Problem Set 1

Name: Gabriel Chiong

Collaborators: None

Problem 1-1.

(a) $(f_5, f_3, f_4, f_1, f_2)$

(b) $(f_1, f_2, f_5, f_4, f_3)$

(c) $(\{f_2, f_5\}, f_4, f_1, f_3)$

(d) $(f_5, f_2, f_1, f_3, f_4)$

Problem 1-2.

(a) We can recursively swap the outer most elements at index i and $i + k - 1$, with each recursive invocation excluding the pair that was swapped. The base case occurs when there are fewer than 2 elements to swap. This algorithm is correct by induction on the left most index i.

Since each insert and delete call causes the indexes of the later elements to shift, there is a certain order of operations we can use to minimize the effects of index shifting. At each recursive call, first define x_2 to be the result of the call to delete the $(i + k - 1)$th element. After which, we obtain the right most element x_1 from a call to deleting the ith element. When inserting the deleted elements in their swapped positions, x_2 must first be inserted at index i before x_1 is inserted at index $i + k - 1$.

There are four $O(\log n)$ operations in each of the $k/2$ recursive calls, therefore the running time of this algorithm is $O(k \log n)$ as required.

(b) We can recursively move the first element starting at index i to before index j, with each recursive call decreasing the value of k by 1. The base case occurs when there is less than 1 element left to move (when $k < 1$). The correctness of this algorithm is proven by induction on k, by maintaining the invariant that i is the index of the first item to be moved, k is the number of items to be moved, and j denotes the index of the item in front of which we must place the items.

The subtlety of this question lies in the need to handle both cases where $i < j$ and $i > j$. When $i < j$, removing the item at i causes the index of the element at j to be shifted down. Similarly, if $i > j$, inserting the element in front of j causes the index of the next recursive call's i to be shifted up.

There are two $O(\log n)$ operations in each of the k recursive calls, therefore the running time of this algorithm is $O(k \log n)$ as required.

Problem 1-3. The idea is to store the n pages in a static array of size $3n$, which can be rebuilt under certain size conditions. The requirement to `read_page(i)` in $O(1)$ time rules out the use of a linked-list data structure. Let us call this array S.

A call to `build(x)` the database takes $O(n)$ time in the worst-case, with $n = |x|$. The layout of the array is as follows:

- The P_1 elements until A occupies the beginning of S. We label the end of this as a_1.
- The next n slots are empty. We label the end of this subsequence a_2.
- The P_2 elements between A and B occupies the next portion of the array. We label the end of this subsequence b_1.
- The next n slots are empty. We label the end of this subsequence b_2.
- The P_3 last elements from B on-wards fill the rest of the array S.

To maintain the correctness of this data structure at all times, the invariant that P_1, P_2, P_3 are stored contiguously in that order with separation of greater than 0 array slots in between.

For `read_page(i)`, we need to consider three cases, depending on whether i is in either of P_1, P_2, P_3.

- If i is in P_1, return $S[i]$.
- If i is in P_2, return $S[a_2 + (i - (a_1 + 1))]$.
- If i is in P_3, return $S[b_2 + (i - (a_1 + 1) - (b_1 + 1))]$.

This returns the correct page as long as the separation invariant on the indices are maintained, and takes $O(1)$ worst-case time, based on array lookup and some arithmetic operations.

The `shift_mark(m,d)` operation for A only changes the position of the mark to either $a_1 + 1$ or $a_2 - 1$. Similarly, when applied to B, the position of the mark changes to either $b_1 + 1$ or $b_2 - 1$. This algorithm is correct as it maintains the index invariant. It only involves one array index lookup and one write, therefore, this takes $O(1)$ time in the worst-case.

The `move_page(m)` operation moves a page from an index in (a_1, b_1) to $(b_1 + 1, a_1 + 1)$ respectively. Any move needs to maintain the index invariant so that the algorithm runs correctly. If any call to `move_page(m)` breaks the separation invariant, the array will need to be rebuilt. The empty space changes by at most 1 slot on each call, and there are n empty slots between adjacent blocks of entries. Thus, the $O(n)$ rebuild is only required after n operations. Each operation takes $O(1)$ time otherwise, therefore this operation takes amortized $O(1)$ time.

Problem 1-4.

(a) The following descriptions involving a new element with `x` assumes that a linked-list node has been created to store data, `x`.

For `insert_first(x)`, first set `x.next` to `L.head`, `x.prev` to `None`, and `L.head.prev` to `x`. Finally, point `L.head` to `x`.

For `insert_last(x)`, point `L.tail.next` to `x`, `x.prev` to `L.tail`, and `x.next` to `None`. Finally, point `L.tail` to `x`.

For `delete_first()`, shift `L.head` to `L.head.next`, and set `L.head.prev` to be `None`. This effectively shifts `L.head` forwards by one element.

For `delete_last()`, shift `L.tail` to `L.tail.prev`, and set `L.tail.next` to be `None`. This effectively shifts `L.tail` backwards by one element.

(b) Set $x_1.prev.next$ to $x_2.next$, and $x_2.next.prev$ to $x_1.prev$. This removes the sublist from $x_1, \ldots, x_2$ inclusive. Then return the pointer to x_1.

(c) First connect L_2 to the correct nodes in L_1 by setting $L_2.head.prev$ to x and $L_2.tail.next$ to $x.next$. Next, we correctly fix the connections in L_1 by setting $x.next.prev$ to be $L_2.tail$ and $x.next$ to be $L_2.head$. Finally, remove the links in L_2 by setting both $L_2.head$ and $L_2.tail$ to be `None`.

(d) Submit your implementation to `alg.mit.edu`.