Proje Ana Alanı: Yazılım

Proje Tematik Alanı : Özgün Algoritma Tasarımı

Proje Adı (Başlığı) :

Asosyatif Dizilere Yeni Bir Bakış Açısı: Rastgele Düzenlenmiş Ağaç (RDA)

Özet

En yaygın veri yapılarından olan asosyatif diziler -veya yaygın adlarıyla sözlükler- birçok algoritmanın temelini oluşturur. Anahtarlara değer atamak ve atanan değerleri sorgulamak çok temel bir ihtiyaç olup bu sorguları daha hızlı yanıtlayan bir asosyatif dizinin tasarlanması birçok alanda gelişmelere yol açacaktır. Bu projede yüksek hacimli anahtarlar ile ilgili işlemleri daha hızlı bir şekilde gerçekleştirmek amacıyla tasarladığımız özgün veri yapısı sunulmuş ve incelenmiştir. Rastgele Düzenlenmiş Ağaç (RDA) adını verdiğimiz asosyatif dizi, çeşitli testler aracılığıyla performans bakımından sınanmış ve analiz edilmiştir. Geniş bir girdi yelpazesi için muadillerinden daha hızlı bir şekilde test sorgularını yanıtlayan RDA, büyük boyutlu anahtarlar ile yapılan testlerde RDA, yakın zamanda ortaya atılmış olduğundan onlarca yıldır geliştirilmekte olan ve doygunluk noktasına ulaşmış muadillerinin aksine geliştirilmeye açıktır. Bilgisayar bilimine ve veri yapıları tasarımına yeni bir bakış açısı katacak olup esnekliği sayesinde birçok mevcut ve gelecek programa entegre edilebilecek RDA veri yapısı, bu raporda tanıtılır.

Anahtar kelimeler: Veri yapıları, optimizasyon, arama ağaçları, asosyatif veri yapıları

Amaç

Arama motorlarından (Jantkal ve Deshpande, 2017) biyoinformatiğe (Sharma, ve diğ., 2018) dosya sistemlerinden (Rodeh, ve diğ., 2013) önbelleklemeye (Ruan, 1998) birçok alanda asosyatif dizi sorgularından yararlanılır. Asosyatif dizi sorgularının verimli bir şekilde yapılması birçok önemli algoritmanın temelidir. Bu durum, önceki veri yapılarından daha verimli ve daha kapsayıcı bir asosyatif dizinin tasarlanması için bir nedendir. Yeni asosyatif dizilerin geliştirilmesini kapsayan problem "sözlük problemi" olarak adlandırılır.

Asosyatif dizilerin kullanıldığı bazı alanlarda büyük boyutlu anahtarlara sıkça ihtiyaç duyulmaktadır. Anahtarların dizi, kelime katarı, çizge gibi veri yapılarından oluştuğu bu durumlarda bazen asosyatif dizilerin kullanımının arkasındaki amaç anahtarlara değer atamak değil, anahtarların kendilerini verimli bir şekilde depolanabilmesidir. Ancak bu tip büyük boyutlu anahtarlar için hazırlanmış özel bir asosyatif dizi bulunmamaktadır. Bu durumun oluşturduğu ihtiyaç yeni bir asosyatif dizinin geliştirilmesi için motivasyondur.

1. Giriş

Verilere veya veriler hakkındaki bilgilere hızlı erişim için verinin depolanma şeklini değiştirmek avantajlı olabilir. Veri yapıları, veriyi düzenleyerek algoritmaların kullandıkları zaman veya hafıza miktarlarını düşürmeyi amaçlarlar.

Asosyatif diziler, (anahtar, değer) ikililerinin eklenmesini, çıkarılmasını ve belirli bir anahtarın sahip olduğu değerin sorgulanmasını destekleyen veri yapılarıdır. Asosyatif dizilerin çok sayıda algoritmanın temelini oluşturması, farklı avantaj ve dezavantajlara sahip birçok asosyatif dizinin tasarımına yol açmıştır. (Frederickson, 1983)

1.1 Yaygın Kullanılan Bazı Asosyatif Diziler ve Özellikleri

1.1.1 Hash Tablosu

En yaygın kullanılan asosyatif dizi olan hash tablosu, neredeyse tüm high-level programlama dillerinde hazır olarak sunulur. Veriler, hash tablosuna "hash fonksiyonu" adlı bir fonksiyonun çıktısına göre yerleştirilir. Bu fonksiyonların çalışma zamanı genellikle anahtar büyüklüğüne (K) göre lineerdir. Eğer hash fonksiyonu iki anahtar için aynı çıktıyı üretirse anahtarlar "çarpışır" ve bu çarpışmalar verinin bozulmasına ya da kaybolmasına yol açabilir. Çarpışmaların önlenmesi için kullanılan çeşitli algoritmalar, hash tablolarını yavaşlatır. (Flajolet, ve diğ., 1988)

Hash Tabloları	Θ Zaman Karmaşıklığı (Average Case)	O Zaman Karmaşıklığı (Worst Case)
Ekleme (Insertion)	$\Theta(K)$	$O(NK)$
Çıkarma (Removal)	$\Theta(K)$	$O(NK)$
Arama (Search)	$\Theta(K)$	$O(NK)$

Tablo-1

Hash tablolarında başvurulan çarpışma önleyici yöntemlerin başında:

- Birleşik Hashleme (Coalesced Hashing) (Vitter, 1982)
- Cuckoo Hashing (Pagh ve Rodler, 2004)
- Hopscotch Hashing (Herlihy, ve diğ., 2008)
- Robin Hood Hashing (Celis, ve diğ., 1985)

gelir.

1.1.2 Arama Ağaçları (Search Trees)

Arama ağaçları, hash tabloları gibi çok yaygın bir şekilde kullanılır ve birçok programlama dilinde hazır olarak sunulur. Arama ağaçları, veriyi sıralı ağaç üzerinde belirli bir kurala göre depolar.

Örneğin ikili arama ağaçlarında veri; bir düğümün sol alt ağacında kendinden küçük, sağ alt ağacında ise kendinden büyük değere sahip anahtarlar bulunacak şekilde düzenlenir. Bu sayede ikili arama ağaçları üzerinde bir değer aranırken ikili arama gerçekleştirilebilir.

Arama ağaçlarında ziyaret edilen her düğümde tek bir karşılaştırma operasyonu gerçekleştirilir. Bu yüzden sorgular yanıtlanırken ağacın derinliği cinsinden lineer miktarda karşılaştırma işlemi gerçekleştirilir. Ağacın derinliği H, sorguda kullanılan anahtar boyutu K olmak üzere, arama ağaçlarının zaman karmaşıklıkları *Tablo-2*'de verilmiştir.

Arama Ağaçları	Θ Zaman Karmaşıklığı (Average Case)	O Zaman Karmaşıklığı (Worst Case)
Ekleme (Insertion)	$\Theta(HK)$	$O(HK)$
Çıkarma (Removal)	$\Theta(HK)$	$O(HK)$
Arama (Search)	$\Theta(HK)$	$O(HK)$

Tablo-2

Kendini dengeleyen arama ağaçları, ağacın derinliği (H) her zaman anahtar sayısı (N) bakımından logaritmik olacak şekilde düğümleri düzenler. Bu sayede çok hızlı sorgular yapılır.

Kendini Dengeleyen Arama Ağaçları	Θ Zaman Karmaşıklığı (Average Case)	O Zaman Karmaşıklığı (Worst Case)
Ekleme (Insertion)	$\Theta(K \log N)$	$O(K \log N)$
Çıkarma (Removal)	$\Theta(K \log N)$	$O(K \log N)$
Arama (Search)	$\Theta(K \log N)$	$O(K \log N)$

Tablo-3

Dengeli arama ağaçları, hash tablolarına göre daha stabil bir zaman karmaşıklığı sunar.

Kendini dengeleyen arama ağaçlarına bazı bilindik örnekler şunlardır:
- Kırmızı-Siyah Ağaç (Guibas ve Sedgewick, 1978)
- AVL Ağacı (Adel′son-Velskii ve Landis, 1962)
- Splay Ağacı (Splay Tree) (Sleator ve Tarjan, 1985)
- Treap (Seidel & Aragon, 1996)
- B-Ağacı (B-Tree) (Bayer ve McCreight)
- B+ Ağacı (B+ Tree) (Comer, 1979)

1.1.3 Ön Ek Ağacı (Trie / Prefix Tree)

Ön ek ağacı, metin eşleştirme ve otomatik metin tamamlama sistemlerinde sıkça kullanılan bir asosyatif dizidir. Ön ek ağaçları, (anahtar, değer) ikililerini birlikte depolamaz. Anahtarlara ait birimler (örn. stringler için karakterler, sayılar için bitler vs.) düğümlerde bulunur, sorgular gerçekleştirilirken ilgili düğümler takip edilir. Ön ek ağaçlarında bir anahtar üzerinde sorgular gerçekleştirilirken anahtarın içerdiği birim sayısı (K) kadar birim ve dolayısıyla düğüm ziyaret edilir.

Ön Ek Ağacı	Θ Zaman Karmaşıklığı (Average Case)	O Zaman Karmaşıklığı (Worst Case)
Ekleme (Insertion)	$\Theta(K)$	$O(K)$
Çıkarma (Removal)	$\Theta(K)$	$O(K)$
Arama (Search)	$\Theta(K)$	$O(K)$

Tablo-4

Bir anahtarın barındırdığı her bir birim için yeni bir düğüm oluşturmak, ön ek ağaçlarının diğer asosyatif dizilere göre daha fazla hafıza ayırmasıyla sonuçlanır. Bu durum aynı zamanda programın yavaş çalışmasına sebep olur. Ön ek ağaçları, genelde küçük boyutlu anahtarlar ile kullanılırlar.
En sık kullanılan ön ek ağacı çeşitleri:
- Ön Ek Ağacı (Fredkin, 1960)
- Sıkıştırılmış Ön Ek Ağacı (Compressed Trie) (Morrison, 1968)
- Son Ek Ağacı (Suffix Tree) (Weiner, 1973)
- Judy Dizisi (Gobeille & Baskins, 2004)

1.2 Yeni Veri Yapıları için Arayış

Büyük anahtarlar ile ilişkili sorgularda yaygın asosyatif dizilerin karşılaştığı bazı problemler şu şekildedir:

- Hash tablolarında kullanılan hashler $O(K)$ zaman karmaşıklığında çalışmaktadır. Hashleme sürecinde çok sayıda yavaş aritmetik işlem uygulandığından, çarpışma yaşanmasa bile hash tabloları asimptotik karmaşıklıklarına rağmen yavaş çalışabilirler.
- İki anahtar arasında karşılaştırma operasyonu gerçekleştirmek $O(K)$ zaman karmaşıklığına sahiptir. Ziyaret edilen her düğümünde bir tane karşılaştırma operasyonu gerçekleştirmek zorunda olan arama ağaçları performans bakımından zayıftır.
- Ön ek ağaçları, büyük anahtarlar ile çalışırken çok yüksek sayıda düğüm oluşturmaya ihtiyaç duyabilirler. Bu durum veri yapısının zayıf yönünü tetiklemekte, yapıyı yüksek miktarlarda hafıza ayırmak zorunda bırakmaktadır.

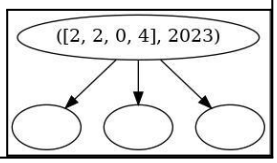
Bu sorunlar göze alındığı zaman, anahtar boyutu üzerinden yapılacak bir optimizasyona ihtiyaç duyulduğu görülür. Pek çok alanda büyük anahtarlar sıkça kullanılırken yaygın asosyatif veri yapılarının bu tip sorunlar ile karşılaşması yeni bir asosyatif veri yapısının tasarımı için neden sunar. Geliştirdiğimiz yeni veri yapısı bu doğrultuda tasarlanmıştır.

2. Yöntem

Bu bölümde RDA veri yapısının temel çalışma prensipleri ve çalışma zamanını düşürmek için uygulanan optimizasyonlar açıklanmıştır.

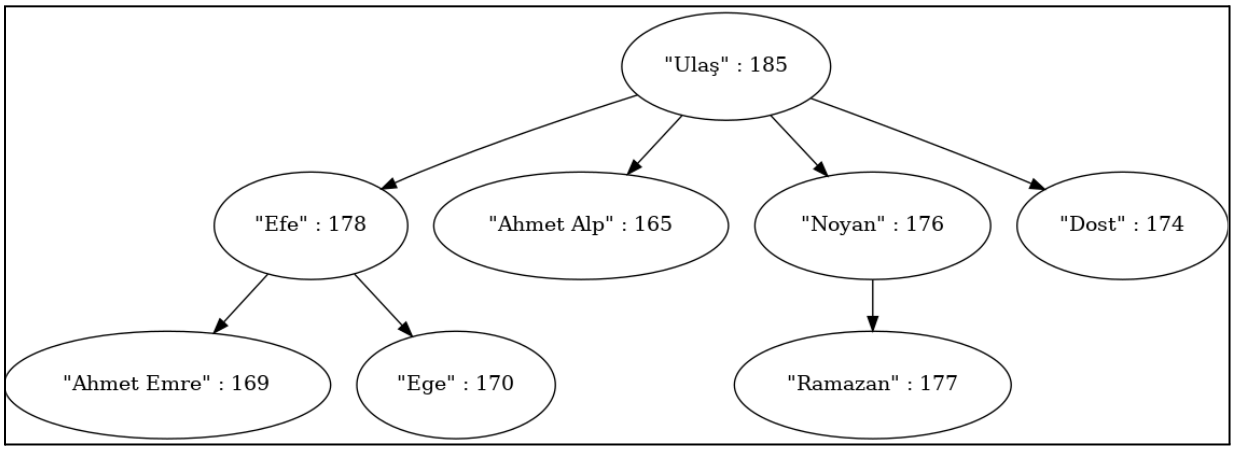
2.1 Veri yapısı

```
template <typename Value_T, const int size = 4>
struct Node {
    char *key = nullptr;
    int keyLength = 0;
    Value_T data;
    Node<Value_T, size> *children[size];
    inline ~Node() {
        delete[] key;
    }
};
```



Bir tür arama ağacı olan veri yapımız, her düğümde bir anahtar ve bu anahtara ait değeri bulundurur. Çocuk düğümlerin işaretçileri de düğümde depolanır. Görsel-1'de bulunan modelde anahtar [2, 2, 0, 4] dizisi olup, 2023 ise bu diziye atanan değerdir. Görsel-1'de bulunan modeldeki düğümün üç çocuk düğümü vardır.

Görsel-1



Görsel-2

Görsel-2'de bir arkadaş grubunun isim ve boy verilerini içeren bir RDA modeli verilmiştir. Düğümlerin değişken sayıda çocuk düğüme sahip olduğu görülse de, ağaçtaki her düğümün sabit sayıda çocuğu olup görselde gösterilmeyen düğümlerin boş olduğu varsayılmalıdır. Görüleceği üzere RDA veri yapısı, öteki arama ağaçlarının aksine düğümleri içerdikleri anahtarlara göre sıralı bir biçimde düzenlemez.

2.2 Sorgular

RDA veri yapısının temel mantığını oluşturan ağaç üzerinde dolaşma (traversal) yöntemi, sorguların hızlı bir şekilde gerçekleştirilmesini sağlar. Dolaşma algoritması, anahtardan gelişigüzel seçtiği indisler ile dolaşacağı düğümleri belirlemeye dayanır. Bir düğümden sonraki düğüme geçiş yaparken sabit zamanlı operasyonlar kullandığından düşük bir çalışma zamanına sahip olan dolaşma algoritmasının çalışma mantığı Görsel-3'te verilen kod parçasında detaylıca açıklanmıştır. Bu dolaşma yöntemi diğer sorguların yanıtlanmasında da ortak olarak kullanılacaktır ve veri yapısının temel çalışma prensibini oluşturur.

```
while (1) {
    //hangi çocuk düğüme ilerleneceği anahtarın indisteki değerine göre belirlenir
    next = key[index] % size;
    cur = cur->children[next];

    //sıradaki indis önceki indise göre belirlenir
    index -= key[index] % len;
    if (index < 0) index += len;
}
```

Görsel-3

Görsel-3'te yer alan kod parçasında bulunan "cur" değişkeni şu anda üzerinde bulunduğumuz, "next" değişkeni ise sıradaki ziyaret edilecek düğümdür. "index", bir sonraki düğüm belirlenirken kullanılacak indistir. Bu değişken döngünün her bir iterasyonunda anahtardan alınan kesite ve eski değerine göre güncellenir. İndisler anahtara bağlı olduğundan farklı anahtarlar için farklı indisler seçilir. Böylece indis seçimine ve dolaylı olarak anahtarların ziyaret edeceği düğümlere rastgelelik katar. Bu nedenle çok sayıda ortak kesit içeren anahtarlar bile çoğunlukla farklı düğümleri ziyaret ederler.

2.2.1 Arama (Query)

Arama fonksiyonumuza ait kod parçası Görsel-4'de verilmiştir. Bu fonksiyon anahtarı girdi olarak alıp anahtarın yer aldığı düğümün adresini döndürmektedir. Aranan anahtarın veri yapısında bulunmaması durumunda nullptr döndürülmektedir.

```
Node<Value_T, size> *find(const char *key, const int len) {
    //değişken tanımları
    int index = len - 1, next;
    Node<Value_T, size> *cur = root;

    //arama döngüsü
    while (1) {
        //düğümdeki anahtar ile aranan anahtar aynı ise düğümü döndür
        if (cur->keyLength == len && comp(cur->key, key, len)) return cur;

        //değilse sıradaki düğümü belirle
        next = key[index] % size;

        //ziyaret edilecek düğüm oluşturulmadıysa anahtar yoktur
        if (cur->children[next] == nullptr) return nullptr;

        //ilerle ve değişkenleri güncelle
        cur = cur->children[next];
        index -= key[index] % len;
        if (index < 0) index += len;
    }
}
```

Görsel-4

Görsel-4'de görüleceği üzere Görsel-3'de açıklanan dolaşma yöntemine arama algoritmasında başvurulmuştur. Yeni bir düğüm ziyaret edildiğinde, bu düğümde yer alan anahtar ile aranan anahtarın eşleşip eşleşmediği kontrol edilir. Eğer anahtarlar eşleşiyorsa düğümün adresi döndürülür. Aksi halde sıradaki düğüme ilerlenir ve aramaya devam edilir. Eğer dolaşılacak bir sonraki düğüm oluşturulmamışsa süreç sonlandırılır.

2.2.2 Ekleme (Insertion)

```
inline Node<Value_T, size> *insert(const char *key, const Value_T data, const int len) {
    //değişken tanımları
    int index = len - 1, next;
    Node<Value_T, size> *res = nullptr, *cur = root;

    //arama döngüsü
    while(1){
        //eğer değeri silinmiş bir düğüm bulunmuşsa anahtarı yerleştirmek için kaydet
        if (cur->keyLength == -1) res = cur;

        //düğümdeki anahtar ile eklenen anahtar aynı ise düğümü güncelle
        if (cur->keyLength == len && comp(cur->key, key, len)) {
            cur->data = data;
            return cur;
        }

        next = key[index] % size;

        //eğer anahtar önceden eklenmediyse anahtarı ağaca ekle
        if (!cur->children[next]) {
            //eğer anahtarın yerleştirilmesi için müsait bir düğüm bulunmuyorsa
            if (!res) {
                //yeni bir düğüm oluştur
                cur->children[next] = new Node<Value_T, size>;
                res = cur->children[next];
            }

            //anahtarı düğüme yerleştir
            res->key = new char[len];
            copy(res->key, key, len);
            res->keyLength = len;
            res->data = data;
            return res;
        }

        //ilerle ve değişkenleri güncelle
        cur = cur->children[next];
        index -= key[index] % len;
        if (index < 0) index += len;
    }
}
```

Görsel-5

Görsel-5'te veri yapımıza yeni (anahtar, değer) çiftlerinin eklenmesi açıklanmıştır. Görsel-3'de açıklanan dolaşma yöntemi bu sorgu tipinde de kullanılmıştır. Yeni bir anahtar ekleneceği zaman Görsel-5'te açıklanan yönteme göre ağaç üzerinde hareket edilir ve uygun düğüm bulunduğunda ya da oluşturulduğunda düğüme (anahtar, değer) çifti yerleştirilir.

Ekleme yapılırken göz önünde bulundurulması gereken 3 temel durum vardır:

1. **Anahtarın yapıda mevcut olması:**
Bu durumda anahtara atanan değer yeni değer ile güncellenir.

2. **Anahtarın yerleştirilebileceği boş bir düğüm bulunması:**
Bu durumda düğüme (anahtar, değer) ikilisi yerleştirilir.

3. **Anahtarın yerleştirilebileceği bir düğümün bulunmaması:**
Bu durumda yeni bir yaprak düğüm oluşturulur ve düğüme (anahtar, değer) ikilisi yerleştirilir.

2.2.3 Çıkarma (Removal)

```
inline void remove(const char *key, const int len) {
    //silinecek anahtara ait düğümü belirle
    Node<Value_T, size> *to_delete = find(key, len);

    //eğer düğüm bulunamadıysa silinecek bir şey yoktur
    if (to_delete == nullptr) return;

    //düğümdeki verileri temizle
    delete[] to_delete->key;
    to_delete->keyLength = -1;
}
```

Görsel-6

Veri yapımızdan bir anahtara çıkarılacağı zaman içerisindeki anahtarı barındıran düğüm belirlenir ve düğüm temizlenir. Bütün süreç bir adet arama ve hafıza boşaltma ile özetlenebilir. Dolayısıyla bu sorgunun çalışma zamanı arama sorgusunun çalışma zamanı ile aynıdır.

2.2.4 Sorguların Zaman Karmaşıklığı

Veri yapımızın (RDA) zaman karmaşıklığı Tablo-5'de verilmiştir.

RDA	Θ Zaman Karmaşıklığı (Average Case)	O Zaman Karmaşıklığı (Worst Case)
Ekleme (Insertion)	$\Theta(K + \alpha \log N)$	$O(KN)$
Çıkarma (Removal)	Anahtara ait bir değer mevutsa $\Theta(K + \alpha \log N)$ Anahtara ait bir değer mevcut değilse $\Theta(\alpha \log N)$	$O(KN)$
Arama (Search)	Anahtara ait bir değer mevutsa $\Theta(K + \alpha \log N)$ Anahtara ait bir değer mevcut değilse $\Theta(\alpha \log N)$	$O(KN)$

Tablo-5

2.2.4.1 Θ Zaman Karmaşıklığı

Geliştirdiğimiz dolaşma algoritmasının doğası gereği, aynı düğümleri ziyaret eden anahtarların ön ekleri arasında bir ilişki bulunmaz. İki anahtarın karşılaştırma maliyeti ön ek benzerliğine bağlı olduğundan bu durum veri yapımızın gerekli karşılaştırmaları hızlı bir şekilde sonuçlandırmasını sağlar. Tablo-5'de bulunan α değişkeni, veri setindeki ön ek benzerliğidir. Rastgele belirlenmiş bir anahtar setinde $\alpha \approx 0$ iken, bütün anahtarların "27.12.2023" tarihi gibi bir ön ek ile başladığı bir veri setinde $\alpha \approx 10$ olacaktır.

2.2.4.2 O Zaman Karmaşıklığı

Bütün anahtarların ağaç üzerinde çok sayıda ortak düğümü ziyaret ettiği, aynı uzunlukta

olduğu, ve/veya ön ek benzerliklerinin çok yüksek olduğu durumda gerçekleşir. Bu durumun engellenmesi için bazı çalışmalar, optimizasyonlar bölümünde anlatılmıştır.

2.3 Optimizasyonlar

Bu bölümde RDA'nın gerçekleştirdiği sorguların zaman ve hafıza bakımından geliştirilmesi için yapılan çeşitli değişiklikler açıklanmıştır. Bazı değişiklikler algoritmanın çalışma biçimini değiştirirken bazı değişiklikler algoritmadaki çeşitli operasyonları hızlandırmaya dayalıdır.

2.3.1 Derinlik Düşürme Optimizasyonları

Sorgular gerçekleştirilirken dolaşılan düğüm sayısının düşük tutulması için ağacın derinliğinin düşük tutulması gerekir. Bazı arama ağaçları bunu sağlamak için dengeleme operasyonu kullanırken (Sleator ve Tarjan, 1985) RDA bu operasyonu desteklemez. Dolayısıyla derinliğin düşük tutulması için farklı yöntemlere başvurulmuştur.

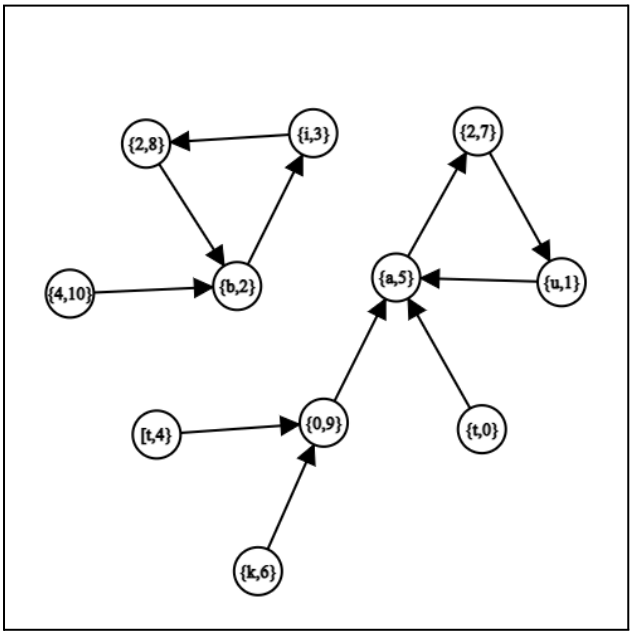
2.3.1.1 Derinliğe Dayalı Kaydırma

Görsel-3'de verilen kod parçasında index değişkeni güncellenirken gerçekleştirilen operasyon formül (1)'de verilmiştir.

$$index := index \; - \; key[index] \,\%\, len \qquad (1)$$

Bu metodun çalışma prensibini bir örnek ile inceleyelim.

Anahtarı "tubitak2204" değerine sahip bir kelime katarı olan bir (anahtar, değer) çifti veri yapısına eklenecek olursa, Görsel-7'deki fonksiyonel graf index değişkeninin değişimini modeller. Örneğin 5. indisten sonra 7. indis, 6. indisten sonra 9. indis gezilecektir. "tubitak2204" anahtarı için index değişkeninin başlangıç değeri 10 olacağından index değişkeninin bu fonksiyonel graf üzerinde izleyeceği yol {10, 2, 3, 8, 2, 3, …} olacaktır. Oluşacak {2, 3, 8} döngüsü diğer indislerin gezilmesini engelleyecektir. Bu durum "??bi????2?4" şeklinde verilen bütün anahtarların arama ağacı üzerinde aynı düğümleri gezeceği anlamına gelir.



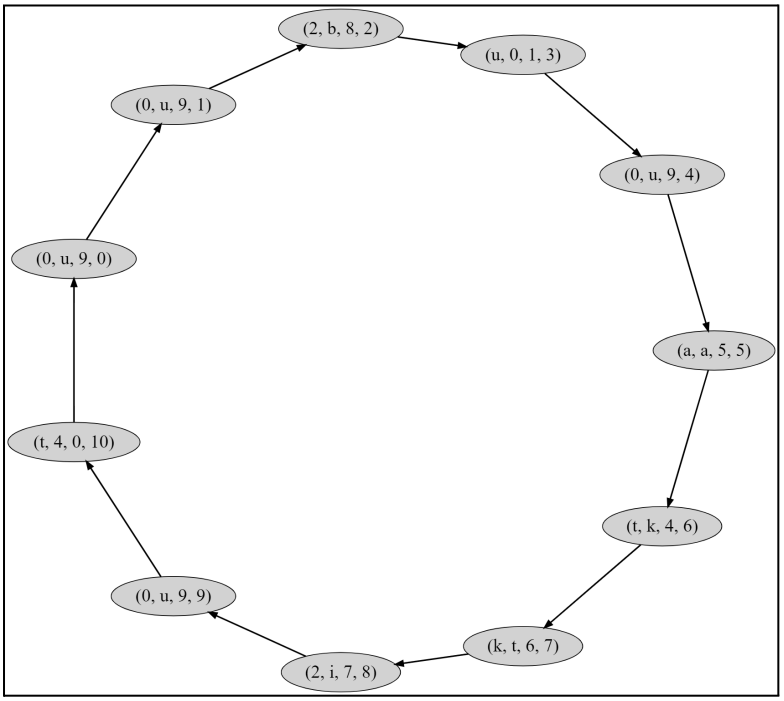
Çok sayıda anahtarın yerleştirilme sürecinde benzer düğümlerin ziyaret edilmesi, ağacın derinliğini arttıracaktır ve sorgular için en kötü durum olan $O(NK)$ zaman karmaşıklığını tetikleyecektir. *Görsel-7*

Daha fazla indisin ziyaret edilmesi için formül (1)'de bulunan yöntem formül (2)'de yöntem ile değiştirilmiştir.

$$index := index \; - \; (key[len - index - 1] \; + key[index] \; + depth) \,\%\, len \qquad (2)$$

Görsel-8'de index değişkeninin anahtar ve derinliğe göre değişimini modelleyen fonksiyonel

grafa ait bir döngü çizimi verilmiştir. Bu fonksiyonel graftaki her düğümde (key[index], key[len-index-1], index, depth) dörtlüsü verilmiştir. $depth\%len$ toplamda len farklı değer alabileceğinden fonksiyonel grafın tamamı toplamda $len * len$ tane düğüm barındıracaktır. Derinliğin len modundan kalanı ile işlem yapılması, oluşacak döngülerin büyüklüğünün anahtar boyutunun bir katı olmasını sağlar. Yani formül (2)'deki metod ile oluşabilecek en az sayıda düğüme sahip döngü, Görsel-3'de bulunan metod kullanılırken oluşabilecek en çok sayıda düğüme sahip döngü ile aynı sayıda düğüme sahip olacaktır. Böylece fonksiyonel graf üzerindeki döngülerin sahip olduğu düğüm sayısı artırılacaktır. Görsel-8'de görüleceği üzere bu yöntem sayesinde "tubitak2204" anahtarının tüm indisleri ziyaret edilecektir. Daha fazla indisin kontrol edilmesini sağlayan bu yöntem çok sayıda farklı anahtarın aynı yolu izleyerek derinliği arttırmasını engeller. Böylece $O(NK)$ zaman karmaşıklığından çoğunlukla kaçınılır.



Görsel-8

2.3.1.2 Orman Optimizasyonu

Derinliğin düşük tutulması için tek bir arama ağacı yerine çok sayıda ağaçtan oluşan bir arama ağacı topluluğu, diğer bir deyişle bir arama ormanı kullanılabilir. Bu sayede derinlik ve böylece sorguların çalışma zamanı düşer. RDA'ya birden fazla ağaç desteği entegre edilirken "J" adlı bir sabit tanımlanmıştır. J sabiti derleme zamanı içerisinde atanır ve ağaç sayısını belirtir.

2.3.2 Bit Optimizasyonları

Görsel-3'de görüleceği üzere geliştirdiğimiz veri yapısında düğümler gezilirken mod operasyonu uygulanır. Bu operasyon diğer aritmetik operasyonların çoğundan yavaş olduğundan büyük performans kayıplarına sebep olabilir. Dolayısıyla mod operatörü yerine "bitwise-and" (&) operatörünün kullanılmasına karar verilmiştir.

Görsel-3'de görüleceği üzere bir düğümden sonra gezilecek bir sonraki düğüm belirlenirken çocuk sayısından kalan hesaplanır. Eğer çocuk sayısı 2'nin bir tam sayı kuvveti ise bu işlem mod operatörü kullanılmadan sadece bir adet bitwise-and işlemi ile gerçekleştirilebilir. Tek bir bitwise-and işlemi ile çocuk sayısından kalan hesaplamak için formül (3)'ten yararlanılır.

$$X \,\%\, 2^n = X \,\&\, (2^n - 1),\ X \in \mathbb{N} \qquad (3)$$

Bu bağlamda Görsel-9'da verilen kod parçası Görsel-10'daki kod parçası ile değiştirilir.

```
next = (key[len - index - 1] + key[index] + depth) % size;
```

Görsel-9

```
next = (key[len - index - 1] + key[index] + depth) & (size - 1);
```

Görsel-10

Bu şekilde veri yapısı güncellenmiş, düğümlerin çocuk sayısı iki sayısının kuvvetleri olacak şekilde kısıtlanmıştır.

Görsel-3'de görüleceği üzere index değişkeni güncellenirken de aynı teknik uygulanabilir. Bu kapsamda len değişkeni ile kalan hesaplamak yerine $k = 2^n - 1, \exists\, n \in \mathbb{N}$ ve $k < len$ şartını sağlayan en büyük k tam sayısı ile "bitwise and" operasyonu gerçekleştirilir. Buna göre Görsel-11'de verilen kod parçası Görsel-12'deki kod parçası ile değiştirilir. Burada "modConst" değişkeni k tam sayısıdır.

```
index -= (key[len - index - 1] + key[index] + depth) % len;
```

Görsel-11

```
index -= (key[len - index - 1] + key[index] + depth) & modConst;
```

Görsel-12

2.4 Performans Testlerinin Yapılması

RDA, 6 farklı veri yapısı ile birlikte çeşitli testlere tabi tutularak performans bakımından incelenmiştir. Bu veri yapıları:

- Hash tabloları
    - ankerl::unordered_dense
    - emhash8
    - robin_hood::unordered_map
- Arama Ağaçları
    - absl::btree_map
    - std::map
- Ön ek ağacı
    - Judy Array

olarak belirlenmiştir.

Yapılan testlerde "11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz × 8" modelli bir işlemci ile 24gb bellek kullanılmıştır. Kullanılan bilgisayar Ubuntu 22.04.3 LTS x86_64 işletim sistemi ile çalışmaktadır. Testlerin gerçekleştirilmesi için kullanılan işlemcinin 2 tane çekirdeği izole edilmiştir ve testler bu çekirdeklerde gerçekleştirilmiştir.

Veri yapılarına 3 farklı test uygulanmıştır. Bu testlerin her biri 5 alt testten oluşmuştur. Bütün alt testler 10 defa tekrarlanmıştır.

2.1 Ekleme-Çıkarma Testleri

Bu testler 2 temel adımdan oluşur. Önce bir miktar anahtar veri yapısına eklenir, sonrasında eklenen bütün anahtarlar veri yapısından çıkartılır. Bu sayede veri yapısında yapılan değişikliklerin ne kadar hızlı gerçekleştiğine dair bilgi edinilir.

Ekleme-Çıkarma testinin alt testleri sırasıyla:

- 10^7 adet 1kb
- 10^6 adet 10kb
- 10^5 adet 100kb
- 10^4 adet 1mb
- 10^3 adet 10mb

boyutunda anahtarlar ile gerçekleştirilmiştir. Sonuçlar, Bulgular bölümünde Tablo-7'de anahtarların bayt cinsinden boyutuna göre verilmiştir.

2.2 Mevcut Veri Arama Testi (Birinci Arama Testi)

Bu test, veri yapılarının içerisinde bulundurduğu anahtar sayısı ile arama sorgularının çalışma zamanı arasındaki ilişkiyi incelemek amacıyla gerçekleştirilmiştir. Testin her adımında 4 adet anahtar veri yapısına eklenirken, ardından eklenen anahtarlardan 200 adet anahtara ait değer sorgulanır.

Birinci arama testinin alt testleri sırasıyla:

- 10^6 adet 1kb anahtar ekleme, $5 * 10^7$ adet 1kb anahtar arama
- 10^5 adet 10kb anahtar ekleme, $5 * 10^6$ adet 10kb anahtar arama
- 10^4 adet 100kb anahtar ekleme, $5 * 10^5$ adet 100kb anahtar arama
- 10^3 adet 1mb anahtar ekleme, $5 * 10^4$ adet 1mb anahtar arama
- 10^2 adet 10mb anahtar ekleme, $5 * 10^3$ adet 10mb anahtar arama

şeklindedir. Bu testin sonuçları Bulgular bölümünde Tablo-8'de verilmiştir.

2.3 Eksik Veri Arama Testi (İkinci Arama Testi)

Bu testlerdeki test verisi ve testlerin formatı birinci arama testleri ile aynıdır. Tek fark bu testlerde aranan anahtarlara önceden değer atanmamış olmasıdır. Test sonuçları Bulgular bölümünde Tablo-9'da verilmiştir.

3. Proje İş-Zaman Çizelgesi

AYLAR				
İşin Tanımı	**Ekim**	**Kasım**	**Aralık**	**Ocak**
Literatür Taraması	**X**	**X**	**X**	
Yazılım Geliştirme		**X**	**X**	
Performans Testlerinin Yapılması			**X**	
Proje Raporu Yazımı			**X**	**X**

Tablo-6

4. Bulgular

Test	RDA	std::map	ankerl::unordered_dense	btree	emhash8	Judy Array	Robin-Hood
1000	10997	53888	10199	39789	17325	MLE	12627
10.000	3806	9464	6236	6989	10908	MLE	8374
100.000	3537	5953	5667	5655	9531	MLE	7125
1.000.000	3502	6290	5803	5699	9937	MLE	7490
10.000.000	3684	7245	6393	6320	11018	MLE	8413

Tablo-7

Test	RDA	std::map	ankerl::unordered_dense	btree	emhash8	Judy Array	Robin-Hood
1000	15241	58363	8226	55898	12635	21353	8088
10.000	6486	9522	6231	8301	10418	MLE	6157
100.000	5488	6652	6001	5178	10228	MLE	6034
1.000.000	5464	8196	6510	5097	10717	MLE	6396
10.000.000	5737	9575	7367	5084	11538	MLE	7263

Tablo-8

Test	RDA	std::map	ankerl::unordered_dense	btree	emhash8	Judy Array	Robin-Hood
1000	1700	8821	3006	9955	8256	2235	2859
10.000	394	1077	2338	1178	7527	MLE	2340
100.000	229	485	2911	488	7931	MLE	3088
1.000.000	251	428	3554	431	8202	MLE	3554
10.000.000	248	494	3535	495	8053	MLE	3538

Tablo-9

Bu tabloların daha iyi yorumlanabilmesi açısından değerler normalize edilmiştir. Tablo-10, Tablo-11 ve Tablo-12'de "1" en iyi test sonucunu gösterirken "2.5" en iyi sonucun 2.5 katı çalışma zamanı ifade eder. Aynı zamanda bu değerlerin geometrik ortalaması verilerek veri yapılarının performansı özetlenmiştir.

Test	RDA	std::map	ankerl::unordered_dense	btree	emhash8	Judy Array	Robin-Hood
1000	1.07	5.28	1	3.9	1.69	MLE	1.23
10.000	1	2.48	1.63	1.83	2.86	MLE	2.20
100.000	1	1.68	1.60	1.59	2.69	MLE	2.01
1.000.000	1	1.79	1.65	1.62	2.83	MLE	2.13
10.000.000	1	1.96	1.73	1.71	2.99	MLE	2.28
Geometrik Ort.	1.01	2.38	1.49	1.99	2.56	MLE	1.92

Tablo-10

Test	RDA	std::map	ankerl::unordered_dense	btree	emhash8	Judy Array	Robin-Hood
1000	1.88	7.21	1.01	6.91	1.56	2.64	1
10.000	1.05	1.54	1.01	1.34	1.69	MLE	1
100.000	1.05	1.28	1.15	1	1.97	MLE	1.16
1.000.000	1.07	1.60	1.27	1	2.10	MLE	1.25
10.000.000	1.12	1.88	1.44	1	2.26	MLE	1.42
Geometrik Ort.	1.19	2.11	1.16	1.56	1.89	MLE	1.15

Tablo-11

Test	RDA	std::map	ankerl::unordered_dense	btree	emhash8	Judy Array	Robin-Hood
1000	1	5.18	1.76	5.85	4.85	1.31	1.68
10.000	1	2.73	5.93	2.98	19.1	MLE	5.93
100.000	1	2.11	12.71	2.13	34.63	MLE	13.48
1.000.000	1	1.70	14.15	1.71	32.67	MLE	14.15
10.000.000	1	1.99	14.25	1.99	32.47	MLE	14.26
Geometrik Ort.	1	2.51	7.68	2.63	20.24	MLE	7.70

Tablo-12

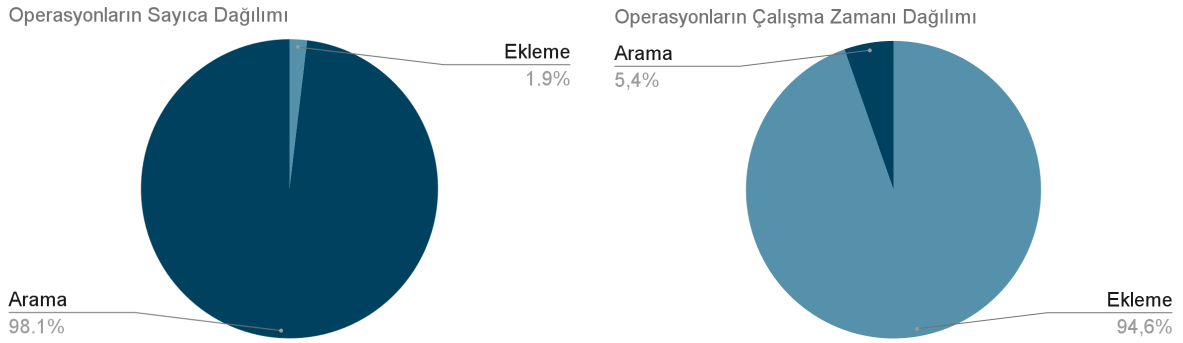
5. Sonuç ve Tartışma

Ekleme-çıkarma testi sonuçları incelendiği zaman RDA'nın diğer veri yapılarından çok daha hızlı bir şekilde bu sorguları yanıtladığı görülmüştür. Bu durumun temel sebebi RDA yapısının dengeleme, kaydırma gibi ek operasyonlara ihtiyaç duymadan veriyi yerleştirmesidir.

Birinci arama testi incelendiği zaman, 1kb anahtarların bulunduğu test sırasında RDA'nın büyük bir performans kaybı yaşadığı görülmüştür. Veri yapısına farklı testler uygulanmış, performans kaybının sebebinin işaretçi aritmetiği (pointer arithmetic) olduğu kanısına varılmıştır. Bütün ağaç yapılarının (absl::btree_map, std::map, Judy Array) karşılaştığı bu

durumdan en az RDA'nın etkilendiği görülmüştür. Öteki testlerde anahtar sayısı daha az olduğundan bu performans kaybı ortadan kalkmıştır.

İkinci arama testi RDA yapısının en yüksek performans gösterdiği test olmuştur. Test sırasında RDA, diğer veri yapılarından katlarca yüksek performans sergilemiştir. Bu yüksek performans, RDA'nın bu test koşulları için sahip olduğu çok düşük zaman karmaşıklığı sayesinde gerçekleşmiştir. Arama operasyonları o kadar hızlı gerçekleştirilmiştir ki, test sırasında tüketilen zamanın büyük bir kısmı arama değil, ekleme operasyonu yaparken harcanmıştır. Tablo-13'te ekleme-arama operasyonlarının bu testteki sayıca dağılımı ve tükettiği çalışma zamanı verilmiştir.



Görsel-13

Bütün test sonuçları birlikte incelendiği zaman, RDA'nın yüksek boyutlu anahtarlar için en uygun veri yapısı olduğu kanaatine varılmıştır.

6. Öneriler

6.1 Dolaşma Yönteminin Optimizasyonu

Açıklanan dolaşma yönteminin oluşturduğu fonksiyonel graflardaki döngü büyüklüklerinin artırılması için yeni dolaşma yöntemleri incelenebilir. RDA veri yapısının kullandığı dolaşma yöntemi geliştirmeye açıktır. Oluşan döngü büyüklükleri üzerine daha yüksek bir alt sınır konulabilirse algoritmanın verimliliği artırılacaktır.

6.2 Dinamik Çocuk Sayısı Optimizasyonu

Veri yapımızda düğüm başına düşen çocuk sayısı arttırıldıkça derinliğin azalması, ve dolaylı olarak hızlanması beklenmektedir. Ancak bunun aksine yapılan bazı testlerde veri yapısının aksi yönde yavaşladığı görülmüştür. Bunun sebebi düğüm başına düşen çocuk düğüm sayısı arttıkça hafıza tüketiminin de artmasıdır. Bazı ağaç yapılarında bu durumun çözümü için çocuk düğümlerin dinamik bir dizide tutulduğu görülebilir. Benzer bir uygulama RDA'yı da hızlandırmak için kullanılabilir.

6.3 Sabitler

RDA yapısında düğüm başına düşen çocuk sayısı ve kök düğüm sayısı derleme zamanında belirlenmektedir. Bu sayılardaki değişimlerin performansı etkilediği bilinse de değerlerin nasıl belirlenmesi gerektiğine dair bir karar verilememiştir. Bu sabitlerin belirlenmesi için bir yöntem bulunması, RDA'yı performans bakımından geliştirebilir.

6.4 Daha Çeşitli Anahtar Desteği

Hazırlanan anahtar operasyonu, tam sayı ve dizi formatındaki anahtarları desteklemektedir. Anahtar desteği genişletilerek daha esnek bir veri yapısı elde edilebilir ve böylece kullanım alanları arttırılabilir.

Kaynaklar

Jantkal, B. A., & Deshpande, S. L. (2017, August). Hybridization of B-Tree and HashMap for optimized search engine indexing. In *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)* (pp. 401-404). IEEE.

[Sharma, N., Goel, M., Verma, A., Jain, A., Parashar, R., & Biswas, P. (2018, December). Discovering non-mutated motifs in DNA sequences: a HashMap based binary search motif finder. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)* (pp. 1-6). IEEE.

Rodeh, O., Bacik, J., & Mason, C. (2013). BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, *9*(3), 1-32.

Ruan, H. H. (1998). Analysis and design considerations for high performance caches (Doctoral dissertation, Massachusetts Institute of Technology).

Frederickson, G. N. (1983). Implicit data structures for the dictionary problem. *Journal of the ACM (JACM)*, *30*(1), 80-94.

Flajolet, P., Poblete, P., & Viola, A. (1998). On the analysis of linear probing hashing. *Algorithmica*, *22*(4), 490-515.

Vitter, J. S. (1982). Implementations for coalesced hashing. *Communications of the ACM*, *25*(12), 911-926.

Pagh, R., & Rodler, F. F. (2004). Cuckoo hashing. *Journal of Algorithms*, *51*(2), 122-144.

Herlihy, M., Shavit, N., & Tzafrir, M. (2008). Hopscotch hashing. In *Distributed Computing: 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings 22* (pp. 350-364). Springer Berlin Heidelberg.

Celis, P., Larson, P. A., & Munro, J. I. (1985, October). Robin hood hashing. In *26th annual symposium on foundations of computer science (sfcs 1985)* (pp. 281-288). IEEE.

Guibas, L. J., & Sedgewick, R. (1978, October). A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)* (pp. 8-21). IEEE.

Adel′son-Velskii, G. M., & Landis, E. M. (1962). An algorithm for organization of information. In *Doklady Akademii Nauk* (Vol. 146, No. 2, pp. 263-266). Russian Academy of Sciences.

Sleator, D. D., & Tarjan, R. E. (1985). Self-adjusting binary search trees. *Journal of the ACM (JACM)*, *32*(3), 652-686.

Seidel, R., & Aragon, C. R. (1996). Randomized search trees. *Algorithmica*, *16*(4-5), 464-497.

Bayer, R., & McCreight, E. (1970, November). Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (pp. 107-141).

Comer, D. (1979). Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, *11*(2), 121-137.

Morrison, D. R. (1968). PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, *15*(4), 514-534.

Fredkin, E. (1960). Trie memory. *Communications of the ACM*, *3*(9), 490-499.

Weiner, P. (1973, October). Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)* (pp. 1-11). IEEE.

Gobeille, R. C., & Baskins, D. L. (2004). *U.S. Patent No. 6,735,595*. Washington, DC: U.S. Patent and Trademark Office.

Ekler

Ek.1 Veri yapısının eksiksiz implementasyonu: *https://github.com/Dost22/RDA*