UMPSA

اونيۏرسيتي مليسيا ڤهڠ السلطان عبد الله

UNIVERSITI MALAYSIA PAHANG AL-SULTAN ABDULLAH

ASSIGNMENT 1

COURSE	**: ALGORITHM AND COMPLEXITY**
COURSE CODE	**: BCI2313**
COURSE COORDINATOR	**: MOHD AZWAN BIN MOHAMAD**
SUBMISSION DATE	**: 25 APRIL 2025**
SESSION/SEMESTER	**: SESSION 2024/2025 SEMESTER II**

NAME	MATRIC ID	SECTION
NUR AMIRA SOFEA BINTI OTHMAN	CB22040	04A)

Table Of Contents

University Malaysia Pahang al-Sultan Abdullah (UMPSA) is organizing a mega interuniversity online quiz competition. As students submit answers, their scores are updated in real-time on a public leaderboard. The leaderboard must always display the top performers sorted in descending order of their scores. Due to the real-time nature of score updates, the system needs an efficient sorting mechanism. One of the most significant challenges faced by organizer is optimizing search and sorting algorithms to enhance real-time scoreboard. The technical team plans to create a real-time scoreboard which can updates the scoreboard after each score change and performed the searching algorithm. 2 The insertion sort can adequately use especially when the data is nearly sorted data resulting in $O(n\ 2\ )$ for worst case and $O(n)$ for best case where n is the number of elements in the dataset. Hence, the technical team focuses on the role of heap sort algorithms in improving real-time scoreboard performance. The Heap Sort algorithm is a highly efficient sorting algorithm with O (n log n) time complexity especially for random data. By analysing heap sort algorithms, it aims to determine the most efficient method for managing scoreboard before searching process takes place. In addition, with searching algorithm such as binary search, it will help accelerate the process. Heap sort algorithm with binary search resulting in O (n log n) complexity. It will ultimately enhanced search speed, reducing load times, and improving customer satisfaction. It aims to analyse its performance, examine its implementation, and evaluate its suitability for real-world, data-intensive applications. Produce a report which will include: 1. Problem Understanding: Elaborate on the definition of the insertion sort and heap sort algorithms to get the algorithm complexity.

QUESTION 1

Problem Understanding: Elaborate on the definition of the insertion sort and heap sort algorithms to get the algorithm complexity.

- ❖ Insertion Sort

  Insertion sort is a straightforward sorting algorithm that creates the final sorted array one element at a time. It does this by taking an element and inserting it into its correct position within the sorted portion of the data. This cycle is completed until no elements are unsorted. This algorithm is quite efficient when used with nearly sorted data.

The process of insertion sort :

1. The second element is the key
2. Shift larger values right until they fit in the right position
3. Insert the key into the appropriate position in the sorted
4. Repeat for all elements

Time Complexity

- ❖ Best Case (Nearly Sorted Data): O(n)
- ❖ Average Case : O(n2)
- ❖ Worst Case (Reverse-Sorted Data): O(n2)

Example : For the array [8 3 5 1 4 2], insertion sort steps through the data sequentially and pushes smaller elements to the left, leading the sorted order in O(n2) steps.

Insertion sort is also efficient when there are very few elements that need to be moved, as for very few updates, which is likely to happen in scoreboards, as only a few scores will need to be updated. However, its worst case is with quadratic time complexity, and thus it becomes inefficient where there are several updates or the score varies randomly.

- ❖ Heap Sort

  Heap Sort is an in-place sorting algorithm that uses a heap data structure to organize data. First, it builds a max heap, so that the max value is at the root(the top most node). Then, it successively removes the maximum item, maintain the heap, and sorts until the while set of data is ordered.

  Time Complexity

  - ❖ Worst Case Complexity : O(nlogn)
  - ❖ Best Case Complexity : O(nlogn)
  - ❖ Average Case Complexity : O(nlogn)

For heap sort, it is uniform in all cases as the structure of the heap allows the sorting to be done in deterministic manner. Moreover, it is not like insertion sort. Heap sort is not much improveable with nearly sorted data. Therefore, it performs consistently well for large datasets as it is the prime choice for real-time applications.

Example, we have list of unsorted values [14, 11, 2, 20, 3, 10, 3] . The max heap is [20, 11, 14, 2, 10, 5, 3], then extract the root which is 20 and rebuilds the heap until sorted.

QUESTION 2

Heap Sort Algorithm and Binary Search: Analyse how the process of the Heap Sort algorithm for assisting the binary search algorithm, including the two main steps of the heap sort algorithm.

Heap sort is a comparison-based algorithm as it quickly finds the maximum or minimum value by leveraging a binary heap and doesn’t require additional memory. The binary tree must be complete, where every level is filled except possibly the last. The binary tree must be in the form of a max heap first, where each parent node is always greater than or equal to its children. This ensures that the tree always contains the largest element to be moved to the end of the array and represented in an array.
To using binary search, the data must be in sorted order. Real-time scoreboard is quite dynamic and since the quiz score are updated constantly, they may get unsorted. The Heap Sort technique is essential to sort the data in ascending or descending order so that at every occasion, the information is prepared for Binary queries.

How does the heap sort support Binary Search?

Binary search needs the data to be sorted because it divides the data set in half to find a target value. Sorting is a must for binary search to work, otherwise, it will not give you the correct result.

Heap sort guarantees that the dataset will be fully sorted irrespective of the original order of data. It would allow us to run binary search effectively on O(log n) time, on subsequent searches after the initial sorting.

- ❖ Heap sort complexity : O(nlogn)
- ❖ Binary Search complexity : O(nlogn)
- ❖ Efficiency Combined : O(nlogn) for sorting and searching

In real time scoreboard, this combination allows for fast updates O(log n) with heap sort and binary search to improve the performance and responsiveness.

<u>Steps In Heap Sort</u>

- ➢ Build a Max Heap
  - First we must build the max heap where every parent in the binary tree has greater value than its children as shown in the figure below
  - This step gives us a heap from the unsorted array
  - Time complexity O(n)

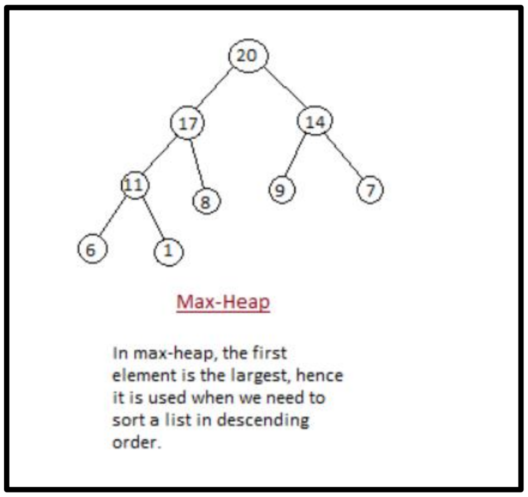
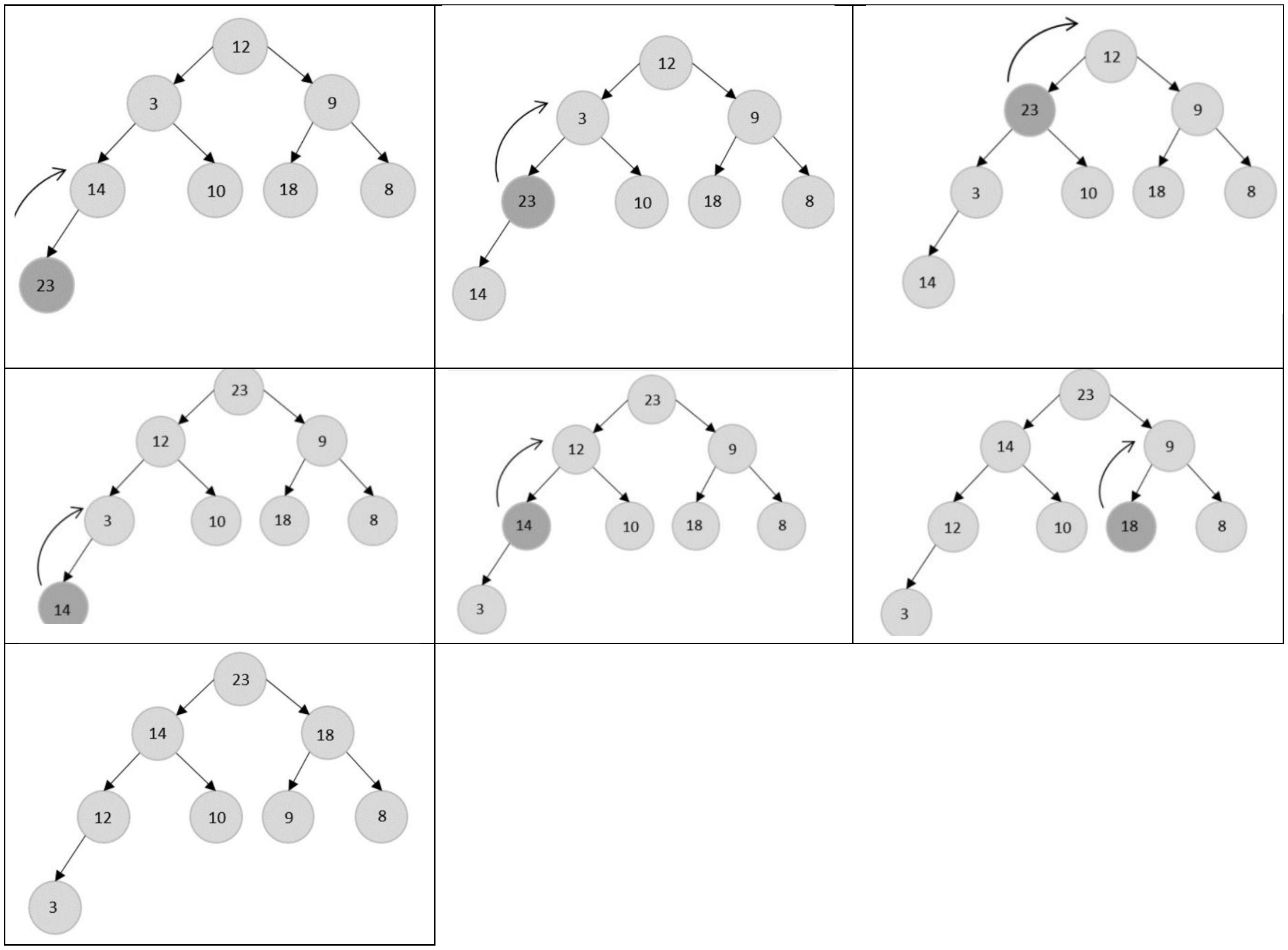

20
17
14
11
8
9
7
6
1
Max-Heap
In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Figure 1 : Max Heap

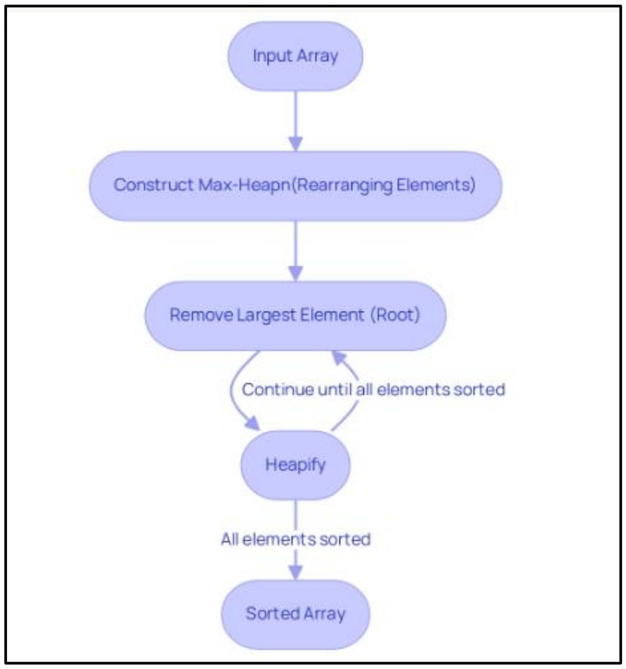
For example unsorted array given : [12, 3, 9, 14,10, 18, 8, 23]



1. For the first step, we compare and swap the the last node as 23(child) > 14(parent). The result [12, 3, 9, 23, 10, 18, 8, 14].
2. Next we check the parent[3] has a child[23] that greater than it. So we swap the value [12, 3, 9,23, 10, 18, 8, 14]
3. The step compare and swap will continue until all the parents are greater than their child where we got the max heap which is [23, 14, 18, 12, 10, 9, 8, 3].

➢ Heapify and Sort

- After the max structure is built, the array needs to be heapified.
- First, we need to build a structure as the parents can be calculated as (I - 2) /2 where I is an index of the descendant and left descendent can be (2*j + 1) and right descendant as (2* j + 2) where j is the index of the parent,
- then remove the first component (the largest in max heap) with the last component that isn't organized
- Then, we put the first element of the heap in array . The sorted sub array keep growing while the unsorted subarray shrinks until no element remain in the unsorted array. After the swapping, the heap becomes broken and needs to be heapified again.
- Implement the heap again with the remaining elements
- This step sorts the whole array in place
- Time complexity: O(n log n) in all cases, where n he number of elements in the array.


Input Array
Construct Max-Heapn(Rearranging Elements)
Remove Largest Element (Root)
Continue until all elements sorted
Heapify
All elements sorted
Sorted Array

Example: [23, 14, 18, 12, 10, 9, 8, 3]

1. We delete the 23(root), and swap with the last element which is 3.
2. Store the deleted element 23 in the last empty element, not in heap
3. Then we compare the parents with the child, if the child is greater, swap it.
4. Then, the array becomes [18, 14,19, 12, 10, 3, 8].
5. The process repeats for the remaining elements

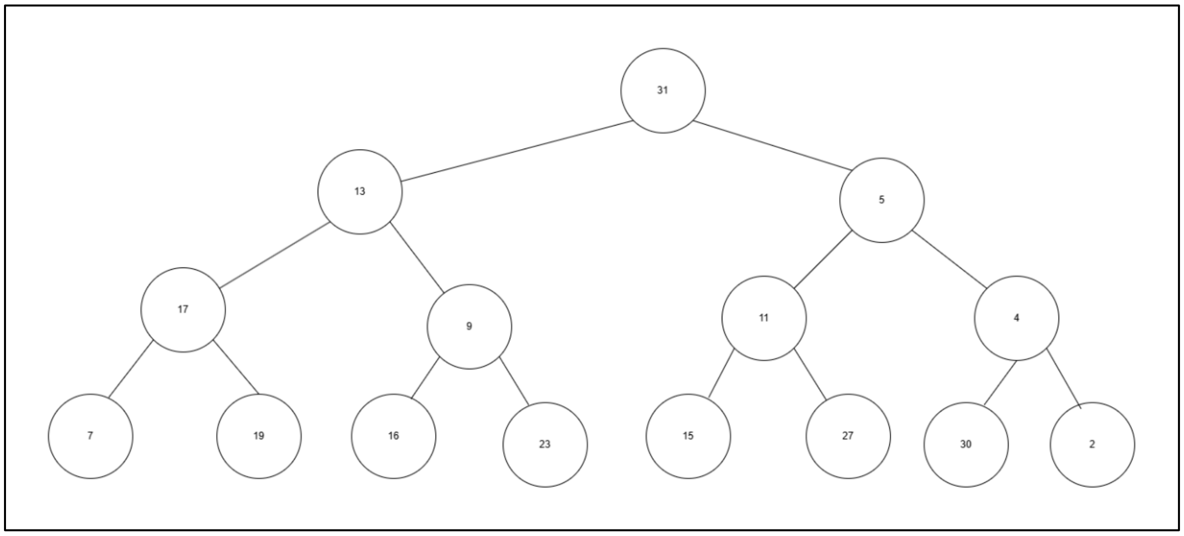
Complexity : O(n log n) for

Heap sort is efficient for large input sizes because there is no need for using more advanced concepts such as recursion. Moreover, the memory needs can be minimal.

QUESTION 3

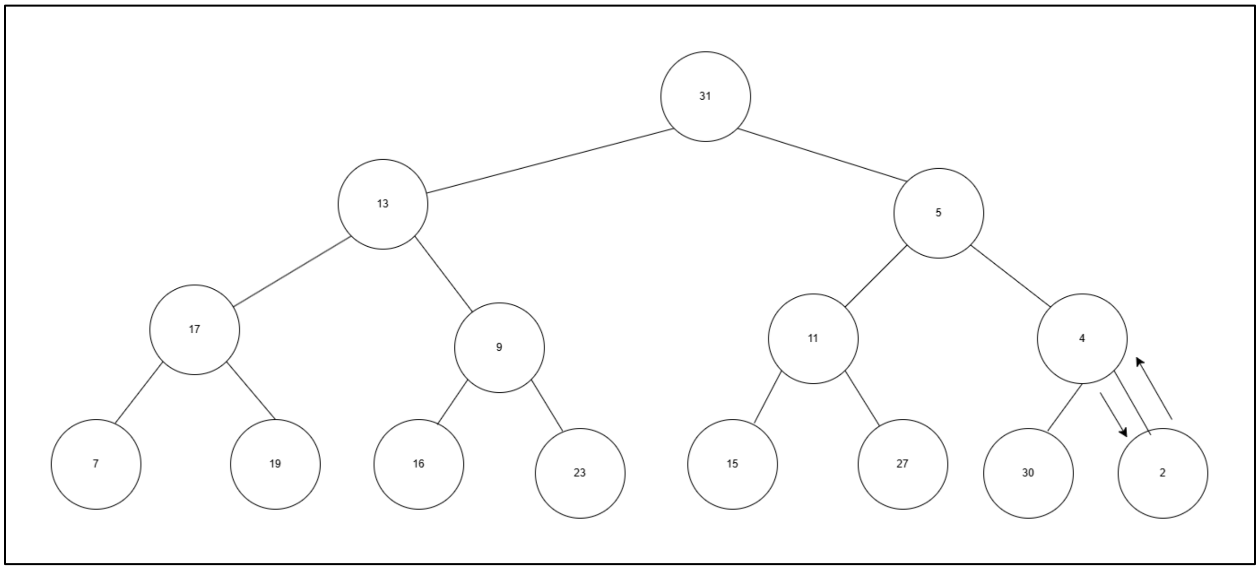
Solve the Problem: Provide a detailed step-by-step solution how heap sort can assist the binary search problem using this array of numbers [31, 13, 5, 17, 9, 11, 4, 7, 19, 16, 23, 15, 27, 30,2]. Let's say we want to search for 13 in the descending-sorted array.

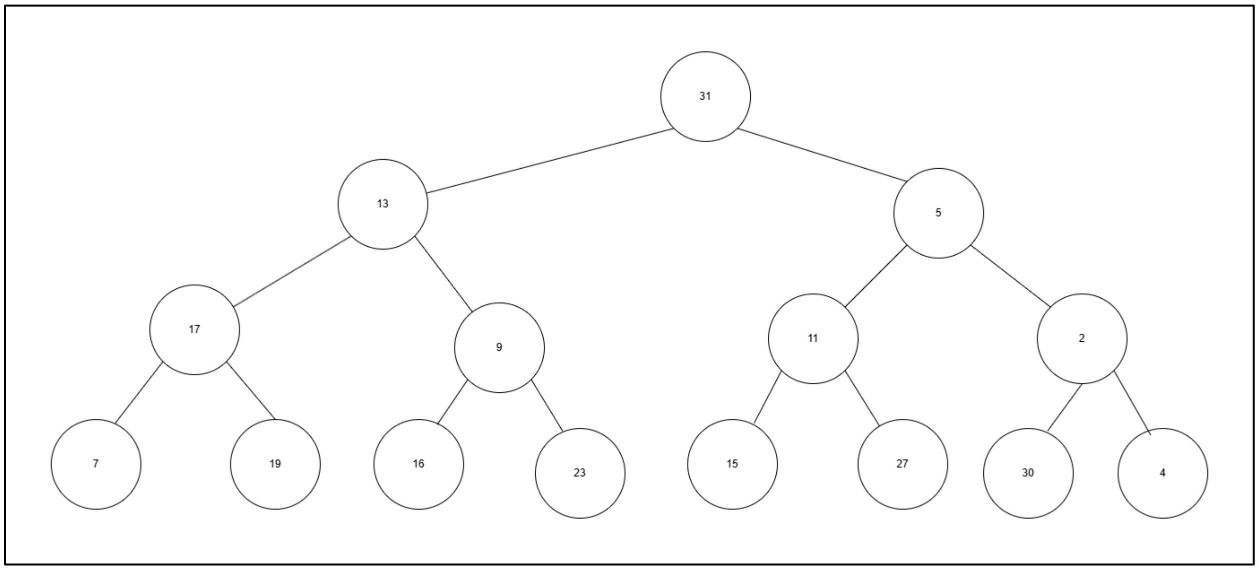
Step 1 : Represents the array into binary tree



Step 2 : Build a min Heap (i=6)

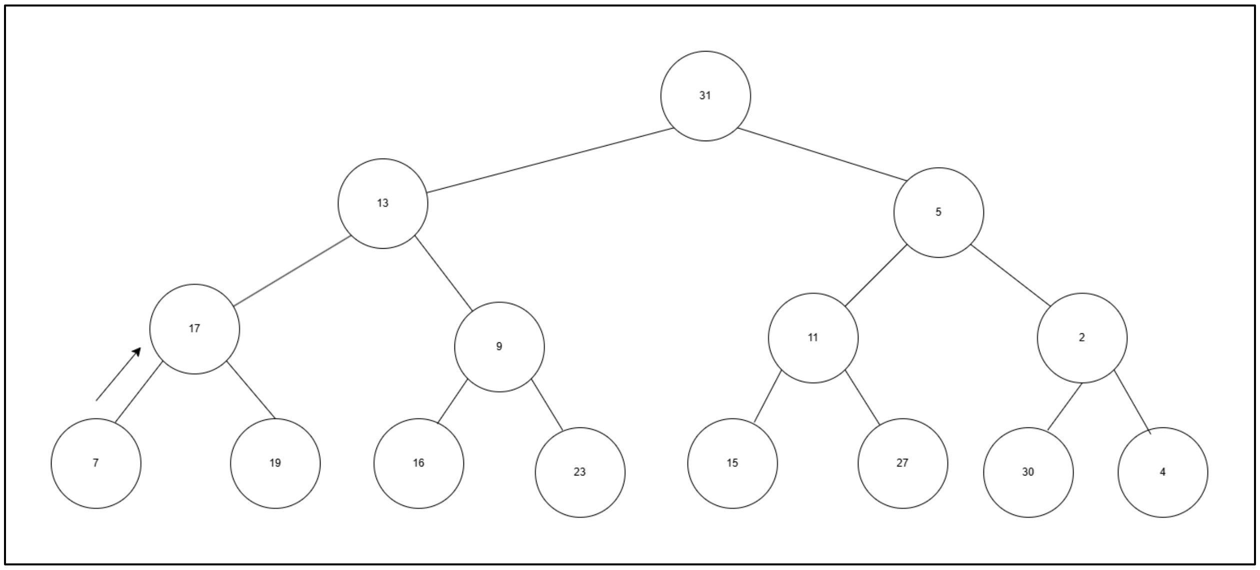
Starting from root at index arr[6] = 4, and the children are arr[13] = 30, arr[14] = 2. We compare 30 and 2, since 2 (child) is lower than 30, and 30 is lower than the parents 4, then we swap 2 with 4. The result is just shown in the figure below. [31, 13, 5, 17, 9, 11, 2, 7, 19, 16, 23, 15, 27, 30, 4]

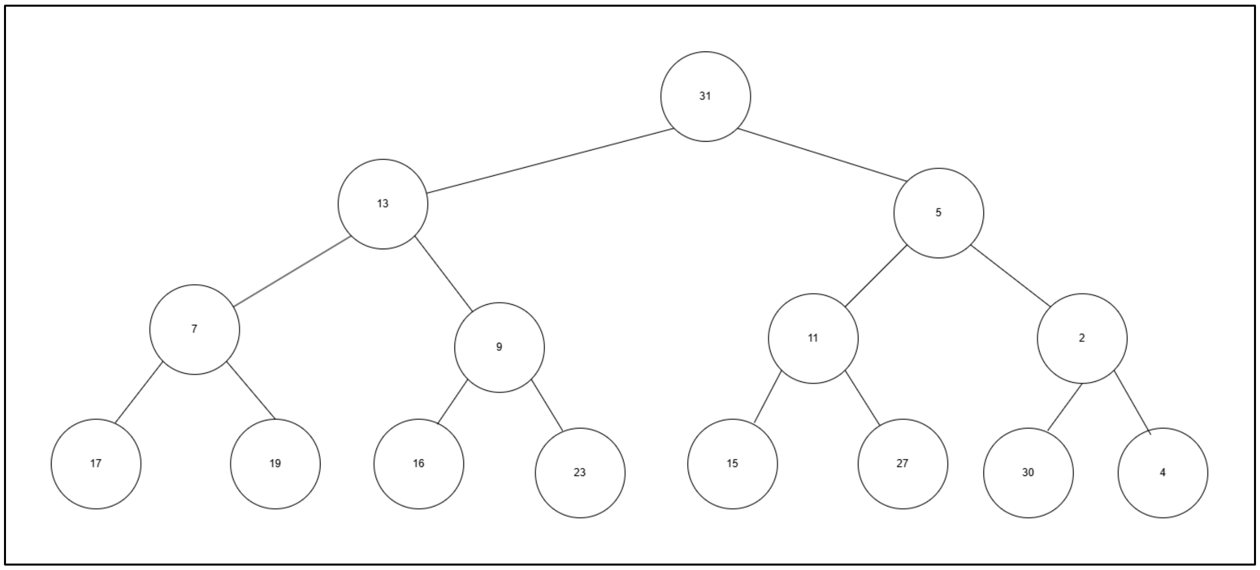




Step 3 : Next we compare i=5, i=4, i=3. Since all of this parents are smaller than their child, then there are no changes at this index.

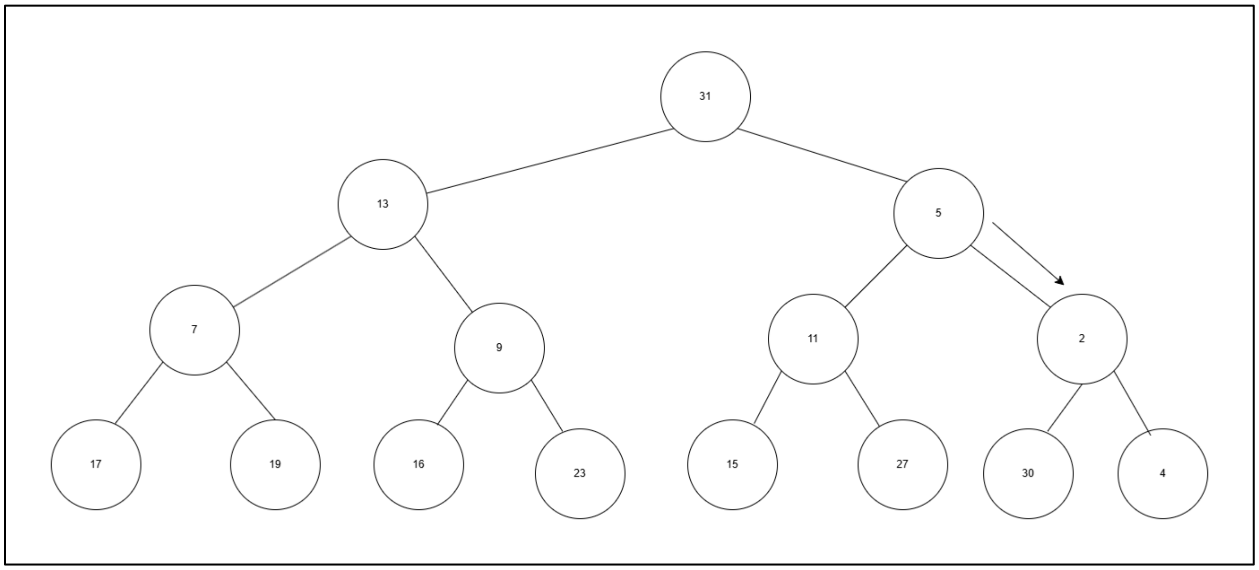
Step 4 : i = 3. We compare the children arr[7]= 7 and arr[8]= 19, since 7 is the smallest then we compare it when the parents. Since arr[3]= 17 is greater than the child arr[7] = 7, we swap them.



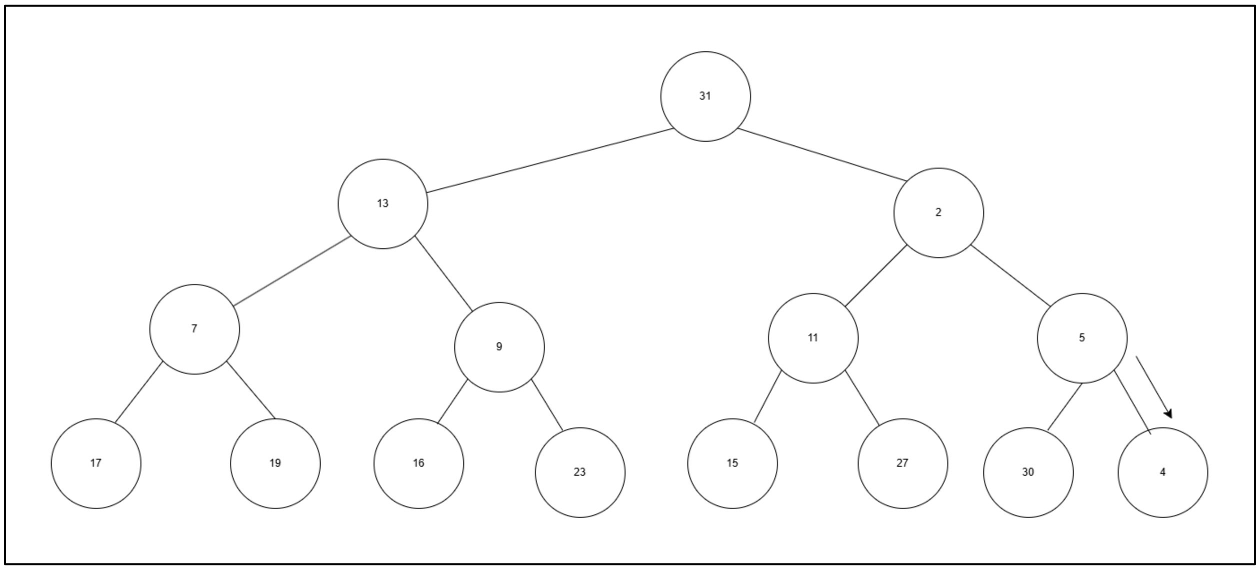


This is the result after we swap the value [31, 13, 5, 7, 9, 11, 2, 17, 19, 16, 23, 15, 27, 30, 4]

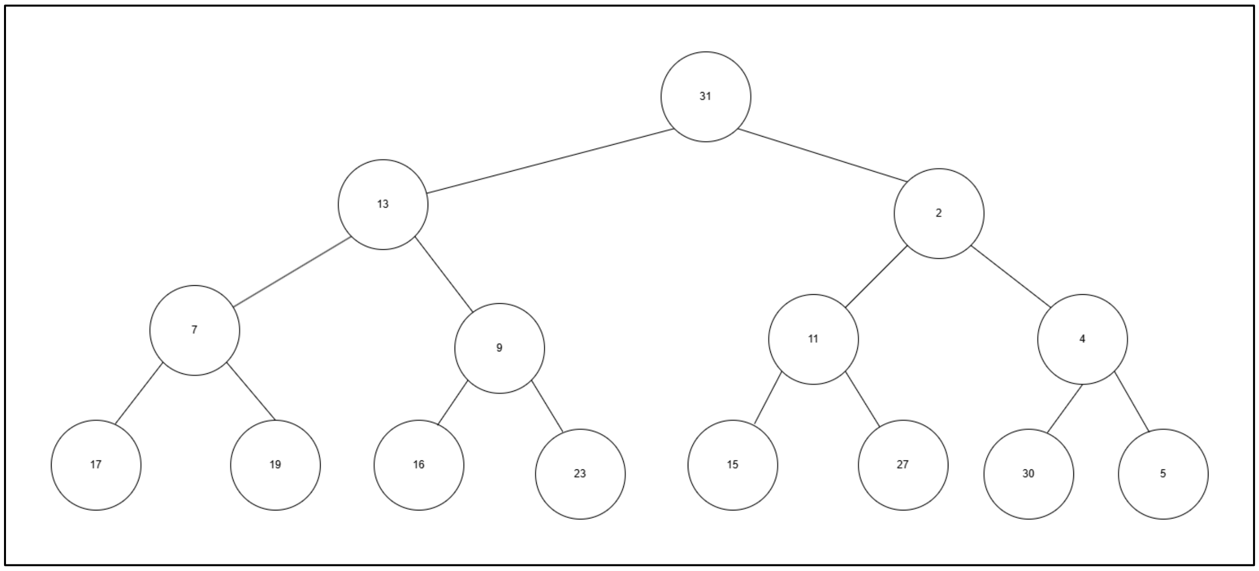
Step 5: Now, we move on to the i = 2. The parent arr[2] = 5, and the children are arr[5] = 11, arr[6] = 2. From the figures .. below, we can see that the parents 5 is greater than 2, then we swap arr[2] with arr[6].



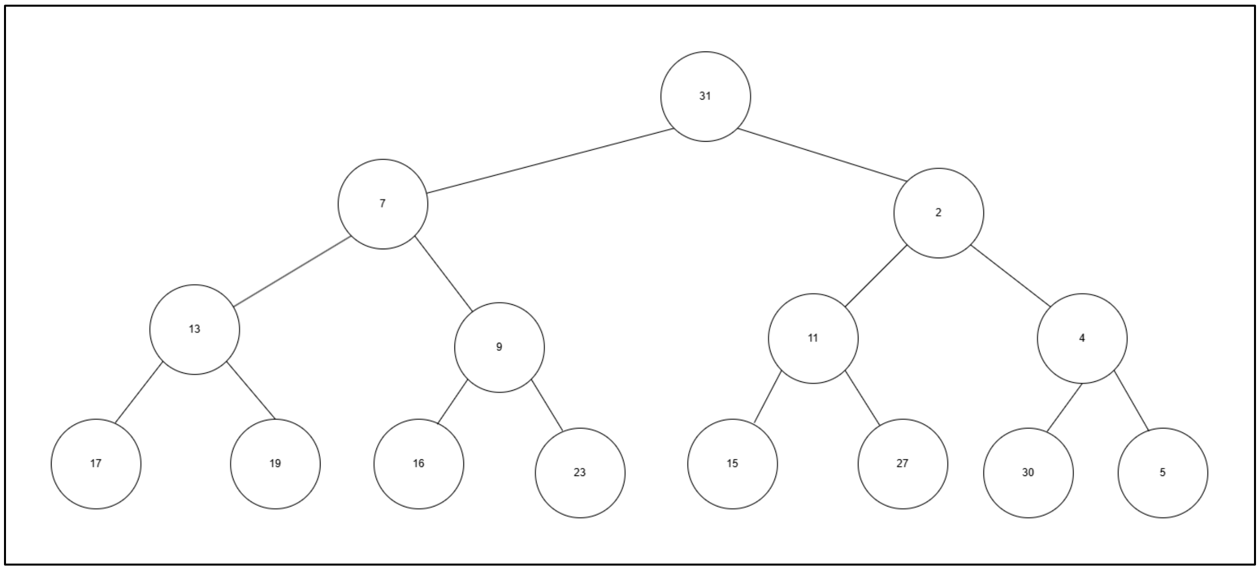
This is the result after the swap [31, 13, 2, 7, 9, 11, 5, 17, 19, 16, 23, 15, 27, 30, 4]



Then , we compare the arr[6] with the children arr[13] and arr[14]. Since 4 is the smallest children and lower than 5(parents) then we swap them. The array becomes [31,13, 2, 7, 9, 11, 4, 17, 19, 16, 23, 15, 27, 30, 5]



Step 6 : Next we move to the index 1= 13 and the children are arr[3] = 7 and arr[4] = 9. We swap 13(parent) with 7 since 7 is the smallest child.



Step 7 : Next, parent arr[0]=31, childrens are arr[1] =7 and arr[2]=2. The smallest children is 2 then we swap 31(parent) with 2.

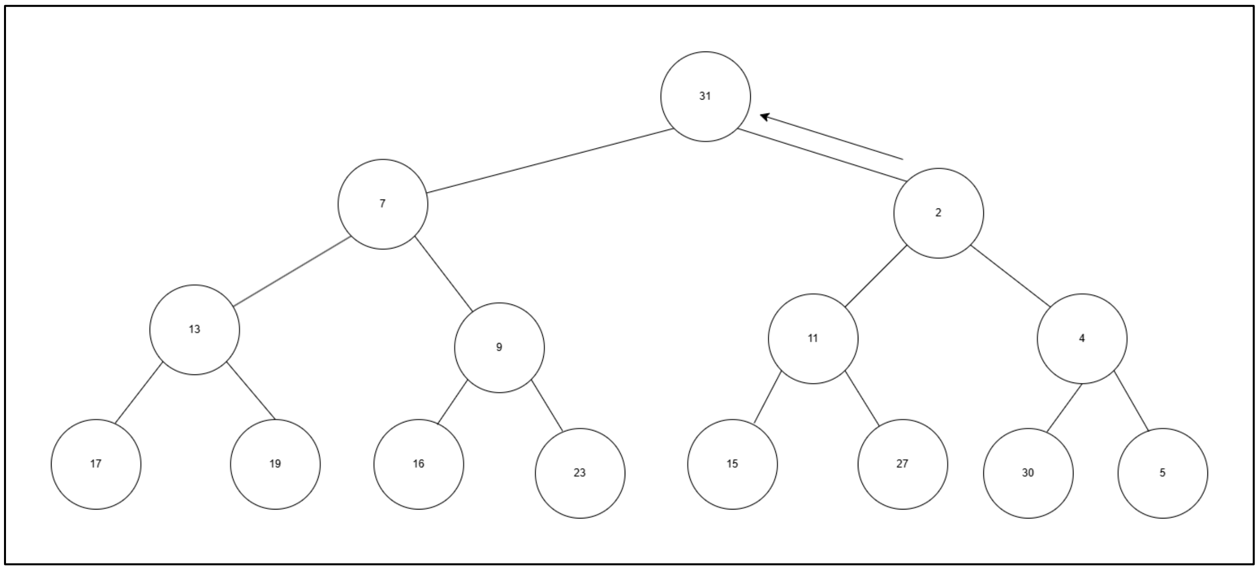
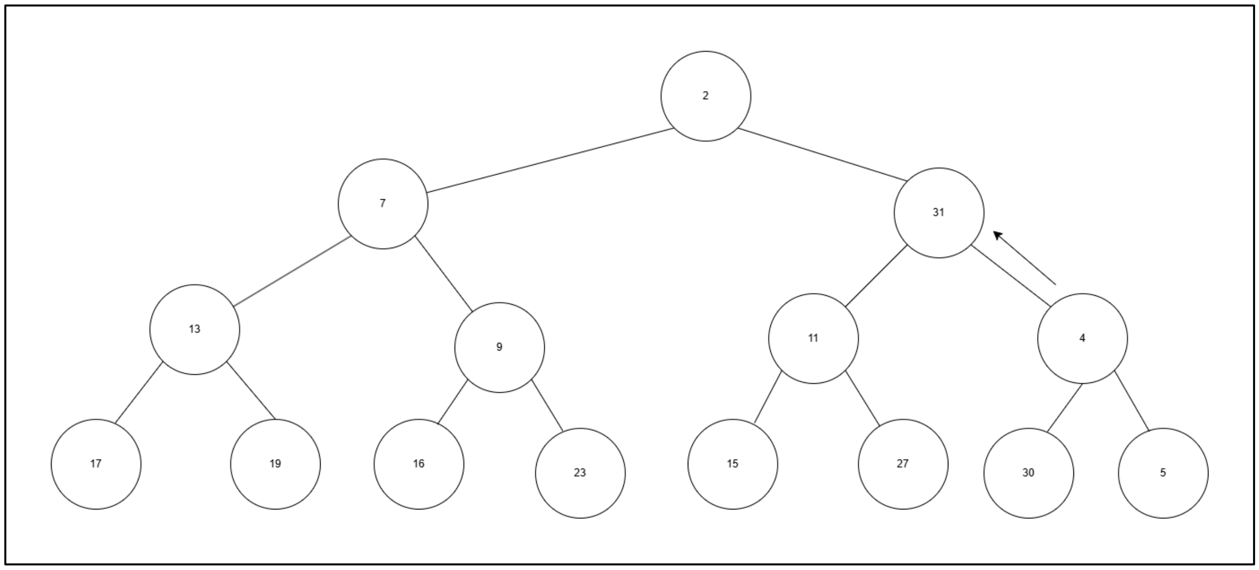


Figure.. shown the result after the swapping. Since the binary is still not in heap min. The process repeated at index 2 arr[2] = 31 and the children are arr[5] = 11 arr[6] = 4. Since 4 is the smallest and then we compare it with the parents 31, since 4 is smaller than 31 then we swap it.



This is the result after the swap process. Moving to the next element which is arr[6]= 31 and the childrens arr[13]=30 and arr[14]= 5. Since 5 is the smallest then we compare it with the 31(parents) and we swap them as 5 is smaller than 31.

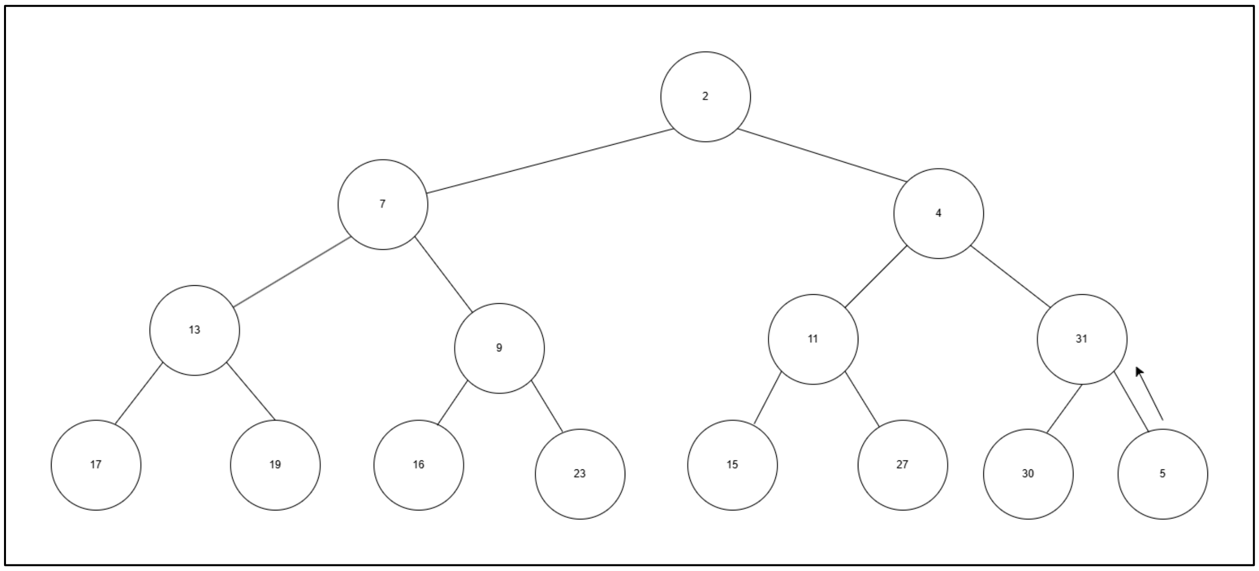
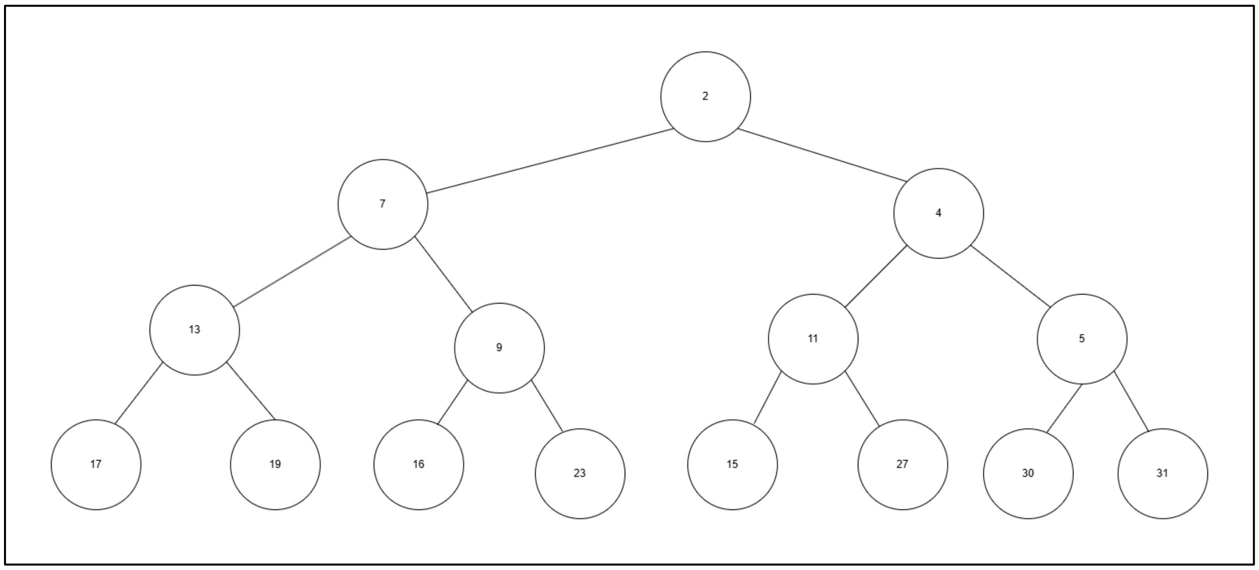


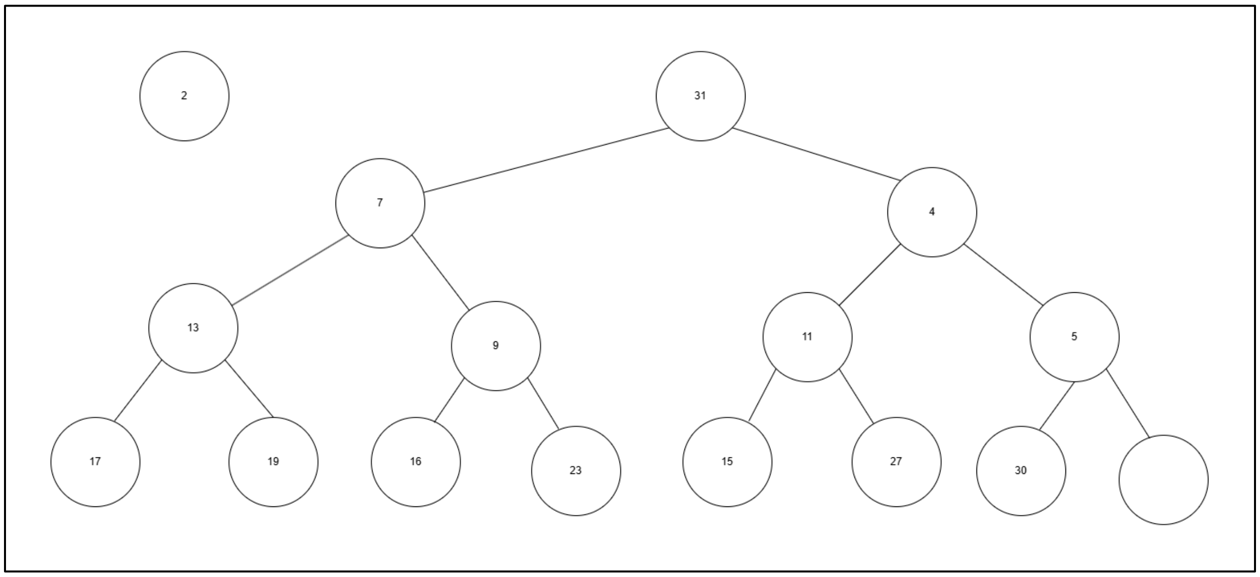
Figure below shows the result. Finally, all the parent nodes is smaller than their children and the tree has achieved the min-heap condition. The array become [2, 7,4,13, 9, 11, 5, 17, 19, 16, 23, 15, 27, 30, 31].



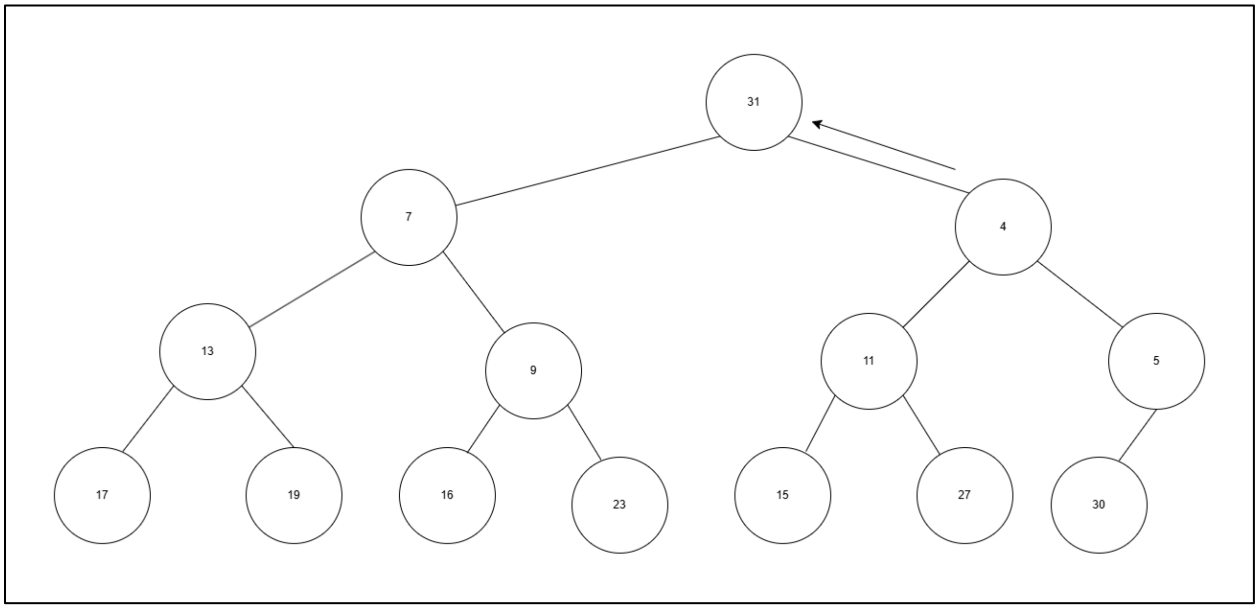
Sorting process

- **First Extraction**

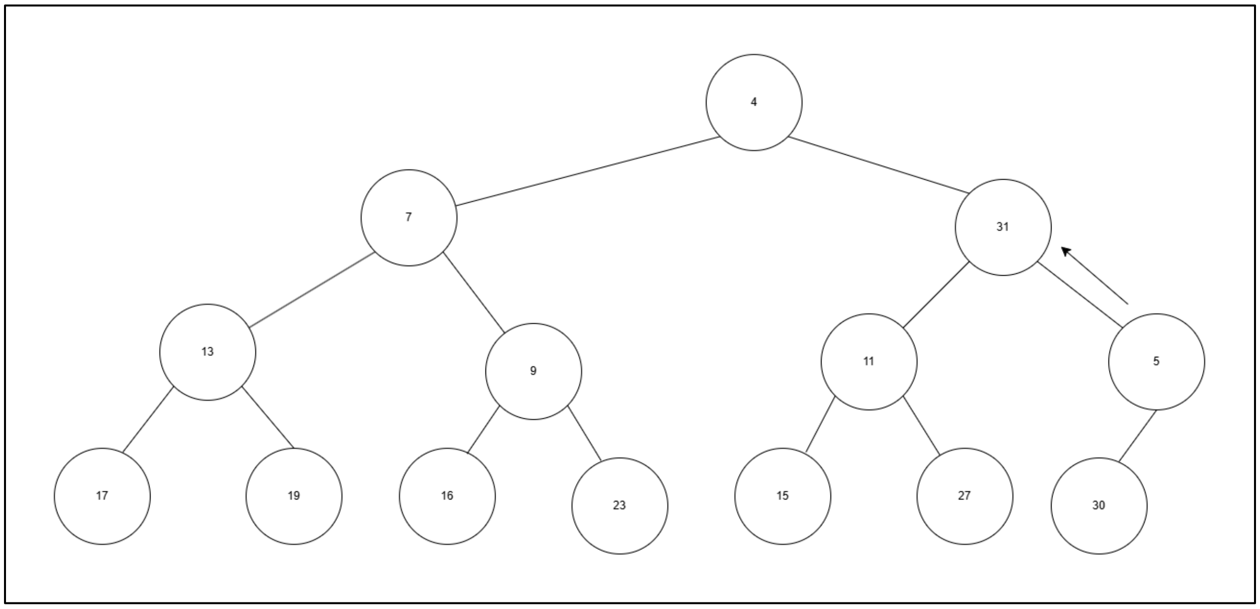
The sorting process started with the deleted of 2 (min element i=0) and we swap the root with the last element which is 31. The deleted element is stored in the last empty array i=14. The array become : [31, 7, 4, 13, 9, 11, 5, 17, 19, 23, 15, 27, 30, 2] as represents in the figure .. below.



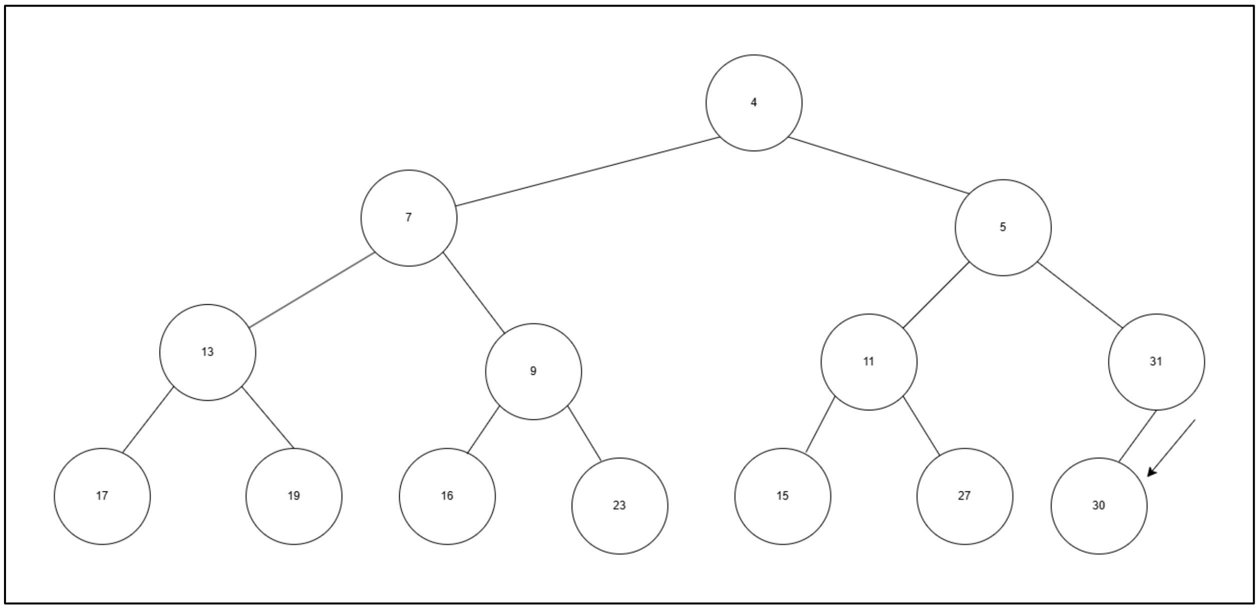
Next, we build the min heap again. At index 0=31, it has children 7 and 4. Since 4 is the smallest children and smaller than 31, then we swap 31(parent) with 4.



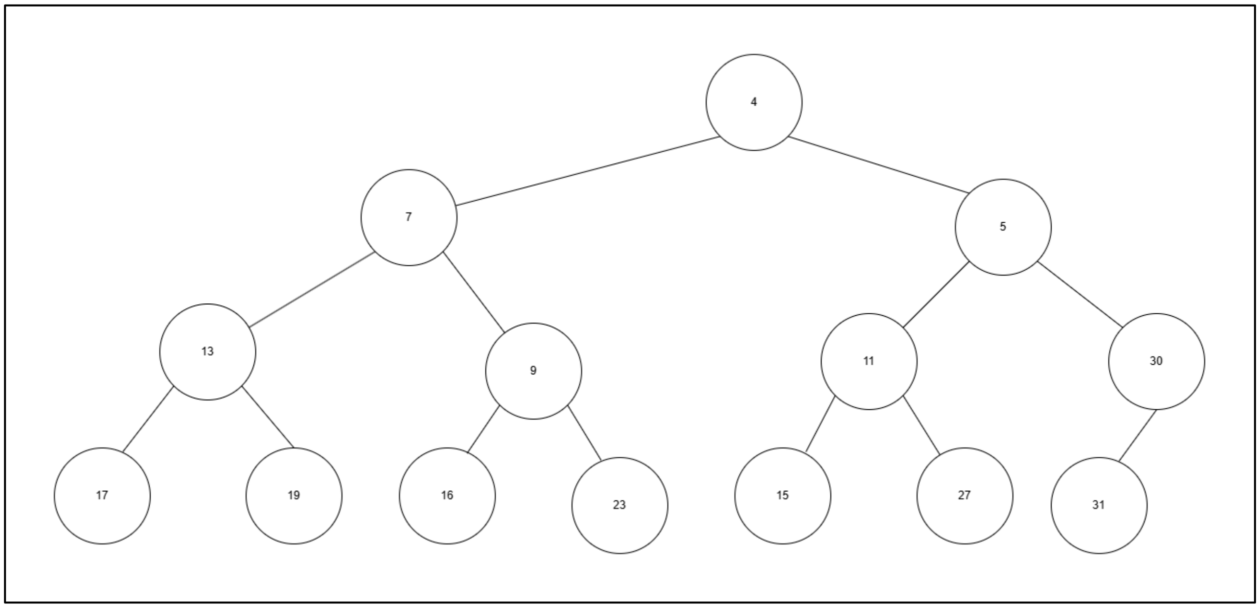
After the swapping process, we still need to heapify at index arr[2]=31 that has children 11 and 5. We need to swap 5 and 31 as 5 is smaller than 31(parent) and the smallest among the children .



This is the result after swapping. We still need to swap the value arr[6] since the children 30 is smaller than 31

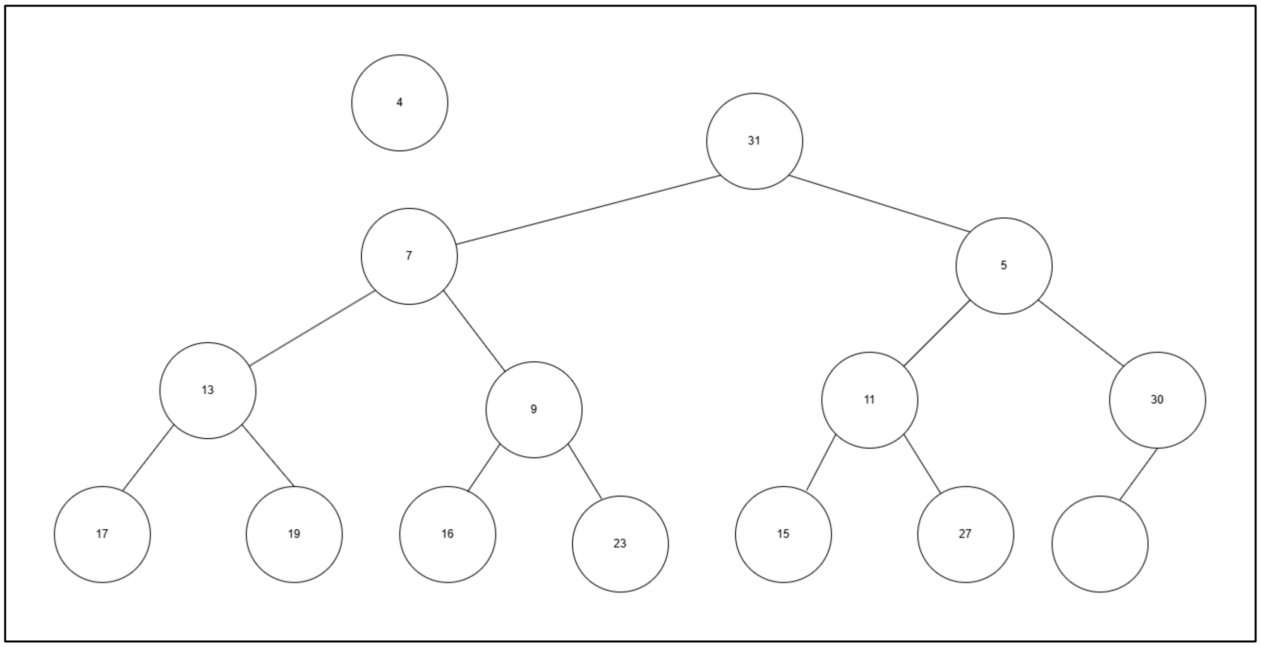


This is the result after first extraction first. [4, 7, 5, 13, 9, 11, 30, 17, 19, 16, 23, 15, 27, 31, 2]

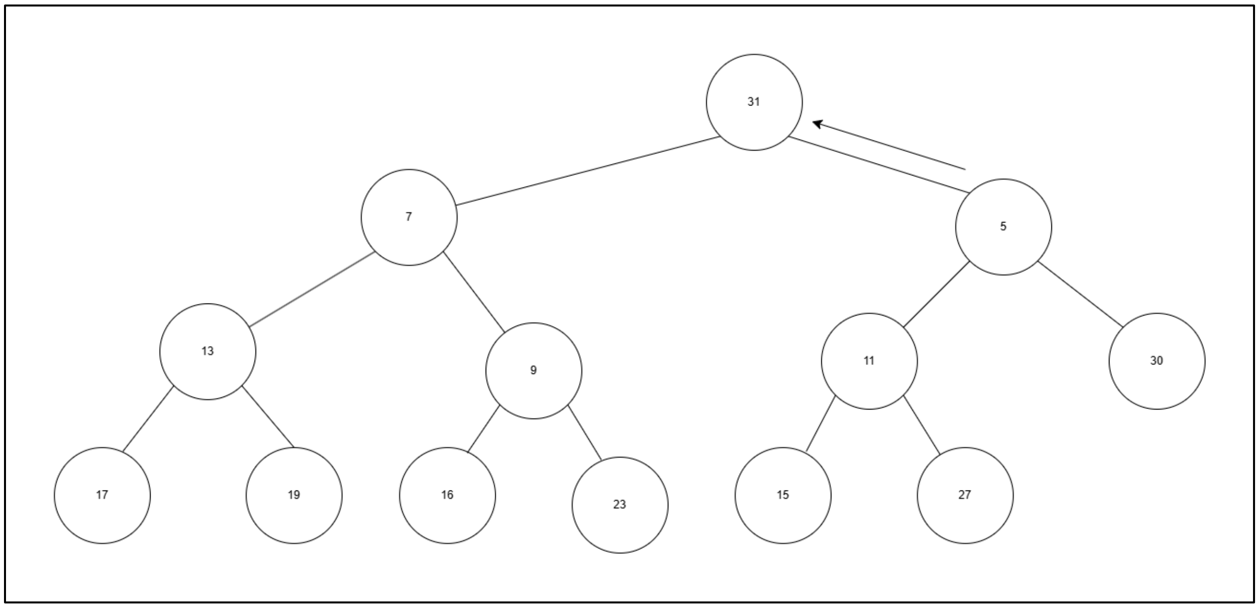


- **2nd extraction**

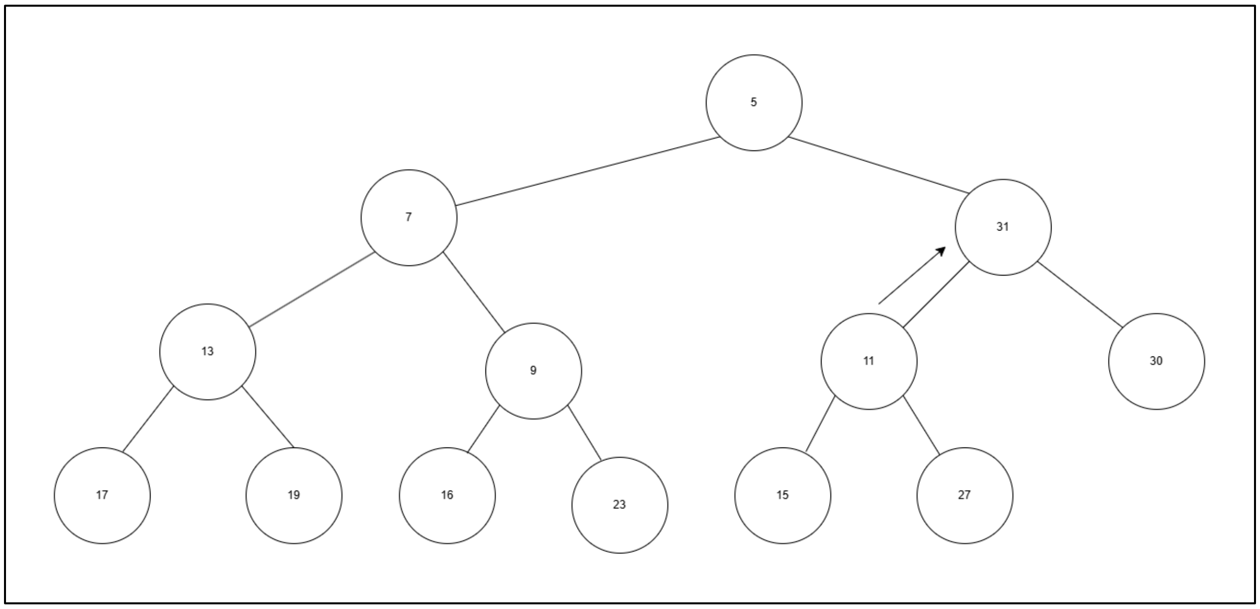
  Next, we deleted the minimum element, which is 4, and swapped the root with the last element, 31. The array becomes [31, 7, 5, 13, 9, 11, 30, 17, 19, 16, 23, 15, 27, 4, 2]



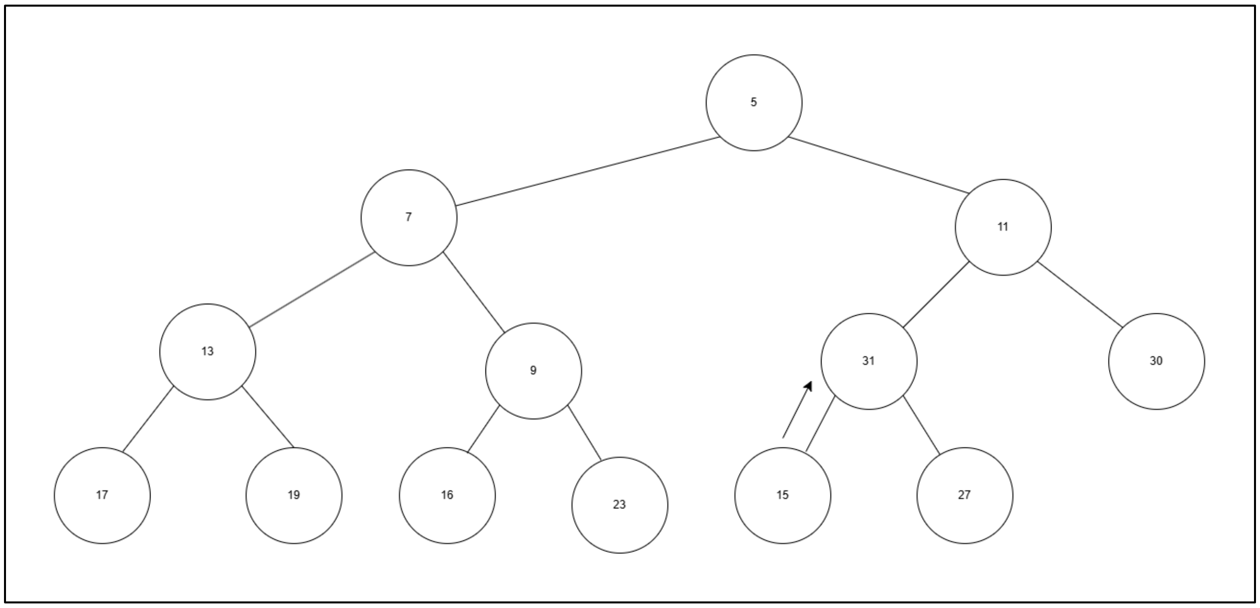
Next, we need to heapify again at index 0. We compare the children 7 and 4, and the smallest value, 4, is compared with 31. Swap 31 (parent) with 5(child) as it is lower than 31.



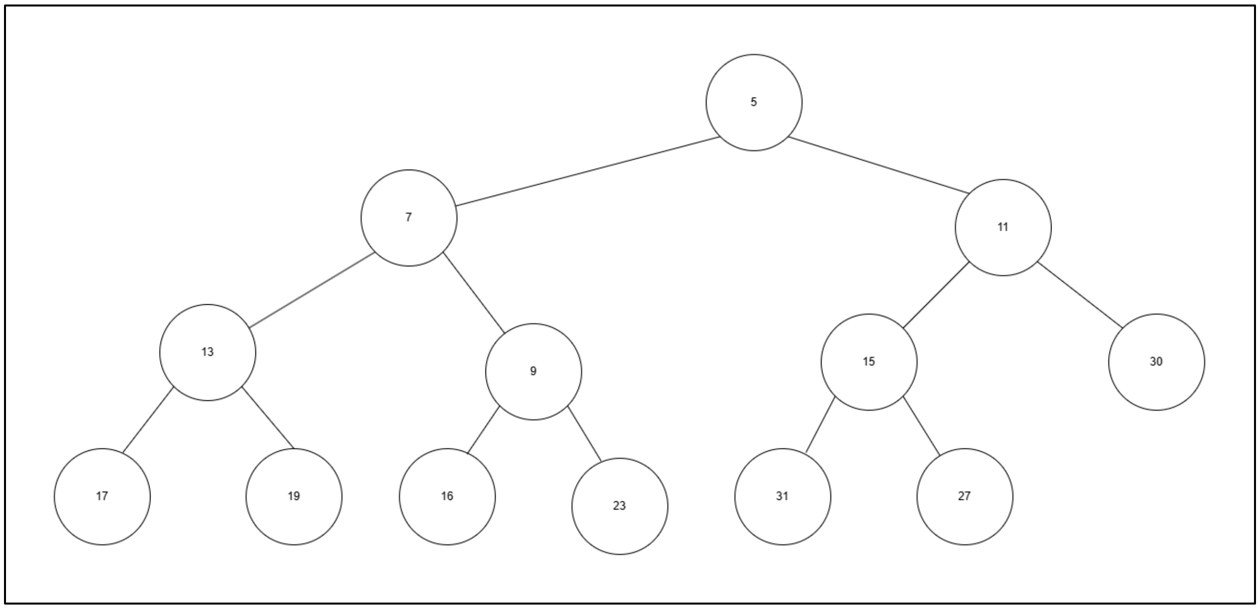
This is the result after the swapping process. Next, move to the next element arr[2] = 31. Since 11 is lower than 31 and 30, we swap them.



This is the result [5, 7, 11, 13, 9, 31, 30, 17, 19, 16, 23, 15, 27, 4, 2]. Then, we heapify again at arr[5]. The children are 15 and 27 since 15 < 27 , we compare it with the 31 and swap 31 with 15 since 15 is smaller than 31.

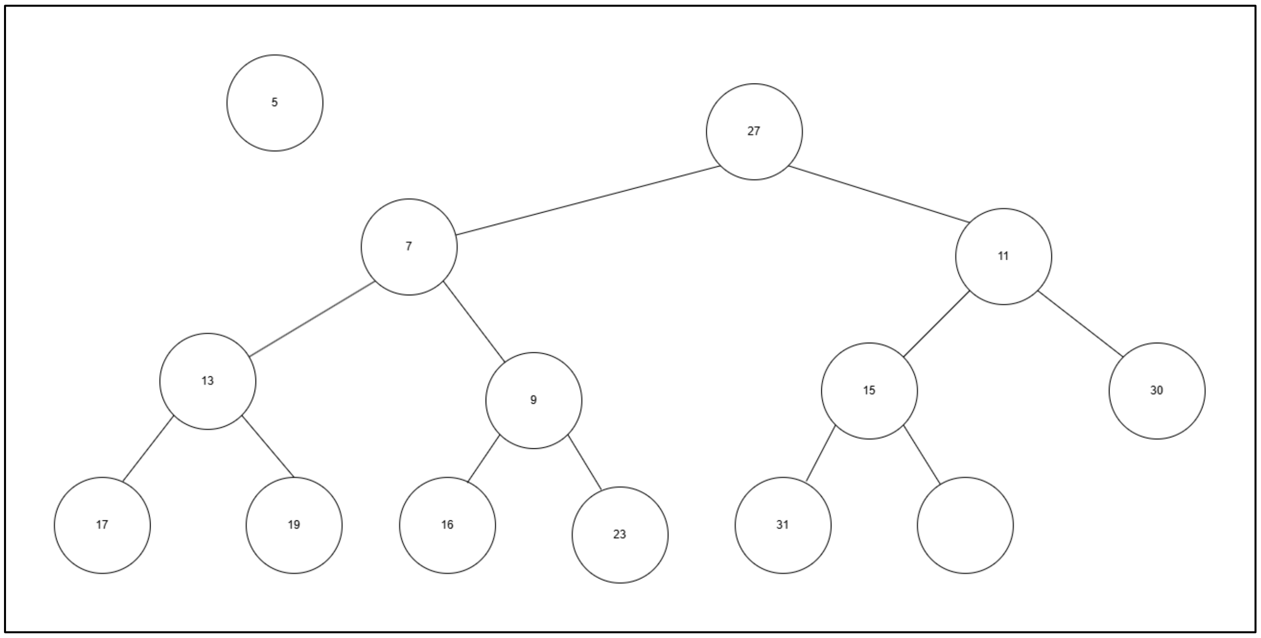


This is the result after the swap process. [5, 7, 11, 13, 9, 15, 30, 17, 19, 16, 23, 31, 27]

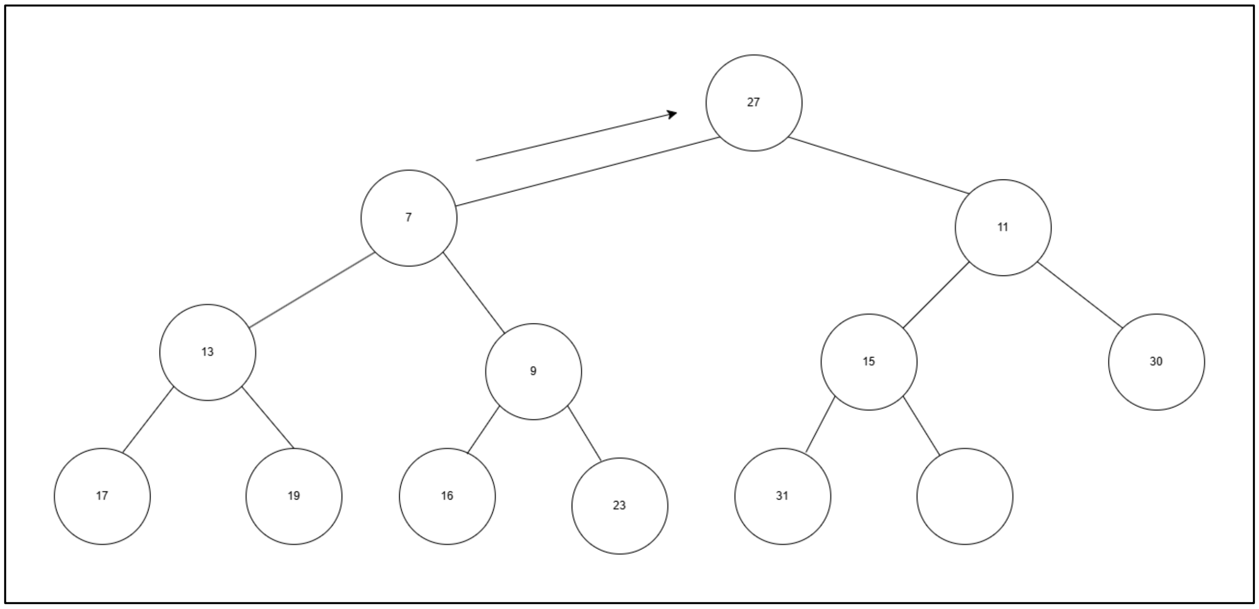


- **3rd extraction**

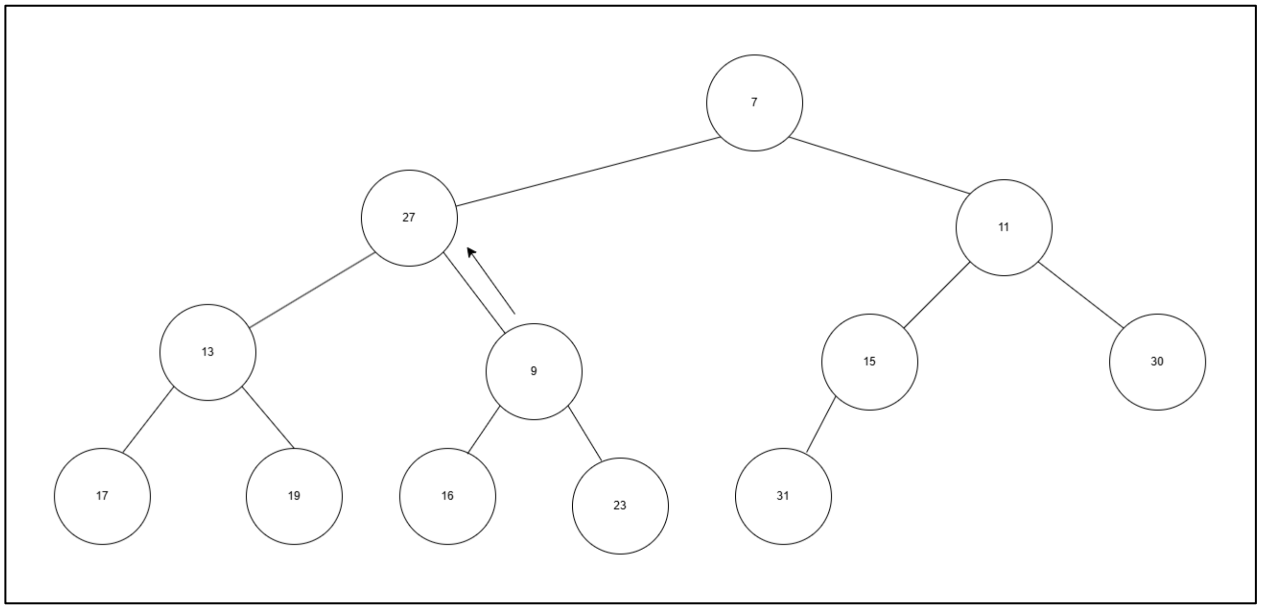
  The sorting process continue with root 5 deleted, and swap 5 move the last element which is 27. The array become 27, 7, 11, 13, 9, 15, 30, 17, 19, 16, 23, 31, 5, 4, 2]



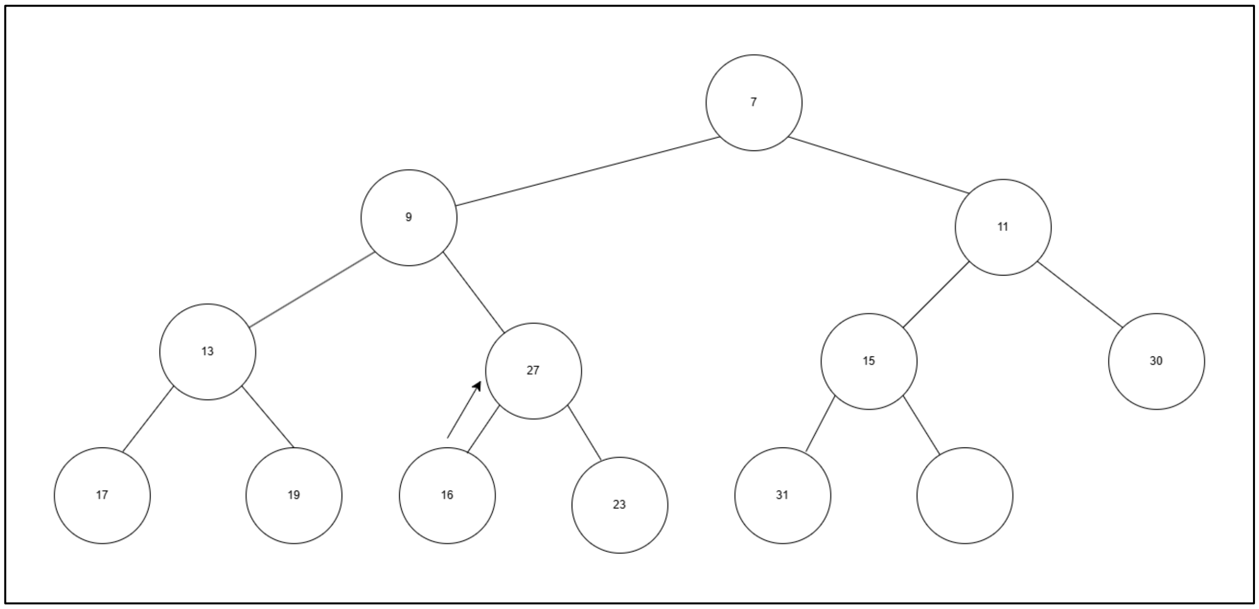
Then, we need to reheapify again. Start with 27 with 7 since 7 is smaller than 27.



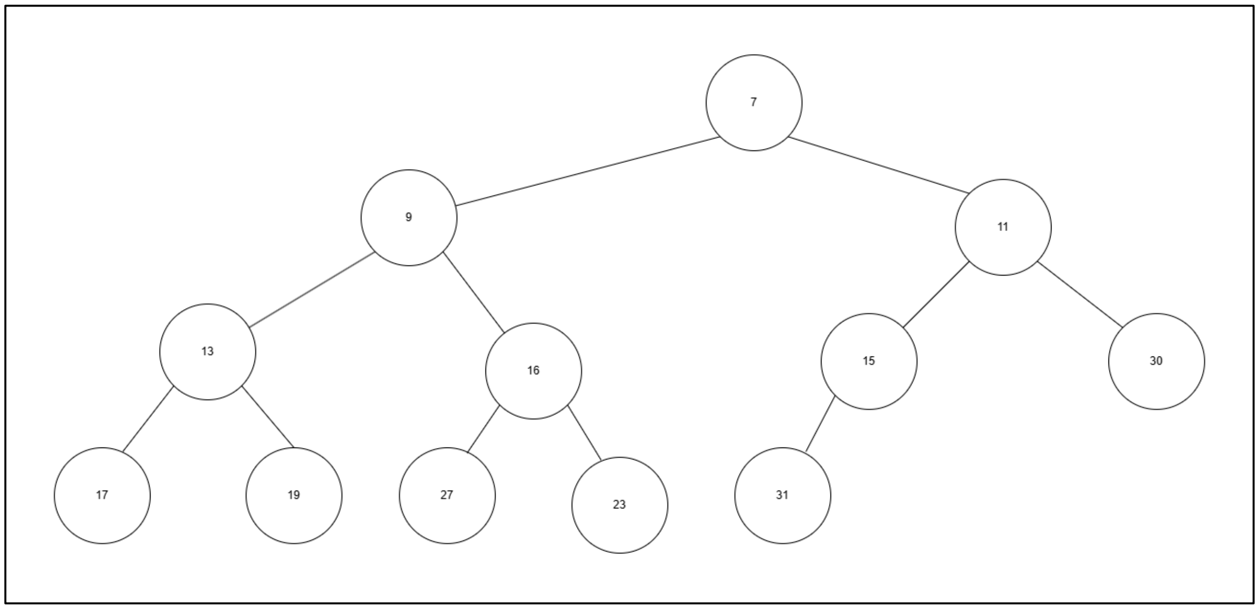
The array becomes [7, 27, 11, 13, 9, 15, 30, 17, 19, 16, 23, 31]. Move to the next element, which is index 1 = 27. Swap 9 with 27



This is the result after swap 27 with 9 : [7, 9, 11, 13, 27, 15, 30, 17, 19, 16, 23, 31]. Next, we swap 27 with 16 since 16 is smaller than 23 and 27.

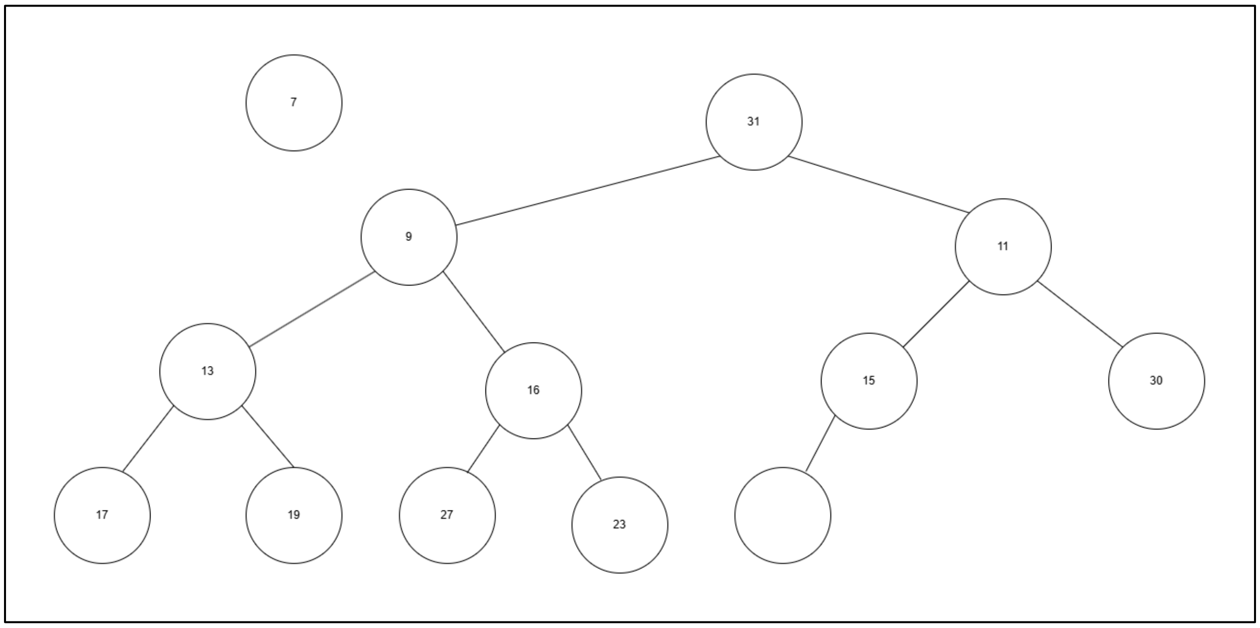


The result after swapping : [7, 9, 11, 13, 16, 15, 30, 17, 19, 27, 23, 31].

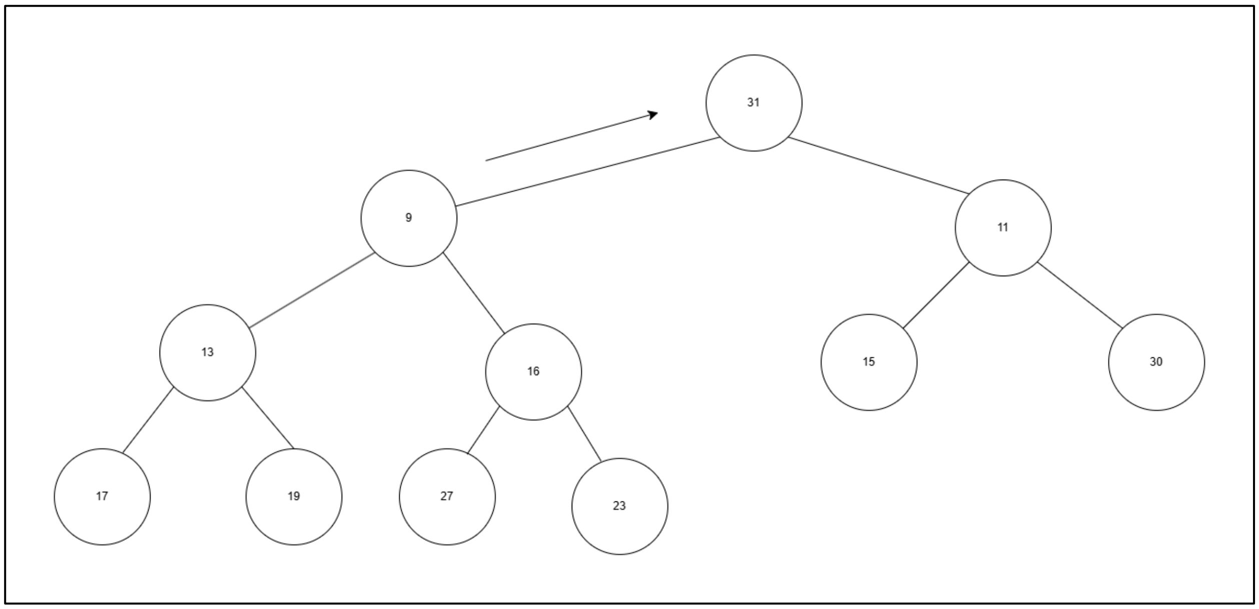


- **4th extraction**

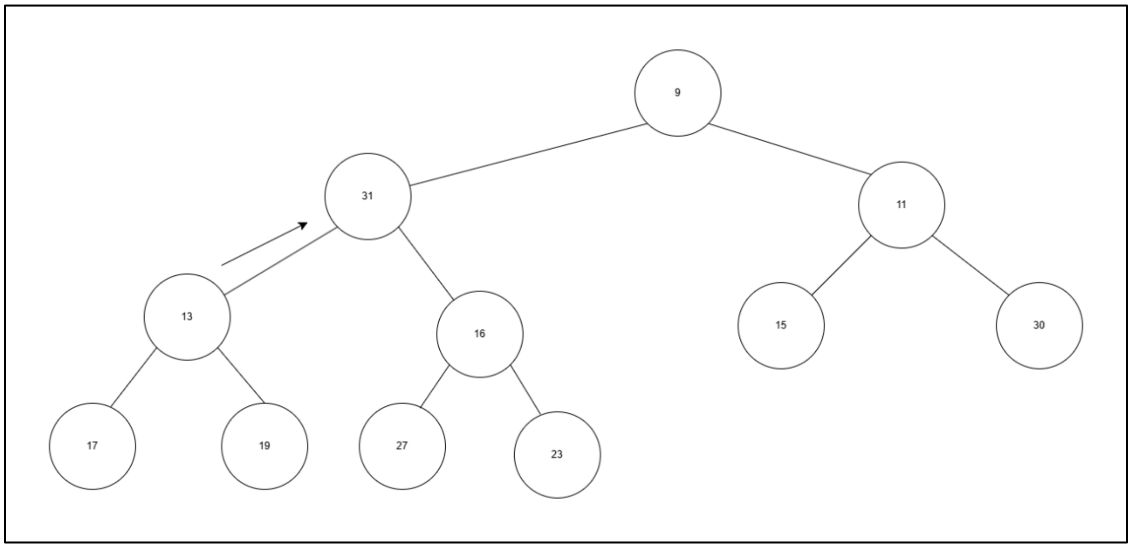
  Next, we delete the root = 7 and replace it with the last element = 31. The array : [31, 9, 11, 13, 16, 15, 30, 17, 19, 27, 23, 7, 5, 4, 2]



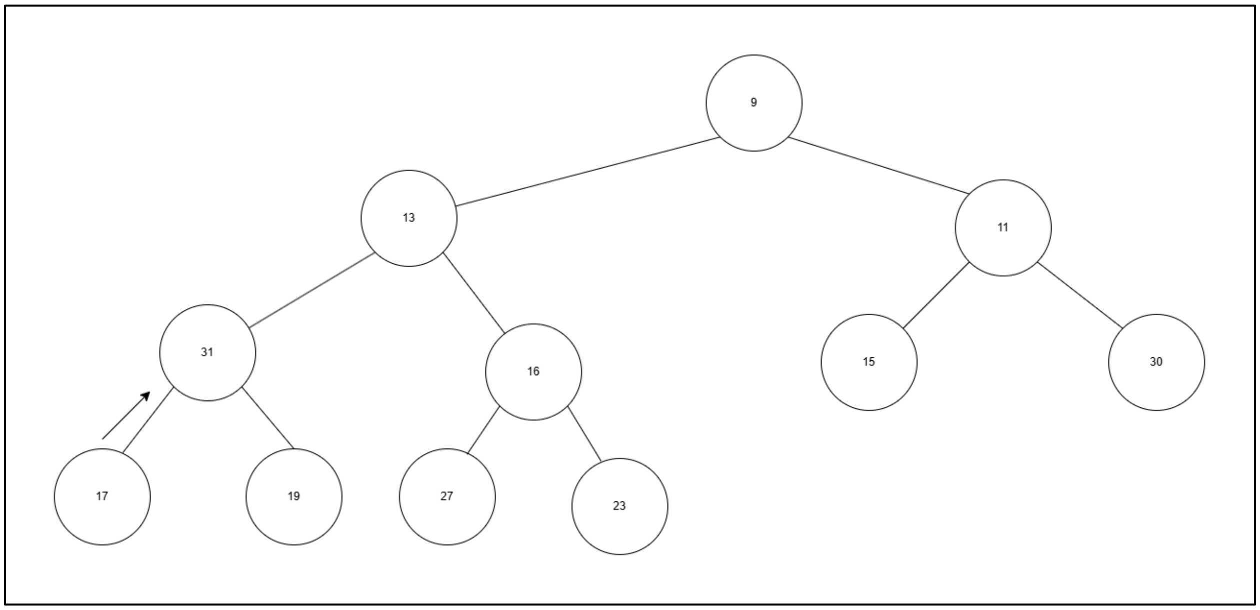
  Next, we heapify again swap 31 with 9.



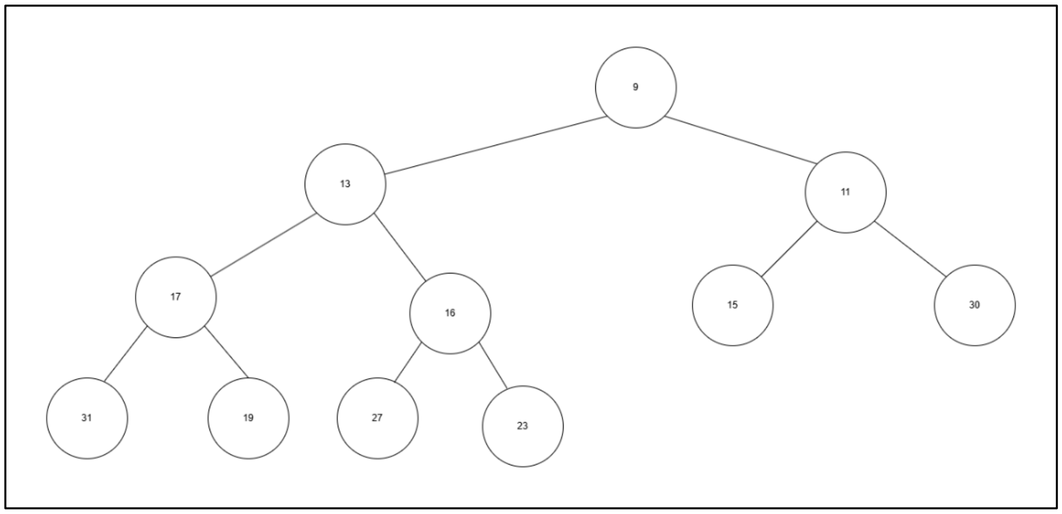
The result after swapping 31 with 9: [9, 31, 11, 13, 16, 15, 30, 17, 19, 27,23]. Then we compare 31(parent) with the children, 13 and 16 and swap 31 with 13.



The result : [9, 13, 11, 31, 16, 15, 30, 17, 19, 27, 23]. Next, compare parent 31 with 17 and 19. Since 17 is the smallest, we swap it with 31(parent).

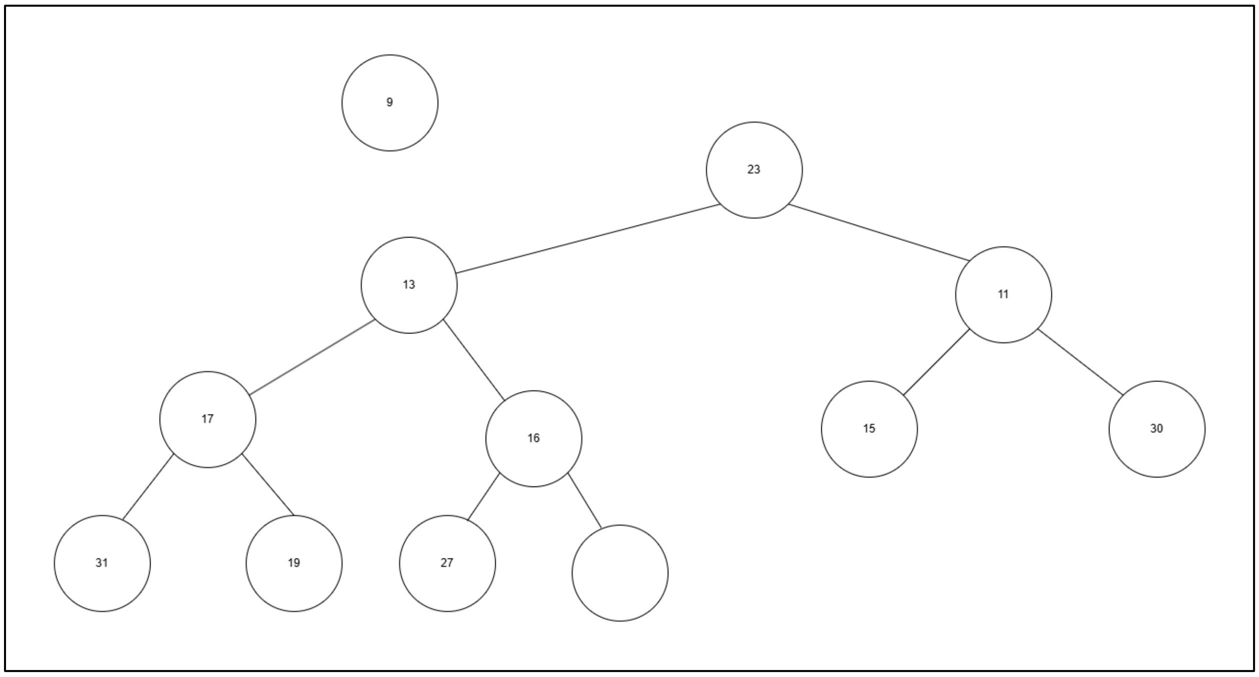


The result : [9, 13, 11, 17, 16, 15, 30, 31, 19, 27, 23].

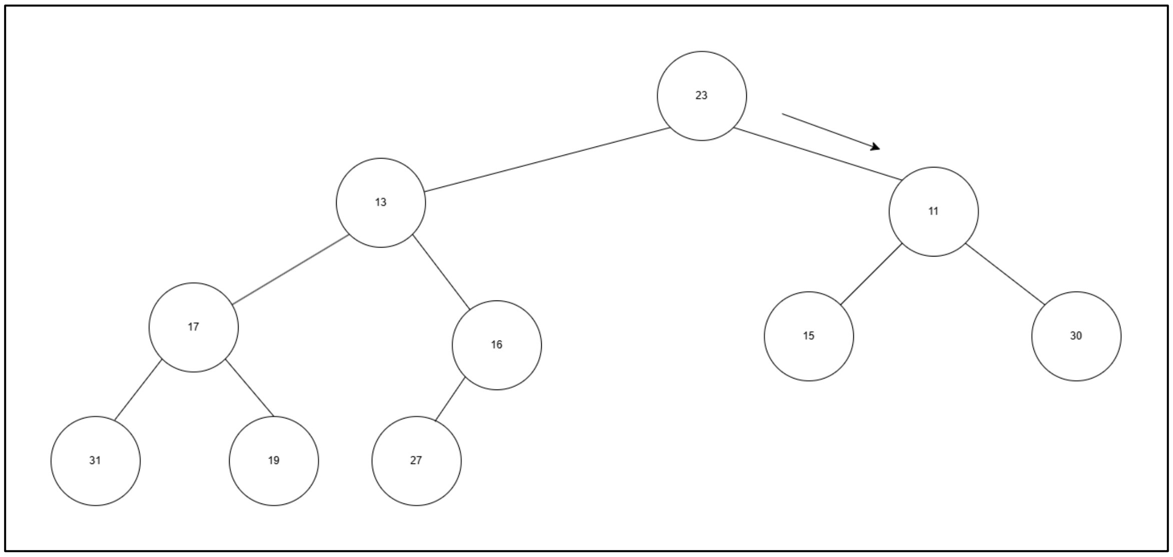


- **5th extraction**

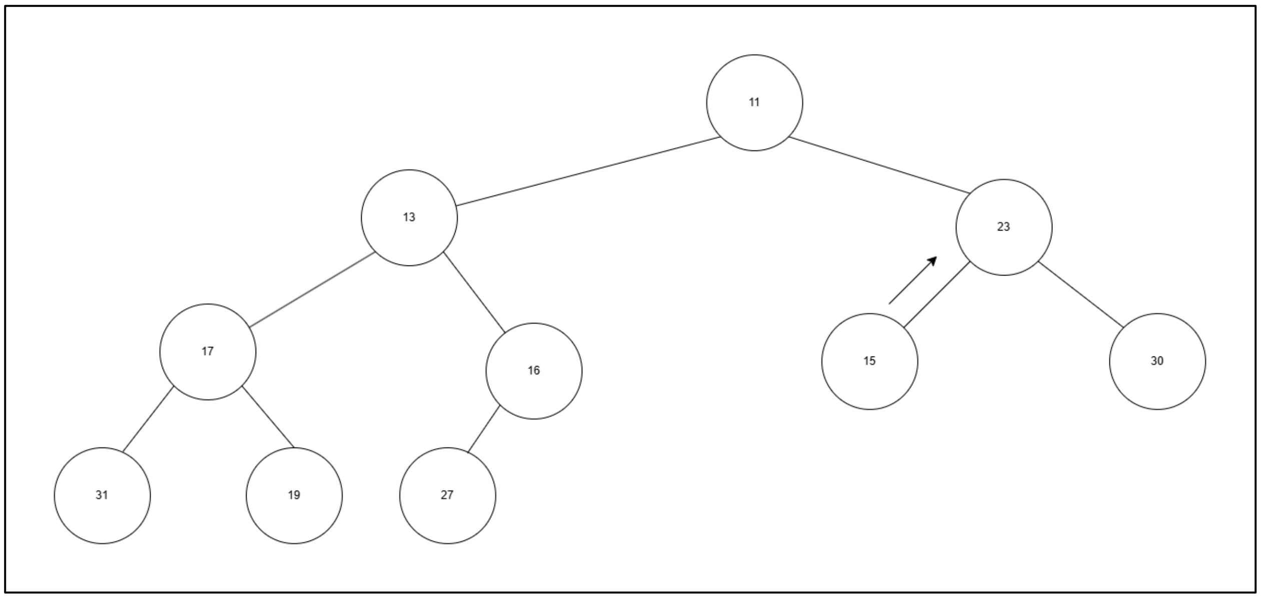
  Next we extract the minimum element, which is 9, and place it at the end of the array. We replace it with the 23(last element).



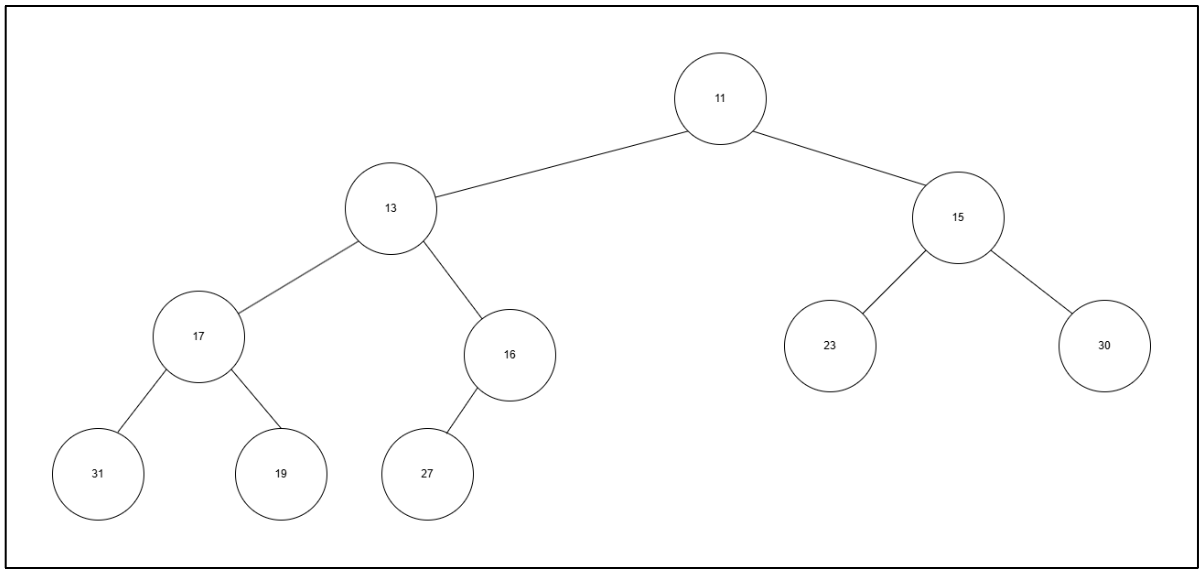
Heapify again at arr[0]=23 with 11 since 11 is smaller than 23 we swap them.



The result after reheapify : [11, 13, 23, 17, 16, 15, 30, 31, 19, 27]. Then , we swap 15 and 23.

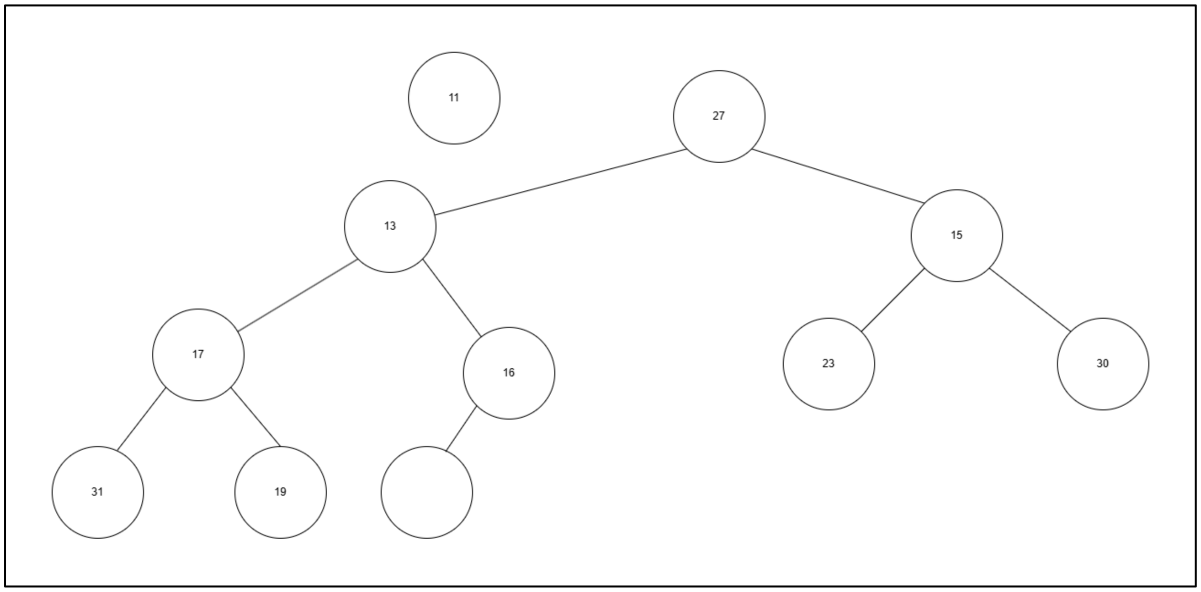


The result after reheapify: [11, 13, 15, 17, 16, 23, 30, 31, 19, 27]

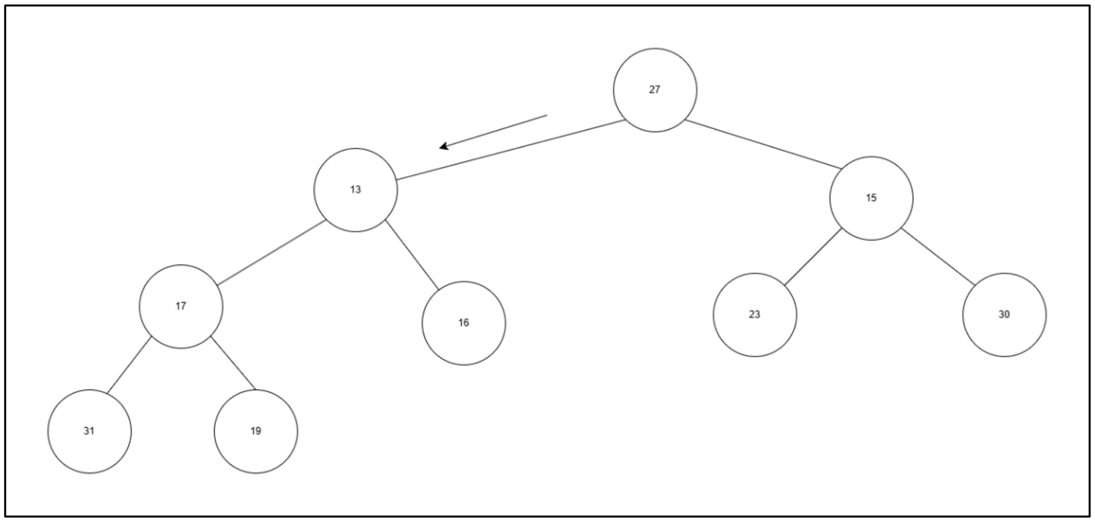


- 6th extraction

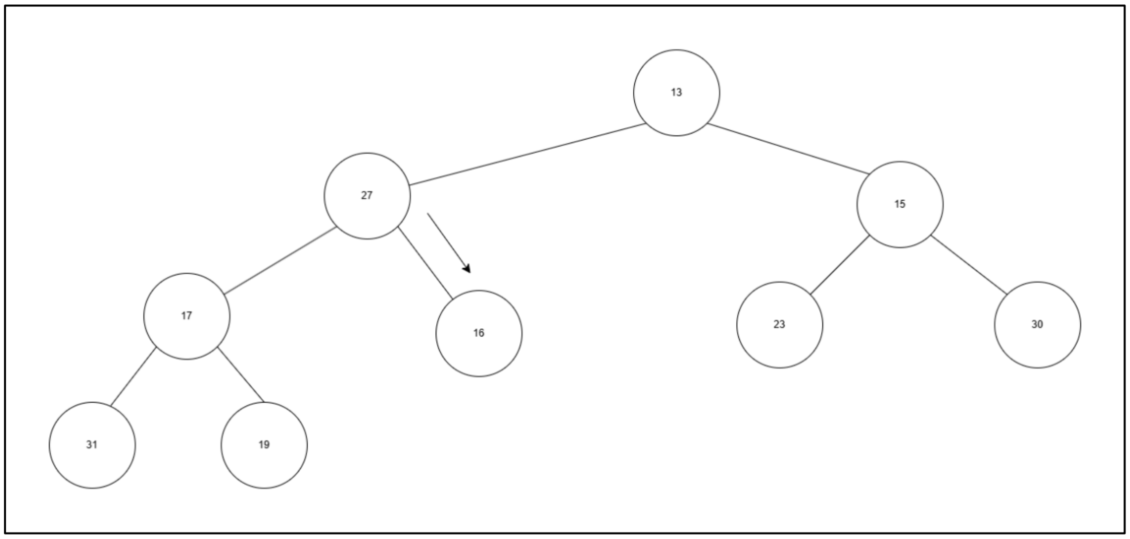
  We continue deleting the min element which is 11 and replace it with the 27. The array becomes : [27, 13, 15, 17, 16, 23, 30, 31, 19, 11, 9, 7, 5, 4, 2]



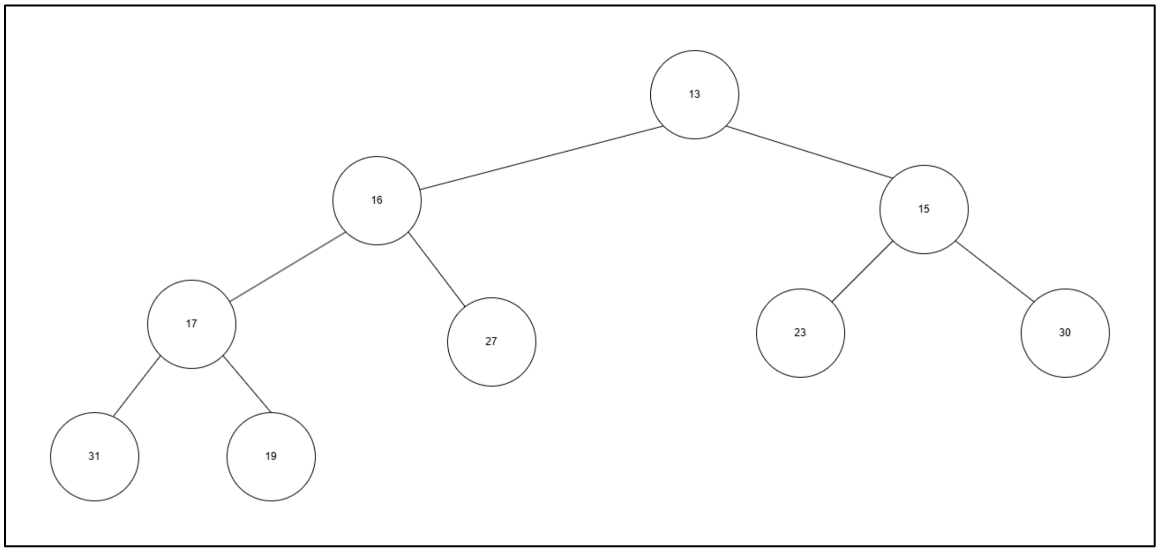
Next, we compare 27(parent) with 13 and 15. Since 13 is the smaller value, we swap it with the 27 (parent)



The array becomes: [13, 27, 15, 17, 16, 23, 30, 31, 19], since the array does not in min heap condition, we need to reheapify again at index 1, swap the value 27 with 16.

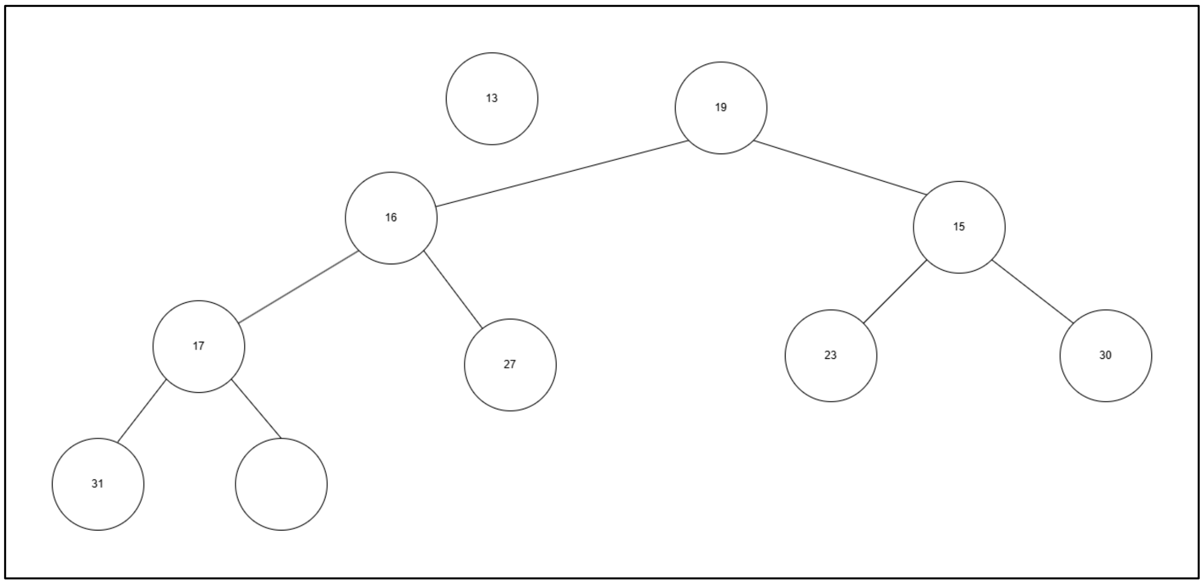


The result after reheapify : [13, 16, 15, 17, 27, 23, 30, 31, 19]

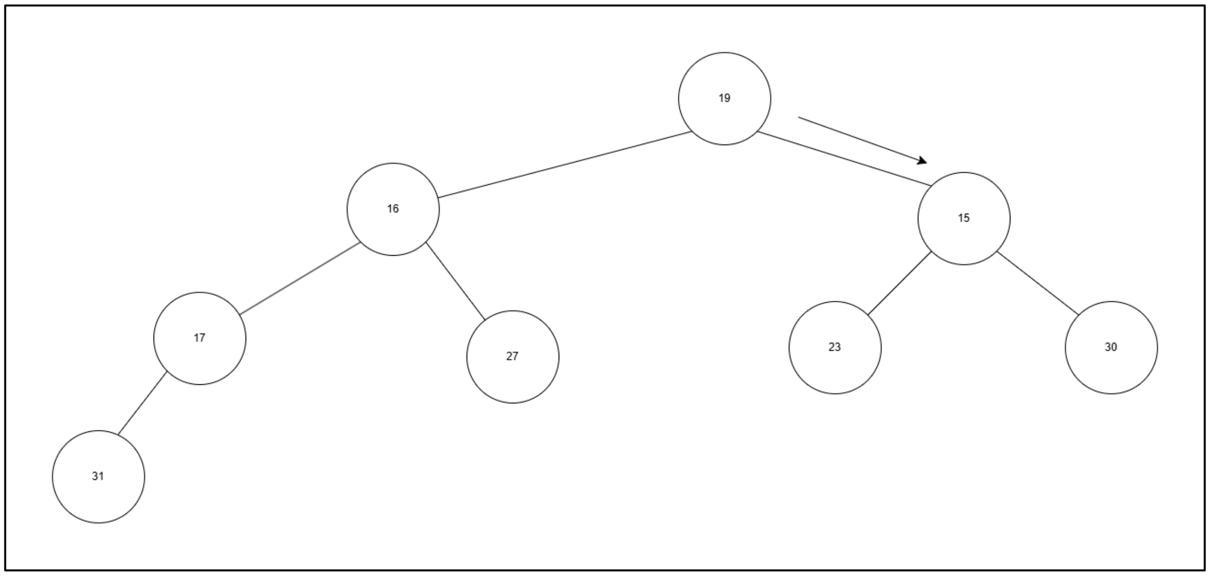


- **7th extraction**

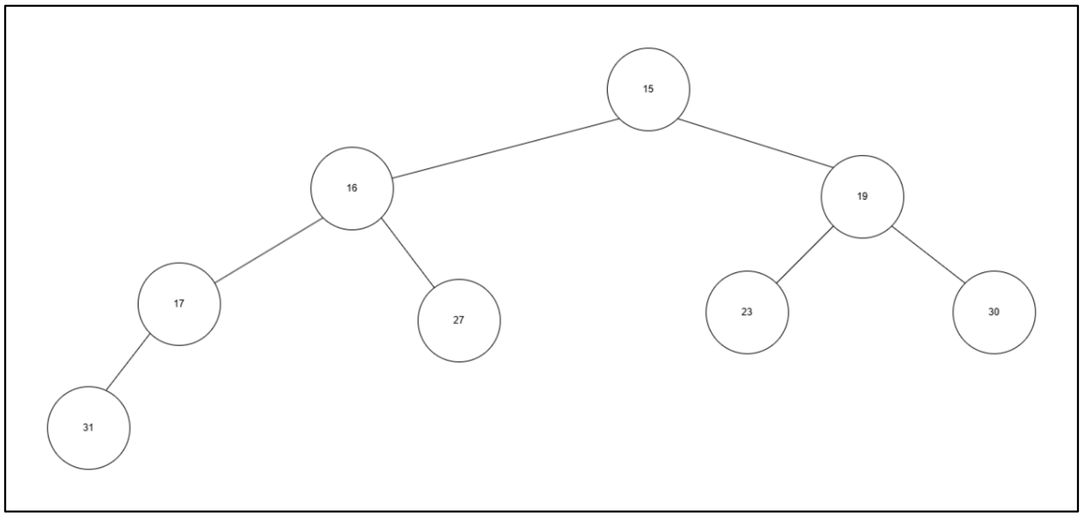
  We continue deleting the root, which is 13, and replace it with 19. The array becomes : [19, 16, 15, 17, 27, 23, 30, 31, 13, 11, 9, 7, 5, 4, 2]



Next, reheapify at index[0] = 19 with 16 and 15. Swap 15 with 19

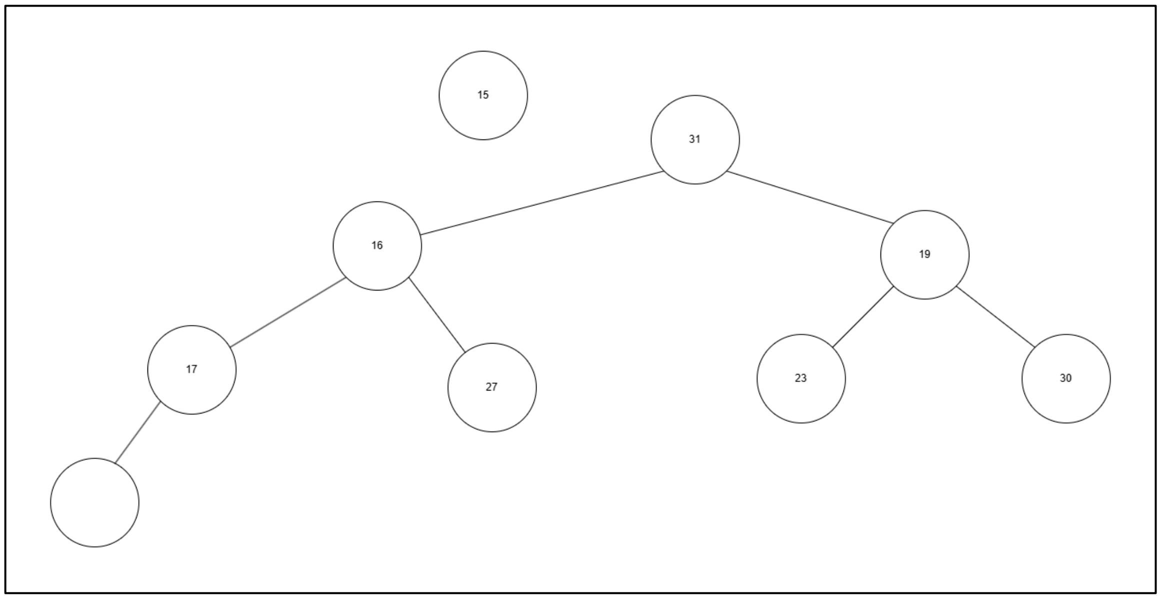


The array : [15, 16, 19, 17, 27,23, 30, 31]

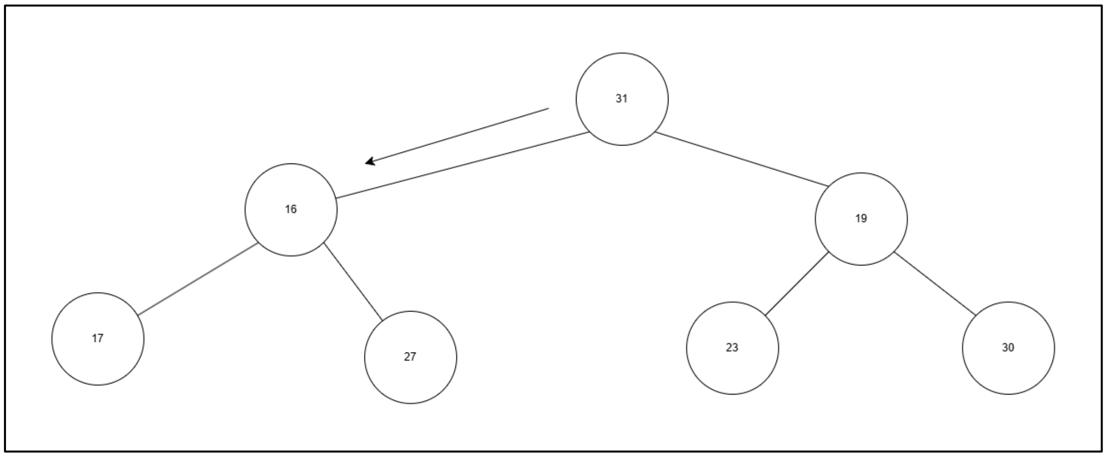


- **8th extraction**

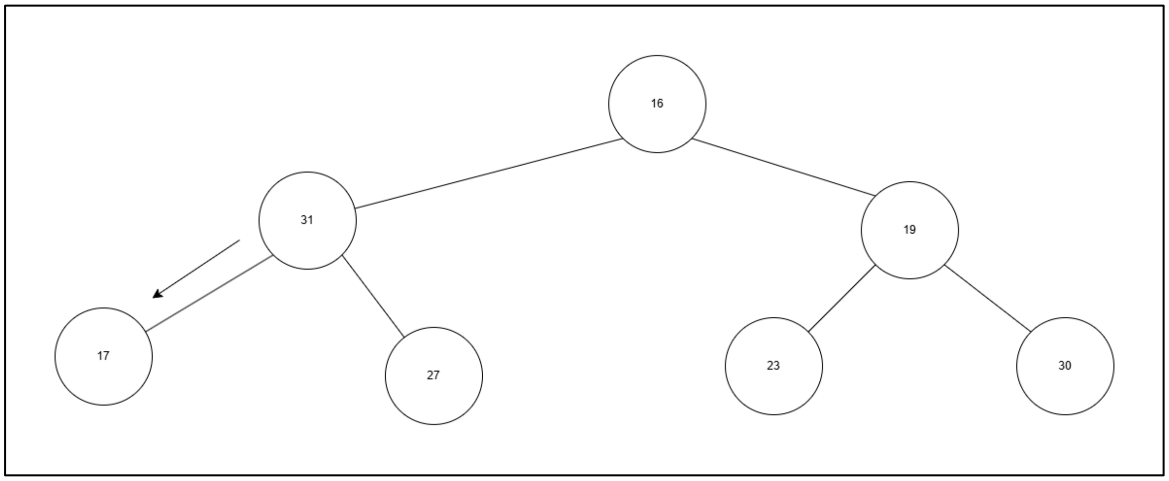
Next, we delete the root 15 and move the last element 31 to the root.



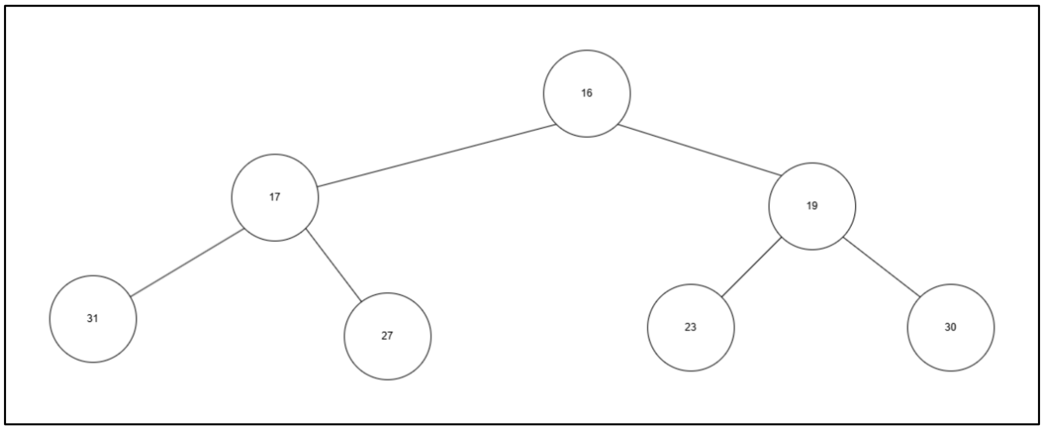
Then,we reheapify 31(parent) with 16 and 19, then we swap 16 and 31 since 16 is the smallest.



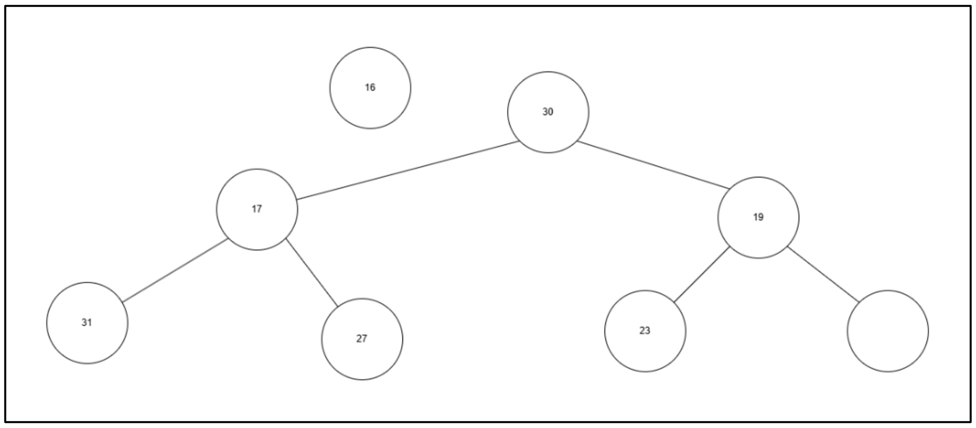
The result after reheapify : [16, 31, 19, 17, 27, 23, 30]. Then, compare the parent 31 with 17 and 27.



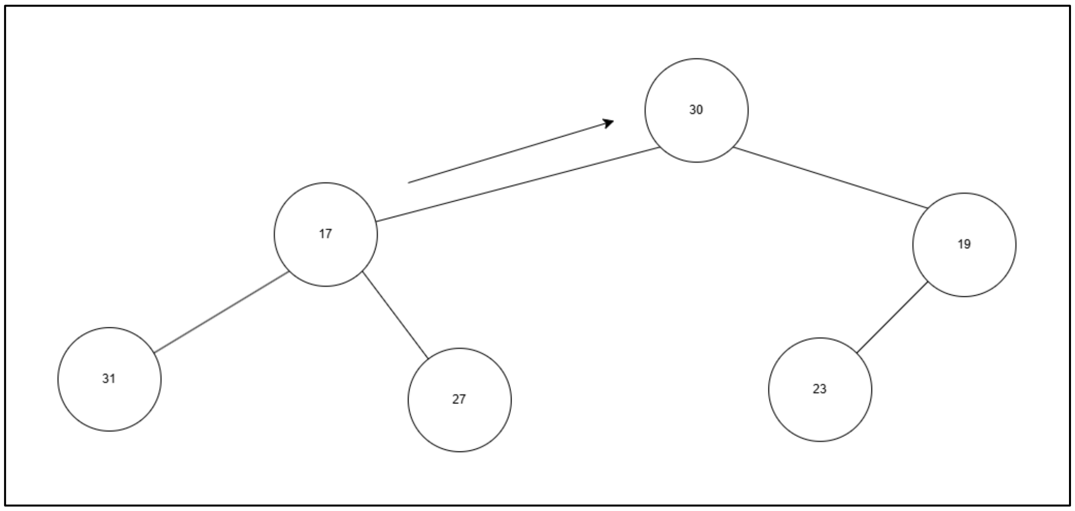
The result after reheapify : [16, 17, 19, 31, 27, 23,30]



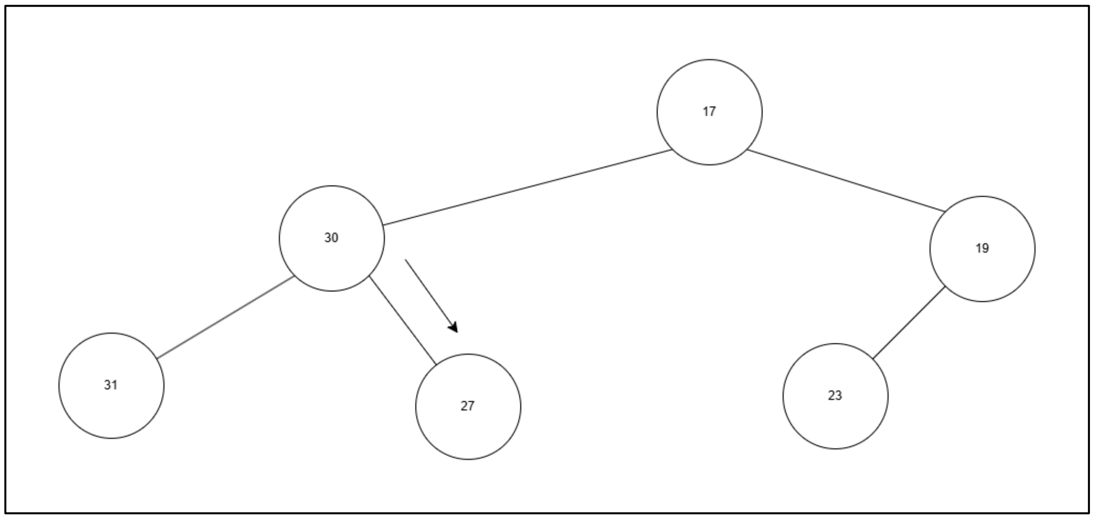
- **9th extraction**

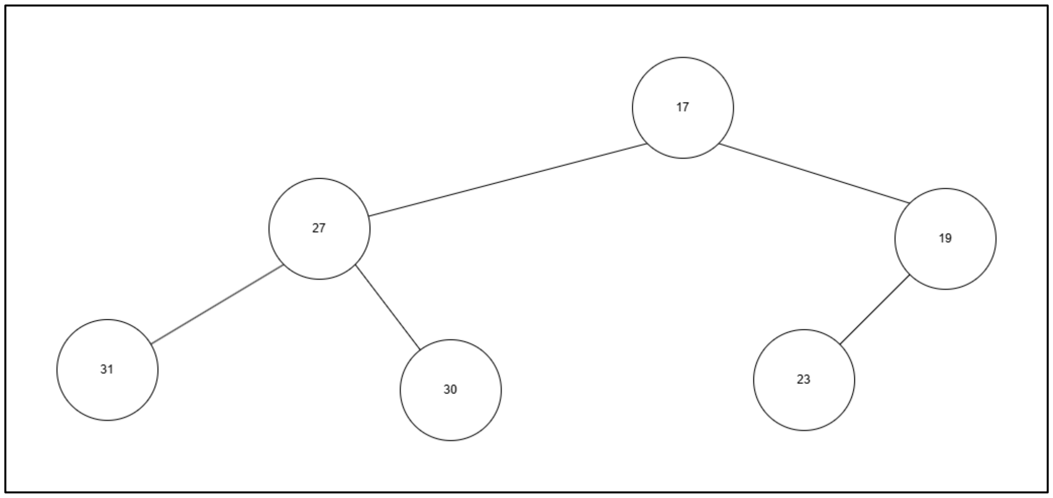


Next, we compare arr[0]=30 with arr[1]=17 and arr[2]=19. Swap 17 with 30

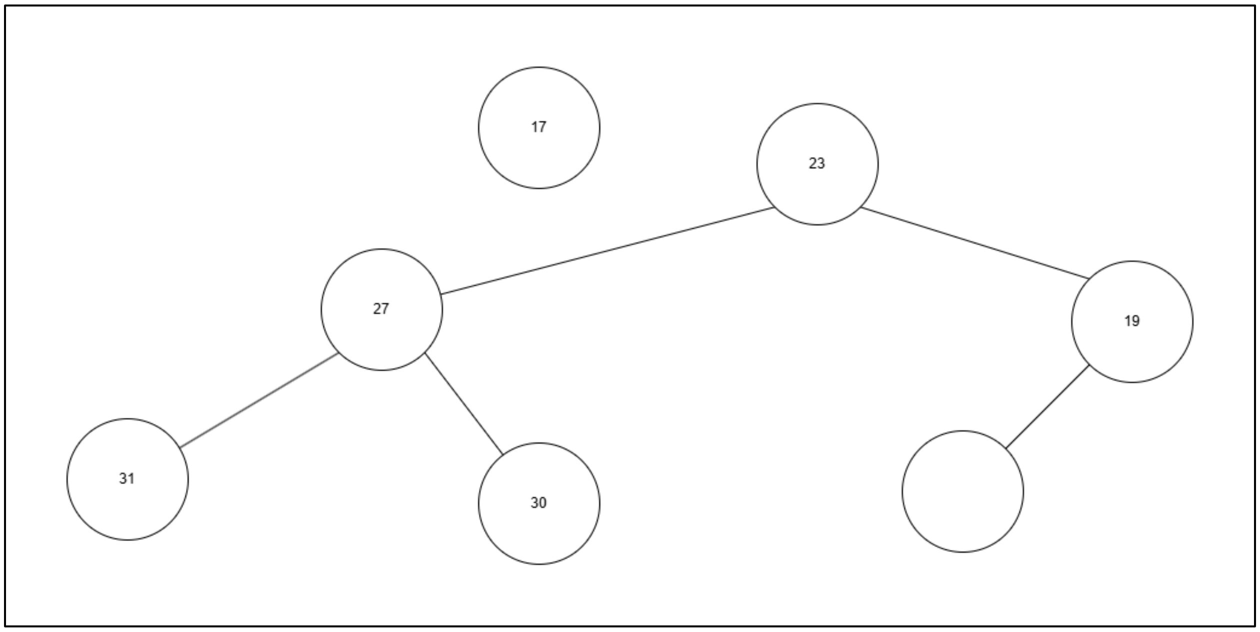


The result shown in the figure below. Next, we compapre arr[1]= 30 with31 and 27. Then, we swap 27 and 30 since it is the smallest.

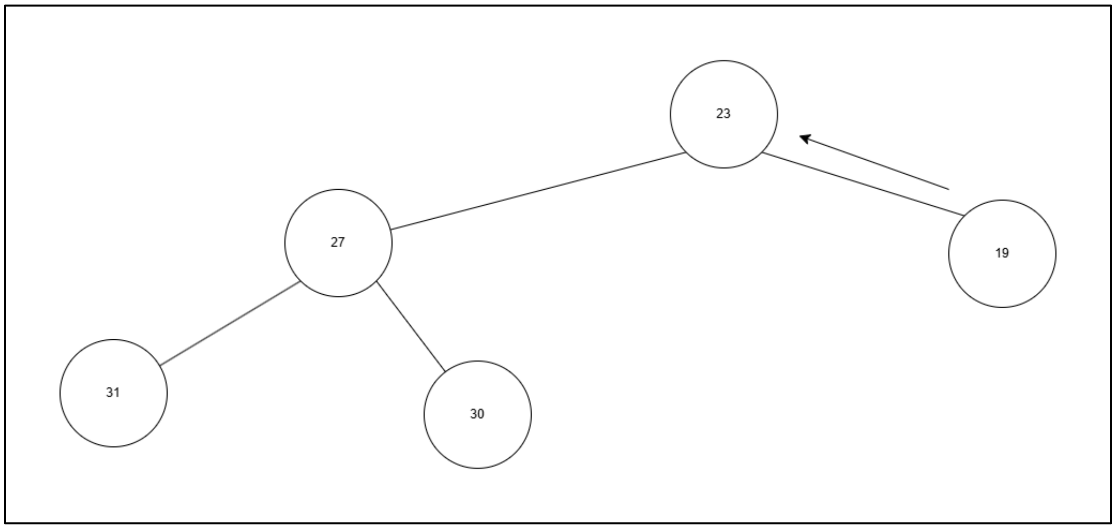




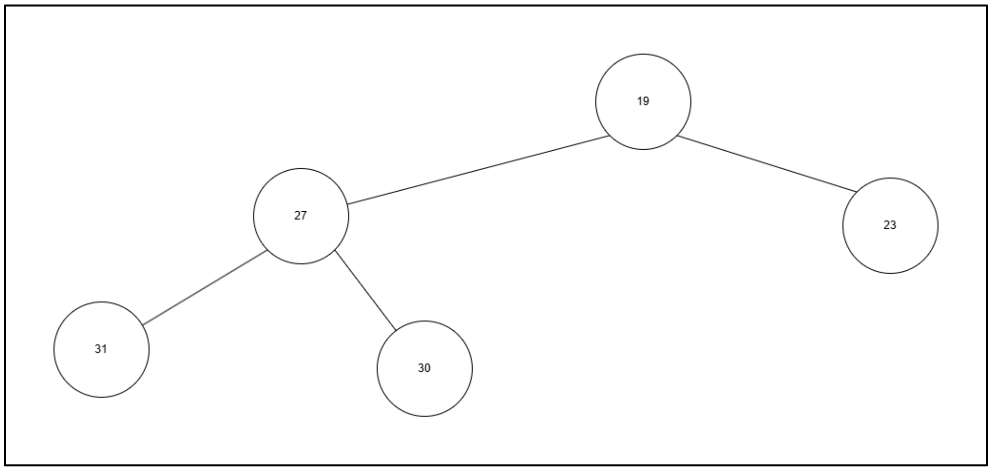
- **10th extraction**
  Then, we continue to sort by deleting 17 and replacing it with 23.



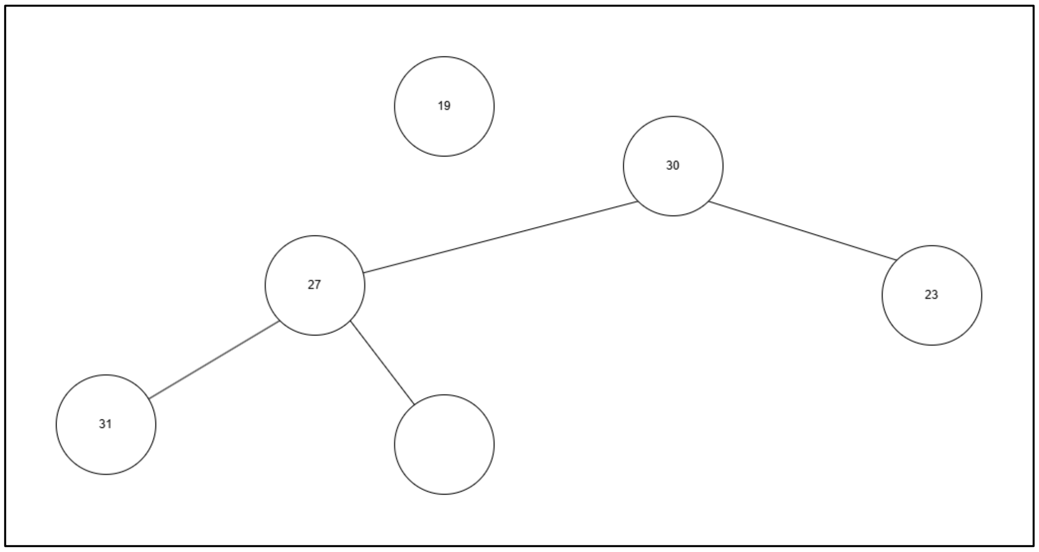
Reheapify : Swap 19 with 23



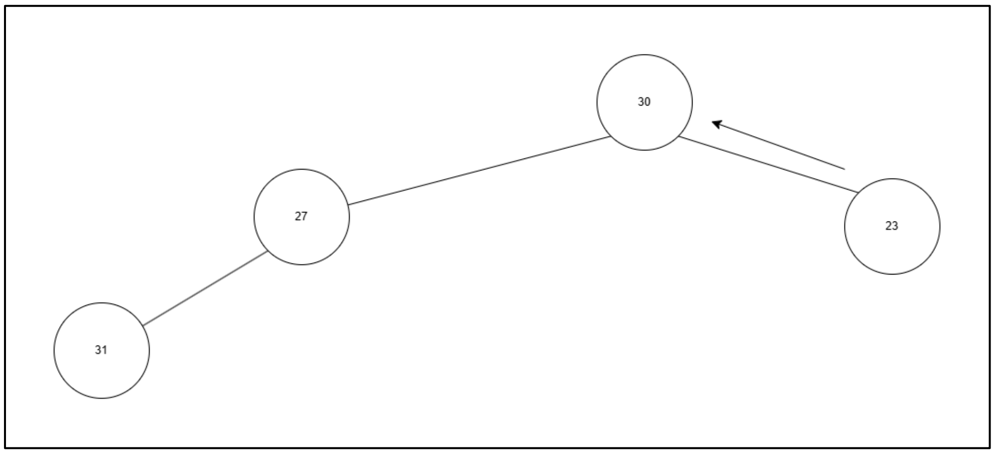
The result : [19, 27, 23, 31, 30]



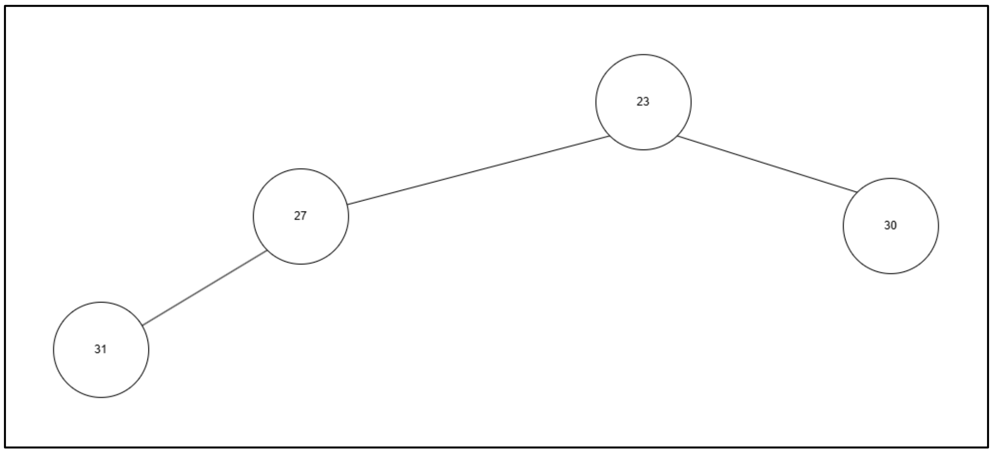
- **11th extraction**
  Next, sorting the tree by deleting 19 and replacing 30 at the root. Store the deleted element in the last empty array.



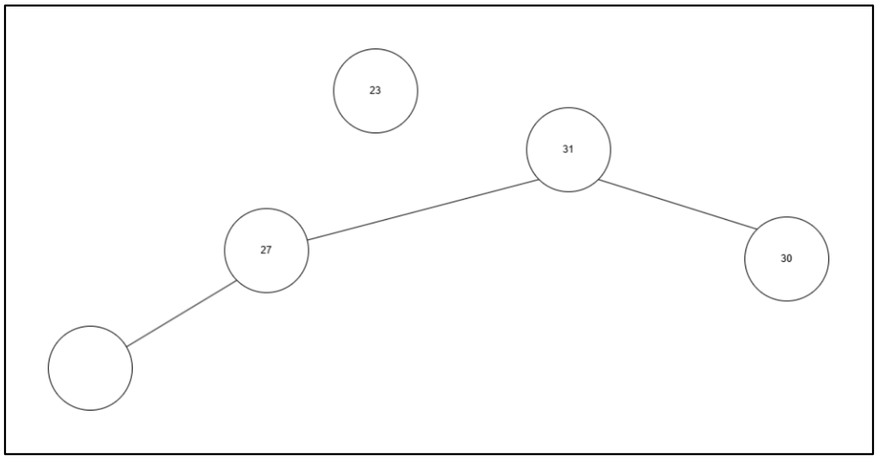
Next, we reheapify at index[0] since 23 is smaller than 30, swap it.



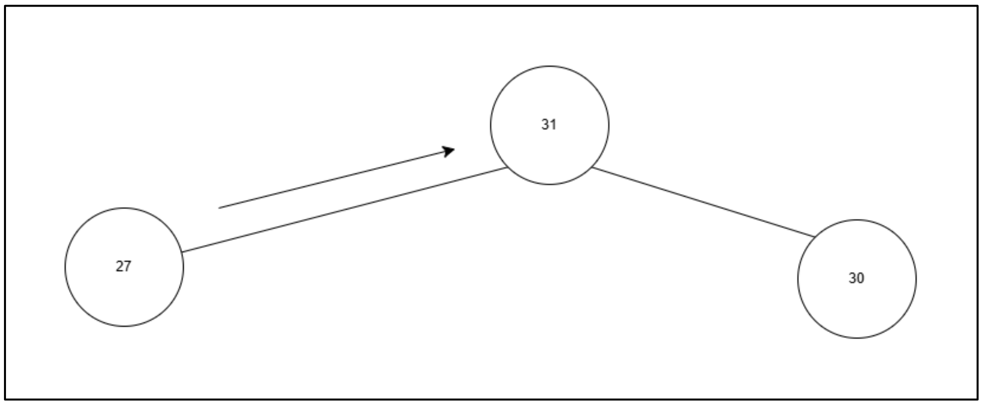
The final array after reheapify is : [23, 27, 30, 31]



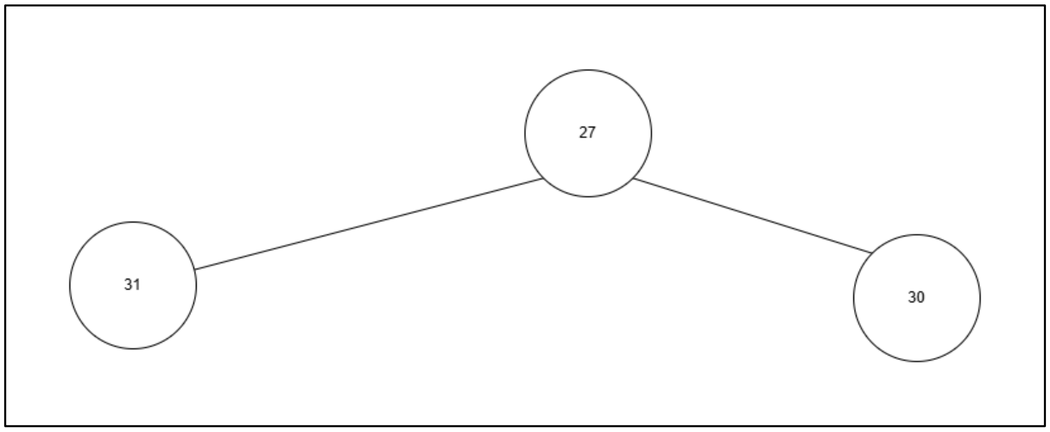
- **12th extraction**
  Then, we continue to extract the min element, which is 23 and and replace it with the last element 31.



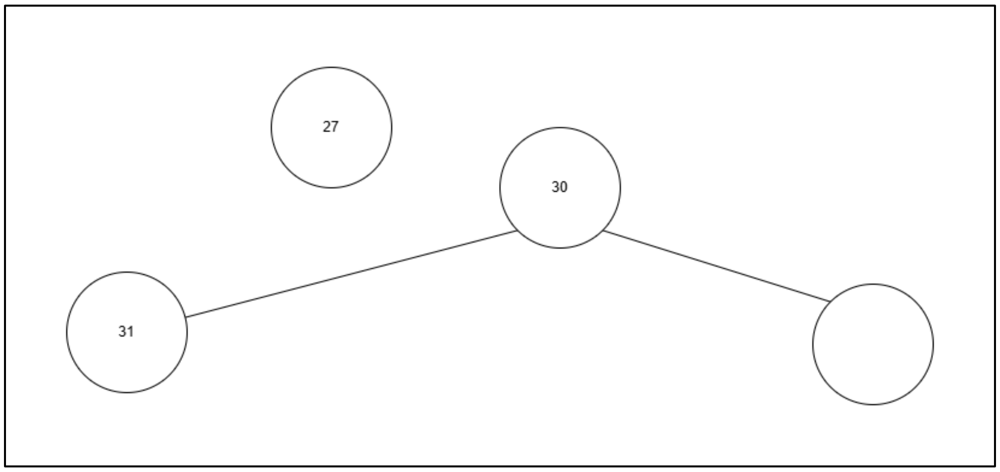
Then, we reheapify. Swap the value 27 with 31 since 27< 31.



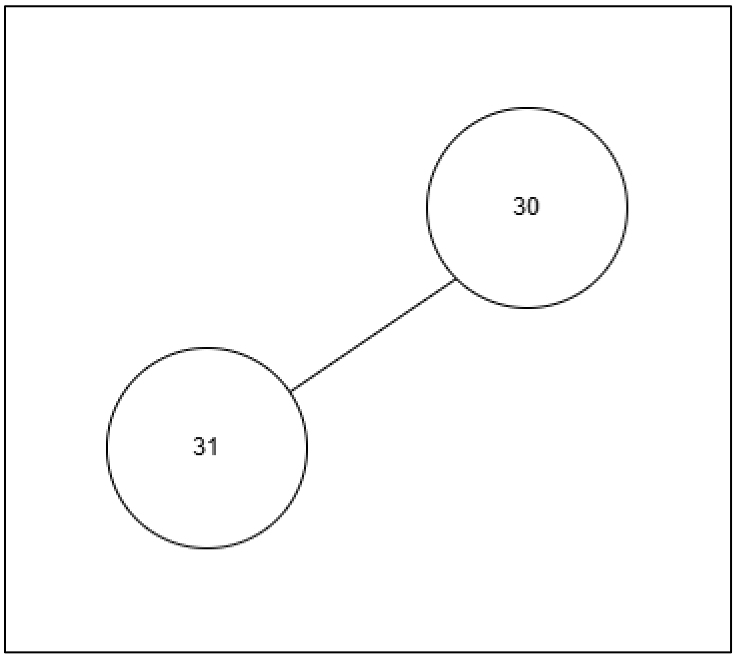
The final result : [27, 31, 30]



- **13th extraction**
  We delete the 27 and replace it with the last element, which is 30

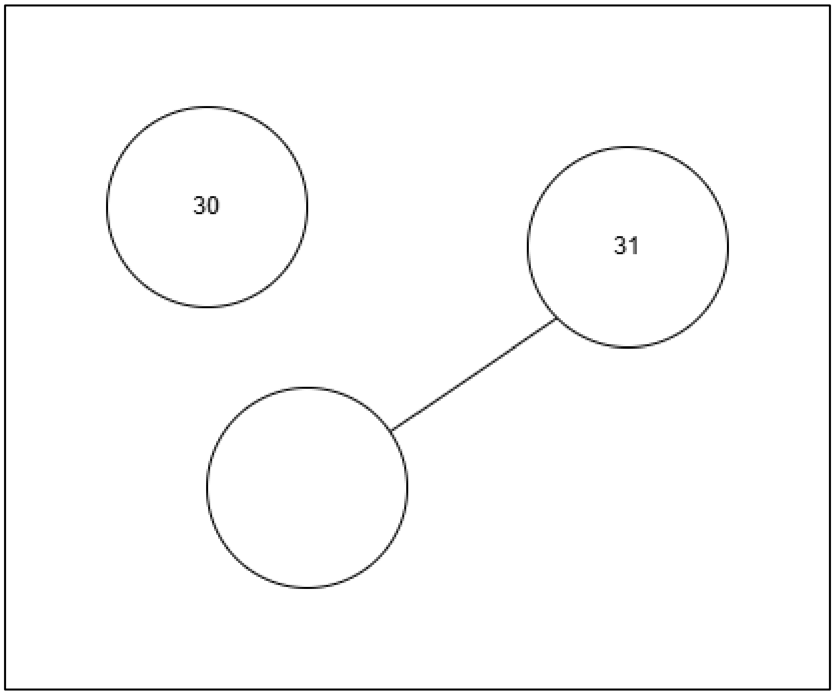


The final result : [30, 31]



- **14th extraction**
  Next, we delete the min element 30 and swap the value with the last element 31. Store the deleted value in the last empty array.

  Array : [31, 30, 27, 23, 19, 17, 16, 15, 13, 11, 9, 7, 5, 4, 2]



Then, the final array after sorting are :

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
31	30	27	23	19	17	16	15	13	11	9	7	5	4	2

Binary Search:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
31	30	27	23	19	17	16	15	13	11	9	7	5	4	2

low = 0, high = 14

Then, we need to find the middle element using the formula mid = (low + high) / 2. So, here array (0 + 14) / 2 = 7. So the array[mid] is located at i=7, which is 15. Then, we split into 2 arrays. Since the target = 13, array[mid] = 15 and the array is descending 13 < 15 , we search the right half is :

[8]	[9]	[10]	[11]	[12]	[13]	[14]
13	11	9	7	5	4	2

Then, the new low is 8.

Then, we recalculate the midpoint. Low= 8, high = 14.

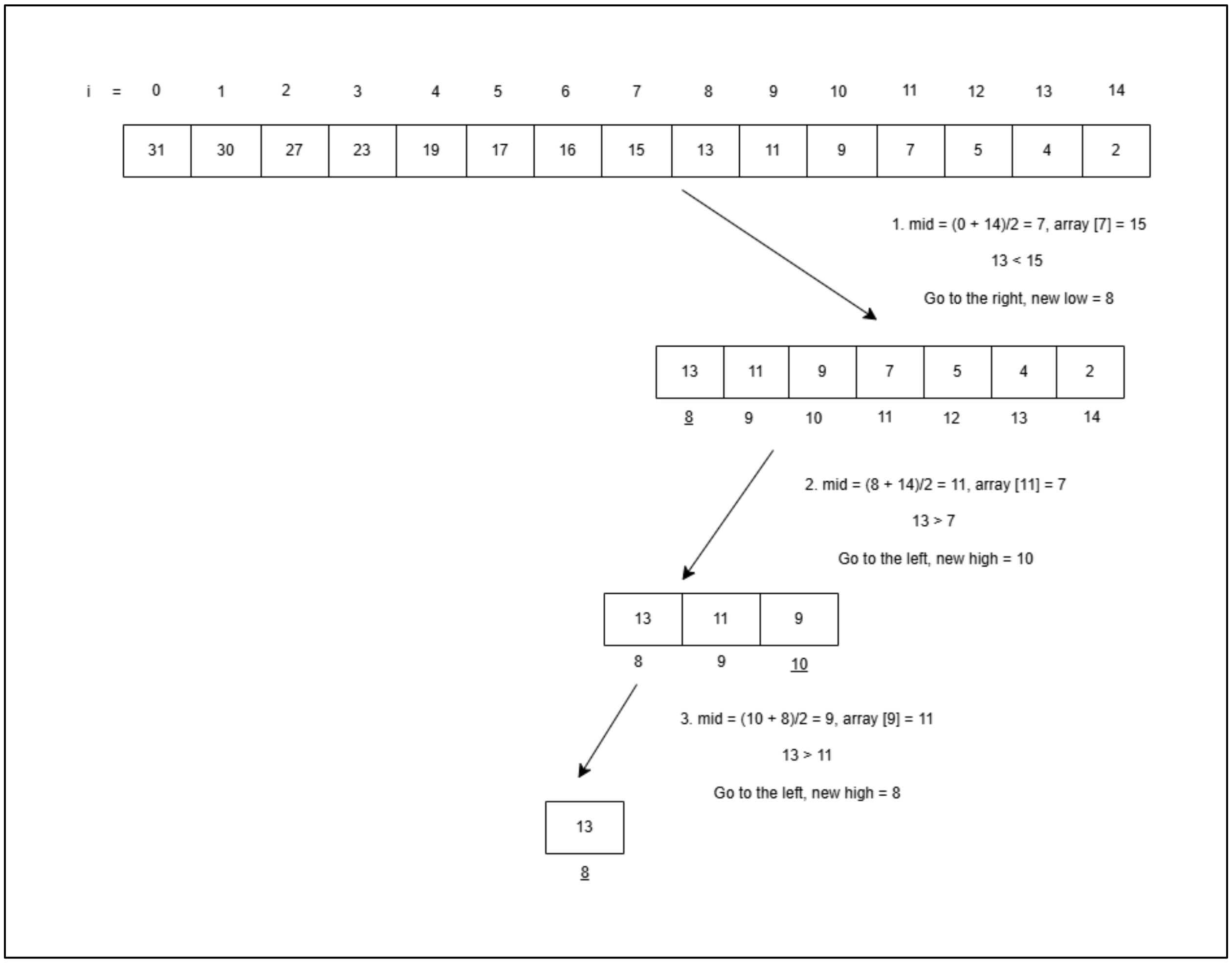
Mid = (8 + 14)/2 = 11

Array[11]= 7

Then, we compare 13 > 7, we search the left half. Then the new high is 10.

[8]	[9]	[10]
13	11	9

Then, we recalculate the midpoint low = 8, high= 10. So we got mid[9]= 11. Since 13 > 11, we search the left half. The new high is 8. Then, the new midpoint is low=8 and high=8 , the mid = 8 which is mid[8]= 13. Since 13=13. Then the key is found at index i=8.


i = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
31 30 27 23 19 17 16 15 13 11 9 7 5 4 2
1. mid = (0 + 14)/2 = 7, array [7] = 15
13 < 15
Go to the right, new low = 8
13 11 9 7 5 4 2
8 9 10 11 12 13 14
2. mid = (8 + 14)/2 = 11, array [11] = 7
13 > 7
Go to the left, new high = 10
13 11 9
8 9 10
3. mid = (10 + 8)/2 = 9, array [9] = 11
13 > 11
Go to the left, new high = 8
13
8

QUESTION 4

Time Complexity and Performance Evaluation: Conduct a comparative analysis between the Heap Sort and Insertion Sort algorithms. Explain their respective time complexities — Heap Sort with $O(n \log n)$ and Insertion Sort with $O(n^2)$ in the worst case and $O(n)$ in the best case — and evaluate their performance in handling product data in e-commerce environments. Discuss scenarios where each algorithm is most effective, considering factors such as dataset size, data order, and system efficiency

➢ Sorting algorithm are fundamental to modern application of computing, particularly in-ecommerce portals are related application, where huge volumes of product records must be sorted for searching, filtering and displaying to the visitors. Heap sort and insertion sort are 2 commonly used sorting algirthms with different advantages and disadvantages due to their structure and time complexity as shown in the table 1 below.

Aspect	**Heap Sort**	**Insertion Sort**
Best Case	O(n log n)	O(n)
Average Case	O(n log n)	O(n^2)
Worst Case	O(n log n)	O(n^2)
Space Complexity	O(1)	O(1)
Stability	No	Yes
Suitability	Fast for large dataset	Efficient for smaller dataset
Use Case	System that prioritize performance	Real-time system that require simplicity

Heap Sort

1. Best Case O (n log n)
    - When we apply heap sort, we always build the heap and then we do n extract for maximum and minimum value in log n comparisons and swaps because of the nature of the heap.
    - Heap sort will not take advantage of the already sorted data, even if it exist,
    - O(n) Time to build the heap O(n) + Time spent each time to extract the root and re-heapify O (log n) each of the n elements
    - Therefore, T(n) = O(n) + O(n log n) = O (n log n)
    - Consistently O(n log n) in worst case and average case

Insertion Sort

1. Best case O(n)

- For best case it occurs when the data are already in ascending order (sorted data)
- This is done once for each element compared to the previous element.
- And since there is not shifting, each iteration is O(1)
- Total time : n-1 comparisons -> O(n)

2. Average Case O(n2)
   - When the data is random
   - For average, each element needs to be compared and shifted approximately half way through the already sorted side
   - Total comparisons and shifts = n(n-1)/4 = O(n2)

3. Worst Case O(n2)
   - When the data is sorted in reverse order or few unique
   - So every new element needs to be compared & shited through all previous elements.
   - Number of comparisons and shifts in total : O(n2) because 1 + 2 + 3 + … + (n - 1) = n (n - 1) / 2 = O (n2)

Performance In E- Commerce Environments

Whereas E-commerse platforms, deals with a huge product data related to attributes like prices, ratings, names, categories, availability, etc. The speed and efficiency of the sorting mechanism plays a vital role in determining the loading time, user experience, and server performance. Therefore, it requires algorithms that are not only fast but reliable as the data grows.

For Heap Sort, in e-commerce they have features : order fulfillment, shipment queuing and many more tasks often depend on priority levels such as urgent orders. Therefore, heap sort is based on the heap structure (priority queue) and as such provides fast access to high priority items in O(1) and in O(log n) update.
Next, e-commerce platform can be responsible for thousands to million of products and therefore, heap sort guarantee O)n log n) performance regardless of the input ordering. So it is excellent for very large datasets unlike Insertion Sort which becomes O(n2) in the worst case.

Moreover, sorting need to be refreshed quickly and consistently when product prices or inventory level changes whenever they have a flash sales. Because the behavior of insertion sort is not consistent, heap sort lends itself far better to real-time sorting algorithm as it has predictable performance, and its good at reordering items.

Feature	Heap Sort O (n log n)	Insertion Sort

Performance for large dataset	Consistent and reliable	Degrades significantly
Dataset Size	Large (N > 100)	Small (N<100)
Real time updates	Predictable and fast	Inef
Priority based order processing	Built in priority queue support	Needs extra logic
Nearly Sorted data / nearly sorted data	Overkill	Very efficient O(n)
Dynamix updates	Operations in-place	Struglle with frequent

From this table, heap sort is much better suited for scalable, high performance requirements with consistent O (n log n) performance. In e-commerce platforms especially when data is larger or unordered, require real time and operations need to run in low memory which the heap sort use minimal memory where it operates in-place, minimizing memory overhead during high traffic. While insertion sort has low overhead for lightweight tasks but inefficient under heavy load.
Moreover, heap sort does bot affected by initial order data order, leading to reliable for unsorted or random elements while insertion sort efficient when the data is already 70-90% sorted. Insertion sort is easy, simple and has great time efficiency on small or already mostly sorted data but it is not best used for workers with small batch data for administrative tools and teaching or conceptual demos.

Performance analysis of heap sort and insertion sort.

Here, I evaluate the efficiency of heap sort and insertion sort by having datasets (3000, 6000, and 9000) elements. The COU time were recorded in the table below.

Dataset	Heap Sort	Insertion Sort
3000	0.000405	0.004029
6000	0.000910	0.016568
9000	0.001426	0.035383

This table shows clearly that heap sort is suitable for large datasets as it increases. Heap sort maintains consistent performance due to its time complexity O(n log n), making it well-suited for large and unsorted data. For insertion sort, we can see that the execution time increases as the input increases. This is because O(n2) time complexity in the average and worst cases, where each element needs to be compared and shifted multiple times, where it tooks a longer time to execute.

Therefore, in e-commerce environments, where large data needs to be sorted in real time for displaying, filtering, or searching, heap sort is the suitable algorithm.

QUESTION 5

Critical Evaluation: Discuss the strengths, limitations, and potential trade-offs of the Heap Sort algorithm for solving this problem.

✧ **Strength of Heap Sort**

Features	**Description**
Time complexity	Throughout all scenarios, Heap Sort maintains performance at O(n log n). This property, for example, benefits UMPSA’s real-time scoreboard, where students’ scores get updated continuously. With such a system, the responsiveness remains consistently high even with thousands of simultaneous updates.
In Place Sorting	Uses minimal memory as it sorts the elements in place and only uses a constant amount of additional memory, O(1)/
Efficient for dynamic updates	Can handle real-time score updates, and inserting a new score into a max heap takes O (n log n) time, which provides quick adjustments to the leader board.
Stability	Since leaderboard data can grow large (e.g., many participants), Heap Sort can efficiently manage sorting without degrading system performance, ensuring that the updated leaderboard is always ready for public display.
Suitable for a priority queue	Heapsort is derived from a heap data structure, which makes it useful in real-time applications, such as data scheduling, where it is sorted by serving the highest priority first.
Compatible with Binary Search	After the data is sorted, binary search O (nlog n) can quickly locate scores which can be

	reducing lookup latency.

✧ **Limitations of Heap Sort**

Features	Description
Lack of stability	Was described earlier. Heap Sort does not keep the relative order of students with duplicate scores. This means two students tied on the scoreboard can be placed out of order as to who reached the position first, without extra mechanisms being added, in which case more complexity is incurred.
Less Intuitive Debugging and Maintenance	Managing the heap property correctly poses a significantly higher challenge than simpler algorithms like Insertion Sort. In mission-critical systems (for example, real-time scoreboards), more intricate algorithms that have not been thoroughly tested can raise the risk of bugs if proper care is not taken for the maintenance and testing of algorithms post-deployment.
Overhead in Frequent Updates	n algorithms like Heap Sort, re-heapifying the structure takes place with every insertion or update of the score. In the case of a very high frequency of updates (like hundreds per second), the system can face small but over time accumulating lags when compared to more dynamic data structures like balanced binary search trees.

Potential trade off

Efficiency vs simplicity	Heap Sort provides comparison-based sorting well-suited for parallel execution, but it is more difficult to implement and maintain than simpler sorting methods. The technical team has to ensure they can ship updates quickly while also maintaining a system that can be debugged and easily updated.
Real-time responsiveness vs stability	The leaderboard will be reordered quickly with the Heap Sort, and with its instability, be cautious if students having the same score exchange positions. It would also have been necessary more coding to maintain submission order.
Memory optimization vs update speed	Heap Sort is highly efficient and requires very little extra memory, which makes it excellent for system programming. quicker dynamic updates are possible using other data structures (e.g. AVL trees or Fibonacci heaps) at the expense of slightly more memory use.

In the UMPSA mega online quiz competition, we may consider to use Heap Sort to keep track of a large and dynamic leaderboard in an efficient way due to its predictable O (n log n) performance, low memory overhead, and good worst-case behaviour. But trade-offs are the complexity of maintaining the heap, possible instability in ordering, and slight inefficiency when the updates are very frequent. These trade-offs should be given consideration for an optimal system performance in real time.

QUESTION 6

Algorithm Implementation: Construct the code to implement the Heap Sort algorithm with the binary searching algorithm. Then, execute the code on datasets containing 10 000, 20 000, 30 000, 40 000, 50 000, 60 000, 70 000, 80 000, 90 000, 100 000 points.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to heapify a subtree rooted with node i which is an index in arr[]
void heapify(int arr[], int n, int i)
{
    int largest = i;     // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i)
    {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

```
// Function to perform heap sort
void heapSort(int arr[], int n)
{
    int i;
    clock_t st = clock();

    // Build heap (rearrange array)
    for (i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (i = n - 1; i > 0; i--)
    {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }

    clock_t et = clock();
    double cpu = ((double)(et - st)) / CLOCKS_PER_SEC;

    printf("Heap Sort CPU time used: %f seconds\n", cpu);
}

// Binary Search function
int binarySearch(int arr[], int n, int key)
{
    int low = 0, high = n - 1;
    while (low <= high)
    {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // Element not found
}
```

```
int main()
{
    int n_sizes[] = {10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000,
100000};
    int num_sizes = sizeof(n_sizes) / sizeof(n_sizes[0]);
    int i, j;

    for (i = 0; i < num_sizes; i++)
    {
        int n = n_sizes[i];
        int *arr = (int *)malloc(n * sizeof(int));

        // Fill array with random numbers
        for (j = 0; j < n; j++)
        {
            arr[j] = rand();
        }

        printf("\nSorting %d elements:\n", n);
        heapSort(arr, n);

        // Pick a random element to search
        int key = arr[rand() % n];
        clock_t st_search = clock();
        int result = binarySearch(arr, n, key);
        clock_t et_search = clock();

        if (result != -1)
            printf("Element %d found at index %d.\n", key, result);
        else
            printf("Element %d not found.\n", key);

        double cpu_search = ((double)(et_search - st_search)) / CLOCKS_PER_SEC;
        printf("Binary Search CPU time used: %f seconds\n", cpu_search);

        free(arr);
    }

    return 0;
}
```

Output

DataSet Size	Heap Sort Time	Binary Search Time	Search Result
10,000	0.001627	0.000001	7537
20,000	0.003832	0.000001	6831
30,000	0.006169	0.000001	14442
40,000	0.007840	0.000000	716
50,000	0.010112	0.000001	44878
60,000	0.010978	0.000001	26716
70,000	0.013512	0.000000	1747
80,000	0.016660	0.000001	56585
90,000	0.017386	0.000001	81156
100,000	0.019894	0.000002	97234

Observation

From this table, we can see that the heap sort time increases as the dataset growing. It consistent with the expected O (n log n) time complexity and approximately linear relative to dataset size. For example, at 10,000 the heap sort time is 0.001627 and when the date data is 100000 the time increases to 0.0199 seconds. Therefore, this prove that the heap sort is suitable for large data datasets and be sorted quickly as the system must always display the top performers sorted order their scores. Moreover, the binary search time complexity is consistent between 0.000001 to 0.000002 seconds for all the size of the dataset. This proves that the time complexity for binary search O(n log n) and the searching after heap sorting is very quick. Therefore, combination of heap sort and binary search is highly efficient for real-time applications that needs quick soring and real time update.

QUESTION 7

Real-World Applications: State and thoroughly elaborate on five real-world applications of the Insertion (TWO applications) and Heap Sort (THREE applications) algorithm.

1. Real-time leaderboard maintenance (insertion)

Schools and colleges need to maintain a daily attendance register and reports of students. List of attendees tends to get modified throughout the day, and is generally small and already mostly sorted. A few students that are late, or some tard changes added at the last minute. For this situation, insertion sort is much better. The majority of the attendance list data is already sorted (students check in sequentially), which results in the best-case of insertion sort O(n). When the straggler arrives, the system just takes the name and stuffs it into the right spot rather than resorting the whole damn thing again. By this method processing time is less, accommodating system resources, and the attendance record is immediately updated.

2. Card Game Application

When you play a digital card game, such as poker or a solitaire app, you have to be able to quickly and easily sort out a small hand of cards (say, 5 to 7 cards). Cards are usually drawn at 1 (or more) intervals and are already partially sorted. Insertion sort is a good choice for small data and nearly sorted data. It arranges the playing cards in a hand by adding each card in its proper place among those already drawn. This allows instant sorting with very low CPU cost. It is great for real-time card games needing a good and smooth UX. Touch card gathering and an independent display card, the card won't feel any delay.

3. Real time e-commerce leaderboard

In an e-commerce flash sale (e.g, 1.1 sales), a platform should maintain a real-time leaderboard of high spenders or fast purchasers to foster competition among customers. To manage these real-time ranking systems, for instance, through leaderboard lists, it is possible to use the Heap Sort strategy with its O(n log n) complexity to have a max-heap with the user's spending totals. The total score of a user is updated whenever the user makes a purchase, and the max-heap ensures that the top buyer can easily be accessed. This feature means that the leaderboard can be updated instantly, without full re-sorting, providing you with a smoother and more dynamic experience that engages your users during huge sale events.

4. Hospital Emergency Room (heap)

In the emergency room, you are treated according to how sick you are, not simply what order your arrival. The conditions of patients can change for the worse, too, leading to a rapid re-prioritization. In the case of heap sort, the priority queue could establish the severity score by which each patient is weighted. With a ma-heap, the most serious patient is at the top, so doctors can immediately take action. New patients can be handled by entering a time of order in O(log n), so we can make rapid and accurate decisions under a stressful situation.

5. Database File Sorting

Large systems such as banking databases, social media platforms, and e-commerce catalogs frequently encounter a problem with very large files that don't fit into memory, and where sorting all the data to extract partial results is required. Heap sort is especially appropriate for use in external sort since it can achieve O(n log n) even in the worst case (regardless of order). For sorting files larger than what will fit into memory, heap sort splits the data into segments that fit into memory, sorts the segments independently, and then merges them, which can be done very efficiently. This is essential for sectors where data needs to be processed in a timely manner, to generate critical reporting for businesses or other organisations.

REFERENCES

Applications of heap sort - FasterCapital. (n.d.). FasterCapital. https://fastercapital.com/topics/applications-of-heap-sort.html

General | Algorithm | HeaP Sort | CodeCademy. (2024, November 18). Codecademy. https://www.codecademy.com/resources/docs/general/algorithm/heap-sort

Bot, B. (2024, April 19). *Everything you need to know about insertion sort Algorithm - Bomberbot*. Bomberbot. https://www.bomberbot.com/algorithms/everything-you-need-to-know-about-insertion-sort-algorithm/

Alhajri, K., Alsinan, W., & Alhmood, F. (2022). Analysis and Comparison of Sorting Algorithm. *Analysis and Comparison of Sorting Algorithms (Insertion, Merge, and Heap) Using Java*, *22*. https://doi.org/10.22937/IJCSNS.2022.22.12.25

Author. (2024, September 17). *Implementing real-time scoring updates for algorithm competitions*. peerdh.com. https://peerdh.com/blogs/programming-insights/implementing-real-time-scoring-updates-for-algorithm-competitions-1

Insertion Sort Algorithm | StudyTonight. (n.d.). https://www.studytonight.com/data-structures/insertion-sorting