

A First Book of ANSI C

Fourth Edition

Chapter 13

Dynamic Data Structures

Objectives

- Introduction to Linked Lists
- Dynamic Memory Allocation
- Stacks
- Queues
- Dynamically Linked Lists
- Common Programming and Compiler Errors

Introduction

- **Dynamic memory allocation:** an alternative to fixed memory allocation in which memory space grows or diminishes during program execution
- Dynamic memory allocation makes it unnecessary to reserve a fixed amount of memory for a scalar, array, or structure variable in advance
 - Also known as **run-time allocation**
 - Requests are made for allocation and release of memory space while the program is running

Introduction to Linked Lists

- An array of structures can be used to insert and delete ordered structures, but this is not an efficient use of memory
 - Better alternative: a linked list
- **Linked list:** set of structures, each containing at least one member whose value is the address of the next logically ordered structure in the list
 - Also known as **self-referencing structures**

Introduction to Linked Lists (continued)

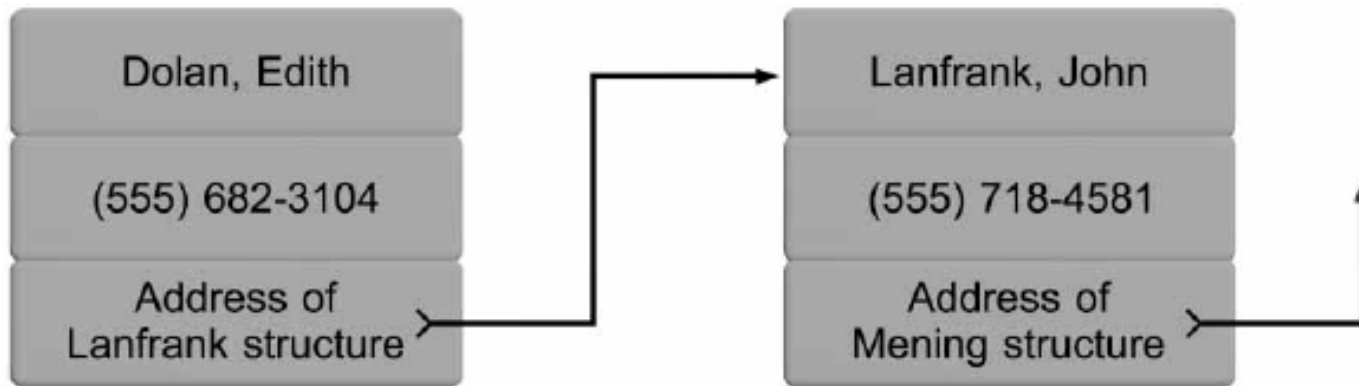


Figure 13.2 Using pointers to link structures

Introduction to Linked Lists (continued)

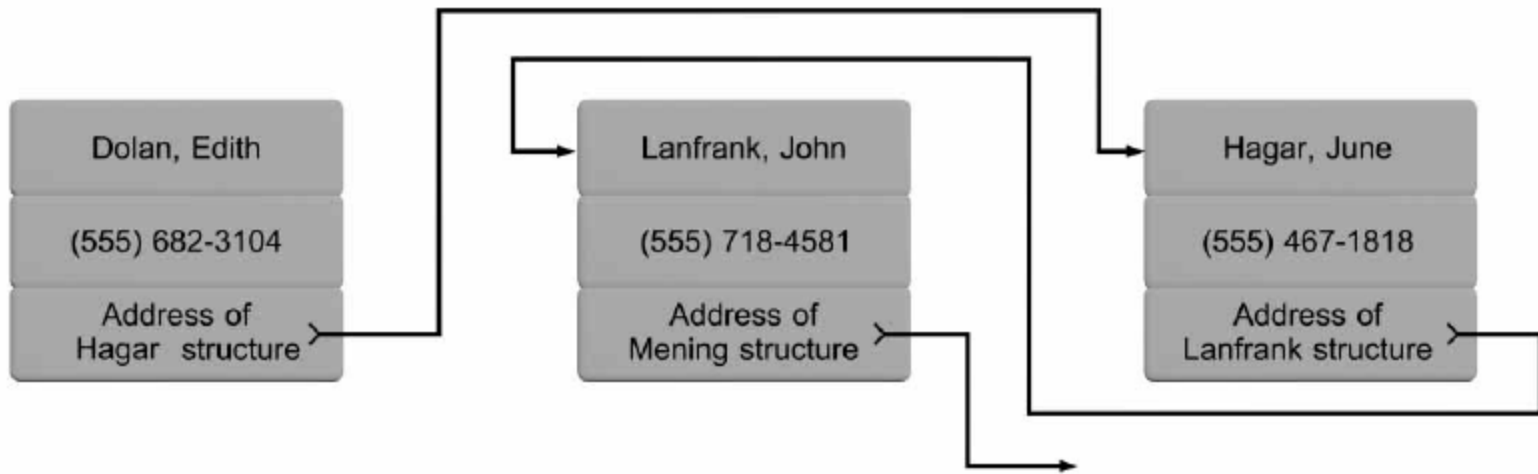


Figure 13.3 Adjusting addresses to point to appropriate structures

Introduction to Linked Lists (continued)

- A NULL acts as a sentinel or flag to indicate when the last structure has been processed
- In addition to an end-of-list sentinel value, we must provide a special pointer for storing the address of the first structure in the list

Introduction to Linked Lists (continued)

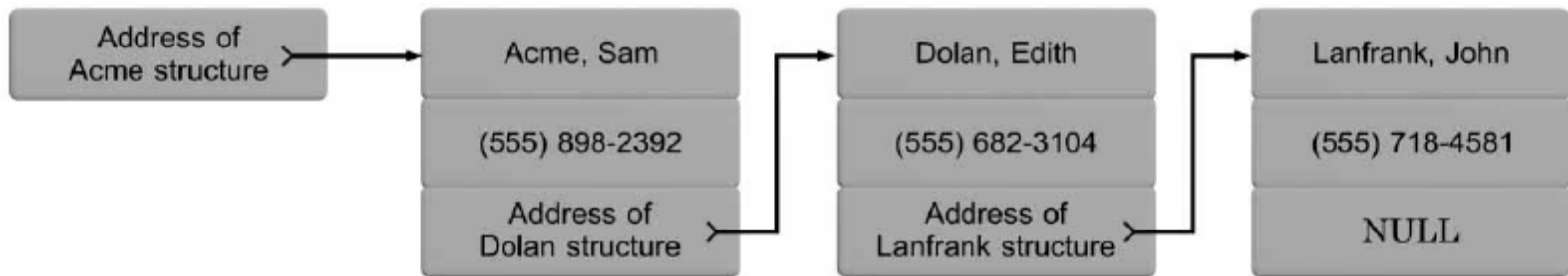


Figure 13.4 Use of the initial and final pointer values

Introduction to Linked Lists (continued)

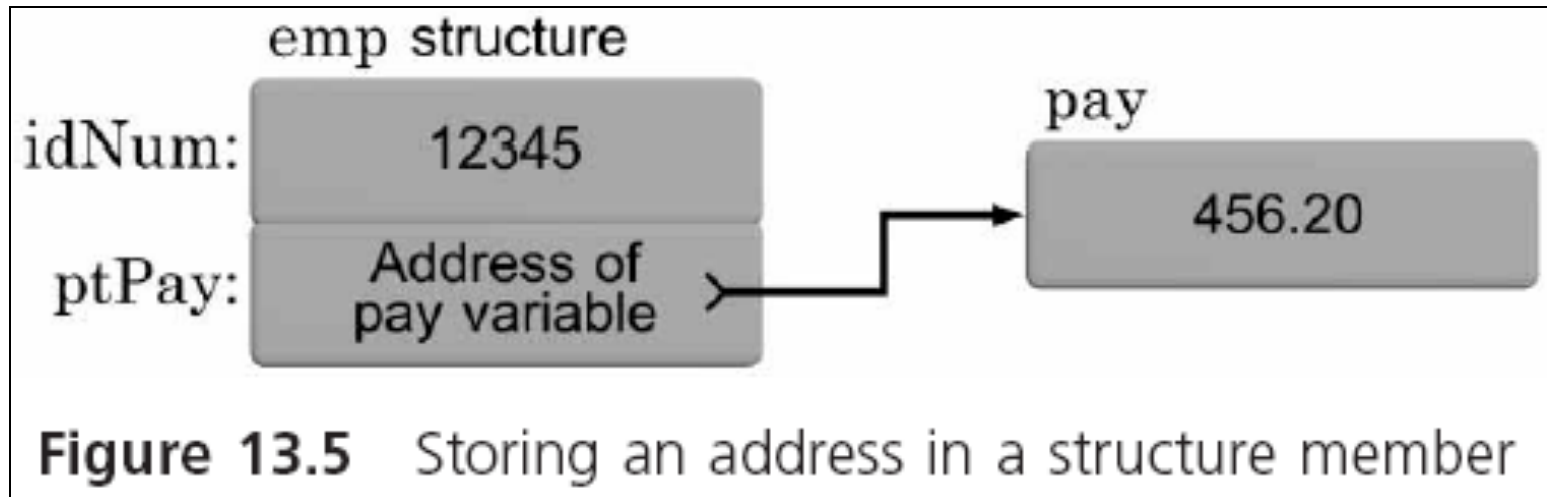


Figure 13.5 Storing an address in a structure member

Introduction to Linked Lists (continued)



Program 13.1

```
1  #include <stdio.h>
2
3  struct Test
4  {
5      int idNum;
6      double *ptPay;
7  };
8
9  int main()
10 {
11     struct Test emp;
12     double pay = 456.20;
13
14     emp.idNum = 12345;
15     emp.ptPay = &pay;
16
17     printf("Employee number %d was paid $%6.2f\n", emp.idNum,
18                                                  *emp.ptPay);
19     return 0;
20 }
```

A structure can contain any data type, including a pointer. A pointer member of a structure is used like any other pointer variable.

Introduction to Linked Lists (continued)



Program 13.2

```
1  #include <stdio.h>
2  #define MAXNAME 30
3  #define MAXPHONE 15
4
5  struct TeleType
6  {
7      char name[MAXNAME];
8      char phoneNum[MAXPHONE];
9      struct TeleType *nextaddr;
10 };
11
12 int main()
13 {
14     struct TeleType t1 = {"Acme, Sam", "(555) 898-2392"};
15     struct TeleType t2 = {"Dolan, Edith", "(555) 682-3104"};
16     struct TeleType t3 = {"Lanfrank, John", "(555) 718-4581"};
17     struct TeleType *first;    /* create a pointer to a structure */
18
19     first = &t1;                /* store t1's address in first */
20     t1.nextaddr = &t2;         /* store t2's address in t1.nextaddr */
21     t2.nextaddr = &t3;         /* store t3's address in t2.nextaddr */
22     t3.nextaddr = NULL;        /* store the NULL address in t3.nextaddr */
23
24     printf("%s\n%s\n%s\n", first->name, t1.nextaddr->name, t2.nextaddr->name);
25
26     return 0;
27 }
```

is evaluated as `(t1.nextaddr)->name`
it can be replaced by `(*t1.nextaddr).name`

Introduction to Linked Lists (continued)

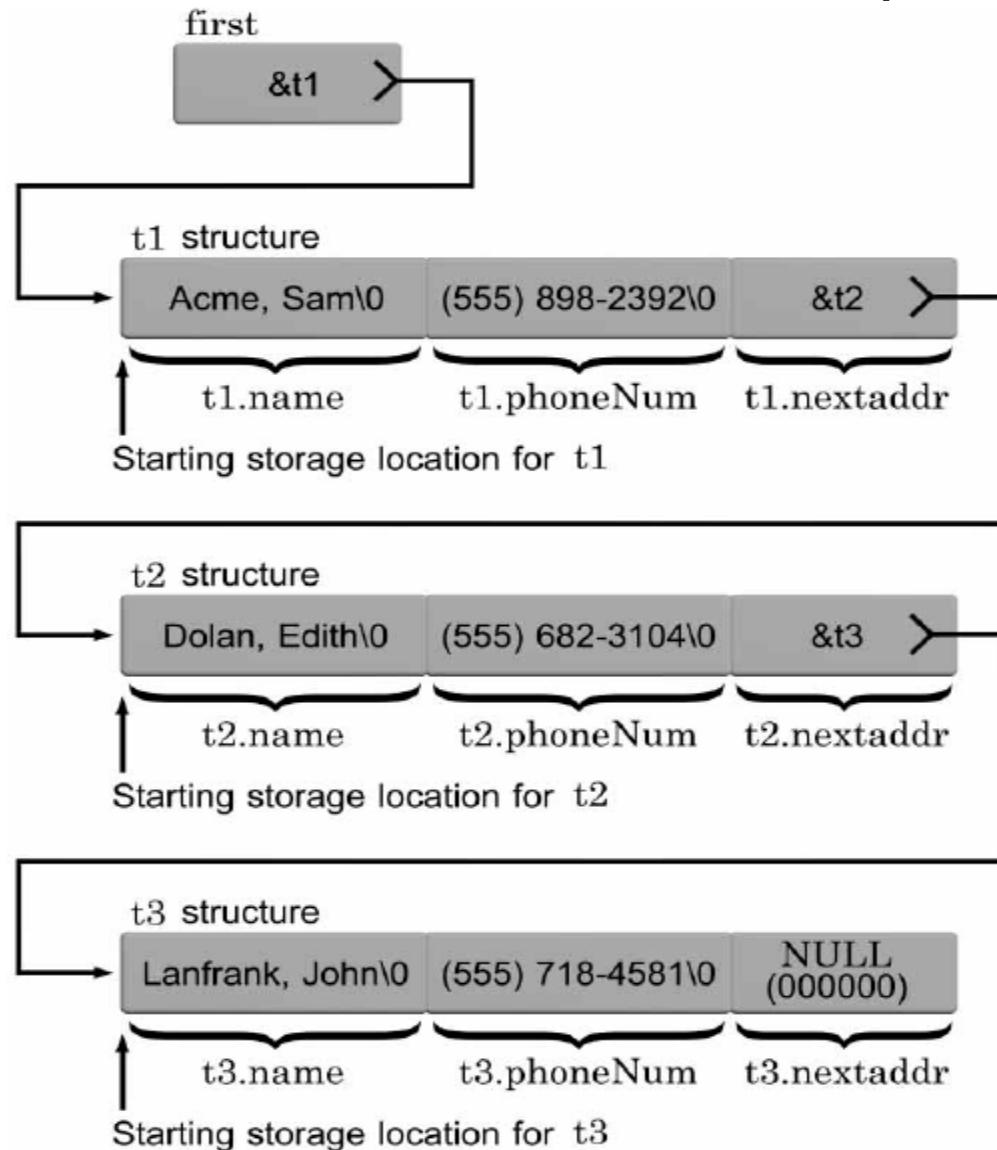


Figure 13.6 The relationship between structures in Program 13.2

Introduction to Linked Lists (continued)




Program 13.3

```
1  #include <stdio.h>
2  #define MAXNAME 30
3  #define MAXPHONE 15
4
5  struct TeleType
6  {
7      char name[MAXNAME];
8      char phoneNum[MAXPHONE];
9      struct TeleType *nextaddr;
10 };
11
12 int main()
13 {
14     struct TeleType t1 = {"Acme, Sam", "(555) 898-2392"};
15     struct TeleType t2 = {"Dolan, Edith", "(555) 682-3104"};
16     struct TeleType t3 = {"Lanfrank, John", "(555) 718-4581"};
17     struct TeleType *first; /* create a pointer to a structure */
18     void display(struct TeleType *); /* function prototype */
```

Disadvantage: exactly three structures are defined in `main()` by name, and storage for them is reserved at compile time

Introduction to Linked Lists (continued)

```
19
20  first = &t1;          /* store t1's address in first */
21  t1.nextaddr = &t2;   /* store t2's address in t1.nextaddr */
22  t2.nextaddr = &t3;   /* store t3's address in t2.nextaddr */
23  t3.nextaddr = NULL; /* store the NULL address in t3.nextaddr */
24
25  display(first);      /* send the address of the first structure */
26
27  return 0;
28 }
29
30 void display(struct TeleType *contents) /* contents is a pointer */
31 {                                       /* to a structure of type TeleType */
32  while (contents != NULL) /* display till end of linked list */
33  {  can be replaced by while(!contents)
34    printf("%-30s %-20s\n", contents->name, contents->phoneNum);
35    contents = contents->nextaddr; /* get next address */
36  }
37 }
```

Dynamic Memory Allocation

Table 13.1 Functions to Dynamically Allocate and Deallocate Memory Space

Function Name	Description
<code>malloc()</code>	Reserves the number of bytes requested by the argument passed to the function. Returns the address of the first reserved location, as an address of a <code>void</code> data type, or <code>NULL</code> if sufficient memory is not available.
<code>calloc()</code>	Reserves space for an array of <code>n</code> elements of the specified size. Returns the address of the first reserved location and initializes all reserved bytes to 0s, or returns a <code>NULL</code> if sufficient memory is not available.
<code>realloc()</code>	Changes the size of previously allocated memory to a new size. If the new size is larger than the old size, the additional memory space is uninitialized and the contents of the original allocated memory remain unchanged; otherwise, the new allocated memory remains unchanged up to the limits of the new size.
<code>free()</code>	Releases a block of bytes previously reserved. The address of the first reserved location is passed as an argument to the function.

Dynamic Memory Allocation (continued)

- The `malloc()` and `calloc()` functions can frequently be used interchangeably
 - The advantage of `calloc()` is that it initializes all newly allocated numeric memory to 0 and character allocated memory to NULL
 - We use `malloc()` because it is the more general purpose of the two functions
 - `malloc(10*sizeof(char))` or `calloc(10, sizeof(char))` requests enough memory to store 10 characters
- The space allocated by `malloc()` comes from the computer's **heap**

Dynamic Memory Allocation (continued)



Program 13.4

```
4 int main()
5 {
6     int numgrades, i;
7     int *grades;
8
9     printf("\nEnter the number of grades to be processed: ");
10    scanf("%d", &numgrades);
11
12    /* here is where the request for memory is made */
13    grades = (int *) malloc(numgrades * sizeof(int));
14
15    /* here we check that the allocation was satisfied */
16    if (grades == (int *) NULL)
17    {
18        printf("\nFailed to allocate grades array\n");
19        exit(1);
20    }
21
22    for(i = 0; i < numgrades; i++)
23    {
24        printf("  Enter a grade: ");
25        scanf("%d", &grades[i]);
26    }
27
28    printf("\nAn array was created for %d integers", numgrades);
29    printf("\nThe values stored in the array are:\n");
30
31    for (i = 0; i < numgrades; i++)
32        printf(" %d\n", grades[i]);
33
34    free(grades);
35
36    return 0;
37 }
```

Necessary because malloc() returns void

Dynamic Memory Allocation (continued)

- `malloc()` is more typically used for dynamically allocating memory for structures

```
struct OfficeInfo *Off;
/* request space for one structure */
Off = (struct OfficeInfo *) malloc(sizeof(struct
    OfficeInfo));
/* check that space was allocated */
if (Off == (struct OfficeInfo*) NULL)
{
    printf("\nAllocation failed\n");
    exit(1);
}
```

Stacks

- Stack: special type of linked list in which objects can only be added to and removed from the top of the list
 - Last-in, first-out (LIFO) list
- In a true stack, the only item that can be seen and accessed is the top item

Stacks (continued)

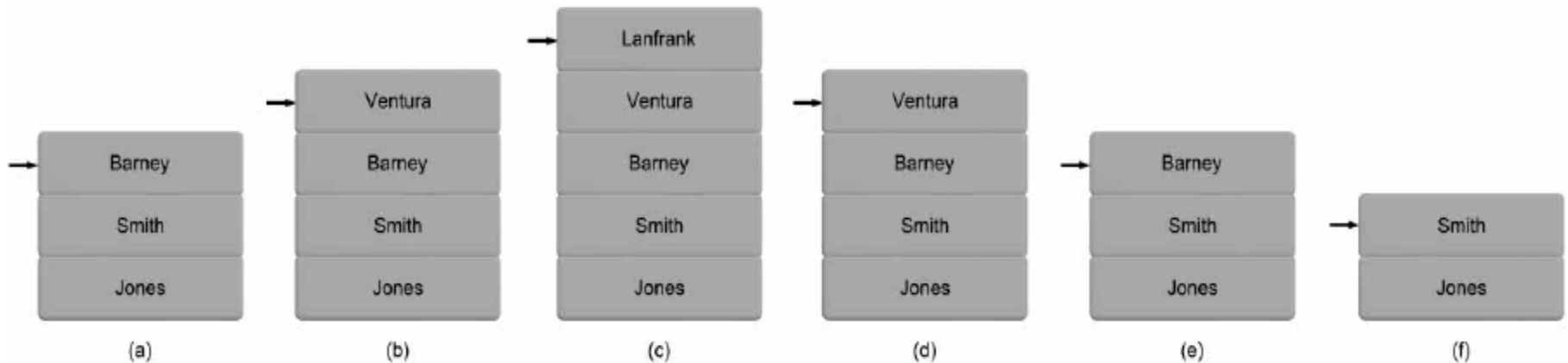


Figure 13.8 An expanding and contracting stack of names

Stack Implementation

- Creating a stack requires the following four components:
 - A structure definition
 - A method of designating the current top stack structure
 - An operation for placing a new structure on the stack
 - An operation for removing a structure from the stack

PUSH and POP

PUSH (add a new structure to the stack)

Dynamically create a new structure

Put the address in the top-of-stack pointer into the address field of the new structure

Fill in the remaining fields of the new structure

Put the address of the new structure into the top-of-stack pointer

POP (remove a structure from the top of the stack)

Move the structure contents pointed to by the top-of-stack pointer into a work area

Free the structure pointed to by the top-of-stack pointer

Move the address in the work area address field into the top-of-stack pointer

PUSH and POP (continued)

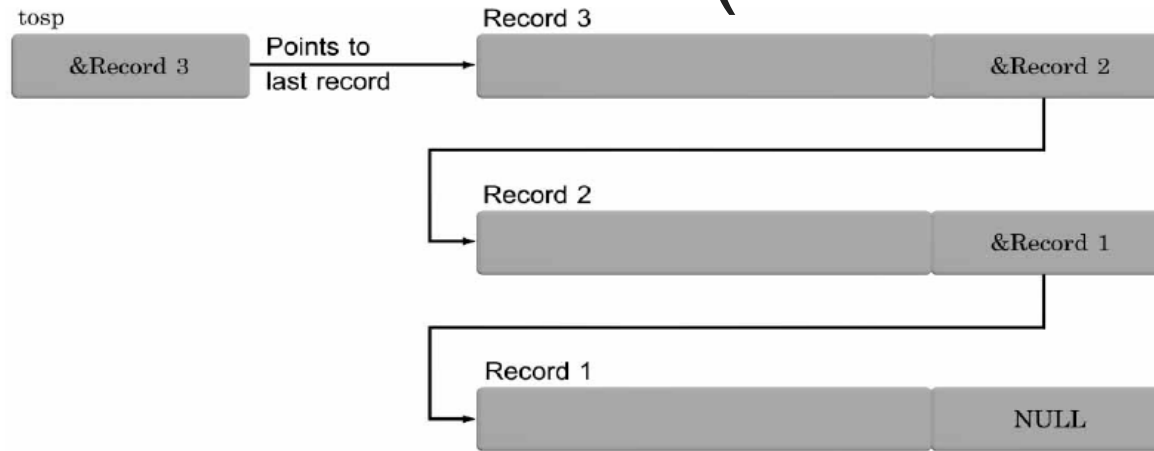


Figure 13.9 A stack consisting of three structures

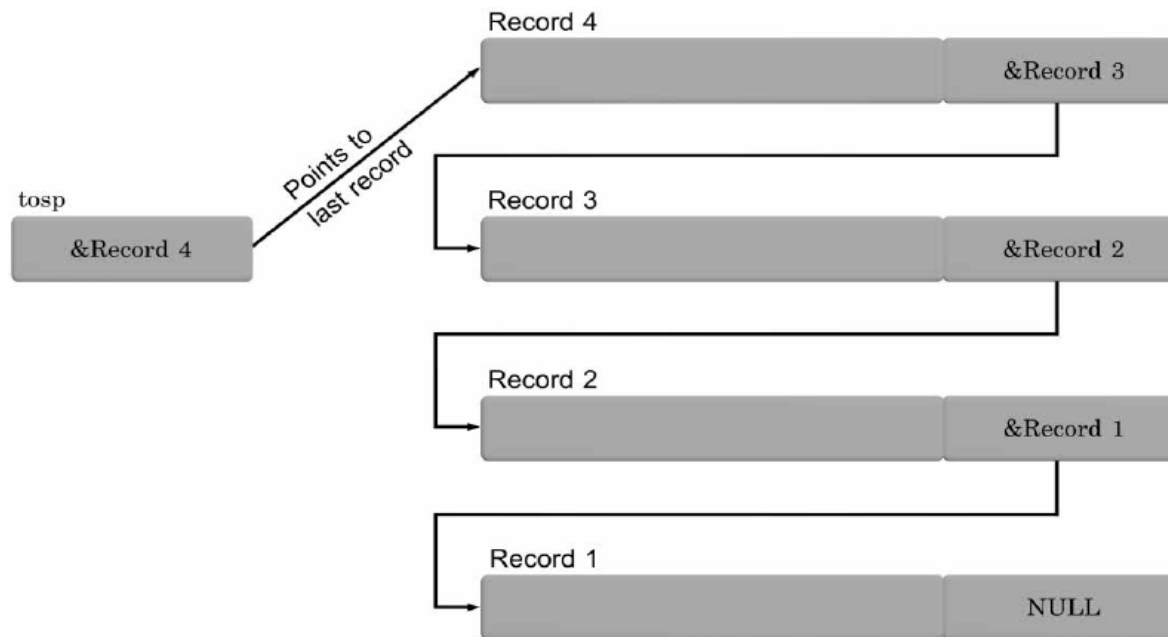


Figure 13.10 The stack after a PUSH

PUSH and POP (continued)



Program 13.5

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAXCHARS 30
5  #define DEBUG 0
6
7  /* here is the declaration of a stack structure */
8  struct NameRec
9  {
10     char name[MAXCHARS];
11     struct NameRec *priorAddr;
12 };
13
14 /* here is the definition of the top-of-stack pointer */
15 struct NameRec *tosp;
16
17 int main()
18 {
19     void readPush(); /* function prototypes */
20     void popShow();
21
22     tosp = NULL; /* initialize the top-of-stack pointer */
23     readPush();
24     popShow();
25
26     return 0;
27 }
```


PUSH and POP (continued)

```
29  /* get a name and push it onto the stack */
30  void readPush()
31  {
32      char name[MAXCHARS];
33      void push(char *);
34
35      printf("Enter as many names as you wish, one per line");
36      printf("\nTo stop entering names, enter a single x\n");
37      while (1)
38      {
39          printf("Enter a name: ");
40          gets(name);
41          if (strcmp(name, "x") == 0)
42              break;
43          push(name);
44      }
45  }
46
47  /* pop and display names from the stack */
48  void popShow()
49  {
50      char name[MAXCHARS];
51      void pop(char *);
52
53      printf("\nThe names popped from the stack are:\n");
54      while (tosp != NULL) /* display till end of stack */
55      {
56          pop(name);
57          printf("%s\n", name);
58      }
59  }
```

PUSH and POP (continued)

```
61 void push(char *name)
62 {
63     struct NameRec *newaddr; /* pointer to structure of type NameRec */
64
65     if (DEBUG)
66         printf("Before the push the address in tosp is %p", tosp);
67
68     newaddr = (struct NameRec *) malloc(sizeof(struct NameRec));
69     if (newaddr == (struct NameRec *) NULL)
70     {
71         printf("\nFailed to allocate memory for this structure\n");
72         exit(1);
73     }
74     strcpy(newaddr->name,name); /* store the name */
75     newaddr->priorAddr = tosp; /* store address of prior structure */
76     tosp = newaddr;          /* update the top-of-stack pointer */
77
78     if (DEBUG)
79         printf("\n After the push the address in tosp is %p\n", tosp);
80 }
82 void pop(char *name)
83 {
84     struct NameRec *tempAddr;
85
86     if (DEBUG)
87         printf("Before the pop the address in tosp is %p\n", tosp);
88     strcpy(name,tosp->name); /* retrieve the name from the top-of-stack */
89     tempAddr = tosp->priorAddr; /* retrieve the prior address */
90     free(tosp); /* release the structure's memory space */
91     tosp = tempAddr; /* update the top-of-stack pointer */
92
93     if (DEBUG)
94         printf(" After the pop the address in tosp is %p\n", tosp);
95 }
96 }
```

PUSH and POP (continued)

- A sample run using Program 13.5 produced the following:

```
Enter as many names as you wish, one per line
To stop entering names, enter a single x
Enter a name: Jane Jones
Enter a name: Bill Smith
Enter a name: Jim Robinson
Enter a name: x
```

```
The names popped from the stack are:
Jim Robinson
Bill Smith
Jane Jones
```

Queues

- A second important data structure that relies on linked structures is called a **queue**
 - Items are removed from a queue in the order in which they were entered
 - It is a first in, first out (FIFO) structure

Queues (continued)

Harriet Wright ← **last name on the queue** (queueIn)
Jim Robinson
Bill Smith
Jane Jones ← **first name on the queue** (queueOut)

Figure 13.11 A queue with its pointers

Queues (continued)

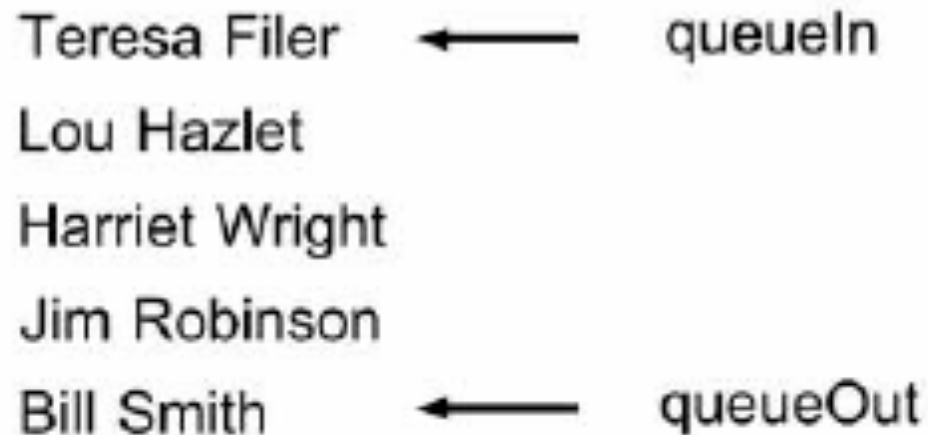


Figure 13.12 The updated queue pointers

Enque and Serve

- **Enqueueing:** placing a new item on top of the queue
- **Serving:** removing an item from a queue

Enque and Serve (continued)

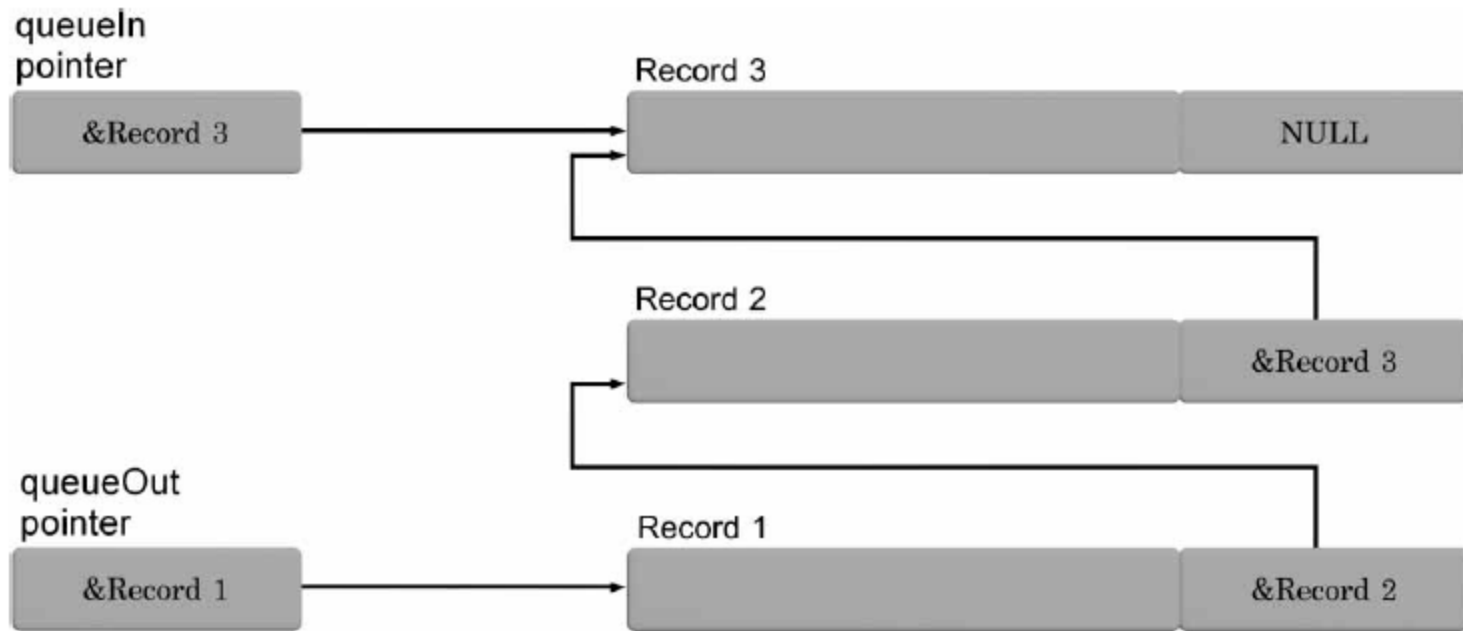


Figure 13.13 A queue consisting of three structures

Enqueue and Serve (continued)

Enqueue (add a new structure to an existing queue)

Dynamically create a new a structure

Set the address field of the new structure to a NULL

Fill in the remaining fields of the new structure

Set the address field of the prior structure (which is pointed to by the queueIn pointer) to the address of the newly created structure

Update the address in the queueIn pointer with the address of the newly created structure

Enqueue and Serve (continued)

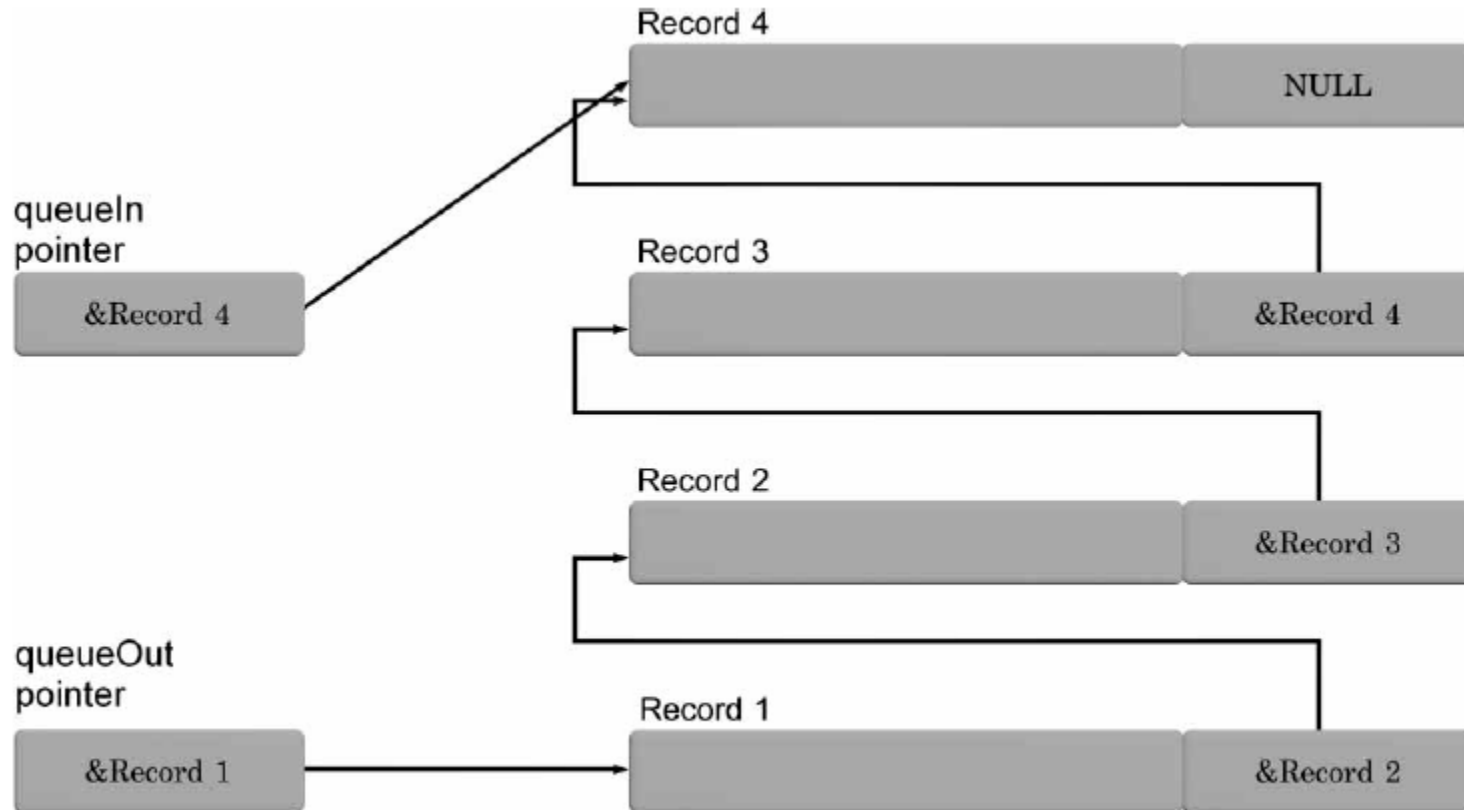


Figure 13.14 The queue after an enqueue (PUSH)

Enque and Serve (continued)

Serve (remove a structure from an existing queue)

Move the contents of the structure pointed to by the queueOut pointer into a work area

Free the structure pointed to by the queueOut pointer

Move the address in the work area address field into the queueOut pointer

Enque and Serve (continued)

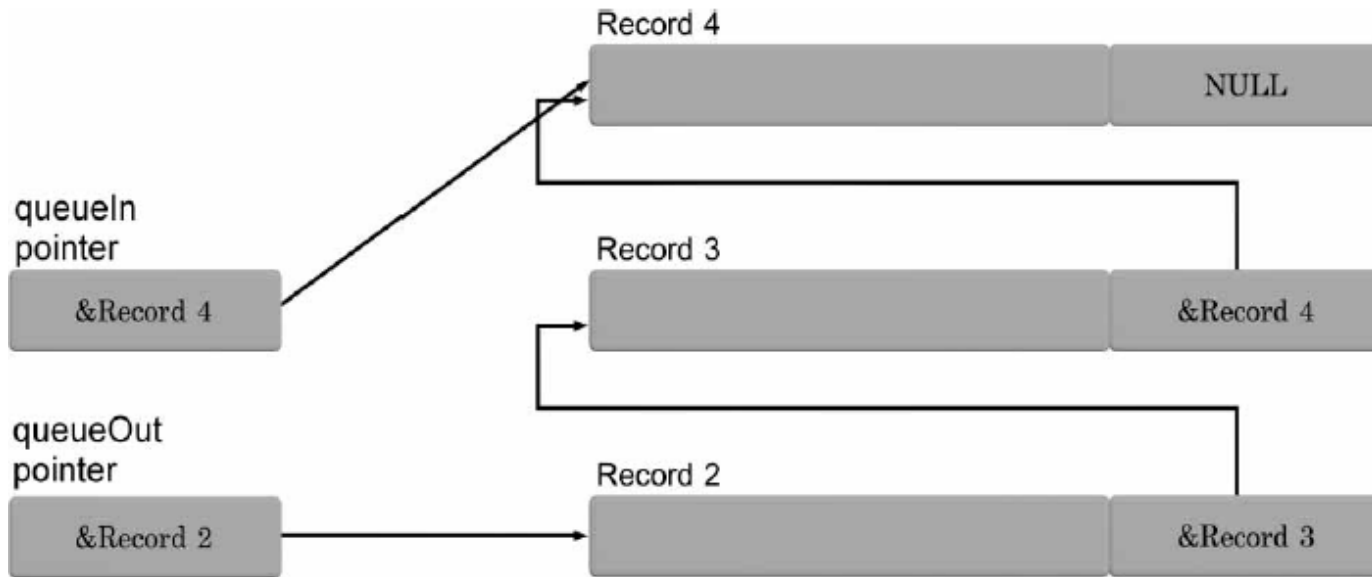


Figure 13.15 The queue after a serve (POP)

Enqueue and Serve (continued)



Program 13.6

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAXCHARS 30
5  #define DEBUG 0
6
7  /* here is the declaration of a queue structure */
8  struct NameRec
9  {
10     char name[MAXCHARS];
11     struct NameRec *nextAddr;
12 };
13
14 /* here is the definition of the top and bottom queue pointers */
15 struct NameRec *queueIn, *queueOut;
16
17 int main()
18 {
19     void readEnqueue(); /* function prototypes */
20     void serveShow();
21     queueIn = NULL; /* initialize queue pointers */
22     queueOut = NULL;
23     readEnqueue();
24     serveShow();
25 }
```

Enque and Serve (continued)

```
26  /* get a name and enqueue it onto the queue */
27  void readEnque()
28  {
29      char name[MAXCHARS];
30      void enqueue(char *);
31
32      printf("Enter as many names as you wish, one per line");
33      printf("\nTo stop entering names, enter a single x\n");
34      while (1)
35      {
36          printf("Enter a name: ");
37          gets(name);
38          if (strcmp(name, "x") == 0)
39              break;
40          enqueue(name);
41      }
42  }
43  /* serve and display names from the queue */
44  void serveShow()
45  {
46      char name[MAXCHARS];
47      void serve(char *);
48
49      printf("\nThe names served from the queue are:\n");
50      while (queueOut != NULL) /* display till end of queue */
51      {
52          serve(name);
53          printf("%s\n", name);
54      }
55  }
```

Enque and Serve (continued)

```
57 void enqueue(char *name)
58 {
59     struct NameRec *newaddr; /* pointer to structure of type NameRec */
60
61     if (DEBUG)
62     {
63         printf("Before the enqueue the address in queueIn is %p", queueIn);
64         printf("\nand the address in queueOut is %p", queueOut);
65     }
66
67     newaddr = (struct NameRec *) malloc(sizeof(struct NameRec));
68     if (newaddr == (struct NameRec *) NULL)
69     {
70         printf("\nFailed to allocate memory for this structure\n");
71         exit(1);
72     }
73
74     /* the next two if statements handle the empty queue initialization */
75     if (queueOut == NULL)
76         queueOut = newaddr;
77     if (queueIn != NULL)
78         queueIn->nextAddr = newaddr; /* fill in prior structure's address field */
79     strcpy(newaddr->name,name); /* store the name */
80     newaddr->nextAddr = NULL; /* set address field to NULL */
81     queueIn = newaddr; /* update the top-of-queue pointer */
82
83     if (DEBUG)
84     {
85         printf("\n After the enqueue the address in queueIn is %p\n", queueIn);
86         printf(" and the address in queueOut is %p\n", queueOut);
87     }
88 }
```

Enqueue and Serve (continued)

```
90 void serve(char *name)
91 {
92     struct NameRec *nextAddr;
93
94     if (DEBUG)
95         printf("Before the serve the address in queueOut is %p\n", queueOut);
96
97     /* retrieve the name from the bottom-of-queue */
98     strcpy(name, queueOut->name);
99
100    /* capture the next address field */
101    nextAddr = queueOut->nextAddr;
102
103    free(queueOut);
104
105    /* update the bottom-of-queue pointer */
106    queueOut = nextAddr;
107    if (DEBUG)
108        printf(" After the serve the address in queueOut is %u\n",
109                queueOut);
110 }
```


Enque and Serve (continued)

- A sample run using Program 13.6 produced the following:

```
Enter as many names as you wish, one per line
```

```
To stop entering names, enter a single x
```

```
Enter a name: Jane Jones
```

```
Enter a name: Bill Smith
```

```
Enter a name: Jim Robinson
```

```
Enter a name: x
```

```
The names served from the queue are:
```

```
Jane Jones
```

```
Bill Smith
```

```
Jim Robinson
```

Dynamically Linked Lists

- Both stacks and queues are examples of linked lists in which elements can only be added to and removed from the ends of the list
- In a dynamically linked list, this capability is extended to permit adding or deleting a structure from anywhere within the list
- Such a capability is extremely useful when structures must be kept within a specified order

INSERT and DELETE

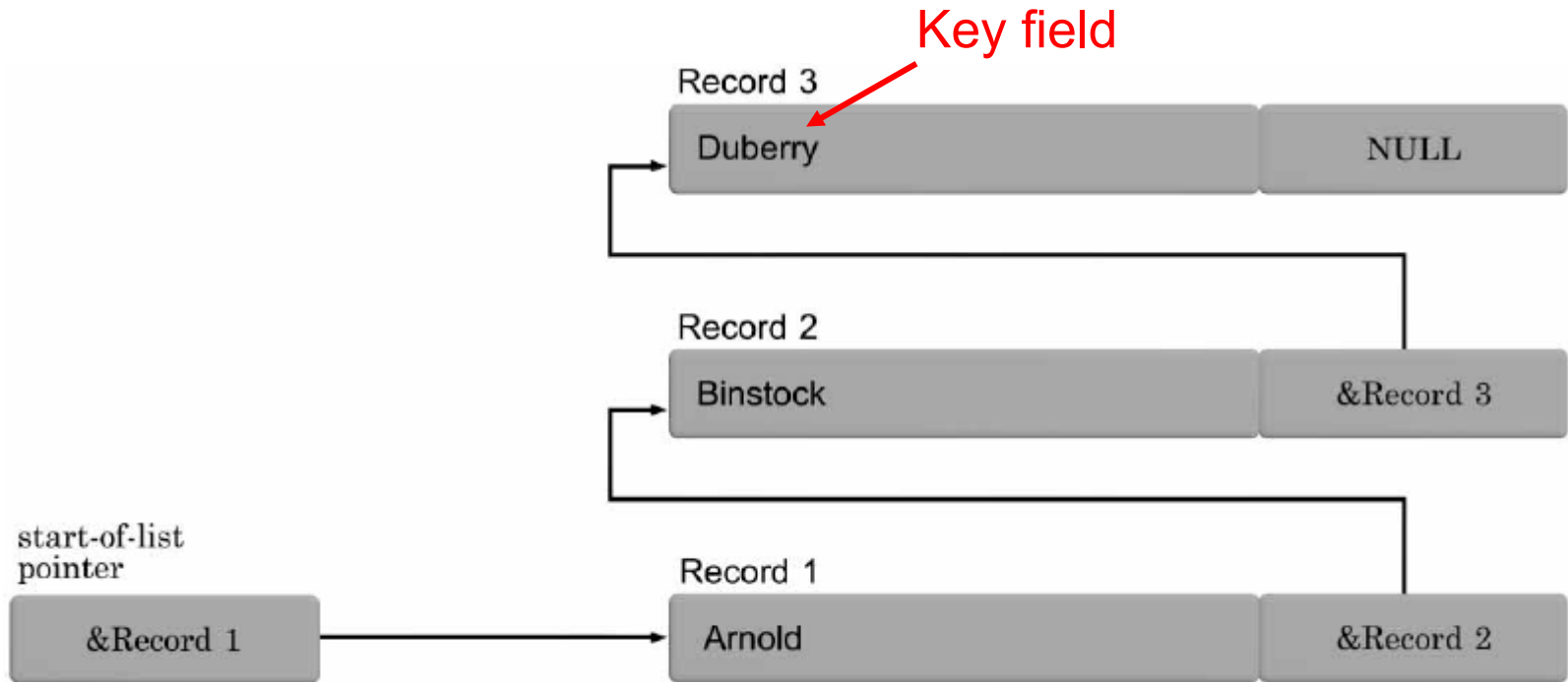


Figure 13.16 The initial linked list

INSERT and DELETE (continued)

INSERT (add a new structure into a linked list)

Dynamically allocate space for a new structure

If no structures exist in the list

Set the address field of the new structure to a NULL

Set address in the first structure pointer to address of newly created structure

Else / we are working with an existing list */*

Locate where this new structure should be placed

If this structure should be the new first structure in the list

Copy current contents of first structure pointer into address field of newly created structure

Set address in the first structure pointer to address of newly created structure

Else

Copy the address in the prior structure's address member into the address field of the newly created structure

Set address of prior structure's address member to address of newly created structure

EndIf

EndIf

INSERT and DELETE (continued)

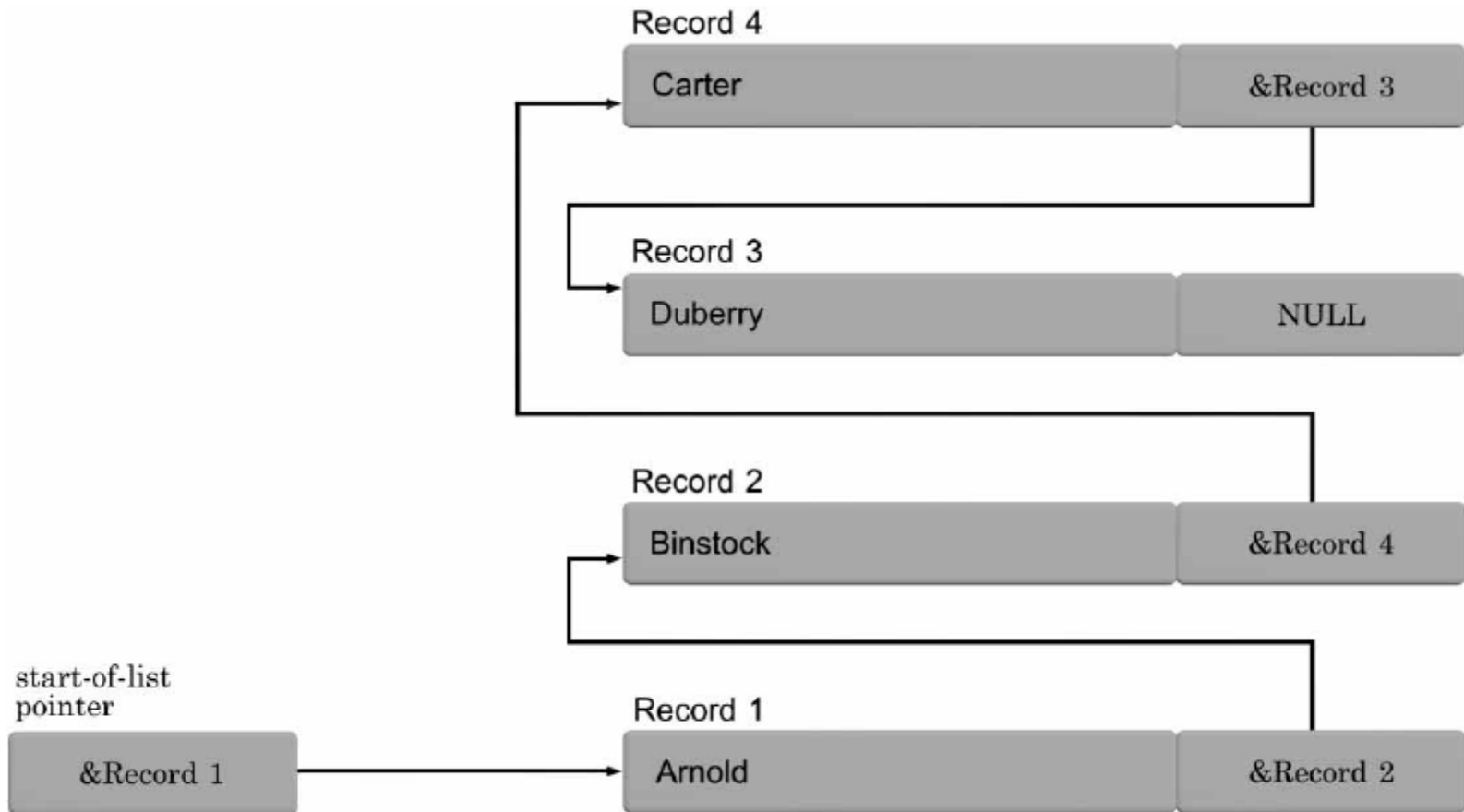


Figure 13.17 Adding a new name to the list

INSERT and DELETE (continued)

LINEAR LOCATION for INSERTING a NEW STRUCTURE

If the key field of the new structure is less than the first structure's key field the new structure should be the new first structure

Else

While there are still more structures in the list

Compare the new structure's key value to each structure key

Stop the comparison when the new structure key either falls between two existing structures or belongs at the end of the existing list

EndWhile

EndIf

INSERT and DELETE (continued)



Program 13.7

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAXCHARS 30
5  #define DEBUG 0
6
7  /* here is the declaration of a linked list structure */
8  struct NameRec
9  {
10     char name[MAXCHARS];
11     struct NameRec *nextAddr;
12 };
13
14 /* here is the definition of the first structure pointer */
15 struct NameRec *firstRec;
17 int main()
18 {
19     void readInsert(); /* function prototypes */
20     void display();
21
22     firstRec = NULL; /* initialize list pointer */
23     readInsert();
24     display();
25
26     return 0;
27 }
```

See [Slide 50](#)

See [Slide 50](#)

INSERT and DELETE (continued)

```
47 void insert(char *name)
48 {
49     struct NameRec *linear Locate(char *); /* function prototype */
50     struct NameRec *newaddr, *here; /* pointers to structure
51                                     of type NameRec */
52
53
54     newaddr = (struct NameRec *) malloc(sizeof(struct NameRec));
55     if (newaddr == (struct NameRec *) NULL) /* check the address */
56     {
57         printf("\nCould not allocate the requested space\n");
58         exit(1);
59     }
60
61     /* locate where the new structure should be placed and */
62     /* update all pointer members */
63     if (firstRec == NULL) /* no list currently exists */
64     {
65         newaddr->nextAddr = NULL;
66         firstRec = newaddr;
67     }
68     else if (strcmp(name, firstRec->name) < 0) /* a new first structure */
69     {
70         newaddr->nextAddr = firstRec;
71         firstRec = newaddr;
72     }
73     else /* structure is not the first structure of the list */
74     {
75         here = linear Locate(name);
76         newaddr->nextAddr = here->nextAddr;
77         here->nextAddr = newaddr;
78     }
79
80     strcpy(newaddr->name, name); /* store the name */
81 }
```


INSERT and DELETE (continued)

```
83  /* This function locates the address of where a new structure
84     should be inserted within an existing list.
85     It receives the address of a name and returns the address of a
86     structure of type NameRec
87  */
88  struct NameRec *linear Locate(char *name)
89  {
90     struct NameRec *one, *two;
91     one = firstRec;
92     two = one->nextAddr;
93
94     if (two == NULL)
95         return(one); /* new structure goes after the existing single structure */
96     while(1)
97     {
98         if(strcmp(name,two->name) < 0) /* if it is located within the list */
99             break;
100        else if(two->nextAddr == NULL) /* it goes after the last structure */
101            {
102                one = two;
103                break;
104            }
105        else /* more structures to search against */
106            {
107                one = two;
108                two = one->nextAddr;
109            }
110    } /* the break takes us here */
111
112    return(one);
113 }
```

INSERT and DELETE (continued)

```
29  /* get a name and insert it into the linked list */
30  void readInsert()
31  {
32      char name[MAXCHARS];
33      void insert(char *);
34
35      printf("\nEnter as many names as you wish, one per line");
36      printf("\nTo stop entering names, enter a single x\n");
37      while (1)
38      {
39          printf("Enter a name: ");
40          gets(name);
41          if (strcmp(name, "x") == 0)
42              break;
43          insert(name);
44      }
45  }
```

...(code for insert() and locate() goes here)...

```
114 /* display names from the linked list */
115 void display()
116 {
117     struct NameRec *contents;
118
119     contents = firstRec;
120     printf("\nThe names currently in the list, in alphabetical");
121     printf("\norder, are:\n");
122     while (contents != NULL) /* display till end of list */
123     {
124         printf("%s\n", contents->name);
125         contents = contents->nextAddr;
126     }
127 }
```

INSERT and DELETE (continued)

- The following sample run shows the results of these tests:

```
Enter as many names as you wish, one per line
To stop entering names, enter a single x
Enter a name: Binstock
Enter a name: Arnold
Enter a name: Duberry
Enter a name: Carter
Enter a name: x
```

```
The names currently in the list, in alphabetical
order, are:
Arnold
Binstock
Carter
Duberry
```

Common Programming Errors

- Not checking the return codes provided by `malloc()` and `realloc()`
- Not correctly updating all relevant pointer addresses when adding or removing structures from dynamically created stacks, queues, and linked lists
- Forgetting to free previously allocated memory space when the space is no longer needed

Common Programming Errors (continued)

- Not preserving the integrity of the addresses contained in the top-of-stack pointer, queue-in, queue-out, and list pointer when dealing with a stack, queue, and dynamically linked list, respectively
- Related to the previous error is the equally disastrous one of not correctly updating internal structure pointers when inserting and removing structures from a stack, queue, or dynamically linked list

Common Compiler Errors

Error	Typical Unix-based Compiler Error Message	Typical Windows-based Compiler Error Message
<p>Forgetting to provide <code>malloc()</code> with a memory size argument when allocating memory.</p> <p>For example:</p> <pre>#include <stdlib.h> int main() { int *p; p = (int *)malloc(); return 0; }</pre>	<p>(E) Missing argument(s).</p>	<p>: error: 'malloc' : function does not take 0 arguments</p>
<p>Forgetting to include the <code>stdlib.h</code> header file whenever <code>malloc()</code> is used in a program.</p> <p>For example:</p> <pre>#include <stdio.h> int main() { int *p; p = (int *)malloc(sizeof(int)); return 0; }</pre>	<p>(W) Operation between types "int*" and "int" is not allowed.</p>	<p>:error: 'malloc': identifier not found, even with argument-dependent lookup</p>

Common Compiler Errors (continued)

Error	Typical Unix-based Compiler Error Message	Typical Windows-based Compiler Error Message
<p>Forgetting to pass a pointer argument to the <code>free()</code> function. For example:</p> <pre>#include <stdio.h> int main() { int *p; p = (int *)malloc(sizeof(int)); free(); return 0; }</pre>	<p>(E) Missing argument(s).</p>	<pre>: error: 'free' : function does not take 0 arguments</pre>
<p>Forgetting to include the indirection operator when creating a pointer member to the structure. For example:</p> <pre>struct NR { char name[30]; struct NR nameRec; }; rather than struct NR { char name[30]; struct NR *nameRec; };</pre>	<p>(E) "struct NR" uses "struct NR" in its definition.</p>	<pre>:error C2460: 'NR::nameRec' : uses 'NR', which is being defined</pre>

Summary

- An alternative to fixed memory allocation for variables at compile time is the dynamic allocation of memory at run time
- `malloc()` reserves a requested number of bytes and returns a pointer to the first reserved byte
- `realloc()` operates in a similar fashion as `malloc()` except it is used to expand or contract an existing allocated space
- `free()` is used to deallocate previously allocated memory space

Summary (continued)

- A stack is a list consisting of structures that can only be added and removed from the top of the list
- A queue is a list consisting of structures that are added to the top of the list and removed from the bottom of the list
- A dynamically linked list consists of structures that can be added or removed from any position in the list