17 Linked Lists

REVIEW QUESTIONS

1. Each element in a linked list must contain data and a link field.
   a. True
3. The first step in adding a node to a linked list is to allocate memory for the new node.
   a. True
5. A(n) __________ is an ordered collection of data in which each element contains the location of the next element.
   e. linked list
7. A(n) __________ identifies the first logical node in a linked list.
   b. head pointer
9. Which of the following statements about linked list deletes is false?
   c. Deletion of the rear node requires a separate test to set the predecessor's link to 0.

EXERCISES

11. At the beginning of a linked link, pCur is pointing the first node as well as pHead and pPre is null. So, the second statement (pPre = pPre->link) does not work.

```
plink = pCur;
pCur  = pCur->link;
```

13. The addition of a dummy node simplifies the operations on a linked list because we can use the same logic for deleting a node anywhere in the list.

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

15. The addition of a dummy node simplifies the operations on a linked list because we can use the same logic for adding a node anywhere in the list.

```
pNew->link = pPre->link;
pPre->link = pNew;We will append the second linked list
at the end of the first linked list.
```

17. We will append the second linked list at the end of the first linked list.

PROBLEMS

19. This solution creates a linked list in the order the data are entered; that is, it appends each new node to the end of the list. It can be used as the driver for the problems that follow.

```
/* This program reads a list of integers from the
   keyboard, creates a linked list out of them,
   and prints the result.
      Written by:
      Date:
*/
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;


struct NODE
   {
    int   key;
    int   data;
    NODE* link;
   }; // NODE

void print_list (NODE*);

int main (void)
{
    cout << "Please enter a list of numbers for a "
         << "linked list (<EOF> to stop):\n";

    int num;
    NODE* p_list = 0;
    NODE* p_new;
    NODE* p_rear = 0;
    while (cin >> num)
       {
        p_new = new NODE;
        if (!p_new)
           {
            cerr << "**Can not allocate node\n";
            exit (100);
           } // if  !p_new

        p_new->key  = num;
        p_new->data = rand();
        p_new->link = 0;

        if (p_list == 0)
           // first node
           p_list  =  p_rear  =  p_new;
        else
           {
            p_rear->link  =  p_new;
            p_rear        =  p_new;
           } // else
       } // while

    cout << "\nLink list complete.\n";
    print_list (p_list);
```

```
   cout << "\n\n*** end of program ***\n\n";
   return 0;
}  // main

/* ================== print_list ==================
   Traverse and print a linked list
      Pre   p_list is a valid linked list
      Post  List has been printed
*/
void print_list (NODE* p_list)
{
   cout << "\nList contains :\n";

   NODE* p_walker =  p_list;;
   int line_count  =  0;
   while (p_walker)
      {
       if (++line_count > 5)
          {
           line_count = 1;
           cout << endl;
          } // if
       cout << setw (5) << p_walker->key
            << setw (7) << p_walker->data
            << " | ";
       p_walker = p_walker->link;
      } // while
   cout << endl;
   return;
}  // print_list
```

21.

```
struct NODE
   {
    int   key;
    int   data;
    NODE* link;
   };  // NODE

/* ================= delete_negative ===============
   Deletes all negative linked list nodes
      Pre  p_list is a pointer to a linked list
           passed by reference
      Post Negative nodes deleted
           Returns number of nodes deleted
*/
int delete_negative (NODE*& p_list)
{
   int   count    =  0;
   NODE* p_walker =  p_list;
   NODE* p_del;

   while (p_walker)
      {
       if (p_walker->key  <  0)
          {
           p_del     =  p_walker;
           p_walker  =  p_walker->link;
           delete_node (p_list, p_del->key);
```

```
            count++;
          }  // if negative key
       else
          p_walker  =  p_walker->link;
      } // while
   return count;
}  // delete_negative

/* ================== delete_node =================
   This function deletes a node from a linked list.
      Pre  p_list is a pointer to a linked list
           target is  key value of delete node
      Post Return true if successful, false if not
*/
 bool delete_node (NODE*& p_list, int target)
{
   NODE* p_pre;
   NODE* p_cur;

   bool success = search_list (p_list, p_pre,
                               p_cur,  target);
   if (success)
      {
       if (p_pre  ==  0)
           p_list  =  p_cur->link;
       else
           p_pre->link  =  p_cur->link;

       delete (p_cur);
      } // if target found
   return success;
}  // delete_node
/* ================= search_list ==================
   This function searches for a node in a linked list.
      Pre  p_list is a pointer to a linked list
           p_pre is a pre pointer
           p_cur is a current pointer
           target is the key value for search
      Post Returns true if found, false if not
*/
bool search_list (NODE*  p_list, NODE*& p_pre,
                  NODE*& p_cur,  int    target)
{
   p_pre  =  0;
   p_cur  =  p_list;
   while (p_cur  &&  p_cur->key  !=  target)
      {
       p_pre  =  p_cur;
       p_cur  =  p_cur->link;
      }

   bool found;
   if (p_cur  &&  p_cur->key  ==  target)
      found  =  true;
   else
      found  =  false;
   return found;
}  // search_list
```

23.

```
/* ================ delete_before_neg ===============
   Traverse a linked list and delete any node that
   immediately followed by a node with a negative key.
      Pre  p_list is a pointer to a linked list
      Post Returns number of nodes deleted
*/
int delete_before_neg (NODE*& p_list)
{
   int   count    = 0;
   NODE* p_walker = p_list;
   NODE* p_pred   = 0;
   NODE* deleteOn = false;
   NODE* p_delete = 0;

   while (p_walker)
      {
       if (p_walker->link && p_walker->link->key < 0)
          {
           if (p_pred)
              // Not first node
              p_pred->link = p_walker->link;
           else
              p_list       = p_walker->link;
           p_delete     = p_walker;
           count++;
          } // if

       if (!p_delete)
       // change p_pred only if not deleting p_walker
          p_pred = p_walker;
       p_walker = p_walker->link;
       if (p_delete)
          {
           delete (p_delete);
           p_delete = 0;
          } // if
      } // while
   return count;
}  // delete_before_neg
```

25.

```
struct NODE
   {
    int   key;
    int   data;
    NODE* link;
   };  // NODE

/* =================== add_node ====================
   Inserts a single node into a linked list with a
   dummy node.
      Pre  pList is a pointer to the list
           key of node to be inserted in list
      Post key inserted in sequence
*/
void add_node (NODE*& pList, int key)
{
   NODE* pNew = new NODE;
   if (!pNew)
```

```
        {
         cerr << "\a\n**Allocate error in add_node\n";
         exit (300);
        } // if  !pNew

    pNew->key  =  key;
    pNew->link =  0;

    NODE* p_pre;
    NODE* p_cur;
    if (search_list (pList, p_pre, p_cur, key))
        {
         cout << "\a\n=== "
              << key << " already in list ===\n\n";
         delete (pNew);
        } // if dupe
    else
        {
         pNew->link  =  p_pre->link;
         p_pre->link =  pNew;
        } // else
    return;
}   // add_node

/*  =================== search_list ===================
    Given key value, finds the location of a node.
       Pre  pList is a pointer to a head node
            pPre is pointer to predecessor
            pCur is pointer to current node
            target is the key being sought
       Post pCur points to first node with >= key
            or null if target > key of last node
            pPre points to largest node < than key
            or null if target <= key of first node
            returns true if found,
                    false if not found
*/
bool search_list (NODE   *pList, NODE*& pPre,
                  NODE*&  pCur,  int    target)
{
    pPre   =  pList;
    pCur   =  pList->link;

    while (pCur  &&  target > pCur->key)
        {
         pPre   =  pCur;
         pCur   =  pCur->link;
        } // while

    return (pCur  &&  pCur->key  ==  target);
}   // search_list
```

27. See Problem 25.

29.

```
/*  ==================== append ====================
    Append second list to the end of the first list.
       Pre  The lists have been created
       Post The second list appended to the first
*/
```

```
void append (NODE*& list1, NODE*& list2)
{
   // See Problem 28 for last_node
   NODE* pLast  =  last_node (list1);
   if (pLast)
      {
       pLast->link  =  list2;
      } // if list1 not empty
   else
      list1  =  list2;
   return;
}  // append
```

31.

```
/*  ==================== swap_node ===================
   Swap two nodes in a linked list. The nodes are
   identified by number and are passed as parameters.
      Pre  pList is a linked list
      Post If successful, return true,
           if unsuccessful, return false
*/

#define ERR1 "\n**Cannot swap node with itself\n"
#define ERR2 "\n**Invalid first number\n"
#define ERR3 "\n**Invalid second numbet\n"

int swap_node (NODE*& pList, int first, int second)
{
   if (first == second)
      {
       cout << ERR1;
       return false;
      } // ERR1

   int   count      =  1;
   NODE* pPreFirst  =  0;
   NODE* pFirst     =  pList;

   while (pFirst && count < first)
      {
       pPreFirst  =  pFirst;
       pFirst     =  pFirst->link;
       count++;
      } // look for first

   bool success;
   if (pFirst  &&  count == first)
       success  =  true;
   else
      {
       success  =  false;
       cout << ERR2;
      } // ERR2

   NODE* pPreSecond;
   NODE* pSecond;
   if (success)
      {
       count      =  1;
       pPreSecond =  0;
```

```
        pSecond    =  pList;
        while (pSecond && count < second)
           {
            pPreSecond  =  pSecond;
            pSecond     =  pSecond->link;
            count++;
           } // look for second

        if (pSecond && count == second)
            success  =  true;
        else
           {
            success  =  false;
            cout << ERR3;
           } // ERR3

     if (success)
        {
         if   (pPreFirst  ==  0
           ||  pPreSecond  ==  0)
             {
              if (pPreFirst  ==  0)
                 {
                  pPreSecond->link   =  pFirst;
                  pList              =  pSecond;
                 } // if first is first node in list
              if (pPreSecond   ==  0)
                 {
                  pPreFirst->link    =  pSecond;
                  pList              =  pFirst;
                 } // if second is first node in list

           }  // if first or second are the first node

         else
             {
              pPreFirst->link   =  pSecond ;
              pPreSecond->link  =  pFirst;
             } // if both nodes are in middle of list

         NODE* pTemp     =  pFirst->link;
         pFirst->link    =  pSecond->link;
         pSecond->link   =  pTemp;
        } // if search for second was successful
    return;
}   // swap_node
```