

Peaps

Heaps implemented without arrays

Author: Paul Picazo (ppicazo@gmail.com)

Course: CS375 - Algorithms

Professor: Dr. Paul Hriljac

Date Submitted: 4/30/2008

Table of Contents

Introduction	2
The Problem.....	3
Background Information	4
The Solution	5
Locating Nodes, Complex.....	5
Locating Nodes, Simple.....	7
Inserting or Removing.....	8
Conclusion	10
References	11

Introduction

Typically heap data structures have been implemented using arrays. Arrays allow for time efficient operations on the heap. Unfortunately the array implementation does not allow students to properly visualize the heap and its operations. Many sources state that heaps built with pointers are not time efficient or must be threaded. Threaded trees have links between successor and predecessor nodes. They are highly complex data structures and require that the threading be kept current in order to function properly. Storing the threading data hurts the space efficiency, while keeping the threads current destroys the time efficiency.

The purpose of this paper is to explore implementation options which preserve the tree structure of the heap and are time efficient. The goal would not only be to have heaps which help students learn, but also provide alternative time and space efficient.

Two unique implementations have been created and studied. Both offer advantages over traditional array implementations. While they do not achieve the same or better performance in practice as traditional array implementations, they do have the same time efficiency overall for all operations other than locating the next open slot in the heap. This allows for equal time efficiency of heap sort with traditional array implementations. These heaps using the created implementations are called Peaps, named after both their creator, Paul Picazo, and because they use pointers rather than arrays.

Abstract Method	Array Implementation	Pointer Implementation(s)
Find last node	$O(1)$	$O(\log n)$
Insert item into heap	$O(\log n)$	$O(\log n)$
Remove min or max from heap*	$O(\log n)$	$O(\log n)$
Get value of min or max of heap*	$O(1)$	$O(1)$
Heap sort	$O(n \log n)$	$O(n \log n)$

Table 1

The Problem

There are a few critical issues to overcome when implementing heaps without arrays. The most significant issue is locating the last node or next open slot when removing or inserting elements in the heap. Heaps implemented in arrays allow these nodes to be located in $O(1)$ time. Locating these nodes in a threaded tree is also trivial; however, maintaining the threading can be considering part of these operations. Maintaining the threading requires complicated algorithms which severely damage the time efficiency.

If we can solve the problem of efficiently finding the last node or next open slot to insert, all other operations will be straightforward and efficient.

Background Information

In order to keep track of specific locations in the almost perfect binary tree, we will number each location as shown in the Figure 1.

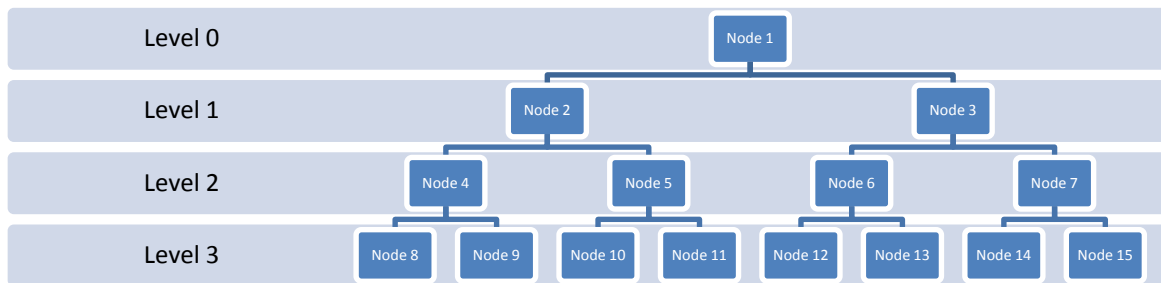


Figure 1: Tree node location and level number system

The abstract properties of the heap data structure are best represented as a binary tree. Comparing the array (shown in Figure 2) and tree (shown in Figure 3) representations clearly shows the advantage for learning and understanding that the tree representation has.

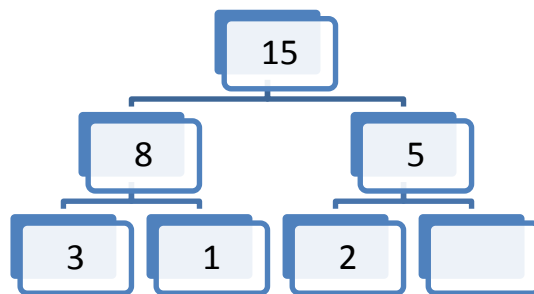


Figure 2: Tree Representation of Example Heap

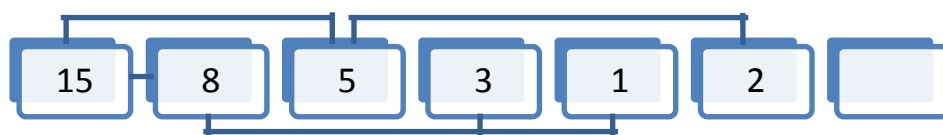


Figure 3: Array Representation of Example Heap

The Solution

To achieve the goal of an efficient heap implemented without arrays we must devise an efficient method for locating specific locations in the heap. The only reference to the tree we have is a pointer to the root node. Each node has two possible children, which may be accessed by a left and right pointer to them. For any children that do not exist, the pointer will point to null.

Locating Nodes, Complex

Two methods were researched, the first method involves calculating the horizontal location in each level a given node is. In the example, when inserting X in to the heap, it is put in location six.

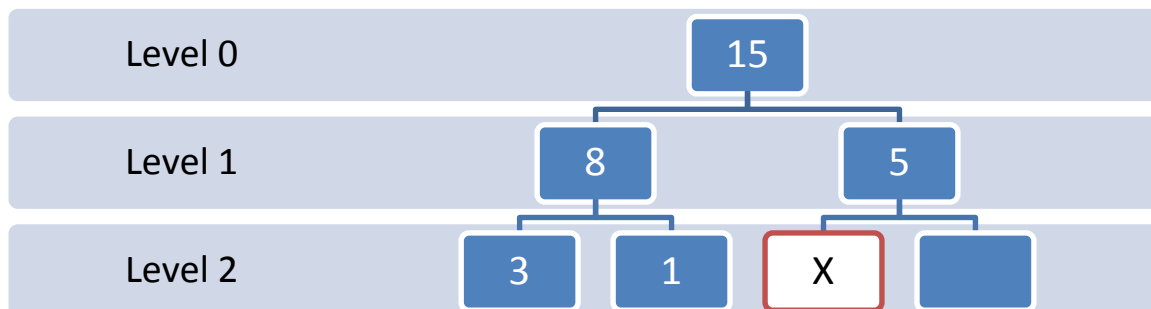


Figure 4: Example Heap

We know how to calculate the depth of any location in an almost perfect binary tree: $\text{level} = \text{floor}(\log_2 \text{location})$. The location six must be on level 2 of the almost perfect binary, because $\text{floor}(\log_2 6) = 2$. The maximum capacity of a given level n is: 2^n . Therefore level 2 has a maximum capacity of four

nodes. The total capacity of a full almost binary tree of n levels is: $2^{(n+1)}-1$. This gives us a maximum capacity of 7 in the example tree.

Level	Max Nodes In Level	Max Nodes in Tree
0	1	1
1	2	3
2	4	7
3	8	15
4	16	31
n	2^n	$2^{n+1}-1$

Table 2

In order to locate the nodes position horizontally we will subtract the node location from the maximum capacity of the tree and then subtract that amount from the maximum capacity of the current level. In our example this gives us: $4-(7-6) = 3$. Therefore our desired node is 75% through the level horizontally. Using this property we can then determine its parent node's horizontal position in its level: Parent Postion in its level = $\text{ceiling}(\text{Child Postion Horizontally in } \% * \text{Maximum nodes in level})$. The will give us: $\text{ceiling}(0.75 * 2) = 2$. If the position in the level is even, then the node is a right child, otherwise it is a left child.

By repeating the process for each level until we reach level zero we can generate the complete traversal path to the desired node. The psuedocode for this is shown Code Listing 1. As it is clearly visible the loop runs once for each level, giving it a time efficiency of $O(\log n)$.

```

$current_level = floor(log($n)/log(2));
$max_nodes_in_level = pow(2,$current_level);
$max_nodes_in_tree = pow(2,$current_level+1)-1;
$spot_in_level = $max_nodes_in_level - ($max_nodes_in_tree - $n);

while($current_level > 0)
{
  if($spot_in_level % 2 == 0)
  {
    array_push($stack,"Right");
  }
  else
  {
    array_push($stack,"Left");
  }
  $percent = $spot_in_level / $max_nodes_in_level;

  $max_nodes_in_level = $max_nodes_in_level / 2;

  $spot_in_level = ceil($percent * $max_nodes_in_level);

  $current_level--;
}

while(count($stack) > 0)
{
  go(array_pop($stack)); //go left or right
}

```

Code Listing 1

Locating Nodes, Simple

The second method is similar to the first method in that it incrementally builds the traversal path to the desired location. It differs however in how it calculates that path. After observing almost perfect binary trees, a clear pattern emerged. This pattern is also used when building heaps in arrays. Each parent nodes location is half of either of its children's locations.

Using the example heap, we can see that the parent of location 6 is at location 3. Once again if we repeat this process until we reach level zero we can determine which nodes are in the traversal path.

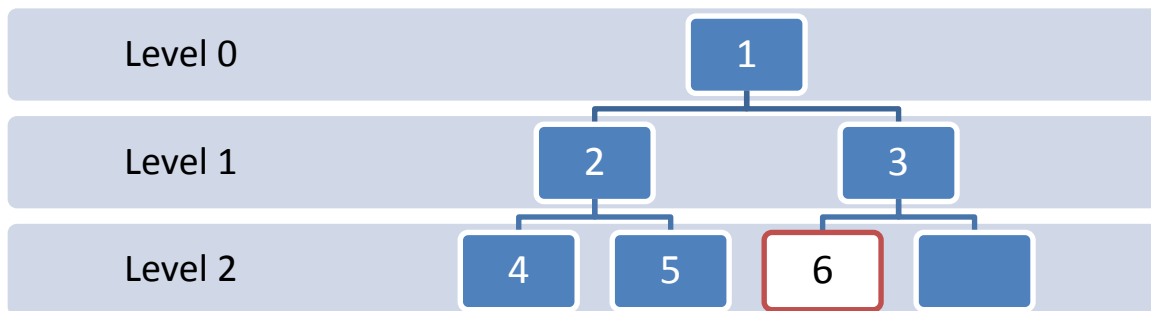


Figure 5

This alone does not allow us to traverse that path through the tree. But another pattern emerges when examining the tree in Figure 5. All even locations are left children and all odd locations are right children. This algorithm (Code Listing 1) is also $O(\log n)$.

```
function find_node($n)
{
    $current_node = $n;
    while($current_node > 1)
    {
        if($current_node % 2 == 1) // if node is odd it is a right child
        {
            push($stack, "Right");
        }
        else // otherwise it is even and left child
        {
            push($stack, "Left");
        }

        $current_node = floor($current_node / 2); // set the current node to the parent
    }
    return $stack; // this stack now contains the path to node n
}
```

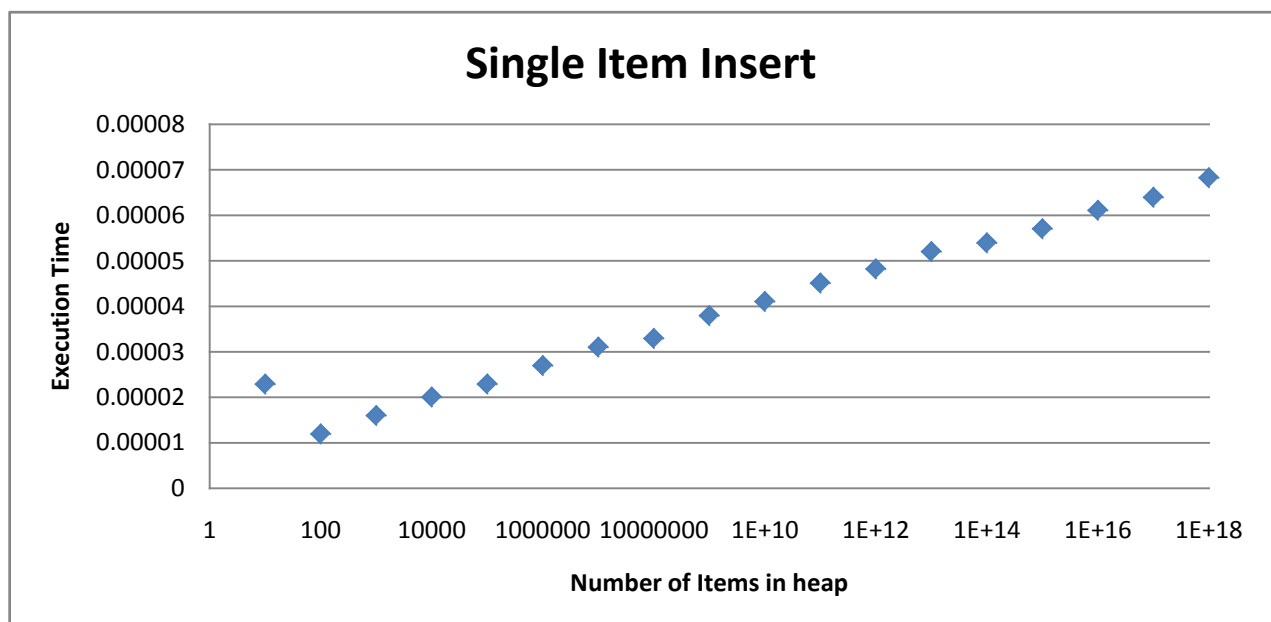
Code Listing 2

Inserting or Removing

In practice when we insert or remove from a heap, we must also bubble up or sift down. Both of these processes are $O(\log n)$. The array implementation allows the correct node to be found in $O(1)$, but the bubbling up or sifting down will make the overall time efficiency of the insert or remove $O(\log n)$. The peap implementation allows for the necessary node to be found in $O(\log n)$ time, making the overall time efficiency of the insert into a peap $O(\log n)$.

Nodes	Execution Time	Rate
10	0.000022888	436906.7
100	0.000011921	8388608
1000	0.000015974	62601552
10000	0.000020027	4.99E+08
100000	0.000022888	4.37E+09
1000000	0.000026941	3.71E+10
10000000	0.000030994	3.23E+11
100000000	0.000032902	3.04E+12
1000000000	0.000037909	2.64E+13
10000000000	0.000041008	2.44E+14
1E+11	0.000045061	2.22E+15
1E+12	0.000048161	2.08E+16
1E+13	0.000051975	1.92E+17
1E+14	0.000053883	1.86E+18
1E+15	0.000056982	1.75E+19
1E+16	0.000061035	1.64E+20
1E+17	0.000063896	1.57E+21
1E+18	0.000068188	1.47E+22

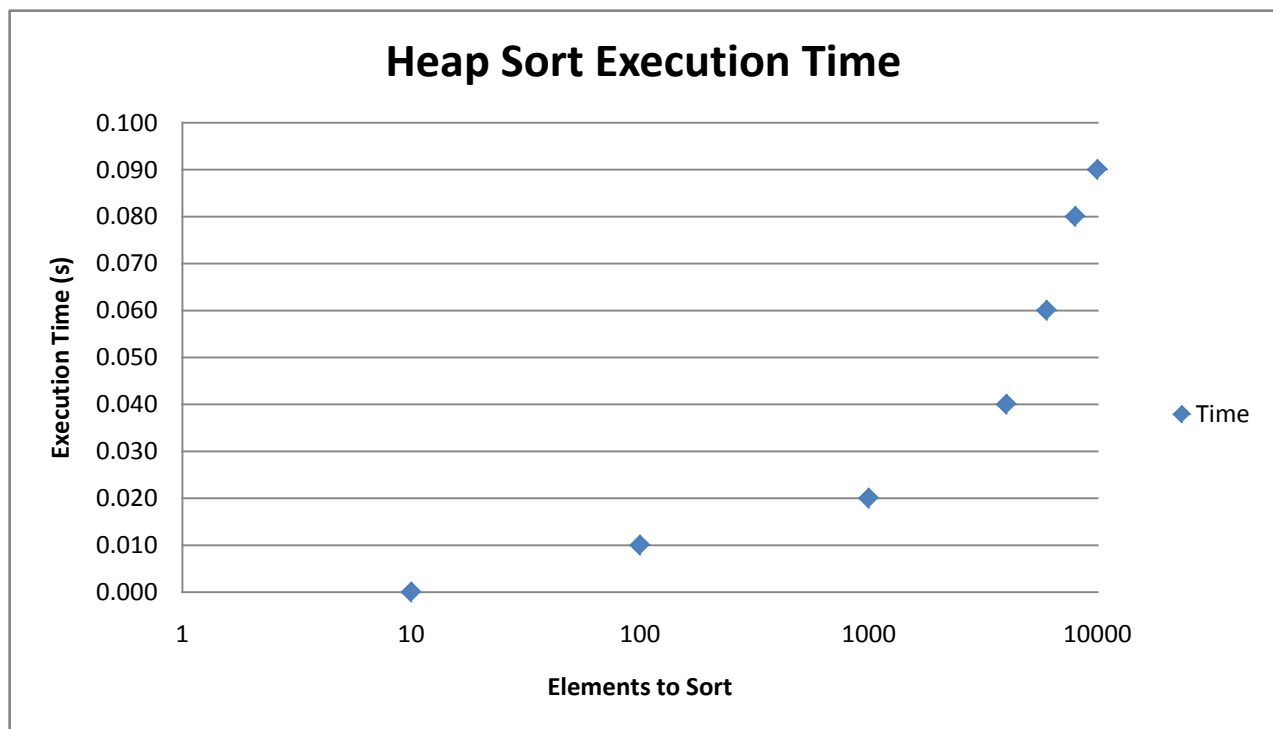
In summary the steps required to insert in to a peap are: find the path to the necessary node, traverse to that node, and sift up to restore the heap property. Each of these steps is clearly $O(\log n)$.



Heap Sort

Given that we can now do heap inserts with the same time efficiency as heaps implemented with arrays, we can also now achieve the same time efficiency with heap sort. We insert n items, and remove n items, resulting in $O(n \log n)$ behavior.

Nodes	Time	N / S
10	0.000	
100	0.010	10000
1000	0.020	50000
4000	0.040	100000
6000	0.060	100000
8000	0.080	100000
10000	0.090	111111



Conclusion

With acceptable performance in both theory and practice, this implementation should be considered when either using heaps or teaching students about heaps. An alternative method allows for more flexibility for programmers. Asymptotically they have the same efficiency as traditional array implementations while it they are visually more appealing and intuitive. Storage space is minimized because there is no need for threaded trees or links to parent nodes.

Many scenarios would benefit from a heap implemented with pointers, especially those which use a programming language that does not allow arrays to be resized. In these cases a new segment of memory must be allocated with the new size and the old array must be copied.

By keeping the abstract properties of a binary tree when implementing heaps, visually and intuitively we can allow computer science students to achieve their full potential.

References

Matthew, Jaffe. Lectures from CS315 at Embry-Riddle, Prescott Campus.