COS226 Group Activity

1. Sorting

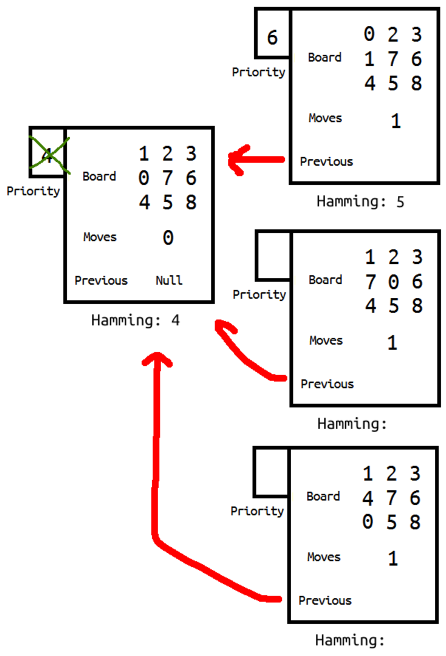
   In class, we've discussed 4 sorting algorithms: Selection, insertion, merge, and quicksort. For each sort, give an example of a situation where that sort would be best from among these four choices.

   Give an example of a computational problem that can be solved using sorting.

2. 8puzzle

   For puzzle07.txt, the PQ starts by containing only one node. This node is then deleted, and a new node is created for each of its board's three neighbors, linked to the parent node, and then added to the priority queue. The links between each node are shown below. The priority of the parent node is crossed out to indicate that the node is no longer in the PQ.

   Fill in the Hamming distances and priorities for the bottom two neighbors.



- Which node will be deleted from the priority queue next?
- Draw the search nodes after this node is removed from the PQ and its valid neighbors are linked up and added to the PQ.
- Why is one of the two neighbors invalid?
- Does this technique guarantee that no board ever appears in the priority queue twice?

3. LRU cache. (Spring 2012 midterm)

   An *LRU cache* is a data structures that stores up to N *distinct* keys. If the data structure is full when a key not already in the cache is added, the LRU cache first removes the key that was *least recently cached*.

   Design a data structure that supports the following API:

`public class LRU<Key>`

	`LRU(int N)`	*create an empty LRU cache with capacity N*
`void`	`cache(Key key)`	*if there are N keys in the cache and the given key is not already in the cache, (i) remove the key that was least recently used as an argument to* `cache()` *and (ii) add the given key to the LRU cache*
`boolean`	`inCache(Key key)`	*is the key in the LRU cache?*

   For example,

```
LRU<String> lru = new LRU<String>(5);
                                // LRU cache   (in order of when last cached)
lru.cache("A");                 // A           (add A to front)
lru.cache("B");                 // B A         (add B to front)
lru.cache("C");                 // C B A       (add C to front)
lru.cache("D");                 // D C B A     (add D to front)
lru.cache("E");                 // E D C B A   (add E to front)
lru.cache("F");                 // F E D C B   (remove A from back; add F to front)
boolean b1 = lru.inCache("C");  // F E D C B   (true)
boolean b2 = lru.inCache("A");  // F E D C B   (false)
lru.cache("D");                 // D F E C B   (move D to front)
lru.cache("C");                 // C D F E B   (move C to front)
lru.cache("G");                 // G C D F E   (remove B from back; add G to front)
lru.cache("H");                 // H G C D F   (remove E from back; add H to front)
boolean b3 = lru.inCache("D");  // H G C D F   (true)
```

   You should design your data structure using the new collections discussed in class (Set, Symbol Table, Priority Queue) as well as data structures we've discussed so far (arrays, linked list, heap). You may assume that Set, Symbol Table, and PQ operations are all fast. Try to craft a fast solution.