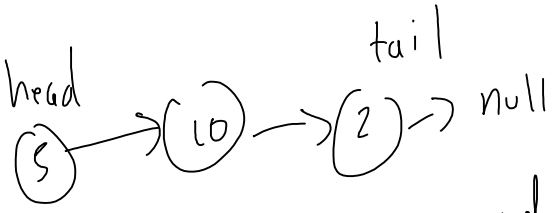
7. Linked Lists

Thursday, May 12, 2022 7:53 AM

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

https://visualgo.net/en/list?slide=1

Nodes
- consists of 2 things:
  - value of data
  - pointer to next node



head is start, tail is end

Why linked list:
- Don't have to shift every element when inserting elements
- while loop to traverse Linked List
  - because linked list doesn't have a fixed size, you just have to hit null
- Delete nodes is easier vs array
- L.L. has sequential order, which can be sorted unlike hashmaps

* prepend	O(1)	- beginning of L.L.
append	O(1)	- end of L.L.
lookup	O(n)	- traversal
insert	O(n)	- only O(n) at worst case
delete	O(n)	

What is a pointer

- reference to a place in memory / object / node

e.g.

```
int number = 5;
int pointer = number;
```

Java has automatic garbage collection, so you don't have to delete value + pointer if unused

Creating a Linked List

- 3 classes

Node
- int data
- Nuode next

LinkedList
- append
- prepend
- insert
- delete

Main
{ }

head -> null , head = null

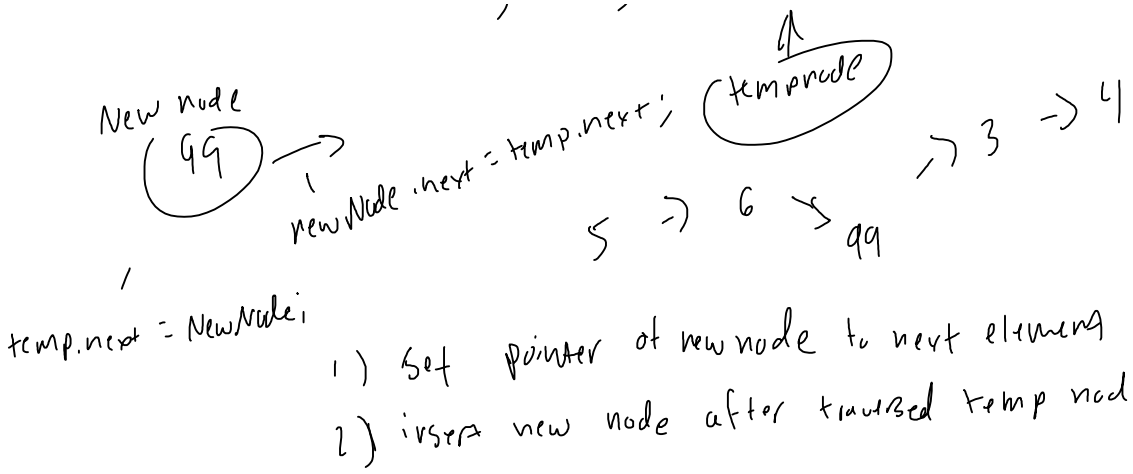
head -> 5 -> 6 -> null

temp node = head (for traverse list) ↗ temp.next != null

insert At(1) 0 1 fail

head 5 -> 6 -> 3 -> 4 -> null

↑ (temp node)



Singly Linked List

```
public class Node {

   int data;
   Node next; // Refers to the next node

   // Constructor
   Node(int d) {
      data = d;
      next = null;
   }

}
```

```
class Main {
   public static void main(String[] args) {
      LinkedList newList = new LinkedList();

      newList.append(3);
      newList.append(4);
      newList.prepend(2);
      newList.prepend(1);
      newList.insert(3,61);
      newList.show();

      System.out.println("----------");
      newList.delete(2);


      newList.show();

   }
}
```

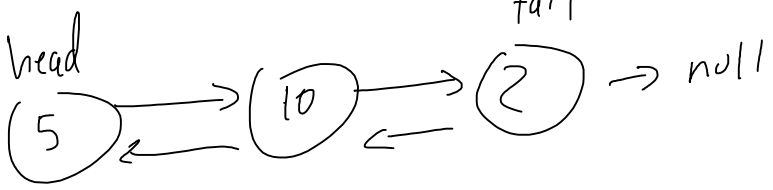
```
public class LinkedList {
   Node head; // refers to first node
   int length = 1;

   // Insert node with input data at end of Linked List
   public Node append(int data) {

      // Create new node with given data
      Node newNode = new Node(data);

      // Check if first node or not, if empty then new node is head
      if (head == null)
      {
         head = newNode;
      }
      else {
         // Traverse to last node then append the new node to the end
         // Temp node refers to head node, then checks the next
         // nodes until it hits last
         Node temp = head;
         while (temp.next != null) {
            temp = temp.next;
         }
         temp.next = newNode;
```

```
        }
        length++;
        return head;
    }

    // Insert node with input data at start of Linked List
    public Node prepend(int data) {

        Node newNode = new Node(data);


        // Let the new Node point to the head
        newNode.next = head;

        // Make new node the head
        head = newNode;

        length++;
        return head;

    }

    // Insert node at position

    public Node insert(int index, int data){

        Node newNode = new Node(data);

        // Edge case checks
        if (index <= 0) {
            prepend(data);
        }
        else if (index >= length) {
            append(data);
        }
        else {
            Node temp = head;
            // Traverse to index where node is to be inserted
            for (int i = 0; i < index - 1; i++) {
                temp = temp.next;
            }

            // new node now points to the next node that temp node
traversed
            newNode.next = temp.next;
            // insert new node after traversed temp node
            temp.next = newNode;


        }

        length++;
        return head;
    }

    public Node delete(int index){
        if (index < 0 ){
            index = 0;
        }
        if (index == 0) {
            head = head.next;
        }


        Node temp = head;
        for (int i = 0 ; i < index -1; i++) {
            temp = temp.next;
```

```
    }
    /* for garbage collection
    Node delete = null;
    delete = temp.next;
    temp.next = delete.next;

     */
    temp.next = temp.next.next;
    length--;
    return head;
  }


  // Prints whole list
  public void show(){
    // Temp node to traverse list
    Node temp = head;

    while (temp.next != null)
    {
      System.out.println(temp.data);
      temp = temp.next;
    }
    // Prints last element
    System.out.println(temp.data);
    System.out.println("length: " + length);
  }


}
```

Doubly Linked List

- Links to node before it with a prev pointer



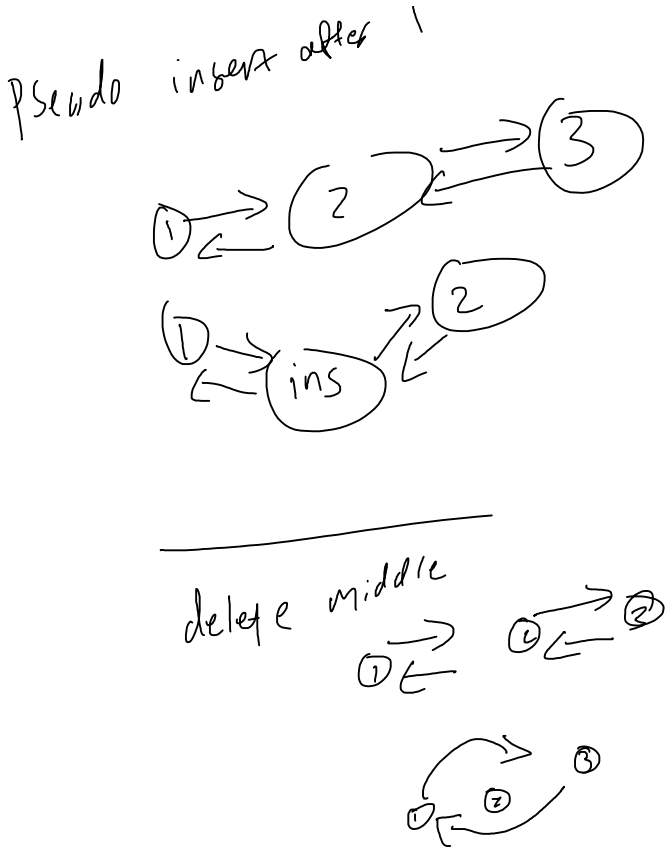
~~*~~ prepend O(1)

append O(1)

lookup O(n/2) → O(n) , traversal

insert O(n)

delete O(n)




```
public class LinkedList {
   Node head; // refers to first node
   int length = 1;

   // Insert node with input data at end of Linked List
   public Node append(int data) {

      // Create new node with given data
      Node newNode = new Node(data);

      // Check if first node or not, if empty then new node is head
      if (head == null)
      {
         head = newNode;
         newNode.prev = null;
         return head;

      }
      else {
         // Traverse to last node then append the new node to the end
         // Temp node refers to head node, then checks the next
         // nodes until it hits last
         Node temp = head;
         while (temp.next != null) {
            temp = temp.next;
         }
         temp.next = newNode; // append newNode to end of list
         newNode.prev = temp; // newNode prev pointer set to node
before
      }
      length++;
      return head;
   }
```

```
public class Node {

   int data;
   Node next; // Refers to the next node
   Node prev; // Refers to prev node

   // Constructor
   Node(int d) {
      data = d;

   }

}
```

```
class Main {
   public static void main(String[] args) {
      LinkedList newList = new LinkedList();

      newList.append(3);
      newList.append(4);
     newList.prepend(2);
      newList.prepend(1);
      newList.insert(2,61);
     newList.show();

      System.out.println("----------");
      newList.delete(2);
```

```
    }
    length++;
    return head;
  }

  // Insert node with input data at start of Linked List
  public Node prepend(int data) {

    Node newNode = new Node(data);


    // Let the new Node point to the head and new node point to null
    newNode.next = head;
    newNode.prev = null;

    // current head prev pointer points to new head
    if (head != null) {
      head.prev = newNode;
    }

    // make newNode the head
    head = newNode;

    length++;
    return head;

  }

  // Insert node at position

  public Node insert(int index, int data){

    Node newNode = new Node(data);

    // Edge case checks
    if (index <= 0) {
      prepend(data);
    }
    else if (index >= length) {
      append(data);
    }
    else {
      Node temp = head;
      // Traverse to index where node is to be inserted
      for (int i = 0; i < index - 1; i++) {
        temp = temp.next;
      }

      // new node now points to the next node that temp node
traversed
      newNode.next = temp.next;
      // insert new node after traversed temp node
      temp.next = newNode;

      // newNode prev pointer points to traversed temp
      newNode.prev = temp;

      // prev pointer of node after inserted new node points to this
new node
      newNode.next.prev = newNode;


    }

    length++;
    return head;
  }
```

```
      newList.show();

    }
  }
```

```
Node@4554617c, data: 1, next:Node@74a14482, prev:null
Node@74a14482, data: 2, next:Node@1540e19d, prev:Node@
4554617c
Node@1540e19d, data: 61, next:Node@677327b6,
prev:Node@74a14482
Node@677327b6, data: 3, next:Node@14ae5a5, prev:Node@
1540e19d
Node@14ae5a5, data: 4, next:null, prev:Node@677327b6
length: 5
----------
Node@4554617c, data: 1, next:Node@74a14482, prev:null
Node@74a14482, data: 2, next:Node@677327b6, prev:Node@
4554617c
Node@677327b6, data: 3, next:Node@14ae5a5, prev:Node@
74a14482
Node@14ae5a5, data: 4, next:null, prev:Node@677327b6
length: 4

Process finished with exit code 0
```

```
  public Node delete(int index){
    if (index < 0 ){
      index = 0;
    }
    if (index == 0) {
      head = head.next;
    }


    Node temp = head;
    for (int i = 0 ; i < index -1; i++) {
      temp = temp.next;
    }
    /* for garbage collection
    Node delete = null;
    delete = temp.next;
    temp.next = delete.next;

     */

    // Deletes node by having current pointer point to node after node
you want to delete
    temp.next = temp.next.next;
    // Prev pointer of the node after your deleted node points to the
node before deleted
    temp.next.prev = temp;
    length--;
    return head;
  }



  // Prints whole list
  public void show(){
    // Temp node to traverse list
    Node temp = head;

    while (temp.next != null)
    {
      System.out.println( temp + ", data: " + temp.data + ", next:" +
temp.next + ", prev:" + temp.prev);
      temp = temp.next;
    }
    // Prints last element
    System.out.println( temp + ", data: " + temp.data + ", next:" +
temp.next + ", prev:" + temp.prev);
    System.out.println("length: " + length);
  }


}
```

Singly

- less memory
- slightly faster

Don't have to

VS

Doubly Linked List

- traversal from front & back
- deleting a prev node is faster
- requires more memory & storage

- Slightly faster (b/c we don't have to update pointers)
- good for fast insertion & deletion and you don't need much searching

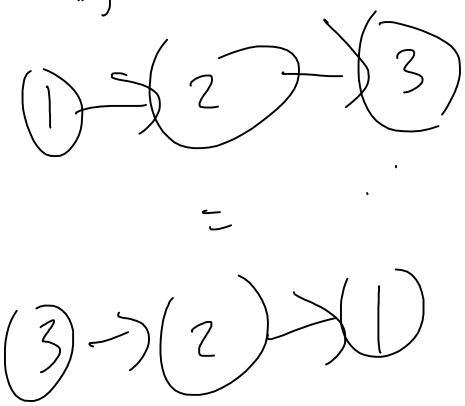
- requires more memory & storage
- good for searching - forwards & back

Reverse a linked list Practice

input: head of a singly linked list

output: return the reversed list

e.g.

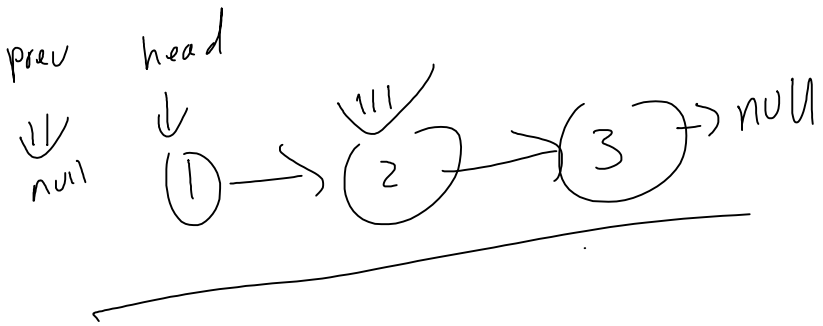


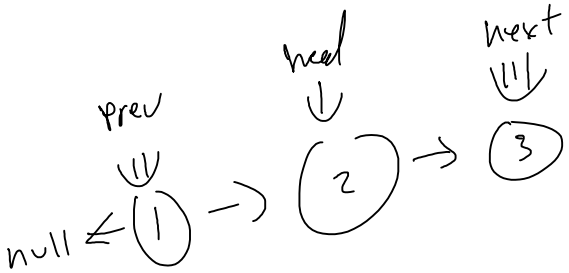
Approach:

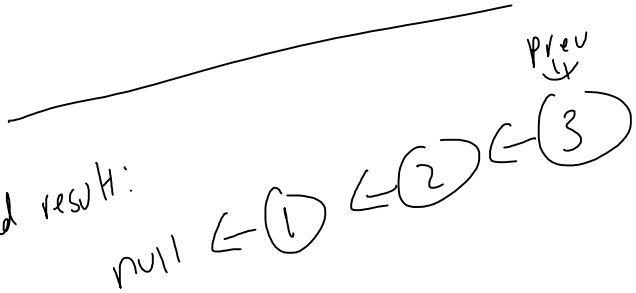
Edge case: head == null
return head

... approach ... prev & head & next pointer

3 pointer ...







End result:

Node prev = null;

(head != null) {

① Node nextNode = head.next

② head -> prev

③ prev = head

④ head = next

}

return prev

```
class Solution {
  public ListNode reverseList(ListNode head) {
    // Edge case check
    if (head == null) {
      return head;
    }

    // 3 Pointer Approach
    // Initialize outside because we return prev and it points the
reverse
    ListNode prev = null;
    while (head != null) {
      ListNode nextNode = head.next; // Initialize inside because it's a
temp placeholder
      head.next = prev;
      prev = head;
      head = nextNode;
    }
    return prev;
  }
  // O(n) time, because while loop depends on N size of the
LinkedList.size()
```

- reverse, current head -> prev
- advance nodes

} big O

```
        return prev;
    }
    // O(n) time, because while loop depends on N size of the
LinkedList.size()
    // O(1) space, because space created was constant and not
dependent on anything
}
```