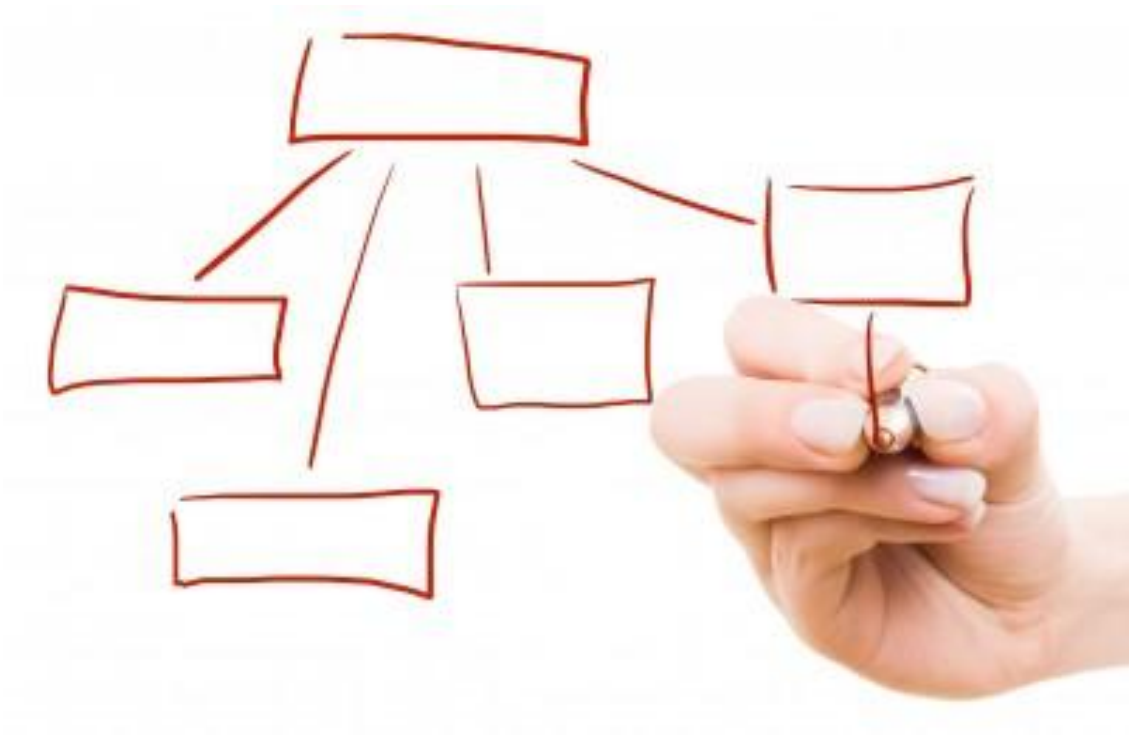
SOMEBODY ELSE'S

(DATA STRUCTURES and ALGORITHMS)

Veri Yapıları ve Algoritmaları



Yazan: Burak Kıymaz

Derleyen: Serhan Aksoy

Veri Yapıları

Abstrak veri yapıları: (Abstract Data Types and Data Structures)

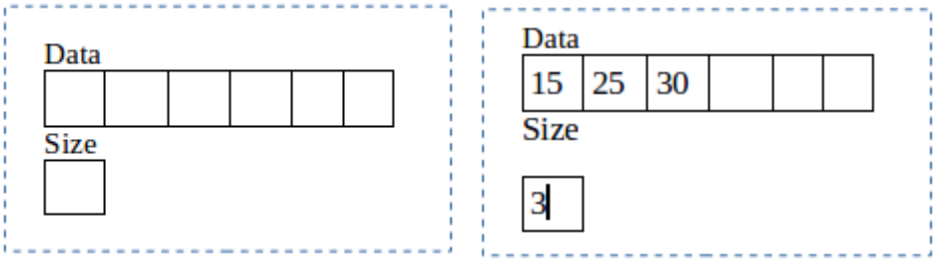
ADT , içerisindeki verilerin ve bu verilerle yapılacak işlemlerin soyut bir şekilde tanımlanmış halidir.

Data Structures : ADT nin gerçeklenmiş halidir.

Data Structures şekilde yapılabilir.
→ Struct ile yapılabilir.
→ Paralel diziler ile yapılabilir
→ Dinamik pointer lar ile yapılabilir.

ÖRNEK (Genel Liste):

Genel Liste ADT:
→ sabit bir integer dizi olsun
→ ekleme silme yazdırma arama fonksiyonları olsun.



Genel Yapı

```
#include <stdio.h>
#include <stdlib.h>
#define listeBoyutu 10

struct genelListe{
        int Data[listeBoyutu];
        int size;// dolu eleman sayısı
    };

    void baslat();
    int doluMu();
    int bosmu();
    void ekle();
    void sil();
    void Bul(int, int*);
    void goster();
    struct genelListe list;

int main(int argc, char** argv) {
    baslat();
    ekle(15);
```

```
    ekle(25);
    ekle(35);
    sil(35);
    goster();


    return (EXIT_SUCCESS);
}

void baslat(){
    list.size = 0;
}
int doluMu() {
    if(list.size == listeBoyutu){
        return 1;
    }
    else{
        return -1;
    }
}

int bosmu(){
    if (list.size == 0)
        return 1;
    else
        return -1;
}
void ekle(int deger){
    if (doluMu() == 1){
        printf("Uzgunuz liste dolu ekleyemiyoruz.. ");
    }
    else{
        list.Data[list.size] = deger;
        list.size++;
        printf("%d listeye eklendi.",deger);
    }
}

void goster(){
    if (bosmu() == 1){
        printf("Liste Bos");
    }
    else{
        int i;
        printf("Listedeki elemanlar: ");
        for (i=0;i<list.size;i++){
            printf("%d, ",list.Data[i]);
        }
    }
}
void Bul(int eleman, int* pos){
    *pos = -1;
    if (bosmu()==-1){
```

```
        int i=0;
        for (i=0;i<list.size;i++)
            if (eleman==list.Data[i]){
                *pos = i;
                break;
            }


    }
}

void sil(int deger){
    int k=-1;
    Bul(deger,&k);
    if (k==-1){
        printf("Eleman yok.");
    }
    else{
        int i=0;
        for (i=k;i<list.size;i++){
            list.Data[i] = list.Data[i+1];
        }
        list.size--;
    }
}
```

Genel liste dezavantajlıdır çünkü bir eleman çıkarıldığı zaman tüm diziyi yerinden oynatmak gerekir. Bu yüzden ram kullanımı çok kötüdür. Bunu düzeltmek için LinkedList kullanılır.

Linked List (Bağlı Liste)

her düğüm (eleman) hem veriyi hem de bir sonrakinin adresini tutacaktır.

Data	Next indis

→ Single LinkedList

→ Double LinkedList

Pre	Data	Next

→ Multiple LinkedList

Pre	Data	...	Data	Next

LinkedList 3 yöntemle yapılabilir.

→ Çoklu dizilerle yapılır.

→ Struct dizisi

→ dinamik pointerlar

ÖRNEK:

int [10][2];

Bellek

İlk →

Indis	Data	Next
0		
1	10	5
2		
3	20	1
4	30	-1
5	3	4
6		
7		
8		
9		

20	1
10	~~5~~ 4
~~3~~	~~4~~
30	-1

Silindi

20	1
10	4
30	-1

Veri Yapıları

0	1	2	3	4	5	6	7	8
10	30	40	50	15	23	45	60	80
1	2	3	4	5	6	7	8	-1

3 5

Boş listesi

1	2	3	4	5	6	7	8
						2	4
						8	-1

```
Head = 0
Boş = -1 // en baştaki değer, tüm liste dolu demek.


void Baslat(){
    int list[10][1];
    int i;
    for (i=0;i<9;i++){
        list[i][1] = i+1;
    }
    list[9][1]=-1;
    int head = -1;
    int bas = 0;
}
// dizi oluşturuldu şu anda dizi boş olduğu için head -1 boş =0 boş un 0
olmasının sebebi hepsi boş olduğu için birbirine bağlı.


    eleman = bos;
    list[bos][0] = 15;
    bos = list[bos][1];
    list[eleman][1] = -1;
    head = eleman;
```

diziye eklenen ilk elemanın gösterdiği eleman 0 yazılır yani hiçbir yeri göstermez. Diziye sonradan eklenen elemanlar head olarak eklenir.

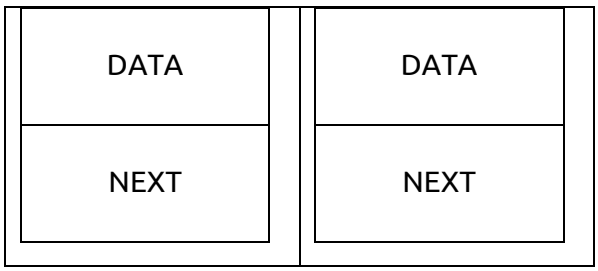
```
eleman = bos;

    list[eleman][0] = 25;

    bos = list[bos][1];

    list[eleman][1];

    head = eleman;
```

```
void yazdir(){

    if (head != -1){

        int temp = head;

        while (temp!= -1) { // temp -1 olana kadar yazdırma işlemini yapar
temp=-1 ise liste bitmiş demektir.

            printf("%d\n",list[temp][0]); // temp in göstersiği indisin
elemanı yazdıırldı

            temp = list[temp][1]; // temp yazdırılan temp in gösterdiği
adresi gösteriyor artık.

        }

    }

}
```

Listeden bir elemanı sildiğimizde bu elemanı boşların bulunduğu listeye head olarak ekleriz.

Struct dizisi ile linked list

```
struct dugum{
    int data;
    int next;
};
struct dugum liste[10];
int main(int argc, char** argv) {
    return (EXIT_SUCCESS);
}
```




```
void Baslat(){
    int i;
    for(i=0;i<9;i++){
        liste[i].next = i+1;
    }
    liste[i].next = -1;
    int head = -1;
    int bos = 0;
}// struct dizisinin başlatma komutu.
```

ÖRNEK UYGULAMA

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

void Baslat();
int doluMU();
int bosMu();
void yazdir();
void ekle(int);
void sil(int);
void dugumVer(int*);
void Bul(int,int*);

struct dugum{
    int data; // burada veri tutulacak
    int next; // burada bir sonraki düğümün indisi tutulacak
};// veri yapısı tanımlardık

struct dugum liste[10];
int head = -1;
int bos = 0;

int main(int argc, char** argv) {
    Baslat();
    int temp;

    dugumVer(&temp);
    liste[temp].data = 15;
    head = temp;
    liste[temp].next = -1;
```

```
    dugumVer(&temp);
    liste[temp].data = 25;
    liste[temp].next = head;
    head = temp; // burada manuel olarak 2 tane eleman eklendi.

    yazdir();

    return (EXIT_SUCCESS);
}

void Baslat(){
    int i;
    for(i=0;i<9;i++){
        liste[i].next = i+1;
    }
    liste[10].next = -1;

}
int doluMU(){
    if (bos == -1)
        return 1;
    else
        return -1;

}
int bosMu(){
    if (head == -1)
        return 1;
    else
        return -1;
```

```
}
void yazdir(){
    if (bosMu() ==1){
        printf("Liste Bos");
    }
    else{
        int temp = head;
        while (temp!=-1){
            printf("%d\n", liste[temp].data);
            temp = liste[temp].next;
        }
    }
}
void dugumVer(int*k){
    if (doluMU() ==1)
        printf("Bos yer yok");
    else{
        *k = bos;
        bos = liste[bos].next;
    }
}
```

Ödev:

İki boyutlu statik matris kullanarak bir linked list tasarlayın. Bu listeye eleman ekleme, çıkarma, listeyi yazdırma, listeyi başlatma, liste dolu mu, liste boş mu fonksiyonlarını yazın ve çalıştırın.

Veri Yapıları

dinamik pointerla bağlı liste yapımı:

DOUBLE LİNKED LİST

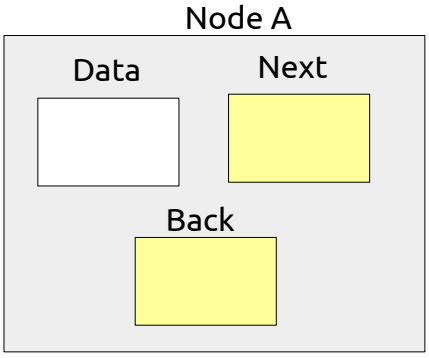
typedef int tamsayi ;

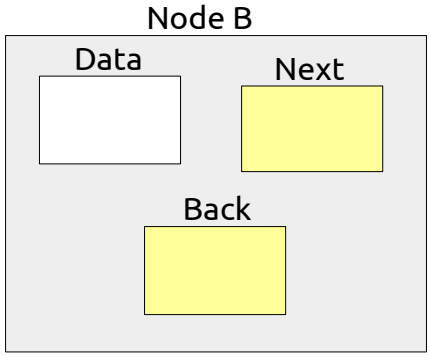
komutu int ifadesini tamsayi ifadesi ile değiştirir. Yani `int x;` ile `tamsayi x;` arasında hiçbir fark bulunmaz.

typedef int pointer*

Aynı şekilde `int* x;` ile `pointer x;` arasında hiçbir fark yoktur.

```
typedef Node* ptrtype;// Bundan sonra Node* yerine ptrpointer
kullanılacak.


struct Node{
    int data;
    struct Node* next;
    struct Node* back;

};

struct Node A;
struct Node B;
```






```
A.next = &B; // Node B yi işaret eder. Next komutu pointer olduğu
için & ile B nin adresini saklayabiliriz.

A.back = &A; // back komutu da aynı şekilde pointer olduğundan
dolayı adres saklayabilir.

A.next = B.back;
B.back->data = 15; // Burada B içerisindeki back pointerının işaret
ettiği yere gidilir. Oradaki Data 15 olarak atanır. Yani buradaki komuta
göre Node B nin Data kısmı 15 olarak değişir. Çünkü öncesinde B.back
```

A.next pointer ı ile eşitlenmiş, A.next ise öncesinde Node B nin adresini tutmaktadır.

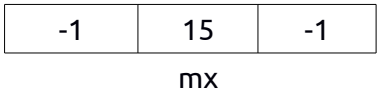
```
B.back->next->data =17;// Burada ise B nin next pointerının gösterdiği düğümün data kısmı 17 olarak değiştirilir.


getNode(ptrtype* k){
     *k = (ptrtype)malloc(sizeof(Node)); // Node struct yapısının boyutu kadar bir alan ayırır. Node un yapılanmasını aynen kopyalar ve bu yapının adresini bizim belirlediğimiz k pointer ı ile fonksiyona gönderilen adrese kopyalar. Yani başka bir deyişle dinamik olarak bellekten bir struct yapısı oluşturulmuş olur.
}
```

!!! getNode fonksiyonunun içerisine pointer adresi gönderilmesi gerekir.

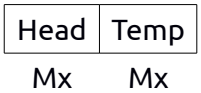
Veri Yapıları

```
struct Node{
    int Data;
    struct Node* next;
    struct Node* back;
};

struct Node A;
struct Node B;
struct Node* head;
struct Node* temp;
```

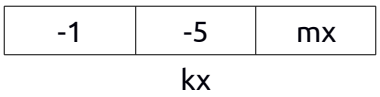

```
head = -1;

// 1. elemanı ekleme

temp = (Node*)malloc(sizeof(Node)); // bana Node büyüklüğünde bir alan ayır ve adresini döndür diyor.
```

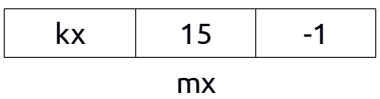



```
Temp->data = 15 // ayrılan alanın datasına 15 yazıldı.
Temp->back = -1;
Temp->next=-1;
head->temp;
```




```
// 2. elemanı ekleme

temp = (Node*)malloc(sizeof(Node));
```





```
Temp->data = -5 // ayrılan alanın datasına -5 yazıldı.
Temp->next=head;
temp->back=-1;
head->back = temp;
head->temp;
```

Head	Temp
kx	kx

// 3. elemanı ekleme

temp = (Node*)malloc(sizeof(Node));

-1	20	kx

cx

cx	-5	mx

kx

kx	15	-1

mx

Temp->data = 20 // ayrılan alanın datasına 20 yazıldı.
Temp->next=head;
temp->back=-1;
head->back = temp;
head->temp;

Head	Temp
kx	kx

Veriyi yazdırma

```
void yazdir(){
    if (head != -1){
        temp=head;
        while (temp!=-1){
            printf("%d",temp->Data);
            temp = temp->next;
        }
    }
}
```

Dugum Alma Fonksiyonu

```
void dugumver(Node** t1){
    *t1 = (Node*)malloc(sizeof(Node));
}

dugumver(&temp);// dugumu alacağımız yerde bu şekilde çağırabiliriz.
```

dugum ver fonksiyonu hafızadan bir yer alır ve adresini temp e verir. Bu sayede istediğimiz dugum adresini sadece fonksiyonu yazarak alabiliriz.

Eleman silme

eleman silmek istediğimiz zaman ilk kontrol etmemiz gereken silinecek eleman ilk eleman mı değil mi?

Ilk elemansa;

```
Temp  = head;
Head = head->next;
head->back = NULL;
free(temp);
```

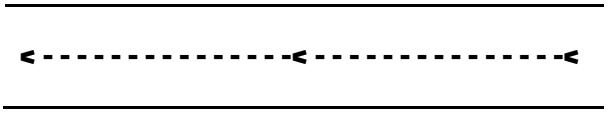
aradaki bir elemansa;

```
temp->back->next = temp->next;
temp->next->back = temp->back;
free(temp);
```

Veri yapıları

Queue

FIFO (first in first out) yani sıraya ilk giren ilk çıkar.



'head' ilk elemanı 'rear' ise son elemnı temsil eder.

kuyrukta hiç eleman yokken head ve rear birer pointer dır..

```
struct Node* head;
struct Node* rear;
struct Node* temp;
```

1. eleman ekleme

2. eleman ekleme


```
Temp->next = -1;
temp-> back;
rear->next = temp;
rear = temp;
```

Kuyruk mantığında araya eleman eklemek ve silmek mümkün değildir çünkü kuyruk mantığına aykırıdır. Kuyruğun linkedlist ten bir farkı yoktur sadece sondan ekleme yapılır kullanımı aynıdır.

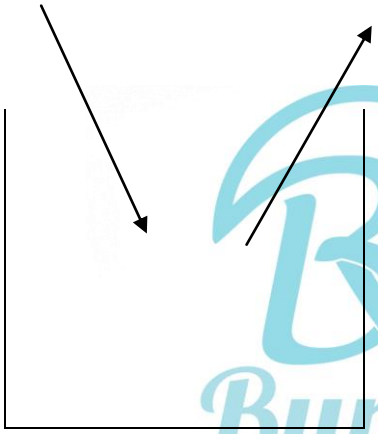
Ekleme Ve Silme Kodu:

```
int pop(){ eleman silme
    int k;
    if (head!=NULL){
        k=head->data;
       head->next->back = head->back; // burada kuyruktaki ilk eleman
     çıkarılmış oldu.
       head = head->next;

    }
}
void push(int eleman){//eleman ekleme
    getNode(&temp);
    temp->data = eleman;
    temp->next = NULL;
    temp-> back = rear;
```

```
    if (rear==NULL)
        head = temp;
    else
        rear->next = temp;

    //rear->next = temp; => bu kısım ilk elemanda çalışmaz bu yüzden üstteki
if-else komutu yazıldı...
    rear = temp;
}
```

Yığınlar

FILO (first in last out) yani ilk giren son çıkar. Bunlar işletim sisteminin omurgasını oluşturur. Derleyiciler de tamamen yığın yapısını kullanır.



Eklenen tüm elemanlar başa eklenir ve head olarak tanımlanır.

Ekleme ve Silme Kodu:

```
int pop(){// eleman silme
    int k;
    if (head!=NULL){
        k=head->data;
        head = head->next;
        head->back = NULL
    }
}

void push(int eleman){//eleman ekleme
    getNode(&temp);
    temp->data = eleman;
    temp->next = head;
    temp-> back = NULL;
    if (head!=NULL)
        head->back=temp;
    head = temp;
```

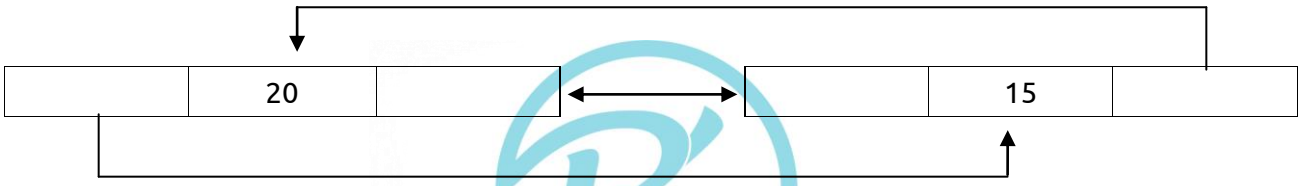
```
}
```

Dairesel Linkedlist:

Kendisinin adresi	15	Kendisinin adresi

head

```
Temp->data =15;
temp->next = temp;
temp->back = temp;
```




```
void push(int eleman){
    getNode(&temp);
    temp->data = eleman;
    if (head!=NULL){
        temp->next=head;
        temp->back = head->back;
        head->back = temp;
        temp->back->next = temp
        head = temp;
    }
    else{
        Temp->data =eleman;
        temp->next = temp;
        temp->back = temp;
    }
}
```


Editörlerdeki Stack (Yığın) Mantığı

Bir fonksiyonda derleme yapılırken başka bir fonksiyon çağırılırsa bir aktivasyon kaydı tutulur ve yığına atılır bu aktivasyon kaydında temel olarak o satırın adresi(en son bulunduğu) o ana kadarki değişkenlerin değerleri tutulur.

```
int main(int argc, char** argv) {
    int a=3, b=4;
    a=a+b;
    b=fonk1(a,b); // buraya kadar gelir ve fonk1 fonksiyonuna döner ve
elindeki işlemi bırakır.

    b=2*b;
    a=fonk2(a,b);
    b=a+b;
    a=fonk1(a,b);

    return (EXIT_SUCCESS);
}

int fonk1(int x, int y){
    x=x+y;
    x=fonk2(x.y);
    x=x-y;
    return x;
}

int fonk2(int x,int y){
    return x-y;
}
```

X=11,y=4 x= fonk2(11.4) adress:f2
A=7, b=4 b= fonk1(7.4); adress : 5

fonk1 içerisindeki fonk2 bittikten sonra yığındaki ilk elemana gidilir. Yığındaki ilk eleman bir sonraki yapılacak komutu belirtir. Ve yığından çıkarılır. Fonk1 bittikten sonra yine yığına gidilir ve ilk elemanda bulunan komut işlenir ve o satır yığından çıkarılır.

A=7,b=6 a=fonk2(7.6) adress 7

Fonk 2 bittikten sonra yine bu satır silinir.

X=2,y=1 x=fonk2(2.1) adress:f2
A=1,b=7 a= fonk1(1.7)

adress:9

Yine aynı işlemler uygulanarak program tamamlanır.

Recursive (Özyinelemeli) Algoritmalar

4! = 4.3!
3! = 3.2!
2! = 2.1!
1! = 1.0!
0! = 1

Faktöriyelin genel mantığı "faktoriyel(n) = n. faktoriyel(n-1);" şeklindedir

```
int Faktoriyel(int n){
    if(n==0)
        return 1;
    else
        return n*Faktoriyel(n-1);
}
```

N=1 return 1 adress: 4
N=1 return 1*F(0) adress: 4
N=2 return 2*F(1) adress: 4
N=3 return 3*F(2) adress: 4
N=4 return 4*F(3) adress:4

En son F(0) ile girdiğimizde return 1 komutu çalışır ve yığından veri çekmeye başlar ve ilk elemanı çeker:

1*1 = 1
2*1 = 2
2*3 = 6
6*4 = 24

ve burada yığın bittiği için ana fonksiyona döner.

Veri Yapıları

Fibonacci sayı sisteminin **recursive kodu**:

```
int main(int argc, char** argv) {

    printf("%d",fibonacci(7));
    return (EXIT_SUCCESS);
}

int fibonacci(int merak){
    int i=0;
    if (merak ==1 || merak == 2){
        return merak-1;
    }
    else{
        return fibonacci(merak-1)+ fibonacci(merak-2);
    }
}
```

çıktı :8

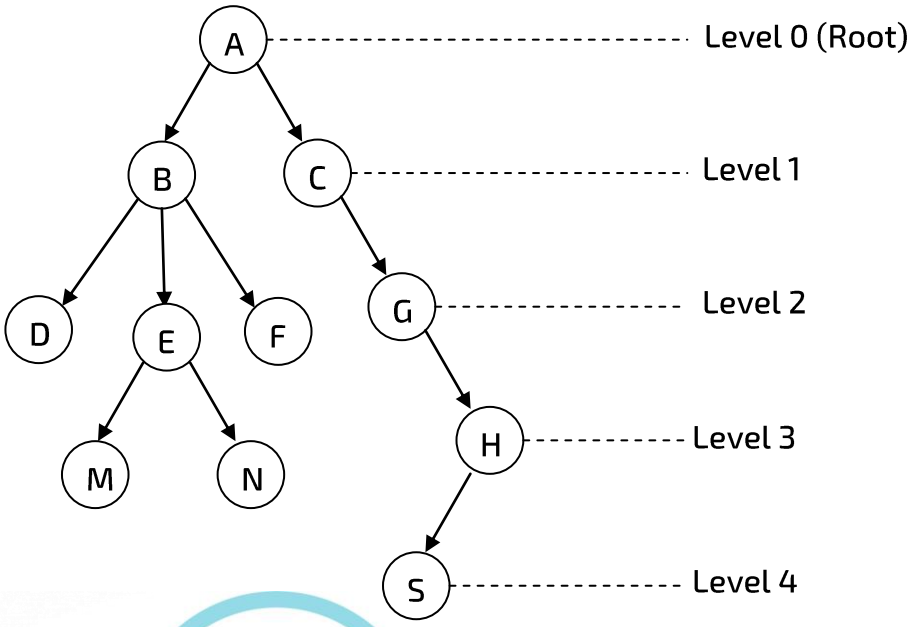
```
int main(int argc, char** argv) {

    suleyman(3);
    return (EXIT_SUCCESS);
}

void suleyman (int n){
1    if (n>0){
2        printf("%2d", n-1);
3        n=n-1;
4        suleyman(n);
5        printf("%2d\n",n);
    }
}
```


N=0 suleyman(0) adres:4
N=1 suleman(1) adres:4
N=2 suleyman(2) adres:4

Çıktı:
2 1 0 0
1
2
2,1,0 ilk printf den dönen değer,

0
1
2 ise ikinci printf den dönen değer

TREE (Ağaçlar)



D, M, N, F ve S çocuğu almadığı için kördüğümdür.
B Parent -> D, E, F nin annesi
Bu ağacın derinliği (depth): 4

BT (Binary Tree)

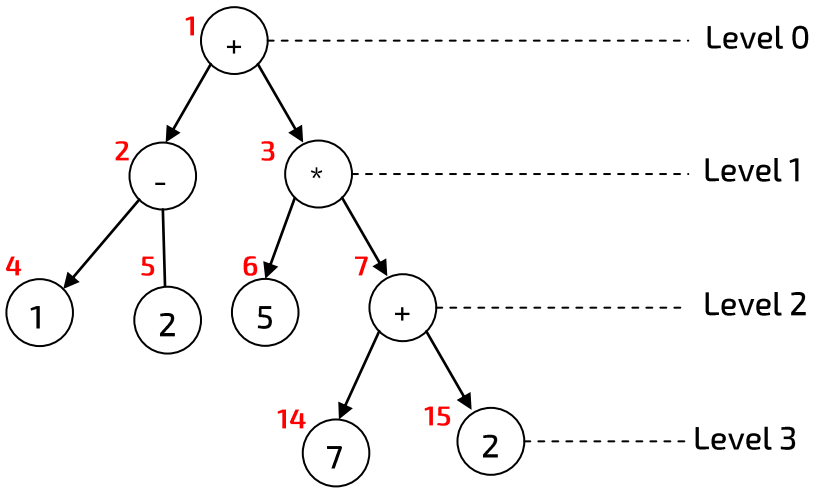
En önemli özelliği her düğümün en fazla 2 tane çocuğu olabilir.

Binary tree nin uygulanması

→ Lineer diziler ile yapılabilir.
→ Diziler ile linkedlist kullanımı ile yapılabilir.
→ Linkedlistin dinamik pointer kullanımı ile yapılabilir.

Lineer Dizi ile Binary Tree Yapımı

n seviyeyi göstermek üzere bir $2^{(n+1)}$-1 tane eleman bulunabilir.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	-	*	1	2	5	+	…	…	…	…	…	…	7	2

Dezavantajı:

Sadece sağ kolda eleman varsa 4. seviyede 5 eleman için 31 karakter alan ayrılacak ve taşıma işlemi yapılacağı zaman veya araya eleman ekleneceği zaman tüm diğer elemanları da oynatmamız gerektiği için tercih edilmeyen bir yöntemdir.

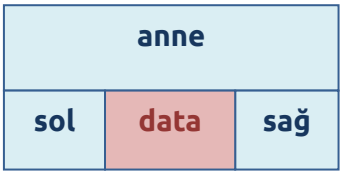
Lineer Dizi + Linkedlist ile Binary Tree Yapımı

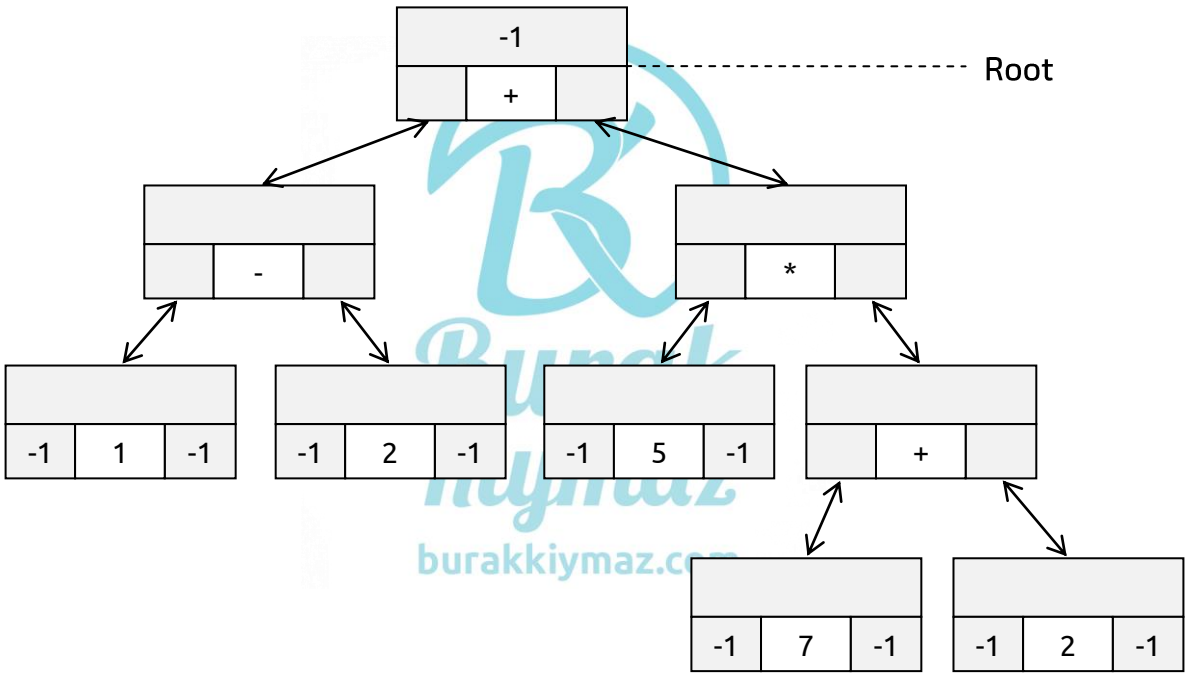
DATA	SOL	SAĞ
+	2	3
-	4	5
*	6	7
1	-1	-1
2	-1	-1
5	-1	-1
+	8	9
7	-1	-1
2	-1	-1

Lineer diziye göre daha kullanışlı bir yöntem fakat 9 eleman eklemek için 27 karakterlik alan ayrıldı. Eleman ekleme ve silme işlemi normal Linkedlist mantığıyla yapılır.

Dinamik Pointer ile Binary Tree

```
struct Node{
    char data;
    struct Node* sag;
    struct Node* sol;
    struct Node* anne ;
};
```

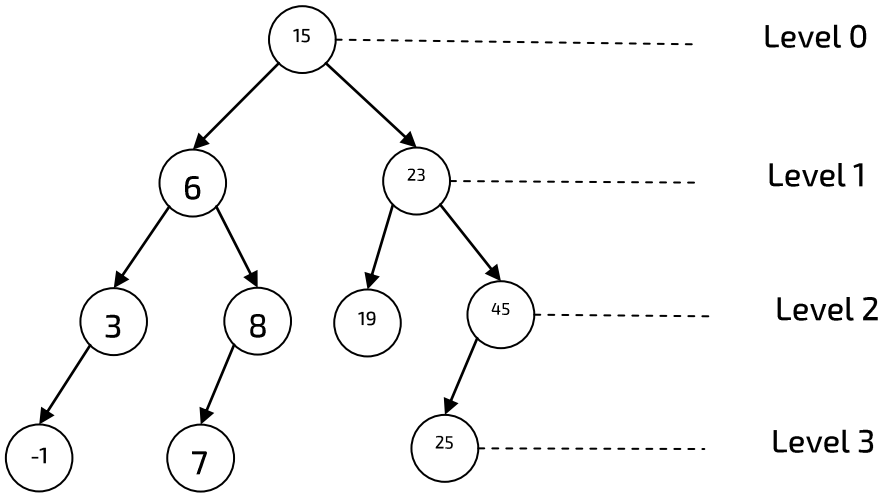




BST (Binary Search Tree)

Bir düğümün sol dalındaki değerler o düğümden küçük eşit, sağındakiler büyük olacaktır. Verilen diziyi okumaya sol baştan başlanır.

[15, 23, 6, 8, 45, 3, 7, -1, 19, 25] dizisini Binary Search Tree ile dizmek istersek;



Binary Tree de arama yaparken bulmak istediğimiz sayının elimizdeki düğümden büyük mü, küçük mü olduğuna bakılır ve buna göre bir eleme işlemi yapılır.

ÖDEV:
Verilen bir diziyi Binary Search Tree ye yerleştiren programı yaz. (Dinamik pointer Linkedkist kullanılarak yapılacak.)

Veri Yapıları

Binary Search Tree

3 yönlü dinamik linked list ile yapılır.

```
node* head, *temp, *p;
```

elemanlarmıız var.

Binary Search Tree ye Eleman Ekleme

Mantık

→Elemanın ekleneceği anne düğümü bul

→O annenin uygun dalına yerleştir.

P eklenecek elemanı göstersin

```
typedef struct linkedList llist;
struct linkedList{
    int data;
    llist* mother;
    llist* left;
    llist* right;
};

llist* head, *temp, *p, *tempMom;


void ekle(int eleman){
    getNode(&p);
    p-> data = eleman;
    p->left = NULL;
    p->right = NULL;

    if (head == NULL){
        p->mother = NULL;
        head = p;
    }

    else{
```

```
        temp = head;
        while(temp!=NULL){
            tempMom = temp;
            if (eleman <= temp->data)
                temp = temp->left;
            else
                temp = temp->right;
        }
    } // burada anne düğüm bulundu.


    p->mother = tempMom;
    if (eleman <= tempMom->data)
        tempMom->left = p;
    else
        tempMom->right = p;
}
```

Binary Search Tree den Eleman Silme

Bst den elaman sileceksek şu üç koşulu göz önünde bulundurmamız gerekmektedir.

→kör düğümse

→tek çocuğu varsa

→iki çocuğu varsa


```
void sil(){
  // p nin sağ ve solu NULL ise kör düğüm demektir.
  // case 1
  if (p->data<=p->mother->data)
    p->mother->left = NULL;
  else
    p->mother->right = NULL;

  if (p->left!= NULL){

  }
```

```
    // case 2
    // temp de p nin çocuğunu göstersin

    if (p->right = NULL){
        temp = p->left;
    }
    else
        temp = p->right;

    temp->mother = p->mother;
    if (p==p->mother->right)
        p->mother->right = temp;
    else
        p->mother->left = temp;
    /*
     * case 3
     * p nin iki çocuğu varsa
     * ya sol daldaki en büyük eleman bulunur ya da sağdaki en küçük eleman bulunur.  temp onu gösterir.
     * bunu düğümü p nin olduğu yere taşırız ve p nin tüm bağlantılarını keseriz.
     */

    temp->left->mother = temp->mother;
    temp->mother->right = temp->left;

    temp->mother = p->mother;
    p->mother->right = temp;

    temp->left = p->left;
    temp->left->mother = temp;

    temp->right = p->right;
    p->right->mother = temp;

    free(p);

}
```

Binary Search Tree de Dolaşma

3 türlü dolaşım vardır.

→inorder dolaşım

→preorder dolaşım

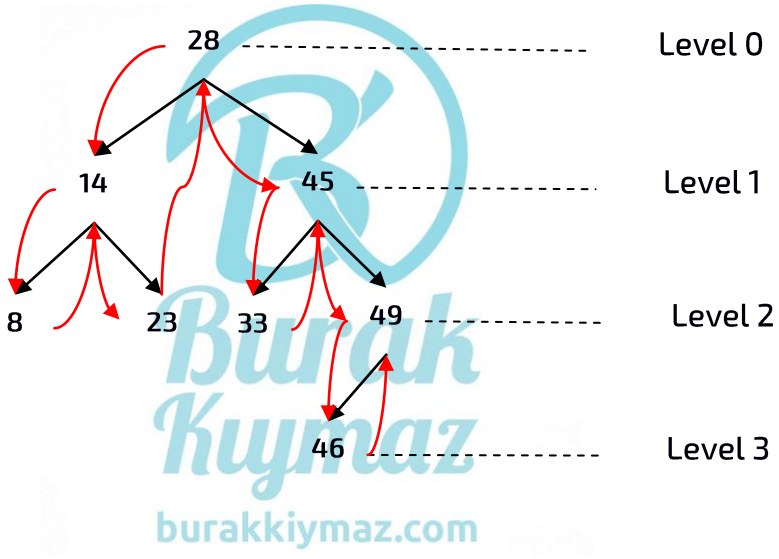
→postorder dolaşım

Inorder Dolaşım

sol-kök-sağ şeklindedir.

3

5 7 → şeklindeki bir ağaçta önce 5 sonra 3 en son 7 yazılır.



8, 14, 23, 28, 33, 45, 46, 49

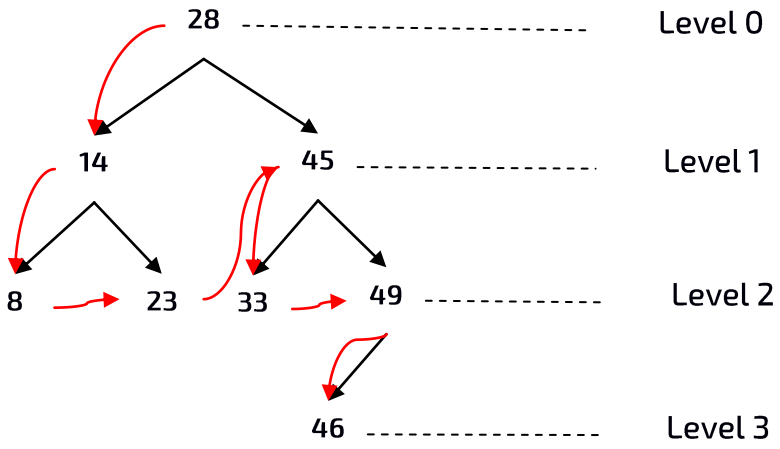
Seçilen eleman eğer kökse ona da bu işlemler uygulanır ta ki kör düğüm bulunana kadar.

Inorder Dolaşım Kodu

```
void inorder(llist* temp){
    if (temp!=NULL){
        inorder(temp->left);
        printf("%d",temp->data);
        inorder(temp->right);
    }
}
```

Preorder Dolaşım

kök-sol-sağ şeklindedir.



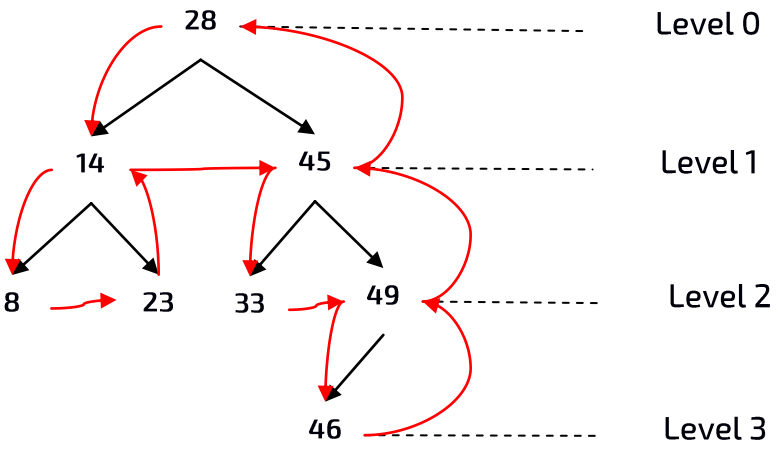
28, 14, 8, 23, 45, 33, 49, 46

Preorder Dolaşım Kodu

```
void preorder(llist* temp){
   if (temp!= NULL){
      printf("%d",temp->data);
      preorder(temp->left);
      preorder(temp->right);
   }
}
```


Postorder dolaşım

sol-sağ-kök şeklinde dolaşım yapılır.



8, 23, 14, 33, 46, 49, 45, 28

Postorder Dolaşım Kodu

```
void postorder(llist* temp){
    if (temp!=NULL){
        postorder(temp->left);
        postorder(temp->right);
        printf("%d",temp->data);
    }
}
```

Bir ifadeden bir ağacı elde etmek için Inorder+ Preorder veya Inorder + Postorder ile ağaç elde edilebilir

ÖRNEK:

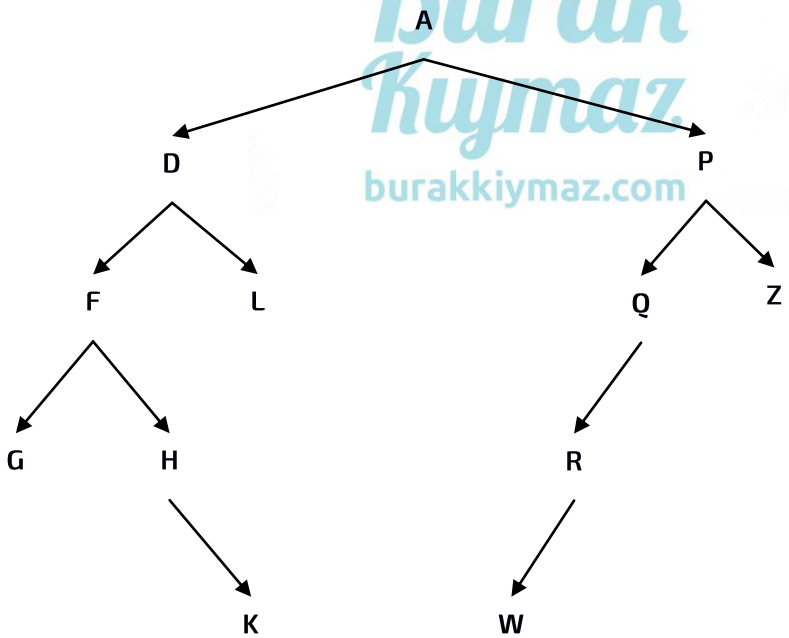
PREORDER: A D F G H K L P Q R W Z

INORDER: G F H K D L A W R Q P Z

Ağacı Bulunuz…

Preorder da men başta A olduğundan dolayı kök(root) A olur.

Inorder da A nın solundakiler hep solda sağındakiler hep sağda kalır.



06.01.2016

Veri Yapıları

Aritmetik İfadeler

Üç ifade yöntemi vardır
- Infix (aratakı)
- Prefix (öntakı)
- Postfix (sontakı)

INFIX

2 + 3 * 5 – 7 / 4 - 2
elemanlarına **Operand**(terim) aradakilere ise **Operatör** denir

PREFIX

Operator terimlerin arasından alınıp sola yazılır.

2*5 -> Infix ifade
*25 -> Prefix ifade

POSTFIX

Oparatör terimlerin sağına alınır.
25* -> Postfix ifade

Infix ifadeyi Postfix e dönderme

- 2+3*5-7/4^2-7
- 2+(3*5)-(7/(4^2))-7
- 2+(35*)-(7(42^)/)-7
- 235*+742^/-7-

… Compilerlar yazılan bir aritmetik ifadeyi postfix e dönüştürüp öyle işlem yapar.

Prefix ile aynı işlem → - - + 2 * 3 5 7 / 7 ^ 4 2 7

Infix To Postfix Algorithm

Operator	^	*	/	+	-	(	)
Öncelik	3	2	2	1	1	4	0

- Infix ifadeyi soldan itibaren tek tek oku;
- Gelen değer terim ise doğrudan göster, gelen değer operatör ise bekle(yığına ata)
  - ✓ Sol parantez ise doğrudan yığına at
  - ✓ Gelen operatorün önceliği yığının top elemanından büyükse yığına at, küçükse yığın tepesindeki elemanın önceliği bizim elemanımızın önceliğinden küçük oluncaya kadar yığını boşalt.
  - ✓ Sağ parantez ise yığındaki sol paranteze kadar ne varsa çıkar ve göster.

NOT: Sol parantez girerken en büyük önceliği sahiptir fakat yığın içerisinde önceliği 0 olur.

06.01.2016

A * (B + C – D / E) / F

Gösterim	Yığın
ABC+D/-*F/	*(2) **(**(0) +(1)(- GELDİ GİREMEZ + SOLA GİTTİ) -(1) **/**(2) **)** (0) (')' GELDİ BURAYA KEDER HEPSİNİ ÇIKAR)

(2 + 8 / 4) ^ (2 – 4) * 2

Gösterim	**Yığın**
284	((0)+/
284/+	
284/+24	^(-
284/+24-^	
284/+24-^2	*
284/+24-^2*	

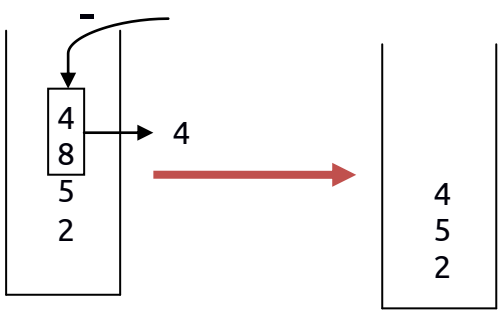
Infix To Prefix Algorithm

- İfadeyi ters çevir.
- Infix to postfix algoritmasını çalıştır
- Çıkan ifadeyi ters çevir.

Postfix İfadenin Hesaplanması

- İfade soldan itibaren okunur.
- Gelen eleman terim ise yığına atılır, gelen eleman operatör ise yığındaki en top iki terim çıkarılır. (ilk eleman sağa ikincisi sola gelecek şekilde) operatör bunlara uygulanır. Çıkan sonuç tekrar yığına atılır.
- İfade bitince yığındaki son eleman sonuçtur.

2+5*(8-4)
2584-*+



06.01.2016

Prefix İfadenin Hesaplanması

- Prefix ifade sağdan sola doğru taranır.
- Gelen eleman terim ise yığına atılır operatör ise top iki eleman yığından çıkarılır. (ilk çıkan eleman sola ikinci sağa gelecek şekilde) operatör bu elemanlara uygulanır ve çıkan sonuç tekrar yığına atılır.