST shapes can there be?
  • Catalan number, which is \( \frac{1}{n+1} \binom{2n}{n} = \Omega\left(\frac{4^n}{n^2}\right) \)
  • (hard!)
counting permutations for a tree

• given a tree shape T we can determine the number of permutations which, if inserted into empty BST, would end up with that tree
• build up number bottom up
• at node x, suppose left subtree of x has n nodes and is generated by r permutations, and
• right subtree has m nodes and is generated by s permutations
• the subtree rooted at x
  • has n+m+1 nodes
  • is generated by $\binom{n+m}{n} \cdot r \cdot s$ permutations
example

- left side generated by 1 permutation: 13 15
- right side by two
  - 20 18 23
  - 20 23 18
- for full tree, pick one permutation each for the left and right sides
- permutation for the whole tree must start with 17 followed by n+m = 2+3 = 5 spaces
  - 17 __ __ __ __ __
- choose two for them for the left tree, which can be done in \( \binom{5}{2} = 10 \) ways
- example: 2\textsuperscript{nd} and 5\textsuperscript{th} positions
  - 17 __ 13 __ __ 15
- either of the two remaining perms can go in remaining three slots
  - 17 20 13 18 23 15
  - 17 20 13 23 18 15
- total number of permutations for whole tree:
  \[ 1 \cdot 2 \cdot \binom{5}{2} = 20 \]
back to sorting theme

• we can build an abstract sort method based on BST
• given unsorted list, insert all values into empty BST
• perform inorder walk

BST SORT
** input list a=(a₁,a₂,...,aₙ)
create BST T

for i=1 to n
    T.insert(aᵢ)

perform T.inorder
    when visiting a node, store value in list b

return b

dthis part is O(n)
expected behavior

• if list a is chosen randomly from among all n! permutations
• how long does “for i=1 to n T.insert(a_i)” take?
• worst case: O(n^2)
• want to argue: on average O(n lg n)

• main fact: expected search time (1+1/n) in BST built from randomly chosen permutation is \(2 \cdot \ln(n + 1) + O(1) \approx 1.38 \log_2 n + O(1)\)
describe a binary search tree on $n$ nodes such that the average depth of a node in the tree is $\Theta(\lg n)$ but the height of the tree is $\omega(\lg n)$