LINKLIST

What is a LinkedList?

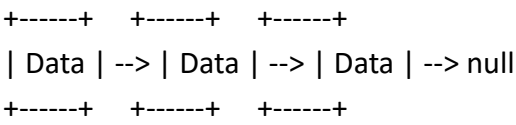
A **LinkedList** is a linear data structure where each element, called a **node**, contains a reference (or link) to the next (and/or previous) node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations. This allows for efficient insertions and deletions at any position in the list.

Structure of a Node

A typical node in a singly linked list contains:

- **Data**: The value stored in the node.
- **Next Reference**: A pointer or reference to the next node in the list.

Visualization of a Singly LinkedList:



Types of LinkedLists

There are several variations of linked lists, each serving different purposes:

1. Singly LinkedList

Each node contains data and a single reference to the next node. The last node's next reference is null.

Characteristics:

- Simple structure.
- Efficient insertion and deletion at the beginning.
- Traversal is only forward.

Example:

```
class Node {
   int data;
```

```
    Node next;
}
```

2. Doubly LinkedList

Each node contains data and two references: one to the next node and another to the previous node.

Characteristics:

- Allows bidirectional traversal.
- More memory consumption due to an extra reference.
- Efficient insertion and deletion from both ends.

Example:

```
class Node {
    int data;
    Node next;
    Node prev;
}
```

3. Circular LinkedList

The last node's next reference points back to the head of the list, forming a circle. This can be applied to both singly and doubly linked lists.

Characteristics:

- Useful for applications that require continuous looping over the list.
- No null references in the list.

Common Operations on LinkedList

Let's explore some common operations provided by Java's **LinkedList** class.

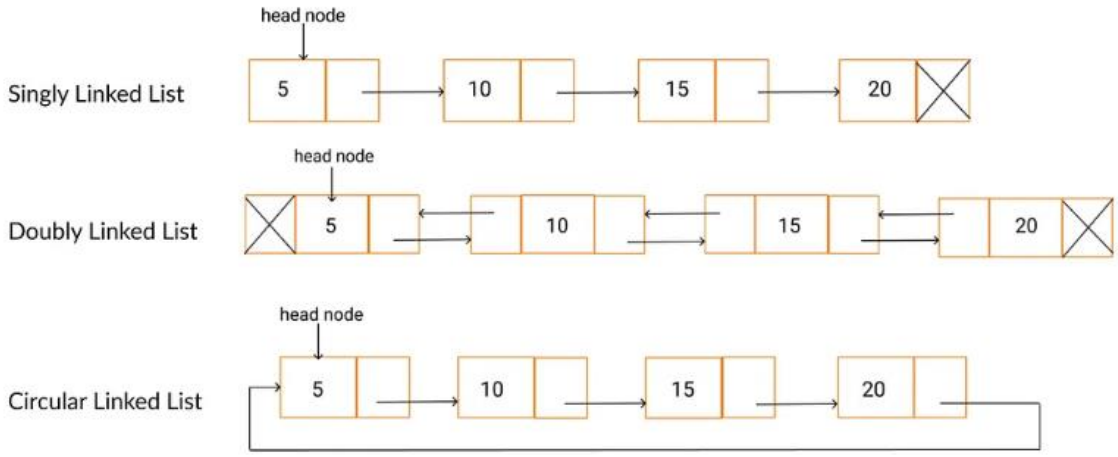
1. Adding Elements

- **add(E e)**: Appends the specified element to the end.
- **add(int index, E element)**: Inserts the element at the specified position.
- **addFirst(E e)**: Inserts the element at the beginning.
- **addLast(E e)**: Inserts the element at the end.

```
LinkedList<Integer> numbers = new LinkedList<>();
numbers.add(10);
numbers.addFirst(5);
numbers.addLast(15);
numbers.add(1, 7); // Inserts 7 at index 1
```

Resulting List: [5, 7, 10, 15]

DIAGRAM OF LINKLIST



2. Removing Elements

- **remove()**: Removes and returns the first element.
- **remove(int index)**: Removes the element at the specified position.
- **removeFirst()**: Removes and returns the first element.
- **removeLast()**: Removes and returns the last element.
- **clear()**: Removes all elements from the list.

Example

```
numbers.remove();      // Removes 5
numbers.removeLast();  // Removes 15
numbers.remove(0);     // Removes 7
```

Resulting List: [10]

3. Accessing Elements

- **get(int index)**: Returns the element at the specified position.
- **getFirst()**: Returns the first element.
- **getLast()**: Returns the last element.

```
int first = numbers.getFirst(); // 10
int last = numbers.getLast();   // 10
```

4. Iterating Through the List

Using Enhanced for-loop:

```
for (int num : numbers) {
   System.out.println(num);
}
```

Using Iterator:

```
Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
   System.out.println(iterator.next());
}
```

5. Checking for Elements

- **contains(Object o)**: Returns true if the list contains the specified element.
- **indexOf(Object o)**: Returns the index of the first occurrence, or -1 if not found.
- **lastIndexOf(Object o)**: Returns the index of the last occurrence, or -1 if not found.

Example:

```
boolean hasTen = numbers.contains(10); // true
int index = numbers.indexOf(10);       // 0
```

Use Cases of LinkedList

LinkedLists are particularly useful in scenarios where:

1. **Frequent Insertions and Deletions:** When your application requires frequent additions and removals from the middle or beginning of the list.

2.Implementing Queues and Stacks: LinkedLists can efficiently implement queue (FIFO) and stack (LIFO) data structures.

3. **Dynamic Data Size:** When the size of the data is unpredictable, and you want to avoid the overhead of resizing arrays.

4. **Graph and Tree Implementations:** Used extensively in implementing graphs and trees where dynamic connections between nodes are necessary.

5. Undo/Redo Functionality: Storing states that can be navigated forwards and backwards efficiently.

Pros and Cons of LinkedList

Pros

- **Dynamic Size:** Easily grows and shrinks during runtime by allocating and deallocating memory as needed.
- **Efficient Insertions/Deletions:** Especially at the beginning and middle without shifting elements.
- **Flexibility:** Can easily implement other data structures like stacks and queues.

Cons

- **Memory Overhead:** Additional memory is required for storing references/pointers.
- **Sequential Access:** Access time is linear (O(n)), making it inefficient for random access compared to arrays.
- **Cache Performance:** Less cache-friendly due to non-contiguous memory allocation.
- **Extra Garbage Collection:** Frequent allocation and deallocation can lead to more work for the garbage collector.

Here are the key differences between ArrayList and LinkedList in Java (IMP*)

1. Internal Structure:

- **ArrayList:**

Uses a dynamic array to store elements. This means elements are stored in contiguous memory locations.

- **LinkedList:**

Uses a doubly linked list structure. Each element is a node containing the data and references (pointers) to the next and previous nodes.

2. Access Time:

- **ArrayList:**

Provides faster random access to elements using index-based access (e.g., get(index)) because of direct array indexing. This operation has a time complexity of O(1).

- **LinkedList:**

Slower for random access because it needs to traverse the list from the beginning or end until the specified index is reached. This operation has a time complexity of O(n).

3. Insertion and Deletion:

- **ArrayList:**

Inefficient for insertions and deletions in the middle of the list. This is because elements after the insertion/deletion point need to be shifted to maintain the contiguous memory structure. This operation has a time complexity of O(n).

- **LinkedList:**

More efficient for insertions and deletions at any position. It only needs to update the references of the surrounding nodes, avoiding the need to shift elements. This operation has a time complexity of O(1).

4. Memory Usage:

- **ArrayList:**

Generally has slightly less memory overhead due to its simple array structure.

- **LinkedList:**

Has more memory overhead because of the additional node references (pointers to the next and previous nodes).

5. Iteration:

- **ArrayList:**

Faster for sequential iteration because it can access elements directly in memory.

- **LinkedList:**

Slower for sequential iteration because it needs to follow node references.

6. Implementation:

- **ArrayList:** Straightforward implementation using dynamic arrays.
- **LinkedList:** Involves more complex pointer manipulation to manage the linked list structure.

7. Dynamic Size:

- Both ArrayList and LinkedList are dynamic data structures, meaning their size can grow or shrink as needed. ArrayList uses dynamic arrays to achieve this, while LinkedList uses its linked nature.

8. Default Size:

- ArrayList has a default capacity of 10, which can be increased as needed.
- LinkedList does not have a default capacity.

9. When to use which:

- **ArrayList:**

Use when you need fast random access and do not frequently insert or remove elements in the middle of the list. It is also preferred for sequential iteration and when memory usage is a concern.

- **LinkedList:**

Use when you need frequent insertions and deletions, especially at the beginning or middle of the list, and random access is not a primary concern.