Solutions to Chapter 15

Review Questions

1. a. True
3. a. True
5. a. the order structure of elements
7. b. last in-first-out.
9. a. anywhere in the list
11. d. none of the above
13. b. queue
15. a. pop
17. d. dequeue
19. c. data structure

Exercises

21. At the beginning of the search, `pCur` is pointing the first node and `pPre` is null. So, the second statement (`pPre = pPre->link`) creates a run-time error. The correct code is:

```
pPre = pCur;
pCur = pCur->link;
```

23. The code to delete a node in the middle of a list is shown below/ Since this is the same code to delete from the beginning of the list, using a dummy node does simplify the insert for a linked list by eliminating the special case of deleting the first node.

```
pPre->link = pCur->link;
   free (pCur);
```

25.

```
pNew->link = pPre->link;
pPre->link = pNew;
```

Since this is the same code as what is used to add a node at the beginning of the list, using a dummy node can simplify the operations to a linked list by eliminating the special case of adding the first data node.

27. We append the second linked list to the first linked list.

29. See Figure 15-1

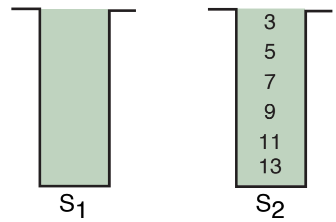

Figure 15-1 Solution to Exercise 29

31. See Figure 15-2.

Figure 15-2 Solution for Exercise 31

33. See Figure 15-3.

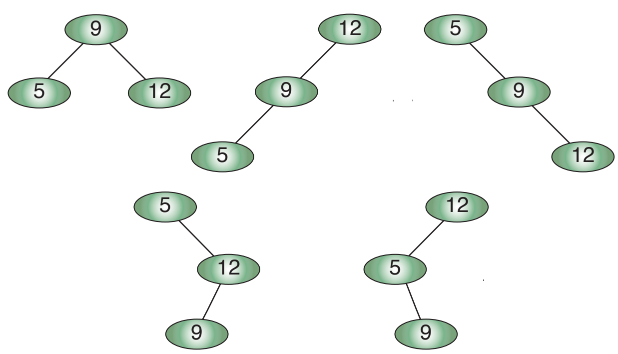

Figure 15-3 Solution to Exercise 33

35. See Figure 15-4.

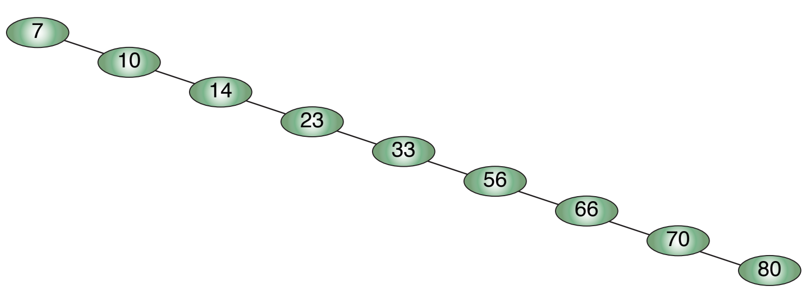

Figure 15-4 Solution to Exercise 35

37. See Figure 15-5.

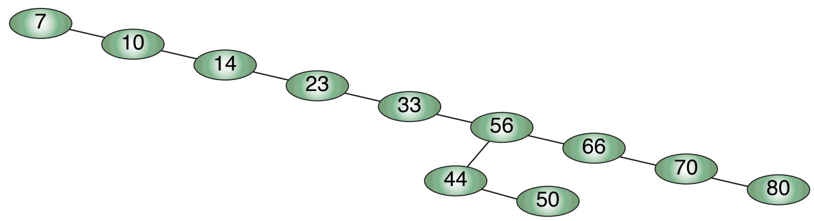

Figure 15-5 Solution to Exercise 37

Problems

39. See Program 15-1.

Program 15-1 Solution to Problem 39

```
// Global Declarations
   typedef int KEY_TYPE;

   typedef struct
      {
       KEY_TYPE key;
       // Other data goes here
      } DATA;

   typedef struct nodeTag
      {
       DATA            data;
       struct nodeTag* link;
      } NODE;

/* ================== findMinimum ===============
   This function accepts a linked list, traverses it,
   and returns the data in node with minimum key value.
      Pre   pList is a pointer to a linked list
      Post  returns pointer to data with minimum key
            NULL if list is empty
*/
DATA* findMinimum (NODE* pList)
{
// Local Declrations
   NODE* pWalker;
   DATA* minPtr =  NULL;

// Statements
   if (pList)
      {
       pWalker = pList;
       minPtr = &(pWalker->data);

       while (pWalker)
          {
           if (pWalker->data.key < minPtr->key)
               minPtr = &(pWalker->data);
           pWalker = pWalker->link;
          } // while
      } // if list !empty

   return minPtr;
}  // findMinimum
```

41. See Program 15-2.

Program 15-2 Solution to Problem 41

```
// Global Declarations
   typedef int KEY_TYPE;

   typedef struct
      {
       KEY_TYPE key;
       // Other data goes here
      } DATA;

   typedef struct nodeTag
      {
       DATA             data;
       struct nodeTag* link;
      } NODE;

/* =================== deleteAfter ================
   This function traverses a linked list and deletes
   all nodes that are after a node with negative key.
      Pre   pList is a pointer to a linked list
      Post  returns pointer to revised list
*/
NODE* deleteAfter (NODE* pList)
{
// Local Declrations
   NODE* pPre = NULL;
   NODE* pCur = pList;

// Statements
   while (pCur)
      {
       pPre = pCur;
       pCur = pCur->link;

       if (pPre->data.key < 0 && pCur)
          {
           pPre->link = pCur->link;
           free (pCur);
           pCur = pPre->link;
          } // if
      } // while
   return pList;
}  // deleteAfter
```

43. See Program 15-3.

Program 15-3 Solution to Problem 43

```
/* ==================== deleteNode ===================
   This function deletes a single node from the link
   designed with a dummy node.
      Pre   pList is a pointer to the head (dummy) node
            pPre points to node before the delete node
            pCur points to the node to be deleted
      Post  deletes and recycles pCur
*/
void deleteNode (NODE* pList, NODE* pPre, NODE* pCur)
{
// Statements
   pPre->link = pCur->link;
   free  (pCur);
   return;
}  // deleteNode
```

45. See Program 15-4.

Program 15-4 Solution to Problem 45

```
/* ==================== lastNode ===================
   This function returns a pointer to the last node in a
   linked list.
      Pre   pList is a valid linked list
      Post  returns a pointer to the last node
*/
NODE* lastNode (NODE* pList)
{
// Local Declrations
   NODE* pWalker;

// Statements
   pWalker = pList;

   while (pWalker->link != NULL)
          pWalker = pWalker->link;

   return pWalker;
}  // lastNode
```

47. See Program 15-5

Program 15-5 Solution to Problem 47

```
/* ================ doubleList ===============
   This function appends a linked list to itself.
      Pre   pList is a valid linked list
      Post  List appended to itself
*/
void doubleList (NODE* pList)
{
// Local Declrations
   NODE* pNewList;
   NODE* pNew;
   NODE* pRear;
   NODE* pPre;
   NODE* pWalker;

// Statements
   pWalker  = pList;
   pNewList = pPre = NULL;

   while (pWalker)
      {
       if (!(pNew = (NODE *) malloc (sizeof (NODE))))
          {
           printf ("\aMemory overflow in doubleList\n");
           exit (100);
          } // if
       pNew->data = pWalker->data;
       pNew->link = NULL;

       if (!pNewList)
           pNewList = pRear = pNew;
       else
           {
            pRear->link = pNew;
            pRear       = pNew;
           } // else not first node in new list

       pPre    = pWalker;
       pWalker = pWalker->link;
      } // while
```

Program 15-5 Solution to Problem 47 (continued)

```
   if (!pPre)
       ;       // list was empty to begin with
   else
      pPre->link = pNewList;
   return;
}  // doubleList
```

NOTE

To make it easier to solve the following stack problems, we created a header file containing the stack functions developed in the text. It is named “`P15-STACK.H`.”

49. See Program 15-6.

Program 15-6 Solution to 49

```
/* Copy the source stack to the destination stack
   preserving the top-to-base ordering.
      Pre      sourceStack is a valid stack
      Post     newStack is a copy of sourceStack
      Returns: true if successful; false if not
*/

// Error Abort Macro
#define ABORT {free (tempStack); return false;}

bool copyStack (STACK* sourceStack, STACK* newStack)
{
// Local Declarations
   STACK* tempStack;
   int    data;

// Statements
   tempStack = malloc(sizeof(STACK));
   if (!tempStack)
       return false;
   tempStack->top   = NULL;
   tempStack->count = 0;

   while (sourceStack->count != 0)
      {
       if (pop  (sourceStack, &data))
           push (tempStack,   data);
       else
           ABORT;
      } // while

   // Now copy data to new and original stack
   while (tempStack->count != 0)
      {
       if (pop (tempStack,  &data))
          {
           if (!push (sourceStack, data))
               ABORT;
           if (!push (newStack,    data))
               ABORT;
          } // if
       else
           ABORT;
      } // while

   // Now free tempStack memory
   free (tempStack);
```

Program 15-6 Solution to 49 (continued)

```
   return true;
}  // copyStack
```

51. See Program 15-7.

Program 15-7 Solution to Problem 51

```
/* ================== equalStack ===================
    This function determines if the contents of one
    stack are identical to that of another.
       Pre    stack1 and stack2 are valid stacks
       Post   contents of stack1 and stack2 compared
       Return true  if the stacks are equal
              false if the stacks are not equal
*/
bool equalStack (STACK* stack1, STACK* stack2)
{
// Local Definitions
   bool   equal;
   int    data1;
   int    data2;
   STACK* temp1;
   STACK* temp2;

// Statements
   if (stack1->count != stack2->count)
      return false;

   equal = true;

   temp1 = malloc(sizeof(STACK));
   if (!temp1)
      printf("Error allocating stack"), exit(100);
   temp1->top   = NULL;
   temp1->count = 0;

   temp2 = malloc(sizeof(STACK));
   if (!temp2)
      printf("Error allocating stack"), exit(100);
   temp2->top   = NULL;
   temp2->count = 0;

   while (pop(stack1, &data1))
          push(temp1, data1);

   while (pop(stack2, &data2))
          push(temp2, data2);

   while (temp1->count != 0)
         {
          pop(temp1, &data1);
          pop(temp2, &data2);

          if (data1 != data2)
             equal = false;

          push(stack1, data1);
          push(stack2, data2);
         } // while

   free (temp1);
   free (temp2);
   return equal;
}  // equalStack

/* ================ printStack ===================
```

Program 15-7 Solution to Problem 51 (continued)

```
   A non-standard function that prints a stack. It is
   non-standard because it accesses the stack structures.
      Pre  stack is a valid stack
      Post stack data printed, top to base
*/
void printStack(STACK* stack)
{
// Local Definitions
   STACK_NODE* node = stack->top;

// Statements
   printf ("Top=>");
   while (node)
      {
       printf ("%3d", node->data);
       node = node->link;
      } // while
   printf("<=Base\n");
   return;
}  // printStack
```

NOTE

To make it easier to solve the following queue problems, we created a header file containing the queue functions developed in the text. It is named "`P15-QUEUEU.H`."

53. See Program 15-8.

Program 15-8 Solution to Problem 53

```
/* =================== stackToQueue ===================
   This algorithm creates a queue from a stack. Top item
   becomes queue front; base item becomes queue rear.
      Pre    stack is a valid stack
      Post   stack is empty--data now in queue
      Return address of new queue--null if failure
*/
QUEUE* stackToQueue (STACK* stack)
{
// Local Declarations
   QUEUE* queue;
   int    data;

// Statements
   queue = malloc (sizeof(QUEUE));
   if (!queue)
       return NULL;
   queue->front = NULL;
   queue->rear  = NULL;
   queue->count = 0;

   while (stack->count)
      {
       pop (stack, &data);
       if (!enqueue (queue, data))
           return NULL;
      } // while
   return queue;
}  // stackToQueue
```

55. See Program 15-9.

Program 15-9 Solution to Problem 55

```
/* ==================== dequeueNeg ===================
   Delete queue nodes with negative integers.
      Pre  queue is a queue of integers
      Post nodes with negative integers deleted
*/
void dequeueNeg (QUEUE* queue)
{
// Local Declarations
   int num;

// Statements
   for (int i = queue->count; i > 0; i--)
       {
        dequeue (queue, &num);
        if (num >= 0)
            enqueue (queue, num);
       } // for
   return;
}  // dequeueNeg
```

NOTE

To make it easier to solve the following binary tree problems, we created a header file containing the queue functions developed in the text. It is named "`P15-TREE.H`."

57. See Program 15-10.

Program 15-10 Solution to Problem 57

```
/* ================== countLeaves ==================
   Count the number of leaves in a binary tree.
      Pre  tree is a pointer to root of a binary tree
      Post returns count of leaves
*/
int countLeaves (NODE* root)
{
// Statements
   if (!root)
      return 0;
   if (!root->left && !root->right)
       return 1;
   return (countLeaves(root->left)
        +  countLeaves(root->right));
}  // countLeaves
```