Week04

- array list

Today is...
✨ **No-Laptop Monday!** ✨

2023 Midterm and Final on website

WARMUP
what is data?
what is a data structure?
why are data structures?

TODO: record lecture

where are we?

the finest lecture hall in all of Williams College

at the **turning point** of CS136

- ~~**part A:** programming fundamentals in Java~~
- **part B:** data structures
  - one data structure per week here we gooo 😀 👍

data structures

data

data is numbers

a **data structure** helps you organizes your data...

...the right data structure for a task is...
- easy to work with -- **programmer time**
- runs fast -- **runtime** (user time)

array list

motivation

arrays are super useful, but...

arrays are super useful, but...
🧠

main limitation of arrays: can't change length

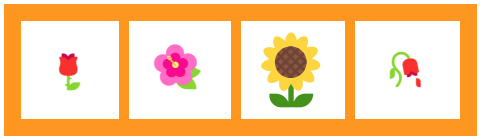
- an **array** is a fixed-length sequence of elements all of the same type
- very fast ($O(1)$ access), very simple
- but what if we don't know how many elements we need?
  - **solution A:** make a BIG array
    - `pool = new Thing[256]; // from HW03`
    - simple
    - might end up being...
      - **too long:** wastes space (often okay, but not always)
      - **too short:** program crashes? (bad bad very bad)
  - **solution B:** *grow the array* as needed (an **array list**)
    - pretty simple
    - pretty fast

review: array metaphor

an array is like a very organized person's flower planter

- one flower per square
- the planter can't change size

(it is made of artisinal woods or something)

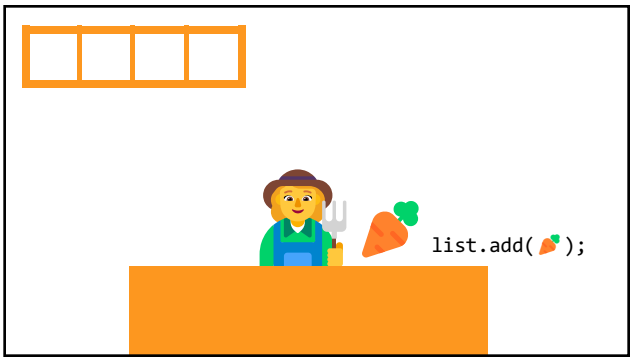


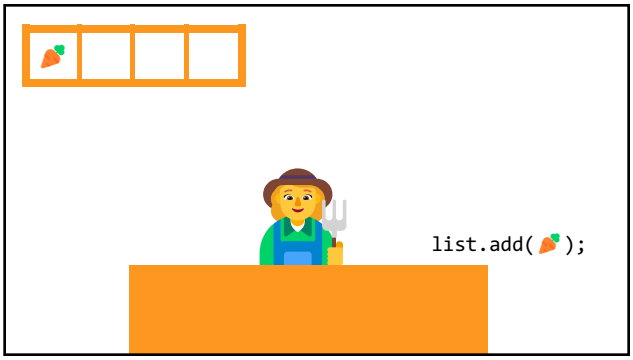
how an array list works (metaphor)

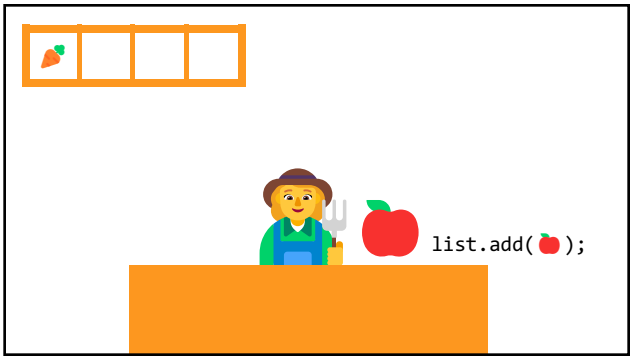
let us imagine...

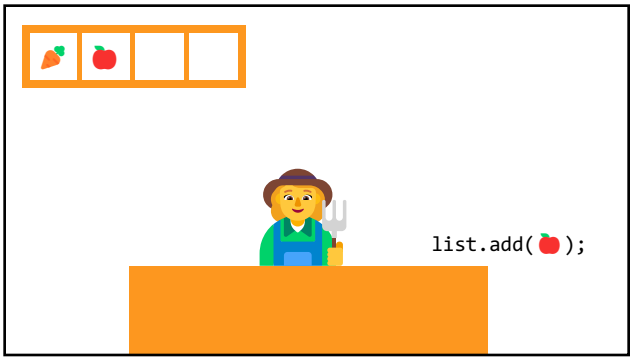
you have a bunch of pieces of produce you need to put somewhere for quick access, but you don't know how many

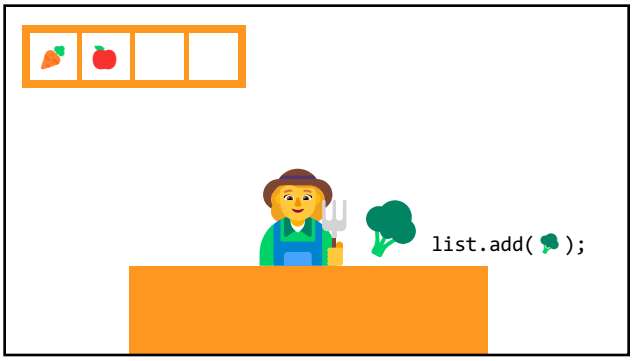
...off to the produce bank we go!

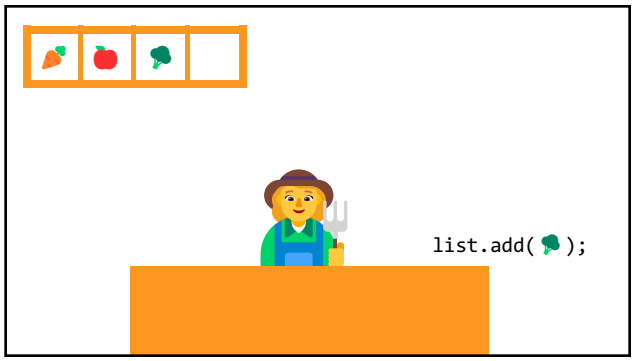


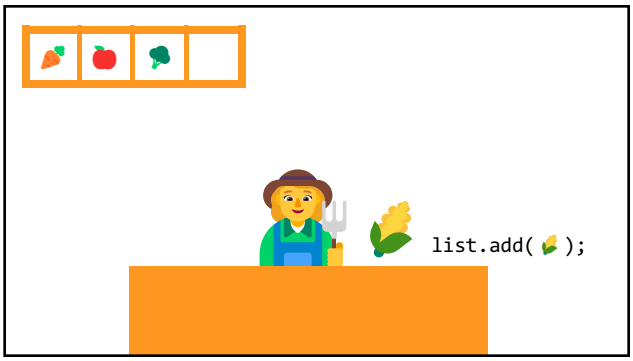


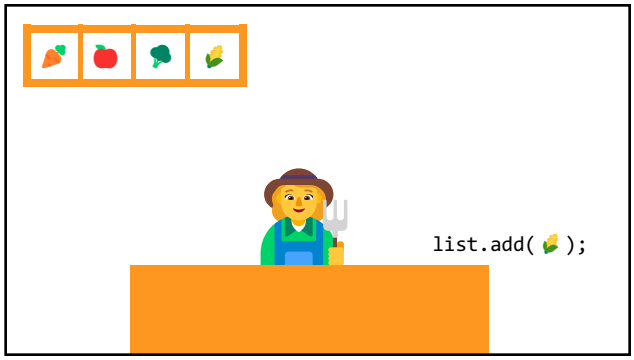


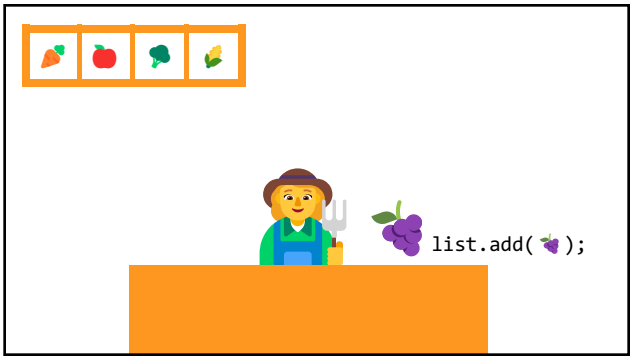


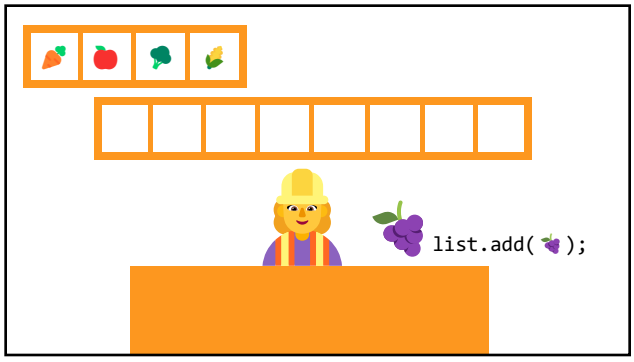


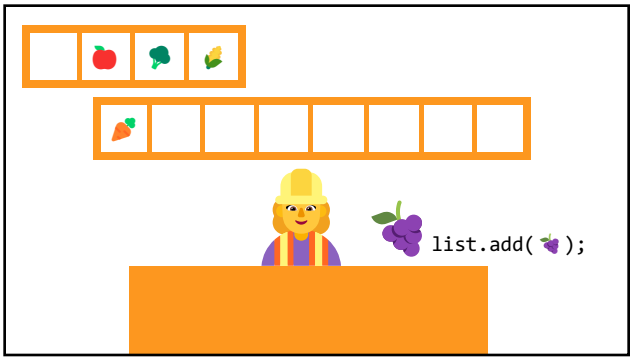

list.add( );

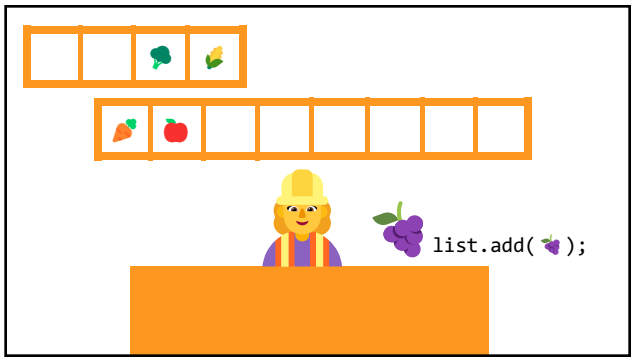

list.add( );

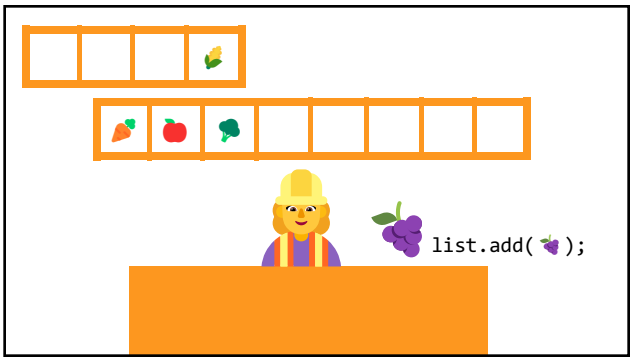

list.add( );

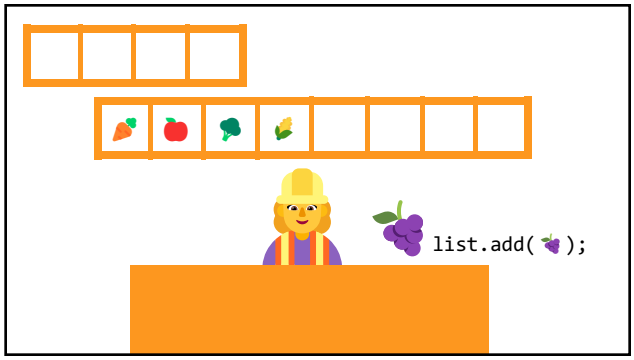

list.add( );

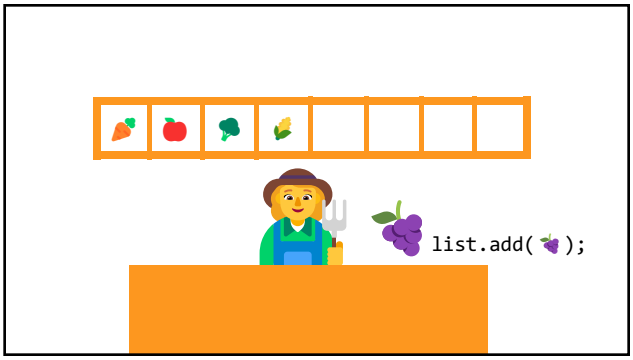

list.add( );

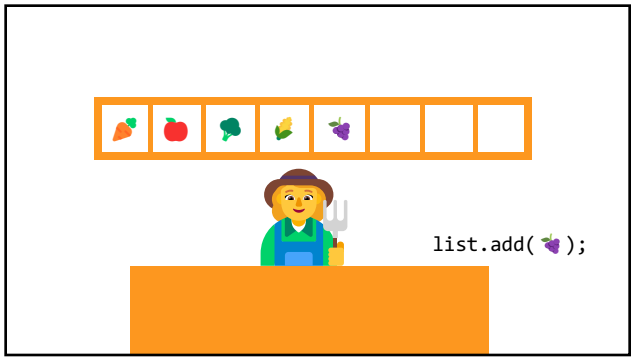

list.add( );

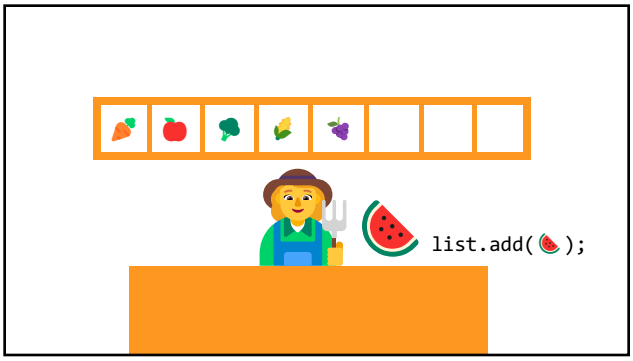

list.add( );

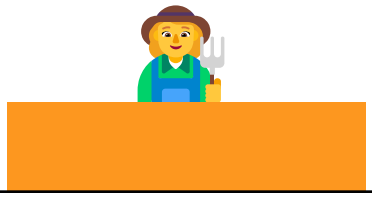

list.add( );


list.add( );


list.add( );


list.add( );


list.add( );



interface

an array list is a kind of list

- an **array list** (dynamic array, stretchy buffer, **vector**) is a kind of **list**
  - like an array, the user can **access** the i-th element in an array list
    - **get** the value of an element that is already there
    - **set** the value of an element that is already there
  - unlike an array, the user can always **add** a new element to an array list, no matter how many elements it already contains
    - **append** (push back) a new element to the end (back)
    - **insert** a new element at any valid index
  - the user can also **remove** elements from an array list

ArrayList<ElementType>

```
class ArrayList<ElementType> {
  void add(ElementType element); // to end
  void add(int index, ElementType element);
  ElementType get(int index);
  void set(int index, ElementType element);
  void remove(int index);
  int size();
  boolean isEmpty(); // (list.size() == 0)
}
```

<ElementType>

- Java's **`ArrayList<ElementType>`** is **generic**, which means you choose `ElementType` when you instantiate the class
  - `ArrayList<Thing> list = new ArrayList<>();`
  - `ArrayList<ArrayList<Thing>> lists = new ArrayList<>();`
  - `ArrayList<Integer> numbers = new ArrayList<>();`
    - **NOTE:** In between the angle brackets, use...
      - `Boolean` instead of `boolean`
      - `Character` instead of `char`
      - `Double` instead of `double`
      - `Integer` instead of `int`
      - 🤷

usage code

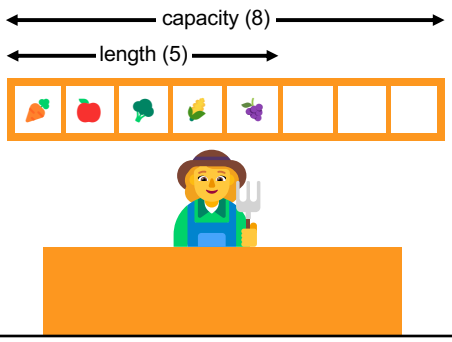
what does this code print? (the home game)

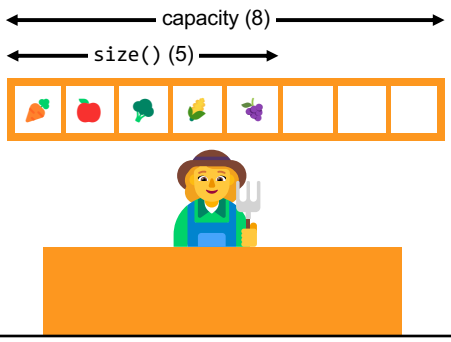
```
ArrayList<String> list = new ArrayList<>();
PRINT(list);
list.add("Hello");
list.add("World");
PRINT(list);
list.add(1, "Cruel"); // NOTE: insert so "Cruel" has index 1
PRINT(list);
PRINT(list.size());
PRINT(list.get(1));
list.remove(0);
PRINT(list);
list.set(1, "Summer");
PRINT(list);
```

usage code

```
ArrayList<String> list = new ArrayList<>();
PRINT(list);          // []
list.add("Hello");
list.add("World");
PRINT(list);          // [Hello, World]
list.add(1, "Cruel");
PRINT(list);          // [Hello, Cruel, World]
PRINT(list.size());   // 3
PRINT(list.get(1));   // Cruel
list.remove(0);
PRINT(list);          // [Cruel, World]
list.set(1, "Summer");
PRINT(list);          // [Cruel, Summer]
```

size (length) of the *list* vs. the *list's capacity*





the internal array

- an array list's **length** (**size**) is the number of elements stored in the list
- an array list stores its elements inside of an internal **array**
  - an array list's **capacity** is the length of this array
  - ✅ **length <= capacity**

```
class ArrayList<ElementType> {
    private int length; // NOTE: call size() to get this

    private ElementType[] internalArray;
    // NOTE: the list's capacity is internalArray.length
}
```

✅ length is not the same thing as capacity

- imagine `ArrayList<String> bestaurants;` with length (size) 3 and capacity 5
- if we could `PRINT(bestaurants.internalArray)`...
  `["Blango", "Sproot", "Sparket", null, null]`
  ...we would see `5 - 3 = 2` "empty slots"

Dinner | $20–30

I ordered general tso's chicken with rice and crab rangoons. Upon biting into the crab rangoon I realized there was absolutely no filling inside of it. Literally the entire thing was bread. When I pay $7 for crab rangoons I 1) expect them to be full of filling 2) there should be more than four. Also the general tso's chicken I got that was almost $15 was not full. My complete order ended up being $30 with tip. And in my opinion was a complete waste of time and money.
0/10 would absolutely never order here again.

implementation continued

`ArrayList() { ... } // constructor`

- a new array list should have...
  - `this.length = 0;`
  - `this.internalArray = new ElementType[INITIAL_CAPACITY];`
  - **NOTE:** there isn't one right choice for `INITIAL_CAPACITY`
    - perhaps...`0`?
    - perhaps `8`?
    - 🤷

`ElementType get(int index) { ... }`

- to **get** an element with a given `index`...
  - `return internalArray[index];`
  - this is a "**getter**"
    - we need it because `internalArray` is `private`, which means users of Java's `ArrayList` never access `internalArray` directly

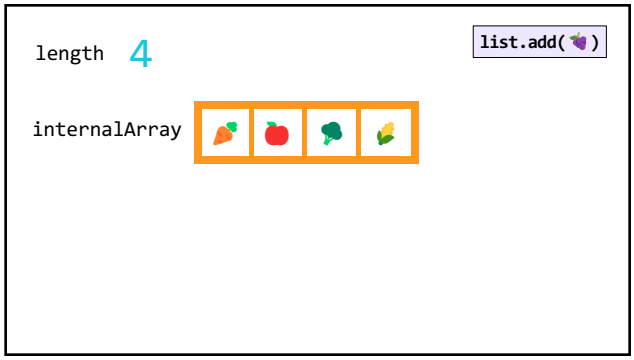
`void add(ElementType element) { ... }`

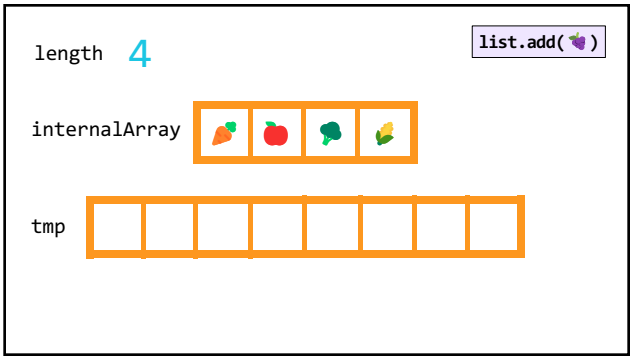
- to **append** (push back) a new element to the back (end) of an array list...
  - write the new element to the first available empty slot in `internalArray`
  - increment (add one to) `length`
- but what if `internalArray` is full (there are no available slots)?

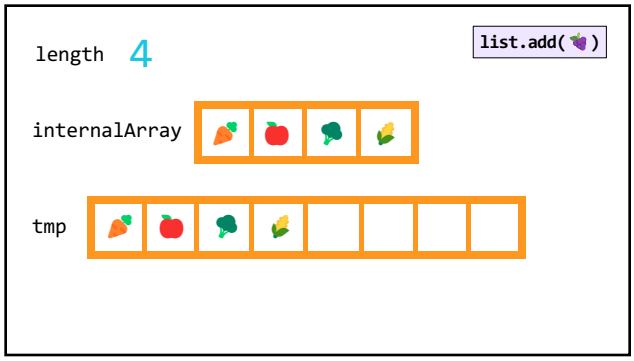
add (details)

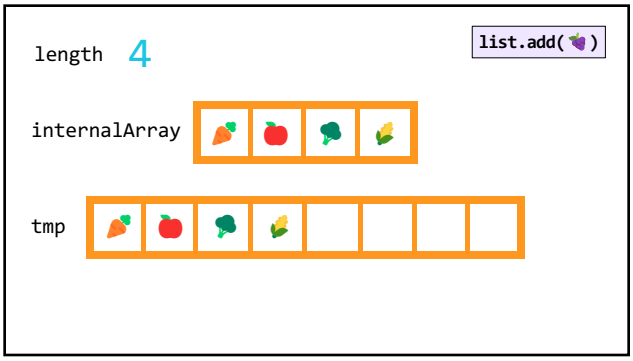
```
void add(ElementType element) { ... }
```

- to **append** (push back) a new element to the back (end) of an array list...
  - if `internalArray` is full...
    - make a new array two times the length of the current internal array
    - copy the elements of the current internal array into this new array (using a for loop)
    - update the `internalArray` reference to refer to this new array
  - write the new element to the first available empty slot in `internalArray`
  - increment `length`









length 4

list.add(🍇)

internalArray

length 5

list.add(🍇)

internalArray

```
void add(int index, ElementType element)
```

- to **insert** a new element so it has a given `index`...
  - `ASSERT` that the given `index` is valid!
    - **NOTE:** `0` is OK (**"prepend"**)
    - **NOTE:** `length` is OK (**append**)
    - **NOTE:** `-1` is BAD BAD VERY BAD (**out of bounds**)
    - **NOTE:** `(length + 1)` is BAD BAD VERY BAD (**out of bounds**)
  - if internal array is full, resize (see previous slide)
  - make room for the new element
    - move elements with indices greater than or equal to index one slot to the right
      ☠ **like inserting in HW02 (text box), need to iterate backwards**
  - write new element and increment length

length 4

list.add(🍋 , 2)

internalArray

length 4

list.add(🍋 , 2)

internalArray

length 4

list.add(🍋 , 2)

internalArray

length 5

list.add(, 2)

internalArray

☠️ like in HW02 (text box), need to iterate backwards

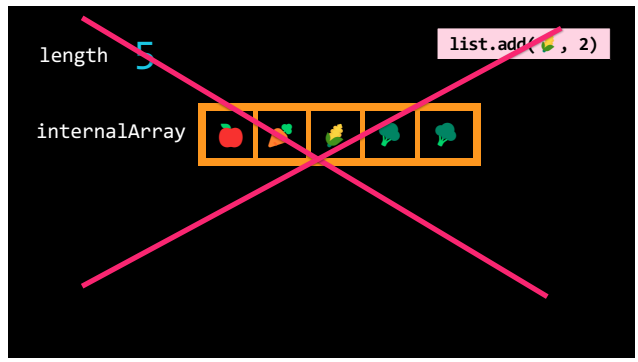
ep|elia

eph|eeee

length 4

list.add(, 2)

internalArray



final note on add

final note on add

- the special-case add to end (append)
  can be implemented using the general-purpose add (insert)

```
void add(ElementType element) {
    add(this.length, element);
}
```

- **however, if i were implementing an array list from scratch,**
  i would still implement the less general version first!
  (it is simpler, and i will learn stuff by implementing it)
  - (baby steps)

Week04

Today is...
✨ Laptop Wednesday! ✨

- ~~array list~~
- array list tutorial

♫ 2023 Midterm and Final on website
2024 Midterm and Final dates on website ♫

WARMUP
get the Tut04 starter code running on your laptop

TODO: record lecture

review animations from Week04a

Tut04

Week04c

Today is...
✨ Fun Friday! ✨

- ~~array list~~
- ~~array list tutorial~~
- array list runtime

WARMUP
what does this code print? why?

```
if (2 + 2 == 5); {
    PRINT("hmm...");
}
```

TODO: record lecture

big-O runtime of

`void add(ElementType e);`

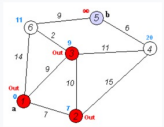
review: big-O runtime

- **big-O runtime** (**running time**, **time complexity**) gives us a big-picture idea of how long a function will take to run (execute, finish)
  - a function that operates on n *things*
    (**NOTE:** we assume n is LARGE)
    could be...
    - $\mathcal{O}(1)$ (**constant time**) 😀 FAST
    - $\mathcal{O}(\log_2 n)$ (**logarithmic time**, **log time**) 😊 fast
    - $\mathcal{O}(n)$ (**linear time**) 😐 meh
    - $\mathcal{O}(n^2)$ (**quadratic time**) 😟 slow
    - $\mathcal{O}(2^n)$ (**exponential time**) 😩 SLOW

big-O runtime

- **big-O runtime** can get complicated
  - sometimes there aren't just n *things*...
    - there are m thing A's, n thing B's, etc.
      - 🔮 **stay tuned for CSCI 256!**
  - sometimes we get more specific about *when* a function takes a certain length of time to run
    - in the **worst case**?
    - in the **best case**?
    - ~~in the average case?~~
    - in the "long run"? (**amortized**)

Dijkstra's algorithm



Dijkstra's algorithm to find the shortest path between *a* and *b*. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.

Class	Search algorithm Greedy algorithm Dynamic programming[1]
Data structure	Graph Usually used with priority queue or heap for optimization[2][3]
Worst-case performance	$\Theta(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$[3]

best case and worst case runtime of adding to the back of an array list

best case

- the **best case** is the shortest time a function can possibly take to run
  - for adding an element to the back of an array list, this is when the array list's internal/private array still has at least one "empty slot"
    - in this case, we have to...
      - write an element to an array $\mathcal{O}(1)$
      - increment a counter + $\mathcal{O}(1)$
      - ---
      - $\mathcal{O}(1)$ 😄

worst case

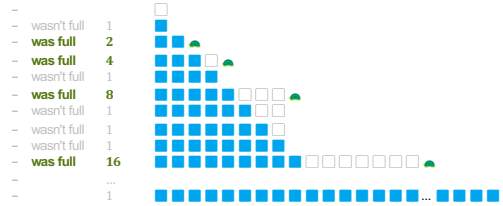
- the **worst case** is the longest time a function can possibly take to run
  - for adding an element to the back of an array list, this is when the array list's internal/private array is full (with n elements)
    - in this case, we have to...
      - allocate an array of $2n$ elements $\mathcal{O}(n)$
      - copy of over n elements + $\mathcal{O}(n)$
      - write an element to an array + $\mathcal{O}(1)$
      - increment a counter + $\mathcal{O}(1)$
      - updating reference to internal array + $\mathcal{O}(1)$
      - ---
      - $\mathcal{O}(n)$ 😄

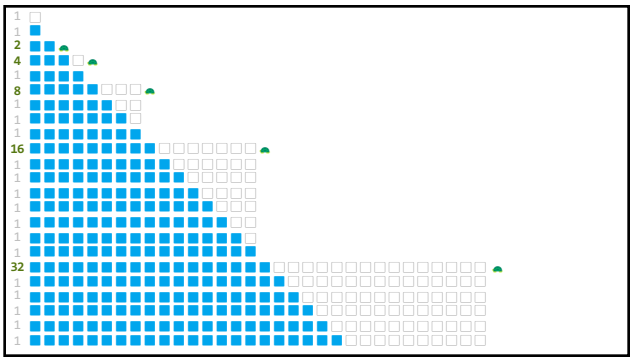
sometimes the best case and worst case are very different

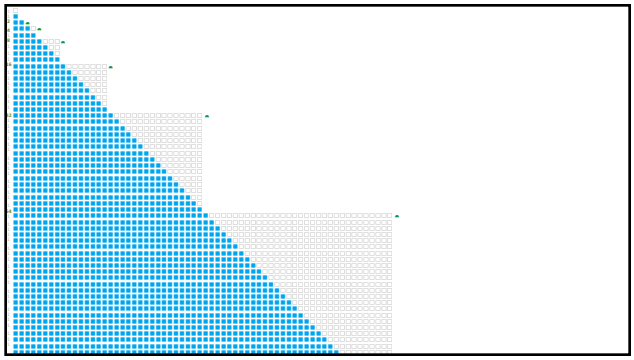
amortized runtime of adding to the back of an array list

amortized

- the **amortized runtime** is how long a function takes (on average) "in the long run"
  - for adding an element to the back of an array list...







amortized

- the **amortized runtime** is how long a function takes (on average) "in the long run"
  - for adding an element to the back of an array list...
    - $\frac{1+2+4+1+8+1+1+1+16+1+1+1+1+1+1+1+32+1+1+1+1+1+1+1+1}{n}$
    - $\frac{\mathcal{O}(1+\cdots+1)}{n}+\frac{\mathcal{O}(1+2+4+8+16+\cdots n)}{n}$
    - $\frac{\mathcal{O}(n)}{n}+\frac{\mathcal{O}(2n-1)}{n}$
    - $\frac{\mathcal{O}(n)}{n}+\frac{\mathcal{O}(n)}{n}$
    - $\mathcal{O}(1)$ 😀

the amortized run time is less "pessimistic" than worst case

the most relevant runtime depends on **context**

- when might worst case be more relevant?
- when might amortized be more relevant?

best case and worst case runtime of inserting into the *front* of an array list

best case and worst cast

- the **best case** of inserting an element into the front of an array list is when the array list's internal array still has at least one "empty slot"
  - in this case, we have to...
    - "move over" n elements $\mathcal{O}(n)$
    - write an element to an array + $\mathcal{O}(1)$
    - increment a counter + $\mathcal{O}(1)$
- the **worst case** of inserting an element into the front of an array list is when the array list's internal array is full
  - in this case, we have to...
    - allocate an array of $2n$ elements $\mathcal{O}(n)$
    - copy of over n elements + $\mathcal{O}(n)$
    - write an element to an array + $\mathcal{O}(1)$
    - increment a counter + $\mathcal{O}(1)$

lesson?: sometimes best case and worst case are the same

choose your own adventure!

deleting while iterating backwards, grugbrain, comments (including stb), Vim, Kahoot! (including Jim's unpublished Kahoot op-ed)

https://grugbrain.dev