

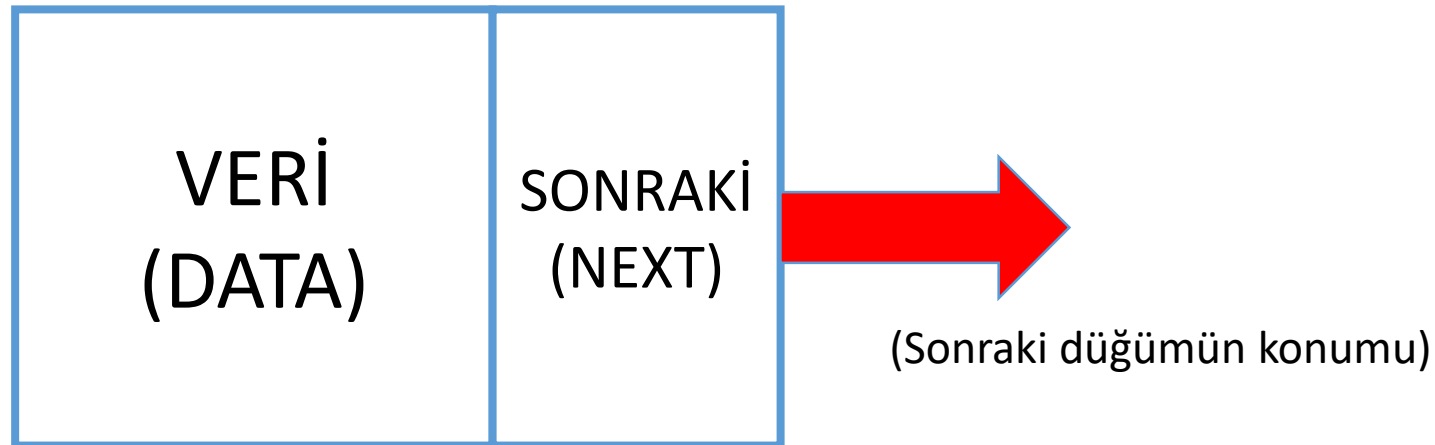
Linked List

Bağlı (veya bağlantılı) liste

- Tek yönlü (Singly Linked List)
- Çift yönlü (DoublyLinked List)

Düğüm (Node) tek yönlü

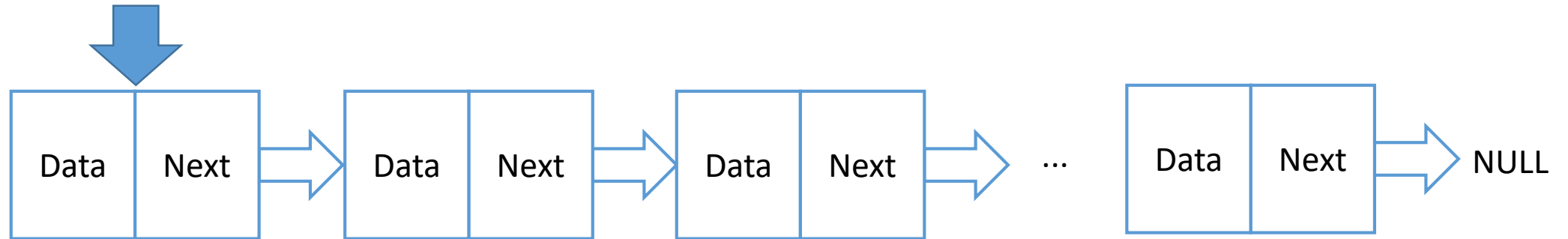
- Bağlı listelerin temel elemanı düğümlerdir. Listeye bir eleman eklendiği zaman eleman, düğüm üzerinde saklanır.
- Tek yönlü bağlantılı bir listenin düğümleri aşağıdaki gibi sonraki düğüme bir bağlantı içerir.



Tek yönlü bağlı liste

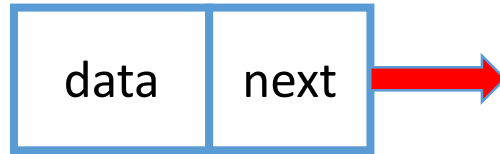
Tek yönlü bağlı listede liste elemanları sonraki elemana bağlantılıdır.

İlk (Baş) eleman, (head)



C++ düğüm yapısı

- Bağlı listede bağlantılar düğüm içerisinde pointer kullanılarak yapılır.
- Düğümleri oluşturmak için class veya struct tercih edilebilir.
- Aşağıda verilen düğüm örneklerinde liste veri tipi integer olarak tanımlanmıştır. Daha sonra template tanımlaması ile nesne veya diğer temel veri tiplerine genişletilecektir.



```
class Node{
public:
    int data;
    Node *next;
    Node(int data, Node *next=NULL) {
        this->data=data;
        this->next=next;
    }
};
```

```
struct Node{
    int data;
    Node *next;
    Node(int data, Node *next=NULL) {
        this->data=data;
        this->next=next;
    }
};
```

Bağlı liste

```
//Düğüm1er
```

```
Node *dugum0=new Node (5) ;
```

```
Node *dugum1=new Node (3) ;
```

```
Node *dugum2=new Node (7) ;
```

```
Node *dugum3=new Node (1) ;
```

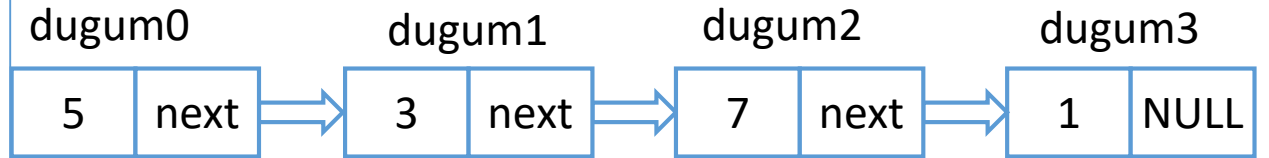
```
//Bağlantılar
```

```
dugum0->next=dugum1 ;
```

```
dugum1->next=dugum2 ;
```

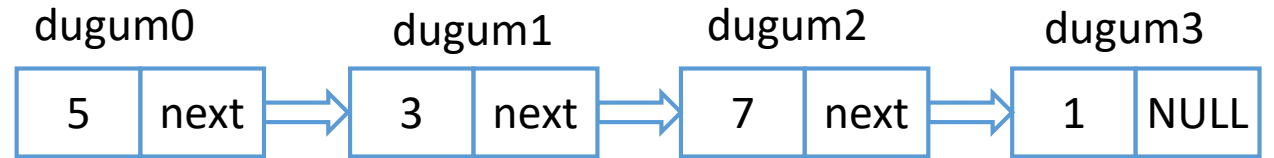
```
dugum2->next=dugum3 ;
```

```
dugum3->next=NULL; // liste sonu
```



- Yandaki örnekte 4 düğüm oluşturulmuş ve bağlantıları yapılarak bağlantılı bir listeye dönüştürülmüştür.
- Son elemandan sonrası NULL yapılarak listenin sonlandığı belirtilmiştir. Burada Node sınıfı içinde next değişkeni zaten default olarak NULL tanımlandığı için bu satırı yazmayabiliriz.

Bağlı liste



```
Node *ilk; //head
```

```
ilk=new Node(5);
```

```
ilk->next=new Node(3);
```

```
ilk->next->next=new Node(7);
```

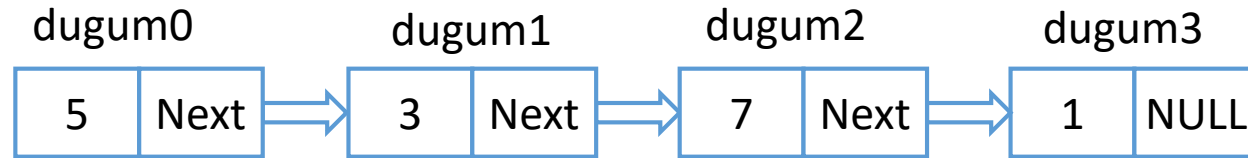
```
ilk->next->next->next=new Node(1);
```

- Her bir düğüm için ayrı bir pointer tanımlamak yerine sadece listenin ilk elemanı için bir tanımlama yapılabilir.

Bağlı liste

```
Node *ilk,*temp;

//ilk eleman
ilk=new Node(5); //head
//sonrakiler
temp=ilk;
temp->next=new Node(3);
temp=temp->next;
temp->next=new Node(7);
temp=temp->next;
temp->next=new Node(1);
```



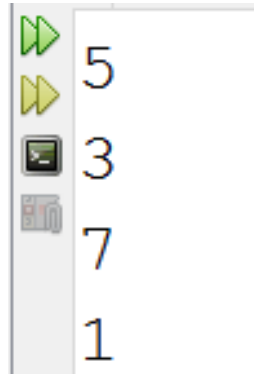
- ilk düğüme ek olarak bir geçici bir pointer tanımlanırsa eleman ekleme daha pratik hale getirilmiş olur.
- Burada **temp=temp->next** ile bir sonraki düğüme gidilir.

Bağlı liste üzerinde dolaşmak

```
int main(int argc, char** argv) {
    Node *ilk,*temp;
    //ilk eleman
    ilk=new Node(5); //head
    //sonrakiler
    temp=ilk;
    temp->next=new Node(3);
    temp=temp->next;
    temp->next=new Node(7);
    temp=temp->next;
    temp->next=new Node(1);

    //elemanları yazdır
    temp=ilk;
    cout<<temp->data<<endl;
    temp=temp->next;
    cout<<temp->data<<endl;
    temp=temp->next;
    cout<<temp->data<<endl;
    temp=temp->next;
    cout<<temp->data<<endl;
}
```

- Oluşturduğumuz listenin elemanlarını yazdırmak için temp ile tanımlanmış pointer tekrar ilk elemana eşitlenir.
- Bir eleman yazdırıldıktan sonra yine **temp=temp->next** ile sonraki düğüme işaret edilir.



Bağlı liste üzerinde dolaşmak

```
//elemanları yazdır
temp=ilk;
while (temp != NULL) {
    cout << temp->data << endl;
    temp = temp->next;
}
```

- Yazdırma işlemi yandaki gibi döngüsel olarak ifade edilebilir.
- İşaretçi ilk elemana konumlandırıldıktan sonra NULL olmadığı sürece elemanı yazdırmakta ve bir sonraki elemana geçmektedir.
- Eğer bir düğümden sonraki eleman yoksa next=NULL olacağı için döngü sonlanır.
- Bundan dolayı listenin son elemanının sıradaki elemanı (next) gösteren pointer mutlaka NULL değildir.
- Bundan dolayı tanımladığımız Node sınıfı içerisinde her bir düğümün next değişkenine bağlangıç değeri olarak NULL atanmaktadır.

Bağlı liste üzerinde dolaşmak

```
void yazdir(Node *temp) {
    while (temp != NULL) {
        cout << temp->data << endl;
        temp = temp->next;
    }
}

int main(int argc, char** argv) {
    Node *ilk,*temp;
    //ilk eleman
    ilk=new Node(5); //head
    //sonrakiler
    temp=ilk;
    temp->next=new Node(3);
    temp=temp->next;
    temp->next=new Node(7);
    temp=temp->next;
    temp->next=new Node(1);
    //elemanları yazdır
    yazdir(ilk);
}
```

- Yazdırma işlemi yanda görüldüğü gibi bir fonksiyon olarak tanımlanabilir.
- temp değişkeni fonksiyon çağırılırken ilk eleman olarak seçilirse tüm liste yazdırılır.
- Elemanlar üzerinde dolaşım aşağıdaki gibi rekürsif olarak da yapılabilir.

```
void yazdir(Node *temp) {
    if (temp != NULL) {
        cout << temp->data << endl;
        yazdir(temp->next);
    }
}
```

Bağlı liste üzerinde dolaşmak

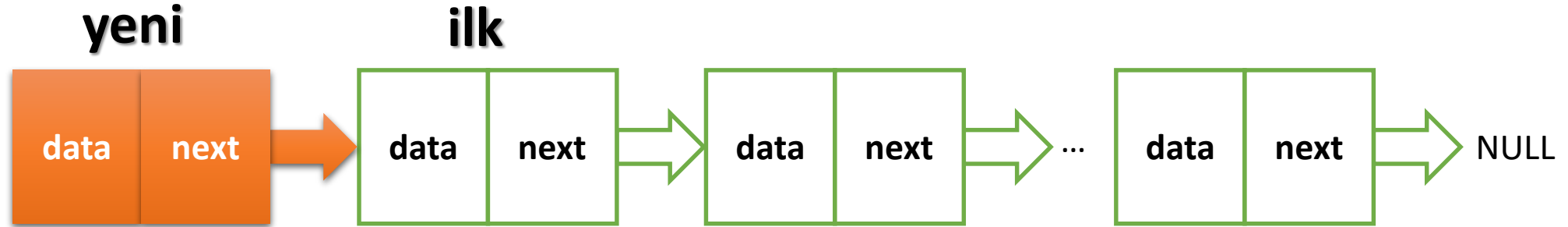
```
int boyut(Node *temp){ //size, length
    int say=0;
    while (temp != NULL) {
        say++;
        temp = temp->next;
    }
    return say;
}

int main(int argc, char** argv) {
    Node *ilk,*temp;
    //ilk eleman
    ilk=new Node(5); //head
    //sonrakiler
    temp=ilk;
    temp->next=new Node(3);
    temp=temp->next;
    temp->next=new Node(7);
    temp=temp->next;
    temp->next=new Node(1);

    cout<<"Eleman sayısı:"
    cout<<boyut(ilk)<<endl;
}
```

- Listenin eleman sayısını belirlemek için de yanda görüldüğü gibi elemanları dolaşıp her eleman için sayacı 1 arttırılmıştır.
- Belirli bir konumdaki elemanı getirmek, belirtilen konuma eleman eklemek/silmek, listenin sonuna gitmek, vb. işlemler için benzer yaklaşımlar kullanılmaktadır.

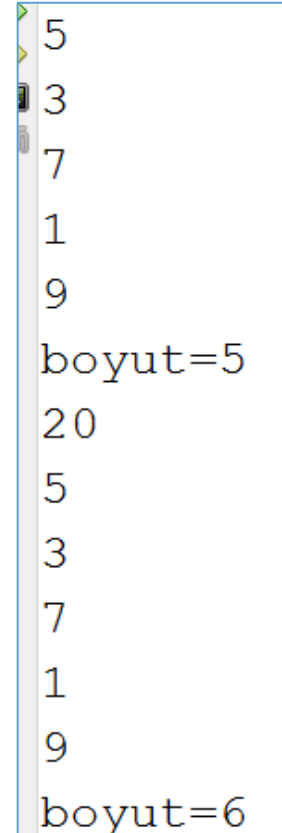
Listenin başına eleman eklemek



- Yeni bir eleman eklenirken liste boşsa yeni eleman ilk eleman olarak atanır.
- Listede bir veya daha fazla eleman varsa listenin başına eleman eklemek için,
yeni->next=ilk;
ile yeni eleman ilk elemanın önüne konumlandırılır.

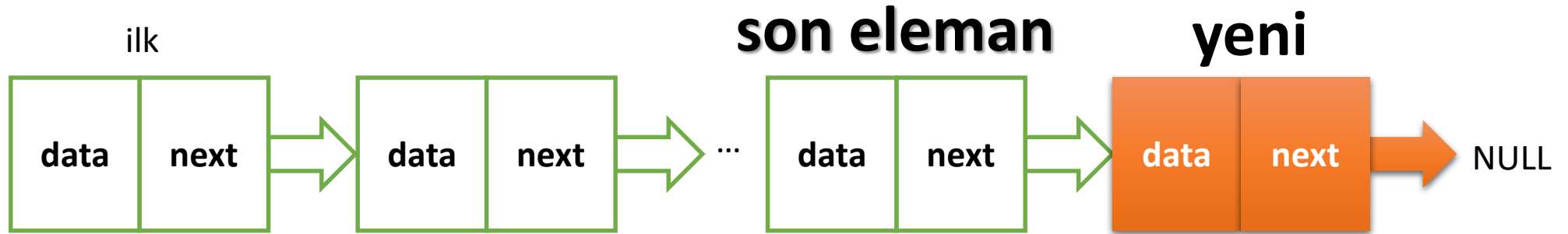
Listenin başına eleman eklemek

```
temp->next = new Node(9);  
//yazdir  
yazdir(ilk);  
cout << "boyut=" << boyut(ilk) << endl;  
  
// listenin başına bir eleman ekle  
Node *yeni = new Node(20);  
if (ilk == NULL)  
    ilk = yeni;  
else {  
    //ilk eleman yeni elemandan sonra  
    yeni->next = ilk;  
    // yeni eleman listenin ilk elemanı  
    ilk = yeni;  
}  
yazdir(ilk);  
cout << "boyut=" << boyut(ilk) << endl;
```



5
3
7
1
9
boyut=5
20
5
3
7
1
9
boyut=6

Listenin sonuna eleman eklemek



- Önceki gibi, liste boşsa yeni eleman ilk eleman olarak atanır.
- Listede bir veya daha fazla eleman varsa listenin son elemanı tespit edilir.
- Yeni düğüm oluşturularak son elemana bağlanır.

Listenin sonuna eleman eklemek

```
//listenin sonuna bir eleman ekleyelim
yeni=new Node(30);

//listenin boş olması durumunda
if (ilk==NULL)
    ilk=yeni;
else{
    //ilk elemandan başla
    temp=ilk;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next=yeni;
}

yazdir(ilk);
cout<<"boyut="<<boyut(ilk)<<endl;
```

20

5

3

7

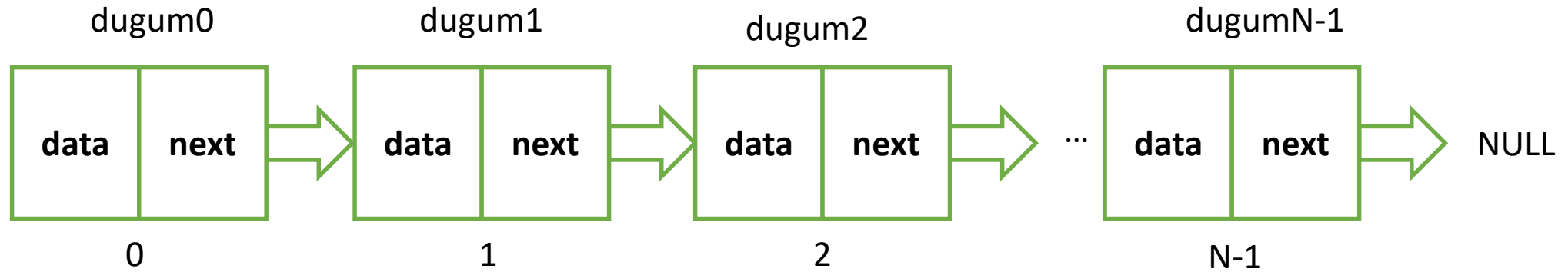
1

9

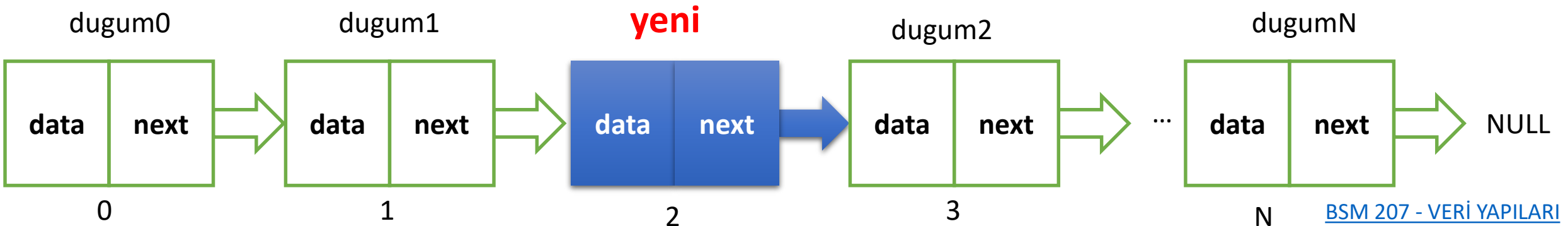
30

boyut=7

Belirtilen konuma eleman eklemek



- `Node *yeni=new Node(veri);`
- `yeni->next=dugum1->next;`
- `dugum1->next=yeni;`



Belirtilen konuma eleman eklemek

```
//yeni düğüm
Node<Nesne> *yeni = new Node<Nesne>(data);

// önceki konumdaki düğümü getir
int sayac = 0;
Node<Nesne> *temp = head;
while (temp->next != NULL) {
    if ((konum - 1) == sayac) break;
    temp = temp->next;
    sayac++;
}
//önceki düğümden sonra yeni düğümü ekle
yeni->next = temp->next;
temp->next = yeni;
```

- Yeni düğümü eklemek için önce belirtilen konumdan bir önceki düğümü tespit ederiz.
- While döngüsü konum-1 üzerinde sonlanır
- temp işaretçisi konum-1 üzerinde kalır.
- Bağlantı gerçekleştirilir.
- Ekleyeceğimiz konum listenin başı ise, daha önce açıkladığımız başa ekleme prosedürü işletilir.

Belirtilen konuma eleman eklemek

```
// önceki konumdaki düğümü getir
Node<Nesne> *temp = oncekiDugum(konum);

//önceki düğümden sonra yeni düğümü ekle
yeni->next = temp->next;
temp->next = yeni;
```

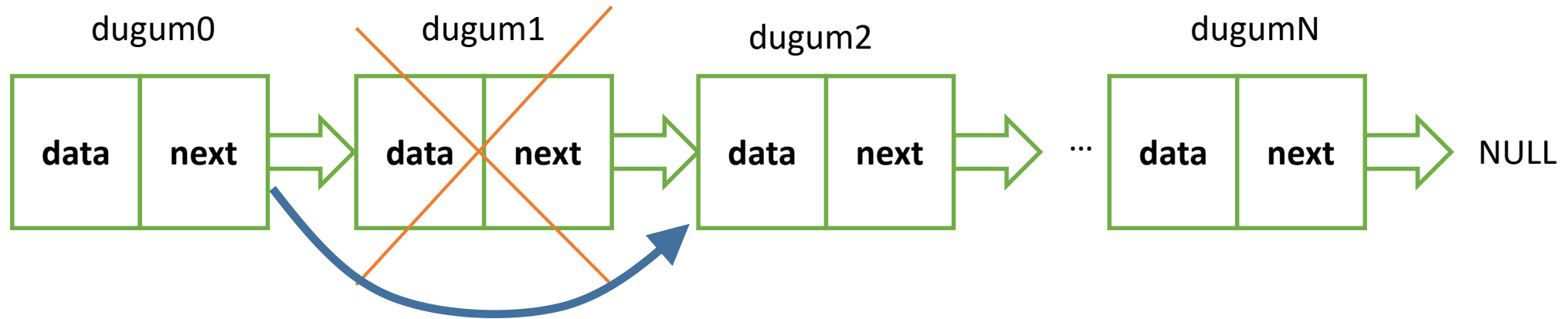
```
inline Node<Nesne>* oncekiDugum(int konum) {
    int sayac = 0;
    Node<Nesne> *temp = head;
    while (temp->next != NULL) {
        if ((konum - 1) == sayac) break;
        temp = temp->next;
        sayac++;
    }
    return temp;
}
```

- Önceki konumdaki elemanı getirmeyi bir fonksiyon olarak tanımlarsak belirtilen konuma düğüm ekleme yandaki gibi basit bir şekilde ifade edilebilir.
- oncekiDugum fonksiyonunun başına inline yazarsak derleme anında kodlar fonksiyonun çağrıldığı yere yazılır programın çalışması sırasında fonksiyon çağrısı yapılmaz ve çalışma zamanı kısalır.

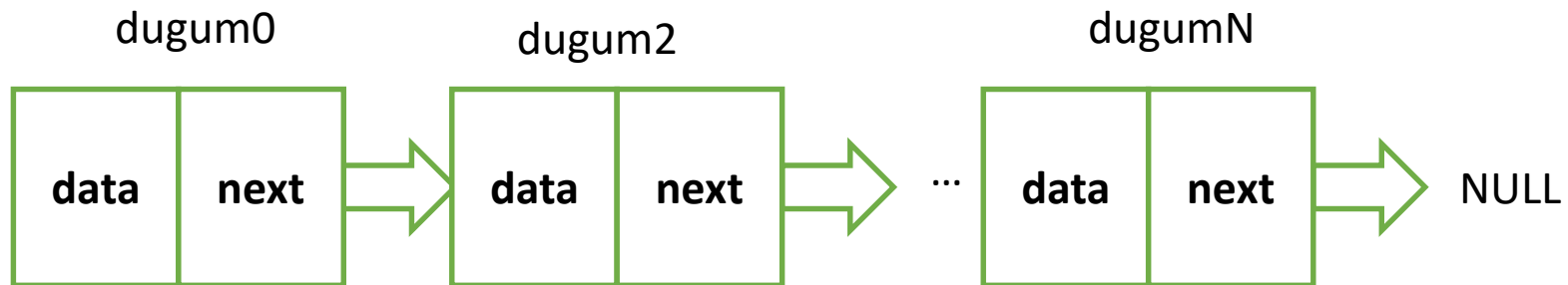
Belirtilen konuma eleman eklemek

```
void insert(int konum, const Nesne& data) throw (Hata) {  
    //konum geçerli mi?  
    if (konum < 0 | konum > length()) throw GecersizKonumHatasi();  
    //yeni düğüm  
    Node<Nesne> *yeni = new Node<Nesne>(data);  
    //yeni düğüm oluştur  
    if (konum == 0) { //liste başına ekle  
        push_front(data);  
    } else {  
        // önceki konumdaki düğümü getir  
        Node<Nesne> *temp = oncekiDugum(konum);  
  
        //önceki düğümden sonra yeni düğümü ekle  
        yeni->next = temp->next;  
        temp->next = yeni;  
    }  
}
```

Belirtilen konumdan eleman silmek



- `dugum0->next=dugum1->next;`
- `delete dugum1;`



Belirtilen konumdan eleman silmek

```
void remove(int konum) throw (GecersizKonumHatasi, BosListeHatasi) {  
    //listede eleman var mı?  
    if (head == NULL) throw BosListeHatasi();  
    //konum geçerli mi?  
    if (konum < 0 | konum > (length() - 1))  
        throw GecersizKonumHatasi();
```

```
    Node<Nesne> *temp= head;
```

```
    if (konum == 0) {  
        head = head->next;  
        delete temp;
```

```
    } else {
```

- Belirtilen konumdaki elemanı silme den önce listenin boş olup olmadığı ve konumun geçerliliği kontrol edilebilir. Burada length() fonksiyonu listenin eleman sayısını döndürüyor. Burada eleman sayısını tutan bir değişken de kullanılabilir.
- Sildiğimiz konum ilk eleman ise ilk eleman head=head->next ile bir sonrakini gösterecek şekilde kaydırılır ve delete ile çöp temizlenir.

Belirtilen konumdan eleman silmek

```
} else {  
    // önceki konumdaki düğümü getir  
    int sayac = 0;  
    while (temp->next != NULL) {  
        if (sayac == (konum - 1)) break;  
        temp = temp->next;  
        sayac++;  
    }  
    //belirtilen konumdaki düğümün referansını al  
    Node<Nesne> *eskidugum = temp->next;  
    // (konum-1) ->next=(konum+1)  
    temp->next = eskidugum->next;  
    //eski düğümü sil  
    delete eskidugum;  
}
```

- Silinecek düğüm ilk eleman değilse, silinecek düğümünden bir önceki düğüm konuma eleman eklemede olduğu gibi tespit edilir. Yine burada öncekiDugum fonksiyonunu kullanabiliriz.
- Önceki düğüm silinecek düğümünden sonrakini gösterecek şekilde bağlantılar yapılır ve eski düğüm silinir.

Belirtilen konumdaki elemanı okumak

```
//const Nesne& at(int konum) throw (Hata) {  
Nesne& at(int konum) throw (Hata) {  
    if (konum < 0 | konum > (length() - 1))  
        throw GecersizKonumHatasi();  
    //konumu bul  
    Node<Nesne> *temp;  
    temp = head;  
    int sayac = 0;  
    while (temp != NULL) {  
        if (sayac == konum) {  
            return temp->data;  
        }  
        temp = temp->next;  
        sayac++;  
    }  
}
```

- Eğer belirtilen konum geçerli ise daha önceki uygulamalara benzer şekilde, sayaç konuma eşit olana kadar düğümler üzerinde ilerlenir.
- Aranılan konumdaki düğümün verisi döndürülür.
- Eğer fonksiyon tanımlamasında geri döndürülen veritipinden önce **const** yazılırsa verinin değiştirilmesi engellenmiş olur.

Belirtilen konumdaki elemanı okumak

```
//bir konumdan eleman oku
cout<<"\nliste1.at(2)="<<liste1.at(2)<<endl;

//eleman değiştir
liste1.at(2) = "degistir";

//bir konumdan eleman oku
cout<<"liste1.at(2)="<<liste1.at(2)<<endl;
```

```
liste1.at(2)=deneme2
liste1.at(2)=degistir
```

- Tanımladığımız fonksiyonda const ifadesi yoksa yandaki gibi çağırdığımız elemana atama yapabiliriz.
- Eğer bu şekilde bir değişim istemiyorsak fonksiyonun başına const yazarak bunu engelleyebiliriz.

iterator

```
template <typename T>
class ListIterator { //liste üzerinde gezmek için
private:
    Node<T> *simdiki;
public:
    ListIterator(Node<T> *simdiki=NULL) {
        this->simdiki = simdiki; //curr
    }
    void ilerle() {
        if (simdiki == NULL) throw Hata("Liste Sonu");
        simdiki = simdiki->next;
    }
    T& getir() const { //const T&getir()const{
        return simdiki->data;
    }
    bool sonaGeldiMi() const {
        return simdiki == NULL;
    }
    template <typename U> friend class LinkedList;
};
```

- ListIterator, liste üzerinde dolaşmak için kullanılır.
- Yapılandırıcı ile konum alınır ve ilerle() ile bu konumdan itibaren liste üzerinde ileri yönlü dolaşabiliriz.
- Çift yönlü bağlı listede bir de gerile() gibi ek bir fonksiyon bulunur.
- Bulduğumuz konumdaki veriye getir() ile ulaşabiliriz.

iterator

```
ListIterator<Nesne> ilk() throw (BosListeHatasi) {  
    if (head == NULL) throw BosListeHatasi();  
    return ListIterator<Nesne>(head);  
}
```

- Yukarıdaki fonksiyon Listenin ilk elemanını kullanarak ListIterator tipinde bir nesne döndürür.
- benzer şekilde ListIterator desteği olan fonksiyonlar yazabiliriz.

iterator

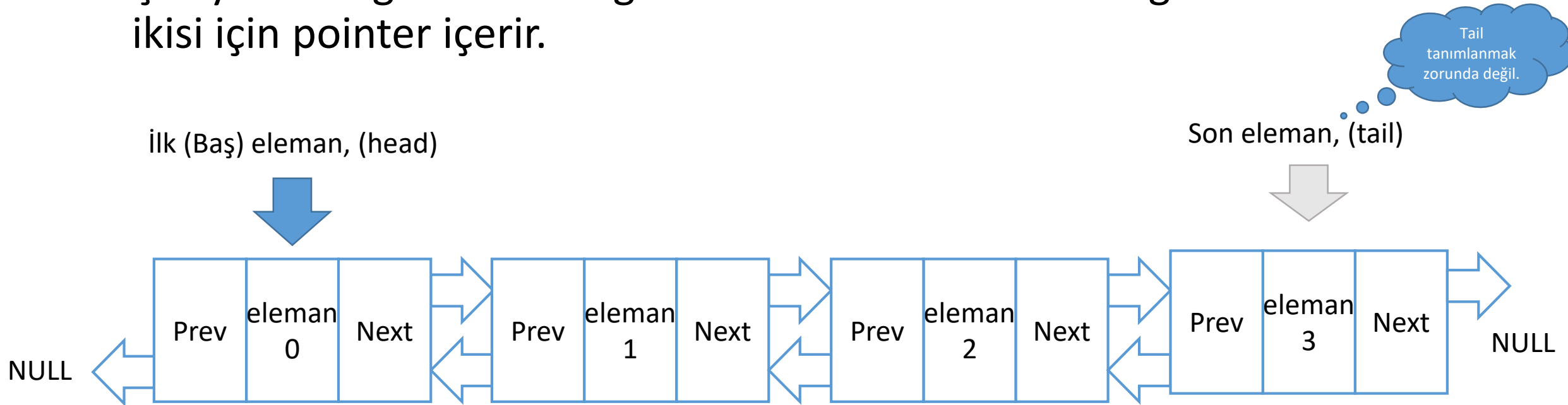
```
//liste
LinkedList<string> liste1;
liste1.push_back("pazartesi");
liste1.push_back("salı");
liste1.push_back("çarşamba");
liste1.push_back("perşembe");
liste1.push_back("cuma");
liste1.insert(0, "1234");

cout<<"\n\nListIterator ile liste elemanlarını yazdır"<<endl;
//iteratör ile elemanları getir
for (ListIterator<string> itr = liste1.ilke();
     !itr.sonaGeldiMi(); itr.ilerle()) {
    cout << itr.getir() << endl;
}
```

- Yandaki örnekte oluşturulan liste üzerinde ListIterator ile dolaşarak elemanlar yazdırılmıştır.
- Burada for döngüsü içerisindeki ListIterator değişkeni başlangıç değerini ilk() fonksiyonundan almıştır.
- For döngüsünün her iterasyonunda ilerle ile sonraki eleman getirilmiştir.
- Bu işlemi **at(konum)** gibi belirtilen konumdaki elemanı getiren bir fonksiyon ile de yapabiliriz. Ancak her defasında elemanı getirmek için liste başından başlar.

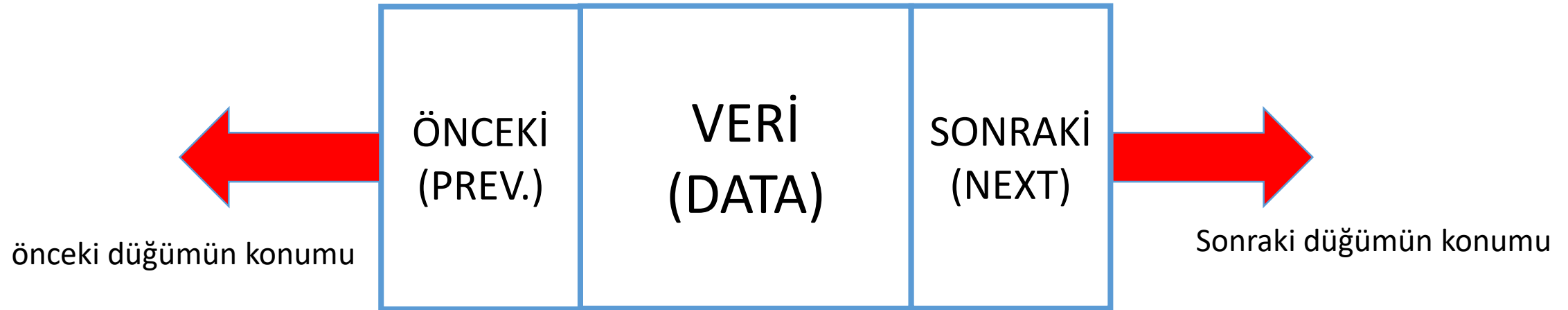
Çift yönlü bağlı liste – Doubly Linked List

- Çift yönlü bağlı listede düğümler önceki ve sonraki düğümlerin her ikisi için pointer içerir.



Çift yönlü bağlı liste -Düğüm (Node) çift yönlü

- Çift yönlü bağlı listeyi oluşturan düğümlerde, önceki ve sonraki düğümlerin ikisine de bağlantı bulunur.



Çift yönlü bağlı liste -Düğüm (Node) çift yönlü

```
template <typename Nesne>
class Node {
public:
    Nesne data; //veri
    Node *next, *prev; //sonraki, önceki

    Node(const Nesne& data, Node<Nesne> * next = NULL,
          Node<Nesne> * prev = NULL) {
        this->data = data;
        this->next = next; //sonraki
        this->prev = prev; //önceki
    }
    //template <typename T> friend class LinkedList;
};
```

Çift yönlü bağlı liste – Sınıf yapısı

```
template <typename Nesne>
class LinkedList {
private:
    Node<Nesne> *head, *tail;
    int elemanSayisi;
public:

    LinkedList() {
        head = NULL; //ilk
        tail = NULL; //son
        elemanSayisi=0;
    }

    void push_front(const Nesne &nesne) {
        Node<Nesne> *yeniNesne = new Node<Nesne>(nesne);
        yeniNesne->next = head;
        head = yeniNesne;
        elemanSayisi++;
    }

    void push_back(const Nesne &nesne) {
        Node<Nesne> *yeniNesne = new Node<Nesne>(nesne);
        yeniNesne->next = NULL;
        if (tail)
            tail->next = yeniNesne;
        else
            head = yeniNesne;
        tail = yeniNesne;
        elemanSayisi++;
    }

    void insert(int konum, const Nesne &nesne) {
        if (konum == 0)
            push_front(nesne);
        else if (konum == elemanSayisi)
            push_back(nesne);
        else {
            Node<Nesne> *temp = head;
            for (int i = 0; i < konum; i++)
                temp = temp->next;
            yeniNesne->next = temp->next;
            temp->next = yeniNesne;
            elemanSayisi++;
        }
    }

    void deleteNode(int konum) {
        if (konum == 0)
            delete head;
            head = head->next;
        else if (konum == elemanSayisi - 1)
            delete tail;
            tail = tail->next;
        else {
            Node<Nesne> *temp = head;
            for (int i = 0; i < konum; i++)
                temp = temp->next;
            delete temp;
            temp = temp->next;
            elemanSayisi--;
        }
    }

    void printList() {
        Node<Nesne> *temp = head;
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

- Çift yönlü bağlı listenin tanımlandığı sınıf içerisinde, tek yönlüde olduğu gibi, listenin ilk elemanını gösteren bir işaretçinin (head) tanımlanması gerekir.
- Bunun yanında son elemanı gösteren bir işaretçi (tail) sona ekleme ve sondan silme işlemlerini hızlandırır.
- Ayrıca eleman sayısını tutmak için bir değişken tanımlamak işlemleri hızlandıracaktır.
- Bu değişkenler tek yönlü bağlı listede de kullanılabilir. Örneğin tek yönlü bağlı listeye bir **tail** eklemek sona eleman eklemeyi ve sondaki elemanı okumayı hızlandıracaktır.

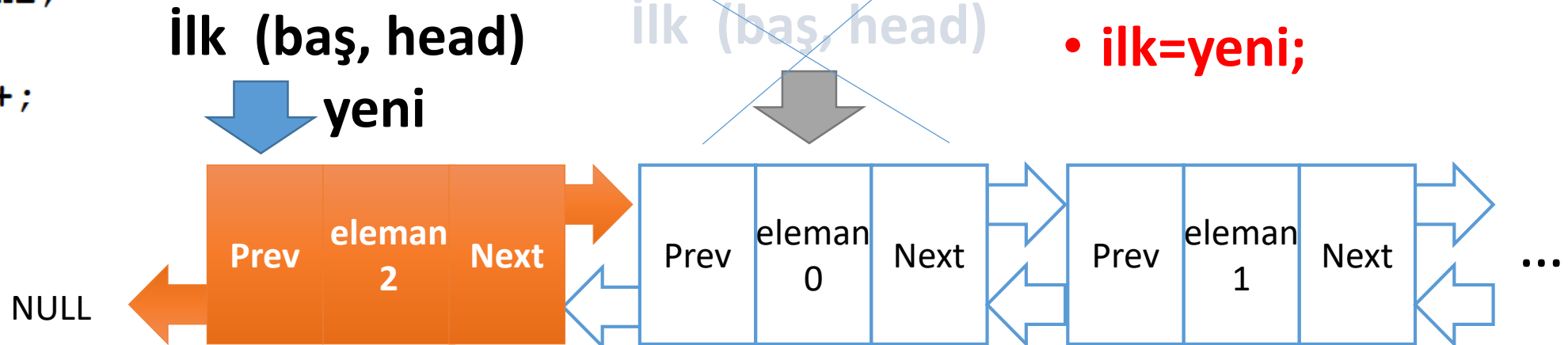
Başa eleman ekleme

```
void push_front(const Nesne& data) { //başa ekle
    Node<Nesne> *yeni = new Node<Nesne>(data);
    //liste boşmu?
    if (head == NULL) {
        head = yeni;
        tail = yeni;
    } else {
        head->prev = yeni;
        yeni->next = head;
        head = yeni;
    }
    elemanSayisi++;
}
```

Liste boşken eleman eklersek, ilk eleman ve son eleman aynı

Listede eleman varsa, yeni elemanı ilk elemanın öncesine ekle ve ilk elemanı (head) yeni eleman yap.

- **yeni->next=ilk;**
- **ilk->prev=yeni;**
- **ilk=yeni;**




Sona Eleman Ekleme

```
void push_back(const Nesne& data) {
    Node<Nesne> *yeni = new Node<Nesne>(data);
    //liste boşmu?
    if (head == NULL) {
        head = yeni;
        tail = yeni;
    } else {
        tail->next = yeni;
        yeni->prev = tail;
        tail = yeni;
    }
    elemanSayisi++;
}
```

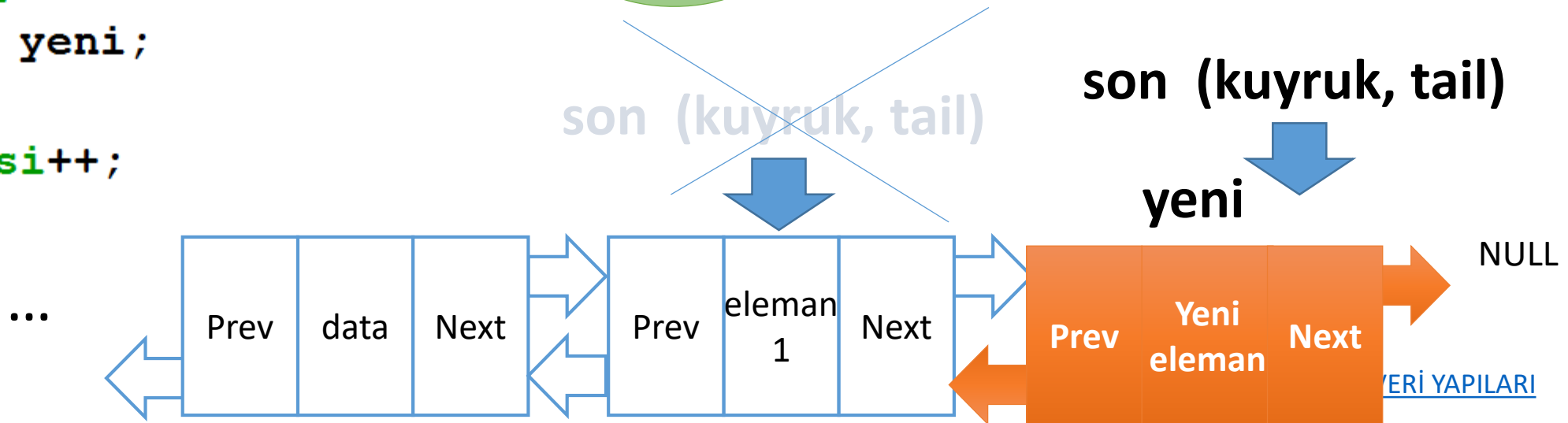
Liste boşken eleman eklersek, ilk eleman ve son eleman aynı

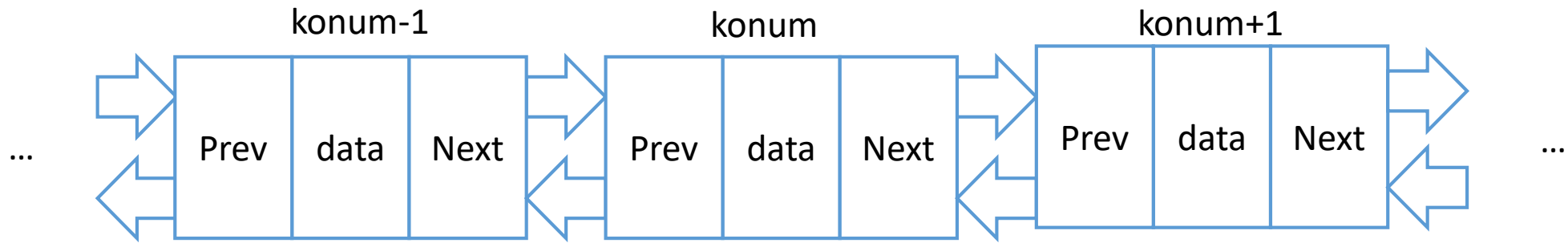
Listede eleman varsa, yeni elemanı son elemanın sonrasına ekle ve son elemanı (tail) yeni eleman olarak değiştir.

~~son (kuyruk)~~



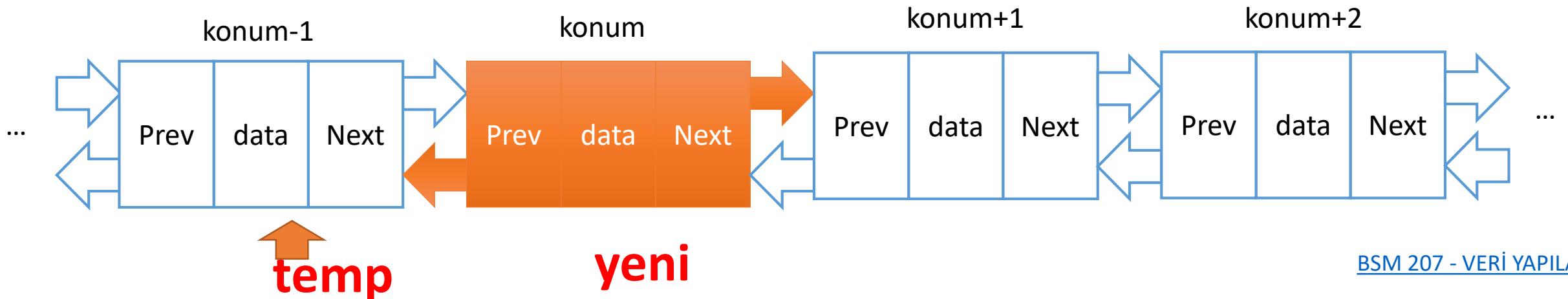
- Sona eleman ekleme başa eklemeye benzer şekilde gerçekleştirilir.
- Ancak sonu gösteren bir pointer (tail) tanımlanmamışsa, yeni düğümü eklemekten önce bir döngü ile ilk elemandan başlayıp son eleman bulunana kadar ilerlenir.





- **yeni->next = temp->next;**
- **temp->next->prev = yeni;**
- **temp->next = yeni;**
- **yeni->prev = temp;**

Tek yönlü bağlı listede olduğu gibi belirtilen konuma düğüm ekleneceği zaman bir önceki (veya çift yönlü olduğu için bir sonraki) düğümün üzerine gelinir ve bağlantılar gerçekleştirilir.

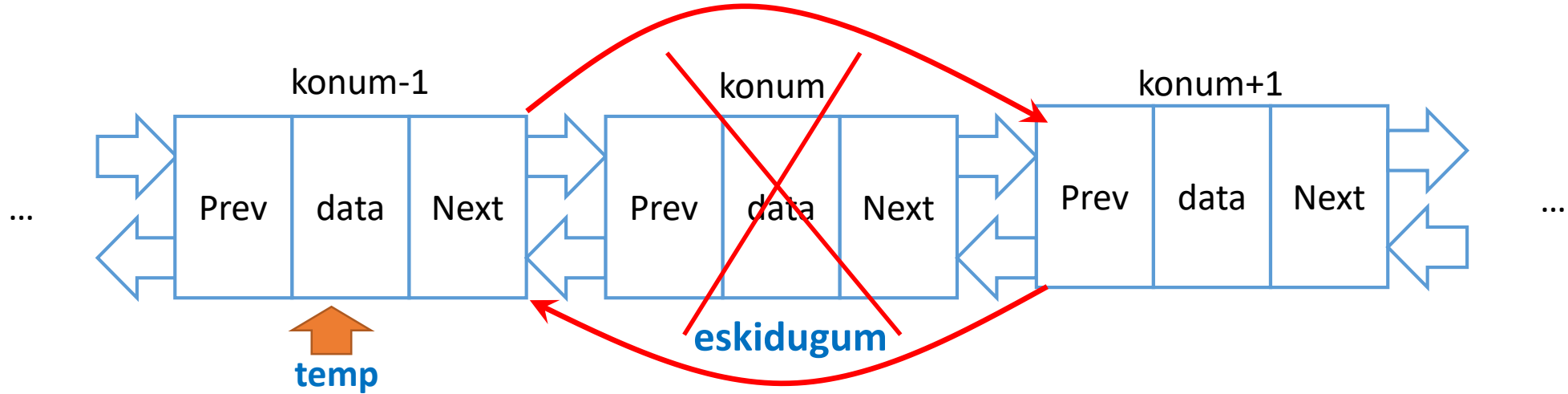


Belirtilen konuma eleman eklemek

```
void insert(int konum, const Nesne& data)
{
    throw (GecersizKonumHatasi) {
        //konum geçerli mi?
        if (konum < 0 | konum > elemanSayisi)
            throw GecersizKonumHatasi();
        //yeni düğüm oluştur
        if (konum == 0) { //liste başına ekle
            push_front(data);
        } else {
            //yeni düğüm
            Node<Nesne> *yeni = new Node<Nesne>(data);
            Node<Nesne> *temp = oncekiDugum(konum);
            //bağlantıyı yap
            yeni->next = temp->next;
            temp->next->prev = yeni;
            temp->next = yeni;
            yeni->prev = temp;
            elemanSayisi++;
        }
    }
}
```

- Eğer konum geçersiz ise fonksiyon hata fırlatılarak sonlandırılır.
- Eğer konum=0 ise bu durumda eleman liste başına eklenir. Burada push_front() çağırılmıştır.
- Eğer konum>0 ise yeni bir düğüm oluşturulduktan sonra düğümünden önceki konumdaki düğüm bulunur ve önceki slaytta anlatıldığı gibi bağlantılar gerçekleştirilir.

Belirtilen konumdan eleman silmek



- **eskidugum = temp->next;**
- **temp->next = eskidugum->next;**
- //silinecek eleman en sonda değilse
- **if (temp->next != NULL)**
- **eskidugum->next->prev = temp;**
- //çöpü temizle
- **delete eskidugum;**

Belirtilen konumdan eleman silmek

```
void remove(int konum)
throw (GecersizKonumHatasi, BosListeHatasi) {
    if (head == NULL) throw BosListeHatasi();
    if (konum < 0 | konum >= elemanSayisi)
        throw GecersizKonumHatasi();
    Node<Nesne> *temp = head;
    if (konum == 0) { //ilk eleman
        head = head->next;
        if (head != NULL)
            head->prev = NULL;
        delete temp;
    } else {
```

- Liste boş veya konum geçersiz ise hata fırlatılarak fonksiyon sonlandırılır.
- Belirtilen konum, listenin ilk elemanı ise ilk eleman işaretçisi bir ileriki konuma kaydırılır. Eğer bir ileriki konumda eleman varsa onun öncesi silinmiş olacağı için NULL yapılır.
- konum=0 için removefront() şeklinde bir fonksiyon da tanımlanabilir.

Belirtilen konumdan eleman silmek

```
} else {  
    //konum-1'deki düğüm  
    temp = oncekiDugum(konum) ;  
    //silinecek düğüm  
    Node<Nesne> *eskidugum = temp->next;  
    temp->next = eskidugum->next;  
    //silinecek eleman en sonda değilse  
    if (temp->next != NULL)  
        eskidugum->next->prev = temp;  
    //düğümü sil  
    delete eskidugum;  
}  
elemanSayisi--;  
}
```

- konum>0 için silinecek düğümden önceki düğüm bulunur.

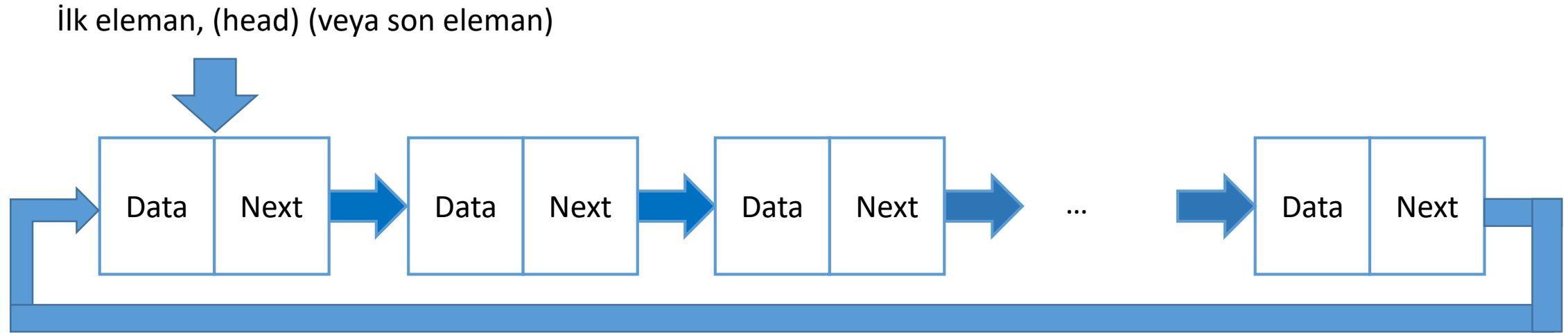
Belirtilen konumdaki elemanı okumak

Burada döndürülen nesne const ile değiştirilmesi engellenebilir:
const Nesne& at(int ..
const yazmazsak
örneğin;
liste1->at(5)=Nesne
şeklinde atama yaparak
elemanı değiştirebiliriz.
Böyle bir atamayı
engellemek için dönüş
tipini const yapmalıyız.

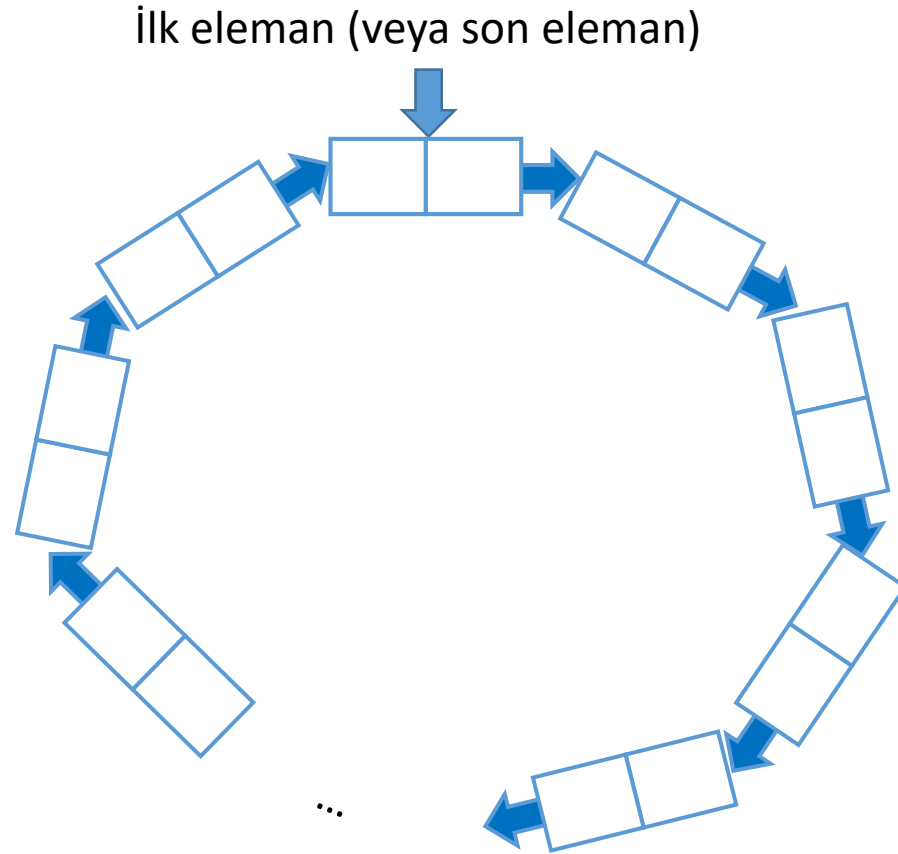
```
// belirtilen konumdaki elemanı döndür
Nesne& at(int konum) throw (Hata) {
    if (konum < 0 | konum > (elemanSayisi - 1))
        throw Hata("Geçersiz konum");
    //konumu bul
    Node<Nesne> *temp;
    temp = head;
    int sayac = 0;
    while (temp != NULL) {
        if (sayac == konum) {
            //eleman bulundu
            return temp->data;
        }
        temp = temp->next;
        sayac++;
    }
}
```

Dairesel Bağlı liste (Circular Linked List)

- Tek yönlü dairesel bağlı



Dairesel Bağlı liste

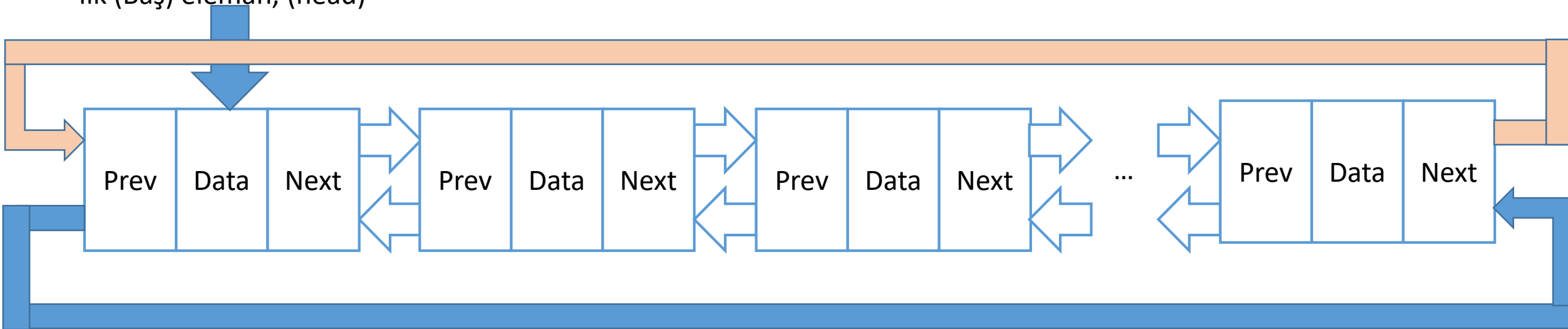


Bir önceki liste
dairese olarak
gösterilebilir

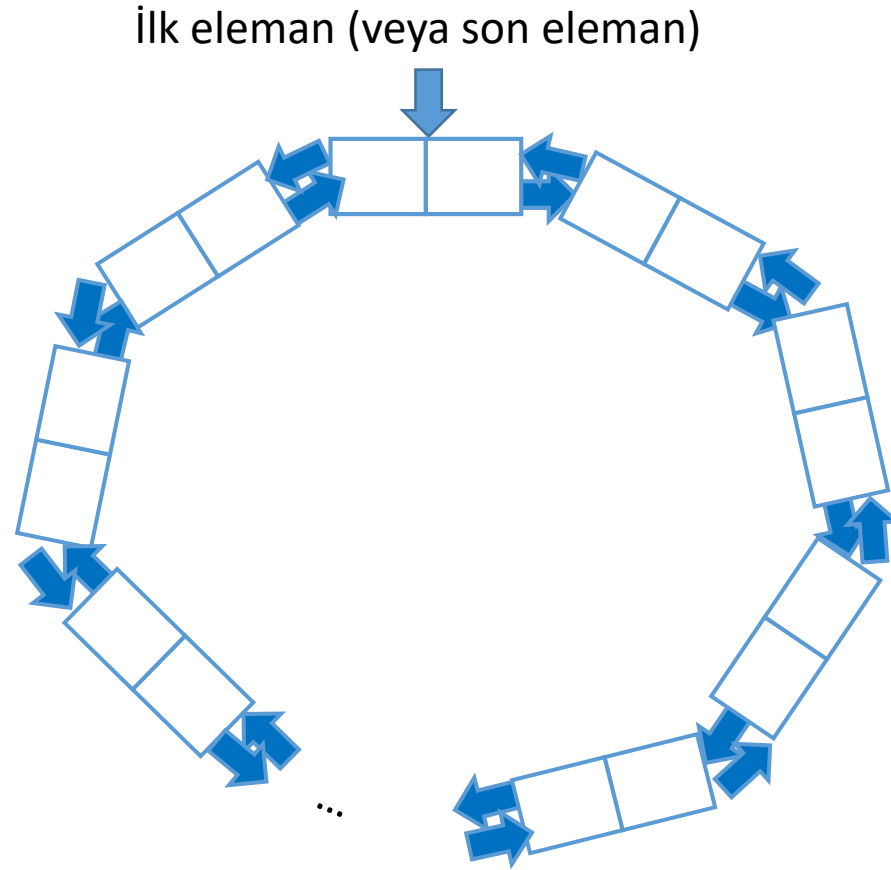
Dairesel Bağlı liste

- Çift yönlü dairesel bağlı

İlk (Baş) eleman, (head)



Dairesel Bağlı liste



C++ da listeler

`std::vector`

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, **vectors use a dynamically allocated array to store their elements**. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

`std::list`

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

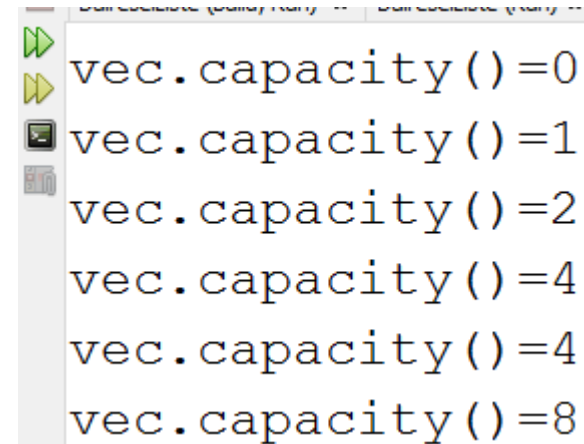
List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

C++ da listeler

```
#include <vector>
```

```
int main(int argc, char** argv) {  
  
    vector<string> vec;  
    cout<<"vec.capacity()="<<vec.capacity()<<endl;  
    vec.push_back("Pazartesi");//sona ekle  
    cout<<"vec.capacity()="<<vec.capacity()<<endl;  
    vec.push_back("Salı");  
    cout<<"vec.capacity()="<<vec.capacity()<<endl;  
    vec.push_back("Perşembe");  
    cout<<"vec.capacity()="<<vec.capacity()<<endl;  
    vec.push_back("Cuma");  
    cout<<"vec.capacity()="<<vec.capacity()<<endl;  
    vec.push_back("Pazar");  
    cout<<"vec.capacity()="<<vec.capacity()<<endl;  
  
    yazdir<string>(vec);  
}
```

```
template <typename T>  
void yazdir(vector<T> vec) {  
    cout<<"\n---liste-----"<<endl;  
    for(vector<string>::iterator itr=vec.begin();  
        itr != vec.end();  
        ++itr) {  
        cout<<" " <<*itr<<endl;  
    }  
}
```



```
vec.capacity()=0  
vec.capacity()=1  
vec.capacity()=2  
vec.capacity()=4  
vec.capacity()=4  
vec.capacity()=8
```

C++ da listeler

```
vec.pop_back(); //sondan sil  
yazdir<string>(vec);
```

```
//ilk elemanı sil  
vec.erase(vec.begin());  
yazdir<string>(vec);
```

```
// 3. konumdaki elemanı sil  
vec.erase(vec.begin()+2);  
yazdir<string>(vec);
```

```
// 3. konuma eleman ekle  
vec.insert(vec.begin()+2, "Çarşamba");  
yazdir<string>(vec);
```

```
//0. konumdaki elemanı değiştir  
cout<<"\nvec.at(0)="<<vec.at(0)<<endl;  
vec.at(0)="Pazartesi";  
cout<<"vec.at(0)="<<vec.at(0)<<endl;
```

C++ da listeler

```
#include <list>
```

```
list<string> list1;

list1.push_back("Pazartesi"); //sona ekle
list1.push_back("Salı");
list1.push_back("Çarşamba");
list1.push_back("Cuma");
yazdir<string>(list1);
```

```
list<string>::iterator itr=list1.begin();
list1.insert(itr,"Perşembe");
itr++;
list1.insert(itr,"Cumartesi");
yazdir<string>(list1);
```

```
template <typename T>
void yazdir(list<T> list1){
    cout<<"\n---liste-----"<<endl;
    for(list<string>::iterator itr=list1.begin();
        itr != list1.end();
        ++itr){
        cout<<" "<<*itr<<endl;
    }
}
```