List ADT

- What is a List?
  ‣ Ordered sequence of elements $A_1, A_2, \ldots, A_N$
- Elements may be of arbitrary type, but all are of the same type
- Common List operations are:
  ‣ Insert, Find, Delete, IsEmpty, IsLast, FindPrevious, First, Kth, Last, Print, etc.

Simple Examples of List Use

- Polynomials
  ‣ $25 + 4x^2 + 75x^{85}$
- Unbounded Integers
  ‣ 4576809099383658390187457649494578
- Text
  ‣ “This is an example of text”
List Implementations

- Two types of implementation:
  - Array-Based
  - Pointer-Based

List: Array Implementation

- Basic Idea:
  - Pre-allocate a big array of size MAX_SIZE
  - Keep track of current size using a variable count
  - Shift elements when you have to insert or delete

Array List Insert Running Time

- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).
List: Pointer Implementation

- Basic Idea:
  - Allocate little blocks of memory (nodes) as elements are added to the list
  - Keep track of list by linking the nodes together
  - Change links when you want to insert or delete

Insertion After

InsertAfter(p : node pointer, v : value_type): {
  x : node pointer;
  x := new node;
  x.value := v;
  x.next := p.next;
  p.next := x;
}

Note: cannot swap two last lines (why?)

Linked List with Header Node

Advantage: “insert after” and “delete after” can be done at the beginning of the list.
**Pointer Implementation Issues**

- Whenever you break a list, your code should fix the list up as soon as possible
  - Draw pictures of the list to visualize what needs to be done
- Pay special attention to boundary conditions:
  - Empty list
  - Single item – same item is both first and last
  - Two items – first, last, but no middle items
  - Three or more items – first, last, and middle items

**Pointer List Insert Running Time**

- Running time for a list with N elements?
- Insert takes constant time (O(1))
- Does not depend on list size
- Compare to array based list which is O(N)

**Linked List Delete**

To delete the node pointed to by Q, need a pointer to the previous node; See book for findPrevious method

**Delete After**

DeleteAfter(p : node pointer): {
  temp : node pointer;
  temp = p.next;
  p.next = temp.next; //p.next.next
  free(temp);
}

Note: p points to the node that comes before the deleted node!

temp – the node to be removed.
Linked List Delete

Doubly Linked Lists

- findPrevious (and hence Delete) is slow \([O(N)]\) because we cannot go directly to previous node
- Solution: Keep a "previous" pointer at each node

Double Link Pros and Cons

- Advantage
  〉 Delete (not DeleteAfter) and FindPrev are faster
- Disadvantages:
  〉 More space used up (double the number of pointers at each node)
  〉 More book-keeping for updating the two pointers at each node (pretty negligible overhead)

Reverse a linked list

```c
Reverse(t : node pointer): node pointer {
    rev : node pointer;
    temp : node pointer;
    rev = NULL;
    while(t != NULL){
        temp = t.next;
        t.next = rev;
        rev = t;
        t = temp;
    }
    return (rev);
}
```

Why do we need temp?
Unbounded Integers Base 10

-4572

\[ \begin{array}{cccccc}
10^3 & 10^2 & 10^1 & 10^0 & \text{sign} \\
4 & 5 & 7 & 2 & -1 \\
\end{array} \]

X : node pointer

348

\[ \begin{array}{cccccc}
10^2 & 10^1 & 10^0 & \text{sign} \\
3 & 4 & 8 & 1 \\
\end{array} \]

Y : node pointer

Recursive Addition

- Positive numbers (or negative numbers)

\[ \begin{array}{c}
3427 \\
+898 \\
\end{array} \]

\[ \begin{array}{c}
7 \\
+8 \\
\end{array} \]

\[ \begin{array}{c}
5 \\
+1 \\
\end{array} \]

Recursive calls

Zero

\[ \begin{array}{c}
nul \\
-1 \\
\end{array} \]

Recursive Addition

- Mixed numbers

\[ \begin{array}{c}
3427 \\
-898 \\
\end{array} \]

\[ \begin{array}{c}
7 \\
-8 \\
\end{array} \]

\[ \begin{array}{c}
9 \\
-1 \\
\end{array} \]

Recursive calls
Example

- Mixed numbers

\[
\begin{array}{cccccc}
100000 & -999999 \\
0 & -9 \\
1 & -1 \\
-10 & -1 \\
0 & -9
\end{array}
\]

Recursive calls

Alternative Addition

- Use an auxiliary function
  > AddAux(p,q : node pointer, cb : integer) which returns the result of adding p and q and the carry/borrow cb.
  > Add(p,q) := AddAux(p,q,0)
  > Advantage: more like what we learned in school (and more like actual binary adders in hardware).

Auxiliary Addition

- Positive numbers

\[
\begin{array}{cccccc}
0 & 7 & 0 & 342 & 7 \\
3427 & +898 & +8 & 342 & +89
\end{array}
\]

Recursive call

Auxiliary Addition

- Mixed numbers

\[
\begin{array}{cccccc}
0 & 7 & 0 & 342 & 7 \\
3427 & -898 & -8 & 342 & -8
\end{array}
\]

Recursive call
Copy

- Design a recursive algorithm to make a copy of a linked list (like the one used for long integers)

  Copy(p : node pointer) : node pointer {
    ?
  }

  next value

  node

Comparing Integers

IsZero(p : node pointer) : boolean {
  //p points to the sign node
  return p.next = null;
}

IsPositive(p: node pointer) : boolean {
  //p points to the sign node
  return not IsZero(p) and p.value = 1;
}

Negate(p : node pointer) : node pointer {
  //destructive
  if p.value = 1 then p.value := -1
  else p.value := 1;
  return p;
}

LessThan(p,q : node pointer) : boolean {
  // non destructive
  p1,q1 : node pointer;
  p1 := Copy(p); q1 := Copy(q);
  return IsPositive(Add(q1,Negate(p1)));
  // x < y iff 0 < y - x
  // We assume Add and Negate are destructive
}

List Mergesort

- Overall sorting plan

  sort
  split into equal size lists
  sort recursively
  merge into one sorted list

Mergesort pseudocode

Mergesort(p : node pointer) : node pointer {
  Case {
    p = null : return p; //no elements
    p.next = null : return p; //one element
    else
      d : duo pointer; // duo has two fields first,second
      d := Split(p);
      return Merge(Mergesort(d.first),Mergesort(d.second));
  }
}

Note: Mergesort is destructive.
Split

Split(p : node pointer) : duo pointer {
    d : duo pointer;
    Case {
        p = null : d := new duo; return d // both fields are null
        p.next = null : d := new duo; d.first := p ; return d
            // d.second is null
        else :
            d := Split(p.next.next);
            p.next.next := d.first;
            d.first := p.next;
            p.next := d.second;
            d.second := p;
            return d;
    }
}

Split Example

After recursive call to Split

After recursive call to Split

After recursive call to Split

After recursive call to Split
Merge

Merge(p, q : node pointer): node pointer{
  case {
    p = null : return q;
    q = null : return p;
    LessThan(p.value, q.value) :
      p.next := Merge(p.next, q);
      return p;
    else :
      q.next := Merge(p, q.next);
      return q;
  }
}

Merge Example

merge

merge return

merge return

merge return
Implementing Pointers in Arrays
– “Cursor Implementation”

- This is needed in languages like Fortran, Basic, and assembly language
- Easiest when number of records is known ahead of time.
- Each record field of a basic type is associated with an array.
- A pointer field is an unsigned integer indicating an array index.

Idea

<table>
<thead>
<tr>
<th>Pointer World</th>
<th>Nonpointer World</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D[ ]</strong> : basic type array</td>
<td><strong>D[ ]</strong> : basic type array</td>
</tr>
<tr>
<td><strong>N[ ]</strong> : integer array</td>
<td><strong>N[ ]</strong> : integer array</td>
</tr>
<tr>
<td>Pointer is an integer</td>
<td>Pointer is an integer</td>
</tr>
<tr>
<td>null is 0</td>
<td>null is 0</td>
</tr>
<tr>
<td>p.data is D[p]</td>
<td>p.data is D[p]</td>
</tr>
<tr>
<td>p.next is N[p]</td>
<td>p.next is N[p]</td>
</tr>
<tr>
<td>Free list needed for node allocation</td>
<td></td>
</tr>
</tbody>
</table>

Initialization

Example of Use

```c
InsertFront(L : integer, x : basic type) {
    q : integer;
    if not(Free = 0) then q := Free
    else return "overflow";
    Free := N[Free];
    D[q] := x;
    N[q] := L;
    L := q;
}
```
Try DeleteFront

• Define the cursor implementation of DeleteFront which removes the first member of the list when there is one.
  › Remember to add garbage to free list.

```java
DeleteFront(L : integer) {
???
}
```

DeleteFront Solution

```java
DeleteFront(L : integer) {
  q : integer;
  if L = 0 then return "underflow"
  else {
    q := L;
    L := N[L];
    N[q] := Free;
    Free := q;
  }
}
```

Copy Solution

```java
Copy(p : node pointer) : node pointer {
  if p = null then return null
  else {
    q : node pointer;
    q := new node; //by convention the value
    //field is 0 and the
    //pointer field is null
    q.value := p.value;
    q.next := Copy(p.next);
    return q;
  }
}
```