inzva Algorithm Programme 2018-2019

Bundle 6

Veri Yapıları- 1

Editor
Tahsin Enes Kuru

Reviewers
Baha Eren Yaldız
Burak Buğrul

Contents

1 Giriş

Bilgisayar biliminde veri yapıları, belirli bir eleman kümesi üzerinde verimli bir şeklide bilgi edinmemize aynı zamanda bu elemanlar üzerinde değişiklikler yapabilmemize olanak sağlayan yapılardır. Çalışma prensipleri genellikle elemanların değerlerini belirli bir kurala göre saklamak daha sonra bu yapıları kullanarak elemanlar hakkında sorulara (Bir dizinin belirli bir aralığındaki en küçük sayı gibi) cevap aramaktır.

2 Dinamik Veri Yapıları

2.1 Linked List

Linked List veri yapısında elemanlar, her eleman kendi değerini ve bir sonraki elemanın adresini tutacak şekilde saklanır. Yapıdaki elemanlar baş elemandan(head) başlanarak son elemana(tail) gidecek şekilde gezilebilir. Diziye karşın avantajı hafızanın dinamik bir şekilde kullanılmasıdır. Bu veri yapısında uygulanabilecek işlemler:

- Veri yapısının sonuna eleman ekleme.
- Anlık veri yapısını baştan(head) sona(tail) gezme.

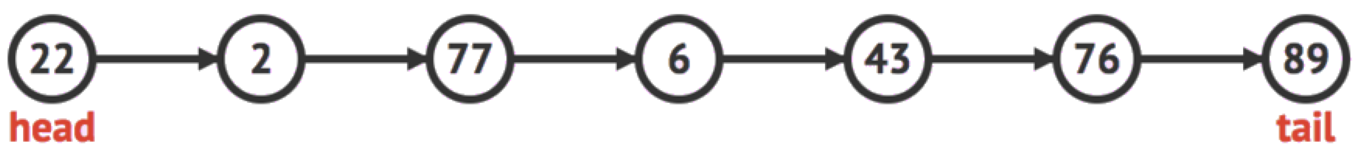

Figure 1: Örnek bir Linked List yapısı

```
// Her bir elemani tutacak struct olusturuyoruz.
struct node
{
  int data;
  node *next;
};
node *head, *tail;

void push_back(int x) {
  // Yeni elemanimizi hafizada olusturuyoruz.
  node *t = (node*)malloc(sizeof(node));
  t -> data = x; // Elemanin verisini atiyoruz.
  t -> next = NULL; // Sona ekledigimizden sonraki elemanina NULL atiyoruz.

  // Eger veri yapimiza hic eleman eklenmediyse head
  // ve tail elemanlarini olusturuyoruz.
  if(head == NULL && tail == NULL) {
    head = t;
    tail = t;
  }
  // Eklenmisse yeni tail elemanimizi guncelliyoruz.
  else {
    tail -> next = t;
    tail = t;
  }
}

void print() {
  // Dizideki tum elemanlari geziyoruz.
  node *t = head;
  while(t != NULL) {
    printf("%d ", t -> data);
    t = t -> next;
  }
}
```

2.2 Stack

Stack veri yapısında elemanlar yapıya son giren ilk çıkar (LIFO) kuralına uygun olacak şekilde saklanır. Bu veri yapısında uygulayabildiğimiz işlemler:

- Veri yapısının en üstüne eleman ekleme.
- Veri yapısının en üstündeki elemana erişim.
- Veri yapısının en üstündeki elemanı silme.

c++ dilindeki stl kütüphanesinde bulunan hazır stack yapısının kullanımı aşağıdaki gibidir.

```
int main() {
  stack < int > st;
  st.push(5); // Stack'in en ustune 5'i ekler. Stack'in yeni hali: {5}
  st.push(7); // Stack'in en ustune 7'yi ekler. Stack'in yeni hali: {7,5}
  st.push(6); // Stack'in en ustune 6'yiekler. Stack'in yeni hali : {6, 7, 5}
  st.pop(); //Stack'in en ustundeki elemani siler. Stack'in yeni hali : {7, 5}
  st.push(1); // Stack'in en ustune 1'i ekler. Stack'in yeni hali : {1 7, 5}
  cout « st.top() « endl; // Stack'in en ustundeki elemana erisir. Ekrana 1 yazirir.
}
```

2.3 Queue

Queue veri yapısında elemanlar yapıya ilk giren ilk çıkar (FIFO) kuralına uygun olacak şekilde saklanır. Bu veri yapısında uygulayabildigimiz işlemler:

- Veri yapısının en üstüne eleman ekleme.
- Veri yapısının en altındaki elemanına erişim.
- Veri yapısının en altındaki elemanı silme.

c++ dilindeki stl kütüphanesinde bulunan hazır queue yapısının kullanımı aşağıdaki gibidir.

```
int main() {
  queue < int > q;
  q.push(5); // queue'in en ustune 5'i ekler. queue'in yeni hali: {5}
  q.push(7); // queue'in en ustune 7'yi ekler. queue'in yeni hali: {7,5}
  q.push(6); // queue'in en ustune 6'yi ekler. queue'in yeni hali : {6, 7, 5}
  q.pop(); //queue'in en altindaki elemani siler. queue'in yeni hali : {6,7}
  q.push(1); // queue'in en ustune 1'i ekler. queue'in yeni hali : {1,6,7}
  cout « q.front() « endl; // queue'in en ustundeki elemana erisir. Ekrana 7 yazirir.
}
```

2.4 Deque

Deque veri yapısı stack ve queue veri yapılarına göre daha kapsamlıdır. Bu veri yapısında yapının en üstüne eleman eklenebilirken aynı zamanda en altına da eklenebilir. Aynı şekilde yapının hem en üstündeki elemanına hem de en alttaki elemanına erişim ve silme işlemleri uygulanabilir. Bu veri yapısında uyguluyabildiğimiz işlemler:

- Veri yapısının en üstüne eleman ekleme.
- Veri yapısının en altına eleman ekleme.
- Veri yapısının en üstündeki elemanına erişim.

- Veri yapısının en altındaki elemanına erişim.
- Veri yapısının en üstündeki elemanı silme.
- Veri yapısının en altındaki elemanı silme.

c++ dilindeki stl kütüphanesinde bulunan hazır dequeu yapısının kullanımı aşağıdaki gibidir.

```
int main() {
  deque < int > q;
  q.push_front(5); // deque'nin en altina 5'i ekler.
  q.push_back(6); // deque'nin en ustune 6'yi ekler.
  int x = q.front(); // deque'nin en altindaki elemanina erisim.
  int y = q.back(); // deque'nin en ustundeki elemanina erisim.
  q.pop_front(); // deque'nin en altindaki elemanini silme.
  q.pop_back(); // deque'nin en ustundeki elemanini silme.
}
```

3 Prefix Sum

Prefix Sum dizisi bir dizinin prefixlerinin toplamlarıyla oluşturulan bir veri yapısıdır. Prefix sum dizisinin i indeksli elemanı girdi dizisindeki 1 indeksli elemandan i indeksli elemana kadar olan elemanların toplamına eşit olacak şekilde kurulur. Başka bir değişle:

$$sum_i = \sum_{j=1}^{i} a_j$$

Örnek bir A dizisi için prefix sum dizisi şu şekilde kurulmalıdır:

A Dizisi	4	6	3	12	1
Prefix Sum Dizisi	4	10	13	25	26
	4	4 + 6	4 + 6 + 3	4 + 6 + 3 + 12	4 + 6 + 3 + 12 + 1

Prefix sum dizisini kullanarak herhangi bir $[l, r]$ aralığındaki elemanların toplamını şu şekilde kolaylıkla elde edebiliriz:

$$sum_r = \sum_{j=1}^{r} a_j$$

$$sum_{l-1} = \sum_{j=1}^{l-1} a_j$$

$$sum_r - sum_{l-1} = \sum_{j=l}^{r} a_j$$

3.1 Örnek Kod Parçaları

$sum_i = sum_{i-1} + a_i$ eşitliği kolayca görülür. Ve bu eşitliği kullanarak Prefix Sum dizisini girdi dizisindeki elemanları sırayla gezerek kurabiliriz.

```
int n,sum[N],a[N];
// a dizisi girdi dizimiz, sum dizisi de prefix sum dizimiz olsun.

void build() {
  for (int i = 1 ; i <= n ; i++)
    sum[i] = sum[i - 1] + a[i];
  return;
}

int query(int l,int r) {
  return sum[r] - sum[l - 1];
}
```

3.2 Zaman Karmaşıklığı

Prefix sum dizisini kurma işlemimizin zaman ve hafıza karmaşıklığı $O(N)$. Her sorguya da $O(1)$ karmaşıklıkta cevap verebiliyoruz.

Prefix sum veri yapısı ile ilgili problem: Link.

4 Sparse Table

Sparse Table aralıklardaki elemanların toplamı, minimumu, maksimumu ve ebobları gibi sorgulara $O(logN)$ zaman karmaşıklığında cevap alabilmemizi sağlayan bir veri yapısıdır. Bazı tip sorgular (aralıktaki minimum, maksimum sayıyı bulma gibi) ise $O(1)$ zaman karmaşıklığında yapmaya uygundur.

Bu veri yapısı durumu değişmeyen, sabit bir veri üzerinde ön işlemler yaparak kurulur. Dinamik veriler için kullanışlı değildir. Veri üzerinde herhangi bir değişiklik durumda Sparse Table tekrardan kurulmalıdır. Bu da maliyetli bir durumdur.

4.1 Yapısı ve Kuruluşu

Sparse table iki bouyutlu bir dizi şeklinde, $O(NlogN)$ hafıza karmaşıklığına sahip bir veri yapısıdır. Dizinin her elemanından 2'nin kuvvetleri uzaklıktaki elemanlara kadar olan cevaplar Sparse tableda saklanır. $ST_{x,i}$, x indeksli elemandan $x + 2^i$ indeksli elemana kadar olan aralığın cevabını saklayacak şekilde sparse table kurulur.

```
//Toplam sorgusu icin kurulmus Sparse Table Yapisi
void build() {

  for (int i = 1 ; i <= n ; i++) {
    // [i,i] araliginin cevabi dizinin i indeksli elemanina esittir.
    ST[i][0] = a[i];
  }

  for (int i = 1 ; i <= LOG ; i++)
    for (int j = 1 ; j <= n ; j++) {
      // [i,i+2^(j)] araliginin cevabi
      // [i,i+2^(j - 1) - 1] araligi ile [i+2^(j - 1),i+2^j] araliginin
      // cevaplarinin birlesmesiyle elde edilir
      ST[i][j] = ST[i][j - 1] + ST[i + (1 « (j - 1))][j - 1];
    }

  return;
}
```

4.2 Sorgu Algoritması

Herhangi bir $[l, r]$ aralığı için sorgu algoritması sırasıyla şu şekilde çalışır:

- $[l, r]$ aralığını cevaplarını önceden hesapladığımız aralıklara parçala. (Sadece 2'nin kuvveti uzunluğunda parçaların cevaplarını sakladığımız için aralığımızı 2'nin kuvveti uzunluğunda aralıklara ayırmalıyız. $[l, r]$ aralığının uzunluğunun iklik tabanda yazdığımızda hangi aralıklara parçalamamız gerektiğini bulmuş oluruz.)

- Bu aralıklardan gelen cevapları birleştirerek $[l, r]$ aralığının cevabını hesapla.

Herhangi bir aralığın uzunluğunun ikilik tabandaki yazılışındaki 1 rakamlarının sayısı en fazla $logN$ olabileceğinden parçalayacağımız aralık sayısı da en fazla $logN$ olur. Dolayısıyla sorgu işlemimiz $O(logN)$ zaman karmaşıklığında çalışır.

Örneğin: $[4, 17]$ aralığının cevabını hesaplamak için algoritmamız $[4, 17]$ aralığını $[4, 11]$,$[12, 15]$ ve $[16, 17]$ aralıklarına ayırır ve bu 3 aralıktan gelen cevapları birleştirerek istenilen cevabı hesaplar.

```
//toplam sorgusu

int query(int l,int r) {

  int res = 0;

  for (int i = LOG ; i >= 0 ; i-) {
    // her seferinde uzunlugu r - l + 1 gecmeyecek
    // en buyuk araligin cevabi ekleyip l'i o araligin sonuna cekiyoruz.
    if (l + (1 « i) <= r) {
      res += ST[l][i];
      l += (1 « i);
    }
  }

  return res;

}
```

4.3 Minimum ve Maksimum Sorgu

Sparse Table veri yapısının diğer veri yapılarından farklı olarak $O(1)$ zaman karmaşıklığında aralıklarda minimum veya maksimum sorgusu yapabilmesi en avantajlı özelliğidir.

Herhangi bir aralığın cevabını hesaplarken bu aralıktaki herhangi bir elemanı birden fazla kez değerlendirmemiz cevabı etkilemez. Bu durum aralığımızı 2'nin kuvveti uzunluğunda maksimum 2 adet aralığa bölebilmemize ve bu aralıkların cevaplarını $O(1)$ zaman karmaşıklığında birleştirebilmemize olanak sağlar.

```cpp
int RMQ(int l,int r) {
  // log dizisinde her sayinin log2 degerleri sakldir.
  int j = log[r - l + 1];
  return min(ST[l][j], ST[r - (1 << j) + 1][j]);
}
```

Sparse Table veri yapısı ile ilgili problem: Link.

5 Binary Indexed Tree

Fenwick tree olarak da bilinen Binary Indexed Tree, Prefix sum ve Sparse Table yapılarına benzer bir yapıda olup dizi üzerinde değişiklik yapabilmemize olanak sağlayan bir veri yapısıdır. Fenwick Tree'nin diğer veri yapılarına göre en büyük avantajı pratikte daha hızlı olması ve hafıza karmaşıklığının $O(N)$ olmasıdır. Ancak Fenwick Tree'de sadece prefix cevapları saklayabildiğimizden aralıklarda minimum, maksimum ve en büyük ortak bölen gibi bazı sorguların cevaplarını elde edemeyiz.

5.1 Yapısı ve Kuruluşu

$g(x)$, x sayısının bit gösteriminde yalnızca en sağdaki bitin 1 olduğu tamsayı olsun. Örneğin 20'nin bit gösterimi $(10100)_2$ olduğundan $g(20) = 4$'tür. Çünkü ilk kez sağdan 3. bit 1'dir ve $(00100)_2 = 4$'tür. Fenwick Tree'nin x indeksli düğümünde, $x - g(x) + 1$ indeksli elemandan x indeksli elemana kadar olan aralığın cevabını saklayacak şeklide kurulur.

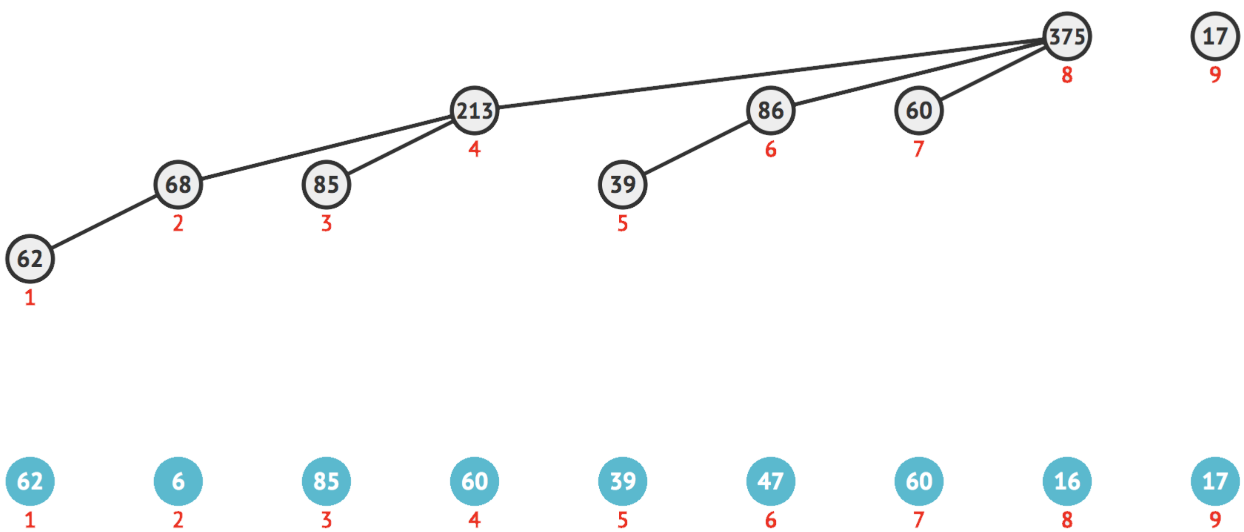

Figure 2: $a = [62, 6, 85, 60, 39, 47, 60, 16, 17]$ dizisinde toplam sorgusu için kurulmuş Fenwick Tree yapısı

5.2 Sorgu Algoritması

Herhangi bir $[1, x]$ aralığı için sorgu algoritması sırası ile şu şeklide çalışır.

1. Aradığımız cevaba $[x - g(x) + 1, x]$ aralığının cevabını ekle.
2. x'in değerini $x - g(x)$ yap. Eğer x'in yeni değeri 0'dan büyük ise 1.işlemden hesaplamaya devam et.

$[1, x]$ aralığının cevabını hesaplamak için yapılan işlem sayısı x sayısının 2'lik tabandaki yazılışındaki 1 sayısına eşittir. Çünkü her döngüde x'den 2'lik tabandaki yazılışındaki en sağdaki 1 bitini çıkartıyoruz. Dolayısıyla sorgu işlemimiz $O(logN)$ zaman karmaşıklığında çalışır. $[l, r]$ aralığının cevabını da $[1, r]$ aralığının cevabından $[1, l - 1]$ aralığının cevabını çıkararak kolay bir şekilde elde edebiliriz.

NOT: $g(x)$ değerini bitwise operatörlerini kullanarak aşağıdaki eşitlikle kolay bir şekilde hesaplayabiliriz:

$$g(x) = x\&(-x)$$

5.3 Eleman Güncelleme Algoritması

Dizideki x indeksli elemanının değerini güncellemek için kullanılan algoritma şu şeklide çalışır.

- Ağaçta x indeksli elemanı içeren tüm düğümlerin değerlerini güncelle.

Fenwick Tree'de x indeksli elemanı içeren maksimum $logN$ tane aralık olduğundan güncelleme algoritması $O(logN)$ zaman karmaşıklığında çalışır.

5.4 Örnek Kod Parçaları

```
int n,tree[N],a[N];

void add(int val,int x) { // x indeksli elemanin degerini val degeri kadar artirir.
  while(x <= n) {
    tree[x] += val;
    x += x & (-x);
  }
  return;
}

int sum(int x) { // 1 indeksli elemandan x indeksli elemana
  int res = 0; // kadar olan sayilarin toplamini verir.
  while(x >= 1) {
    res += tree[x];
    x -= x & (-x);
  }
  return res;
}

int query(int l,int r) { // [l,r] araligindaki elemanlarin toplamini verir.
  return sum(r) - sum(l - 1);
}

void build() { // a dizisi uzerine fenwick tree yapisini kuruyoruz.
  for (int i = 1 ; i <= n ; i++)
    add(a[i],i);
  return;
}
```

5.5 Aralık Güncelleme ve Eleman Sorgu

Bir a dizisi üzerinde işlemler yapacağımızı varsayalım daha sonra a dizisi b dizisinin prefix sum dizisi olacak şekilde bir b dizisi tanımlayalım. Başka bir değişle $a_i = \sum_{j=1}^{i} b_j$ olmalıdır. Sonradan oluşturduğumuz b dizisi üzerine fenwick tree yapısını kuralım. $[l, r]$ aralığındaki her elemana x değerini eklememiz için uygulamamız gereken işlemler:

- b_l değerini x kadar artır. Böylelikle l indeksli elemandan dizinin sonuna kadar tüm elemanların değeri x kadar artmış olur.

- b_{r+1} değerini x kadar azalt. Böylelikle $r + 1$ indeksli elemandan dizinin sonuna kadar tüm elemanların değeri x kadar azalmış olur. Bu işlemelerin sonucunda sadece $[l, r]$ aralığındaki elemanların değeri x kadar artmış olur.

5.5.1 Örnek Kod Parçaları

```
int n,a[N],b[N];

void add(int val,int x) { // x indeksli elemanin degerini val degeri kadar artirir.
  while(x <= n) {
    tree[x] += val;
    x += x & (-x);
  }
  return;
}

int sum(int x) { // 1 indeksli elemandan x indeksli elemana
  int res = 0; // kadar olan sayilarin toplamini verir.
  while(x >= 1) {
    res += tree[x];
    x -= x & (-x);
  }
  return res;
}
void build() {
  for (int i = 1 ; i <= n ; i++)
    b[i] = a[i] - a[i - 1]; // b dizisini olusturuyoruz.

  for (int i = 1 ; i <= n ; i++)
    add(b[i],i); // b dizisi uzerine fenwick tree kuruyoruz.
}

void update(int l,int r,int x) {
  add(x,l);
  add(-x,r + 1);
}

void query(int x) {
  return sum(x);
}
```

Fenwick Tree veri yapısı ile ilgili problem: Link.

6 SQRT Decomposition

Square Root Decomposition algoritması dizi üzerinde $O(\sqrt{N})$ zaman karmaşıklığında sorgu yapabilmemize ve $O(1)$ zaman karmaşıklığında ise değişiklik yapabilmemize olanak sağlayan bir veri yapsıdır.

6.1 Yapısı ve Kuruluşu

Dizinin elemanları her biri yaklaşık $O(\sqrt{N})$ uzunluğunda bloklar halinde parçalanır. Her bir bloğun cevabı ayrı ayrı hesaplanır ve bir dizide saklanır.

Blokların Cevapları	21				13				50				32			
Dizideki Elemanlar	3	6	2	10	3	1	4	5	2	7	37	4	11	6	8	7
Elemanların İndeksleri	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Örnek bir dizi üzerinde toplam sorgusu için kurulmuş SQRT Decompostion veri yapısı.

```
void build() {
  for (int i = 1 ; i <= n ; i++) {
    if (i % sq == 1) { // sq = sqrt(n)
      t++; //yeni blok baslangici.
      st[t] = i; // t.blok i indisli elemanda baslar.
    }
    fn[t] = i; // t.blogun bitisini i indisli eleman olarak guncelliyoruz.
    wh[i] = t; // i indeksli eleman t.blogun icinde.
    sum[t] += a[i]; // t. blogun cevabina i indeksli elemani ekliyoruz.
  }
}
```

6.2 Sorgu Algoritması

Herhangi bir $[l, r]$ aralığı için sorgu algoritması sırası ile şu şekilde çalışır.

1. Cevabını aradığımız aralığın tamamen kapladığı blokların cevabını cevabımıza ekliyoruz.

2. Tamamen kaplamadığı bloklardaki aralığımızın içinde olan elemanları tek tek gezerek cevabımıza ekliyoruz.

Cevabını aradığımız aralığın kapsadığı blok sayısı en fazla $\sqrt{N}$ olabileceğinden 1. işlem en fazla $\sqrt{N}$ kez çalışır. Tamamen kaplamadığı ancak bazı elemanları içeren en fazla 2 adet blok olabilir. (Biri en solda diğeri en sağda olacak şekilde). Bu 2 blok için de gezmemiz gereken eleman sayısı maksimum $2\sqrt{N}$ olduğundan bir sorgu işleminde en fazla $3\sqrt{N}$ işlem yapılır dolayısıyla sorgu işlemimiz $O(\sqrt{N})$ zaman karmaşıklığında calışır.

Blokların Cevapları	21				13				50				32			
Dizideki Elemanlar	3	6	2	10	3	1	4	5	2	7	37	4	11	6	8	7
Elemanların İndeksleri	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Örnek dizideki $[3, 13]$ aralığının cevabını 2. ve 3. blokların cevapları ile $3, 4$ ve 11 indeksli elemanların toplanmasıyla elde edilir.

```cpp
// [l,r] araligindaki elemanlarin toplamini hesaplayan fonksiyon.
int query(int l,int r) {

  int res = 0;

  if (wh[l] == wh[r]) { // l ve r ayni blogun icindeyse
    for (int i = l ; i <= r ; i++)
      res += a[i];
  }

  else {
    for (int i = wh[l] + 1 ; i <= wh[r] - 1 ; i++)
      res += sum[i]; // tamamen kapladigimiz bloklarin cevaplarini ekliyoruz.

    // tamamen kaplamadigimiz bloklardaki araligimiz icindeki
    // elemanlarin cevaplarini ekliyoruz.

    for (int i = st[wh[l]] ; i <= fn[wh[l]]; i++)
      if (i >= l && i <= r)
        res += a[i];

    for (int i = st[wh[r]] ; i <= fn[wh[r]] ; i++)
      if (i >= l && i <= r)
        res += a[i];
  }

  return res;
}
```

6.3 Eleman Güncelleme Algoritması

Herhangi bir elemanın değerini güncellerken o elemanı içeren bloğun değerini güncellememiz yeterli olacaktır. Dolayısıyla güncelleme işlemimimiz $O(1)$ zaman karmaşıklığında çalışır.

```
void update(int x,int val) {
  // x indeksli elemanin yeni degerini val degerine esitliyoruz.
  sum[wh[x]] -= a[x];
  a[x] = val;
  sum[wh[x]] += a[x];
}
```

SQRT Decomposition veri yapısı ile ilgili problem: Link.

7 Segment Tree

Segment Tree bir dizide $O(logN)$ zaman karmaşıklığında herhangi bir $[l, r]$ aralığı icin minimum, maksimum, toplam gibi sorgulara cevap verebilmemize ve bu aralıklar üzerinde güncelleme yapabilmemize olanak sağlayan bir veri yapısıdır.

Segment Tree'nin, Fenwick Tree ve Sparse Table yapılarından farklı olarak elemanlar üzerinde güncelleme yapılabilmesi ve minimum, maksimum gibi sorgulara da olanak sağlaması yönünden daha kullanışlıdır. Ayrıca segment tree $O(N)$ hafıza karmaşıklığına sahipken Sparse Table yapsınında gereken hafıza karmaşıklığı $O(NlogN)$'dir.

7.1 Yapısı ve Kuruluşu

Segment Tree, "Complete Binary Tree" yapısına sahiptir. Segment Tree'nin yaprak düğümlerinde dizinin elemanları saklıdır ve bu düğümlerin atası olan her düğüm kendi çocuğu olan düğümlerinin cevaplarının birleşmesiyle oluşur. Bu sayede her düğümde belirli aralıkların cevapları ve root düğümünde tüm dizinin cevabı saklanır. Örneğin toplam sorgusu için kurulmuş bir segment tree yapısı için her düğümün değeri çocuklarının değerleri toplamına eşittir.

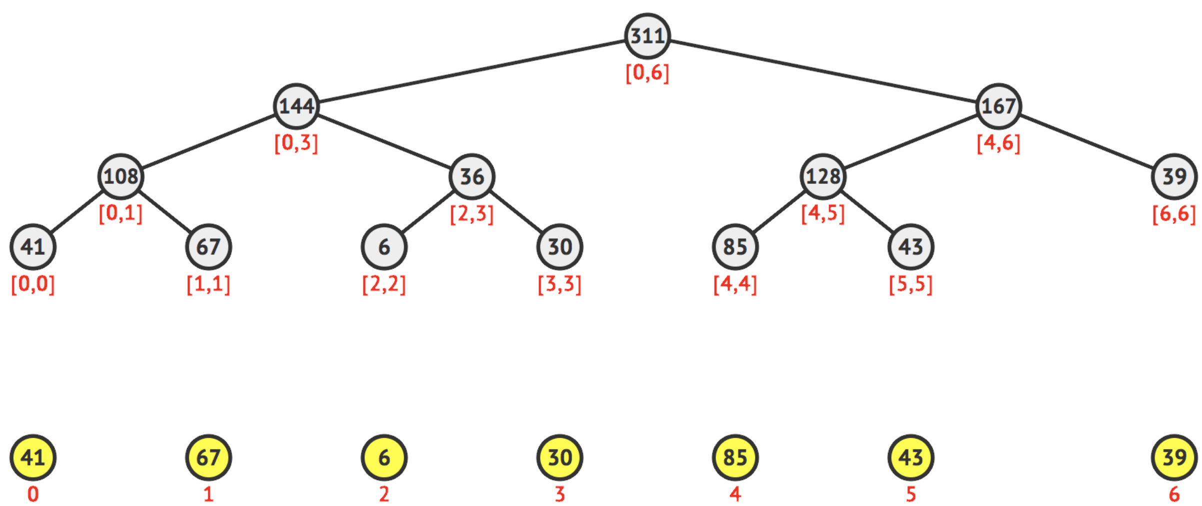

Figure 3: $a = [41, 67, 6, 30, 85, 43, 39]$ dizisinde toplam sorgusu icin oluşturulmuş segment tree yapısı

```
void build(int ind,int l,int r) {
  // tree[ind] dizinin [l,r] araliginin cevabini saklar.

  if (l == r) { // yaprak dugum'e ulasti
    tree[ind] = a[l]; //bu dugum dizinin l. elamaninin cevabini saklar
  }
  else {
    int mid = (l + r) / 2;
    build(ind * 2,l,mid);
    build(ind * 2 + 1,mid + 1,r);
    // [l,r] araliginin cevabini
    // [l,mid] ve [mid + 1,r] araliklarinin cevaplarinin birlesmesiyle olusur.
    tree[ind] = tree[ind * 2] + tree[ind * 2 + 1];
  }
  return;
}
```

7.2 Aralık Sorgu ve Eleman Güncelleme

7.2.1 Sorgu Algoritması

Herhangi bir $[l, r]$ aralığı için sorgu algoritması sırası ile şu şekilde çalışır:

- $[l, r]$ aralığını ağacımızda cevapları saklı olan en geniş aralıklara parçala.
- Bu aralıkların cevaplarını birleştirerek istenilen cevabı hesapla.

Ağacın her derinliğinde cevabımız için gerekli aralıklardan maksimum 2 adet bulunabilir. Bu yüzden sorgu algoritması $O(logN)$ zaman karmaşıklığında çalışır.

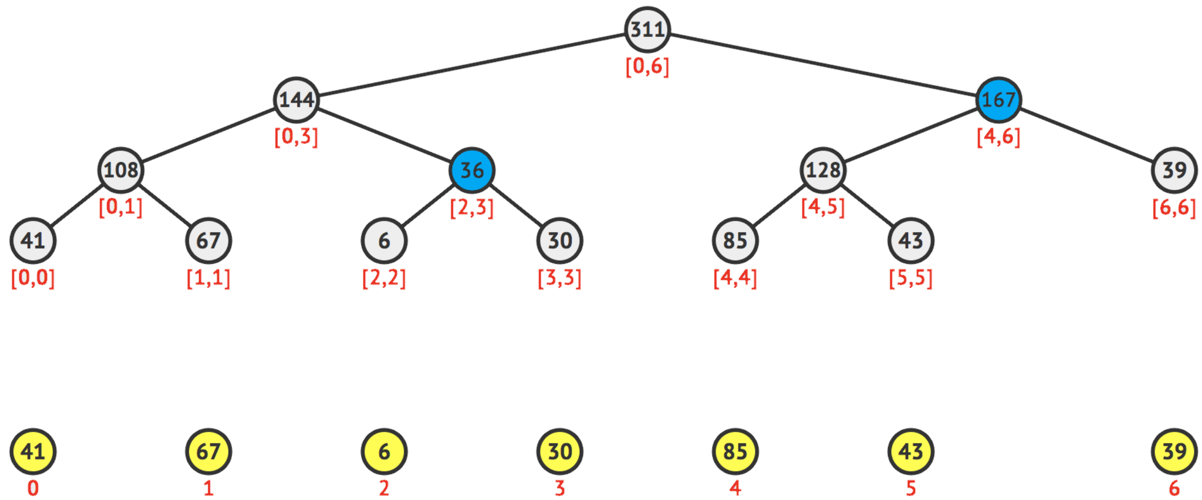

Figure 4: $a = [41, 67, 6, 30, 85, 43, 39]$ dizisinde $[2, 6]$ aralığında sorgu işlemi

$a = [41, 67, 6, 30, 85, 43, 39]$ dizisinde [2,6] aralığının cevabı $[2, 3]$ ile $[4, 6]$ aralıklarının cevaplarının birleşmesiyle elde edilir. Toplam sorgusu için cevap $36 + 167 = 203$ şeklinde hesaplanır.

```
// [lw,rw] sorguda cevabini aradigimiz aralik.
// [l,r] ise agactaki ind nolu node'da cevabini sakladigimiz aralik.

int query(int ind,int l,int r,int lw,int rw) {
  if (l > rw or r < lw) //bulundugumuz aralik cevabini aradigimiz araligin disinda.
    return 0;
  if (l >= lw and r <= rw) //cevabini aradigimiz aralik bu araligi tamamen kapsiyor.
    return tree[ind];
  int mid = (l + r) / 2;
  //Agacta recursive birseklide araligimizi
  // araliklara bolup gelen cevaplari birlestiyoruz.
  return query(ind * 2,l,mid,lw,rw) + query(ind * 2 + 1,mid + 1,r,lw,rw);
}
```

7.2.2 Eleman Güncelleme Algoritması

Dizideki x indeksli elemanının değerini güncellemek için kullanılan algoritma şu şeklide çalışır.

- Ağaçta x indeksli elemanı içeren tum düğümlerin değerlerini güncelle.

Ağaçta x indeksli elemanın cevabını tutan yaprak düğümden root düğüme kadar toplamda $logN$ düğümün değerini güncellememiz yeterlidir. Dolayısıyla herhangi bir elemanın değerini güncellemenin zaman karmaşıklığı $O(logN)$'dir.

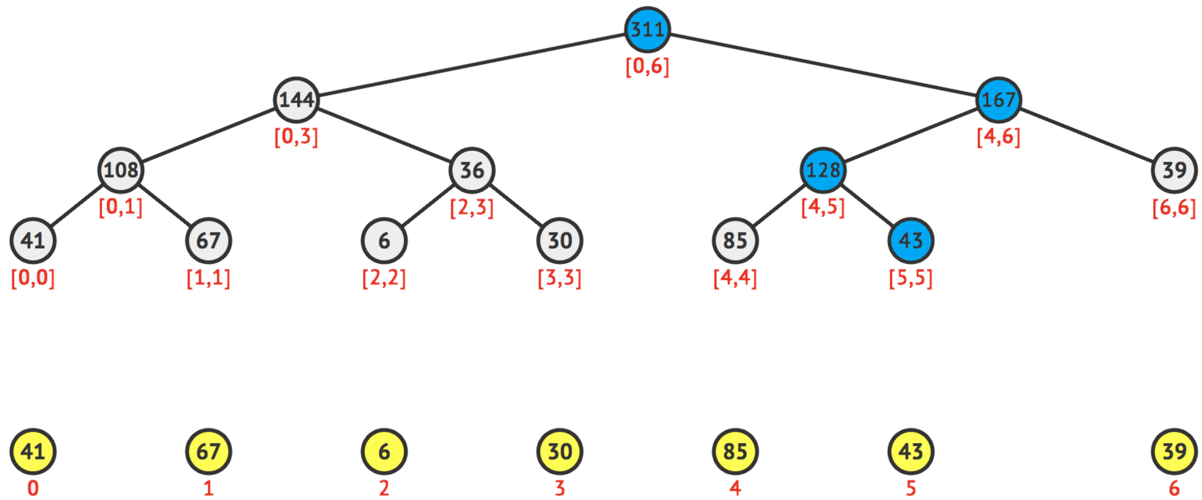

Figure 5: $a = [41, 67, 6, 30, 85, 43, 39]$ dizisinde 5 indeksli elemanın cevabını güncellerken güncellememiz gereken düğümler sekildeki gibidir.

```
void update(int ind,int l,int r,int x,int val) {
  if (l > x || r < x) //bulundugumuz aralik x indeksli elemani icermiyor.
    return;
  if (l == x and r == x) {
    tree[ind] = val; //x indeksli elemani iceren yaprak dugumunun cevabini guncelliyoruz.
    return;
  }
  int mid = (l + r) / 2;
  // recursive bir sekilde x indeksli elemani iceren
  // tum araliklarin cevaplarini guncelliyoruz.
  update(ind * 2,l,mid,x,val);
  update(ind * 2 + 1,mid + 1,r,x,val);
  tree[ind] = tree[ind * 2] + tree[ind * 2 + 1];
}
```

Segment Tree veri yapısı ile ilgili problem: Link.

8 Örnek Problemler

Veri yapıları üzerinde pratik yapabilmeniz için önerilen problemler :

1. Link
2. Link
3. Link
4. Link
5. Link
6. Link.

References

[1] https://en.wikipedia.org/wiki/Data_structure

[2] https://cp-algorithms.com/data_structures/sparse-table.html

[3] https://cp-algorithms.com/data_structures/segment_tree.html

[4] https://cp-algorithms.com/data_structures/fenwick.html

[5] https://cp-algorithms.com/data_structures/sqrt_decomposition.html

[6] https://cses.fi/book/book.pdf

[7] https://visualgo.net/en/segmenttree

[8] https://visualgo.net/en/fenwicktree

[9] https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/

[10] http://www.cs.ukzn.ac.za/ hughm/ds/slides/20-stacks-queues-deques.pdf

[11] https://www.geeksforgeeks.org/stack-data-structure/

[12] https://www.geeksforgeeks.org/queue-data-structure/

[13] https://www.geeksforgeeks.org/deque-set-1-introduction-applications/

[14] https://www.geeksforgeeks.org/linked-list-set-1-introduction/

[15] https://www.geeksforgeeks.org/binary-indexed-tree-range-updates-point-queries/

[16] https://visualgo.net/en/list