Lecture 8: Direct Access & Hashing

Review

Set Interface	Operation, Worst Case $O(\cdot)$						
	Set	Dynamic (D)		Order (O)		D + O	Space
Data Structure Implementation	`find(k)`	`insert(x)`	`delete(k)`	`find_next(k)`	`find_max()`	`delete_max()`	$\sim \times n$
Unsorted Array	n	n	n	n	n	n	1
Linked List	n	1	n	n	n	n	3
Dynamic Array	n	$1_{(a)}$	n	n	n	n	4
Sorted Array	$\lg n$	n	n	$\lg n$	1	n	1
Max-Heap	n	$\lg n_{(a)}$	n	n	1	$\lg n$	1
Balanced BST	$\lg n$	$\lg n$	$\lg n$	$\lg n$	(1)	$\lg n$	5

- AVL Trees have $O(\log n)$ performance across operations
- Really good! Recall $\log n \leq w \leq 64$ for most computers, but still not constant
- **Idea!** Search is a very common operation. Can we `find(k)` better than $O(\log n)$?
- Answer is no (lower bound)! (But sometimes, yes...!?)

Comparison Model

- Can only differentiate between items via comparisons
- Comparisons are $<, \leq, >, \geq, =, \neq$, outputs are binary: True or False
- **Comparable items**: black boxes only supporting comparisons between pairs
- **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
- For a comparison algorithm, the decision tree is binary **(draw example)**
- An internal node represents a **binary comparison**, branching either True or False
- 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?
- $\geq$ # comparisons made by algorithm $\geq$ length of any root-to-leaf path in decision tree
- What is minimum height of any binary tree on $\geq n + 1$ nodes?
- Minimum height when binary tree is complete (like a heap's tree)
- Height $\geq \lceil \lg n \rceil - 1 = \Omega(\log n)$, thus running time of any comparison sort is also $\Omega(\log n)$
- Sorted array and AVL Trees achieve this bound! Yay!
- Actually, height of any tree with max branching factor $b = O(1)$ is at least $\Omega(\log_b n)$
- For 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!
- Associate a meaning to each index of array (like heap, but simpler)
- **Idea!** Give each item **unique** integer key k in $\{0, \ldots, u - 1\}$, store item at array index k
- Anything in computer memory is a binary integer, or use (static) 64-bit address in memory
- If keys fit in $O(1)$ words, i.e. $k \in O(\log n)$, worst-case $O(1)$ dynamic set operations! Yay!
- 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
- Recall $w \geq \lg n$: so hash fits in $O(1)$ words, $O(1)$ time to compare
- If $m \ll u$, no hash function is injective by pigeonhole principle
- Exists keys a, b such that $h(a) = h(b) \to$ **Collision**! :(
- Can't store both items at index $h(a)$, so where to store? Either:
  - store somewhere else in the array (**open addressing**)
  - 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. $h(k) = c$, 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 this with some additional complexity
- 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): $h_{ab}(k) = (((ak + b) \bmod p) \bmod m)$

- Hash Family $\mathcal{H}(p, m) = \{h_{ab} \mid a, b \in \{1, \ldots, p-1\}\}$
- Parameterized by a fixed prime $p > u$, with a and b chosen from range $\{1, \ldots, p\}$
- $\mathcal{H}$ is a **Universal** family: $\Pr_{h \in \mathcal{H}}\{h(k_i) = h(k_j)\} \leq 1/m \quad \forall k_i \neq k_j \in \{0, \ldots, u-1\}$
- 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**!
- If n/m far from 1, rebuild with new random 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! :)

Set Interface	Operation, Worst Case $O(\cdot)$						Space
	Set	Dynamic (D)		Order (O)		D + O	
Data Structure Implementation	`find(k)`	`insert(x)`	`delete(k)`	`find_next(k)`	`find_max()`	`delete_max()`	$\sim \times n$
Unsorted Array	n	n	n	n	n	n	1
Linked List	n	1	n	n	n	n	3
Dynamic Array	n	$1_{(a)}$	n	n	n	n	4
Sorted Array	$\lg n$	n	n	$\lg n$	1	n	1
Max-Heap	n	$\lg n_{(a)}$	n	n	1	$\lg n$	1
Balanced BST	$\lg n$	$\lg n$	$\lg n$	$\lg n$	(1)	$\lg n$	5
Direct Access	1	1	1	u	u	u	u/n
Hash Table	$1_{(e)}$	$1_{(e,a)}$	$1_{(e,a)}$	n	n	n	4