Dynamic Arrays
Abstraction

• Abstract data types
• e.g. class or template class
• Hide implementation details
• Leaves simple interface for accessing
  – Easier comparison of data structures
• In array-type structures:
  – Insert
  – Delete
  – Access
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
Dynamic Array (Vector or ArrayList)

- The dynamic array 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
Capacity
\( = \text{cap} \)

Size
\( = \text{size} \)

Partially Filled Dynamic Array
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:

<table>
<thead>
<tr>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>cap</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

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

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

After reallocation:

How much bigger should we make it?
Before reallocation:

- data = [ ]
- size = 8
- cap = 8

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

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

Big-Oh of insertion?
- Best Case
- Worst Case
Adding to Middle

Must make space for new value

Be Careful!

Loop from bottom up while copying data

Big-Oh?
Best Case
Worst Case

Add at $\text{idx}$
Remove also requires a loop
This time, should it be from top (e.g. at idx) or bottom?

Remove \textbf{idx}

Before

\textbf{idx} \rightarrow

After

Big-Oh?
Best Case
Worst Case
Side Note

- `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?
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

• If an operation requires a constant number of operations on average, then we say it has complexity $O(1^+)$ – amortized constant cost!
Intuition for Amortized Analysis

• Consider inserting \( n \) elements into a dynamic array.

• All inserts are \( O(1) \) except when there is a resize.
  – So we would like to ensure only a small number of resize operations relative to the number of inserts
  – If the number of resizes is very small compared to \( n \), then the average runtime is dominated by the basic insertion operations and not the resize operations
Intuition for Amortized Analysis

Suppose we always double the capacity after a resize.

What is the maximum number of resize (and copy) operations for n inserts?

\[ \text{# of resizes} \leq \log(n) \]

So, #of resizes is very small compared to n!
Detailed Amortized Analysis

Suppose that copying an element to a memory location requires $O(1)$ time.

We must copy each element the first time it is inserted and then again for each of the following $\log(n)$ resize operations.

What is the max number of copies for inserting $n$ elements?

# of copies for first insertions = $n$

# of copies due to all resize

\[
\begin{align*}
\text{# of copies due to all resize} &= (\text{# for last resize}) + (\text{# for second to last}) + \ldots + (\text{# for first}) \\
&\leq n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots \\
&= n \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots\right)
\end{align*}
\]
Detailed Amortized Analysis

# of copies for first insertions = n

# of copies due to all resizes

= (# for last resize) + (# for second to last) + ….. + (# for first)
≤ n + n/2 + n/4 + n/8 ..... 
= \( n \cdot \left( 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots \right) \)
≤ n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} 
= n \cdot 2

Adding these together we get  
\( n + 2n = 3n = O(n) \) operations

So we do O(n) work for n insertions. Average is n/n = 1
Thus O(1+)
Banker’s Method (Alternative Approach)

- 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>0</td>
<td>3</td>
<td>1</td>
<td>(6-1) = 5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>3</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>

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 (just doubled 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