Binary Heaps

Introduction

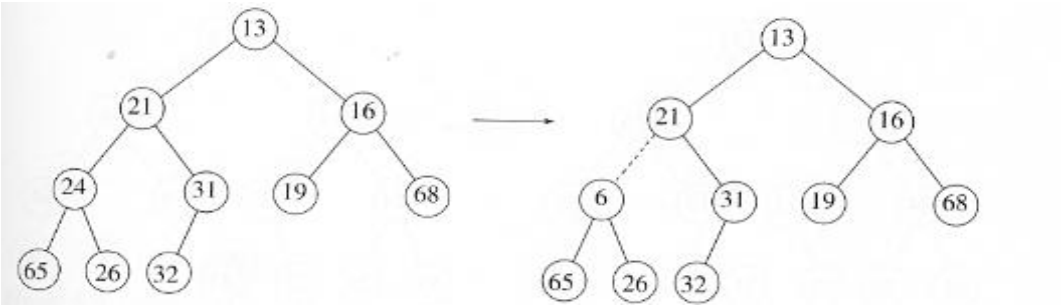
In most sorting algorithms the elements to be sorted are assumed fixed and entirely known before the sorting takes place. However, quite often the elements to be sorted are not known beforehand, and may present themselves in a dynamic manner. In this lecture we look at the binary heap, which is a data structure that is capable of dynamically sorting elements as they become available.

To define a binary heap, we first define the notion of a complete binary tree. Given a binary tree, assign to each of its nodes an index i. To start, assign the root an index $i = 1$. Now suppose a node has been assigned an index i. Then assign its left child (if it exists) the index $2i$, and assign its right child (if it exists) the value $2i + 1$. Then we say that the tree is **complete** iff the set of assigned indices is a **contiguous** set; meaning that, if i and k are assigned indices, with $i < k$, then so is j, for every $i < j < k$.

Example 1. Give examples of both complete and incomplete binary trees.

A **min heap** (respectively, **max heap**) is a complete binary tree T whose nodes store comparable elements of some type, such as integers, real numbers, etc.. Moreover, if $n \in T$, and $c \in T$ is a child of n, then the element stored at n is less than (respectively, greater than) or equal to the element stored at c. Another name for a min heap is that of **priority queue**. Note also that, by the property of being complete, a heap may also be viewed as a contiguous array of elements. This observation can be used to implement heaps using an array structure.

Two complete trees, one of which is a min heap:



Operations on Min Heaps

- **insert():** insert an element into the heap.
- **pop():** pop the least element from the heap.
- **build_heap:** build a heap from a list of elements.

More advanced operations include merge(), which merges two heaps. This operation requires more advanced heaps.

Example 2. Insert the following nodes into an initally empty min heap: 12,5,15,9,13,7,15,10,3,20,4.

Psuedocode for inserting into a Heap of Integers

```
void insert(Heap heap, int x)
{
    int array[ ] = heap.array;

    //percolate up
    int hole = heap.load; //number of elements in heap

    for(; hole > 1 && x < array[hole/2]; hole /= 2)
         array[hole] = array[hole/2];

    array[hole] = x;
    heap.load++;
}
```

Theorem 1. The insertion of n elements into an initally empty heap will take $O(n \log n)$ steps in the worst-case.

Proof. In the worst case, the i th element will require at most $\mathrm{O}(\lceil \log(\mathrm{i}) \rceil)$ percolation steps up the tree. Thus, the running time

$$T(n) \leq \sum_{i=1}^{n} c \cdot \lceil \log(i) \rceil = \mathrm{O}(\mathrm{n} \log \mathrm{n}).$$

Example 3. For the heap constructed in Example 2, successively pop the minimum element from the heap until it is empty.

Psuedocode for Popping a Heap of Integers

```
int pop(Heap heap)
{
    int array[ ] = heap.array;
    int x = array[1]; //x is the element to be returned
    array[1] = array[heap.load--];
    percolate_down(heap,1);
    return x;
}

//Percolate the element located at array[index] down the tree
void percolate_down(Heap heap, int index)
{
    int array[ ] = heap.array;
    int load = heap.load;
    int orphan = array[index];
    int child_index;

    for(; (child_index=2*index) <= load; index = child_index)
    {
        if(child_index != load && array[child_index+1] < array[child_index])
            child_index++;

        if(array[child_index] < orphan)
            array[index] = array[child_index];
        else
            break;
    }

    array[index] = orphan; //found a home for orphaned integer
}
```

Psuedocode for Building a Heap of Integers

```
Heap build_heap(int array[ ], int load)
{
    Heap heap = new_Heap(array, load);
    int i;

    for(i= load/2; i>0; i--)
        percolate_down(heap, i);

    return heap;
}
```

Example 4. Repeat Example 1, but now use `build_heap()`.

Lemma 1. For a perfect binary tree of height h, the sum of the heights of the nodes is

$$2^{h+1} - 1 - (h+1).$$

Proof. At depth i, there are 2^i nodes, and the height of each of those nodes is $h - i$. Hence, letting S denote the sum of the heights of all nodes,

$$S = \sum_{i=0}^{h} 2^i (h-i).$$

Multiplying both sides by 2 yields

$$2S = \sum_{i=0}^{h} 2^{i+1} (h-i).$$

Now subtracting the first equation from the second yields

$$S = 2^h + 2^{h-1} + \cdots + 2 - h = 2^h + 2^{h-1} + \cdots + 2 + 1 - (h+1) =$$

$$2^{h+1} - 1 - (h+1).$$

Theorem 2. The `build_heap()` operation takes $O(n)$ steps.

Proof of Theorem 2. The number of steps needed for `build_heap()` is proportional to the total number of times that elements must be swapped down the binary heap. This number is in turn bounded by the sum of the heights of the nodes which, by Lemma 1, is bounded by

$$2^{h+1} - h - 2.$$

Finally, given that $h = \lfloor \log n \rfloor$, we see that the bound on the number of steps is in fact linear in n.

Heap Sort

`HeapSort` is a sorting algorithm that sorts an array in place. It accomplishes this by building a max heap from the given array of elements. It then successively calls the `pop()` operation and places the returned elements one-by-one starting from the end of the array, to the beginning of the array.

Example 5. Perform `HeapSort` using the data from Example 1.

Exercises.

1. Where in a min heap of integers might the largest element reside (assuming all elements are distinct)? Explain.

2. Insert integers 5,3,17,10,85,2,19,6,22,4 one-by-one into an initially-empty min heap. Re-draw the heap each time an insertion causes one or more swaps.

3. Repeat the previous problem, but now use the `build_heap` algorithm. Redraw the heap each time a call to `percolate_down` causes one or more swaps.

4. For the binary heap of Exercise 1, show the result of performing four consecutive `pop` operations. Re-draw the heap after each pop.

5. Use induction to prove that $1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$. Conclude that $2^{h+1} - 1$ is the maximum number of elements that can be stored in a binary heap having height h.

6. Provide the minimum and maximum number of elements that can be stored in a binary heap that has height h. Show work and explain.

7. Is the array with values 23, 17, 14, 6, 13, 20, 10, 11, 5, 7, 12 a max-heap? If not, convert it to one using the buildheap algorithm.

8. Prove that a binary heap with n nodes has exactly $\lceil n/2 \rceil$ leaves.

9. Give an example which shows that the buildheap algorithm does not work if one begins percolating down with the first internal node, rather than the last internal node.

10. Prove that a binary heap with n elements has height $\lfloor \log n \rfloor$.

11. Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

12. Suppose that instead of binary heaps, we wanted to work with ternary heaps. Suggest an appropriate indexing scheme so that a complete tree will yield a contiguous sequence. Hint: Let the root have an index of 0. Demonstrate your indexing scheme for a complete ternary tree of size 12.

13. Prove that the worst-case running time of `HeapSort` is $O(n \log n)$.

Exercise Hints and Answers.

1. Any leaf.
2. Final Heap: 2,4,3,6,5,17,19,10,22,85
3. Final Heap: 2,3,5,6,4,17,19,10,22,85
4. Final Heap: 6,10,17,19,22,85,19
5. Inductive assumption: $1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$, for some $h \geq 0$. Show that $1 + 2 + 4 + \cdots + 2^{h+1} = 2^{h+2} - 1$.

$$1 + 2 + 4 + \cdots + 2^{h+1} = (1 + 2 + 4 + \cdots + 2^h) + 2^{h+1} =$$

$$2^{h+1} - 1 + 2^{h+1} = 2 \cdot 2^{h+1} - 1 = 2^{h+2} - 1,$$

   where the third to last equality uses the inductive assumption.
6. Minimum: 2^h, maximum: $2^{h+1} - 1$.
7. No. After using build heap, final heap: 23,17,20,11,13,14,10,6,5,7,12.
8. If a binary heap has n nodes, then the last internal node has index $\lfloor n/2 \rfloor$ (why?). Hence there are $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ leaves.
9. Use the array $a = 4, 2, 3, 5, 6, 1, 7$
10. Use the results of Exercise 6.
11. Call the original heap H_0. By Exercise 8, H_0 has $\lceil n/2^1 \rceil$ leaves (i.e. nodes with height $h = 0$). So it is true for $h = 0$. If these leaves are removed from H_0, then the resulting tree, call it H_1, is itself a heap, and whose leaves are the nodes that have height 1 in the orginal tree. Then H_1 has $\lfloor n/2 \rfloor$ nodes, and so it has $\lceil (\lfloor n/2 \rfloor)/2 \rceil \leq \lceil n/2^2 \rceil$ leaves. So the result is true for $h = 1$. Now remove these leaves from H_1 to produce yet another heap H_2 whose leaves are the nodes of the original tree that have height $h = 2$. Argue that H_2 as no more than $n/2^2$ nodes, and again apply Exercise 8. In general, prove that the k th heap, H_k, has no more than $n/2^k$ nodes, and that these nodes correspond with the nodes of height k in H_0. Then apply Exercise 8.
12. Let the root have index 0. Then the children of a node with index i, will be $3i + 1$, $3i + 2$, and $3i + 3$.
13. Assuming an array of distinct integers, the worst case is $\mathrm{O}(n \log n)$ since the i th pop forces the last element in the heap to percolate down to a leaf that has a worst-case depth of $\lfloor \log(n - i) \rfloor$. Hence, the running time has worst case

$$\sum_{i=0}^{n-1} \lfloor \log(n - i) \rfloor = \Omega(n \log n)$$

   by applying the Integral Theorem to the series $\sum_{i=1}^{n} \log i$.