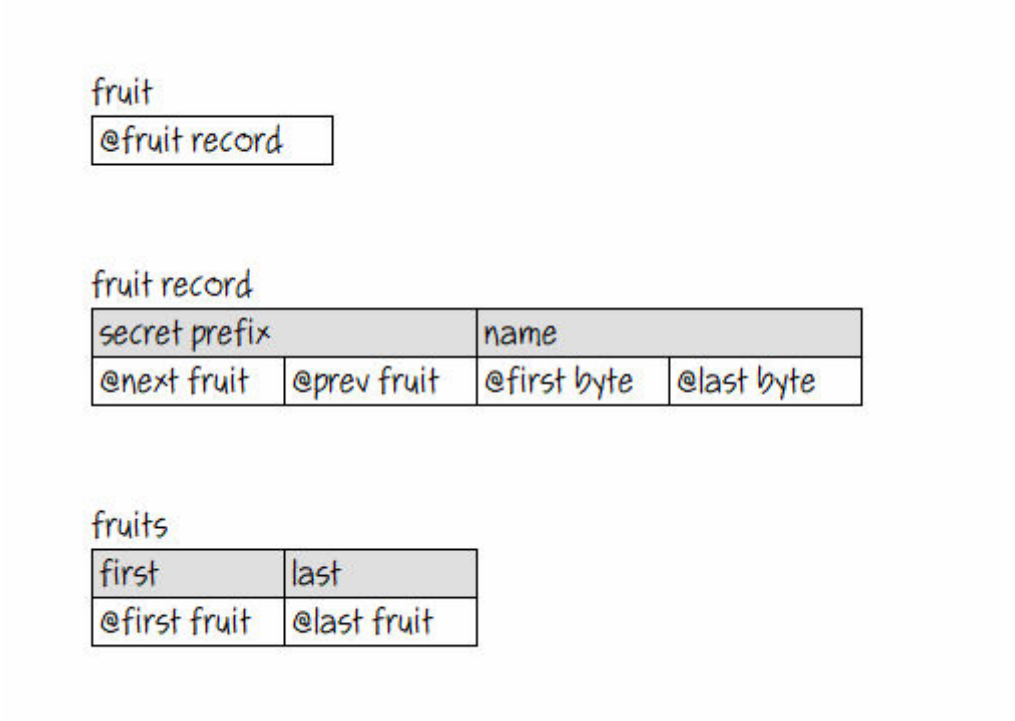
A Simple Merge Sort

gerryrzeppa Uncategorized May 4, 2018 3 Minutes

The only compound data types native to Osmosian Plain English are records and doubly-linked lists. When we need to sort a list, we use the simple recursive merge sort that I'll show you below. But first we need something to sort. Let's sort fruits, and let's begin with a type definition:

```
A fruit is a thing with a name.
```

When the word "thing" appears in a type definition, our compiler works a little magic behind the scenes to make things (pun intended) more powerful and flexible. The above definition, for example, actually causes the following data types to be defined:



So a Fruit is really nothing but a pointer containing the address of a Fruit Record.

And each 16-byte Fruit Record has a hidden 8-byte prefix with two pointers for linking these records into lists, together with the fruit's name, which is a string. Plain English strings are stored in the Heap and can be any length. So the Name in the Fruit Record is actually just two pointers to the first and last bytes of the string on the Heap, respectively. *String* memory is managed automatically, but *thing* memory is managed by the programmer.

The third type generated by the compiler serves as the anchor for lists of Fruit Records. Such lists are simply (and intuitively) called Fruits, the plural of Fruit.

Now let's add some fruits to a list, in random order, and sort them. Here are the top level sentences in our test program:

```
To run:
Start up.
Create some fruits.
Write the fruits on the console.
Skip a line on the console.
Sort the fruits.
Write the fruits on the console.
Destroy the fruits.
Wait for the escape key.
Shut down.
```

And here are the routines that will be called to "Create some fruits":

```
To create some fruits:
Add "banana" to the fruits.
Add "apple" to the fruits.
Add "orange" to the fruits.
Add "bacon" to the fruits.
Add "pineapple" to the fruits.
Add "pomegranate" to the fruits.
Add "tomato" to the fruits.
Add "grape" to the fruits.
Add "fig" to the fruits.
Add "date" to the fruits.

To add a name to some fruits:
Allocate memory for a fruit.
Put the name into the fruit's name.
Append the fruit to the fruits.
```

Now we're ready to sort. This is the sort routine:

```
To sort some fruits:
If the fruits' first is the fruits' last, exit.
Split the fruits into some left fruits and some right fruits.
Sort the left fruits.
Sort the right fruits.
Loop.
Put the left fruits' first into a left fruit.
Put the right fruits' first into a right fruit.
If the left fruit is nil, append the right fruits to the fruits; exit.
If the right fruit is nil, append the left fruits to the fruits; exit.
If the left fruit's name is greater than the right fruit's name,
move the right fruit from the right fruits to the fruits; repeat.
Move the left fruit from the left fruits to the fruits.
Repeat.
```

When we run this program, the output on the console looks like this:

```
banana
apple
orange
bacon
pineapple
pomegranate
tomato
grape
fig
date

apple
bacon
banana
date
fig
grape
orange
pineapple
pomegranate
tomato
```

But is it fast? Let's see using this modified test program:

```
To run:
Start up.
Write "Working..." on the console.
Put 10000 into a count.
Create some fruits using "apple" and the count.
Start a timer. Sort the fruits. Stop the timer.
Write the timer then " milliseconds for " then the count on the console.
Destroy the fruits.
Put 100000 into the count.
Create the fruits using "apple" and the count.
Start the timer. Sort the fruits. Stop the timer.
Write the timer then " milliseconds for " then the count on the console.
Destroy the fruits.
Put 1000000 into the count.
Create the fruits using "apple" and the count.
Start the timer. Sort the fruits. Stop the timer.
Write the timer then " milliseconds for " then the count on the console.
Destroy the fruits.
Wait for the escape key.
Shut down.
```

The fruits this time, at the start, look like this:

apple 0010000
apple 0009999
apple 0009998
...

And the output on the console looks like this:

```
Working...
63 milliseconds for 10000
750 milliseconds for 100000
8875 milliseconds for 1000000
```

Not quite linear, I admit. But not exponentially bad, either. Ten times as many records take *roughly* ten times as long to sort. There's a technical way of saying this using big "O's" and little "n's' and "logs" and stuff, but Plain English programmers don't generally think that way. And it's stable, as a Plain English programmer would expect — records with duplicate sort values retain their original sequence.

Good, simple, useful stuff.