Lists & Trees

Lecture 8
CS211 – Fall 2005

List Overview

- Arrays
  - Random access: :)
  - Fixed size: cannot grow on demand after creation: : >((
- Characteristics of some applications:
  - Do not need random access
  - Require a data structure that can grow and shrink dynamically to accommodate different amounts of data

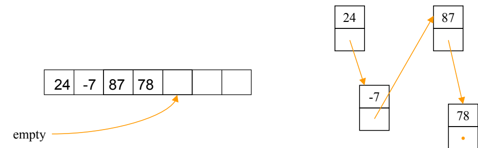
  Lists satisfy these requirements
- Let us study
  - List creation
  - Accessing elements in a list
  - Inserting elements into a list
  - Deleting elements from a list

List Operations

- ADT (Abstract Data Type):
  - Specify public functionality
  - Hide implementation detail from users
  - Allows us to improve/replace implementation
  - Forces us to think about fundamental operations Interface) separately from the implementation
- List Operations:
  - Create
  - Insert object
  - Delete object
  - Find object
  - Get Length, Full?, Empty?, Replace Object, …
  - Usually sequential access (not random access)

List Data Structures

- Can use an array
  - Need to specify array size
  - Inserts & Deletes require moving elements
  - Must copy array (to a larger array) when it gets full
- Can use a sequence of linked cells
  - We'll focus on this kind of implementation
  - We define a class ListCell from which we build lists



Class ListCell

```
class ListCell {
   private Object datum;
   private ListCell next;

   public ListCell(Object o, ListCell n){
      datum = o;
      next = n;
   }
   public Object getDatum() {//sometimes called car
      return datum;
   }
   public ListCell getNext() {//sometimes called cdr, tail, rest
      return next;
    }
   public void setDatum(Object o) {//sometimes called rplaca
      datum = o;
   }
   public void setNext(ListCell l) {//sometimes called rplacd
       next = l;
   }
}
```

ListCell

datum Object:

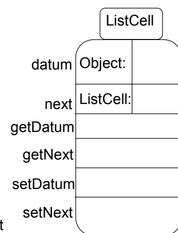
next ListCell:

getDatum

getNext

setDatum

setNext



By convention, we will not show the instance methods when drawing cells.

Building a List

l ListCell:

ListCell l = new ListCell(new Integer(24), null);

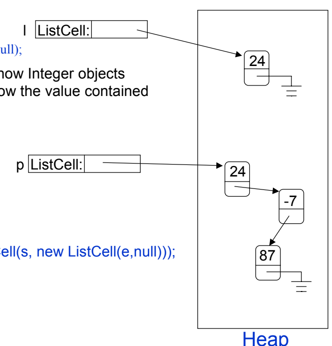
To keep things simple, we will not show Integer objects explicitly in our pictures, but only show the value contained in them.

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

p ListCell:

One way:

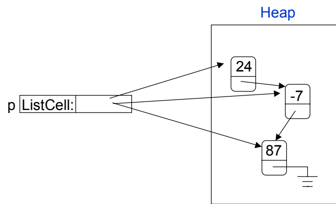
ListCell p = new ListCell(t, new ListCell(s, new ListCell(e,null)));



Building a List (cont'd)

Another way:

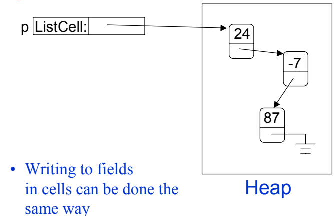
```
Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);

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



Note: assignment of form p = new ListCell(s,p); does *not* create a circular list.

Accessing List Elements

- Lists are sequential-access data structures.
  - to access the 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
  - p.setDatum(new Integer(53));//update data field of first cell
  - p.getNext().setDatum(new Integer(53));//update field of second cell
  - p.getNext().setNext(null);//chop off third cell

Access Example: Linear Search

```
//scan list looking for object o and return true if found
public static boolean search(Object x, ListCell l) {
      for (ListCell current = l; current != null; current = current.getNext())
          if (current.getDatum().equals(x)) return true;
      return false;
 }
 …..
 ListCell p = new ListCell("hello", new ListCell("dolly", new ListCell("polly", null)));
 search("dolly", p); //returns true
 search("molly", p); //returns false
 search("dolly", null); //returns false
….
//Here is another version. Why does this work? Draw stack picture to understand.
public static boolean search(Object x, ListCell l) {
      for (; l != null; l =l.getNext())
          if (l.getDatum().equals(x)) return true;
      return false;
 }
```

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 (smaller) list obtained by eliminating first cell…
- Many list problems can be solved very simply by using this idea.
  - Some problems though are easier to solve iteratively.

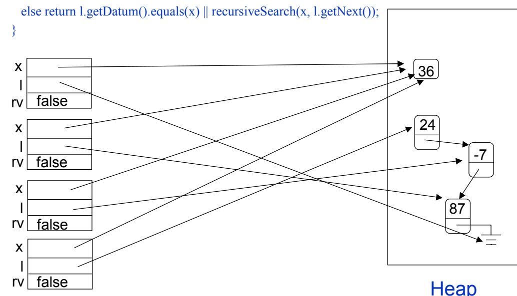
Recursion Example: Linear Search

- Base case: empty list
  - return false
- Recursive case: non-empty list
  - if data in first cell equals object o, return true
  - else return result of doing linear search on rest of list

```
public static boolean recursiveSearch(Object x, ListCell l) {
   if  (l == null) return false;
   else return l.getDatum().equals(x) || recursiveSearch(x, l.getNext());
}
```

Execution of Recursive Program

```
public static boolean recursiveSearch (Object x, ListCell l) {
   if  (l == null) return false;
   else return l.getDatum().equals(x) || recursiveSearch(x, l.getNext());
}
```



Iteration is Sometimes Better

- Given a list, create a new list with elements in reverse order from input list.

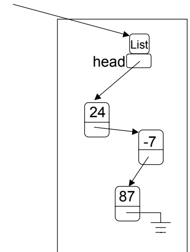
```
//intuition: think of reversing a pile of coins
public static ListCell reverse (ListCell l) {
   ListCell rev = null ;
   for (  ; l != null; l = l.getNext())
       rev = new ListCell(l.getDatum(), rev);
   return rev;
}
```

- It is not obvious how to write this simply in a recursive divide-and-conquer style.

List with Header

- Some authors prefer to have a List class that is distinct from ListCell class.
- List object is like a head element that always exists even if list itself is empty.

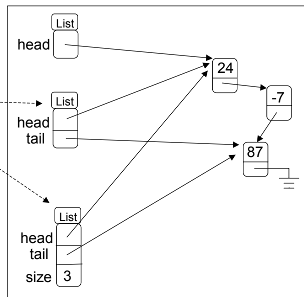
```
class List {
    protected ListCell head;
    public List (ListCell l) {
       head = l;
    }
    public ListCell getHead()
    ……….
  public void setHead(ListCell l)
   ……….
}
```



Heap

Variations of 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
  - …..



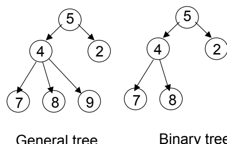
Heap

Special Cases to Worry About

- Empty list
  - add
  - find
  - delete?(!)
- Front of list
  - insert
- End of list
  - find
  - delete
- Lists with just one element

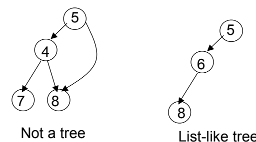
Tree Overview

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



General tree

Binary tree

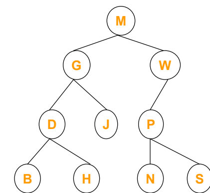


Not a tree

List-like tree

Tree Terminology

- M is the *root* of this *tree*
- G is the *root* of the left *subtree* of M
- B, H, J, N, and S are *leaves*
- N is the *left child* of P; S is the *right child*
- P is the *parent* of N
- M and G are *ancestors* of D
- P, N, and S are *descendents* of W
- Node J is at *depth* 2 (i.e., *depth* = length of path from root)
- Node W is at *height* 2 (i.e., *height* = length of *longest* path from leaf)
- A collection of several trees is called a *??*

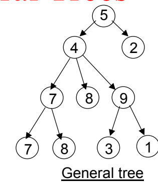


Class for Binary Tree Cells

```
class TreeCell {
    private Object datum;
    private TreeCell left;
    private TreeCell right;

    public TreeCell (Object x) {
     datum = x;
    }
    public TreeCell (Object x, TreeCell l, TreeCell r) {
       datum = x;
       left = l;
       right = r;
    }
    methods called getDatum, setDatum,
    getLeft, setLeft, getRight, setRight
    with obvious code
}
```

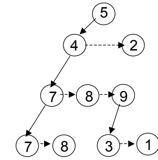
Class for General Trees

```
class GTreeCell{
  private Object datum;
  private GTreeCell left;
  private GTreeCell sibling;
  ….appropriate getter and setter
   methods
}
```



General tree

- Parent node points directly only to its leftmost child
- Leftmost child has pointer to next sibling which points to next sibling, etc.



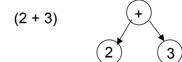
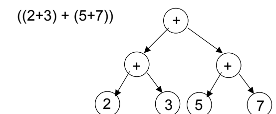
Tree represented using GTreeCell

Applications of Trees

- Most languages (natural and computer) have a recursive, hierarchical structure
- This structure is implicit in ordinary textual representation
- Recursive structure can be made explicit by representing sentences in the language as trees: *Abstract Syntax Trees* (ASTs)
- ASTs are easier to optimize, generate code from, etc. than textual representation
- Converting textual representations to AST: job of parser!

Example

- Expression grammar:
  - $E \to$ integer
  - $E \to (E + E)$
- In textual representation
  - Parentheses show hierarchical structure
- In tree representation
  - Hierarchy is explicit in the structure of the tree.

Text	Tree representation
-34	-34
(2 + 3)	
((2+3) + (5+7))	

Recursion on trees

- Recursive methods can be written to operate on trees in the obvious way
- In most problems
  - Base case: empty tree
    - Sometimes base case is leaf node
  - Recursive case: solve problem on left and right sub-trees then put solutions together to compute solution for full tree

Example: Delete from a List

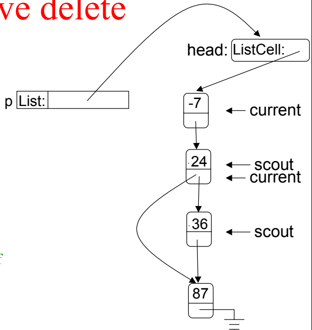
- Delete first occurrence of object x from list l
  - Recursive delete
  - Iterative delete
- Intuitive idea of recursive code
  - If list l is empty, return null
  - If first element of l is x, return rest of list l
  - Otherwise, return list consisting of
    - First element of l, and
    - List that results from deleting x from rest of list l

Recursive Code for Delete

```
public static ListCell deleteRecursive (Object x, ListCell l) {
    // If list is empty, nothing to do
    if (l == null) return null;
    // Otherwise check first element of list
    else if (l.getDatum().equals(x))
          return l.getNext();
    // Otherwise delete x from rest of list and update next field of l
    else  {l.setNext(deleteRecursive(x, l.getNext()));
            return l;
          }
  }
}
```

Iterative delete

- Two steps:
  - Locate cell that is the predecessor of cell to be deleted
    - Keep two cursors, scout and current, that traverse the list in lock step
    - Scout is always one cell ahead of current
    - Current starts at head of list
    - Stop when scout finds cell containing x, or falls off end of list
  - If scout finds cell, update next field of current cell to next field of scout cell to splice out object x from list



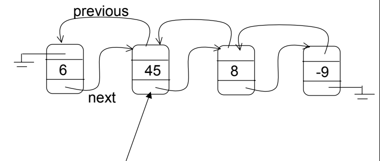
Iterative Code for Delete

```
public void delete (Object x) {
  // Empty list?
  if (head == null) return;
  // Is first element equal to x; if so splice first cell out
  if (head.getDatum().equals(x)) {
    head = head.getNext();
    return;
    }
  // Walk down list; at end of loop, scout will point to first cell containing x, if any
  ListCell current = head;
  ListCell scout = head.getNext();
  while ((scout != null) && ! scout.getDatum().equals(x)) {
      current = scout;
      scout = scout.getNext();
   }
   if (scout != null) // Found occurrence of x
      current.setNext(scout.getNext()); // Splice out cell containing o
}
```

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 previous;
   …..
}
```



Doubly-Linked vs. Singly-Linked

- It is often easier to work with doubly-linked lists than with (singly-linked) lists
  - For example, reversing a DLL can be done simply by swapping the previous and next fields of each cell
- Trade-off: DLLs require more heap space than singly-linked lists

Fancy Lists

- 2-D lists:
  - References to cells left, right, up, down
- 3-D lists, …
- Rings, pipes, torus lists
- Lists of lists (nested lists)
  - ((This is a sentence.)
    (This is a sentence, too.)
    (This is another sentence.)
    …)

List Summary

- Lists are sequences of ListCell elements
  - Recursive data structure
  - Grow and shrink on demand
  - Not random-access but sequential access data structures
- List operations
  - Create a list
  - Access a list and update data
  - Change structure of list by inserting/deleting cells
    - Cursors
- Recursion makes perfect sense on lists. Usually
  - Base case: empty list
  - Recursive case: non-empty list
- Subspecies of lists
  - List with header
  - Doubly-linked lists

Tree Summary

- A *tree* is a recursive data structure built from TreeCell elements
  - Special case: binary tree
- Binary tree cells have both a left and a right "successor"
  - Called children rather than successors
  - Similarly, parent rather than predecessor
  - Generalization of parent and child to ancestors and descendants
- Trees are useful for exposing the recursive structure of natural language programs and computer programs

LISP

- List languages first developed for AI
- LISP: List Processing Language
  - Developed in 50-60's by John McCarthy, et al.
- Lists and list processing are a fundamental part of LISP language
  - Lists are primitive data type
  - Functions operate directly on lists
  - Program itself expressed as list of lists
- "car": contents address register (getDatum())
- "cdr": contents decrement register (getNext())
- "caddr" = (car (cdr (cdr list))) = object in 3rd element