Abstract

This report discusses the implementation of a C++ library revolving around the idea of storing array data in an alternate layout as opposed to a sorted array for faster search speeds on large values of $n$; the eytzinger ordered array. With a faster search speed comes negative side effects of other operations. The focus is to look at different efficiencies and implementations of algorithms revolving around this data structure to minimize the negative side effects. Specifically we look at searching, traversal, and moving in place from sorted order to eytzinger ordered and vice versa; with an emphasis on optimizing the later two.

With traversal, the end goal would be to match the speed of traversing a sorted array. Although each iteration was done in constant time, traversing the eytzinger ordered array with the current implementation is about 14 times slower than that of a sorted array. Moving from sorted order to eytzinger order can be done in linear time, with the best implementation being able to handle about $10^9$ elements in around four seconds.
Acknowledgements

I would like to acknowledge Dr. Pat Morin for his previous research in the area and his assistance throughout the term when needed. Also to Paul-Virak Khuong, he and Pat Morin authored the paper driving the main motivation for this project.
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>5</td>
</tr>
<tr>
<td>1.1 Objectives of Search</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Objectives of Traversal</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Objectives of Sorted to Eytzinger Permutation</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Search</strong></td>
<td>7</td>
</tr>
<tr>
<td><strong>3 Traversal</strong></td>
<td>8</td>
</tr>
<tr>
<td>3.1 Basic Traversal</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Constant Traversal</td>
<td>10</td>
</tr>
<tr>
<td>3.2.1 Constant Next Functions</td>
<td>10</td>
</tr>
<tr>
<td>3.2.2 Constant Previous Function</td>
<td>11</td>
</tr>
<tr>
<td><strong>4 Sorted to Eytzinger Permutation</strong></td>
<td>12</td>
</tr>
<tr>
<td>4.1 Prime Outshuffle</td>
<td>14</td>
</tr>
<tr>
<td>4.2 Blocked Outshuffle</td>
<td>15</td>
</tr>
<tr>
<td>4.3 Swap Outshuffle</td>
<td>16</td>
</tr>
<tr>
<td><strong>5 Eytzinger to Sorted Permutation</strong></td>
<td>17</td>
</tr>
<tr>
<td><strong>6 The API</strong></td>
<td>18</td>
</tr>
<tr>
<td>6.1 Iteration and Search</td>
<td>19</td>
</tr>
<tr>
<td>6.1.1 Consumer Code for Iteration</td>
<td>19</td>
</tr>
<tr>
<td>6.1.2 Consumer Code for Search</td>
<td>20</td>
</tr>
<tr>
<td>6.2 Array Manipulation Functions</td>
<td>20</td>
</tr>
<tr>
<td><strong>7 Results</strong></td>
<td>20</td>
</tr>
<tr>
<td>7.1 Outshuffle Results</td>
<td>20</td>
</tr>
<tr>
<td>7.2 Permutation From Sorted to Eytzinger Results</td>
<td>21</td>
</tr>
<tr>
<td>7.3 Iteration Results</td>
<td>22</td>
</tr>
<tr>
<td><strong>8 Conclusion</strong></td>
<td>22</td>
</tr>
<tr>
<td>8.1 Future Work</td>
<td>23</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The Eytzinger Layout.</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Implementation of search in an eytzinger ordered array.</td>
</tr>
<tr>
<td>Figure 3</td>
<td>The two different cases handled in traversal of an eytzinger ordered array.</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Pseudo code for basic, case 1 next.</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Pseudo code for basic, case 2 next.</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Implemented next function.</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Implemented previous function.</td>
</tr>
<tr>
<td>Figure 8</td>
<td>An outshuffle on twelve elements.</td>
</tr>
<tr>
<td>Figure 9</td>
<td>An unbalanced binary search tree where $m = 3$.</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Takes an index $i_a$ and length $n$, and returns the index $i_b$.</td>
</tr>
<tr>
<td>Figure 11</td>
<td>A visualization of blocked outshuffle.</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Example code that swaps consecutive elements, starting at index $i$, incrementing i by 4 each time.</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Example search and iteration usage of the eytzinger API.</td>
</tr>
<tr>
<td>Figure 14</td>
<td>A Summary of results (all on $n = 10^8$ items).</td>
</tr>
</tbody>
</table>
1 Introduction

The Eytzinger layout (see figure 1) is an array representation of a complete binary search tree as if it was traversed breadth first, from left to right.

![Eytzinger layout diagram](image)

Figure 1: The eytzinger layout.

The main motivation for this project is driven from results showing that the right search algorithm on an eytzinger ordered array can outperform the standard binary search on a sorted array for larger values of $n$. Note that $n$ here refers to the size of a given array, and will be used as such throughout the remainder of the report. These results are primarily explored by Paul-Virak Khuong and Pat Morin (2017) in the research paper, “ARRAY LAYOUTS FOR COMPARISON BASED SEARCHING”. The results shown in this paper drive the exploration into algorithms around the eytzinger ordered array, and the motivation to create a c++ library with a good api for general usage. It is important to note that the search efficiency as compared to other array layouts it not the focus of this project. Specifically, besides the standard search function with the eytzinger layout, the focus of this report and the library implemented includes the ordered traversal of an eytzinger ordered array as well as in place shuffling an ordered array to the eytzinger form, and vice versa. We will discuss the approaches used to tackle these problems as well as the time complexity and performance achieved with different solutions.
1.1 Objectives of Search

The standard search algorithm on an eytzinger ordered array is somewhat straightforward. First understand that for a given array index \( i \), the left and right child of the element at index \( i \) can be found in constant time by calculating \( 2i + 1 \) and \( 2i + 2 \) respectively. The basic concept of searching is exactly similar to searching a binary search tree, where you start at position \( i = 0 \) and traverse the tree left or right based upon comparison of the target element as opposed to the element at position \( i \). There are different ways to code this in c++ that result in varying runtimes, which is explored by Paul-Virak Khuong and Pat Morin (2017). For this reason, the c++ library discussed here only utilizes the fastest implementation and does not attempt to improve on the search algorithm.

1.2 Objectives of Traversal

Traversal of the eytzinger ordered array is simple in understanding but complex in nature to perform optimally. Note traversal here and so forth will be referring to traversing the eytzinger ordered array as if it was in sorted order. Because we understand from 1.1 where in a given eytzinger ordered array, a parents children can be accessed in constant time, we can easily traverse the array as if we were traversing a binary search tree, up or down, from any node. Note from now on that \( arr \) will be referred to as an array containing \( n \) elements, with items arbitrarily assessed at position \( i \). To traverse the tree, we need functions to find the next (respectively, previous) elements, successing (respectively, preceding) a given element \( arr[i] \), that returns the index of the element that would be at position \( i + 1 \) (respectively, \( i - 1 \)) of the sorted order version of \( arr \). The trivial implementation, discussed later, can be done easily in \( O(\log(n)) \) time. This means that a full, in order traversal, would take \( O(n\log(n)) \). This is not ideal, and we have optimized these algorithms using similar bit tricks discussed by Paul-Virak Khuong and Pat Morin (2017) to obtain a linear runtime in order traversal of a given eytzinger ordered array.
1.3 Objectives of Sorted to Eytzinger Permutation

It is important to be able to easily obtain an eytzinger ordered array if we want usage of this data structure to gain in popularity. For this reason we look at constructing an eytzinger ordered array from a given sorted array, as well as being able to revert back from eytzinger to sorted order. Note from Figure 1, every even indexed element \( i = 0, 2, \ldots, 14 \) of the sorted array where \( n = 15 \) is placed in order sequentially at the end of the eytzinger order array. This is referred to as an ‘outshuffle’, and is the building block in the algorithm to move from an sorted order to eytzinger order. The basic concept in a situation where \( n = 2^x - 1 \) is to recursively call this outshuffle on \( n/2 \) elements of the array. This can be visualized also in Figure 1, as an outshuffle called on an \( arr = [2, 4, 6, 8, 10, 12, 14] \) would result in \([4, 8, 12, 2, 6, 10, 14]\). Performing an in-place outshuffle is non trivial, and we will discuss later different methods to accomplish this. Being able to perform an outshuffle in linear time would result in a linear runtime of a to_eytzinger function, as recursively calling a linear runtime function on \( n/2 \) elements each time results in an overall runtime of \( O(2n) \) which is still linear. Pat Morin has done previous research in this area and has already accomplished a linear runtime to_eytzinger. We focus in this project on understanding his approaches as well as developing my own alternatives and comparing runtimes. The c++ library developed utilizes the fastest current implementation.

2 Search

Search is heavily discussed in previous research by Paul-Virak Khuong and Pat Morin (2017), so we will only discuss on a higher level what is happening to obtain a basic understanding of the implementation used in the c++ library implemented. See Figure 2 for an implementation of a search on an eytzinger ordered array. We first start by constructing a ‘path’ that will contain the upper bound of the element we wish to search for. The ‘path’ is the binary representation of the integer \( i + 1 \) after the while loop in figure 2. The depth, \( d \), of the upper bound of the element we wish to search for is the position of the highest order 1 bit in \( i + 1 \). Knowing the depth of the target element, as well as the path it is on, we can easily return the index of the upper bound of the element we wish to search for, or \( n \) if it does not
exist. Again this was not one of the main topics of the report, but fundamental to the purpose of the eytzinger layout. We will get into the bit ticks used here more when we expand on the next and previous functions touched on in 3.2.

```cpp
template<class Data, typename Index>
Index Eytzinger<Data, Index>::upper_bound(Data element) {
  Index i = 0;
  while (i < n) {
    i = (element <= arr[i]) ? (2 * i + 1) : (2 * i + 2);
  }
  Index j = (i + 1) >> __builtin_ffs(~(i + 1));
  return (j == 0) ? n : j - 1;
}
```

Figure 2: Implementation of search in an eytzinger ordered array.

3 Traversal

In order traversal if the eytzinger ordered array array can be achieved by handling two different cases depending on where the current index i is in the array. If the current index has a right child, we are in one case, and if the current position does not have a right child, we are in another case. See Figure 3 for a clear display of determining the different cases for an eytzinger ordered array where n = 15. From here on, we will refer to ‘case 1’ if we are dealing with a node that does not have a right child, and ‘case 2’ otherwise.

Figure 3: The two different cases handled in traversal of an eytzinger ordered array.
3.1 Basic Traversal

Basic traversal of the array involves just following the tree up or down, depending on which case we are in, until we arrive at the index of the element successing or preceding the current element at index \(i\). Note that to get to the parent of a node at position \(i\), you simply have to calculate \(n/2\) or \(n/2 - 1\) depending if you are a left or right child, respectively. Let us first consider case 1, where the current node does not have a right child (see Figure 4 for pseudo code). While you are not a left child, you move up to your parent node. Afterwards, you go to your parent one more time and arrive at the next ordered element in the eytzinger ordered array.

```
while(not_left_child){
    goto_parent();
}
```

Figure 4: Pseudo code for basic, case 1 next.

For case 2, where you have a right child, you have to go down the tree. You first start by going to your right child, then follow the tree left as far as possible to arrive at the next ordered element in the eytzinger ordered array (see Figure 5 for pseudo code).

```
goto_right_child();
while(has_left_child()){
    goto_left_child();
}
```

Figure 5: Pseudo code for basic, case 2 next.

Both of these cases clearly run in logarithmic time with respect to \(n\), and are trivial to see with any element in Figure 3. For example, going to the next element from the root node (which has \(i = 0\) and \(arr[i] = 8\)), you go down to the right child, then down to the left as far as you can; which will yield the result of \(i = 11\) (\(arr[11] = 9\)). The previous operation is extremely similar, with the same two cases just handled slightly different.
3.2 Constant Time Traversal

Utilizing bit tricks and observations made by Paul-Virak Khuong and Pat Morin (2017), we can do the same ‘next’ and ‘previous’ operations discussed in 3.1, but much faster. First, we must reiterate and expand on some points made in 2. For a given index $i$, the binary representation of $i + 1$ encodes the path from the root to the node at position $i$. This is a fundamental concept to understand with the etzinger array layout. The depth, $d$, of $i$ is the index of the highest order 1 bit of the binary representation of $i + 1$. To understand further the encoded path, let $b$ be the binary representation of $i + 1$. Then the bits $b_{d-1} ... b_0$ encode the path to $i$ from the root node, where $b_x = 0$ means to go to the left child, and $b_x = 1$ means go to the right child. With this understanding, we can systematically and in constant time forge the path from a given node to its ordered next, or ordered previous node, in the etzinger ordered array. Note that both the next and prev functions implemented return -1 if the iteration loops over the end of the array. This is used later when we discuss the api chosen for the library.

3.2.1 Constant Next Function

Let us consider case 1, where a node $i$ has no right child. This can be calculated in constant time by checking if $2i + 2$ is larger than $n - 1$. If this is the case, we know that the next element has to be on the path from the root node to $i$, (which we know by the binary representation of $i + 1$). All we have to do then is get to the proper depth of the node that would be the successor of arr[i]. To do this, we right shift $i + 1$ by $k$ bits, where $k$ is the index of the lowest order 0 bit. This is because the lowest order 0 bit represents the last move to a left child before arriving at $i$. After right shifting $i + 1$ by $k$ bits, we have the binary representation of the the path to the item succeeding arr[i], so to go from path to index, we just have to subtract 1. This is seen in the first part of the if statement in Figure 6.

Let us now consider case 2, where a node $i$ indeed has a right child. In this case, we just go to to the right child of $i$, then to the furthest left child possible. This can be easily done by altering the path to the original node. We know that $i$ has one right child, so we first left shift $i + 1$, then add one. The result of this is the path to the right child of $i$. We then left shift the
remaining bits by the difference between the tree height and the depth of \( i \), to simulate going to the furthest left child. The return statement handles the case where the tree is not balanced, and the depth of the leaves can differ; this is explained further when we look at permuting from sorted order to eytzinger order. Because case 2 always has a next value, if we are about to return a value greater than \( n - 1 \), we know the tree is unbalanced and we need to right shift back once. An example \( \text{c++} \) implementation is in figure 6, with case 2 specifically in the else block.

```cpp
template<class Data, typename Index>
Index Eytzinger<Data, Index>::next(Index i){
    if (2 * i + 2 > n - 1){
        int j = (i + 1) >> __builtin_ffs(~(i + 1));
        return j - 1;
    } else{
        int iDepth = 32 - __builtin_clz(i + 1);
        int treeDepth = 32 - __builtin_clz(n);
        int j = (((i + 1) << 1) + 1) << (treeDepth - iDepth - 1);
        return (j - 1) > (n - 1) ? (j - 1) >> 1 : (j - 1);
    }
}
```

Figure 6: Implemented next function.

3.2.2 Constant Previous Function

The implementations of a previous functions is quite similar to that of a next function, except everything specific to left or right children is reversed. Again, there are the same two cases to handle. In Case 1, we still right shift \( i + 1 \) by \( k \) bits, but \( k \) in this case is the index of the lowest order 1 bit (as opposed to the lowest order 0 bit in the next function). In Case 2, we again forge the path to the next node by altering the path to \( i + 1 \), but instead we go to the left child of \( i \), then the furthest right child. An example \( \text{c++} \) implementation of the previous function given a starting index \( i \) on an eytzinger ordered array is shown in figure 7.
4 Sorted to Eytzinger Permutation

We now get to explore the non trivial algorithm of moving from a sorted order array to an eytzinger ordered array, in place. We must first have a good understanding of exactly what an outshuffle is. An outshuffle on \( n \) elements involves moving every even indexed element to the end of the array, all while preserving order. The preservation of the order of the odd indexed elements is also preserved. See Figure 8 for a visualization of an outshuffle.

Repeated calls to outshuffle on an array where \( n \) is of the form \( 2^x - 1 \), recursively on \( n/2 \) elements, will result in an array being in eytzinger order. This is only the case if the binary search tree is balanced. From here on, \( m \) will represent the number of elements on the lowest depth of the non-balanced eytzinger ordered array. See figure 9 for a visualization of where these \( m \) elements would be placed on a binary search tree.
In the case where a tree is not exactly of the form $2^x - 1$, then all we have to do is move the correct $m$ items to the end of the array while keeping the other $n - m$ items sorted. The elements left over after this operation are of the form $2^x - 1$, and using the same method of repeated calls to outshuffle will result in a proper eytzinger ordered array. The $m$ items can be easily identified as the first $m$ even indexed elements \{0, 2, ..., $2(m - 1)$\}. To separate these $m$ items, we can again use outshuffle. If we call outshuffle on $2m$ items, we will be left with the array in three ‘pieces’.

1. The first $m$ odd indexed elements, with their sorted order retained
2. The first $m$ even indexed elements, with their sorted order retained
3. The remaining $n - 2m$ elements, untouched

All we need to do now is perform a rotate on the elements left over after this operation to be left with $2^x - 1$ sorted elements, and the correct $m$ elements at the end of the array. After this, all that is left to do is put these $2^x - 1$ in eytzinger order, which we already know how to do with repeated calls to outshuffle. Doing this will result in all $n$ elements being correctly placed in an eytzinger ordered array.

It is important to note here that we will consistently use the C++ standard template library’s rotate to perform a rotate operation. This is done in time linear in proportion to the number of items being rotated.
Up to this point we have been using outshuffle as a black box, but do not know how to actually perform an outshuffle. This is the bulk of the work when looking at permuting from sorted order to eytzinger order. With a high level understanding of moving from sorted order to eytzinger order, we can now look at different methods to perform an outshuffle.

4.1 Prime outshuffle

When performing an outshuffle on an array, there is an interesting observation to be made with how the elements are shuffled around. For certain values of $n$, each element involved in the outshuffle on $n$ items moves to a new location. In addition to this, following the cycle of moving one item, say that is at position $i_a$ to its post-outshuffle position, $i_b$, and moving the item that used to be at $i_b$ to it’s new post-outshuffle position and so on and so forth creates a cycle of length $n$. If this is the case, we will denote all such values of $n$ as ‘shuffle primes’. If $n$ is a shuffle prime, we can compute the $i_b$ for any $i_a$ in constant time. We can use this knowledge, and prerequisite knowledge for different values of shuffle primes to perform a full outshuffle. To do this is fairly straightforward.

1. Get the largest shuffle prime, here on known as $m$ where $m \leq n$
2. Follow the cycle through $0..(m − 1)$.
3. Recurse on $n − m$.
4. Call rotate to ensure the even ordered indices and odd ordered indices are grouped after the recursion completes.

To complete step 1 in the most efficient time possible, we hard code shuffle primes from $n = 2$ up to $n = 15136693483361901396$. Step 2 can be done using the example code in Figure 10. We then recurse on step 1 and step 2 until there are no more items to outshuffle. We are then left with different blocks of grouped even and odd indexed items that we must group together. With repeated calls to rotate (through the recursion), we then are left with a complete outshuffle on $n$ items.

The overall complexity for this method is $O(n)$. Step 1 is constant time, we perform this operation at most $\log(n)$ times. Step 2 is linear in time with respect to $m$. We then recurse on
$n - m$ items. Afterwards, we do a rotate on at most $n$ items, recursing on at most $n/2$ items. The overall time complexity is then $O(\log(n)) + O(n) + O(2n)$. This is overall $O(n)$. We will look at the actual runtime of an implemented ‘prime outshuffle’ later, as well as some of the other outshuffle methods described in this section.

```
template<typename I>
inline I outshuffle_perm3(I i, I n) {
    I out = i/2;
    I fix = (i & 1) ? 0 : n/2;
    return out+fix;
}
```

Figure 10: Takes an index $i_a$, and size of the current shuffle prime, and returns the index $i_b$.

### 4.2 Blocked Outshuffle

We now look at the fastest current implementation of outshuffle, referred to as ‘blocked outshuffle’. We utilize the trick that we can hard code the sequence for a shuffle prime, say of size $2B$, and end up with ‘blocks’ of size $B$ of ordered, grouped, even and odd indexed elements. Performing this operation on all the blocks of size $2B$ in $n - n \% 2B$ items, we then effectively reduce the array size in question to a size of $n/B + n \% 2B$. To reiterate and give a full example of how to perform a blocked outshuffle, here are the complete steps involved.

1. Let $m$ be the number of elements in which we can from groups of $2B$, then $m = n - n \% 2B$
2. For each group of size $2B$ in $arr$, we perform an outshuffle; this is very efficient because we have a reference to the shuffle prime cycle that correlates to $n = 2B$
3. Call `prime_outshuffle` on $m/(2B)$ items, where each item here is a block of size $B$ in the array $arr$
4. Call prime outshuffle on the $n \% 2B$ items at the end of $arr$
5. We are left with 4 groups in $arr$, two groups of the even indexed items, and two groups of the odd indexed items. Perform a call to rotate to group the odd and even indexed items to complete the outshuffle.
See Figure 11 for a visualization of what happens during a blocked outshuffle. Blocked outshuffle still uses prime outshuffle to do the heavy lifting, however it effectively reduces the work that prime outshuffle has to do. This is the algorithm we have found the best current results for, which we will highlight later.

4.3 Swap Outshuffle

There is a third algorithm I want to highlight mainly because the primary idea was one of my own creation, and also I believe it could have potential. This is an outshuffle algorithm that mainly utilizes swapping two consecutive elements in an array. It does the exact same steps as blocked outshuffle discussed in 4.2, but blocks differently. Instead of knowing a shuffle prime’s cycle, we use a swap to group two even and odd indexed elements. For this reason, we will focus on expanding Step 2 from section 4.2, but using the swap method of grouping instead.

For a given arr of size n, repeated calls to a swap function effectively groups elements. If we consider the first 4 elements of an array where $n > 3$, swapping the second and third element will effectively group the items indexed at 1 and 3, as well as group the items indexed at 2 and 4. Repeating this on all groups of 4 within arr will result in blocking the array where $B = 2$. For an implementation in c++, see Figure 12.
We can then recurse on this ‘group by swapping’ algorithm, treating each block as a single element. After each step in the recursion, the block size doubles. This brings up one slight disadvantage, which is you can only block to a size of the form \(2^k\).

It is also worth noting that if we recurse enough, we would end up completing the outshuffle and not even needing to do step 4 described in section 4.2. In this case however, if the data is not exactly of the form \(2^k\), we must take into account that some of the blocks are different sizes which can pose issues. This can be solved by using rotate instead of swap, but is much less efficient.

The results of the swap outshuffle that simply replace step 4 of blocked outshuffle are quite promising. To achieve a similar performance to that of blocked outshuffle where the block size is 96, we block to 1024 with the swap outshuffle. We will go further into performance later.

### 5 Eytzinger to Sorted Permutation

If we give the ability to go from sorted order to eytzinger order, we must also provide a way to go from eytzinger to sorted. Luckily, most of the heavy lifting has already been done. We simply have to code an inverse of all the functions used to permute from sorted to eytzinger. Let us first consider inshuffle. An inshuffle on \(n\) elements involves taking the first \(n/2\) elements, and placing them at all the odd indices of \(n\), all while retaining sorted order. The second \(n/2\) elements would therefore end up at all of the even indices of \(n\), and they too retain sorted order. For a visualization see Figure 8, except we now start at the second array.
and result in the first. An inshuffle is a basic build block when permuting from eytzinger to sorted. All of the different types of shuffles described in section 5 have a similar inshuffle function. For simplicity, we will only look at blocked inshuffle. Again, let \( m \) be the number of elements in which we can from groups of exactly \( 2B \), then \( m = n - n \% 2B \). Also, because \( 2B \) here is a shuffle prime, we know the shuffle cycle for a given \( 2B \). This allows us to have a version of inshuffle prebuilt so that we can easily inshuffle \( 2B \) items. The steps to perform a blocked inshuffle are as follows.

1. Rotate the array so we have \( m \) elements at the front of the array, and \( 2 \% 2B \) items at the end of the array. The \( m \) items, as well as the \( 2 \% 2B \) items, should both result in being structured as if it there was just an outshuffle performed.
2. Block the \( m \) elements into blocks of size \( B \), and call prime_inshuffle (the same as prime_outshuffle, just reversed) on the resulting \( m/B \) elements.
3. Perform the prebuilt inshuffle on each group of size \( 2B \) resulting from step 2.
4. Perform prime_inshuffle on the later \( 2 \% 2B \) items.

See figure 11 for a visualization, except start at step 5 and go back to step 1. All of the different inshuffles match the time complexity of their outshuffle counterparts. It may now be foreseen that going from eytzinger to sorted just entails repeated calls to inshuffle. To do this, just repeat inshuffle on \( i \) elements, starting at \( i = 3, 7, 15, ..., (2^k - 1) \) where \( k \) is the largest integer such that \( 2^k - 1 < n \). There is then \( m \) elements left to get in sorted order, where \( m = n - (2^k - 1) \), at the end of the array. Simply rotate these elements to the front of the array, and perform one last inshuffle on \( 2m \) items. The final result is a sorted order array.

**6 The API**

This report is overall based on implementing all of these algorithms in a usable c++ library, but the complex nature of the algorithms behind simple operations makes a good API important. I chose to specifically make everything available in one c++ header file, eytzinger.hpp. This header file, as well as all of the final source code, is available in the Appendix.
6.1 Iteration and Search

Because iteration and search are somewhat non trivial, we provide an Eytzinger class that can provide a good interface to do both. The class is initialized with an eytzinger ordered array, trying to instantiate an Eytzinger object with an array not in the eytzinger order will throw an exception. This decision is so people only use the class as it should be used, as usage of this class with a random array returns seemingly random results.

6.1.1 Consumer Code for Iteration

```cpp
#include <iostream>
#include "eytzinger.hpp"

int main(){
    const int n = 10;
    int a[n] = {7, 4, 9, 2, 6, 8, 10, 1, 3, 5};
    Eytzinger<int,int> arr(a, n);

    std::cout << arr.search(10) << std::endl;
    for(Eytzinger<int,int>::Iterator i = arr.begin(); i != arr.end(); i++){
        std::cout << *i << " ";
    }
    std::cout << std::endl;
}
```

Figure 13: Example search and iteration usage of the eytzinger API.

Once an eytzinger object is instantiated, you have access to a basic forward and a reverse iterator that abstracts out the next and previous calls, as well as their implementation to the user. This is allows for simple and clean usage when wanting to in order traverse an eytzinger ordered array. See Figure 13 for example usage of the eytzinger library created for iteration.
6.1.2 Consumer Code for Search

Search is a very straightforward function once you have constructed an Eytzinger object. Search will return the index where the element lies within the eytzinger ordered array, or \( n \) if it does not exist. There is also a very similar function, \( \text{upper\_bound} \) that does the same thing as search, except will return the index of the smallest item that is greater than or equal to the item we are searching for. See Figure 13 for an example of using the search function with an eytzinger object.

6.2 Array Manipulation Functions

By including the eytzinger header file, it will also provide access to general usage functions for array manipulation. Specifically, an in place \( \text{to\_eytzinger} \) function as well as an in place \( \text{to\_sorted} \) function. The shuffles used by these algorithms are also exposed to the user.

7 Results

The main reason for exploring different shuffle algorithms stems from wanting to make the eytzinger layout more accessible. For the layout to be more accessible, we need to be able to efficiently construct the layout, in place, from a sorted array. Outshuffle is the main building block to accomplish this, and for that reason we look at accomplishing an outshuffle in the most efficient manner possible. We also explore in order traversal, and the time it takes as compared to just traversing a sorted array.

7.1 Outshuffle Runtimes

Blocked outshuffle (when using a block size \( B = 96 \)) currently performs the fastest when compared to all of the other outshuffle algorithms explored during the project. When compared to swap_outshuffle, it is only marginally slower. It is important to note that the swap outshuffle compared here preformed swaps until the block size was \( B = 1024 \). Both
prime_outshuffle and jain_outshuffle (a fourth outshuffle algorithm not discussed here, see the Appendix for an implementation) perform ~4.5 times slower when used without any blocking techniques. See Figure 14 for a quick visualization of real runtimes.

<table>
<thead>
<tr>
<th>Outshuffle</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocked_outshuffle</td>
<td>1.48604s</td>
</tr>
<tr>
<td>swap_outshuffle</td>
<td>1.51428s</td>
</tr>
<tr>
<td>prime_outshuffle</td>
<td>8.99839s</td>
</tr>
<tr>
<td>jain_outshuffle</td>
<td>8.581s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sorted to Eytzinger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using blocked_outshuffle</td>
</tr>
<tr>
<td>Using swap_outshuffle</td>
</tr>
<tr>
<td>Using prime_outshuffle</td>
</tr>
<tr>
<td>Using jain_outshuffle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic iteration on a sorted array</td>
</tr>
<tr>
<td>Basic iteration on an eytzinger array, using the pre increment operator</td>
</tr>
<tr>
<td>Basic iteration on an eytzinger array, using the post increment operator</td>
</tr>
</tbody>
</table>

Figure 14: A summary of results (all on \( n = 10^8 \) items).

### 7.2 Permuting from Sorted to Eytzinger Runtimes

Because an outshuffle is again the main building block of permuting from sorted order to eytzinger order, the runtimes here are very correlated to that of the outshuffle. Performing a to_eytzinger permutation using blocked_outshuffle is the fastest, with swap_outshuffle about 500ms behind on \( n = 10^8 \) items. Using prime_outshuffle and jain_outshuffle as the building blocks fall behind very quickly.
7.3 Iteration Runtimes

In-order iteration through an eytzinger ordered array currently takes much more time than that of sorted order iteration. Note at each step of the iteration we perform modular arithmetic on the item being accesses as to stray away from compiler optimizations of purely iterating through a sorted array. When iterating over a sorted array, the only time taking into account is incrementing the current index, and dereferencing the array at that index. This is much faster than the calculations we used to arrive at the index of the element succeeding any index of an element in an eytzinger ordered array. Also note that with the post increment operator, it takes much more time (\textit{i++ is a post increment where as ++i is a pre increment}). This is because in order to properly perform a post increment, we have to make a copy of the iterator at each step.

8 Conclusion

The eytzinger ordered array is an interesting layout that provides for complex challenges and a lot of exploration into runtime optimization. The main reason to use this layout is for more optimal search times on large values of n. To make the data structure more accessible, we have implemented a c++ library of the current most efficient algorithms for switching to and from sorted order, as well as iteration. Blocked outshuffle is currently most optimal, and used in the library's implementation of a \textit{to_eytzinger} function. For iteration, we stray away from the trivial implementation and use the properties of the binary representation of each index to be smarter about how to iterate efficiently. With the current lack of performance on iteration, this data structure should be used situationally used. If the time to construct the eytzinger layout plus the time to perform \( k \) searches on the resulting layout is less than that of performing \( k \) binary searches on the sorted array, then it is clear eytzinger should be used. The c++ library implemented allows easy access to do just that.
8.1 Future Work

With any library, there is always room to make improvements and additions. There are three areas that could be potentially beneficial to explore. First, implementing the ability to insert and delete from an eytzinger ordered array would be the same implementation as that of maintaining a balanced binary search tree. This would expand the capabilities of the library. Second, the iteration is depressingly slower than that of a sorted array. If iteration was just as quick to that of incrementing an integer, the negative side effects of the eytzinger layout would be almost negated and this layout should be preferred in most cases over sorted order. Thirdly, a deeper dive into the possibilities of the idea to use a swapping method to perform a full outshuffle could provide beneficial to reduce the runtimes of permuting from sorted order to eytzinger order (and vice versa). The main roadblock encountered when attempting to do this was that the blocking method used required the creation of an object which has an instance array of dynamic size. C++ does not allow for arrays of variable length, so you would have to have all of the objects used for blocking readily available. This is problematic because for different values of $n$, you would only need certain block sizes. Overall, eytzinger is a promising layout that could positively impact performances across many areas of computer science that perform a large number of search operations.
References

Appendix

Eytzinger.hpp

```cpp
#include <iostream>
#include <algorithm>

//shuffle primes

/**************************************************************************
 General Utility Functions
**************************************************************************/

template<typename Data, typename Index>
int is_eytzinger(Data *a, Index length){
    for(Index i = 0; i < length; i++){
        if(2 * i + 1 == length){ continue; }
        if(a[i] < a[2 * i + 1]){ return false; }

        if(2 * i + 2 == length){ continue; }
        if(a[i] > a[2 * i + 2]){ return false; }
    }

    return true;
}

/**************************************************************************/
template<typename I>
inline I outshuffle_perm3(I i, I n) {
    I out = i/2;
    I fix = (i & 1) ? n/2;
    return out+fix;
}

template<typename Data, typename Index>
void prime_outshuffle(Data *a, Index n) {
    if (n <= 1) return;

    // Compute appropriate value of m
    int i = 0;
    while (sprimes[i+1] <= n) i++;
    Index m = sprimes[i];

    // Follow the cycle through a[0,...,m-1];
    Index cur = 0;
    Data t[2];
    int flipflop = 0;
    t[flipflop] = a[cur];
    do {
        Index nxt = outshuffle_perm3(cur, m);
        t[!flipflop] = a[nxt];
        a[nxt] = t[flipflop];
        flipflop = !flipflop;
        cur = nxt;
    } while (cur != 0);

    // Recurse on a[m,...,n-1]
    prime_outshuffle(a+m, n-m);

    // Regroup odds and evens
    std::rotate(a+m/2, a+m, a+m+(n-m)/2);
}

template<unsigned R, typename Data, typename Index>
void prime_outshuffle_pf(Data *a, Index n) {
    if (n <= 1) return;

    // Compute appropriate value of m
    int i = 0;
    while (sprimes[i+1] <= n) i++;
    Index m = sprimes[i];

    Index cur = 0;
    Index pf = cur;
    for (unsigned i = 0; i < R; i++)
        __builtin_prefetch(a+(pf = outshuffle_perm3(pf, m)), 1);

    Data t[2];
    int flipflop = 0;
    t[flipflop] = a[cur];
    do {
        __builtin_prefetch(a+(pf = outshuffle_perm3(pf, m)), 1);
        Index nxt = outshuffle_perm3(cur, m);
        t[!flipflop] = a[nxt];
        a[nxt] = t[flipflop];
        flipflop = !flipflop;
        cur = nxt;
    } while (cur != 0);

    // Recurse on a[m,...,n-1]
void
template<}

// Regroup odds and evens
std::rotate(a+m/2, a+m, a+m+(n-m)/2);
}

template<typename Data>
void rev_outshuffle192(Data *a) {
  const static int A = 2, B = 95;
  const static std::uint32_t cycles[A][B] = {
    {1, 96, 48, 24, 12, 6, 3, 97, 144, 72, 36, 18, 9, 100, 50, 25, 108, 54,
      27, 109, 150, 75, 133, 162, 81, 136, 68, 34, 17, 104, 52, 26, 13, 102, 51, 121, 156, 78,
      128, 60, 30, 15, 103, 147, 169, 180, 90, 45, 118, 59, 125, 158, 79, 135, 163, 177, 184,
      92, 46, 23, 107, 149, 170, 85, 138, 69, 130, 65, 128, 64, 32, 16, 8, 4, 2,
    },
    {7, 99, 145, 168, 84, 42, 21, 106, 53, 122, 61, 126, 63, 127, 159, 175,
      183, 187, 189, 190, 95, 143, 167, 179, 185, 189, 94, 47, 119, 155, 173, 182, 91, 141,
      166, 83, 137, 164, 82, 41, 126, 58, 29, 110, 55, 123, 157, 174, 87, 139, 165, 178, 89,
      148, 70, 35, 113, 152, 76, 38, 19, 105, 148, 74, 37, 114, 57, 124, 62, 31, 111, 151, 171,
      181, 186, 93, 142, 71, 131, 161, 176, 88, 44, 22, 11, 101, 146, 73, 132, 66, 33, 112, 56,
      28, 14}
  }
  for (int i = 0; i < A; i++) {
    Data t1 = a[cycles[i][B-1]];
    for (int j = 0; j < B; j++) {
      Data t2 = a[cycles[i][j]];
      a[cycles[i][j]] = t1;
      t1 = t2;
    }
  }
}

template<unsigned B = 96, bool prefetch=false, typename Data, typename Index>
void blocked_outshuffle(Data *a, Index n) {
  Index r = n % (2*B);
  Index m = n - r;
  prime_outshuffle(a+m, r);
  for(Index i = 0; i < m/(2*B); i++) {
    static_assert(B==96, "Assuming B==96");
    rev_outshuffle192(a+(2*B)*i);
  }
  struct block { Data dummy[B]{};};
  if (prefetch) {
    prime_outshuffle_pf4((block*)a, m/B);
  } else {
    prime_outshuffle((block*)a, m/B);
  }
  // Regroup odds and evens
  std::rotate(a+m/2, a+m, a+m+r/2);
}

template<typename Data, typename Index>
void to_eytzinger(Data *a, Index n) {
  const unsigned BLOCK=96;
  int i = 0;
if (((n + 1) & n) != 0) {
    i |= 1 << (32 - __builtin_clz(n) - 1);
    i = n - (i - 1);

    blocked_outshuffle<BLOCK>(a, i * 2);
    std::rotate(a, a + i, a + i * 2);
    std::rotate(a, a + i, a + n);
}

int todo = n - i;
while (todo > 1) {
    blocked_outshuffle<BLOCK>(a, todo);
    todo = todo / 2;
}

/*--------------------------------------------
Algorythms to change from sorted to eytzinger in place
--------------------------------------------*/

template<typename I>
inline I inshuffle_perm3(I i, I n) {
    I out = 2*i + 1;
    I fix = (i < n/2) ? 0 : n+1;
    return out-fix;
}

template<typename Data, typename Index>
void prime_inshuffle(Data *a, Index n) {
    while (n > 1) {
        // Compute appropriate value of m
        int i = 0;
        while (sprimes[i+1] <= n) i++;
        Index m = sprimes[i];

        // Move correct m elements to front of the array
        std::rotate(a+m/2, a+n/2, a+n/2+m/2);

        // Now use Jain's trick to shuffle a[0,...,m-1];
        Index cur = 0;
        Data t = a[cur];
        do {
            Index nxt = inshuffle_perm3(cur, m);
            Data t2 = a[nxt];
            a[nxt] = t;
            t = t2;
            cur = nxt;
        } while (cur != 0);

        // Recurse on a[m,...n-1]
        a += m;
        n -= m;
    }

    // implemented an inshuffle on 146 items
    template<typename Data>
    void inshuffle146(Data* a) {
        const int A = 292;

        for (int i = 0; i < A; i++) {
            if ((n + 1) & n) != 0) {
    i |= 1 << (32 - __builtin_clz(n) - 1);
    i = n - (i - 1);

    blocked_outshuffle<BLOCK>(a, i * 2);
    std::rotate(a, a + i, a + i * 2);
    std::rotate(a, a + i, a + n);
}

int todo = n - i;
while (todo > 1) {
    blocked_outshuffle<BLOCK>(a, todo);
    todo = todo / 2;
}

/*--------------------------------------------
Algorythms to change from sorted to eytzinger in place
--------------------------------------------*/

template<typename I>
inline I inshuffle_perm3(I i, I n) {
    I out = 2*i + 1;
    I fix = (i < n/2) ? 0 : n+1;
    return out-fix;
}

template<typename Data, typename Index>
void prime_inshuffle(Data *a, Index n) {
    while (n > 1) {
        // Compute appropriate value of m
        int i = 0;
        while (sprimes[i+1] <= n) i++;
        Index m = sprimes[i];

        // Move correct m elements to front of the array
        std::rotate(a+m/2, a+n/2, a+n/2+m/2);

        // Now use Jain's trick to shuffle a[0,...,m-1];
        Index cur = 0;
        Data t = a[cur];
        do {
            Index nxt = inshuffle_perm3(cur, m);
            Data t2 = a[nxt];
            a[nxt] = t;
            t = t2;
            cur = nxt;
        } while (cur != 0);

        // Recurse on a[m,...n-1]
        a += m;
        n -= m;
    }

    // implemented an inshuffle on 146 items
    template<typename Data>
    void inshuffle146(Data* a) {
        const int A = 292;
template<
  unsigned B = 146,
  typename Data,
  typename Index
>
void blocked_inshuffle(Data *a, Index n) {
  if (n <= 1) { return; }
  Index r = n % (2*B);
  Index m = n - r;
  if(m > 0) {
    std::rotate(a + m / 2, a + m / 2 + r / 2, a + m + r / 2);
    struct block { Data dummy[B];
    prime_inshuffle(((block*)a) + 1, m/B - 1);
    for(int i = 0; i < m/(2*B); i++) {
      inshuffle146(a+(2*B)*i);
    }
  }
  prime_inshuffle(a + m, r);
}

template<
  typename Data,
  typename Index
>
void to_sorted(Data* a, Index n) {
  if (n <= 1) { return; }
  int i = 0;
  if (((n + 1) & n) != 0) {
    i |= 1 << (32 - __builtin_clz(n) - 1);
    i = n - (i - 1);
  }
  for(int j = 2; j - 1 <= n - i; j *= 2) {
    blocked_inshuffle(a, j - 1);
  }
  std::rotate(a, a + n - 1, a + n);
  blocked_inshuffle(a + 1, i * 2 - 1);
template <class Data, typename Index>
class Eytzinger {
    Data *arr;
    Index n;
public:
    class Iterator{
        public:
            Iterator(Index *p, Eytzinger *d) : pos(p), data(d) {};
            Iterator(Iterator* src) : data(src->data) { pos = new Index(*src->pos); }
            ~Iterator() { delete pos; }

            Iterator& operator++() {*pos = data->next(*pos); return *this; }
            Iterator operator++(int) {Iterator temp(this); *pos = data->next(*pos);
            return temp; }
            bool operator==(const Iterator& other) { return *pos == *other.pos; }
            bool operator!=(const Iterator& other) { return *pos != *other.pos; }

            Data& operator*() { return data->arr[*pos]; }
        protected:
            Index *pos;
            Eytzinger *data;
    }

class IteratorReverse{
        public:
            IteratorReverse(Index *p, Eytzinger *d) : pos(p), data(d) {};
            IteratorReverse(IteratorReverse* src) : data(src->data) { pos = new Index(*src->pos); }
            ~IteratorReverse() { delete pos; }

            IteratorReverse& operator++() {*pos = data->prev(*pos); return *this; }
            IteratorReverse operator++(int) {IteratorReverse temp(this); *pos = data->prev(*pos);
            return temp; }
            bool operator==(const IteratorReverse& other) { return *pos == *other.pos; }
            bool operator!=(const IteratorReverse& other) { return *pos != *other.pos; }

            Data& operator*() { return data->arr[*pos]; }
        protected:
            Index *pos;
            Eytzinger *data;
    }
Iterator begin()
{
    Index* first = new Index((1 << (32 - __builtin_clz(n) - 1)) - 1);
    return Iterator(first, this);
}

Iterator end()
{
    Index* last = new Index(-1);
    return Iterator(last, this);
}

IteratorReverse rbegin()
{
    Index* first = new Index(~(~0 << (32 - __builtin_clz(n + 1) - 1)) - 1);
    return IteratorReverse(first, this);
}

IteratorReverse rend()
{
    Index* last = new Index(-1);
    return IteratorReverse(last, this);
}

Eytzinger (Data *arr, Index n);
Index search(Data element);
Index upper_bound(Data element);
Index next(Index i);
Index prev(Index i);
};

template<class Data, typename Index>
Eytzinger<Data, Index>::Eytzinger(Data *arr, Index n) {
    if (!is_eytzinger(arr, n))
        throw std::invalid_argument("'Data* arr' of size 'n' given to Eytzinger<Data, Index>::Eytzinger(Data* arr, Index n) not of the form: Eytzinger.");
    this->arr = arr;
    this->n = n;
}

// Implemented branchy_search
template<class Data, typename Index>
Index Eytzinger<Data, Index>::search(Data element) {
    Index i = 0;
    while (i < n) {
        i = (element <= arr[i]) ? (2 * i + 1) : (2 * i + 2);
    }
    Index j = (i + 1) >> __builtin_ffs(~(i + 1));
    return arr[j - 1] == element ? j - 1 : n;
}

// Implemented branchy_search straight from Eytzinger paper
template<class Data, typename Index>
Index Eytzinger<Data, Index>::upper_bound(Data element) {
    Index i = 0;
    while (i < n) {
        i = (element <= arr[i]) ? (2 * i + 1) : (2 * i + 2);
    }
    Index j = (i + 1) >> __builtin_ffs(~(i + 1));
// Returns the index of the item succeeding the eytzinger layed-out array element
arr[i]

template<class Data, typename Index>
Index Eytzinger<Data, Index>::next(Index i){
    if (2 * i > n - 1)
        int j = (i + 1) >> __builtin_ffs(~(i + 1));
    return j - 1;
} else{
    int iDepth = 32 - __builtin_clz(i + 1);
    int treeDepth = 32 - __builtin_clz(n);
    int j = (((i + 1) << 1) << (treeDepth - iDepth - 1));
    return (j - 1) > (n - 1) ? (j - 1) >> 1 : (j - 1);
}
}

template<class Data, typename Index>
Index Eytzinger<Data, Index>::prev(Index i){
    if (2 * i + 1 > n - 1)
        int j = (i + 1) >> __builtin_ffs(i + 1);
    return j - 1;
} else{
    int iDepth = 32 - __builtin_clz(i + 1);
    int treeDepth = 32 - __builtin_clz(n);
    int j = ~(~((i + 1) << 1) << (treeDepth - iDepth - 1));
    return (j - 1) > (n - 1) ? ((j-1) >> 1) - 1 : j - 1;
}
}

Everything below here is dealing with in place algorithms
to move from sorted order to eytzinger order, and vise-versa.
None of these are not used in the implementation for to_eytzinger
and to_sorted as they are currently lacking in performance.
---------------------------------------------------------------------*/

/*--------------------------------------------
Using Jain's trick for shuffles
---------------------------------------------------------------------*/

template<typename Data, typename Index>
void jain_inshuffle(Data *a, Index n) {
    while (n > 1) {
        // Compute appropriate value of m
        Index m = 1;
        while (3*m-1 <= n) m *= 3;
        m -= 1;
        // Move correct m elements to front of the array
        std::rotate(a+m/2, a+n/2, a+n/2+m/2);
        // Now use Jain's trick to shuffle a[0,...,m-1];
        for (Index g = 1; g < m; g *= 3) {
            Index cur = g-1;
            Data t = a[cur];
            do {
                Index nxt = inshuffle_perm3(cur, m);
                Data t2 = a[nxt];
            }
        }
    }
}
```cpp
void jain_outshuffle(Data *a, Index n) {

    if (n <= 1) return;

    // Compute appropriate value of m
    Index m = 1;
    while (3 * m - 1 <= n) m *= 3;
    m -= 1;

    // Use Jain's trick to shuffle a[0,...,m-1];
    for (Index g = 1; g < m; g *= 3) {
        Index cur = g - 1;
        Data t[2];
        int flipflop = 0;
        t[flipflop] = a[cur];
        do {
            Index nxt = outshuffle_perm3(cur, m);
            t[!flipflop] = a[nxt];
            a[nxt] = t[flipflop];
            flipflop = !flipflop;
            cur = nxt;
        } while (cur != g - 1);
    }

    // Recurse on a[m,...n-1]
    jain_outshuffle(a + m, n - m);
}

// Regroup odds and evens
std::rotate(a + m / 2, a + m, a + m + (n - m) / 2);
}
```

```cpp
int to_eytzinger_jain(Data *a, Index n) {

    int i = 0;
    if (((n + 1) & n) != 0) {
        i |= 1 << (32 - __builtin_clz(n) - 1);
        i = n - (i - 1);

        jain_outshuffle(a, i * 2);
        std::rotate(a, a + i, a + i * 2);
        std::rotate(a, a + i, a + n);
    }

    int todo = n - i;

    while (todo > 1) {
        jain_outshuffle(a, todo);
        todo = todo / 2;
    }
```
template<typename Data, typename Index>
void swap_next(Data *a, Index i){
    Data x = a[i + 1];
    a[i+1] = a[i];
    a[i] = x;
}

template<typename Data, typename Index>
void blockto_1024_custom_swap(Data *a, Index n){
    struct block1 { Data dummy[1]; };  
    struct block2 { Data dummy[2]; };  
    struct block4 { Data dummy[4]; };  
    struct block8 { Data dummy[8]; };  
    struct block16 { Data dummy[16]; }; 
    struct block32 { Data dummy[32]; }; 
    struct block64 { Data dummy[64]; }; 
    struct block128 { Data dummy[128]; }; 
    struct block256 { Data dummy[256]; }; 
    struct block512 { Data dummy[512]; }; 

    block1* b1 = (block1*) a;
    for(int i = 1; i < n / 1 - 1; i += 4){
        swap_next(b1, i);
    }

    block2* b2 = (block2*) a;
    for(int i = 1; i < n / 2 - 1; i += 4){
        swap_next(b2, i);
    }

    block4* b4 = (block4*) a;
    for(int i = 1; i < n / 4 - 1; i += 4){
        swap_next(b4, i);
    }

    block8* b8 = (block8*) a;
    for(int i = 1; i < n / 8 - 1; i += 4){
        swap_next(b8, i);
    }

    block16* b16 = (block16*) a;
    for(int i = 1; i < n / 16 - 1; i += 4){
        swap_next(b16, i);
    }

    block32* b32 = (block32*) a;
    for(int i = 1; i < n / 32 - 1; i += 4){
        swap_next(b32, i);
    }
}
block64* b64 = (block64*) a;
    for(int i = 1; i < n / 64 - 1; i += 4){
        swap_next(b64, i);
    }
block128* b128 = (block128*) a;
    for(int i = 1; i < n / 128 - 1; i += 4){
        swap_next(b128, i);
    }
block256* b256 = (block256*) a;
    for(int i = 1; i < n / 256 - 1; i += 4){
        swap_next(b256, i);
    }
block512* b512 = (block512*) a;
    for(int i = 1; i < n / 512 - 1; i += 4){
        swap_next(b512, i);
    }

}template<typename Data, typename Index>
void blockto_1024(Data *a, Index n){
    struct block1 { Data dummy[1];
    };
    struct block2 { Data dummy[2];
    };
    struct block4 { Data dummy[4];
    };
    struct block8 { Data dummy[8];
    };
    struct block16 { Data dummy[16];
    };
    struct block32 { Data dummy[32];
    };
    struct block64 { Data dummy[64];
    };
    struct block128 { Data dummy[128];
    };
    struct block256 { Data dummy[256];
    };
    struct block512 { Data dummy[512];
    };

    block1* b1 = (block1*) a;
    for(int i = 1; i < n / 1 - 1; i += 4){
        // std::rotate(b1 + i, b1 + i + 1, b1 + i + 2);
        swap_next(b1, i);
    }
block2* b2 = (block2*) a;
    for(int i = 1; i < n / 2 - 1; i += 4){
        std::rotate(b2 + i, b2 + i + 1, b2 + i + 2);
    }
block4* b4 = (block4*) a;
    for(int i = 1; i < n / 4 - 1; i += 4){
        std::rotate(b4 + i, b4 + i + 1, b4 + i + 2);
    }
block8* b8 = (block8*) a;
    for(int i = 1; i < n / 8 - 1; i += 4){
        std::rotate(b8 + i, b8 + i + 1, b8 + i + 2);
    }
block16* b16 = (block16*) a;
for (int i = 1; i < n / 16 - 1; i += 4) {
    std::rotate(b16 + i, b16 + i + 1, b16 + i + 2);
}

block32* b32 = (block32*) a;
for (int i = 1; i < n / 32 - 1; i += 4) {
    std::rotate(b32 + i, b32 + i + 1, b32 + i + 2);
}

block64* b64 = (block64*) a;
for (int i = 1; i < n / 64 - 1; i += 4) {
    std::rotate(b64 + i, b64 + i + 1, b64 + i + 2);
}

block128* b128 = (block128*) a;
for (int i = 1; i < n / 128 - 1; i += 4) {
    std::rotate(b128 + i, b128 + i + 1, b128 + i + 2);
}

block256* b256 = (block256*) a;
for (int i = 1; i < n / 256 - 1; i += 4) {
    std::rotate(b256 + i, b256 + i + 1, b256 + i + 2);
}

block512* b512 = (block512*) a;
for (int i = 1; i < n / 512 - 1; i += 4) {
    std::rotate(b512 + i, b512 + i + 1, b512 + i + 2);
}

}

template<
unsigned B=1024,
bool prefetch=false,
typename Data,
typename Index>
void swap_outshuffle(Data *a, Index n) {
    Index r = n % (2*B);
    Index m = n - r;
    prime_outshuffle(a+m, r);
    blockto_1024_custom_swap(a, m);

    struct block { Data dummy[B]; }
    prime_outshuffle((block*)a, m/B);
    // Regroup odds and evens
    std::rotate(a+m/2, a+m, a+m+r/2);
}

template<
typename Data,
typename Index>
int to_eytzinger_swap(Data *a, Index n) {
    int i = 0;
    if (((n + 1) & n) != 0) {
        i = 1 << (32 - __builtin_clz(n) - 1);
        i = n - (i - 1);
```
swap_outshuffle(a, i * 2);
std::rotate(a, a + i, a + i * 2);
std::rotate(a, a + i, a + n);
}

int todo = n - i;
while(todo > 1){
    swap_outshuffle(a, todo);
    todo = todo / 2;
}

/*--------------------------------------------
Using std::rotate as a building block for outshuffle
--------------------------------------------*/

template<typename Data, typename Index>
void outshuffle(Data *a, Index n) {
    if (n <= 1) return;
    if (n <= 3){
        std::rotate(a, a + 1, a + 2);
        return;
    }

    int last = n;
    int every = 4;
    for(int i = 1; i < n / 2; i *= 2){
        for(int j = i; j < n; j += every){
            std::rotate(a + j, a + j + i, a + j + i * 2);
        }
        every *= 2;
    }

    std::rotate(a, a + n / 2 + 1, a + n);
}

int preshuffle(Data *a, Index n) {
    // No need to preshuffle
    if (((n + 1) & n) == 0) return n;

    int i = 0;
    i |= 1 << (32 - __builtin_clz(n) - 1);

    int items = n - (i - 1);
    i = 2 * items - 1;
    for(int j = 2 * items - 3; j >= 1; j -= 2){
        std::rotate(a + j, a + j + 1, a + i);
        i -= 1;
    }

    std::rotate(a, a + items, a + n);
    return n - items;
}
```
template<typename Data, typename Index>
int preshuffle_2(Data *a, Index n) {
    // No need to preshuffle
    if (((n + 1) & n) == 0) return n;

    // we need to preshuffle i items, without messing up [last + 1..n-1]
    int i = 0;
    i |= 1 << (32 - __builtin_clz(n) - 1);
    i = n - (i - 1);
    int last = 2 * i - 1;

    int every = 4;
    for (int k = 1; k < i; k *= 2) {
        for (int j = k; j < i * 2; j += every) {
            int edge = j + k * 2;
            if (edge > last) {
                edge = last;
                last = last - k;
            }
            std::rotate(a + j, a + j + k, a + edge);
        }
        every *= 2;
    }
    std::rotate(a, a + i, a + n);

    return n - i;
}

// Initial results showing this yields minimal improvement over preshuffle_2
// This is currently not working... ***

template<typename Data, typename Index>
int preshuffle_3(Data *a, Index n) {
    // No need to preshuffle
    if (((n + 1) & n) == 0) return n;

    int i = 0;
    i |= 1 << (32 - __builtin_clz(n) - 1);
    i = n - (i - 1);

    int every = 4;
    for (int k = 1; k < i; k *= 2) {
        for (int j = k; j < i * 2; j += every) {
            std::rotate(a + j, a + j + k, a + j + k * 2);
        }
        every *= 2;
    }
    std::rotate(a, a + i, a + n);

    // *** need to regroup l items, no clue how yet...
    // accounted for in preshuffle_2 but with an if statement in the nested for loop
    int l = 0;
    l |= 1 << (32 - __builtin_clz(l));
    l -= i;

    return n - i;
}

template<typename Data, typename Index>
int to_eytzinger_rotate(Data *a, Index n) {

int todo = preshuffle_2(a, n);

while(todo > 1){
    outshuffle(a, todo);
    todo = todo / 2;
}
}

template<typename Data, typename Index>
void rev_outshuffle(Data *a, Index n) {
    if (n <= 1) return;
    if (n <= 3){
        std::rotate(a, a + 1, a + 2);
        return;
    }
    std::rotate(a, a + n / 2, a + n);
    int i = pow(2, 32 - __builtin_clz(n) - 2);
    while(i > 0){
        for(int j = i; j < n; j += i * 4){
            std::rotate(a + j, a + j + 1, a + j + i * 2);
        }
        i = i / 2;
    }
}

template<typename Data, typename Index>
int postshuffle(Data *a, Index n) {
    // No need to postshuffle
    if (((n + 1) & n) == 0) return n;
    int i = 0;
    i |= 1 << (32 - __builtin_clz(n) - 1);
    i = n - (i - 1);
    std::rotate(a, a + n - i, a + n);
    for(int j = i - 1; j > 0; j--){
        std::rotate(a + j, a + j + 1, a + j * 2 + 1);
    }
}