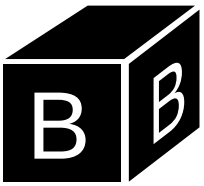


Chapter *17*

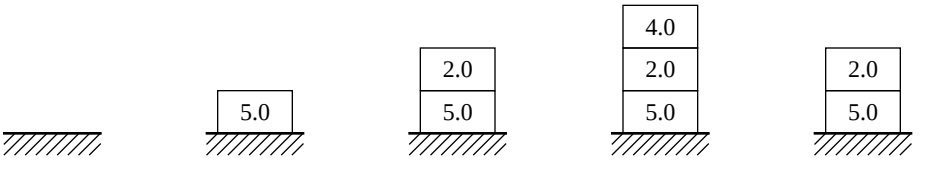
Stack and List Implementations

```
DEFINITION PboxStackADS;

    CONST
        capacity = 8;

    PROCEDURE Clear;
    PROCEDURE NumItems (): INTEGER;
    PROCEDURE Pop (OUT val: REAL);
    PROCEDURE Push (val: REAL);

END PboxStackADS.
```

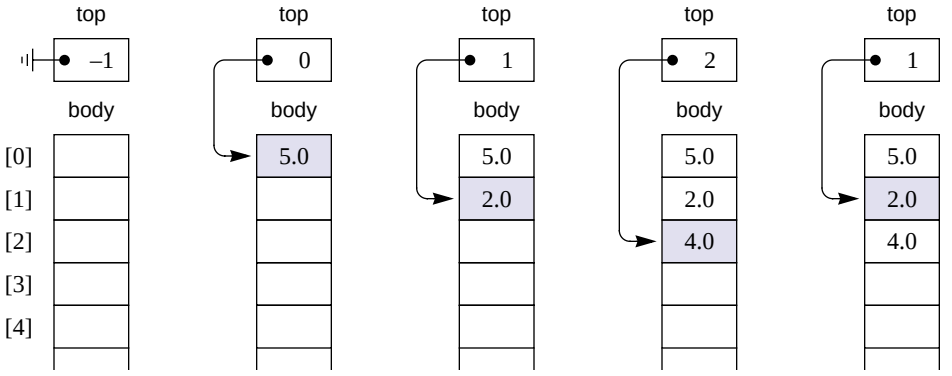
Figure 17.1
The interface of the stack abstract data structure.

Figure 17.2
A sequence of operations on a stack.

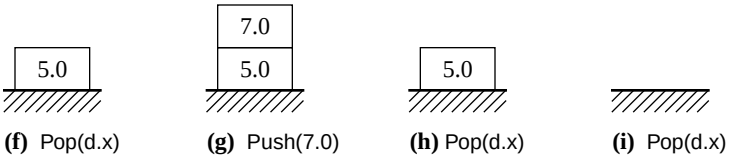
Figure 17.3
A Component Pascal implementation of a stack.



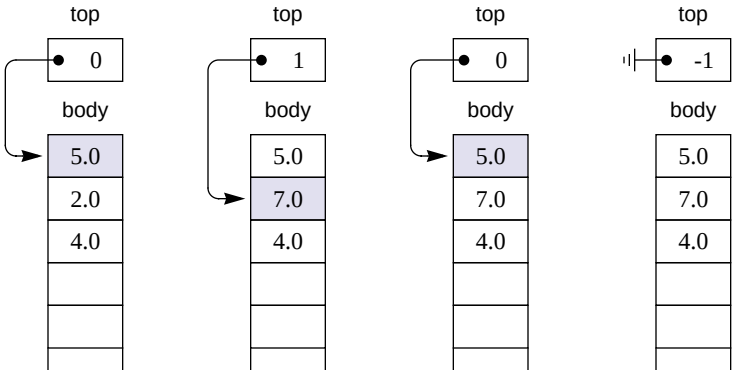
(a) BEGIN **(b)** Push(5.0) **(c)** Push(2.0) **(d)** Push(4.0) **(e)** Pop(d.x)



(a) BEGIN **(b)** Push(5.0) **(c)** Push(2.0) **(d)** Push(4.0) **(e)** Pop(d.x)



(f) Pop(d.x) **(g)** Push(7.0) **(h)** Pop(d.x) **(i)** Pop(d.x)



(f) Pop(d.x) **(g)** Push(7.0) **(h)** Pop(d.x) **(i)** Pop(d.x)

```
MODULE  PboxStackADS;

    CONST
        capacity* = 8;
    VAR
        body: ARRAY capacity OF REAL;
        top: INTEGER;

    PROCEDURE  Clear*;
    BEGIN
        top := -1
    END Clear;

    PROCEDURE  NumItems* (): INTEGER;
    BEGIN
        RETURN top + 1
    END NumItems;
```

Figure 17.4
The implementation of the stack abstract data structure.

```
    PROCEDURE  Pop* (OUT val: REAL);
    BEGIN
        ASSERT(0 <= top, 20);
        val := body[top];
        DEC(top)
    END Pop;

    PROCEDURE  Push* (val: REAL);
    BEGIN
        ASSERT(top < capacity - 1, 20);
        INC(top);
        body[top] := val
    END Push;

END PboxStackADS.
```

- IF in the client.
- ASSERT in the server.

The design-by-contract rule

```
DEFINITION PboxStackObj;

    CONST
        capacity = 8;

    TYPE
        Stack = RECORD
            (VAR s: Stack) Clear, NEW;
            (IN s: Stack) NumItems (): INTEGER, NEW;
            (VAR s: Stack) Pop (OUT val: REAL), NEW;
            (VAR s: Stack) Push (val: REAL), NEW
        END;

END PboxStackObj.
```

Figure 17.5
The interface of the stack class.

```
MODULE PboxStackObj;

    CONST
        capacity* = 8;
    TYPE
        Stack* = RECORD
            body: ARRAY capacity OF REAL;
            top: INTEGER
        END;

    PROCEDURE (VAR s: Stack) Clear*, NEW;
    BEGIN
        s.top := -1
    END Clear;

    PROCEDURE (IN s: Stack) NumItems* (): INTEGER, NEW;
    BEGIN
        RETURN s.top + 1
    END NumItems;
```

Figure 17.6
The implementation of the stack class.

```
    PROCEDURE (VAR s: Stack) Push* (val: REAL), NEW;
    BEGIN
        ASSERT(s.top < capacity - 1, 20);
        INC(s.top);
        s.body[s.top] := val
    END Push;

    PROCEDURE (VAR s: Stack) Pop* (OUT val: REAL), NEW;
    BEGIN
        ASSERT(0 <= s.top, 20);
        val := s.body[s.top];
        DEC(s.top)
    END Pop;

END PboxStackObj.
```

```
DEFINITION PboxListADT;

    CONST
        capacity = 8;

    TYPE
        List = RECORD END;
        T = ARRAY 16 OF CHAR;

    PROCEDURE Clear (VAR lst: List);
    PROCEDURE Display (IN lst: List);
    PROCEDURE GetElementN (IN lst: List; n: INTEGER; OUT val: T);
    PROCEDURE InsertAtN (VAR lst: List; n: INTEGER; IN val: T);
    PROCEDURE Length (IN lst: List): INTEGER;
    PROCEDURE RemoveN (VAR lst: List; n: INTEGER);
    PROCEDURE Search (VAR lst: List; IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN);

END PboxListADT.
```

Figure 17.7
The interface of the list abstract data type.

```
MODULE  PboxListADT;
   IMPORT StdLog;

   CONST
      capacity* = 8;
   TYPE
      T* = ARRAY 16 OF CHAR;
      List* = RECORD
         body: ARRAY capacity + 1 OF T; (* + 1 necessary for procedure Search *)
         lastIndex: INTEGER
      END;

   PROCEDURE  Clear* (VAR lst: List);
   BEGIN
      lst.lastIndex := -1
   END Clear;
```

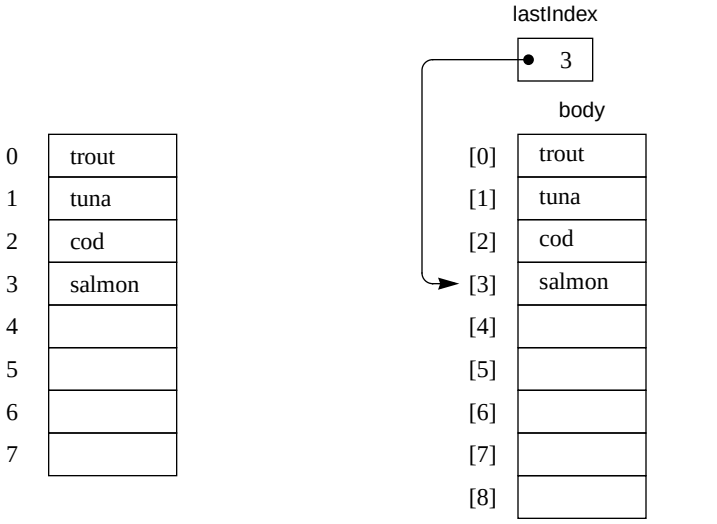
Figure 17.8
The implementation of the list abstract data type.

```
PROCEDURE  Display* (IN lst: List);
    VAR
        i: INTEGER;
BEGIN
    StdLog.Ln;
    FOR i := 0 TO lst.lastIndex DO
        StdLog.Int(i); StdLog.String(" "); StdLog.String(lst.body[i]); StdLog.Ln
    END
END Display;

PROCEDURE  GetElementN* (IN lst: List; n: INTEGER; OUT val: T);
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(n <= lst.lastIndex, 21);
    val := lst.body[n]
END GetElementN;
```

```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i: INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

```
PROCEDURE  Length* (IN lst: List): INTEGER;
BEGIN
   RETURN lst.lastIndex + 1
END Length;

PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
   VAR
      i: INTEGER;
BEGIN
   ASSERT(0 <= n, 20);
   IF n <= lst.lastIndex THEN
      FOR i := n TO lst.lastIndex - 1 DO
         lst.body[i] := lst.body[i + 1]
      END;
      DEC(lst.lastIndex)
   END
END RemoveN;
```

```
    PROCEDURE  Search* (VAR lst: List; IN srchVal: T; OUT n: INTEGER; OUT fnd: BOOLEAN);
    BEGIN
        lst.body[lst.lastIndex + 1] := srchVal;
        n := 0;
        WHILE lst.body[n] # srchVal DO
            INC(n)
        END;
        fnd := n <= lst.lastIndex
    END Search;

END PboxListADT.
```

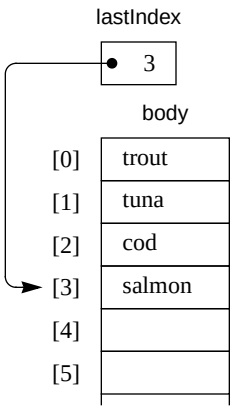


(a) The abstract list

(b) The array implementation

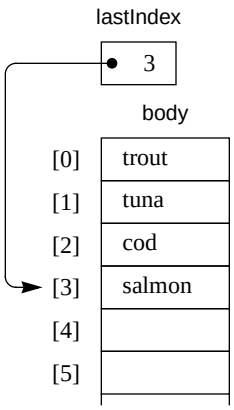
Figure 17.9
The abstract list ADT and its array implementation.

```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR                                                    n       val       i
        i:INTEGER;                                         2       bass
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```



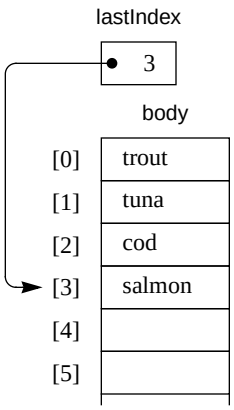
```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	



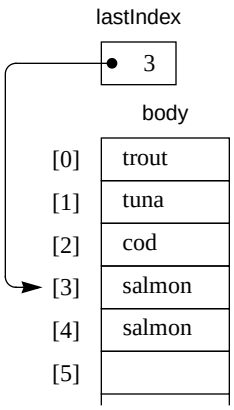
```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	3



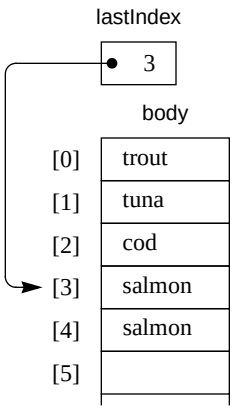
```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	3



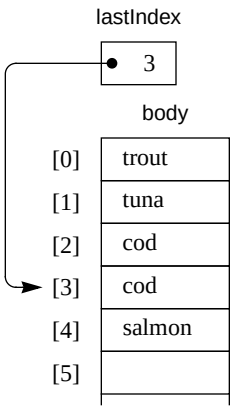
```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	2



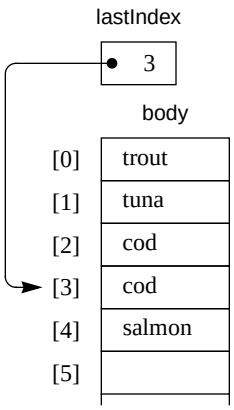
```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	2



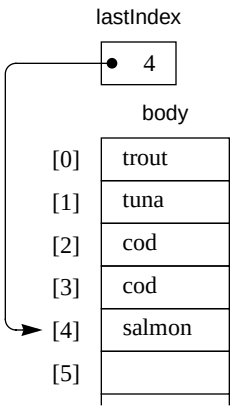
```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	1



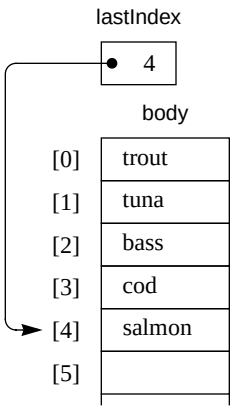
```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	1



```
PROCEDURE  InsertAtN* (VAR lst: List; n: INTEGER; IN val: T);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    ASSERT(lst.lastIndex < capacity - 1, 21);
    IF n > lst.lastIndex + 1 THEN
        n := lst.lastIndex + 1
    END;
    FOR i := lst.lastIndex TO n BY -1 DO
        lst.body[i + 1] := lst.body[i]
    END;
    INC(lst.lastIndex);
    lst.body[n] := val
END InsertAtN;
```

n	**val**	**i**
2	bass	1



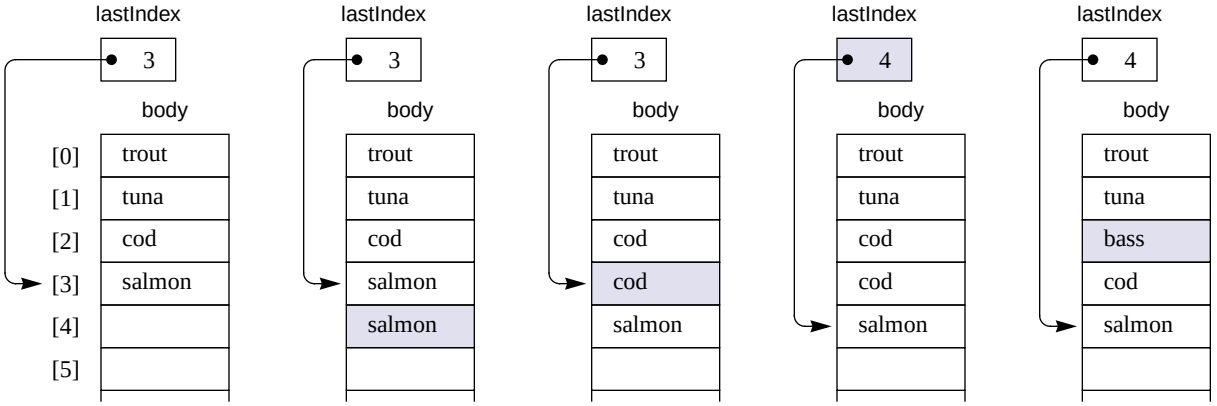
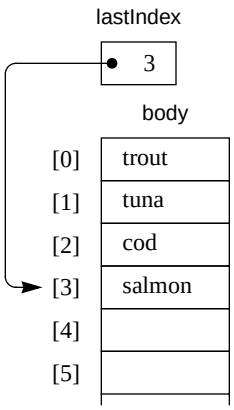


Figure 17.10
Execution of InsertAtN with 2 for n and bass for val.

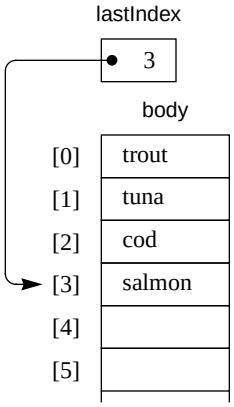
```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

n **i**
1



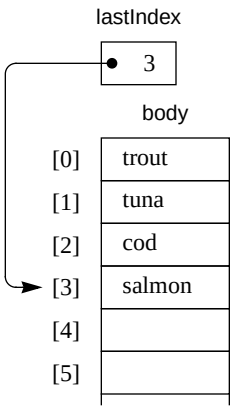
```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

n **i**
1



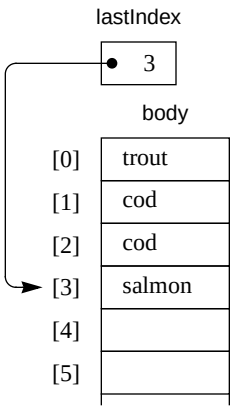
```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

n	**i**
1	1



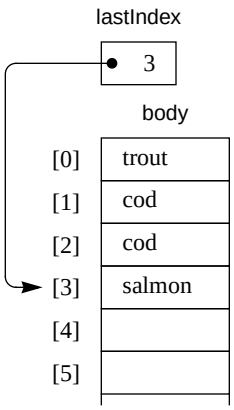
```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

n	**i**
1	1



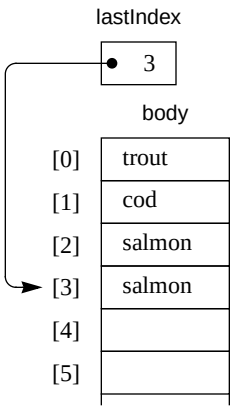
```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

n	**i**
1	2



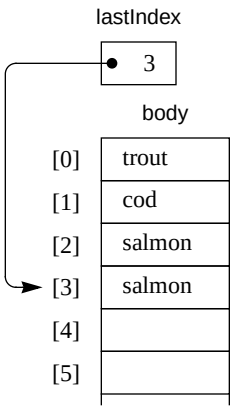
```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

n	**i**
1	2



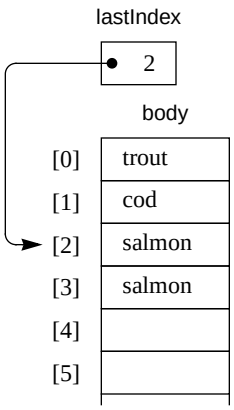
```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

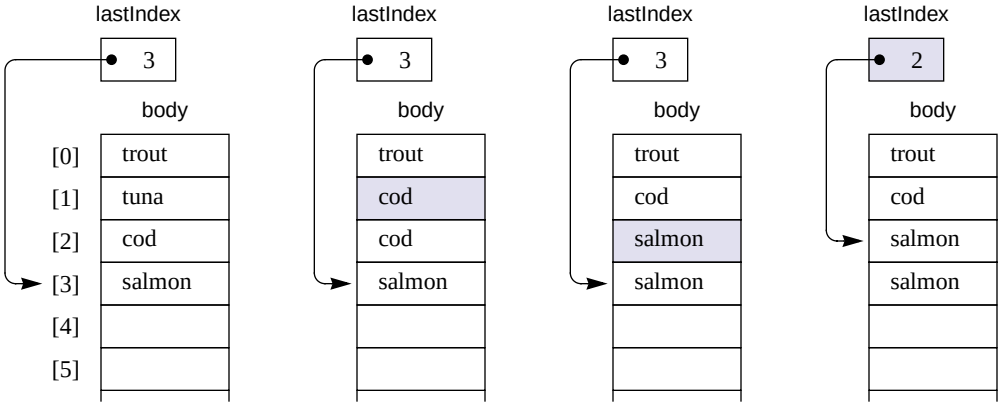
n	**i**
1	3



```
PROCEDURE  RemoveN* (VAR lst: List; n: INTEGER);
    VAR
        i:INTEGER;
BEGIN
    ASSERT(0 <= n, 20);
    IF n <= lst.lastIndex THEN
        FOR i := n TO lst.lastIndex - 1 DO
            lst.body[i] := lst.body[i + 1]
        END;
        DEC(lst.lastIndex)
    END
END RemoveN;
```

n	**i**
1	3





(a) Initial

(b) lst.body[1] := lst.body[2]

(c) lst.body[2] := lst.body[3]

(d) DEC (lst.lastIndex)

Figure 17.11
Execution of RemoveN with 1 for n.