Data Structure and Algorithm
handout 5
Heaps

Apurba Sarkar

October 25, 2017

1 Heap Sort

In this section we are going to talk about another tree structure called *heap*. It is used to implement an elegant sorting algorithm called *hepsort*. Suppose H is a complete binary tree with n elements. (Unless otherwise stated, we assume that H is maintained in memory by linear array TREE using sequential representation of H, not a linked representation.) Then H is called a *heap*, or a *maxheap*, if each node N of H has the following property: *The value at* N *is greater than or equal to the value at each of the children of* N. Accordingly, *the value at* N *is greater than or equal to the value at any of the descendants of N.* Likewise we define a *minheap* if each node N of H has the following property: *The value at* N *is less than or equal to the value at each of the children of* N.

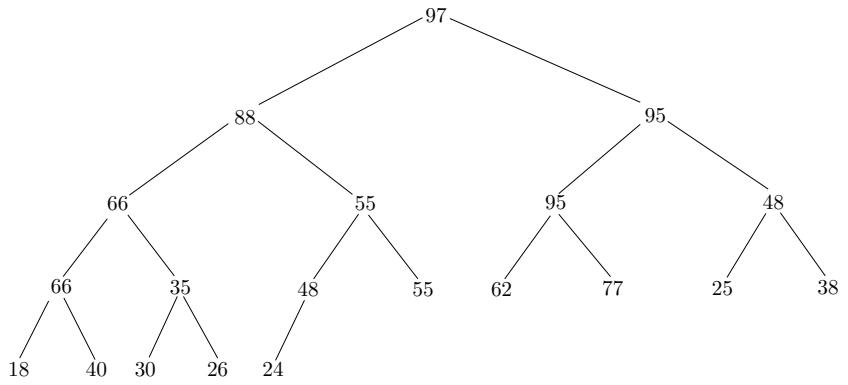
For example, consider the complete tree H in Fig. 1(a). Observe that H is a heap. This means, in particular that, the largest element in H appears at the top of the heap. Figure. 1(b) shows the sequential representation of H by the array TREE. That is TREE[1] is the root of H and the left and right children of TREE[K] are respectively at TREE[2K] and TREE[2K+1]. This means that the parent of any nonroot node TREE[J] is the node TREE[J÷2] (where J÷2 means integer division). Observe that the nodes of H on the same level appear one after the other in the array TREE.

1.1 Inserting into a Heap

To insert a new element into a *heap*, H, following two steps are followed

1. First adjoin the new element at the end of the H so that H still remains complete but not necessarily a *heap*
2. Then let the new element rise to its proper place in H so that H finally becomes a *heap*.

We illustrate the way this procedure works before stating the procedure formally. Consider the heap in Fig. 1. Suppose we want to insert 70 into it. First we add 70 as the next element in the complete tree that is TREE[21]= 70. Then 70 is the right child of TREE[10]= 48. The path from 70 to the root H is pictured in Fig. 2(a). The appropriate place of 70 is now found out as follows.



(a)

TREE	97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
	1	2	3							...										20

(b)

Figure 1: (a)Heap and (b) its sequential representation

1. compare 70 with its parent, 48. Since 70 is greater than 48, interchange 70 and 48. path will now look like Fig. 2(b)

2. compare 70 with its new parent, 55. Since 70 is greater than 55, interchange 70 and 55. path will now look like Fig. 2(c)

3. compare 70 with its new parent, 88. Since 70 is less than 88, 70 has risen to its proper place in H.

Figure. 2(d) shows the final tree.

Suppose we want to build a *heap* H from the following list of numbers:

$$44, 30, 50, 22, 60, 55, 77, 55$$

Let us now write down the procedure for inserting an element into a heap

```
procedure insHeap(int TREE[], int N, int ITEM)
// A heap H with N elements is stored in the array TREE, and an ITEM of
    information is given. This procedure inserts ITEM as a new elements
    of H. PTR gives the location of ITEM as it rises in the tree and
    PAR denotes the location of the parent of ITEM.
{  N = N+1;
   PTR = N;
   while(PTR < 1)
   {
      PAR = floor(PTR/2);
      if(ITEM <= TREE[PAR])
         TREE[PTR] = ITEM;
         return;
      TREE[PTR] = TREE[PAR];
      PTR = PAR;
   }
```

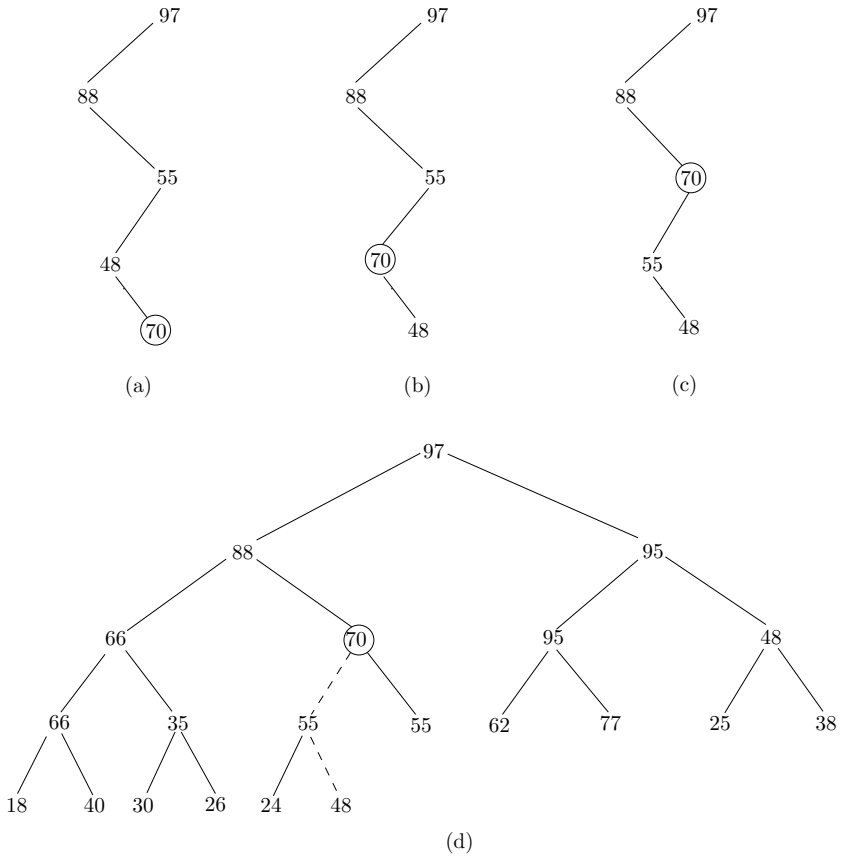


Figure 2: Insertion of 70 into the heap

```
   TREE[1]= ITEM;
}
```

1.2 Deleting the Root of a Heap

Suppose H is a heap with N elements, and suppose we want to delete the root R of H. The steps are as follows:

1. Assign the root R to some variable ITEM
2. Replace the deleted node R by the last node L of H so that H is still a complete tree, but not necessarily a heap.
3. (Reheap) Let L sink to its appropriate place in H so that H is finally a heap.

Again let us see how the procedure works with an example. Consider the heap H in Fig. 3(a), where R=95 is the root and L=22 is the last node in the tree. Step 1 of the above procedure deletes R=95 and step 2 replaces R=95 by L=22. This gives the complete tree of Fig. 3(b) which is not a heap. Observe, however,

that both right and left subtrees of 22 are still heaps. Applying Step 3, we find the appropriate place of 22 in the heap as follows.

1. compare 22 with is two children 85 and 70. Since 22 is less than larger child, 85, interchange 22 and 85 so the tree now looks like Fig. 3(c).

2. compare 22 with is two children 55 and 33. Since 22 is less than larger child, 55, interchange 22 and 55 so the tree now looks like Fig. 3(d).

3. compare 22 with is two children 15 and 20. Since 22 is greater than both children, node 22 has dropped to its appropriate place in H.

Thus, Fig. 3(d) is the required heap.

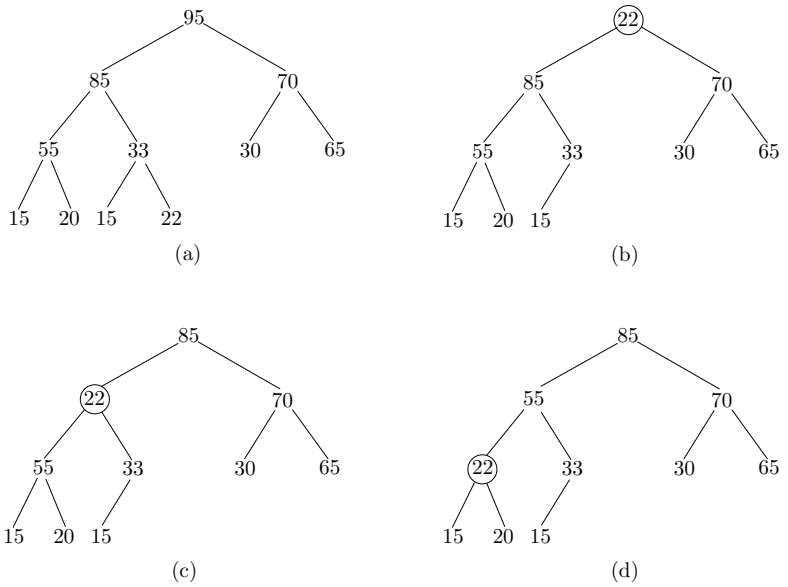


Figure 3: Deletion of R=95 from the heap

Let us now write down the entire procedure in a formal way.

```
procedure delHeap(int TREE[], int N, int ITEM)
// A heap H with N elements is stored in the array TREE. This procedure
    assigns the root TREE[1] of H to the variable ITEM and reheaps the
    remaining elements. The LAST saves the value of the original last
    node of H. The pointers PTR, LEFT and RIGHT give the location of
    LAST and its left and right children as LAST sinks in the tree.
{  ITEM = TREE[1]; // Remove root of H
   LAST = TREE[N];
   N = N-1;
   PTR = 1; LEFT = 2; RIGHT = 3;
   while(RIGHT <= N)
   {
      if(LAST >= TREE[LEFT] && LAST >= TREE[RIGHT])
         TREE[PTR] = LAST;
         return;
      if( TREE[RIGHT] <= TREE[LEFT];
```

```
      TREE[PTR] = TREE[LEFT];
      PTR = LEFT;
    else
      TREE[PTR] = TREE[RIGHT];
      PTR = RIGHT;
    LEFT = 2*PTR;
    RIGHT = 2*PTR + 1;
  }
  if(LEFT == N )
    if( LAST < TREE[LEFT])
      PTR = LEFT;
  TREE[PTR] = LAST;
}
```

1.3 Application to Sorting

Now let us see how heap can be used to sort elements. Suppose and array A with N elements is given. The heapsort algorithm to sort A consists of two phases as follows:

Phase A Build a heap H out of the elements of A.

Phase B Repeatedly delete the root of H.

Since the root of H always contains the largest element in H, Phase B deletes the element of A in the descending order. A formal presentation of the algorithm is as follows:

```
procedure heapSort(int A[], int N)
// An array A with N elements is given. This algorithm sorts the
    elements of A.
{
  for(j=1; j < N; j++)
  {
    insHeap(A, J, A[j+1]);
  }
  while(N > 1)
  {
    delHeap(A, N, ITEM);
    B[N+1] = ITEM;
    N--;
  }
}
```