International Islamic University Chittagong
Department of Computer Science & Engineering
Spring - 2021

Course Code: CSE-2321
Course Title: Data Structures

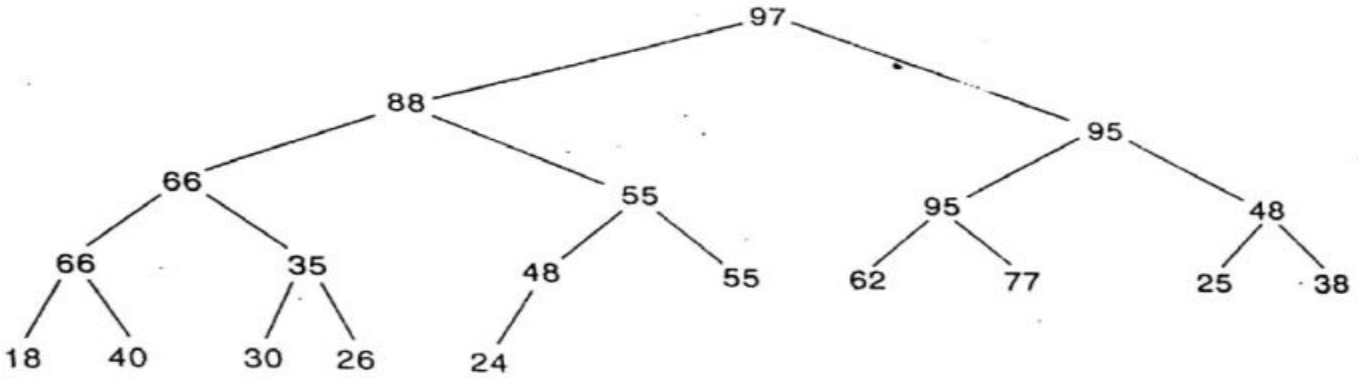
Mohammed Shamsul Alam
Professor, Dept. of CSE, IIUC

Lecture – 15

Tree 2

Heap

Suppose H is a complete binary tree with n elements. (Unless otherwise stated, we assume that H is maintained in memory by a linear array TREE using the 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. (A *minheap* is defined analogously: The value at N is less than or equal to the value at any of the children of N.)



(a) Heap

TREE

97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

(b) Sequential representation

Fig. 7.58

Inserting into a Heap

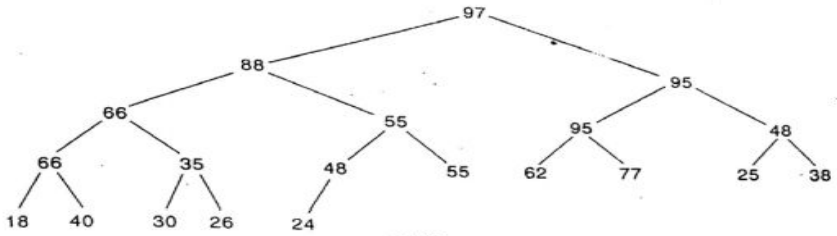
Suppose H is a heap with N elements, and suppose an ITEM of information is given. We insert ITEM into the heap H as follows:

(1) First adjoin ITEM at the end of H so that H is still a complete tree, but not necessarily a heap.
(2) Then let ITEM rise to its "appropriate place" in H so that H is finally a heap.

We illustrate the way this procedure works before stating the procedure formally.

Example 7.33

Consider the heap H in Fig. 7.58. Suppose we want to add ITEM = 70 to H. First we adjoin 70 as the next element in the complete tree; that is, we set TREE[21] = 70. Then 70 is the right child of TREE[10] = 48. The path from 70 to the root of H is pictured in Fig. 7.59(a). We now find the appropriate place of 70 in the heap as follows:



(a) Heap

TREE

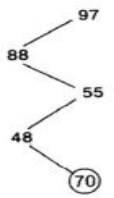
97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

(b) Sequential representation

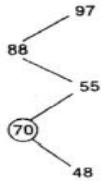
Fig. 7.58

(a) Compare 70 with its parent, 48. Since 70 is greater than 48, interchange 70 and 48; the path will now look like Fig. 7.59(b).

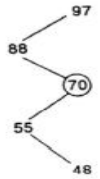
(b) Compare 70 with its new parent, 55. Since 70 is greater than 55, interchange 70 and 55; the path will now look like Fig. 7.59(c).



(a)



(b)



(c)

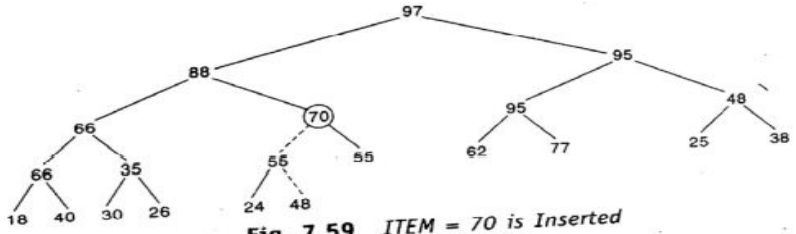


Fig. 7.59 *ITEM = 70 is Inserted*

Remark: One must verify that the above procedure does always yield a heap as a final tree, that is, that nothing else has been disturbed. This is easy to see, and we leave this verification to the reader.

Example 7.34

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

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

This can be accomplished by inserting the eight numbers one after the other into an empty heap H using the above procedure. Figure 7.60(a) through (h) shows the respective pictures of the heap after each of the eight elements has been inserted. Again, the dotted line indicates that an exchange has taken place during the insertion of the given ITEM of information.

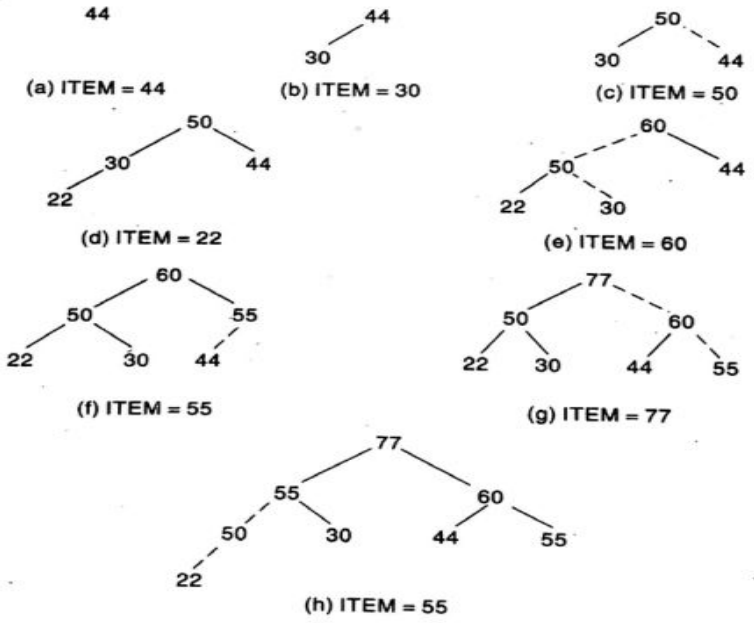


Fig. 7.60 *Building a Heap*

The formal statement of our insertion procedure follows:

Procedure 7.9: INSHEAP(TREE, N, 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 element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

1. [Add new node to H and initialize PTR.]
Set N := N + 1 and PTR := N.
2. [Find location to insert ITEM.]
Repeat Steps 3 to 6 while PTR < 1.
3. Set PAR := ⌊PTR/2⌋. [Location of parent node.]
4. If ITEM ≤ TREE[PAR], then:
Set TREE[PTR] := ITEM, and Return.
[End of If structure.]
5. Set TREE[PTR] := TREE[PAR]. [Moves node down.]
6. Set PTR := PAR. [Updates PTR.]
[End of Step 2 loop.]
7. [Assign ITEM as the root of H.]
Set TREE[1] := ITEM.
8. Return.

Observe that ITEM is not assigned to an element of the array TREE until the appropriate place for ITEM is found. Step 7 takes care of the special case that ITEM rises to the root TREE[1].

Suppose an array A with N elements is given. By repeatedly applying Procedure 7.9 to A, that is, by executing

Call INSHEAP(A, J, A[J + 1])

for J = 1, 2, ..., N − 1, we can build a heap H out of the array A.

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. This is accomplished 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 we illustrate the way the procedure works before stating the procedure formally.

Example 7.35

Consider the heap H in Fig. 7.61(a), where R = 95 is the root and L = 22 is the last node of 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 in Fig. 7.61(b), which is not a heap. Observe, however, that both the right and left subtrees of 22 are still heaps. Applying Step 3, we find the appropriate place of 22 in the heap as follows:

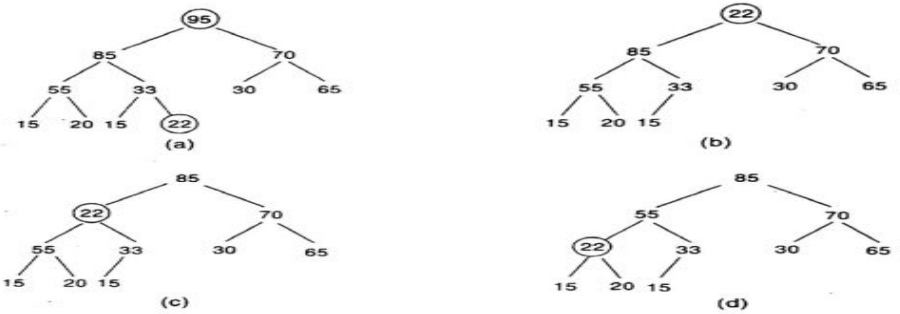


Fig. 7.61 *Reheaping*

(a) Compare 22 with its two children, 85 and 70. Since 22 is less than the larger child, 85, interchange 22 and 85 so the tree now looks like Fig. 7.61(c).
(b) Compare 22 with its two new children, 55 and 33. Since 22 is less than the larger child, 55, interchange 22 and 55 so the tree now looks like Fig. 7.61(d).
(c) Compare 22 with its new children, 15 and 20. Since 22 is greater than both children, node 22 has dropped to its appropriate place in H.

Thus Fig. 7.61(d) is the required heap H without its original root R.

Remark: As with inserting an element into a heap, one must verify that the above procedure does always yield a heap as a final tree. Again we leave this verification to the reader. We also note that Step 3 of the procedure may not end until the node L reaches the bottom of the tree, i.e., until L has no children.

The formal statement of our procedure follows.

Procedure 7.10: DELHEAP(TREE, N, 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 then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PTR, LEFT and RIGHT give the locations of LAST and its left and right children as LAST sinks in the tree.

1. Set ITEM := TREE[1]. [Removes root of H.]
2. Set LAST := TREE[N] and N := N – 1. [Removes last node of H.]
3. Set PTR := 1, LEFT := 2 and RIGHT := 3. [Initializes pointers.]
4. Repeat Steps 5 to 7 while RIGHT ≤ N:
5. If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT], then:
Set TREE[PTR] := LAST and Return.
[End of If structure.]
6. IF TREE[RIGHT] ≤ TREE[LEFT], then:
Set TREE[PTR] := TREE[LEFT] and PTR := LEFT.
Else:
Set TREE[PTR] := TREE[RIGHT] and PTR := RIGHT.
[End of If structure.]
7. Set LEFT := 2*PTR and RIGHT := LEFT + 1.
[End of Step 4 loop.]
8. If LEFT = N and if LAST < TREE[LEFT], then: Set PTR := LEFT.
9. Set TREE[PTR] := LAST.
10. Return.

The Step 4 loop repeats as long as LAST has a right child. Step 8 takes care of the special case in which LAST does not have a right child but does have a left child (which has to be the last node in H). The reason for the two "If" statements in Step 8 is that TREE[LEFT] may not be defined when LEFT > N.

Application to Sorting

Suppose an array A with N elements is given. The heapsort algorithm to sort A consists of the two following phases:

Phase A: Build a heap H out of the elements of A.
Phase B: Repeatedly delete the root element of H.

Since the root of H always contains the largest node in H, Phase B deletes the elements of A in decreasing order. A formal statement of the algorithm, which uses Procedures 7.9 and 7.10, follows.

Algorithm 7.11: HEAPSORT(A, N)
An array A with N elements is given. This algorithm sorts the elements of A.

1. [Build a heap H, using Procedure 7.9.]
   Repeat for J = 1 to N − 1:
   Call INSHEAP(A, J, A[J + 1]).
   [End of loop.]
2. [Sort A by repeatedly deleting the root of H, using Procedure 7.10.]
   Repeat while N > 1:
   (a) Call DELHEAP(A, N, ITEM).
   (b) Set A[N + 1] := ITEM.
   [End of Loop.]
3. Exit.

The purpose of Step 2(b) is to save space. That is, one could use another array B to hold the sorted elements of A and replace Step 2(b) by

Set B[N + 1] := ITEM

However, the reader can verify that the given Step 2(b) does not interfere with the algorithm, since A[N + 1] does not belong to the heap H.

Complexity of Heapsort

Suppose the heapsort algorithm is applied to an array A with n elements. The algorithm has two phases, and we analyze the complexity of each phase separately.

Phase A. Suppose H is a heap. Observe that the number of comparisons to find the appropriate place of a new element ITEM in H cannot exceed the depth of H. Since H is a complete tree, its depth is bounded by $\log_2 m$ where m is the number of elements in H. Accordingly, the total number $g(n)$ of comparisons to insert the n elements of A into H is bounded as follows:

$$g(n) \leq n \log_2 n$$

Consequently, the running time of Phase A of heapsort is proportional to $n \log_2 n$.

Phase B. Suppose H is a complete tree with m elements, and suppose the left and right subtrees of H are heaps and L is the root of H. Observe that reheaping uses 4 comparisons to move the node L one step down the tree H. Since the depth of H does not exceed $\log_2 m$, reheaping uses at most 4 $\log_2 m$ comparisons to find the appropriate place of L in the tree H. This means that the total number $h(n)$ of comparisons to delete the n elements of A from H, which requires reheaping n times, is bounded as follows:

$$h(n) \leq 4n \log_2 n$$

Accordingly, the running time of Phase B of heapsort is also proportional to $n \log_2 n$.

Since each phase requires time proportional to $n \log_2 n$, the running time to sort the n-element array A using heapsort is proportional to $n \log_2 n$, that is, $f(n) = O(n \log_2 n)$. Observe that this gives a worst-case complexity of the heapsort algorithm. This contrasts with the following two sorting algorithms already studied:

(1) *Bubble sort* (Sec. 4.6). The running time of bubble sort is $O(n^2)$.
(2) *Quicksort* (Sec. 6.5). The average running time of quicksort is $O(n \log_2 n)$, the same as heapsort, but the worst-case running time of quicksort is $O(n^2)$, the same as bubble sort.

Other sorting algorithms are investigated in Chapter 9.

7.19 GENERAL TREES

A *general tree* (sometimes called a *tree*) is defined to be a nonempty finite set T of elements, called *nodes*, such that:

(1) T contains a distinguished element R, called the *root* of T.
(2) The remaining elements of T form an ordered collection of zero or more disjoint trees $T_1, T_2, \ldots, T_m$.

The trees $T_1, T_2, \ldots, T_m$ are called *subtrees* of the root R, and the roots of $T_1, T_2, \ldots, T_m$ are called *successors* of R.

Terminology from family relationships, graph theory and horticulture is used for general trees in the same way as for binary trees. In particular, if N is a node with successors $S_1, S_2, \ldots, S_m$, then N is called the *parent* of the S_i's, the S_i's are called *children* of N, and the S_i's are called *siblings* of each other.

The term "tree" comes up, with slightly different meanings, in many different areas of mathematics and computer science. Here we assume that our general tree T is *rooted*, that is, that T has a distinguished node R called the root of T; and that T is *ordered*, that is, that the children of each node N of T have a specific order. These two properties are not always required for the definition of a tree.

Example 7.38

Figure 7.68 pictures a general tree T with 13 nodes,

$$A, B, C, D, E, F, G, H, J, K, L, M, N$$

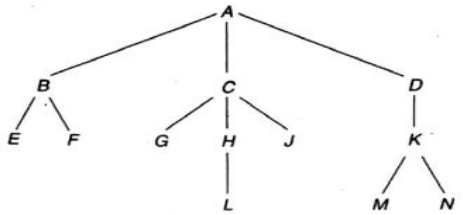


Fig. 7.68

Unless otherwise stated, the root of a tree T is the node at the top of the diagram, and the children of a node are ordered from left to right. Accordingly, A is the root of T, and A has three children; the first child B, the second child C and the third child D. Observe that:

(a) The node C has three children.
(b) Each of the nodes B and K has two children.
(c) Each of the nodes D and H has only one child.
(d) The nodes E, F, G, L, J, M and N have no children.

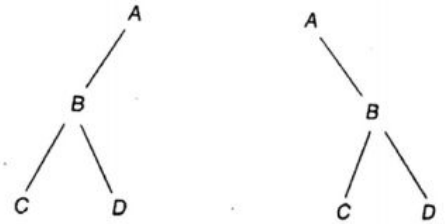
The last group of nodes, those with no children, are called *terminal nodes*.

A binary tree T' is not a special case of a general tree T: binary trees and general trees are different objects. The two basic differences follow:

(1) A binary tree T' may be empty, but a general tree T is nonempty.
(2) Suppose a node N has only one child. Then the child is distinguished as a left child or right child in a binary tree T', but no such distinction exists in a general tree T.

The second difference is illustrated by the trees T_1 and T_2 in Fig. 7.69. Specifically, as binary trees, T_1 and T_2 are distinct trees, since B is the left child of A in the tree T_1 but B is the right child of A in the tree T_2. On the other hand, there is no difference between the trees T_1 and T_2 as general trees.

A *forest* F is defined to be an ordered collection of zero or more distinct trees. Clearly, if we delete the root R from a general tree T, then we obtain the forest F consisting of the subtrees of R (which may be empty). Conversely, if F is a forest, then we may adjoin a node R to F to form a general tree T where R is the root of T and the subtrees of R consist of the original trees in F.



(a) Tree T_1 (b) Tree T_2

Fig. 7.69

Computer Representation of General Trees

Suppose T is a general tree. Unless otherwise stated or implied, T will be maintained in memory by means of a linked representation which uses three parallel arrays INFO, CHILD (or DOWN) and SIBL (or HORZ), and a pointer variable ROOT as follows. First of all, each node N of T will correspond to a location K such that:

(1) INFO[K] contains the data at node N.
(2) CHILD[K] contains the location of the first child of N. The condition CHILD[K] = NULL indicates that N has no children.
(3) SIBL[K] contains the location of the next sibling of N. The condition SIBL[K] = NULL indicates that N is the last child of its parent.

Furthermore, ROOT will contain the location of the root R of T. Although this representation may seem artificial, it has the important advantage that each node N of T, regardless of the number of children of N, will contain exactly three fields.

The above representation may easily be extended to represent a forest F consisting of trees T_1, T_2, ..., T_m by assuming the roots of the trees are siblings. In such a case, ROOT will contain the location of the root R_1 of the first tree T_1; or when F is empty, ROOT will equal NULL.

Example 7.39

Consider the general tree T in Fig. 7.68. Suppose the data of the nodes of T are stored in an array INFO as in Fig. 7.70(a). The structural relationships of T are obtained by assigning values to the pointer ROOT and the arrays CHILD and SIBL as follows:

(a) Since the root A of T is stored in INFO[2], set ROOT := 2.
(b) Since the first child of A is the node B, which is stored in INFO[3], set CHILD[2] := 3. Since A has no sibling, set SIBL[2] := NULL.
(c) Since the first child of B is the node E, which is stored in INFO[15], set CHILD[3] := 15. Since node C is the next sibling of B and C is stored in INFO[4], set SIBL[3] := 4.

	INFO
1	
2	A
3	B
4	C
5	
6	G
7	H
8	J
9	N
10	M
11	L
12	K
13	
14	F
15	E
16	D

(a)

	CHILD	SIBL
1	5	
2	3	0
3	15	4
4	6	16
5	13	
6	0	7
7	11	8
8	0	0
9	0	0
10	0	9
11	0	0
12	10	0
13	0	
14	0	0
15	0	14
16	12	0

ROOT = 2, AVAIL = 13

(b)

Fig. 7.70

And so on. Figure 7.70(b) gives the final values in CHILD and SIBL. Observe that the AVAIL list of empty nodes is maintained by the first array, CHILD, where AVAIL = 1.

Correspondence between General Trees and Binary Trees

Suppose T is a general tree. Then we may assign a unique binary tree T' to T as follows. First of all, the nodes of the binary tree T' will be the same as the nodes of the general tree T, and the root of T' will be the root of T. Let N be an arbitrary node of the binary tree T'. Then the left child of N in T' will be the first child of the node N in the general tree T and the right child of N in T' will be the next sibling of N in the general tree T.

Example 7.40

Consider the general tree T in Fig. 7.68. The reader can verify that the binary tree T' in Fig. 7.71 corresponds to the general tree T. Observe that by rotating counterclockwise the picture of T' in Fig. 7.71 until the edges pointing to right children are horizontal, we obtain a picture in which the nodes occupy the same relative position as the nodes in Fig. 7.68.

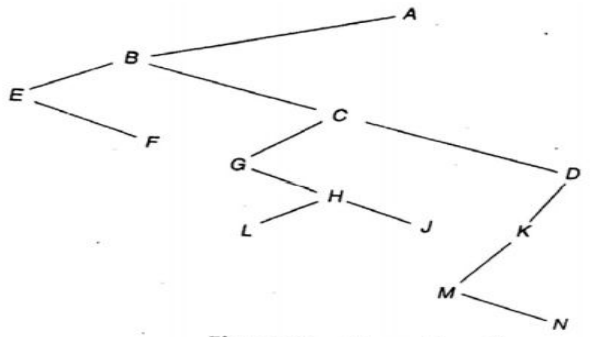


Fig. 7.71 *Binary Tree T'*

The computer representation of the general tree T and the linked representation of the corresponding binary tree T' are exactly the same except that the names of the arrays CHILD and SIBL for the general tree T will correspond to the names of the arrays LEFT and RIGHT for the binary tree T'. The importance of this correspondence is that certain algorithms that applied to binary trees, such as the traversal algorithms, may now apply to general trees.