T.C.
İNÖNÜ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ATLAMALI HALKA: DAİRESEL VE ATLAMALI LİSTE TEMELLİ YENİ BİR VERİ YAPISI

MUSTAFA AKSU

DOKTORA TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

Mayıs – 2016

Tezin Başlığı : Atlamalı Halka: Dairesel ve Atlamalı Liste Temelli Yeni Bir Veri Yapısı

Tezi Hazırlayan : Mustafa AKSU

Sınav Tarihi : 09.05.2016

Yukarıda adı geçen tez jürimizce değerlendirilerek Bilgisayar Mühendisliği Ana Bilim Dalında Doktora Tezi olarak kabul edilmiştir.

Sınav Jüri Üyeleri

Tez Danışmanı : **Prof. Dr. Ali KARCI** .....
İnönü Üniversitesi

Doç. Dr. Davut HANBAY .....
İnönü Üniversitesi

Doç. Dr. Mehmet Emin TAĞLUK .....
İnönü Üniversitesi

Yrd. Doç. Dr. Murat DEMİR .....
Muş Alparslan Üniversitesi

Yrd. Doç. Dr. Murat CANAYAZ .....
Yüzüncü Yıl Üniversitesi

Prof. Dr. Alaattin ESEN
Enstitü Müdürü

ONUR SÖZÜ

Doktora Tezi olarak sunduğum “Atlamalı Halka: Dairesel ve Atlamalı Liste Temelli Yeni Bir Veri Yapısı” başlıklı bu çalışmanın bilimsel ahlak ve geleneklere aykırı düşecek bir yardıma başvurmaksızın tarafımdan yazıldığını ve yararlandığım bütün kaynakların, hem metin içinde hem de kaynakçada yöntemine uygun biçimde gösterilenlerden oluştuğunu belirtir, bunu onurumla doğrularım.

Mustafa AKSU

ÖZET

Doktora Tezi

ATLAMALI HALKA: DAİRESEL VE ATLAMALI LİSTE TEMELLİ YENİ BİR VERİ YAPISI

Mustafa AKSU

İnönü Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

99 + xi sayfa

2016

Danışman: Prof. Dr. Ali KARCI

Atlamalı liste (skip list) veri yapısında bağlı listeler kullanılır. Katmanlı bir yapıdan oluşur ve en alt katmanda tüm düğümler bulunur; bu düğümler üst katmanlara doğru yarıya düşürülerek piramit şeklinde bir yapı oluşturulur. Böylece arama, ekleme, silme işlemlerinde kolaylık sağlanması amaçlanır. Bunun yanında bu veri yapısı daha da iyileştirilebilir.

Bu tez çalışmasındaki amacımız; Atlamalı liste veri yapısını analiz ederek bu veri yapısındaki problemleri tespit edip, tespit edilen problemleri çözerek atlamalı liste veri yapısında iyileştirmeler yapmak ve daha sonra bu iyileştirmeleri önereceğimiz yeni veri yapısına uygulamaktır. Atlamalı listedeki iyileştirmeler dikkate alınarak önerilen yeni veri yapısı atlamalı halka (skip ring), dairesel bağlı liste ve atlamalı liste veri yapılarından faydalanılarak oluşturulmuştur. Önerdiğimiz yeni veri yapısı koni şeklinde birbirine bağlı katmanlar halinde dairesel bağlı listelerden oluşur. Böylece N elemanlı bir atlamalı halka veri yapısında arama, ekleme, silme işlemlerinin zaman karmaşıklığı O(lgN) olur. Önerilen yeni veri yapısı uygulamalı olarak ikili arama ağaçları, kırmızı-siyah ağaçlar ve atlamalı liste veri yapıları ile kıyaslanmış sonuçları incelenmiştir. Ayrıca atlamalı halka temelli yeni bir sıralama ve arama algoritması önerilmiştir. Önerilen bu algoritmalar mevcut sıralama ve arama algoritmaları ile uygulamalı karşılaştırılmıştır.

Sonuç olarak, atlamalı liste ve dairesel bağlı listelerin özeliklerinden faydalanılarak geliştirilen atlamalı halka (skip ring) veri yapısı ağaç temelli bazı veri yapıları ile karşılaştırılmış iyi sonuçlar elde edilmiştir. Ayrıca sıralama (sorting), arama (searching) gibi her zaman güncel bazı alanlara etkin bir şekilde uygulanabilirliği gösterilmiştir.

ANAHTAR KELİMELER: Atlamalı Halka, Atlamalı Liste, Piramit Arama, Atlamalı Halka Sıralama, Veri yapıları ve Algoritmalar

ABSTRACT

Ph.D.Thesis

SKIP RING: A NEW DATA STRUCTURE BASED ON CIRCULAR AND SKIP LISTS

Mustafa AKSU

İnönü University
Graduate School of Natural and Applied Sciences
Departmant of Computer Engineering

99 + xi pages

2016

Supervisor: Prof. Dr. Ali KARCI

Linked lists are used in skip list data structure. Skip list data structure consists of layered structure and all nodes are in the lowest layer. These nodes are reduced by half towards upper layers and are formed a pyramid-shaped structure. Thus, it is aimed to enable searching, insertion and deletion. Besides, this data structure can be improved further.

The purpose of our study is to identify problems in data structure by analyzing skip list data structure, to make improvements in skip list data structure by resolving the identified problems and to implement and then to apply these improvements to the new data structure that we proposed. New data structure that we proposed by considering improvements in skip list is formed by utilizing skip ring, circular linked list and skip list data structure. New data structure that we proposed composes of circular linked list in cone-shaped cohesive layers. Thus, time complexity of search, insertion and deletion progress is O(lgN) in N-Element skip list data structure. Proposed new data structure was compared practically with binary search trees, red-black trees and skip list data structures and the results was evaluated. Moreover, a new sorting based on skip ring and search algorithm were proposed. These proposed algorithms were compared practically with current sorting and search algorithms.

Consequently, skip ring data structure which was improved by utilizing the properties of skip list and circular linked lists was compared with some tree-based data structures and good results were obtained. And also, effective applicability in some areas such as sorting and searching was demonstrated.

KEYWORDS: Skip Ring, Skip List, Pyramid Search, Skip Ring Sort, Data Structures and Algorithms

TEŞEKKÜR

Doktoramın her aşamasında bilgi ve tecrübeleriyle yardımcı olan ayrıca her türlü ilgi ve teşvikleriyle bana destek olan danışman hocam Sayın Prof. Dr. Ali KARCI'ya;

Çalışmalarım boyunca, bana destek olan Anabilim Dalındaki tüm değerli hocalarıma;

Çalışma yaptığım alandaki yabancı kaynakların çevirisinde, yorumlanmasında bana her türlü desteği sağlayan kardeşim Cuma AKSU ve arkadaşlarına;

Çalışmalarım sırasında her türlü fedakârlığa katlanan, her zaman desteğini esirgemeyen, uykusuz gecelerde beni yalnız bırakmayan eşim Gönül DOĞRU AKSU'ya, çocuklarıma ve tüm arkadaşlarıma

teşekkür ederim.

İÇİNDEKİLER

ÖZET i
ABSTRACT ii
TEŞEKKÜR iii
İÇİNDEKİLER iv
ŞEKİLLER LİSTESİ vii
ÇİZELGELER LİSTESİ ix
SİMGELER VE KISALTMALAR x
1. GİRİŞ **1**
1.1. Tezin Güncelliği 4
1.2. Tezin Amacı 5
1.3. Amaca Ulaşmak için Tezde Çözümlenen Problemler 5
1.4. Tezin Pratik Önemi 6
1.5. Tezde Yapılan Çalışmalar 6
2. KURAMSAL TEMELLER **8**
2.1. Veri Yapıları 8
2.1.1. Diziler 8
2.1.2. Listeler 9
2.1.3. Bağlı listeler 9
2.1.4. Ağaçlar 10
Ağaçlarda gezinti işlemi 11
2.1.5. Dengeli ikili ağaçlar 12
2.1.6. Kırmızı-siyah ağaç (red-black tree) veri yapısı 13
Kırmızı-siyah ağaçlarda döndürme işlemi 14
2.2. Algoritmalar 15
2.2.1. Algoritma analizi 16
2.2.2. Arama algoritmaları 18
2.2.3. Sıralama algoritmaları 19
2.3. İşletim Sistemlerinde Görev Zamanlayıcı Algoritmalar 20
2.3.1. Linux işletim sisteminde görev zamanlayıcı 20
3. ATLAMALI LİSTE VE YAPILAN İYİLEŞTİRMELER **23**
3.1. Atlamalı Listenin Oluşturulması 23
3.2. Atlamalı liste için seviye (level) oluşturma 26

3.3. Düğüm Arama 30
3.4. Düğüm Ekleme 32
3.5. Düğüm Silme 34
3.6. Atlamalı Listeyi Yeniden Düzenleme (Reorganization) Gereksinimi 36
4. YENİ VERİ YAPISI ATLAMALI HALKA-DAİRESEL ATLAMALI LİSTE (SKIP RING-CIRCULAR SKIP LIST) 42
4.1. Dairesel Bağlı Liste 42
4.2. Önerilen Veri Yapısı Atlamalı Halka (Skip Ring) 43
4.2.1. Düğüm arama 45
4.2.2. Düğüm ekleme 47
4.2.3. Düğüm silme 50
4.3. Atlamalı Halka (Skip Ring) Veri Yapısının Özellikleri 52
4.4. Atlamalı Halka (Skip Ring) Veri Yapısının Zaman Analizi 55
5. DENEYSEL BULGULAR VE TARTIŞMA 58
5.1. Örnek Uygulama 1: Atlamalı Halka (Skip Ring) ve Ağaç Veri Yapılarının (Binary Search Tree, Red-Black Tree) Karşılaştırılması 58
5.1.1. Ağaç veri yapıları (ikili arama ağaçları, kırmızı-siyah ağaçlar gibi) ve atlamalı halka (skip ring) veri yapısının performansının uygulamalı karşılaştırılması 59
5.2. Örnek Uygulama 2: Atlamalı Halka (Skip Ring) Veri Yapısı Temelli Sıralama İşlemi (Karşılaştırmalı Uygulama) 62
5.2.1. Atlamalı halka sıralama (skip ring sort) ile verilerin sıralanması 64
5.2.2. Atlamalı halka sıralama algoritmasının değerlendirmesi 69
5.3. Örnek Uygulama 3: Atlamalı Halka (Skip Ring) Veri Yapısı Temelli Yeni Arama Algoritması: Piramit Arama (Pyramid Search) 70
5.3.1. Piramit arama (pyramid search) ve ikili arama (binary search) 72
5.3.2. Piramit arama (pyramid search) algoritması ve diğer arama algoritmalarının uygulamalı karşılaştırılması. 76
5.3.3. Piramit arama algoritmasının değerlendirmesi 81
5.4. Örnek Çalışma 4: Fair Priority Scheduler (FPS): Atlamalı Halka Veri Yapısı Temelli Yeni Görev Zamanlayıcı Algoritması 81
5.4.1. Linux'ta görev zamanlayıcı algoritmaları 81
5.4.2. Yeni görev zamanlayıcı (process scheduler) algoritması 85
5.4.3. Yeni görev zamanlayıcı (process scheduler) algoritmasının değerlendirilmesi.. 87

6. SONUÇLAR VE ÖNERİLER..... 89
7. KAYNAKLAR..... 94
ÖZGEÇMİŞ..... 99

ŞEKİLLER LİSTESİ

Şekil 2.1. Bağlı Liste 10
Şekil 2.2. Tek Yönlü Bağlı Liste Düğüm Yapısı 10
Şekil 2.3. Ağaç Veri Yapısı 10
Şekil 2.4. (a) Kırmızı-Siyah Ağaç (b) Dengesiz Ağaç 14
Şekil 2.5. Ağacı Döndürme 15
Şekil 2.6. O(1) Görev zamanlayıcı 21
Şekil 2.7. CFS için Kırmızı-Siyah Ağaç 22
Şekil 3.1. Atlamalı listenin bağlı listelerden oluşturulması 23
Şekil 3.2. Atlamalı liste veri yapısı 24
Şekil 3.3. Atlamalı Liste (Gerçek Yapı) 25
Şekil 3.4. (a) İdeal atlamalı liste (b) Gerçekleşebilecek atlamalı liste 28
Şekil 3.5. Atlamalı liste veri yapısında düğüm arama 31
Şekil 3.6. (a) Atlamalı liste düğüm ekleme işlemi (b) Düğüm Eklenmiş Hali 34
Şekil 3.7. (a) Atlamalı liste düğüm silme işlemi (b) Düğüm silinmiş hali 35
Şekil 3.8. Düzensiz bir atlamalı liste 37
Şekil 3.9. Atlamalı liste (çökmüş hali)= Bağlı liste 37
Şekil 3.10. Şekil 3.9'daki atlamalı listenin ideal hali (Reorganized algoritması ile yeniden inşa edilmiş hali) 37
Şekil 3.11. Farklı P değerlerinin atlamalı liste veri yapısının yüksekliğine etkisi 41
Şekil 4.1. Dairesel Bağlı Listelerde Düğüm Yapısı 42
Şekil 4.2. Dairesel Bağlı Liste 42
Şekil 4.3. Atlamalı Halka (Skip ring) veri yapısının inşası 43
Şekil 4.4. Atlamalı Halka (Skip ring) (P=1/2 için) 44
Şekil 4.5. Atlamalı Halka (Skip ring) (P=1/4 için) 45
Şekil 4.6. Atlamalı halkada düğüm arama 46
Şekil 4.7. (a) Yeni düğüm ekleme işlemi (b) Düğüm eklenmiş hali 48
Şekil 4.8. (a) Düğüm silme işlemi (b) Atlamalı halkayı güncelleme 52
Şekil 4.9. Atlamalı halkada P=1/4 ve P=1/2 için düğümlerin seviyelere dağılımı. . 56
Şekil 4.10. Atlamalı halka veri yapısının ağaç şeklinde görünümü 56
Şekil 4.11. Atlamalı halka veri yapısının ağaç şeklinde analizi 57
Şekil 5.1. Atlamalı halka (Skip Ring), İkili arama ağaçları (Binary Search Tree) ve Kırmızı-siyah ağaç (Red-black tree) performans karşılaştırması 60

Şekil 5.2. Sıralı (A-Z) diziler üzerinde SR, RBT ve BST için performans karşılaştırması. 60
Şekil 5.3. Ters sıralı (Z-A) diziler üzerinde SR, RBT ve BST için performans karşılaştırması. 61
Şekil 5.4. Atlamalı Halka Sıralama (Skip ring Sort) (Adım adım) 64
Şekil 5.5. Karışık verilerde, O(NlgN) grubu bazı sıralama algoritmaları ile SR sıralama algoritmasının karşılaştırması 66
Şekil 5.6. Karışık verilerde, $O(N^2)$ grubu bazı sıralama algoritmaları ile SR sıralama algoritmasının karşılaştırması 67
Şekil 5.7. Karışık verilerde SR, RBT, ve BST sıralama algoritmalarının sıralama sürelerinin karşılaştırması. 68
Şekil 5.8. Atlamalı liste (Bağlı liste temelli veri yapısı) 71
Şekil 5.9. Atlamalı listenin gerçek yapısı 72
Şekil 5.10. Atlamalı Halka **(**Skip ring) (P=1/4 için) 72
Şekil 5.11. Piramit Arama (Pyramid search) (P=1/4 için; Şekil 5.10'da 'dive' iki defa, 'map' dört defa ve 'vary' üç defa aranıyor) 76
Şekil 5.12. Sıralı verilerde LS, BS ve PS Arama algoritmaları için Performans (Aranan eleman dizinin baş tarafına yakın) 78
Şekil 5.13. Sıralı verilerde LS, BS ve PS Arama algoritmaları için Performans (Aranan eleman dizinin ortalarında) 79
Şekil 5.14. Sıralı verilerde LS, BS ve PS Arama algoritmaları için performans (Aranan eleman dizinin sonlarına yakın) 79
Şekil 5.15. Sıralı verilerde BS ve PS Arama algoritmaları için Performans. 80
Şekil 5.16. Görevler için kırmızı-siyah ağaç ve hiyerarşi yapısı (CFS) 84
Şekil 5.17. Atlamalı halka veri yapısının SJF (Shortest Job First) zamanalayıcı algoritmasında kullanımı (Şekil 5.20 – Priority 0 level) 84
Şekil 5.18. Atlamalı halka yapısının öncelik temelli zamanlayıcı (priority based scheduling) algoritmasında kullanımı. 85
Şekil 5.19. Atlamalı halka veri yapısının RR (Round-Robin) zamanlama algoritmasında kullanımı (Şekil 5.20 – Priority 0 seviyesi) 85
Şekil 5.20. Atlamalı halka (liste) temelli yeni görev zamanlayıcı için görevlerin yerleşimi (FPS-Fair Priority Scheduler) 86
Şekil 5.21. Atlamalı halka veri yapısının FPS (Fair Priority Scheduler) algoritmasına göre görevleri yürütmesi (Şekil 5.20 için) 87

ÇİZELGELER LİSTESİ

Çizelge 2.1. Bazı fonksiyonların Büyük O gösterimi 17
Çizelge 3.1. Atlamalı liste işlemlerinin zaman karmaşıklığı 26
Çizelge 3.2. Rastgele eklenen 16 düğümün seviyelere dağılımı..... 29
Çizelge 3.3. Rastgele eklenen 25 düğümün seviyelere dağılımı..... 30
Çizelge 3.4. P eşik değeri 0.1 için..... 39
Çizelge 3.5. P eşik değeri 0.25 için..... 40
Çizelge 3.6. P eşik değeri 0.5 için..... 40
Çizelge 3.7. P eşik değeri 0.75 için..... 40
Çizelge 3.8. P eşik değeri 0.9 için..... 40
Çizelge 5.1. Rastgele üretilen diziler için BST, RBT ve SR oluşturma süreleri..... 59
Çizelge 5.2. Sıralı (A-Z) diziler kullanarak BST, RBT ve SR oluşturma süreleri... 60
Çizelge 5.3. Ters sıralı (Z-A) diziler kullanarak BST, RBT, SR oluşturma süreleri61
Çizelge 5.4. 1000-200000 elemanlı karışık (random) veri kümesinde sıralama..... 66
Çizelge 5.5. 1000-100000 elemanlı sıralı (A-Z) veri kümelerinde sıralama algoritmalarının çalışma sürelerinin karşılaştırması..... 66
Çizelge 5.6. 1000-100000 elemanlı ters sıralı (Z-A) verilerde sıralama süreleri.... 67
Çizelge 5.7. Karışık verilerde ağaç veri yapısı temelli sıralama süreleri. 68
Çizelge 5.8. 1000-100000 elemanlı ters sıralı (A-Z) verilerde sıralama süreleri.... 68
Çizelge 5.9. 1000-100000 elemanlı ters sıralı (Z-A) verilerde sıralama süreleri.... 69
Çizelge 5.10. Şekil 5.10'daki düğümlerin aranma frekanslarına göre seviyelere yerleştirilmesi. 73
Çizelge 5.11. Şekil 5.11'deki düğümlerin aranma frekanslarına göre seviyelere yerleştirilmesi 74
Çizelge 5.12. Sıralı verilerde LS, BS ve PS Arama algoritmaları için performans karşılaştırması (Aranan eleman dizinin baş tarafına yakın) 77
Çizelge 5.13. Sıralı verilerde LS, BS ve PS Arama algoritmaları için Performans karşılaştırması (Aranan eleman dizinin ortalarında) 78
Çizelge 5.14. Sıralı verilerde LS, BS ve PS Arama algoritmaları için performans karşılaştırması (Aranan eleman dizinin sonlarına yakın) 78
Çizelge 5.15. Sıralı verilerde BS ve PS Arama algoritmaları için Performans..... 80
Çizelge 5.16. Görevler ve Gerekli Süre (Processes and required time)..... 86

SİMGELER VE KISALTMALAR

log N	N sayısının 10 tabanındaki değeri
$\log_2 N$	N sayısının 2 tabanındaki değeri
lg	2 Tabanında logaritma
P	Seviye Oluşturmada [0-1) arası olasılıksal değer
B-Tree	B ağaçları
FPS	Adil öncelikli görev zamanlayıcı (Fair Priority Scheduler)
O	Büyük O notasyonu, En kötü durumu ifade eder
CFS	Tamamen adil zamanlayıcı (Completely Fair Scheduler)
h	Atlamalı liste, atlamalı halka, ağaç veri yapılarının yüksekliği
Θ	Büyük teta notasyonu, Ortalama durumu ifade eder
Ω	Büyük omega notasyonu, En iyi durumu ifade eder
c, C	Fonksiyonlarda sabit terim
N^2	N'in kare üstel değeri
®	Tescil
O(1)	O(1) görev zamanlayıcı, Ayrıca sabit bir süreyi ifade eder
Dev-C++	C derleyicisi
ms	Mili saniye
GB	Gigabyte
Ghz	Giga hertz
Z^+	Pozitif tamsayılar
$\sum$	Toplam sembolü
$\supseteq$	Kapsama işareti
$\lfloor . \rfloor$, $\lceil . \rceil$	Taban ve tavan fonksiyonu
T(N)	İşlem süresi
SR	Atlamalı halka (Skip Ring)
RBT	Kırmızı-siyah ağaç (Red-black tree)
BST	İkili arama ağacı (Binary search tree)
N	Eleman (Düğüm) sayısı
LS	Doğrusal arama (Lineer search)
BS	İkili arama (Binary search)
PS	Piramit arama (Pyramid search)

FIFO	İlk giren ilk çıkar (First-In-First-Out)
SJF	En kısa görev ilk iletilecek (Shortest-Job-First)
RR	Round-Robin görev zamanlayıcı
rb	Kırmızı-siyah (red-black)

1. GİRİŞ

Veri yapıları ve algoritmalar bilgisayar bilimlerinin birçok alanında doğrudan veya dolaylı yollardan kullanılmaktadır. Birçok problemin çözümünde değişik veri yapıları ve bunlara ait algoritmalar kullanılmaktadır. Zaman zaman mevcut veri yapılarının ihtiyaçlara cevap verememesinden veya işlem, zaman, donanım gibi kısıtlamalardan dolayı yeni veri yapılarına ve algoritmalara ihtiyaç duyulmuştur. Bazen dinamik, bazen statik bir yapıya ihtiyaç olduğundan farklı veri yapıları ve algoritmalar ortaya çıkmıştır [1]. Mesela ortak özellik taşıyan veriler topluluğu statik olarak önceden rezerve edilmiş sabit bir dizide tutulabilir. Genellikle rezerve edilen bu dizinin boyutu ne kadar veri tutulacağı önceden belli olmadığından büyük olmaktadır. Böyle bir durumda kaynaklar lüzumsuz yere meşgul edilecektir. Eğer bu veriler için dinamik bir yapı olan bağlı listeler veya ağaç yapıları kullanılırsa önceden bellek rezerve etmeye gerek kalmayacaktır. Çünkü bu tür veri yapılarında eleman eklendikçe bağlı listenin ya da ağaç veri yapısının kullandığı bellek boyutu artacaktır. Yani lüzumsuz yere önceden sabit bir veri topluluğu için daha fazla yeri statik olarak ayırmaya gerek kalmayacaktır. İkinci bir örnek ise, bir veri topluluğu üzerinde hem ileri yönde hem de geri yönde gezinmek gerekiyorsa bağlı listeler yerine çift yönlü bağlı listeler kullanılmalıdır. Bazı durumlarda ise, işlem hızından dolayı bir veri yapısı ya da algoritma diğerine tercih edilmektedir. Mesela arama işleminin hızlı gerçekleşmesi için bağlı listeleri kullanma yerine atlamalı liste (skip list) tercih edilmektedir [2, 3]. Bir başka örnek ise, bir veri kümesini sıralamak için seçmeli sıralama (selection sort) yerine daha hızlı olan hızlı sıralama (quick sort) veya birleştirerek sıralama (merge sort) algoritmalarından biri tercih edilebilir.

Bütün bu durumlar göz önünde bulundurulduğunda, yeni veri yapıları ve algoritmalar ortaya çıkmaya devam edecektir [1].

Zamanla ihtiyaçlara göre ortaya çıkan veri yapılarına örnek olarak; bağlı listeler, B-ağaçları, atlamalı liste (skip list), dinamik tablolar, splay veri yapıları, çizgeler (graph) v.b. verilebilir. Algoritmalara örnek olarak ise; seçmeli sıralama (selection sort), hızlı sıralama (quicksort), birleştirerek sıralama (mergesort), yığın sıralama (heapsort) gibi sıralama algoritmaları, B-ağacı algoritmaları, optimizasyon algoritmaları, işletim sistemlerindeki disk erişim, bellek yönetim, görev zamanlayıcı algoritmaları verilebilir.

Atlamalı liste veri yapısı, Pugh [4] tarafından dengeli ağaçlara alternatif olarak önerilen bağlı liste temelli bir veri yapısıdır. Dengeli ağaçlarda zorunlu bir dengeleme işlemi gerçekleştirilirken, atlamalı listeler olasılıksal bir dengeleme kullanır. Bunun sonucu olarak, atlamalı listedeki ekleme ve silme algoritmaları dengeli ağaçlardaki eşdeğer algoritmalardan daha sade ve önemli ölçüde hızlıdır. Pugh tarafından önerilen bu veri yapısının, analizi ve iyileştirilmesi adına 1989'dan günümüze kadar bir takım çalışmalar yapılmıştır.

Pugh [5] yaptığı bu çalışmada, atlamalı listelere kısa bir değinmiş, sıralı bağlı listelerin eş zamanlı yürütülmesinde basit metotlar önermiştir. Bu metotların doğruluğunu ispatlamıştır ve atlamalı listelerin eşzamanlı gerçekleştirilmesinde bu metotların nasıl geliştirilebileceğine ilişkin basit ve etkili algoritmalar önermiştir.

Pugh [6], atlamalı listelerin dengeli ağaçlar gibi çok yönlü olduğunu göstermiştir. Pugh bu çalışmasında atlamalı listelerde arama, birleştirme ve bölme algoritmalarını tanımlamış ve analiz etmiştir. Ayrıca atlamalı liste işlemlerini kullanarak doğrusal liste işlemlerini gerçekleştirmiştir. Bu işlemler için atlamalı liste algoritmaları dengeli ağaç çeşitlerinden daha hızlı ve sadedir. Atlamalı listeler için önerilen birleştirme algoritması dengeli ağaçlar için daha önceden tanımlanan birleştirme algoritmasından daha iyi asimtotik zaman karmaşılığına sahiptir.

Herlihy et al. [7] atlamalı liste veri yapısı temelli sadeliği ve ölçeklenebilirliği ön plana çıkan eş zamanlı atlamalı liste algoritması önermiştir. Bu algoritma düğüm ekleme, silme işleminden önce kilitleme yapmadan arama yapar, kısa kilit-esaslı doğrulamayı müteakiben daha iyi eşzamanlama sağlar.

Kirschenhofer et al. [8] atlama listeleri için optimize edilmiş bir arama algoritması analizini yapmışlardır. Herlihy et al. [9] atlamalı listede arama, ekleme ve silme işlemleri için daha basit, anlaşılır ve iyi sonuçlar veren algoritmalar önermişlerdir. Devroye [10] çalışmasında, atlamalı listenin bir ikili ağaç şeklinde gösterimini kullanarak atlamalı listedeki yaprakların (düğüm) yol uzunluklarını elde etmek için limit kuralını kullanmıştır. Papadakis [11] ise, yaptığı çalışmada olasılıksal atlamalı liste algoritmalarının analizini yapmıştır. Papadakis [12] et al. yaptıkları çalışmada, atlamalı listede (skip list) ortalama arama ve güncelleme maliyetini incelemiştir. Kirschenhofer ve Prodinger [13], olasılıksal atlamalı

listelerde arama maliyetinin analizini, toplam arama maliyeti veya yol uzunluğunun asimptotik analizini gerçekleştirmek suretiyle, farklı bir şekilde gerçekleştirmişlerdir.

Poblete et al. [14] binom dönüşüm teorisinin, Pugh tarafından önerilen olasılıksal bir veri yapısı olan atlamalı listenin performansını analiz etmek için, nasıl uygulanacağını göstermiştir.

Munro et al. [15] olasılıksal atlamalı liste yerine deterministik atlamalı liste yapısı önermişlerdir. Böylece logaritmik (log N) arama, ekleme ve silme maliyetini garanti etmeyi amaçlamışlardır. 1-2 atlamalı liste ve 1-2-3 atlamalı liste türleri önermiş olup, log N düzeyindeki bir yüksekliği sağlamaya çalışmışlardır.

Aksu et al. [1] yaptıkları çalışmada, atlamalı liste (skip list) veri yapısında tutarsız seviye üretme sorununa çözüm önermiştir. Bu sorun, atlamalı liste (skip list) veri yapısındaki toplam düğüm sayısından level=$\log_2$(nodecount) formülü ile ideal seviye üretilerek çözülmüştür. Böylece atlamalı liste (skip list) oluşturulurken veya düğüm eklenirken çok yüksek seviyelerin oluşturulması sorunu ortadan kalkmıştır.

Atlamalı listede rastgele seviye üretme probleminin ele alındığı makalede [16], farklı "P" eşik değerlerinin (0.1, 0.25, 0.5, 0.75, 0.9 gibi) performansı nasıl etkilediği ele alınıp çözüm önerilmiştir. Atlamalı liste veri yapısındaki yüksek seviye üretme problemi, optimum P eşik değeri bulunarak çözülmüştür. Böylece, atlamalı liste veri yapısı için P eşik değerlerine bağlı olarak ideal seviyeler oluşturulmuştur.

Atlamalı liste (skip list) veri yapısından faydalanılarak değişik veri yapıları ve algoritmalar geliştirilmiştir. Atlamalı çizgeler (Skip graphs) [17], türdeş ağlarda (peer to peer) etkin arama yapmak için tasarlanmış atlamalı liste temelli bir veri yapısıdır. Atlamalı B-ağaçları (Skip B-trees) [18], atlamalı çizgelerin (skip graphs) avantajı ile B-ağaçlarının (B-tree) özellikleri birleştirilerek oluşturulmuş bir veri yapısıdır. Böylece, atlamalı B-ağacı (Skip B-Tree) veri yapısı etkin bir şekilde arama, ekleme ve silme işlemi sağlar. Atlamalı ağaçlar (Skip trees) [19], atlamalı listeye alternatif olarak logaritmik zaman karmaşıklığını azaltmak için geliştirilmiştir. Atlamalı asansör (Skip lift) [20], atlamalı liste temelli kırmızı-siyah ağaçlara alternatif olasılıksal bir veri yapısı olarak önerilmiştir.

Öncelikli arama (Priority Search) [21], atlamalı liste temelli yeni bir arama algoritması olarak önerilmiştir. Bu arama algoritmasında, aranan veriler her arama sonrasında atlamalı liste veri yapısında bir üst seviyeye çıkarılmaktadır. Böylece aranan veriler arama sıklığına göre piramit şeklindeki yapının tepesinden altlara doğru yerleştirilmektedir. Bunun sonucu olarak daha verimli bir arama sağlanması amaçlanmaktadır.

Tiara [22], türdeş ağların kendi kendini dengeleyerek sürekliliğini sağlayan deterministik atlamalı liste ve atlamalı çizgeyi kullanan bir algoritmadır. Bu algoritmanın tamamen deterministik olması iyi bir performans sağlar. Ayrıca logaritmik bir arama ve topolojik bir güncellemeye imkan verir.

Atlamalı liste temelli öncelik kuyrukları, Lotan ve Shavit [23] tarafından önerilmiştir. Bu çalışmada, geniş ölçekli çoklu-işlemcili (yüzlerce işlemci ile çalışan) sistemlerin eşzamanlı ölçeklenebilir öncelik kuyruklarının tasarım problemi ele alınmıştır. Öncelik kuyrukları modern çoklu-işlemcili algoritmaların tasarımında temel bir göreve sahiptir.

Bu tez çalışmasında mevcut veri yapıları ve algoritmalar incelenip yeni bir veri yapısı ve bu veri yapısına ait algoritmalar oluşturulmuştur.

1.1. Tezin Güncelliği

Günümüzde veri yapıları ve algoritmalar birçok problemin çözümünde doğrudan ya da dolaylı yollardan kullanılmaktadır. Gün geçtikçe işlenecek veri miktarı artmaktadır. Artan bu verileri saklamak ve işlemek için değişik veri yapılarına ihtiyaç vardır. Ayrıca bu verileri işleyip bunlardan sonuçlar elde etmek için değişik çözüm yöntemlerine, yani algoritmalara ihtiyaç vardır. Bu açıdan bakıldığında veri yapıları ve bunları işleme yöntemi olan algoritmalar her zaman güncel bir konu olarak hayatımızda yer almaya devam edecektir. Ortaya çıkan ya da var olan problemleri daha etkin bir şekilde çözmek için yeni algoritmalar ve veri yapıları geliştirilecektir. Bu bağlamda önerilen yeni veri yapısı atlamalı halka (skip ring), her zaman güncel bir konu olan veri kümesini sıralama ve veri kümesi üzerinde arama işlemi için kullanılabilmektedir.

1.2. Tezin Amacı

Tezde yeni bir veri yapısı önerilip, farklı uygulamalarının gerçekleştirilmesi amaçlanmıştır. Öncelikle, atlamalı liste (skip list) veri yapısı incelenip problemlerin tespit edilerek iyileştirmeler yapılması ve sonuçların uygulamalı olarak gözlemlenmesi amaçlanmıştır. Bu sonuçlardan hareketle dairesel bağlı liste (circular linked list) ve atlamalı liste (skip list) veri yapısı temelli yeni veri yapısı, atlamalı halka (skip ring), önerilip atlamalı liste veri yapısındaki iyileştirmeler ve elde edilen bulguların önerilen yeni veri yapısına uyarlanması amaçlanmaktadır.

1.3. Amaca Ulaşmak için Tezde Çözümlenen Problemler

- Atlamalı liste veri yapısı incelenerek bazı problemli tarafları tespit edilip çözümler önerilmiştir.
- Dairesel bağlı liste ve atalamalı liste veri yapılarının özelliklerinden faydalanarak, yeni bir veri yapısı önerilmiş ve bu veri yapısına ait arama, ekleme, silme algoritmaları oluşturulmuştur. Yeni veri yapısına dünya literatüründe kullanılması amacıyla skip ring (atlamalı halka) adı verilmiştir.
- Önerilen yeni veri yapısının zaman karmaşıklık analizi yapılmıştır.
- Ağaç veri yapıları, özellikle ikili arama ağaçları ve dengeli ağaç türü olan kırmızı-siyah ağaçlar incelenmiştir. Önerilen veri yapısının, ikili arama ağaçları ve kırmızı-siyah ağaçlarla performans kıyaslaması yapılmıştır.
- Önerilen veri yapısının uygulanabileceği alanlardan bazıları tespit edilmiştir. Bu kapsamda atlamalı halka temelli yeni bir sıralama ve piramit şeklinde yeni bir arama algoritmasının esasları belirlenmiştir.
- Esasları belirlenen atlamalı halka temelli sıralama algoritması (skip ring sort) oluşturulmuştur. Geliştirilen uygulama ile sıralama işlemi gerçekleştirilmiş ve farklı sıralama algoritmaları ile kıyaslama yapılmıştır.
- Atlamalı halka (skip ring) veri yapısının katmanlı halinden faydalanılarak yeni bir arama algoritması (piramit arama-pyramid search) önerilmiştir. Bu arama algoritmasına ait uygulama gerçekleştirilmiştir.
- Atalamlı halka (skip ring) veri yapısını kullanan yeni bir görev zamanlayıcı algoritması önerilmiştir.

1.4. Tezin Pratik Önemi

Önerilen yeni veri yapısı, literatürde var olan ağaç temelli bazı veri yapıları ile uygulamalı olarak karşılaştırılmıştır. Atlamalı halka veri yapısı temelli bir sıralama algoritması önerilmiş ve önerilen algoritma diğer bazı sıralama algoritmaları ile uygulamalı karşılaştırılmıştır. Ayrıca atlamalı halka temelli öncelikli arama algoritması önerilmiş ve önerilen algoritma sıralı arama, ikili arama algoritmaları ile karşılaştırılmıştır.

1.5. Tezde Yapılan Çalışmalar

Doktora tez çalışması altı bölümden oluşmaktadır.

1. Bölümde bu tezin konusu ile ilgili literatürde yapılan çalışmalar vurgulanmıştır.

2. Bölümde bu tez çalışmasında gerekli olan temel veri yapıları anlatılmış olup bu kapsamda diziler, listeler, ağaç veri yapıları incelenmiştir. Ayrıca bu bölümde farklı algoritmalar ve algoritma analizi üzerinde durulmuş algoritma analiz yöntemleri anlatılmıştır. Notasyonlar incelenip üzerinde durulmuştur.

3. Bölümde bağlı liste temelli atlamalı liste (skip list) veri yapısı ve özellikleri incelenmiştir. Bu veri yapısı üzerinde bir takım iyileştirmeler yapılmış ve bu iyileştirmelerin atlamalı liste (skip list) veri yapısı üzerindeki performans etkileri uygulamalı olarak gerçekleştirilip sonuçları gözlenmiştir.

4. Bölümde dairesel bağlı listeler ele alınıp incelenmiştir. Daha sonra atlamalı liste ve dairesel bağlı liste özelliklerinden faydalanılarak yeni bir veri yapısı ve bu veri yapısına ait algoritmalar geliştirilmiştir. Yeni veri yapısına atlamalı halka (skip ring) adı verilmiştir. Ayrıca bu bölümde atlamalı halka veri yapısının işleyişi, algoritmaları ve özellikleri belirlenmiştir. Atlamalı listeden faklı ve üstün yönleri üzerinde durulmuştur.

5. Bölümde önerilen atlamalı halka veri yapısının uygulamaları gerçekleştirilmiştir. Bu bölümde ilk uygulama olarak dengeli ağaçlardan kırmızı-siyah ağaçlar (red-black tree), ikili arama ağaçları (binary search tree), atlamalı liste (skip list) ve yeni veri yapısı atlamalı halka (skip ring) uygulamalı olarak

karşılaştırılmıştır. Bu veri yapıları, oluşturulma süreleri yönünden değişik veri kümeleri kullanılarak kıyaslanmıştır.

İkinci uygulama olarak ise, atlamalı halka (skip ring) veri yapısı temelli bir sıralama algoritması önerilmiştir. Bundan dolayı yeni sıralama algoritmasına atlamalı halka sıralama (skip ring sort) ismi verilmiştir. Geliştirilen bir uygulama ile yeni önerilen atlamalı halka sıralama algoritması, $O(N^2)$ grubu algoritmalar ve O(NlgN) grubu algoritmalar ile kıyaslanmıştır.

Üçüncü uygulama ise, atlamalı halka (skip ring) veri yapısı temelli önerilen yeni arama algoritmasına ait uygulamadır. Yeni arama algoritmasına piramit arama (pyramid search) ismi verilmiştir. Gerçekleştirilen uygulamada önerilen yeni arama algoritması, ikili arama (binary searching) ve doğrusal arama (linear searching) algoritmaları ile kıyaslanmıştır. Elde edilen sonuçlar grafik ve çizelgelerle sunulmuştur.

Dördüncü olarak, işletim sistemlerinde görev zamanlayıcı (process scheduler) üzerinde durulmuş özellikle Linux işletim sisteminde kullanılan O(1) ve CFS (Completely Fair Scheduler- Tamamen Adil Zamanlayıcı) görev zamanlayıcı algoritmaları araştırılmıştır. CFS algoritması kırmızı-siyah ağaçları kullanmaktadır. Bu algoritmada kırmızı-siyah ağaçlar yerine, yeni veri yapısı atlamalı halka etkin bir şekilde kullanılmıştır. Ayrıca bu bölümde atlamalı halka veri yapısını kullanan yeni bir görev zamanlayıcı algoritması önerilmiştir. Bu algoritmaya FPS (Fair Priority Scheduler - Adil Öncelikli Zamanlayıcı) adı verilmiştir.

6. Bölümde bu tez çalışmasından elde edilen sonuç ve öneriler yer almaktadır.

2. KURAMSAL TEMELLER

2.1. Veri Yapıları

Veri yapıları bilgisayar bilimlerinde çok yaygın olarak kullanılır. Veri yapıları tek bir veriyi saklamak için kullanılan değişkenden başlayıp, birden fazla veriyi saklamak için kullanılan diziler, listeler, ağaçlar, çizgeler (graphs) gibi veri tutma ve işleme yöntemlerinin tümüdür. Kısaca veri veya verilerin bellekte tutulma ve işlenme şekline göre değişik veri yapıları kullanılır. Veri yapıları genellikle birbirlerinden türetilirler.

Algoritmalar, problemlerin çözümü için kullanılan yöntemlerdir. Algoritmalar bilgisayar bilimlerinde birçok alanda kullanılmaktadır. Örneğin verileri sıralama için sıralama algoritmaları, arama için arama algoritmaları, optimizasyon için kullanılan optimizasyon algoritmaları bulunmaktadır. Birçok arama, sıralama ve optimizasyon algoritması vardır. Bir algoritma doğru yerde kullanılırsa iyi sonuçlar elde edilir.

Algoritmaların etkinliğini belirlemek için durum analizleri yapılır. En iyi durum analizi, ortalama durum analizi, en kötü durum analizi bu analizlerden bazılarıdır.

2.1.1. Diziler

Diziler, elemanları aynı türden olan ve bellekte art arda sıralanmış bir biçimde bulunan veri yapılarıdır. Dizilerin iki önemli temel özelliği vardır:

- Diziler birden fazla elemandan oluşur ve bu elemanlar aynı türden olmak zorundadır.
- Dizi elemanları bellekte art arda sıralanmış bir biçimde tutulurlar.

Dizilerin her bir elemanının indis değeri vardır. Bu indis değeri bilinen dizi elemanlarına O(1) zaman karmaşıklığında, yani çok hızlı bir şekilde erişilir. Erişim dizinin tüm elemanlarına aynı sürede gerçekleşir. Şüphesiz dizi elemanlarına çok hızlı erişilmesi onların belleğe ardışıl yerleştirilmelerinden kaynaklanmaktadır. Eğer aranan bir elemanın indis değeri bilinmiyorsa erişim süresi artar. Diziden eleman silme, ekleme işlemleri çok zaman alır. Diziye eleman eklerken ve diziden eleman silerken kendinden sonra gelen elemanları kaydırma işlemi söz konusu olduğundan çok zaman alır [24, 25].

Diziler boyut sayısına göre; tek boyutlu diziler, iki boyutlu diziler ve çok boyutlu diziler şeklinde isimlendirilirler.

2.1.2. Listeler

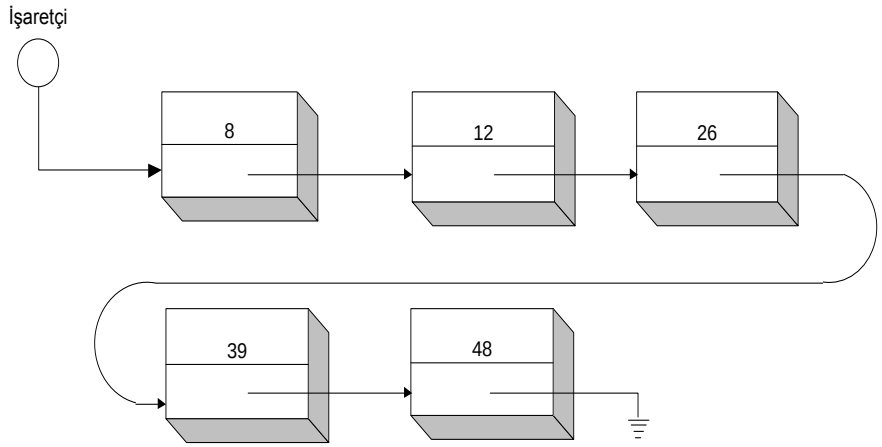
Günlük yaşamda listeler pek çok yerde kullanılmaktadır. Alışveriş listeleri, adres listeleri, davetli listeleri gibi. Bilgisayar bilimleri alanında özellikle programlamada listeler yararlı ve yaygın olarak kullanılan veri yapılarındandır. Programlama açısından liste, aralarında doğrusal ilişki olan veriler topluluğu olarak görülebilir. Veri yapılarında değişik biçimlerde listeler kullanılmakta ve üzerlerinde değişik işlemler yapılmaktadır [26].

2.1.3. Bağlı listeler

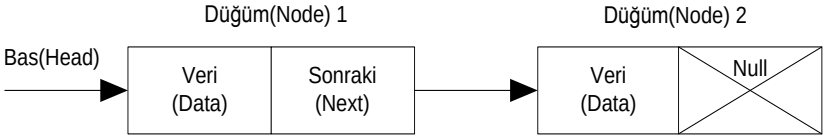
Statik veri yapılarına örnek olarak diziler, statik yığıtlar, statik kuyruklar verilebilir. Bağlı listelerde, veriler belli bir sırada yerleştirilirler ve o ana kadar kaç tane veri geldiyse, bağlı listenin boyutu odur. Programın icrası sırasında bağlı listenin boyutu değişebilmektedir. Bundan dolayı bağlı listelere dinamik veri yapıları denir. Dinamik veri yapıları hafızanın etkin bir şekilde kullanılması gerektiği durumlarda kullanılması gereken önemli yapılar arasındadır. Bağlı listelerin tanımlanması için işaretçi veri tipleri kullanılır. İşaretçi bir veri tipi olup hafıza hücrelerinin adresini tutan değişkenlerdir. Bu değişkenler başka bir değişkenin bulunduğu hafıza adresini tuttuğundan dolayı bu yolla değişkenler birbirine bağlanır. Bir zincirin halkaları gibi başlangıç değişkenden yola çıkarak diğer değişkenlerin değerleri elde edilebilir.

Bağlı listeler, düğüm adı verilen veri parçacıklarının bir araya getirilip birbirlerine bağlanmasıyla oluşturulan bir veri yapısıdır (Şekil 2.1). Bağlı listelere erişim için bir başlangıç düğümüne (işaretçi) ihtiyaç vardır. Her düğüm veri(ler) ve bir sonraki düğüme bağlantı (sonraki) bileşenlerinden oluşur. Ayrıca bağlı listenin bittiğini gösteren bir sonlandırıcıya ihtiyaç vardır [27, 28] (Şekil 2.2).

Bu sıralı yapı herhangi bir pozisyona düğüm ekleme, silme ve düğüm arama gibi işlemlerde etkin bir şekilde kullanılır [29]. Bağlı listeler, çift yönlü bağlı listeler, dairesel bağlı listeler, atlamalı liste (skip list), kuyruk gibi farklı şekillerde oluşturulup, farklı amaçlar için kullanılabilir. Bilgisayar bilimlerinde yaygın olarak kullanılan yığıt, kuyruk, atlamalı çizge (skip graph) gibi veri yapıları da bağlı listelerden oluşur.



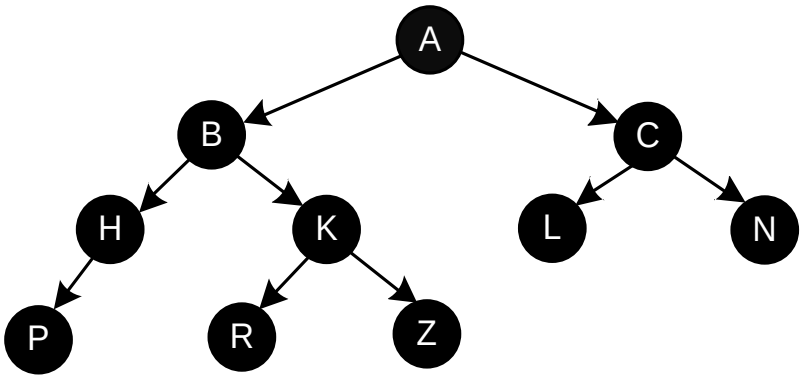
Şekil 2.1. Bağlı Liste



Şekil 2.2. Tek Yönlü Bağlı Liste Düğüm Yapısı

2.1.4. Ağaçlar

Ağaç veri yapıları (Şekil 2.3) doğrusal olmayan belirli niteliklere sahip ağaç biçiminde oluşturulan ve kök, dal ve yaprak şeklinde düzenlenen veri yapılarıdır. Ağaç veri yapısı kısıtlamaların az olduğu ve birçok probleme kolaylıkla uyarlanabilecek bir yapı olduğundan birçok alanda kullanılmaktadır.



Şekil 2.3. Ağaç Veri Yapısı

Ağacın her bir elemanına düğüm (node) adı verilir. Şekil 2.3'teki ağaç yapısında A, C, K birer düğümdür. Ağacın en üstteki düğümüne kök (root) adı verilir. Bu ağaç yapısında A düğümü köktür.

İki düğüm arasındaki bağa dal adı verilir. Örneğin Şekil 2.3'teki ağaçta A ile B, K ile R arasında dal vardır [30].

Bir düğüme 1. dereceden bağlı olan düğümlere o düğümün çocukları (child) denir. Şekil 2.3'teki ağaçta B ve C, A'nın; R ve Z, K'nin çocuklarıdır. Bir düğüm, kendisine bağlı ilk düğümlerin atası (parent)'dır. Yani B ve C'nin atası A, P'nin atası H'dir.

Sol ve sağ bağı boş olan düğümlere yaprak (leaf) adı verilir. Şekil 2.3'teki ağaç yapısında P, R, Z, L, N yapraklardır [31].

Aynı ataya sahip düğümlere kardeş düğüm adı verilir. Örnek olarak L ile N ve H ile K kardeştir.

Ağaçlar özelleştirilerek farklı isimler ve özellikler taşıyacak şekle getirilmiştir. Örneğin ikili ağaçlar, ikili arama ağaçları, kırmızı-siyah ağaçlar, splay ağaçları v.b.

Ağaçlarda gezinti işlemi

Ağaç veri yapıları üzerinde herhangi bir düğüme ya da düğümlere erişmek (arama, değiştirme, silme, sıralama gibi) için ağacın düğümlerini gezmek (traverse) gerekmektedir. Bir ağacı en çok bilinen şu üç yöntemle [25, 28, 30, 32, 33] gezebiliriz:

- Kökten başlayarak (Pre-order-Önce kök)
- Sondan başlayarak (Post-order-Sonra kök)
- Sıralı (In-order-Ortada kök)

Kökten başlayarak (pre-order-önce kök) gezintide ilk olarak köke uğranır. Daha sonra sol alt ağaca geçilir ona ait kökten başlanarak gezintiye devam edilir. Sol alt ağaç bitince ağacın köküne bağlı sağ alt ağaç için benzer adım tekrarlanır. Sağ alt ağaçta kökten başlayarak dolaşılır. Şekil 2.3'teki ağaç için önce-kök (pre-order) gezinti sonucu A-B-H-P-K-R-Z-C-L-N düğümlerine erişilerek gezilir.

Sondan başlayarak (post-order-sonra kök) gezintide ilk olarak en soldaki alt ağacın soldaki yaprağına (P) uğranır. Daha sonra aynı ağacın sağ yaprağına uğranır, daha sonra bu alt ağacın kökü (H) alınır. En soldaki alt ağaçtan yukarı doğru aynı işlemler devam ettirilir. Sol alt ağaç bitince ağacın köküne bağlı sağ alt ağaç için benzer adım tekrarlanır. Şekil 2.3'teki ağaç için sonra-kök (postorder) gezinti sonucu P-H-R-Z-K-B-L-N-C-A düğümlerine erişilerek gezilir.

Sıralı (in-order-ortada kök) gezintide ilk olarak en soldaki alt ağaç sol yaprak-kök-sağ yaprak şeklinde gezilir. Yukarı doğru aynı işlemler köke kadar devam eder. Daha sonra kök sonrada sağ ağaca geçilir. Sağ ağacın en solundan başlanılıp aynı işlemler tekrarlanır. Şekil 2.3'teki ağaç için ortada-kök (in-order) gezinti sonucu P-H-B-R-K-Z-A-L-C-N düğümlerine erişilerek gezilir. Sıralı gezinti ikili arama ağaçlarında ve kırmızı-siyah ağaçlarda verileri sıralama amaçlı da kullanılabilir.

2.1.5. Dengeli ikili ağaçlar

İkili ağaçlarda her bir düğümün hiç çocuğu olmayabilir, 1 çocuğu olabilir ya da en fazla 2 çocuğu olabilir. İki çocuklu bir düğümün solundakine sol çocuk, sağındakine ise sağ çocuk denir. Bir ikili ağaçta her bir n düğümü şu özellikleri taşır [34]:

- Bir n düğümünün değeri, kendine bağlı sol alt ağaçtaki tüm düğümlerin değerlerden büyüktür;
- Bir n düğümünün değeri, kendine bağlı sağ alt ağaçtaki tüm düğümlerin değerlerden küçüktür;

Ağaçta bir düğümü (elemanı) aramak için ilk olarak aranan eleman ağacın kökü ile karşılaştırılacaktır. Eğer bu eleman kök düğümün değerinden büyükse, aranan eleman sağ-çocuk düğümle karşılaştırılır. Eğer aranan eleman kök düğümün değerinden küçükse, aranan eleman sol-çocuk düğümle karşılaştırılır. Bundan dolayı yapılacak karşılaştırma sayısı N düğümlü bir ağaçta (log N) şeklinde olacaktır.

Eğer ikili ağaç, dengeli bir ikili ağaçsa şu özelikleri taşımalıdır:

- Bir ağaçta her iki alt ağacın derinlik farkı, yani │sağ alt ağaç-sol alt ağaç│<= 1 olmalıdır.
- Her iki alt ağaçta dengeli olmalıdır.

Dengeli ağaçların tanımına bağlı olarak, N düğümlü bir ağacın derinliği şöyle ifade edilebilir [34]:

Derinlik (Depth) = h olmak üzere, $2^h <= N < 2^{(h+1)}$ olur.

2.1.6. Kırmızı-siyah ağaç (red-black tree) veri yapısı

Kırmızı-siyah ağaç veri yapıları (Şekil 2.4.a) kendi kendini dengeleyen ikili bir ağaç türüdür. Bu ağaçlarda denge, bazı özelikleri de gözeterek ağaçtaki her bir düğümün kırmızı ve siyah renge boyanması ile sağlanır. Ağaç değiştirildiğinde (düğüm ekleme, silme gibi), yeni ağaç kırmızı-siyah ağaç özelliklerine göre yeniden düzenlenir ve düğümler yeniden renklendirilir. Bu işlemlerden sonra ağacın performansı artar. Aksi durumda dengesiz bir ağaç yapısı (Şekil 2.4.b) ortaya çıkar ki, bu durumda en kötü durum (worst case) oluşur (Çizelge 5.2, Çizelge 5.3). Buna bağlı olarak N elemanlı bir ağaçta zaman karmaşıklığı O(log N) düzeyinden, O(N) düzeyine çıkar [35].

Dengeli ağaçlar tamamen mükemmel değildir, fakat N elemanlı bir ağaçta O(log N) zaman karmaşıklığı ile aramayı garanti etmesi bakımından önemlidir [36]. Ayrıca kırmızı-siyah ağaçlarda düğüm ekleme, silme gibi işlemler de ağacın yeniden düzenlenmesi ve yeniden renklendirilmesi sayesinde O(log N) zaman karmaşıklığında gerçekleştirilir [37].

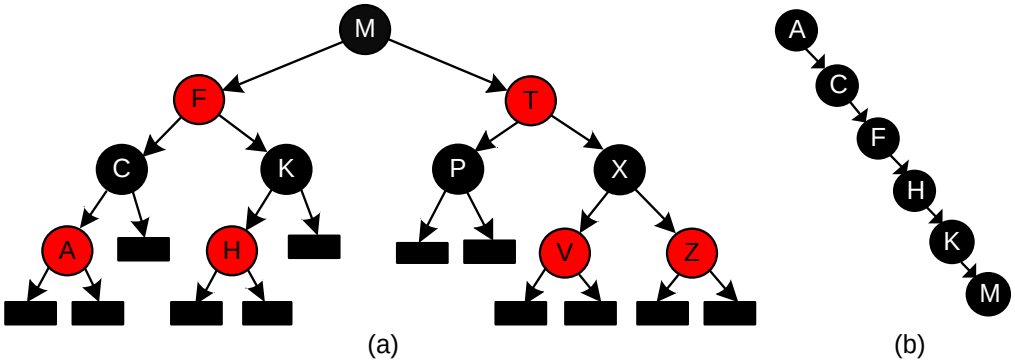
Kırmızı-siyah ağaçlar, ikili arama ağaçlarının sahip oldukları özelliklerin yanında, aşağıdaki ek özelliklere de sahiptirler [36, 38]:

- Ağaçtaki her düğüm ya kırmızıdır ya da siyah. Düğümler, bu renklere bağlı olarak kırmızı düğüm ve siyah düğüm olarak isimlendirilir.
- Ağaçta tek bir kök düğüm vardır ve siyahtır.
- Ağaçtaki bütün yapraklar siyahtır.
- Bir kırmızı düğümün her iki çocuğu da siyahtır.
- Bir düğümden atalarına doğru giden tüm basit yollar aynı sayıda siyah düğüm içerir.

Kırmızı-siyah ağaçlar bazı faydalı özelliklere sahiptirler. Bu açıdan birçok yerde tercih edilirler. Bu özellikler şunlardır [39]:

- İlk olarak, kırmızı-siyah ağaçta arama işlemi O(log N) sürede gerçekleşir.
- İkinci olarak, ağaç kendi kendini dengelediğinden yaklaşık olarak sol ağacın uzunluğu, sağ ağacın uzunluğuna eşittir.

Bu özelliklere göre {A,C,F,H,K,M,P,T,V,X,Z} elemanlarından oluşan kırmızı-siyah ağaç yapısı Şekil 2.4.a'ki gibidir



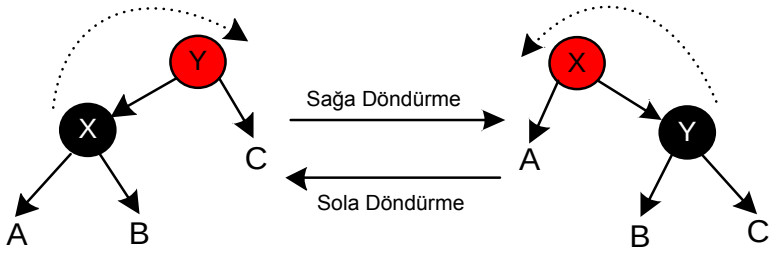
Şekil 2.4. (a) Kırmızı-Siyah Ağaç (b) Dengesiz Ağaç

Kırmızı-siyah ağaçlarda döndürme işlemi

Ağaçlarda döndürme işlemi alt ağaçların boylarını eşitlemek amacıyla yapılır. Bir ağaç döndürüldüğünde ağaçtaki düğümlerden bazıları yukarı bazıları aşağı hareket eder. Ağacın şeklini değiştirmek için ağaç döndürülür, bu döndürme sonucu derinliği az olan alt ağacın boyu artar, derinliği fazla olan alt ağacın boyu azalır böylece ağaç dengelenerek performansı artar.

Şekil 2.5'te görülen sağa döndürme işlemi sonucunda oluşan yeni ağacın kökü X olur. Bu işlem ağacın saat yönünde döndürülmesi sonucu oluşur. Bu işlemin tersi ise sola döndürme işlemidir ki saat yönünün tersine hareket etmektir. Bu işlem sonunda ise ağacın kökü Y düğümü olur. Bu işlemleri daha iyi kavrayabilmek için ağaç üzerinde yer değiştiren düğümleri takip etmek gerekmektedir. Ayrıca boyu kısalan ve uzayan alt ağaçları gözlemlemek gerekmektedir. Sağa döndürme işleminde X kök düğüm olurken, Y düğümü onun çocuğu olmaktadır. Bir düğümün en fazla iki çocuğu olacağından X düğümünün çocuğu olan B düğümü, Y düğümünün sol çocuğu olacaktır. Sola döndürme işleminde ise Y kök düğüm olurken, X düğümü onun çocuğu olmaktadır. Bir düğümün en fazla iki çocuğu olacağından Y düğümünün çocuğu olan B düğümü, X düğümünün sağ çocuğu

olacaktır. Şekil 2.5'teki her iki duruma da göz atıldığında yaprakların sırası değişmemektedir [28, 40].



Şekil 2.5. Ağacı Döndürme

Ağaçlarda döndürme işlemi sonrasında, ortada-kök (in-order) gezinti sonucu elde edilen düğüm sırası aynı kalmaktadır. Şekil 2.5'teki her iki ağaç için elde edilen ortada-kök (in-order) gezinti sonucu aynı olup şöyledir:

A X B Y C

Kırmızı-siyah ağaç (red-black tree) veri yapısında, her düğüm ekleme ve silme işleminde ağaçta dengenin sağlanması için yukarıdaki özelliklerin korunması gerekmektedir. Bu özellikleri korumak için düğüm ekleme ve silme işlemlerinde ağaçta sola döndürme(ler), sağa döndürme(ler) ve yeniden renklendirme renk değiştirme (kırmızı-siyah) işlemleri gerçekleştirilir. Bazen bir düğüm ekleme işleminde çok sayıda renk değişikliği ve döndürme işlemi gerçekleşebilir. Bu işlemler kırmızı-siyah ağaçların performansını olumsuz etkiler. Yani N elemanlı bir kırmızı-siyah ağaçta O(log N) olan düğüm arama, silme ve ekleme işlemlerinin zaman karmaşıklığı (time comlexity) üzerine bu maliyetlerin de eklenmesi gerekmektedir.

2.2. Algoritmalar

Algoritma belirli bir görevi yerine getiren sonlu sayıdaki işlemler dizisidir. Daha geniş tanımlamayla algoritma, verilen herhangi bir problemin çözümüne ulaşmak için uygulanması gerekli adımların hiç bir yoruma yer vermeksizin açık, düzenli ve sıralı bir şekilde sözlü ve yazılı ifade edilmesidir. Algoritmaları oluşturan adımlar anlaşılır, basit ve açık olarak sıralanmalıdır.

Her insan, gün içerisinde hayatı ile ilgili, yapacağı şeylere ait zihninden yüzlerce algoritma kurar ve problemlerinin bazılarını belirlediği çözüm adımlarını

uygulayarak çözer. Örneğin, evden çıkıp okula veya işe gitme problemini her öğrenci ya da çalışan günlük olarak çözer.

Bir algoritmanın taşıması gereken bazı temel özellikler vardır. Bunlar: sınırlılık (boundedness), belirlilik (definiteness), giriş (input), çıkış (output), basitlik (simplicity), genellik (generality), doğruluk (correctness), etkililik (efficiency) gibi özellikleridir [41].

2.2.1. Algoritma analizi

Algoritmalar tasarlandıktan sonra genellikle analizi yapılır. Bunun birkaç sebebi vardır. Bunlar:

- Tasarlanan algoritmanın performansını ölçmek
- Farklı algoritmalarla karşılaştırmak
- Başarımı iyi mi? İyileştirme olabilir mi? gibi sorulara cevap bulmaktır.

Algoritmalarda analiz, algoritmanın çalışma zamanını ve kullandığı bellek alanını hesaplamak için yapılır [32]. Bu tez çalışmasında daha çok algoritmaların zaman karmaşıklığı yönünden analizi üzerinde durulmuştur.

Karmaşıklığı ifade etmek için asimtotik ifadeler kullanılmaktadır. Bu amaçla O(n) (O notasyonu), Ω(n) (Omega notasyonu), Θ(n) (Teta notasyonu) gibi tanımlamalara başvurulur. Büyük O (Big-Oh) notasyonu asimptotik üst sınırı (En kötü durum analizi), Ω notasyonu asimtotik alt limiti (En iyi durum analizi), Θ notasyonu (Ortalama durum analizi) için kullanılır [24, 29, 32, 33, 41].

N elemanlı bir dizi üzerinde sıralı arama algoritması kullanılarak arama işleminin yapıldığını düşünürsek, bu üç durumu şöyle ifade edebiliriz:

- Doğrusal arama algoritmasında, karşılaşabileceğimiz en kötü durum aranan sayının dizinin en sonunda bulunması veya dizide hiç bulunmamasıdır. Bu durumda dizideki bütün elemanlara bakılması gerekecektir. Bundan dolayı en kötü durumda karmaşıklık N (O(N)) olacaktır.
- Ortalama durum analizi ise bu algoritmanın çok sefer çalışması sonucunda istatistiksel olarak ortalama kaç elemana bakılacağıdır. N elemanlı bu dizide bulunan sayıların hepsinin aranma oranlarının eşit olduğunu kabul edersek, ortalama durum N/2 olur.

- En iyi durum analizi ise, ilk bakılan sayının, aranan sayı olmasıdır. Bu durumda tek bir bakma işlemi yeterlidir. Bu durumda karmaşıklık 1 olacaktır.

Bir fonksiyon için O gösterimi üst sınırı (upper bound), Ω gösterimi ise alt sınırı (lower bound) ifade eder.

Büyük O notasyonunda, fonksiyonun davranışını en büyük dereceli terim belirler. Örneğin $F(n) = n^3 + 80n^2 + 5n + 30$ fonksiyonu, en büyük dereceli terimi n^3 olduğundan karmaşıklık olarak $O(n^3)$ şeklinde ifade edilir. Diğer terimler ve sabit (C=30) ihmal edilir. Aynı şekilde $G(n) = n^3 + 5n + 300000000$ ifadesi için de zaman karmaşıklığı $O(n^3)$ olur. Bu fonksiyonda küçük n değerleri için sabit değer (C) baskındır. Bu ifadeye dikkat edilirse diğer terimlerin, işlem süresini etkilemedikleri anlamına gelmez; bu yaklaşım N'nin çok büyük değerlerinde diğer terimlerin önem taşımadıkları anlamına gelir.

Çizelge 2.1. Bazı fonksiyonların büyük O gösterimi [27, 28, 42]

Notasyon	**Artış**	**Açıklama**
O(1)	Sabit	Algoritmadaki icra sayısı belliyse sabit bir değerle gösterilir. Örneğin, bir sayının tek mi çift mi olduğunun bulunması.
O(log N)	Logaritmik	N değerinin büyüyen değerlerine karşın algoritma çok daha az yavaşlıyorsa logaritmik bir durum söz konusudur. Örneğin, ikili arama ile sıralı bir dizide arama yapmak.
O(N)	Lineer	N değerinin büyümesine karşılık algoritmanın lineer bir şekilde yavaşlaması söz konusudur. Örnek, sırasız bir listeden bir değeri bulmak.
O(N logN)	Loglineer	Bir problemi alt problemlere bölüp bağımsız olarak çözen, daha sonra bu sonuçları birleştiren algoritmalarda görülür. Örnek, birleştirmeli sıralama (merge sort) algoritması.
$O(N^2)$	Karesel	İç içe döngüler ile verileri ikişerli şekilde inceleyen algoritmalarda görülür. Örnek, seçmeli sıralama (selection sort)
$O(2^N)$	Üstel	Girilen veriye göre iki kat yavaşlama görülen bu algoritmalar hiç pratik değildir. Örnek, seyyar satıcı problemi.

Birçok farklı kaynakta, notasyonlar kullanılarak arama, sıralama gibi algoritmaların zaman karmaşıklık analizleri, bellek gereksinimleri gibi analizleri yapılmıştır [24, 27, 28, 29, 32, 43, 45].

2.2.2. Arama algoritmaları

Sıralı olmayan verilerde kullanılan arama algoritmalarından birisi doğrusal arama (lineer searching) algoritmasıdır [44]. Bu algoritmada arama, veri kümesinin ilk elemanından başlanılarak son elemana kadar doğrusal bir şekilde devam eder [24]. Eğer aranan eleman dizinin sonlarına yakınsa arama çok yavaş gerçekleşir. Bu yüzden N elemanlı bir veri kümesinde arama zaman karmaşıklığı O(N) olur. Bu arama yönteminde arama yapılacak veri kümesinin sıralı ya da sırasız olması önemli değildir. Ancak sıralı veri kümeleri üzerinde verimli bir arama yöntemi değildir [46, 47, 48].

Diğer bir arama algoritması ikili arama (binary searching) algoritmasıdır. Bu algoritmanın bir veri kümesine uygulanabilmesi için veri kümesinin sıralı olması gerekmektedir. Eğer veri kümesi sıralı değilse önce sıralama algoritmalarından biri kullanılıp verilerin sıralanması gerekmektedir. Bu bir dezavantajdır.

İkili arama (Binary searching) algoritması şöyle çalışmaktadır [3, 25, 47, 49, 50, 51]:

- Dizi sıralı değilse önce dizi sıralanır.
- Daha sonra sıralı dizi eşit ya da birbirine yakın iki parçaya bölünür.
- Aranan eleman hangi parçada ise o alınır diğer parça atılır.
- Kalan parça tekrar aynı şekilde iki parçaya bölünür.
- Bu şekilde devam edilerek her defasında veri kümesi ikiye bölünerek aranan eleman bulununcaya kadar işlem devam eder.

N elemanlı sıralı bir veri kümesinde ikili arama algoritması için zaman karmaşıklığı en fazla O(lgN) olur.

Ayrıca, dengeli ağaçlarda da arama işlemi, ikili arama (binary searching) algoritmasında olduğu gibi en fazla O(lgN) sürede gerçekleşmektedir. Bağlı listelerde ise, düğüm arama işlemi doğrusaldır. N elemanlı bir bağlı listede bir düğümü arama zaman karmaşıklığı (time complexity) O(N) olarak gerçekleşir.

2.2.3. Sıralama algoritmaları

Sıralama algoritmaları bir veri kümesini küçükten büyüğe ya da büyükten küçüğe, yani artan veya azalan sırada oluşturmak için kullanılır. Diğer bir deyişle sıralama işlemi veriyi bilinen permütasyona dönüştürme işlemidir. Çok büyük veri kümelerinde verileri sıralamak çok önemlidir. Bu sayede veri kümesi üzerindeki bir elemana erişmek, üzerinde işlem yapmak kolaylaşır. Sıralı olmayan N elemanlı bir veri kümesinde aranan bir elemanı bulmak için O(N) zaman harcamak gerekmektedir. Çünkü bu veri kümesinde doğrusal bir arama söz konusudur. Fakat aynı veri kümesi sıralı hale getirilip ikili arama algoritması kullanılırsa bu karmaşıklık en fazla O(lg N) düzeyine düşer. Bu sonuçları göz önünde bulundurduğumuzda çok büyük veri kümelerinde sıralama işleminin ne kadar önemli olduğu görülmektedir [25,41,46].

Günümüze kadar birçok yeni sıralama algoritması geliştirilmiş ya da var olanlar üzerinde iyileştirmeler yapılmıştır. Bu çalışmalardan bazıları şunlardır: hızlı bir sıralama algoritması olan hızlı sıralama (quicksort) [52], karşılaştırma temelli bir sıralama algoritması yığın sıralama (heapsort) [53], birleştirmeli sıralama (merge-sort) algoritmasının daha etkin bir varyantı olan ve ikili arama ağaçlarına dayanan dengesiz birleştirmeli sıralama (unbalanced merge–sort) [54], kabarcık sıralama (bubble sort) ve seçmeli sıralama (selection sort) algoritmalarının iyileştirilmiş hali olan iki yeni algoritma [55], yığın sıralama (heapsort) algoritmasından daha hızlı yeni bir sıralama algoritması [56], araya yerleştirerek sıralama (insertion sort) algoritmasından daha hızlı olan, optimize edilmiş seçmeli sıralama (selection sort) algoritmasının karşılaştırmalı uygulaması [57], kabuk sıralama (shellsort) algoritmasının farklı bir versiyonu rastgele seçilmiş kabuk sıralama (randomized shellsort) [58], doğrusal derinlemesine sıralama (linear probing sort) ve kova sıralama (bucket sort) algoritmalarının analizi [59].

Sıralama algoritmalarını birçok yönden gruplandırmak mümkündür. Bu çalışmada daha çok zaman karmaşıklığı (time complexity) üzerinde durulup gruplandırma ona göre yapılacaktır. Yani $O(N^2)$ grubu sıralama algoritmaları (selection sort, bubble sort, shell sort gibi) ve O(NlgN) grubu sıralama algoritmaları [33] (merge sort, quick sort, heap sort gibi) şeklinde gruplandırma yapılacaktır.

2.3. İşletim Sistemlerinde Görev Zamanlayıcı Algoritmalar

Görev zamanlayıcılar (process scheduler) işletim sistemlerinin önemli bir bileşenidir. Görev zamanlayıcı, görevlerin sistem kaynaklarına erişimini yönetir ve ne kadar süreyle kullanacağını belirler. Bu zamanlama işlemi için farklı veri yapıları ve algoritmalar kullanır. Bir görev zamanlayıcı için sistem kaynaklarının verimli kullanımı çok önemlidir.

Linux işletim sisteminde yaygın kullanılan iki önemli görev zamanlayıcı algoritması vardır. Bunlar CFS (Completely Fair Scheduler) ve O(1) görev zamanlayıcı algoritmalarıdır. Bu iki farklı zamanlayıcı farklı tasarım ve performansa sahiptir, kullanıcılar kendi gereksinimlerine göre bu farklı görevlerden birini seçmelidir [39].

Her işletim sisteminin kendine göre farklı dönemlerde kullandığı farklı görev zamanlayıcı algoritmaları vardır. Her bir görev zamanlayıcı algoritmasını gerçekleştirebilmek için farklı veri yapıları (bağlı listeler, kırmızı-siyah ağaçlar gibi) kullanılmaktadır. Bu tez çalışmasında, FPS (Fair Priority Scheduler) algoritmasını gerçekleştirebilmek için önerilen atlamalı halka (skip ring) veri yapısı kullanılmıştır.

2.3.1. Linux işletim sisteminde görev zamanlayıcı

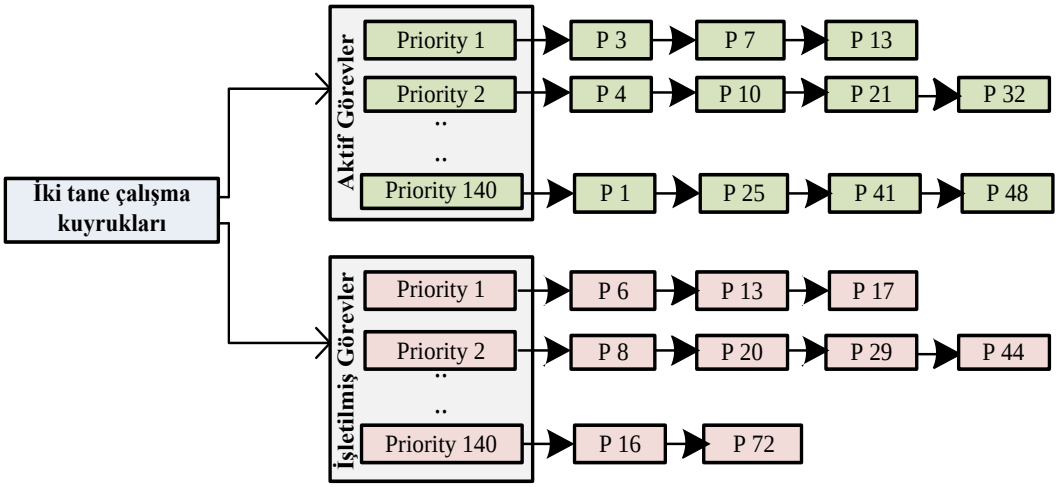
Görev zamanlayıcı bir işletim sisteminin en önemli parçasıdır. Linux® gelişmeye ve bu alandaki yeniliklerine devam etmektedir. Her işletim sisteminde olduğu gibi Linux işletim sisteminin de ilk sürümlerinde daha basit görev zamanlayıcılar kullanılmıştır. Günümüzdeki gibi büyük mimariler, çoklu işlemciler, çoklu çekirdekler ve bir çekirdekte birden çok iş parçacığının yürütülmesi gibi yenilikler yoktu. Linux 1.2 sürümü round-robin görev zamanlayıcı algoritmasını kullanmıştır. Görevleri tutmak ve yönetmek için dairesel kuyruk yapıları kullanılmıştır. Linux 2.2 versiyonu ile zamanlayıcı sınıfları (scheduling classes), gerçek-zamanlı görevler (real-time tasks) için zamanlayıcı politikalarına izin, kesilemeyen görevler (non-preemptible tasks) ve gerçek-zamanlı olmayan görevler (non-real-time tasks) ortaya çıkmıştır [39].

Linux 2.6 sürümüyle beraber O(1) (Şekil 2.6) olarak adlandırılan bir görev zamanlayıcı daha önceki görevlerin birçok problemini çözmek için tasarlanmıştır.

O(1) görev zamanlayıcı, öncelik temelli zamanlama politikasını kullanır. Bu görev zamanlayıcı görev öncelik kuyruğundan en uygun görevi çalışmak için belirler. O(1) zamanlayıcı algoritması çoklu kuyrukları kullanır (Şekil 2.6) . O(1) zamanlayıcı algoritmasının temel yapı taşı çalışma kuyruklarıdır. Bu kuyruklar iki gruba ayrılır;

- Aktif çalışma kuyrukları (active runqueue) : Çalışmayı bekleyen kuyruklar
- Çalışma süresi bitmiş kuyruklar (expired runqueue) : Kendine verilmiş olan çalışma süresini bitiren, fakat sonlanmayan yani tamamen bitmeyen görevlerin tutulduğu kuyruklar

Çekirdek bu iki çalışma kuyruklarına işaretçilerle (pointer) ulaşır. Bu iki çalışma kuyrukları basit bir işaretçi (pointer) ile karşılıklı yer değiştirilir. Yani tüm aktif kuyruklar bittikten sonra pasif olan kuyruk aktif yapılır.



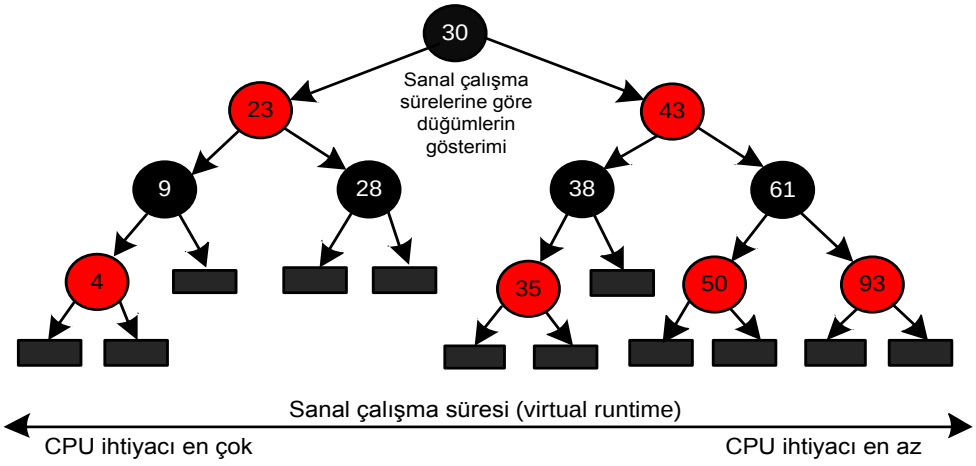
Şekil 2.6. O(1) Görev zamanlayıcı

Aktif kuyruklar içinde görevlerin önceliklerine göre oluşturulmuş 140 tane görev öncelik seviyesi vardır. Aynı önceliğe sahip görevler aynı öncelik kuyruğunda tutulurlar. Örneğin, Şekil 2.6'da Priority 2 öncelik kuyruğunda aynı önceliğe sahip P4, P10, P21 ve P32 görevleri vardır. Her bir kuyruk ise FIFO (fist-in-first-out) algoritmasına göre işletilir, yani ilk gelen ilk işletilecek demektir [34, 60].

Diğer bir görev zamanlayıcı (process scheduler) algoritması ise CFS (Completely Fair Scheduler)'dir. CFS algoritması ise görevlere adil bir çalışma süresi vaat eder. CFS görevlere işlem sürelerine göre adil bir pay verir. CFS

algoritmasında bu belirlenen çalışma süresi, sanal çalışma süresi (virtual runtime) olarak adlandırılır. Yani her görev için O(1) gibi sabit bir süre değil de adiliyet esasına dayalı her görev için farklı bir süre belirlenir.

O(1) zamanlayıcıda her bir göreve verilen zaman dilimi CFS ile aynıdır. CFS görev zamanlayıcı algoritmasında, bir göreve verilen çalışma süresi, sanal çalışma süresi olarak isimlendirilir. O(1) zamanlayıcı algoritmasında, bir görev yüksek öncelikli ise en kısa zamanda işletilecek demektir. CFS'de ise, daha küçük sanal çalışma süresine sahip olan görev en kısa zamanda işletilecek demektir. Görevleri yönetmek için O(1) temel olarak aktif ve pasif (expired) olmak üzere iki öncelik kuyruğu kullanır, CFS ise görevleri yönetmek için kırmızı-siyah ağaç veri yapısını kullanır. Kırmızı-siyah ağaç kullanılmasının asıl sebebi kendi kendini dengeleyen (self balancing) bir yapıda olmasıdır. İkinci olarak ise N düğümden oluşan bir kırmızı-siyah ağaçta bir düğüm ekleme veya silme işlemi O(log N) sürede gerçekleşir [61]. 5. Bölümde yeni geliştirdiğimiz atlamalı halka (skip ring) veri yapısı ile kırmızı-siyah ağaçlar kıyaslanmıştır.



Şekil 2.7. CFS için Kırmızı-Siyah Ağaç [34]

Şekil 2.7'deki sanal çalışma süresi değeri ağırlıklandırılmış zaman dilimi (weighted timeslice) olarak düşünülebilir. Sistemdeki görevler sanal çalışma süresi değerine göre kırmızı-siyah ağaca yerleştirilir. Bir görevin sanal çalışma süresi değeri ne kadar küçükse işlemciye ihtiyacı o kadar fazladır. Sanal çalışma süresi değeri küçük olan en solda ve en önce işletilecek görevi ifade eder. Yani görevler en soldan başlanılarak işletilirler biten görevler silinir yeni gelen görevler kırmızı-siyah ağaca eklenir.

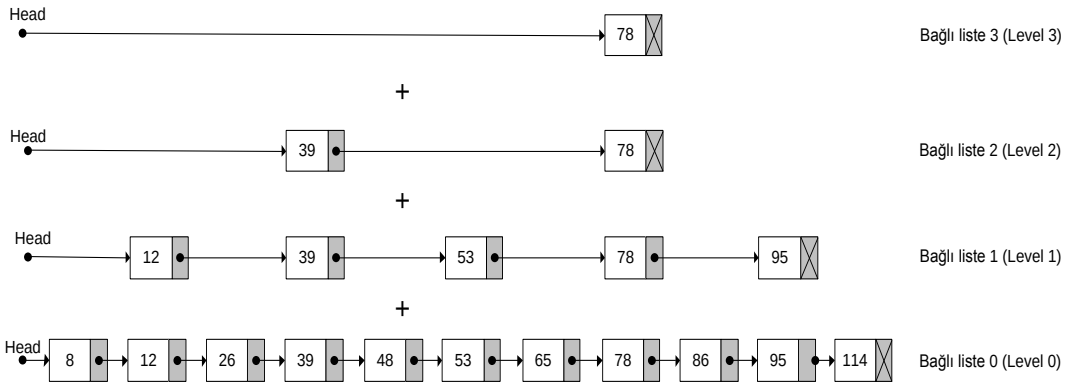
3. ATLAMALI LİSTE VE YAPILAN İYİLEŞTİRMELER

Atlamalı liste veri yapısında bağlı listeler kullanılır ve bağlı liste elemanları sıralı olarak değişik seviyelere (level) yerleştirilerek düğüm arama, ekleme, silme işlemlerinde kolaylık sağlanması amaçlanır. Bağlı listeler üst üste birbirinin fihristi olacak şekilde yerleştirilip birbirlerine bağlanırsa atlamalı liste veri yapısı ortaya çıkar (Şekil 3.1, Şekil 3.2). Atlamalı liste veri yapısında, bağlı liste veri yapısında olduğu gibi listenin her bir elemanı düğüm olarak kabul edilir ve her bir eleman bir anahtar ve bir değer ile ifade edilir.

Atlamalı liste veri yapısı bağlı listelerden oluştuğu için, bağlı liste veri yapısına bir alternatiftir. Katmanlı yapısı sayesinde bağlı listelere göre işlemler daha hızlı gerçekleşir. Bağlı listelerin kullanıldığı yerlerde atlamalı liste daha etkin bir şekilde kullanılabilir.

3.1. Atlamalı Listenin Oluşturulması

{8, 12, 26, 39, 48, 53, 65, 78, 86, 95, 114} düğümlerinden oluşan bir atlamalı listenin bağlı listelerden oluşturulması Şekil 3.1'de ve oluşturulmuş hali Şekil 3.2'de görülmektedir.



Şekil 3.1. Atlamalı listenin bağlı listelerden oluşturulması [1]

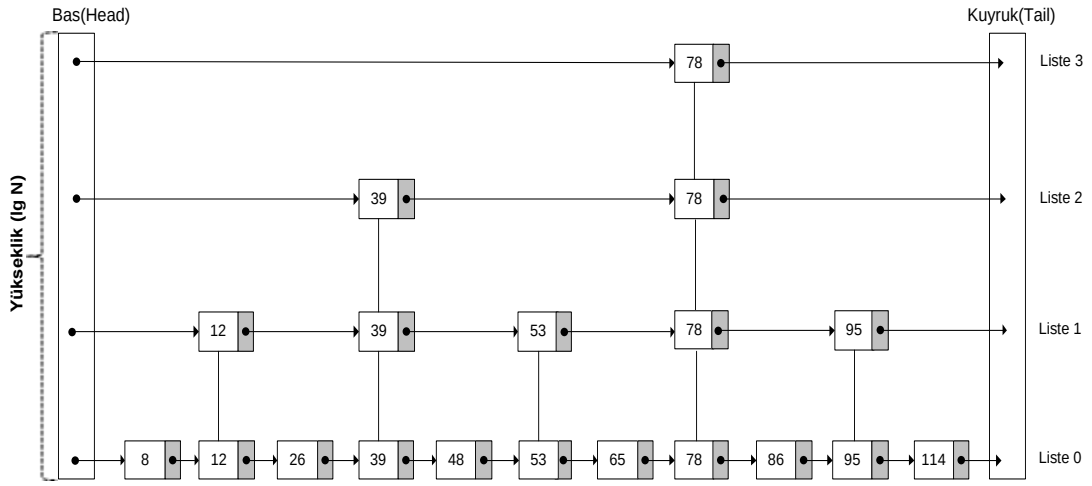
İlk olarak tüm düğümler level 0'da yer alır ve sol taraftan başlanarak her 2^i. düğüm (i=0,..,maxlevel) atlanarak üste doğru her seviyeyi (level) temsil eden işaretçiler oluşturulur.

Level 0 (Bağlı liste 0) ⊇ Level 1 (Bağlı liste 1) ⊇ .....⊇ Level (k) (Bağlı liste k). Bağlı liste 0 atlamalı liste veri yapısında en alt seviyedeki bağlı liste olup tüm düğümleri kapsar. Basit olarak bağlı liste düğümleri üzerinde bir fihrist oluşturularak bu liste üzerinde yapılacak işlemlerin daha kısa sürede yapılabilmesi sağlanır. Alttan üste doğru her liste bir altındaki listenin fihristi şeklinde sıralanır [51].

Bağlı listelerde yapılan düğüm arama, ekleme, silme gibi işlemler O(N) zaman karmaşıklığında yapılmakta iken, atlamalı liste veri yapısı sayesinde bu karmaşıklık O(lg N=$\log_2$N) düzeyine inmiştir.

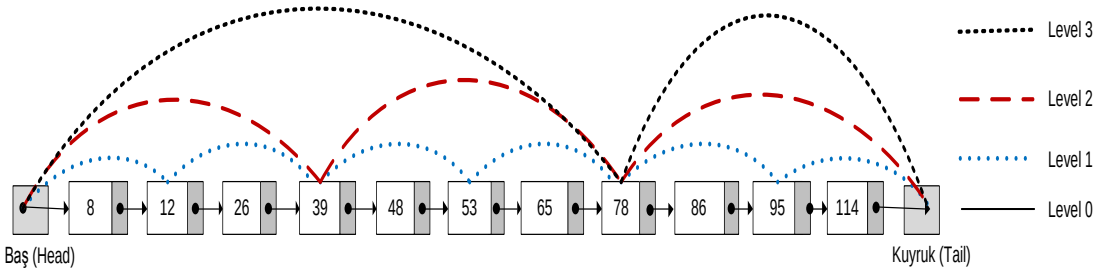
Atlamalı liste veri yapısı N tane sıralı düğümden oluşsun. Bağlı liste 0, bu N tane sıralı düğümlerin tamamından oluşur (Şekil 3.2-Level 0).

Bağlı liste 0 (Level 0)'da her 2 düğümden biri kendisinden 2 düğüm sonra gelen düğüme fazladan bir bağ içerirse, bağlı liste 1 (level 1) oluşur (Şekil 3.2-Level 1). Bağlı liste 1 (Level 1) en fazla (N/2+1) düğüm oluşur. Bağlı liste 0 (level 0)'da her 4 düğümden biri kendisinden 4 düğüm sonra gelen düğüme fazladan bir bağ içerirse bağlı liste 2 (Level 2) oluşur (Şekil 3.2-Level 2). Bağlı liste 2 (Level 2) seviyesi en fazla (N/4+2) düğümden oluşur. Bu şekilde devam edilerek bağlı liste 0'da her 2^i. düğümden biri kendinden sonra gelen 2^i. düğüme fazladan bir işaretçi ile bağlanırsa bağlı liste i oluşur. Bağlı liste i en fazla N/2^i+i tane düğümden oluşur. Böylece arama işleminde aranan bir düğüme ulaşmak için zaman karmaşıklığı O(lg N=$\log_2$ N) olur.



Şekil 3.2. Atlamalı liste veri yapısı [16]

Şekil 3.3'te ise {8, 12, 26, 39, 48, 53, 65, 78, 86, 95, 114} düğümlerinden oluşan gerçek bir atlamalı liste yapısı görülmektedir. Şekil 3.3'e dikkat edilirse gerçekte tek bir bağlı liste mevcuttur. Fakat farklı atlama noktaları oluşturularak yapı seviyeli (level) hale getirilmiştir.



Şekil 3.3. Atlamalı Liste (Gerçek Yapı)

Atlamalı liste veri yapısı ekleme, silme, arama işlemleri için kullanılabilmektedir.

Arama algoritmasında aranacak düğüm değeri üst seviyelerden (level) başlanarak aşağı seviyelere doğru aranır.

Ekleme işleminde önce eklenecek düğüm aranır bulunamamışsa eşleşen konuma rastgele (random) belirlenen seviyeden itibaren yeni düğüm eklenir, işaretçiler ve listeler güncellenir. Aynı işlem alt seviyeler için tekrarlanır.

Silme işleminde ise, en üst seviyeden başlanarak alt seviyelere doğru arama gerçekleştirilir silinecek düğüm bulununca silinir, işaretçiler ve listeler güncellenir. İşlem alt seviyeler için tekrarlanır.

Atlamalı liste veri yapısına düğüm ekleme, silme ve arama ile ilgili daha detaylı bilgi ve algoritmalar Pugh [4, 5]'un makalelerinde mevcuttur.

Bağlı ve sıralı listeler kullanıldığında atlamalı liste algoritmalarının (arama, ekleme, silme) zaman karmaşıklığı en fazla O(lg N) olmakta, bu da diğer algoritmalar ile karşılaştırıldığında (mesela bağlı listede arama (O(N))) önemli bir zaman farkı avantajı oluşturmaktadır.

Atlamalı liste veri yapısında arama, ekleme, silme işlemlerindeki zaman karmaşıklık (time complexity) tablosu Çizelge 3.1'de verilmiştir.

Çizelge 3.1. Atlamalı liste işlemlerinin zaman karmaşıklığı

İşlem	**Zaman Karmaşıklığı**
Arama	$O(\log_2 N = \lg N)$
Ekleme	$O(\log_2 N = \lg N)$
Silme	$O(\log_2 N = \lg N)$

Pugh'un atlamalı liste veri yapısı için geliştirmiş olduğu algoritmalarda bazı problemler vardır. Bu problemler atlamalı liste veri yapısının işlemlerinde (arama, ekleme, silme) performansı düşürmektedir.

Bu problemlerden bazıları:

- Yapıya bir düğüm eklerken eklenecek bu düğüm için seviye belirlemede kullanılan randomLevel algoritması (P=1/2 için) yüksek seviye üretmektedir.
- Atlamalı liste yapısında yüksek seviyelere çok sayıda düğüm eklenirse atlamalı liste olması gereken seviyeden çok yüksek olabilmektedir.
- Yapıdan çok miktarda yüksek seviyeli düğüm silinirse atlamalı liste veri yapısı düzensizleşmekte, performansı kötüleşmekte ve performans O(lg N)'den O(N)'e doğru kaymaktadır. Yani yapı adeta bağlı bir listeye dönüşmektedir.

Tezin ilerleyen bölümlerinde bu sorunlar ele alınıp, bunlara çözüm önerileri üzerinde durulacaktır.

3.2. Atlamalı liste için seviye (level) oluşturma

Atlamalı liste veri yapısında bir düğümü yapıya eklerken eklenecek düğümün hangi seviyede (level) olacağı önemlidir. Her düğüm aynı seviyede olmamalı ve düğümlerin seviyelere dağılımı dengeli olmalıdır. Bunun için seviye üreten bir algoritmaya ihtiyaç vardır. Atlamalı liste için iki şekilde seviye üretmek mümkündür. Bunlar:

1. Olasılıksal olarak seviye (level) üretme (**Algoritma 1**)
2. Düğüm sayısından faydalanarak seviye (level) üretme (**Algoritma 2**)

İlk olarak olasılıksal seviye üretme algoritması (**Algoritma 1**) ele alınmıştır. Atlamalı liste veri yapısı oluşturulurken, düğümler olasılıksal olarak Algoritma 1 ile

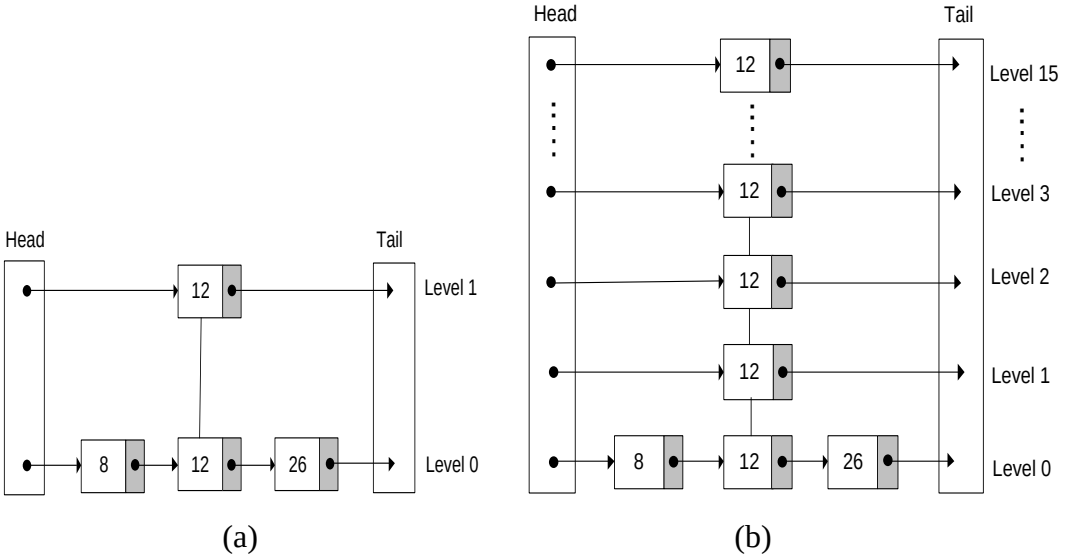
rastgele oluşturulan seviyelere yerleştirilir. Algoritma 1'deki MaxLevel değeri atlamalı listenin (skip list) alabileceği en yüksek seviyesidir. Pugh kendi algoritmalarında bu değeri 15 almıştır. Pugh'un **randomLevel()** algoritması (**Algoritma 1**) düğüm eklerken seviye (level) oluşturmak için olasılıksal olarak 0..MaxLevel arası rastgele bir değer üretir [1]. Bu değer (level) üretme şöyle gerçekleşir: İlk olarak lvl (seviye) başlangıç değeri 0 alınır. Daha sonra yazı tura atma işlemi gibi rastgele 0-1 arası ondalıklı bir değer üretilir. Bu üretilen değer, **P (1/2)** eşik değerinden küçük ve lvl<MaxLevel olduğu sürece lvl değeri 1 artırılır. İşlem bu şekilde devam eder. Ne zaman üretilen sayı **P** eşik değerinden büyük olursa ya da lvl değeri MaxLevel değerini ulaşmışsa, lvl üretilmiş olur. Eklenecek düğüm, üretilen bu lvl değeri esas alınarak yapıya eklenir. Örneğin, eklenecek bir düğüm için yazı tura atma işlemi gerçekleştirilsin, eğer üst üste 10 defa tura gelmişse eklenecek düğüm için seviye (level) 10 alınmalıdır.

Atlamalı liste veri yapısı inşa edilirken, düğümler rastgele belirlenen seviyelere yerleştirilir. Pugh'un geliştirdiği randomLevel algoritması (Algoritma 1, P=1/2 için) düğüm eklerken seviye (level) oluşturmak için 0..MaxLevel arası rastgele bir değer üretir. Buda atlamalı liste veri yapısında 2-3 düğüm varken bile seviye (level) olarak yüksek değerler oluşturabilmektedir (Şekil 3.4.b). Bu istenmeyen bir durumdur. Hâlbuki 2 veya 3 düğüm varken ideal seviye level=$\log_2$(N=3)'den 2 olmalıdır. Fakat bu şekilde seviye hesaplamak için atlamalı listedeki toplam düğüm sayısının bilinmesi gerekir.

İkinci olarak ise, atlamalı listedeki toplam düğüm sayısından seviye üretme algoritmasını (**Algoritma 2**) ele alalım. Bu algoritma için atlamalı listedeki toplam düğüm sayısının bilinmesi gereklidir. Bunun için düğüm sayısını saklayacak bir değişken (nodecount) tanımlanıp atlamalı liste oluşturulurken başlangıç değeri 0 alınmalıdır. Bu değişken (nodecount) atlamalı liste yapısına her düğüm eklemede 1 arttırılıp her düğüm silme işleminde 1 azaltılarak güncel tutulmalıdır. Bu bilgiler doğrultusunda randomLevel algoritması **(Algoritma 2)** oluşturulmalıdır.

İdeal atlamalı listede, N (nodecount) sayıda düğüm varsa, bu yapı için olması gereken en yüksek seviye düğüm sayısının 2 tabanında logaritması alınarak bulunur. Atlamalı listenin o anki olması gereken seviyesi level = $\log_2$(N=nodecount) ile bulunur.

Şekil 3.4'te {8,12,26} düğümlerinden oluşan iki ayrı atlamalı liste yapısı görülmektedir. Şekil 3.4.a'da olması gereken ideal seviye level=$\log_2$(N=3)'den 2 olmaktadır. Şekil 3.4.b'de ise istenmeyen (0..15) level düzeyinde gerçekleşen atlamalı liste görülmektedir.



(a) (b)

Şekil 3.4. (a) İdeal atlamalı liste (b) Gerçekleşebilecek atlamalı liste (Pugh) [1]

Pugh'un ***randomLevel*** algoritması şöyledir [4]:

Algoritma 1: Olasılıksal Seviye (Level) Üretme

```
1: randomLevel()
2:  lvl ← 0
3:   -- random() that returns
4:  --a random value in [0...1)
5: while random()< P and lvl < MaxLevel do
6:    lvl ← lvl + 1
7: return lvl
```

Bu tez çalışmasında kullandığımız ***randomLevel*** algoritması (Algoritma 2) ise nodecount (düğüm sayısı) değerini esas alarak seviye(level) üreten bir algoritmadır [1].

Algoritma 2: Düğüm Sayısından Seviye (Level) Üretme

```
1:  randomLevel(nodecount)
2:    lvl, level;
3:  if ( nodecount=0 then     {Atlamalı liste boşsa level=0 al}
4:     level ← 0;
5:  else
6:    {
7:        lvl ← log2(nodecount);   {nodecount değerinden level hesaplama}
8:       if  ( lvl ≠ head→level ) then
9:          level ← lvl
10:      else
11:        level ← random() % (head→level+1);   { Mod alma işlemi }
12:   }
13:  return level;
```

Önerilen algoritmada düğüm sayısına (nodecount) bağlı seviye üretildiği için çok yüksek seviyeler üretilmez. Yapı düzenli oluşacağından arama, ekleme, silme işlemlerinde zaman kaybı olmaz. Yani level = $\log_2$(nodecount) ifadesinin sonucu olan seviye (level) değeri atlamalı listenin o anki olması gereken ideal seviyesidir.

İlk olarak geliştirilen uygulama ile 17 düğümden oluşan bir atlamalı liste yapısı oluşturuldu. Daha sonra randomLevel algoritmasıyla (Algoritma 2) rastgele seviyeler üretilip bu seviyelere rastgele 16 yeni düğüm eklendi. Sonuç olarak bu 16 yeni düğümün seviyelere dağılımı Çizelge 3.2'de görülmektedir. Çizelge 3.2'ye dikkat edilirse eklenen düğümlerin seviyelere dağılımı düzenlidir.

Çizelge 3.2. Rastgele eklenen 16 düğümün seviyelere dağılımı

Seviyeler (level)	0	1	2	3	4
Düğüm Sayısı	4	3	4	3	2

Aynı işlemler 72 düğümden oluşan bir atlamalı liste yapısı için tekrarlandı. Sonra randomLevel algoritmasıyla (Algoritma 2) rastgele seviyeler üreterek bu seviyelere rastgele 25 yeni düğüm eklendi. Sonuç olarak bu 25 yeni düğümün

seviyelere dağılımı Çizelge 3.3'de görülmektedir. Çizelge 3.3'e dikkat edilirse eklenen düğümlerin seviyelere dağılımı düzenlidir.

Çizelge 3.3. Rastgele eklenen 25 düğümün seviyelere dağılımı

Seviyeler (level)	0	1	2	3	4	5	6
Düğüm Sayısı	4	3	4	2	4	5	3

Çizelge 3.2 ve Çizelge 3.3 incelendiğinde düğüm sayısına bağlı seviye üreten randomLevel algoritmasıyla elde edilen seviyeler idealdir. Ayrıca random olarak üretilen seviyelerin dağılımı da düzenlidir. Atlamalı liste için olması gereken ideal seviye korunmuştur. Pugh'un kendi ifadesine göre geliştirdiği olasılıksal randomLevel algoritması (Algoritma 1, P=1/2 için) 15 düğüm varken bile 14 seviyeli bir yapı oluşturabilmektedir (Şekil 3.4.b). Yani Pugh bu eklenen 15 düğümden bazıları seviye (level) 14 düzeyinde olabilir demektedir [4]. Bu kötü bir durumdur. Oysa 15 düğümlü bir atlamalı liste yapısı 4 seviyeden (level) fazla olmamalıdır. Bu problem düğüm sayısından seviye (level) üretilerek (Algoritma 2) çözülmüştür.

3.3. Düğüm Arama

Atlamalı liste veri yapısında arama işlemi gerçekleştirilirken düğüm ekleme veya silme işlemi olmadığından seviye (level) artırma veya azaltma işlemi yoktur. Bu yüzden Pugh'un geliştirdiği algoritmada bir problem yoktur.

Aramaya en üst seviyedeki (level) listeden başlanılarak aranan eleman bulununcaya veya en alt seviyede dahil her seviyede arama yapılıncaya kadar arama işlemi devam eder.

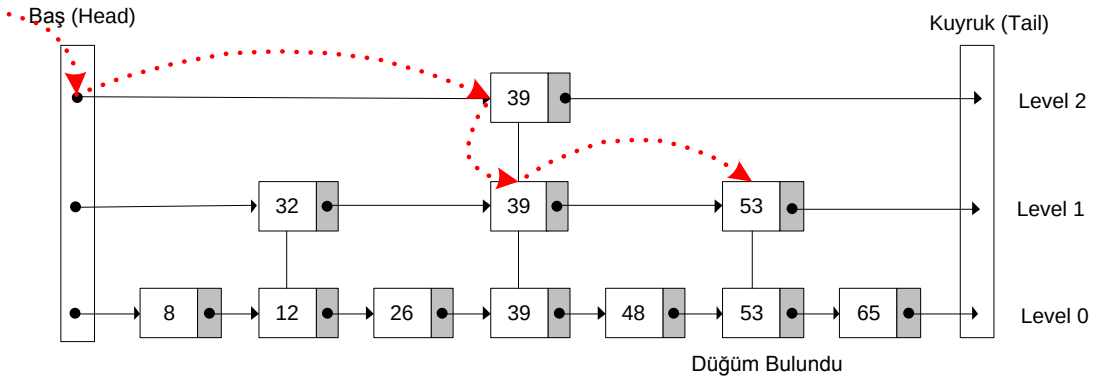
Atlamalı liste veri yapısında arama işlemi için aşağıdaki adımları takip etmek gerekmektedir.

Bu adımlar:

- Aranacak X değerini belirle
- Atlamalı liste (skip list) yapısı boş mu kontrol et
- Boş değilse X düğümünü en üst seviyeden başlayarak
  alt seviyelere doğru ara
- Eğer düğüm bulundu ise değeri döndür değilse false döndür.

Arama işlemi için Pugh [4]'un arama algoritması veya bu tez çalışmasında oluşturulan Algoritma 3 düğüm arama algoritması olarak kullanılabilir.

{8, 12, 26, 39, 48, 53, 65} düğümlerinden oluşan atlamalı liste veri yapısında "53" elemanının nasıl bulunacağı Şekil 3.5'te gösterilmektedir.



Şekil 3.5. Atlamalı liste veri yapısında düğüm arama

Algoritma 3: Atlamalı Liste Veri Yapısında Düğüm Arama

```
1:  searchNode (list, data)
2:    level ← list→level;
3:    temp ← list→head;
4:    if  temp→next[0] = null or level<0 then
5:          return  false;
6:    while ( level>=0 )
7:       if ( temp→next[level]→value=data ) then
8:            return   true
9:       if ( temp→next[level]→value<data ) then
10:           temp ← temp→next[level];
11:      if ( temp→next[level] →value > data)  or  (temp→next[level] = null) then
12:           level←level-1
13:   end while
14: return  false;
```

3.4. Düğüm Ekleme

Düğüm ekleme algoritmasında üst seviyeden en alt seviyeye doğru eklenecek konum bulunduktan sonra ekleme gerçekleşir. Burada en önemli problem eklenecek düğümün seviyesidir (level). Pugh'un seviye (level) üretme algoritması 0..MaxLevel (15) arası rastgele (random) bir sayı üretip onu seviye alarak eklenecek düğümü o seviyeden itibaren tüm alt seviyelere doğru eklemektedir. 0..15 (MaxLevel) arası rastgele bir sayı üretilmesi sonucu atlamalı liste veri yapısında 2-3 düğüm varken bile seviye (level) olarak yüksek değerler (Şekil 3.4.b) oluşturabilmektedir. Bu istenmeyen bir durumdur. Hâlbuki 2 veya 3 düğüm varken seviyesi level=$\log_2$(N=3)'den 2 olmalıdır (Şekil 3.4.a). Bunun için atlamalı listedeki toplam düğüm sayısının bilinmesi gerekir. Bu bilgiler doğrultusunda ekleme algoritması düğüm sayısını güncel tutacak şekilde oluşturulmalıdır.

Yeni bir düğüm ekleme işlemi için aşağıdaki adımların takip edilmesi gerekmektedir.

Bu adımlar:

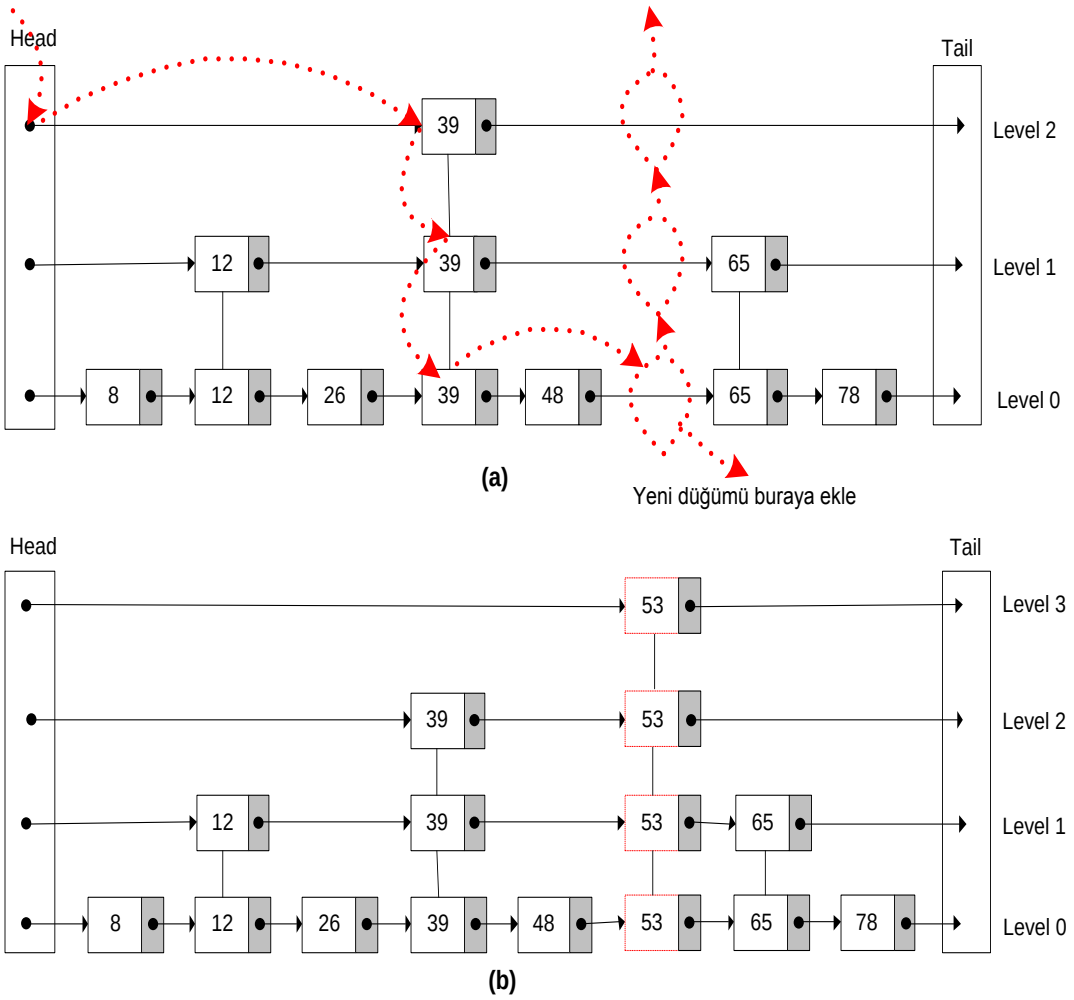
- Eklenecek X değerini belirle
- X değerini en üst seviyeden başlayarak alt seviyelere doğru ara
- Eğer X değeri bulundu ise false döndür çık
- Değilse rastgele (random) seviye (level) oluştur
- Yeni bir düğüm (P) oluştur. Bu düğümün value değerini X yap
- P'den önce gelen düğümün işaretçisi P'yi;
  P'nin işaretçisi P'den sonra gelen düğümü gösterecek şekilde yapıyı güncelle
- Düğümü gereken diğer seviyelere de ekle

Pugh'un insertNode algoritmasına ***nodecount*** değişkeni eklenerek ***insert*** algoritması yeniden düzenlenmiştir (Algoritma 4). Algoritmanın bu halinde atlamalı liste yapısına eleman eklenirken her düğüm eklemede ***nodecount*** değeri 1 artırılmaktadır. Bu sayede atlamalı listedeki toplam eleman sayısı bulunmaktadır. Nodecount parametresinin o anki değeri atlamalı listenin level 0 düzeyindeki toplam düğüm sayısını verir.

Algoritma 4 : Atlamalı Liste Veri Yapısına Düğüm Ekleme

```
1:  insertNode (list, searchkey, newvalue, nodecount)
2:    update[1..MaxLevel];
3:    temp ← list→head;
4:  for i ← list→level downto 1 do
5:     while temp→next[i]→key < searchkey do
6:        temp ← temp →next[i]
7:     end while
8:     update[i] ← temp
9:  end for
10:  temp ← temp →next[1];
11:  if temp → key = searchkey then
12:    temp → value ← newvalue
13:  else
14:     newlevel ← randomlevel(nodecount);
15:     if newlevel > list→level then
16:        update[i] ← list→head
17:        list→level ← newlevel
18:     end if;
19:    temp ← MakeNode(newlevel,searchkey,newvalue);
20:     for i ← 1 to newlevel do
21:       temp →next[i] ← update[i]→next[i];
22:       update[i]→next[i] ← temp
23:     end for
24:    nodecount ← nodecount+1;
25:  end if-else
26: return nodecount;
```

{8, 12, 26, 39, 48, 65, 78} düğümlerinden oluşan atlamalı liste veri yapısına 53 elemanının nasıl ekleneceği Şekil 3.6'da gösterilmektedir. Önce '53' elemanı aranıyor yoksa hangi konuma eklenecekse, eşleşen yer bulunuyor. Daha sonra ekleme işlemi gerçekleştirilip işaretçiler güncelleniyor.



Şekil 3.6. (a) Atlamalı liste düğüm ekleme işlemi (b) Düğüm Eklenmiş Hali

3.5. Düğüm Silme

Atlamalı liste veri yapısından bir düğümü silmek için önce arama algoritması kullanılarak silinecek düğüm bulunmalıdır. Düğüm bulunduktan sonra bulunduğu tüm seviyelerden silinir. Tüm işaretçiler güncellenir. Silme algoritmasında arama algoritmasında olduğu gibi ilk dikkat edilmesi gereken husus atlamalı liste veri yapısında hiç eleman olmayabilir. Bu durum kontrol edilmelidir.

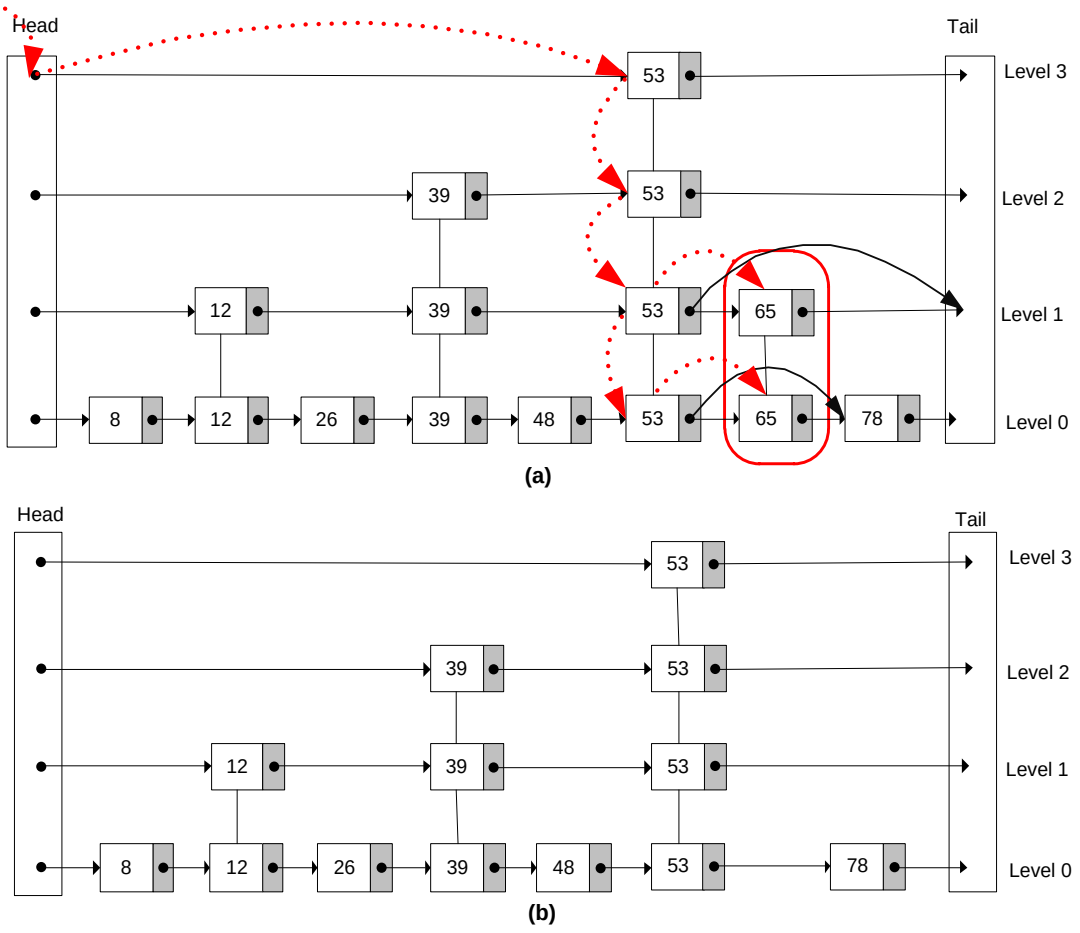
DeleteNode algoritmasını atlamalı liste yapısındaki düğüm değerini güncel tutacak şekilde düzenlemek gerekmektedir. Yani atlamalı liste yapısından düğüm silerken her düğüm silme işleminden sonra **nodecount** değeri 1 azaltılmalıdır. Bu sayede atlamalı listedeki toplam düğüm sayısı güncel tutulmaktadır. DeleteNode algoritması (Algoritma 5) bunlar dikkate alınarak oluşturulmuştur.

Silme işlemi için ise aşağıdaki adımların takip edilmesi gerekmektedir.

Bu adımlar:

- Silinecek P düğümünü belirle
- Bu düğümü en üst seviyeden başlayarak alt seviyelere doğru ara
- Eğer P düğümü bulundu ise P'den önce gelen düğümün işaretçisi P'den sonra gelen düğümü gösterecek şekilde güncelle
- P düğümünü var olan her seviyeden sil
- Düğümü bellekten sil

{8, 12, 26, 39, 48, 53, 65, 78} düğümlerinden oluşan atlamalı liste yapısından 65 düğümünün nasıl silineceği Şekil 3.7'de gösterilmiştir.



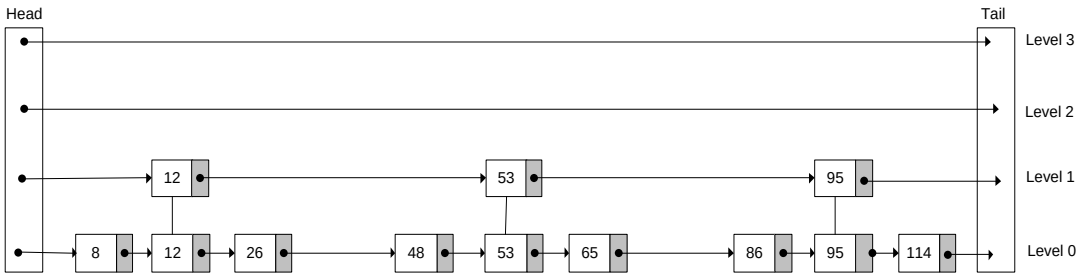
Şekil 3.7. (a) Atlamalı liste düğüm silme işlemi (b) Düğüm silinmiş hali

Algoritma 5: Atlamalı Liste Veri Yapısından Düğüm Silme

```
1:  deleteNode (list, searchkey, nodecount)
2:       update[1..MaxLevel];
3:       temp ← list→head;
4:    for i ← list→level downto 1 do
5:      while temp→next[i]→key < searchkey do
6:         temp ← temp→next[i]
7:      end while;
8:      update[i] ← temp
9:    end for;
10: temp ← temp→next[1];
11: if temp→key = searchkey then
12:      for i ← 1 to list→level do
13:        if update[i]→next[i] ≠ temp then
14:            break
15:         end if;
16:         update[i]→next[i] ← temp→next[i]
17:      end for;
18:      free(temp);
19:      while list→level > 1 and  list→head→next[list→level] = NIL  do
20:         list→level ← list→level – 1
21:      end while
22:    nodecount ← nodecount-1;
23:  end if;
24: return nodecount;
```

3.6. Atlamalı Listeyi Yeniden Düzenleme (Reorganization) Gereksinimi

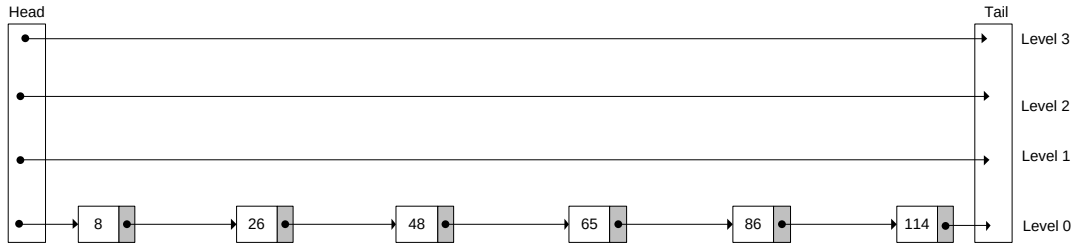
Atlamalı liste veri yapısında başka bir problem de sürekli yüksek seviyeli düğümler silinirse atlamalı liste veri yapısı düzensizleşir. Şekil 3.2'deki atlamalı liste yapısından yüksek seviyeli '39' ve '78' düğümleri silinirse atlamalı liste yapısı Şekil 3.8'deki hale dönüşür. Atlamalı listede seviye (level) sorunu ortaya çıkar.



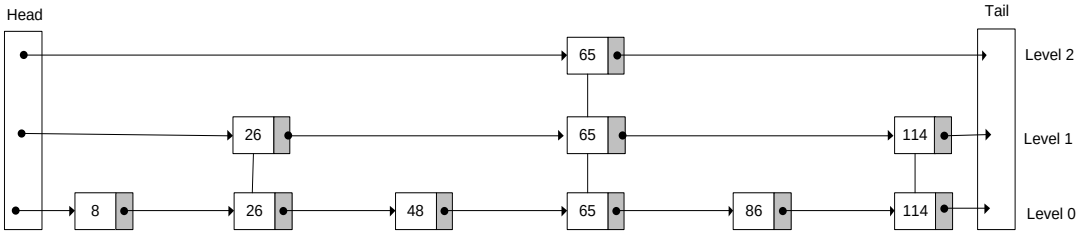
Şekil 3.8. Düzensiz bir atlamalı liste

Habuki bu halde ideal seviye ***level = log₂(9)***'dan 4 (level 0, level 1, level 2, level 3) olması gerekir. Bu problemi çözmek için **nodecount** değerinden faydalanılır. Atlamalı liste yapısı düzensizleştikçe mevcut düğümlerle yapı yeniden inşa edilmelidir (**Algoritma 6, Algoritma 7**).

Şekil 3.8'deki yapıdan düğüm silmeye devam edip '12', '53', '95' düğümleri de silinirse yapı tamamen çöker bağlı liste haline dönüşür (Şekil 3.9). Hâlbuki Şekil 3.8'deki yapıdan '12', '53', '95' düğümleri silindikten sonra geriye 6 düğüm kalır ve ideal seviye ***level = log₂(6)'***dan **3 (**level 0, level 1, level 2) olmalıdır (Şekil 3.10).



Şekil 3.9. Atlamalı liste (çökmüş hali)= Bağlı liste



Şekil 3.10. Şekil 3.9'daki atlamalı listenin ideal hali (Reorganized algoritması ile yeniden inşa edilmiş hali)

Şekil 3.8 ve Şekil 3.9 incelendiğinde atlamalı liste veri yapısından çok fazla düğüm silindiğinde bu veri yapısının dengesi bozulur. Bunun sonucu olarak arama, ekleme, silme işlemleri verimsiz hale gelir. Hatta Şekil 3.9 incelendiğinde çok fazla

yüksek seviyeli düğüm silinmesi sonucu atlamalı liste yapısı tamamen çökmüştür. Yerine adeta bağlı liste gelmiştir. İşte bu tür durumlarda nodecount değerine bağlı olarak yapı yeniden inşa (rebuild) edilmelidir.

Yani atlamalı liste yapısının ***level*** değeri; ***lvl=log$_2$(nodecount)*** işlemi ile elde edilen ***lvl*** değerinden farklı ise yapıyı yeniden inşa etme (rebuild) işlemi gerçekleşir değilse gerçekleşmez. Bu işlem otomatik hale getirilebilir. Yeniden inşa işleminden sonra Şekil 3.9'daki yapı Şekil 3.10'dakine dönüşür. Yani yeniden inşa işleminden sonra tekrar ideal atlamalı liste oluşturulur.

Algoritma 6: Atlamalı listeyi Yeniden İnşa Etme -1

```
1:  Reorganized (nodecount)
2:      node p ← head, tmp[nodecount+1];
3:      head→level ←log2(nodecount);
4:    for i ← 1 to nodecount+1 do
5:      tmp[i] ← MakeNode();
6:      tmp[i] →value ← p→next[0] →value;
7:      p ← p→next[0];
8:    end for
9:    for i ← 0 to head→level  do
10:     j ← 2;
11:     k ← pow(2, i);
13:     head→next[i] ← tmp[k];
14:     p ← head→next[i];
15:     while ( j*k <= nodecount ) do
16:       p→next[i] ← tmp[j*k];
17:       p ← p→next[i];
18:       j ← j+1;
19:     end while
20:     p→next[i] ← tmp[nodecount+1];
21:   end for
22: return nodecount;
```

Atlamalı listede yapılan bu iyileştirmelere ait uygulamalar yapılarak sonuçlar karşılaştırılmıştır. Uygulama standart C programlama dili komutları kullanılarak Dev-C++ ortamında geliştirilmiştir. Sonuçlar, intel core i5-3230M 2.60 Ghz işlemci ve 8 GB belleğe sahip bir bilgisayar üzerinde koşturularak elde edilmiştir.

İlk uygulamada düğüm sayısından seviye oluşturan algoritma kullanılıp atlamalı liste oluşturulmuş, seviye ve bu seviyelere düğümlerin dağılımları Çizelge 3.2 ve Çizelge 3.3'te sunulmuştur. Çizelge 3.2 ve Çizelge 3.3'e dikkat edilirse, oluşan seviyeler ve bu seviyelere düğümlerin dağılımları düzenlidir.

Algoritma 7: Atlamalı Listeyi Yeniden İnşa Etme - 2

// Atlamalı listeye N tane düğümün ekleneceğini farzedelim.

1. N elemanlı bir bağlantılı liste olsun ve N elemandan oluşan bu bağlantılı liste, atlamalı liste (skip list) veri yapısının ilk seviyesi (Level 0) kabul edilsin.
2. for k←1,2,…,N
    a. Mask←1
    b. for j←1,2,..,$\lceil \lg(N) \rceil$
        i. if Mask⊕k then
            1. Upward k^{th} node to $Level_j$.
        ii. ShiftLeftOneBit(Mask);

Diğer bir uygulamada ise Pugh'un randomLevel algoritması kullanılarak farklı **P** ([0...1)) eşik değerleri alınıp atlamalı listeler oluşturulmuş ve sonuçlar incelenmiştir. Elde edilen bu sonuçlar Çizelge 3.4, Çizelge 3.5, Çizelge 3.6, Çizelge 3.7, Çizelge 3.8'de sunulmuştur. Geliştirilen uygulama ile farklı sayıda düğümlerden oluşan atlamalı liste yapıları oluşturarak, farklı P eşik değerleri için (0.1, 0.25, 0.5, 0.75, 0.9) sonuçların nasıl değiştiği incelendi. Elde edilen veriler Çizelge 3.4, Çizelge 3.5, Çizelge 3.6, Çizelge 3.7, Çizelge 3.8'de görülmektedir. Çizelgelerdeki atlamalı liste oluşturulma süreleri milisaniye (ms) cinsindendir [16].

Çizelge 3.4. P eşik değeri 0.1 için

Düğüm sayısı	5000	10000	20000	50000	100000	200000	500000
Oluşan Seviye ~	3	4	5	6	5	5	5
Atlamalı liste Oluşma Süresi~	3	6.2	15.6	59	78	218	780

Çizelge 3.5. P eşik değeri 0.25 için

Düğüm sayısı	5000	10000	20000	50000	100000	200000	500000
Oluşan Seviye ~	5	6	7	7	8	10	11
Atlamalı Liste Oluşma Süresi ~	3	6	12	31	62	171	593

Çizelge 3.6. P eşik değeri 0.5 için

Düğüm sayısı	5000	10000	20000	50000	100000	200000	500000
Oluşan Seviye ~	13	14	15	16	17	16	19
Atlamalı Liste Oluşma Süresi ~	3	6.2	16	46	78	179	624

Çizelge 3.7. P eşik değeri 0.75 için

Düğüm sayısı	5000	10000	20000	50000	100000	200000	500000
Oluşan Seviye ~	33	36	38	41	44	48	45
Atlamalı Liste Oluşma Süresi ~	4	7	17	53	94	202	640

Çizelge 3.8. P eşik değeri 0.9 için

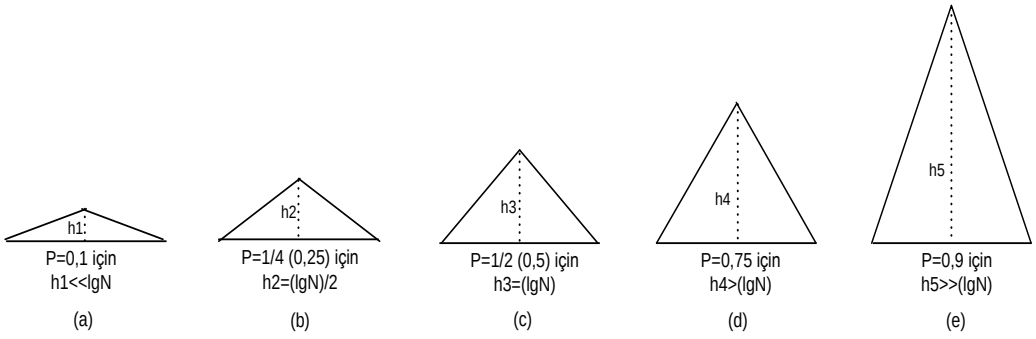
Düğüm sayısı	5000	10000	20000	50000	100000	200000	500000
Oluşan Seviye ~	82	87	93	112	116	117	139
Atlamalı Liste Oluşma Süresi ~	6	12	26	61	124	296	936

Çizelge 3.4, Çizelge 3.5, Çizelge 3.6, Çizelge 3.7 ve Çizelge 3.8'deki değerler incelendiğinde atlamalı listenin seviyesi (yüksekliği) fazlaysa, (Çizelge 3.6, Çizelge 3.7 ve Çizelge 3.8) atlamalı listenin oluşturulma süresi de artmaktadır. Ayrıca yüksekliği fazla olan atlamalı listenin düğüm arama, ekleme, silme işlemlerinin performansı da kötü olacaktır. Çizelge 3.4 incelendiğinde çok küçük **P** (0.1) eşik değerleri de atlamalı liste veri yapısının oluşturulma süresini artırmaktadır. **P**=0.1 iken seviye olması gerekenden az olduğu (Şekil 3.11.a) için performans olumsuz etkilenmektedir.

Yukarıdaki çizelgelere (Çizelge 3.4, Çizelge 4.5, Çizelge 3.6, Çizelge 3.7, Çizelge 3.8) dikkat edilirse, Pugh'un randomLevel algoritması için en ideal P eşik değeri ~1/4 (0.25)'tür. 100000 (yüzbin) düğümlü bir atlamalı listede Çizelge 3.4 **(P=0.1)** ve Çizelge 3.8 **(P=0.9)** kıyaslandığında, Çizelge 3.4'de oluşan seviye=5 (yükseklik) ve atlamalı listenin oluşma süresi=78 ms iken Çizelge 3.8'de oluşan

seviye=116 (yükseklik) ve atlamalı listenin oluşma süresi=124 ms olmaktadır. Şekil 3.11'de atlamalı liste yapısının yüksekliğinin farklı **P** değerleri için nasıl değiştiği gösterilmektedir. Çizelge 3.4 ve Çizelge 3.8'deki sonuçlara dikkat edilirse **P=0.1** (Şekil 3.11.a) için atlamalı liste yaklaşık olarak bir bağlı listeye (satıra), **P=0.9** (Şekil 3.11.e) değeri için adeta dikey bir sütuna dönüşmektedir. Yani arama, ekleme, silme gibi işlemler logaritmik yerine doğrusala dönüşmektedir. Çizelge 3.5 incelendiğinde 100000 (yüzbin) düğümlü bir atlamalı listede **P** eşik değeri ~1/4 alınırsa, oluşan seviye=8 (yükseklik) ve atlamalı listenin oluşma süresi=62 ms olmaktadır. Yani P eşik değeri ~1/4 alındığında çok yüksek seviyeler üretilmez. Bu durum düğüm sayısı çok fazla olunca (500000) daha da belirgin bir hal almaktadır. Bundan dolayı atlamalı liste oluşturma süresi, arama, ekleme, silme gibi işlemlerin süresi diğerlerine göre daha iyidir. Bu durumda atlamalı liste yapısı düzenli oluşacağından arama, ekleme, silme işlemlerinde zaman kaybı olmaz. Yani elde edilen ***level*** değeri, atlamalı listenin o anki olması gereken ideal seviyesine yakın bir seviyede olur [16].

Atlamalı listenin analizi ve iyileştirilmesine yönelik yapılan bu çalışmalar sonucunda düğüm sayısına bağlı seviye oluşturma (Algoritma 2) ile olasılıksal seviye oluşturma algoritmasının (Algoritma 1) P=1/4 (Şekil 3.11.b) alındığında yakın sonuçlar ürettiği görülmüştür.



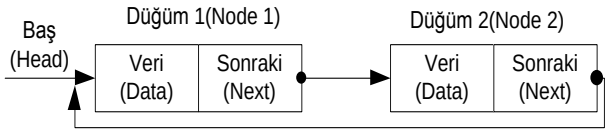
Şekil 3.11. Farklı P değerlerinin atlamalı liste veri yapısının yüksekliğine etkisi

4. YENİ VERİ YAPISI ATLAMALI HALKA-DAİRESEL ATLAMALI LİSTE (SKIP RING-CIRCULAR SKIP LIST)

Önerilen yeni veri yapısı, atlamalı liste ve dairesel bağlı listenin özellikleri göz önünde bulundurularak oluşturulmuştur. Ayrıca yeni veri yapısı oluşturulurken, atlamalı liste veri yapısındaki analizler ve iyileştirmeler göz önünde bulundurulmuştur.

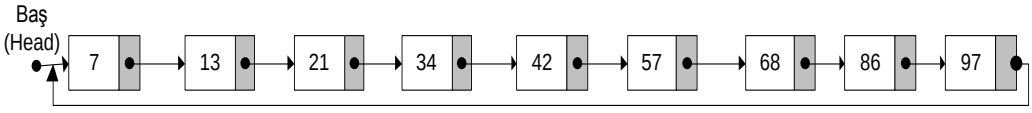
4.1. Dairesel Bağlı Liste

Dairesel bağlı listeler, bağlı listelerin bir türü olup listenin en son düğümü ilk düğümü gösteren bir işaretçiye sahiptir. Tek veya çift yönlü bağlı listelerde en son düğümün işaretçisi "NULL" olarak belirlenmiştir, oysaki dairesel bağlı listelerde en son düğümün işaretçisinin "next"'i dairesel bağlı listedeki ilk düğümü gösterir [27] (Şekil 4.1). Bu yüzden bu yapı dairesel bağlı liste olarak isimlendirilir. Yani bir dairesel bağlı listede, liste sonu liste başını gösterdiğinden tüm düğümler dairesel olarak birbirlerine bağlıdırlar (Şekil 4.2).



Şekil 4.1. Dairesel Bağlı Listelerde Düğüm Yapısı

Tek yönlü bağlı liste kullanılarak gerçekleştirilen tüm işlemler, dairesel bağlı listeler kullanılarak ta gerçekleştirilebilir. Ayrıca bir dairesel bağlı liste yığıt veya kuyruk olarak ta kullanılabilir.

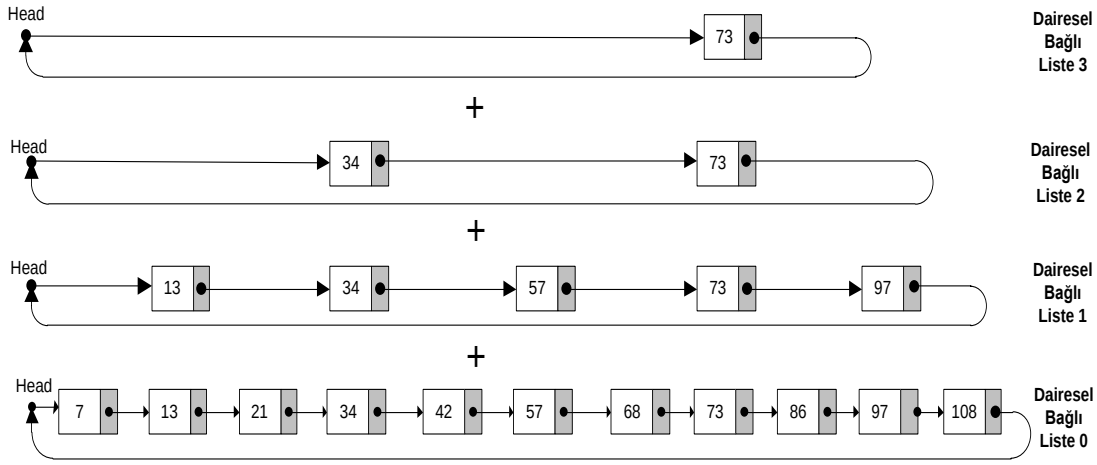


Şekil 4.2. Dairesel Bağlı Liste

Dairesel bağlı listeler tek yönlü ya da çift yönlü olabilir. Dairesel bağlı listeler tek bir işaretçi ile ilk ve son kayda hızlı bir erişim sağlar [2, 3, 27]. Dairesel işlem gerektiren yerlerde etkili bir şekilde kullanılabilirler. Özellikle işletim sistemlerin görev zamanlayıcı algoritmalarında kullanılmaktadır.

4.2. Önerilen Veri Yapısı Atlamalı Halka (Skip Ring)

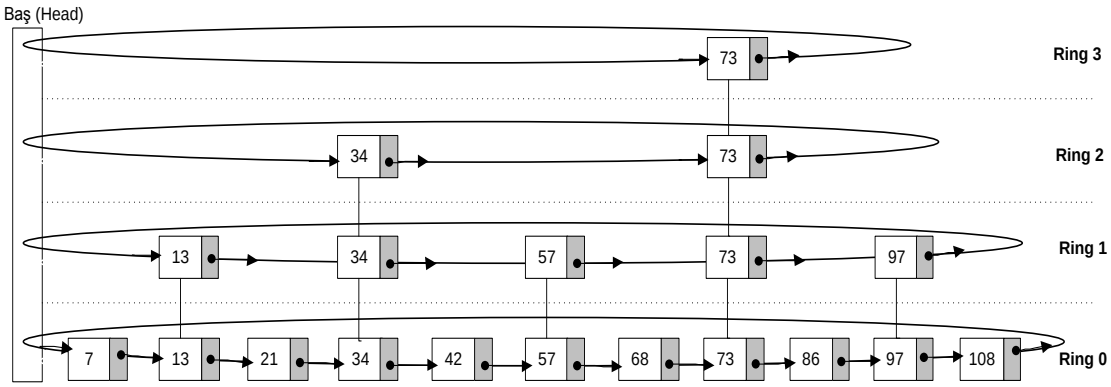
Yukarıda anlatılan dairesel bağlı liste (circular linked list) ve atlamalı liste (skip list) veri yapılarından hareketle yeni veri yapısı atlamalı halka (Skip ring - circular skip list) önerilmiştir. Dairesel bağlı listede (Şekil 4.2) sadece bir sıralı liste üzerinde düğüm arama, ekleme, silme gibi işlemler gerçekleştirilir. Bu tezdeki yaklaşım ise birbirine bağlı seviyeler halinde birbirinin fihristi olan dairesel bağlı listeler (Şekil 4.3- Şekil 4.4) üzerinde düğüm arama, ekleme, silme gibi işlemler gerçekleştirilmektedir.



Şekil 4.3. Atlamalı Halka (Skip ring) veri yapısının inşası

Koni şeklindeki bu yeni veri yapısında Dairesel bağlı liste *0 = Ring 0*, Dairesel bağlı liste *1 = Ring 1*, ... , Dairesel bağlı liste *(k)=Ring (k)* şeklinde ifade edilmektedir (Şekil 4.3). Atlamalı halka (Skip ring) veri yapısında her bir halka kendi üstündeki halkayı kapsar, Ring 0 $\supseteq$ Ring 1 $\supseteq$ .....$\supseteq$ Ring k . Ring 0, atlamalı halka veri yapısındaki en alt seviyedeki halka olup tüm elemanları kapsar. Alttan üste doğru her halka altındaki halkaların fihristi şeklinde sıralanır. Ayrıca atlamalı halka veri yapısı atlamalı liste veri yapısına da benzerdir. Atlamalı halka (Skip ring) veri yapısında atlamalı listeden (skip list) farklı olarak Baş (head) ve kuyruk (tail)'un ikisine birden gerek yoktur; sadece Baş (head) yeterli olmaktadır. Atlamalı liste veri yapısının kuyruk (tail) kısmı yok edilerek her seviyedeki dairesel bağlı listenin son elemanı ilk elemanı (head) gösterecek şekilde birbirine bağlanılırsa, yeni veri yapısı ortaya çıkar (Şekil 4.4).

Atlamalı halka veri yapısındaki halkalar (Ring 0, Ring 1,.., Ring k) oluşturulurken seviyeler (level) rastgele oluşturulur. Atlamalı halka veri yapısı N tane sıralı düğümden oluşsun. Ring 0, bu N tane sıralı düğümlerin tamamından oluşur (Şekil 4.4-Ring 0).

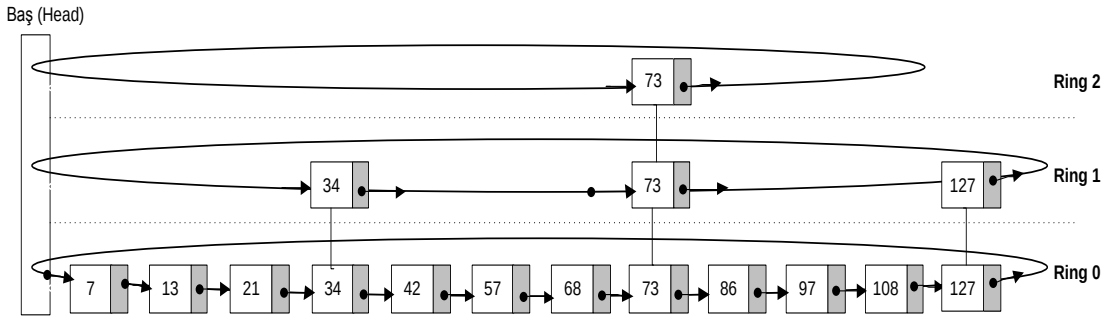


Şekil 4.4. Atlamalı Halka (Skip ring) (P=1/2 için) [62]

Ring 0 seviyesindeki arama işlemi sıralı N tane eleman olduğundan en fazla O(N) zaman karmaşıklığında olur. Ring 0'da 2. elemandan başlayarak her eleman kendinden 2 eleman sonra gelen elemana fazladan bir bağ içerirse, Ring 1 oluşur (Şekil 4.4-Ring 1). Ring 1 seviyesi en fazla (N/2+1) elemandan oluşmaktadır. Ring 0'da 4. elemandan başlayarak her eleman kendisinden 4 eleman sonra gelen elemana fazladan bir bağ içerirse, Ring 2 oluşur (Şekil 4.4-Ring 2). Ring 2 seviyesi en fazla (N/4+2) elemandan oluşmaktadır. Bu şekilde devam edilerek Ring 0'da her 2^i. düğüm kendinden sonra gelen 2^i. düğüme fazladan bir işaretçi ile bağlanırsa, Ring i seviyesinde en fazla ($N/2^i$+i) tane eleman vardır. Böylece arama işleminde aranan bir düğüme ulaşmak için zaman karmaşıklığı en fazla O(lg N) olur.

Atlamalı listedeki iyileştirme göz önüne alınarak Algoritma 9'da P=1/4 alınıp seviye oluşturulursa oluşan yapı ve üste doğru halkalar (Ring 0, Ring 1,.., Ring k) Şekil 4.5'deki gibi olur. N elemanlı bir atlamalı halkada (Şekil 4.5) Ring 0 seviyesinde toplam N tane düğüm vardır. Ring 0 seviyesindeki her dört düğümden biri kendinden sonra gelen dört düğümden birine fazladan bir bağ içerirse Ring 1 oluşur. Ring 1 seviyesinde en fazla (N/4+1) düğüm bulunur. Ring 1 seviyesindeki her dört düğümden biri kendinden sonra gelen dört düğümden birine fazladan bir bağ içerirse, Ring 2 oluşur, bu şekilde devam edilerek yapı oluşturulur. Böylece N elemanlı bir atlamalı halka veri yapısının yüksekliği yaklaşık olarak h=(lg N)/2 olur.

Munro et al. [15] tarafından önerilen deterministik 1-2 atlamalı liste yerine, 1-4 atlamalı liste alternatif olarak kullanılabilir. Yani, en alttan başlanarak her seviyedeki her dört düğümden 4.sü bir üst seviyeye çıkarılarak (Şekil 4.5 - Ring 0) deterministik bir atlamalı liste oluşturulur. Böylece oluşan yapının yüksekliği yaklaşık h=(lg N)/2 olur.



Şekil 4.5. Atlamalı Halka (Skip ring) (P=1/4 için)

Atlamalı liste (skip list) veri yapısındaki arama, ekleme, silme [4,5,6] işlemlerinin hepsi önerilen atlamalı halka (skip ring) veri yapısında da mümkündür.

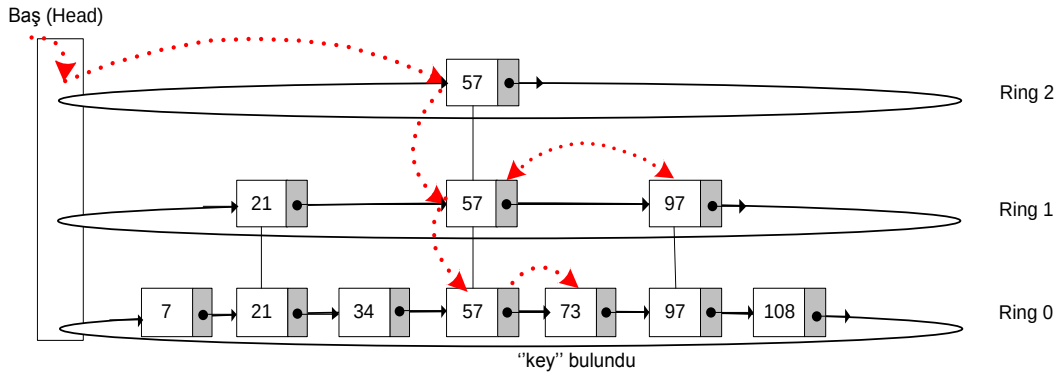
4.2.1. Düğüm arama

Önerilen atlamalı halka (skip ring) veri yapısında önceki 3. Bölüm'de anlatıldığı üzere her 2^i. (i=0,..,MaxLevel(15/31)) düğüm kendinden sonra gelen 2^i. düğüme bir işaretçi ile bağlanır. Böylece koni şeklinde bir yapı ortaya çıkar. Bundan dolayı arama işleminde aranan bir düğüme zaman karmaşıklığı en fazla O(lg N) olacak şekilde ulaşılır. Aramaya en üst seviyedeki halkadan başlanılarak alt seviyelerdeki halkalara doğru inilir.

Yeni veri yapısında arama işlemi için aşağıdaki adımların takip edilmesi gerekmektedir. Bu adımlar:

- Aranacak X (73) değerini belirle
- Atlamalı halka veri yapısı boş mu kontrol et
- Boş değilse X (73) düğümünü en üst seviyeden başlayarak aşağıya doğru her seviyede ara
- Eğer düğüm bulundu ise değeri döndür değilse false döndür.

{7,21,34,57,73,97,108} elemanlarından oluşan atlamalı halka veri yapısında "73" değerine sahip düğümün nasıl bulunacağı Şekil 4.6'da gösterilmektedir.



Şekil 4.6. Atlamalı halkada düğüm arama

Atlamalı halka veri yapısında arama, atlamalı liste veri yapısına benzediği için orada kullanılan algoritma [4] düzenlenerek gerçekleştirilebilir. Arama işleminde ilk dikkat edilmesi gereken atlamalı halka veri yapısında hiç eleman olmayabilir. Bundan dolayı öncelikle bu durum aşağıdaki gibi kontrol edilmelidir:

```
temp ← ring→head;
 if (temp→next[0]= ring→head) or (level<0)
     return false
```

Algoritma 8: Atlamalı halka veri yapısında düğüm arama

```
1:  SearchNode(ring, key)
2:    temp ← ring→head
3:    level ← ring→level
4:   if (temp→next[0] = ring→head) or (level<0)
5:        return false
6:   for i ← level downto 0 do
7:       while(temp→next[i]≠ ring→head and temp→next[i]→value < key)
8:            temp ← temp→next[i]
9:    temp ← temp→next[0]
10:  if (temp ≠ ring→head and temp→value = key)
11:       return true;
12:  return false;
```

4.2.2. Düğüm ekleme

Yeni bir düğüm eklemek için önce eklenecek düğümün hangi pozisyona ekleneceğinin bulunması gerekmektedir. Bunun için arama işlemi söz konusudur. Arama işleminde aranan bir düğüme zaman karmaşıklığı en fazla O(lg N) olacak şekilde ulaşılır. Bundan dolayı düğüm ekleme işleminin zaman karmaşıklığı da O(lg N) olacaktır.

Yeni bir düğüm ekleme işlemi için aşağıdaki adımların takip edilmesi gerekmektedir.

Bu adımlar [62]:

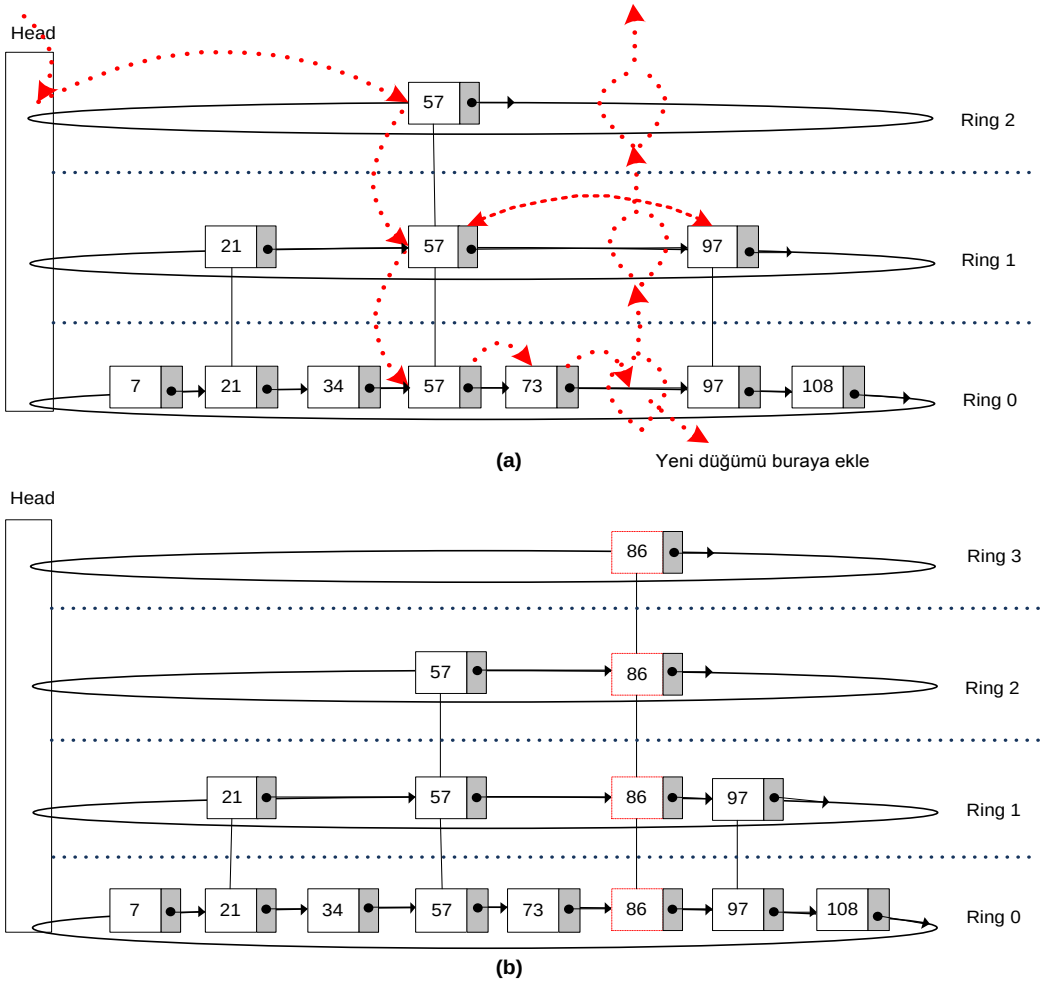
- Eklenecek "key" değerini belirle
- "key" değerini en üst seviyeden başlayarak alt seviyelere doğru ara
- Eğer "key" değeri bulundu ise false döndür ve çık
- Değilse yeni bir düğüm oluştur (86), rastgele seviye oluştur (randomLevel), Oluşturulan bu düğümün değerine "key" değerini ata
- 86 düğümünden önceki düğüm 86 gösterecek ve 86 düğümü de kendinden sonraki gelen düğümü gösterecek şekilde işaretçilere güncelle.
- Oluşturulan düğümü gereken diğer seviyelere de ekle.
- Yapıyı güncelle.

Düğümün eklenecek konumu bulunduktan sonra bu düğüm için seviye (level) oluşturmak gerekmektedir. Seviye oluşmak için, 3. Bölüm'de anlatıldığı gibi ya düğüm sayısından faydalanılır (Algoritma 2) ya da olasılıksal olarak (Algoritma 1) seviye üretilir. Eğer düğüm sayısı gerekli değilse ya da bilinmiyorsa Algoritma 9, eğer düğüm sayısı biliniyorsa ya da gerekli ise Algoritma 2 tercih edilir. Olasılıksal olarak seviye üreten algoritma (Algoritma 1), Pugh'un önerdiği gibi P=1/2 alınarak kullanılırsa çok yüksek seviyeler üretmektedir. Bunun yerine, bu algoritmanın (Algoritma 1) güncellenmiş hali olan algoritma (Algoritma 9) tercih edilmelidir. 3. Bölüm'deki random_level() algoritması (Algoritma 1), P=1/4 ve MaxLevel=32 alınarak Algoritma 9'daki gibi güncellenmiştir. Bu tezde önerilen ve P=1/4 alındığında elde edilen seviyeler, düğüm sayısından seviye oluşturmaya yakın olduğu için olasılıksal seviye oluşturma da tercih edilebilir.

Algoritma 9: Rastgele Seviye (Level) Üretme

```
1: random_level()
2:   level← 0;
3:   frand←rand()/RAND_MAX
4:         { frand değeri [0..1) arasında }
5:   while (frand<P) and (level<MaxLevel)
6:         { MaxLevel=32, P= 1/4 veya 1/2 }
7:      level←level+1;
8: return level;
```

{7,21,34,57,73,97,108} elemanlarından oluşan atlamalı halka (Skip ring) veri yapısına bir düğümün (“86”) nasıl ekleneceği Şekil 4.7’de gösterilmektedir.



Şekil 4.7. (a) Yeni düğüm ekleme işlemi (b) Düğüm eklenmiş hali

Algoritma 10: Atlamalı halka veri yapısına yeni düğüm ekleme

```
1:  InsertNode(ring, key)
2:      temp←ring→head ,
3:      level←ring→level
4:      update[MaxLevel +1]
5:  for i ← level downto 0 do
6:    while (temp→next[i] ≠ ring→head and temp→next[i] →value < key)
7:        temp←temp→next[i];
8:    update[i] ← temp;
9:  end for
10: temp←temp→next[0];
11:  if (temp = ring→head or temp→value ≠ key)
12:      { random_level() algoritması ile yeni seviye (level) üretme }
13:    newlvl ← random_level();
14:    if (newlvl > level)
15:      for i←level+1 to newlvl do
16:          update[i] ←temp;
17:      level←newlvl;
18:     end if
19:       {Yeni düğüm oluşturma}
20:    temp←make_node(newlvl,value);
21:   for i←0 to newlvl do
22:     temp→next[i]←update[i]→next[i];
23:     update[i]→next[i]←temp;
24:   end for
25:  end if
```

Algoritma 9'daki MaxLevel değeri atlamalı halkanın alabileceği en yüksek seviyesidir. Bu tez çalışmasında MaxLevel değeri 32 alınmıştır. Önerilen **random_level()** algoritması (**Algoritma 9**) düğüm eklerken seviye (level) oluşturmak için olasılıksal olarak 0..MaxLevel arası rastgele bir seviye değeri üretir. Bu seviye (level) üretme işlemi şöyle gerçekleşir: level (seviye) başlangıç değeri 0

alınır. Daha sonra rastgele [0..1) arası ondalıklı bir değer üretilir. Bu üretilen değer, **P (1/4)** eşik değerinden küçük ve level<MaxLevel olduğu sürece level değeri 1 artırılır. İşlem bu şekilde devam eder. Ne zaman üretilen sayı, **P** eşik değerinden büyük olursa ya da level değeri MaxLevel değerini ulaşmışsa level üretilmiş olur. Eklenecek düğüm, üretilen bu level değeri esas alınarak yapıya eklenir. Bu seviye üretmeyi şöyle bir örnekle açıklayacak olursak; eklenecek bir düğüm için bir torbaya 3 siyah, 1 tane beyaz bilye atılsın. Çekilen bilye tekrar torbaya koyulmak şartıyla torbadan bilye çekiliyor beyaz çekilirse level değeri bir artırılıyor, işlem bu şekilde beyaz bilye çekildikçe devam ediyor. Siyah bilye çekilince bitiyor. Yani 3 defa arka arkaya beyaz bilye çektik diyelim, level değeri 3 alınıp düğüm level=3 seviyesine ekleniyor.

4.2.3. Düğüm silme

Bir düğümü silmek için önce silinecek düğümün bulunması gerekmektedir. Bunun içinde arama işlemi söz konusudur. Arama işleminde aranan bir düğüme zaman karmaşıklığı en fazla O(lg N) olacak şekilde ulaşılır. Bu yüzden düğüm silme işlemi için zaman karmaşıklığı en fazla O(lg N) olacaktır.

Atlamalı halka (Skip ring) veri yapısındaki düğüm silme işlemi, atlamalı listedeki düğüm silme işlemine benzer bir yapıdadır [4, 5].

Düğüm silme algoritmasında arama algoritmasında olduğu gibi ilk dikkat edilmesi gereken husus atlamalı halka veri yapısında hiç eleman olmayabilir. Bundan dolayı öncelikle bu durum şu şekilde kontrol edilmelidir:

```
temp ← ring→head;
 if (temp→next[0]= ring→head)
      return false
```

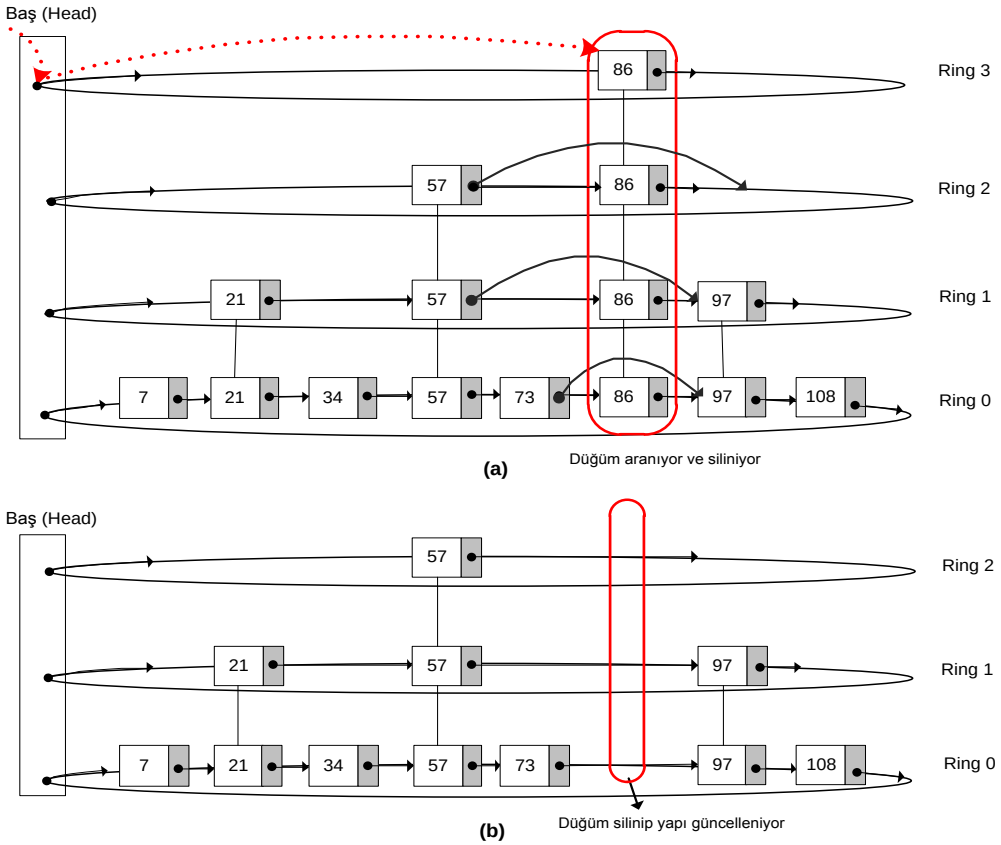
Silme işlemi için ise aşağıdaki adımların takip edilmesi gerekmektedir. Bu adımlar şöyledir:

- Silinecek “key” değerini belirle
- “key” değerini en üst seviyeden başlayarak alt seviyelere doğru ara
- Eğer “key” değeri bulunamadı ise false döndür ve çık
- Eğer “key” değerli düğüm (86) bulundu ise 86’dan önce gelen düğümün işaretçisini 86’dan sonra gelen düğümü gösterecek şekilde güncelle.
- Düğümü (86) olduğu tüm seviyelerden sil
- Düğümü (86) bellekten sil
- Yapıyı güncelle.

Algoritma 11: Atlamalı halka veri yapısında düğüm silme

```
1:  DeleteNode(ring, key);
2:    temp←ring→head
3:    level←ring→level
4:    update[MaxLevel+1];
5:   if (temp→next[0] = ring→head) or (level<0)
6:      return false
7:   for i←level downto 0 do
8:     while (temp→next[i] ≠ ring→head and temp→next[i]→value < key)
9:          temp←temp→next[i];
10:   update[i] ←temp;
11:  end for
12:   temp←temp→next[0];
13:  if (temp→value = value)
14:   for i←0 to level do
15:     if (update[i]→next[i] ≠ temp)
16:         break;
17:    update[i]→next[i] = temp→next[i];
18:  end for
19:  free(temp);
20:  while (level> 0 and temp→next[level] = ring→head)
21:        level←level-1;
22:  end if
```

{7,21,34,57,73,86,97,108} elemanlarından oluşan Atlamalı halka veri yapısından bir düğümün "86" nasıl silineceği Şekil 4.8'de gösterilmektedir.



Şekil 4.8. (a) Düğüm silme işlemi (b) Atlamalı halkayı güncelleme

4.3. Atlamalı Halka (Skip Ring) Veri Yapısının Özellikleri

Atlamalı halka (skip ring) veri yapısı dairesel bağlı liste ve atlamalı liste (skip list) veri yapılarından faydalanılarak oluşturulmuş bir veri yapısıdır. Bundan dolayı bu veri yapısının özelliklerinin ortaya konulması gerekmektedir. Bu özellikler ele alınırken hem veri yapısının bir matematiksel model olarak özellikleri üzerinde durulacaktır hem de bu veri yapısı üzerinde işlemler yapılırken taşıdığı özellikler [62] ele alınacaktır (P=1/2 alınmıştır).

Teorem 1. $0 \le j \le MaxLevel$ ve $1 \le k \le MaxIndex_j$ olmak üzere S(j,k) atlamalı halka (skip ring) veri yapısı olsun. S(0,k) atlamalı halka (skip ring) veri yapısının en alt seviyesindeki bağlı liste olsun ve 0 seviyesindeki düğümlerin indeks kümesi $I(S(0,k))=\{i_1,i_2,\ldots,i_k,..,i_{MaxIndex}\}$. Eğer $\frac{i_k}{2^j} \in Z^+$ ise, $i_k \in I(S(j,k))$ olur (I(S(j,k)) kümesi Seviye j'nin indeks kümesidir).

İspat. Bu teoremin ispatı tümevarım yöntemi ile yapılabilir.

1.adım: Seviye 0'dan bazı düğümler seviye 1'e çıkarılır. N seviye 0'da yer alan düğüm sayısı olmak üzere bir üst seviyeye çıkarılacak olan düğümlerin indeksleri eğer N çift ise, {2, 4, …, N} olur ve Seviye 1' deki indeksler {2/2,4/2,…,N/2}=I(S(1,...)) şeklinde olur. Eğer N tek ise, Seviye 1'e çıkarılacak düğümlerin indeksleri {2,4,…, (N-1)} olur ve Seviye 1'deki indeksleri $\left\{\frac{2}{2},\frac{4}{2},...,\left\lfloor\frac{N}{2}\right\rfloor=\frac{N-1}{2}\right\}$=I(S(1,…)) şeklinde olur. $\lfloor.\rfloor$ fonksiyonu taban fonksiyonudur.

2. adım: Seviye=j=MaxLevel-1 için durumun doğru olduğu kabul edilsin.

3. adım: Seviye MaxLevel-1 yer alan düğümlerin kümesi I(S(MaxLevel-1,…)) olmak üzere bu seviyede yer alan düğümlerin indeksleri sıralı olarak {1,2,…,$MaxIndex_{MaxLevel-1}$} şeklinde olsun. Eğer $MaxIndex_{MaxLevel-1}$ çift ise, MaxLevel seviyesine çıkarılacak düğümlerin indeks listesi {2,4,…,$MaxIndex_{MaxLevel-1}$}=I(S(MaxLevel-1,…)) olacaktır ve MaxLevel seviyesindeki indeks kümesi {2/2, 4/2, …, ($MaxIndex_{Maxlevel-1}$)/2}=I(S(MaxLevel,…)) şeklinde olacaktır. Eğer $MaxIndex_{MaxLevel-1}$ tek ise, MaxLevel seviyesine çıkarılacak düğümlerin indeks listesi {2,4,…,($MaxIndex_{MaxLevel-1}$-1)}=I(S(MaxLevel-1,…)) olacaktır ve MaxLevel seviyesindeki indeks kümesi {2/2, 4/2, …, ($MaxIndex_{Maxlevel-1}$-1)/2}=I(S(MaxLevel,…)) şeklinde olacaktır.

Teorem 2. $0\le i\le MaxLevel$ ve $1\le j\le MaxIndex_i$ olmak üzere S(i,j) bir Skip ring veri yapısı olsun. MaxLevel en yüksek seviyenin seviye indeksidir ve |S(i,..)| ise, i. seviyedeki düğüm sayısını verir. N en alttaki seviyedeki düğüm sayısı olmak üzere

$$\sum_{i=1}^{\lceil \lg N\rceil}\sum_{j=1}^{MaxIndex_i}1 \le N.$$

İspat. |S(0,…)|=N ve $\lceil.\rceil$ fonksiyonu ise, tavan fonksiyonudur. Seviye 0 düğümlerinin indeksi çift olanlar bir üst seviyeye çıkarılır ve indeksleri ise, Seviye 0 indekslerin yarısı olarak belirlenir. Bu durumda Seviye 1' de yer alan düğüm sayısı Seviye 0' da yer alan düğüm sayısının yarısı veya yarısının bir eksiği kadardır. Bunun anlamı şudur; Seviye 1'de yer alan düğüm sayısı $\left\lfloor\frac{|S(0,..)|}{2}\right\rfloor$ olur. Seviye 2'de yer alan düğüm sayısı da $\left\lfloor\frac{|S(1,..)|}{2}\right\rfloor$ şeklinde olacaktır. Böyle devam edilerek Seviye

MaxLevel'da yer alan düğüm sayısı $\left\lfloor \frac{|S(MaxLevel-1,..)|}{2} \right\rfloor$ olur. Bu durumda Seviye 0 hariç diğer seviyelerde yer alan düğüm sayısı $\sum_{i=1}^{MaxLevel} \left\lfloor \frac{|S(i,...)|}{2} \right\rfloor$ ve MaxLevel=$\lceil N \rceil$ olur. Bu toplam $\sum_{i=1}^{MaxLevel} \left\lfloor \frac{|S(i,...)|}{2} \right\rfloor$ için oluşabilecek maksimum değer N sayısının 2'nin kuvveti olması durumudur. $N=2^r$ ve $r \in Z^+$ olsun. Bu durumda her seviyede yer alan düğüm sayıları $|S(1,\ldots)|=2^{r-1}$, $|S(2,\ldots)|=2^{r-2}$, …, $|S(MaxLevel,\ldots)|=1$ şeklinde olur ve $\lceil \lg N \rceil = r$ olur. Seviye 0 hariç diğer seviyelerde yer alan düğüm sayısı $\sum_{i=1}^{r-1} 2^i = \frac{1-2^r}{1-2} = 2^{\lceil \lg N \rceil} - 1 = 2^{\lg N} - 1 = N - 1$ olur.

Teorem 3. S bir atlamalı halka (skip ring) olmak üzere bu atlamalı halkada yer alan düğüm sayısı $\sum_{i=0}^{\lceil \lg N \rceil} \left\lfloor \frac{N}{2^i} \right\rfloor = N + \sum_{i=1}^{\lceil \lg N \rceil} \left\lfloor \frac{N}{2^i} \right\rfloor$ ve i. seviyede $\left\lfloor \frac{N}{2^i} \right\rfloor$ olur.

İspat. $N_0=N$ olmak üzere N_1 Seviye 1'de yer alan düğüm sayısı, N_2 Seviye 2' de yer alan düğüm sayısı ve böyle devam ederek son seviyede $N_{\lceil \lg N \rceil}$ tane düğüm olsun.

$$N_i = \left\lfloor \frac{N_{i-1}}{2} \right\rfloor = \begin{cases} \frac{N_{i-1}}{2} & \frac{N_{i-1}}{2} \in Z^+ \quad ise \\ \frac{N_{i-1}-1}{2} & \frac{N_{i-1}-1}{2} \in Z^+ \quad ise \end{cases}$$

olarak verilen bir taban fonksiyonu olmak üzere seviyeler arası düğüm sayıları arasındaki bağıntı

$$N_1 = \left\lfloor \frac{N_0}{2} \right\rfloor,\ N_2 = \left\lfloor \frac{N_1}{2} \right\rfloor,\ \ldots,\ N_{\lceil \lg N \rceil} = \left\lfloor \frac{N_{\lceil \lg N \rceil - 1}}{2} \right\rfloor$$

şeklinde olur ve bu da iddianın doğruluğunu göstermektedir.

Teorem 4. S bir atlamalı halka (skip ring) olmak üzere eğer $\forall k_1$, k_2 düğüm indeksleri ve i ve j seviye indeksleri olmak üzere aşağıda durumlar geçerlidir.

a) Eğer i=j ise, $k_1<k_2$ için $S(i,k_1) \le S(j,k_2)$ olur.

b) $i<j$ ve $k_1 \le 2^{j-i}k_2$ ise, $S(i,k_1) \le S(j,2^{j-i}k_2)$ veya $j<i$ ve $k_1 \le 2^{j-i}k_2$ ise, $S(j,k_1) \le S(j,2^{i-j}k_2)$ olur.

İspat. İlk olarak atlamalı halkanın her seviyesinde yer alan veri yapısı bir bağlı listedir ve bu bağlı listeye düğümler küçükten büyüğe sıralı olarak yerleştirildiği için teoremin a) şıkkı ispatlanmış olur.

b) En alt seviyede yer alan düğümler içerisinde indeksi çift olanlar bir seviyeye çıkarılır ve indeksler ise, alt seviyedeki, düğümlerin indeksi 2' ye bölünerek elde edilir. Bu durumda i<j ve j-i>0 ise, j. seviyedeki düğümün indeksi j-i defa 2' ye bölünmüştür. Bundan dolayı indeks 2^{j-i} ile çarpıldığında en alt seviyedeki düğümün indeksi elde edilir. Bu durumda eğer i<j ise, k_1 indeksi k_2 indeksinin 2^{j-i} katıdır. Bu durumda $k_1 \leq 2^{j-i}k_2$ olur ve bağlı listenin sıralı olma özelliğinden dolayı $S(i,k_1) \leq S(j,2^{j-i}k_2)$ özelliği sağlanır. Diğer durum da bunun tersidir.

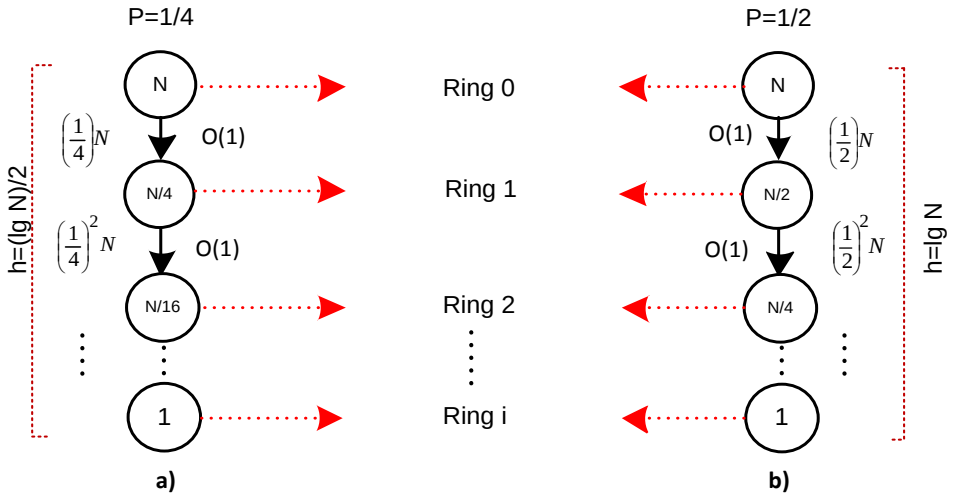
Teorem 5. En alt seviyede (Seviye 0) yer alan düğümlerin sayısı N olmak üzere atlamalı halka veri yapısında arama işleminde en uzun yolun uzunluğu $\lceil \lg N \rceil + 1$ şeklindedir (P=1/2 için).

İspat. En üst seviyede yer alacak olan düğüm sayısı normal şartlar altında 1 olacaktır. Bu durumda bu seviyede bir tane karşılaştırma ile bir alt seviyeye inip inilmeyeceğine karar verilir. Eğer aranan düğüm bu seviyede ise bulunmuş olur ve yol bitmiş olur. Eğer bitmezse, bir alt seviyeye iner ve alt seviyede yapacağı karşılaştırma ~1 olur. Bu şekilde en alt seviyeye kadar inildiğinde atlamalı halka veri yapısındaki seviye kadar yani $\sim \lceil \lg N \rceil + 1$ bir yol takip edilir.

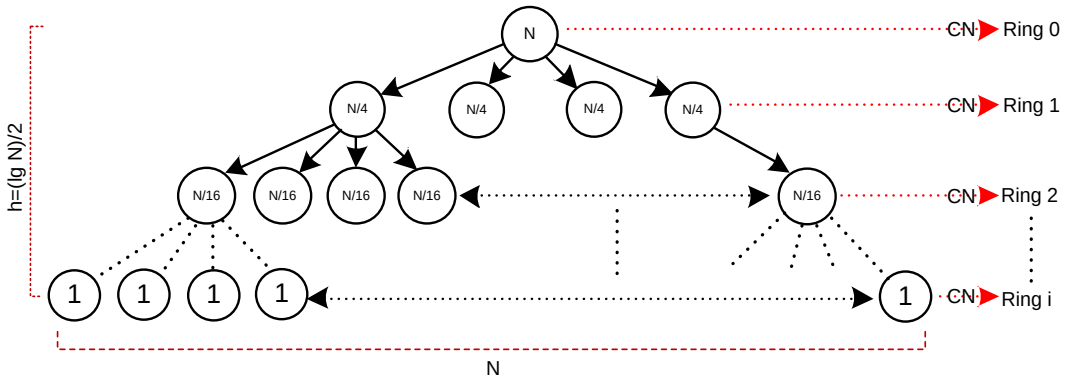
4.4. Atlamalı Halka (Skip Ring) Veri Yapısının Zaman Analizi

Atlamalı halka veri yapısının zaman karmaşıklık analizi Şekil 4.9, Şekil 4.10 ve Şekil 4.11 dikkate alınarak yapılabilir. Atlamalı halka veri yapısında en üst seviyedeki (Ring i) düğümlere erişim için zaman karmaşıklığı Θ (1)'dir. Yani en üst seviyedeki düğümler için arama işlemleri Θ (1) zaman karmaşıklığında gerçekleşir. Atlamalı halka veri yapısı oluşturulurken Şekil 4.9.a'da olduğu gibi P=1/4 alınırsa (yani bir halkadaki her dört düğümden biri üstteki halkaya çıkarılırsa); Arama, ekleme, silme işlemleri için zaman karmaşıklığı T(N)=O(h)=O(1/2lgN)=O(lg N) olur. Şöyle ki N elemanlı bir atlamalı halka üzerinde Ring 0 düzeyinden O(1) süresinde N/4 elemanlı Ring 1 düzeyine, Ring 1 düzeyinden O(1) süresinde N/16 elemanlı Ring 2 düzeyine ve benzer şekilde devam edilerek Ring i düzeyine geçilir.

Bu işlemler için zaman karmaşıklığı = O(1)* O(1/2lgN) = O(lg N) olur.



Şekil 4.9. Atlamalı halkada P=1/4 ve P=1/2 için düğümlerin seviyelere dağılımı.



Şekil 4.10. Atlamalı halka veri yapısının ağaç şeklinde görünümü

Şekil 4.10'daki gibi N elemanlı bir atlamalı halka veri yapısını oluşturmak için zaman karmaşıklığı T(N)=O(hN)=O(1/2NlgN)= O(Nlg N) olur. Çünkü N>1 için (Bazı K değerleri için $N=4^K$ olduğu varsayılırsa)

$T(N) = 4T(N/4)+N$ {Ring 1}

$= 4(4T(N/16)+N/4)+N$ {Genişlet}

$= 4^2T(N/16)+2N$ {Ring 2}

$= 4^2(4T(N/64)+N/16+2N$ {Genişlet}

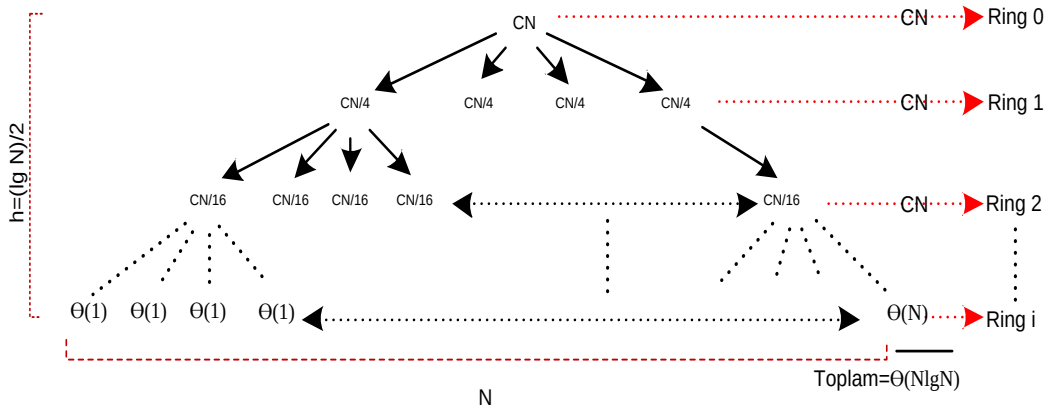
$= 4^3T(N/4^3)+3N$ {Gözlemle} {Ring 3}

$= 4^KT(N/4^K)+KN$ {Ring K veya Ring i}

$= 4^{\log_4 N} T(N/N)+N\log_4 N$

$= N+ 1/2NlgN$

olur.



Şekil 4.11. Atlamalı halka veri yapısının ağaç şeklinde analizi

5. DENEYSEL BULGULAR VE TARTIŞMA

Tezin bu bölümünde, önerilen yeni veri yapısının kullanılabileceği bazı alanlara ait uygulamalar gerçekleştirilmiştir. İlk örnek uygulamada, önerilen atlamalı halka veri yapısı ile bazı ağaç temelli veri yapıları karşılaştırılmıştır. İkinci örnek uygulamada, atlamalı halka veri yapısı kullanılarak sıralama işlemi gerçekleştirilmiş ve elde edilen sonuçlar bazı sıralama algoritmaları karşılaştırılmıştır. Üçüncü örnek uygulamada, arama frekansına bağlı yeni bir arama algoritması önerilmiş uygulamaları yapılıp sonuçları kıyaslanmıştır. Son olarak ise, atlamalı halka veri yapısının görev zamanlayıcı algoritmalarında (process scheduler) kullanılabileceği gösterilmiş ve yeni bir görev zamanlayıcı algoritması önerilmiştir.

5.1. Örnek Uygulama 1: Atlamalı Halka (Skip Ring) ve Ağaç Veri Yapılarının (Binary Search Tree, Red-Black Tree) Karşılaştırılması

Tezin bu bölümünde, önerilen atlamalı halka (skip ring) veri yapısı ile ağaç veri yapıları (kırmızı-siyah ağaç, ikili arama ağacı gibi) uygulamalı olarak karşılaştırılmış sonuçları sunulmuştur. Elde edilen sonuçlar göstermektedir ki, atlamalı halka (skip ring) veri yapısı çoğunluğu sıralı yada ters sıralı olan verilerde ikili ağaçlardan, kırmızı-siyah ağaçlardan ve atlamalı liste (skip list) veri yapılarından performans olarak daha iyidir.

Kırmızı-siyah ağaçlar (Şekil 2.4.a) kendi kendini dengeleyen ikili bir ağaç türüdür. İkili arama ağaçlarının sahip oldukları özelliklerin yanında, 2. Bölüm'de anlatılan ek özelliklere de sahiptirler [36, 38].

Kırmızı-siyah ağaçlar kullanışlı bazı özelliklere sahiptir. İlk olarak, kırmızı-siyah ağaçta arama işlemi O(log N) sürede gerçekleşir. İkinci olarak, ağaç kendi kendini dengelediğinden yaklaşık olarak sol ağacın uzunluğu, sağ ağacın uzunluğuna eşittir [39].

Kırmızı-siyah ağaç (red-black tree) veri yapısında her düğüm ekleme ve silme işleminde ağaçta dengenin sağlanması için kırmızı-siyah ağaç kurallarının korunması gerekmektedir. Bu özellikleri korumak için düğüm ekleme ve silme işlemlerinde ağaçta sola döndürme(ler), sağa döndürme(ler) ve yeniden renklendirme yani renk değiştirme (kırmızı-siyah) işlemleri gerçekleştirilir [36, 37, 39]. Bazen bir düğüm

ekleme işleminde çok sayıda döndürme ve renk değişikliği işlemi gerçekleşebilir. Bu işlemler kırmızı-siyah ağaçların performansını olumsuz etkiler. Yani, N elemanlı bir kırmızı-siyah ağaçta en fazla O(log N) olan düğüm ekleme ve silme işlemlerinin zaman karmaşıklığı üzerine bu maliyetin de eklenmesi gerekmektedir.

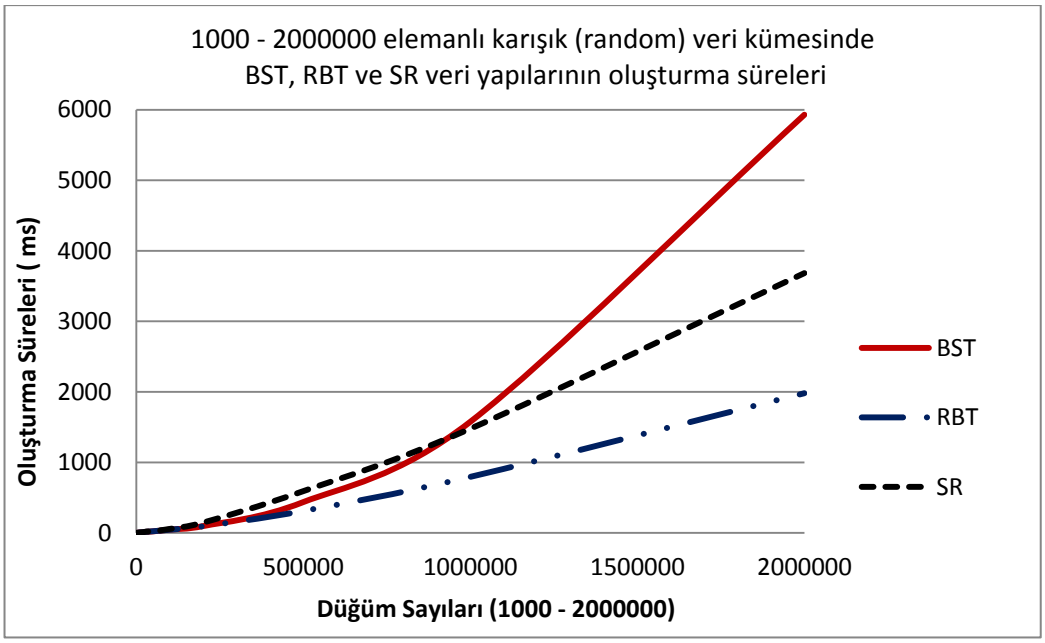
5.1.1. Ağaç veri yapıları (ikili arama ağaçları, kırmızı-siyah ağaçlar gibi) ve atlamalı halka (skip ring) veri yapısının performansının uygulamalı karşılaştırılması

SR (Skip Ring-Atlamalı halka), RBT (Red-Black Tree-Kırmızı-siyah ağaç) ve BST (Binary Search Tree-İkili arama ağacı) veri yapılarını karşılaştırmak için geliştirilen uygulamada, farklı veri kümeleri (Rastgele oluşturulmuş karışık veri kümeleri, sıralı veri kümeleri, ters sıralı veri kümeleri) kullanılmıştır. Bu veri kümeleri 1000 elemandan başlayarak 200000 elemana kadar çoğaltılıp her üç veri yapısının oluşturulma süreleri elde edilmiştir. Elde edilen sonuçlar Çizelge 5.1, Çizelge 5.2, Çizelge 5.3'te görülmektedir. Çizelgelerdeki oluşturulma süreleri milisaniye (ms) cinsindendir. Aynı şekilde veri kümeleri 2000000'a kadar çıkarılıp Şekil 5.1, Şekil 5.2 ve Şekil 5.3'teki sonuçlar elde edilmiştir. Elde edilen sonuçlar aynı veri kümesi üzerinde ve aynı ortamda elde edilmiştir.

Çizelge 5.1. Rastgele üretilen diziler için BST, RBT ve SR oluşturma süreleri

Düğüm Sayısı	1000	5000	10000	20000	30000	50000	100000	200000
BST	0	3.2	6.2	9.2	15.2	21.8	47	96.6
RBT	0	3.4	6.4	9.6	16.5	23.5	45.2	93.5
SR	0	2.1	5.7	9.4	18	24	56	146

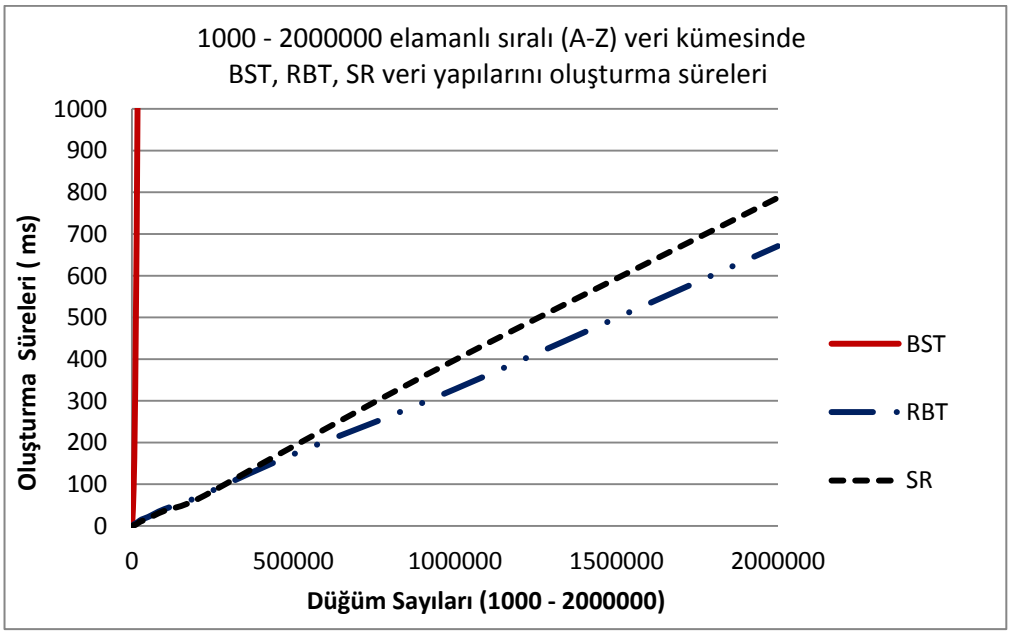
Çizelge 5.1, Çizelge 5.2, Çizelge 5.3 ve Şekil 5.1'deki sonuçlar incelendiğinde atlamalı halka (skip ring) veri yapısının performansının sıralı ve ters sıralı verilerde daha iyi olduğu görülmektedir. Asimptotik yaklaşımda her üç veri yapısı için rastgele karışık verilerden yapının oluşturulma karmaşıklığı en fazla O(NlgN)'dir. Asimptotik yaklaşımda, zaman karmaşıklığı hesaplanırken en büyük dereceli terim belirleyicidir. Yani, N^2, $4N^2$ ve $N^2+N+2000000$ ifadeleri asimptotik olarak ele alındığında zaman karmaşıklığı (time complexity) aynı olup $O(N^2)$ olur. Çizelge 5.1, Çizelge 5.2 ve Çizelge 5.3'teki sonuçlar göstermektedir ki asimptotik olarak aynı zaman karmaşıklığında olan farklı algoritmaların gerçek bilgisayar uygulamalarında çalışma süreleri farklı olabilmektedir.



Şekil 5.1. Atlamalı halka **(**Skip Ring), İkili arama ağaçları (Binary Search Tree) ve Kırmızı-siyah ağaç (Red-black tree) performans karşılaştırması

Çizelge 5.2. Sıralı (A-Z) diziler kullanarak BST, RBT ve SR oluşturma süreleri

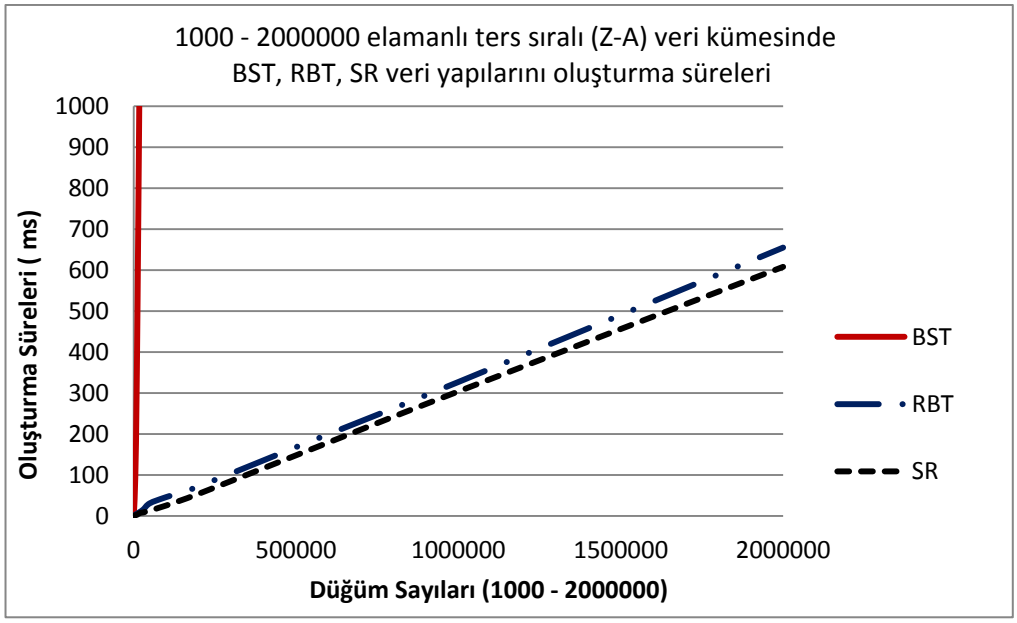
Düğüm Sayısı	1000	5000	10000	20000	30000	50000	100000	200000
BST	10	88	327	1315	2971	8387	16347	65320
RBT	0	3	5.4	10.3	15,5	21,5	40,4	68
SR	0	1.8	3.2	7.4	11.6	18	36	63



Şekil 5.2. Sıralı (A-Z) diziler üzerinde SR, RBT ve BST için performans karşılaştırması.

Çizelge 5.3. Ters sıralı (Z-A) diziler kullanarak BST, RBT, SR oluşturma süreleri

Düğüm Sayısı	1000	5000	10000	20000	30000	50000	100000	200000
BST	12	93	359	1326	2989	8174	16347	65312
RBT	0	3	5.4	10.3	15	31	46	68
SR	0	2	2.8	7.4	9.6	13,5	31	54



Şekil 5.3. Ters sıralı (Z-A) diziler üzerinde SR, RBT ve BST için performans karşılaştırması.

Çizelge 5.2, Çizelge 5.3 ve Şekil 5.2, Şekil 5.3 incelendiğinde dengelenmemiş bir ağaç (binary search tree) için zaman karmaşıklığı (time complexity) $O(N^2)$ olmaktadır. Bunun sebebi, ikili arama ağacı sıralı ve ters sıralı veri kümelerinden oluşturulurken adeta bir bağlı liste oluşmasıdır (Şekil 2.4.b). Yani sola ya da sağa merdiven basamakları şeklinde uzanan bir yapı oluşur. Bu durum göz önüne alındığında, dengeli ağaçlar (kırmızı-siyah ağaç gibi) önem kazanmaktadır.

Çizelge 5.1, Çizelge 5.2 ve Çizelge 5.3'teki sonuçlar dikkate alındığında SR, RBT ve BST veri yapılarında düğüm ekleme ve silme işlemlerinde de SR veri yapısının performansının iyi olduğu görülecektir. Şöyle ki, bu veri yapılarının oluşturulması işlemi zaten düğüm ekleme işlemidir. Örneğin, 5000 düğümlü bir veri yapısı oluşturma işlemi için SR, RBT ve BST veri yapılarına 5000 tane düğüm ekleme işlemi yapılmaktadır. Buradan hareketle, SR veri yapısının düğüm ekleme işlem performansı, RBT ve BST veri yapılarından iyi olacaktır.

N tane düğümden oluşan dengeli ikili ağaçlarda (RBT, BST vs.) ağacın yüksekliği (h) olup, h = (lg N) olur. Bu tez çalışmasında önerilen atlamalı halka (skip ring) veri yapısında ise yükseklik Algoritma 1'deki P (olasılık-probability) eşik değerlerine bağlıdır. Daha önce bu tez kapsamında yapılan bir çalışmada [16], P eşik değerlerinin oluşan yapının yüksekliğine ve performansa etkisi incelenmiştir. Yapılan bu çalışmada performansın, P=1/4 alındığında çok daha iyi olduğu gösterilmiştir. P=1/2 alınınca, N düğümlü atlamalı halka (skip ring) veri yapısının yüksekliği dengeli ikili ağaçlardaki gibi h = (lg N) olmaktadır. Eğer P=1/4 alınırsa, yani Şekil 4.4'deki Ring 0 düzeyindeki her dört elemandan biri bir üst seviyeye (Ring 1) çıkarılırsa ve öyle devam edilerek atlamalı halka veri yapısı oluşturulursa, oluşan yapının (Şekil 4.5) yüksekliği yaklaşık h/2 olur. RBT ve BST veri yapılarında yükseklik h iken, atlamalı halka (skip ring) veri yapısında yükseklik h/2 olur. Ayrıca dengeli ağaçlarda düğüm ekleme ve silme işlemlerinde döndürme(ler) ve yeniden düzenlemeler fazladan maliyettir. Ayrıca kımızı-siyah ağaçlarda renklendirme, renk değiştirme işlemi söz konusudur. Bu da zaman maliyetini artırıcı bir etkendir. Bütün bunlar dikkate alındığında, atlamalı halka (skip ring) veri yapısında düğüm arama, ekleme ve silme işlemleri için performansın pratikte iyi olacağı aşikârdır.

Gerçekleştirilen uygulama sonucuna bakıldığında ağaç veri yapılarında dengeleme (balance) işleminin önemi görülmektedir. Özellikle Çizelge 5.2 ve Çizelge 5.3'e bakıldığında, dengesiz ağaçlarda (Şekil 2.4.b) sıralı veya ters sıralı verilerden ağaç oluşturulurken zaman karmaşıklığı (time complexity) $O(N^2)$ olmaktadır. Yani yapı bağlı liste (lidked list) haline dönmektedir. Ayrıca Çizelge 5.1, Çizelge 5.2 ve Çizelge 5.3'teki sonuçlar ve Şekil 5.1'deki grafik incelendiğinde atlamalı halka (skip ring) veri yapısının performansının sıralı ve ters sıralı verilerde ikili ağaçlar ve kırmızı-siyah ağaçlardan iyi olduğu görülmektedir.

5.2. Örnek Uygulama 2: Atlamalı Halka (Skip Ring) Veri Yapısı Temelli Sıralama İşlemi (Karşılaştırmalı Uygulama)

Tezin bu bölümünde, atlamalı halka veri yapısını kullanan bir sıralama algoritması geliştirilip bu algoritma diğer bazı sıralama algoritmaları ile uygulamalı olarak kıyaslanmıştır. Bu tez çalışmasında sıralama algoritmaları karışık, sıralı ve ters sıralı diziler üzerinde aynı bilgisayar ve veri kümesi üzerinde karşılaştırmalı olarak kıyaslanmıştır.

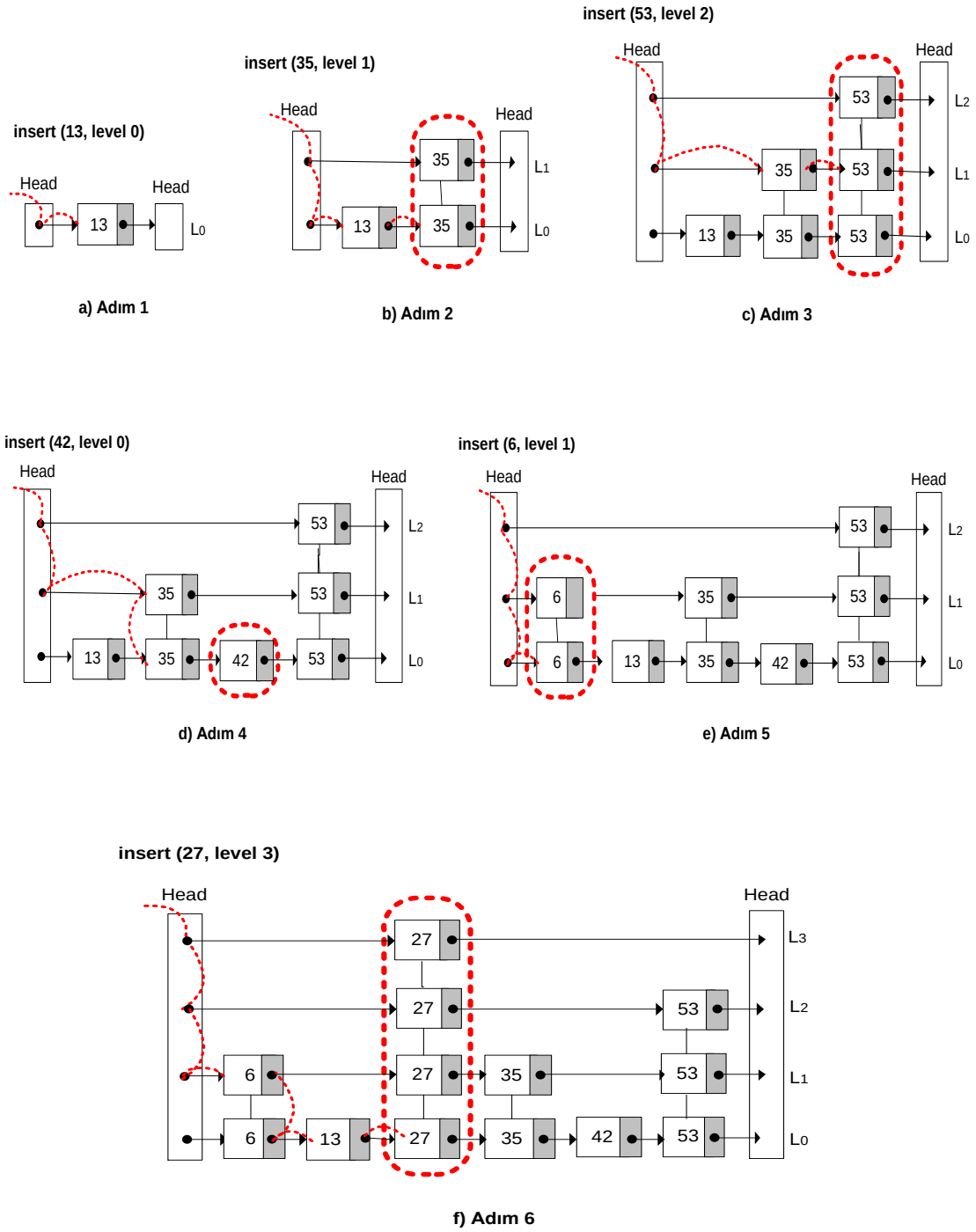
Önerilen yeni sıralama algoritmasının daha iyi anlaşılabilmesi için atlamalı halka (skip ring) veri yapısının işleyişinin ve seviyeli yapısının bilinmesi gerekmektedir. Bundan dolayı atlamalı liste ve atlamalı halka veri yapısının detaylı anlatıldığı 3. ve 4. Bölüm'e tekrar göz atılmalıdır.

Üzerinde arama, ekleme, silme işlemi yapılacak bir veri kümesinin sıralı olması çok önemlidir. Çünkü sıralı olmayan, N elemanlı bir veri kümesinde aranan bir elemanı bulmak için O(N) zaman harcamak gerekmektedir. Çünkü bu veri kümesinde doğrusal bir arama söz konusudur. Fakat aynı veri kümesi sıralı hale getirilip, ikili arama algoritması kullanılırsa, harcanan süre en fazla O(lg N) düzeyine düşer. Bu husular göz önünde bulundurulduğunda, çok büyük veri kümelerinde sıralama işleminin ne kadar önemli olduğu görülmektedir [46].

Sıralama algoritmalarını birçok yönden gruplandırmak mümkündür. Bu çalışmada, daha çok zaman karmaşıklığı (time complexity) üzerinde durulup gruplandırma ona göre yapılmıştır. Yani, $O(N^2)$ grubu sıralama algoritmaları (selection sort, bubble sort, shell sort gibi) ve O(NlgN) grubu sıralama algoritmaları (quick sort, heap sort, merge sort, red-black tree sort gibi) şeklinde gruplandırma yapılıp karşılaştırmalar ona göre yapılmıştır.

Önerilen yeni algoritma, atlamalı liste (skip list) ya da atlamalı halka (skip ring) veri yapılarına benzer bir yapıda sıralama gerçekleştirdiğinden atlamalı halka sıralama (skip ring sort) ismi verilmiştir. Bölüm 4'te anlatılan atlamalı halka veri yapısının koni şeklindeki seviyeli şekli göz önünde bulundurularak yeni sıralama algoritması geliştirilmiştir. Veriler sıralanırken yerleştirme işlemi seviyeler şeklinde gerçekleştirilmektedir. Böylece, bir eleman olması gereken sıraya en fazla O(lgN) zaman karmaşıklığında yerleştirilmektedir. Atlamalı halka sıralama (skip ring sort) ile N elemanlı bir veri kümesinin sıralanması ise en fazla O(NlgN) zaman karmaşıklığında gerçekleşmektedir.

Şekil 5.4'te {13,35,53,42,6,27} elemanlarından oluşan bir veri kümesinin atlamalı halka sıralama (skip ring sort) algoritması ile nasıl sıralandığı adım adım gösterilmiştir. Sıralanacak her eleman kendi yerine yerleştirilirken *random_level* (Algoritma 9) algoritması ile seviye (**level)** oluşturularak işlem gerçekleştirilmektedir.



Şekil 5.4. Atlamalı Halka Sıralama (Skip ring sort) (Adım adım)

5.2.1. Atlamalı halka sıralama (skip ring sort) ile verilerin sıralanması

Önerilen yeni sıralama algoritması (Algoritma 12) kullanılarak 1000-200000 elemanlı diziler üzerinde sıralama yapılmıştır. Elde edilen sonuçlar Çizelge 5.4, Çizelge 5.5 ve Çizelge 5.6'da görülmektedir. Çizelge 5.4'teki veri kümeleri rastgele (random) oluşturulmuş dizilerdir. Çizelge 5.5, küçükten büyüğe (A-Z) sıralı diziler

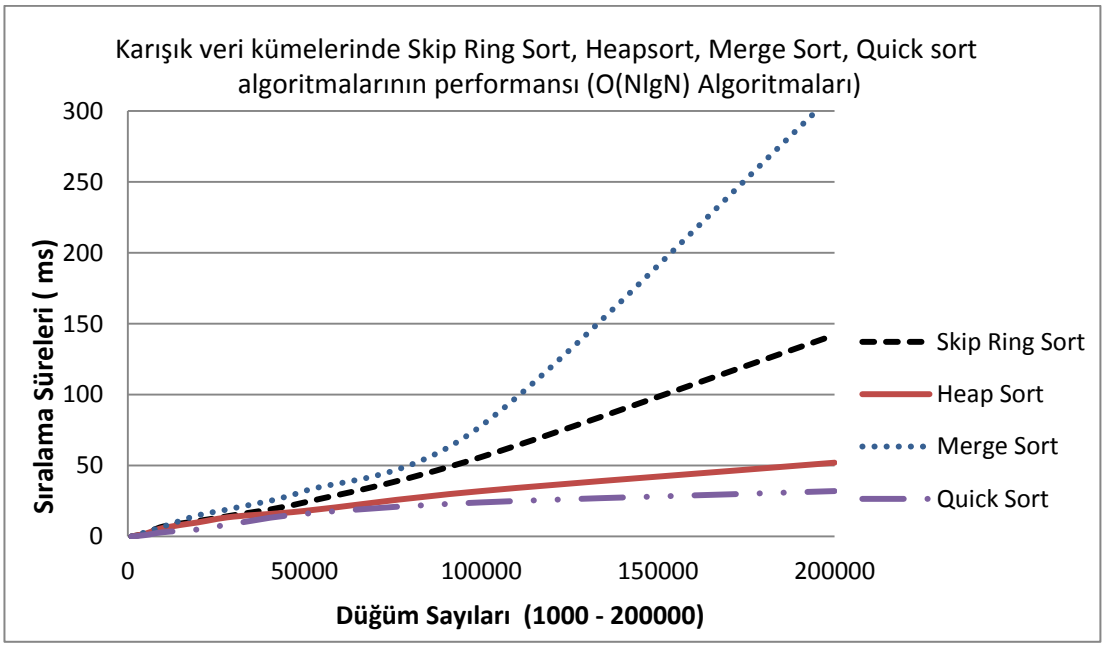
üzerinde yapılmış sıralama sonuçlarını göstermektedir. Çizelge 5.6 ise, tersten sıralı (Z-A) dizileri üzerinde yapılan sıralama sonuçlarını göstermektedir. Her üç çizelgede, önerilen algoritmanın sonuçları ile $O(N^2)$ grubu algoritmalar ve O(NlgN) grubu algoritmaların sonuçları kıyaslanmıştır. Gerçekleştirilen uygulamalardaki tüm sonuçlar milisaniye (ms) olarak ölçülmüştür. Çizelgelerdeki N (düğüm sayısı), SR (Skip Ring), BST (Binary Search Tree), RBT (Red-Black Tree), ms (mili saniye), lg N ($\log_2 N$) ve O (Büyük O notasyonu) ifadelerinin kısaltmasıdır.

Algoritma 12: Önerilen Sıralama Algoritması *{ Dizilerin Sıralanması }*

```
1:  SkipRingsort (sr, data[])
2:   for  j←0 to N-1 do  { N eleman sayısı}
3:        temp ← sr→head;
4:            update[MaxLevel + 1];
5:            for  i←sr→level downto 0 do
6:                while (temp→next[i] != sr→head && temp→next[i] →value < data[j] )
7:                     temp ← temp→next[i];
8:             update[i] ← temp;
9:            end for
10:           temp ← temp→next[0];
11:           lvl ← random_level();      { Random olarak seviye üretme }
12:            if (lvl > sr→level)
13:               for  i← sr→level + 1 to lvl do
14:                    update[i] ← sr→head;
15:               sr→level ← lvl;
16:            end if
17:           temp ← make_node (lvl, data[j]);     { Yeni düğüm oluşturma }
18:            for  i←0 to lvl do
19:                      temp→next[i] ← update[i] →next[i];
20:                      update[i] →next[i] ← temp;
21:           end for
22:  end for
```

Çizelge 5.4. 1000-200000 elemanlı karışık (random) veri kümesinde sıralama

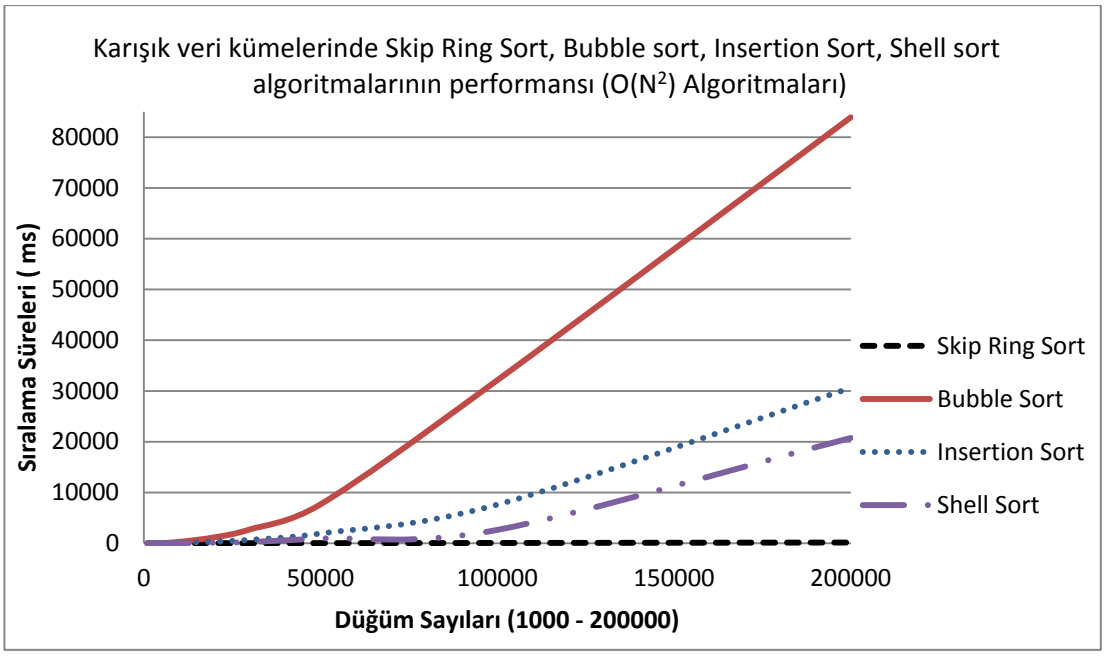
	O (N^2) Sıralama Algoritmaları			O (**NlgN**) Sıralama Algoritmaları			
N	Bubble	Insertion	Shell	**SR Sort**	Heap	Merge	Quick
1000	0-1	0-1	0-1	**0**	0	0	0
5000	47	18	15	**2**	2	3	1
10000	265	78	32	**7**	6	6-7	3
20000	1170	297	109	**11**	10	15	5
30000	2732	687	238	**15**	14	20	9
50000	7683	1923	934	**24**	18	32	16
100000	32129	7612	2637	**56**	32	78	24
200000	83928	30612	20748	**146**	52	312	32



Şekil 5.5. Karışık verilerde, O(NlgN) grubu bazı sıralama algoritmaları ile SR sıralama algoritmasının karşılaştırması

Çizelge 5.5. 1000-100000 elemanlı sıralı (A-Z) veri kümelerinde sıralama algoritmalarının çalışma sürelerinin karşılaştırması

	O (N^2) Sıralama Algoritmaları			O (**NlgN**) Sıralama Algoritmaları			
N	Bubble	Insertion	Shell	**SR Sort**	Heap	Merge	Quick
1000	8	0	0	**0**	0	0	0
5000	46	0	0-1	**2**	2	3-4	1
10000	162	0	1	**5**	4	6	1-2
20000	639	1	1	**9**	8	16	2-3
30000	1437	1	1-2	**14**	15	31	5
50000	3972	1	2	**15**	20	36	8
100000	15646	1	2-3	**31**	24	62	15



Şekil 5.6. Karışık verilerde, $O(N^2)$ grubu bazı sıralama algoritmaları ile SR sıralama algoritmasının karşılaştırması

Çizelge 5.6. 1000-100000 elemanlı ters sıralı (Z-A) verilerde sıralama süreleri

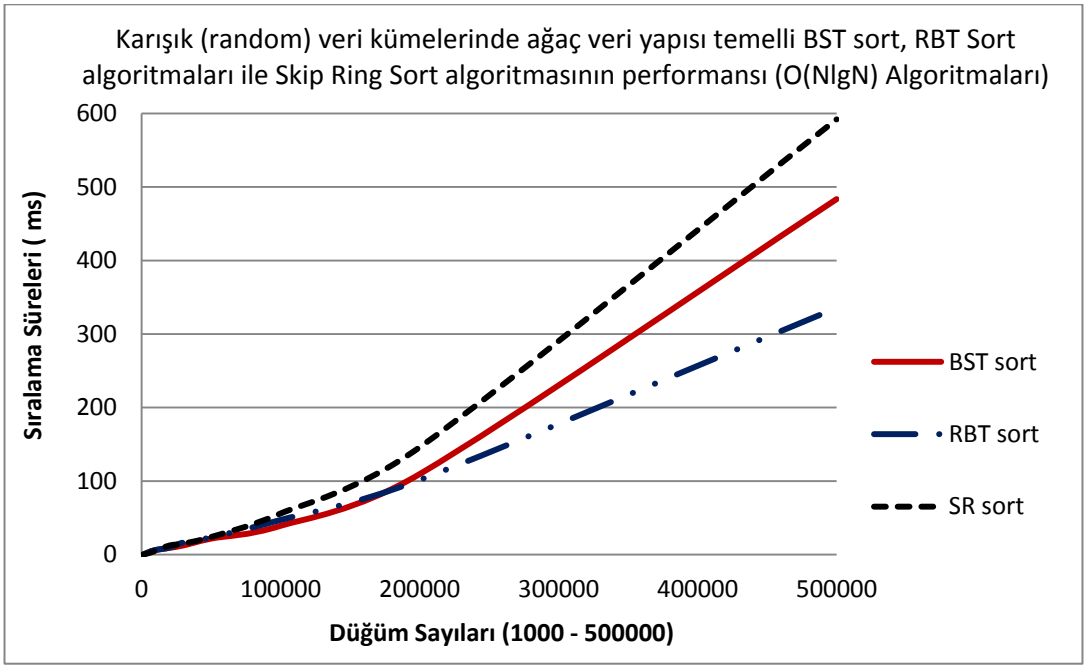
	O (N^2) Sıralama Algoritmaları			O (**NlgN**) Sıralama Algoritmaları			
N	Bubble	Insertion	Shell	**SR Sort**	Heap	Merge	Quick
1000	0-7	0-6	0-4	**0**	0	0	0
5000	47	38	16	**2**	2	3	1
10000	202	156	63	**4**	4	5	2
20000	842	624	218	**8**	6	9	4
30000	1919	1386	468	**11**	10	16	7
50000	5226	3835	1312	**15**	16	32	10
100000	21122	15272	5465	**31**	28	78	15

SR, RBT ve BST temelli sıralama işlemini gerçekleştirmek için rastgele (random), sıralı (A-Z) ve ters sıralı (Z-A) diziler kullanılmıştır. Kırmızı-siyah ağaçlarda ve ikili arama ağaçlarında sıralı veri elde etmek için ağaç oluşturulduktan sonra, 2. Bölüm'de anlatılan kök-ortada (in-order) gezinti işlemine ihtiyaç vardır. SR, RBT ve BST sıralama algoritmalarının sıralama süreleri rastgele oluşturulan 1000 elemanlı bir diziden başlanarak 100000 elemanlı bir diziye kadar devam ettirilip Çizelge 5.7'deki sonuçlar elde edilmiştir. Aynı şekilde 1000-500000 elemanlı rastgele (random) oluşturulmuş dizilerin sıralanma sonuçları Şekil 5.7'deki gibidir. Tüm sonuçlar aynı bilgisayar ve tamsayı diziler kullanılarak elde edilmiştir.

Çizelge 5.7 göstermektedir ki dizilerin boyutu küçük olunca (<10000) BST iyi, fakat dizilerin boyutu arttıkça BST'nin performansı azalmaktadır. Dizi boyutu büyüdükçe, RBT temelli sıralama işlemi BST ve SR'den daha iyi performans sergilemektedir. Bu durum Şekil 5.7'de görülmektedir.

Çizelge 5.7. Karışık verilerde ağaç veri yapısı temelli sıralama süreleri.

Düğüm Sayısı	1000	5000	10000	20000	30000	50000	100000	200000
BST sort	0	3.2	6.2	9.2	12.5	21.8	39.2	109.5
RBT sort	0	3.2	6.4	9.6	16.5	23.5	47.2	101.5
SR sort	0	2	5.2	12.4	15	24	56	146



Şekil 5.7. Karışık verilerde SR, RBT ve BST sıralama algoritmalarının sıralama sürelerinin karşılaştırması.

Çizelge 5.8. 1000-100000 elemanlı ters sıralı (A-Z) verilerde sıralama süreleri

Düğüm Sayısı	1000	5000	10000	20000	30000	50000	100000
BST sort	8.5	93.5	328	1287	2907	8065	74640
RBT sort	0	2.4	7.5	10.7	15.5	31	47
SR sort	0	2	3	9.2	12.4	19.6	31.5

Çizelge 5.8 ve Çizelge 5.9'daki sonuçlara dikkat edilirse tamamı veya çoğunluğu sıralı (A-Z) ya da ters sıralı (Z-A) olan bir veri setini sıralamak için ikili ağaç veri yapısı kullanılınca çok kötü bir performans sergilemektedir. Bunun sebebi

oluşan ikili ağacın dengesiz olması hatta doğrusal bir yapıya dönüşmesidir (Şekil 2.4.b). Sonuçların tamamına bakıldığında ise, yeni geliştirilen atlamalı halka (skip ring) veri yapısının oluşturulma, düğüm arama, ekleme, silme ve sıralama işlemlerinde BST ve RBT'den iyi olduğu görülmektedir.

Çizelge 5.9. 1000-100000 elemanlı ters sıralı (Z-A) verilerde sıralama süreleri

Düğüm Sayısı	1000	5000	10000	20000	30000	50000	100000
BST sort	9.5	93.5	336.	1318	2987.5	9154.8	75340.6
RBT sort	0	2.4	7.5	12.7	15.5	31.2	46.5
SR sort	0	2.1	3.2	6.2	12.6	18.6	34.3

Atlamalı halka sıralama (Algoritma 12) girdi parametreleri değiştirilerek bağlı listelerin sıralanması amacıyla da kullanılabilir.

5.2.2. Atlamalı halka sıralama algoritmasının değerlendirmesi

Atlamalı halka (skip ring) veri yapısının katmanlı haline benzer bir yapıda sıralama gerçekleştirdiğinden bu sıralama algoritmasına atlamalı halka sıralama (skip ring sort) ismi verilmiştir. Önerilen yeni sıralama algoritması, atlamalı halka veri yapısının seviyeli şekli göz önünde bulundurularak oluşturulmuştur. Veriler sıralanırken her bir eleman olması gereken sıraya katmanlı bir şekilde yerleştirilmektedir. Yeni bir eleman olması gereken sıraya O(lgN) zaman karmaşıklığında yerleştirilmektedir. N elemanlı bir veri kümesinin sıralanması ise, en fazla O(NlgN) zaman karmaşıklığında gerçekleşmektedir. P = 1/4 (0.25) alınırsa oluşana yapının yüksekliği h=(lgN)/2 olmaktadır dolayısıyla süre daha da azalmaktadır.

Önerilen yeni algoritma karışık veriler üzerinde bazı sıralama algoritmasından daha iyi bir performans sergilemektedir. Ayrıca sıralı ve tersten sıralı veri kümelerinde karışık (random) veri kümelerinden 1,5-2 kat daha hızlı çalışmaktadır.

Atlamalı halka sıralama (skip ring sort) algoritmasının performansı özellikle ters sıralı ve sıralı algoritmalarda daha iyidir. Ayrıca bu algoritma ile sıralanan veriler soldan sağa doğru sıralı olarak atlamalı halka veri yapısı üzerinde tutulmaktadır. Yani atlamalı halka veri yapısının tüm özelikleri arama, ekleme, silme işlemleri bu sıralı veriler üzerinde en fazla O(lgN) sürede gerçekleştirilmektedir. Diğer sıralama algoritmalarına göre bu çok büyük bir avantajdır.

Atlamalı halka sıralama (skip ring sort) algoritmasından iyi sonuçlar elde etmek için optimum seviye üretilmesi çok önemlidir. Bu da random_level() algoritmasındaki P eşik değerine bağlıdır. Atlamalı halka sıralama (skip ring sort) uygulamasında, random_level() algoritmasında P=1/4 alınmıştır. Bu algoritma (Algoritma 9) ile üretilen seviyelere düğümler sıralı olarak eklenmektedir. Eklenen bu düğümlerin seviyeleri çok yüksek (P eşik değeri 0.5, 0.75, 0.9 gibi) ya da çok düşük (P eşik değeri 0.1 gibi) ise, atlamalı halka sıralama algoritmasının performansı olumsuz etkilenmektedir. Bu uygulamada, P eşik değeri ~1/4 (0.25) alınarak gerçekleştirilmiş ve elde edilen sıralama sonuçları Çizelge 5.4, Çizelge 5.5, Çizelge 5.6, Çizelge 5.7, Çizelge 5.8, Çizelge 5.9 ve Şekil 5.5, Şekil 5.6 Şekil 5.7'de sunulmuştur.

Atlamalı halka temelli sıralama işlemi yapıyı inşa etme işlemi ile benzerdir. Çünkü N tane elemanı sıralamak N elemanlı bir atlamalı halka yapısını oluşturmak anlamına gelir. Atlamalı halka veri yapısı temelli sıralama işlemi için zaman karmaşıklığı T(N)=O(hN)=O(1/2NlgN)= O(Nlg N) olur. Çünkü N>1 için (Bazı K değerleri için $N=4^K$ olduğu varsayılırsa)

$T(N) = 4T(N/4)+N$ {Ring 1}

$= 4(4T(N/16)+N/4)+N$ {Genişlet}

$= 4^2T(N/16)+2N$ {Ring 2}

$= 4^2(4T(N/64)+N/16+2N$ {Genişlet}

$= 4^3T(N/4^3)+3N$ {Gözlemle} {Ring 3}

$= 4^KT(N/4^K)+KN$ {Ring K veya Ring i}

$= 4^{\log_4 N} T(N/N)+N\log_4 N$

$= N+ 1/2NlgN$

olur.

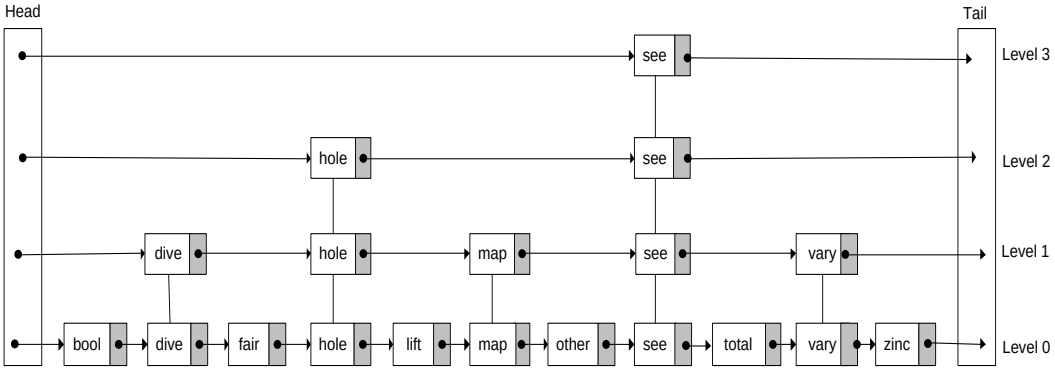
5.3. Örnek Uygulama 3: Atlamalı Halka (Skip Ring) Veri Yapısı Temelli Yeni Arama Algoritması: Piramit Arama (Pyramid Search)

Piramit arama (Pyramid search) algoritması, atlamalı halka veri yapısı ve ona ait algoritmalardan faydalanılarak geliştirilmiştir. Yeni arama algoritması atlamalı halka veri yapısının piramit şeklindeki yapısı göz önüne alınarak geliştildiği için bu

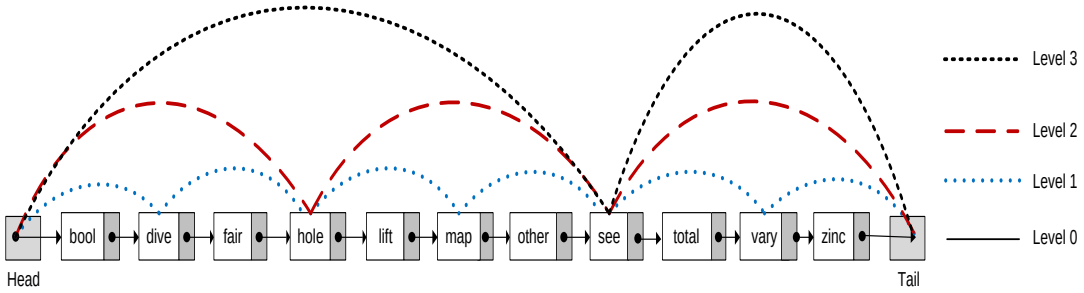
isim verilmiştir. Atlamalı halka veri yapısındaki standart algoritmalar yani arama, ekleme ve silme algoritmaları için N elemanlı bir atlamalı halka yapısı için zaman karmaşıklığı O(lgN)'dir. Önerilen yeni piramit arama (PS-Pyramid Search) algoritmasında arama frekansı esas alınarak veriler piramit şeklindeki yapıya yerleştirilmektedir. Böylece, N elemanlı bir veri kümesinde arama işlemi için zaman karmaşıklığı (time complexity) sık aranan veriler için Θ(1) olmaktadır. Bu çalışmada piramit arama (Pyramid search) algoritması ile doğrusal arama ve ikili arama algoritmaları karşılaştırılmıştır. Zaman karmaşıklık analizi sonuçları ve uygulama sonuçları çok aranan veriler için önerilen yeni piramit arama (PS) algoritmasının iyi olduğunu göstermektedir.

Bağlı listeler verilere (düğüm) rastgele erişime izin vermezler, bundan dolayı dizilerde olduğu gibi indeksi bilinen bir veriye ulaşılamaz. Yani çok basit işlem olan, bir elemana ulaşmak için listenin başından sonuna doğru taramak gerekmektedir [2 3]. Bağlı listelerde düğüm arama işlemi doğrusaldır. N elemanlı bir bağlı listede bir düğümü arama zaman karmaşıklığı (time complexity) O(N) olarak gerçekleşir.

Şekil 5.8'de {zinc, bool, fair, hole, dive, lift, map, total, vary, other, see} elemanlarından oluşan bir atlamalı liste görülmektedir. Bazı araştırmacılar tarafından, atlamalı liste (skip list), dengeli ağaçlara (balanced tree) alternatif olarak sunulmaktadır. Ağaç veri yapıları ile yapısal hiçbir benzerlik yoktur. Fakat Şekil 5.9'a dikkat edilirse, atlamalı listenin bağlı listelere alternatif bir veri yapısı olduğu açık bir şekilde görülecektir. Gerçekte tek bir bağlı liste vardır ve bu liste üzerinde bazı düğümlere atlama noktaları oluşturulmuştur.

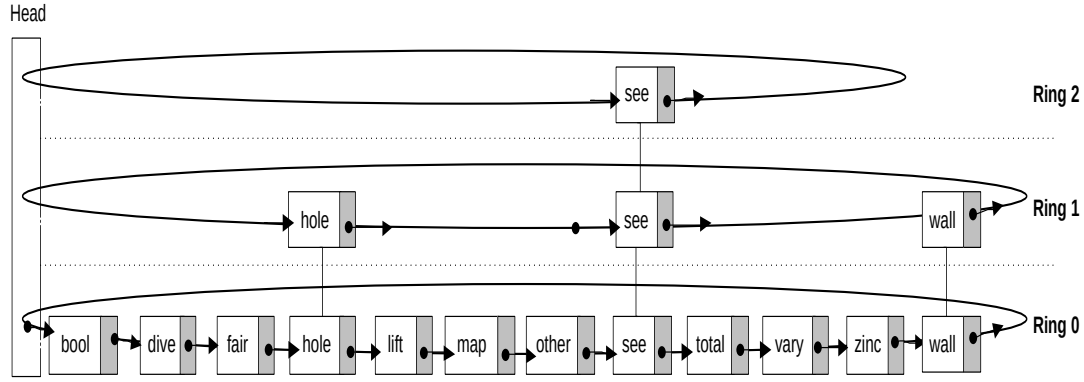


Şekil 5.8. Atlamalı liste (Bağlı liste temelli veri yapısı)



Şekil 5.9. Atlamalı listenin gerçek yapısı [21]

Atlamalı halka veri yapısındaki halkalar oluşturulurken Algoritma 9'da P=1/4 alınıp seviye oluşturulursa oluşan yapı ve üste doğru halkalar (Ring 0, Ring 1,.., Ring k) Şekil 5.10'daki gibi olur. N elemandan oluşan bir atlamalı halka veri yapısında, bu halkalardaki düğüm sayıları dağılımları farklılık göstermektedir. Şekil 5.10 - Ring 0 seviyesinde toplam N tane düğüm vardır. Ring 0 seviyesindeki her dört düğümden biri kendinden sonra gelen dört düğümden birine fazladan bir bağ içerirse, Ring 1 oluşur (Şekil 5.10 – Ring 1). Ring 1 seviyesinde en fazla $\lceil \frac{N}{4} \rceil + 1$ düğüm bulunur. Ring 1 seviyesindeki her dört düğümden biri kendinden sonra gelen dört düğümden birine fazladan bir bağ içerirse Ring 2 oluşur, bu şekilde devam edilerek yapı oluşturulur. Böylece N elemanlı bir atlamalı halka veri yapısının yüksekliği yaklaşık olarak h=(lg N)/2 olur.



Şekil 5.10. Atlamalı Halka **(**Skip ring) (P=1/4 için) [21]

5.3.1. Piramit arama (pyramid search) ve ikili arama (binary search)

Yeni arama algoritması atlamalı halka (skip ring) veri yapısı üzerinde geliştirilmiştir. Atlamalı halka veri yapısının piramit şeklindeki katmanlı yapısından faydalanılmıştır. Atlamalı halka veri yapısında standart arama işlemi (Algoritma 8)

yapının en üst seviyesinden başlar alt seviyelere doğru aranan eleman bulununcaya kadar devam eder. Geliştirilen yeni arama algoritmasında (Algoritma 13) ise, bir eleman için ne kadar çok arama yapılmışsa, o elaman atlamalı halka veri yapısının üst seviyelerine çıkarılır. Yani piramit şeklindeki yapının üst katmanlarına doğru en çok aranan elemanlar, alt katmanlara doğru ise, en az aranan elemanlar yerleşmektedir. Piramit şeklindeki yapı elemanların aranma sayısına (frekans) göre oluşturulmaktadır (Şekil 5.10 ve Şekil 5.11). Atlamalı halka veri yapısında arama (Algoritma 8) zaman karmaşıklığı O(lgN) iken piramit arama algoritmasında (Algoritma 13) bu işlem çok aranan elemanlar için Θ(1)'e yaklaşmaktadır. Piramit arama algoritmasında arama işlemi için zaman karmaşıklığı Θ(1)-O(lgN) arasında değişmektedir. Yani en çok aranan elemanlar piramidin tepesine yakın olduğu için arama zaman karmaşıklığı ~ Θ(1) olmaktadır. En az aranan elemanlar ise, en altta olduğu için zaman karmaşıklığı O(lgN)'dir.

Çizelge 5.10. Şekil 5.10'daki düğümlerin aranma frekanslarına göre seviyelere yerleştirilmesi.

Düğümler	**bool**	**dive**	**fair**	**hole**	**lift**	**map**
Frekans	0	0	0	1	0	0
Seviyeler	0	0	0	1	0	0
Düğümler	**other**	**see**	**total**	**vary**	**zinc**	**wall**
Frekans	0	2	0	0	0	1
Seviyeler	0	2	0	0	0	1

Çizelge 5.10'daki 'dive' 2 defa, 'map' 4 defa ve 'vary' 3 defa arandığında Çizelge 5.11'deki durum oluşur. Şekil 5.11'de ise Çizelge 5.11'deki değerlerden oluşan atlamalı halka yapısı görülmektedir. Şekil 5.11 oluşturulurken Piramit arama algoritması (Algoritma 13) kullanılmıştır. Şekil 5.11 elemanların aranma sayısı (frekans) temel alınarak oluşturulmuştur. Yani aranan bir eleman her arama işlemi sonunda bir üst seviyeye çıkarılmıştır. Böylece, çok aranan düğüm piramidin tepesinde en az aranan eleman ise en altında yer almaktadır.

Çizelge 5.11. Şekil 5.11'deki düğümlerin aranma frekanslarına göre seviyelere yerleştirilmesi

Düğümler	**bool**	**dive**	**fair**	**hole**	**lift**	**map**
Frekans	0	2	0	1	0	4
Seviyeler	0	2	0	1	0	4
Düğümler	**other**	**see**	**total**	**vary**	**zinc**	**wall**
Frekans	0	2	0	3	0	1
Seviyeler	0	2	0	3	0	1

Piramit arama algoritması için atlamalı halka veri yapısının geliştirilip kullanılmasının sebebi, bu veri yapısının seviyeli yapıda olmasıdır. Ayrıca bu veri yapısında N elemanlı yapıda bir elemanın standart arama (Algoritma 8) için zaman karmaşıklığı O(lg N)'dir. Atlamalı halka veri yapısı oluşturulurken her seviyedeki elemanlar küçükten büyüğe sıralanmaktadır (Şekil 4.4 Ring 0, Ring 1, Ring 2, Ring 3). Verilerin sıralı ve katmanlı yapıda olması atlamalı halka veri yapısının en önemli özelliğidir.

Doğrusal arama (lineer search) algoritmasında arama, veri kümesinin ilk elemanından başlanılarak son elemana kadar doğrusal bir şekilde devam eder. Eğer aranan eleman dizinin sonlarına yakınsa, arama çok yavaş gerçekleşir. Bu yüzden N elemanlı bir veri kümesinde zaman karmaşıklığı O(N) olur. Bu arama yönteminde arama yapılacak veri kümesinin sıralı ya da sırasız olması önemli değildir. Ancak sıralı veri kümeleri üzerinde verimli bir arama yöntemi değildir [46, 47, 48].

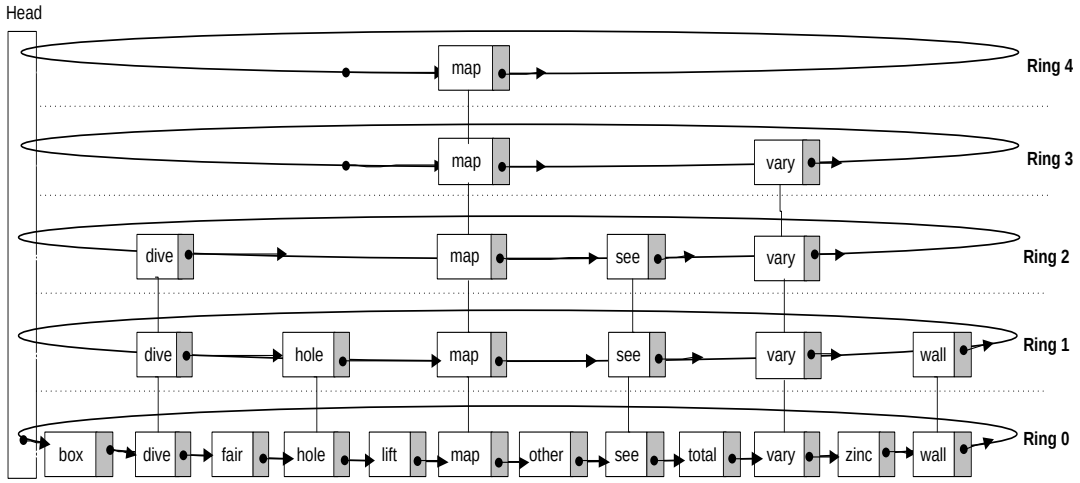
Diğer bir arama algoritması ikili arama (binary searching) algoritmasıdır. Bu algoritmanın bir veri kümesine uygulanabilmesi için veri kümesinin sıralı olması gerekmektedir. Eğer veri kümesi sıralı değilse önce sıralama algoritmalarından biri kullanılıp verilerin sıralanması gerekmektedir.

N elemanlı bir veri kümesinde ikili arama algoritması için zaman karmaşıklığı en fazla O(lgN) olur.

Ayrıca dengeli ağaçlarda da arama işlemi ikili arama (binary searching) algoritmasında olduğu gibi en fazla O(lgN) sürede gerçekleşmektedir.

Algoritma 13: Piramit Arama (Pyramid Search) Algoritması

```
1:  PyramidSearch(ring, search_value)
2:    temp←ring→head ,
3:    level←ring→level
4:    update[MaxLevel +1]
5:    while (level >=0)
6:    if  (temp →next[level] →value=search_value) then
7:     for i ← level downto 0 do
8:     while (temp →next[i] ≠ ring→head and temp →next[i]→value < search_value)
9:          temp ← temp →next[i]
10:      update[i] ← temp
11:    end for
12:     temp ← temp →next[0]
13:     int lvl ← level +1;
15:    if (lvl > ring→level)
16:      update[lvl] ←ring->head
17:      ring→level ← lvl
18:    end if
19:    temp→next[lvl] ← update[lvl] →next[lvl]
20:    update[lvl] →next[lvl] ← temp;
21:  return true
21: end if
22: if (temp→next[level] →value<search_value)
23:    temp ← temp→next[level]
24: if (temp→next[level] →value>search_value)
25:    level ← level -1
26: end while
27: return false;
```



Şekil 5.11. Piramit Arama (Pyramid search) (P=1/4 için; Şekil 5.10'da 'dive' iki defa, 'map' dört defa ve 'vary' üç defa aranıyor)

Piramit arama algoritması, arama motorlarında etkili bir şekilde kullanılabilir. Şöyle ki, aranan düğümler piramit şeklinde arama frekansına göre tepeden alta doğru yerleşirse arama işlemi hızlanacaktır. En çok aranan anahtarlar piramidin tepesine, daha az arananlar ise alt katmanlara yerleşecektir. Arama işlemi tepeden başladığı için arama motoru çok aranan anahtarlara daha kısa sürede cevap verecektir. Aramaları frekanslarına göre sınıflandırmak daha kolay olacaktır. Arama motorlarında milyonlarca anahtar mevcut olup en çok aranandan en az aranana doğru piramit şeklinde bir yapı oluşturularak aramanın gerçekleştirilmesi çok büyük avantajlar sağlayacaktır.

Hafızada sürekli çalışan bir sözlük uygulamasında bu arama algoritması kullanılarak en çok aranan kelimeler piramidin en tepesine doğru ve en az aranan kelimeler tabanına doğru yer alacağından çok hızlı arama sonuçları elde edilecektir.

5.3.2. Piramit arama (pyramid search) algoritması ve diğer arama algoritmalarının uygulamalı karşılaştırılması.

Piramit arama (Pyramid Search-PS), doğrusal arama (Linear Search-LS) ve ikili arama (Binary Search-BS) algoritmaları farklı boyutlarda sıralı diziler kullanılarak karşılaştırılmıştır. 1000-500000 elemanlı diziler üzerinde PS, LS ve BS arama algoritmalarının zaman karmaşıklıkları Çizelge 5.12, Çizelge 5.13 ve Çizelge 5.14'te görülmektedir. Ayrıca her veri kümesi üzerinde (örneğin 1000 elemanlı dizi) birden fazla (1000, 10000 gibi) arama gerçekleştirilip bunların ortalaması alınarak

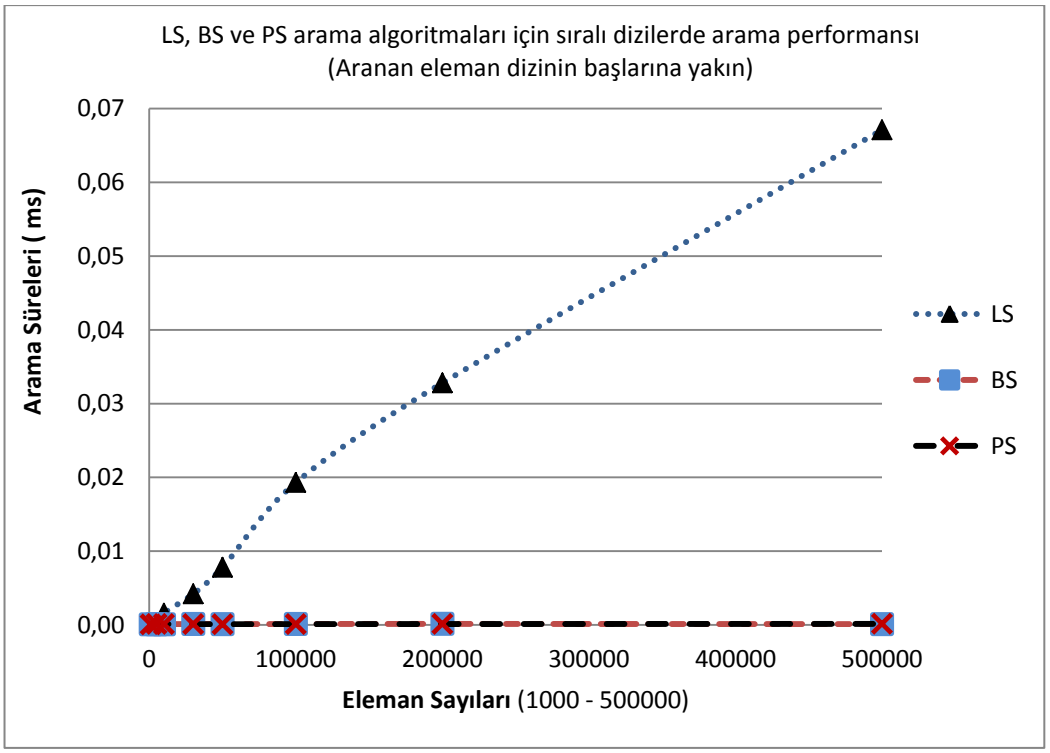
sonuçlar elde edilmiştir. Burada amaç daha gerçekçi sonuç elde etmektir. Eğer her dizide (1000, 5000, …) sadece bir defa arama yapılarak sonuç alınsa BS ve PS algoritmaları için gerçek sonuç elde edilemeyecektir. Çünkü PS algoritması için arama sadece tepedeki eleman için olursa Θ(1), aynı işlem en alttaki eleman için olursa, O(lgN) olur. Aynı şekilde BS algoritması için dizinin ikiye bölündüğü nokta aranan elemana denk gelirse, yine arama sonucu çok kısa sürecektir. Bunların olmaması için her dizide birden fazla (1000, 10000 gibi) arama yapılmış bunların ortalaması alınmıştır.

Çizelge 5.12, Çizelge 5.13 ve Çizelge 5.14'teki sonuçlar aynı bilgisayar üzerinde elde edilmiştir. Elde edilen sonuçlar mili saniye cinsindendir. Elde edilen sonuçlar şunu göstermektedir; eğer arama yapılacak dizinin boyutu küçükse LS normal bir performans sergilemektedir. Eğer dizinin boyutu büyüyorsa, PS ve BS algoritmalarının performansı artmakta, LS algoritmasının performansı düşmektedir. Şekil 5.12, Şekil 5.13 ve Şekil 5.14 bu durumu göstermektedir.

Çizelge 5.12. Sıralı verilerde LS, BS ve PS Arama algoritmaları için performans karşılaştırması (Aranan eleman dizinin baş tarafına yakın)

Eleman Sayıları	**1000**	**5000**	**10000**	**30000**	**50000**	**100000**	**200000**
LS	0,0000000	0,0000000	0,0015900	0,0042000	0,0078000	0,0193000	0,0328000
BS	0,0000780	0,0000940	0,0000940	0,0001250	0,0001090	0,0001250	0,0001410
PS	0,0000620	0,0001240	0,0001100	0,0001090	0,0000930	0,0001090	0,0001250

Çizelge 5.12, Çizelge 5.13, Çizelge 5.14 ve Şekil 5.12, Şekil 5.13, Şekil 5.14 incelendiğinde arama zamanı yönünden PS ve BS algoritmalarının LS algoritmasından çok daha iyi olduğu görülmektedir. PS ve BS için sıralı dizilerde arama zaman karmaşıklığı O(lgN)'dir. Ayrıca uygulama sonuçlarına bakıldığında PS algoritmasının çok aranan veriler için BS arama algoritmasından daha iyi olduğu görülmektedir.



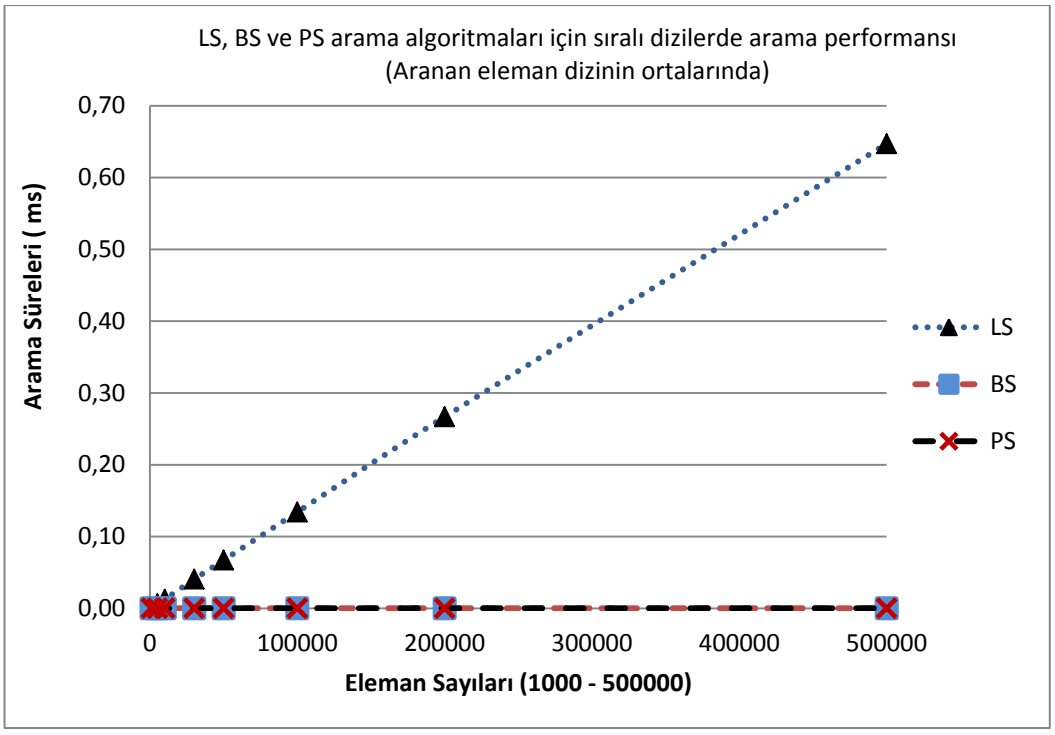
Şekil 5.12. Sıralı verilerde LS, BS ve PS Arama algoritmaları için Performans (Aranan eleman dizinin baş tarafına yakın)

Çizelge 5.13. Sıralı verilerde LS, BS ve PS Arama algoritmaları için Performans karşılaştırması (Aranan eleman dizinin ortalarında)

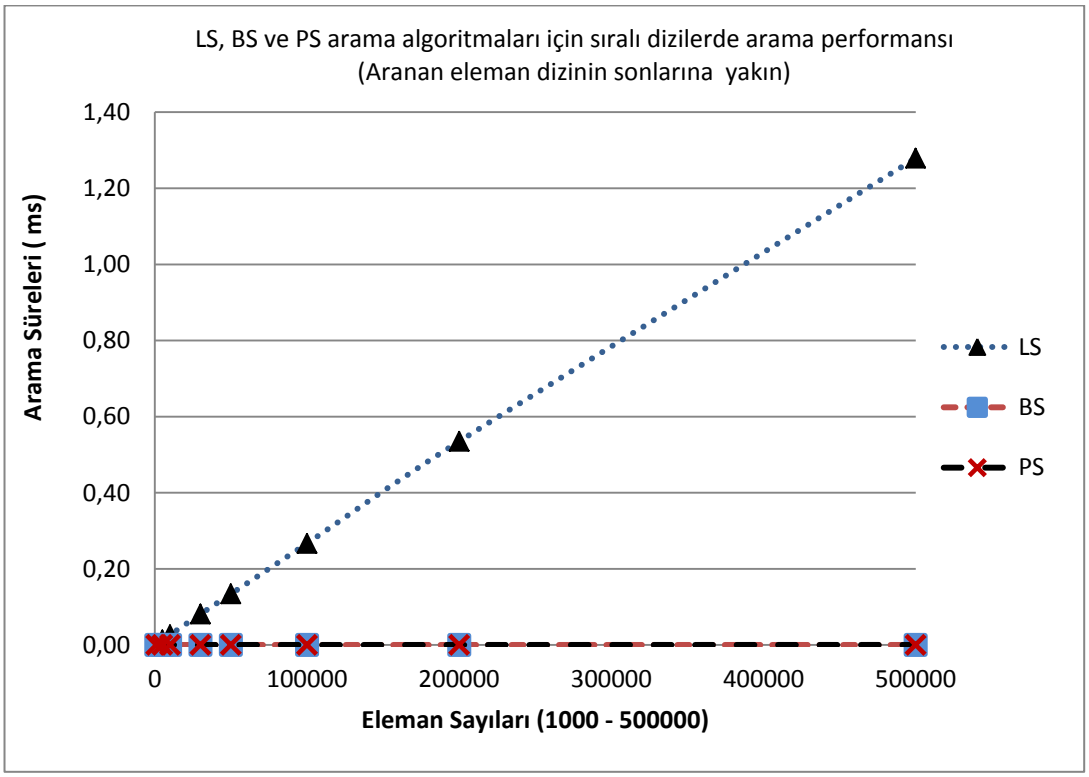
Eleman Sayıları	**1000**	**5000**	**10000**	**30000**	**50000**	**100000**	**200000**
LS	0,001590	0,006200	0,012500	0,040600	0,067100	0,134100	0,266700
BS	0,000078	0,000109	0,000125	0,000110	0,000109	0,000125	0,000125
PS	0,000094	0,000124	0,000125	0,000078	0,0001090	0,000109	0,000124

Çizelge 5.14. Sıralı verilerde LS, BS ve PS Arama algoritmaları için performans karşılaştırması (Aranan eleman dizinin sonlarına yakın)

Eleman Sayıları	**1000**	**5000**	**10000**	**30000**	**50000**	**100000**	**200000**
LS	0,003100	0,013200	0,026500	0,081100	0,134100	0,266800	0,535000
BS	0,000078	0,000094	0,000093	0,000109	0,000125	0,000109	0,000141
PS	0,000094	0,000142	0,000125	0,000094	0,000140	0,000125	0,000142



Şekil 5.13. Sıralı verilerde LS, BS ve PS Arama algoritmaları için Performans (Aranan eleman dizinin ortalarında)



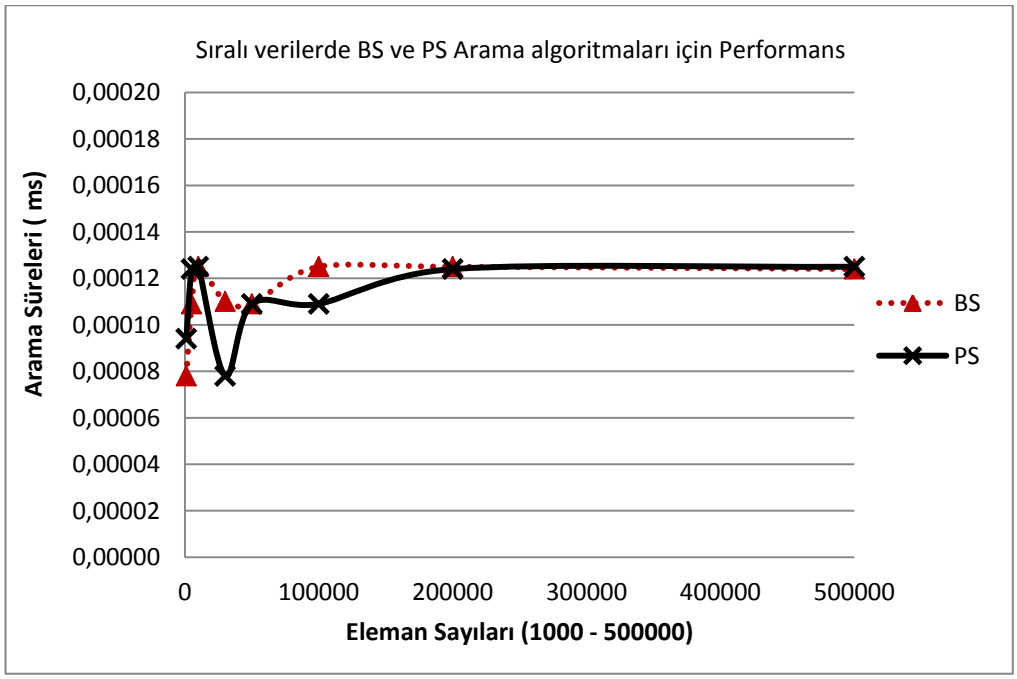
Şekil 5.14. Sıralı verilerde LS, BS ve PS Arama algoritmaları için performans (Aranan eleman dizinin sonlarına yakın)

Çizelge 5.15. Sıralı verilerde BS ve PS Arama algoritmaları için Performans.

Eleman Sayıları	1000	5000	10000	30000	50000	100000	200000
BS	0,000078	0,000109	0,000125	0,000110	0,000109	0,000125	0,000125
PS	0,000094	0,000124	0,000125	0,000078	0,000109	0,000109	0,000124

Çizelge 5.12, Çizelge 5.13, Çizelge 5.14'e dikkat edilirse PS arama algoritmasının performansı LS ve BS arama algoritmalarının performansından iyi olduğu görülecektir. Ayrıca Çizelge 5.15'te ise PS arama algoritmasının sık aranan verilerde BS arama algoritmasından daha verimli olduğu görülmektedir.

Arama yapılacak diziler sıralı değilse, doğrusal arama (linear searching) algoritmasının performansı diğerlerinden daha iyi olacaktır. Çünkü diğer arama algoritmalarında dizinin sıralı olması gerekmektedir. Sıralama işlemi maliyeti ise, N elemanlı bir dizide en fazla O(NlgN) olacaktır. Bu maliyet arama üzerine eklendiğinde PS ve BS için zaman karmaşıklığı O(NlgN) olacaktır. Halbuki doğrusal arama (lineer searching) için zaman karmaşıklığı en kötü durumda O(N)'dir.



Şekil 5.15. Sıralı verilerde BS ve PS Arama algoritmaları için Performans.

5.3.3. Piramit arama algoritmasının değerlendirmesi

Piramit arama algoritmasında arama işlemi için zaman karmaşıklığı Θ(1)-O(lgN) arasında değişmekte olup en çok aranan elemanlar için Θ(1), en az aranan elemanlar için O(lg N) olmaktadır.

Piramit arama algoritması, arama işlemlerinde etkin bir şekilde kullanılabilir. Piramit arama algoritması arama motorlarında en çok aranandan en aza doğru aranan verileri sınıflandırmada, veri tabanı uygulamalarında örneğin bir sözlük (İngilizce-Türkçe gibi) uygulamasında en çok aranan kelimelere daha hızlı erişmede etkin bir şekilde kullanılabilir. Piramit arama algoritması büyük veri kümeleri üzerindeki arama işlemlerinde önemli avantajlar sağlayacaktır.

5.4. Örnek Çalışma 4: Fair Priority Scheduler (FPS): Atlamalı Halka Veri Yapısı Temelli Yeni Görev Zamanlayıcı Algoritması

Görev zamanlama (Process scheduling) işletim sistemlerinin en önemli konularından biridir. Linux işletim sistemi 6.23 sürümüyle beraber görev zamanlayıcı olarak CFS (Completely Fair Scheduler) algoritmasını gerçeklerken kırmızı-siyah ağaç veri yapısını kullanmaktadır. Oysa görev zamanlayıcı işlemleri için kırmızı-siyah ağaçlar yerine atlamalı halka veri yapısı etkin bir şekilde kullanılabilir. Ayrıca bu çalışmada atlamalı halka veri yapısını kullanan yeni bir görev zamanlayıcı algoritması (Fair Priority Scheduler (FPS)) geliştirilmiş ve adımları anlatılmıştır.

Her işletim sisteminin kendine göre farklı dönemlerde kullandığı farklı görev zamanlayıcı (process scheduler) algoritmaları vardır. İşletim sistemleri görev zamanlayıcı algoritmasını gerçekleştirebilmek için farklı veri yapıları (bağlı liste, kırmızı-siyah ağaç gibi) kullanmaktadır. FPS algoritmasını gerçekleştirebilmek için geliştirilen atlamalı halka veri yapısı kullanılmıştır.

5.4.1. Linux'ta görev zamanlayıcı algoritmaları

Görev zamanlama işlemi, bir işletim sisteminin en önemli parçasıdır. Linux® gelişmeye ve bu alandaki yeniliklerine devam etmektedir. Her işletim

sisteminde olduğu gibi Linux işletim sisteminin de ilk sürümlerinde daha basit görev zamanlayıcılar kullanılmıştır [39]. Zamanla görev zamanlayıcılar geliştirilmiştir.

Linux 2.6 sürümüyle beraber O(1) (Şekil 2.6) olarak adlandırılan bir görev zamanlayıcı daha önceki görev zamanlayıcıların birçok problemini çözmek için tasarlanmıştır.

O(1) görev zamanlayıcı, öncelik temelli zamanlama politikasını kullanır. Bu görev zamanlayıcı, görev öncelik kuyruğundan en uygun görevi çalışmak için belirler. O(1) zamanlayıcı algoritması çoklu kuyrukları kullanır (Şekil 2.6) . O(1) zamanlayıcı algoritmasının temel yapı taşı çalışma kuyruklarıdır. Bu kuyruklar iki gruba ayrılır;

- Aktif çalışma kuyrukları (active runqueue) : Çalışmayı bekleyen kuyruklar
- Çalışma süresi bitmiş (expired runqueue) : Kendine verilmiş olan çalışma süresini bitiren fakat sonlanmayan yani tamamen bitmeyen görevlerin tutulduğu kuyruklar

Çekirdek bu iki çalışma kuyruklarına işaretçilerle (pointer) ulaşır. Bu iki çalışma kuyrukları basit bir işaretçi (pointer) ile karşılıklı yer değiştirilir. Yani tüm aktif kuyruklar bittikten sonra pasif olan kuyruk aktif yapılır.

Aktif kuyruklar içinde görevlerin önceliklerine göre oluşturulmuş 140 tane görev öncelik seviyesi vardır. Aynı önceliğe sahip görevler aynı öncelik kuyruğunda tutulurlar. Örneğin, Şekil 2.6'da Priority 2 öncelik kuyruğunda aynı önceliğe sahip P4, P10, P21 ve P32 görevleri vardır. Bu her bir kuyruk ise FIFO (First In First Out) algoritmasına göre, yani ilk gelen ilk işletilecek demektir [34, 60].

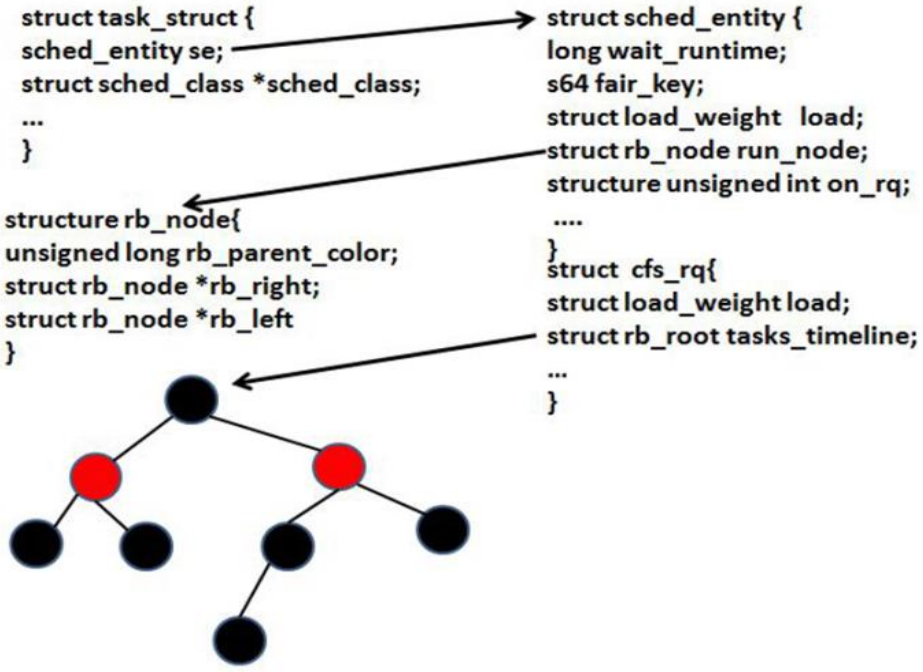
Diğer bir görev zamanlayıcı (process scheduler) algoritması ise, CFS (Completely Fair Scheduler)'dir. CFS algoritması ise görevlere adil bir çalışma süresi vaat eder. CFS görevlere işlem sürelerine göre adil bir pay verir. CFS algoritmasında bu belirlenen çalışma süresi, sanal çalışma süresi (virtual runtime) olarak adlandırılır. Yani her görev için O(1) zamanlayıcı gibi sabit bir çalışma süresi değil de adiliyet esasına dayalı her görev için farklı bir süre belirlenir.

CFS görev zamanlayıcı algoritmasında, bir göreve verilen çalışma süresi, sanal çalışma süresi olarak isimlendirilir. O(1) zamanlayıcı algoritmasında, bir görev yüksek öncelikli ise, en kısa zamanda işletilecek demektir. CFS'de ise, daha küçük sanal çalışma süresine sahip olan görev en kısa zamanda işletilecek demektir. Görevleri yönetmek için O(1) temel olarak aktif ve pasif (expired) olmak üzere iki öncelik kuyruğu kullanır, CFS ise görevleri yönetmek için kırmızı-siyah ağaç veri yapısını kullanır. Kırmızı-siyah ağaç kullanılmasının asıl sebebi kendi kendini dengeleyen (self balancing) bir yapıda olmasıdır. İkinci olarak ise, N düğümden oluşan bir kırmızı-siyah ağaçta, düğüm ekleme veya silme işlemi O(log N) sürede gerçekleşir. Örnek Uygulama 1'de, kırmızı-siyah ağaçlarla yeni geliştirdiğimiz atlamalı halka (skip ring) veri yapısı kıyaslanmıştır.

Şekil 2.7'deki sanal çalışma süresi ağırlıklandırılmış zaman dilimi (weighted timeslice) olarak düşünülebilir. Sistemdeki görevler sanal çalışma süresi değerine göre kırmızı-siyah ağaca yerleştirilir. Bir görevin sanal çalışma süresi değeri ne kadar küçükse, işlemciye ihtiyacı o kadar fazladır. Sanal çalışma süresi değeri küçük olan en solda ve en önce işletilecek görevi ifade eder. Yani görevler en soldan başlanılarak işletilirler biten görevler silinir yeni gelen görevler kırmızı-siyah ağaca eklenir.

Çizelge 5.1, Çizelge 5.2, Çizelge 5.3 ve Şekil 5.1, Şekil 5.2, Şekil 5.3 incelendiğinde atlamalı halka veri yapısının performansı kırmızı-siyah ağaç veri yapısından daha iyi olduğundan CFS algoritmasında atlamalı halka veri yapısı kullanılabilir. Atlamalı halka veri yapısının level 0 seviyesi en soldan itibaren işletilecek olursa, CFS algoritması gerçeklenmiş olur. Böylece CFS algoritmasının performansı artar.

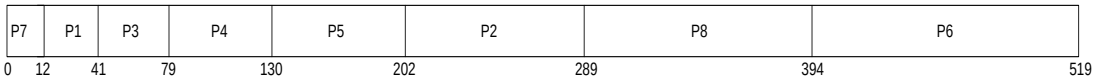
Şekil 5.16'da kırmızı-siyah ağaç veri yapısına ait tanımlamalar görülmektedir. Linux işletim sistemindeki tüm görevler task_struct olarak ifade edilen bir yapıyla sunulur. Bu sunumda en üste task_struct konumlanır, diğer tanımlamalara buradan erişilir. Bu yapıdaki rb ifadeleri kırmızı-siyah ağaç tanımlamalarını göstermektedir. Buradaki tanımlamalar, atlamalı halka veri yapısını tanımlayacak şekilde değiştirilebilir. Diğer tanımlamalar da değiştirilerek, CFS algoritması atlamalı halka veri yapısı ile gerçekleştirilmiş olur.



Şekil 5.16. Görevler için kırmızı-siyah ağaç ve hiyerarşi yapısı (CFS) [34]

Atlamalı halka (skip ring) veri yapısı diğer bazı görev zamanlayıcı (process scheduler) algoritmalar için de kullanılabilir. Eğer zamanlayıcı algoritması dairesel bir döngüye ihtiyaç duymuyorsa atlamalı halka (skip ring) veri yapısı yerine atlamalı liste (skip list) veri yapısı da kullanılabilir.

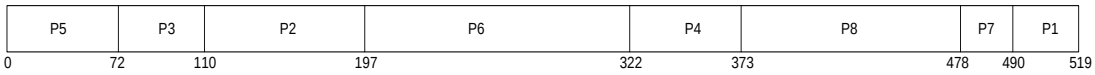
İlk olarak SJF (Shortest Job First -En kısa görev ilk) görev zamanlayıcı algoritması için atlamalı halka veri yapısı kullanılabilir. Şekil 5.17'da priority 0 seviyesine bakılırsa görevler en kısa süreliden en uzun süreliye sıralıdır. Priority 0 düzeyindeki en soldan başlanıp devam edilirse SJF (shortest-job-first) algoritması gerçeklenmiş olur. Atlamalı halka veri yapısı kullanıldığı için görev ekleme ve silme zaman karmaşıklığı O(lg N) olur.



Şekil 5.17. Atlamalı halka veri yapısının SJF (Shortest Job First) zamanalayıcı algoritmasında kullanımı (Şekil 5.20 – Priority 0 level)

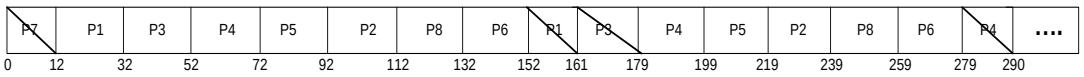
İkinci olarak, öncelik temelli zamanlama (priority based scheduling) algoritması için atlamalı halka veri yapısı kullanıldığında Şekil 5.20'da ilk olarak P5 görevi bitene kadar işlem görür. Daha sonra P3 ve ondan sonra P2 ve P6 görevleri

bitene kadar devam ederler. Bu şekilde önceliklere göre devam eder. Biten görevler silinir yeni gelenler yapıya eklenir.



Şekil 5.18. Atlamalı halka yapısının öncelik temelli zamanlayıcı (priority based scheduling) algoritmasında kullanımı.

Üçüncü olarak, atlamalı halka veri yapısı RR (Round-Robin) görev zamanlama algoritmasında kullanılabilir. Şöyle ki, Şekil 5.20'de Pritority 0 seviyesi alınarak, belirlenen quantum süresi kadar görevler soldan sağa doğru işletilirse RR (Round-Robin) algoritması gerçekleştirilmiş olur (Şekil 5.19). Yani sırası ile Şekil 5.20'de Priority 0 seviyesindeki P7, P1, P3, P4, P5, P2, P8, P6 görevleri quantum miktarı (Örneğin 20 ms) kadar dairesel olarak işletilir. Biten görevler (P7, P1, P3, P4, etc.) atlamalı halka yapısından silinir ve gelenler eklenir. İşlem bu şekilde devam eder. Yeni bir görev ekleme ya da silme işlemi atlamalı halka veri yapısında O(lgN) sürede gerçekleşeceğin büyük bir avantaj sağlar.



Şekil 5.19. Atlamalı halka veri yapısının RR (Round-Robin) zamanlama algoritmasında kullanımı (Şekil 5.20 – Priority 0 seviyesi)

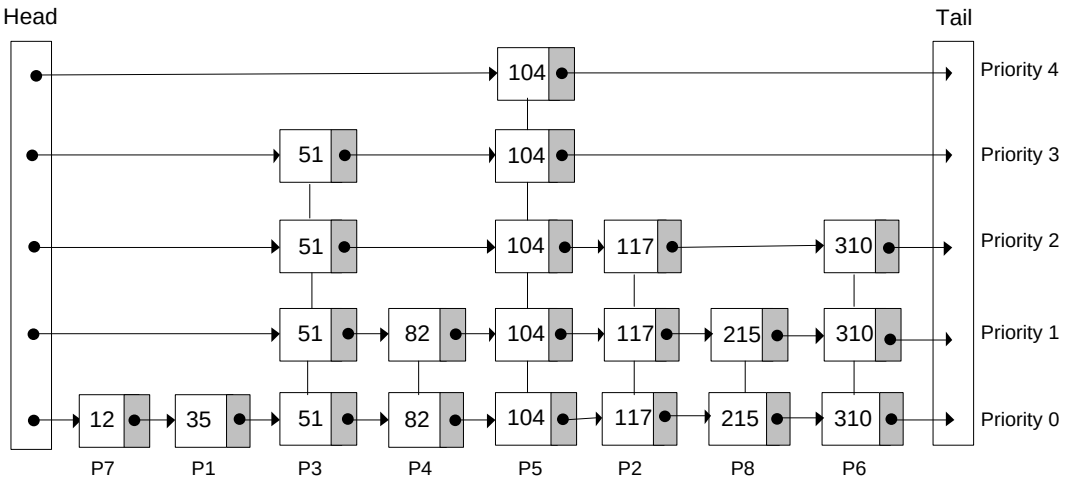
5.4.2. Yeni görev zamanlayıcı (process scheduler) algoritması

2. Bölüm 'de ifade edildiği gibi Linuxta yaygın kullanılan O(1) görev zamanlayıcı (sucheduler) algoritması tamamen öncelik kuyruklarına dayanan listelerle gerçekleştirilmektedir. CFS algoritması ise ağırlıklara göre hesaplanan sanal çalışma süresi değerine göre çalışmaktadır. Her iki yönteminde kendine göre avantaj ve dezavantajları mevcuttur. Önerilen yeni görev zamanlama algoritması FPS (Fair Priority Scheduler) her iki yöntemin birleşimi olarak çalışmaktadır. Yani hem öncelik hem de sanal çalışma süresi değerine göre görevlerin çalışmasını sağlamaktadır. Bu algoritmayı gerçeklemek için 4. Bölüm'de anlatılan atlamalı halka veri yapısı kullanılmıştır.

Çizelge 5.16. Görevler ve Gerekli Süre (Processes and required time)

Görevler	Öncelik (Priority)	İşletim Süresi
P1	0	35
P2	2	117
P3	3	51
P4	1	82
P5	4	104
P6	2	310
P7	0	12
P8	1	215

Çizelge 5.16'teki görevlerin işletim süresi değerleri kullanılarak Şekil 5.20 oluşturulmuştur. Bu tablodaki görevler öncelik (priority) değerlerine göre atlamalı halka veri yapısının insertNode() algoritması kullanılarak seviyelere sıralı olarak yerleştirilir.

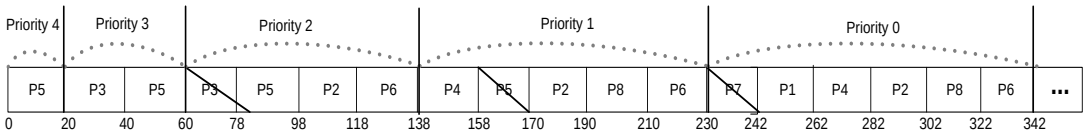


Şekil 5.20. Atlamalı halka (liste) temelli yeni görev zamanlayıcı için görevlerin yerleşimi (FPS-Fair Priority Scheduler)

FPS algoritması şöyle çalışır (Şekil 5.20) : ilk olarak en üst seviyedeki (priority 4) görevler (P5) soldan sağa doğru belirlenmiş bir zaman dilimi (timeslice vaya quantum) kadar işletilir sonra alt seviyeye inilir (piriority 3). Bu seviyedeki görevler (P3 ve P5) işletilir ve bir alt seviyeye inilir (piriority 2). Bu seviyedeki görevler (P3, P5, P2, P6) işletilir ve bir alt seviyeye inilir (piriority 1). Bu seviyedeki görevler (P3, P4, P5, P2, P8, P6) belirlenen zaman dilimi (timeslice) kadar işletilir ve bir alt

seviyeye inilir. Bu seviyedeki görevler (P7, P1, P3, P4, P5, P2, P8, P6) belirlenen zaman dilimi kadar işletilir. Yani Şekil 5.20'deki tüm öncelik (priority) seviyelerindeki görevler işletildiğinde P5 görevi 5 defa işletilecek, P3 görevi 4 defa işletilecek, P2 ve P6 görevleri 3 defa işletilecek, P4 ve P8 görevleri 2 defa işletilecek, P7 ve P1 görevleri ise 1 defa işletilecek. İşlem bu şekilde devam ederek tekrar başa dönecek. Bu arada P5, P3 ve P7 görevleri bittiği için silinecek işlem geriye kalan P1, P4, P2, P8 ve P6 görevleri ile devam edecektir. Bu işlemler devam ederken biten görevler silinecek ve yeni görevler atlamalı halka yapısına eklenecektir.

Bu algoritma ilk olarak önceliği esas alıp daha sonra her seviyede en soldaki görevden başlayıp işlem yapmaktadır (Şekil 5.21). Yani hem piriority hem de sanal çalışma süresi değerine göre görevleri işletmektedir. Şekil 5.20'de priority 2 düzeyini bakacak olursak önce P3 (en solda) daha sonra P5 ve P2, P6 şeklinde devam edecek. Böylece tamamen önceliği esas alan algoritmalar ya da toplam işletim süresini esas alan algoritmalara göre daha kullanışlı bir algoritma olacaktır.



Şekil 5.21. Atlamalı halka veri yapısının FPS (Fair Priority Scheduler) algoritmasına göre görevleri yürütmesi (Şekil 5.20 için)

Görev önceliğine göre çalışan diğer algoritmalar genellikle öncelik sırası gelen görev tamamen bitmeden diğer görevlere sıra gelmez. Yani Şekil 5.20'deki P5 görevi tamamen bitmeden P3'e sıra gelmez. P3 bitmeden de diğerlerine sıra gelmez. Bu durum sıradaki görevlerin çok uzun süre beklemesine sebep olur.

5.4.3. Yeni görev zamanlayıcı (process scheduler) algoritmasının değerlendirilmesi

Atlamalı halka veri yapısı yeni görev zamanlayıcı algoritması olan FPS (Fair Priority Scheduler)'yi gerçekleştirmek için geliştirilmiştir.

Atlamalı halka (skip ring) veri yapısı bellek yönetimi, görevlerin yönetimi için kullanılabilir. Linux işletim sisteminde kırmızı-siyah ağaçların yerine görev zamanlayıcı algoritmasında kullanılablir.

Önerilen yeni görev zamanlayıcı algoritması (FPS) öncelik uygularken daha adil bir yöntem kullanmaktadır. Yani hem önceliği hem de görev işletim süresini göz önünde bulundurarak işletmektedir. Böylece tüm görevlere önceliği de göz önünde bulundurarak daha kısa sürede sıra gelmektedir.

Atlamalı liste (Skip list) veri yapısı yerine, atlamalı halka veri yapısının kullanılmasının sebebi görev zamanlayıcı (scheduling) algoritmalarının büyük bir çoğunluğu dairesel (circular) yapıdadır. Yani görev listesi tekrar başa dönmektedir bundan dolayı önerilen atlamalı halka veri yapısı kullanılmıştır. Dairesel olmayan durumlarda ise, atlamalı liste veri yapısı P=1/4 (0.25) alınarak kullanılabilir.

6. SONUÇLAR VE ÖNERİLER

Bu tezde, bazı veri yapıları ve algoritmalar incelenmiş, bu veri yapıları ve algoritmaların nasıl geliştirildiği ve nerelerde kullanıldığı araştırılmıştır. İncelenen liste temelli veri yapılarından dairesel bağlı liste (circular linked list) ve atlamalı liste (skip list) veri yapısından esinlenilerek yeni bir veri yapısı geliştirilmiştir. Atlamalı listelerdeki bazı sorunlar ele alınıp çözülmüş ve iyileştirmeler yapılmıştır. Atlamalı listelerdeki iyileştirme sonucu elde edilen değerler dikkate değerdir.

Atlamalı halka (skip ring) veri yapısı geliştirilirken atlamalı listelerdeki iyileştirmeler yeni veri yapısına uygulanarak daha etkin hale getirilmeye çalışılmıştır. Atlamalı liste veri yapısında, bir düğümü yapıya eklerken eklenecek düğümün hangi seviyede (level) olacağı önemlidir. Her düğüm aynı seviyede olmamalı ve düğümlerin seviyelere dağılımı dengeli olmalıdır. Bunun için dengeli seviye üreten bir algoritmaya ihtiyaç vardır. Atlamalı liste için iki şekilde seviye üretilmesi mümkündür.

- Olasılıksal olarak seviye (level) üretme
- Düğüm sayısından faydalanarak seviye (level) üretme

Atlamalı listenin (skip list) analizi ve iyileştirilmesine yönelik yapılan çalışmalar sonucunda düğüm sayısına bağlı seviye oluşturma ile olasılıksal seviye oluşturma algoritmasının P=1/4 alındığında yakın sonuçlar ürettiği görülmüştür. Pugh kendi çalışmalarında P=1/2 almıştır. Önerilen yeni veri yapısında bu değer P=1/4 olarak alınmıştır. Önerilen algoritmalar, P=1/4 alındığında elde edilen sonuçlar daha iyi olmaktadır. Ayrıca, P=1/4 alındığında atlamalı halka (skip ring) veri yapısının yüksekliği azalmakta buna bağlı olarak arama, ekleme, silme performansı artmaktadır.

Önerilen atlamalı halka (skip ring) veri yapısı, dairesel bağlı liste veri yapısındaki arama, ekleme, silme işlemlerini koni şeklindeki katmanlı yapısından dolayı O(N) zaman karmaşıklığından, O(lg N) zaman karmaşıklığına indirmiştir. Önerilen yeni veri yapısının zaman karmaşıklık analizi yapılmıştır. Atlamalı halka (skip ring) veri yapısında en üst seviyedeki (Ring i) düğümlere erişim için zaman karmaşıklığı Θ(1)'dir. Atlamalı halka veri yapısı oluşturulurken P=1/4 alınırsa (yani bir halkadaki her dört düğümden biri üstteki halkaya çıkarılırsa); Arama, ekleme,

silme işlemleri için zaman karmaşıklığı T(N)=O(h)=O(1/2lgN)=O(lg N) olur. Elde edilen bu sonuçlar logaritmik olması açısından önemlidir. Bundan dolayı dairesel bağlı liste veri yapısının kullanıldığı yerlerde, atlamalı halka veri yapısı daha etkin bir şekilde kullanılabilecektir.

Dairesel bağlı listeler, birçok dairesel (çevrimsel) işlem gerektiren yerde kullanılmaktadır. Aynı şekilde dairesel işlem gerektiren her yerde daha etkin bir şekilde (O(lg N)) geliştirilen atlamalı halka veri yapısı kullanılabilir. Mesela bellek yönetiminde, görevlerin yönetiminde v.b birçok yerde atlamalı halka (skip ring) veri yapısı kullanılabilir.

Bu tez çalışmasında, geliştirilen yeni veri yapısının arama, ekleme, silme işlemlerini nasıl gerçekleştireceği belirlenmiştir. Bu kapsamda atlamalı halka (skip ring) veri yapısına ait tanımlamalar, algoritmalar oluşturulmuştur. Bu veri yapısının algoritmaları koda dönüştürülerek bazı uygulamalar ve atlamalı halka temelli bazı yeni algoritmalar geliştirilmiştir.

Tezde ilk olarak, ağaç veri yapıları ele alınıp bu veri yapılarının performansını olumsuz etkileyen döndürme, güncelleme özelikle kırmızı-siyah ağaçlarda düğümleri renklendirme ve renk değişikliği gibi işlemleri üzerinde durulmuştur. Daha sonra ise bir uygulama geliştirilip bu uygulama ile ikili arama ağaçları (binary search tree), kırmızı-siyah ağaçlar ve atlamalı halka veri yapıları karşılaştırılmıştır. Uygulama sonucuna bakıldığında, ağaç veri yapılarında dengeleme (balance) işleminin önemi görülmektedir. Özellikle dengesiz ağaçlarda, sıralı veya ters sıralı verilerden ağaç oluşturulurken zaman karmaşıklığı $O(N^2)$ olmaktadır. Yani, yapı bağlı liste (lidked list) haline dönmektedir. Ayrıca bu uygulama sonuçları incelendiğinde genel olarak atlamalı halka (skip ring) veri yapısının performansının ikili ağaçlar ve kırmızı-siyah ağaçlardan iyi olduğu görülmektedir. Bunun sebebi ağaçlarda denge sağlamak için bol miktarda döndürme, güncelleme yapılmasıdır. Ayrıca kırmızı-siyah ağaçlarda yeniden renklendirme işlemi de yapılmaktadır.

Önerilen sıralama algoritması, atlamalı halka (skip ring) veri yapısının katmanlı haline benzer bir yapıda sıralama gerçekleştirdiğinden bu sıralama algoritmasına atlamalı halka sıralama (skip ring sort) ismi verilmiştir. Önerilen yeni sıralama algoritması, atlamalı halka veri yapısının seviyeli şekli göz önünde

bulundurularak oluşturulmuştur. Veriler sıralanırken her bir eleman olması gereken sıraya katmanlı bir şeklinde yerleştirilmektedir. Yani, bir eleman olması gereken sıraya O(lgN) zaman karmaşıklığında yerleştirilmektedir. N elemanlı bir veri kümesinin sıralanması ise en fazla O(NlgN) zaman karmaşıklığında gerçekleşmektedir. P = 1/4 (0.25) alınırsa oluşan yapının yüksekliği h=(lgN)/2 olmaktadır dolayısıyla verilerin sıralanma süresi daha da azalmaktadır.

Atlamalı liste (skip list) ve atlamalı halka (skip ring) veri yapıları kullanılarak geliştirilen sıralama algoritması ile 1000-500000 elemanlı diziler üzerinde sıralama yapılmıştır. Elde edilen sonuçlar bazı $O(N^2)$ grubu ve O(NlgN) grubu sıralama algoritmalarının sonuçları ile kıyaslanmıştır. Önerilen sıralama algoritması karışık veriler üzerinde birçok sıralama algoritmasından daha iyi bir performans sergilemektedir. Sonuçlar incelendiğinde bu veri yapısının sıralama işleminde kullanıldığında güzel sonuçlar elde edileceği görülmüştür.

Atlamalı halka sıralama (skip ring sort) algoritmasının performansı, özellikle ters sıralı ve sıralı veri kümelerinde diğerlerine göre çok daha iyidir. Ayrıca sıralı ve ters sıralı veri kümelerinde karışık (random) veri kümelerinden 1,5-2 kat daha hızlı çalışmaktadır. Önerilen algoritma ile sıralanan veriler soldan sağa doğru sıralı olarak atlamalı halka veri yapısı üzerinde tutulmaktadır. Yani atlamalı halka veri yapısının arama, ekleme, silme işlemleri sıralı veriler üzerinde en fazla O(lgN) sürede gerçekleştirilmektedir. Diğer sıralama algoritmalarına göre bu çok büyük bir avantajdır.

Atlamalı halka sıralama algoritmasından iyi sonuçlar elde etmek için optimum seviye üretilmesi çok önemlidir. Sıralama işleminde, seviye (level) oluşturmak için kullanılan *random_level()* algoritmasında P=1/4 (0.25) alınarak uygulama gerçekleştirilmiştir. Olasılıksal olarak, P eşik değerlerine bağlı seviyeler üretilip üretilen bu seviyelere düğümler sıralı olarak eklenmektedir. Bu eklenen düğümlerin seviyeleri çok yüksek ($P \geq 0.5$) ya da çok düşük ($P \leq 0.1$) ise atlamalı halka sıralama algoritmasının performansı olumsuz etkilenmektedir.

Atlamalı halka veri yapısı üzerinde çalışan piramit şeklinde yeni bir arama algoritmasının esasları belirlenmiştir. Algoritma oluşturulmuş ve koda dönüştürülerek uygulama gerçekleştirilmiştir. Elde edilen sonuçlara göre önerilen

yeni arama yöntemi, sık aranan verilerde sıralı arama ve ikili arama yöntemlerinden daha iyi çalışmaktadır. Piramit arama algoritmasında arama işlemi için zaman karmaşıklığı Θ(1)-O(lgN) arasında değişmekte olup en çok aranan elemanlar için Θ(1), en az aranan elemanlar için O(lg N) olmaktadır.

Piramit arama algoritmasında frekans temelli bir arama gerçekleştirildiğinden büyük veri kümelerinde kullanılabilir. Şöyle ki, en çok aranan elemanların piramidin tepesine doğru en az arananların da piramidin altlarına doğru yerleştiği bir yapıda en çok aranan elemanlara ulaşmak daha az zaman alacaktır. Piramit arama algoritması arama motorlarında en çok aranandan en aza doğru aranan verileri sınıflandırmada, veri tabanı uygulamalarında örneğin bir sözlük (İngilizce-Türkçe gibi) uygulamasında en çok aranan kelimelere daha hızlı erişmede etkin bir şekilde kullanılabilir.

Önerilen yeni görev zamanlayıcı algoritması FPS, adeta CFS ve O(1) görev zamanlama algoritmalarının birleşimi olarak çalışmaktadır. Yani hem öncelik hem de sanal çalışma süresi değerine göre görevlerin çalışmasını sağlamaktadır. Önerilen yeni görev zamanlayıcı algoritması (FPS) öncelik uygularken daha adil bir yöntem kullanmaktadır. Yani hem önceliği hem de görev işletim süresini göz önünde bulundurarak işletmektedir. Böylece tüm görevlere önceliği de göz önünde bulundurarak daha kısa sürede sıra gelmektedir.

Atlamalı liste veri yapısı yerine, atlamalı halka veri yapısının kullanılmasının sebebi görev zamanlayıcı algoritmalarının büyük bir çoğunluğu dairesel (circular) yapıdadır. Yani görev listesi tekrar başa dönmektedir bu yüzden atlamalı halka veri yapısı kullanılmıştır. Dairesel (çevrimsel) olmayan durumlarda atlamalı liste veri yapısı tercih edilebilir.

Önerilen yeni veri yapısı, yapılan bu çalışmalara bakıldığında değişik alanlara verimli bir şekilde uygulanabilmektedir. Verileri sıralamak için, veriler üzerinde arama yapmak için kullanılabilmektedir. Ayrıca işletim sistemlerinde görev zamanlayıcı algoritması tasarımında kullanılabilmektedir. Kısaca dairesel bağlı listelerin kullanıldığı hemen her yerde, ağaç veri yapılarının kullanıldığı birçok yerde atlamalı halka (skip ring) veri yapısını kullanmak mümkündür.

TEZDEN TÜRETİLEN YAYINLAR/SUNUMLAR

Aksu, M., Karcı, A.," Skip List Data Structure Based New Searching Algorithm and Its Applications: Priority Search", (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 7, No. 2, 2016, pp.149-154
(DOI: https://dx.doi.org/10.14569/IJACSA.2016.070220)

Aksu, M., Karcı, A., "Skip Ring/Circular Skip List: Circular Linked List Based New Data Structure", Computer Engineering and Intelligent Systems. 6(5), pp. 35-41 (2015)

Aksu, M., Karcı, A., Yılmaz, Ş., "Effects of P Threshold Values in Creation of Random Level and to the Performance of Skip List Data Structure", Bitlis Eren University Journal of Science. 2(2), pp. 148-153 (2013)

Aksu, M.; Karcı, A., Yılmaz, Ş., "Level optimization in Skip List data structure", 1ST International Symposium on Innovative Technologies in Engineering and Science (ISITIES2013), Adapazarı, 7-9 June 2013, pp.389-396

Aksu, M., Canayaz, M., Karcı, A., "Image Representation as Skip Graph", 1st International Eurasian Conference on Mathematical Sciences and Applications (IECMSA-2012), KOSOVO, PRISHTINE, 03-07 September 2012, pp. 152-153.

7. KAYNAKLAR

[1] M. Aksu, A. Karcı, Ş. Yılmaz, Level optimization in Skip List data structure, Proceeding of 1ST International Symposium on Innovative Technologies in Engineering and Science (ISITIES2013), (2013), pp. 389-396.

[2] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2009, pp. 229–338.

[3] C. A. Shaffer, Data Structures & Algorithm Analysis in C++, Dover Publications, 2013, 615 pages.

[4] W. Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees, **Communications of the ACM**, Vol. 33, No. 6, (1990), pp. 668-676.

[5] W. Pugh, Concurrent Maintenance of Skip Lists, Dept. of Computer Science, University of Maryland, College Park, Technical report, TR–2222.1, 1989.

[6] W. Pugh, A Skip List Cookbook, Dept. of Computer Science, University of Maryland, College Park, Technical report, CS–TR–2286.1, 1989.

[7] M. Herlihy, Y. Lev, V. Luchangco, N. Shavit, A Provably Correct Scalable Concurrent Skip List, Proceedings of the 10th International Conference On Principles Of Distributed Systems (OPODIS), 2006, 15 pages.

[8] P. Kirschenhofer, C. Martinez, H. Prodinger, Analysis of an optimized search algorithm for skip lists, **Theoretical Computer Science**, Vol. 144, 1995, pp. 199-220.

[9] M. Herlihy, Y. Lev, V. Luchangco, N. Shavit, A Simple Optimistic Skiplist Algorithm, Proceeding of SIROCCO, Lecture Notes in Computer Science, Vol. 4474, 2007, pp. 124-138.

[10] L. Devroye, A limit theory for random skip lists, **Annals of Applied Probability**, Vol. 2, No. 3, (1992), pp. 597– 609.

[11] T. Papadakis, Skip lists and probabilistic analysis of algorithms, PhD Thesis, University of Waterloo, Tech. Report CS-93-28, 1993.

[12] T. Papadakis, J. I. Munro, P. V. Poblete, Average search and update costs in skip lists, **BIT**, Vol. 32, (1992), pp. 316-332.

[13] P. Kirschenhofer, H. Prodinger, The path length of random skip lists, **Acta Informatica**, Vol. 31, No. 8, (1994), pp. 775-792.

[14] P. V. Poblete, J. I. Munro, T. Papadakis, The binomial transform and the analysis of skip lists, **Theoretical Computer Science**, Vol. 352, (2006), pp. 136-158.

[15] J. I. Munro, T. Papadakis, P. V. Poblete, Deterministic Skip Lists, Proceeding of SODA '92 Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms, (1992), pp. 367-375.

[16] M. Aksu, A. Karcı, Ş. Yılmaz, Effects of P Threshold Values in Creation of Random Level and to the Performance of Skip List Data Structure, **Bitlis Eren University Journal of Science**, Vol. 2, No. 2, (2013), pp. 148-153.

[17] J. Aspnes, G.Shah, Skip graphs, Proceeding of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), (2003), pp. 384–393.

[18] I. Abraham, J. Aspnes, J. Yuan, Skip B-trees, Proceeding of OPODIS 2005, Principles of Distributed Systems; 9th International Conference, Italy, (2005), pp. 366–380.

[19] X. Messeguer, Skip trees, an alternative data structure to Skip lists in a concurrent approach, **RAIRO - Theoretical Informatics and Applications**, Volume: 31, Issue: 3, (1997), pp. 251-269.

[20] P. Bose, K. Douïeb, P. Morin, Skip lift: A probabilistic alternative to red–black trees, **Journal of Discrete Algorithms**, 14, (2012), pp. 13–20.

[21] M. Aksu, A. Karcı, Skip List Data Structure Based New Searching Algorithm and Its Applications: Priority Search, **(IJACSA) International Journal of Advanced Computer Science and Applications**, Vol. 7, No. 2, (2016), pp.149-154.

[22] T. Clouser, M. Nesterenko, C. Scheideler, Tiara: A self-stabilizing deterministic skip list and skip graph, **Theoretical Computer Science**, Volume 428, (2012), pp. 18–35.

[23] I. Lotan and N. Shavit, SkipList-Based Concurrent Priority Queues, Proceeding of International Parallel and Distributed Processing Symposium, Cancun, Mexico, 2000.

[24] R. ÇÖLKESEN, Veri Yapıları ve Algoritmalar, PAPATYA YAYINCILIK, 2012, 480 p.

[25] N. Wirth, Algorithms and Data Structures, Oberon version, 2004 (Last update 2014-10-05), 211 Pages.

[26] U. Aybars, Veri Yapıları Ders Notları, Department of Computer Engineering, Ege University, Fifth Edition, İzmir, 2009, 93 p.

[27] K. Loudon, Mastering Algorithms with C, O'Reilly Media, 1999, 562 p.

[28] G. Barnett, L.D. Tongo, Data Structures and Algorithms: Annotated Reference with Examples, First Edition ,2008, 112 p, http://dotnetslackers.com/projects/Data-Structures-And-Algorithms/

[29] M. H. Alsuwaiyel, ALGORITHMS DESIGN TECHNIQUES AND ANALYSIS, World Scientific Publication, 1999, 544 p.

[30] M. EGE, VERİ YAPILARI ve ALGORİTMALAR Ders notları, http://hpss.endustri.cu.edu.tr/ders/ENS255/Veri%20Yap%C4%B1lar%C4%B1%20ve%20Algoritmalar%202.pdf, 91 p.

[31] İ.H. Cedimoğlu, Veri Yapıları ve Programlama, Sakarya Üniversitesi, Adapazarı Meslek Yüksekokulu, 2006, pp. 4-6.

[32] R. Sedgewick, P. Flajolet, AN INTRODUCTION TO THE ANALYSIS OF ALGORITHMS, Second Edition, Addison Wesley, 2013, 572 p.

[33] S. S. Skiena, The Algorithm Design Manual, Second Edition, New York, USA, Springer Science+Business Media, 2008, 730 Pages.

[34] G. Cheng, A Comparison of Two Linux Schedulers, Master thesis, Department of Informatics, University of Oslo, Oslo, Norway, 2012.

[35] Wikipedia, The free encyclopedia, http://en.wikipedia.org/wiki/Red-black_tree. (on-line accessed on 15-10-2015)

[36] J. Besa, Y. Eterovic, A concurrent red–black tree, **J. Parallel Distrib. Comput.** 73; (2013), pp. 434–449.

[37] J. Morris. Data Structures and Algorithms: Red-Black Trees, Department of Computer Science, The University of Auckland, 1998.

[38] Red-black and b trees. "http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html#RTFToC1", Lecture Notes, Computer Science, San Diego State University, 1995. (on-line accessed on 15-01-2016)

[39] M. T. Jones. Inside the linux 2.6 completely fair scheduler. IBM developer Works, http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/, (2009), pp. 1-8,

[40] Wikipedia, The free encyclopedia, http://en.wikipedia.org/wiki/Tree_rotation. (on-line accessed on 25-11-2015)

[41] A. Levitin, Introduction to Design and the Analysis of Algorithms, Pearson Education, 2012, 565 Pages.

[42] http://www.algoritma.org/2011/05/algoritma-karmasklg-ve-buyuk-o.html, (on-line accessed on 15-10-2014)

[43] T. Niemann, SORTING AND SEARCHING ALGORITHMS, Erişim Tarihi: 21.09.2015, http://epaperpress.com/sortsearch/download/sortsearch.pdf

[44] M. Ü. Karakaş, Bilgisayar Yazılımında Veri Yapıları ve Algoritmalar, İkinci Basım, Beta Yayıncılık, İstanbul, 2000, 290 p.

[45] J. Kleinberg, É. Tardos, Algorithm Design, Pearson Education, 2006, 838 Pages.

[46] D. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, second ed., Addison-Wesley, 1998.

[47] M. J. Dinnen, G. Gimel'farb, M. C. Wilson, Introduction to Algorithms, Data Structures and Formal Languages, Pearson Education, Second edition, 2009, 254 p.

[48] R. Sedgewick, K. Wayne, Algorithms, Addison Wesley, Fourth Edition, America, 2011.

[49] K. Mehlhorn, P. Sanders, Algorithms and Data Structures; The Basic Toolbox, Springer, 2007, 285 p.

[50] M. McMillan, Data Structures and Algorithms Using C#, Cambridge University Press, 2007, 355 p.

[51] M. T. Goodrich, R. Tamassia, Algorithm Design and Applications, Wiley, America, 2014, 784 p.

[52] C.A.R. Hoare, Quicksort. **The Computer Journal**, Vol. 5, No. 1, (1962), pp. 10-15.

[53] J. W. J. Williams, Algorithm 232 - Heapsort, **Communications of the ACM** 7 (6): (1964), pp. 347–348, doi:10.1145/512274.512284.

[54] E. Nardelli, G. Proietti, Efficient unbalanced merge–sort, **Information Sciences**, Vol. 176, (2006), pp. 1321–1337, doi:10.1016/j.ins.2005.04.008.

[55] J. Alnihoud, R. Mansi, An Enhancement of Major Sorting Algorithms, **The International Arab Journal of Information Technology**, Vol. 7, No. 1, (2010), pp. 55–62.

[56] K. K. Sundararajan, S. Chakraborty, A new sorting algorithm, **Applied Mathematics and Computation**, Vol. 188, (2007), pp. 1037–1041, doi:10.1016/j.amc.2006.10.065.

[57] S. Jadoon, S. F. Solehria, M. Qayum, Optimized Selection Sort Algorithm is faster than Insertion Sort Algorithm: a Comparative Study, **International Journal of Electrical & Computer Sciences (IJECS-IJENS)** Vol. 11, No. 02, (2011), pp. 18-23.

[58] M. T. Goodrich, Randomized Shellsort: A Simple Oblivious Sorting Algorithm, Proceeding of 21st ACM-SIAM Symposium on Discrete Algorithms (SODA), (2010), pp. 1262-1277.

[59] P. M. E. Shutler, S. W. Sim, W. Y. S. Lim, Analysis of Linear Time Sorting Algorithms, **THE COMPUTER JOURNAL**, The British Computer Society, Vol. 51 No. 4, (2008), doi:10.1093/comjnl/bxm097.

[60] A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts, John Wiley & Sons, Eighth edition, America, 2009, pp. 183-222, 815-820.

[61] F. Pfenning, Red/Black Trees, Department of Computer Science, Carnegie Mellon University, Lecture Notes in Computer Science, (2010), pp. 2-15.

[62] Aksu, M., Karcı, A., Skip Ring/Circular Skip List: Circular Linked List Based New Data Structure, **Computer Engineering and Intelligent Systems**. 6(5), (2015), pp. 35-41.

ÖZGEÇMİŞ

Ad Soyad	**:** Mustafa AKSU
Doğum Yeri ve Tarihi	**:** KAHRAMANMARAŞ – 05.10.1973
Adres	**:** Kahramanmaraş Sütçü İmam Üniversitesi, Teknik Bilimler Meslek Yüksek Okulu, Bilgisayar Teknolojileri Bölümü , Kahramanmaraş
E-Posta	**:** m.aksu@ksu.edu.tr, m.aksu@hotmail.com
Lisans	**:** Kocaeli Üniversitesi Bilgisayar Bilimleri Mühendisliği
Yüksek Lisans	**:** Sütçü İmam Üniversitesi Elektrik-Elektronik Mühendisliği
Mesleki Deneyimler	**:** Hasan Şahan Kışlası Elektro Optik Sistemler Bakım Merkez Müdürlüğü'nde bilgi işlem ve yazılım sorumlusu Kocaeli Üniversitesi Bilgisayar Mühendisliği Bölümü araştırma görevlisi (2000-2001) Haziran 2001 tarihinden itibaren Kahramanmaraş Sütçü İmam Üniversitesi Meslek Yüksek Okulu Bilgisayar Teknolojileri Bölümünde öğretim görevlisi olarak çalışmaktadır.