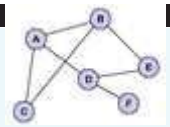


BÖLÜM

Queues (*Kuyruklar*) 4

4.1 GİRİŞ

Kuyruk (*queue*) veri yapısı `stack` veri yapısına çok benzer. Kuyruk veri yapısında da veri ekleme (*enqueue*) ve kuyruktan veri çıkarma (*dequeue*) şeklinde iki tane işlem tanımlıdır. Yığınların tersine FIFO (*First In First Out*) prensibine göre çalışırlar ve ara elemanlara erişim doğrudan yapılamaz. Veri ekleme `stack`'teki gibi listenin sonuna eklenir. Fakat veri çıkarılacağı zaman listenin son elemanı değil ilk elemanı çıkarılır. Bu ise, kuyruk veri yapısının ilk giren ilk çıkar tarzı bir veri yapısı olduğu anlamına gelir.

Kuyruğun çalışma mantığını anlatan gündelik hayatta karşılaştığımız olaylara birkaç örnek verirsek;

- sinema gişesinden bilet almak için bekleyen insanlar,
- bankamatikten para çekmek için bekleyen insanlar,

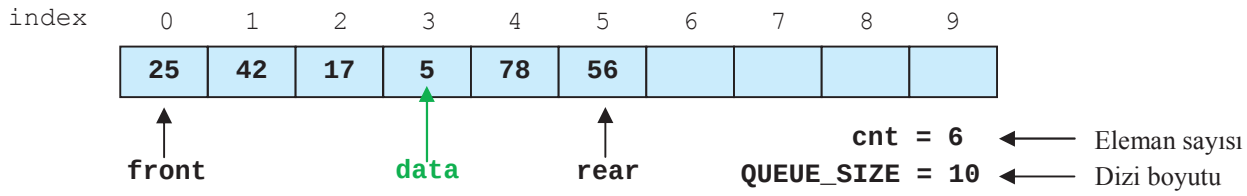
gibi, kuyruğa ilk gelen kişi ilk hizmeti alır (*ilk işlem görür*) ve kuyruktan çıkar.

Kuyruk (*queue*) yapısı bilgisayar alanında; ağ, işletim sistemleri, istatistiksel hesaplamalar, simülasyon ve çoklu ortam uygulamalarında yaygın olarak kullanılmaktadır. Örneğin, yazıcı kuyruk yapısıyla işlemleri gerçekleştirmektedir. Yazdır komutu ile verilen birden fazla belgeden ilki yazdırılmakta, diğerleri ise kuyrukta bekletilmekte, sırası geldiğinde ise yazdırılmaktadır.

Kuyruk veri yapısı bir sabit dizi olarak gösterilebilir veya bir bağlı liste olarak tanımlanabilir. Kuyrukların array implementasyonlarında, kuyruğun eleman sayısını tutan bir değişken ile kuyruğun başını gösteren (*genellikle* ***front*** *olarak isimlendirilir*) ve kuyruğun sonunu gösteren (*genellikle* ***rear*** *olarak isimlendirilir*) üç adet `int` türden değişken ile saklanacak verinin türüne uygun bir dizi bulunur. Kuyrukların bağlı liste implementasyonlarında ise kuyruğun başını ve sonunu gösteren `struct node` türden iki işaretçi ile eleman sayısını tutan `int` türden bir counter'ı bulunur. Kuyruğun başını gösteren işaretçi her veri çıkarıldığında bir sonraki veriyi gösterir. Kuyruğun sonunu gösteren işaretçi ise her veri eklendiğinde yeni gelen veriyi gösterecek şekilde değiştirilir.

Kuyruk (*queue*) yapısında da yığın yapısında olduğu gibi eleman ekleme ve eleman çıkarma olmak üzere iki işlem söz konusudur. Kuyruğa eleman eklemek için kuyruğun dolu olmaması gerekir ve ilk olarak kuyruğun dolu olup olmadığı kontrol edilir ve boş yer varsa eleman eklenir; boş yer yoksa ekleme işlemi başarısız olur. Kuyruğa eleman ekleme işlemi **ENQUEUE** olarak adlandırılır. Kuyruktan eleman çıkarmak için de kuyruğun boş olmaması gerekir. Bu çıkarma işlemi için de ilk olarak kuyruğun boş olup olmadığı kontrol edilir ve kuyruk boş değilse, eleman çıkarma işlemi gerçekleştirilir. Eğer boşsa eleman çıkarma işlemi başarısız olur. Kuyruktan eleman çıkarma işlemi **DEQUEUE** olarak bilinir.

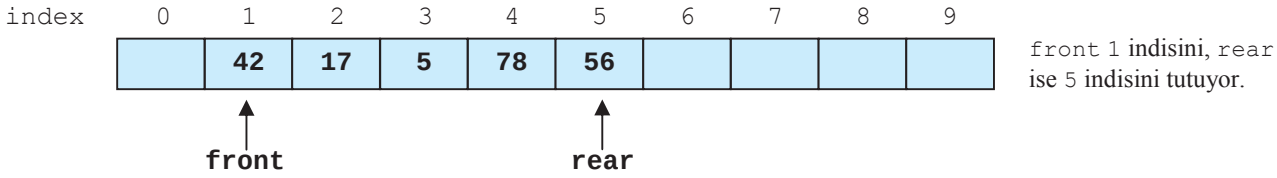
4.2 KUYRUKLARIN DİZİ (Array) İMPLEMENTASYONU



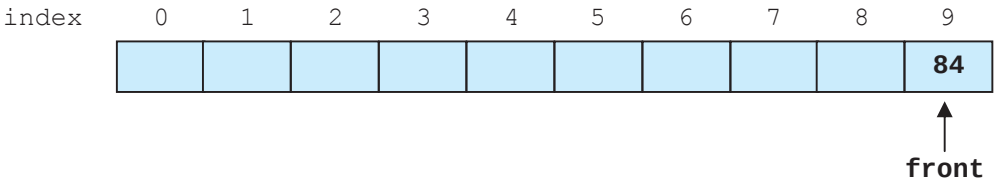
Şekil 4.1 Kuyruklarda (*queue*) array implementasyonunun mantıksal gösterimi.

Müşteri kuyruklarında öndeki müşteri beklediği hizmeti alıp kuyruktaki yerinden ayrılınca, kuyruktakiler birer adım kolayca, ön tarafa istekli biçimde yürürler. Yazılım içinde oluşturulmuş olan kuyruklarda ise bu işlem kendiliğinden ve kolay uygulanır bir nitelik değildir. Basit yapılı kuyruk denen bu modelin sakıncası, kuyruktaki öğelerin sayısı kadar kaydırma işleminin kuyruktan her ayrılma olayında yapılmasıdır. Ekleme ve çıkarma işlemlerinde ön indeks (`front`) daima sabittir. Çıkarma (*dequeue*) işleminde, kuyruğun önündeki eleman çıkarılır ve diğer bütün elemanlar bir öne kayar, ayrıca `rear` değişkeninin değeri de `1` azaltılır. Ekleme (*enqueue*) işleminde ise arka indeks (`rear`) dizide ileri doğru hareket eder. Şekil 4.1'de basit yapılı bir kuyruk modeli verilmiştir.

Sürekli kaydırma işleminin maliyeti düşünüldüğünde eleman çıkarmalarda verileri kaydırmak yerine `front` ön indeksi kaydırılsın şeklinde bir çözüm akla gelebilir. Fakat bu defa da başka bir problem ortaya çıkmaktadır. Yukarıdaki şekil dikkate alındığında kuyruklarda `enqueue` işlemi sağ taraftan, `dequeue` işlemi ise sol taraftan yapılmaktadır. Şekil 4.2'de görüldüğü gibi verileri kaydırmayıp `front` indeksini kaydıracağımıza göre kuyruğun solundan bir eleman çıkartıldığında orada boşluk oluşacak, `front` ise `1` no'lu indisi gösterecektir. Sürekli bir `dequeue` işleminde başka bir önlem alınmazsa `front` indeksi dizi sonuna kadar ulaşabilecektir. Bu durumu da Şekil 4.3'te görüyorsunuz.

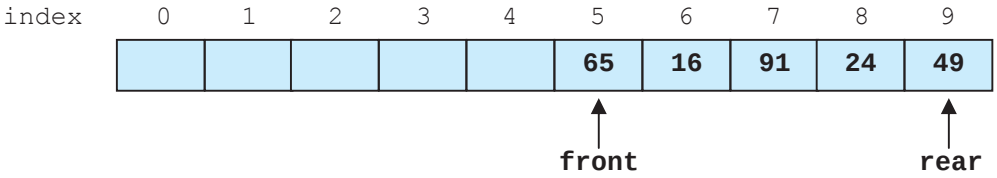


Şekil 4.2 `dequeue` işlemi sonrası kuyruğun görünümü.



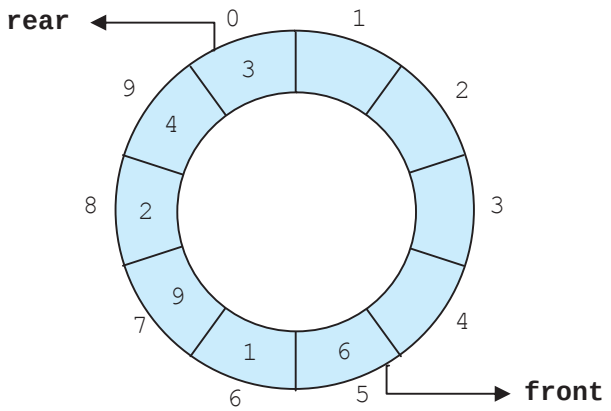
Şekil 4.3 `front` indeksi kuyruğun son indisini gösteriyor.

Şimdi Şekil 4.2'deki duruma göre birkaç defa `enqueue` ve `dequeue` işlemleri sonucunda Şekil 4.4'te görüldüğü gibi `rear` indeksinin kuyruğun son indisine kadar geldiğini kabul edelim. Eleman eklemek istediğimizde bu defa `overflow` hatasıyla karşılaşacağız demektir. Ayrıca kuyruğun solunda bulunan boşluklara da eleman eklenemeyecektir.



Şekil 4.4 Basit yapılı kuyrukta meydana gelen problem.

Böyle bir durumdan kurtulmanın yolu, kuyruğa dairesel dizi (*circular array*) efekti vermektir. Kuyruk işlemlerinde sürekli baştan ya da sondan çıkarma veya ekleme yaptığımıza göre elemanlar arasında boşluk yok demektir. Boşluklar ya baş taraftadır, ya da kuyruğun son tarafındadır. Kuyruklarda `enqueue` işleminde önce `rear` indeksi `1` arttırılır ve sonra eleman eklenir. `rear` kuyruğun son indisini gösterdiğinde ekleme yapmadan önce `1` arttırıldığına göre indeks dizi boyutuna eşit olacak ve taşma hatası meydana gelecektir. Bunun önüne geçmek için `rear` indeksi `0` olacak şekilde update edilerek elemanın kuyruğun `0` no'lu indisine eklenmesi sağlanır. Peki ama bir kuyrukta `0` no'lu indisin boş olduğunu nereden bileceğiz? Bu sorunun cevabı listeler ve yığınlarda olduğu gibi kuyruk dolu mu şeklinde bir kontrol yapmak olmalıdır. Eğer dolu değilse en azından `0` no'lu indisin boş olduğu kesin olarak bilinmektedir. Gerçekte dairesel bir dizi olmamasına karşın, böyle bir efekt verilmiş model Şekil 4.5'te gösterilmiştir.



Şekil 4.5 Dairesel dizi (*circular array*) modeli.

Şekil 4.5'te kuyruğun sonundayken dairesel efekt vermek için `rear` indeksinin değeri `0` yapılıyor ve ilk indise eleman ekleniyor. Şimdi `rear` `0`, `front` ise `5` indis değerini tutuyor. `rear` indeksinin `front` indeksinden daha önde yer

alması uygulanan efektin bir sonucudur. Artık kuyruk modeli biçimlendiğine göre array implementasyonlarındaki yapı tanımlanabilir. Önce kuyruk boyutunu belirleyen sabiti `define` bildirimiyle yazıyoruz.

```
#define QUEUE_SIZE 10

typedef struct {
    int front; // Hatırlatma, sadece dizi indislerini tutacakları için int türden
    int rear;
    int cnt;
    int data[QUEUE_SIZE];
}queue;
```

Bir Kuyruğa Başlangıç Değerlerini Vermek (`initialize`)

`queue` yapısını kullanmadan önce `initialize` etmek gerekir. Fonksiyon oldukça basittir ve yapı değişkenlerinin ilk değerlerini vermektedir.

```
void initialize(queue *q) {
    q -> front = 0;
    q -> rear = -1; // Önce arttırılacağı ve sonra ekleme yapılacağı için -1
    q -> cnt = 0;
}
```

Kuyruğun Boş Olup Olmadığının Kontrolü (isEmpty)

Yığınlarda ki fonksiyonla aynıdır. Eleman sayısını tutan `cnt` yapı değişkeni `0` ise kuyruk boştur ve geriye `true` döndürülür. Eğer `cnt` `0`'dan büyükse kuyrukta eleman vardır ve geriye `false` değeri döndürülür.

```
typedef enum {false, true}boolean;

boolean isEmpty(queue *q) {
    return(q -> cnt == 0);
}
```

Kuyruğun Dolu Olup Olmadığının Kontrolü (isFull)

Yığınlarda ki fonksiyonla aynıdır. Eleman sayısını tutan `cnt` yapı değişkeni `QUEUE_SIZE`'a eşit ise kuyruk doludur ve geriye `true` döndürülür. Eğer `cnt` `QUEUE_SIZE`'dan küçükse kuyrukta dolu değildir ve geriye `false` değeri döndürülür.

```
boolean isFull(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}
```

Kuyruğa Eleman Eklemek (enqueue)

Bir kuyruğa eleman ekleme fonksiyonu `enqueue` olarak bilinir ve bu fonksiyon, kuyruğun adresini ve eklenecek elemanın değerini parametre olarak alır. Geri dönüş değeri olmayacağı için türü `void` olmalıdır. Ayrıca eleman ekleyebilmek için kuyruğun dolu olmaması gerekir.

```
void enqueue(queue *q, int x) {
    if(!isFull(q)) {
        // kuyruk dolu mu diye kontrol ediliyor
        q -> rear ++; // ekleme öncesi rear arttırılıyor
        q -> cnt ++;

        if(q -> rear == QUEUE_SIZE)
            q -> rear = 0;
         /* Ekleme yapmadan önce rear 1 arttırılıyordu. Kuyruk dolu olmayabilir fakat
            rear indeksi dizinin son indisini gösteriyor olabilir. Böyle bir durumda
            dizide taşma hatası olmaması için rear'in değeri kontrol ediliyor. rear son
            indisi gösteriyor ve kuyruk dolu değilse rear değeri 0'a set ediliyor */
        q -> data[q -> rear] = x;
    }
}
```

Eleman eklendikten sonra `rear` değişkeni en son eklenen elemanın indis değerini tutmaktadır.

Kuyruktan Eleman Çıkarma İşlemi (dequeue)

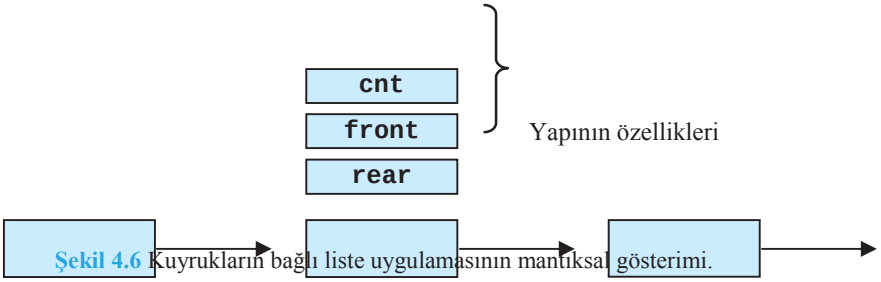
`dequeue` fonksiyonu olarak bilinir. İlk giren elemanı çıkarmak demektir. Çıkarılan elemanın veri değeriyle geri döneceği için fonksiyonun tipi `int` olmalıdır.

```
int dequeue(queue *q) {
    if(!isEmpty) { // kuyruk boş mu diye kontrol ediliyor
        int x = q -> data[q -> front]; // front değerinin saklanması gerekiyor
        q -> front++;
        q -> cnt--;

        if(q -> front == QUEUE_SIZE)
         /* front dizinin sonunu gösteriyor olabilir. Dizi taşma hatası olmaması
            için kontrol edilmesi gerekir. */
            q -> front = 0;
        return x; // çıkarılan elemanın data değeriyle çağrıldığı yere geri dönüyor
    }
}
```

4.3 KUYRUKLARIN BAĞLI LİSTE (*Linked List*) İMPLEMENTASYONU

Kuyrukların bağlı liste uygulamasında, üç adet özellikleri bulunmaktadır. Bunlardan `cnt` eleman sayısını tutan `int` türden bir yapı değişkenidir. İkincisi `struct node` türden `front` işaretçisi kuyruğun önünü göstermektedir ve üçüncüsü ise yine `struct node` türden `rear` işaretçisidir. `rear` ise kuyruğun arkasını göstermektedir. Bu uygulamada eleman ekleme işlemi `rear` üzerinden, çıkarma işlemi ise `front` üzerinden yapılır. Bunlar bir kuyrukta ekleme ve çıkartma işlemlerini kolay bir şekilde yapabilmek için tanımlanmıştır. Yine ekleme ve çıkarma işlemlerinde kolay erişim için bir de `cnt` değişkeni tanımlanmıştır.



Şekil 4.6 Kuyrukların bağlı liste uygulamasının mantıksal gösterimi.

Veri yapısı aşağıdaki gibi tanımlanmıştır.

```
typedef struct {
    int cnt;
    struct node *front;
    struct node *rear;
}queue;
```

Kuyruğa Başlangıç Değerlerini Vermek (`initialize`)

Her zaman olduğu gibi ilk önce `initialize` yapılması gerekiyor. Fonksiyon şu şekilde tanımlanabilir;

```
void initialize(queue *q) {
    q -> cnt = 0;
    q -> front = q -> rear = NULL; // NULL değeri her iki işaretçiye de atanır
}
```

`initialize` fonksiyonunda `q -> front = q -> rear = NULL;` satırındaki gibi bir işlemde `NULL` değeri ilk önce `q -> rear`'a atanır. Bu defa da `q -> front` göstericisine `q -> rear` değeri atanarak iki işaretçi de `NULL` değerini almış olur.

Kuyruğun Boş Olup Olmadığının Kontrolü (isEmpty)

Eleman sayısını tutan `cnt` yapı değişkeni `0` ise kuyruk boştur ve geriye `true` döndürülür. Eğer `cnt` `0`'dan büyükse kuyrukta eleman vardır ve geriye `false` değeri döndürülür.

```
int isEmpty(queue *q) {
    return(q -> cnt == 0);
}
```

Kuyruğun Dolu Olup Olmadığının Kontrolü (`isFull`)

Yığınlarda ki fonksiyonla aynıdır. Eleman sayısını tutan `cnt` yapı değişkeni `QUEUE_SIZE`'a eşit ise kuyruk doludur ve geriye `true` döndürülür. Eğer `cnt` `QUEUE_SIZE`'dan küçükse kuyrukta dolu değildir ve geriye `false` değeri döndürülür.

```
int isFull(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}
```

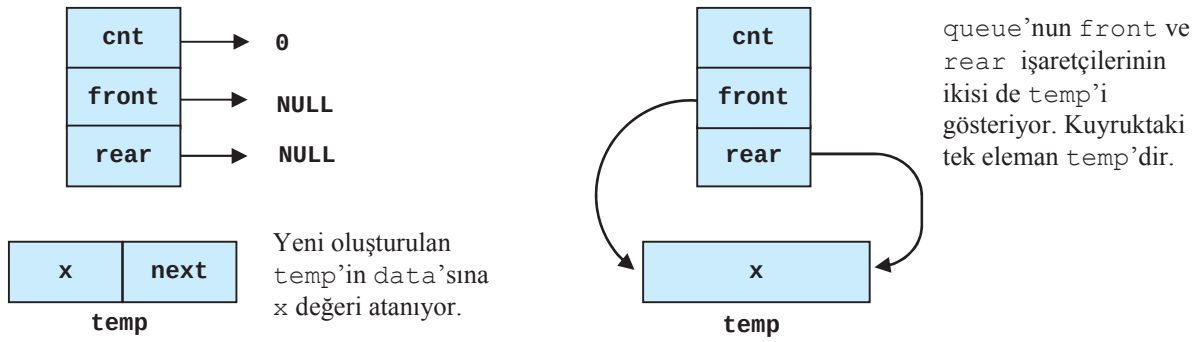
Kuyruğa Eleman Eklemek (enqueue)

Bir kuyruğa eleman ekleme fonksiyonu `enqueue` olarak bilinir ve bu fonksiyon, kuyruğun adresini ve eklenecek elemanın değerini parametre olarak alır. Geri dönüş değeri olmayacağı için türü `void` olmalıdır. Ayrıca eleman ekleyebilmek için kuyruğun dolu olmaması gerekir. `stack`'lerdeki `push` işlemi gibidir.

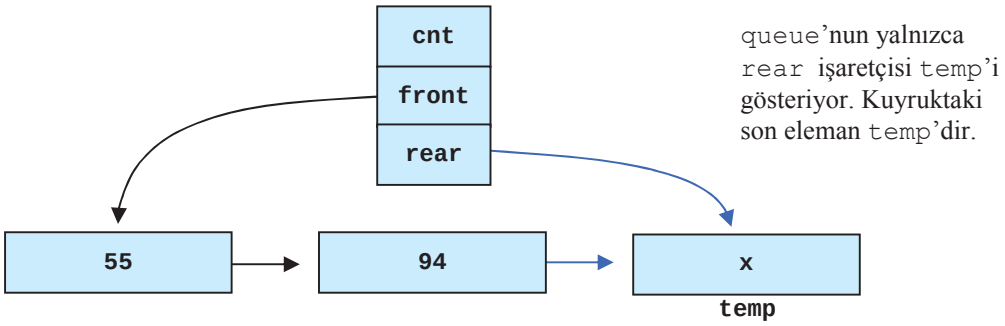
```
void enqueue(queue *q, int x) {
    if(!isFull(q)) {
        // kuyruk dolu mu diye kontrol ediliyor
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        // C++'ta struct node *temp = new node(); şeklinde
        temp -> next = NULL;
        temp -> data = x;

        if(isEmpty(q))
            q -> front = q -> rear = temp;
        else {
            q -> rear -> next = temp;
            q -> rear = temp;
        }
        q -> cnt ++;
    }
}
```

Fonksiyon içerisinde `struct node` türünden `temp` adında bir yapı daha oluşturuyoruz. Öncelikle atamaları `temp`'in elemanlarına yapıp sonra kuyruğun arkasına ekliyoruz. Kuyruk tamamen boş ise `front` ve `rear` işaretçilerinin ikisi de kuyruğa yeni eklenen düğümü gösterecektir. Çünkü ilk eleman da son eleman da eklenen bu yeni düğümdür. Eğer kuyruk boş değilse sadece arkaya ekleme yapıldığına dikkat ediniz. Kuyruğun tamamen boş olması durumu ve en az bir elemanı olması durumuna göre `enqueue` işlemini şekillerle modelleyelim.



Şekil 4.7 a) Boş bir kuyruğa eleman eklenmesi.



Şekil 4.7 b) Dolu bir kuyruğa eleman eklenmesi.

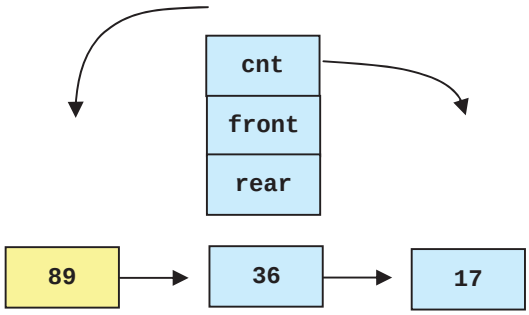
Ekleme yapmadan önce `rear` içinde `94` verisi bulunan düğümü gösteriyordu. Şimdi ise `temp`'i gösteriyor. Burada anlaşılması gereken nokta şudur; `queue` yalnızca elemanların başını, sonunu ve sayısını tutan tek bir yapıdır. Eklenen tüm veriler `struct` node türden birer düğümdür.

Kuyruktan Eleman Çıkarma İşlemi (dequeue)

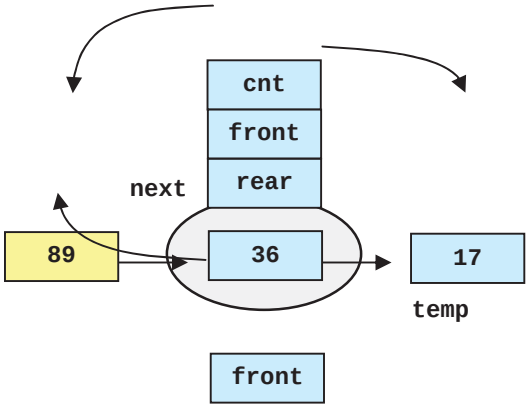
Bir kuyruktan eleman çıkarma fonksiyonu `dequeue` olarak bilinir ve fonksiyon, kuyruğun adresini parametre olarak alır. İlk giren elemanı çıkarmak demektir. Çıkarılan elemanın veri değeriyle geri döneceği için fonksiyonun tipi `int` olmalıdır. Eleman çıkartabilmek için kuyruğun boş olmaması gerekir. `stack`'lerdeki `pop` işlemi gibidir.

```
int dequeue(queue *q) {
    if(!isEmpty(q)) { // kuyruk boş mu diye kontrol ediliyor
        struct node *temp = q -> front;
        int x = temp -> data; // silmeden önce geri döndürülecek veri saklanıyor
        q -> front = temp -> next;
        free(temp); // Daha önce front'un gösterdiği adres belleğe geri veriliyor
        q -> cnt--;
        return x; // çıkarılan elemanın data değeriyle çağrıldığı yere geri dönüyor
    }
}
```

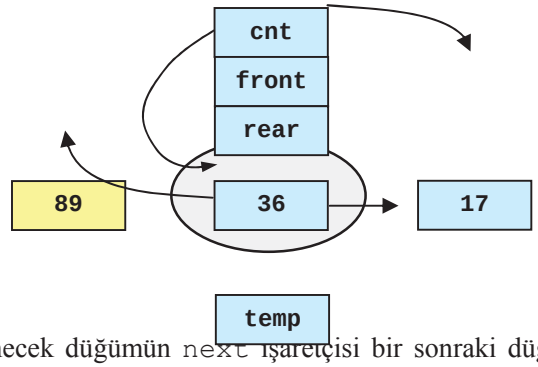
Kuyruğun `front` işaretçisinin gösterdiği adres `temp`'e atanmıştır. Şimdi aynı düğümün adresi hem `front` işaretçisinde hem de `temp` işaretçisindedir. Şekilde `dequeue` işlemi açık bir şekilde olarak gösterilmiştir.



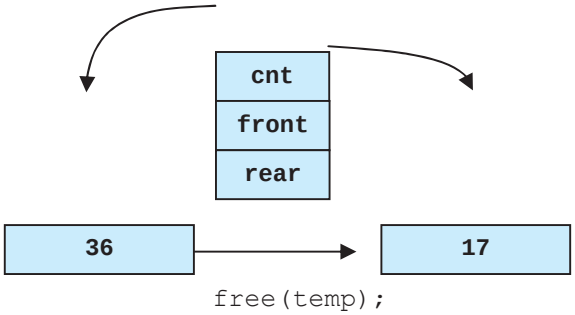
Şekil 4.8 a) Kuyruktan çıkarılacak eleman sarı renkli olarak görülüyor.



Şekil 4.8 b) Şimdi `temp` de silinecek elemanı gösteriyor.



Şekil 4.8 c) `front` bir sonraki elemanı gösteriyor.



Şekil 4.8 d) Kuyruktan eleman silme işlemi tamamlanıyor.

Silinecek düğümün `next` işaretçisi bir sonraki düğümün adresini tutmaktadır (*`front->next`, ve `temp->next` de aynı adresi tutuyor demektir*). Şu anda hem `front`'un hem de `temp`'in `data`'sı olan `89` değeri `x` değişkenine aktarılıyor. Sonra `temp->next` değeri `front` işaretçisine atanarak `36` verisinin bulunduğu düğümü göstermesi sağlanıyor. Nihayet `temp` işaretçisinin gösterdiği adres `free()` fonksiyonuyla belleğe geri kazandırılıyor ve eleman sayısını tutan `cnt` değişkeni de `1` azaltılarak `dequeue` işlemi tamamlanıyor.

Örnek 4.1: Palindrom, tersten okunuşu da aynı olan cümle, sözcük ve sayılara denilmektedir (*Ey Edip, Adana'da pide ye, 784521125487 ...vb*). Verilen bir stringin palindrom olup olmadığını belirleyen `C` kodunu, noktalama işaretleri, büyük harfler ve boşlukların ihmal edildiğini varsayarak `stack` ve `queue` yapılarıyla yazalım.
ANSI C'de;

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

```
#include <conio.h>
#include <ctype.h>
#define SIZE 100
#define STACK_SIZE 100
#define QUEUE_SIZE 100

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

typedef struct {
   struct node *top;
   int cnt;
}stack;

typedef struct {
    int cnt;
    struct node *front;
    struct node *rear;
}queue;

void initialize_stack(stack *stk) {
    stk -> top = NULL;
    stk -> cnt = 0;
}

void initialize_queue(queue *q) {
    q -> cnt = 0;
    q -> front = q -> rear = NULL;
}

typedef enum {false, true}boolean;
boolean isEmpty_stack(stack *stk) {
    return(stk -> cnt == 0);
}

boolean isFull_stack(stack *stk) {
    return(stk -> cnt == STACK_SIZE);
}

void push(stack *stk, int c) {
    if(!isFull_stack(stk)) {
        struct node *temp = (struct node *)malloc(sizeof(struct node));

        temp -> data = c;
        temp -> next = stk -> top;
        stk -> top = temp;
        stk -> cnt++;
    }
    else
         printf("Stack dolu\n");
}

int pop(stack *stk) {
    if(!isEmpty_stack(stk)) {
        struct node *temp = stk -> top;
        stk -> top = stk -> top -> next;
        int x = temp -> data;
        free(temp);
        stk -> cnt--;
        return x;
    }
}
int isEmpty_queue(queue *q) {
    return(q -> cnt == 0);
}
```

```
int isFull_queue(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}

void enqueue(queue *q, int x) {
    if(!isFull_queue(q)) {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> next = NULL;
        temp -> data = x;

        if(isEmpty_queue(q))
            q -> front = q -> rear = temp;
        else
            q -> rear = q -> rear -> next = temp;
        q -> cnt ++;
    }
}

int dequeue(queue *q) {
    if(!isEmpty_queue(q)) {
        struct node *temp = q -> front;
        int x = temp -> data;
        q -> front = temp -> next;
        free(temp);
        q -> cnt--;
        return x;
    }
}

int main() {
    char ifade[SIZE];
    queue q;
    stack s;
    int i = 0, mismatches = 0;

    initialize_stack(&s);
    initialize_queue(&q);

    printf("Bir ifade giriniz...\n");
    gets(ifade);

    for(i = 0; i != strlen(ifade); i++) {
        if(isalpha(ifade[i])) {
            enqueue(&q, tolower(ifade[i]));
            push(&s, tolower(ifade[i]));
        }
    }
    while(!isEmpty_queue(&q)) {
        if(pop(&s) != dequeue(&q)) {
            mismatches = 1;
            break;
        }
    }

    if(mismatches == 1)
        printf("Girdiginiz ifade palindrom degildir!\n");
    else
        printf("Girdiginiz ifade bir palindromdur!\n");

    getch();
    return 0;
}
```

C++'ta ise birkaç farklılık dışında bir şey yoktur.

```
#include <cstdlib>
#include <string>
#include <iostream>
```

```
#include <conio.h>
#include <ctype.h>
#define SIZE 100
#define STACK_SIZE 100
#define QUEUE_SIZE 100
using namespace std;

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

typedef struct {
   struct node *top;
   int cnt;
}stack;

typedef struct {
    int cnt;
    struct node *front, *rear;
}queue;

void initialize_stack(stack *stk) {
    stk -> top = NULL;
    stk -> cnt = 0;
}

void initialize_queue(queue *q) {
    q -> cnt = 0;
    q -> front = q -> rear = NULL;
}

bool isEmpty_stack(stack *stk) {
    return(stk -> cnt == 0);
}

bool isFull_stack(stack *stk) {
    return(stk -> cnt == STACK_SIZE);
}

void push(stack *stk, int c) {
    if(!isFull_stack(stk)) {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> data = c;
        temp -> next = stk -> top;
        stk -> top = temp;
        stk -> cnt++;
    }
    else
        printf("Stack dolu\n");
}

int pop(stack *stk) {
    if(!isEmpty_stack(stk)) {
        struct node *temp = stk -> top;
        stk -> top = stk -> top -> next;
        int x = temp -> data;
        free(temp);
        stk -> cnt--;
        return x;
    }
}
int isEmpty_queue(queue *q) {
    return(q -> cnt == 0);
}

int isFull_queue(queue *q) {
```

```
    return(q -> cnt == QUEUE_SIZE);
}

void enqueue(queue *q, int x) {
    if(!isFull_queue(q)) {
        struct node *temp = new node();
        temp -> next = NULL;
        temp -> data = x;

        if(isEmpty_queue(q))
            q -> front = q -> rear = temp;
        else
            q -> rear = q -> rear -> next = temp;
        q -> cnt ++;
    }
}

int dequeue(queue *q) {
    if(!isEmpty_queue(q)) {
        struct node *temp = q -> front;
        int x = temp -> data;
        q -> front = temp -> next;
        free(temp);
        q -> cnt--;
        return x;
    }
}

int main() {
    string the_string;
    queue q;
    stack s;

    initialize_stack(&s);
    initialize_queue(&q);
    int i = 0, mismatches = 0;

    cout << "Bir ifade giriniz" << endl;
    cin >> the_string;

    while(i < the_string.length()) {
        if(isalpha(the_string[i])) {
            enqueue(&q, tolower(the_string[i]));
            push(&s, tolower(the_string[i]));
        }
        i++;
    }
    while(!isEmpty_queue(&q)) {
        if(pop(&s) != dequeue(&q)) {
            mismatches = 1;
            break;
        }
    }
    if(mismatches == 1)
        cout << "Girdiginiz ifade palindrom degildir!" << endl;
    else
        cout << "Girdiginiz ifade bir palindromdur!" << endl;
    getch();
    return EXIT_SUCCESS;
}
```