CS211 Recitation: Points about link lists
6-8 April 2004

This recitation should round out your knowledge of linked lists. You have seen linked lists and linked lists with headers. We mentioned that for some applications, the header of a linked list could have a head and a tail pointed. And you saw *briefly*, only, doubly linked lists. It is time to give them some applications that use linked and a few other details.

Hashing.
You know about hashing. In the way we first did hashing, all the hashed items are in the array itself, and a new array is created, with twice the size, if the load factor gets too big.
2. Another way to handle collisions is to use "separate chaining hashing" instead of linear or quadratic probing. Each element of b[k] of the hash array is a linked list of items that hashed to the integer k. So, the basic algorithm is:

```
// Add s to this set
public void add(String s) {
int k= hashCode(s);
if (b[k] == null) {
      b[k]= (a new linked list with
      one value, s);
      return;
}
if (linked list b[k] contains s)
      return;
Add s to the beginning of linked list b[k];
}
```

Weiss, in section 20.5, discusses separate chaining hashing. Suppose the load factor lf = N/M, where N is the number of items in the hash set and M is the size of the array. Note that lf can be bigger than 1. Then, the average linked list has length lf, so a successful search for an item takes an average of 1+lf/2 probes. A hash set could have 2000 items in it, in a 1000-element array, and we would expect to make 1.5 probes to find an item that is in it.

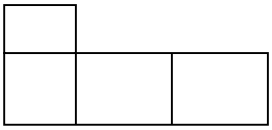
When using this technique, one does not re-hash at all. The programming is extremely easy.

Doubly linked lists

Suppose we use the following for the node of a doubly-linked list ll:

```
public class LLnode {
private int value;  // the item that this node
                    // contains
private LLnode prev;  // the node that contains
          // the previous item (null if none)
private LLnode next;  // the node that contains
                  // the next item (null if none)
    }

/* Constructor: node with value it, previous field
  p, and next field n */
public LLnode(int it, LLnode p, LLnode n)
```



The above is how we might draw a manilla folder for a node of a doubly linked list like this. Note: In the powerpoint slides for the lecture on linked list, slide 33 used class DLLCell instead of LLNode. Weiss uses the name DoublyLinked-ListNode for this class. The name doesn't matter here; the concept does.

We can develop methods for inserting a node and deleting a node (shown below).The way to develop the bodies is to first draw the initial state, then draw the final state, and then to develop the code. Note that one must always be sure that a value is not null before using it to reference a component of an object.

```
// Delete node v from its linked list
public void delete(LLNode v) {
    if (v.prev != null) v.prev.next= v.next;
    if (v.next != null) v.next.prev= v.prev;
}

// Add item it after node v
public void add(int it, LList v) {
    Llist ll= new Llist( it, v, v.next);
    if (v.next != null) v.next= ll;
    if (ll.next != null) ll.next.prev= ll;
}
```

Circular doubly linked list

Let b and e be the beginning and end nodes of a doubly linked list. If we change b.prev to e and e.next to b, we have a circular list. It doesn't matter which node is first on the list. It is circular.

Circular lists are useful when the order of the values of a list doesn't matter.

Here are two examples of the uses of a circular list:

1. Use a circular doubly linked list to maintain the points that define a convex hull of a set

of points in the plane. These would be the points of a convex polygon.

2. Josephus problem. Josephus Flavius was a famous Jewish historian of the first century at the time of the Second Temple destruction. During the Jewish-Roman war he got trapped in a cave with a group of 39 soldiers surrounded by Romans. The legend has it that preferring suicide to capture, the Jews decided to form a circle and, proceeding clockwise around it, to kill every seventh person until only one was left, who must then commit suicide. Josephus, an accomplished mathematician, quickly found the safe spot in the circle (24th) to be the last to go. But when the time came, instead of killing himself he joined the Roman side. The problem rightfully raises the question of how someone might be able to quickly compute the correct place to stand.

So, use a circular linked list with 39 items, with some variable v containing the name of one of the nodes of the list. Now write the algorithm that, iteratively, kills every seventh person (removing them from the circl) until one is left.

BigInts

Types **int** and **long** deal only with a finite set of the integers. We now look at implementing a type integer, which contains all the integers, in a class BigInt. This will provides a more thorough understanding of representing integers in different bases.

Each instance of BigInt contains an instance of class List, which contains the unsigned integer.

Maintaining an integer in some base. Let base b be an integer, 2 <= b. Then a positive integer n can be written as

$$n = n_0*b^0 + n_1*b^1 + \ldots + n_{k-1}*b^{k-1} + n_k*b^k$$

where:

(0) Each n_i satisfies $0 <= n_i < b$
(1) The high-order digit n_k is not 0

The integer 0 is represented by $0*b^0$, or 0.

For example, the integer 341 in decimal is

$$341 = 1*10^0 + 4*10^1 + 3*b^2$$

Here, k is 2.

Comparing integers. Given the base b representations of nonnegative m and n, the expression n < m can be evaluated as follows. If one is longer than the other, the shorter one is smaller. Otherwise, compare n_0 and m_0 , then n_1 and m_1, then n_2 and m_2 , etc.; at each step, maintain a variable d that is -1, 0, or 1, depending on whether the part of n that has been compared thus far is less than, equal to, or greater than the part of m that has been compared.

Addition. Addition in base b is just like addition in base 10, e.g. we evaluate 5432+649 as shown below, where the top line represents the carry from the previous (lower-order) digit.

```
  0 1 0 1      (this is the carry)
    5 4 3 2
+     6 4 9    DONE IN BASE 10
  ---------
    6 0 8 1
```

Thus, 2+9 = 11, which is treated as 1 with a carry of 1. Then, 1+3+4 = 8, which is treated as 8 with a carry of 0. Then, 0+4+6 = 10, which is treated as 0 with a carry of 1. Then, 1+5=6, which is treated as 6 with a carry of 0.

Note that the digits are processed from low order to high order.

You should tru carrying this out in base b arithmetic, for some other b besides 10. do it by hand, for b – 2 or 8. When carrying this out in base b, suppose the carry plus two digits sums to s. Then the value for that position is s%b and the carry is s/b.

Subtraction. Subtraction of nonegative integers is similar. Here, we always subtract the larger integer from the smaller, so that a nonnegative integer results. This requires comparing the integers before doing the subtraction. And, when subtracting, there may be a "carry" of -1, as shown below.

```
  -1 -1 -1     (this is the carry)
    5  4  3 2
-     6 2 9    DONE IN BASE 10
  ---------
    4 7 0 3
```

For a particular position, suppose the carry plus the first digit minus the second digit is s. If s>=0, then the value for that position is s and the carry is 0. If s<0, then the value for that position is s+b (where b is the base) and the carry is -1.

The carry from the high-order position is always 0, because the smaller integer is subtracted from the larger one.

There is often a need to eliminate leading 0's. For example, using the scheme above, 5000-4999 = 0001, and this has to be changed to 1.

```
/** An instance wraps an integer (of any size) */
public class BigInt {
/* An integer is maintained in base BASE in
field bigint; its sign is in variable sign. The ele-
ments in bigint are always of class Integer. An
unsigned integer

          i = bk*BASE^k + ... + b2*BASE^2 +
              b1*BASE^1 + b0*BASE^0

is maintained as the list (b0, b1, b2, ..., bk), so
that the low-order digit is first and the high-order
digit is last.
     0 is represented by bigint = (0) andsign be-
ing true.
     If the integer is not 0, the high order digit is
nonzero.
     Note: in one or two places, integers are kept
in another base. This will be explained at the
right time.
*/
public List bigint
public boolean sign;   // sign of the integer

/** Constructor: the integer 0 */
public BigInt() {
     bigint= new List(int0,null);
     sign= true;
  }

Here are some methods of class BigInt.

/** = no. digits this integer requires in base
     BASE */
public int numberDigits()
    {  return List.length(this.bigint) }

/** = "this BigInt is less than b".
          Precondition: b != null */
public boolean less(BigInt b) {
     if (this.sign != b.sign)
        {  return b.sign;  }
     return this.sign ? less(this.bigint, b.bigint)
                      : less(b.bigint, this.bigint);
}

// = "unsigned int b1 is less than unsigned int b2"
//   Precondition: b1 and b2 are nonnull
private static boolean less(List b1, List b2) {
     List p1= b1;
     List p2= b2;
     int d= 0;
     /*inv: p1 is a node of b1 (or null when done)
            p2 is the corresponding node of b2 (or
               null when done)
          d = -1, 0, or 1 depending on whether the
          value before p1 is less, equal, or greater
           than the value before p2 */
     while (p1 != null && p2 != null) {
        int i1= ((Integer)p1.element).intValue();
        int i2= ((Integer)p2.element).intValue();
        if (i1 < i2) { d= -1; }
        else if (i1 > i2) { d= 1; }
        p1= p1.next; p2= p2.next;
     }
     // { at least one list is fully processed}
     if (p1 == null && p2 == null)
        {  return d < 0;  }
     // {one list was not fully processed. It is
     //  the larger of the two lists}
     return p1 == null;
  }

/** = the unsigned integer that is b1 + b2.
      Precondition: b1 and b2 are not null
   */
public static List add(List b1, List b2) {
  List p1= b1;
  List p2= b2;
  List res= new List(new Integer(0), null);
  List p3 = res;

  int carry=0; int nextCarry=0;

  /*inv: p1 is a node of b1 (or null when done).
         p2 is the corresponding node of b2 (or
              null when done).
         p3 is the last node of res.
         res is the sum of lists b1 and b2 up to
              and including nodes p1 and p2.
          carry is the carry.
    */

  while(p1!= null || p2 != null){
     nextCarry=0;
     Integer added= new Integer(0);
     int toAdd=0; //the next number to add to
                  // res in int form
     //case 1: only p2 is non-null
     if(p2 != null && p1 == null){
          toAdd= carry + ((Intger)p2.element).
                         intValue();
          if(toAdd>BASE && p2.next!= null){
             int temp = toAdd;
             toAdd=toAdd%BASE;
             nextCarry=(temp-toAdd)/BASE;
          }
          added = new Integer(toAdd);
     }
     //case 2: only p1 is non-null
     if(p1 !=null && p2==null){
          toAdd= carry + ((Integer)p1. element)
                          .intValue();
          if(toAdd>BASE && p1.next!=null){
```

```
            int temp = toAdd;
            toAdd= toAdd%BASE;
            nextCarry=(temp-toAdd)/BASE;
          }
          added = new Integer(toAdd);
      }
      //case 3: both p1 and p2 are non-null
      if(p1 != null && p2 != null){
          toAdd=carry + ((Integer)p1.element).
                              intValue() +
                  ((Integer)p2.element).
                              intValue();
          if(toAdd>=BASE &&
             (p1.next!= null || p2.next!= null)){
            int temp=toAdd;
            toAdd=toAdd%BASE;
            nextCarry=(temp-toAdd)/BASE;
          }
          if(toAdd>=BASE &&
            (p1.next==null && p2.next==null)){
            int temp=toAdd;
            toAdd=toAdd%BASE;
            nextCarry=(temp-toAdd)/BASE;
          }
          added = new Integer(toAdd);
      }

      //{added contains the value that should be
      //  placed in the solution,
      //  nextCarry contains the carry}
      carry=nextCarry;
      p3.next = new List(added, null);
      if (p1 !=null) p1=p1.next;
      if (p2 != null) p2=p2.next;
      p3 = p3.next;
      //{it is possible that we have reached the last
      //  node of both p1 and p2 and
      //  that our result will be longer than either of
      //  these. So we need to store
      //  carry, if it exists, in the next node}
      if(p1==null && p2==null && carry!=0){
          p3.next= new List(
                      new Integer(carry), null);
      }
    }
      //{due to the nature of the algorithm,
      //   res.element will always be empty.}
      res= res.next;
      return res;
}
```