Lists and Trees

WEEK
1

Linked lists and their associated data types are considered advanced topics in C. Most beginning C books touch lightly on the topic of linked lists and leave it to the programmer to figure them out when they are needed. There are several classes of linked lists. Single-linked lists are the most commonly used. There are also double-linked lists, binary trees, and more. Today you will learn:

- ☐ What linked lists are.
- ☐ How to use single-linked lists.
- ☐ About double-linked lists.
- ☐ How to use stacks and queues.
- ☐ How to use binary trees.

Linked Structures

Linked list is a general classification for several methods of storing information in which the data stored is connected, or linked, together. As you should be able to guess, this linking takes place with the use of pointers.

Expert Tip: Linked lists are not used a great deal. There are instances where linked lists are a perfect solution. On Day 17 and Day 18, you will use both single-linked and double-linked lists, so it is important that you understand the concepts involved in using them.

There are two types of linked lists that are commonly used: single- and double-linked lists. The specific type of linked list is determined by how the data groups are connected. A *single-linked list* has a single connection between each group of information. A *double-linked list* has two connections between each data group.

Note: Single-linked lists are generally referred to as “linked lists” or “linear linked lists.”

Using a Single-Linked List

Single-linked lists, and linked lists in general, are used for several reasons. The main reason is speed. When working with sorted disk files, it can be time consuming to add in a new element. If you add a new element to the beginning of a disk file, each of the following elements must be shifted. With a linked list, you work in memory. Because memory is much faster than disk access, you can manipulate the data in the list faster. In addition, sorted disk files are generally stored in sorted order. With a linked list, pointers are used to keep the elements sorted. If a new element is to be added, it can be placed anywhere. Only the links need to be adjusted. This again increases the speed.

Linked lists involve using a structure that contains a pointer member. The pointer member, however, is special. In the structure

```
struct element {
    int data;
    struct element *next;
};
```

the pointer contains the address of another structure. In fact, it's a pointer to a structure of its own type. The pointer in the structure, called `next` in this case, points to another element structure. This means that each structure, or link, can point to another structure at the same time. Figure 3.1 illustrates a single link using the preceding element structure. Figure 3.2 illustrates using a linked list of such structures.

Figure 3.1. *An element link.*

Figure 3.2. *Links in a linked list.*

Notice that in Figure 3.2, each element points to the next. The last element doesn't point to anything. To help show that the last element doesn't point to an additional link, the pointer is assigned the value of NULL. In C, NULL is equal to zero.

The last link in a single-linked list always points to NULL; however, how do you locate the other links? To prevent the loss of links, you must set up an additional pointer. This pointer is commonly referred to as a head pointer. The *head pointer* always points

to the first element in the link. If you know where the first pointer is, you can access its pointer to the second element. The second element's pointer can then be accessed to get to the third. This can continue until you reach a NULL pointer, which would signify the end of the list. It's possible that the head pointer could be NULL, which would mean the list is empty. Figure 3.3 illustrates the head pointer along with a linked list.

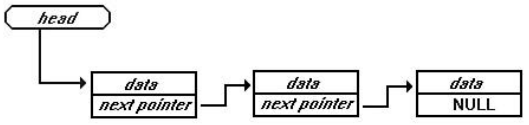


Figure 3.3. *The head pointer.*

Note: The head pointer is a pointer to the first element in a linked list. The head pointer is sometimes referred to the "first element pointer" or "top pointer."

Listing 3.1 presents a program that isn't very practical. It creates a three element linked list. Each of the elements is used by going through the original element. The purpose of this program is to illustrate the relationship between elements. In the following sections, you'll see more practical ways of creating and using linked lists.

Listing 3.1. A first look at a linked list.

```
1:   /* Program:  list0301.c
2:    * Author:   Bradley L. Jones
3:    * Purpose:  Demonstrate the relations in a linked list
4:    * Note:     Program assumes that malloc() is successful.
5:    *           You should not make this assumption!
6:    *=========================================================*/
7:
8:   #include <stdio.h>
9:   #include <stdlib.h>
10:
11:  #define NULL 0
12:
13:  struct list
14:  {
15:    char   ch;
16:    struct list *next;
```

```
17:  };
18:
19:  typedef struct list LIST;
20:
21:  typedef LIST *LISTLINK;
22:
23:  int main( void )
24:  {
25:    LISTLINK  first;            /* same as a head pointer */
26:
27:    first = (LISTLINK) malloc( sizeof(LIST) );
28:
29:    first->ch = 'a';
30:    first->next = (LISTLINK) malloc( sizeof(LIST) );
31:
32:    first->next->ch = 'b';
33:    first->next->next = (LISTLINK) malloc( sizeof(LIST) );
34:
35:    first->next->next->ch = 'c';
36:    first->next->next->next = NULL;
37:
38:    printf("\n\nPrint the character values...");
39:
40:    printf("\n\nValues from the first link:");
41:    printf("\n   ch is %c", first->ch );
42:    printf("\n   next is %d", first->next );
43:
44:    printf("\n\nValues from the second link:");
45:    printf("\n   ch is %c", first->next->ch );
46:    printf("\n   next is %d", first->next->next );
47:
48:    printf("\n\nValues from the third link:");
49:    printf("\n   ch is %c", first->next->next->ch );
50:    printf("\n   next is %d", first->next->next->next );
51:
52:    free( first->next->next );
53:    free( first->next );
54:    free( first );
55:
56:    return(0);
57:  }
```

Output

```
Print the character values...

Values from the first link:
   ch is a
   next is 1618

Values from the second link:
   ch is b
   next is 1626
```

```
Values from the third link:
   ch is c
   next is 0
```

Analysis As stated before, Listing 3.1 isn't the most practical listing; however, it demonstrates many important points regarding linked lists. First, in reviewing the listing you should notice that the linked list's structure is declared in lines 13 through 17. In addition, lines 19 and 21 use the `typedef` command to create two constants. The first is `LIST`, which will be a new data type for declaring a structure of type `list`. The second defined constant, in line 21, is a pointer to a `LIST` data type called `LISTLINK`. This data type, `LISTLINK`, will be used to create the links to the different `LIST` elements in the linked list.

The main part of the program actually starts in line 25 where a pointer to the list structure is declared using the `LISTLINK` constant. This pointer, called `first`, will be used to indicate the beginning of the linked list that is being created. Line 27 allocates the first element in the link. Using `malloc()`, enough space is allocated for one `LIST` element. A pointer is returned by `malloc()` and is stored in `first`. Notice that the program doesn't check to ensure that `malloc()` was successful. This is a poor assumption on the program's part. It's a good programming practice to always check the return value of a memory allocation function.

Line 29 assigns a value to the character variable, `ch`, in the structure. If the linked list element contained other data, it could also be filled in at this point. Line 30 contains the pointer called `next`, that links this element with the next element in the list. If this were the only element in the list, the value of NULL, or zero, could be assigned to the `next` pointer as follows:

```
first->next = NULL;
```

Because an additional link is being added to the list, the `next` pointer is used. In this case, another `malloc()` statement is called to allocate memory for the following element of the list. Upon completion of the allocation, line 32 assigns a value of 'b' to the data item, `ch`. Line 33 repeats the process of allocating memory for a third element. Because the third element is the last being assigned, line 36 caps off the linked list by assigning the value of NULL to the `next` pointer.

Lines 40 through 50 print the values of the elements to the screen so that you can observe the output. The values printed for `next` may vary. Lines 52 through 54 release the memory allocated for the elements in reverse order that they were allocated in.

This program accesses each element by starting with the first element in the list. As you can see, this could be impractical if you have a large number of links. This program is only effective for providing an example, but it's impractical for actually using linked lists.

Using a Linked List

Linked lists are used similar to disk files. Elements or links can be added, deleted, or modified. Modifying an element presents no real challenge; however, adding and deleting an element may. As stated earlier, elements in a list are connected with pointers. When a new element is added, the pointers must be adjusted. Where the new element is added affects how pointers are modified. Elements can be added to the beginning, middle, or end of a linked list. In addition, if the element is the first to be added to the list, the process is a little different.

Adding the First Link

You'll know you are adding the first element to a linked list if the head pointer is NULL. The head pointer should be changed to point to the new element. In addition, because the element being added is the only element, the "next" pointer should be set to NULL. Figure 3.4 illustrates the final result.

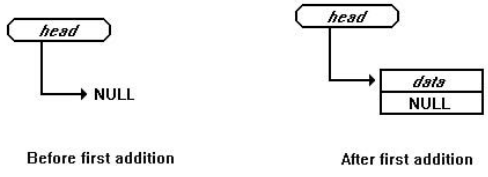


Figure 3.4. *Adding the first element to a linked list.*

The following code fragment includes the element structure defined previously along with two type definitions:

```
struct _element {
    int data;
    struct _element *next;
};

typedef struct _element ELEMENT;
typedef ELEMENT *LINK;
```

The _element structure will be used by means of the two type definitions. When an instance of _element needs to be declared, ELEMENT will be used. ELEMENT is a defined data type for the _element structure. The second defined data type, LINK, is a pointer to an _element structure. These defined constants will be used in later examples. The following code fragment illustrates adding an initial element to a linked list:

```
LINK first = NULL;
LINK new = NULL;

/* enter a new item */
new = (LINK) malloc( sizeof(ELEMENT) );
scanf("%d", &(next->data));

if (first == NULL)
{
    new->next = NULL;
    first = new;
}
```

This fragment starts by including two declarations for pointers to an _element structure using the LINK typedef. Because these are pointer values, they are initialized to NULL. The first LINK pointer, called first, will be used as a head pointer. The second LINK, called new, will contain the link that will be added to the list. The link is created and then data for the link is retrieved. The addition of the new element to the list occurs in the last five lines. If the first pointer—which is the head pointer—is equal to NULL, then you know the list is empty. You can set the pointer in the new element to NULL because there isn't another one to point to. You can then set the head pointer, first, to the new element. At this point, the element is linked in.

Notice that malloc() is used to allocate the memory for the new element. As each new element is added, only the memory needed for it is allocated. The calloc() function could also be used. You should be aware of the difference between these two functions. The main difference is that calloc() will clear out the new element; the malloc() function won't.

Warning: The malloc() in the preceding code fragment didn't ensure that the memory was allocated. You should always check the return value of a memory allocation function.

Tip: When possible, initialize pointers to NULL when you declare them.

Adding to the Beginning

Adding an element to the beginning of a linked list is similar to adding an element to a new list. When an element is added to the beginning, two steps are involved. First, the "next" pointer of the new element must be set to the original first element of the list. This can be done by setting the new element's "next" pointer equal to the head pointer. Once this is done, the head pointer must be reset to point to the new element that now begins the list. Figure 3.5 illustrates this process.

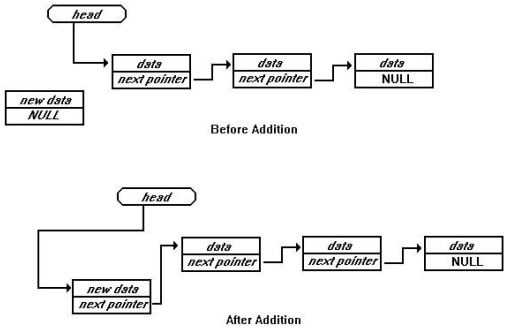


Figure 3.5. *Adding an element to the beginning of a linked list.*

Again using the `element` structure, the following code fragment illustrates the process of adding an element to the beginning of a linked list:

```
/* adding an element to the beginning of a list */
{
    new->next = first;
    first = new;
}
```

The `next` pointer in the `new` element is set to point to the value of the head pointer, `first`. Once this is set, the head pointer is reset to point to the `new` element.

Warning: It is important to take these two steps in the correct order. If you reassign the head pointer first, you will lose the list!

Adding to the Middle

Adding an element to the middle of a list is a little more complicated, yet this process is still relatively easy. Once the location for the new element is determined, you'll adjust the pointers on several elements. Figure 3.6 illustrates the process of adding an element to the middle of a linked list.

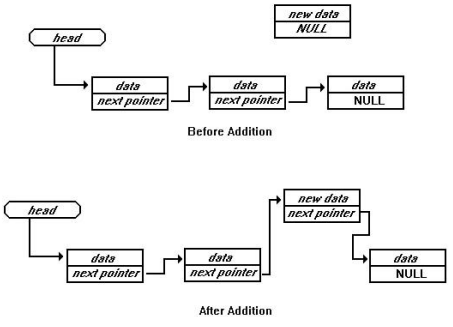


Figure 3.6. *Adding an element to the middle of a linked list.*

As you can see from Figure 3.6, when a new element is added to the middle, two pointers have to be adjusted. The "next" pointer of the previous element has to be adjusted to point to the new element. In addition, the "next" pointer of the new element needs to be set to the original value of the "next" pointer in the previous element. Once these pointers are readjusted, the new element is a part of the list. The following code fragment illustrates this addition:

```
/* adding an element to the middle */
insert_link( LINK prev_link, LINK new_link )
{
    new_link->next = prev_link->next
    prev_link->next = new_link;
}
```

This fragment presents a function that moves the previous link's `next` pointer to the new link's `next` pointer. It then sets the previous link's `next` pointer to point to the new element.

Adding to the End

The final location to which you can add a link is the end of a list. Adding an element to the end is identical to adding a link to the middle. This case is mentioned separately because the value of the previous element's "next" pointer is NULL (or zero). Figure 3.7 illustrates adding an element to the end of a linked list.

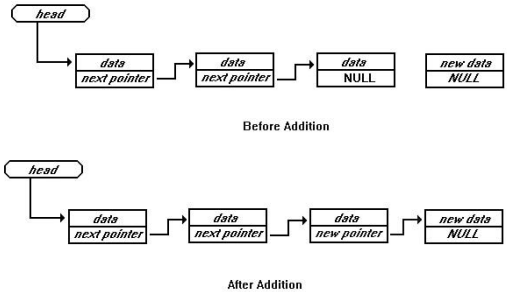


Figure 3.7. *Adding an element to the end of a linked list.*

Implementing a Linked List

Now that you've seen the ways to add links to a list, it's time to see them in action. Listing 3.2 presents a program that uses a linked list to hold a set of characters. The characters are stored in memory by using a linked list.

Type

Listing 3.2. Adding to a linked list of characters.

```
1:  /* Program:  list0302.c
2:   * Author:   Bradley L. Jones
3:   *           Gregory L. Guntle
4:   * Purpose:  Inserts a char in a list
5:   *           Used on a single link list
6:   *=========================================================*/
7:
8:  #include <stdio.h>
9:  #include <stdlib.h>
10:
11:
12: #ifndef NULL
13: #define NULL 0
14: #endif
15:
16: struct list
17: {
```

continues

Listing 3.2. continued

```
18:    char   ch;
19:    struct list *next_rec;
20:  };
21:
22:  typedef struct list LIST;
23:  typedef LIST *LISTPTR;          /* Pointer to the structure list */
24:
25:  LISTPTR add_to_list( char, LISTPTR );   /* Function to add new item
                                              to list */
26:  void show_list(void);
27:  void free_memory_list(void);
28:  void insert_list( char, LISTPTR );
29:
30:  LISTPTR first = NULL;                 /* same as a head pointer */
31:
32:
33:  int main( void )
34:  {
35:    LISTPTR rec_addr;
36:    int i=0;
37:
38:    rec_addr = add_to_list( 'A', (LISTPTR)NULL );  /* Add 1st char */
39:    first = rec_addr;                              /* Start of our
                                                        list */
40:
41:    /*  Build initial list */
42:    printf("Before insertion\n");
43:    while ( i++<5 )
44:       rec_addr = add_to_list( 'A'+i, rec_addr );
45:    show_list();                /* Dumps the entire list - BEFORE */
46:
47:    printf("\n\nAfter insertion\n");
48:    /*  Now insert two chars into current list */
49:    i=0;
50:    rec_addr = first;      /* Start at beginning */
51:    while( i<2 )           /* Travel chain to 3rd position */
52:    {
53:       rec_addr = rec_addr->next_rec;
54:       i++;
55:    }
56:    insert_list( 'Z', rec_addr );
57:    show_list();                /* Dumps the entire list - AFTER */
58:
59:    free_memory_list();                 /* Release all memory */
60:
61:    return(0);
62:  }
63:
64:  /* Function: add_to_list
65:   * Purpose : Inserts new record at end of the list
```

```
66:    *
67:    * Entry   : char ch = character to store
68:    *           LISTPTR prev_rec = address to previous data record
69:    *
70:    * Returns : Address to this new record
71:    *=========================================================*/
72:
73:  LISTPTR add_to_list( char ch, LISTPTR prev_rec )
74:  {
75:     LISTPTR new_rec=NULL;             /* Holds address of new rec */
76:
77:     new_rec = (LISTPTR)malloc(sizeof(LIST)); /* Get memory location */
78:     if (!new_rec)                        /* Unable to get memory */
79:     {
80:        printf("\nUnable to allocate memory!\n");
81:        exit(1);
82:     }
83:
84:     new_rec->ch = ch;        /* Store character into new location */
85:     new_rec->next_rec = NULL;  /* Last record always pts to NULL */
86:
87:     if (prev_rec)              /* If not at first record */
88:       prev_rec->next_rec = new_rec; /* Adjust pointer of previous rec
89:                                        to pt to this new one */
90:     return(new_rec);         /* return address of this new record */
91:  }
92:
93:  /* Function: insert_list
94:   * Purpose : Inserts new record anywhere in list
95:   * Entry   : char ch = character to store
96:   *           LISTPTR prev_rec = address to previous data record
97:   *
98:   * Returns : Address to this new record
99:   *=========================================================*/
100:
101: void insert_list( char ch, LISTPTR prev_link )
102: {
103:    LISTPTR new_rec=NULL;             /* Holds address of new rec */
104:
105:    new_rec = (LISTPTR)malloc(sizeof(LIST)); /* Get memory location */
106:    if (!new_rec)                        /* Unable to get memory */
107:    {
108:       printf("Unable to allocate memory!\n");
109:       exit(1);
110:    }
111:
112:    new_rec->ch = ch;
113:    new_rec->next_rec = prev_link->next_rec;
```

continues

Listing 3.2. continued

```
114:    prev_link->next_rec = new_rec;
115: }
116:
117:
118: /* Function: show_list
119:  * Purpose : Displays the information current in the list
120:  * Entry   : N/A
121:  * Returns : N/A
122:  *=========================================================*/
123:
124: void show_list()
125: {
126:      LISTPTR cur_ptr;
127:      int counter = 1;
128:
129:      printf("Rec addr  Position  Data  Next Rec addr\n");
130:      printf("========  ========  ====  =============\n");
131:      cur_ptr = first;
132:      while (cur_ptr)
133:      {
134:         printf("  %X   ", cur_ptr );   /* Address of this record */
135:         printf("     %2i       %c", counter++, cur_ptr->ch);
136:         printf("      %X   \n",cur_ptr->next_rec);  /* Address of
                                                          next rec */
137:         cur_ptr = cur_ptr->next_rec;
138:      }
139: }
140:
141: /* Function: free_memory_list
142:  * Purpose : Frees up all the memory collected for list
143:  * Entry   : N/A
144:  * Returns : N/A
145:  *=========================================================*/
146:
147: void free_memory_list()
148: {
149:      LISTPTR cur_ptr, next_rec;
150:      cur_ptr = first;         /* Start at beginning */
151:      while (cur_ptr)          /* Go until hit end of list = NULL */
152:      {
153:         next_rec = cur_ptr->next_rec; /* Get address of next record */
154:         free(cur_ptr);                /* Free current record */
155:         cur_ptr = next_rec;           /* Adjust current */
156:      }
157: }
```

Output

```
Before insertion
Rec addr  Position  Data  Next Rec addr
========  ========  ====  =============
  654          1       A       65C
  65C          2       B       664
  664          3       C       66C
  66C          4       D       674
  674          5       E       67C
  67C          6       F       0

After insertion
Rec addr  Position  Data  Next Rec addr
========  ========  ====  =============
  654          1       A       65C
  65C          2       B       664
  664          3       C       684
  684          4       Z       66C
  66C          5       D       674
  674          6       E       67C
  67C          7       F       0
```

Analysis

This listing demonstrates several times when a link can be added. In line 37, the first link is added to an empty list. In line 43, a function is called to add a link to the current position at the end of the list. Finally, in line 55, a function is called to add a link at a position in the middle of the list.

This listing uses a structure in lines 15 through 19 that defines the link that will be used. The list will contain a character, `ch`, and a pointer to the next link, `next_rec`. Lines 21 and 22 create type definitions for the link structure called `LIST`. In addition, in line 22, a constant for a pointer to the structure is declared as `LISTPTR`. These constants make the declarations later in the listing easier to follow.

In line 29, the head pointer, `first`, is declared and initialized to NULL. The `first` pointer is declared with the pointer previously defined as `LISTPTR`. The `first` pointer will be used to always point to the beginning of the linked list. Additional variables are declared in `main()`.

Line 37 uses a function called `add_to_list()` to add the first link. This link will contain the character 'A.' Lines 63 through 90 contain the `add_to_list()` function. This function takes the new character to be added and the previous link's address, and it returns the new link's address. Line 76 allocates memory for the new link. Line 77 ensures the allocation was successful. If it wasn't, then an error message is printed and the program exits. Line 83 assigns the new character to the newly allocated link. The

next pointer assigns NULL to the next pointer. If this isn't the first link, the previous link's `next_rec` pointer is assigned to this new link, which effectively places the new link at the end of the list. Line 89 returns the address of the new link. This function adds links in a manner similar to what was shown in Figure 3.7.

Back in `main()`, lines 42 and 43 loop through adding five more links. Line 44 then prints the list to the screen using `show_list()` in lines 117 through 138. The `show_list()` function prints headings and then, starting with the first link, navigates through the list. This is done by accessing the next link through the `next_rec` pointer (line 136).

In lines 48 through 53, the program starts at the beginning of the list and loops through two links. Line 55 then uses the `insert_list()` function to insert a link in the middle of the list. The `insert_list()` function, in lines 92 through 114, declares a temporary link called `new_rec`. Lines 104 through 109 allocate and verify the `new_rec` link. Line 111 assigns the character to the new link. Line 112 assigns the value from the previous link's `next_rec` pointer to the new link's `next_rec` pointer. The previous link's `next_rec` pointer is then assigned the value of the new link. This places the link into the list in a fashion similar to what was presented in Figure 3.6.

Deleting from a Linked List

The ability to add information to a linked list is good; however, there are times when you'll want to remove information too. Deleting links, or elements, is similar to adding them. You can delete links from the beginning, middle, and end of linked lists. In addition, you can delete the last link in the list. In each case, the appropriate pointers need to be moved. Also, the memory used by the deleted link needs to be freed.

Note: Don't forget to free memory when deleting links!

DO	DON'T

DON'T forget to free any memory allocated for links when deleting them.

DO understand the difference between `calloc()` and `malloc()`. Most importantly, remember that `malloc()` doesn't initialize allocated memory—`calloc()` does.

Stacks

A *stack* is a special linked list. A stack differs from the normal linked list in that it's always accessed from its top. This means new elements are always added at the top, and if an element is to be removed, it's taken from the top. This gives the stack a *Last In First Out*, or LIFO, order. It's this LIFO nature that makes a stack what it is by nature. For comparison, consider dishes; you stack them on a shelf one at a time. To remove the first dish that you placed on the shelf—the one on the bottom—you must remove each of the dishes that you placed on top of it. The first dish placed on the shelf is the last dish you can remove. Figure 3.8 illustrates a stack.

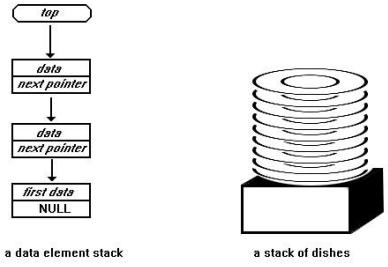


Figure 3.8. *Two sketches of stacks; elements in a linked list and dishes on a shelf.*

Using a stack basically requires the use of four functions. These functions check whether the stack is empty, return the value of the top item on the stack, push a new item onto the stack, and pop an old item off of the stack. Each of these functions is necessary for the completion of a program using a stack. Listing 3.3 illustrates the use of these four functions.

Type

Listing 3.3. STACK.C. Using a stack.

```
1:   /* Program:  stack.c
2:    * Author:   Bradley L. Jones
3:    * Purpose:  Demonstration of a stack.  (LIFO)
4:    * Note:     Program assumes that malloc() is successful.
5:    *           You should not make this assumption!
6:    *=========================================================*/
7:
8:   #include <stdio.h>
9:   #include <stdlib.h>
```

continues

Listing 3.3. continued

```
10:
11:  #define NULL 0
12:
13:  struct stack
14:  {
15:    int    value;
16:    struct stack *next;
17:  };
18:
19:  typedef struct stack LINK;
20:
21:  typedef LINK *LINK_PTR;
22:
23:  /*** prototypes ***/
24:  void push_stack( LINK_PTR *link1, int val );
25:  void pop_stack( LINK_PTR *link1, int *val );
26:  int is_stack_empty( LINK_PTR link1 );
27:  int get_stack_data( LINK_PTR link );
28:
29:
30:  int main( void )
31:  {
32:    LINK_PTR first = NULL;
33:
34:    int ctr,
35:        nbrs[10];
36:
37:    for( ctr = 0; ctr < 10; ctr ++ )
38:    {
39:      nbrs[ctr] = ctr;
40:      printf("\nPush # %d, nbrs[ctr] = %d", ctr, nbrs[ctr]);
41:      push_stack(&first, nbrs[ctr]);
42:    }
43:
44:    printf("\n------------");
45:
46:    for( ctr = 0; ctr < 10; ctr ++ )
47:    {
48:      pop_stack(&first, &nbrs[ctr]);
49:      printf("\nPop # %d, nbrs[ctr] = %d", ctr, nbrs[ctr]);
50:    }
51:
52:    return(0);
53:  }
54:
55:  /*-------------------------*
56:   * Name:    push_stack()
57:   * Purpose: Places a value into a new link on the stack.
58:   *          Returns the value of the data stored.
59:   * Params:  link = the next field from the previous link
```

```
60:    *          val  = value being placed on the stack.
61:    * Return:  None
62:    *-----------------------------*/
63:
64:   void push_stack( LINK_PTR *link1, int val )
65:   {
66:     LINK_PTR tmp_link;
67:
68:     tmp_link = (LINK_PTR) malloc( sizeof(LINK) );
69:     tmp_link->value = val;
70:     tmp_link->next  = *link1;
71:     *link1 = tmp_link;
72:   }
73:
74:   /*-----------------------------*
75:    * Name:    pop_stack()
76:    * Purpose: Removes a link from the stack.
77:    *          Returns the value of the data stored.
78:    * Params:  link = the current link that is to be removed.
79:    *          val  = value of the removed link
80:    * Return:  None
81:    *-----------------------------*/
82:
83:   void pop_stack( LINK_PTR *link1, int *val )
84:   {
85:     LINK_PTR first = *link1;
86:
87:     if ( is_stack_empty(first) == 0 )   /* if not empty */
88:     {
89:       *val   = first->value;
90:       *link1 = first->next;
91:       free( first );
92:     }
93:     else
94:     {
95:       printf("\n\nStack is empty");
96:     }
97:   }
98:
99:   /*-----------------------------*
100:   * Name:     is_stack_empty()
101:   * Purpose:  Checks to see if a link exists.
102:   * Params:   link1 = pointer to links
103:   * Return:   0 if the stack is not empty
104:   *           1 if the stack is empty
105:   *-----------------------------*/
106:
107:  int is_stack_empty( LINK_PTR link1 )
108:  {
109:    int rv = 0;
```

continues

Listing 3.3. continued

```
110:
111:    if( link1 == NULL )
112:       rv = 1;
113:
114:    return( rv );
115: }
116:
117: /*--------------------------*
118:  * Name:     get_stack_data()
119:  * Purpose:  Gets the value for a link on the stack
120:  * Params:   link = pointer to a link
121:  * Return:   value of the integer stored in link
122:  *--------------------------*/
123:
124: int get_stack_data( LINK_PTR link )
125: {
126:    return( link->value );
127: }
```

Output

```
Push # 0, nbrs[ctr] = 0
Push # 1, nbrs[ctr] = 1
Push # 2, nbrs[ctr] = 2
Push # 3, nbrs[ctr] = 3
Push # 4, nbrs[ctr] = 4
Push # 5, nbrs[ctr] = 5
Push # 6, nbrs[ctr] = 6
Push # 7, nbrs[ctr] = 7
Push # 8, nbrs[ctr] = 8
Push # 9, nbrs[ctr] = 9
----------------------
Pop # 0, nbrs[ctr] = 9
Pop # 1, nbrs[ctr] = 8
Pop # 2, nbrs[ctr] = 7
Pop # 3, nbrs[ctr] = 6
Pop # 4, nbrs[ctr] = 5
Pop # 5, nbrs[ctr] = 4
Pop # 6, nbrs[ctr] = 3
Pop # 7, nbrs[ctr] = 2
Pop # 8, nbrs[ctr] = 1
Pop # 9, nbrs[ctr] = 0
```

Analysis

This listing may seem complex at first glance; however, a large portion of it should be easily discernible. Line 11 defines the constant, `NULL`, to the value of zero. Lines 13 through 21 define the structure, `stack`, that will be used for the linked list to create the actual stack. Lines 24 through 27 present prototypes for the four functions that are typically needed when using stacks. After the prototypes, the program is ready to begin in line 30.

This program pushes several numbers onto a stack and then pops them off. The numbers are pushed on within a `for` loop in lines 37 through 42. Line 41 calls `push_stack()` to place the number. Because the numbers are always added to the top, the first structure in the linked list is passed to the `push_stack()` function. The value being pushed is stored in `nbrs[ctr]`. This happens to be the same value as the counter variable, `ctr` (see line 39). Line 48 takes each value off of the stack. Each value is placed in the `nbrs` array as it is taken off. The value is then printed in line 49. Once all ten values are popped off of the stack, the program ends.

The `push_stack()` function is relatively simple. Line 66 declares a `LINK_PTR` pointer called `tmp_link` that is used to hold the value of the new link being passed to the function. Lines 68 through 71 then perform the necessary steps to copy a link to the beginning of a list. Notice that the return value of `malloc()` is never checked to ensure that it succeeded. You should always check the return values of memory allocation functions.

The `pop_stack()` function accepts a pointer to the first link in the list along with a pointer to the value being placed on the stack. Line 85, like line 66, creates a new `LINK_PTR` pointer and copies the first pointer in the linked list to it. Line 87 calls the `is_stack_empty()` function. This function simply checks to ensure the first link is equal to `NULL` (line 111). If it is, there is no reason to pop any more links because the stack is empty. If the stack isn't empty, then the value of the first link is assigned to the parameter, `val`. The first pointer that was passed in is passed to the `next` pointer so it can be removed without losing the pointer to the beginning of the string. Finally, line 91 releases the memory used by the original beginning link back to the system using a call to `free()`.

The fourth function commonly used isn't used by this program. The function is `get_stack_data()`. If the link was more complex than a simple single data type, you would want a function that could manipulate and return the data in a link. This function could have been used if there was a need to see a data value without actually removing it from the stack.

Queues

Queues, like stacks, are special forms of linked lists. A queue is similar to a stack in that it's accessed in consistent ways. However, where a stack was LIFO (Last In First Out), a queue is *First In First Out* (FIFO). That is, instead of being accessed only from the top like a stack, it's accessed from both the top and the bottom. A queue has new items

added only to the top. When an element is removed, it's always taken from the bottom. A queue can be compared to a ticket line. The person who gets in line first is served first. In addition, people must always enter at the end of the line. Figure 3.9 illustrates a queue.

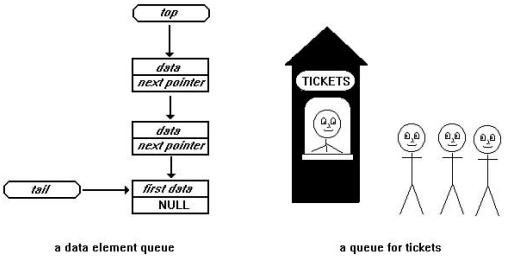


Figure 3.9. *Queues.*

A queue can be used with the same basic functions used to access a stack. You'll want the ability to see whether the queue is empty, to add an item to the top (or beginning) of the queue, to get an element from the bottom (or end) of the queue, and to see what is next in the queue. Each of these functions helps to complete a queue program.

A queue could be accomplished with a single-linked list; however, working with a queue becomes much easier with a double-linked list.

Double-Linked List

Single-linked lists enable the user to move between the elements starting at the top and working toward the bottom (or end). Sometimes it's advantageous to be able to work back toward the top. A queue would be one example of many such instances. You can traverse a list from both ends by adding a second set of links (pointers) between the elements. The double set of pointers causes the list to be double-linked. Figure 3.10 illustrates a double-linked list.

In Figure 3.10, you should notice that all the components of a single-linked list are present. In addition, a head pointer and a tail pointer are both present. As with a single-linked list, the head pointer will always point at the top or first element of the list. The tail pointer will always point at the last. If there are no elements in the list, the head and the tail pointers will both be NULL. If there is only one element, then the two pointers will be equal to the first—and only—element.

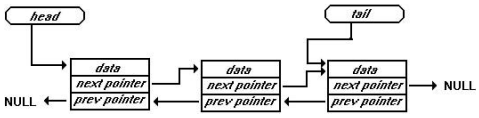


Figure 3.10. *A double-linked list.*

Note: A double-linked list must have a tail pointer in addition to its head pointer. With the exception of queues, single-linked lists don't have to have tail pointers; they are only required in double-linked lists.

A structure for an element in a double-linked list is different from that of a single-linked list in that it contains an additional pointer. The format of a structure for a double-linked list would be similar to the following:

```
struct element {
    <data>
    struct element *next;
    struct element *previous;
};
```

The `<data>` can be whatever data you are storing in your linked list. The `next` pointer points to the following element in the list. If the element is the last in the list, then `next` will contain the NULL value. The `previous` pointer contains the address of the previous element in the list. In the case of the first element—where there isn't a previous element, the value of `previous` will be NULL.

Binary Trees

A *binary tree* is a special double-linked list. It is also a special type of data tree. A *data tree* is a set of data elements that are all linked together into a hierarchical structure. Each element in a tree is called a *node*. Like a linked list that starts at its head pointer, a tree starts with what is its *root*. The root then has *sub-nodes* that are each connected to it. Sub-nodes can have even more sub-nodes below them. The bottom nodes, those that do not have any additional sub-nodes, are called *leaf nodes*. Figure 3.11 illustrates a tree structure.

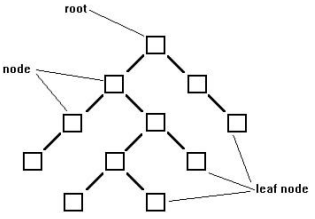


Figure 3.11. *A tree structure.*

A binary tree differs from a general tree in that each node can have a maximum of two sub-nodes. In a binary tree, the sub-nodes are commonly referred to as the left node and right node. Figure 3.11 could be considered a binary tree because none of the nodes have more than two sub-nodes.

As with linked lists, nodes in a tree are connected by using pointers to the element structures. For a binary tree, the structure contains two pointers—one for the left node and one for the right. Following is a generic structure for a binary tree node:

```
struct node {
    <data>
    struct node *left;
    struct node *right;
};
```

The `<data>` can be any data that is to be joined together in the binary tree. The node pointer, `left`, points to the sub-node to the left. The node pointer, `right`, points to the sub-node to the right.

Using a Binary Tree

A binary tree offers faster access time over a regular linked list. To find a single element in a linked list, you must access each element from one end of the list until you find the appropriate element. With a binary tree, a logarithmic number of checks can be made to determine where a specific link is.

Consider the order in which a binary tree's nodes are accessed. There are three general orders for accessing the elements in a linked list. You can access them *in order*, starting with the left sub-node working toward the root and then down the right sub-node. *Pre-order* access is to access the root first, and then the left sub-node, followed by the right sub-node. The third way to access a binary tree is in *post-order*. This is accessing the left sub-node first, and then the right sub-node, followed by the root. Figure 3.12 illustrates the order that the nodes would be accessed in each of these methods.

Expert Tip: Binary trees are used to sort information. You should use a binary tree instead of a single-linked list or double-linked list when you need to access a single element in the fewest steps.

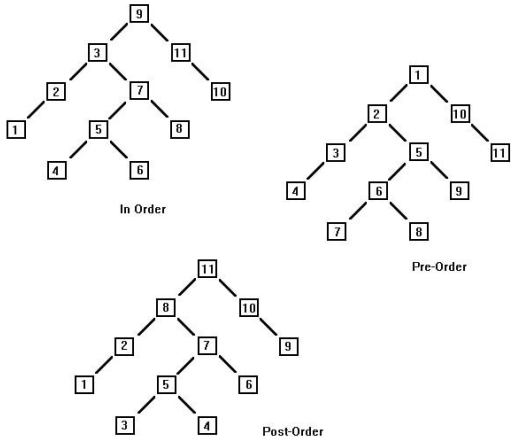


Figure 3.12. *Binary tree access orders.*

Listing 3.4 illustrates the three orders of accessing a linked list. This program enables you to enter a number of names. Each name is added to the tree based on where it fits. The values are added in order. Once all the values are entered, the program enables you to print them in each of the three orders described previously.

Type

Listing 3.4. Using a binary tree.

```
1:   /* Program:  list0304.c
2:    * Author:   Bradley L. Jones
3:    *           Gregory L. Guntle
4:    * Purpose:  Demonstrate binary tree
5:    *=========================================================*/
6:
7:   #include <stdio.h>
8:   #include <stdlib.h>
9:   #include <string.h>
10:  #include <conio.h>
11:  #include <ctype.h>
12:
13:  #ifndef NULL
```

continues

Listing 3.4. continued

```
14:  #define NULL 0
15:  #endif
16:
17:  #define FALSE 0
18:  #define TRUE  1
19:
20:  typedef struct  name_addr NAME;
21:  typedef NAME    *NAMEPTR;
22:  typedef NAMEPTR *REFtoNAMEPTR;
23:
24:  struct name_addr
25:  {
26:    char last_name[20];
27:    char first_name[10];
28:    struct name_addr *left_rec;
29:    struct name_addr *right_rec;
30:  };
31:
32:  NAMEPTR root;                  /* root of tree */
33:
34:  void get_names(NAME);
35:  void display_menu(void);
36:  void display_header(char *);
37:  void search_list(NAME, REFtoNAMEPTR);
38:  void dump_postorder(NAMEPTR);
39:  void dump_preorder(NAMEPTR);
40:  void dump_in_order(NAMEPTR);
41:  NAMEPTR get_space(void);
42:
43:  int main( void )
44:  {
45:    NAME hold_name;
46:    int menu_sel;
47:
48:    do
49:    {
50:      printf("\n\n");
51:      printf("\tBinary Tree List\n");
52:      printf("\t    Main Menu\n");
53:      printf("\t==================\n\n");
54:      printf("\tA  Add name to list\n");
55:      printf("\tD  Display list\n");
56:      printf("\tX  eXit\n");
57:      printf("\n\t   Enter selection: ");
58:      menu_sel = getche();
59:      menu_sel = toupper( menu_sel );
60:      switch( menu_sel )
61:      {
62:          case 'A':   get_names(hold_name);
63:                      break;
```

```
64:          case 'D':   display_menu();
65:                      break;
66:          case 'X':
67:                      break;
68:          default :   break;
69:       }
70:
71:    } while( menu_sel != 'X' );
72:
73:    return 0;
74:  }
75:
76:  /*=========================================================*
77:   * Function: get_names                                    *
78:   * Purpose : Accepts name into the list                   *
79:   * Returns : N/A                                          *
80:   *=========================================================*/
81:
82:  void get_names(NAME name_rec)
83:  {
84:    int finished = FALSE;
85:
86:    printf("\n\n\tAdding New Names\n");
87:    while (!finished)
88:    {
89:       printf("Last name  (enter only to exit): ");
90:       gets(name_rec.last_name);
91:       if (strlen(name_rec.last_name))    /* Is there a name */
92:       {
93:         printf("First name: ");
94:         gets(name_rec.first_name);
95:         search_list(name_rec, &root);    /* Add info to tree */
96:         printf("\n%s has been added to the list.\n",
name_rec.last_name);
97:       }
98:       else
99:         finished = TRUE;
100:   }
101: }
102:
103: /*=========================================================*
104:  * Function: display_menu                                 *
105:  * Purpose : Menu for dumping the data from the list      *
106:  * Returns : N/A                                          *
107:  *=========================================================*/
108:
109: void display_menu()
110: {
111:   int menu_sel;
112:
```

continues

Listing 3.4. continued

```
113:   do
114:   {
115:     printf("\n\n");
116:     printf("\tDisplay Names Menu\n");
117:     printf("\t==================\n\n");
118:     printf("\tI  In Order\n");
119:     printf("\tP  PreOrder\n");
120:     printf("\tO  PostOrder\n");
121:     printf("\tQ  Quit back to Main Menu");
122:     printf("\n\n\t   Enter selection: ");
123:     menu_sel = getche();
124:     menu_sel = toupper( menu_sel );
125:     switch( menu_sel )
126:     {
127:         case 'I':   display_header("In ORDER");
128:                     dump_in_order(root);
129:                     break;
130:
131:         case 'P':   display_header("Pre-ORDER");
132:                     dump_preorder(root);
133:                     break;
134:
135:         case 'O':   display_header("Post-ORDER");
136:                     dump_postorder(root);
137:                     break;
138:
139:         case 'Q':
140:         default :   break;
141:     }
142:   } while(menu_sel != 'Q');
143: }
144:
145: /*=========================================================*
146:  * Function: display_header                                *
147:  * Purpose : Displays a header for the report               *
148:  * Returns :                                               *
149:  *=========================================================*/
150:
151: void display_header(char *title)
152: {
153:   printf("\n\n\tDUMP OF DATA IN LIST - %s\n", title);
154:   printf("Addr Last Name           First Name ");
155:   printf("  Left Right\n");
156:   printf("==== =========           ========== ");
157:   printf("  ==== =====\n");
158: }
159:
160: /*=========================================================*
161:  * Function: get_space                                     *
162:  * Purpose : Gets memory for new record                    *
```

```
163:  * Returns : Address to free memory                        *
164:  *=========================================================*/
165:
166: NAMEPTR get_space()
167: {
168:    NAMEPTR memspace;
169:
170:    /* Get memory location */
171:    memspace = (NAMEPTR)malloc(sizeof(NAME));
172:
173:    if (!memspace)              /* If unable to get memory */
174:    {
175:       printf("Unable to allocate memory!\n");
176:       exit(1);
177:    }
178:    return(memspace);
179: }
180:
181: /*=========================================================*
182:  * Function: search_tree                                   *
183:  * Purpose : Displays the information current in the list *
184:  * Entry   : N/A                                           *
185:  * Returns : N/A                                           *
186:  *=========================================================*/
187:
188: void search_list(NAME new_name, REFtoNAMEPTR record)
189: {
190:    int result;               /* Holds result of comparison */
191:    NAMEPTR newrec;           /* Holds addr of prev rec-for */
192:                              /* storing later              */
193:
194:    if ( *record == NULL )    /* The place for new name? */
195:    {
196:       newrec = get_space();  /* get space for holding data */
197:         /* store information */
198:       strcpy(newrec->last_name, new_name.last_name);
199:       strcpy(newrec->first_name, new_name.first_name);
200:       newrec->left_rec = NULL;
201:       newrec->right_rec = NULL;
202:       *record = newrec;      /* Place this new rec addr into
203:                                 last rec's appropriate pointer */
204:    }
205:    else
206:    {
207:       newrec = *record;  /* Get address of record past - will
208:                           * be used to link to new record - at
209:                           * this point this variable holds
210:                           * either the left or right branch
211:                           * address.
212:                           */
```

continues

Listing 3.4. continued

```
213:
214:       /* Compare new name against this rec */
215:       result = strcmp(new_name.last_name, newrec->last_name);
216:
217:       if (result < 0)        /* Send addr not content */
218:          search_list(new_name, &newrec->left_rec);
219:       else
220:          search_list(new_name, &newrec->right_rec);
221:    }
222: }
223:
224: /*=========================================================*
225:  * Function: dump_postorder                                *
226:  * Purpose : Displays the contents of the list in POST    *
227:  *           ORDER                                        *
228:  * Entry   : root of tree                                 *
229:  * Returns : N/A                                          *
230:  *=========================================================*/
231:
232: void dump_postorder(NAMEPTR data)
233: {
234:   if (data)    /* If there is data to print */
235:   {
236:         /* keep going left until hit end */
237:      dump_postorder(data->left_rec);
238:         /* Now process right side */
239:      dump_postorder(data->right_rec);
240:      printf("%4X %-20s %-10s  %4X  %4X\n", data,
241:                   data->last_name, data->first_name,
242:                   data->left_rec,  data->right_rec);
243:   }
244: }
245:
246: /*=========================================================*
247:  * Function: dump_preorder                                *
248:  * Purpose : Displays the contents of the list PREORDER   *
249:  * Entry   : root of tree                                 *
250:  * Returns : N/A                                          *
251:  *=========================================================*/
252:
253: void dump_preorder(NAMEPTR data)
254: {
255:   if (data)              /* If there is data to print */
256:   {
257:      printf("%4X %-20s %-10s  %4X  %4X\n", data,
258:              data->last_name, data->first_name,
259:              data->left_rec,  data->right_rec);
260:         /* Now process left side */
261:      dump_preorder(data->left_rec);
262:         /* then right */
```

```
263:      dump_preorder(data->right_rec);
264:    }
265: }
266:
267: /*=========================================================*
268:  * Function: dump_in_order                                 *
269:  * Purpose : Displays the contents of the list in order    *
270:  * Entry   : root of tree                                  *
271:  * Returns : N/A                                           *
272:  *=========================================================*/
273:
274: void dump_in_order(NAMEPTR data)
275: {
276:   if (data)          /* If there is data to print */
277:   {
278:         /* keep going left until hit end */
279:      dump_in_order(data->left_rec);
280:      printf("%4X %-20s %-10s  %4X  %4X\n", data,
281:             data->last_name, data->first_name,
282:             data->left_rec,  data->right_rec);
283:      dump_in_order(data->right_rec);
284:   }
285: }
```

```
        Binary Tree List
            Main Menu
        ==================

        A  Add name to list
        D  Display list
        X  eXit

           Enter selection: A

        Adding New Names
Last name  (enter only to exit): Jones
First name: Bradley

Jones has been added to the list.
Last name  (enter only to exit): Guntle
First name: Gregory

Guntle has been added to the list.
Last name  (enter only to exit): Johnson
First name: Jerry

Johnson has been added to the list.
Last name  (enter only to exit): Zacharia
First name: Zelda
```

```
Zacharia has been added to the list.
Last name  (enter only to exit):
```

In order:

```
        DUMP OF DATA IN LIST - In ORDER
Addr Last Name            First Name   Left Right
==== =========            ==========   ==== =====
 94E Guntle                Gregory         0   974
 974 Johnson               Jerry           0     0
 928 Jones                 Bradley       94E   99A
 99A Zacharia              Zelda           0     0
```

Pre-order:

```
        DUMP OF DATA IN LIST - Pre-ORDER
Addr Last Name            First Name   Left Right
==== =========            ==========   ==== =====
 928 Jones                 Bradley       94E   99A
 94E Guntle                Gregory         0   974
 974 Johnson               Jerry           0     0
 99A Zacharia              Zelda           0     0
```

Post-order:

```
        DUMP OF DATA IN LIST - Post-ORDER
Addr Last Name            First Name   Left Right
==== =========            ==========   ==== =====
 974 Johnson               Jerry           0     0
 94E Guntle                Gregory         0   974
 99A Zacharia              Zelda           0     0
 928 Jones                 Bradley       94E   99A
```

Analysis This is a long listing that deserves a detailed explanation; however, I am only going to provide an overview. You'll find that until you need to actually use a linked list, understanding the low-level details isn't extremely important. However, it is important to understand the concepts involved.

Lines 34 through 41 prototype the functions that are used in this listing. These functions include `get_names()`, which is presented in lines 76 through 101. This function simply prompts the user for names and adds them to the binary tree. The next function prototyped is `display_menu()`. In lines 103 through 143, this function uses `printf()` to display a reporting menu on the screen. The `getche()` function is used to get the user's input.

The third function prototyped is `display_header()`. In lines 145 through 158, the `display_header()` function is used by each of the three reporting functions to print headers on the report. The three reports are also prototyped. The reports are each in their own functions, `dump_postorder()`, `dump_preorder()`, and `dump_inorder()`.

Each of these functions operates similarly. They read the tree and print each node. The difference is in the order that the nodes are navigated.

Two additional functions, `search_list()` and `get_space()`, have been prototyped. The `search_list()` function, which appears in lines 181 through 222, works to navigate through the list for inserting new records. The `get_space()` function, which appears in lines 160 through 179, simply allocates space for new links in the tree.

All of these functions combined with the `main()` function provide a menuing program that enables you to create a binary tree. While somewhat long, this program uses some of the same logic provided in the single-linked list and the stack that were presented earlier today.

DO	DON'T
DO use a double-linked list when you must be able to go forward and backward in a linked list.	
DO use a binary tree when access time is critical.	

Summary

Today, procedures were presented for enabling data to be used in sorted orders. These procedures included linked lists and trees. These constructs are generally used when speed is important. Because these constructs work in memory, access time is extremely fast. In addition, pointers are used to always keep the elements in order. Two special linked lists, stacks and queues, were presented. Stacks use a LIFO, or Last In First Out, order of access. Queues use a FIFO, or First In First Out, order of access. Double-linked lists enable both the preceding and following elements to be accessed. This is different from a single-linked list that can only be traversed from beginning to end. Binary trees are another form of linked data. Binary trees have even quicker retrieval time at the cost of storing additional linkage information.

Q&A

Q What is the difference between a single-linked list, a linear linked list, and a single linked list?

A There is no difference. These are three terms for the same thing. In addition, a double-linked list is the same thing as a double linked list. These are all different ways of saying the same thing.

Q Are any additional pointers, other than the tail pointer and head pointer, used with a linked list?

A A third pointer that is external to the elements in a linked list may be used. This is a "current" pointer. This additional pointer may be used if it's important to know where you are currently working in a list.

Q What is the advantage of a binary tree over a linked list?

A A binary tree allows for quicker searching of the element saved. The cost of the quicker search is the need to store additional pointer information.

Q Are there other trees than just binary trees?

A Yes. Binary trees are a special form of tree. They are easier to understand than general trees. A general tree involves much more complex manipulations than were presented with the binary trees.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What does NULL equal?
2. What does it mean if the head pointer is equal to NULL?
3. How are single-linked lists connected?
4. How does a stack differ from a normal linked list?
5. How does a queue differ from a normal linked list?
6. What is a tail pointer? What is a top pointer?
7. Is a tail pointer needed in a single-linked list?
8. What is the advantage of a double-linked list over a single-linked list?

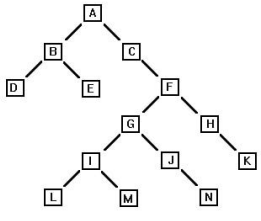
9. What is the benefit of using `calloc()` over `malloc()` when allocating memory for new list elements?
10. What is the advantage of using a binary tree over a single-linked list?

Exercises

1. Write a structure that is to be used in a single-linked list. The structure should hold the name of a Disney character and the year the character was first introduced.
2. Write a structure that is to be used in a double-linked list. The structure is to hold the name of a fruit or vegetable.
3. Write a structure that is to be used with a binary tree. The structure is to hold the first name and ages of all the people in your family.
4. **ON YOUR OWN:** Write a function that will count the number of elements in the tree created in Listing 3.4. Add this to the program's menu.
5. **BUG BUSTER:** What is wrong with the following linked list structure?

```
struct customer {
    char lastname[20+1];
    char firstname[15+1];
    char middle_init;
    struct customer next;
};
```

6. In what order will the nodes in the following figure be accessed when using in order access? (Consider a single sub-node as being a left node even though the figure may show it going right.)



7. In what order will the nodes be accessed if using pre-order?
8. In what order will the nodes be accessed if using post-order?
9. **ON YOUR OWN:** Rewrite Listing 3.6 to sort the linked list elements in zip code order.