More on Lists

CSE 373
Data Structures
Lecture 4
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 grade school (and more like actual binary adders in hardware).
Auxiliary Addition

- Positive numbers (or negative numbers)

0

3427

+898

0

7

+8

5

10

1

342

+89

Recursive call
Auxiliary Addition

- Mixed numbers

\[
\begin{array}{c}
3427 \\
-898 \\
\hline
2529
\end{array}
\]

\[
\begin{array}{c}
0 \\
0 \\
\hline
0
\end{array}
\]

\[
\begin{array}{c}
7 \\
-8 \\
\hline
-1
\end{array}
\]

\[
\begin{array}{c}
342 \\
-89 \\
\hline
253
\end{array}
\]

Recursive call
Copy

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

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

node

next value
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

sort recursively
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

1st call to split
2nd call to split
Last call to split
Split Example

After recursive call to Split
Split Example
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

\[ \text{null} \]

\[ \text{null} \]
Merge Example
Merge Example

merge

merge return

4 9

20 30

null

3 13 17 19

q
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>n nodes</td>
<td>D</td>
</tr>
<tr>
<td>data next</td>
<td>N</td>
</tr>
<tr>
<td>data : basic type</td>
<td>1</td>
</tr>
<tr>
<td>next : node pointer</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
</tbody>
</table>
|               | n                |• D[ ] : basic type array
|               |                  |• N[ ] : integer array
|               |                  |• Pointer is an integer
|               |                  |• null is 0
|               |                  |• p.data is D[p]
|               |                  |• p.next is N[p]
|               |                  |• Free list needed for node allocation
Initialization

Free = n

```
1  |  D   | N   |
---|------|-----|
1  |      | 0   |
2  |      | 1   |
3  |      | 2   |
4  |      | 3   |
5  |      | 4   |
.  |      |     |
n  |      | n-1 |

D     NFree = n
1  |      |     |
2  |      |     |
3  |      |     |
4  |      |     |
5  |      |     |
.  |      |     |
n  |      |     |
```

null

means

Free
Example of Use

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.

```c
DeleteFront(L : integer) {
  ???
}
```
Copy Solution

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;
    }
}
DeleteFront Solution

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