

May 1, 2024

Introduction to Algorithms: 6.006

Massachusetts Institute of Technology

Instructors: Erik Demaine, Jason Ku, and Justin Solomon

Lecture 4: Hashing

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	$\log n$

premise of today's lecture

- **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

→ Approach to show if

- 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 $n+1$

- An internal node represents a **binary comparison**, branching either True or False

- For a comparison algorithm, the decision tree is binary (draw example)

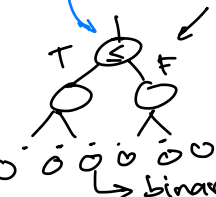
- A leaf represents algorithm termination, resulting in an algorithm **output**

- 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

execution is only one path!
→ longest path = height of tree...

internal nodes are comp comparisons



leaves = terminate algo → output

traversing the tree (or algo) should be able to return any of the n elems (find(k)) or that the key is not present - hence # leaves = $n+1$

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? $\rightarrow \Theta(\log n)$ *will do this in recit. requires tree is balanced*
- 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 $\Omega(\log n)$ *means find(key) $\in O(\log n)$*
- Sorted arrays achieve this bound! Yay!
- More generally, height of tree with $\Theta(n)$ leaves and max branching factor b is $\Omega(\log_b n)$ *hence if we can adjust b , we can improve $\Omega(\log_b n)$*
- To get faster, need an operation that allows super-constant $\omega(1)$ branching factor. How??

Direct Access Array

- a Set Data Structure (not sequence!)*
- Exploit Word-RAM $O(1)$ time random access indexing! Linear branching factor! *if we have a large branching factor instead of comparison's $b=2$, we can decrease $\Omega(\log_b n)$. How*
 - **Idea!** Give item unique integer key k in $\{0, \dots, u-1\}$, store item in an array at index k *use key as idx to store the item of that key on loc idx of memory.*
 - 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! *allows for $O(1)$ access (like in static arrays).*
 - 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 $n \ll u$... *size of largest key \rightarrow mem alloc*
 - **Example:** if keys are ten-letter names, for one bit per name, requires $26^{10} \approx 17.6$ TB space, *means since addr will be key's address a fn of key's address. then memory will be big is long.*
 - 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:** $h(k) : \{0, \dots, u-1\} \rightarrow \{0, \dots, m-1\}$ (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


Lecture 4: Hashing

- Always exists keys a, b such that $h(a) = h(b) \rightarrow$ **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**)

size of smaller mem alloc (set per page)
i.e. - set $m > n$ so there's space...

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?

stored items w/ same keys
 $h(a) = h(b)$ will be stored
in another DS


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 create $O(n)$ size chain ✖

- **Idea!** Don't use a fixed hash function! Choose one randomly (but carefully)!

hence...

way to map {large set} \rightarrow {smaller set}, mod it!

size of smaller set.

universal hash fn

Lecture 4: Hashing

like the previous but
 ① does linear trans.
 ② does mod twice.

Universal (good, theoretically):

$$h_{ab}(k) = (((ak + b) \bmod p) \bmod m)$$

- Hash Family $\mathcal{H}(p, m) = \{h_{ab} \mid a, b \in \{0, \dots, p-1\} \text{ and } a \neq 0\} \rightarrow$
 fixed, determines the family. \leftarrow random
- Parameterized by a fixed prime $p > u$, with a and b chosen from range $\{0, \dots, p-1\}$

- \mathcal{H} is a **Universal** family: $\Pr\{h(k_i) = h(k_j)\} \leq 1/m \quad \forall k_i \neq k_j \in \{0, \dots, u-1\}$
 if we randomly choose h from \mathcal{H} \rightarrow prob of collision $\leq 1/m$ for all diff 2 keys... } proof done in O4h
- 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
 binary probability $p=1-q$.

- Size of chain at index $h(k_i)$ is random variable $X_i = \sum_{j=1}^{n-1} X_{ij}$ \rightarrow 1 if k_i, k_j collides

- Expected size of chain at index $h(k_i)$

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

over choosing h from \mathcal{H} \rightarrow independent \rightarrow sum of all collisions of k_i among diff k_j ...
 $k_i \neq k_j$ diff collide when they're same key
 \leftarrow universal property above
 \leftarrow $n-1$ keys...

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

as long as we choose $m > n$ then m will be constant ...

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

word size of key
 $u = 2^w$

\rightarrow because next isn't hashed next to k ...

May 2, 2024

Introduction to Algorithms: 6.006

Massachusetts Institute of Technology

Instructors: Erik Demaine, Jason Ku, and Justin Solomon

Recitation 4

Recitation 4

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	$\log n$

We've learned how to implement a set interface using a sorted array, where query operations are efficient but whose dynamic operations are lacking. Recalling that $\Theta(\log n)$ growth is much closer to $\Theta(1)$ than $\Theta(n)$, a sorted array provides really good performance! But one of the most common operations you will do in programming is to search for something you're storing, i.e., `find(k)`. Is it possible to `find` faster than $\Theta(\log n)$? It turns out that if the only thing we can do to items is to compare their relative order, then the answer is **no**!

Comparison Model

The comparison model of computation acts on a set of **comparable** objects. The objects can be thought of as black boxes, supporting only a set of binary boolean operations called **comparisons** (namely $<$, \leq , $>$, \geq , $=$, and \neq). Each operation takes as input two objects and outputs a Boolean value, either **True** or **False**, depending on the relative ordering of the elements. A search algorithm operating on a set of n items will return a stored item with a key equal to the input key, or return no item if no such item exists. In this section, we assume that each item has a unique key.

If binary comparisons are the only way to distinguish between stored items and a search key, a deterministic comparison search algorithm can be thought of as a fixed binary **decision tree** representing all possible executions of the algorithm, where each node represents a comparison performed by the algorithm. During execution, the algorithm walks down the tree along a path from the root. For any given input, a comparison sorting algorithm will make some comparison first, the comparison at the root of the tree. Depending on the outcome of this comparison, the computation will then proceed with a comparison at one of its two children. The algorithm repeatedly makes comparisons until a leaf is reached, at which point the algorithm terminates, returning an output to the algorithm. There must be a leaf for each possible output to the algorithm. For search, there are $n + 1$ possible outputs, the n items and the result where no item is found, so there must be at least $n + 1$ leaves in the decision tree. Then the worst-case number of comparisons that must be made by any comparison search algorithm will be the height of the algorithm's decision tree, i.e., the length of any longest root to leaf path.

Exercise: Prove that the smallest height for any tree on n nodes is $\lceil \lg(n+1) \rceil - 1 = \Omega(\log n)$.

Solution: We show that the maximum number of nodes in any binary tree with height h is $n \leq T(h) = 2^{h+1} - 1$, so $h \geq (\lg(n+1)) - 1$. Proof by induction on h . The only tree of height zero has one node, so $T(0) = 1$, a base case satisfying the claim. The maximum number of nodes in a height- h tree must also have the maximum number of nodes in its two subtrees, so $T(h) = 2T(h-1) + 1$. Substituting $T(h)$ yields $2^{h+1} - 1 = 2(2^h - 1) + 1$, proving the claim. \square

$T(h)$ = # of nodes in a binary tree of height h .

We're proving: $T(h) = 2^{h+1} - 1$ (which is equiv to: $h \geq \lg(n+1) - 1$) \hookrightarrow can also be thought as $T(h+1) = 2T(h) + 1$
 $2^{h+2} - 1 = 2(2^{h+1} - 1) + 1$

A tree with $n+1$ leaves has more than n nodes, so its height is at least $\Omega(\log n)$. Thus the minimum number of comparisons needed to distinguish between the n items is at least $\Omega(\log n)$, and the worst-case running time of any deterministic comparison search algorithm is at least $\Omega(\log n)$! So sorted arrays and balanced BSTs are able to support $\text{find}(k)$ asymptotically **optimally**, in a comparison model of computation.

Comparisons are very limiting because each operation performed can lead to at most constant branching factor in the decision tree. It doesn't matter that comparisons have branching factor two; any fixed constant branching factor will lead to a decision tree with at least $\Omega(\log n)$ height. If we were not limited to comparisons, it opens up the possibility of faster-than- $O(\log n)$ search. More specifically, if we can use an operation that allows for asymptotically larger than constant $\omega(1)$ branching factor, then our decision tree could be shallower, leading to a faster algorithm.

Direct Access Arrays \hookrightarrow "let $\text{idx} = \text{int}(x.\text{key})$ and store x in idx " \hookrightarrow allows $O(1)$ $\text{find}(x)$ but requires huge mem!

Most operations within a computer only allow for constant logical branching, like if statements in your code. However, one operation on your computer allows for non-constant branching factor: specifically the ability to randomly access any memory address in constant time. This special operation allows an algorithm's decision tree to branch with large branching factor, as large as there is space in your computer. To exploit this operation, we define a data structure called a **direct access array**, which is a normal static array that associates a semantic meaning with each array index location: specifically that any item x with key k will be stored at array index k . This statement only makes sense when item keys are integers. Fortunately, in a computer, any thing in memory can be associated with an integer—for example, its value as a sequence of bits or its address in memory—so from now on we will only consider integer keys.

Now suppose we want to store a set of n items, each associated with a **unique integer key in the bounded range** from 0 to some large number $u - 1$. We can store the items in a length u direct access array, where each array slot i contains an item associated with integer key i , if it exists. To find an item having integer key i , a search algorithm can simply look in array slot i to respond to the search query in **worst-case constant time!** However, order operations on this data structure will be very slow: we have no guarantee on where the first, last, or next element is in the direct access array, so we may have to spend u time for order operations.

$\hookrightarrow O(n)$

$u-1 = \text{max key value}$

Worst-case constant time search comes at the cost of storage space: a direct access array must have a slot available for every possible key in range. When u is very large compared to the number of items being stored, storing a direct access array can be wasteful, or even impossible on modern machines. For example, suppose you wanted to support the set $\text{find}(k)$ operation on ten-letter names using a direct access array. The space of possible names would be $u \approx 26^{10} \approx 9.5 \times 10^{13}$; even storing a bit array of that length would require 17.6 Terabytes of storage space. How can we overcome this obstacle? The answer is hashing!

```

1 class DirectAccessArray:
2     def __init__(self, u): self.A = [None] * u      # O(u)
3     def find(self, k):     return self.A[k]         # O(1)
4     def insert(self, x):   self.A[x.key] = x        # O(1)
5     def delete(self, k):   self.A[k] = None         # O(1)
6     def find_next(self, k):
7         for i in range(k, len(self.A)):             # O(u)
8             if A[i] is not None:
9                 return A[i]
10    def find_max(self):
11        for i in range(len(self.A) - 1, -1, -1):     # O(u)
12            if A[i] is not None:
13                return A[i]
14    def delete_max(self):
15        for i in range(len(self.A) - 1, -1, -1):     # O(u)
16            x = A[i]
17            if x is not None:
18                A[i] = None
19            return x

```

→ create array of len = max possible k value.

} returns the next $A[i]$ after k that is not none... distance bet. k and i that leads to $A[i] \neq \text{None}$ is almost u .

max() isn't necessarily @ $\text{len}(A)$
 $\text{len}(A) = 2^{\text{word-len-of-keys}}$

Hashing

Is it possible to get the performance benefits of a direct access array while using only linear $O(n)$ space when $n \ll u$? A possible solution could be to store the items in a smaller **dynamic** direct access array, with $m = O(n)$ slots instead of u , which grows and shrinks like a dynamic array depending on the number of items stored. But to make this work, we need a function that maps item keys to different slots of the direct access array, $h(k) : \{0, \dots, u-1\} \rightarrow \{0, \dots, m-1\}$. We call such a function a **hash function** or a **hash map**, while the smaller direct access array is called a **hash table**, and $h(k)$ is the **hash** of integer key k . If the hash function happens to be injective over the n keys you are storing, i.e. no two keys map to the same direct access array index, then we will be able to support worst-case constant time search, as the hash table simply acts as a direct access array over the smaller domain m .

open addr = store colliding item somewhere else in the same direct access array
 chaining = store in a separate array.

Unfortunately, if the space of possible keys is larger than the number of array indices, i.e. $m < u$, then any hash function mapping u possible keys to m indices must map multiple keys to the same array index, by the pigeonhole principle. If two items associated with keys k_1 and k_2 hash to the same index, i.e. $h(k_1) = h(k_2)$, we say that the hashes of k_1 and k_2 **collide**. If you don't know in advance what keys will be stored, it is extremely unlikely that your choice of hash function will avoid collisions entirely¹. If the smaller direct access array hash table can only store one item at each index, when collisions occur, where do we store the colliding items? Either we store collisions somewhere else in the same direct access array, or we store collisions somewhere else. The first strategy is called **open addressing**, which is the way most hash tables are actually implemented, but such schemes can be difficult to analyze. We will adopt the second strategy called **chaining**.

Chaining → instead of storing items in the direct access arr, (or hash table) we store addr of chains instead, so that when there's a collision, we just add a link to the chain at index $h(x.key)$.

Chaining is a collision resolution strategy where colliding keys are stored separately from the original hash table. Each hash table index holds a pointer to a **chain**, a separate data structure that supports the dynamic set interface, specifically operations `find(k)`, `insert(x)`, and `delete(k)`. It is common to implement a chain using a linked list or dynamic array, but any implementation will do, as long as each operation takes no more than linear time. Then to insert item x into the hash table, simply insert x into the chain at index $h(x.key)$; and to find or delete a key k from the hash table, simply find or delete k from the chain at index $h(k)$.

Ideally, we want chains to be small, because if our chains only hold a constant number of items, the dynamic set operations will run in constant time. But suppose we are unlucky in our choice of hash function, and all the keys we want to store hash all of them to the same index location, into the same chain. Then the chain will have linear size, meaning the dynamic set operations could take linear time. A good hash function will try to minimize the frequency of such collisions in order to minimize the maximum size of any chain. So what's a good hash function?

Hash Functions

Division Method (bad): The simplest mapping from an integer key domain of size u to a smaller one of size m is simply to divide the key by m and take the remainder: $h(k) = (k \bmod m)$, or in Python, `k % m`. If the keys you are storing are uniformly distributed over the domain, the division method will distribute items roughly evenly among hashed indices, so we expect chains to have small size providing good performance. However, if all items happen to have keys with the same remainder when divided by m , then this hash function will be terrible. Ideally, the performance of our data structure would be **independent** of the keys we choose to store.

¹If you know all of the keys you will want to store in advance, it is possible to design a hashing scheme that will always avoid collisions between those keys. This idea, called **perfect hashing**, follows from the Birthday Paradox.

Universal Hashing (good): For a large enough key domain u , every hash function will be bad for some set of n inputs². However, we can achieve good **expected** bounds on hash table performance by choosing our hash function **randomly** from a large family of hash functions. Here the expectation is over our choice of hash function, which is independent of the input. **This is not expectation over the domain of possible input keys.** One family of hash functions that performs well is:

$$\mathcal{H}(m, p) = \left\{ h_{ab}(k) = ((ak + b) \bmod p) \bmod m \mid a, b \in \{0, \dots, p-1\} \text{ and } a \neq 0 \right\},$$

where p is a prime that is larger than the key domain u . A single hash function from this family is specified by choosing concrete values for a and b . This family of hash functions is **universal**³: for any two keys, the probability that their hashes will collide when hashed using a hash function chosen uniformly at random from the universal family, is no greater than $1/m$, i.e.

over diff hash func's. $\Pr_{h \in \mathcal{H}} \{h(k_i) = h(k_j)\} \leq 1/m, \quad \forall k_i \neq k_j \in \{0, \dots, u-1\}.$

If we know that a family of hash functions is universal, then we can upper bound the expected size of any chain, **in expectation over our choice of hash function** from the family. Let X_{ij} be the indicator random variable representing the value 1 if keys k_i and k_j collide for a chosen hash function, and 0 otherwise. Then the random variable representing the number of items hashed to index $h(k_i)$ will be the sum $X_i = \sum_j X_{ij}$ over all keys k_j from the set of n keys $\{k_0, \dots, k_{n-1}\}$ stored in the hash table. Then the expected number of keys hashed to the chain at index $h(k_i)$ is:

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

independent (pointing to the sum over j)

mod m (pointing to the denominator m)

i.e. is m is based on n (pointing to the denominator m)

If the size of the hash table is at least linear in the number of items stored, i.e. $m = \Omega(n)$, then the expected size of any chain will be $1 + (n-1)/\Omega(n) = O(1)$, a constant! Thus a hash table where collisions are resolved using chaining, implemented using a randomly chosen hash function from a universal family, will perform dynamic set operations in **expected constant time**, where the expectation is taken over the random choice of hash function, independent from the input keys! Note that in order to maintain $m = O(n)$, insertion and deletion operations may require you to rebuild the direct access array to a different size, choose a new hash function, and reinsert all the items back into the hash table. This can be done in the same way as in dynamic arrays, leading to **amortized bounds for dynamic operations.**

not amortized, but expected. (pointing to 'expected constant time')

²If $u > nm$, every hash function from u to m maps some n keys to the same hash, by the pigeonhole principle.

³The proof that this family is universal is beyond the scope of 6.006, though it is usually derived in 6.046.

```

1 class Hash_Table_Set:
2     def __init__(self, r = 200):                # O(1)
3         self.chain_set = Set_from_Seq(Linked_List_Seq)
4         self.A = []
5         self.size = 0
6         self.r = r                             # 100/self.r = fill ratio
7         self.p = 2**31 - 1
8         self.a = randint(1, self.p - 1)
9         self._compute_bounds()
10        self._resize(0)
11
12    def __len__(self): return self.size          # O(1)
13    def __iter__(self):                          # O(n)
14        for X in self.A:
15            yield from X
16
17    def build(self, X):                          # O(n)e
18        for x in X: self.insert(x)
19
20    def _hash(self, k, m):                      # O(1)
21        return ((self.a * k) % self.p) % m      b=0
22
23    def _compute_bounds(self):                  # O(1)
24        self.upper = len(self.A)
25        self.lower = len(self.A) * 100*100 // (self.r*self.r)
26
27    def _resize(self, n):                      # O(n)
28        if (self.lower >= n) or (n >= self.upper):
29            f = self.r // 100
30            if self.r % 100: f += 1
31            # f = ceil(r / 100)
32            m = max(n, 1) * f
33            A = [self.chain_set() for _ in range(m)]
34            for x in self:
35                h = self._hash(x.key, m)
36                A[h].insert(x)
37            self.A = A
38            self._compute_bounds()
39
40    def find(self, k):                          # O(1)e
41        h = self._hash(k, len(self.A))
42        return self.A[h].find(k)
43
44    def insert(self, x):                        # O(1)ae
45        self._resize(self.size + 1)
46        h = self._hash(x.key, len(self.A))
47        added = self.A[h].insert(x)
48        if added: self.size += 1
49        return added
50
51

```

self.lower
is a certain
percentage of
self.upper...

square of fill ratio

} what for?

m is based on n

create array
of chains

→ rehash all items.

(same hash fn for all since m doesn't change)
independent of x.

get hash
and find on the chain @ h[hash(k)]

```
52     def delete(self, k):                                # O(1)ae
53         assert len(self) > 0
54         h = self._hash(k, len(self.A))
55         x = self.A[h].delete(k)
56         self.size -= 1
57         self._resize(self.size)
58         return x
59
60     def find_min(self):                                  # O(n)
61         out = None
62         for x in self:
63             if (out is None) or (x.key < out.key):
64                 out = x
65         return out
66
67     def find_max(self):                                  # O(n)
68         out = None
69         for x in self:
70             if (out is None) or (x.key > out.key):
71                 out = x
72         return out
73
74     def find_next(self, k):                              # O(n)
75         out = None
76         for x in self:
77             if x.key > k:
78                 if (out is None) or (x.key < out.key):
79                     out = x
80         return out
81
82     def find_prev(self, k):                              # O(n)
83         out = None
84         for x in self:
85             if x.key < k:
86                 if (out is None) or (x.key > out.key):
87                     out = x
88         return out
89
90     def iter_order(self):                                # O(n^2)
91         x = self.find_min()
92         while x:
93             yield x
94             x = self.find_next(x.key)
```

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

Exercise

Given an unsorted array $A = [a_0, \dots, a_{n-1}]$ containing n positive integers, the **DUPLICATES** problem asks whether two integers in the array have the same value.

- 1) Describe a brute-force **worst-case** $O(n^2)$ -time algorithm to solve **DUPLICATES**. (compare each element with all others)
Solution: Loop through all $\binom{n}{2} = O(n^2)$ pairs of integers from the array and check if they are equal in $O(1)$ time.
- 2) Describe a **worst-case** $O(n \log n)$ -time algorithm to solve **DUPLICATES**. \rightarrow build a sorted array $O(n \log n)$ via merge-sort, then scan adjacent pairs for duplicates (few loops!).
Solution: Sort the array in worst-case $O(n \log n)$ time (e.g. using merge sort), and then scan through the sorted array, returning if any of the $O(n)$ adjacent pairs have the same value.
- 3) Describe an **expected** $O(n)$ -time algorithm to solve **DUPLICATES**. \rightarrow use the integers as hash keys to create a hash table @ $O(n)$. In each insert, $O(1)$ as check if it has a pair in the chain.
Solution: Hash each of the n integers into a hash table (implemented using chaining and a hash function chosen randomly from a universal hash family⁴), with insertion taking expected $O(1)$ time. When inserting an integer into a chain, check it against the other integers already in the chain, and return if another integer in the chain has the same value. Since each chain has expected $O(1)$ size, this check takes expected $O(1)$ time, so the algorithm runs in expected $O(n)$ time. $\rightarrow O(1)$ also since length of chain is $O(1)$.
- 4) If $k < n$ and $a_i \leq k$ for all $a_i \in A$, describe a **worst-case** $O(1)$ -time algorithm to solve **DUPLICATES**. \rightarrow # of ints \rightarrow int values is at most less than how many there are \rightarrow then at least one integer would always have a pair...
Solution: If $k < n$, a duplicate always exists, by the pigeonhole principle. \rightarrow number at most k
- 5) If $n \leq k$ and $a_i \leq k$ for all $a_i \in A$, describe a **worst-case** $O(k)$ -time algorithm to solve **DUPLICATES**. \rightarrow values \leq most $\leq k$ $\rightarrow a_i$ can be used as idx since it's bounded by k .
Solution: Insert each of the n integers into a direct access array of length k , which will take worst-case $O(k)$ time to instantiate, and worst-case $O(1)$ time per insert operation. If an integer already exists at an array index when trying to insert, then return that a duplicate exists.

⁴In 6.006, you do not have to specify these details when answering problems. You may simply quote that hash tables can achieve the expected/amortized bounds for operations described in class.