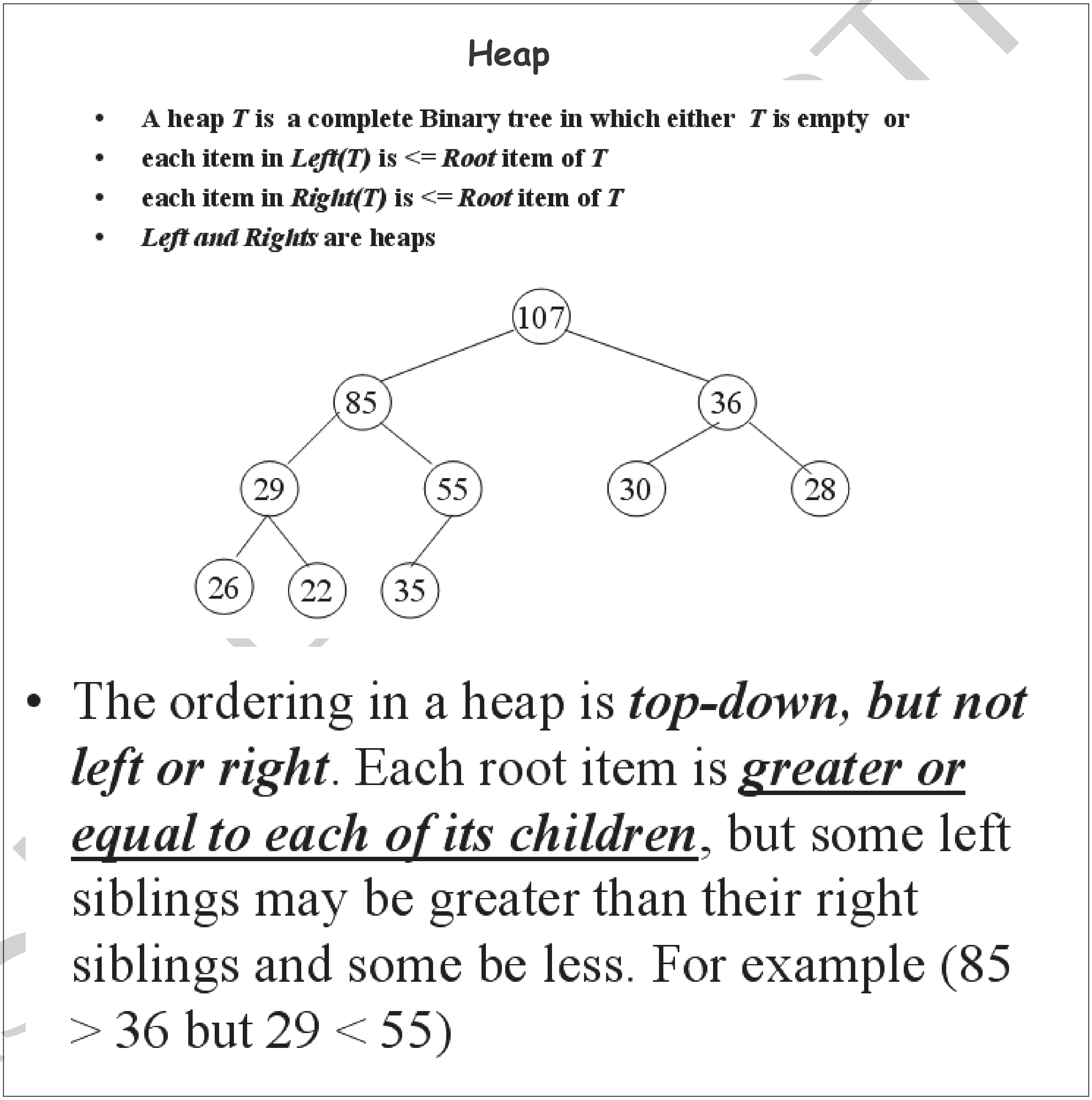
Chapter 10

Heap

10.1 Introduction

10.2 What is a heap?



Heap

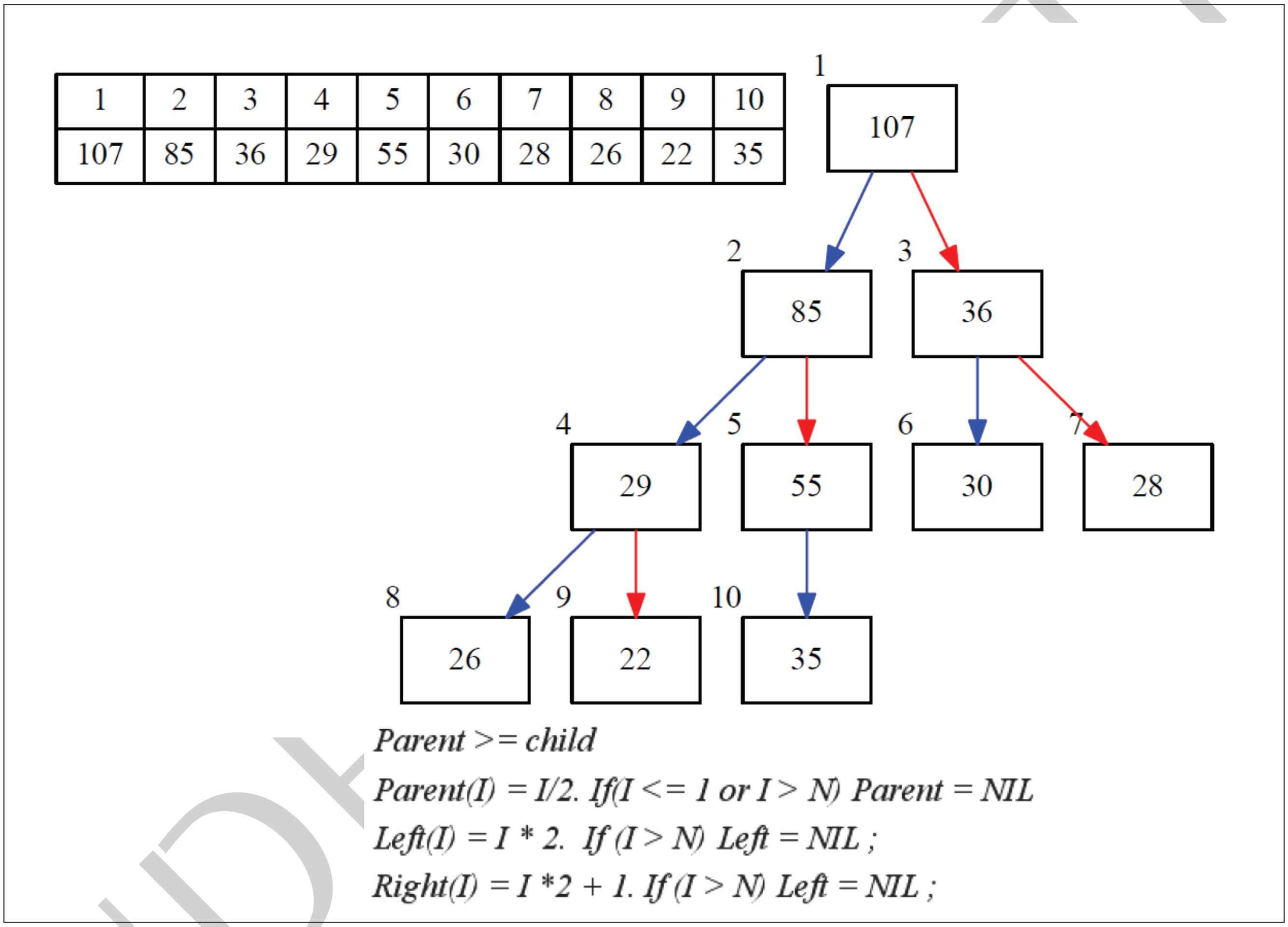
- **A heap *T* is a complete Binary tree in which either *T* is empty or**
- **each item in *Left(T)* is <= *Root* item of *T***
- **each item in *Right(T)* is <= *Root* item of *T***
- ***Left and Rights* are heaps**

107
85
36
29
55
30
28
26
22
35

- The ordering in a heap is ***top-down, but not left or right***. Each root item is ***<u>greater or equal to each of its children</u>***, but some left siblings may be greater than their right siblings and some be less. For example (85 > 36 but 29 < 55)

Figure 10.1: Max heap of **int**

10.3 Representation of heap as an array



1	2	3	4	5	6	7	8	9	10
107	85	36	29	55	30	28	26	22	35

Parent >= child

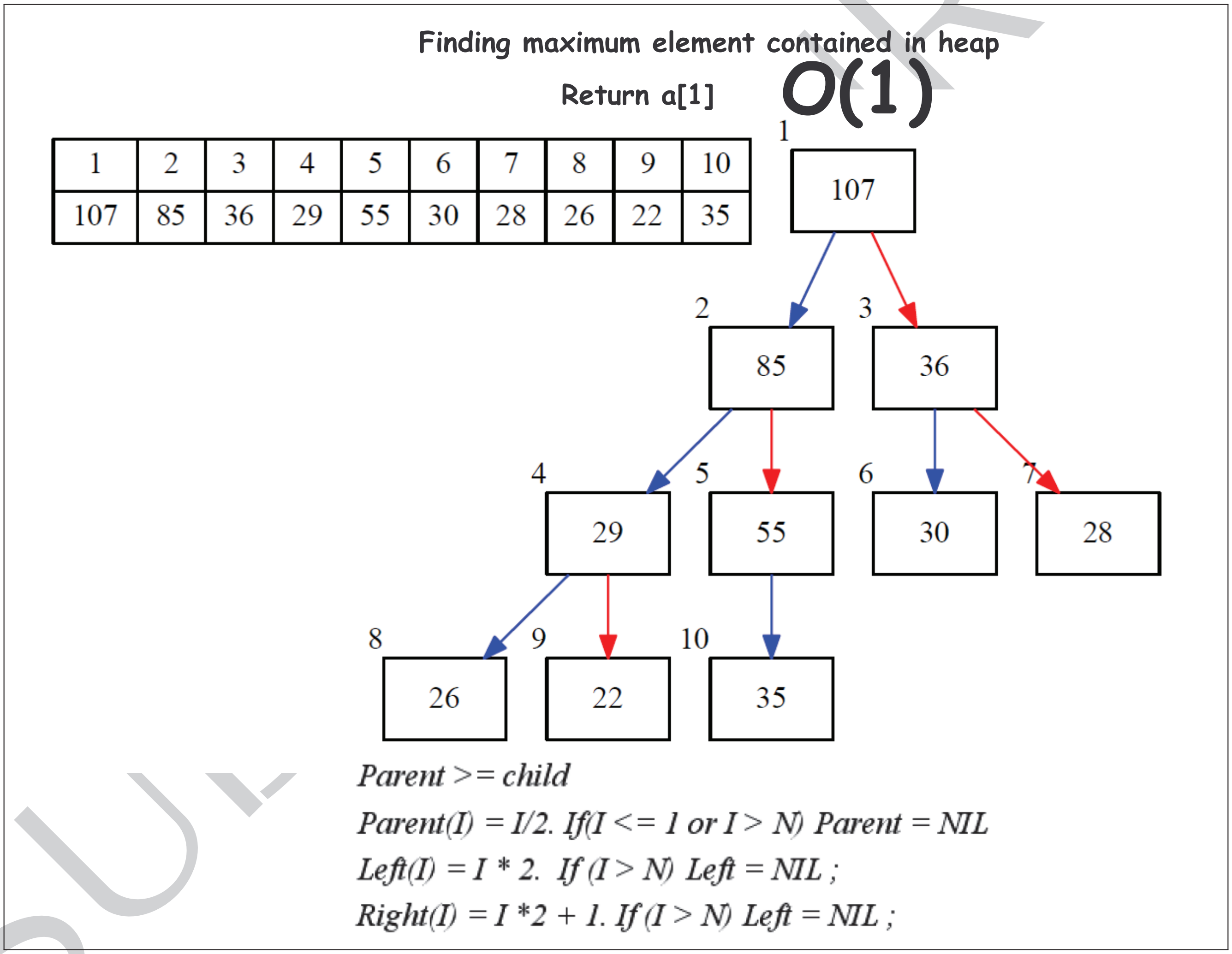
Parent(I) = I/2. If(I <= 1 or I > N) Parent = NIL

*Left(I) = I * 2. If (I > N) Left = NIL ;*

*Right(I) = I *2 + 1. If (I > N) Left = NIL ;*

Figure 10.2: Representation of max heap of **int**

10.4 Finding maximum element of a heap



Finding maximum element contained in heap

Return a[1] **O(1)**

1	2	3	4	5	6	7	8	9	10
107	85	36	29	55	30	28	26	22	35

Parent >= child

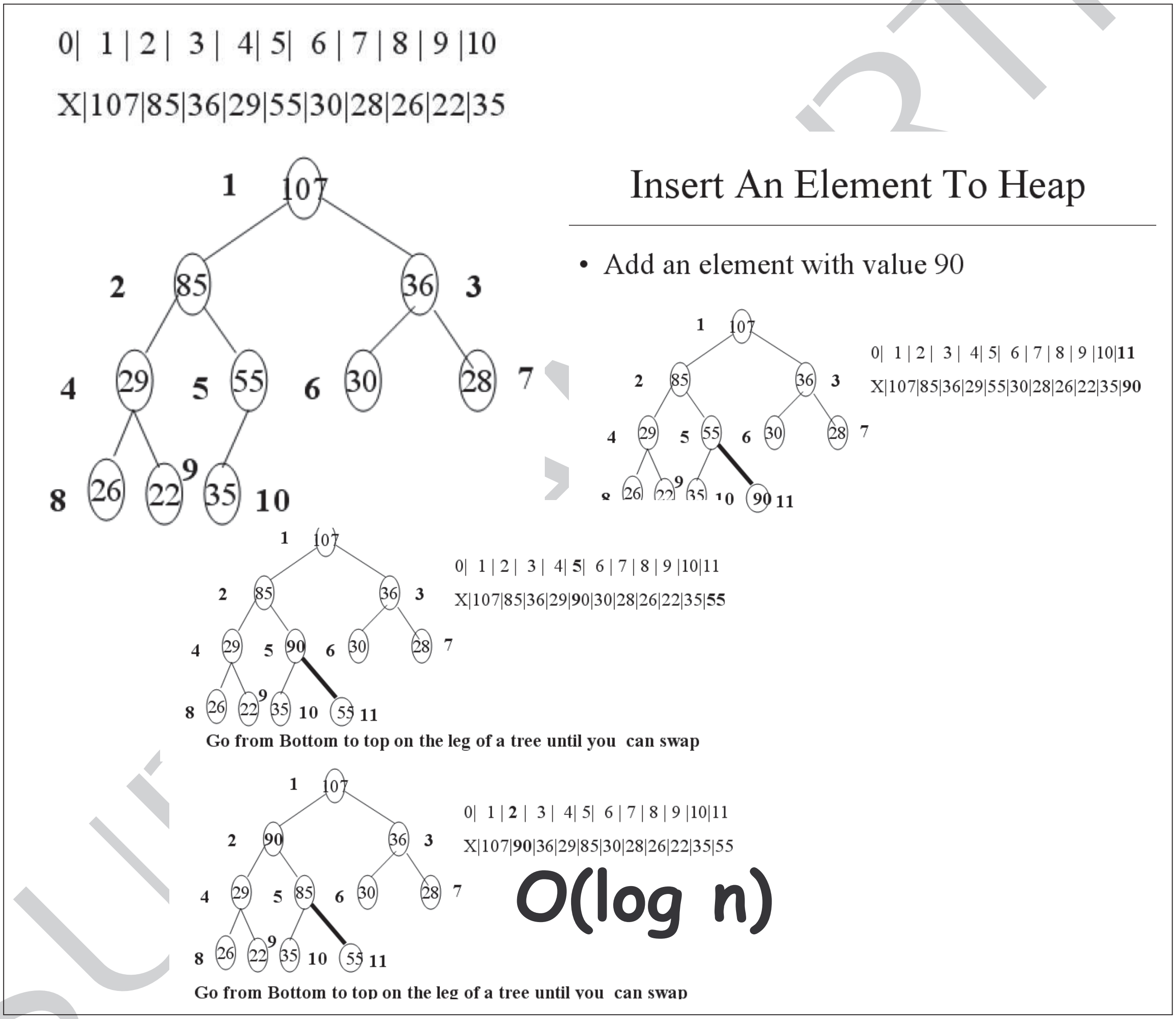
Parent(I) = I/2. If(I <= 1 or I > N) Parent = NIL

*Left(I) = I * 2. If (I > N) Left = NIL ;*

*Right(I) = I *2 + 1. If (I > N) Left = NIL ;*

Figure 10.3: Finding max element

10.5 Inserting an element to the heap


0| 1 | 2 | 3 | 4| 5| 6 | 7 | 8 | 9 |10

X|107|85|36|29|55|30|28|26|22|35

Insert An Element To Heap

- Add an element with value 90

0| 1 | 2 | 3 | 4| 5| 6 | 7 | 8 | 9 |10|**11**

X|107|85|36|29|55|30|28|26|22|35|**90**

0| 1 | 2 | 3 | 4| **5**| 6 | 7 | 8 | 9 |10|11

X|107|85|36|29|**90**|30|28|26|22|35|**55**

Go from Bottom to top on the leg of a tree until you can swap

0| 1 | **2** | 3 | 4| 5| 6 | 7 | 8 | 9 |10|11

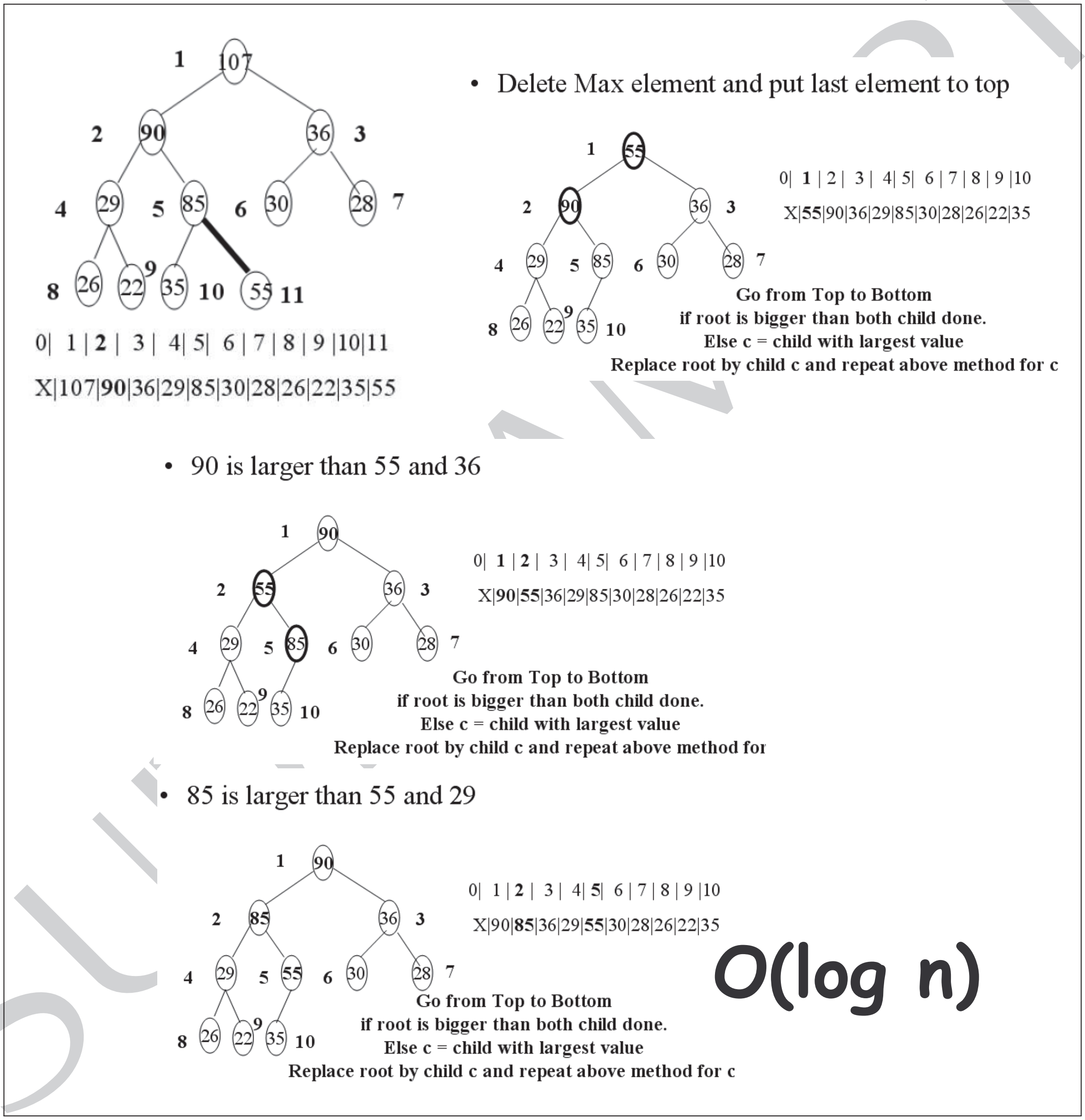
X|107|**90**|36|29|85|30|28|26|22|35|55

O(log n)

Go from Bottom to top on the leg of a tree until you can swap

Figure 10.4: Inserting an element to the heap

10.6 Deleting maximum element from the heap



0| 1 | **2** | 3 | 4| 5| 6 | 7 | 8 | 9 |10|11

X|107|**90**|36|29|85|30|28|26|22|35|55

- Delete Max element and put last element to top

0| **1** | 2 | 3 | 4| 5| 6 | 7 | 8 | 9 |10

X|**55**|90|36|29|85|30|28|26|22|35

Go from Top to Bottom
if root is bigger than both child done.
Else c = child with largest value
Replace root by child c and repeat above method for c

- 90 is larger than 55 and 36

0| **1** | **2** | 3 | 4| 5| 6 | 7 | 8 | 9 |10

X|**90**|**55**|36|29|85|30|28|26|22|35

Go from Top to Bottom
if root is bigger than both child done.
Else c = child with largest value
Replace root by child c and repeat above method for

- 85 is larger than 55 and 29

0| 1 | **2** | 3 | 4| **5**| 6 | 7 | 8 | 9 |10

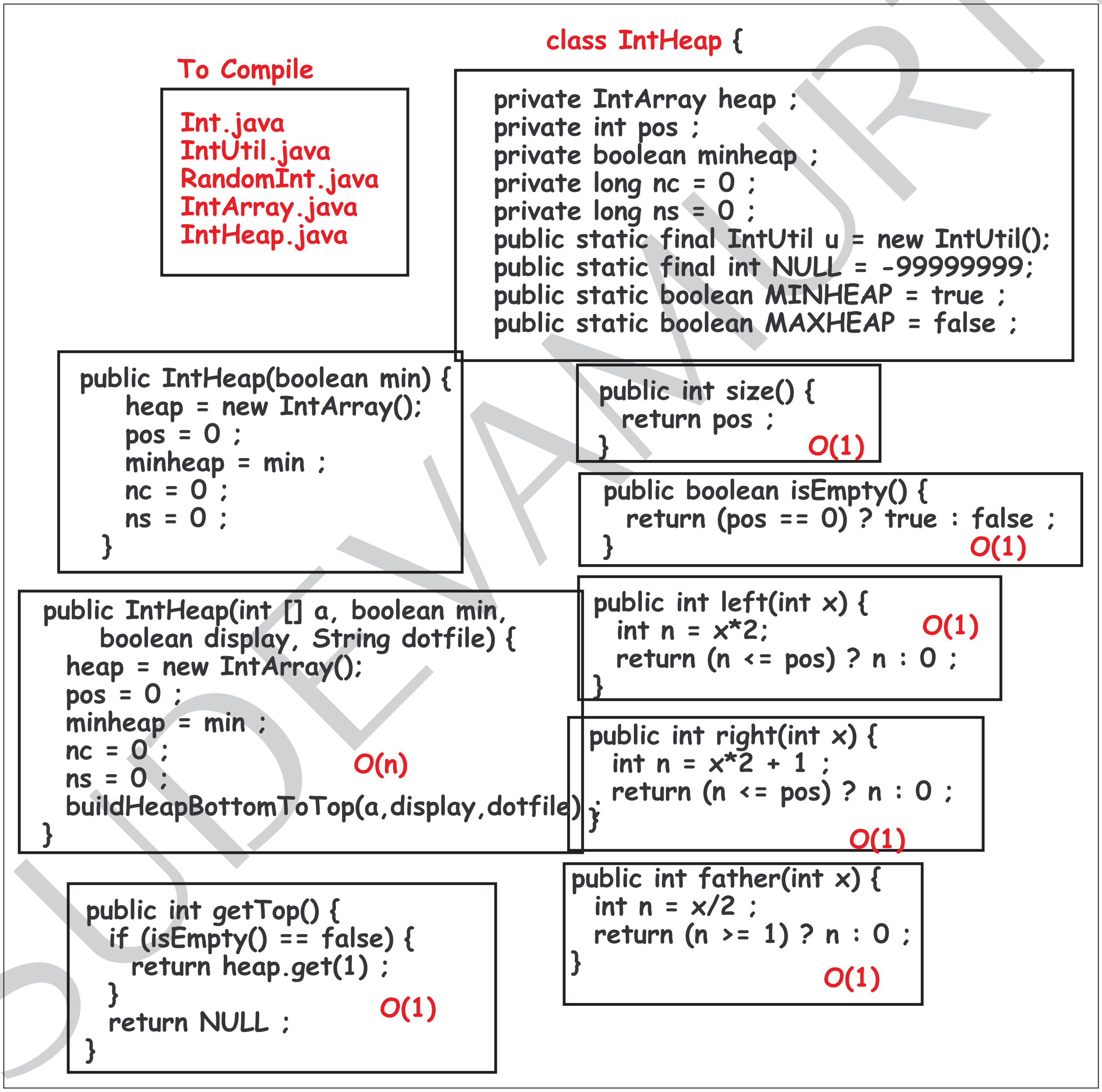
X|90|**85**|36|29|**55**|30|28|26|22|35

Go from Top to Bottom
if root is bigger than both child done.
Else c = child with largest value
Replace root by child c and repeat above method for c

O(log n)

Figure 10.5: Deletion

10.7 Writing class IntHeap



```
To Compile

Int.java
IntUtil.java
RandomInt.java
IntArray.java
IntHeap.java
```

```
class IntHeap {
    private IntArray heap ;
    private int pos ;
    private boolean minheap ;
    private long nc = 0 ;
    private long ns = 0 ;
    public static final IntUtil u = new IntUtil();
    public static final int NULL = -99999999;
    public static boolean MINHEAP = true ;
    public static boolean MAXHEAP = false ;
```

```
public IntHeap(boolean min) {
    heap = new IntArray();
    pos = 0 ;
    minheap = min ;
    nc = 0 ;
    ns = 0 ;
}
```

```
public int size() {
  return pos ;
}                    O(1)
```

```
public boolean isEmpty() {
  return (pos == 0) ? true : false ;
}                              O(1)
```

```
public IntHeap(int [] a, boolean min,
     boolean display, String dotfile) {
  heap = new IntArray();
  pos = 0 ;
  minheap = min ;
  nc = 0 ;
  ns = 0 ;                    O(n)
  buildHeapBottomToTop(a,display,dotfile) ;
}
```

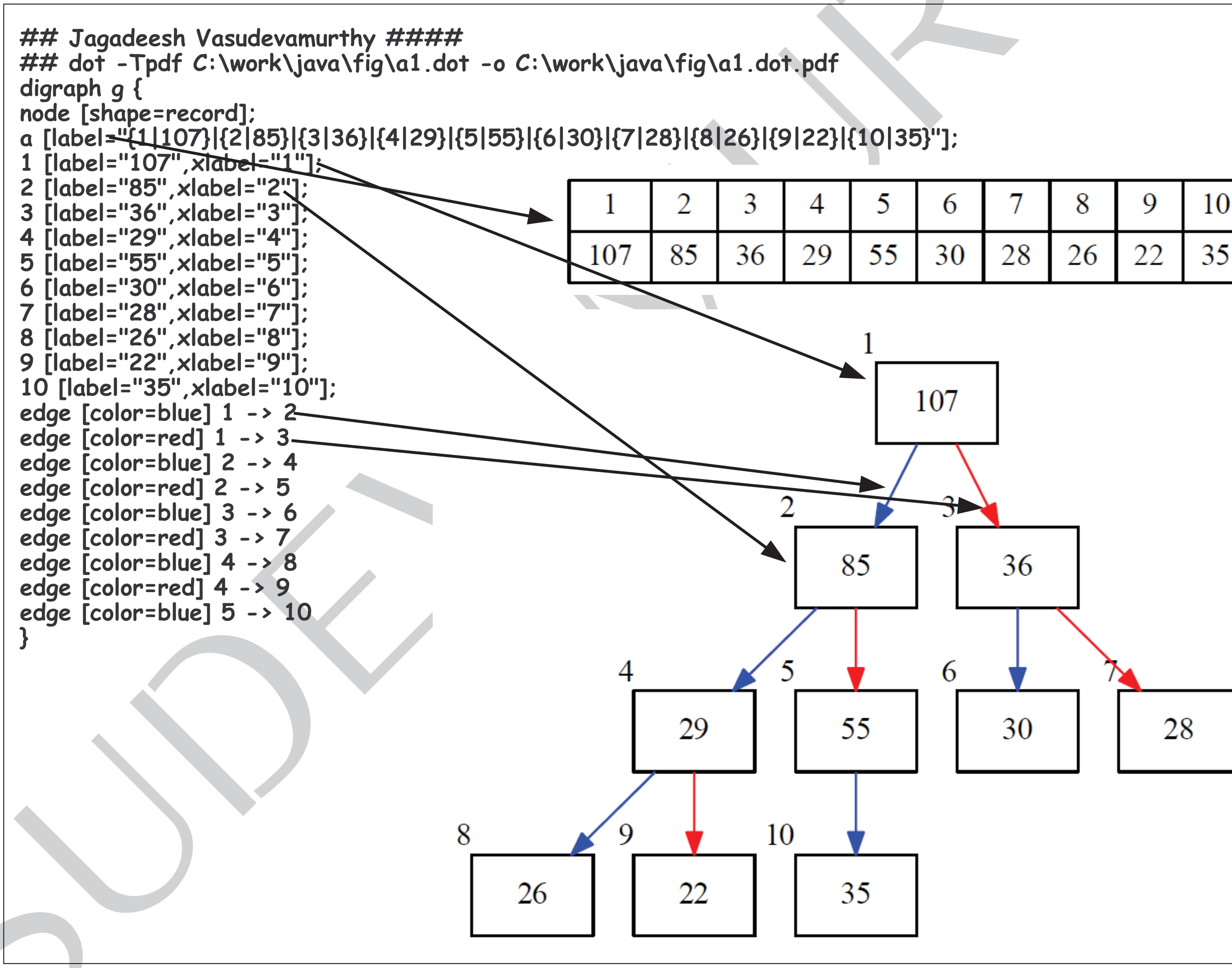
```
public int left(int x) {
  int n = x*2;                O(1)
  return (n <= pos) ? n : 0 ;
}
```

```
public int right(int x) {
  int n = x*2 + 1 ;
  return (n <= pos) ? n : 0 ;
}
                      O(1)
```

```
public int father(int x) {
  int n = x/2 ;
  return (n >= 1) ? n : 0 ;
}
                      O(1)
```

```
public int getTop() {
  if (isEmpty() == false) {
    return heap.get(1) ;
  }                        O(1)
  return NULL ;
}
```

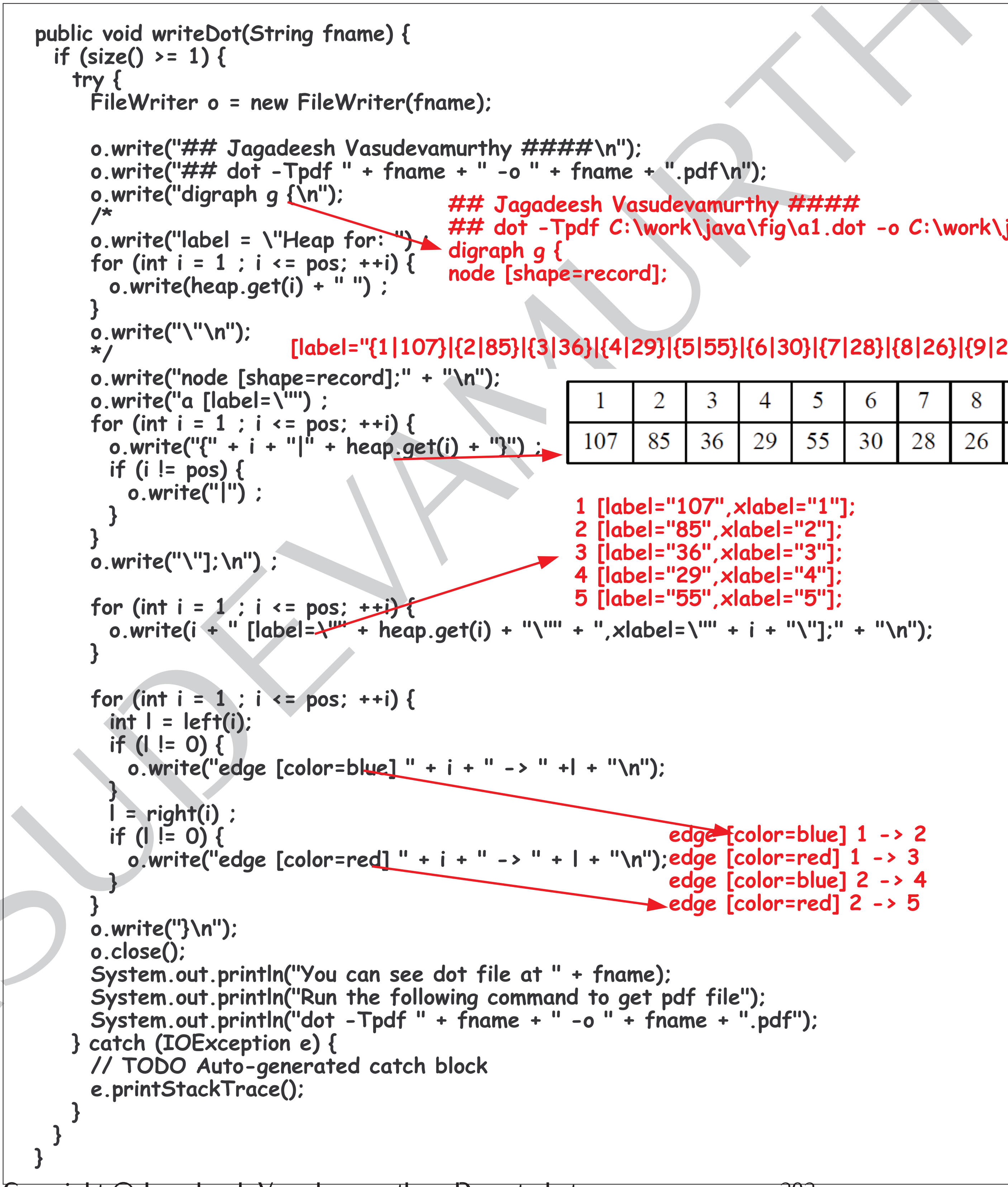
Figure 10.6: class IntHeap



```
## Jagadeesh Vasudevamurthy ####
## dot -Tpdf C:\work\java\fig\a1.dot -o C:\work\java\fig\a1.dot.pdf
digraph g {
node [shape=record];
a [label="{1|107}|{2|85}|{3|36}|{4|29}|{5|55}|{6|30}|{7|28}|{8|26}|{9|22}|{10|35}"];
1 [label="107",xlabel="1"];
2 [label="85",xlabel="2"];
3 [label="36",xlabel="3"];
4 [label="29",xlabel="4"];
5 [label="55",xlabel="5"];
6 [label="30",xlabel="6"];
7 [label="28",xlabel="7"];
8 [label="26",xlabel="8"];
9 [label="22",xlabel="9"];
10 [label="35",xlabel="10"];
edge [color=blue] 1 -> 2
edge [color=red] 1 -> 3
edge [color=blue] 2 -> 4
edge [color=red] 2 -> 5
edge [color=blue] 3 -> 6
edge [color=red] 3 -> 7
edge [color=blue] 4 -> 8
edge [color=red] 4 -> 9
edge [color=blue] 5 -> 10
}
```

1	2	3	4	5	6	7	8	9	10
107	85	36	29	55	30	28	26	22	35

Figure 10.7: Printing heap as a **dot** file



```
public void writeDot(String fname) {
  if (size() >= 1) {
    try {
      FileWriter o = new FileWriter(fname);

      o.write("## Jagadeesh Vasudevamurthy ####\n");
      o.write("## dot -Tpdf " + fname + " -o " + fname + ".pdf\n");
      o.write("digraph g {\n");
      /*
      o.write("label = \"Heap for: ") ;
      for (int i = 1 ; i <= pos; ++i) {
        o.write(heap.get(i) + " ") ;
      }
      o.write("\"\n");
      */
      o.write("node [shape=record];" + "\n");
      o.write("a [label=\"") ;
      for (int i = 1 ; i <= pos; ++i) {
        o.write("{" + i + "|" + heap.get(i) + "}") ;
        if (i != pos) {
          o.write("|") ;
        }
      }
      o.write("\"];\n") ;

      for (int i = 1 ; i <= pos; ++i) {
        o.write(i + " [label=\"" + heap.get(i) + "\"" + ",xlabel=\"" + i + "\"];" + "\n");
      }

      for (int i = 1 ; i <= pos; ++i) {
        int l = left(i);
        if (l != 0) {
          o.write("edge [color=blue] " + i + " -> " +l + "\n");
        }
        l = right(i) ;
        if (l != 0) {
          o.write("edge [color=red] " + i + " -> " + l + "\n");
        }
      }
      o.write("}\n");
      o.close();
      System.out.println("You can see dot file at " + fname);
      System.out.println("Run the following command to get pdf file");
      System.out.println("dot -Tpdf " + fname + " -o " + fname + ".pdf");
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
}
```

```
## Jagadeesh Vasudevamurthy ####
## dot -Tpdf C:\work\java\fig\a1.dot -o C:\work\j
digraph g {
node [shape=record];
```

```
[label="{1|107}|{2|85}|{3|36}|{4|29}|{5|55}|{6|30}|{7|28}|{8|26}|{9|2
```

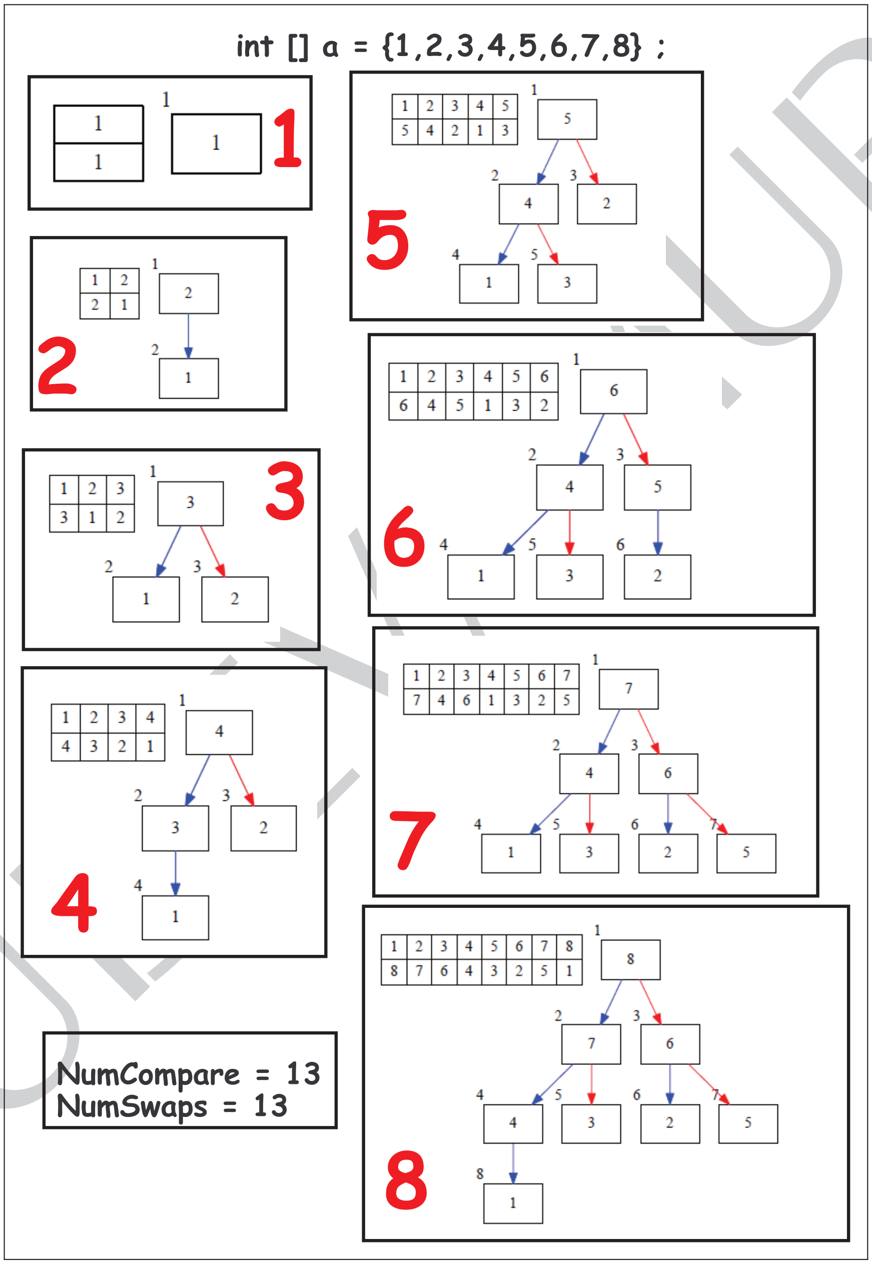
1	2	3	4	5	6	7	8
107	85	36	29	55	30	28	26

```
1 [label="107",xlabel="1"];
2 [label="85",xlabel="2"];
3 [label="36",xlabel="3"];
4 [label="29",xlabel="4"];
5 [label="55",xlabel="5"];
```

```
edge [color=blue] 1 -> 2
edge [color=red] 1 -> 3
edge [color=blue] 2 -> 4
edge [color=red] 2 -> 5
```

Figure 10.8: Code for printing heap as a **dot** file

10.8 Building a heap of n elements from top to bottom

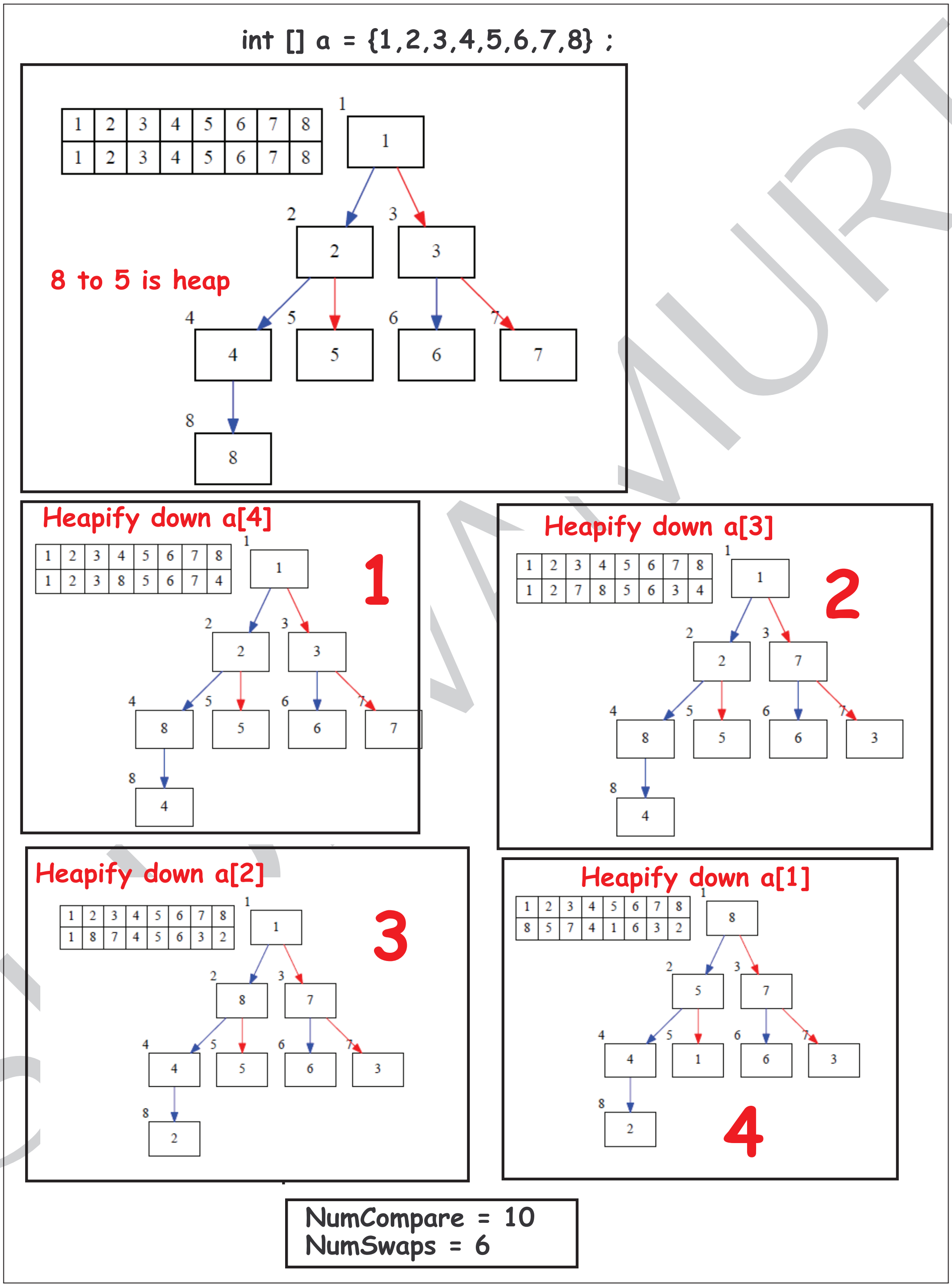


```
int [] a = {1,2,3,4,5,6,7,8} ;
```

```
NumCompare = 13
NumSwaps = 13
```

Figure 10.9: Building a heap of *n* elements **from top to bottom**

10.9 Building a heap of *n* elements from bottom to top



1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

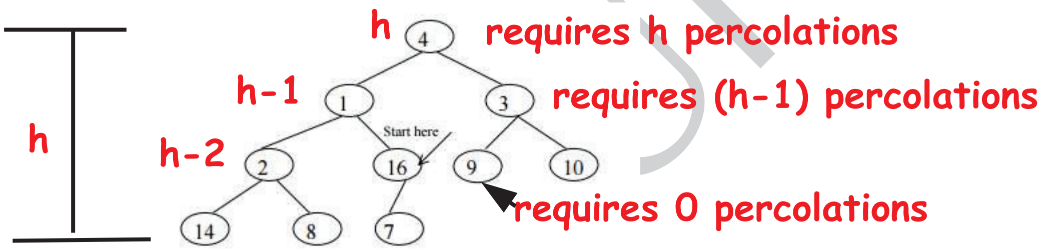
1	2	3	4	5	6	7	8
1	2	3	8	5	6	7	4

1	2	3	4	5	6	7	8
1	2	7	8	5	6	3	4

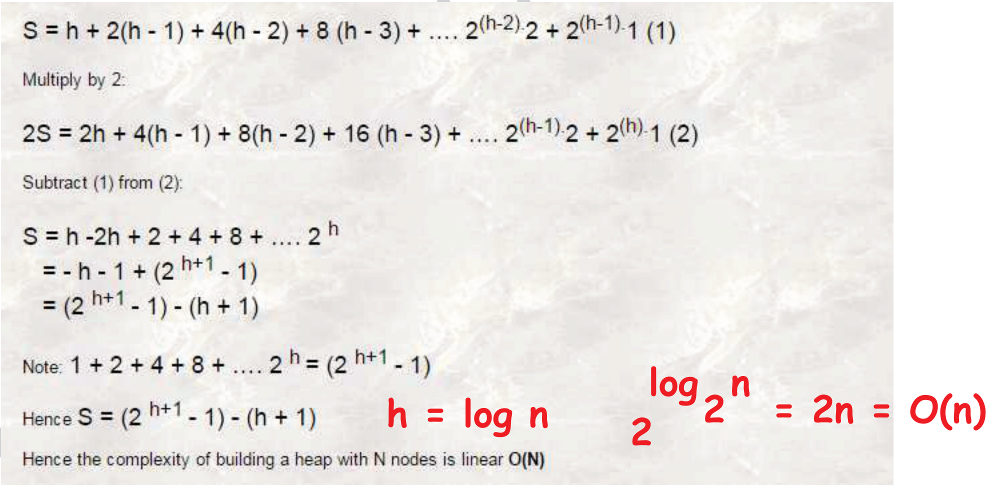
1	2	3	4	5	6	7	8
1	8	7	4	5	6	3	2

1	2	3	4	5	6	7	8
8	5	7	4	1	6	3	2

Figure 10.10: Building a heap of *n* elements **from bottom to top**



S= h + 2(h-1) + 4(h-2) + --- +



S = h + 2(h - 1) + 4(h - 2) + 8 (h - 3) + ... $2^{(h-2)} \cdot 2 + 2^{(h-1)} \cdot 1$ (1)

Multiply by 2:

2S = 2h + 4(h - 1) + 8(h - 2) + 16 (h - 3) + ... $2^{(h-1)} \cdot 2 + 2^{(h)} \cdot 1$ (2)

Subtract (1) from (2):

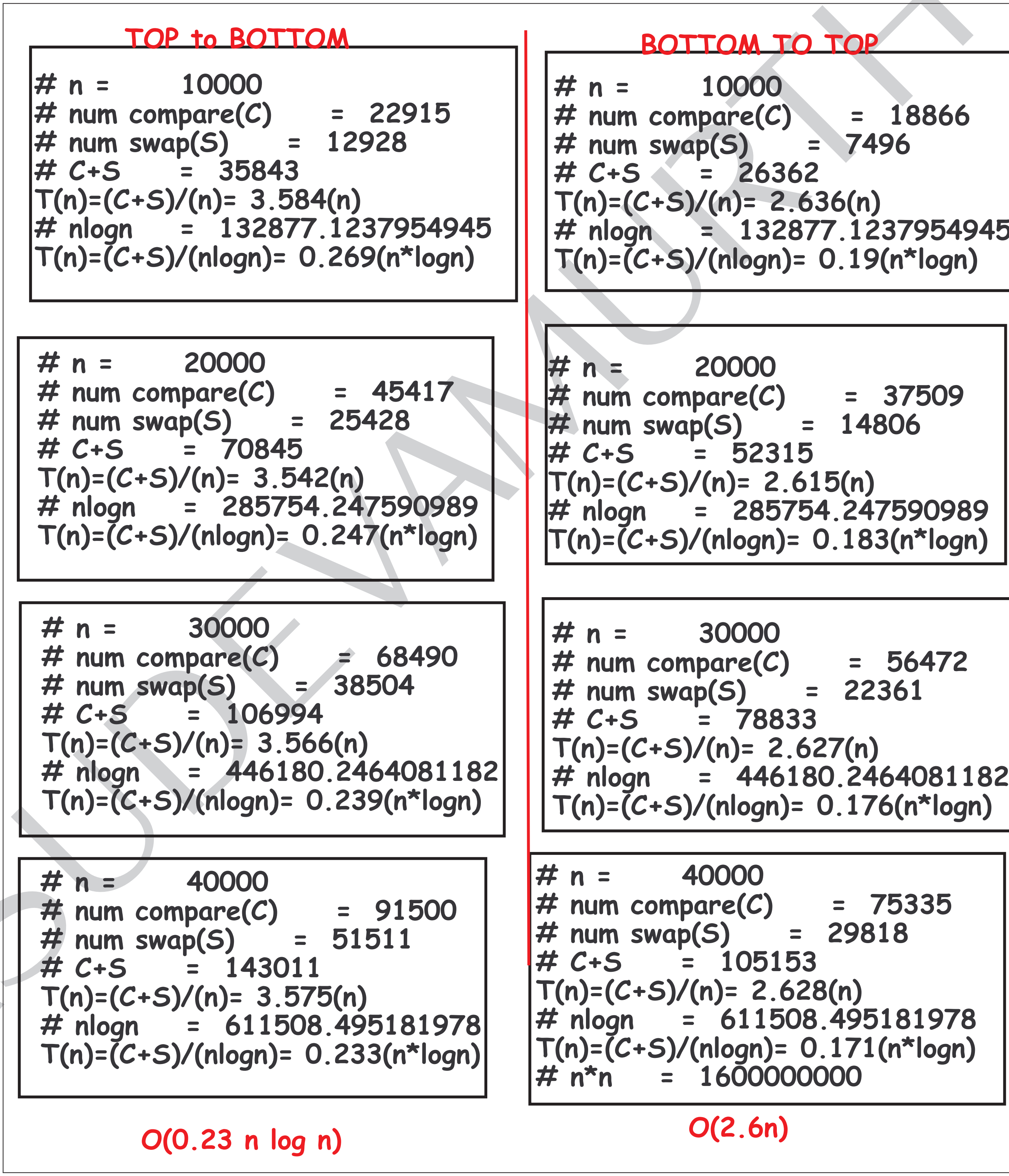
S = h -2h + 2 + 4 + 8 + ... 2^h
= - h - 1 + (2^{h+1} - 1)
= (2^{h+1} - 1) - (h + 1)

Note: 1 + 2 + 4 + 8 + ... 2^h = (2^{h+1} - 1)

Hence S = (2^{h+1} - 1) - (h + 1)   h = log n $2^{\log_2 n}$ = 2n = O(n)

Hence the complexity of building a heap with N nodes is linear **O(N)**

Figure 10.11: Analysis of building a heap of *n* elements **from bottom to top**



TOP to BOTTOM

```
# n =       10000
# num compare(C)      =  22915
# num swap(S)      =  12928
# C+S      =  35843
T(n)=(C+S)/(n)= 3.584(n)
# nlogn     =  132877.1237954945
T(n)=(C+S)/(nlogn)= 0.269(n*logn)
```

```
# n =       20000
# num compare(C)      =  45417
# num swap(S)      =  25428
# C+S      =  70845
T(n)=(C+S)/(n)= 3.542(n)
# nlogn     =  285754.247590989
T(n)=(C+S)/(nlogn)= 0.247(n*logn)
```

```
# n =       30000
# num compare(C)      =  68490
# num swap(S)      =  38504
# C+S      =  106994
T(n)=(C+S)/(n)= 3.566(n)
# nlogn     =  446180.2464081182
T(n)=(C+S)/(nlogn)= 0.239(n*logn)
```

```
# n =       40000
# num compare(C)      =  91500
# num swap(S)      =  51511
# C+S      =  143011
T(n)=(C+S)/(n)= 3.575(n)
# nlogn     =  611508.495181978
T(n)=(C+S)/(nlogn)= 0.233(n*logn)
```

O(0.23 n log n)

BOTTOM TO TOP

```
# n =       10000
# num compare(C)      =  18866
# num swap(S)      =  7496
# C+S      =  26362
T(n)=(C+S)/(n)= 2.636(n)
# nlogn     =  132877.1237954945
T(n)=(C+S)/(nlogn)= 0.19(n*logn)
```

```
# n =       20000
# num compare(C)      =  37509
# num swap(S)      =  14806
# C+S      =  52315
T(n)=(C+S)/(n)= 2.615(n)
# nlogn     =  285754.247590989
T(n)=(C+S)/(nlogn)= 0.183(n*logn)
```

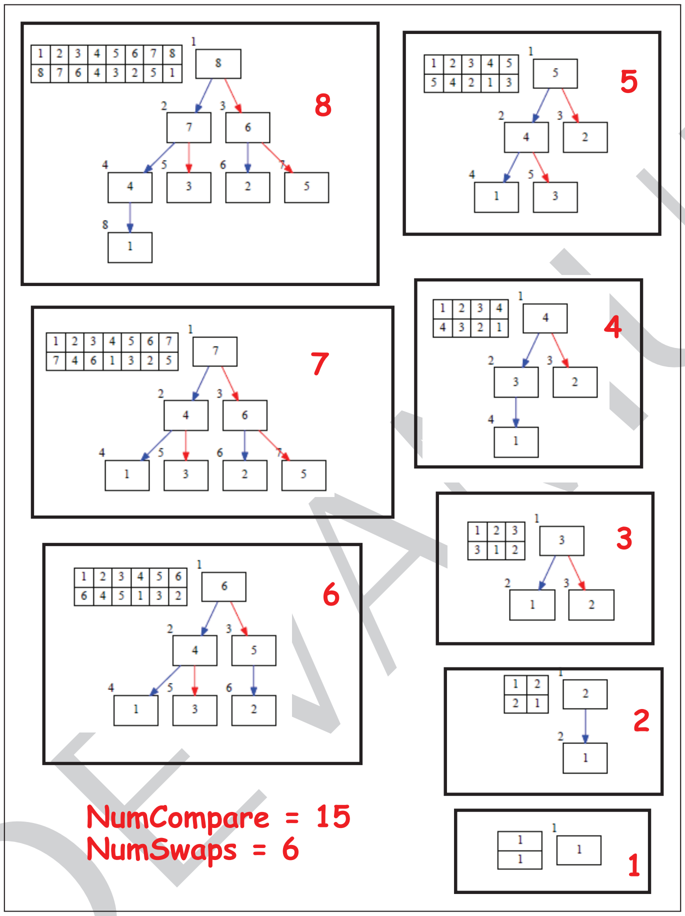
```
# n =       30000
# num compare(C)      =  56472
# num swap(S)      =  22361
# C+S      =  78833
T(n)=(C+S)/(n)= 2.627(n)
# nlogn     =  446180.2464081182
T(n)=(C+S)/(nlogn)= 0.176(n*logn)
```

```
# n =       40000
# num compare(C)      =  75335
# num swap(S)      =  29818
# C+S      =  105153
T(n)=(C+S)/(n)= 2.628(n)
# nlogn     =  611508.495181978
T(n)=(C+S)/(nlogn)= 0.171(n*logn)
# n*n     =  1600000000
```

O(2.6n)

Figure 10.12: Run time for building a heap

10.10 Heap sort algorithm



NumCompare = 15
NumSwaps = 6

Figure 10.13: Heap sort

10.11 Testing IntHeap

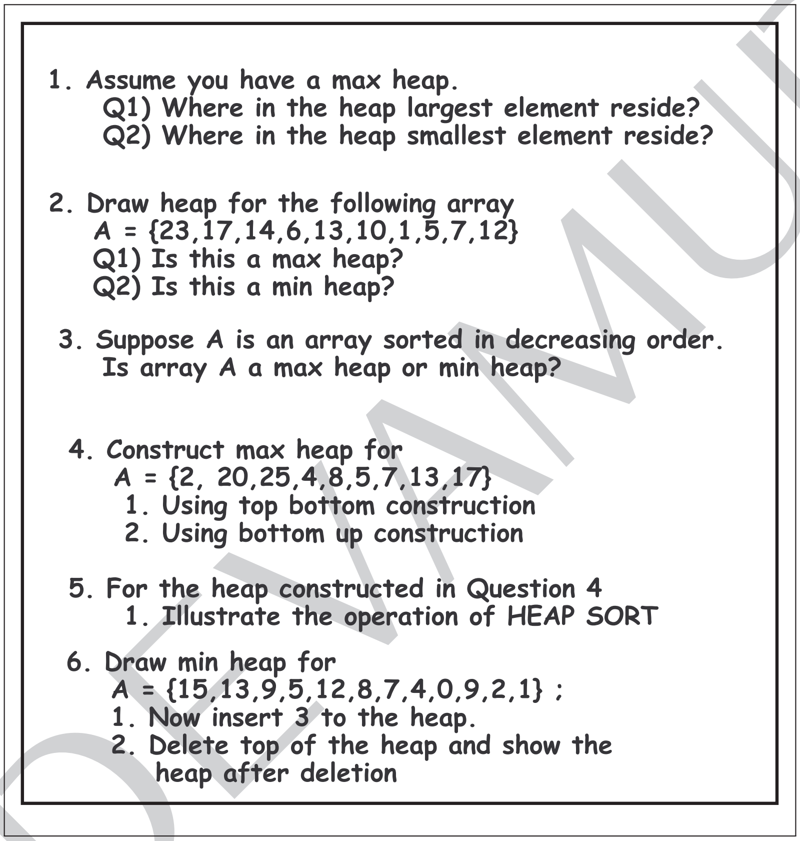
```
                Int.java
                IntUtil.java
                RandomInt.java                class IntHeapTest {
                IntHeap.java
                IntHeapTest.java

private static IntHeap buildHeapTopToBottom(int [] a, boolean min,
       boolean display,String dotfile) {
    IntHeap h = new IntHeap(min) ;
    h.buildHeapTopToBottom(a,display,dotfile);
    return h ;
  }
 private static IntHeap buildHeapBottomToTop(int [] a, boolean min,
    boolean display,String dotfile) {
    IntHeap h = new IntHeap(a,min,display,dotfile) ;
    return h ;
  }
 private static void test1() {
     int [] a = {1,2,3,4,5,6,7,8} ;
     IntHeap h = buildHeapTopToBottom(a,IntHeap.MAXHEAP,true,"C:\\work\\java\\fig\\a")
     System.out.println("Add 90 to heap") ;
     long ns = 0 ;
     long nc = 0 ;
     h.insert(90);
     ns = ns + h.numSwap();
     nc = nc + h.numCompare();
     System.out.println("To insert 90: NumCompare = " + nc + " NumSwaps " + ns) ;
     h.writeDot("C:\\work\\java\\fig\\a90.dot") ;
     System.out.println("delete top " + h.getTop()) ;
     ns = 0 ;
     nc = 0 ;
     h.deleteTop();
     ns = ns + h.numSwap();
     nc = nc + h.numCompare();
     System.out.println("To make heap right after deleting top:
          NumCompare = " + nc + " NumSwaps " + ns) ;
     h.writeDot("C:\\work\\java\\fig\\a91.dot") ;
  }
 private static void test4(boolean min) {
    for (int n = 10000; n < 50000; n = n + 10000) {
      int [] a = u.generateRandomNumber(n,false,1,2*n) ; //Generate positive and negative
      IntHeap h1 = buildHeapTopToBottom(a,min,true,null) ;
      IntHeap h2 = buildHeapBottomToTop(a,min,true,null) ;
    }
  }
```

Figure 10.14: Testing the heap class

10.12 Problem set

Problem 10.12.1. Answer all the questions asked in figure 10.15 by hand and attached the scanned paper. Must not attach word document.



1. Assume you have a max heap.
Q1) Where in the heap largest element reside?
Q2) Where in the heap smallest element reside?

2. Draw heap for the following array
A = {23,17,14,6,13,10,1,5,7,12}
Q1) Is this a max heap?
Q2) Is this a min heap?

3. Suppose A is an array sorted in decreasing order.
Is array A a max heap or min heap?

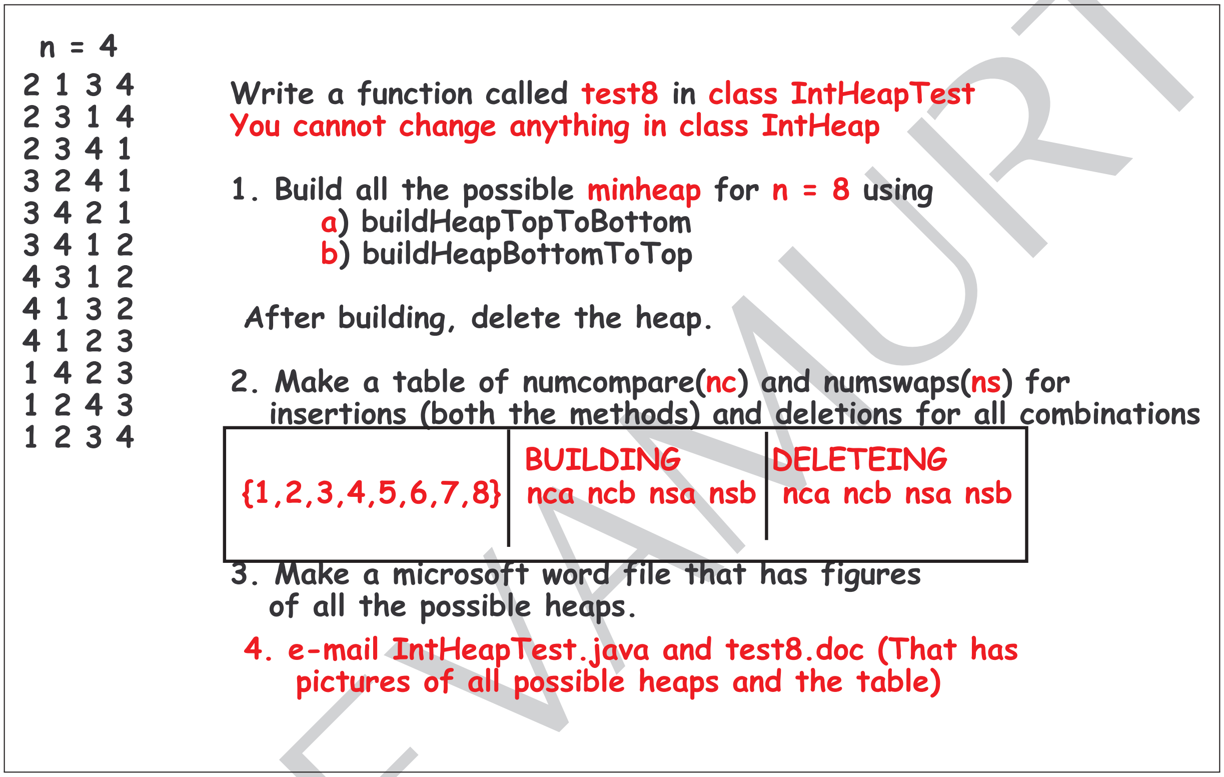
4. Construct max heap for
A = {2, 20,25,4,8,5,7,13,17}
1. Using top bottom construction
2. Using bottom up construction

5. For the heap constructed in Question 4
1. Illustrate the operation of HEAP SORT

6. Draw min heap for
A = {15,13,9,5,12,8,7,4,0,9,2,1} ;
1. Now insert 3 to the heap.
2. Delete top of the heap and show the
heap after deletion

Figure 10.15: Interview questions on heap

Problem 10.12.2. Implement a function called **test8** that builds and destroys all possible combinations of **heaps** as shown in figure 10.16.



n = 4

2 1 3 4
2 3 1 4
2 3 4 1
3 2 4 1
3 4 2 1
3 4 1 2
4 3 1 2
4 1 3 2
4 1 2 3
1 4 2 3
1 2 4 3
1 2 3 4

Write a function called test8 in class IntHeapTest
You cannot change anything in class IntHeap

1. Build all the possible minheap for n = 8 using
   a) buildHeapTopToBottom
   b) buildHeapBottomToTop

After building, delete the heap.

2. Make a table of numcompare(nc) and numswaps(ns) for insertions (both the methods) and deletions for all combinations

	BUILDING	DELETEING
{1,2,3,4,5,6,7,8}	nca ncb nsa nsb	nca ncb nsa nsb

3. Make a microsoft word file that has figures of all the possible heaps.

4. e-mail IntHeapTest.java and test8.doc (That has pictures of all possible heaps and the table)

Figure 10.16: All possible **minheap** of eight numbers