Lecture 4: Hashing

Review

Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$

- **Idea!** Want faster search and dynamic operations. Can we `find(k)` faster than $\Theta(\log n)$?
- Answer is no (lower bound)! (But actually, yes...!?)

Comparison Model

- In this model, assume algorithm can only differentiate items via comparisons
- **Comparable items**: black boxes only supporting comparisons between pairs
- Comparisons are $<, \leq, >, \geq, =, \neq$, outputs are binary: True or False
- **Goal:** Store a set of n comparable items, support `find(k)` operation
- Running time is **lower bounded** by # comparisons performed, so count comparisons!

Decision Tree

- Any algorithm can be viewed as a **decision tree** of operations performed
- An internal node represents a **binary comparison**, branching either True or False
- For a comparison algorithm, the decision tree is binary (draw example)
- A leaf represents algorithm termination, resulting in an algorithm **output**
- A **root-to-leaf path** represents an **execution of the algorithm** on some input
- Need at least one leaf for each **algorithm output**, so search requires $\geq n + 1$ leaves

Comparison Search Lower Bound

- What is worst-case running time of a comparison search algorithm?
- running time $\geq$ # comparisons $\geq$ max length of any root-to-leaf path $\geq$ height of tree
- What is minimum height of any binary tree on $\geq n$ nodes?
- Minimum height when binary tree is complete (all rows full except last)
- Height $\geq \lceil \lg(n+1) \rceil - 1 = \Omega(\log n)$, so running time of any comparison sort is $\Omega(\log n)$
- Sorted arrays achieve this bound! Yay!
- More generally, height of tree with $\Theta(n)$ leaves and max branching factor b is $\Omega(\log_b n)$
- To get faster, need an operation that allows super-constant $\omega(1)$ branching factor. How??

Direct Access Array

- Exploit Word-RAM $O(1)$ time random access indexing! Linear branching factor!
- **Idea!** Give item **unique** integer key k in $\{0, \ldots, u-1\}$, store item in an array at index k
- Associate a meaning with each index of array
- If keys fit in a machine word, i.e. $u \leq 2^w$, worst-case $O(1)$ find/dynamic operations! Yay!
- 6.006: assume input numbers/strings fit in a word, unless length explicitly parameterized
- Anything in computer memory is a binary integer, or use (static) 64-bit address in memory
- But space $O(u)$, so really bad if $n \ll u$... :(
- **Example:** if keys are ten-letter names, for one bit per name, requires $26^{10} \approx 17.6$ TB space
- How can we use less space?

Hashing

- **Idea!** If $n \ll u$, map keys to a smaller range $m = \Theta(n)$ and use smaller direct access array
- **Hash function**: $h(k) : \{0, \ldots, u-1\} \to \{0, \ldots, m-1\}$ (also hash map)
- Direct access array called **hash table**, $h(k)$ called the **hash** of key k
- If $m \ll u$, no hash function is injective by pigeonhole principle

- Always exists keys a, b such that $h(a) = h(b) \to$ **Collision**! :(
- Can't store both items at same index, so where to store? Either:
  - store somewhere else in the array (**open addressing**)
    - complicated analysis, but common and practical
  - store in another data structure supporting dynamic set interface (**chaining**)

Chaining

- **Idea!** Store collisions in another data structure (a chain)
- If keys roughly evenly distributed over indices, chain size is $n/m = n/\Omega(n) = O(1)$!
- If chain has $O(1)$ size, all operations take $O(1)$ time! Yay!
- If not, many items may map to same location, e.g. $h(k) = \text{constant}$, chain size is $\Theta(n)$:(
- Need good hash function! So what's a good hash function?

Hash Functions

Division (bad): $h(k) = (k \bmod m)$

- Heuristic, good when keys are uniformly distributed!
- m should avoid symmetries of the stored keys
- Large primes far from powers of 2 and 10 can be reasonable
- Python uses a version of this with some additional mixing
- If $u \gg n$, every hash function will have some input set that will a create $O(n)$ size chain
- **Idea!** Don't use a fixed hash function! Choose one randomly (but carefully)!

Universal (good, theoretically): $h_{ab}(k) = (((ak + b) \bmod p) \bmod m)$

- Hash Family $\mathcal{H}(p, m) = \{h_{ab} \mid a, b \in \{0, \ldots, p-1\} \text{ and } a \neq 0\}$
- Parameterized by a fixed prime $p > u$, with a and b chosen from range $\{0, \ldots, p-1\}$
- $\mathcal{H}$ is a **Universal** family: $\Pr_{h \in \mathcal{H}}\{h(k_i) = h(k_j)\} \leq 1/m \quad \forall k_i \neq k_j \in \{0, \ldots, u-1\}$
- Why is universality useful? Implies short chain lengths! (in expectation)
- X_{ij} indicator random variable over $h \in \mathcal{H}$: $X_{ij} = 1$ if $h(k_i) = h(k_j)$, $X_{ij} = 0$ otherwise
- Size of chain at index $h(k_i)$ is random variable $X_i = \sum_j X_{ij}$
- Expected size of chain at index $h(k_i)$

$$\begin{aligned}
\mathop{\mathbb{E}}_{h \in \mathcal{H}}\{X_i\} = \mathop{\mathbb{E}}_{h \in \mathcal{H}}\left\{\sum_j X_{ij}\right\} = \sum_j \mathop{\mathbb{E}}_{h \in \mathcal{H}}\{X_{ij}\} &= 1 + \sum_{j \neq i} \mathop{\mathbb{E}}_{h \in \mathcal{H}}\{X_{ij}\} \\
&= 1 + \sum_{j \neq i} (1) \Pr_{h \in \mathcal{H}}\{h(k_i) = h(k_j)\} + (0) \Pr_{h \in \mathcal{H}}\{h(k_i) \neq h(k_j)\} \\
&\leq 1 + \sum_{j \neq i} 1/m = 1 + (n-1)/m
\end{aligned}$$

- Since $m = \Omega(n)$, load factor $\alpha = n/m = O(1)$, so $O(1)$ **in expectation**!

Dynamic

- If n/m far from 1, rebuild with new randomly chosen hash function for new size m
- Same analysis as dynamic arrays, cost can be **amortized** over many dynamic operations
- So a hash table can implement dynamic set operations in expected amortized $O(1)$ time! :)

Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access Array	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n