I2A 98: Notes on implementation of B-trees.

The implementation of BTrees is tedious, nothing more. It's necessary for each level of the tree to be sensitive to splitting of its child on insertion, and shrinking of its child on deletion.

In the case of insertion, splitting can be indicated if the child returns a reference to the new subtree (that is, a null answer means no split).

I give an approximation to the encoding for those who like to read Java. Since Java doesn't allow us control over arrays (they are always references to 'array objects') we can't really do an implementation in Java, but we can pretend.

This code hasn't been tested, nor even compiled. It is shown simply to demonstrate, to those who can read it, that it *is* possible to cover all the various cases laid down for B-trees in Weiss and in the slides.

We need a wrapper class to use B-trees as dictionaries:

```
class RootBTree implements Dictionary {
  BTree t;
  public RootBTree () { t = new LeafBTree(); }
  public Data search(Key k) { return t.search(k); }
  public void insert(Key k, Data d) {
    BTree that = t.insert(k,d);
    if (that!=null) { // split the root
      InternalBTree t1 = new InternalBTree();
      t1.count=2; t1.key[0]=t.key[0]; t1.key[1]=t1.key[1];
      t1.subt[0]=t; t1.subt[1]=that;
      t=t1;
    }
  }
  public void delete(Key k) {
    t.delete(k);
    if (t.count==1 && t instanceof InternalBTree) // lift singleton child to root
      t=((InternalBTree)t).subt[0];
  }
}
```

To avoid too much code duplication, we make all B-tree objects contain <Key,Object> pairs. We define *find* to help in binary chop. We define a couple of useful methods – *shift* and *ins* – which help in insertion and deletion. We define *fromLeft* and *fromRight* methods to be used to move stuff into this node from a sibling.

```
abstract class BTree {
  protected int count; protected Key[] key; protected Object[] data;
  protected final int half;

  abstract public Data search(Key k);
  abstract public BTree insert(Key k, Data d);
  abstract public void delete(Key k);

  protected int find(Key k) { // for use only when count!=0
    for (int m=0, n=count; m+1!=n; ) {
      int i = (m+n)/2;
      if (key[i].lesseq(k)) m=i;
      else n=i;
    }
    return m;
  }

  protected void shift(int m, int dist) { // shift up or down by dist
    int inc, i0, i1;
    if (dist<0) i0=m, i1=count, inc=1; // m..count-1 down by dist
    else i0=count-1, i1=m-1, inc=-1;   // m..count-1 up by dist
    for (int i=i0; i!=i1; i+=inc) { key[i+dist]=key[i]; data[i+dist]=data[i]; }
    count+=dist;
  }

  private void ins(Key k, Object d, int m) {
    shift(m,1); key[m]=k; data[m]=d;
  }

  protected void fromLeft(BTree left, int n);
    shift(0,n); left.count-=n;
    for (int i=0, j=left.count; i<n; i++,j++) {
      key[i]=left.key[j]; data[i]=left.data[j];
    }
  }

  protected void fromRight(BTree right, int n); {
    for (int i=count, j=0; j!=n; i++,j++) {
      key[i]=right.key[i]; data[j]=right.data[j];
    }
    count+=n; right.shift(n,-n);
  }

}
```

We don't need an empty tree: a leaf which contains no <key,data> pairs will do.

In the Leaf class the *shift* and *ins* procedures are useful when inserting and deleting stuff.

```
class LeafBTree extends BTree {
  public LeafBTree() { count=0; key=new Key(L); data=new Data(L); }

  public Data search(Key k) {
    if (count==0) return null;
    else { int m=find(k); return k.equals(key[m]) ? (Data)data[m] : null; }
  }

  public BTree insert(Key k, Data d) {
    if (count==0) { count=1; key[0]=k; data[0]=d; return null; }
    else {
      int m=find(k);
      if (k.equals(key[m])) { data[m]=d; return null; }
      else {
        if (key[m].lesseq(k)) m++; // choose position for insertion
        if (count<L) { ins(k,d,m); return null; } // room for one more
        else { // split the node
          LeafBTree that = new LeafBTree();
          that.fromLeft(this,L/2);
          if (m<count) ins(k,d,m); else that.ins(k,d,m-count);
          return that;
        }
      }
    }
  }

  public void delete(Key k) {
    if (count!=0) {
      int m=find(k); if (k.equals(key[m])) shift(m+1, -1);
    }
  }

}
```

Insertion in an internal node is very like insertion in a leaf, except that you have to be careful to record the first key in your child. Deletion is very different, and involves moving child <Key,Object> pairs around to keep the half-full invariants and thus the high branching ratio which we need.

```
class InternalBTree extends BTree {
  public BTree[] subt; protected final int half = I/2;

  public InternalBTree { count=0; key=new Key(I); data=new BTree(I); }

  public Data search (Key k) { return ((BTree)data[find(k)]).search(k); }

  public BTree insert(Key k, Data d) {
    int m=find(k); BTree t0 = (BTree)data[m], that = t0.insert(k,d);
    key[m]=t0.key[0]; // child may have inserted at position 0
    if (that==null) return null; // child didn't split
    else
    if (count<I) { ins(that.key[0],that,m+1); return null; } // accept new child
    else { // split into two
      InternalBtree t2 = new InternalBtree();
      t2.fromLeft(this,I/2);
      if (m+1<count) ins(that.key[0],that,m+1);  // accept child
      else t2.ins(that.key[0],that,m+1-count); // give child away
      return t2;
    }
  }

  public void delete(Key k) {
    int m=find(k); Btree s = (BTree)subt[m];
    s.delete(k); key[m]=s.key[0];
    if (s.count<s.half) { // steal from neighbour, if poss
      if (m!=0 && ((BTree)subt[m-1]).count>s.half) {
        s.fromLeft((BTree)subt[m-1],1); key[m]=s.key[0];
      }
      else
      if (m+1<count && ((BTree)subt[m+1]).count>s.half)
        s.fromRight((BTree)subt[m+1],1);
      else
      if (m!=0 && ((BTree)subt[m-1]).count)==s.half) {
        ((BTree)subt[m-1]).fromRight(s,s.count); shift(m+1,-1);
      }
      else
      if (m+1<count && ((BTree)subt[m+1]).count==s.half) {
        ((BTree)subt[m+1]).fromLeft(s,s.count); key[m+1]=((BTree)subt[m+1]).key[0];
        shift(m+1,-1);
      }
    }
  }

}
```