12

C Data Structures

© 2007 Pearson Education, Inc. All rights reserved.

12.1 Introduction

- **Dynamic data structures**
  - **Data structures that grow and shrink during execution**
- **Linked lists**
  - **Allow insertions and removals anywhere**
- **Stacks**
  - **Allow insertions and removals only at top of stack**
- **Queues**
  - **Allow insertions at the back and removals from the front**
- **Binary trees**
  - **High-speed searching and sorting of data and efficient elimination of duplicate data items**

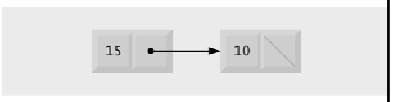
© 2007 Pearson Education, Inc. All rights reserved.

12.2 Self-Referential Structures

- **Self-referential structures**
  - **Structure that contains a pointer to a structure of the same type**
  - **Can be linked together to form useful data structures such as lists, queues, stacks and trees**
  - **Terminated with a `NULL` pointer (`0`)**

```
struct node {
   int data;
   struct node *nextPtr;
}
```

- `nextPtr`
  - **Points to an object of type `node`**
  - **Referred to as a link**
    - **Ties one `node` to another `node`**



© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 12.1

Not setting the link in the last node of a list to `NULL` can lead to runtime errors.

→ Assign `NULL` to the link member of a new node. Pointers should be initialized before they are used.

© 2007 Pearson Education, Inc. All rights reserved.

12.3 Dynamic Memory Allocation

- **Dynamic memory allocation**
  - **Obtain and release memory during execution**
- **`malloc`**
  - **Takes number of bytes to allocate**
    - **Use `sizeof` to determine the size of an object**
  - **Returns pointer of type `void *`**
    - **A `void *` pointer may be assigned to any pointer**
    - **If no memory available, returns `NULL`**
  - **Example**

```
newPtr = malloc( sizeof( struct node ) );
```

- **`free`**
  - **Deallocates memory allocated by `malloc`**
  - **Takes a pointer as an argument**
  - `free ( newPtr );`

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 12.2

Assuming that the size of a structure is simply the sum of the sizes of its members is a logic error.

→ Use the `sizeof` operator to determine the size of a structure.

© 2007 Pearson Education, Inc. All rights reserved.

Error-Prevention Tip 12.1

When using `malloc`, test for a `NULL` pointer return value. Print an error message if the requested memory is not allocated.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Error 12.3

Not returning dynamically allocated memory when it is no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."

→ When memory that was dynamically allocated is no longer needed, use `free` to return the memory to the system immediately.

© 2007 Pearson Education, Inc. All rights reserved.

Common Programming Errors

12.4: Freeing memory not allocated dynamically with `malloc` is an error.

12.5: Referring to memory that has been freed is an error.

© 2007 Pearson Education, Inc. All rights reserved.

21

Classes: A Deeper Look, Part 2

© 2006 Pearson Education, Inc. All rights reserved.

21.6 Dynamic Memory Management with Operators `new` and `delete`

- **Dynamic memory management**
  - **Enables programmers to allocate and deallocate memory for any built-in or user-defined type**
  - **Performed by operators `new` and `delete`**
  - **For example, dynamically allocating memory for an array instead of using a fixed-size array**

© 2006 Pearson Education, Inc. All rights reserved.

21.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- **Operator `new`**
  - **Allocates (i.e., reserves) storage of the proper size for an object at execution time**
  - **Calls a constructor to initialize the object**
  - **Returns a pointer of the type specified to the right of `new`**
  - **Can be used to dynamically allocate any fundamental type (such as `int` or `double`) or any class type**
- **Free store**
  - **Sometimes called the heap**
  - **Region of memory assigned to each program for storing objects created at execution time**

© 2006 Pearson Education, Inc. All rights reserved.

21.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- Operator `delete`
  - Destroys a dynamically allocated object
  - Calls the destructor for the object
  - Deallocates (i.e., releases) memory from the free store
  - The memory can then be reused by the system to allocate other objects

© 2006 Pearson Education, Inc. All rights reserved.

21.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- Initializing an object allocated by `new`
  - Initializer for a newly created fundamental-type variable
    - Example
      - `double *ptr = new double( 3.14159 );`
  - Specify a comma-separated list of arguments to the constructor of an object
    - Example
      - `Time *timePtr = new Time( 12, 45, 0 );`

© 2006 Pearson Education, Inc. All rights reserved.

21.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- `new` operator can be used to allocate arrays dynamically
  - Dynamically allocate a 10-element integer array:
    ```
    int *gradesArray = new int[ 10 ];
    ```
  - Size of a dynamically allocated array
    - Specified using any integral expression that can be evaluated at execution time

© 2006 Pearson Education, Inc. All rights reserved.

21.6 Dynamic Memory Management with Operators `new` and `delete` (Cont.)

- Delete a dynamically allocated array:
  ```
  delete [] gradesArray;
  ```
  - This deallocates the array to which `gradesArray` points
  - If the pointer points to an array of objects
    - First calls the destructor for every object in the array
    - Then deallocates the memory
  - If the statement did not include the square brackets (`[]`) and `gradesArray` pointed to an array of objects
    - Only the first object in the array would have a destructor call

© 2006 Pearson Education, Inc. All rights reserved.

Common Programming Error 21.9

Using `delete` instead of `delete []` for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator `delete []`. Similarly, always delete memory allocated as an individual element with operator `delete`.

© 2006 Pearson Education, Inc. All rights reserved.

Error-Prevention Tip 21.2

After deleting dynamically allocated memory, set the pointer that referred to that memory to `0`. This disconnects the pointer from the previously allocated space on the free store. This space in memory could still contain information, despite having been deleted. By setting the pointer to `0`, the program loses any access to that free-store space, which, in fact, could have already been reallocated for a different purpose. If you didn't set the pointer to `0`, your code could inadvertently access this new information, causing extremely subtle, nonrepeatable logic errors.

© 2006 Pearson Education, Inc. All rights reserved.

12.4 Linked Lists

- **Linked list**
  - **Linear collection of self-referential class objects, called nodes**
  - **Connected by pointer links**
  - **Accessed via a pointer to the first node of the list**
  - **Subsequent nodes are accessed via the link-pointer member of the current node**
  - **Link pointer in the last node is set to `NULL` to mark the list's end**
- **Use a linked list instead of an array when**
  - **You have an unpredictable number of data elements**
  - **Your list needs to be sorted quickly**

© 2007 Pearson Education, Inc. All rights reserved.

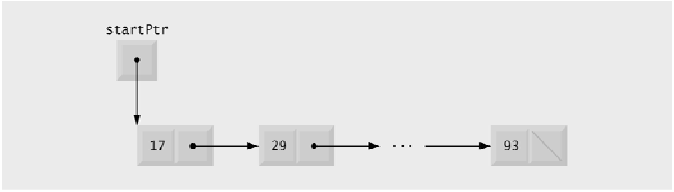


Fig. 12.2 | Linked list graphical representation.

© 2007 Pearson Education, Inc. All rights reserved.

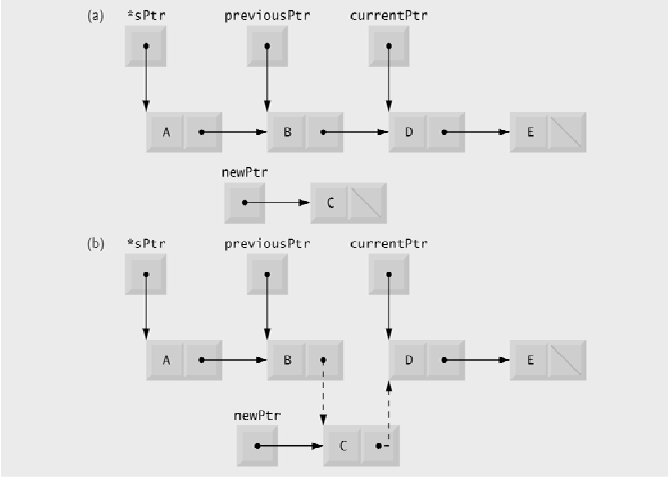


Fig. 12.5 | Inserting a node in order in a list.

© 2007 Pearson Education, Inc. All rights reserved.

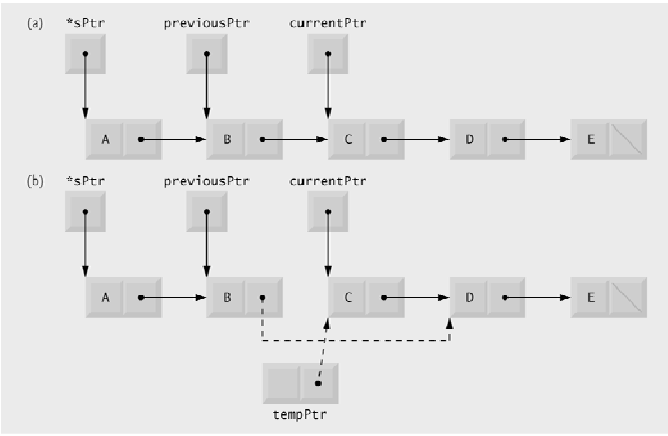


Fig. 12.6 | Deleting a node from a list.

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 12.1

An array can be declared to contain more elements than the number of data items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations.

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 12.2

Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 12.3

The elements of an array are stored contiguously in memory. This allows immediate access to any array element because the address of any element can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate access to their elements.

© 2007 Pearson Education, Inc. All rights reserved.

Performance Tip 12.4

Using dynamic memory allocation (instead of arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that the pointers take up space, and that dynamic memory allocation incurs the overhead of function calls.

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
1  /* Fig. 12.3: fig12_03.c
2     Operating and maintaining a list */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* self-referential structure */
7  struct listNode {
8     char data; /* each listNode contains a character */
9     struct listNode *nextPtr; /* pointer to next node */
10 }; /* end structure listNode */
11
12 typedef struct listNode ListNode; /* synonym for struct listNode */
13 typedef ListNode *ListNodePtr; /* synonym for ListNode* */
14
15 /* prototypes */
16 void insert( ListNodePtr *sPtr, char value );
17 char delete( ListNodePtr *sPtr, char value );
18 int isEmpty( ListNodePtr sPtr );
19 void printList( ListNodePtr currentPtr );
20 void instructions( void );
21
22 int main( void )
23 {
24    ListNodePtr startPtr = NULL; /* initially there are no nodes */
25    int choice; /* user's choice */
26    char item;  /* char entered by user */
27
28    instructions(); /* display the menu */
29    printf( "? " );
30    scanf( "%d", &choice );
```

Each node in the list contains a data element and a pointer to the next node

fig12_03.c

(1 of 8)

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
31
32    /* loop while user does not choose 3 */
33    while ( choice != 3 ) {
34
35       switch ( choice ) {
36
37          case 1:
38             printf( "Enter a character: " );
39             scanf( "\n%c", &item );
40             insert( &startPtr, item ); /* insert item in list */
41             printList( startPtr );
42             break;
43
44          case 2: /* delete an element */
45
46             /* if list is not empty */
47             if ( !isEmpty( startPtr ) ) {
48                printf( "Enter character to be deleted: " );
49                scanf( "\n%c", &item );
50
51                /* if character is found, remove it*/
52                if ( delete( &startPtr, item ) ) { /* remove item */
53                   printf( "%c deleted.\n", item );
54                   printList( startPtr );
55                } /* end if */
56                else {
57                   printf( "%c not found.\n\n", item );
58                } /* end else */
59
60             } /* end if */
```

fig12_03.c

(2 of 8)

Function **`insert`** inserts data into the list

Function **`delete`** removes data from the list

© 2007 Pearson Education, Inc. All rights reserved.

```
61             else {
62                printf( "List is empty.\n\n" );
63             } /* end else */
64
65             break;
66
67          default:
68             printf( "Invalid choice.\n\n" );
69             instructions();
70             break;
71
72       } /* end switch */
73
74       printf( "? " );
75       scanf( "%d", &choice );
76    } /* end while */
77
78    printf( "End of run.\n" );
79
80    return 0; /* indicates successful termination */
81
82 } /* end main */
83
```

Outline

fig12_03.c

(3 of 8)

© 2007 Pearson Education, Inc. All rights reserved.


```
84 /* display program instructions to user */
85 void instructions( void )
86 {
87    printf( "Enter your choice:\n"
88       "   1 to insert an element into the list.\n"
89       "   2 to delete an element from the list.\n"
90       "   3 to end.\n" );
91 } /* end function instructions */
92
93 /* Insert a new value into the list in sorted order */
94 void insert( ListNodePtr *sPtr, char value )
95 {
96    ListNodePtr newPtr;      /* pointer to new node */
97    ListNodePtr previousPtr; /* pointer to previous node in list */
98    ListNodePtr currentPtr;  /* pointer to current node in list */
99
100   newPtr = malloc( sizeof( ListNode ) ); /* create node */
101
102   if ( newPtr != NULL ) { /* is space available */
103      newPtr->data = value; /* place value in node */
104      newPtr->nextPtr = NULL; /* node does not link to another node */
105
106      previousPtr = NULL;
107      currentPtr = *sPtr;
108
109      /* loop to find the correct location in the list */
110      while ( currentPtr != NULL && value > currentPtr->data ) {
111         previousPtr = currentPtr;          /* walk to ...   */
112         currentPtr = currentPtr->nextPtr;  /* ... next node */
113      } /* end while */
```

Outline

fig12_03.c

(4 of 8)

To insert a node into the list, memory must first be allocated for that node

while loop searches for new node's place in the list

© 2007 Pearson Education, Inc. All rights reserved.

```
114
115      /* insert new node at beginning of list */
116      if ( previousPtr == NULL ) {
117         newPtr->nextPtr = *sPtr;
118         *sPtr = newPtr;
119      } /* end if */
120      else { /* insert new node between previousPtr and currentPtr */
121         previousPtr->nextPtr = newPtr;
122         newPtr->nextPtr = currentPtr;
123      } /* end else */
124
125   } /* end if */
126   else {
127      printf( "%c not inserted. No memory available.\n", value );
128   } /* end else */
129
130 } /* end function insert */
131
132  /* Delete a list element */
133 char delete( ListNodePtr *sPtr, char value )
134 {
135   ListNodePtr previousPtr; /* pointer to previous node in list */
136   ListNodePtr currentPtr;  /* pointer to current node in list */
137   ListNodePtr tempPtr;     /* temporary node pointer */
138
```

If there are no nodes in the list, the new node becomes the "start" node

Otherwise, the new node is inserted between two others (or at the end of the list) by changing pointers

Outline

fig12_03.c

(5 of 8)

© 2007 Pearson Education, Inc. All rights reserved.


```
139   /* delete first node */
140   if ( value == ( *sPtr )->data ) {
141      tempPtr = *sPtr; /* hold onto node being removed */
142      *sPtr = ( *sPtr )->nextPtr; /* de-thread the node */
143      free( tempPtr ); /* free the de-threaded node */
144      return value;
145   } /* end if */
146   else {
147      previousPtr = *sPtr;
148      currentPtr = ( *sPtr )->nextPtr;
149
150      /* loop to find the correct location in the list */
151      while ( currentPtr != NULL && currentPtr->data != value ) {
152         previousPtr = currentPtr;         /* walk to ...   */
153         currentPtr = currentPtr->nextPtr; /* ... next node */
154      } /* end while */
155
156      /* delete node at currentPtr */
157      if ( currentPtr != NULL ) {
158         tempPtr = currentPtr;
159         previousPtr->nextPtr = currentPtr->nextPtr;
160         free( tempPtr );
161         return value;
162      } /* end if */
```

Outline

fig12_03.c

(6 of 8)

while loop searches for node's place in the list

Once the node is found, it is deleted by changing pointers and freeing the node's memory

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig12_03.c

(7 of 8)

```
163
164   } /* end else */
165
166   return '\0';
167
168 } /* end function delete */
169
170 /* Return 1 if the list is empty, 0 otherwise */
171 int isEmpty( ListNodePtr sPtr )
172 {
173   return sPtr == NULL;
174
175 } /* end function isEmpty */
176
177 /* Print the list */
```

If the start node is **NULL**, there are no nodes in the list

© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig12_03.c

(8 of 8)

```
178 void printList( ListNodePtr currentPtr )
179 {
180
181   /* if list is empty */
182   if ( currentPtr == NULL ) {
183      printf( "List is empty.\n\n" );
184   } /* end if */
185   else {
186      printf( "The list is:\n" );
187
188      /* while not the end of the list */
189      while ( currentPtr != NULL ) {
190         printf( "%c --> ", currentPtr->data );
191         currentPtr = currentPtr->nextPtr;
192      } /* end while */
193
194      printf( "NULL\n\n" );
195   } /* end else */
196
197 } /* end function printList */
```

© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL


? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL
```

(continued on next slide…)

© 2007 Pearson Education, Inc. All rights reserved.

Outline

(continued from previous slide…)

```
? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.

? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

© 2007 Pearson Education, Inc. All rights reserved.