Introduction To Algorithms

Konrad Markowski

March 2024

1 Introduction

2 Sorting Algorithms

2.1 Insertion Sort

Illustration of insertion sort algorithm is shown below (1). Main iteration starts with the for loop from the second number in the array which in that case is 3 and continues to the end of the array. For the main for loop green numbers are chosen. Within every for loop there is a while loop which iterates back as long as it finds number bigger than smaller green one. If smaller number is found green number is inserted after found number and index of red numbers is incremented. If smaller number is not found and while loop iterates all the way back to the start chosen green number is inserted at the very beginning of the array. Time complexity

Examples of code logic on specific examples is shown below. Example od case when while loop iterates all the way through to the start.

- in main for loop green number 3 is chosen,
- while loop begins and chooses 4 as its first and in that case the last number,
- 4 is bigger than 3 so it iterates to the beginning of the array,
- 3 is inserted as the first element of the array and index of number 4 is incremented.

Example of case when while loop does not iterate all the way back to the start.

- in main for loop green number 5 is chosen,
- while loop begins and chooses 12 as its first number,
- 12 is bigger than 5 so while loop iterates further and 10 is chosen,
- 10 is bigger that 5 so it iterates further and 4 is chosen,
- 4 is not bigger than 5 so 5 is inserted after 4 and indexes of numbers 10 and 12 are incremented.

Figure 1: Insertion sort algorithm illustration

Implementation of code in C++ as well as the console output is shown below.

```
void InsertionSort::sort(std::vector<int>& vec)
{
    Utils::print(vec);
    for (int8_t i = 1; i < vec.size(); i++)
    {
        int key = vec[i];
        int8_t j = i - 1;
        while(j >= 0 and vec[j] > key)
        {
            vec[j + 1] = vec[j];
            j--;
        }
        vec[j + 1] = key;
        Utils::print(vec);
    }
}
```

```
konradmarkowski@Konrads-MacBook-Air Algorytmy % ./makefile.sh
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12
```

Considering time complexity insertion sort performs rather badly because of while loop nested in for loop therefore insertion sort has worst and the average case time complexity of $\Theta(n^2)$. For the best case where given array is already sorted algorithm iterates only once though the array making the time complexity of $\Omega(n)$. Space complexity of insertion sort is constant and not dependent of given array size $O(1)$. All indicators considering time and space complexity are shown in table (1) below.

time complexity worst case	$O(n^2)$
time complexity average case	$\Theta(n^2)$
time complexity best case	$\Omega(n)$
space complexity	$O(1)$

Table 1: Insertion sort algorithm time and space complexity

2.2 Selection Sort

Illustration of selection sort algorithm is shown below (2). It's the simplest sorting algorithm and it's composed of two for loops one nested in the other. Steps executed by the sorting function are shown below:

- in the first main iteration the smallest number from whole array is chosen which is 13,
- the number 13 is swapped with the first element of array which is 29,
- the next main iteration starts from second element which is 72 and the smallest number is chosen which is 29,
- the number 29 is swapped with the second element of array which is 72.
- this process lasts until the end of the array and therefore array is sorted.

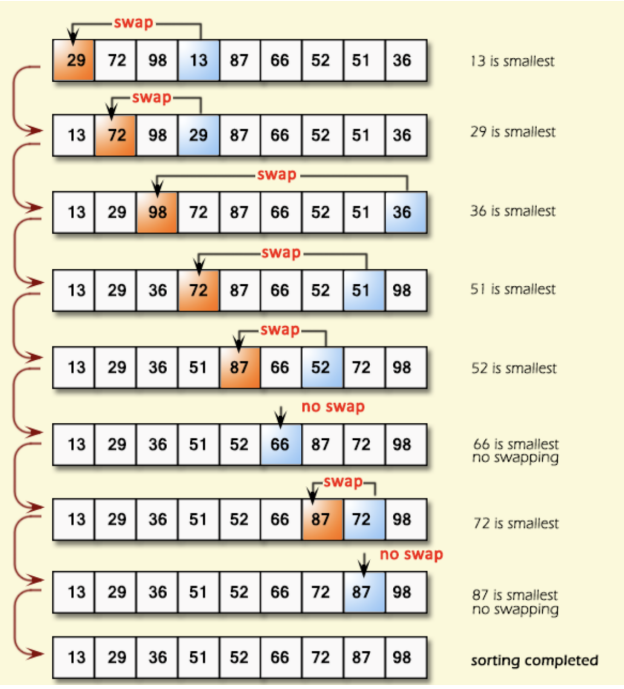

Figure 2: Insertion sort algorithm illustration

Implementation of code in C++ as well as the console output is shown below.

```
void SelectionSort::sort(std::vector<int> &vec)
{
    Utils::print(vec);
    for (uint8_t i = 0; i < vec.size(); i++)
    {
        int smallestCurrentNumber = vec[i];
        uint8_t smallestCurrentNumberIndex = i;
        for (uint8_t j = i + 1; j < vec.size(); j++)
        {
            if (smallestCurrentNumber > vec[j])
            {
                smallestCurrentNumber = vec[j];
                smallestCurrentNumberIndex = j;
            }
        }
        vec[smallestCurrentNumberIndex] = vec[i];
        vec[i] = smallestCurrentNumber;
        Utils::print(vec);
    }
}
```

```
konradmarkowski@Konrads-MacBook-Air algorithms % ./makefile.sh
29 72 98 13 87 66 52 51 36
13 72 98 29 87 66 52 51 36
13 29 98 72 87 66 52 51 36
13 29 36 72 87 66 52 51 98
13 29 36 51 87 66 52 72 98
13 29 36 51 52 66 87 72 98
13 29 36 51 52 66 87 72 98
13 29 36 51 52 66 72 87 98
13 29 36 51 52 66 72 87 98
13 29 36 51 52 66 72 87 98
```

Considering time complexity selection sort performs worst among other sorting algorithms. Selection sort implementation has for loop nested in other for loop therefore selection sort has average case time complexity of $\Theta(n^2)$. For the best case where given array is already sorted algorithm nonetheless iterates through whole array making it $\Omega(n^2)$ time complexity as well. Space complexity of selection sort is constant and does not dependent of given array size $O(1)$. All indicators considering time and space complexity are shown in table (Fig. 2) below.

time complexity worst case	$O(n^2)$
time complexity average case	$\Theta(n^2)$
time complexity best case	$\Omega(n^2)$
space complexity	$O(1)$

Table 2: Selection sort algorithm time and space complexity

2.3 Bubble Sort

Illustration of selection sort algorithm is shown below (Fig. 3). It's one of the simplest sorting algorithm and it's composed of main for loop and for loop nested in prior. It's based on doing swaps of two neighbouring numbers if one is greater than the other so the bigger number is further away from the start. While loop continues until array is sorted or in other words when no swaps were executed in previous iteration. Nested for loop iterates through all numbers in the array doing swaps. Steps executed by the sorting function are shown below:

- main while loop starts as well as for loop and first comparison between 5 and 1 is made,
- 5 is greater than 1 so swap is made so 5 is further away from start,
- next comparison is made between swapped earlier 5 and 4,
- 5 is greater than 4 so swap is made and so on,
- in the illustration (Fig. 3) in the third pass (third while loop iteration) no swap is made thus it's the last iteration and array is sorted.

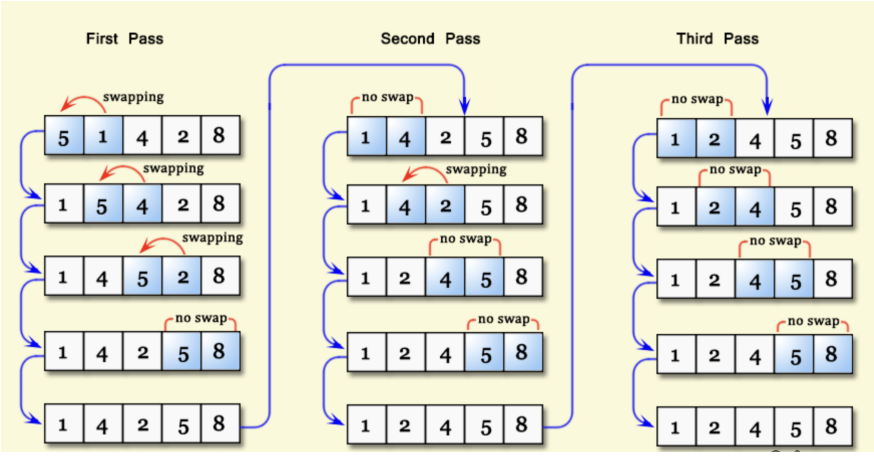

Figure 3: Bubble sort algorithm illustration

Implementation of code in C++ as well as the console output is shown below.

```
void BubbleSort::sort(std::vector<int> &vec)
{
    Utils::print(vec);
    bool sorted{false};
    while (not sorted)
    {
        Utils::print(vec);
        bool swapped{false};
        for (uint8_t i = 0; i < vec.size() - 1; i++)
        {
            if (vec[i] > vec[i + 1])
            {
                int temp = vec[i];
                vec[i] = vec[i + 1];
                vec[i + 1] = temp;
                swapped = true;
            }
        }
        sorted = not swapped;
    }
    Utils::print(vec);
}
```

```
konradmarkowski@Konrads-MacBook-Air algorithms % ./makefile.sh
5 1 4 2 8
5 1 4 2 8
1 4 2 5 8
1 2 4 5 8
1 2 4 5 8
```

Considering time complexity bubble sort performs bad among other sorting algorithms. Selection sort implementation has while loop in which for loop is nested therefore bubble sort has average case time complexity of $\Theta(n^2)$. For the best case where given array is already sorted algorithm can spot it making it $\Omega(n)$ time complexity as well. Space complexity of bubble sort is constant and does not dependent of given array size $O(1)$. All indicators considering time and space complexity are shown in table (Fig. 3) below.

time complexity worst case	$O(n^2)$
time complexity average case	$\Theta(n^2)$
time complexity best case	$\Omega(n)$
space complexity	$O(1)$

Table 3: Selection sort algorithm time and space complexity

2.4 Merge Sort

Illustration of selection sort algorithm is shown below (Fig. 4). It's based on divide and conquer method which in that case is based on dividing given array into two smaller arrays recursively. Division process takes place until arrays are indivisible and more. Then merging starts after comparing each number so newly created array is sorted. Merging takes place until all sub-arrays are merged thus the array is sorted. Steps executed by the sorting function are shown below:

- in the first step input array of size 4 is divided into two sub-arrays of size 2 and the division recursive process is continued until sub-arrays are of size 1. This process in done by *sort*() function shown below.
- already divided sub-arrays are merged back into arrays one by one forming firstly sub-arrays of size 2 and then output array of size 4. This process is done by *merge*() function.

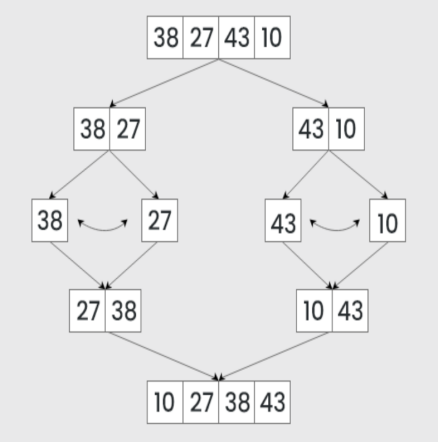

Figure 4: Merge sort algorithm illustration

Implementation of code in C++ as well as the console output is shown below.

```
void MergeSort::sort(std::vector<int> &vec)
{
    Utils::print(vec);
    sort(vec, 0, vec.size() - 1);
    Utils::print(vec);
}

void MergeSort::sort(std::vector<int> &vec,
    const uint8_t begin, const uint8_t end)
{
    if (begin >= end)
    {
        return;
    }
    uint8_t mid = (begin + end) / 2;
    sort(vec, begin, mid);
    sort(vec, mid + 1, end);
    merge(vec, begin, mid, end);
}
```

```
void MergeSort::merge(std::vector<int> &vec,
    const uint8_t begin, const uint8_t mid, const uint8_t end)
{
    std::vector<int> leftSubArray{};
    std::vector<int> rightSubArray{};
    uint8_t leftSubArraySize = mid - begin + 1;
    uint8_t rightSubArraySize = end - mid;
    for (uint8_t i = 0; i < leftSubArraySize; i++)
    {
        leftSubArray.push_back(vec[begin + i]);
    }

    for (uint8_t j = 0; j < rightSubArraySize; j++)
    {
        rightSubArray.push_back(vec[mid + 1 + j]);
    }

    Utils::print(leftSubArray);
    Utils::print(rightSubArray);

    uint8_t leftSubArrayIndex = 0;
    uint8_t rightSubArrayIndex = 0;
    uint8_t mergedArrayIndex = begin;
    while (leftSubArrayIndex < leftSubArray.size() and
           rightSubArrayIndex < rightSubArray.size())
    {
        if (leftSubArray[leftSubArrayIndex] <=
            rightSubArray[rightSubArrayIndex])
        {
            vec[mergedArrayIndex] = leftSubArray[leftSubArrayIndex];
            leftSubArrayIndex++;
        }
        else
        {
            vec[mergedArrayIndex] = rightSubArray[rightSubArrayIndex];
            rightSubArrayIndex++;
        }
        mergedArrayIndex++;
    }

    while (leftSubArrayIndex < leftSubArray.size())
    {
        vec[mergedArrayIndex] = leftSubArray[leftSubArrayIndex];
        leftSubArrayIndex++;
        mergedArrayIndex++;
    }

    while (rightSubArrayIndex < rightSubArray.size())
    {
       vec[mergedArrayIndex] = rightSubArray[rightSubArrayIndex];
        rightSubArrayIndex++;
        mergedArrayIndex++;
    }
}
```

```
konradmarkowski@Konrads-MacBook-Air algorithms % ./makefile.sh
38 27 43 10
38
27
43
10
27 38
10 43
10 27 38 43
```

Considering time complexity merge sort performs well among other sorting algorithms. Merge sort implementation is based on divide and conquer method thus average case time complexity of $\Theta(n \log_2 n)$. For the best case where given array is already sorted algorithm can't spot it making it $\Omega(n \log_2 n)$ time complexity as well. Space complexity of merge sort is linearly dependant on input arrays size $O(n)$ which is a result of sub-arrays creation by the *merge*() function. All indicators considering time and space complexity are shown in table (Fig. 4) below.

time complexity worst case	$O(n \log_2 n)$
time complexity average case	$\Theta(n \log_2 n)$
time complexity best case	$\Omega(n \log_2 n)$
space complexity	$O(n)$

Table 4: Merge sort algorithm time and space complexity

2.5 Quick Sort

Illustration of quick sort algorithm is shown below (Fig. 5). It's based on divide and conquer method which in that case is based on choosing pivot point (most likely at the end of an array) finding its correct position and repeating same process recursively around that pivot point. In this scenario we are sure that numbers to the right of of pivot are larger than it as well as numbers to the left of pivot are smaller than it. No sub-arrays are created and partitioning is based purely on iterators. Steps executed by the sorting function are shown below:

- at first the pivot is chosen at the end of the array which in that case 13,
- two iterators are initialised j at the beginning and i which point one index behind j,
- if number on which j iterator is pointing is smaller than pivot iterator i is incremented and number witch index j and i are swapped.
- when j comes to an end $i + 1$ becomes the index at which pivot should bo so the swap is made,
- given operation is done recursively for the part to the left of pivot and to the right of pivot.

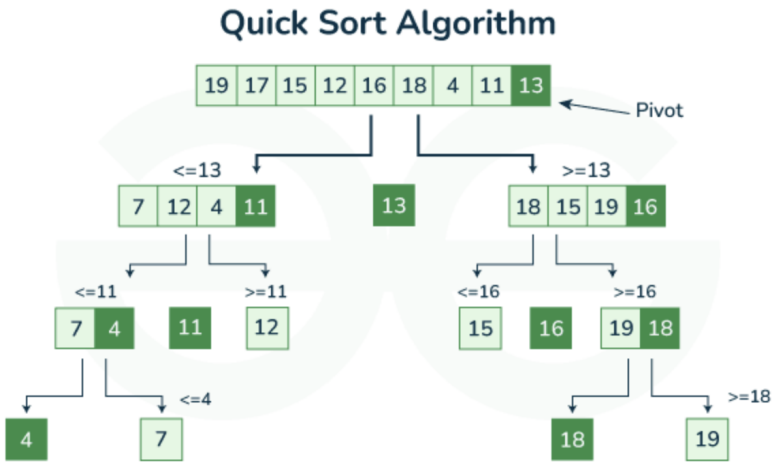

Figure 5: Quick sort algorithm illustration

```
void QuickSort::sort(std::vector<int> &vec)
{
    Utils::print(vec);
    sort(vec, 0, vec.size() - 1);
    Utils::print(vec);
}
```

```
void QuickSort::sort(std::vector<int> &vec,
    const int8_t begin, const int8_t end)
{
    if (begin < end)
    {
        Utils::print(vec);
        int8_t pivotIndex = partition(vec, begin, end);
        sort(vec, begin, pivotIndex - 1);
        sort(vec, pivotIndex + 1, end);
    }
}

int8_t QuickSort::partition(std::vector<int> &vec,
    const int8_t begin, const int8_t end)
{
    int pivot = vec[end];
    int8_t i = begin - 1;
    for (int8_t j = begin; j < end; j++)
    {
        if (vec[j] < pivot)
        {
            i++;
            int temp = vec[j];
            vec[j] = vec[i];
            vec[i] = temp;
        }
    }
    vec[end] = vec[i + 1];
    vec[i + 1] = pivot;
    return i + 1;
}
```

Considering time complexity quick sort performs well among other sorting algorithms. Quick sort implementation is based on divide and conquer method thus average case time complexity of $\Theta(n \log_2 n)$. By average case its consider that numbers in input array are ordered in a random way. For the best case where given array is already sorted algorithm can't spot it making it $\Omega(n \log_2 n)$ time complexity as well. At the worst case however in which always pivot is the biggest or smallest number in the array time complexity become $\Omega(n^2)$. In that case insertion sort can perform better than quick sort. Space complexity of quick sort is linearly dependant on input arrays size $O(n)$ which is a result of sub-arrays creation by the *merge*() function. All indicators considering time and space complexity are shown in table (Fig. 5) below.

time complexity worst case	$O(n^2)$
time complexity average case	$\Theta(n \log_2 n)$
time complexity best case	$\Omega(n \log_2 n)$
space complexity	$O(n)$

Table 5: Quick sort algorithm time and space complexity

2.6 Heap Sort

Algorithm of heap sort is similar to the selection sort however additional data structure is used - binary heap. Implementation is based on building max heap in which every parent node is larger than the child nodes. Function which is recursively creates max heap starting from highest node and going to the very bottom of given tree is "maxHeapify" method which behaviouir on specific node (in that case node with index 2) is shown below (Fig. 6). Implementation in C++ in shown below.

```
void HeapSort::maxHeapify(
    std::vector<int> &vec,
    const uint8_t parent,
    const uint8_t end)
{
    uint8_t largest = parent;
    uint8_t left = parent * 2 + 1;
    uint8_t right = parent * 2 + 2;
    if (left < end and vec[left] > vec[largest])
    {
            largest = left;
    }
    if (right < end and vec[right] > vec[largest])
    {
            largest = right;
    }
    if (largest != parent)
    {
            Utils::swap(vec[largest], vec[parent]);
            maxHeapify(vec, largest, end);
    }
}
```

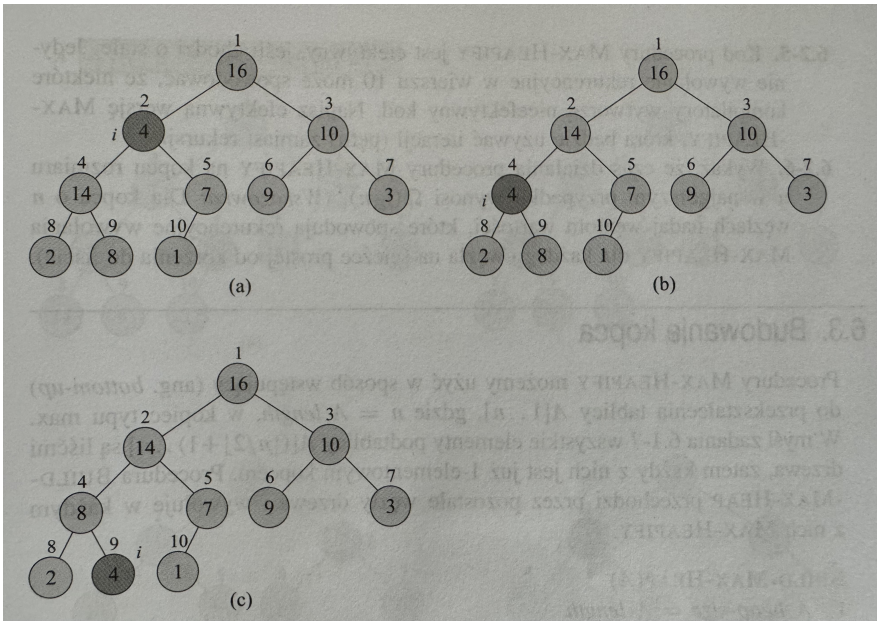

Figure 6: MaxHeapify method illustration

In that paragraph behaviour of "maxHeapify" will be discussed in depth.

- first the function is called with parent node with index 2 and end at index 10,
- then 4 is compared to 14 and 7 from which the bigger one, 14 is swapped with 4,
- second function call starts with parent node with index 4 and end at index 10,
- then 4 is compared to 2 and 8 from which the bigger one, 8 is swapped with 4.

In order to execute heap sort first array must be converted into max heap which is done by swapping values in place and not using any additional container. The base idea is to make sure that every value in parent node in tree is bigger than the value of left and right child (definition of max heap). In order to achieve that "maxHeapify" method is called for every parent node in structure. Behaviour of "buildMaxHeap" function is shown below (Fig. 7). Implementation of "buildMaxHeap" is shown below.

```
void HeapSort::buildMaxHeap(std::vector<int> &vec)
{
    for (int8_t i = (vec.size() - 1) / 2; i >= 0; i--)
    {
            maxHeapify(vec, i, vec.size());
    }
}
```

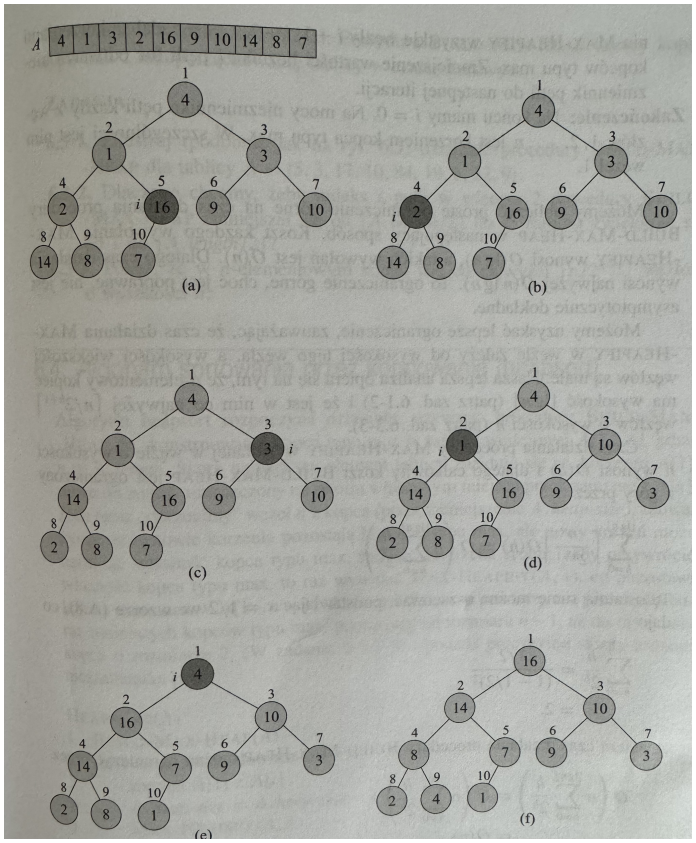

Figure 7: CreateMaxHeapify method illustration

Finally when it comes to heap sort algorithm which behaviour is shown below (Fig. (8). Implementation in C++ is shown below.

```
void HeapSort::sort(std::vector<int> &vec)
{
    Utils::print(vec);
    buildMaxHeap(vec);
    for (int8_t i = vec.size() - 1; i > 0; i--)
    {
            Utils::swap(vec[0], vec[i]);
            maxHeapify(vec, 0, i);
    }
    Utils::print(vec);
}
```

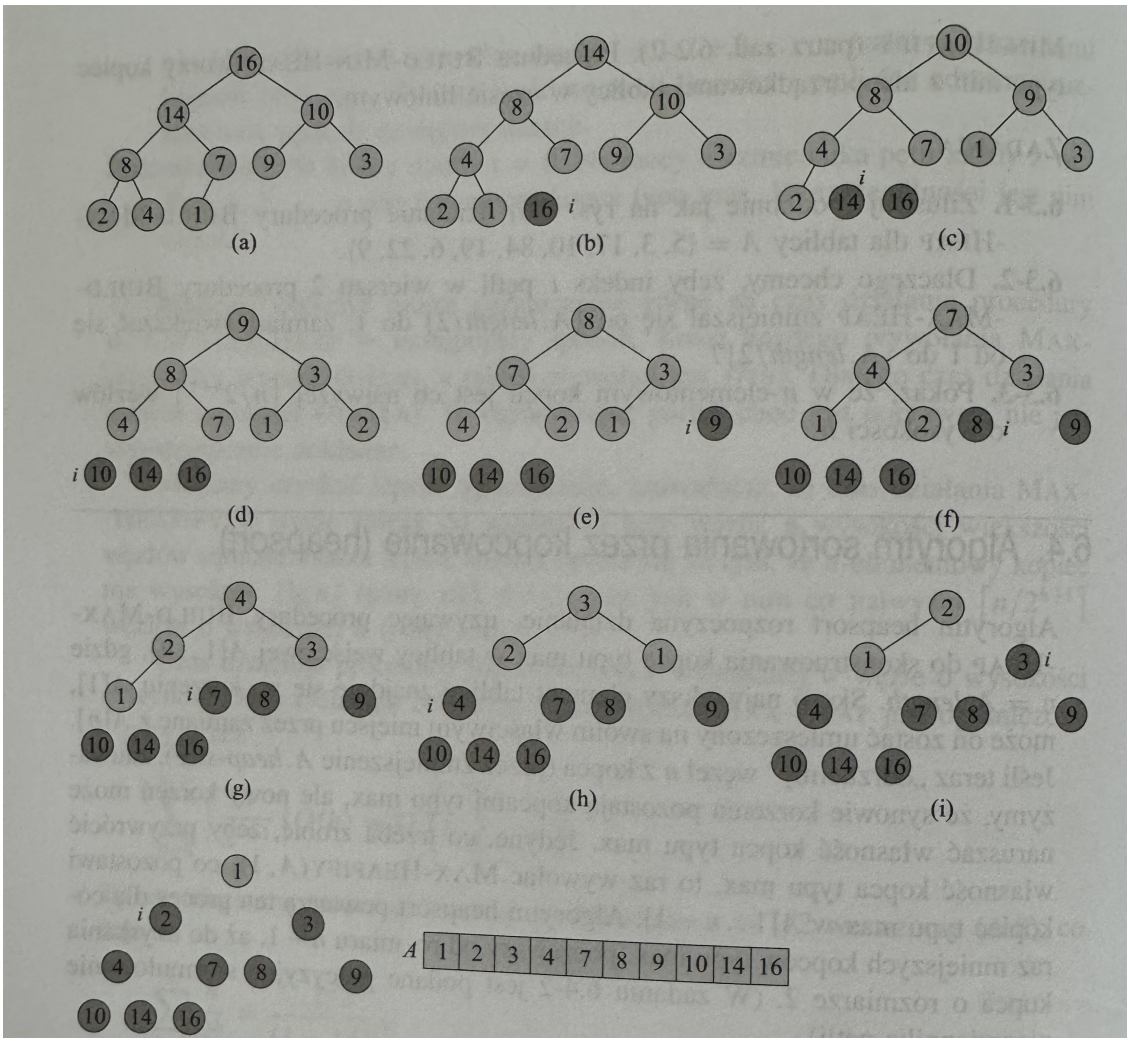

Figure 8: Heap sort algorithm illustration

In that paragraph behaviour of heap sort algorithm will be discussed in depth.

- first the creation of max heap takes place,
- next the loop starts with i = size - 1,
- swap between first and the last index is made and now the max element witch value 16 is at the end of the list,
- next iteration starts with i = size - 2 and so on ...

Considering time complexity heap sort performs well among other sorting algorithms. Heap sort implementation is based on binary heap data structure thus average case time complexity of $\Theta(n \log_2 n)$ as well as the worst case $O(n \log_2 n)$ and the best case $\Omega(n \log_2 n)$. This algorithm's time complexity is independent on scenario. Space complexity of heap sort is independent on input arrays size $O(1)$ which is a result of creating a binary max heap in place. All indicators considering time and space complexity are shown in table (Tab. 6) below.

time complexity worst case	$O(n \log_2 n)$
time complexity average case	$\Theta(n \log_2 n)$
time complexity best case	$\Omega(n \log_2 n)$
space complexity	$O(1)$

Table 6: Heap sort algorithm time and space complexity

2.7 Comparison

Sorting Algorithm	Use Case
Bubble Sort	Educational purposes, small datasets, nearly sorted data
Selection Sort	Small datasets, memory-constrained environments
Insertion Sort	Nearly sorted datasets, small datasets
Merge Sort	Large datasets, stability-required scenarios
Quick Sort	Large datasets, in-place sorting, average case performance
Heap Sort	Large datasets, in-place sorting, priority-based systems

Sorting Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$

3 Data Structures Comparison

Data Structure	Use Case
std::array	Fixed-size arrays, efficient random access
std::vector	Dynamic arrays, efficient random access
std::list	Doubly linked lists, frequent insertions/deletions
std::stack	LIFO operations, managing function calls, undo systems
std::queue	FIFO operations, task scheduling
std::priority_queue	Priority-based scheduling, Dijkstra's algorithm
std::set	Unique, ordered elements, range-based queries
std::unordered_set	Unique, unordered elements, hash-based lookups
std::map	Key-value pairs, ordered, range-based queries
std::unordered_map	Key-value pairs, unordered, hash-based lookups

4 Time And Space Complexity

4.1 Standard Array

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
```

4.1.1 Access Element

```
std::cout << arr[2] << std::endl;
std::cout << arr.at(2) << std::endl;
```

4.1.2 Access Value

```
arr[3] = 10;
arr.at(1) = 20;
```

4.1.3 Iteration

```
for (int i = 0; i < arr.size(); i++)
    std::cout << arr[i] << " ";

for (int val : arr)
    std::cout << val << " ";
```

4.1.4 Copying

```
std::array<int, 5> arr2 = arr;
```

4.1.5 Fill

```
arr.fill(0);
```

4.1.6 Summary

Operation	Time Complexity	Space Complexity
Access element	$O(1)$	$O(1)$
Assign value	$O(1)$	$O(1)$
Iteration	$O(n)$	$O(1)$
Copying	$O(n)$	$O(n)$
Fill	$O(n)$	$O(1)$

4.2 Standard Vector

```
std::vector<int> vec = {10, 20, 30, 40, 50};
```

4.2.1 Access Element

```
std::cout << vec[2] << std::endl;
std::cout << vec.at(2) << std::endl;
```

4.2.2 Access Value

```
vec[3] = 10;
vec.at(1) = 20;
```

4.2.3 Iteration

```
for (int i = 0; i < vec.size(); i++)
    std::cout << vec[i] << " ";

for (int val : vec)
    std::cout << val << " ";
```

4.2.4 Copying

```
std::vector<int> vec2 = vec;
```

4.2.5 Fill

```
std::fill(vec.begin(), vec.end(), 100);
```

4.2.6 Push Back

```
vec.push_back(10);
```

4.2.7 Pop Back

```
vec.pop_back();
```

4.2.8 Insert in the Middle or Beginning

```
vec.erase(vec.begin());
vec.erase(vec.begin() + 1);
```

4.2.9 Erase in the Middle or Beginning

```
vec.insert(vec.begin(), 5);
vec.insert(vec.begin() + 2, 7);
```

4.2.10 Resize

```
vec.resize(7);
vec.resize(2);
```

4.2.11 Summary

Worst case happens when memory allocated for vector is not sufficient and new memory allocation is to be made.

Operation	**Time Complexity**	**Space Complexity**
Access element	$O(1)$	$O(1)$
Assign value	$O(1)$	$O(1)$
Iteration	$O(n)$	$O(1)$
Copying	$O(n)$	$O(n)$
Fill	$O(n)$	$O(1)$
Push back (amortized)	$O(1)$	$O(1)$
Push back (worst − case)	$O(n)$	$O(n)$
Pop back	$O(1)$	$O(1)$
Insert (middle or beginning)	$O(n)$	$O(n)$
Erase (middle or beginning)	$O(n)$	$O(n)$
Resize	$O(n)$	$O(n)$

4.3 Standard Unordered Set

```
std::unordered_set<int> uset;
```

4.3.1 Insert

```
uset.insert(10);
```

4.3.2 Erase

```
uset.erase(20);

auto it = uset.find(30);
if (it != uset.end()) uset.erase(it);
```

4.3.3 Find

```
auto it = uset.find(30);
```

4.3.4 Check if Empty

```
uset.empty()
```

4.3.5 Size

```
uset.size()
```

4.3.6 Clear

```
uset.clear()
```

4.3.7 Rehash

```
uset.rehash(20);
```

4.3.8 Iteration

```
for (const int &val : uset)
    std::cout << val << " ";
```

4.3.9 Summary

Worst case time complexity scenario happens when hash function is poor and puts multiple elements in the same hash bucket.

For space complexity worst case happens when rehash is triggered when inserting new element. This happens when set exceeds load factor (usually 1.0) when inserting new element. Then allocating new hash set in needed thus linear space complexity.

Operation	**Time Complexity**	**Space Complexity**
$Insert$	$O(1)$ avg, $O(n)$ worst	$O(n)$
$Erase$	$O(1)$ avg, $O(n)$ worst	$O(1)$
$Find$	$O(1)$ avg, $O(n)$ worst	$O(1)$
$Check\ if\ Empty$	$O(1)$	$O(1)$
$Size$	$O(1)$	$O(1)$
$Clear$	$O(n)$	$O(1)$
$Rehash$	$O(n)$	$O(n)$
$Iteration$	$O(n)$	$O(1)$

4.4 Standard Unordered Map

```
std::unordered_map<int, std::string> umap =
    {{1, "One"}, {2, "Two"}, {3, "Three"}};
```

4.4.1 Insert

```
umap.insert({1, "One"});
umap[3] = "Three";
```

4.4.2 Erase

```
umap.erase(2);

auto it = umap.find(3);
if (it != umap.end()) {
    umap.erase(it);
}
```

4.4.3 Find

```
auto it = umap.find(2);
```

4.4.4 Access Element

```
umap[1] = "One";
```

4.4.5 Check if Empty

```
umap.empty(
```

4.4.6 Size

```
umap.size()
```

4.4.7 Clear

```
umap.clear();
```

4.4.8 Rehash

```
umap.rehash(10);
```

4.4.9 Iteration

```
for (const auto& [key, value] : umap)
    std::cout << key << ": " << value << std::endl;
```

4.4.10 Summary

Worst case time complexity scenario happens when hash function is poor and puts multiple elements in the same hash bucket.

For space complexity worst case happens when rehash is triggered when inserting new element. This happens when set exceeds load factor (usually 1.0) when inserting new element. Then allocating new hash set in needed thus linear space complexity.

Operation	**Time Complexity**	**Space Complexity**
Insert	$O(1)$ avg, $O(n)$ worst	$O(n)$
Erase	$O(1)$ avg, $O(n)$ worst	$O(1)$
Find	$O(1)$ avg, $O(n)$ worst	$O(1)$
Access Element	$O(1)$ avg, $O(n)$ worst	$O(1)$
Check if Empty	$O(1)$	$O(1)$
Size	$O(1)$	$O(1)$
Clear	$O(n)$	$O(1)$
Rehash	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(1)$

4.5 Standard Set

```
std::set<int> s;
```

4.5.1 Insert

```
s.insert(10);
```

4.5.2 Erase

```
s.erase(20);

auto it = s.find(30);
if (it != s.end()) s.erase(it);
```

4.5.3 Find

```
auto it = s.find(30);
```

4.5.4 Iteration

```
for (const int& val : s)
    std::cout << val << " ";
```

4.5.5 Size

```
m.size();
```

4.5.6 Clear

```
m.clear();
```

4.5.7 Lower or Upper Bound

```
auto lb = s.lower_bound(25);
auto ub = s.upper_bound(25);
```

4.5.8 Summary

Operation	Time Complexity	Space Complexity
Insert	$O(\log n)$	$O(1)$
Erase	$O(\log n)$	$O(1)$
Find	$O(\log n)$	$O(1)$
Iteration (overall)	$O(n)$	$O(1)$
Size	$O(1)$	$O(1)$
Clear	$O(n)$	$O(1)$
Lower Bound / Upper Bound	$O(\log n)$	$O(1)$

4.6 Standard Map

```
std::map<int, std::string> m;
```

4.6.1 Insert

```
m.insert({1, "One"});
m[2] = "Two";
```

4.6.2 Erase

```
m.erase(2);

auto it = m.find(3);
if (it != m.end()) m.erase(it);
```

4.6.3 Find

```
auto it = m.find(3);
```

4.6.4 Iteration

```
for (const int& val : m)
    std::cout << val << " ";
```

4.6.5 Size

```
s.size();
```

4.6.6 Clear

```
s.clear();
```

4.6.7 Lower or Upper Bound

```
auto lb = m.lower_bound(25);
auto ub = m.upper_bound(25);
```

4.6.8 Summary

Operation	Time Complexity	Space Complexity
Insert	$O(\log n)$	$O(1)$
Erase	$O(\log n)$	$O(1)$
Find	$O(\log n)$	$O(1)$
Access Element	$O(\log n)$	$O(1)$
Iteration (overall)	$O(n)$	$O(1)$
Size	$O(1)$	$O(1)$
Clear	$O(n)$	$O(1)$
Lower Bound / Upper Bound	$O(\log n)$	$O(1)$

4.7 Standard Linked List

```
std::list<int> lst;
```

4.7.1 Insert

```
lst.push_front(10);
lst.push_back(20);
lst.insert(lst.begin() + 2, 15);
```

4.7.2 Erase

```
lst.pop_front();
lst.pop_back();
lst.erase(lst.begin() + 2);
```

4.7.3 Access

```
auto it = lst.begin() + 1;
std::advance(lst.begin(), 1);
```

4.7.4 Iteration

```
for (int val : lst)
    std::cout << val << " ";
```

4.7.5 Size

```
lst.size();
```

4.7.6 Clear

```
lst.clear();
```

4.7.7 Summary

Operation	Time Complexity	Space Complexity
Insert (at front or back)	$O(1)$	$O(1)$
Insert (at arbitrary position)	$O(n)$	$O(1)$
Erase (at front or back)	$O(1)$	$O(1)$
Erase (at arbitrary position)	$O(n)$	$O(1)$
Access by iterator	$O(1)$	$O(1)$
Access by index (random access)	$O(n)$	$O(1)$
Iteration (overall)	$O(n)$	$O(1)$
Size	$O(1)$	$O(1)$
Clear	$O(n)$	$O(1)$

4.8 Standard Stack

```
std::stack<int> s;
```

4.8.1 Push

```
s.push(10);
```

4.8.2 Pop

```
s.pop();
```

4.8.3 Top

```
s.top();
```

4.8.4 Size

```
s.size()
```

4.8.5 Empty

```
s.wmpty()
```

4.8.6 Clear

```
while (!s.empty())
    s.pop();
```

4.8.7 Summary

Operation	Time Complexity	Space Complexity
Push	$O(1)$	$O(1)$
Pop	$O(1)$	$O(1)$
Top	$O(1)$	$O(1)$
Size	$O(1)$	$O(1)$
Empty	$O(1)$	$O(1)$
Clear (*indirectly via repeated Pop*)	$O(n)$	$O(1)$

4.9 Standard Queue

```
std::queue<int> q;
```

4.9.1 Push

```
q.push(10);
```

4.9.2 Pop

```
q.pop();
```

4.9.3 Front

```
q.front();
```

4.9.4 Back

```
q.back();
```

4.9.5 Size

```
q.size()
```

4.9.6 Empty

```
q.empty()
```

4.9.7 Clear

```
while (!q.empty())
    q.pop();
```

4.9.8 Summary

Operation	Time Complexity	Space Complexity
Push (*enqueue*)	$O(1)$	$O(1)$
Pop (*dequeue*)	$O(1)$	$O(1)$
Front	$O(1)$	$O(1)$
Back	$O(1)$	$O(1)$
Size	$O(1)$	$O(1)$
Empty	$O(1)$	$O(1)$
Clear (*indirectly via repeated Pop*)	$O(n)$	$O(1)$

4.10 Standard Priority Queue

```
std::priority_queue<int> pq;
```

4.10.1 Push

```
pq.push(10);
```

4.10.2 Pop

```
pq.pop();
```

4.10.3 Top

```
pq.top();
```

4.10.4 Size

```
pq.size()
```

4.10.5 Empty

```
pq.empty()
```

4.10.6 Clear

```
while (!pq.empty())
    pq.pop();
```

4.10.7 Summary

Operation	Time Complexity	Space Complexity
Push (insert)	$O(\log n)$	$O(1)$
Pop (remove max or min)	$O(\log n)$	$O(1)$
Top	$O(1)$	$O(1)$
Size	$O(1)$	$O(1)$
Empty	$O(1)$	$O(1)$
Clear (indirectly via repeated Pop)	$O(n \log n)$	$O(1)$

4.11 Searching Algorithms

Algorithm	Time Complexity	Space Complexity	Best Use Case
Linear Search	$O(n)$	$O(1)$	*Small or unsorted datasets*
Binary Search	$O(\log n)$	$O(1)$	*Large sorted datasets*

4.12 Sorting Algorithms

Algorithm	Time Complexity (Best)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$ $(in-place)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$