Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16360
e-ISSN : 2541-2019
p-ISSN : 2541-044X

A Comprehensive Analysis of Heap Sort Algorithm for Efficient Sorting Using C++ Programming Language

Rakhmat Purnomo[1)], Tri Dharma Putra[*2)]
[1,2)]Department of Informatics, Faculty of Computer Science, Universitas Bhayangkara Jakarta Raya
rakhmat.purnomo@dsn.ubharajaya.ac.id, tri.dharma.putra@dsn.ubharajaya.ac.id*

Submitted : May 6, 2026 | **Accepted** : July 5, 2026 | **Published** : July 7, 2026

Abstract: Sorting is a need in the computational system including in big data analysis, database management systems, and real time applications. Heap sort is an efficient, comparison-based sorting algorithm that visualizes an array as a binary tree and transforms it into a heap data structure (usually a max heap for ascending sort). The algorithm repeatedly takes the largest element from the root of the heap, swaps it with the last element, and thus reduces the heap size until the heap is sorted The algorithm repeatedly takes the largest element from the root of the heap, swaps it with the last element, and thus reduces the heap size until the heap is sorted. The steps of this algorithm: a. Create Max Heap: Convert the input array into a Max Heap. b. Sort: Swap the root element (the largest element) with the last element, decrease the heap size by 1, and then convert the new root element into a heap. c. Repeat step 2 until the heap is empty. C++ is a known programming language. In this journal we use C++ programming to sort unsorted array. The code is presented in the details. One thorough step by step simulation is given in real data with heap sort and the program is run. The analysis is given by 7 data, namely: [13, 10, 30, 2, 6, 7, 9]. The result is a presented with sorted heap sort. With 7 datasets to be analysed, it is concluded that 6 swaps happened.

Keywords: ascending, algorithm, C++, descending, heap sort, max heap

INTRODUCTION

Heap sort is very efficient in terms of sorting data. It is a fundamental base in a wide array of computational tasks, from data analysis, big data, until database management systems, and real time applications. The main issue in heapsort is the time calculation of it. New challenges appear in big data analysis since large datasets are there in real time application. Hence the efficiency issue comes up (Putra & Purnomo, 2025)(Purnomo & Putra, 2023). This journal presents detailed simulation of heapsort algorithm with a great concept in data structure with trees depicting the processes of algorithm. Theoretical analysis is given, along with the C++ programming and execution results.

Wiredu and friends proposed modification that detects and handles duplicate values more efficiently by reducing unnecessary comparison and swaps at the root of the heap and restructuring the heap more strategically. The conclusion is more efficient algorithm can be detected (No & Wiredu, 2025). Chandra and friends proposed a comparison by implementing and optimized A-star algorithm with heap sort and native A-star on an NPC in a dungeon-crawling game genre by applying five different scenarios. Each has a different start position of the NPC. The result is 50-80% faster than using only A-star algorithm (Istiono, 2023). Ilham and friends studied the comparative analysis of memory performance and processing time of sorting algorithms. The result is that quicksort is the algorithm with the fastest execution (Ilham et al., 2025). Basir analyzed how well performance in sorting data of heapsort and insertion sort algorithms and comparing the results. The result reveals that heap sort gave process time which are consistent in term of time compared to amount of data (Basir, 2020). Wiredu and friends did an experimental research.

The proposed PBPS algorithm leverages proximity-based principles, such as absolute numerical difference, to group and sort similar elements efficiently, reducing unnecessary comparisons and computational overhead. By incorporating dynamic pivot selection (initially using the last element as the pivot, with plans to explore more advanced selection strategies in future work) and adaptive merging strategies, PBPS achieves significant time improvements in both time complexity and memory efficiency. Experimental results demonstrate that PBPS outperforms traditional methods, including merge sort, radix sort, heap sort, and quicksort, particularly with

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026 e-ISSN : 2541-2019
DOI : https://doi.org/10.33395/sinkron.v10i3.16360 p-ISSN : 2541-044X

datasets that exhibit a high degree of data locality (Wiredu & Baagyere, 2025). Sundaramoorthy and friends did a study. This study considers both theoretical aspects, such as time, space complexity, and algorithmic stability, and complements them with empirical analysis using the MATLAB platform. Execution times were benchmarked across a variety of data types, including positive and negative integers, decimal numbers, and characters, over data sizes ranging from 100 to 100,000 elements. Results indicate that the bucket sort delivers the fastest performance for uniformly distributed numeric and character data, while the counting sort excels with positive integers (Sundaramoorthy & Karunanidhi, 2025). Rob and his friends did research in merge sort. This study introduces Wall-L Merge Sort, which combines quadratic sorting with a modifiable multi-layer merging approach. By setting a single parameter, L, which determines the number of merge layers, Wall-L Sort shows a transition in the time complexity from $O(n2)$ to $O(n \log n)$ without any modification in the unique idea. This degree of freedom enables a broad variety of input sizes to be encompassed and expands to several constraint platforms. The results show that Wall-L Sort and K-way Merge Sort have the built-in ability to handle different situations where other algorithms fail without assistance functions. Wall-L Merge Sort is the only sorting algorithm that combines complexity tuning, cache efficiency, recursion depth control, parallelism, and broad adaptability into one framework (Rob et al., 2026). Li and his friends proposed A novel pipelined BISN architecture, which leads to enhanced operating frequency and throughput. The experimental results show that the pre-comparison layer reduces the number of iterations by 6% to 50%. Throughput is improved by more than four times, and operating frequency is increased by more than two times due to the pipelined BISN. The proposed hybrid sorting network reduces sorting time or resource usage, while enabling the sorting of large-scale data sets that other methods cannot support (Sorting, 2025). Fatmaluna and her friends did a research testing. The testing was conducted by measuring execution speed, sorting stability, and memory usage efficiency. The findings from this study show that Bubble Sort ranks lowest in terms of performance and is less suitable for large data sets. Insertion Sort shows better results on small data sets and those with similar patterns. Intro Sort emerges as the most effective algorithm with stable processing time, high adaptability, and faster and more efficient sorting results for various data sizes (Fatmaluna et al., 2026).

Details of arrays in binary tree are given. Each with step by step explanations. Analysis of the tree and array are given to conclude the research gap given. And finally, the algorithm in C++ programming language is ran. The analysis of these arrays and trees are the research gap. The use of seven data arrays are given to simulate step by step explanation thoroughly so that it can easily understood. Trees and arrays are depicted in pictures and explanation of each picture. The choice of 7 data arrays are given to minimalize complicated explanation if data used are large and becomes complicated in the pictures. Finaly this research gives an analysis about the steps doing to run this algorithm thoroughly. The steps explain the analysis. The focus is on heapsort algorithm, yes the comparison between other sorting algorithm is suggested.

The remainder of this paper is discussed as follows: 1. Introduction, this is a brief explanation of what is this algorithm, and why it is worth to be discussed. Several previous works are given in this introduction. 2. Methods. Two sections of analysis are given. A. is the theory behind this algorithm and the B. is the programming of C++ presented. Here the programming running in Visual Studio Code is given and the results were presented. 3. Simulation of Heap sort. Here the simulation analysis is given in details with figures of the tree data structures are presented. 4. Discussion. Thorough discussions are given in this section with explanations in details. 5. Conclusion. In this section conclusion final is given.

METHODS

Heap sort is an efficient, comparison-based sorting algorithm that visualizes an array as a binary tree and transforms it into a heap data structure (usually a max heap for ascending sort) . The algorithm repeatedly takes the largest element from the root of the heap, swaps it with the last element, and thus reduces the heap size until the heap is sorted (Haeupler et al., n.d.)(Wibowo et al., 2024)(Sabah et al., 2023).

Concept of heap sort algorithm

Heap Data Structure: A heap is a complete binary tree in which each parent node has a value greater than or equal to the value of its child nodes (a max heap)(Ali et al., 2021)(Ayazuddin & Scholar, 2025). Array Representation: A binary tree is mapped to an array: For an element index i, the left child element is located at 2i+1 and the right child element is located 2i+2.

Heapify Process: This is the core of the algorithm for maintaining the heap property. If a node violates the heap rule, it is swapped with a larger child element, and the process continues downwards.(Pujiono et al., 2025)(Angga et al., 2025)

Algorithm steps(Wibowo et al., 2024):

1. Create Max Heap: Convert the input array into a Max Heap.
2. Sort: Swap the root element (the largest element) with the last element, decrease the heap size by 1, and then convert the new root element into a heap.
3. Repeat step 2 until the heap is empty.

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16360
e-ISSN : 2541-2019
p-ISSN : 2541-044X

The Heap Sort Algorithm

The heapsort algorithm begins by restructuring the array into a maximal binary heap. The algorithm then repeatedly swaps the heap's root (the largest remaining e lement) with the last element, which is then declared as part of the sorted suffix. The heap, damaged by swapping the root, is repaired so that the largest element is once again at the root. This process is repeated until only one value remains in the heap.

The steps are:

The `heapify()` function is applied to the array. This creates a heap from the array in O(n) operations. The first element of the array (the largest element in the heap) is swapped with the last element of the heap. The size of the heap is reduced by one. The `siftDown()` function is applied to the array to move the new first element to its correct position in the heap. Step (2) is repeated until the array contains only one element. The `heapify()` operation is executed only once and has a throughput of O(n). The `siftDown()` function is called n times and requires O(log n) overhead on each call, since the traversal starts at the root node. Therefore, the throughput of this algorithm is O(n + n log n) = O(n log n). The core of this algorithm is the `siftDown()` function. This function creates a binary heap from a smaller heap and can be understood in two equivalent ways:

Given two binary heaps and a common parent node that belongs to neither, these are merged into a single, larger binary heap. Alternatively, given a "corrupt" binary heap where the maximum heap property (no child is larger than its parent node) applies everywhere except possibly between the root node and its children, the goal is to correct this to create a fault-free heap. To determine the maximum heap property at the root, up to three nodes (the root and its two children) must be compared. The largest node becomes the root. This is easily achieved by finding the largest child and comparing it to the root. There are three cases:
If there are no children (both original heaps are empty), the heap property is trivially satisfied, and no further action is required.

If the root is greater than or equal to the largest child, the heap property is satisfied, and no further action is required.

If the root is less than the largest child, the two nodes are swapped. The heap property now applies to the newly promoted node (greater than or equal to its two children and even greater than any of its children). However, it may be violated between the newly promoted original root and its new children.

Here is the detailed pseudo code of heap sort algorithm. By using C++ programming language as a tool for this sorting analysis.

```
// Function to make max-heap from subtree with root 'i'
void heapify(int arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // if left child is bigger then root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // if right child is bigger than the largest
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // if the largest is not the root, swaps and continue heapify
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// main function of Heap Sort
void heapSort(int arr[], int n) {
    // making heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // taking element one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        // move root this time to the last
        swap(arr[0], arr[i]);

        // call max heapify on the heap that was subtracted
        heapify(arr, i, 0);
    }
```

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16360
e-ISSN : 2541-2019
p-ISSN : 2541-044X

```
}

// function to print array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Array of sorting: \n";
    printArray(arr, n);
```

RESULTS

Defining data to be sorted

Let's say we have unsorted array in the following with numbers to start with, and we will sort them using heap sort: [13, 10, 30, 2, 6, 7, 9].

To convert an array to a binary tree, in the first step, started by inserting the first element of the array as a node in the tree and then continue this process in breadth-first order until all of the array elements have been added. The result is, in a binary tree with all the array elements, as depicted below in figure 1.

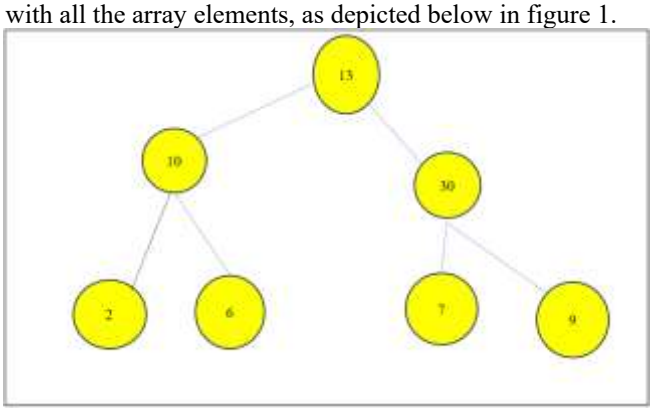


Figure 1. The tree

Converting a Binary Tree to a Max. Heap

After inserting all of elements in the array into a binary tree, then the step after that is to convert this binary tree to a max. heap. All parent nodes in a max. heap then must have a value that is greater than or equal to the value of its child nodes. This ensures that the in the root of a tree, the highest value is always at the root of it. For this tree, explained above, this means swapping the positions of nodes 13 and 30 in the tree to meet the requirements of a max. heap. As shown below, in figure 2.

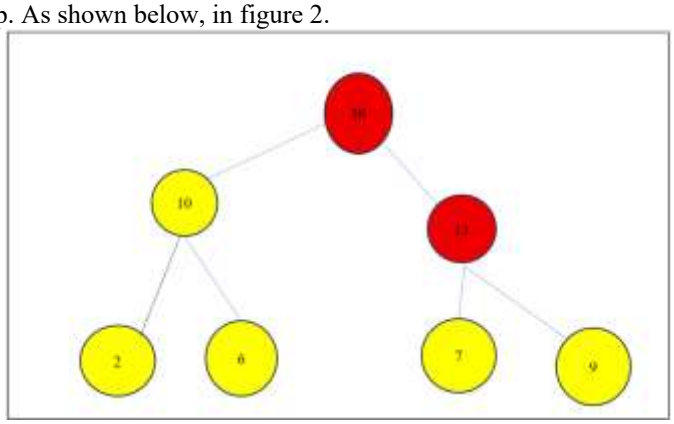


Figure 2. swapping

The tree is converted to be max. heap.
max heap = [30, 10, 13, 2, 6, 7, 9]

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026 e-ISSN : 2541-2019
DOI : https://doi.org/10.33395/sinkron.v10i3.16360 p-ISSN : 2541-044X

To confirm that the max. heap above meets all the necessary properties, we can double-check that it is a complete binary tree and that none of the parent nodes has a value that is less than any of the children. If these situations are met, then it is ready to be used in this algorithm of sorting.

Swap the root node with the last element in the heap

The next step in the process of sorting is to swap the root node. This root node contains the largest element, you can see the last node in the heap. It is concluded you then swap the element at the top of the max-heap array with the bottom element of the max-heap array. Regardless of how you choose. To analyse this step, 30 ends up at the bottom of the array, and 9 ends up at the top of the array. This process is illustrated in the data below.

Max heap = **30**, 10, 13, 2, 6, 7, **9**
[**9**,10,13,2,6,7,30]

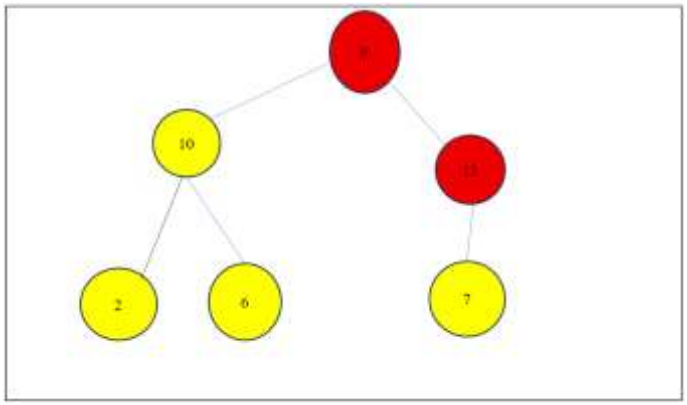


Figure 3. max heap

You may notice that the element in the last of the array, contains the value 30, is colored in red. Hence, because in future iteration and the next step, it will removed the last value, as it is in a position that is already sorted. Therefore, we continue with the following array for the next step: [9, 10, 13, 2, 6, 7]. Now we convert the array to a tree, and then the tree to a max. heap. This step is illustrated in the figure 3., above:

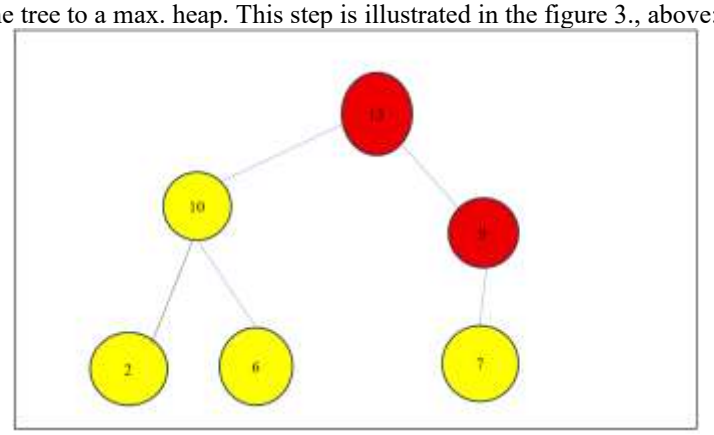


Figure 4. swapping

Thus, we will swap the root node by using the last element in the heap, then the visualized heap now will have one element less. Please take a look at figure 4. above. This is the result of the largest element placed at the end of the array and the iterations subsequently will be excluded. Then this is must be excluded, otherwise the sorting algorithm will never complete. Therefore, the next iteration, whenever the root node is swapped, must exclude the last element in the heap. In the scenario described above, we can see one less node in both the binary tree of the array and its max heap representation.

Call the heapify() function

Now, let us call the process to convert a tree or array to be a max. heap heapify. This will help with the function naming in the section of implementation of this journal.

It is also important to note that the tree structure may no longer meet the max. heap requirements after the root node swap in the given example. The heapify() function must be called again to restore the max heap property. This will result in a rearrangement of the heap as shown:

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026 e-ISSN : 2541-2019
DOI : https://doi.org/10.33395/sinkron.v10i3.16360 p-ISSN : 2541-044X

[13, 10, 9, 2, 6, 7]
And once again we swap the values at the first and last positions in the max. heap array representation.

Max heap = **13**, 10, 9, 2,6,**7**
[7,10, 9, 2,6, 13, 30]

3.1. Repeat steps c and d until the stack is sorted.

You will notice that the number '13' is now highlighted in red, indicating that it has been removed from the next step. Another observation is that the red elements are sorted in the order of ascending. Once you have completed the steps, all elements will be highlighted in red, indicating that they have been removed from the steps, and they will also be sorted in the order of ascending.

The figure below, figure 5., illustrates the complete iteration of the sorting process and also depicting its steps.

[7,10, 9, 2,6]

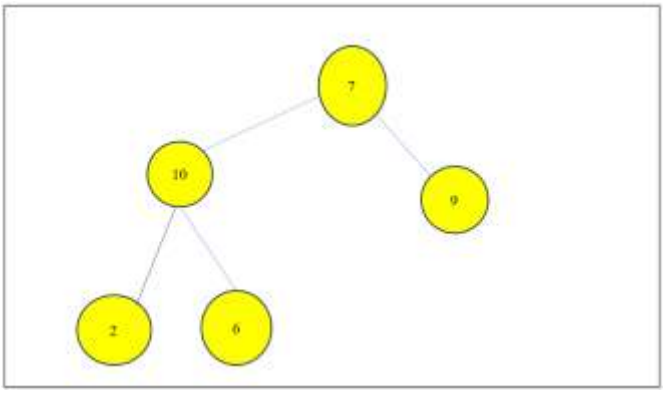


Figure 5. Build binary tree

Please take a look of figure 6. below, it is the max. heap.
Max heap = **10**,7,9,2,**6**
[6,7,9,2,10, 13,30]
The next step is sorting this data:
[6, 7, 9, 2]

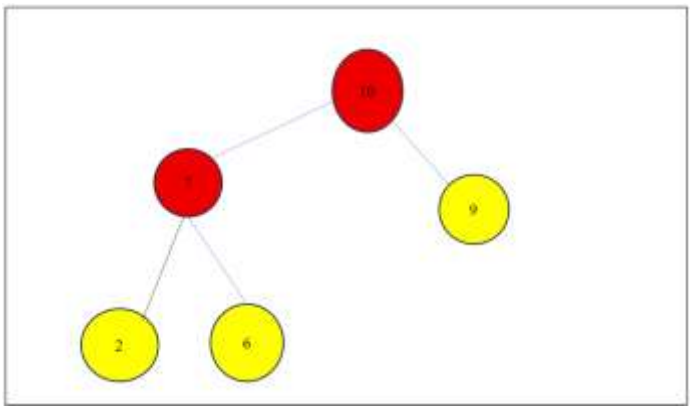


Figure 6. Build max heap

Again this is the step to build binary tree in figure 7. Below the root is 6, and below 6 there are binary tree which are 7 and 9 and below 7 is 2..

*name of corresponding author
This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16360
e-ISSN : 2541-2019
p-ISSN : 2541-044X

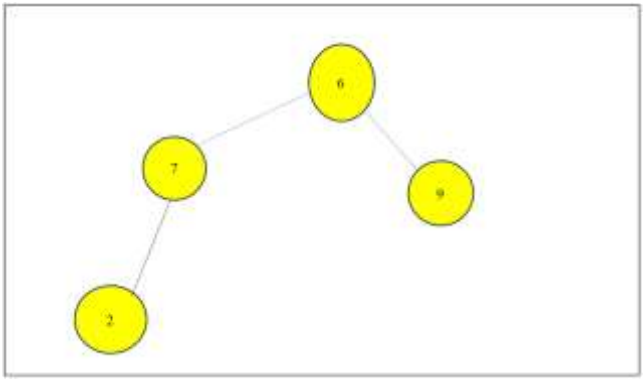


Figure 7. Build binary tree

Please take a look at figure 8 below, this is the max. heap. The 9 is swapped with 6. So the root is 9 now.

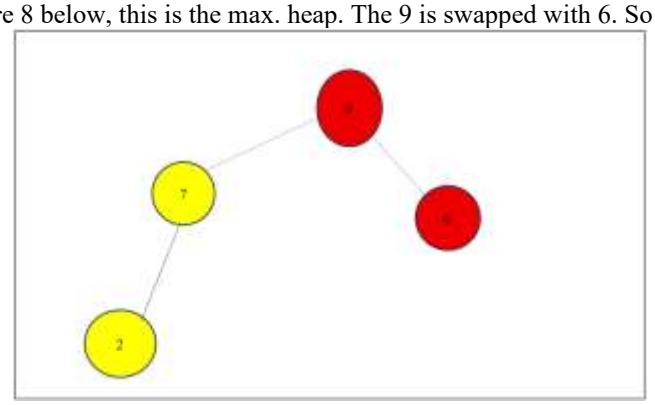


Figure 8. Build max. heap

Max heap=[9,7,6,2]
[2, 7, 6, 9, 11, 13, 31]
The next to be sorted is:
[2, 7, 6]

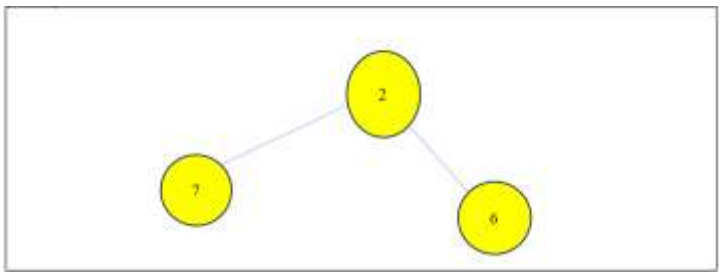


Figure 9. Build binary tree

Please take a look at figure 9. above, this is the binary tree. There are only 7, 2 and 6 left. Then we swap the 7 with 6.

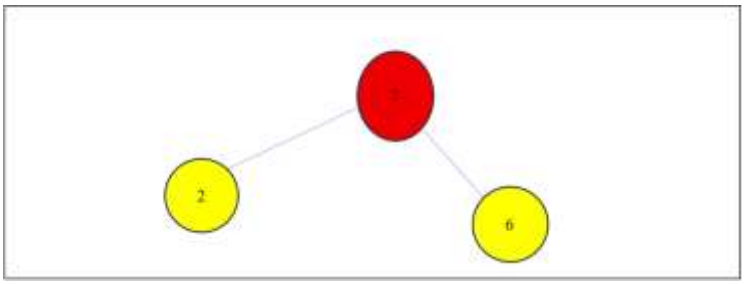


Figure 10. Build max. heap

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

The figure 10. above, the the max. heap.

Max heap = [**7,** 2, **6**]
[6, 2, 7, 9, 10, 13, 30]

Finally the sort is in ascending
[2,6,7,9,10,13,30]

Final result:
2, 6, 7, 9, 10, 13, 30

DISCUSSION

From the analysis result above, we found that the total swaps were 6 times. The swap is replacing one data into another with different array. Each from the max. heap sort algorithm in the tree. The swaps will be increased along with the additional datasets. In this result we only use 7 datasets. The more datasets will make the more time to execute this algorithm. In this simulation, we only used 7 data since we want to explain in in steps, with pictures of array and tree. If larger data used, then the pictures analysis could be complicated. It is known that the complexities of heap sort is not affected by many different data conditions and forms. The comparison needed that must be done for the first cycle is n. Comparison that needed to applied in the 2nd cycle is n-1. And it is continued like that afterwards. The total comparison is 2(n-1). The use of C++ as a programming language is because this is a powerful programming language, and many people familiar with it. Again C++ is a powerful programming language, that is why we chose C++ in this research.

CONCLUSION

The heapsort algorithm begins by restructuring the array into a maximal binary heap. The algorithm then repeatedly swaps the heap's root (the largest remaining element) with the last element, which is then declared as part of the sorted suffix. The heap, damaged by swapping the root, is repaired so that the largest element is once again at the root. This process is repeated until only one value remains in the heap. We have already conducted analysis of heap sort in 7 datasets. Heap sort methods is categorized as sorting method with the best time processes performance. The analysis given are an unsorted array in the following with numbers to start with, and we will sort them using heap sort: [13, 10, 30, 2, 6, 7, 9]. It is concluded that 6 swaps happened in this simulation. The more datasets will make more time to do the execution. For future works, comparison with other known sorting algorithms is suggested, like insertion sort, merged sort. It is suggested also to use other programming language, like java or R to see whether there will be improvements in time processing. The use of C++ as a programming language is because this is a powerful programming language, and many people familiar with it. Not only people is familiar with C++ programming language, but also this language is used in many large or small applications. This C++ programming language is a powerful language.

REFERENCES

Ali, H., Nawaz, H., Maitlo, A., & Soomro, I. (2021). *Performance Analysis of Heap Sort and Insertion Sort Algorithm*. *9*(May), 580–586.

Angga, R., Putra, B., Prihatmanto, A. S., & Yusuf, R. (2025). *Heap Optimization in A * Pathfinding for Horror Games*. *7*(1), 924–940. https://doi.org/10.51519/journalisi.v7i1.941

Ayazuddin, R., & Scholar, G. (2025). *A Comprehensive Study of Sorting Algorithm Performance Using Real-World Dataset Metrics A Comprehensive Study of Sorting Algorithm Performance Using Real-World Dataset Metrics*. 0–12. https://doi.org/10.20944/preprints202509.2550.v1

Basir, R. R. (2020). Analisis Kompleksitas Ruang dan Waktu Terhadap Laju Pertumbuhan Algoritma Heap Sort, Insertion Sort dan Merge dengan Pemrograman Java. *STRING: Satuan Tulisan Riset Dan Inovasi Teknologi*, *5*(2), 109–118. https://doi.org/http://dx.doi.org/10.30998/string.v5i2.6250

Fatmaluna, S. N., Aulia, N. G., Rahma, A., Aulia, S., & Pujiono, I. P. (2026). *Comparison of Memory Efficiency and Computation Time of Bubble Sort , Insertion Sort , and Intro Sort Algorithms UsinComparison of Memory Efg the C ++ Programming Language*. *5*(2).

Haeupler, B., Hladík, R., Iacono, J., & Nov, D. S. (n.d.). *Fast and Simple Sorting Using Partial Information*.

Ilham, M. N., Setiawan, A. F., Kholifatun, I., & Aldiansyah, M. H. (2025). *Comparative Analysis of Memory Performance and Processing Time of Five Sorting Algorithms Using C ++ Programming Language*. *4*(3).

Istiono, W. (2023). *Heap-sort Algorithm on NPC*.

No, I., & Wiredu, J. K. (2025). *Available Online at www.ijarcs.info Optimizing Heap Sort for Repeated Values : A Modified Approach to Improve Efficiency in Duplicate-Heavy Data Sets*. *15*(6), 12–18.

Pujiono, I. P., Rachmawanto, E. H., Anisa, N., & Winarsih, S. (2025). *Array Sorting Algorithm vs Algoritma Pengurutan Tradisional : Analisis Efisiensi Memori dan Waktu Array Sorting Algorithm vs Traditional*

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Sorting Algorithm : Memory and Time Efficiency Analysis. *15*(April), 47–59.
Purnomo, R., & Putra, T. D. (2023). Theoretical Analysis of Standard Selection Sort Algorithm. *SinkrOn*, *8*(2), 666–673. https://doi.org/10.33395/sinkron.v8i2.12153
Putra, T. D., & Purnomo, R. (2025). *Exchange Sort and Selection Sort Algorithms : Comparison and Theoretical Analysis*. *7*, 732–741. https://doi.org/10.30865/json.v7i2.9306
Rob, M. A., Hossen, Z., Hossen, K., Ali, M., & Roy, B. (2026). *Wall-L merge sort : A tunable and adaptive sorting algorithm for diverse computing environments*. 1–18. https://doi.org/10.1371/journal.pone.0341993
Sabah, A. S., Abu-naser, S. S., Helles, Y. E., Abdallatif, R. F., Samra, F. Y. A. A., Helmi, A., Taha, A., Massa, N. M., & Hamouda, A. A. (2023). *Comparative Analysis of the Performance of Popular Sorting Algorithms on Datasets of Different Sizes and Characteristics*. *7*(6), 76–84.
Sorting, A. D. (2025). *A Scalable Sorting Network Based on Hybrid Algorithms for*. 19–21.
Sundaramoorthy, S., & Karunanidhi, G. (2025). *A systematic analysis on performance and computational complexity of sorting algorithms*.
Wibowo, F. R., Faisal, M., Magisterinformatika, P. S., Negeri, U., Maulana, I., & Ibrahim, M. (2024). *Comparative Analysis of Sorting Algorithms : TimSort Python and Classical Sorting Methods*. *07*(01), 11–18.
Wiredu, J. K., & Baagyere, E. Y. (2025). *A Novel Proximity-based Sorting Algorithm for Real-Time Numerical Data Streams and Big Data Applications*. *186*(71), 1–10.

*name of corresponding author

This is anCreative Commons License This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.