Linked Lists

Computer Programming II
Profs. H. Harmanani and W. Keirouz
Lebanese American University
Byblos

Linked List Manipulation

- **Need techniques for carrying out computations on an entire list**
  - Computing the number of nodes in a list
  - General Linked List traversal
  - Finding a Node in a Linked List
  - Copying a Linked List
- **A major application for a linked list is a dynamic implementation of the Bag ADT**
  - We briefly look at the Dynamic Bag ADT.

`listLength` Method – Specification

- **Specification**
  - Computes the number of nodes in a linked list
- **Parameter**
  - *head*, a reference to the head node of the list (maybe empty)
- **Return Value**
  - `listLength` returns an integer that is equal to the number of nodes in the list.
  - The number of nodes must be less than `Integer.MAX_VALUE`.
- **Note that `listLength` must be a static method**
  - It is not activated by any one node.
  - It is rather activated by `IntNode.listLength`
    - Any ideas why?

`listLength` Method – Specification

- **An ordinary method is activated by the head node of a list**
  - It is easier however a static method is better since it can be used even with an empty list
  - The following code segment creates an empty list and then print its length

```
IntNode empty = null;
System.out.println(IntNode.listLength(empty);
```

`listLength` Method – Implementation

- **The implementation uses a reference variable, cursor, that steps through the nodes of the list one at a time**
- **The algorithm can be summarized as follows:**
  - Initialize a variable, *length*, that keeps track of the number of nodes
  - Make *cursor* refer to each node of the list, starting at the head node. Each time cursor moves, increment *length*
  - Once cursor becomes null, the method returns *length*
    - What kind of variables will *length* and *cursor* be?

`listLength` Method – Implementation

```
public static int listLength(IntNode head)
{
    IntNode cursor;
    int length;

    length = 0;
    for (cursor = head; cursor != null;
         cursor = cursor.link)
      length++;

    return(length);
}
```


Linked List Traversal – General Notes

- **The *pattern* that is used to traverse a linked list is the always the same and will be used throughout this course**
  - Start with a node, say `cursor`
  - `cursor.link` refers to the next node
- **To move `cursor` to the next node, need to advance the reference one node further**
  - Use `cursor = cursor.link` or `cursor = cursor.getLink();`
- **If there is no next node, `cursor` will become *null***

`listSearch` Method – Specification

- **Prototype**
  - `public static IntNode listSearch (IntNode head, int target)`
- **Specification**
  - Traverse a linked list searching for an element and returns a reference to the node that contains that element
- **Parameters**
  - `head` – the head reference for a linked list (maybe empty)
  - `target` – a piece of data to search for
- **Returns**
  - A reference to the first node that contains the specified target. If there is no such node, the null reference is returned

`listSearch` Method – Implementation

- **The method should return a reference to a node in a linked list that contains a certain parameter, say `target`**
- **If target does not appear in the list, the method should return a certain value that indicates this fact, say `null`**
- **The algorithm is then trivial**
  - Traverse the list using the usual list traversal pattern that we just described, using a local variable `cursor`
  - At every node of the list, we test for if we have found the element.
  - If so, return immediately. Otherwise, step through the next node

`listSearch` Method – Implementation

```
public static IntNode listSearch(IntNode head, int target)
{
    IntNode cursor;

    for (cursor = head; cursor != null; cursor = cursor.link)
       if (target == cursor.data)
              return cursor;
    return null;
}
```


`listPosition` Method – Specification

- **Prototype**
  - `public static IntNode listPosition (IntNode head, int position)`
- **Specification**
  - Finds a node in a linked list by its position
- **Parameters**
  - `head` – the head reference for a linked list (maybe empty)
  - `position` – a node number
- **Returns**
  - A reference to the node specified position in the list. If there is no such position, returns null
- **Throws**
  - `IllegalArgumentException` – Indicates that position is not positive

`listPosition` Method – Implementation

```
public static IntNode listPosition(IntNode head, int position)
{
    IntNode cursor;
    int i;

    if (position <= 0)
       throw new IllegalArgumentException
                              ("Position is not Positive");

    cursor = head;
    for (i = 0; (i < position) && (cursor != null); i++)
        cursor = cursor.link

    return cursor;
}
```

`listCopy` Method – Specification

- **Prototype**
  - `public static IntNode listCopy(IntNode source)`
- **Specification**
  - Copy a list
- **Parameter**
  - `source` – the head reference for a linked list that will be copied
- **Returns**
  - A copy of the linked list starting at `source`. Return value is the head reference for the copy
- **Throws**
  - `OutOfMemoryError` – Indicates that there is insufficient memory for the new list

`listCopy` Method – Implementation

- **Creates a completely separate copy of a linked list while the initial list remains intact**
- **Need two references that will be maintained as the head and tail references for the new list**
  - Creates a new head node of the new list. Tail and Head refer to this node
  - Make original list now refer to the second node. Add one node to the tail of the new list and move the tail forward.
  - Repeat above step until all nodes in the original list have been tarversed

`listCopy` Method – Implementation

```
public static IntNode listCopy(IntNode source)
{
    IntNode copyHead, copyTail;
    if (source == null)
       return null;
    copyHead = new IntNode(source.data, null);
    copyTail = copyHead;
    while (source.link != null) {
       source = source.link;
       copyTail.addNodeAfter(source.data);
       copyTail = copyTail.link;
    }
    return copyHead;
}
```


`listCopyWithTail` Method – Specification

- **Prototype**
  - `public static IntNode[] listCopyWithTail (IntNode source)`
- **Specification**
  - Copy a list, returning both a head and a tail reference
- **Parameter**
  - `source` – the head reference for a linked list that will be copied
- **Returns**
  - A copy of the linked list starting at `source`. Return value is an array where [0] element is a head reference for the copy and the [1] element is a tail reference for the copy
- **Throws**
  - `OutOfMemoryError` – Indicates that there is insufficient memory for the new list

`listCopyWithTail` – Implementation

- **Makes a copy of a list with the difference is that it returns two references, a head and a tail reference**
  - Return value is an array with two components
    - [0] component contains the head reference for the new list
    - [1] component contains the tail reference for the new list
- **A method can return an array**
  - Useful whenever we need to return more than one piece of information

`listCopyWithTail`– Implementation

```
public static IntNode[] listCopyWithTail(IntNode source)
{
     IntNode copyHead, copyTail;
     IntNode[] answer = new IntNode[2];

     if (source == null) return answer;

     copyHead = new IntNode(source.data, null);
     copyTail = copyHead;
     while (source.link != null) {
        source = source.link;
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link;
     }

     answer[0] = copyHead;
     answer[1] = copyTail;
     return answer;
}
```


`listPart` Method – Specification

- **Prototype**
  - `public static IntNode[] listPart(IntNode start, IntNode end)`
- **Specification**
  - Copy part of a linked list, providing head and tail reference for the new copy
- **Parameters**
  - `start` and `end` – references to two nodes of a linked list
- **Returns**
  - A copy of part of a linked list, from the specified start node to the specified end node. Return value is an array where [0] component is a head reference for the copy and the [1] component is a tail reference for the copy
  - Throws – `IllegalArgumentException` (start and end do not satisfy the precondition) and `OutOfmemoryError` (Indicates that there is insufficient memory for the new list

`listPart` Method – Implementation

```
public static IntNode[] listPart (IntNode start, IntNode end) {
   IntNode copyHead, copyTail;
   IntNode[] answer = new IntNode[2];
   if (source == null)
       throw new IllegalArgumentException("Start is null");
   if (end == null)
       throw new IllegalArgumentException("end is null");
   copyHead = new IntNode(source.data, null);
   copyTail = copyHead;
   while (start != null) {
       start = start.link;
       if (start == null)
          throw new IllegalArgumentException("end not found");
       copyTail.addNodeAfter(start.data);
       copyTail = copyTail.link;
   }
   answer[0] = copyHead; answer[1] = copyTail;
   return answer;
}
```

YABI: Yet Another Bag Implementation!

- **Can rewrite the Bag ADT using a linked list**
- **See Code for example and difference with the static approach**

`remove` method – Implementation

```
public boolean remove(int target)
{
     IntNode targetNode;

     targetNode = IntNode.listSearch(head, target);

     if (targetNode == null)
        return false;
     else
     {
        targetNode.SetData(head.getData() );
        head = head.getLink();
        manyNodes--;
        return true;
     }
}
```


`Grab` method – Specification

- **Prototype**
  - `public int grab ()`
- **Specification**
  - Accessor method to retrieve a random element from this bag
- **Returns**
  - A randomly selected element from this bag
- **Throws**
  - `IllegalStateException` – Indicates that the bag is empty

`Grab` method – Implementation

- **Generate a random number between 1 and the size of the bag**
  - Use `random()` method that generates a random number between 0.0 and 1.0
- **Use the random value to select a node from the bag**
  - Use `listPosition` method

`Grab` method – Implementation

```
public int grab()
{
     IntNode cursor;
     int i;

     if (manyNodes == 0)
        throw new IllegalStateException("Bag size is zero");

     i = (int) ( Math.random() * manyNodes ) + 1;
     cursor = IntNode.listPosition (head, i);

     return cursor.getData();
}
```