A PostScript Heap Sort

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

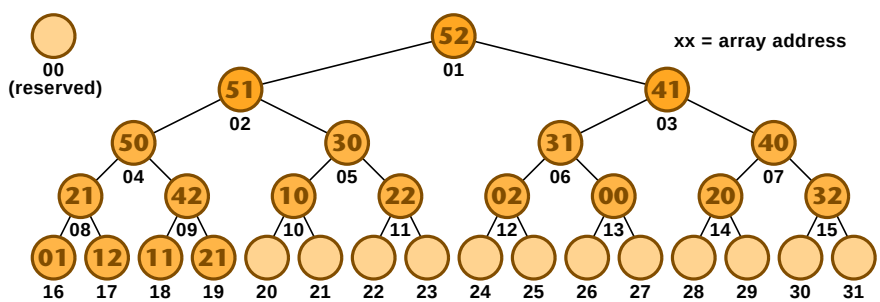
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Back in our previous **GuruGram #31**, we looked at some **PostScript** approaches to **Bubble Sorting** and **Presorted Bubble Sorting**. It turns out there is a "better" way to sort that is called a **heap sort**. A heap sort is very fast and very efficient, especially for very large **n**. It also uses very few resources and is exceptionally easy on **PostScript** virtual memory and garbage collection.

On the negative side, the incredibly elegant heap sort is somewhat hard to understand, involves somewhat longer and much more obtuse code, and is somewhat slower for very small **n**. Ferinstance, sorting 300 items with a heap sort might take 20 milliseconds, compared to 12 milliseconds for a presorted bubble sort. Or 140 milliseconds for a conventional bubble sort. For nearly all values of **n**, though, a heap sort is either ridiculously faster or else is more than fast enough that it does not really matter.

What is a heap?

A **heap** is not just a slang term. It is instead a carefully organized **binary tree data structure** . One having very exacting rules for its creation and use. Starting with some data of **[(01)(02)(50)(10)(11)(20)(41)(21)(22)(52)(30)(31)(00)(32)(40)(42)(12)(51)(21)]**, a heap might initially look like this...



We see a heap-shaped "triangular" data structure. Our first rule is that…

> **A heap has to be built SLOWLY from the top down but RAPIDLY from left to right.**

A heap can hold up to **2^n** elements. **n** is continuously chosen to be just larger than needed for the available data. Should more data arrive, **n** is bumped by one and a new base row is added to the data structure.

From our above build rule, we can also see that…

> **Any "empty" elements ALWAYS have to be at the LOWER RIGHT on the BOTTOM row.**

Each **parent** or **apex** element of a heap **triad** normally has **two** children. One crucially important heap rule is that…

> **On any triad, the apex parent must ALWAYS be EQUAL or LARGER than either child.**

When you are building a heap, numerous swaps that "ripple up" or **upchuck** through the tree might be needed to force this rule.

Similarly, when you are extracting data from a heap, numerous swaps that "ripple down" or **downchuck** through the tree might be needed as well. The beauty of a large heap sort is that **ridiculously fewer comparisons will be needed** than will be required with bubble and certain other sorts.

A heap is easily constructed inside an array. Just place each row into the array in sequential order. **Starting with location 1, NOT location 0!** And, conveniently, the results of a heap sort can go back into that very same array.

Heap array addressing is remarkable simple…

> **The TOP of the heap always goes in array position #1**
> **Note that this is NOT position 0!**
>
> **The LEFTMOST triad address will be apex*2.**
> **The RIGHTMOST triad address will be (apex*2) + 1**
>
> **The APEX triad address will be Int(leftmost/2)**
> **The APEX triad address will also be Int(rightmost/2)**

There are two steps to doing a heap sort. You first **fill** or create the heap. While forcing the above rules. At this point, your **highest** data value will always end up in position #1 and is easily grabbed **without any comparisons or tests at all!**

You then **empty** your heap to complete the sort. Do this by taking the current **last** heap item and swapping it with the **first**. This should then place the **highest remaining** item in proper position at the end of the array.

You then do a **downchuck** that ripples the moved item as far into the heap as needed to guarantee each parent of a triad is larger than its children.

The process repeats as the heap shrinks down. Until nothing is left but your correctly sorted array.

Some Code

As always, you can use **Acrobat Distiller as a General Purpose PostScript Computer**. Following the details of **GuruGram #29**. More details can be found in our **PostScript** library.

Let's look at some heap sort code from the outside in….

```
/heapsort {
  /heap stddat length              % create heap array
     1 add array store
  fillheap                         % create heap from strings
  emptyheap                        % sort heap by emptying
            } bind store
```

We simply fill the heap and then empty it. All the while following our exacting heap rules. We'll assume you are starting with an unsorted arrray of strings that has been named **stddat**. Here's the fill routine…

```
/fillheap {/hposn 1 store          % initialize to position 1
    stddat {upchuck                % enter one string at a time
    /hposn hposn 1 add store       % advance position counter
            } forall               % repeat for all strings
          } bind store
```

Each new string is tentatively placed at the end of the heap, which is usually towards the lower right of the binary tree. Working slowly down and rapidly left to right.

A recursive **upchuck** routine is then called to move the string as far up into the heap as is needed to force the "parent is always bigger or equal than child" rule…

```
/upchuck {/cposn hposn store   % initialize pointer to end
  {/pposn cposn 2 idiv store   % begin recursive loop
   pposn 0 le                  % parent address <1?
     {heap exch cposn exch     % yes, save child
      put exit} if             %    and exit
   heap pposn get              % get parent value
   2 copy  ge {                % is swap needed?
    heap exch cposn exch put}  %     yes, swap
    {pop heap exch cposn exch  % no, save child
     put exit} ifelse          %     and exit
   /cposn pposn store          % old parent = new child
   } loop                      % repeat until parent>child
       } bind store
```

For each trip, a new parent address is calculated from **cposn** by using **2 idiv**.

There are **three** possible outcomes: If the parent address is less than 1, then you have previously reached the top of the heap and you exit, saving the child.

If the **pposn** parent address is within the heap and the parent value equals or exceeds the child, you are also done and exit, saving the child.

If the parent value is less than the child, you swap child and parent, save the new child, and then recursively go up one level. The child string always remains on the stack until needed.

It takes many iterations to get from your initial data to a properly filled heap. To fully understand this process, start with our original data above and try to get the heap result previously shown. Remember to **fill your heap slowly from top to bottom** and **rapidly from left to right** and that **the parent of each heap triad must equal or exceed either child.**

Here's the mid-level **emptyheap** routine…

```
/emptyheap {
  hposn 1 sub -1 2             % loop through all strings
    {/h1posn exch store        % remaining heap size
     heap h1posn get           % end item to stack
     heap 1 get                % get highest apex value
     heap exch h1posn exch put % place at end
     downchuck                 % rework heap triads
     } for                     % redo smaller heap
                 } bind store
```

At any time, the **largest** remaining string will be on **top** of the heap in address #1. This gets swapped with the **last** string in the heap, stacking up everything in order from the array end. **downchuck** is then used to ripple the last string into its proper mid-heap position…

```
/downchuck { /apex 1 store          % starting at top...
   {/maybe apex 2 mul store         % left child address
   maybe h1posn 2 sub le            % full triad available?
     {basecheck swapcheck}          % find larger base & swap?
     {maybe h1posn 1 sub eq         % left only available?
       {heap maybe get              % get left base then swap?
        swapcheck}
       {heap exch apex exch         % no, save apex and exit
        put exit} ifelse
            } ifelse
    /apex maybe store               % reset apex for next triad
    } loop                          % continue recursively
             } bind store
```

Downchucking is somewhat more complicated and time intensive than upchucking. This time, there are four possible outcomes:

If the next child address exceeds the current heap size, you are done and exit saving the last parent. If the next child address equals the current heap size, you will have only one child. This child is then compared using **swapcheck**.

If the next child address is less than one less than the current heap size, you will have two children. These needed tested against each other using **basecheck**. The larger of the two is then compared using **swapcheck**.

swapcheck in turn will decide whether a swap is needed or not. If no swap is needed, you exit while saving the last parent. If a swap is needed, that swap is made and the test child becomes the new parent for yet another recursive round.

Here is **basecheck**…

```
/basecheck { heap maybe get     % get left string
    heap maybe 1 add get        % get right string
    lt {/maybe maybe 1 add      % set address of larger point
       store} if
    heap maybe get              % get larger
             } bind def
```

The left and right triad bottom strings are compared, and the larger one is chosen.

Finally, here is **swapcheck**…

```
/swapcheck {2 copy lt                % is a swap needed?
    {heap exch apex exch put}        % yes, swap and save apex
    {pop heap exch apex exch         % no, save apex only and exit
     put exit} ifelse
            } bind store
```

The stack will normally hold the **maybe** string on top of the **apex** string. Letting **swapcheck** compare the two. If a swap is needed, the new apex is saved, and testing continues. If not, the old apex is saved, and you exit.

Some ready-to-use code and sample string data appears as file **HEAPPS01.PSL**.

For More Help

Additional **PostScript** and **Acrobat** and assistance is available per the previously shown web links. Custom programming and design services are now available at our standard consulting rates. Per our **InfoPack Services**. Or you can directly **email** me.

Additional **GuruGrams** columns await your ongoing support as a **Synergetics Partner**.