BCS304 - **DATA STRUCTURES AND APPLICATIONS**

Module-5

HASHING: Introduction, Static Hashing, Dynamic Hashing, **PRIORITY QUEUES:** Single and double ended Priority Queues, Leftist Trees, **INTRODUCTION TO EFFICIENT BINARY SEARCH TREES:** Optimal Binary Search Trees

Hashing

- Hashing is an effective way to store and retrieve data in some data structure.
- Hashing technique is designed to use a special function called the hash function which is used to map a given value with a particular key for faster access of elements.
- The efficiency of mapping depends on the efficiency of the hash function used.

Types of Hashing Techniques:

a. Static Hashing b. Dynamic Hashing

a. Static Hashing:

- Is a hashing technique in which the table(bucket) size remains the same (Fixed during compilation time) is called static hashing?
- Various techniques of static hashing are linear probing and chaining
- As the size is fixed, this type of hashing consist handling overflow of elements (Collision) efficiently.

Example:

Elements to be stored: 24, 93, 45

H (24) = 24 % 10 = 4

H (93) = 93 % 10 = 3

H (45) = 45 % 10 = 5

0	
1	
2	
3	93
4	24
5	45
6	
7	
8	
9	

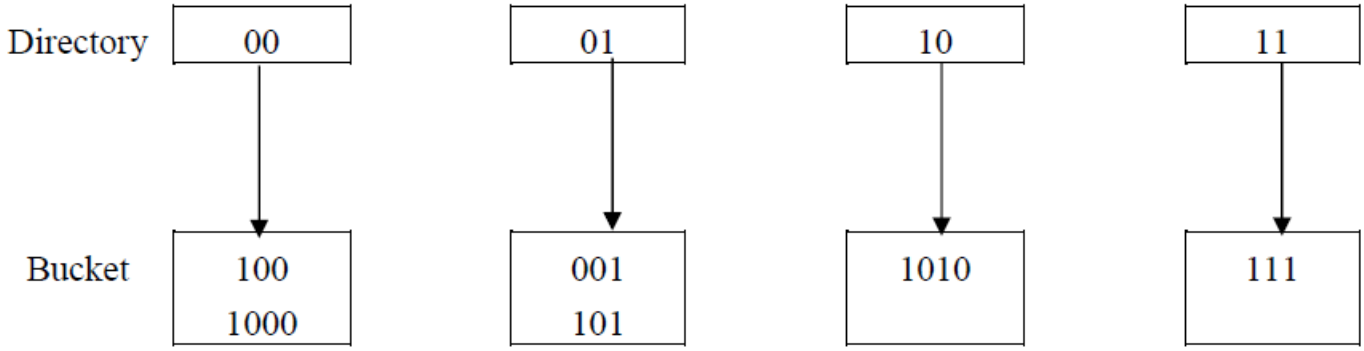
b. Dynamic Hashing:

- This is an hashing technique in which the bucket(table) size is not fixed. It can grow or shrink according to the increase or decrease of records.
- Typical example of dynamic hashing is **Extensible hashing**.
- Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.
- In extensible hashing referring to the size of directory the elements are to be placed in buckets.

Example:

To insert 1,4,5,7,8,10.

Assume each page (bucket) can hold 2 data entries



Overflow Handling (Collision Resolution Techniques):

- The phenomenon of two or more keys being hashed to the same location of the hash table (Fixed size table) is called collision.

Example:

The data elements to be placed: 44, 73, 17, 77

Hash function be K%10

0	
1	
2	
3	73
4	44
5	
6	
7	17
8	
9	

- If we try to place 77 in the hash table, we get the hash value 7 and and index 7 already 17 is placed. This situation is called collision.

- The various technique using which collision can be avoided are:

  1. Open Addressing (Linear Probing)

  2. Chaining

1. Open Addressing (Linear Probing)

- Here, it involves static hashing, hash table size is fixed. In such a atble, collision can be avoided by finding another unoccupied location in the array.
- The collision can be avoided using Linear Probing

Example:

Let size of hash table = 100. Let the hash function: h(k)=k % m, m is the size of hash table.

Items to be inserted: 1234, 2548, 3256, 1299, 1298, 1398

h (1234)	=1234%100	=34	
h (2548)	=2548%100	=48	
h (3256)	=3256%100	=56	
h (1299)	=1299%100	=99	
h (1298)	=1298%100	=98	
h (1398)	=1398%100	=98	Collision

0	1	2	.....	34	.....	48	.....	56	.....	98	99
				1234		2548		3256		1298	1299

- Collision is detected while inserting 1398 into 98th position. To overcome this linear probing may be used. In linear probing, it checks for the next available(empty) location. So, the next available location is 0.

0	1	2	.....	34	.....	48	.....	56	.....	98	99
1398				1234		2548		3256		1298	1299

2. Chaining Method:

- Chaining technique avoids collision using an array of liked lists (run time).
- If more than one key has same hash value, then all the keys will be inserted at the end of the list (insert rear) one by one and thus collision is avoided.

Example:

Construct a hash table of size and store the following words: like, a, tree, you, first, a, place, to

Let $H(str)=P_0+P_1+P_2+\ldots\ldots+P_{n-1}$; where P_i is position of letter in English alphabet series. Then calculate the hash address = Sum % 5

H(like) = 12 + 9 + 11 + 5 = 37 % 5 = 2

H(a) = 1 %5 =1

H(tree) = 20 + 18 + 5 + 5 = 48 % 5 = 3

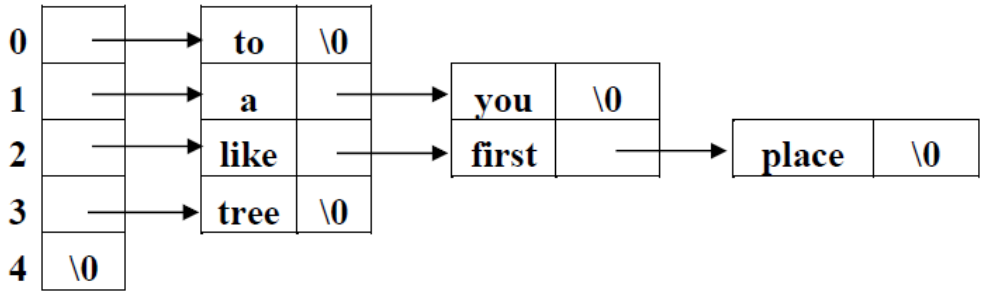
H(you) = 25 + 15 + 21 = 61 % 5 = 1

H(first) = 6 + 9 + 18 + 19 + 20 = 72 % 5 = 2

H(a) = 1 % 5 = 1

H(place) = 16 + 12 + 1 + 3 + 5 = 37 % 5 =2

H(to) = 20 + 15 = 35 % 5 = 0



Address Calculation Sort

- Uses Hashing technique.
- The hash function should have the property that x1 ≤ x2, the hash(x1) ≤ hash(x2). The function which exhibits this property is called order processing or Non-decreasing hashing function.

Example: 25, 57, 48, 37, 12, 92, 86, 33

- Here the largest number is 92, so we divide all the elements using the hash function h(k) = k/10.

H (25) = 25 / 10 = 2

H (57) = 57 / 10 = 5

H (48) = 48 / 10 = 4

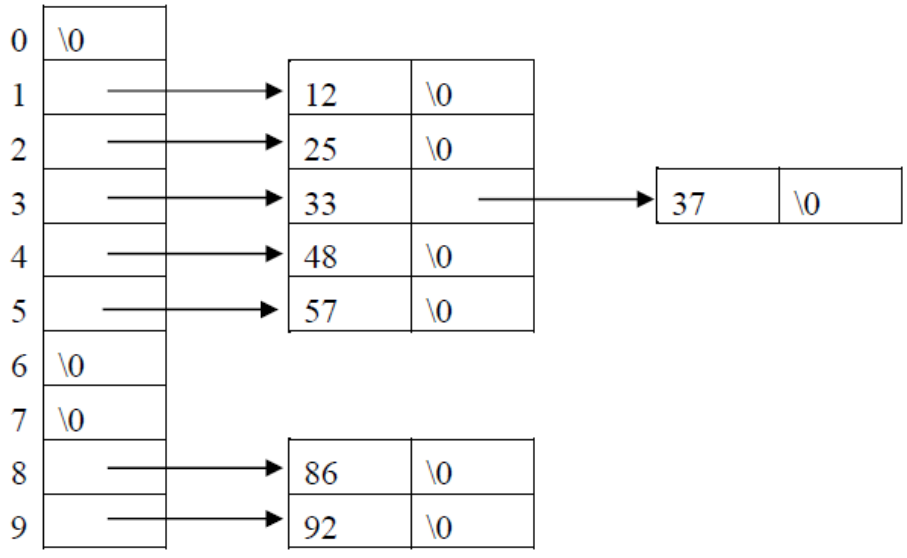
H (37) = 37 / 10 = 3

H (12) = 12 / 10 = 1

H (92) = 92 / 10 = 9

H (86) = 86 / 10 = 8

H (33) = 33 / 10 = 3



- Here 3 which is repeated. It is inserted in 3rd sub file only, but must be checked with the existing elements for its proper position in this sub file.\

Priority Queues

- A priority queue is
  - Collection of zero or more elements with priority
- A min priority queue is
  - Find the element with minimum priority
  - Then, Remove the element
- A max priority queue is
  - Find the element with maximum priority
  - Then, Remove the element

- Priority queue is a conceptual queue where the output element has a certain property (i.e., priority)

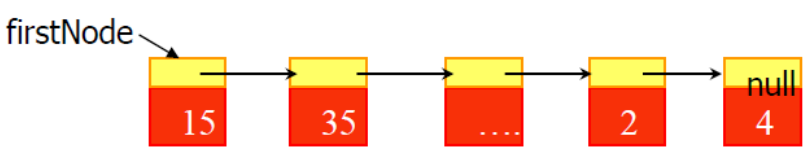
The ADT MaxPriorityQueue

```
AbstractDataType MaxPriorityQueue {
    instances
        finite collection of elements, each has a priority
    operations
        isEmpty()    : return true if the queue is empty
        size()       : return number of elements in the queue
        getMax()     : return element with maximum priority
        put(x)       : insert the element x into the queue
        removeMax()  : remove the element with largest priority
                       from the queue and return this element;
}
```

Linear Lists for Priority Queue

- Suppose Linear List for max priority queue with n elements
- Unordered linear list for a max queue
  - Array
    - Insert() or Put() : Θ(1) // put the new element the right end of the array
    - RemoveMax(): Θ(n) // find the max among n elements

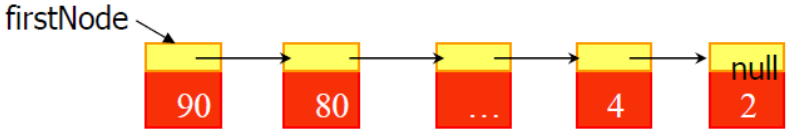
  - Linked List
    - Insert() or Put() : Θ(1) // put the new element at the front of the chain
    - RemoveMax(): Θ(n) // find the max among n elements



Ordered linear list for a max queue

- Array
  - location(i) = i (i.e., array-based) where the max element is located in the last address (i.e., the nondecreasing order)
  - Insert() or Put() : $\Theta(n)$
  - RemoveMax() : $\Theta(1)$

- Linked List
  - chain (i.e., linked) where the max element is located in the head of chain (i.e., the nonincreasing order)
  - Insert() or Put() : $\Theta(n)$
  - RemoveMax() : $\Theta(1)$



SNU

Max Tree & Max Heap

- A max tree is a tree in which the value in each node is greater than or equal to those in its children

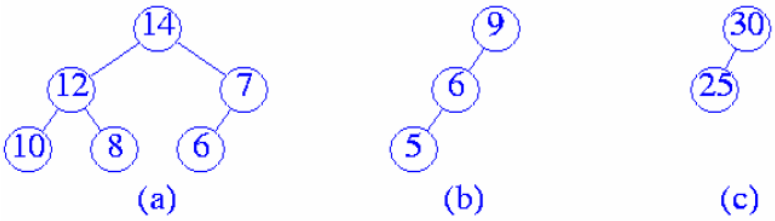


Figure 13.1 Max trees

- A max heap is
  - A max tree that is also a complete binary tree
  - Figure 13.1(b) : not CBT, so not max heap

Min Tree & Min Heap

- A min tree is a tree in which the value in each node is less than or equal to those in its children

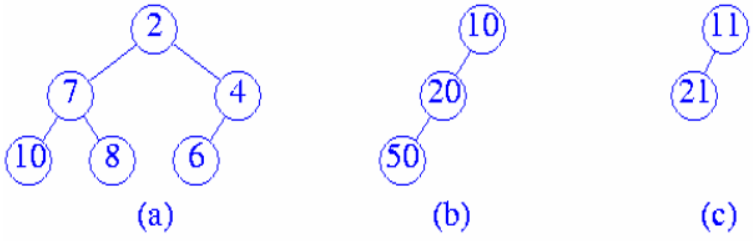


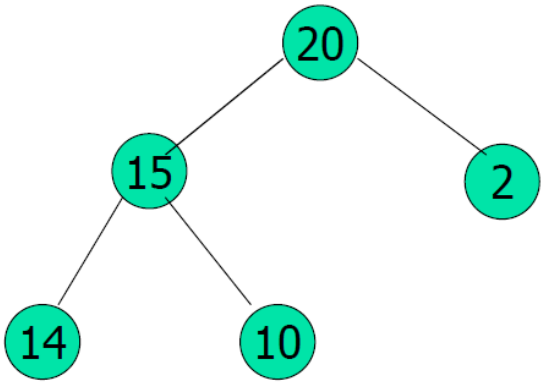
Figure 13.2 Min trees

- A min heap is
  - A min tree that is also a complete binary tree
  - Figure 13.2(b) : not CBT, so not min heap

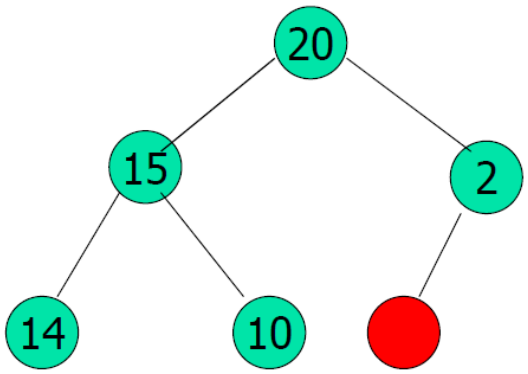
- Heap is a complete binary tree
  - **A heap with n elements has height** $\lceil \log_2(n+1) \rceil$
- put(): 0(height) ➔ 0(log n)
  - Increase array size if necessary
  - Find place for the new element
    - The new element is located as a leaf
    - Then moves up the tree for finding home
- removeMax(): 0(height) ➔ 0(log n)
  - Remove heap[1], so the root is empty
  - Move the last element in the heap to the root
  - Reheapify

Put into Max Heap

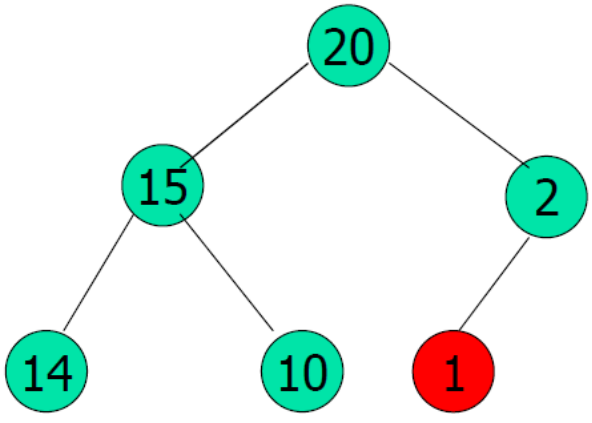
Max heap with five elements



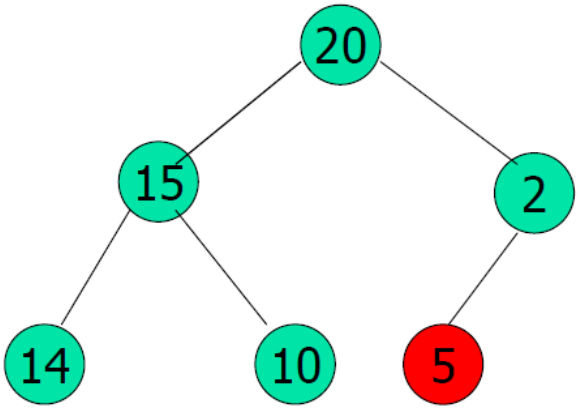
- When an element is added to this heap, the location for a new element is the red zone



- Suppose the element to be inserted has value 1, the following placement is fine

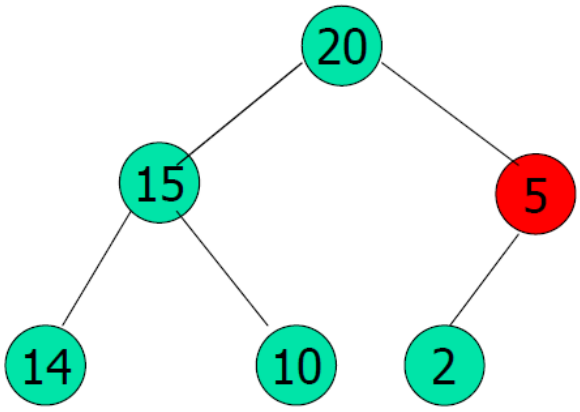


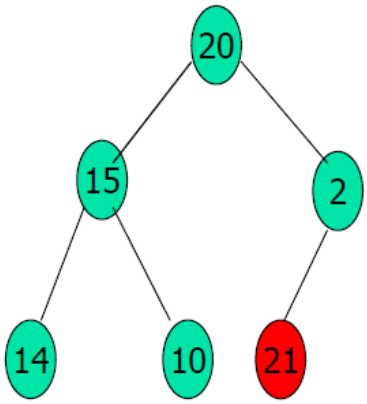
- Suppose the element to be inserted has value 5



Suppose the element to be inserted has value 21

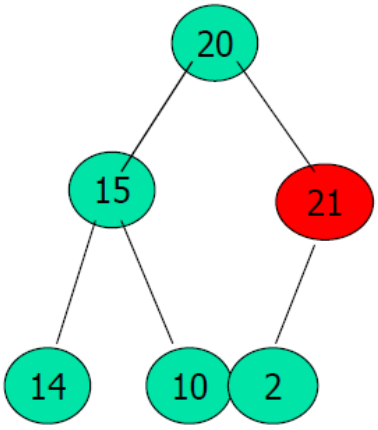
- The elements 2 and 5 must be swapped for maintaining the heap property

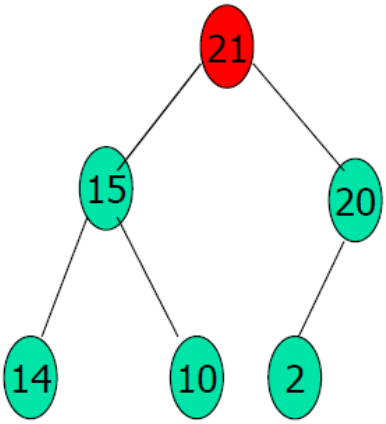




The new element 21 will find its position by continuous swapping with the existing elements for maintaining the heap property

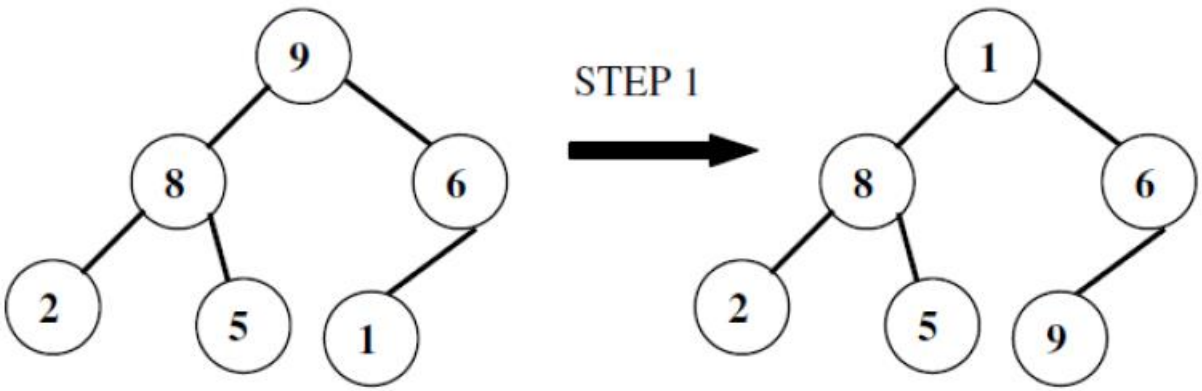
Finally the new element 21 goes to the top

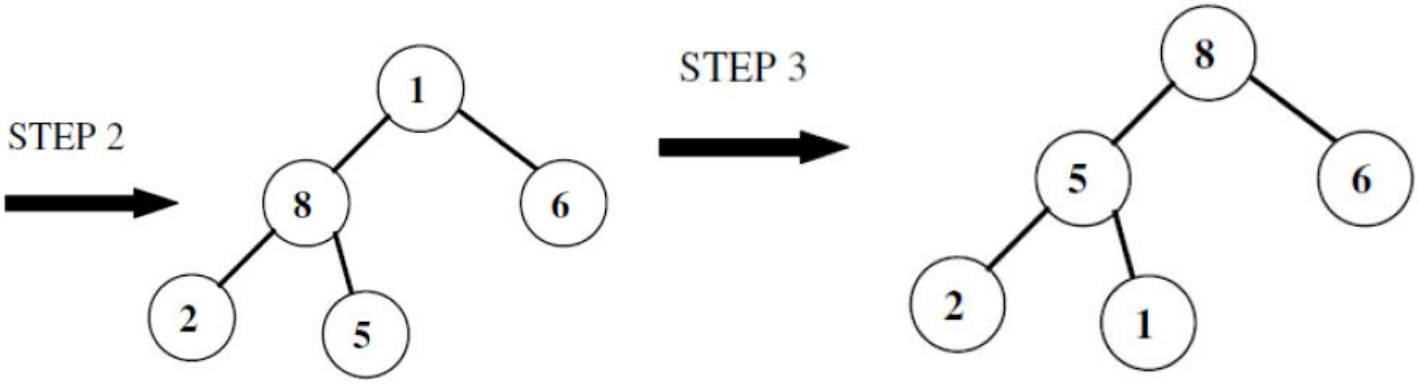




removeMax() from a MaxHeap

Step 1: Exchange the root's key with the last key K of the heap.
Step 2: Decrease the heap's size by 1
Step 3: "Heapify" the smaller tree.

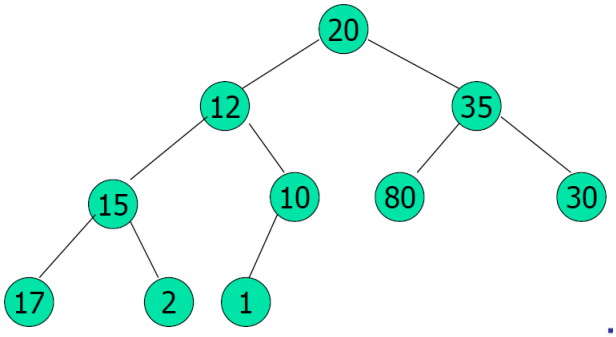




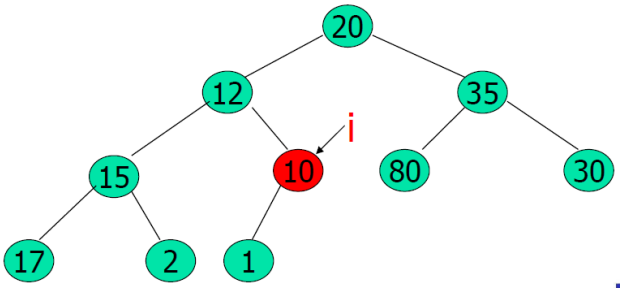
MaxHeap Initialization

- **Steps**
  - **Allocate the elements in an array**
  - **Form a complete binary tree**
  - **In the array, start with the rightmost node having a child**
    - **node number → n/2**
  - **Fix the heap in the node**
  - **Reverse back to the first node in the array**

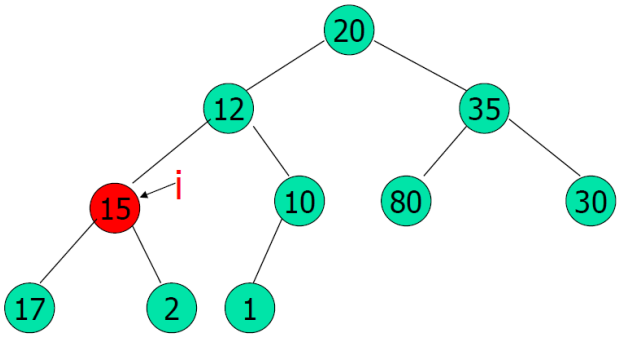
- Input array = [20, 12, 35, 15, 10, 80, 30, 17, 2, 1]
- Just make a complete binary tree



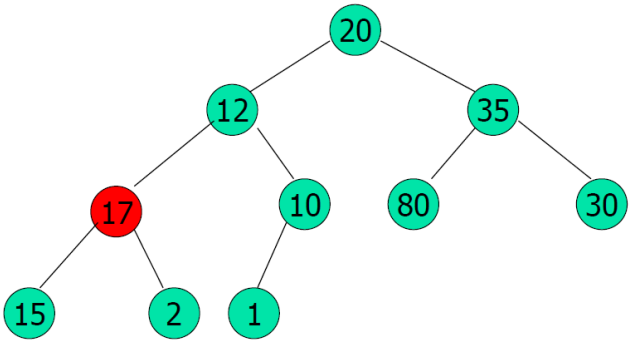
- Start at rightmost array position that has a child.
- Index i is $(n/2)^{th}$ of the array.



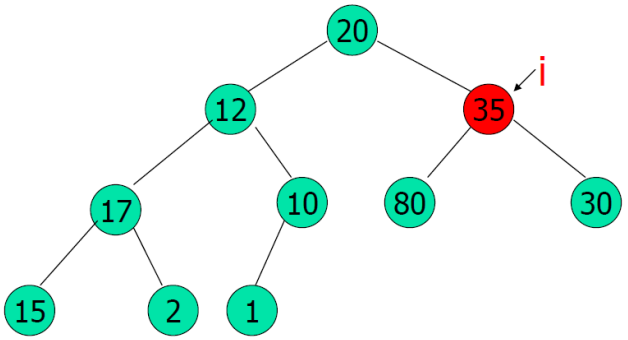
- Move to next lower array position.



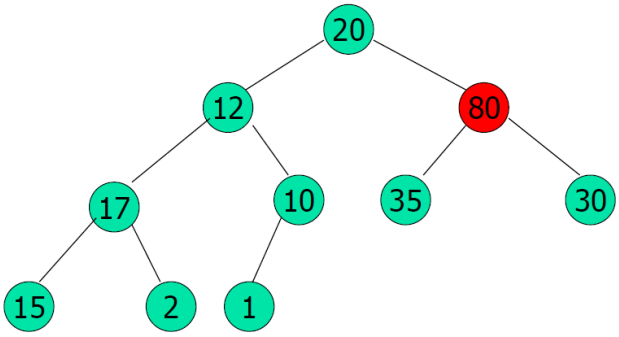
- Find a home for 15



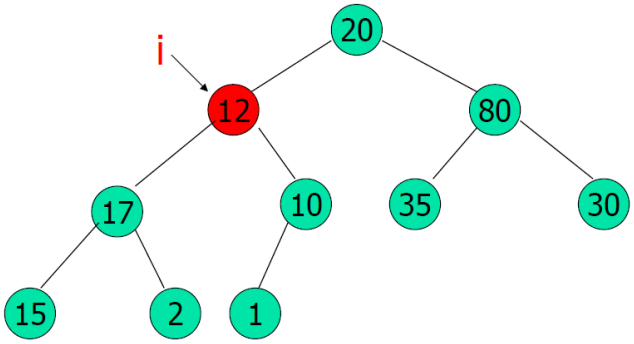
- Move to next lower array position.



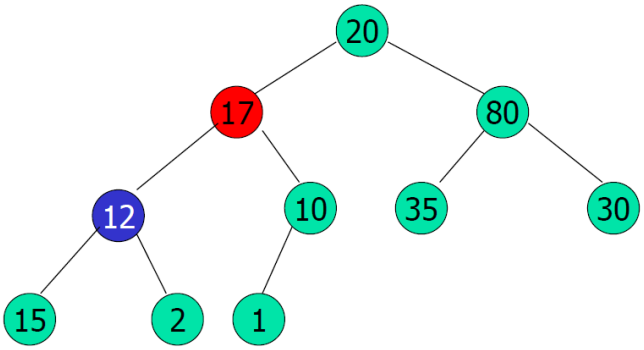
- Find a home for 35



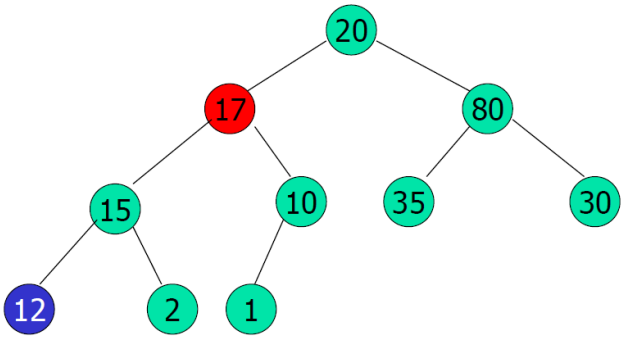
- Move to next lower array position.



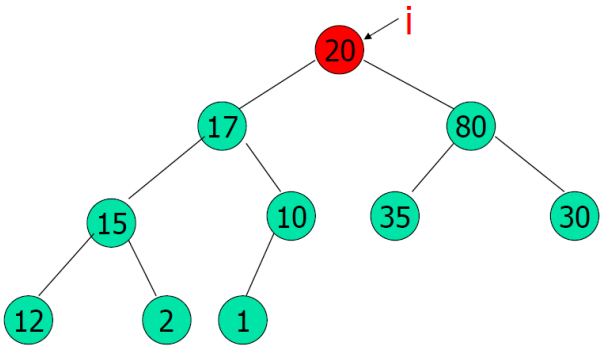
- Find a home for 12



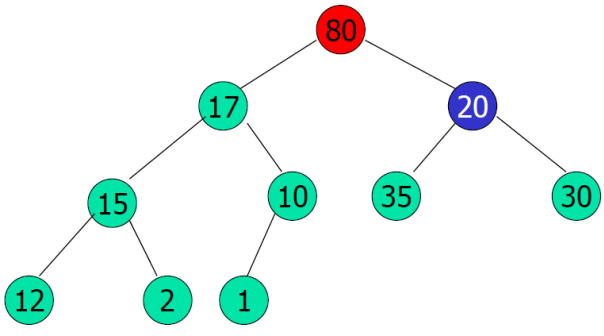
- Find a home for 12



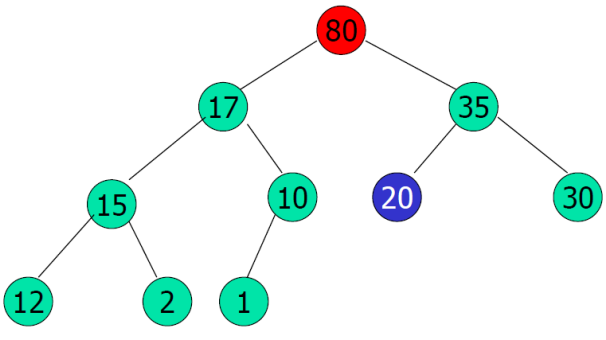
- Move to next lower array position.



- Find a home for 20

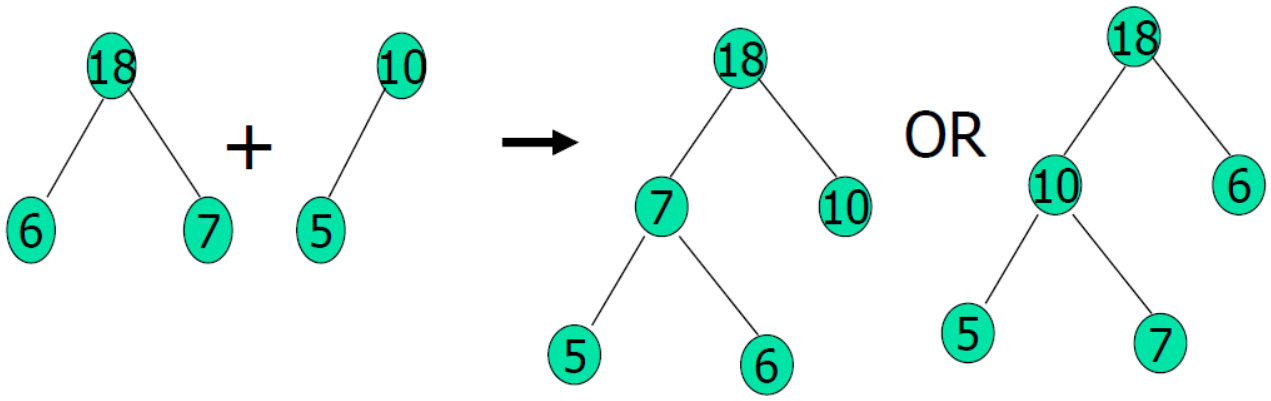


- Result the max heap



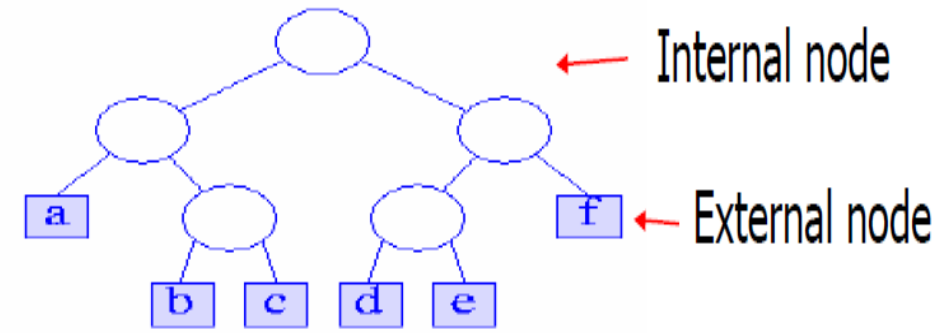
Leftist Trees for Priority Queue

- Heap is efficient for priority queue
- Some applications require merging two or more priority queues
- Heap is not suitable for merging two or more priority queues
- Leftiest tree is powerful in merging two or more priority queues

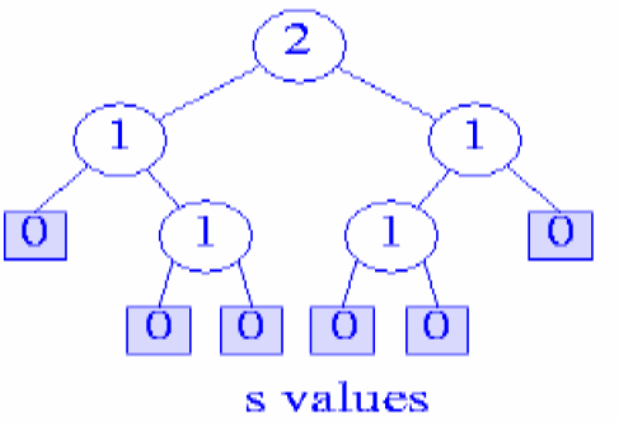


Height-Biased Leftist Tree (HBLT)

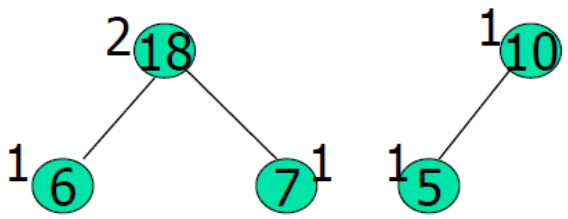
- Extended Binary Tree: Add an external node replaces each empty subtree.



- Let s(x) be the length of a shortest path from node x to an external node in its subtree.



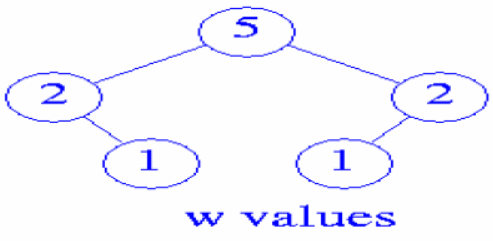
- A binary tree is a height-biased leftist tree (HBLT) iff at every internal node, the s value of the left child is greater than or equal to the s value of the right child.
  - A max HBLT is an HBLT that is also a max tree.
  - A min HBLT is an HBLT that is also a min tree.



- S values in HBLT contributes to make complete binary tree!!!!!!
- [Theorem] Let x be any internal node of an HBLT
  - The number of nodes in the subtree with root x is at least $2^{s(x)} - 1$
  - If the subtree with root x has m nodes, s(x) is at most $\log_2(m+1)$
  - The length of the right-most path from x to an external node is s(x)

Weight Biased Leftist Tree (WBLT)

- Let w(x) be the weight from node x to be the number of internal nodes in the subtree with root x



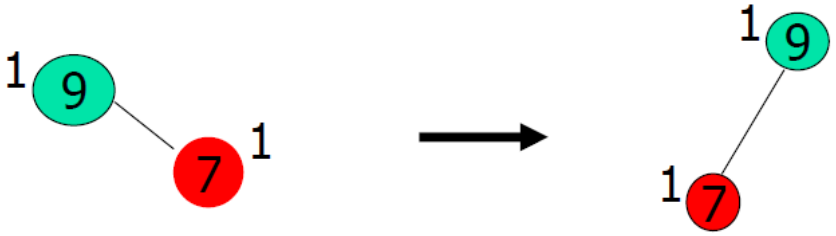
- A binary tree is weight-biased leftist tree (WBLT) iff at every internal node the w value of the left child is greater than or equal to the w value of the right child
  - A max WBLT is a max tree that is also a WBLT
  - A min WBLT is a min tree that is also a WBLT

Meld Two HBLTs

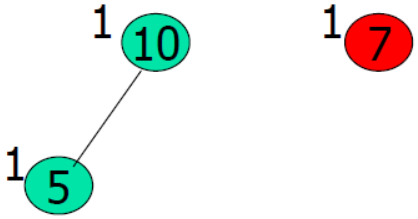
- Let A & B be the two HBLTs
- Compare the root of A & B
- The bigger value is the new root for the melded tree
  - Assume the root of A is bigger & A has left subtree L
- Meld the right subtree and B ➔ result C
- A has the left subtree L and the right subtree C
- Compare the S values of L & C
- Swap if necessary

- Consider the two max HBLTs

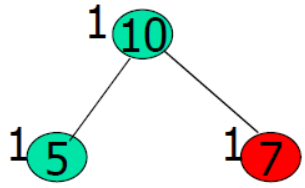
- 9 > 7, so 9 is root.
- The s value of the left subtree of 9 is 0 while the s value of the right subtree is 1 ➔ Swap the left subtree and the right subtree



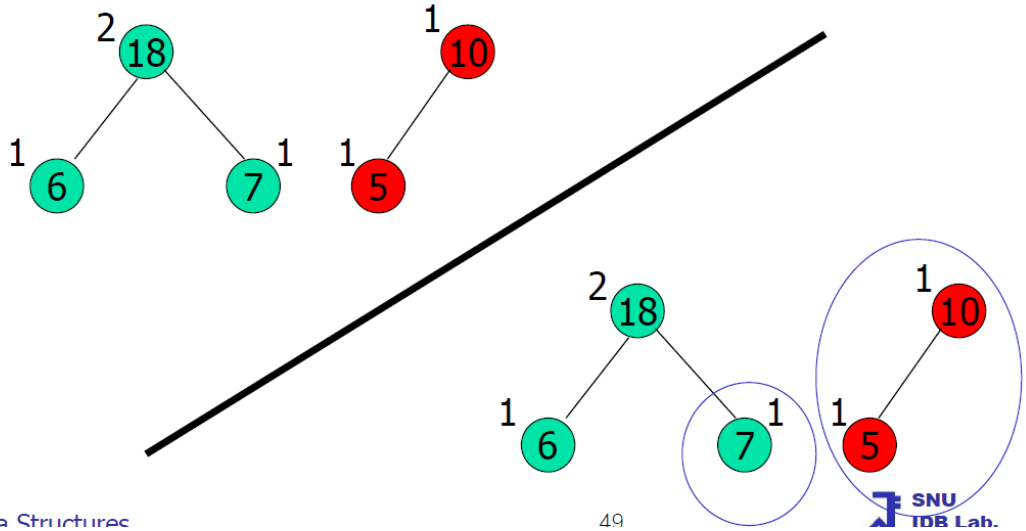
- Consider the two max HBLTs



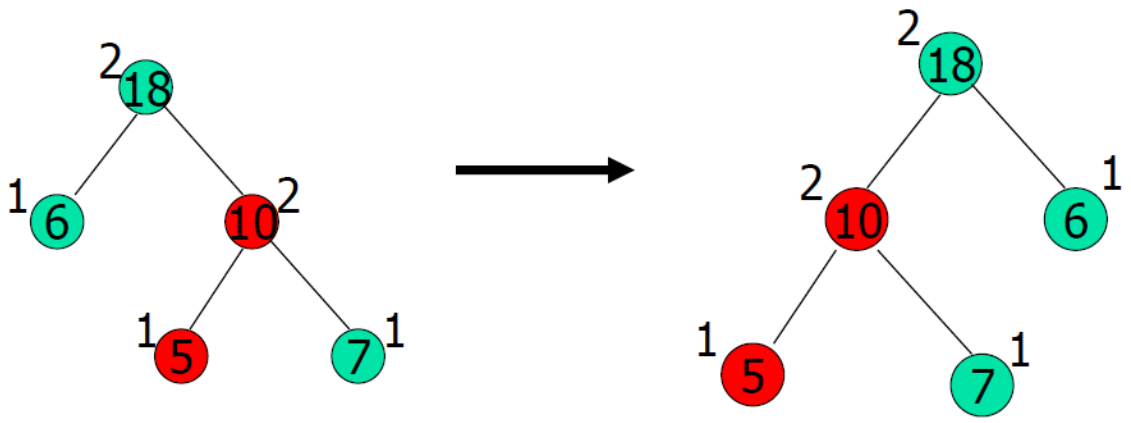
- 10 > 7, so root is 10



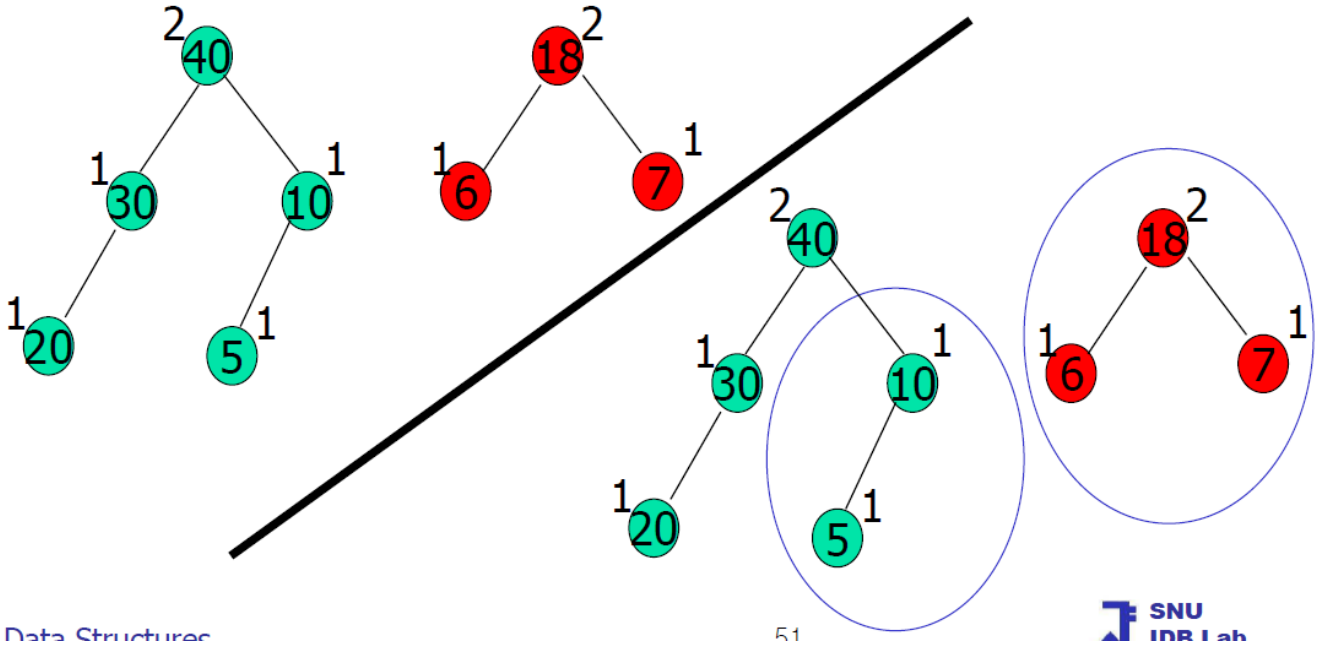
- Comparing the s values of the left and right children of 10, a swap is not necessary
- Consider the two max HBLTs



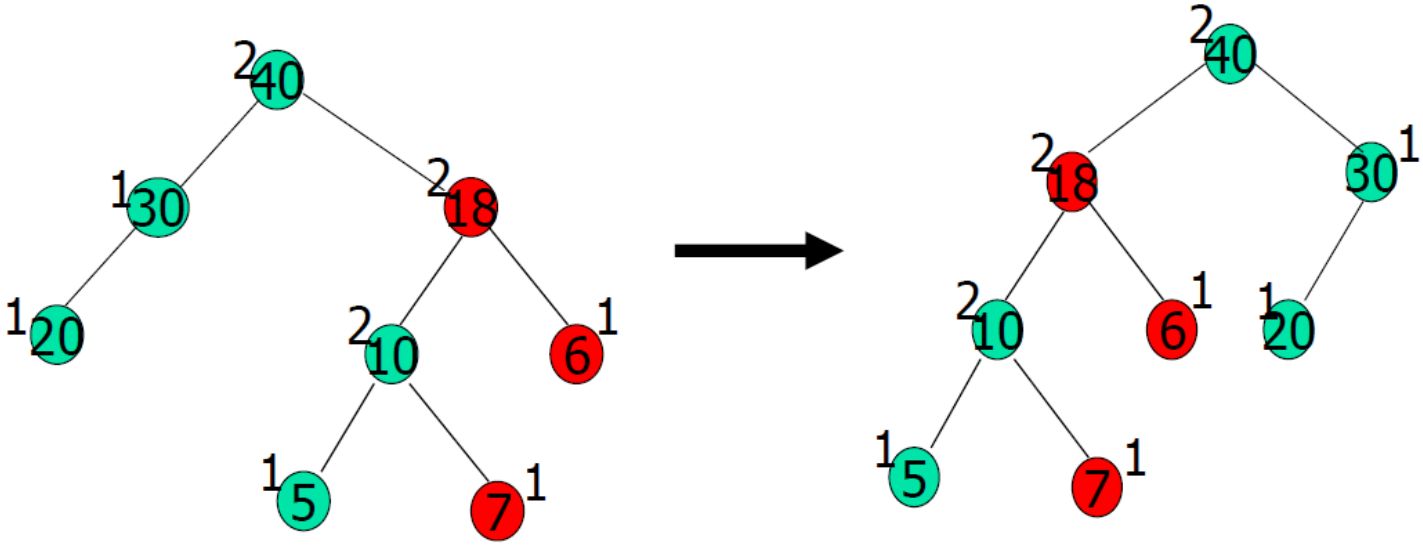
- 18 > 10, root is 18
- Meld the right subtree of 18
- s(left) < s(right), swap left and right subtree



- Consider the two max HBLTs



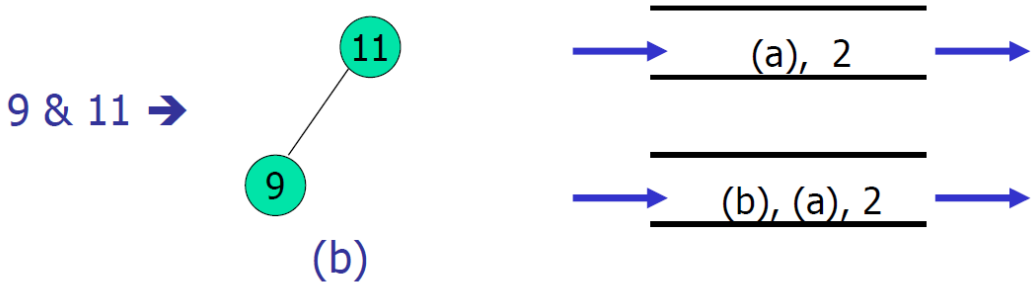
- 40 > 18, root is 40
- Meld the right subtree of 40
- s(left) < s(right), swap left and right subtree



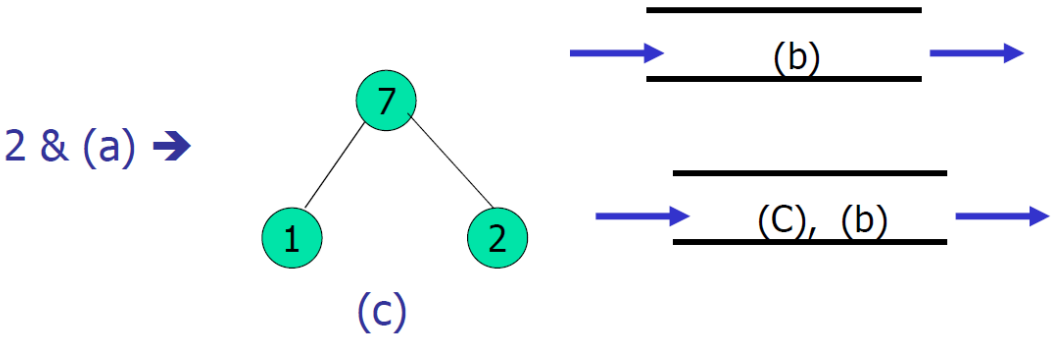
Initializing a Max HBLT

- Create a max HBLT with the five elements 7, 1, 9, 11, and 2
- Five single-element max HBLTs are created and placed in a FIFO queue
- The max HBLTs 7 and 1 are deleted from the queue and melded into (a)

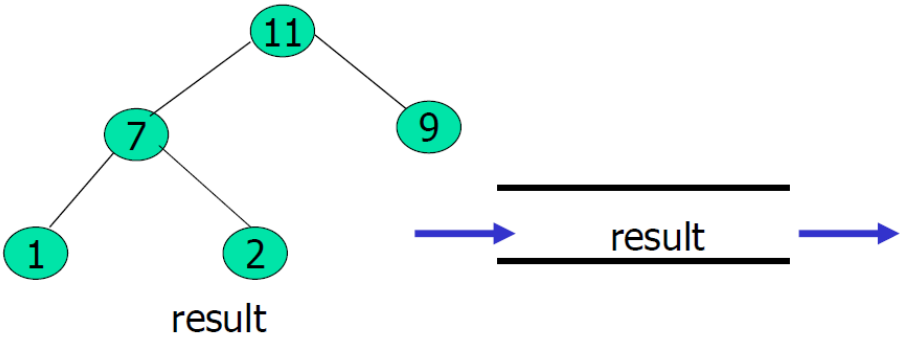
- The result (a) is added to the queue
- The max HBLTs 9 and 11 are deleted from the queue and melded into (b)



- The result (b) is added to the queue
- The max HBLTs 2 and (a) are deleted from the queue and melded into (c)



- The result (c) is added to the queue
- The max HBLTs (b) and (c) are deleted from the queue and melded into the result

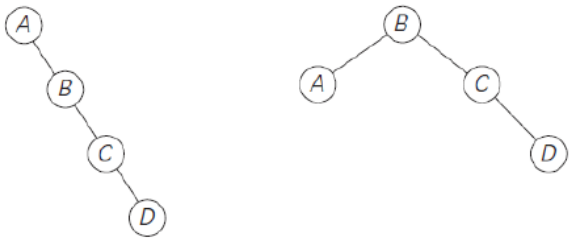


- The result is added to the queue
- The queue now has just one max HBLT, and we are done with the initialization

4. Optimal Binary Search Trees

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

As an example, consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. The figure depicts two out of 14 possible binary search trees containing these keys.



The average number of comparisons in a successful search in the first of these trees is 0.1 * 1+ 0.2 * 2 + 0.4 * 3+ 0.3 * 4 = 2.9, and for the second one it is 0.1 * 2 + 0.2 * 1+ 0.4 * 2 + 0.3 * 3= 2.1. Neither of these two trees is, in fact, optimal.

Following the classic dynamic programming approach, we will find values of C(i, j) for all smaller instances of the problem, although we are interested just in C(1, n). To derive a recurrence underlying a dynamic programming algorithm, we will consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree (Figure 8.8), the root contains key ak, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree T^j_{k+1}contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged. (Note how we are taking advantage of the principle of optimality here.)

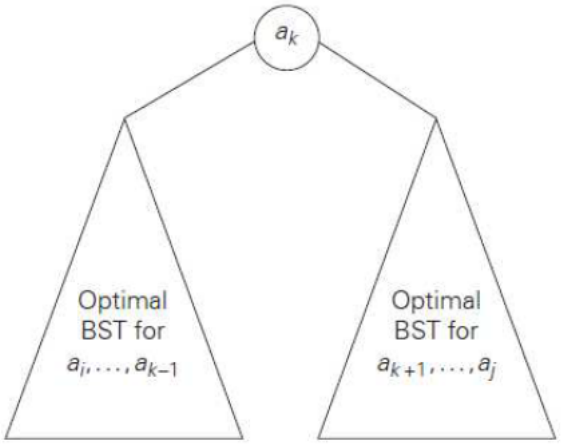


FIGURE 8.8 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j.

If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

Example: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Let us compute C(1, 2):

$$C(1,2) = \min \left\{ \begin{array}{ll} k=1: & C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: & C(1,1) + C(3,2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\}$$
$$= 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4. On finishing the computations we get the following final tables:

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since R(1, 4) = 3, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D. To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since R(1, 2) = 2, the root of the optimal tree containing A and B is B, with A being its left child (and the root of the one-node tree: R(1, 1) = 1). Since R(4, 4) = 4, the root of this one-node optimal tree is its only key D. Figure given below presents the optimal tree in its entirety.