CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Linked Lists from the JCF

Department of Computer Science
University of Maryland, College Park

What is LinkedList in JCF?

- **LinkedList<E>** is part of **Java Collections Framework (JCF)**. Implements two interfaces:
  - **List<E>** → Allows indexed access, insertion, and deletion.
  - **Deque<E>** → Supports **queue** and **stack** operations.

    https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedList.html
- **Underlying Structure:** Uses a **doubly linked list**, where each node contains **data, a reference to the next node, and a reference to the previous node**.
- No fixed size; dynamically grows and shrinks.
- **Key Advantage:** Efficient insertions/deletions in the middle. Still O(n) to reach node, but actual insertion is O(1)
- **Key Disadvantage:** Slower random access compared to ArrayList.

Constructing a LinkedList

- **Creating an Empty LinkedList**

```
LinkedList<String> list = new LinkedList<>();
```

- **Initializing from Another Collection**

```
List<String> words = Arrays.asList("A", "B", "C");
LinkedList<String> list = new LinkedList<>(words);
```

Why is this useful?

- Supports dynamic resizing.
- Can easily convert from other list types.
- Works well with generics (LinkedList<Integer>, LinkedList<Person>).

Common List<E> Methods in LinkedList

- **Adding Elements**
  - add(E element): Append to the end.
  - add(int index, E element): Insert at a specific index.
  - addAll(Collection<E> c): Append all elements from another collection.
- **Accessing Elements**
  - get(int index): Retrieves an element.
  - set(int index, E element): Replaces an element.
- **Removing Elements**
  - remove(int index): Removes at a specific position.
  - remove(Object o): Removes the first occurrence of an object.
  - clear(): Removes all elements.
- **Performance Note:**
  - add(int, E), remove(int) → **O(n)** (traversal required).
  - add(E), removeLast() → **O(1)** (direct link updates).

See: `LinkedListMethodsExample`

Deque<E> Methods in LinkedList

- Since LinkedList implements Deque<E>, it has **queue and stack** operations:
- **Queue Operations (FIFO - First In, First Out)**
  - offer(E e), poll(), peek()
  - offerFirst(E e), pollFirst(), peekFirst()
  - offerLast(E e), pollLast(), peekLast()
- **Stack Operations (LIFO - Last In, First Out)**
  - push(E e), pop(), peek()
- **Why Use These?**
  - offer() and poll() avoid exceptions (return null if empty).
  - push() and pop() mimic a stack structure.

See: `DequeMethodsExample`

Performance Comparison (LinkedList vs. ArrayList)

Operation	LinkedList	ArrayList
Add at End	O(1) (direct link update)	O(1) (amortized)
Add at Index	O(n) (traverse list)	O(n) (shift elements)
Remove at End	O(1)	O(1)
Remove at Index	O(n) (traverse list)	O(n) (shift elements)
Get by Index	O(n) (must traverse)	O(1) (direct access)

Best Use Cases:

- Use LinkedList **when frequent insertions/deletions** in the middle are needed.
- Use ArrayList **when frequent random access is needed**.

Introduction to Iterators (Iterator<E>)

- Used to **traverse a LinkedList efficiently** without using get(index).

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Iterator.html

- Avoids ConcurrentModificationException when removing elements. See: `ConcurrentModificationExample`
- **Key Methods:**
  - hasNext(): Checks if more elements exist.
  - next(): Moves to the next element.
  - remove(): Removes the last returned element.
- **Why Use Iterators?**
  - **Efficient traversal** (avoids repeated get(index)).
  - **Safer modifications** while looping.
  - **Works with any Collection<E> type.**

ListIterator<E> (Bidirectional Iterator)

- **Enhancement over Iterator<E>**
  Moves **forward & backward** in a list.
  Can **modify elements** during traversal.
  Only available for List<E> (not Set<E>).

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ListIterator.html

- **Key Methods:**
  - hasPrevious(), previous(): Allows reverse traversal.
  - set(E e): Modifies the last returned element.
  - add(E e): Inserts an element at the iterator's position.
  - remove(): Removes the last returned element.
- **When to Use?**
  - When **modifications** (set(), add(), remove()) are needed during traversal.
  - When **backward traversal** is necessary.

See: `ListIteratorExample`

Common Pitfalls with LinkedList and Iterators

- **Using get(index) in a loop**

```
for (int i = 0; i < list.size(); i++)
{ System.out.println(list.get(i)); // O(n) per access, very slow! }
```

- **Use an Iterator instead**

```
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
        System.out.println(iter.next()); // Efficient traversal }
```

- **Removing elements incorrectly**

```
for (String item : list) {
        if (item.equals("B")) {
                list.remove(item); // Causes ConcurrentModificationException! }
   }
```

- **Use Iterator.remove() instead**

```
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
        if (iter.next().equals("B"))
                { iter.remove(); // Safe removal } }
```

Summary & Best Practices

- LinkedList is **efficient** for insertions/deletions but **slow** for random access.
- Implements **List<E>** (like ArrayList) and **Deque<E>** (for stacks & queues).
- **Use Iterator or ListIterator** for efficient traversal and modification.
- Avoid **using get(index) in loops** to prevent performance issues.
- **When to Use LinkedList?**
  - When frequent **insertions & deletions in the middle** are required.
  - When **queue or stack operations** are needed.
  - When **iterating efficiently using an Iterator**.