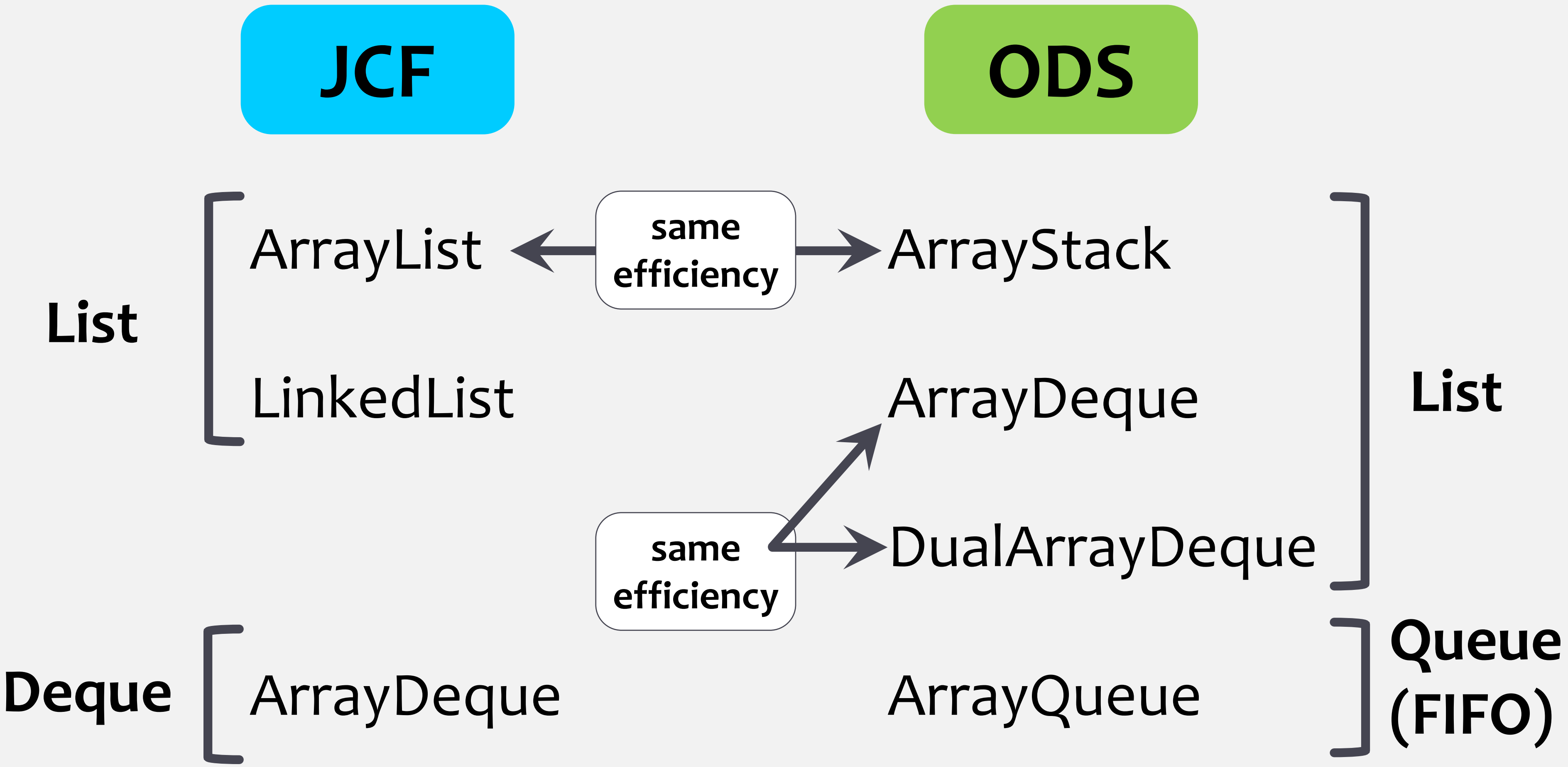
Carleton University

Array-based lists

part 3

Alina Shaikhet

Overview


JCF
ODS
List
ArrayList
LinkedList
same efficiency
ArrayStack
ArrayDeque
same efficiency
DualArrayDeque
List
Deque
ArrayDeque
ArrayQueue
Queue (FIFO)

ListSpeed.java

Adds n elements to the **end** of the list and then removes all the elements from the **end**.

Data Structures tested:

- ArrayList
- ArrayStack
- FastArrayStack
- LinkedList
- ArrayDeque
- DualArrayDeque

ods

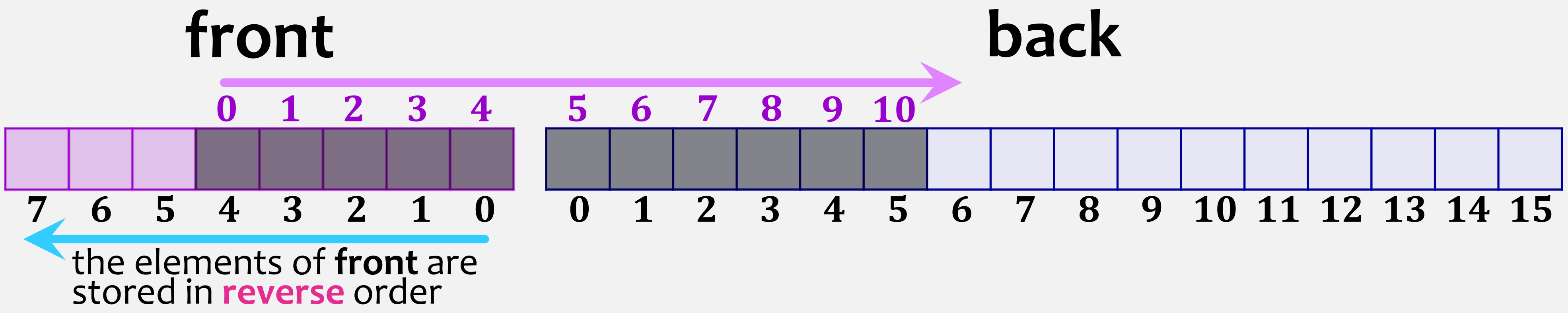
JCF

```
javac ListSpeed.java
java ListSpeed 100000000
```

For the last three DSs we also test adding/removing to the **front** of the list

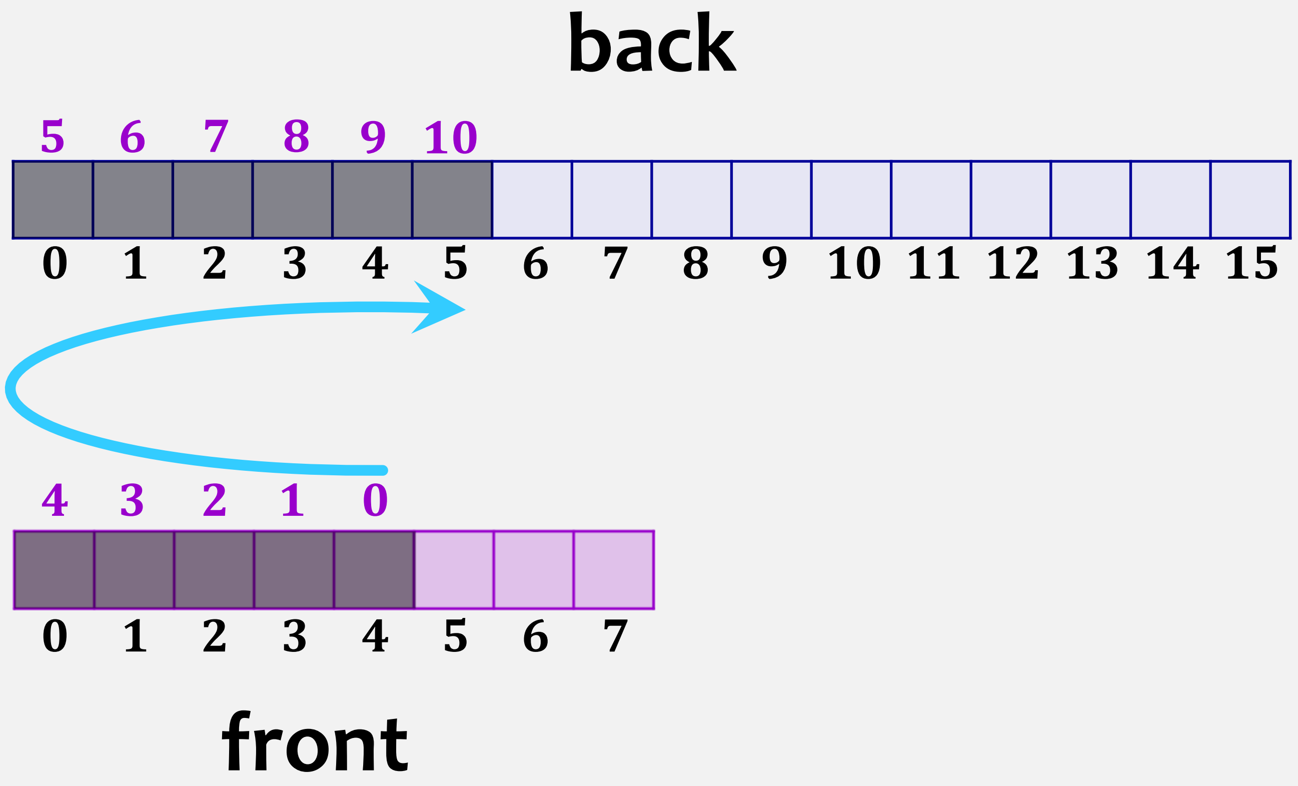
DualArrayDeque

DualArrayDeque implements the **List** interface using two **ArrayStacks**.



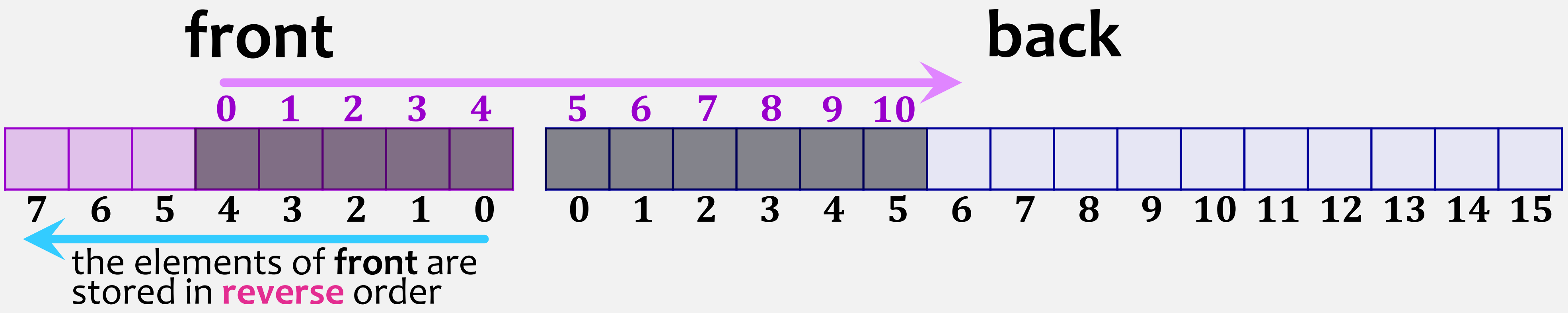
DualArrayDeque

DualArrayDeque implements the **List** interface using two **ArrayStacks**.



DualArrayDeque

DualArrayDeque implements the **List** interface using two **ArrayStack**s.

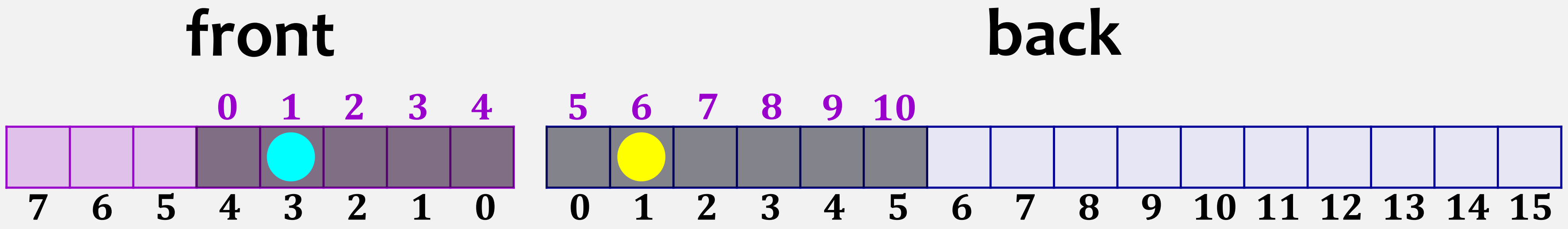


ArrayStack is fast when the operations on it modify elements near the **end.**
DualArrayDeque places two **ArrayStack**s (**front** and **back**), back-to-back so that operations are fast at either end.

$$n = \mathbf{front}.\text{size}() + \mathbf{back}.\text{size}()$$

```
ArrayStack front;
ArrayStack back;
```

DualArrayDeque



Example:

front.size() **is 5**

back.size() **is 6**

get(6)

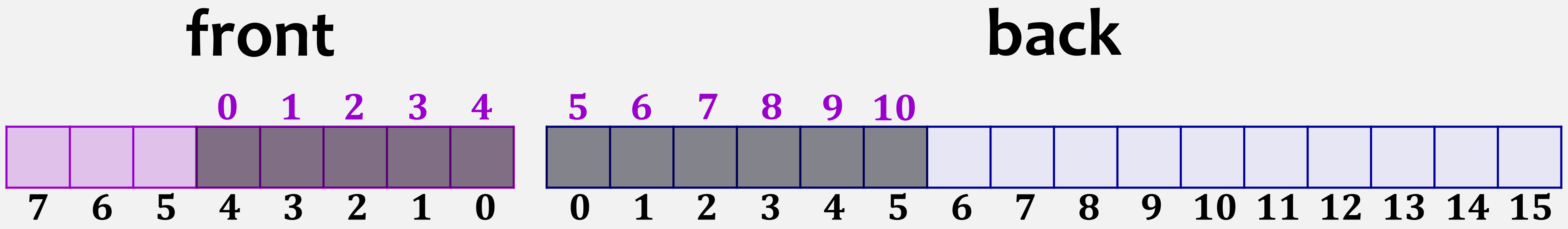
get(1)

size():
    return **front**.size() + **back**.size();

T get($\boldsymbol{i}$):
    if ($\boldsymbol{i}$ < **front**.size()) then
        return **front**.get(**front**.size() − $\boldsymbol{i}$ − **1**);
    else
        return **back**.get($\boldsymbol{i}$ − **front**.size());

$\boldsymbol{O(1)}$

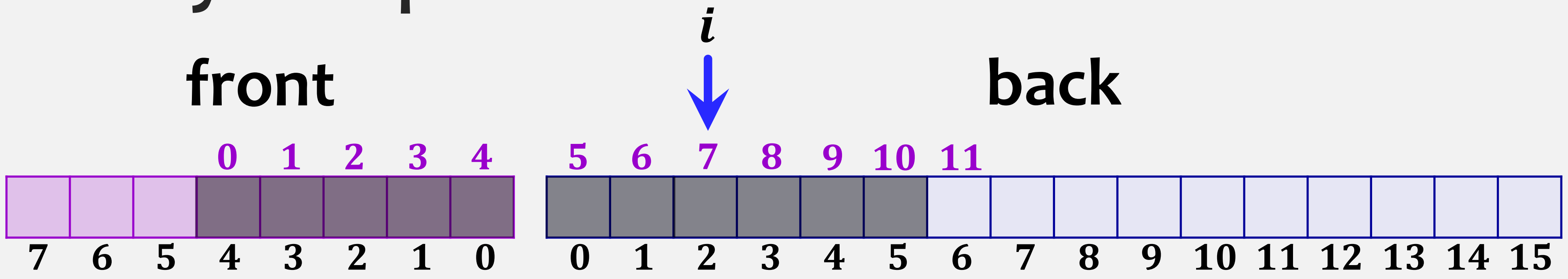
DualArrayDeque




```
T set(i, x):
    if (i < front.size()) then
        return front.set(front.size() – i – 1, x);
    else
        return back.set(i – front.size(), x);
```

$O(1)$

DualArrayDeque



Example:

$\left.\begin{array}{l}\textbf{add}(\mathbf{0}, \boldsymbol{x}) \\ \textbf{add}(\mathbf{11}, \boldsymbol{x})\end{array}\right\}$ fast

add$(\mathbf{7}, \boldsymbol{x})$

back.add$(\mathbf{2}, \boldsymbol{x})$

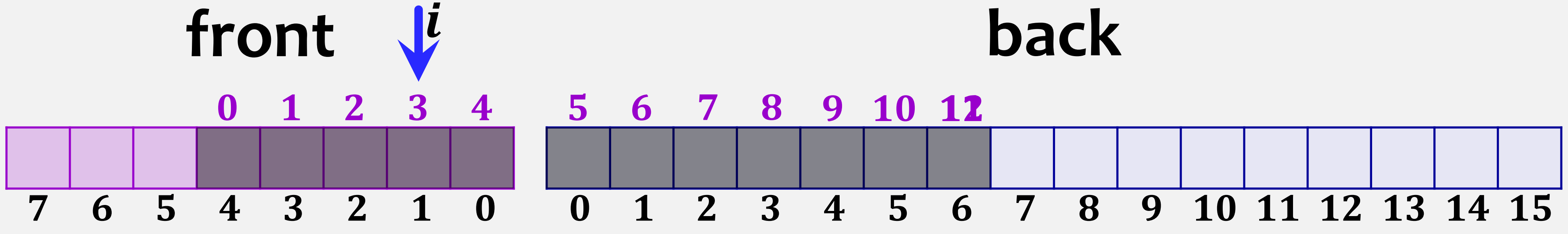
void add($\boldsymbol{i}$, $\boldsymbol{x}$):

  if ($\boldsymbol{i}$ < **front**.size()) then

  else

    back.add($\boldsymbol{i}$ − **front**.size(), $\boldsymbol{x}$);

DualArrayDeque



Example:

add$(3, x)$
$\shortparallel$
front.add$(2, x)$

balance() ensures that, unless size() < 2, **front**.size() and **back**.size() do not differ by more than a factor of 3.

void add(i, x):
    if (i < **front**.size()) then
        front.add(**front**.size() – i, x);
    else
        back.add(i – **front**.size(), x);
    balance();

$O(1 + \min(i, n - i))$

$O(i + 1)$

$O(n - i + 1)$

regular **ArrayStack** runtime of **add()**

Alina Shaikhet – COMP 2402 – Carleton University

add() – Analysis

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

regular **ArrayStack** runtime of **add()**

$\mathbf{front}.\text{add}(\mathbf{front}.\text{size}() - \boldsymbol{i},\ \boldsymbol{x});$

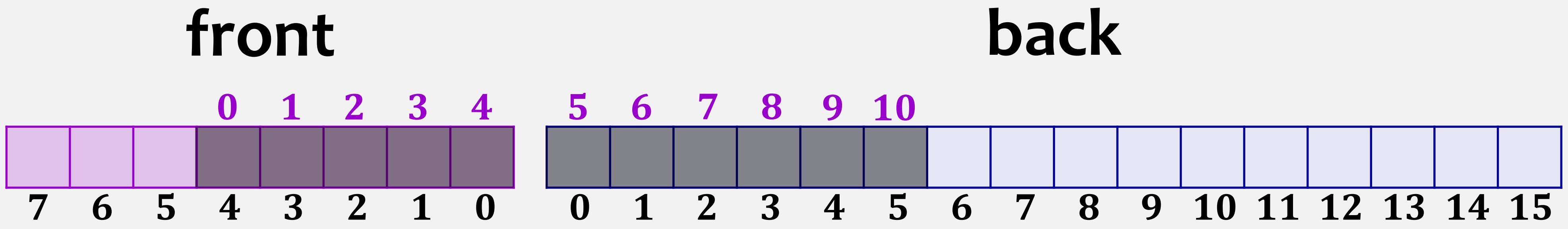
$$\boldsymbol{O}(\boldsymbol{size} - \boldsymbol{index} + \mathbf{1}) = \boldsymbol{O}(\mathbf{front}.\text{size}() - (\mathbf{front}.\text{size}() - \boldsymbol{i}) + \mathbf{1}) = \boxed{\boldsymbol{O}(\boldsymbol{i} + \mathbf{1})}$$

$\mathbf{back}.\text{add}(\boldsymbol{i} - \mathbf{front}.\text{size}(),\ \boldsymbol{x});$

$$\boldsymbol{O}(\boldsymbol{size} - \boldsymbol{index} + \mathbf{1}) = \boldsymbol{O}(\mathbf{back}.\text{size}() - (\boldsymbol{i} - \mathbf{front}.\text{size}()) + \mathbf{1}) =$$

$$\boldsymbol{O}(\mathbf{back}.\text{size}() + \mathbf{front}.\text{size}() - \boldsymbol{i} + \mathbf{1}) = \boxed{\boldsymbol{O}(\boldsymbol{n} - \boldsymbol{i} + \mathbf{1})}$$

DualArrayDeque

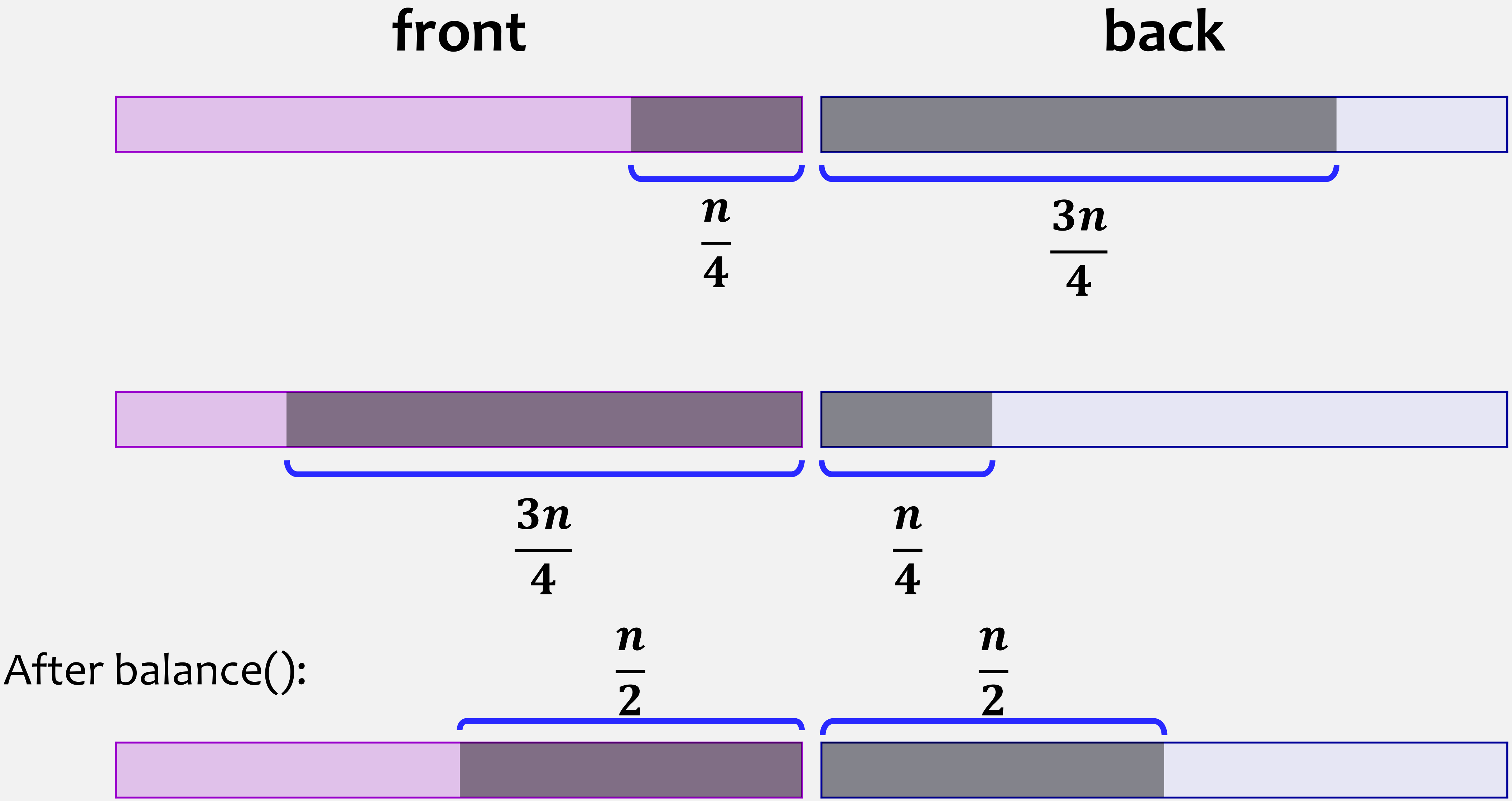


$O\left(1 + \min(i, n - i)\right)$

T remove(i):

T x;
if (i < **front**.size()) then
    x = **front**.remove(**front**.size() − i − 1);
else
    x = **back**.remove(i − **front**.size());
balance();
return x;

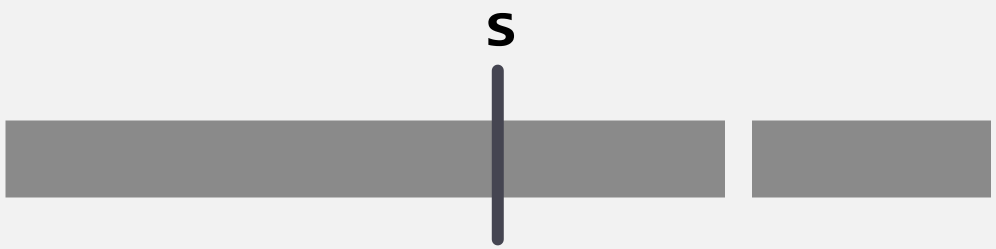
balance()

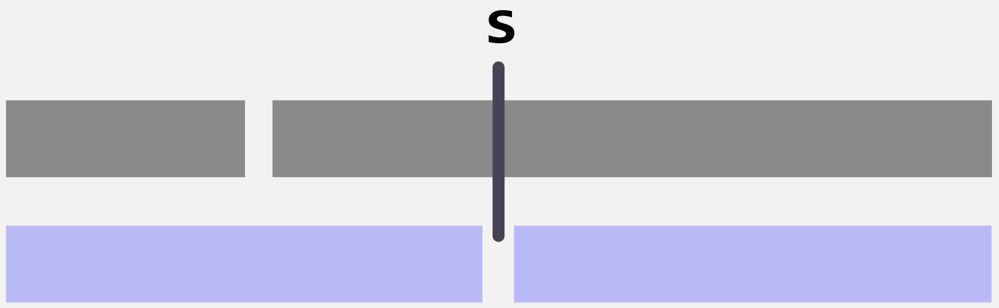

front
back
$\frac{n}{4}$
$\frac{3n}{4}$
$\frac{3n}{4}$
$\frac{n}{4}$
After balance():
$\frac{n}{2}$
$\frac{n}{2}$

DualArrayDeque.java

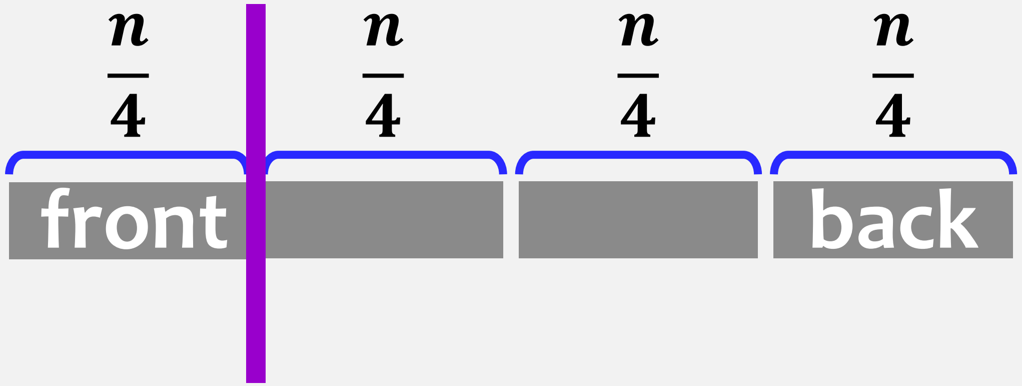
$O(n)$

```java
protected void balance() {
    int n = size();
    if (3*front.size() < back.size()) {
        int s = n/2 - front.size();
        List<T> l1 = newStack();
        List<T> l2 = newStack();
        l1.addAll(back.subList(0,s));
        Collections.reverse(l1);
        l1.addAll(front);
        l2.addAll(back.subList(s, back.size()));
        front = l1;
        back = l2;
    } else if (3*back.size() < front.size()) {
        int s = front.size() - n/2;
        List<T> l1 = newStack();
        List<T> l2 = newStack();
        l1.addAll(front.subList(s, front.size()));
        l2.addAll(front.subList(0, s));
        Collections.reverse(l2);
        l2.addAll(back);
        front = l1;
        back = l2;
    }
}
```





Analysis



List of $\boldsymbol{n}$ elements stored in a DualArrayDeque:

Consider add$(\boldsymbol{i}, \boldsymbol{x})$ and remove$(\boldsymbol{i})$ operations.

if $\boldsymbol{i} < \frac{\boldsymbol{n}}{\boldsymbol{4}}$ then $\boldsymbol{i}$ belongs to the **front** – add$(\boldsymbol{i}, \boldsymbol{x})$ takes $\boldsymbol{O(i+1)}$ time

if $\boldsymbol{i} \geq \frac{\boldsymbol{3n}}{\boldsymbol{4}}$ then $\boldsymbol{i}$ belongs to the **back** – add$(\boldsymbol{i}, \boldsymbol{x})$ takes $\boldsymbol{O(n-i+1)}$ time

if $\frac{\boldsymbol{n}}{\boldsymbol{4}} \leq \boldsymbol{i} < \frac{\boldsymbol{3n}}{\boldsymbol{4}}$ then $\boldsymbol{\min\{i, n-i\}} \geq \frac{\boldsymbol{n}}{\boldsymbol{4}}$ – add$(\boldsymbol{i}, \boldsymbol{x})$ takes $\boldsymbol{O(n) = O(i) = O(n-i)}$ time

```
void add(i, x):
    if (i < front.size()) then
        front.add(front.size() – i, x);
    else
        back.add(i – front.size(), x);
    balance();
```

$\boxed{\boldsymbol{O(i+1)}}$

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

runtime of add$(\boldsymbol{i}, \boldsymbol{x})$ if we ignore the cost of the call to balance():

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

Theorem 2.4

A **DualArrayDeque** implements the **List** interface. Ignoring the cost of calls to resize() and balance(), a **DualArrayDeque** supports the operations

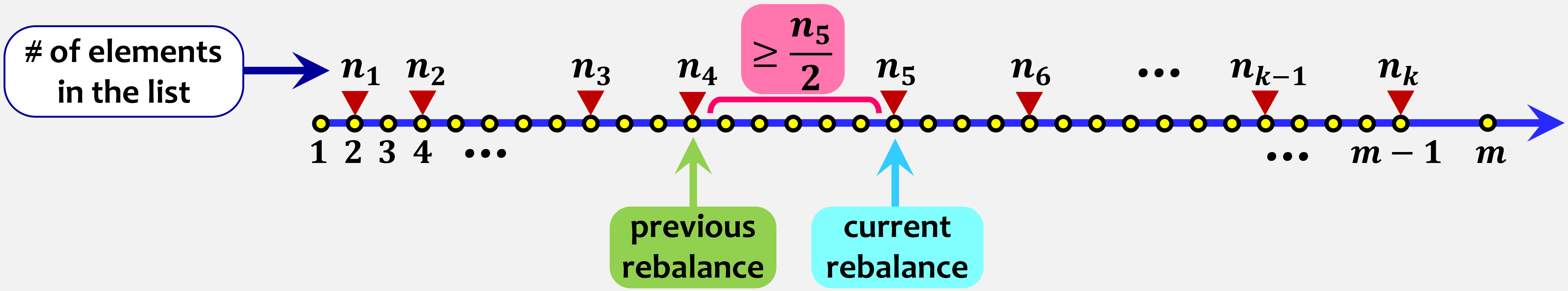
- get($\boldsymbol{i}$) and set($\boldsymbol{i}, \boldsymbol{x}$) in $\boldsymbol{O}(\mathbf{1})$ time per operation; and
- add($\boldsymbol{i}, \boldsymbol{x}$) and remove($\boldsymbol{i}$) in $\boldsymbol{O}\big(\mathbf{1} + \mathbf{min}(\boldsymbol{i}, \boldsymbol{n} - \boldsymbol{i})\big)$ time per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

Furthermore, beginning with an empty **DualArrayDeque**, 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() and balance().

Theorem 2.4 – proof

▼ actual rebalancing

● add/remove operation

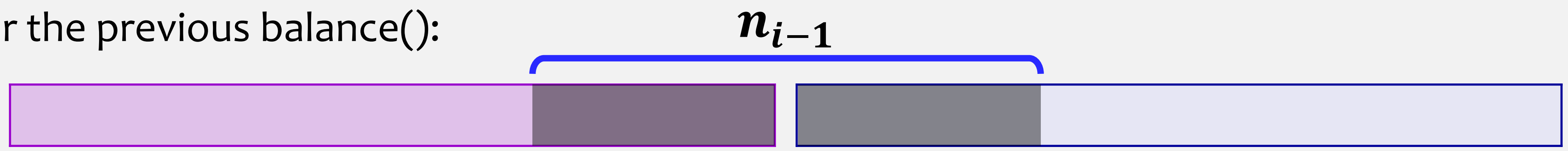


$$m \geq \frac{n_1}{2} + \frac{n_2}{2} + \frac{n_3}{2} + \cdots + \frac{n_k}{2} = \frac{1}{2}(n_1 + n_2 + n_3 + \cdots + n_k)$$

$$2m \geq n_1 + n_2 + n_3 + \cdots + n_k = \left(\begin{array}{c}\text{total number of elements}\\ \text{copied due to } \textbf{rebalancing}\end{array}\right)$$

Theorem 2.4 – proof

right after the previous balance():



$\vdots \quad \geq \dfrac{n_i}{2}$

right before the current balance():

$\geq \dfrac{n_i}{2}$