Introduction to Data Structures

Systems Programming

Intro to Data Structures

- Self-referential Structures
- Dynamic Memory Allocation
- A Simple malloc Example
- Linear Lists
- Linked Lists
- Insertion Example
- Linked List Example

Self-Referential Structures

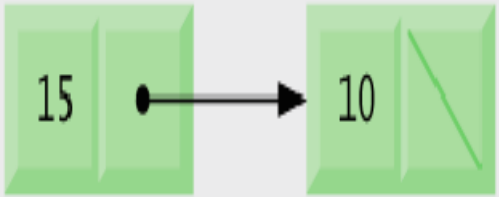
- Self-referential structures contain a pointer member that points to a structure of the same structure type.

Example:

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

- nextPtr
  - is a pointer member that points to a structure of the same type as the one being declared.
  - is referred to as a link. Links can tie one node to another node.
- Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees.

Fig. 12.1 Self-referential structures linked together



Terminated with a NULL pointer (0)

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

12.3 Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires **dynamic memory allocation,** namely, the ability obtain and release memory at execution time.
- In C, the functions **malloc** and **free** and the operator **sizeof** are essential to dynamic memory allocation.

malloc

- malloc
  - Takes as its argument the number of bytes to allocate.
    - Thus, sizeof is used to determine the size of an object.
  - Returns a pointer of type void *
    - A void * pointer may be assigned to any pointer.
    - If no memory is available, malloc returns NULL.

Example

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

Note – the allocated memory is NOT initialized.

free

- free
  - Deallocates memory allocated by malloc.
  - Takes a pointer as an argument.

Example:

```
free ( newPtr );
```

A Simple malloc Example

```
int main ()
{
  int x = 11;
  int *pptr, *qptr;

  pptr = (int *) malloc(sizeof (int));
  *pptr = 66;
  qptr = pptr;
  printf ("%d %d %d\n", x, *pptr, *qptr);
  x = 77;
  *qptr = x + 11;
  printf ("%d %d %d\n", x, *pptr, *qptr);
  pptr = (int *) malloc(sizeof (int));
  *pptr = 99;
  printf ("%d %d %d\n", x, *pptr, *qptr);
  return 0;
}
```

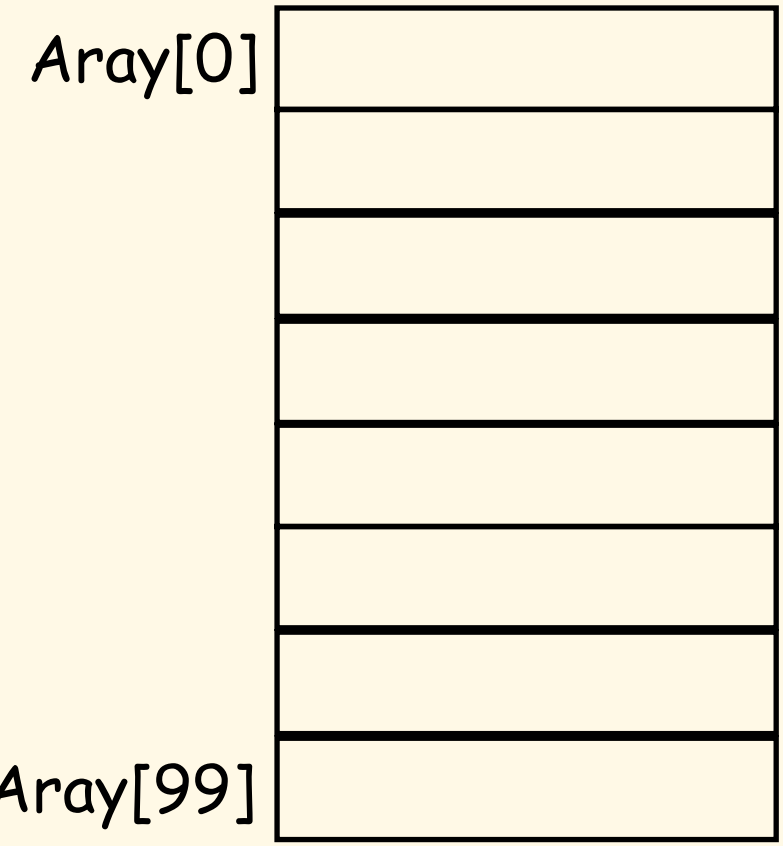
```
$./malloc
11 66 66
77 88 88
77 99 88
```

A Simple free Example

```
int main ()
{
  int x = 11;
  int *pptr, *qptr;
  pptr = (int *) malloc(sizeof (int));
  *pptr = 66;
  qptr = (int *) malloc(sizeof (int));
  *qptr = 55;
  printf ("%d %d %d\n", x, *pptr, *qptr);
  free(pptr);
  x = 77;
  pptr = qptr;
  qptr = (int *) malloc(sizeof (int));
  *qptr = x + 11;
  printf ("%d %d %d\n", x, *pptr, *qptr);
  pptr = (int *) malloc(sizeof (int));
  *pptr = 99;
  printf ("%d %d %d\n", x, *pptr, *qptr);
  free(qptr);
  printf ("%d %d %d\n", x, *pptr, *qptr);
  return 0;
}
```

```
./free
11 66 55
77 55 88
77 99 88
77 99 0
```

Linear Lists



- With a linear list, the assumption is the next element in the data structure is **implicit** in the index.
- This saves space, but is expensive for **insertions!**

Linked Lists

12.4 Linked Lists

- Linked list
  - A 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.
  - The 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.

Fig. 12.2 Linked list graphical representation

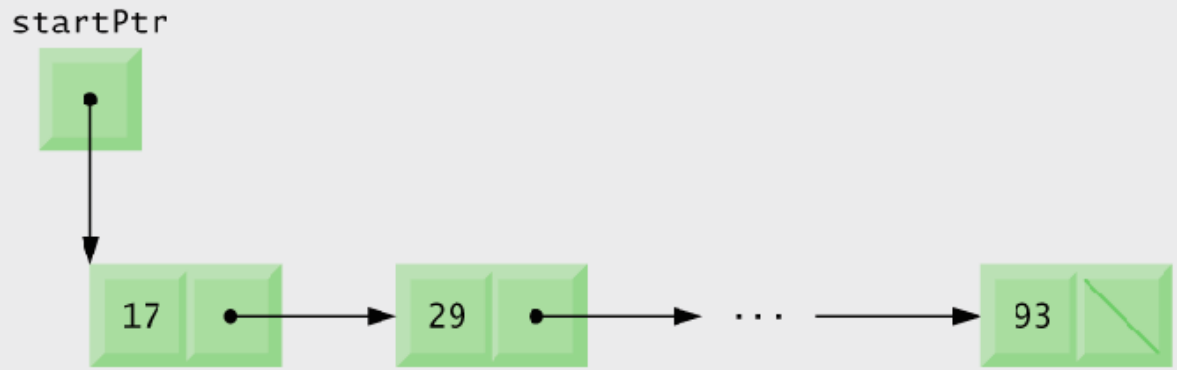


Fig. 12.5 Inserting a node in an ordered list

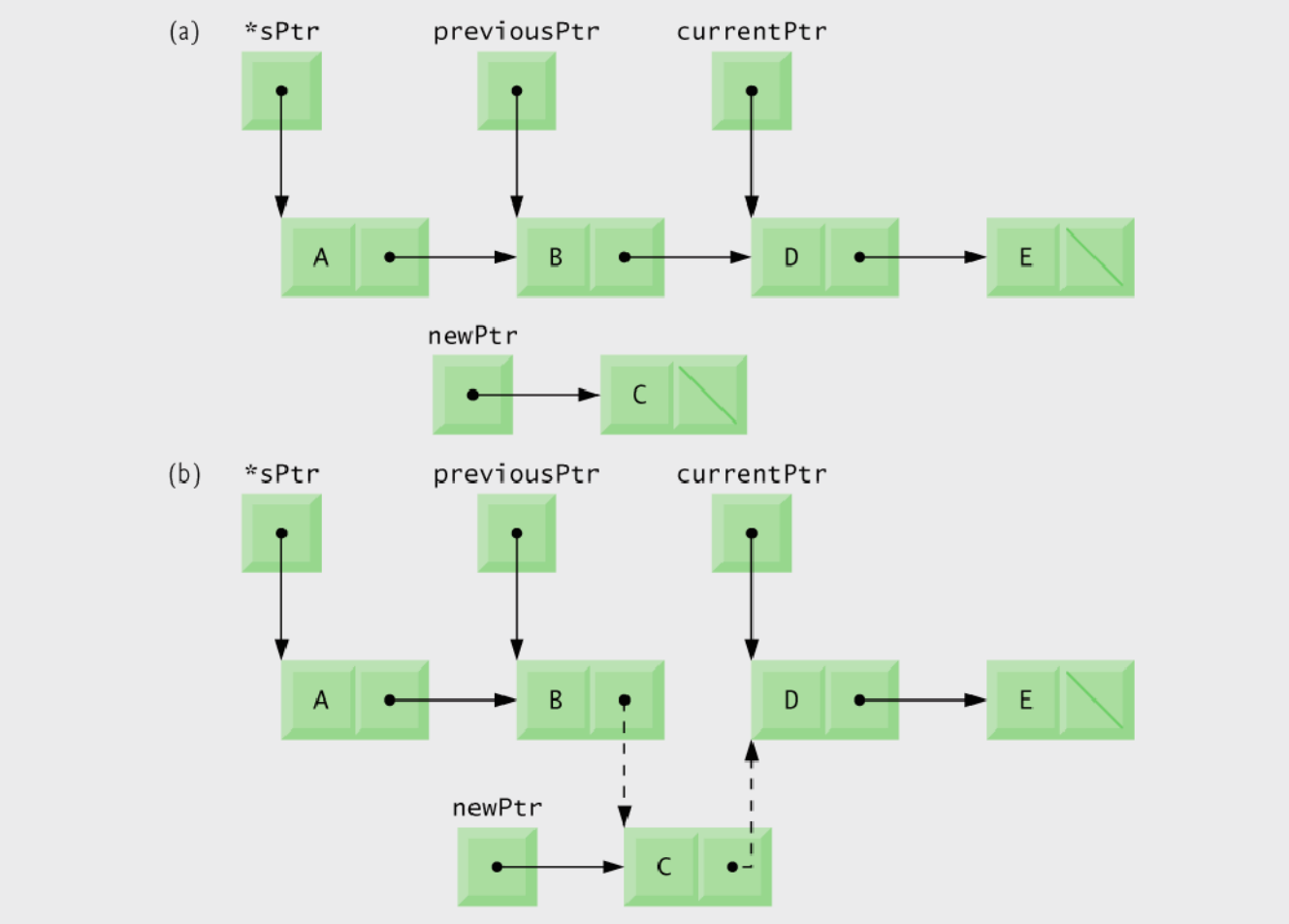
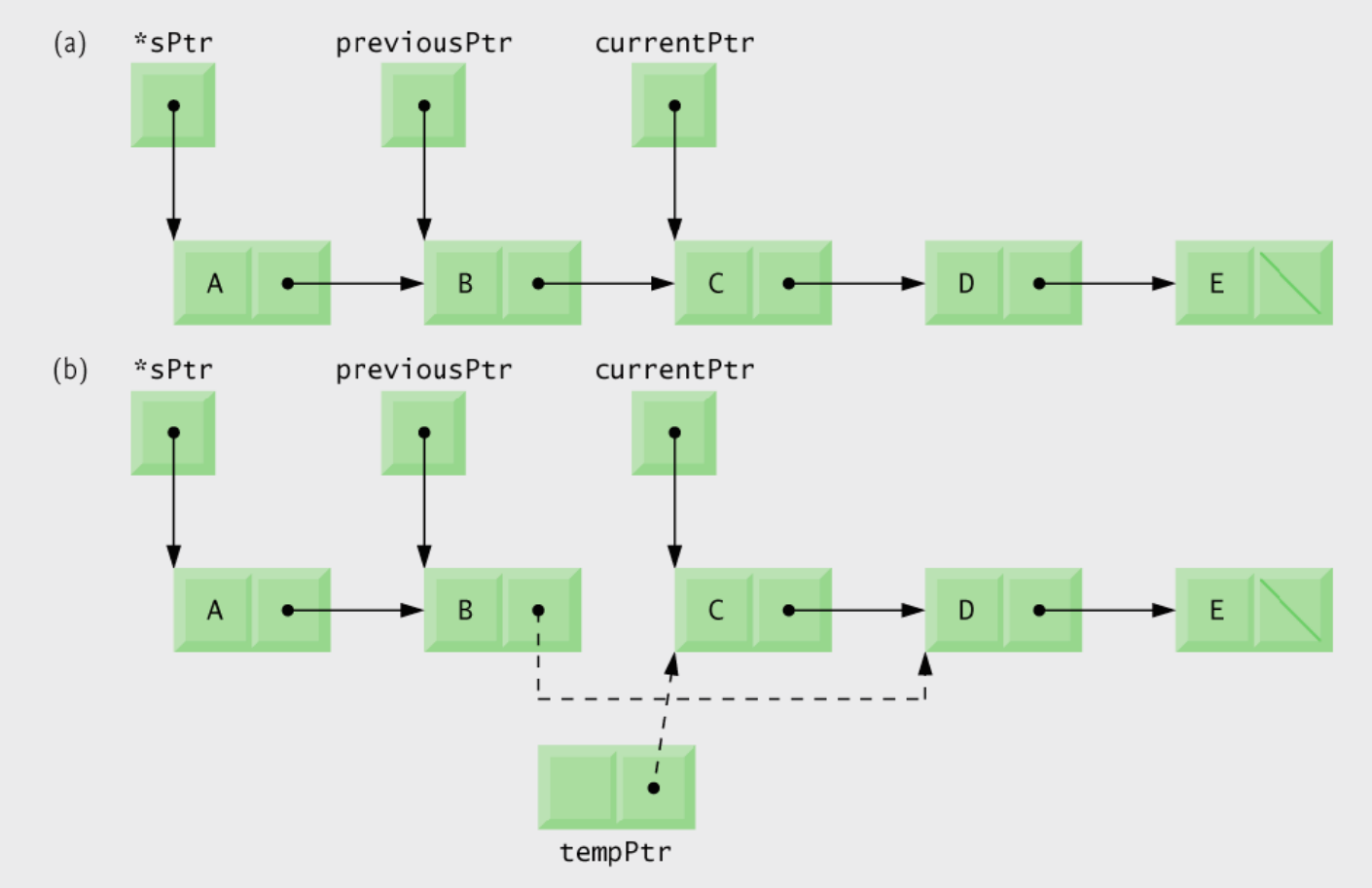


Fig. 12.5 Deleting a node in an ordered list



Linked List Insertion Example

```
/* An Example that uses strings in character
      arrays as part of a linked list          */
#define SIZE 4
#include <stdio.h>
#include <string.h>

typedef struct {
    char name[4];
    struct Node *link;
 } Node;
  typedef Node *Link;

void init (Link *sptr, char cray[])  /* Note this uses a pointer to a pointer */
{
  Link nptr;
  nptr = malloc(sizeof(Node));

  if (nptr != NULL)
  {
   strcpy(nptr->name, cray);
   nptr->link = *sptr;
   *sptr = nptr;
  }
}
```

Linked List Insertion Example

```
/* Insert a new value into the
          list in sorted order */
void insert( Link *sPtr, char *cray )
{
   Link  newPtr;       /* new node */
   Link  previousPtr; /* previous node */
   Link  currentPtr;  /* current node */

   /* dynamically allocate memory */
   newPtr = malloc( sizeof( Node ) );

   /* if newPtr does not equal NULL */
   if ( newPtr ) {
      strcpy(newPtr->name, cray);
      newPtr->link = NULL;

      previousPtr = NULL;
  /* set currentPtr to start of list */

      currentPtr = *sPtr;
```

```
/* loop to find correct location in list */
  while ( currentPtr != NULL && strcmp(cray,
   currentPtr->name) > 0) {
         previousPtr = currentPtr;
         currentPtr = currentPtr->link;
   } /* end while */
 /* insert at beginning of list */
      if ( previousPtr == NULL ) {
         newPtr->link = *sPtr;
         *sPtr = newPtr;
      } /* end if */
      else { /* insert node between previousPtr
                and currentPtr */
        previousPtr->link = newPtr;
        newPtr->link = currentPtr;
      } /* end else */

} /* end if */
  else {
      printf( "%s not inserted. No memory
   available.\n", cray );
} /* end else */

} /* end function insert */
```

Linked List Insertion Example

```
/* Print the list */
void printList( Link currentPtr ) /* Note here just the pointer itself is passed */
{
   /* if list is empty */
   if ( !currentPtr ) {
      printf( "List is empty.\n\n" );
   } /* end if */
   else {

   /* loop while currentPtr does not equal NULL */
    while ( currentPtr ) {
        printf( "%s ", currentPtr->name );
        currentPtr = currentPtr->link;
    } /* end while */

    printf( "*\n" );
   } /* end else */

} /* end function printList */
```

Linked List Insertion Example

```
int main (void)
{
  int i;
  /* Five strings to place in the linked list */
  char b[] = "Bat";
  char c[] = "Cat";
  char h[] = "hat";
  char m[] = "mat";
  char v[] = "vat";
  char n[4];

  char atray[5][4];
  Link lptr = NULL;    /* Set the linked list to empty */
  strcpy (&atray[0][0], v);
  strcpy (&atray[1][0], m);
  strcpy (&atray[2][0], c);
  strcpy (&atray[3][0], b);

  for (i = 0; i < SIZE; i++)
    printf("%s ", &atray[i][0]);
  printf("\n");
```

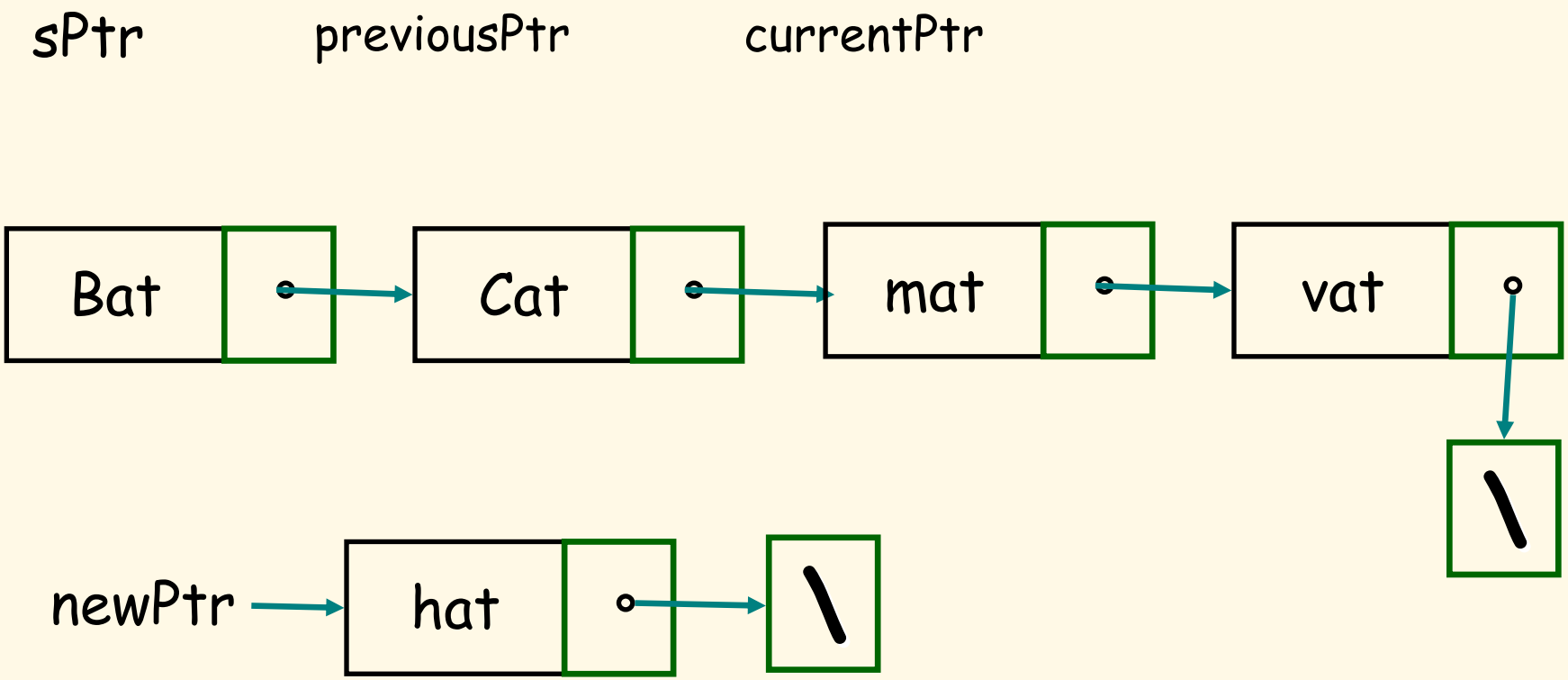
Linked List Insertion Example

```
/* init assumes the linked list is being initialized
    with node elements in reverse alphabetical order */

 for (i = 0; i < SIZE; i++)
   init(&lptr, &atray[i][0]);
 printList (lptr);

 /* insert is the same as in the textbook except it
    compares strings to insert in alphabetical order */
 insert (&lptr, h);
 printList (lptr);
 scanf("%s", n);
 while (strcmp(n, "EOF") != 0)
 {
   insert (&lptr, n);
   printList (lptr);
   scanf("%s", n);
 }
 return;
}
```

Linked List Insertion Example



Linked List Implementation

```
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

Linked List Implementation

```
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 */
```

Function **insert** inserts data into the list

Function **delete** removes data from the list

Linked List Implementation

```
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
```

Linked List Implementation

```
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 */
```

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

Linked List Implementation

```
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

Linked List Implementation

```
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 */
```

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

Linked List Implementation

```
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

Linked List Implementation

```
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 */
```

Output from Linked List Implementation

```
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…)

Output from Linked List Implementation

(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.
```

Summary/Review

- Introduced malloc and free.
- Discussed the tradeoffs between a linear list and a linked list.
- Provided two linked examples.
- Explained event lists as one instance of linked lists.
- Important operations on linked lists:
  - Insertion in the list.
  - Taking a random node out of the list.
  - Taking the 'next' node off the front of the list.
  - Printing the linked list in order

Review of Data Structures Seen

- Arrays
- Arrays of pointers to strings
- Arrays of structs (note: this is a linear list)
- Linked lists
  - singly-linked lists
  - [not] doubly-linked lists!!