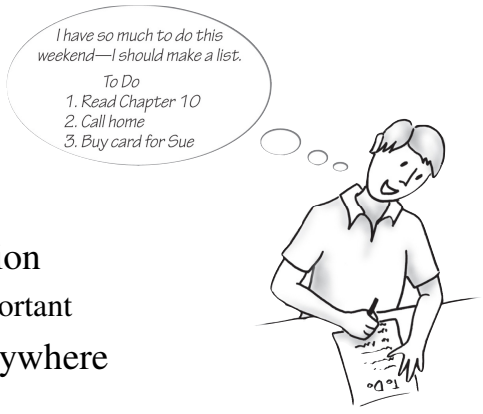
CSC 273 – Data Structures

Lecture 8 – Lists (*Extended*)

Lists

- A way to organize data
- Examples
  - To-do list
  - Gift lists
  - Grocery Lists
- Items in list have position
  - May or may not be important
- Items may be added anywhere

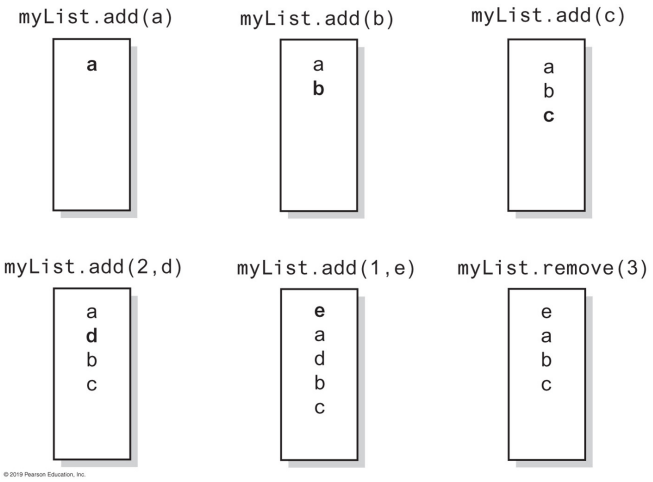


A To-Do List

Specifications for the ADT List

- `add(newEntry)`
- `add(newPosition, newEntry)`
- `remove(givenPosition)`
- `clear()`
- `replace(givenPosition, newEntry)`
- `getEntry(givenPosition)`
- `toArray()`
- `contains(anEntry)`
- `getLength()`
- `isEmpty()`

Specifications for the ADT List



The effect of ADT list operations on an initially empty list

Specifications for the ADT List

- Data
  - A collection of objects in a specific order and having the same data type
  - The number of objects in the same collection

Specifications for the ADT List

Pseudocode	UML	Description
add(newEntry)	**+add(newEntry: T): void**	Task: Adds newEntry to the end of the list. Input: newEntry is an object. Output: None.
add(newPosition, anEntry)	**+add(newPosition: integer, anEntry: T): void**	Task: Adds newEntry at position newPosition within the list. Position 1 indicates the first entry in the list. Input: newPosition is an integer, newEntry is an object. Output: Throws an exception if newPosition is invalid for this list before the operation.

Specifications for the ADT List

Pseudocode	UML	Description
remove(givenPosition)	**+remove(givenPosition: integer): T**	Task: Removes and returns the entry at position givenPosition. Input: givenPosition is an integer. Output: Either returns the removed entry or throws an exception if givenPosition is invalid for this list. Note that any value of givenPosition is invalid if the list is empty before the operation.
clear()	+clear(): void	Task: Removes all entries from the list. Input: None. Output: None.

Specifications for the ADT List

Pseudocode	UML	Description
replace(givenPosition, anEntry)	**+replace(givenPosition: integer, anEntry: T): T**	Task: Replaces the entry at position givenPosition with newEntry. Input: givenPosition is an integer, newEntry is an object. Output: Either returns the replaced entry or throws an exception if givenPosition is invalid for this list. Note that any value of givenPosition is invalid if the list is empty before the operation.

Specifications for the ADT List

Pseudocode	UML	Description
getEntry (givenPosition)	**+getEntry(givenPosition: integer): T**	Task: Retrieves the entry at position givenPosition. Input: givenPosition is an integer. Output: Either returns the entry at position givenPosition or throws an exception if givenPosition is invalid for this list. Note that any value of givenPosition is invalid if the list is empty before the operation.
toArray()	+toArray: T[ ]	Task: Retrieves all entries that are in the list in the order in which they occur. Input: None. Output: Returns a new array of the entries currently in the list.

Specifications for the ADT List

Pseudocode	UML	Description
contains(anEntry)	+contains(anEntry: T): boolean	Task: Sees whether the list contains anEntry. Input: anEntry is an object. Output: Returns true if anEntry is in the list, or false if not.
getLength()	+getLength(): integer	Task: Gets the number of entries currently in the list. Input: None. Output: Returns the number of entries currently in the list.
isEmpty()	+isEmpty(): boolean	Task: Sees whether the list is empty. Input: None. Output: Returns true if the list is empty, or false if not.

ListInterface.java

```
public interface ListInterface<T>  {

   /*
    *  add() – Adds a new entry to the end of this list.
    *  Entries currently in the list are unaffected.

    *  The list's size is increased by 1.
    *  @param newEntry  The object to be added

    *  as a new entry.
    */
   public void add(T newEntry);
```

```
  /*
   * add() – Adds a new entry at a specified position
   *         within this list.
   *  Entries originally at and above the specified
   *  position are at the next higher position within the
   *  list.

   *   The list's size is increased by 1.

   *   @param newPosition  An integer that specifies the
   *         desired position of the new entry.
   *   @param newEntry   The object to be added as a new
   *         entry.
   *   @throws  IndexOutOfBoundsException if either
   *    newPosition < 1 or newPosition > getLength() + 1.
   */
   public void add(int newPosition, T newEntry);
```

```
/*
 * remove() – Removes the entry at a given position from
 *            this list.

 * Entries originally at positions higher than the given
 * position are at the next lower position within the
 * list, and the list's size is decreased by 1.

 * @param givenPosition  An integer that indicates the
 *     position of the entry to be removed.
 * @return  A reference to the removed entry.
 * @throws  IndexOutOfBoundsException if either
 *    givenPosition < 1 or givenPosition > getLength()
 */
public T remove(int givenPosition);
```

```
/*
 * clear () – Removes all entries from this list. */
public void clear();

/*
 * replace() – Replaces the entry at a given position
 *         in this list.

 * @param givenPosition  An integer that indicates the
 *            position of the entry to be replaced.
 * @param newEntry  The object that will replace the
 *            entry at the position givenPosition.
 * @return  The original entry that was replaced.
 * @throws  IndexOutOfBoundsException if either
 *   givenPosition < 1 or givenPosition > getLength()
 */
public T replace(int givenPosition, T newEntry);
```

```
/*
 * getEntry() – Retrieves the entry at a given position
 *              in this list.
 *   @param givenPosition  An integer that indicates the
 *          position of the desired entry.
 *   @return  A reference to the indicated entry.
 *   @throws  IndexOutOfBoundsException if either
 *    givenPosition < 1 or givenPosition > getLength()
 */
public T getEntry(int givenPosition);
```

```
/*
 * toArray() – Retrieves all entries that are in this
 * list in the order in which they occur in the list.

 * @return  A newly allocated array of all the entries
 *    in the list.
 *  If the list is empty, the returned array is empty.
 */
public T[] toArray();
```

```
    /*
     * contains() - Sees whether this list contains a given
     *              entry.

     * @param anEntry  The object that is the desired
     *              entry.
     * @return  True if the list contains anEntry,
     *       or false if not.
     */
    public boolean contains(T anEntry);
```

```
    /*
     * getLength() - Gets the length of this list.

     * @return  The integer number of entries currently in
     *          the list
     */
    public int getLength();

    /*
     * isEmpty() - Sees whether this list is empty.

     * @return  True if the list is empty, or false if not.
     */
    public boolean isEmpty();
} // end ListInterface
```

Using the ADT List

A list of numbers that identify runners in the order in which they finish

RoadRace.java

```
/*
 * A driver that uses a list to track the runners in a
 * race as they cross the finish line.
 */
public class RoadRace  {
  public static void main(String[] args) {
    recordWinners();
  }  // end main

  public static void recordWinners() {
    ListInterface<String> runnerList = new AList<>();

    //  runnerList has only methods in ListInterface
    runnerList.add("16"); // Winner
    runnerList.add(" 4"); // Second place
    runnerList.add("33"); // Third place
    runnerList.add("27"); // Fourth place
    displayList(runnerList);
  } // end recordWinners
```

```
  public static void displayList
           (ListInterface<String> list)    {
    int numberOfEntries = list.getLength();
    System.out.println("The list contains "
          + numberOfEntries + " entries, as follows:");

    for (int position = 1;
           position <= numberOfEntries; position++)
      System.out.println(list.getEntry(position)
                + " is entry " + position);
    System.out.println();
   } // end displayList
} // end RoadRace
```

Using the ADT List

```
// Make an alphabetical list of names as students
enter a room
ListInterface<String> alphaList = new AList<>();

alphaList.add(1, "Amy");     // Amy
alphaList.add(2, "Elias");   // Amy Elias
alphaList.add(2, "Bob");     // Amy Bob Elias
alphaList.add(3, "Drew");    // Amy Bob Drew Elias
alphaList.add(1, "Aaron");
            // Aaron Amy Bob Drew Elias
alphaList.add(4, "Carol");
            // Aaron Amy Bob Carol Drew Elias
```

Using the ADT List

```
// Make a list of names as you think of them
ListInterface<Name> nameList = new AList<>();
Name amy = new Name("Amy", "Smith");
nameList.add(amy);
nameList.add(new Name("Tina", "Drexel");
nameList.add(new Name("Robert", "Jones");

Name secondName = nameList.getEntry(2);
```

A list of Name objects, rather than String

Java Class Library: The Interface `List`

- `public void add(int index, T newEntry)`
- `public T remove(int index)`
- `public void clear()`
- `public T set(int index, T anEntry) // Like replace`
- `public T get(int index)// Like getEntry`
- `public boolean contains(Object anEntry)`
- `public int size()// Like getLength`
- `public boolean isEmpty()`

Java Class Library: The Class **`ArrayList`**

- Available constructors
  - **`public ArrayList()`**
  - **`public ArrayList(int initialCapacity)`**
- Similar to **`java.util.vector`**
  - Can use either **`ArrayList`** or **`Vector`** as an implementation of the interface **`List`**.

Advantages of Linked Implementation

- Uses memory only as needed
- When entry removed, unneeded memory returned to system
- Avoids moving data when adding or removing entries

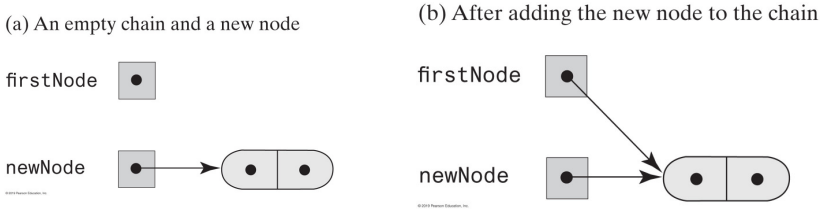
Adding a Node at Various Positions

- Possible cases:
  - Chain is empty
  - Adding node at chain's beginning
  - Adding node between adjacent nodes
  - Adding node to chain's end

Adding a Node

This pseudocode establishes a new node for the given data

1. newNode *references a new instance of* Node
2. *Place* newEntry *in* newNode
3. firstNode = *address of* newNode



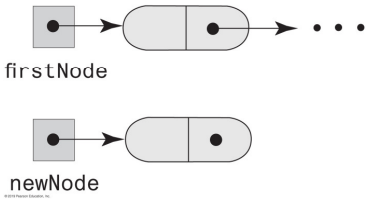
Adding a node to an empty chain

Adding a Node

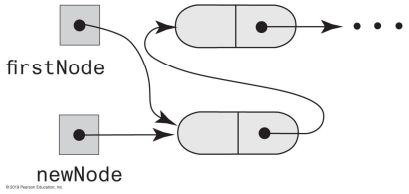
This pseudocode describes the steps needed to add a node to the beginning of a chain.

1. newNode *references a new instance of* Node
2. *Place* newEntry *in* newNode
3. *Set* newNode*'s link to* firstNode
4. *Set* firstNode *to* newNode

(a) A chain of nodes and a new node



(b) After adding the new node to the beginning of the chain



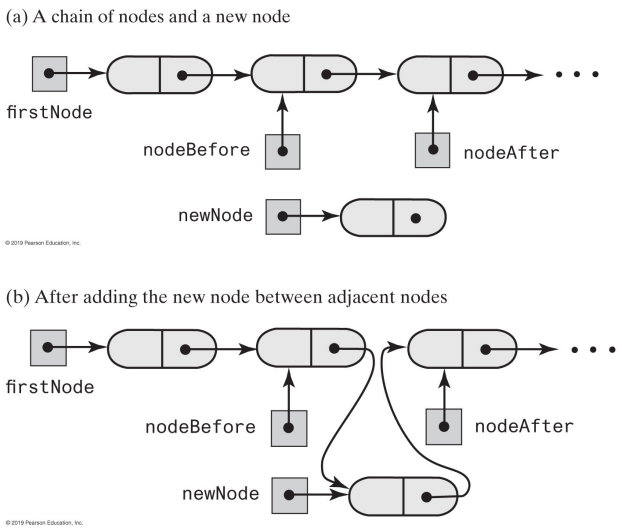
Adding a node to the beginning of a chain

Adding a Node

Pseudocode to add a node to a chain between two existing, consecutive nodes

1. newNode *references the new node*
2. *Place* newEntry *in* newNode
3. *Let* nodeBefore *reference the node that will be before the new node*
4. *Set* nodeAfter *to* nodeBefore*'s link*
5. *Set* newNode*'s link to* nodeAfter
6. *Set* nodeBefore*'s link to* newNode

Adding a Node



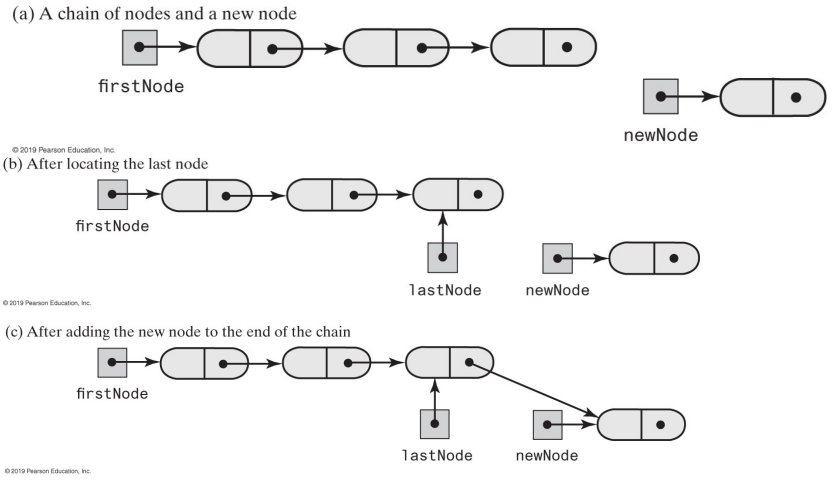
Adding a node between two adjacent nodes

Adding a Node

Steps to add a node at the end of a chain.

1. newNode *references a new instance of* Node
2. *Place* newEntry *in* newNode
3. *Locate the last node in the chain*
4. *Place the address of* newNode *in this last node*

Adding a Node



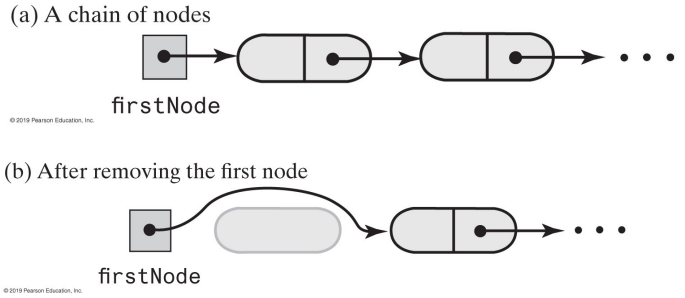
Adding a node to the end of a chain

Removing a Node

- Possible cases
  - Removing the first node
  - Removing a node other than first one

Steps For Removing The First Node

1. *Set* firstNode *to the link in the first node;* firstNode *now either references the second node or is* null *if the chain had only one node.*
2. *Since all references to the first node no longer exist, the system automatically recycles the first node's memory.*

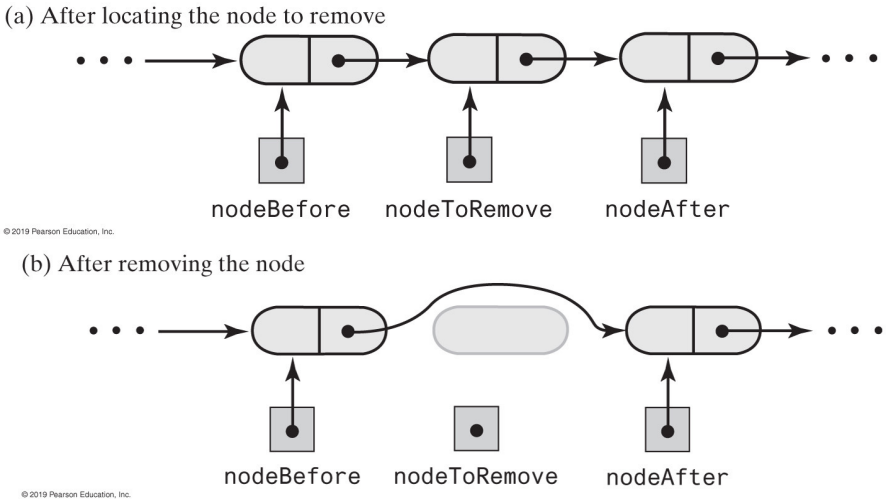


Removing the first node from a chain

Removing a node other than the first one

1. *Let* nodeBefore *reference the node before the one to be removed.*
2. *Set* nodeToRemove *to* nodeBefore*'s link;* nodeToRemove *now references the node to be removed.*
3. *Set* nodeAfter *to* nodeToRemove*'s link;* nodeAfter *now either references the node after the one to be removed or is* null.
4. *Set* nodeBefore*'s link to* nodeAfter. *(*nodeToRemove *is now disconnected from the chain.)*
5. *Set* nodeToRemove *to* null.
6. *Since all references to the disconnected node no longer exist, the system automatically recycles the node's memory.*

Removing a Node



Removing an interior node from a chain

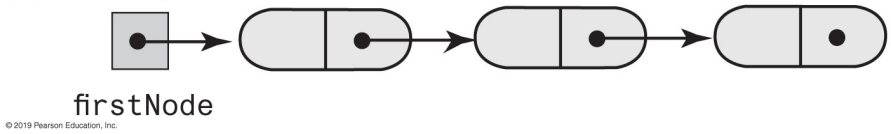
getNodeAt()

```
// Returns a reference to the node at a given
// position.

// Precondition: The chain is not empty;
//          1 <= givenPosition <= numberOfEntries.
private Node getNodeAt(int givenPosition)   {
  // Assertion: (firstNode != null) &&
  //          (1 <= givenPosition)
  //          && (givenPosition <= numberOfEntries)

  Node currentNode = firstNode;
```

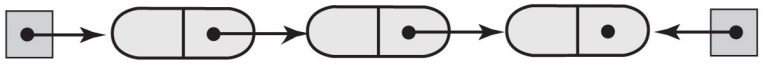
```
    // Traverse the chain to locate the desired node
    // (skipped if givenPosition is 1)
    for (int counter = 1;
           counter < givenPosition; counter++)
      currentNode = currentNode.getNextNode();
    // Assertion: currentNode != null

    return currentNode;
  } // end getNodeAt
```

Using a Tail Reference

(a) With only a head reference



(b) With both a head reference and a tail reference



Two linked chains

Data Fields and Constructor

```
public class LList<T>
      implements ListInterface<T> {
  // Reference to first node of chain
  private Node firstNode;
  private int  numberOfEntries;

  public LList()    {
      initializeDataFields();
  } // end default constructor

  public void clear()    {
      initializeDataFields();
   } // end clear
    … …
```

```
  // public methods add, remove, replace, getEntry,
  // contains, getLength, isEmpty, and toArray go
  // here

  // Initializes the class's data fields to indicate
  // an empty list.
  private void initializeDataFields()    {
    firstNode = null;
    numberOfEntries = 0;
  } // end initializeDataFields
```

```
// Returns a reference to the node at a given
// position.
// Precondition: The chain is not empty;
//         1 <= givenPosition <= numberOfEntries.
private Node getNodeAt(int givenPosition)   {
  // Assertion: (firstNode != null)
  //       && (1 <= givenPosition)
  //       && (givenPosition <= numberOfEntries)
  Node currentNode = firstNode;

  // Traverse the chain to locate the desired node
  // (skipped if givenPosition is 1)
  for (int counter = 1;
        counter < givenPosition; counter++)
  currentNode = currentNode.getNextNode();
```

```
  // Assertion: currentNode != null
  return currentNode;
} // end getNodeAt

private class Node  {
  private T    data; // Entry in list
  private Node next; // Link to next node

  private Node(T dataPortion)  {
    data = dataPortion;
    next = null;
  } // end constructor
```

```
    private Node(T dataPortion, Node nextNode) {
      data = dataPortion;
      next = nextNode;
    } // end constructor

    private T getData()  {
      return data;
    } // end getData

    private void setData(T newData)  {
      data = newData;
    } // end setData

    private Node getNextNode()  {
      return next;
    } // end getNextNode
```

```
    private void setNextNode(Node nextNode)   {
      next = nextNode;
    } // end setNextNode
  } // end Node
} // end LList
```

Adding at the End of a List

```
// OutOfMemoryError possible
public void add(T newEntry) {
  Node newNode = new Node(newEntry);

  if (isEmpty())
    firstNode = newNode;
  else   {  // Add to end of nonempty list
    Node lastNode = getNodeAt(numberOfEntries);

    // Make last node reference new node
    lastNode.setNextNode(newNode);
  } // end if
  numberOfEntries++;
} // end add
```

Adding at a Given Position

```
// OutOfMemoryError possible
public void add(int givenPosition, T newEntry) {
  if ((givenPosition >= 1)
       && (givenPosition <= numberOfEntries + 1)){
    Node newNode = new Node(newEntry);

    if (givenPosition == 1) {
      // Case 1
      newNode.setNextNode(firstNode);
      firstNode = newNode;
    }
```

```
    else  {
      // Case 2: list is not empty
      // and givenPosition > 1
      Node nodeBefore
            = getNodeAt(givenPosition - 1);
      Node nodeAfter = nodeBefore.getNextNode();
      newNode.setNextNode(nodeAfter);
      nodeBefore.setNextNode(newNode);
    } // end if
    numberOfEntries++;
  }
  else
    throw new IndexOutOfBoundsException
      ("Illegal position given to add "
          +  "operation.");
 } // end add
```

isEmpty()

```
public boolean isEmpty()    {
  boolean result;

  if (numberOfEntries == 0) {
    // Assertion: firstNode == null
    result = true;
  }
  else        {
    // Assertion: firstNode != null
    result = false;
  } // end if

  return result;
} // end isEmpty
```

toArray()

```
public T[] toArray()  {
  // The cast is safe because the new array
  // contains null entries
  @SuppressWarnings("unchecked")
  T[] result = (T[])new Object[numberOfEntries];

  int index = 0;
  Node currentNode = firstNode;
```

```
  while ((index < numberOfEntries)
              && (currentNode != null))        {
    result[index] = currentNode.getData();
    currentNode = currentNode.getNextNode();
    index++;
  } // end while

    return result;
 } // end toArray
```