MIT
6.102 Software Construction
Max Goldman, Prof. Mitchell Gordon, and Prof. Rob Miller

solutions

# 6.102 Spring 2026 Exam 1

- Please arrive to the exam with this page open and **all other apps, windows, and tabs already closed**.
- Your exam location is:
- You have **80 minutes** to complete this exam. There are **5 problems** that vary in length.
- The exam is **closed-book** and closed-notes, but you are allowed to bring a single 8.5×11″ double-sided page of notes, handwritten directly on the paper (not computer-printed or photocopied), readable without a magnifying glass, created by you.
- You may also use blank scratch paper. Staff cannot provide scratch paper.
- You may use **nothing else on your computer** or other devices: no 6.102 website; no TypeScript interpreter or programming tools; no web search or discussion with other people.
- Before you begin: you must **check in** by having the course staff scan the QR code at the top of the page.
- This page **automatically saves your answers** as you work. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the exam.
- If you feel the need to **write a note to the grader**, or if you want to write a note to yourself, you can click the gray pencil icon to the right of the answer.
- If you have a question, or need to use the restroom, please raise your hand.
- If you find yourself bogged down on one part of the exam, remember to keep going and work on other problems, and then come back.
- To **leave early**: enter *done* at the very bottom of the page, and show your screen with the check-out code to a staff member.
- You **may not discuss** details of the exam with anyone other than course staff until exam grades have been assigned and released.

Good luck!

Arrays in JavaScript/TypeScript, like lists in Python, grow to accommodate new elements without bound. This exam considers **fixed-size buffers** whose capacity is set at construction time: once they reach capacity, they do not grow further. The code used throughout this exam is at the **bottom of this page**, and you can **open it in a separate tab**.

Alyssa asks the AI for an implementation of a fixed-size buffer, and it provides the code for **`BufferA`**.

Bobby also asks the AI for a fixed-size buffer implementation, and it provides the code for **`BufferB`**.

The AI doesn't know any better, so it generates code assuming TypeScript's *no unchecked indexed access* option is **off**. That means indexing into an `Array<Element>` has static return type `Element`, not `Element|undefined`, even though the runtime value could be `undefined`! (We had *no unchecked indexed access* off for PS0 and PS1. We turned it on for PS2.)

Despite the missed opportunity for static checking, the code for `BufferA` and `BufferB` is correct.

You can see that the two implementations are very similar:

- The constructors are exactly the same. In their code, `new Array(capacity)` creates a new sparse array with length `capacity`, and then `fill(undefined)` fills in that array with `undefined` at every index so it is no longer sparse.
- `BufferA` provides `addLast(..)` and `BufferB` provides `addFirst(..)`, each implemented with the same code.
- `at(..)` and `toArray()` each have the same spec in the two classes, and they each have similar (but not identical) code.

Alice can use `BufferA` like so:

```
const things = new BufferA<string>(2);
things.addLast("Thing 1");
console.log(things.at(0)); // "Thing 1"
things.addLast("Thing 2");
things.addLast("Thing 3");
console.log(things.at(0)); // "Thing 2"
console.log(things.toArray()); // ["Thing 2",
                                  "Thing 3"]
```

... and Bobby can use `BufferB` like so:

```
const things = new BufferB<string>(2);
things.addFirst("Thing 1");
console.log(things.at(0)); // "Thing 1"
things.addFirst("Thing 2");
things.addFirst("Thing 3");
console.log(things.at(0)); // "Thing 3"
console.log(things.toArray()); // ["Thing 3",
                                  "Thing 2"]
```

Yasmin wants a buffer that supports both `addLast(..)` and `addFirst(..)`, and starts writing the code for **`BufferY`**, which is not complete yet.

- Note the differences in the rep and constructor compared to the previous two implementations.

Finally, we will discuss Ryan's **`BufferR`** starting in Problem 3.

# Problem ×1

Before you proceed, make sure you understand the abstraction provided by each of `BufferA`, `BufferB`, and `BufferY`. Their code is at the bottom of this page, and you can open it in a separate tab. Recall the essential idea that an abstract data type is defined by its:

| operations |
|---|

Write an excellent spec, in TypeDoc format, that works for *all three* of the constructors for `BufferA`, `BufferB`, and `BufferY` as they are implemented:

```
/**
 *   Make a new empty fixed-size buffer of the given capacity.
 *   @param capacity nonnegative integer maximum number of elements
 *
 *
 *
 */
```

Write an excellent spec, in TypeDoc format, that works for *both* `BufferA.addLast` and `BufferY.addLast` as they are implemented:

```
/**
 *   Modify this buffer to add the given element at the end. If the buffer would have more than
 *   capacity elements, also removes the element at index 0.
 *   @param elt the element to add
 *
 *
 */
```

Write an excellent rep invariant and abstraction function for `BufferA`. Write them clearly as functions *RI(...) = ...* and *AF(...) = ...*:

Rep invariant:

RI(capacity, count, buff) = capacity is a nonnegative integer
and count is a nonnegative integer
and buff.length = capacity
and buff[i] != undefined for all 0 <= i < min(count, capacity)

Abstraction function:

AF(capacity, count, buff) = the fixed-size buffer of capacity capacity
that contains min(count, capacity) elements, where the element at index i
is buff[(max(count, capacity) + i) % capacity]

*Must describe exactly one abstract fixed-size buffer with its particular sequence of elements.*

Will the rep invariant and the abstraction function for `BufferB` be identical, or will one or both be different?

- (●) RI for `BufferB` is identical to the RI for `BufferA`
- ( ) RI for `BufferB` is different

... and if different, explain why in one clear sentence: *(you do not need to write out the `BufferB` rep invariant)*

- ( ) AF for `BufferB` is identical to the AF for `BufferA`
- (●) AF for `BufferB` is different

... and if different, explain why in one clear sentence: *(you do not need to write out the `BufferB` abstraction function)*

For a BufferA rep value R that represents abstract buffer A, that same rep value R for Buffer B represents a buffer with its elements in reverse order from A.

# Problem ×2

Complete the implementation of `BufferY` with straightforward, readable code that assumes *no unchecked indexed access* is **on**:

```
public at(idx: number): Element {
```

```
if (idx < 0 || idx >= this.buff.length) {
    throw new Error();
}
return this.buff[idx] ?? assert.fail();
```

```
}
```

```
public toArray(): Array<Element> {
```

```
return this.buff.slice();
```

```
}
```

Write an excellent safety from rep exposure argument for `BufferY` that accords with your completed implementation:

Safety from rep exp.:

The rep fields are capacity and buff:
capacity is unreassignable and immutable
buff is unreassignable
buff is of mutable type Array, the only operation to take or return an Array is toArray, where we make a copy
Element values in buff may be mutable, but we rely only on their object identity

# Problem ×3

Notice that for `BufferA` and `BufferB`, each of which only provides one of `addLast(..)` or `addFirst(..)`, adding a new item always discards the oldest item (if necessary), because items are only added at one end of the buffer. But that is *not* true of `BufferY`, because `BufferY` provides both `addLast` and `addFirst`, and always discards the item at the opposite end of the buffer (if necessary).

Ryan uses the fact that `Map` remembers the insertion order of its key-value pairs to start implementing `BufferR` that supports both `addLast(..)` and `addFirst(..)`, but also always discards the oldest item (if necessary) in both.

The code compiles and runs, but there is a bug in the provided `BufferR.trim()`. Explain the bug in one clear sentence:

> The "return" inside of the function argument to buff.forEach does not return from trim(), it only returns from that single function invocation by forEach!

Say what effect it will have for the clients of `BufferR` in another one clear sentence.
If you are not sure, state the source of your uncertainty, then make a reasonable guess that is as clear as possible:

> When the buffer is at capacity and the client adds another item, the buffer becomes empty instead.
>
> *If unsure, we might speculate that delete during forEach will instead raise an error or delete only some keys when adding an element at capacity.*

And fix the bug:

```
private trim(): void {
    if (this.buff.size > this.capacity) {
        for (const key of this.buff.keys()) {
            this.buff.delete(key);
            return;
        }
    }
}
```

Complete the `BufferR` implementation of `toArray()` in exactly one additional line of code:

```
public toArray(): Array<Element> {
    const entries: Array<[number, Element]> = [...this.buff.entries()];
    entries.sort((a, b) => a[0] - b[0]);
    return entries.map(entry => entry[1]);
}
```

*(You are not asked to complete the `BufferR` implementation of `at(..)`.)*

Write an excellent abstraction function for `BufferR`. Write it clearly as a function *AF(...) = ...*:

Abstraction function:

> AF(capacity, buff) = the fixed-size buffer of capacity capacity where
> the elements are the values in the key/value pairs of buff,
> ordered by the numerical value of the keys,
> and added in the insertion order of those pairs
>
> *Must describe exactly one abstract fixed-size buffer with its particular sequence of elements and memory of the order in which they were added.*

# Problem ⤭4

Suppose we would like `BufferA`, `B`, `Y`, and `R` to all implement an interface called `FixSizeBuffer`:

```
export interface FixSizeBuffer<Element> {
    ...
}
```

What could we include in that interface? Pick all that apply:

⤭ ☐ An overall spec stronger than any of the implementations
a. ■ An overall spec weaker than any of the implementations

Explain your answer for these options in one clear sentence:

The specs of the implementations must be stronger, not weaker, than those of the interface in order to be subtypes.

⤭ ☐ The RIs of the implementations
b. ☐ The AFs of the implementations
☐ The SREs of the implementations

Explain your answer for these options in at most two sentences:

All of these relate to a particular rep, which the interface does not define. It provides only a spec.

⤭ ☐ A declaration for `buff`
c. ☐ A spec for the constructor
☐ An implementation for the constructor
☐ Specs for `addLast` and `addFirst`
☐ Implementations for `addLast` and `addFirst`
■ Specs for `at` and `toArray`
☐ Implementations for `at` and `toArray`

Explain your answers for these options in at most three sentences:

We do not include the rep or any implementation code in an interface, and we cannot use an interface to define constructors. We can include the specs for at and toArray. Since BufferA and BufferB do not implement addFirst and addLast, respectively, we cannot include those methods in the interface.

# Problem ⤭5

Imagine we would like to build a digital picture frame that uses a `FixSizeBuffer<Photo>` to store the photos. We have an interface for photos, where you should assume reasonable specs in the places we have omitted them:

```
/**
 * An immutable photo.
 */
export interface Photo {
    dimensions(): Size;
    getPixels(): ReadonlyArray<ReadonlyArray<Color>>;
    equalValue(other: Photo): boolean;
}

export type Size = readonly [number, number];
export type Color = readonly [number, number, number];
```

(The keyword "`readonly`" in the definitions of `Size` and `Color` has the same effect as `ReadonlyArray` for those fixed-length array types.)

⤭a. One of the intrepid fixed-sized buffer authors adds an implementation for photos that also have a short video snippet, called *e.g.* "live photos" or "motion photos" on various platforms:

```
// A still photo with a short snippet of video that was recorded before and after it.
export class LivePhoto implements Photo {
    // ... fields, private methods, etc. ...

    /**
     * The number of frames of video in this "live" photo,
     * where frame number  floor(frameCount/2)  is the still photo.
     */
    public readonly frameCount: number;

    /**
     * Set the frame returned by getPixels(), which initially returns the still photo frame.
     * @param frame the integer frame number, 0 <= frame < this.frameCount
     */
    public show(frame: number): void { ... }

    /** @inheritdoc */
    public dimensions(): Size { ... }

    /** @inheritdoc */
    public getPixels(): ReadonlyArray<ReadonlyArray<Color>> { ... }

    /** @inheritdoc */
    public equalValue(other: Photo): boolean { ... }
}
```

According to TypeScript's structural subtyping, `LivePhoto` is a subtype of `Photo`. Explain clearly and succinctly why it is not a subtype according to specifications:

> Photo is immutable, but LivePhoto.show(..) is a mutator since it changes what getPixels() will return!

Propose an excellent fix by explaining changes to the spec of `LivePhoto` clearly and succinctly:

> getPixels() always returns the still photo. Replace show(..) with an operation videoFrame(frame: number) that takes the frame number (like show does) and returns the 2D-array of pixels for that frame.

⤭b. We would like to ensure that the `FixSizeBuffer<Photo>` in our photo frame does not contain duplicate photos, even if *e.g.* the photo is re-uploaded to the frame, creating a new `Photo` object. Unfortunately, this is not how any of our `FixSizeBuffer` implementations work.

We might create a new class like this:

```
export class FixSizeUniqPhotoBuffer {
    // ... fields, constructor, other methods ...

    // only add the photo if it is not .equalValue(..) to a current photo
    public addLast(photo: Photo): void { ... }
}
```

But if we anticipate that we might encounter this problem of unique items again (maybe with videos, or likes, or users, ...), `FixSizeUniqPhotoBuffer` is not *ready for change*. Let's plan a different approach:

```
1  export class FixSizeUniqBuffer<Element extends Equatable<Element>> {
.      ...
.      ...
.  }

2  export interface Equatable<Other> {
3      /** ... TODO ... */
4      equalValue(other: Other): boolean;
5  }

6  export interface Photo extends Equatable<Photo> {
7      /** ... TODO ... */
8      equalValue(other: Photo): boolean;
.      // ... other methods ...
.  }
```

On line 1, the syntax "`<Element extends Equatable<Element>>`" is a new kind of generic declaration. It means that `Element` is a generic placeholder for a type that is chosen by the client — but requires that the type must be a subtype of `Equatable<Element>`. Neat!

Explain the purpose of `Equatable` and its contents (lines 2–5) in one clear sentence:

> e.g. A subtype of Equatable<Other> will be comparable by equalValue to values of type Other.
> -or-
> Defines an interface for types that provide an equalValue operation allowing comparison with type Other.

Explain the purpose of "`Photo extends Equatable<Photo>`" (on line 6) in one clear sentence:

> e.g. Photo is a subtype of Equatable<Photo> and thus must include an equalValue that takes a Photo input.
> -or-
> All implementations of Photo will have an equalValue that allows them to be compared with other Photo values.

Write the method signature (and just the signature) of `addLast` as it will appear in this `FixSizeUniqBuffer` class:

```
public addLast(elt: Element): void;
```

---

You can open the code below in a separate tab.

# Fixed-size buffer implementations

Alyssa asks the AI for an implementation of a fixed-size buffer:

```
/**
 * Fixed-size buffer.
 * @template Element type of elements, may not be undefined.
 */
export class BufferA<Element> {

    private count: number;
    private readonly buff: Array<Element>;

    public constructor(public readonly capacity: number) {
        this.count = 0;
        this.buff = new Array(capacity).fill(undefined);
    }

    public addLast(elt: Element): void {
        this.buff[this.count % this.capacity] = elt;
        this.count++;
    }

    // Get the element at the given index, which must be an
    // integer. Throws an error for invalid indices.
    public at(idx: number): Element {
        if (idx < 0 || idx >= this.capacity) {
            throw new Error();
        }
        if (this.count < this.capacity) {
            if (idx >= this.count) { throw new Error(); }
            return this.buff[idx];
        }
        return this.buff[(this.count + idx) % this.capacity];
    }

    // Get the contents of this buffer as an array.
    public toArray(): Array<Element> {
        if (this.count < this.capacity) {
            return this.buff.slice(0, this.count);
        }
        const i = this.count % this.capacity;
        return this.buff.slice(i, this.capacity)
                  .concat(this.buff.slice(0, i));
    }
}
```

Bobby also asks the AI for a fixed-size buffer implementation:

```
/**
 * Fixed-size buffer.
 * @template Element type of elements, may not be undefined.
 */
export class BufferB<Element> {

    private count: number;
    private readonly buff: Array<Element>;

    constructor(public readonly capacity: number) {
        this.count = 0;
        this.buff = new Array(capacity).fill(undefined);
    }

    public addFirst(elt: Element): void {
        this.buff[this.count % this.capacity] = elt;
        this.count++;
    }

    // Get the element at the given index, which must be an
    // integer. Throws an error for invalid indices.
    public at(idx: number): Element {
        if (idx < 0 || idx >= this.capacity) {
            throw new Error();
        }
        if (this.count <= this.capacity && idx >= this.count) {
            throw new Error();
        }

        return this.buff[(this.count - idx - 1) % this.capacity];
    }

    // Get the contents of this buffer as an array.
    public toArray(): Array<Element> {
        if (this.count < this.capacity) {
            return this.buff.slice(0, this.count).toReversed();
        }
        const i = this.count % this.capacity;
        return this.buff.slice(i, this.capacity)
                  .concat(this.buff.slice(0, i)).toReversed();
    }
}
```

Yasmin starts writing a buffer that supports both `addLast` and `addFirst`:

```
/**
 * Fixed-size buffer.
 * @template Element type of elements, may not be undefined.
 */
export class BufferY<Element> {

    private readonly buff: Array<Element> = [];

    public constructor(public readonly capacity: number) {}

    public addLast(elt: Element): void {
        this.buff.push(elt);   // add elt at the end of this.buff
        if (this.buff.length > this.capacity) {
            this.buff.shift(); // remove one element from the front
        }
    }

    public addFirst(elt: Element): void {
        this.buff.unshift(elt); // add elt at the front of this.buff
        if (this.buff.length > this.capacity) {
            this.buff.pop();    // remove one element from the end
        }
    }

    // Get the element at the given index, which must be an integer.
    // Throws an error for invalid indices.
    public at(idx: number): Element {
        // ... TODO ...
    }

    // Get the contents of this buffer as an array.
    public toArray(): Array<Element> {
        // ... TODO ...
    }
}
```

Ryan starts writing a buffer that supports both `addLast` and `addFirst`, and remembers insertion order:

```
/**
 * Fixed-size buffer.
 * @template Element type of elements, may not be undefined.
 */
export class BufferR<Element> {

    private readonly buff: Map<number, Element> = new Map();

    public constructor(public readonly capacity: number) {}

    public addLast(elt: Element) {
        const key = Math.max(0, ...this.buff.keys());
        this.buff.set(key + 1, elt);
        this.trim();
    }

    public addFirst(elt: Element) {
        const key = Math.min(0, ...this.buff.keys());
        this.buff.set(key - 1, elt);
        this.trim();
    }

    private trim(): void {
        if (this.buff.size > this.capacity) {
            // Map.entries, forEach, etc. iterate in insertion order
            this.buff.forEach((value: Element, key: number) => {
                this.buff.delete(key);
                return; // done after deleting one
            });
        }
    }

    // Get the element at the given index, which must be an integer.
    // Throws an error for invalid indices.
    public at(idx: number): Element {
        // ... TODO ...
    }

    // Get the contents of this buffer as an array.
    public toArray(): Array<Element> {
        const entries: Array<[number, Element]> = [...this.buff.entries()];
        entries.sort((a, b) => a[0] - b[0]);
        // ... TODO ...
    }
}
```