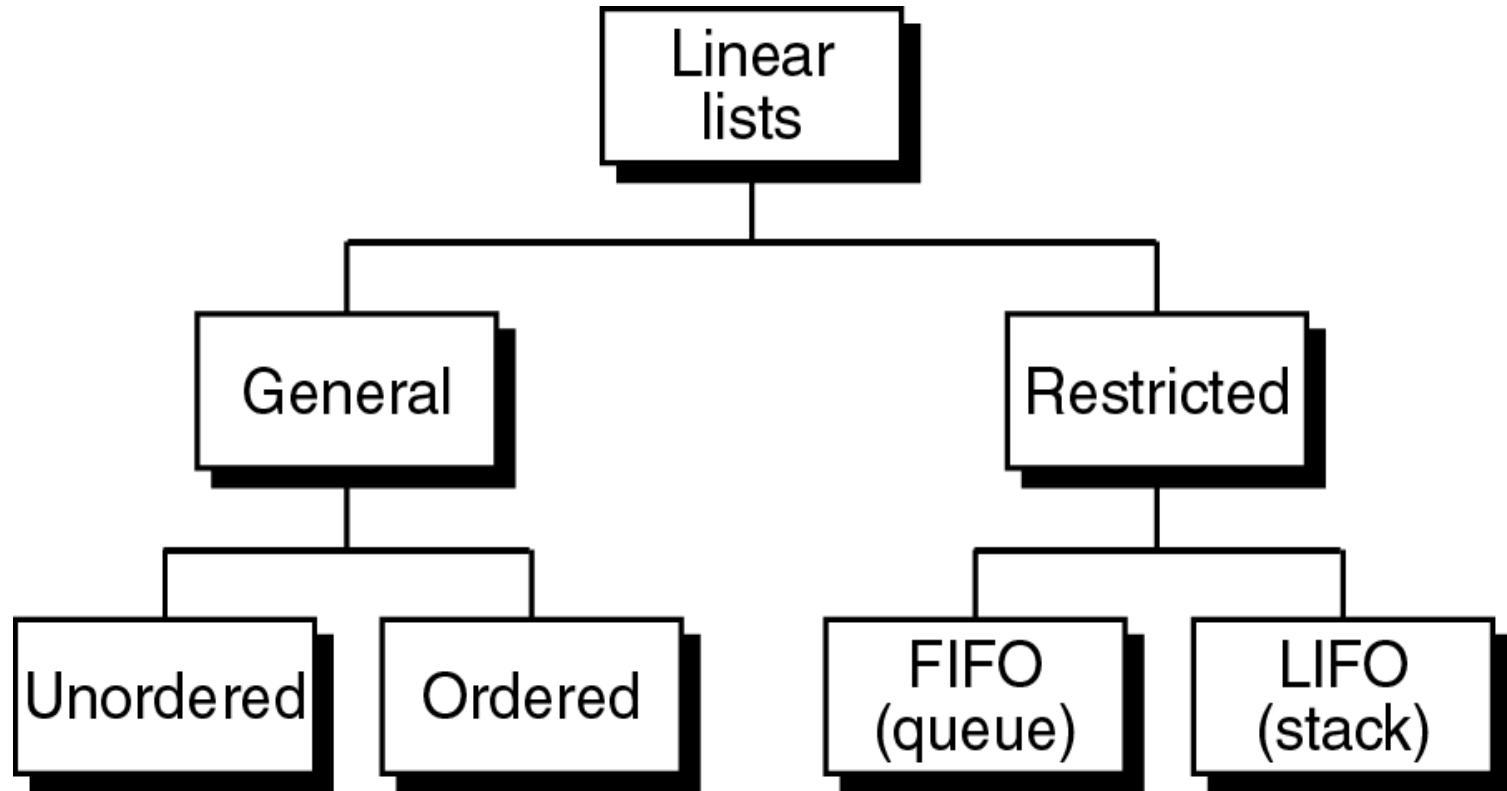


# **BLM212 Veri Yapıları**

**Stacks**  
**(Yığın)**

# Linear Lists



Operations are;

1. Insertion
2. Deletion
3. Retrieval
4. Traversal (exception for restricted lists).

# Linear Lists (Doğrusal Listeler)

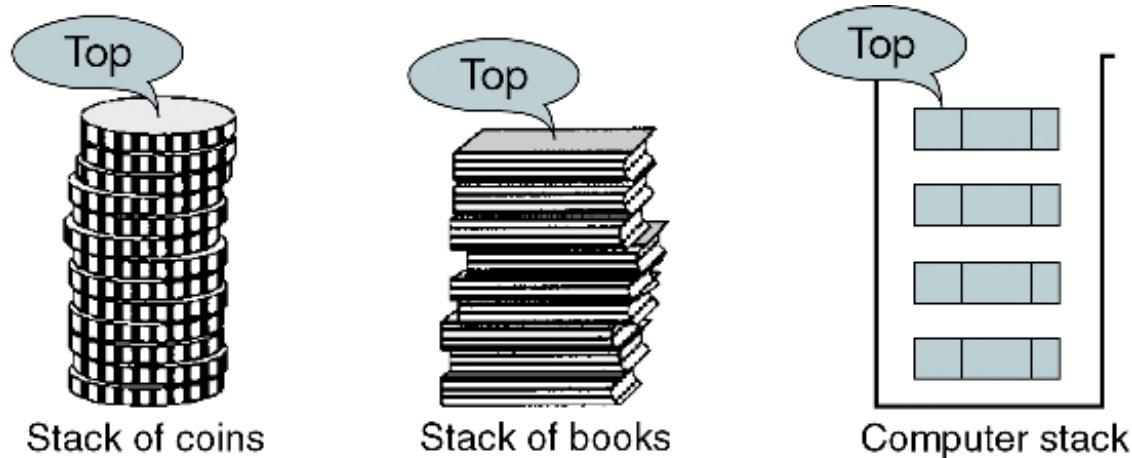
- Doğrusal Liste, her bir ögenin benzersiz bir halefi (successor) olduğu bir listedir.
- **Kısıtlanmış** bir doğrusal listede (**restricted linear list**), verilerin eklenmesi ve silinmesi listenin uçlarından olacak şekilde sınırlanmıştır.
- Genel bir doğrusal listede (**general linear list**), her noktadan verilerin eklenmesine ve silinmesine izin verilir.

# Stack (Yığın)

- Yığın (**Stack**), verilerin eklenmesi ve silinmesinin tepe (**top**) adı verilen listenin bir ucu ile kısıtlandığı doğrusal bir listedir.
- Bir dizi veriyi bir yığına yerleştirip çıkarırsak, veri sırası tersine çevrilir.
- Bu özellik **last in – first out (LIFO)** bilinir.

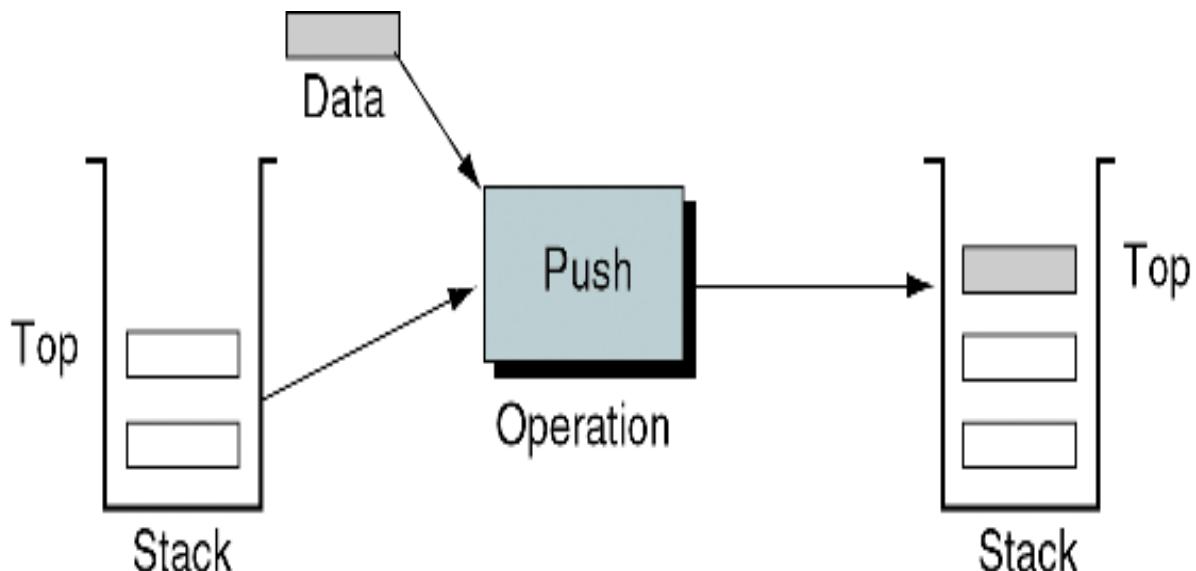
**Yalnızca en üste bir nesneyi ekleyebileceğiniz veya çıkarabileceğiniz herhangi bir durum bir yiğindir.**

**Üstteki nesneden başka herhangi bir nesneyi çıkarmak istiyorsanız, önce onun üzerindeki tüm nesneleri kaldırmanız gereklidir.**



**FIGURE 3-1 Stack**

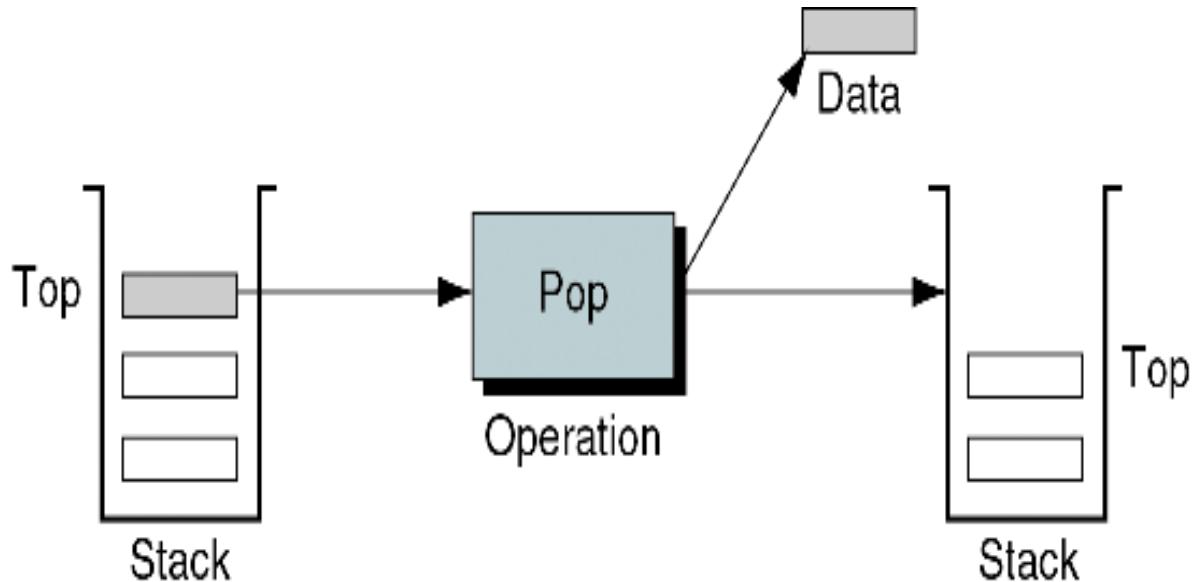
A stack is a last in-first out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.



---

FIGURE 3-2 Push Stack Operation (Yığına itme)

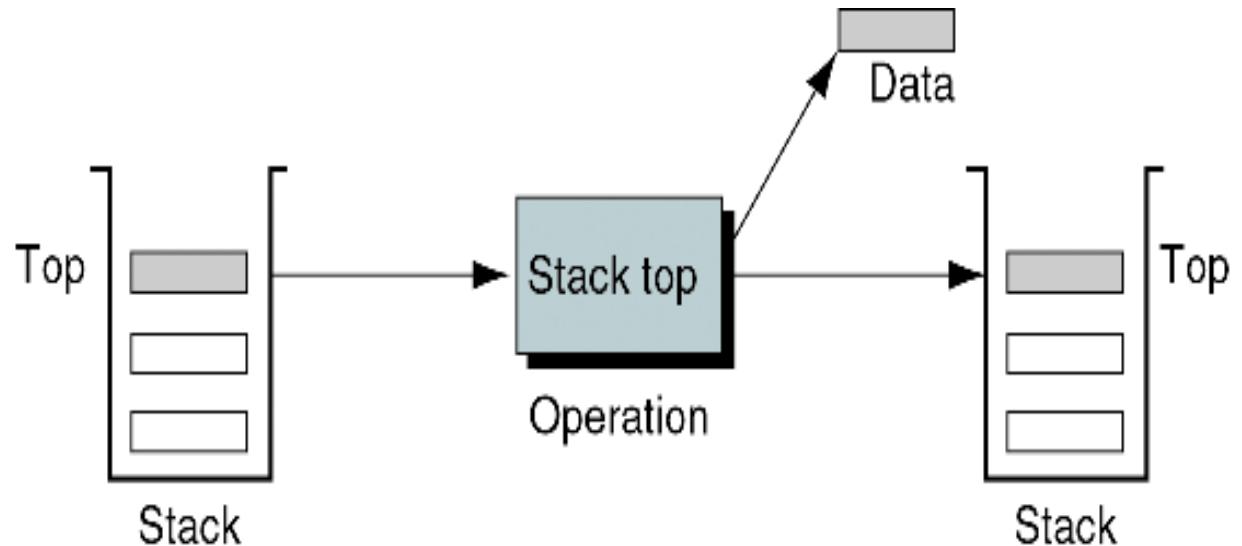
- Bu basit işlemle ilgili tek potansiyel sorun, yeni eleman için yer olduğundan emin olmamız gerektidir.
- Yeterli yer yoksa;
  - yığın taşıma (**Overflow**) durumundadır ve öge eklenemez.



---

FIGURE 3-3 Pop Stack Operation (Yığından çekme)

- Yığındaki son eleman yığından çekildiğinde yığın boş (empty) durumuna geçirilmelidir.
- Eğer yığın boş iken **Pop** işlemi yapılrsa
  - yığın **Underflow** durumundadır



---

FIGURE 3-4 Stack Top Operation

- Bu işlem yiğinin tepesindeki elemanı kopyalayıp onu kullanıcıya döndürür.
  - O elemanı yiğinden çıkarmaz/silmez.
- Yani yiğinin tepesini okuma işlemi olarak görülebilir.

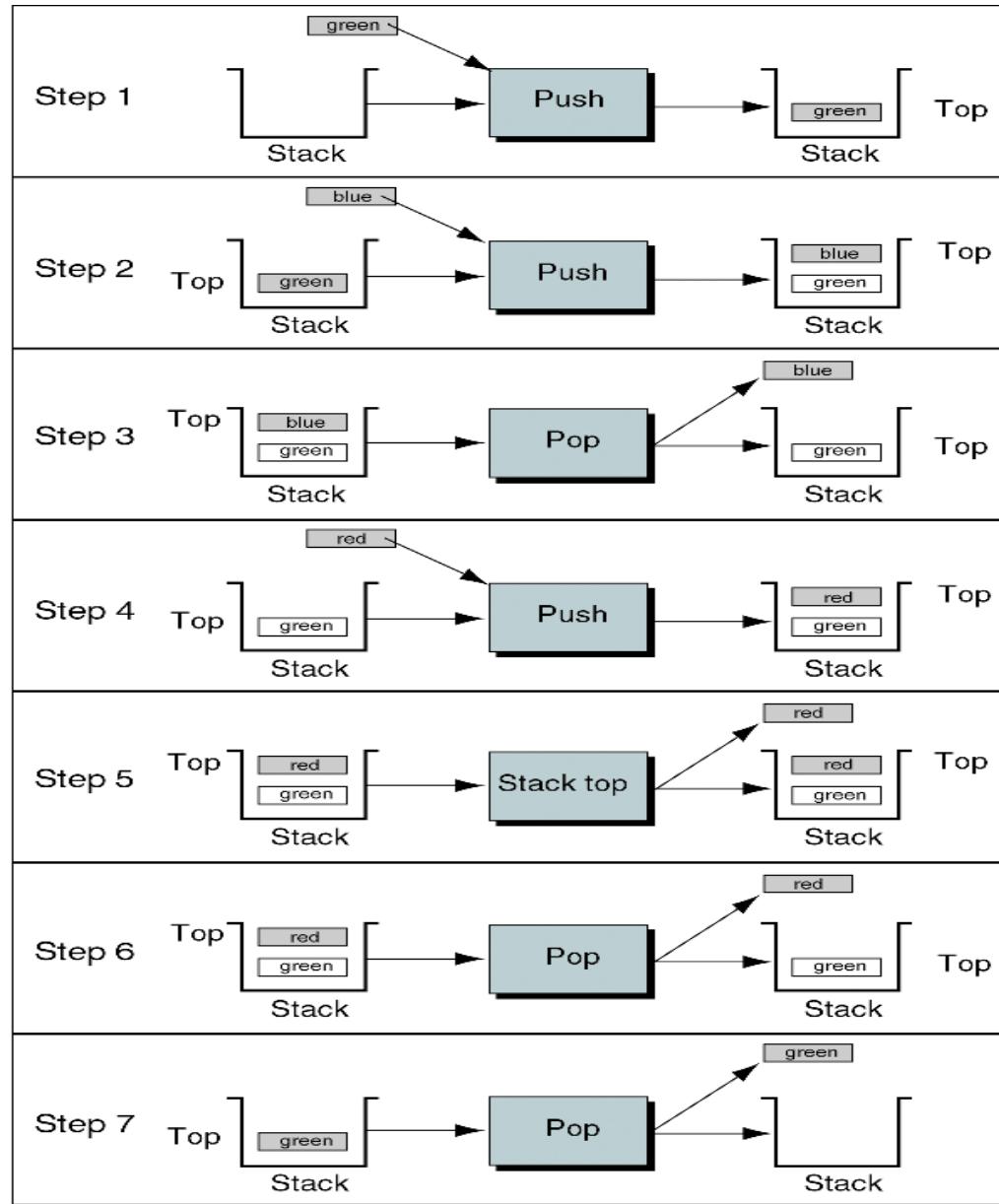


FIGURE 3-5 Stack Example

# Stack - Linked List Implementation

- Bir yiğini gerçekleştirmek (implement) için birkaç yol vardır.
- Bunlardan birisi
  - Yiğini bağlı liste olarak gerçekleştirmek

- Bağlı liste ile yiğin gerçekleştirmek için iki yapıya (structure) ihtiyaç vardır:
  - Head node
  - Data node

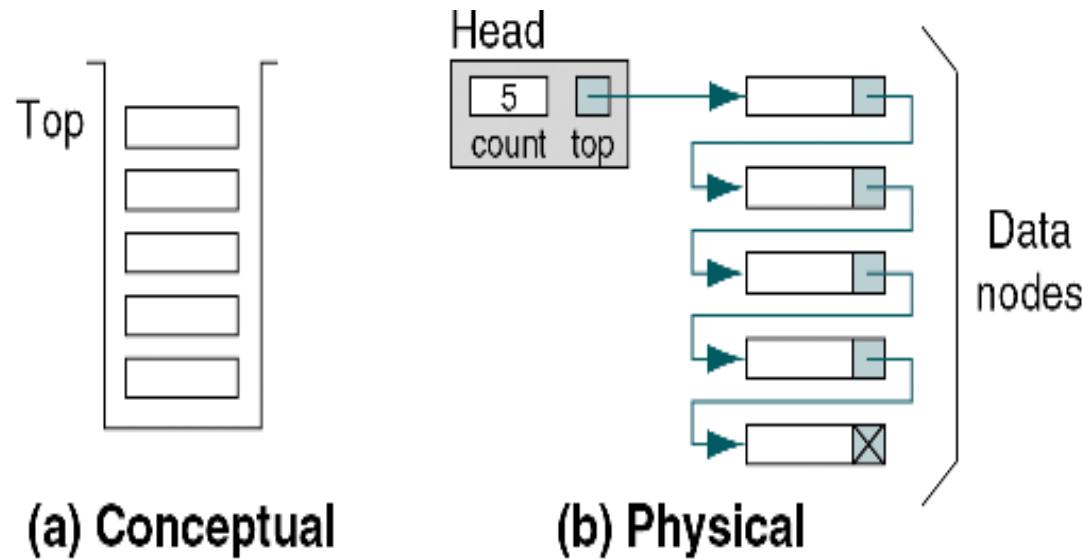
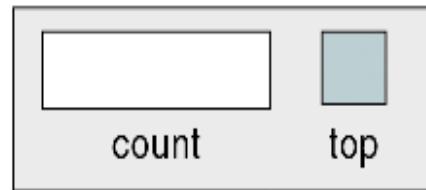
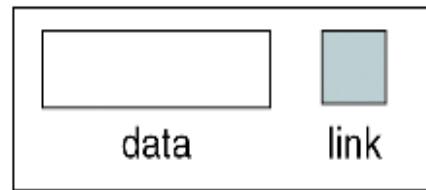


FIGURE 3-6 Conceptual and Physical Stack Implementations

- Head structure
  - **metadata** (yani veri hakkında veriyi) ve
  - yiğinin tepesine işaret eden bir **pointer** barındırır
- Data structure
  - Veriyi ve
  - Yiğında bir sonraki düğüme işaret eden bağlantı pointer'ı barındırır.



Stack head structure



Stack node structure

```

stack
    count
    top
end stack

node
    data
    link
end node

```

**FIGURE 3-7** Stack Data Structure

- **Stack head** genellikle 2 şey gerektirir:
  - Yığının tepesine işaret eden bir pointer
  - Yığındaki eleman sayısını tutan sayıcı
  - Bunun dışında yığın ne zaman oluşturuldu ve yığının gördüğü en fazla eleman sayısı gibi bilgiler de tutulabilir
- **Stack Data Node**
  - Veri yapısının geri kalanı tipik bir bağlantılı liste veri düğümüdür.

# Stack Algorithms

- Bu bölümde tanımlanan sekiz yığın işlemi, herhangi bir temel yığın problemini çözmek için yeterlidir.
- Eğer bir uygulama ilave yığın işlemleri gerektiriyorsa, bunlar da kolayca eklenebilir.

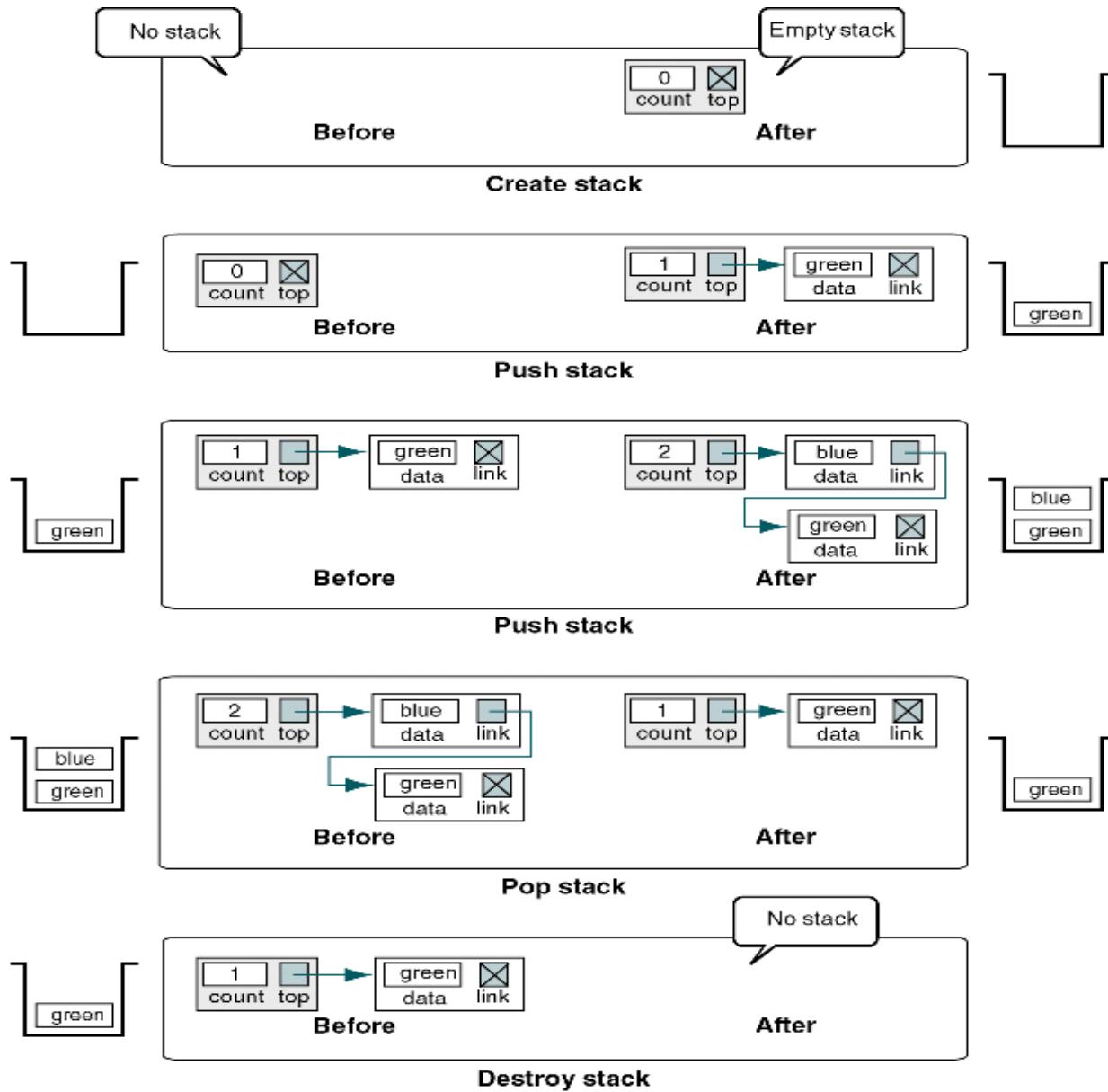


FIGURE 3-8 Stack Operations

# Create Stack

algorithm **createStack**

Allocates memory for a stack head node from dynamic memory and returns its address to the caller.

**Pre** Nothing

**Post** Head node allocated or error returned

**Return** pointer to head node or null pointer if no memory

1. if (memory available)

1. allocate (stackPtr)
2. stackPtr→count = 0
3. stackPtr→top = null

2. else

1. stackPtr = null

3. return stackPtr

end **createStack**

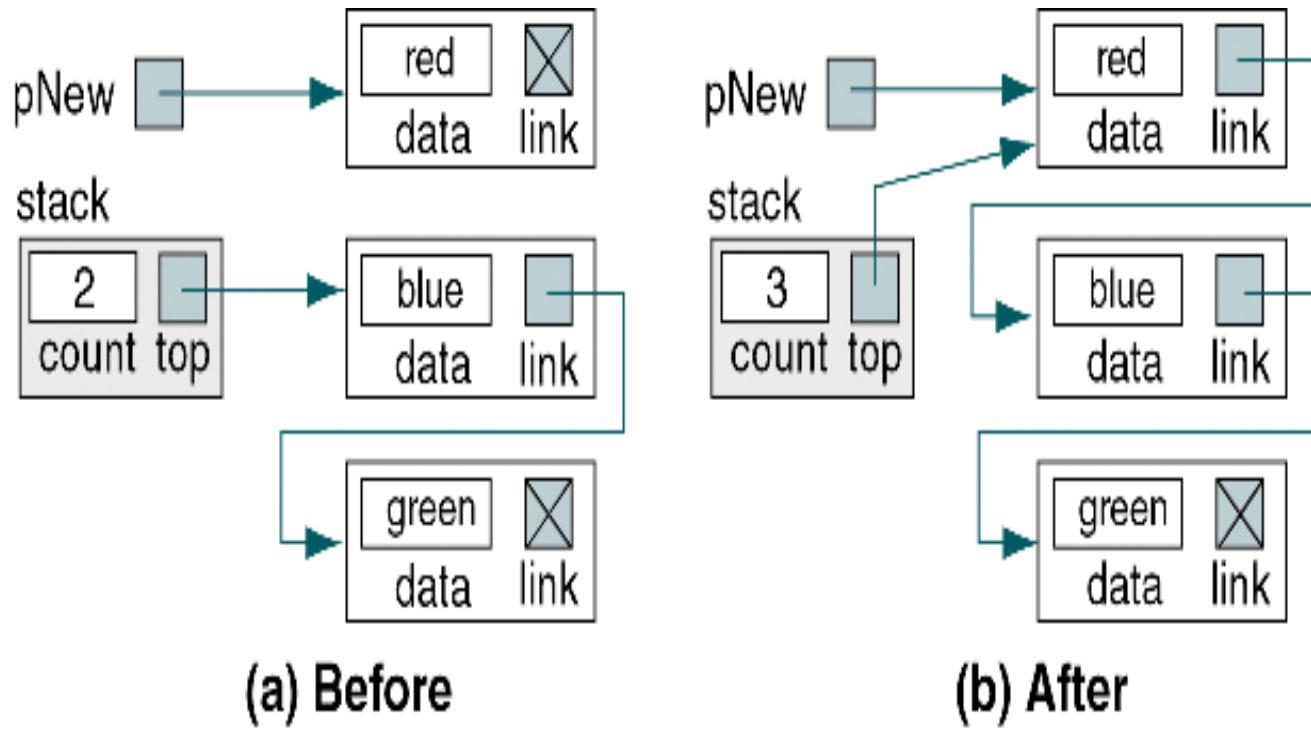


FIGURE 3-9 Push Stack Example

algorithm **pushStack**(val stack <head pointer>, val data <data type>)

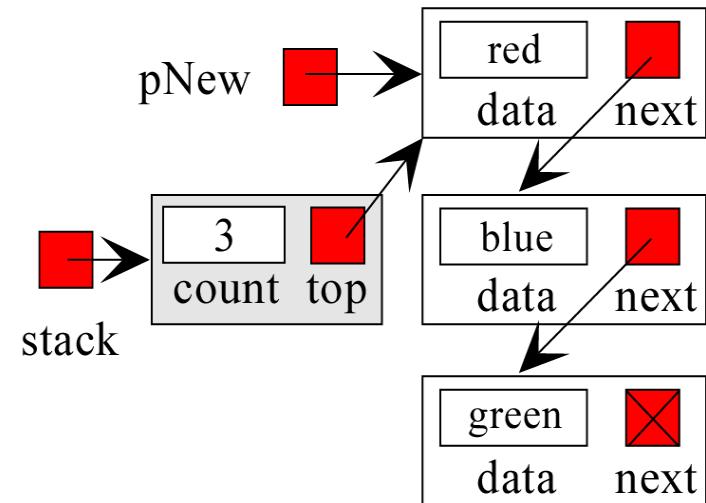
Insert (push) one item into the stack.

Pre stack is a pointer to the stack head structure (stack passed by reference). data contains data to be pushed into stack.

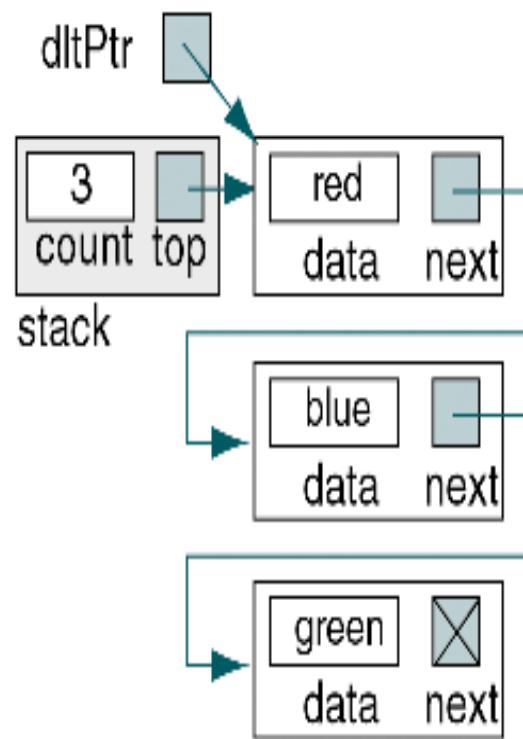
Post data have been pushed in stack.

1. if (stack full)
    1. success = false
  2. else
    1. allocate (newPtr)
    2. newPtr→data= data
    3. newPtr→next= stack→top
    4. stack→top=newPtr
    5. stack→count = stack→count+1
    6. success=true
  3. return success
- end **pushStack**

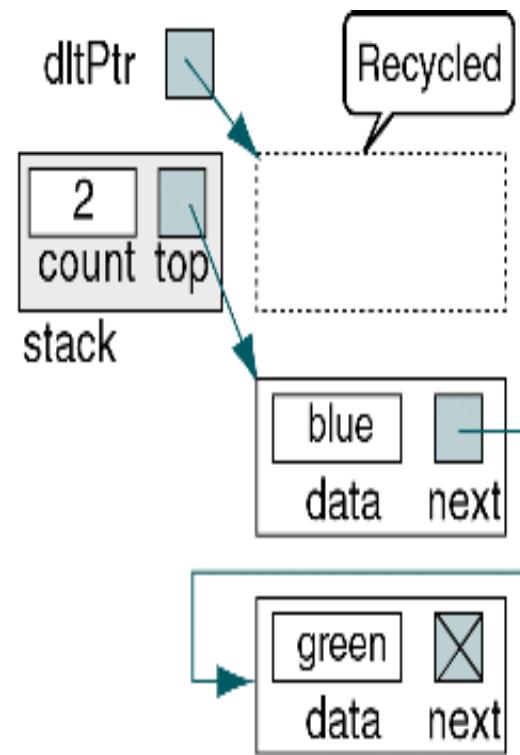
# Push Stack



(b) after



(a) Before



(b) After

FIGURE 3-10 Pop Stack Example

```
algorithm popStack(val stack <head pointer>,
                    ref dataOut <data type>)
```

Pops the item on the top of the stack and returns it to the user.

**Pre** stack is a pointer to the stack head structure.

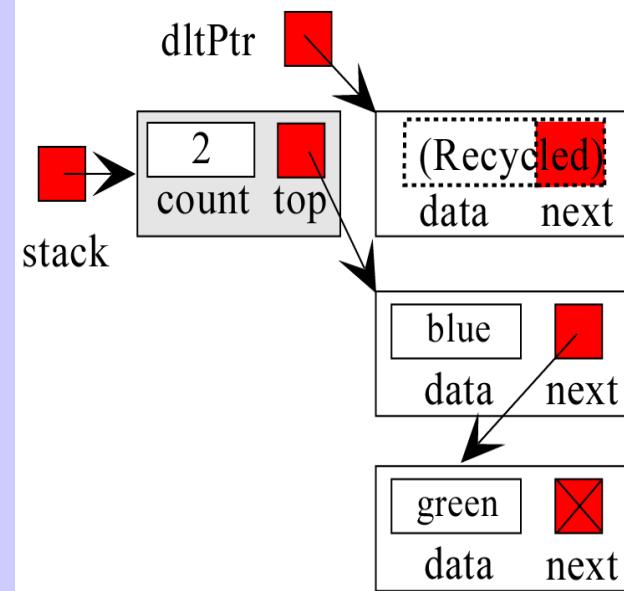
dataOut is a reference variable to receive the data.

**Post** data have been returned to the calling algorithm.

**Return** true if successful; false if underflow

1. if (stack empty)
    1. success = false
  2. else
    1. dltPtr= stack→top
    2. dataOut = stack→top→data
    3. stack→top = stack→top→next
    4. stack→count = stack→count – 1
    5. recycle(dltPtr)
    6. success=true
  3. return success
- end **popStack**

# Pop Stack



(b) after

## ALGORITHM 3-4 Stack Top Pseudocode

```
Algorithm stackTop (stack, dataOut)
This algorithm retrieves the data from the top of the stack
without changing the stack.

    Pre    stack is metadata structure to a valid stack
           dataOut is reference variable to receive data
    Post   Data have been returned to calling algorithm
           Return true if data returned, false if underflow

1 if (stack empty)
    1 set success to false
2 else
    1 set dataOut to data in top node
    2 set success to true
3 end if
4 return success
end stackTop
```

Silme kısmı dışında **Stack Top** mantığı **Pop Stack** mantığı ile aynıdır.

## ALGORITHM 3-5 Empty Stack

```
Algorithm emptyStack (stack)
Determines if stack is empty and returns a Boolean.
    Pre    stack is metadata structure to a valid stack
    Post   returns stack status
    Return true if stack empty, false if stack contains data
1  if (stack count is 0)
    1  return true
2 else
    1  return false
3 end if
end emptyStack
```

Yapısal programlamadaki veri gizleme (data hiding) konseptini gerçekleştirmek için «**Empty Stack**» sağlanmıştır.

- Tüm program yiğin baş yapısına (head structure) erişebiliyorsa, buna gerek yoktur.

Bununla birlikte, yiğin diğer programlarla ilişkilendirilmek üzere ayrı bir şekilde derlenmiş bir program olarak uygulanırsa, çağrıran program yiğin baş düğümüne erişemeyebilir.

- Bu durumlarda, yiğinin boş olup olmadığını belirlemenin bir yolunu bulmak gereklidir.

## ALGORITHM 3-6 Full Stack

```
Algorithm fullStack (stack)
```

Determines if stack is full and returns a Boolean.

Pre stack is metadata structure to a valid stack

Post returns stack status

Return true if stack full, false if memory available

```
1 if (memory not available)
```

```
    1 return true
```

```
2 else
```

```
    1 return false
```

```
3 end if
```

```
end fullStack
```

## ALGORITHM 3-7 Stack Count

```
Algorithm stackCount (stack)
    Returns the number of elements currently in stack.
    Pre    stack is metadata structure to a valid stack
    Post   returns stack count
    Return integer count of number of elements in stack
1  return (stack count)
end stackCount
```

# Destroy Stack

algorithm **destroyStack**(val stack <head pointer>)

This algorithm releases all nodes back to the dynamic memory.

**Pre** stack is a pointer to the stack head structure.

**Post** stack empty and all nodes recycled

1. if (stack not empty)
  1. Loop
    1. temp = stack→top
    2. stack→top = stack→top→link
    3. recycle (temp)
  2. Recycle (stack)
  3. return null pointer

end **destroyStack**

# ADT Implementation

- Soyut veri türü (Abstract Data Type) oluşturmak için bir veri yapısından daha fazlası gereklidir:
  - Yığını destekleyen işlemler (operations) de olmalıdır.

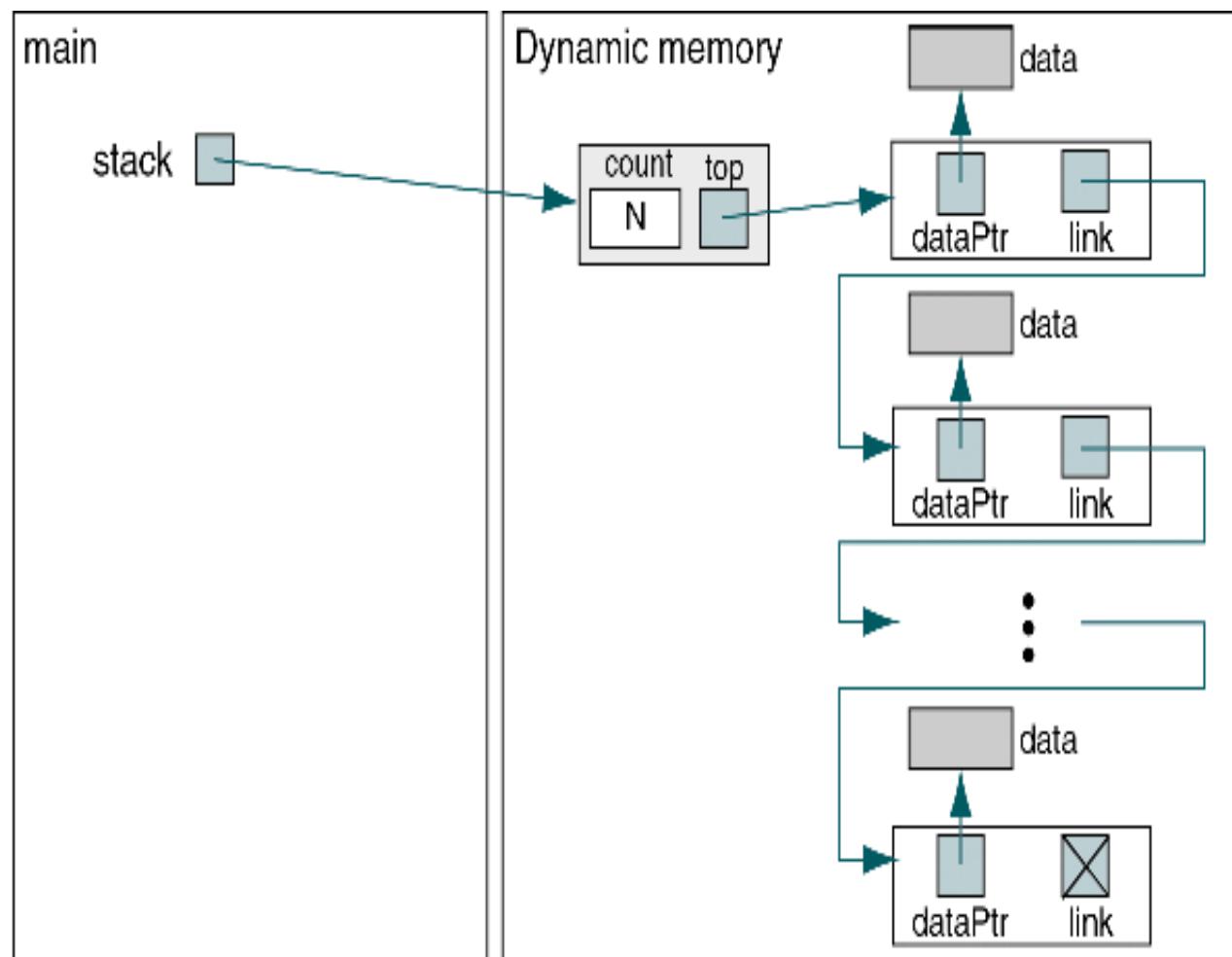


FIGURE 3-12 Stack ADT Structural Concepts

## PROGRAM 3-6 Stack ADT Definitions

```
1 // Stack ADT Type Definitions
2     typedef struct node
3     {
4         void*           dataPtr;
5         struct node*   link;
6     } STACK_NODE;
7
8     typedef struct
9     {
10        int            count;
11        STACK_NODE*   top;
12    } STACK;
```

The stack abstract data type structure

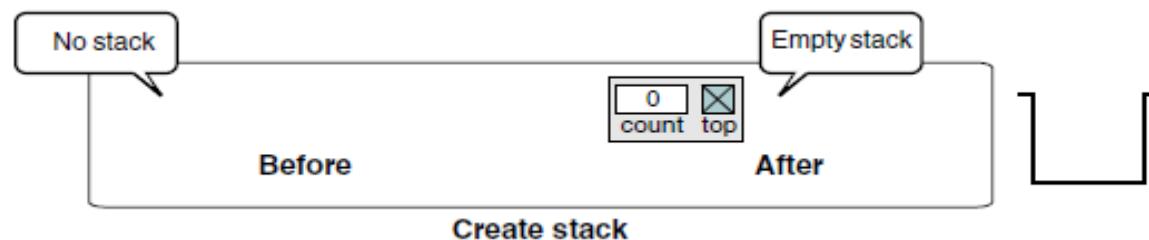
# Create Stack

- Dinamik bellekte bir stack head node için yer tahsis eder
  - **Top pointer**'a null değeri atar
  - **Count** değerini sıfırlar
- Oluşturulan head node'un adresi çağrıran fonksiyona döndürülür

```
stack = createStack ( );
```

## PROGRAM 3-7 ADT Create Stack

```
1  /* ===== createStack =====
2      This algorithm creates an empty stack.
3          Pre Nothing
4          Post Returns pointer to a null stack
5                  -or- NULL if overflow
6 */
7  STACK* createStack (void)
8  {
9  // Local Definitions
10     STACK* stack;
11
12 // Statements
13     stack = (STACK*) malloc( sizeof (STACK));
14     if (stack)
15     {
16         stack->count = 0;
17         stack->top   = NULL;
18     } // if
19     return stack;
20 } // createStack
```



# Push Stack

PROGRAM 3-8 Push Stack

```
1  /* ===== pushStack =====
2   This function pushes an item onto the stack.
3   Pre    stack is a pointer to the stack
4           dataPtr pointer to data to be inserted
5   Post   Data inserted into stack
6   Return true  if successful
7           false if overflow
8 */
9  bool pushStack (STACK* stack, void* dataInPtr)
10 {
11 // Local Definitions
12     STACK_NODE* newPtr;
13
14 // Statements
15     newPtr = (STACK_NODE* ) malloc(sizeof( STACK_NODE));
16     if (!newPtr)
17         return false;
18
19     newPtr->dataPtr = dataInPtr;
20
21     newPtr->link      = stack->top;
22     stack->top        = newPtr;
23
24     (stack->count)++;
25     return true;
26 } // pushStack
```

# Pop Stack

PROGRAM 3-9 ADT Pop Stack

```
1  /* ===== popStack =====
2   This function pops item on the top of the stack.
3   Pre stack is pointer to a stack
4   Post Returns pointer to user data if successful
5   NULL if underflow
6  */
7  void* popStack (STACK* stack)
8  {
9  // Local Definitions
10 void*      dataOutPtr;
```

```
11 STACK_NODE* temp;
12
13 // Statements
14 if (stack->count == 0)
15     dataOutPtr = NULL;
16 else
17 {
18     temp      = stack->top;
19     dataOutPtr = stack->top->dataPtr;
20     stack->top = stack->top->link;
21     free (temp);
22     (stack->count)--;
23 } // else
24 return dataOutPtr;
25 } // popStack
```

# (Retrieve) Stack Top

PROGRAM 3-10 Retrieve Stack Top

```
1  /* ===== stackTop =====
2   Retrieves data from the top of stack without
3   changing the stack.
4   Pre stack is a pointer to the stack
5   Post Returns data pointer if successful
6           null pointer if stack empty
7 */

8 void* stackTop (STACK* stack)
9 {
10 // Statements
11     if (stack->count == 0)
12         return NULL;
13     else
14         return stack->top->dataPtr;
15 } // stackTop
```

# Empty Stack

PROGRAM 3-11 Empty Stack

```
1  /* ===== emptyStack =====
2   This function determines if a stack is empty.
3   Pre  stack is pointer to a stack
4   Post returns 1 if empty; 0 if data in stack
5 */
6  bool emptyStack (STACK* stack)
7  {
8  // Statements
9  return (stack->count == 0);
10 } // emptyStack
```

# Full Stack

PROGRAM 3-12 Full Stack

```
1  /* ===== fullStack =====
2   This function determines if a stack is full.

3   Full is defined as heap full.
4       Pre    stack is pointer to a stack head node
5       Return true if heap full
6           false if heap has room
7   */
8   bool fullStack (STACK* stack)
9   {
10  // Local Definitions
11  STACK_NODE* temp;
12
13 // Statements
14  if ((temp =
15      (STACK_NODE*)malloc (sizeof(*((stack->top))))))
16  {
17      free (temp);
18      return false;
19  } // if
20
21  // malloc failed
22  return true;
23 } // fullStack
```

# Stack Count

PROGRAM 3-13 Stack Count

```
1  /* ===== stackCount =====
2   Returns number of elements in stack.
3   Pre stack is a pointer to the stack
4   Post count returned
5 */
6 int stackCount (STACK* stack)
7 {
8 // Statements
9   return stack->count;
10 } // stackCount
```

# Destroy Stack

PROGRAM 3-14 Destroy Stack

Yığın boyunca  
gezinerek ilk  
önce  
kullanıcının  
**veri**  
**düğümlerinin**  
daha sonra da  
**yığın**  
**düğümlerinin**  
kapladığı  
alanlar geri  
kazanılır.

```
1  /* ===== destroyStack =====
2   This function releases all nodes to the heap.
3     Pre A stack
4     Post returns null pointer
5 */
6 STACK* destroyStack (STACK* stack)
7 {
8 // Local Definitions
9   STACK_NODE* temp;
10
11 // Statements
12 if (stack)
13 {
14   // Delete all nodes in stack
15   while (stack->top != NULL)
16   {
17     // Delete data entry
18     free (stack->top->dataPtr);
19
20     temp = stack->top;
21     stack->top = stack->top->link;
22     free (temp);
23   } // while
24
25   // Stack now empty. Destroy stack head node.
26   free (stack);
27 } // if stack
28 return NULL;
29 } // destroyStack
```

In the loop we walk through the stack, first **recycling** the user's **data nodes** and then **recycling** the **stack nodes**.