UMPSA

اونيۏرسيتي مليسيا ڤهڠ السلطان عبدالله

UNIVERSITI MALAYSIA PAHANG
AL-SULTAN ABDULLAH

BCI2313 ALGORITHM & COMPLEXITY

SEM II 2024/2025

ASSIGNMENT 1

NAME	MUHAMMAD SYAHMI DANIEL BIN SHIRMI
ID	CA22011
SECTION	04A
LECTURER'S NAME	DR. MOHD AZWAN BIN MOHAMAD @ HAMZA

1. Problem Understanding: Elaborate on the definition of the insertion sort and heap sort algorithms to get the algorithm complexity. [10 Marks]

Answer:

Definition :

Insertion Sort is an intuitive sorting method that arranges the sorted array incrementally, comparing the current element with the preceding ones and placing it in the appropriate spot. Beginning with the second element, it checks elements to the left and moves them if they are bigger until the appropriate position is located.

Time Complexity :

- Optimal Scenario: O(n) – when the data is presorted.
- Typical Scenario: $O(n^2)$
- Worst Scenario: $O(n^2)$ – data sorted in reverse order.
- Space: O(1) – functions in-place.
- Stable: Affirmative

Definition :

Heap Sort is an effective sorting algorithm based on comparisons that utilizes a binary heap data structure (max-heap for sorting in descending order, min-heap for ascending).

Procedures:

- Construct a max-heap using the array.
- Continuously remove the largest element and move it to the end of the array while preserving the heap structure.

Complexity of Time :

- Optimal, Average, Least Case: O(n log n)
- Espacio: O(1) – en el lugar
- Unstable: No – fails to maintain the sequence of identical elements.
- Stable: Confirmed

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. [10 Marks]

Answer:

Heap Sort is a two-step algorithm:

Step 1: Build Max-Heap

- Convert the unsorted array into a **max-heap**, a complete binary tree where each parent node is greater than or equal to its children.
- This is done in O(n) time using the bottom-up heapify approach.

Step 2: Heap Sort

- Repeatedly swap the first element (maximum) with the last element of the heap.
- Reduce the heap size by one and re-heapify the root to restore the max-heap property.
- Continue until all elements are sorted.
- Time complexity: O(n log n)

Assisting Binary Search:

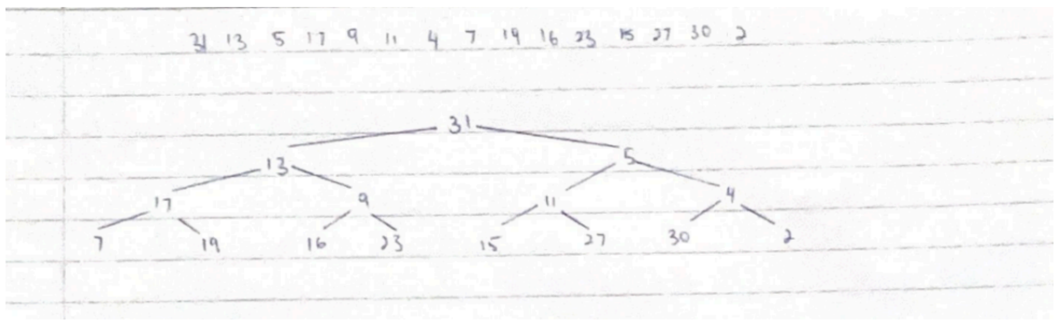
- Binary Search requires a sorted array.
- Heap Sort ensures that the array is sorted (typically in ascending or descending order).
- Once the array is sorted using Heap Sort:
    - Binary Search can be applied to quickly search for a value.
    - Time complexity of Binary Search is O(log n).

Combined Effectiveness:

1. Total time complexity: Heap Sort O(n log n) + Binary Search O(log n) ≈ O(n log n)
2. This approach is efficient for systems that need real-time leaderboard updates and fast search operations.

3. Solve the Problem: Provide a detailed step-by-step solution on 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.
   [20 Marks]

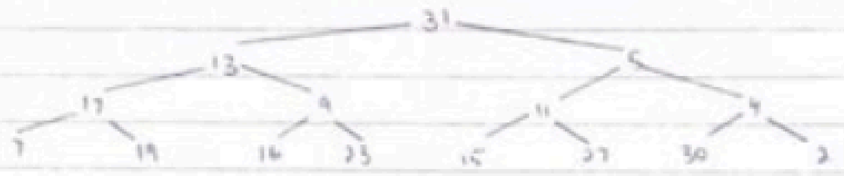
   Answer:
   Step 1: Treat the Array as a Complete Binary Tree
   First, view the array as a full binary tree. In an array of size n, the root is index 0, the left child of an element at index is 2i +1 and the right child is 2i + 2.



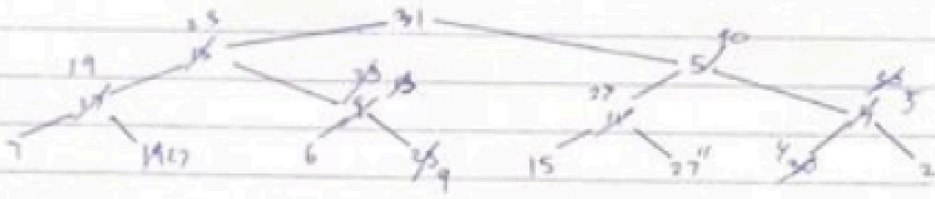
Step 2: Build a Max Heap

Convert the tree to a Max Heap, with each parent node greater than or equal to its children. Then, begin heapifying from the last non-leaf node, which is at index 6, heapify on each node from index 6 to 0.

31 13 5 17 9 11 4 7 19 16 23 15 27 30 2



Step 2 : build Max Heap



Iteration1 : 31 , 13 , 5 , 17 , 9 , 11 , 30 , 7 , 19 , 16 , 23 , 15 , 27 , 4 , 2

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

Iteration 3 : 31 , 13, 5 , 17 , 23 , 27 , 30 , 7 , 19 , 16 , 9 , 15 , 11 , 4 , 2

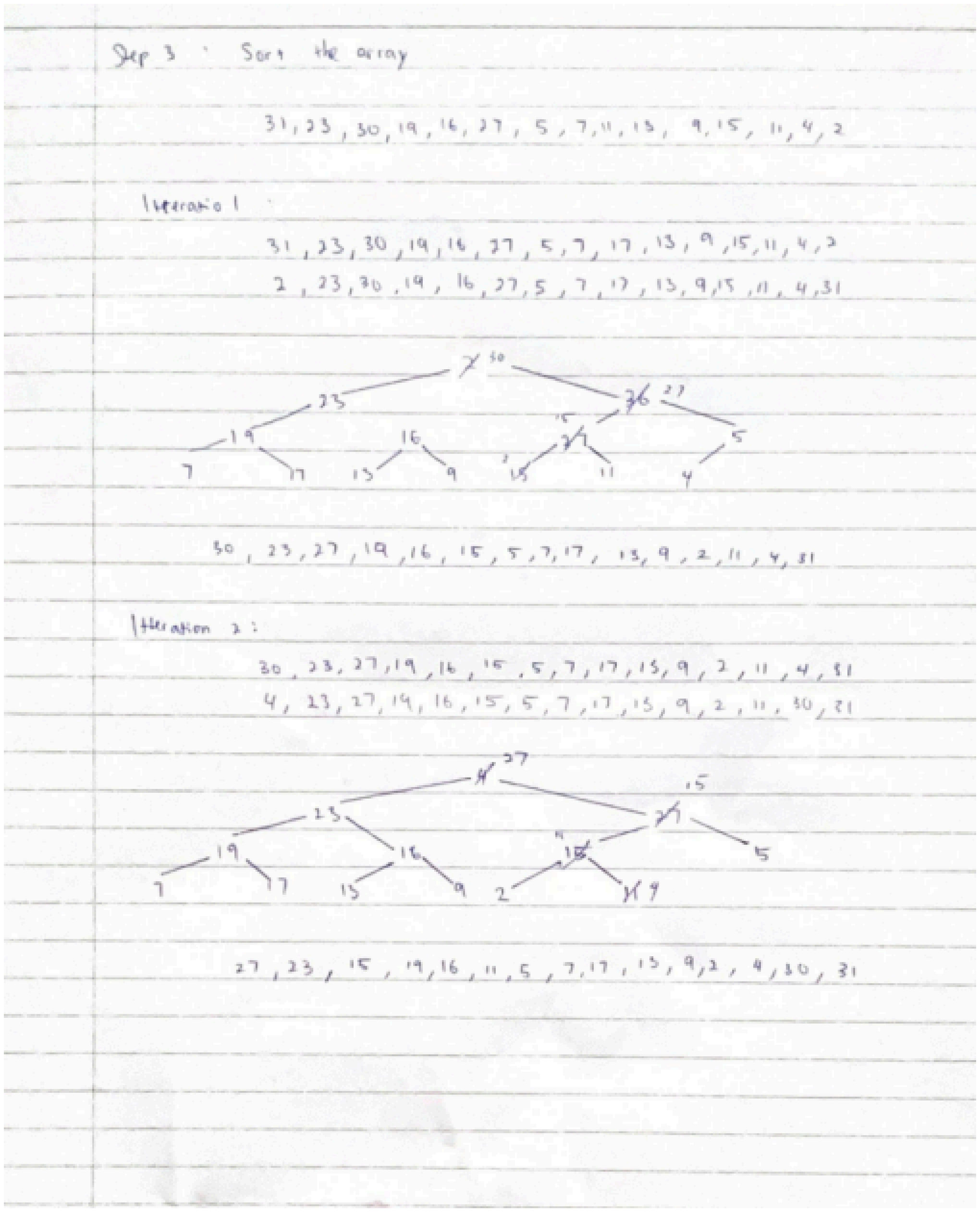
Iteration 4 : 31 , 13 , 5 , 19 , 23 , 27 , 30 , 7 , 17 , 16 , 9 , 15 , 11 , 4 , 2

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

Iteration 6 : 31 , 23 , 30 , 19 , 16 , 27 , 5 , 7 , 17 , 13 , 9 , 15 , 11 , 4 , 2

Step 3: Sort the Array (Heap Sort)

1. Swap the root (largest element) with the last element.
2. Reduce the heap size by 1.
3. Heapify the root to restore Max Heap in the reduced array.
4. Repeat until the array is sorted.



Step 3 : Sort the array

31, 23, 30, 19, 16, 27, 5, 7, 11, 13, 9, 15, 11, 4, 2

Iteratio 1 :

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

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

~~2~~ 30
23
~~30~~ 27
19
16
15
~~27~~
5
7
17
13
9
15
11
4

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

Iteration 2 :

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

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

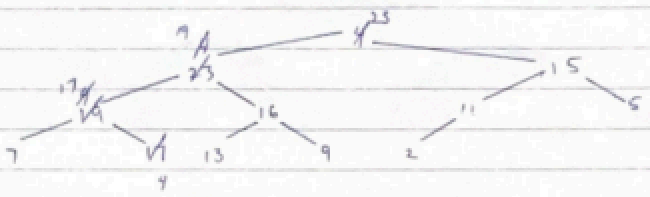
~~4~~ 27
15
23
~~27~~
19
16
11
~~15~~
5
7
17
15
9
2
~~11~~ 4

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

Iteration 3:

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

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

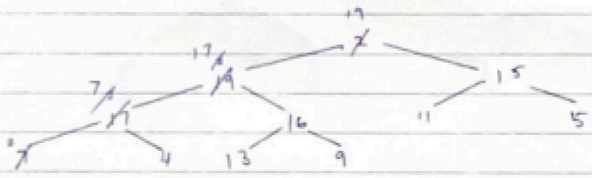


23, 19, 15, 17, 16, 11, 5, 7, 4, 13, 9, 2, 27, 30, 31

Iteration 4:

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

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

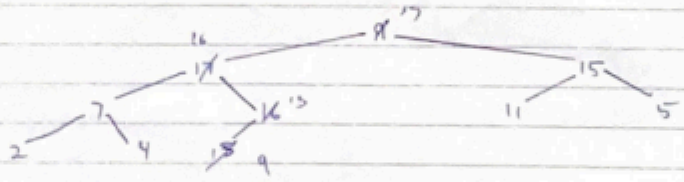


19, 17, 15, 7, 16, 11, 5, 2, 4, 13, 9, 23, 27, 30, 31

Iteration 5:

19, 17, 15, 7, 16, 11, 5, 2, 4, 15, 9, 23, 27, 30, 31

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

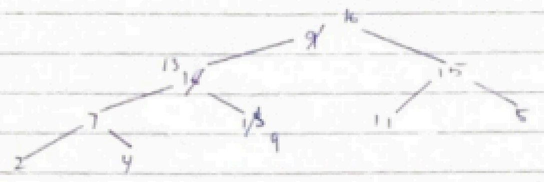


17, 16, 15, 7, 13, 11, 5, 2, 4, 9, 19, 23, 27, 30, 31

Itteration 6 :

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

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

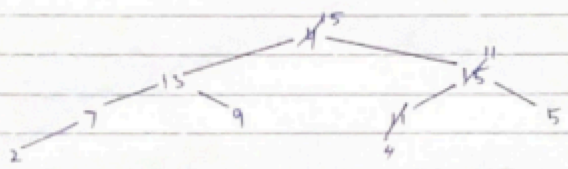


16, 13, 15, 7, 9, 11, 5, 2, 4, 17, 19, 23, 27, 30, 31

Itteration 7 :

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

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

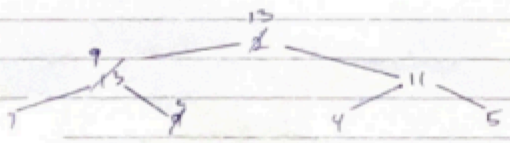


15, 13, 11, 7, 9, 4, 5, 2, 16, 17, 19, 23, 27, 30, 31

Itteration 8 :

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

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

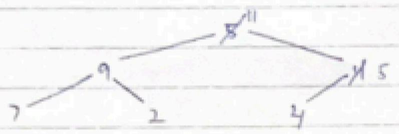


13, 9, 11, 7, 2, 4, 5, 15, 16, 17, 19, 23, 27, 30, 31

Iteration 9 :

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

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

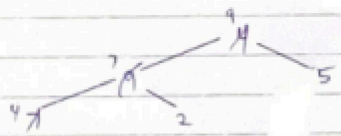


11, 9, 5, 7, 2, 4, 13, 15, 16, 17, 19, 23, 27, 30, 31

Iteration 10 :

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

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

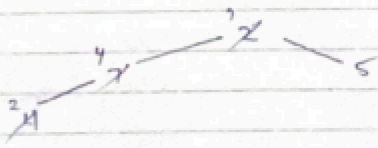


9, 7, 5, 4, 2, 11, 13, 15, 16, 17, 19, 23, 27, 30, 31

Iteration 11 :

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

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

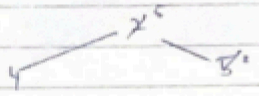


7, 4, 5, 2, 9, 11, 13, 15, 16, 17, 19, 23, 27, 30, 31

Iteration 12 :

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

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

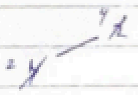


5, 4, 2, 7, 9, 11, 13, 15, 16, 17, 19, 23, 27, 30, 31

Itteration 13 :

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

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



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

Iteration 14 :

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

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

2

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

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

Simply reverse the array to get the descending order.

Step 4: Perform Binary Search

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

Initialize:

- low = 0
- high = 14
- target = 13

Step:

1. First iteration:

   Mid index = $\lfloor \frac{0+14}{2} \rfloor$ = 7 Mid

   value = array [7] = 15

   Since 15 > 13, low = mid - 1

   low = 8

2. Second iteration:

   Mid index = $\lfloor \frac{8+14}{2} \rfloor$ = 11

   Mid value = array [11] = 7

Since 7 < 13, high = mid - 1

high = 10

3. Third iteration:

Mid index = $\left\lfloor \frac{8+10}{2} \right\rfloor$ = 9 Mid

value = array [9] = 11

Since 11 < 13, high = mid - 1

high = 8

4. Fourth iteration:

Mid index = $\left\lfloor \frac{8+8}{2} \right\rfloor$ = 8

Mid value = array [8] = 13

Found target 13 at index 8.

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. [15 marks]

Answer:

Insertion Sort is a straightforward algorithm that organizes data by comparing each element with those preceding it and placing it in the appropriate position. The optimal scenario arises when the data is pre-sorted, leading to an O(n) time complexity. Nevertheless, in average and worst-case situations, particularly when the data is arranged in reverse, the complexity increases to O(n^2). It requires O(1) space and is a stable algorithm, which makes it well-suited for small or almost sorted datasets.

Heap Sort, on the other hand, constructs a binary heap (typically a max-heap) and continuously removes the maximum element to organize the array. Its time complexity remains O(n log n) across best, average, and worst scenarios. Heap Sort similarly requires O(1) space, although it does not maintain stability. It is ideal for extensive datasets where uniform performance is required no matter the sequence of input. In small e-commerce businesses, where product lists are brief and typically pre-sorted, Insertion Sort excels with its O(n) best-case efficiency and low memory consumption. For instance, organizing a list of recently viewed products or refreshing a cart containing several items is quick and effective using Insertion Sort.

Nevertheless, for extensive systems containing thousands of elements like inventory updates, price comparisons, and sorting based on ratings, Heap Sort is favored. It manages large volumes of unordered data effectively with reliable O(n log n) performance. Heap Sort also facilitates real-time functions, such as refreshing a best-selling product list during a flash sale, by ensuring minimal processing times during high system demand.

Insertion Sort is best suited for small datasets or scenarios where the data is nearly sorted. Its simplicity, stability, and in-place sorting make it a good choice for lightweight applications.Heap Sort is ideal for large, unsorted datasets and real-time applications that require consistent performance regardless of data order. It is especially useful in e-commerce systems where scalability and performance under pressure are critical.

5. Critical Evaluation: Discuss the strengths, limitations, and potential trade-offs of the Heap Sort algorithm for solving this problem. [10 Marks]

Answer:

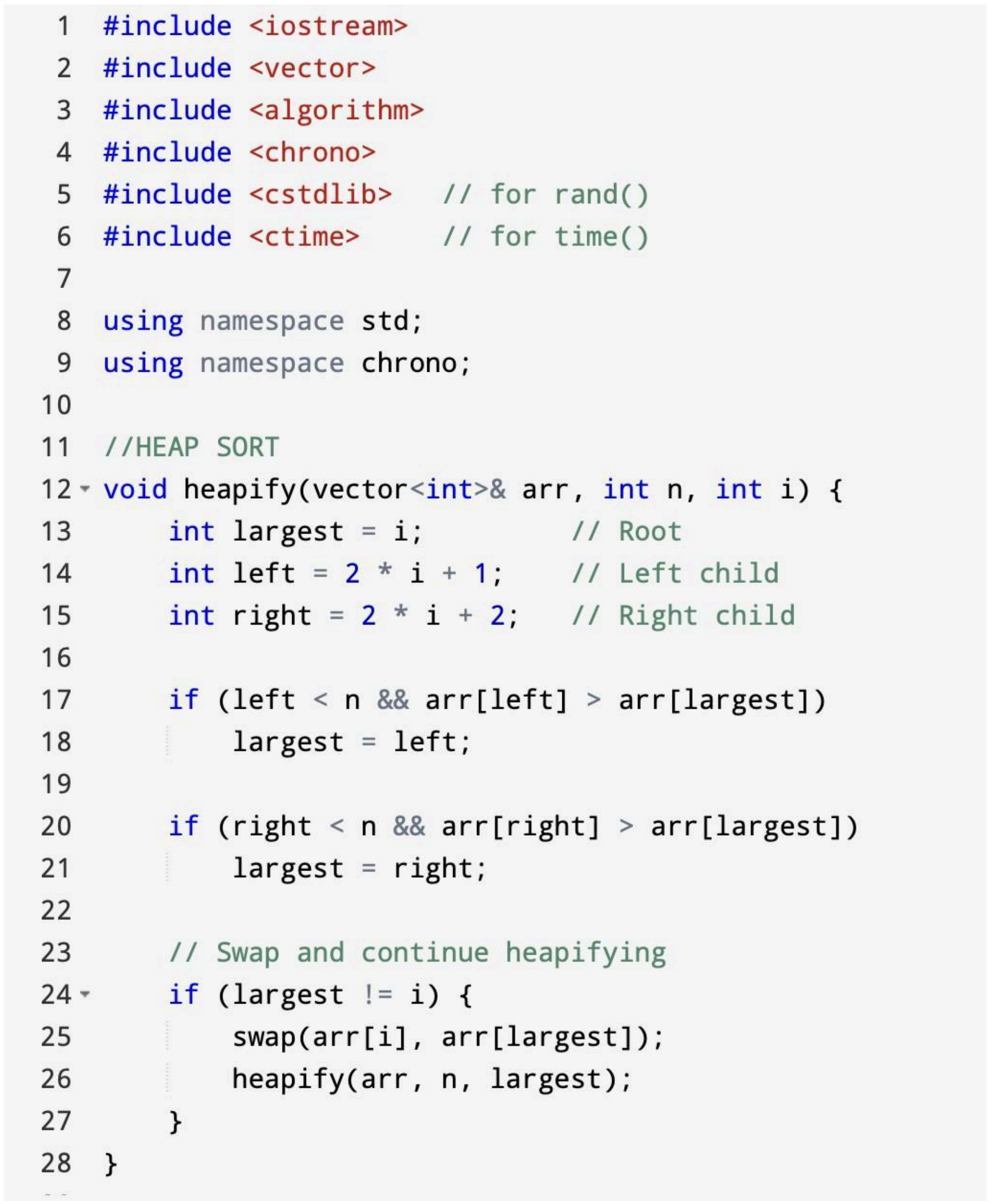
A key advantage of Heap Sort is its time complexity, which consistently holds at O(n log n) across best, average, and worst-case situations. This makes it a dependable option for extensive and randomly arranged datasets where consistent performance is crucial. Moreover, Heap Sort is an in-place algorithm, indicating that it does not need additional memory for sorting, which is advantageous for systems with constrained resources. Heap Sort remains unaffected by the initial arrangement of elements, distinguishing it from other algorithms such as Insertion Sort, which excel on nearly sorted data. This renders Heap Sort exceptionally resilient and adaptable for real-time applications like leaderboard standings, inventory control, and extensive data processing settings.

Even though Heap Sort is efficient, it has various drawbacks. A significant disadvantage is that it doesn't function as a stable algorithm. This implies that identical elements might lose their initial relative arrangements, which is a drawback in situations where that ordering matters—like sorting through multiple criteria (e.g., price and date added).Moreover, the construction and upkeep of the heap necessitate frequent exchanges of elements, which may lead to suboptimal cache performance relative to algorithms such as Merge Sort or Quick Sort. This might somewhat lower its real runtime speed in practice, despite its advantageous time complexity.

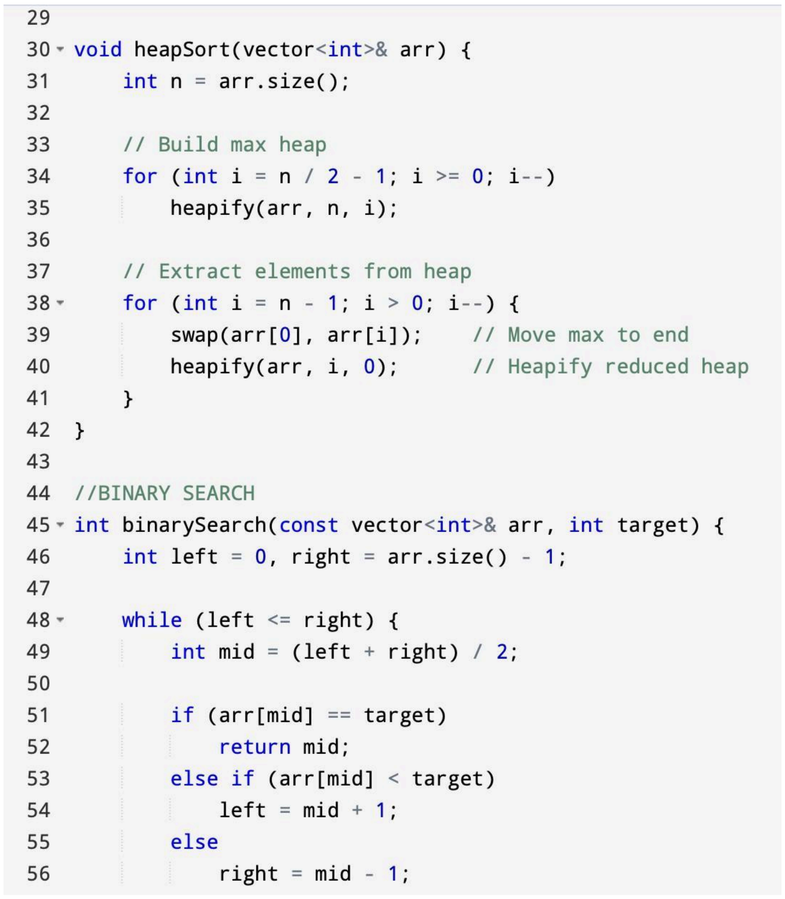
Utilizing Heap Sort entails a balance between stability and efficiency. Although it ensures O(n log n) efficiency, its instability might be unsuitable for applications that need multi-criteria sorting. Additionally, while it requires O(1) space, the complexity of executing heap operations makes it harder to code and debug than simpler algorithms such as Insertion Sort. In systems where performance is crucial and speed along with memory usage take precedence over order stability, Heap Sort is a great option. Nonetheless, in cases where the sequence is important or when the data is partially organized, more appropriate algorithms may be available.

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.

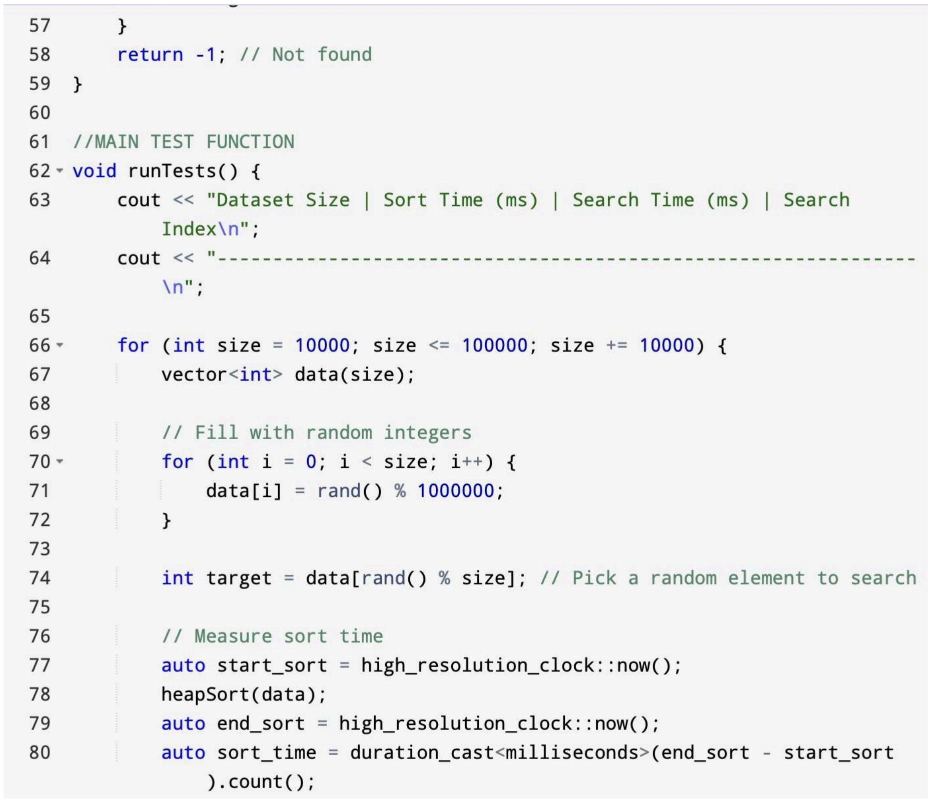
Answer:



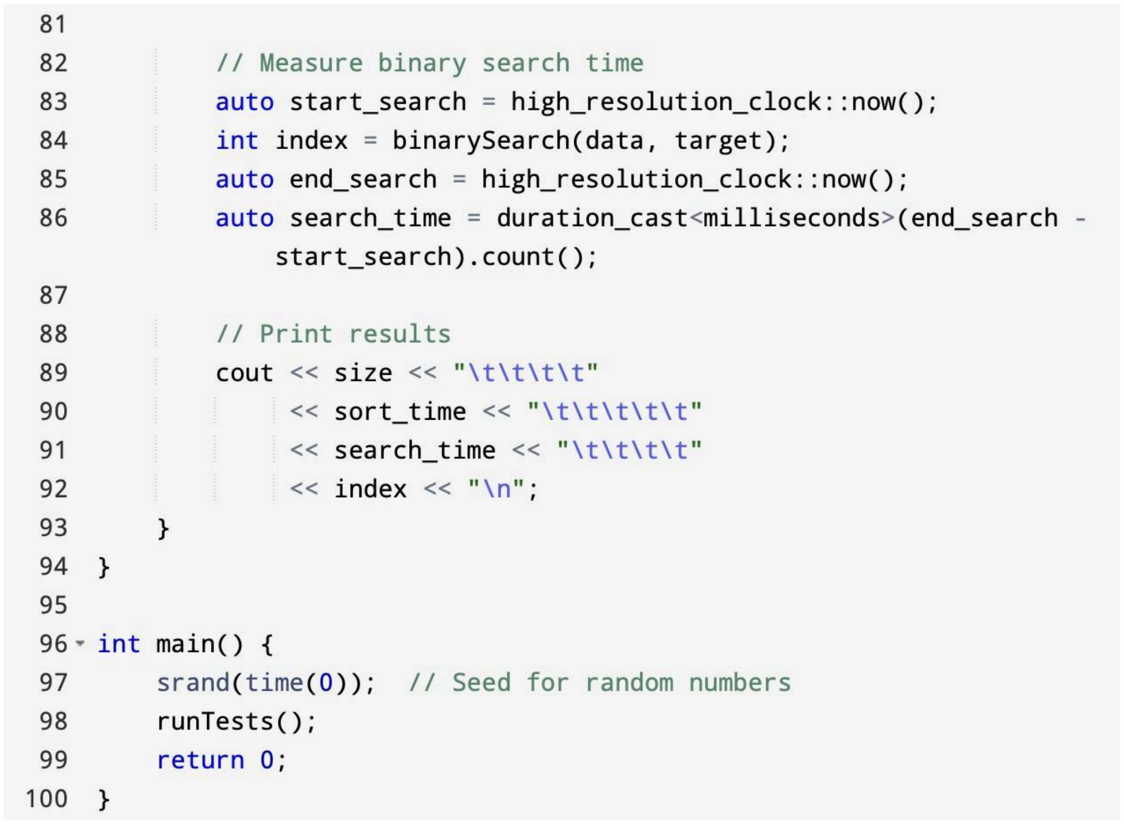
```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <cstdlib>   // for rand()
#include <ctime>     // for time()

using namespace std;
using namespace chrono;

//HEAP SORT
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;         // Root
    int left = 2 * i + 1;    // Left child
    int right = 2 * i + 2;   // Right child

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

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

    // Swap and continue heapifying
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```



```cpp

void heapSort(vector<int>& arr) {
    int n = arr.size();

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

    // Extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);    // Move max to end
        heapify(arr, i, 0);      // Heapify reduced heap
    }
}

//BINARY SEARCH
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;

    while (left <= right) {
        int mid = (left + right) / 2;

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

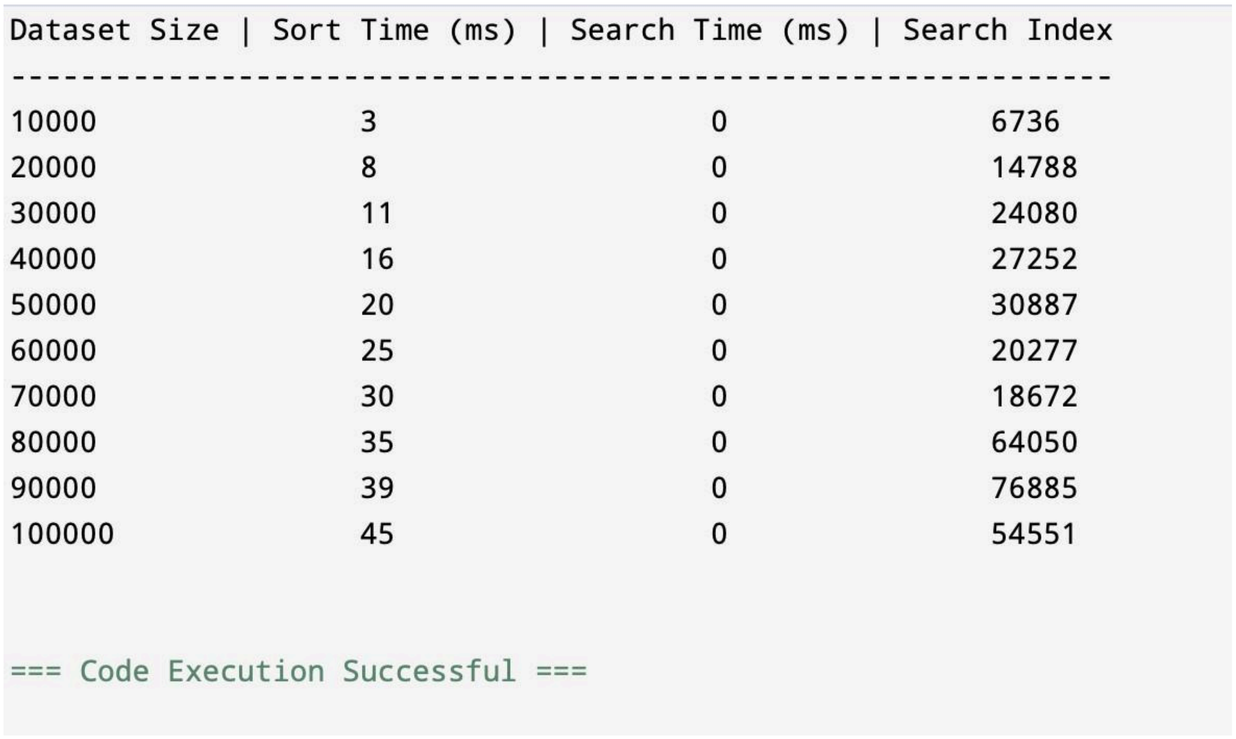


```cpp
    }
    return -1; // Not found
}

//MAIN TEST FUNCTION
void runTests() {
    cout << "Dataset Size | Sort Time (ms) | Search Time (ms) | Search Index\n";
    cout << "--------------------------------------------------------------\n";

    for (int size = 10000; size <= 100000; size += 10000) {
        vector<int> data(size);

        // Fill with random integers
        for (int i = 0; i < size; i++) {
            data[i] = rand() % 1000000;
        }

        int target = data[rand() % size]; // Pick a random element to search

        // Measure sort time
        auto start_sort = high_resolution_clock::now();
        heapSort(data);
        auto end_sort = high_resolution_clock::now();
        auto sort_time = duration_cast<milliseconds>(end_sort - start_sort).count();
```



```cpp

        // Measure binary search time
        auto start_search = high_resolution_clock::now();
        int index = binarySearch(data, target);
        auto end_search = high_resolution_clock::now();
        auto search_time = duration_cast<milliseconds>(end_search - start_search).count();

        // Print results
        cout << size << "\t\t\t\t"
             << sort_time << "\t\t\t\t\t"
             << search_time << "\t\t\t\t"
             << index << "\n";
    }
}

int main() {
    srand(time(0));  // Seed for random numbers
    runTests();
    return 0;
}
```

Output:



```
Dataset Size | Sort Time (ms) | Search Time (ms) | Search Index
---------------------------------------------------------------
10000               3                   0               6736
20000               8                   0               14788
30000               11                  0               24080
40000               16                  0               27252
50000               20                  0               30887
60000               25                  0               20277
70000               30                  0               18672
80000               35                  0               64050
90000               39                  0               76885
100000              45                  0               54551


=== Code Execution Successful ===
```

7. Real-World Applications: State and thoroughly elaborate on five real world applications of the Insertion (TWO applications) and Heap Sort (THREE applications) algorithm. [15 Marks]

Answer:

Insertion Sort:

Contact List Management in Mobile Phones

Insertion sort is commonly used in real-world applications such as managing contacts on mobile phones. When a user adds a new contact, the system does not reorder the entire contact list. Instead, it places the new contact in its proper alphabetical position within the already-sorted list. This simulates the behavior of the insertion sort method, which is useful for tiny datasets or when the data is mostly sorted. It provides a quick and simple approach to preserve order without requiring a sophisticated sorting technique, making it ideal for this common mobile phone feature.

Auto-Complete Suggestions in Mobile Keyboards

Another practical application of insertion sort is in mobile keyboard apps that provide auto-complete or word recommendations. As the user inputs, the system updates and sorts a tiny list of suggested terms based on their frequency or recent usage. Insertion sort is excellent in this case since it works well with tiny datasets and allows for rapid insertion of new words into the sorted list. This ensures that users obtain rapid and relevant word recommendations with minimal delay, hence enhancing typing efficiency and experience.

Heap Sort:

Event Scheduling in Calendar or Reminder Apps

Heap sort is commonly used in scheduling applications such as calendars and reminder apps. These systems frequently have to organize and sort a huge number of activities based on urgency or deadline. Heap sort can be used to efficiently prioritize these jobs, ensuring that impending or critical events are at the top. This guarantees that users are consistently reminded of the most critical tasks first, allowing them to stay organized and on schedule.

Ride-Hailing Apps

Drivers are assigned to riders on ride-hailing platforms such as Grab or Uber based on a number of criteria, including location, rating, and availability. Heap sort helps handle these dynamic priority queues by efficiently sorting drivers based on these criteria. This guarantees that the best potential match is found promptly, reducing wait times and enhancing service quality for both the driver and the rider.

Search Result Ranking in Search Engines

Search engines such as Google and YouTube deal with vast amounts of data and must prioritize the most relevant results at the top. Heap sort is important in this application because it offers a consistent and efficient method for sorting huge datasets. When a user types in a query, the search engine ranks the results based on relevance scores and uses heap sort to display the top results first. This ensures that users get the most useful information immediately, which improves their overall search experience.

time spent, and engagement. Heap sort displays top postings quickly without having to browse through everything, providing a pleasant user experience.