Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Zachary Abel, Erik Demaine, Jason Ku

September 19, 2018
Recitation 4

Recitation 4

Sequence Interface

Data Structure	Operation, Worst Case $O(\cdot)$					Space
	Static		Dynamic			
	`at(i)` `set(i,x)`	`left()` `right()`	`insert_at(i,x)` `delete_at(i)`	`insert_left(x)` `delete_left()`	`insert_right(x)` `delete_right()`	$\times n$
Array	1	1	n	n	n	~ 1
Linked List	n	1	n	1	1	~ 3
Dynamic Array	1	1	n	n	$1_{(a)}$	~ 4

Dynamic Arrays

Computer memory is a finite resource. On modern computers many processes may share the same main memory store, so an operating system will assign a fixed chunk of memory addresses to each active process. The amount of memory assigned depends on the needs of the process and the availability of free memory. For example, when a computer program makes a request to store a variable, the program must tell the operating system how much memory (i.e. how many bits) will be required to store it. To fulfill the request, the operating system will find the available memory in the process's assigned memory address space and reserve it (i.e. allocate it) for that purpose until it is no longer needed. Memory management and allocation is a detail that is abstracted away by many high level languages including Python, but know that whenever you ask Python to store something, Python makes a request to the operating system behind-the-scenes, for a fixed amount of memory in which to store it.

Now suppose a computer program wants to store two arrays, each storing ten 64-bit words. The program makes separate requests for two chunks of memory (1024 bits each), and the operating system fulfills the request by, for example, reserving the first ten words of the process's assigned address space to the first array A, and the second ten words of the address space to the second array B. Now suppose that as the computer program progresses, an eleventh word w needs to be added to array A. It would seem that there is no space near A to store the new word: the beginning of the process's assigned address space is to the left of A and array B is stored on the right. Then how can we add w to A? One solution could be to shift B right to make room for w, but tons of data may already be reserved next to B, which you would also have to move. Better would be to simply request eleven new words of memory, copy A to the beginning of the new memory allocation, store w at the end, and free the first ten words of the process's address space for future memory requests.

The above operation requires linear time with respect to the length of array A. Is there a 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 Python code implementing `insert_right` (i.e., Python list `append`) and `delete_right` (i.e., Python list `pop`) for a custom `DynamicArray` class 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 forth 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 DynamicArray:
    def __init__(self):
        self._items = [None]
        self._n = 0

    def __len__(self):
        return self._n

    def __iter__(self):
        for i in range(self._n):
            yield self._items[i]

    def _rebuild(self, n):
        temp = [None] * n             # allocation of a fixed amount of space
        for i in range(self._n):
            temp[i] = self._items[i]
        self._items = temp

    def at(self, i):
        return self._items[i]

    def set_at(self, i, x):
        self._items[i] = x

    def insert_right(self, x):
        if self._n == len(self._items):
            self._rebuild(2 * self._n)
        self._items[self._n] = x
        self._n += 1

    def delete_right(self):
        if self._n < 1:
            raise IndexError('pop from empty list')
        if 4 * self._n < len(self._items):
            self._rebuild(2 * self._n)
        self._n -= 1
```

Set Interface

Data Structure	Operation, Worst Case $O(\cdot)$							Space
	Static	Dynamic		Order			Dyn. Order	$\sim \times n$
	`find(k)`	`insert(x)`	`delete(k)`	`next(k)`	`min()`	`max()`	`delete_max()`	
Unsorted Array	n	n	n	n	n	n	n	1
Linked List	n	1	n	n	n	n	n	3
Dynamic Array	n	$1_{(a)}$	n	n	n	n	n	4
Sorted Array	$\log n$	n	n	$\log n$	1	1	n	1
Min-Heap	n	$\log n_{(a)}$	n	n	1	n	n	1
Max-Heap	n	$\log n_{(a)}$	n	n	n	1	$\log n$	1
Balanced BST	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	5
Direct Access	1	1	1	u	u	u	u	u/n
Hash Table	$1_{(e)}$	$1_{(a,e)}$	$1_{(a,e)}$	n	n	n	n	4