CSC115 Lecture 7

LillAnne Jackson

Today:

1. Reference-Based List ADT
2. Measuring time
3. Comparing List and Array

IntegerList ADT

A collection of data

- Head

A set of operations on that data

- `void addFront (int x);`
- `void addBack (int x);`
- `int size();`
- `int get (int pos);`
- `String toString();`

Other operations typical in List ADTs:

- `void removeFront();`
- `void removeBack();`
- `void addAt(int x, int pos);`
- `void removeAt(int pos);`

Measuring Time

How long does each operation take?

*Its not about the *actual* amount of time, just the relative amounts!*

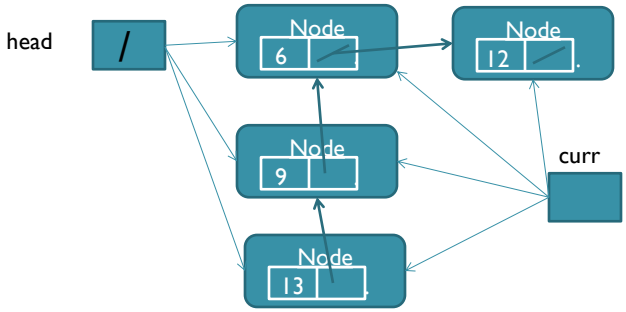
- `addFront (int x);`
- `addBack (int x);`
- `size();`
- `get (int pos);`
- `toString();`
- `removeFront();`
- `removeBack();`
- `addAt(int x,int pos);`
- `removeAt(int pos);`

Array	Linked List
O(n)	O(1)
O(n)	O(n)
O(1)	O(n)
O(1)	O(n)
O(n)	O(n)
O(n)	O(1)
O(1)	O(n)
O(n)	O(n)
O(n)	O(n)

Worst Case Analysis

Reference-Based Linked Lists

```
LinkedList testList = new LinkedList();
testList.addFront(6);
testList.addFront(9);
testList.addBack(12);
testList.addFront(13);
testList.removeBack();
```



Variations of the Linked List: Tail References

- Points at the end of the linked list
- To add a node to the end of a linked list

```
tail.setNext(new Node(request, null));
```

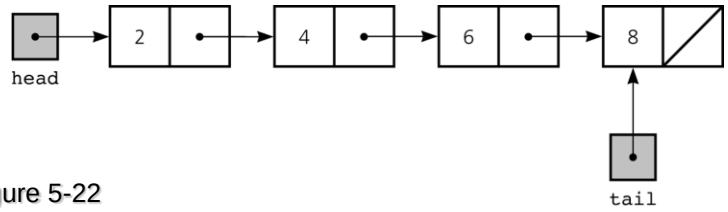


Figure 5-22

A linked list with ***head*** and ***tail*** references

Measuring Time

With Tail Reference

How long does each operation take?

*Its not about the *actual* amount of time, just the relati amounts!*

- **addFront (int x);**
- **addBack (int x);**
- **size();**
- **get (int pos);**
- **toString();**
- **removeFront();**
- **removeBack();**
- **addAt(int x,int pos);**
- **removeAt(int pos);**

Array	Linked List
O(n)	O(1)
O(n)	O(1)
O(1)	O(n)
O(1)	O(n)
O(n)	O(n)
O(n)	O(1)
O(1)	O(1)
O(n)	O(n)
O(n)	O(n)

Circular Linked List

- Last node references the first node
- Every node has a successor

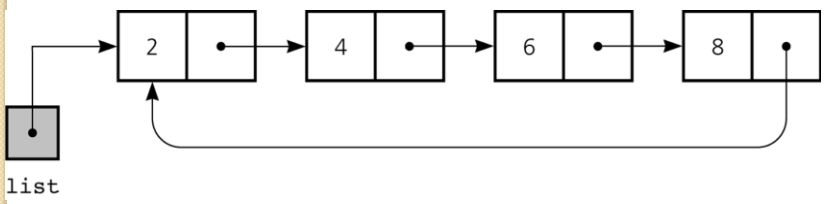


Figure 5-23

A circular linked list

Doubly Linked List

- Each node references both its predecessor and its successor

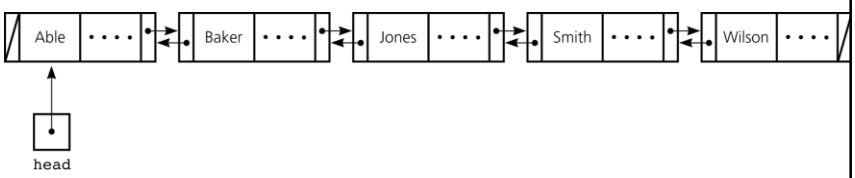


Figure 5-26

A doubly linked list

Doubly Linked List

- To delete the node that `curr` references

```
curr.getPrecede().setNext(curr.getNext());
curr.getNext().setPrecede(curr.getPrecede());
```

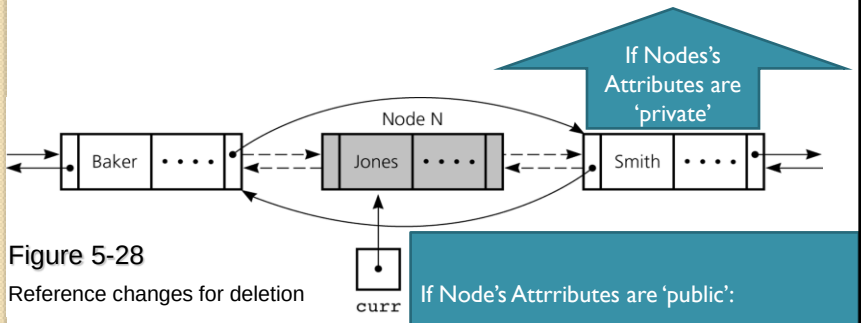


Figure 5-28
Reference changes for deletion

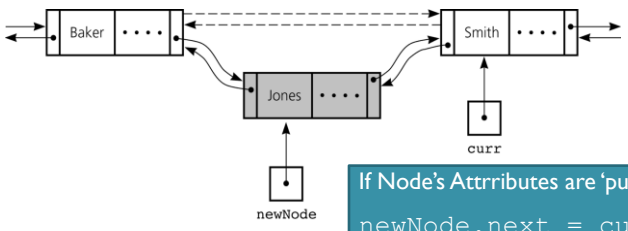
If Node's Attrributes are 'public':

```
curr.prev.next= curr.next;
curr.next.prev=curr.prev;
```


Doubly Linked List

- To insert a new node that `newNode` references before the node referenced by `curr`

```
newNode.setNext(curr);
newNode.setPrecede(curr.getPrecede());
curr.setPrecede(newNode);
newNode.getPrecede().setNext(newNode);
```



If Node's Attrributes are 'public':

```
newNode.next = curr;
newNode.prev = curr.prev;
curr.prev=newNode;
newNode.prev.next = newNode;
```

Passing a Linked List to a Method

- A method with access to a linked list's `head` reference has access to the entire list
- When head is an actual argument to a method, its value is copied into the corresponding formal parameter

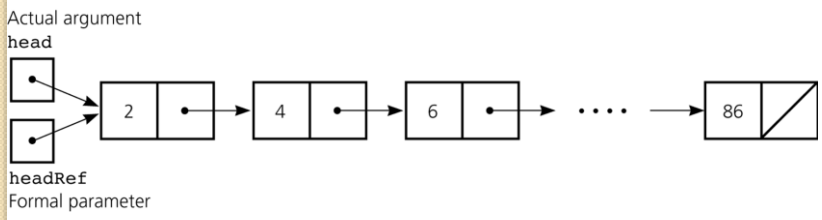


Figure 5-19

A head reference as an argument

Processing Linked Lists Recursively

- Traversal
  - Recursive strategy to display a list
    - Write the first node of the list
    - Write the list minus its first node
  - Recursive strategies to display a list backward
    - `writeListBackward` strategy
      - Write the last node of the list
      - Write the list minus its last node backward
    - `writeListBackward2` strategy
      - Write the list minus its first node backward
      - Write the first node of the list

Processing Linked Lists Recursively

- Insertion
  - Recursive view of a sorted linked list

    The linked list that `head` references is a sorted linked list if `head` is `null` (the empty list is a sorted linked list)

    or

    `head.getNext()` is `null` (a list with a single node is a sorted linked list)

    or

    `head.getItem() < head.getNext().getItem()`, and `head.getNext()` references a sorted linked list

