Stack

Stack

- A stack is linear list in which all additions and deletions are restricted to one end, called top

- If you insert a data series into a stack and then remove it, the order of the data will be reverse. i.e. data input as {5,10,15,20} is removed as {20,15,10,5}

- For this reversing attribute stack is called **LIFO- Last in First out**

Stack

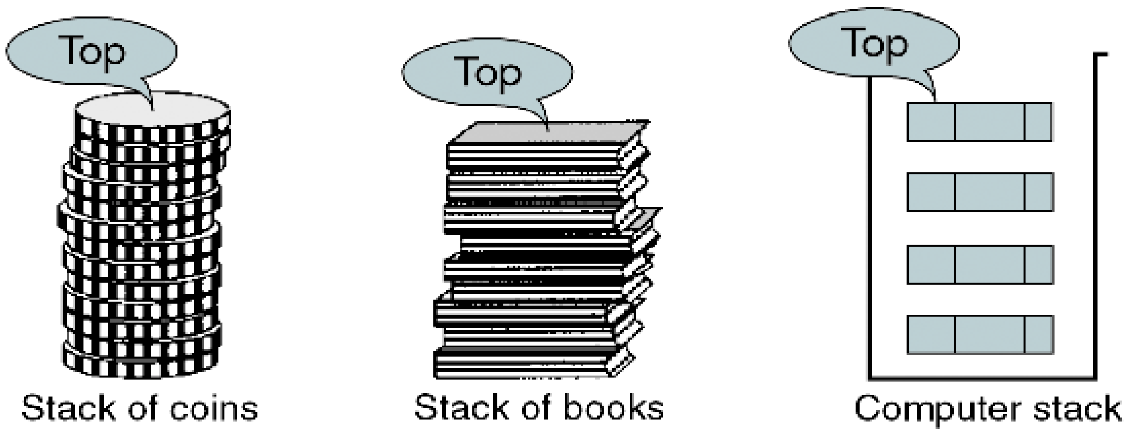

FIGURE 3-1 Stack

Basic Stack Operations

The stack concept is introduced and three basic stack operations are discussed.

- **Push**
- **Pop**
- **Stack Top**

Stack Example

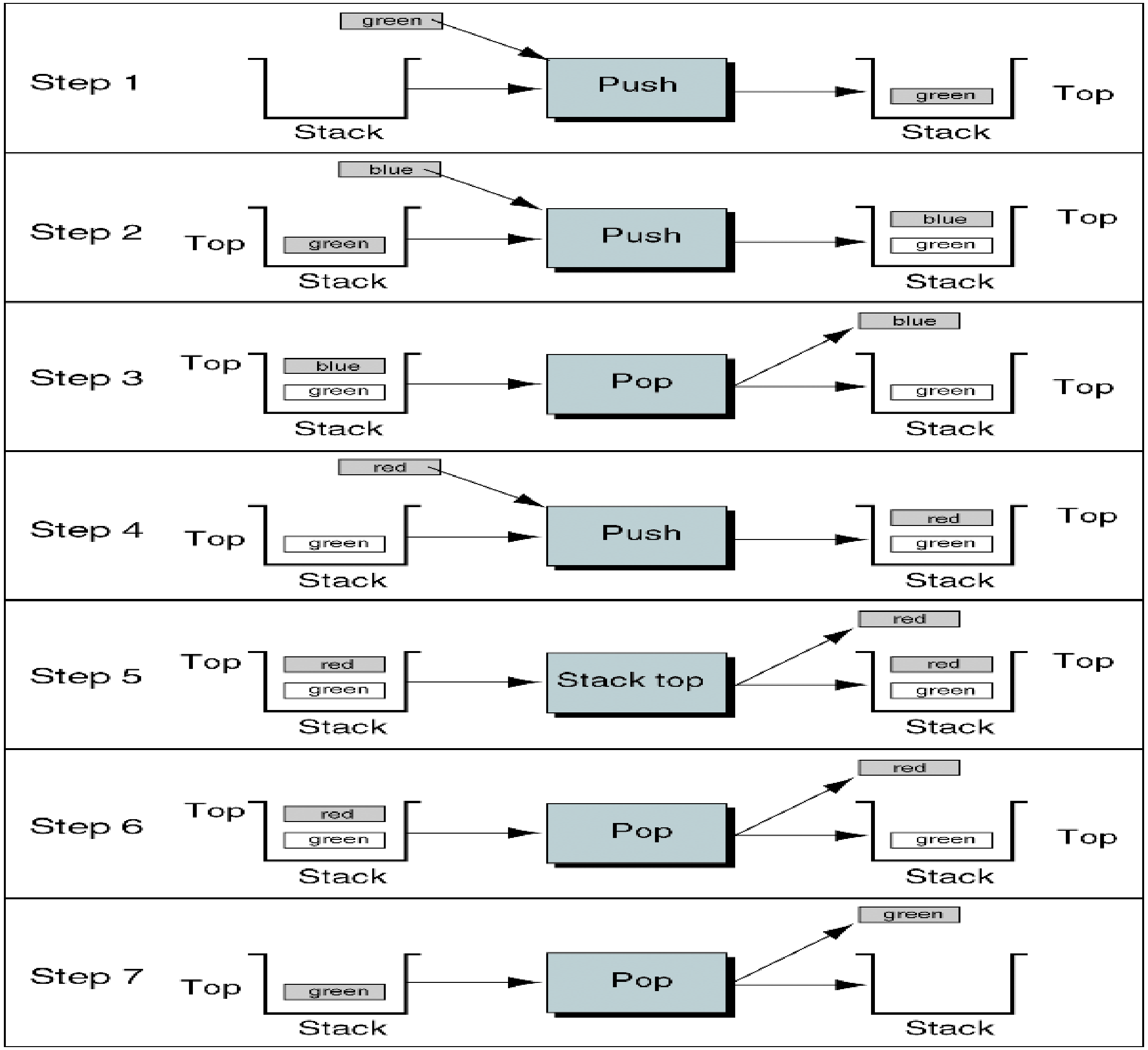

FIGURE 3-5 Stack Example

Stack Linked List Implementation

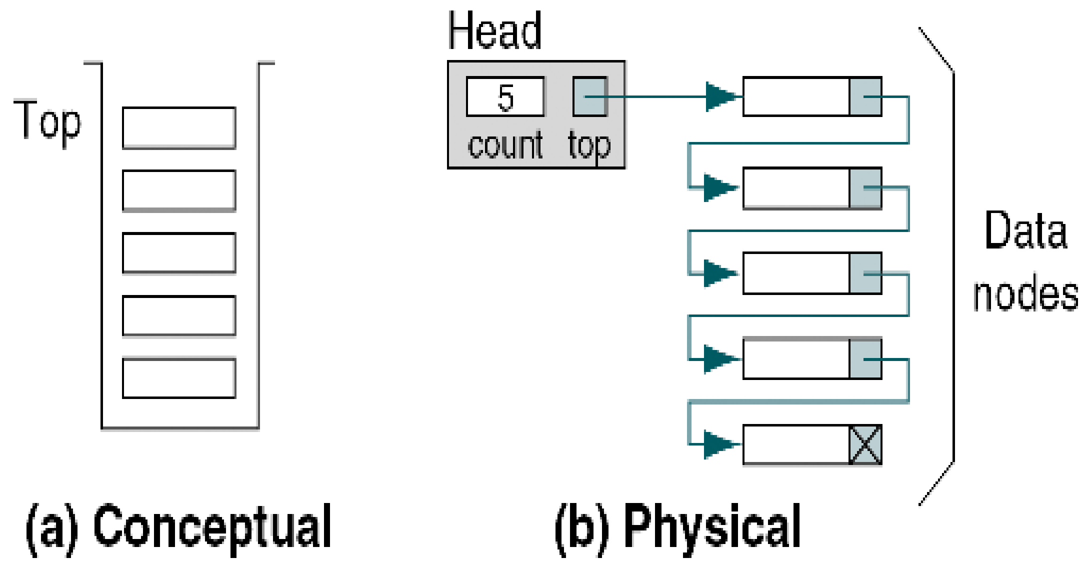

FIGURE 3-6 Conceptual and Physical Stack Implementations

Stack Linked List Implementation

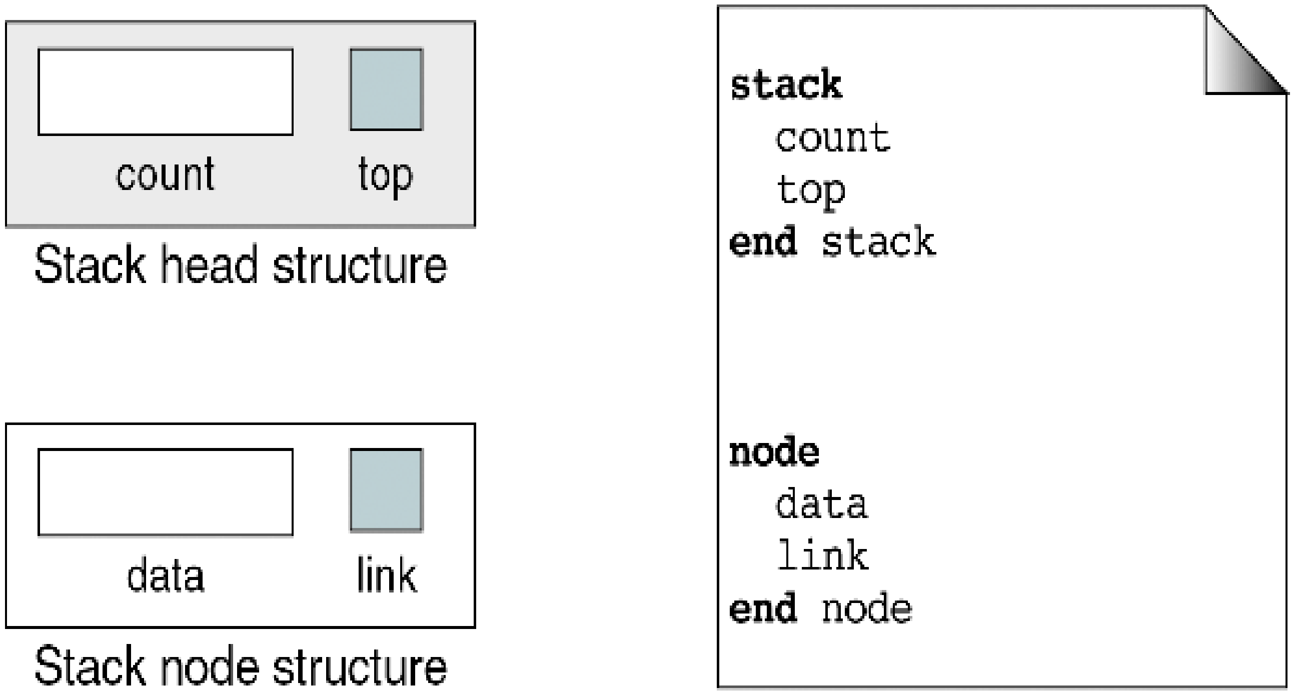

FIGURE 3-7 Stack Data Structure

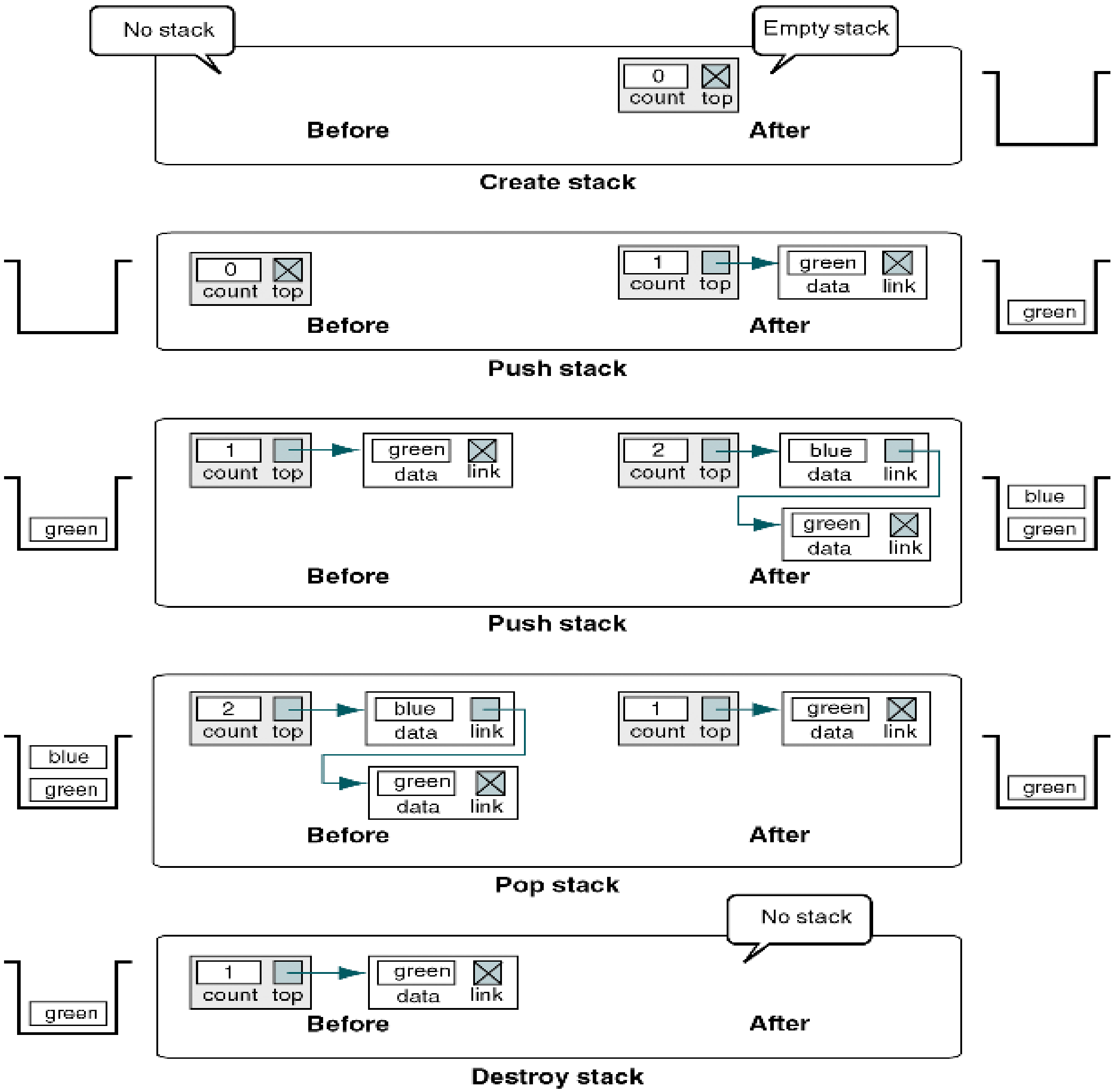

FIGURE 3-8 Stack Operations

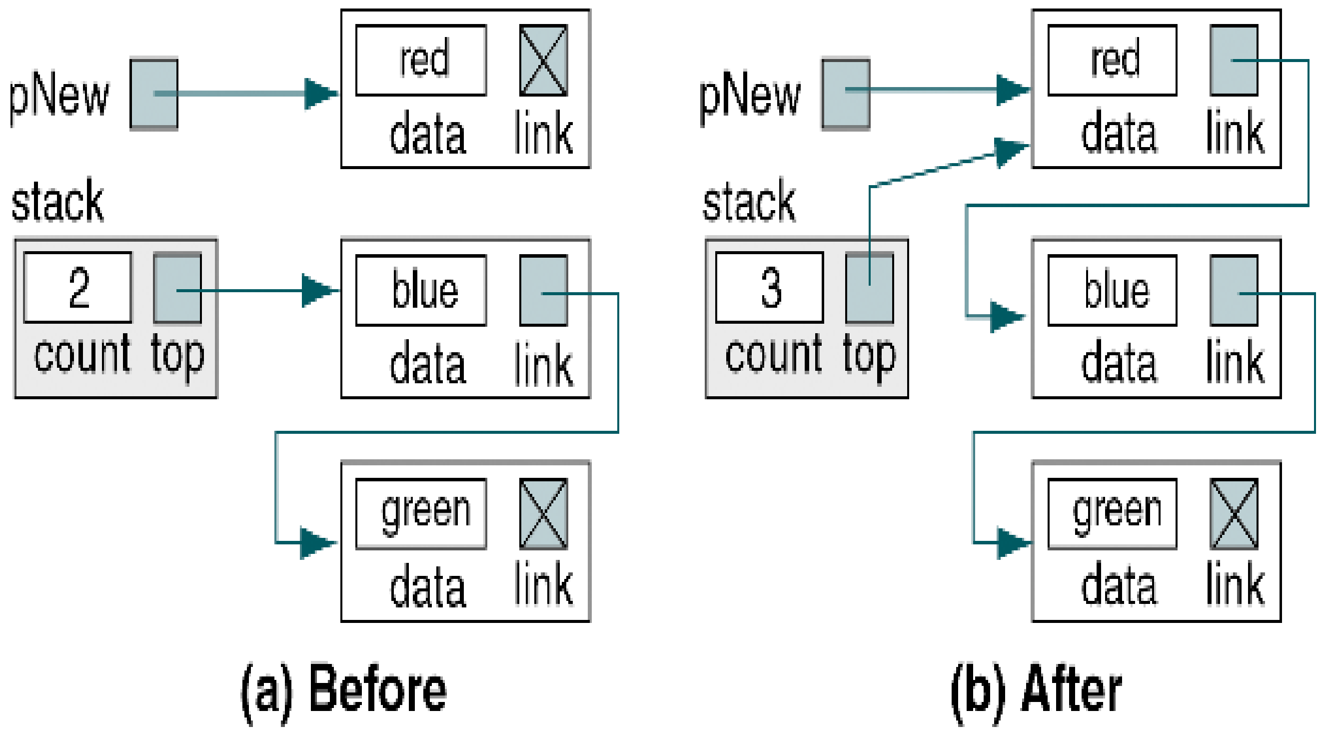

FIGURE 3-9 Push Stack Example

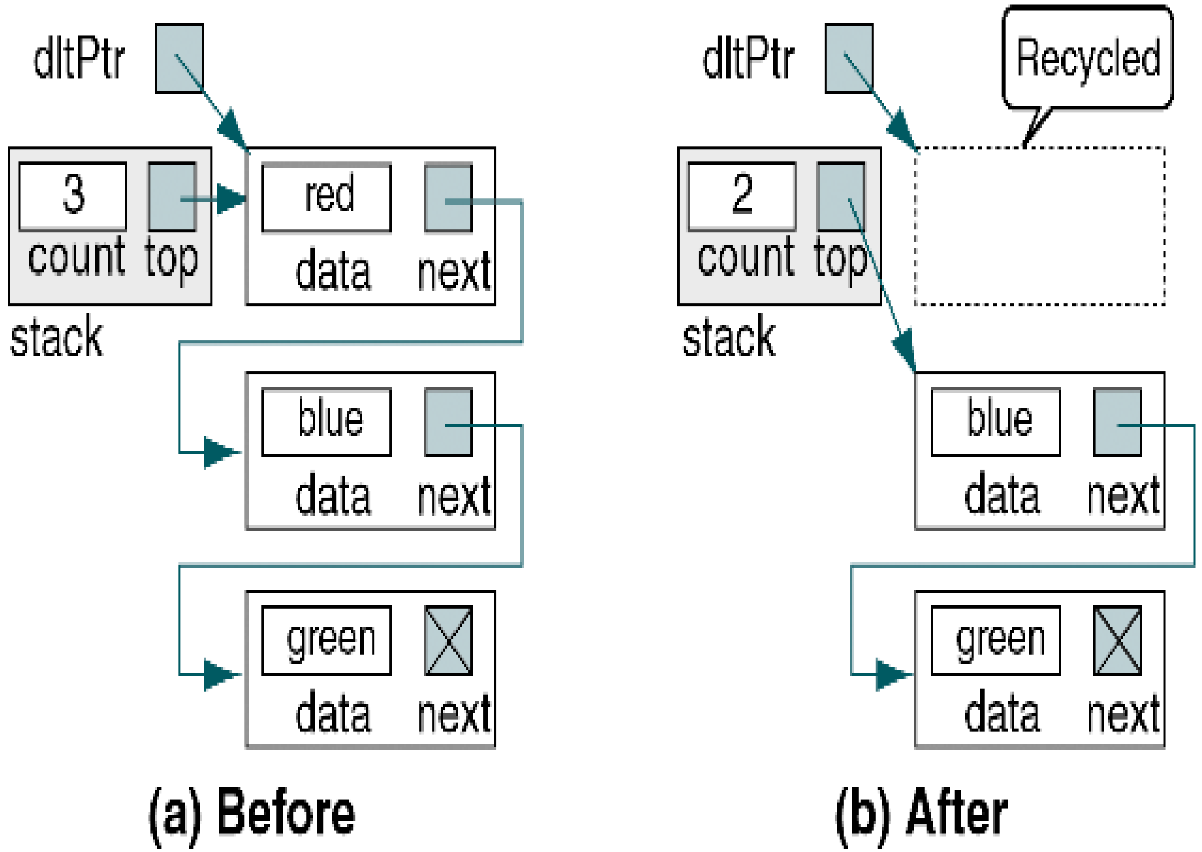

FIGURE 3-10 Pop Stack Example

3-4 Stack ADT

We begin the discussion of the stack ADT with a discussion of the stack structure and its application interface. We then develop the required functions.

- **Data Structure**
- **ADT Implemenation**

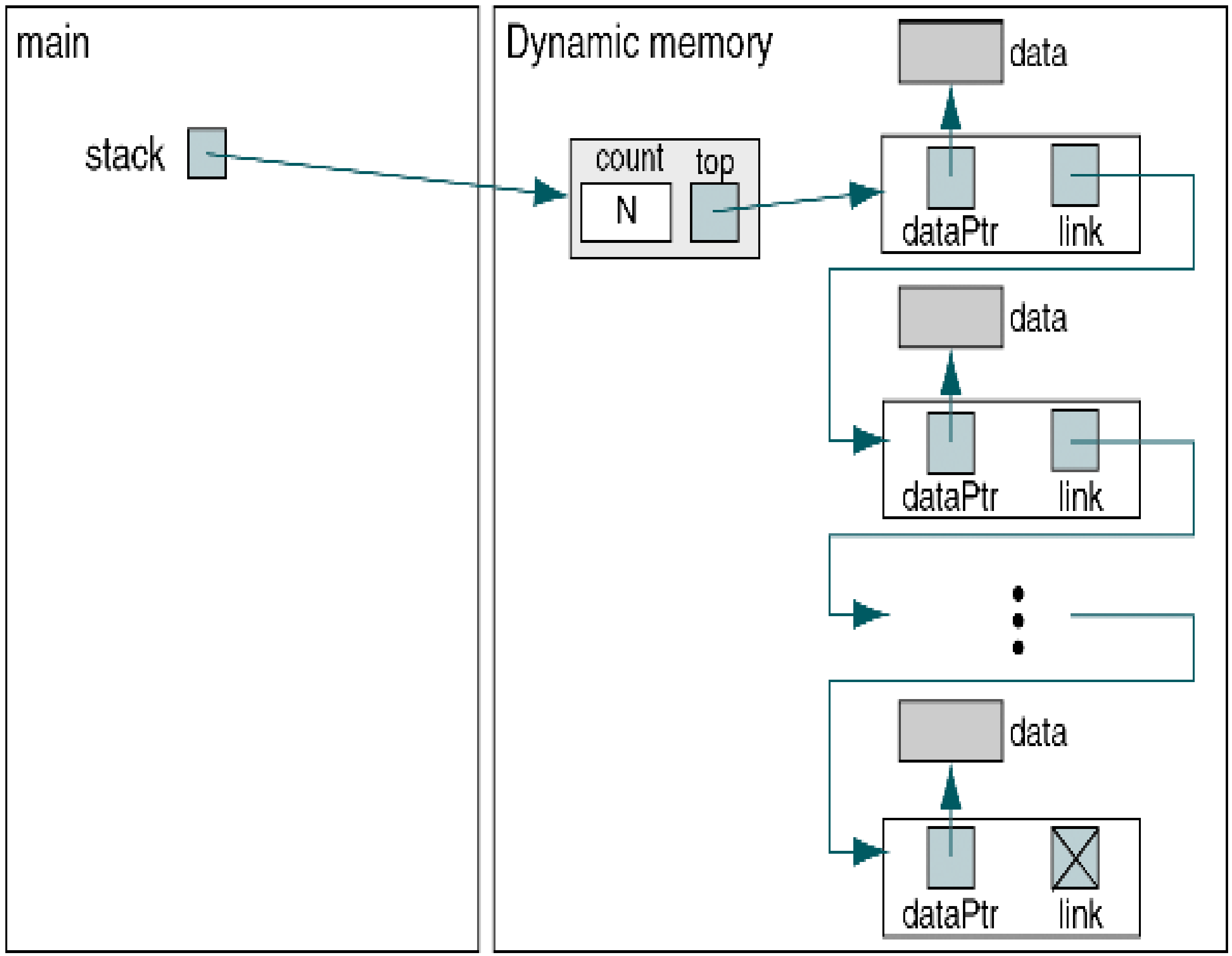

FIGURE 3-12 Stack ADT Structural Concepts

PROGRAM 3-6 Stack ADT Definitions

```
// Stack ADT Type Defintions
   typedef struct node
      {
       void*         dataPtr;
       struct node* link;
      } STACK_NODE;

   typedef struct
      {
       int          count;
       STACK_NODE* top;
      } STACK;
```

PROGRAM 3-7 ADT Create Stack

```
/* ================ createStack ===============
   This algorithm creates an empty stack.
      Pre  Nothing
      Post Returns pointer to a null stack
              -or- NULL if overflow
*/
STACK* createStack (void)
{
// Local Definitions
   STACK* stack;

// Statements
   stack = (STACK*) malloc( sizeof (STACK));
   if (stack)
      {
       stack->count = 0;
       stack->top   = NULL;
      } // if
   return stack;
}  // createStack
```

PROGRAM 3-8 Push Stack

```c
/* ================= pushStack ================
   This function pushes an item onto the stack.
      Pre     stack is a pointer to the stack
              dataPtr pointer to data to be inserted
      Post    Data inserted into stack
      Return  true  if successful
              false if underflow
*/
bool pushStack (STACK* stack, void* dataInPtr)
{
// Local Definitions
   STACK_NODE* newPtr;

// Statements
   newPtr = (STACK_NODE* ) malloc(sizeof( STACK_NODE));
   if (!newPtr)
```

PROGRAM 3-8 Push Stack *(continued)*

```c
       return false;

   newPtr->dataPtr = dataInPtr;

   newPtr->link    = stack->top;
   stack->top      = newPtr;

   (stack->count)++;
   return true;
} // pushStack
```

PROGRAM 3-9 ADT Pop Stack

```
/* ================== popStack ==================
   This function pops item on the top of the stack.
      Pre  stack is pointer to a stack
      Post Returns pointer to user data if successful
                   NULL if underflow
*/
void* popStack (STACK* stack)
{
// Local Definitions
   void*       dataOutPtr;
```

PROGRAM 3-9 ADT Pop Stack *(continued)*

```
   STACK_NODE* temp;

// Statements
   if (stack->count == 0)
       dataOutPtr = NULL;
   else
      {
       temp       = stack->top;
       dataOutPtr = stack->top->dataPtr;
       stack->top = stack->top->link;
       free (temp);
       (stack->count)--;
      } // else
   return dataOutPtr;
}  // popStack
```

PROGRAM 3-10 Retrieve Stack Top *(continued)*

```
void* stackTop (STACK* stack)
{
// Statements
   if (stack->count == 0)
       return NULL;
   else
       return stack->top->dataPtr;
}  // stackTop
```

PROGRAM 3-11 Empty Stack

```
/* ================= emptyStack ================
   This function determines if a stack is empty.
      Pre  stack is pointer to a stack
      Post returns 1 if empty; 0 if data in stack
*/
bool emptyStack (STACK* stack)
{
// Statements
   return (stack->count == 0);
}  // emptyStack
```

PROGRAM 3-12 Full Stack

```
/* ================= fullStack =================
   This function determines if a stack is full.
   Full is defined as heap full.
      Pre    stack is pointer to a stack head node
      Return true if heap full
             false if heap has room
*/
bool fullStack (STACK* stack)
{
// Local Definitions
STACK_NODE* temp;

// Statements
   if ((temp =
      (STACK_NODE*)malloc (sizeof(*(stack->top)))))
      {
       free (temp);
       return false;
      } // if

   // malloc failed
   return true;
}  // fullStack
```

PROGRAM 3-13 Stack Count

```
/* ================== stackCount ==================
   Returns number of elements in stack.
      Pre  stack is a pointer to the stack
      Post count returned
*/
int stackCount (STACK* stack)
{
// Statements
   return stack->count;
}  // stackCount
```

PROGRAM 3-14 Destroy Stack

```c
/* ================== destroyStack ==================
   This function releases all nodes to the heap.
      Pre  A stack
      Post returns null pointer
*/
STACK* destroyStack (STACK* stack)
{
// Local Definitions
   STACK_NODE* temp;

// Statements
   if (stack)
      {
       // Delete all nodes in stack
       while (stack->top != NULL)
          {
           // Delete data entry
           free (stack->top->dataPtr);

           temp = stack->top;
           stack->top = stack->top->link;
           free (temp);
          } // while

       // Stack now empty. Destroy stack head node.
       free (stack);
      } // if stack
   return NULL;
}  // destroyStack
```

Applications of Stack

- **Arithmetic Expression Evaluation:** Calculators use a stack structure to hold values for calculation.
- **Syntax Parsing:** Many compilers use a stack for parsing the syntax of expressions before translating into low level code.
- **Solving Search Problem**
- **Runtime Memory Management:** Almost all computer runtime memory environments use a special stack (the “call stack") to hold information about procedure/function calling and nesting in order to switch contexts.

Types of Arithmetic Expression

- **Infix Expression**

  - Operators are placed between its two operands.

  - Example: 2 + 4, a – b / c

- **Prefix Expression(Polish Notation)**

  - Operators are placed before its two operands.

  - Example: + 2 4, - a / b c

- **Postfix Expression (Reverse Polish Notation or RPN)**

  - Operators are placed after its two operands.

  - Example: 2 4 +, a b c / -

Algorithm: Infix-to-Postfix (Q, P)

Here Q is an arithmetic expression in infix notation and this algorithm generates

the postfix expression P using stack.

1. **Scan the infix expression Q from left to right.**
2. **Initialize an empty stack.**
3. **Repeat step 4 to 5 until all characters in Q are scanned.**
4. **If the scanned character is an operand, add it to P.**
5. **If the scanned character is an operator Φ, then**

    (a) If stack is empty, push Φ to the stack.

    (b) Otherwise repeatedly pop from stack and add to P each operator which has the same or higher precedence than Φ.

    (c) Push Φ to the stack.

6. If scanned character is a left parenthesis “(“, then push it to stack.

7. If scanned character is a right parenthesis “)”, then

(a) Repeatedly pop from stack and add to P each operator until “(“ is encountered.

(b) Remove “(“ from stack.

8. If all the characters are scanned and stack is not empty, then

(a) Repeatedly pop the stack and add to P each operator until the stack is empty.

9. Exit.

Example: Q: 5 * (6 + 2) - 12 / 4 and P: ?

Infix Expression Q	Stack	Postfix Expression P
5		5
*	*	5
(	* (	5
6	* (	5, 6
+	* (+	5, 6
2	* (+	5, 6, 2
)	*	5, 6, 2, +
-	-	5, 6, 2, +, *
12	-	5, 6, 2, +, *, 12
/	- /	5, 6, 2, +, *, 12
4	- /	5, 6, 2, +, *, 12, 4
	-	5, 6, 2, +, *, 12, 4, /
		5, 6, 2, +, *, 12, 4, /, -

Postfix Expression P : 5, 6, 2, +, *, 12, 4, /, -

Example: Q: A * ((B + C) - D) / E and P: ?

Infix Expression Q	Stack	Postfix Expression P
A		A
*	*	A
(	* (	A
(	* ((	A
B	* ((	A B
+	* ((+	A B
C	* ((+	A B C
)	* (	A B C +
-	* (-	A B C +
D	* (-	A B C + D
)	*	A B C + D -
/	/	A B C + D - *
E	/	A B C + D - * E
		A B C + D - * E /

Postfix Expression P : A B C + D - * E /

<u>Algorithm: Postfix-Evaluation (P, Value)</u>

Here P is an arithmetic expression in postfix notation and this algorithm finds the value of this expression using stack.

1. **Scan the postfix expression P from left to right.**
2. **Initialize an empty stack.**
3. **Repeat step 4 to 5 until all characters in P are scanned.**
4. **If the scanned character is an operand, push it to the satck.**
5. **If the scanned character is an operator Φ, then**
   - **(a) Remove two top elements of stack where A is the top element and B is the next-to-top element.**
   - **(b) Evaluate T = B Φ A and push T to the stack.**
6. **Pop the stack and assign the top element of the stack to Value.**
7. **Exit**

Example: P : 5, 6, 2, +, *, 12, 4, /, - and Value: ?

Postfix Expression Q	Stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

Value:37

Review Questions

- Write a function to implement a stack using a linked list/an array.
- You have an unsorted stack of numbers and must sort them using only one additional stack. What algorithm would you use to sort the stack? What is the time complexity of this approach?
- How would you use a stack to convert an infix expression (e.g., 3 + 5 * (2 - 8)) to postfix (3 5 2 8 - * +)? How would you evaluate the postfix expression using a stack?
- How can a stack be used to check if a sequence of brackets (e.g., {[()]}) is balanced or not?

Review Questions

6.4 Suppose stacks A[1] and A[2] are stored in a linear array STACK with N elements, as pictured in Fig. 6.37. Assume TOP[K] denotes the top of stack A[K].

(a) Write a procedure PUSH(STACK, N, TOP, ITEM, K) which pushes ITEM onto stack A[K].

(b) Write a procedure POP(STACK, TOP, ITEM, K) which deletes the top element from stack A[K] and assigns the element to the variable ITEM.

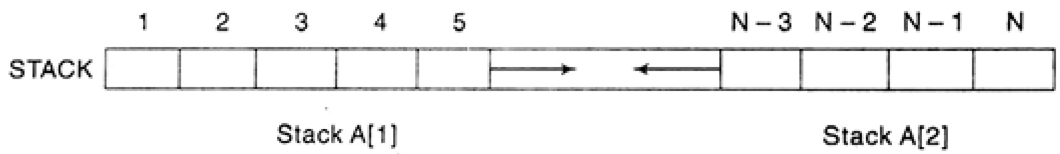

Fig. 6.37

6.5 Write a procedure to obtain the capacity of a linked stack represented by its top pointer TOP. The capacity of a linked stack is the number of elements in the list forming the stack.