DFS Gezinme Algoritması

DFS Algoritmasına göre graflar üzerinde gezinme işlemi gerçekleştirmek için yığıt yapısından yararlanılır. Yığıt yapısını diziler ile gerçekleştirmek daha pratik olduğundan dizi ile yığıt oluşturulmuştur. Diziler ile yığıt modellerken yığıt işaretçisi kullanılmalıdır. Yığıt işaretçisi yığıttaki son elemanın yerini vermektedir. Aşağıda max 10 elemanlı bir yığıt için gereken değişkenler verilmiştir.

```
public class Yigit {
    int [] yigit = new int[10];
    int isaretci = -1;
```

Yığıta ekleme işleminde önce yığıtın dolu olup olmama durumu kontrol edilir. Yığıt dolu ise ekleme işlemi gerçekleştirmez. Yığıtta yer varsa yığıt işaretçisi bir artırılarak eklenecek eleman buraya eklenir. push işlemi için gereken kod aşağıda verilmiştir.

```
void push(int eleman) {
    if(isaretci < 9) {
        yigit[++isaretci] = eleman;
    }else {
        System.out.println("Yığıt Dolu");
    }
}
```

Yığıttan çıkarma yapılırken ilk olarak yığıtın boş olma durumu kontrol edilir. Yığıt boş ise geçersiz bir değer döndürülür. Yığıtta eleman varsa yığıt işaretçisinin bulunduğu yerdeki eleman döndürülür ve yığıt işaretçisi 1 azaltılır. Pop işlemi için gereken kod aşağıda verilmiştir.

```
int pop() {
    if(isaretci >= 0) {
        return yigit[isaretci--];
    }else {
        System.out.println("Yığıt boş");
        return -1;
    }
}
```

DFS'de yığıt dolu olduğu sürece döngü devam ettirilir. Bunun için yığıtın boş olup olmadığını bulan fonksiyonun yazılması gerekir. Bu fonksiyon aşağıda verilmiştir.

```
boolean bosMu() {
    if(isaretci>=0)
        return false;
    else
        return true;
}
```

Yazmış olduğumuz yığıt kodunun doğruluğunu test etmek için yazılan kodlar aşağıda verilmiştir.

```
void listele() {
    for (int i = 0; i <= isaretci; i++) {
        System.out.print(yigit[i] + " ");
    }
}
```

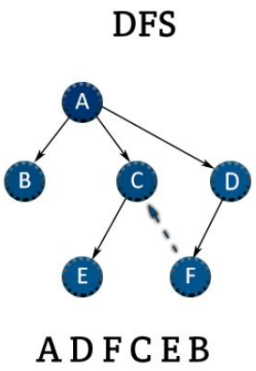
```
public static void main(String[] args) {
    Yigit y = new Yigit();
    y.pop();
    y.push(5);
    y.push(10);
    y.listele();

    System.out.println("---"+ y.pop());
    y.listele();
}
```

DFS Algoritması için öncelikle elimizde bir graf yapısının olması gerekmektedir. Komşuluk matrisi yöntemi ile bu graf yapısı oluşturulmuştur. Oluşturulan komşuluk matrisi ve bunun graf olarak çıktısı aşağıda verilmiştir. Matris 0 = A'yı 5=F'yi temsil etmektedir.

```
int [][] komsuluk = new int[6][6];
int [] ziyaret = new int[6];
Yigit y = new Yigit();

DFSProje() {
    komsuluk[0][1] = 1; // A->B
    komsuluk[0][2] = 1; // A->C
    komsuluk[0][3] = 1; // A->D
    komsuluk[2][4] = 1; // C->E
    komsuluk[3][5] = 1; // D->F
    komsuluk[5][2] = 1; // F->C
}
```



DFS gezinmesinde yığıt yapısı kullanılır. Yığıt boş olmadığı sürece ve o anki düğümün komşuları olduğu sürece döngü devam ettirilir. Yeni gelen komşular yığıta eklenir. Mevcut düğümün yeni komşusu yoksa yığıttan eleman çekilir. Yığıttan çekilen düğümün daha önce ziyaret etmediği komşuları aranır. Döngü yeni komşu bulunmadığında ve yığıt boş olduğunda sonlanır. DFS kodu aşağıda verilmiştir. Bu kod için komşuların bulunması gerekir. Komşu bulunurken düğümün daha önce ziyaret edilip edilmediği göz önüne alınmıştır. Komşu bulma algoritması da aşağıda verilmiştir.

```
void DFSgez(int dugum) {
    int v = dugum;
    System.out.println(v);
    ziyaret[v] = 1;

    while(komsuBul(v) != -1 || !y.bosMu()) {
        if(komsuBul(v) != -1) {
            y.push(v);
            v = komsuBul(v);
            ziyaret[v] = 1;
            System.out.println(v);
        }else {
            v = y.pop();
        }
    }
}
```

```
int komsuBul(int dugum) {
    for (int i = 5; i >= 0; i--) {
        if(komsuluk[dugum][i] != 0 && ziyaret[i] == 0) {
            return i;
        }
    }
    return -1;
}
```

Yazılan kodun doğruluğunu test etmek için aşağıdaki kod parçacığı kullanılmıştır.

```
public static void main(String[] args) {
    DFSProje x = new DFSProje();

    x.DFSgez(0);
}
```