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. $\square$

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 `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

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.

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 `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!

```
class DirectAccessArray:
    def __init__(self, u):  self.A = [None] * u   # O(u)
    def find(self, k):      return self.A[k]      # O(1)
    def insert(self, x):    self.A[x.key] = x     # O(1)
    def delete(self, k):    self.A[k] = None      # O(1)
    def find_next(self, k):
        for i in range(k, len(self.A)):           # O(u)
            if A[i] is not None:
                return A[i]
    def find_max(self):
        for i in range(len(self.A) - 1, -1, -1):  # O(u)
            if A[i] is not None:
                return A[i]
    def delete_max(self):
        for i in range(len(self.A) - 1, -1, -1):  # O(u)
            x = A[i]
            if x is not None:
                A[i] = None
                return x
```

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, $h(k) : \{0, \dots, u-1\} \to \{0, \dots, m-1\}$. 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.

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 entirely[1]. 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

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 has 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.

[1] If 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 inputs[2]. 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:

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

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.

$$\Pr_{h \in \mathcal{H}}\{h(k_i) = h(k_j)\} \leq 1/m, \quad \forall k_i \neq k_j \in \{0, \ldots, u-1\}.$$

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 $\{k_0, \ldots, k_{n-1}\}$ stored in the hash table. Then the expected number of keys hashed to the chain at index $h(k_i)$ is:

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

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**.

[2]If $u > nm$, every hash function from u to m maps some n keys to the same hash, by the pigeonhole principle.
[3]The proof that this family is universal is beyond the scope of 6.006, though it is usually derived in 6.046.

```
class Hash_Table_Set:
    def __init__(self, r = 200):              # O(1)
        self.chain_set = Set_from_Seq(Linked_List_Seq)
        self.A = []
        self.size = 0
        self.r = r                            # 100/self.r = fill ratio
        self.p = 2**31 - 1
        self.a = randint(1, self.p - 1)
        self._compute_bounds()
        self._resize(0)

    def __len__(self):  return self.size      # O(1)
    def __iter__(self):                       # O(n)
        for X in self.A:
            yield from X

    def build(self, X):                       # O(n)e
        for x in X: self.insert(x)

    def _hash(self, k, m):                    # O(1)
        return ((self.a * k) % self.p) % m

    def _compute_bounds(self):                # O(1)
        self.upper = len(self.A)
        self.lower = len(self.A) * 100*100 // (self.r*self.r)

    def _resize(self, n):                     # O(n)
        if (self.lower >= n) or (n >= self.upper):
            f = self.r // 100
            if self.r % 100:     f += 1
            # f = ceil(r / 100)
            m = max(n, 1) * f
            A = [self.chain_set() for _ in range(m)]
            for x in self:
                h = self._hash(x.key, m)
                A[h].insert(x)
            self.A = A
            self._compute_bounds()

    def find(self, k):                        # O(1)e
        h = self._hash(k, len(self.A))
        return self.A[h].find(k)

    def insert(self, x):                      # O(1)ae
        self._resize(self.size + 1)
        h = self._hash(x.key, len(self.A))
        added = self.A[h].insert(x)
        if added:   self.size += 1
        return added
```

```
    def delete(self, k):                    # O(1)ae
        assert len(self) > 0
        h = self._hash(k, len(self.A))
        x = self.A[h].delete(k)
        self.size -= 1
        self._resize(self.size)
        return x

    def find_min(self):                     # O(n)
        out = None
        for x in self:
            if (out is None) or (x.key < out.key):
                out = x
        return out

    def find_max(self):                     # O(n)
        out = None
        for x in self:
            if (out is None) or (x.key > out.key):
                out = x
        return out

    def find_next(self, k):                 # O(n)
        out = None
        for x in self:
            if x.key > k:
                if (out is None) or (x.key < out.key):
                    out = x
        return out

    def find_prev(self, k):                 # O(n)
        out = None
        for x in self:
            if x.key < k:
                if (out is None) or (x.key > out.key):
                    out = x
        return out

    def iter_order(self):                   # O(n^2)
        x = self.find_min()
        while x:
            yield x
            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 $A = [a_0, \ldots, a_{n-1}]$ 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.
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.
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.
Solution: Hash each of the n integers into a hash table (implemented using chaining and a hash function chosen randomly from a universal hash family[4]), 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.

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.
Solution: If $k < n$, a duplicate always exists, by the pigeonhole principle.

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.
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.

[4] In 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.