In chapter 6 we considered several representations for sets whose elements are integers of arbitrary size. However, all those representations are rather inefficient: at least one of the operations (membership test, adding/deleting an element) runs in time proportional to the number of elements in the set. This is unacceptable in almost all practical applications.

It is possible to find a set representation where all three operations mentioned run in time $C \log n$ (in the worst case) for sets with $n$ elements. One such representation is considered in the next chapter. In this chapter, we consider another set representation that may require $n$ operations in the worst case but is very efficient in a “typical” case. The method is called hashing.

We consider two versions of this technique. Open addressing (section 13.1) is somehow simpler (and more efficient in terms of space), especially if we do not need deletion. Then we consider (section 13.2) hashing with lists; this version of hashing is more flexible and easier to analyze.

### 13.1 Hashing with open addressing

Suppose we want to store a set of elements of type $T$, where the number of elements is guaranteed to be less than $n$. Choose a function $h$ that is defined on elements of type $T$ and whose values are integers in the range $0..n-1$. It is desirable that this function has different values for different elements of the set that we are trying to represent (the worst case is when all the function values are the same). This function is called a hash function.

Our representation uses two arrays

```pascal
val: array [0..n-1] of T;
used: array [0..n-1] of Boolean;
```

(we write $n-1$ in the type definition though it is not permitted in Pascal). The set consists of $val[i]$ for all $i$ such that $used[i]$ is true. (The values $val[i]$ are all
different.) When possible, we store an element $t$ at position $h(t)$, which is considered a “natural place” for $t$. However, it may happen that a new element $t$ appears whose place $h(t)$ is already used by another element (that is, $\text{used}[h(t)]$ is true). In this case, we search to the right looking for the first unused place and put the element $t$ there. (Here “to the right” means that the index increases; when we reach $n-1$, the index wraps around.) Recall that we assume that the number of elements is always less than the number of places, therefore free places do exist.

Formally speaking, the invariant relation that we maintain is the following: For any element, the interval between its natural place and its actual place is filled completely.

This invariant makes the membership test easy. Suppose we want to check if an element $t$ is in the set. We find the natural place for $t$ and then go to the right until we find an empty slot or $t$. In the first case, the element $t$ is not in the set (a consequence of our invariant); in the second case, the element is in the set. If it is absent, we may add it (filling the unused place found). If not, we can delete it by putting $\text{False}$ in the corresponding cell of the $\text{used}$ array.

13.1.1. The last passage has a severe error. Find it and correct it.

Solution. The delete operation implemented as described can destroy the invariant and create an empty position between the natural and actual positions of some element. We should be more careful. After a gap appears, we move from left to right until we find another gap or an element that is not at its natural place. If the gap appears first, we have nothing to worry about. If an element is found not at its natural place, we check whether it needs to be moved to the gap that we’ve created. If not, we continue our search. If yes, we move the element found to the gap. A new gap appears which we deal with in the same way. □

13.1.2. Write the programs for membership test, adding and deleting elements.

Solution.

function is_element (t: T): Boolean;
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| is_element := used [i] and (val [i] = t);
end;

procedure add (t: T);
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end;
