PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

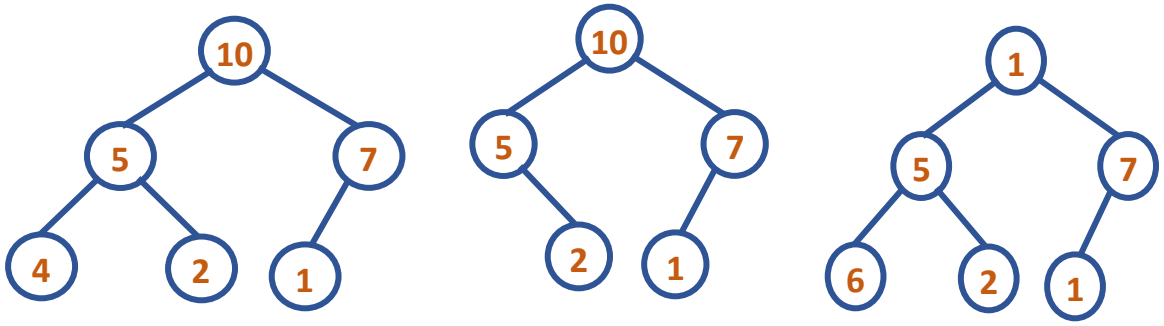
UE19CS202: Data Structures and its Applications (4-0-0-4-4)

Trees
Heap: Implementation using arrays

Dr. Shylaja S S
Ms. Kusuma K V

Heap: Concept and Implementation

Definition: A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:
1. The tree's shape requirement - The binary tree is essentially complete, that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing
2. The parental dominance requirement - The key at each node is greater than or equal to the keys at its children. (This condition is considered automatically satisfied for all leaves.)



In the above figures, only the left most binary tree is a heap. The binary tree in the middle is not a heap because it doesn't satisfy the shape requirement. The rightmost binary tree is not a heap because it doesn't satisfy the parental dominance requirement.

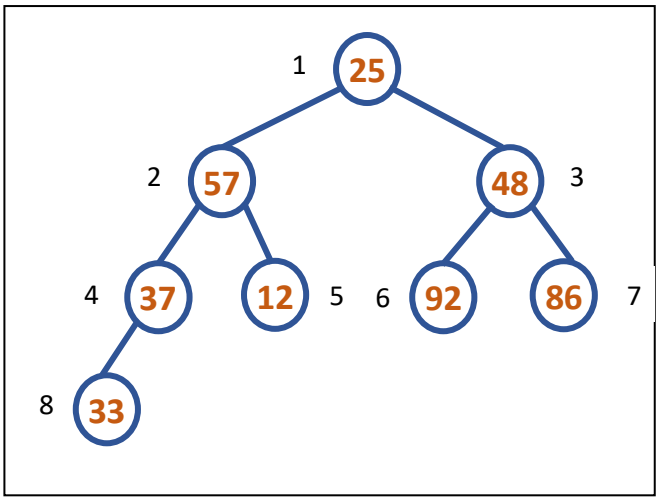
Properties of Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$
2. The root of a heap always contains its largest element
3. A node of a heap considered with all its descendants is also a heap
4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in the heap.
   In such a representation,
   a) The parental node keys will be in the first $\lfloor n/2 \rfloor$ positions of the array, while the leaf keys will occupy the last $\lceil n/2 \rceil$ positions
   b) The children of a key in the array's parental position i ($1 <= i <= \lfloor n/2 \rfloor$) will be in positions 2i and 2i + 1, and, correspondingly, the parent of a key in position i ($2 <= i <= n$) will be in position $\lfloor n/2 \rfloor$

Bottom up Heap Construction for the elements: 25, 57, 48, 37, 12, 92, 86, 33

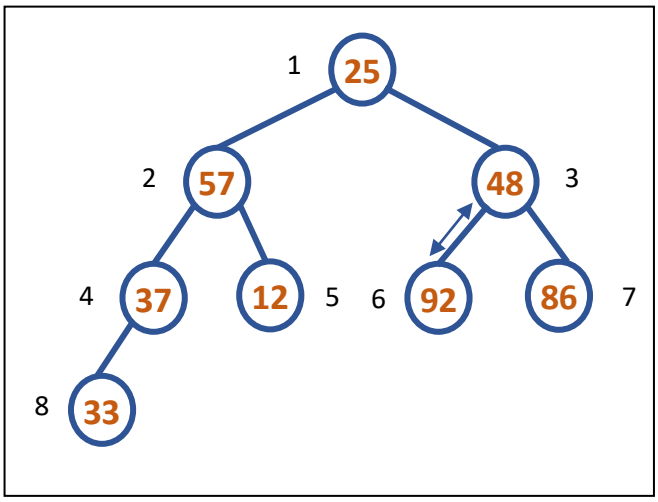
-	25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7	8

At k = 4, v = 37
Compare 37 with its only child 33
37 > 33, it's a heap at k=4



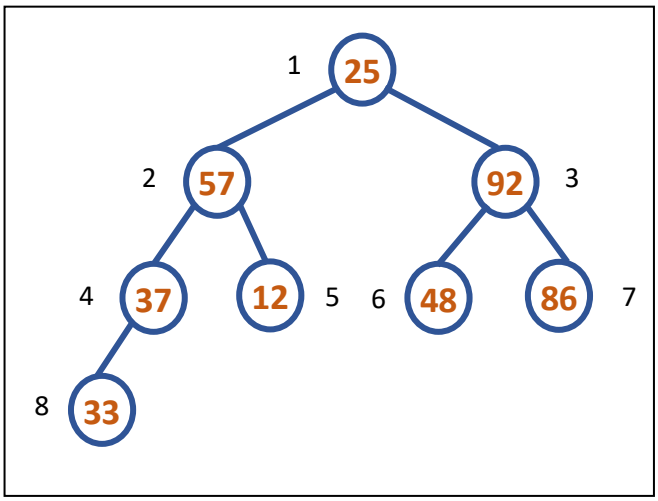
-	25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7	8

At k = 3, v = 48
Largest child: 92
Compare 48 with its largest child
48 < 92, Heapify



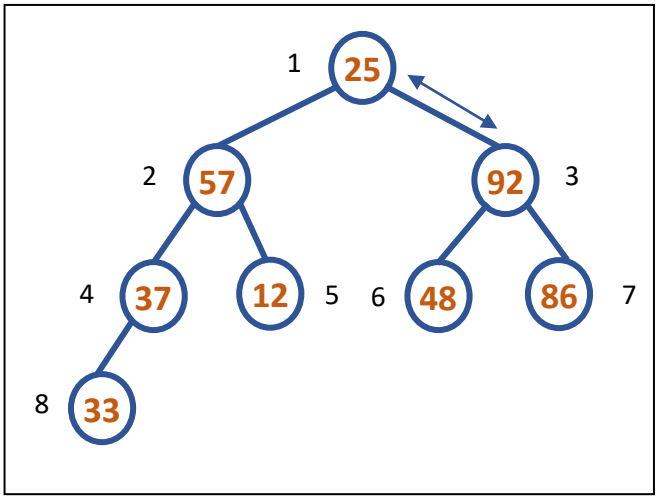
-	25	57	92	37	12	48	86	33
0	1	2	3	4	5	6	7	8

At k = 2, v = 57
Largest child: 37
Compare 57 with its largest child
57 > 37, It's a heap at k=2



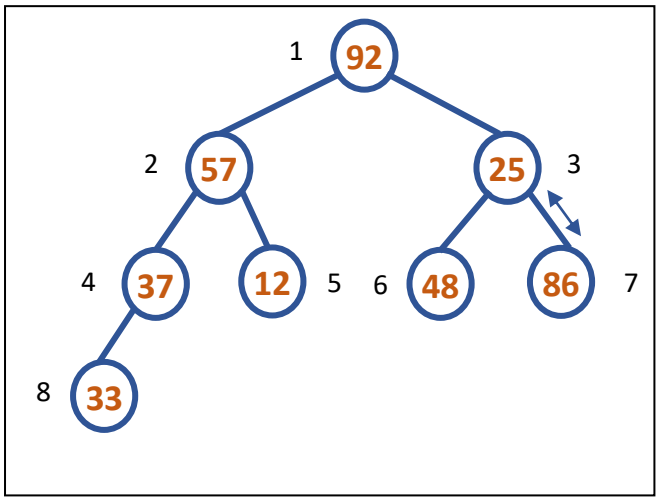
-	25	57	92	37	12	48	86	33
0	1	2	3	4	5	6	7	8

At k = 1, v = 25
Largest child:92
Compare 25 with its largest child
25 < 92, Heapify

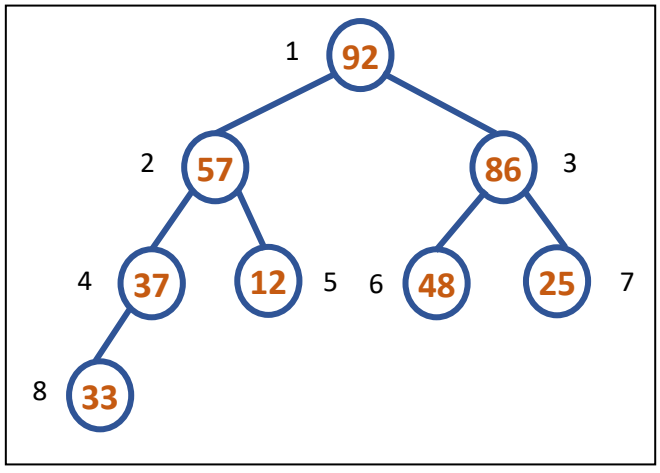


-	92	57	25	37	12	48	86	33
0	1	2	3	4	5	6	7	8

At k = 3, v = 25
Largest child:86
Compare 25 with its largest child
25 < 86, Heapify



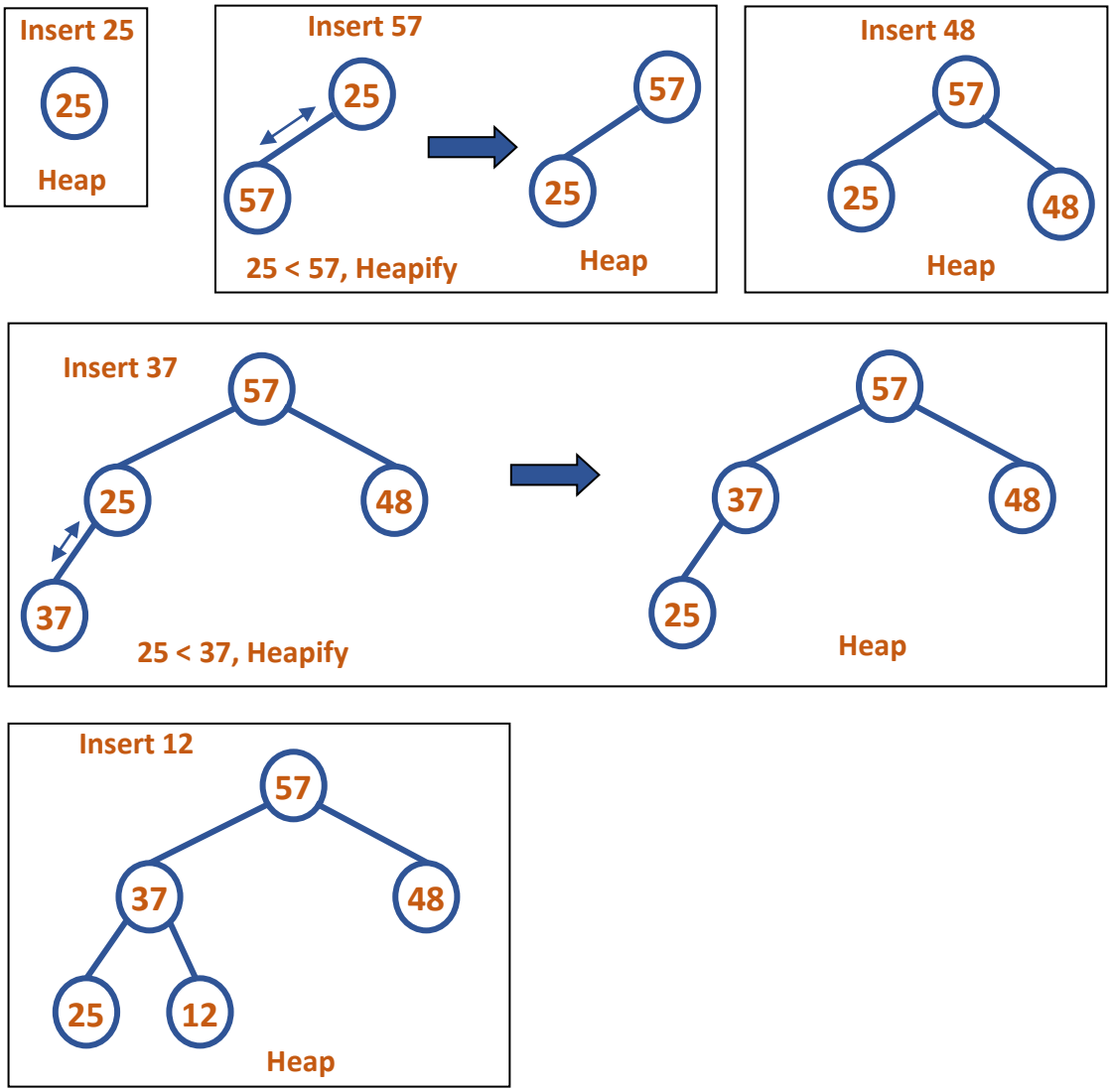
-	92	57	86	37	12	48	25	33
0	1	2	3	4	5	6	7	8

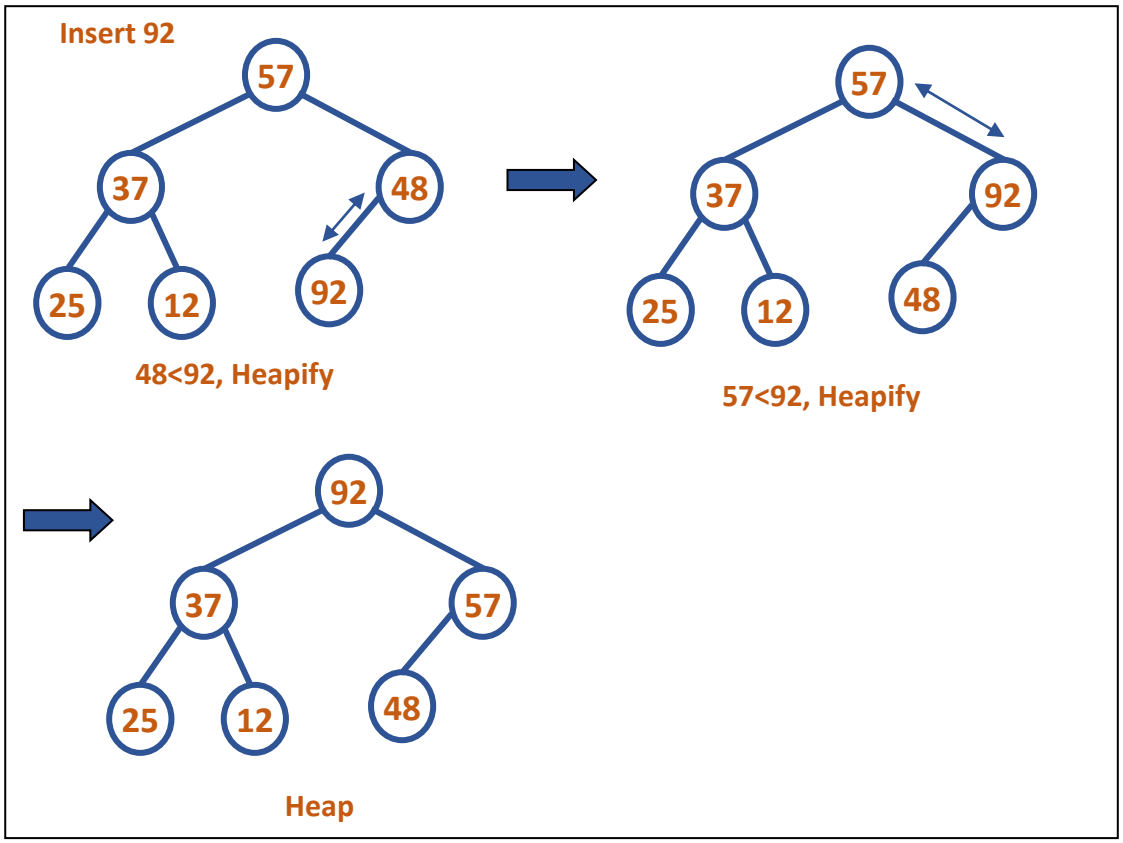


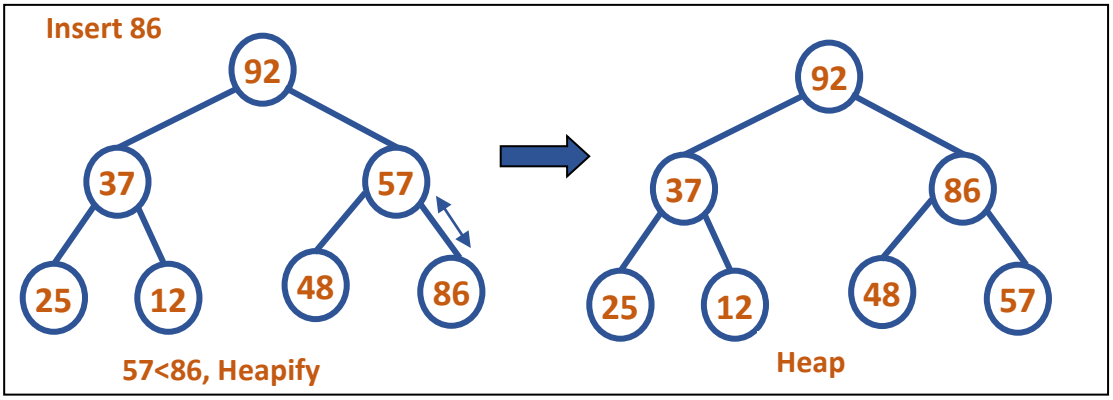
Bottom up Heap Constructed

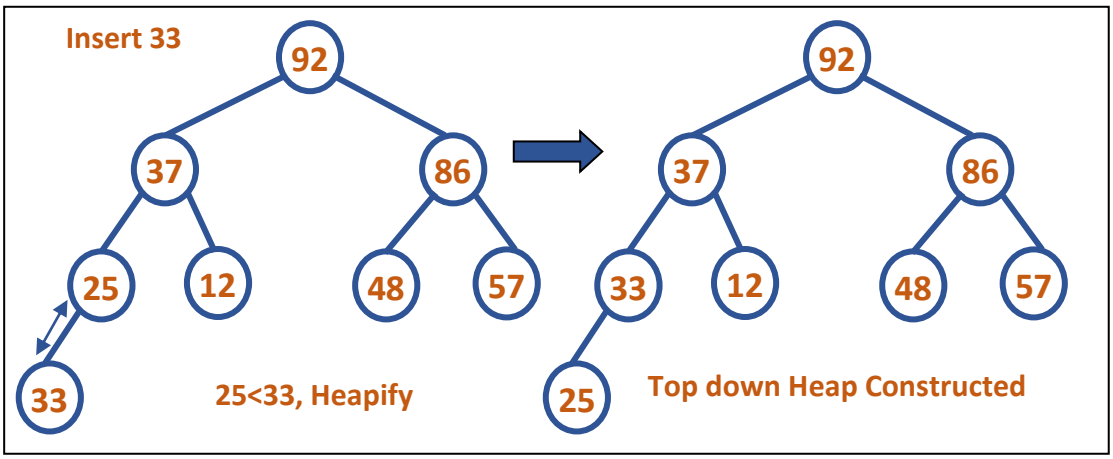
```
ALGORITHM HeapBottomUp(H[1…n])
//Constructs a heap from the elements of a given array by the bottom-up algorithm
//Input: An array H[1…n] of orderable items
//Output: A heap H[1…n]
for i = ⌊n/2⌋ downto 1 do
        k = i; v=H[k]; heap = false
        while not heap and 2*k ≤ n do
                j = 2*k
                if j < n                        //there are two children
                   if H[j] < H[j+1]   j = j+1
                if v ≥ H[j]
                   heap = true
                else H[k] = H[j];   k = j
        H[k] = v
```

Top down Heap Construction for the elements: 25, 57, 48, 37, 12, 92, 86, 33




Insert 92
57
37
48
25
12
92
48<92, Heapify
57
37
92
25
12
48
57<92, Heapify
92
37
57
25
12
48
Heap


Insert 86
92
37
57
25
12
48
86
57<86, Heapify
92
37
86
25
12
48
57
Heap


Insert 33
92
37
86
25
12
48
57
33
25<33, Heapify
92
37
86
33
12
48
57
25
Top down Heap Constructed

Top down Heap Construction

1. First, attach a new node with key K in it after the last leaf of the existing heap
2. Then sift K up to its appropriate place in the new heap as follows
3. Compare K with its parent's key: if the latter is greater than or equal to K, stop (the structure is a heap);
4. Otherwise, swap these two keys and compare K with its new parent.
5. This swapping continues until K is not greater than its last parent or it reaches the root.
6. In this algorithm, too, we can sift up an empty node until it reaches its proper position, where it will get K's value.

```
//C program to demonstrate top-down heap construction and heap sort
#include<stdio.h>
#define MAX 50
int main()
{
 int a[MAX];
 int i,c,p,n,elt;

 printf("enter the number of elements\n");
 scanf("%d",&n);

 printf("enter the elements\n");
 for(i=0;i<n;i++)
  scanf("%d",&a[i]);

 //Heapify
 for(i=1;i<n;i++)
  {
    elt=a[i];
    c=i;
    p=(c-1)/2;
   while(c>0 && a[p]<elt)
     {
       a[c]=a[p];
       c=p;
       p=(c-1)/2;
     }
   a[c]=elt;
  }
```

```
//display heapified elements
printf("\nElements of heap:\n");
for(i=0;i<n;i++)
 printf("%d ",a[i]);

//Heap sort
for(i=n-1;i>0;i--)
 {
  elt=a[i];
  a[i]=a[0];
  p=0;
 if(i==1)
  c=-1;
 else
  c=1;
 if(i>2 && a[2]>a[1])
  c=2;
 while(c>=0 && elt<a[c])
  {
   a[p]=a[c];
   p=c;
   c=2*p+1;
   if(c+1<=i-1 && a[c]<a[c+1])
    c=c+1;
   if(c>i-1) c=-1;
   }
  a[p]=elt;
 }

printf("\nSorted elements(Heap sort):\n");
for(i=0;i<n;i++)
 printf("%d ",a[i]);

printf("\n");

return 0;
}
```