Since multiple structs are needed for the binary heap and graph implementations, the purpose of this thread is to clear up what all we need and to answer any questions about them (including my own).

The following descriptions include only data in each struct, not the methods relevant to each class. This is my understanding of each struct we need, what it should contain, and why (based on my work so far and what I've gathered from several other forum posts):

**Binary heap** - to implement the priority queue for Prim's algorithm
- a comparator - a pointer to a function that will compare keys
- pointer to a linked list of roots - essential to the binary heap structure
- a node pointer to the extreme value in the root list - for extractMin
- the size of the heap - important for consolidation

**Circular doubly linked list (CDLL)** - essential to the binary heap structure
- the size of the list - important for consolidation
- a head (node) pointer - helps make it circular
- a tail (node) pointer - helps make it circular

(I think I have the above two correct; here is where I get confused.)

**Node** - what the linked list and binary heap are comprised of
- next node pointer - for the list
- previous node pointer - for the list
- parent node pointer - points to a node's parent in the binary heap
- children linked list pointer - since you store children as a linked list in a binary heap
- a value - I'm unclear on this. I believe this should be a void pointer (for the sake of generality) that contains a vertex struct. However, it's possible that it could just be a vertex pointer. Help?

**Vertex** - The node's value
- predecessor - I have no idea what this is/does...Help?
- owner node pointer - the node in the binomial heap that holds the vertex as its value. I know what this is but not what it's used for.
- a key - I have no idea what this is/does...Help?

We also need an adjacency matrix to store the actual graph. The dimensions of this should not exceed the number of the highest-numbered vertex. It's OK if this graph is sparsely populated, since we'll only be using the matrix to directly access edges, which takes constant time (plus memory is cheap).

Any help on the questions I've mentioned or corrections on things I may have wrong would be very greatly appreciated!
That cool thing about void* is that you can store any pointer in them. So, our Node's are to store a void* value. For the purpose of this project, we are going to store a vertex* in it.

The vertex predecessor is helpful in determining the level that the vertex should appear in the BFS. You are going to update the predecessor whenever you update the key value to a vertex in the binheap (depending on if it has been added to the tree or not).

The vertex owner is used to link between the vertex class and the binheap class. Because our binheap takes Nodes and knows how to work with those Node's, we need to pass a node in that contains that vertex - thus the name v->owner.

A key is going to be the current edgeweight to that vertex and serves as the key for our binheap (priority queue).

Understood about the void * in nodes.

Understood about the vertex predecessor.

I don't really understand why it's necessary for a vertex to store the node that owns it. Is there a point at which you have a pointer to a vertex and you need to find its node? If you could point me to a specific function, that'd be great.

Understood about the key. So I also need vertex->number, so that its edgeweight and identity are both stored.

A couple of notes. You don't need a tail pointer in a doubly-linked list if it is circular.

You need the node in the binomial heap that holds the vertex whose key value is being reduced by Prim's algorithm. This is because you have to call the binomial heap's decreaseKey operation when the vertex's key value changes and because decreaseKey takes a node as an argument.
Since I couldn't edit the original post, here's an updated version, with Ben and Dr. J's comments taken into account:

Binary heap - to implement the priority queue for Prim's algorithm
- a comparator - a pointer to a function that will compare keys
- a swapper - not needed if you just hard code in the vertices. Needed if you want to swap values.
- pointer to a linked list of roots - essential to the binary heap structure
- a node pointer to the extreme value in the root list - for extractMin
- the size of the heap - important for consolidation

Circular doubly linked list (CDLL) - essential to the binary heap structure
- the size of the list - important for consolidation
- a head (node) pointer - helps make it circular
- a tail (node) pointer - not necessary, will probably make your life more difficult

Node - what the linked list and binary heap are comprised of
- next node pointer - for the list
- previous node pointer - for the list
- parent node pointer - points to a node's parent in the binary heap
- children linked list pointer - since you store children as a linked list in a binary heap
- a value - a void * (void pointer) that will store vertices for our purpose

Vertex - The node's value
- predecessor - a vertex pointer, helpful in determining the level that the vertex should appear in the BFS. You are going to update the predecessor whenever you update the key value to a vertex in the binheap (depending on if it has been added to the tree or not).
- owner node pointer - the node in the binomial heap that holds the vertex as its value.
  Necessary for calling the binomial heap's decreaseKey operation on the proper node when the vertex's key value changes and because decreaseKey takes a node as an argument.
- a key - a(n) (unsigned) int, the current edgeweight to that vertex and serves as the key for our binheap (priority queue).

We also need an adjacency matrix to store the actual graph. The dimensions of this should not exceed the number of the highest-numbered vertex. It's OK if this graph is sparsely populated, since we'll only be using the matrix to directly access edges, which takes constant time (plus memory is cheap).

You'll also have an array of vertices. So essentially when you read in a new vertex, you add it to both your adjacency matrix and your vertex array.

Lastly, you'll need a queue class for the level-order traversal; this should be quite similar to the one from assignment 2!