1. Intro. I learned recently from Marçal Garo a new data structure. It supports the operations **void** *push*(**int** *x*), **void** *pop*( ), and **int** *min*( ). The *push* and *pop* are queue operations, except *pop* returns nothing. The *min* operations is performed in constant time; a sequence of n pushes and n pops takes $O(n)$ time, so the amortized cost for all operations is constant.

2. Some boilerplate first.

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  ⟨ Global data 3 ⟩
  void push(int x) ⟨ Push body 6 ⟩
  void pop( ) ⟨ Pop body 5 ⟩
  int min( ) ⟨ Min body 4 ⟩
  int main( )
  {
    int v;    /∗ holds the value to insert in the queue ∗/
    char buf[1 ≪ 8];    /∗ hold the last line read from stdin ∗/
    while (fgets(buf, 1 ≪ 8, stdin)) {
      if (sscanf(buf, "PUSH␣%d", &v) ≡ 1) push(v);
      else if (¬memcmp(buf, "POP", 3)) pop( );
      else if (¬memcmp(buf, "MIN", 3)) printf("%d\n", min( ));
      else printf("I␣don't␣understand:␣%s\n", buf);
    }
  }
```

3. The idea is to keep a queue of possible answers to the *min*( ) query.

```
⟨ Global data 3 ⟩ ≡
  struct Node {
    int count;    /∗ this Node represents the count values of those pushed ∗/
    int min_value;    /∗ the minimum of the represented values ∗/
    Node ∗prev;    /∗ the older values that were pushed ∗/
    Node ∗next;    /∗ the newer values that were pushed ∗/
  };
  Node ∗oldest;    /∗ the oldest values in the queue ∗/
  Node ∗newest;    /∗ the newest values in the queue ∗/
```

This code is used in section 2.

4. We will maintain the invariant that $n\to min\_value < n\to next\to min\_value$ whenever $n\to next \neq \Lambda$. Therefore the overall minimum is always $oldest\to min\_value$ (if the queue is non-empty).

```
#define ∞ #7fffffff    /∗ almost infinity ∗/
⟨ Min body 4 ⟩ ≡
  {
    if (oldest) return oldest→min_value;
    return ∞;
  }
```

This code is used in section 2.

5. Popping is almost as normal popping.

```
⟨Pop body 5⟩ ≡
  {
    assert(oldest);
    if (oldest→count ≡ 1) {
      Node *todelete = oldest;
      oldest = oldest→next;
      if (oldest) oldest→prev = Λ;
      if (¬oldest) newest = Λ;
      free(todelete);
    }
    else −−oldest→count;
  }
```

This code is used in section 2.

6. To push x we 'compress' nodes if x is smaller than their corresponding minimum.

```
⟨Push body 6⟩ ≡
  {
    int count = 1;
    Node *p, *q;
    p = newest;
    while (p ∧ p→min_value ≥ x) {
      count += p→count;
      q = p;
      p = q→prev;
      free(q);
      if (p) p→next = Λ;
    }
    newest = p;
    p = (Node *) malloc(sizeof(Node));
    p→count = count;
    p→min_value = x;
    if (newest) newest→next = p;
    p→prev = newest;
    p→next = Λ;
    if (¬newest) oldest = p;
    newest = p;
  }
```

This code is used in section 2.

MINQUEUE