1 - C PROGRAMLAMA DİLİ

1.1. GİRİŞ

C programlama dili, Pascal, Basic, Fortran ve diğer birçok dil gibi yordamsal (procedural) bir dildir. Yordamsal paradigmada problemler 3 temel işlem ile çözülmektedir. Bunlar, sıralı işlemler (sequential commands), koşullu işlemler (conditional commands) ve döngülerdir (loops). Tabi bunlara ek olarak girdi/çıktı (input/output) işlemleri ve bilgilerin saklanması için değişkenler (variables) ile bilgilerin aktarılması için de en temel işlem olarak atama işlemi (assignment) bulunmaktadır.

1.2. C PROGRAMLARININ YAPISI

Basit bir C programı tanımlamalar ve işlevlerden oluşur. İşlevler ise tanımlamalar ve işlevi oluşturan cümlelerden oluşur. Bütün bu tanımlamalar ve cümleler noktalı virgül (;) ile biter. Bu yapılar aşağıdakileri içermektedir:

1.2.1. Veri Yapıları Tanımları

Bu tanımlamanın sözdizimi şöyledir:

veri-tipi değişken listesi;

C dilinde, tanımlandıktan sonra, o noktanın altında, programın her yerinde geçerli olacak şekilde (global), yani işlev tanımlarının dışında da herhangi bir yerde, veri yapıları tanımlanabilir. İşlevlerin içinde ise sadece o işlevde geçerli olan tanımlamalar (local) yapılabilir. Bu tanımlamalar işlevin başlangıcında tüm diğer işlemlerden önce yapılır. C dilinde bulunan temel veri yapıları şunlardır; int, short, long, float, double, char,.... Meselâ,

int i,say;

tanımlaması ile 2 adet tamsayı değişken tanımlanmış olur. Değişkenlere ilk değer atanması tanımlama sırasında da yapılabilir. Bu işlemin sözdizimi şu şekildedir:

veri-türü değişken-adı = ilk-değer....

Örneğin yukarıdaki tanımlamalar sırasında bu değişkenlerden i'nin ilk değer olarak 0 ile atanması tanımlama sırasında şöyle gerçekleştirilebilir:

int i=0,say;

1.2.2. İşlev Tanımları

İşlev tanımlama sözdizimi aşağıdaki gibidir:

[dönen-veri-türü] işlev-adı (? veri-türü değişken? listesi) { }

İşlevler programın herhangi bir yerinde tanımlanabilir. Genel amaçlı main işlevi de tamamen bağımsız bir işlev olmak zorunda olup, program çalıştırıldığında ilk çağrılacak (call) işlevdir. Eğer işlevlerin çağırıldıkları ortama bir değer döndürmeleri istenirse bu durumda işlevin döndürdüğü veri türü işlev tanımı sırasında belirtilir, değilse işlevin veri türü olarak void (boş) belirtilir. İşlevin bir değer döndürebilmesi için işlevin tanımlandığı kodun içinde özel bir komut olan return komutu kullanılır. Bu komutun sözdizimi şöyledir:

return *(ifade);*

Bu komutun döndürdüğü değerin türü ile işlev sırasında belirtilen işlev türü aynı olmalıdır. İşlevler şöyle çağırılabilir; işlevin döndürdüğü türden değerlerin kullanılabileceği ifadelerde bu türden bir değer yerine işlev adı ve işleve gönderilmek istenen değerler de parametre olarak yazılarak işlev çağırılır. Eğer işlev bir değer döndürmüyorsa basit bir cümle olarak işlev çağırılabilir. Bu işlemin sözdizimi şu şekildedir:

işlev-adı (parametre listesi)

Bu çağırma işlemi sırasında parametre listesindeki her bir eleman için, o ifadenin sonucu bulunarak bu değer işlev tanımında aynı sırada tanımlanmış parametre değişkene atanır ve işlev çalışmaya başlarken parametre değişken bu ilk değere sahip olur. İşlev tanımlarında o işlev için yapılan lokal tanımlamalar gibi parametreler de işlev tamamlanıp çağırıldığı ortama dönerken yok edilir. Zaten lokal tanımlamalar ile parametre olarak tanımlanan değişkenler arasındaki tek fark sadece, işlevde parametre olarak tanımlanan değişkenlerin işlev çağırılması sırasında parametrelerde bulunan ifadelerin sonuçlarını ilk değerleri olarak almalarıdır.
İşlevlerin tanımı programın içinde istenilen sırada yapılabilir. Ancak bir işlevin diğerini çağırabilmesi için çağırılanın daha yukarıda tanımlanmış olması gerekir. Bunun için en iyi yöntem işlemlerin prototip tanımlarını (yani {} kısmı ile yapılan gerçek tanımları dışında kalan kısımları) programın en başında yaparak tüm işlevleri tüm işlevler için tanımlı hale getirmektir. Bundan sonra standart yöntem önce ana işlev olan main işlevinin, onun altına da diğer işlevlerin tanımlarının yapılmasıdır. Programın başında tüm işlevlerin prototiplerinin tanımları yer alacağı için aşağıda yapılan gerçek tanımlar sırasında her bir işlem diğerlerini çağırabilecektir.
Aşağıdaki 3. örnekte main işlevi dışında işlevler de tanımlanmıştır.

1.2.3. Aritmetik, Mantıksal İfadeler

Aritmetik ifadeler +,-,*,/ gibi aritmetik operatörler ve bu işlemlere katılan sabit ve değişkenlerden oluşur ve işlem sonunda ilgili türde bir veri oluşur.

Meselâ 12-3*3 aritmetik ifadesi sonucunda 3 değeri oluşur. Bu ifadedeki sabitlerin yerine tamsayı değerlere sahip değişkenler de yer alabilir. Aritmetik ifadeler gerçekleştirilirken işlem sırası farklı operatörler bulunurken bu operatörlere göre belirlenir. Bu belirleme kuralına öncelik sırası denir.
Örneğin yukarıdaki ifadede önce çarpma işlemi (3*3) gerçekleşecek ve ondan sonra çıkarma işlemi (12-9) yapılacaktır. Öncelik sırası parantezler kullanılarak kullanıcı tarafından da belirlenebilir. Çarpma ve bölme işlevleri toplama ve çıkarma işlemlerinden daha yüksek bir önceliğe sahiptir. Ayrıca öncelik sırası aynı olan operatörler arasında ise (örneğin 3-2+4) işlemler soldan sağa doğru gerçekleştirilir (bu örnekte sonuç 3-2'nin sonucu 1 ve 1+4'ü sonucu da 5 olacaktır). C dilinde başka bir çok operatör bulunmaktadır. Bunlardan bazıları daha yüksek bazları daha düşük öncelik sıralarına ve bazları da sağdan sola işlenme özelliğine sahiptir. Bunların tam listesi için referans kitaplara başvurmanızı öneririz.
Aritmetik ifadelere benzer şekilde mantıksal ifadeler oluşturmak için de >,>= (büyük ya da eşit), == (eşit),... gibi birçok karşılaştırma operatörü kullanılarak sonucu DOĞRU ya da YANLIŞ olarak belirlenecek olan mantıksal ifadeler oluşturulabilir. Bu mantıksal ifadelerin sonucu doğru ise bunu ifade etmek için 0'dan farklı herhangi bir değer, yanlış ise 0 değeri kullanılır. Örneğin 3>2 ifadesiin sonucu 0'dan farklı herhangi bir değer, yani DOĞRU olacak; ama 3<2 ifadesinin sonucu YANLIŞ'ı ifade etmek için 0 olacaktır. Aritmetik operatörlerde olduğu gibi bu operatörler de soldan sağa gerçekleştirilir. Ancak sonuçları YANIŞ ve DOĞRU'ları ifade ettiği için karışıklıkları önlemek için parantezlenerek kullanılmaları yararlı olacaktır. Ayrıca bu karşılaştırma operatörlerine ek olarak mantıksal operatörler olan VE (and) için &&, VEYA (or) için ||, DEĞİL (not) için ise ! operatörleri kullanılarak istenilen mantıksal ifade oluşturulabilir. Yine aritmetik operatörlerde olduğu gibi değişkenler ya da sabitler bu ifadelerde kullanılabilir. Örneğin (3>=3) && (2==3) ifadesi VE operatörünün solundaki mantıksal ifade DOĞRU olmasına rağmen, sağındaki ifade YANLIŞ olduğu için sonuç olarak YANLIŞ'ı ifade eden 0 değerini döndürecektir. Bu konunun da tüm detaylarını referans kitaplarda bulabilirsiniz.

1.2.4. Atama İşlemleri:

Atama işlemi bir değişkene yeni bir değerin yüklenmesini sağlayan basit bir işlemdir. Bu nedenle atama işlemi, değişken kavramı ile birlikte anlam kazanmakta ve yordamsal dillerdeki en temel işlemi oluşturmaktadır. Yordamsal dillerde değişkenlere yüklenen veriler atama işlemleri ile problemlerin çözüm aşamalarında değişebilmekte ve bu şekilde adım adım problem çözülebilmektedir. C dilindeki en basit atama işleminin sözdizimi şöyledir:
değişken = ifade ;
Atama işlemi = operatörü ile yapılmakta ve bu cümle işlendiği zaman = operatörünün sağ tarafındaki ifadenin sonucu belirlenip sol taraftaki değişkene atanmaktadır. Meselâ,

*i=12-3*3;*

satırının gerçekleştirilmesinden sonra i değişkeni 3 değerine sahip olacak, eğer önceden bu değişkenin başka bir değeri varsa bu değer de yok olacaktır.
C dilinde bazı çok kullanılan özel atama işlemleri için farklı sözdizimleri de geliştirilmiştir. Bunun amacı derleyiciye bu özel atama işlemlerinin daha verimli bir şekilde yapılabileceğini belirtmektir. Bunlar değişkenin değerini 1 arttırıp azaltan atama işlemleri ile değişkenin değerine bir ifadenin sonucunu ekleyen, çıkaran, bir ifadenin sonucu ile çarpıp, bölen atama işlemleridir. Bunların sözdizimleri şöyledir:

değişken++	değişkenin değerini 1 arttırır
++değişken	değişkenin değerini 1 arttırır
değişken--	değişkenin değerini 1 azaltır
--değişken	değişkenin değerini 1 azaltır
değişken+=ifade	değişkenin değerine ifadenin sonucunu ekler
değişken-=ifade	değişkenin değerinden ifadenin sonucunu çıkarır
değişken*=ifade	değişkenin değerini ifadenin sonucu ile çarpar
değişken/=ifade	değişkenin değerini ifadenin sonucuna böler

İlk iki arttırma ve sonraki iki azaltma işlemleri yalnız başına tek bir cümle olarak kullanıldıklarında aynı şekilde çalışırlar. Ancak bu işlemler başka atama cümlelerinde ya da başka ifadelerde de yer alabilirler. Bu gibi durumlarda ++ ve -- operatörünün değişkenin solunda ya da sağında yer alması farklı sonuçlar çıkarır. Eğer bu operatörler değişkenin sol tarafında yer alırsa önce operatör değişkene uygulanır ve daha sonra değişken içinde bulunduğu ifadede bu yeni değeri ile kullanılır. Eğer bu operatörler değişkenin sağ tarafında yer alırsa bu durumda değişkenin operatör uygulanmadan önceki değeri içinde bulunduğu ifadede kullanıldıktan sonra operatör değişkene uygulanır. Meselâ,

i=3;
k=++i;

satırları işlendiğinde k'nın değeri ve i'nn değeri 4 olacaktır. Halbuki:

i=3;
k=i++;

satırları işlendiğinde ise k'nın değeri 3ç i'nin değeri ise 4 olacaktır.
Diğer atama işlemleri ise sol taraftaki değişken üzerine bir işlem yapılmasını sağlar. Meselâ,

i=4;
i=12-3;*

satırları işlendiğinde aslında i=i*(12-3) atama işlemine eş bir sonuç oluşturacak ve i'nin değeri 36 olacaktır.

1.2.5. Gerekçeli İşlemler, Döngüler, Bloklar:
Yordamsal dillerde programın akışını kontrol eden en temel iki işlem gerekçeli işlemler ve döngülerdir.
Gerekçeli işlemler için birçok yordamsal dilde olduğu gibi C dilinde de 'EĞER gerekçe doğru İSE birinci cümleyi DEĞİLSE ikinci cümleyi gerçekleştir' şeklinde bir iş akışını oluşturan iki tür 'if' ve 'if-else' cümleleri bulunmaktadır.
Bu cümlelerin sözdizimleri şöyledir:

if *(koşul) cümle_1;*
if *(koşul) cümle_1;* ***else*** *cümle_2;*

Bu iki türden cümlede eğer koşul mantıksal ifadesi DOĞRU yani 0'dan farklı bir değer döndürürse 1. cümle gerçekleştirilir. İkinci cümlede koşul mantıksal ifadesinin sonucu YANLIŞ yani 0 olursa 2. cümle gerçekleştirilir. İki durumda da bu işlemlerden sonra programın akışı bir sonraki cümlenin çalıştırılması ile sürer. Aşağıdaki örneklerde bu cümlelerin uygulamalarını bulabilirsiniz.
Bir diğer gerekçeli işlem cümlesi ise switch cümlesidir. Bu cümlenin sözdizimi şu şekildedir:

```
switch (tamsayı-ifade) {
   case-başlıklı-ifade
   ...
   case-başlıklı-ifade
}
```

Her bir case-başlıklı-ifade ise şu sözdizimine sahiptir:

case *tamsayı : cümle_1; ... cümle_n;*

ya da:

default: *cümle_1; ... cümle_n;*

Bir **switch** cümlesinde önce tamsayı-ifade'nin sonucu bulunur ve bu değere karşılık gelen tamsayı'nın bulunduğu case satırındaki cümleler gerçekleştirilir. Eğer hiçbir **case** satırı bu değere sahip değilse varsa **default** satırındaki cümleler gerçekleştirilir. Bu cümlenin ilginç bir özelliği eğer herhangi bir

case satırından itibaren cümleler gerçekleştirilmeye başlanırsa bu işlem bir sonraki **case** satırında son bulmaz ve o satırdaki cümleler de gerçekleştirilerek devam eder. Bunu durdurabilmek için özel bir komut olan **break;** komutu kullanılır. Standard kullanımda her bir **case** cümle grubunun sonunda **break;** komutu yer alır. Örneğin:

```
c=2; i=0; j=0; k=0; l=0; m=0; n=0;
switch (c) {
   case 1: i=1; break;
   case 2: j=2; k=3;
   case 3: l=4; break;
   case 4: m=5;break;
   default: n=6;
}
```

bu program parçası çalıştığında j=2;k=3;l=4;break; cümleleri gerçekleştirileceğinden j, k, ve l değişkenlerinin değerleri değişecek, i, m, ve n değişkenleri 0 olarak kalacaktır.
C dili döngüler açısından da çok zengindir. 3 çeşit döngünün sözdizimleri şu şekildedir:

```
While (koşul)
   cümle;
do    cümle;
while (koşul);
for (başlangıç-cümleleri;koşul;döngü-sonu-cümleleri)
   cümle;
```

İlk döngü cümlesinde koşul DOĞRU olduğu sürece cümle gerçekleştirilir. İkincisinde ise cümle ilk kez her koşulda gerçekleştirildikten sonra koşul'un sonuna bakılarak bundan sonra cümle koşul DOĞRU olduğu sürece gerçekleştirilir. Meselâ,

```
i=0;
while (i<5) i++;
```

program parçasında i++ işlemi 5 defa yapıldıktan sonra i'nin değeri 5 olduğunda döngü tamamlanır. Ya da:

```
i=6;
do i++; while (i<5);
```

program parçasında i'nin başlangıç değeri bile koşulu sağlamadığı halde i++ işlemi bir kez gerçekleştirilir ve hiç bir döngü yapılmadan çıkılırç i'nin değeri ise 7 olur.

Üçüncü döngü cümlesi, yani for cümlesi birçok değişik şekillerde kullanılabilir. Bu cümlenin while cümleleri cinsinden yazılımı şöyledir:

```
başlangıç-cümleleri;
while (koşul)
{
   cümle;
   döngü-sonu-cümleleri;
}
```

Genellikle sayaçlar aracılığıyla önceden belirlenen sayıda döngüler yapmak için kullanılan bu döngü cümlesiç başlangıç ve döngü sonunda virgül (,) ile ayrılmış birden çok cümle bulunabildiği için ve koşu ile döngünün sürdürülmesi kontrol edilebildiği için çok karışık döngüler yapmak için kullanılabilir. Meselâ,

```
k=0;
for (i=0,j=10;i<j;i++,j--) k+=i+j;
```

program parçasında k+=i+j cümlesi 5 defa gerçekleştirilecek (i=0, j=10; i=1, j=9; i=2, j=8; i=3, j= 7; i=4, j=6) ve k'nın değeri 50 olacaktır. En son döngüden sonra i=5 ve j=5 olduğunda i<j koşulu YANLIŞ olacağı için k+=i+j cümlesi gerçekleştirilmeyecektir.

Tüm koşullu cümleler ve döngü cümlelerinde sadece bir tek cümle yerine bir grup cümle (blok) tanımlanmak istenirse yukarıda for cümlesinin while cümlesi ile tanımlanmasında olduğu gibi küme parantezleri kullanılarak ({}) birçok cümle ; ile ayrılarak yazılabilir. Herhangi bu tür bir bloktan birkaç cümle gerçekleştirilip dışarı çıkılmak istenirse switch cümlesinde olduğu gibi break komutu kullanılabilir. Ayrıca döngü cümlelerinde bir blok içerisinde birkaç cümleden sonra bir sonraki döngüye atlamak için continue komutu kullanılır.

1.2.6. Girdi-Çıktı İşlemleri:

C dilinde girdi-çıktı işlemleri birçok değişik biçimde gerçekleştirilebilir. En basit girdi-çıktı işlemlerinin sözdizimi (syntax) aşağıdaki gibidir:

Girdi işlemi: *scanf("format", &değişken [, &değişken]) ;*
Çıktı işlemi: *printf("format", ifade [, ifade]) ;*

Şimdilik sistemde hazır bulunan (built-in) basit veri yapıları için geçerli olan yukarıdaki girdi-çıktı tanımlarını kullanacağız. Bu kullanımda girdi için kullanılan değişkenlerin ve çıktı işlemleri için kullanılan ifadelerin sonuçlarının veri türü basit veri türü olmalıdır.Kullanıcının tanımlayabileceği veri yapıları ve daha karmaşık veri türleri için girdi-çıktı işlemlerinin nasıl

yapılabileceği bu noktada anlatılmamıştır. Her türlü girdi/çıktı işlemi için format belirtilmesi gerekmektedir. Bunlardan bazıları şöyledir: tamsayılar için (int) %d, real sayılar için (float) %f, karakterler için (char) %c, karakter dizileri için (string) %s. Aşağıdaki program örneklerinde bu formatların kullanımı görülebilir. Ayrıca satır sonu için olduğu gibi (\n) birçok özel karakter de çıktılarda kullanılabilir. Çıktı formatlarında % ile başlayan format komutları ve \ ile başlayan özel karakterler dışında yazılan her şey ekrana basılacaktır. Bu en basit girdi/çıktı işlemleri klavyeden veri alıp ekrana yazdırmak için kullanılır. Bunun dışındaki kütük işlemleri için kullanılan yapıları referans kitaplarda bulabilirsiniz.

1.3. ÖRNEKLER

Tabloda yer alan C programlarını ASCII dosyası halinde, 1.c, 2.c, 3.c bağlantılarını kullanarak elde edebilirsiniz.
Pembe ile işaretlenen satırlar kütüphaneden programa ithal etme işlevini yapmaktadır. Mavi ile işaretlenen satırlar ise ana programın başlarını göstermektedir. Yeşil ve mor ile işaretlenen satırlar girdi-çıktı işlemlerinin yapıldığı satırlarıdır. Programlarda yer alan ve // ile başlayan satırlar, ve diğer satırlardaki // sembollerini takip eden kısımlar ise sadece açıklama için konulmuş olup, programla ilgisi olmayan comment satırlarıdır.

Örnek 1:
Bu konsol uygulama programı, iki tamsayıyı klavyeden okur ve bu iki sayının toplamını ve birinci ile ikinci sayı arasındaki farkı bularak ekrana bastırır.

```
#include <stdio.h>
void main()
 {
      int i1,i2;
      int sum,diff;
                  printf("1. tamsayı ?");
                  scanf("%d",&i1);
                  printf("2. tamsayı ?");
                  scanf("%d",&i2);
sum = i1 + i2;
diff = i1 - i2;
printf("Toplam: %d Fark: %d",sum,diff) ;
}
```

Örnek 2:

```
#include <stdio.h>
void main()
int num1,num2,num3;
int minValue;
```

```
    printf("Birinci tamsayı ?");
    scanf("%d",&num1);
    printf("Ikinci tamsayı ?");
    scanf("%d",&num2);
    printf("Üçüncü tamsayı ?");
    scanf("%d",&num3);

if (num1<num2)
minValue = num1;
else
minValue = num2;
if (num3<minValue)
minValue = num3;
printf("En küçük değer: %d", minValue);
}
```

Örnek 3

```
#include <stdio.h>
int comb(int,int);
int perm(int,int);
int fact(int);
        void main()
{
      int n, r, combVal, permVal;
                printf("n değeri ? ");
                scanf("%d",&n);
                printf("r değeri ? ");
                scanf("%d",&r);
          combVal = comb(n,r);
          permVal = perm(n,r);
          printf("Kombinasyon C(n,r): %d\n",combVal);
          printf("Permütasyon P(n,r): %d\n", permVal);
}
int comb(int n, int r) {
return fact(n)/(fact(r)*fact(n-r));
}
int perm(int n, int r) {
        return fact(n)/fact(n-r);
}
int fact(int n) {
          int i,val;
          val = 1;
          for (i=1; i<=n; i=i+1)
          val = val*i;
          return val;
```

```
}
```

1.4. KULLANILACAK C DERLEYİCİSİNE İLİŞKİN AÇIKLAMA

Bildiğiniz gibi dersimizde çeşitli uygulamaları gerçekleştirmek için C proglamlama dilini kullanacağız. Yapılan gerçekleştirimler arasında belirli bir standardı korumak amacıyla tüm öğrencilerden aynı derleyiciyi kullanmalarını beklemekteyiz. Turbo C++ 3.0 derleyicisi oldukça basit ve C'nin tüm özelliklerine sahip ancak profesyonel kullanımlar için yetersiz (Windows uygulamaları için yetersiz) bir derleyicidir. Hafızada da çok az yer kaplamaktadır. Bu nedenle biz bu derleyiciyi kullanacağız.

2 - GÖSTERGEÇLER (POİNTERS) VE DİZİLER (ARRAYS)

2.1. GÖSTERGEÇ (POİNTER) VERİ TÜRÜ

Göstergeçler, aslında diğer temel veri türleri gibi bir veri türüdür. Ancak onlardan farklı olarak göstergeçler, normalde kullanıcının programını geliştirirken ihtiyaç duyacağı verilerin tanımlanması için değil de bu verilerin adreslerinin belirlenebilmesi için kullanılırlar. Göstergeç değişkeni, hafızadaki başka türden objelerin adreslerinin tutularak bu objelere dolaylı bir şekilde ulaşılmasına olanak sağlar. Göstergeçlerin en tipik kullanımı bağlantılı liste (linked list) veri yapısının oluşturulmasında ve bunun üzerinde işlemlerin yapılmasında görülebilir.

Her göstergeç değişkeni bir veri türü ile ilişkilendirilmek zorundadır ve ancak bu türden verilerin saklandığı adresleri göstermek amacıyla kullanılmalıdır. Bunun dışındaki kullanımlar programın çalışması sırasında beklenmeyen sonuçlar alınmasına neden olabilir. Örneğin int türünden nesneleri göstermek amacıyla tanımlanan bir göstergeç değişkenine float türünden bir nesnenin adresi atanır ve bunun üzerinde bir işlem gerçekleştirilirse, dolaylı olarak kullanılan bu verinin gerçek değeri ile tamamen ilgisiz bir değere ulaşılacaktır.

Bu nedenle göstergeç değişkenleri kullanılırken çok dikkatli olunmalıdır, çünkü bu tür yanlışlıkların derleme sırasında tesbiti olanaksızdır.

Göstergeç değişkeni tanımlandığında sadece dolaylı olarak erişilip gösterilecek verinin adresinin tutulabilecegi bir hafıza oluşturulur. Asıl verinin saklanması için gerekli olan hafızanın tanımlanması göstergeç değişkeninin tanımlanmasından tamamen bağımsız bir işlemdir. Bu durum da genellikle hatalara neden olabilmektedir. Asıl veriyi saklamak için bir değişken ve hafıza tanımlamadan sadece bir göstergeç değişkeni tanımlayıp, bu değişkenle dolaylı olarak bir veri oluşturulup saklanmaya ve bu veriye ulaşılmaya çalışılması tüm programın çalışmasını bozacak sonuçlar çıkarabilir.

Göstergeç değişkeninin tanımlanması diğer veri türünden değişkenlerin tanımlanması ile çok benzerlikler göstermektedir. Tek farkı değişken adından önce * sembolünün yer almasıdır. Örneğin int türünden verileri saklamak için i adlı bir değişken

int i;

şeklinde tanımlanırken, int türünden verilere göstergeç ise

*int *pi;*

şeklinde tanımlanabilir. Bu tanımlardan sonra i değişkeni int türünde herhangi bir verinin saklanabileceği ve erişilebileceği bir hafızayı tanımlarken, pi değişkeni ise int türünden verilerin saklandığı hafızaların adreslerinin saklanabildiği bir hafızayı tanımlamıştır. Göstergeç tanımlamalarını yaparken

dikkat edilmesi gereken bir nokta * sembolünün aslında bir dereferencing operatörü olup tanımlama sırasında tüm göstergeç değişkenlerinden önce yer alması gerektiğidir. Meselâ,

int *pj, pk;

tanımlaması ile aslında bir adet int türünden verilere göstergeç değişkeni (pj) ve bir adet int değişkeni (pk) tanımlanacaktır.
Göstergeç değişkenleri aslında diğer veri türlerinin saklandığı değişkenlerin hafıza adreslerini saklamak ve bu sayede dolaylı olarak bu değişkenlerdeki veriler üzerinde işlem yapmak amacıyla geliştirilmişlerdir. Bu nedenle adresleri de elde edebilmeyi sağlayacak bir takım araçlara gereksinimimiz vardır. C/C++ dilinde bunun için kullanılan operatör & operatörüdür. Bu operatör herhangi bir değişkenin önünde kullanıldığında o değişkenin adresini döndürür. Örneğin yukarıda tanımladığımız int değişkeni i'nin adresini

&i

ifadesi ile elde edebiliriz. Bu operatörü kullanarak istediğimiz değişkenin adresini elde edebiliriz. Normal tanımlanmış değişkenlerin adreslerini saklamak dışında temel olarak göstergeçler, tanımlayıcı bir adı (identifier) olmayan, yani diğer değişkenler gibi tanımlanmayan, kullanıcı tarafından programın çalışması sırasında sadece özel fonksiyon ya da operatörler kullanılarak yaratılan ve yok edilebilen ve varlığı kullanıcının direk kontrolünde olan değişkenlerin (daha doğrusu hafızadaki yerler) adreslerini saklamak için de kullanılır. Bunların örnek uygulamalarını veri yapıları içerisinde göreceğiz.

Göstergeç veri türü üzerinde yapılabilecek işlemler için aşağıdaki 3 adet temel operatör bulunmaktadır:

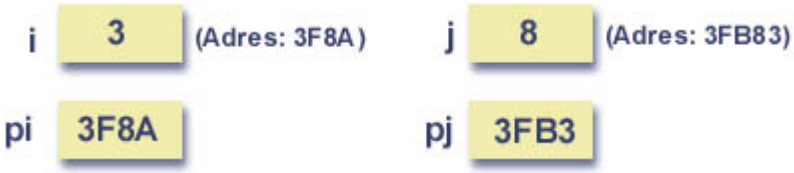
i**. * (dereferencing operatörü) –** İki adet kullanımı mevcuttur: Göstergeç tanımlamak için, tanımlama sırasında göstergeç değişkeninin önünde yer alır; Normal program ifadesinde göstergeç değişkeninin önünde yer aldığında, göstergeçin gösterdiği adresteki veriye dolaylı olarak ulaşılmasını sağlar.

ii. **& (adres operatörü) –** Herhangi bir değişkenin önünde kullanıldığında o değişkenin adresini döndürür.

iii. **= (atama operatörü) –** Göstergeçlerle ilgili olarak, bu operatör bir göstergeç değişkenindeki değeri (adresi) başka bir göstergeçe ya da bir adresi (adres operatörü kullanılarak elde edilmiş) bir göstergeçe atamakta kullanılır. Meselâ,

(1) int i, j, *pi, *pj;

(2) i = 3;
(3) j = i + 5;
(4) pi = &i;

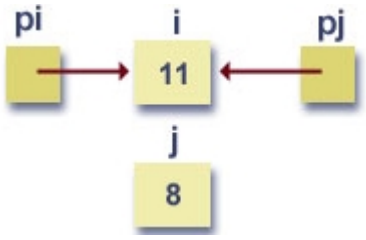


(5) printf("%d",*pi);
çıktı: 3
(6) *pi = j + *pi;
(7) pj = pi;

(8) printf("%d",*pj);
çıktı:11

Yukarıdaki program parçasında (1) no'lu satırda 2 adet int değişken (i ve j) ve 2 adet de int değişkenine göstergeç (pi ve pj) tanımlanmaktadır. (2) ve (3) no'lu satırlar sıradan int değişkenleri üzerinde atama işlemlerini içermektedir. (4) no'lu satırda ise pi adlı göstergece i adlı int değişkeninin adresi yüklenmekte. Bu andan itibaren pi adlı göstergeç üzerinden i değişkenin değerine dolaylı olarak ulaşmak olanaklı olmaktadır. (5) no'lu satırda i değişkeninin değerine dolaylı olarak ulaşılarak çıktıya gönderilmektedir. (6) no'lu satırda ise gene pi göstergeci üzerinden dolaylı olarak hem i değişkeninin değerine ulaşılmakta hem de değiştirilmektedir. Atama işleminin sol tarafında yer alan *pi işlemi - aslında orada i değişkeninin bulunması ile aynı sonucu doğurarak - dolaylı olarak i değişkeninin kendisine (içerdiği değere değil de değişkenin kendisine) ulaşılmasını sağlamaktadır. Aslında her türlü değişken için atama operatörü benzer bir etki oluşturmaktadır.
Değişken = operatörünün sağ tarafında ise değişkenin içerdiği değere ulaşılırken, operatörün sol tarafında iken değişkenin kendisine ulaşılarak = operatörü ile değişiklik yapılması sağlanmaktadır. (7) no'lu satır iki göstergeç arasında bir atama yapılmasını sağlamakta ve bu sayede (8) no'lu satırda gene i değişkeninin içerisindeki değere bu sefer pj göstergeci ile dolaylı olarak ulaşılabilmektedir.
Genellikle değişkenlerde saklanan adres değerleri (yani göstergeçlerin sakladıkları değerler) ve değişkenlerin adresleri programın çalışmasının

anlaşılması için gerekli olmayıp karışıklıklara da neden olabilmektedir. Bu nedenle gösterimlerde adres değerleri kullanmak yerine, bir göstergeç değişkeninde saklanan adres ve göstergecin gösterdiği bu adreste bulunan değişken (hafıza) ilişkisi adresten hafızaya doğru bir ok şeklinde gösterilmektedir. Zaten adres değerlerini yazmak çok anlamsızdır, çünkü gerçek adres değerleri hiçbir zaman kullanılmaz, sadece ilişkiler program açısından önemlidir ve ayrıca aynı bilgisayarda dahi programın her çalışmasında tamamen farklı adresler kullanılabilir. Bu yöntemle yukarıdaki programın son halini aşağıdaki gibi gösterebiliriz:



2.1.1. Göstergeç Aritmetiği

Göstergecin sakladığı adres değeri de arttırılıp azaltılabilir. Meselâ pi = pi + 3; geçerli bir cümle olup bu işlem sonucunda pi göstergecinin sakladığı adres değeri 6 byte artar. Bu göstergecin türü int olduğu ve her bir int verisi hafızada 2 byte kapladığı için pi'nın 3 arttırılmasıyla aslında 3 ilerideki tamsayıya ulaşılabilmesi amaçlanmakta ve pi'nın sakladığı adres değeri 2*3 = 6 byte artmaktadır. Yani göstergeç aritmetik işlemleriyle yapılan arttırma/azaltma miktarları göstergecin gösterdiği veri türüne bağlı olarak o türdeki nesnelerin hafızada kaç byte yer kapladığına bağlı olarak belirlenir.

2.1.2. Dizgi Göstergeçleri (String Pointers)

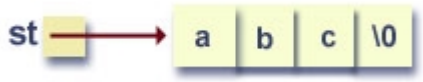
Dizgi (string) veri türünü C dilinde ifade edebilmenin tek yolu karakter (char) türünden bir göstergeç oluşturmaktır. Bütün dizgi işlemleri karaktere göstergeçler üzerinden yapılır.

*char *st;*

tanımlaması genellikle sadece bir karakterlik bir değişkenin adresinin saklanması amacıyla bir göstergeç oluşturulması için değil de (bu amaçla da kullanılabilir) dizgi türünden bir verinin başlangıç adresinin tutulmasını ve böylece bu dizgi üzerinde işlemler yapılabilmesi sağlamak amacıyla yapılır.
Yukarıdaki verdiğimiz örneklerde hep göstergeçlerin gösterdikleri adreslerin daha önceden tanımlanmış olan değişkenler olduğunu gördük. Halbuki dizgi veri türünü oluşturmak için bu şekilde bir değişken tanımlanması uygun değildir. Çünkü C dili direk dizgi veri türünü sağlamayıp sadece karakter

(char) veri türünü sağlamakta ve dizgiyi tanımlamak için bir karakter serisi oluşturulması gerekmektedir. C dilinde dizgiler çift tırnak içinde karakter serileri seklinde gösterilir.
Meselâ "abc" dizgisi aslında 4 karakterden oluşan bir dizgidir. (karakterler ise tek tırnak içinde gösterilir) ve 'a', 'b', 'c', ve '\0' (boş karakter) karakterlerinden oluşur. Her dizginin sonunda mutlaka bir boş karakter yer alır. Aslında bu "abc" dizgisini oluşturmak için hafızada 4 karakterlik yer ayrılması (ya da 4 char türünde değişken oluşturulması) gereklidir. Ancak bu yeri char türünde değişkenler tanımlayarak önceden açmak (tek tek değişkenler şeklinde) olanaklı olmadığından dizginin tanımlanması sırasında atama işlemi yaparak gerekli hafızanın önceden alınması ve bu dizginin değeri ile doldurulması sağlanabilir.
Bu durumda da ad verilmemiş değişkenlerin yaratıldığı düşünülebilir, çünkü bu ayrılmış hafızadaki bilgilerin, diğer değişkenler gibi, okunması ya da değiştirilmesi de olanaklıdır.

*char *st ="abc";*



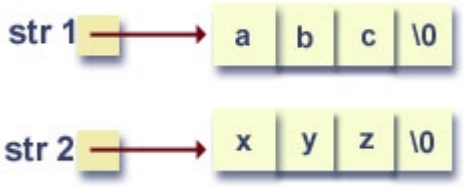
tanımı ile st göstergeç tarafından başlangıç adresi gösterilen bir "abc" dizgisi oluşturulur. Bu oluşturulan dizgi ve onun için ayrılan hafızadaki veriler daha sonra da değiştirilebilir. Ancak burada dikkat edilmesi gereken nokta bu tanımlama ile ancak 4 karakterlik yerin ayrıldığı (sadece 3 karaktere kadar olan dizgilerin saklanabileceği) bu nedenle bu hafızada daha uzun dizgiler saklanmaya çalışılması durumunda bunun tüm programda hatalara neden olabilecek sonuçlar doğurabileceğidir. Bu problemin çözümü ise dinamik olarak hafızada yer ayırmaya yarayan işlev ve operatörlerin kullanılmasıdır. Bu konulara daha sonra detaylı olarak değineceğiz.
Dizgi veri türü üzerinde çeşitli işlemleri yapabilmek amacıyla hazır birçok fonksiyon tanımlanmıştır. Bu fonksiyonların kullanılabilmesi için standart "string.h" kütüğünün programa yüklenmesi gerekmektedir. Bu fonksiyonların bazıları şunlardır:

// str1 ile gösterilen dizgiyi str2 ile gösterilen yere kopyala

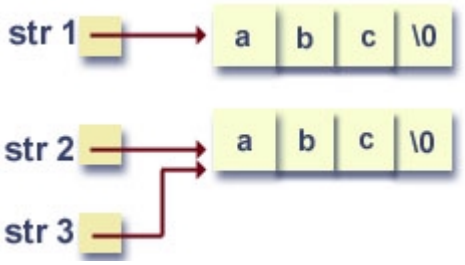
*char *strcpy (char *str2, char * str1) ;*

Bu fonksiyonun iki parametresi de dizgi olup (ya da char *, yani dizginin başlangıç karakterini gösteren bir göstergeç) sonuçta da str2'nin adresini döndürür.

char *str1 = "abc", *str2 ="xyz";



*char *strcpy (char *str2, char *str1);*
*char *str3 = strcpy (str2, str1);*



Yukarıdaki program parçası çalıştırıldığında str1 ve str2 farklı adreslerde aynı "abc" dizgisini gösterecek, yani str1'in gösterdiği dizgi str2'nin gösterdiği adrese kopyalanmış olacaktır. Ayrıca str3 de str2'nin gösterdiği adresi gösterecektir.

// İki stringi karşılaştır
*int strcmp (char *str1, char *str2);*

Bu fonksiyonun iki parametresi de yukarıdaki fonksiyon gibi iki adet dizgi olup fonksiyonun çağırılması sonucunda bu dizgiler karşılaştırılıp eğer ikisi birbirine eşitse 0, değilse 0'dan farklı bir tamsayı döndürür.

// stringin uzunluğu
*int strlen (char *str1);*

Bu fonksiyon ise sadece bir adet dizgi parametre alıp sonuçta dizginin boyu olan bir adet tamsayı döndürür. Burada dikkat edilmesi gereken nokta dizginin sonundaki '\0' (bos) karakterin dizginin boyuna dahil edilmediğidir. Ayrıca dizginin sonunda boş karakter yoksa hatalı ve genellikle büyük bir tamsayı döndürülecektir. Bunun nedeni sistemin ilk boş karaktere kadar ilerlemeye çalışarak o karaktere kadar her şeyin bir dizgi tanımladığını varsaymasıdır.

2.2.Dizi (İndisli) Veri Türü:
Dizi, çok sayıda aynı tür nesnenin bir araya gelmesinden oluşan bir veri türüdür. Bu veri türünde tek tek nesnelerin adlandırılması yerine tüm nesneler

hep birlikte dizi değişkeni ile adlandırılır. Ayrıca her bir nesne de dizinin içindeki pozisyonuna göre bir indeksle tanımlanabilir. Örneğin int i; tek başına sadece bir adet tamsayı (int) nesnesi oluştururken int ia[10]; ise 10 adet (ilk eleman her zaman 0'dan başlamak üzere bu örnek için ia[0]'dan ia[9]'a kadar) int nesnesi tanımlar.

2.2.1. Dizi tanımlama ve her bir elemanına ulaşma sözdizimi

<veri türü> < dizi adı>[<büyüklük>];
< dizi adı>[<indeks>]

Meselâ,

int i; int ia[10];
for (i=0;i<10;i++) ia[i] = i+1;

program parçasının çalışması sonucunda ia dizisinin her bir elemanına 1'den 10'a kadar değerler atanır.
Aslında dizinin tanımlanması sırasında da değer atanması mümkündür.
Aşağıdaki tanımlama yukarıdaki program parçası ile aynı sonucu oluşturur.

int ia[10] = {1,2,3,4,5,6,7,8,9,10};

2.2.2. Çok Boyutlu Diziler (Arraylar)

Çok boyutlu dizi tanımlanması ve erişimi de tek boyutlu dizi tanımına benzerdir. Meselâ iki boyutlu bir dizi tanımı ve elemanlarına ulaşımı aşağıdaki gibi yapılır:

inta ia[4][3];
ia[0][0] = 5; ia[3][2]=9; ....

2.2.3. Dizi ve Göstergeç (Pointer) Türleri Arasındaki İlişkiler

Dizi tanımlama işlemi sırasında dizinin adını ifade eden sembol aslında dizinin ilk elemanının hafızadaki adresini gösteren bir göstergeci ifade eder. Yani < dizi adı> ile &< dizi adı>[0] ifadeleri birbirlerine eşittir. Ya da başka bir değişle < dizi adı>, aslında dizinin elemanları türünden bir göstergeç olup, bircok açıdan diğer göstergeçler gibi kullanılabilir. Diğer göstergeçlerden tek farkı <dizi adı>'nın sabit bir göstergeç olup değerinin değiştirilmesinin olanaksız olmasıdır. Örneğin <dizi adı> atama işleminin sol tarafında yer alamaz ya da değerini değiştirebilecek diğer işlemlere tabi tutulamaz.

Dizi tanımı <büyüklük> ile belirlenen miktarda dizinin <veri türü>'nden değişken (adları <dizi adı> ve <indeks> değeri ile belirlenen) tanımlanmasını sağlar. Bu değişkenlere de aslında iki türlü erişilebilir; birincisi yukarıda bahsettiğimiz normal dizi ifadesi ile, diğeri ise < dizi adı>'nı göstergeç olarak kullanıp, göstergeç aritmetiği yardımıyla.

```
char tst[10];
char *str = tst;
tst[0] = 'a'; tst[1] = 'b';
printf("%d %d\n",*(tst+1),tst[1]);
```

Yukarıdaki programda iki çıktı da 'b' (yani aynı) olacaktır. Bunun nedeni göstergeç aritmetiği kullanarak yazdığımız (tst+1) ifadesinin tst'nin başlangıcından 1 sonraki nesneye ait adrese (veri türü char olduğu için 1 byte sonrası) ulaşması ve bu ifadenin önüne gelen * operatörü ile de bu değişkendeki değeri elde etmemizidir. Bu da aslında tst dizisinin tst[1] ile gösterilen 2. nesnesidir (ilk nesnenin tst[0] ile gösterildiğine dikkat edin). Ya da başka bir deyişle (tst+1) ifadesi ile &(tst[1]) ifadeleri aynı adresleri gösteren tamamen eşit ifadelerdir. Dizinin adı dizinin 1. nesnesinin adresini verdiği için (*tst) ifadesi ile tst[0] ifadeleri de aynı nesneye ulaşılmasını sağlar.
Dizi tanımlama işlemi aslında hafızada bir bloğun ayrılmasını (allocation) sağlar. Bu blok tanımı sabit olduğu için dizi adı ile tanımlanan göstergeç değişkeni sabit veri türü gibi kabul edilir ve değiştirilemez (örneğin tst++ illegal bir işlemdir). Dizi tanımlama işlemi ile sabit bir hafıza tanımlanabildiğinden ve diziler ile göstergeçler ilgili kavramlar olduğu için dizgi (string) işlemlerinde de dizi ve göstergeç kavramları sıkça ilişkili olarak kullanılırlar. Meselâ,

```
char *str1 = "abc";
char str2[] = "abc";
char str3[4] = {'a','b','c','\0'};
```

tanımları birbirine eşittir ve her biri 4 karakterlik dizgi (string) olarak kullanılabilecek bir hafıza ayrılıp bu hafızaya "abc" değerinin atanmasını sağlarlar. Ayrıca maksimum uzunluğu belli sabit bir hafızayı da dizi olarak tanımlayıp en fazla o uzunluktaki dizgileri (strings) saklamak için kullanmak olanaklıdır. Örneğin char str4[65] tanımı ile oluşturulan dizi en fazla 64 karakterlik dizgi (strings) saklamak için kullanılabilir. Bu dizi ile oluşturulan hafızaya

```
strcpy(str4,"abcde");
```

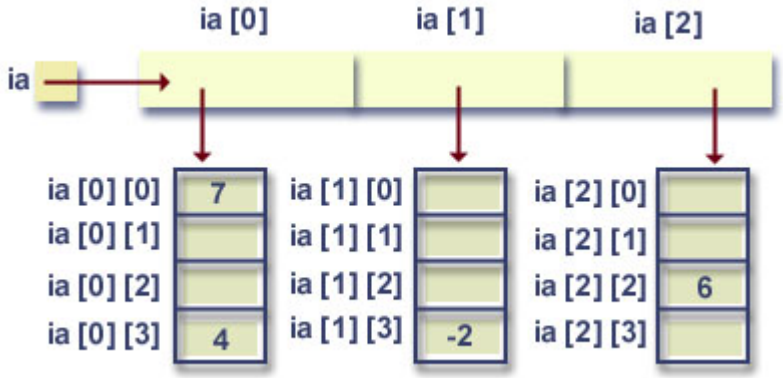
fonksiyonunu çağırarak, başlangıç elemanının adresi str4 olmak üzere "abcde" dizgisi (string) atanabilir.

Bu dizginin (string) her bir elemanına (karakterine) de str4[0], str4[1],... ifadeleri ile ulaşılabilir. Yukarıdaki kopyalama fonksiyonunun çalışması sonucunda str4 ile gösterilen dizgide (string) (ya da str4 dizisinde) aşağıdaki değerler olur:

str4[0] : 'a', str4[1] : 'b', str4[2] : 'c', str4[3] : 'd', str4[4] : 'e', str4[5] : '\0' (dizginin sonu boş karakter)

Tek boyutlu dizgilerde, dizgi adı aslında bir göstergeç olduğu için çok boyutlu dizgilerde da aslında boyut sayısının bir eksiği kadar boyutta (son boyut hariç) göstergeç dizileri tanımlamaktadır. Son boyut ise gerçek değişkenler için gerekli hafızayı ayırmaktadır. Örneğin int ia[3][4]; aslında ia adında bir göstergece göstergeç (int **) ve ia[0], ia[1], ve ia[2] ile de 3 adet göstergeç (int *) tanımlayarak 3 adet tek boyutlu int dizisi oluşturmaktadır. Yani ia[0] tek boyutlu bir dizi olarak kullanılabilir ve bu dizinin elemanlarına ia[0][1] vs şeklinde ulaşılabilir.
Aşağıda 2 boyutlu bir dizgi örneği verilmektedir:

int ia [3] [4];
ia [0] [0] = 7;
ia [0] [3] = 4;
ia [2] [2] = 6;
ia [1] [3] = -2;
...



3 - PARAMETRE GEÇİŞİ, BOŞ BELLEK ALIMI

3.1. İşlevler (Functions) ve Parametre Geçirme

İşlevlerle ilgili olarak iki temel işlem bulunmaktadır:

İşlevin tanımlanması – parametrelere biçimsel (formal) parametre denir, sadece bir tanımlama ifade eder.

İşlevin çağırılması – parametrelere gerçek (actual) parametre denir ve tanımlanan işlevin gerçek parametre değerleri ile çalıştırılmasını sağlar.

İşlevler çağırılırken gerçek parametre değerlerinin biçimsel (formal) parametre değerleri ile eşlendirilmesi işlemi için değişik teknikler uygulanabilir. Bu konu çerçevesinde C dilindeki bu teknikleri inceleyeceğiz.

C dilindeki işlev tanımları

döndürülen-tür işlev-adı(parametre-listesi)

Parametre listesinde ise her bir parametre

parametre-türü parametre-adı

şeklinde tanımlanabilir.

3.1.1. Değerle Çağırma (Call By Value)

Örneğin 1. parametre olarak belirlenen bir tamsayının 2. parametre olarak belirlenen herhangi bir pozitif tamsayı üssünü almak için aşağıdaki işlevi tanımlayalım:

```
int power (int n, int k)
{
        int t=1;
        for(;k>0;k--)
        t=t*n;
        return t;
}
```

Bu işlev iki adet int parametresi almakta ve sonuçta da bir adet int parametre döndürmektedir. Aşağıdaki ana programda bu işlevi çağırarak 4 üssü 3'ü hesaplamak istersek:

```
main ()
{
        int i=4, j=3, p=0 ;
        p = power(i,j);
        printf("%d ussu %d = %d",i,j,p);
}
```

p=power (i,j) komutu işletildiğinde i ve j değişkenlerinin değerleri n ve k'ya kopyalanmaktadır.
programın çıktısı “4 ussu 3 = 64”, yani istediğimiz gibi, olacaktır. Ancak yukarıda tanımlanan power işlevinde önce ana programdan j’nin değeri (3) 2. parametre olan k’ya geçirilmekte, sonra işlevin çalışması sırasında da k’nın değeri 0’a inmektedir. Burada değerle çağırma tekniği kullanıldığı için ana programdaki j power işlevinden etkilenmemektedir. Parametre geçirme sırasında k değişkeni için yeni bir hafıza yaratılmakta ve j’nin değeri o hafızaya atanmaktadır. power işlevi çalışırken, k tanımlayıcısı bu değişkeni kullanmakta, işlev sona erdiğinde de bu değişken yok olmaktadır. Bu arada ana programdaki j değişkeni ise k’dan hiç etkilenmemektedir.

İşlevler sadece bir tane değer döndürebilirler. Döndürülme işlemi return operatörü ile yapılır. Döndürülen değerin türü işlevin tanımında belirtilen tür olmalıdır, yoksa programın çalışması sırasında beklenmedik hatalar oluşabilir. Değerle çağırma tekniği ile işlevin değer döndürmesi işlemi sadece işlev tanımı sırasında belirlenen türde bir veriyi geri döndürdüğü için işlevin çağırılması sırasında bu değerin alınıp aynı türde bir değişkene yüklenmesi de mümkündür. Bu nedenle işlev ifadelerde bu türde değer beklenen herhangi bir yerde çağırılabilir, ve sonuçta döndürdüğü değer kullanılabilir. Gene yukarıdaki örnekte power işlevi çağırılıp elde edilen değer atama işlemi ile p değişkenine atanabilmektedir. Bu örnekte hem parametreler hem de işlev sonucu döndürme işlemi değerle çağırma şeklinde adlandırılabilir. Parametreler aracılığı ile çağırılan ortamdan işleve, döndürülme aracılığıyla da işlevden çağırılan ortama veri taşınmaktadır ve bütün bu veri taşıma işlemleri verilerin taşındığı ortamlarda atamalar şeklinde yapılmaktadır ve iki ortam arasında global değişkenler hariç bir değişken paylaşımı olmadığından direk olarak ortamların birbirlerini etkilememeleri amaçlanmıştır.

3.1.2. Göstergeç Parametre (Referansla Çağırmanın Simulasyonu):

C dilinde * operatörü ile ifade edilerek göstergeç değişkenleri aracılığı ile referansla çağırma işlemi simule edilmektedir.

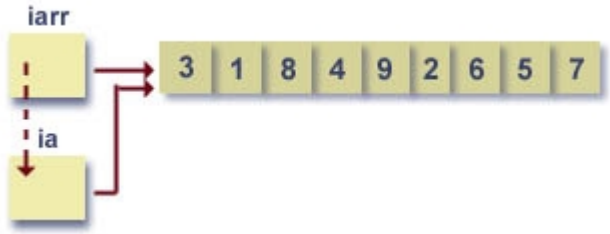
Değerle çağırma tekniği, eğer çağırılan ortamdan işleve yollanacak verinin miktarı çok fazla ise, bu veriyi işleve yollarken hem çok fazla atama işlemi yaparak zaman açısından, hem de tüm bu verinin işlev içerisinde bir kopyasının daha tutulması zorunluluğu ortaya çıktığı için bellek açısından çok verimsiz olmaktadır. Bu nedenle verinin kopyalanmasına gerek kalmadan işleve taşınmasını sağlamak amacıyla sadece verinin başlangıç adresini gösteren göstergecin kopyalanması ile referansla çağırma işlemi simule edilebilir. Özellikle dizileri işlevlere geçirmek için bu zorunlu bir yöntemdir. Çünkü tüm dizinin kopyalandığı bir atama işlemi tanımlanmadığı için dizi adları arasındaki atama ile aslında diziyle saklanan verinin başlangıç adresi kopyalanmış olmaktadır. C dilinde parametre olarak geçirilen bir (pozitif) tamsayı dizisinin en büyük elemanını bulup döndüren bir işlev (2. parametre dizinin eleman sayısını vermektedir) aşağıda tanımlanmıştır:

```
int max(int ia[], int l)
{
        int i, t = 0;
        for (i=0;i>l;i++)
        if (ia[i]>t) t=ia[i];
        return t;
}
```

Böyle bir işlevi ana programdan şu şekilde çağırabiliriz:

```
main()
{
        int s=9, iarr[9]={3,1,8,4,9,2,6,5,7};
        printf("%d",max(iarr,s));
}
```

Gerçek parametre olan iarr dizininin biçimsel parametre olan ia dizinine geçirilmesi ile aşağıdaki yapı oluşur:



Aslında max işlevinin tanımlanması sırasında dizi parametresi int ia[] şeklinde olduğu gibi sadece int* ya da gerçek parametre ile aynı olmak şartı ile (yoksa hata olabilir) int ia[9] şeklinde de belirtilebilir.

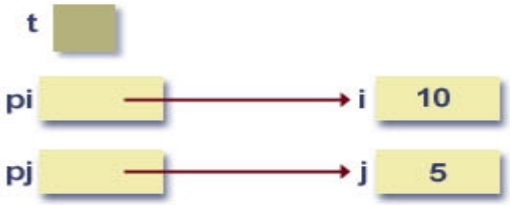
Burada biçimsel parametrenin göstergeç olduğu durumları incelediğimize göre gerçek parametrenin de ya göstergeç (dizilerin göstergeçlerle ilişkisini daha önce görmüştük) ya da göstergecin alabileceği bir değer olan adres olduğu durumları da göz önüne almamız gerekmektedir. Adres (&) operatörü ile C dilinde herhangi bir değişkenin adresine ulaşabileceğimizi biliyoruz. Bütün bu araçları kullanarak iki değişkenin değerlerini yer değiştiren bir işlevi aşağıdaki gibi tanımlayabiliriz:

```
void swap(int *pi, int *pj)
{
int t = *pi;
*pi = *pj;
*pj = t;
}
main()
{
        int i =10, j =5;
        printf("%d %d\n",i,j);
        swap(&i, &j);
```

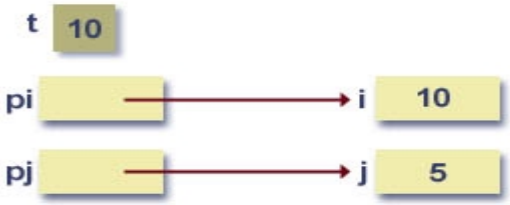
```
        printf("%d %d\n",i,j);
}
```

Ana programda swap(&i,&j) işlemi ile swap işlevinin çağırılması ile aşağıdaki işlemler gerçekleşmektedir:

1. swap (&i, &j), çağrılınca pi ve pj göstergeçleri i ve j değişkenlerini gösterir hale geliyor.

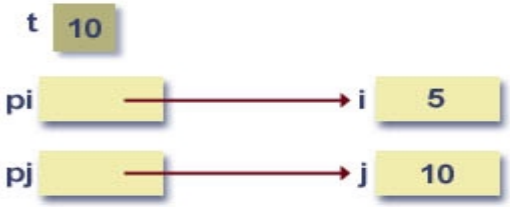


2. 'int t = *pi' işlemiyle t değişkenine i değişkeninin değeri aktarılıyor.



3. *pi = *pj işlemiyle j'nin değeri i'ye aktarılıyor.

4. t'de saklanan i değeri (10) *pj = t işlemiyle j'ye aktarılıyor.



Ana program çalıştırıldığında çıktının ilk satırının "10 5", ikinci satırının ise "5 10" olduğu görülecektir. Ana programda swap işlevi çağırılırken i ve j değişkenlerinin adresleri gerçek parametre olarak yollanmaktadır.

İşlevde ise göstergeçler aracılığı ile i ve j değişkenlerine dolaylı yoldan ulaşılarak yer değiştirme işlemi gerçekleştirilmektedir.

Bir dizinin elemanları üzerinde değişiklikler yapan bir işlevi tanımlamak istersek (örneğin dizinin elemanlarını ters çeviren bir işlev), yukarıdaki örneğin aksine, diziyi parametre olarak geçirirken

gerçek parametre olarak dizinin sadece adını kullanmamız yeterli olacaktır. Çünkü, aslında dizi adları göstergeç olduklarından, işlev çalışırken, biçimsel parametre değişkeni aracılığla direk gerçek parametre olarak kullanılan dizinin elemenlarına ulaşılacak ve bu şekilde de, işlev ile dizide yapılan tüm değişiklikler işlev sonucunda, işlevin çağırıldığı ortama dönüldüğünde, gerçek parametre olan diziye yansımış olacaktır.

İşlevlerin döndürdükleri tür de göstergeç türü olabilir. Bu durumda böyle bir işlev bu türden göstergeçlerin kullanılabileceği yerlerde çağırılabilir ve bu şekilde işlevin döndürdüğü göstergeç de çağıran ortamca kullanılabilir.Bu türden örnekleri ilerki konularda (yapı (struct) konseptini incelerken) göreceğiz. Ayrıca döndürülen tür göstergeç (ya da başka bir deyişle adres) olduğunda endirek olarak adres üzerinden gerçek değişkene de ulaşılabileceği için böyle bir işlev çağırma işlemi atama işleminin sol tarafında da yer alabilir.

Aşağıda bu türden basit bir program örneği verilmektedir.

```
int *f(int *pi)
{  return pi;}
        main()
        {
        int j =10;
        printf("%d\n", j );
        *(f(&j)) = 5; // işlev adres döndürmekte ve adres üzerinden gerçek
        değişkene ulaşılmakta
        printf("%d\n",                    j                    );
        } //Bu programın çıktısı 10 ve 5 olacaktır.
```

3.2. Boş Bellek Alınması (Free Storage Allocation)

İşlevlerin içerisinde (global tanımlamalar hariç) normal değişken tanımlama işlemlerinde (durağan (static) değişkenler hariç), değişkenler tanımın yapıldığı noktandan itibaren yaratılmakta ve tanımın yapıldığı işlev (main de olabilir) çalışmasını tamamladığında ise yok edilmektedirler. Kullanıcı, bu sağlanan ve programın sözdizimine bağlı standart yöntem dışında, yaşam süresini tamamen program akışı ile kendi istediği şekilde kontrol edebildiği isimsiz (anonymous) değişkenler yaratarak, bunları göstergeçler aracılığyla kullanıp istediği zaman da yok edebilir. C dilinde alloc (ve türevleri) ve free işlevleri de vardır. Biz sadece malloc ve free fonksiyonlarını inceleyeceğiz.

Yeni değişken yaratmak amacıyla bellek almak için kullanılan malloc fonksiyonunun sözdizimi şöyledir:

(pointer türü) malloc(miktar);

Bu fonksiyon ayrılan belleğin başlangıç adresini döndürür. Bu nedenle bu fonksiyonun sonucu belirtilen veri türü'nden bir göstergece atanarak alınan bellek kullanılabilir. Kullanıcı istediği zaman başlangıç adresini bir

göstergeçle belirterek aşağıda sözdizimi verilen free fonksiyonu ile bu belleği yok edebilir:

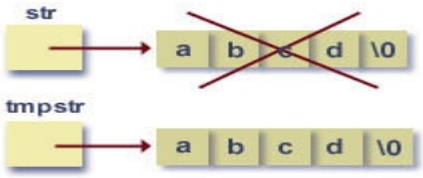
free(pointer adı)

Örneğin parametre olarak gelen bir dizgeyi (string) kopyasının kendisine yapıştırarak 2 katına çıkaran bir işlev ve bunu kullanan ana program aşağıda verilmiştir:

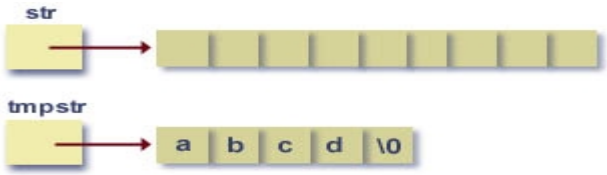
```
void doublestr(char *str)
{
int len = strlen(str);
char tmpstr[len+1];
strcpy(tmpstr,str);
free(str);
str = (char *)malloc(2*len +1);
strcpy(str,tmpstr);
for (int i=len;i ?2*len;i++)
str[i]=tmpstr[i-len];
}
 main()
{
char *str ;
str = (char *) malloc(5) ;
strcpy(str,?abcd?);
printf("%s\n", str);
doublestr(str);
printf("%s\n", str);
}
```

8 double işlevi çağrıldığında önce 'strcpy' işlemiyle 'tmpstr' değişkenine str kopyalanmakta.

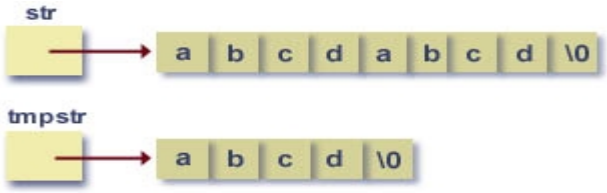
9 free(str), işlemiyle str göstergecinin bellekte gösterdiği alan silinmekte.



10 str = (char *)malloc(2 x len + 1) işlemiyle önceki dizgenin iki katı büyüklüğünde bir yer ayrılmakta



11 strcpy komutu ve for döngüsü ile önceki dizgi iki defa bu yeni bölüme kopyalanmakta



Bu örnekte başlangıçta en fazla 4 karakterlik dizgelerin (strings) saklanabileceği dinamik bir bellek ayrılıp (new ile) str göstergeci aracılığıyla bu bellekte "abcd" dizgesi oluşturulmaktadır. Daha sonra double işlevi çalıştırıldığında ise önce lokal olarak tanımlanmış olan tmpstr dizisine (dizgesine) abcd dizgesi kopyalanmakta ve dinamik olarak yaratılan (malloc ile) bellek yok edilmektedir (free ile). Burada dikkat edilmesi gereken bir nokta ancak malloc ile ayrılan belleklerin free ile yok edilebileceğidir. Ayrıca bu bellek yok edilmese de programımızda hiçbir hata oluşmayacaktı. Ancak str göstergeci aracılığıyla yeniden bir dinamik bellek ayrılınca str artık o belleği göstereceği için önceki belleği gösteren hiçbir aracımız kalmayacak ve önceki belleği kullanmamız olanaksızlaşacaktı. Ama önceki belleği free ile yok etmediğimiz zaman kullanılamayacak şekilde hafızada yer işgal etmeye devam edecek ve ancak program bittiğinde kullanıma açılabilecekti. Bu nedenle dinamik olarak alınan belleğin işi bittiğinde mutlaka free ile yok edilmesi gerekir. Yoksa boş yere bellek işgali oluşur. Bu durum büyük uygulamalarda tüm sistemin çalışmasını engelleyecek kadar büyük problemlere neden olabilir.

Programımızda str ile gösterilen dinamik belleğin free ile yok edilmesinden sonra eskisinin 2 katı uzunluğunda dizgeyi (string) saklayabileceğimiz miktarda yeni bir bellek ayrılıp gene str göstergeci ile gösterilmektedir. Bundan sonraki satırlar daha önce tmpstr aracılığıyla bir kopyasını yaptığımız önceki dizgemizin peş peşe iki kopyasını bu yeni oluşturulan belleğe koymaktadır.

Ana programın çalışması ile elde edeceğimiz çıktıda ilk satırda "abcd" ikinci satırda da "abcdabcd" bulunacaktır.

YAPILAR (STRUCTURES)

4.1. Yapılar (structures)

C dilinde kulanıcının ilgili tanımlamaları bir arada yapabilmesini sağlayan yapı (structure) tanımı bulunmaktadır. Yapı kavramı birbiri ile bağlantılı değişken tanımlarının ortak bir ad altında toplanması olarak tanımlanabilir. Daha kolay anlaşılır bir kod üretmek ve tanımlamalar yapmak amacıyla geliştirilen bu kavram birçok programlama dilinde yer almaktadır. Nesnesel programlama ise, bu kavramı daha ileri götürerek, sadece ilgili değişken tanımlamalarının değil, aynı zamanda bir nesneyi oluşturmak için gerekebilecek tüm işlevlerin bir araya gelmesi ile oluşan eksiksiz bir tanımlamayı aynı ad altında toplama fikrinin programlama dillerinde uygulanmasını sağlamıştır.

Yapı kavramının sözdizimi şöyledir:

struct yapı adı { üye tanımlamaları }

Karakterlerden oluşan bir tampon (buffer) tanımlamak için gerekli değişkenlerin yer aldığı bx adlı yapıyı aşağıdadır: (Bu tanımlamadaki MAXBUF değeri önceden belirlenmiş bir sabittir.)

```
struct bx {
char buf[MAXBUF+1];
int size, front,
rear;
};
```

Bu bx yapısında tamponun kendisini tutmak için buf adlı bir karakter dizisi (array), uzunluğunu tutmak için size adlı bir tamsayı değişkeni ile dizinin içinde tamponda saklanan karakterlerin başlangıç ve sonuç indekslerini belirtmek için de front ve rear adlı iki adet tamsayı değişkeni bulunmaktadır. Bir karakter tamponu oluşturmak için gerekli tüm değişkenlerin yer aldığı bu tanımlamayı kullanarak gerçek tampon nesneleri yaratabiliriz

bx b1, b2;

Bu bx yapısının değişkenlerine . (nokta) operatörü ile ulaşabiliriz:

b1.front = 3;

b2.buf[4] = ‘a;

b2.front = b1.front;

Yukarıdaki tanımlama ile aslında 2 tane yapı değişkeni oluşturulmuştur. Bunların ikisi de bx türündedir ve diğer değişkenler gibi bunların da sembolik isimleri vardır. Bu örnekte tanımlanan iki adet bx yapısı türündeki değişkenin isimleri b1 ve b2'dir. Bu iki değişkeni diğer basit veri türüne ait değişkenlerden ayıran temel özellik bu değişkenlerin içinde aslında birden çok değişik veri türünden parçanın bulunması ve bunların herbirine bx yapısının tanımlanması sırasında belirtilen isimler ile ve nokta operatörü kullanılarak ulaşılabilmesidir.

Diğer veri türlerinde olduğu gibi yapılar üzerinde de göstergeçler tanımlanabilir. Örneğin

*bx *bxp;*

tanımlaması bx türünden değişkenlere (örneğin b1 ve b2'ye) göstergeç olarak kullanılabilir. Bu tanımlamanın bir yapı değişkeni oluşmadığına dolayısı ile nokta opertörünün de bu değişken için bir anlamı olmadığına dikkat ediniz.

Örneğin bu göstergeç ile yukarıda tanımlanan örneğin b1 değişkenine aşağıdaki gibi bir göstergeç oluşturabiliriz:

bxp = &b1;

Bu atama işlemi önceki derslerimizde gördüğümüz göstergeçe atama işleminden farklı değildir. Burada da bxp adlı değişken aslında b1 değişeninin adresine sahiptir. Göstergeçler üzerinden * operatörü aracılığı ile gerçek değişkene ulaşabildiğimizi görmüştük. Burada da (*bxp) ifadesi bize bxp göstergeçinin gösterdiği yapı değişkenini döndürür (yani bu örnekte b1'i). Yapı değişkenlerinin alt parçalarına ise nokta operatörleri ile ulaşabildiğimiz için göstergeç üzerinden yapı değişkenine ulaştıktan sonra nokta operatörü kullanarak bu yapının parçalarına erişebiliriz. Örneğin (*bxp).front aslında yukarıdaki atama işleminden dolayı b1.front ile aynı olacaktır. Yapı göstergeçleri ile yapıların parçalarına ulaşmak için kullanılan ve * ve . operatörleri içeren ifadeler çok karışık olduğu için buna ek basit bir başka sözdizim (syntax) daha bulunmaktadır. Bu sözdizimde * ve . operatörleri yerine daha anlaşılır bir tek -> operatörü vardır. Örneğin (*bxp).front yerine bxp->front kullanılabilir. Bu özel operatör, göstergeçin anlamına da yakın olan ve ok sembolüne benzer bir operatör olduğu için, solda bulunan göstergeç değişeni üzerinden, sağda bulunan ve bu göstergeçin gösterdiği yapının bir parçası olan değişkene ulaşma anlamında kullanılmak üzere tanımlanmıştır.

Göstergeçleri daha önceki konularımızda özellikle işlevlere parametre geçirmek amacıyla kullanmıştık. Buna ek olarak dinamik bellek alma işlemleri için de göstergeç kullanma gereği olduğunu belirtmiştik. Yapılar üzerinde de göstergeçler benzer şekillerde kullanılacaktır.

4.2.Örnek Program:

Yapılarla ilgili kavramları bir örnek program üzerinden anlatmak, kavramların anlaşılmasını çok daha kolaylaştıacağı için tüm yukarıda bahsettiğimiz kavramları örnekleyen bir programı verip bunun üzerinden açıklamalar vereceğiz.

Örnek programımızda kullanıcı tarafından girilen bilgilerle oluşturulan bir öğrenci yapısı dizisi üzerinden numarası verilen bir öğrenciyi bulup o öğrenci ile ilgili bilgileri ekrana yazdıran bir fonksiyon bulunacak. Öncelikle öğrenci yapısı şu şekilde tanımlanmıştır:

struct student {

```
int no;
char name[30];
int age;
int year;
char dept[20];
}
```

Bu tanımda da görüleceği gibi, öğrenci (student) yapısı, öğrencinin numarası (no: tamsayı), adı (name: en fazla 30 karakterlik dizin), yaşı ve yılı (age ve year: tamsayı), bölümü (dept: en fazla 20 karakterlik dizin) şeklinde bölümlere sahip olacak.

Bulunan öğrencinin bilgilerini yazdıracak fonksiyon (print) öğrenciye ait bir göstergeç alıp, göstergeç üzerinden öğrencinin bilgilerine ulaşıp bu bilgileri ekrana yazdıracak. Bu fonksiyon aşağıdaki şekilde tanımlanmıştır:

```
void print(struct student *pst)
{
printf("Öğrencinin numarası: %d\n",pst->no);
printf("Öğrencinin adı: %s\n", pst->name);
printf("Öğrencinin yaşı: %d\n", pst->age);
printf("Öğrencinin bölümü:%s\n",pst->dept),
printf("Öğrencinin yılı: %d\n",pst->year);
}
```

Bu fonksiyonda parametre göstergeç yerine normal bir yapı olarak da tanımlanabilirdi. Bu fonksiyon parametreyi değiştirmeye gerek duymadığı için "değerle çağırma" yöntemini uygulamanın bu açıdan bir zararı olmazdı. Ancak, yapılar çok büyük olabilirler, bu nedenle değerle çağırma yöntemi kullanıldığında gereksiz yere tüm yapının bir kopyasının yaratılması hem gereksiz hafıza işgaline neden olacağı için, hem de bu kopyalama işlemi de geresiz yere zaman alacağı için, göstergeç üzerinden yapılara ulaşmak birçok durumda daha avantajlı olmaktadır ve genellikle bu yöntem kullanılır (aynı fonksiyonun göstergeçsiz versiyonunu yazın).

Bu fonksiyon ana programdan çağırılmadan önce başka bir fonksiyon tarafından öğrenci bilgileri dizisi oluşturulacak. Bunu gerçekleştirecek olan fonksiyon (create) aşağıda tanımlanmıştır:

```
int create (struct student stlist[])
{
        int i, no;
        printf("Döngüyü bitirmek için öğrenci numarası olarak 0 girin");
        printf("\n Yeni öğrenci numarasını girin:");
        scanf("%d",&no);
        for (i=0;no;i++)
        {
        stlist[i].no=no;
        printf("\n Öğrenci adını girin:");
        scanf("%s",stlist[i].name);
```

```
        printf("\n Öğrencinin yaşını girin:");
        scanf("%d",&stlist[i].age);
        printf("\n Öğrencinin bölümünü girin:");
        scanf("%s",stlist[i].dept);
        printf("\n Öğrencinin sınıfını girin");
        scanf("%d",&stlist[i].year);
        printf("\n Yeni öğrenci numarasını girin:");
        scanf("%d",&no);
        }
        return (i);
}
```

Bu fonksiyonun parametresi öğrenci dizisi olduğu için sadece dizinin adını kullanarak bu dizi üzerinde değişiklikler yapılabilmiştir. Ayrıca bu fonksiyon çağırıldığı ana fonksiyona topam kaç adet öğrenci bilgisi girildiğini de döndürmektedir. Özellikle girdi cümlelerinde dizgi ve tamsayı değerlerin girilmesi sırasında kullanılan yöntem farkına dikkat edin.

Ana programda ilk olarak bu fonksiyon çağırılarak öğrenci dizisi yaratıldıktan sonra, kullanıcıdan hakkında bilgi istenilen öğrencinin numarası sorulacak (programın kullanım mantıksızlığını bu noktada lütfen gözardı edin, örneği basitleştirmek için bu tür bir yöntem kullanmamız gerekti) ve bu numaraa sahip bir öğrencinin var olup olmadığı aranarak, eğer varsa bu öğrenci ile ilgili bilgileri yazdırmak için print fonksiyonu çağırılacak, değilse böyle bir öğrencinin olmadığı mesajı ekrana yazdırılacaktır. Bu işlemi gerçekleştiren fonksiyon şu şekilde tanımlanmıştır:

```
void search(int stno, struct student stlist[], int n)
{
        int i;
        for (i=0;i<n;i++)
        if (stno==stlist[i].no)
        {
                print(&stlist[i]);
                i=n+1;
        }
        if (i==n) printf("Bu numaraya ait bir öğrenci yoktur.\n");
}
```

Bu fonksiyonun ilk parametresi aranan öğrencinin numarasını içermektedir; ikinci parametre ise öğrenci bilgilerinin saklandığı dizidir; son parametre ise dizide saklanan öğrenci sayısını içerektedir. Eğer aranan öğrenci bulunursa print fonksiyonuna bu öğrenci bilgisinin bulunduğu yapının (dizinin ilgili elemanının) adresi yollanmaktadır. Ayrıca döngüyü bitirmek için i'ye n'den büyük bir sayı atanmaktadır. Eğer döngü boyunca 1. parametrede verilen numaraya ait hiç bir öğrenci bilgisi bulunamazsa döngü bittiğinde i'nin değeri n'ye eşit olacağı için bu öğrencinin bulunamadığı mesajı ekrana yazdırılabilecektir.

Şimdi bütün bu tanımlardan sonra ana programımızı verebiliriz:

```
main()
{
        struct student student_list[100];
        int n, sno;
        n=create(student_list);
        printf("Aramak istediğiniz öğrencinin numarasını girin:");
        scanf("%d",&sno);
        search(sno,student_list,n);
}
```

ÖZYİNELEME (RECURSİON) VE ALGORİTMA ANALİZİ

5.1. Özyineleme (Recursion)

Matematik ve bilgisayar bilimlerinin temel kavramlarından biri olan "özyineleme", programlamada da sıkça kullanılmaktadır. Kısaca özyinelemeli program kendi kendini çağıran program olarak tanımlanabilir. Özyinelemeli bir programın kendi kendisini çağırmak dışında, bir de sonuçta durabilmesi için özyinelemeli olmayan "bitme koşulu" (termination condition) olarak adlandırılan kısma sahip olması gerekmektedir.

Aslında birçok ilginç algoritma özyinelemeli olarak daha doğal ve daha anlaşılır biçimde kolayca ifade edilebildiği için, birçok durumda özyinelemeli olarak algoritma tasarlamak tercih edilmektedir. Bu konuda, özyineleme kavramını bir araç olarak nasıl kullanabileceğimizi, özyineleme kavramının programlamada kullanılmasının ne gibi durumlarda olumlu ne gibi durumlarda ise olumsuz olabileceğini ve özyinelemeden kaçınmak gereken durumlarda ne yapılabileceğini inceleyecegiz.

Matematikte birçok fonksiyon özyinelemeli olarak tanımlanmaktadır. Bu şekilde tanımlanan en bilinen örnek aşağıdaki faktoriyel tanımıdır.

0! = 1 ve N>= 0 ise N! = N * (N-1)!

Bu tanımlamadaki 0! bitme koşulunu N! tanımında yer alan (N-1)! ise özyinelemeli kısmı oluşturmaktadır. C dilinde bu tanımın direk kod haline dönüştürülmesi ile aşağıdaki fonksiyonu tanımlayabiliriz:

```
{ if (N == 0) return 1;

return N*factorial(N-1);

}
```

Aslında bu özyinelemeli fonksiyon çok basit bir for döngüsü ile yapılabilecek bir işi gerçekleştirmektedir. Bu nedenle bu örnek özyineleme kavramının gerçek gücünü göstermemektedir. Aynı fonksiyonun for döngüsü kullanarak özyinelemesiz tanımı şöyle yapılabilir:

```
int factorial(int N)

{ int f=1;

for (int i=1;i<=N;i++)

                f=f*i;

return f;
}
```

Bir diğer çok bilinen özyinelemeli tanım ise Fibonacci sayılarıdır:

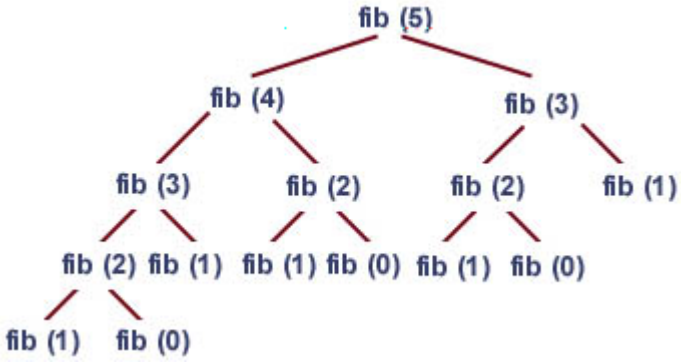
$$F_0= F_1=1 \text{ ve } N>=2 \text{ ise } F_N= F_{(N-1)} + F_{(N-2)}$$

Bu tanımlama 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ... serisini oluşturur. Yine bu tanımın direk C koduna dönüştürülmesi ile aşağıdaki fonksiyonu tanımlayabiliriz:

```
int fibonacci(int N)

{ if (N <= 1) return 1;

return fibonacci(N-1)+ fibonacci(N-2);
}
```



Bu örnek ise özyineleme kavramının gücünü göstermekten öte özyinelemenin bazen ne kadar kötü dahi olabileceğini göstermektedir. Aslında fibonacci sayısını verilen bir N değeri için özyinelemesiz olarak bir for döngüsü ile yaklaşık N adet işlem (döngü) yaparak hesaplamak mümkünken yukarıdaki algoritma birçok tekrar yaparak aynı sonucu yaklaşık a^N (1<a<2 oldugu için N büyüdükçe verimsizlik çok hızla artmaktadır, bkz. algoritma analizi) sayıda işlemle (özyineleme) bularak çok verimsiz çalışmaktadır.

Yukarıdaki ağaç yapısından ne kadar çok gereksiz tekrar yapıldığı bu küçük örnekte bile açıkça görülmektedir. Bu örnekte fib(2) 3 defa fib(3) 2 defa hesaplanmaktadır. Özyinelemesiz çözüm aşağıda verilmiştir:

```
int fibonacci (int N)

{ int fibn2 = 1, fibn1=1, fibn;

for (int i=2;i<=N;i++)

{
        fibn=fibn1+fibn2;

        fibn2 = fibn1;

        fibn1 = fibn;
}

return fibn;
```

```
}
```

Özyinelemeli program aslında kullanıcının açıkça göremediği ek bir yük de oluşturmaktadır. Aslında her işlev çağırılması o anda bulunan ortamdaki tüm değiskenlerin saklanmasını, çağırılan işlevlere parametreleri geçirmek için atama işlemleri yapılmasını ve yeni işlevin ortamındaki değişkenlerin yaratılmasını gerektirmektedir. Özyinelemeli programlarda ise aynı işlev kendi kendisini genellikle çok sayıda çağırdığı için hem bellek hem de zaman açısından verimsiz olabilmektedir. Yukarıdaki örneklerde olduğu gibi bazı özyinelemeli tanımların özyinelemesiz versiyonlarının oluşturulması çok kolaysa bunların özyinelemesiz yazılması tercih edilmelidir. Ancak bazı problemlerde özyinelemesiz hale dönüştürme işlemi çok karışık olabilir. Ayrıca sistemin yaptığı bellek ve zaman açısından verimsizlige neden olan işlev çağırma işlemlerinin de ortadan kaldırılması aslında bunların kullanıcının kendi programı içerisinde aynı işlemleri farklı bir biçimde gerçekleştirmesi haline dönüşebilir. Bu gibi durumlarda özyineleme tercih edilmelidir. İlerki konularda göreceğimiz bazı veri yapısı işlemleri özyinelemeli olarak verilecektir. Bu işlemlerin genellikle özyinelemesiz versiyonları da özyinelemelilerden daha verimli olmadığı için özyinelemelileri tercih edeceğiz.

Sadece özel bir durumda özyinelemeyi ortadan kaldırmak çok basit ve verimli olmaktadır. Bu özel durum "son özyineleme" (end or tail recursion) olarak adlandırılan durumdur. Eğer işlevdeki son işlem özyinelemeli çağırma işlemi ise bu tür özyinelemeler son özyineleme olarak adlandırılmaktadır. Örneğin yukarıdaki özyinelemeli faktoriyel tanımı böyle bir tanım değildir, çünkü özyinelemenin sonucu çarpma isleminde kullanılmaktadır. Yani son işlem çarpma işlemidir. Faktoriyel işleminin bir diğer özyinelemeli tanımı aşağıda verilmiştir (bu tanımlama ile işlev çağırılırken 1. parametre N'e karşılık gelirken 2. parametre1olmalıdır):

```
int factorial(int N, int f)

{ if (N == 0) return f;

return factorial(N-1, N*f);

}
```

Bu tanımlama ile 5! hesaplanması söyle olmaktadır:

factorial(5,1)

factorial(4,5)

factorial(3,5*4)

factorial(2,5*4*3)

factorial(1,5*4*3*2)

factorial(0,5*4*3*2*1)

120

Görüldüğü gibi aslında her özyinelemeli çağırma işleminde parametrelere yeni değerler atanıp işlev baştan tekrar çalışmaktadır. Bu nedenle aynı işlevi goto ve döngü kullanarak özyinelemesiz olarak şöyle yazabiliriz:

```
int factorial(int N, int f)

{ 1: if (N == 0) return f;

N = N-1; f = N*f; // özyinelemeli çağırma işlemi sırasındaki parametreler

goto 1;

}
```

Bu şekilde özyinelemeyi kaldırmak verimlilik sağlamasi açısından tercih edilmelidir. Hernekadar faktoriyel örneği için çok daha basit for döngü içeren bir yöntemi geliştirmek çok kolay olsa da başka son özyinelemeli işlevler için bu tür bir çözüm bulmak çok zor olabilir. Bu gibi durumlarda goto gibi kodun anlaşılırlığını bozduğu için tercih edilmeyen bir cümle kullanarak dahi özyinelemesiz bir işlev yazmak daha yararlı olacaktır. Gerçi birçok derleyici

optimizasyonu son özyinelemeyi tesbit edip derleme sırasında otomatik olarak ortadan kaldırmaktadır.

5.2. Algoritma Analizi

Problemlerin çoğu için birden fazla değişik algoritma geliştirilebilir. Bu algoritmaların karşılaştırılması ve en uygun olanının seçilmesi oldukça zor bir işlemdir.

Genellikle problemlere girdi olarak gelen verinin miktarı problemin doğal yapısı ile ilgili bir fikir verir. Girdinin miktarını genellikle N (pozitif bir tamsayı) ile belirtebiliriz. Örneğin sıralama probleminde girdi bir tamsayı kümesi olabilir, ve bu durumda bu problemin girdisinin miktarını N adet tamsayı olarak ifade edebiliriz. Bir problemi çözmek amacıyla geliştirilen algoritmanın gereksinim duyacağı kaynakların miktarı, girdi miktarına bağlı bir fonksiyon olarak belirtilir. Yine sıralama algoritması örneğinde, geliştirdiğimiz bir algoritmanın gereksinim duyduğu kaynakların miktarını "bellekte 2*N'lik bir yere ve çalışma zamanı olarak da N^2birim zamana gereksinim duymakta" şeklinde tanımlayabiliriz.

Algoritmalar karşılaştırılırken en çok, girdi miktarına göre ortalama durumda ya da en kötü durumda (algoritmanın en çok zaman almasını sağlayacak, o algoritma için en kötü olabilecek girdi konfigürasyonu) ne kadar zaman aldıklarının fonksiyonu kullanılır. Programcının hiçbir etki ya da kontrolünde olmayan bir çok faktör de programın hızını etkilemesine rağmen algoritma analizinde kullanılmaz. Örnegin C ile yazılan bir program derleyici tarafından önce makine diline çevrilmekte daha sonra bu kod bilgisayar tarafından çalıştırılmaktadır. Bu çevirme işlemi sırasında hangi derleyicinin ne tür bir kod ürettiği, kullanılan bilgisayarın hangi makine dili komutlarının olup, bunların her birinin ne kadar birim zaman aldıkları, özellikle çok

kullanıcılı ortamlarda aynı anda birçok kullanıcının bulunması sonucu kaynak paylaşımının işletim sistemince nasıl gerçekleştirildiği gibi birçok faktör geliştirilen algoritmanın kullanılan bilgisayara, derleyiciye vs. bağlı olarak verimliliğini etkilemesine rağmen bütün bu faktörlerin tümü kodu geliştiren programcının kontrolünde olmadığı için analiz dışında bırakılır.

Algoritmaların analizinde yukarıda sayılan ve analiz dışında bırakılan bir çok faktör bulunması nedeniyle, kesinlik yerine yaklaşık karşılaştırma yaparak algoritmaların birbirleri ile görece (relative) durumlarını belirtebilen teknikler kullanılır. Örneğin çoğunlukla bir algoritmanın en kötü durumda, girdi konfigürasyonu ne olursa olsun girdinin büyüklüğüne bağlı olarak nasıl bir

fonksiyonu aşmayacak sayıda temel işlem yaptığı belirtilmeye çalışılır. Bunun için algoritmanın çalışması sırasında yaptığı temel işlemler sayılarak girdi miktarı ile belirtilen bir fonksiyonu elde edilir. Bu fonksiyonda da sadece en büyük terim, katsayılarından ayrıştırılarak kullanılır. Örneğin bir algoritmanın en kötü durumda "$3*N^2+7*N+5$" temel işlem yaptığı belirlenirse bu ifadedeki en büyük terim N^2'li olduğu için sadece N^2 terimi katsayısız olarak bu algoritmanın en kötü durumda harcadığı zamanı belirtmek amacıyla kullanılır. Girdi miktarı N üzerinden algoritmaların çoğunu aşağıdaki sınıflara ayırabiliriz:

1 : Girdinin miktarından bağımsız olarak bir ya da sabit bir sayıda komutun çalıştığı algoritmalardır. Bunlara sabit zamanlı algoritmalar denir. Örnegin 3 adet sayının en büyüğünü bulmak için geliştirilecek bir algoritma (zaten girdinin miktarı da burada N olmayıp sabit, 3, oldugu için onu da -yani katsayısını atarak- 1 ile belirtebiliriz) bu sınıfa girer.

log N: Girdi miktarı N iken herhangi bir logaritma tabanında log N işlem yapan algoritmalar logaritmik olarak adlandırılır. Örnegin taban 2 için girdi 1000 ise yaklasık 10, girdi 100.000 iken 18 (ya da katları) civarında işlem miktarını içeren bir algoritmayı belirtir. Büyük problemlerin bölünüp küçültüldüğü algoritmalardır. Daha sonraki konularda gorecegimiz sıralı bir dizide eleman arama problemi icin bu türde bir algoritma bulunmaktadır

N: Bu tür algoritmalar dogrusal (linear) olarak adlandırılır ve girdi ile işlem miktarı oranı bir katsayı ile belirlenebilen algoritmaları tanımlar. N girdisi (ya da çıktısı) olan bir problem için optimum algoritma bu sınıfa girer. Örneğin verilen gelişigüzel bir sayı dizisindeki en büyük sayıyı bulan bu hesaplama karışıklığına sahip bir algoritma kolaylıkla yazılabilir.

NlogN: Oldukça çok algoritma bu gruba girer. Doğrusal algoritmalardan daha çok zaman almasına rağmen N'e bağlı olarak çok da hızlı büyümedigi için birçok problem için oldukça verimli sayılabilir. Genellikle problemin küçük parçalara bölünüp ayrı ayrı çözümlerinin birleştirildiği türde algoritmalardır. Sıralama problemi icin bu türde algoritmalar geliştirilmiştir.

N^2: Bu tür algoritmalar quadratik (quadratic) olarak adlandırılır. Genellikle girdideki tüm veri çiftlerinin işlendigi algoritmalardır, çok büyük N değerleri için verimsiz olur. Sıralama problemi icin geliştirilen temel ve basit algoritmalar bu turdendir

N^3: Bu tür algoritmalar kübik (cubic) olarak adlandırılır. Büyük N degerleri için çok verimsiz olur.

2^N: Bu tür algoritmalar ekponansiyel (exponential) olarak adlandırılır. Genellikle sadece çok küçük N değerleri dışında pratik olarak kullanılamazlar, çünkü N'e baglı olarak zaman çok hızlı büyür. Bütün olasılıkların denendiği algoritmalar olarak düşünülebilir.

Yukarıda tanımlanan algoritma sınıflarını karşılaştırabilmek için, değişik N değerleri için bu ifadelerin değerlerini gösteren bir tablo aşağıda verilmiştir:

$\log_2 N$	N	N $\log_2 N$	N^2	N^3	2^N
3	10	30	100	1000	1024
6	100	600	10000	1000000	10^{30}
9	1000	9000	1000000	1000000000	-----
13	10000	130000	100000000	10^{12}	-----
16	100000	1600000	10^{10}	10^{15}	-----
19	1000000	19000000	10^{12}	10^{18}	-----

5.2.1. Hesaplama Karışıklığı (Computational Complexity)

Algoritmaları incelerken sabit faktörleri ve küçük terimleri atıp sadece en büyük terimle yapılan işlem sayısını girdi miktarına bağlı olarak ifade etmeye yarayan matematiksel araç olan ve "O gösterimi" (O notation) olarak adlandırılan yöntem şöyle tanımlanabilir:

Bir g(N) fonksiyonu için, eğer $c_0 f(N)$ ifadesini tüm $N>N_0$ değerleri için g(N)'den büyük yapabilecek herhangi bir c_0 ve N_0 sayıları bulabilirsek, o zaman g(N) = O(f(N)) olarak iade edebiliriz.

Örneğin algoritmamız $g(N) = 3*N^2+7*N+5$ işlem yapsın. $c_0= 4$ ve $N_0= 10$ değerleri (böyle birçok değer bulabiliriz) $f(N) = N^2$ için g(N) = O(f(N)) = $O(N^2)$ diyebiliriz. Bu durumda $O(N^2)$ kadar işlem yapan algoritmamızı quadratic olarak tanımlayabiliriz.

Bu ifade algoritmanın yaklaşık olarak hangi ifade ile orantılı bir verime sahip olduğunu verir. Bu da karışık ve detayli analizler yerine algoritmaları

karşılaştırabilmek için yeterli bir fikir verir. Aslında O ifadesi ile bir algoritmanın işlem zamanını belirtmek algoritmanın o kadar sayıda işlem yapacağı anlamına gelmez ve çoğunlukla da bu en üst sınırdır. Bu da sadece eğer elimizde O(NlogN) ve $O(N^2)$'lik iki algoritma varsa büyük olasılıkla O(NlogN)'lik olanın daha az zaman alacağı anlamına gelmektedir. Tabi bu algoritmanın çok büyük bir katsayıya diğerinin ise çok küçük bir katsayıya sahip olmaları durumunda pratik ölçekte birçok girdi miktari için $O(N^2)$'lik olanın daha az zaman alması da sürpriz olmamalıdır.

Özyinelemeli olmayan algoritmaların "hesaplama karışıklığı" algoritmanın içerdiği temel işlemlerin sayılmasi ile kolayca elde edilebilir. Bunu gerçekleştirirken iki temel kurala uyulması gerekmektedir:

1) Toplama Kuralı: Algoritmada peş peşe (sequential) gelen bölümlerin her birinin "hesaplama karışıklığı" (O-gösterimi) tesbit edilmişse, tümünün "hesaplama karışıklığı" bu ifadelerden en büyüğüne eşittir.

example(n)

```
{
        segment1        //      O(n)
        segment2        //      O(n^2)
        segment3        //      O(nlogn)
}
```

Yukarıdaki örnek algoritmanın hesaplama karışıklığı n, n^2, ve nlogn terimlerinin en büyüğü olan n^2 terimi olduğu için $O(n^2)$ olacaktır.

2) Çarpma Kuralı: Algoritma iç içe (nested) bölümlerden oluşuyorsa ve her bir bölümün "hesaplama karışıklığı" ayrı ayrı tesbit edilmişse, tümünün "hesaplama karışıklığı" bu ifadelerin çarpımına eşittir.

example (n)

```
{
        for (i=1;i<=n;i++)              //      O(n)
    for (j=1;j<=n-i;j++)    //  O(n) ... n-i <= n
//  ve  daha  küçük  bir  O  gösterimi  yok
    f(j)        //  O(n) ... önceden bulunduğunu varsayarak
}
```

Yukarıdaki örnek algoritmanın hesaplama karışıklığı 3 adet n ifadesinin çarpımı olan O(n3) olur.

İç içe döngülerde en içteki döngüdeki işlemlerin toplam kaç defa yapıldığının belirlenmesi ve katsayılarının atılması ile o program parçasının "hesaplama karışıklığı" elde edilebilir. Döngü sayısı 2'den fazla olduğunda eğer döngü değişkenleri ile sınır değişkenleri arasında çok fazla ilişki varsa en içteki döngünün tam olarak kaç defa yapıldığının belirlenmesi oldukça gücleşebilir. Ama her zaman için tüm katsayılar atılacağından yaklaşık olarak sonucun belirlenmesi bir çok durumda yeterli olabilmektedir.

Birçok algoritmanın özyinelemeli olarak tanımlandığından daha önce bahsetmistik. Bu tür algoritmaların girdi miktarına bağlı olarak yaptıkları işlem sayısı da özyinelemeli ilişkiler ile tanımlanabilir. Bu durumda girdi miktarına bağlı olarak böyle bir algoritmanın yaptığı işlem sayısını belirlemek için bu ilişkinin çözülmesi gerekir, çünkü işlem sayılarının sayılarak belirlenmesi çok zor olacaktır.

Örneğin girdi olarak aldığı bir diziyi her seferinde en küçük elemanı bulup onu ilk pozisyona yerleştirip dizinin uzunluğunu bir eksilten ve kendi kendisini tekrar çağıran özyinelemeli bir sıralama algoritmasının yaptığı işlemlerin sayısını

$$C(N) = C(N-1) + N$$

şeklinde özyinelemeli bir ilişki ile tanımlayabiliriz. Her seferinde en küçük elemanı bulmak için tüm dizi baştan sona taranmalı (N) ve en küçük eleman baslangıç pozisyonuna konduktan sonra dizinin uzunluğu bir azaltılarak aynı işlem tekrarlanmalıdır (C(N-1)). Böyle bir ilişki çözüldügünde $C(N) = ½(N(N+1))$ elde edilecektir. Bu nedenle böyle bir algoritma $O(N^2)$ ile ifade edilebilir.

YIĞITLAR (STACKS)

6.1. Giriş:

Yığıt, son giren-ilk çıkar (last-in/first-out) kuralıyla eleman erişiminin sınırlandırıldığı bir veri yapısıdır. Aşağıda bu veri yapısının C kodu (stack.c) ile tanımı verilmiştir:

```
/* STACKSIZE yığıtın en fazla alabileceği eleman
 sayısını gösteren sabit */

#define STACKSIZE 100

struct Stack{

        int top;

        int nodes[STACKSIZE];

       };

    struct Stack *create_stack(struct Stack *ps);

    void push(struct Stack *ps,int i);

    int pop(struct Stack *ps);

     int empty(struct Stack *ps);
```

```
int full(struct Stack *ps);
```

Yığıt, yukarıda verilen tanımda da görüldüğü gibi temel olarak bu yapıya eleman ekleme ve eleman çıkarma operasyonlarının yapılabildiği bir veri yapısıdır.

Bu yapıda ilk erişilecek eleman yığıta en son eklenen elemandır. Bu son eleman yığıttan çıkarıldıktan sonra, doğal olarak bir sonraki eleman yapıya en son eklenen eleman haline dönüşmekte ve bir sonraki adımda da bu elemana ulaşmak mümkün olmaktadır. Bu şekilde herhangi bir yığıtta, yığıta en yeni eklenen elemandan ilk eklenen elemana doğru bir sırada elemanlara erişmek olasıdır.

Bu veri yapısı üzerinde tanımlanan push(iteleme) ve pop(alma) işlemleriyle bu veri yapısına eleman ekleme ve çıkarma işlemlerini yürütmek ve bu veri yapısını değiştirmek mümkündür. Bu iki işlemin yanısıra yukarıdaki tanımda, kullanılan yığıtın dolu yada boş olduğu bilgisini verecek full ve empty adında iki ayrı işlem daha tanımlanmıştır.

Yukarıda tanımlanan yığıtın elemanlarının tamsayılar (integer) olarak belirlendiğine dikkat ediniz. Tanımlanan bu tamsayı yığıtına çesitli elemanlar ekleyen ve çıkaran örneğe aşağıdaki bağlantıdan erişebilirsiniz.

6.2. Yığıt İşlemlerinin Gerçekleştirimi:

Yığıt üzerindeki işlemlerin ne gibi sonuçlar verdiğini inceledikten sonra şimdi bu işlemlerin geçekleştirimini daha detaylı inceleyebiliriz.

Bir önceki bölümde verilen tanımdaki işlemler aşağıda verilmiştir:

```
struct Stack *create_stack(struct Stack *ps)

    {

     ps=malloc(sizeof(struct Stack));

     ps->top=-1;
```

```
        return ps;
      }
void push(struct Stack *ps,int i)
      {
         if ( full(ps) )
           printf("Yığıt dolu, eleman yerleştirilemedi.\n") ;
         else
           { ps->top++; ps->nodes[ps->top] = i;}
      }
 int  full(struct Stack *ps)
        {return (ps->top== (STACKSIZE -1));}
    int  pop(struct Stack *ps)
      {
         if ( empty(ps) )
             printf("Yığıt boş, eleman alınamadı.\n") ;
         else
             { int t= ps->nodes[ps-> top] ; ps->top--; return t;}
      }
```

```
int empty(struct Stack *ps)

{return (ps->top== -1);}
```

Yukarıda tanımlanan işlemlerde top değişkeni herhangi bir anda yığıtta bulunan eleman sayısını göstermektedir. Bu değişkenin yığıt büıüklüğüne ulaşması yığıtın dolduğunu, sıfırın altına düşmesi ise yığıtın boş olduğunu göstermektedir. Bu değişkenin değerine bağlı olarak empty ve full fonksiyonları tanımlanmış ve bu iki fonksiyon kullanılarak da diğer iki yığıt işlemi gerçekleştirilmiştir.Ayrıca tanımlanan işlemler için bir yığıt göstergeci kullandığı için, bir yığıt yaratma fonksiyonu da tanımlanmış ve bu fonksiyonda (create_stack) parametre olarak verilen göstergeç için bellekte gerekli yer ayrılıp, "Top=-1" komut satırı ile de yığıt boş hale getirilmiştir.

6.3. Yığıt Uygulamaları:

Programlama dünyasında yığıt veri yapıları birçok algoritmada kullanılmaktadır. Bu dersimizde sondan hesaplama (postfix evaluation) adını verdiğimiz algoritmayı ve burada yığıt veri yapısının kullanımını inceleyeceğiz.

6.3.1. Sondan Gösterim (postfix representation)

Matematiksel ifadelerin bildik gösteriminde operatör (ikili), bu operatörün iki argümanı arasına yerleştirilir. Eğer matematiksel bir ifade içiçe argümanlar ve operatörlere sahipse, bu durumda işlem sırasının ve her operatörün argümanlarını belirleyebilmek için parantezleme yöntemi uıgulanır. Buna ek olarak proglamlama dillerinde operatörlerin öncelik sıraları ve işlem yönleri ile de parantezler kullanılmadan işlem sıraları belirlenebilir. Aşağıdaki matematiksel ifade parantezleme yöntemi kullanan bir gösterime örnek oluşturmaktadır:

((5*9)+8)*((4*6)+7)

Ancak matematiksel ifadeler bu gösterimin dışında, sondan gösterim (postfix representation) diye adlandırılan biçimde de ifade edilebilir. Bu gösterim biçiminin en büyük özelliği işlem sırasının belirlenebilmesi için parantezleme

yöntemine gereksinim duymamasıdır. Bu gösterimde operatör iki argumanının sağına yerleştirilmektedir. Örneğin 4+5 ifadesi bu gösterimle 4 5 + şekline dönüşecektir. Yukarıda verilen örneğin sondan gösterim (postfix representation) ile ifadesi ise şöyle olacaktır:

5 9 * 8 + 4 6 * 7 + *

Bu gösterimde işlem sırası daima soldan sağadır ve her operatör, solunda yer alan ilk iki arguman üzerinde işletilecektir. Yukardaki örnekte ilk çarpma işleminin argumanları 5 ve 9 olurken, bir sonraki toplama işleminin argumanları bu ilk iki sayı çarpımının sonucu ve 8 rakamı olacaktır. Bu şekilde soldan sağa bir işlemeyle verilen ifadenin sonucunu hesaplamak mümkündür.

6.3.2. Yığıt Kullanarak Sondan Hesaplamanın (postfix evaluation) Gerçekleştirimi:

Tanımlanan gösterimde verilmiş bir ifadeyi yığıt kullanarak gerçekleştirmek için aşağıdaki algoritmayı kullanabiliriz:

1) Verilen ifadenin birinci elemanından sonuncu elemanına kadar:

1.1) Sıradaki eleman eğer bir arguman ise,

1.1.1) Bu elemanı yığıta koy.

1.2) Sıradaki eleman bir operatör ise,

1.2.1) Yığıttan sırayla iki eleman al.

1.2.2) Bu iki eleman üzerinde operasyonu uygula.

1.2.3) Sonucu tekrar yığıta koy.

Verilen bu algoritmanın gerçekleştiriminin ve bir örnek ifade üzerinde yığıtın uğradığı değişikliklerin gösterildiği örneğe aşağıdaki bağlantıdan erişebilirsiniz.

KUYRUK VERİ YAPISI (QUEUES)

7.1. Giriş:

Kuyruk, ilk giren-ilk çıkar (first-in/first-out) kuralıyla eleman erişiminin sınırlandırıldığı bir veri yapısıdır.

Asağıda bu veri yapısının C kodu ile tanımı (kuyruk.c) verilmiştir:

```
/* QUEUESIZE kuyruğun en fazla alabileceği eleman
      sayısını gösteren sabit */

 #define QUEUESIZE 100

 struct  Queue {

            int front,rear,size;
            nodes[QUEUESIZE];

            };

   Queue *create_queue(struct Queue *pq);

   void insert(struct Queue *pq,int i);

   int del(struct Queue *pq);

   int empty(struct Queue *pq);

   int full(struct Queue *pq);
```

7.2. Kuyruk İşlemlerinin Gerçekleştirimi:

Veri yapımız üzerindeki işlemlerin ne gibi sonuçlar verdiğini inceledikten sonra, şimdi bu işlemlerin gerçekleştirimini daha detaylı inceleyebiliriz.

Bir önceki bölümde verilen tanımdaki işlemleri gerçekleştiren program parçaları aşağıda verilmiştir:

```
struct Queue *create_queue(struct Queue *pq)

    {

     pq=(struct Queue *)malloc(sizeof(struct Queue));

     pq->front=0; pq->rear=QUEUESIZE-1; pq->size=0;

     return pq;

    }



  void insert(struct Queue *pq,int i)

       {

          if ( full(pq) )

            printf(" Kuyruk dolu, eleman yerlestirilemedi.i\n");

          else

           { pq->rear=(pq->rear+1) % QUEUESIZE; pq->nodes[pq->rear]=i;
             pq->size++;}

       }



  int full(struct Queue *pq)

       { return (pq->size== QUEUESIZE ); }
```

```
int del(struct Queue *pq)
    {
        int t;
        if ( empty(pq) )
            printf(" Kuyruk boş, eleman alınamadı.\n");
        else
            {
                t= pq->nodes[pq->front] ;
                pq->front=(pq->front+1) %QUEUESIZE;
                pq->size--;
                return t;
            }
    }
        int  empty(struct Queue *pq)
    { return (pq->size== 0); }
```

Yukarıda tanımlanan işlemlerde “size" değişkeni herhangibir anda kuyrukta bulunan eleman sayısını gösteren bir değişkendir. Bu değişkenin kuyruk büyüklüğüne ulaşması kuyruğun dolduğunu, sıfıra düşmesi ise kuyruğun boş olduğunu göstermektedir. Bu değişkenin değerine bağlı olarak empty ve full fonksiyonları tanımlanmış ve bu iki fonksiyon kullanılarak da diğer iki kuyruk işlemi gerçekleştirilmiştir. Eleman yerleştirme ve silme fonksiyonlarında ayrıca kuyruğun başını ve sonunu belirleyen değişkenler front ve rear, yerleştirilecek ve silinecek elemanların dizinin hangi elemanı olduğunu belirlemek için kullanılmıştır.

Yukarıdaki algoritmayı incelediğimizde kuyruğa çeşitli elemanlar eklenip çıkarıldıkça front ve rear değişkenlerinin değerleninin arttığını ve kuyruk büyüklüğüne doğru yaklaştığını görüyoruz. Burada ilginç nokta aşağıdaki şekilde de görüldüğü gibi kuyruk sonunun dizinin sonuna ulastığı durumda ortaya çıkmaktadır. Kuyruk sonu dizinin sonuna ulastığında, aşağıdaki şekilde de görüldüğü gibi eğer bazı elemanlar kuyruğun başından silinmişse dizinin baş tarafında bir bölüm boşalmış olacak ve de kuyruk henüz dolu olmayacaktır.

Bu durumda yeni bir eleman yerleştirmek için bir yöntem varolan tüm elemanları dizinin başına kaydırmak ve dizinin sonunda yeni elemanlar için yer yaratmak olabilir. Ancak bu işlemin getireceği ek işlem yükünden kurtulmak için verilen algoritmada farklı bir yöntem izlenmiştir.

Aşağıdaki şekilde de görüldüğü gibi algoritmamızda var olan elemanları kaydırmak yerine kuyruk elemanlarının koyulduğu dizi dairesel bir biçimde kullanılarak, kuyruk sonu dizinin baş tarafına doğru

SAYFA-36

uzatılmıştır. Algoritmada yer alan mod işlemi de dizinin bu tür kullanımını gerçekleştirmek için, kuyruk başı ve sonu değişkenlerinin (front, rear) dizi büyüklüğüne eristiğinde tekrar dizinin başına döndürülmeleri için kullanılmıştır.

Kuyruk veri yapısında erişilecek veya bu veri yapısından çıkarılacak eleman, var olan elemanların içinde veri yapısina ilk koyulan eleman olarak belirlenmiştir. Yani bu veri yapısında elemanlar veri yapısına giriş sırasına göre dizilirler ve bu sıra kullanılarak veri yapısından alınabilirler.

Bu veri yapısı üzerinde tanımlanan insert(yerleştirme) ve delete(silme) işlemleriyle bu veri yapısına eleman ekleme ve çıkarma işlemleri gerçekleştirilmektedir. Yine bu işlemleri gerçekleştirmek için yukarıdaki kuyruk yapısının dolu veya boş olduğunu belirlemek için gereksinim duyulan full ve empty fonksiyonları da tanımlanmıştır. Ayrıca kuyruk veri yapısının manipülasyonu için gereksinim duyulan, front, rear ve size adını verdiğimiz ve sırasıyla kuyruğun başını, sonunu ve büyüklüğünü belirten değişkenler de yukarıdaki tanımda yer almaktadır. Yine önceki derste olduğu gibi gerçekleştirilen işlemler için parametre olarak bir kuyruk göstergeci kullanıldığı için yine bir yaratma fonksiyonu tanımlanmış ve bu fonksiyonda (create_queue) parametre olarak verilen göstergeç için bellekte gerekli yer ayrılıp, değişkenlere ilk atamalar yapılmıştır. Bu değişkenlerin ve tanımlanan fonksiyonlarin kullanımını bir sonraki bölümde detaylı olarak inceleyeceğiz.

Bu incelemeden önce, bir kuyruk veri yapısına çeşitli elemanlar ekleyen ve çıkaran örneğe aşağıdaki bağlantıdan erişebilirsiniz.

7.3. Kuyruk Uygulamaları:

Programlama dünyasında kuyruk veri yapısı birçok algoritmada kullanılmaktadır. Bu dersimizde kuyruk veri yapısını, bir markette müşterilerin kasada oluşturdukları kuyruğun simulasyonunu gerçekleştirerek inceleyecegiz.

7.3.1. Kuyruk Simulasyonunun Gerçekleştirimi:

İnceleyeceğimiz uygulamada bir markette yaptığı alışverişin parasını ödemek için dakikada bir müşteri para kasasının başına gelmektedir. Bazı müşterilerin yaptığı alışveriş miktarı fazla olduğu için, bu müşteriler kasa başında bir dakikadan fazla kalmakta, bu yüzden de kuyruk oluşmaktadır.

Aşağıdaki bağlantıda ilk müşteriden son müşteriye servis verilene kadar kasa başında oluşan kuyruşun durumunu simule eden program parçasına erişebilirsiniz.

LİSTELER (LİSTS)

8.1. Giriş:

Liste, bu veri yapısında yer alan elemanların sıralı bir biçimde saklandığı ve erişilebildiği bir veri yapısıdır. Bu veri yapısındaki elemanları düz bir çizgi üzerinde birbirine bağlanarak sıralanmış olarak düşünebiliriz.

Aşağıda bu veri yapısının C kodu (list.c)ile tanımı verilmiştir:

```
struct Item {

    struct Item *next;

    int val;

};


struct List {

    struct Item *list;

};

// Boş liste yaratır
struct List *create_list(struct List *pl);

//Listenin başına bir tamsayı ekle
void insert_elm(struct List *pl,int);

// Listenin sonuna bir tamsayı ekle
void append_elm(struct List *pl,int);

// Belirtilen tamsayıyı listeden çıkar
void remove_elm(struct List *pl,int);

// Listedeki eleman sayısını bul
int length(struct List *pl);

// Listedeki elemanları ekrana yaz
void display(struct List *pl);

// Listenin sonunda yer alan elamanın göstergecini bul
struct Item *atEnd(struct List *pl);
```

Yukarıdaki tanımlamada da görüldüğü gibi liste veri yapısının sonuna yada başına eleman ekleme işlemleri append ve insert işlemleriyle gerçekleştirilmektedir. Remove işlemi ise verilen bir elemanı (listenin herhangibir yerinde olabilir) listeden çıkarmaktadır. Liste veri yapısında elemanlar arasındaki sıralama sadece elemanların listedeki pozisyonlarına dayalı olacaktır.

Yukarıda list yapısının (structure) tanımından önce Item adını verdiğimiz bir başka bir yapı tanımı kullanılmıştır. Bu yapı listede yer alacak elemanları belirlemektedir ve list adı ile yarattığımız listeler aslında Item'lardan oluşmaktadır. Yukarıdaki Item tanımına göre yaratılan liste yapısında tam sayılar (integers) eleman olarak yer alacaklardır. Sadece bu yapının tanımını değiştirerek farklı tür ve yapılarda elemanların yer alabileceği değişik listeler tanımlamak da mümkün olabilir. Bu tanımlamada val değişkeni ile listenin elemanlarının türü belirtilirken, next değişkeni ile herhangibir elemanı bir sonraki elemana bağlayacak göstergeç tanımlanmıştır. Liste tanımındaki insert, append gibi fonksiyonlarla bu değeri değiştirmek ve böylece yeni elemanı listede başka elemanlara bağlamak mümkün olmaktadır. Tanımlanan item yapısı aşağıdaki şekilde de görüldüğü gibi liste elemanının saklandığı ve bir sonraki elemana bağlantıyı sağlayan göstergecin yer aldığı, iki farklı kısımdan oluşmaktadır.

Aşağıdaki şekilde ise Item tanımıyla elde edilen eleman yapılarının oluşturduğu bir liste yer almaktadır.

8.2. Liste İşlemlerinin Gerçekleştirimi:

Liste üzerindeki işlemlerin ne gibi sonuçlar verdiğini inceledikten sonra şimdi bu işlemlerin gerçekleştirimini daha detaylı inceleyebiliriz.

Bir önceki bölümde verilen sınıf tanımındaki işlemler aşağıda tanımlanmıştır:

```
struct List *create_list(struct List *pl)
  // Boş liste yaratır

    {
```

```
   pl=(struct List *)malloc(sizeof(struct List));
   pl->list=0;
   return pl;
  }

void insert_elm(struct List *pl,int val)
// listenin başına yeni bir eleman ekler.
     {
         struct Item *pt=(struct Item *)malloc(sizeof(struct Item));
         pt->val=val;
         pt->next = pl->list;
         pl->list = pt;

     }
struct Item *atEnd(struct List *pl)
// listenin son elemanını bulup onu gösteren göstergeçi döner.
     {
         struct Item *curr;
         if (pl->list == 0)
```

```
            return 0; // liste boş.


        curr = pl->list;

        while (curr->next)

            curr = curr->next;

        return curr;

    }
void append_elm(struct List *pl,int val)
 // listenin sonuna yeni bir eleman ekler.

    {

        struct Item *pt=(struct Item *)malloc(sizeof(struct Item));

        pt->val=val;

        pt->next=0;

        if (pl->list==0)

            pl->list = pt;

        else

            (atEnd(pl))->next = pt;

    }


 void remove_elm(struct List *pl,int val)
 // Parametre olarak belirtilen değere sahip elamanı bulup listeden çıkarır.
```

```
{
    struct Item *prv, *pt = pl->list;
    // listenin başındaki eleman için.
    if ( (pt) && (pt->val==val) )
        {
            pl->list= pt->next;
            free(pt);
        }

    // listenin ortasında yer alan elemanlar için.
    else if (pt)
        {
        for ( prv=pt, pt= pt->next;
            ((pt) && (pt->val != val));
            prv=prv->next, pt=pt->next);
        if (pt->val==val)
            {
                prv->next=pt->next;
```

```
                free(pt);
            }
        }
    }

int length(struct List *pl)
    {
        struct Item *tmp=pl->list;
        int i=0;
        while (tmp)
            {
                i++;
                tmp=tmp->next;
            }
        return i;
    }
```

```
void display(struct List *pl)
    {
        struct Item *tmp=pl->list;
        while (tmp)
            {
                printf("%d ", tmp->val) ;
                tmp=tmp->next;
            }
    }
```

Yukarıda tanımlanan işlemlerde insert ve append listenin başına ve sonuna eleman ekleme işlemlerini gerçekleştirmektedir. Append fonsiyonunda listenin son elemanının bulunması için endList fonksiyonu kullanılmıştır. Remove fonksiyonunda ise ilk if ifadesinde eğer listenin ilk elemanı val parametresinde belirtilen değere sahip ise, bu eleman listeden çıkarılmakta, ikinci if ifadesinde yer alan döngüde ise eğer listenin ortasında uygun bir eleman varsa, bu eleman listeden çıkarılmaktadır. Listenin başında yer alan bir elemanın silinmesi için list göstergecinin bir sonraki elemanı gösterir hale getirilmesi ve bu elemanın silinmesi yeterliyken, listenin ortasındaki bir elemanın silinmesi için aşağıdaki şekilde de gösterildiği gibi, ek bir işlem olarak silinecek elemandan bir önceki elemanın next göstergeci, silinecek elemandan bir sonraki elemanı gösteren bir hale getirilmektedir. Bu farklı iki durumdan dolayı, remove fonksiyonu iki ayri bölümden oluşmaktadır. Yine bu veri yapısı için de bir create fonksiyonu tanımlanmış ve bu fonksiyon aracılığı ile bellekten gerekli yer alınmıştır. Ayrıca liste veri yapısı ilk tanımlandığında herhangibir elemana sahip olmadığı için list göstergeci bu yaratma fonksiyonunda sıfıra eşitlenmiştir. Bu işlem yapılmaz ise göstergecin bellekte rastgele bir adresi göstereceğine ve bunun da hatalara yol açabileceğine dikkat ediniz.

Ayrica listenin uzunluğunu bulan ve listenin elemanlarını gösteren fonksiyonlar, length ve display yukarıda tanımlanmıştır.

8.3. Liste Uygulamaları:

Bu dersimizde bir sınıftaki öğrencilerin bilgilerinin eleman olarak yer aldığı bir listede arama işlemini uygulama olarak inceleyeceğiz.

UYGULAMA:

Bu uygulmada öğrenci bilgilerinin yer aldığı bir listede arama yapılmaktadır. Öğrenci bilgileri öğrencinin adı, soyadı, numarası, adresi, kaçıncı sınıfta olduğu bilgilerini içermektedir. Öğrenci bilgileri yeni bir yapı olarak tanımlammış ve derste verilen Item yapı tanımı bu bilgileri içerecek şekilde aşağıda değiştirilmiştir.

```
Struct Student {

    struct Item *next;

    int stdno;

    char *name;

    char *surname;

    char *adress;

    int year;

};


struct Item {

     struct Item *next;
```

```
    struct Student std;

};



Struct List {

   struct Item *list;

};
```

Bu tanımlamada liste yapısının değişmediğine ancak item yapısında derste olduğu gibi bir int değeri yerine bir Student yapısının kullanıldığına dikkat ediniz. Bu durumda derste tanımlanan fonksiyonlarda listeye sadece bir tamsayı eklenip çıkarıldığından fonksiyon parametrelerinde tamsayılar yer almıştir. Ancak bu yapıyla bu fonksiyonları kullanmak için bu parametreleri ve fonksiyon icersindeki atamaları Student yapısı üzerinde çalışabilecek şekilde değiştirmek gerekmektedir. Aşağıdaki fonksiyonda ise öğrenci numarası verilen öğrenci bu yeni tanımladığımız listede aranmakta ve bu öğrenciye ait bilgiler ekrana gönderilmektedir.

```
void  search_list(struct List *pl,int key)

   {

      Item *pt = pl;

      while ( (pt) && (pt->std.stdno!=key) )

         pt = pt->next;

      if (pt)

         {

            printf("Student No: %d \n",pt->std.stdno);
```

```
            printf("Student Name: %s \n",pt->std.name);

            printf("Student Surname: %s \n",pt->std.surname);

            printf("Student Adress: %s \n",pt->std.adress);

            printf("Student Class: %d th year\n",pt->std.year);

        }

    else

        printf("No student exists with student number %d\n", key);

}
```

Aşağıdaki çizimlerde "search" fonksiyonun verilen l1 listesi için l1.search(3456) şeklinde çağrıldığında işletimini inceleyebilirsiniz.

i) l1 listesi

ii) search_list (l1,3456) fonksiyonu çağrıldığında liste ve pt göstergecinin durumu

iii) Fonksiyondaki while döngüsü işletildiğinde liste ve pt göstergecinin durumu

iv) İf-else yapısıyla ekrana gönderilen veri

Student No: 3456
Student Name: Tanyel
Student Surname: Türkaslan
Student Adress: ODTÜ Mim. Fak.
Student Class: 5th year

AĞAÇLAR (TREES)

9.1. Giriş:

Ağaç veri yapısı, elemanlarının birbirlerine liste veri yapısında olduğu gibi göstergeçler (pointers) aracılığı ile bağlandığı bir veri yapısıdır. Bu veri yapısının listelerden farklılığı, liste veri yapısının aksine bir elemanın birden fazla elemana bağlanmasının mümkün olmasıdır. Bu sayede ağaç veri yapısında yer alan elemanlar arasında hierarşik bir diziliş sağlamak mümkündür. Bir ağaç veri yapısında bir elemanın en fazla kaç elemana bağlanabileceği o ağaç veri yapısının derecesini(order) belirtmektedir. Derecesi iki olan ağaç veri yapısı en basit ağaç türü olarak oluşmakta ve ikili ağaç diye adlandırılmaktadır. Aşağıdaki şekilde elemanları tamsayı olan bir ikili ağaç örneği verilmiştir.

Liste veri yapısında herhangi bir elemandan önce (predessor) veya sonra gelen (successor) sadece bir veya sıfır(liste başı ve sonu için) eleman bulunmaktaydı. Agaç veri yapısında ise her elemanın yine bir veya sıfır önce geleni (predessor) bulunurken, o elemandan sonra gelen eleman sayısı birden fazla (en fazla ağacın derecesi kadar) olabilir. Her ağaçta kendisinden önce hiçbir eleman bulunmayan bir eleman bulunur. Bu eleman yukarıdaki şekilde de görüldüğü gibi en üstte yer alan elemandır ve ağacın kökü (root) olarak adlandırılacaktır. Ayrıca kendisinden sonra hiç eleman bulunmayan elemanlar da ağaçta yer alır ve bu elemanlar da ağacın yaprakları (leaves) olarak adlandırılır. Yukarıdaki örnek ikili ağaçta 5 değerine sahip eleman ağacın kökü ve 3,6 ve 13 değerlerine sahip elemanlar da ağacın yapraklarıdır. Ayrıca ağaç veri yapısında herhangibir elemandan sonra gelen elemanlar, o elemanın çocukları (children) olarak adlandırılırken, yine herhangibir elemandan önce yer alan eleman da, o elemanın velisi (parent) olarak adlandırılmıştır. Ayrıca ağaç veri yapısının oluşması için her elemanın sadece bir tek (kök hariç) velisi olması gerektiğine dikkat ediniz.

Liste veri yapısında olduğu gibi ağaç veri yapısında da elemanlar arasında bir sıralama kullanmak mümkündür. İkili arama ağacı (binary search tree) diye adlandırdığımız ağaç veri yapısında bu sıralama ölçütü her elemanın sol tarafında yer alan çocuğundan büyük olması ve sağ tarafında yer alan çocuğundan küçük olması olarak belirlenmiştir. Bu dersimizde anlatım kolaylığının sağlanabilmesi için sunacağımız program parçaları ve uygulamalarda ikili arama ağacı kullanılacaktır. Ancak bu uygulama ve

programlar baska ağaç türleri için de kolaylıkla güncellenebilir. Bu noktadan sonra agaç kavramı da ikili arama ağacı anlamında kullanılacaktır.

Aşağıda ağaç veri yapısının C kodu (tree.c) ile tanımı verilmiştir:

```
struct Node {

        struct Node *leftchild, *rightchild;

        int key;

        };

struct Tree {

        struct Node *root;

        };


        struct Tree *create_tree(struct Tree *tree);
        // Agaç göstergeci için bellekte yer ayırır

        struct Node *insert(int, struct Node *root);
        //Ağaca yeni bir eleman ekle

        struct Node *search(int,struct Node *root);
        // Belirtilen tamsayıyı ağaçta bul

        struct Node *delete_node(int,struct Node *root);
       // Değeri belirtilen elemanı ağaçtan çıkar
```

Yukarıdaki tanımda da görüldüğü gibi ağaç veri yapısına eleman ekleme ve çıkarma işlemleri insert ve delete işlevleriyle gerçekleştirilmektedir. Search

işlevi ise değeri belirtilen bir elemanı ağaç içersinde bularak bu elemanı işaret eden bir göstergeç döndürmektedir.

Yukarıda Tree tanımından önce Node adını verdiğimiz bir başka bir yapı tanımı kullanılmıştır. Bu yapı ağaçta yer alacak elemanların türünü belirlemektedir ve Tree yapısı olarak yarattığımız ağacın elemanları Node yapısında tanımlanan şekilde olacaktır. Bu da ağaç yapısı içindeki kök (root) bölümünün Node yapısına bir göstergeç olarak tanımlanmasıyla elde edilmiştir. Yukarıdaki Node tanımına göre yaratılan ağaç tam sayılardan (integers) oluşacaktır. Sadece bu tanımı değiştirerek farklı yapılarda elemanların yer alabileceği değişik ağaçlar tanımlamak da mümkün olabilir. Node tanımında key değişkeni ile ağaçtaki elemanlarının türü belirtilirken, rightchild ve leftchild değişkenleri ile herhangibir elemanı çocuklarına bağlayan göstergeçler tanımlanmıştır.

Tanımlanan Node yapısı aşağıdaki şekilde de görüldüğü gibi ağaç elemanının saklandığı ve çocuklara bağlantıyı sağlayan göstergeçlerin yer aldığı, üç farklı kısımdan oluşmaktadır.

Aşağıdaki şekilde ise Node tanımıyla elde edilen eleman yapılarının oluşturduğu bir ağaç yer almaktadır.

9.2. Ağaç İşlemlerinin Gerçekleştirimi:

Ağaç veri yapısı üzerindeki işlemlerin ne gibi sonuçlar verdiğini inceledikten sonra şimdi bu işlemlerin gerçekleştirimini daha detaylı inceleyebiliriz.
Bir önceki bölümde verilen tanımdaki işlemler aşağıda gerçekleştirilmiştir:

```
struct Tree *create_tree(struct Tree *tree)
// Agaç gostergeci için bellekte yer ayırır

    {

     tree=(struct Tree *)malloc(sizeof(struct Tree));

     tree->root=0;
```

```
    return tree;
  }

struct Node *insert(int key,struct Node *root)
// Ağaca yeni bir eleman ekler.
  {
    struct Node *pt=(struct Node *)malloc(sizeof(struct Node));
    pt->key=key;
    pt->leftchild=0;
    pt->rightchild=0;
      if (!root)
            root=pt;
      else
          {
            if (key > root->key)
              root->rightchild=insert(key,root->rightchild);
            else
              root->leftchild=insert(key,root->leftchild);
          }
```

```
            return root;
    }


struct Node *search(int key,struct Node *root)
// key değişkeninde belirtilen elemanı  bulur.
    {
        if (!root)
            return root;
        else
            {
                if (key == root->key)
                    return root;
                else if (key > root->key)
                    return search(key,root->rightchild);
                else
                    return search(key,root->leftchild);
            }
    }
```

```
struct Node *delete_node(int key,struct Node *root)
// Parametre olarak belirtilen değere sahip elamanı bulup ağaçtan çıkarır.
    {

        struct Node *marker=root, *parent=0, *child=root, *temp;
        //Silinecek elemanı bul
        while (marker && marker->key!=key)
            {
                parent=marker;
                if (key < marker->key)
                    marker= marker->leftchild;
                else
                    marker= marker->rightchild;
            }
        if (!marker)
            printf( "Eleman ağaçta bulunamadı\n");
        else
            {
```

```
if (!parent) //kök silinecek
    {
    if (!marker->rightchild) //kökün sağ çocuğu yok
        root=marker->leftchild;
    else if (!marker->leftchild) //kökün sol çocuğu yok
        root=marker->rightchild;
    else { //kökün iki çocuğu da var
                for (temp=marker,child=marker->leftchild;
                    child-> rightchild;
                    temp=child, child=child->rightchild);
                if (child!=marker->leftchild)
                    {
                        temp->leftchild=child->leftchild;
                        child->leftchild=root->leftchild;
                    }
                child->rightchild=root->rightchild;
                root=child;
                }
      }
    else if (!marker->rightchild)
```

```
    //silinecek elemanın sag çocuğu yok
    {
        if (parent->leftchild==marker)
            parent->leftchild=marker->leftchild;
        else
            parent->rightchild=marker->leftchild;
    }
else if (!marker->leftchild)
    //silinecek elemanın sol çocuğu yok
    {
        if (parent->leftchild==marker)
            parent->leftchild=marker->rightchild;
        else
            parent->rightchild=marker->rightchild;
    }
        else //Silinecek elemanın iki çocuğu da var.
    {
        for (temp=marker,child=marker->leftchild;
            child->rightchild;
```

```
                temp=child, child=child->rightchild);
            if (child!=marker->leftchild)
                {
                    temp->rightchild=child->leftchild;
                    child->leftchild=marker->leftchild;
                }
            child->rightchild=marker->rightchild;
            if (parent->leftchild==marker)
                parent->leftchild=child;
            else
                parent->rightchild=child;
        }
    free(marker);
    }
  return root;
}
```

Yukarıda tanımlanan işlemlerde Insert ve Search işlevleri özyinemeli olarak tanımlanmışlardır. Eleman ekleme (insert) işlevinde eğer ağaç boş ise yeni bir eleman yaratılmış ve kök (root) göstergeci bu yeni elamanı gösterir hale getirilmiştir. Eğer ağaçta bazı elemanlar var ise bu durumda özyinemeli

çağrılarla yeni elemanın ağacın uygun yerine yerleştirilmesi sağlanmıştır. Bu işlev sonuçta ağacın yeni halini gösteren kök degişkenine dönmektedir. Benzer bir yaklaşımla arama (search) işlevinde ise aranan eleman ağaçta bulunmakta ve o elemanı işaret eden göstergeç belirlenmektedir. Eğer aranan eleman ağaçta yok ise göstergecin sıfır olarak döndürüleceğine dikkat ediniz.

Yukarıda tanımlanan işlevlerden eleman silme işlevi (delete) en karmaşık işlev olarak karşımıza çıkmaktadır. Bunun nedeni, işlevin herhangibir eleman silindiğinde ağacın yine ikili tarama ağacı özelliklerini korumasını sağlamasıdır. İkili tarama ağacının korunması için bu işlevde elemanın silinmesinin yanısıra, ağaç üzerinde bazı düzenlemeler de gerçekleştirilmiştir.

Aslında bu işlevi, silinecek elemanın ağacın hangi bölümünde olabileceği olasılıklarına göre değişik alt parçalardan oluşmuş olarak düşünebiliriz. Her alt parçacık oluşan durumda gereken düzenlemeyi yapmak üzere tanımlanmıştır. Şimdi bu işlevi bu alt parçalara göre adım adım inceleyebiliriz.

İ) İşlevin başında yer alan while döngüsü silinecek elemanın ağaç üzerindeki yerini tespit etmektedir. Eğer eleman bulunamaz ise marker göstergeci sıfır değerini almakta ve ekrana elemanın bulunamadığına dair bir mesaj gönderilmektedir

İİ) Elemanın bulunduğu durumda çeşitli olasılıklar tek tek incelenmektedir. İlk olasılık olarak silinecek elemanın kök elemanı olması ele alınmıştır. Bu durum kökün sağ veya sol çocuğunun bulunmaması veya iki çocuğunun da bulunması olarak üç alt durumda incelenmiştir. Kökün sadece bir çocuğu olması durumunda bu çocuk yeni kök haline getirilmektedir. Kökün iki çocuğu olduğu durumda yapılması gereken düzenleme biraz daha karmaşıktır. Kök silineceği için kökün yerine ondan bir önceki veya bir sonraki eleman yerleştirilmelidir. Bu kökün sol tarafında yer alan alt ağacın en sağındaki eleman olabileceği gibi, sağ tarafında yer alan alt ağacın en solunda yer alan eleman da olabilir. Tanımlanan işlevde sol alt ağaç kullanılmıştır. Aşağıdaki şekilde örnek bir ağaç üzerinde kök silindiğinde yapılacak düzenleme ve elde edilen yeni ağaç gösterilmiştir.

Bu bölümde göstergeçler üzerinde yapılan değişiklikler yukarıdaki şekilde gösterilen düzenlemenin gerçekleştirilmesi için yapılmaktadır. Ağacın yeni şeklinin oluşturulurken elemanların kopyalanmayıp sadece göstergeçlerinin değiştirildiğine dikkat ediniz.

iii) Silme işlevinde ikinci büyük alt parça silinecek eleman kökten farklı bir eleman olduğunda karşımıza çıkmaktadır. Burada yine üç alt bölümde silinecek elemanın sağ veya sol çocuklarının bulunmaması veya iki çocuğunun da bulunması durumları ele alınmıştır. Çocuklardan birisi bulunmadığında yine kökün silinmesi durumunda olduğu gibi, silinecek eleman, velisinin sağında veya solunda olduğu gözetilerek aradan çıkarılmaktadır. Silinecek elemanın iki çocuğunun bulunduğu durumda ise yine aşağıdaki şekilde gösterildiği gibi bu elemanın sol alt ağacının en sağında bulunan eleman silinecek elemanın yerine kaydırılmaktadır.

9.3. Ağaç Uygulamaları:

İkili arama ağacında saklanan verileri, kullanılan erişim sırasına göre üç değişik şekilde taramak mümkündür. Bu tarama şekilleri sıralı (inorder), sıra önceli (preorder) ve sıra sonralı (postorder) olarak adlandırılmaktadırlar.

8 Sıra önceli(preorder) taramada her elemana o elemanın sol ve sağ alt ağaçlarındaki tüm elemanlardan önce erişilmektedir.

9 Sıralı(inorder) taramada her elemana , o elemanın sol alt ağacındaki elemanlardan sonra, fakat sağ alt ağacındaki tüm elemanlardan önce ulaşılmaktadır.

iii) Sıra sonralı(postorder) taramada her elemena, o elemanın sol ve sağ alt ağaçlarındaki tüm elemanlardan sonra erişilmektedir.

Aşağıdaki şekilde bir ikili ağaç, ve bu ağacın elemanlarının bu üç değişik tarama yöntemiyle erişilme sıraları verilmiştir.

Bu tarama yöntemlerinin C işlevleri biçiminde gerçekleştirimi aşağıda verilmiştir.

```
void inordertraversal(struct Node *root)
    {
       if (root)
            {
                inordertraversal(root->leftchild);
                printf("%d ",root->key);
                inordertraversal(root->rightchild);
            }
     }

  void preordertraversal(struct Node *root)
    {
       if (root)
            {
                printf("%d ",root->key);
                preordertraversal(root->leftchild);
                preordertraversal(root->rightchild);
            }
```

```
}

void postordertraversal(struct Node *root)
{
    if (root)
        {
            postordertraversal(root->leftchild);
            postordertraversal(root->rightchild);
            printf("%d ",root->key);
        }
}
```

BASİT SIRALAMA ALGORİTMALARI (ELEMENTARY SORTİNG ALGORİTHMS)

10.1. Giriş:

Bilgisayar dünyasında yaplan işlemlerin önemli bir kısmı kullanıcıya verilerin belirli bir ölçüte göre sıralanarak sunulmasına ayrılmaktadır. Örneğin uygulamalarda sık sık isimlerin alfabetik sıralamaya göre, sayısal verilerin ise matematiksel büyükllüklere göre sıralanması gerekmektedir. Verileri belirli bir ölçüte göre sıralamak için çeşitli algoritmalar kullanılmaktadır. Bu algoritmalari belirli bir veri dizisini sıralamak için ne kadar işleme gereksinim duyduklarına göre sınıflamak mümkündür.

Bu yaklaşımı kullanarak sıralama algoritmalarını *Basit Sıralama Algoritmaları (Elementary Sorting Algorithms)* ve *İleri Sıralama Algoritmaları (Advanced Sorting Algorithms)* olmak üzere iki sınıfa ayırmak mümkündür. Bu dersimizde *Seçerek sıralama (Selecetion sort), Yerleştirerek sıralama (Insertion sort)* ve *Kabarcık sıralaması (Bubble sort)* olarak adlandırılan üç değişik basit sıralama algoritmasını inceleyeceğiz. Basit sıralama algoritmaları sıralama problemine anlaşılır ve basit bir giriş olanağı sağladığı gibi, veri miktarının çok büyük olmadığı birçok uygulamada, ileri sıralama algoritmalarına göre daha kolay bir biçimde gerçekleştirilebildikleri için tercih edilmektedirler

10.2. Sıralama Algoritmalarının Gerçekleştirimi:

Aşağıda, bir tamsayı dizisi *(integer array)* üzerinde işletilen bu üç sıralama işlevinin gerçekleştirimi (kod.c) verilmiştir. Ayrıca sıralama işlevlerinde kullanılan ve dizinin belirtilen indekslerindeki iki değeri yer değiştiren *swap* işlevi de aşağıda tanımlanmıştır.

```
void selection_sort()
{

        int i,j, min;
        for (i=0;i<ARRAYSIZE-1; i++)
                {
                        min=i;
                        for                     (j=i+1;j
                        <=ARRAYSIZE-1;j++)
                                if (array[j] <
                                array[min])
                                min=j;
                        swap(array, min,i);
                }
```

```
}
void insertion_sort()
{
        int i,j, v;
        for (i=1;i<=ARRAYSIZE-1; i++)
                {
                        v=array[i]; j=i;
                        while (j>0 && array[j-1] >
                        v)
                                { array[j] =
                                array[j-1]; j-
                                -; }
                        array[j] = v;
                }
}
void bubble_sort()
{
        int i,j;
        for (i=ARRAYSIZE-1;i>=1; i--)
                for (j=1; j<=i; j++)
                        if (array[j-1] > array[j])
                        swap(array, j-1, j);
}


void swap(int array[], int i, int j)
{
        int tmp;
        tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
}
```

Şimdi bu sıralama algoritmalarının gerçekleştirimini sırayla inceleyebiliriz.

10.2.1.Seçerek Sıralama (Selection Sort):

Seçerek sıralama algoritmasının işletimini şöyle özetleyebiliriz. Dizideki en küçük elemanı bul, bu elemanı dizinin ilk (yer olarak) elemanıyla yer değiştir. Daha sonra ikinci en küçük elemanı bul ve bu

elemanı dizinin ikinci elemanıyla yer değiştir. Bu işlemi dizinin tüm elemanları sıralanıncaya kadar sonraki elemanlarla tekrar et.

Yukarıdaki işlev gerçekleştirimine baktığımızda *i* değişkeninin sıfırdan başlayıp dizinin sonuna kadar ilerlediğini görüyoruz. Bu değişken arttıkça bu değişkenin o an için belirttiği dizi elemanının solunda kalan elemanların sıralanmış hale geldiğine dikkat ediniz. Böylelikle bu değişken dizinin büyüklüğüne ulaştığında tüm dizi sıralanmış hale gelmektedir.

10.2.2. Yerleştirerek Sıralama (Insertion Sort):

Yerleştirerek sıralama işlevi belirli bir anda dizinin belirli bir kısmını sıralı tutarak ve bu kısmı her adımda biraz daha genişleterek çalışmaktadır. Sıralı kısım işlev son bulunca dizinin tamamına ulaşmaktadır. Yukarıda verilen işlev tanımında da görüldüğü gibi, işlev çalıştırıldığında dizinin belirli bir kısmını sıralı hale getirebilmek için *i* değişkeninin belirlediği eleman dıştaki döngünün her işletilişinde bu dizi parçası içinde uygun yere yerleştirilmektedir. Bu yerleştirme işlemi de i değişkeninin belirlediği elemandan büyük elemanların sağa doğru kaydırılması ve boşalan yere bu elemanın yerleştirilmesiyle gerçekleştirilmektedir. Bir önceki algoritmada olduğu gibi yerleştirerek sıralamada da i değişkeninin değeri arttıkça dizinin sıralı hale gelen bölümü artmakta ve i değişkeni dizinin uzunluğuna ulaştığında tüm dizi sıralanmış olmaktadır.

10.2.3.Kabarcık Sıralaması (Bubble Sort):

Üçüncü sıralama algoritması olarak sunulan kabarcık sıralamanın işletimini ise şöyle özetleyebiliriz. Dizinin elemanları üzerinden tekrar tekrar geçilir ve her geçiste sadece yan yana bulunan iki eleman arasında sıralama yapılır ve bu işlem tüm elemanlar sıralanıncaya kadar devam ettirilir.

Dizinin başından sonuna kadar tüm elemanlar bir kez işleme tabi tutulduğunda dizinin son elemanı en büyük eleman haline gelecektir. Çünkü iç döngüde yer alan ve iki elemanı karşılaştırıp gerekli yer değiştirmeyi yapan işlem sayesinde dizinin en büyük elemanı dizinin neresinde bulunursa bulunsun en sağa dogru kaydırılmış olacaktır. Böylelikle bu iç döngünün bir kez işletimi sonucu en büyük eleman yerine yerleştirilmiş olacaktır. Bir sonraki tarama ise bu en sağdaki eleman dışarıda bırakılarak gerçekleştirilmektedir. Bu dışarıda bırakma işlemi de diış döngüdeki *i* değişkeninin değerinin her işletimde bir azaltılmasıyla sağlanmaktadır. *i* değişkeninin değeri 1 değerine ulaştığında ise dizinin solunda kalan son iki eleman da sıralanmakta ve sıralama işlemi tamamlanmaktadır.

Kabarcık sıralama algoritmasını da belirli veriler için daha iyi performans gösterebilmesi için geliştirmek mümkündür. Yukarıdaki

gerçekleştirimde de görebileceğiniz gibi, bu sıralama algoritmasında iç döngüde ikili sıralamalar yapılmaktadır. Eğer herhangi bir işletim aşamasında bu iç döngüde hiçbir yer değiştirme işlemi yapılmaz ise bu dizinin geriye kalan bölümünün sıralanmış olduğunu gösterecektir. Aşağıda kabarcık sıralama algoritmasının geliştirilmiş hali (bubble.c)verilmiştir. Bu gerçekleştirimde iç döngüde herhangi bir yer değiştirmenin gerçekleşip gerçekleşmediği *flag* değişkeniyle gözlenmekte, eğer bir değişiklik gerçekleşmez ise sıralama işlemi sonlandırılmaktadır.

```
void bubble_sort()
{
        int i, j, flag=1;
        i=ARRAYSIZE-1;
        while ( (i>0) && (flag) )
                {
                        flag=0;
                        for (j=1; j<=i; j++)
                                if (array[j-1]
                                > array[j])
                                {
                                        s
                                        w
                                        a
                                        p
                                        (
                                        a
                                        r
                                        r
                                        a
                                        y
                                        ,

                                        j
                                        -
                                        1
                                        ,

                                        j
                                        )
                                        ;
```

```
                                        f
                                        l
                                        a
                                        g
                                        =
                                        1
                                        ;
                                }
                        i--;
                }
}
```

10.3. Sıralama Algoritmalarının Performans Analizi:

Bu üç algoritmanın performanslarını incelerken sıralanan verilerin bazı özellikleri ön plana çıkmaktadır. Ancak genel olarak bu algoritmalardan seçerek sıralama algoritmasının en çok dizinin sıralanmamış kısmındaki en küçük elemanı bulmak için, kabarcık sıralama algoritmasının ise daha çok dizinin sıralanmamış bölümünde gerçekleştirdiği kısmi sıralamalar için zaman harcadığını söylemek mümkündür. Yerleştirerek sıralama algoritmasında ise dizinin sıralanmamış bölümünde herhangi bir işlem yapılmazken, daha çok işlem yükü dizinin sıralanmış bölümde gerçekleştirilen eleman kaydırma işlemleri üzerindedir.

Aşağıda ise N elemanlı bir dizi için sıralanacak verilerin özellikleri de göz önünde bulundurularak bu algoritmaların bazı performans özellikleri sunulmuştur.

10 Seçerek sıralama algoritmasi yaklaşık $N^2/2$ karşılaştırma ve N yer değiştirme işlemi gerçekleştirmektedir:

Bu özelliği seçerek sıralama işlevinin gerçekleştiriminden çıkarmak mümkündür. Dış döngünün her işletiminde bir tek yer değiştirme işlemi gerçekleştirildiğinden, bu döngü N adet işletildiğinde *(N=ARRAYSIZE)* N tane yer değiştirme işlemi gerçekleştirilecektir. Bu döngünün her işletiminde ayrıca N-i adet karşılaştırma gerçekleştirildiğini göz önüne alırsak toplam karsşılaştırma sayısı $(N-1)+(N-2)+...+2+1» N^2/2$ olacaktır.

11 Yerleştirerek sıralama algoritması ortalama $N^2/4$ karşılaştırma ve $N^2/8$ yer değiştirme işlemi gerçekleştirir ve bu işlem sayısı en kötü durumda iki katına çıkar.

Bir önceki özellikte kullanılan yöntemle yerleştirerek sıralama algoritmasının da $N^2/2$ karşılaştırma yapacağını öne sürebiliriz. Ancak bu sayı bu algoritma için ancak en kötü durumda *(worst case)* ortaya çıkabilir, çünkü burada karşılaştırma işlemi *while (array[j-1] > v)* koşuluna bağlı olarak gerçekleştirilmektedir. Rastgele seçilmiş bir veri kümesi üzerinde ise bu karşılaştırmanın ortalama olarak, üzerinde çalışılan dizi parçasında yer alan elemanların yarısı üzerinde gerçekleşeceğini öne sürebiliriz. Bu durumda toplam karşılaştırma sayısı en kötü durumun yarısı yani $N^2/4$ olacaktır.

12 Kabarcık sıralama algoritması ortalama $N^2/2$ karşılaştırma ve $N^2/2$ yer değiştirme işlemi gerçekleştirir ve bu işlem sayısı en kötü durumda da aynıdır.

Kabarcık sıralama algoritmasının dış döngüsünün her işletiminde N-i adet karşılaştırma ve yer değiştirme gerçekleşmektedir. Bu işlemlerin toplamı da bize Seçerek sıralama algoritmasında olduğu gibi $N^2/2$ sayısını vermektedir.

13 Verinin zaten sıralı olduğu durumda yerleştirerek sıralama hemen hemen doğrusal (linear) bir işletim zamanına sahiptir.

Böyle bir durumda yerleştirerek sıralama algoritmasının iç döngüyü hiç işletmeyeceğini ve sadece dış döngünün işlem yükünü oluşturacağına dikkat ediniz. Bu da algoritmanin dogrusal (linear) bir zamanda bitmesini sağlayacaktır.

HIZLI SIRALAMA ALGORİTMASI (QUİCKSORT)

11.1. Giriş:

Sıralama algoritmaları içinde Hızlı Sıralama algoritması (Quicksort), kolay gerçekleştirimi, diğer algoritmalara göre daha az kaynak kullanması ve de birçok durumda kullanıcıya iyi performanslar sağlaması açısından ön plana çıkmaktadır. Hızlı sıralama algoritmasi N elemanlı bir veri kümesinin ortalama NlogN adımda sıralanmasını sağlamaktadır. Ancak hızlı sıralama algoritmasının özyinemeli gerçekleştirilmesi, (özyineleme kullanılmaksızın gerçekleştirim karmaşıklaşmaktadır), en kötü durumda (worst-case) N2 adımda çalışması ve gerçekleştirimde yapılacak ufak bir hatanın kötü performans sonuçlarına yol açabilmesi, bu algoritmanin dezavantajları olarak karşımıza çıkmaktadır

11.2. Hızlı Sıralama Algoritmasının Gerçekleştirimi:

Hızlı sıralama algoritması bir veri kümesini her adımda ilk bölümde küçükler ve ikinci bölümde büyükler şeklinde iki ayrı bölüme ayırıp (partition) daha sonra bu iki parçanın birbirinden bağımsız bir şekilde sıralanması yaklaşımını kullanmaktadır. Bu ayırma işlemi için bu verilerden birisi (genellikle sonuncusu) referans elemanı (pivot) olarak seçilmekte ve buna göre iki ayrı bölüm oluşturulup, referans elemanından küçük olanlar ilk bölüme, büyük olanlar ikinci bölüme ve referans elemanı da sıralamadan sonra bulunması gereken pozisyona yerleştirilmektedir. Bu parçalama işlemi elde bulunan veri kümesinin özelliklerine göre değişmektedir. Aşağıda bu algoritmanın (quick.c) C kodu sunulmuştur:

```
void quicksort(int a[], int l, int r)
        {
                int i,j,v;
                if ( r > l )
                        {
                                v= a[r]; i=l-
                                1; j=r;
                                while ( i < j
                                )
                                {
                                        w
                                        h
                                        i
                                        l
```

```
e
(
a
[
-
-
j
]
>
v
)
;
i
f
(
i
<
j
)
s
w
a
p
(
a
,
i
,
j
)
;
}
swap(a,i,r);
```

```
                quicksort(a,
                l, i-1);
                quicksort(a,
                i+1, r);
            }
    }
```

Yukarıdaki gerçekleştirimin bir tamsayılar (integers) dizisini sıraladığına dikkat ediniz. Bu işlevde kullanılan *swap* işlevi basit sıralama algoritmaları için de kullandığımız yer değiştirme işlevi olup gerçekleştirimi aşağıda verilmiştir:

```
void swap(int array[], int i, int j)
    {
        int tmp;
        tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
    }
```

Gerçekleştirimi verilen hızlı sıralama işlevinin *l* ve *r* parametreleri sıralanacak alt dizinin sınırlarını belirlemektedir. Eğer dizide *N* eleman var ise işlevin *quicksort(a,1,N)* şeklinde çağrılması tüm diziyi sıralayacaktır. Özyinemeli bölümlerde ise çağrı dizinin belirtilen parçaları için gerçekleştirilecektir.

Tanımlanan işlevin kritik noktasını dizinin iki bölüme bölünmesi oluşturmaktadır. Bu bölme işlemi aşağıdaki üç koşulun gerçekleşeceği biçimde gerçekleştirilmektedir.

İ)*a[i]* elemanı doğru pozisyonunayerleştirilecektir.

14 *a[1], ..., a[i-1]* elemanlarının hiçbiri *a[i]* elemanından büyük olmayacaktır.

iii) *a[i+1], ..., a[*r] elemanlarının hiçbiri *a[i]* elemanından küçük olmayacaktır

Yukarıdaki gerçekleştirimde *a[r]* değeri doğru yerine yerleştirilecek eleman olarak belirlenmiştir. *While* döngüsüyle o an için dizinin sıralanan bölümünün sınırlarını gösteren *i (dizinin başı)* ve *j (dizinin sonu)* değişkenlerinin belirttiği elemanlardan başlayarak dizinin tüm elemanları taranmakta ve *a[r]* değerinden tüm büyük değerler dizinin sonuna gönderilirken bu değerden küçük değerler de dizinin baş tarafına toplanmaktadır. Bunun yapılması için *i* ile taranan

elemanlardan referans elemanından büyük, *j* ile taranan elemanlardan ise referans elemanından küçük bir elemana rastlandığında, bu iki eleman (küçük olan referans elemanının soluna, büyük olan ise sağına geçecek şekilde) yer değiştirmektedir. Bu işlem bu iki işaretçi *i* ve *j* değişkenleri birbirlerini geçene kadar sürdürülmektedir (yine *i* ve *j* değişkenleri referans elemanından büyük/küçük elemanlara rastlayınca durmak üzere). En son olarak da bu iki değişken birbirini geçtiğinde *i* değişkeninin gösterdiği yerdeki elemanla (referans elemanından büyük) referans elemanı yer değiştirmektedir. Bu son işlem de *while* döngüsünden sonra yapılan bir *swap* çağrısı ile gerçekleştirilmektedir. Böylece dizi *a[r]* değerinden daha büyük olanların *a[r]* değerinin sağına, küçük olanların da soluna toplandığı iki bölüme ayrılmış olmaktadır. Daha sonra özyinemeli çağrılarla da bu alt dizilerin bağımsız olarak sıralanmaları sağlanmaktadır.

11.3. Hızlı Sıralama Algoritmasinin Performans Analizi

Hızlı sıralama algoritması için en iyi durum her parçalama işleminde üzerinde çalışılan dizinin daima tam ortadan bölünmesi durumunda ortaya çıkmaktadır. Bu durumda bu algoritmayla *N* elemanlı bir dizide yapılan karşılaştırma sayısı aşağıdaki formülde verilmiştir:

$C_N = 2C_{N/2} + N$

Bu formülde $2C_{N/2}$ elde edilen iki alt dizinin özyineli olarak sıralanması için gereken karşılaştırma sayısıdır. *N* ise ilk parçalama işlemi için gerekecek karşılaştırma sayısıdır. Bu ilişkinin çözümü ise şöyle olacaktır:

$$C_N = N \lg N$$

Yukarıda açıklandığı gibi, bu işlem sayısı diziyi parçalama işleminin daima diziyi eşit iki parçaya böldüğü durumda elde edilecektir. Bu inceleme bu yüzden en iyi durum olarak (best-case) düşünülebilir. Ancak bu durum çok az ortaya çıkacak bir durumdur. Rastgele elemanlardan oluşan bir dizide dizinin her adımda hangi noktadan bölüneceği eşit olasılıklara sahip olacaktır. Algoritmanın ortalama performansı ise ancak durumun incelenmesiyle ortaya çıkarılabilir.

N elemanın rasgele dağılımını ele aldığımızda toplam karşılaştırma sayısını aşağıdaki gibi ifade edebiliriz:

$$C_N = N + 1/N \sum_{1 \le k \le N} C_{k-1} + C_{N-k} \quad (\text{for } N \ge 2 \text{ with } C_1 = C_0 = 0)$$

Bu ifadede her *k* elemanının *1/N* eşit olasılığıyla diziyi parçalayan eleman olabileceği varsayımı kullanılmış ve böylece her oluşabilecek iki alt dizi işlem (C_{k-1} *ve* C_{N-k}) yükü *1/N* değeriyle çarpılarak ortalama işlem yükü ifade edilmiştir. İfadenin başında yer alan *N* değeri ise yine ilk diziyi parçalamak için gereken karşılaştırma sayısıdır. Bu ilişkinin çözümü burada sunulmayacaktır. Ancak bu ilişki çözüldüğü

zaman elde edilen değer *2 ln N'dir.* Bu ise en iyi durumda elde edilen performansın bir sabitle çarpılmış halidir. Bu da bize hızlı sıralama algoritmasının ortalama durumda (average-case) da *O(ln N)* düzeyinde işleme gereksinim duydugunu göstermektedir.

Hızlı sıralama algoritması için en kötü durum (worst-case) ise verilen dizinin zaten sıralı (ya da ters sıralı) olduğu durumda ortaya çıkmaktadır. Bu durumda her defasında en küçük elemanı diziyi bölme işlemi için kullanacağından, sadece bu en küçük elemandan oluşan ilk bölüm ve diğer tüm elemanların yer alacağı ikinci bölüm olmak üzere var olabilecek en kötü parçalama ortaya çıkacaktır. Bu durum işlevin kendisini *N* defa çağırmasıyla $N^2/2$'lik bir işlem yükü getireceği gibi, *N* adet özyinemeli çağrı yer açısından da çok verimsiz bir sonuç ortaya çıkaracaktır. Bir sonraki bölümde böyle bir kötü sonucun önlenmesine yönelik gerçekleştirilebilecek önlemleri inceleyeceğiz.

11.4. Hızlı Sıralama (Quicksort) Algoritmasının Geliştirilmesi:

11.4.1. *Küçük Alt Diziler:*

Yapılabilecek ilk geliştirme kullanılan özyinemeli çağrıların azaltılmasına yönelik olabilir. Hızlı sıralama algoritmasının gerçekleştirimini incelediğimizde özyinemeli çağrıların her defasında dizinin daha küçük parçaları üzerinde işletildiği ve en sonunda bu işletimin sadece ikili eleman çiftlerinin sıralanması için dahi kullanıldığını görmekteyiz. Ancak çok az verinin sıralanması için ayrı bir özyinemeli çağrı yapmanın kaynak kullanımı açısından verimsiz bir durum olduğunu öne sürebiliriz. Böyle bir durumu önlemek için belirlenen bir *M* değerinden daha küçük alt dizilerin sıralanması için özyinemeli olmayan basit bir sıralama algoritmasını kullanmak sıralama algoritmamızı daha verimli bir hale getirecektir. Bu gelişimi gerçekleştirmek için özyinemeli çağrıların gerçekleştirildiği bölüme *"if (r-l £ M) insertion(l,r)"* koşulunu ekleyerek, bu koşul doğru olduğu durumda hızlı sıralama algoritması yerine yerleştirerek sıralama (insertion sort) algoritmasını kullanmak, özyinemeli hızlı sıralama çağrılarına ise koşulun gerçekleşmediği durumlarda devam etmek gerekecektir.

11.4.2. Üçünün Ortası Parçalama Yöntemi (Median-of-Three Partitioning):

Bu yöntem bir önceki bölümde tanımlanan en kötü durumun (worst-case) önlenmesi için kullanılabilecek bir yöntemdir. En kötü durum parçalama işlemi için seçilen referans elemanın en küçük veya en büyük eleman olması durumunda

ortaya çıkmaktaydı. Üçünün ortası parçalama yöntemi ile ise dizinin başında, ortasında ve en sonunda yer alan üç eleman sıraya dizilmekte ve bu elemanlardan ortada bulunan eleman referans elemanı olarak belirlenmektedir. Bu durumda seçilen elemanın yine en büyük veya en küçük eleman olabilmesi için üç elemandan en az ikisinin dizinin en büyük iki veya en küçük iki elamanı olması gerekmektedir. Bu durumun oluşması ise olasılık olarak oldukça düşüktür. Bu yüzden bu yöntemin kullanılmasıyla ortalama olarak parçalama işleminde gelişme sağlamak mümkün olmaktadır.

TEMEL ARAMA METODLARI VE DOĞRAMA YÖNTEMİ (BASİC SEARCHİNG METHODS AND HASHİNG):

12.1. Giriş:

Bilgisayar dünyasında çok sık kullanılan bir başka temel işlemi, belirli bir veri parçasının daha önceden oluşturulmuş geniş bir veri kümesi içerisinde aranarak bulunması oluşturmaktadır.

Sıralama işlemi için olduğu gibi arama işlemi için de yıllardan beri kullanılan bazı arama veri yapıları ve bu yapılar üzerinde çalışan arama algoritmaları bulunmaktadır. Bu hafta bu arama yöntemlerinden sırayla arama (sequential search), ikili arama (binary search) ve doğrama (hashing) methodlarını detaylı olarak inceleyeceğiz.

Bildiginiz gibi bilgisayar dünyasında veriler birbiriyle ilişkili kayıtlar (records) şeklinde saklanmaktadır. Her kaydın, o kaydı diğer kayıtlardan ayırmaya yarayan belirli bir anahtar (key) değeri bulunmaktadır. Sıralama veya arama algoritmaları ile bu anahtarlar kullanılarak veriler sıralanabilmekte veya belirli bir kayıt varolan veri kümesi içinde aranabilmektedir. Anahtar ve kayıt kavramının daha iyi anlaşilabilmesi için günlük yaşamımızda kullandığımız sözlükler iyi bir örnek oluşturmaktadır. Sözlükte yer alan kelimeler sözlükte yer alan verilerin anahtarları, yer alan kelime açıklamaları ve tanımlamaları ise kayıtlar olarak düşünülebilir. Herhangi bir kelimenin açıklamasına sözlükte ulaşmaya çalıştığımızda aradığımız kelimenin kendisini aradığımız bilgi için bir anahtar olarak kullanıp, gereksinim duyduğumuz açıklamaya ulaşırız.

Aynı şekilde sıralama algoritmalarını da belirli anahtarlarla belirlenmiş bir kayıt kümesi üzerinde işlem yapan yöntemler olarak düşünebiliriz. Aslında arama algoritmalarını birden fazla işlemden oluşan bir paket olarak düşünmek mümkündür. Bu işlem paketinde:

i)Kullanılacak veri yapısının ve anahtarın oluşturulması

ii)Yeni bir kayıtın varolan kayıt kümesine eklenmesi

iii)Belirli bir kaydın silinmesi

iv)Belirli bir kaydın anahtarlar tarafından oluşturulan sözlük aracılığı ile aranması

gibi işlemler yer alacaktır. Daha önceki haftalarda çeşitli veri yapılarını incelerken bu tür işlem paketlerini sınıflar halinde tanımlamış ve üzerinde incelemeler yapmıştık. Bu hafta arama yöntemleri üzerinde yapacağımız incelemeyi aslında daha önce oluşturduğumuz çeşitli paketlere arama işlevlerinin eklenmesi olarak düşünebiliriz. Yine daha önceki haftalarda olduğu gibi bu haftaki incelememizde de veri olarak tamsayılar (integers) yer alacaktır. Arama işlevleri bu tamsayıların yer aldığı veri yapıları üzerinde işletilecek ve arama işlevinde kullanacağımız kayıt kümesini bu tamsayılar oluşturacaktır. Bu durumda tamsayılar hem kayıtların anahtarlarını hem de kayıtta bulunan verinin kendisini temsil etmektedir. Böyle bir inceleme gerçekleştirim ve anlatım kolaylığının sağlanması için tercih edilmiştir. Ancak kolaylıkla tanımlanan işlevlerde kullanılan veri türünün tanımını değiştirerek daha karmaşık veri türleri için de işlevlerimizi güncellemek olanaklıdır.

12.2. Sırayla Arama (Sequential Search):

Arama yöntemleri içinde en basit yöntem olarak karşımıza çıkan sırayla (sequential) arama yöntemi bir dizi (array) veya bağlantılı liste (linked list) içinde yer alan elemanların içinde belirli bir verinin, elemanların sırayla kontrol edilerek aranması olarak tanımlanabilir. Bu arama yöntemini kullanmak için veri yapısına eleman ekleme işleminde hiçbir özel yönteme gereksinim yoktur. Elemanlar dizinin veya bağlantılı listenin sonuna sırayla yerleştirilebilirler. Herhangi bir eleman bulunacağı zaman da veri yapısı ilk elemandan son elemana kadar tek tek kontrol edilecektir.

Burada sırayla arama yöntemini incelemek için daha önce incelediğimiz bağlantılı liste (linked list) yapısını kullanacağız. Aşağıda tanımlanan arama işlevi daha önce tanımlanan *List* yapısının varolduğu varsayılarak tanımlanmıştır. Bu yapının tanımlamasının detaylarını hatırlamak için lütfen gerekli ders notlarınızı tekrar inceleyiniz.

```
int seq_search(struct List *pl,int key)

    {

        struct Item *t=pl->list;
```

```
    while ( (t) && (t->val!=key) )
        t=t->next;
    if (t)
        return 1;
    else
        return 0;
}
```

Yukarıda bir bağlantılı liste üzerinde sırayla arama yapan işlev (search.c) tanımlanmıştır. Bu işlevde kullanılan *t* değişkeni listenin en başından başlayarak sırayla elemanların değerlerinin parametre olarak verilen değere uyup uymadığını kontrol etmektedir. Eğer eleman bulunursa işlevin *1* değerini, bulunmaz ise *0* değerini döndürdüğüne dikkat ediniz. Kullandığımız eleman türü sadece tamsayılardan oluştuğu için işlevin sadece aranan tamsayının veri yapısında bulunup bulunmadığını bildirmesi yeterli olmaktadır. Ancak veri yapısındaki elemanlarımız eğer karmaşık bir yapıya sahip olsaydı, parametre olarak işleve girilen anahtar kullanılarak bu karmaşık yapıya sahip eleman belirlenecek ve onu işaret eden bir göstergeç işlevden döndürülecekti.

Sırayla arama işlevinin eğer aranan eleman listede yoksa bütün listeyi baştan başa geçmek zorunda kalacağı için *N+1* (*N* listedeki eleman sayısı) adımda sonlanacaktır. Elemanın listede bulunduğu durumda ise işlevimiz ortalama *N/2* adımda sonlanacaktır. Çünkü rastgele bir dağılımda aranan elemanın listenin herhangi bir yerinde olması olasılığı birbirine eşit olacaktır. Bu yüzden ortalama çalışma süresi *(1+2+...+N-1+N)/N = N/2* olacaktır.

12.3. İkili Arama:

Aramada kullanılan kayıt sayısının çok fazla olduğu durumlarda elemanları tek tek taramak işlem yükünü oldukça fazlalastıracaktır. Bu durumda ikili arama işlem yükünü azaltan bir yöntem olarak karşımıza çıkmaktadır. İkili aramanın gerçekleştirilebilmesi için veri yapımızda yer alan elemanların daha önceden sıralanmış olması gerekmektedir. Bu da ikili arama yönteminin getirdigi ek işlem

yükünü oluşturmaktadır. Elemanlar sıralı olduğu durumda ikili arama metodu önce aranan elemanın anahtarını veri yapısının tam ortasında yer alan elemanla karşılaştırmakta ve bu elemandan büyük veya küçük olma durumuna göre aramaya aynı yöntemle veri yapısının alt veya üst bölümünde devam etmektedir. Bu yöntemle her bir adımla aramanın gerçeklestirildiği veri kümesinin büyüklüğünün yarıya düşürüldüğüne dikkat ediniz. Bu da arama işlevini hızlandıran bir etken olmaktadır. Aşağıda bir dizide yer alan sıralı tamsayılar üzerinde arama işlevini gerçekleştiren işlev(binary.c) tanımlanmıştır.

```
int binary_search(int array[], int key)

    {

    int l=0; int r=ARRAYSIZE -1;
    int x;

    while (r>=l)

       {

          x=(l+r)/2;

          if (key==array[x])

              return 1;

          if (key < array[x])

              r=x-1;

          else

              l=x+1;

       }

    return 0;
```

```
}
```

Tanımlanan ikili arama işlevi hızlı arama (quicksort) işlevinde olduğu gibi dizinin belirli noktalarını gösteren *r* ve *l* değişkenlerini kullanmaktadır. Başlangıçta bu iki değişken dizinin başını ve sonunu gösterirken, aranan anahtarın dizinin hangi bölümünde bulunabileceğinin belirlenmesinden sonra, (orta elemanla karşılaştırarak) belirlenen bölümün sınırlarını gösterir şekilde güncellenmektedirler. Eğer eleman bulunursa yine *1* değeri, eleman bulunamayıp *while* döngüsünden çıkılırsa *0* değeri döndürülmektedir.

İkili arama yöntemi başarılı veya başarısız olsun hiçbir zaman *lg(N+1)* adımdan fazla işlem yapmamaktadır. Bu özellik her adımda arama yapılan veri büyüklüğünün yarıya düşürülmesinden kaynaklanmaktadır. Aşağıdaki ifade ikili arama işlevinin gereksinim duyacağı işlem sayısını göstermektedir:

$$C_N = C_{N/2} + 1$$

Burada $C_{N/2}$ dizi ikiye bölündükten sonra gereksinim duyulacak işlem adedini gösterirken *1* ise diziyi ikiye bölmek için gereken karşılaştırma işlemine denk düşmektedir. Bu ilişkinin çözümüyle ise *lg(N+1)* değeri elde edilmektedir.

12.3.1. *İkili Arama Ağacı (Binary Search Tree):*

Bildiğiniz gibi 9.hafta ağaç (tree) veri yapısını incelerken ikili arama ağacını ve bu veri yapısı üzerinde gerçekleştirilen işlevleri detaylı bir biçimde incelemiştik. İkili arama ağacı da üzerinde ikili aramanın kolaylıkla gerçekleştirilebildiği bir veri yapısı olarak karşımıza çıkmaktadır. Bu veri yapısının detayları burada tekrarlanmayacaktır. Lütfen bu detayları hatırlamak için 9. ders notlarınıza tekrar göz atınız.Bu veri yapısı üzerinde belirli bir kaydın bulunması için gerekli arama işlevi şöyle tanımlanmıştı (tree_search.c):

```
struct Node *search(int key,struct Node *root)
// key değişkeninde belirtilen elemanı  bulur.

    {

        if (!root)

            return root;
```

```
else
    {
        if (key == root->key)
            return root;
        else if (key > root->key)
            return search(key,root->rightchild);
        else
            return search(key,root->leftchild);
    }
}
```

İkili arama işlemini gerçekleştiren yukarıdaki işlev ikili arama ağacının özelliğini kullanarak her defasında aranan anahtar değerinin o anki elemandan büyük veya küçük olmasına göre ağacın sol tarafına veya sağ tarafına doğru arama işlevini sürdürmektedir. Bu arama işlevinin kaç adımda sonlanacağının üzerinde işlem yapılan ikili arama ağacının yapısına bağlı olduğuna dikkat ediniz. Eğer var olan ağacın dengeli bir yapısı varsa dizi üzerinde gerçekleştirilen ikili aramada olduğu gibi yine aynı nedenlerden dolayı arama işlevi başarılı veya başarısız durumda da *lg(N+1)* adımda sonuca ulaşacaktır. Ağacın dengeli bir yapıya sahip olması, kök elemanının sağında ve solunda yer alan alt ağaçların yüksekliklerinin birbirlerine yakın olması anlamına gelmektedir. Dengeli ikili ağaç veri yapısı çok sık kullanılan bir veri yapısıdır. Dengeli ağaç yapısını oluşturmak için ‘ekleme’ ve ‘çıkarma’ işlemleri sırasında çeşitli işlemlerle ağacın dengesinin bozulmamasını sağlayan algoritmalar geliştirilmiştir. Ancak bu algoritmaların verimsiz (lgN’den kötü) olması dengeli ağaç avantajını yok edecektir. Bu avantajı da koruyabilmek için yapılan işlemler ise bu konudaki algoritmaları oldukça karmaşıklaştırmıştır. Sıradan

bir ikili ağaçta ise bu denge ise ağaca yerleştirilen elemanların özelliğine göre değişecektir.

Örneğin elemanlar büyüklük sırasına göre yerleştirilirse oluşan ağaç sırasal bir yapıya sahip olacaktır. Aşağıdaki şekilde bu durum gösterilmiştir. Bu durumun oluşabilmesi için 1,2,3,4 elemanlarının sırayla ağaca yerleştirilmesi gerekmektedir. Bu durumun olusmasını daha detaylı incelemek için 9.haftada tanımlanan ikili ağaca yerleştirme (insert) işlevini inceleyebilirsiniz. Verilerin zaten sıralı olduğu durumda aşağıda da olduğu gibi yerleştirilen eleman daha önceki elemanların tümünden büyük olacağı için daima ağacın en sağ tarafına yerleştirilecektir. Bu durum ikili arama ağacı için en kötü durum olarak karşımıza çıkmaktadır. Bu yapıya sahip bir ağaçta ise arama işlemi sırayla arama (sequential search) yöntemiyle aynı performansa sahip olacaktır.

Ancak bu durumun oluşması düşük bir olasılıktır ve rastgele daüğılmış verilerden oluşturulan bir ağaçta ortalama arama süresi lg(N+1) olacaktır. Bu performans yine her ağaç yapısının oluşmasının aynı olasılığa sahip olduğu varsayımından hareketle bulunmaktadır.

12.4. Doğrama (Hashing) Yöntemi:

Arama yöntemleri içinde doğrama (hashing) yöntemi kullandığı farklı yaklaşımla ve performans açısından gösterdiği önemli başarıyla ön plana çıkmaktadır. Daha önceki arama yöntemlerinde elemanlar belirli bir veri yapısına birtakım özellikler kullanılarak yerleştirilmekte, arama işlemi de bu özellikler kullanılarak kolaylaştırılmaktaydı. Örneğin ikili arama ağacının özelliği sayesinde arama işleminde birçok elemanı kontrol etmeden $ln(N+1)$ (N eleman sayisi) adımda aranan verinin o ağaçta bulunup bulunmadığını belirleyebiliyorduk. Doğrama (hashing) yönteminde ise elemanların veri yapısına nasıl yerleştirileceği özel bir yöntemle belirlenerek, bu yöntem sayesinde bir elemanın o veri yapısında bulunup bulunmadığının tek bir adımda bulunabilmesi amaçlanmıştır. Böyle bir yöntemin kullanilabileceği uygun bir örnek 1'den N,e kadar tamsayılarla anahtarlanmış kayıtların N büyüklüğünde bir diziye yerleştirilmesi olabilir. Burada her anahtarın dizide kendi değerine eşit indekse sahip pozisyona yerleştirilmesiyle istenen özellik sağlanabilir. Böylelikle örneğin k anahtar değerine sahip bir kayda sadece dizinin k'inci pozisyonunda bulunan anahtarın gösterdiği elemana bakarak tek bir adımda ulaşmak mümkün olacaktır. Bu yapıyla aranan elemanın dizinin neresinde olduğunu belirlemeye çalışma yükü tek adıma indirilmiş olacaktır. Yani aradığımız

elemanları veri yapımıza ekleme ve veri yapımızdan çıkarma sadece birer adımda gerçekleştirilebilecektir.

Ancak birçok uygulamada kayıtlarımızın anahtarları tamsayı olmayacaktır veya tamsayı oldugu durumda ise sayıların aralığı (range) dizimizin veya elemanlarımızı saklayacağımız tablonun indeks aralığı ile aynı olmayabilir. Yani anahtar olarak kullandığımız tamsayılar ardarda ve sıfırdan başlayıp dizi boyuna kadar düzenli bir şekilde artan sayılar olmak zorunda değildir. Doğrama (hashing) tekniği genel olarak bütün bu durumlarda bir fonksiyon aracılığı ile anahtar değerlerini belirli bir tablo adresine (dizi indeksine) eşleyen ve kayıtların yukarıda anlatıldığı gibi tek adımla tabloya yerleştirilmesini ve alınmasını sağlayan yöntem olarak tanımlanabilir.

Mükemmel durumda kullanılan doğrama fonksiyonunun (hash function) her eleman için başka elemanlarınkinden farklı bir adres ataması gerekmektedir. Ancak birçok uygulamada böyle mükemmel bir fonksiyon bulabilmek imkansızdır. O yüzden doğrama tekniği kullanıldığında bazı elemanların aynı pozisyonlara atanması ve çakışmaların (collision) oluşması oldukça sık görülen bir durumdur. Böyle durumlarda ise çeşitli teknikler kullanarak bu çakışma durumlarında veri kaybının önlenmesi ve verimliliğin korunması sağlanmaya çalışılmaktadır. Yani doğrama yönteminde iki önemli işlev yerine getirilmelidir. Verilen anahtarların özelliklerine uygun, çakışma sayısını en azda tutabilecek bir fonksiyonun belirlenmesi ve çakışma olduğu durumda yapılacak işlemler. Şimdi sırasıyla bu iki önemli işlevi detaylı bir biçimde inceleyeceğiz.

12.4.1. *Doğrama (Hashing) Fonksiyonları:*

Doğrama fonksiyonlarğnğ belirlemek için çok çeşitli yöntemler kullanmak mümkündür. Bu yöntemleri incelerken anahtar değerlerinin tamsayı olduğunu kabul edeceğiz. (Dizge-string ya da başta türdeki veriler de oldukça kolay bir şekilde tamsayıya çevrilebilir.) Bu yöntemlerden hemen hemen en çok kullanılanı bölme (division) yöntemidir. Bu yöntemi :

$$H(key) = key \bmod m$$

Seklinde tanımlamak mümkündür. Burada *m* sayısı elemanların yerleştirileceği dizinin büyüklüğünü göstermektedir. Bu sayının asal bir sayı olması elemanların dağılımını iyileştiren bir faktör olacaktır. Aslında bu fonksiyonun tanımlanmasında karşımıza bellek ve hız ikilemi çıkmaktadır. Eğer sinirsiz belleğe sahip olsak hiç *mod* işlemini kullanmadan direkt anahtar değerini bellek adresi olarak kullanır ve hiç çakışma olmayacağı için her elemana tek bir adımda erişimi garantilemiş olurduk. Ancak bellek sınırı bizi kullandığımız alanı daraltmaya itmektedir. Eğer kullandığımız toplam alan tam elemanların sığacağı kadar büyüklükte (eleman saysısna eşit

uzunlukta tablo) olursa bu durumda yer olarak en verimli duruma ulaşmış oluruz. Ancak bu durumda da çakışma sayısı artacağı için hızdan bir kayıp söz konusu olacaktır. Bu yüzden kullanılan alan ve fonksiyon problemin önemine ve var olan kaynaklara bağlı olarak çok çeşitli şekillerde tanımlanabilir.

Örneğin elimizde 1 ile 400 aralığında değişen değerlerle anahtarlanmış yüz adet kayıt bulunsun. Bu kayıtları bir tabloya yerleştirmek için

H(key) = key mod 101

Fonksiyonunu kullanabiliriz. Bu fonksiyon bize yer açısından oldukça verimli bir durum sağlayacaktır, ancak bu fonksiyonla *101, 202, 303* veya *100, 200, 300, 400* gibi değerlerin aynı adrese atanacagına dikkat ediniz. (Örnekler çoğaltılabilir) Bu çakışma sayısını azaltmak için yukarıda anlatıldığı gibi kullanılan dizinin boyunu arttırabiliriz. Örneğin dizinin boyunu iki katına çıkardığımızda çakışma sayısı yarıya düşecektir ama yine bu bize kullanılan bellek açısından bir verim düşüşü de yaratacaktır.

Bir önceki bölümde kullanılan anahtar değerlerinin sayı olmadığı durumlar da bulunduğunu belirtilmişti. Birçok uygulamada kayıtların belirli isimler veya başka dizgeler (strings) kullanılarak da anahtarlanması mümkündür. Bu durumda bu dizge değerleri karakterlerin ASCII kodları kullanılarak sayısal değere çevrilebilir. Bu çevirimde karakterlerin ASCII değerlerini tek tek toplamak veya çarpmak gibi yöntemler izlenebilir.

Kullanılan doğrama fonksiyonunun (hash function) belirlenmesi aslında çok çeşitli yöntemlerle gerçekleştirilebilir. Yukarıda anlatılan bölme metodu en çok kullanılan yöntemdir. Ancak uygulamanın özelliklerine göre bazen anahtar değerinin belirli rakamlarının belirleyici olması durumunda bu rakamların seçilmesi, anahtar değerinin ikili gösteriminin kullanılması gibi birçok yöntem bu işlemde kullanılabilir.

12.4.2. Çakışmalar (Collisions):

Daha önce de belirtildiği gibi hiçbir doğrama fonksiyonu mükemmel bir fonksiyon olmayacaktır ve belirli elemanların aynı adrese atanmasıyla bazı çakışmalar doğacaktır. Bu yüzden bu tekniğin gerçekleştiriminde bu çakışma durumları da göz önüne alınmalı ve bir çakışma oluştuğunda yeni yerleştirilen elemanın başka bir pozisyona aktarılması sağlanmalıdır. Bu işlem için de birçok değişik yöntem izlenebilir. İzleyen iki alt bölümde bu yöntemlerden yine çok kullanılan ikisi anlatılmıştır.

12.4.2.1. Ayrı Ayrı Zincirleme (Separate Chaining):

Bu yöntem problemi aşmak için en temel yöntemlerden birisidir. Bu yöntemle kayıtlarımızın anahtarlarını yerleştirdiğimiz dizinin her bir

elemanı bir bağlantılı liste (linked list) olarak tasarlanmakta ve eğer herhangi bir dizi pozisyonunda bir çakışma olursa burada yer alacak elemanlar bir liste şeklinde aynı pozisyonda tutmaktadır. Aşağıdaki şekilde gösterilen örnekte karakterlerle anahtarlanmış belirli kayıtların bir doğrama fonksiyonuyla elde edilen tablo adresleri ve daha sonra da bu elemanların 11 elemanlı bir diziye yerleştirilmeleri sonucu elde edilen yapı gösterilmiştir.

Anahtarlar	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
Tablo adresleri	1	8	5	1	7	3	8	9	3	7	5	2	1	2	5	1	5

Ayrı ayrı zincirleme yöntemini kullandıığımızda çakışmalar sonucu dizinin elemanları olan bağlantılı listeler birden fazla elemana sahip olacaktır. Bu durumda bir elemana erisilmesi için önce o elemanin adresi dograma fonksiyonu araciligi ile bulunacak daha sonra da bu adresteki bağlantılı liste elemanları tek tek kontrol edilecektir. Eğer belirli adreslerde çok fazla çakışma gerçekleşirse bu adresteki bağlantılı listenin boyunun artacağına, bunun da eleman ekleme ve erişme işlemlerinde performans düşüşüne yol açacağına dikkat ediniz.

12.4.2.2. ***Doğrusal Yerleştirme (Linear Probing):***

Çakışma durumunda kullanılabilecek bir başka temel yöntem de doğrusal yerleştirmedir. Bu yöntemle, tablonun herhangi bir pozisyonunda çakışma doğarsa yerleştirilecek yeni eleman tablonun bir sonraki pozisyonuna yerleştirilmektedir. Eğer bu pozisyonda zaten bir eleman varsa boş bir pozisyon bulunana kadar tabloda ilerlenmektedir. Bu durumda tabloda bir eleman arama işlemi de söyle gerçekleştirilecektir:

i)Doğrama fonksiyonu aracılığı ile elemanın hangi adreste olabileceğini belirle.
ii)Eğer eleman bu adreste yok ise ve bu adrese başka bir eleman daha önceden yerleştirilmiş ise boş bir pozisyona rastlayana veya elemanı bulana kadar sonraki pozisyonları kontrol et.

Aradığımız eleman daha önceden tabloya yerleştirilmiş ve adresi dolu olduğu için, doğru pozisyonundan sonra yer alan ilk boş pozisyona kaydırılmış olabilir. Yukarıdaki boş pozisyona kadar arama işlemi böyle bir olasılık var olduğu için gerekmektedir.

Bu yöntemi kullandığımızda kullanılan tablo boyunun en az eleman sayımız kadar olması gerektiğine dikkat ediniz. Ayrı ayrı zincirleme (separate chaining) yönteminde tablonun bir elemanına birden fazla eleman bağlantılı liste sayesinde yerleştirilebildiğinden bu yöntemde

eleman sayımızdan daha ufak bir tablo kullanmamız (performansı düşürecek olsa bile) mümkündü, ancak doğrusal yerleştirme yönteminde her tablo pozisyonunda sadece tek bir eleman bulunacağı için tablomuzun büyüklüğü en az eleman sayımız kadar olmalıdır.

Doğrusal yerleştirme yönteminin önceki yönteme göre bir dezavantajı bulunmaktadır. Eğer sadece bir gurup anahtar arasında çakışma oranı yüksekse, bu durum ayrı ayrı zincirleme yönteminde tablonun bir elemanının büyük bir listeye sahip olmasına ve bu elemanların erişim performansının düşmesine yol açabilir. Ancak bu durum diğer elemanların erişim performanslarını etkilemeyecektir. Doğrusal yerleştirme yönteminde ise çakışma durumunda elemanların başka elemanların pozisyonlarını işgal etmesine izin verildiği için herhangi bir eleman kümesinde yer alan çakışmalar zincirleme olarak bu küme dışında yer alan başka elemanlarla da çakışmalar doğuracaktır. Bu durumda ise birçok eleman tablodaki gerçek bulunması gereken yerden başka pozisyonlara kaymış olacak ve bütün elemanlar için erişim ve yerleştirme performansları düşecektir.

12.5. Doğrama Metodunun Gerçekleştirimi:

Aşağıda doğrusal yerleştirme metodunu kullanarak gerçekleştirilen bir doğrama yapısı tanımı (hash.c) verilmiştir.

```
struct Node
  {
     int key;
     char info;
  };
struct Hash
  {
    int m;
    struct Node *a;
  };
```

```
struct Hash *create_hash_table(int sz,struct Hash *hx)
    {
      int i;
      hx=(struct Hash *)malloc(sizeof(struct Hash));
      hx->m=sz;
      hx->a=(struct Node *)malloc(sz*sizeof(struct Node));
      for (i=0;ia)[i]).info='\0';
      return hx;
    }
int hashfunc(int v,int sz)
      { return v % sz;}


 char search(int v,int sz,struct Node *a)
    {
          int x=hashfunc(v,sz);
          while ( (a[x].info!='\0' ) && (v!=a[x].key) )
              x=(x+1) % sz;
```

```
            return a[x].info;

        }

void insert(int v, int sz,char info,struct Node *a)

        {
```

Yukarıdaki gerçeklestirimde anahtar türü olarak tamsayılar (integers) kullanılmıştır. Yaratılan dizide saklanan elemanların türü ise *char* olarak belirtilmiştir. Kolaylıkla bu türü bir sınıf veya başka bir temel veri türü olarak tanımlayıp gerçekleştirimi başka veri türleri için de kullanmak mümkündür. Tanimlanan doğrama fonksiyonu ise basitçe tablonun büyüklüğünü *(sz)* kullanarak *mod* işlemiyle anahtar değerini tabloda bir adrese eşlemektedir. Yine anahtarın türüne göre bu işlevi değişik şekillerde tanımlamak mümkündür. Arama ve yerleştirme işlevleri de daha önceki bölümde anlatıldığı gibi önce elemanın ilk tablo adresini hesaplamakta daha sonra o adresi kullanarak tabloda belirlenen uygun adrese göre aranan elemana erişmekte veya yeni elemanı yerleştirmektedir. Aşağıdaki bağlantıyı kullanarak bir diziye çeşitli karakterleri doğrama yöntemini kullanarak yerleştiren uygulamayı inceleyebilirsiniz.

ALIŞTIRMALAR

Alıştırma 1: Girdi olarak verilen bir n sayısı için n X n'lik 1'den n*n'e kadar bütün sayıları tek satırlar düz, çift satırlar ters olarak yazan bir program yazınız. Örneğin n=5 için çıktı şöyle olacaktır:

```
1    2    3    4    5
10   9    8    7    6
11   12   13   14   15
20   19   18   17   16
21 22 23 24 25
```

Alıştırma 2: Girdi olarak verilen bir n sayısı için 1'den n'e kadar bütün sayıları üçgen biçiminde, bir satır soldan, bir satır sağdan sıralanmış ve her satırda eleman sayısını bir arttırarak yazan bir program yazınız.

Örneğin n=15 için çıktı şöyle olacaktır:

```
                                                1
                                3               2
                        4               5       6
                10              9               8       7
 11 12 13 14 15
```

Alıştırma 3 : Verilen nXn'lik bir 2 boyutlu dizinin diagonallerinin yerlerini değştiren bir program yazınız.

Örneğin

```
1   2   3   4
5           6           7           8
9           10          11          12
13  14  15  16
```

dizisi girdi olarak verilirse bu diziyi aşağıdaki forma çevirip, ondan sonra çıktıya yollamanız gerekiyor (diagonaller yukarıda ve aşağıda koyu renk ve yatık olarak belirltilerek değişiklikler gösterilmiştir).

```
 4          2          3          1
        5          7          6          8
        9          11         10         12
16          14          15          13
```

Alıştırma 4: Verilen iki tamsayı dizisinin elemanlarının değerlerini karşılıklı toplayarak yeni bir dizi üreten bir program geliştiriniz. Bunun için yazdığınız bir fonksiyona girdi olarak gelen 2 dizi ve çıktı olacak diziyi; ayrıca girdi olark gelen dizilerin boyutlarını parametre olarak geçirin. Fonksiyon tamamlandığında üçüncü dizinin elemanlarını oluşturarak ana programa dönmeli. Örneğin 3 elemanlı ve değerleri 1, 2, 4 olan bir dizi ile 5 elemanlı ve değerleri 9, 8, 7, 6, 5 olan bir dizi toplandığında 5 elemanlı ve değerleri 10, 10, 11, 6, 5 olan yeni bir dizi üretilmeli.

Alıştırma 5: Verilen iki tamsayının değerleri arasındaki sayıların toplamını döndüren bir fonksiyon yazın. Bu fonksiyonu ana programınız aracılığı ile değişik girdi değerleri için test edin. Örneğin fonksiyonunuz 3 ve 7 değerleri ile çağırıldığında 3+4+5+6+7 = 25 değerini döndürmeli. Bu fonksiyonu yukarıda tanımlanan power fonksiyonunu da kullanarak geliştirip verilen bu sayıların 3. parametrede belirtilen üslerinin toplamını bulacak şekilde ge□liştirin (örneğin 3'den 7'ye kadar olan sayıların karelerinin toplamını, 5. kuvvetlerinin toplamını vs bulmaya yarayacak şekilde).

Alıştırma 6: Verilen bir dizinin en büyük, en küçük elemanları ile ortalama değerini bulup parametreleri aracılığı ile aa programa döndüren bir fonksiyon geliştiren ve yazacağınız ana programla bu fonksiyonu test edin.

Ders 4'teki örnek programa aşağıdaki fonksiyonları ekleyiniz:
- Numarası verilen öğrencinin kullanıcı tarafından tüm bilgilerini değiştirme fonksiyonu
- Yaşı verilen tüm öğrencilerin hem yaşını hem yılını bir arttıran fnksiyon

Alıştırma 8:

f(1) = 1
f(2) = 1
f(n) = f(n-1) + f (n div 2)

Olarak tanımlanan f işlevinin verilen n icin değerini bulan özyinelemeli ve özyinelemesiz birer program yazın.

Alıştırma 9:

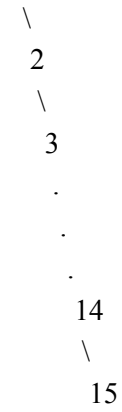
Simulasyon örneğinde verilen programı oluşan kuyruğun en uzun halini ve de kaçıncı müşterinin en çok kuyrukta bekleyeceğini belirleyen bir hale dönüştürünüz.

Cevap :

i) n elemanı küçükten büyüğe doğru ağaca eklediğimizde istenilen ağaç
oluşacaktır. Yapı aşağıdaki örnek gibi olacaktır. Burada 1'den 15'e kadar
tamsayılar küçükten büyüğe doğru ağaca yerleştirildiğinde ağaç sağa
doğru gelişecek ve sıralı ve sıra önceli taramaları aynı olacaktır.

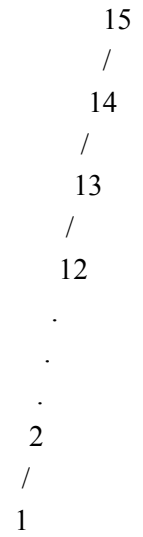
örnek:

1



ii) Bunun için elemanların büyükten küçüğe yerleştirilmesi gerekir. Oluşacak ağaç yapısı aşağıdadır.

örnek:



iii) Sıra önceli taramada bir elemanın değeri o elemanın iki çocuğundaki değerlerden önce, sıra sonralı taramada ise bir elemanın değeri o elemanın iki çocuğundaki değerlerden sonra taranmaktadır. Bu yüzden bir herhangi bir elemanın iki taramada da aynı yerde çıkabilmesi için sağ ve

sol çocukları bulunmaması gerekir. Bunun ağaçtaki tüm elemanlar için

geçerli olması ancak tek elemanlı ağaçta mümkün olmaktadır. Bu yüzden

birden fazla elemanlı ağaçlarda bu iki taramanın aynı diziyi vermesi olanaksızdır.

Alıştırma 10:

Derste tanımlanan "insert" fonksiyonunu listeyi daima sıralı bir halde oluşturacak bir biçime dönüştürünüz. Listemizin elemanlarını sayılar oluşturduğu için sıralamayı sayıların küçükten büyüğe sıralanmış biçimde listede yer alması olarak düşünebilirsiniz. Böyle bir sıralamayı oluşturmak için insert fonksiyonu verilen elemanı listenin başı yerine liste içinde bir arama gerçekleştirerek uygun yere yerleştirmelidir. "Append" fonksiyonuna ise gerek kalmamaktadır.

7.Ders Alıştırma yanıtı: Bu bağlantıdan erişeceğiniz programda kuyruk sınıf tanımı kullanılmış, ayrıca 7.Ders simulasyon apletinde yer alan kod bir fonksiyona çevrilmiştir. Bu fonksiyon içinde ufak bir bölüm eklenerek en çok sırada bekleyecek müşterinin kaçınçı müsteri olduğu ve ne kadar bekleyeceği belirlenmiştir. Fonksiyonda altı müşteriden oluşan bir örnek kullanılmıştır. last değiskeni ve time_req dizisi içinde yer alan değerleri değistirerek değişik örnekler üzerinde çalışabilirsiniz. Kullanılan yöntemde time değiskeni her arttığında yeni bir müşteri geldigi için, her artışta yeni müsterinin ne kadar bekleyeceği hesaplanmıştır. Bu da önceki müşterinin bekleme süresi artı önceki müsterinin ne kadar zaman servis alacağı ve yeni müsteri geldiğinde bir birimlik servis verilmiş olacaği için eksi bir olarak hesaplanmaktadır.

Alıştırma 11:

i) Sıralı (inorder) ve sıra önceli (preorder) taramalar sonucu elde edilen eleman dizileri aynı olan n elemanlı bir ikili tarama ağacı tasarlayınız.

ii) Sıralı (inorder) ve sıra sonralı (postorder) taramalar sonucu elde edilen eleman dizileri aynı olan n elemanlı bir ikili tarama ağacı tasarlayınız.

iii) Birden fazla elemana sahip bir ikili arama ağacının sıra önceli (preorder) ve sıra sonralı (postorder) taramalarla elde edilen eleman dizilerinin neden hiçbirzaman aynı olamayacağını açıklayınız.

8.Ders Alıştırma yanıtı: Bu alıştırmada gerçekleştirilen "insert" fonksiyonunda, yerleştirilecek elemanın değeri var olan elemanlarla karşılaştırılarak elemanın doğru yeri belirlenmektedir. Ortaya çıkabilecek durumlar, listenin boş olması veya dolu olması, dolu olması durumunda elemanın listenin başına, sonuna veya ortalarda biryere eklenmesi, if-else yapıları kullanılarak ayrı ayrı ele alınmıştır.

Alıştırma 12:

i) Birbirine eşit N adet elemanın sıralanması için hızlı sıralama algoritması kaç adet işlem yapacaktır.

ii) Hızlı sıralama algoritması ile dizide yer alan en büyük elemanın yeri en fazla kaç defa değiştirilecektir.

iii) Hızlı sıralama algoritmasını özyineleme kullanmadan gerçekleştirmek mümkündür. Bu gerçekleştirim bir yığıt (stack) kullanılarak oluşturulabilir. Bu yaklaşımda parçalama işlemiyle oluşan alt bölümlerin sınırlarını gösteren indeksler bir yığıta atılıp, daha sonra yeni bir parçalama işlemine geçilirken yığıt üzerinde bulunan ilk bölüm alınarak bu bölüm parçalama işlemine tabi tutulacaktır. Bu işlem yığıtta hiç alt bölüm kalmayıncaya kadar devam edecektir. Bu yaklaşımı kullanarak hızlı sıralama algoritmasını özyineleme olmaksızın gerçekleştiriniz.

Alıştırma 13:

Aşağıda verilen tamsayıları 9 elemanlık bir doğrama tablosuna (diziye) doğrusal yerleştirme metodunu kullanarak yerleştirdiğinizde hangi elemanın kaçıncı dizi pozisyonuna yerleştirileceğini belirleyiniz.

Sayılar : 3 33 41 5 18 21 26

EV ÖDEVİ (1) :

Verilen sondan gösterimli bir ifadenin hatalı olup olmadığını belirlemek için dersin uygulama bölümünde örnek olarak verilen programı değiştiriniz.

Aşağıda verilen iki ifade hatalıdır. Burada ilk örnekte A argümanı hiçbir operatörün argümanı olamamakta, bu yüzden de bu hata fazla argüman hatası olarak nitelendirilmektedir. İkinci ifadede ise ikinci operatör olan * işlemi sadece tek bir argümana sahip olacaktır. Bu yüzden de buradaki hata fazla operatör hatası olarak nitelendirilmiştir.

1) ABC+ (fazla argüman)
2) AB+* (fazla operatör)

EV ÖDEVİ (2):

EPS-SORT algoritması aslında INSERTION-SORT algoritmasının bir miktar degiştirilmiş halidir. INSERTION-SORT algoritmasında ileriki adımlarda eğer kücük bir sayı gelirse, bu sayının listenin baş kısımlarına eklenmesi çok uzun bir liste kısmının saga doğru kaydırılması ile sonuçlandığı için oldukça verimsiz olmaktadır. Bu durumu engellemek için listede aynı anda k (k büyük bir sayıdan başlayıp küçülerek sonuçta 1 olacak) adet ayrı liste varmış ve hepsine ayrı ayrı INSERTION-SORT uygulanıyormuş gibi calışan bir teknik geliştirilmiştir. Bu teknikte ilk aşamada n elemanlı bir listede n/2 adet alt liste var kabul edilip, onlar ayrı ayrı INSERTION-SORT ile sıralanmakta, sonra listeler ikişer ikişer birleştirilerek n/4 adet liste üzerinde aynı işlem yapılmakta ve her aşamada liste sayısı yarıya inerek sonuçta tek bir liste üzerinde INSERTION-SORT işlemi yapılmaktadır. Aynı anda bir arada bulunan r adet liste varsa bu listelerin elemanları asil listede birbirlerine r uzaklıkta yer almaktadırlar.

örneğin:

Elimizde 8 elemanlı [6,7,4,1,8,2,5,3] listemiz varsa ilk olarak 4 alt liste yani [6,8], [7,2], [4,5] ve [1,3] listeleri INSERTION-SORT ile

sıralanmakta ve tüm liste [6,2,4,1,8,7,5,3] haline gelmekte. 1. alt listenin ana listedeki 1. ve 5., ikinci alt listenin 2. ve 6., ... elemanlardan oluştuğuna dikkat ediniz. Yani index artışı 4'tür (liste sayısı). Ikinci aşamada alt liste sayısı 2'ye inmekte ve [6,4,8,5] ve [2,1,7,3] listeleri kendi aralarında INSERTION-SORT ile sıralanmakta ve tüm liste [4,1,5,2,6,3,8,7] haline gelmektedir. Bu sefer de 1. listedeki elemanlar ana listedeki 1., 3., 5., ve 7., 2. listedeki elemanlar ise gene asıl listedeki 2., 4., 6., ve 8. elemanlardan oluşmustur. Son aşamada ise tüm listeye INSERTION-SORT uygulanarak liste [1,2,3,4,5,6,7,8] haline dönüşerek sıralanma işlemi tamamlanmaktadır.

Sizden istenen INSERTION-SORT algoritmasını bu şekilde degiştiren EPS-SORT algoritmasını C ile yazarak tamsayı (integer)'lar için çalıştırmanız. Listenin uzunluğu işlemlerin kolaylığı açısından 2'nin üssü kabul edilecektir ve en cok 128 olacaktır. Ancak asıl listeyi alt listeler halinde işlerken yeniden diziler açmadan sadece bir tek dizi üzerinde işlemlerinizi yapmanız gerekmektedir. Yani örneğin 1., 3., 5., ve 7. elemanlardan oluşan alt liste üzerinde INSERTION-SORT işlemini uygulayabilmek için sadace asıl dizide bu elemanlar üzerinde normal INSERTION-SORT'u uygulayabilmelisiniz. Yoksa yeni diziler yaratıp verileri oradan oraya taşırsanız bütün verimlilik kaybedilir ve verimsiz bir sıralama algoritması ortaya çıkar.

SORULAR

1) void RecEm(int i)

```
{ if (i < 8) {
        i++;
        RecEm(i);
        printf("%d ",i);
        }
}
```

Yukarıdaki fonksiyon RecEm(4) ile çağrıldığında **çıktı** ne olur?

2) void RecEm(int *i)

```
{ if (*i < 8) {
```

```
            (*i)++;
            RecEm(i);
    printf("%d ",*i);
            }
  }
```

Yukarıdaki fonksiyon int k=4; RecEm(&k) ile çağrıldığında **çıktı** ne olur?

3)
```
void RecEm(int i)
    { if (i < 8) {
            i++;
            RecEm(i);
            }
    }
```
Yukarıdaki fonksiyon RecEm(-3) ile çağrıldığında **i++ satırı toplam kaç defa** çalıştırılır?

4)
```
void RecEm(int i)
    { if (i < 8) {
            RecEm(i);
            }
    }
```

Yukarıdaki fonksiyon RecEm(3) ile çağrıldığında **i<8 ifadesi toplam kaç defa** çalıştırılır?

5)
```
struct Stack *s;
s=create_stack(s);
push(s,5);
push(s,7);
printf("%d ",pop(s));
push(s,3);
push(s,5);
printf("%d %d",pop(s),pop(s));
```

Yukarıdaki program **çıktısı** ne olur?

6) 2 9 5 * 3 / + sondan gösterim (postfix) ifadesinin normal (infix) gösterimi nedir (sayılar tek basamaklıdır) ?

7) 3 2 3 2 + * – sondan gösterim (postfix) ifadesi yığıt kullanılarak gerçekleştirilirken en son hangi işlem gerçekleştirilecektir?

8) struct Stack *s; s=create_stack(s);

```
    void pushlist(int i)
        {       push(s,i);
                if (i>0) pushlist(i-1);
                push(s,i);
        }

    main();
        {       pushlist(3);
                while (!empty(s))
                        printf("%d ",pop(s));
        }
```

Yukarıdaki program **çıktısı** ne olur?

9) struct Queue *q;

```
    q=create_queue(q);
insert(q,5);
insert(q,7);
printf("%d ",del(q));
insert(q,3);
insert(q,5);
printf("%d %d",del(q), del(q));
```

Yukarıdaki programın **çıktısı** ne olur?

10) struct Queue *q;

```
    q=create_queue(q);
void insertlist(int i)
        {       insert(q,i);
```

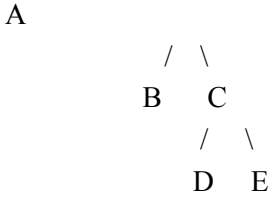
```
            if (i>0) pushlist(i-1);
            insert(q,i);
        }

    main();
        {       insertlist(3);
            while (!empty(q))
        printf("%d ",del(q));
        }
```

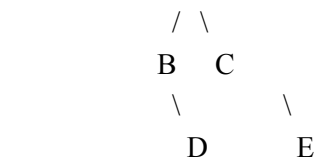
Yukarıdaki program **çıktısı** ne olur?

11) struct List l;
create_list(l);
insert(l,5);
append(l,7);
insert(l,3);
append(l,5);
remove(l,5);
Bu işlemlerden sonra listede **<u>ilk ve son elemanlar</u>** hangileridir?

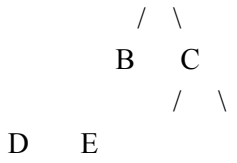
12) Aşağıdaki ağaçta hangi düğüm C düğümünün **velisidir (parent)** ?



13) Aşağıdaki ağaçta hangi düğüm C düğümünün **çocuğudur (child)** ?



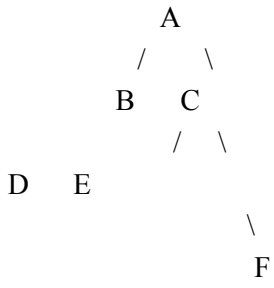
14) Aşağıdaki ağaçta kaç adet yaprak **(leaf) vardır**?



15) 9 adet düğümü bulunan bir ikili ağaçta **en fazla** kaç tane **yaprak (leaf)** bulunur?

16) 9 adet düğümü bulunan bir ikili ağaçta **en az** kaç tane **yaprak (leaf)** bulunur?

(20-22 soruları için)



17) Yukarıdaki ağacı **sıralı (inorder)** taranması hangi sonucu verir?

18) Yukarıdaki ağacı **sıra önceli (preorder)** taranması hangi sonucu verir?

19) Yukarıdaki ağacı **sıra sonralı (postorder)** taranması hangi sonucu verir?

Aşağıdaki tanımlama [23-25] no'lu sorularda kullanılacaktır:

```
int a[]= {5,2,9,7,1,6,4,3,8};
```

20) Yukarıdaki dizinin (array) değerleri (dizin 0'dan başlayarak) **seçerek sıralama (selection sort)** algoritmasındaki **iç for döngünün ilk** tamamlanmasından sonra ne olur?

21) Yukarıdaki dizinin (array) değerleri (dizin 0'dan başlayarak**) yerleştirerek sıralama (insertion sort)** algoritmasındaki **iç while döngüsünün ilk** tamamlanmasından sonra ne olur?

22) Yukarıdaki dizinin (array) değerleri (dizin 0'dan başlayarak) **kabarcık sıralama (buble sort)** algoritmasındaki **iç for döngüsünün ilk** tamamlanmasından sonra ne olur?

23)Hızlı sıralama (quicksort) algoritması ilk özyinemeli çağrıdan önce 5 2 9 7 1 6 4 3 8 listesini hangi sıraya koyar?

24)Hızlı sıralama (quicksort) algoritması 5 2 9 7 1 6 4 3 8 listesi için ilk iki özyinemeli çağırma işlemlerini **kaçar elemanlı listeler** için yapar?

25)5 2 9 7 1 6 4 3 8 sayılarının yer aldığı bir listede **sırayla arama (sequential search)** algoritması ile 4'ün bulunması için **hangi elemenlar** 4 ile karşılaştırılacaktır?

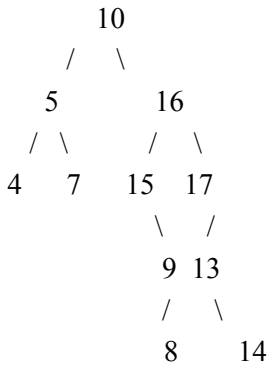
26)5 2 9 7 1 6 4 3 8 sayılarının yer aldığı bir listede **sırayla arama (sequential search)** algoritması ile 4'ün bulunması için **hangi elemenlar** 4 ile karşılaştırılacaktır?

27)1 2 3 4 5 6 7 8 9 sayılarının yer aldığı bir listede **ikili arama (binary search)** algoritması ile 4'ün bulunması için **hangi elemanlari** 4 ile karşılaştırılacaktır?

28)n elemanlı (n 2'nin üssü olmak üzere) sıralı bir listede **ikili arama (binary search)** tekniği ile aranan bir sayının bulunması için **en az kaç karşılaştırma** yapmak gerekir?

29)n elemanlı (n 2'nin üssü olam üzere) sıralı bir listede aranan eleman en sonda ise **ikili arama (binary search)** tekniği ile aranırken bulunması için **en az kaç adet karşılaştırma** yapmak gerekir?

30)n elemanlı (n 2'nin üssü olam üzere) bir **ikili arama ağacında (binary search tree)** aranan bir sayının bulunabilmesi için **en çok kaç adet karşılaştırma** yapmak gerekir?



Yukarıdaki ikili arama ağacında (binary search tree) 14'ün bulunması için hangi düğümler ziyaret edilmelidir?

31)3, 33, 41, 5, 18, 21, 26 sayılarının 10 elemanlı bir tabloya (0'dan 9'a kadar elemanlı) **ayrı ayrı zincirleme (separate chaining)** ile yerleştirilmesinde oluşacak **en uzun zincir** kaç elemanlıdır? (h(k)= k % 10 doğrama fonksiyonunun kullanıldığını varsayınız)

32)3, 33, 41, 5, 18, 21, 26 sayılarını 10 elemanlı bir tabloya (0'dan 9'a kadar elemanlı) **ayrı ayrı zincirleme (separate chaining)** ile yerleştirince tabloda **kaç adet yer boş** yer kalacaktır? (h(k)= k % 10 doğrama fonksiyonunun kullanıldığını varsayınız)

33)3, 33, 41, 5, 18, 21, 26 sayılarının 10 elemanlı bir tabloya (0'dan 9'a kadar elemanlı) **ayrı ayrı zincirleme (separate chaining)** ile yerleştirirken **hangi sayılarda çakışma (collision)** oluşacaktır? (h(k)= k % 10 doğrama fonksiyonunun kullanıldığını varsayınız)

34)3, 33, 41, 5, 18, 21, 26 sayıları 10 elemanlı bir tabloya (0'dan 9'a kadar elemanlı) **doğrusal yerleştirme (linear probing)** ile verilen sırada yerleştirilince **33 sayısı hangi indekse** yerleşir?(h(k)= k % 10 doğrama fonksiyonunun kullanıldığını varsayınız)

35)3, 33, 41, 5, 18, 21, 26 sayıları 10 elemanlı bir tabloya (0'dan 9'a kadar elemanlı) **doğrusal yerleştirme (linear probing)** ile verilen sırada yerleştirildikten sonra **1 sayısı hangi indekse** yerleştirilir? (h(k)= k % 10 doğrama fonksiyonunun kullanıldığını varsayınız)

CEVAPLAR

1) 8 7 6 5
2) 8 8 8 8
3) 11
4) i++ olursa 5 defa verildiği gibi olursa sonsuz defa.
5) 7 3 5
6) ((5*9)/3)+2 sonuç:17
7) 3-10 sonuç:-7
8) 0 0 1 2 3
9) 5 3 7
10) 3 2 1 0 0
11) ilk eleman:3 son eleman:7
12) A
13) E
14) 3 yaprak (B D E)
15) 5 tane
16) 4 tane
17) B, A, D, C, E, F
18) A, B, C, D, E, F
19) B, D, F, E, C, A
20) 1 5 9 7 2 6 4 3 8
21) 2 5 9 7 1 6 4 3 8
22) 2 5 7 1 6 4 3 8 9
23) 8'i pivot kabul ederek 5 2 7 1 6 4 3 8 9
24) 2 1 3 5 7 6 4 8 9
25) 1 2 3
26) 5, 3 ve 4
27) 1
28) $\log_2 n + 1$
29) n + 1
30) 10 16 15 5 7 13 14
31) 2
32) 5
33) 3 ve 33, 41 ve 21
34) 4
35) 7