

BLM212 Veri Yapıları

Stacks – part 2

(Continuation)

Array Implementation of Stacks
&
Stack Applications

2021-2022 Güz Dönemi

Array Implementation of Stacks

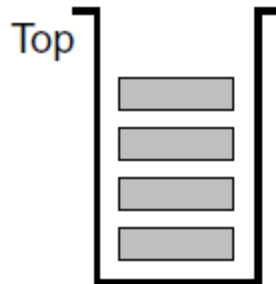
- Bir yığının boyutunun **maksimum** seviyesi, program yazılmadan önce hesaplanabiliyorsa, yığının **dizi** ile gerçekleştirilmesi, **bağlı liste** kullanarak gerçekleştirilen dinamik uygulamadan **daha verimlidir**.
- Ayrıca dizi ile yığın gerçekleştirmek
 - ✓ çok daha kolay ve anlaşılır.

Array Implementation of Stacks

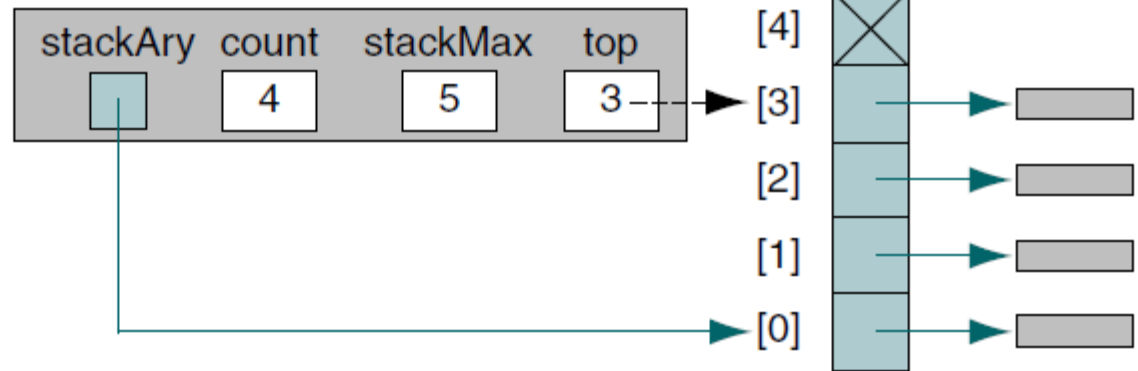
(Kavramsal)

(Fiziksel yapı)

(a) Conceptual



(b) Physical structure



Stack Array Implementation

Maksimum 5 elemanlı bir yığının dizi ile gerçekleştirilmesi

Array Data Structure

Dizi uygulaması için veri yapısında üç değişiklik olur:

1. **Stack top** bir işaretçi yerine bir indekstir.
2. Yığında izin verilen maksimum eleman sayısını tutmamız gerekir
3. Sonraki elemana işaret eden «**next**» alanlarına ihtiyacımız yoktur.
 - ❖ her elemanın kendinden öncekiyle bir **fiziksel komşuluğu** vardır.

Array Data Structure

Veri yapısı önemli ölçüde farklı olsa da, bir **yığının dizi implementasyonu**, bağlı/bağlantılı liste implementasyonunda kullanılan **aynı temel algoritmaları** gerektirir.

ADT deklarasyonu **PROGRAM F-1** de görülmektedir.

PROGRAM F-1 Stack Array Definition

```
1  // Stack Definitions for Array Implementation
2  typedef struct
3  {
4      void**  stackAry;
5      int     count;
6      int     stackMax;
7      int     top;
8  } STACK;
9
10 // Prototype Declarations
11 STACK* createStack (int maxSize);
12 STACK* destroyStack (STACK* queue);
13
14 bool  pushStack  (STACK* stack, void* itemPtr);
15 void* popStack   (STACK* stack);
16 void* stackTop   (STACK* stack);
17 int   stackCount (STACK* stack);
18 bool  emptyStack (STACK* stack);
19 bool  fullStack  (STACK* stack);
20 // End of Stack ADT Definitions
```

Create Stack

PROGRAM F-2 Create Stack

```
1  /* ===== createStack =====
2      This algorithm creates an empty stack by allocating
3      the head structure and the array from dynamic memory.
4      Pre    maxSize is max number of elements
5      Post   returns pointer to stack head structure
6             -or- NULL if overflow
7  */
8  STACK* createStack (int maxSize)
9  {
10     // Local Definitions
11     STACK* stack;
12
13     // Statements
14     stack = (STACK*) malloc( sizeof (STACK));
15     if (!stack)
16         return NULL;
17
18     // Head allocated. Initialize & allocate stack.
19     stack->count    = 0;
20     stack->top      = -1;
21     stack->stackMax = maxSize;
22     stack->stackAry = (void**)calloc(stack->stackMax,
23                                     sizeof(void*));
24     if (!stack->stackAry)
25     {
26         free (stack);
27         return NULL;
28     } // if
29     return stack;
30 } // createStack
```

calloc vs malloc?

The name **malloc** and `calloc()` are library functions that allocate memory dynamically. It means that memory is allocated during runtime(execution of the program) from heap segment.

- **Initialization:** `malloc()` allocates memory block of given size (in bytes) and returns a pointer to the beginning of the block. `malloc()` doesn't initialize the allocated memory. If we try to access the content of memory block then we'll get garbage values.



```
void * malloc( size_t size );
```



`calloc()` allocates the memory and also initializes the allocated memory. If we try to access the content of these blocks then we'll get 0.



```
void * calloc( size_t num, size_t size );
```



- **Number of arguments:** Unlike `malloc()`, `calloc()` takes two arguments.
 - 1) Number of blocks to be allocated.
 - 2) Size of each block.
- **Return Value:** After successful allocation in `malloc()` and `calloc()`, a pointer to the allocated memory is returned otherwise **NULL** value is returned which indicates memory allocation failed.

void * calloc (size_t nitems, size_t size) C kütüphanesi fonksiyonu istenen hafızayı tahsis eder ve onun başlangıç adresine işaret eden pointer döndürür.

malloc ve calloc'taki fark, **calloc** tahsis edilen hafıza hücrelerini **sıfıra set ederken**, malloc bunu yapmaz.

malloc ile yeni tahsis edilmiş bellek hücresinin içeriğine erişmek istediğimizde garbage değerleri elde ederiz.

calloc vs malloc?

```
// C program to demonstrate the use of calloc()
// and malloc()
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *arr;

    // malloc() allocate the memory for 5 integers
    // containing garbage values
    arr = (int *)malloc(5 * sizeof(int)); // 5*4bytes = 20 bytes

    // Deallocates memory previously allocated by malloc() function
    free( arr );

    // calloc() allocate the memory for 5 integers and
    // set 0 to all of them
    arr = (int *)calloc(5, sizeof(int));

    // Deallocates memory previously allocated by calloc() function
    free(arr);

    return(0);
}
```

Push Stack

PROGRAM F-3 Push Stack

```
1  /* ===== pushStack =====
2      This function pushes an item onto the stack.
3      Pre  stack is pointer to stack head structure
4          dataInPtr is pointer to be inserted
5      Post returns true if success; false if overflow
6  */
7  bool pushStack (STACK* stack, void* dataInPtr)
8  {
9      // Statements
10     if (stack->count == stack->stackMax)
11         return false;
12
13     (stack->count)++;
14     (stack->top)++;
15     stack->stackAry[stack->top] = dataInPtr;
16
17     return true;
18 } // pushStack
```

Pop Stack

PROGRAM F-4 Pop Stack

```
1  /* ===== popStack =====
2      This function pops the item on the top of the stack.
3      Pre  stack is pointer to stack head structure
4      Post returns pointer to user data if successful
5              NULL if underflow
6  */
7  void* popStack (STACK* stack)
8  {
9      // Local Declarations
10     void* dataPtrOut;
11
12     // Statements
13     if (stack->count == 0)
14         dataPtrOut = NULL;
15     else
16     {
17         dataPtrOut = stack->stackAry[stack->top];
18         (stack->count)--;
19         (stack->top)--;
20     } // else
21
22     return dataPtrOut;
23 } // popStack
```

Stack Top

PROGRAM F-5 Stack Top

```
1  /* ===== stackTop =====
2      This function retrieves the data from the top of the
3      stack without changing the stack.
4          Pre  stack is pointer to the stack
5          Post returns data pointer if successful
6              -or- 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->stackAry[stack->top];
15 } // stackTop
```

Empty Stack, Full Stack, Stack Count

PROGRAM F-6 Empty Stack

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

PROGRAM F-7 Full Stack

```
1  /* ===== fullStack =====
2      This function determines if a stack is full.
3      Pre  stack is a pointer to a stack head structure
4      Post returns true if full; false if empty elements
5  */
6  bool fullStack (STACK* stack)
7  {
8      // Statements
9      return (stack->top == stack->stackMax);
10 } // fullStack
```

PROGRAM F-8 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 F-9 Destroy Stack

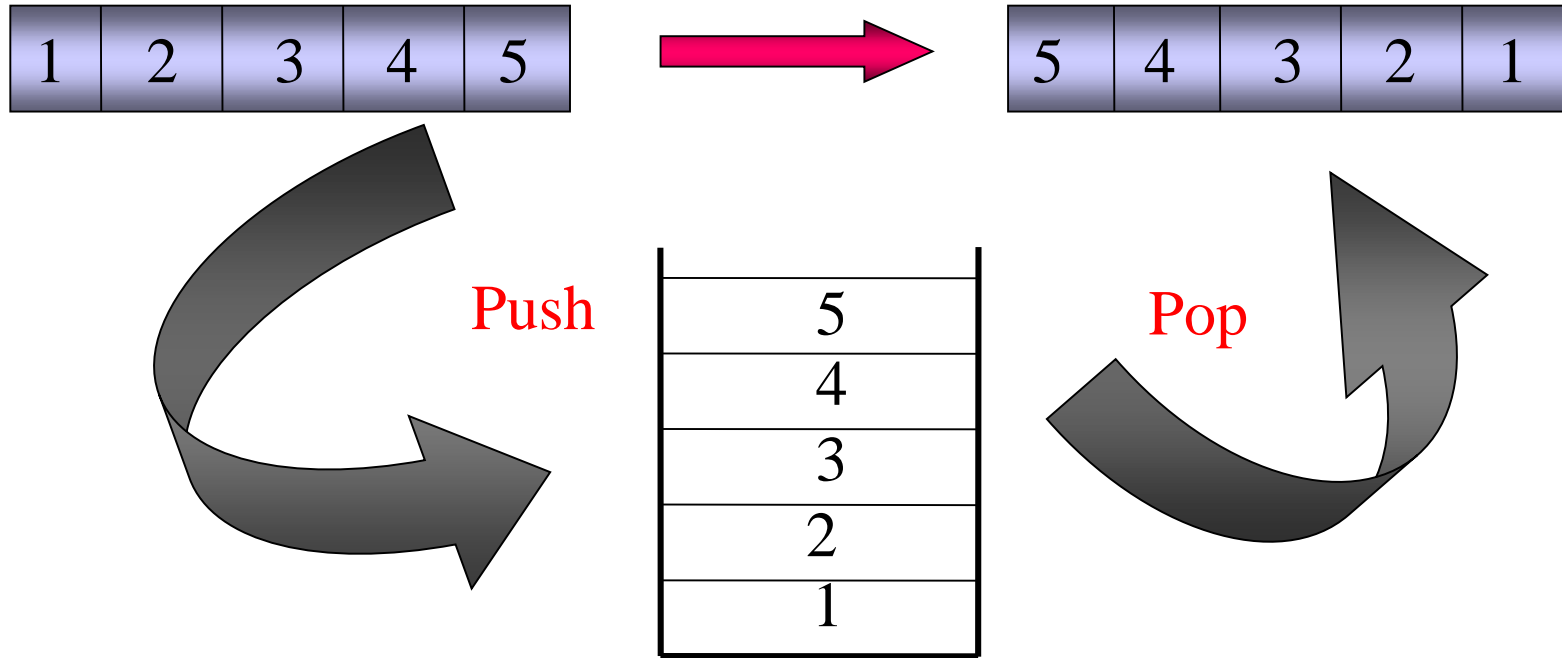
```
1  /* ===== destroyStack =====
2      This function releases all memory to the heap.
3      Pre  stack is pointer to stack head structure
4      Post returns null pointer
5  */
6  STACK* destroyStack (STACK* stack)
7  {
8      // Statements
9      if (stack)
10     {
11         // Release data memory
12         for (int i = 0; i < stack->count; i++)
13             free (stack->stackAry[i]);
14
15         // Release stack array
16         free (stack->stackAry);
17
18         // Now release memory for stack head
19         free (stack);
20     } // if stack
21     return NULL;
22 }
```

Stack Applications

- Yığın uygulamaları 4 geniş kategoriye ayrılabilir:
 1. **Reversing** data (Tersine çevirme)
 2. **Parsing** data (Öğelerine ayırma)
 3. **Postponing** data usage (Geciktirme/Erteleme)
 4. **Backtracking** steps (Geri-izsürme)

Stack Applications

Reversing Data



Reversing a list

- Yığın uygulamalarından biri listeyi tersine çevirmektir.
- Örneğin, **PROGRAM 3-15** bir tam sayı dizisini okur ve bunları tersten yazdırır.

PROGRAM 3-15 Reverse a Number Series

```
1  /* This program reverses a list of integers read
2     from the keyboard by pushing them into a stack
3     and retrieving them one by one.
4     Written by:
5     Date:
6  */
7  #include <stdio.h>
8  #include <stdbool.h>
9  #include "stacksADT.h"
10
11 int main (void)
12 {
13     // Local Definitions
14     bool done = false;
15     int* dataPtr;
16
17     STACK* stack;
18
19     // Statements
20     // Create a stack and allocate memory for data
21     stack = createStack ();
22
23     // Fill stack
24     while (!done)
25     {
```

continued

PROGRAM 3-15 Reverse a Number Series (continued)

```
26         dataPtr = (int*) malloc (sizeof(int));
27         printf ("Enter a number: <EOF> to stop: ");
28         if ((scanf ("%d" , dataPtr)) == EOF
29             || fullStack (stack))
30             done = true;
31         else
32             pushStack (stack, dataPtr);
33     } // while
34
35     // Now print numbers in reverse
36     printf ("\n\nThe list of numbers reversed:\n");
37     while (!emptyStack (stack))
38     {
39         dataPtr = (int*)popStack (stack);
40         printf ("%3d\n", *dataPtr);
41         free (dataPtr);
42     } // while
43
44     // Destroying Stack
45     destroyStack (stack);
46     return 0;
47 } // main
```

Results:

```
Enter a number: <EOF> to stop: 3
Enter a number: <EOF> to stop: 5
Enter a number: <EOF> to stop: 7
Enter a number: <EOF> to stop: 16
Enter a number: <EOF> to stop: 91
Enter a number: <EOF> to stop:
```

The list of numbers reversed:

```
91
16
7
5
3
```

Bu basit programda dikkat edilmesi gereken önemli nokta, yığın yapısına asla doğrudan başvurmadığımızdır. Tüm yığın referansları, yığın ADT arayüzü üzerinden yapıldı. Bu, kapsülleme (**encapsulation**) ve fonksiyon yeniden kullanılabilirliği (**reusability**) yapısal programlama ilkeleri açısından önemlidir.

Convert Decimal to Binary

- Bir diziyi tersine çevirme fikri, ondalık sayıları ikili sayıya dönüştürmek gibi klasik sorunların çözümünde kullanılabilir.

```
1 read (number)
2 loop (number > 0)
  1 set digit to number modulo 2
  2 print (digit)
  3 set number to quotient of number / 2
3 end loop
```

- Ancak bu kodda bir sorun var:
 - İkili sayıyı geriye doğru oluşturur. Böylece decimal 19, binary 10011 yerine 11001 olarak ortaya çıkar.

Convert Decimal to Binary

- Bir yığın kullanarak bu sorunu çözebiliriz.
 - Binary digit üretildiği anda ekrana basmak yerine, **yığına iteriz (push)**.
 - Daha sonra, sayı tamamen dönüştürüldükten sonra **yığından çekip (pop)** ve tek bir satırda yan yana ekrana basarız.

Convert Decimal to Binary

PROGRAM 3-16 Convert Decimal to Binary

```
1  /* This program reads an integer from the keyboard
2     and prints its binary equivalent. It uses a stack
3     to reverse the order of 0s and 1s produced.
4     Written by:
5     Date:
6  */
7  #include <stdio.h>
8  #include "stacksADT.h"
9
10 int main (void)
11 {
12     // Local Definitions
13     unsigned int    num;
14     int*    digit;
15     STACK* stack;
16
17     // Statements
18     // Create Stack
19     stack = createStack ();
20
21     // prompt and read a number
22     printf ("Enter an integer:      ");
23     scanf ("%d", &num);
24
25     // create 0s and 1s and push them into the stack
26     while (num > 0)
```

continued

Convert Decimal to Binary

PROGRAM 3-16 Convert Decimal to Binary (*continued*)

```
27     {
28         digit = (int*) malloc (sizeof(int));
29         *digit = num % 2;
30         pushStack (stack, digit);
31         num = num / 2;
32     } // while
33
34     // Binary number created. Now print it
35     printf ("The binary number is : ");
36     while (!emptyStack (stack))
37     {
38         digit = (int*)popStack (stack);
39         printf ("%ld", *digit);
40     } // while
41     printf ("\n");
42
43     // Destroying Stack
44     destroyStack (stack);
45     return 0;
46 } // main
```

Results:

Enter an integer: 45
The binary number is : 101101

Stack Applications

Parsing Data

Daha sonra işlenmek üzere verileri bağımsız parçalara ayırır.

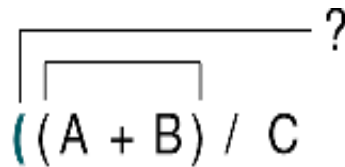


Parsing

- Yığınların bir başka uygulaması da öğelerine ayırma/ayrıştırmadır.
- Daha sonraki bir aşamada işlenmek üzere verileri bağımsız parçalara ayıran bir mantıktır.
- Örneğin, bir kaynak programı makine diline çevirmek için, bir derleyicinin programı aşağıdaki gibi ayrı bölümlere ayırması gerekir.
 - anahtar kelimeler (**keywords**),
 - adlar (**names**)
 - simgeler/belirteçler (**tokens**)

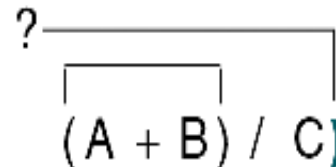
Parse Parentheses

Yaygın bir programlama problemi, bir cebirsel ifadedeki eşleştirilmemiş parantezlerdir.



$((A + B) / C$

(a) Opening parenthesis not matched



$(A + B) / C)$

(b) Closing parenthesis not matched

FIGURE 3-14 Unmatched Parentheses Examples

Lab Uygulaması 1: Kitaptaki PROGRAM 3-17 Verify Parentheses Paired in Source Program C kodu incelenecek.

ALGORITHM 3-9 Parse Parentheses

Algorithm parseParens

This algorithm reads a source program and parses it to make sure all opening-closing parentheses are paired.

```
1 loop (more data)
  1 read (character)
  2 if (opening parenthesis)
    1 pushStack (stack, character)
  3 else
    1 if (closing parenthesis)
      1 if (emptyStack (stack))
        1 print (Error: Closing parenthesis not matched)
      2 else
        1 popStack(stack)
      3 end if
    2 end if
  4 end if
2 end loop
3 if (not emptyStack (stack))
  1 print (Error: Opening parenthesis not matched)
end parseParens
```

Postponement

- Genellikle bir uygulamanın mantığı, verilerin kullanımının daha sonraki bir noktaya kadar ertelenmesini gerektirir.
- Eğer uygulamanız verilerin kullanımının bir süre ertelenmesini gerektiriyorsa, yığın kullanılabilir.

Stack Applications

Posponement

- Aritmetik ifadeler, üç farklı formatta temsil edebilir:

1. Prefix $+ a b$
2. Infix $a + b$
3. Postfix $a b +$

Aritmetik öncelik;

toplama ve çıkarmadan önce çarpma ve bölme gelir

Infix to Postfix Transformation

- **Infix** gösterimin dezavantajlarından birisi
 - operatörlerin değerlendirmesini kontrol etmek için parantez kullanmamız gerekir.
 - Dolayısıyla parantez ve iki operatör öncelikli sınıf içeren bir değerlendirme yöntemine ihtiyaç vardır
- **Postfix** ve **prefix** notasyonlarında parantez gerekli değildir;
 - Dolayısıyla yalnızca bir değerlendirme kuralı vardır.

Infix to Postfix Transformation

- Bazı yüksek seviyeli dillerde **infix** gösterim kullanılsa da, bu ifadeler doğrudan değerlendirilemez.
 - Aksine, ifadelerin değerlendirileceği (*evaluation*) sırayı belirlemek için analiz edilmelidirler.
- Yaygın bir değerlendirme tekniği, ifadeleri değerlendirmek için kod oluşturmadan önce **postfix** notasyonuna dönüştürmektir.

Infix to Postfix Transformation

- **infix**'ten **postfix**'e dönüşüm, yüksek seviyeli dillerde yazılmış kaynak kodların ön işlenmesinin çok önemli bir parçasıdır.
- Bu dönüşüm operand'ları operatörlerden ayırır

Infix to Postfix Transformation

- $A * B \rightarrow AB^*$
- Hangi dönüşüm doğrudur?
- $A*B+C \rightarrow ?$ ABC^*+
- $A*B+C \rightarrow ? \rightarrow AB^*C+$
- Doğru temsili sağlamak için bir öncelik kuralı göz önünde bulundurulmalıdır.
- Dolayısıyla **ikinci gösterim** doğrudur.

Infix to Postfix Transformation

Manual Transformation

– Kurallar

1. Aritmetik öncelikleri ve belirgin parantezleri kullanarak ifadeyi tamamen parantezli yapıya kavuşturun.
2. En içteki ifadelerden başlayarak her parantez içindeki tüm infix notasyonları, postfix'e çevirin.
(Postfix notasyonuna dönüşüm, operatörü ifadenin kapanış parantezinin önüne getirerek yapılır.)
3. Bütün parantezleri atın.

Infix to Postfix Transformation

Basit bir örnek

$$A + B * C$$

- Place the paranthesis

$$(A + (B * C))$$

- Replace it in **postfix** format

$$(A(BC^*)+)$$

- Remove all paranthesis

$$ABC^*+$$

Implementation by computer is too hard!

Infix to Postfix Transformation

Karmaşık bir örnek

$(A + B) * C + D + E * F - G$

Step 1 adds parentheses.

$((((A + B) * C) + D) + (E * F)) - G$

Step 2 then moves the operators.

$((((A B +) C *) D +) (E F *) +) G -)$

Step 3 removes the parentheses.

$A B + C * D + E F * + G -$

Implementation by computer is too hard!

Infix to Postfix Transformation

Algorithmic Transformation

- Yığın kullanarak kolayca yapılır
 - Görünen en basit çözüm, tüm operatörleri yığına itmek ve sonra yığından çekmektir.
- İki operandın çarpımından oluşan basit bir örnek

A * B converts to A B *

- Problem çarpma operatörünün nasıl kullanılacağıdır;
- sağdaki operandı (B) okuyana kadar operatörü çıktıya koymayı ertelememiz gerekiyor

Infix to Postfix Transformation

Algorithmic Transformation

- Daha karmaşık bir örnek

$A * B + C$ converts to $A B * C +$

- Operatörleri daha önce yaptığımız gibi yığına koyarsak ve tüm operandlar okunduktan sonra yığından çekip postfix ifadesine yönlendirirsek, **yanlış** sonuç alırız.
- Bir şekilde, iki operatörü doğru operandlar ile eşleştirmeliyiz.
- Olası bir kural, bir operatörü yalnızca başka bir operatör gelene kadar ertelemek olabilir.
 - Ardından, ikinci operatörü yığına itmeden önce, ilkini çekip çıktı ifadesine yerleştirebiliriz.

Infix to Postfix Transformation

Algorithmic Transformation

- Bu mantık bu örnek için işler, ancak başkaları için olmaz. Aşağıdaki örneği düşünün.

A + B * C converts to A B C * +

- Daha önce bahsedildiği gibi, infix ifadeleri, operandların ve operatörlerin bir ifadede nasıl gruplanacağını belirlemek için bir **öncelik kuralı** (precedence rule) kullanır.
- infix'i postfix'e dönüştürdüğümüzde de aynı kuralı kullanabiliriz.
- Bir operatörü yığına itmemiz gerektiğinde, önceliği yığının tepesindeki operatörden daha yüksekse, devam edip yığına iteriz.
- Tersine, yığının tepesindeki operatör, mevcut operatörden daha yüksek önceliğe sahipse, yığından çekilir ve çıktı ifadesine yerleştirilir.

Infix to Postfix Transformation

- Let us obtain $A+B*C \rightarrow ABC*+$

| | Expression | Stack |
|------------------------------------------------------------------------------|------------|-------|
| Copy operand A to output expression. | A | |
| Push operator $+$ into stack. | A | $+$ |
| Copy operand B to output expression. | AB | $+$ |
| Push operator $*$ into stack (Priority of $*$ is higher than $+$) | AB | $*+$ |
| Copy operand C to output expression. | ABC | $*+$ |
| Pop operator $*$ and copy to output expression. | $ABC*$ | $+$ |
| Pop operator $+$ and copy to output expression. | $ABC*+$ | |

Infix to Postfix Transformation

Algorithmic Transformation

- Mantığı tamamlamak için bir kural daha koymalıyız.
- Daha düşük veya eşit önceliğe sahip mevcut bir operatör yığının tepesindeki operatörü yığından çıkarmaya zorlarsa, yığının yeni tepe operatörünü kontrol etmeliyiz.
- Yığının tepesindeki operatör mevcut operatörden daha büyükse, yığından çekilip çıktı ifadesine koyulur. Bu işlemden sonra yığının tepesindeki yeni operatör mevcut operatörle aynı önceliğe sahipse o da yığından çekilip çıktı ifadesine koyulur.
- Sonuç olarak, yeni operatörü yığına itmeden önce birkaç operatörü çıktı ifadesine götürebiliriz.

Infix to Postfix Transformation

- Daha da karmaşık bir örnek
 - Bütün temel operatörleri kullanır

$A + B * C - D / E$ converts to $A B C * + D E / -$

| | Infix | Stack | Postfix |
|-----|-------------|--------|-----------|
| (a) | $A+B*C-D/E$ | | |
| (b) | $+B*C-D/E$ | | A |
| (c) | $B*C-D/E$ | + | A |
| (d) | $*C-D/E$ | + | AB |
| (e) | $C-D/E$ | * + | AB |
| (f) | $-D/E$ | * + | ABC |
| (g) | D/E | - | ABC*+ |
| (h) | $/E$ | - | ABC*+D |
| (i) | E | / - | ABC*+D |
| (j) | | / - | ABC*+DE |
| (k) | | | ABC*+DE/- |

Infix to Postfix Transformation

- Nihai algoritma (En genel durum):

1) i^{th} operand çıktı ifadesine kopyalanır.

2) i^{th} operator yığına itilir.

3) Bir sonraki operand ($i+1^{st}$) çıktı ifadesine kopyalanır

4) Bir sonraki operatorün önceliği, i . operatörün önceliğinden yüksekse, yığına itilir, aksi takdirde i . operatör yığından çekilir, çıktı ifadesine kopyalanır ve ardından ele alınan operatör yığına itilir.

5) Son operand çıktı ifadesine kopyalanana kadar önceki adımları tekrarlanır.

6) Yığında kalan operatörler çekilip çıktı ifadesine kopyalanır.

Infix to Postfix Transformation

- Priorities:
- 2: $*$ $/$
- 1: $+$ $-$
- 0: $($

Example infix formula: $(A+(B*C))$

ABC^*+

ALGORITHM 3-10 Convert Infix to Postfix

```

Algorithm inToPostFix (formula)
Convert infix formula to postfix.
  Pre    formula is infix notation that has been edited
         to ensure that there are no syntactical errors
  Post   postfix formula has been formatted as a string
  Return postfix formula
1 createStack (stack)
2 loop (for each character in formula)
  1 if (character is open parenthesis)
    1 pushStack (stack, character)
  2 elseif (character is close parenthesis)
    1 popStack (stack, character)
    2 loop (character not open parenthesis)
      1 concatenate character to postFixExpr
      2 popStack (stack, character)
    3 end loop
  3 elseif (character is operator)
    Test priority of token to token at top of stack
    1 stackTop (stack, topToken)
    2 loop (not emptyStack (stack)
      AND priority(character) <= priority(topToken))
      1 popStack (stack, tokenOut)
      2 concatenate tokenOut to postFixExpr
      3 stackTop (stack, topToken)
    3 end loop
    4 pushStack (stack, token)
  4 else
    Character is operand
    1 Concatenate token to postFixExpr
  5 end if
3 end loop
Input formula empty. Pop stack to postFix
4 loop (not emptyStack (stack))
  1 popStack (stack, character)
  2 concatenate token to postFixExpr
5 end loop
6 return postFix
end inToPostFix

```

Lab Uygulaması 2:
 Kitaptaki PROGRAM 3-18
 Convert Infix to Postfix C
 kodu incelenecek.

Evaluating Postfix Expressions

- Şimdi daha önce geliştirdiğimiz postfix ifadeleri değerlendirmek için yığın (**stack**) ertelemeyi nasıl kullanabileceğimizi görelim.

A B C + *

- ve A'nın **2**, B'nin **4** ve C'nin **6** olduğunu varsayarsak, ifadenin değeri nedir?

Evaluating Postfix Expressions

- Dikkat edilmesi gereken ilk şey, operandların operatörlerden önce geldiğidir.
- Bu, operandların kullanımını ertelememiz gerektiği anlamına gelir,
 - bu sefer operatörler değil!
- Bu yüzden operandları yığına iteriz.
- **Bir operatör bulduğumuzda, yığının tepesinden iki operan çekeriz ve işlemi gerçekleştiririz.**

Stack Applications

Evaluation of Postfix Expression

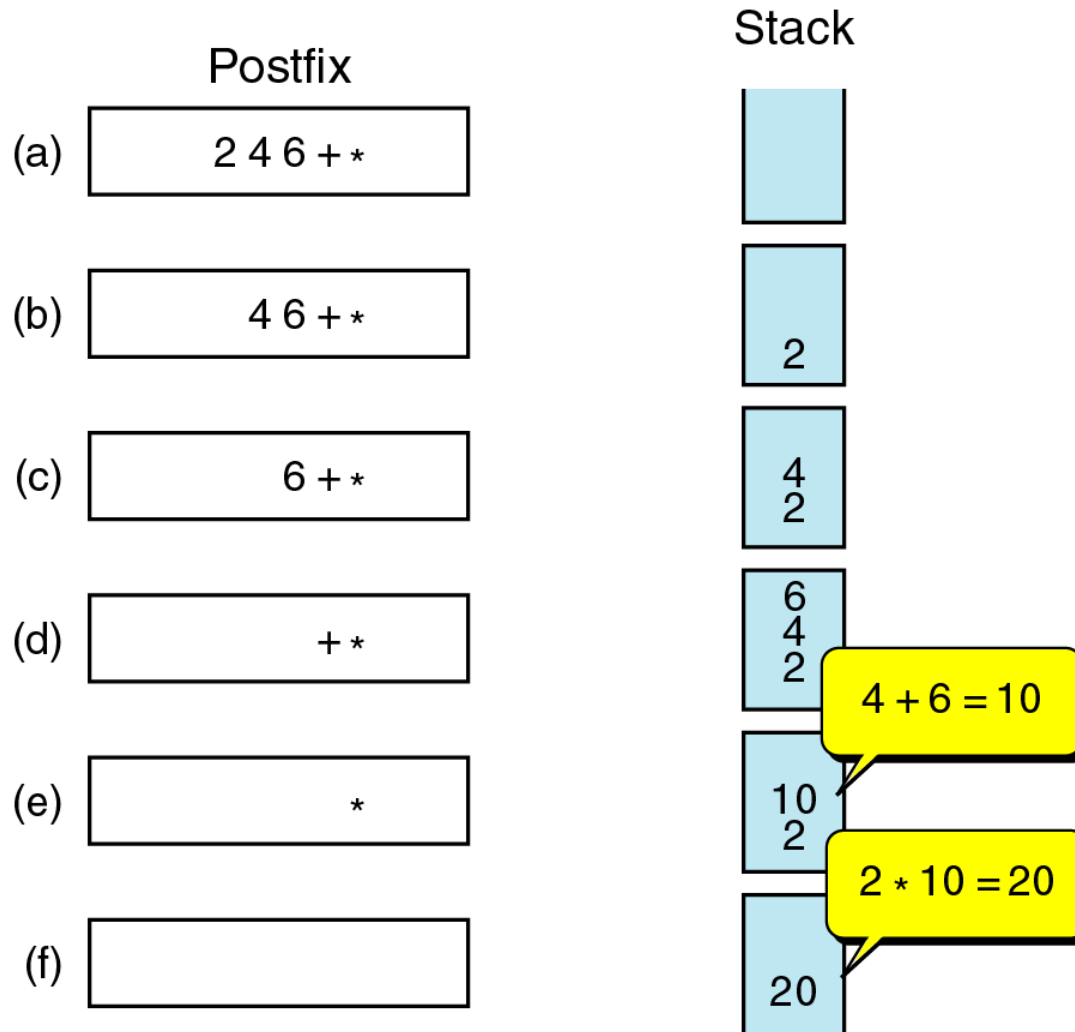


Figure 4-13

Evaluating Postfix Expressions

ALGORITHM 3-11 Evaluation of Postfix Expressions

Algorithm postFixEvaluate (expr)

This algorithm evaluates a postfix expression and returns its value.

Pre a valid expression

Post postfix value computed

Return value of expression

1 createStack (stack)

2 loop (for each character)

1 if (character is operand)

1 pushStack (stack, character)

2 else

1 popStack (stack, oper2)

2 popStack (stack, oper1)

3 operator = character

4 set value to calculate (oper1, operator, oper2)

5 pushStack (stack, value)

3 end if

3 end loop

4 popStack (stack, result)

5 return (result)

end postFixEvaluate

Lab Uygulaması 3: Kitaptaki PROGRAM 3-19 Evaluate Postfix Expression C kodu incelenecek.

Excercise

- Algoritmik yöntemi (yığın) kullanarak aşağıdaki infix ifadesini postfix ifadesine çevirin.

$a+b*c-d$

Solution: $abc*+d-$

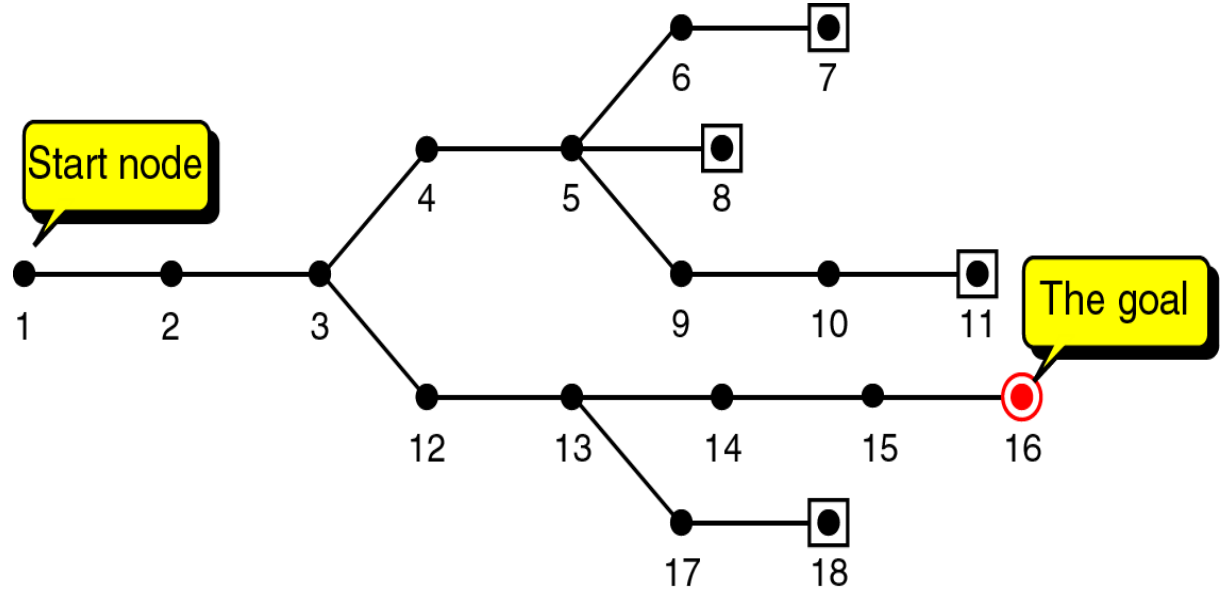
- Infix String : $a+b*c-d$
- The first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.
- Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.
- The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.
- Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :
- End result :
- * Infix String : $a+b*c-d$
- * Postfix String : $abc*+d-$

Backtracking

- **Geri-izsürme** (Backtracking), bir takım noktalardan belirli bir hedefe giden uygun bir yol bulma yöntemidir.
 - karar analizinde,
 - uzman sistemlerde ve
 - bilgisayar oyunlarında yaygın olarak kullanılır.
- Hedef arama (goal seeking), istenen bir hedefe giden benzersiz yolu bulmak için kullanılan tipik bir geri-izsürme algoritmasıdır.

Backtracking

Hedef arama uygulaması örneği



Backtracking Example

- Problemi canlandırmanın bir yolu, adımları birkaç alternatif yol içeren bir **graf** biçiminde yerleştirmektir.
- Şekildeki yollardan sadece bir tanesi istenen **hedefe** ulaşmaktadır.
- Şekle baktığımızda doğru yolu hemen görebilsek de,
 - doğru yolu belirlemek için bilgisayarın bir **algoritma**ya ihtiyacı vardır.

Backtracking

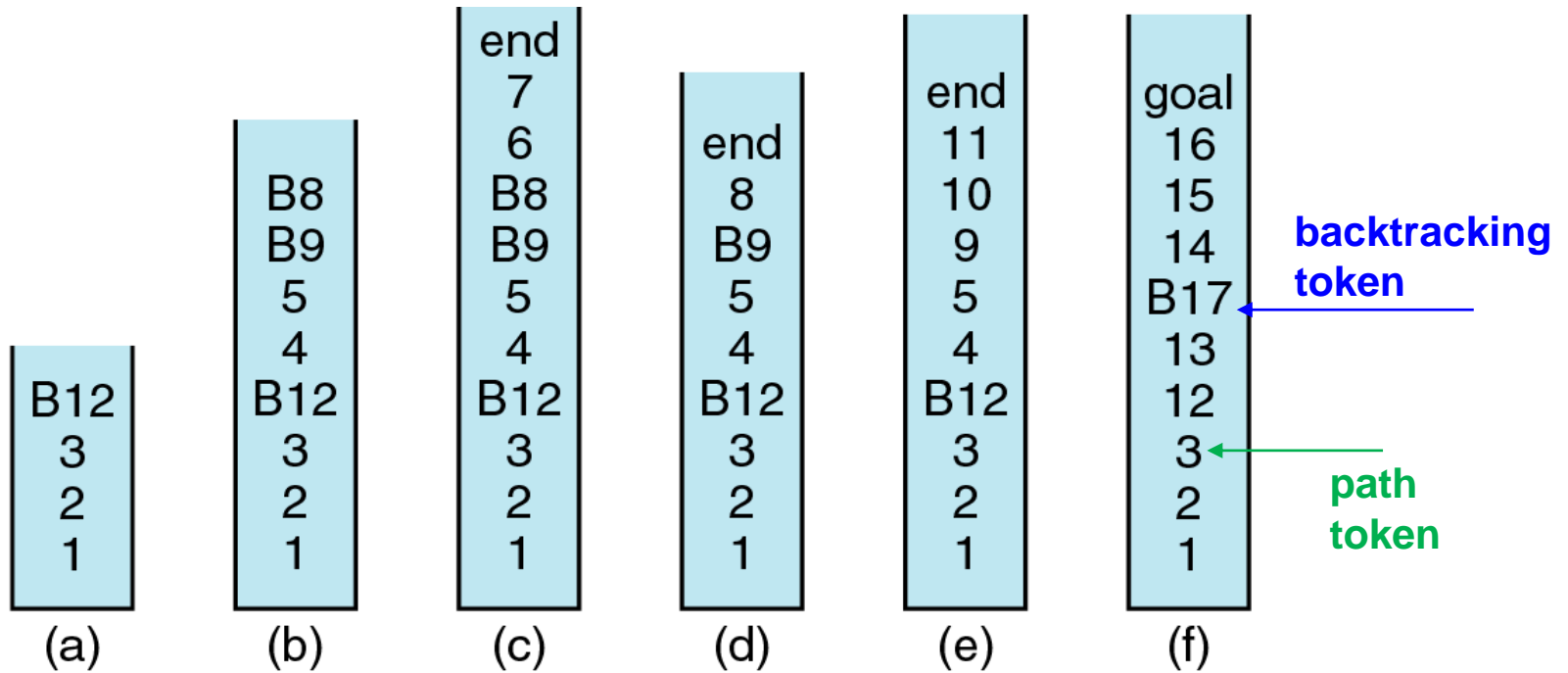
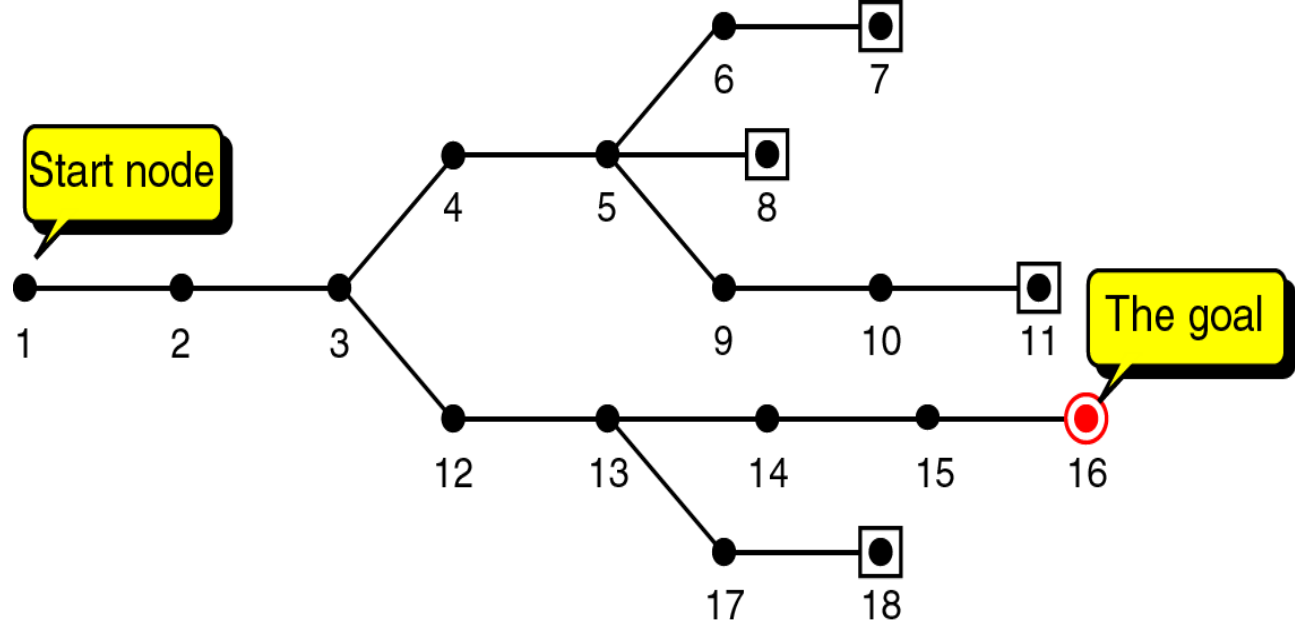
- ❖ Bu iş için kafamızda bir algoritma formülize edelim.
 - Ne zaman bir karar noktasına (yol ayrımına) varsak, nerede olduğunu hatırlamalıyız,
 - böylece **gerektiğinde geri dönebiliriz.**
 - Geri izsürdüğümüzde, ileriye devam etmeden önce en yakın noktaya kadar geri gitmek istiyoruz;
 - yani, tekrar en baştan başlamak istemiyoruz.
 - Bu problemin çözümü için **LIFO veri yapısı** kullanılır
 - ✓ Yani **yığın (stack)**

Backtracking

- Peki bu iş için neyi yığına koyacağız?
- **Yalnızca hedefi** içeren düğümü bulmamız gerekirse,
 - **dallanma noktası düğümlerini** yığına koyarız.
- Ancak, işimiz bittiğinde hedefimize giden yolu yazdırmak istiyorsak,
 - **Geçerli yoldaki düğümleri** yığına koymalıyız
- Yığına iki şey koyduğumuz için, onları birbirinden ayırmamız gerekir.

Backtracking

- Yığına iki şey koyduğumuz için, onları birbirinden ayırmamız gerekir.
 - Bunu bir bayrak (**flag**) ile yapabiliriz.
 - Düğüm geçerli yoldaki düğümlerden ise, bir yol belirtecini (**path token**) yığına iteriz.
 - Bir geri-izsürme noktası saklıyorsak, bayrağı bir geri-izsürme belirtecine (**backtracking token**) ayarlarız.



Backtracking Stack Operation

ALGORITHM 3-12 Print Path to Goal

```
Algorithm seekGoal (map)
This algorithm determines the path to a desired goal.
  Pre  a graph containing the path
  Post path printed
1 createStack (stack)
2 set pMap to starting point
3 loop (pMap not null AND goalNotFound)
  1 if (pMap is goal)
    1 set goalNotFound to false
  2 else
    1 pushStack (stack, pMap)
    2 if (pMap is a branch point)
      1 loop (more branch points)
        1 create branchPoint node
        2 pushStack (stack, branchPoint)
      2 end loop
    3 end if
    4 advance to next node
  3 end if
4 end loop
5 if (emptyStack (stack))
  1 print (There is no path to your goal)
6 else
  1 print (The path to your goal is:)
  2 loop (not emptyStack (stack))
    1 popStack (stack, pMap)
    2 if (pMap not branchPoint)
      1 print(map point)
    3 end if
  3 end loop
  4 print (End of Path)
7 end if
8 return
end seekGoal
```

Stacks and Subroutines

- Tüm parametreler, çağıran programdan alt yordama (veya tersi yönde) yığın üzerinden aktarılır.
- Gerekli tüm sistem verileri (işlemci registerlerinin içeriği, işlemci durum kelimesinin (PSW) içeriği, geri dönüş adresi, vb.) bir alt yordam çağrılmadan önce yığına itilmelidir.

```
program testPower
```

```
1 read (base, exp)
```

```
2 result = power (base, exp)
```

```
3 print ("base**exp is", result)
```

```
end testPower
```

```
algorithm power (base, exp)
```

```
1 set num to 1
```

```
2 loop while exp greater 0
```

```
1 set num to num * base
```

```
2 decrement exp
```

```
3 end loop
```

```
4 return num
```

```
end power
```

When power is **called**, the **stackframe** is created and **pushed** into a system stack.

FIGURE 3-22 Call and Return

When it **concludes**, the **stackframe** is **popped**, the local variables are replaced, the return value is stored, and processing resumes in the calling algorithm.

Stacks and Subroutines

- **stack frame**, belirli bir alt yordam için yığında tahsis edilen bir alt alandır.
- Bir **stack frame** 4 farklı eleman içerir:
 - 1) Çağırılan (called) algoritma tarafından işlenecek parametreler;
 - 2) Çağırılan (calling) algoritmadaki sistem verileri (register ve PSW içeriği)
 - 3) Çağırılan algoritmada dönüş adresi;
 - 4) Dönüş değerini alacak olan ifade (eğer varsa, alt yordamın fonksiyon olması durumunda)

Parameter Passing

- Genel amaçlı işlemci registerlarının çağırın program ve alt yordam tarafından ayrı ayrı kullanılabileceğini dikkate almak gerekir.
 - Bu, alt yordamı çağırmadan önce içeriklerinin korunması gerektiği anlamına gelir.

Stack Frame

- **stack frame**

- Alt yordam için yığın içerisinde ayrılan özel bir çalışma alanıdır. Alt yordama girildiği esnada oluşturulur ve kontrol alt yordamdan çağırılan programa devredildiği esnada serbest bırakılır.
- Bellek alanından tasarruf etmek ve alt yordam tarafından kullanılan verilere erişimi kolaylaştırmak için, alt yordam tarafından kullanılan yerel bellek değişkenleri de **stack frame** içine yerleştirilebilir.
- **frame pointer (FP)** alt yordama iletilen parametrelere ve alt yordam tarafından kullanılan yerel bellek değişkenlerine erişim için kullanılan bir işaretçi registeridir.

Problem-Example 1

- Bir alt yordam için bellekte yer tahsis etmek istediğimizi varsayalım.
 - Bu alt yordam **R0**, **R1** genel amaçlı registerları ve çağıran programdan iletilecek **PAR1**, **PAR2**, **PAR3**, ve **PAR4** parametrelerini kullanacak olsun.
- Bu durumda yığının yapısı?

Problem-Example 1

Stack

- PAR1
- PAR2
- PAR3
- PAR4
- (R0)
- (R1)
- RETURN ADDRESS

Comment

- Parameter PAR1
- Parameter PAR2
- Parameter PAR3
- Parameter PAR4
- The old contents of R0
- The old contents of R1
- Address for return to the calling program

Problem-Example 2

- Diyelim ki bellekte iki alt yordam için yer ayırmamız gerekiyor: SUB1 ve SUB2.
 - SUB1 **R0** registerini ve çağırان programdan iletilen **A** parametresini kullanacak.
 - SUB2, SUB1 yordamından çağrılacak ve **R0**, **R1** registerlerini ve SUB1 den iletilen **B,C,D** parametrelerini kullanacak.
- Bu durumda yığının yapısı?

Problem-Example 2

Stack

- D
- C
- B
- (R0)
- (R1)
- SUB 2 RETURN ADDR.
- A
- (R0)
- SUB 1 RETURN ADDR.

Comment

- Parameter D
- Parameter C
- Parameter B
- R0 from SUB1
- The old contents of R1
- Return address for SUB2
- Parameter A
- The old contents of R0
- Return address for SUB1

Recursive function (subroutine) and stack

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

FIGURE 2-2 Recursive Factorial Algorithm Definition

Factorial (3): Decomposition and solution

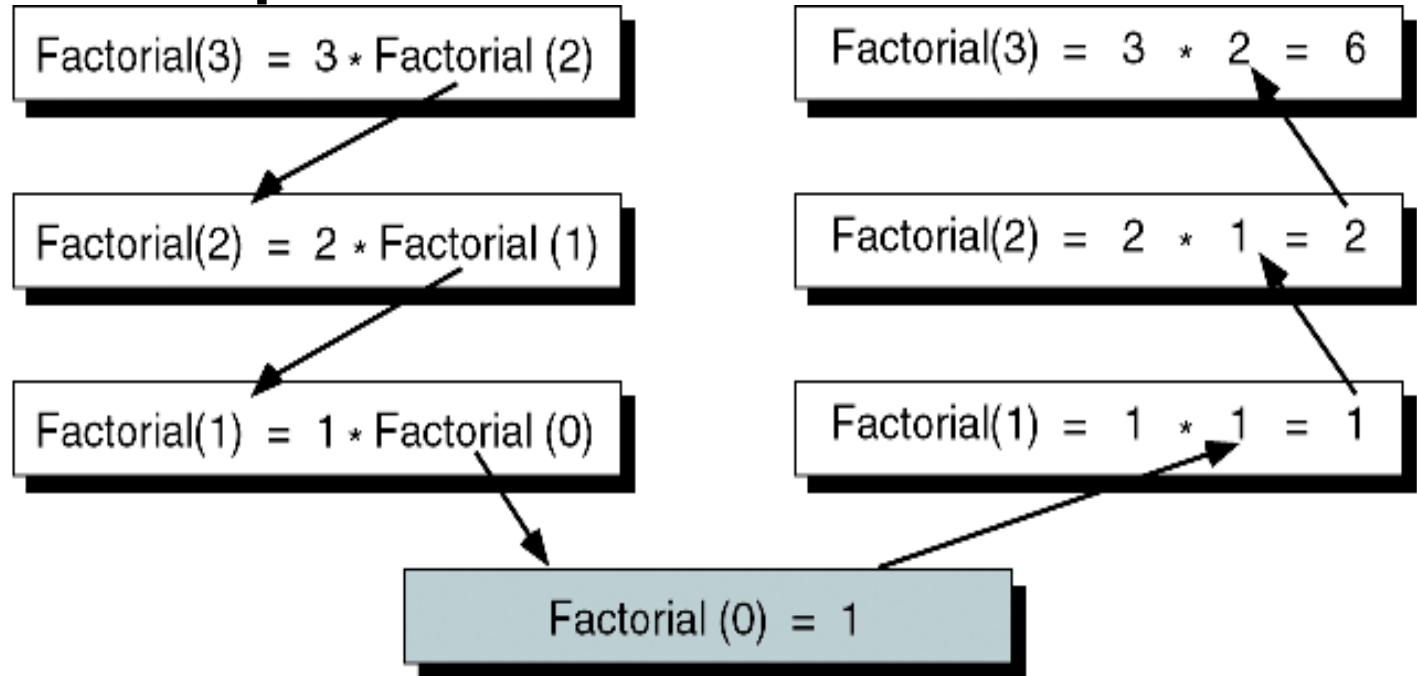


FIGURE 2-3 Factorial (3) Recursively

ALGORITHM 2-2 Recursive Factorial

```
Algorithm recursiveFactorial (n)
```

```
Calculates factorial of a number using recursion.
```

```
Pre    n is the number being raised factorially
```

```
Post   n! is returned
```

```
1 if (n equals 0)
```

```
1 return 1
```

```
2 else
```

```
1 return (n * recursiveFactorial (n - 1))
```

```
3 end if
```

```
end recursiveFactorial
```

Ödev 4

Normal parantez ve süslü parantez karakterlerini kabul eden bir program yazın. Karakter çiftlerinin eşleşip eşleşmediğini belirlemek için bir yığın kullanın. Bu doğrultuda aşağıdaki fonksiyonun gövdesini tamamlayın.

```
bool balanced(const char p[ ], size_t n)
```

```
// Precondition: p[0]...p[n-1] her biri '(', ')', '{' veya '}' olan n adet karakter içerir
```

```
// Postcondition: Karakterler, her biri '(' ile eşleşen ')' ve '{' ile eşleşen
```

```
// '}' ile doğru şekilde dengelenmiş parantezler dizisi oluşturursa,
```

```
// fonksiyon true değerini döndürür.
```

```
// ( { } ) Gibi bir dizinin dengeli(eşleştirilmiş) olmadığına dikkat edin, çünkü
```

```
// parantezleri eşleriyle eşleştirmek için çizgiler çizdiğimizizde, çizgiler
```

```
// birbiriyle kesişir geçer. Diğer taraftan, ( { } ) ve { ( ) } dengesi dengelidir.
```