Carleton University

Array-based lists

part 1

Alina Shaikhet

Array-based implementations

of the **List** and **Queue** interfaces

	get(i) / set(i,x)	add(i,x) / remove(i)
ArrayStack	$\boldsymbol{O(1)}$	$\boldsymbol{O(1+n-i)}$
ArrayDeque	$\boldsymbol{O(1)}$	$\boldsymbol{O(1+\min\{i,n-i\})}$
DualArrayDeque	$\boldsymbol{O(1)}$	$\boldsymbol{O(1+\min\{i,n-i\})}$
RootishArrayStack	$\boldsymbol{O(1)}$	$\boldsymbol{O(1+n-i)}$

Interface - List

List – represents an indexed sequence of elements

size()	– returns the number of elements on the list $(\boldsymbol{n})$	$\boldsymbol{O(1)}$
isEmpty()	– returns whether list is empty	
get($\boldsymbol{i}$)	– returns the element at position $\boldsymbol{i}$	
set($\boldsymbol{i}, \boldsymbol{x}$)	– update the element at position $\boldsymbol{i}$ to be $\boldsymbol{x}$	
add($\boldsymbol{i}, \boldsymbol{x}$)	– add the element $\boldsymbol{x}$ to position $\boldsymbol{i}$	$\boldsymbol{O(n-i+1)}$ amortized time
remove($\boldsymbol{i}$)	– remove element at position $\boldsymbol{i}$	

The Java Collections Framework documentation about List interface:
https://docs.oracle.com/javase/8/docs/api/java/util/List.html

ArrayStack (aka ArrayList)

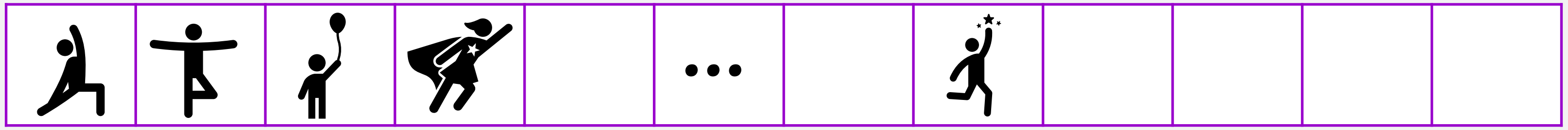
ODS JCF

ArrayStack is an array-based List implementation. It is equivalent to the **ArrayList** in the Java Collections Framework (JCF). It has the same performance.

Array $\boldsymbol{a}$ of type **T** and of size $\boldsymbol{a}.\mathbf{length}$

T[] $\boldsymbol{a}$ →

empty



$\mathbf{0}$ $\mathbf{1}$ $\mathbf{2}$ $\mathbf{3}$ $\boldsymbol{\cdots}$ $\boldsymbol{n-1}$

int $\boldsymbol{n}$ ← size of the list (can be different from $\boldsymbol{a}.\mathbf{length}$)

char[] $\boldsymbol{a}$

0	1	2	3	4	5	6	7	8	9	10	11
$\boldsymbol{c}$	$\boldsymbol{b}$	$\boldsymbol{x}$	$\boldsymbol{y}$	$\boldsymbol{z}$	$\boldsymbol{q}$						

$\boldsymbol{n = 6}$

$\boldsymbol{a}.\mathbf{length} = \mathbf{12}$

ArrayStack

Constructor **ArrayStack()**: $\boldsymbol{a} = \textbf{new T}[\mathbf{1}];$
$\boldsymbol{n} = \mathbf{0};$

$\mathbf{0}$

notice, not $\boldsymbol{i} > \boldsymbol{a}.\mathbf{length}$

returns the element at position $\boldsymbol{i}$

T get($\boldsymbol{i}$):
if $(\boldsymbol{i} < \mathbf{0} \;||\; \boldsymbol{i} \geq \boldsymbol{n})$ then throw new IndexException;
return $\boldsymbol{a}[\boldsymbol{i}]$;

$\boldsymbol{O}(\mathbf{1})$

char[] $\boldsymbol{a}$

$\boldsymbol{c}$	$\boldsymbol{b}$	$\boldsymbol{x}$	$\boldsymbol{y}$	$\boldsymbol{z}$	$\boldsymbol{q}$						
$\mathbf{0}$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$	$\mathbf{4}$	$\mathbf{5}$	$\mathbf{6}$	$\mathbf{7}$	$\mathbf{8}$	$\mathbf{9}$	$\mathbf{10}$	$\mathbf{11}$

$\boldsymbol{n} = \mathbf{6}$

$\boldsymbol{a}.\mathbf{length} = \mathbf{12}$

ArrayStack

- saves the element at position $\boldsymbol{i}$;
- updates the element at position $\boldsymbol{i}$ to be $\boldsymbol{x}$;
- returns the element that was at position $\boldsymbol{i}$

T set(i, x):

if $(i < 0 \;||\; i \geq n)$ then throw new IndexException;
T $y = a[i]$;
$a[i] = x$;
return y;

$O(1)$

set($4, w$):

char[] a

0	1	2	3	4	5	6	7	8	9	10	11
c	b	x	y	$\cancel{z}$ w	q						

$n = 6$

$a.\text{length} = 12$

ArrayStack

returns the number of elements on the list ($\boldsymbol{n}$)

```
int size():
        return n;
```

$\boldsymbol{O(1)}$

we do not return $\boldsymbol{a.\text{length}}$

char[] $\boldsymbol{a}$

c	b	x	y	w	q						
0	1	2	3	4	5	6	7	8	9	10	11

$\boldsymbol{n = 6}$

$\boldsymbol{a.\text{length} = 12}$

ArrayStack

- check bounds
- if $\boldsymbol{n}$ is equal to $\boldsymbol{a}.\mathbf{length}$ then resize the array $\boldsymbol{a}$
- shift elements $\boldsymbol{i}$ through $\boldsymbol{n-1}$ up an index (to indices $\boldsymbol{i+1, \ldots, n}$)
- insert the element $\boldsymbol{x}$ to position $\boldsymbol{i}$
- increase $\boldsymbol{n}$ by $\mathbf{1}$ to reflect the update

$\boldsymbol{i}$ can be equal $\boldsymbol{n}$

$O(1+n-i)$

void **add($\boldsymbol{i}, \boldsymbol{x}$):**

if $(\boldsymbol{i} < \mathbf{0} \ || \ \boldsymbol{i} > \boldsymbol{n})$ then throw IndexException;
if $(\boldsymbol{n} + \mathbf{1} > \boldsymbol{a}.\mathbf{length})$ then **resize()**;
for $(\boldsymbol{j} = \boldsymbol{n};\ \boldsymbol{j} > \boldsymbol{i};\ \boldsymbol{j}--)$
  $\boldsymbol{a}[\boldsymbol{j}] = \boldsymbol{a}[\boldsymbol{j}-\mathbf{1}];$
$\boldsymbol{a}[\boldsymbol{i}] = \boldsymbol{x};$
$\boldsymbol{n}++;$

shift

we will see this function later

after a call to **resize()**, we can be sure that $\boldsymbol{a}.\mathbf{length} > \mathbf{n}$.

add($\mathbf{2}, \boldsymbol{h}$):

char[] $\boldsymbol{a}$

0	1	2	3	4	5	6	7	8	9	10	11
c	b	$\cancel{x}$ h	y	w	q						

$\boldsymbol{a}.\mathbf{length} = \mathbf{12}$

$\boldsymbol{n} = \cancel{6}\ 7$

ArrayStack

- check bounds;
- save the element at position $\boldsymbol{i}$;
- shift elements $\boldsymbol{i+1}$ through $\boldsymbol{n-1}$ down an index (to indices $\boldsymbol{i, \ldots, n-2}$);
- decrease $\boldsymbol{n}$ by $\mathbf{1}$;
- if $\boldsymbol{a}$ is $\mathbf{3}$ times longer than necessary then resize the array;
- return the element that was at position $\boldsymbol{i}$.

$\boldsymbol{O(1+n-i)}$

T remove($\boldsymbol{i}$):

if $(\boldsymbol{i<0} \;||\; \boldsymbol{i \geq n})$ then throw IndexException;
T $\boldsymbol{x = a[i]}$;
for $(\boldsymbol{j = i;\ j < n-1;\ j++})$
  $\boldsymbol{a[j] = a[j+1]}$; **shift**
$\boldsymbol{n--}$;
$\boldsymbol{a[n] = \mathbf{null}}$;
if $(\boldsymbol{3n \leq a.\mathbf{length}})$ then **resize()**;
return $\boldsymbol{x}$;

remove(3):

char[] $\boldsymbol{a}$

c	b	h	x	y	w	q					
0	1	2	3	4	5	6	7	8	9	10	11

$\boldsymbol{a.\mathbf{length} = 12}$

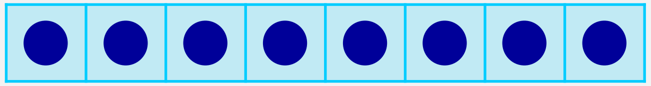
$\boldsymbol{n} = \cancel{7}\ 6$

Alina Shaikhet – COMP 2402 – Carleton University

ArrayStack – resize()

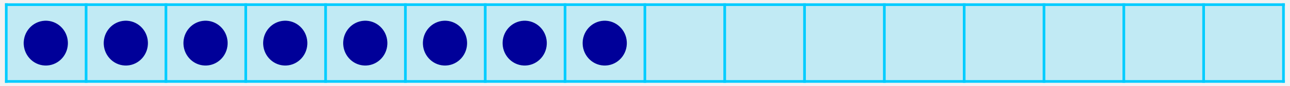
There are two situations that trigger **resize()**:

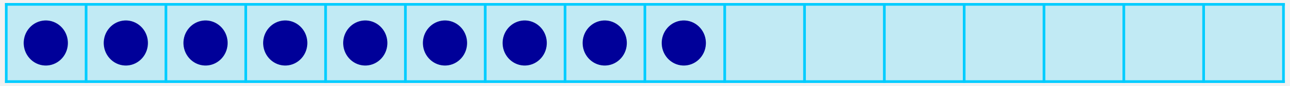
1. Array is full and we want to add another element



add(i, x):

resize():

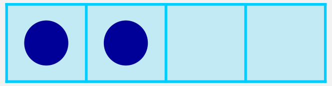




2. We remove an element, and the array becomes 2/3 empty

remove(i):

resize():



ArrayStack – resize()

to avoid 0-length array

void **resize()**:

T[] $\boldsymbol{b} =$ new array$(\mathbf{max}\{\mathbf{2}\boldsymbol{n}, \mathbf{1}\})$

executes $\boldsymbol{n}$ times

for $(\boldsymbol{i} = \mathbf{0}; \boldsymbol{i} < \boldsymbol{n};\ \boldsymbol{i}++)$

$\boldsymbol{b}[\boldsymbol{i}] = \boldsymbol{a}[\boldsymbol{i}];$

$\boldsymbol{a} = \boldsymbol{b};$

$\boldsymbol{O}(\boldsymbol{n})$

- create new array $\boldsymbol{b}$;
- copy everything from $\boldsymbol{a}$ to $\boldsymbol{b}$;
- the new array becomes $\boldsymbol{a}$.

ArrayStack – resize()

void **resize()**:

$\quad$ T[] $\boldsymbol{b} =$ new array of size $\boldsymbol{s}$
$\quad$ for $(\boldsymbol{i} = \mathbf{0}; \boldsymbol{i} < \boldsymbol{n};\ \boldsymbol{i}++)$
$\quad\quad$ $\boldsymbol{b}[\mathbf{i}] = \boldsymbol{a}[\mathbf{i}];$
$\quad$ $\boldsymbol{a} = \boldsymbol{b};$

- take $\boldsymbol{s} \gg \boldsymbol{n}$
  - \+ you never have to **resize()** again (fast!)
  - – wastes a lot of space
- take $\boldsymbol{s} = \boldsymbol{n} + \mathbf{1}$
  - \+ does not waste any space
  - – wastes too much time in **resize()** over many calls
- take $\boldsymbol{s} = \mathbf{max}\{\mathbf{2}\boldsymbol{n}, \mathbf{1}\}$
  - – wastes $\boldsymbol{O}(\boldsymbol{n})$ space
  - \+ gives us $\boldsymbol{O}(\mathbf{1})$ **amortized** run time

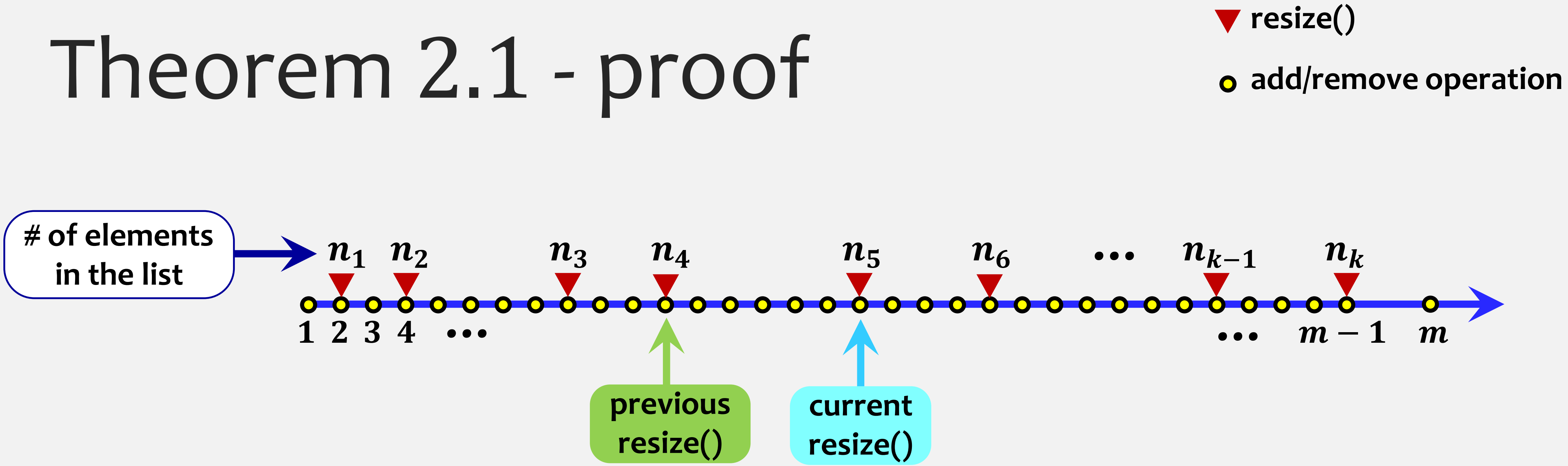
Theorem 2.1

An **ArrayStack** implements the **List** interface. Ignoring the cost of calls to **resize()**, an **ArrayStack** supports the operations

- **get**$(\boldsymbol{i})$ and **set**$(\boldsymbol{i}, \boldsymbol{x})$ in $\boldsymbol{O}(\mathbf{1})$ time per operation; and
- **add** $(\boldsymbol{i}, \boldsymbol{x})$ and **remove**$(\boldsymbol{i})$ in $\boldsymbol{O}(\mathbf{1} + \boldsymbol{n} - \boldsymbol{i})$ time per operation.

Furthermore, beginning with an empty **ArrayStack** and performing any sequence of $\boldsymbol{m}$ **add** $(\boldsymbol{i}, \boldsymbol{x})$ and **remove**$(\boldsymbol{i})$ operations results in a total of $\boldsymbol{O}(\boldsymbol{m})$ time spent during all calls to **resize()**.

Theorem 2.1 - proof



The total # of elements copied by all the calls to **resize()** is at most

$$n_1 + n_2 + n_3 + \cdots + n_k$$

Theorem 2.1 - proof

We resize in two cases:

- on **add**, when $\boldsymbol{n+1 > a.\mathrm{length}}$ (when $\boldsymbol{a.\mathrm{length} = n}$)
- on **remove**, when $\boldsymbol{3n \leq a.\mathrm{length}}$

We resize to $\boldsymbol{max\{2n, 1\}}$ (i.e., $\boldsymbol{2n}$ unless $\boldsymbol{n = 0}$)

Right after the **resize()** (in either of the two cases) our array is half full

$$\boldsymbol{\frac{a.\mathrm{length}}{2} - 1 \leq n \leq \frac{a.\mathrm{length}}{2}}$$

Suppose we have $\boldsymbol{m}$ calls to **add** $\boldsymbol{(i, x)}$ and **remove**$\boldsymbol{(i)}$. We want to show that the total time spent during all calls to **resize()** is $\boldsymbol{O(m)}$.

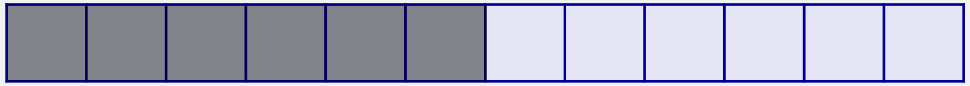
Theorem 2.1 - proof

$\boldsymbol{n}$ is the size of the list **NOW**

Consider two consecutive **resize()** operations: current $(\boldsymbol{j})$ and previous $(\boldsymbol{j-1})$.

There are no **resize()** operations in between. Therefore, the size of the array right after the $(\boldsymbol{j-1})$-st **resize()** and just before the $\boldsymbol{j}$-th **resize()** is the same.

array right after the $(\boldsymbol{j-1})$-st **resize()**



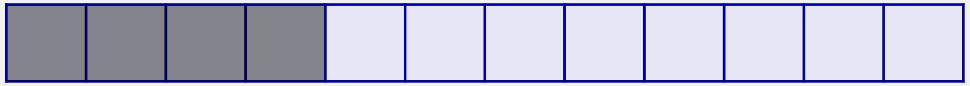
THEN

$$\frac{\boldsymbol{a}.\mathbf{length}}{\mathbf{2}} - \mathbf{1} \leq \boldsymbol{n} \leq \frac{\boldsymbol{a}.\mathbf{length}}{\mathbf{2}}$$

array just before the $\boldsymbol{j}$-th **resize()** (triggered by either add or remove operation):

1. **add** $(\boldsymbol{i}, \boldsymbol{x})$ and $\boldsymbol{a}.\mathbf{length} = \boldsymbol{n}$

2. **remove**$(\boldsymbol{i})$ and $\boldsymbol{a}.\mathbf{length} \geq \mathbf{3}\boldsymbol{n}$

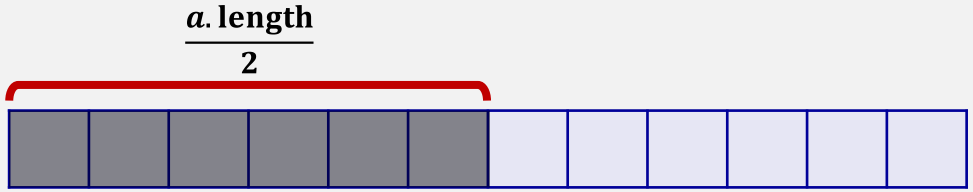


NOW

Theorem 2.1 - proof

n is the size of the list **NOW**

array right after the $(j-1)$-st **resize()**



THEN

array just before the j-th **resize()** (triggered by either add or remove operation):

1. **add** (i, x) and $a.\text{length} = n$

NOW

We added at least (# elements at j-th **resize()**) $-$ (# elements at $(j-1)$-st **resize()**)

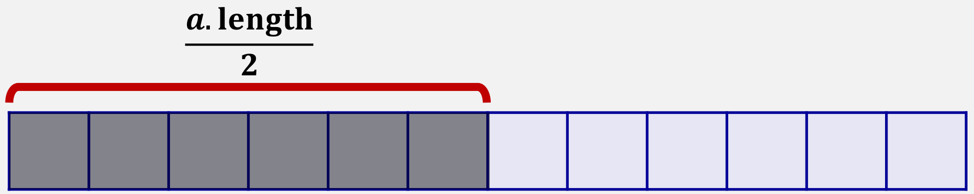
We added $\geq$ $a.\text{length} - \dfrac{a.\text{length}}{2} = \dfrac{a.\text{length}}{2} = \dfrac{n}{2}$

We have at least $n/2$ **add** (i, x) operations between these two **resize()** operations (maybe more if we also have **remove**(i) operations)

Theorem 2.1 - proof

$\boldsymbol{n}$ is the size of the list **NOW**

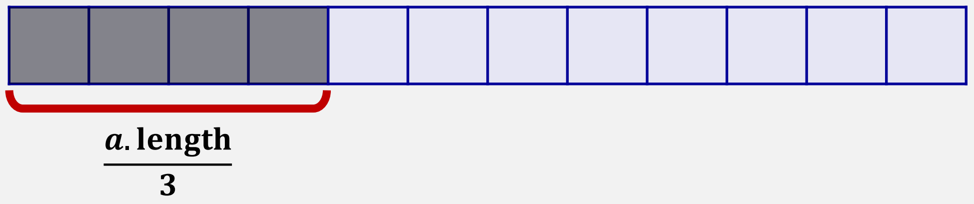
array right after the $(\boldsymbol{j}-\mathbf{1})$-st **resize()**



THEN

array just before the $\boldsymbol{j}$-th **resize()** (triggered by either add or remove operation):

2. **remove(**$\boldsymbol{i}$**)** and $\boldsymbol{a}.\mathbf{length} \geq \mathbf{3}\boldsymbol{n}$



NOW

We removed at least (# elements at $(\boldsymbol{j}-\mathbf{1})$-st **resize()**) – (# elements at $\boldsymbol{j}$-th **resize()**)

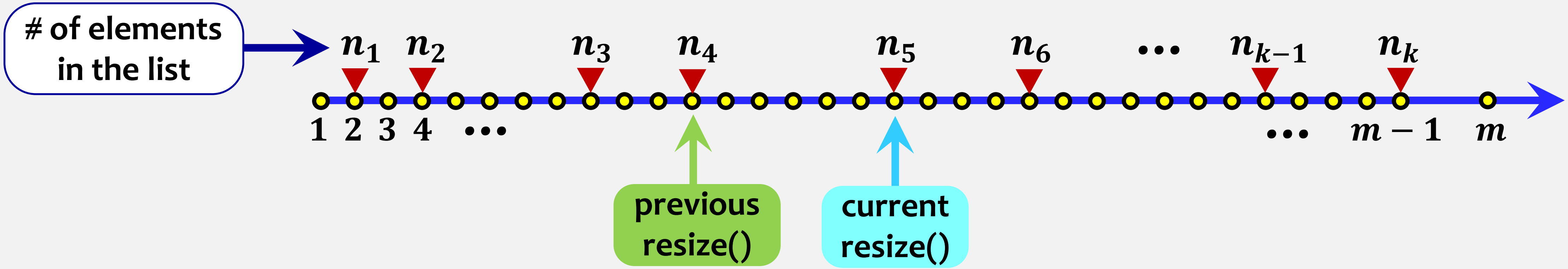
We removed $\geq \dfrac{a.\text{length}}{2} - 1 - \dfrac{a.\text{length}}{3} = \dfrac{a.\text{length}}{6} - 1 \geq \dfrac{3n}{6} - 1 = \dfrac{n}{2} - 1$

We have at least $\boldsymbol{n}/\mathbf{2}-\mathbf{1}$ **remove(**$\boldsymbol{i}$**)** operations between these two **resize()** operations (maybe more if we also have **add** $(\boldsymbol{i}, \boldsymbol{x})$ operations)

Theorem 2.1 - proof

resize()

add/remove operation



The total # of elements copied by all the calls to **resize()** is at most

$$n_1 + n_2 + n_3 + \cdots + n_k$$

$$m \geq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_3}{2} + \cdots + \frac{n_k}{2} = \frac{1}{2}\left(\begin{array}{c}\text{total \# of elements copied} \\ \text{by all the calls to } \textbf{resize()}\end{array}\right)$$

$$2m \geq \left(\begin{array}{c}\text{total \# of elements copied} \\ \text{by all the calls to } \textbf{resize()}\end{array}\right)$$

Total running time of all the calls to **resize()** is $O(m)$.

ArrayStack

The **ArrayStack** is an efficient way to implement a **Stack.**

In particular, we can implement:

- **push**($\boldsymbol{x}$) as **add**($\boldsymbol{n}, \boldsymbol{x}$)
- **pop**() as **remove**($\boldsymbol{n} - \mathbf{1}$)

in which case these operations will run in $\boldsymbol{O}(\mathbf{1})$ amortized time.

Why?

Recall:

add ($\boldsymbol{i}, \boldsymbol{x}$) and **remove**($\boldsymbol{i}$) run in $\boldsymbol{O}(\mathbf{1} + \boldsymbol{n} - \boldsymbol{i})$ time per operation (ignoring **resize()**).