Dynamic Arrays
Arrays: Pros and Cons

• Pro: only core data structure designed to hold a collection of elements

• Pro: random access: can quickly get to any element $\rightarrow O(1)$

• Con: fixed size:
  – Maximum number of elements must be specified when created
The dynamic array (called Vector or ArrayList in Java, same thing, different API) gets around this by **encapsulating a partially filled array that can grow when filled**

- Hide memory management details behind a simple API
- Is still randomly accessible, but now it grows as necessary
Unlike arrays, a dynamic array can change its capacity

- *Size* is logical collection size:
  - Current number of elements in the dynamic array
  - What the programmer thinks of as the size of the collection
  - Managed by an internal data value

- *Capacity* is physical array size: # of elements it can hold before it must resize
Partially Filled Dynamic Array

data = [empty]
size = 10
cap = 16

Size (size)
Capacity (cap)
Adding an element

• Adding an element to end is usually easy — just put new value at end and increment the (logical) size

• What happens when size reaches capacity?
Before reallocation:

\[
\begin{align*}
\text{data} &= \quad \\
\text{size} &= 8 \\
\text{cap} &= 8
\end{align*}
\]

Must allocate new (larger) array and copy valid data elements

Also...don’t forget to free up the old array

After reallocation:
• Adding an element to middle can also force reallocation (if the current size is equal to capacity)

• But will ALWAYS require that elements be moved to make space
  – Our partially filled array should not have gaps so that we always know where the next element should go

• Adding to anywhere other than end is therefore \(O(n)\) worst case
Adding to Middle (cont.)

Must make space for new value

Be Careful!

Loop from bottom up while copying data

Before

Add at idx

After

idx →

idx →
Removing an Element

• Removing an element will also require “sliding over” to delete the value
  – We want to maintain a contiguous chunk of data so we always know where the next element goes and can put it there quickly!

• Therefore is $O(n)$ worst case
Remove Element

Remove also requires a loop
This time, should it be from top (e.g. at idx) or bottom?

Remove idx

Before

After
• `realloc()` can be used in place of `malloc()` to do resizing and *may* avoid ‘copying’ elements if possible
  – It’s still $O(n)$ when it fails to enlarge the current array!
• For this class, use `malloc` only (so you’ll have to copy elements on a resize)
Something to think about...

• In the long term, are there any potential problems with the dynamic array?
  – hint: imagine adding MANY elements in the long term and potentially removing many of them.
Amortized Analysis

• What’s the cost of adding an element to the end of the array?

Here?  

Here?
Amortized Analysis

• To analyze an algorithm in which the worst case only occurs seldomly, we must perform an amortized analysis to get the average performance

• We’ll use the Accounting or *Banker’s Method*
Banker’s Method

- Assign a cost $c'_i$ to each operation.
- When you perform the operation, if the actual cost $c_i$, is less, then we save the credit $c'_i - c_i$ to hand out to future operations.
- Otherwise, if the actual cost is more than the assigned cost, we borrow from the saved balance.
- For $n$ operations, the sum of the total assigned costs must be $\geq$ sum of actual costs

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$
### Example – Adding to Dynamic Array

<table>
<thead>
<tr>
<th>Add Element</th>
<th>Old Capacity</th>
<th>New Capacity</th>
<th>Copy Count</th>
<th>$c'_i$</th>
<th>$c_i$</th>
<th>$b_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>$(3-1) = 2$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>$(1+1) = 2$</td>
<td>$(5-2) = 3$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>$(2+1) = 3$</td>
<td>$(6-3) = 3$</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td>1</td>
<td></td>
<td>$(6-1) = 5$</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$c_i = \text{actual cost} = \text{insert (1)} + \text{copy cost (1)}$

$b_i = \text{bank account }_i = \text{bankaccount}_{i-1} + \text{current deposit} - \text{actual cost} = (b_{i-1} + c'_i) - c_i$

We say the add() operation is therefore $O(1^+) – \text{amortized constant cost!}$
Imagine you’re starting with a partially filled array of size n. It already has n/2 elements in it. For each element you add we’ll put a cost of 3 in the bank.

1: to assign each of the new n/2 elements into the array

1: to copy each of the new n/2 elements into a larger array when it’s time to resize

1: to copy the other n/2 that were already in the array