

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:



Ask programming questions



Answer and help your peers



Get recognized for your expertise

## Implement Heap using a Binary Tree

Work on work you love. From home.



This question has been asked before in Stack Exchange but it went unanswered.

Link to the previously asked question: [Binary Heap Implemented via a Binary Tree Structure](#)

How do I implement heap in a binary tree. To implement heap, it is important to know the last filled node and the first unoccupied node. This could be done in level ordering of the tree, but then the time complexity will be  $O(n)$  just to find the first unoccupied node. So, how to implement heap in a binary tree in  $O(\log n)$ ?

Thanks Shekhar

java heap binary-tree

asked Aug 14 '13 at 20:02



[user2200660](#)

335 1 4 13

1 It has been answered. What's wrong with the given answer? – [hexafraction](#) Aug 14 '13 at 20:04

The answer doesn't mention how to find first unoccupied leaf, it just mentions we need to add the new element in the first unoccupied leaf. To my understanding, you need to level order the tree to find the next unoccupied leaf and that will take  $O(n)$  – [user2200660](#) Aug 14 '13 at 20:07

As far as I can see you're basically keeping track of it by storing it. – [hexafraction](#) Aug 14 '13 at 20:09

Yes right, I tried to code it. The problem is, if you don't keep a pointer to the parent, then it's a problem to keep track of the next unoccupied leaf. We can maintain a variable to store this info, but calculating this will take  $O(n)$ . Suppose we are in the 4th level (root is 0) and we have 4 elements starting from left in the 4th level. Now, to get next unoccupied leaf, we need to get the sibling of 2nd level, means go to 1st level parent. This takes  $O(n)$  because in a way we are doing level ordering. – [user2200660](#) Aug 14 '13 at 20:14

## 6 Answers

To implement a heap with a binary tree with  $O(\log n)$  time complexity, you need to store the total number of nodes as an instance variable.

Suppose we had a heap of 10 total nodes.

If we were to add a node...

We increment the total number of nodes by one. Now we have 11 total nodes. We convert the new total number of nodes (11) to its binary representation: 1011.

With the binary representation of the total nodes (1011), we get rid of the first digit. Afterwards, we use 011 to navigate through the tree to the next location to insert a node in. 0 means to go left and 1 means to go right. Therefore, with 011, we would go left, go right, and go right...which brings us to the next location to insert in.

We examined one node per level, making the time complexity  $O(\log n)$

Learned this in my programming class :-).

answered Jan 27 '14 at 21:02

Solace

736 ● 5 ● 13

2 That's pretty neat. If you cannot get a binary representation of the number, you can still determine (starting from the root) whether to go left or right. You know how many levels the tree has:  $\text{level} = \text{floor}(\log(11)/\log(2)) = 3$ ; You know the offset from the left most element in this level:  $\text{offset} = 11 - (2^{\text{level}} - 1)$ ; And how many nodes there can maximally be at this level:  $\text{max} = 2^3 = 8$ ; If the offset is less than half of max, then you are to be in the left subtree, if more than half, go right. As you go down, you update the level and offset and done! – Bartel May 4 '14 at 20:23

1 @Bartel +1 that's pretty neat too! Did you either of you learn this from [cpp.edu/~ftang/courses/CS241/notes/...](http://cpp.edu/~ftang/courses/CS241/notes/...) ? Just curious – xlm Mar 29 '15 at 5:19

Try it free for 30 days  
Get \$200 in open source-friendly cloud.

SIGN ME UP →

You won't implement the heap *IN* binary tree, because the heap is *A* binary tree. The heap maintains the following order property - given a node  $V$ , its parent is greater or equal to  $V$ . Also the heap is complete [binary tree](#). I had ADS course at uni so I will give you my implementation of the heap in Java later in the answer. Just to list the main methods complexities that you obtain:

- `size()`  $O(1)$
- `isEmpty()`  $O(1)$
- `insert()`  $O(\log n)$
- `removeMin()`  $O(\log n)$
- `min()`  $O(1)$

Here is my `Heap.java` file:

```
public class Heap<E extends Comparable<E>> {

    private Object S[];
    private int last;
    private int capacity;

    public Heap() {
        S = new Object[11];
        last = 0;
        capacity = 7;
    }

    public Heap(int cap) {
        S = new Object[cap + 1];
        last = 0;
        capacity = cap;
    }

    public int size() {
        return last;
    }

    //
    // returns the number of elements in the heap
    //

    public boolean isEmpty() {
        return size() == 0;
    }

    //
    // is the heap empty?
    //

    public E min() throws HeapException {
        if (isEmpty())
            throw new HeapException("The heap is empty.");
        else
            return (E) S[1];
    }

    //
    // returns element with smallest key, without removal
    //

    private int compare(Object x, Object y) {
        return ((E) x).compareTo((E) y);
    }

    public void insert(E e) throws HeapException {
        if (size() == capacity)
            throw new HeapException("Heap overflow.");
    }
}
```

```

        else{
            last++;
            S[last] = e;
            upHeapBubble();
        }
    }

    // inserts e into the heap
    // throws exception if heap overflow
    //

    public E removeMin() throws HeapException {
        if (isEmpty())
            throw new HeapException("Heap is empty.");
        else {
            E min = min();
            S[1] = S[last];
            last--;
            downHeapBubble();
            return min;
        }
    }

    //
    // removes and returns smallest element of the heap
    // throws exception is heap is empty
    //

    /**
     * downHeapBubble() method is used after the removeMin() method to reorder the
     * elements
     * in order to preserve the Heap properties
     */
    private void downHeapBubble(){
        int index = 1;
        while (true){
            int child = index*2;
            if (child > size())
                break;
            if (child + 1 <= size()){
                //if there are two children -> take the smalles or
                //if they are equal take the Left one
                child = findMin(child, child + 1);
            }
            if (compare(S[index],S[child]) <= 0 )
                break;
            swap(index,child);
            index = child;
        }
    }

    /**
     * upHeapBubble() method is used after the insert(E e) method to reorder the elements
     * in order to preserve the Heap properties
     */
    private void upHeapBubble(){
        int index = size();
        while (index > 1){
            int parent = index / 2;
            if (compare(S[index], S[parent]) >= 0)
                //break if the parent is greater or equal to the current element
                break;
            swap(index,parent);
            index = parent;
        }
    }

    /**
     * Swaps two integers i and j
     * @param i
     * @param j
     */
    private void swap(int i, int j) {
        Object temp = S[i];
        S[i] = S[j];
        S[j] = temp;
    }

    /**
     * the method is used in the downHeapBubble() method
     * @param leftChild
     * @param rightChild
     * @return min of left and right child, if they are equal return the Left
     */
    private int findMin(int leftChild, int rightChild) {
        if (compare(S[leftChild], S[rightChild]) <= 0)
            return leftChild;
        else
            return rightChild;
    }

    public String toString() {
        String s = "[";
        for (int i = 1; i <= size(); i++) {

```

```

        s += S[i];
        if (i != last)
            s += ",";
    }
    return s + "];"
}
//
// outputs the entries in S in the order S[1] to S[last]
// in same style as used in ArrayQueue
//
}

```

HeapException.java:

```

public class HeapException extends RuntimeException {
    public HeapException();
    public HeapException(String msg){super(msg);}
}

```

The interesting part that gives you  $O(\log n)$  performance is the `downHeapBubble()` and `upHeapBubble()` methods. I will add good explanation about them shortly.

`upHeapBubble()` is used when inserting new node to the heap. So when you insert you insert in the last position and then you need to call the `upHeapBubble()` like that:

```

last++;
S[last] = e;
upHeapBubble();

```

Then the last element is compared against its parent and if the parent is greater - swap: this is done max  $\log n$  times where  $n$  is the number of nodes. So here comes the  $\log n$  performance.

For the deletion part - you can remove only min - the highest node. So when you remove it - you have to swap it with the last node - but then you have to maintain the heap property and you have to do a `downHeapBubble()`. If the node is greater than its child swap with the smallest one and so on until you don't have any children left or you don't have smaller children. This can be done max  $\log n$  times and so here comes the  $\log n$  performance. You can explain yourself why this operation can be done max  $\log n$  times by looking in the binary tree pictures [here](#)

edited Mar 11 '14 at 12:26

answered Aug 14 '13 at 20:23



Anton Belev

1,659 ● 7 ● 25 ● 52

Anton, I know the implementation of heap using arrays. I was interested in tree implementation. bdw thanks for the answer. – [user2200660](#) Aug 14 '13 at 20:27

I'll come to @Anton Belev's defense. S is what I would call an array-based implementation of a binary tree. Each node (array element) has a left and right child which happen to be found by formula and not by pointer, but I don't think this defines what a binary tree is. Even if I produced a suggestion for the other case, I think the OP should have been more explicit. – [Mario Rossi](#) Aug 14 '13 at 21:06

Why do you reserve an array of 11 slots by default? – [Nick L.](#) Mar 4 at 15:37

## HEAP IMPLEMENTATION USING TREE

I am answering my own question that takes  $O(\log n)$ , but the limitation is to keep a pointer to the parent. if we don't keep a pointer to the parent, we need approximately  $O(n)$ . I posted this question to get a solution for  $O(\log n)$

Here are the steps to calculate next unoccupied leaf (we have a pointer to the parent node):

```

x = last inserted node. We save this after every insertion.
y = tmp node
z = next unoccupied node (next insertion)
if x is left child
    z = x -> parent -> rightchild (problem solved.. that was easy)
else if x is right child
    go to x's parent, until parent becomes left child. Let this node be y
    (subtree rooted at y's sibling will contain the next unoccupied node)
    z = y -> parent -> right -> go left until null

```

This is  $O(\log n)$ , but needs a pointer to the parent.

$O(n)$  solution would be pretty easy, just level order the tree and we get the location of the next unoccupied node.

My question is: how to locate next unoccupied node in  $O(\log n)$  without using a parent pointer.

Thanks.

edited Jun 12 '14 at 13:56

answered Aug 14 '13 at 20:36

Shreedhan Shrestha  
70 ● 1 ● 1 ● 8

user2200660  
335 ● 1 ● 4 ● 13

I am sorry for the formatting. I did cntrl k to format it and it became like this. — user2200660 Aug 14 '13 at 20:38

Assuming you want to use a **linked** binary tree, with no pointers to parent nodes, then the only solution I can think of is keeping a counter of number of children in each node.

```
availableLeaf(node) {  
    if( node.left is Empty || node.right is Empty )  
        return node ;  
    else  
        if( node.left.count < node.right.count )  
            return availableLeaf(node.left)  
        else  
            return availableLeaf(node.right)  
}
```

This strategy also balances the number of nodes on each side of each subtree, which is beneficial (though extremely slightly).

This is  $O(\log n)$ . Keeping track of count on insertion requires to come all the way up to the roof, but this doesn't change the  $O(\log n)$  nature of this operation. Similar thing with deletion.

Other operations are the usual, and preserve their performance characteristics.

Do you need the details or prefer to work them out by yourself?

If you want to use a linked binary tree, with no other information than left and right pointers, then I'd suggest you to initiate a bounty for at least 100,000 points. I'm not saying it's impossible (because I don't have the math to prove it), but I'm saying that this has not been found in several decades (which I do know).

edited Aug 14 '13 at 21:02

answered Aug 14 '13 at 20:24

Mario Rossi  
5,423 ● 10 ● 30

My implementation of heap


```
public class Heap <T extends Comparable<T>> {  
    private T[] arr;  
    private int size;  
  
    public Heap(T[] baseArr) {  
        this.arr = baseArr;  
        size = arr.length - 1;  
    }  
  
    public void minHeapify(int i, int n) {  
        int l = 2 * i + 1;  
        int r = 2 * i + 2;  
  
        int smallest = i;  
        if (l <= n && arr[l].compareTo(arr[smallest]) < 0) {  
            smallest = l;  
        }  
        if (r <= n && arr[r].compareTo(arr[smallest]) < 0) {  
            smallest = r;  
        }  
  
        if (smallest != i) {  
            T temp = arr[i];  
            arr[i] = arr[smallest];  
            arr[smallest] = temp;  
            minHeapify(smallest, n);  
        }  
    }  
  
    public void buildMinHeap() {  
        for (int i = size / 2; i >= 0; i--) {  
            minHeapify(i, size);  
        }  
    }  
  
    public void heapSortAscending() {  
        buildMinHeap();  
        int n = size;  
        for (int i = n; i >= 1; i--) {  
            T temp = arr[0];  
            arr[0] = arr[i];  
            arr[i] = temp;  
        }  
    }  
}
```

```

        n--;
        minHeapify(0, n);
    }
}

```

answered May 6 '14 at 21:19


[Puneet Jaiswal](#)  
 86 ● 2 ● 7

The binary tree can be represented by an array:

```

import java.util.Arrays;

public class MyHeap {
    private Object[] heap;
    private int capacity;
    private int size;

    public MyHeap() {
        capacity = 8;
        heap = new Object[capacity];
        size = 0;
    }

    private void increaseCapacity() {
        capacity *= 2;
        heap = Arrays.copyOf(heap, capacity);
    }

    public int getSize() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public Object top() {
        return size > 0 ? heap[0] : null;
    }

    @SuppressWarnings("unchecked")
    public Object remove() {
        if (size == 0) {
            return null;
        }
        size--;
        Object res = heap[0];
        Object te = heap[size];
        int curr = 0, son = 1;
        while (son < size) {
            if (son + 1 < size
                && ((Comparable<Object>) heap[son + 1])
                    .compareTo(heap[son]) < 0) {
                son++;
            }
            if (((Comparable<Object>) te).compareTo(heap[son]) <= 0) {
                break;
            }
            heap[curr] = heap[son];
            curr = son;
            son = 2 * curr + 1;
        }
        heap[curr] = te;
        return res;
    }

    @SuppressWarnings("unchecked")
    public void insert(Object e) {
        if (size == capacity) { // auto scaling
            increaseCapacity();
        }
        int curr = size;
        int parent;
        heap[size] = e;
        size++;
        while (curr > 0) {
            parent = (curr - 1) / 2;
            if (((Comparable<Object>) heap[parent]).compareTo(e) <= 0) {
                break;
            }
            heap[curr] = heap[parent];
            curr = parent;
        }
        heap[curr] = e;
    }
}

```

Usage:

```
MyHeap heap = new MyHeap(); // it is a min heap
heap.insert(18);
heap.insert(26);
heap.insert(35);
System.out.println("size is " + heap.getSize() + ", top is " + heap.top());
heap.insert(36);
heap.insert(30);
heap.insert(10);
while(!heap.isEmpty()) {
    System.out.println(heap.remove());
}
```

answered Jun 14 '15 at 5:35



[coderz](#)

1,342 ● 1 ● 10 ● 27

---