DİNAMİK BELLEK YÖNETİMİ

Göstericilerin (pointer) bellek adreslerini tutan değişkenler olduğu daha önce açıklanmıştı. C yapısal bir dil olduğundan, programlama için bazı sabit kurallara sahiptir. Örneğin bir dizinin boyutunu tanımlandıktan sonra değiştirmezsiniz. Dinamik bellek yönetimi ile çalıştırma anında istenilen boyuta göre dizi oluşturulabilir. Çalıştırma anında istenilen boyutta oluşturulabilen bu dizileri de göstericiler (pointer) sayesinde işleriz.

Dinamik olarak tahsis edilen bellekteki veriler öbek (heap) bellekte tutulur. Yerel değişkenlerin son konulan verinin ilk geri alındığı yığın (stack) bellekte tutulduğu, statik ve global değişkenlerin de veri (data) bellekte tutulduğu daha önce anlatılmıştı.

Çalıştırma anındaki bellek tahsisine dinamik bellek tahsisi (dynamic memory allocation) denir. Bellek tahsisine ilişkin fonksiyonlar `stdlib.h` başlık dosyasında bulunmaktadır. Bu başlıktaki `malloc()`, `calloc()` ve `realloc()` fonksiyonları kullanılarak yapılan dinamik bellek tahsisi yapılır `free()` fonksiyonu ile tahsis edilen belek bölgesi serbest bırakılır.

malloc() fonksiyonu

```
ptr = (cast-type*) malloc(byte-size);
```

Bu fonksiyon, belirtilen boyutta tek bir büyük bellek bloğunu bayt cinsinden dinamik olarak tahsis etmek için kullanılır. Herhangi bir biçimdeki bir göstericiye dönüştürülebilen `void` türünde bir gösterici döndürür.

```
int* ptr1;
ptr1 = (int*) malloc(50 * sizeof(int)); /* 50 adet int içerecek bellek bloğu tahsis edilerek
göstericisi ptr1 değişkenine atandı. Bellek ayrılamasaydı NULL gösterici geri döndürecekti.*/
```

calloc() fonksiyonu

```
ptr = (cast-type*) calloc(n, element-size);
```

Bu fonksiyon, belirtilen veri tipindeki belirtilen sayıda bellek bloğunu birbirine bitişik olarak (contiguous) dinamik olarak tahsis etmek için kullanılır.

```
float* ptr2;
ptr2 = (float*) calloc(20, sizeof(float)); /* 20 adet float içerecek bellek bloğu tahsis
edilerek göstericisi ptr2 değişkenine atandı. Bellek ayrılamasaydı NULL gösterici geri
döndürecekti.*/
```

realloc() fonksiyonu

```
ptr = realloc(ptr, newSize);
```

Bu fonksiyon, daha önce tahsis edilmiş bir belleğin miktarını değiştirmek için kullanılır.

```
ptr1 = (int*) realloc(ptr1, 10 * sizeof(int));
ptr2 = (float*) realloc(ptr2, 40 * sizeof(float));
/* Bellek yeniden ayrılamasaydı NULL gösterici geri döndürecekti.*/
```

free() fonksiyonu

```
free(ptr);
```

Bu fonksiyon, tahsis edilen belleği serbest bırakmak için kullanılır. `malloc()`, `calloc()` ve `realloc()` fonksiyonları kullanılarak tahsis edilen bellek serbest bırakılır.

```
free(ptr1);
ptr1 = NULL;
```

Örnek Program

Aşağıda öğrenci sayısını çalıştırma anında alıp, bu öğrencilere ilişkin notları rastgele belirleyen bir program verilmiştir.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    int* notlar=NULL;
    int n, i;
    srand(time(NULL)); //Notlar rastgele atanacak.

    printf("Öğrenci Sayısını Giriniz:");
    scanf("%d",&n);
    printf("%d Elemanlı Notlar Dizisi Oluşturulacak\n", n);

    notlar = (int*) malloc(n * sizeof(int));
    if (notlar == NULL) {
        printf("Hafıza tahsis edilemedi!\n");
        exit(-1); //İşletim sistemine hata ile dönülüyor.
    } else {
        printf("Hafıza başarılı bir şekilde tahsis edildi.\n");
        for (i = 0; i < n; ++i)
            notlar[i] = rand()%101;
        printf("Öğrenci Notları:\n");
        for (i = 0; i < n; ++i)
            printf("%d, ", notlar[i]);
    }
    free(notlar);
    return 0;
}
```

Aşağıda karakter sayısını çalıştırma anında alıp, yığın bellekte bir dizgi (string) oluşturan bir program verilmiştir.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char* string;
    int n;
    printf("Girilen Metin En Fazla Kaç Karakter:");
    scanf("%d",&n);

    string = (char*) malloc(n);
    if (string == NULL) {
        printf("Hafıza tahsis edilemedi!\n");
        exit(-1); //İşletim sistemine hata ile dönülüyor.
    } else {
        printf("Hafıza başarılı bir şekilde tahsis edildi.\n");
        puts("Metni Giriniz:");
        scanf("%s",string);
        printf("Girilen Metin:\n%s",string);
    }
    return 0;
}
```

Dinamik Veri Yapıları

Çalıştırma anında bellek tahsis edilerek kullanılan veri yapılarına dinamik veri yapıları (dynamic data structure) denir. Bunlar; Bağlı Listeler (linked list), İstifler (stack), Kuyruklar (queue), İkili Ağaçlar

(binary tree) ve Sözlüktür (dictionary). Buradaki veri yapılarının çoğu bir önceki bölümde anlatılan öz referanslı yapılar (self-referential structure) kullanılarak hayata geçirilir.

Bağlantılı Liste

Bağlantılı liste (linked list), düğüm olarak bilinen her bir öğenin göstericiler kullanılarak bir sonraki düğüme bağlandığı doğrusal bir veri yapısıdır. Diziden farklı olarak, bağlantılı listenin öğelerinin her biri (düğüm olarak adlandırılmaktadır), öbek (heap) bellekte ayrı ayrı oluşturulduğundan, rastgele bellek konumlarında saklanır.

Düğümlerde saklanan veri tek bir değişken olacağı gibi, birden fazla değişken de olabilir. Bağlantılı liste;

- Aşağıda verilen örnekte görüldüğü gibi tek yönlü olabileceği gibi,
- Hem sonraki düğümü hem de önceki düğümü gösteren iki yönlü liste,
- Veya son ve ilk düğümü olmayan halka şekilde liste tanımlanabilir.

```
#include <stdio.h>
#include <stdlib.h>

struct dugumYapi {
  int veri;
  //char veri2; ...
  struct dugumYapi* sonraki;
};
typedef struct dugumYapi Dugum;
void ilkDugumeEkle(Dugum**,int);
void sonaDugumEkle(Dugum*,int);
void listeyiYaz(Dugum*);
void ilkDugumuSil(Dugum**);
void sonDugumuSil(Dugum*);
int main() {
    Dugum* ilk=NULL;  //Başlangıçta hiç düğüm yok.
    ilkDugumeEkle(&ilk,10);
    sonaDugumEkle(ilk,20);
    sonaDugumEkle(ilk,30);
    ilkDugumeEkle(&ilk,-10);
    listeyiYaz(ilk);
    return 0;
}
void ilkDugumeEkle(Dugum** ilkDugumGostericisi, int pVeri) {
    Dugum * yeni;
    yeni = (Dugum *) malloc(sizeof(Dugum));//yeni düğüm için bellek tahsisi
    yeni->veri = pVeri;//tahsis edilen düğüme veri konuluyor
    yeni->sonraki = *ilkDugumGostericisi; // sonraki düğüm ilkDugumGostericisi olsun.
    *ilkDugumGostericisi = yeni;//İlk düğüm göstericisi yeni düğüm olsun
}
void sonaDugumEkle(Dugum* pIlk, int pVeri) {
    Dugum* hangi = pIlk; //hangi ilk düğümü göstersin.
    if (hangi==NULL) {
        printf("Liste olmadığından eklenemedi!\n");
        return;
    }
    while (hangi->sonraki != NULL) {
        hangi = hangi->sonraki; //listenin sonuna kadar ilerle
    }
    hangi->sonraki = (Dugum*) malloc(sizeof(Dugum)); //sonuna yeni düğüm ekle
    hangi->sonraki->veri = pVeri; //yeni düğüme verisi konuluyor
    hangi->sonraki->sonraki = NULL; //son düğümün sonraki düğüm NULL olsun.
}
void listeyiYaz(Dugum* pIlk) {
    Dugum* hangi = pIlk;  int sayac=0;
    while (hangi != NULL) {
```

```
        printf("Dugum (%d): veri=%d\n",
               ++sayac, hangi->veri);
        hangi = hangi->sonraki;
    }
}
void ilkDugumuSil(Dugum** ilkDugumGostericisi) {
    Dugum* sonrakiDugum = NULL;
    if (*ilkDugumGostericisi == NULL) return;
    sonrakiDugum = (*ilkDugumGostericisi)->sonraki;
    free(*ilkDugumGostericisi);
    *ilkDugumGostericisi = sonrakiDugum;
}
void sonDugumuSil(Dugum* pIlk) {
    if (pIlk->sonraki == NULL) {
        free(pIlk);
        return;
    }
    Dugum* hangi = pIlk;
    while (hangi->sonraki->sonraki != NULL)
        hangi = hangi->sonraki;
    free(hangi->sonraki);
    hangi->sonraki = NULL;
}
```

Yığın Veri Yapısı

Yığınlar (stack), bağlantılı listenin (linked list), kısıtlanmış halidir. Haliyle doğrusal bir veri yapısıdır. Yığın veri yapısında;

- Bağlantılı listeye yeni düğüm, ancak listenin başına eklenir. Bu işlem itme (push) olarak adlandırılır.
- Bağlantılı listede silme işlemi yalnızca listenin başındaki düğüme uygulanır. Bu işleme çekme (pop) adı verilir.
- Böylece listeye son giren veri ilk alınır (last in first out-LIFO).

```
#include <stdio.h>
#include <stdlib.h>

struct dugumYapi {
  int veri;
  struct dugumYapi* sonraki;
};
typedef struct dugumYapi Dugum;
void it(Dugum**,int);
int cek(Dugum**);
int main() {
  Dugum* istif=NULL;
  it(&istif,10);
  it(&istif,20);
  it(&istif,30);
  int veri=cek(&istif);
  printf("Çekilen Veri:%d\n",veri);
  veri=cek(&istif);
  printf("Çekilen Veri:%d\n",veri);
  return 0;
}
void it(Dugum** ilkDugumGostericisi, int pVeri) {
    Dugum* yeni;
    yeni = (Dugum *) malloc(sizeof(Dugum));
    yeni->veri = pVeri;
    yeni->sonraki = *ilkDugumGostericisi;
    *ilkDugumGostericisi = yeni;
}
int cek(Dugum** ilkDugumGostericisi) {
```

```
    Dugum * sonrakiDugum = NULL;
    if (*ilkDugumGostericisi == NULL) {
        printf("İstifde düğüm yok!");
        exit(-1);
    }
    int veri=(*ilkDugumGostericisi)->veri;
    sonrakiDugum = (*ilkDugumGostericisi)->sonraki;
    free(*ilkDugumGostericisi);
    *ilkDugumGostericisi = sonrakiDugum;
    return veri;
}
```

Kuyruk Veri Yapısı

Kuyruk (queue) de, bağlantılı listenin (linked list), kısıtlanmış halidir. Haliyle bu veri yapısı da doğrusal bir veri yapısıdır. Kuyruk veri yapısında;

- Bağlantılı listeye yeni düğüm, ancak listenin başına eklenir. Bu işlem, kuyruğa sokma (enqueue) olarak adlandırılır.
- Bağlantılı listede silme işlemi yalnızca listenin sonundaki düğüme uygulanır. Bu işleme kuyruktan çıkarma (dequeue) adı verilir.
- Böylece listeye ilk giren veri ilk alınır (first in first out-FIFO).

```
#include <stdio.h>
#include <stdlib.h>

struct dugumYapi {
  int veri;
  struct dugumYapi* sonraki;
};
typedef struct dugumYapi Dugum;
void enqueue(Dugum**,int);
int dequeue(Dugum*);
int main() {
  Dugum* kuyruk=NULL;
  enqueue(&kuyruk,10);
  enqueue(&kuyruk,20);
  enqueue(&kuyruk,30);
  int veri=dequeue(kuyruk);
  printf("Alınan Veri:%d\n", veri);
  veri=dequeue(kuyruk);
  printf("Alınan Veri: %d\n",veri);
  return 0;
}
void enqueue(Dugum** ilkDugumGostericisi, int pVeri) {
    Dugum* yeni;
    yeni = (Dugum*) malloc(sizeof(Dugum));
    yeni->veri = pVeri;
    yeni->sonraki = *ilkDugumGostericisi;
    *ilkDugumGostericisi = yeni;
}
int dequeue(Dugum* pIlk) {
    int veri;
    if (pIlk->sonraki == NULL) {
        veri=pIlk->veri;
        free(pIlk);
        return veri;
    }
    Dugum* hangi = pIlk;
    while (hangi->sonraki->sonraki != NULL)
        hangi = hangi->sonraki;
```

```
    veri=hangi->sonraki->veri;
    free(hangi->sonraki);
    hangi->sonraki = NULL;
    return veri;
}
```

İkili Ağaç Veri Yapısı

Bu veri yapısı, bir kök ve dallarından oluşan veri yapısıdır. İkili (binary) denmesinin sebebi bir düğümden, genellikle sağ ve sol olarak adlandırılan, yalnıza iki dal çıkabildiğindendir. Haliyle bu veri yapısı da doğrusal olmayan (nonlinear) hiyerarşik bir veri yapısıdır.

```
struct dugumYapi {
  int veri;
  //char veri2;
  //float veri3;
  //...
  struct dugumYapi* sag;
  struct dugumYapi* sol;
};
typedef struct dugumYapi Dugum;
```

Ağacın ilk düğümüne kök düğüm (root node) adı verilir. Kök düğümden dallanan iki düğüm çocuk düğüm (child node) olarak adlandırılır. Bu şekilde her dala eklenen düğüm ile ters bir ağaç oluşur. En uçtaki çocuk düğüme yaprak düğüm (leaf node) adı verilir. İkili bir ağaçta gerçekleştirilebilecek temel işlemler şunlardır:

- Ekleme (insertion)
- Silme (deletion)
- Arama (search) ve
- Gezinme (traversing). Üç şekilde yapılır;
  - Ön Sıralı (pre-order traversal)
  - Son Sıralı (post-order traversal)
  - Sıralı (in-order traversal)

```
#include <stdio.h>
#include <stdlib.h>

struct agacDugum {
    int veri;
    struct agacDugum* sol;
    struct agacDugum* sag;
};
typedef struct agacDugum Dugum;
Dugum* yeniDugum(int pVeri) {
    Dugum* yeni = (Dugum*)malloc(sizeof(Dugum));
    if (yeni != NULL) {
        yeni->veri = pVeri; // Tahsis yapılan düğüme veri aktarılıyor
        yeni->sol = NULL;
        yeni->sag = NULL;
    }
    return yeni;
}
Dugum* dugumEkle(Dugum* kok, int pVeri) {
    if (kok == NULL)
        return yeniDugum(pVeri); // Ağaç yoksa yeni düğüm ekle
    if (pVeri < kok->veri) // Eklenecek veriyi karşılaştır
        kok->sol = dugumEkle(kok->sol, pVeri);
    else if (pVeri > kok->veri)
        kok->sag = dugumEkle(kok->sag, pVeri);
    return kok; // Değişen kök düğümü döndür.
}
```

```
void siraliGezinme(Dugum* kok) {
    //   Sıralı Gezinti: sol altagac, kok, sag altagac
    if (kok != NULL) {
        siraliGezinme(kok->sol);
        printf("%d ", kok->veri);
        siraliGezinme(kok->sag);
    }
}
void onSiraliGezinme(Dugum* kok) {
    //   Ön Sıralı Gezinti: kok, sol altagac, sag altagac
    if (kok != NULL) {
        printf("%d ", kok->veri);
        onSiraliGezinme(kok->sol);
        onSiraliGezinme(kok->sag);
    }
}
void sonSiraliGezinme(Dugum* kok) {
    //   Ön Sıralı Gezinti: kok, sol altagac, sag altagac
    if (kok != NULL) {
        sonSiraliGezinme(kok->sol);
        sonSiraliGezinme(kok->sag);
        printf("%d ", kok->veri);
    }
}
void agaciBelllektenKaldir(Dugum* kok) {
    if (kok != NULL) {
        agaciBelllektenKaldir(kok->sol);
        agaciBelllektenKaldir(kok->sag);
        free(kok);
    }
}
int main() {
    Dugum* kok = NULL;
    int dugumVerisi;
    char secim;
    kok=dugumEkle(kok, 20);
    kok=dugumEkle(kok, 30);
    kok=dugumEkle(kok, 40);
    kok=dugumEkle(kok, 50);
    kok=dugumEkle(kok, 60);
    kok=dugumEkle(kok, 70);
    kok=dugumEkle(kok, 80);
    printf("\nSıralı Gezinti ile Ağaç: ");
    siraliGezinme(kok);
    printf("\n");
    printf("\nOn Sıralı Gezinti ile Ağaç: ");
    onSiraliGezinme(kok);
    printf("\n");
    printf("\nSon Sıralı Gezinti ile Ağaç: ");
    sonSiraliGezinme(kok);
    printf("\n");
    agaciBelllektenKaldir(kok);
    kok=NULL;
    return 0;
}
```

Sözlük Veri Yapısı

Bu veri yapısı, tekil olarak anahtarları (key) barındıran ve her anahtara bir değer (value) verilebilen bir yapıdır. Tanımından hareketle bu veri yapısına sözlük (dictionary) veya eşleme (map) adı verilir.

```
#include <stdio.h>
```

```
#include <string.h>

#define MAX_ESLEME 100 //Sözlükte en fazla 100 eşleşme olacak.

int eslesmeSayisi = 0; // Aktif sözlük Uzunluğu
char keys[MAX_ESLEME][100];
int values[MAX_ESLEME];
int indisiBul(char key[])
{
   int i;
   for (i = 0; i < eslesmeSayisi; i++) {
      if (strcmp(keys[i], key) == 0)
         return i;
   }
   return -1;
}
void sozlugeEkle(char key[], int value)
{
   int index = indisiBul(key);
   if (index == -1) {
      strcpy(keys[eslesmeSayisi], key);
      values[eslesmeSayisi] = value;
      eslesmeSayisi++;
   } else
      values[index] = value;
}
int sozlukteBul(char key[])
{
   int index = indisiBul(key);
   if (index == -1)
      return -1;
   else
      return values[index];
}
void sozluguYazdir()
{
   for (int i = 0; i<eslesmeSayisi; i++)
      printf("%s: %d\n", keys[i], values[i]);
}
int main()
{
   sozlugeEkle("Elma", 100);
   sozlugeEkle("Armut", 200);
   sozlugeEkle("Muz", 300);

   printf("Sozluk İçeriği: \n");
   sozluguYazdir();

   printf("\nElmanın Sözlükteki Değeri: %d\n",
       sozlukteBul("Elma"));
   printf("Muzun İndisi: %d\n",
       indisiBul("Muz"));
   return 0;
}
```