```
   Key::comparisons = 0;
   clock.reset();
   for (i = 0; i < searches; i++) {
      target = 2 * number.random_integer(0, list_size);
      if (method == 'r') {
       if (sequential_search(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
      }
      else if (method == 's')
       if (sentinel_search(the_list, target, found_at) == success)
         cout << "Error: Found unexpected target " << target
              << " at " << found_at << endl;
   }
   print_out("Unsuccessful", clock.elapsed_time(),
                             Key::comparisons, searches);
}
```

The Key class and other programs are as in Project P1.

P3. *What changes are required to our sequential search function and testing program in order to operate on simply linked lists as developed in Section 6.2.3? Make these changes and apply the testing program from Project P1 for linked lists to test linked sequential search.*

linked sequential search

Answer The solution is included with that of Project P1.

7.3 BINARY SEARCH

Exercises 7.3

E1. *Suppose that* the_list *contains the integers 1, 2, . . . , 8. Trace through the steps of* binary_search_1 *to determine what comparisons of keys are done in searching for each of the following targets:* **(a)** *3,* **(b)** *5,* **(c)** *1,* **(d)** *9,* **(e)** *4.5.*

Answer

(a) Position three of the list is examined first; the key in that position is greater than the target; and the top of the sublist reduces to three. Position one is then examined and the bottom of the sublist is increased to two because the target is larger than the key in position one. Position two is examined and, since the key is not greater than the target, the top is reduced to two. The loop stops as top is no longer larger than bottom. The comparison outside the loop verifies that the target has been found. A total of four key comparisons were done.

(b) In searching for the key 5, the positions three, five, and four of the list are compared to the target (four key comparisons).

(c) In searching for the key 1, the positions three, one, and zero of the list are compared to the target (four key comparisons).

(d) In searching, unsuccessfully, for the key 9, the positions three, five, six, and (outside the loop) seven of the list are compared to the target (four key comparisons).

(e) In searching, unsuccessfully, for the key 4.5, the positions three, five, four, and (outside the loop) four again of the list are compared to the target (four key comparisons).

E2. *Repeat Exercise E1 using* binary_search_2.

Answer

(a) In searching for the key 3, the key in position three is compared to the target, found to be not equal, compared again, found to be less than the target, and the top of the sublist is reduced to two. The key in position one of the list is compared to the target, found to be not equal, compared again, found to be not less than the target and the bottom of the sublist is increased to two. The key position two is compared to the target, found to be equal, and the comparisons terminate after five comparisons of keys.

(b) In searching for the key 5, the keys in positions three, five, and four are compared before equality is found (five key comparisons).

(c) In searching for the key 1, the keys in positions three, one, and zero are compared before equality is found (five key comparisons).

(d) In searching for the key 9, the keys in positions three, five, six, and seven are compared before bottom exceeds top and the comparisons terminate (eight key comparisons).

(e) In searching for the key 4.5, the keys in positions three, five, and four are compared before bottom exceeds top and the comparisons terminate (six key comparisons).

E3. *[Challenging] Suppose that L_1 and L_2 are ordered lists containing n_1 and n_2 integers, respectively.*

(a) *Use the idea of binary search to describe how to find the median of the $n_1 + n_2$ integers in the combined lists.*

Answer Without loss of generality we assume that $n_1 \le n_2$, for, if not, we can interchange the two lists before starting. Let x_1 and x_2 be the medians of L_1 and L_2, respectively, so that x_1 appears in position $\lceil n_1/2 \rceil - 1$ and x_2 in position $\lceil n_2/2 \rceil - 1$ of their respective lists. Suppose, first, that $x_1 \le x_2$. The median of the combined list must be somewhere between x_1 and x_2, inclusive, since no more than half the elements are less than the smaller of the two medians and no more than half are larger than the larger median. Hence no element strictly to the left of x_1 in L_1 could be the median, nor could any element to the right of x_2 in L_2. Since $n_1 \le n_2$, we can delete all elements strictly to the left of x_1 from L_1 and an equal number from the right end of L_2 and, in doing so, we will not have changed the median.

When $x_1 > x_2$ similar conditions hold, and we can delete the elements strictly to the right of x_1 from L_1 and an equal number from the left end of L_2.

In the same way as binary search, we continue this process, at each step removing almost half the elements from L_1 and an equal number from L_2. When L_1 has length 2, however, this process may remove no further elements, since the (left) median of L_1 is its first element. It turns out, however, that when L_1 has even length and $x_1 \le x_2$, then x_1 itself may be deleted without changing the median. Hence we modify the method so that, instead of deleting the elements strictly on one side of x_1, we delete $\lfloor n_1/2 \rfloor$ elements from L_1 and from L_2 at each stage. This process terminates when the length of L_1 is reduced to 1.

It is, finally, easy to find the median of a list with one extra element adjoined. Let a be the unique element in L_1, let b be the element of L_2 in position $\lfloor n_2/2 \rfloor$ (so b is the left median if n_2 is even and is one position left of the median if n_2 is odd), and let c be the element of L_2 in position $\lfloor n_2/2 \rfloor + 1$. It is easy to check that if $a \le b$ then b is the median of the combined list, else if $a \le c$ then a is the median, else c is the median.

(b) *Write a function that implements your method.*

Answer

```
Error_code find_median(const Ordered_list &l1, const Ordered_list &l2,
                       Record &median)
/* Pre:  The ordered lists l1,l2 are not empty and l2 contains at least as many entries as l1.
   Post: The median of the two lists combined in order is returned as the output parameter
         median. */
{
  int reduce;                                  //  removed from both ends at each iteration
  int mid1, bottom1, top1;
  int mid2, bottom2, top2;
  Record a, b, c;                              //  used to check values in short lists
  if (l1.size() <= 0 || l1.size() > l2.size())
      return fail;
  bottom1 = bottom2 = 0;
  top1 = l1.size() - 1;
  top2 = l2.size() - 1;
  /* Invariant: median is either between bottom1 and top1 in l1 or between bottom2 and top2
     in l2, inclusive; median is the median of the combined lists from bottom1 to top1 and
     bottom2 to top2. */
```

Sorting

8

8.2 INSERTION SORT

Exercises 8.2

E1. *By hand, trace through the steps insertion sort will use on each of the following lists. In each case, count the number of comparisons that will be made and the number of times an entry will be moved.*

(a) *The following three words to be sorted alphabetically:*

triangle square pentagon

Answer

triangle	square	pentagon
square	triangle	square
pentagon	pentagon	triangle

There are 5 moves in total and 3 comparisons.

(b) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in increasing order*

Answer In this case, there would be no moves made and 2 comparisons.

(c) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in decreasing order*

Answer

triangle (3)	square (4)	pentagon (5)
square (4)	triangle (3)	square (4)
pentagon (5)	pentagon (5)	triangle (3)

There are 5 moves in total and 3 comparisons.

(d) *The following seven numbers to be sorted into increasing order:*

26 33 35 29 19 12 22

Answer

26	26	26	26	19	12	12
[33]	33	33	29	26	19	19
35	[35]	35	33	29	26	22
29	29	[29]	35	33	29	26
19	19	19	[19]	35	33	29
12	12	12	12	[12]	35	33
22	22	22	22	22	[22]	35

In the above table, the entries in each column shown in [boxes] are the ones currently being processed by insertion sort. There are a total of 19 entries moved and 19 comparisons made.

(e) *The same seven numbers in a different initial order, again to be sorted into increasing order:*

12 19 33 26 29 35 22

Answer

12	12	12	12	12	12	12
[19]	19	19	19	19	19	19
33	[33]	33	26	26	26	22
26	26	[26]	33	29	29	26
29	29	29	[29]	33	33	29
35	35	35	35	[35]	35	33
22	22	22	22	22	[22]	35

There are a total of 9 moves and 12 comparisons in this case.

(f) *The following list of 14 names to be sorted into alphabetical order:*

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

Answer

Tim	Dot	Dot	Dot	Dot	Dot	Dot	Amy	Amy	Amy	Amy	Amy	Amy	Amy
[Dot]	Tim	Eva	Eva	Eva	Eva	Eva	Dot	Dot	Ann	Ann	Ann	Ann	Ann
Eva	[Eva]	Tim	Roy	Roy	Kim	Guy	Eva	Eva	Dot	Dot	Dot	Dot	Dot
Roy	Roy	[Roy]	Tim	Tim	Roy	Kim	Guy	Guy	Eva	Eva	Eva	Eva	Eva
Tom	Tom	Tom	[Tom]	Tom	Tim	Roy	Kim	Jon	Guy	Guy	Guy	Guy	Guy
Kim	Kim	Kim	Kim	[Kim]	Tom	Tim	Roy	Kim	Jon	Jim	Jim	Jim	Jan
Guy	Guy	Guy	Guy	Guy	[Guy]	Tom	Tim	Roy	Kim	Jon	Jon	Jon	Jim
Amy	Amy	Amy	Amy	Amy	Amy	[Amy]	Tom	Tim	Roy	Kim	Kay	Kay	Jon
Jon	Jon	Jon	Jon	Jon	Jon	Jon	[Jon]	Tom	Tim	Roy	Kim	Kim	Kay
Ann	Ann	Ann	Ann	Ann	Ann	Ann	Ann	[Ann]	Tom	Tim	Roy	Ron	Kim
Jim	Jim	Jim	Jim	Jim	Jim	Jim	Jim	Jim	[Jim]	Tom	Tim	Roy	Ron
Kay	Kay	Kay	Kay	Kay	Kay	Kay	Kay	Kay	Kay	[Kay]	Tom	Tim	Roy
Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	Ron	[Ron]	Tom	Tim
Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	Jan	[Jan]	Tom

TOC
Index
Help

```
Uses: Methods for the class Record; the contiguous
      List implementation of Chapter 6.
*/
{
   int i = 1;
   if (size() >= 2) do {
      if (entry[i] >= entry[i - 1]) i++;
      else {
         swap(i, i - 1);
         if (i > 0) i--;
      }
   } while (i != size());
}
```

(b) *Compare the timings for your program with those of* insertion_sort.

Answer Refer to the comparison tables in Project P6 of Section 8.10.

P4. *A well-known algorithm called* ***bubble sort*** *proceeds by scanning the list from left to right, and whenever a pair of adjacent keys is found to be out of order, then those entries are swapped. In this first pass, the largest key in the list will have "bubbled" to the end, but the earlier keys may still be out of order. Thus the pass scanning for pairs out of order is put in a loop that first makes the scanning pass go all the way to* count, *and at each iteration stops it one position sooner.* **(a)** *Write a C++ function for bubble sort.* **(b)** *Find the performance of bubble sort on various kinds of lists, and compare the results with those for insertion sort.*

bubble sort

Answer

```
template <class Record>
void Sortable_list<Record>::bubble_sort()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
Uses: Methods for the class Record; the contiguous
      List implementation of Chapter 6.
*/
{
   for (int top = size() - 2; top >= 0; top--)
      for (int i = 0; i <= top; i++)
         if (entry[i] > entry[i + 1]) swap(i, i + 1);
}
```

The function does $1 + 2 + \cdots + (n-1) = \frac{1}{2}(n-1)n$ comparisons of keys and about $\frac{1}{4}n^2$ swaps of entries (See Knuth, vol. 3, pp. 108–110). Refer to the comparison table in Project P6 of Section 8.10 for the figures on test runs of this function.

8.3 SELECTION SORT

Exercises 8.3

E1. *By hand, trace through the steps selection sort will use on each of the following lists. In each case, count the number of comparisons that will be made and the number of times an entry will be moved.*

(a) *The following three words to be sorted alphabetically:*

triangle square pentagon

Answer

triangle	pentagon	pentagon
square	square	square
pentagon	triangle	triangle

There will be three comparisons and one swap.

(b) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in increasing order*

Answer All entries are in their proper positions so no movement will occur. There will be 3 comparisons.

(c) *The three words in part (a) to be sorted according to the number of sides of the corresponding polygon, in decreasing order*

Answer The results will be the same as those in Part (a) of this question.

(d) *The following seven numbers to be sorted into increasing order:*

26 33 35 29 19 12 22

Answer

26	26	26	**26**	19	**19**	12
33	**33**	12	12	12	**12**	19
35	22	22	22	**22**	22	22
29	29	**29**	**19**	26	26	26
19	19	**19**	29	29	29	29
12	**12**	33	33	33	33	33
22	35	35	35	35	35	35

In this diagram, the two entries in each column shown in **boxes** are the ones that will be swapped during that pass of the function. Only one boxed entry in a column means that the entry is currently in the proper position. There will be 21 comparisons and 5 swaps done here.

(e) *The same seven numbers in a different initial order, again to be sorted into increasing order:*

12 19 33 26 29 35 22

Answer

12	12	12	12	12	12	12
19	19	19	19	19	**19**	19
33	**33**	22	22	**22**	22	22
26	26	26	**26**	26	26	26
29	29	**29**	29	29	29	29
35	**22**	33	33	33	33	
22	35	35	35	35	35	35

(f) *The following list of 14 names to be sorted into alphabetical order:*

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

```
template <class Record>
int Sortable_list<Record>::max_key(int low, int high)
/*
Pre:  low and high are valid positions in the Sortable_list and
      low <= high.
Post: The position of the entry between low and high with the largest
      key is returned.
Uses: The class Record.
      The linked List implementation of Chapter 6.
*/

{
   int largest, current;
   largest = low;
   Record big, consider;
   retrieve(low, big);
   for (current = low + 1; current <= high; current++) {
      retrieve(current, consider);
      if (big < consider) {
         largest = current;
         big = consider;
      }
   }
   return largest;
}

template <class Record>
void Sortable_list<Record>::selection_sort()
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
Uses: max_key, swap.
*/

{
   for (int position = count - 1; position > 0; position--) {
      int max = max_key(0, position);
      swap(max, position);
   }
}
```

8.4 SHELL SORT

Exercises 8.4

E1. *By hand, sort the list of 14 names in the "unsorted" column of Figure 8.7 using Shell sort with increments of* **(a)** *8, 4, 2, 1 and* **(b)** *7, 3, 1. Count the number of comparisons and moves that are made in each case.*

Answer

(a)

Incr.	*8*	*4*	*2*	*1*
Tim	Jon	Jon	Eva	Amy
Dot	Ann	Ann	Amy	Ann
Eva	Eva	Eva	Guy	Dot
Roy	Kay	Amy	Ann	Eva
Tom	Ron	Ron	Jim	Guy
Kim	Jan	Dot	Dot	Jan
Guy	Guy	Guy	Jon	Jim
Amy	Amy	Kay	Jan	Jon
Jon	Tim	Tim	Ron	Kay
Ann	Dot	Jan	Kay	Kim
Jim	Jim	Jim	Tim	Ron
Kay	Roy	Roy	Kim	Roy
Ron	Tom	Tom	Tom	Tim
Jan	Kim	Kim	Roy	Tom

(b)

Incr.	*7*	*3*	*1*
Tim	Amy	Amy	Amy
Dot	Dot	Dot	Ann
Eva	Ann	Ann	Dot
Roy	Jim	Eva	Eva
Tom	Kay	Jan	Guy
Kin	Kim	Jon	Jan
Guy	Guy	Guy	Jim
Amy	Tim	Kay	Jon
Jon	Jon	Kim	Kay
Ann	Eva	Jim	Kim
Jim	Roy	Roy	Ron
Kay	Tom	Tom	Roy
Ron	Ron	Ron	Tim
Jan	Jan	Tim	Tom

For the increments given in part (a), there will be 61 comparisons and 69 assignments made. With the increments in part (b), there will be 48 comparisons and 49 assignments.

E2. *Explain why Shell sort is ill suited for use with linked lists.*

Answer Shell sort requires random access to the list.

Programming Projects 8.4

P1. *Rewrite the method* insertion_sort *to serve as the function* sort_interval *embedded in* shell_sort.

Answer

```
template <class Record>
void Sortable_list<Record>::sort_interval(int start, int increment)
/*
Pre:  start is a valid list position and increment is a valid
      stepping length.
Post: The entries of the Sortable_list have been rearranged so that the
      keys in all the entries are sorted into nondecreasing order based
      on certain positions calculated by start and increment.
Uses: Methods for classes Record and Key.
      The contiguous List implementation of ?list_ch?.
*/

{
   int first_unsorted;     //  position of first unsorted entry
   int place;              //  searches sorted part of list
   Record current;         //  used to swap entries
```

```
#include "../linklist/insert.cpp"
template <class Record>
void Sortable_list<Record>::distribution(int max)
/*
Post: The entries of the Sortable_list
      have been rearranged so that the keys in all the
      entries are sorted into nondecreasing order.
*/

{
    Record data;

    Sortable_list<Record> *chains = new Sortable_list<Record>[max];
    int i;
    for (i = 0; i < size(); i++) {  // form the chains
       retrieve(i, data);
       int estimate = (int) (((Key) data).the_key() * max);
       while (estimate >= max) estimate--;
       chains[estimate].insert(0, data);
     }

    for (i = 0; i < max; i++)       // sort the chains
       chains[i].insertion_sort();

    int k = 0;                      // reform the original list (sorted)
    for (i = 0; i < max; i++)
       for (int j = 0; j < chains[i].size(); j++) {
          chains[i].retrieve(j, data);
          replace(k++, data);
       }
    delete []chains;
}
```

TOC Index Help

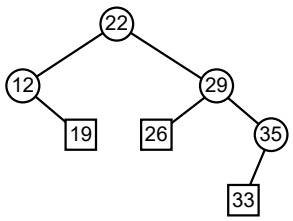
8.6 DIVIDE-AND-CONQUER SORTING

Exercises 8.6

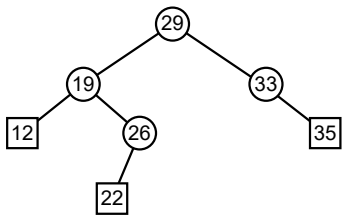
E1. *Apply quicksort to the list of seven numbers considered in this section, where the pivot in each sublist is chosen to be* **(a)** *the last number in the sublist and* **(b)** *the center (or left-center) number in the sublist. In each case, draw the tree of recursive calls.*

Answer

(a)



(b)



E2. *Apply mergesort to the list of 14 names considered for previous sorting methods:*

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

Answer

Tim Dot | Eva Roy Tom Kim | Guy Amy Jon | Ann Jim Kay Ron | Jan
Dot Tim | Eva Roy Kim Tom | Guy Amy Jon | Ann Jim Kay Ron | Jan
Dot Eva Roy Tim Guy Kim Tom Amy Ann Jim Jon Jan Kay Ron
Dot Eva Guy Kim Roy Tim Tom Amy Ann Jan Jim Jon Kay Ron
Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

E3. *Apply quicksort to this list of 14 names, and thereby sort them by hand into alphabetical order. Take the pivot to be* **(a)** *the first key in each sublist and* **(b)** *the center (or left-center) key in each sublist.*

Answer

(a)

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan
Dot Eva Roy Kim Guy Amy Jon Ann Jim Kay Ron Jan | Tim | Tom
Amy Ann | Dot | Eva Roy Kim Guy Jon Jim Kay Ron Jan Tim Tom
Amy | Ann | Dot Eva | Roy Kim Guy Jon Jim Kay Ron Jan Tim Tom
Amy Ann Dot Eva | Kim Guy Jon Jim Kay Ron Jan Roy Tim Tom
Amy Ann Dot Eva | Guy Jon Jim Kay Jan | Kim | Ron Roy Tim Tom
Amy Ann Dot Eva Guy | Jon Jim Kay Jan Kim Ron Roy Tim Tom
Amy Ann Dot Eva Guy | Jim Jan | Jon | Kay Kim Ron Roy Tim Tom
Amy Ann Dot Eva Guy | Jan Jim Jon Kay Kim Ron Roy Tim Tom
Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

(b)

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan
Dot Eva Amy Ann | Guy | Tim Roy Tom Kim Jon Jim Kay Ron Jan
Dot Amy Ann | Eva Guy | Jim Jan | Jon | Tim Roy Tom Kim Kay Ron
Amy | Dot Ann | Eva Guy | Jan | Jim Jon | Tim Roy Kim Kay Ron | Tom
Amy | Ann | Dot Eva Guy Jan Jim Jon | Kay | Kim | Tim Roy Ron | Tom
Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim | Ron | Roy | Tim | Tom
Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

E4. *In both divide-and-conquer methods, we have attempted to divide the list into two sublists of approximately equal size, but the basic outline of sorting by divide-and-conquer remains valid without equal-sized halves. Consider dividing the list so that one sublist has size 1. This leads to two methods, depending on whether the work is done in splitting one element from the list or in combining the sublists.*

(a) *Split the list by finding the entry with the largest key and making it the sublist of size 1. After sorting the remaining entries, the sublists are combined easily by placing the entry with largest key last.*

(b) *Split off the last entry from the list. After sorting the remaining entries, merge this entry into the list.*

Show that one of these methods is exactly the same method as insertion sort and the other is the same as selection sort.

Answer The first method is that used by selection sort. It works by finding the largest key and placing it at the end of the list. The second method is that used by insertion sort, which at each stage takes the next unsorted item and inserts it into the proper place relative to the previously sorted items.

8.7 MERGESORT FOR LINKED LISTS

Exercises 8.7

E1. *An article in a professional journal stated, "This recursive process [mergesort] takes time $O(n \log n)$, and so runs 64 times faster than the previous method [insertion sort] when sorting 256 numbers." Criticize this statement.*

```
    cout << "Program to demonstrate radix sort" << endl;
    cout << "Unsorted list \n";
    the_list.traverse(write_entry);
    cout << "\n";
    the_list.radix_sort();
    cout << "\n Sorted list \n";
    the_list.traverse(write_entry);
    cout << "\n";
}
```

Test results using this version of radix sort will be implementation dependent.

9.6 HASHING

Exercises 9.6

E1. *Prove by mathematical induction that* $1 + 3 + 5 + \cdots + (2i - 1) = i^2$ *for all integers* $i > 0$.

Answer For $i = 1$ both sides are equal to 1, so the base case holds. Assume the result is true for case $i - 1$:

$$1 + 3 + 5 + \cdots + (2i - 3) = (i - 1)^2.$$

Then we have

$$\begin{aligned} 1 + 3 + 5 + \cdots + (2i - 3) + (2i - 1) &= (i - 1)^2 + (2i - 1) \\ &= i^2 - 2i + 1 + (2i - 1) \\ &= i^2, \end{aligned}$$

which was to be proved.

E2. *Write a C++ function to insert an entry into a hash table with open addressing using linear probing.*

Answer

```
Error_code Hash_table :: insert(const Record &new_entry)
/* Post: If the Hash_table is full, a code of overflow is returned. If the table already contains an
         item with the key of new_entry a code of duplicate_error is returned. Otherwise: The
         Record new_entry is inserted into the Hash_table and success is returned.
   Uses: Methods for classes Key, and Record. The function hash. */
{
  Error_code result = success;
  int probe_count,                                  //  Counter to be sure that table is not full.
      probe;                                        //  Position currently probed in the hash table.
  Key null;                                         //  Null key for comparison purposes.
  null.make_blank();
  probe = hash(new_entry);
  probe_count = 0;
  while (table[probe] != null                       //  Is the location empty?
         && table[probe] != new_entry               //  Duplicate key?
         && probe_count < hash_size) {              //  Has overflow occurred?
    probe_count++;
    probe = (probe + 1) % hash_size;
  }
  if (table[probe] == null) table[probe] = new_entry;   //  Insert new entry.
  else if (table[probe] == new_entry) result = duplicate_error;
  else result = overflow;                           //  The table is full.
  return result;
}
```