



اونيورسيتي مليسيا قهغ السلطان عبدالله
UNIVERSITI MALAYSIA PAHANG
AL-SULTAN ABDULLAH

BCI2313

ALGORITHM AND COMPLEXITY

ASSIGNMENT 1

(Semester 2 2024/2025)


LECTURER NAME: AHMAD FAKHRI BIN AB. NASIR

NAME : AIDID QAYYUM BIN ZULKIFLI

MATRIC NUMBER : CB23093

SECTION : 02A

DATE OF SUBMISSION : 25 APRIL 2025

	SUBJECT: ALGORITHM & COMPLEXITY		MARKS: /100
	TOPIC: SORTING ALGORITHM	CODE: BCI 2313	
	ASSESSMENT: ASSIGNMENT (15%)	DURATION: 5 Hours	

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.

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.

- 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 one of the most basic sorting algorithms. It works in a way that is quite like how we sort playing cards in our hands. We go through the list one element at a time, comparing each new element to the ones before it, and inserting it in its correct place. The process continues until the whole list is sorted. In terms of time complexity, the best case occurs when the data is already nearly sorted. In that scenario, the algorithm only needs to perform a few comparisons, leading to a time complexity of $O(n)$. However, in the worst-case scenario when the data is in completely reverse order, it takes $O(n^2)$ time because every element has to be compared and shifted.

Algorithm Complexity:

Case	Time Complexity
Best	$O(n)$
Average	$O(n^2)$
Worst	$O(n^2)$
Space	$O(1)$

Heap Sort

Heap Sort is a method that uses a structure called a heap to sort data. A heap is like a special tree where the biggest number is always at the top (called a max heap). Heap Sort always works in $O(n \log n)$ time. That means it stays fast and efficient, even if the data is random or messy. This is better than Insertion Sort, which can become slow ($O(n^2)$) if the data isn't already nearly sorted.

Algorithm Complexity:

Case	Time Complexity
Best	$O(n \log n)$
Average	$O(n \log n)$
Worst	$O(n \log n)$
Space	$O(1)$

- 2) **Heap Sort Algorithm and Binary Search:** Analyze the process of the Heap Sort algorithm for assisting the binary search algorithm, including the two main steps of the heap sort algorithm.

Before we can use Binary Search, the data must be sorted. That is where Heap Sort comes in. Heap Sort helps by sorting the data in descending order. Once the scores are sorted, Binary Search can quickly find a specific score or user.

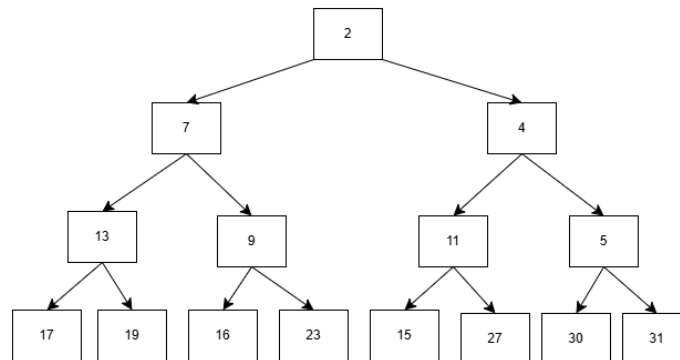
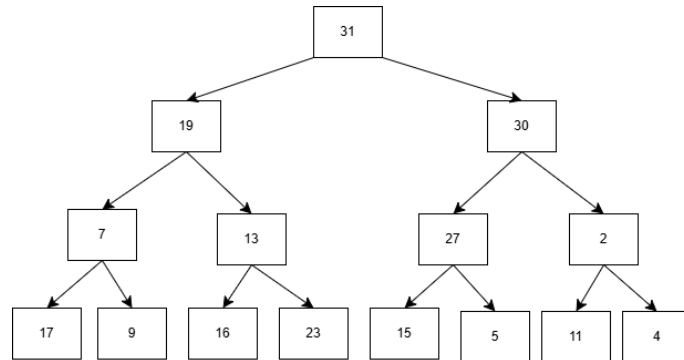
Heap Sort has two main steps:

- 1) Build a Max Heap
 - The algorithm arranges the array so that the largest value is at the top.
 - This step makes sure the parent is always bigger than its children.
- 2) Heapify and sort
 - After that, it swaps the top value with the last value in the array.
 - Then, it reduces the heap size and rebuilds the heap for the remaining data.
 - This process repeats until all data is sorted.

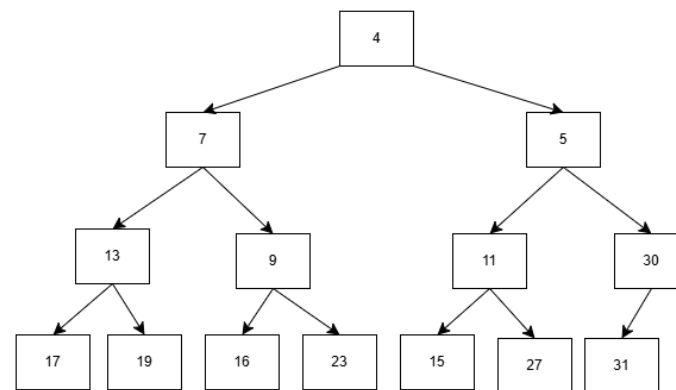
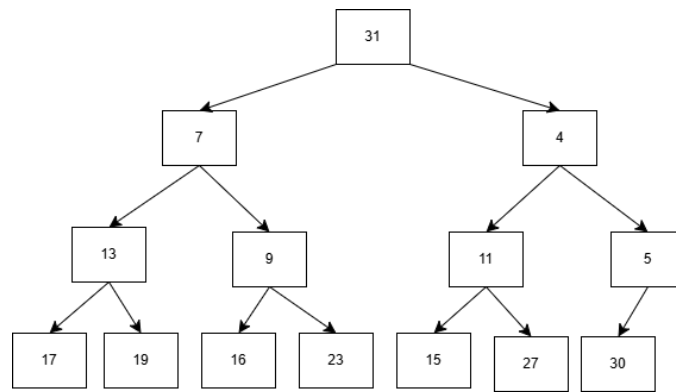
After the array is sorted, we can use Binary Search:

- First, it looks at the middle value.
- If the value we want is smaller or bigger, it continues searching only half of the array.
- It repeats this process until it finds the value or confirms it's not in the list.

- 3) **Solve the Problem:** Provide a detailed step-by-step solution how heap sort can assist the binary search problem using this array of number [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.

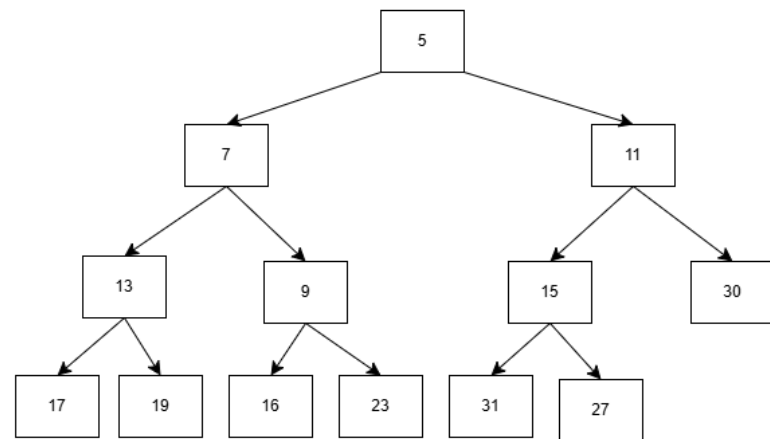
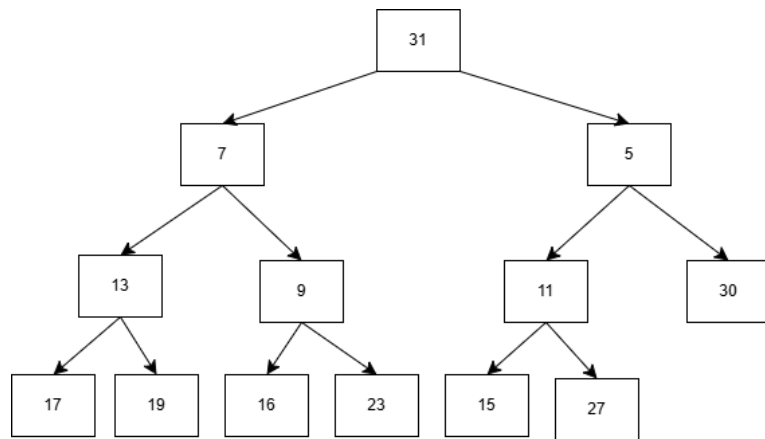


Iterative	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Value	31	13	5	17	9	11	4	7	19	16	23	15	27	30	2



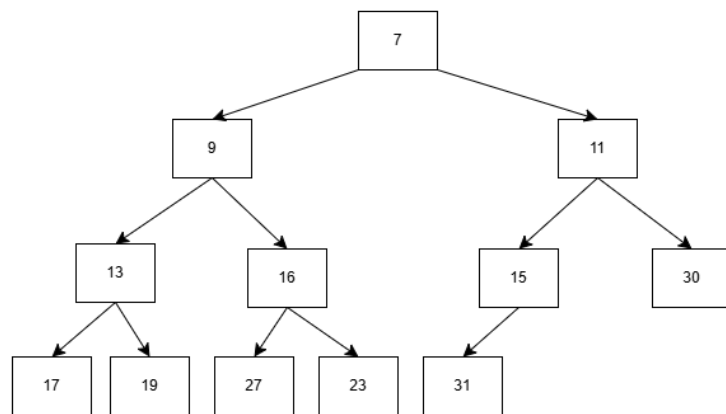
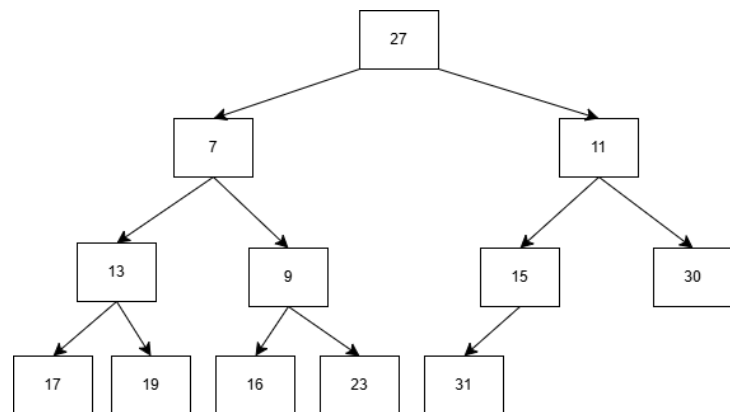
Iteration 1

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



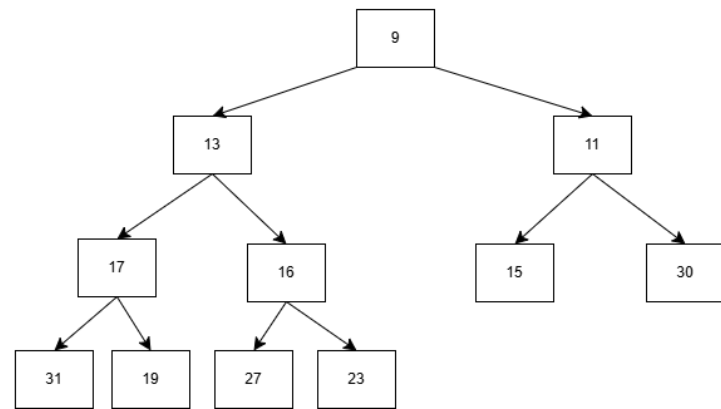
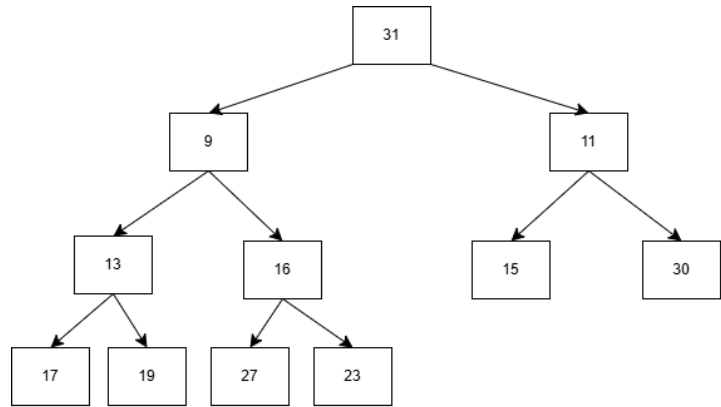
Iteration 2

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



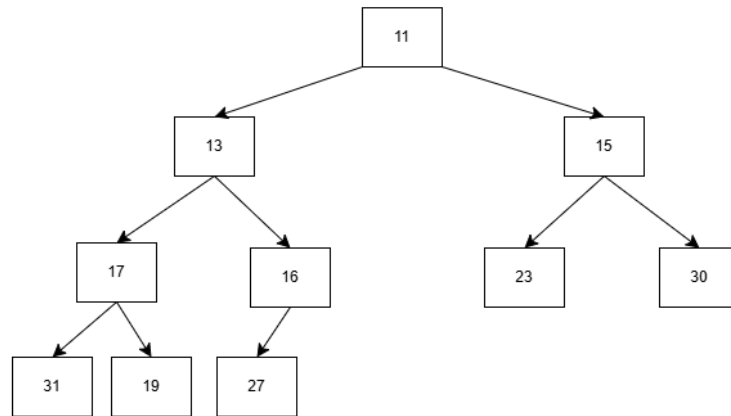
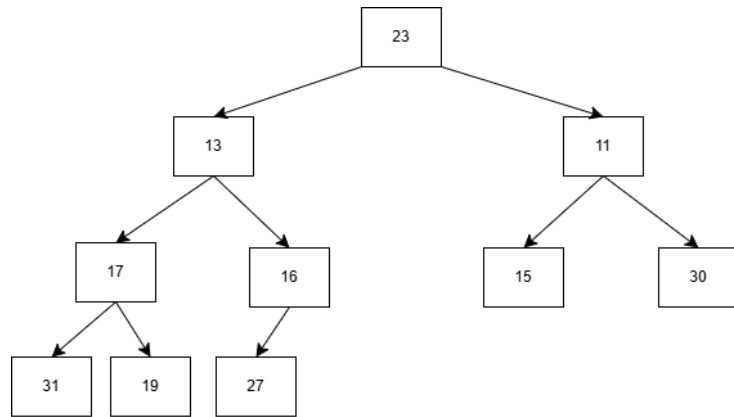
Iteration 3

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



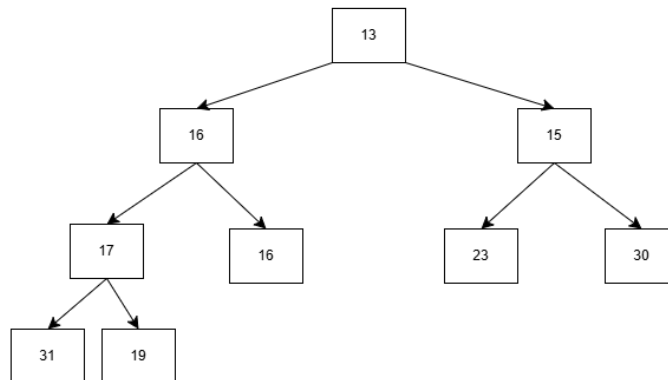
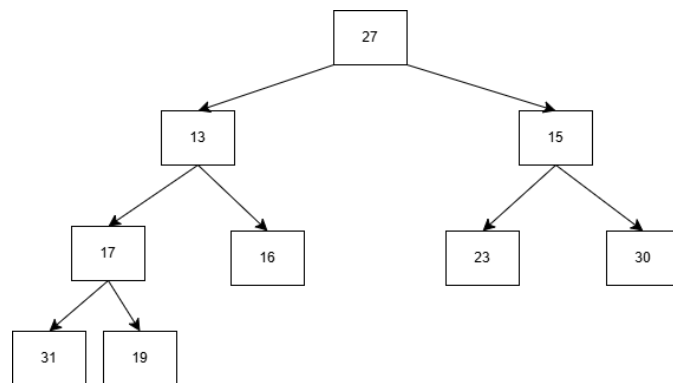
Iteration 4

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



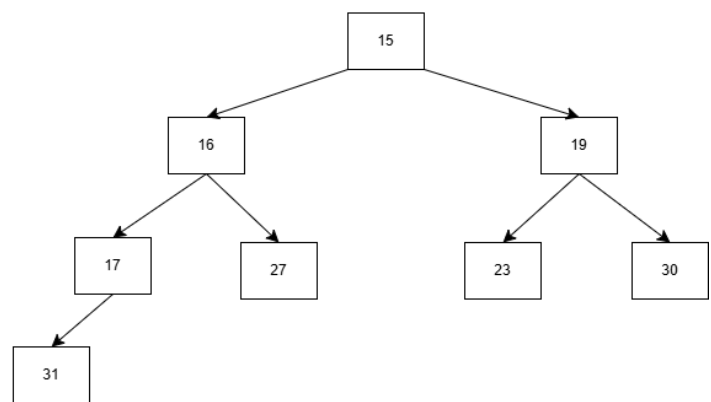
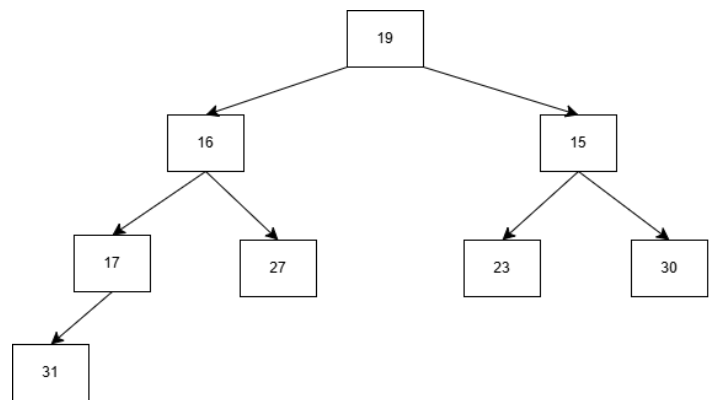
Iteration 5

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



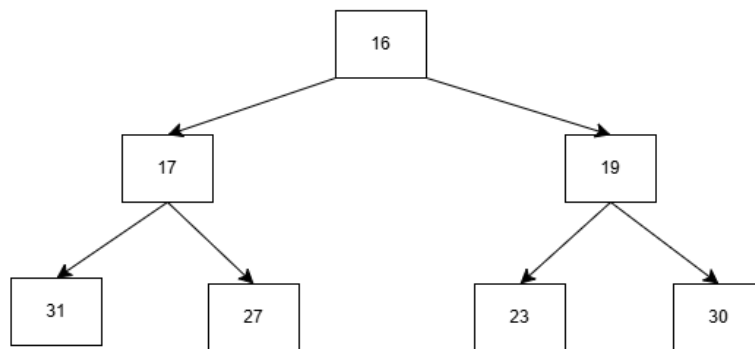
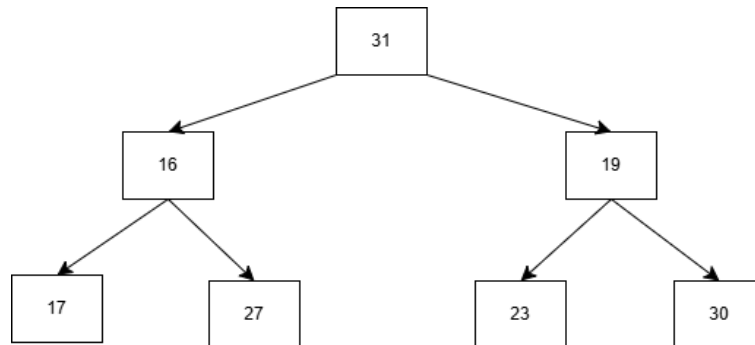
Iteration 6

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



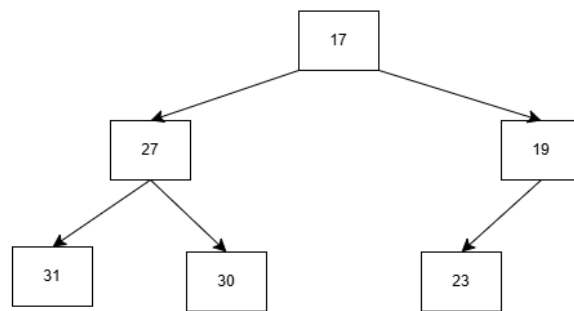
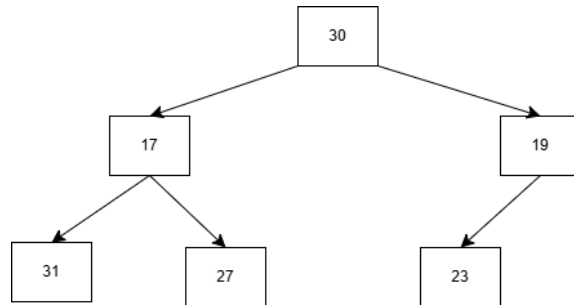
Iteration 7

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



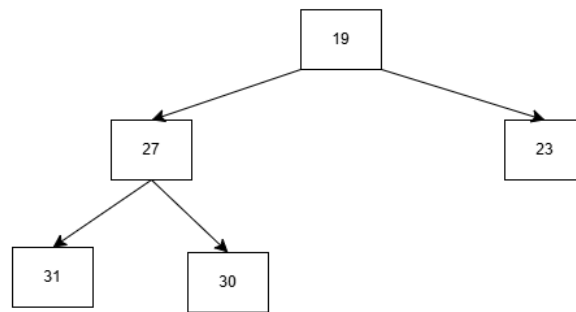
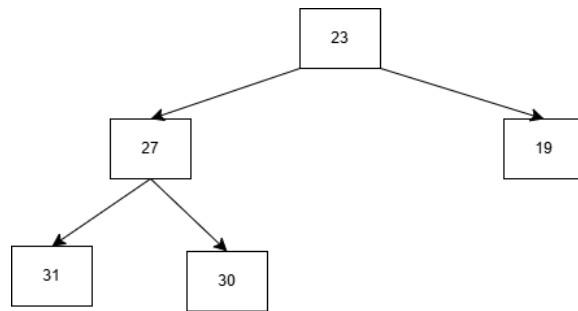
Iteration 8

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



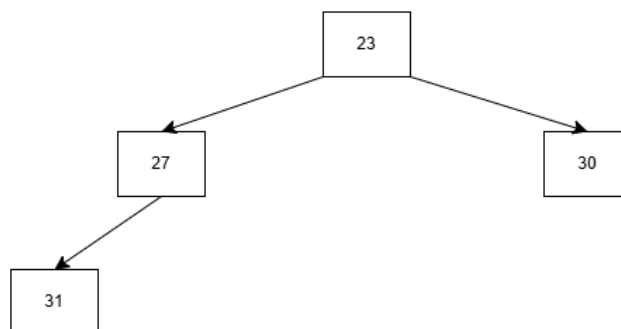
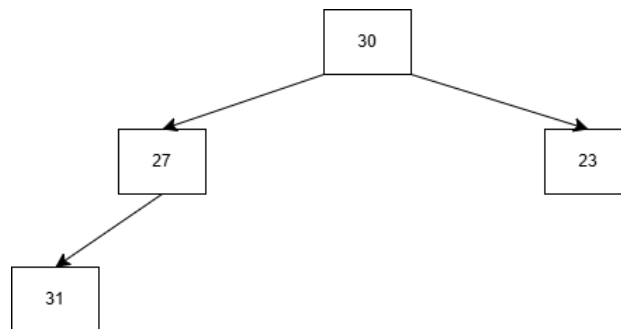
Iteration 9

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



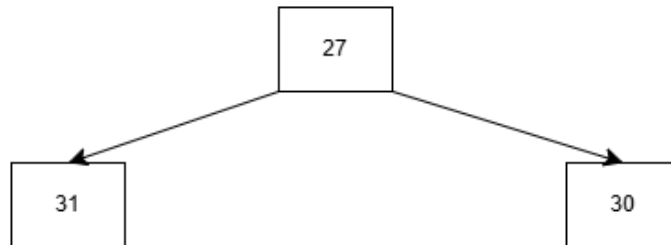
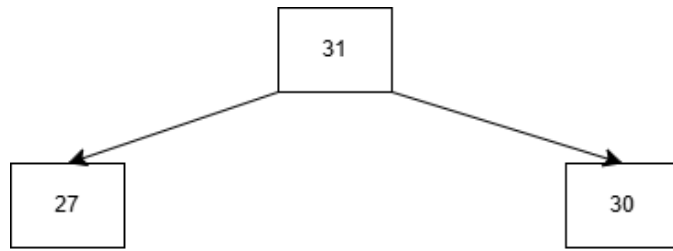
Iteration 10

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



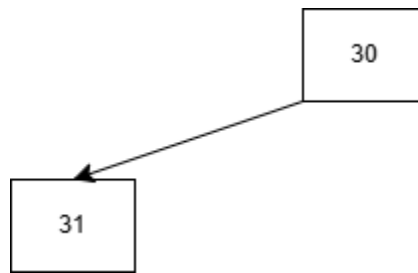
Iteration 11

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



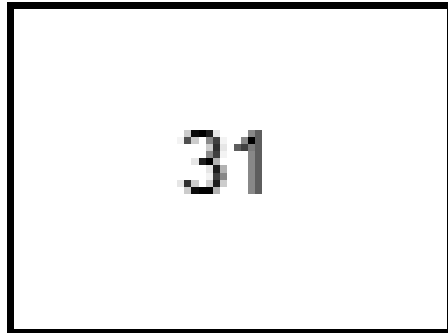
Iteration 12

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



Iteration 13

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



Iteration 14

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

- 1) Initial Left = 0, Right = 14
- 2) $\text{Mid} = (0 + 14) / 2 = 7 \rightarrow$ value at index 7 = 15
 $13 < 15 \rightarrow$ so we move to the right half
- 3) Left = 8, Right = 14
- 4) $\text{Mid} = (8 + 14) / 2 = 11 \rightarrow$ value at index 11 = 7
- 5) $13 > 7 \rightarrow$ move to the left half
- 6) Left = 8, Right = 10
- 7) Mid = 9 \rightarrow value = 11
- 8) $13 > 11 \rightarrow$ move left again
- 9) Left = 8, Right = 8
- 10) Mid = 8 \rightarrow value = 13 Found!

- 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.

Time Complexity:

Algorithm	Best Case	Average Case	Worst Case
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

Insertion Sort:

Insertion Sort is quick only when the list is almost sorted. But if the data is in a messy order, it gets very slow because it must check and move elements one by one.

- Best for small sets of data.
- It works great if you only have a few values to add into sorted list.
- But for big data, it is not the best choice since it slows down quickly.
- Example: Updating a short list of top 5 best selling items.

Heap Sort:

Heap Sort is more consistent. It keeps the same performance whether the data is sorted, reversed, or random. It always runs in $O(n \log n)$ time.

- This one is good when you are working with large and unorganized data.
- It does not matter if the data is already sorted or not because it just works well every time.
- Example: In an e-commerce site that has thousands of products that need to be sorted by price, rating, or sales.

Conclusion:

If I had to choose, I would use Insertion Sort for small, nearly sorted tasks but for large and constantly changing data like real time scoreboards or online stores, Heap Sort is the better, more reliable option.

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

Strength:

- **Consistent Time Complexity:**
One of the best things about Heap Sort is that it always runs in $O(n \log n)$ time, even in the worst case. This makes it very dependable, especially when dealing with large sets of data that can change unpredictably.
- **Great for Real-Time Systems:**
Because it handles frequent updates well, Heap Sort is a great fit for real-time applications like live scoreboards.

Limitations:

- **Not a Stable Sort:**
Heap Sort does not keep the original order of items that have the same value. For example, if two users have the same score, their order might get flipped. This can be a problem.
- **More Complex to Implement**
Even though it is efficient, building and managing the heap takes more effort than simple algorithms like Insertion Sort. So it needs more careful coding to get it right.

Trade-Offs:

- If you need a fast and consistent sorting method and don't care about keeping the original order of equal items, Heap Sort is a great choice.
- If your dataset is small or already mostly sorted or you need to maintain stability in sorting, another algorithm like Insertion Sort might be more suitable.

Conclusion:

Heap Sort is powerful and dependable for sorting large, dynamic datasets. However, its lack of stability and extra complexity means it is not always the perfect fit. Choosing Heap Sort means trading off a bit of simplicity for better performance and scalability.

- 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>
#include <sys/time.h>

// Function to swap two values
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Heapify function
void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

// Heap Sort function
void heapSort(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Sort the heap
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]); // move max to the end
        heapify(arr, i, 0);    // heapify remaining elements
    }
}

// Binary search for descending order
int binarySearch(int arr[], int n, int target) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = (left + right) / 2;

        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            right = mid - 1;
        else
            left = mid + 1;
    }
}

```

```

    }
    return -1; // not found
}

// Function to generate random data
void generateRandomData(int arr[], int size) {
    for (int i = 0; i < size; i++)
        arr[i] = rand() % 100000; // random number 0-99999
}

// Function to print array (for testing)
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Function to get current time in ms
long long currentTimeMillis() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (long long)(tv.tv_sec) * 1000 + (tv.tv_usec) / 1000;
}

int main() {
    // Fixed array from Question 3
    int data[] = {31, 13, 5, 17, 9, 11, 4, 7, 19, 16, 23, 15, 27, 30, 2};
    int n = sizeof(data) / sizeof(data[0]);
    int target = 13;

    // Sort the array in ascending then reverse to get descending
    heapSort(data, n);
    for (int i = 0; i < n / 2; i++)
        swap(&data[i], &data[n - 1 - i]);

    printf("Sorted Array (Descending): ");
    printArray(data, n);

    // Search for value
    int index = binarySearch(data, n, target);
    if (index != -1)
        printf("Target %d found at index %d\n", target, index);
    else
        printf("Target %d not found\n", target);

    // Performance Test
    printf("\nTesting on Large Datasets:\n");
    int sizes[] = {10000, 20000, 30000, 40000, 50000,
        60000, 70000, 80000, 90000, 100000};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    for (int i = 0; i < num_sizes; i++) {
        int size = sizes[i];
        int *largeData = (int *)malloc(size * sizeof(int));
    }
}

```

```
        generateRandomData(largeData, size);

        long long start = currentTimeMillis();
        heapSort(largeData, size);
        long long duration = currentTimeMillis() - start;

        printf("Size: %6d | Time: %5lld ms\n", size, duration);

        free(largeData);
    }

    return 0;
}
```


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

Insertion Sort Applications:

- **Auto-sorting small items (cards, names, files)**
For example, when typing a contact list on a phone, the app might use Insertion Sort to insert a new name into an already sorted list because it is simple and fast for small changes.
- **Sorting in embedded systems**
Devices like washing machines, digital watches often have very limited memory. In such systems, Insertion Sort is a good choice because it is memory efficient and does not require extra space.

Heap Sort Applications:

- **Real time leaderboard or scoreboard**
Heap Sort is perfect for sorting scores quickly in descending order. It can handle frequent updates and large numbers of participants efficiently.
- **Job scheduling systems**
Operating systems use Heap Sort in job scheduling algorithms to assign tasks with priorities. Since Heap Sort can quickly find the highest priority task, it is useful in deciding which job to run next.
- **Event management systems**
In large scale event platforms, Heap Sort is used to sort and prioritize tasks or attendees such as ranking who registered first or allocating VIP priority. It helps maintain an organized and fair queue especially when many users interact with the system at the same time.

Conclusion:

- Insertion Sort is used when the dataset is small and mostly sorted.
- Heap Sort is used when the system needs fast and consistent performance, especially for large or dynamic datasets.