Data Structure

Outline

- Homework
- Linked List
- Stack
- Queue
- Tree

Assignment 2

- Solve 10 CodeAbbey Problems
- Target total blessings: 50
- You are only allowed to submit 10 solutions not more
- Same grading & submission rules as last time
- Due: in a week

Self Referential Structure

- Self referential structure is useful for Linked List
- However to use it efficiently we need malloc
- Due to the nature of List which grows & shrink dynamically

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

Malloc and Free

- Function malloc takes as an argument the number of bytes to be allocated and returns a pointer of type void * (pointer to void)
  - a void * pointer may be assigned to a variable of any pointer type.
- Malloc typically mixed with sizeof, ex:

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

- Function free deallocates memory, ex:

```
free(newPtr);
```

Calloc & Realloc

- Calloc is similar to malloc in purpose

```
void *calloc(size_t nmemb, size_t size);
```

- The difference, it clears the memory it allocates to 0 values
- Malloc is faster than calloc
- Function realloc changes the size of an object allocated by a previous call to malloc, calloc, or realloc. Ex:

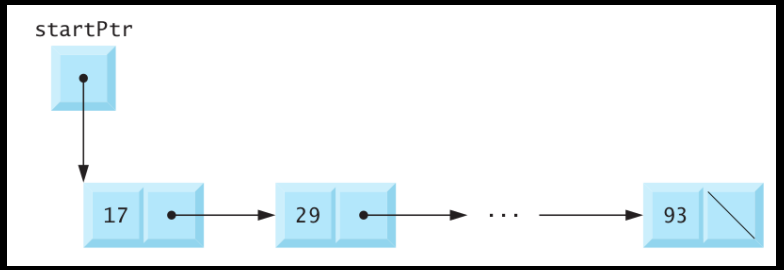
```
void *realloc(void *ptr, size_t size);
```

Data Structure Overview

- Linked lists are collections of data items “lined up in a row”—insertions and deletions are made anywhere in a linked list.
- Stacks are important in compilers and operating systems—insertions and deletions are made only at one end of a stack—its top.
- Queues represent waiting lines; insertions are made only at the back (also referred to as the tail) of a queue and deletions are made only from the front (also referred to as the head) of a queue.
- Binary trees facilitate high-speed searching and sorting of data, efficiently eliminating duplicate data items and compiling expressions into machine language.

Linked List

- Self Referential Structure is a List Attribute.
- However List is not just data, it needs to be able to operate at minimum insert & delete.
- Hence a List in C is both Struct & Function



```c
// Fig. 12.3: fig12_03.c
// Inserting and deleting nodes in a list
#include <stdio.h>
#include <stdlib.h>

// self-referential structure
struct listNode {
   char data; // each listNode contains a character
   struct listNode *nextPtr; // pointer to next node
};

typedef struct listNode ListNode; // synonym for struct listNode
typedef ListNode *ListNodePtr; // synonym for ListNode*

// prototypes
void insert(ListNodePtr *sPtr, char value);
char delete(ListNodePtr *sPtr, char value);
int isEmpty(ListNodePtr sPtr);
void printList(ListNodePtr currentPtr);
void instructions(void);

```

```
22  int main(void)
23  {
24     ListNodePtr startPtr = NULL; // initially there are no nodes
25     char item; // char entered by user
26
27     instructions(); // display the menu
28     printf("%s", "? ");
29     unsigned int choice; // user's choice
30     scanf("%u", &choice);
31
32     // loop while user does not choose 3
33     while (choice != 3) {
34
35        switch (choice) {
36           case 1:
37              printf("%s", "Enter a character: ");
38              scanf("\n%c", &item);
39              insert(&startPtr, item); // insert item in list
40              printList(startPtr);
41              break;
42           case 2: // delete an element
43              // if list is not empty
44              if (!isEmpty(startPtr)) {
45                 printf("%s", "Enter character to be deleted: ");
46                 scanf("\n%c", &item);
47
```

```
48                  // if character is found, remove it
49                  if (delete(&startPtr, item)) { // remove item
50                     printf("%c deleted.\n", item);
51                     printList(startPtr);
52                  }
53                  else {
54                     printf("%c not found.\n\n", item);
55                  }
56               }
57               else {
58                  puts("List is empty.\n");
59               }
60
61               break;
62            default:
63               puts("Invalid choice.\n");
64               instructions();
65               break;
66         }
67
68         printf("%s", "? ");
69         scanf("%u", &choice);
70      }
71
72      puts("End of run.");
73   }
```

```
75  // display program instructions to user
76  void instructions(void)
77  {
78     puts("Enter your choice:\n"
79        "   1 to insert an element into the list.\n"
80        "   2 to delete an element from the list.\n"
81        "   3 to end.");
82  }
83
```

```
84  // insert a new value into the list in sorted order
85  void insert(ListNodePtr *sPtr, char value)
86  {
87     ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node
88
89     if (newPtr != NULL) { // is space available?
90        newPtr->data = value; // place value in node
91        newPtr->nextPtr = NULL; // node does not link to another node
92
93        ListNodePtr previousPtr = NULL;
94        ListNodePtr currentPtr = *sPtr;
95
96        // loop to find the correct location in the list
97        while (currentPtr != NULL && value > currentPtr->data) {
98           previousPtr = currentPtr; // walk to ...
99           currentPtr = currentPtr->nextPtr; // ... next node
100       }
```

```
101
102         // insert new node at beginning of list
103         if (previousPtr == NULL) {
104            newPtr->nextPtr = *sPtr;
105            *sPtr = newPtr;
106         }
107         else { // insert new node between previousPtr and currentPtr
108            previousPtr->nextPtr = newPtr;
109            newPtr->nextPtr = currentPtr;
110         }
111      }
112      else {
113         printf("%c not inserted. No memory available.\n", value);
114      }
115   }
```

```
117 // delete a list element
118 char delete(ListNodePtr *sPtr, char value)
119 {
120    // delete first node if a match is found
121    if (value == (*sPtr)->data) {
122       ListNodePtr tempPtr = *sPtr; // hold onto node being removed
123       *sPtr = (*sPtr)->nextPtr; // de-thread the node
124       free(tempPtr); // free the de-threaded node
125       return value;
126    }
127    else {
128       ListNodePtr previousPtr = *sPtr;
129       ListNodePtr currentPtr = (*sPtr)->nextPtr;
130
131       // loop to find the correct location in the list
132       while (currentPtr != NULL && currentPtr->data != value) {
133          previousPtr = currentPtr; // walk to ...
134          currentPtr = currentPtr->nextPtr; // ... next node
135       }
136
137       // delete node at currentPtr
138       if (currentPtr != NULL) {
139          ListNodePtr tempPtr = currentPtr;
140          previousPtr->nextPtr = currentPtr->nextPtr;
141          free(tempPtr);
142          return value;
143       }
144    }
145
146    return '\0';
147 }
```

```
149 // return 1 if the list is empty, 0 otherwise
150 int isEmpty(ListNodePtr sPtr)
151 {
152    return sPtr == NULL;
153 }

155 // print the list
156 void printList(ListNodePtr currentPtr)
157 {
158    // if list is empty
159    if (isEmpty(currentPtr)) {
160       puts("List is empty.\n");
161    }
162    else {
163       puts("The list is:");
164
165       // while not the end of the list
166       while (currentPtr != NULL) {
167          printf("%c --> ", currentPtr->data);
168          currentPtr = currentPtr->nextPtr;
169       }
170
171       puts("NULL\n");
172    }
173 }
```

```
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
```

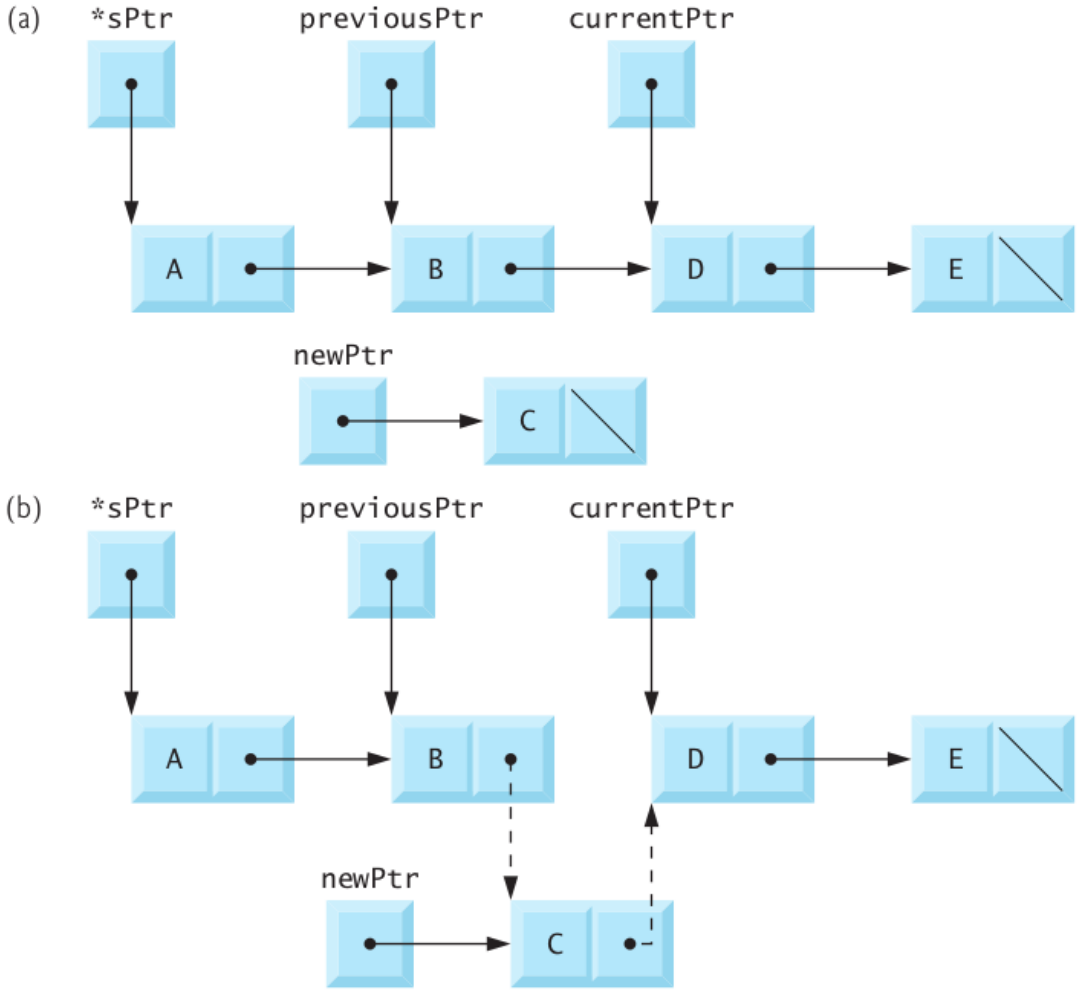


Fig. 12.5 | Inserting a node in order in a list.

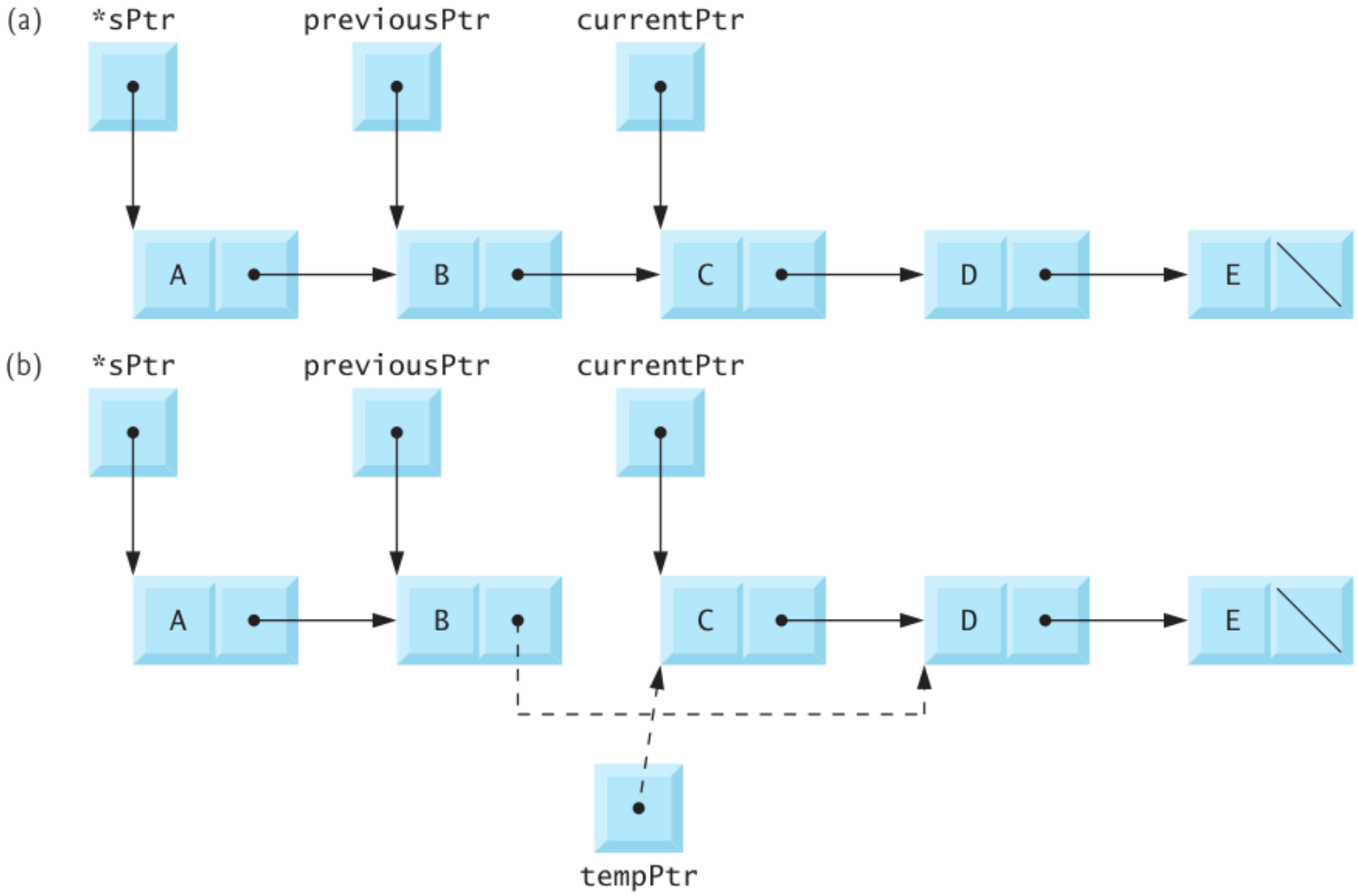


Fig. 12.6 | Deleting a node from a list.

Stack

- A Variant of List which differs in operations
- Insertion in Stack is the same as List
- However deletion can only be done at the tail of the List simulating LIFO (Last In First Out)
- Note that insertion on previous snippets is unusual

```
 6  // self-referential structure
 7  struct stackNode {
 8     int data; // define data as an int
 9     struct stackNode *nextPtr; // stackNode pointer
10  };
11
12  typedef struct stackNode StackNode; // synonym for struct stackNode
13  typedef StackNode *StackNodePtr; // synonym for StackNode*
14
15  // prototypes
16  void push(StackNodePtr *topPtr, int info);
17  int pop(StackNodePtr *topPtr);
18  int isEmpty(StackNodePtr topPtr);
19  void printStack(StackNodePtr currentPtr);
20  void instructions(void);
```

```
75  // insert a node at the stack top
76  void push(StackNodePtr *topPtr, int info)
77  {
78     StackNodePtr newPtr = malloc(sizeof(StackNode));
79
80     // insert the node at stack top
81     if (newPtr != NULL) {
82        newPtr->data = info;
83        newPtr->nextPtr = *topPtr;
84        *topPtr = newPtr;
85     }
86     else { // no space available
87        printf("%d not inserted. No memory available.\n", info);
88     }
89  }
```

```
91  // remove a node from the stack top
92  int pop(StackNodePtr *topPtr)
93  {
94     StackNodePtr tempPtr = *topPtr;
95     int popValue = (*topPtr)->data;
96     *topPtr = (*topPtr)->nextPtr;
97     free(tempPtr);
98     return popValue;
99  }
```

```
22  // function main begins program execution
23  int main(void)
24  {
25     StackNodePtr stackPtr = NULL; // points to stack top
26     int value; // int input by user
27
28     instructions(); // display the menu
29     printf("%s", "? ");
30     unsigned int choice; // user's menu choice
31     scanf("%u", &choice);
32
33     // while user does not enter 3
34     while (choice != 3) {
35
36        switch (choice) {
37           // push value onto stack
38           case 1:
39              printf("%s", "Enter an integer: ");
40              scanf("%d", &value);
41              push(&stackPtr, value);
42              printStack(stackPtr);
43              break;
44           // pop value off stack
45           case 2:
46              // if stack is not empty
47              if (!isEmpty(stackPtr)) {
48                 printf("The popped value is %d.\n", pop(&stackPtr));
49              }
50
51              printStack(stackPtr);
52              break;
```

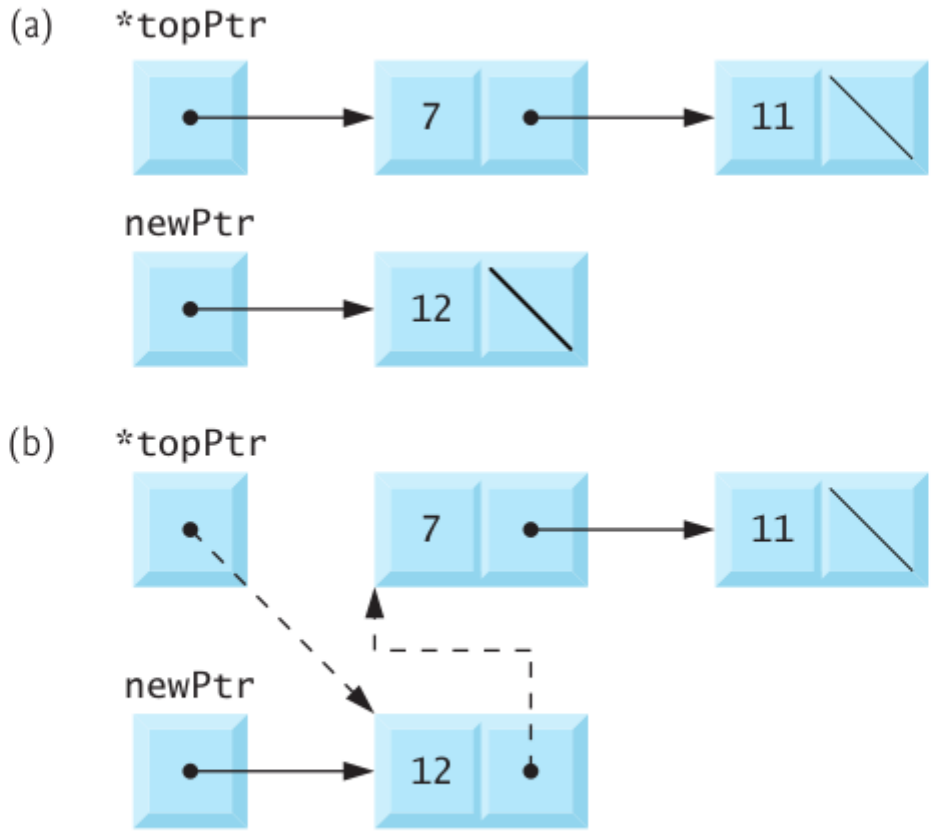


Fig. 12.10 | push operation.

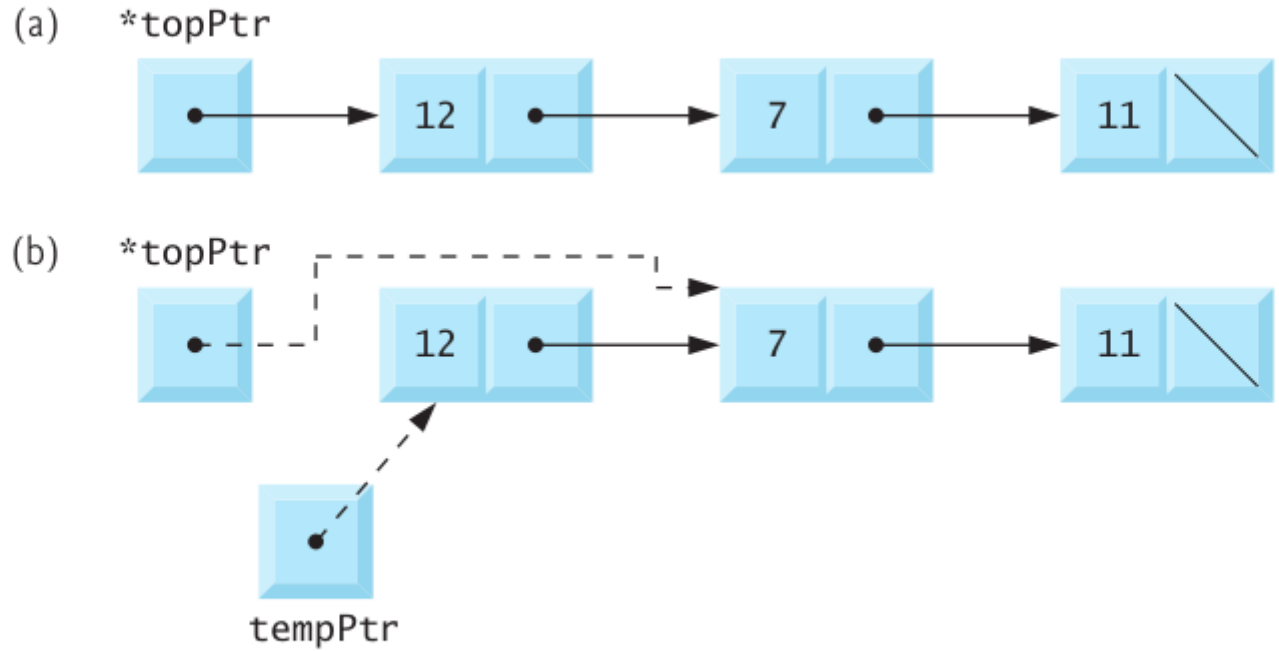
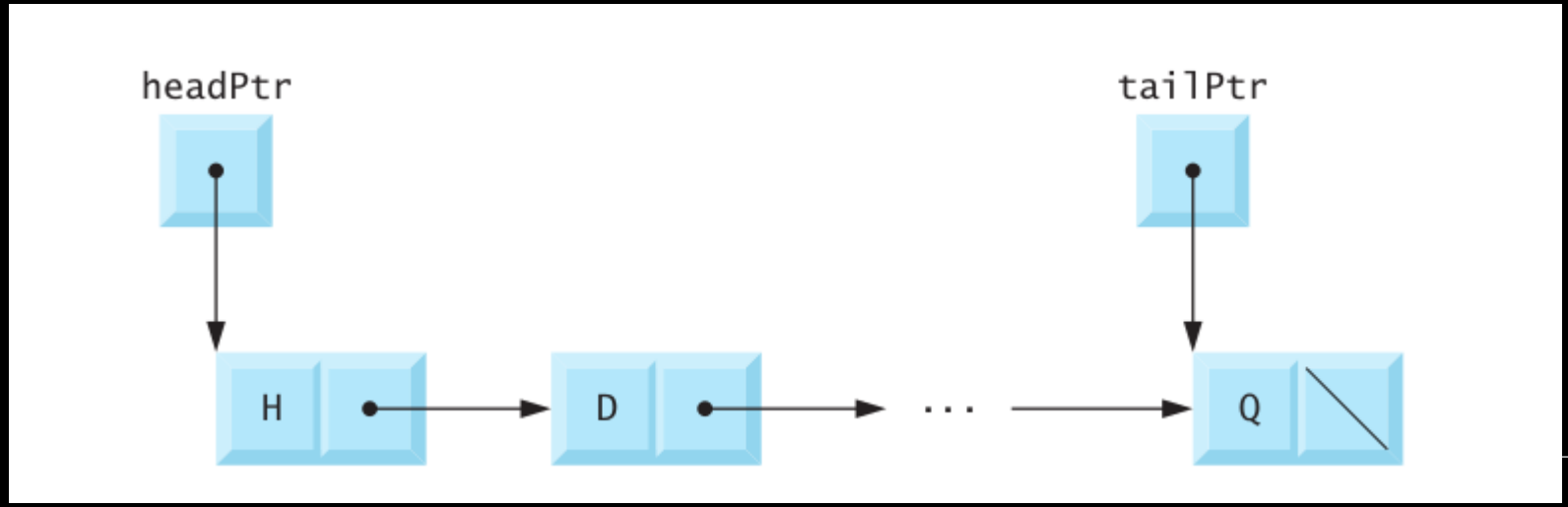


Fig. 12.11 | pop operation.

Queue

- Another variant of List which operates on FIFO, First In First Out
- Since it needs FIFO, there are two pointers to be tracked: head pointer & tail pointer



```c
// Fig. 12.13: fig12_13.c
// Operating and maintaining a queue
#include <stdio.h>
#include <stdlib.h>

// self-referential structure
struct queueNode {
   char data; // define data as a char
   struct queueNode *nextPtr; // queueNode pointer
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

// function prototypes
void printQueue(QueueNodePtr currentPtr);
int isEmpty(QueueNodePtr headPtr);
char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value);
void instructions(void);
```

```
77  // insert a node at queue tail
78  void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value)
79  {
80     QueueNodePtr newPtr = malloc(sizeof(QueueNode));
81
82     if (newPtr != NULL) { // is space available?
83        newPtr->data = value;
84        newPtr->nextPtr = NULL;
85
86        // if empty, insert node at head
87        if (isEmpty(*headPtr)) {
88           *headPtr = newPtr;
89        }
90        else {
91           (*tailPtr)->nextPtr = newPtr;
92        }
93
94        *tailPtr = newPtr;
95     }
96     else {
97        printf("%c not inserted. No memory available.\n", value);
98     }
99  }
```

```
101  // remove node from queue head
102  char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr)
103  {
104     char value = (*headPtr)->data;
105     QueueNodePtr tempPtr = *headPtr;
106     *headPtr = (*headPtr)->nextPtr;
107
108     // if queue is empty
109     if (*headPtr == NULL) {
110        *tailPtr = NULL;
111     }
112
113     free(tempPtr);
114     return value;
115  }
```

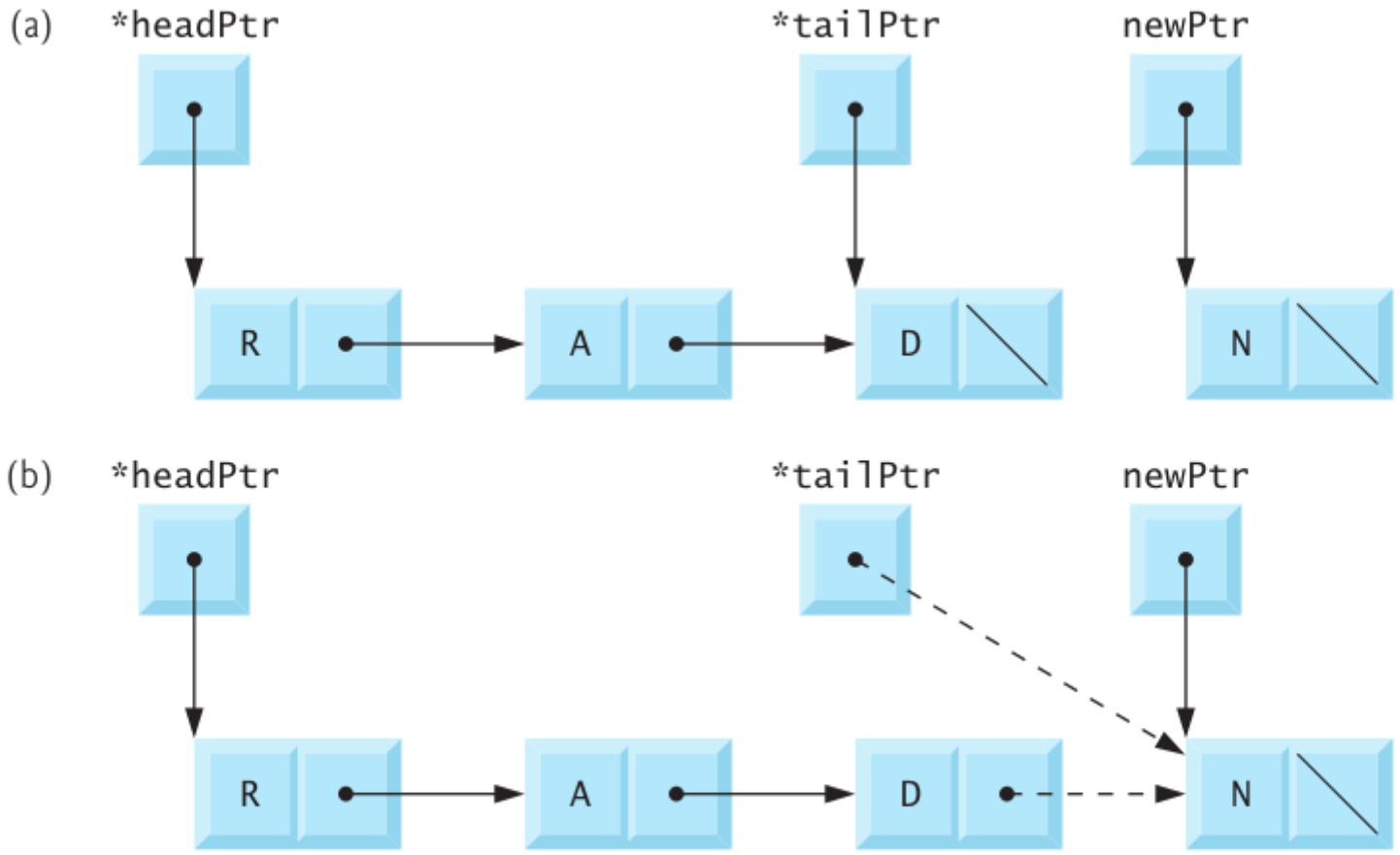
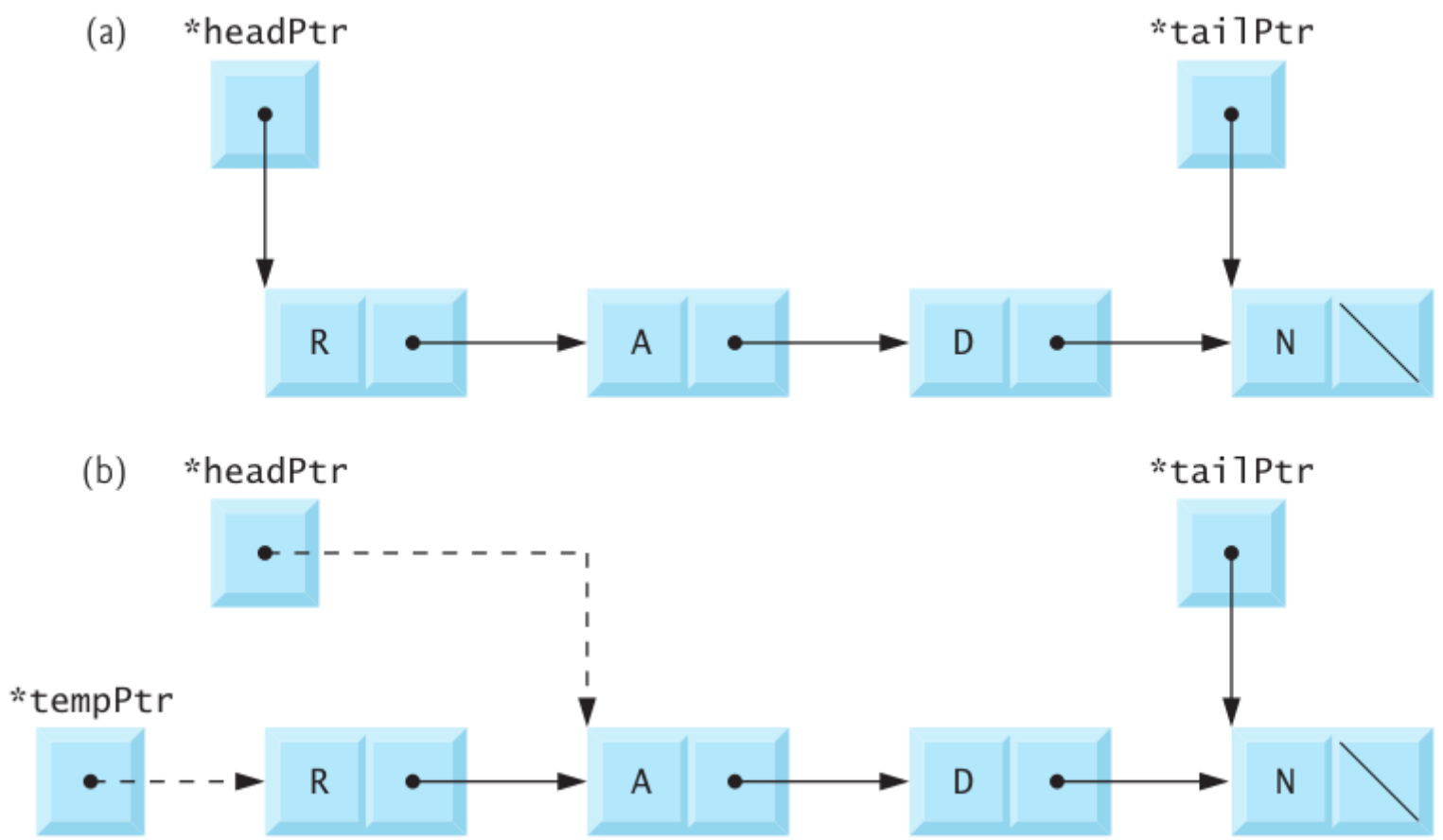
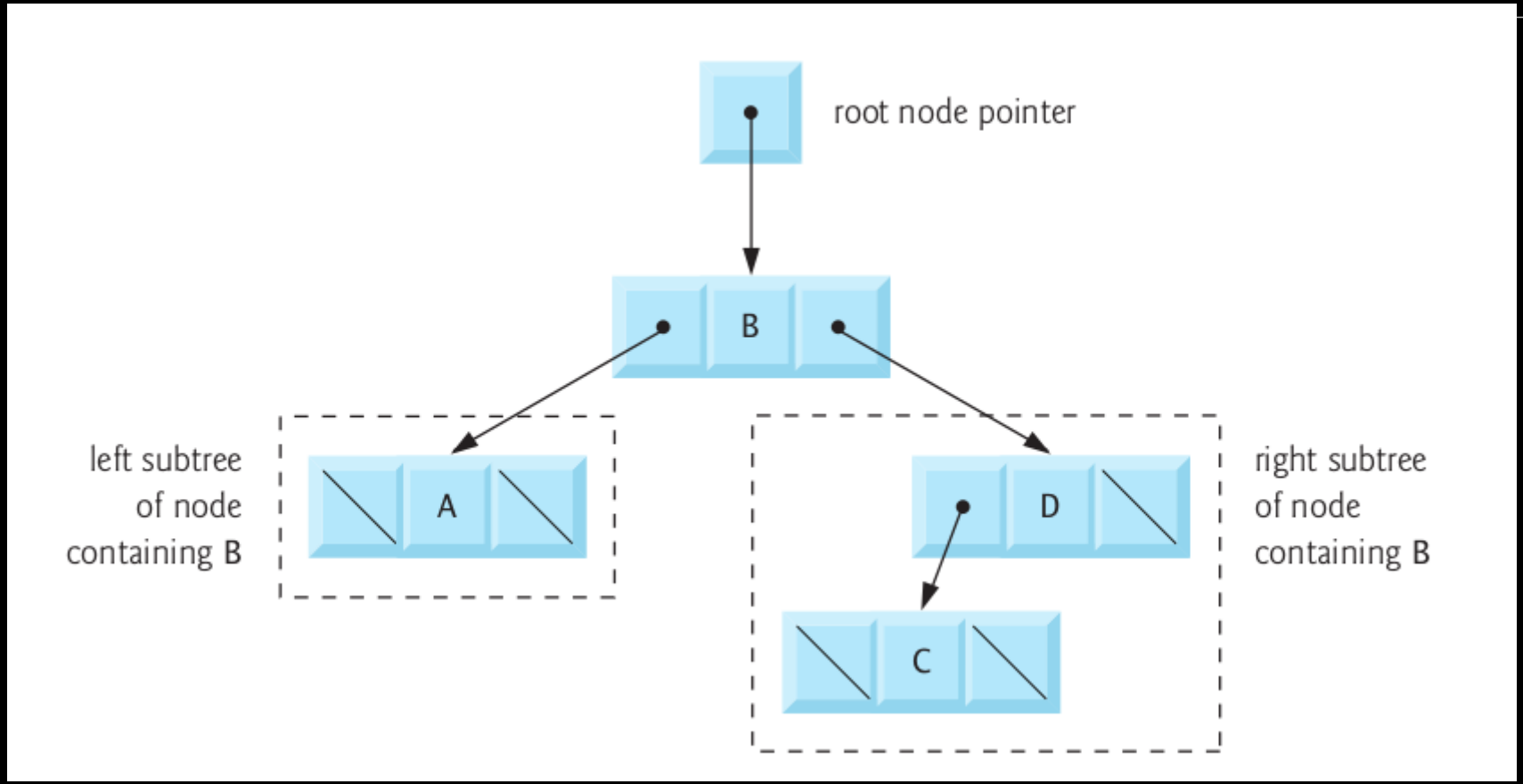


Fig. 12.15 | enqueue operation.


(a)
*headPtr
*tailPtr
R
A
D
N
(b)
*headPtr
*tailPtr
*tempPtr
R
A
D
N

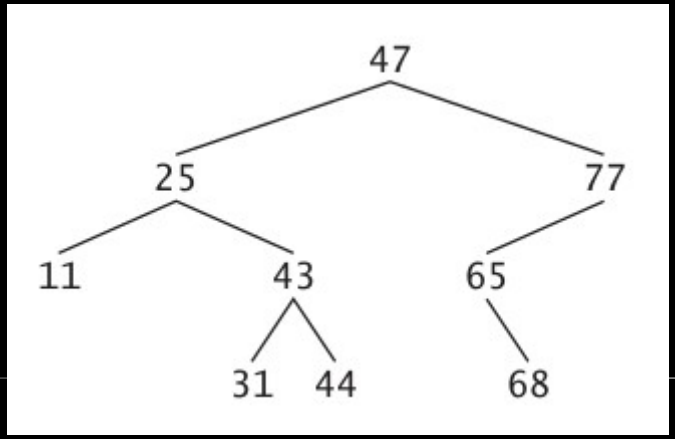
Trees

- A tree is a nonlinear, two-dimensional data structure with special properties
- Binary Tree nodes contain two links
- The root node is the first node in a tree
- Each link in the root node refers to a child
- The children of a node are called siblings
- A node with no children is called a leaf node


root node pointer
B
left subtree
of node
containing B
A
D
C
right subtree
of node
containing B

Binary Search Tree

- Values in any left subtree are less than the value in its parent node
- Values in any right subtree are greater than the value in its parent node



```c
// Fig. 12.19: fig12_19.c
// Creating and traversing a binary tree
// preorder, inorder, and postorder
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// self-referential structure
struct treeNode {
   struct treeNode *leftPtr; // pointer to left subtree
   int data; // node value
   struct treeNode *rightPtr; // pointer to right subtree
};

typedef struct treeNode TreeNode; // synonym for struct treeNode
typedef TreeNode *TreeNodePtr; // synonym for TreeNode*

// prototypes
void insertNode(TreeNodePtr *treePtr, int value);
void inOrder(TreeNodePtr treePtr);
void preOrder(TreeNodePtr treePtr);
void postOrder(TreeNodePtr treePtr);

// function main begins program execution
int main(void)
{
   TreeNodePtr rootPtr = NULL; // tree initially empty

   srand(time(NULL));
   puts("The numbers being placed in the tree are:");

```

```
32      // insert random values between 0 and 14 in the tree
33      for (unsigned int i = 1; i <= 10; ++i) {
34         int item = rand() % 15;
35         printf("%3d", item);
36         insertNode(&rootPtr, item);
37      }
38
39      // traverse the tree preOrder
40      puts("\n\nThe preOrder traversal is:");
41      preOrder(rootPtr);
42
43      // traverse the tree inOrder
44      puts("\n\nThe inOrder traversal is:");
45      inOrder(rootPtr);
46
47      // traverse the tree postOrder
48      puts("\n\nThe postOrder traversal is:");
49      postOrder(rootPtr);
50   }
```

```
52  // insert node into tree
53  void insertNode(TreeNodePtr *treePtr, int value)
54  {
55     // if tree is empty
56     if (*treePtr == NULL) {
57        *treePtr = malloc(sizeof(TreeNode));
58
59        // if memory was allocated, then assign data
60        if (*treePtr != NULL) {
61           (*treePtr)->data = value;
62           (*treePtr)->leftPtr = NULL;
63           (*treePtr)->rightPtr = NULL;
64        }
65        else {
66           printf("%d not inserted. No memory available.\n", value);
67        }
68     }
69     else { // tree is not empty
70        // data to insert is less than data in current node
71        if (value < (*treePtr)->data) {
72           insertNode(&((*treePtr)->leftPtr), value);
73        }
74
75        // data to insert is greater than data in current node
76        else if (value > (*treePtr)->data) {
77           insertNode(&((*treePtr)->rightPtr), value);
78        }
79        else { // duplicate data value ignored
80           printf("%s", "dup");
81        }
82     }
83  }
```

```
85  // begin inorder traversal of tree
86  void inOrder(TreeNodePtr treePtr)
87  {
88     // if tree is not empty, then traverse
89     if (treePtr != NULL) {
90        inOrder(treePtr->leftPtr);
91        printf("%3d", treePtr->data);
92        inOrder(treePtr->rightPtr);
93     }
94  }
95
96  // begin preorder traversal of tree
97  void preOrder(TreeNodePtr treePtr)
98  {
99     // if tree is not empty, then traverse
100    if (treePtr != NULL) {
101       printf("%3d", treePtr->data);
102       preOrder(treePtr->leftPtr);
103       preOrder(treePtr->rightPtr);
104    }
105 }
106
107 // begin postorder traversal of tree
108 void postOrder(TreeNodePtr treePtr)
109 {
110    // if tree is not empty, then traverse
111    if (treePtr != NULL) {
112       postOrder(treePtr->leftPtr);
113       postOrder(treePtr->rightPtr);
114       printf("%3d", treePtr->data);
115    }
116 }
```

```
The numbers being placed in the tree are:
  6  7  4 12  7dup  2  2dup  5  7dup 11

The preOrder traversal is:
  6  4  2  5  7 12 11

The inOrder traversal is:
  2  4  5  6  7 11 12

The postOrder traversal is:
  2  5  4 11 12  7  6
```

Fig. 12.20 | Sample output from the program of Fig. 12.19.