CSE 250 Recitation

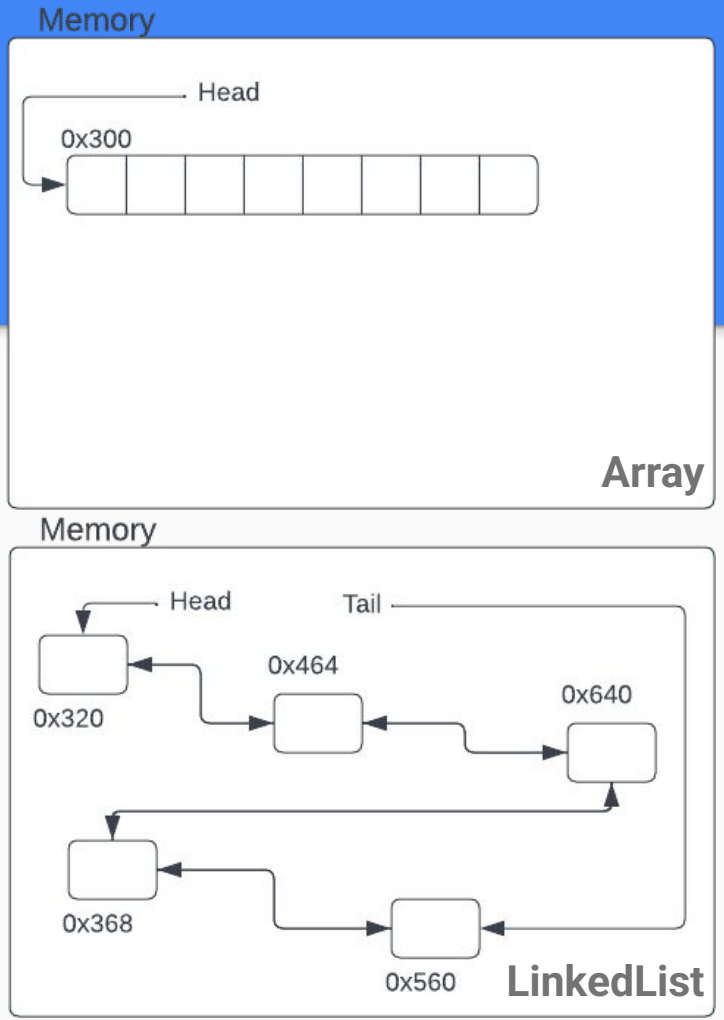
September 22 - 23: ADTs, Amortized Runtime

List ADT

Discussion: What is the runtime of `list.add(i, v)` when `List` is implemented using:

1. An `ArrayList`
2. A `LinkedList`
3. A `LinkedList` **and** you have a reference to the node representing index `i`

Breakdown the cost for each implementation



List ADT

An **`ArrayList`**

1. **If** out of space, create a new array and copy data	$O(n)$, Amortized $\Theta(1)$
2. Shift elements to the right to make space for new element	$O(n)$
3. Set element at index **`i`** to **`v`**	$\Theta(1)$
Total:	$O(n)$

A **`LinkedList`**

1. Iterate from the head node to the insertion point	$O(n)$
2. Create a new node	$\Theta(1)$
3. Assign next/prev pointers accordingly	$\Theta(1)$
Total:	$O(n)$

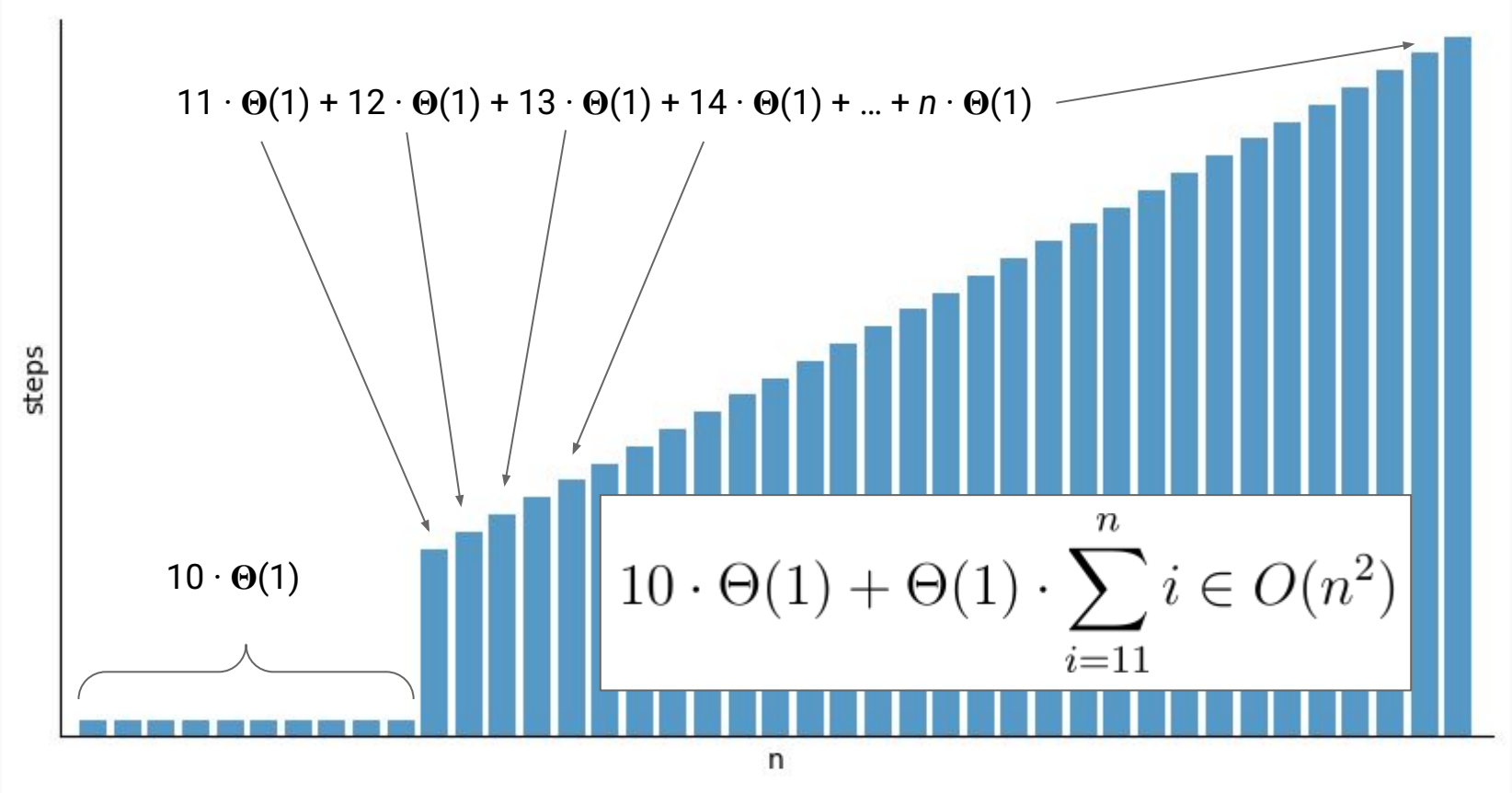
A **`LinkedList`** w/relevant reference

1. Create a new node	$\Theta(1)$
1. Assign next/prev pointers accordingly	$\Theta(1)$
Total:	$\Theta(1)$

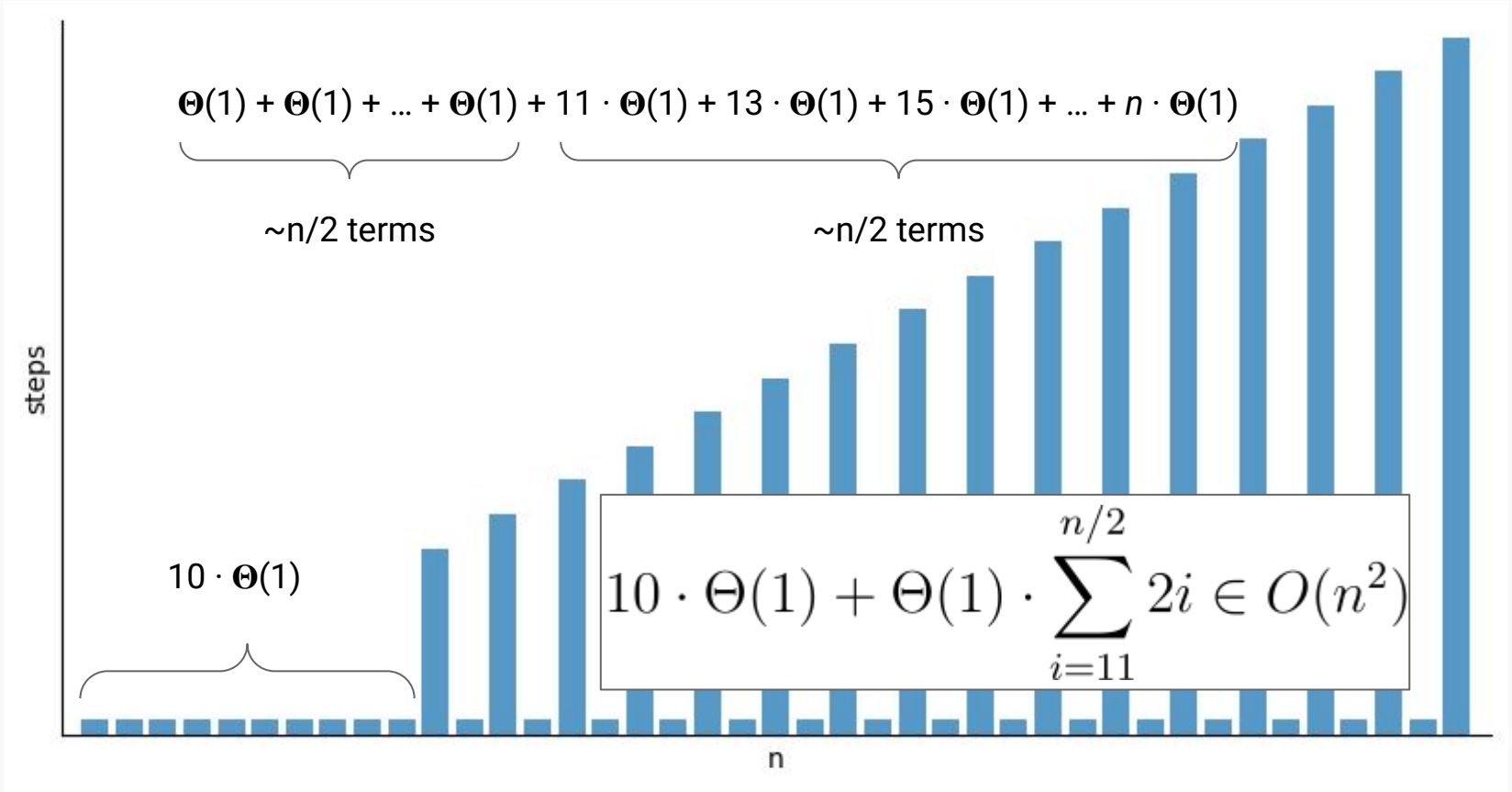
Amortized Runtime

If ***n*** calls to a function take **Θ(f(*n*))** steps

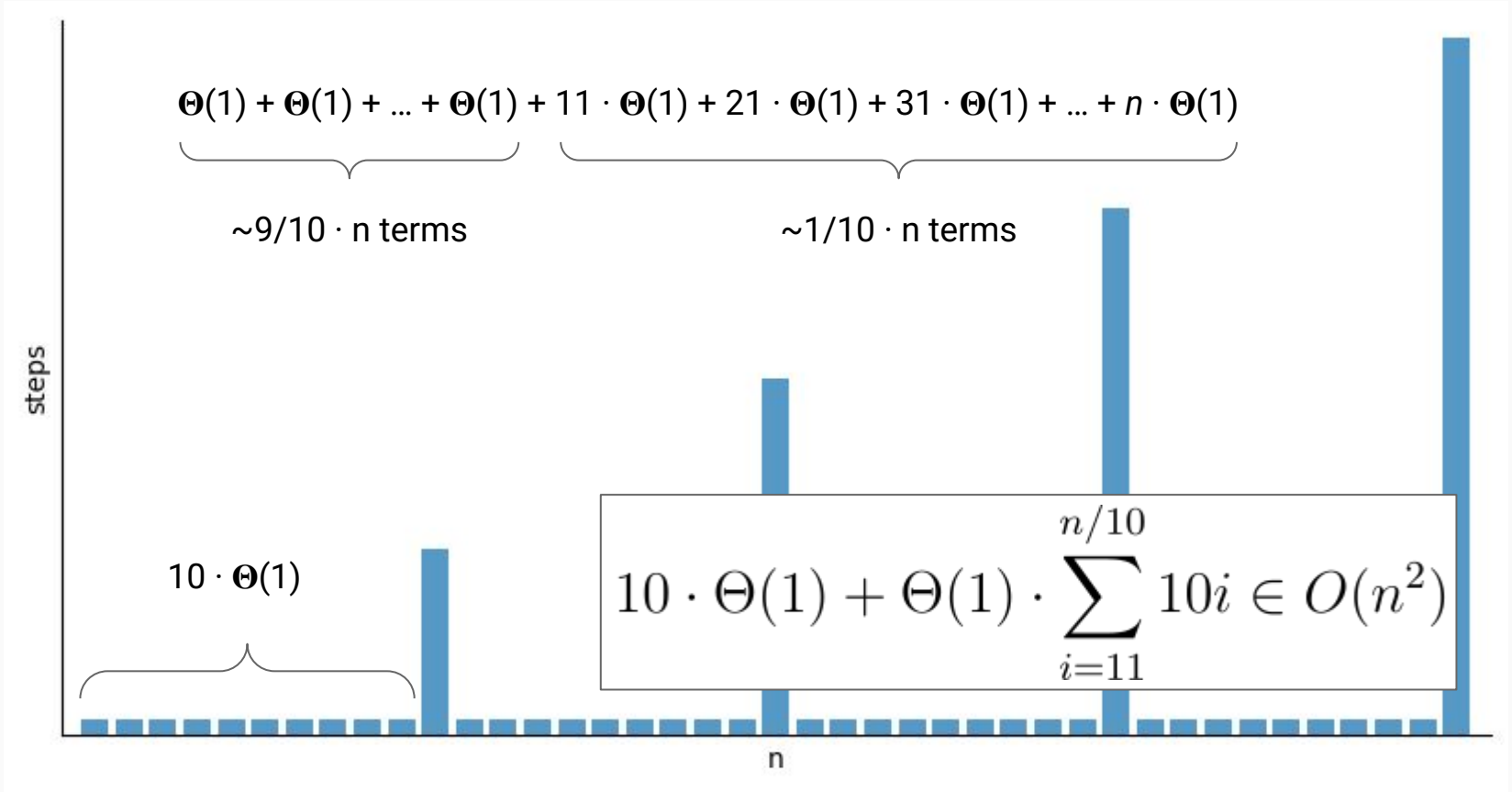
Then the **amortized runtime** of that function is **Θ(f(*n*)/*n*)**



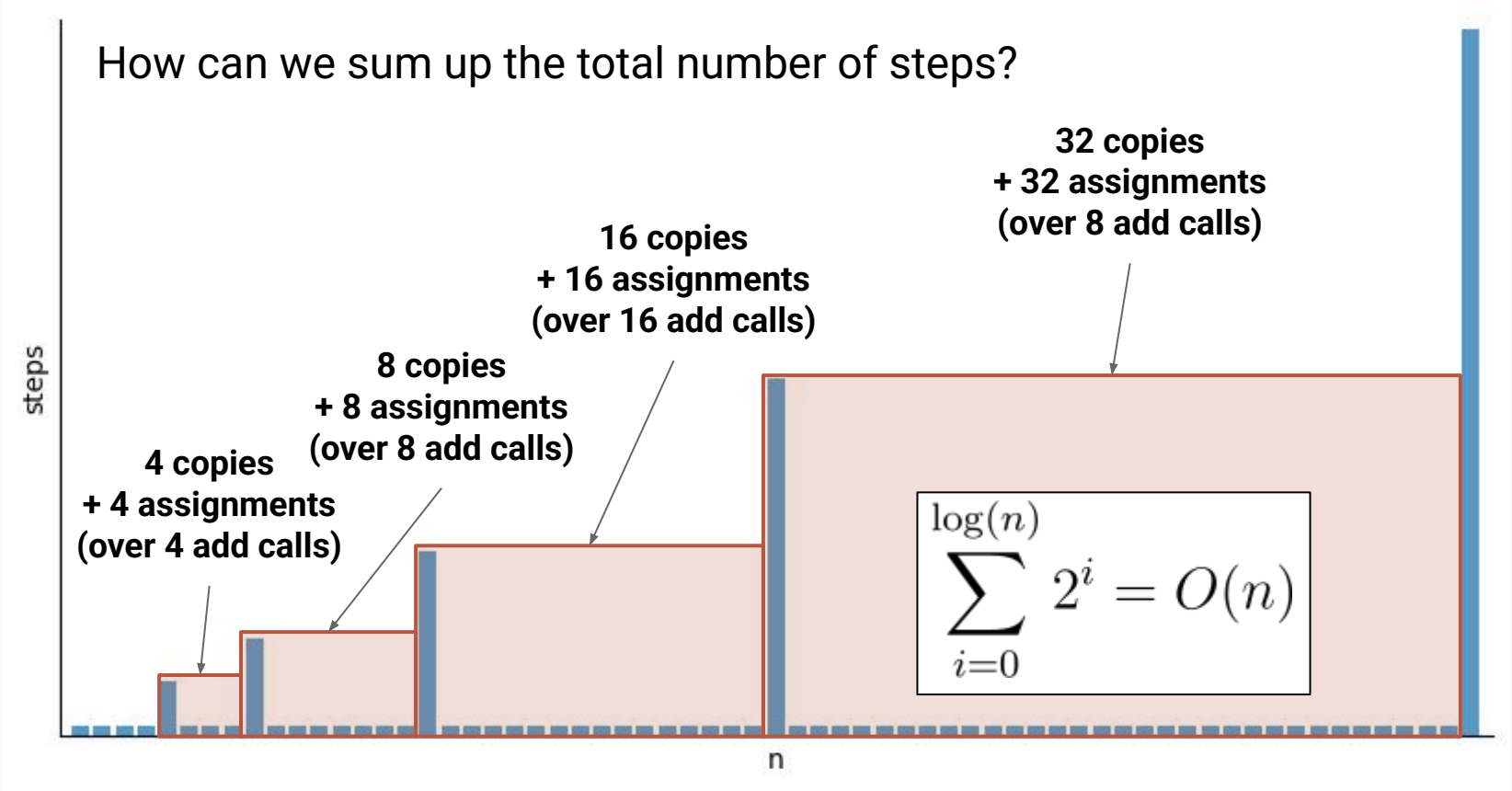
Cost of each call when the initial array size is 10, and newLength = data.length + 1



Cost of each call when the initial array size is 10, and newLength = data.length + 2



Cost of each call when the initial array size is 10, and newLength = data.length + 10



Cost of each call when the initial array size is 4, and newLength = data.length x 2

Amortized Runtime Analysis

```
1  public class Team {
2    private List<Player> players;
3
4    public void addPlayer(Player p) { /* ... */ }
5    public void importRoster(File f) { /* ... */ }
6    /* ... */
7  }
```

```
1  public void addPlayer(Player p) {
2    System.out.println("Welcome to the team " + p.name());
3    players.add(p);
4  }
```

Exercise: What are the unqualified and amortized runtime bounds of the **`addPlayer`** method when **`List`** is a **`LinkedList`**? an **`ArrayList`**?

Amortized Runtime Analysis

```
1  public class Team {
2    private List<Player> players;
3
4    public void addPlayer(Player p) { /*      */ }
5    public void importRoster(File
6    /* ... */
7  }
```

	players is LinkedList	players is ArrayList
addPlayer runtime	Unqualified $\Theta(1)$ Amortized $\Theta(1)$	Unqualified $O(n)$ Amortized $\Theta(1)$

```
1  public void addPlayer(Player p) {
2    System.out.println("Welcome to the team " + p.name());
3    players.add(p);
4  }
```

Exercise: What are the unqualified and amortized runtime bounds of the **`addPlayer`** method when **`List`** is a **`LinkedList`**? an **`ArrayList`**?

Amortized Runtime Analysis

```
1  public void importRoster(File f) {
2    BufferedReader br = new BufferedReader(new FileReader(f));
3    String line;
4    while (br.ready()) {
5      String line = br.readLine();
6      Player p = new Player(line);
7      addPlayer(p);
8    }
9  }
```

	`players` is `LinkedList`	`players` is `ArrayList`
`addPlayer` runtime	Unqualified $\Theta(1)$ Amortized $\Theta(1)$	Unqualified $O(n)$ Amortized $\Theta(1)$

Exercise: What are the unqualified and amortized runtime bounds of the **`importRoster`** method when **`List`** is a **`LinkedList`**? an **`ArrayList`**?

*(You can assume that opening the file, reading a line, and creating a **`Player`** are constant-time calls)*

Amortized Runtime Analysis

```
1  public void importRoster(File f) {
2    BufferedReader br = new BufferedReader(new FileReader(f));
3    String line;
4    while (br.ready()) {
5      String line = br.readLine();
6      Player p = new Player(line);
7      addPlayer(p);
8    }
9  }
```

	`players` is `LinkedList`	`players` is `ArrayList`
`addPlayer` runtime	Unqualified $\Theta(1)$ Amortized $\Theta(1)$	Unqualified $O(n)$ Amortized $\Theta(1)$
`importRoster` runtime	Unqualified $\Theta(n)$ Amortized $\Theta(n)$	Unqualified $\Theta(n)$ Amortized $\Theta(n)$

Exercise: What are
`importRoster` met

*(You can assume that opening the file, reading a line, and creating a **`Player`** are constant-time calls)*

Example Scenario

A company is developing two different applications and needs to choose the most appropriate data structure for each.

1. In the first application, the program frequently needs to retrieve elements at arbitrary positions (e.g., "get the 5000th element") as fast as possible, but insertions and deletions happen rarely.

2. In the second application, the program needs to support continuous appending of new records at the end of the list. The total number of records is unbounded, and the company wants to avoid the overhead of resizing operations as the list grows.

Discussion: What ADT and data structure would be best suited for each scenario?