Introduction To Data Structures

This section introduces the concept of a data structure as well as providing the details of a specific example: 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 these 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.

What Is A Data Structure

• A composite type that has a set of basic operations (e.g., display elements of a list) that may be performed on instances of that type.
 - It can be accessed as a whole (e.g., pass the entire list as a parameter to a function).
 - Individual elements can also be accessed (e.g., update the value for a single list element).

• The type may be a built-in part of the programming language
 - e.g., lists are included with the Python language and need not be defined before they can be used

• The type may also be defined by the programmer inside a program (for languages which don't include this composite type)

```
class List
{
    :   :
}
```

What Is A Data Structure (2)

 - In some cases the data structure may only be partially implemented as part of the language, some operations must be manually written by the programmer.
 - Example: The ability to add an element to a list has been implemented as a pre-created Python function.

```
aGrid = []          # Creates an empty list
aGrid.append (12)    # Adds a number to the end of the list
```

 - In a language such as 'C' a list is implemented as an array but the operation to add elements to the end of the list must be written by the programmer.
 - Moral: when choosing a programming language look for built-in support for key features.

Lists

- Lists are a type of data structure (one of the simplest and most commonly used).
  - e.g., grades for a lecture can be stored in the form of a list
- List operations: creation, adding new elements, searching for elements, removing existing elements, modifying elements, displaying elements, sorting elements, deleting the entire list).
- List implementation in Java: array, linked list.

Arrays

- An array of 'n' elements will have an index of zero for the first element up to index (n-1) for the last element.
- The array index is an integer and indicates which element to access (excluding the index and just providing the name of the list means that the program is operating on the entire list).
- Similar to objects, arrays employ dynamic memory allocation (the name of the array is actually a reference to the array).
- Many utility methods exist.
- Several error checking mechanisms are available.

Arrays

- An array of 'n' elements will have an index of zero for the first element up to index (n-1) for the last element.

- The array index is an integer and indicates which element to access (excluding the index and just providing the name of the list means that the program is operating on the entire list).

- **Similar to objects, arrays employ dynamic memory allocation (the name of the array is actually a reference to the array).**

- Many utility methods exist.

- Several error checking mechanisms are available.

Declaring Arrays

- Arrays in Java involve a reference to the array so creating an array requires two steps:
  1) Declaring a reference to the array
  2) Allocating the memory for the array

Declaring A Reference To An Array

•**Format:**
```
// The number of pairs of square brackets specifies the number of
// dimensions.
<type> [] <array name>;
```

•**Example:**
```
int [] arr;
int [][] arr;
```

Allocating Memory For An Array

•**Format:**
```
<array name> = new <array type> [<no elements>];
```

•**Example:**
```
arr = new int [SIZE];
arr = new int [ROW SIZE][COLUMN SIZE];
```

(Both steps can be combined together):
```
int [] arr = new int[SIZE];
```

Arrays: An Example

•The name of the online example is: listsFirstExample

```java
public class Driver
{
    public static void main (String [] args)
    {
        int i;
        int len;
        int [] arr;
```

Arrays: An Example

```java
        Scanner in = new Scanner (System.in);
        System.out.print("Enter the number of array elements: ");
        len = in.nextInt ();
        arr = new int [len];
        System.out.println("Array Arr has " + arr.length + " elements.");
        for (i = 0; i < arr.length; i++)
        {
            arr[i] = i;
            System.out.println("Element[" + i + "]=" + arr[i]);
        }
    }
}
```

Arrays

• An array of 'n' elements will have an index of zero for the first element up to index (n-1) for the last element.

• The array index is an integer and indicates which element to access (excluding the index and just providing the name of the list means that the program is operating on the entire list).

• Similar to objects, arrays employ dynamic memory allocation (the name of the array is actually a reference to the array).

• Many utility methods exist.

• **Several error checking mechanisms are available.**
  - Null array references
  - Array bounds checking

Using A Null Reference

```
int [] arr = null;
arr[0] = 1;                        NullPointerException
```

Exceeding The Array Bounds

```
int [] arr  = new int [4];
int i;
for (i = 0; i <= 4; i++)
  arr[i] = i;
```

ArrayIndexOutOfBoundsException

(when i = 4)

List Operations: Arrays (Creation)

•Simply declare an array variable

<array name> = new *<array type>* [*<no elements>*];

List Operations: Arrays (Display)

• Unless it can be guaranteed that the list will always be full (unlikely) then some mechanism for determining that the end of the list has been reached is needed.

• If list elements cannot take on certain values then unoccupied list elements can be 'marked' with an invalid value.

• Example: grades

[0]	100
[1]	75
[2]	65
[3]	0
[4]	80
[5]	-1
[6]	-1
[7]	-1

List Operations: Arrays (Display: 2)

• If list elements can't be marked then a special variable ("last" index) can be used to mark the last occupied element.

• Alternatively a special variable can also be used to mark the next element free.

[0]		
[1]		
[2]		
[3]		←─lastOccupiedElement = 3
[4]		
[5]		
[6]		
[7]		

List Operations: Arrays (Search)

•Compare the item being searched for ("key") and compare vs. the list element.

[0]	Bob	◄—Index	Bob = Kayla? (no match)
[1]	Mary	◄—Index	Mary = Kayla? (no match)
[2]	Alice	◄—Index	Alice = Kayla? (no match)
[3]	Kayla	◄—Index	Kayla = Kayla? (Match found)
[4]	Bruce		
[5]			
[6]			
[7]			
[8]			
[9]			
[10]			
[11]		**Key**	Kayla

James Tam

List Operations: Arrays (Insertion)

•Insertion at the end.
  - Some mechanism is needed to either find or keep track of the last occupied element.

[0]	Bob		
[1]	Mary		
[2]	Alice	◄—Last	
[3]	Kayla	◄—Increment last (new index of last element)	
[4]			
[5]			
[6]			
[7]			
[8]			
[9]			
[10]			
[11]			

James Tam

List Operations: Arrays (Insertion: 2)

•In order insertion.
  - Some mechanism is needed to find the insertion point (search).
  - Elements may need to be shifted.

[0]	123
[1]	125
[2]	135
[3]	155
[4]	161
[5]	166
[6]	167
[7]	167
[8]	169
[9]	177
[10]	178
[11]	

165
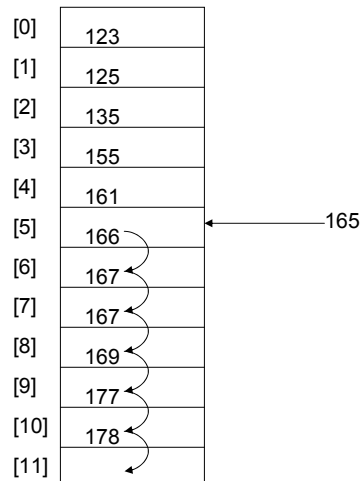
List Operations: Arrays (Removing Elements)

•A search is needed to find the element to remove.

•Depending upon the index of the element to be deleted, other elements may need to be shifted.

[0]	123
[1]	125
[2]	135
[3]	155
[4]	161
[5]	166
[6]	167
[7]	167
[8]	

Remove
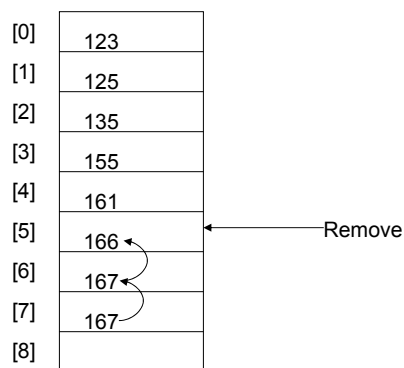
List Operations: Arrays (Destroying The Entire List)

- Recall that Java employs automatic garbage collection.
- Setting the reference to the array to null will eventually allow the array to be garbage collected.
  <array name> = null;
- Note: many languages do not employ automatic garbage collection and in those cases, either the entire array or each element must be manually de-allocated in memory.

Memory Leak

- A technical term for programs that don't free up dynamically allocated memory.
- It can be serious problem because it may result in a drastic slowdown of a program.

Linked Lists

• An alternate implementation of a list.

• The program code is somewhat more complex but some operations are more efficient (e.g., additions and deletions don't require shifting of elements).

• Also linked lists tend to be more memory efficient that arrays.
  - The typical approach with an array implementation is to make the array larger than needed. (Unused elements are allocated in memory and the space is wasted).
  - With a linked list implementation, elements only take up space in memory as they're needed.

Start   End

List Elements: Nodes

Freight "data"

Connector

| Data (data) | Pointer/reference (connector) | → |
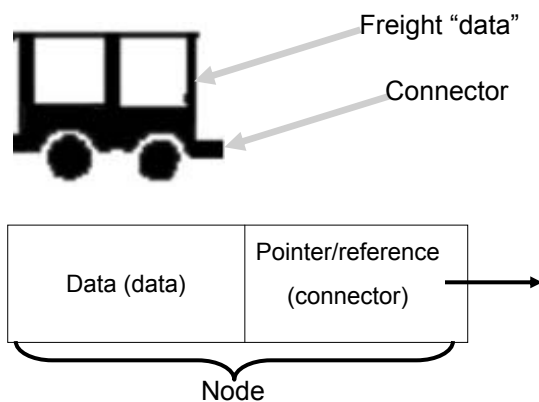
Node

Example: Defining A Node

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

Information stored by each element

Connects list elements

Example: Marking The Start Of The List

```
public class Manager
{
    private BookNode head;
}
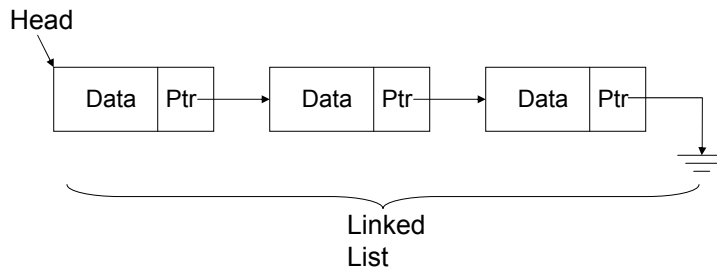```

**Case 1:** Empty list
head
　　　null

**Case 2:** Non-empty list
head
　　　┌──────────┐
　　　│ First node │
　　　└──────────┘

# 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.

Head

| Data | Ptr | → | Data | Ptr | → | Data | Ptr |

Linked
List

•Caution! Take care to ensure the reference to the first element is never lost.

---

# More On Connections: The Next Pointer

•Because linked lists only create elements as needed a special marker is needed for the end of the list.

•The 'next' attribute of a node will either:
  - Contain a reference/address of the next node in the list.

| Data | Next | → | Data | Next | → | ... |

  - Contain a null value.

| Data | Next |
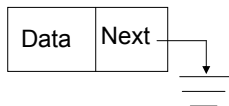
•(That means there is a reference to the start of the list, the next pointer of each element can be used to traverse the list).
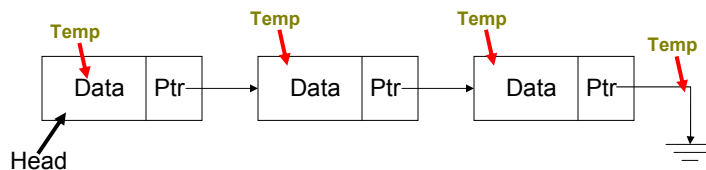
# List Operations: Linked Lists (Creation)

•After a type for the list has been declared then creating a new list requires that an instance be created and initialized.

•Example:

```
BookNode head = null;
```

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
```

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:

1. Temp = null (end)?

Temp 2. Data match?

| Data | Ptr | | Data | Ptr | | Data | Ptr |

Head
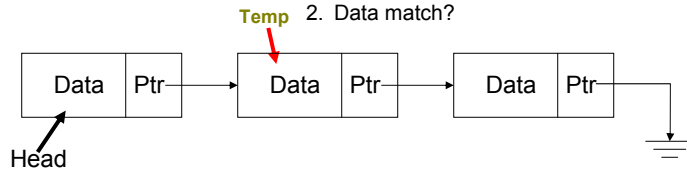
List Operations: Linked Lists (Search: 2)

•Pseudo code algorithm:

```
Temp refers to beginning of the list
If (temp is referring to 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
```

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).

•Once the 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.

List Operations: Linked Lists (Insertion)

•Graphical illustration of the algorithm:
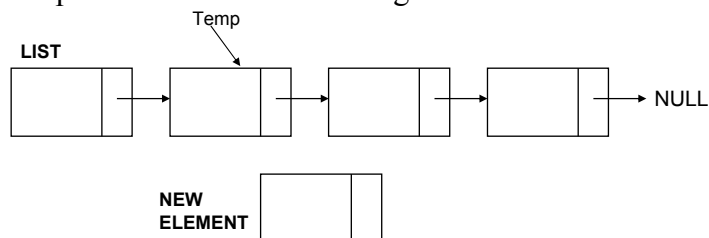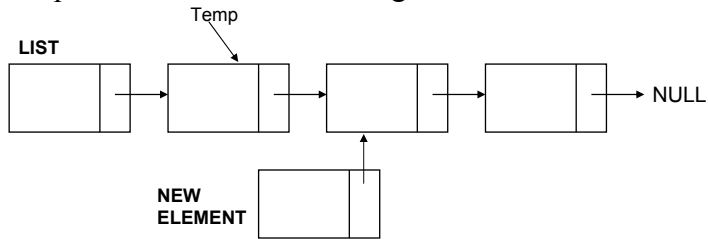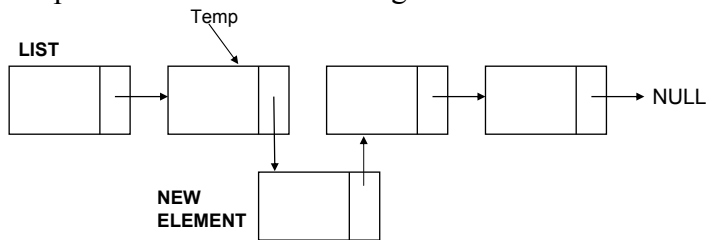
List Operations: Linked Lists (Insertion: 2)

•Graphical illustration of the algorithm:

James Tam

List Operations: Linked Lists (Insertion: 3)
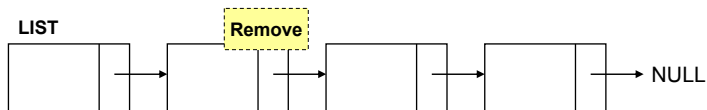
•Graphical illustration of the algorithm:

James Tam

List Operations: Linked Lists (Insertion: 4)

•Pseudo code algorithm:

Node to be inserted refers to node after insertion point

Node at insertion point refers to the node to be inserted

List Operations: Linked Lists (Removing Elements)

•Graphical illustration of the algorithm
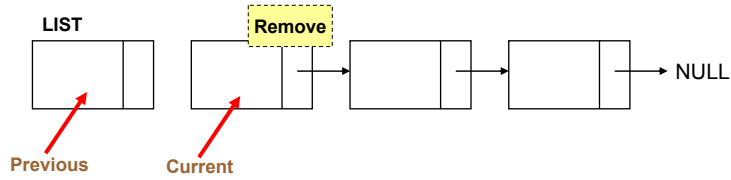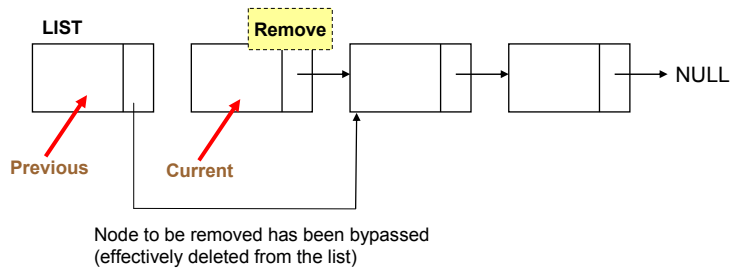
List Operations: Linked Lists (Removing Elements: 2)

•Graphical illustration of the algorithm

James Tam

List Operations: Linked Lists (Removing Elements: 2)

•Graphical illustration of the algorithm

Node to be removed has been bypassed
(effectively deleted from the list)

James Tam

List Operations: Linked Lists (Removing Elements: 3)

•Pseudo code algorithm:

Previous node refers to the node referred by current node

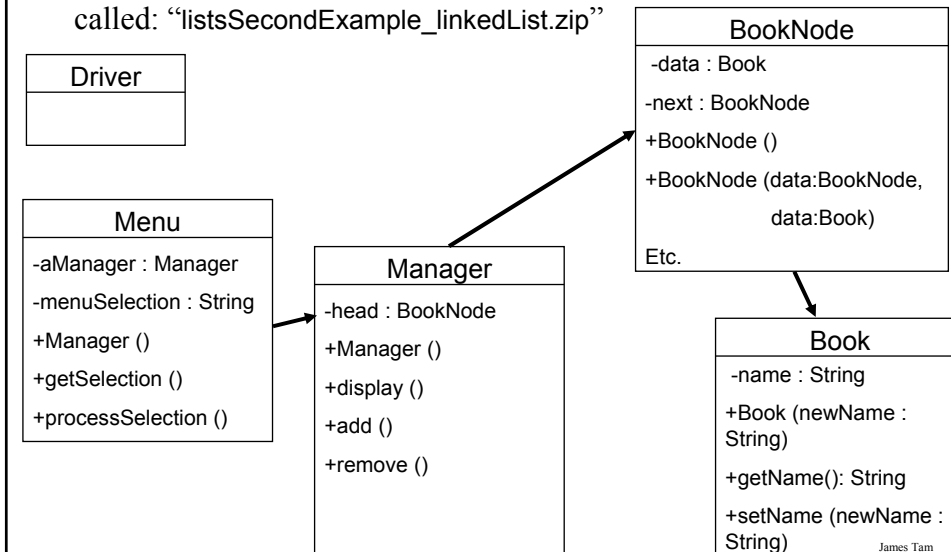## List Operations: Linked Lists (Destroying The Entire List)

•In Java: removing an entire list is similar to how it's done with the array implementation.

head = null;

•Important reminder: many languages to not employ automatic garbage collection and in those cases each node must be manually de-allocated in memory (step through each element in the list and free up the memory but take care not to lose the connection with the rest of the list).

Linked List Example

•The full example can be found on the course web page and is called: "listsSecondExample_linkedList.zip"

Driver

Menu
-aManager : Manager
-menuSelection : String
+Manager ()
+getSelection ()
+processSelection ()

Manager
-head : BookNode
+Manager ()
+display ()
+add ()
+remove ()

BookNode
-data : Book
-next : BookNode
+BookNode ()
+BookNode (data:BookNode, data:Book)
Etc.

Book
-name : String
+Book (newName : String)
+getName(): String
+setName (newName : String)

Linked List Example: Driver Class

```
public class Driver
{
    public static void main (String [] args)
    {
        Menu aMenu = new Menu ();
        aMenu.processSelection();
    } // End of main.
} // End of class Driver.
```

Linked List Example: Menu Class

```
public class Menu
{
  private Manager aManager;
  private String menuSelection;

  public Menu ()
  {
    aManager = new Manager ();
    menuSelection = null;
  }
```

Linked List Example: Menu Class (2)

```
public void display ()
{
  System.out.println("\n\nLIST MANAGEMENT PROGRAM: OPTIONS");
  System.out.println("\t(d)isplay list");
  System.out.println("\t(a)dd new book to end of list");
  System.out.println("\t(r)emove a book from the list");
  System.out.println("\t(q)uit program");
  System.out.print("Selection: ");
}

public void getSelection ()
{
  Scanner in = new Scanner (System.in);
  menuSelection = in.nextLine ();
}
```

Linked List Example: Menu Class (3)

```
public void processSelection ()
{
  do
  {
    display();
    getSelection();

    if (menuSelection.equalsIgnoreCase("d"))
      aManager.display ();
    else if (menuSelection.equalsIgnoreCase("a"))
      aManager.add ();
    else if (menuSelection.equalsIgnoreCase("r"))
      aManager.remove ();
    else if (menuSelection.equalsIgnoreCase("q"))
      System.out.println ("Quiting program.");
```

Linked List Example: Menu Class (4)

```
    else
      System.out.println("Please enter one of 'd','a','r' or 'q'");
  } while (!(menuSelection.equalsIgnoreCase("q")));
  }
}
```

Linked List Example: Manager Class

```
public class Manager
{
   private BookNode head;
   public Manager ()
   {
     head = null;
   }


 public void display()
 {
    // Code for displaying list: details to be provided later
 }
```

Linked List Example: Manager Class (2)

```
public void add ()
{
    // Code for displaying list: details to be provided later
}

public void remove ()
{
   // Code for displaying list: details to be provided later
}
```

Linked List Example: Class Node

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

   public BookNode ()
   {
     data = null;
     next = null;
   }

   public BookNode (Book data,
                    BookNode next)
   {
     setData(data);
     setNext(next);
   }
```

Linked List Example: Class Node (2)

```
   public void setData (Book data) {   this.data = data;  }

   public Book getData () {  return data;  }

   public void setNext (BookNode next) {   this.next = next;  }

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

Traversing The List: Display

- **Description:**
- 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 reference to refer to the next node).
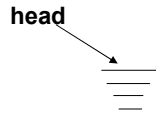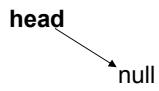
Traversing The List: Display (2)

```
public void display()
{
    int i = 1;
    BookNode temp = head;
    System.out.println("Displaying list");
    if (head == null)
        System.out.println("\tList is empty");
    while (temp != null)
    {
        System.out.println("\tTitle No. " + i + ": "+ temp.getData().getName());
        i = i + 1;
        temp = temp.getNext();
    }
}
```
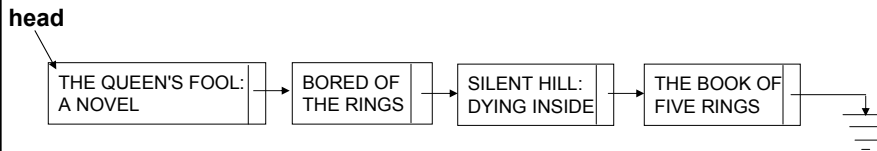
Traversing The List: Display (3)

head

null

head

Traversing The List: Display (4)

head

| THE QUEEN'S FOOL: A NOVEL | | → | BORED OF THE RINGS | → | SILENT HILL: DYING INSIDE | → | THE BOOK OF FIVE RINGS | |

Adding A Node To The End Of The List

- **Description:**
- Variables
  1. There are two references to the list:
     a) Current reference: traverses the list from beginning to end.
     b) Previous (to current reference): refers to the node immediately before the node referred to by current. (If current refers to the first node then previous is null – nothing comes before the first node).

Adding A Node To The End Of The List (2)

- **Steps:**
  1. Assign the current reference to the front of the list.
  2. If the current reference is null, then the list is empty. Add the node to the front of the list by changing the reference to the beginning (head) and stop.
  3. ("Two step" approach, an alternate is to "peek ahead") Otherwise traverse the list with two references, one reference (the current reference) goes past the end of the list (to the null value), the other reference (previous reference) stays one node behind the current reference.
  4. Attach the new node to the last node in the list (which can be reached by the previous reference).
  5. Whether the node is attached to an empty or non-empty list, the next reference of the new node becomes null (to mark the end of the list).

Adding A Node To The End Of The List (3)

```
public void add ()
{
 System.out.print("Enter a title for the book: ");
 Scanner in = new Scanner(System.in);
 String newName = in.nextLine();
 Book aBook = new Book (newName);
 BookNode aBookNode = new BookNode(aBook,null);

 if (head == null)
   head = aBookNode;
```
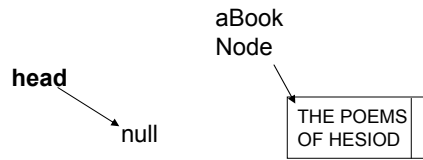
Adding A Node To The End Of The List (4)

```
 else
 {
   BookNode current = head;
   BookNode previous = null;
   while (current != null)
   {
    previous = current;
    current = current.getNext();
   }
   previous.setNext(aBookNode);
 }
}
```
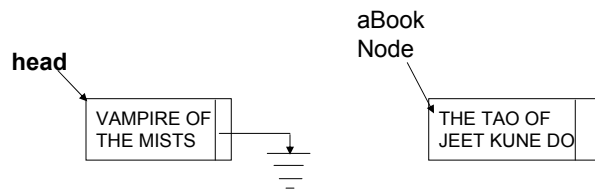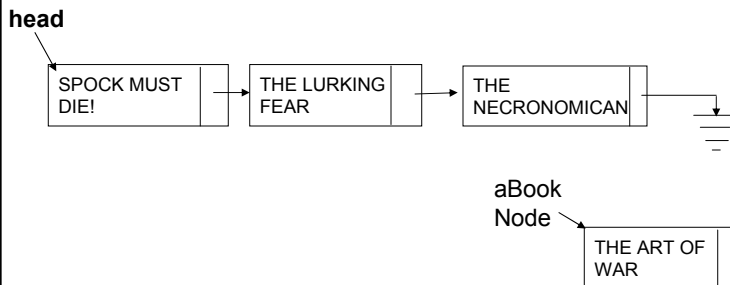
Adding A Node To The End Of The List (5)

aBook
Node

head

null

THE POEMS
OF HESIOD

Adding A Node To The End Of The List (6)

aBook
Node

head

VAMPIRE OF
THE MISTS

THE TAO OF
JEET KUNE DO

Adding A Node To The End Of The List (7)

head

| SPOCK MUST DIE! | | THE LURKING FEAR | | THE NECRONOMICAN | |

aBook
Node

| THE ART OF WAR | |

Removing A Node From The List

Description:

- Main variables:
    1. A temporary reference: refers to the node to be deleted. It is needed so that the program can retain a reference to this node and free up the memory allocated for it after the node has been 'bypassed' (step 4A on the next slides).
    2. A previous reference: refer to the node just prior to the one to be deleted. The 'next' field of this reference will be set to skip over the node to be deleted and will instead point to the node that immediately follows the node to be deleted.
    3. The head reference: The actual reference (and not a copy) is needed if the first node is deleted.
    4. The search key – in this example it is a string but it could be any arbitrary type as long as a comparison can be performed.
    5. A boolean variable that stores that status of the search (the search flag). (Start the search by assuming that it's false and the flag is set to true when a successful match occurs.

Removing A Node From The List (2)

- **Steps**

1. Initialize the main variables.
   a) The temporary reference starts at the front of the list.
   b) The boolean flag is set to false (no matches have been found yet).
   c) The previous reference is set to null (to signify that there is no element prior to the first element).

2. If the list is empty (temporary reference is null) display a status message to the user (e.g., "list is empty") and stop the removal process.

3. While the end of the list has not been reached (temporary reference is not null) AND no matches have been found yet (boolean flag is false) :
   a) Compare the search key with the appropriate field in the node referred to by the temporary reference.
   b) If there's a match then set the search flag to true (it's true that a match *has* been found now).
   c) If no match has been found set the previous reference to the node referred to by the temporary reference and move the temporary reference to the next node in the list.

Removing A Node From The List (3)

4. (At this point either the whole list has been traversed or there has been successful match and the search has terminated early):
   a. If the search flag is set to true then a match has been found.
      i. If the first node is the one to be deleted (previous reference is null) then set the head reference to the second node in the list.
      ii. If any other node is to be deleted then bypass this node by setting the 'next' field of the node referred to by the previous reference to the node immediately following the node to be deleted.
      iii. In both cases the temporary reference still refers to the node to be deleted. (If applicable) free up the allocated memory using the temporary reference.
   b. If the search flag is set to false no matches have been found, display a status message to the user (e.g., "no matches found").

Removing A Node From The List (4)

```
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
 {
    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();
```

Removing A Node From The List (5)

```
    while ((current != null) &&
         (isFound == false))
    {
       currentName = current.getData().getName();
       if (searchName.compareToIgnoreCase(currentName) == 0)
         isFound = true;
       else
       {
         previous = current;
         current = current.getNext();
       }
    }
```
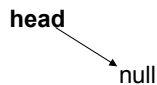
Removing A Node From The List (6)

```
        // 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
          System.out.println("No book called " + searchName + " in the
                              collection.");
      }
    }
  }
```

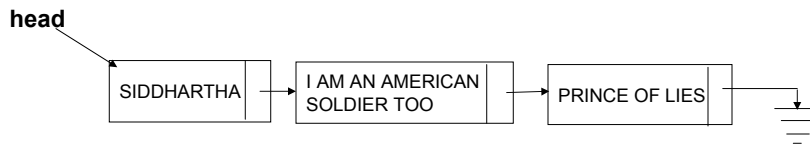
Removing A Node From The List (7)

• Case 1: Empty List

head

null

searchName: [   ]

isFound: [   ]

Removing A Node From The List (8)
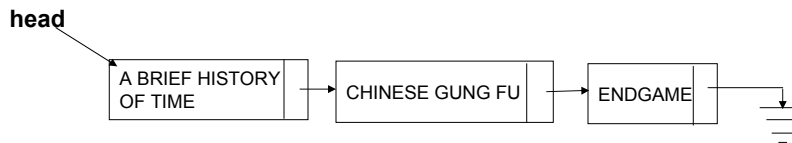
• Case 2A: Remove first element

head

| SIDDHARTHA | → | I AM AN AMERICAN SOLDIER TOO | → | PRINCE OF LIES | → |

searchName: SIDDHARTHA

isFound:

Removing A Node From The List (9)

• Case 2B: Remove any node except for the first

head

| A BRIEF HISTORY OF TIME | → | CHINESE GUNG FU | → | ENDGAME | → |

searchName: ENDGAME

isFound:

Removing A Node From The List (10)

•Case 3: No match

head

```
A CHRISTMAS          THE PRICE OF FREEDOM:          SPYWORLD
CAROL                A WING COMMANDER NOVEL
```

searchName: MOBY DICK
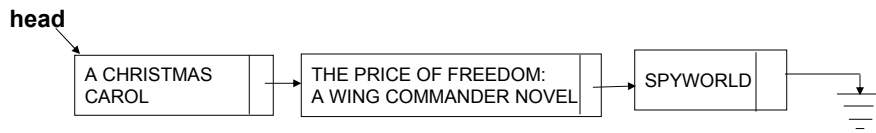
isFound:

Arrays Of Objects (References)

•An array of objects is actually an array of references to objects
  e.g., Foo [] arr = new Foo [4];

•The elements are initialized to null by default
  arr[0].setNum(1); ← **NullPointerException**

Arrays Of References To Objects: An Example

•The complete example will be covered in tutorial and can be found in UNIX under:
/home/219/tutorials/lists/listsTutorialExample_arrayReferences.zip

The Book Class

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

The Manager Class

```java
public class Manager
{
   public final int MAX_ELEMENTS = 10;
   private Book [] bookList;
   private int lastElement;

   public Manager ()
   {
      bookList = new Book[MAX_ELEMENTS];
      int i;
      for (i = 0; i < MAX_ELEMENTS; i++)
      {
         bookList[i] = null;
      }
      lastElement = -1;
   }
```

The Manager Class (2)

```java
public void display()
{
   int i;
   System.out.println("Displaying list");
   if (lastElement == -1)
      System.out.println("\tList is empty");
   for (i = 0; i <= lastElement; i++)
   {
      System.out.println("\tTitle No. " + (i+1) + ": "+ bookList[i].getName());
   }
}
```

The Manager Class (3)

```java
public void add ()
{
    String newName;
    Scanner in;
    if ((lastElement+1) < MAX_ELEMENTS)
    {
        System.out.print("Enter a title for the book: ");
        in = new Scanner (System.in);
        newName = in.nextLine ();
        lastElement++;
        bookList[lastElement] = new Book(newName);
    }
    else
    {
        System.out.print("Cannot add new element: ");
        System.out.println("List already has " + MAX_ELEMENTS + "
        elements.");
    }
}
```

The Manager Class (4)

```java
public void remove ()
{
    if (lastElement > -1)
    {
        bookList[lastElement] = null;
        lastElement--;
        System.out.println("Last element removed from list.");
    }
    else
    {
        System.out.println("List is already empty: Nothing to remove");
    }
}
}
```

The Menu Class

```java
public class Menu
{
  private Manager aManager;
  private String menuSelection;

  public Menu ()
  {
    aManager = new Manager ();
    menuSelection = null;
  }
```

The Menu Class (2)

```java
public void display ()
{
   System.out.println("\n\nLIST MANAGEMENT PROGRAM: OPTIONS");
   System.out.println("\t(d)isplay list");
   System.out.println("\t(a)dd new book to end of list");
   System.out.println("\t(r)emove book from the list");
   System.out.println("\t(q)uit program");
   System.out.print("Selection: ");
}

public void getSelection ()
{
   Scanner in = new Scanner (System.in);
   menuSelection = in.nextLine ();
}
```

The Menu Class (3)

```
public void processSelection ()
{
   do
   {
      display();
      getSelection();
      if (menuSelection.equalsIgnoreCase("d"))
         aManager.display ();
      else if (menuSelection.equalsIgnoreCase("a"))
         aManager.add ();
      else if (menuSelection.equalsIgnoreCase("r"))
         aManager.remove ();
      else if (menuSelection.equalsIgnoreCase("q"))
         System.out.println ("Quitting program.");
      else
         System.out.println("Please enter one of 'd','a','r' or 'q'");
   } while (!(menuSelection.equalsIgnoreCase("q")));
}
}
```

James Tam

The Driver Class

```
public class Driver
{
   public static void main (String [] args)
   {
      Menu aMenu = new Menu ();
      aMenu.processSelection();
   } // End of main.
} // End of class Driver.
```

James Tam

After This Section You Should Now Know

•What is a data structure

•How a data structure may be defined in Java

•Common list operations

•How a Java array employs dynamic memory allocation

•What is a memory leak

•How the common list operations are implemented using linked lists

•What are the advantages and disadvantages of implementing a list as an array vs. as a linked list