Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — February 24, 2021
Instructors: Mauricio Karchmer, Anand Natarajan, Nir Shavit — Recitation 3

Recitation 3

Dynamic Array Sequence

The array's dynamic sequence operations require linear time with respect to the length of array A. Is there another way to add elements to an array without paying a linear overhead transfer cost each time you add an element? One straight-forward way to support faster insertion would be to over-allocate additional space when you request space for the array. Then, inserting an item would be as simple as copying over the new value into the next empty slot. This compromise trades a little extra space in exchange for constant time insertion. Sounds like a good deal, but any additional allocation will be bounded; eventually repeated insertions will fill the additional space, and the array will again need to be reallocated and copied over. Further, any additional space you reserve will mean less space is available for other parts of your program.

Then how does Python support appending to the end of a length n Python List in worst-case $O(1)$ time? The answer is simple: **it doesn't**. Sometimes appending to the end of a Python List requires $O(n)$ time to transfer the array to a larger allocation in memory, so **sometimes** appending to a Python List takes linear time. However, allocating additional space in the right way can guarantee that any sequence of n insertions only takes at most $O(n)$ time (i.e. such linear time transfer operations do not occur often), so insertion will take $O(1)$ time per insertion **on average**. We call this asymptotic running time **amortized constant time**, because the cost of the operation is amortized (distributed) across many applications of the operation.

To achieve an amortized constant running time for insertion into an array, our strategy will be to allocate extra space in proportion to the size of the array being stored. Allocating $O(n)$ additional space ensures that a linear number of insertions must occur before an insertion will overflow the allocation. A typical implementation of a dynamic array will allocate double the amount of space needed to store the current array, sometimes referred to as **table doubling**. However, allocating any constant fraction of additional space will achieve the amortized bound. Python Lists allocate additional space according to the following formula (from the Python source code written in C):

```
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
```

Here, the additional allocation is modest, roughly one eighth of the size of the array being appended (bit shifting the size to the right by 3 is equivalent to floored division by 8). But the additional allocation is still linear in the size of the array, so on average, $n/8$ insertions will be performed for every linear time allocation of the array, i.e. amortized constant time.

What if we also want to remove items from the end of the array? Popping the last item can occur in constant time, simply by decrementing a stored length of the array (which Python does). However, if a large number of items are removed from a large list, the unused additional allocation could occupy a significant amount of wasted memory that will not available for other purposes.

When the length of the array becomes sufficiently small, we can transfer the contents of the array to a new, smaller memory allocation so that the larger memory allocation can be freed. How big should this new allocation be? If we allocate the size of the array without any additional allocation, an immediate insertion could trigger another allocation. To achieve constant amortized running time for any sequence of n appends or pops, we need to make sure there remains a linear fraction of unused allocated space when we rebuild to a smaller array, which guarantees that at least $\Omega(n)$ sequential dynamic operations must occur before the next time we need to reallocate memory.

Below is a Python implementation of a dynamic array sequence, including operations `insert_last` (i.e., Python list `append`) and `delete_last` (i.e., Python list `pop`), using table doubling proportions. When attempting to append past the end of the allocation, the contents of the array are transferred to an allocation that is twice as large. When removing down to one fourth of the allocation, the contents of the array are transferred to an allocation that is half as large. Of course Python Lists already support dynamic operations using these techniques; this code is provided to help you understand how **amortized constant** `append` and `pop` could be implemented.

```python
class Dynamic_Array_Seq(Array_Seq):
    def __init__(self, r = 2):                         # O(1)
        super().__init__()
        self.size = 0
        self.r = r
        self._compute_bounds()
        self._resize(0)

    def __len__(self):  return self.size               # O(1)

    def __iter__(self):                                # O(n)
        for i in range(len(self)): yield self.A[i]

    def build(self, X):                                # O(n)
        for a in X: self.insert_last(a)

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

    def _resize(self, n):                              # O(1) or O(n)
        if (self.lower < n < self.upper): return
        m = max(n, 1) * self.r
        A = [None] * m
        self._copy_forward(0, self.size, A, 0)
        self.A = A
        self._compute_bounds()

    def insert_last(self, x):                          # O(1)a
        self._resize(self.size + 1)
        self.A[self.size] = x
        self.size += 1

    def delete_last(self):                            # O(1)a
        self.A[self.size - 1] = None
        self.size -= 1
        self._resize(self.size)

    def insert_at(self, i, x):                        # O(n)
        self.insert_last(None)
        self._copy_backward(i, self.size - (i + 1), self.A, i + 1)
        self.A[i] = x

    def delete_at(self, i):                           # O(n)
        x = self.A[i]
        self._copy_forward(i + 1, self.size - (i + 1), self.A, i)
        self.delete_last()
        return x
                                                      # O(n)
    def insert_first(self, x):  self.insert_at(0, x)
    def delete_first(self):     return self.delete_at(0)
```

Exercises:

- Suppose the next pointer of the last node of a linked list points to an earlier node in the list, creating a cycle. Given a pointer to the head of the list (without knowing its size), describe a linear-time algorithm to find the number of nodes in the cycle. Can you do this while using only constant additional space outside of the original linked list?

  Solution: Begin with two pointers pointing at the head of the linked list: one slow pointer and one fast pointer. The pointers take turns traversing the nodes of the linked list, starting with the fast pointer. On the slow pointer's turn, the slow pointer simply moves to the next node in the list; while on the fast pointer's turn, the fast pointer initially moves to the next node, but then moves on to the next node's next node before ending its turn. Every time the fast pointer visits a node, it checks to see whether it's the same node that the slow pointer is pointing to. If they are the same, then the fast pointer must have made a full loop around the cycle, to meet the slow pointer at some node v on the cycle. Now to find the length of the cycle, simply have the fast pointer continue traversing the list until returning back to v, counting the number of nodes visited along the way.

  To see that this algorithm runs in linear time, clearly the last step of traversing the cycle takes at most linear time, as v is the only node visited twice while traversing the cycle. Further, we claim the slow pointer makes at most one move per node. Suppose for contradiction the slow pointer moves twice away from some node u before being at the same node as the fast pointer, meaning that u is on the cycle. In the same time the slow pointer takes to traverse the cycle from u back to u, the fast pointer will have traveled around the cycle twice, meaning that both pointers must have existed at the same node prior to the slow pointer leaving u, a contradiction.

- Given a data structure implementing the Sequence interface, show how to use it to implement the Set interface. (Your implementation does not need to be efficient.)

  Solution:

```python
def Set_from_Seq(seq):
    class set_from_seq:
        def __init__(self):   self.S = seq()
        def __len__(self):    return len(self.S)
        def __iter__(self):   yield from self.S

        def build(self, A):
            self.S.build(A)

        def insert(self, x):
            for i in range(len(self.S)):
                if self.S.get_at(i).key == x.key:
                    self.S.set_at(i, x)
                    return
            self.S.insert_last(x)

        def delete(self, k):
            for i in range(len(self.S)):
                if self.S.get_at(i).key == k:
                    return self.S.delete_at(i)

        def find(self, k):
            for x in self:
                if x.key == k:  return x
            return None

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

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

        def find_next(self, k):
            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):
            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_ord(self):
            x = self.find_min()
            while x:
                yield x
                x = self.find_next(x.key)

    return set_from_seq
```

Faster Sets

We reduced the Set interface to the Sequence Interface (we simulated one with the other). This directly provides a Set data structure from an array (albeit a poor one).

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

We would like to do better, and we will spend significant time trying to do exactly that! One of the simplest ways to get a faster Set is to store our items in a **sorted** array, where the item with the smallest key appears first (at index 0), and the item with the largest key appears last. Then we can simply binary search to find keys and support Order operations! This is still not great for dynamic operations (items still need to be shifted when inserting or removing from the middle of the array), but finding items by their key is much faster! But how do we get a sorted array in the first place?

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)`
Sorted Array	?	$\log n$	n	1	$\log n$

```python
class Sorted_Array_Set:
    def __init__(self):     self.A = Array_Seq()  # O(1)
    def __len__(self):      return len(self.A)    # O(1)
    def __iter__(self):     yield from self.A     # O(n)
    def iter_order(self):   yield from self       # O(n)

    def build(self, X):                           # O(?)
        self.A.build(X)
        self._sort()

    def _sort(self):                              # O(?)
        ??

    def _binary_search(self, k, i, j):            # O(log n)
        if i >= j:          return i
        m = (i + j) // 2
        x = self.A.get_at(m)
        if x.key > k:       return self._binary_search(k, i, m - 1)
        if x.key < k:       return self._binary_search(k, m + 1, j)
        return m

    def find_min(self):                           # O(1)
```

```
        if len(self) > 0:       return self.A.get_at(0)
        else:                   return None

    def find_max(self):                                 # O(1)
        if len(self) > 0:       return self.A.get_at(len(self) - 1)
        else:                   return None

    def find(self, k):                                  # O(log n)
        if len(self) == 0:      return None
        i = self._binary_search(k, 0, len(self) - 1)
        x = self.A.get_at(i)
        if x.key == k:          return x
        else:                   return None

    def find_next(self, k):                             # O(log n)
        if len(self) == 0:      return None
        i = self._binary_search(k, 0, len(self) - 1)
        x = self.A.get_at(i)
        if x.key > k:           return x
        if i + 1 < len(self):   return self.A.get_at(i + 1)
        else:                   return None

    def find_prev(self, k):                             # O(log n)
        if len(self) == 0:      return None
        i = self._binary_search(k, 0, len(self) - 1)
        x = self.A.get_at(i)
        if x.key < k:           return x
        if i > 0:               return self.A.get_at(i - 1)
        else:                   return None

    def insert(self, x):                                # O(n)
        if len(self.A) == 0:
            self.A.insert_first(x)
        else:
            i = self._binary_search(x.key, 0, len(self.A) - 1)
            k = self.A.get_at(i).key
            if k == x.key:
                self.A.set_at(i, x)
                return False
            if k > x.key:   self.A.insert_at(i, x)
            else:           self.A.insert_at(i + 1, x)
        return True

    def delete(self, k):                                # O(n)
        i = self._binary_search(k, 0, len(self.A) - 1)
        assert self.A.get_at(i).key == k
        return self.A.delete_at(i)
```

Sorting

Sorting an array A of comparable items into increasing order is a common subtask of many computational problems. Insertion sort and selection sort are common sorting algorithms for sorting small numbers of items because they are easy to understand and implement. Both algorithms are **incremental** in that they maintain and grow a sorted subset of the items until all items are sorted. The difference between them is subtle:

- **Selection sort** maintains and grows a subset the **largest** i items in sorted order.
- **Insertion sort** maintains and grows a subset of the **first** i input items in sorted order.

Selection Sort

Here is a Python implementation of selection sort. Having already sorted the smallest items into sub-array A[:i], the algorithm repeatedly scans the rest of the array for the smallest item not yet sorted and swaps it with item A[i]. As can be seen from the code, selection sort can require $\Omega(n^2)$ comparisons, but will perform at most $O(n)$ swaps in the worst case.

```python
def selection_sort(A):                      # Selection sort array A
    for i in range(len(A)):                 # O(n) loop over array
        m = i                               # O(1) initial index of min
        for j in range(i, len(A)):          # O(i) search for min in A[i:]
            if A[m] >= A[j]:                # O(1) check for larger value
                m = j                       # O(1) new max found
        A[m], A[i] = A[i], A[m]             # O(1) swap
```

Insertion Sort

Here is a Python implementation of insertion sort. Having already sorted sub-array A[:i], the algorithm repeatedly swaps item A[i] with the item to its left until the left item is no larger than A[i]. As can be seen from the code, insertion sort can require $\Omega(n^2)$ comparisons and $\Omega(n^2)$ swaps in the worst case.

```python
def insertion_sort(A):                      # Insertion sort array A
    for i in range(1, len(A)):              # O(n) loop over array
        j = i                               # O(1) initialize pointer
        while j > 0 and A[j] < A[j - 1]:    # O(i) loop over prefix
            A[j - 1], A[j] = A[j], A[j - 1] # O(1) swap
            j = j - 1                       # O(1) decrement j
```

In-place and Stability

Both insertion sort and selection sort are **in-place** algorithms, meaning they can each be implemented using at most a constant amount of additional space. The only operations performed on the array are comparisons and swaps between pairs of elements. Insertion sort is **stable**, meaning that items having the same value will appear in the sort in the same order as they appeared in the input array. By comparison, this implementation of selection sort is not stable. For example, the input $(2, 1, 1')$ would produce the output $(1', 1, 2)$.

Merge Sort

Merge sort is an asymptotically faster algorithm for sorting large numbers of items. The algorithm recursively sorts the left and right half of the array, and then merges the two halves in linear time. The recurrence relation for merge sort is then $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$. An $\Theta(n \log n)$ asymptotic growth rate is **much closer** to linear than quadratic, as $\log n$ grows exponentially slower than n. In particular, $\log n$ grows slower than any polynomial n^ε for $\varepsilon > 0$.

```python
def merge_sort(A, a = 0, b = None):          # Sort sub-array A[a:b]
    if b is None:                            # O(1) initialize
        b = len(A)                           # O(1)
    if 1 < b - a:                            # O(1) size k = b - a
        c = (a + b + 1) // 2                 # O(1) compute center
        merge_sort(A, a, c)                  # T(k/2) recursively sort left
        merge_sort(A, c, b)                  # T(k/2) recursively sort right
        L, R = A[a:c], A[c:b]                # O(k) copy
        i, j = 0, 0                          # O(1) initialize pointers
        while a < b:                         # O(n)
            if (j >= len(R)) or (i < len(L) and L[i] < R[j]): # O(1) check side
                A[a] = L[i]                  # O(1) merge from left
                i = i + 1                    # O(1) decrement left pointer
            else:
                A[a] = R[j]                  # O(1) merge from right
                j = j + 1                    # O(1) decrement right pointer
            a = a + 1                        # O(1) decrement merge pointer
```

Merge sort uses a linear amount of temporary storage (`temp`) when combining the two halves, so it is **not in-place**. While there exist algorithms that perform merging using no additional space, such implementations are substantially more complicated than the merge sort algorithm. Whether merge sort is stable depends on how an implementation breaks ties when merging. The above implementation is not stable, but it can be made stable with only a small modification. Can you modify the implementation to make it stable? We've made CoffeeScript visualizers for the merge step of this algorithm, as well as one showing the recursive call structure. You can find them here:
`https://codepen.io/mit6006/pen/RYJdOG` `https://codepen.io/mit6006/pen/wEXOOq`

Build a Sorted Array

With an algorithm to sort our array in $\Theta(n \log n)$, we can now complete our table! We sacrifice some time in building the data structure to speed up order queries. This is a common technique called **preprocessing**.

	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
Data Structure	`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	$\log n$