Exercise 2-1. Do Exercise 17.3–1 on p. 462 in CLRS.

Exercise 2-2. Do Exercise 17.3–3 on p. 462 in CLRS.

Problem 2-1. FIFO with MIN

Show how to implement dynamic set that implements a FIFO supporting the usual FIFO operations ENQUEUE and DEQUEUE, as well as a MIN operation that returns the minimum value of all the elements in the FIFO. All operations should run in $\Theta(1)$ amortized time. (Hint: Use a FIFO and a deque.) Use an accounting argument to analyze your data structure. Then use a potential-function argument.

Solution: Augment a regular FIFO with a deque (double-ended queue that allows append, pop operations to either end of the queue). All elements are stored in the FIFO, while the deque keeps track of the elements that could be returned by MIN at any time in the future. The key observation is that elements larger than the last element added to the FIFO can never be returned by MIN. To deal with non-unique values, we store tuples $(value, count)$ in the deque.

Min

```plaintext
1 return PEAK-FRONT(DEQUE).value
```

Enqueue($x$)

```plaintext
1 ENQUEUE(FIFO, $x$)
2 while $x < PEAK-DEQUE-END.DEQUE.value$
3 POP-END(DEQUE)
4 if $x == PEAK-DEQUE-END.DEQUE.value$
5 increment PEAK-DEQUE-END.DEQUE.count
6 else
7 APPEND-END(DEQUE, ($value = x, count = 1$))
```

Dequeue

```plaintext
1 $x = DEQUEUE(FIFO)$
2 if $x == PEAK-FRONT.DEQUE.value$
3 decrement PEAK-DEQUE-END.DEQUE.count
4 if PEAK-DEQUE-END.DEQUE.COUNT == 0
5 POP-FRONT(DEQUE)
6 return $x$
```
**Correctness:** MIN always returns the smallest element in the FIFO. We show the following invariant:

**Lemma 1** Deque holds all elements in FIFO smaller or equal to the last element in FIFO, in increasing order.

**Proof.** We use induction. Initially deque is empty so lemma holds. The invariant can only be changed on ENQUEUE or DEQUEUE.

Before ENQUEUE\(x\), deque is in increasing order (by inductive assumption). ENQUEUE\(x\) removes in the while loop all elements from deque smaller than \(x\) which now becomes the last element in FIFO. Thus the deque now holds all elements in FIFO smaller or equal \(x\). Finally, \(x\) is appended to end of deque or the count is increased, hence deque keeps correct count of all values in increasing order.

If DEQUEUE removes \(x\) from the FIFO and finds that it equals the value at the front of the deque, then the count is updated to reflect the accurate value. If the count becomes zero, the tuple is removed from the deque, thus the invariant holds. If \(x\) is not at the front of the deque, then that value smaller than \(x\) is in the FIFO and (by inductive assumption) \(x\) is not in the deque at all. Deque is not modified, hence invariant holds. 

Therefore, the front of deque is always the smallest element in the FIFO.

**Accounting:** The actual work done by MIN and DEQUEUE is constant so we focus on ENQUEUE. Assume that the FIFO enqueue/dequeue and deque append/pop operations cost 1, and so does each comparison. Then we set the price \(\hat{c}_E = 5\). Suppose we perform \(m\) enqueues, which puts \(mc_E\) into the bank. The actual work done is \(3m + 2k\), where \(3m\) comes from enqueue to FIFO, compare to end of deque, and append to deque, while \(2k\) comes from \(k\) compare-and-pop from the end of deque. Since \(k \leq m\), we have that the account balance \(5m - (3m + 2k) = 2(m - k) \geq 0\). Thus amortized cost of Enqueue is \(\Theta(1)\)

**Potential:** Define \(\Phi(D) = 2|\text{deque}|\). Observe \(\Phi(D_0) = 0\) and \(\Phi(D) \geq 0\). MIN does not modify the deque, so let us focus on ENQUEUE and DEQUEUE. Suppose that the number of elements removed from deque during ENQUEUE is \(k\). Then:

\[
\hat{c}_E = 3 + 2k + \Delta \Phi(D) = 3 + 2k + (2 - 2k) = 5
\]

Similarly, suppose that the number of elements removed from deque during DEQUEUE is \(k \in \{0, 1\}\). The amortized cost of DEQUEUE is:

\[
\hat{c}_D \leq 3 + k - 2k \leq 3
\]
Problem 2-2. Paging

Virtual memory is one of the important paradigms of a modern computer system. It is an abstraction that allows a process to have a larger address space than the primary memory (DRAM) on the machine. Before virtual memory was introduced, only those programs that could fit in primary memory could be run on a system. With virtual memory, programs larger than primary memory can be run using a strategy called paging.

Paging works as follows. The program’s virtual memory is divided into a set of equal-sized regions, called pages, which are typically 4096 bytes. Every page resides in a swapping file on the large but slow hard disk of the system. (Actually, as an optimization, pages of all zeroes are usually not stored.) When a location is accessed by the processor, the paging system checks to see whether the page containing the location resides in DRAM. If so, it translates the virtual address of the page to the physical location in DRAM, and the processor instruction that is accessing the location completes. If the location does not reside in DRAM, however, a page fault occurs. A page in DRAM is chosen for eviction. If any location on the page has been written, it must be copied out to the swapping file on the hard disk. Then, the page with the needed location is brought into DRAM, the virtual address is translated to the physical location, and the processor instruction that is accessing the location completes.

The paging problem can be formalized as follows. We have a slow memory (hard disk) with \( N \) distinct pages and a fast memory (DRAM) \( M \), sometimes referred to as a page cache, which can contain at most \( k \) pages, where \( k < N \). Thus, at any time \( t \), we have the set \( M_t \) of pages in \( M \) at time \( t \) is \( M_t \subseteq \{1, 2, \ldots, N\} \), where \( |M_t| \leq k \). Typically, we assume that \( M_0 = \emptyset \). At each time step \( t \), a page request \( \sigma_t \in \{1, 2, \ldots, N\} \) arrives. If we have \( \sigma_t \in M_{t-1} \), then nothing happens, and the page cache satisfies the request. If we have \( \sigma_t \not\in M_{t-1} \), however, then a page fault occurs. We choose one page \( p \in M_{t-1} \) to evict according to the page-replacement algorithm, and then \( M_t = M_{t-1} - \{p\} \cup \{\sigma_t\} \). The goal of the page-replacement algorithm is to minimize the total number of page faults.

Key in this process is the choice of which page to evict. Since there is only a limited space available in DRAM, paging systems must determine the page to evict by choosing one that is unlikely to be accessed in the near future. Many page-replacement algorithms have been devised. A popular one is LRU (least recently used), which evicts the page whose last access was most distant in the past.

In this problem, we shall consider a specific class of page-replacement algorithms called marking algorithms, which conceptually use one mark bit per page for bookkeeping. A marking algorithm partitions the sequence \( \Sigma \) of page requests into phases and operates as follows:

- At the beginning of each phase, all pages in the page cache are unmarked.
- Whenever a page is requested, it is marked.
- When a page must be evicted, the algorithm chooses an unmarked page to evict.
- When an eviction is necessary and all the pages are marked, all pages are unmarked, a new phase is started, and the algorithm chooses an unmarked page to evict.

During a phase, at each request, any marking algorithm must respond in one of the following ways:
A. If an unmarked page in the page cache is requested, it is marked.
B. If a marked page is requested, it remains marked.
C. If the requested page is not in the page cache, an unmarked page is evicted if necessary, and
the requested page is brought into the page cache and marked.

Events A and B incur no page faults, and Event C incurs 1 page fault.

(a) Prove that LRU is effectively a marking algorithm. (Hint: Argue that all the marked
pages precede all the unmarked pages when pages are sorted in order of their most
recent request.)

Solution: We can consider LRU to be a marking algorithm if we can show that its
execution is indistinguishable to any marking algorithm. In other words, if we can
show that the responses of LRU are chosen as if it were conceptually keeping track of
a mark bit for every page in the cache and serves page requests according to Events
A-C defined in the problem description. Let us suppose that LRU actually marks the
pages as they are requested and that when all pages are marked and a request causes a
page fault, LRU starts a new phase by unmarking the entire cache and then marking
the requested page. It follows from our setting that during a phase, a marked page is
never unmarked as we never unmark a page during a phase.

It remains to show that LRU evicts only unmarked pages. Suppose a page request \( \sigma \)
causes a page fault and forces LRU to evict the last recently used page, denoted as \( p \).
There are two cases

- \( \sigma \) triggers a new phase. LRU unmarks the entire cache to start the new phase, and
  so \( p \) is unmarked when evicted.
- \( \sigma \) doesn’t trigger a new phase. Suppose that \( p \) is marked when evicted. All other
  pages in the cache must be marked because they were more recently accessed
  than \( p \). Thus, all pages in the cache are marked, contradicting the assumption that
  \( \sigma \) doesn’t trigger a new phase. Thus, \( p \) is unmarked when evicted.

We shall now analyze the performance of LRU compared with the optimal offline page-replacement
algorithm OPT which has a priori knowledge of the sequence \( \Sigma \) of page requests. In particular,
we shall compare LRU(\( k \)), which executes the LRU algorithm on a page cache of \( k \) pages, with
OPT(\( h \)), which executes the optimal offline algorithm OPT on \( h \leq k \) pages.

(b) In any phase, argue that LRU(\( k \)) can evict a maximum of \( k \) pages.

Solution: Note that any page request during a phase either keeps the total count of
marked pages the same or increases the count by 1. In particular when a page request
evicts an unmarked page, then the count definitely increases by 1. Suppose \( k \) evictions
occur in the same phase, then the count after the \( k \)-th eviction is \( k \), i.e. all pages are
marked. Thus, the next request to cause an eviction cannot be in the same phase, or else it will evict a marked page and contradict that LRU\((k)\) is a marking algorithm.

(c) Prove that LRU\((k)\) never evicts a page twice in the same phase.

**PROOF.** If any page \(p\) has to be evicted twice in the same phase, it must have been that it was evicted once in the same phase. In other words, \(p\) was evicted and brought into its cache during same phase. Recall from part (a) that LRU is a marking algorithm. Thus, \(p\) would be marked since it was accessed in the ongoing phase. Thus, the page request that evicts \(p\) a second time is evicting a marked page (and no pages are unmarked during a phase), contradicting LRU\((k)\) is a marking algorithm.

(d) Show that OPT\((h)\) faults on at least \(k - h + 1\) pages during any phase of LRU\((k)\) except perhaps the last.

**PROOF.** A lower bound of \(k - h\) is straight forward. Every phase of LRU\((k)\) has at least \(k\) distinct pages and OPT\((h)\) has only \(h\) pages on its cache. Thus, OPT\((h)\) faults on at least \(k - h\) pages in any phase of LRU\((k)\).

On the other hand, this is a loose bound. When we were computing the bound of \(k - h\), we have implicitly assumed that in any phase OPT\((h)\) “magically” finds \(h\) of the \(k\) pages being requested on its cache. In a sense, we have given too much power to OPT\((h)\). The goal of this part is to see if we can penalize OPT a little more than \(k - h\).

Since we have made most of the page faults being incurred during a phase, there is very little that we can hope to do over there. The period of phase transition is something which we really have not looked into and is a possible candidate to squeeze in a page fault or two to improve the earlier lower bound. All that we are doing is to improve the earlier estimate of the page faults incurred by OPT\((h)\). Everything else including the definition of a phase, remains same.

Before we go ahead and do anything else, we have a subtle issue. Consider the following: a page fault is incurred during the dusk of an old phase and the dawn of the new phase (the sense will be clear in the subsequent paragraph). It cannot be the case that these page faults are not associated with any phase and are thus free! So it is imperative that we associate them in an unambiguous manner. From this point, we associate any ambiguous page faults with the old phase. Even if you choose the other way around, the spirit of the argument can be used to construct a new proof analogous to the one presented below.

Consider any two consecutive phases \(\Sigma, \Pi\) of the LRU\((k)\) algorithm. Let \(\sigma_1 \ldots \sigma_k\) be the \(k\) distinct pages requested during \(\Sigma\), let \(\sigma_{k+1}\) be the first page requested during \(\Pi\). Note that \(\sigma_{k+1} \notin \{\sigma_1 \ldots \sigma_k\}\) as \(\sigma_{k+1}\) embarks a new phase \(\Pi\).

Assume that OPT\((h)\) has \(\sigma_1\) in its cache. Now consider the set of requests \(\sigma_1 \ldots \sigma_k\) on OPT. After processing the request for page \(\sigma_1\), OPT\((h)\) already has \(\sigma_1\) in its
cache. Thus, \( \text{OPT}(h) \) has at most \( h - 1 \) pages in its cache other than \( \sigma_1 \). Therefore, on the sequence \( \sigma_2 \ldots \sigma_{k+1} \), \( \text{OPT}(h) \) will fault at least \( k - (h - 1) \) times (since the aforementioned sequence has \( k \) distinct pages). From our convention, the page fault incurred during a phase transition is awarded to the old phase, which is \( \Sigma \).

Since, we have not made any specific assumptions on \( \Sigma \), \( \Pi \); we can extend the arguments to any two consecutive phases \( \Psi \) and \( \Delta \) and establish that \( \text{OPT}(h) \) faults on at least \( k - h + 1 \) pages during \( \Psi \). In other words, the above argument can be extended to any phase which has a non-empty phase succeeding it. Thus, \( \text{OPT}(h) \) must fault \( k - h + 1 \) times in every phase (with the exception of the last phase) of LRU(\( k \)).

An alternate way of stating the result proved in part (d) is: for every \( k \) faults incurred by LRU(\( k \)), \( \text{OPT}(h) \) faults on at least \( k - h + 1 \) pages.

(e) Prove that over any phase of LRU(\( k \)) except perhaps the last, the competitive ratio of LRU(\( k \)) with respect to \( \text{OPT}(h) \) is \( k/(k - h + 1) \).

PROOF. We invoke the results of part (b) and part (c) to infer that LRU(\( k \)) incurs at most \( k \) page faults in every phase (except the last). Also, part (d) establishes that \( k - h + 1 \) is a lower bound on the number of faults that \( \text{OPT}(h) \) incurs in any phase of LRU(\( k \)). Thus, the competitive ratio of LRU(\( k \)) w.r.t. \( \text{OPT}(h) \) is most \( k/k - h + 1 \) per phase.

(f) Conclude that the over the entire sequence \( \Sigma \) of page requests, the competitive ratio of LRU(\( k \)) with respect to \( \text{OPT}(h) \) is \( k/(k - h + 1) \).

Solution: In part (d) we have established that over any phase (except the last one) of a sequence of page requests, if LRU(\( k \)) incurs \( k \) page faults then \( \text{OPT}(h) \) has to make at least \( k - h + 1 \) faults. From the definition of a phase, it follows that any two phases are disjoint. Also, in every phase (except the last) LRU(\( k \)) can make at most \( k \) page faults. Thus, the maximum number of faults incurred by LRU(\( k \)) is No. of Phases - 1 \( \times \) \( k \). We also know that over entire sequence, \( \text{OPT}(h) \) incurs at least No. of Phases - 1 \( \times \) \( (k - h + 1) \). So, the competitive ratio of LRU(\( k \)) w.r.t. \( \text{OPT}(h) \) is at most \( k/k - h + 1 \).

(g) Under what circumstances is LRU(\( k \)) \( O(1) \)-competitive with \( \text{OPT}(h) \)?

Solution: If \( k = 2h \), then we find that LRU(\( 2h \)) is \( 2 \)-competitive w.r.t \( \text{OPT}(h) \). In particular, if the LRU has a cache \( n \) times larger than OPT, we get a \( (n/n - 1) \)-competitive algorithm. This approach of comparing an online algorithm with a crippled OPT is referred to as resource augmentation.

(h) (Optional.) Prove that any deterministic page-replacement algorithm that operates with a page cache of \( k \) pages (not just LRU(\( k \))) cannot be \( (k - 1) \)-competitive with
OPT(k). (Hint: Consider a virtual memory with \( k + 1 \) distinct pages. Construct a sequence of requests where the online algorithm produces a page fault on every request. Use the fact that the algorithm is deterministic and OPT(k) can know \textit{a priori} which page will be evicted.)

**Proof.** We shall now construct a sequence \( \Sigma \) of page requests such that any deterministic algorithm \( A(k) \) faults on every request. Assume that there are \( k + 1 \) distinct pages in the slow memory and that the initial state of \( A(k) \)’s cache is \( \psi \). Since the cache of \( A(k) \) can hold at most \( k \) pages, there is at least one page \( x \) which is not in the cache of \( A(k) \). We now let the first page in \( \Sigma \) to be \( x \). Since \( A(k) \) is deterministic, we can simulate the reaction of \( A(k) \) to any page request given the state of the cache prior to the request. Thus, we can predict which page will be evicted from \( A(k) \)’s cache.

The second page in \( \Sigma \) will be the page evicted after the request for \( x \). Along similar lines, we can set the \( i^{th} \) page of \( \Sigma \) to be the page evicted after the \( i – 1 \) request. And from the construction, \( A(k) \) should fault on the \( i^{th} \) request as it is not in its cache after \( i – 1 \) request. Hence, we can construct a sequence \( \Sigma \) which results in a page fault on every page request.

Now consider any sub-sequence \( \sigma \) in \( \Sigma \) such that \( \sigma \) has \( k + 1 \) pages. Even though OPT(k) knows the sequence \( \sigma \), it has to fault at least once as it can hold only \( k \) distinct pages in its cache. However we know (from previous paragraph) that \( A(k) \) faults on all the pages of \( \Sigma \). And so during \( \sigma \), \( A(k) \) faults on at least \( k + 1 \) pages while OPT(k) faults only once.

To establish the claim, divide \( \Sigma \) into sub-sequences \( \sigma_0, \sigma_1, \sigma_2 \ldots \sigma_l \), each having \( k + 1 \) distinct pages. In any sub-sequence, \( A(k) \) faults at least \( k + 1 \) times while OPT(k) faults only once. Thus the competitive ratio in any sub-sequence is bounded by \( k + 1 \). Thus, the ratio of the faults incurred by \( A(k) \) to that of OPT(k) is bounded from below by \( k + 1 \). And, we have not any assumptions about \( A \) modulo the fact that it is deterministic. Thus, there is no deterministic algorithm with a cache of size \( k \) that is \( k \)-competitive to an OPT with the same amount of cache.

\[ \square \]