CECS 274: Data Structures
Array-Based Lists

California State University, Long Beach

Sample Applications

- How to write a program to check if the parenthesis in the following expression are valid?

$$((a+b)*(c*d)+(a+(b*(c/d))))$$

STACK?

- How to write a program to serve customers at a call center following the arrival order?

QUEUE

Relevant Interfaces

To help us answer these questions, we will define data structures that implement the `Queue`, `Stack`, `Deque`, and `List` interfaces.

We will implement these interfaces using an array called the *backing array*. In Python, we will use in particular an array structure from the `numpy` library.

First, let's take a look at what operations are defined in these interfaces.

The `Queue` Interface

The `Queue` interface represents a collection of elements to which we can add new items, and remove in a particular order. The operations supported by the `Queue` interface are:

- `add(x)`: add the value `x` to the sequence.
- `remove()`: remove the next value (according to some order) from the sequence and return it
- `size()`: returns the number of values in the sequence.

Preview on Queue Implementation

The two most common implementations of the `Queue` interface add and remove in the following manner:

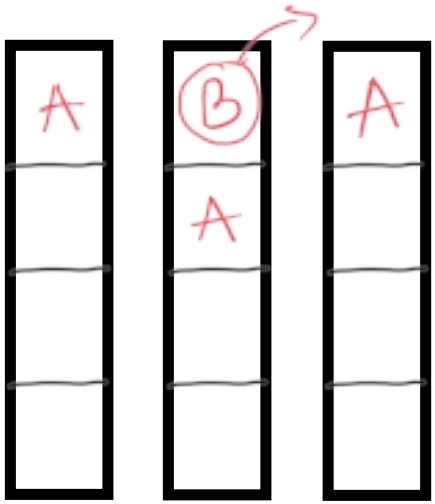
- FIFO Queue: In a First-In, First-Out queue, the elements are removed in the order that they were added to the queue. Analogy: People in line at a bank teller are helped in FIFO queue order, first who arrived is first to get helped, followed by the next arrival. This is the standard queue order that is usually referred to when speaking of a queue data structure.
- Priority Queue: the elements are removed in order of priority where the priority is inherent to the nature of the element. For example, if the integers 5, 1, 6, 3, 2 are stored into a priority queue, they might get removed as 1, 2, 3, 5, 6 if lower numbers are interpreted to have higher priority.

The `Stack` Interface

The `Stack` interface represents a specialization of a queue data structure in that elements are added and removed in a specific order. In particular, the `Stack` interface supports the operations:

- `push(x)`: adds element `x` to the beginning of the sequence.
- `pop()`: removes the element at the beginning of the sequence.

push(A)
push(B)
pop()
push(C)
push(D)



Stack or Queue?

Some textbooks do not define a separate interface for a stack data structure, and instead implement a stack as a Last-In, First-Out (LIFO) `Queue` so that `add(x)` will add to the top of the `List`, and `remove()` will remove also from the top of the `List`.

To avoid confusions between a FIFO Queue and LIFO Queue, we will define a separate `Stack` interface with the `push(x)` and `pop()` methods listed in the previous slide.

The `Deque` Interface

The `Deque` interface defines a double-ended queue where elements can be added and removed at the beginning and end of the sequence. It supports the following operations:

- `add_first(x)`: adds element `x` to the beginning of the sequence.
- `remove_first()`: removes the element at the beginning of the sequence and returns it.
- `add_last(x)`: adds element `x` to the end of the sequence.
- `remove_last()`: removes the element at the end of the sequence and returns it.


```
arr.add_first("A")
arr.add_last("C")
arr.add_first("D")
x = arr.remove_last()   "C"
```

The `List` Interface

The `List` interface defines a general sequence with the following operations:

- `size()`: returns the length of the sequence
- `get(i)`: return the element at the `i`-th position of the sequence.
- `set(i, x)`: set the value of the element at the `i`-th position to be `x`. Return the value of the element that was replaced.
- `add(i, x)`: insert element `x` at position `i`
- `remove(i)` remove element at position `i`; this results in displacing all elements back one index, beginning at index $i + 1$.

Efficiency Considerations for Array-Based Implementations

As mentioned earlier, we will implement the `Stack`, `Queue`, `Deque` and `List` interfaces using an array, called the *backing array*. We will do so by creating `ArrayStack`, `ArrayQueue`, `ArrayDeque` and `ArrayList` data structures that implement one or more of these interfaces. The data structures will offer the following operation efficiencies:

	`add(x)` / `remove()`	`add(i, x)`/`remove(i)`
`ArrayStack`	$O(1)$	$O(n-i)$
`ArrayQueue`	$O(1)$	$O(\min(i, n-i))$
`ArrayDeque`	$O(1)$	$O(\min(i, n-i))$
`ArrayList`	N/A	$O(\min(i, n-i))$

An array is a sequence of elements that are referenced by the same variable name.
An element in the array can be accessed by specifying the index of the memory bin it is stored in
The size of the array is static
i.e. once the number of bins has been defined, the array can NOT shink or grow

Review: Array

Suppose a is an array of 5 elements:

$a =$

23	43	13	65	0

Variable	Address (Hex)	Value
	0x00000000 0x0000FFFF ⋮	
a	0x01FFFFBB 0x02FFFFBA 0x03FFFFB9 0x04FFFFB8 0x05FFFFB7	23 43 13 65 0
n	0x06FFFFB6 ⋮	4

a[1] = 43

Review: Array

- For the array a:
  - $a[0]$ contains the value 23 at $0x01FFFFBB$
  - $a[1]$ contains the value 43 at $0x02FFFFBA$
  - $a[2]$ contains the value 13 at $0x03FFFFB9$
  - $a[3]$ contains the value 65 at $0x04FFFFB8$
  - $a[4]$ contains the value 0 at $0x05FFFFB7$
  - $a[5]$ will *crash* (stop the program) with an overflow exception
  - $a[-1]$ will *crash* with an underflow exception
- Accessing $a[i]$ takes $O(1)$ time.
- Arrays cannot expand or shrink

Review: Methods

```
class TVController(Controller):
    """
    This class instantiates TVController object with the basic
    functionalities to turn ON/OFF controlled device, and
    increase/decrease volume.  TVController implements the
    Controller interface
    """
    default_volume = 15

    def __init__(self):
        """
        default constructor initializes TVControl object with attributes
        to determine if device is ON or OFF, and to determine volume
        """
        self.is_on = False
        self.volume = TVController.default_volume

    def power(self):
        self.is_on = not self.is_on
        return self.is_on

    def volume_up(self):
        if self.is_on:
            self.volume += 1
        return self.is_on

    def volume_down(self):
        if self.is_on:
            self.volume -= 1
```

* Precondition
* Effect

Review: Methods

- Methods are used to access and modify the state of the object

```
method(parameters)
    Precondition:
    Effect:
```

- *Precondition*: The condition or *predicate* that must always be true prior to the execution. Otherwise, the effect becomes undefined
- *Effect*: Defines the steps that modify the state or access the state

Review: Methods

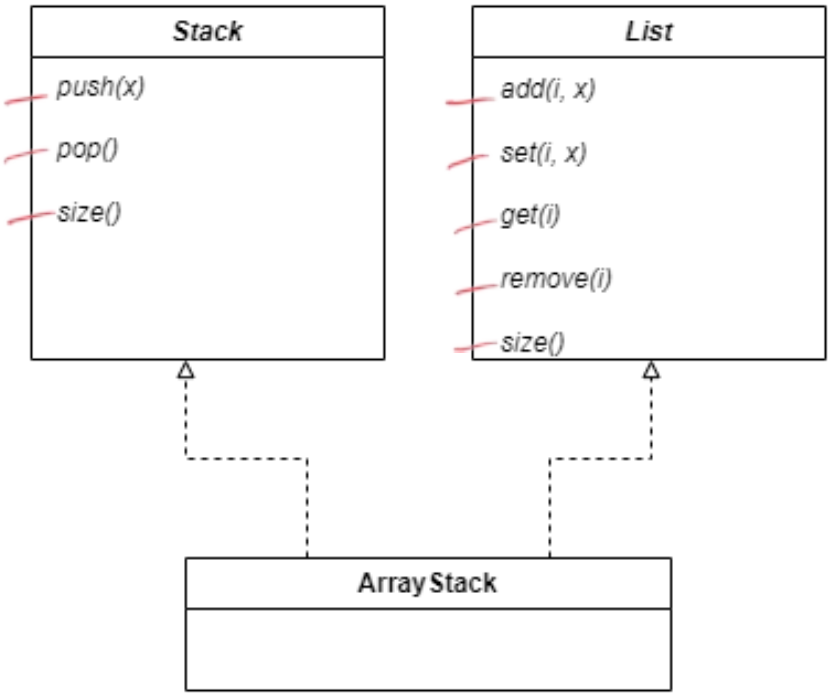
```
    def volume_up(self):
        if self.is_on:
            self.volume += 1
        return self.is_on

    def volume_down(self):
        if self.is_on:
            self.volume -= 1
        return self.is_on
```

★ Precondition
★ Effect
★ Precondition
★ Effect

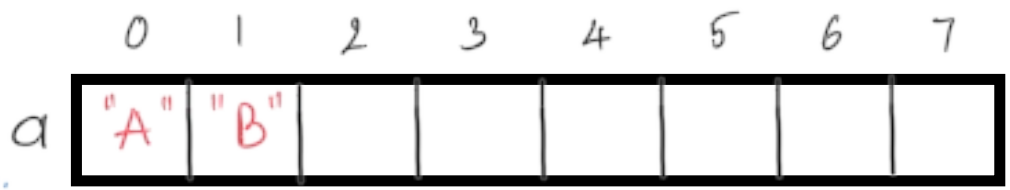
`ArrayStack`: Intro

The `ArrayStack` data structure will implement the `Stack` and `List` interfaces. This means it will offer all the functionalities of a traditional `Stack` but also have the general operations of a `List`.



ArrayStack: Intro

Not allowed because there are too many empty bins.



$len(a) = 8$
$n = 2$

$len(a) = 8 \not\leq 3 \cdot n = 3 \cdot 2 = 6$

We will adopt the *invariant* (i.e. a condition on the state of the structure that must always be true) that all times the length of the backing array a, is no less than n, where n is the number of elements in the stack, and no greater than 3 times the number of elements.

$n \leqslant len(a) \leqslant 3n$

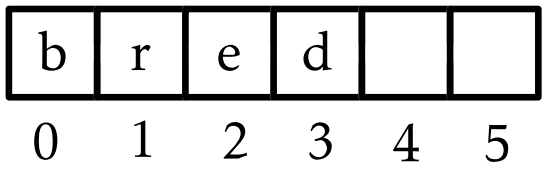
$len(a) = 4$
$n = 2$
$2 \leqslant 4 \leqslant 3 \times 2$
✓

ArrayStack: The Basics

In summary,

- State variables:
  - n: Number of elements in a. Initially set to 0.
  - a: Backing array where $a[i]$ stores the element i.
- Invariant: At all times the length of a is no less than n and no greater than $3n$. Formally,

$$n \leq length(a) \leq 3n$$



$n = 4$

$length(a) = 6$

$4 \leq 6 \leq 3 \times 4$

(12)

ArrayStack: Constructor

- $initialize()$: default constructor, sets all the state variables to the initial state

1. Initialize n to 0: $n \leftarrow 0$ # assign to n the value 0

2. Initialize an array a of size 1: $a \leftarrow$ new_array(1)

NOTE: $n \leq len(a) \leq 3n$

This is the only time we make an exception to the invariant, because if we initialize an array of size 0, we will not be able to add any elements.

ArrayStack: List method get(i)

- get(i): return the value of the element i in the List
  - Precondition: $0 \leq i < n$, where n = number of elements
  - Effect: return a[i]
  - Does the invariant hold? Yes, invariant is unaffected by simple access, i.e. we did not remove or add to affect n or len(a).

ArrayStack: List method set(i, x)

- `set(i, x)`: set the value of the element at the `i`-th position to be `x`. Return the value of the element that was replaced.
  - Precondition: $0 \leq i < n$, where n = number of elements
  - Effect:

    1. Store a[i] into a temp variable: $y \leftarrow a[i]$
    2. Overwrite a[i] with x: $a[i] \leftarrow x$
    3. Return y: return y

ArrayStack: Python

```
class ArrayStack(Stack):
    def __init__(self):
        self.a = self.new_array(1)
        self.n = 0

    def new_array(self, n: int):
        return numpy.zeros(n, numpy.object)

    def get(self, i : int) -> np.object:
        if i < 0 or i >= self.n:  raise IndexError()
        return self.a[i]

    def set(self, i : int, x : numpy.object) -> numpy.object:
        if i < 0 or i >= self.n:  raise IndexError()
        y = self.a[i]
        self.a[i] = x
        return y
```

n = 5

[0, 0, 0, 0, 0]

① temp variable
② overwrite
③ return the value that was replaced

ArrayStack: Java

```
public class ArrayStack implements Stack  {
    protected int n;
    protected Object[] a;

    public ArrayStack()  {
        n = 0;
        a = new_array(1);
    }

    public Object[] new_array(int n) {
        return new Object[n];
    }

    public Object get(int i) throws ArrayIndexOutOfBoundsException {
        if (i < 0 || i >= n)
                throw new ArrayIndexOutOfBoundsException();
        return  a[i];
    }

    public Object set(int i, Object x) throws ArrayIndexOutOfBoundsException  {
        if (i < 0 || i >= n)
            throw new ArrayIndexOutOfBoundsException();
        Object y = a[i];
        a[i]= x;
        return y;
    }
```

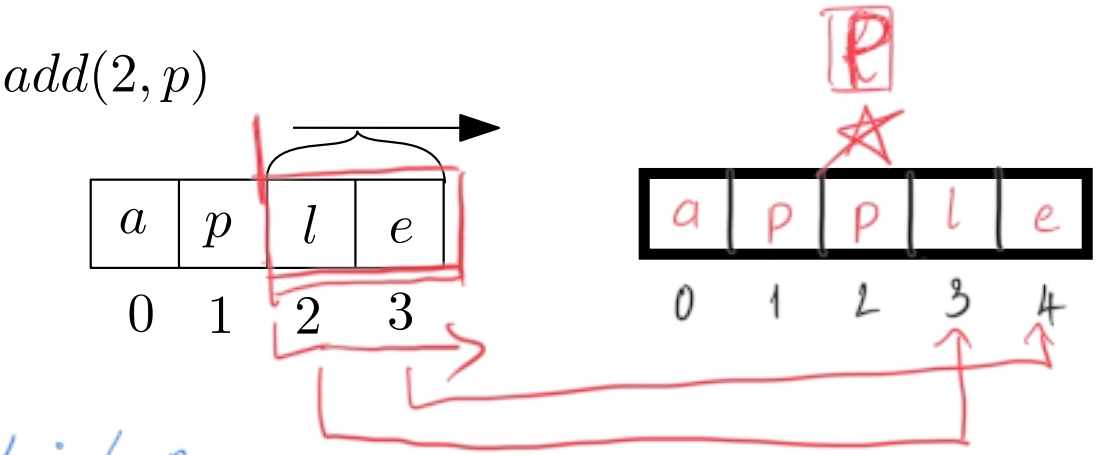
constructor

getter

setter

ArrayStack: List method add(i,x)

$add(i, x)$: insert element x at position i.



- Precondition: $0 \leq i \leq n$

- Effect:
  1. Shift forward to make enough room for the new variant
     i.e. Overwrite a[n] with a[n-1]
     a[n-1] with a[n-2]
     ⋮
     a[i+1] with a[i]
  2. Overwrite a[i] with x
  3. Increment n by 1

- Does the invariant hold?
  Before we make the insertion, we must verify whether the array has enough capacity. If not, we have to increment the capacity.
  i.e. if len(a) = n before insertion, we cannot add any more elements so we must create a larger array.

ArrayStack: List method add(i,x)

$\text{add}(i, x)$

Precondition: if $i < 0$ or $i > n$: Error

Invariant: if $\text{len}(a) == n$: resize() $\leftarrow O(?)$

Step 1: for $k = n, n-1, \ldots, i+1$: $a[k] \leftarrow a[k-1]$ — $n - i$ iterations, 2 visits / iter $\Big\} (n-i) \cdot 2$

Step 2: $a[i] \leftarrow x$ $\leftarrow$ 1 visit

Step 3: $n \leftarrow n + 1$

Runtime: $O(?) + (n-i) \cdot 2 + 1$

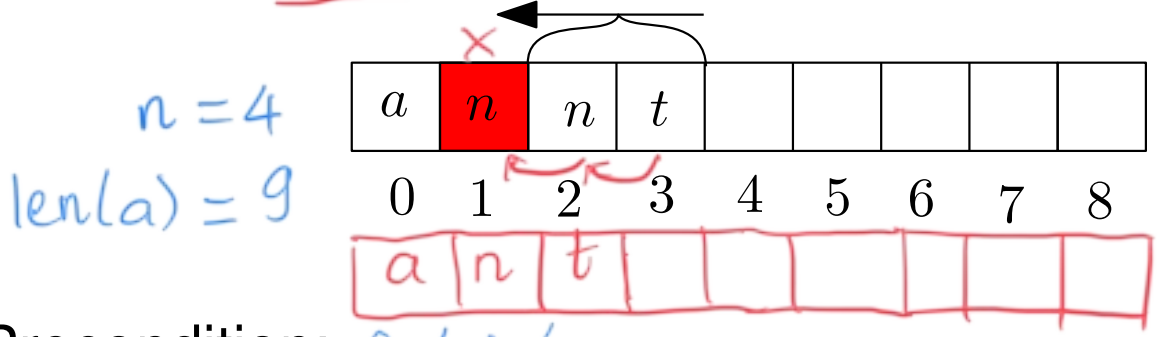
$O(n)$ worst-case

ArrayStack: List method remove(i)

$remove(i)$: remove element at position i and recover space.

returns the removed value

$remove(1)$



$n \leq len(a) \leq 3n$

* $n = 5$
  $len(a) = 9 \leq 3n = 3.3$?
* $n = 2$
  $len(a) = 9 \not\leq 3n = 3.2$?
  -> shrink

- Precondition: $0 \leq i < n$

- Effect:

1. Store a[i] into a temp variable
2. "Shift" values left: overwrite
   a[i] ← a[i+1]
   a[i+1] ← a[i+2]
   ⋮
   a[n-2] ← a[n-1]
3. Decrement n by 1
4. Return the value of temp variable

- Does the invariant hold?

Depends. If we remove enough elements eventually $len(a) \not\leq 3n$. In such case, we will have too many empty bins (waste of memory space). So, we must shrink the size of the array.

ArrayStack: List method remove(i)

$i = 5, n = 10$ [5, 6, 7, 8]

remove(i)

Precondition: if $i < 0$ or $i \geq n$: Error

Step 1: $y \leftarrow a[i]$ ← 1 visit

Step 2: for $k = i, i+1, \ldots, n-2$: $(n-2) - (i-1) = n-i-1$ iterations

$a[k] \leftarrow a[k+1]$ 2 visits/iter ⇒ Total $= (n-i-1) \cdot 2 = 2(n-i) - 2$

Step 3: $n \leftarrow n - 1$

Invariant: if $len(a) > 3n$: resize() ← $O(?)$

Step 4: return y

- How many operations?

$1 + 2(n-i) - 2 + O(?)$ → worst-case $O(n)$

ArrayStack: Helper method resize()

resize()

1. Create a new array b of size $\max(1, 2n)$ ← in case $n = 0$
2. Copy elements from a to b
3. Reassign a so that it references array b

Variable	Address (Hex)	Value
a	0x01FFFFBB	'a'
	0x02FFFFBA	'n'
b	0x1FFFFB800	'a'
	0x2FFFFB700	'n'
	0x2FFFFB700	
	0x2FFFFB700	

$n = 2$

$len(a) = 2$

$len(b) = 4$

`ArrayStack`: Helper method `resize()`

resize()

Step 1: $b \leftarrow$ new_array$(\max(1, 2n)) \leftarrow O(1)$

Step 2: for $k = 0, 1, \ldots, n-1$: — n iterations, 1 visit / iter $\} \; n \cdot 1 = n$

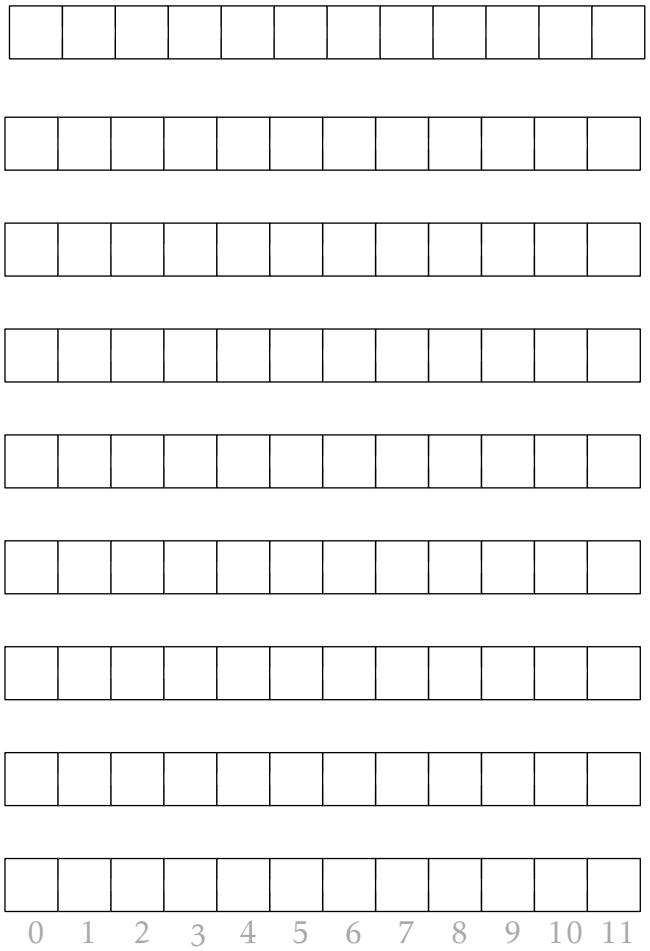
$b[k] \leftarrow a[k]$

Step 3: $a \leftarrow b$

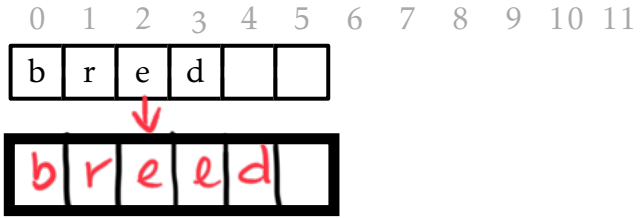
- How many operations? $O(1) + n = O(n)$

ArrayStack: Example

time	0	1	2	3	4	5	6	7	8	9	10	11	n	$len(a)$	*Operation*
1	b	r	e	d									4	6	



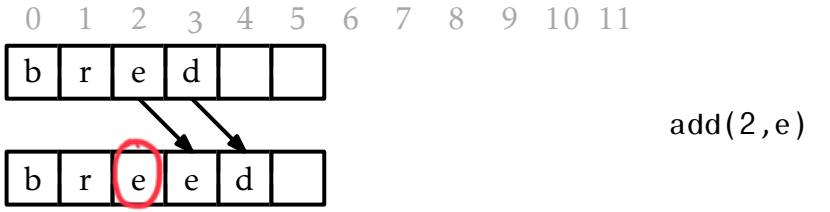
ArrayStack: Example



add(2,e)

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example

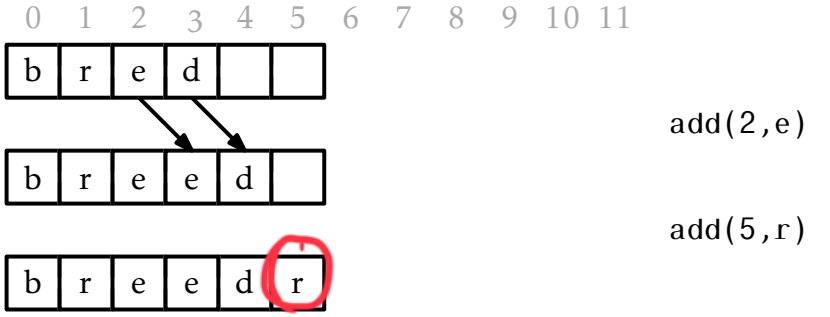


add(2,e)

add(5,r)

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



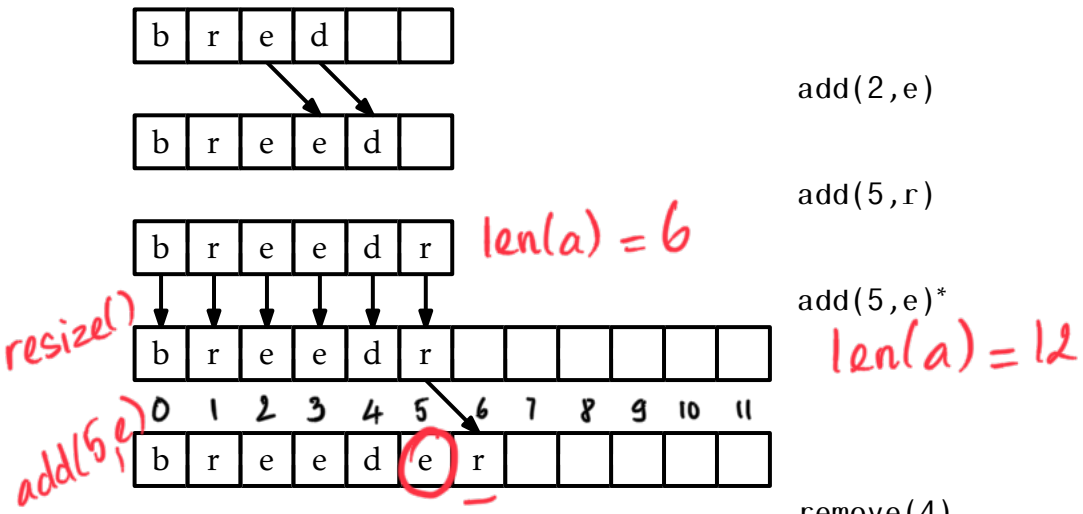
add(2,e)

add(5,r)

add(5,e)*

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



add(2,e)

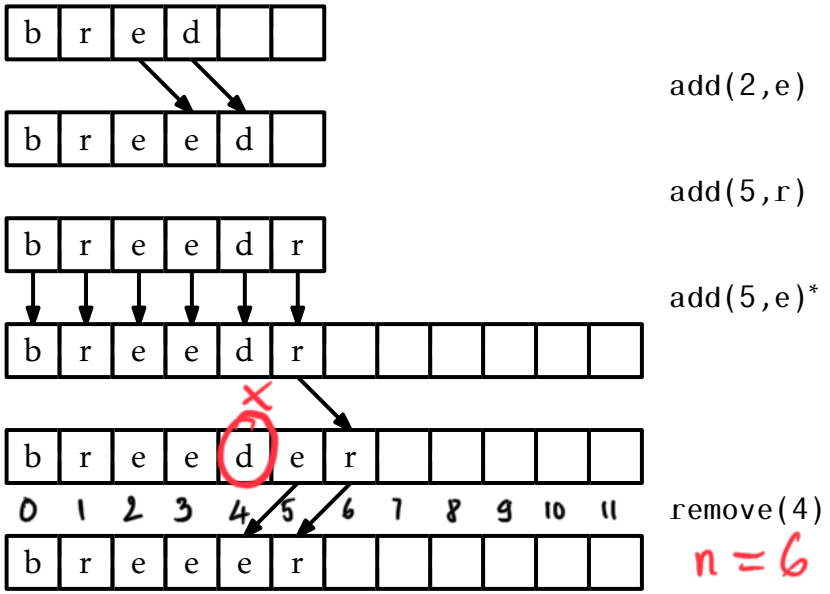
add(5,r)

add(5,e)*

remove(4)

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



add(2,e)

add(5,r)

add(5,e)*

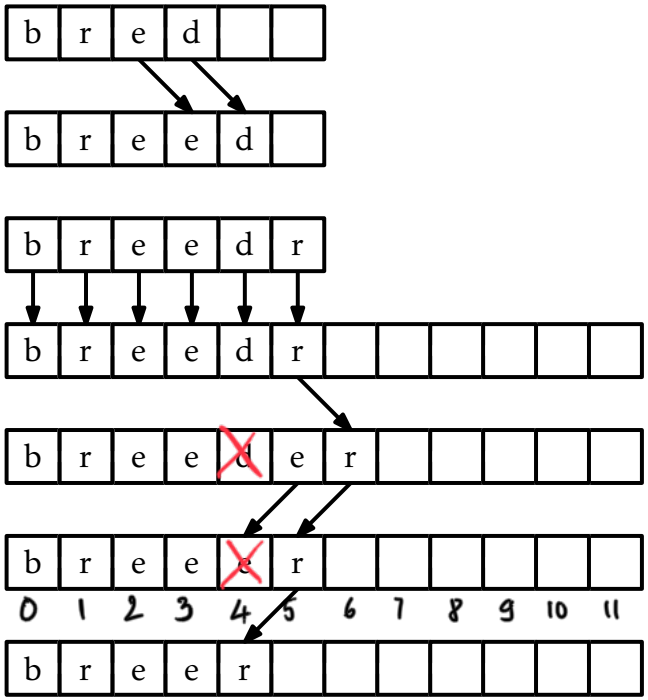
remove(4)

n = 6

remove(4)

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



add(2,e)

add(5,r)

add(5,e)*

remove(4)

remove(4)

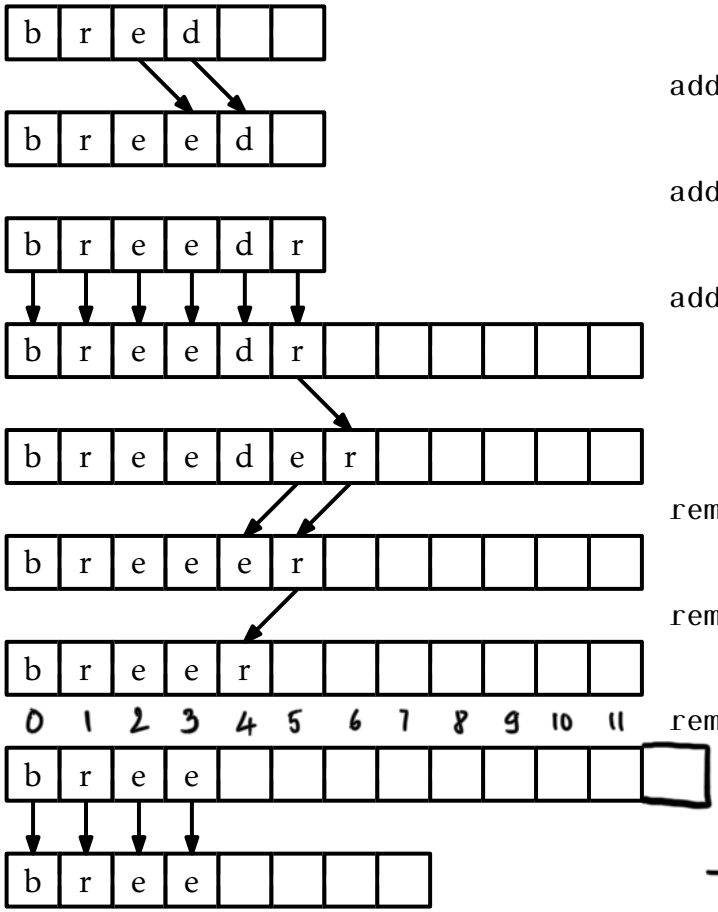
remove(4)*

$n = 4$

$len(a) = 12 \leq 3n = 3 \cdot 4$

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



add(2,e)

add(5,r)

add(5,e)*

remove(4)

remove(4)

remove(4)*

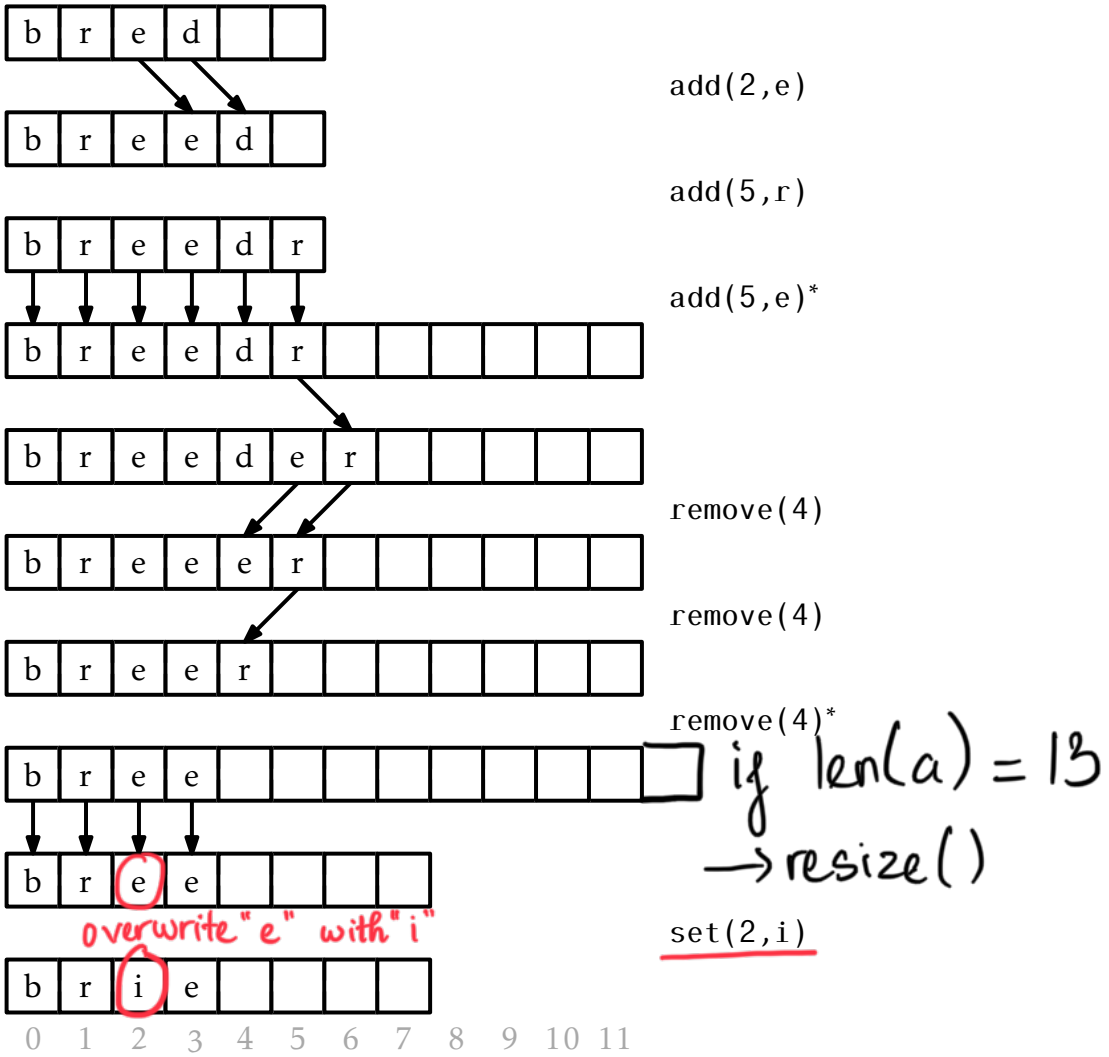
set(2,i)

if len(a) = 13 $\not<$ 3.n = 3.4 = 12

→ resize() len(b) = 2.n = 2.4 = 8

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



ArrayStack: Stack method push(x) / pop()

Now that the `add(i,x)` and `remove(i)` methods have been fully implemented, we can easily implement the `Stack` methods `push(x)` and `pop()` as `add(n, x)` and `remove(n-1)`, respectively.

```
push(x):
    add(n, x)


pop():
    return remove(n-1)
```

Amortized Run-time

The amortized run-time of an operation, refers to average run-time (as oppose to worse case run-time) over a sequence of several calls. Amortized analysis considers both costly and less costly operations over the entire sequence of operations.

Amortized analysis is appropriate for analyzing the efficiency of operations over which the worst-case scenario (i.e., the costly scenario) does not happen often.

Since the `resize()` method is not called at every add or remove operation, it is appropriate for us to consider the amortized run-time of `add(x)`, `remove()`, `add(i,x)` and `remove(i)`.

Amortized Run-time: `push(x)`

To calculate the amortized run-time of `push(x)`, we use the formula for amortized run-time for n calls to the operation:

$$\text{amortized run-time} = \frac{\text{Total all Cost of normal calls} + \text{Total all Cost of expensive calls}}{n}$$

Notice that,

- Cost of 1 normal call to `push(x)` = $O(1)$ because element x is simply stored in the last available bin of the array.
- Cost of 1 expensive call to `push(x)` = $O(j)$ where j is the current size of the array after the method is done executing. E.g.: $j = 4$

Suppose we grow an `ArrayStack` to size n by calling $push(x)$ n consecutive times. How many of those n-times will result in an expensive call? To answer this, let's consider when is `resize()` called...

n calls ⇔ How many were normal calls? How many were expensive calls?

Amortized Run-time: `push(x)`

Notice: calls to resize() happen when $j = 2^i + 1$ for $i = 0, 1, 2, \ldots$

Let j represent the current size of the `ArrayStack`.

Call to `push(x)`	Backing array a	$j =$	Call to `resize()`?
`add(0, x)`	[x] len(a) = 1	1	No
`add(1, x)`	[x, x] len(a) = 2	2 (= 2^0+1)	Yes ✓
`add(2, x)`	[x, x, x,] len(a) = 4	3 (= 2^1+1)	Yes ✓
`add(3, x)`	[x, x, x, x] len(a) = 4	4	No
`add(4, x)`	[x, x, x, x, x, , ,] len(a) = 8	5 (= 2^2+1)	Yes ✓
`add(5, x)`	[x, x, x, x, x, x, ,] len(a) = 8	6	No
`add(6, x)`	[x, x, x, x, x, x, x,] len(a) = 8	7	No
`add(7, x)`	[x, x, x, x, x, x, x, x] len(a) = 8	8	No
`add(8, x)`	[x, x, x, x, x, x, x, x, x, , , , ,] len(a) = 16	9 (= 2^3+1)	Yes ✓

9 calls

Amortized Run-time: `push(x)`

Notice that a call to `resize()` happens at call $j = 2^i + 1$ for $i = 0, 1, 2, 3, \ldots$. Without loss of generality, suppose that the number of calls we make to `push(x)` is $n = 2^r + 1$ for some integer r. Then,

$n = 9$ elements $= 2^3 + 1 \Rightarrow r = 3$

calls to resize() $= 3 + 1 = 4$

- # calls to `resize()` = $r + 1$ because the array had to double $r + 1$ times in order to store $2^r + 1$ elements.

Write r in terms of n:

$$n = 2^r + 1$$
$$n - 1 = 2^r$$
$$\log_2(n-1) = r$$

$\Rightarrow$ # of calls to resize() in n calls to push(x) is $\log_2(n-1) + 1$

- At each call $j = 2^i + 1$ of this type, the size of the `ArrayStack` is $2^i + 1$. Since `resize()` has linear run-time, each call has efficiency $O(2^i)$.

Amortized Run-time: `push(x)`

The discussion from the previous slides lead to the following:

$$\text{Total Cost of expensive calls} \leq \sum_{k=1}^{1+\log_2(n-1)} 2^k = \sum_{k=0}^{\log_2(n-1)} 2^{k+1}$$

Justification:

Total cost calls to resize() $= O(2) + O(3) + O(5) + \ldots + O(2^r+1)$ (i=0→j=2, i=1→j=3, i=2→j=5, i=r)

$$\leq c\left[2 + 3 + 5 + \ldots + 2^r + 1\right]$$

$$= c\left[\sum_{k=0}^{r} 2^k + 1\right]$$

$$= c\left[\sum_{k=0}^{\log_2(n-1)} 2^k + 1\right]$$

$$\leq \sum_{k=0}^{\log_2(n-1)} 2^k \cdot 2 = \sum_{k=0}^{\log_2(n-1)} 2^{k+1}$$

$g(n) = O(f(n)) \Rightarrow g(n) \leq c \cdot f(n)$

GEOMETRIC SUM FORMULA:

$$\sum_{k=0}^{N} b^k \cdot c = c\left[\frac{b^{N+1} - 1}{b - 1}\right]$$

Amortized Run-time: `push(x)`

Applying Geometric Sum Formula:

Total cost of expensive calls to "push"

$$= \text{Total cost of calls to resize()} \leq \sum_{k=0}^{\log_2(n-1)} 2^k \cdot 2$$

$$N+1$$

$$= 2\left[\frac{2^{\log_2(n-1)+1} - 1}{2-1}\right]$$

$$= 2\left[2^{\log_2(n-1)} \cdot 2 - 1\right]$$

$$= 2\left[(n-1)2 - 1\right] = 2\left[2n - 2 - 1\right]$$

$$= 2\left[2n - 3\right] = 4n - 6$$

Amortized Run-time: `push(x)`

Total cost of normal calls to "push":

$= O(1)$ for one call $\times$ $(n - (r + 1))$ calls ← # of expensive calls

$= O(1)$ for one call $\times$ $(n - \log_2(n-1) - 1)$ calls

less than n calls

$\leq O(n) \approx c.n$

$$\text{Amortized runtime}_{\text{push}(x)} = \frac{\text{total cost of expensive calls} + \text{total cost of normal calls}}{n}$$

$$\leq \frac{(4n-6) + c.n}{n} = \frac{(4+c)n}{n} - \frac{6}{n}$$

$$= O(1)$$

Amortized Run-time: `resize()`

$$\text{Amortized runtime} = \frac{\text{runtime of 1 call to the operation}}{\begin{array}{l}\#\text{ of calls to a parent operation}\\ \text{that had to occur in order for the}\\ \text{current operation to take place.}\end{array}}$$

* A call to resize() happens $j = 2^i + 1$ where j is the number of elements.
* we had to call add(i, x) $2^i + 1$ times to make a call to resize()

$$\Rightarrow \underbrace{\text{Amortized runtime}}_{\text{resize()}} = \frac{2^i + 1}{2^i + 1} = 1 \Rightarrow O(1)$$

Theorem

Theorem

An `ArrayStack` implements the `List` interface. Ignoring the cost of calls to resize(),

- `get(i)` *and* `set(i,x)` *have* $O(1)$ *run-time efficiency.*
- `remove(i)` *and* `add(i,x)` *have* $O(n-i)$ *run-time efficiency.*

`ArrayStack` efficiently implements the `Stack` interface by using already implemented `List` methods.

- push(x): add(n, x)
- pop(): remove($n-1$),

these operations will run in $O(1)$ amortized time.

Applications

- How can we write a program to check if the parenthesis in the following expression are valid?

$$((a + b) * (c * d) + (a + (b * (c/d))))$$

Idea:

1. Create a stack
2. Iterate through each character of the string
   - ↳ If the character is open parenthesis "(", push it to the stack
   - ↳ If the character is closed parenthesis ")", pop from the stack
3. If the stack is empty, the parenthesis were balanced because every open parenthesis had a corresponding closed parenthesis to pop it.

Applications

e.g. ((a+b)*(c*d)+(x+y))

stack = [ ]

① stack = ["("]

② stack = ["(", "("]

③ stack = ["("]

④ stack = ["(", "("]

⑤ stack = ["("]

⑥ stack = ["(", "("]

⑦ stack = ["("]

⑧ stack = [ ] ← empty

→ parenthesis are matched!

(a + b + (c + d)

stack = [ ]

① stack = ["("]

② stack = ["(", "("]

③ stack = ["("]

→ missing a closing parenthesis

stack - L.I.F.O

Applications

$a + b) + (c + d)$ (1) (2) (3)

stack = [ ]

(1) Error gets raised because we attempt to pop from an empty stack

(2)

(3)

→ missing an initial open parenthesis

(1) (2) (3)
$(a + b) + c + d)$

stack = [ ]

(1) stack = [" ("]

(2) stack = [ ]

(3)
Error gets raised because we attempt to pop from an empty stack

→ missing the open parenthesis!