VERİ YAPILARI DERS NOTLARI

BÖLÜM 4 – STACK (YIĞIN, YIĞIT)

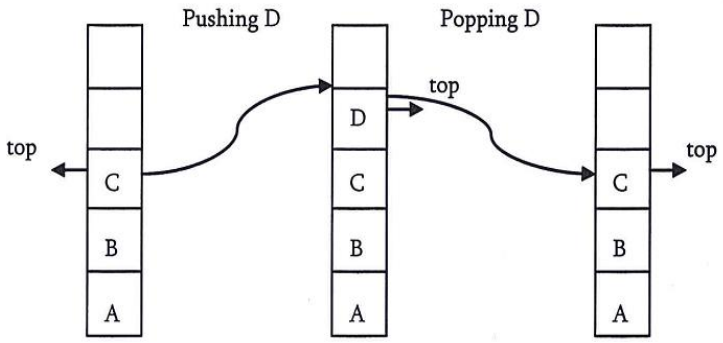
Yard. Doç. Dr. Deniz KILINÇ

CELAL BAYAR ÜNİVERSİTESİ, YAZILIM MÜHENDİSLİĞİ

2015-2016

1. Tanım

Stack, doğrusal artan bir veri yapısı olup; insert (push) ve delete (pop) işlemleri, listenin sadece "top" adı verilen bir ucunda yani stack'in en üstünden gerçekleştirilir. Bu nedenle *stack* Son Giren İlk Çıkar (*Last In First Out* - **LIFO**) mantığı ile işleyen bir veri yapısıdır. Aşağıdaki şekilde bir stack VY için push-pop işlemleri gösterilmiştir.



Şekil. Stack VY için push, pop işlemleri

Boş bir stack'ta *pop* işlemi ***underflow*** ve tamamen dolu bir stack'a *push* işlemi ***overflow*** hatalarını (exceptionlar) üretir. Her iki durum, Stack ADT tanımlamalarında ve stack implementasyonlarında dikkate alınmalıdır.

2. Stack Kullanım Alanları

Stack çok basit bir VY olmasına karşın birçok kullanım alanı bulunmaktadır. Aşağıda farklı kaynaklardan derlenen kullanım alanlarına örnekler verilmiştir:

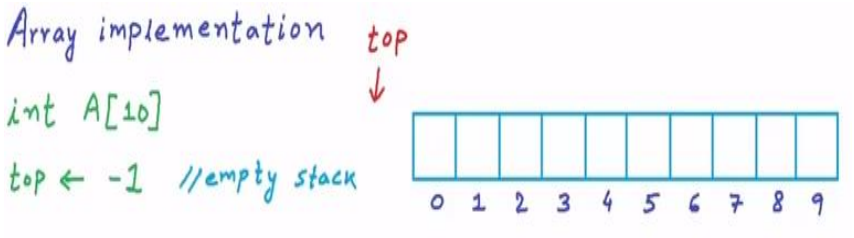
- Word, Excel, Photoshop gibi yazılımlarda yapılan işlemlerin sırayla kayıt edildiği ve geri alınabilecek şekilde tutulduğu *undo fonksiyonu* bir stack uygulamasıdır.
- Bir web tarayıcıda ileri-geri adres gezmek için stack yapısı kullanır.
- C# veya Java gibi programlama dillerinde açılan parantezin doğru kapatılması kontrolünde ("Matching Bracket" – "Parantez Eşleştirme" kontrolü) kullanılır.
- Polish Notasyon: Infix olarak bilinen A*(B+C/D)-E cebirsel gösteriminin yerine hesap makinelerinde kullanılan postfix ABCD/+*E notasyonuna çevirme işleminde stack kullanır.
- HTML-XML'de tag'lerin eşleştirilmesi bir stack uygulamasıdır.
- Stack'ların bir diğer uygulama alanı labirent türü problemlerin çözümünde backtracking (bir yola gir yol tıkanırsa en son ki yol ayrımına geri gel başka yola devam et!) yöntemiyle kullanılır.
    - Yol bilgisi bir stack yapısına push edilir yol yanlışsa son gidilen yanlış nokta pop edilir önceki noktayla devam edilir.
- Java derleyicisi program kodunun tamamını postfix'e çevirirken stack kullanır.
- Java Virtual Machine (JVM) byte code'ları execute ederken altyapısında yine stack kullanır.
- Recursion ve function call işlemlerinin Bellekte gerçekleştirilmesinde stack kullanılır.

3. Stack İmplementasyonu

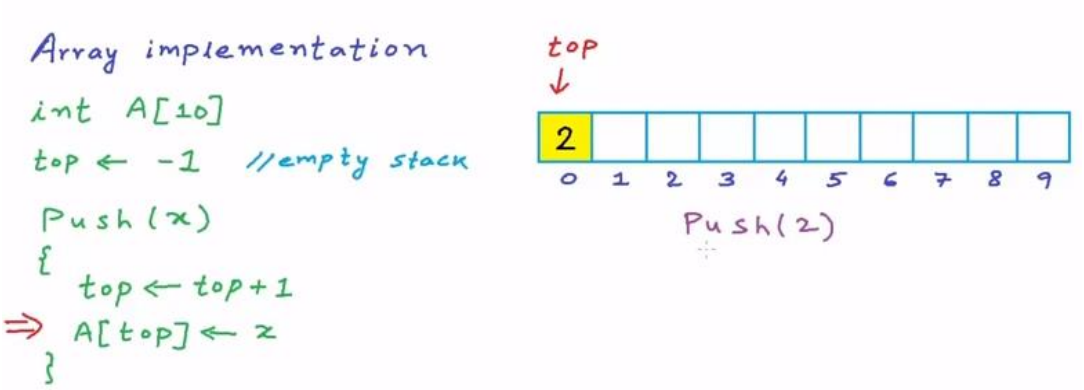
Stack'lar bağlı liste veya diziler kullanarak implemente edilebilirler. Ancak implementasyon için en basit yöntem *tek boyutlu bir dizi* tanımlamaktır.

3.1. Stack Dizi İmplementasyonu

i) `n = 10` uzunluğunda bir dizi ve stack'ın en üstünü tutan *top* değişkeni oluşturulur ve boş stack için değişkenin ilk değeri "-1" olarak atanır.

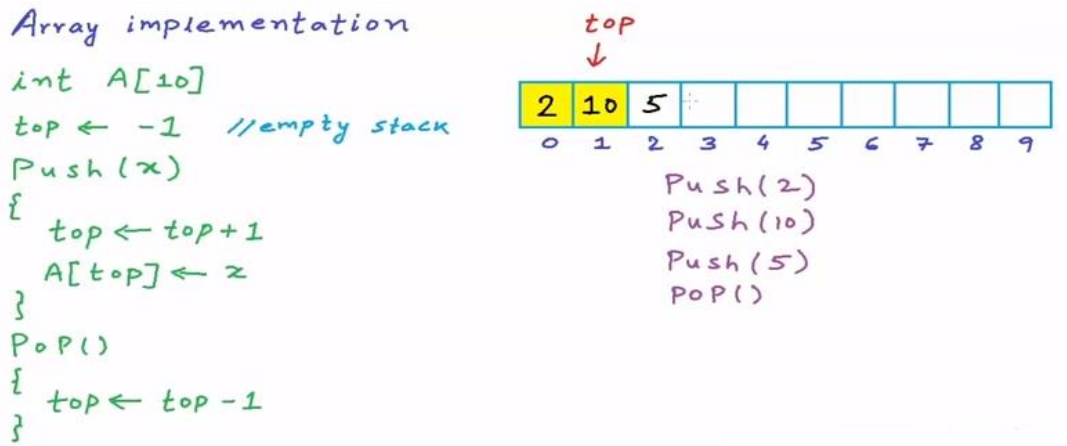


ii) Push ile stack'e eleman eklenir. Örneğin 2 sayısını stack'a yazalım:



`push(2)` metodu çalıştığında, 2 değeri (top=top+1) indeksine yazılır.

iii) Pop ile en üst eleman (burada 5) stack'tan ayıralım:



Bunun için pop() metodu top=top-1 ile çalıştırılır. Burada, top değişkeni artık 5'e işaret etmediği için 5'i silmeden de çalışmaya devam edebiliriz. Çünkü sonraki adımda yeni bir değer push ederken o hücreye yeni değer girileceğinden, yeni değeri 5'in üzerine yazarız.

iv) Peek ile en üst elemanın değerini okuyalım:

```
Peek ()
{
	return A[top]
}
```

v) Stack boş mu kontrolü yapalım:

```
IsEmpty()
{
	if top==-1
		return true
	else
		return false
}
```

Stack ADT ve Dizi İle Stack Oluşturma

Stack için kullanılacak metotlar ve özellikler; Push(), Pop(), Peek(), IsEmpty() ve Top() olmalıdır. Bunun için tanımlanan C# interface'i ve ArrayTypedStack sınıfının imlementasyonu aşağıda verilmiştir.

```
public interface IStack
{
    void Push(object item);
    object Pop();
    object Peek();
    bool IsEmpty();
    int Top { get; set; }
}

public class ArrayTypedStack: IStack
{
        private object[] items;
        private int top = -1;

        public ArrayTypedStack(int itemCount)
        {
            this.items = new object[itemCount];
        }
        public void Push(object item)
        {
            if (items.Length == Top + 1)
            {
                throw new Exception("Hata: Stack doldu... (Overflow)");
            }
            items[++Top] = item;
        }

        public object Pop()
        {
            if (IsEmpty())
            {
                throw new Exception("Hata: Stack boş...(Downflow)");
            }
            Object temp = items[Top--];
            return temp;
        }
```

```
        public object Peek()
        {
            return items[Top];
        }

        public bool IsEmpty()
        {
            return Top == -1 ? true : false;
        }

        public int Top
        {
            get
            {
                return top;
            }
            set
            {
                top = value;
            }
        }
}
```

Günlük Hayatta Kullanılan Bazı Stack Örnekleri

Örnek 1: Stack Kullanarak String'i Tersten Yazma (Reverse)

`ArrayTypedStack` veri yapısını kullanarak verilen bir string ifadeyi ters çevirecek algoritmayı geliştirelim.

Çözüm:

- String ifadedeki her bir karakter stack'e Push metodu ile aktarılır.
- Stack'deki her bir karakter Pop'ile stack'den okunur.

```
string test = txtOriginal.Text;

char[] chrArr = test.ToCharArray();
ArrayTypedStack ats = new ArrayTypedStack(chrArr.Length);

for (int i = 0; i < chrArr.Length; i++)
{
       ats.Push(chrArr[i]);
}

test = "";
for (int i = 0; i < chrArr.Length; i++)
{
       test += ats.Pop().ToString();
}
txtReverse.Text = test;
```

Örnek 2: Parantez Eşleştirme Kontrolü (Matching Bracket Kontrol)

İç içe parantezler içeren bir ifadede parantezlerin geçerli olması için 2 kural işletilebilir:

1. Açılan ve kapanan toplam parantez sayısı eşit olmalıdır. Aç"(" ve kapa ")" parantezlerin eşitliğine bakılır.

2. Kapanan her parantezden önce bir parantez açılmış olmalıdır. Her “)”için bir “(“olup olmadığına bakılır.

Örneğin:
- “((A+B)” ve “A+B(“ →1. şarta uymaz.
- “)A+B(-C” ve “(A+B))-(C+D” →2. şarta uymaz.

Problemin çözümü için her açılan parantezde bir **sayaç** arttırılır ve kapanan parantezde de sayaç **azaltılır**. İfadenin herhangi bir noktasındaki *Nesting Depth* (parantez derinliği) o ana kadar açılmış fakat kapanmamış parantezlerin sayısıdır. Parantezleri geçerli bir ifadede aşağıdaki şartları içermelidir:

1. İfadenin sonunda sayaç 0 olmalıdır. İfadede ya hiç parantez yoktur veya açılan parantezlerin sayısı ile kapanan parantezlerin sayısı eşittir.
2. İfadenin hiçbir noktasında parantez sayısı negatif olmamalıdır. Bu kontrol parantez açılmadan bir parantezin kapanmadığını garantiler.

Aşağıdaki ifade iki şarta da uyar.
7 - ((x * ((x + y) / (j – 3)) + y) / (4 – 2.5))
0 0 1 2 2 2 3 4444 3 3 4 444 3 2 22 1 1 2 2 2 2 1 0

Aşağıdaki ifade 1. şarta uymaz. İfade sonunda 1 parantez arttı, sayaç sıfırdan büyük oldu.
((A + B)
1 2 2 2 2 1

Aşağıdaki ifade 2. şarta uymaz. Çünkü ara durumda sayaç negatif oldu.
(A + B)) - (C + D
1 1 1 1 0 -1 -1 0 0 0 0

Parantez eşleştirme problemini çözmek için stack kullanılabilir. Algoritma aşağıdaki gibidir:
- Bir açılış parantezi ile karşılaşıldığında (“(”, “{”, “[”) stack’e Push edilir.
- İlgili parantezlerin karşılığı olan parantez ile karşılaşıldığında (“)”, “}”, “]”) stack’e bakılır, *stack boş değilse* stack’ten bir eleman Pop edilerek, doğru karşılık olup olmadığı kontrol edilir.
    - Doğruysa işlem sürdürülür.
    - Doğru değilse ifade geçersizdir.
- Stack sonuna ulaşıldığında stack boş olmalıdır. Aksi halde açılmış ama kapanmamış parantez olabilir.

`ArrayTypedStack` veri yapısını kullanarak problemi çözen algoritmanın implementasyonunu yapalım.

Çözüm:

```
public bool IsExpressionValid(string expr)
{
            char[] chrExpr = expr.ToCharArray();
            char chRead, chPopped;
            bool isValid = true;

            ArrayTypedStack ats = new ArrayTypedStack(chrExpr.Length);
            for (int i = 0; i < chrExpr.Length; i++)
```

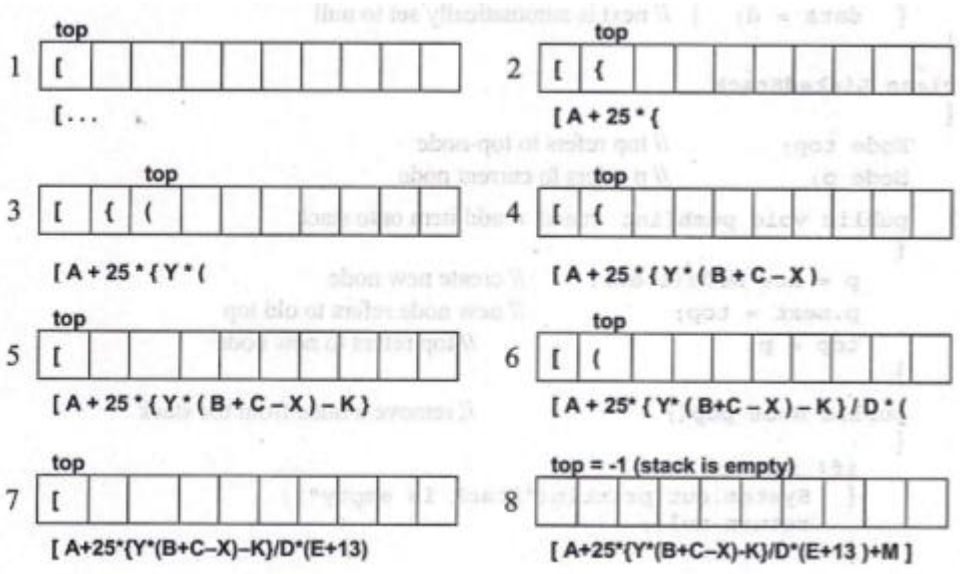
```
        {
            chRead = chrExpr[i];
            //Sadece açılış parantezlerini stacke ekliyoruz
            if (chRead == '[' || chRead == '(' || chRead == '{')
            {
                ats.Push(chRead); //Stack' ekle
            }

            //Kapanış Parantezi
            if (chRead == ']' || chRead == ')' || chRead == '}')
            {
                //Kapanış parantezi olduğu halde,
                //stackte açılış parantezi kalmadıysa
                if (ats.IsEmpty())
                    isValid = false;
                else
                {
                    //Stackten aldığım açılış parantezi ile
                    //okuduğum kapanış parantezi tutarlı mı?
                    chPopped = (char)ats.Pop();

                    //Stackten aldığım açılış
                    if (chPopped == '[' && chRead == ']') isValid = true;
                    if (chPopped == '{' && chRead == '}') isValid = true;
                    if (chPopped == '(' && chRead == ')') isValid = true;
                }
            }
        }
        //Döngü sonunda stack artık boş olmalı.
        if (!(ats.IsEmpty()))
            isValid = false;
        return isValid;
}
```

Test: [A+25*{Y*(B+C-X)-K}/D*(E+13)+M]

Algoritma



Örnek 3: Polish Notasyon

Polish notasyonu Bilgisayar Bilimleri alanındaki önemli konulardan bir tanesidir. Operatörleri, operandlardan önce veya sonra gösterme metodu olarak tanımlanabilir.

- **Infix:** Bildiğimiz klasik gösterim.
- **Prefix:** Operatörler operandlardan önce yazılır.
- **Postfix:** Operatörler operandlardan sonra yazılır.

Örnek: A+B

- Operatör (işlemci) : +
- Operand (işlenenler) A, B
- Infix gösterim: A+B
- Prefix Gösterim: +AB (benzeri bir gösterim add(A,B) fonksiyonu)
- Postfix Gösterim: AB+

Infix	**Postfix**
A+B-C	AB+C-
(A+B)*(C-D)	AB+CD-*
A^B*C-D+E/F/(G+H)	AB^C*D-EF/GH+/+

- Postfix formda parantez kullanımına gerek yoktur.
- Infix formdan Postfix forma çevrilen bir ifadede operand'ların bağlı olduğu operator'leri (+,-,*,/) görmek zorlaşır (3 4 5 * + ifadesinin sonucunun 23'e, 3 4 + 5 * ifadesinin sonucunun 35'e karşılık geldiğini bulmak Infix gösterime alışık olduğumuz için zor gibi görünür). Fakat parantez kullanmadan tek anlama gelen bir hale dönüşür. İşlemleri, hesaplamaları yapmak kolaylaşır.
- Birçok derleyici 3*2+5*6 gibi bir Infix ifadenin değerini hesaplayacağı zaman Postfix forma dönüştürdükten (belirsizliği ortadan kaldırdıktan sonra) sonucu hesaplar : "3 2 * 5 6 * +"
- Hem Infix→Postfix dönüşümünde hem de Postfix hesaplamasında stack kullanılabilir.

Örnek 4: Java Compiler ve Java Virtual Machine Stack Kullanımı

Orijinal Java kodu

```
...
x = x + 1;
if ( x > 2 )
   { y = 2 * ( x - 3 ); }
...
```

Adım 1: Java kodu, Java compiler tarafından stack veri yapısı kullanılarak Postfix forma çevrilir.

```
...
x 1 + =x ;
x 2 > if
    2 x 3 - * =y ;
...
```

Adım 2: Java compiler Postfix formatından yola çıkarak bytecode'u ".class" dosyasına yazar.

```
...
load x
loadconst 1
add
storeinto x
load x
loadconst 2
greaterthan
test_and_jump_if_false_to LabelA
loadconst 2
load x
loadconst 3
subtract
multiply
storeinto y
LabelA:
...
```

Adım 3: Java Virtual Machine class dosyasında her bir instruction'ı yine stack yapısı kullanarak execute eder.

4. Stack İşlem Karmaşıklığı

Dizi ile tanımlanan stack'ta boyut önceden belirli olmalıdır. Bu bağlamda stack'taki eleman sayısı n olarak kabul edilirse bir stack için işlem karmaşıklıkları Tablo 1'deki gibi olur.

Space Complexity (for n push operations)	$O(n)$
Time Complexity of push()	O(1)
Time Complexity of pop()	O(1)
Time Complexity of size()	O(1)
Time Complexity of isEmpty()	O(1)
Time Complexity of isFullStack()	O(1)
Time Complexity of deleteStack()	O(1)

Tablo 1 Dizi ile Implemente Edilen Stack İşlem Karmaşıklıkları

Bağlı liste ile implemente edilen stack yapısında da karmaşıklık aynıdır.