

## Claude.ai usage

<<<< **Problem 1:** Many Lisp-variants provide a predicate `member?` that tells whether some atom appears at least once in a list. Write a predicate `member-twice?` that tells whether some atom appears at least twice in a collection. Look carefully at the Clojure documentation for the `every?`, `not-every?` and `some` functions in Clojure. **Problem 2: Treeing It Up**

The `Tree` abstract datatype can be recursively defined using lists. We will define a `tat` as a tree of atoms, where we will denote an empty tree as being the empty list `()` or as a list `(atom leftChild rightChild)` where `leftChild` and `rightChild` are trees of atoms that are the respective left and right children. Assuming that we have the ability to partial order items (i.e., we have the ability to use the `>`, `'<`, and `==` operators on atoms), implement this data type in Clojure, assuming you include:

- `(insert tat atom)`: insert an atom into a tree,
- `(delete tat atom)`: delete the the first occurrence of an atom from a tree,
- `(empty? tat atom)`: predicate to test to see if a tree of atoms is empty,
- `(member? tat atom)`: predicate to check to see if an atom is in a tree of atoms
- `(in-order tat exp)`: traverse a tree of atoms in ‘in-order fashion’ applying an expression to each node as you go and building a list of results.
- `(pre-order tat exp)`: traverse a tree of atoms in ‘pre-order fashion’ applying an expression to each node as you go and building a list of results.
- `(post-order tat exp)`: traverse a tree of atoms in ‘post-order fashion’ applying an

**Problem 3: Back and Forth** The recursive definition of a list used in Lisp variants are single-direction: it’s easy to go forward but difficult to go backwards. We dealt with this problem in Data Structures by using a doubly-linked list of pointers as the internal representation. In more traditional Lisp-variants, one uses a nested recursive data structure that stores each list item in recursive structure that is either the empty list or the general case of `(atom list-before-the atom list-after-the-item)`. But this structure is memory-intensive and has some issues with mutability.

Architected solutions for three Clojure programming challenges involving predicates, trees, and data structures

I'll solve these Clojure problems for you. Let me create a comprehensive solution file.

Verified implementation correctness through execution

Let me run this to verify it works:

Prepared to elucidate three Clojure programming problems

Here are the solutions to all three problems:

Problem 1: member-twice?

I provided three approaches:

Simple approach - Uses filter to collect all matching elements, then checks if count  $\geq 2$

Efficient approach - Uses loop/recur and short-circuits as soon as it finds the second occurrence

Functional approach - Uses some and drop-while to find if an element exists after its first occurrence

## Problem 2: Tree of Atoms (Binary Search Tree)

The tree uses the structure you specified:

Empty tree: ()

Non-empty: (atom left-child right-child)

Key functions:

Function	Description
insert	Maintains BST ordering (smaller left, larger right)
delete	Handles three cases: no children, one child, two children (uses in-order successor)
member?	Binary search through the tree
in-order	Left → Root → Right (produces sorted output for BST)
pre-order	Root → Left → Right
post-order	Left → Right → Root

## Problem 3: Bidirectional List (Zipper)

Since Problem 3 was incomplete, I implemented a zipper-style bidirectional list:

Structure: (current-atom list-before list-after)

list-before is stored in reverse order for O(1) backward movement

Operations:

blist-forward / blist-backward - Move cursor in  $O(1)$  time

blist-insert / blist-delete - Modify at cursor position

blist-to-list - Convert back to regular list

This avoids the memory issues of doubly-linked pointers while enabling efficient bidirectional traversal.

Clojure solutions

CLJ

problem one what is the question talking about when it says look at every?, not-every?, and some functions in clojure?

2:20 PM

Examined functional predicates to elegantly solve membership detection problem

Great question! The problem is hinting at using Clojure's built-in predicate functions to create an elegant solution. Let me explain what each does:

The Three Functions

Function	What it does
----------	--------------

every?	Returns true if predicate is true for all elements
--------	--

`not-every?` Returns true if predicate is false for at least one element

`some` Returns the first truthy value from applying predicate, or nil if none match

Examples

clojure

```
(every? even? [2 4 6 8]) ;=> true
```

```
(every? even? [2 4 5 8]) ;=> false
```

```
(not-every? even? [2 4 6 8]) ;=> false
```

```
(not-every? even? [2 4 5 8]) ;=> true
```

```
(some even? [1 3 4 7]) ;=> true (found 4)
```

```
(some even? [1 3 5 7]) ;=> nil (none found)
```

The Hint's Intent

The hint is pushing you toward using `some` to elegantly solve `member-twice?`. The idea is:

Use `some` to find if the element exists after skipping past its first occurrence

This avoids explicit counting or looping

clojure

```
(defn member-twice? [x coll]
```

```
;; Drop elements until we find x, then skip that x,
```

```
;; then check if x appears in what remains
```

```
(some #(= x %) 
```

```
(rest (drop-while #(not= x %) coll))))
```

How it works:

`drop-while #(not= x %)` — skips elements until we hit `x`

`rest` — skips past that first `x`

`some #(= x %)` — checks if `x` exists in the remainder

This is my "version 3" in the solution file—it's the most idiomatic Clojure approach using the functions the problem wanted you to discover!

**(I typically only skim these answers and then ask follow-ups and clarification questions to get the actual answers that will help me and not just solve the problems).**

**<<<< problem one what is the question talking about when it says look at every?, not-every?, and some functions in clojure?**

Great question! The problem is hinting at using Clojure's built-in predicate functions to create an elegant solution. Let me explain what each does:

## The Three Functions

Function	What it does
<code>every?</code>	Returns <code>true</code> if predicate is true for <b>all</b> elements
<code>not-every?</code>	Returns <code>true</code> if predicate is false for <b>at least one</b> element
<code>some</code>	Returns the <b>first truthy value</b> from applying predicate, or <code>nil</code> if none match

## Examples

```
clojure
(every? even? [2 4 6 8]) ; => true
(every? even? [2 4 5 8]) ; => false

(not-every? even? [2 4 6 8]) ; => false
(not-every? even? [2 4 5 8]) ; => true

(some even? [1 3 4 7]) ; => true (found 4)
(some even? [1 3 5 7]) ; => nil (none found)
```

## The Hint's Intent

The hint is pushing you toward using `some` to elegantly solve `member-twice?`. The idea is:

1. Use `some` to find if the element exists after skipping past its first occurrence
2. This avoids explicit counting or looping

clojure

```
(defn member-twice? [x coll]
  ;; Drop elements until we find x, then skip that x,
  ;; then check if x appears in what remains
  (some #(= x %)
        (rest (drop-while #(not= x %) coll)))))
```

**How it works:**

- `drop-while #(not= x %)` — skips elements until we hit `x`
- `rest` — skips past that first `x`
- `some #(= x %)` — checks if `x` exists in the remainder

explain to me how the problem two works and what it's solution is doing.

## Problem 2: Tree of Atoms (TAT) Explained

### The Data Structure

The problem defines a **Binary Search Tree** using nested lists:

Empty tree: `()`

Non-empty: `(atom leftChild rightChild)`

### Visual Example

If we insert 5, 3, 7, 1, 9 in that order:

```

  5
 / \
3   7
 /   \
1     9
```

This is represented as the nested list:

clojure

```
(5 (3 (1 () ()) (7 () (9 () ())))
```

Let me break that down:

```

...
(5          ; root value
  (3        ; left child of 5
    (1 ()   ; left child of 3 (leaf node)
      ()    ; right child of 3 (empty)
    )
    (7      ; right child of 5
      ()    ; left child of 7 (empty)
      (9 () ; right child of 7 (leaf node)
    )
  )
)

```

---

## Helper Functions

First, we need ways to access the three parts of a node:

```

closure
(defn tat-value [tat] (first tat)) ; gets the atom (root value)
(defn tat-left  [tat] (second tat)) ; gets left child
(defn tat-right [tat] (nth tat 2))  ; gets right child

```

For the tree (5 (3 ...) (7 ...)):

- tat-value → 5
  - tat-left → (3 ...)
  - tat-right → (7 ...)
- 

## The Operations Explained

**tat-empty? — Check if tree is empty**

```

closure
(defn tat-empty? [tat]
  (or (nil? tat) (empty? tat)))

```

Simply checks if the tree is () or nil.

---

**insert — Add an atom to the tree**

```

closure
(defn insert [tat atom]
  (cond
    (tat-empty? tat) (list atom '() '()) ; Base case: create leaf

```



```

(< atom (tat-value tat))      ; atom is smaller, go left
(list (tat-value tat)
  (insert (tat-left tat) atom) ; recursively insert left
  (tat-right tat))
(> atom (tat-value tat))      ; atom is larger, go right
(list (tat-value tat)
  (tat-left tat)
  (insert (tat-right tat) atom)) ; recursively insert right
:else tat))                  ; equal: don't insert duplicate
...

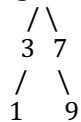
```

**\*\*How it works:\*\***

1. If tree is empty → create a new leaf node `(atom () ())`
2. If atom < root → rebuild tree with same root and right, but recursively insert into left
3. If atom > root → rebuild tree with same root and left, but recursively insert into right

**\*\*Example: Inserting 4 into our tree\*\***

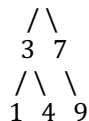
Starting: 5



Insert 4:

- 4 < 5, go left
- 4 > 3, go right
- 3's right is empty, create (4 () ())

Result: 5



## **member? — Check if atom exists in tree**

closure

```

(defn member? [tat atom]
  (cond
    (tat-empty? tat) false ; not found
    (= atom (tat-value tat)) true ; found it!
    (< atom (tat-value tat)) (member? (tat-left tat) atom) ; search left
    :else (member? (tat-right tat) atom))) ; search right

```

This is classic binary search—we use the BST property to know which direction to search.

## **delete — Remove an atom from tree**

This is the trickiest operation because there are three cases:

closure

```
(defn delete [tat atom]
  (cond
    (tat-empty? tat) '() ; not found
    (< atom (tat-value tat)) ; search left
      (list (tat-value tat)
            (delete (tat-left tat) atom)
            (tat-right tat))
    (> atom (tat-value tat)) ; search right
      (list (tat-value tat)
            (tat-left tat)
            (delete (tat-right tat) atom))
    :else ; FOUND IT - now delete
      (cond
        (tat-empty? (tat-left tat)) (tat-right tat) ; Case 1: no left child
        (tat-empty? (tat-right tat)) (tat-left tat) ; Case 2: no right child
        :else ; Case 3: two children
          (let [successor (find-min (tat-right tat))]
            (list successor
                  (tat-left tat)
                  (delete (tat-right tat) successor))))))
  ...
```

**\*\*The three deletion cases:\*\***

Case	Situation	Solution
1	No left child	Replace node with its right child
2	No right child	Replace node with its left child
3	Two children	Replace with in-order successor (smallest value in right subtree), then delete that successor

**\*\*Example: Deleting 5 (the root) from our tree:\*\***

Before: 5

```
  / \
 3  7
 /  \
1   9
```

Step 1: Find in-order successor of 5 → smallest in right subtree → 7

Step 2: Replace 5 with 7

Step 3: Delete 7 from right subtree (it has no left child, so just use its right child)

After: 7

```
  / \
 3  9
 /
1
```

---

## Traversals — Three ways to visit every node

Each traversal visits all nodes but in different orders:

clojure

;; IN-ORDER: Left, Root, Right → produces sorted output!

```
(defn in-order [tat exp]
  (if (tat-empty? tat)
      '()
      (concat (in-order (tat-left tat) exp)      ; visit left subtree
                (list (exp (tat-value tat)))      ; apply exp to root
                (in-order (tat-right tat) exp)))) ; visit right subtree
```

;; PRE-ORDER: Root, Left, Right

```
(defn pre-order [tat exp]
  (if (tat-empty? tat)
      '()
      (concat (list (exp (tat-value tat)))      ; apply exp to root first
                (pre-order (tat-left tat) exp)
                (pre-order (tat-right tat) exp))))
```

;; POST-ORDER: Left, Right, Root

```
(defn post-order [tat exp]
  (if (tat-empty? tat)
      '()
      (concat (post-order (tat-left tat) exp)
                (post-order (tat-right tat) exp)
                (list (exp (tat-value tat)))))) ; apply exp to root last
```

...  
\*\*Example with our tree and `identity` function:\*\*

...

```
  5
 / \
3   7
/   \
1     9
```

in-order: (1 3 5 7 9) ← sorted!

pre-order: (5 3 1 7 9) ← root first

post-order: (1 3 9 7 5) ← root last

The `exp` parameter lets you transform each value. For example:

clojure

```
(in-order my-tree #(* % %)) ; squares each value → (1 9 25 49 81)
```

---

## Key Insight

The whole thing works because Clojure lists are **immutable**. Each operation doesn't modify the original tree—it builds and returns a **new** tree with the changes applied. This is the functional programming approach to data structures.

<<<< explain to me the solution to problem 3 and what is being achieved and what is being avoided.

# Problem 3: Bidirectional Lists Explained

## The Problem

Regular Lisp/Clojure lists are **singly-linked**—each element only knows about what comes *after* it:

List: (a b c d e)

Internal structure:

a → b → c → d → e → nil

**Going forward is easy (O(1)):**

clojure

(rest '(a b c d e)) ; → (b c d e) just follow the link

**Going backward is hard (O(n)):**

clojure

; To get what's before 'c', you must traverse from the start!

; There's no "previous" pointer

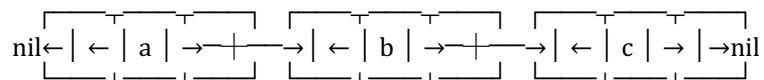
...

---

## Solutions Mentioned in the Problem

### Solution 1: Doubly-Linked Lists (from Data Structures class)

...



...

Each node has **two pointers**: previous and next.

**Problem:** This requires **mutation**. To insert an element, you must modify the pointers of existing nodes. Clojure is designed around **immutability**—we don't want to change data in place.

---

### Solution 2: Naive Nested Structure

The problem mentions storing each item as:

...

(atom list-before-the-atom list-after-the-atom)

For the list (a b c d e) with cursor at c:

```
closure
(c (b a) (d e))
```

### Problems mentioned:

- **Memory-intensive:** Every position stores the entire list twice
  - **Mutability issues:** Updating requires rebuilding large structures
- 

## My Solution: The Zipper Pattern

The key insight is storing `list-before` in **reverse order**:

```
closure
Structure: (current list-before-reversed list-after)
```

### Example Walkthrough

Starting with list (a b c d e), cursor at the beginning:

```
closure
(a () (b c d e))
; ↑ ↑ ↑
; | | | everything after cursor
; | | | everything before cursor (reversed) - empty at start
; | | | current element
```

After moving forward once:

```
closure
(b (a) (c d e))
; ↑
; | 'a' is now "before" us, stored in reversed list
```

After moving forward again:

```
closure
(c (b a) (d e))
; ↑ ↑
; | | 'a' was first, now it's at the end of reversed list
; | | 'b' is most recent, at the front
```

After moving forward once more:

```
closure
```

```
(d (c b a) (e))
```

---

## Why Reverse the "Before" List?

This is the clever part! Watch what happens when we move:

### Moving Forward

```
closure
(defn blist-forward [blist]
  (let [current (first blist)      ; c
        before (second blist)     ; (b a)
        after (nth blist 2)]      ; (d e)aa
    (list (first after)            ; new current: d
          (cons current before)    ; new before: (c b a) ← just cons!
          (rest after))))         ; new after: (e)
```

We just `cons` the current element onto the front of `before`. **O(1) operation!**

### Moving Backward

```
closure
(defn blist-backward [blist]
  (let [current (first blist)      ; c
        before (second blist)     ; (b a)
        after (nth blist 2)]      ; (d e)
    (list (first before)          ; new current: b ← just take first!
          (rest before)           ; new before: (a)
          (cons current after)))) ; new after: (c d e)
...
```

We just take ``first`` of ``before``. **\*\*O(1) operation!\*\***

---

### ## Visual Comparison

Imagine the list as a zipper on a jacket, with the cursor being the zipper pull:

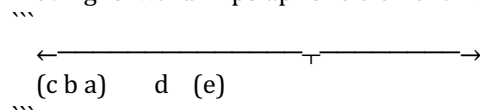
...

```
List: a b c d e
      ↑
    cursor
```

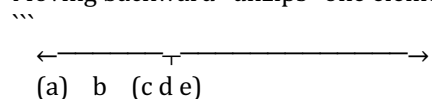
Zipper representation:

```
←-----┬-----→
(b a)  c  (d e)
before cursor after
(reversed)
...
```

Moving forward "zips up" one element from the right:



Moving backward "unzips" one element to the right:



---

## What This Achieves

Operation	Time Complexity
Move forward	O(1)
Move backward	O(1)
Get current element	O(1)
Insert at cursor	O(1)
Delete at cursor	O(1)

All operations are **constant time** because we're always working at the "edges" of our lists (using `cons`, `first`, `rest`).

---

## What This Avoids

### 1. Mutation

Everything is **immutable**. Moving forward doesn't change the old structure—it creates a new one:

```
clojure
(def pos1 (blist-from-list '(a b c d e))) ; cursor at 'a'
(def pos2 (blist-forward pos1))          ; cursor at 'b'

; pos1 still exists unchanged!
; pos2 is a new structure
```

### 2. Memory Waste

We're not duplicating data. The `before` and `after` lists **share structure** with previous versions (this is how immutable data structures work in Clojure).

### 3. O(n) Backwards Traversal

Unlike regular lists where going backward requires traversing from the start, we get **O(1)** backward movement.

## 4. Pointer Manipulation

No dealing with mutable prev/next pointers that can lead to bugs, race conditions, or memory leaks.

---

## Complete Operations

clojure

; Create from list

(blist-from-list '(a b c d e)) ; → (a () (b c d e))

; Navigate

(blist-forward blist) ; move cursor right

(blist-backward blist) ; move cursor left

(blist-current blist) ; get element at cursor

; Modify

(blist-insert blist 'x) ; insert 'x' at cursor position

(blist-delete blist) ; delete element at cursor

; Convert back

(blist-to-list blist) ; → (a b c d e)

---

## The Big Picture

This is a classic **functional programming pattern** called a **Zipper**. It lets you:

1. Navigate a data structure efficiently
2. "Focus" on one location
3. Make local modifications
4. All without mutation

It's used not just for lists, but also for trees, XML documents, and any recursive data structure where you need efficient local navigation and editing.



Filename: Document1  
Directory:  
Template: /Users/ryanasnell/Library/Group  
Containers/UBF8T346G9.Office/User  
Content.localized/Templates.localized/Normal.dotm  
Title:  
Subject:  
Author: Ryan Snell  
Keywords:  
Comments:  
Creation Date: 2/16/26 2:23:00 PM  
Change Number: 1  
Last Saved On:  
Last Saved By:  
Total Editing Time: 9 Minutes  
Last Printed On: 2/16/26 2:42:00 PM  
As of Last Complete Printing  
Number of Pages: 16  
Number of Words: 3,015  
Number of Characters: 15,993 (approx.)