CompOrg Final Project

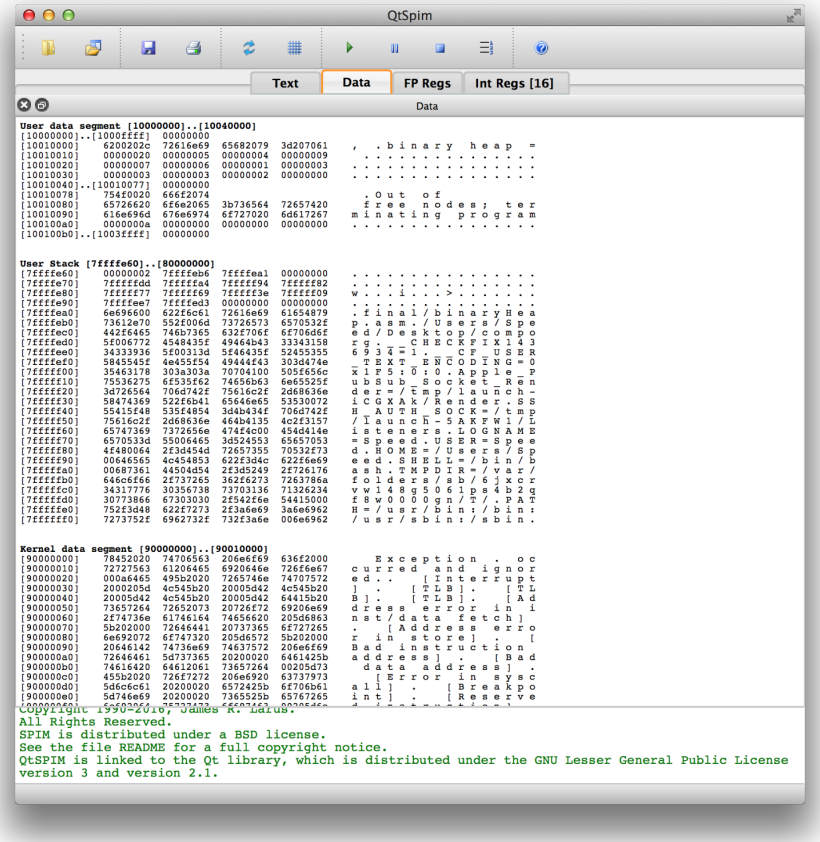
John Enquist
3/15/16
John Rieffel

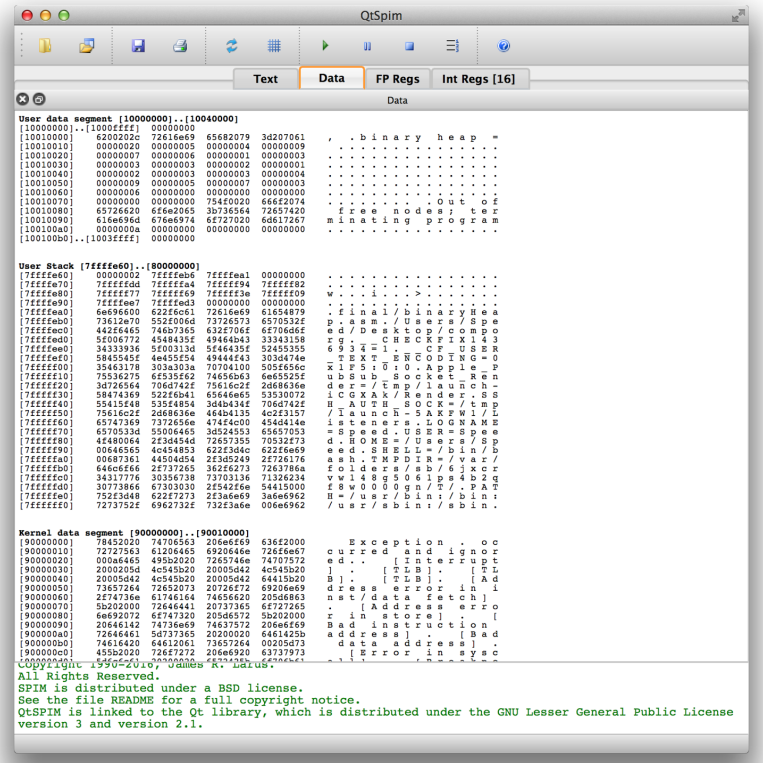
For my final project I decided to implement a binary heap in mips assembly. While being able to design any data structure in mips, I was most interested in the binary heap due to the lack of pointers used in this data structure. Throughout the term the projects we had to do in mips relied heavily on pointers, and I was interested in how a data type that used no pointers could be constructed. First, I had to decide what way I was going to store the data. Like a priority queue, I made the heap array-based, where the smallest elements are at the top of the heap and the biggest are at the bottom. This meant that for each index in the array i, the children of index i could be found at indexes 2*i+1 and 2*i+2. Similarly, at any index i, the parent of that index could be found at index i/2 if you rounded the remainder down.

There are two main ways to implement a minimum binary heap; recursively and iteratively. Because recursion is rather difficult in mips, I decided on the iterative method. Once I decided on an array based implementation with iterative subroutines, it was time to decide what the subroutines I would need to make this data structure would be. I needed a remove method that takes the minimum element from the heap, which also happens to be the first element, and returns the value. Similar to popping from the top of a stack, the minimum heap is only able to remove the smallest item from the top. I also need an insert method for adding new elements to the heap in the proper place. For both of these methods I had to devise a way to balance and restructure the heap after each insert and remove. To accomplish this, I made a percUp and percDown subroutine for each method respectively. Given a numbers index, percUp works by checking the indexes parent to see if the parent is larger or smaller than the child. If it is larger, then the two elements are swapped and the process is repeated. PercUp continues until the

parent of the number is smaller than it. PercDown does the opposite, starting from the index of a parent node, and checking to make sure the two children are bigger than it. If the minimum child of the parent index is smaller than the parent, the two are swapped. In order to make percDown, I decided to write a subroutine minChild. Given a parent index, minChild would look at the two children and return the smaller of the two. This was very useful for percDown as I didn't have to worry about finding the indexes of child nodes in the method. Once I had written the percDown method, I was ready to write the buildHeap method. Given an unsorted array, buildHeap would take the elements and insert them into a space allocated for the heap in the proper location by calling percDown after each insertion. Now that I had made a heap that could insert and remove, I added a find method, which given a number n, would check to see if the number was in the heap and return 1 if it was. While not necessary, It is always useful to have a find method in a data structure. Finally, I wrote a print method so the user could get an easy to look at view of the completed heap.

Data Before heaping:





Data After heaping

Console after buildHeap, RemoveMin, and Insert 1