OPAL

Overpowered Algorithm Library

Rafał Żelazko

April 29, 2023

Contents

1 Introduction

1.1 What OPAL is and What it is not

OPAL stands for Overpowered Algorithm Library. It is a collection of algorithms and data structures packaged as a header-only library for easy use with C++ 20 and higher. OPAL, however, is not a replacement for the standard library. It is not meant to replace any of the standard components as it is not optimized for speed in every way possible. It should rather be regarded as a learning tool and reference implementation for people who want to learn about algorithms and data structures.

1.2 Why You Should Choose OPAL

There are many other libraries that provide algorithms or data structures, but most of them are either too complicated or contain too few or no helpful comments at all. First and foremost, OPAL aims to provide an easy to understand implementation built on good coding practices. Every algorithm and structure in OPAL is implemented using modern C++ template techniques. This allows for vast flexibility in use and makes it easier to extend the library with new functionality.

1.3 The Purpose of OPAL

OPAL is a project that I started in order to build a framework that would help me easily create algorithm implementations for lectures at my university. I wanted to refresh my knowledge of modern C++, and this is why I decided to use it as the core programming language in OPAL. I hope that OPAL will be useful to other people as well, and I will be happy to receive any feedback or suggestions.

2 Insertion Sort

2.1 Description

Insertion sort is a simple sorting algorithm that builds the final sorted array linearly. It is much less efficient on large lists than more advanced algorithms such as quicksort or merge sort. Insertion sort, however, provides certain advantages:

- It is in-place: only a constant amount of additional memory is needed.
- It is stable: equal elements will never be reordered.
- It is online: it can sort a list as it receives it.

Because of its stability and portability, insertion sort is often used as a subroutine in more advanced divide-and-conquer algorithms.

2.2 Procedure

The canonical implementation of the algorithm [1]:

```
1: A ← ...
2: for i = 2, 3, ..., LENGTH(A) do
3:     a ← A[i]
4:     j ← i − 1
5:     while j ≠ 0 and a < A[j] do
6:         A[j + 1] ← A[j]
7:         j ← j − 1
8:     end while
9:     A[j + 1] ← a
10: end for
```

1. The procedure starts by focusing each element starting from the second one up to the last one. (line 2)
2. The currently focused element is saved to a separate variable as it might be overwritten in subsequent steps. (line 3)
3. Previous elements are shifted one position to the right until the right spot for the focused value is found. (line 5)
4. Finally, the focused value is inserted in the old place of the most recently shifted element. If no elements were shifted, the focused value will be inserted in the same place where it has been. (line 9)

2.3 Computational Complexity

To analyze the computational complexity of insertion sort, one must count the number of times that both the outer for-loop and the inner while-loop are iterated. The number of outer-loop iterations is equal to the number of elements in the array minus one. The number of inner-loop iterations ranges from one to the number of elements in the array minus one, but it also depends on the order of the elements in the array. However, the average number of inner-loop iterations is approximately equal to the number of elements in the array divided by two. This results in the following formulae:

$$\begin{aligned} N_i &= n - 1 \\ N_j &= \frac{1}{2}n \end{aligned}$$

The best case scenario occurs when the array is already sorted. In this case, the algorithm will perform only N_i outer-loop iterations and no inner-loop iterations at all.

$$\begin{aligned} T(n) &= N_i + N_i \cdot 0 \\ &= n - 1 \\ &= \Omega(n) \end{aligned}$$

The worst case scenario is when the array is sorted in reverse order. In this case, the algorithm will perform N_i outer-loop iterations and each time around N_j more inner-loop iterations.

$$\begin{aligned} T(n) &= N_i + N_i \cdot N_j \\ &= (n-1) + \frac{1}{2}n(n-1) \\ &= \frac{1}{2}n^2 + \frac{1}{2}n - 1 \\ &= O(n^2) \end{aligned}$$

The algorithm performs similarly in the average case scenario as well. This produces the following table:

Computational Complexity		
Best	**Worst**	**Average**
$\Omega(n)$	$O(n^2)$	n^2

It is worth noting that such low computational complexity in the best case scenario for this algorithm makes it an excellent choice to use with nearly sorted arrays as the performance hit is negligible.

3 Merge Sort

3.1 Description

Merge sort is a more advanced sorting algorithm that takes advantage of the famous divide-and-conquer strategy. Because of this, merge sort is much more efficient than insertion sort and other simple algorithms. Merge sort provides several advantages:

- It is stable: equal elements will never be reordered.
- It is parallelizable: various parts of the algorithm can be run concurrently without data races or loss of stability.
- It is optimal: the average time complexity is $O(n \log n)$.

However, merge sort is not without its drawbacks. It is not in-place as it requires additional memory for merging. It is also not online and is way slower with small arrays than most $O(n^2)$ algorithms.

3.2 Procedure

Common implementation of the algorithm:

```
1: A ← ...
2: MERGE-SORT(A, 1, LENGTH(A) + 1)
3: function MERGE-SORT(A, l, u)
4:     if u − l > 1 then
5:         m ← ⌊(l + u) ÷ 2⌋
6:         MERGE-SORT(A, l, m)
7:         MERGE-SORT(A, m, u)
8:         MERGE(A, l, m, u)
9:     end if
10: end function
```

1. The algorithm begins by checking whether the array is long enough and quits instantly if the array is already sorted (i.e. contains a single element). In this implementation u is not included in the sorting range. (line 4)
2. If the array needs to be sorted, it is split into two sub-arrays of roughly the same length, each one of which is sorted recursively using merge sort itself. (lines 6 and 7)

3. Finally, the two sorted sub-arrays are merged into one. The merge procedure is the most important part of the algorithm as it is the major factor in its time complexity. Amazingly, merge can be carried out in $O(n)$ time, which is the reason of merge sort's great performance. (line 8)

```
1: function MERGE(A, l, m, u)
2:     B ← empty array
3:     i ← l
4:     j ← m
5:     while i < m  &  j < u do
6:         if A[i] ≤ A[j] then
7:             B ← B + A[i]
8:             i ← i + 1
9:         else
10:            B ← B + A[j]
11:            j ← j + 1
12:        end if
13:    end while
14:    B ← B + A[i ... m)
15:    B ← B + A[j ... u)
16:    A[l ... u) ← B
17: end function
```

1. The merge procedure begins by allocating an auxiliary buffer that will store the result of merging.

2. Next, two iterators, i and j, are initialized. They keep track of the current progress in two of the sub-arrays. (lines 3 and 4)

3. The two sub-arrays are then merged into one by comparing the elements at indices i and j. The smaller element is added to the buffer and the index of the sub-array it came from is incremented. (line 5)

4. If one of the sub-arrays is exhausted, the remaining elements of the other sub-array are added to the buffer. (lines 14 and 15)

5. Finally, the buffer is copied back to the right position in the original array. (line 16)

3.3 Computational Complexity

The computational complexity of the merge sort is a more complicated matter. The algorithm is based on the divide-and-conquer strategy, which means that it utilizes recursion. On each level of recursion, the problem of size n is divided into two sub-problems of roughly the same size $\frac{n}{2}$. Then, the results of the two sub-problems are joined together in $O(n)$ time. This results in the following recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

The master theorem can be utilized in order to calculate the complexity of this algorithm:

$$\begin{aligned} a, b = 2 &= const \\ \log_b a = \log_2 2 &= 1 \\ f(n) = n = n^1 &= n^{\log_b a} \\ \Rightarrow T(n) &= \Theta(n \log n) \end{aligned}$$

It follows that the worst, the best, and the average case complexity of merge sort all are $\Theta(n \log n)$ which is the best possible performance for any comparison-based sorting algorithm.

$$T(n) = \Theta(n \log n)$$

3.4 Optimizations

In order to push the performance of merge sort even further, two concurrent execution threads can be spawned at each level of recursion. This is possible because both sub-arrays do not overlap at all which in turn eliminates data dependencies. The two threads can then sort their respective sub-arrays independently and let the supervising thread merge the results.

For optimal performance the total number of bottom-level threads should be a little higher than the number of CPU cores as possible, but not too high as scheduling too many concurrent jobs might cause a significant overhead.

Thread count limiting can be implemented by checking whether the sub-array is long enough to be put on two extra threads. If it is not, the sub-array is sorted entirely on the current thread. The array length threshold below which the sub-array is not split can be calculated before the algorithm is executed:

$$N_{jobs} = 2^{\lceil \log_2 N_{threads} \rceil}$$
$$n_{threshold} = \max(n \div N_{jobs}, 1024)$$

Ideally, the worker threads should be stored in a thread pool and reused for subsequent merge sort calls. This way, the overhead of creating and destroying threads is eliminated.

Temporary merge buffers can also be pooled, but for most uses it is enough to just allocate a single buffer of size n at the beginning of the algorithm and reuse it for all merge calls. In this scenario each thread offsets its working area in the buffer to $[l, u)$ which guarantees that each merge operation will get its own isolated piece of the buffer.

Finally, other $O(n^2)$ sorting algorithms can be used as fallback for even smaller sub-arrays as they typically are much faster at solving smaller sorting problems. For this purpose, OPAL makes use of insertion sort with a threshold of $n = 64$ which decimates the running time of the algorithm by 20% on average.

4 Heap Sort

4.1 Description

Heap sort is a $O(n \log n)$ comparison-based sorting algorithm, based on the heap data structure. It is a bit more complicated than single-threaded merge sort and does not exceed in performance. However, heap sort does not require any additional memory and is therefore a good alternative to merge sort when memory is scarce. Some notable properties of heap sort are:

- It is in-place: only a constant amount of additional memory is needed.
- It is not stable: equal elements can be reordered.
- It is optimal: the average time complexity is $O(n \log n)$.

Finally, heap sort is an excellent improvement over $O(n^2)$ sorting algorithms such as insertion sort and selection sort.

4.2 Procedure

The algorithm works by maintaining a max-heap tree that consists of the values in the array. The heap always occupies the front portion of the array and it is used to progressively pick out the largest elements in the tree. At each step, the largest element is moved outside of the heap to the end of the array. Heap sort procedure is built around the `heapify` function that is used to restore the max-heap property of the tree if one of its nodes is misplaced.

1: $A \leftarrow$ ...
2: $n \leftarrow \textsc{length}(A)$
3: $i \leftarrow \lfloor n \div 2 \rfloor$
4: **while** $i \geq 1$ **do**
5: $\quad \textsc{heapify}(A, n, i)$
6: $\quad i \leftarrow i - 1$
7: **end while**
8: **while** $n \geq 2$ **do**
9: $\quad \textsc{swap}(A[1], A[n])$
10: $\quad n \leftarrow n - 1$
11: $\quad \textsc{heapify}(A, n, 1)$
12: **end while**

1. The algorithm begins by finding the index of the last non-leaf node in the tree. Tree is represented as an array containing the elements in level order. (line 3)

2. The tree is converted to a max-heap. This is done by calling `heapify` on each non-leaf node in the heap starting from the bottom of the tree. (line 4)

3. The heap is then sorted by repeatedly reducing the size of the heap by one element and moving the root of it (i.e. the largest element) outside of the heap. (line 9)

4. Since the root is replaced by another element, each time the heap property needs to be restored. This is done by calling `heapify` once on the root node. (line 11)

5. Every time before the heap is shrunk by one element, the largest element in the heap is moved to the end of it. This way, a sorted sequence is progressively built at the end of the array.

```
1:  function HEAPIFY(A, n, i)
2:      l ← 2i
3:      r ← 2i + 1
4:      x ← i
5:      if l ≤ n and A[l] > A[x] then
6:          x ← l
7:      end if
8:      if r ≤ n and A[r] > A[x] then
9:          x ← r
10:     end if
11:     if x ≠ i then
12:         SWAP(A[i], A[x])
13:         HEAPIFY(A, n, x)
14:     end if
15: end function
```

1. The function begins by finding the indices of the left and right child nodes of the current node. (lines 2 and 3)

2. The index of the largest element is initialized to the current node. Subsequently, the left and right child nodes are compared to the current node and the largest one of them is found. (line 4)

3. If the largest element is one of the children, the current node is swapped with it and the function is called recursively on the child node. (lines 11 and 13)

4. After the last recursive call finishes, the heap property is restored.

4.3 Computational Complexity

In order to find the total time complexity of heap sort, we shall first analyze the `heapify` sub-procedure. `heapify` descends one level down during each recursive call, so in the worst case, the total number of recursive calls is equal to the depth of the heap (i.e. a well balanced binary tree), which is $\lfloor \log_2 n \rfloor$. Other than that, `heapify` does not use any loops, so the final complexity of `heapify` is $O(\log n)$.

The loop used to build the max-heap is executed $\lfloor n \div 2 \rfloor$ times, so the time complexity of the heap building step is:

$$\begin{aligned} T(n) &= \sum_{i=1}^{\lfloor n \div 2 \rfloor} O(\log n) \\ &= O(n \log n) \end{aligned}$$

The second loop used to shrink the tree and fix the heap property is executed $n - 1$ times, so the time complexity of the heap sorting step is:

$$\begin{aligned} a &= const. \\ T(n) &= \sum_{i=1}^{n-1} (O(\log n) + a) \\ &= O(n \log n) \end{aligned}$$

Finally, the overall worst-case computational complexity of heap sort is the maximum of the time complexities of the two steps, which is $O(n \log n)$.

Since the heap sorting step is always executed after the max-heap building step, it will always replace the largest element in the heap with one of the smallest elements in the tree. This means that calls to `heapify` during the heap sorting step will always have the same time complexity of $\Theta(\log n)$. This ensures that the total time complexity of heap sort is exactly $\Theta(n \log n)$.

$$T(n) = \Theta(n \log n)$$

Note: If the array only contained the same value repeated multiple times, `heapify` would never be called recursively, so the computational complexity of heap sort would be $\Omega(n)$. In some literature this edge case is taken care of by using inclusive bounds in the `heapify` function to ensure the exact time complexity of $\Theta(n \log n)$.

5 Quick Sort

5.1 Description

Quick sort, similarly to merge sort is a divide-and-conquer algorithm. However in most cases, it is much more efficient than merge sort. One drawback of quick sort is that it is not stable, so it is not a brain-dead replacement for merge sort. Some properties of quick sort:

- It is in-place: only a constant amount of additional memory is needed.
- It is parallelizable: various parts of the algorithm can be run concurrently without data races.
- It is not stable: equal elements can be reordered.
- It is optimal: the average time complexity is $O(n \log n)$.

5.2 Procedure

The algorithm works by recursively partitioning the array into two sub-arrays. The first sub-array contains all elements that are smaller or than equal to the pivot and the second one contains all elements that are greater than or equal to the pivot. Quick sort comes in multiple flavors, meaning that there are multiple ways to choose the pivot. The most common two are the Lomuto partition scheme and the Hoare partition scheme. In OPAL I use the Hoare partition scheme, which is usually more efficient than the Lomuto partition scheme. One notable difference between the two partitioning schemes is that in Hoare partitioning the pivot must be included in the first sub-array, whereas in Lomuto partition scheme it can be left out of both sub-arrays. The procedure for quick sort is as follows:

```
1: A ← ...
2: QUICK-SORT(A, 1, LENGTH(A) + 1)
3: function QUICK-SORT(A, l, u)
4:     if u − l > 1 then
5:         p ← PARTITION-HOARE(A, l, u)
6:         QUICK-SORT(A, l, p)
7:         QUICK-SORT(A, p, u)
8:     end if
9: end function
```

1. The algorithm begins by checking if the sub-array is of size greater than one. (line 4)

2. If the sub-array is of size greater than one, it is partitioned into two sub-arrays. Each sub-array only contains elements that are smaller than or equal to the pivot and greater than or equal to the pivot respectively. (line 5)

3. Each of the two sub-arrays is then recursively sorted. (lines 6 and 7)

In Hoare partitioning the pivot is chosen as the first element in the array. The partitioning is done by iterating the array from both ends and swapping elements that are on the wrong side of the pivot. In code, it is implemented as follows:

```
1: function PARTITION-HOARE(A, l, u)
2:     p ← A[l]
3:     i ← l − 1
4:     j ← u
5:     loop
6:         do
7:             i ← i + 1
8:         while A[i] < p
9:         do
10:            j ← j − 1
11:        while A[j] > p
12:        if i ≥ j then
13:            return j + 1
14:        end if
15:        SWAP(A[i], A[j])
16:    end loop
17: end function
```

1. The function begins by choosing the first element in the array as the pivot. (line 2)

2. The function then initializes two iterators, one before the beginning of the array and one right beyond the last element of the array. (lines 3 and 4)

3. The function then iterates the array from both ends and swaps elements that are on the wrong side of the pivot. (lines 7, 10 and 15)

4. If the two iterators meet, the function returns the index of the pivot. (line 13)

5. The function continues until the two iterators meet, at which point it returns the index after the right iterator as the pivot.

It's important to note that the pivot element must be included in the first sub-array. This is because the element at pivot index will not necessarily end up as the largest element in the first sub-array.

5.3 Computational Complexity

In order to find the total time complexity of quick sort with Hoare partitioning, we need to analyze both the partitioning step and the recursive branching routine.

The infinite loop in the partitioning function is guaranteed to terminate because the two iterators will eventually meet. In particular, they meet after approximately n comparisons. Therefore, the time complexity of the partitioning step is $\Theta(n)$.

After partitioning the array, quick sort recursively sorts the two sub-arrays created by the partitioning step. The time complexity of quick sort can be expressed recursively as:

$$T(n) = \begin{cases} O(1), & n \leq 1 \\ T(k_n - 1) + T(n - (k_n - 1)) + \Theta(n), & n > 1 \end{cases}$$

...where k_n is the index of the pivot for the given array length.

In the worst case, the pivot is either the second or the last element in the array. The former is true, because the furthest the right iterator will ever stop is the first index. Then it will be incremented once just before being returned. As for the latter, we choose the first element as the initial pivot which ensures that the right iterator will be decremented by at least 2. This leads to $k_n = 2$ or $k_n = n$ respectively. That in turn means that in the worst case one of the sub-arrays will have a size of 1. This can be described using the following recurrence relation:

$$T_{worst}(n) = \begin{cases} O(1), & n \leq 1 \\ T(n-1) + \Theta(n), & n > 1 \end{cases}$$

Solving this recurrence relation using the substitution method, we get:

$$T_{worst}(n) = \sum_{i=1}^{n} \Theta(n) = \Theta(n^2)$$

However, this worst-case scenario is very unlikely to occur in practice, especially if we use a good pivot selection method such as the Hoare's scheme. In the average case, both sub-arrays will have sizes of around $\frac{n}{2}$:

$$T_{avg}(n) = \begin{cases} O(1), & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \end{cases}$$

We can solve this recurrence relation using the master theorem:

$$\begin{aligned} T_{avg}(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ a &= 2 \\ b &= 2 \\ f(n) &= \Theta(n) \\ \log_b a &= 1 \\ f(n) &= \Theta(n^{\log_b a}) \\ \implies T(n)_{avg} &= \Theta(n^{\log_b a} \log n) \\ &= \Theta(n \log n) \end{aligned}$$

Therefore, the overall worst-case computational complexity of quick sort with Hoare partitioning is $O(n^2)$, and the average-case computational complexity is $O(n \log n)$.

Computational Complexity		
Best	**Worst**	**Average**
$\Omega(n \log n)$	$O(n^2)$	$n \log n$

5.4 Optimizations

To improve the performance of quick sort, we can switch to a different sorting algorithm at some depth of recursion. Such a technique may additionally reduce the worst-case complexity of quick sort to $O(n \log n)$.

In OPAL, quick-sort can switch to two different algorithms at different points of recursion. The first algorithm is insertion sort, which is used for small sub-arrays whose size is less than 64. This constant is taken directly from OPAL's merge sort. (3.4) The second algorithm is heap sort, which is used if the size of the array drops below a dynamic threshold:

$$n_{threshold} = 2\lfloor \log_2 n \rfloor$$

Please note that this exact technique does not reduce the worst-case complexity of quick sort, but it has significantly improved the average-case performance during testing.

As a third mean of improving the performance of quick sort, OPAL makes use of multithreading. Namely, quick sort offloads each new level of recursion to two new threads. This technique is exactly the one used in merge sort along with the thread count limiting, which is done in an identical manner. (3.4)

6 Counting Sort

6.1 Description

Counting sort is a different kind of sorting algorithm. It is not comparison-based, but rather uses the actual values of the elements in the input list to order them. It is only applicable to lists of integers, but it is extremely fast and stable. Notable properties of counting sort include:

- It is not in-place: it must allocate auxiliary buffers.
- It is stable: equal elements will never be reordered.
- It is linear: it runs in linear time.

6.2 Procedure

The canonical implementation of the algorithm [1]:

1: $A \leftarrow ...$
2: $C \leftarrow$ array of $\text{MAX}(A) + 1$ zeros
3: **for** $i = 1, 2, ..., \text{LENGTH}(A)$ **do**
4: $\quad C[A[i]] \leftarrow C[A[i]] + 1$
5: **end for**
6: **for** $i = 2, 3, ..., \text{LENGTH}(C)$ **do**
7: $\quad C[i] \leftarrow C[i] + C[i-1]$
8: **end for**
9: $B \leftarrow$ array of size $\text{LENGTH}(A)$
10: **for** $i = \text{LENGTH}(A), \text{LENGTH}(A) - 1, ..., 1$ **do**
11: $\quad B[C[A[i]]] \leftarrow A[i]$
12: $\quad C[A[i]] \leftarrow C[A[i]] - 1$
13: **end for**
14: $A \leftarrow B$

1. The procedure starts by allocating an array of counters. The size of the array is equal to the maximum value in the input array plus one so that it can fit all possible values from zero up to the maximum value. (line 2)
2. The algorithm then counts the number of occurrences of each value in the input array. (line 3)
3. The algorithm then computes the number of elements that are less than or equal to each value in the input array. The resulting counts are also

the correct indices for the last occurrence of each value in the output array. (line 6)

4. The algorithm then initializes an output buffer of the same size as the input array. (line 9)

5. The algorithm then iterates over the input array in reverse order. For each element, it places the element in the output array at the index given by the corresponding counter. It then decrements the counter to point to the previous possible place for the same value. (line 10)

6. Finally, the output array is copied back into the input array. (line 14)

6.3 Computational Complexity

As mentioned before, counting sort is a linear time algorithm. The first loop iterates over the input array once, then the second loop iterates over the counters, and finally the third loop iterates over the input array again. This results in the following formulae:

$$\begin{aligned} T(n) &= N_i + N_j + N_k \\ &= n + k + n \\ &= 2n + k \\ &= O(n + k) \end{aligned}$$

...where n is the length of the input array and k is the maximum value in the input array. This also means that the algorithm has the exact computational complexity of $\Theta(n + k)$.

$$T(n) = \Theta(n + k)$$

6.4 Sorting Non-Integer Arrays

Counting sort is only applicable to lists of integers. However, it is possible to sort lists of non-integer values using counting sort by first mapping the values to integers. This can be done by using a hash function to map the values to integers. The hash function must be able to map each value to non-negative integers. OPAL for this purpose allows the user to define a custom indexer function:

```
1: function INDEXER(x)
2:     return non-negative integer representation of x
3: end function
```

OPAL's counting sort implementation is also able to automatically generate the indexer function for integer types. This enables the algorithm to sort negative integers without requiring any additional configuration.

7 Radix Sort

7.1 Description

Radix sort which works by sorting numbers digit by digit. This means that, like counting sort, it is only applicable to arrays of non-negative integers. However, it is extremely fast with huge arrays. Some characteristic aspects of radix sort are:

- It is stable: equal elements will never be reordered.
- It is linear: it runs in linear time.

7.2 Procedure

Radix sort is usually implemented using counting sort as a subroutine. The algorithm first sorts the input array by the least significant digit, then by the second least significant digit, and so on until the most significant digit. This can be described using an extremely simplified pseudocode:

1: $A \leftarrow ...$
2: $d \leftarrow$ number of digits in the maximum value in A
3: **for** $i = d, d-1, ..., 1$ **do**
4: stable-sort A while identifying its elements by their i-th digit
5: **end for**

It is important for the sorting subroutine to be stable, otherwise the algorithm will not work as expected because sorting by any digit will scramble previous work.

7.3 Computational Complexity

The computational complexity of radix sort mainly depends on the sorting subroutine. Usually it is counting sort, which has a computational complexity of $\Theta(n+k)$, where n is the length of the input array and k is the maximum value in the input array. In the case of radix sort k is equal to the chosen radix. Radix sort uses a single loop that iterates over the number of digits in the maximum value in the input array. This results in the following formulae:

$$\begin{aligned} T(n) &= \sum_{i=1}^{d}(n+k) \\ &= d(n+k) \\ &= \Theta(d(n+k)) \end{aligned}$$

...where d is the number of digits in the maximum value in the input array and k is the radix. This leads to the conclusion that the computational complexity of radix sort is precisely $\Theta(d(n+k))$.

$$T(n) = \Theta(d(n+k))$$

7.4 Optimizations

One optimization radix sort can take advantage of is to use a radix different than 10. Counting sort has to allocate a temporary array of k counters, but this buffer can pre-cached before sorting and thus it is feasible to allocate a larger one. This is why radix sort benefits from using a larger radix. OPAL by default uses a radix of 256 which greatly improves performance of radix sort.

Additionally, the output buffer used by counting sort can similarly be pre-allocated and reused between iterations.

Double-buffering can be used to avoid copying the output buffer back to the input buffer after each iteration. This is done by alternating between using the input buffer as the output buffer and vice versa.

7.5 Sorting Non-Integer Arrays

Similarly to counting sort, radix-sort can employ hash functions to sort non-integer arrays. See 6.4 for details.

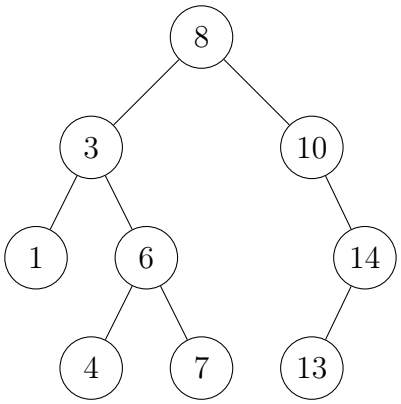
8 Binary Search Tree

8.1 Description

Binary search tree (BST) is a data structure that allows for efficient lookup, insertion and deletion of elements. It is a binary tree in which all nodes in the left subtree of a node have a value which is less than the value of the node, and all nodes in the right subtree have a value which is greater than the value of the node.

$$\forall N \in BST : \begin{cases} \text{VALUE}(N) < \text{MIN}(\text{RIGHT}(N)) \\ \text{VALUE}(N) > \text{MAX}(\text{LEFT}(N)) \end{cases}$$

This allows for efficient lookup, insertion and deletion of elements because each of those operations benefits from binary search.



Example of a BST

8.2 Binary Search

Binary search is what allows for efficient lookup in a binary search tree. It works by comparing the value of the node with the value of the element which in conjunction with the property of BST allows for extremely efficient search in logarithmic time.

```
1: function BINARY-SEARCH(N, x)
2:     if N = 0 or VALUE(N) = x then
3:         return N
4:     end if
5:     if x < VALUE(N) then
6:         return BINARY-SEARCH(LEFT(N), x)
```

```
    else
        return BINARY-SEARCH(RIGHT(N), x)
    end if
end function
```

8.3 Insertion

BST insertion is carried out by first finding the place for the new node using binary search, and then inserting the node in the place of a leaf node.

```
function INSERT(N, x)
    if N = 0 then
        N ← NODE(x)
    else if x < VALUE(N) then
        INSERT(LEFT(N), x)
    else
        INSERT(RIGHT(N), x)
    end if
end function
```

8.4 Erasure

BST erasure is done in three different ways depending on the number of children of the node to be erased. If the node has no children, it is simply removed from the tree. If the node has one child, the child is moved to the place of the node. And if the node has both children, the node is replaced with its in-order successor, which is the smallest node in the right subtree of the node.

```
function ERASE(N, x)
    if N = 0 then
        return
    end if
    if x < VALUE(N) then
        ERASE(LEFT(N), x)
    else if x > VALUE(N) then
        ERASE(RIGHT(N), x)
    else
        if LEFT(N) = 0 and RIGHT(N) = 0 then
            N ← 0
        else if LEFT(N) = 0 then
            N ← RIGHT(N)
```

```
14:         else if RIGHT(N) = 0 then
15:             N ← LEFT(N)
16:         else
17:             N ← NEXT(N)
18:         end if
19:     end if
20: end function
```

8.5 Implementation

OPAL implements BST as an STL-compatible container. It is structured as a tree in which each node has a pointer to its parent, and two unique pointers to its children. Keeping track of node's parent allows for efficient insertion and deletion of values, because this way the operations do not have to remember the parent node. The parent pointer also enables `bst_iterator` to easily find path to the next node in order.

Node

```
template <typename Value>
class bst_node {
public:
    Value                    value;
    bst_node                *parent;
    unique_ptr<bst_node> left, right;

    // ...
};
```

Note: `bst_node` is not a part of interface of the container, and is only used internally. `bst_iterator` is a front-end to `bst_node`.

Tree Container

```
template <typename Value>
class bst {
public:
    // ...

private:
    unique_ptr<bst_node<Value>> root;
};
```

OPAL also provides a `bst_iterator`, which is an STL-compatible iterator. It is implemented as a wrapper around a pointer to the current node

and the root of the tree. It also stores the type of the iterator, which is used to determine how to walk to the next node in order. The pointer to the root is needed to correctly handle end iterator which by design points to a `nullptr`.

Iterator

```
enum class bst_iterator_type {
    in_order,
    pre_order,
    post_order
};

template <typename Value>
class bst_iterator {
public:
    // ...

private:
    bst_node<Value>  *root;
    bst_node<Value>  *node;
    bst_iterator_type type;
};
```

Note: The iterator walks to the next node by following branches around the tree. This is not the most efficient way to execute a full traversal of the tree, but it allows for most flexibility. If the user wants to execute a full traversal with the highest performance possible, they can do so by recursively using `bst_iterator::left` and `bst_iterator::right` methods.

9 References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein; *Introduction to Algorithms*; MIT Press, 2001