Dinamik Diziler (Veri Yapıları 2. Bölüm)

Yazan: Kaan Aslan - Sebahat Ersoy
9 Ekim 2014

1. Giriş

Programlama dillerinin çoğunda diziler bir kere yaratıldıktan sonra artık büyütülemezler ve küçültülemezler. Oysa bazı durumlarda açacağımız dizinin uzunluğunu işin başında bilemeyiz. Programın çalışma zamanı sırasında onların dinamik bir biçimde büyütülmesini ya da küçültülmesini isteyebiliriz. Programın çalışma zamanı sırasında büyütülebilen ya da küçültülebilen dizilere dinamik diziler (*dynamic arrays*) denilmektedir.[1] Dinamik diziler *C*, *C++*, *Java* ve *C#* gibi dillerde temel bir veri yapısı değildir. Yani bu dillerde dinamik diziler dilin sentaksı tarafından doğrudan desteklenmezler. Bunların fonksiyonlar ya da sınıflar kullanılarak gerçekleştirilmeleri gerekir. Ancak *Perl* ve *Ruby* gibi dinamik dizilerin dilin sentaksı tarafından temel bir veri türü olarak desteklendiği diller de vardır.

2. Dinamik Dizilere Neden Gereksinim Duyulmaktadır?

Bir dizinin uzunluğu işin başında belli değilse ve birtakım olaylar sonucunda büyüyor ya da küçülüyorsa dinamik bir dizi gereksiniminin olduğunu söyleyebiliriz. Örneğin bir çizim yaparken fareyi hareket ettirdikçe elde edilen noktaları bir dizide saklayacak olalım. Kullanıcının fareyi ne kadar sürükleyeceğini bilemeyeceğimiz için noktaları yerleştireceğimiz dizinin uzunluğunu da bilemeyiz. Ya da örneğin bir metin dosyası içerisindeki belli sözcüklerin hangi satırlarda bulunduğunu bir dizi içerisinde saklamak isteyelim. Dosyanın içerisinde o sözcükten kaç tane olduğunu bilemeyiz. Benzer biçimde bir sunucu (*server*) programı yazdığımızı düşünelim. Sunucumuza bağlanan istemcilerin (*clients*) bilgilerini bir dizide tutmak isteyebiliriz. Oysa sunucumuza kaç istemcinin bağlanacağı önceden belli değildir. Örnekleri çoğaltabiliriz. Bu örneklerin hepsinde uzunluğu önceden belli olmayan ve duruma göre artıp azalabilen bilgilerin tutulması söz konusudur. İşte dinamik diziler bu tür gereksinimlerin karşılanmasında kullanılırlar.

3. Dinamik Dizilerin Algoritmik Özellikleri

Dinamik bir dizide elemana erişim sabit zamanlı (yani *O(1)* karmaşıklıkta) yapılmaktadır. Dinamik dizinin sonuna eleman ekleme ise ek maliyeti olabilen sabit zamanlı (*amortized constant time*) bir işlemdir. Bu ek maliyet dizinin büyütülmesi ya da küçültülmesi sırasında karşımıza çıkar. (Çünkü dinamik dizilerin büyütülüp küçültülmesi sırasında yeni bir alan tahsis edilip eski alandaki elemanların bu yeni alana kopyalanması gerekmektedir.) Dinamik dizilerde araya ekleme işlemi ve aradan eleman silme işlemi ise kaydırma gerektirdiği için doğrusal (yani *O(n))* karmaşıklıktadır.

İşlem	Karmaşıklık
Elemana erişim	O(1)
Sona eleman ekleme	O(1) (Ek maliyetli)
Araya eleman ekleme	O(n)
Aradan eleman silme	O(n)

4. Dinamik Dizilerin Bellek Kullanım Özellikleri

Genel olarak dinamik diziler bellekte normal dizilerden daha fazla yer kaplarlar. Çünkü dinamik dizilerin gerçekleştirilmesinde yeniden tahsisat (*reallocation*) sayısını azaltmak için gerektiğinden daha fazla alan tahsis edilmiş olarak tutulmaktadır. Bunun dışında dinamik dizilerin ardışıl alana gereksinim duymaları da bir dezavantajdır. Ardışıl tahsisatlar genel olarak belleğin bölünmesine (*fragmentation*), bölünmüş bellek de bellek kullanım veriminin düşmesine yol açar.

5. Dinamik Dizilerin Diğer Veri Yapılarıyla Karşılaştırılması

Kullanım alanı bakımından dinamik dizilerle bağlı listeler birbirlerine benzerdir. Her iki veri yapısı da uzunluğu baştan bilinmeyen, dinamik olarak büyütülüp küçültülebilen liste oluşturmak için kullanılır. Ancak dinamik dizilerde elemana erişimin çok hızlı yapılması bağlı listelere göre bir avantajdır. Bağlı listelerde belli bir elemana erişim (eğer o elemanın yerini saklamamışsak) doğrusal karmaşıklıktadır. Hem dinamik dizilerde hem de bağlı listelerde sona eleman ekleme benzer maliyetlere sahiptir. Fakat dinamik dizilerde kapasite artırımı yapılırken ek bir maliyet gerektiğini belirtelim. Araya eleman ekleme ve aradan eleman silme durumlarında -eğer bu elemanın yeri bir biçimde tutulmuşsa- bağlı listeler daha avantajlıdır. Çünkü bu durumda araya ekleme ya da aradan silme sabit zamanlı yapılabilmektedir. Halbuki dinamik dizilerde araya eleman eklenmesi ve aradan eleman silinmesi her zaman doğrusal karmaşıklıktadır. Son olarak dinamik dizilerin bir ardışıllık gerektirdiğine dikkat çekmek istiyoruz. Yukarıda da belirtildiği gibi, ardışıllık gereksinimi bölünmeye (f*ragmentation*) dolayısıyla da belleğin verimsiz kullanımına yol açan bir etmendir. Hem dinamik dizilerde hem de bağlı listelerde bir miktar bellek alanı veri yapısının organizasyonunu sağlamak için harcanmaktadır. Dinamik dizilerin gerçekleştirimlerinde –sonraki bölümde ele alınıyor- dizide tutulan eleman sayısından daha fazla bir alan boş olarak saklanır. Bağlı listelerde ise önceki elemanın sonraki elemanı göstermesi için kullanılan göstericiler ek yer kaplarlar. Durumdan duruma değişebilse de genel olarak dinamik büyütülen dizilerin daha fazla bellek kullanma eğiliminde olduğunu söyleyebiliriz. Elemanların *byte* uzunluğu ve sayıları arttıkça dinamik dizilerin bağlı listelere göre harcadığı bellek miktarı da artmaktadır.

İşlem	Dinamik Diziler	Bağlı Listeler
Elemana erişim	O(1)	O(n)
Sona eleman ekleme	O(1) (Ek maliyetli)	O(1)
Araya eleman ekleme	O(n)	O(1)
Aradan eleman silme	O(n)	O(1)
Bellek kullanımı	Ardışıl alan gerekiyor. Dizide tutulan elemandan daha fazlası için yer ayrılıyor	Ardışıl alan gerekmiyor. Fakat bağlar için gereken göstericiler ek yer kaplıyor

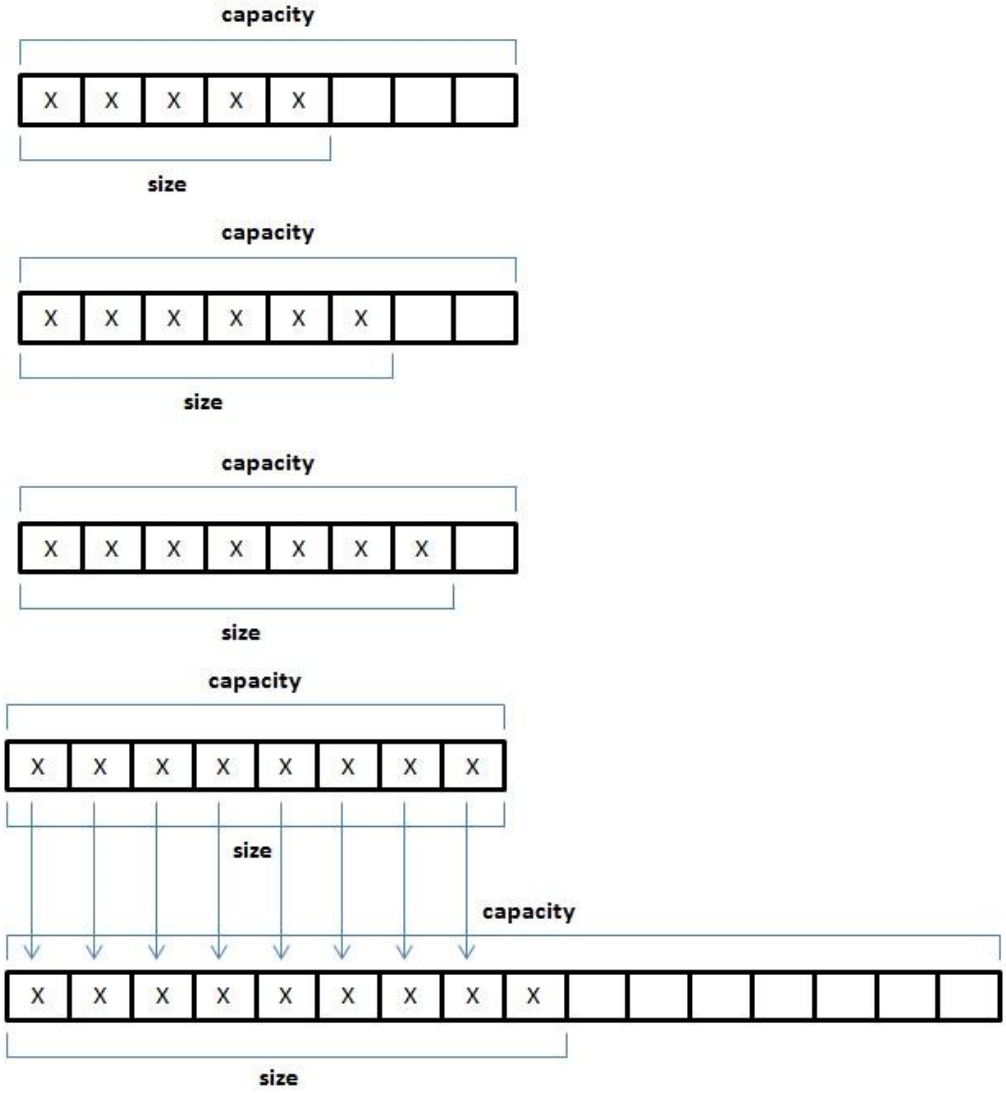
Bağlı dinamik dizilerle bağlı listelerin karşılaştırılması aşağıdaki tabloyla özetlenmektedir:

6. Dinamik Dizilerin Gerçekleştirilmesi

Dinamik diziler tipik olarak şöyle gerçekleştirilirler:

1) Başlangıçta küçük bir alan dinamik olarak *heap* üzerinde tahsis edilir. Dizide o anda kaç elemanın tutulduğu ve dizi için kaç elemanlık bir alanın tahsis edilmiş olduğu bilgisi bir yerde saklanır. Geleneksel olarak dizide tutulan eleman sayısına *size*, dizi için tahsis edilen eleman sayısına ise *capacity* denilmektedir.

2) Eleman ekleme işlemi *size* ile belirtilen indekse yapılır ve *size* değeri 1 artırılır. *size* değeri *capacity* değerine eriştiğinde *capacity* değeri daha büyük olan yeni bir dizi tahsis edilir. Eski dizideki değerler bu yeni diziye kopyalanır ve eski dizi serbest bırakılır. Artık işlemlere yeni diziden devam edilir. Örneğin dinamik dizinin içerisinde 5 eleman bulunuyor olsun. Dizi için 8 elemanlık yer ayrılmış olduğunu düşünelim. (Yani *size* = 5, *capacity* = 8 olsun.) Dinamik diziye 4 elemanı teker teker eklemek isteyelim. Oluşan durum aşağıda şekilsel olarak gösterilmektedir:



Burada önemli bir nokta, *size* değeri *capacity* değerine eriştiğinde *capacity* değerinin ne kadar artırılacağıdır. Artırım genellikle önceki değerin belirli bir katı biçiminde (tipik olarak 2 katı olacak biçimde) yapılır. Bu durumda örneğin başlangıç *capacity* değeri 1 olan bir dinamik diziye elemanlar eklendiğinde *capacity* değerleri sırasıyla 2, 4, 8, 16, 32, 64, 128, ... biçiminde olacaktır. Pekiyi, *capacity* artırımı neden önceki uzunluğun *k* fazlası biçiminde değil de önceki uzunluğun belli bir katı biçiminde yapılmaktadır? Artırımın bu biçimde yapılması ekleme işleminin karmaşıklığını ek maliyetli sabit zamanlı (*amortized constant time*) tutmak için gerekmektedir. Eğer artırım önceki miktardan *k* fazla olacak biçimde yapılsaydı toplam *n + 1* tane elemanın eklenmesi durumunda yapılacak kopyalama sayısı şöyle olurdu:

$$C(n) = k + 2k + 3k + \cdots + \frac{n}{k}\,k$$

$$= \frac{(\frac{n}{k})\,(\frac{n}{k} + 1)}{2}\,k$$

$$= \frac{n(n+k)}{2k}$$

Bu değeri *n*'e bölersek ortalama kopyalama sayısını buluruz:

$$\frac{n+k}{2k}$$

Görüldüğü gibi burada *Big O* notasyonuna göre *O(n)* karmaşıklığı söz konusudur. Eğer artırım geometrik olarak yapılırsa toplam kopyalama sayısı şöyle olur (n = 2^m olsun):

$$C(n) = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^m = 2^{m+1} - 1 \cong 2n$$

Ortalama kopyalama sayısı da şöyledir:

$$\frac{2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^m}{n} \cong 2$$

Kategori olarak bu tür karmaşıklıklara ek maliyetli sabit zamanlı (*amortized constant time*) karmaşıklık denilmektedir. Bir işlemin ek maliyetli sabit zamanlı olması demek bu işlemin genellikle sabit zamanlı olarak yapılması fakat nadiren doğrusal karmaşıklıkta yapılması demektir. Buradaki nadir koşulu geometrik artırımla sağlanabilmektedir.

6.1 Dinamik Dizilerin C'de Gerçekleştirilmesi

Dinamik dizilerin gerçekleştimi *C*'de tipik olarak *handle* sistemiyle yapılabilir. Bunun için veri yapısını kontrol eden bir *handle* alanı dinamik olarak bir yapı (*struct*) biçiminde oluşturulur. Bu veri yapısının adresi *handle* biçiminde kullanılarak diğer fonksiyonlara parametre yoluyla aktarılır. Kullanım bitince de tahsis edilmiş tüm alanlar boşaltılır. Biz burada önce tüm gerçekleştirimin kodunu vereceğiz, sonra da açıklamalar yapacağız. Örnekte dinamik dizi için *vector* terimi kulanılmıştır. Veri yapısını *vector.h* ve *vector.c* isimli iki dosya biçiminde organize ettik. *vector.h* dosyası içerisinde bildirimler, sembolik sabitler ve fonksiyon prototipleri bulunmaktadır. Fonksiyon tanımlamaları *vector.c* içerisindedir.

```
/* vector.h */

#ifndef VECTOR_H_
#define VECTOR_H_

#include <stddef.h>

#define FALSE               0
#define TRUE                1

#define DEFAULT_CAPACITY    2

/* Type Declarations */

typedef int BOOL;
typedef int DATATYPE;

typedef struct tagVECTOR {
    DATATYPE *pArray;
    size_t size;
    size_t capacity;
} VECTOR, *HVECTOR;

/* Function Prototypes */

HVECTOR CreateVectorWithCapacity(size_t capacity);
HVECTOR CreateVector(void);
void CloseVector(HVECTOR hVector);
BOOL AddItem(HVECTOR hVector, DATATYPE val);
BOOL InsertItem(HVECTOR hVector, size_t index, DATATYPE val);
BOOL DeleteItem(HVECTOR hVector, size_t index);
void Clear(HVECTOR hVector);
DATATYPE GetItem(HVECTOR hVector, size_t index);
void SetItem(HVECTOR hVector, size_t index, DATATYPE val);
DATATYPE *GetArray(HVECTOR hVector);
size_t Getsize(HVECTOR hVector);
size_t GetCapacity(HVECTOR hVector);
BOOL SetCapacity(HVECTOR hVector, size_t capacity);

#endif

/* vector.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vector.h"

static BOOL setCapacity(HVECTOR hVector, size_t capacity, size_t index);

HVECTOR CreateVectorWithCapacity(size_t capacity)
{
    HVECTOR hVector;

    if ((hVector = (HVECTOR) malloc(sizeof(VECTOR))) == NULL)
        return NULL;

    if ((hVector->pArray = (DATATYPE *) malloc(sizeof(DATATYPE) * capacity)) == NULL) {
        free(hVector);
        return NULL;
    }

    hVector->capacity = capacity;
    hVector->size = 0;

    return hVector;
}
```

```
HVECTOR CreateVector()
{
    return CreateVectorWithCapacity(DEFAULT_CAPACITY);
}

void CloseVector(HVECTOR hVector)
{
    free(hVector->pArray);
    free(hVector);
}

BOOL AddItem(HVECTOR hVector, DATATYPE val)
{
    if (hVector->size == hVector->capacity &&
                !setCapacity(hVector, hVector->capacity * 2, hVector->size))
        return FALSE;

    hVector->pArray[hVector->size++] = val;

    return TRUE;
}

BOOL InsertItem(HVECTOR hVector, size_t index, DATATYPE val)
{
    if (index > hVector->size)
        return FALSE;

    if (hVector->size == hVector->capacity &&
                !setCapacity(hVector, hVector->capacity * 2, index))
        return FALSE;

    memmove(&hVector->pArray[index + 1], &hVector->pArray[index],
                (hVector->size - index) * sizeof(DATATYPE));

    hVector->pArray[index] = val;
    ++hVector->size;

    return TRUE;
}

BOOL DeleteItem(HVECTOR hVector, size_t index)
{
    if (index >= hVector->size)
        return FALSE;

    memmove(&hVector->pArray[index], &hVector->pArray[index + 1],
            (hVector->size - index - 1) * sizeof(DATATYPE));

    --hVector->size;

    return TRUE;
}

void Clear(HVECTOR hVector)
{
    hVector->size = 0;
}

DATATYPE GetItem(HVECTOR hVector, size_t index)
{
    return hVector->pArray[index];
}
```

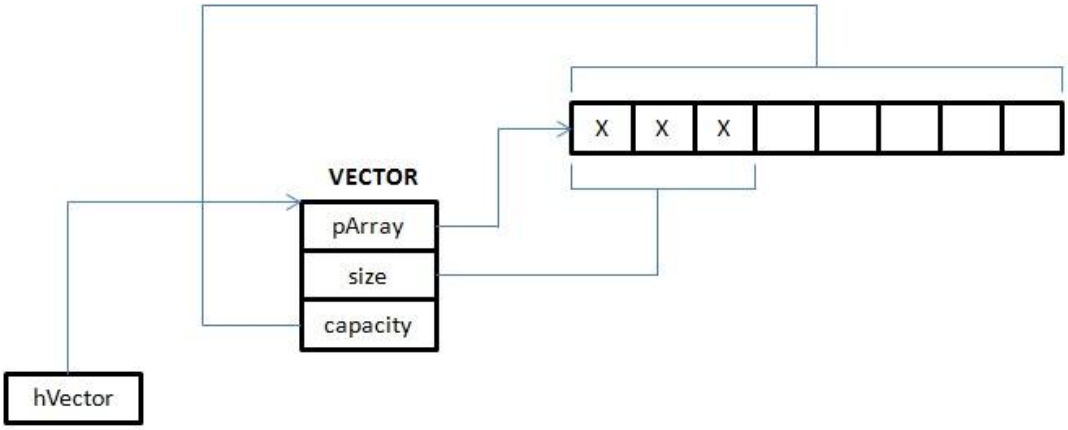
```
void SetItem(HVECTOR hVector, size_t index, DATATYPE val )
{
    hVector->pArray[index] = val;
}

DATATYPE *GetArray(HVECTOR hVector)
{
    return hVector->pArray;
}

size_t Getsize(HVECTOR hVector)
{
    return hVector->size;
}

size_t GetCapacity(HVECTOR hVector)
{
    return hVector->capacity;
}

BOOL SetCapacity(HVECTOR hVector, size_t capacity)
{
    if (capacity < hVector->size)
        return FALSE;

    if (capacity != hVector->capacity)
        return setCapacity(hVector, capacity, hVector->size);

    return TRUE;
}

static BOOL setCapacity(HVECTOR hVector, size_t capacity, size_t index)
{
    DATATYPE *pTemp;
    size_t newCapacity;

    if (index > hVector->size)
        return FALSE;

    newCapacity = capacity > DEFAULT_CAPACITY ? capacity : DEFAULT_CAPACITY;
    pTemp = (DATATYPE *) malloc(sizeof(DATATYPE) * newCapacity);
    if (pTemp == NULL)
        return FALSE;
    if (hVector->size) {
        memcpy(pTemp, hVector->pArray, index * sizeof(DATATYPE));
        memcpy(pTemp + index + 1, hVector->pArray + index,
                    (hVector->size - index) * sizeof(DATATYPE));
    }

    free(hVector->pArray);
    hVector->pArray = pTemp;
    hVector->capacity = newCapacity;

    return TRUE;
}
```

```
/* Test Kodu */

#if 1

void DispVector(HVECTOR hVector)
{
    size_t i;

    for (i = 0; i < Getsize(hVector); ++i)
        printf("%d ", GetItem(hVector, i));
    printf("\nsize = %lu, Capacity = %lu\n", (unsigned long)Getsize(hVector),
                (unsigned long)GetCapacity(hVector));
    printf("------------------------\n");
}

int main(void)
{
    HVECTOR hVector;
    int i;

    if ((hVector = CreateVector(0)) == NULL) {
        fprintf(stderr, "Cannot create vector!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        AddItem(hVector, i);
    DispVector(hVector);

    InsertItem(hVector, 5, 1000);
    DispVector(hVector);

    DeleteItem(hVector, 5);
    DispVector(hVector);

    return 0;
}

#endif
```

Örneğimizde dinamik dizi *VECTOR* isimli bir yapı ile temsil edilmiştir. *VECTOR* yapısı tahsis edilen dizinin adresini, uzunluğunu ve dolu eleman sayısını tutmaktadır:

```
typedef struct tagVECTOR {
    DATATYPE *pArray;
    size_t size;
    size_t capacity;
} VECTOR, *HVECTOR;
```

pArray dinamik dizinin başlangıç adresini, *size* dolu olan eleman sayısını ve *capacity* de dizi için tahsis edilmiş alanı tutmaktadır:



Bu örnek çizimde *size* değeri 3, *capacity* değeri ise 8'dir. Dinamik dizinin yaratılması *CreateVector* ve *CreateVectorWithCapacity* fonksiyonuyla yapılmaktadır. *CreateVector* başlangıç kapasitesini *DEAFULT_CAPACITY* değeriyle, *CreateVectorWithCapacity* ise programcının belirlediği bir değerle oluşturur. Örneğimizde *DEFAULT_CAPACITY* değerini 2 olarak aldık. Dinamik dizi yaratıldığında *size* değerinin 0 olduğunu görüyorsunuz. *AddItem* fonksiyonu dinamik diziye eleman eklemektedir:

```
BOOL AddItem(HVECTOR hVector, DATATYPE val)
{
    if (hVector->size == hVector->capacity &&
            !setCapacity(hVector, hVector->capacity * 2, hVector->size))
        return FALSE;

    hVector->pArray[hVector->size++] = val;

    return TRUE;
}
```

Fonksiyonda önce *size* değerinin *capacity* değerine erişip erişmediğine bakılmıştır. Eğer *size* değeri *capacity* değerine erişmişse *setCapacity* fonksiyonuyla dizi eskisinin iki katı olacak biçimde büyütülmektedir. *setCapacity* fonksiyonu şöyle tanımlanmıştır:

```
static BOOL setCapacity(HVECTOR hVector, size_t capacity, size_t index)
{
    DATATYPE *pTemp;
    size_t newCapacity;

    if (index > hVector->size || index < 0)
        return FALSE;

    newCapacity = capacity > DEFAULT_CAPACITY ? capacity : DEFAULT_CAPACITY;
    pTemp = (DATATYPE *) malloc(sizeof(DATATYPE) * newCapacity);
    if (pTemp == NULL)
        return FALSE;
    if (hVector->size) {
        memcpy(pTemp, hVector->pArray, index * sizeof(DATATYPE));
        memcpy(pTemp + index + 1, hVector->pArray + index,
                (hVector->size - index) * sizeof(DATATYPE));
    }
```

```
    free(hVector->pArray);
    hVector->pArray = pTemp;
    hVector->capacity = newCapacity;

    return TRUE;
}
```

setCapacity fonksiyonu başka fonksiyonlar tarafından da çağrılan ortak bir fonksiyondur. Fonksiyonda önce *malloc* ile *newCapacity* kadar yeni bir dizi tahsis edildiğini görüyorsunuz. (Eğer *newCapacity* değeri *DEFAULT_CAPACITY* değerinden küçükse tahsisat en az *DEFAULT_CAPACITY* kadar yapılmaktadır. Böylece bu fonksiyonu 0 gibi bir değerle çağırdığınızda *DEFAULT_CAPACITY* kadar tahsisat yapılacaktır. Kodu anlamak için şimdilik bu özel duruma fazlaca takılmayın) Eski dizideki değerler yeni diziye *memcpy* fonksiyonuyla kopyalanmıştır. Daha sonra eski dizinin *free* edildiğini görüyorsunuz.

Örnek kodumuzda dinamik dizinin arasına eleman eklemek için *InsertItem* fonksiyonu kullanılmaktadır. Araya eleman eklemek için eklenecek yerden itibaren tüm elemanların bir kaydırılması gerekir. Tabii *insert* işlemi sırasında da kapasite büyütmesine gerek olup olmadığına bakılmıştır:

```
BOOL InsertItem(HVECTOR hVector, size_t index, DATATYPE val)
{
    size_t i;

    if (index > hVector->size)
        return FALSE;

    if (hVector->size == hVector->capacity &&
                !setCapacity(hVector, hVector->capacity * 2, index))
    return FALSE;

    memmove(&hVector->pArray[index + 1], &hVector->pArray[index],
                (hVector->size - index) * sizeof(DATATYPE));

    hVector->pArray[index] = val;
    ++hVector->size;

    return TRUE;
}
```

Gördüğünüz gibi kaydırma işlemi *memmove* fonksiyonuyla yapılmıştır. (Çakışık blokların kopyalanmasında *memcpy* fonksiyonunun kullanılmaması gerektiğini anımsayınız.) Şüphesiz *memmove* yerine kaydırma işlemini aşağıdaki gibi bir döngü ile de yapabilirdik:

```
 for (i = hVector->size; i >= index; --i)
    hVector->pArray[i + 1] = hVector->pArray[i];
```

DeleteItem fonksiyonu aradan eleman silmek için kullanılmaktadır. Dinamik dizilerden eleman silme işleminde genellikle *capacity* değeri düşürülmez. (Bunun nedenini siz düşününüz :-))

```
BOOL DeleteItem(HVECTOR hVector, size_t index)
{
    size_t i;

    if (index >= hVector->size)
        return FALSE;

    memmove(&hVector->pArray[index], &hVector->pArray[index + 1],
                (hVector->size - index - 1) * sizeof(DATATYPE));

    --hVector->size;

    return TRUE;
}
```

Aradan eleman silindiğinde silinen elemanın üzerini kaplayacak biçimde bir sıkıştırma yapılmıştır. Bunun için yine *memmove* fonksiyonunun kullanıldığına dikkat ediniz. Tabi aynı işlemi yine aşağıdaki gibi bir döngüyle de yapabilirdik:

```
for (i = index; i < hVector->size - 1; ++i)
      hVector->pArray[i] = hVector->pArray[i + 1];
```

Gerçekleştirimimizdeki *Clear* fonksiyonu dinamik dizideki tüm elemanları siliyor. Bu silme sırasında yine *capacity* değerinin değiştirilmediğine dikkatinizi çekelim. *Clear* sırasında yapılan tek şey size değerinin sıfırlanmasıdır:

```
void Clear(HVECTOR hVector)
{
    hVector->size = 0;
}
```

Pekiyi *capacity* değerini istediğimiz zaman azaltıp çoğaltabilir miyiz? İşte bunun için *SetCapacity* fonksiyonu bulundurulmuştur. Fonksiyon *capacity* ayarlamasını *static setCapacity* fonksiyonu ile yapmaktadır:

```
BOOL SetCapacity(HVECTOR hVector, size_t capacity)
{
    if (capacity < hVector->size)
        return FALSE;

    if (capacity != hVector->capacity)
        return setCapacity(hVector, capacity, hVector->size);

    return TRUE;
}

static BOOL setCapacity(HVECTOR hVector, size_t capacity, size_t index)
{
    DATATYPE *pTemp;
    size_t newCapacity;

    if (index > hVector->size || index < 0)
        return FALSE;

    newCapacity = capacity > DEFAULT_CAPACITY ? capacity : DEFAULT_CAPACITY;
    pTemp = (DATATYPE *) malloc(sizeof(DATATYPE) * newCapacity);
```

```
    if (pTemp == NULL)
        return FALSE;
    if (hVector->size) {
        memcpy(pTemp, hVector->pArray, index * sizeof(DATATYPE));
        memcpy(pTemp + index + 1, hVector->pArray + index,
                (hVector->size - index) * sizeof(DATATYPE));
    }

    free(hVector->pArray);
    hVector->pArray = pTemp;
    hVector->capacity = newCapacity;

    return TRUE;
}
```

Koddan da gördüğünüz gibi bu fonksiyonla *capacity* değeri *size* değerinin altına çekilememektedir. Yukarıda açıkladığımız fonksiyonların dışında dinamik dizinin belli indeksindeki elemanı bize veren, eleman *set* eden, *size* ve *capacity* değerlerini veren iki fonksiyonlar da vardır:

```
DATATYPE GetItem(HVECTOR hVector, size_t index)
{
    return hVector->pArray[index];
}

void SetItem(HVECTOR hVector, size_t index, DATATYPE val)
{
    hVector->pArray[index] = val;
}

DATATYPE *GetArray(HVECTOR hVector)
{
    return hVector->pArray;
}

 size_t Getsize(HVECTOR hVector)
{
    return hVector->size;
}

size_t GetCapacity(HVECTOR hVector)
{
    return hVector->capacity;
}
```

Şimdi siz, “Zaten *VECTOR* yapısına erişebiliyoruz. Bu işlemleri doğrudan da yapabilirdik. Bu fonksiyonlara ne gerek var?” diye sorabilirsiniz. Bu fonksiyonlar sayesinde *VECTOR* yapısıyla temsil edilen *handle* alanının elemanlarını bilmek zorunda kalmıyoruz. Aynı zamana *VECTOR* yapısının elemanlarına doğrudan erişmemekle orada yapılabilecek değişikliklerden veri yapısını kullanan kodlarımızı korumuş oluyoruz. Yapmaya çalıştığımız şeye nesne yönelimli teknikte kapsülleme (*encapsulation*) ve veri elemanlarının gizlenmesi (*data hiding*) denilmektedir. Yukarıdaki fonksiyonlar birer satırık küçük fonksiyonlar olduğuna göre onları makro olarak yazıp *<vector.h>* dosyası içerisine koyarak fonksiyon çağırmanın ek maaliyetinden kurtulabilirsiniz. Örneğimizdeki küçük fonksiyonların makro karşılıklarını da aşağıda veriyoruz:

```
#define GetItem(hVector, index)       ((hVector)->pArray[(index)])
#define GetSize(hdArray)              ((hVector)->size)
#define GetCapacity(hdArray)          ((hVector)->capacity)
#define SetItem(hVector, index, val)  ((hVector)->pArray[(index)] = (val))
#define Clear(hVector)                ((hVector)->size = 0)
```

6.1.1 Dinamik Dizilerin *C*'de Türden Bağımsız Bir Biçimde Gerçekleştirilmesi

C'de *template* ya da *generic* gibi bir mekanizma olmadığı için yukarıdaki *C* gerçekleştirimi *DATATYPE* ismiyle *typedef* edilmiş olan tek bir türe bağlı durumdadır. Yani yukarıdaki gerçekleştirimi kullandığımızda biz aynı türleri tutan birden fazla dinamik dizi oluşturabiliriz fakat farklı türleri tutan birden fazla dinamik dizi oluşturamayız. *C*'de türden bağımsız veri yapısı iki yöntemle oluşturulabilmektedir:

1) Her tür için aynı fonksiyonların farklı isimlerle yeniden yazılması yöntemi: *C*'de farklı parametrik yapılara ilişkin aynı isimli fonksiyonların (*function overloading*) bulunmadığını anımsayınız. *C++*, *Java* ve *C#* dillerindeki *template* ve *generic* mekanizmalarıyla derleyici tarafından aslında böylesi bir işlem yapılmaktadır.

2) Fonksiyonların her türle çalışabilecek biçimde genel yazılması yöntemi: Bu yöntemde fonksiyonlar *void* göstericilerle nesnelerin adreslerini alacak biçimde tasarlanırlar. Bir karşılaştırma gerekirse bu işlem fonksiyon göstericileriyle kullanıcıya bırakılır. Şüphesiz burada biz ikinci yöntemi uygulayacağız.

Aşağıda dinamik dizilerin türden bağımsız gerçekleştirimine bir örnek verilmiştir:

```
/* Vector.h */

#ifndef VECTOR_H_
#define VECTOR_H_

#define FALSE               0
#define TRUE                1

#define DEFAULT_CAPACITY    2

#include <stddef.h>

/* Type Declarations */

typedef int BOOL;

typedef struct tagVECTOR {
    void *pArray;
    size_t typeSize;
    size_t size;
    size_t capacity;
} VECTOR, *HVECTOR;

/* Function Prototypes */

HVECTOR CreateVectorWithCapacity(size_t typeSize, size_t capacity);
HVECTOR CreateVector(size_t typeSize);
void CloseVector(HVECTOR hVector);
BOOL AddItem(HVECTOR hVector, const void *pVal);
BOOL InsertItem(HVECTOR hVector, size_t index, const void *pVal);
BOOL DeleteItem(HVECTOR hVector, size_t index);
void Clear(HVECTOR hVector);
```

```
void GetItem(HVECTOR hVector, size_t index, void *pVal);
void SetItem(HVECTOR hVector, const void *pVal, size_t index);
BOOL SetCapacity(HVECTOR hVector, size_t capacity);
void *GetArray(HVECTOR hVector);
size_t GetSize(HVECTOR hVector);
size_t GetCapacity(HVECTOR hVector);

#endif


/* Vector.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Vector.h"

static BOOL setCapacity(HVECTOR hVector, size_t capacity, size_t index);

HVECTOR CreateVectorWithCapacity(size_t typeSize, size_t capacity)
{
    HVECTOR hVector;

    if ((hVector = (HVECTOR) malloc(sizeof(VECTOR))) == NULL)
        return NULL;

    if ((hVector->pArray = malloc(typeSize * capacity)) == NULL) {
        free(hVector);
        return NULL;
    }
    hVector->typeSize = typeSize;
    hVector->capacity = capacity;
    hVector->size = 0;

    return hVector;
}

HVECTOR CreateVector(size_t typeSize)
{
    return CreateVectorWithCapacity(typeSize, DEFAULT_CAPACITY);
}

void CloseVector(HVECTOR hVector)
{
    free(hVector->pArray);
    free(hVector);
}

BOOL AddItem(HVECTOR hVector, const void *pVal)
{
    char *pDest;

    if (hVector->size == hVector->capacity &&
                !setCapacity(hVector, hVector->capacity * 2, hVector->size))
        return FALSE;

    pDest = (char *) hVector->pArray + hVector->typeSize * hVector->size;
    memcpy(pDest, pVal, hVector->typeSize);
    ++hVector->size;

    return TRUE;
}
```

```
BOOL InsertItem(HVECTOR hVector, size_t index, const void *pVal)
{
    char *pSource, *pDest;

    if (index > hVector->size)
        return FALSE;

    pSource = (char *)hVector->pArray + index * hVector->typeSize;
    pDest = (char *)hVector->pArray + (index + 1) * hVector->typeSize;

    if (hVector->size == hVector->capacity && !setCapacity(hVector,
                    hVector->capacity * 2, index))
        return FALSE;

    memmove(pDest, pSource, (hVector->size - index) * hVector->typeSize);
      memcpy(pSource, pVal, hVector->typeSize);
    ++hVector->size;

    return TRUE;
}

BOOL DeleteItem(HVECTOR hVector, size_t index)
{
    char *pSource = (char *) hVector->pArray + (index + 1) * hVector->typeSize;
    char *pDest = (char *) hVector->pArray + index * hVector->typeSize;

    if (index >= hVector->size)
        return FALSE;

    memmove(pDest, pSource, (hVector->size - index) * hVector->typeSize);
    --hVector->size;

    return TRUE;
}

void Clear(HVECTOR hVector)
{
    hVector->size = 0;
}

void GetItem(HVECTOR hVector, size_t index, void *pVal)
{
    char *pSource = (char *)hVector->pArray + index * hVector->typeSize;

    memcpy(pVal, pSource, hVector->typeSize);
}

void SetItem(HVECTOR hVector, const void *pVal, size_t index)
{
    char *pDest = (char *)hVector->pArray + index * hVector->typeSize;

    memcpy(pDest, pVal, hVector->typeSize);
}

void *GetArray(HVECTOR hVector)
{
    return hVector->pArray;
}

size_t Getsize(HVECTOR hVector)
{
    return hVector->size;
}
```

```
size_t GetCapacity(HVECTOR hVector)
{
    return hVector->capacity;
}

BOOL SetCapacity(HVECTOR hVector, size_t capacity)
{
    if (capacity < hVector->size)
        return FALSE;

    if (capacity != hVector->capacity)
        return setCapacity(hVector, capacity, hVector->size);

    return TRUE;
}

static BOOL setCapacity(HVECTOR hVector, size_t capacity, size_t index)
{
    void *pTemp;

    if (index > hVector->size)
        return FALSE;

    pTemp = malloc(hVector->typeSize *
                   (capacity >= DEFAULT_CAPACITY ? capacity : DEFAULT_CAPACITY));
    if (pTemp == NULL)
        return FALSE;

    if (hVector->size) {
        memcpy(pTemp, hVector->pArray, index * hVector->typeSize);
        memcpy((char *)pTemp + (index + 1) * hVector->typeSize,
                (char *)hVector->pArray + index * hVector->typeSize,
                (hVector->size - index) * hVector->typeSize);
    }

    free(hVector->pArray);
    hVector->pArray = pTemp;
    hVector->capacity = capacity >= DEFAULT_CAPACITY ? capacity : DEFAULT_CAPACITY;

    return TRUE;
}

void DispVector(HVECTOR hVector)
{
    size_t i;
    int val;

    for (i = 0; i < Getsize(hVector); ++i) {
        GetItem(hVector, i, &val);
        printf("%d ", val);
    }
    printf("\nsize = %lu, Capacity = %lu\n",
                (unsigned long)Getsize(hVector), (unsigned long)GetCapacity(hVector));
    printf("------------------------\n");
}
```

```
// Test kodu

#if 1

int main(void)
{
    HVECTOR hVector;
    int i, val;

    if ((hVector = CreateVector(sizeof(int))) == NULL) {
        fprintf(stderr, "Cannot create vector!..\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < 10; ++i)
        AddItem(hVector, &i);

    val = 1000;
    InsertItem(hVector, 5, &val);
    DispVector(hVector);

    DeleteItem(hVector, 5);
    DispVector(hVector);

    CloseVector(hVector);

    return 0;
}

#endif
```

Kodu anlamlandırabilmeniz için iki ipucu vermek istiyoruz.

1) Veri yapısı içerisinde dinamik dizinin başlangıç adresi, *capacity* ve *size* değerinin yanı sıra diziyi oluşturan elemanların *byte* uzunlukları da tutulmuştur.

```
typedef struct tagVECTOR {
    void *pArray;
    sie_t typeSize;
    size_t size;
    size_t capacity;
} VECTOR, *HVECTOR;
```

2) Dizi elemanlarının hangi türden olduğunu bilmeye gerek yoktur. Bir elemanın hangi uzunlukta olduğunu bilmek yeterlidir.

6.2 Dinamik Dizilerin C++'ta Gerçekleştirilmesi

Dinamik diziler *C++*'ta şablon (*template*) özelliği kullanılarak türden bağımsız bir biçimde gerçekleştirilebilir. Ancak şablonlarla çalışırken şablon argümanının bir sınıf türünden de olabileceği gözden kaçırılmamalıdır. Bu durumda nesnelerin yaratılması ve yok edilmesi sırasında başlangıç (*constructor*) ve bitiş (*destructor*) fonksiyonları çağrılır. Ayrıca *C++* gerçekleştirimlerinde bazı fonksiyonlara *exception* güvenliği vermek gerekebilir. (Örneğin *C++*'ın dinamik büyütülen diziyi temsil eden standart *vector* sınıfında *push_back* ve *insert* gibi fonksiyonların sıkı bir *exception* garantisi (*strong exception guarantee*) vardır.

Aşağıda dinamik bir dizinin *C++*'taki örnek bir gerçekleştirimini veriyoruz:

```
/* Vector.hpp */

#ifndef VECTOR_HPP
#define VECTOR_HPP

#include <cstddef>
#include <new>

template <class T>
class Vector {
public:
    Vector();
    Vector(std::size_t capacity);
    ~Vector();
    bool Add(const T &val);
    bool Insert(std::size_t index, const T &r);
    T &operator[](std::size_t index);
    const T &operator[](std::size_t index) const;
    std::size_t Capacity() { return m_capacity;    }
    bool SetCapacity(std::size_t capacity);
    std::size_t Size() const { return m_size; }
private:
    bool setCapacity(std::size_t capacity, std::size_t index);
    static const int DEFAULT_CAPACITY = 4;

    T *m_pVector;
    std::size_t m_capacity;
    std::size_t m_size;
};

template <class T>
Vector<T>::Vector()
{
    m_pVector = reinterpret_cast<T *>(new char[sizeof(T) * DEFAULT_CAPACITY]);
    m_capacity = DEFAULT_CAPACITY;
    m_size = 0;
}

template <class T>
Vector<T>::Vector(std::size_t capacity)
{
    m_pVector = reinterpret_cast<T *>(new char[sizeof(T) * capacity]);
    m_capacity = capacity;
    m_size = 0;
}

template <class T>
bool Vector<T>::Add(const T &val)
{
    if (m_size == m_capacity && !setCapacity(m_capacity * 2, m_size))
        return false;

    new (m_pVector + m_size) T(val);
    ++m_size;

    return true;
}
```

```
template<class T>
T &Vector<T>::operator[](std::size_t index)
{
    return m_pVector[index];
}

template<class T>
const T &Vector<T>::operator[](std::size_t index) const
{
    return m_pVector[index];
}

template<class T>
Vector<T>::~Vector()
{
    for  (std::size_t i = 0; i < m_size; ++i)
        (m_pVector + i)->~T();
    delete [] reinterpret_cast<char *>(m_pVector);
}

template<class T>
bool Vector<T>::setCapacity(std::size_t capacity, std::size_t index)
{
    if (index > m_size)
        return false;

    T *pVector;
    std::size_t i;

    try {
        i = 0;
        int newCapacity = capacity > DEFAULT_CAPACITY ? capacity : DEFAULT_CAPACITY;
        pVector = reinterpret_cast<T *>(new char[sizeof(T) * newCapacity]);
        for  (i = 0; i < index; ++i)
            new (pVector + i) T(m_pVector[i]);
        for  (i = index; i < m_size; ++i)
            new (pVector + i + 1) T(m_pVector[i]);

        for  (i = 0; i < m_size; ++i)
            (m_pVector + i)->~T();
        delete [] reinterpret_cast<char *>(m_pVector);

        m_pVector = pVector;
        m_capacity = capacity;
    }
    catch (...) {
        for (--i; i >= 0; --i)
            (pVector + i)->~T();
        return false;
    }

    return true;
}

#endif
```

```
// Test Kodu: App.cpp

#if 1

#include <iostream>
#include "Vector.hpp"

using namespace std;

int main()
{
    Vector<int> v;

    for (size_t i = 0; i < 100; ++i)
        v.Add(i);

    for (size_t i = 0; i < v.Size(); ++i)
        cout << v[i] << endl;

    return 0;
}

#endif
```

Kodu anlamlandırırken en zorluk çekeceğiniz kısım *Insert* ve *Add* fonksiyonlarıdır. Bu fonksiyonları sıkı (*strong*) *exception* garantisi verecek biçimde yazdık. Sıkı *exception* garantisine sahip bir fonksiyonda *exception* oluşursa, *exception* yakalandığında sanki *exception*'a yol açan fonksiyon hiç çağrılmamış gibi bir etki söz konusu olur. Sıkı *exception* garantisini sağlamaya çalışırken *exception*'ın herhangi bir noktada oluşabileceği (örneğin başlangıç fonksiyonlarında, atama operatör fonksiyonlarında vs.) dikkate alınmalıdır. Kodda da gördüğünüz gibi belli bir noktada *exception* oluşursa bir geri alım (*rollback*) yapılmaktadır:

```
try {
    i = 0;
      int   newCapacity   =   capacity   >   DEFAULT_CAPACITY   ?   capacity   :
DEFAULT_CAPACITY;
    pVector = reinterpret_cast<T *>(new char[sizeof(T) * newCapacity]);
    for  (i = 0; i < index; ++i)
        new (pVector + i) T(m_pVector[i]);
    for  (i = index; i < m_size; ++i)
        new (pVector + i + 1) T(m_pVector[i]);

    for  (i = 0; i < m_size; ++i)
        (m_pVector + i)->~T();
    delete [] reinterpret_cast<char *>(m_pVector);

    m_pVector = pVector;
    m_capacity = capacity;
}
catch (...) {
    for (--i; i >= 0; --i)
        (pVector + i)->~T();

    return false;
}
```

6.3 Dinamik Dizilerin C#'ta Gerçekleştirilmesi

Bilindiği gibi *C#* ve *Java* dillerinde yerel düzeyde sınıf nesneleri yaratılamazlar. Bu dillerde tüm nesneler *new* operatörü ile *heap*'te yaratılmak zorundadır. Çöp toplayıcı (*garbage collection*) mekanizma sayesinde bu dillerde *heap*'te tahsis edilen nesneler artık onları hiçbir referans göstermediğinde otomatik olarak *heap*'ten yok edilmektedir. Aşağıda *C#*'ta *generic* biçimde örnek bir dinamik dizi gerçekleştirimini görüyorsunuz:

```
using System;
using System.Text;

namespace CSD
{
    class MyArrayList<T>
    {
        private T[] m_elems;
        private int m_count;

        public MyArrayList()
            : this(2)
        { }

        public MyArrayList(int capacity)
        {
            if (capacity <= 0)
                throw new ArgumentOutOfRangeException
                                ("Capacity must be positive value");

            m_elems = new T[capacity];
        }

        public int Capacity
        {
            get
            {
                return m_elems.Length;
            }
            set
            {
                if (value < m_count)
                    throw new ArgumentOutOfRangeException
                                ("capacity must be greater or equalto Count");

                if (value > m_elems.Length)
                    this.resizeCapacity(value, m_count);
            }
        }

        public int Count
        {
            get
            {
                return m_count;
            }
        }
```

```
public int Add(T val)
{
    if (m_count == m_elems.Length)
        this.resizeCapacity(2 * m_elems.Length, m_count);

    m_elems[m_count++] = val;

    return m_count - 1;
}

public void Insert(int index, T val)
{
    if (index < 0 || index > m_count)
        throw new ArgumentOutOfRangeException("index out of range");

    if (m_count == m_elems.Length)
        this.resizeCapacity(2 * m_elems.Length, index);
    else
        for (int i = m_count - 1; i >= index; --i)
            m_elems[i + 1] = m_elems[i];
    m_elems[index] = val;
    ++m_count;

}

public void Delete(int index)
{
    if (index < 0 || index >= m_count)
        throw new ArgumentOutOfRangeException("index out of range");

    for (int i = index; i < m_count - 1; ++i)
        m_elems[i] = m_elems[i + 1];
    --m_count;
}


public T this[int index]
{
    get
    {
        if (index < 0 || index >= m_count)
            throw new ArgumentOutOfRangeException("index out of range");

        return m_elems[index];
    }

    set
    {
        if (index < 0 || index >= m_count)
            throw new ArgumentOutOfRangeException("index out of range");

        m_elems[index] = value;
    }
}
```

```
        private void resizeCapacity(int capacity, int index)
        {
            T[] elems = new T[capacity];
            Array.Copy(m_elems, 0, elems, 0, index);
            Array.Copy(m_elems, index, elems, index + 1, m_count - index);

            m_elems = elems;
        }

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();

            for (int i = 0; i < m_count; ++i)
            {
                if (i != 0)
                    sb.Append(", ");
                sb.Append(m_elems[i].ToString());
            }
            sb.AppendFormat(" (Count = {0}, Capacity={1})", m_count, m_elems.Length);

            return sb.ToString();
        }
    }

    class App
    {
        public static void Main()
        {
            MyArrayList<int> al = new MyArrayList<int>();

            for (int i = 0; i < 10; ++i)
                al.Add(i);
            Console.WriteLine(al);

            al.Insert(5, 100);
            Console.WriteLine(al);

            al.Delete(6);
            Console.WriteLine(al);
        }
    }
}
```

6.4 Dinamik Dizilerin Java'da Gerçekleştirilmesi

C# ile *Java* Programlama dilleri (fakat *.NET* ve *Java* platformları değil) birbirlerine çok benzemektedir. *C#* programlama dili olarak *Java*'dan alınmış (açıkçası kopya çekilmiş), çok daha iyi hale getirilmiş ve biraz daha *C++*'a yaklaştırılmıştır. Bu nedenle dinamik dizilerin *C#* gerçekleştirimi ile *Java* gerçekleştirimleri de birbirlerine çok benzerdir. Aşağıda *generic* biçimde oluşturulmuş örnek bir gerçekleştirim veriyoruz:

```
package CSD;

public class MyArrayList<E>
{

    private E[] m_elems;
    private int m_count;

    public MyArrayList()
    {
        this(2);
    }

    public MyArrayList(int capacity)
    {
        if (capacity <= 0)
            throw new IllegalArgumentException("Capacity must be positive value");

        m_elems = (E[])new Object[capacity];
    }

    public int capacity()
    {
        return m_elems.length;
    }

    public void setCapacity(int capacity)
    {

        if (capacity < m_count)
            throw new IllegalArgumentException
                        ("capacity must be greater or equalto Count");

        if (capacity > m_elems.length)
            resizeCapacity(capacity, m_count);
     }

    public int count()
    {
        return m_count;
    }

    public int Add(E val)
    {
        if (m_count == m_elems.length)
            resizeCapacity(2 * m_elems.length, m_count);

         m_elems[m_count++] = val;

         return m_count - 1;
    }
```

```
public void Insert(int index, E val)
{
   if (index < 0 || index > m_count)
        throw new IndexOutOfBoundsException("index out of range");

   if (m_count == m_elems.length)
        this.resizeCapacity(2 * m_elems.length, index);
   else
        for (int i = m_count - 1; i >= index; --i)
             m_elems[i + 1] = m_elems[i];
    m_elems[index] = val;
    ++m_count;
}

public void Delete(int index)
{
    if (index < 0 || index >= m_count)
        throw new IndexOutOfBoundsException("index out of range");

    for (int i = index; i < m_count - 1; ++i)
        m_elems[i] = m_elems[i + 1];

    --m_count;
}

public E get(int index)
{
    if (index < 0 || index >= m_count)
        throw new IndexOutOfBoundsException("index out of range");

    return m_elems[index];
}

public void set(int index, E val)
{
    if (index < 0 || index >= m_count)
        throw new IndexOutOfBoundsException("index out of range");

    m_elems[index] = val;
}

public String toString()
{
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i < m_count; ++i)
    {
        if (i != 0)
            sb.append(", ");
        sb.append(m_elems[i].toString());
    }
    sb.append(String.format(" (Count=%d, Capacity=%d)", m_count, m_elems.length));

    return sb.toString();
}
```

```
    private void resizeCapacity(int capacity, int index)
    {
        E[] objs = (E[])new Object[capacity];
        System.arraycopy(m_elems, 0, objs, 0, index);
        System.arraycopy(m_elems, index, objs, index + 1, m_count - index);

        m_elems = objs;
    }
}
```

---

[1]: Dinamik dizi kavramı ile dinamik olarak tahsis edilen dizi kavramları birbirlerine karışabilmektedir. Dinamik dizi denildiğinde programın çalışma zamanı sırasında büyütülüp küçültülen diziler anlaşılır. Dinamik olarak tahsis edilen dizi ise *malloc* gibi dinamik bellek fonksiyonlarıyla ya da *new* gibi operatörlerle *heap* üzerinde tahsis edilen dizilerdir.

[2]: *C*, *C++*, *Java* ve *C#* gibi dillerde && (*AND*) ve || (*OR*) oeratörlerinin kısa devre özellikleri olduğunu anımsayınız. Dolayısıyla burada *size* değeri *capacity* değerine eşit değilse *setCapacity* fonksiyonu hiç çağrılmayacaktır.

[3]: *C#*, programlama dili olarak *Java*'dan alınmış (açıkçası kopya çekilmiş), fakat yapılan eklemelerle çok daha iyi hale getirilmiş ve biraz daha *C++*'a yaklaştırılmıştır.