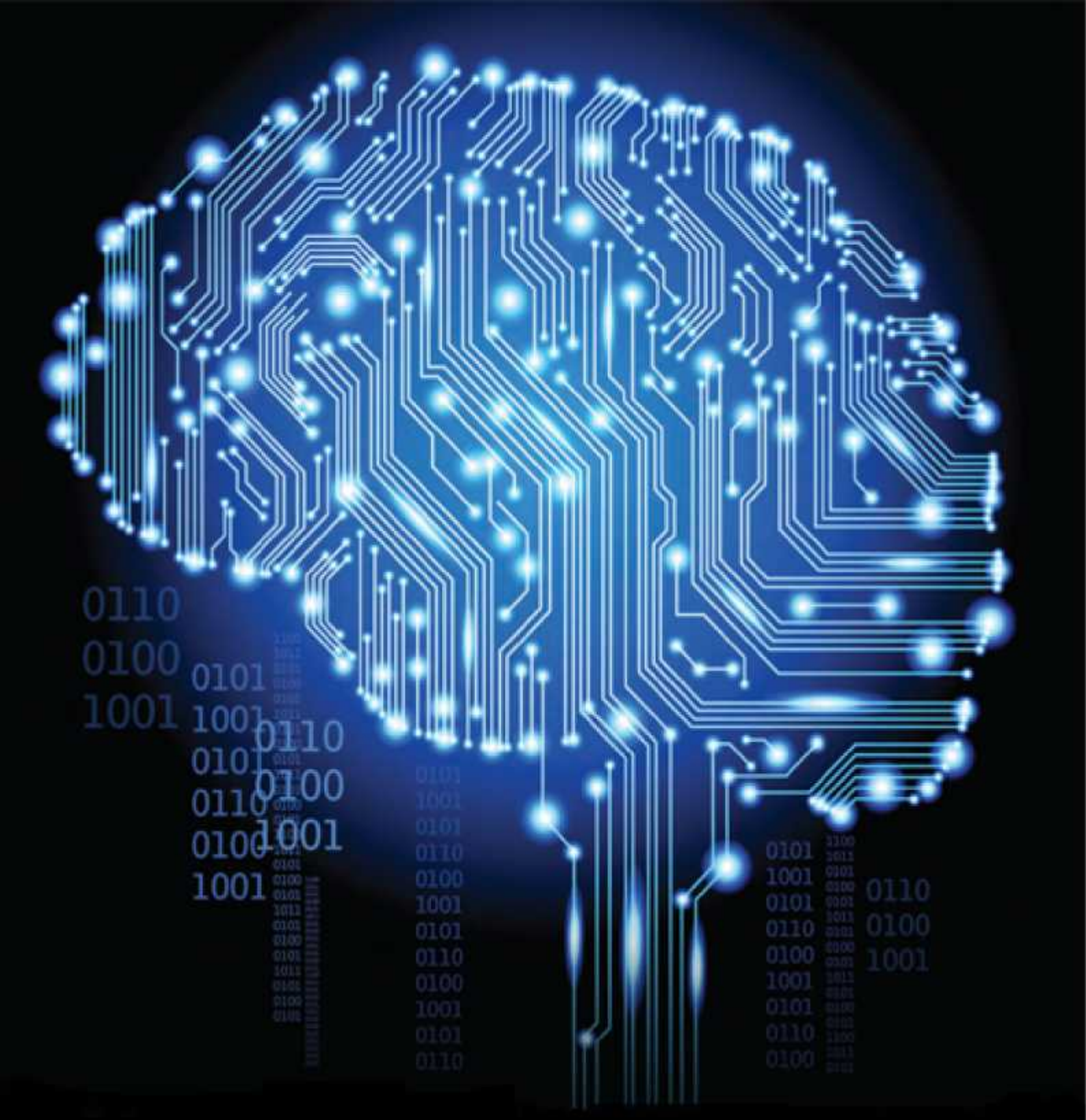


VERİ YAPILARI

DATA STRUCTURES

Asst.Prof.Dr. HAKAN KUTUCU

HAKAN KUTUCU

VERİ YAPILARI
(DATA STRUCTURES)

Düzenleyen
SAİM MEHMET ÖZTÜRK

KARABÜK ÜNİVERSİTESİ
Mühendislik Fakültesi Merkez Kampüsü – Karabük 2014

İçindekiler

2- Uygulama (*Implementation*): Matematiksel ve mantıksal modellemenin uygulamasıdır. Diziler, programlamada çok kullanışlı veri yapıları olmasına rağmen bazı dezavantajları ve kısıtları vardır. Örneğin;

- Derleme aşamasında dizinin boyutu bilinmelidir,
- Diziler bellekte sürekli olarak yer kaplarlar. Örneğin `int` türden bir dizi tanımlanmışa, eleman sayısı çarpı `int` türünün kapladığı alan kadar bellekte yer kaplayacaktır,
- Ekleme işleminde dizinin diğer elemanlarını kaydırmak gerekir. Bu işlemi yaparken dizinin boyutunun da aşılmaması gerekir,
- Silme işlemlerinde değeri boş elemanlar oluşacaktır.

Bu ve bunun gibi sorunların üstesinden gelmek ancak Bağlı Listelerle (*Linked List*) mümkün hale gelir.

Soyut veri tipleri (*ADT*)'nin resmi tanımını şu şekilde yapabiliriz; Soyut veri tipleri (*ADTs*) sadece veriyi ve işlemleri (*operasyonları*) tanımlar, uygulama yoktur. Bazı veri tipleri;

- Arrays (*Diziler*)
- Linked List (*Bağlı liste*)
- Stack (*Yığın*)
- Queue (*Kuyruk*)
- Tree (*Ağaç*)
- Graph (*Graf, çizge*)

Veri yapılarını çalışırken,

1- Mantıksal görünüşüne bakacağız,
2- İçinde barındırdığı işlemelere bakacağız (*operation*),

Bir bilgisayar programında uygulamasını yapacağız (*biz uygulamalarımızı C programlama dilini kullanarak yapacağız*).

2.5 BAĞLI LİSTELER (Linked Lists)

Liste (*list*) sözcüğü aralarında bir biçimde öncelik-sonralık ya da altlık-üstlük ilişkisi bulunan veri öğeleri arasında kurulur. Doğrusal Liste (*Linear List*) yapısı yalnızca öncelik-sonralık ilişkisini yansıtabilecek yapıdadır. Liste yapısı daha karmaşık gösterimlere imkan sağlar. Listeler temel olarak tek bağlı ve çift bağlı olmak üzere ikiye ayrılabilir. Ayrıca listelerin dairesel veya doğrusal olmasına göre de bir gruplandırma yapılabilir. Tek bağlı listelerde node'lar sadece bir sonraki node ile bağlanarak bir liste oluştururlar. Çift bağlı (*iki bağlı*) listelerde ise bir node'da hem sonraki noda hem de önceki noda bir bağlantı vardır. Bu bağlantılar Forward Link (*ileri bağ*) ve Backward Link (*geri bağ*) olarak adlandırılırlar. Doğrusal listelerde listede bulunan en son node'un başka hiçbir noda bağlantısı yoktur. Bağ değeri olarak `NULL` alırlar. Dairesel listelerde ise en sondaki node, listenin başındaki node'a bağlanmıştır. Aşağıda buna göre yapılan sınıflandırma görülmektedir.

- Tek Bağlı Listeler (*One Way Linked List*)
  - Tek Bağlı Doğrusal Listeler (*One Way Linear List*)
  - Tek Bağlı Dairesel Listeler (*One Way Circular List*)
- Çift Bağlı listeler (*Double Linked List*)
  - Çift Bağlı Doğrusal Listeler (*Double Linked Linear List*)
  - Çift Bağlı Dairesel Listeler (*Double Linked Circular List*)

İlerleyen kısımlarda bu listeler ve bunlarla ilgili program parçaları anlatılacaktır. Şimdilik bilinmesi gereken ayrıntı, çift bağlı listelerde `previous` adında ikinci bir pointer daha vardır. Diğerleri aynı yapıdadır.

Bağlı Listeler İle İşlemler

Bağlı listeler üzerinde;

1- Liste oluşturma,
2- Listeye eleman eklemek,
3- Listeden eleman silmek,
4- Arama yapmak,
5- Listenin elemanlarını yazmak,
6- Listenin elemanlarını saymak.

vb. gibi ve kuşkusuz daha fazla işlemler yapılabilir. Bu işlemlerden bazılarını açıklayalım ve fonksiyon halinde yazalım.

2.6 TEK BAĞLI DOĞRUSAL LİSTELER

Tek bağlı doğrusal liste, öğelerinin arasındaki ilişki (*Logical Connection*)'ye göre bir sonraki öğenin bellekte yerleştiği yerin (*Memory Location*) bir gösterge ile gösterildiği yapıdır. Bilgisayar belleği doğrusaldır. Bilgiler sıra sıra hücrelere saklanır. Her bir bilgiye daha kolay ulaşmak için bunlara numara verilir ve her birine **node** adı verilir. Data alanı, numarası verilen node'da tutulacak bilgiyi ifade eder. Next (*link*) alanı ise bir node'dan sonra hangi node gelecekse o node'un bellekteki adresi tutulur.

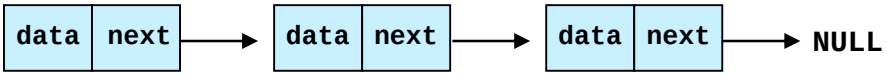
Tek bağlı listelerin genel yapısı aşağıda verilmiştir. Konu anlatılırken daima bu temel yapı kullanılacağından unutmamalısınız.

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

Bağlı listeler içerisindeki düğümlerin yukarıdaki tanımlamayla iki öğesinin olduğu görülüyor. Birinci öğe olan `data`, her türden veri içerebilir, örneğin telefon numaraları, TC kimlik numaraları vb. gibi. Biz `int` türden bir nesneyi yeterli bulduk. İkinci öğe olan `next`, bir bağlı listede mutlaka bulunması gereken bir öğedir. Dikkat edilirse `struct node` tipinde bir işaretçidir.

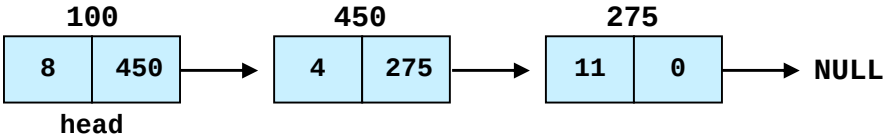
Şekil 2.1 Tek Bağlı Listelerde bir node'un mantıksal yapısı.

Tek bağlı listelerin yapısında bulunan `node` tipindeki `*next` işaretçisi, rekürsif fonksiyonlarla karıştırılmamalıdır. Burada `next`, bir sonraki `node` türden düğümün adresini gösterecektir.



Şekil 2.2 Tek bağlı listeler.

Alttaki şekilde her çiftli kutucuk liste yapısını temsil etmektedir. Bu kutucukların üstündeki numaralar ise bellekte bulundukları yerin adresidir. Burada önemli olan, `next` göstericisinin değerlerinin, yani düğüm adreslerinin sıralı olmayışıdır. İlk düğümün adresi 100, ikincisinin 450 ve üçüncüsünün ise 275'tir. Yani bellekte neresi boşsa o adresi almışlardır. Oysa dizilerde tüm elemanlar sıralı bir şekilde bellekte yer kaplıyordu.



Şekil 2.3 Liste yapısı ve bellekteki adreslerinin mantıksal gösterimi.

Listenin ilk elemanı genellikle `head` olarak adlandırılır. `head`'den sonra diğer elemanlara erişmek kolaydır. Bazı kaynaklarda listenin sonundaki elemanın ismi `tail` olarak adlandırılmıştır. Fakat biz bu ismi notlarımızda kullanmayacağız.

Tek Bağlı Doğrusal Liste Oluşturmak ve Eleman Eklemek

Bu örnekte ilk olarak listeyi oluşturacağız, ardından eleman ekleme yapacağız.

```
main() {
    struct node *head; // henüz bellekte yer kaplamıyor
    head = (struct node *)malloc(sizeof(struct node));
    // artık bellekte yer tahsis edilmiştir.

    head -> data = 1;
    head -> next = NULL;

    /* listeye yeni eleman ekleme */
    /* C++'ta head -> next = new node() şeklinde kullanılabilir. */
    head -> next = (struct node *)malloc(sizeof(struct node));
    head -> next -> data = 3;
    head -> next -> next = NULL;
```

```
}
```

Peki, eleman eklemek istersek sürekli olarak `head->next->next`... diye uzayacak mı? Tabi ki hayır! Elbette ki bunu yapmanın daha kolay bir yolu var.

Tek Bağlı Doğrusal Listenin Başına Eleman Eklemek

Bir fonksiyonla örnek verelim. Siz isterseniz `typedef` anahtar sözcüğünü kullanarak sürekli `struct` yazmaktan kurtulabilirsiniz fakat biz akılda kalıcı olması açısından `struct`'ı kullanmaya devam edeceğiz.

```
// Tek bağlı doğrusal listenin başına eleman eklemek
struct node *addhead(struct node *head,int key) {

    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp -> data = key;
    temp -> next = head; // temp'in next'i şu anda head'i gösteriyor.
    /* Bazen önce listenin boş olup olmadığı kontrol edilir, ama gerekli değil
     aslında. Çünkü boş ise zaten head=NULL olacaktır ve temp olan ilk düğümün
     next'i NULL gösterecektir. */
    head = temp; /* head artık temp'in adresini tutuyor yani eklenen düğüm
                    listenin başı oldu. */
    return head;
}
```

Tek Bağlı Doğrusal Listenin Sonuna Eleman Eklemek

Listenin sonuna ekleme yapabilmek için liste sonunu bilmemiz gerekiyor. Listede eleman olduğunu varsayıyoruz. Liste sonunu bulabilmek içinse bu liste elemanları üzerinde tek tek ilerlemek gerektiğinden `head`'in adresini kaybetmemek için bir değişken oluşturacağız ve `head`'in adresini bu değişkene atayacağız.

```
struct node *addlast(struct node *head,int key) {
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    /* C++'ta struct node *temp = new node();
     * şeklinde kullanılabileceğini unutmayınız. */
    temp -> data = key;
    temp -> next = NULL;
    struct node *temp2 = head;
    /* Aşağıdaki while yapısı traversal(dolaşma) olarak adlandırılır */
    while(temp2 -> next != NULL)
        temp2 = temp2 -> next;
    temp2 -> next = temp;
    return head;
}
```

Tek Bağlı Doğrusal Liste Elemanlarının Tüm Bilgilerini Yazdırmak

Listedeki elemanların adreslerini, data kısımlarını ve sonraki düğüm adresini ekrana basan `listinfo` ismindeki fonksiyon aşağıdaki gibi yazılabilir.

```
void listinfo(struct node* head) {
    int i = 1;
    while(head != NULL) {
        printf("%d. Dugumunun Adresi = %p \n", i, head);
        printf("%d. Dugumunun verisi = %d \n", i, head -> data);
        printf("%d. Dugumunun Bagli Oldugu Dugumun Adresi= %p\n\n",i, head->next);
        head = head -> next;
        i++;
    }
}
```

Tek Bağlı Doğrusal Listenin Elemanlarını Yazdırmak

Fonksiyon sadece ekrana yazdırma işi yapacağından `void` olarak tanımlanmalıdır. Liste boşsa ekrana listede eleman olmadığına dair mesaj da verilmelidir.

```
void print(struct node *head) {
    if(head == NULL) {
```

```
        printf("Listede eleman yok");
        return;
    }
    struct node *temp2 = head;
    while(temp2!= NULL) { // temp2->next!=NULL koşulu olsaydı son düğüm yazılmazdı
        printf("%d\n", temp2 -> data);
        temp2 = temp2 -> next;
    }
}
```

Şüphesiz elemanları yazdırmak için özyinelemeli bir fonksiyon da kullanılabilirdi.

```
//Tek bağlı liste elemanlarını özyinelemeli yazdırmak
void print_recursive(struct node *head) {
    if(head == NULL)
        return;
    printf("%d\t", head -> data);
    print_recursive (head -> next);
}
```

SORU: Yukarıdaki fonksiyon aşağıdaki gibi yazılırsa çıktısı ne olur.

```
void print_recursive2(struct node *head) {
    if(head == NULL)
        return;
    print_recursive2 (head -> next);
   printf("%d\t", head -> data);

}
```

Tek Bağlı Doğrusal Listenin Elemanlarını Saymak

Listenin elemanlarını saymak için `int` tipinden bir fonksiyon oluşturacağız. Ayrıca listede eleman olup olmadığını da kontrol etmeye gerek yoktur. Çünkü eleman yok ise while döngüsü hiç çalışmayacak ve 0 değerini döndürecektir.

```
int count(struct node *head) {
    int counter = 0;
    while(head != NULL) { // head->next!=NULL koşulu olsaydı son düğüm sayılmazdı
        counter++;
        head = head -> next;
    }
    return counter;
}
```

Bu işlemi özyinelemeli yapmak istersek:

```
int count_recursive(struct node *head) {
    if (head == NULL)
        return 0;
    return count_recursive(head->next) + 1;
}
```

Tek Bağlı Doğrusal Listelerde Arama Yapmak

Bu fonksiyon ile liste içinde arama yapılmaktadır. Eğer aranan bilgi varsa, bulunduğu `node`'un adresiyle geri döner. Bu fonksiyon bulundu ise true bulunmadı ise false döndürecek şekilde de düzenlenebilir.

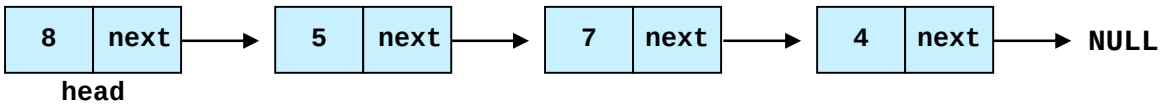
```
struct node* locate(struct node* head, int key) {
    struct node* locate = NULL;
    while(head != NULL)
        if(head -> data != key)
            head = head -> next;
        else {
            locate = head;
            break;
        }
    return(locate);
}
```

Tek Bağlı Doğrusal Listelerde İki Listeyi Birleştirmek

`list_1` ve `list_2` adındaki iki listeyi birleştirmek için `concatenate` fonksiyonunu kullanabiliriz.

```
void concatenate(struct node*& list_1, node* list_2) { // parametrelere dikkat
    if(list_l == NULL)
        list_1 = list_2;
    else
        last(list_l) -> next = list_2; // last isimli fonksiyona çağrı yapılıyor
}
```

Tek Bağlı Doğrusal Listelerde Verilen Bir Değere Sahip Düğümü Silmek



Şekil 2.4 Tek bağlı listelerin mantıksal yapısı.

Tek bağlı listelerde verilen bir düğümü silme işlemi, o düğümün başta ya da ortada olmasına göre farklılık gösterir. İlk düğüm silinmek istenirse ikinci elemanın adresini yani `head`'in `next` işaretçisinin tuttuğu adresi `head`'e atayarak baştaki eleman silinebilir. Eğer ortadan bir düğüm silinmek istenirse bir önceki düğümü, silinmek istenen düğümden bir sonraki düğüme bağlamak gerekir. Şimdi `_remove` ismini vereceğimiz bu fonksiyonu yazalım.

```
struct node *remove(struct node *head, int key) {
    if(head == NULL) {
        printf("Listede eleman yok\n");
        return;
    }
    struct node *temp = head;
    if(head -> data == key) { // ilk düğüm silinecek mi diye kontrol ediliyor.
        head = head -> next; // head artık bir sonraki eleman.
        free(temp);
    }
    else if(temp -> next == NULL) { // Listede tek düğüm bulunabilir.
         printf("Silmek istediginiz veri bulunmamaktadir.\n\n");
         return head;
    }
    else {
        while(temp -> next -> data != key) {
            if(temp -> next -> next == NULL) {
                printf("Silmek istediginiz veri bulunmamaktadir.\n\n");
                return head;
            }
            temp = temp -> next;
        }
        struct node *temp2 = temp -> next;
        temp -> next = temp -> next -> next;
        free(temp2);
    }
    return head;
}
```

Tek Bağlı Doğrusal Listelerde Verileri Tersten Yazdırmak

Tek bağlı listenin elemanlarını tersten yazdıran `print_reverse` adlı bir fonksiyon yazalım. Bu fonksiyonu yazarken her veriyi yeni bir listenin başına ekleyeceğiz ve böylece ilk listenin tersini elde etmiş olacağız. Bunun için `addhead` adlı fonksiyonu kullanacağız.. `print_reverse` fonksiyonunda `struct node*` türden yeni bir değişken daha tanımlayacağız ve `head`'in elemanlarını bu yeni listenin sürekli başına ekleyerek verileri ters bir biçimde sıralayacağız ve yazdırma işlemini gerçekleştireceğiz. Aslında tersten yazdırma işini rekürsif olarak yapan bir fonksiyon daha önce SORU olarak sorulmuş fonksiyondur. Rekürsif fonksiyonları iyice kavramanız için bolca örnek yapmalısınız. Çünkü veri yapılarında rekürsif fonksiyonların çok büyük bir önemi vardır.

```
void print_reverse(struct node *head) {
    struct node *head2 = NULL; // yeni listenin başını tutacak adres değişkeni
```

```
    struct node *temp = head;
    while(temp != NULL) {
        head2 = addhead(head2, temp -> data);
        temp = temp -> next;
    }
    print(head2);
}
```

Tek Bağlı Doğrusal Listenin Kopyasını Oluşturmak

`head`'in kopyası oluşturulup `kopya` geri gönderilmektedir. `kopya` listesinde veriler aynı fakat adresler farklıdır.

```
struct node* copy(struct node* head) {
    struct node* kopya = NULL;
    if(head != NULL)
        do {
            concatenate(kopya, cons(head -> data));
            head = head -> next;
        } while(head != NULL);
    return kopya;
}
```

Listeyi Silmek

Listelerin kullanımı bittiğinde bunların bellekte yer işgal etmemesi için tüm düğümlerinin silinmesi gereklidir. `head` düğümünün silinmesi listenin kullanılmaz hale gelmesine neden olur ancak `head` ten sonraki düğümler hala bellekte yer kaplayama devam eder.

```
struct node *destroy(struct node *head) {
    if(head == NULL) {
        printf("Liste zaten bos\n");
        return;
    }
    struct node *temp2;
    while(head!= NULL) { // while içindeki koşul temp2 -> next, NULL değilse
        temp2=head;
        head = head->next;
        free(temp2);
    }
    return head;
}
```

SORU: destroy fonksiyonunu özyinelemeli olarak yazınız.

Main Fonksiyonu İçinde Tanımladığımız Tüm Fonksiyonların Çağırılması

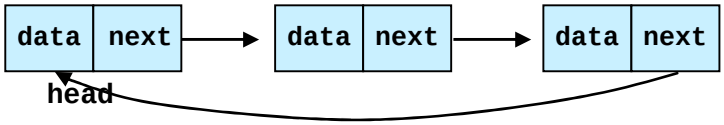
Yukarıda tanımladığımız fonksiyonların çalıştırılması için tüm fonksiyonlar, struct tanımlaması dahil aşağıdaki main() fonksiyonu üzerinde yazılır.

```
main(){
    int secim,data;
    struct node *head = NULL;
    while(1){
        printf("1-Listenin Basina Eleman Ekle\n");
        printf("2-Listenin Sonuna Eleman Ekle\n");
        printf("3-Listeyi Gorme\n");
        printf("4-Listeden verilen bir degere sahip dugum silmek\n");
        printf("5-Listeyi sil\n");
```

```
            printf("6-Listedeki eleman sayisi\n");
            printf("7-Listenin tum eleman bilgileri\n");
            printf("8-Programdan Cikma\n");
            printf("Seciminiz....?");
            scanf("%d",&secim);
            switch(secim){
            case 1:
                printf("Eklemek istediginiz degerini giriniz..?");
                scanf("%d",&data);
                head=addhead(head,data);
                break;
            case 2:
                printf("Eklemek istediginiz degerini giriniz..?");
                scanf("%d",&data);
                head=addlast(head,data);
                break;
            case 3:
                print(head);
                break;
            case 4:
                printf("Silmek istediginiz degerini giriniz..?");
                scanf("%d",&data);
                head=delete(head,data);
                break;
            case 5:
                head=destroy(head);
                break;
            case 6:
                printf("Listede %d eleman vardir\n",count(head));
                break;
            case 7:
                listinfo(head);
                break;
            case 8:
                exit(1);
                break;
            default: printf("Yanlis secim\n");
            }
        }
    }
```

2.7 TEK BAĞLI DAİRESEL (Circle Linked) LİSTELER

Tek bağlı dairesel listelerde, doğrusal listelerdeki birçok işlem aynı mantık ile benzer şekilde uygulanır; fakat burada dikkat edilmesi gereken nokta, dairesel bağlı listede son elemanının `next` işaretçisi `head`'i göstermektedir..



Şekil 2.5 Dairesel bağlı listeler.

Tek Bağlı Dairesel Listelerde Başa Eleman Eklemek

Tek bağlı listelerde yaptığımızdan farklı olarak `head`'in global olarak tanımlandığı varsayılmıştır. Liste yoksa oluşturuluyor, eğer var ise, `struct node*` türden `temp` ve `last` düğümleri oluşturularak `last`'a `head`'in adresi atanıyor (*`head`'in adresini kaybetmememiz gerekiyor*). `temp`'in `data`'sına parametre değişkeninden gelen veri aktarıldıktan sonra `last` döngü içerisinde ilerletilerek listenin son elemanı göstermesi sağlanıyor. `temp` `head`'i gösterecek şekilde atama yapıldıktan sonra listenin son elemanını gösteren `last`'ın `next` işaretçisine de `temp`'in adresi atanıyor. Şu anda `last` eklenen verinin bulunduğu düğümü gösteriyor değil mi? `temp`'in `next` işaretçisi ise `head`'i gösteriyor. `head`'e `temp` atanarak işlem tamamlanmış oluyor. **DİKKAT!** Artık `temp`'in `next` göstericisi, `head`'in bir önceki adres bilgisini tutuyor.

```
void insertAtFront(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        struct node *last = head;
        temp -> data = key;
        while(last -> next != head) // listenin son elemanı bulunuyor.
            last = last -> next;
        temp -> next = head;
        last -> next = temp;
        head = temp;
    }
}
```

Tek Bağlı Dairesel Listelerde Sona Eleman Eklemek

Fonksiyon, başa ekleme fonksiyonuna çok benzemektedir. Listenin `NULL` olup olmadığı kontrol ediliyor.

```
void insertAtLast(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        struct node *last = head;
        temp -> data = key;
        // listenin son elemanı bulunuyor.
        while(last -> next != head)
            last = last -> next;
        temp -> next = head;
        last -> next = temp;
    }
}
```

Görüldüğü gibi başa ekleme fonksiyonunun sonundaki `head = temp;` satırını kaldırmak yeterlidir. Aynı fonksiyonu aşağıdaki şekilde de yazabilirdik. Burada `temp` isminde bir değişkene ihtiyacımızın olmadığını gözlemleyin.

```
void insertAtLast(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
    }
    else {
        struct node *last = head;
        while(last -> next != head) // listenin son elemanı bulunuyor.
            last = last -> next;
        last -> next = (struct node *)malloc(sizeof(struct node));
        last -> next -> next = head;
        last -> next -> data = key;
    }
}
```

Yazılan fonksiyonları siz de bilgisayarınızda kodlayarak mantığı kavramaya çalışınız. Ancak çok miktarda alıştırma yaptıktan sonra iyi bir pratik kazanabilirsiniz.

Tek Bağlı Dairesel Listelerde İki Listeyi Birleştirmek

`concatenate` fonksiyonu tek bağlı dairesel listelerde iki listeyi verilen ilk listenin sonuna ekleyerek birleştiren fonksiyon olarak tanımlanacaktır. Bu fonksiyonun yazımında önemli olan doğru işlem sırasını takip etmektir. Eğer öncelikli yapılması gereken bağlantılar sonraya bırakılırsa son `node`'un bulunmasında sorunlar çıkacaktır. `list_1` boş ise `list_2`'ye eşitleniyor. Eğer boş değilse her iki listenin de `last()` fonksiyonuyla son elemanları bulunuyor ve `next` işaretçilerine bir diğerinin gösterdiği adres değeri atanıyor.

```
// list_1 listesinin sonuna list_2 listesini eklemek
void concatenate(struct node*& list_1, struct node* list_2){ //parametrelere dikkat
    if(list_l == NULL)
        list_1 = list_2;
    else {
       // Birinci listenin son düğümünü last olarak bulmak için
        struct node *last=list_1;
        while(last -> next != list_1)
            last = last -> next;
        last->next=list_2; //Birinci listenin sonu ikinci listenin başına bağlandı
        // İkinci listenin son düğümünü last olarak bulmak için
        last=list_2;
        while(last -> next != list_2)
            last = last -> next;
        last->next=list_2; //İkinci listenin sonu birinci listenin başına bağlandı
     }
}
```

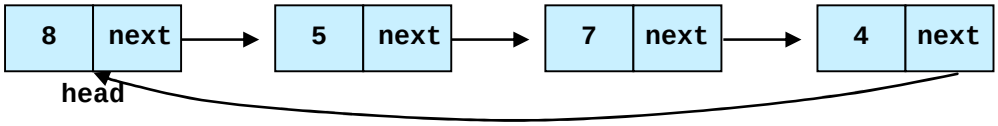
Tek Bağlı Dairesel Listede Arama Yapmak

Bu fonksiyon ile liste içinde arama yapılmaktadır. `node`'lar taranarak `data`'lara bakılır. Eğer aranan bilgi varsa, bulunduğu `node`'un adresiyle geri döner.

```
//head listesinde data'sı veri olan node varsa adresini alma
struct node* locate(int veri, struct node* head) {
    struct node* locate = NULL;
    struct node* temp = head;
    do {
        if(head -> data != veri)
            head = head -> next;
        else {
            locate = head;
            break;
    } while(head != temp);
    return(locate);
}
```

Tek Bağlı Dairesel Listelerde Verilen Bir Değere Sahip Düğümü Silmek

Silme işlemine dair fonksiyonumuzu yazmadan önce tek bağlı dairesel listelerin mantıksal yapısını tekrar gözden geçirmekte fayda vardır. Bu liste yapısında son düğümün `next` işaretçisi `head`'i gösteriyor ve dairesel yapı sağlanıyor.



Şekil 2.6 Tek bağlı listelerin mantıksal yapısı.

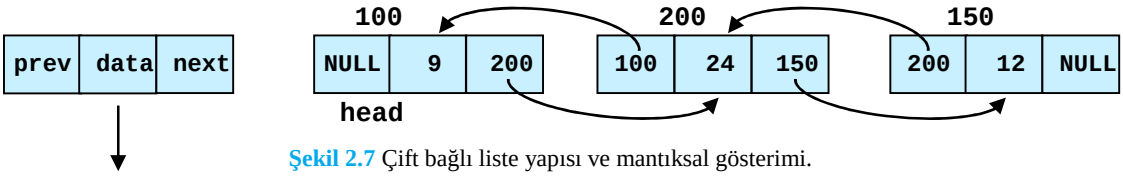
Tek bağlı dairesel listelerde verilen bir değere sahip düğümü silme işlemi için `deletenode` isimli bir fonksiyon yazacağız. Fonksiyonda liste boş ise hemen fonksiyondan çıkılır ve geriye `false` değeri döndürülür. Eğer silinecek düğüm ilk düğüm ise daha önce yaptığımız gibi ilk düğüm listeden çıkarılır. Ancak burada listenin son düğümünü yeni head düğümüne bağlamak gereklidir. Bu yüzden önce son düğüm bulunur.

```
struct node *deletenode(struct node *head, int key) {

    if(head == NULL) {
        printf("Listede eleman yok\n");
        return;
    }
    struct node *temp = head;
    if(head -> data == key) { // ilk düğüm silinecek  mi diye kontrol ediliyor.
        struct node *last=head;
        while(last -> next != head)
            last = last -> next;
        head = head -> next; // head artık bir sonraki eleman.
        last->next=head;
        free(temp);
    }
    else if(temp -> next == NULL) { // Listede tek düğüm bulunabilir.
         printf("Silmek istediginiz veri bulunmamaktadir.\n\n");
    }
    else {
        while(temp -> next -> data != key) {
            if(temp -> next -> next == NULL) {
                printf("Silmek istediginiz veri bulunmamaktadir.\n\n");
                return head;
            }
            temp = temp -> next;
        }
        struct node *temp2 = temp -> next;
        temp -> next = temp -> next -> next;
        free(temp2);
    }
    return head;
}
```

2.8 ÇİFT BAĞLI DOĞRUSAL(Double Linked) LİSTELER

Çift bağlı listelerin mantıksal yapısı Şekil 2.7'de gösterilmiştir. Her düğümün `data` adında verileri tutacağı bir değişkeni ile kendinden önceki ve sonraki düğümlerin adreslerini tutacak olan `prev` ve `next` isminde iki adet işaretçisi vardır. Listenin başını gösteren işaretçi `head` yapı değişkenidir. Şekilde `head`'in adresi `100`'dür ve `head`'in `prev` işaretçisi herhangi bir yeri göstermediğinden `NULL` değer içermektedir. `next` işaretçisi ise bir sonraki düğümün adresi olan `200` değerini içermektedir. İkinci düğümün `prev` işaretçisi `head`'in adresi olan `100` değerini tutmakta, `next` işaretçisi ise son düğümün adresi olan `150` değerini tutmaktadır. Nihayet son düğümün `prev` işaretçisi kendinden önceki düğümün adresini yani `200` değerini tutmakta ve `next` işaretçisi ise `NULL` değer içermektedir.



Şekil 2.7 Çift bağlı liste yapısı ve mantıksal gösterimi.

Çift bağlı listelerin struct yapısı aşağıda verilmiştir;

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

Çift Bağlı Doğrusal Listenin Başına Eleman Eklemek

`head` değişkeninin global olarak tanımlandığını varsayarak `insertAtFirst` fonksiyonunu yazalım.

```
void insertAtFirst(int key) {
    if(head == NULL) { // liste yoksa oluşturuluyor
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = NULL;
        head -> prev = NULL;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> data = key;
        temp -> next = head;
        temp -> prev = NULL;
        head -> prev = temp;
        head = temp;
    }
}
```

Çift Bağlı Doğrusal Listenin Sonuna Eleman Eklemek

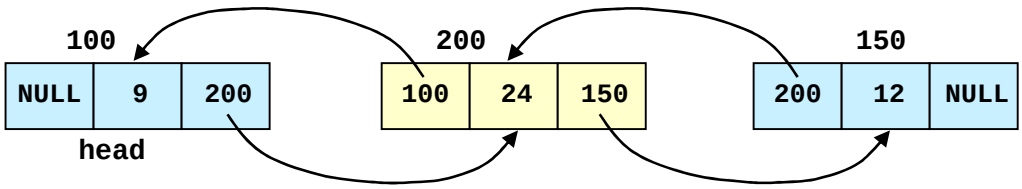
```
void insertAtEnd(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = NULL;
        head -> prev = NULL;
    }
    else {
        struct node *temp = head;
        struct node *temp2 = (struct node *)malloc(sizeof(struct node));
        while(temp -> next != NULL) // listenin sonunu bulmamız gerekiyor.
            temp = temp -> next;
        temp2 -> data = key;
        temp2 -> next = NULL;
        temp2 -> prev = temp;
        temp -> next = temp2;
    }
}
```

Çift Bağlı Doğrusal Listelerde Araya Eleman Eklemek

```
// head listesinin n. düğümünün hemen ardına other_node düğümünü ekleme
void addthen(node* other_node, node*& list, int n) {
    node* temp = head;
    int i = 1;

    while(i < n) {
        head = head -> next;
        i++;
    }
    other_node -> prev = head;
    other_node -> next = head -> next;
    head -> next = other_node;
    head = temp;
}
```

Çift Bağlı Doğrusal Listelerde Verilen Bir Değere Sahip Düğümü Silmek



Şekil 2.8 Silinmek istenen ortadaki düğüm sarı renkte gösterilmiştir.

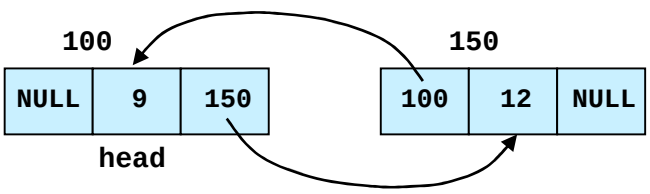
Silme işlemi diğer liste türlerine göre biraz farklılık göstermektedir. İlk düğüm silinmek istenirse `head`'in `next` işaretçisinin tuttuğu adresi `head`'e atadıktan sonra `prev` işaretçisinin değerini `NULL` yapmamız gerekir. Sonra `free()` fonksiyonuyla baştaki eleman silinebilir. Eğer ortadan bir düğüm silinmek istenirse, silinecek düğümün üzerinde durup bir önceki düğümü, silinmek istenen düğümden bir sonraki düğüme bağlamak gerekir. Silinecek düğüm Şekil 2.8'de görüldüğü gibi ortadaki düğüm olsun. `double_linked_remove` isimli fonksiyonumuzu yazarak konuyu kavrayalım.

```
void double_linked_remove(int key) {
    struct node *temp = head;

    if(head -> data == key) { // silinecek değerin ilk düğümde olması durumu.
        head = head -> next;
        head -> prev = NULL;
        free(temp);
    }
    else {
        while(temp -> data != key)
            temp = temp -> next;
        temp -> prev -> next = temp -> next;
        /* silinecek düğümden bir önceki düğümün next işaretçisi, şimdi silinecek
           düğümden bir sonraki düğümü gösteriyor. */

        if(temp -> next != NULL) // silinecek düğüm son düğüm değilse
            temp -> next -> prev = temp -> prev;
            /* silinecek düğümden bir sonraki düğümün prev işaretçisi, şimdi
               silinecek düğümden bir önceki düğümü gösteriyor. */
        free(temp);
    }
}
```

Listenin, ortada bulunan düğümü silindikten sonraki görünümünü Şekil 2.9'da görüyorsunuz.



Şekil 2.9 Silme işleminden sonra yeni listenin görünümü.

Çift Bağlı Doğrusal Listelerde Arama Yapmak

```
// head listesinde data'sı veri olan node varsa adresini alma
struct node* locate(int veri, struct node* head) {
    struct node* locate = NULL;
    while(head != NULL) {
        if(head -> data != veri) {
            head = head -> next; // aranan veri yoksa liste taranıyor
        }
        else {
            locate = head;
            break; // veri bulunursa döngüden çıkılarak geri döndürülüyor
        }
    }
   return locate;
}
```

Çift Bağlı Doğrusal Listede Karşılaştırma Yapmak

Verilen `node`'un bu listede var olup olmadığını kontrol eden fonksiyondur. Fonksiyon eğer listede `node` varsa `1`, yoksa `0` ile geri döner.

```
bool is_member(struct node* other_node, struct node* head) {
    while(head != NULL && head != other_node)
        head = head -> next;
    return(head == other_node); // ifade doğruysa 1, değilse 0 geri döndürülür.
}
```

Verilen örneklerdeki fonksiyonlar, varsayılan bir listenin yahut global tanımlanmış `head` değişkeninin varlığı, yine global olarak tanımlanmış yapıların olduğu kabul edilerek yazılmıştır. Eğer bu kabulümüz olmasaydı şüphesiz kontrol ifadelerini de içeren uzun kod satırları meydana gelirdi.

Çift Bağlı Doğrusal Listelerin Avantajları ve Dezavantajları

Avantajları

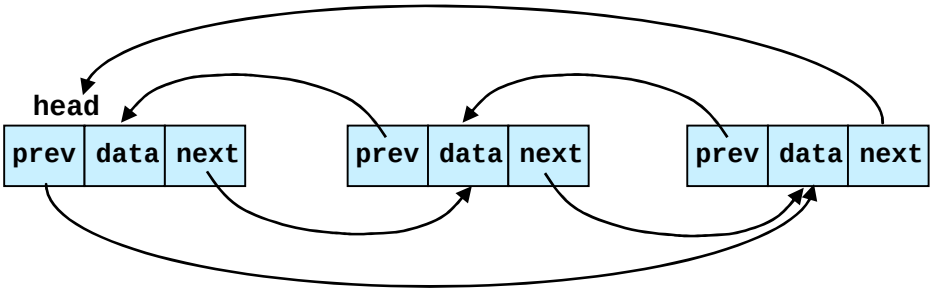
- Her iki yönde gezilebilir,
- Ekleme,Silme gibi bazı işlemler daha kolaydır.

Dezavantajları

- Bellekte daha fazla yer kaplar,
- Her düğümün `prev` ve `next` adında iki işaretçisi olduğu için liste işlemleri daha yavaştır,
- Hata olasılığı yüksektir. Örneğin düğümlerin `prev` işaretçisinin bir önceki düğüme bağlanması ya da `next` işaretçisinin bir sonraki düğüme bağlanması unutulabilir.

2.9 ÇİFT BAĞLI DAİRESEL(Double Linked) LİSTELER

İki bağlı doğrusal listelerde listenin birinci düğümünün `prev` değeri ve sonuncu düğümünün `next` değeri `NULL` olmaktaydı. Ancak iki bağlı dairesel listelerde listenin birinci düğümünün `prev` değeri sonuncu düğümünün `data` kısmını, sonuncu düğümünün `next` değeri de listenin başını (*yani listenin ismini*) göstermektedir. Fonksiyonlarda bununla beraber değişmektedir.



Şekil 2.10 Çift bağlı dairesel listeler.

Çift Bağlı Dairesel Listelerde Başa Düğüm Eklemek

Listenin başına bir düğüm ekleyen `insertAtFirst` fonksiyonunu yazalım.

```
void addhead(struct node*&head, int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
        head -> prev = head;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> data = key;
        struct node *last = head;
        // liste çift bağlı ve dairesel olduğu için son eleman head->prev dir.
        head->prev->next=temp;
        temp->next=head;
        temp->prev=head->prev;
        head->prev=temp;
        head = temp;
    }
}
```

Çift Bağlı Dairesel Listenin Sonuna Eleman Eklemek

`addhead` fonksiyonundaki son satır `head = temp;` yazılmaz ise listenin sonuna ekleme yapılmış olur.

```
void addlast(struct node* temp, struct node*&head) {
    if(!head)
        head = temp;
    else {
        temp -> next = last(head) -> next;
        temp -> prev = last(head);
        last(head) -> next = temp; // last fonksiyonu ile son düğüm bulunuyor
        head -> prev = temp;
    }
}
```

Çift Bağlı Dairesel Listelerde İki Listeyi Birleştirmek

```
// list_1 listesinin sonuna list_2 listesini eklemek
void concatenate(struct node*& list_1, struct node* list_2){ //parametrelere dikkat
    if(list_l == NULL)
        list_1 = list_2;
    else {
       // Birinci listenin son düğümünü last olarak bulmak için
        struct node *last=list_1;
        while(last -> next != list_1)
            last = last -> next;
        last->next=list_2; //Birinci listenin sonu ikinci listenin başına bağlandı
        list2->prev=last; //İkinci listenin başı birinci listenin sonuna bağlandı
        // İkinci listenin son düğümünü last olarak bulmak için
        last=list_2;
        while(last -> next != list_2)
            last = last -> next;
        last->next=list_1; //İkinci listenin sonu birinci listenin başına bağlandı
        list1->prev=last; //Birinci listenin başı ikinci listenin sonuna bağlandı
    }
}
```

Çift Bağlı Dairesel Listelerde Araya Eleman Eklemek

```
// head listesinin n. düğümünün hemen ardına other_node düğümünü ekleme
void addthen(struct node* other_node, struct node*&head, int n) {
    node* temp = head;
    int i = 1;

    while(i < n) {
        head = head -> next;
        i++;
    }

    head -> next -> prev = other_node;
    other_node -> prev = head;
    other_node -> next = head -> next;
    head -> next = other_node;
    head = temp;
}
```