SE 504 Formal Methods and Models
Development of a Majority Vote Program

Recall that a *bag* B is a collection of elements of some type T, possibly containing duplicates. For each $x \in T$, the number of occurrences of x in B is referred to as the *multiplicity* of x (in B). A *set* is a bag in which an element's multiplicity cannot exceed one. The *cardinality* of B is denoted $|B|$ and is equal to the total number of occurrences of elements in it. The multiplicity of an element x in a bag B is sometimes written $|B|_x$.

For example, the bag

$$B = \{|\ g, @, g, p, A, g, A\ |\}^1$$

has cardinality seven, with the characters g, A, @, and p having multiplicities three, two, one, and one, respectively. (All other characters have multiplicity zero, of course!)

If x satisfies $|B|_x > \frac{|B|}{2}$ (i.e., the multiplicity of x exceeds half the cardinality of B), then x is said to be the *majority* of B. Obviously, a bag has at most one majority.

Consider the problem of determining, for a bag B given as input, whether or not B has a majority and, if it does, which element it is.

To concretize this, we must make some assumptions about the form in which B is given. If, for example, B is given as a list of pairs $\langle (e_1, m_1), (e_2, m_2), \cdots, (e_p, m_p) \rangle$ denoting that B contains exactly m_i occurrences of e_i, for each i $(1 \leq i \leq p)$, the problem reduces to that of scanning the list to (a) sum up the m_i's in order to calculate $|B|$ and (b) find the pair (e_j, m_j) having the maximum m component. Then e_j is the majority of B if $m_j > \frac{|B|}{2}$; otherwise, B has no majority.

To make things more interesting, we will suppose that B is given in the form of an array b, each element of which represents one occurrence of a value in B. For example, one possible representation of the bag B illustrated above is the array

```
    0     1     2     3     4     5     6
  +-----+-----+-----+-----+-----+-----+-----+
b | 'g' | '@' | 'g' | 'p' | 'A' | 'g' | 'A' |
  +-----+-----+-----+-----+-----+-----+-----+
```

Perhaps the most obvious solution would be to scan the array while maintaining a "lookup table" that keeps a count of how many occurrences of each value have been observed so far. Each entry in the table would be an ordered pair (v, m) satisfying the condition that value v occurs exactly m times among the array elements that have been examined so far. Each time an array element is examined, we look up its value in the table. If it is found, its corresponding counter is incremented. Otherwise, we insert a new entry into the table indicating that that

[1] We use curly braces augmented with vertical bars, {| and |}, when enumerating the elements of a bag. Unaugmented curly braces are used when enumerating the elements of a set.

value has been seen once. If the entire array is scanned and no counter has reached the threshold of being greater than half of the length of the array, there is no majority.

There are various ways of implementing the lookup table. One would be to use a hash table, in which case this approach would take time proportional to $N = \#b$ (i.e., $O(N)$ time), assuming that we had a good hash function. However, the table would (in the worst case) require $O(N)$ space, too.

Another approach would be to sort b so that all duplicate values are adjacent to one another. Let $m = \lfloor \frac{N}{2} \rfloor$. After sorting b, the only value that could possibly be a majority is $b.m$. (*Reason*: Because b has been sorted, any value not equal to $b.m$ occurs on only one side or the other of location m. But there are only m values "to the left" of location m and only $N - m - 1$ values "to the right" of location m. By our choice of m, neither m nor $N - m - 1$ can be greater than $\frac{N}{2}$ $(= \frac{|B|}{2})$.) To determine whether or not $b.m$ is a majority, we find the largest values for i and j satisfying $b.(m - i) = b.m = b.(m + j)$. As $b.m$ occurs $i + j + 1$ times in b, the answer is given by the boolean expression $i + j + 1 > \frac{N}{2}$.

The running time of this approach is dominated by the sorting step, which takes time $O(N \cdot \lg N)$.[2] Assuming that we are allowed to modify b (and that we choose a sorting algorithm that uses only a constant amount of extra space), this approach takes only a constant amount (i.e., $O(1)$) of extra space.

Among our two suggested solutions, one takes only $O(N)$ time (but also $O(N)$ extra space) while the other takes $O(N \cdot \lg N)$ time but only $O(1)$ extra space. Is it possible to find a solution that combines the best of the performance characteristics of the two suggested solutions, namely linear time and constant space?

The answer is yes! One of the keys to arriving at such a solution is to exploit this theorem:

Theorem 1: Let B be a bag and let B' be any bag obtained by removing from B one copy of each of two (distinct) elements. Then, if x is a majority of B, x is also a majority of B'.

Proof: Let $|B| = n$ and $|B|_x = k$. (Recall that $|B|_x$ denotes the number of occurrences of x in B.) As $|B'| = n - 2$ and either $|B'|_x = k - 1$ or $|B'|_x = k$ (depending upon whether or not a copy of x was removed from B in obtaining B'), it suffices to show $(k > \frac{n}{2}) \Rightarrow (k - 1 > \frac{n-2}{2})$, which is easy. *End of proof.*

Consider the procedure in Figure 1 that, given a bag B, "reduces" it to a bag in which at most one element occurs. (Note that the procedure is nondeterministic; as a consequence, the resulting bag is not unique.) Let *Red.B* (the "reduction of B") be the set containing any bag that can be obtained (in the end) by applying this procedure to B.

Theorem 2: Let B and C be bags such that $C \in Red.B$. If x is a majority of B, then x is also a majority of C.

Proof: Let x be a majority of B. Consider an arbitrary execution of the procedure in Figure 1. Let m be the number of loop iterations that occur before termination. Denote by W_i,

[2]This assumes that we have a total ordering defined upon the values in the array. Otherwise, sorting takes time proportional to the square of $\#b$.

```
W := B;
while W contains occurrences of two distinct elements {
    u := some element in W;
    v := some element in W distinct from u;
    W := W - {| u,v |};   // remove from W one occurrence of each of u and v
}
// Assertion: W ∈ Red.B
```

Figure 1: Bag Reduction Procedure

$0 \leq i \leq m$, the value of the program variable `W` immediately after i iterations have been completed. In particular, $W_0 = B$ and $W_m = C$. We show by mathematical induction on i that x is also a majority of W_i, for all i. As $C = W_m$, this suffices.

Basis: $i = 0$. As $W_0 = B$, obviously x is a majority of W_0.
Inductive Step: Let k be arbitrary, $0 \leq k < m$. As an inductive hypothesis, assume that x is a majority of W_k. W_{k+1} is obtained by removing one occurrence of each of two elements from W_k. Hence, by Theorem 1, x also must be a majority of W_{k+1}. *End of proof.*

Obviously, if $C \in Red.B$, either C is empty or it contains some (positive) number of occurrences of some y and no occurrences of any other value. Thus, what Theorem 2 tells us is that, if we apply the above-described Reduction Procedure to B, thereby obtaining C, then the value in C (if any) is the *only possible* candidate for being a majority of B. If C is empty, B has no majority; otherwise we compute $|B|_y$ (where y is the value that appears in C) and $|B|$ and then compare them to determine whether or not y is the majority in B.

Here we concentrate on the computation of C, as that is the interesting step. Suppose that we examine the members of B in some arbitrary order. For n satisfying $0 \leq n < |B|$, let x_n be the n-th member examined and let B_n be the bag comprising the first n members examined (i.e., $B_n = \{|\, x_0, x_1, \ldots, x_{n-1} \,|\}$). Consider the following program skeleton:

```
|[ C, n := ∅, 0;
  { loop invariant P : C ∈ Red.B_n ∧ 0 ≤ n ≤ |B| }
  { bound function t : |B| − n}
  do n ≠ |B| ⟶
      C := ?;
      {P(n := n + 1)}
      n := n + 1;
  od
  {C ∈ Red.B_n ∧ n = |B|}
  {C ∈ Red.B}
]|
```

The crucial missing part is the right-hand side of the assignment to C. The purpose of the assignment is to transform C from a member of $Red.B_n$ into a member of $Red.B_{n+1}$. As B_n and B_{n+1} are closely related (precisely, $B_{n+1} = B_n \cup \{|\, x_n \,|\}$), it seems reasonable to suspect

that a relatively minor tweak of C will suffice. To attempt to figure out exactly what that tweak might be, we explore the following question:

Suppose that F and G are bags such that $F \in \mathit{Red}.G$, and let $G' = G \cup \{|\ x\ |\}$. Is there a bag F' similar to F satisfying $F' \in \mathit{Red}.G'$?

Consider applying the Bag Reduction Procedure to G' and just suppose that, in doing so, we "set aside" one occurrence of x and agree not to involve it in the computation. (In effect, then, we are reducing G and bringing an occurrence of x along for the ride.) Eventually (after some number of iterations) we will be left with a bag $H = F \cup \{|\ x\ |\}$, where $F \in \mathit{Red}.G$. At this point, no further reduction is possible without considering the set-aside occurrence of x, so we allow it "back into the game".

There are three possibilities:
(1) F is empty. Then $H = \{|\ x\ |\}$ fails to contain two distinct elements, so no further loop iterations are called for and we have $\{|\ x\ |\} \in \mathit{Red}.G'$.

(2) $F = \langle y, k\rangle$ with $k > 0$ and $y \neq x$.[3] Then $H = \langle y, k\rangle \cup \langle x, 1\rangle$, so that one more iteration (during which one occurrence of each of x and y are removed) yields $\langle y, k-1\rangle$, after which (there no longer being two distinct elements) the loop terminates. We get that $\langle y, k-1\rangle \in \mathit{Red}.G'$.

(3) $F = \langle y, k\rangle$ with $k > 0$ and $y = x$. Then $H = \langle x, k\rangle \cup \langle x, 1\rangle$, i.e., $H = \langle x, k+1\rangle$, so no further loop iterations are called for and we have $\langle x, k+1\rangle \in \mathit{Red}.G'$.

The discussion above is, in effect, a proof of

Theorem 3: Let F and G be bags satisfying $F \in \mathit{Red}.G$, and let $G' = G \cup \{|\ x\ |\}$.
(1) If $F = \emptyset$, then $\{|\ x\ |\} \in \mathit{Red}.G'$ (i.e., $\langle x, 1\rangle \in \mathit{Red}.G'$).
(2) If $F = \langle y, k\rangle$ (with $k > 0$) and $x \neq y$, then $\langle y, k-1\rangle \in \mathit{Red}.G'$.
(3) If $F = \langle y, k\rangle$ (with $k > 0$) and $x = y$, then $\langle y, k+1\rangle \in \mathit{Red}.G'$.

We now have the insight by which to complete the program. We replace the (abstract) variable C by variables y and k with the understanding that $C = \langle y, k\rangle$. Also, B is replaced by the array b with the understanding that $B_n = b[0..n)$.

---

[3] We use $\langle y, k\rangle$ to denote the bag containing k occurrences of y and no occurrences of any element distinct from y.

$|[$ con b : array of T; // $b[0..i)$ represents B_i for all i
  var $y : T$;
  var k : int; // $C = \langle y, k \rangle$
  var n : int;
  $k, n := 0, 0$;
  { (abstract) loop invariant $P : C \in Red.B_n \wedge 0 \leq n \leq |B|$}
  { (concrete) loop invariant $P : \langle y, k \rangle \in Red.(b[0..n)) \wedge 0 \leq n \leq \#b$}
  { (abstract) bound function $t : |B| - n$}
  { (concrete) bound function $t : \#b - n$}
  do $n \neq \#b \longrightarrow$
    if $k = 0 \longrightarrow y, k := b.n, 1;$
    [] $k \neq 0 \wedge y \neq b.n \longrightarrow k := k - 1;$
    [] $k \neq 0 \wedge y = b.n \longrightarrow k := k + 1;$
    fi;
    $\{P(n := n + 1)\}$
    $n := n + 1;$
  od
  { (concrete) postcondition: $\langle y, k \rangle \in Red.b[0..n) \wedge n = \#b$}
  { (abstract) postcondition: $C \in Red.B_n \wedge n = |B|$}
  { (abstract) postcondition: $C \in Red.B$}
$]|$