Cpt S 122 – Data Structures

Data Structures

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Topics

- Introduction
- Self Referential Structures
- Dynamic Memory Allocation
- Linked List
  - `insert, delete, isEmpty, printList`
- Stack
  - `push, pop`
- Queue
  - `enqueue, dequeue`
- Binary Search Tree
  - `insertNode, inOrder, preOrder, postOrder`

Introduction

- Fixed-size data structures
  - single-subscripted arrays, double-subscripted arrays and `struct`s.
- Dynamic data structures with sizes that grow and shrink at execution time
  - 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.

Introduction

- 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
  - efficient elimination of duplicate data items,
  - representing file system directories and compiling expressions into machine language.

- Each of these data structures has many other interesting applications.

Introduction

- We'll discuss each of the major types of data structures
  - implement programs that create and manipulate them.
- In C++ we'll study data abstraction and abstract data types (ADT).
  - notion of an object (from object-oriented programming) is an attempt to combine abstractions of data and code.
  - ADT is a set of objects together with a set of operations
    - e.g., List, Operations on a list: Insert, delete, search, sort
  - C++ class are perfect for ADTs
- Enable us to build the data structures in a dramatically different manner designed for producing software that's much easier to maintain and reuse.

Self Referential Structures

- A *self-referential structure* contains a pointer member that points to a structure of the *same* structure type.
- For example, the definition
  ```
  struct node {
     int data;
     struct node *nextPtr;
  }; // end struct node
  ```
  defines a type, `struct node`.
- A structure of type `struct node` has two members
  - integer member `data` and pointer member `nextPtr`.

Self Referential Structures (Cont.)

- Member `nextPtr` points to a structure of type `struct node`
  - a structure of the *same* type as the one being declared here, hence the term "*self-referential structure*."
- Member `nextPtr` is referred to as a link
  - link a structure of type `struct node` to another structure of the same type.
- Self-referential structures can be *linked* together to form useful data structures
  - lists, queues, stacks and trees.

Self Referential Structures (Cont.)

- Two self-referential structure objects linked together to form a list.
- A slash represents a NULL pointer
  - placed in the link member of the second self-referential structure
  - indicate that the link does not point to another structure.
- A `NULL` pointer normally indicates the end of a data structure just as the null character indicates the end of a string.

Example: Self Referential Structures

Fig. 12.1 | Self-referential structures linked together.

Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation
  - *obtain more memory space at execution time* to hold new nodes.
  - *release space no longer needed*.

- Functions `malloc` and `free`, and operator `sizeof`, are essential to dynamic memory allocation.

Dynamic Memory Allocation (Cont.)

- Function `malloc` takes as an argument the number of bytes to be allocated
  - returns a pointer of type `void *` (pointer to void) to the allocated memory.
- Function `malloc` is normally used with the `sizeof` operator.
- A `void *` pointer may be assigned to a variable of *any* pointer type.

Dynamic Memory Allocation (Cont.)

- For example, the statement

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

  - evaluates `sizeof(struct node)` to determine the size in bytes of a structure of type `struct node`,
  - *allocates a new area in memory* of that number of bytes and stores a pointer to the allocated memory in variable `newPtr`.
- The allocated memory is *not* initialized.
- If no memory is available, `malloc` returns `NULL`.

Dynamic Memory Allocation (Cont.)

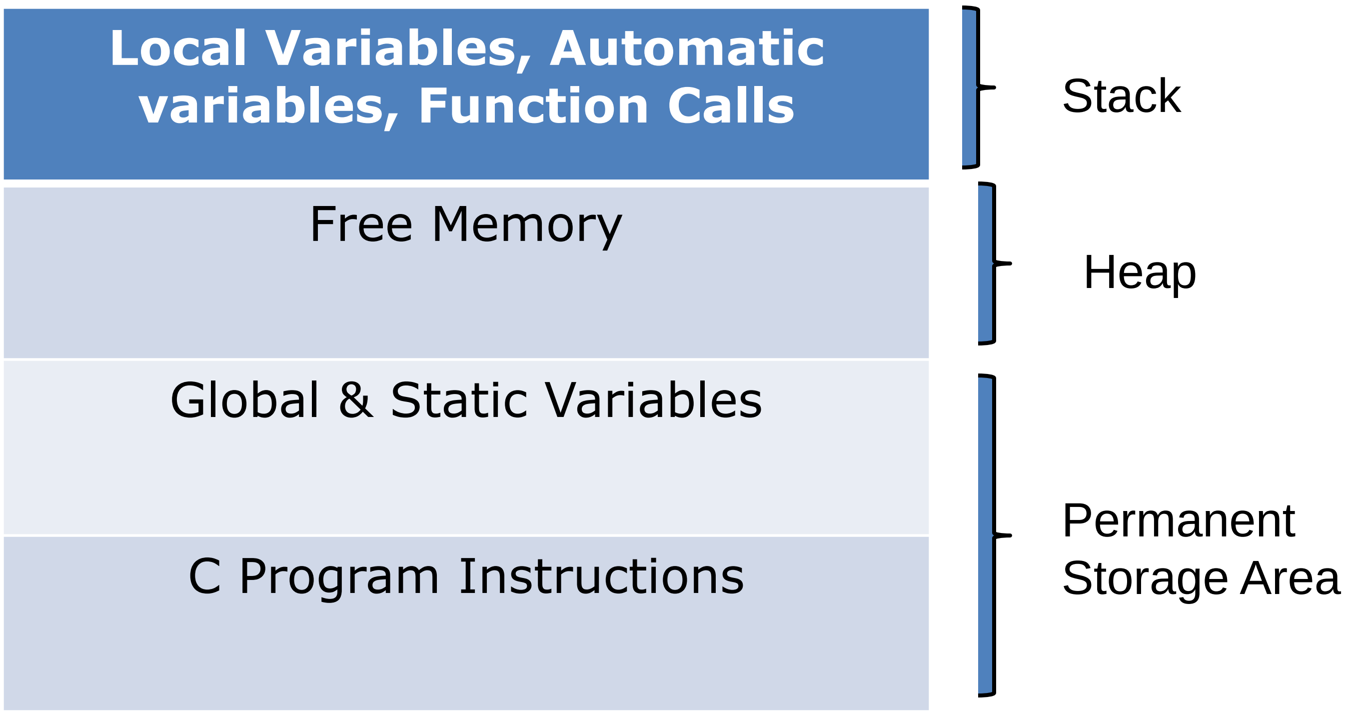
- Function `free` *deallocates* memory
  - the memory is *returned* to the system so that it can be reallocated in the future.
- To *free* memory dynamically allocated by the preceding `malloc` call, use the statement
  - `free( newPtr );`
- C also provides functions `calloc` and `realloc` for creating and modifying *dynamic arrays*.
  - `calloc` allocates multiple blocks of storage, each of the same size.
  - `realloc` changes the already allocated memory size.

Observations

- When using `malloc` test for a `NULL` pointer return value.
- Memory Leak: Not returning dynamically allocated memory when it's no longer needed can cause system to run out of memory prematurely. This is known as "*`memory leak`*".
    - Use `free` to return the memory to system.

Memory Allocation Process

- C programming language manages memory statically, automatically, or dynamically.



Conceptual view of storage of a C program in memory

Linked Lists

Linked Lists

- A linked list is a linear collection of self-referential structures
  - known as nodes, connected by pointer links.
- A linked list is accessed via a pointer to the first node of the list.
  - Subsequent nodes are accessed via the link pointer member stored in each node.
  - The link pointer in the last node of a list is set to `NULL` to mark the end of the list.
- Data is stored in a linked list dynamically
  - each node is created as necessary.

Linked Lists (Cont.)

- A node can contain data of *any* type including other `struct` objects.
- Stacks and queues are also linear data structures,
  - constrained versions of linked lists.
- Trees are *nonlinear* data structures.
- The size of an array created at compile time is fixed.
  - Arrays can become full.
  - Linked lists become full only when the system has *insufficient memory* to satisfy dynamic storage allocation requests.

Linked Lists & Array Comparison

- Lists of data can be stored in arrays, but linked lists provide several advantages.
  - A linked list is appropriate when the number of data elements to be represented in the data structure is *unpredictable*.
  - Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
  - Provide flexibility in allowing the items to be rearranged efficiently.
  - Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list.
  - Insertion & deletion in a sorted array can be time consuming
    - All the elements following the inserted and deleted elements must be shifted appropriately.

Linked Lists & Array Comparison (Cont.)

- Linked-list nodes are normally *not* stored contiguously in memory.
  - Logically, however, the nodes of a linked list *appear* to be contiguous.
- The elements of an array are stored contiguously in memory.
- Linked use more storage than an array with the same number of items.
  - Each item has an additional link field.
- Dynamic overhead incurs the overhead of function calls.

Linked Lists Functions

- The primary functions of linked lists are `insert` and `delete`.
- Function `isEmpty` is called a predicate function
  - It *does not* alter the list in any way.
  - It determines whether the list is empty (i.e., the pointer to the first node of the list is `NULL`).
  - If the list is empty, `1` is returned; otherwise, `0` is returned.
- Function `printList` prints the list.

Linked Lists Example

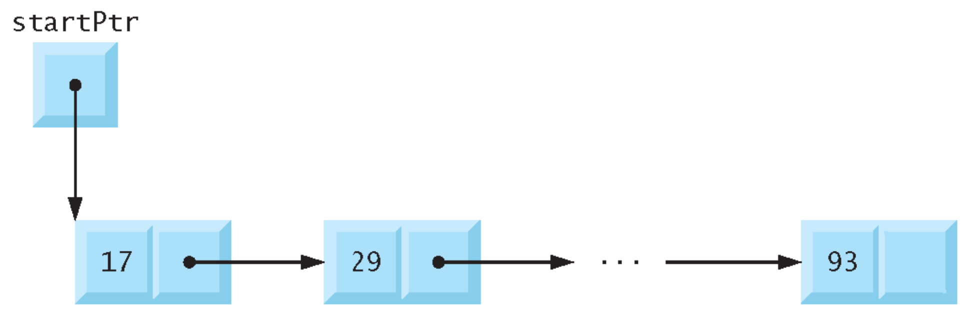


Fig. 12.2 | Linked list graphical representation.

Example of a Pointer to Pointer (Double indirection)

Function prototype	Function description
`double strtod( const char *nPtr, char **endPtr );`	
	Converts the string `nPtr` to `double`.
`long strtol( const char *nPtr, char **endPtr, int base );`	
	Converts the string `nPtr` to `long`.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base );`	
	Converts the string `nPtr` to `unsigned long`.

Fig. 8.5 | String-conversion functions of the general utilities library.

- The function uses the `char **` argument to modify a `char *` in the calling function (`stringPtr`)
- `d = strtod( string, &stringPtr )`
  - indicates that `d` is assigned the double value converted from `string`
  - `stringPtr` is assigned the location of the first character after the converted value in `string`.

Passing Arguments to Functions by Reference

```
void swap(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

void main(void) {

  int x = 3; y = 5;

  printf("x = %d, y = %d\n", x, y);
  swap(&x, &y);
  printf("x = %d, y = %d\n", x, y);

}
```

Linked Lists Example Code

- Manipulates a list of characters.
- insert a character in the list in alphabetical order (function `insert`).
- delete a character from the list (function `delete`).

Linked Lists Operation Examples

```
// 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
}; // end structure listNode

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 );

```

address

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 1 of 8.)

Linked Lists Example (Cont.)

```
int main( void )
{
   ListNodePtr startPtr = NULL; // initially there are no nodes
   unsigned int choice; // user's choice
   char item; // char entered by user

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

   // loop while user does not choose 3
   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:
            printf( "%s", "Enter a character: " );
            scanf( "\n%c", &item );
            insert( &startPtr, item ); // insert item in list
            printList( startPtr );
            break;
         case 2: // delete an element
            // if list is not empty
            if ( !isEmpty( startPtr ) ) {
               printf( "%s", "Enter character to be deleted: " );
               scanf( "\n%c", &item );
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 2 of 8.)

Linked Lists Example (Cont.)

```

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

            break;
         default:
            puts( "Invalid choice.\n" );
            instructions();
            break;
      } // end switch

      printf( "%s", "? " );
      scanf( "%u", &choice );
   } // end while

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 3 of 8.)

Function `Insert`

- Characters are inserted in the list in *alphabetical order*.
- Function `insert` receives the address of the list and a character to be inserted.
- The list's address is necessary when a value is to be inserted at the *start* of the list.
- Providing the address enables the list (i.e., the pointer to the first node of the list) to be *modified* via a call by reference.
- Because the list itself is a pointer (to its first element)
  - passing its address creates a pointer to a pointer (i.e., double indirection).
- This is a complex notion and requires careful programming.

Insert Example

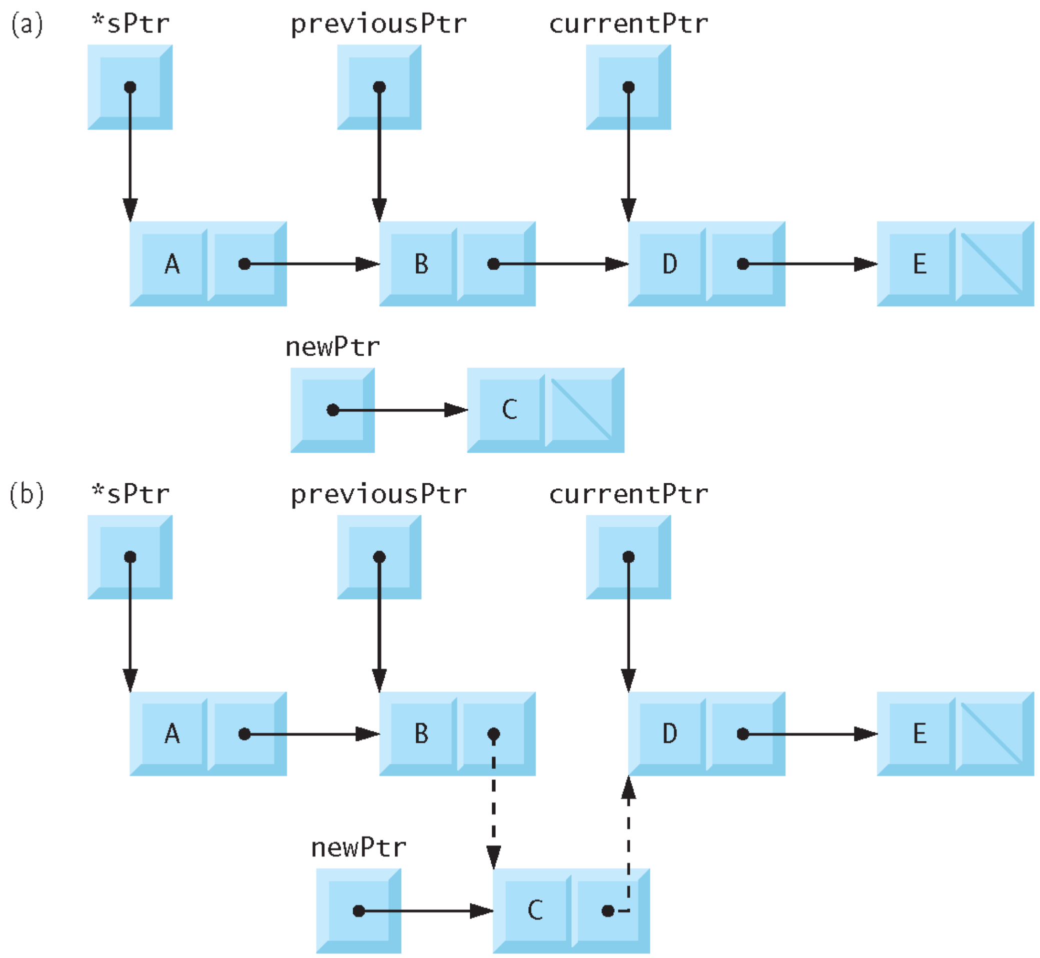


Fig. 12.5 | Inserting a node in order in a list.

Function insert

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

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

```c
   puts( "End of run." );
} // end main

// display program instructions to user
void instructions( void )
{
   puts( "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." );
} // end function instructions

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

   newPtr = malloc( sizeof( ListNode ) ); // create node

   if ( newPtr != NULL ) { // is space available
      newPtr->data = value; // place value in node
      newPtr->nextPtr = NULL; // node does not link to another node

```

insert

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 4 of 8.)

Function insert (Cont.)

```
         previousPtr = NULL;
         currentPtr = *sPtr;

         // loop to find the correct location in the list
         while ( currentPtr != NULL && value > currentPtr->data ) {
            previousPtr = currentPtr; // walk to ...
            currentPtr = currentPtr->nextPtr; // ... next node
         } // end while

         // insert new node at beginning of list
         if ( previousPtr == NULL ) {
            newPtr->nextPtr = *sPtr;
            *sPtr = newPtr;
         } // end if
         else { // insert new node between previousPtr and currentPtr
            previousPtr->nextPtr = newPtr;
            newPtr->nextPtr = currentPtr;
         } // end else
      } // end if
      else {
         printf( "%c not inserted. No memory available.\n", value );
      } // end else
   } // end function insert

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 5 of 8.)

Function `insert` (Cont.)

- The steps for inserting a character in the list are as follows:
  - *Create a node* by calling `malloc`, assigning to `newPtr` the address of the allocated memory. Assigning the character to be inserted to `newPtr->data` . Assigning `NULL` to `newPtr->nextPtr`.
  - Initialize `previousPtr` to `NULL` and `currentPtr` to `*sPtr`, the pointer to the start of the list. Pointers `previousPtr` and `currentPtr` store the locations of the node *preceding* the insertion point and the node *after* the insertion point.
  - While `currentPtr` is not `NULL` and the value to be inserted is greater than `currentPtr->data`, assign `currentPtr` to `previousPtr` and advance `currentPtr` to the next node in the list. This locates the insertion point for the value.

Function `insert` (Cont.)

- If `previousPtr` is `NULL`, //insert at the beginning
  - Insert the new node as the first node in the list.
  - Assign `*sPtr` to `newPtr->nextPtr` (the new node link points to the former first node) and assign `newPtr` to `*sPtr` (`*sPtr` points to the new node).

- Otherwise, if `previousPtr` is not `NULL`, the new node is inserted in place. //insert in the middle
  - Assign `newPtr` to `previousPtr->nextPtr` (the *previous* node points to the new node).
  - Assign `currentPtr` to `newPtr->nextPtr` (the *new* node link points to the *current* node).

delete Example

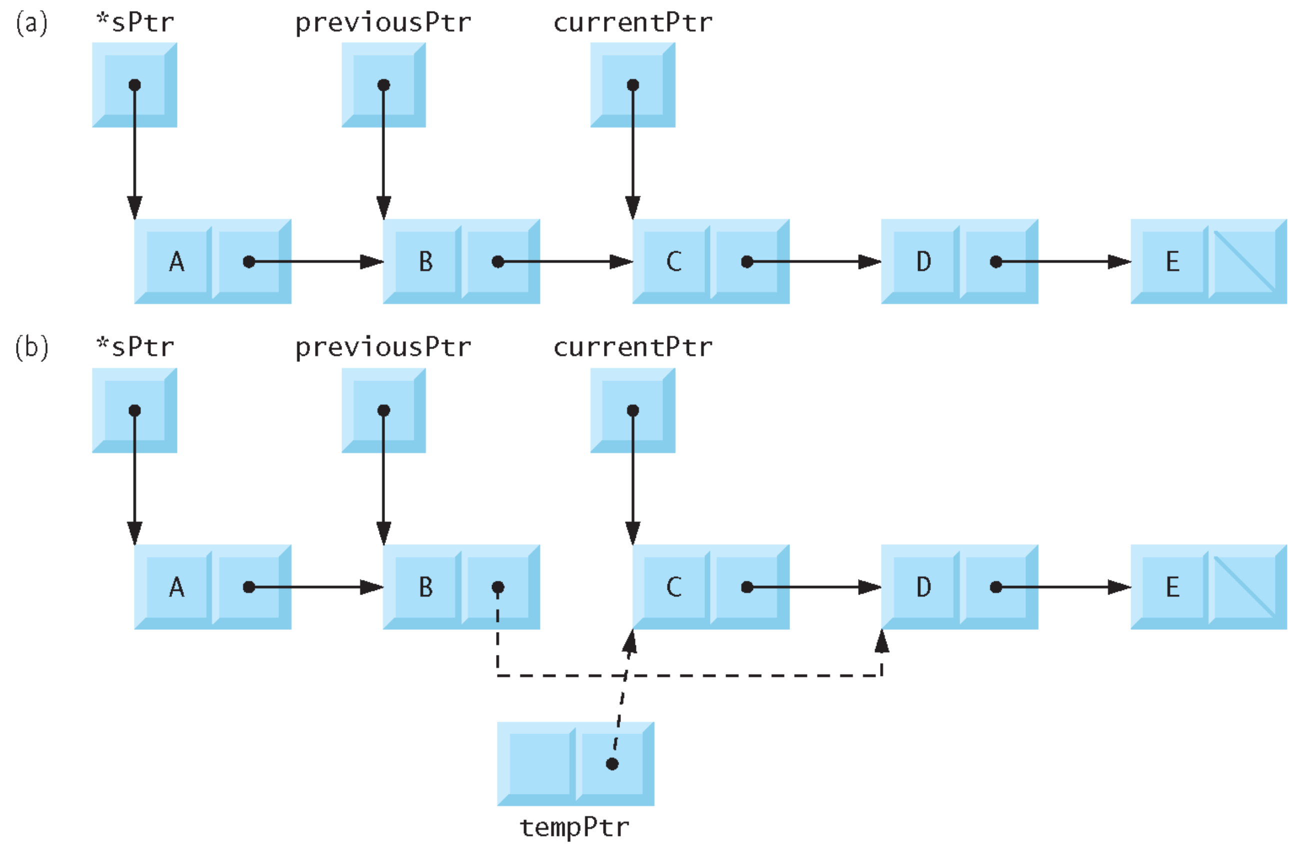


Fig. 12.6 | Deleting a node from a list.

Function delete

```c
// delete a list element
char delete( ListNodePtr *sPtr, char value )
{
   ListNodePtr previousPtr; // pointer to previous node in list
   ListNodePtr currentPtr; // pointer to current node in list
   ListNodePtr tempPtr; // temporary node pointer

   // delete first node
   if ( value == ( *sPtr )->data ) {
      tempPtr = *sPtr; // hold onto node being removed
      *sPtr = ( *sPtr )->nextPtr; // de-thread the node
      free( tempPtr ); // free the de-threaded node
      return value;
   } // end if
   else {
      previousPtr = *sPtr;
      currentPtr = ( *sPtr )->nextPtr;

      // loop to find the correct location in the list
      while ( currentPtr != NULL && currentPtr->data != value ) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      } // end while

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 6 of 8.)

Function delete (Cont.)

```
      // delete node at currentPtr
      if ( currentPtr != NULL ) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return value;
      } // end if
   } // end else

   return '\0';
} // end function delete

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

```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 7 of 8.)

Function `delete`

- Function `delete` receives the address of the pointer to the start of the list and a character to be deleted.
- The steps for deleting a character from the list are as follows:
  - If the character to be deleted matches the character in the first node of the list, assign `*sPtr` to `tempPtr` (`tempPtr` will be used to `free` the unneeded memory), assign `(*sPtr)->nextPtr` to `*sPtr` (`*sPtr` now points to the second node in the list), `free` the memory pointed to by `tempPtr`, and return the character that was deleted.
  - Otherwise, initialize `previousPtr` with `*sPtr` and initialize `currentPtr` with `(*sPtr)->nextPtr` to advance the second node.
  - While `currentPtr` is not `NULL` and the value to be deleted is not equal to `currentPtr->data`, assign `currentPtr` to `previousPtr`, and assign `currentPtr->nextPtr` to `currentPtr`. This locates the character to be deleted if it’s contained in the list.

Function `delete` (Cont.)

- If `currentPtr` is not `NULL`, assign `currentPtr` to `tempPtr`, assign `currentPtr->nextPtr` to `previousPtr->nextPtr`, free the node pointed to by `tempPtr`, and return the character that was deleted from the list .
- If `currentPtr` is `NULL`, return the null character (`'\0'`) to signify that the character to be deleted was not found in the list.

Function printList

```c
// print the list
void printList( ListNodePtr currentPtr )
{
   // if list is empty
   if ( isEmpty( currentPtr ) ) {
      puts( "List is empty.\n" );
   } // end if
   else {
      puts( "The list is:" );

      // while not the end of the list
      while ( currentPtr != NULL ) {
         printf( "%c --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      } // end while

      puts( "NULL\n" );
   } // end else
} // end function printList
```

Fig. 12.3 | Inserting and deleting nodes in a list. (Part 8 of 8.)

Function `printList`

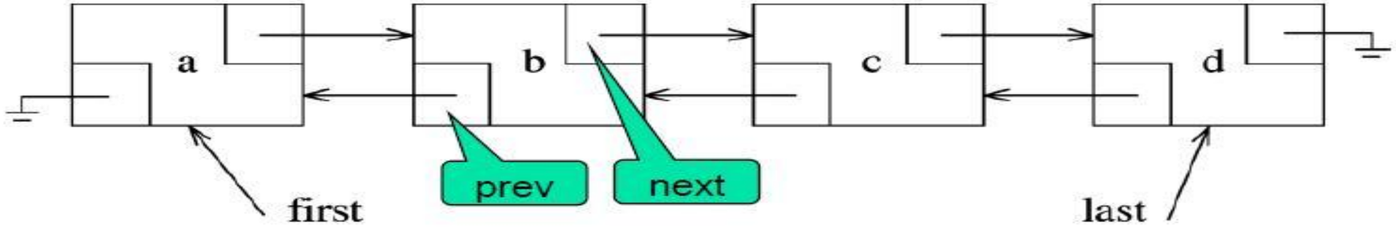
- Function `printList` receives a pointer to the start of the list as an argument and refers to the pointer as `currentPtr`.
- The function first determines whether the list is empty and, if so, prints “`List is empty.`" and terminates.
  - Otherwise, it prints the data in the list.

Function `printList`

- While `currentPtr` is not `NULL`, the value of `currentPtr->data` is printed by the function, and `currentPtr->nextPtr` is assigned to `currentPtr` to advance to the next node.
- The printing algorithm is identical for linked lists, stacks and queues.

Doubly-Linked List (DLL)

- In the linked lists, each node provides information about where is the next node in the list.
  - No knowledge about where the previous node lies in memory.
  - If we are at say 100th node in the list, then to reach the 99th node we have to traverse the list right from the first node.
- To avoid this we can store in each node not only the address of next node but also the address of the previous node in linked list.
  - This arrangement is often known as 'Doubly-Linked List'.



Exercise: (Homework/Programming 3)

- Write a C program to implement the Doubly-Linked List (DLL).
- For example, structure representing a node of the doubly-linked list,

```
struct dnode {
    struct dnode *prevPtr;
    int data;
    struct dnode *nextPtr;
}; // end struct dnode
```

  defines a type, `struct dnode`.
- The `prevPtr` of the first node and `nextPtr` of the last node is set to `NULL`.

Conclusions

- Self Referential Structures
- Dynamic Memory Allocation Function and Process
- Linked List
  - `insert, delete, isEmpty, printList`
- Doubly-Linked List