VERİ YAPILARI VE ALGORİTMALAR

3.Ödev Raporu

Okutman: M.Elif Karslıgil

Konu : Graf İşlemleri

Ozan Türker Demir

NO:15011611

Github.com/turkerozan/DataStructuresH3

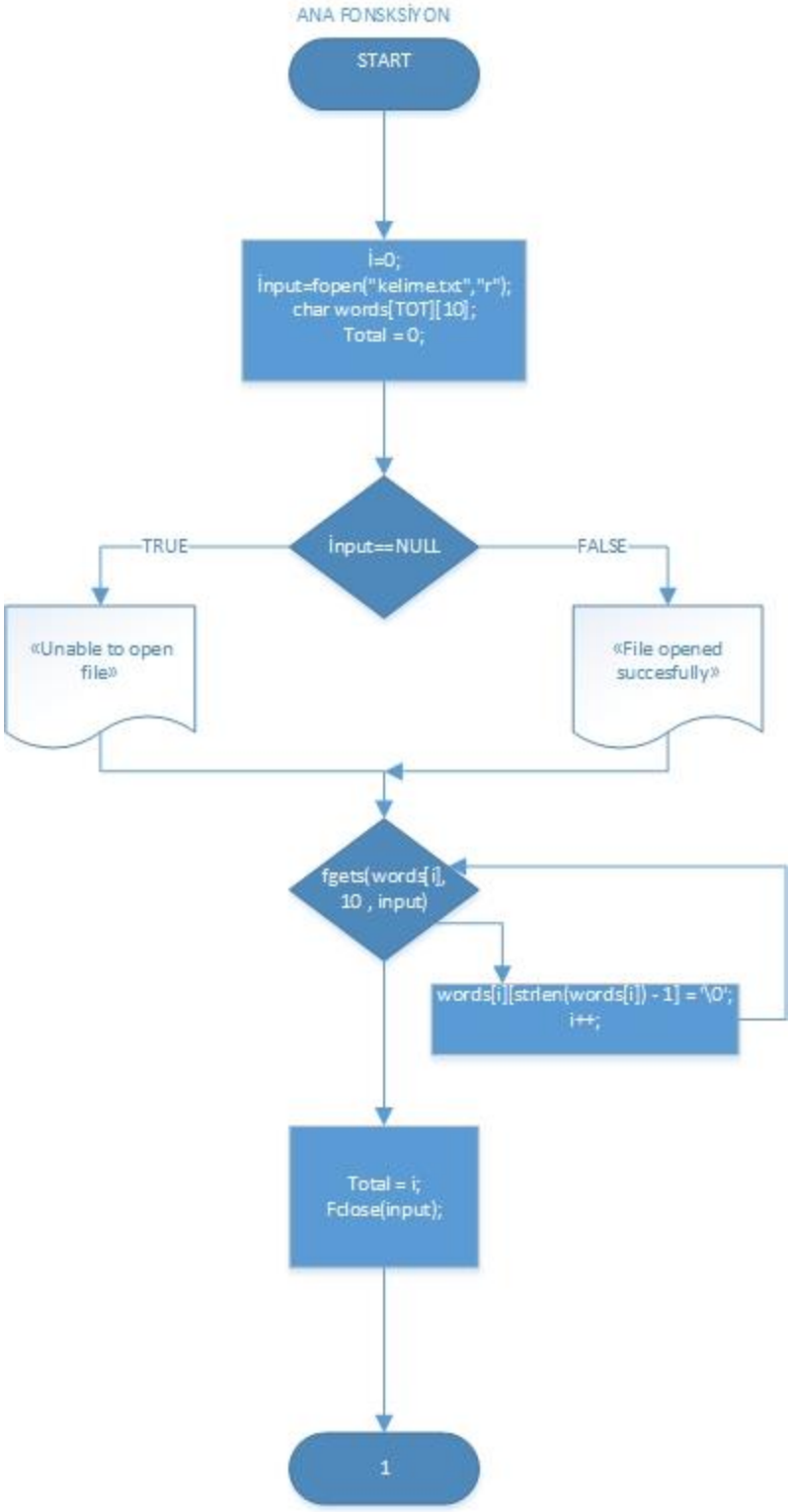
1. YÖNTEM

Raporun bu bölümünde sırasıyla problemler verilip, ardından akış diyagramları gösterilecektir. Öncelikle problem tanımı verilen şekli ile;

Problem: Bu ödevde verilen iki kelime için, **her adımda sadece 1 harfi değiştirerek** 1. kelimenin, 2. kelimeye dönüşüp dönüşmediğini, dönüşüyorsa arada hangi kelimelerden geçildiğini bulan bir kelime oyunu yazılacaktır. Aşağıdaki örnek, **prove** kelimesinin **guess** kelimesine dönüşümünü
göstermektedir.
prove → prose → prese → prest → wrest → weest → geest → guest → guess
Verilen bir kelimeden, her adımda bir harf değiştirerek bir başka kelimeye ulaşmak için **graf veri yapısı** kullanılacaktır. Grafın **düğümlerini kelimeler** oluşturmalıdır. Eğer bir kelimenin **sadece 1 harfini değiştirerek** diğer kelime elde ediliyorsa iki kelime arasında **bağlantı** vardır. Örneğin yukarıdaki örnekte **prove** ve **prose** kelimeleri arasında bağlantı vardır. Fakat **prove** ve **wrest** kelimeleri arasında bağlantı yoktur.

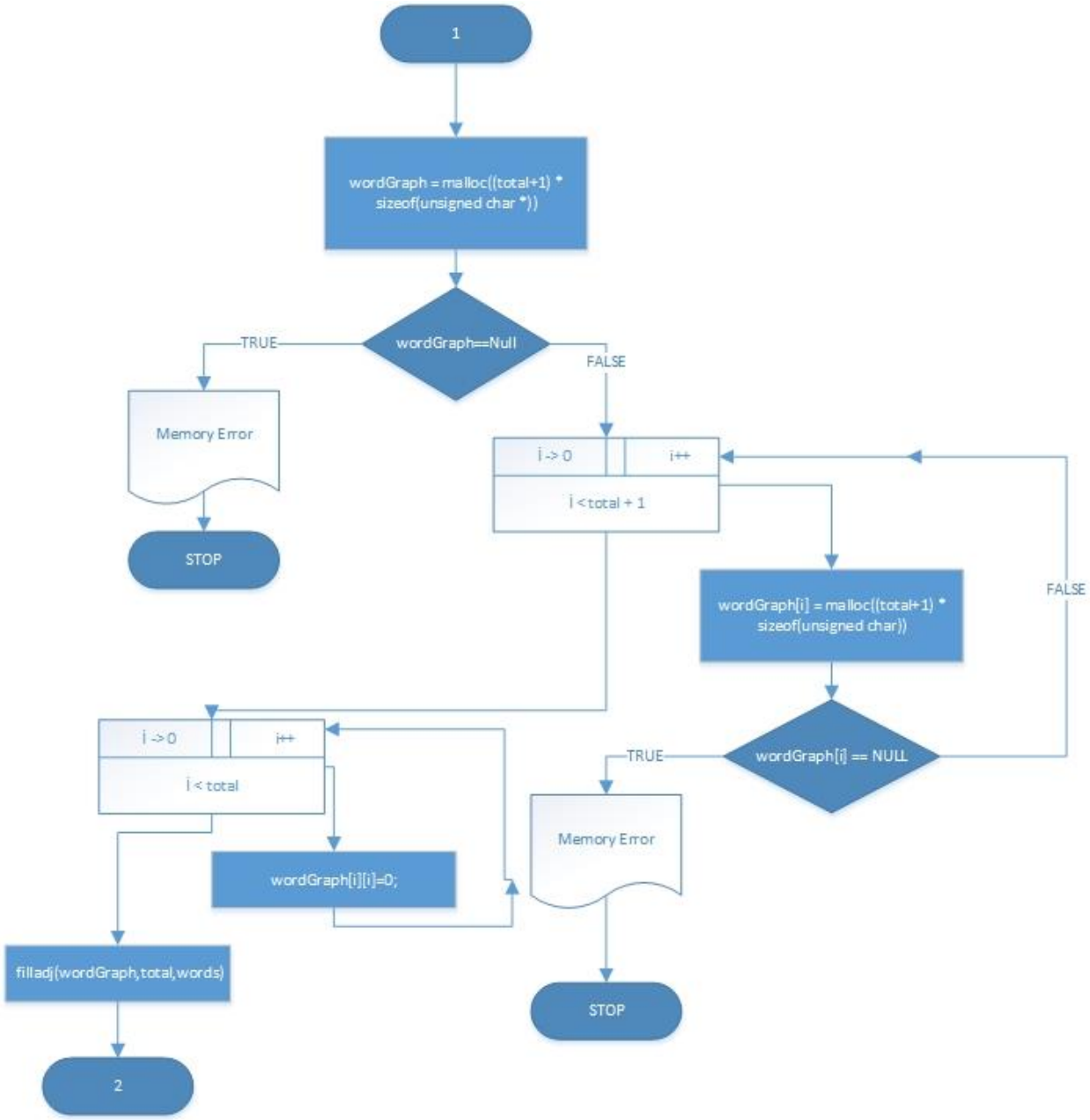
Bu sekmeye kadar baktığımızda, bizden istenilen, ilk olarak bir kelime listesinden okuma yapmak, ardından bir komşuluk matrisi oluşturmak, ve bu komşuluk matrisini doldurmak. Doldurulan matrisi kullarak gerekli işlemleri gerçekleştirip sonuca ulaşmak. Yukarıda verilen tanımda, kelimeler arasındaki bağlantı için graf veri yapısı kullanılması gerektiği açıkca bahsedilmiştir.

Aşağıda dosya okuma işleminin akış diyagramı verilmiştir. Dosya önce read modunda açılmakta, ardından açılıp açılmadığı kontrol edilmektedir. Buna uygun olarak *words* dizisine *i* iteratorü kullanılarak, boyutu 10 olan bufferlar ile atama yapılmaktadır. Her atamada *i* arttırılmakta ve kelimenin son elemanı yeni satır elemanı yerine, null karakteri ile değiştirilmektedir.



Dosya okuma akış diyagramı

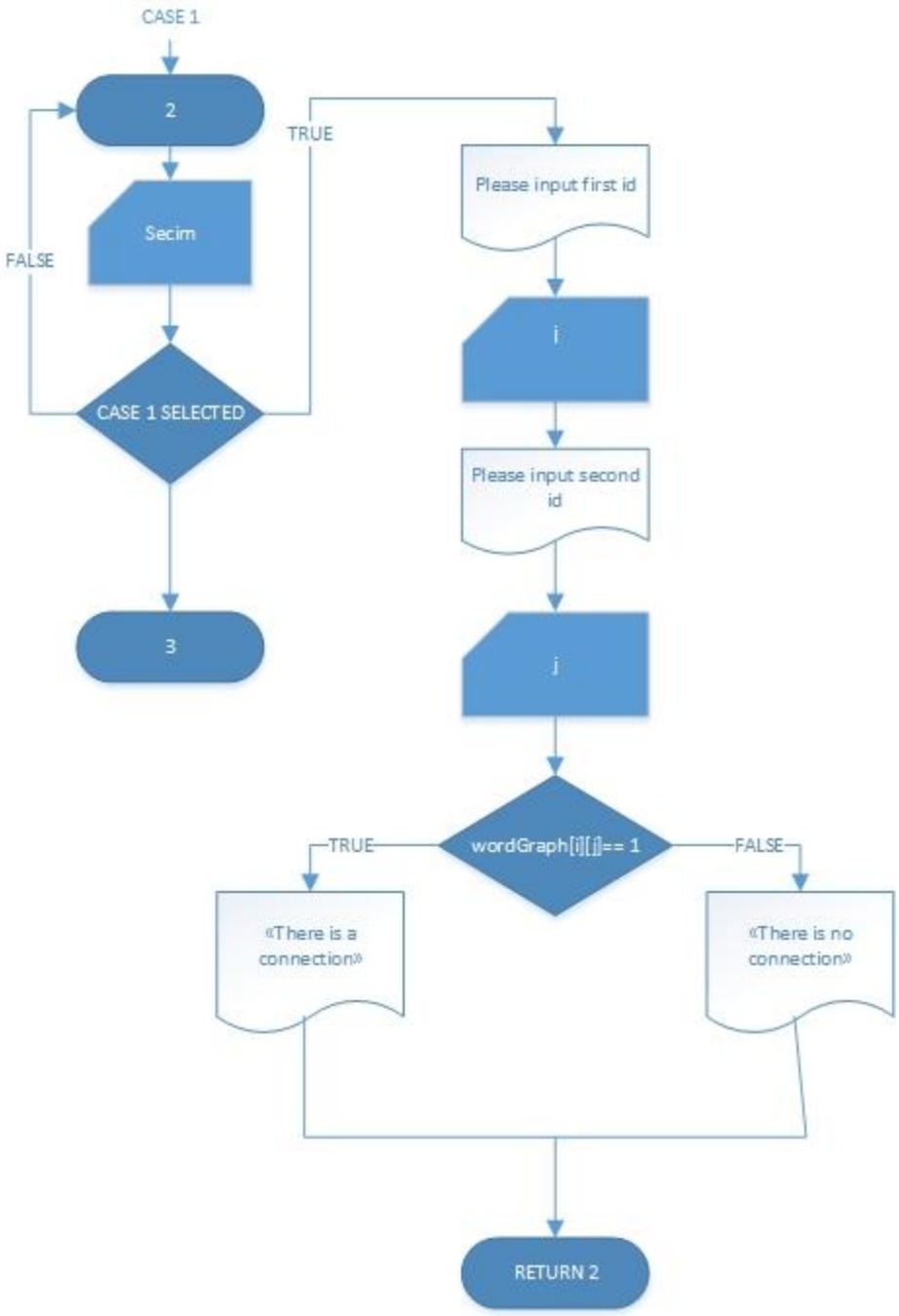
Dosyalar okunduktan sonra, komşuluk matrisi için bir graph oluşturulacaktır. Bir önceki dosya okuma listesinde toplam kelime sayısı alındıktan sonra dinamik hafıza yönetimi ile NxN kare matris oluşturulmuştur. Ardından matrisin bütün elemanları başlangıç değeri olarak '0' lanmıştır. Burada dikkat edilmesi gereken *malloc* fonskyonu çağırılırken *total+1* kadar yer ayırılmasıdır. Çünkü iteratörümüz 0 dan başlamıştır. Bu nedenle aslında ihtiyacımız olan *wordGraph* 2 boyutlu dizisinin toplam boyutu (*total+1)x(total+1) olmalıdır.* Matrisimiz 0 ile doldurulduktan sonra olması gereken değerler ile doldurmak üzere *filladj* fonskyonu çağırılmıştır. Aşağıda akış şemasını bulabilirsiniz.Fonskyonların akış şeması ayrı olarak verilecektir.



Bu adıma kadar yapılan her şey ön işlemdir. Yani programımızın çalışması için kesin gerekli olan, kullanıcıdan bağımsız, lineer bir doğrultuda giden adımlardan oluşmaktadır. Bundan sonra menü işlemleri gelmektedir. Burada menü işlemleri anlaşılabilirliği kolaylaştırmak amacı ile, case 1, case 2, case 3 başlıkları altında ayrı ayrı akış şemaları verilip anlatılacaktır. Ana kod içerisinde case 4, çıkış olarak kullanılmaktadır.

Case 1:

Bu senaryoda, ödev tanımında istenilen, grafı doğru bulduğumuzu ispatlamak için kullanılacak 2 yöntemden ilki gerçeklenmektedir. Bu durumda, kullanıcıdan iki adet kelime *ID* istenilmektedir. Kelime *ID* leri aslında kelimelerin bulunduğu sıra olarak değerlendirilebilir. Böylelikle kelime listemizdeki ilk kelime, *words* arrayinde 0. Elemana, *wordsGraph* matrisinde ise, 0. Kolona denk gelmektedir. Alıcağımız ikinci kelime idsini, *wordsGraph* matrisinden kontrol ederek iki kelimenin komşuluk durumunu sorgulamış oluruz. Bu senaryonun akış diyagramı aşağıda verilmiştir. Burada dikkat edilmesi gereken koşul, eğer case 1 seçilmedi ise tekrar main line a dönüp case 2 ve 3 için kontrolün gerçekleşecek olmasıdır. Bu gerçeklemeler, diğer akış şemalarında gösterilmiştir. Üzerinde durulması gereken bir diğer husus ise seçim bittikten sonra tekrar geri seçim ekranına dönülüyor olmasıdır.

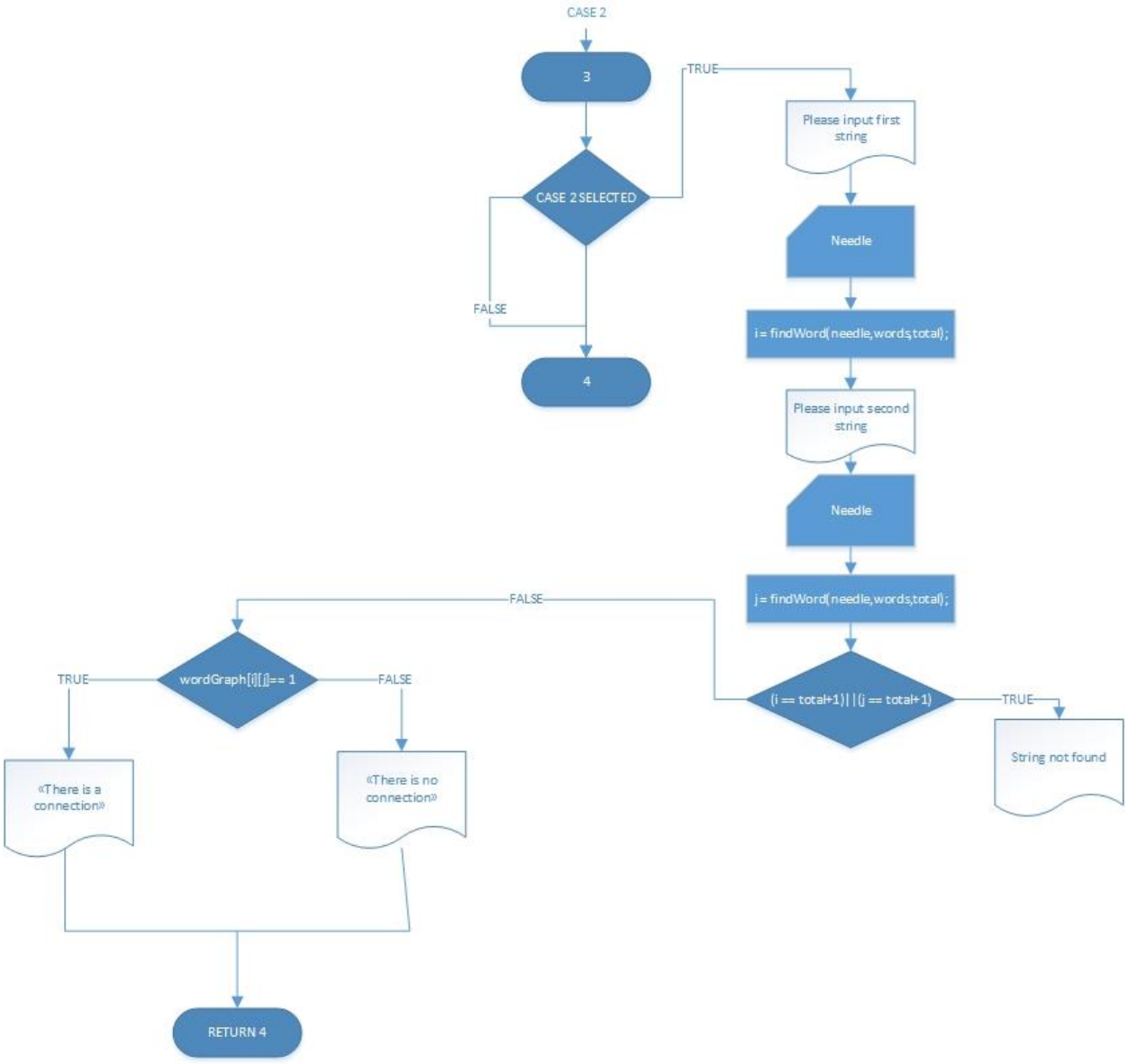


CASE 1 AKIŞ ŞEMASI

Case 2:

Bu senaryoda ise, kullanıcıdan girdi olarak string alınmaktadır. Alınan string önce idsi bulunmakta ardından case 1 deki işlemler aynen yerine getirilmektedir. Burada yine i ve j değişkenleri kullanılmaktadır ve buna ek olarak bu değişkenlere findWord fonskyonundan, stringin posizyonu döndürülmektedir. Fonksyonun içeriği ileride akış şeması ile açıklanacaktır, bu noktada bilinmesi gereken, eğer verilen stringi, kelime sözlüğümüzde bulamadıysa, dönüş adresi olarak *total+1* döndürüyor olmasıdır. Dönen adres değerlerine göre işlemler yapılıp, ekrana çıktı olarak verilmektedir.

Aşağıda akış şeması bulunmaktadır. Eğer dönen i veya j adreslerinden biri *total+1* e eşit ise, demektir ki girilen stringler, kelime listesinde yok. Eğer değilse, case 1 deki gibi kontrol edebiliriz.



CASE 2 AKIŞ ŞEMASI

Case 3 için, verilen iki kelime arasında bağıntı olup olmadığını bulmamız, var ise kaç adımda olduğunu belirlememiz gerekmektedir. Tasarım kararı olarak ayrı bir fonksyon yerine, problemin ana çözümü olduğu için ufak fonksyonlarla birlikte çalışan, ana fonksyon lineerliği üzerinde gidilen bir uygulama tasarımı kullanılmıştır. İlk önce *startstring* ve *stopstring* kullanıcıdan input olarak alınmaktadır. Bunlar tıpkı case 2 deki gibi, önce *i* ve *j* değişkenlerine, *findword* fonskyonu ile verilen kelimelerin idleri atanmıştır. Sonra stringlerin olup olmadığı case 2 deki gibi kontrol edilmiştir. Bu nedenle akış diyagramında, hali hazırda case 2 deki adımların aynısı gerçeklendiği için konulmamıştır. Daha sonra *backqueue* isimli bir değişken ilkendirilmektedir. Bu değişken satır sayısı olarak gelimeleri, sütün sayısı olarak 2 adet, ilki queueye eklenip eklenmediğini kontrol etmek için, ikincisi ise eklendiği sırada hangi queue parenti idi onun idsini tutmak amacı ile kullanılmıştır. Bu nedenle bütün nodların 0. Sütünu 0, 1. Sütunu -1, yani parentsiz olarak

ilklendirilmiştir. Ardından ilk nod, *queue*'ye eklenmiştir. İlk nodeun parenti, -2, kuyruğa eklendiği için kuyruğa eklenme durumu 1 olarak değiştirilmiştir.

```
backqueue[i][0] = 1;//make it visited
backqueue[i][1] = -2;//for controll, we made root node parent id -2, we will use that information if we will
```

-2 sayısı, ileride roota ulaşıp ulaşmadığımızı kontrol etmek için kullanılacaktır. Ardından *stop* isimli bir değişken, queuden çıkan nodun idsini tutması için -1 ile ilklendirilmiştir.

Eğer soruya bakacak olursak;

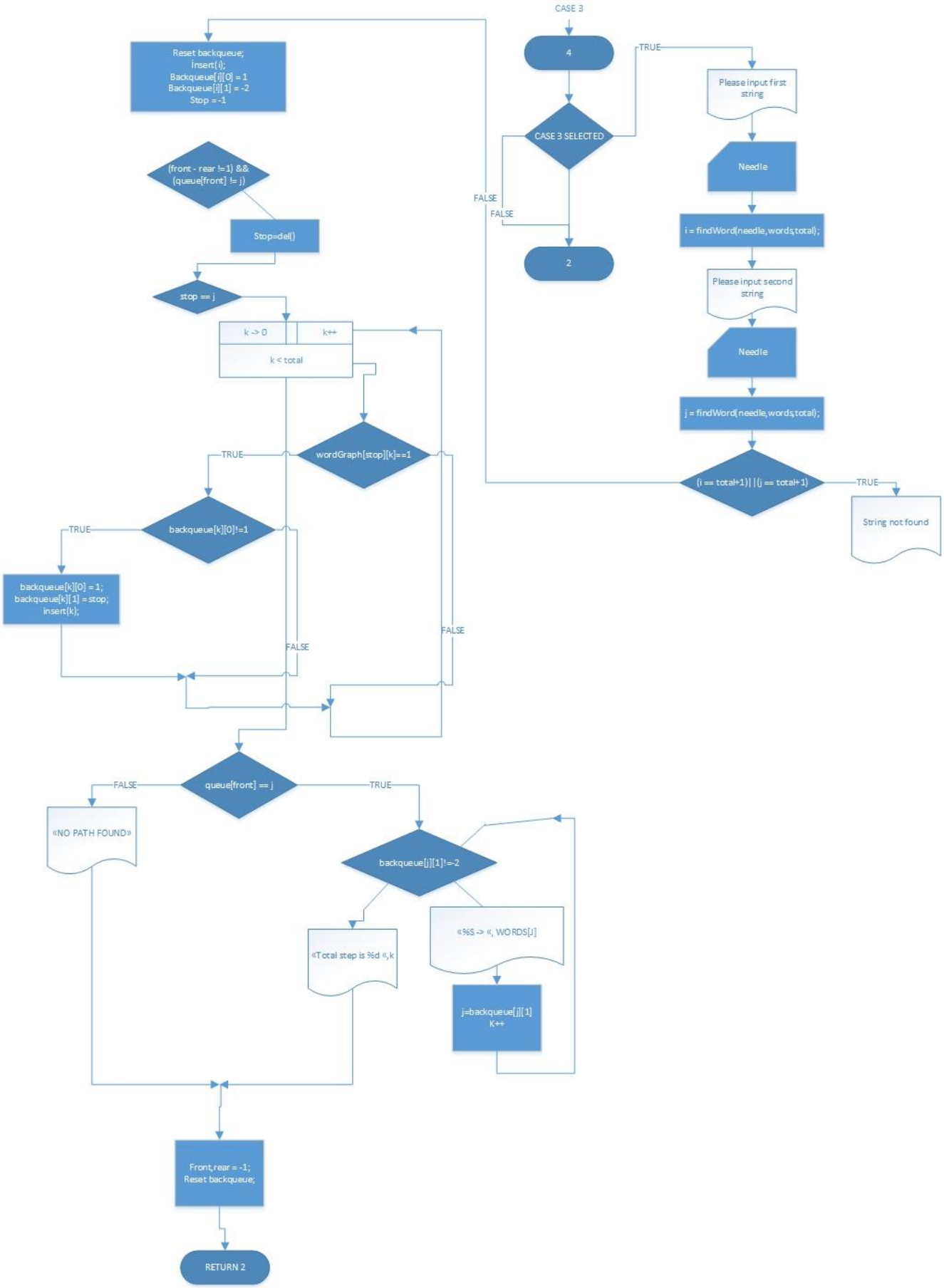
Verilen iki kelime için, birinci kelimeden ikinci kelimeye dönüşüm olup olmadığını bulmak için **kuyruk(queue)** veri yapısı kullanılacaktır. Bunun için aşağıdaki adımları uygulayınız:

- İlk adımda **birinci kelime'ye ait düğümü** kuyruğa yerleştiriniz.
- Kuyruk **tamamen boşalana kadar veya çıkış düğümüne(ikinci kelime) ulaşana kadar** aşağıdaki işlemleri yapınız:
  - Kuyruğun başındaki düğümü kuyruktan çekiniz.
  - Eğer **bu düğüm ikinci kelime ise** işlem tamamlanmıştır.
  - Eğer **bu düğüm ikinci kelime değilse** düğüme ait kelimeyi ekrana yazdırınız, Bu **düğümün daha önceden kuyruğa gimemiş** bütün komşularını kuyruğa ekleyiniz. Bir düğümün daha önceden kuyruğa girmiş olup olmadığını görmek için yedek bir dizi kullanabilirsiniz.

Şimdiye kadar yaptığımız adımla birinci kelimeye ait düğümü eklemek. Sorunun devamında bir düğümün daha önceden kuyruğa girmiş olup olmadığını görmek için kullanılacak yedek dizi ise *backqueue[i][0]* olarak kullanılmıştır. Buna ek olarak *backqueue* matrisinde parent idler tutulmaktadır. Daha sonra okutmanımızın yazdığı algoritma gerçeklenmiştir. Bu algoritma, mantık olarak kuyruğa ilk giren ilk çıkacağı için, önce 1. Dereceden komşuları, ardından 2. Dereceden komşuları vs. Diye kontrol etmektedir. Ve eğer çıkış düğümüne ulaşırsa durmaktadır. Birnevi Breadth First Search yapılmaktadır. Burada, birden fazla yol olabilir fakat soruda *EN KISA YOLU BULUNUZ, FARKLI YOLLARI GÖSTERİNİZ* gibi bir ifade kullanılmadığı için, hedef düğüm bulunduğunda, iterasyondan çıkılmakta, ardından iterasyondan neden çıkıldığı kontrol edilmektedir. Eğer çıkış düğümü bulunduğu için çıkıldıysa bir yol var demektir, eğer yoksa iki kelime arasında path yok denilebilir. Çıkış düğümü bulunduysa, çıkış düğümünden itibaren, *backqueue* matrisinde parentlardan itare edilerek path ekrana çıktı olarak verilir. Bütün işlemler tamamlandıktan sonra ana menüye dönmeden önce *rear* ve *front* değişkenleri -1 yapılarak kuyruk 0 lanmış olur. Kuyruk implemantasyonunda, struct kullanmak yerine *queue, rear* ve *front* değişkenleri kullanılarak çözüm sağlanmıştır. Derste anlatılan aksine kuyruğu tamamen yazdırmak, ortasına eklemek vs. Gibi işlemler bu soru için gerekli olmadığından, sadece insert ve delete fonksyonları gerektiğinden basit bir çözüm ortaya konulmuştur. Bahsi geçen fonksyonlar ileride anlatılacaktır. Ayrıca bilinmesi gerekenler, insert içeriye aldığı id yi , kuyruğun sonuna ekler, delete ise kuyruk başından eleman çıkarır, çıkardığı elemanın idsini ise return eder. İkinci kelime değilse, bu düğümün önceden kuyruğa girmemiş bütün komşularını kuyruğa ekleme kısmı;

```
printf("%s ||",words[stop]);//print that word
for(k=0;k<total;k++){
    if(wordGraph[stop][k]==1){//get connected nodes
           // printf("%s(%d) komsudur %s(%d)  ",words[stop],stop,words[k],k);
           // printf("\n backqueue %d \n", backqueue[k][0]);
        if(backqueue[k][0]!=1){//if it is in queue before, skip, else;
            backqueue[k][0] = 1;//make it visited
            backqueue[k][1] = stop;//save parent id
            insert(k);//insert to queue
        }
    }
}
```

Kod parçası ile gerçeklenmiştir. Burada *backqueue[k][0] kısmı,* düğümlerin daha önce kuyruğa girip girmediğini kontrol eder. Eğer kuyruğa daha önceden girmemiş ise, aynı indisi 1 yaparak ziyaret edilmiş olarak işaretlenir, ayrıca parenti da tutulur, ardından kuyruğa eklenir. Buna ek olarak, while döngüsü içerisindeki *(front - rear !=1) && (queue[front] != j)* argümanlardan ilki, kuyruğun boş olmadığını, ikincisi ise kuyruğun başında çıkış düğümü bulunmadığını kontrol eder.Aşağıda akış diyagramı verilmiştir. Görsellerin yüksek çözünürlüklü halleri rapor dosyasında veya github sayfasında bulunabilir.


CASE 3
4
CASE 3 SELECTED
TRUE
FALSE
FALSE
2
Please input first string
Needle
i = findWord(needle,words,total);
Please input second string
Needle
j = findWord(needle,words,total);
(i == total+1)||(j == total+1)
TRUE
String not found
Reset backqueue;
Insert(i);
Backqueue[i][0] = 1
Backqueue[i][1] = -2
Stop = -1
(front - rear != 1) &&
(queue[front] != j)
Stop=del()
stop == j
k -> 0
k++
k < total
wordGraph[stop][k]==1
TRUE
backqueue[k][0]!=1
TRUE
backqueue[k][0] = 1;
backqueue[k][1] = stop;
insert(k);
FALSE
FALSE
queue[front] == j
FALSE
TRUE
«NO PATH FOUND»
backqueue[j][1]!=-2
«Total step is %d «,k
«%S -> «, WORDS[J]
j=backqueue[j][1]
K++
Front,rear = -1;
Reset backqueue;
RETURN 2

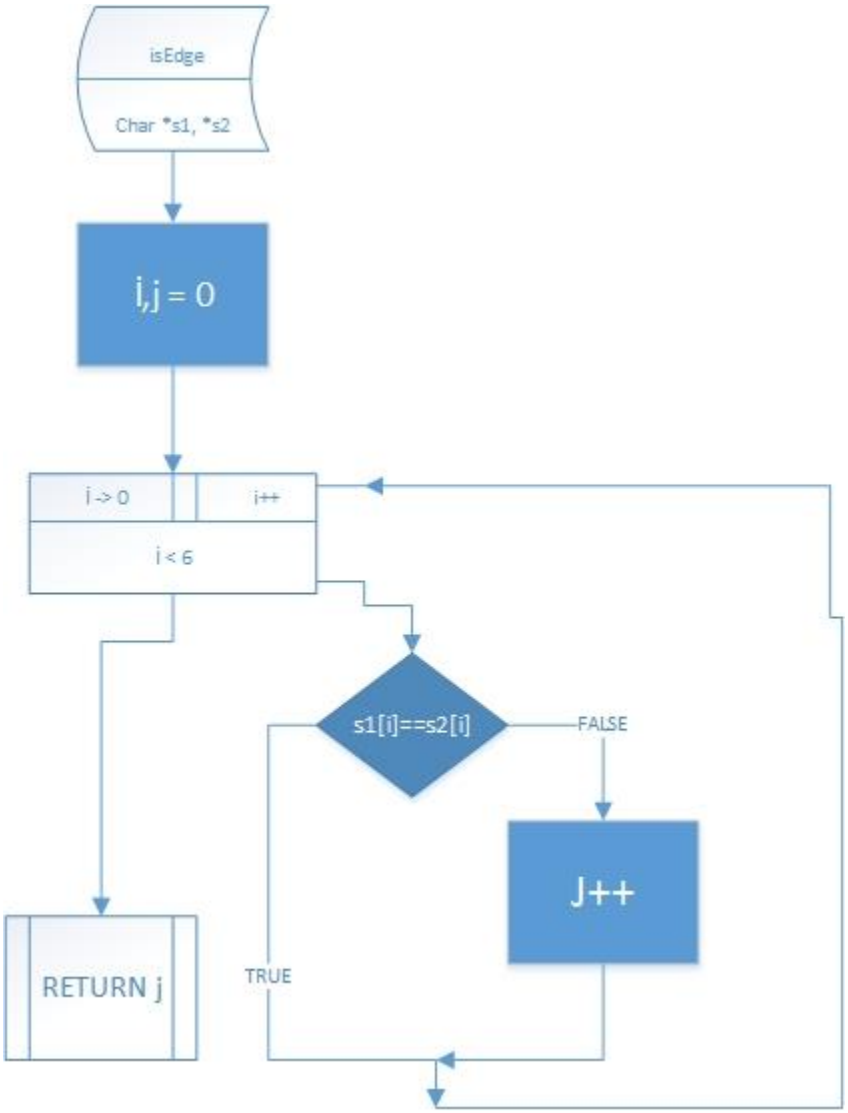
Diğer fonksyonlar:

Yukarıda bahsi geçen yardımcı olarak kullanılan fonksyonlar ve içeriye aldığı değerler ;

```
int isEdge(char *s1, char *s2);
void filladj(unsigned char **wordGraph, int total,char words[TOT][10]);
int findWord(char *needle,char words[TOT][10],int total);
void insert(int id);
```

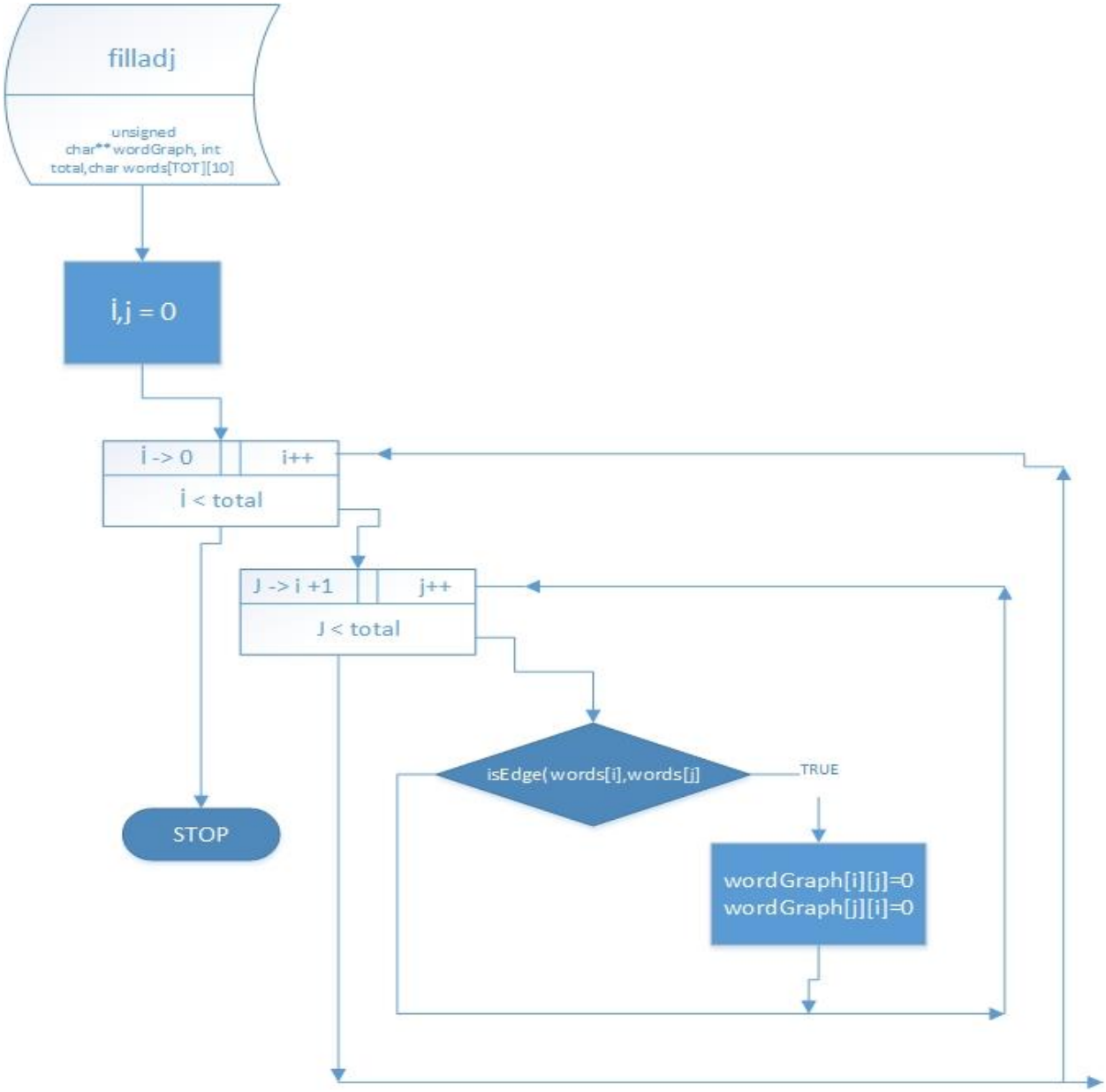
isEdge fonksyonu:

isEdge fonksyonu, verilen iki karakter arasındaki toplam değişik karakter miktarını geriye döndürür. Kullanım amacı, geriye dönen sayı 1 ise , edge var demek için olduğundan ismi isEdge olarak verilmiştir.İçerisinde i ve j isminde iki farklı değişken tanımlanmıştır. İ değişkeni 0 dan 5 e kadar yani, bütün karakterleri kontrol eder, kelimelerimizin hepsi 5 basamaklı olduğu için i<6 koşulu getirilip, verilen string üzerinde karakter kıyaslaması yapılmaktadır. Eğer karakterler birbirinden farklı ise, j değişkeni 1 arttırılır. Fonksyon geriye j değişkenini döndür. Fonksyona ait akış şeması aşağıdadır;



Filladj fonksyonu:

Filladj fonskyonu içeriye nxn matris alır, toplam karakter sayısı ve kelime dizisini alır. Bu fonksyonun amacı adj. Matrisin içini doldurmaktır. Bunu yaparken klasik simetrik matris doldurma stratejisi kullanılmaktadır. Burada matrisi gezmek için 2 değişken kullanılır, birisi i diğeri j, i değişkeni 0 dan toplam kelime sayısına kadar ilerlerken, j değişkeni onun içerisinde i+1 den başlayıp toplam kelime sayısına kadar ilerler. Bu ilerlemede, i ve j düğümleri arasındaki komşuluğa bakılır, eğer komşuluk var ise grafın i,j. Ve simetrik olduğu için j,i. Düğümleri 1 yapılır. Değilse zaten 0 ile başlangıçda doldurulduğu için bir şey yapılmaz.

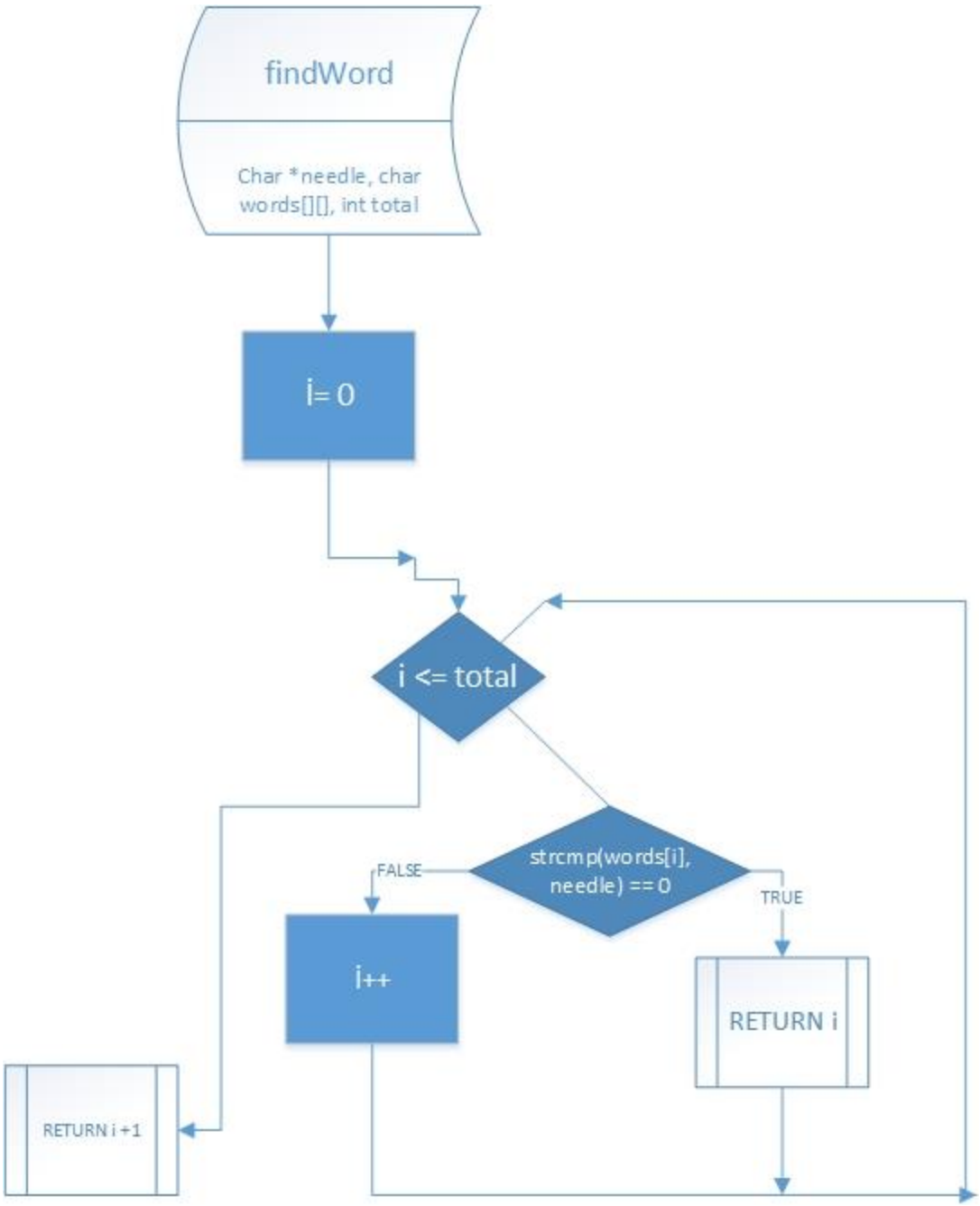


İnsert ve del fonksyonları;

Bu fonksyonlar kuyruğa eleman ekleme ve çıkartma işlemlerinde kullanılmaktadır. Kuyruğun başlangıcında front ve rear -1 olarak ilklendirilmiştir. İnsert edildiğinde, eğer kuyruk boş ise, front = 0 yapılır. Ardından rear 1 arttırılarak içeriye gelen eleman idsi, queuenin rearıncı indisine eklenir. Delete fonksyonu ise frontu 1 arttırır. Böylece aslında kuyruktan eleman fiziksel olarak silinmez ama başlangıç adresi değiştiği için silinmiş kabul edilir. Silinen id geri döndürülür.

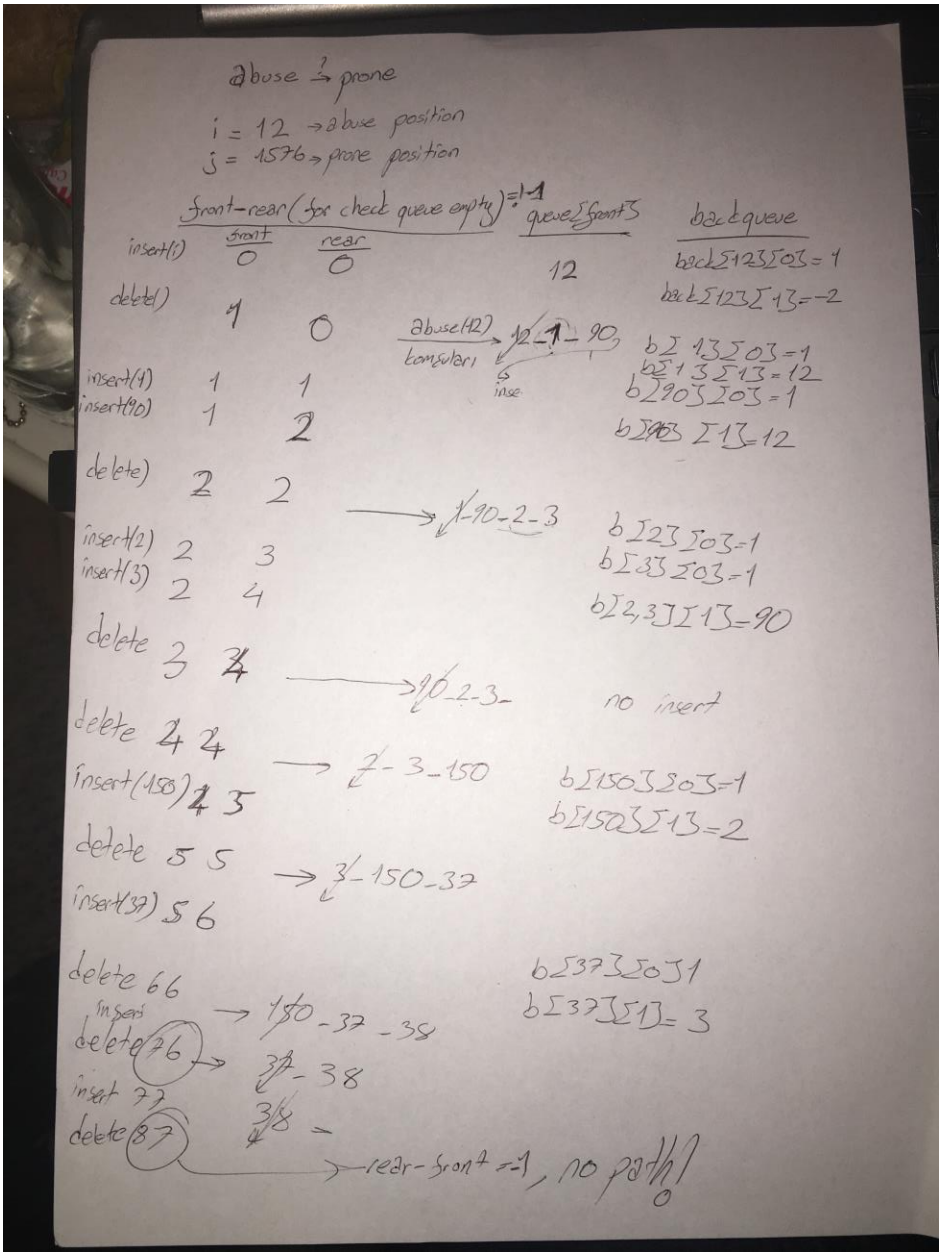
Findword fonksyonu:

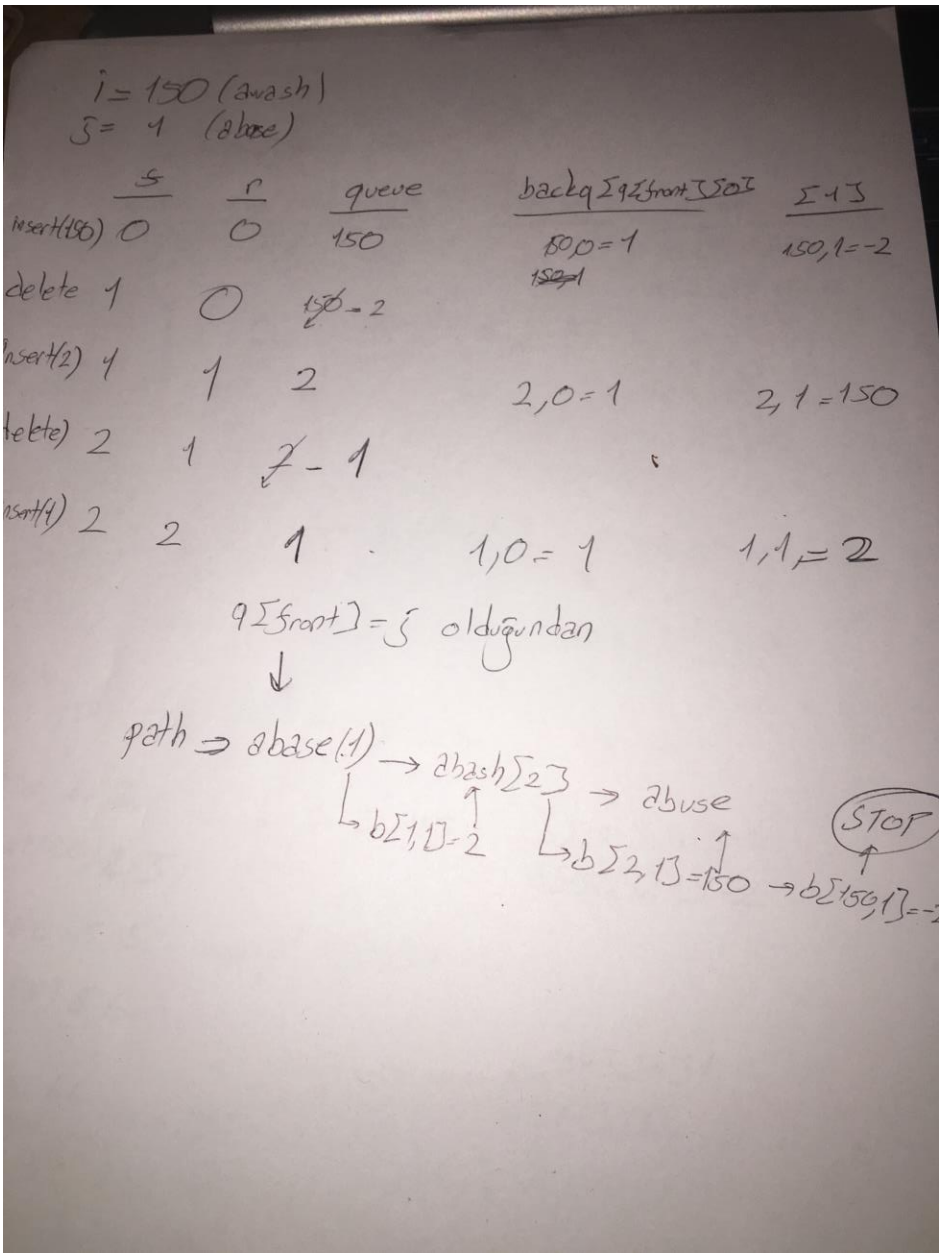
Bu fonksyon, verilen kelimenin hangi indiste olduğu bilgisini geri döndürmekle görevlidir. İçeriye *needle, words ve total* değişkenlerini alır. İçeride bütün düğümleri gezmeye yarıyacak while i<=total koşulu ile iterasyona başlanır. Eğer strcmp fonksyonu, 0 döndürür ise, kelime bulunmuş demektir, geriye indis olan i döndürülür. Eğer whiledan çıkılır ise, kelime bulunmamış olduğundan ve i nin son değeri total olduğundan geriye hiçbir kelimeye işaret etmeyen *total+1* değişkeni döndürülür. Akış şeması aşağıdaki gibidir;

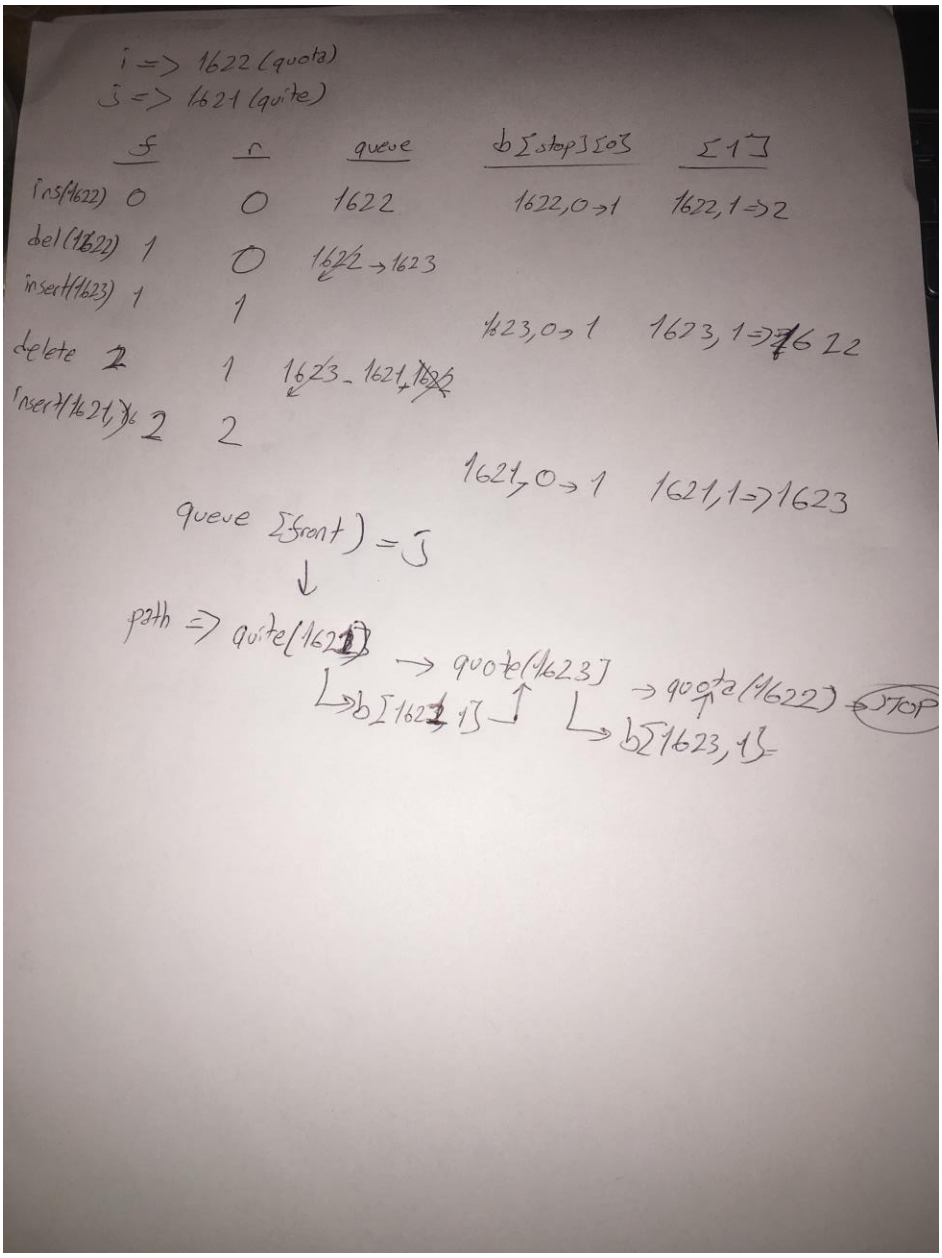

findWord
Char *needle, char words[][], int total
i= 0
i <= total
strcmp(words[i], needle) == 0
FALSE
TRUE
i++
RETURN i
RETURN i +1

2.UYGULAMA BÖLÜMÜ

Burada el yordamıyla 3 adet örnek çözümlenip, analizin görselleri aşağıda verilmiştir.




i = 150 (awash)
j = 1 (abase)
s r queve backq[q[front]][0] [1]
insert(150) 0 0 150 80,0 = 1 150,1 = -2
delete 1 0 150 - 2
insert(2) 1 1 2 2,0 = 1 2,1 = 150
delete) 2 1 2 - 1
insert(1) 2 2 1 1,0 = 1 1,1 = 2
q[front] = j olduğundan
path ⇒ abase(1) → abash[2] → abuse
b[1,1] = 2
b[2,1] = 150 → b[150,1] = -2
STOP



3.KOD BÖLÜMÜ

```
/* @source Yıldız Technical University, İstanbul
** avaible @ www.github.com/turkerozan/dataStructuresH3
** This was created for assigment solution, no requests will be accepted
** All viewer and/or forkers will be reported to instructor if it is
requested
** If there is a copy situation, there is a lots of proof in github so you
can see author, so be careful
**
** @author: Ozan Turker Demir, (ozanturkerdemir@gmail.com)
** @@
**
*/
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

#define TOT 2415 //number of words

int isEdge(char *s1, char *s2);
void filladj(unsigned char **wordGraph, int total,char words[TOT][10]);
int findWord(char *needle,char words[TOT][10],int total);
void insert(int id);
int del();
int queue[TOT];//Global variable queue,
int rear = -1;//queue's rear position
int front = -1;//queue's front position
int main()
{
    char words[TOT][10];//for saving all words,
    int backqueue[TOT][2];//for saving state of queue, if it is visited, we
will change 0.column, other column is used to save parent id
    FILE *input; //file pointer
    int i = 0;//i and j are used for getting position of words
    int j = 0;
    char needle[10];//for string search
    int k;//iterator
    int choice;//menu choice variable
    int total = 0;//total number of words, we can use TOT but if something
goes wrong, or if we want to debug process, we may want to see that
    unsigned char **wordGraph; // adj. matrix variable
    //declaring pointer for reading file
    input = fopen("kelime.txt","r");
    //error handling if open fails
    if(input == NULL ){
        printf("unable to open file\n");
        exit(1);
    }
    //for debug purpose
    else{
        printf("File opened successfully\n");
    }
    while(fgets(words[i], 10 , input)){//10 is buffer size
        words[i][strlen(words[i]) - 1] = '\0';//we need to get rid of \0
character, basic reading function
        i++;
    }
    total = i;//we get readed number of words
    fclose(input);//close the pointer

    //now we got total number of words before closing input
    //for debug purpose
   // for(i = 0; i < total; i++){
    //printf("%d : %s\n",i, words[i]);}
    //We can allocate memory; NxN adjacent matrix
    wordGraph = malloc((total+1) * sizeof(unsigned char *));//total+1 becuase
iterator start with 0 but we need +1 more for memory
    if(wordGraph == NULL){
        printf("Memory error \n");
        exit(1);
    }
```

```
    for(i = 0; i< total+1; i++){
        wordGraph[i] = malloc((total+1) * sizeof(unsigned char));
        if(wordGraph[i] == NULL ){
        printf("Memory error \n");
        exit(1);
        }
    }
    //For edges it should be 0 first, we will fill later
    for(i=0; i< total;i++){
        wordGraph[i][i]=0;
    }
    //Fill adj. Matrix with function
    filladj(wordGraph,total,words);
    //Menu implementation after all of work are done without errors
    do{
        do{
            printf("\n 1.Compare strings with id");
            printf("\n 2.Compare strings with input");
            printf("\n 3.Find Transaction between strings");
            printf("\n 4.Exit");
            scanf("%d",&choice);
            if(choice<1||choice>4)
                printf("\n Invalid choice ");
        }while(choice <1 || choice >4);
        switch(choice){
    case 1:
        printf("\n Please input id respectively");
        //Id means the position in array, if known
        printf("\n First ID : ");
        scanf("%d",&i);
        printf("\n Second ID : ");
        scanf("%d",&j);
        if(wordGraph[i][j]== 1){//if it is 1; then it means there is an
connection
        printf("\n WORD %s and WORD %s has %d
connection",words[i],words[j],wordGraph[i][j]);}
        else{
            printf("\n NO CONNECTION");
        }
        break;
    case 2://findword function gets position of given word
        printf("\n Enter string 1 : ");
        scanf("%s",needle);
        i = findWord(needle,words,total);
        printf("\n Enter string 2 : ");
        scanf("%s", needle);
        j = findWord(needle,words,total);;
        //if findword gives us latest number, we can say it cannot find
string if it overflowed total number of words
        if((i == total+1)||(j == total+1)){
            printf("\n String Not found ");
        }
        else{//else we found them and print positions and condition
            printf("\n positions %d :%d", i,j);
            if(wordGraph[i][j]== 1){//if there is a connection, our adj.
matrix will tell us
```

```
        printf("\n WORD %s and WORD %s has %d
connection",words[i],words[j],wordGraph[i][j]);}
        else{
            printf("\n NO CONNECTION");
        }
        }
        break;
    case 3:
        printf("\n Start string :");//same as case 2, find positions first
        scanf("%s",needle);
        i = findWord(needle,words,total);
        printf("\n Stop string :");
        scanf("%s",needle);
        j = findWord(needle,words,total);
        if((i == total+1)||(j == total+1)){
            printf("\n String not found");
        }else{//if we found strings, than initilaze backqueue array with 0
and -1, 0 means no visited, -1 is parent id which is not avaible for first
time
            //RESET BACKQUEUE
        for(k=0;k<TOT;k++){
            backqueue[k][0] = 0;
            backqueue[k][1] = -1;
        }
        // i and j are positions of words.
        printf("First word pos : %d second word pos : %d \n",i,j);
        insert(i);//insert first node
        backqueue[i][0] = 1;//make it visited
        backqueue[i][1] = -2;//for controll, we made root node parent id -2,
we will use that information if we will find path
        int stop = -1;//for controlling deleted member
        while((front - rear !=1) && (queue[front] != j)){//while will broke
when queue is empty or front member is 2nd word
            stop = del();//take word from queue, return id
            if(stop == j){//if it is 2nd word(desired word) then print
status,
                printf("islem tamam \n");
            }
            else{
                printf("%s ||",words[stop]);//print that word
                for(k=0;k<total;k++){
                    if(wordGraph[stop][k]==1){//get connected nodes
                            // printf("%s(%d) komsudur %s(%d)
",words[stop],stop,words[k],k);
                            // printf("\n backqueue %d \n", backqueue[k][0]);
                        if(backqueue[k][0]!=1){//if it is in queue before,
skip, else;
                            backqueue[k][0] = 1;//make it visited
                            backqueue[k][1] = stop;//save parent id
                            insert(k);//insert to queue
                        }
                    }
                }
                //printf("\n QUEUE \n");
                //display();
            }
        }
```

```
        if(queue[front] == j){//when while is broked, we have to check ,we
found path or not?
        printf("\n %s",words[j]);//if top member of queue is 2nd word, then
we can say we found a path
        printf(" \n There is a path like : ");//print path
        k=0;//step size iterator
        do{
                k++;
                printf("%s->",words[j]);//get word id and print it
                //printf("parent : %d %s ",backqueue[j][1],
words[backqueue[j][1]]);
                j=backqueue[j][1];//change id to parent
        }while(backqueue[j][1]!=-2);//when we hit root node
        printf("%s",words[j]);//print root node
        printf("\n Total step size is : %d ",k);
        }
        else{
            printf("\n No path found ");
        }
        front = -1;//when all calculations done, reset the queue
        rear = -1;
         //RESET BACKQUEUE
        for(k=0;k<TOT;k++){//also reset the backqueue
            backqueue[k][0]=0;
            backqueue[k][1]=-1;
        }
        }
        break;
    default:
        printf("\n End of program");//Before end, do not forget to free our
matrix
        for(i = 0; i < total +1; i++)
        free(wordGraph[i]);
      free(wordGraph);
        }
    }while(choice != 4);

    return 0;
}
int findWord(char *needle,char words[TOT][10],int total){
    //in this function, we need a word, and word array and total number of
words
    int i = 0;//iterator
    int tmp = 1;//status
while(i <= total ){//start 0 to total, if strcmp returns 0, then it means we
found word, return it number
            tmp = strcmp(words[i], needle);
            if(tmp==0){
                    //printf("%d",i);
                return i;
            }
            i++;
        }
        return i+1;//if we cannot found, return total+1
}

void filladj(unsigned char **wordGraph, int total,char words[TOT][10]){
```

```
    int i,j,k;
for(i = 0; i <= total ; i++){
        for(j = i+1; j<total ; j++){
                k = isEdge(words[i],words[j]);//edge returns number of
different chars
            if( k == 1){//if k = 1; then it means only 1 difference occured
                wordGraph[i][j]=1;//it is symetrical
                wordGraph[j][i]=1;//thats why i,j = j,i
            }
            else{//else fill with 0
                wordGraph[i][j]=0;
                wordGraph[j][i]=0;
            }
        }
    }
}

int isEdge(char *s1, char *s2){
int i,j;
j = 0;//this function gets two strings
for(i=0;i<6;i++){//max number of char is 5, so we compare to 5
    if(s1[i]==s2[i]){//if char is same, then do nothing
    //printf("karakter %d ayni\n", i);
    }
    else{j++;}//if chars are not same, increase iterator
}
return j;//return amount of chars that are different
}

void insert(int id){
    if(rear == 2414){//if we hit the size of stack
        printf("overflow error \n");
    }
    else{//if not
        if(front == -1){//if it is first one
            front = 0;}//make front 0
            rear ++;//increase rear, make area for node
            queue[rear]= id;//insert node to last position of queue

    }
}
int del(){
if(front == -1 ){//if we delete from empty queue
    printf("underrflow error \n");
    return; //return with no int, so program will exit
}
    else{
        front++;//increase front, so queue front is poped
    }
return queue[front - 1];//return poped value
}
```