[Sorting]

Design and Analysis of Algorithms (CSc 523)

Samujjwal Bhandari
Central Department of Computer Science and Information Technology (CDCSIT)
Tribhuvan University, Kirtipur,
Kathmandu, Nepal.

We talk about sorting very much and in every part of our study of computer science sorting is studied more than others. The reasons for this are sorting is the most fundamental operation that is done most by the computer, the sorting algorithms are widely studied problem and different varieties of algorithms are known. Most of the ideas for algorithm design and analysis can be obtained from sorting algorithms like divide and conquer, lower bounds etc.

Applications of Sorting:

Sorting being the fundamental operation in computer science has many applications.

Searching:

Searching speeds up by the use of sorted elements. Binary search is an example that takes sorted sequence of keys to search and the whole searching operation is done in O(log n) time.

Closest pair:

Given *n* numbers, we have to find the pair which are closest to each other.Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an O(n) linear scan completes the job.

Element uniqueness:

Given n elemnts, are there any duplicate values for all elements or they are unique. When the elements are sorted we can check every adjacent elements by linear searching.

Frequency Distribution:

Given n numbers of element, when the elements are sorted we can linearly search the elements to calculate the number of times the elements repeats. This can be applied for determining Huffman codes for each letters.

Median and selection:

Given n elements, to find kth largest or smallest element we sort the elements and find it in constant time (array is used).

Convex hulls:

Given n numbers of points find the smallest area polygon such that every points is within that polygon. When elements are sorted we can improve the efficiency.

Selection Sort

In selection sort the all the elements are examined to obtain smallest (greatest) element at one pass then it is placed into its position, this process continues until the whole array is sorted.

Algorithm:

```
Selection-sort(A)
{
        for( i = 0;i < n ;i++)
                for ( j = i + 1;j < n ;j++)
                        if (A[j] < A[i])
                                swap(A[i],A[j])
}
```

From the above algorithm it is clear that time complexity is $O(n^2)$. We can also see that for every instances of input complexity is $O(n^2)$.

Bubble Sort

The bubble sort algorithm Compare adjacent elements . If the first is greater than the second, swap them. This is done for every pair of adjacent elements starting from first two elements to last two elements.here the last element is the greatest one. The whole process is repeated except for the last one so that at each pass the comparision becomes fewer.

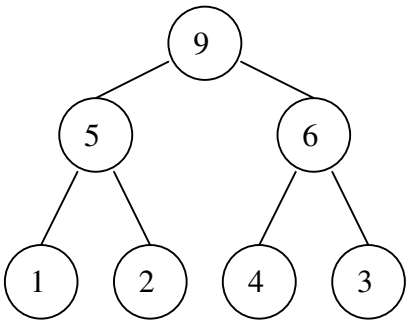
Algorithm:

```
BubbleSort(A, n)
{
 for(i = n - 1; i > 0; i--)
   for(j = 0; j < i; j++)
     if(A[j] > A[j+1])
{
       temp = A[j];
       A[j] = A[j+1];
       A[j+1] = A;
     }
}
```

The above algorithm for any instances of input runs in $O(n^2)$ time. This algorithm is similar to insertion sort algorithm studied earlier but operates in reverse order. However there is no best-case linear time complexity for this algorithm as of insertion sort.

Heap Sort

We studied about the heap data structure. Here we present the implementation of the heap in sorting called heap sort. We can consider heap as max heap or min heap. Consider the heap below:

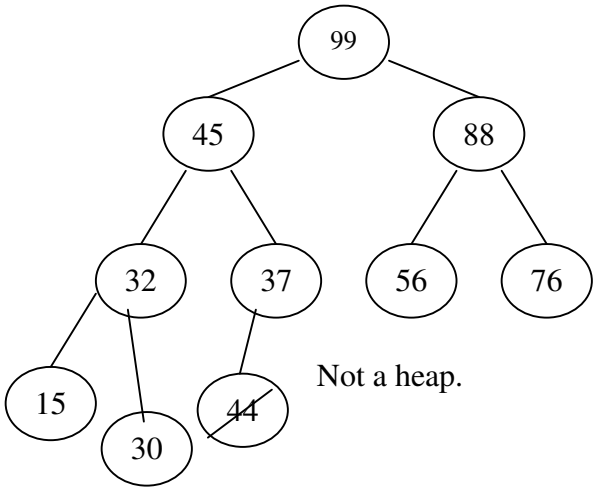


We represent the above max heap as an array like:

1	2	3	4	5	6	7
9	5	6	1	2	4	3

Remember if parent p is at A[i] place then its left child is at A[2i] and the right child is at A[2i + 1]

Is the sequence 99, 45, 88, 32, 37, 56, 76, 15, 30, 44 a max heap?



Constructing Heaps (max)

To create a heap, insert an element into the leftmost empty slot of an array if the inserted element is larger than its parent then swap the elements and recur. At every step we swap the element with parent to maintain the heap order.

All the levels but the last one are filled completely so height (h) of n elements is bounded as

$$\sum_{i=1}^{h} 2^i = 2^{h+1} - 1 >= n, \text{ so } h = \lfloor \log n \rfloor$$

Doing n such insertion takes Θ(nlogn).

Parent(i)

return $\lfloor i/2 \rfloor$

Left(i)

return 2i

Right(i)

return 2i+1

Building a Heap

The bottom up procedure for constructing heap is a good way but there is a process that uses merge method to create a heap called heapify. Here giving the two heaps and new element those two heaps are merged for single root.

Algorithm:

```
BuildHeap(A)
{
    heapsize[A] = length[A]
    for i = ⌊length[A]/2⌋ to 1
        do Heapify(A,i)
}
```

In the above algorithm the first step takes constant time and the second and third steps run about n/2 so the time complexity is *Complexity of heapify * n/2*. see analysis below.

Algorithm:

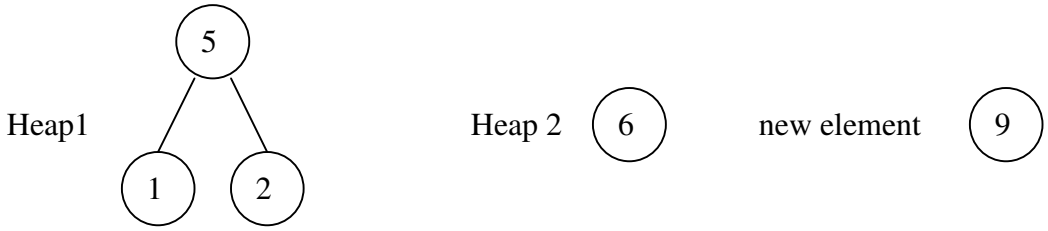
```
Heapify(A,i)
{
        l = left(i)
        r = Right(i)
        if l <= heapsize[A] and A[l] > A[i]
                max = l
        else
                 max = i
        if r <= heapsize[A] and A[r] > A[max]
                max = r
        if max != i
        {
                swap(A[i],A[max])
                Heapify(A,max)
        }
}
```

Analysis :

Time complexity for *Heapify(A,i)* for n node tree is given by

$T(n) <= T(2n/3) + O(1)$

Here 2/3 represents for worst case where the two heaps that are getting merged differ by level 1. i.e. the last level is exactly half filled.



What is the resulting heap after merging? See the number of elements in last level.

Use master method on the above recurrence relation where a = 1, b =3/2, f(n) = O(1) and $\log_{3/2} 1 = 0 = O(1)$.

Observe that it gives **case 2** so the solution is Θ(logn).

So the building of heap is Θ(nlogn). This bound is asymptotic but not tight.

Exact Analysis:

In full binary tree of n nodes there are n/2 nodes that are leaves, n/4 nodes that have height 1, n/8 nodes of height 2, … and so on so heapify acts on small heaps for many times.

So there are at most $\lceil n/2^{h+1} \rceil$ nodes at height h, so cost of building heap is

$$\sum_{h=0}^{\log n} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\log n} h/2^h)$$

see above that the series is not quite geometric. However the series converges so that there is some ending point.

Proving convergence:

We know

$$\sum_{i=0}^{\infty} x^k = 1/1\text{-}x \text{ for } 0< x < 1$$

Taking derivatives of both sides

$$\sum_{i=0}^{\infty} kx^{k-1} = 1/(1\text{-}x)^2.$$

Multiplying by x on both sides

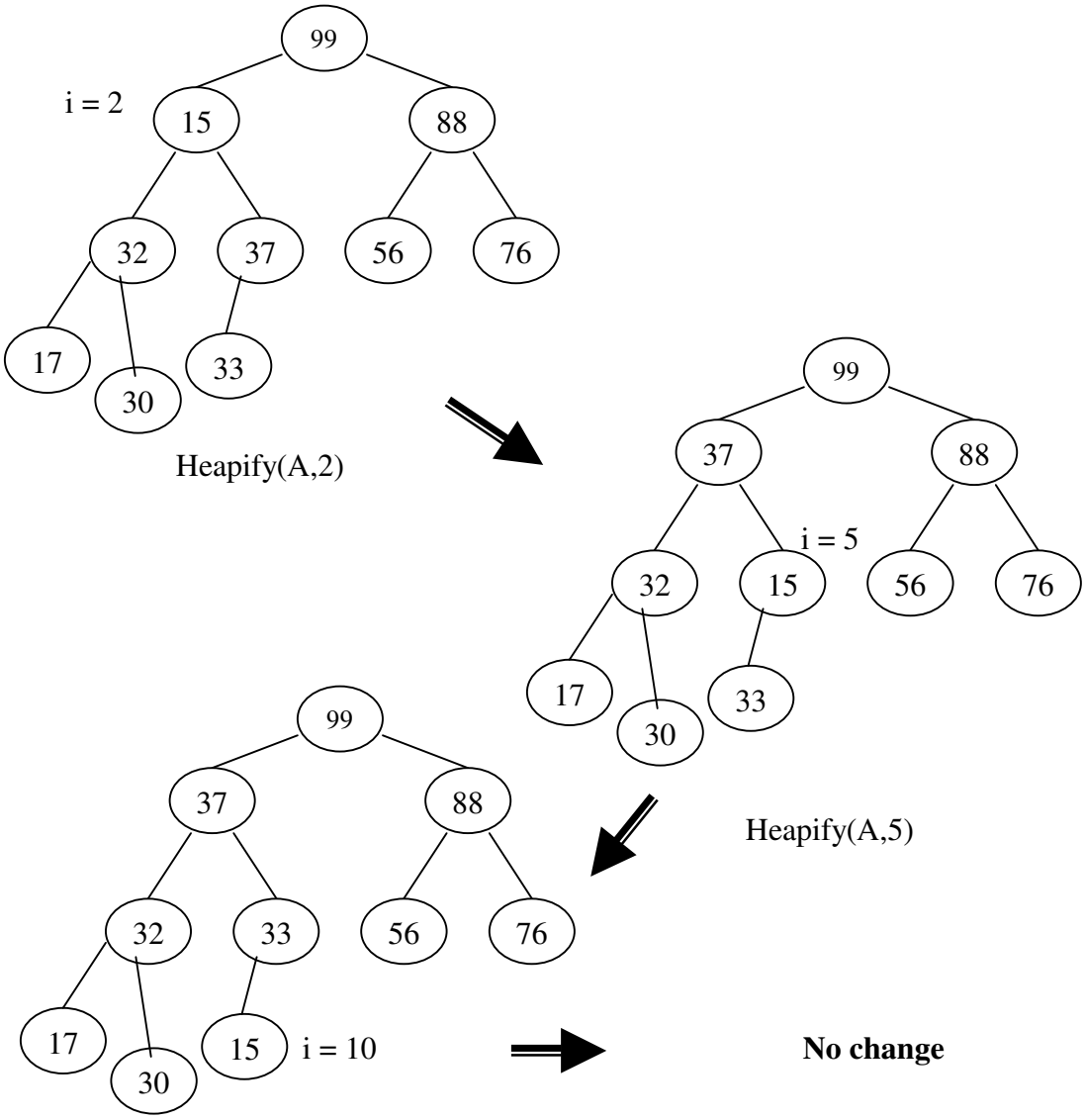
$$\sum_{i=0}^{\infty} kx^k = x/(1\text{-}x)^2.$$

Substituting ½ for x we get summation as 2.

So BuildHeap(A) time complexity is O(2n) i.e. linear.

Lesson from above analysis: do not over estimate, analyze carefully to get tighter bound.

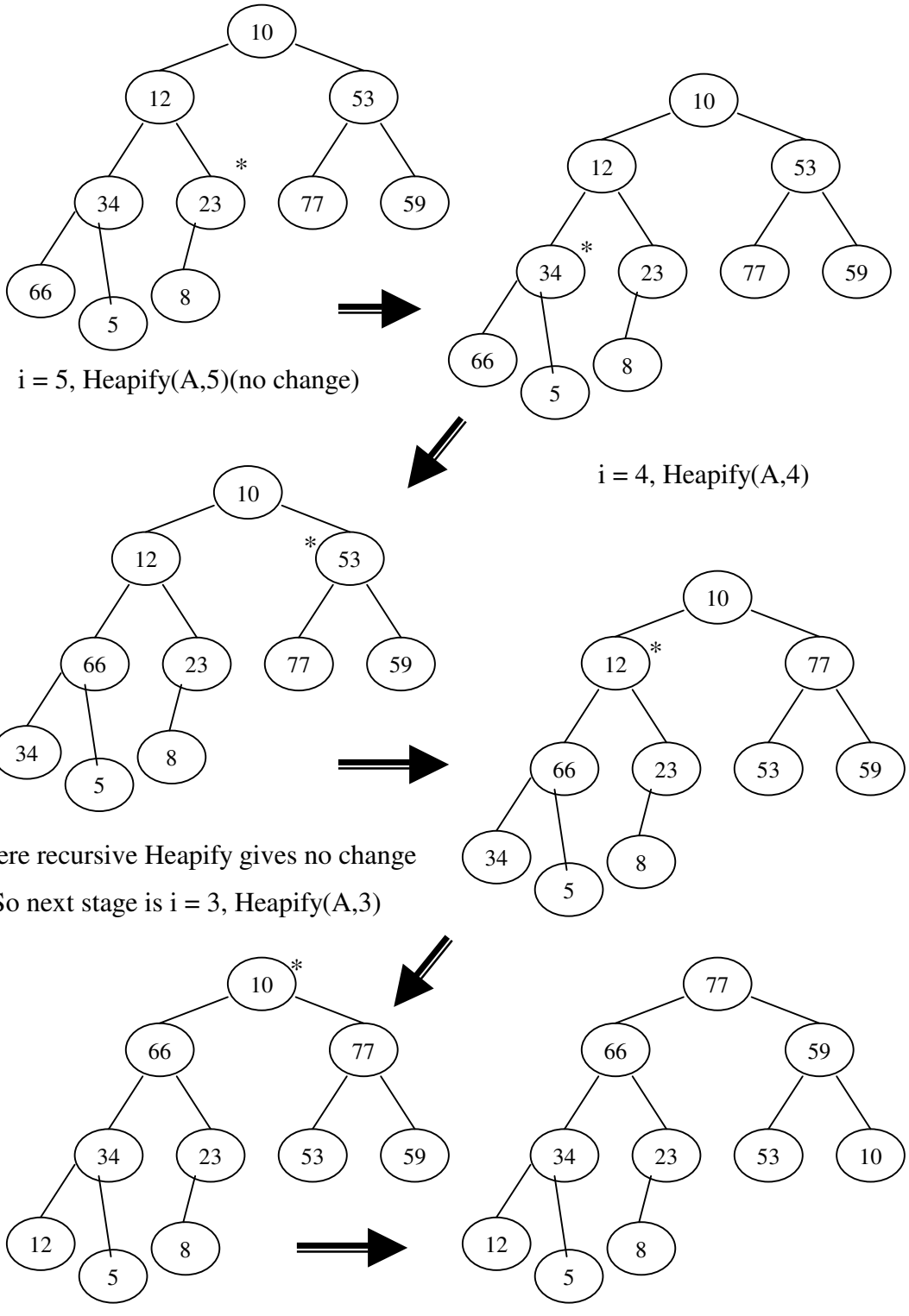
Running Example of Heapify for heapsize[A] = 10



In the above running example the action of Heapify(A,2) is shown. Where the recursive calls to Heapify(A,5) and then Heapify(A,2) are called. When Heapify(A,2) is called no changes occurs to the data structure.

Running Example of BuildHeap

A[] = {10,12,53, 34, 23, 77, 59, 66, 5, 8}



Algorithm:

```
HeapSort(A)
{
        BuildHeap(A);          //into max heap
        m = length[A];
        for(i = m ; i >= 2; i--)
        {
                swap(A[1],A[m]);
                m = m-1;
                Heapify(A,1);
        }
}
```

Analysis:

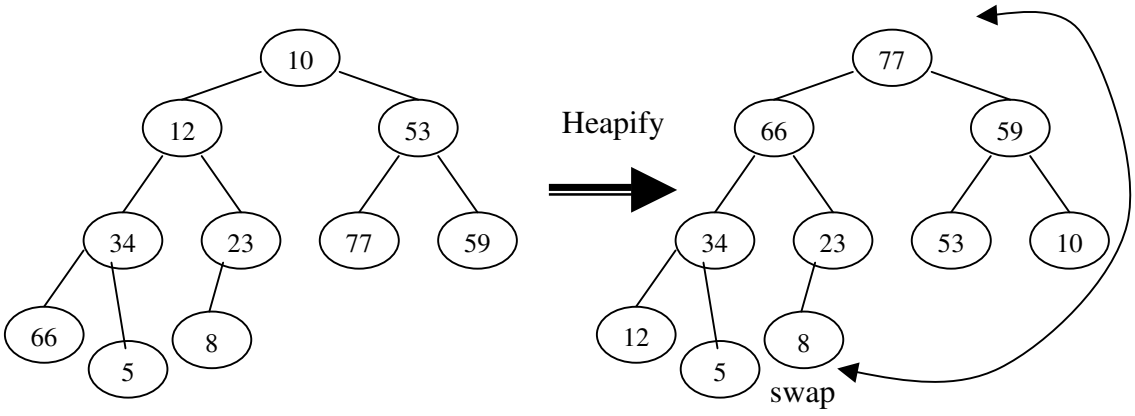
BuildHeap takes O(n) times.

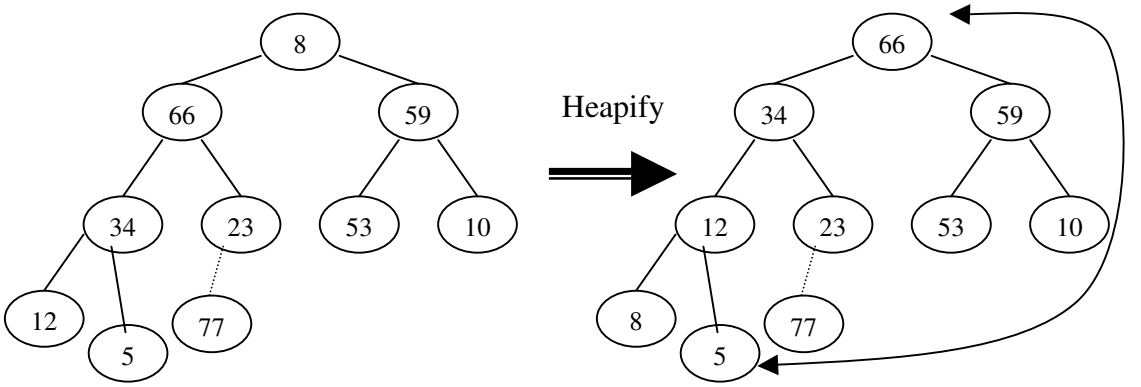
For loop executes for n times but each instruction inside loop executes for n-1 times where instruction inside loops takes O(logn) time.

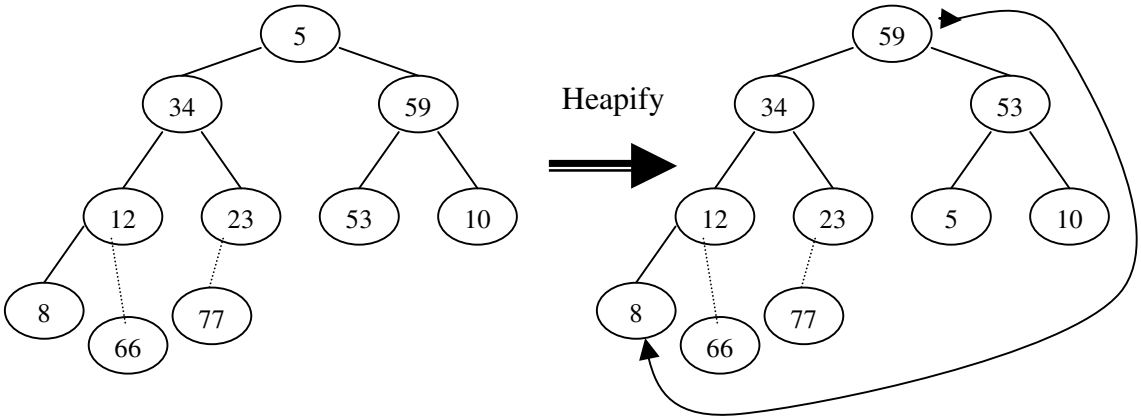
So total time T(n) = O(n) + (n-1) O(logn) = O(nlogn).

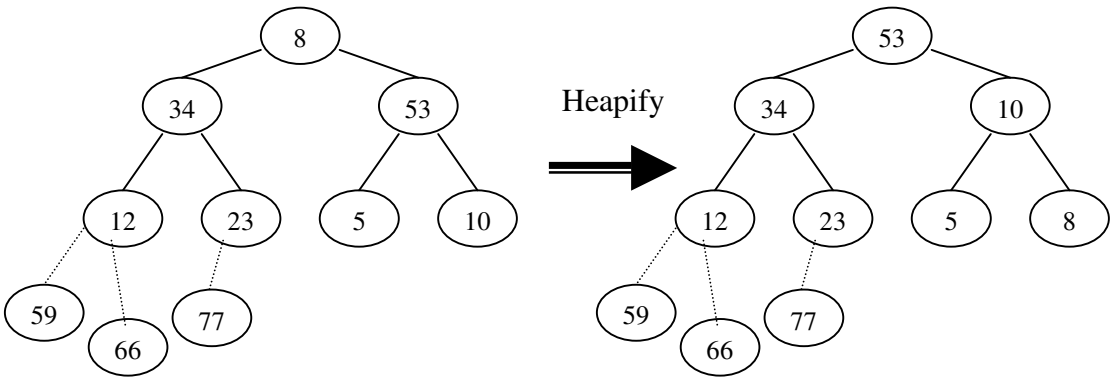
Running example of HeapSort:

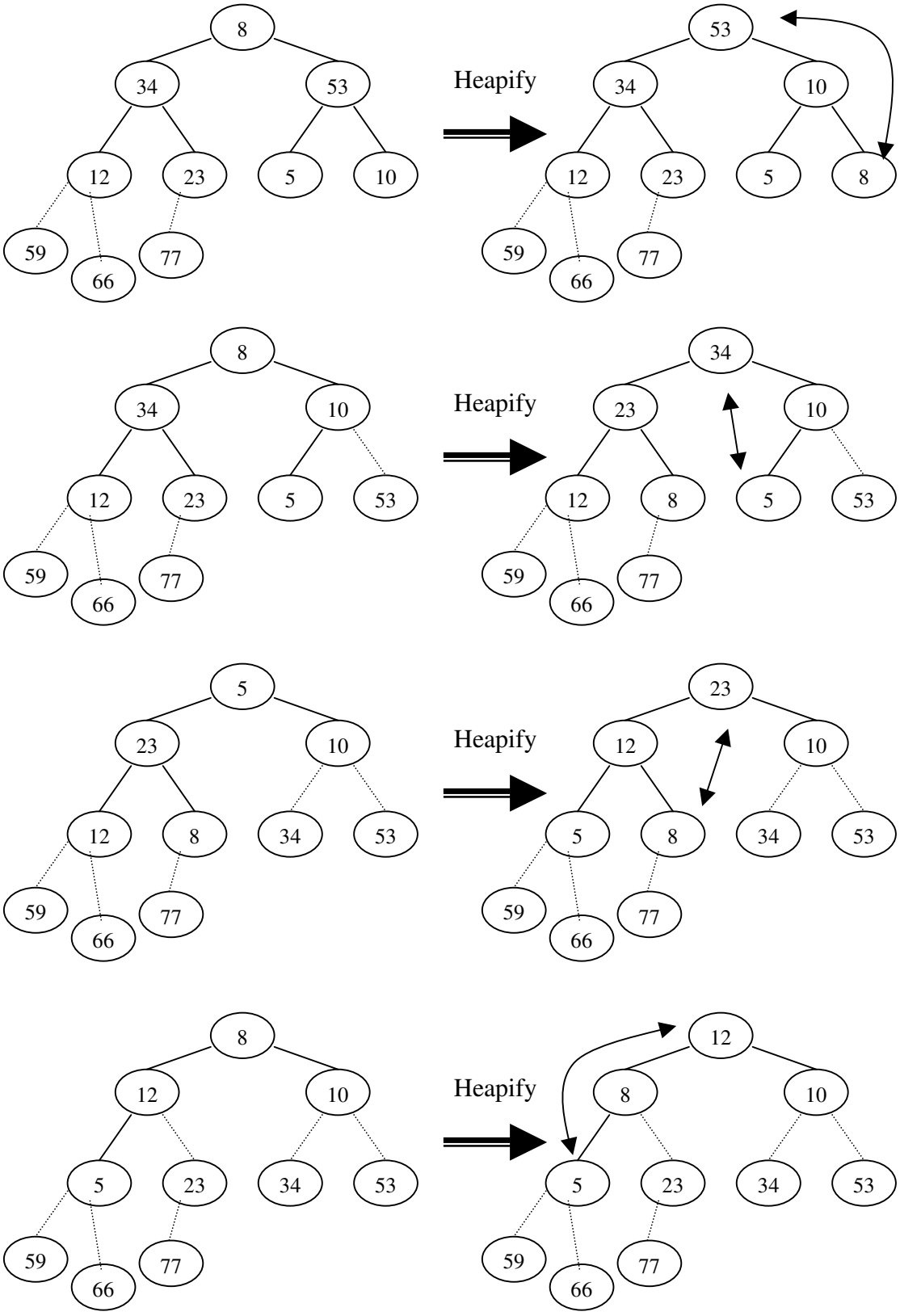
A[] = {10,12,53, 34, 23, 77, 59, 66, 5, 8}

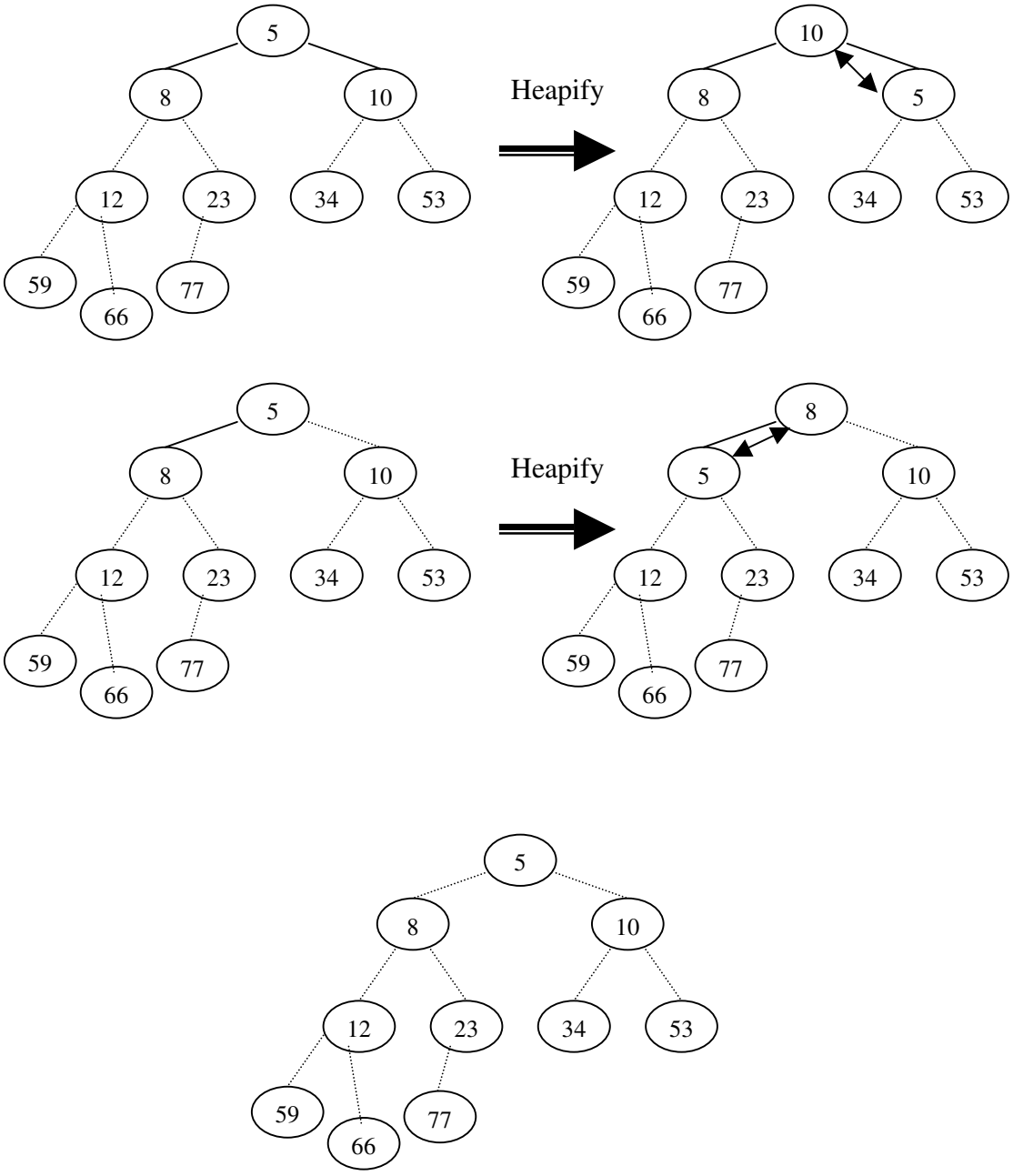



8
66
59
34
23
53
10
12
77
5
Heapify
66
34
59
12
23
53
10
8
77
5


5
34
59
12
23
53
10
8
77
66
Heapify
59
34
53
12
23
5
10
8
77
66


8
34
53
12
23
5
10
59
77
66
Heapify
53
34
10
12
23
5
8
59
77
66


8
34
53
12
23
5
10
59
66
77
Heapify
53
34
10
12
23
5
8
59
66
77
8
34
10
12
23
5
53
59
66
77
Heapify
34
23
10
12
8
5
53
59
66
77
5
23
10
12
8
34
53
59
66
77
Heapify
23
12
10
5
8
34
53
59
66
77
8
12
10
5
23
34
53
59
66
77
Heapify
12
8
10
5
23
34
53
59
66
77



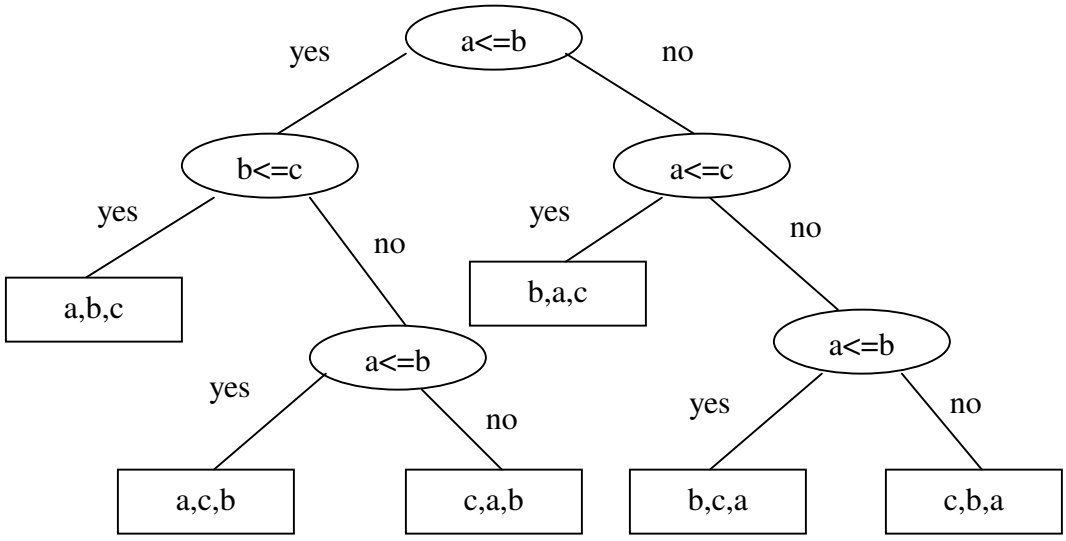
Sorted sequence

A[] = {5, 8, 10, 12, 23, 34, 53, 59, 66, 77}

Self study: priority queues

Lower Bound for Sorting

Comparison sorts algorithms compare pair of data for relation greater than or smaller than. They work in same manner for every kind of data. Consider all the possible comparison runs of sorting algorithm on input size n. When we unwind loops and recursive calls and focus on the point where there is comparison, we get binary decision tree. The tree describes all the possibilities for the algorithm execution on an input of n elements.



At the leaves, we have original input in the sorted order. By comparison we can generate all the possible permutations of the inputs are possible so there must be at least n! leaves in the tree. A complete tree of height h has 2^h leaves, so our decision tree must have h>= log(n!), but logn! is Ω(nlogn). Thus, the number of comparison in the worst case is Ω(nlogn). So we can say that runtime complexity of any comparison sort is Ω(nlogn).

Exercises

1. Understand what is stable sorting algorithm and identify the stable sorting algorithms we have studied.
2. Give the running example for Heapify, BuildHeap and Heapsort for the 14 elements input.
3. what is the running time of heap sort on array A of length n that is already sorted in increasing order? what about decreasing order?