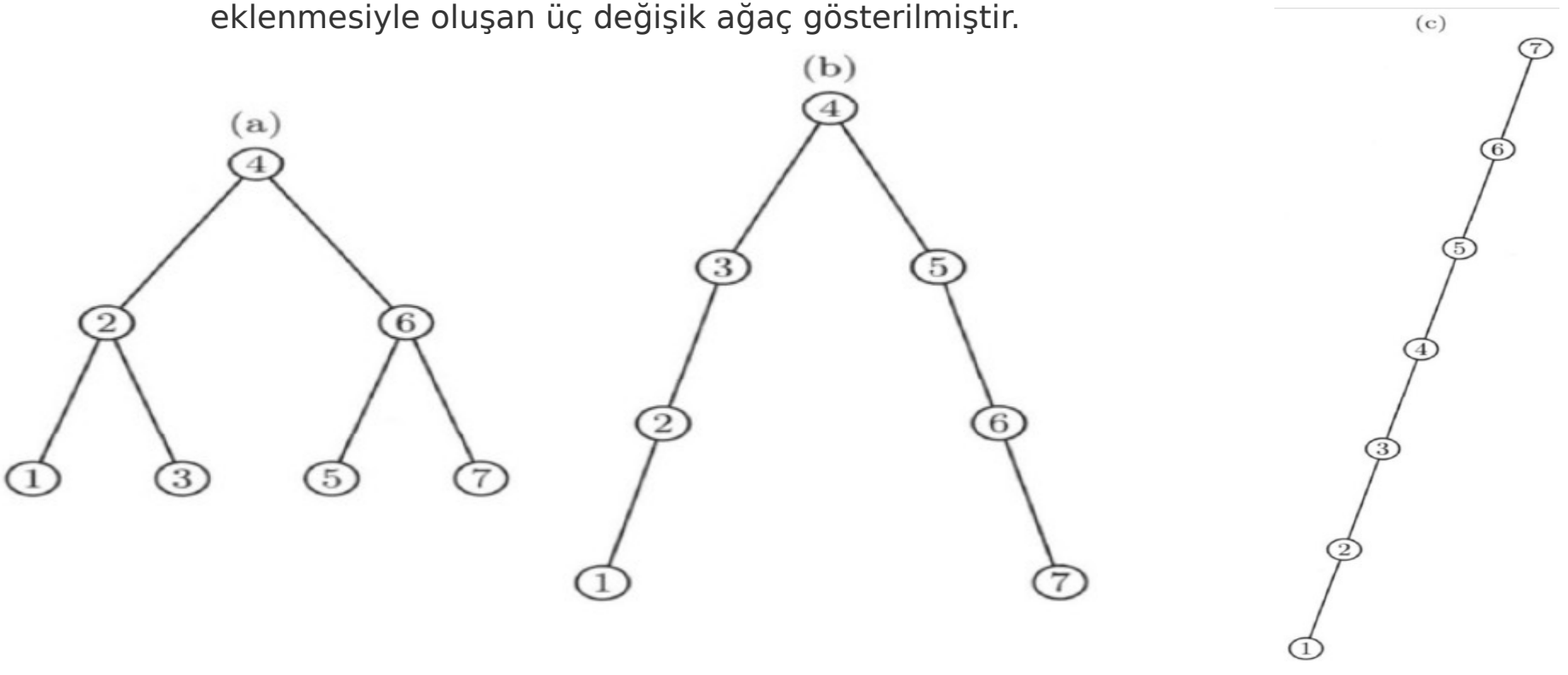
Algoritma Analizi

Ders 12

Doç. Dr. Mehmet Dinçer Erbaş
Bolu Abant İzzet Baysal Üniversitesi
Mühendislik Fakültesi
Bilgisayar Mühendisliği Bölümü

AVL Ağaçları

- İkili arama ağaçlarını hatırlayalım:
  - İkili arama ağacı dengeli olduğu sürece ağaçta belirli bir elemanı aramak, eleman eklemek, eleman silmek gibi işlemler O(lg N) zamanda yapılabilmektedir.
  - Ağaç dengesiz olduğunda bu süre O(N) zamana kadar çıkabilir.
  - Aşağıda 1 ile 7 arası değerleri içeren düğümlerin farklı sırada ikili arama ağacına eklenmesiyle oluşan üç değişik ağaç gösterilmiştir.



AVL Ağaçları

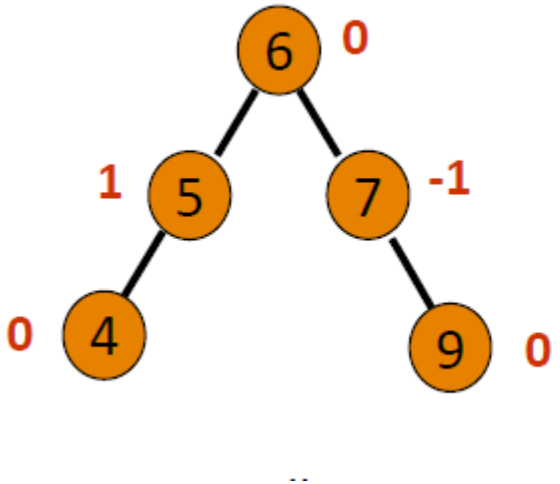
- İkili arama ağaçlarını hatırlayalım:
  - Sayıların hangi sırayla geleceği önceden bilinemeceğinden önceki slaytta görülen üç ağaçla da karşılaşılabilinir.
  - Bu durumda dengeli ağaçlar, yani düğümlerin her seviyeyi yeterince doldurarak yerleştirildiği ağaçlar, tercih edilir.
  - Dolayısıyla, ağacın dengeli olduğunu garanti edebilmek için dengenin bozulduğu durumda ağacın tekrar dengelenmesi gerekir.

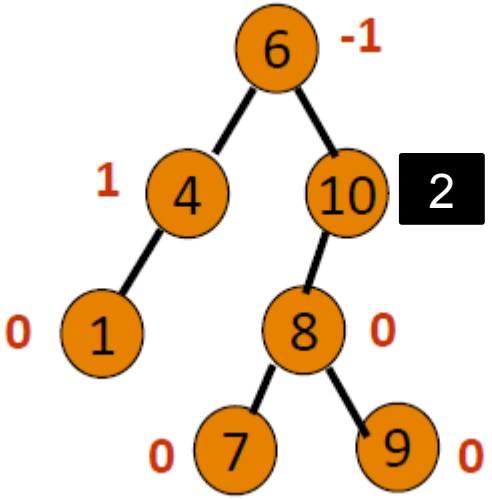
AVL Ağaçları

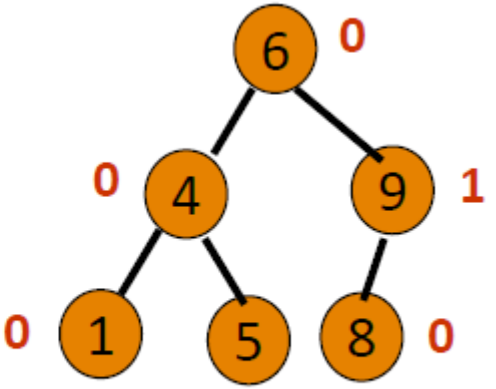
- AVL yöntemi sovyet bilimadamları G.M. Adelson-Velskii ve E.M. Landis tarafından geliştirilmiştir.
- AVL ağacı bir ikili arama ağacıdır.
  - Yani ikili arama ağacı özelliğini taşırlar.
- AVL ağacının ek özelliği, her düğümün sağ alt ağacı ile sol alt ağacı arasındaki yükseklik farkının en fazla bir olmasıdır.
  - Bu koşul doğru olduğunda ağacın dengesi sağlanmış olur.
  - Bir ikili arama ağacında herhangi bir düğümün sağ ve sol alt ağaçlarının yükseklik farkı 1’den büyükse o ikili arama ağacı, AVL ağacı değildir.
- Normal ikili arama ağaçları için ekleme ve silme işlemleri ağacın orantısız büyümesine, yani ağacın dengesinin bozulmasına neden olabilir.
- Bir dalın fazla büyümesi ağacın dengesini bozar ve ağaç üzerinde yapılacak olan işlemlerin alacağı süre artmaya başlar.
- Öyleyse ağacın dengesi bozulduğunda ağacı yeniden dengelemek gerekir.

AVL Ağaçları

- AVL ağacında bilinmesi gereken bir kavram denge faktörüdür (Balance Factor).
- Ağacın her düğümü için denge faktörü şu şekilde hesaplanır:
  - Denge faktörü = Sol ağacın yüksekliği – Sağ ağacın yüksekliği
- Denge faktörünün -1,0,1 arasında değerler alabilir.
- Denge faktörü bunun dışında bir değer aldığında ağacın dengesi bozulmuş demektir.

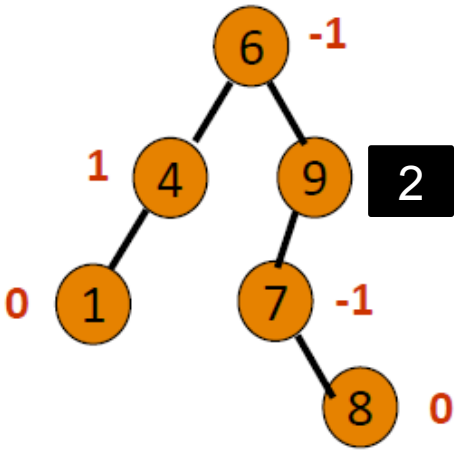


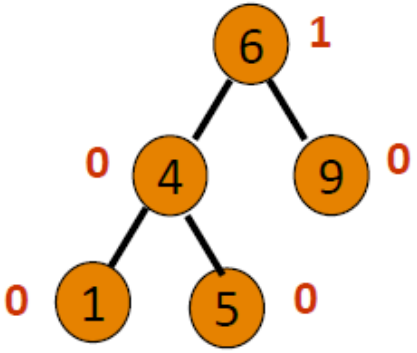


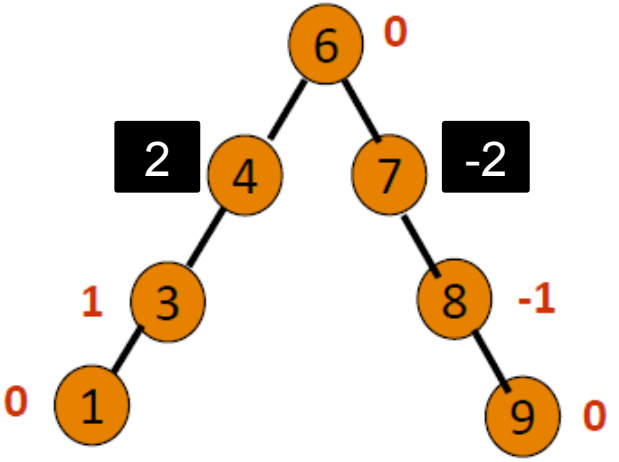


AVL Ağaçları

- AVL ağacında bilinmesi gereken bir kavram denge faktörüdür (Balance Factor).
- Başka örnekler.







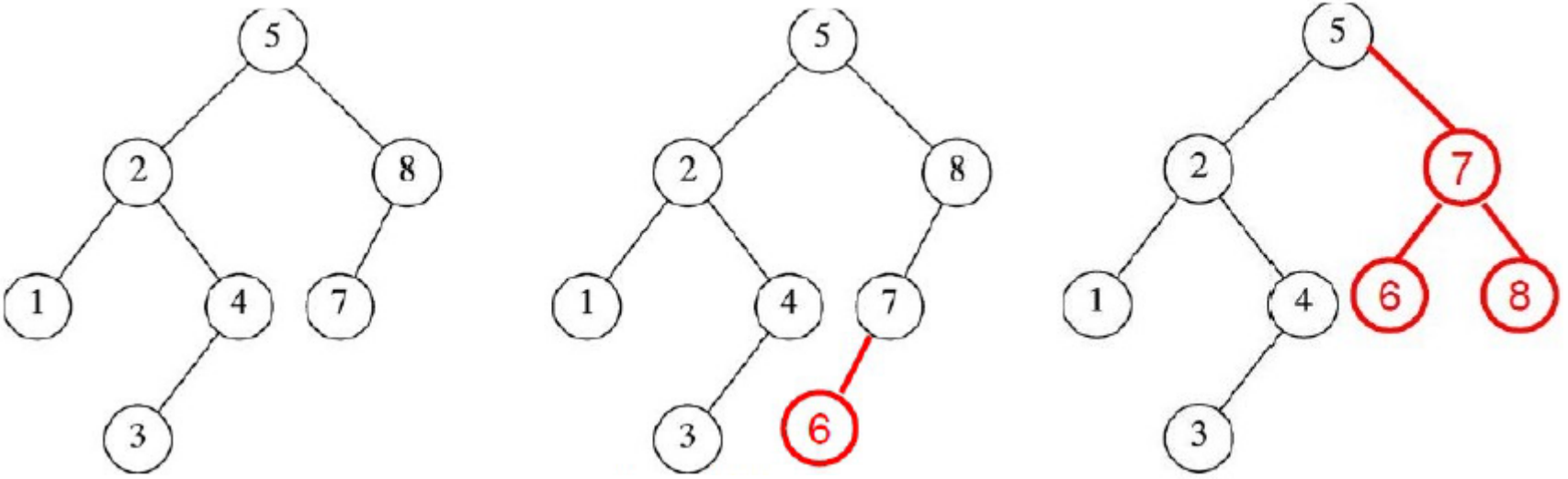
AVL Ağaçları

- AVL ağacında denge faktörünü hesaplayabilmek için her düğümde yükseklik bilgisi saklanmalıdır.
- Bu bilgiyi içeren AVL ağacı düğüm tanımı şu şekilde yapılabilir:

```
typedef struct node
{
        int key;
        struct node *left;
        struct node *right;
        int height;
}node;
```

AVL Ağaçları

- AVL ağacına yeni bir düğüm eklendiğinde, düğümün eklendiği taraftaki herhangi bir ata düğümün denge faktörü -2 veya 2 olabilir, yani dengesi bozulabilir.
- Bu durumun tespiti için yeni eklenen düğümün bütün ata düğümlerinin denge faktörünün hesaplanması gerekir.
- Denge faktörü bozulan düğümler üzerinde döndürme işlemi yapılarak denge faktörünün düzeltilmesi sağlanmaldır.

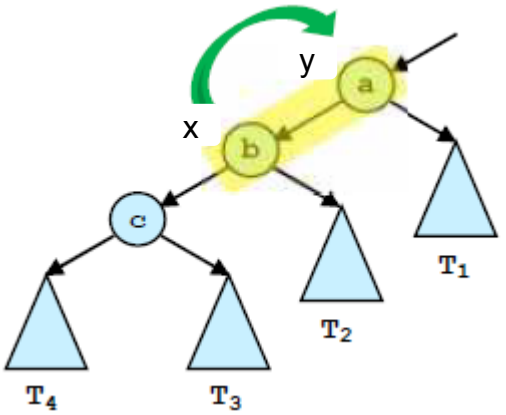


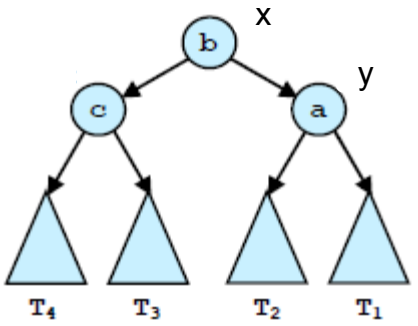
AVL Ağaçları

- Döndürme işlemi:
  - AVL özelliği bozulmuş düğüm d olsun.
    - İkili arama ağacında bir düğümün en fazla iki çocuğu olduğundan, dengesizlik ancak d’nin iki alt ağacının boylarının farkının iki olmasıyla mümkündür.
  - Yeni eklenen düğümün d’ye göre pozisyonuna göre 4 farklı durumdan bahsedebiliriz.
    - 1. d'nin sol çocuğunun sol alt ağacına eklenebilir.
    - 2. d'nin sol çocuğunun sağ alt ağacına eklenebilir.
    - 3. d'nin sağ çocuğunun sol alt ağacına eklenebilir.
    - 4. d'nin sağ çocuğunun sağ alt ağacına eklenebilir.
  - 1 ve 4. durumlar birbirine benzerdir. Aynı şekilde 2 ve 3. düğümler birbirine benzerdir.
    - 1 ve 4. durumlar tek rotasyon ile çözülebilir.
    - 2 ve 3. durumlar çift rotasyon ile çözülebilir.

AVL Ağaçları

- Döndürme işlemi:
  - Tek rotasyon (sağ dönüş)
  - Sol çocuğun sol alt ağacına düğüm eklendiğinde denge faktörü bozulmuşsa sağ dönüş yapılır.

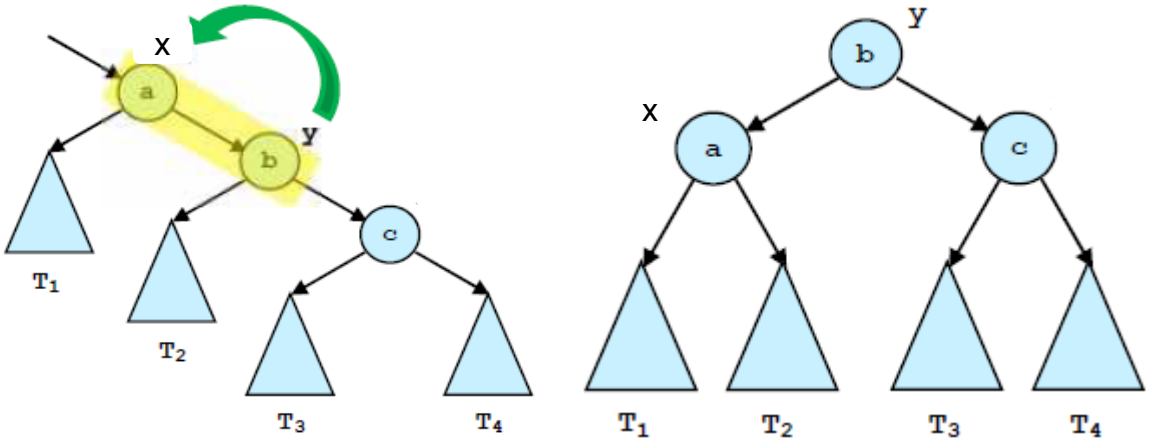





```
node *rightRotate( node *y)
{
        node *x = y->left;
        node *T2 = x->right;

        x->right = y;
        y->left = T2;

        y->height = max(height(y->left), height(y->right))+1;
        x->height = max(height(x->left), height(x->right))+1;

        return x;
}
```

AVL Ağaçları

- Döndürme işlemi:
  - Tek rotasyon (sol dönüş)
  - Sağ çocuğun sağ alt ağacına düğüm eklendiğinde denge faktörü bozulmuşsa sol dönüş yapılır.

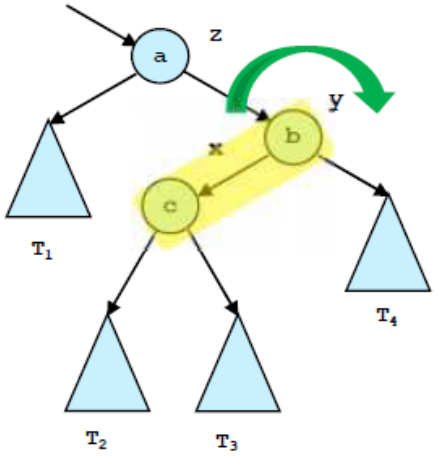


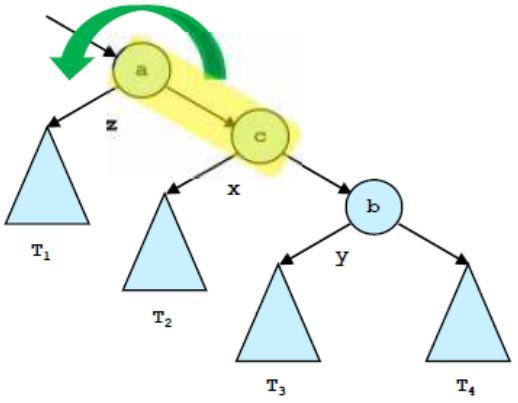

```
node *leftRotate( node *x)
{
        node *y = x->right;
        node *T2 = y->left;

        y->left = x;
        x->right = T2;

        x->height = max(height(x->left), height(x->right))+1;
        y->height = max(height(y->left), height(y->right))+1;

        return y;
}
```

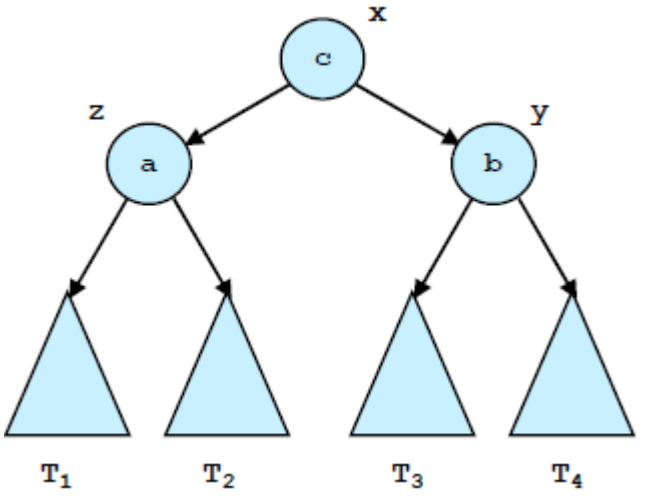
AVL Ağaçları

- Döndürme işlemi:
  - Çift rotasyon (sağ – sol dönüş)
  - Sağ çocuğun sol alt ağacına düğüm eklendiğinde denge faktörü bozulmuşsa, sağ – sol dönüş yapılır.

```
z->right = rightRotate(z->right);
return leftRotate(z);
```



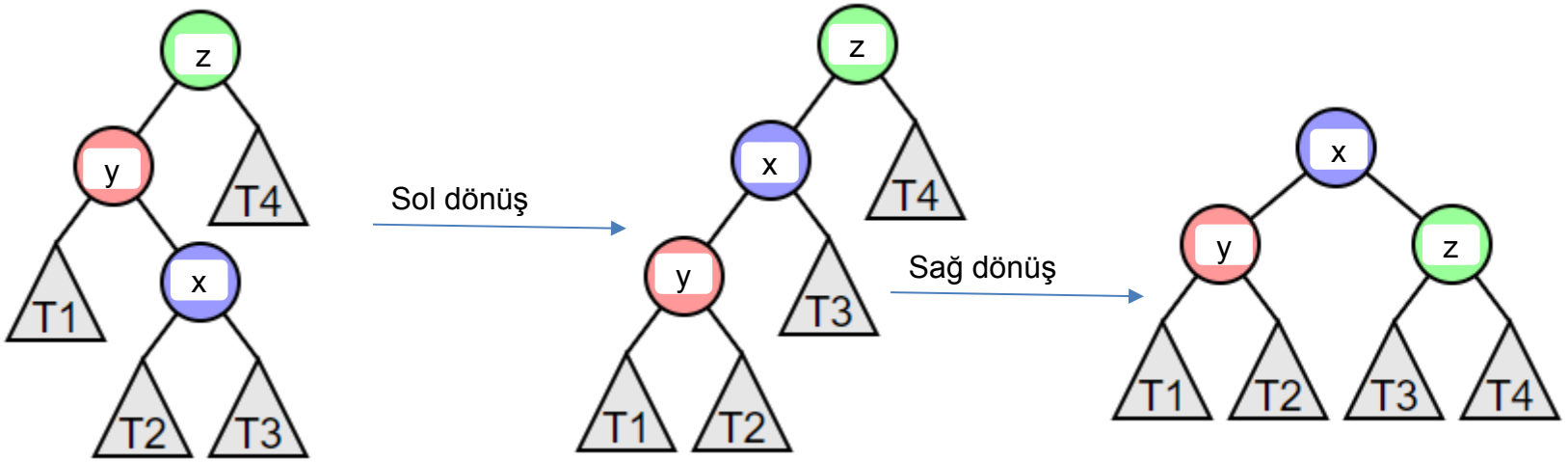




AVL Ağaçları

- Döndürme işlemi:
  - Çift rotasyon (sol – sağ dönüş)
  - Sol çocuğun sağ alt ağacına düğüm eklendiğinde denge faktörü bozulmuşsa, sol – sağ dönüş yapılır.

```
z->left = leftRotate(z->left);
rightRotate(z);
```



AVL Ağaçları

- Döndürme fonksiyonlarını kullanarak AVL ağacına düğüm ekleme fonksiyonunu yazabiliriz.
  - Ayrıca ekleme fonksiyonu aşağıda belirtilen iki fonksiyonu kullanacak.

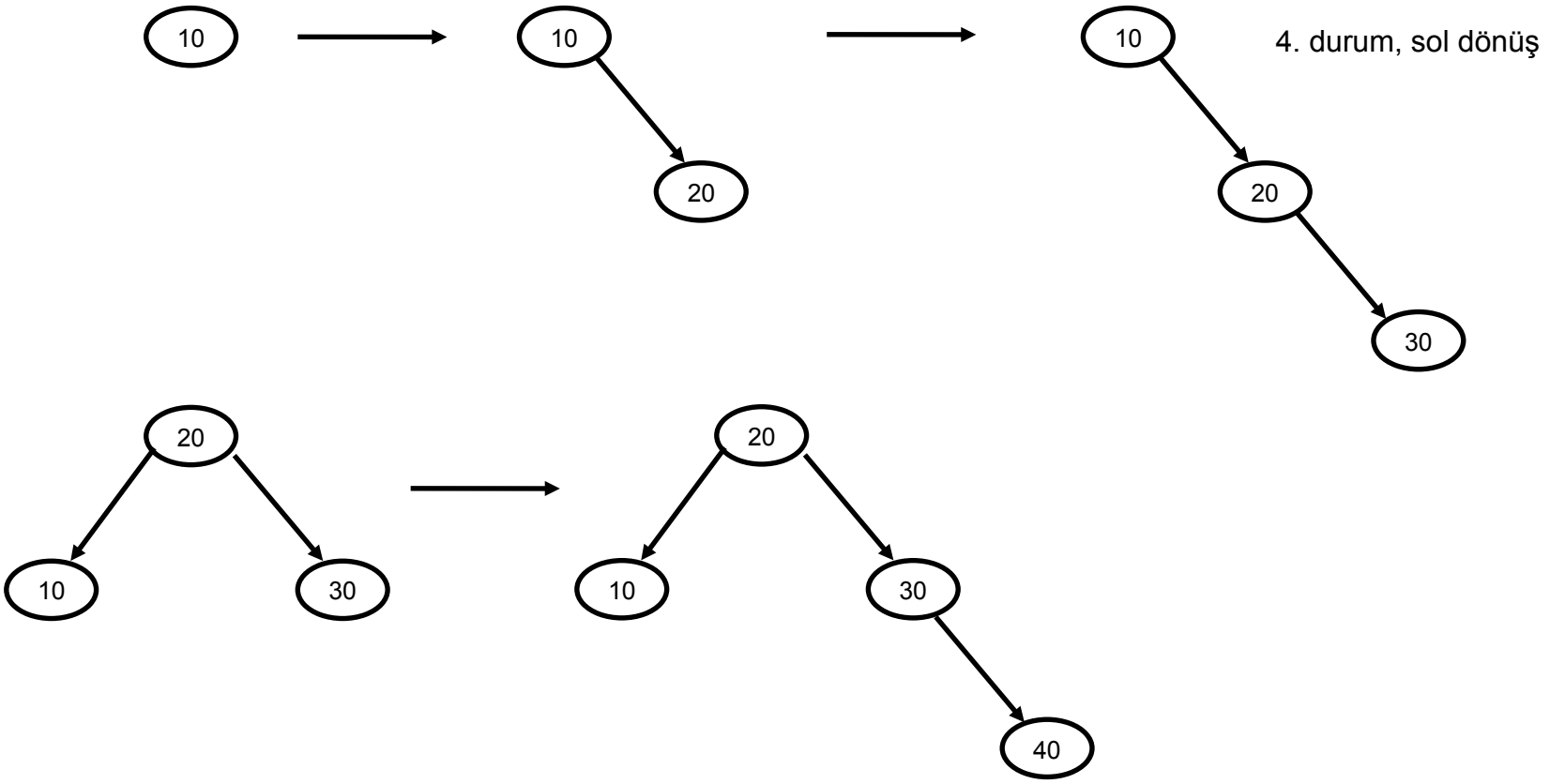
```
int height( node *N)
{
        if (N == NULL)
                return 0;
        return N->height;
}

int max(int a, int b)
{
        return (a > b)? a : b;
}
```

AVL Ağaçları

```
node* insert( node* node, int key)
{
            if (node == NULL)
                        return(CreateNewNode(key));

            if (key < node->key)
                        node->left = insert(node->left, key);
            else
                        node->right = insert(node->right, key);

            node->height = 1 + max(height(node->left),height(node->right));

            int balance = height(node->left) - height(node->right);

            // Durum 1
            if (balance > 1 && key < node->left->key)
                        return rightRotate(node);
            // Durum 4
            if (balance < -1 && key > node->right->key)
                        return leftRotate(node);
            // Durum 2
            if (balance > 1 && key > node->left->key)
            {
                        node->left = leftRotate(node->left);
                        return rightRotate(node);
            }
            // Durum 3
            if (balance < -1 && key < node->right->key)
            {
                        node->right = rightRotate(node->right);
                        return leftRotate(node);
            }

            return node;
}
```

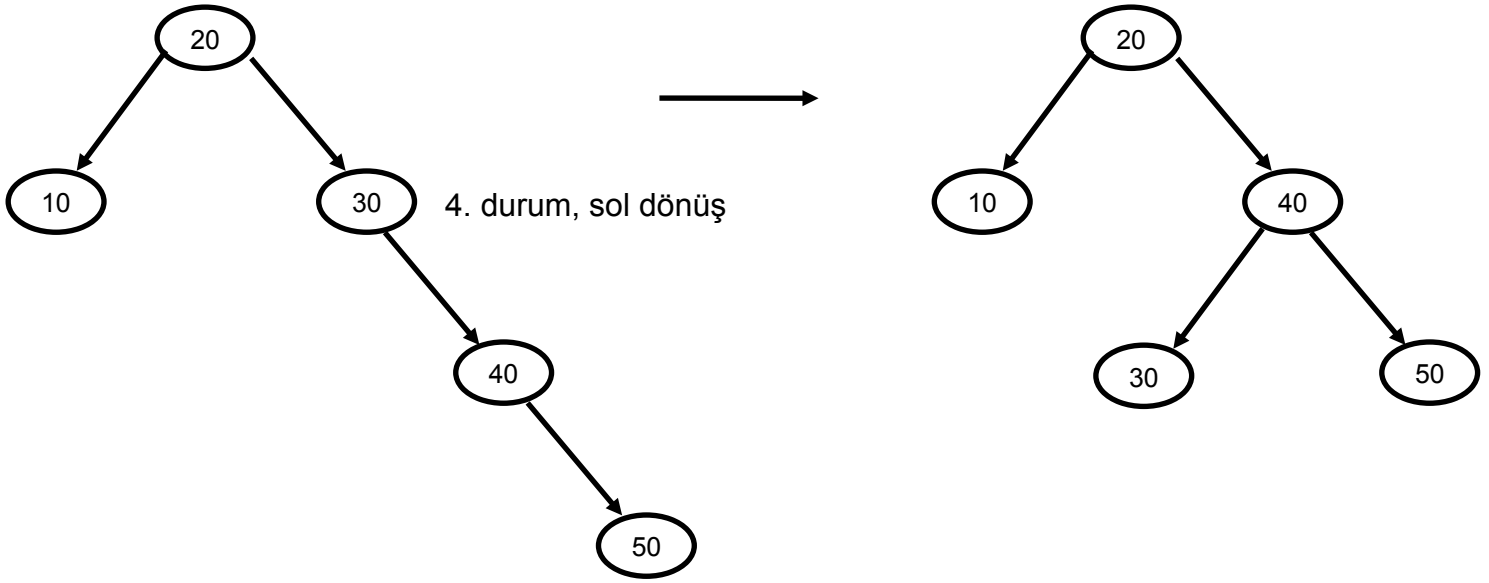
AVL Ağaçları

- Örnek: Yandaki düğümleri sırasıyla ağacımıza ekleyelim (10,20,30,40,50,25)



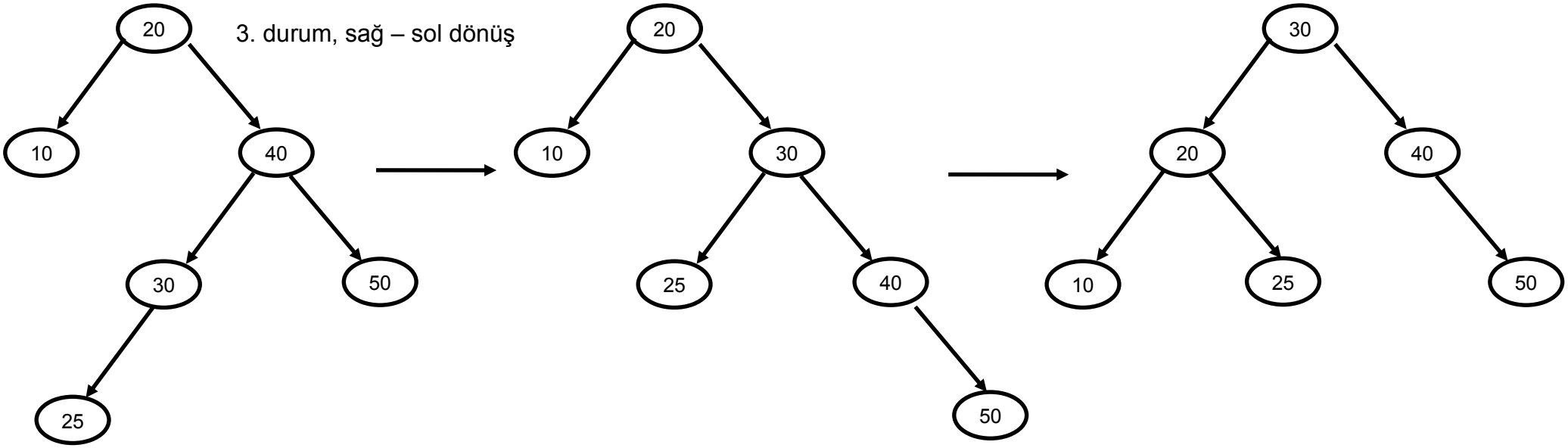
AVL Ağaçları

- Örnek: Yandaki düğümleri sırasıyla ağacımıza ekleyelim (10,20,30,40,50,25)



AVL Ağaçları

- Örnek: Yandaki düğümleri sırasıyla ağacımıza ekleyelim (10,20,30,40,50,25)



AVL Ağaçları

- AVL ağaçları ağaçtaki herhangi bir düğümün denge faktörü bozulduğunda rotasyonlar sayesinde denge faktörünü tekrar sağlar
- Bu sayede ağacın dengeli olacağının garantisi verilmiş olur.
- Ağaç dengeli olduğuna ekleme, silme, arama, en küçük, en büyük gibi dinamik küme operasyonlarının tamamı AVL ağaçlarında O(lg n) sürede yapılır.
    - Dengeyi tekrar sağlama işlemi algoritmanın karmaşıklığını etkilemez.

Hash Tabloları

- Birçok farklı uygulama dinamik bir küme üzerinde INSERT, SEARCH ve DELETE gibi sözlük işlemlerin yapılmasını gerektirir.
  - Örneğin bir derleyici yazdıysanız, bu derleyicinin önemli bir parçası sembol tablosudur.
  - Sembol tablosu derlenen programın içerdiği değişken isimlerini ve özelliklerini içerecektir.
  - Bu tabloya birçok kez erişip ilgili değişkenin bilgilerinin okunması gerekir.
- Hash tabloları (İng: hash tables) bu gibi sözlük işlemlerinin yapılmasına olanak veren bir veri yapısıdır.
  - Ayrıca komut çizelgesi, çırpma tablosu, anahtarlı tablo olarak da bilinir.
- SEARCH işlemi en kötü durumda $\Theta(n)$ sürede çalışabilir. Ancak makul varsayımlar ile $O(1)$ sürede aranılan elemana erişmek mümkündür.

Hash Tabloları

- Hash tabloları birden fazla nesneyi saklarken kullandığımız basit dizilere benzer.
  - Dizinin herhangi bir indeksinde bulunan elemana O(1) sürede erişebiliriz.
  - Elemanın indeksini kullanarak belli bir pozisyona erişmeye açık adresleme denir.
  - Her eleman için bir pozisyon var ise açık adresleme kullanılabilir.
- Hash tabloları gerçekte saklayacağımız anahtar sayısı, anahtarların alabileceği değer kümesi ile karşılaştırıldığında küçük ise mantıklı bir seçim haline gelir.
  - Hash tabloları genelde saklanacak eleman sayısı ile orantılı büyüklükte diziler kullanırlar.
  - Anahtarı kullanılan dizideki indeks değeri olarak kullanmak yerine, indeks anahtar kullanılarak hash fonksiyonu ile hesaplanır.

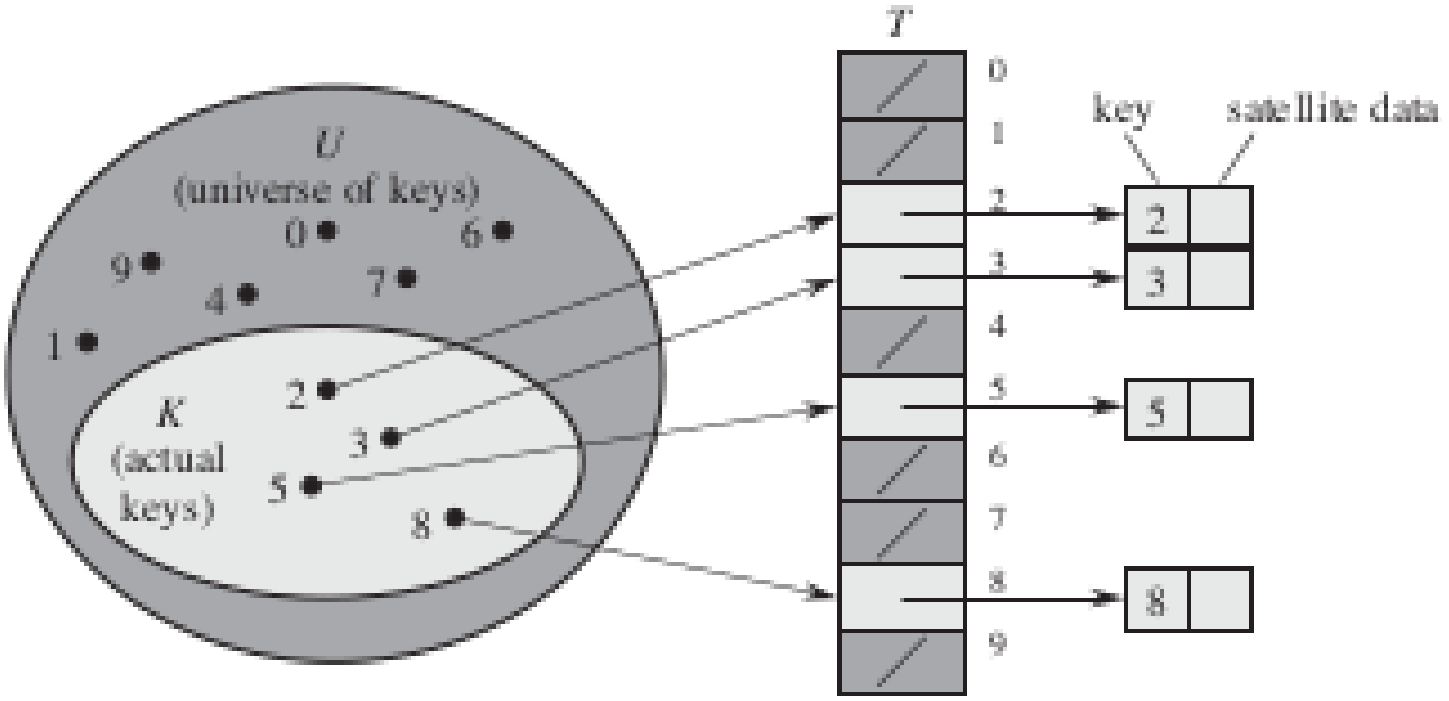
Hash Tabloları

- Anahtarı kullanılan dizideki indeks değeri olarak kullanmak yerine, indeks anahtar kullanılarak hash fonksiyonu ile hesaplanır.
  - Aynı indekse iki farklı anahtar eşlenirse çarpışma (İng: collision) oluşur.
  - Bu durumda zincerleme (İng: chaining) metodu kullanarak anahtarlar hesaplanan indekslere yerleştirilebilir.

Direk adresleme tablosu

- Anahtarları sahip olabileceği değer kümesi küçük olduğunda direk adresleme metodu kullanılabilir.
- Bir uygulamanın dinamik bir kümeden oluşan elemanlar kullandığını ve bu elemanların her birinin U = {0,1,2,...,m-1} evrensel kümesinden bir anahtara sahip olduğunu varsayalım.
  - Ayrıca iki farklı elemanın aynı anahtara sahip olamayacağını varsayıyoruz.
- Dinamik kümeyi temsil etmek için direk adresleme tablosunu veya bir dizi kullanıyoruz.
  - T[0..m-1], her bir pozisyon evrensel kümedeki bir anahtara karşılık geliyor.
  - k numaralı pozisyon kümedenin k anahtarına sahip elemanına işaret ediyor.

Direk adresleme tablosu



Direk adresleme tablosu

```
DIRECT-ADDRESS-SEARCH(T, k)
1   return T[k]

DIRECT-ADDRESS-INSERT(T, x)
1   T[x.key] = x

DIRECT-ADDRESS-DELETE(T, x)
1   T[x.key] = NIL
```

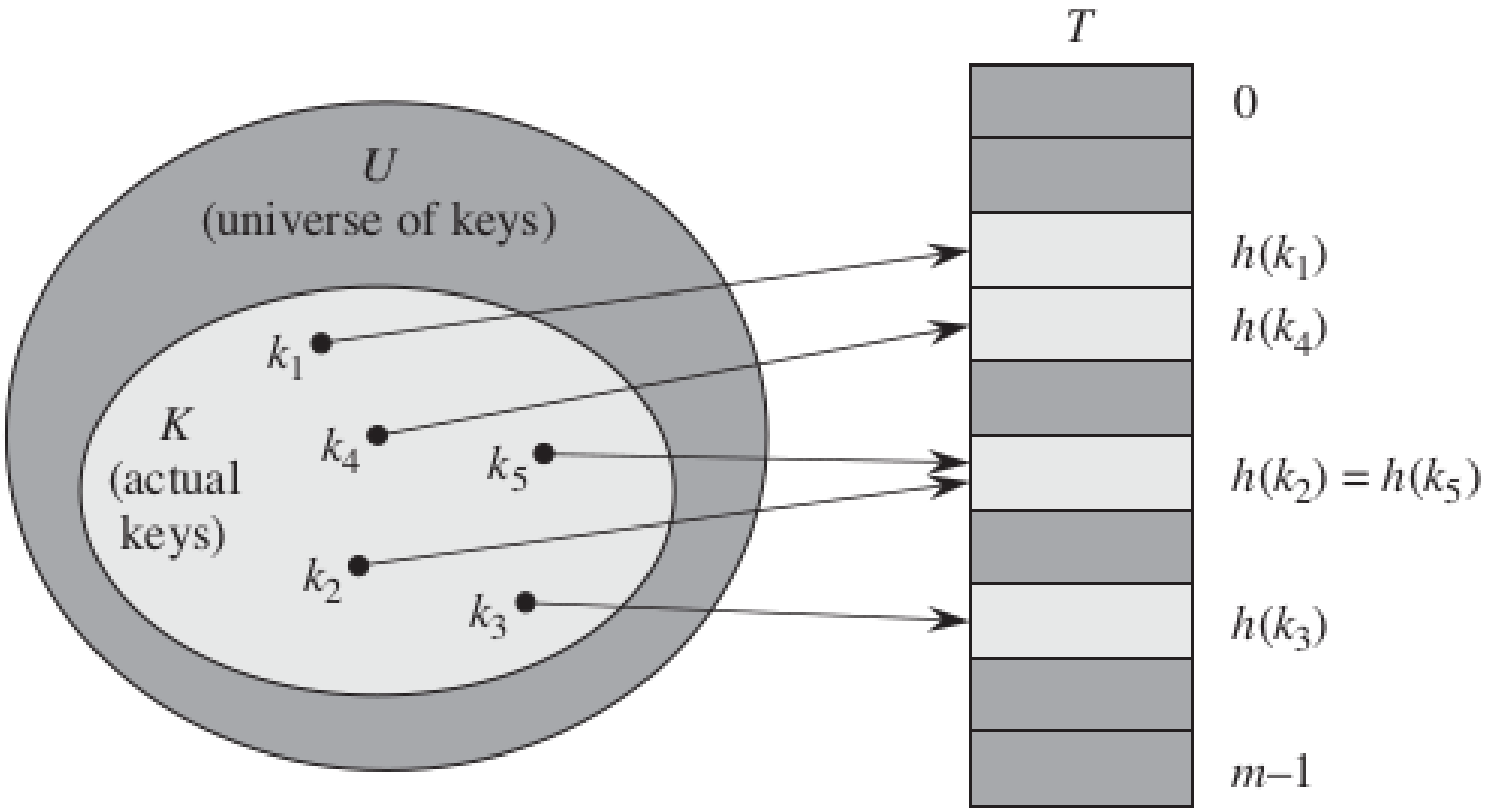
Hash Tabloları

- Direk adresleme konusundaki sorun şudur:
  - Eğer anahtarların alabileceği değer kümesi, |U|, çok büyük ise bu büyüklükte bir dizi fazla miktarda yer gerektirecektir.
  - Ayrıca saklanacak eleman sayısı,k, görece az olduğunda ayrılan hafızanın önemli kısmı boşa gidecektir.
- Direk adresleme metodu kullanıldığında k anahtarına sahip eleman k numaralı pozisyonda saklanır.
- Hash yönteminde ise k anahtarına sahip eleman h(k) numaralı pozisyonda saklanır.
  - Hash fonksiyonu h kullanılarak k değerinden elemanın pozisyonu hesaplanır.
  - h fonksiyonu evrensel kümedeki anahtarları tablodaki pozisyonlara T[0..m-1] eşler.
    - h: U → [0,1,…,m-1]

Hash Tabloları

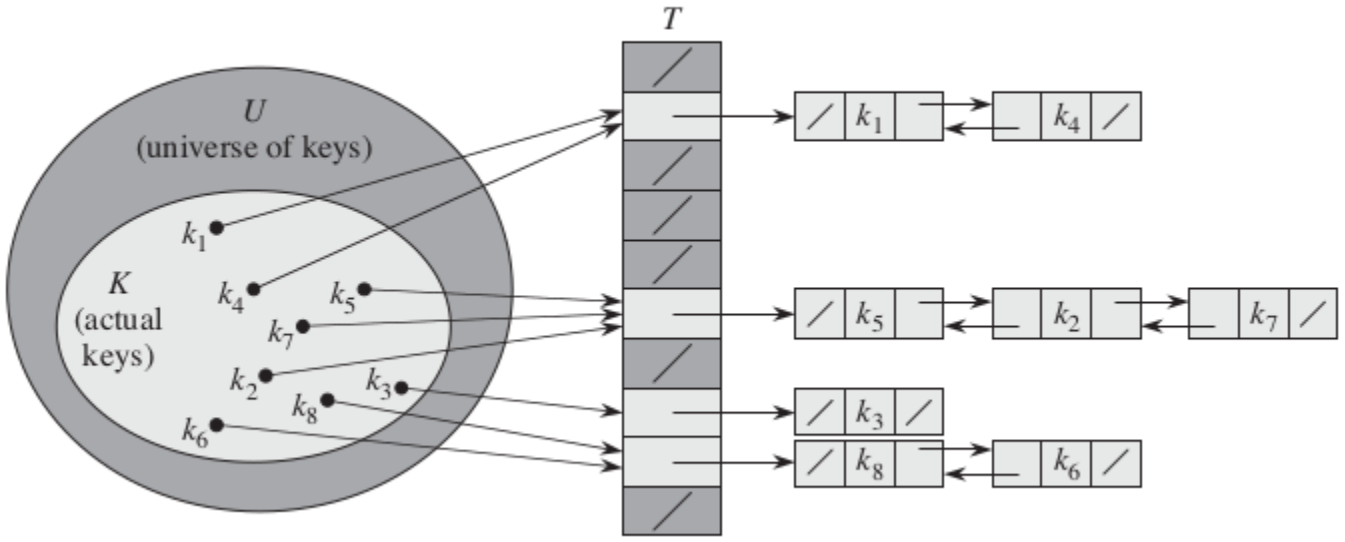
- k elemanın eşlendiği pozisyon h(k)'dır.
- h(k), k elemanının tablo değeridir.
- Hash fonksiyonu kullanarak, tablomuzun büyüklüğünü azaltabiliriz.
  - Tablonun büyüklüğü |U| yerine m olacaktır.
- Ancak hash fonksiyonu iki farklı elemanı aynı pozisyona eşleyebilir.
  - Bu duruma çarpışma (İng: collision) diyoruz.
- Çarpışma durumunu çözmek için bazı yöntemler öğreneceğiz.
- Öncelikle doğru oluşturulmuş bir hash fonksiyonu kullanarak çarpışma sayısını minimize edebiliriz.
- Ancak |U| > m olduğuna göre bu yöntemle çarpışmaları tamamen ortadan kaldırmak mümkün olmayacaktır.

Hash Tabloları



Hash Tabloları

- Zincirleme yöntemi ile çarpışma çözme
  - Zincirleme yöntemi ile aynı pozisyona eşlenen bütün elemanları bir bağlı liste kullanarak sıralıyoruz.



Hash Tabloları

- Zincirleme yöntemi ile çarpışma çözme için aşağıda belirtilen fonksiyon ve yapılar tanımlanmalıdır.
  - Öncelikle tabloya ekleyeceğimiz elemanlar için bir yapı tanımlamalıyız.

```
typedef struct node
{
    int data;
    struct node *next;
}node;
```

  - Tablomuzu her elemanı bir bağlı liste olan bir dinamik dizi olarak tanımlayabiliriz.

```
node **head;
```

Hash Tabloları

- Zincirleme yöntemi ile çarpışma çözme için aşağıda belirtilen fonksiyon ve yapılar tanımlanmalıdır.
  - Tablomuzu aşağıdaki şekilde oluşturabiliriz.

```
void SetTable()
{
    int i;
    head = (node**)malloc(sizeof(node*)*TABLE_SIZE);
    for(i = 0; i < TABLE_SIZE;i++)
    {
        head[i] = NULL;
    }
}
```

  - Yeni elemanlarımızı aşağıdaki şekilde oluşturabiliriz.

```
node* CreateNewNode(int key)
{
    node * newnode=(node *)malloc(sizeof(node));
    newnode->data=key;
    newnode->next = NULL;
    return newnode;
}
```

Hash Tabloları

- Zincirleme yöntemi ile çarpışma çözme için aşağıda belirtilen fonksiyon ve yapılar tanımlanmalıdır.
  - Tabloya ekleme fonksiyonu aşağıdaki şekilde oluşturabiliriz.

```
void HashInsert(int key)
{
    int pos;
    pos=HashCarpma(key);
    node * newnode=CreateNewNode(key);
    if(head[pos] == NULL)
    {
        head[pos] = newnode;
    }
    else
    {
        newnode->next = head[pos];
        head[pos] = newnode;
    }
}
```

  - Hash fonksiyonu hakkında birazdan konuşacağız.

Hash Tabloları

- Zincirleme yöntemi ile çarpışma çözme için aşağıda belirtilen fonksiyon ve yapılar tanımlanmalıdır.
  - Tablodan eleman silme fonksiyonu aşağıdaki şekilde oluşturabiliriz.

```
void HashDelete(int key)
{
    int pos;
    pos=HashCarpma(key);
    node* c=head[pos];
    if(c == NULL)
        return;
    else if(c->data == key)
    {
        if(c->next != NULL)
            head[pos] = c->next;
        else
            head[pos] = NULL;
        free(c);
    }
    else
    {
        while(c->next != NULL && c->next->data != key)
            c = c->next;
        if(c->next != NULL)
        {
            node *t = c->next;
            c->next = c->next->next;
            free(t);
        }
    }
}
```

Hash Tabloları

- Zincirleme yöntemi ile çarpışma çözme için aşağıda belirtilen fonksiyon ve yapılar tanımlanmalıdır.
  - Tabloda eleman arama fonksiyonu aşağıdaki şekilde oluşturabiliriz.

```
int HashSearch(int key)
{
    int pos;
    node *c;
    pos = HashCarpma(key);
    if(head[pos] == NULL)
        return -1;
    else
    {
        for(c=head[pos];c!=NULL;c=c->next)
        {
            if(c->data == key)
            {
                return pos;
            }
        }
        return -1;

    }
}
```

Hash Tabloları

- Zincirleme metodunun analizi
  - Öncelikle, hash fonksiyonunun uygulanması sonucunda her elemanın eşit olasılıkla m pozisyondan birine eşlendiğini varsayalım.
    - Bu özelliğe basit düzgün tablo (basic uniform hashing) özelliği diyoruz.
  - n anahtarımız var ve m tane pozisyon var ise yük faktörü $\alpha = n/m$ şeklinde hesaplayabilir.
    - Her pozisyona düşen anahtar sayısı
  - Başarısız bir arama yapmak ne kadar süre alacaktır?
    - $O(1 + \alpha)$
  - Başarılı bir arama yapmak ne kadar süre alacaktır?
    - $O(1 + \alpha/2) = O(1 + \alpha)$

Hash Tabloları

- Zincirleme yönteminin analizi
  - Öyleyse arama işleminin harcadığı zaman O(1 + α)‘dır.
  - Eğer saklanacak olan anahtar sayısı çizeldeki pozisyon sayısı ile orantılı olursa
    - α = O(1)
  - Bu durumda yük faktörünü sabit tutabilirsek arama operasyonunu sabit beklenen sürede yapabiliriz.

Hash Tabloları

- Hash fonksiyonunun doğru seçilmesi hash tablolarının hızlı çalışabilmesi için önemlidir.
  - Çok kötü bir hash fonksiyonu ne yapacaktır?
    - Her anahtarı aynı pozisyona eşler.
  - Bu durumda arama operasyonu ne kadar süre alacaktır.
    - O(n)
  - Doğru bir hash fonksiyonunun sahip olması gereken özellikler
    - Anahtarları pozisyonlara düzgün olarak dağıtmalıdır.
    - Verinin dağılımından bağımsız olarak eşleme yapmalıdır.
  - Hash fonksiyonlarının önemli bir kısmı anahtarlar doğal sayı olmasını bekler, yani anahtarlar N = {1,2,...} kümesinin elemanları olmalıdır.
  - Anahtarlar doğal sayı değilse bir şekilde doğal sayıya çevirmenin bir yolunu bulmak gerekir.

Hash Tabloları

- Hash fonksiyonları
  - Bölme metodu
    - Bu metot kullanıldığında k anahtarını m pozisyondan birine k'nın m ile bölünmesinden kalan şeklinde eşliyoruz.
      - h(k) = k mod m
    - Örnek: hash tablosunun büyüklüğü m = 12 ise k = 100 ise, h(k) = 4.
    - Bölme metodu kullanıldığında m için bazı değerler seçilmemelidir.
      - Örneğin m, 2'nin kuvveti olmamalıdır. Yani m = 2^p olmamalıdır.
        - Bu durumda hash değeri sadece en düşük dereceli bit değerlerine bağlı olarak hesaplanacaktır.
    - Genellikle m değeri için 2'nin kuvvetlerine yakın olmayan bir asal sayı seçilir.

Hash Tabloları

- Hash fonksiyonları
  - Çarpma metodu
    - $h(k) = \lfloor m \, (kA - \lfloor kA \rfloor) \rfloor$
      - Çarpma işlemin $0 < A < 1$ değeri kullanılır.
    - Bu yöntemin güzel tarafı m değerinin önemli olmamasıdır.
      - Bu durumda 2‘nin kuvveti olan bir m değeri seçerek işlem hızlı bir şekilde yapılabilir.
    - A değeri olarak herhangi bir değer seçilebilir.
      - Ancak 0 veya 1‘e yakın bir değer seçilmemelidir.
      - Knuth tarafında önerilen değer $A \approx (\sqrt{5} - 1) / 2$.

Hash Tabloları

- Evrensel hashleme
  - Sabit bir hash fonksiyonu kullanılması durumunda, her elemanın aynı pozsiyona eşleneceği bir anahtarlar kümesi bulmak mümküdür.
  - Böyle durumları engellemenin daha önce kullandığımız bir yöntemi vardır.
    - Algoritmayı rastgele hale getirme.
  - Hash fonksiyonunu bir hash fonksiyonu kümesinden rastgele seçeriz.
    - Bu yönteme evrensel hashleme denir.
  - Böylece en kötü çalışma süresine sebep olacak bir girdi bulunması mümkün olmaz.

Hash Tabloları

- Evrensel hashleme
  - H hash fonksiyonlarımızın (sınırlı) kümesi olsun
    - Bu fonksiyonlar U evrensel kümesinden gelen anahtarları {0,1,…,m – 1} kümesindeki pozisyonlara eşlesin.
    - H aşağıdaki koşulda evrenseldir
      - Her x,y ϵ U farklı anahtarları için h ϵ H fonksiyonlarından h(x) = h(y) durumuna sebep olan hash fonksiyonlarının sayısı |H| / m dir.
      - Yani H kümesinden rastgele seçilmiş bir hash fonksiyonunun x ve y farklı anahtarları için çarpışmaya sebep olma ihtimali 1/m ‘dir.
      - Buna göre
        - H hümesinden bir h fonksiyonu rastgele seçilir.
        - n tane anahtarın her biri m pozisyondan birine yerleştirilir, n ≤ m.
        - Bu durumda belli bir anahtar için beklenen çarpışma sayısı 1‘den küçüktür.

Hash Tabloları

- Açık adresleme
  - Çarpışma olması durumunda zincirleme metodunun yerine açık adresleme metodu kullanılabilir.
  - Bu metot kullanıldığında saklanacak anahtarlar direk olarak hash tablosunin içerisindedir.
    - İşaretçi kullanarak zincir oluşturmaya gerek kalmaz.
  - Bu sebeple tablonun tamamen dolması mümkündür.
  - Tabloya yeni bir anahtar eklemek istediğimizde boş bir pozisyon bulana kadar tablonun üzerinde gezinir.
    - Bu işleme deneme (İng: probing) denir.
  - Arama yapılacağında ekleme işlemindeki aynı deneme işlemi yapılarak belirtilen pozisyonlar kontrol edilir.
    - Eğer aranan anahtar mevcutsa, anahtar dönülür.
    - Tabloda boş bir pozisyonla karşılaşılırsa, anahtar tabloda değildir.

Hash Tabloları

- Açık adresleme
  - Doğrusal (lineer) deneme
    - Verilen sıradan h‘ U: → {0,1,...m – 1} kullanılarak doğrusal deneme ile yapılandırılan h fonksiyonu şu şekildedir:
      - $h(k,i) = (h'(k) + i) \bmod m$
    - Bu yöntem oldukça basit şekilde kodlanabilir.
    - Ancak anahtarların belli bölümlerde kümelenmesine ve bu sebeple arama süresinin giderek artmasına sebep olabilir.
  - Quadratic deneme
    - Verilen sıradan h‘ U: → {0,1,...m – 1} kullanılarak quadratic deneme ile yapılandırılan h fonksiyonu şu şekildedir:
      - $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  - İkili hashleme
    - h_1 ve h_2 hash fonksiyonları kullanılarak yapılır:
      - $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

Hash Tabloları

- Açık adresleme
  - Bu metot, sil operasyonunun yapılmadığı durumlarda oldukça iyi çalışır.
    - Örneğin yazım kontrolü için kullanılabilir.
  - Sil operasyonu yapıldığında ise yerleştirilen anahtarı sadece silmek yeterli olmayacaktır.
    - Bu elemanın sonrasına anahtarlar yerleştirilmiş ise arama işlemi devam etmelidir.