Heapsort (C)

From LiteratePrograms

Other implementations: **C** | Erlang

The heapsort sorting algorithm is a sorting algorithm which interprets its input as an implicit max-heap data structure, a kind of binary tree data structure where every node's value is larger than that of both its children. It requires random access, so this implementation only supports heapsort on arrays.

Contents

Interface and element access

Like quicksort, heapsort is a comparison sort. We will give it the same argument list as the standard library's `qsort()`, allowing it to sort any data given a comparison predicate (which returns negative, zero, or positive values depending on whether its first argument is less than, equal to, or greater than its second):

<<heapsort declaration>>=
```
void heapsort(void *base, size_t num_elements, size_t element_size,
              int(*comparer)(const void *, const void *))
```

Since accessing elements in such a representation is nontrivial, we supply two helper functions, one to compare two elements and one to exchange two elements, along with helper macros to increase readability:

<<array element accessors>>=
```
static int compare_elements_helper(void *base, size_t element_size, int idx1, int idx2,
                                   int(*comparer)(const void *, const void*))
{
    char* base_bytes = base;
    return comparer(&base_bytes[idx1*element_size], &base_bytes[idx2*element_size]);
}

#define element_less_than(i,j)  (compare_elements_helper(base, element_size, (i), (j), comparer) < 0
#define element_greater_or_equal(i,j)  (compare_elements_helper(base, element_size, (i), (j), compar

static void exchange_elements_helper(void *base, size_t element_size, int idx1, int idx2)
{
    char* base_bytes = base;
    int i;
    for (i=0; i<element_size; i++)
    {
        char temp = base_bytes[idx1*element_size + i];
        base_bytes[idx1*element_size + i] = base_bytes[idx2*element_size + i];
        base_bytes[idx2*element_size + i] = temp;
    }
}

#define exchange_elements(i,j)  (exchange_elements_helper(base, element_size, (i), (j)))
```

We need `stddef.h` for `size_t`:

```
<<header files for types>>=
#include <stddef.h>
```

The helper for exchanging elements could be made more efficient on many platforms by moving larger addressable units at one time, for example by using `memcpy()`, but we don't address this optimization here.

Main algorithm

The basic structure of heapsort is very simple: we divide the array into two pieces, the first representing the max-heap and the second representing the *k* largest elements of the list. We then repeatedly extract the root of the heap, which is the largest element in the heap, and prepend it to the second part, until the heap is empty:

```
<<heapsort definition>>=
heapsort declaration
{
    int heap_size = num_elements;
    rearrange array into implicit heap
    while (heap_size > 0)
    {
        exchange_elements(HEAP_ROOT, heap_size - 1);
        heap_size--;

        preserve heap properties
    }
}
```

The specific heap representation we choose is as follows:

- The root is located at index 0 of the array:
- The two children of the element at index *i* are located at indices $2i+1$ and $2i+2$.
- The parent of the element at index *i* is located at index floor(($i-1$)/2).

We implement these with the following macros:

```
<<heap representation macros>>=
#define HEAP_ROOT       0
#define LEFT_CHILD(i)   (2*(i) + 1)
#define RIGHT_CHILD(i)  (2*(i) + 2)
#define PARENT(i)       ((i-1) / 2)
```

Preserving heap properties

Our initial task is to take an arbitrary array and arrange it into a valid max-heap. In a max heap, each element is greater than or equal to both its children. One straightforward way to do this is go through the elements one by one, exchanging each one with its parent repeatedly until it is less than or equal to its parent (this repeated exchanging is referred to as "bubbling up" the element). This works because of the following loop invariant:

> *After bubbling up the first* k *elements, the first* k *elements will form a valid max-heap*.

This is easy to see, since during this process, existing heap elements are only exchanged with greater-or-equal elements (therefore, if your parent is greater-or-equal now, it will stay that way). The complete process looks like this:

<<rearrange array into implicit heap>>=
```
int i;
for (i=0; i<num_elements; i++)
{
    /* Bubble up element i */
    while (!element_greater_or_equal(PARENT(i), i))
    {
        exchange_elements(PARENT(i), i);
        i = PARENT(i);
    }
}
```

Once we've done this, we have a max-heap, and the largest element will be at index zero. Note that this step takes only O(*n* log *n*) time, because the height of the heap is limited to O(log *n*).

However, exchanging the heap root with element `heap_size-1`, the next thing we do, may invalidate the heap data structure because now the element at index zero may have children larger than itself. To deal with this, we perform a very similar "bubble down" process where we repeatedly exchange the new root element with one of its children until it is greater than or equal to both its children. An oversimplistic approach might look like this:

<<incorrect bubble down 1>>=
```
while (!element_greater_or_equal(i, LEFT_CHILD(i)) ||
       !element_greater_or_equal(i, RIGHT_CHILD(i)))
{
    exchange_elements(i, LEFT_CHILD(i));
    i = LEFT_CHILD(i);
}
```

However, this may not preserve the heap property, because if the left child of element *i* is less than the right child, the exchange will make the left child the parent of a larger element, which is forbidden. Instead, we must exchange element *i* with the larger of its two children:

<<incorrect bubble down 2>>=
```
while (!element_greater_or_equal(i, LEFT_CHILD(i)) ||
       !element_greater_or_equal(i, RIGHT_CHILD(i)))
{
    if (element_greater_or_equal(LEFT_CHILD(i), RIGHT_CHILD(i)))
    {
        exchange_elements(i, LEFT_CHILD(i));
        i = LEFT_CHILD(i);
    }
    else
    {
        exchange_elements(i, RIGHT_CHILD(i));
        i = RIGHT_CHILD(i);
    }
}
```

This is almost correct except for one problem: element *i* might not have a left or right child! If it's near the bottom of the heap, it may have no right child or no children at all, and we can determine whether this is the case by comparing their indexes to the heap size:

<<determine whether each child of element i exists>>=

```
int has_left_child  = LEFT_CHILD(i)  < heap_size;
int has_right_child = RIGHT_CHILD(i) < heap_size;
```

Now we can write the final, correct, bubble down procedure:

<<preserve heap properties>>=

```
/* Bubble down the root */
i = 0;
while (1)
{
    determine whether each child of element i exists
    if (has_left_child && !element_greater_or_equal(i, LEFT_CHILD(i)) &&
        (!has_right_child || element_greater_or_equal(LEFT_CHILD(i), RIGHT_CHILD(i))))
    {
        exchange_elements(i, LEFT_CHILD(i));
        i = LEFT_CHILD(i);
    }
    else if (has_right_child && !element_greater_or_equal(i, RIGHT_CHILD(i)))
    {
        exchange_elements(i, RIGHT_CHILD(i));
        i = RIGHT_CHILD(i);
    }
    else
    {
        break;
    }
}
```

This bubbling down takes O(log *n*) time, again because the heap has height O(log *n*). Since we perform it once for each element, this part of the sort also takes O(*n* log *n*) time. This completes the implementation, and the total time is O(*n* log *n*) in the worst case.

Files

We can place the routine in source files for reuse, exposing only the main heapsort function for calling:

<<heapsort.h>>=

```
#ifndef _HEAPSORT_H_

header files for types

heapsort declaration;

#endif
```

<<heapsort.c>>=

```
#include "heapsort.h"
```

```
heap representation macros
array element accessors
heapsort definition
```

Download code (http://en.literateprograms.org/index.php?title=Special:Downloadcode/Heapsort_(C)&oldid=18432)

Retrieved from "http://en.literateprograms.org/index.php?title=Heapsort_(C)&oldid=18432"

Categories: Programming language:C | Environment:Portable | Heapsort