A binomial heap class is easy to implement if you have a robust dynamic array class to work with since most of the heap’s methods simply call the methods of the array class. The root list of a binomial heap is implemented as an array of nodes. The children of a node in a binomial heap are kept in an array 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:

```javascript
function newBinomial(display, compare, update) {
    set b to the allocation of a binomial heap
    set b’s display to the given display function
    set b’s comparator to the given comparator
    set b’s updater to the given update function
    set b’s root list to a new dynamic array (via array constructor)
    set b’s extreme pointer to null
    set b’s size to zero
    return b
}
```

The `merge` method sweeps the donor array, inserting each node of the donor into the recipient heap. to merge the root list of the donor heap into the root list of the recipient heap. Note that the donor is emptied out:

```javascript
function merge(b, donor) {
    // b is a binomial heap, donor is an array
    for each node in the donor
        set the node’s parent to itself
        consolidate b and that node
    free the donor array (optional)
}
```

Multiple calls to `consolidate` should run in amortized constant time for each call, so `merge` should run in amortized linear time.

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:

```javascript
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) dynamic array
    consolidate b and the new node
    increment b’s size
    return n
}
```

Since there is only one call to `consolidate`, `insert` should run in amortized constant time.

The `decreaseKey` method is straightforward; it simply updates the value of a given node and bubbles up that new value as much as necessary. As binomial trees have log n height, the method runs in log time:

```javascript
function decreaseKey(b, n, v) {
    set n’s value to the new value v
    bubble up the new value using b’s comparator
    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 its parent. If the heap's updater function is not null, the updater is called. The purpose of the updater is to inform values that they are now residing in new nodes:

```javascript
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 more extreme than its parent's
        return n;
    else
    {
        call the updater with n's value and n's parent
        call the updater with n's parent's value and n
        swap the values of n and n's parent
        return bubbleUp(b,n's parent);
    }
}
```

The updater function is only used if the decrease key and delete methods are needed. Otherwise, the updater can be set to null.

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

```javascript
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 after the consolidation routine runs):

```javascript
function extractMin(b)
{
    set a variable y to the extreme node in b
    remove y from b's root list (place a null in y's spot)
    // the children of y are an array
    merge the children of y into b's root list (via merge)
    decrement b's size
    store y's value
    free the extreme node
    find the new extreme node
    return the stored value
}
```

To find the new extreme value, one searches the root list from degree 0 to degree \( \log(\text{size of } b) / \log(2) \), inclusive. The basic idea behind consolidation is to combine subtrees of like degree until the growing tree has a unique degree:

```javascript
function consolidate(b,n)
{
    set a variable degree to the number of n's children
    using the array size (not capacity) method
    while b's root list at index degree is not empty
    {
        set n to the combination of n and the subtree stored at the index
        set the slot at index degree to null, since that slot is now empty
        ++degree;
    }
    //degree now indexes an empty slot
    place n at index degree, growing the root list array if necessary
}
```
Note that if the degree is equal to the size of the root list, then that slot would be considered empty.

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

```javascript
function combine(b, x, y)
{
  if b's comparator says x's value is more extreme than y's
  {
    set i to be y's degree
    place y in x's child array at the index i
    growing the array if necessary
    set the parent of y to x
    return x
  }
  else
  {
    set i to be x's degree
    place x in y's child array at the index i
    growing the array if necessary
    set the parent of x to y
    return y
  }
}
```

Although some function names reflect a min-heap (decreaseKey and extractMin), whether the heap is actually 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.