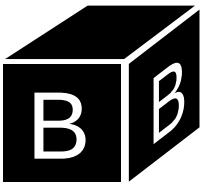


Chapter *21*

Linked Lists

```
MODULE Pbox21A;
   IMPORT StdLog;

   PROCEDURE PointerExample1*;
      TYPE
         Node = RECORD
            i: INTEGER;
            x: REAL
         END;
      VAR
         a: POINTER TO Node;
      BEGIN
         NEW(a);
         a.i := 6;
         a.x := 15.2;
         StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
         StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
      END PointerExample1;

END Pbox21A.
```

Figure 21.1
A program that illustrates the Component Pascal pointer type.

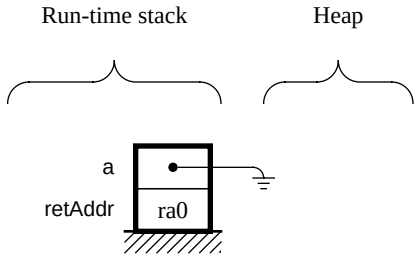
The procedure NEW is a standard Component Pascal procedure that does two things:

The two actions of NEW

- It allocates storage from the heap. Because a was previously declared to be a pointer to a record with an integer and a real component, NEW(a) allocates enough memory to store a record with those components.
- It assigns to a the location of this newly allocated storage. So a now points to the location of a record.

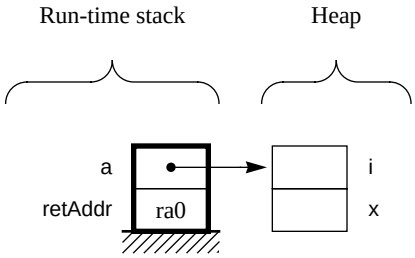
```
PROCEDURE PointerExample1*;
   TYPE
      Node = RECORD
         i: INTEGER;
         x: REAL
      END;
   VAR
      a: POINTER TO Node;
   BEGIN
      NEW(a);
      a.i := 6;
      a.x := 15.2;
      StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
      StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
   END PointerExample1;
```

Log



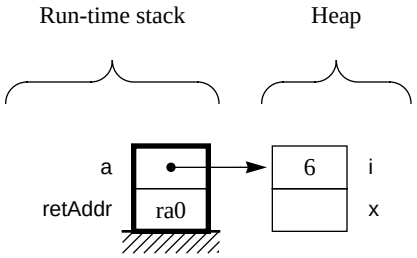
```
PROCEDURE PointerExample1*;
   TYPE
      Node = RECORD
         i: INTEGER;
         x: REAL
      END;
   VAR
      a: POINTER TO Node;
   BEGIN
      NEW(a);
      a.i := 6;
      a.x := 15.2;
      StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
      StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
   END PointerExample1;
```

Log



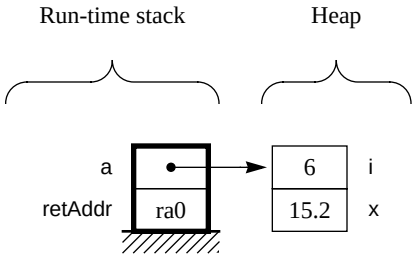
```
PROCEDURE PointerExample1*;
    TYPE
        Node = RECORD
            i: INTEGER;
            x: REAL
        END;
    VAR
        a: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 6;
        a.x := 15.2;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
    END PointerExample1;
```

Log



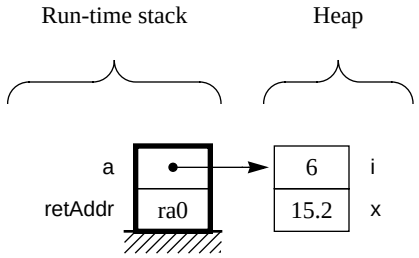
```
PROCEDURE PointerExample1*;
    TYPE
        Node = RECORD
            i: INTEGER;
            x: REAL
        END;
    VAR
        a: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 6;
        a.x := 15.2;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
    END PointerExample1;
```

Log



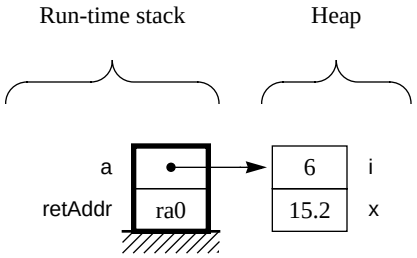
```
PROCEDURE PointerExample1*;
   TYPE
      Node = RECORD
         i: INTEGER;
         x: REAL
      END;
   VAR
      a: POINTER TO Node;
   BEGIN
      NEW(a);
      a.i := 6;
      a.x := 15.2;
      StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
      StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
   END PointerExample1;
```

Log
a.i = 6



```
PROCEDURE PointerExample1*;
   TYPE
      Node = RECORD
         i: INTEGER;
         x: REAL
      END;
   VAR
      a: POINTER TO Node;
   BEGIN
      NEW(a);
      a.i := 6;
      a.x := 15.2;
      StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
      StdLog.String("a.x = "); StdLog.Real(a.x); StdLog.Ln
   END PointerExample1;
```

Log
a.i = 6
a.x = 15.2



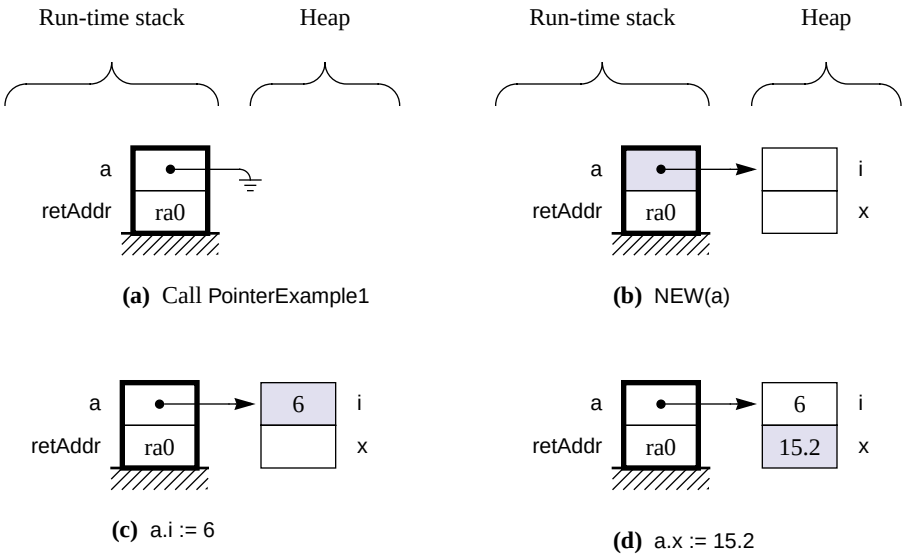


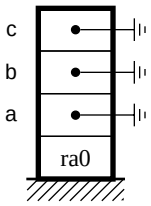
Figure 21.2
The trace of the procedure in Figure 21.1.

```
MODULE Pbox21B;
   IMPORT StdLog;

   PROCEDURE PointerExample2*;
      TYPE
         Node = RECORD
            i: INTEGER
         END;
      VAR
         a, b, c: POINTER TO Node;
      BEGIN
         NEW(a); a.i := 5;
         NEW(b); b.i := 3;
         c := a;
         a := b;
         a.i := 2 + c.i;
         StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
         StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
         StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
      END PointerExample2;

END Pbox21B.
```

Figure 21.3
The effect of the assignment operation on pointers.

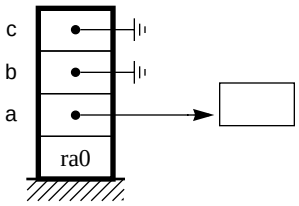
```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log



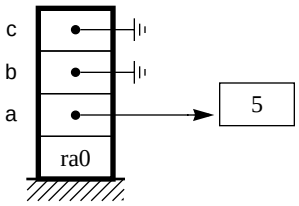
```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log



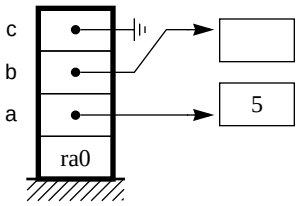
```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log



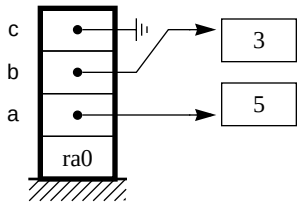
```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log



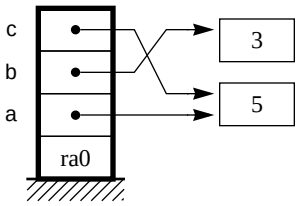
```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log



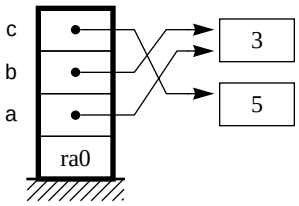
```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log



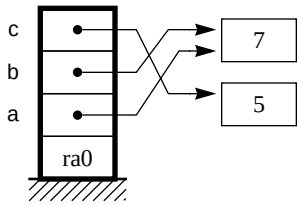
```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log



```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

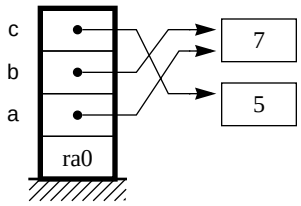
Log



```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log

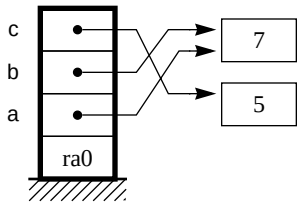
```
a.i = 7
```



```
PROCEDURE PointerExample2*;
   TYPE
      Node = RECORD
         i: INTEGER
      END;
   VAR
      a, b, c: POINTER TO Node;
   BEGIN
      NEW(a);
      a.i := 5;
      NEW(b);
      b.i := 3;
      c := a;
      a := b;
      a.i := 2 + c.i;
      StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
      StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
      StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
   END PointerExample2;
```

Log

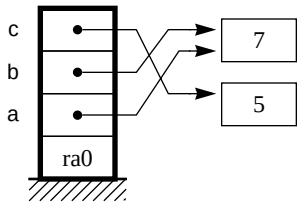
```
a.i = 7
b.i = 7
```

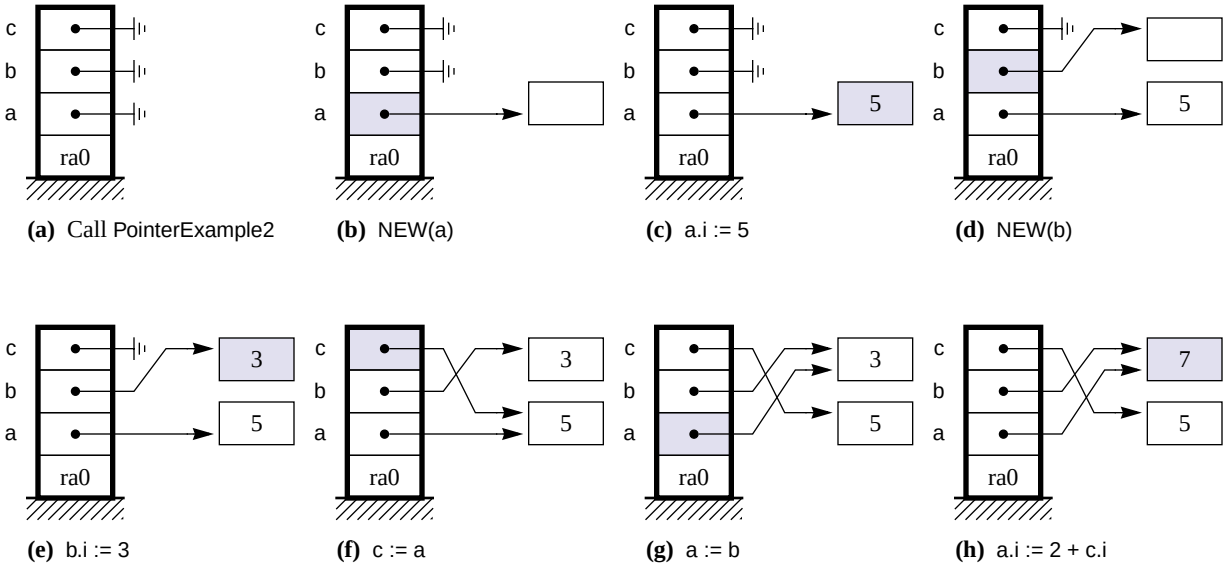


```
PROCEDURE PointerExample2*;
    TYPE
        Node = RECORD
            i: INTEGER
        END;
    VAR
        a, b, c: POINTER TO Node;
    BEGIN
        NEW(a);
        a.i := 5;
        NEW(b);
        b.i := 3;
        c := a;
        a := b;
        a.i := 2 + c.i;
        StdLog.String("a.i = "); StdLog.Int(a.i); StdLog.Ln;
        StdLog.String("b.i = "); StdLog.Int(b.i); StdLog.Ln;
        StdLog.String("c.i = "); StdLog.Int(c.i); StdLog.Ln
    END PointerExample2;
```

Log

```
a.i = 7
b.i = 7
c.i = 5
```





(a) Call PointerExample2 **(b)** NEW(a) **(c)** a.i := 5 **(d)** NEW(b)

(e) b.i := 3 **(f)** c := a **(g)** a := b **(h)** a.i := 2 + c.i

Figure 21.4
A trace of Figure 21.3.

In general, if pointer p points to a record r that contains field f, then

- p is a pointer
- p^ is the record r to which p points
- p^.f is field f of the record r to which p points.

The notation p.f is an abbreviation for the longer notation p^.f.

The period abbreviation for pointers to records

```
TYPE
    List = POINTER TO Node;
    Node = RECORD
        value: REAL;
        next: List
    END;
```

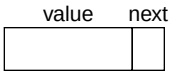
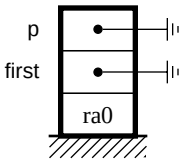


Figure 21.5
The structure of a record of type Node in Figure 21.6.

```
MODULE Pbox21C;
   IMPORT StdLog;

   PROCEDURE LinkedListExample*;
      TYPE
         List = POINTER TO Node;
         Node = RECORD
            value: REAL;
            next: List
         END;
      VAR
         first, p: List;
      BEGIN
         (* Create linked list *)
         NEW(first); first.value := 4.5;
         p := first;
         NEW(first); first.value := 1.2;
         first.next := p; p := first;
         NEW(first); first.value := 7.3;
         first.next := p;
         (* Output linked list *)
         p := first;
         StdLog.Real(p.value); StdLog.String("  ");
         p := p.next;
         StdLog.Real(p.value); StdLog.String("  ");
         p := p.next;
         StdLog.Real(p.value); StdLog.String("  ")
      END LinkedListExample;

END Pbox21C.
```

Figure 21.6
A program that constructs a linked list of three real numbers.

```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```



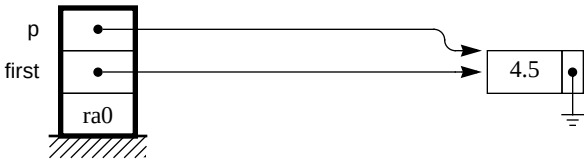
Log

```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log

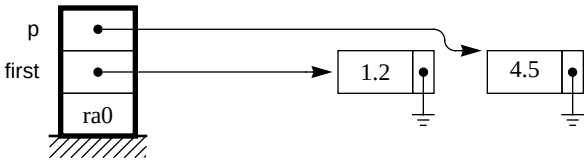
```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log



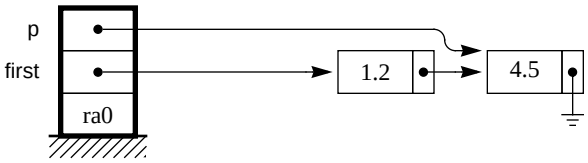
```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log



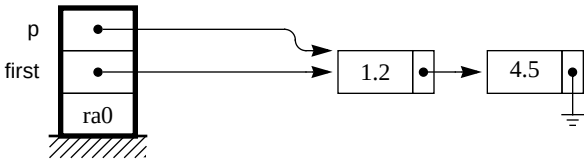
```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log



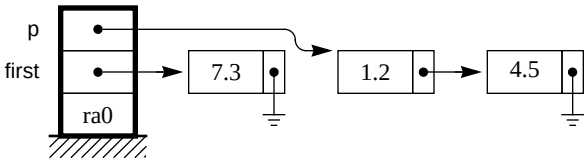
```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log



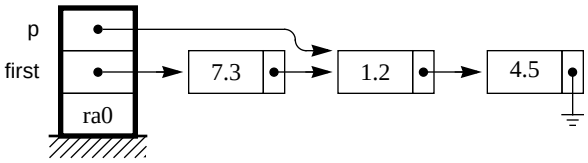
```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log



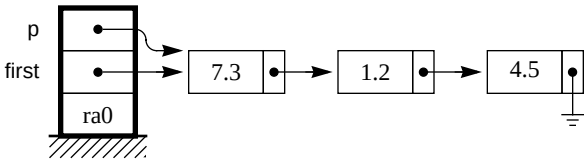
```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log



```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

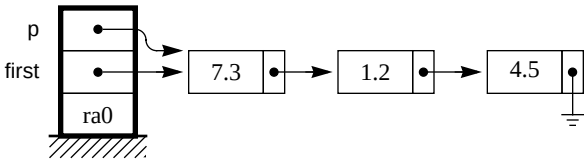
Log



```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log

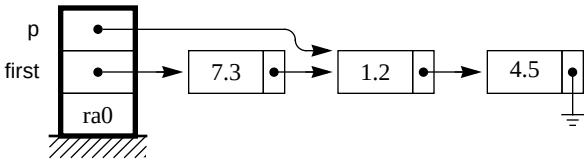
```
7.3
```



```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log

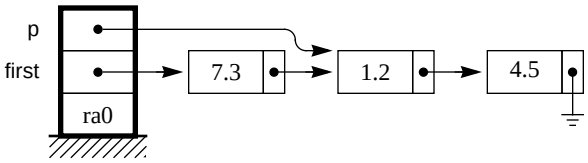
```
7.3
```



```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log

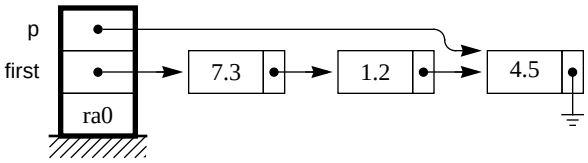
```
7.3  1.2
```



```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

Log

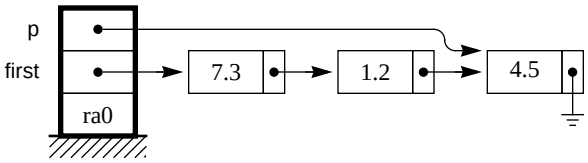
```
7.3  1.2
```

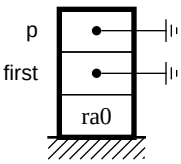


```
BEGIN
    (* Create linked list *)
    NEW(first); first.value := 4.5;
    p := first;
    NEW(first); first.value := 1.2;
    first.next := p;
    p := first;
    NEW(first); first.value := 7.3;
    first.next := p;
    (* Output linked list *)
    p := first;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next;
    StdLog.Real(p.value); StdLog.String("  ")
END LinkedListExample;
```

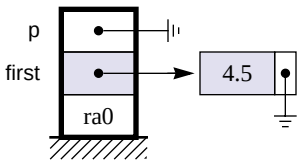
Log

```
7.3  1.2  4.5
```

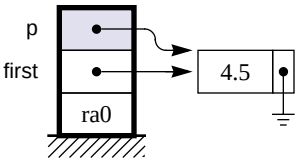




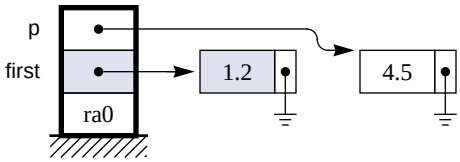
(a) Call LinkedListExample



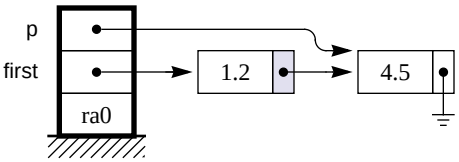
(b) NEW(first); first.value := 4.5



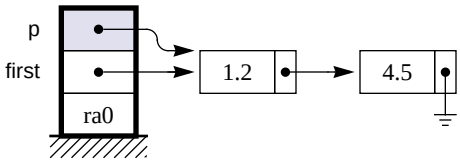
(c) p := first



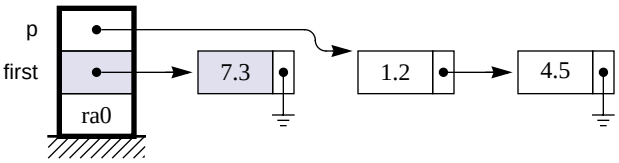
(d) NEW(first); first.value := 1.2



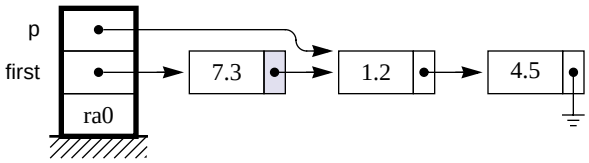
(e) first.next := p



(f) p := first

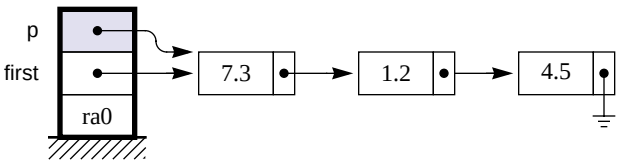


(g) NEW(first); first.value := 7.3

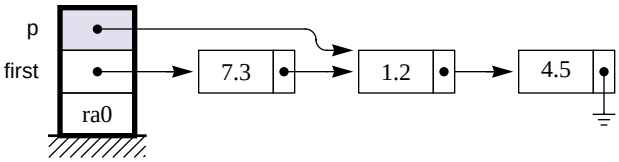


(h) first.next := p

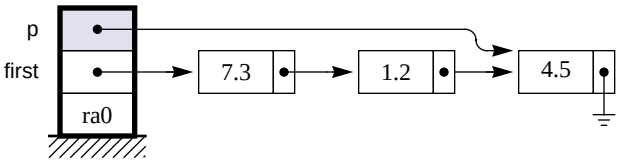
Figure 21.7
The trace of Figure 21.6 to create a linked list.



(a) p := first



(b) p := p.next



(c) p := p.next

Figure 21.8
The trace of the procedure in Figure 21.6 to output a linked list.

```
p := first;
WHILE p # NIL DO
    StdLog.Real(p.value); StdLog.String("  ");
    p := p.next
END
```

```
TYPE
    AlphaPtr = POINTER TO Alpha;
    BetaPtr = POINTER TO Beta;
    GammaPtr = POINTER TO Gamma;
VAR
    alphaPtr: AlphaPtr;
    betaPtr: BetaPtr;
    gammaPtr: GammaPtr;
```

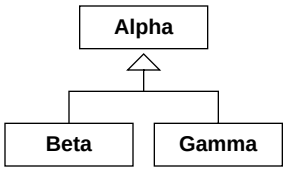


Figure 21.9
The object-oriented relationship of inheritance between Alpha, Beta, and Gamma.

You can assign the specific to the general, but you cannot assign the general to the specific.

The fundamental class assignment rule

```
betaPtr := alphaPtr

alphaPtr := betaPtr
```

You can assign the specific to the general, but you cannot assign the general to the specific.

The fundamental class assignment rule

betaPtr := alphaPtr Not legal

alphaPtr := betaPtr

You can assign the specific to the general, but you cannot assign the general to the specific.

The fundamental class assignment rule

betaPtr := alphaPtr	Not legal
alphaPtr := betaPtr	Legal

```
DEFINITION PboxCListADT;

    TYPE
        CList = POINTER TO Node;

    PROCEDURE Clear (OUT lst: CList);
    PROCEDURE Empty (lst: CList): BOOLEAN;
    PROCEDURE GoNext (VAR lst: CList);
    PROCEDURE Insert (VAR lst: CList; val: POINTER TO ANYREC);
    PROCEDURE NodeContent (lst: CList): POINTER TO ANYREC;

END PboxCListADT.
```

Figure 21.10
The interface of the circular list abstract data type.

PROCEDURE **Clear** (OUT lst: CList)
post
lst is cleared to the empty list.

PboxCListADT.Clear(myList)

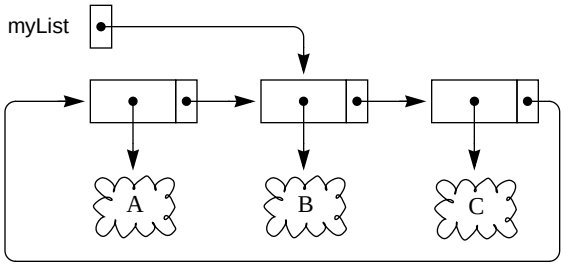


Figure 21.11
The result of calling procedure Clear.

PROCEDURE **Clear** (OUT lst: CList)
post
lst is cleared to the empty list.

PboxCListADT.Clear(myList)

Figure 21.11
The result of calling procedure Clear.

PROCEDURE **Empty** (lst: CList): BOOLEAN
post
Returns TRUE if lst is empty. Otherwise returns FALSE.

PROCEDURE **GoNext** (VAR lst: CList)
pre
lst is not empty. 20
post
The next location in lst is designated as the current item.

PboxCListADT.GoNext(myList)

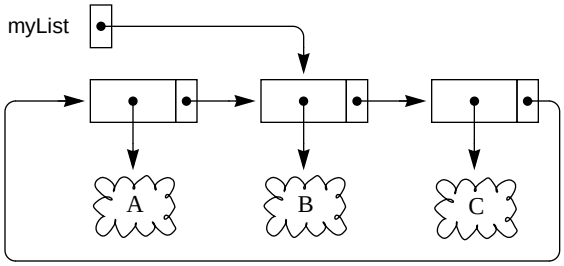


Figure 21.12
The result of calling procedure GoNext.

PROCEDURE **GoNext** (VAR lst: CList)
pre
lst is not empty. 20
post
The next location in lst is designated as the current item.

PboxCListADT.GoNext(myList)

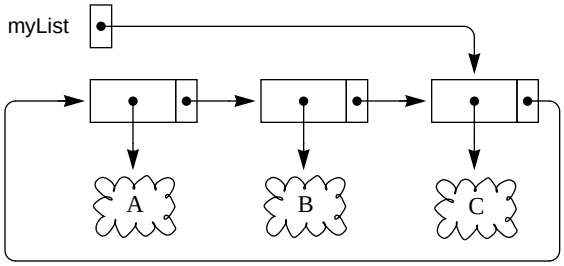


Figure 21.12
The result of calling procedure GoNext.

PROCEDURE **Insert** (VAR lst: CList; val: POINTER TO ANYREC)
post
Value val is inserted in lst after the current item, and it becomes the current item.

PboxCListADT.Insert(myList, myPtr)

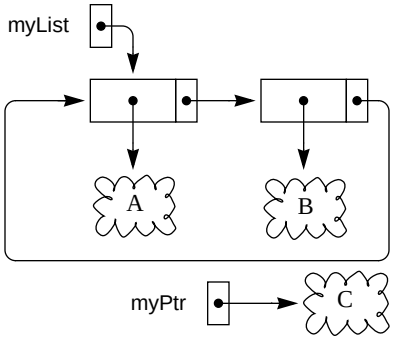


Figure 21.13
The result of calling procedure Insert.

PROCEDURE **Insert** (VAR lst: CList; val: POINTER TO ANYREC)
post
Value val is inserted in lst after the current item, and it becomes the current item.

PboxCListADT.Insert(myList, myPtr)

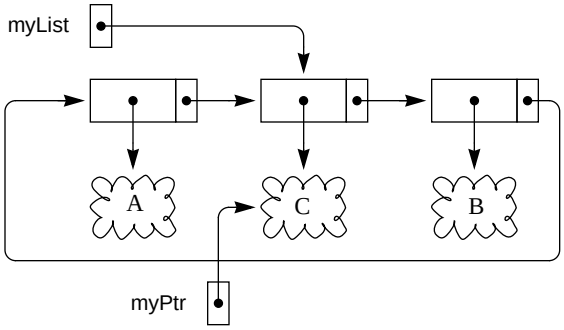


Figure 21.13
The result of calling procedure Insert.

PROCEDURE **NodeContent** (lst: CList): POINTER TO ANYREC
pre
lst is not empty. 20
post
Returns the content from the current item of lst.

```
myPtr := PboxCListADT.NodeContent(myList) (MyPtr)
```

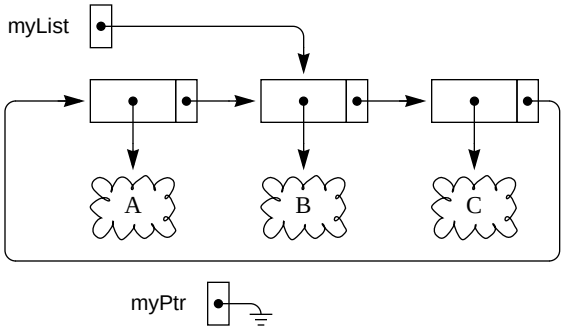


Figure 21.14
The result of calling procedure NodeContent when myList initially has the structure of Figure 21.11(a).

PROCEDURE **NodeContent** (lst: CList): POINTER TO ANYREC
pre
lst is not empty. 20
post
Returns the content from the current item of lst.

myPtr := PboxCListADT.NodeContent(myList) (MyPtr)

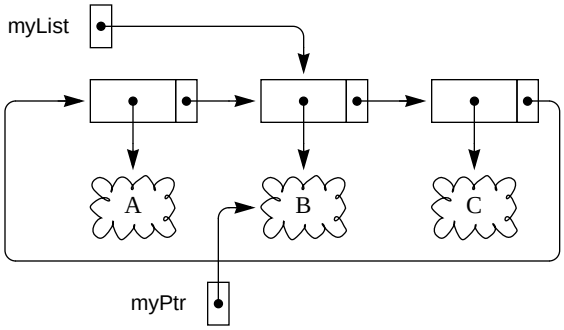


Figure 21.14
The result of calling procedure NodeContent when myList initially has the structure of Figure 21.11(a).

```
TYPE
    String64 = ARRAY 64 OF CHAR;
    Book = POINTER TO RECORD
        title: String64;
        author: String64;
        price: REAL
    END;
```

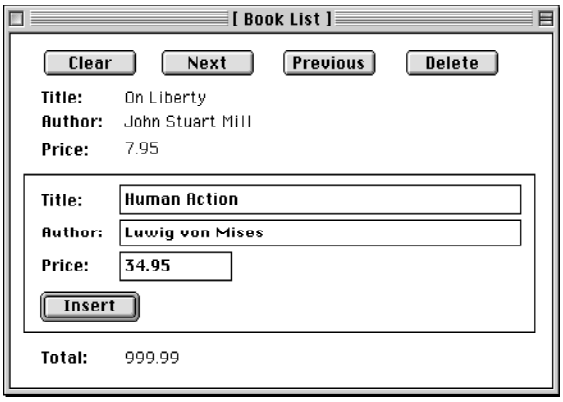


Figure 21.15
The dialog box for a program that uses the circular list from module PboxCListADT.

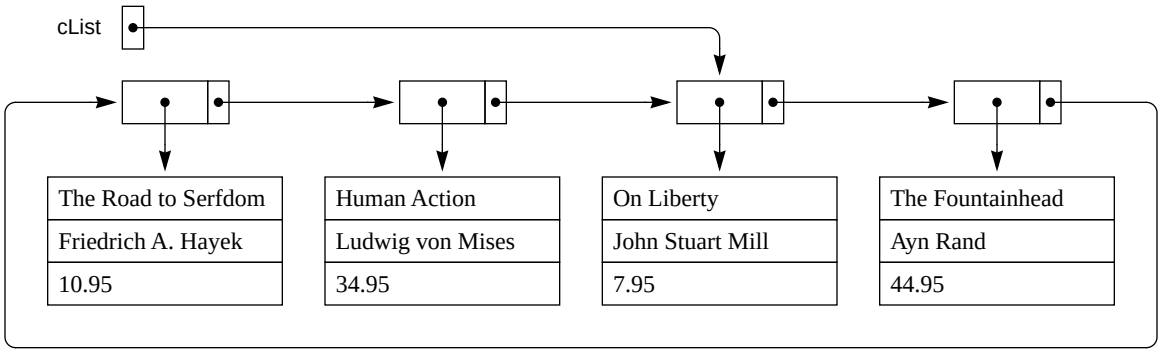


Figure 21.16
The circular list corresponding to the dialog box in Figure 21.15.

```
MODULE Pbox21D;
    IMPORT Dialog, PboxCListADT;

    TYPE
        String64 = ARRAY 64 OF CHAR;
        Book = POINTER TO RECORD
            title: String64;
            author: String64;
            price: REAL
        END;

    VAR
        d*: RECORD
            titleOut-, authorOut- : String64;
            priceOut-: REAL;
            titleIn*, authorIn* : String64;
            priceIn*: REAL;
            total-: REAL
        END;
        cList: PboxCListADT.CList;
```

Figure 21.17
The program that implements the dialog box of Figure 21.15.

```
PROCEDURE ClearDialog;
BEGIN
    d.titleOut := ""; d.authorOut := ""; d.priceOut := 0.0;
    d.titleIn := ""; d.authorIn := ""; d.priceIn := 0.0;
    d.total := 0.0
END ClearDialog;

PROCEDURE SetBookOut (b: Book);
BEGIN
    d.titleOut := b.title;
    d.authorOut := b.author;
    d.priceOut := b.price
END SetBookOut;

PROCEDURE SetTotal;
BEGIN
    (* A problem for the student *)
    d.total := 999.99
END SetTotal;
```

```
PROCEDURE Clear*;
BEGIN
    ClearDialog;
    PboxCListADT.Clear(cList);
    Dialog.Update(d)
END Clear;

PROCEDURE Next*;
    VAR
        book: Book;
BEGIN
    IF ~PboxCListADT.Empty(cList) THEN
        PboxCListADT.GoNext(cList);
        book := PboxCListADT.NodeContent(cList) (Book);
        SetBookOut(book);
        Dialog.Update(d)
    END
END Next;
```

```
PROCEDURE Previous*;
BEGIN
   (* A problem for the student *)
END Previous;

PROCEDURE Delete*;
BEGIN
   (* A problem for the student *)
END Delete;
```

```
    PROCEDURE Insert*;
        VAR
            book: Book;
    BEGIN
        NEW(book);
        book.title := d.titleIn;
        book.author := d.authorIn;
        book.price := d.priceIn;
        PboxCListADT.Insert(cList, book);
        SetBookOut(book);
        SetTotal;
        Dialog.Update(d)
    END Insert;

BEGIN
    Clear
END Pbox21D.
```

```
MODULE  PboxCListADT;

    TYPE
        CList* = POINTER TO Node;
        Node = RECORD
            value: POINTER TO ANYREC;
            next: CList
        END;

    PROCEDURE Clear* (OUT lst: CList);
    BEGIN
        lst := NIL
    END Clear;

    PROCEDURE Empty* (lst: CList): BOOLEAN;
    BEGIN
        RETURN lst = NIL
    END Empty;
```

Figure 21.18
Implementation of the circular list PboxCListADT.CList.

```
PROCEDURE GoNext* (VAR lst: CList);
BEGIN
   ASSERT (lst # NIL, 20);
   lst := lst.next
END GoNext;

PROCEDURE NodeContent* (lst: CList): POINTER TO ANYREC;
BEGIN
   ASSERT (lst # NIL, 20);
   RETURN lst.value
END NodeContent;
```

```
    PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
        VAR
            temp: CList;
    BEGIN
        IF lst = NIL THEN
            NEW(lst);
            lst.value := val;
            lst.next := lst
        ELSE
            temp := lst.next;
            NEW(lst.next);
            lst := lst.next;
            lst.value := val;
            lst.next := temp
        END
    END Insert;

END PboxCListADT.
```

```
PROCEDURE Clear* (OUT lst: CList);
BEGIN
    lst := NIL
END Clear;
```

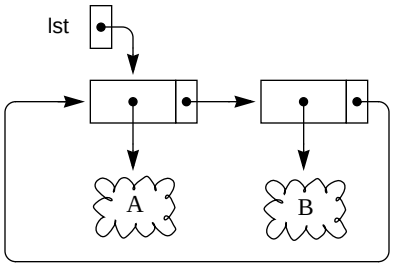


Figure 21.19
Implementation of procedure Clear with automatic garbage collection.

```
PROCEDURE Clear* (OUT lst: CList);
BEGIN
    lst := NIL
END Clear;
```

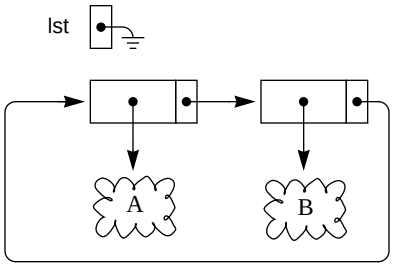


Figure 21.19
Implementation of procedure Clear with automatic garbage collection.

```
PROCEDURE Clear* (OUT lst: CList);
BEGIN
    lst := NIL
END Clear;
```

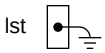
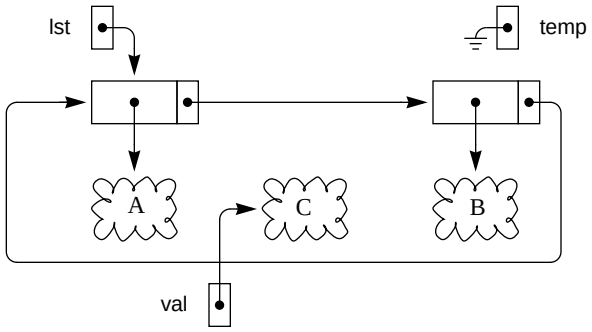


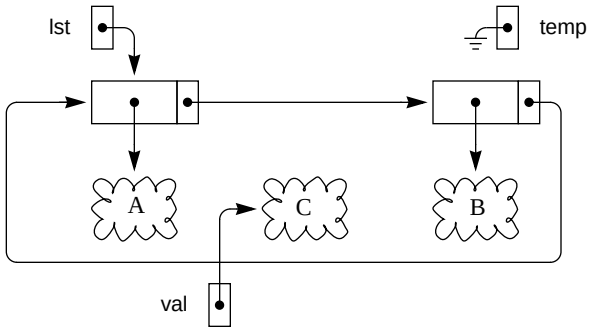
Figure 21.19
Implementation of procedure Clear with automatic garbage collection.

Figure 21.20
Execution of procedure PboxCListADT.Insert with a nonempty list.

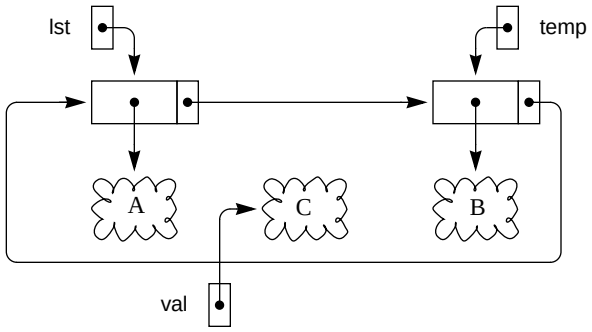
```
PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
    VAR
        temp: CList;
BEGIN
    IF lst = NIL THEN
        NEW(lst);
        lst.value := val;
        lst.next := lst
    ELSE
        temp := lst.next;
        NEW(lst.next);
        lst := lst.next;
        lst.value := val;
        lst.next := temp
    END
END Insert;
```



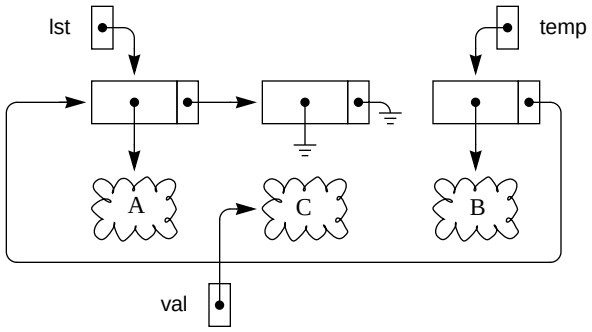
```
PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
   VAR
      temp: CList;
BEGIN
   IF lst = NIL THEN
      NEW(lst);
      lst.value := val;
      lst.next := lst
   ELSE
      temp := lst.next;
      NEW(lst.next);
      lst := lst.next;
      lst.value := val;
      lst.next := temp
   END
END Insert;
```



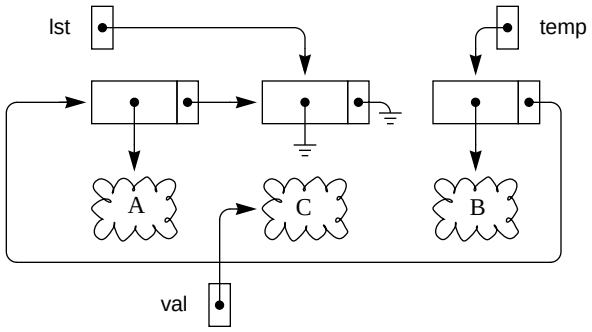
```
PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
    VAR
        temp: CList;
BEGIN
    IF lst = NIL THEN
        NEW(lst);
        lst.value := val;
        lst.next := lst
    ELSE
        temp := lst.next;
        NEW(lst.next);
        lst := lst.next;
        lst.value := val;
        lst.next := temp
    END
END Insert;
```



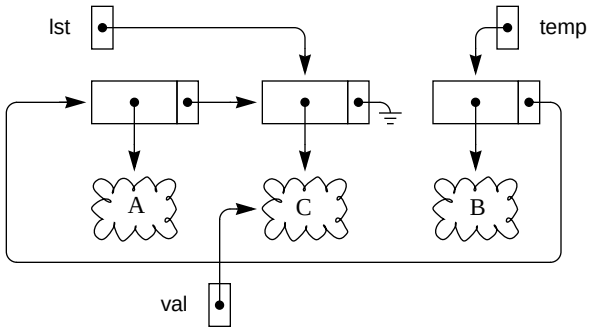
```
PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
    VAR
        temp: CList;
BEGIN
    IF lst = NIL THEN
        NEW(lst);
        lst.value := val;
        lst.next := lst
    ELSE
        temp := lst.next;
        NEW(lst.next);
        lst := lst.next;
        lst.value := val;
        lst.next := temp
    END
END Insert;
```



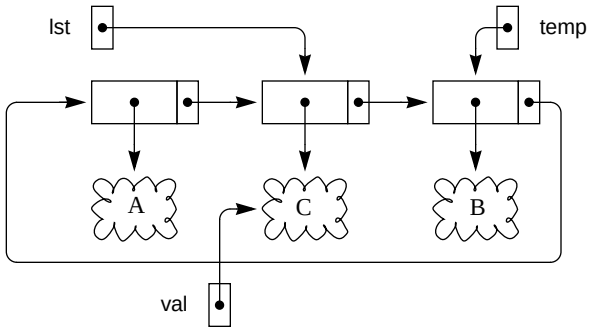
```
PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
    VAR
        temp: CList;
BEGIN
    IF lst = NIL THEN
        NEW(lst);
        lst.value := val;
        lst.next := lst
    ELSE
        temp := lst.next;
        NEW(lst.next);
        lst := lst.next;
        lst.value := val;
        lst.next := temp
    END
END Insert;
```



```
PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
    VAR
        temp: CList;
BEGIN
    IF lst = NIL THEN
        NEW(lst);
        lst.value := val;
        lst.next := lst
    ELSE
        temp := lst.next;
        NEW(lst.next);
        lst := lst.next;
        lst.value := val;
        lst.next := temp
    END
END Insert;
```



```
PROCEDURE Insert* (VAR lst: CList; val: POINTER TO ANYREC);
    VAR
        temp: CList;
BEGIN
    IF lst = NIL THEN
        NEW(lst);
        lst.value := val;
        lst.next := lst
    ELSE
        temp := lst.next;
        NEW(lst.next);
        lst := lst.next;
        lst.value := val;
        lst.next := temp
    END
END Insert;
```



```
Node = RECORD
    prev: CList;
    value: POINTER TO ANYREC;
    next: CList
END;
```

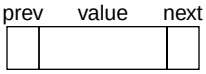


Figure 21.21
The structure of a record of type Node for a doubly-linked list.

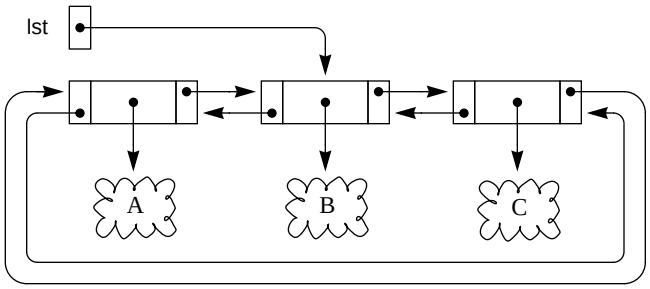


Figure 21.22
A circular doubly-linked list that corresponds to the singly-linked list of Figure 21.11

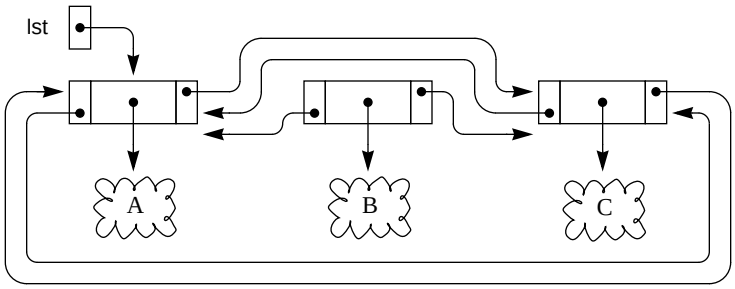


Figure 21.23
Deleting a node from a doubly linked circular list with the list initially as in Figure 21.22.

```
TYPE
   Composer = RECORD
      name: ARRAY 32 OF CHAR;
      birthYear: INTEGER
   END;
VAR
   composerA, composerB: Composer;

   composerA := composerB
```

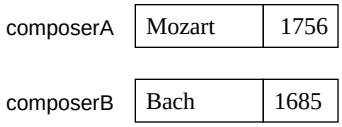


Figure 21.24
Record assignment.

```
TYPE
   Composer = RECORD
      name: ARRAY 32 OF CHAR;
      birthYear: INTEGER
   END;
VAR
   composerA, composerB: Composer;
```

composerA := composerB

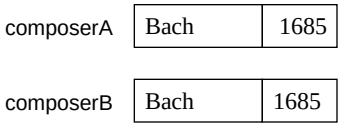


Figure 21.24
Record assignment.

```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```

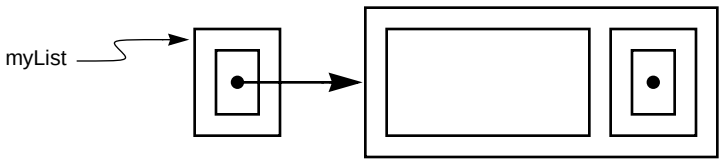


Figure 21.25
Diagrams for the node structure of List.

(a) Node structure for List.

```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```

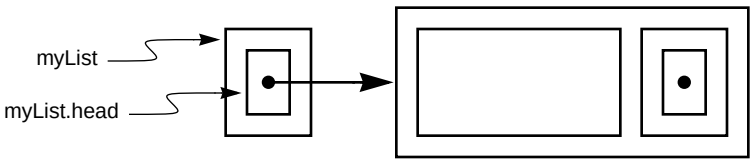


Figure 21.25
Diagrams for the node structure of List.

(a) Node structure for List.

```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```

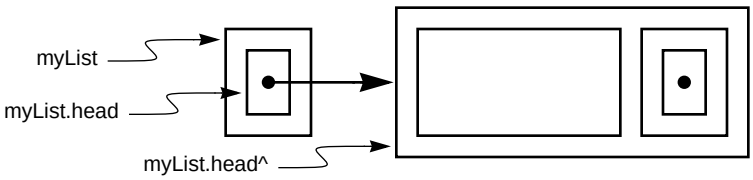
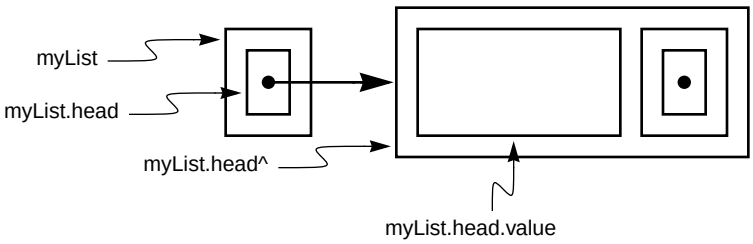


Figure 21.25
Diagrams for the node structure of List.

(a) Node structure for List.

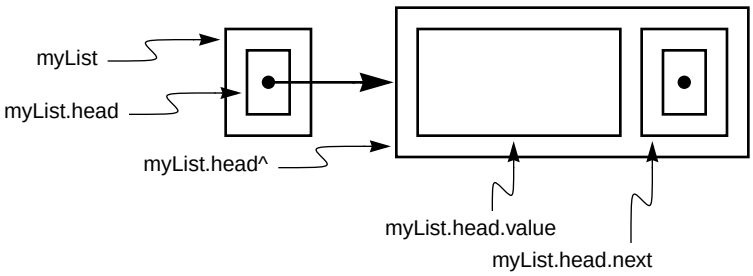
```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```



(a) Node structure for List.

Figure 21.25
Diagrams for the node structure of List.

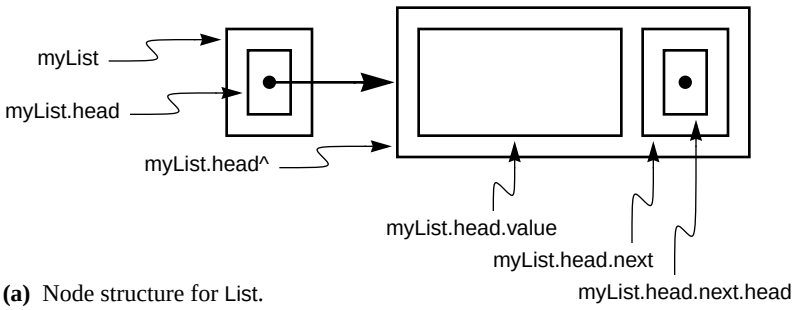
```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```



(a) Node structure for List.

Figure 21.25
Diagrams for the node structure of List.

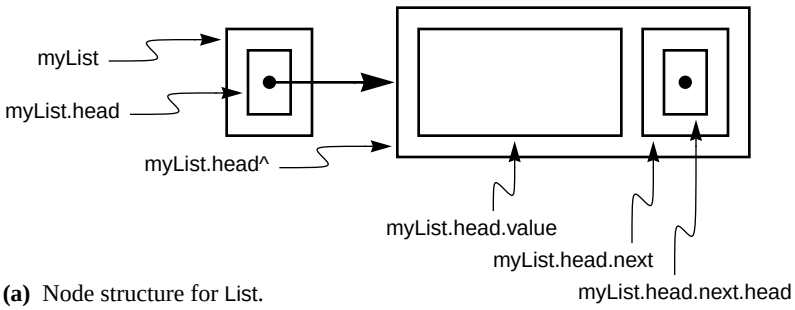
```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```



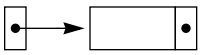
(a) Node structure for List.

Figure 21.25
Diagrams for the node structure of List.

```
List* = RECORD
    head: POINTER TO Node
END;
Node = RECORD
    value: T;
    next: List
END;
```



(a) Node structure for List.



(b) Abbreviated diagram of the same structure as in (a).

Figure 21.25
Diagrams for the node structure of List.

```
DEFINITION PboxLListObj;

    TYPE
        T = ARRAY 16 OF CHAR;
        List = RECORD
            (VAR lst: List) Clear, NEW;
            (IN lst: List) Display, NEW;
            (IN lst: List) GetElementN (n: INTEGER; OUT val: T), NEW;
            (VAR lst: List) InsertAtN (n: INTEGER; IN val: T), NEW;
            (IN lst: List) Length (): INTEGER, NEW;
            (VAR lst: List) RemoveN (n: INTEGER), NEW;
            (IN lst: List) Search (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW
        END;

END PboxLListObj.
```

Figure 21.26
The interface of the linked list class PboxLListObj.

TYPE **List**
The linked list class supplied by PboxLListObj.

TYPE **T**
The type of each element in the list, a string of at most 15 characters.

PROCEDURE (VAR lst: List) **Clear**
Post
List lst is initialized to the empty list.

PROCEDURE (IN lst: List) **Display**
Post
List lst is output to the Log, one element per line with each element preceded by its position.

PROCEDURE (IN lst: List) **GetElementN** (n: INTEGER; OUT val: T)
Pre
0 <= n 20
n < lst.Length() 21
Post
val gets the data value of the element at position n of list lst.
Note: 0 is the position of the first element in the list.

Figure 21.27
The documentation of the linked list class PboxLListObj.

PROCEDURE (VAR lst: List) **InsertAtN** (n: INTEGER; IN val: T)
Pre
0 <= n 20
Post
val is inserted at position n in list lst, increasing lst.Length() by 1.
If n > lst.Length(), val is appended to the list.

PROCEDURE (IN lst: List) **Length** (): INTEGER
Post
Returns the number of elements in list lst.

PROCEDURE (VAR lst: List) **RemoveN** (n: INTEGER)
Pre
0 <= n 20
Post
If n < lst.Length(), the element at position n in list lst is removed.
Otherwise, the list is unchanged.

PROCEDURE (IN lst: List) **Search** (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN)
Post
If srchVal is in list lst, fnd is set to TRUE and n is set to the first position where srchVal is found.
Otherwise, fnd is set to FALSE and n is undefined.

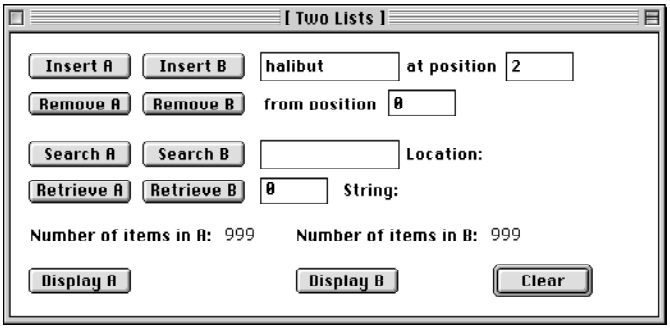


Figure 21.28
The dialog box for manipulating two lists.

```
MODULE Pbox21E;
   IMPORT Dialog, PboxLListObj, PboxStrings;

   TYPE
      String32 = ARRAY 32 OF CHAR;

   VAR
      d*: RECORD
         insertT*: PboxLListObj.T; insertPosition*: INTEGER;
         removePosition*: INTEGER;
         searchT*: PboxLListObj.T; searchPosition-: String32;
         retrievePosition*: INTEGER; retrieveT-: PboxLListObj.T;
         numItemsA-, numItemsB-: INTEGER;
      END;
      listA, listB: PboxLListObj.List;
```

Figure 21.29
The program for the dialog box of Figure 21.28.

```
PROCEDURE InsertAtA*;
BEGIN
   listA.InsertAtN(d.insertPosition, d.insertT);
   d.numItemsA := listA.Length();
   Dialog.Update(d)
END InsertAtA;

PROCEDURE InsertAtB*;
BEGIN
   listB.InsertAtN(d.insertPosition, d.insertT);
   d.numItemsB := listB.Length();
   Dialog.Update(d)
END InsertAtB;
```

```
PROCEDURE RemoveFromA*;
BEGIN
   listA.RemoveN(d.removePosition);
   d.numItemsA := listA.Length();
   Dialog.Update(d)
END RemoveFromA;

PROCEDURE RemoveFromB*;
BEGIN
   listB.RemoveN(d.removePosition);
   d.numItemsB := listB.Length();
   Dialog.Update(d)
END RemoveFromB;
```

```
PROCEDURE SearchForA*;
VAR
    found: BOOLEAN;
    position: INTEGER;
BEGIN
    listA.Search(d.searchT, position, found);
    IF found THEN
        PboxStrings.IntToString(position, 1, d.searchPosition);
        d.searchPosition := "At position " + d.searchPosition + "."
    ELSE
        d.searchPosition := "Not in list."
    END;
    Dialog.Update(d)
END SearchForA;

PROCEDURE SearchForB*;
VAR
    found: BOOLEAN;
    position: INTEGER;
BEGIN
    listB.Search(d.searchT, position, found);
    IF found THEN
        PboxStrings.IntToString(position, 1, d.searchPosition);
        d.searchPosition := "At position " + d.searchPosition + "."
    ELSE
        d.searchPosition := "Not in list."
    END;
    Dialog.Update(d)
END SearchForB;
```

```
PROCEDURE RetrieveFromA*;
BEGIN
   listA.GetElementN(d.retrievePosition, d.retrieveT);
   Dialog.Update(d)
END RetrieveFromA;

PROCEDURE RetrieveFromB*;
BEGIN
   listB.GetElementN(d.retrievePosition, d.retrieveT);
   Dialog.Update(d)
END RetrieveFromB;
```

```
PROCEDURE DisplayListA*;
BEGIN
   listA.Display()
END DisplayListA;

PROCEDURE DisplayListB*;
BEGIN
   listB.Display()
END DisplayListB;
```

```
    PROCEDURE ClearLists*;
    BEGIN
        listA.Clear; listB.Clear;
        d.insertT := ""; d.insertPosition := 0;
        d.removePosition := 0;
        d.searchT := ""; d.searchPosition := "";
        d.retrievePosition := 0; d.retrieveT := "";
        d.numItemsA := 0; d.numItemsB := 0;
        Dialog.Update(d)
    END ClearLists;

BEGIN
    ClearLists
END Pbox21E.
```

```
MODULE  PboxLListObj;
   IMPORT StdLog;

   TYPE
      T* = ARRAY 16 OF CHAR;
      List* = RECORD
         head: POINTER TO Node
      END;
      Node = RECORD
         value: T;
         next: List
      END;

   PROCEDURE (VAR lst: List) Clear*, NEW;
   BEGIN
      lst.head := NIL
   END Clear;
```

Figure 21.30
Implementation of the linked list class that is used in Figure 21.29.

```
PROCEDURE (IN lst: List) Display*, NEW;
    VAR
        p: List;
        i: INTEGER;
BEGIN
    i := 0;
    p := lst;
    WHILE p.head # NIL DO
        StdLog.Int(i); StdLog.String(" "); StdLog.String(p.head.value); StdLog.Ln;
        INC(i);
        p := p.head.next
    END
END Display;
```

```
PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    VAR
        p: List;
        i: INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    p := lst;
    FOR i := 1 TO n DO
        ASSERT(p.head # NIL, 21);
        p := p.head.next
    END;
    ASSERT(p.head # NIL, 21);
    val := p.head.value
END GetElementN;
```

```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;
        i: INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN (* Insert at beginning *)
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```

```
    PROCEDURE (IN lst: List) Length* (): INTEGER, NEW;
    BEGIN
       (* A problem for the student *)
       RETURN 999
    END Length;

    PROCEDURE (VAR lst: List) RemoveN* (n: INTEGER), NEW;
    BEGIN
       (* A problem for the student *)
    END RemoveN;

    PROCEDURE (IN lst: List) Search* (IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN), NEW;
    BEGIN
       (* A problem for the student *)
       fnd := FALSE
    END Search;

END PboxLListObj.
```

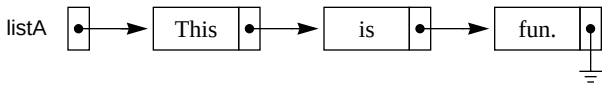
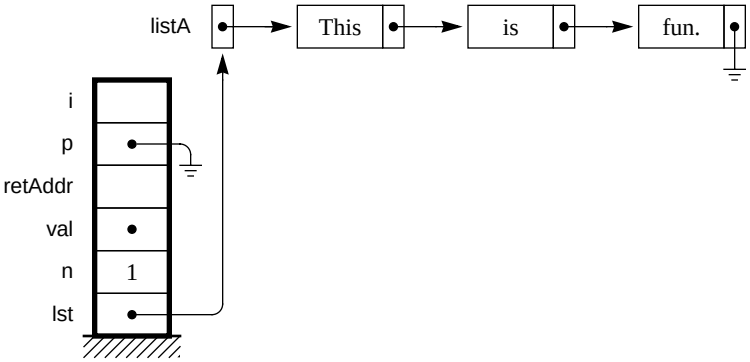


Figure 21.31
Memory allocation when method GetElementN is called.

(a) Before call to GetElementN



(b) Call listA.GetElementN(d.retrievePosition, d.retrieveT)

Figure 21.31
Memory allocation when method GetElementN is called.

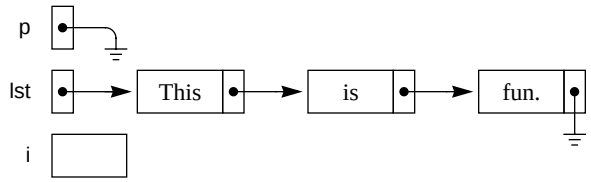
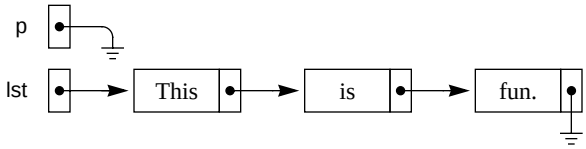


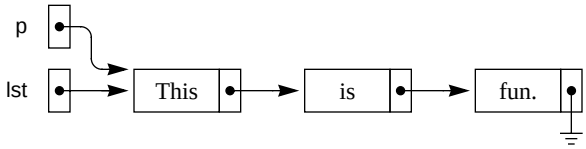
Figure 21.31
Memory allocation when method GetElementN is called.

(c) Abbreviated version of (b)

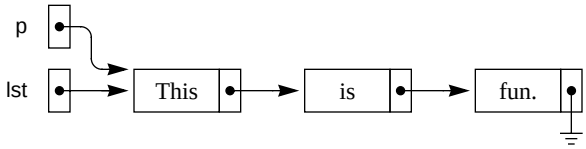
```
PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    VAR
        p: List;                                    n       val     i
        i: INTEGER;                                 1
BEGIN
    ASSERT(0 <= n, 20);
    p := lst;
    FOR i := 1 TO n DO
        ASSERT(p.head # NIL, 21);
        p := p.head.next
    END;
    ASSERT(p.head # NIL, 21);
    val := p.head.value
END GetElementN;
```



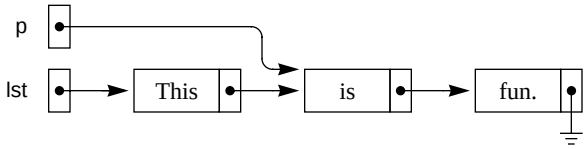
```
PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    VAR
        p: List;                                    n       val     i
        i: INTEGER;                                 1
BEGIN
    ASSERT(0 <= n, 20);
    p := lst;
    FOR i := 1 TO n DO
        ASSERT(p.head # NIL, 21);
        p := p.head.next
    END;
    ASSERT(p.head # NIL, 21);
    val := p.head.value
END GetElementN;
```



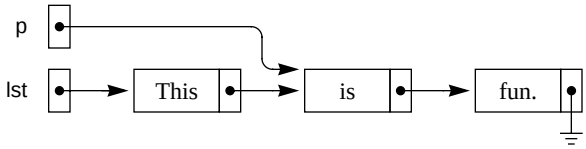
```
PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    VAR
        p: List;                                    n          val        i
        i: INTEGER;                                 1                     1
BEGIN
    ASSERT(0 <= n, 20);
    p := lst;
    FOR i := 1 TO n DO
        ASSERT(p.head # NIL, 21);
        p := p.head.next
    END;
    ASSERT(p.head # NIL, 21);
    val := p.head.value
END GetElementN;
```



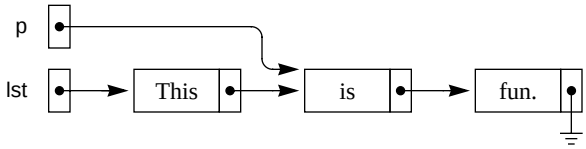
```
PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    VAR
        p: List;                                    n          val        i
        i: INTEGER;                                 1                     1
BEGIN
    ASSERT(0 <= n, 20);
    p := lst;
    FOR i := 1 TO n DO
        ASSERT(p.head # NIL, 21);
        p := p.head.next
    END;
    ASSERT(p.head # NIL, 21);
    val := p.head.value
END GetElementN;
```

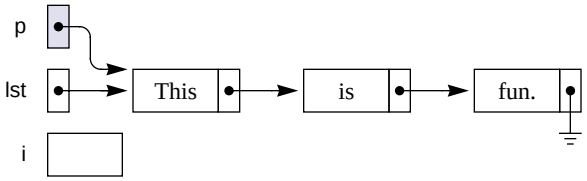


```
PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    VAR
        p: List;                                    n          val        i
        i: INTEGER;                                 1                     1
BEGIN
    ASSERT(0 <= n, 20);
    p := lst;
    FOR i := 1 TO n DO
        ASSERT(p.head # NIL, 21);
        p := p.head.next
    END;
    ASSERT(p.head # NIL, 21);
    val := p.head.value
END GetElementN;
```

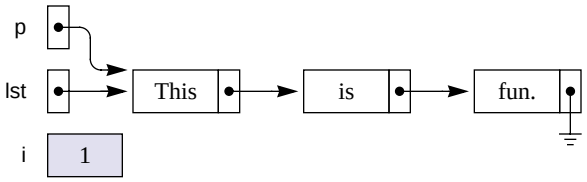


```
PROCEDURE (IN lst: List) GetElementN* (n: INTEGER; OUT val: T), NEW;
    VAR
        p: List;                                    n       val     i
        i: INTEGER;                                 1       is      1
BEGIN
    ASSERT(0 <= n, 20);
    p := lst;
    FOR i := 1 TO n DO
        ASSERT(p.head # NIL, 21);
        p := p.head.next
    END;
    ASSERT(p.head # NIL, 21);
    val := p.head.value
END GetElementN;
```

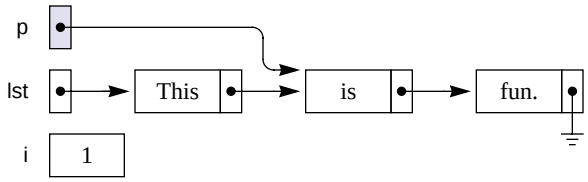




(a) p := lst



(b) FOR i := 1 TO n DO



(c) p := p.head.next

Figure 21.32
A trace of procedure GetElementN.

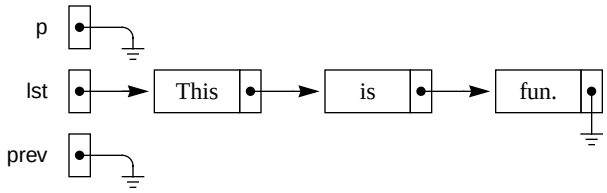
In the assignment statement

```
p := p.head.next
```

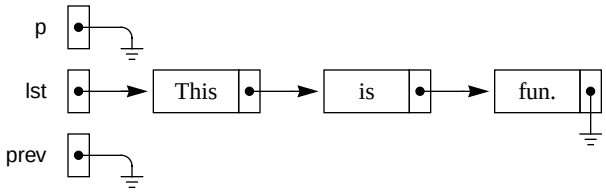
the relevant types are

- p is a record (which is also a List)
- p.head is a pointer to a Node
- p.head.next is a record (which is also a List)

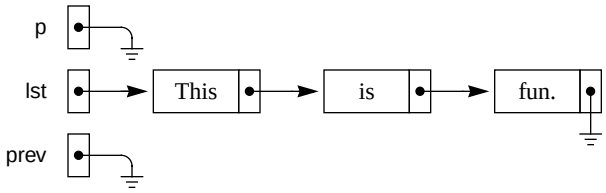
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



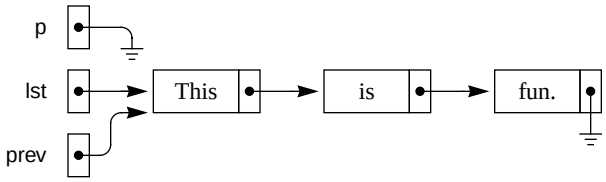
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
   VAR
      prev, p: List;                          n        val      i
      i: INTEGER;                             2        such
BEGIN
   ASSERT(0 <= n, 20);
   IF (n = 0) OR (lst.head = NIL) THEN
      p := lst;
      NEW(lst.head);
      lst.head.value := val;
      lst.head.next := p
   ELSE
      i := 1;
      prev := lst;
      p := lst.head.next;
      WHILE (i < n) & (p.head # NIL) DO
         INC(i);
         prev := p;
         p := p.head.next
      END;
      NEW(prev.head.next.head);
      prev.head.next.head.value := val;
      prev.head.next.head.next := p
   END
END InsertAtN;
```



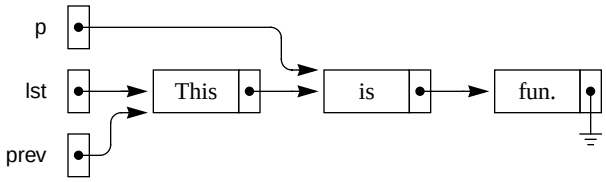
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    1
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



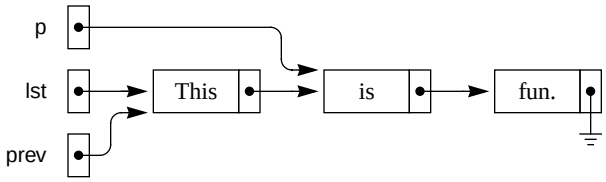
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    1
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



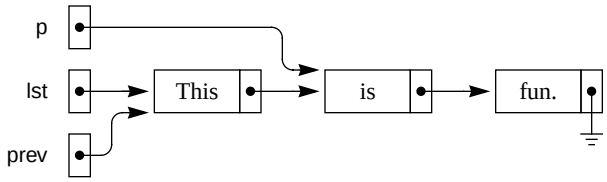
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    1
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



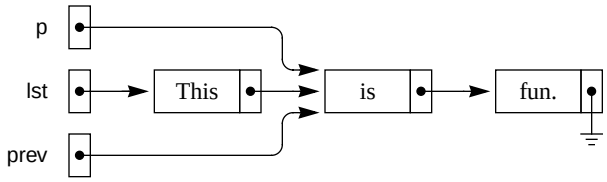
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    1
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



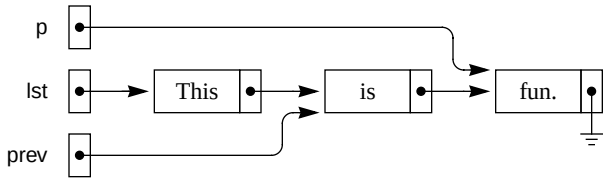
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    2
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



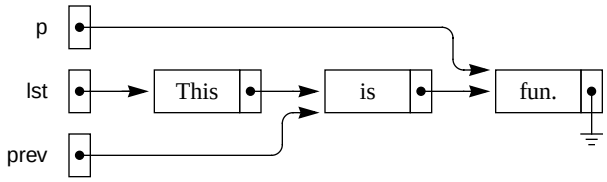
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    2
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



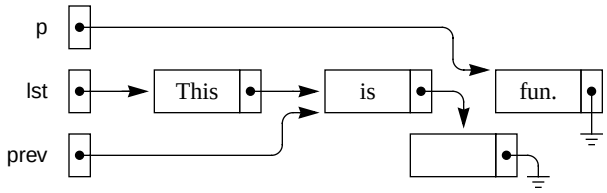
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    2
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



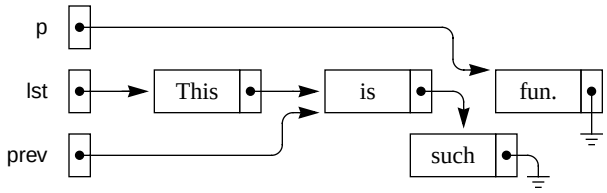
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    2
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



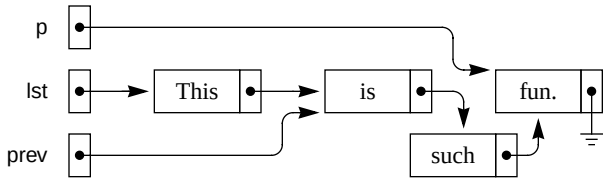
```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    2
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```

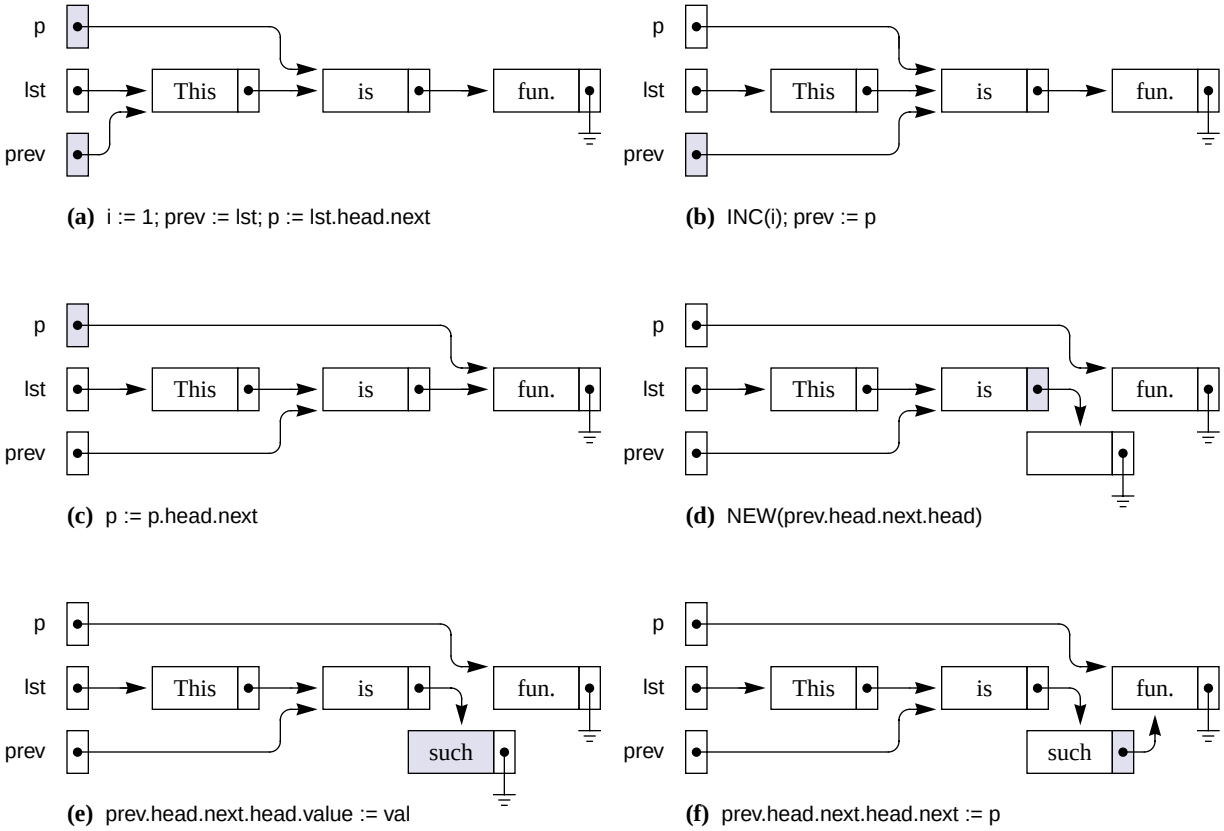


```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    2
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```



```
PROCEDURE (VAR lst: List) InsertAtN* (n: INTEGER; IN val: T), NEW;
    VAR
        prev, p: List;                          n       val     i
        i: INTEGER;                             2       such    2
BEGIN
    ASSERT(0 <= n, 20);
    IF (n = 0) OR (lst.head = NIL) THEN
        p := lst;
        NEW(lst.head);
        lst.head.value := val;
        lst.head.next := p
    ELSE
        i := 1;
        prev := lst;
        p := lst.head.next;
        WHILE (i < n) & (p.head # NIL) DO
            INC(i);
            prev := p;
            p := p.head.next
        END;
        NEW(prev.head.next.head);
        prev.head.next.head.value := val;
        prev.head.next.head.next := p
    END
END InsertAtN;
```





(a) i := 1; prev := lst; p := lst.head.next

(b) INC(i); prev := p

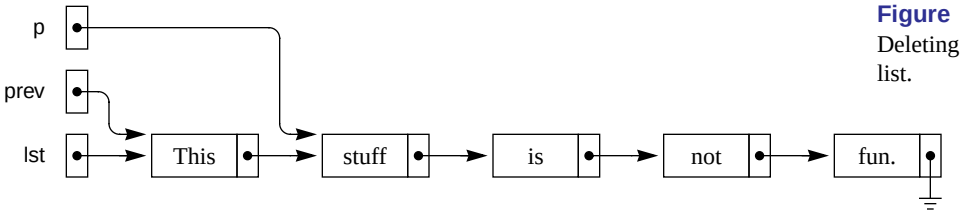
(c) p := p.head.next

(d) NEW(prev.head.next.head)

(e) prev.head.next.head.value := val

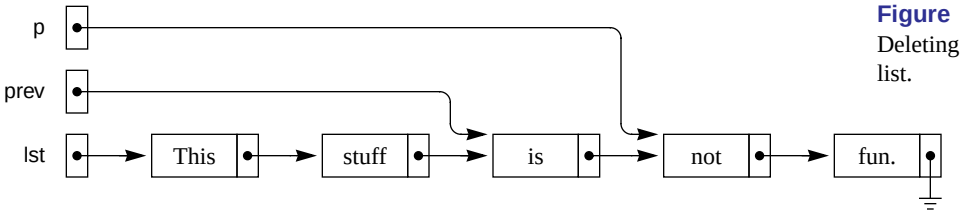
(f) prev.head.next.head.next := p

Figure 21.33
Execution of procedure InsertAtN to insert the word "such" at position 2.



(a) Initialize p and prev.

Figure 21.34
Deleting a node in a linked list.



(b) Find the node to delete.

Figure 21.34
Deleting a node in a linked list.

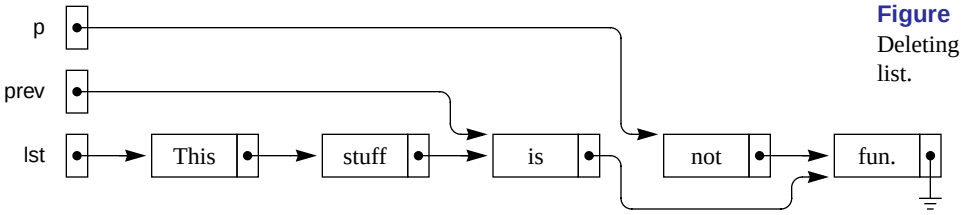
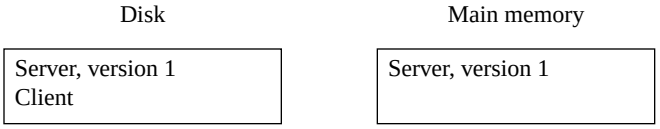


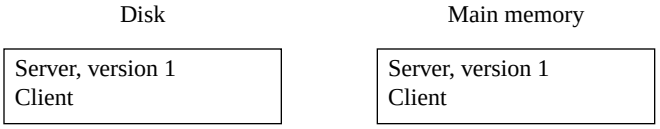
Figure 21.34
Deleting a node in a linked list.

(c) Unlink the node from the list.



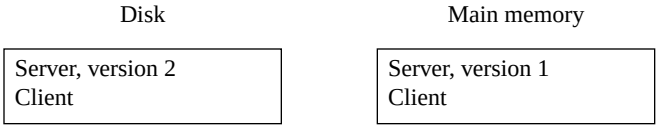
(a) Server module loads.

Figure 21.35
Developing a server module.



(b) Client module loads.

Figure 21.35
Developing a server module.



(c) Compile and Unload Server, version 2. Unloading fails.

Figure 21.35
Developing a server module.

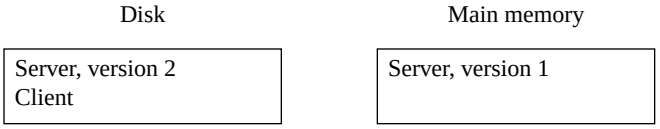


Figure 21.35
Developing a server module.

(d) Client must be unloaded first.

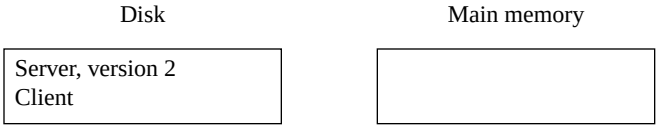


Figure 21.35
Developing a server module.

(e) Then Server, version 1 can be unloaded.