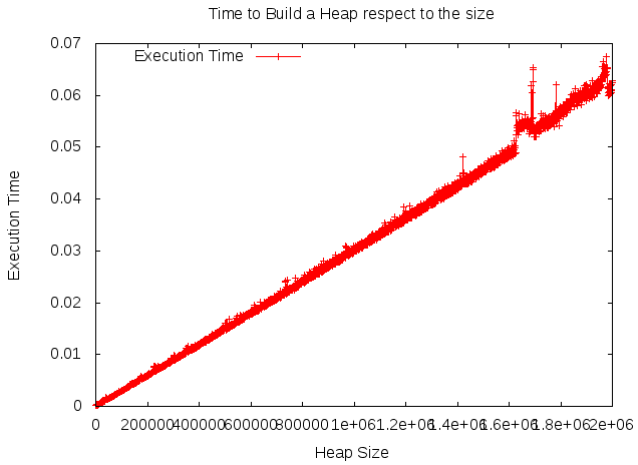
Bin Heap Implementation

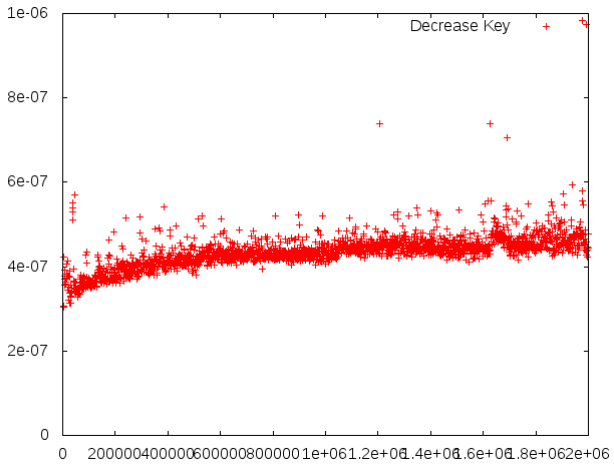
Maria Grazia Berni

Brief Description of the Implementation

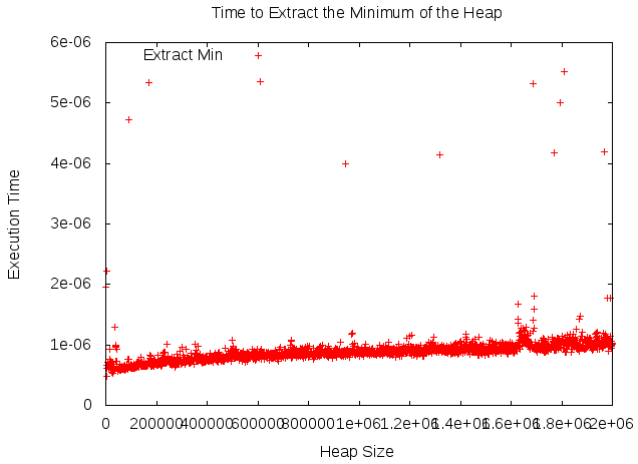
In my array implementation of bin heap, this is represented with the struct heap. The struct contains an array that stores the nodes of the heap, the size of the heap, the capacity, the size of the key and the particular function that imposes the total order. The size of the array containing the nodes has no limits, in fact every time the capacity reach zero the size of the array is doubled. To build the array is necessary to copy the nodes provided by the user in the array data of the struct, and then call heapify to fix the heap property. So the complexity to build the heap is $\Theta(n)$ (it should be at most $\mathcal{O}(n)$ when is not necessary to copy the array representing the heap). Here a plot of the execution time to build the array respect ot the size:



The plot clearly shows a linear dependence of the execution time respect to the size of the heap. The implementation misses an infinity value for the insert function. When inserting a new value, instead of using an infinity value and then decrease,the bigger value among the value to insert and the one of the parent of the right most leaf is chosen(where the value must be inserted). In this case if the parent is the bigger one its value will be decreased, otherwise the node is already in his right position. Inserting value has complexity $\mathcal{O}(\log(n))$, which is the complexity of decreasing the key. Here some plot of the execution time of the function decrease_key :



and of the execution time of extracting the minimum value, whose complexity coincides with that of heapify, $\mathcal{O}(\log(n))$.



Implementation without swapping the key

In order to implement the binary heap avoiding to swap the keys, I added to the struct two integer arrays, Pos and $Free\_Pos$, and two more counters $total\_nodes$ and $place$. Now the counter num_of_elem indicates the number of element belonging to the heap, while $total\_nodes$ is the total number of element stored in the array of the keys, even if they don't belong anymore to the heap, $place$ indicates the number of position in the keys array that are storing an element, but that element is no more in the heap. The $i-element$ of the array Pos, indicates the position of the $i-element$ of the heap, so that $ADDR[H, Pos[i]]$ is the address of that element in the keys array. When extracting the minimum, this is not physically erased from the

keys array, but only from the array Pos, but before doing this, its position is saved in the first free position of the array $Free\_Pos$, and the variable $place$ is increased by one. So, when inserting a new element, if the variable $total\_nodes$ is less then the maximum size of the keys array, the insertion is done as usual, saving the key in the first free position of the keys array, that is in $ADDR(H, num\_of\_elem)$, otherwise, the variable $place$ is scrolled one position back, and the new key is saved in $ADD(H, Pos\_free[place])$, this position is then saved in $Pos[num\_of\_elem]$.

Exercises number 4

taking the node with index $\left\lfloor \frac{n}{2} \right\rfloor + 1$ and calculate its left child, this will have index :

$$LEFT\_CHILD(\left\lfloor \frac{n}{2} \right\rfloor + 1) = 2(\lfloor \frac{n}{2} \rfloor + 1) > n$$

so the child is not a valid node, i.e the node indexed by $\left\lfloor \frac{n}{2} \right\rfloor + 1$ is a leaf. Taking the previous element, indexed by $\left\lfloor \frac{n}{2} \right\rfloor$, in the case n is even it will have only a left child child indexed by n, in the case n is odd it will have a left child indexed by n-1 and a right child indexed by n. So, the leaves in the heap are equal to $\lfloor \frac{n}{2} \rfloor$ and they start form $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n.

Exercises number 5

Consider a min-heap to build on a array. This happen, for instance in the procedure for building the heap, if it happens that the bigger element is in the root. Then a call to heapify will propagate along all the levels in order to restore the heap property. Since the number of levels in a heap with n elements is $\log(n)$ and swapping the key (one time for each level) costs $\Theta(1)$, the overall cost of heapify in the worst case is $\Theta(\log(n))$, so the lower bound is $\Omega(\log(n))$.

Exercises number 6

Proof by induction. Consider a tree H with n nodes. It holds for the base case $h = 0$, where the number of leaves is $\lceil \frac{n}{2} \rceil = \lceil \frac{n}{2^{0+1}} \rceil$ Suppose that the property holds for $h - 1$ and let N_h be the number of nodes at height h of the tree H. Consider the tree H' obtained removing the leaves of H. So it has $n' = \lfloor \frac{n}{2} \rfloor$ elements (ex 4). The nodes that in H are at height h in H' would be at height h-1. If N'_{h-1} is the number of nodes at height h-1 in H', so $N_h = N'_{h-1}$, but $N'_{h-1} = \lceil \frac{n'}{2^h} \rceil$ so $N_h = \lceil \frac{\lfloor \frac{n}{2} \rfloor}{2^h} \rceil <= \lceil \frac{(\frac{n}{2})}{2^h} \rceil = \lceil \frac{n}{2^{h+1}} \rceil$

Homework of 24/03/2020 a

In case the array A contains n elements, we will repeat the function $esxtact\_min$ n times, and at each of these steps the dimension of the heap will decrease by one, so the complexity to empty the tree can be calculated in this way (so the complexity of the entire while loop):

$$T(n) = \sum_{i=0}^{n-1} \Theta(n-i) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

which is the overall complexity of the entire piece of code in these circumstances.

Homework of 24/03/2020 b

In case the array A contains n elements, we will repeat the function $extract\_min$ n times, and at each of these steps the dimension of the heap will decrease by one, so the complexity to empty the tree can be calculated in this way (so the complexity of the entire while loop):

$$T_{empty}(n) = \sum_{i=0}^{n-1} \mathcal{O}(\log(n-i)) = \sum_{i=1}^{n} \mathcal{O}(\log(i)) = \mathcal{O}n(\log(n))$$

so the overall complexity of the entire code is $T(n) = \Theta(n) + \mathcal{O}(n\log(n)) = \mathcal{O}(n\log(n))$.