COMP2402
Abstract Data Types and Algorithms

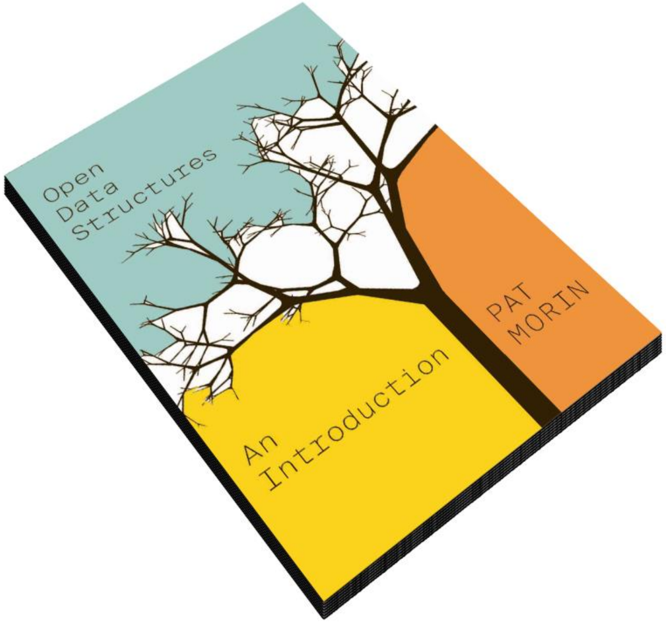
Dual-Array Deques and Rootish-Array Stacks

Reading Assignment

Open Data Structures in Java

by Pat Morin

Chapter 2.5, 2.6



Theorems, Revisited

the "ArrayStack"† Supports‡:

† the textbook name for a list backed by a non-circular array
‡ if we (momentarily) ignore the cost of resizing

- `get(i)` with Time Complexity O(1)
- `set(i)` with Time Complexity O(1)
- `add(i,o)` with Time Complexity O(1+n-i) ~ O(n)
- `remove(i)` with Time Complexity O(1+n-i) ~ O(n)

Starting from an Empty Array and Performing a Sequence of m `add` or `remove` Operations entails that the Time Complexity Associated with all Calls to `resize` is O(m)

ArrayStack can be an Efficient Implementation of Stack `push`/`pop` run in Constant Amortized Time

Theorems, Revisited

the "ArrayQueue"† Supports‡:

† *the textbook name for a list backed by a circular array*
‡ *if we (momentarily) ignore the cost of resizing*

- `get(i)` with Time Complexity O(1)
- `set(i)` with Time Complexity O(1)
- `add(i,o)` with Time Complexity O(1+n-i) ~ O(n)
- `remove(i)` with Time Complexity O(1+n-i) ~ O(n)

Starting from an Empty Array and Performing a Sequence of m `add` or `remove` Operations entails that the Time Complexity Associated with all Calls to `resize` is O(m)

ArrayQueue can be an Efficient Implementation of Queue `enqueue`/`dequeue` run in Constant Amortized Time

Theorems, Revisited

the "ArrayDeque"† Supports‡:

† *the textbook name for another list backed by a circular array*

‡ *if we (momentarily) ignore the cost of resizing*

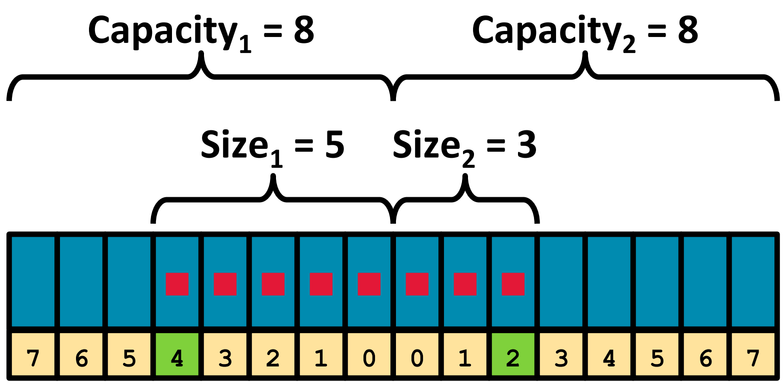
- `get(i)` with Time Complexity O(1)
- `set(i)` with Time Complexity O(1)
- `add(i,o)` with Time Complexity O(1+min{i,n-i})
- `remove(i)` with Time Complexity O(1+min{i,n-i})

Starting from an Empty Array and Performing a Sequence of m `add` or `remove` Operations entails that the Time Complexity Associated with all Calls to `resize` is O(m)

`addFirst`/`addLast`/`removeFirst`/`removeLast` all run in Constant Amortized Time with ArrayDeque

Dual Array Deque

an Alternative Approach to the Implementation of Deque
Uses Two Stacks instead of one circular array



Stack*$_1$*.push()
↓
addFirst()

***top*$_1$**
↓
Front of Deque

***top*$_2$**
↓
Back of Deque

Stack*$_2$*.push()
↓
addLast()

Stack*$_1$*.pop()
↓
removeFirst()

Stack*$_2$*.pop()
↓
removeLast()

Dual Array Deque

the Stack Containing the Deque Front is "$front$"
the Stack Containing the Deque Back is "$back$"

the Dual Array Deque must be Maintained such that

$$front.size \le 3 \cdot back.size$$

or

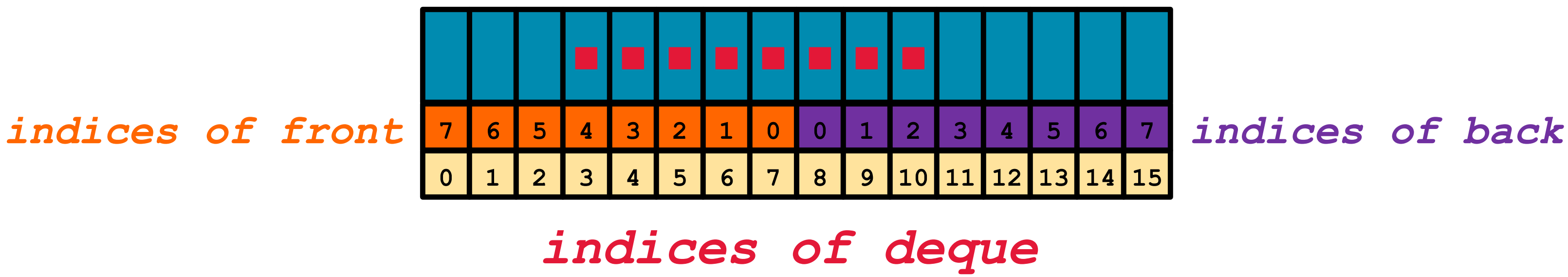
$$back.size \le 3 \cdot front.size$$

Whenever this Condition Fails, New Stacks are Created and the Elements are Divided Equally Between Them

`balance()`

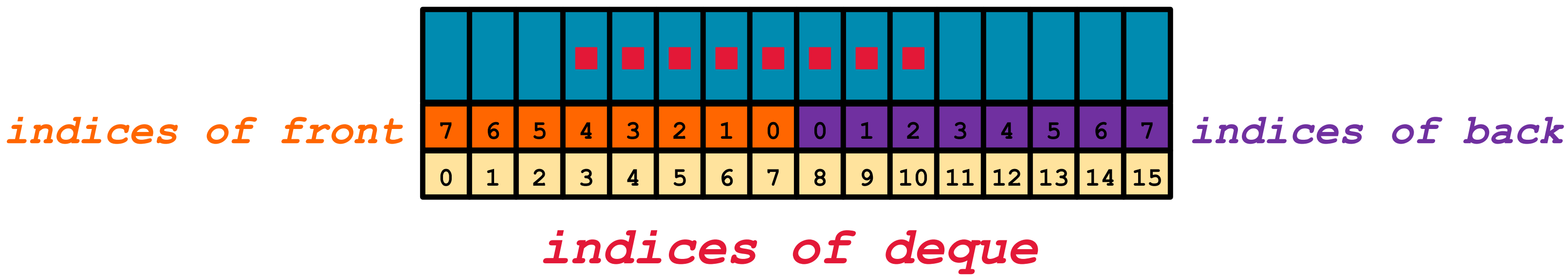
Inserting into a Dual Array Deque

(Momentarily) Ignoring the `balance` Cost, What is the Cost of Inserting an Element at Index i (of Deque)?



Inserting into a Dual Array Deque

(Momentarily) Ignoring the `balance` Cost, What is the Cost of Inserting an Element at Index i (of Deque)?



it Depends on the Value of i

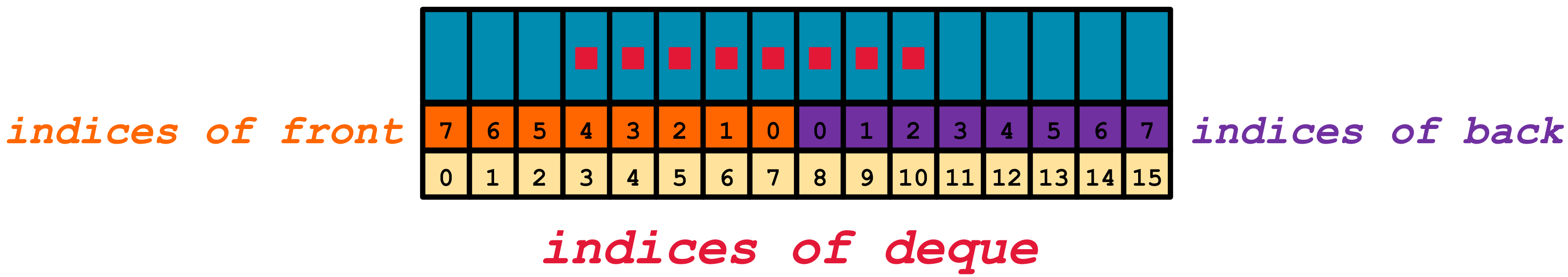
$$\mathbf{0 \leq i \leq \frac{n}{4}}$$

Index i is Near the Top of Front Stack?

Shift i Elements on Front → O(i) Cost

Inserting into a Dual Array Deque

(Momentarily) Ignoring the `balance` Cost, What is the Cost of Inserting an Element at Index i (of Deque)?



it Depends on the Value of i

$$\frac{3n}{4} \leq i \leq n$$

Index i is Near the Top of Back Stack?

Shift i Elements on Back → O(n-i) Cost

Inserting into a Dual Array Deque

$$\frac{n}{4} < i < \frac{3n}{4}$$

we Could Claim to Need to Shift Up to n Items

(i.e., the maximum number of items, since n is the size)

but a More Accurate Impression is Possible

Inserting into a Dual Array Deque

$$\frac{n}{4} < i < \frac{3n}{4}$$

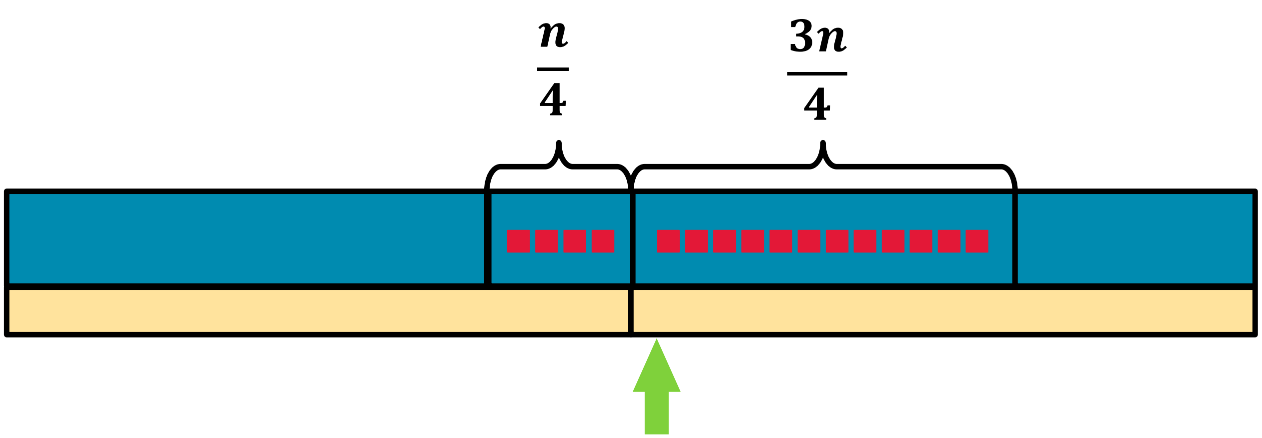
we Could Claim to Need to Shift Up to n Items

(i.e., the maximum number of items, since n is the size)

but a More Accurate Impression is Possible

What is the Worst-Case Scenario?

Adding to Bottom of the Larger of Two Unbalanced Stacks



Theorem for DualArrayDeque

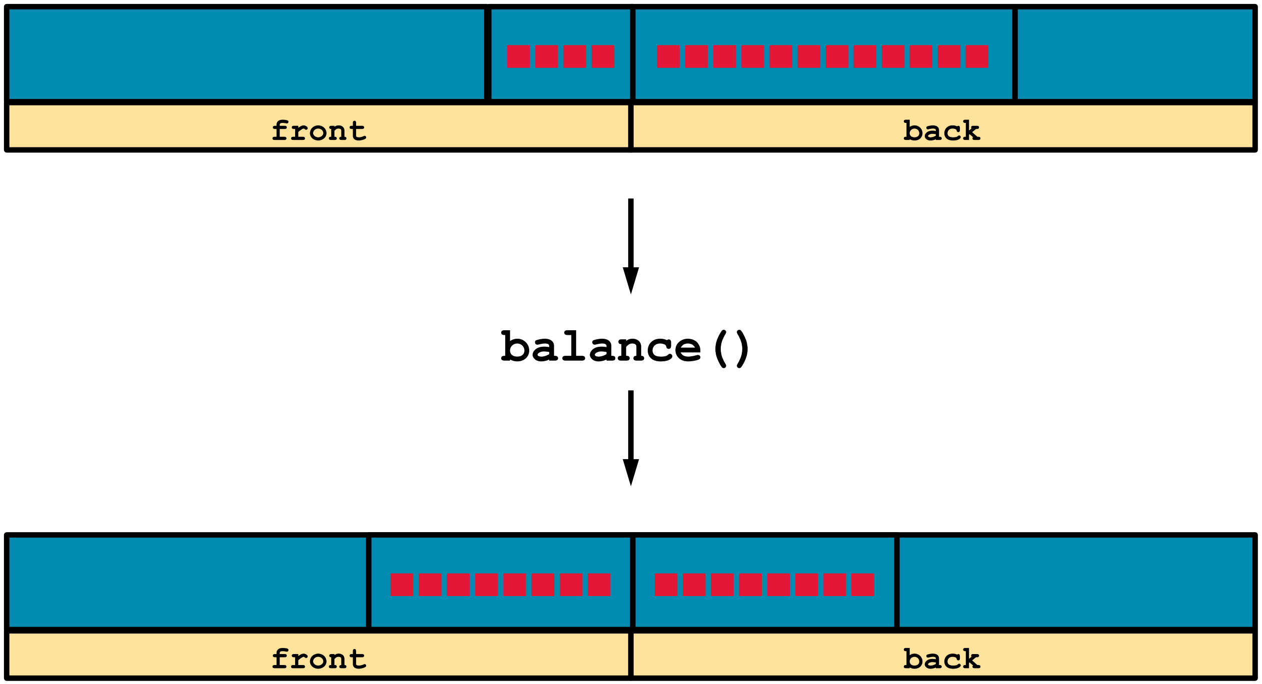
the "DualArrayDeque"† Supports‡:

† *the textbook name for a list backed by two non-circular array stacks*

‡ *if we (momentarily) ignore the cost of resizing and balancing*

- `get(i)` with Time Complexity $O(1)$
- `set(i)` with Time Complexity $O(1)$
- `add(i,o)` with Time Complexity $O(1+\min\{i,n-i\})$
- `remove(i)` with Time Complexity $O(1+\min\{i,n-i\})$

Balancing the Dual Arrays



if n is the **Number** of **Elements** in the **Deque** and **Each** of the **Elements Must** be **Relocated After Balancing**, the **Worst-Case Time Complexity** of `balance` is **O(n)**...

...**But How Often** is this **Cost Incurred**?

Inserting into a Dual Array Deque

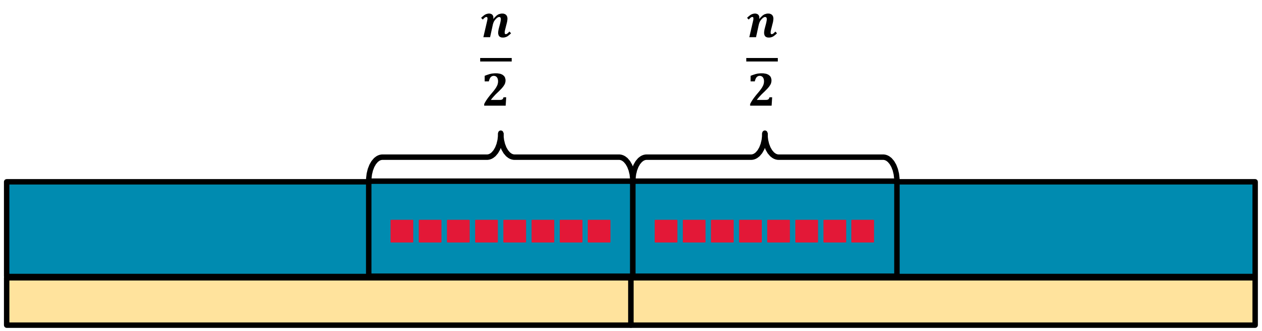
the Potential Φ Associated with a Dual Array Deque
is the Difference (Magnitude Only; an Absolute Value) in
Size between the Front Stack and the Back Stack

if the Stacks are Similar in Size, the Potential is Small
if the Stacks Differ in Size, the Potential is Large

What is the Potential...
...Immediately After a Dual Array Deque is Balanced?
...Immediately Before a Dual Array Deque is Balanced?

Inserting into a Dual Array Deque

the Potential Immediately After the Deque is Balanced

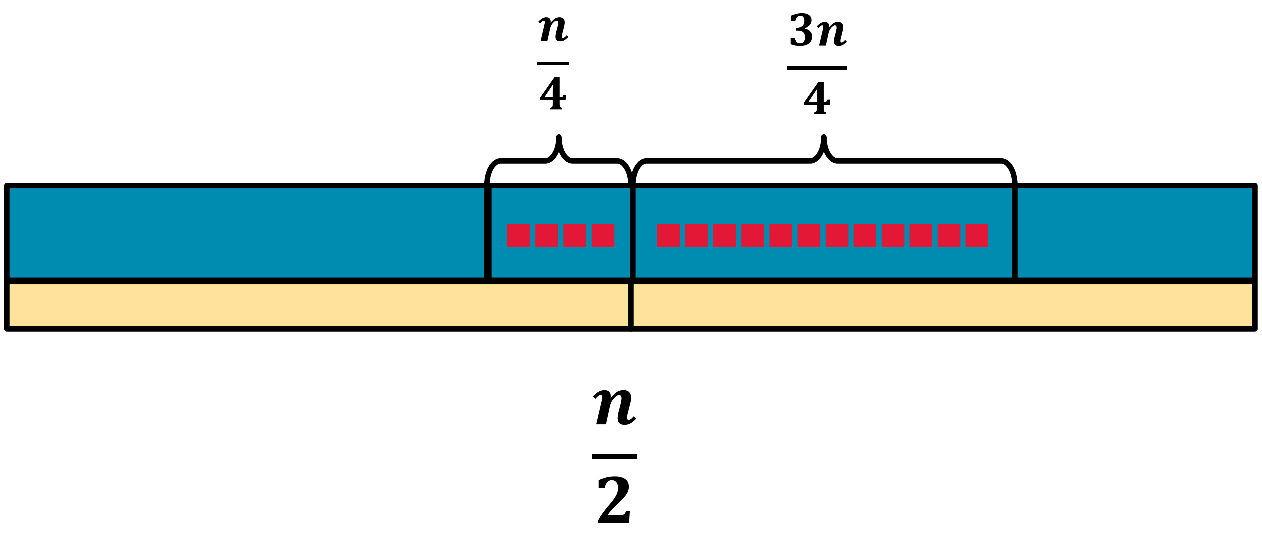


0

(or 1 if the Size is Odd)

Inserting into a Dual Array Deque

the Potential Immediately Before the Deque is Balanced



Inserting into a Dual Array Deque

Minimum Potential is ~0
Maximum Potential is ~n/2

Each Insert/Remove Operation Changes the Potential by 1

Between Calls to `balance`, Φ will Increase from 0 to n/2

Since `balance` Requires Moving n Elements
(i.e., Twice the Number that were Added or Removed)
Each Rebalance at O(n) is Followed by n/2 Operations
that Cannot Possibly Require a call to Balance

this is Amortized Constant Time

Theorem for DualArrayDeque, Revisited

the "DualArrayDeque"† Supports‡:

† the textbook name for a list backed by two non-circular array stacks
‡ if we (momentarily) ignore the cost of resizing and balancing

- `get(i)` with Time Complexity O(1)
- `set(i)` with Time Complexity O(1)
- `add(i,o)` with Time Complexity O(1+min{i,n-i})
- `remove(i)` with Time Complexity O(1+min{i,n-i})

Starting from an Empty Array and Performing a Sequence of m `add` or `remove` Operations entails that the Time Complexity Associated with all Calls to `resize` is O(m)

`addFirst`/`addLast`/`removeFirst`/`removeLast` all run in Constant Amortized Time with ArrayDeque

Space Complexity Considerations

Assume Alternating Inserts and Removes...

What is the Best-Case Scenario in terms of Space Usage?

when the Size is at Capacity - 1

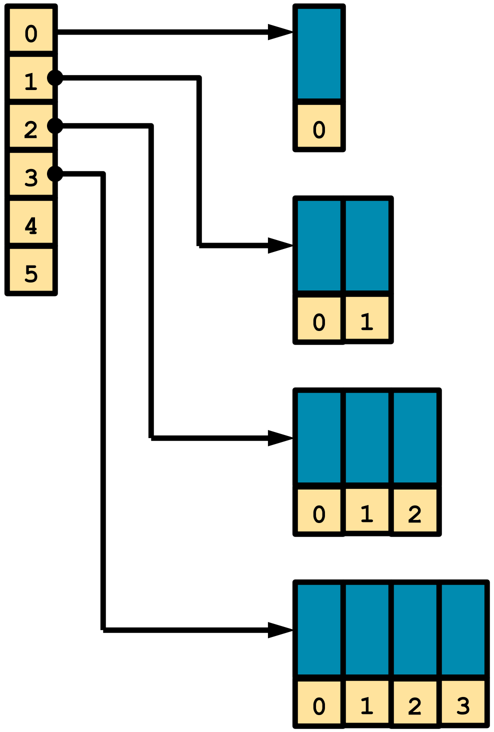
What is the Worst-Case Scenario in terms of Space Usage?

when the Size is at $^1/_3$ Capacity

in other words, for Every Unit of Storage Used
Two Units of Storage might be Wasted

Rootish Array Stacks

Rootish Array Stacks (related: Compact Dynamic Array) Have O($\sqrt{n}$) Wasted Space and this is Demonstrably the Best Possible Space Efficiency (if Inserting and Removing)



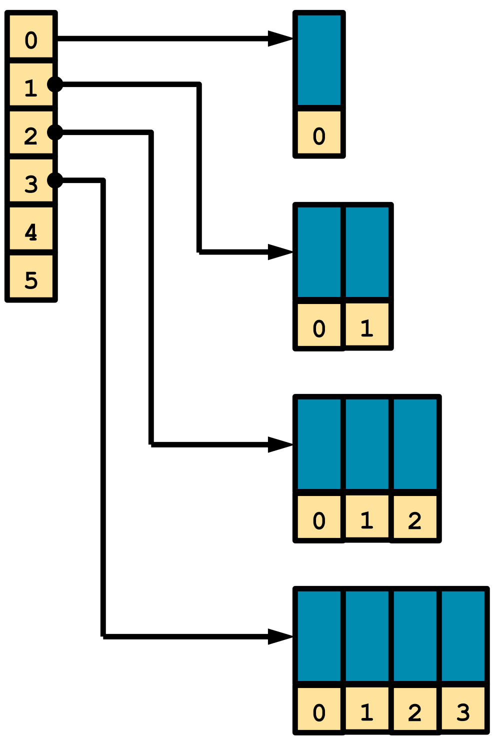
a List of Arrays of Increasing Size

n.b., at first glance it might seem that the overhead associated with a list of array pointers would make this structure less efficient...

Rootish Array Stacks

Each Smaller Array is called a Block

How Much Data can be stored in r Blocks?



Number of Blocks	Maximum Size
1	1
2	1 + 2
3	1 + 2 + 3
…	…
r	$1 + 2 + 3 + \ldots + r$

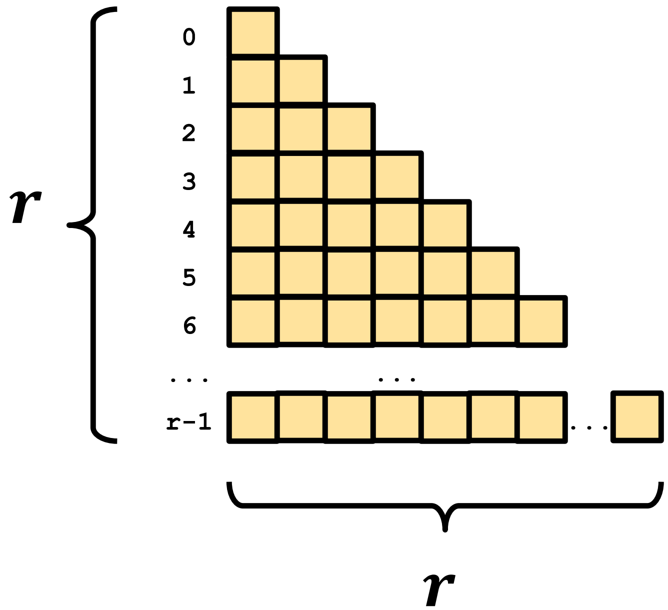
What is Formula for *this*?

Rootish Array Stacks

the Formula for the Inclusive Sum of the Integers 1 to r is

$$\frac{r(r+1)}{2}$$

this can be Derived Intuitively or Proven by Induction

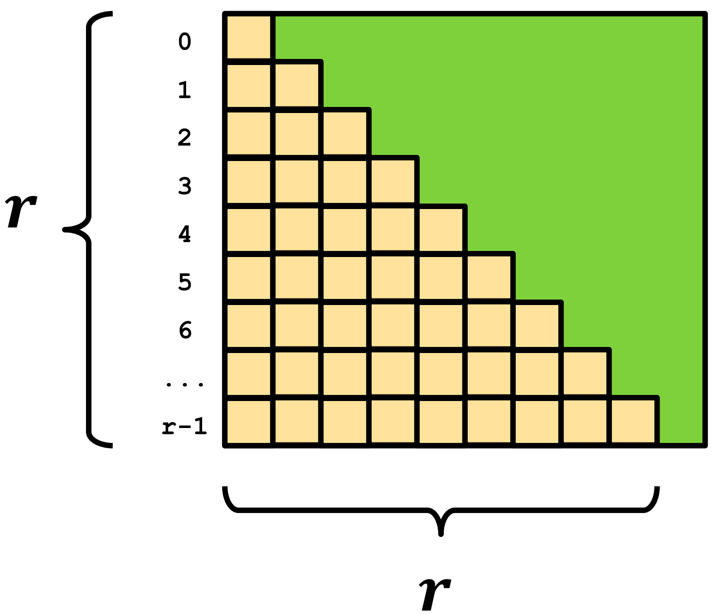


Rootish Array Stacks

the **Formula** for the **Inclusive Sum** of the **Integers 1 to r** is

$$\frac{r(r+1)}{2}$$

this can be **Derived** **Intuitively** or **Proven by Induction**



$$r \cdot (r+1)$$

is Size of the Rectangle
with Dimensions
r and $(r+1)$

← this is Half that Rectangle

Rootish Array Stacks

the Formula for the Inclusive Sum of the Integers 1 to r is

$$\frac{r(r+1)}{2}$$

this Can be Derived Intuitively or Proven by Induction

Base Case

$$1 = \frac{1(1+1)}{2}$$

Inductive Assumption

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

$$1 + 2 + \cdots + k + (k+1)$$

$$\frac{k(k+1)}{2} + (k+1)$$

$$\frac{k^2 + 3k + 2}{2}$$

$$\frac{(k+1)(k+2)}{2}$$

Q.E.D.

Rootish Array Stacks

with no loss of generality: $\frac{r(r+1)}{2} > \frac{r^2}{2}$

If $\frac{r^2}{2} > n$ then there is Sufficient Storage for n items

$$r > \sqrt{2n}$$

How can we Claim $O(\sqrt{n})$ Wasted Space?

How can we Claim $O(\sqrt{n})$ Wasted Space?

1. the last block might not be completely full - in fact, the last block might hold only one item and since the last block is of size r $\rightarrow$ waste ~r locations

Rootish Array Stacks

How can we Claim $O(\sqrt{n})$ Wasted Space?

1. **the last block might not be completely full - in fact, the last block might hold only one item and since the last block is of size r $\rightarrow$ waste ~r locations**

2. **another empty block is frequently appended to the end as space into which the data structure can grow this block is of size r+1 $\rightarrow$ waste ~r locations**

How can we Claim $O(\sqrt{n})$ Wasted Space?

1. the last block might not be completely full - in fact, the last block might hold only one item and since the last block is of size r $\rightarrow$ waste ~r locations

2. another empty block is frequently appended to the end as space into which the data structure can grow this block is of size r+1 $\rightarrow$ waste ~r locations

3. recalling the "wasted" pointers for each block and since there are r blocks $\rightarrow$ waste ~r locations

Rootish Array Stacks

How can we Claim $O(\sqrt{n})$ Wasted Space?

1. the last block might not be completely full - in fact, the last block might hold only one item and since the last block is of size r $\rightarrow$ waste ~r locations

2. another empty block is frequently appended to the end as space into which the data structure can grow this block is of size r+1 $\rightarrow$ waste ~r locations

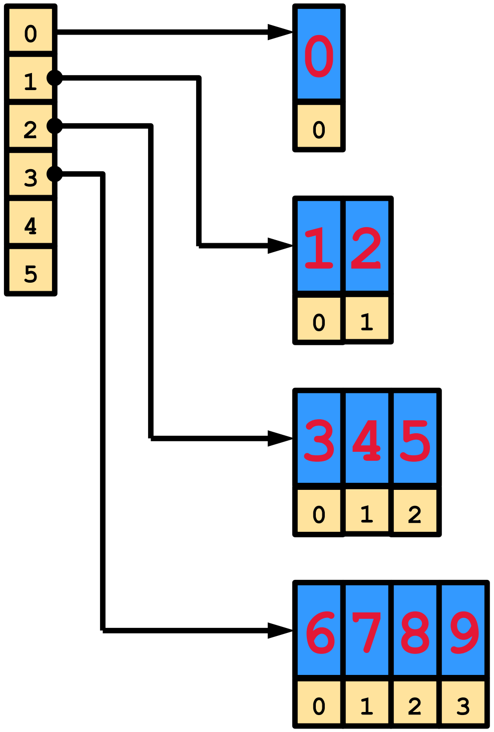
3. recalling the "wasted" pointers for each block and since there are r blocks $\rightarrow$ waste ~r locations

Wasted Space is $O(3r) \approx O(3\sqrt{2n}) \approx O(\sqrt{n})$

Indexing Rootish Array Stacks

the Ability to Locate Elements is Necessary for Implementing `get`/`set`/`add`/`remove`

Index Alone is No Longer Sufficient; Need Block Number



Number of Elements Stored in Lists 0 to b (i.e., $(b+1)$ lists) is:

$$1 + 2 + \cdots + (b+1) = \frac{(b+1)(b+2)}{2}$$

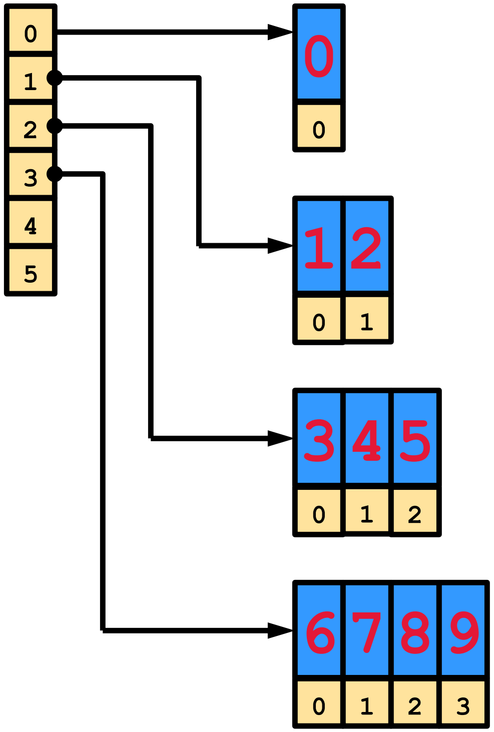
$\therefore$ Indices $0..\left(\frac{(b+1)(b+2)}{2} - 1\right)$ are Stored in Lists 0 to b

Indexing Rootish Array Stacks

to Find the Block that Contains index i, Solve:

$$\frac{(b+1)(b+2)}{2} - 1 = i$$

(more accurately, find the smallest b such that $\frac{(b+1)(b+2)}{2} - 1 \geq i$)



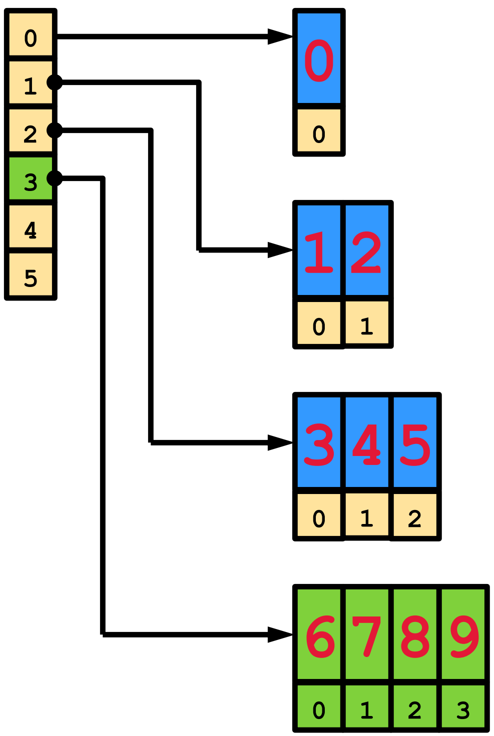
e.g., Find the Block that Holds Index 8

Indexing Rootish Array Stacks

to Find the Block that Contains index i, Solve:

$$\frac{(b+1)(b+2)}{2} - 1 = i$$

(more accurately, find the smallest b such that $\frac{(b+1)(b+2)}{2} - 1 \geq i$)



e.g., Find the Block that Holds Index 8

$$\frac{(b+1)(b+2)}{2} - 1 = i$$

$$b^2 + 3b - 2i = 0$$

$$\left\lceil \frac{-3 \pm \sqrt{3^2 - 4 \cdot 1 \cdot (-2i)}}{2} \right\rceil$$

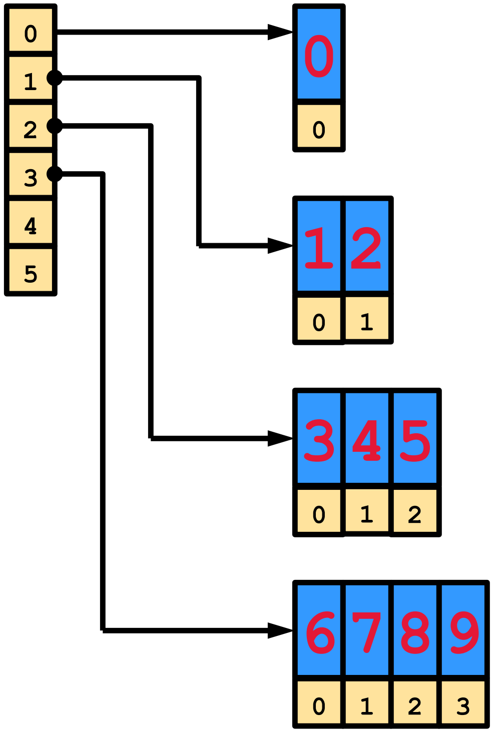
$$\lceil 2.772 \rceil = 3$$

Indexing Rootish Array Stacks

to Find the Block that Contains index i, Solve:

$$\frac{(b+1)(b+2)}{2} - 1 = i$$

(more accurately, find the smallest b such that $\frac{(b+1)(b+2)}{2} - 1 \geq i$)



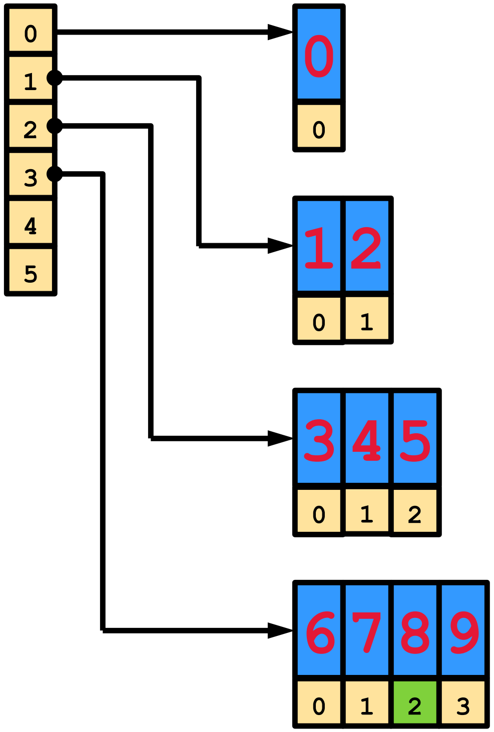
If the Block is 3, then Find the Index j Into this Block (i.e., corresponding to i = 8)

Indexing Rootish Array Stacks

to Find the Block that Contains index i, Solve:

$$\frac{(b+1)(b+2)}{2} - 1 = i$$

(more accurately, find the smallest b such that $\frac{(b+1)(b+2)}{2} - 1 \geq i$)



If the Block is 3, then Find the Index *j* Into this Block (i.e., corresponding to i = 8)

$$j = i - \frac{b(b+1)}{2}$$

$$j = 8 - 6 = 2$$