NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CX2001 Algorithms **2019/20 Semester 1**

Tutorial 3: Sorting

School of Computer Science and Engineering *Nanyang Technological University*

Week 6 (Q1-Q3):

Q1 The worst case for Insertion Sort occurs when the keys are initially in decreasing order. Suppose the performance of Insertion Sort is measured by the total number of comparisons between array elements (not counting the number of swaps). Show at least two other initial arrangements of keys that are also worst cases for Insertion Sort.

Q2 Use the divide and conquer approach to design an algorithm that finds both the largest and the smallest elements in an array of n integers. Show that your algorithm does at most roughly $1.5n$ comparisons of the elements. Assume $n = 2^k$.

Q3 Show how Merge Sort sorts each of the arrays below and give the number of comparisons among array elements in sorting each array.

(a) 14 40 31 28 3 15 17 51

(b) 23 23 23 23 23 23 23 23

Week 7 E-learning (Q4-Q6):

Q4 Suppose that, instead of using E[middle] as pivot, Quick Sort also can use the median of E[first], E[(first + last)/2] and E[last]. How many key comparisons will Quick Sort do in the worst case to sort n elements? (Remember to count the comparisons done in choosing the pivot.)

Q5 Each of n elements in an array may have one of the key values red, white, or blue. Give an efficient algorithm for rearranging the elements so that all the reds come before all the whites, and all the whites come before all the blues. (It may happen that there are no elements of one or two of the colours.) The only operations permitted on the elements are examination of a key to find out what colour it is, and a swap, or interchange, of two elements (specified by their indices). What is the asymptotic order of the worst case running time of your algorithm? (There is a linear-time solution.)

Q6 Suppose we have an unsorted array A of n elements and we want to know if the array contains any duplicate elements.

(a) Outline (clearly) an efficient method for solving this problem.

(b) What is the asymptotic order of the running time of your method in the worst case? Justify your answer.

(c) Suppose we know the n elements are integers from the range $1, \ldots, 2n$, so other operations besides comparing keys may be done. Give an algorithm for the same problem that is specialized to use this information. Tell the asymptotic order of the worst case running time for this solution. It should be of lower order than your solution for part (a).

Week 9 (Q7-Q9):

Q7 Given an array with content (of type date): 1 Jul, 30 Jan, 22 Mar, 22 Dec, 30 May, 21 Feb, 3 Nov, 7 Jun, 22 Feb, 21 Nov, 30 Dec; all of the same year. Suppose an earlier date is considered bigger than a later date; for example, "30 Jan" is bigger than "30 Dec". In order to sort these dates by Heap Sort, let us construct a maximizing heap. Show the content of the array after the heap construction phase.

Q8 An array of distinct keys in decreasing order is to be sorted (into increasing order) by Heap Sort.

(a) How many comparisons of keys are done in the heap construction phase (Algorithm constructHeap() in lecture notes on Sorting) if there are 10 elements?

(b) How many are done if there are n elements? Show how you derive your answer.

(c) Is an array in decreasing order the best case, the worst case, or an intermediate case for this algorithm? Justify your answer.

Q9 Given k lists with a total of n numbers, where $k \geq 2$ and each list has been sorted in decreasing order, design an algorithm to merge the k lists into one list sorted in decreasing order, in running time $\mathcal{O}(n \log_2 k)$.

Q2 Use the divide and conquer approach to design an algorithm that finds both the largest and the smallest elements in an array of n integers. Show that your algorithm does at most roughly $1.5n$ comparisons of the elements. Assume $n = 2^k$.

Binary search a cyclically sorted Array.

Similarity
- Recursive function
→ must have base case

Divide & Conquer (MergeSort, QuickSort)

```
solve(problem of size n)
{   if (n is small enough)
    {
        find the sol^n & return
    }
    divide the input array into 2 halves
    use prob dep^t strategy to determine
    which half we should go rec^e call
    solve(problem of size n/2)
}
```

$T_n = T_{n/2} + b$

$T_n : O(\log_2 n)$

VS

```
solve(problem of size n)
{   if (n is small enough)
    {
        find sol^n & return
    }
    divide & input array into 2 halves
    solve first half → S1 : solve(problem size n/2)
    solve 2nd half → S2 : solve(problem size n/2)
    combine S1 & S2 to form sol^n of
    original prob
}
```

$T_n = 2\,T_{n/2} + b$

Question 2

```
void findMinMax(int[] ar, int start, int end, int[] minMax)
// minMax is an array of two integers to return the minimum
// and maximum found
{
      int tempMinMax1[] = new int[2];
      int tempMinMax2[] = new int[2];
      int mid;

      if (start == end) { // one element array
              minMax[0] = ar[start];
              minMax[1] = ar[start];
      }
      else if (end - start == 1) { // two element array
              if (ar[start] > ar[end]) {
                      minMax[1] = ar[start];
                      minMax[0] = ar[end];
              } else {
                      minMax[0] = ar[start];
                      minMax[1] = ar[end];
              }
      }
```

Base cases

Base case

```
        else {   // array longer than two elements
                 mid = (start + end)/2;

                 findMinMax(ar, start, mid, tempMinMax1);
                 findMinMax(ar, mid+1, end, tempMinMax2);

                 if (tempMinMax1[0] < tempMinMax2[0])
                         minMax[0] = tempMinMax1[0];
                 else    minMax[0] = tempMinMax2[0];

                 if (tempMinMax1[1] > tempMinMax2[1])
                         minMax[1] = tempMinMax1[1];
                 else    minMax[1] = tempMinMax2[1];
        }
```

part that divides.

compare min max 1st 1/2 , , 2nd 1/2

combine s1 & s2 to come up in final Ans.

T_n = # of comparisons for input array of size n

Step ①. $T_1 = 0$, $T_2 = 1$

Step ② $T_n = \boxed{2} T_{n/2} + \boxed{2}$

$n = 2^k$

$T_{2^k} = 2T_{2^{k-1}} + 2$

$\times 2$: $2T_{2^{k-1}} = 2^2 T_{2^{k-2}} + 2^2$

$\times 2^2$: $2^2 T_{2^{k-2}} = 2^3 T_{2^{k-3}} + 2^3$

$\times 2^{k-1}$: $2^{k-2} T_{2^2} = 2^{k-1} T_{2^1} + 2^{k-1}$

k-1 eqns.

sum of Geometric seq.

$T_{2^k} = 2^{k-1} + (2^1 + 2^2 + \dots 2^{k-1})$

$= \frac{n}{2} + \frac{2^1 - 2}{2-1}$

$= \frac{n}{2} + \frac{2^k - 2}{2-1}$

$= \frac{n}{2} + n - 2$

$= O(n)$

How to solve?

difficult: characteristic func

$T_k = \square T_{k-1} + T_{k-2} + \square$

easy:

$T_k = \square T_{k-1} + \square$

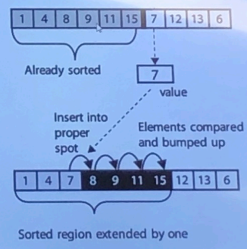
if not ☺ need to further split it up.

Keep reducing index until reach base case

Question 1

- The worst case for Insertion Sort occurs when the keys are initially in decreasing order. Suppose the performance of Insertion Sort is measured by the total number of comparisons between array elements (not counting the number of swaps). Show at least two other initial arrangements of keys that are also worst cases for Insertion Sort.

Best	Average	Worst
O(n)	$O(n^2)$	$O(n^2)$



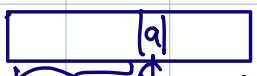
Question 1

- **Ans**: Two other worst-case input for Insertion Sort are:

$(n, n-1, n-2, \ldots, 3, 1, 2)$
$(1, n, n-1, n-2, \ldots, 2)$

Note: In each of above lists, the *i*th key still requires ***i*** **comparisons** to find its correct position, although a comparison may not be followed by a **swap** of keys.

Insertion sort: there are n-1 iterations

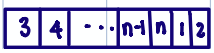
iteration i 

already sorted elements

(i+1) element

To reach the worst, we must have i comparisons in the ith iteration.

after (n-3) iterations



compare '1' with all the n-2 elements

sorted.

Need (n-2) swaps.

after (n-2) iterations sorted

1	3	4	..	n-1	n	2

sorted.

compare '2' with n-1 elements
Need n-2 swaps.

e.g (1, n, n-1, ..., 3, 2)

Iteration 1 | 1 | n | - — — | 1 comparison, no swaps

sorted

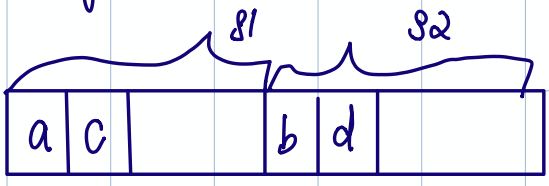
Iteration 2 | 1 | n | n-1 | . . . . | 2 comparisons, 1 swap

sorted

Iteration n-1 | 1 | 3 | - . . | n-1 | n | 2 | n-1 comparisons, n-2 swaps.

sorted.

merge sort



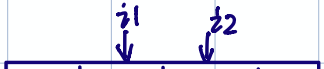
case1: a<b, ↑ i1

case 2: a>b shift, swap & ↑ i2

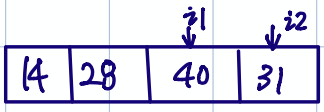
case 3: a=b shift, swap & ↑ i1 & i2

i1 i2

e.g

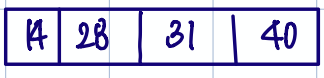


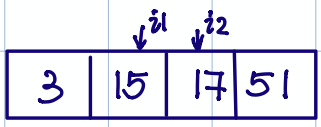
14<28 : case 1



40>28 case 2

40>31 case 2





3<17 case 1

15<17 case 1

CE2001/CZ2001 Algorithms

Tutorial 3 (Sorting)

Fall 2020

Question 1

- The worst case for Insertion Sort occurs when the keys are initially in decreasing order. Suppose the performance of Insertion Sort is measured by the total number of comparisons between array elements (not counting the number of swaps). Show at least two other initial arrangements of keys that are also worst cases for Insertion Sort.
- **Ans**: Two other worst-case input for Insertion sort are:

$(n, n-1, n-2, \ldots, 3, 1, 2)$
$(1, n, n-1, n-2, \ldots, 2)$

Note: In each of above lists, the ***i***th key still requires ***i* comparisons** to find its correct position, although a comparison may not be followed by a **swap** of keys.

Question 2

- Use the divide and conquer approach to design an algorithm that finds both the largest and the smallest elements in an array of n integers. Show that your algorithm does at most roughly $1.5n$ comparisons of the elements. Assume $n = 2^k$.
- Ans:

```
void findMinMax(int[] ar, int start, int end, int[] minMax)
// minMax is an array of two integers to return the minimum
// and maximum found
{
     int tempMinMax1[] = new int[2];
     int tempMinMax2[] = new int[2];
     int mid;

     if (start == end) { // one element array
             minMax[0] = ar[start];
             minMax[1] = ar[start];
     }
```

```
        else if (end - start == 1) { // two element array
                if (ar[start] > ar[end]) {
                        minMax[1] = ar[start];
                        minMax[0] = ar[end];
                } else {
                        minMax[0] = ar[start];
                        minMax[1] = ar[end];
                }
        }
        else {   // array longer than two elements
                mid = (start + end)/2;

                findMinMax(ar, start, mid, tempMinMax1);
                findMinMax(ar, mid+1, end, tempMinMax2);

                if (tempMinMax1[0] < tempMinMax2[0])
                        minMax[0] = tempMinMax1[0];
                else    minMax[0] = tempMinMax2[0];

                if (tempMinMax1[1] > tempMinMax2[1])
                        minMax[1] = tempMinMax1[1];
                else    minMax[1] = tempMinMax2[1];
        }
}
```

Question 2

- To show that the above algorithm does at most roughly $1.5n$ comparisons of the elements (assume $n = 2^k$):
- Let W_i be the number of comparisons for i elements, then the recurrence equation is:

$$W_1 = 0$$
$$W_2 = 1$$
$$W_n = W_{n/2} + W_{n-n/2} + 2$$

- If $n = 2^k$, where k is a positive integer,

$$W_n = 2W_{n/2} + 2$$

$$
\begin{aligned}
W_n &= 2W_{n/2} + 2 \\
&= 2\left(2W_{n/2^2} + 2\right) + 2 \\
&= 2^2 W_{n/2^2} + 2^2 + 2 \\
&= 2^2\left(2W_{n/2^3} + 2\right) + 2^2 + 2 \\
&= 2^3 W_{n/2^3} + 2^3 + 2^2 + 2 \\
&\cdots \\
&= 2^{k-1} W_{n/2^{k-1}} + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 \\
&= 2^{k-1} + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 \\
&= 2^{k-1} + 2(2^{k-2} + 2^{k-3} + \cdots + 2 + 1) \\
&= 2^{k-1} + 2(2^{k-1} - 1) \\
&= 2^{k-1} + 2^k - 2 \\
&= n/2 + n - 2 = \frac{3}{2}n - 2
\end{aligned}
$$

Since $n = 2^k$
So $n/2^{k-1} = 2$
$W_2 = 1$

Geometric series

Question 3

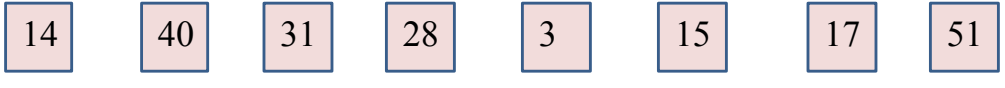
- Show how MergeSort sorts each of the arrays below and give the number of comparisons among array elements in sorting each array.

  (1) 14 40 31 28 3 15 17 51

  Ans: The array is first divided into two equal parts

Each part is then sorted by MergeSort. The process begins by dividing each part into equal parts

and then each of these parts into equal parts



This subdivision process now ends because each part contains only one item.

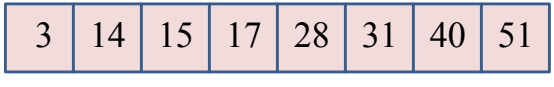
Each pair is then merged

4 comparisons

Each of these pairs is then merged

5 comparisons

Finally these pairs are merged



7 comparisons

to obtain the sorted array. Totally, it takes 4 + 5 + 7 = 16 comparisons.

Question 3

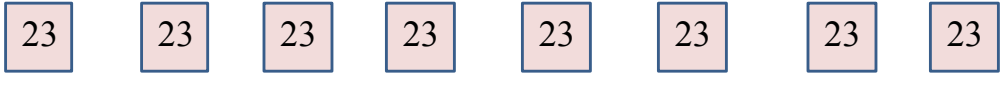
- Show how MergeSort sorts each of the arrays below and give the number of comparisons among array elements in sorting each array.

  (2) 23 23 23 23 23 23 23 23

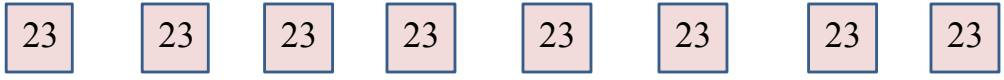
  (2) **Ans**: The array is first divided into two equal parts

Each part is then sorted by mergesort. The process begins by dividing each part into equal parts

and then each of these parts into equal parts



This subdivision process now ends because each part contains only one item.



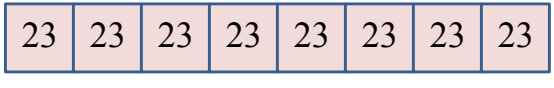
Each pair is then merged

4 comparisons

Each of these pairs is then merged

4 comparisons

Finally these pairs are merged



4 comparisons

to obtain the sorted array. Totally, it takes 4 + 4 + 4 = 12 comparisons.

Question 4

- Suppose that, instead of using E[middle] as pivot, QuickSort also can use the median of E[first], E[(first + last)/2] and E[last]. How many key comparisons will QuickSort do in the worst case to sort *n* elements? (Remember to count the comparisons done in choosing the pivot.)
- **Ans**:
- Consider a subrange of *k* elements to be partitioned.
- Choosing the pivot requires 3 key comparisons.
- Now $k - 3$ elements remain to be compared to the pivot.
  - It's easy to arrange not to compare elements that were candidates for the pivot again by swapping them to the extremes of the subrange
- So *k* comparisons are done by the time partition is finished.
- This is one more than if E[middle] is simply chosen as the pivot.

- An example of the worst case might be E[first] = 100, E[first + last/2] = 99, E[last] = 98, and all other elements are over 100.
- Then, 99 becomes the pivot, 98 is the only element less than the pivot, and k – 2 elements are greater than the pivot.
- The remaining keys can be chosen so that the worst case repeats – that is, we can arrange that the median-of-3 always comes out to be the second-smallest key in the range.
- Since the larger subrange is reduced by only 2 at each depth of recursion in QuickSort, it will require $n/2$ calls to partition(), with ranges of size n, $n - 2$, $n - 4$, …, 2 (suppose n is even), then the total number of comparisons is

$$f = n + (n-2) + (n-4) + \cdots + 2$$

$$f = 2 + 4 + \cdots + (n-2) + n$$

$$\therefore 2f = (n+2) + (n+2) + \cdots + (n+2) = \frac{n(n+2)}{2}$$

$n/2$ terms

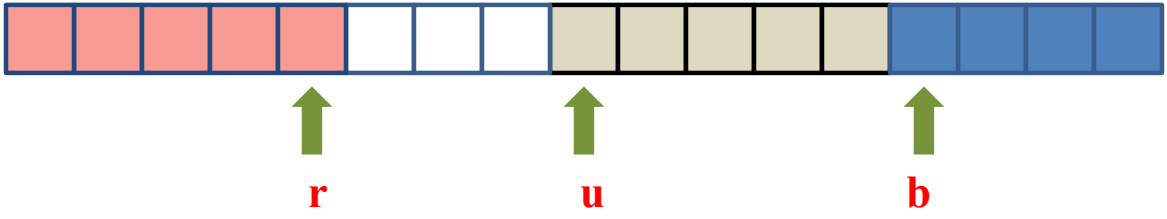
$$\therefore f = \frac{n^2 + 2n}{4} = \Theta(n^2)$$

- When n is odd, the total number of comparisons is also about $n^2/4$.
- So the worst-case time complexity with the median-of-3 strategy is $\Theta(n^2)$.

Question 5

- Each of n elements in an array may have one of the key values *red*, *white*, or *blue*.
- Give an efficient algorithm for rearranging the elements so that all the *reds* come before all the *whites*, and all the *whites* come before all the *blues*. (It may happen that there are no elements of one or two of the colours.)
- The only operations permitted on the elements are examination of a key to find out what colour it is, and a swap, or interchange, of two elements (specified by their indices).
- What is the asymptotic order of the worst case running time of your algorithm? (There is a linear-time solution.)

- We assume the elements are stored in the range 1, …, *n*
- At an intermediate step the elements are divided into four regions:
  - The first contains only reds
  - The second contains only whites
  - The third contains elements of colors unknown yet
  - The fourth contains only blues
- There are three indexes: r (last red), u (first unknown), b (first blue)



- The algorithm is:

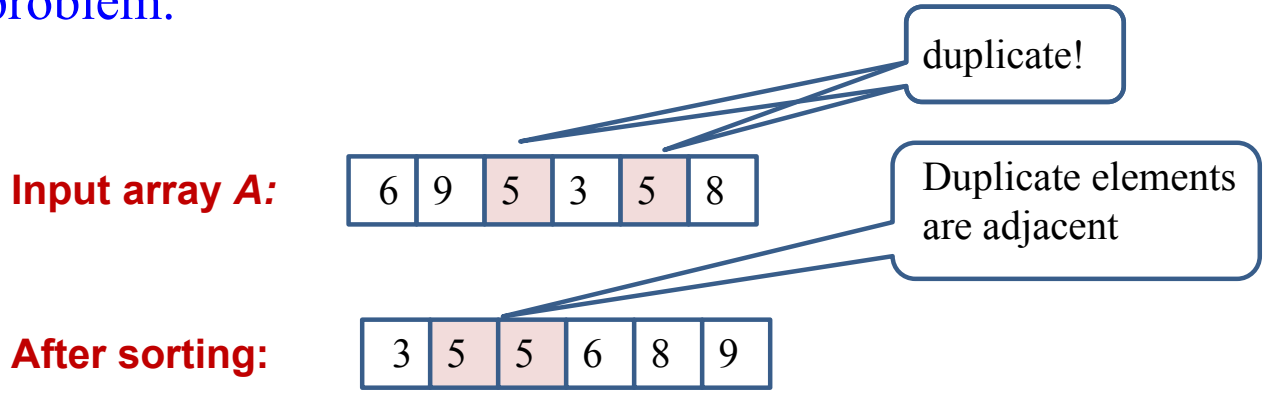
```
int r; // index of last red
int u; // index of first unknown
int b; // index of first blue

r = 0; u = 1; b = n+1;
while (u < b)
        if (E[u] == red)
                Interchange E[r+1] and E[u];
                r ++;
                u ++;
        else if (E[u] == white)
                u ++;
        else // (E[u] == blue)
                Interchange E[b-1] and E[u]
                b --;
```

- With each iteration of the while loop, either u is increased by one or b is decreased by one, so there are n iterations.
- Each iteration takes constant time, so the overall time is linear $\Theta(n)$.

Question 6

- Suppose we have an unsorted array A of n elements and we want to know if the array contains any duplicate elements.

(a) Outline (clearly) an efficient method for solving this problem.



- **Ans**: First, sort the array in increasing (or decreasing) order. Then, do a post-scan of the sorted array to check for equal adjacent elements.

Question 6

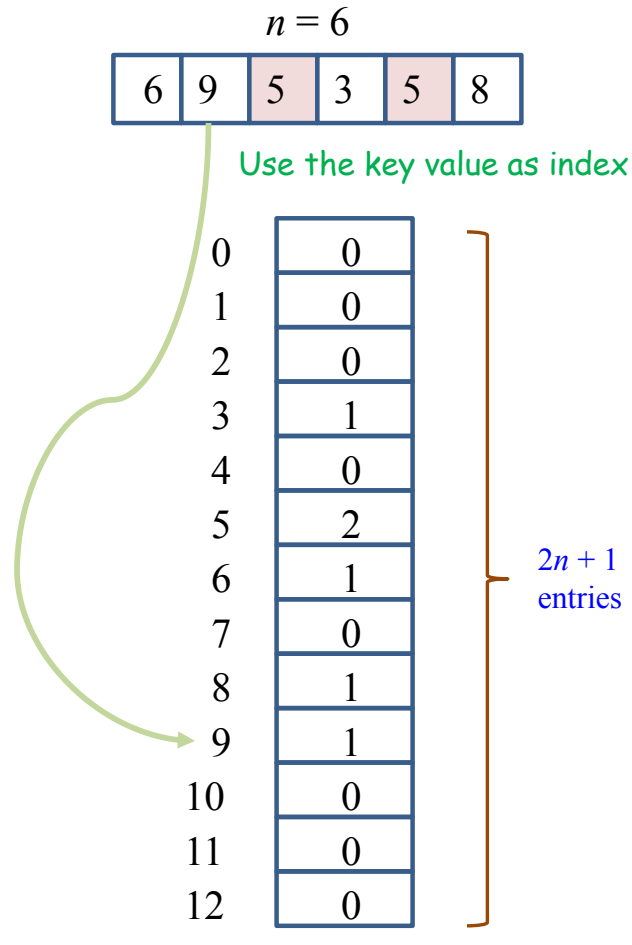
```
Bool ExistDuplicate(int A[], int n) {
     MergeSort(A, n); // or Heapsort
     for (int i = 1; i < n; i++) {
            if (A[i] == A[i-1])
                   return true;
     }
     return false;
}
```

- (b) What is the asymptotic order of the running time of your method in the worst case? Justify your answer.
- **Ans**: The worst-case time for sorting is $\Theta(n\lg n)$ using Mergesort (or Heapsort). The time for post-scan is $\Theta(n)$. Therefore, the overall worst-case running time of the algorithm is $\Theta(n\lg n)$.

Question 6

(c) Suppose we know the n elements are integers from the range 1, …, $2n$, so other operations besides comparing keys may be done. Give an algorithm for the same problem that is specialized to use this information. Tell the asymptotic order of the worst case running time for this solution. It should be of lower order than your solution for part (a).

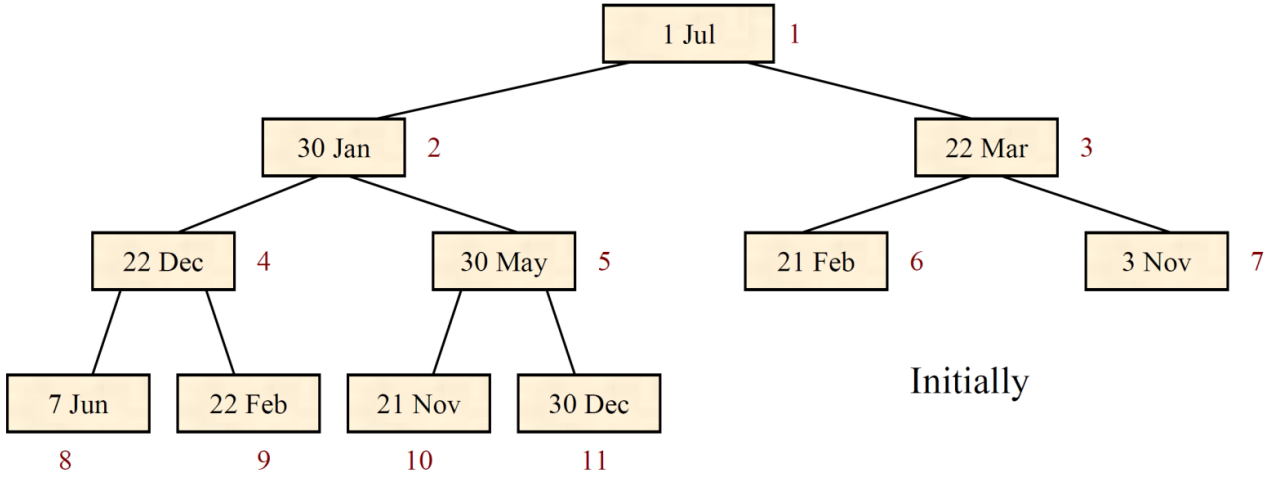
- **Ans**: Simply count how many have each key value, using an array of $2n + 1$ entries to store the counts.
- We can quit as soon as any count becomes 2.
- Since we scan the n-elements array once, and each element takes constant time to count, the overall worst-case running time is $\Theta(n)$.



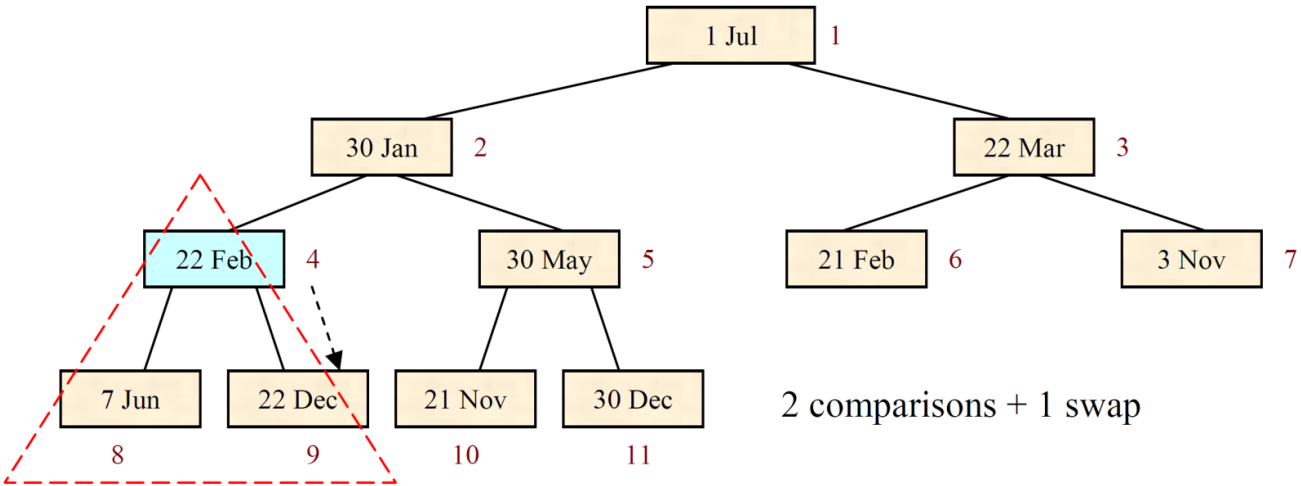
Question 7

- Given an array with content (of type date): 1 Jul, 30 Jan, 22 Mar, 22 Dec, 30 May, 21 Feb, 3 Nov, 7 Jun, 22 Feb, 21 Nov, 30 Dec; all of the same year.
- Suppose an earlier date is considered bigger than a later date; for example, “30 Jan” is bigger than “30 Dec”.
- In order to sort these dates by Heapsort, let us construct a maximising heap. Show the content of the array after the heap construction phase.

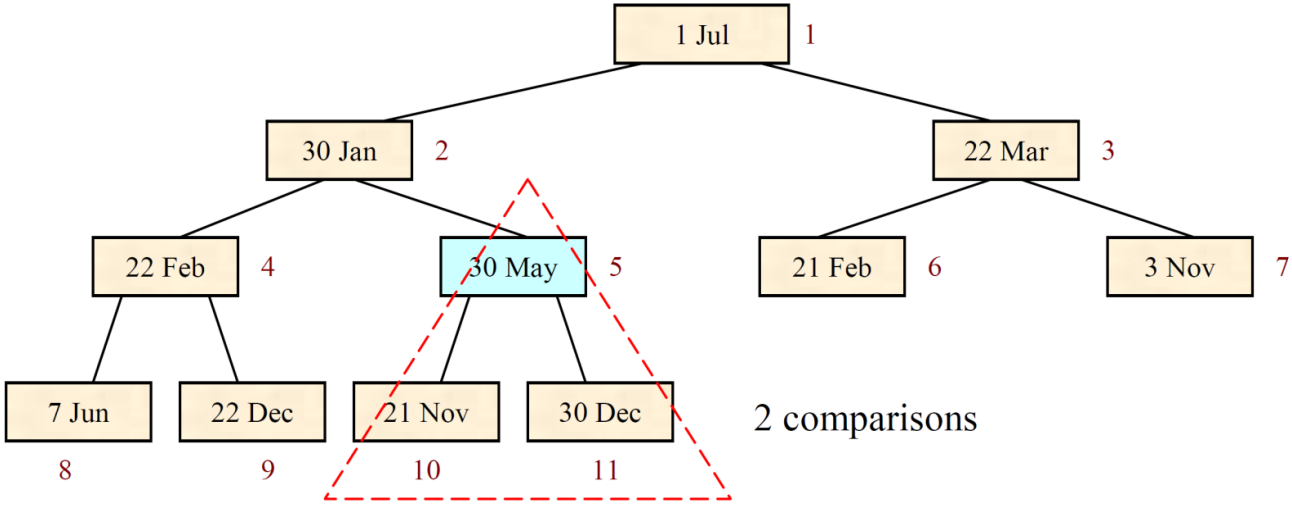
1	2	3	4	5	6	7	8	9	10	11
1 Jul	30 Jan	22 Mar	22 Dec	30 May	21 Feb	3 Nov	7 Jun	22 Feb	21 Nov	30 Dec



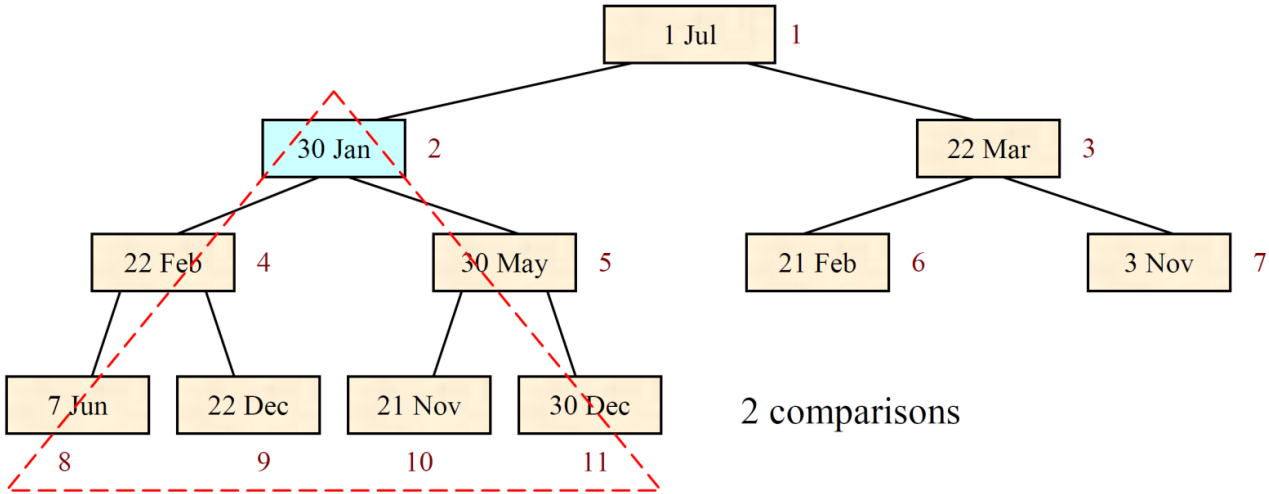
1	2	3	4	5	6	7	8	9	10	11
1 Jul	30 Jan	22 Mar	22 Dec	30 May	21 Feb	3 Nov	7 Jun	22 Feb	21 Nov	30 Dec



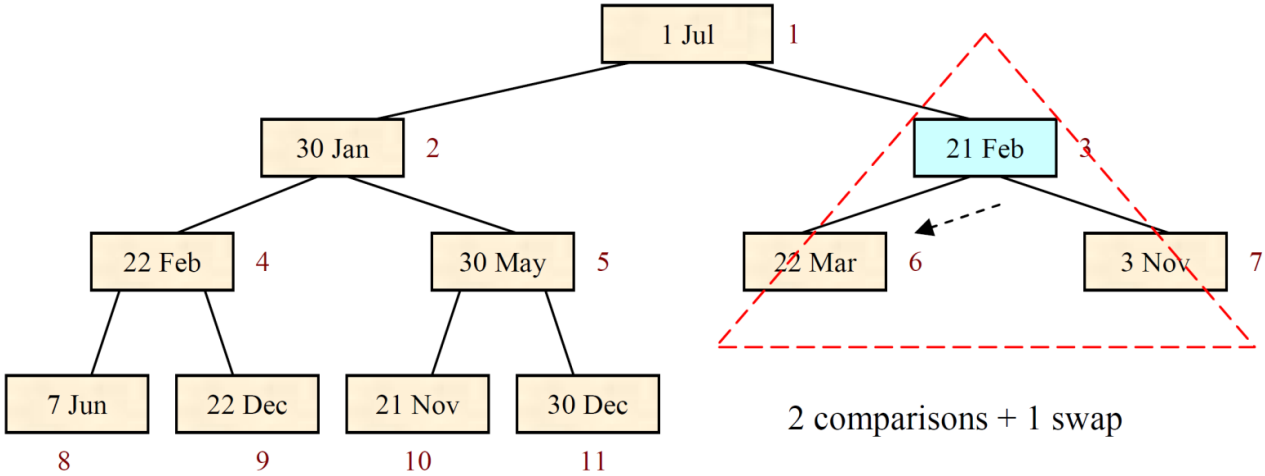
1	2	3	4	5	6	7	8	9	10	11
1 Jul	30 Jan	22 Mar	22 Feb	30 May	21 Feb	3 Nov	7 Jun	22 Dec	21 Nov	30 Dec



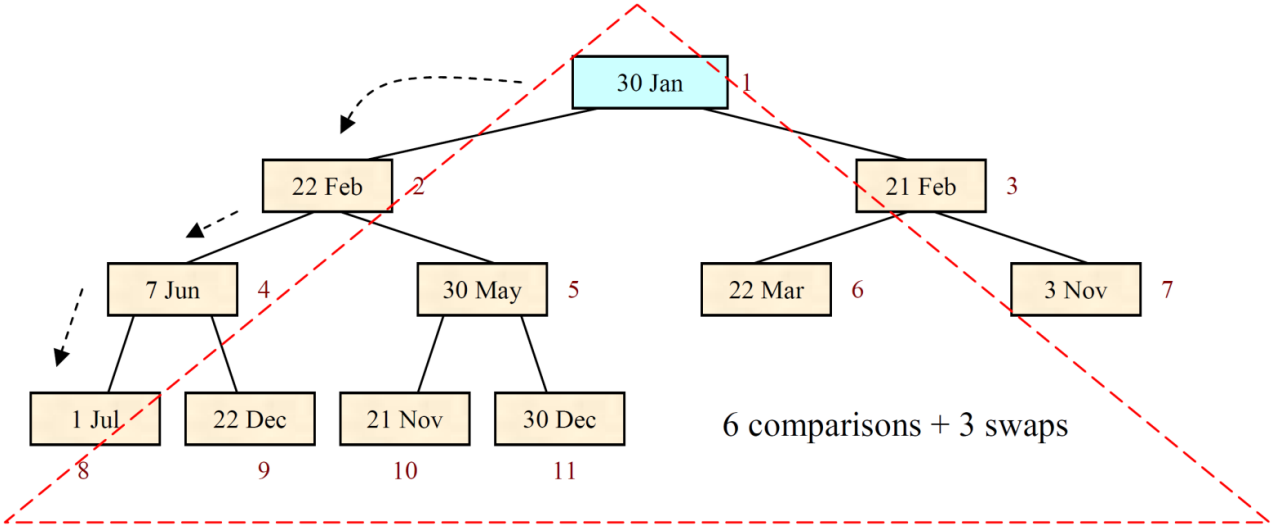
1	2	3	4	5	6	7	8	9	10	11
1 Jul	30 Jan	22 Mar	22 Feb	30 May	21 Feb	3 Nov	7 Jun	22 Dec	21 Nov	30 Dec



1	2	3	4	5	6	7	8	9	10	11
1 Jul	30 Jan	22 Mar	22 Feb	30 May	21 Feb	3 Nov	7 Jun	22 Dec	21 Nov	30 Dec



1	2	3	4	5	6	7	8	9	10	11
1 Jul	30 Jan	21 Feb	22 Feb	30 May	22 Mar	3 Nov	7 Jun	22 Dec	21 Nov	30 Dec

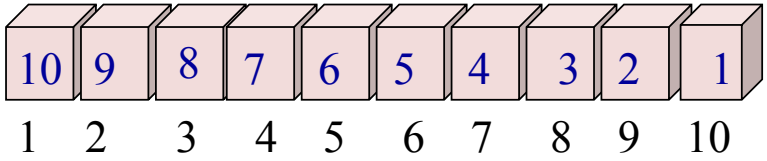


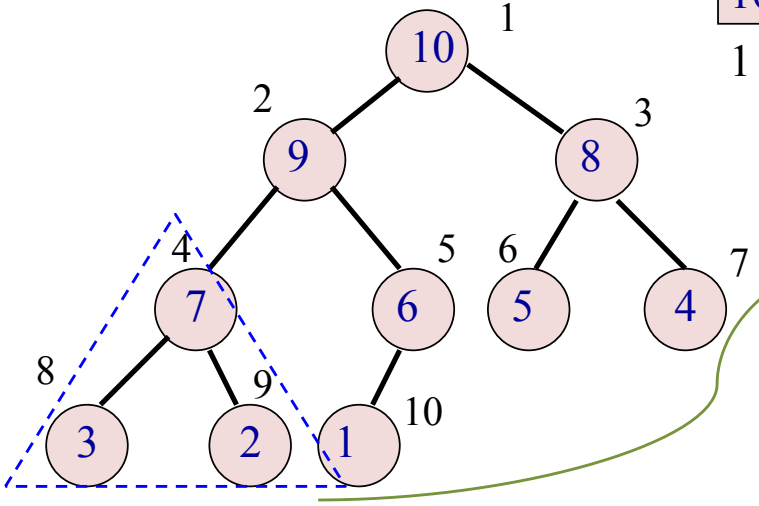
- **Ans**: Heapify recursively performs **14** date comparisons and **5** swaps in the heap construction phase, and the content in the resulting array becomes:

1	2	3	4	5	6	7	8	9	10	11
30 Jan	22 Feb	21 Feb	7 Jun	30 May	22 Mar	3 Nov	1 Jul	22 Dec	21 Nov	30 Dec

Question 8

- An array of distinct keys in decreasing order is to be sorted (into increasing order) by HeapSort.
- (a) How many comparisons of keys are done in the heap construction phase (Algorithm constructHeap() of lecture notes) if there are 10 elements?
- **Ans**: 2 + 1 + 2 + 2 + 2 = 9.





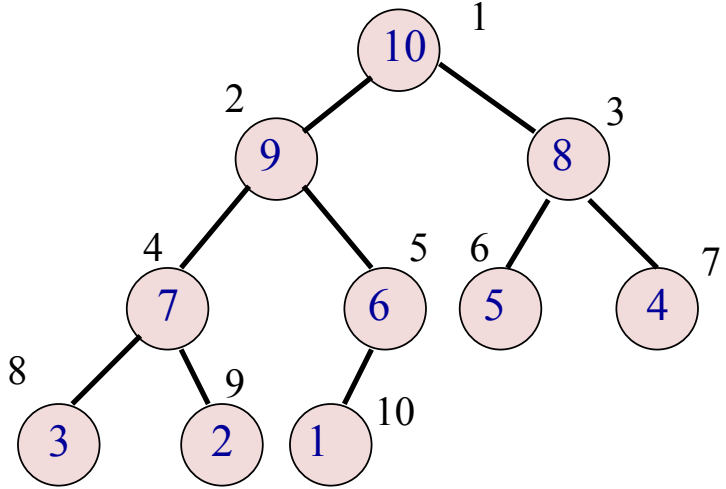
ID of nodes to be compared	No. comparisons
4, 8, 9	2
5, 10	1
2, 4, 5	2
3, 6, 7	2
1, 2, 3	2

Question 8

- (b) How many are done if there are n elements? Show how you derive your answer.
- **Ans**: Node of index i is an *internal* node (that is, not a *leaf*) if and only if $1 \le 2i \le n$, so $½ \le i \le n/2$. Thus,
  - If n is even, there are $n/2$ internal nodes
  - If n is odd, there are $(n - 1)/2$ internal nodes
- Since the keys are in decreasing order, fixHeap does only one or two key comparisons each time it is called; no keys move
- If n is **even**, the last internal node has only one child, so
  - fixHeap does only one key comparison at that node
  - For each other internal node, two key comparisons are done
  - Total number of key comparisons is $2 \times (n/2 - 1) + 1 = n - 1$
- If n is **odd**, each of the $(n - 1)/2$ internal nodes has two children, so
  - Two key comparisons are done by fixHeap for each internal node
  - Total number of key comparisons is $2 \times (n - 1)/2 = n - 1$
- In both cases, the total is $n - 1$ key comparisons

Question 8

- (c) Is an array in decreasing order a best case, worst case, or intermediate case for this algorithm (of heap construction)? Justify your answer.
- **Ans**: It is the best case.
- Because there is no key move, so in the heapifying() algorithm, the fixHeap() function only compares the root with its two children, and no need to call fixHeap() recursively as in intermediate or worst case.

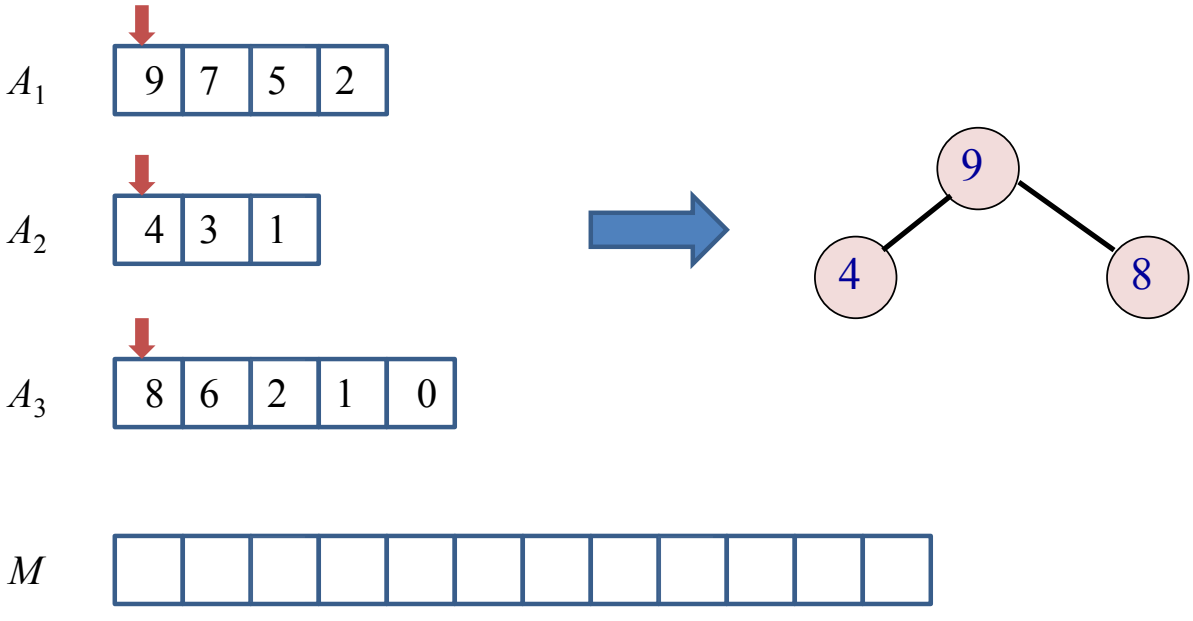


Question 9

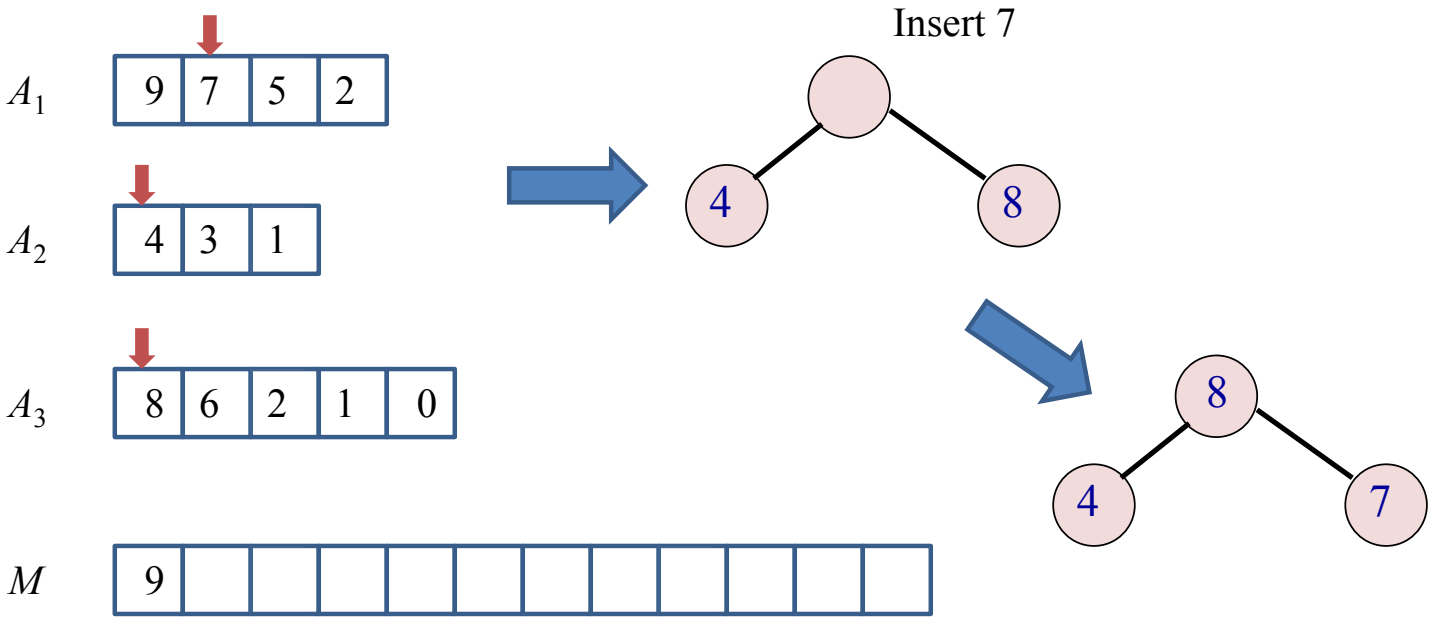
- Given k lists with a total of n numbers, where $k \geq 2$ and each list has been sorted in decreasing order, design an algorithm to merge the k lists into one list sorted in decreasing order, in running time $O(n\log_2 k)$.
- **Ans**: One algorithm is to use a maximizing heap:
  - First, insert the ***k*** elements from the heads of the ***k*** sorted lists into a maximizing heap. This will take running time $O(k)$.
  - Then, use getMax() to obtain the maximum element from the heap and append it at the end the merged list (initially empty). Suppose this merged element was originally from the jth list. Then, insert the element from the *next* position of the jth list into the heap using fixHeap.
  - Continue in this fashion until all the k lists have been exhausted.

  Analysis: For one element, an insertion using fixHeap takes time $O(\log_2 k)$, hence for the n elements the running time is $O(n\log_2 k)$.

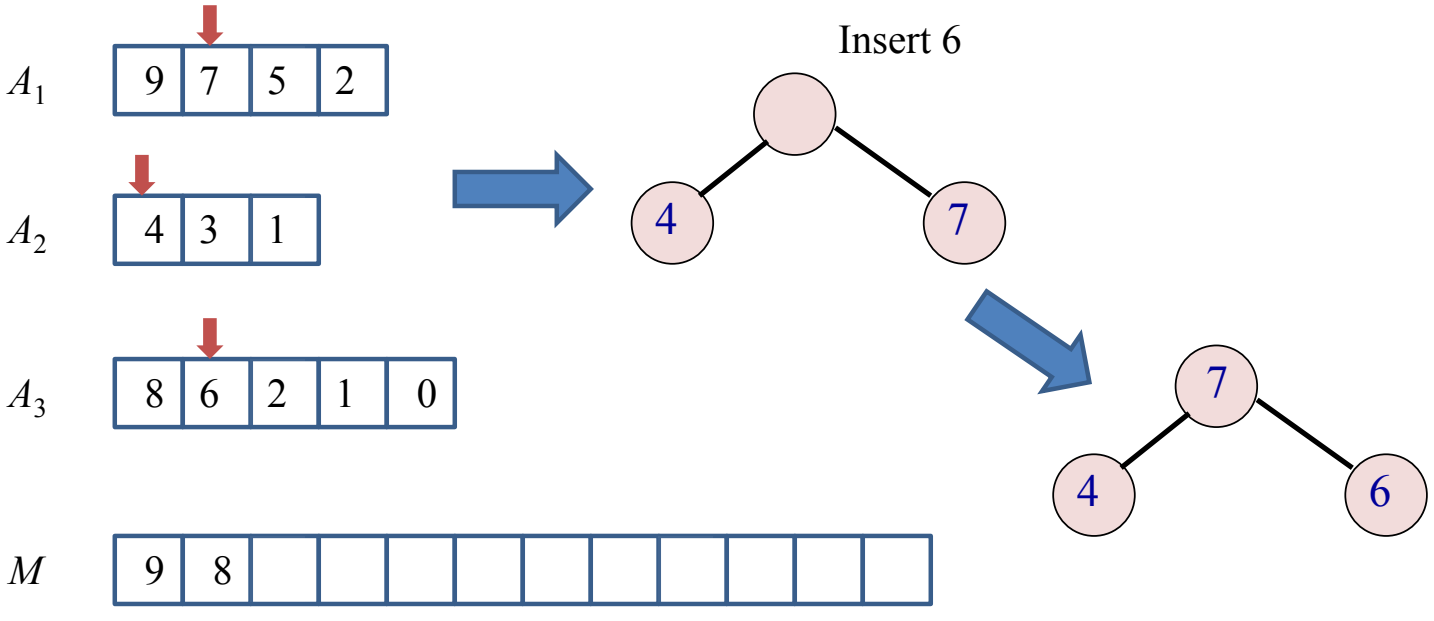
- Example:
  - Input: 3 sorted arrays
  - Output: A merged list M



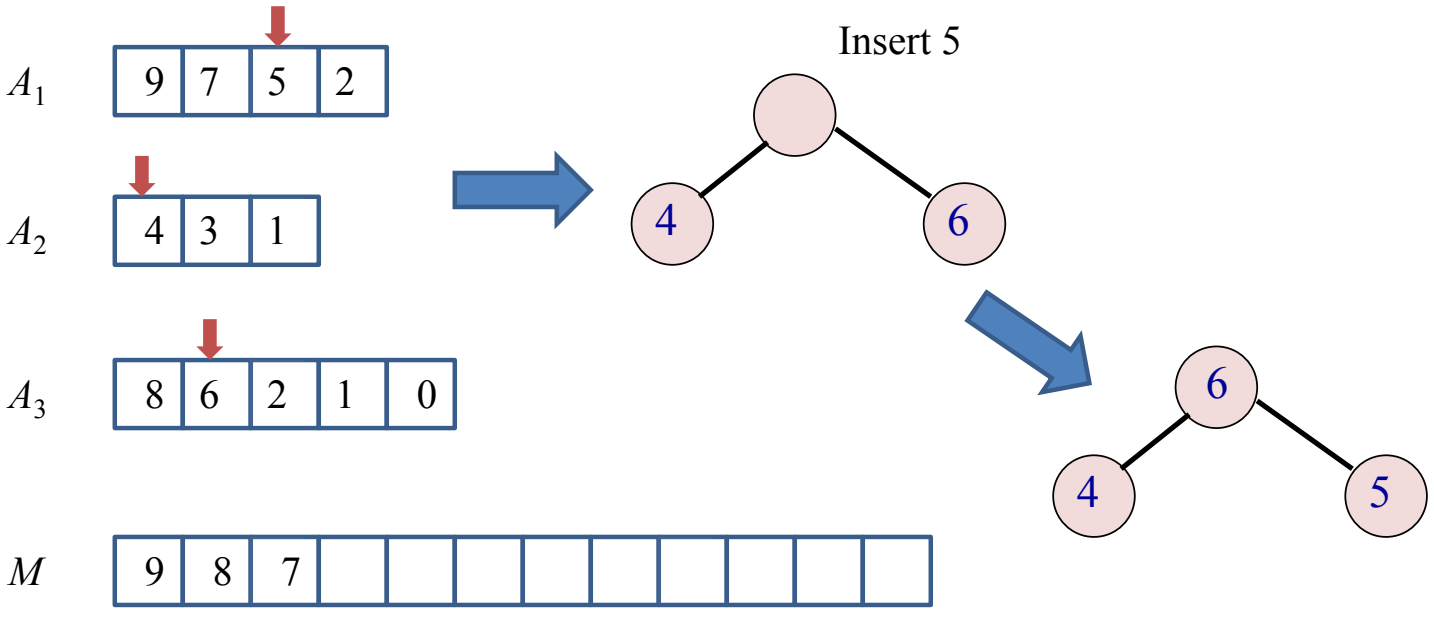
- Use getMax() to obtain the maximum element from the heap and append it at the end the merged list. Suppose this merged element was originally from the *j*th list. Then, insert the element at the *next* position of the *j*th list into the heap.



- Use getMax() to obtain the maximum element from the heap and append it at the end the merged list. Suppose this merged element was originally from the *j*th list. Then, insert the element at the *next* position of the *j*th list into the heap.



- Use getMax() to obtain the maximum element from the heap and append it at the end the merged list. Suppose this merged element was originally from the *j*th list. Then, insert the element at the *next* position of the *j*th list into the heap.



Continue in this fashion until all the input lists are exhausted…