Lists & Trees

Lecture 8
CS2110 – Fall 2008
Announcements

- A1 grades & solutions are up
  - Submit regrades online
  - Regrades accepted until 9/29

- New Recitation Section
  - Wednesday evening -- check website for exact time & place
List Overview

• Purpose
  ▪ Maintain an ordered set of elements (with possible duplication)

• Common operations
  ▪ Create a list
  ▪ Access elements of a list sequentially
  ▪ Insert elements into a list
  ▪ Delete elements from a list

• Arrays
  ▪ Random access : )
  ▪ Fixed size: cannot grow or shrink after creation : (

• Linked Lists
  ▪ No random access : (
  ▪ Can grow and shrink dynamically : )
A Simple List Interface

```java
public interface List<T> {
    public void insert(T element);
    public void delete(T element);
    public boolean contains(T element);
    public int size();
}
```
List Data Structures

- **Array**
  - Must specify array size at creation
  - Insert, delete require moving elements
  - Must copy array to a larger array when it gets full

- **Linked list**
  - uses a sequence of linked cells
  - we will define a class ListCell from which we build lists

```
  24   -7   87   78   
```

eight empty

```
  24

  -7

  87

  78
```
List Terminology

- Head = first element of the list
- Tail = rest of the list
class ListCell<T> {
    private T datum;
    private ListCell<T> next;

    public ListCell(T datum, ListCell<T> next) {
        this.datum = datum;
        this.next = next;
    }

    public T getDatum() { return datum; }
    public ListCell<T> getNext() { return next; }
    public void setDatum(T obj) { datum = obj; }
    public void setNext(ListCell<T> c) { next = c; }
}
Building a Linked List

ListCell<Integer> c
    = new ListCell<Integer>(new Integer(24), null);

Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);

ListCell<Integer> p =
    new ListCell<Integer>(t,
        new ListCell<Integer>(s,
            new ListCell<Integer>(e, null)));
Another way:

```java
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
//Can also use "autoboxing"

ListCell<Integer> p = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);
```

**Note:** `p = new ListCell<Integer>(s, p);` does *not* create a circular list!
Building a Linked List (cont’d)

Another way:

```java
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
//Can also use "autoboxing"
```

```
ListCell<Integer> p
    = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);
```

**Note:** `p = new ListCell<Integer>(s,p);` does *not* create a circular list!
Building a Linked List (cont’d)

Another way:

```java
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
//Can also use "autoboxing"

ListCell<Integer> p = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);
```

Note: p = new ListCell<Integer>(s,p); does not create a circular list!
Another way:

```java
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
//Can also use "autoboxing"
```

```java
ListCell<Integer> p
    = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);
```

**Note:** p = new ListCell<Integer>(s, p); does not create a circular list!
Another way:

```java
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
//Can also use "autoboxing"

ListCell<Integer> p
    = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);
```

**Note:** `p = new ListCell<Integer>(s, p);` does not create a circular list!
Building a Linked List (cont’d)

Another way:

```java
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
//Can also use "autoboxing"
```

```java
ListCell<Integer> p
    = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);
```

Note: `p = new ListCell<Integer>(s, p);` does *not* create a circular list!
Another way:

```
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
//Can also use "autoboxing"
```

```
ListCell<Integer> p
    = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);
```

Note: `p = new ListCell<Integer>(s, p);` does not create a circular list!
Accessing List Elements

- Linked Lists are sequential-access data structures.
  - To access contents of cell n in sequence, you must access cells 0 ... n-1
- Accessing data in first cell: `p.getDatum()`
- Accessing data in second cell: `p.getNext().getDatum()`
- Accessing next field in second cell: `p.getNext().getNext()`

- Writing to fields in cells can be done the same way
  - Update data in first cell: `p.setDatum(new Integer(53));`
  - Update data in second cell: `p.getNext().setDatum(new Integer(53));`
  - Chop off third cell: `p.getNext().setNext(null);`
Access Example: Linear Search

```java
// Scan list looking for x, return true if found
public static boolean search(Object x, ListCell c) {
    for (ListCell lc = c; lc != null; lc = lc.getNext()) {
        if (lc.getDatum().equals(x)) return true;
    }
    return false;
}
```

Note: we’ve left off the <Integer> for simplicity
Access Example: Linear Search

// Scan list looking for x, return true if found
public static boolean search(Object x, ListCell c) {
    for (ListCell lc = c; lc != null; lc = lc.getNext()) {
        if (lc.getDatum().equals(x)) return true;
    }
    return false;
}

// Here is another version. Why does this work?
public static boolean search(Object x, ListCell c) {
    for (; c != null; c = c.getNext()) {
        if (c.getDatum().equals(x)) return true;
    }
    return false;
}

Note: we’ve left off the <Integer> for simplicity
Recursion on Lists

• Recursion can be done on lists
  ▪ Similar to recursion on integers

• Almost always
  ▪ Base case: empty list
  ▪ Recursive case: Assume you can solve problem on the tail, use that in the solution for the whole list

• Many list operations can be implemented very simply by using this idea
  ▪ Although some are easier to implement using iteration
Recursive Search

• Base case: empty list
  ▪ return false

• Recursive case: non-empty list
  ▪ if data in first cell equals object x, return true
  ▪ else return the result of doing linear search on the tail
public static boolean search(Object x, ListCell c) {
    if (c == null) return false;
    if (c.getDatum().equals(x)) return true;
    return search(x, c.getNext());
}
Recursive Search

public static boolean search(Object x, ListCell c) {
    if (c == null) return false;
    if (c.getDatum().equals(x)) return true;
    return search(x, c.getNext());
}

public static boolean search(Object x, ListCell c) {
    return c != null &&
            (c.getDatum().equals(x) || search(x, c.getNext()));
}
Reversing a List

- Given a list, create a new list with elements in reverse order
- Intuition: think of reversing a pile of coins

```java
public static ListCell reverse(ListCell c) {
    ListCell rev = null;
    for (; c != null; c = c.getNext()) {
        rev = new ListCell(c.getDatum(), rev);
    }
    return rev;
}
```

- It may not be obvious how to write this recursively...
public static ListCell reverse(ListCell c) {
    return reverse(c, null);
}

private static ListCell reverse(ListCell c, ListCell r) {
    if (c == null) return r;
    return reverse(c.getNext(), new ListCell(c.getDatum(), r));
}
List with Header

• Sometimes it is preferable to have a List class distinct from the ListCell class
• The List object is like a head element that always exists even if list itself is empty

```java
class List {
    protected ListCell head;
    public List(ListCell c) {
        head = c;
    }
    public ListCell getHead() {
        ......
    }
    public void setHead(ListCell c) {
        ......
    }
}
```
Variations on List with Header

- Header can also keep other info
  - Reference to last cell of list
  - Number of elements in list
  - Search/insertion/deletion as instance methods
  - ...

Heap
Variations on List with Header

• Header can also keep other info
  ▪ Reference to last cell of list
  ▪ Number of elements in list
  ▪ Search/insertion/deletion as instance methods
  ▪ ...

Heap
Variations on List with Header

• Header can also keep other info
  ▪ Reference to last cell of list
  ▪ Number of elements in list
  ▪ Search/insertion/deletion as instance methods
  ▪ ...

Heap
Special Cases to Worry About

- Empty list
  - add
  - find
  - delete
- Front of list
  - insert
- End of list
  - find
  - delete
- Lists with just one element
Example: Delete from a List

- Delete *first occurrence* of x from a list
- Intuitive idea of recursive code:
  - If list is empty, return null
  - If datum at head is x, return tail
  - Otherwise, return list consisting of
    - head of the list, and
    - List that results from deleting x from the tail
Example: Delete from a List

- Delete *first occurrence* of x from a list
- Intuitive idea of recursive code:
  - If list is empty, return null
  - If datum at head is x, return tail
  - Otherwise, return list consisting of
    - head of the list, and
    - List that results from deleting x from the tail

```java
// recursive delete
public static ListCell delete(Object x, ListCell c) {
    if (c == null) return null;
    if (c.getDatum().equals(x)) return c.getNext();
    c.setNext(delete(x, c.getNext()));
    return c;
}
```
Iterative Delete

- Two steps:
  - Locate cell that is the predecessor of cell to be deleted (i.e., the cell containing x)
    - Keep two cursors, scout and current
    - scout is always one cell ahead of current
    - Stop when scout finds cell containing x, or falls off end of list
  - If scout finds cell, update next field of current cell to splice out object x from list

- Note: Need special case for x in first cell

```
head: ListCell:

-7
-24
36
87

List:

p List:
```

delete 36 from list
Iterative Delete

• Two steps:
  ▪ Locate cell that is the \textit{predecessor} of cell to be deleted (i.e., the cell containing x)
    ▪ Keep two cursors, \textit{scout} and \textit{current}
    ▪ \textit{scout} is always one cell ahead of \textit{current}
    ▪ Stop when \textit{scout} finds cell containing x, or falls off end of list
  ▪ If \textit{scout} finds cell, update \textit{next} field of \textit{current} cell to splice out object x from list
• Note: Need special case for x in first cell

delete 36 from list
Iterative Delete

- Two steps:
  - Locate cell that is the predecessor of cell to be deleted (i.e., the cell containing x)
    - Keep two cursors, scout and current
    - scout is always one cell ahead of current
    - Stop when scout finds cell containing x, or falls off end of list
  - If scout finds cell, update next field of current cell to splice out object x from list
- Note: Need special case for x in first cell

Delete 36 from list
Iterative Delete

- Two steps:
  - Locate cell that is the predecessor of cell to be deleted (i.e., the cell containing x)
    - Keep two cursors, scout and current
    - scout is always one cell ahead of current
    - Stop when scout finds cell containing x, or falls off end of list
  - If scout finds cell, update next field of current cell to splice out object x from list
- Note: Need special case for x in first cell

Delete 36 from list
Iterative Code for Delete

```java
public void delete (Object x) {
    if (head == null) return;
    if (head.getDatum().equals(x)) {  //x in first cell?
        head = head.getNext();
        return;
    }

    ListCell current = head;
    ListCell scout = head.getNext();
    while ((scout != null) && !scout.getDatum().equals(x)) {
        current = scout;
        scout = scout.getNext();
    }
    if (scout != null) current.setNext(scout.getNext());
    return;
}
```
Doubly-Linked Lists

• In some applications, it is convenient to have a **ListCell** that has references to both its predecessor and its successor in the list.

class DLLCell {
    private Object datum;
    private DLLCell next;
    private DLLCell prev;
    ...
}
Doubly-Linked vs Singly-Linked

• Advantages of doubly-linked over singly-linked lists
  ▪ some things are easier – e.g., reversing a doubly-linked list can be done simply by swapping the previous and next fields of each cell
  ▪ don't need the scout to delete

• Disadvantages
  ▪ doubly-linked lists require twice as much space
  ▪ insert and delete take more time
Java ArrayList

• “Extensible array”
• Starts with an initial $capacity = \text{size of underlying array}$
• If you try to insert an element beyond the end of the array, it will allocate a new (larger) array, copy everything over invisibly
  ▪ Appears infinitely extensible

• Advantages:
  ▪ random access in constant time
  ▪ dynamically extensible

• Disadvantages:
  ▪ Allocation, copying overhead
Tree Overview

- **Tree**: recursive data structure (similar to list)
  - Each cell may have two or more successors (or children)
  - Each cell has at most one predecessor (or parent)
    - Distinguished cell called *root* has no parent
  - All cells reachable from *root*
- **Binary tree**: tree in which each cell can have at most two children: a left child and a right child

![General tree](image)

![Binary tree](image)

![Not a tree](image)

![List-like tree](image)