VERİ YAPILARI VE ALGORİTMALAR

BLM2512 Gr.2

2020-2021 Bahar Yarıyılı (Uzaktan Eğitim)

Dr.Öğr.Üyesi Göksel Biricik

PRIORITY QUEUE

Öncelikli Kuyruk

Öncelikli Kuyruk (Priority Queue)

- Her biri bir anahtar ile ilişkilendirilmiş bir eleman kümesini tutan bir veri yapısıdır.
- Anahtarların (büyük ya da küçük) değerlerine göre elemanları önceliklendirebilir ve bu sıra ile işleyebiliriz.
- İlk işlenecek eleman her zaman 1.gözde olduğu için bize kolaylık sağlar.
  - Heap ile ekleme ve çıkarma işlemlerini **O(logN)** karmaşıklıkla yapabiliriz.

Öncelikli Sıra (Priority Queue)

- Max-priority-queue → Max-Heap ile
  - **insert**(s,x) : x önceliğine sahip bir elemanı kuyruğa ekleme
  - **max**(s) : En büyük önceliği sahip elemanı sorgulama
  - **extract_max**(s) : En büyük öncelikli elemanı kuyruktan çıkarma
  - **increase_key**(s,x,k) : x önceliğine sahip elemanın önceliğini k'ya çıkarma (k >= x)
- Ör: İşletim sisteminde işlemleri (process) sıralamak için.
- Her işlemin bir önceliği vardır.
- Bir iş biterse ya da kesintiye uğrarsa, zamanlayıcı (scheduler) sıradaki en yüksek önceliğe sahip işlemi extract_max ile devreye alır.
- Zamanlayıcı herhangi bir anda insert ile yeni bir iş ekleyebilir.
- Gerektiğinde bir işin önceliği arttırılabilir (ya da düşürülebilir).

Öncelikli Sıra (Priority Queue)

- Min-priority-queue → Min-Heap ile
  - **insert**(s,x)
  - **min**(s)
  - **extract_min**(s)
  - **decrease_key**(s,x,k)
- Ör: Olay güdümlü simülatör. Simüle edilecek işlerin oluşma (gerçekleşme) zamanları, anahtarlarıdır.
- Olaylar gerçekleşme zamanlarına göre simüle edilir, çünkü bir olayın simülasyonu gelecekte başka olayların da simüle edilmesine neden olabilir.
- Simülatör, extract_min ile bir sonraki işi seçer.
- Yeni oluşan olaylar, insert ile kuyruğa eklenir.

En büyük (ya da küçük) elemanı bulmak

- Heap'in kök elemanı ☺

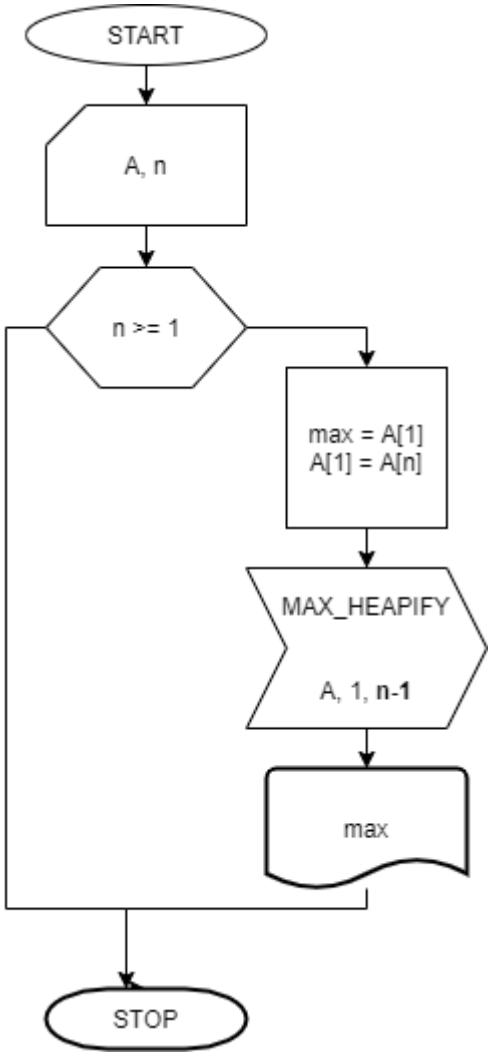
```
HEAP_MAXIMUM ( A )
        return A [ 1 ]
```

- O(1)

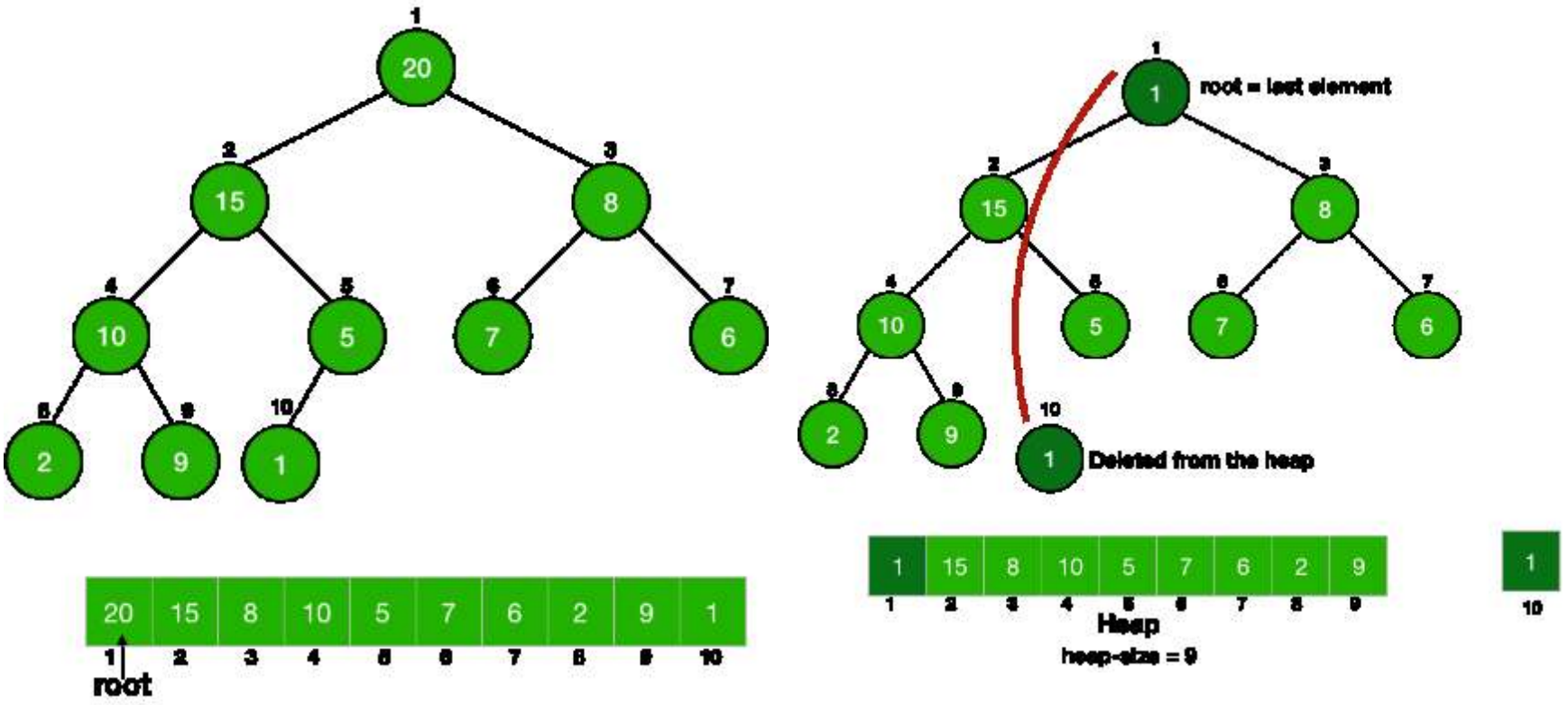
En büyük elemanı kuyruktan çıkarmak

- Heap boş olmamalı
- Kök elemanın kopyası çıkarılır (max)
- Son düğüm kök düğümü yapılır
- Bir az eleman sayısı ile «MAX_HEAPIFY» yapılır (heap boyutu 1 azaltılır)
- max değeri işlemler için çağırana döndürülür

- O(logN)



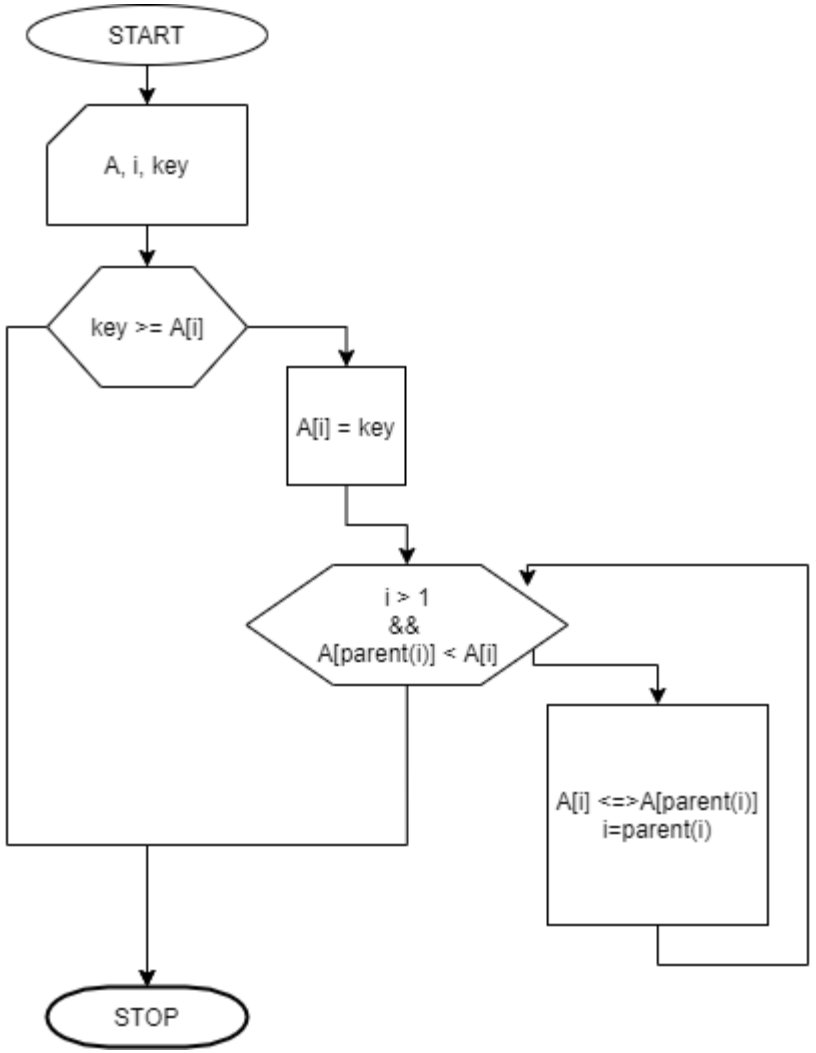
En büyük elemanı kuyruktan çıkarmak



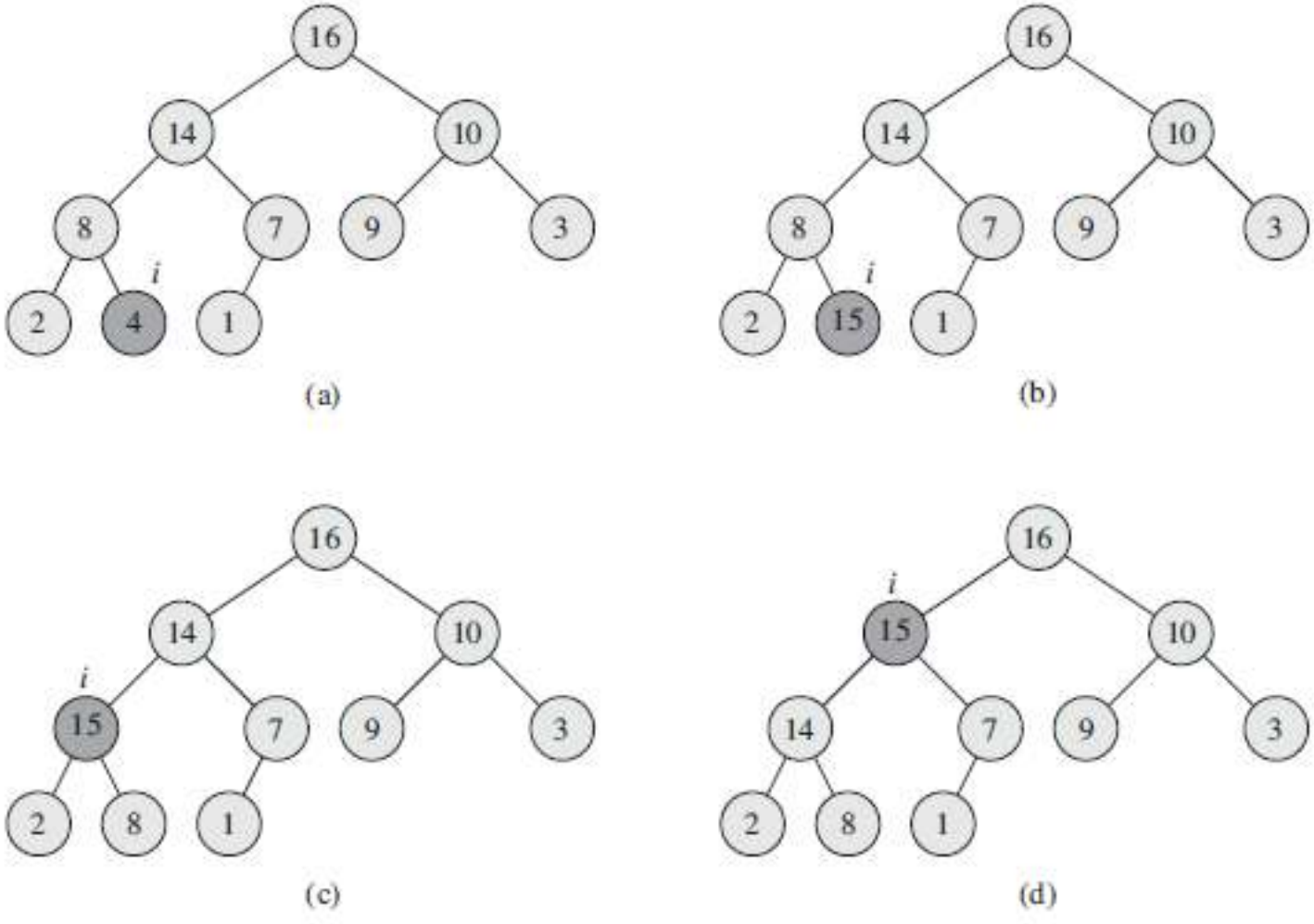
Kuyrukta anahtar değerini arttırmak

- k (yeni değer), mevcuttan büyük-eşit olmalı
- x düğümünün değeri k yapılır
- ebeveyn(ler)iyle karşılaştırılarak, gerektiğinde (ebeveyninden büyük olduğu halde) yeri değiştirilir

- O(logN)



Kuyrukta anahtar değerini arttırmak



Kuyruğa yeni bir eleman eklemek

- Elimizde HEAP_INCREASE_KEY fonksiyonu var.
- Max_Heap'in boyu bir arttırılır, son elemanın anahtar değerine –INF atanır.
- -INF yerine «k» değeri HEAP_INCREASE_KEY ile atanır.

```
MAX_HEAP_INSERT(A, key, n)
        A[n+1] ← - ∞
        HEAP_INCREASE_KEY (A, n+1, key)
```

- O(logN)

Max-Priority Queue C Kodu

```
int tree_array_size = 20;
int heap_size = 0;
const int INF = 100000; //use INT_MAX in limits.h

void swap( int *a, int *b ) {
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
```

Max-Priority Queue C Kodu

```
int get_right_child(int A[], int index) {
  if((((2*index)+1) < tree_array_size) && (index >= 1))
    return (2*index)+1;
  return -1;
}

int get_left_child(int A[], int index) {
    if(((2*index) < tree_array_size) && (index >= 1))
        return 2*index;
    return -1;
}

int get_parent(int A[], int index) {
  if ((index > 1) && (index < tree_array_size)) {
    return index/2;
  }
  return -1;
}
```

Max-Priority Queue C Kodu

```
void max_heapify(int A[], int index) {
  int left_child_index = get_left_child(A, index);
  int right_child_index = get_right_child(A, index);

  // finding largest among index, left child and right child
  int largest = index;

  if ((left_child_index <= heap_size) && (left_child_index>0)) {
    if (A[left_child_index] > A[largest]) {
      largest = left_child_index;
    }
  }

  if ((right_child_index <= heap_size && (right_child_index>0))) {
    if (A[right_child_index] > A[largest]) {
      largest = right_child_index;
    }
  }

  // largest is not the node, node is not a heap
  if (largest != index) {
    swap(&A[index], &A[largest]);
    max_heapify(A, largest);
  }
}
```

Max-Priority Queue C Kodu

```
void build_max_heap(int A[]) {
  int i;
  for(i=heap_size/2; i>=1; i--) {
    max_heapify(A, i);
  }
}
```

Max-Priority Queue C Kodu

```
int extract_max(int A[]) {
  int maxm = A[1];
  A[1] = A[heap_size];
  heap_size--;
  max_heapify(A, 1);
  return maxm;
}
```

Max-Priority Queue C Kodu

```
void increase_key(int A[], int index, int key) {
  A[index] = key;
  while((index>1) && (A[get_parent(A,index)] < A[index])) {
    swap(&A[index], &A[get_parent(A, index)]);
    index = get_parent(A, index);
  }
}

void decrease_key(int A[], int index, int key) {
  A[index] = key;
  max_heapify(A, index);
}
```

Max-Priority Queue C Kodu

```
void insert(int A[], int key) {
  heap_size++;
  A[heap_size] = -1*INF;
  increase_key(A, heap_size, key);
}
```

Max-Priority Queue C Kodu

```
void print_heap(int A[]) {
  int i;
  for(i=1; i<=heap_size; i++) {
    printf("%d\n",A[i]);
  }
  printf("\n");
}

int main() {
  int A[tree_array_size];

  insert(A, 20); insert(A, 15); insert(A, 8); insert(A, 10); insert(A, 5);
  insert(A, 7); insert(A, 6); insert(A, 2); insert(A, 9); insert(A, 1);
  print_heap(A);

  increase_key(A, 5, 22);
  print_heap(A);

  decrease_key(A, 1, 13);
  print_heap(A);

  printf("%d\n\n", maximum(A));
  printf("%d\n\n", extract_max(A));
  print_heap(A);

  printf("%d\n", extract_max(A)); printf("%d\n", extract_max(A)); printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A)); printf("%d\n", extract_max(A)); printf("%d\n", extract_max(A));
  printf("%d\n", extract_max(A)); printf("%d\n", extract_max(A)); printf("%d\n", extract_max(A));
  return 0;
}
```

HUFFMAN AĞACI

Huffman Ağacı

- Veriyi kayıpsız sıkıştırma için, karakterleri kullanım sıklıklarına göre **önceliklendirir**.
  - Öncelikli kuyruk ile gerçekleştirebiliriz.
- En sık kullanılan karaktere en kısa kodu,
- En az kullanılan karaktere en uzun kodu

verirsek, sabit kod uzunluğuna göre daha kısa bir çıktı üretebiliriz.

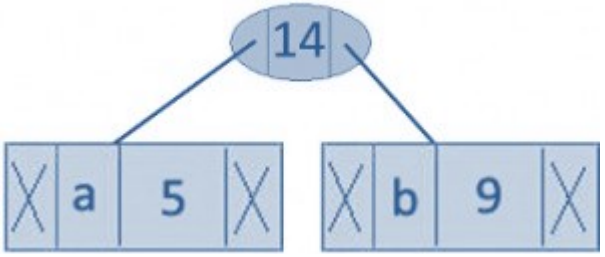
  - Denk sıklıktaki karakterler aynı uzunluktaki kodu alacaktır.

Huffman Ağacı Oluşturma

1. Tüm tekil karakterleri, kullanım sıklıklarına göre bir Min_Priority_Queue içine alırız.
   - En az rastlanan kökte olacaktır.
   - Bu sayede en az sıklıktaki karakterleri öncelikle çekip işleriz, sonuç ağacımızda en dipte yer alırlar.
   - En sık rastlanan karakterler en sonlarda işleneceği için, sonuç ağacında kökte/köke yakın yer alırlar (en kısa kod uzunluğu)
2. Kuyruktan iki (minimum frekanslı) düğüm çıkarılır.
3. Bu iki düğümün frekansları toplanır, kullanım sıklığı olarak bir ara düğüme atanır. Düğüm kuyruğa eklenir.
   - Orijinal min_1 ve min_2 düğümleri bu ara düğümün sol ve sağ çocuklarıdır.
4. Kuyrukta tek bir düğüm kalana kadar 2 ve 3 tekrarlanır.
   - Kalan düğüm Huffman ağacımızın köküdür (en yüksek frekans)

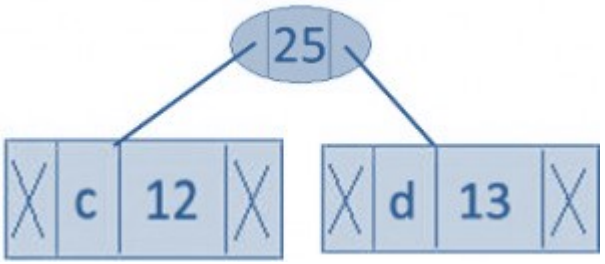
Huffman Ağacı Oluşturma-1

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45



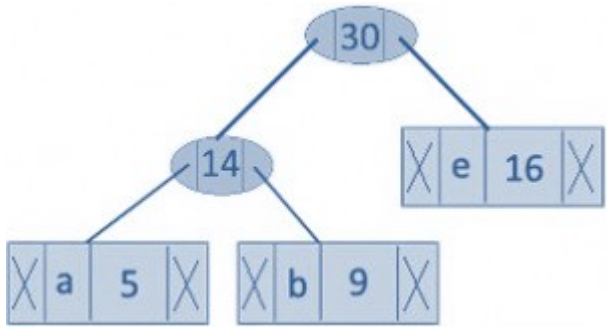
Huffman Ağacı Oluşturma-2

character	Frequency
c	12
d	13
Int.Node	14
e	16
f	45



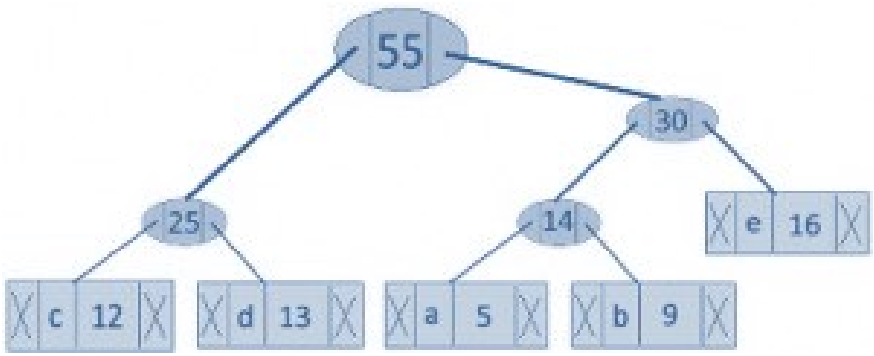
Huffman Ağacı Oluşturma-3

character	Frequency
Int.Node	14
e	16
Int.Node	25
f	45



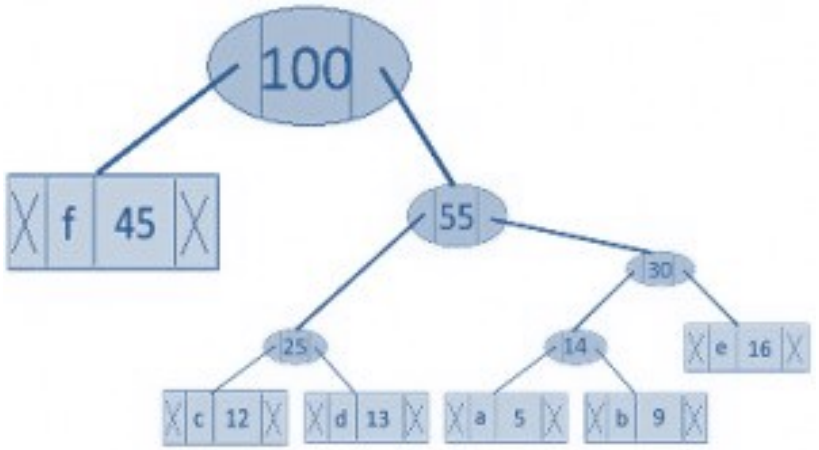
Huffman Ağacı Oluşturma-4

character	Frequency
Int.Node	25
Int.Node	30
f	45



Huffman Ağacı Oluşturma-5

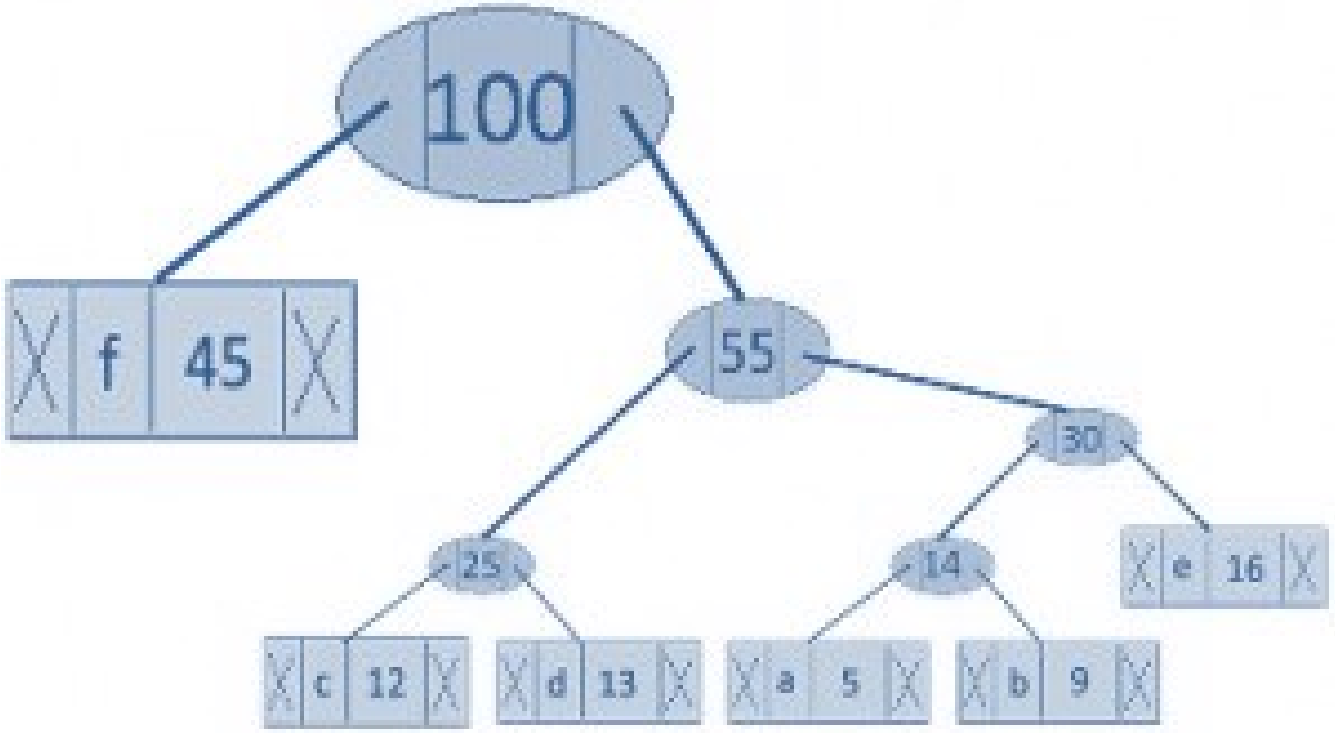
character	Frequency
f	45
Int.Node	55



Huffman Ağacı Oluşturma-6

character Frequency

Int.Node 100



Huffman Ağacı Kodları

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111