

- Five calls to **Heapify** are made. After each of these calls, the tree looks like: 4, 11, 3, 5, 15, 23, 7, 9, 14, 8; 4, 11, 3, 14, 15, 23, 7, 9, 5, 8; 4, 11, 23, 14, 15, 3, 7, 9, 5, 8; 4, 15, 23, 14, 11, 3, 7, 9, 5, 8; and 23, 15, 7, 14, 11, 3, 4, 9, 5, 8.
- We can think of the hash function  $h$  as one that partitions the elements of  $U$  into  $m$  parts. If  $|U| > mn$ , by pigeon-hole principle, one of these parts will be of size  $\geq \frac{|U|}{m} > n$ . The function  $h$  hashes all the elements in this part to the same slot.
- Maintain an array  $a[]$  of  $n$  bits and a linked list  $L$  of all the elements in the set  $S$ . Bit  $a[i]$  is set to 1 if integer  $i$  exists in the set  $S$ , otherwise  $a[i] = 0$ .

**INSERT(i)** If  $a[i]$  is 0 then set  $a[i]$  to 1 and append the element  $i$  to  $L$ .

**DELETE** Delete first element of  $L$  and set the bit for this element in  $a[]$  to 0.

**MEMBER(i)** If  $a[i]$  is 1 then  $i$  is a member of the set else it is not a member.

- We can make use of any data structure that supports the following operations: 1) **SEARCH** for an arbitrary element; 2) **INSERT** an arbitrary element; and 3) **DELETE** the minimum. One such data structure is a 2-3 tree. We can use a modified version of any such data structure. Each node in this data structure will be a linked list of identical keys. Sorting, as pointed out in class, amounts to inserting the given keys into the empty data structure and deleting the minimum one by one. When inserting a key, we first check if this key is present in the data structure. If it is, the new key will be inserted as the head of the corresponding list. If it is not, a new node will be created (using the **INSERT** algorithm of the data structure) and a list with one key will be stored in the node.

Note that the data structure will contain at most  $d$  nodes at any time. Time for inserting a single element into the data structure is  $\log d + c$ , where  $c$  is a constant time needed to insert an element into linked list. Time for inserting all the elements into the data structure is  $O(n \log d)$ . Total time for all the deletes is  $O(n + d \log d)$ . Therefore, the run time of the algorithm is  $O(n \log d + n + d \log d) = O(n \log d)$ .

- Using Master theorem,  $T(n) = \Theta(n^{\log_4 12})$ .
  - We use repeated substitutions here.  $T(n) = T(\sqrt{n}) + \log n = T(n^{1/4}) + \frac{\log n}{2} + \log n$ .  
 $= T(n^{1/8}) + \frac{\log n}{4} + \frac{\log n}{2} + \log n$ . Proceeding in this manner we see that  $T(n) = \Theta(\log n)$ .
- Let  $U = A$  and  $I = \emptyset$ . Sort  $A$  in  $O(m \log m)$  time. For each element  $x$  in  $B$  do a binary search in  $A$  to see if  $x \in A$ . If  $x \in A$ , add  $x$  to the set  $I$  else add  $x$  to  $U$ . After this,  $U$  has the union and  $I$  has the intersection. Binary searches take a total of  $O(n \log m)$  time. Thus the run time of the entire algorithm is  $O(n \log m)$ .
- One way of solving this problem is to divide and conquer on  $\ell$ . Recursively merge the first  $\frac{\ell}{2}$  sequences to get  $S_1$ . Similarly recursively merge the last  $\frac{\ell}{2}$  sequences to get  $S_2$ . Finally, merge  $S_1$  and  $S_2$  to get the desired result. Note that  $S_1$  and  $S_2$  can be merged in  $O(n)$  time. Let  $T(\ell)$  be the run time needed to merge  $\ell$  sequences. We have:  $T(\ell) = T(\ell/2) + O(n)$  which solves to  $T(n) = O(n \log \ell)$ .