Binomial Heaps

NOTE: THIS PSEUDOCODE IS THE SIXTH DRAFT. EXPECT UPDATES.

A binomial heap class is easy to implement if you have a robust linked list class to work with since most of the heap's methods simply call the methods of the underlying linked list class. The root list of a binomial heap is implemented as a circular doubly-linked list. The children of a node in a binomial heap are kept as a circular doubly-linked list as well.

We will assume that the ordering of the heap (min-heap or max-heap) is specified via a comparator function that is passed into the constructor of the heap. The constructor caches this comparator and initializes its root list, a pointer to the most extreme node in the root list, and the heap's size:

```
function newBinheap(comparator)
    {
    set b to the allocation of a binheap
    set b's comparator to the given comparator
    set b's root list to a new linked list (via linked-list constructor)
    set b's extreme pointer to null
    set b's size to zero
    return b
    }
```

The *union* method makes use of the underlying linked list's union method to merge the root list of the donor heap into the root list of the recipient heap. Note that the donor is emptied out:

```
function union(b,donor)
    {
    //b and donor are binomial heaps
    merge the root list of the donor into the root list of b (via linked-list union)
    increment b's size by donor's size
    set donor's root list to a new empty linked list
    set donor's size to zero
    set donor's extreme pointer to null
    consolidate the root list of b using b's comparator
    }
```

The *insert* routine gets a value to be inserted and creates a node from that value. It then sets the children and parent pointers appropriately. Next, it inserts the new node into its root list. Finally it consolidates the root list:

```
function insert(b,v)
    {
    //b is a binomial heap, v is the value to be inserted
    set a variable n to a new node containing value v
    set the parent of n to n
    set the children pointer of n to a new (empty) linked list
    insert n into the rootlist of b (via linked-list insert)
    increment b's size
    consolidate the root list of b using b's comparator
    return n
    }
```

The *decreaseKey* method is straightforward; it simply updates the value of a given node and bubbles up that new value as much as necessary:

```
function decreaseKey(b,n,v)
    {
    set n's value to the new value v
    bubble up the new value using b's comparator and
        return the node where the new value finishes up
        (also update b's extreme value pointer, if necessary)
    }
```

The *bubble* up method swaps node values if a node's value is less than (or greater than for a max-heap) the value of it's parent. For the CS201 project, it also needs to inform values of their new owners:

```
function bubbleUp(b,n)
    {
    if n is the root of a subheap
        return n;
    else if b's comparator says the n's value isn't smaller than its parent's
        return n;
    else
        {
        inform the value of n that n's parent is the new owner
        inform the value of n's parent that n is the new owner
        swap the values of n and and n's parent
        return bubbleUp(b,n's parent);
        }
    }
```

The *delete* method uses *null* to signify infinity for the heap's comparator:

```
function delete(b,n)
    {
    decreaseKey(b,n,null)
    extractMin(b);
    }
```

Note: the comparator function has to be implemented so that it treats a null value as more extreme than any other value.

The *extractMin* function returns the minimum value in the heap if the heap is a min heap and the maximum value otherwise. It makes use of the cached extreme value (the most extreme value is updated any time the consolidation routine runs). It also makes use of the linked-list's *remove* and *union* methods:

```
function extractMin(b)
    {
    set a variable y to the extreme node in b
    remove y from b's root list (via linked-list remove)
    // the children of y are a linked list
    set each of the children's parent to point to themselves
    merge the children of y into b's root list (via linked-list union)
    consolidate b's root list using b's comparator
    decrement b's size
    return y's value
    }
```

Consolidation is the most complicated task in maintaining a binomial heap because there is no analogous linked-list method. The basic idea is to move all the subheaps from the root root to a consolidation array and then moving them back into the root list:

```
function consolidate(b)
    {
    //create the consolidation array D
    calculate D's size: (log (base 2) of b's size) + 1
    set a variable D to the allocation of an array of D's size
    initialize the D array to nulls

    //place all the subheaps in the D array, combining as necessary */
    while b's root list is not empty
        {
        set a variable spot to the head node in b's root list
        remove spot from the root list (via linked-list remove)
        update the D array with spot
        }

    //transfer the D array back to the heap, keeping track of the extreme value
    set b's extreme pointer to null
    for (i = 0; i < D's size; ++i)
```

```
        {
        if D[i] not equal to null
            {
            insert D[i] into b's root list (via the linked-list's insert method)
           update b's extreme pointer if it's null or b's comparator indicates that D[i] is more extreme
            }
        }
    free the D array (if needed)
    }
```

Updating the consolidation array involves putting the subheap into an empty slot in the array, as indicated by the degree of the subheap. If the slot indicated by the degree is not empty, the original subheap and the subheap in the consolidation array are merged. This merged subheap is then placed into the consolidation array at the next higher slot. Of course, if this new slot is not empty, the merging process continues:

```
function updateConsolidationArray(D,spot)
    {
    set a variable degree to the number of spot's children (using linked-list size)
    while (D[degree] != null)
        {
        combine spot and D[degree], setting spot to the combined subheap
        set D[degree] to null, since that slot is now empty
        increment degree
        }
    set D[degree] to spot
    }
```

The *combine* routine takes two subheaps and makes the subheap, whose root is less extreme, a child of the other:

```
function combine(b,x,y)
    {
    if b's comparator says x's value is more extreme than y's
        {
        insert y into the children of x (via linked list insert)
        set the parent of y to x
        return x
        }
    else
        {
        insert x into the children of y (via linked list insert)
        set the parent of x to y
        return y
        }
    }
```

Although some function names reflect a min-heap (*decreaseKey* and *extractMin*), whether the heap is acutally a min-heap or a max-heap is under the control of the comparator function. Typically, if a min-heap is desired, the comparator returns a negative number if the first value is less than the second, zero if the values are the same, and a positive number if the first value is greater than the second. If a max-heap is desired, the comparator return a negative number if the first value is *greater* than the second, and so on.

The remaining functions of a binomial implementation are left to the reader.