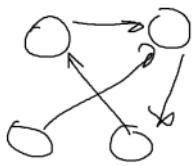
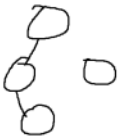
Graph ve Ağaç

26 Kasım 2025 Çarşamba 19:15

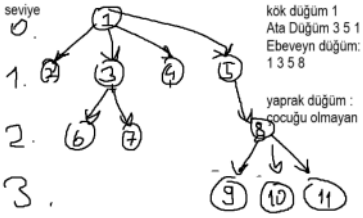


Döngü içeren bağlı graph



bağlı graph

Döngü olmayan yönlü ve bağlı grapha ağaç denir



kök düğüm 1
Ata Düğüm 3 5 1
Ebeveyn düğüm:
1 3 5 8

yaprak düğüm :
çocuğu olmayan 2 4

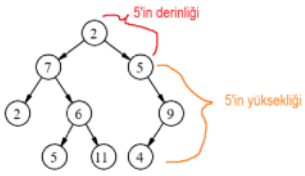
düğümün yüksekliği:
o düğümden ulaşılabilen en uzaktaki yaprağa olan kenar sayısı
örneğin h(5)=2
h(7) = 0
h(3) = 1

Ağaçın yük = kök yük

Derinlik : köke olan uzaklık (yukardan)

yükseklik : aşağıdan

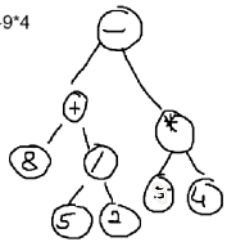
1.dereceden ataya ebeveyn denir. Yani bir üstteki hem ebeveyn hem ata iken daha da üstteki düğümleri sadece atadır



İkili ağaç:
En fazla 2 çocuğu olmalıdır.

İfade Ağacı:

8+5/2-9*4



Ağaç Dolaşımları

26 Kasım 2025 Çarşamba 19:32

İn Order Dolaşım:	Sol dolaş kendini yaz sağ dolaş	Sol ken sağ	infix

Preorder: kendini yaz sol dolaş sağ dolaş	Ken sol sağ	prefix

Postorder: Sol dolaş sağ dolaş kendini yaz	Sol sağ ken	postfix

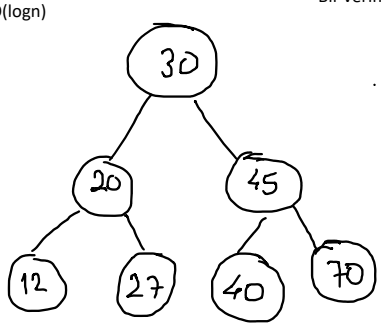
LevelOrder:	Her seviye soldan sağa dolaşılır	levelfix

BST

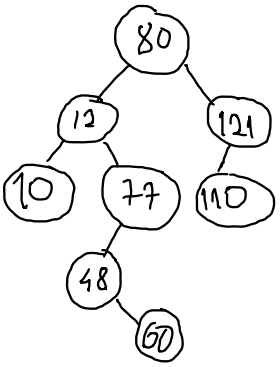
26 Kasım 2025 Çarşamba 19:45

Dizi O(n)
Liste O(n)
İkili Arama Ağacı O(logn)

Bir verinin solundaki değerler daima ondan küçük sağındakiler daima ondan büyük olmalıdır.



Örnek 2: 80 12 77 48 19 121 110 60



h = 4

Silme İşlemi:

1- Silinecek düğümün hiç çocuğu yok (direkt silinir)
2- Silinecek düğümün 1 Çocuğu var (liste benzeri silme işlemi ebeveyni bağla)
3- Silinecek düğümün 2 Çocuğu var (solun en sağına sağın en soluna gidilir.)

Diziye yerleştirilmesi level order ile yapılır

80	12	121	10	77	110				48											60

Sol cocuk: 2*index+1
Sag cocuk: 2*index+2
Parent = (index-1) /2

Aynı uzunlukta bir dizi daha oluşturulur

Heap sortta buna gerek yoktur çünkü tam dolu ağaç kullanılır.
Dizili versiyonda silme kodunu bilmene gerek yok

Linked BST'de düğüm silmek:
İlk if: ya sol çocuğu var yada hiç çocuğu yok, hiç yoksa sıkıntı yok çünkü subNode = subNode->left derken direkt null yapacak silmiş olacak.

İkinci if:
ilk if i geçtiği için her türlü sağ çocuğa sahip demektir sol çocuğu yoksa direkt parenti sağ çocuğuna bağla

Else (iki çocuğu varsa):
subNode u parent diye bir değişkene at. Solun en sağına inicez bu sebeple p yi de sol çocuk yap. Sonra daha fazla sağa inemeyene kadar bu ikisi birbirini kovalar. p harfi parentın sol çocuğuydu bu sebeple ilk hamle olarak parent sola iner sonra sağa inerek devam eder.

p'nin sağı kalmayana kadar şu tekrarlanır:

parent p'ye git.
p, sağa git.

en son şu kontrol edilir parent ilk hamlesini yani sola inme işlemini yaptı mı? Yapamadıysa parentın solu p'nin soluna bağlanır. Eğer yaptıysa koparılacak ağaç sağdadır parentın sağı, p'nin soluna bağlanır.

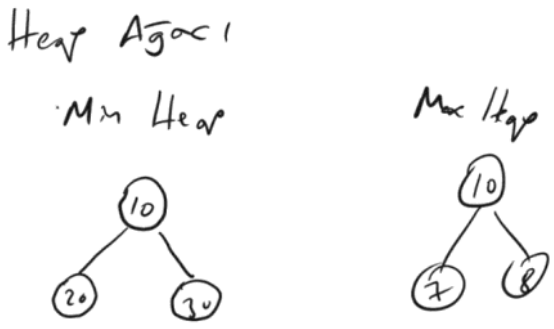
Çünkü p'nin sağı kalmadı varsa en fazla solu vardır onu kaybetmemek için bağlamamız gerek.

Eğer parent hiç hareket etmediyse o zaman p onun solunda kalmış olacak
yine p yi silmeden önce p nin solunu bağlamamız gerek ama bu sefer parentın soluna.

```
bool deleteNode(Node *&subNode){
        Node *p = subNode;
        if(subNode->right == NULL) subNode = subNode->left;
        if(subNode->left == NULL) subNode = subNode->right;
        //iki çocuğu varsa
        else{
            p = subNode->left;
            Node *parent = subNode;
            while(p->right != NULL){
                parent = p;
                p = p->right;
            }
            subNode->data = p->data;
            if(parent == subNode) subNode->left = p->left;
            else parent->right = p->left;
        }
        delete p;
        return true;
    }
```

Heap tree

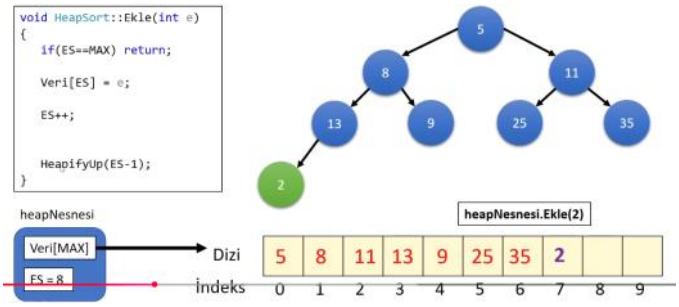
19 Aralık 2025 Cuma 12:20



Küçükten büyüğe büyükten küçüğe

HeapfyUp() ve HeapfyDown() fonkları vardır.

Heap Ağacına Eleman Ekleme



heapfyUp fonku ekleme yapılırken çağrılır. Küçükse yukarı taşır. Başta sondakinin (l-1) indexini alır index 0 değilse parentini bulur eğer parenti daha büyükse swapler ve parent için heapfyUp çağrılır (artık içinde swaplendiği için parentin içinde yeni eklediğimiz değeri var) eğer küçük kalmaya devam ederse köke kadar böyle devam eder.

```
void HeapTree::add(int item)
{
    if (capacity == length) throw "agac dolu";
    items[length] = item;
    length++;
    HeapfyUp(length-1);
}
```

Küçük değeri çıkardıktan sonra ağaç yapısını korumak ve boşluğu önlemek için sondaki eleman köke (başa) getirilir. Sonrasında dengelemek için eğer sayı çocuklardan büyükse en küçük olan çocukla yer değiştirilir.

```
int HeapTree::getMinValue()
{
    if(isEmpty()) throw "Tree is empty";
    int minValue = items[0];
    items[0] = items[length-1];
    length--;
    if (length > 0) HeapfyDown(0);
    return minValue;
}

void HeapTree::HeapfyDown(int index)
{
    int rightIndex, leftIndex, minIndex;
    leftIndex = LeftChildIndex(index);
    rightIndex = RightChildIndex(index);
    if(rightIndex >= length){
        if(leftIndex > length) return;
        else minIndex = leftIndex;
    }
    else{
        if(items[leftIndex] <= items[rightIndex])
            minIndex = leftIndex;
        else minIndex = rightIndex;
    }
    if(items[index] > items[minIndex]){
        swap(items[index], items[minIndex]);
        HeapfyDown(minIndex);
    }
}
```

```
void HeapTree::HeapfyUp(int index)
{
    int parentIndex;
    if(index != 0){
        parentIndex = ParentIndex(index);
        if(items[parentIndex] > items[index]) {
            swap(items[parentIndex],items[index]);
            HeapfyUp(parentIndex);
        }
    }
}
```

```
void HeapTree::Swap(int & item1, int & item2)
{
    int tmp;
    tmp = item1;
    item1 = item2;
    item2 = tmp;
}

//swap referanslı alır
```

Önce sağ çocuğun beklenen indexi length'i aşıyor mu? (yani sağ yok mu)
Sonra sol çocuğun beklenen indexi length'i aşıyor mu (yani sol yok mu)

Sağ yok sol yok -> hiçbişi yapma return
Sağ yok sol var -> minIndex = sol

İki durum da uymadı.
Sağ var sol var -> sağ sol karşılaştır sağ büyükse minIndex = sol
Yoksa -> mindex -> sağ

Bizim değer, minIndexten büyük ise swaple ve swaplenen yani artık yeni küçük olan değer üzerinden heapfyDown'ı yeniden çağır.

BST(Dizi)

26 Aralık 2025 Cuma 02:04

```
void add(const T& item){
    int suankiIndex = 0;
    while(true){
        if(itemStatus[suankiIndex] == 0){
            items[suankiIndex] = item;
            itemStatus[suankiIndex] = 1;
            break;
        }
        else if(item < items[suankiIndex])
            suankiIndex = 2*suankiIndex+1;
        else if(item > items[suankiIndex])
            suankiIndex = 2*suankiIndex+2;
        else break;
    }
}
```

//curr = 0

İtemstatus dizisi eleman varsa 1 yoksa 0 tutar

CurrIndexteki statü 0 olan yeri bulana kadar
item küçükse sol çocuğa
İtem büyükse sağ çocuğa

currIndexi ilerlet

currIndex statüsü boş bulduğun yere yerleş statüsünü 1 yap

Ağaçların görevleri

1 Ocak 2026 Perşembe 20:47

Heap Tree

- Dizi üzerinde gerçekleştirilen tam dolu bir bt olması sayesinde heapfyup ve heapfydown fonksiyonlarını kullanarak kökteki değerin türüne göre (maxheap, minheap) min ya da max olmasını garanti ederek O(nlogn)'lik bir performans sağlar.
- heapfyup ve heapfydown fonksiyonlarını kullanarak öncelik hiyerarşisi kurup min yada max değerin kökte bulunmasını garanti ederek O(nlogn)'lik bir performans sağlamak.

BST

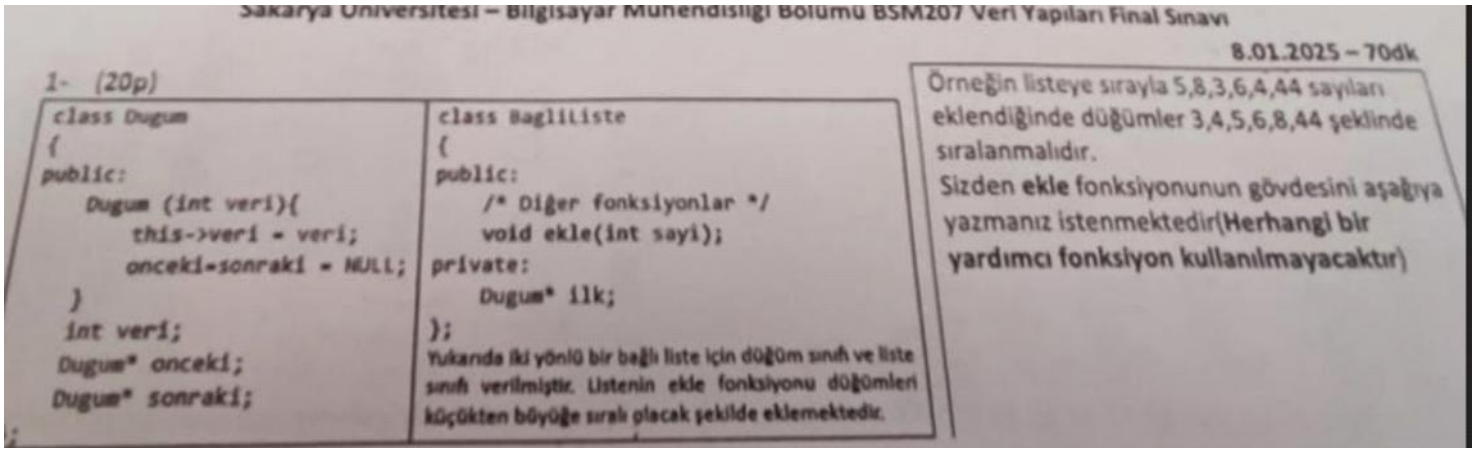
Arama uzayını yarıya bölerek O(logn)'lik bir performans sağlamak

AVL

BST ağacının O(logn)'den O(n)'e kayan performansını ağacı dengeleyerek korumaya çalışır.

sınav çalışması

1 Ocak 2026 Perşembe 22:51



Sakarya Üniversitesi – Bilgisayar Mühendisliği Bölümü BSM207 Veri Yapıları Final Sınavı

8.01.2025 – 70dk

1- (20p)

```
class Dugum
{
public:
    Dugum (int veri){
        this->veri = veri;
        onceki=sonraki = NULL;
    }
    int veri;
    Dugum* onceki;
    Dugum* sonraki;
};
```

```
class BagliListe
{
public:
    /* Diğer fonksiyonlar */
    void ekle(int sayi);
private:
    Dugum* ilk;
};
```

Yukarıda iki yönlü bir bağlı liste için düğüm sınıfı ve liste sınıfı verilmiştir. Listenin ekle fonksiyonu düğümleri küçükten büyüğe sıralı olacak şekilde eklemektedir.

Örneğin listeye sırayla 5,8,3,6,4,44 sayıları eklendiğinde düğümler 3,4,5,6,8,44 şeklinde sıralanmalıdır.
Sizden **ekle** fonksiyonunun gövdesini aşağıya yazmanız istenmektedir(**Herhangi bir yardımcı fonksiyon kullanılmayacaktır**)

```
void ekle(int sayi) {
    Dugum* yeni = new Dugum(sayi);

    // DURUM 1: Liste boşsa
    if (ilk == NULL) {
        ilk = yeni;
        return;
    }

    // DURUM 2: En başa eklenecekse (Gelen sayı, ilk düğümden küçükse)
    if (sayi < ilk->veri) {
        yeni->sonraki = ilk;
        ilk->onceki = yeni;
        ilk = yeni;
        return;
    }

    // DURUM 3: Araya veya en sona eklenecekse
    Dugum* itr = ilk;

    // İlerleyebildiğimiz kadar ilerleyelim (Sıradaki eleman var ve bizim sayıdan küçükse git)
    while (itr->sonraki != NULL && itr->sonraki->veri < sayi) {
        itr = itr->sonraki;
    }

    // Döngü bittiğinde 'itr' düğümünün hemen sonrasına ekleme yapacağız.

    yeni->sonraki = itr->sonraki; // 1. Yeni düğümün sağını bağla
    yeni->onceki = itr;          // 2. Yeni düğümün solunu bağla

    // Eğer sona eklemiyorsak, sağımızdaki düğümün 'onceki'sini bize bağla
    if (itr->sonraki != NULL) {
        itr->sonraki->onceki = yeni;
    }

    itr->sonraki = yeni;         // 3. Solumuzdaki düğümün sağını bize bağla
}
```

Önemli İpucu: İki yönlü bağlı listede araya ekleme yaparken **4 adet** bağlantı (pointer) güncellemesi yapıldığından emin olmalısınız (`yeni->ileri` , `yeni->geri` , `eski->ileri` , `sonroki->geri`).

```
void ekle(int sayi) {
    Dugum* p = new Dugum(sayi);

    // DURUM 1: Liste Boşsa
    if (ilk == NULL) {
        ilk = p;
        return;
    }

    Dugum* tmp = ilk;
    while (tmp != NULL) {
        // DURUM 2 ve 3: Araya veya Başa Ekleme (Küçük sayı bulduk)
        if (sayi < tmp->veri) {
            if (tmp == ilk) {
                // En başa ekleme
                p->sonraki = tmp;
                tmp->onceki = p;
                ilk = p;
                return;
            } else {
                // Araya ekleme (tmp'nin soluna)
                tmp->onceki->sonraki = p; // Soldaki düğümü p'ye bağla
                p->onceki = tmp->onceki;  // p'nin solunu bağla
                p->sonraki = tmp;         // p'nin sağını bağla
                tmp->onceki = p;          // tmp'nin solunu p yap
                return;
            }
        }

        // DURUM 4: Sona Ekleme
        // Eğer tmp son düğümse ve sayı hala büyükse, sona ekle
        if (tmp->sonraki == NULL) {
            tmp->sonraki = p;
            p->onceki = tmp;
            return;
        }

        tmp = tmp->sonraki;
    }
}
```

5- Parametre olarak aldığı veriyi iki yönlü bağıl listenin sonuna ekleyen ve ekledikten sonra iki yönlü bağli listeyi tersten ekrana yazdıran fonksiyonu C++ dilinde yazınız. Sadece istenen işlemlerin kodu yazılacaktır.

```
void ekleVeYaz(Node*& head, int veri){
```

```
void ekleVeYaz(Node*& head, int veri) {
    // 1. Yeni düğümü oluştur
    Node* p = new Node(veri);

    // 2. Liste boşsa durumu
    if (head == NULL) {
        head = p;
        cout << head->veri << endl;
        return; // BURAYA noktalı virgül eklendi
    }

    // 3. Listenin sonuna git
```

```
    Node* itr = head;
    while (itr->sonraki != NULL) { // 'While' -> 'while' yapıldı
        itr = itr->sonraki;
    }

    // 4. Yeni düğümü sona ekle ve bağları kur
    itr->sonraki = p;   // Sonuncunun sağına yeni düğümü koy
    p->onceki = itr;    // Yeni düğümün soluna eski sonuncuyu koy

    // Artık 'p' listenin en son elemanı oldu.
    // 'itr' göstericisini en sona (p'ye) taşıyalım.
    itr = p;

    // 5. Tersten Yazdırma (Sondan başa git)
    while (itr != NULL) {
        cout << itr->veri << endl;
        itr = itr->onceki; // Geri geri git
    }
}
```