Recitation 4

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, $h(k) : \{0, \ldots, u-1\} \to \{0, \ldots, m-1\}$. 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. 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, and all the keys we want to store hash 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.

To analyze the performance of chaining on average, it will help to make the following assumption on the behavior of our hash function for random keys.

Simple Uniform Hashing Assumption (SUHA): A random key will be equally likely to hash to any one of the m buckets $\{0, \ldots, m-1\}$, independently of any other keys.

This assumption allows us to study the average-case performance of chaining when hashing random keys. In this case, it can be shown (though we do not prove it here), that all of the set operations take $O(1 + \alpha)$ time, where $\alpha = n/m$ is the load factor of the hash table (the number of elements divided by the number of buckets).

Open Addressing

Open addressing is a different way of dealing with collisions. Whereas in *chaining* we use a different data structure to deal with collisions, in open addressing collisions are stored in the hash table itself!

Different open addressing schemes are defined by different *probing* strategies. A probing strategy dictates which slot to try next if the particular slot being looked at is occupied with a different key. Mathematically, an open addressing hashing scheme is defined by a function (the probing scheme)

$$h : \mathcal{U} \times \{0, \ldots, m-1\} \mapsto \{0, \ldots, m-1\}$$

For a given key k, we first attempt to store k in $h(k, 0)$, if occupied, we try $h(k, 1)$, then $h(k, 2)$ and so on (see Figure 1).

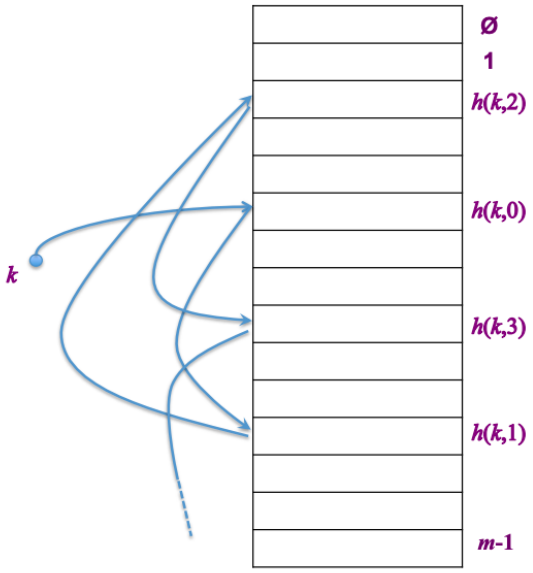

Figure 1: A probing scheme

Exercise: When using open addressing, what happens when the load factor α becomes larger than 1?

Given a fixed probing scheme h, how do we perform an insert, delete or search operation?

1. INSERT: When inserting a key k, we just follow the probing scheme, in order, until we find an open slot. See Figure 2. In the example, we are inserting key 496.

2. SEARCH: When searching for a key k, we again follow the probing scheme until we either find k, or we find an empty slot. If we find an empty slot, we report NOT FOUND.

3. DELETE: When deleting keys, one needs to use *lazy-deleting*. If one just deletes entries, one may prevent existing keys from being found. Consider the example in Figure 2. If we delete key 586, then we wouldn't be able to find key 496! Rather, instead of deleting an item, we replace it with a special key (say -1). This special key is treated as "occupied" when searching, but as "empty" when inserting.

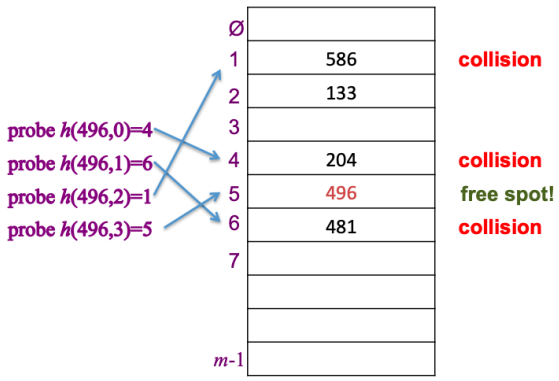

Figure 2: Inserting a key

Assuming that the probing scheme function h can be evaluated in $O(1)$ time, open addressing has similar performance to chaining, with some efficiencies especially for small load factors.

Exercise: What are the pros and cons of open addressing relative to chaining?

Probing schemes

Before discussing some of the known probing schemes, we will discuss an assumption which could help us prove formal statements. It is the analogue of the *simple uniform hashing assumption*. Even though most probing schemes do not satisfy the assumption, it still gives us something to aim for.

Uniform Hashing Assumption: The probe sequence of a random key is equally likely to be any of the $m!$ permutations of $\{0, \ldots, m-1\}$.

Note that the *UHA* is quite a strong assumption. Not only the initial problem is uniformly chosen at random (like in the *simple uniform hashing assumption*), but every subsequent probe is also chosen uniformly among the cells not yet probed.

Linear probing

We will now discuss some of the standard probing schemes. The first one is the simplest to implement. It is called *linear probing* for obvious reasons.

The linear probing scheme is defined as $h(k,i) = (h'(k) + i) \mod m$ where h' is a regular hash function.

Does the linear probing scheme have the uniform hashing assumption? No! If two keys are mapped to the same initial slot by h', then they will follow the same probing sequence. So there are only m possible permutations instead of the required $m!$.

Double hashing

Double hashing is usually a better alternative to linear probing. We define $h(k,i) = (h_1(k) + ih_2(k)) \mod m$ where h_1 and h_2 are two hash functions. Clearly, one must ensure that no key is mapped to 0 by h_2 (why?). Also, one tries to ensure that, for all k, $h_2(k)$ is relatively prime to m. Otherwise, for that particular key k, one wouldn't reach all slots (why?). This is easily done by using prime m.

To better visualize the scheme, we can take $h_1(k) = k \mod m$ and $h_2(k) = 1+(k \mod (m-1))$ for m prime. Clearly, both requirements are satisfied.

Hash Functions

Division Method: 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 on average 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. Nevertheless, the division method often works in practice for some choices of m: one heuristic is to choose m to be a large prime number.

Regardless of the choice of hash function, let us suppose that the simple uniform hashing assumption (SUHA) holds. Under this assumption, if the size of the hash table is at least linear in the number of items stored, i.e. $m = \Omega(n)$, then the average 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, operating on random data that satisfy the SUHA assumption, will perform dynamic set operations in **average-case constant time**. 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**[1].

[1] See `https://codepen.io/mit6006/pen/rNWYYax` for a demo.

```
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


```

```
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	$n_{(SUHA)}$	$1_{(SUHA)}$	$1_{(a)(SUHA)}$	n	n

Exercise

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

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 **average-case** $O(n)$-time algorithm to solve DUPLICATES.
Solution: Hash each of the n integers into a hash table, with insertion taking average-case $O(1)$ 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 the expected size of the chains is $O(1)$, this check takes $O(1)$ on average, so the algorithm runs in average-case $O(n)$ 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.