Lists

History of List Processing in CS

- List languages first developed for AI
- LISP: List Processing Language
  - Developed in 50-60's by John McCarthy et al.
- LL: List Language
  - Developed in 50's by Allen Newell and Herb Simon
- Lists and list processing fundamental part of 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

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 this requirement.
- 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

- Implemented using arrays
  - Size of array
  - Number of elements in list
  - Inserts & Deletes require moving elements
  - Must copy array when it gets full
- Implemented using Java Vectors
  - `import java.util.Vector (or java.util.*)`
  - Size automatically expands as necessary
  - Automatically maintains number of elements
  - Inserts & Deletes still require moving elements
- Implemented as sequence of linked cells
  - We'll focus on this kind of implementation

Class ListCell

```java
class ListCell {
    protected Object datum;
    protected 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;
    }
}
```

Lists

- List is a sequence of cells in which each cell contains
  - a data item of type Object
  - a reference to the next cell in the sequence
    - null if this is the last cell in the sequence
  - empty list: null
- List is a sequential-access data structure
  - to access data in position \( n \) of sequence, we must access cells \( 0, \ldots, n-1 \)
- We will define a class called ListCell from which we will build lists.

Building a list

```java
ListCell l = new ListCell( new Integer(24), null);
ListCell p = new ListCell(t, new ListCell(s, new ListCell(e, null)));
```

By convention, we will not show the instance methods when drawing cells.
Building a list (contd.)

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
• 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 o, ListCell l) {
  for (ListCell current = l; current != null; current = current.getNext())
    if (current.getDatum().equals(o)) return true;
  //drop out of loop if object not found
  return false;
}

Access example: linear search
//scan list looking for object o and return true if found
public static boolean search(Object o, ListCell l) {
  for (ListCell current = l; current != null; current = current.getNext())
    if (current.getDatum().equals(o)) return true;
  //drop out of loop if object not found
  return false;
}

Recursion on lists

• Recursion can be done on lists
  – similar to recursion on integers
• Almost always
  – base case: empty list
  – recursive case: assuming you can solve problem on
(smaller) list obtained by eliminating first cell, write
down solution for list
• 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

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

Special Cases to Worry About

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

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

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
  - …..

Example of use of List class

- Let us write code to
  - insert object into unsorted list
  - delete the first occurrence of an object in an unsorted list.
- We will use the List class to show how to use this class.
  - It is just as easy to write code without the header element.
- Methods for insertion/deletion will be instance methods in the List class.
- signatures:
  - public void insert(Object o);
  - public void delete(Object o);
- invocation:
  - p.insert(o); p.delete(o);

Insertion into list

- Let us write two insert methods
  - insert at head of list
    ```java
    public void insertHead(Object o) {
        head = new ListCell(o, head);
    }
    ```
  - insert at tail of list
    ```java
    public void insertTail(Object o) {
        if (head == null) {
            head = new ListCell(o, null);
        } else {
            ListCell current = head;
            while (current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(new ListCell(o, null));
        }
    }
    ```
- invocation:
  - p.insertHead(new Integer(54));
  - p.insertTail(new Integer(54));
**Example of use of insert methods**

```
List p = new List(null); // create List object with empty list
p.insertHead(new Integer(-7)); // list now contains -7
p.insertHead(new Integer(24)); // list contains 24 and -7
p.insertTail(new Integer(87));
```

```
// extract first element of list
public Object deleteFirst(){
    // if list is not empty
    if (head != null) {
        Object t = head.getDatum();
        head = head.getNext();
        return t;
    }
    // otherwise, attempt to get from empty list
    else return "error";
}
```

```
// extract last element of list
public Object deleteLast(){
    if (head == null) return "error";
    // only one element in list?
    if (head.getNext() == null) {
        Object t = head.getDatum();
        head = null;
        return t;
    }
    // at least two elements in list
    // current and scout are cursors into list
    // both advance in lock step, scout is one cell ahead
    // stop if scout points to last cell
    ListCell current = head;
    ListCell scout = head.getNext();
    while (scout.getNext() != null){
        current = scout;
        scout = scout.getNext();
    }
    current.setNext(null); // remove last cell from list
    return scout.getDatum();
}
```

**Remove first item from list**

**Remove last item on list**

**Delete object from list**

- Delete first occurrence of object o 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 o, return rest of list l.
  - Otherwise, return list consisting of first element of l, and list that results from deleting o from rest of list l.
Recursive code for delete

```java
class List{
    protected ListCell head;
    .......
    public void delete(Object o) {
        head = deleteRecursive(o, head);
    }
    public static ListCell deleteRecursive(Object o, ListCell l) {
        if (l == null) return null;
        else if (l.getDatum().equals(o))
            return l.getNext();
        else
            return (new ListCell(l)).setNext(deleteRecursive(o, l.getNext()));
    }
}
```

Iterative delete

- Two steps:
  - locate cell that is the predecessor of cell to be deleted
  - if scout finds cell, update next field of current cell to next field of new cell

Iterative code for delete

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

Insertion and deletion into sorted lists

- Assume that we have a list of Comparables sorted in increasing order.
- We want to splice a new Comparable into this list, keeping new list in sorted order as shown in figure.
- Code shows recursive code for insertion and deletion.
- We will show code that uses ListCell class directly.
**Recursive insertion**

Let us use notation \([f,n]\) to denote ListCell whose
- datum is \(f\)
- next is \(n\)

**Pseudo-code:**

insert (Comparable \(c\), ListCell \(l\)):
- if \(l\) is null return new ListCell\((c,\text{null})\);
- else
  - suppose \(l\) is \([f,n]\)
  - if \((c < f)\) return new ListCell\((c,l)\);
  - else return new ListCell\((f,\text{insert}(c,n))\);

**Compactify:**
- \(\text{insert}(c,\text{null}) = [c,\text{null}]\)
- \(\text{insert}(c,[f,n]) = \begin{cases} [c,[f,n]] & \text{if } c < f \\ [f,\text{insert}(c,n)] & \text{if } c \geq f \end{cases}\)

---

**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.

```java
class DLLCell {
    protected Object datum;
    protected DLLCell next;
    protected DLLCell previous;
    ...
}
```
In general, it is easier to work with doubly-linked lists than with 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.)
  ...

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
- Sub-species of lists
  - list with header
  - doubly-linked lists