BÖLÜM 3

DİZİLER VE BAĞLI LİSTELER

ANAHTAR KAVRAMLAR
Dizi, İşaretçi, Bağlı Liste,
Dairesel Bağlı Liste, Çift Yönlü Bağlı Liste

ÖĞRENME HEDEFLERİ

1. Tek boyutlu ve iki boyutlu dizileri tanımlayabilir ve dizi elemanlarına değer atayabilir
2. Diziler üzerinde gezinebilir ve dizi elemanları üzerinde işlem yapabilir
3. Bağlı listeleri tanımlayabilir ve kullanabilir
4. Bağlı listelerde kullanılan düğümler için bellekte dinamik olarak yer ayırabilir
5. Bağlı listeler üzerinde gezinebilir, listeye ekleme yapabilir ve listeden eleman çıkarabilir
6. Farklı bağlı liste tiplerini bilir ve onlar üzerinde işlem yapabilir

ÜÇÜNCÜ BÖLÜM

GİRİŞ

Bir problemi çözmek için kullanılan algoritmaların verimli bir şekilde çalışabilmesi gerekir. Programda kullanılan verilere en hızlı şekilde ulaşmak, veriler içerisinde arama yapmak ve veriler üzerinde gerçekleşen işlemleri başarılı bir şekilde tamamlamak istenen bir durumdur. Bu durumu sağlayabilmek ve verileri belirli kurallar çerçevesinde organize etmek için veri yapıları kullanılır. Veri yapıları aynı zamanda veri üzerinde gerçekleşen temel işlemleri tanımlar. Farklı veri yapıları farklı işlemler ve problemler için uygun olabilir. Bu bölümde diziler ve bağlı listeler anlatılacaktır. Diziler ve bağlı listeler temel veri yapılarıdır ve aynı zamanda ileriki bölümlerde yer alan farklı veri yapılarının tanımlanması ve uygulanmasında kullanılacaklardır.

1. VERİ YAPILARI

Bu bölümde öncelikli olarak veri yapıları kavramı açıklanacak, veri yapıları üzerinde gerçekleştirilen temel işlemler ifade edilecektir. Veri yapısı, bir programda kullanılan verilerin belirli mantıksal kurallar çerçevesinde belirli temel işlemleri uygulayabilmek amacıyla organize edilmiş durumu olarak tanımlanabilir. Veri yapıları doğrusal ve doğrusal olmayan olmak üzere iki grupta değerlendirilebilir. Bir veri yapısında elemanlar ardışık bir düzende yer alıyorsa bu yapılar doğrusal aksi takdirde doğrusal olmayan olarak kabul edilirler. Diziler, bağlı listeler, kuyruklar ve yığınlar doğrusal, ağaçlar ve grafikler doğrusal olmayan veri yapılarına örnek olarak verilebilirler. Veri yapılarındaki temel işlemler aşağıdaki gibi listelenebilir:

Gezinme (Traversal): Bir veri yapısındaki tüm elemanlara belirli bir sıra ve düzen içerisinde erişme işlemidir. Bu gezinme kapsamında ilgili yapının elemanları üzerinde belirli işlemler uygulanabilir. Örneğin veri yapısının elemanları ekrana yazdırılabilir.

Ekleme (Insertion): Bir veri yapısına yeni bir eleman yerleştirme işlemidir. Veri yapısının özelliğine göre ekleme işlemi farklılık gösterebilir. Burada temel amaç ekleme işleminden önce geçerli kurallara sahip veri yapısının ekleme işleminden sonra yeni bir elemanla geçerli kuralları sağlamaya devam etmesidir. Her veri yapısının elemanları arasında tanımlanmış kendine has kuralları mevcuttur. Bu kuralların ekleme ve silme benzeri işlemlerde bozulmaması gerekir.

Silme (Deletion): Bir veri yapısından bir elemanı çıkartma işlemidir. Burada silme işlemi veri yapısının özelliğine göre iki farklı şekilde olabilir. Birincisi silinen elemanın kullanılmak amacıyla veri yapısından uzaklaştırılmasıdır. İkincisi ise artık ihtiyaç duyulmayan bir elemanın veri yapısından çıkartılması işlemidir. Silme işleminin amacı ne olursa olsun bu işlem sonrasında veri yapısı geçerli kuralları sağlayan durumunu bir eksik elemanla sağlamaya devam etmelidir. Bazı durumlarda ekleme ve silme işlemlerinden sonra veri yapısının karakterine göre belirli düzeltme operasyonları gerçekleştirilebilir.

Arama (Search): Herhangi bir değerin bir veri yapısı içerisinde yer alıp almadığının araştırılması işlemidir. Aranan değer yapı içerisinde bulunduğunda, değerin yapı içerisindeki yeri (bellek adresi veya indisi) çevrilir. Ancak arama işlemi sonuncunda ilgili değer bulunamazsa arama işlemi başarısız olarak kabul edilir ve bu durumu ifade etmek için NULL veya -1 benzeri bir değer çevrilir.

Sıralama (Sorting): Bir veri yapısını oluşturan elemanların artan veya azalan düzende sıralanması işlemidir.

Birleştirme (Merge): Aynı türden iki veri yapısının birleştirilme işlemidir. Bu işlem sonucunda tek bir veri yapısı tüm elemanlara sahip olur. Bu veri yapısının geçerli tüm kuralları sağlaması beklenir.

Yukarıda ifade edilen işlemler farklı veri yapıları için farklı şekilde gerçekleştirilirler. Veri yapısının karakterine göre bazı işlemlerden sonra düzeltme işlemleri gerekebilir. Bazı veri yapılarını oluşturmak için kendileri de birer veri yapısı olarak kabul edilen diziler veya bağlı listeler kullanılır. Bazı veri yapıları hem dizi hem de bağlı listeler ile beraber gerçekleştirilebilir. Her iki yöntemin de avantajları ve dezavantajları vardır.

Her bir veri yapısı üzerinde yapılan işlemler çalışma süresi ve gerektirdiği bellek alanı açısından karşılaştırılabilir.

Öncelikli olarak bu bölümde, diziler ve bağlı listeler yapıları detaylı olarak yer alacaktır. Sonraki bölümlerde kuyruklar, yığınlar, ağaçlar ve hash tabloları anlatılacaktır.

2. DİZİLER

Bir programda, bellekte ayrılan bir alan, değişken olarak kullanılabilir. Programdaki amaçlarına bağlı olarak değişkenler, farklı veri tiplerinden tanımlanabilirler. Her veri tipinin bellekte kapladığı alan (kullandığı bayt sayısı) farklıdır. Örneğin *char* veri tipi 1 bayt kullanırken *double* veri

tipi 8 baytlık bir alana sahiptir. Kullanılan programlama diline ve programın çalıştırıldığı makineye bağlı olarak veri tipleri için ayrılan alanlar farklılık gösterebilirler. Burada ifade edilen veri tipleri kullanılarak tanımlanan değişkenler sadece bir değer tutmak için oluşturulurlar. Ancak bazı durumlarda sadece bir değişken programdaki ihtiyacı karşılamayabilir. Örneğin, bir derste sınıftaki tüm öğrencilerin aldığı final notlarına programımızda erişmek için aynı veri tipini öğrenci sayısı kadar tekrar edecek şekilde kullanabilmemiz gerekir. Diziler bu tür problemlere çözüm getirmek için tanımlanmış veri yapılarıdır. Diziler de farklı veri tipleri kullanılarak tanımlanabilirler. Bir dizi değişkeni tanımlamak için değişkenin veri tipini, ismini ve dizinin boyutunu tanımlama esnasında belirlememiz gerekir.

int a[5];

ifadesi 'a' isminde 5 elemana sahip bir tamsayı dizisi oluşturmak için kullanılır. Bir dizinin elemanları bellekte ardışık olarak yer alır. Bellekte dizinin elemanları arasında herhangi bir boşluk yer almaz.

Dizinin elemanlarına dizi tanımlanırken veya sonrasında değer atanabilir.

int a[5]={13,22,37,-4,5};

ifadesi ile 'a' dizisi tanımlanırken, dizinin elemanları atanmıştır (Şekil 3.1).

a[0]	a[1]	a[2]	a[3]	a[4]
13	22	37	-4	5

Şekil 3.1. Tek boyutlu 5 elemanlı bir tamsayı dizisi

Programın çalışması süresince [] işleci dizinin ismi ile beraber kullanılarak dizinin elemanlarına ulaşılabilir. Bir dizideki indisler 0 ile başlar. Dolayısı ile dizinin n. elemanına erişmek için (n-1) indisi kullanılır. **a[2] = 10;** ifadesi ile dizinin 3. elemanına 10 değeri atanmıştır. Aynı şekilde aşağıdaki ifade ile kullanıcıdan, dizinin 2. elemanına değer okunmaktadır.

cin>>a[1];

Dizinin ismi sabit bir işaretçi olarak kabul edilir ve program içerisinde gösterdiği adres hiçbir şekilde değiştirilemez. Bu değer aynı zamanda dizinin ilk elemanının adresidir. Bir dizinin herhangi bir elemanına erişmek için dizi alt simgeleme işleci kullanılabilir. Aynı zamanda dizi ismi ile beraber sapma (offset) gösterimi ile de ilgili dizi elemanına ulaşılabilir.

Aşağıdaki kod parçası ile dizinin 3. elemanı iki farklı şekilde yazdırılmaktadır:

cout<<a[2]<<endl;

cout<<*(a+2)<<endl;

Bir dizinin herhangi bir elemanına indis kullanarak doğrudan ulaşmak mümkündür. Bu dizilerin en önemli avantajlarından birisidir. Diziler üzerinde gezinme işlemi bir *for* veya *while* döngüsü yardımı ile basit bir şekilde gerçekleştirilebilir. Döngü değişkeni dizi indisi olarak kullanılıp ilgili dizi elemanına erişim sağlanıp istenilen işlem gerçekleştirilir.

Aşağıdaki kod parçası ile 5 elemanlı bir dizinin tüm elemanlarına 10 değeri eklenmektedir:

```
for(int i=0; ,i<=4; i++)
    a[i]=a[i]+10;
```

Bir diziye yeni bir eleman eklemek için dizinin boyutunun yeterli olması gerekmektedir. Eğer dizinin tüm elemanlarının değeri var ise yeni bir eleman ekleme işleme mümkün olmayacaktır. Ekleme işlemi dizinin ilk boş elemanına gerçekleştirilir. Dolayısıyla bir değişken ile programda dizinin ilk boş elemanının yeri tutulabilir. Bu değişken 'top' olarak isimlendirilmiş olsun. Ekleme işlemi aşağıdaki ifade ile gerçekleştirilir:

a[top]=newValue;

top++;

Değer atama işleminden sonra 'top' değişkeninin değeri bir artırılarak boş alanı göstermeye devam etmesi sağlanır.

Bazı durumlarda dizinin belirli bir indisine ekleme yapmak gerekebilir. Mevcut elemanları kaybetmemek için tüm dizi elemanları eğer boş alan var ise sağa kaydırılır. İlgili indis yeni bir eleman eklemek için hazırlanır.

Bir diziden bir eleman silmek için 'top' değişkeninin değerinin bir azaltılması yeterlidir.

top--;

Ancak silinecek elemanı dizinin son elamanı değil ise silinen elemandan başlamak üzere tüm elemanlar sağdan sola bir eleman kaydırılır. 'top' değişkeninin değeri bir azaltılır.

2.1. İki Boyutlu Diziler

Bazı problemlerde, programda kullanacağımız verinin tablo olarak ifade edilmesi daha uygun olur. Örneğin bir derste belirli sayıda öğrencinin, belirli sayıda sınavdan aldığı notlar bir tablo ile gösterilebilir (Şekil 3.2). Bu tür durumlarda iki boyutlu diziler kullanılır. İki boyutlu dizilerde, dizi değişkenimiz satır ve sütunlara sahiptir. Her iki boyut da 0 indisi ile başlar.

	S1	S2	S3
Ö1	60	70	30
Ö2	70	50	20
Ö3	45	80	25
Ö4	50	35	40

Şekil 3.2. 4 öğrencinin 3 farklı sınavdan aldıkları notlar

Şekil 3.2 de yer alan tabloyu göstermek için iki boyutlu bir dizi tanımlayabiliriz.

int d[4][3];

Tanımlanan bu dizinin elemanları, tek boyutlu dizilerde olduğu gibi tanımlama sırasında veya daha sonra kullanıcıdan okunarak atanabilir. Programın çalışma sürecinde bireysel olarak da dizinin elemanlarına değer verilebilir.

Tanımlama sırasında, aşağıdaki şekilde ilgili tablonun değerleri iki boyutlu diziye atanabilir.

int d[4][3]={{60, 70, 30}, {70, 50, 20}, {45, 80, 25}, {50, 35, 40}};

İki boyutlu bir dizinin elemanlarına satır ve sütun indisleri kullanarak erişilir. Örneğin yukarıda tanımlanan 'd' dizisinin elemanlarını ekrana yazdırmak için aşağıdaki kod parçası kullanılabilir.

```
for(int i=0; ,i<=4; i++)
{
    for(int j=0; ,j<=4; j++)
        cout<<d[i][j]<<' ';
    cout<<endl;
}
```

ARAŞTIRALIM

İki boyutlu bir dizi değişkenini bir fonksiyona gönderme işlemini araştıralım. Dizinin tüm elemanlarını veya tek bir elemanını gönderme seçeneklerini değerlendirelim.

2.2. Bağlı Listeler

Bir diziyi tanımlamak için bellekte ardışık olarak yeterli alana sahip olmak gerekir. Dizinin elemanları, elemanlar arası boşluk olmadan sıralı olarak belleğe yerleştirilirler. Ayrıca bir dizinin tanımlandığı zaman belirlenen boyutu programın çalışması sırasında değiştirilemez. Bazen programın başlangıcında değişkenler belirlenirken gerekli boyut konusunda yeterli bilgiye sahip olunamaz. Bu dezavantajları ortadan kaldırmak amacıyla bağlı listeler kullanılır. Bağlı listeler oluşturulduğu zaman önceden boyut bilgisine sahip olunması gerekmez. Ayrıca bağlı listeyi oluşturan elemanlar belleğin farklı alanlarında yer alabilirler. Birbirini takip eden bellek alanları bağlı listelerde kullanılmak zorunda değildir.

Bağlı listelerde listenin her bir elemanını göstermek için *struct* yapısı kullanılır. *struct* ile oluşturulan bu elemanlara düğüm ismi verilir. Şekil 3.3'teki *struct* tanımlaması bir bağlı listede kullanılacak düğümler için oluşturulmuştur.

```
1   struct node
2   {
3       in value;
4       struct node *ptr;
5   }
```

Şekil 3.3. Bağlı liste düğümü için struct tanımı

Burada tanımlanan *struct* ile oluşturulan düğümde değer olarak sadece bir tamsayı tutulmaktadır. Ancak farklı bağlı listelerde burada kullanılan tamsayı yerine farklı nesneler kullanılabilir. Buradaki tanımlamada yer alan *struct node* işaretçisi bağlı listedeki her bir düğüm tarafından, listede yer alan bir sonraki düğümün adresini göstermek amacıyla kullanılır. Listedeki son düğümün 'ptr' işaretçisinin değeri bir sonraki düğüm söz konusu olmadığı için NULL olarak tanımlanır.

Şekil 3.4. Tek yönlü bağlı liste

Yukarıda, tanımı verilen *struct* yapısı ile oluşturulan örnek bir tek yönlü bağlı liste verilmiştir (Şekil 3.4). Bu liste dört düğümden oluşmakta en son düğümün 'ptr' işaretçisi NULL değerini göstermektedir. Bu bağlı listeyi kullanmak için sadece ilk düğümünün adresini tutmak yeterli olur. Bu değer genellikle 'head' veya 'start' işaretçisi olarak isimlendirilir. Bağlı listelerde istenilen değere ulaşmak için tüm listedeki diğer elemanların üzerinde gezinmek gerekir. Dolayısı ile dizilerde olduğu gibi doğrudan istenilen düğüme ulaşmak söz konusu değildir. Ancak bağlı listelerin bellek kullanımı daha verimlidir. Bağlı listelerde, listedeki eleman sayısının önceden belirlenmesi zorunlu değildir. Bağlı listeye yeni bir düğüm ekleneceği zaman, bellekten düğüm için yer ayrılır. Herhangi bir düğüm bağlı listeden silindiğinde ayrılan alan belleğe geri verilir.

struct node *head=new struct node;

ifadesi ile bellekten yer ayrılır ve *struct node* cinsinden tanımlanan 'head' işaretçisine ilgili alanın adresi atanır.

Şekil 3.5'teki display fonksiyonu bir bağlı listedeki tüm değerleri sırasıyla ekrana yazdırır.

```
1    void display (struct node *head)
2    {
3        struct node *temp;
4        temp=head;
5        while (temp!=NULL)
6        {
7            cout << temp-> value <<' ';
8            temp=temp-> ptr;
9        }
10       cout <<endl;
11   }
```

Şekil 3.5. Bağlı listenin elemanlarını yazdırma fonksiyonu

Bu fonksiyon, bağlı listede yer alan ilk düğümü gösteren adresi parametre olarak alır. Fonksiyon içinde geçici bir düğüm işaretçisine (temp) bu adres değeri atanır. Bir *while* döngüsü yardımı ile tüm bağlı listenin elemanları ekrana yazdırılır. Bu döngünün içinde, **temp=temp->ptr;** ifadesi ile işaretçinin bir sonraki düğümü göstermesi sağlanır. 'temp' işaretçisi en son düğümü gösterdiği zaman bu düğümün değeri ekrana yazdırılır ve **temp=temp->ptr;** ifadesi ile 'temp' değişkeni NULL değerini alır. Bu şekilde *while* döngüsü sonlanır.

DİKKAT EDELİM

Programın çalışması sırasında, bir düğümü gösteren, NULL değerine sahip bir işaretçinin elemanlarına erişmeye çalışmak, hata üretir.

Şekil 3.6'da verilen *insert* fonksiyonu bağlı listenin başına yeni bir düğüm yerleştirir.

```
1    struct node *insert (struct node *head)
2    {
3        struct node *newNode;
4        int newValue;
5        newNode = new struct node;
6        cin>>newValue;
7        newNode->value=newValue;
8        newNode->ptr=head;
9        head = newNode;
10       return head;
11   }
```

Şekil 3.6. Bağlı listeye yeni bir düğüm ekleme fonksiyonu

DİKKAT EDELİM

Bağlı listeye yeni bir düğüm eklemek için bellekte yer ayırmak gerekir.

Fonksiyon öncelikle 'newNode' isminde yeni bir düğüm oluşturur. Bu yeni düğüm için gerekli alan *new* fonksiyonu ile ayrılır. Kullanıcıdan 'newValue' isminde yeni bir tamsayı okunur ve yeni düğümün 'value' değişkenine atanır. Yeni düğümün işaretçisi de önceki 'head' düğümünü gösterir. Yeni eklenen düğüm bağlı listenin yeni 'head' düğümü olur ve fonksiyon bu değeri çevirir. Fonksiyonun dönüş tipi bir *struct node* işaretçisidir.

Şekil 3.7'de verilen fonksiyon bir bağlı listenin sonuna yeni bir düğüm eklemektedir.

```
1    void InsertEnd (struct node *head)
2    {
3        struct node *newNode;
4        struct node *temp
5        int newValue;
6
7        temp=head
8        newNode=new struct node;
9        cin>>newValue;
10       newNode->value=newValue;
11       newNode->ptr=NULL;
12
13       while(temp->ptr!=NULL)
14           temp=temp->ptr;
15
16       temp->ptr=newNode;
17   }
```

Şekil 3.7. Bağlı listenin sonuna yeni bir düğüm ekleme fonksiyonu

Yukarıdaki fonksiyonda yine yeni bir düğüm oluşturulur. Bu düğümü bağlı listenin sonuna eklemek için listenin sonuna ulaşmak gerekir. Bu da 'temp' isminde geçici bir *struct node* işaretçisi ile sağlanır. Listenin ilk düğümü 'temp' değişkenine atanır ve bir *while* döngüsü ile listenin son düğümüne kadar ilerlenir (temp listenin son düğümünü gösterir). Sonra yeni düğüm 'temp' değişkeninin bir sonraki düğümü gösteren 'ptr' işaretçisine atanır. Tabii ki yeni düğümün 'value' değişkenine kullanıcıdan okunan değer, 'ptr' değişkenine de NULL değeri atanmış durumdadır. Bu fonksiyonda dikkat edilmesi gereken durum herhangi bir işaretçinin çevrilmemiş olmasıdır. Bağlı listenin başına yeni bir düğüm eklediğimizde 'head' işaretçisi çevrilmişti. Çünkü listenin ilk düğümü değişmişti. Ancak bağlı listenin sonuna ekleme yapıldığında 'head' işaretçisinde bir değişiklik olmadı ve bu değişken çevrilmedi.

Şekil 3.8'deki *deleteNode* fonksiyonu bir bağlı listenin ilk düğümünü silmek için yazılmıştır.

```
1    struct node *deleteNode (struct node *head)
2    {
3        struct node *temp;
4        temp=head;
5        head=head->ptr;
6        delete temp;
7        return head;
8    }
```

Şekil 3.8. Bağlı listenin ilk düğümünü silme fonksiyonu

Yukarıda verilen fonksiyonda bağlı listenin silinecek ilk düğümü geçici bir işaretçiye atanmıştır. Sonra ilk düğüm (head) bağlı listenin ikinci düğümü olacak şekilde değiştirilmiştir. Geçici işaretinin gösterdiği alan (eski ilk düğüm) *delete* fonksiyonu ile belleğe geri verilmiştir. Bağlı listenin ilk düğümü değiştiği için bu değer çevrilmiştir.

Şekil 3.9'daki *deleteEnd* fonksiyonu bir bağlı listenin son düğümünü silmek için yazılmıştır.

```
1     void deleteEnd (struct node *head)
2     {
3          struct node *temp1, *temp2;
4          temp1=head;
5          temp2=head;
6
7          while(temp1->ptr!=NULL)
8          {
9           temp2 =temp1;
10          temp1=temp1->ptr;
11         }
12
13         temp2->ptr=NULL
14        delete temp1;
15    }
```

Şekil 3.9. Bağlı listenin en son düğümünü silme fonksiyonu

Son düğümü silmek için yukarıda yer alan fonksiyonda iki farklı geçici işaretçi kullanılmıştır. *while* döngüsü ile iki işaretçi bağlı liste üzerinde arka arkaya hareket ettirilmiş ve döngü sonlandığında işaretçilerin son iki düğümü göstermesi sağlanmıştır. Son düğümü gösteren işaretçi, bu alanı belleğe geriye vermek için *delete* fonksiyonu ile silinmiştir. Bir önceki düğümün 'ptr' işaretçi değişkeni NULL değerini almış ve listeyi sonlandırmıştır. Bu şekilde bağlı listenin son düğümü silindiğinde, listenin ilk düğümünün adresi değişmemiş bu nedenle fonksiyondan herhangi bir değer çevrilmemiştir.

ARAŞTIRALIM

Bağlı listelerin başı ve sonu dışında herhangi bir noktasına yeni bir düğüm eklemeyi veya ilk ve son düğümleri dışındaki herhangi bir düğümü silmeyi araştıralım.

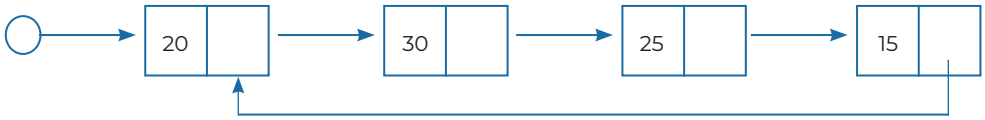
Bağlı listeler üzerinde arama işlemi ilk düğümden son düğüme kadar gezinerek sıralı bir şekilde gerçekleştirilir (Şekil 3.10). Aranan değer bulunduğu zaman ilgili düğümün adresi çevrilir. Eğer aranan değer bulunamaz ise NULL işaretçi değeri döndürülür.

```
1     struct node *find (struct node *head, int key)
2     {
3          struct node *temp;
4          temp=head;
5          if (temp->value == key)
6                return temp;
7
8          while(temp!=NULL)
9          {
10              temp=temp->ptr;
11              if(temp->value==key)
12                 return temp;
13         }
14       return NULL;
15    }
```

Şekil 3.10. Bağlı listede verilen bir değeri arama fonksiyonu

2.3. Dairesel Bağlı Listeler

Bağlı listelerde, listenin sonundaki düğümde yer alan ve bir sonraki düğümü gösteren işaretçi NULL değerini alır. Dairesel bağlı listelerde ise listenin sonundaki düğüm ilk düğümü gösterir (Şekil 3.11).



Şekil 3.11. Dairesel bağlı liste

Dairesel bağlı listeler üzerinde işlem yaparken dikkat edilmesi gerekli durum listenin sonuna ulaşılıp ulaşılmadığının kontrol edilmesidir. Bu kontrol işleminde *struct* yapısında yer alan 'ptr' işaretçisinin gösterdiği değer NULL değeri ile değil, ilk düğümün adresi ile karşılaştırılır.

Şekil 3.12'de yer alan *display* fonksiyonu bir dairesel bağlı listedeki düğümlerin değerlerini ekrana yazdırmak için oluşturulmuştur.

```
1    void display (struct node *head)
2    {
3        struct node *temp;
4        temp=head;
5
6        cout <<temp->value<<' ';
7        while (temp->ptr!=head)
8        {
9            temp=temp->ptr;
10           cout <<temp->value<<' ';
11       }
12       cout<<endl;
13   }
```

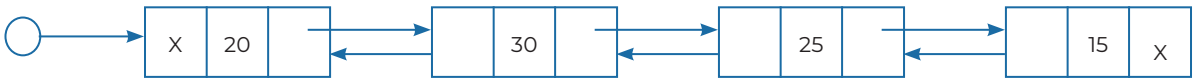
Şekil 3.12. Dairesel bağlı liste yazdırma fonksiyonu

2.4. Çift Yönlü Bağlı Listeler

Bir önceki kısımda çalıştığımız bağlı listelerde bir düğüm sadece kendisinden bir sonraki düğümü göstermekteydi. Bir düğümün aynı zamanda kendisinden önce ve kendisinden sonraki düğümleri gösterdiği bağlı listelere çift yönlü bağlı listeler denir. Çift yönlü bağlı listelerdeki düğümlerde kullanılan *struct* yapısı Şekil 3.13'te verilmiştir.

```
1    struct nodeDouble
2    {
3        int value;
4        struct node *ptrNext;
5        struct node *ptrPrev;
6    }
```

Şekil 3.13. Çift yönlü bağlı liste düğümü için struct tanımı



Şekil 3.14. Çift yönlü bağlı liste

Çift yönlü bağlı listelerde yer alan düğümler iki adet işaretçiye sahiptir. İlk işaretçi bir sonraki düğümü, ikinci işaretçi bir önceki düğümü göstermektedir (Şekil 3.14). Çift yönlü bağlı listedeki

ilk düğümün 'ptrPrev' işaretçisi NULL değerini gösterir. Aynı şekilde çift yönlü bağlı listedeki son düğümün 'ptrNext' işaretçisi NULL değerine sahiptir.

Şekil 3.15'te verilen *insert* fonksiyonu çift yönlü bağlı listenin başına yeni bir düğüm yerleştirir.

```
1     struct nodeDouble *insert(struct nodeDouble * head)
2     {
3          struct nodeDouble *newNode;
4          newNode=new struct nodeDouble;
5          int newValue;
6
7          cin>>newValue;
8          newNode->value=newValue;
9          newNode->ptrNext=head;
10         newNode->ptrPrev=NULL;
11         head->ptrPrev=newNode;
12         head=newNode;
13
14         return head
15    }
```

Şekil 3.15. Çift yönlü bağlı listenin başına düğüm ekleme fonksiyonu

Fonksiyon öncelikle 'newNode' isminde yeni bir çift yönlü bağlı liste düğümü oluşturur. Bu yeni düğüm için gerekli alan *new* fonksiyonu ile ayrılır. Kullanıcıdan 'newValue' isminde yeni bir tamsayı okunur ve yeni düğümün 'value' değişkenine atanır. Yeni düğümün 'ptrNext' işaretçisi önceki 'head' düğümünü gösterir. Önceki 'head' düğümün bir önceki düğümü gösteren 'ptrPrev' işaretçisi, yeni eklenen 'newNode' düğümünü gösterir. Yeni eklenen düğüm bağlı listenin yeni 'head' düğümü olur ve fonksiyon bu değeri çevirir. Fonksiyonun dönüş tipi bir *struct nodeDouble* işaretçisidir.

Şekil 3.16'daki *insertEnd* fonksiyonu çift yönlü bağlı listenin sonuna yeni bir düğüm eklemektedir.

```
1     void insertEnd (struct nodeDouble *head)
2     {
3          struct nodeDouble *newNode;
4          struct nodeDouble *temp
5          int newValue;
6
7          temp=head;
8          newNode=new struct nodeDouble;
9          cin>>newValue;
10         newNode->value=newValue;
11         newNode->ptrNext=NULL;
12
13         while (temp->ptrNext!=NULL)
14                temp=temp->ptrNeşt;
15
16         temp->ptrNext=newNode;
17         newNode->ptrPrev=temp;
18    }
```

Şekil 3.16. Çift yönlü bağlı listenin sonuna düğüm ekleme fonksiyonu

Öncelikli olarak yeni bir düğüm oluşturulur. Düğümün 'value' değeri olarak kullanıcıdan alınan değer atanır. Çift yönlü bağlı listenin sonuna bir düğüm eklemek için, listenin sonuna kadar ilerlemek gerekir. Listenin son düğümüne ulaştığımızda, bu düğümün 'ptrNext' işaretçisi kullanılarak listenin sonuna yeni düğüm eklenir. Yeni düğümün bir önceki düğümü olarak listenin önceki son düğümü atanır. Çift yönlü bağlı listenin ilk düğümü değişmediği için herhangi bir değer bu fonksiyon tarafından çevrilmemektedir. Fonksiyonun dönüş tipi *void* olarak belirlenmiştir.

Şekil 3.17'deki *deleteNode* fonksiyonu bir çift yönlü bağlı listenin ilk düğümünü silmek için kullanılır.

```
1    struct nodeDouble *deleteNode (struct nodeDouble *head)
2    {
3        struct nodeDouble *temp;
4        temp=head
5        head=head->ptrNext;
6        head->ptrPrev=NULL;
7        delete temp;
8        return head;
9    }
```

Şekil 3.17. Çift yönlü bağlı listenin ilk düğümünü silme fonksiyonu

Bu fonksiyonda çift yönlü bağlı listenin silinecek ilk düğümü geçici bir işaretçiye atanmıştır. Sonra ilk düğüm (head) çift yönlü bağlı listenin ikinci düğümü olacak şekilde değiştirilmiştir. Yeni ilk düğümün 'ptrPrev' işaretçisine NULL değeri verilmiştir. Geçici işaretinin gösterdiği alan (eski ilk düğüm) *delete* fonksiyonu ile belleğe geri verilmiştir. Çift yönlü bağlı listenin ilk düğümü değiştiği için bu değer fonksiyon tarafından çevrilmiştir.

Şekil 3.18'deki *deleteEnd* fonksiyonu bir çift yönlü bağlı listenin son düğümünü silmek için yazılmıştır.

```
1    void deleteEnd(struct nodeDouble *head)
2    {
3        struct nodeDouble *temp;
4        temp=head;
5
6        while(temp->ptrNext!=NULL)
7            temp=temp->ptrNext;
8
9        temp->ptrPrev->ptrNext=NULL;
10       delete temp;
11   }
```

Şekil 3.18. Çift yönlü bağlı listenin son düğümünü silme fonksiyonu

Bu fonksiyonda, bir geçici işaretçi (temp) kullanılarak çift yönlü bağlı listenin en son elemanı silinmiştir. Bir *while* döngüsü içinde 'temp' işaretçisi dizinin sonuna kadar hareket eder. Döngü sonlandığında 'temp' işaretçisi en son düğümü gösterir. Bir önceki düğümü gösteren 'ptrPrev' işaretçisi kullanılarak bir önceki düğüme erişilir ve bu düğümün 'ptrNext' işaretçi değeri NULL yapılır. Son olarak en son düğümü gösteren 'temp' işaretçisi *delete* ile silinir ve ilgili düğümün kapladığı alan belleğe geri verilir.

BÖLÜM ÖZETİ

Bu bölümde öncelikli olarak veri yapıları kavramı ve önemi anlatılmıştır. Temel veri yapıları işlemleri sıralanmıştır. Bir veri yapısı üzerinde gerçekleşmesi beklenen temel işlemler gezinme, ekleme, silme, arama ve birleştirme olarak ifade edilebilir. İlk olarak anlatılan veri yapısı dizilerdir. Dizilerin temel özellikleri çalışılmış, bir dizi değişkeninin nasıl tanımlanacağı, dizi elemanlarına ne şekilde değer atanacağı gösterilmiştir. Bir dizi üzerinde gezinme işlemi, diziye yeni eleman ekleme, diziden bir eleman silme ve diziler üzerinde arama işlemi anlatılmıştır. Dizi elemanları bellekte birbirini takip eden alanlarda yer alırlar. Ayrıca diziler tanımlandığında boyutlarının belirlenmiş olması gerekir. Dizilerden sonra çalışılan ve dizilerin yukarıda ifade edilen iki dezavantajına çözüm getiren veri yapısı, bağlı listelerdir. Bağlı listelerin boyutu liste ilk oluşturulduğu zaman belirlenmek zorunda değildir. Yeni bir eleman listeye eklendiğinde dinamik olarak bellekten yer ayrılır. Ayrıca bağlı listelerin elemanlarının bellekte ardışık alanlarda yer alma zorunluluğu bulunmamaktadır. Bağlı listeler, tek yönlü bağlı listeler, dairesel bağlı listeler ve çift yönlü bağlı listeler olarak farklı gruplara ayrılabilir. Tek yönlü bağlı listelerde her bir düğüm listede kendisinden sonra gelen düğümü gösterir. Dairesel bağlı listelerde listenin son düğümünden listenin ilk düğümüne bağlantı vardır. Çift yönlü bağlı listelerde her bir düğüm iki işaretçi içerir. Bu işaretçiler yardımı ile düğümler kendilerinden önceki ve sonraki düğümleri gösterirler.

GÖZDEN GEÇİRELİM

1. **Aşağıdakilerden hangisi temel veri yapıları işlemlerinden birisi değildir?**

   a) Gezinme
   b) Ekleme
   c) Arama
   d) Silme
   e) Çarpma

2. **Aşağıdaki dizi tanımlamalarından hangisi yanlıştır?**

   a) int a[5]={0};
   b) int a[4];
   c) int a[4]={1, 2, 3, 4, 5};
   d) int a[5]={1, 2, 3};
   e) int a[]={1, 2, 3};

3. **int a[5]={1, 2, 3, 4, 5};**

   tanımlaması verilmiş olsun. Aşağıdakilerden hangisi dizinin ilk elemanına erişmek için kullanılamaz?

   a) a[0]
   b) *a
   c) *(&a[0])
   d) a[1]
   e) a[4-3-1]

4. **int b[5]={0, 2, 4, 6, 8};**

   tanımlaması verilmiş olsun. Aşağıdakilerden hangisi dizinin tüm elemanlarını doğru bir şekilde ekrana yazdırır?

   a) for(int i=0; i<5; i++)
      cout<<b[i]<<' ';
   b) for(int i=1; i<5; i++)
      cout<<b[i]<<' ';
   c) for(int i=0; i<=5; i++)
      cout<<b[i]<<' ';
   d) for(int i=0; i<4; i++)
      cout<<b[i]<<' ';
   e) for(int i=1; i<=5; i++)
      cout<<b[i]<<' ';

5. **int c[5]={1, 3, 5, 7, 9};**

   tanımlaması verilmiş olsun. Aşağıdakilerden hangisi dizinin 3. elemanına değer atamak için kullanılamaz?

   a) cin>>a[2];
   b) a[2]=15;
   c) 20=a[2];
   d) a[2]=a[2]+5;
   e) a[2]++;

6. **Aşağıdakilerin hangisi bağlı listelerin dizilere göre üstün olduğu durumlardan biridir?**

   a) Doğrudan elemanlara erişim
   b) Boyutunun önceden belirlenme zorunluluğu olmaması
   c) Kolay arama işlemi
   d) Kolay kodlama işlemi
   e) İşlemlerin daha hızlı yapılabilmesi

7. **Aşağıdakilerin hangisi dizilerin bağlı listelere göre üstün olduğu durumlardan biridir?**

   a) Boyutunun önceden belirlenme zorunluluğu

   b) Bellek yönetimi

   c) Sabit belirlenmiş boyut

   d) Doğrudan elemanlara erişim

   e) Kolay arama işlemi

8. **Aşağıdakilerin hangisi basit çift yönlü bağlı listenin özelliklerinden birisi değildir?**

   a) Her düğümde iki işaretçiye sahip olması

   b) Liste üzerinde iki yönde gezinmeye imkan vermesi

   c) Tek yönlü bağlı listeye göre fazla bellek kullanımı olması

   d) Bir listede en az iki NULL işaretçi olması

   e) Listenin sabit boyutlu olması

9. **struct node{**

   int v;

   struct node *p;

   }*node1;

   Yukarıdaki düğüm yapısı verildiğinde, aşağıdakilerin hangisi tek yönlü bağlı listede, listenin son düğümünde olup olmadığımızı kontrol için kullanılır?

   a) if(node1.p==NULL)

   b) if(node1->p==NULL)

   c) if(node1==NULL)

   d) if(p==NULL)

   e) if(node1==p)

10. **struct node{**

    int v;

    struct node *p;

    }*node1;

    Yukarıdaki düğüm yapısı verildiğinde, aşağıdakilerin hangisi tek yönlü bağlı listede, bir sonraki düğüme ilerlemek amacıyla kullanılır?

    a) node1=node1->p;

    b) node1=p;

    c) node1=node1.p;

    d) node1.p=NULL;

    e) node1=node1+1;

Yanıt Anahtarı: 1-E 2-C 3-D 4-A 5-C 6-B 7-D 8-E 9-B 10-A

Kaynakça

Deitel, P., Deitel, H. (2017). C++ How to Program. 10th Edition. Pearson.

Thareja R. (2014) Data Structures Using C, 2nd Edition, Oxford University Press.

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.