1. Describe a method for performing a card shuffle of a list of $2n$ elements, by converting it into two lists. A card shuffle is a permutation where a list $L$ is cut into two lists $L_1$ and $L_2$, where $L_1$ is the first half of $L$ and $L_2$ is the second half of $L$, and then these two lists are merged into one by taking the first element in $L_1$, then the first element in $L_2$, followed by the second element in $L_1$, the second element in $L_2$, and so on.

**Algorithm.** The algorithm is shown in Algorithm 1 and essentially does as the problem suggests. Without loss of generality, assume that the Positional List ADT is used. Continually remove elements from $L$ to build $L_1$ and $L_2$ and then progressively take one card from each list to repopulate $L$.

**Algorithm 1 cardShuffle()**

<table>
<thead>
<tr>
<th>Input: List $L$ containing $2n$ elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Lists $L_1 ← \emptyset$, $L_2 ← \emptyset$</td>
</tr>
<tr>
<td>2. for $i ← 1 \ldots n$ do</td>
</tr>
<tr>
<td>3. $L_1.addLast(L.remove(L.first()))$</td>
</tr>
<tr>
<td>4. $L_2.addFirst(L.remove(L.last()))$</td>
</tr>
<tr>
<td>5. for $1 ← 1 \ldots n$ do</td>
</tr>
<tr>
<td>6. $L.addLast(L_1.remove(L_1.first()))$</td>
</tr>
<tr>
<td>7. $L.addLast(L_2.remove(L_2.first()))$</td>
</tr>
</tbody>
</table>

**Time Complexity.**

**Theorem 1.** Algorithm 1 runs in $O(n)$ time, where $2n$ is the number of elements in the input List.

*Proof.* Let the List be implemented using a doubly-linked-list. In this case, $addFirst()$, $addLast()$, $first()$, $last()$, and $remove(p)$ will run in $O(1)$ time. The first loop clearly executes $n$ iterations and will do $O(1)$ work per iteration. The second loop clearly executes $n$ iterations and will do $O(1)$ work per iteration. Thus, in total the algorithm can execute in $O(n)$ time. \qed

**Memory Complexity.**

**Theorem 2.** Algorithm 1 uses $O(1)$ additional memory.

*Proof.* At first glance, it might seem that there is $O(n)$ extra memory used by the algorithm. However, by removing the items from one list and moving them to another list, no extra memory is needed except temporarily storing the element in a temporary. Thus, the entire algorithm will run with $O(1)$ additional memory. \qed
2. A useful operation in databases is the natural join. If we view a database as a list of ordered pairs of objects, then the natural join of databases $A$ and $B$ is the list of all ordered triples $(x, y, z)$ such that the pair $(x, y)$ is in $A$ and the pair $(y, z)$ is in $B$. Describe and analyze an efficient algorithm for computing the natural join of a list $A$ of $n$ pairs and a list $B$ of $m$ pairs.

**Algorithm.** To compute the natural join of two lists of ordered pairs, we do the following: For each ordered pair of $A$ scan for candidate matches in $B$. If a candidate match is found copy the data to construct the ordered triplet. This process is shown in Algorithm 2.

```
Algorithm 2 Natural Join
Input: Databases (Lists) $A$ and $B$
Output: Database (List) of the natural join of $A$ and $B$
1: List $N \leftarrow \emptyset$
2: for $i = 1 \ldots n$ do
3:   Ordered pair $p_1 \leftarrow A\text{.get}(i)$
4:   for $j = 1 \ldots m$ do
5:     Ordered pair $p_2 \leftarrow B\text{.get}(i)$
6:     if $p_1\text{.second}() = p_2\text{.first}()$ then
7:       $N\text{.add}(N\text{.size}(),\{p_1\text{.first}(), p_1\text{.second}(), p_2\text{.second}()\})$
8: return $N$
```

**Time Complexity.**

**Theorem 3.** Algorithm 2 runs in $O(nm)$ time, where $n$ is the number of ordered pairs in database $A$ and $m$ is the number of ordered pairs in database $B$.

*Proof.* Let all Lists be implemented with a growable array. Because of this, all of the list operations used in the algorithm will run in $O(1)$ time. Note that the $\text{add}(i, e)$ function always adds at the end of the array and that we could preallocate enough room for $nm$ elements to avoid resizing entirely. The outer for-loop will clearly execute $n$ iterations of the inner for loop. The inner for-loop will clearly execute $nm$ iterations of $O(1)$ work for reasons already justified. In total, the algorithm is able to run in $O(nm)$ time.

**Memory Complexity.**

**Theorem 4.** Algorithm 2 uses $O(nm)$ additional memory in the worst case.

*Proof.* Essentially, the algorithm will copy data over and over to construct all triples of the natural join. So we must create a bound on the number of triples computed. This will be $O(nm)$ pairs in the worst case because the worst case performance is all elements of $A$ matching all elements of $B$.

3. **Bonus.** An array is sparse if most of its entries are null. A list $L$ can be used to implement such an array, $A$, efficiently. In particular, for each nonnull cell $A[i]$, we can store a pair $(i, e)$ in $L$, where $e$ is the element stored at $A[i]$. This approach allows us to represent $A$ using $O(m)$ storage, where $m$ is the number of nonnull entries in $A$. Describe and analyze efficient ways of performing the methods of the array list ADT on such a representation.

**Algorithm.** The essential idea is to store the elements in order based on index in a list. So all of the core operations (excluding $\text{size}()$ and $\text{isEmpty}()$) will perform a search for a specific node and then perform the update. They are as follows.
Algorithm 3 get(i)
\textbf{Input:} Index $i$
\textbf{Output:} Element $e$ at $i$
1: \textbf{for all } $n \in L$ \textbf{do}
2: \hspace{1em} \textbf{if } $n.\text{getIndex}() = i$ \textbf{then}
3: \hspace{2em} \textbf{return } $n.\text{getElement}()$
4: \hspace{1em} \textbf{return } null

Algorithm 4 set(i, e)
\textbf{Input:} Index $i$ and Element $e$
1: \textbf{for all } $n \in L$ \textbf{do}
2: \hspace{1em} \textbf{if } $n.\text{getIndex}() = i$ \textbf{then}
3: \hspace{2em} $n.\text{setElement}(e)$
4: \hspace{1em} \textbf{break}

Algorithm 5 add(i, e)
\textbf{Input:} Index $i$ and Element $e$
1: \textbf{Iterator } $n \leftarrow L.\text{begin}()$
2: \textbf{while } $n.\text{hasNext}() \land n.\text{getIndex}() < i$ \textbf{do}
3: \hspace{1em} $n \leftarrow n.\text{next}()$
4: \hspace{1em} \text{InsertBefore}(n, i)$
5: \textbf{while } $n.\text{hasNext}()$ \textbf{do}
6: \hspace{1em} $n.\text{updateIndex}()$ \{Only done if $i$ shifts all other indices in list\}
7: \hspace{1em} $n \leftarrow n.\text{next}()$

Algorithm 6 remove(i)
\textbf{Input:} Index $i$ and Element $e$
1: \textbf{for all } $n \in L$ \textbf{do}
2: \hspace{1em} \textbf{if } $n.\text{getIndex}() = i$ \textbf{then}
3: \hspace{2em} $L.\text{remove}(n)$
4: \hspace{1em} \textbf{break}

\textbf{Time Complexity.}
\textbf{Theorem 5.} Algorithms 3, 4, 5, and 6 all run in $O(m)$ time, where $m$ is the number of nonnull elements of the array.

\textit{Proof.} Essentially, all of the algorithms contain a search over all of the elements (in the worst case). Assuming, the list is implemented with either a growable array or a doubly-linked-list, then this search will be $O(m)$ time.

\textbf{Memory Complexity.}
\textbf{Theorem 6.} Algorithms 3, 4, 5, and 6 all use $O(1)$ additional memory.

\textit{Proof.} None of the algorithms use anything but a constant number of temporary variables. Thus, the additional memory size is $O(1)$. 

\hspace{1em}