CH. 12

2. An array contains the elements shown below. Show the contents of the array after it has gone through a one-increment pass of the shell sort. The increment factor is $k = 3$.

```
23 3 7 13 89 7 66 2 6 44 18 90 98 57
```

4. An array contains the elements shown below. What would be the value of the elements in the array after three passes of the heap sort algorithm?

```
44 78 22 7 98 56 34 2 38 35 45
```

6. An array contains the elements shown below. Using a quick sort, show the contents of the array after the first pivot has been placed in its correct location. Identify the three sublists that exist at that point.

```
44 78 22 7 98 56 34 2 38 35 45
```

8. After two passes of a sorting algorithm, the following array:

```
80 72 66 44 21 33
```

has been rearranged as shown below.

```
21 33 80 72 66 44
```

Which sorting algorithm is being used (straight selection, bubble, or straight insertion)? Defend your answer.

10. Show the result after each merge phase when merging the following two files:

```
6 12 19 23 34 · 8 11 17 20 25 · 9 10 15 25 35
```

```
13 21 27 28 29 · 7 30 36 37 39
```

17. Modify Program 12-2, “Heap Sort,” to count the number of data moves needed to order an array of 1000 random numbers. A data move is a movement of an element of data from one position in the array to another, to a hold area, or from a hold area back to the array. Display the array before and after the sort. At the end of the program, display the total moves needed to sort the array.

PROGRAM 12-2 Heap Sort

```
/* ===================== heapSort =====================
   Sort an array, [list0 .. last], using a heap.
      Pre  list must contain at least one item
           last contains index to last element in list
      Post list has been sorted smallest to largest
*/
void heapSort (int  list[ ], int  last)
{
// Local Definitions
   int sorted;
   int holdData;

// Statements
   // Create Heap
   for (int walker = 1; walker <= last; walker++)
      reheapUp (list, walker);

   // Heap created. Now sort it.
   sorted = last;
   while (sorted > 0)
      {
       holdData     = list[0];
       list[0]      = list[sorted];
       list[sorted] = holdData;
```

continued

PROGRAM 12-2 Heap Sort *(continued)*

```
         sorted--;
         reheapDown (list, 0, sorted);
        } // while
    return;
}  // heapSort

/* ==================== reheapUp ====================
   Reestablishes heap by moving data in child up to
   correct location heap array.
      Pre  heap is array containing an invalid heap
           newNode is index location to new data in heap
      Post newNode has been inserted into heap
*/
void reheapUp (int* heap, int newNode)
{
// Local Declarations
   int parent;
   int hold;

// Statements
   // if not at root of heap
   if (newNode)
      {
       parent = (newNode - 1)/ 2;
       if ( heap[newNode] > heap[parent] )
          {
           // child is greater than parent
           hold           = heap[parent];
           heap[parent]   = heap[newNode];
           heap[newNode] = hold;
           reheapUp (heap, parent);
          } // if heap[]
      } // if newNode
   return;
}  // reheapUp

/* ==================== reheapDown ====================
   Reestablishes heap by moving data in root down to its
   correct location in the heap.
      Pre  heap is an array of data
           root is root of heap or subheap
           last is an index to the last element in heap
      Post heap has been restored
*/
void reheapDown   (int* heap, int root, int last)
{
// Local Declarations
   int hold;
```

continued

PROGRAM 12-2 Heap Sort *(continued)*

```
   int leftKey;
   int rightKey;
   int largeChildKey;
   int largeChildIndex;

// Statements
   if ((root * 2 + 1) <= last)
       // There is at least one child
      {
       leftKey   = heap[root * 2 + 1];
       if ((root * 2 + 2) <= last)
          rightKey  = heap[root * 2 + 2];
       else
          rightKey  = -1;

       // Determine which child is larger
       if (leftKey > rightKey)
          {
           largeChildKey   = leftKey;
           largeChildIndex = root * 2 + 1;
          } // if leftKey
       else
          {
           largeChildKey   = rightKey;
           largeChildIndex = root * 2 + 2;
          } // else
       // Test if root > larger subtree
       if (heap[root] < heap[largeChildIndex])
           {
            // parent < children
            hold       = heap[root];
            heap[root] = heap[largeChildIndex];
            heap[largeChildIndex] = hold;
            reheapDown (heap, largeChildIndex, last);
           } // if root <
   } // if root
   return;
}  // reheapDown
```

22. Write an algorithm that applies the incremental idea of the shell sort to a selection sort. The algorithm first applies the straight section sort to items n / 2 elements apart (first, middle, and last). It then applies it to n / 3 elements apart, then to elements n / 4 apart, and so on.

CH. 13

2. An array contains the elements shown below. Using the binary search algorithm, trace the steps followed to find 20. At each loop iteration, including the last, show the contents of `first`, `last`, and `mid`.

```
18 13 17 26 44 56 88 97
```

3. Using the modulo-division method and linear probing, store the keys shown below in an array with 19 elements. How many collisions occurred? What is the density of the list after all keys have been inserted?

```
224562 137456 214562
140145 214576 162145
144467 199645 234534
```

5. Repeat Exercise 3 using the digit-extraction method (first, third, and fifth digits) and quadratic probing.

6. Repeat Exercise 5 using a linked list method for collisions. Compare the results in this exercise with the results you obtained in Exercise 5.

7. Repeat Exercise 3 using the midsquare method, with the center two digits, for hashing. Use a pseudorandom-number generator for rehashing if a collision occurs. Use `a` = 3 and `c` = −1 as the factors.

8. Repeat Exercise 7 using a key-offset method for collisions. Compare the results in this exercise with the results you obtained in Exercise 7.

11. Repeat Exercise 3 using the rotation method for hashing. First rotate the far-right digits two to the left and then use digit extraction (first, third, and fifth digits). Use the linear probe method to resolve collisions.

12. Repeat Exercise 11 using a key-offset method for collisions. Compare the results in this exercise with the results you obtained in Exercise 11.

Note: 這部分雖然多與其他題有相關，但主要需完成紅框題。

19. Write a program that uses a hashing algorithm to create a list of inventory parts and their quantities sold in the past month. After creating the hashed list, write a simple menu-driven user interface that allows the user to select from the following options:

a. Search for an inventory item and report its quantity sold
b. Print the inventory parts and their quantities sold
c. Analyze the efficiency of the hashing algorithm

The parts data are contained in a text file, as shown in Table 13-4. The key is the three-digit part number. The quantity represents the units sold during the past month.

Part number	Quantity
112	12
130	30
156	56
173	17
197	19
150	50
166	66
113	13
123	12
143	14
167	16
189	18
193	19
117	11
176	76

Three outputs are required from your program.

a. Test the following searches and return appropriate messages. You may test other part numbers if you desire, but the following tests must be completed first:

   - Search for 112
   - Search for 126
   - Search for 173

b. When requested, analyze the efficiency of the hashing algorithm for this set of data. Your printout should follow the report format shown below.

```
Percentage of Prime Area Filled:xx%
Average nodes in linked lists:  nn
Longest linked list             nn
```

c. The printout of the entire contents of the list should use the following format:

```
       Home Addr  Prime Area   Overflow List
          0         130/30
          1
          2         112/12
          3         123/12     143/14, 173/17, 193/19
          .
          .
          .
```