Linked Lists

- A dynamically resizable, efficient implementation of a list

Tip For Success: Reminder

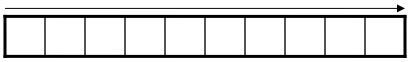
- Look through the examples and notes before class.
- This is especially important for this section because the execution of this programs will not be in sequential order.
- Instead execution will appear to 'jump around' so it will be harder to understand the concepts and follow the examples illustrating those concepts if you don't do a little preparatory work.
- Also the program code is more complex than most other examples.
- For these reasons tracing the code in this section is more challenging

James Tam

Lists

- Data that only includes one attribute or dimension
- Example data with one-dimension
  - –Tracking grades for a class
  - –Each cell contains the grade for a student i.e., `grades[i]`
  - – There is one dimension that specifies which student's grades are being accessed

One dimension (which student)



James Tam

Array Implementation Of A List: Advantage

- Ease of use: arrays are a simple structure

James Tam

Array Implementation Of A List: Disadvantage (Waste)

- Some array implementations cannot be automatically resized
  - E.g.,

```
int [] array = new int[10];
```

- Adding more elements requires the creation of a new array and the copying of existing data into the new array
  - E.g.

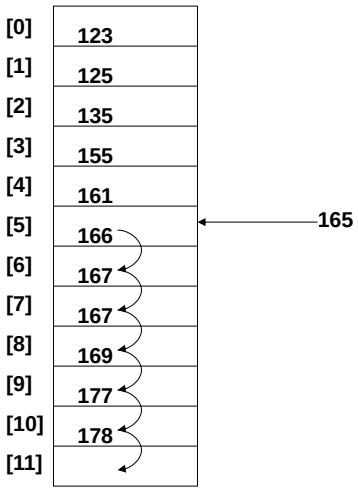
```
int [] bigger = new int[20];
int i; = 0;
while (i < array.length)
{
    bigger[i] = array[i];
    i++;
}
```

- That means that the array must be made larger than is typically needed.

James Tam

Array Implementation Of A List: Disadvantage (Inefficient)

- Inserting new elements to an ordered lists can be inefficient: requires 'shifting' of elements

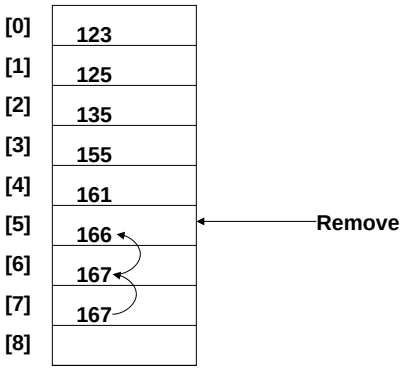


JT: If the size of each element is large (e.g., array of objects and not an array of references) then program speed can be degraded

James Tam

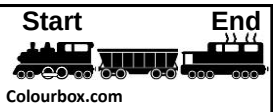
Array Implementation Of A List: Disadvantage (Inefficient)

- Similarly removing elements from an ordered lists can be inefficient: requires 'shifting' of elements



James Tam

Linked Lists



- An alternate implementation of a list.
  - As the name implies, unlike an array the linked list has explicit connections between elements
  - This connection is **the only thing** that holds the list together.
    - Removing a connection to an element makes the element inaccessible.
    - Adding a connection to an element makes the element a part of the list.
- The program code is more complex but some operations are more efficient (e.g., additions and deletions don't require shifting of elements).
  - Just change some connections.
- Also linked lists tend to be more memory efficient that arrays.
  - Again: the typical approach with an array is to make the array larger than needed. (Unused elements wastes space in memory).
  - With a linked list implementation, elements only take up space in memory as they're needed.

James Tam

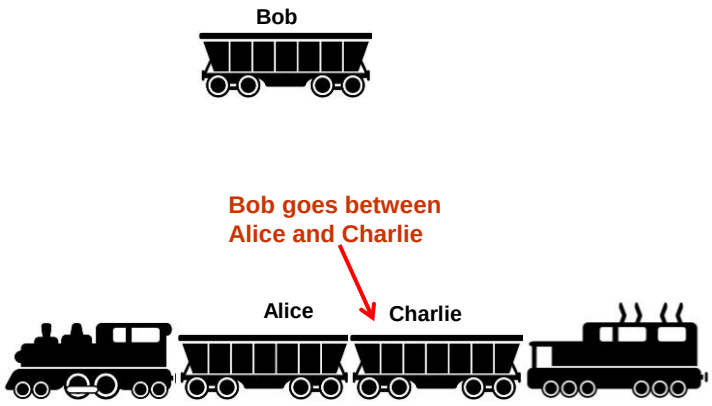
Linked Lists

- Insertions and removal of elements can be faster and more efficient because no shifting is required.
- Elements need only be linked into the proper place (insertions) or bypassed (deletions)

James Tam

Insertion

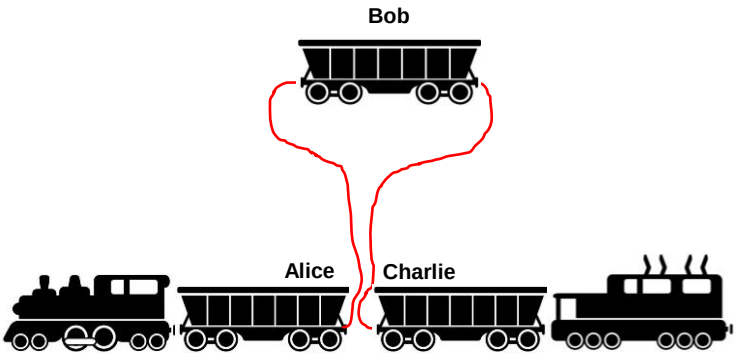
- Find the insertion point



James Tam

Insertion (2)

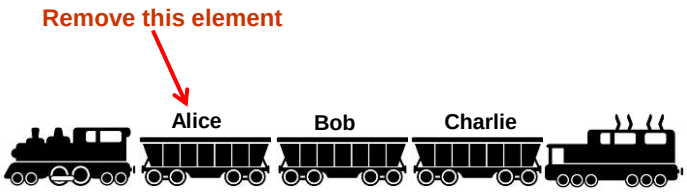
- Change the connections between list elements so the new element is inserted at the appropriate place in the list.



James Tam

Deletions

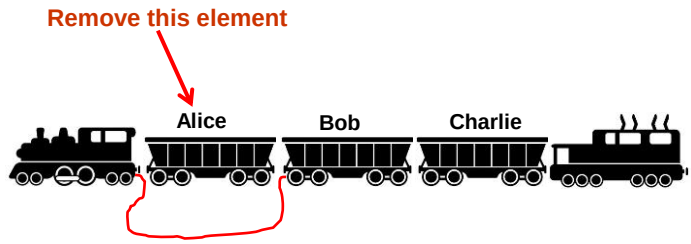
- Find location of the element to be deleted



James Tam

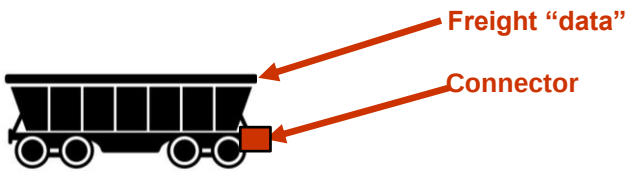
Deletions (2)

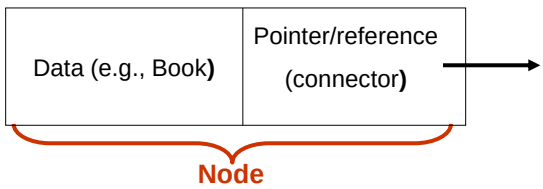
- Change the connections so that the element to be deleted is no longer a part of the list (by-passed).



James Tam

List Elements: Nodes

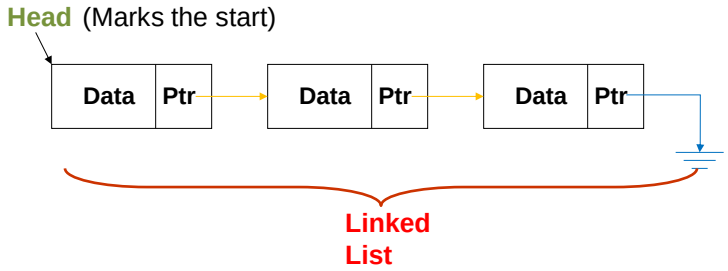




James Tam

Linked Lists: Important Details

- Unlike arrays, many details must be manually and explicitly specified by the programmer: **start of the list**, **connections between elements**, **end of the list**.



- Caution! Take care to ensure the reference to the first element is never lost.
  - Otherwise the entire list is lost

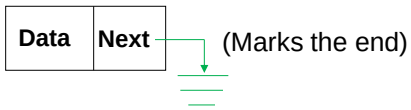
1 The approximate equivalent of a pointer ("ptr") in Java is a reference.

James Tam

More On Connections: **The Next Pointer**

- A special marker is needed for the end of the list.
- The '`next`' attribute of a node will either:
  1. Contain a **reference/address** of the next node in the list.

  2. Contain a `null` value.



- (That means once there is a reference to the start of the list (copy of the "head"), the next pointer of each element can be used to traverse the list).

James Tam

Location Of The Full Example

- Due to the complexity of this program it will be decomposed into sections :
  - List operation e.g., adding elements, removing elements
- The sections may have sub-sections
  - Sub-cases of list operations e.g., removing first element, removing any element except for the first etc.
- Full example:
  - `/home/233/examples/linkedLists`

James Tam

Outline Of The Example

Book

- The 'freight', the data stored in each list element
- In the example books only have a title attribute

Colourbox.com

BookNode

- The 'train cars'
- In the example a node has two attributes:
  - A book (data/'freight')
  - **Next reference**

Manager

- Implements all the list operations: insertions, removals, display of elements etc.

Colourbox.com

Driver

James Tam

Defining The Data: A Book

```
public class Book
{
    private String name;
    public Book(String aName) { ... }
    public String getName() { ... }
    public void setName(String aName) { ... }
    public String toString() { ... }
}
```

James Tam

Example: Defining A Node

```
public class BookNode {
    private Book data;
    private BookNode next;
}
```

Information stored by each element

Connects list elements = null or address of next element

James Tam

Class BookNode

```
public class BookNode
{
    private Book data;
    private BookNode next;

    public BookNode()
    {
        setData(null);
        setNext(null);
    }

    public BookNode(Book someData, BookNode nextNode)
    {
        setData(someData);
        setNext(nextNode);
    }
```

James Tam

Class BookNode (2)

```
    public Book getData() {  return(data);  }

    public BookNode getNext() { return(next);  }

    public void setData(Book someData) {  data = someData;  }

    public void setNext(BookNode nextNode) { next = nextNode; }

    public String toString() {
        return(data.toString());
    }
}
```

```
// Book.toString()
public String toString()
{
    String temp;
    if (name != null)
        temp = "Book name: " + name;
    else
        temp = "Book name: No-name";
    return(temp);
}
```

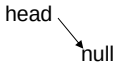
James Tam

Creating A New Manager (And New List)

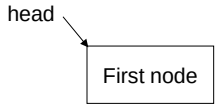
```
public class Driver
{
    public static void main (String [] args)
    {
        Manager aManager = new Manager();  // New manager
        ...
```

```
public class Manager
{
    private BookNode head;   // Recall: marks start of list
    public Manager ()  {
        head = null;  // New (empty) list
    }
```

Case 1: Empty list



Case 2: Non-empty list



James Tam

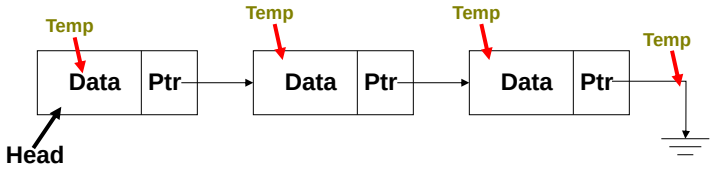
A More Detailed Outline Of Class Manager

```
public class Manager {

    public void add() {
        // Add new node to end of the list
    }

    public void display() {
        // Iterative: in-order display
    }

    public void displayRecursive() {
        // Recursive: in-order display
    }

     public void eraseList() { ... }

     public void remove() {
         // Search and remove node
     }
```

James Tam

List Operations: Linked Lists (Display)

- A temporary pointer/reference is used when successively displaying the elements of the list.
- When the temporary pointer is null, the end of the list has been reached.
- Graphical illustration of the algorithm:



- Pseudo code algorithm:

```
while (temp != null)
    display node
    temp = address of next node
```

James Tam

First List Operation: Display

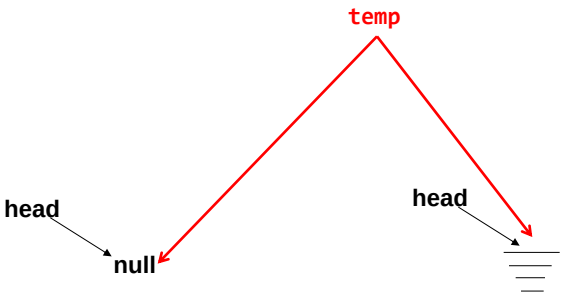
- **Case 1**: Empty List

```
// Driver
Manager listManager = new Manager();
listManager.display();

// Manager
public void display()
{
    int count = 1;
    BookNode temp = head;
    System.out.println(LIST_HEADER);
    for (int i = 0; i < LIST_HEADER.length(); i++)
        System.out.print("-");
    System.out.println();
    if (temp == null)
        System.out.println("\tList is empty: nothing to display");
```

```
private String LIST_HEADER =
"DISPLAYING LIST";
```

James Tam

Displaying The List: Iterative Implementation (Empty)



First List Operation: Display (2)

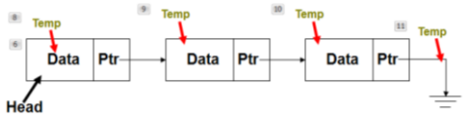
- **Case 2**: Non-empty list

```
// Driver
listManager.add();
listManager.add();
listManager.add();
listManager.display();
```

James Tam

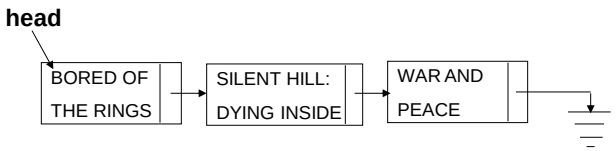
Manager.Display()

```
public void display()
{
    int count = 1;
    BookNode temp = head;
    System.out.println("LIST_HEADER");
    for (int i = 0; i < LIST_HEADER.length(); i++)
        System.out.print("-");
    System.out.println();
    if (temp == null)
        System.out.println("\tList is empty: nothing to display");
    while (temp != null)
    {
        System.out.println("\t#" + count + ": "+temp); //temp.toString()
        temp = temp.getNext();
        count = count + 1;
    }
    System.out.println();
}
```



James Tam

Displaying The List: Iterative Implementation (Non-Empty)



Traversing The List: Display

- **Study guide:**
- Steps (traversing the list to *display* the data portion of each node onscreen)
  1. Start by initializing a temporary reference to the beginning of the list.
  2. If the reference is '`null`' then display a message onscreen indicating that there are no nodes to display and stop otherwise proceed to next step.
  3. While the temporary reference is not `null`:
     a) Process the node (e.g., display the data onscreen).
     b) Move to the next node by following the current node's next reference (set the temp reference to refer to the next node).

Second List Operation: Destroying List

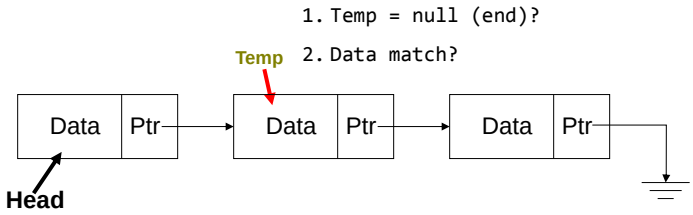
```
public void eraseList ()
{
    head = null;
}
```

- Caution! This works in Java because of automatic garbage collection.
- Be aware that you would have to manually free up the memory for each node prior to this step with other languages.

James Tam

List Operations: Linked Lists (Search)

- The algorithm is similar to displaying list elements except that there must be an additional check to see if a match has occurred.
- Conditions that may stop the search:



James Tam

List Operations: Linked Lists (Search: 2)

- Pseudo code algorithm:

```
Temp refers to beginning of the list
If (temp is referring to an empty list)
    display error message "Empty list cannot be searched"
While (not end of list AND match not found)
    if (match found)
        stop search or do something with the match
    else
        temp refers to next element
```

James Tam

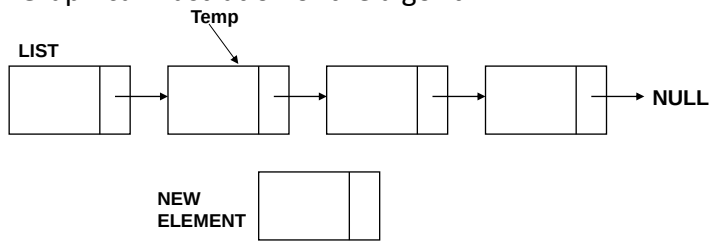
List Operations That Change List Membership

- These two operations (add/remove) change the number of elements in a list.
- The first step is to find the point in the list where the node is to be added or deleted (typically requires a search even if the search is just for the end of the list).
- Once that point in the list has been found, changing list membership is merely a reassignment of pointers/references.
  - Again: unlike the case with arrays, no shifting is needed.

James Tam

List Operations: Linked Lists (Insertion)

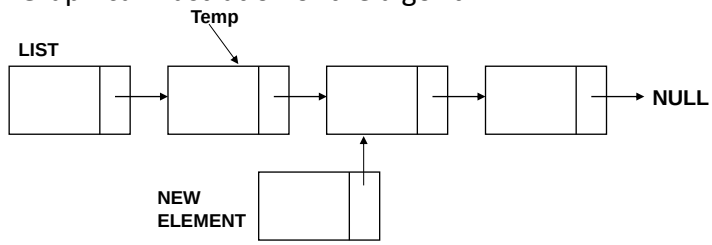
- Graphical illustration of the algorithm:



James Tam

List Operations: Linked Lists (Insertion: 2)

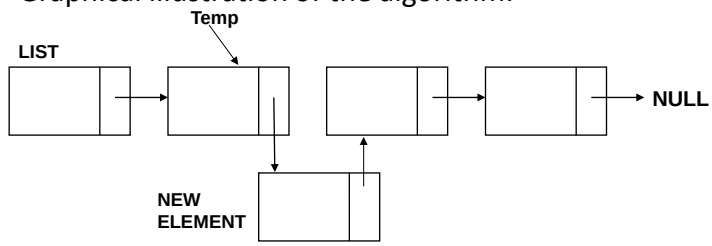
- Graphical illustration of the algorithm:



James Tam

List Operations: Linked Lists (Insertion: 3)

- Graphical illustration of the algorithm:



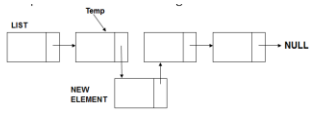
James Tam

List Operations: Linked Lists (Insertion: 4)

- Pseudo code algorithm (requires a search to be performed to find the insertion point even if the insertion occurs at the end of the list).

- # Search: use two references (one eventually points to the match
- # while the other points to the node immediately prior).

```
While (not end of list AND match not found)
    if (match found)
        stop search or do something with the match
    else
        temp refers to next element
```



- # Insert

Node to be inserted refers to node after insertion point
Node at insertion point refers to the node to be inserted

James Tam

Third List Operation: Add/Insert At End

```
// Driver
listManager.add();  // Empty list at this point
listManager.add();
listManager.add();
```

James Tam

Manager.Add()

```
public void add()
{
    String title;
    Book newBook;
    BookNode newNode;

    System.out.println("Adding a new book");
    System.out.print("\tBook title: ");
    title = in.nextLine();  // Get title from user
    newBook = new Book(title);
    newNode = new BookNode(newBook,null);

    // Case 1: List empty: new node becomes first node
    if (head == null)
    {
        head = newNode;
    }
```

James Tam

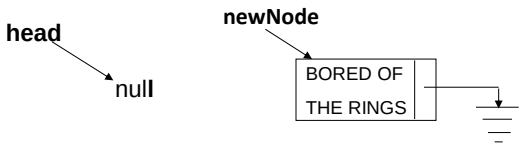
Manager.Add() : 2

```
    // Case 2: Node not empty, find insertion point (end of list)
    else
    {
        BookNode current = head;
        BookNode previous = null;
        while (current != null)
        {
            previous = current;
            current = current.getNext();
        }
        previous.setNext(newNode);
        // Adds node to end: since a node's next field is already
        // set to null at creation nothing else need be done.
    }
}
```

James Tam

Adding A Node To The End Of The List: Empty List

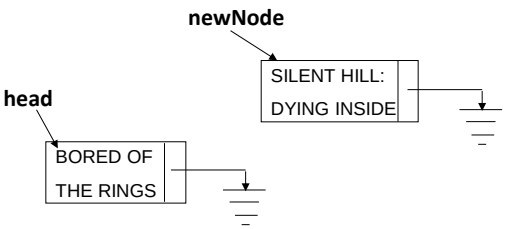
Adding first node to empty list



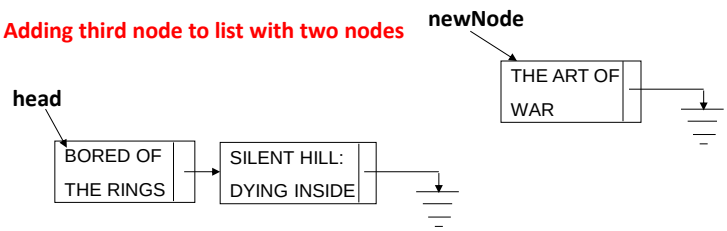
James Tam

Adding A Node To The End Of The List: Non-Empty List

Adding second node to list with one node



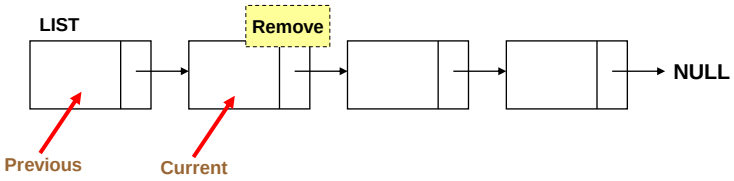
Adding third node to list with two nodes



James Tam

List Operations: Linked Lists (Removing Elements)

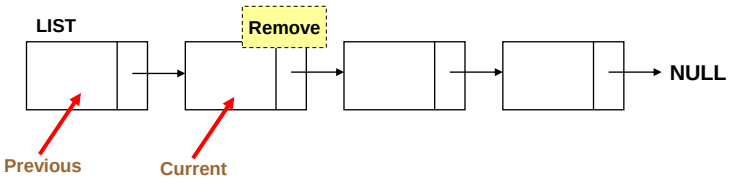
- Graphical illustration of the algorithm
- (Note that the search algorithm must first be used to find the location of the node to be removed)
  - Current: marks the node to be removed
  - Previous: marks the node prior to the node to be removed



James Tam

List Operations: Linked Lists (Removing Elements: 2)

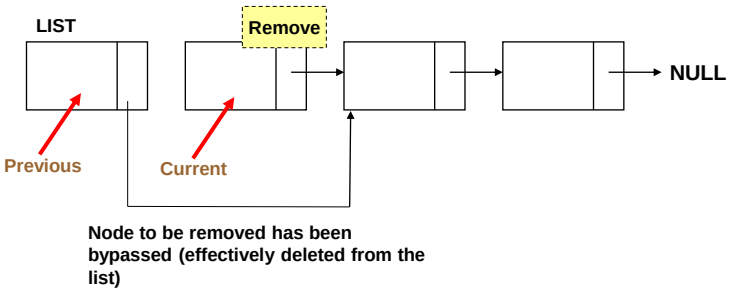
- Graphical illustration of the algorithm



James Tam

List Operations: Linked Lists (Removing Elements: 2)

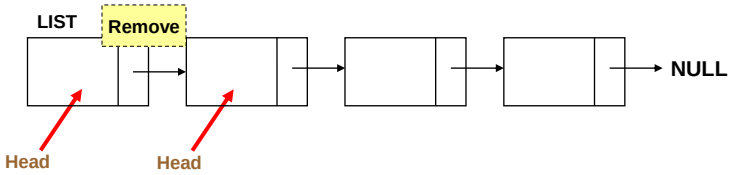
- Graphical illustration of the algorithm



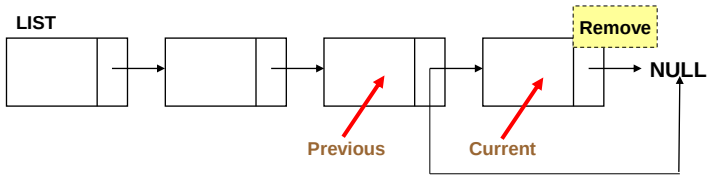
James Tam

List Operations: Linked Lists (Removing Elements: 3)

- The algorithm should work with the removal of any node
  - First node



  - Last node

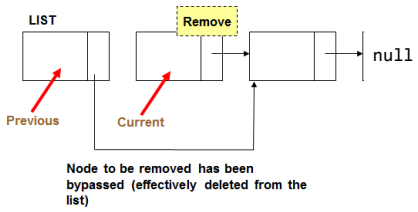


James Tam

List Operations: Linked Lists (Removing Elements: 4)

- The search algorithm will find the node to be deleted and mark it with a reference
- The node prior to the node to be deleted must also be marked.
- Pseudo code algorithm (removal)

  Previous node refers to the node referred by current node (by pass the node to be deleted)



James Tam

Manager.Remove()

```
public void remove ()
{

   // CASE 1: EMPTY LIST
   if (head == null)
       System.out.println("List is already empty:
                           Nothing to remove");

   // CASE 2: NON-EMPTY LIST
   else
   {
        removeNonempty();
    }
}
```

James Tam

Manager.RemoveNonempty()

```
// Case 2 & 3:
  private void removeNonempty()
  {
     BookNode previous = null;
     BookNode current = head;
     String searchName = null;
     boolean isFound = false;
     String currentName;
     Scanner in = new Scanner(System.in);
     System.out.print("Enter name of book to remove: ");
     searchName = in.nextLine();  // User selects name
```

James Tam

Manager.RemoveNonempty() : 2

```
    // Determine if match exists
    // current points to node to delete
    // previous is one node prior
    while ((current != null) && (isFound == false))
    {
        currentName = current.getData().getName();
        if (searchName.compareToIgnoreCase(currentName) ==
            MATCH)
            isFound = true;  // Match found, stop traversal
        else  // No match: move onto next node
        {
            previous = current;
            current = current.getNext();
        }
    }
```

James Tam

Manager.RemoveNonempty() : 3

```
        // CASE 2A OR 2B: MATCH FOUND (REMOVE A NODE)
        if (isFound == true)
        {
            System.out.println("Removing book called " +
                               searchName);
            // CASE 2A: REMOVE THE FIRST NODE
            if (previous == null)
                head = head.getNext();
            // CASE 2B: REMOVE ANY NODE EXCEPT FOR THE FIRST
            else
                previous.setNext(current.getNext());
        }
        // CASE 3: NO MATCHES FOUND (NOTHING TO REMOVE).
        else  // isFound == false
            System.out.println("No book called " + searchName +
                               " in the collection.");
    }
```

James Tam

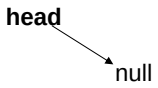
Removing A Node From An Empty List

```
// Main(), Case 1
listManager.eraseList();  // Reminder: Blows away entire list
listManager.display();
listManager.remove();     // Trying to remove element from empty
```

James Tam

Removing A Node From An Empty List (2)

- **Case 1**: Empty List



searchName:

isFound :

James Tam

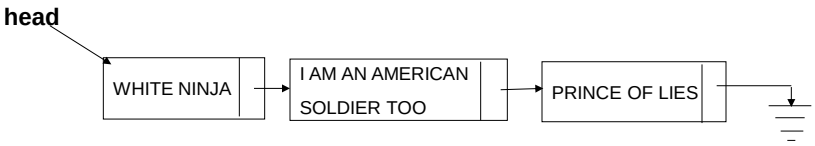
Removing A Node From A Non-Empty List

```
// Main(), Case 2A
listManager.add();
listManager.add();
listManager.add();
listManager.remove();
```

James Tam

Non-Empty List: Remove

- **Case 2A:** Remove first element



searchName: WHITE NINJA

isFound:

James Tam

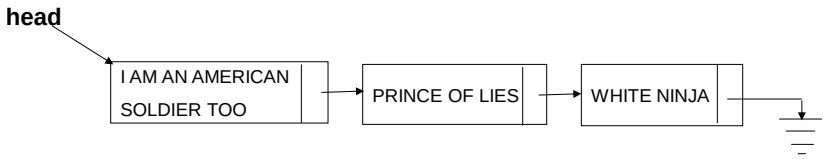
Removing Any Node Except The First

```
// Main(), Case 2B
listManager.add();  //e.g., add back "White Ninja"
listManager.remove();
```

James Tam

Non-Empty List: Remove (2)

- **Case 2B**: Remove any node except for the first



searchName: PRINCE OF LIES

isFound:

James Tam

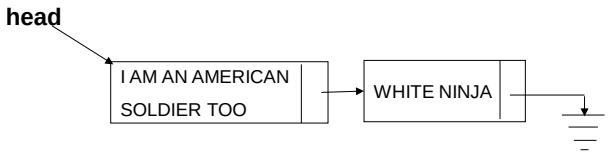
Removing Non-Existent Node

```
// Main() Case 3: no match
listManager.remove();
listManager.display();
```

James Tam

Non-Empty List: Trying To Remove Non-Existent Node

- **Case 3:** No match



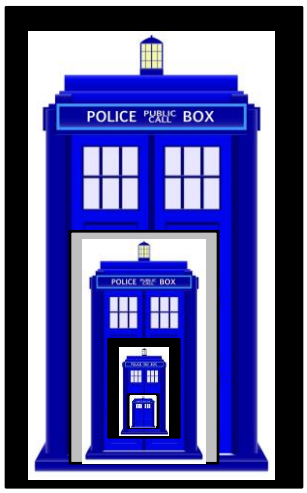
searchName: MOBY DICK

isFound:

James Tam

Related Material: Recursion

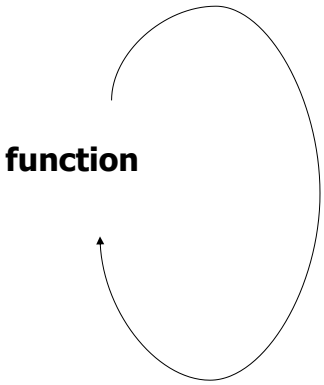
- "*A programming technique whereby a function or method calls itself either directly or indirectly*."



'Tardis' images: colourbox.com

James Tam

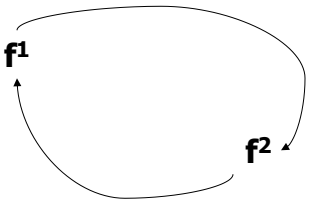
Direct Call




```
void fun()
   ...
   fun();
```

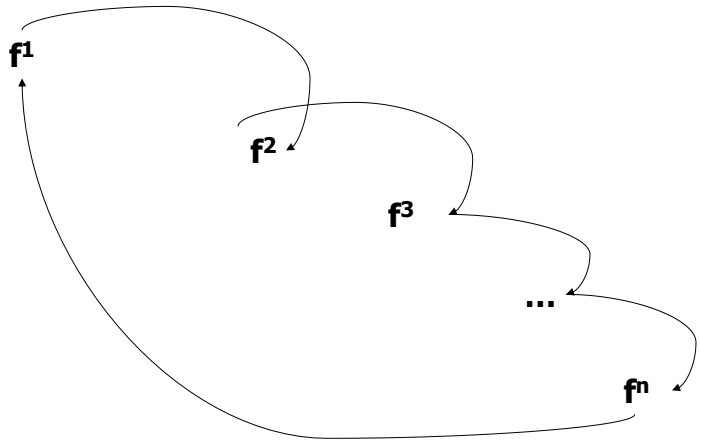
James Tam

Indirect Call



James Tam

Indirect Call



James Tam

Recursive Programs

- Location of full examples:
  - `/home/233/examples/linkedList/recursion`

James Tam

Types Of Recursion:

- **Tail recursion**:
  - Aside from a return statement, the last instruction in the recursive function or method is another recursive call.

```
fun(int x) {
    System.out.println(x);
    if (x < 10)
        fun(++x); // Last real instruction (implicit return)
}
```

  - This form of recursion can easily be replaced with a loop.
- **Non-tail recursion**:
  - The last instruction in the recursive function or method is NOT another recursive call e.g., an output message

```
fun(int x) {
    if (x < 10)
        fun(++x);
    System.out.println(x); // Last instruction
}
```

  - This form of recursion is difficult to replace with a loop (stopping condition occurs BEFORE the real work begins).

James Tam

Simple Counting Example

- First example: can be directly implemented as a loop

```
public class DriverTail
{
    public static void tail (int no)
    {
        if (no <= 3)
        {
            System.out.println(no);
            tail(no+1);
        }
        return;
    }

    public static void main (String [] args)
    {
        tail(1);
    }
}
```

James Tam

'Reversed' Counting Example

```
public class DriverNonTail
{
    public static void nonTail(int no)
    {
        if (no < 3)
            nonTail(no+1);
        System.out.println(no);
        return;
    }

    public static void main (String [] args)
    {
        nonTail(1);
    }
}
```

James Tam

Recursive List Display

- The pseudo code and the diagrammatic trace are the same as the iterative solution.
- The difference is that repetition occurs with repeated calls to a recursive method instead of a loop.
  - Calls:
    - `Driver.main() ->`
    - `Manager.displayRecursive() ->`
    - `Manager.displayAndRecurse()`
  - The first method called will be used for statements that only *execute once each time the list is displayed* (e.g., a header with underlining)
  - The second method called will be used to display a node at a time. After displaying the node the program moves onto the next node and calls the method again.

James Tam

Manager.DisplayRecursive()

```
public void displayRecursive()
{
    BookNode temp = head;
    System.out.println("DISPLAYING LIST (R)"); // Display once
    for (int i = 0; i < LIST_HEADER.length(); i++)
        System.out.print("-");
    System.out.println();
     if (temp == null)  // Case 1: Empty
        System.out.println("\tList is empty: nothing to
                            display");
    else // Case 2: Non-empty
    {
        int count = 1;
        displayAndRecurse(temp,count);
    }
    System.out.println();
}
```

James Tam

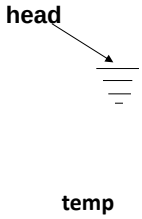
Manager.DisplayAndRecurse()

```
private void displayAndRecurse(BookNode temp, int count)
{
    // Stop when end of list reached
    if (temp == null)
        return;
    else
    {
        // Display data and move onto next element
        System.out.println("\t#" + count + ": " + temp);
        temp = temp.getNext();  // Get address of next node
        count = count + 1;
        displayAndRecurse(temp,count);
    }
}
```

James Tam

Recursive Display Of List

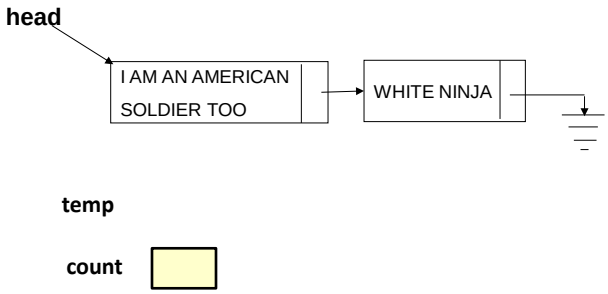
- **Case 1:** Empty list



James Tam

Recursive Display Of List

- **Case 2:** Non-empty list



James Tam

After This Section You Should Now Know

- What is a linked list and how it differs from an array implementation
- How to implement basic list operations using a linked list
  - Creation of new empty list
  - Destruction of the entire list
  - Display of list elements (iterative and recursive)
  - Searching the list
  - Inserting new elements
  - Removing existing elements
- How to write a recursive equivalent of an iterative solution
- How to trace a recursive program

James Tam