

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/340806238>

Black-White Array: A New Data Structure for Dynamic Data Sets

Preprint · April 2020

CITATIONS
0

READS
114

1 author:



Z. George Mou
Tigma

24 PUBLICATIONS 175 CITATIONS

SEE PROFILE

Black-White Array: A New Data Structure for Dynamic Data Sets

Z. George Mou
gmou5813@gmail.com

Abstract

A new array based data structure named black-white array (BWA) is introduced as an effective and efficient alternative to the list or tree based data structures for dynamic data set. It consists of two sub-arrays, one white and one black of half of the size of the white. Both of them are conceptually partitioned into segments of different ranks with the sizes grow in geometric sequence. The layout of BWA allows easy calculation of the meta-data about the segments, which are used extensively in the algorithms for the basic operations of the dynamic sets.

The insertion of a sequence of unordered numbers into BWA takes amortized time logarithmic to the length of the sequence. It is also proven that when the searched or deleted value is present in the BWA, the asymptotic amortized cost for the operations is $O(\log(n))$; otherwise, the time will fall somewhere between $O(\log(n))$ and $O(\log^2(n))$.

It is shown that the state variable `total`, which records the number of values in the BWA captures the dynamics of state transition of BWA. This fact is exploited to produce concise, easy-to-understand, and efficient coding for the operations. As it uses arrays as the underlying structure for dynamic set, a BWA need neither the space to store the pointers referencing other data nodes nor the time to chase the pointers as with any linked data structures.

A C++ implementation of the BWA is completed. The performance data were gathered and plotted, which confirmed the theoretic analysis. The testing results showed that the amortized time for the insert, search, and delete operations is all just between 105.949 and 5720.49 nanoseconds for BWAs of sizes ranging from 2^{10} to 2^{29} under various conditions.

Keywords: data structure and algorithm, dynamic data set, one dimensional space search, space efficiency, complexity analysis, performance, point cloud registration

1 INTRODUCTION

The size of a dynamic data set (DDS) may grow and shrink over time with insert, search, and delete operations [3]. The values in DDS are assumed to be drawn from some linearly ordered domains¹ such as integers, real numbers, lexicographically ordered words, and keys as in systems such as database. The linearity of the domain, though may not stated explicitly, is of critical importance to the implementations of the dynamic data sets.

It can be observed that linked structures, such as lists and trees, are almost exclusively used in the implementations of dynamic data sets [1, 20, 3, 17, 21]. The performance of the operations in these data structures is generally tied to the heights of the trees. This leads to the invention of a fairly large family of self-balancing data structures, including the AVL tree by Georgy and Evgenii (1962) [4], B-tree by Bayer and McCreight [2], 2-3 Tree by Hopcroft (1970) [1], Red-black tree by Guibas and Sedgwick (1978) [5] and many of their variants [3].

These tree based self-balancing data structures for dynamic data sets all can perform the basic operations in time logarithmic to the number of the values in the structure, and in this sense they can be said to be optimal². However, the flexibility of the tree structures does come at a price. First of all, additional space is needed for each data node to store the pointers to the children, parent (except the root), and some metadata related to the tree structure; secondly, extra time is required to perform the referencing and dereferencing of the data nodes since the values at the

¹More formally, totally ordered sets

²The height of a balanced tree with n nodes has necessarily a height of $\Omega(n)$.

nodes can only be accessed via their parents or children (pointer chasing); and finally, even though an elaborate scheme for tree rebalancing may not affect the asymptotic complexity, it could have yielded some rather large constant coefficient in front of the logarithmic function.

This brings up the question: is it possible to use structures other than lists or trees, e.g. arrays, to implement a dynamic data set with performance comparable to, or even in some aspects better than, those solutions based on linked structures?

The work reported here is, in part, an intellectual inquiry into the question, and, in part, an exploration aimed at providing a fast solution to the real-world problem of point cloud registration [16]. The Iterative Closest Point (ICP) method [6] for the problem calls for repeated neighborhood search in a three dimensional space. For better efficiency, main stream method with k -d trees [7] for 3-d space search [9] was not adopted. Instead, a dimensional shuffle transform (DST) was proposed [10, 11, 13, 12, 14], that reduces the search of a region in a high dimensional space [20, 19] to a small number of regional search in one dimensional space. This has been proven to lead to order(s) of magnitude improvement in searching speed by bypassing the high cost normally associated with the maintenance, traversal, and rebalancing of tree structures, which, in this case, is the k -d tree. The regional search over one dimensional space calls for DDS data structures. The idea of black-white array was then conceived and experimented with [15], which has proven to be superior in performance to the list and tree based solutions that were examined at the time³.

The paper is organized as follows. In Section 2, the layout of the black-white array is introduced; the algorithms in pseudocode for the operations are provided in Section 3; the asymptotic complexity of the operations are formally analyzed in Section 4.1; a C++ implementation and its performance data are presented in Section 4.2. Some extensions to the black-white array with augmented operations are discussed in Section 5; finally, concluding remarks are given in Section 6.

2 LAYOUT

A *black-white array* (BWA) of size $N = 2^k$ consists of a pair of arrays, the *black* (B) and the *white* (W), with lengths of $N/2$ and N respectively. The black array's indices ranges from 1 to $\frac{1}{2}N - 1$, while the white from 1 to $N - 1$.

Both the black and white arrays are conceptually divided into segments of different ranks, where the segment of rank i contains the entries with indices in the closed interval of $[2^i, 2^{i+1} - 1]$ for $i = 0$ to $k - 1$. It follows that a segment of rank i contains exactly 2^i entries, and there are $k - 1$ and k segments in the black and white arrays respectively. An illustration of the BWA layout is provided in Figure 1.

Indexing of the i th entry of the black and white arrays will be written as $B[i]$ and $W[i]$ respectively. The starting and terminating indices of a segment of rank j are respectively denoted by $S(j)$ and $T(j)$. It is easy to see that $S(j) = 2^j$, and $T(j) = 2^{j+1} - 1$, which can be calculated with $S(j) = 1 \ll j$, and $T(j) = (1 \ll (j + 1)) - 1$ respectively, where " \ll " is the left shift of a (binary form of the) integer to the left by the number of positions shown on the right side.

The black array is used only during the transitional states of the BWA, and at any stable state of the BWA, all the meaningful values are held in some segments of the white array. We say that a segment is *active* if the white segment with that rank is holding meaningful values. As will be seen, it is important to decide if a segment of a given rank is active at a given time.

A state variable named `total` is used to keep track the number of values stored in the BWA⁴. It turns out that this variable captures the configuration of the segments in the sense that a segment of rank i is active if and only if the i th least significant bit of `total` has a value of one. We therefore can decide if segment of rank i is active by a simple micro: `active(i) = total & (1 << i)`, where " $\&$ " is the bitwise AND, which returns a non-zero value if and only if the segment of rank i is active.

3 OPERATIONS

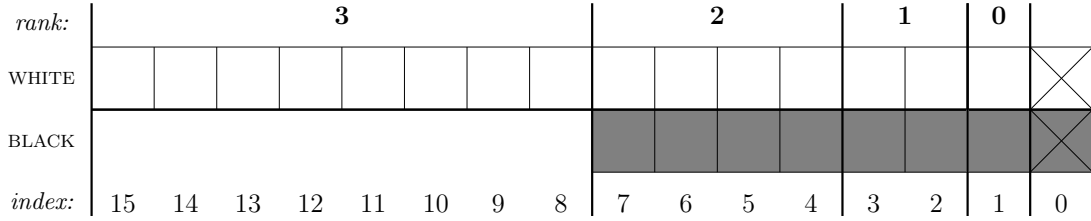
This section provides the procedures for the Insert, Search, and Delete operations of the BWA.

3.1 INSERT

The Insert of a value (Listing 1) is performed by first checking if the rank 0 segment is active. If not, the value is simply put at $W[1]$ (Line 3); otherwise, it is put at $B[1]$, followed by a merge of

³In particular, the skip list in Redis [17, 21] and red-black tress [5].

⁴VOID produced by a delete operation is counted as a value, though special.



A White-Black Array with R=4, N=16,

Figure 1: The layout of a white-black array with four segments

the black and white segments of rank 0 (Line 5, 6). Note that the “!” in Line 2 denotes the logical NOT operator.

```

1  Insert (v)
2      if (!active(0))
3          W[1] = v
4      else
5          B[1] = v
6          merge (0)
7      end
8  end

```

Listing 1: The Insert operation of BWA.

The following code is for the merge of black and white segments of rank i . Again, the result of the merge is put in the white segment of next higher rank, if it is not active; otherwise, the result is put in the black segment of higher rank, followed by a recursive call on the merge of two segments of that higher rank:

```

1  merge (i)
2      s = S(i)
3      t = T(i)
4      if (!active (i+1))
5          merge (s, t, W)
6      else
7          merge (s, t, B)
8          merge (i+1)
9      end
10     total = total +1
11 end

```

Listing 2: The merge of black and white segments of rank i .

Note that the operator `merge` is overloaded: `merge (i)` refers to the merge of black and white segments of rank i , whereas `merge (s, t, arr)` refers the merge of sections of black and white arrays with indices in the range of $[s, t]$. The argument `arr` indicate targeted destination of the merged result (black or white).

Figure 2 is an illustration of a recursive merge process. Note that, with this example, when the new value “52” is being inserted (Figure 2(a)), the total number of values in the BWA is $7 = [0111]$, which indicates the lowest ranked three white segments are active. It follows that the insert will recurse three times (Figures 2(b), (c)), and (d) before it settles. At that time, we increase the state variable `total` by one to become $8 = [1000]$. It follows that the next insertion will be done by one line of code (Line 3 of Listing 1).

The merge of two segments will always leave the old values that were merged in their original places, the corresponding segment then becomes *inactive*. It should be noted that there is no need to clear out the values in the inactive segments, for they will always be indicated by the “0” valued bit in the variable `total` after the recursion is completed, and will not be referred to until the segments are filled with new values and become active again. In Figure 2, we have used horizontal line to cross out those values that are in the inactive segments.

In the following, the *weight* of an integer refers to the number of non-zero bits in its binary form. For example, the weights of number 7 and 32 are respectively 3 and 1. Observe that the programs

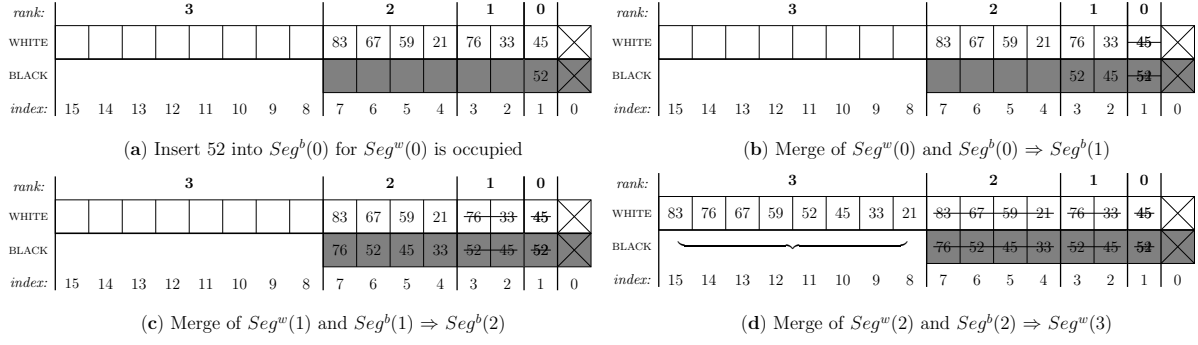


Figure 2: The insertion of an element and the invoked recursion process

```

1  merge (s, t, A)
2      i = s
3      j = t
4      while ((i <= t) && (j <= t))
5          if B[i] <= W[j]
6              A[k] = B[i]
7              i = i+1
8          else
9              A[k] = W[j]
10             j = j+1
11         end
12         k = k+1
13     end
14     if (i > t)
15         for (u = j; u <= t; u++)
16             A[k] = W[u]
17             k = k+1
18         end
19     else
20         for (u = i; u <= t; u++)
21             A[k] = B[u]
22             k = k+1
23         end
24     end
25 end

```

Listing 3: The merging procedure of segments with indices between s and t.

in Listing 1, 2, and 3 lead to the following properties of the dynamics of the Insert operation:

Property 1 (Insert). *Let the number of values stored in a BWA be $\text{total} = (b_{m-1}, \dots, b_i, \dots, b_0)$ with weight w , then there are exactly w active segments in the BWA; (b) a segment of rank i is active if and only if $b_i = 1$; (c) values in all active segments are sorted in ascending order; (d) when $\text{total} = 2^k$ is a k th power of two, there is only one active segment with rank of k , and all the values in the BWA are stored in that segment.*

A number n is said to be k -trailed if it has the binary form of $n = [b_{m-1}, \dots, b_{k+1}, \underbrace{0, 1, \dots, 1}_k]$.

For instance, the number 7, 11, and 9 are respectively 3-, 2-, and 1-trailed. A number is *strongly* k -trailed if it is k -trailed, and $b_i = 0$, for all $i > k$. The number 15, e.g. is strongly 4-trailed⁵. Note that any even number is, by definition, 0-trailed.

Theorem 1 (Recursion of Merge). *Let n be the number of values stored in a BWA, and it is k -trailed, then an insertion of a value into the BWA with n values leads to exactly k recursive merges.*

⁵It is easy to show that any integer n of the form $2^k - 1$ is strongly k -trailed.

Corollary 1 (Destination of Merge). *Suppose there are n values in the BWA, and n is k -trailed. When a new values is inserted, then (a) the result of the k recursive merges will be held in the white segment of rank k , which include all the values in segments with ranks smaller than k ; (b) All the segments with rank smaller than k will become inactive once the insertion is completed; (c) if n is strongly k -trailed, after the insertion, all the values in the BWA will be held in the segment of rank k .*

The variable `total`, which records the number of values stored in the BWA, therefore, encoded all the information about the BWA configuration. When it increases with an insertion, the changing process in its binary form is isomorphic to the merging dynamics of the segments. This is an elegant as well as useful property, which is exploited throughout the coding of the BWA operations.

3.2 SEARCH

The following is the program for the Search operation:

```

1  Search(v)
2      r = Nil
3      for (i=k-1; i ≥ 0; i--)
4          if (active(i))
5              r = segSearch(v, i)
6              if (r != Nil)
7                  break
8          end
9      end
10     end
11     return r
12 end

```

Listing 4: The Search for a value v in a BWA with size $N = 2^k$

Thus, the search for a value in a BWA is reduced to the searches over active segments, starting from the active segment with highest rank⁶. If the the value is found, the index of the entry with the value is returned. It returns Nil if and only if the value is not found in any of the active segments.

The search of a segment is performed by `segSearch`,

```

1  segSearch(v, i)
2      s = S(i)
3      t = T(i)
4      r = binarySearch(v, s, t)
5      return r
6  end

```

Listing 5: The search of a value within a given segment of rank i

The above listing calls a `binarySearch` (Line 4) over a section of an array with indices ranging from s to t inclusive. The algorithm is just a standard and simple binary search. For the reason of completeness, a version of the code is given in Listing 6.

Note that the middle point between s and t is calculated at Line 8 with a binary right shift.

3.3 DELETE

The deletion of value from BWA is straightforward by itself once a search for it is performed. When the value is found, a special value VOID is used to replace it. However, to prevent the deterioration of performance when the value of VOID becomes dominating in some segments, some care must be taken.

For this reason, we introduce an array V , referred to as the *occupancy vector* of length k for a BWA of size $N = 2^k$, where $V[i]$ records the number of real (non-VOID) values of segment of rank i . A procedure named *demote* is introduced, which will be invoked whenever the number of real values reached a threshold, which is half of the length of a given segment.

The code for Delete is given below:

⁶A search starting from the lower ranked to higher ranked active segments will not affect the correctness of the search operations, however, it is not as efficient as the other way around when a value may appear multiple times in the BWA.

```

1  binarySearch(v, s, t)
2      r = Nil
3      if ((t - s) ≤ 1)
4          if (W[s] == v) return s
5          if (W[t] == v) return t
6          return r
7      else
8          m = (s+t) ⋋ 1
9          if (W[m] ≤ v)
10             r = binarySearch(m, t, v)
11         else
12             r = binarySearch(s, m, v)
13         end
14     end
15     return r
16 end

```

Listing 6: Binary search of a value v between indices s and t

```

1  Delete(v)
2      r = Search(v)
3      if (r == Nil) return Nil
4      W[r] = VOID
5      i = seg(r)
6      V[i] = V[i] - 1
7      if (V[i] ≤ (size(i) ⋋ 1))
8          demote(j)
9          if (active(j-1))
10             merge (j-1)
11             total = total - size (j-1)
12             V[j] = V[j] + V[j-1]
13         else
14             total = total - size (j)
15             V(j-1) = V[j]
16         end
17     end
18 end

```

Listing 7: Delete of a value v from a BWA

As can be seen in Listing 7, `delete` always starts from a search for the value (Line 2). If the value is not found, it simply returns Nil to indicate the failure of the operation (Line 3); otherwise, the index of the value is returned, and the deleted value is replaced by a VOID (Line 4). With a successful delete, the value $V[i]$ is decreased by one (Line 6) to reflect the loss of a real value in the segment in which the deletion occurred. The occupancy rate of the segment is then checked (Line 7). If it does not reach 50%, the deletion is completed; otherwise, a demotion of the segment to the next lower rank will take place (Line 8). Depending on whether the lower ranked segment is active, the demote will copy all the non-VOID values to either the black or the white segment of lower rank. In and only in the former case, a merge of the black and white segments will take place at the lower rank (Line 10). Lines 11, 12, 14, 15 are the code to update the state variable *total* and occupancy vector V of the BWA.

Note that the call to `seg` at Line 5 of the above listing returns the segment its argument index belongs to. It can be easily calculated by an examination of the binary of the argument index, the position of most significant bit⁷ that equals to one is the rank the segment that the index belongs, e.g. $\text{seg}(7) = 2$ and $\text{seg}(11) = 3$.

The code for `demote` that is referred to at Line 8 in Listing 7 is given below:

⁷Counting from right to left, starting from zero.

```

1  demote(i)
2  s = S(i)
3  t = T(i)
4  arr = (active(i-1)? B:W)
5  j = S(i-1)
6  for (j= s; j <= t; j++)
7      v = W[j]
8      if (v != VOID)
9          arr[j] = v
10     end
11     j = j+1
12 end
13 end

```

Listing 8: The demote operation of a half emptied segment of rank i

Figure 3 illustrates a delete of the value 59 to take place in segment of rank 3 (a); the number of non-VOID values in the segment after the deletion decreased to half of its length, a demotion is then to be invoked (b); since segment of rank 2 is active, the demotion result is put in black segment (c); a merge at rank 2 takes place, all the values from the white and black segments of rank 2 are now in segment of rank 3 (d). Note that the value of VOID is denoted by ϕ in the illustration.

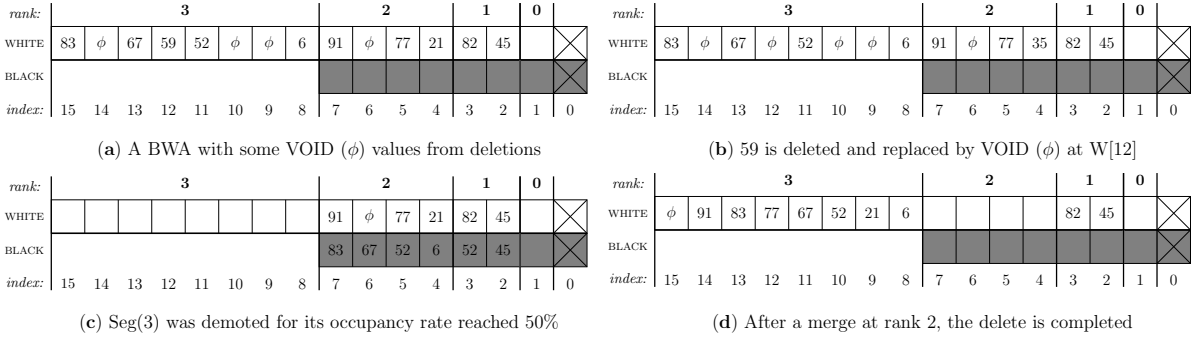


Figure 3: The demotion process invoked by a delete

The code in Listing 7 and 8 leads to the following properties:

Property 2 (Delete). (a) The delete of a value leads to a demotion if and only if the number of non-VOID values decreased to half of the segment size due to the deletion; (b) when a segment of rank i is demoted, all its non-VOID values will be moved to segment of rank $(i - 1)$; (c) when the segment of rank $(i - 1)$ is inactive, the result of demotion will be put in the white segment of rank $(i - 1)$; (d) otherwise, when the segment of rank $(i - 1)$ is active, the demoted segment will be held in the black segment of rank $(i - 1)$, followed by a merge of black and white segments of rank $(i - 1)$, the result is then put back to the white segment of rank i .

In the following, the *occupancy rate* of a segment refers to the ratio between the number of non-VOID values in the segment and the size of the segment.

Theorem 2 (Occupancy Rate). A demote operation results in either a new active segment of rank lower by one with an occupancy of 100% if the lower ranked segment was inactive; or a new segment of the same rank with an occupancy rate strictly greater than 75% after the merge with the segment of rank lower by one.

Corollary 2 (Occupancy Lower Bound). At any stable state of a BWA, the occupancy rate of any active segment is always strictly greater than 50%.

It is this lower bound of the occupancy rate that guarantees the BWA performance will not deteriorate with arbitrary sequences of operations, including deletes.

4 PERFORMANCE

The dynamics and the complexities of the BWA operations are discussed Section 4.1. In Section 4.2, a C++ implementation of BWA is described, the performance data of the BWA in C++ are

presented and plotted, serving in part as a validation to the theoretical analysis.

4.1 ANALYSIS

When a sequence of values are inserted into a BWA, the merging process can be illustrated with a *BWA merge tree*. For a BWA of size $N = 2^{k+1}$, the merge tree has k levels, corresponding to the different ranks of the segments. The leaf nodes correspond to the input values from left to right in the order they are inserted. An internal node represents a merge between the two segments denoted by its children. A white colored node represents a white segment, and a black colored node represents a black segment. Figure 4 illustrates such a merge tree with four levels.

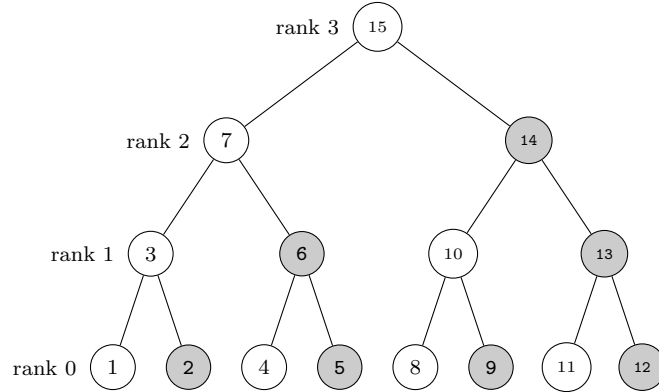


Figure 4: The BWA merge tree and its post order traversal

The insert sequence in BWA leads to the following simple dynamics of the tree: (a) the input values are put to the leaf nodes, one at a time; (b) for any internal node, a merge of its two children of different colors is invoked whenever both of its children are present⁸.

A close examination of the BWA merge, then, leads to

Lemma 1 (BWA Merge Tree). *The order of the merges represented by the nodes in a BWA merge tree is a post order traversal of the tree.*

Proof. Obvious. □

In Figure 4, the labels inside the nodes of the BWA merge tree have been used to indicate their sequencing number in a post-order traversal of the tree.

It is interesting to compare the BWA merge tree with that of mergesort. Figure 5 is an illustration of the merge tree of mergesort for eight values. It is obvious that the merge trees for BWA and mergesort share the same tree structure, the difference lies only in the order the nodes are traversed, which represent the sequence of merge operations that an algorithm performs⁹.

The traversal order of the merge tree for mergesort is one level at a time, from the leaf level to the root. The internal nodes, which represent merge operations, at a higher level, are not to be visited until all the nodes of the lower level have been visited. An illustration of the merge tree for **mergesort** is given in Figure 5 with the labelled order by which the nodes are traversed.

Theorem 3 (Insert Time). *The insertion of a sequence of values of length $n = 2^k$ into a BWA takes $O(n \log(n))$ time.*

Proof. There are $\frac{n}{2^{i+1}}$ merges for segments of rank i with length 2^i , for $i = 0$ to $(k - 1)$, therefore the total number of comparisons, $T(n)$, is given by

$$T(n) = \sum_{i=0}^{k-1} \frac{n}{2^{i+1}} \times 2^i = O(n \log(n))$$

□

⁸This can also be understood from the view of a data flow paradigm: an operation is invoked whenever its arguments are ready.

⁹Hence, BWA can be said to perform “incremental merge sort” with exactly the same complexity as merge sort, together with some other functionalities that merge sort does not provide.

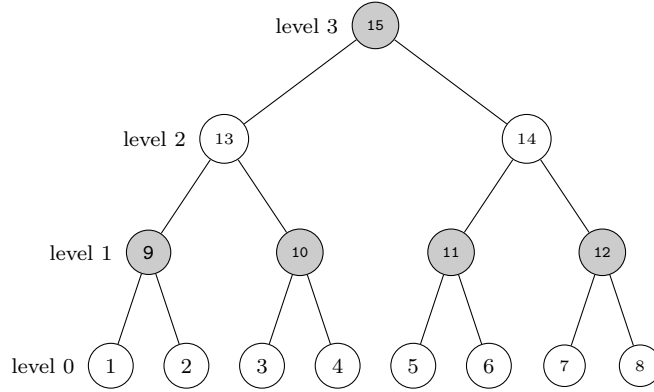


Figure 5: The operational order in mergesort

Corollary 3 (Amortized Cost of Insert). *Insertion of n values into a BWA takes average $O(\log(n))$ time per insert.*

This conclusion, while seemingly trivial, is actually interesting in that it holds despite the fact that one can infer from the code in the Listing 1 that the time taken by an insert is $O(1)$ every other time an insertion is performed.

A search is said to be a *hit* if the searched value is found in the BWA, otherwise, a *miss*. There can be some difference between the time spent on hit and miss under certain conditions. In the following, we will make a distinction between hit and miss in some cases.

The search time obviously depends on the number and the ranks of the active segments in a BWA at the time of the search. We will start the analysis with a simple case:

Theorem 4 (Search Time, case $n = 2^m$). *When the number of values in a BWA is a power of two of the form $n = 2^m$, the time to search for a value in the BWA is $O(\log(n))$.*

Proof. In this case, all the values are in segment of rank m of length n . □

Theorem 4 holds true regardless whether the search is a hit or miss under the given condition. However, when n is not a power of two, more than one segment may need to be searched. The worst case occurs when $n = 2^m - 1$, and in this case, every segments of rank smaller than m is active, and as such, could be potentially searched¹⁰.

Theorem 5 (Search Time (Hit)). *For a BWA with up to $n = 2^m$ values, the amortized time of a hit search is $O(\log n)$, provided that the values are uniformly distributed.*

Proof. The probability that a values falls in a given segment of BWA is proportional to the length of the segment under the uniform distribution assumption. The search algorithm starts the search from active segment with highest rank, which has a length half of the size of the BWA, and hence the probability of finding the value in it is $\frac{1}{2}$. The probability of finding the value in the next lower rank segment is $\frac{1}{4}$, and, generally, the probability of finding the value in a segment of rank i is half of that of rank $(i + 1)$. Note that finding a value in segment i means that all the segments of higher ranks have been searched, so the cost for searching the higher ranked segments should also be charged together with the time taken to search the current segment. The search is performed only over active segments, the worst case would be all the segments are active¹¹, and in that case each segment will be searched until the value is found.

¹⁰We say potentially searched, for the search could be terminated before a segment is searched for the value was found in a segment of higher rank.

¹¹This worst case occurs when $n = 2^k - 1$, and in this case, all the least significant bits take the value of one.

Size	Insert	Search	Delete	Search*	Delete*	Size	Insert	Search	Delete	Search*	Delete*
2^{10}	105.949	142.289	239.594	540.43	576.862	2^{20}	182.601	296.176	611.42	1845.1	1955.55
2^{11}	99.254	139.51	254.676	619.694	660.022	2^{21}	193.025	373.591	664.332	2026.22	2211.12
2^{12}	110.299	149.991	271.268	710.861	771.738	2^{22}	200.564	540.384	730.102	2350.39	2485.58
2^{13}	119.603	158.841	292.494	788.732	881.442	2^{23}	211.788	644.83	789.652	2717.32	2823.35
2^{14}	128.13	162.641	324.852	880.209	954.138	2^{24}	213.38	724.457	851.47	3062.46	3249.46
2^{15}	135.945	173.694	350.686	994.758	1091.76	2^{25}	222.911	799.269	911.798	3542.61	3631.58
2^{16}	144.256	185.698	390.012	1155.82	1244.09	2^{26}	235.683	876.615	944.468	4056.0	4110.58
2^{17}	152.413	215.686	461.078	1293.16	1391.76	2^{27}	239.855	968.382	1061.95	4702.94	4593.06
2^{18}	160.31	222.679	478.018	1461.91	1497.18	2^{28}	249.329	1087.05	1159.89	4886.36	5231.23
2^{19}	162.527	239.829	546.856	1640.62	1735.51	2^{29}	263.82	1246.46	1145.87	4960.39	5720.49

Table 1: Amortized time (ns) of operations with perfect/random(*) BWA configurations

$\frac{1}{1.5} = \frac{2}{3} \approx 0.667$, which is much higher than that of any tree based data structures¹³ for dynamic data sets.

4.2 IMPLEMENTATION AND PERFORMANCE

The BWA has been implemented in C++ as a class. The size of the BWA N , the logarithmic k of N , the occupancy vector V , where $V[i]$ shows the number of non-VOID values in segment of rank i , are all private members of the class. The operations, Insert, Search, Delete, are made public, whereas merge, demote, and some other auxiliary functions are private. The programs in C++ follow closely in structure and style of the code presented in Section 3.

The testing of the performance was conducted on a Mac Pro laptop computer running Mojave MacOS. The system comes with a i7 processor with six cores, clocked at 2.6 GHz. It has 9 MB L3 cache, and 256 KB L2 cache per core. At the time to start the testing, the free memory is maintained at the level between four to six GB, whereas the CPU is about 90% idle. The values used in the tests are randomly generated with a range beyond the size of the BWA. The chrono time library [18], which comes with C++ 11, is used for the timing.

The tests are conducted for BWAs with size N ranges between $2^{10} = 1,024$ and $2^{29} = 536,870,912$. The testing values are four byte integers. Hence, the space that the BWA itself takes up during the tests will go up to 3 GB. The random values used during the tests are pre-generated and being held an array of size up to 2 GB. The Performance Monitor of the Mac system indicated that the testing program, as an application, took around 99% of the computing time of one core, and 10 GB of system memory.

The performance data for Insert, Search, and Delete operations gathered during the tests are listed in Table 4.2 and plotted in Figure 6. The horizontal axis, representing the size of the BWA, is scaled logarithmically, whereas the vertical axis, showing the time that operations take in the unit of nanosecond, is linearly scaled. Hence, a logarithmic function will show up as a straight line in such a figure.

The tests are conducted for two different cases. In the first case, the state variable `total` is a perfect power of two, and in the other, a random number ranging up to the size of the BWA. It follows that the number of active segment equals one in the former case, and equals to the weight of the random number in the latter case. For the latter case, the average time of a thousand runs is taken, each of which is on a different configuration of the BWA as captured by the state variable `total`. The left and right parts of Table 4.2 showed the performance for the two cases respectively, which are in turn respectively plotted in Figure 6(a) and 6(b).

A comparison of Figure 6(a) and 6(b) shows that the Search and Delete operations over random configurations indeed cost more than that over the “ideal” configurations when the state variable `total` happens to be a power of two. In contrast, the Insert operation is insensitive to the difference of configurations, therefore, its timings are identical in the two figures. This should come as no surprise for it is simply a manifestation of Corollary 3.

As shown in Table 4.2 and Figure 6, the amortized time taken by the operations in a BWA up to the size of 2^{29} is from 263.82 to 5720.49 nanoseconds in all the cases.

Observe that whenever the number of inserted values is a k -th power of two, all the inserted value will appear in the BWA as one sorted sequence in the segment of rank k . It is therefore sensible to compare the performance data with other sorting algorithms. As pointed out in Lemma 1, and shown in Figure 4 and 5, the BWA merge tree is isomorphic to that of mergesort with only a differences in the order the nodes are traversed. It follows that the Insert of BWA is at least as fast

¹³In tree based dynamic date structures, each node of the tree has to hold pointers to its children and parent as well as a piece of data.

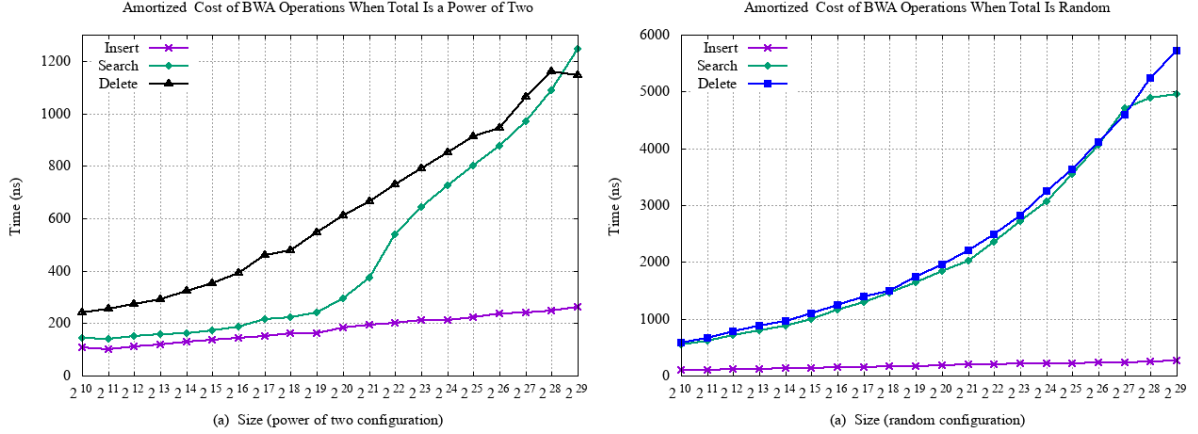


Figure 6: Insert, Search and Delete time per operation (C++ on MacPro)

as that in mergesort per operation. Hence it should come as no surprise to see that the amortized cost went from 105.949 to only 263.82 nanoseconds as the number of inserted values increased 2^{19} folds from 1,024 to 536,870,912 with a fairly flat slop as shown in Figure 6.

5 AUGMENTATIONS

We have shown that black-bwhite array can be deployed as a data structure for dynamic set with Insert, Search and Delete operations. In this section, we outline some fairly straightforward augmentations to its functionality, including

- **Max(Min)** : returns the maximum (minimum) value;
- **ExtractMax(Min)**: Find and delete the maximum (minimum) value;
- **LowerBound (UpperBound) (v)**: finds the position of the smallest (largest) number which is greater (smaller) than value v ;
- **Interval (s, t)**: Returns all the values falling between the interval of $[s, t]$;
- **IncrementalSort**: incrementally sorts a sequence of unordered values, one at a time.

It is the property of a BWA that the maximum (minimum) value in a segment of rank m is always at the top (bottom)¹⁴ of the segment with index of $(2^{m+1} - 1)(2^m)$ provided that is not a VOID entry, otherwise it is to be found at the index closest to the top (bottom) on the left (right) where the entry is not a VOID. To find the global maximum, we just need to go through all the active segments, compare and throw out the smaller value each time a new segment is examined. It follows that Max, Min, ExtractMax, and ExtractMin operations can all be done in $O(\log(n))$ time¹⁵. In applications such as k nearest neighbors [8], the ExtractMin operations can be applied k times.

The LowerBound (UpperBound) can be achieved with fairly simple modifications of the Listings 4, 5 and 6. The results will be the LowerBound (s) and UpperBound (t) indices in each of the active segments, if any¹⁶. Interval (s, t) can then be applied to pull out all the values in between.

We have shown that a sequence of length n can be inserted one by one in $O(n \log(n))$ time (Section 4.1). BWA can then be used as an incremental sorting algorithm at least as efficient as a well-implemented mergesort in time and space.

There are other augmented operations for BWA with practical applications, which will be discussed in separate works.

¹⁴Meaning the position with the highest (the lowest) index.

¹⁵When the number of values in the BWA, or in other words, the state variable `total` is a power of 2, the time will become $O(1)$.

¹⁶The values in a segment may and may not overlap with the interval $[s, t]$.

6 CONCLUSION

The black-white array (BWA) is presented as an array based data structure for dynamic data set supporting effectively and efficiently the insert, search, and delete operations, with mathematically elegant structure, easy-to-understand code, and competitive performance against those based on linked structures such as lists and trees.

The performance data of a C++ implementation of the BWA were presented, which validated the formal analysis of the amortized costs of the operations. The nanosecond and low microsecond wall-clock time per operation suggests that the random access nature of the underlying array structure indeed can minimize the constant factor in front of the predicated asymptotic functions. When it comes to space utilization, BWA is far more efficient than any of those list or tree based approaches by a wide margin.

The versatility of the BWA structure also allows it to easily support augmented operations such as Max, Min, Interval, and IncrementalSort. Hence, it subsumes in functionality some other data structures and algorithms e.g. priority queue and mergesort. As an alternative to those well-known list or tree based data structures for dynamic data sets such as skip lists [17, 21] and red-black tree [1], BWA can be deployed in a broad range of applications. One of the use cases, which in part motivated this work, is to apply it to point cloud registration with Iterative Closest Point (ICP) method as described in [14, 15].

REFERENCES

- [1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] R Bayer and E McCreight. Organization and maintenance of large ordered indices. In *The Boeing Scientific Research Libraries*, volume doi:10.1145/356770.1734671, 3 1970.
- [3] T. H. Corman, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms E. and Ullman, J. D.* MIT Press, 1990.
- [4] Adelson-Velsky Georgy and Landis Evgenii. An algorithm for the organization of information (in Russian). In *Processings of the USSR Academy of Science*, pages 146:263–266, 1962.
- [5] Leonidas.J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Annual Symposium on Foundations of Computer Science*, volume doi:10.1109, pages 8–21, 3 1978.
- [6] Wikipedia Iterative Closest Point. https://en.wikipedia.org/wiki/iterative_closest_point.
- [7] Wikipedia k-d tree. https://en.wikipedia.org/wiki/k-d_tree.
- [8] Wikipedia KNN. https://en.wikipedia.org/wiki/k-nearest_neighbors_algorithm.
- [9] Point Cloud Library. <http://www.pointclouds.org>.
- [10] Zhijing George Mou. Methods and system for multi-dimensional and geographic search. In *Pub. No.: US2010/0174721 A1*. USPTO, Jul. 8 2010.
- [11] Zhijing George Mou. Methods and system for multi-dimensional and geographic search. In *Patent No.: US 8,510,310 B2*. USPTO, Aug 13 2013.
- [12] Zhijing George Mou. Methods and systems for real time 3-d space search and point cloud registration using a dimensional shuffle. In *WO 2018/183754 A1*. World Intellectual Property Organization, International Bureau, October 4 2018.
- [13] Zhijing George Mou. Methods and systems for real time 3-d space search and point cloud registration using a dimensional shuffle. In *Pub. NO. US2019/0130525 A1*. USPTO, May 2 2019.
- [14] Zhijing George Mou. Methods and systemsfor real time 3-d space search and point cloud registration using a dimensional shuffle. In *Patent No.: US 10,580,114 B2*. USPTO, March 3 2020.
- [15] Zhijing George Mou. Methods and systemsfor real time 3-d space search and point cloud registration using a dimensional shuffle. In *CIP of US PAT 10,580,114 B2*. USPTO, March 2020.
- [16] Wikipedia Point Cloud Registration. http://en.wikipedia.org/wiki/point_set_registration.
- [17] RedisLabs. <https://redis.io/topics/data-types>.

- [18] C++ 11 Reference. <http://www.cplusplus.com/reference/chrono/>.
- [19] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publisher, 2006.
- [20] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, 1992.
- [21] Wikipedia Skip List. https://en.wikipedia.org/wiki/skip_list.