Dynamic Arrays
Computer Memory

- Lowest level: sequence of bytes
- Each byte has a 32-bit or 64-bit address
- Every byte is equally easy to access
  - "Random access" memory

...
Arrays

- Finite sequence of uniformly-sized segments
  - Starting address, item size (in bytes), item count (fixed)
- Each location is a **cell** located at a zero-based **index** offset from the start
  - Address of cell \( i \) is \( \text{start} + (i \times \text{item\_size}) \)

![Diagram showing arrays with a starting address, item size, and item count.](diagram.png)
Array Allocation

- **Stack (C)**
  - `int my_array[n]`
- **Heap (C)**
  - `my_array = (int*)malloc(n*sizeof(int))`
- **Heap (Java)**
  - `my_array = new int[n]`
Dynamic Arrays

- Goal: Add items to an array
- Issue: Arrays are fixed-length
Dynamic Arrays

- Goal: Add items to an array
- Issue: Arrays are fixed-length
- Naive solution: Resize the array's memory
  - Problem: no guarantee that we can do this!
Dynamic Arrays

• Goal: Add items to an array
• Issue: Arrays are fixed-length
• Naive solution: Resize the array's memory
  – Problem: no guarantee that we can do this!
• More robust solution: Dynamic arrays
  – Allocate more space than currently needed
  – Re-allocate and copy when the original size is exceeded
Dynamic Arrays

original: 0 1 2 3

1 □ □ □
### Dynamic Arrays

Original:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic Arrays

original: 

```
  0  1  2  3
original: 1 1 2
```
Dynamic Arrays

original:

0  1  2  3

1  1  2  3
Dynamic Arrays

original:

0 1 2 3
1 1 2 3
Dynamic Arrays

original: 1 1 2 3

new_arr: [empty]
Dynamic Arrays

original:  
0 1 2 3

new_arr:  
0 1 2 3 4 5 6 7
Dynamic Arrays

original: [1, 1, 2, 3]

new_arr: [1, 1, 2, 3]
Dynamic Arrays

new_arr: 1 1 2 3
0 1 2 3 4 5 6 7
Dynamic Arrays

new_arr: 1 1 2 3 5
0 1 2 3 4 5 6 7
Dynamic Arrays

● State information:
  – **array**: array pointer
  – **capacity**: current maximum element count
  – **size**: current element count

● Invariant: **capacity** >= **size**
Dynamic Arrays

- How big should we initialize new arrays?
  - For now let's make it big enough for a single element

- How much extra space should we allocate when we need to resize it?
  - For now, let's assume we double the size
**Dynamic Arrays**

typedef struct dynarray {
    int *array;
    size_t capacity;
    size_t size;
} dynarray_t;

void dynarray_append(dynarray_t *a, int value) {
    if ((a->size + 1) > a->capacity) {
        // allocate new storage array
        int *new_array = (int *)malloc(sizeof(int) * (a->capacity*2));
        // TODO: check new_array for NULL

        // copy old elements over
        for (int i = 0; i < a->size; i++) {
            new_array[i] = a->array[i];
        }

        // deallocate old storage array
        free(a->array);

        // update state information
        a->array = new_array;
        a->capacity *= 2;
    }
    // add new element
    a->array[a->size++] = value;
}
Dynamic Arrays

• Big-O analysis
  – Create empty array:
  – Access element:
  – Modify element:
  – Get length:
  – Append element: ???
Dynamic Arrays

• Big-O analysis
  – Create empty array: $O(1)$
  – Access element: $O(1)$
  – Modify element: $O(1)$
  – Get length: $O(1)$
  – Append element: ???
    • Let's measure cost in "copy operations"
Dynamic Arrays

- Big-O analysis
  - Create empty array: $O(1)$
  - Access element:  $O(1)$
  - Modify element:  $O(1)$
  - Get length:  $O(1)$
  - Append element:
    - If $\text{capacity} > \text{size}$:  $O(1)$
    - If $\text{capacity} == \text{size}$:  $O(n)$
Dynamic Arrays

• Big-O analysis
  – Create empty array: O(1)
  – Access element:  O(1)
  – Modify element:  O(1)
  – Get length: O(1)
  – Append element:
    • If capacity > size:  O(1)
    • If capacity == size:  O(n)

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>cap</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>ops</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>
Dynamic Arrays

- Can we argue that the average cost of the append operation is $O(1)$, despite its occasional $O(n)$ cost?
- Yes! Use *amortized* analysis
  - Sometimes called the "accounting method"
- Basic idea: charge algorithm $$ to perform operations
  - Overcharge for some (inexpensive) operations
  - Use saved $$ to pay for expensive operations
  - Show that the total $$ spent is $O(n)$ for $n$ operations
  - Thus, each operation can be considered $O(1)$
Amortized Analysis

- Intuition: Cost of rare expensive operations grows inversely proportionally to frequency.
Amortized Analysis

- Intuition: Cost of rare expensive operations grows inversely proportionally to frequency
Amortized Analysis

- Intuition: Cost of rare expensive operations grows inversely proportionally to frequency
Amortized Analysis

• Idea: Charge extra for $O(1)$ insertions to “save up” and “pay for” the $O(n)$ insertions
Amortized Analysis

• Idea: Charge extra for O(1) insertions to “save up” and “pay for” the O(n) insertions
Amortized Analysis

- How much extra do we charge?
  - Let's try charging 1 extra operation
  - Total of 2 operations per append
Amortized Analysis

• How much extra do we charge?
Amortized Analysis

- How much extra do we charge?

Need 1; have 2; save one!
Amortized Analysis

- How much extra do we charge?
Amortized Analysis

- How much extra do we charge?

Need 3; have 2+1=3
Amortized Analysis

- How much extra do we charge?

Need 1; have 2; save one!
Amortized Analysis

• How much extra do we charge?

Need 5; have 2+1=3  !!
Amortized Analysis

- How much extra do we charge?

Need 1, have 2; save one each!
Amortized Analysis

- How much extra do we charge?

Need 9, have 2+3=5  !!
Amortized Analysis

- How much extra do we charge?

Need 17, have 2+7=9 !!
Amortized Analysis

- How much extra do we charge?
  - Let's try charging 2 extra operations
  - Total of 3 operations per append
Amortized Analysis

- How much extra do we charge?

Need <=3; have 3
Amortized Analysis

- How much extra do we charge?

Need 1; have 3; save two!
Amortized Analysis

- How much extra do we charge?

Need 5; have 3+2=5
Amortized Analysis

- How much extra do we charge?

Need 1, have 3; save two each!
Amortized Analysis

- How much extra do we charge?

Need 9, have 3+6=9
Amortized Analysis

- How much extra do we charge?

Need 17, have 3+14=17
Amortized Analysis

- How much extra do we charge?

assume we're here
Amortized Analysis

• How much extra do we charge?

we'll add another n elements before the next "big" operation
Amortized Analysis

- How much extra do we charge?
Amortized Analysis

- How much extra do we charge?

so we should charge 2 extra for each of the $n$ elements in between
Amortized Analysis

• How much extra do we charge?
  – If we're doubling the size each time...
    • We will need to make $2n$ copies at the next increase
    • We will have $n$ new appends during that period
  – So we need to “save up” two extra operations per cheap append to pay for the expensive appends
  – Charge 3 total operations for each append
Amortized Analysis

- Total # of operations to add n items: 3n
  - Which is O(n)
- Average operations per append = 3n/n = 3
- More generally: the total # of operations is O(n), so the amortized cost per append is O(1)
Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?
Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?
Amortized Analysis

• Does the same argument apply to a constant increase when the capacity is reached?
Amortized Analysis

- Does the same argument apply to a constant increase when the capacity is reached?
Amortized Analysis

• Does the same argument apply to a constant increase when the capacity is reached?
  – No! The amount of operations “saved” is always constant between increases, but the amount of work done by the capacity increases grows linearly with the size of the array.
  – This actually leads to $\Omega(n^2)$ total operations for $n$ appends, instead of $O(n)$ total operations.
Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?
Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?
Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?
Amortized Analysis

- Does the same argument apply to a tripling increase when the capacity is reached?
  - Yes! Charge three extra operations instead of two, and then we will have saved roughly 3n operations before the next capacity increase.
  - Total operations for n appends: 4n ∈ O(n)
    - The amortized cost for each append is still O(1)
  - In fact, the argument works for any geometric progression
Amortized Analysis

• Fundamental idea: Overcharge for cheap operations to “save up” credit for expensive operations
  - If the total cost for $n$ operations can be shown to be $O(n)$, then the average cost for each individual operation is $O(1)$

• For PA2, you will use a dynamic array to implement the Set ADT