Linked Lists

Linked list

- Linked list
  an ordered collection of data in which each element contains the location of the next element.
- Each element contains two parts: data and link.
- The link contains a pointer (an address) that identifies the next element in the list.

- Singly linked list
- The link in the last element contains a null pointer, indicating the end of the list.

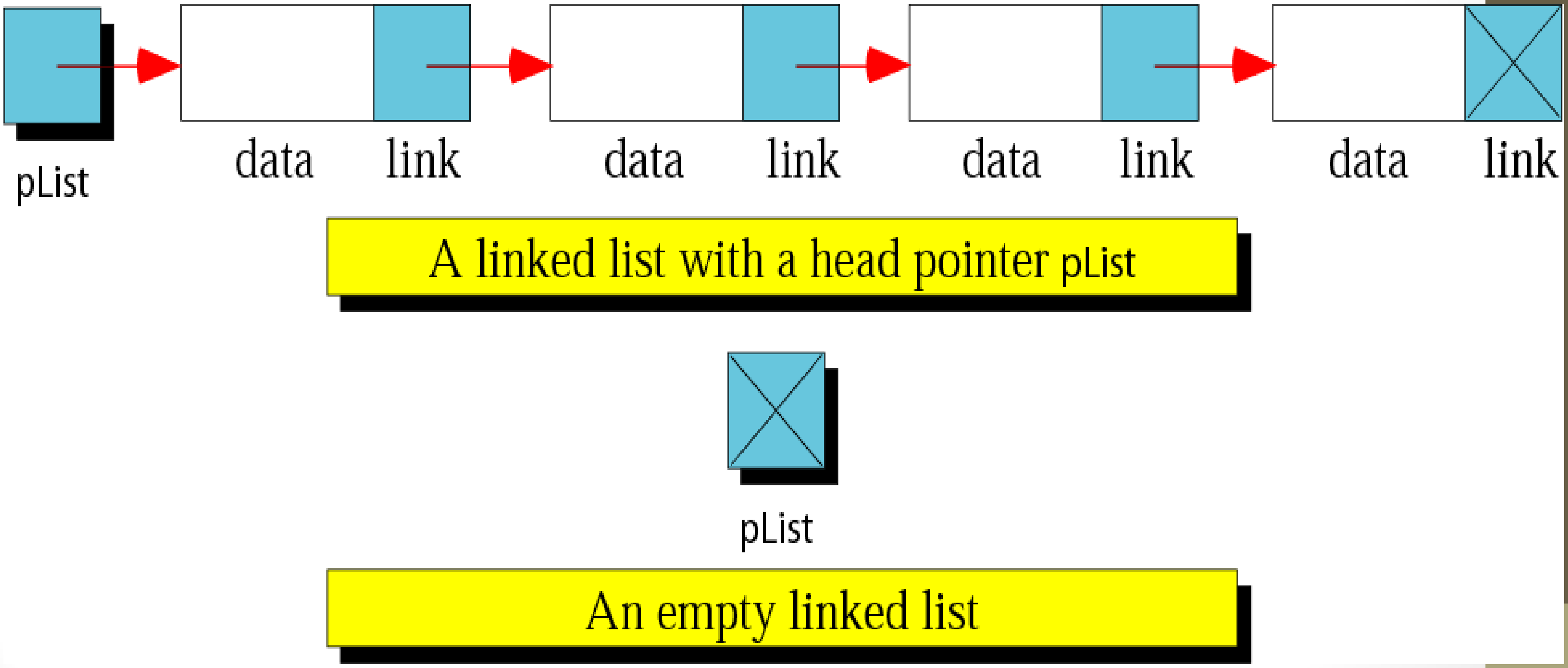
Types of linked lists:

- Singly linked list
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction
- Circular, singly linked
  - Pointer in the last node points back to the first node
- Doubly linked list
  - Two “start pointers” – first element and last element
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards
- Circular, doubly linked list
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

- 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

Figure 11-10

Linked lists

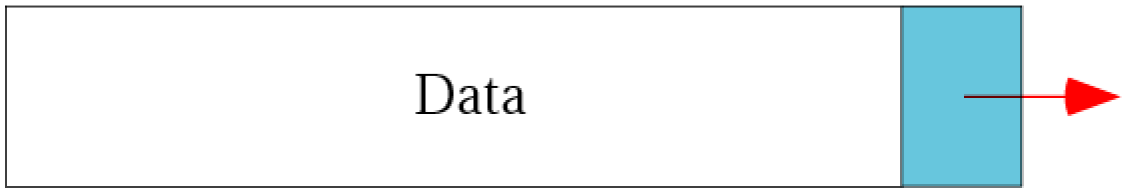


A linked list with a head pointer pList

An empty linked list

Figure 11-11

Node



- Nodes : the elements in a linked list.

- The nodes in a linked list are called self-referential records.
- Each instance of the record contains a pointer to another instance of the same structural type.

Inserting a node

Figure 11-12

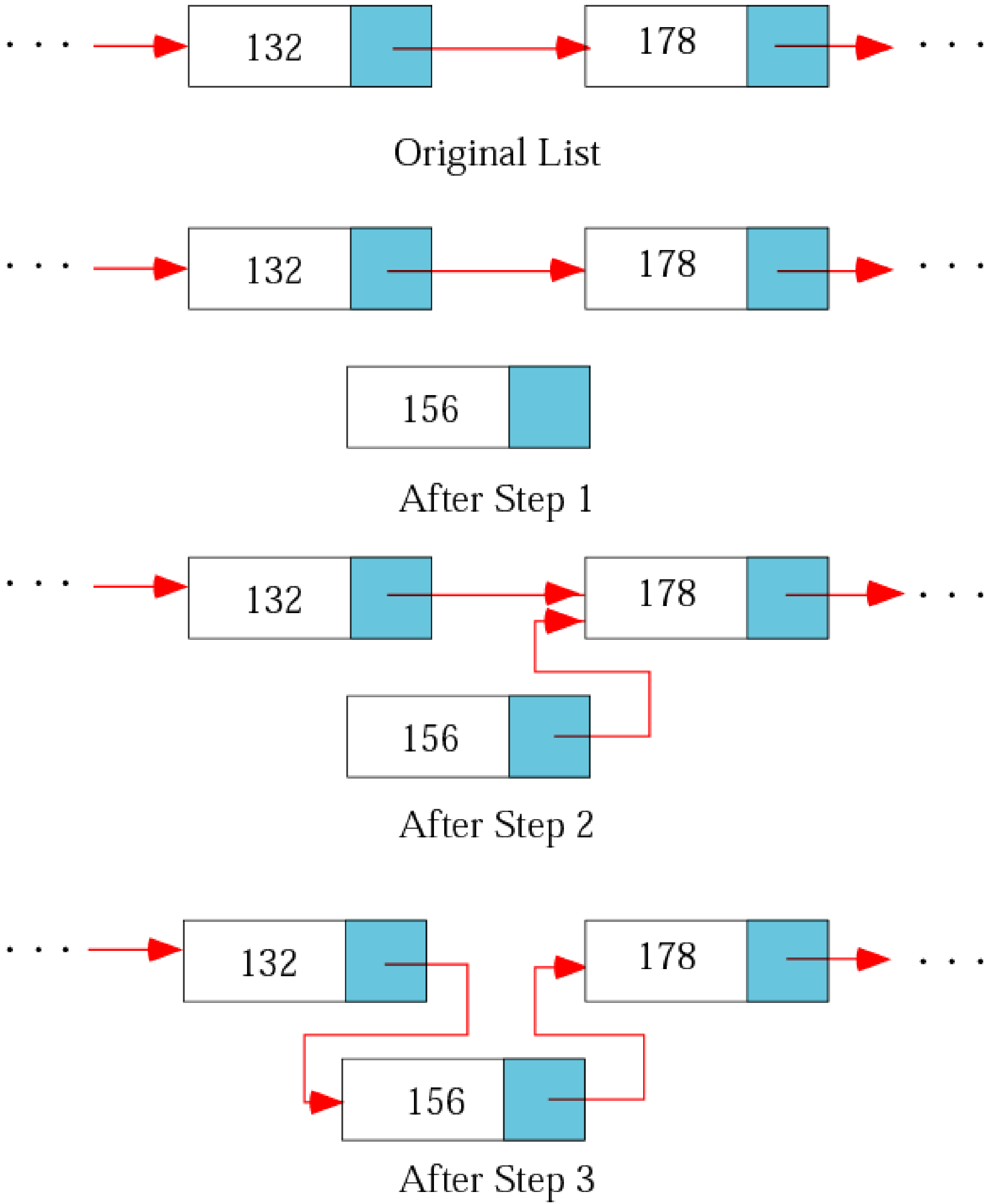
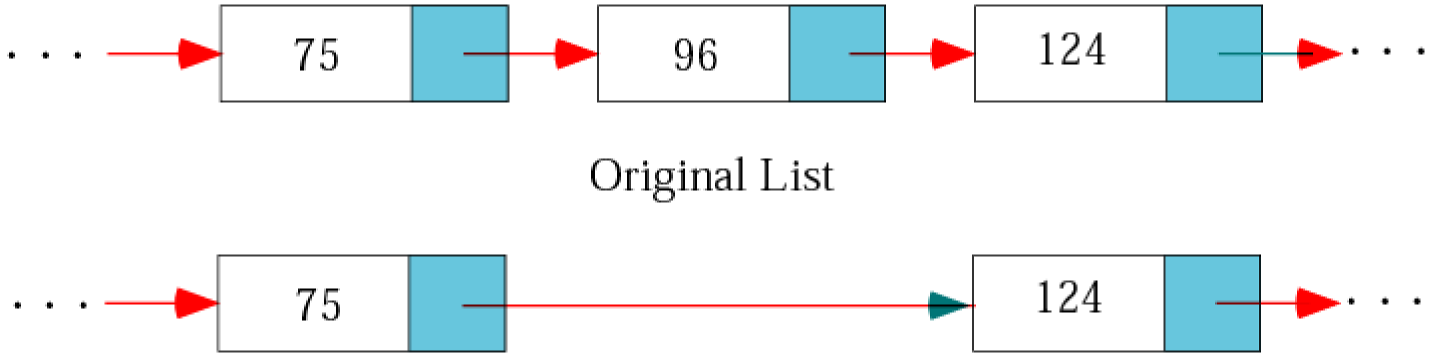


Figure 11-13

Deleting a node



Original List

After Delete

Implementation by using C language

```
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`**

- 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 );`**

```c
/* Fig. 12.3: fig12_03.c
   Operating and maintaining a list */
#include <stdio.h>
#include <stdlib.h>

struct listNode {    /* self-referential structure */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

void insert( ListNodePtr *, char );
char delete( ListNodePtr *, char );
int isEmpty( ListNodePtr );
void printList( ListNodePtr );
void instructions( void );

int main()
{
   ListNodePtr startPtr = NULL;
   int choice;
   char item;

   instructions();  /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );

   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            insert( &startPtr, item );
            printList( startPtr );
            break;
         case 2:
            if ( !isEmpty( startPtr ) ) {
               printf( "Enter character to be deleted: " );
               scanf( "\n%c", &item );

               if ( delete( &startPtr, item ) ) {
                  printf( "%c deleted.\n", item );
                  printList( startPtr );
               }
               else
                  printf( "%c not found.\n\n", item );
            }
            else
               printf( "List is empty.\n\n" );

            break;
         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}

/* Print the instructions */
void instructions( void )
{
   printf( "Enter your choice:\n"
          "   1 to insert an element into the list.\n"
          "   2 to delete an element from the list.\n"
          "   3 to end.\n" );
}

/* Insert a new value into the list in sorted order */
void insert( ListNodePtr *sPtr, char value )
{
   ListNodePtr newPtr, previousPtr, currentPtr;

   newPtr = malloc( sizeof( ListNode ) );

   if ( newPtr != NULL ) {     /* is space available */
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      previousPtr = NULL;
      currentPtr = *sPtr;

      while ( currentPtr != NULL && value > currentPtr->data ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      }

      if ( previousPtr == NULL ) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      }
      else {
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
   else
      printf( "%c not inserted. No memory available.\n", value );
}

/* Delete a list element */
char delete( ListNodePtr *sPtr, char value )
 {
   ListNodePtr previousPtr, currentPtr, tempPtr;

   if ( value == ( *sPtr )->data ) {
      tempPtr = *sPtr;
      *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
      free( tempPtr );             /* free the de-threaded node */
      return value;
   }
   else {
      previousPtr = *sPtr;
      currentPtr = ( *sPtr )->nextPtr;

      while ( currentPtr != NULL && currentPtr->data != value ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      }

      if ( currentPtr != NULL ) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return value;
      }
   }

   return '\0';
 }

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( ListNodePtr sPtr )
 {
   return sPtr == NULL;
 }

 /* Print the list */
void printList( ListNodePtr currentPtr )
 {
   if ( currentPtr == NULL )
```

```
      printf( "List is empty.\n\n" );
   else {
      printf( "The list is:\n" );

      while ( currentPtr != NULL ) {
         printf( "%c --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      }

      printf( "NULL\n\n" );
   }
}
```