Colorado Mesa University
Computer Science and Engineering

CSCI112 Spring 2021

Practicum
Due Friday, in class.

Aim:
To implement and analyse Heapsort.

Task:
Heapsort is one of the most efficient sorting algorithms, especially since it has no worst case. In fact, many libraries use Heapsort in the guise of QuickSort, as it has the same O(nlogn) performance. So let's see how easy it is to implement.

`Prac12main.cpp` is a driver for generating random shuffles of sets of integers. We are now going to write the function

```
void Sort(int array[], int nval, int& ncomp);
```

to use Heapsort to sort the data and return the number of comparisons.

The first step of Heapsort is to create the balanced binary tree containing the original data – that's already done.
In the lecture notes you'll find the code for a function called `MakeHeap`. This is the basis for the Heapsort code. This assumes the following data items:

```
int n, nlast;
int *value;
```

We make these three items, plus the comparison count

```
int cCount;
```

global to the file `sort.cpp` which will contain all the functions. How do we do that? We place them in the global area with the keyword `static` in front of them. Then, in the implementation of `Sort`, copy `nval` to `n`, and `array` (the address) to `value`. Set `nlast` to be the location of the last parent in the tree, namely `n/2-1`. Initialise `cCount` to zero – this will be stored in `ncomp` at the completion of the sorting. Copy the code to `MakeHeap` from the lecture notes. You will find it useful to write the function

```
bool Compare(int,int);
```

which tests whether `value[`*`first arg`*`]` is less than `value[`*`second arg`*`]` and increments the comparison count, and replace the data comparisons in `MakeHeap` by a call to this function.

Next, copy and adjust the code for making the entire tree into a heap from the lecture notes into the function `Sort`.

To use this function to sort we now move the top of the tree, `value[0]`, to the end of the tree, `value[n-1]`. We then reduce the tree's size by 1, and adjust the value of `nlast`. We place this into a loop which terminates when the size of the set to sort is 1.

Link `sort.cpp` with `L12main.cpp` to produce the executable. You can use `–DPRINT` to be able to see whether your sort is in fact sorting, before you generate the graph to see the performance.

$ g++ -DPRINT Prac12main.cpp sort.cpp