Ardışık Taşıyıcılar

Şimdiye kadar gösterilen yapılardan dizi ile sınırlı işlemler yapılabiliyordu, ayrıca sınırlı bir kapasitesi bulunmaktaydı. Fakat yine diziyi kullanıp ekstra özellikler ekleyip daha kullanışlı bir yapıya sokulabilir. Bu ilk örnek olarak vektörler verilebilir.

Vektörler

Vektör veri yapısı yine sıralı bir yapıya sahiptir. Homojen yani aynı türden elemanları içerir. Vektörün uzunluğu n ise ilk elemanın indeksi 0 iken, son elemanın indeksi n-1'dir. Vektör genel itibarı ile aşağıdaki metotlara sahiptir.

at(i) : i numaralı indeksteki elemanı döndür.

set(i,e): i numaralı indeksteki elemanı e elemanı ile değiştir.

insert(i,e): i numaralı indekse e elemanını ekler. Dolayısıyla i numaralı indeksin dolu olma ihtimaline karşı o indeksi boşaltacaktır. Yani insert metodunda vektördeki herhangi bir eleman kaybolmaz değişime uğramaz.

erase(i): i numaralı indekste bulunan elemanı siler.

Yukarıdaki metotların hepsi, eğer verilen indeks numarası sınırların dışında ise hata fırlatacaktır. Eleman ekleme çıkarma işlemleri ile vektörde bulunan bir elemanın bulunduğu indeks numarası değişebilir.

Dizi ile Gerçekleştirim

Vektör aslına bakıldığında bir listedir. Fakat dizi ile gerçekleştirilmiş ve dizi özellikleri arttırılmış bir listedir. Vektörü gerçekleştirebilmek için önce elemanların tutulduğu biri dizi ve bu dizinin boyutu aynı zamanda dizi içerisindeki eleman sayısını tutabilmek için de ayrıca bir değişkene ihtiyaç vardır.

```
#ifndef VEKTOR_HPP
#define VEKTOR_HPP

#include "Tasma.hpp"
#define boyut 100
template <typename Nesne>
class Vektor{
        private:
                Nesne *elemanlar;
                int elemanSayisi;
…
…
```

Yukarıdaki koda bakıldığında vektör sınıfının başlık dosyası görülmekte fakat başlık dosyasında sadece metot imzaları bulunur, ama kodun devamına bakıldığında metot içeriklerinin de başlık dosyasında olduğu görülecektir. Bunun temel nedeni şablon sınıfların (yani türden bağımsız) tüm kodunun aynı dosyada bulunma zorunluluğudur. Vektör elemanları bir gösterici değişkeni içinde tutulmaktadır. Bu değişken aslında yapıcı metoda da bakıldığında bir diziyi ifade etmekte ve ilk adresini tutmaktadır.

```
Vektor()   // Yapıcı metot
{
   elemanSayisi = 0;
   elemanlar = new Nesne[boyut];
}
```

Vektör ilk oluşturulduğunda hiç elemanı olmadığı için eleman sayısı sıfıra eşitleniyor. Oluşan dizinin boyutu sabit ve 100 elemanlı.

Eleman eklemek için insert ve set metotları kullanılıyor. Aralarındaki fark insert araya eleman ekler, set ise var olan elemanı değiştirir. Her ikisinde de parametre olarak yeni elemanı const (sabit) olarak almışlardır.

```
void insert(int i,const Nesne& yeni) throw(Tasma)
{
        if(i<0 || i>elemanSayisi) throw Tasma("indeks sinirlar disinda!");
        for(int j=elemanSayisi-1;j>=i;j--) elemanlar[j + 1] = elemanlar[j];
        elemanlar[i] = yeni;
        elemanSayisi++;
}
void set(int i,const Nesne& yeni) throw(Tasma)
{
        if(i<0 || i>=elemanSayisi) throw Tasma("indeks sinirlar disinda!");
        elemanlar[i] = yeni;
}
```

Yukarıdaki iki metot ta gerçekleşme durumunda indeks sınırlar dışında hatasını fırlatmaktadır. Eleman silmek için aşağıda verilen erase metodu kullanılmaktadır. Erase metodu belirtilen indeksteki elemanı silmek için, belirtilen indeksten sonraki tüm elemanları bir eleman sola kaydırır.

```
void erase(int i) throw(Tasma)
{
        if(i<0 || i>=elemanSayisi) throw Tasma("indeks sinirlar disinda!");
        for(int j=i+1;j<elemanSayisi;j++) elemanlar[j - 1] = elemanlar[j];
        elemanSayisi--;
}
```

Herhangi bir indeksteki elemanı görebilmek için at metodu kullanılır. at metodunda iki adet const ifadesi bulunmaktadır. Bunun nedeni döndürdüğü elemanın sabit olduğunu diğer const ise metodun içinde herhangi bir değer değiştirme işleminin olmayacağını yani metodun kendisinin sabit olduğunu belirtir.

```
const Nesne& at(int i) const throw(Tasma)
{
        if(i<0 || i>=elemanSayisi) throw Tasma("indeks sinirlar disinda!");
        return elemanlar[i];
}
```

Vektör sınıfının yıkıcı metodunda delete elemanlar yerine delete []elemanlar; yazılmıştır bunun nedeni elemanlar göstericisi heap bellek bölgesinde dizi alanı göstermekte ve her indisin ayrı ayrı yıkıcı metotları çağrılması gerekiyor.

```
~Vektor()
{
   delete [] elemanlar;
}
```

Genişleyebilen Dizi ile Gerçekleştirim

Yukarıda anlatılan vektör sabit genişliğe sahipti fakat eklenen eleman sayısı bu genişliği geçebileceği durumlarda olabilir. Bu durumda yukarıdaki gerçekleştirimi biraz daha geliştirerek genişleyebilen bir yapıya sokmak gerekecektir. Bunun için sadece bir değişken ve bir metot eklemek yeterli olacaktır.

```
#ifndef VEKTOR_HPP
#define VEKTOR_HPP

#include "Tasma.hpp"

template <typename Nesne>
class Vektor{
        private:
                Nesne *elemanlar;
                int elemanSayisi;
                int kapasite;
…
…
```

Yapıcı metotta ufak bir değişiklik yapmak gerekecek.

```
Vektor()   // Yapıcı metot
{
   elemanSayisi = 0;
   kapasite=0;
   elemanlar = NULL;
}
```

Yeni eklenecek olan reserve metodu istenen miktarda dizi için yer ayıracaktır. Bu işlem için önce geçici bir alan ayırır daha sonra tüm elemanları bu geçici yere kopyalar. Daha önceki elemanların tutulduğu yer geri döndürülür. Geçici yer elemanlara eşitlenir. Kapasite boyut ile eşitlenir.

```
void reserve(int boyut)
{
    if(kapasite>=boyut) return; // O kadar yer zaten var
    Nesne *tmp = new Nesne[boyut];
    for(int j=0;j<elemanSayisi;j++) tmp[j]= elemanlar[j];
    if(elemanlar != NULL) delete [] elemanlar;
    elemanlar = tmp;
    kapasite = boyut;
}
```

Eskisine göre artık genişleyen bir yapıya sahip olan vektör ne zaman genişlemeye gerek duyar? Yeni eleman ekleneceği vakit bundan dolayı insert metodunu değiştirmemiz gerekiyor. Genişleyen bir yapıya sahip olduğu için artık taşma hatasını fırlatmasına gerek yok. Genişleme var olan boyutun iki katı kadar gerçekleşiyor.

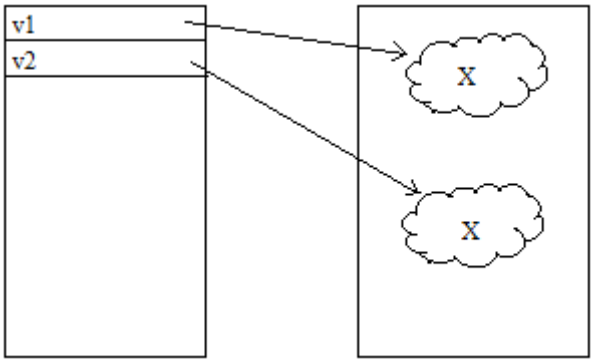
```
void insert(int i,const Nesne& yeni)
{
	if(elemanSayisi>= kapasite) reserve(max(1,2*kapasite));
	for(int j=elemanSayisi-1;j>=i;j--) elemanlar[j + 1] = elemanlar[j];
	elemanlar[i] = yeni;
	elemanSayisi++;
}
```

Copy Constructor

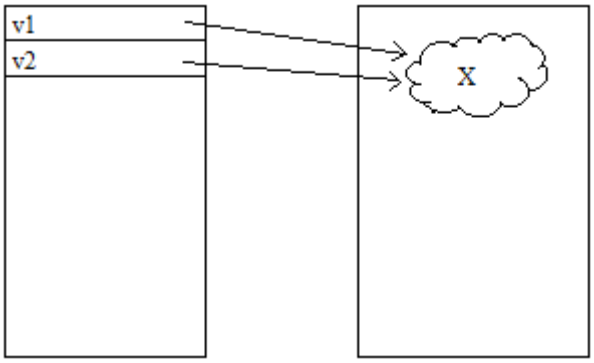
Yapıcı metot bir nesne oluşturulacağı zaman çağrılan metottur. Copy constructor ise bir nesne oluşacağı zaman bir başka nesnenin aynısı şeklinde oluş mesajını verir. Kopyalayan yapıcı metottur. Bu durumda bellekte aynı nesneden iki tane bulunmaktadır.

```
Vektor(const Vektor &sag)
{
	elemanSayisi = 0;
	elemanlar = new Nesne[boyut];
	for(int i=0;i<sag.size();i++){
		insert(i,sag.at(i));
	}
}
```

```
Vektor<int> *v1 = new Vektor<int>();
Vektor<int> *v2 = new Vektor<int>(*v1);
```




```
Vektor<int> *v1 = new Vektor<int>();
Vektor<int> *v2 = v1;
```



Listeler

Daha önce de bahsedildiği gibi vektör yapısı da aslında bir listedir. Fakat bizim burada bahsedeceğimiz listede vektöre ek olarak birkaç özellik daha ekleyip daha gelişmiş bir yapı tasarlamak. Burada anlatılacak olan listenin dizi ile gerçekleştirimi, C# ve Java gibi dillerde buna verilen özel bir isim olan ArrayList gibi bir yapı bulunmaktadır. İşte bizim burada yapacağımız yapı, ArrayList'e benzer bir yapı olacaktır.

Liste Dizi ile Gerçekleştirimi

Bağlı listede göreceğimiz düğüm yapısı burada tam anlamıyla yok, dizi ile gerçekleştirdiğimiz için dizinin her indeksinde bir eleman bulunduruyoruz. Liste şablon olarak tasarlandığı için başlık ve kaynak dosyası ayrı olamayacaktır.

```
#ifndef ARRAYLIST_HPP
#define ARRAYLIST_HPP

#include "ElemanYok.hpp"
#include "ListeBos.hpp"
#include "Tasma.hpp"

template <typename Nesne>
class ArrayList{
	private:
		Nesne *elemanlar;
		int elemanSayisi;
		int kapasite;
…
```

Yukarıdaki kod bloğunda liste dizi ile gerçekleştirildiği için ismi ArrayList olarak tanımlanmış. Alt alanlar olarak yine vektöre benzer şekilde elemanların tutulduğu bir dizi, eleman sayısı değişkeni ve dinamik boyutlu bir dizi olduğu için kapasite ayrıca değişken olarak tutulmuştur. Liste ilk oluşturulduğunda yapıcı metotta da görüldüğü gibi 0 uzunluğunda bir liste oluşturuluyor.

```
ArrayList() //Varsayılan Yapıcı metot
{
	elemanSayisi = 0;
	kapasite=0;
	elemanlar = NULL;
}
```

Bir liste başka bir listenin aynısı şeklinde oluşturulacaksa copy constructor kullanılmalıdır.

```
ArrayList(const ArrayList &sag) // Copy constructor
{
	elemanSayisi = 0;
	elemanlar = new Nesne[sag->length()];
	for(int i=0;i<sag.size();i++){
	insert(i,sag.at(i));
	}
}
```

Listenin uzunluğunu getiren ve listenin boş olup olmadığını kontrol eden metotlar. Metotların içinde herhangi bir değer değiştirme işlemi olmayacağından metotlar sabit (const) olarak tanımlanıyor.

```
int length() const
{
	return elemanSayisi;
}
bool isEmpty() const
{
	return length() == 0;
}
```

Aşağıdaki fonksiyonlar bir eleman arandığında konumunu getiren (indexOf) ve bir konumdaki elemanı getiren (elementAt) fonksiyonlarıdır.

```
int indexOf(const Nesne& eleman) const throw(ElemanYok)
{
	for(int i=0;i<elemanSayisi;i++){
		if(elemanlar[i] == eleman)return i;
	}
	throw ElemanYok("Eleman bulunamadi");
}
const Nesne& elementAt(int i) const throw(ElemanYok)
{
	if(i<0 || i>=elemanSayisi) throw ElemanYok("Eleman bulunamadi");
	return elemanlar[i];
}
```

Aşağıdaki fonksiyonlar verilen elemanı listeden silen (remove) ve bir konumdaki elemanı silen (removeAt) fonksiyonlarıdır. Şimdiye kadar bahsedilen fonksiyonlardaki yapı vektör ile aynı olup sadece fazladan özellik eklenerek daha iyi bir yapı haline getirilmiştir.

```
void remove(const Nesne& eleman) throw(ElemanYok)
{
	for(int i=0;i<elemanSayisi;i++){
		if(elemanlar[i] == eleman){
			for(int j=i+1;j<elemanSayisi;j++) elemanlar[j - 1] = elemanlar[j];
			elemanSayisi--;
			return;
		}
	}
	throw ElemanYok("Eleman bulunamadi");
}
void removeAt(int i) throw(ElemanYok)
{
	if(i<0 || i>=elemanSayisi) throw ElemanYok("Eleman bulunamadi");
	for(int j=i+1;j<elemanSayisi;j++) elemanlar[j - 1] = elemanlar[j];
	elemanSayisi--;
}
```

Listeye eleman eklemenin iki yolu bulunmaktadır. Birincisi verilen elemanı listenin en sonuna ekler. Bir diğeri verilen elemanı verilen konuma ekler.

```
void add(const Nesne& yeni) //En sona ekler
{
	if(elemanSayisi >= kapasite) reserve(max(1,2*kapasite)); //Yer yoksa 2 kat yer ayır.

	elemanlar[elemanSayisi] = yeni;
	elemanSayisi++;
}
void add(int i,const Nesne& yeni) throw(Tasma)
{
	if(i<0 || i>elemanSayisi) throw Tasma("Indeks sinirlar disinda");
	if(elemanSayisi >= kapasite) reserve(max(1,2*kapasite)); //Yer yoksa 2 kat yer ayır.
	for(int j=elemanSayisi-1;j>=i;j--) elemanlar[j + 1] = elemanlar[j];
	elemanlar[i] = yeni;
	elemanSayisi++;
}
```

Yine birçok liste uygulamasında görülebilecek olan listenin ilk ve son elemanını getiren fonksiyonlar bu ArrayList sınıfına da eklenmiştir.

```
const Nesne& first() const throw(ListeBos)
{
	if(elemanSayisi == 0) throw ListeBos("Liste bos");
	return elemanlar[0];
}
const Nesne& last() const throw(ListeBos){
	if(elemanSayisi == 0) throw ListeBos("Liste bos");
	return elemanlar[elemanSayisi-1];
}
```

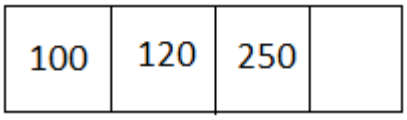
Tüm listenin temizlenmesi için dizideki her elemanı silmeye gerek yok. Yapılacak işlem eleman sayısı değişkenini ve kapasiteyi sıfıra eşitleyip heap bellek bölgesinden ayrılmış olan diziyi geri döndürmek. Böylelikle ArrayList sınıfı yeni oluşturulduğu durumuna dönecek ve liste temizlenecektir.

```
void clear()
{
	elemanSayisi = 0;
	kapasite=0;
	delete [] elemanlar;
	elemanlar = NULL;
}
```

Sınıf tasarımcısı olarak bizler, bizim sınıfı kullanacak kişilere, sınıfı ekrana yazmayı sağlayacak bir yol sunmalıyız. Bu EkranaYaz metodu olabileceği gibi daha kullanışlı ve gösterişli olabilecek << operatörünü yeniden tanımlamak ta olabilir. Böylelikle oluşan listeyi ekrana yazmak için direk cout<<liste; şeklinde bir yazım yetecektir.

```
friend ostream& operator<<(ostream& ekran,ArrayList<Nesne>& sag){
        ekran<<endl;
        for(int i=0;i<sag.length();i++){
                ekran<<sag.elementAt(i);
                if(i+1!=sag.length())ekran<<" || ";
        }
        return ekran;
}
```

```
//Test Kodu
int main(){
        ArrayList<int> *liste = new ArrayList<int>();
        liste->add(100);
        liste->add(120);
        liste->add(250);
        cout<<*liste;
        delete liste;
        return 0;
}
```



Hazırlayan
Arş. Gör. Dr. M. Fatih ADAK