K.T.Ü.

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

2011-2012 Güz Dönemi

Veri Yapıları Ders Notları

add: Yeni eleman eklemek

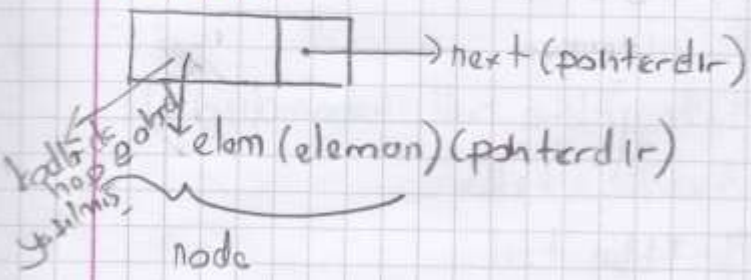
node: Her bir eleman

remove: Eleman silmek

v→ : v'nin işaret ettiği (*v) ile aynı

"Or" v→next

v'nin işaret ettiği bağlı liste elemanın next pointerle işaret ettiği



BAĞLI LİSTELERİN DİZİLERE GÖRE AVANTAJI

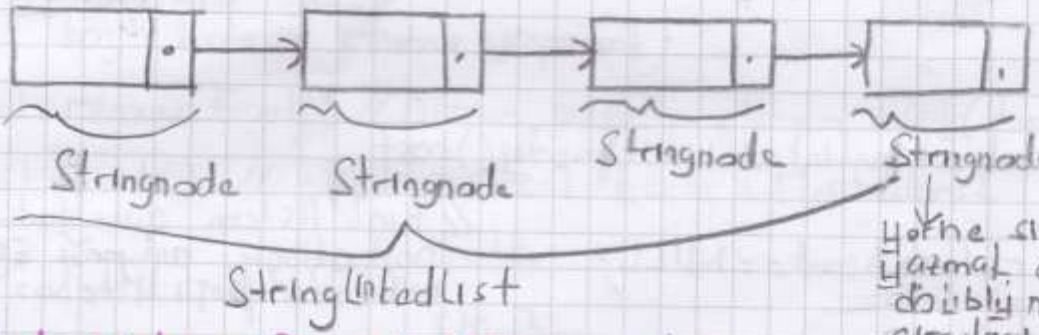
Dizilerde boyut değiştirmek, eleman eklemek, eleman silmek zordur. Dizilerde tüm elemanlar için hafızada yer ayırırlar.

1) TEK YÖNLÜ BAĞLI LİSTELER (SINGLY LINKED LİST)

next pointer null ise herhangi bir işaret etmiyor demektir.

head: Bağlı listenin ilk elemanını işaret ediyor

Bağlı Listenin Başından Eleman Silmek



Yerine singlynode yazmak daha mantıklı

doubly node

circulary node

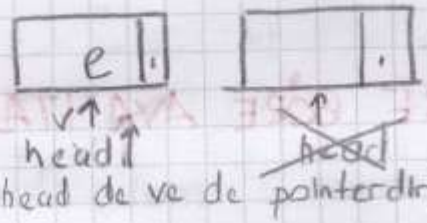
KOD!

Bağlı Listenin Başında Eleman silmek

```
void StringLinkedList::remove front
{   String *old = head; // headi silmek için headi pointerını başka bir pointere atıyoruz.
    head = old->next;
    delete old;
}
```

Bağlı Listenin; Başına Eleman Eklemek

```
void StringLinkedList::addfront(const string& e)
{   StringNode *v = new StringNode;
    v->elem = e;
    v->next = head;
    head = v;
}
```



(head de ve de pointerdır)

```
StringLinkedList::StringLinkedList()
  : head(NULL) { }  // head null yaptık
```

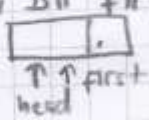
başlangıçta constructor.

```
StringLinkedList::~StringLinkedList()   destructor
{   while(!empty()) removefront();      // en sonunda elemanları
}                                       bellekten kaldırıyor

bool StringLinkedList::empty() const
{   return head == NULL;                // Bağlı listenin dolu yada
}                                       boş olduğunu anlıyoruz. Eğer head
                                        null eşitse bağlı liste boş

const string& StringLinkedList::front() const
{   return head->elem;  // elemanı ekrana head'den sonra
}                          eklediğimiz için head pointerının
                           gösterdiği here elemanı ekliyor

void StringLinkedList::print()
{   stringNode *first = head  // head pointera yeni bir first
                              pointer atıyoruz
```



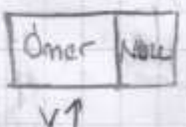

```
while (first != NULL) // first sıfırı gösterdiğinde
                      bağlı listenin sonu demektir.
{   cout << first->elem << endl; // first'ten (first'in işaret ettiği
    first = first->next;          aslında head'in işaret ettiği elemandır)
                                  başlayarak ve her seferinde first'i
```

SONUÇ: İlk yazdığımız eleman bağlı listenin en sonunda olduğundan en son çıktı olarak o eleman en son yazdığımız eleman bağlı listenin en başı olduğundan ekran çıktısı olarak ilk onu göreceğiz.

Uygulamada: list.addFront("Ömer"); // Listeye ilk Ömeri
list.addFront("Oğuzhan"); // İkinci Oğuzhanı
list.addFront("Fatih"); // Üçüncü Fatihi
list.addFront("AliOsman); // En son AliOsmanı eklediğip
list.print(); // print yaparsak
ekran çıktısı olarak AliOsman
Fatih.
Oğuzhan
Ömeri görürüz

Bu bağlı listeye remove yaparsak;
list.removeFront(); // Dizinin başından eleman sildik (Yani son eklenen eleman)
list print(); // print yaparsak ekran çıktısı olarak
Fatih
Oğuzhan
Ömeri görürüz.

ÖR: İlk eleman eklemek



Başlangıçta head pointerı nulla setlemiştik constructorsda. Boş bağlı liste oluşturuyoruz.
V pointeri bağlı listeye işaret ediyor.
head = V (headde v de pointer içinde adres tutuyor aynı adresi işaret ediyorlar. head neyi işaret ediyorsa v de ona işaret ediyor.)

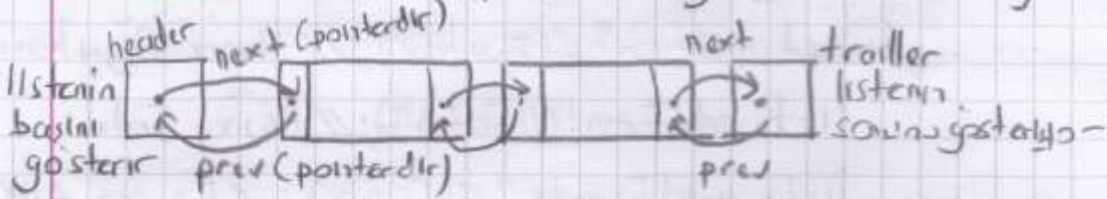
ÇİFT YÖNLÜ BAĞLI LİSTELER (DOUBLY LINKED LIST)

Hem kendisinden sonraki hemde kendisinden önceline pointer verdim.

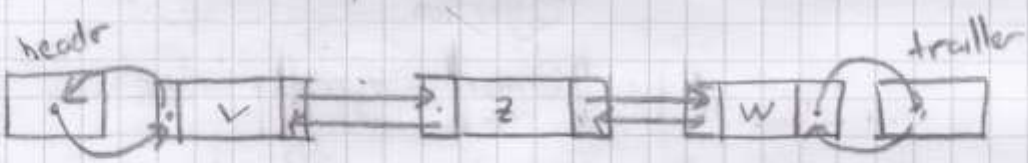
header: Bağlı listenin başını gösteriyor.

trailler: Listenin sonunu gösteriyor.

current: Listede aktif eleman bilgilerine ulaşabilceğimiz eleman



- Make z's prev. link point to v
- Make z's next lnt point to w
- Make w's next prev link point to z
- Make v's next lnL point to z



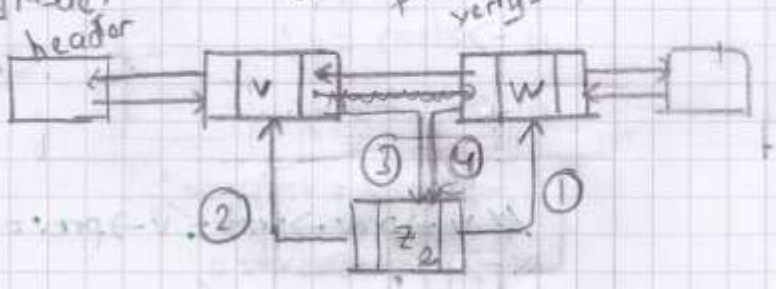
KOD 2

```
void DoublyLinkedList::addFront(const string& e)
{
    add(header->next, e); // header pointerının next pointerı
                          e yi gösteriyo.
}
void DoublyLinkedList::addBack(const string& e)
{
    add(trailler, e); // trailler pointerıda e yi gösteriyo
}
```

Eleman Eklemek

```
void DoublyLinkedList::add(DoublyNode *w, const string &e)
{
  DoublyNode *z = new DoublyNode;
  z->elem = e;
  z->next = w;
  z->prev = w->prev;
  w->prev->next = z;
  w->prev = z;
}
```

neden önce eklicek onu parametre olarak veriyoruz.

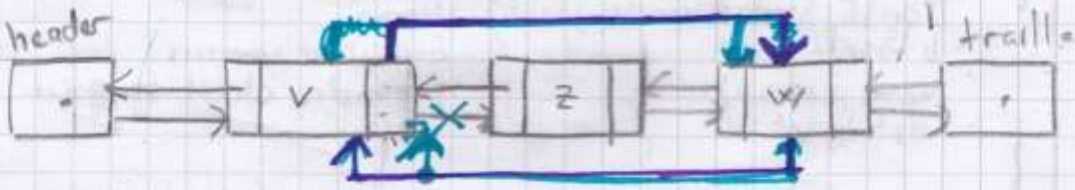


VERİ SILMEK

```
void DoublyLinkedList::remove(DoublyNode *z)
{
  DoublyNode *v = z->prev;
  DoublyNode *w = z->next;
  v->next = w;
  w->prev = v
  delete z;
}
```

silceğimiz elemanı parametre veriyoruz

Listeden eleman silerken silceğimiz elemanın öncesini ve sonrasını parametre olarak vermek zorundayız.

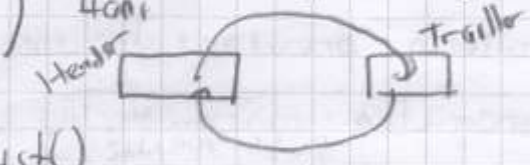


SINAV: Add le veri ekllerken ekleyeceğimiz veriyi parametre olarak verdiğimizden önce demiyoruz (Çünkü add kendinden önceki elemanı parametre olarak alıyor Bağlı listenin sonuna eleman ekllerken, traillerin previusu diyoruz (Traillerin previousu dersek ilki öncekini parametre olarak vermiş oluyoruz) addback

2. Kodun Devamı

```
void DoublyLinkedList::add(DoubleNode* v, const string& e) {   //neden önce ekleme yapacaksak ona parametre veriyoruz
  DoublyNode *u = new DoublyNode;
  u->elem = e;
  u->next = v;
  u->prev = v->prev;
  // v->prev->next = v->prev = u (CORRECT
  v->prev->next = u;
  v->prev = u;
}
void DoublyLinkedList::removeFront()
{
  remove(header->next);   // headerden sonraki elemanı siliyor.
}
void DoublyLinkedList::removeBack()
{
    remove(trailler->prev); // trailerden önceki elemanı siliyor.
}
void DoublyLinkedList::remove(DoublyNode* v)   // Silinecek elemanı
{  DoublyNode *u = v->prev; // Silinecek elemanın önceki ve sonraki
   DoublyNode* w = v->next; // elemanı parametre olarak veriyorum
   u->next = w;
   w->prev = v;
   delete v;
}
DoublyLinkedList::DoublyLinkedList()
{
   header = new DoublyNode;
   trailler = new DoublyNode;
   header->next = trailler;
   trailler->prev = header;
}
DoublyLinkedList::~DoublyLinkedList()
{
   while(!empty()) removeFront();
   delete header;
   delete trailler;
}
```

constructors
// header ve trailler pointerları yerleştiriliyor



//destructor

```
bool DoublyLinkedList::empty()const
{ return (header->next==trailler); // Listenin bos olup olmadigina bakiyor. Eger liste bossa header.in next'ine isaret etti pointer trailler pointeri olcak
}

const string &DoublyLinkedList::front()const
{   return header->next->elem // önden elemanı ekliyor, Yani headerin nextinden sonra
}
const string &DoublyLinkedList::back()const
{   return trailler->prev->elem; // sondan elemanı ekliyor yani trailler previousunu
}
void DoublyLinkedList::print()
{ DoublyNode *first= header;                    // headerden baslayarak elemanları ekrana basmaya baslıycak Burdada ilk elemanı sonda olcagından ekrandan en son gösterecek
  while (!(first->next==trailler))
  {
    cout<<first->next->elem<<endl;
        first=first->next;
  }
}
```

Uygulaması

Elemanları önden ellersek.

```
list.addFront("Ömer")
list.addFront("Öğrahan")
list.addFront("Fatih")
list.addFront("Ali Osman")
list.print(); (Ali Osman - Fatih - Ögrahan - Ömer)

list.removeFront();
list.print(); (Fatih - Ögrahan - Ömer)

list.removeBack()
list.print(); (Fatih - Öğrahan)
```

Elemanları geriden yani trailerden önce ellersek

```
list2.addBack("Ömer")
list2.addBack("Öğrahan")
list2.addBack("Fatih")
list2.addBack("Ali Osman")
list2.print(); (Ömer, Öğrahan, Fatih, Ali Osman)

list2.removeFront();
list2.print(); (Öğrahan, Fatih, Ali Osman)

list2.removeBack();
list2.print(); (Öğrahan, Fatih)
```

3) DAİRESEL BAĞLI LİSTELER (CİRCULARLY LINKED LISTS)

Tek yönlü bağlı listelerden farkı dizi elemanları sürekli birbirini işaret ediyor. Hem başı ve sonu belli değildir.

Cursor: Dizinin başı

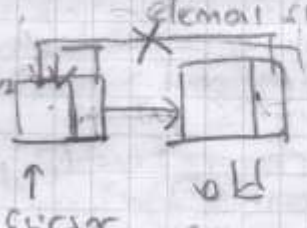
- Diziye veri eklersek cursordan sonra ekliyor.
- Diziden veri silersek cursorun işaret ettiğinden sonrakini siliyor.

Herhangi bir pointer null eşitse nesneye yada bir bağlı liste elemanına işaret etmiyor demektir.

İlk eklediğimiz veri dizinin başını gösteriyor ve o veri aynı zamanda cursor oluyor.

Kod

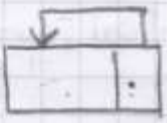
```
bool CircularyLinkedList::empty() const
{
    return cursor == NULL; // liste boş mu?
                           // eğer cursor null eşitse liste boş.
}
const string &CircularyLinkedList::back() const
{
    return cursor->elem;
}
const string &CircularyLinkedList::front() const
{
    return cursor->next->elem;
}
void CircularyLinkedList::advance()
{
    cursor = cursor->next;   // advance cursorın cursorun nexti işaret ettiği yapıyor.
}
void CircularyLinkedList::remove()
{
    CircularlyNode *old = cursor->next; // cursordan sonrayı
    if (old == cursor)
        cursor = NULL; // tek elemansa cursoru null yapıyoruz
    else
        cursor->next = old->next; //
    delete old;
}
```



cursorun nexti oldu işaret ettiğinden old'

```
CircularlyLinkedList::CircularlyLinkedList()
 :cursor(NULL) { }                    // constructor
CircularlyLinkedList::~CircularlyLinkedList()
{  while(!empty()) remove() // destructor
}

void CircularlyLinkedList::add(const string& e)
{  CircularlyNode *v=new CircularlyNode;
   v->elem=e;
   if (cursor==NULL)
{  v->next=v;          //
   cursor=v;
}
 else
{  v->next=cursor->next
   cursor->next=v
}
}
```



Eğer liste boşsa ilk elemanı ekliyorsak demektir. Başta bir eleman olmadığı için kendini işaret ede

cursorun next'i v'yi işaret ettiği için v'nin next'ini cursora getiriyoruz

advance() yaparsak

!!

Ömer - Oğuzhan - Fatih - A.Osm.

① cursor ② cursor ③

```
void CircularlyLinkedList::print()
{  CircularlyNode* first=cursor;
   while(!(first->next==cursor))
   {  cout<<first->elem<<endl;
      first=first->next;
   }
   cout<<first->elem<<endl;
}
```

çıktısı: A.Osman
Ömer
Oğuzhan
Fatih

!! advance eleman ile de yapsak aynı oktı çıktısı

Uygulamasında

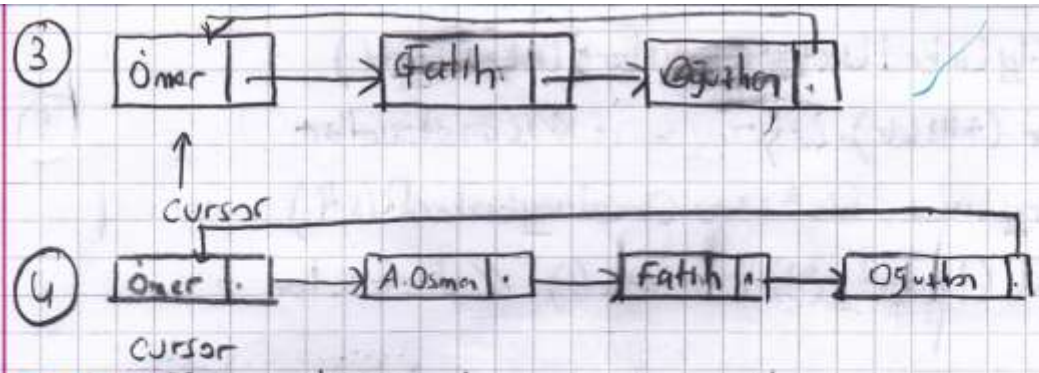
```
list.add("Ömer")
list.add("Oğuzhan")
list.add("Fatih")
list.add("Ali Osman")
```

list.print() (Ömer - Ali Osman - Fatih - Oğuzhan)

①

②

// ilk durumda cursorun next'i Ömer'i gösterdiğinden yani next'ini Ömer'e getirdik



Cursor: Cursorun bir sonraki cursor yapıyor. Yani cursorun next'i cursor. Cursord eleman sona gelyor

RECURSION

Fonksiyonun kendini başka bir parametreyle çağırması

1-Lineer Recursion: fonksiyon kendini yanlızca 1 noktada çağırıyo

2-Binary Recursion: fonksiyon kendini 2 noktada çağırıyor

3-Multiple Recursion fonksiyon kendini 3 veya daha fazla noktada çağırıyor.

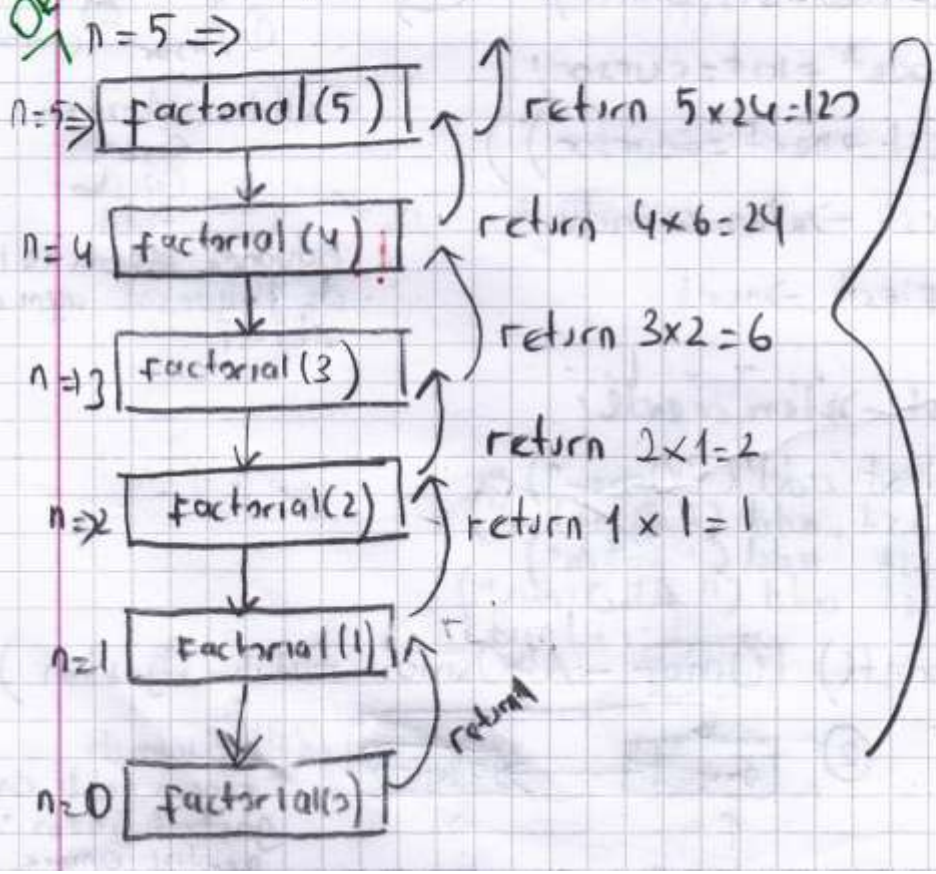
Recursive Fatorial

```
int factorial(int n)
{ if(n==0) return 1; // base case (Recursive fonksiyondan çıkmak i
  else return n*factorial(n-1)//recursive case (Recursive fonksiyon
                                                 kendisi)
}
```

return yapılan yer

!! recursive font returnların nereye yapıldığı önemlidir

Ör:



ÖR

A = {1, 2, 3, 4, 5, 6, 7}
indis 0 1 2 3 4 5 6

void reverseArray (int A[], int i, int j) //

↑ A diye de gösterilebilir

{ if (i<j)
int temp = A[j]
A[j] = A[i];
A[i] = temp;
return reverseArray (A, i+1, j-1)

reverse (A, 0, 6)
A = {7, 2, 3, 4, 5, 6, 1}
reverse (A, 1, 5)
A = {7, 6, 3, 4, 5, 2, 1}
reverse (A, 2, 4)
A = {7, 6, 5, 4, 3, 2, 1}

A (3, 3, 3) → koşulu sağlamı yor.

SINAV

int f (int k)
{ if (k==1) return 1;
if (k==2) return 3;
return f(k-1) + f(k-2) + k
}
void main()
{
printf ("f (olod) = olod", 10, f(10));

k=10 ⇒ f(9) + f(8) + 10 f(10) = 364
k=9 ⇒ f(8) + f(7) + 9 f(9) = 221
k=8 ⇒ f(7) + f(6) + 8 f(8) = 133
k=7 ⇒ f(6) + f(5) + 7 f(7) = 79
k=6 ⇒ f(5) + f(4) + 6 f(6) = 46
k=5 ⇒ f(4) + f(3) + 5 f(5) = 26
k=4 ⇒ f(3) + f(2) + 4 f(4) = 14
k=3 ⇒ f(2) + f(1) + 3 f(3) = 7
↓ 3 ↓ 1

ÖR

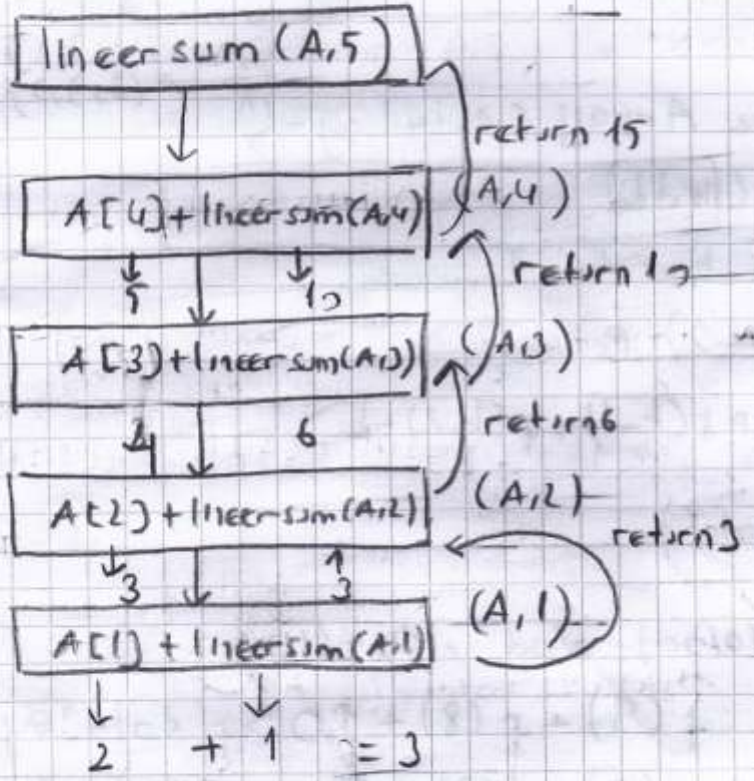
int sum (int n)
{ if (n==1) return 1;
else return n + sum(n-1)
}

1'den n'e kadar sayıların toplamı

Örnek // Birden n'e kadar olan sayıların toplamının dizide tutularak yapılmış

A = {1, 2, 3, 4, 5}

```
int lineer sum (int A[], int n)  ---> (int A, int n)
{  if (n == 1) return A[0]
   else return A[n-1] + lineer sum (A, n-1)
}
```



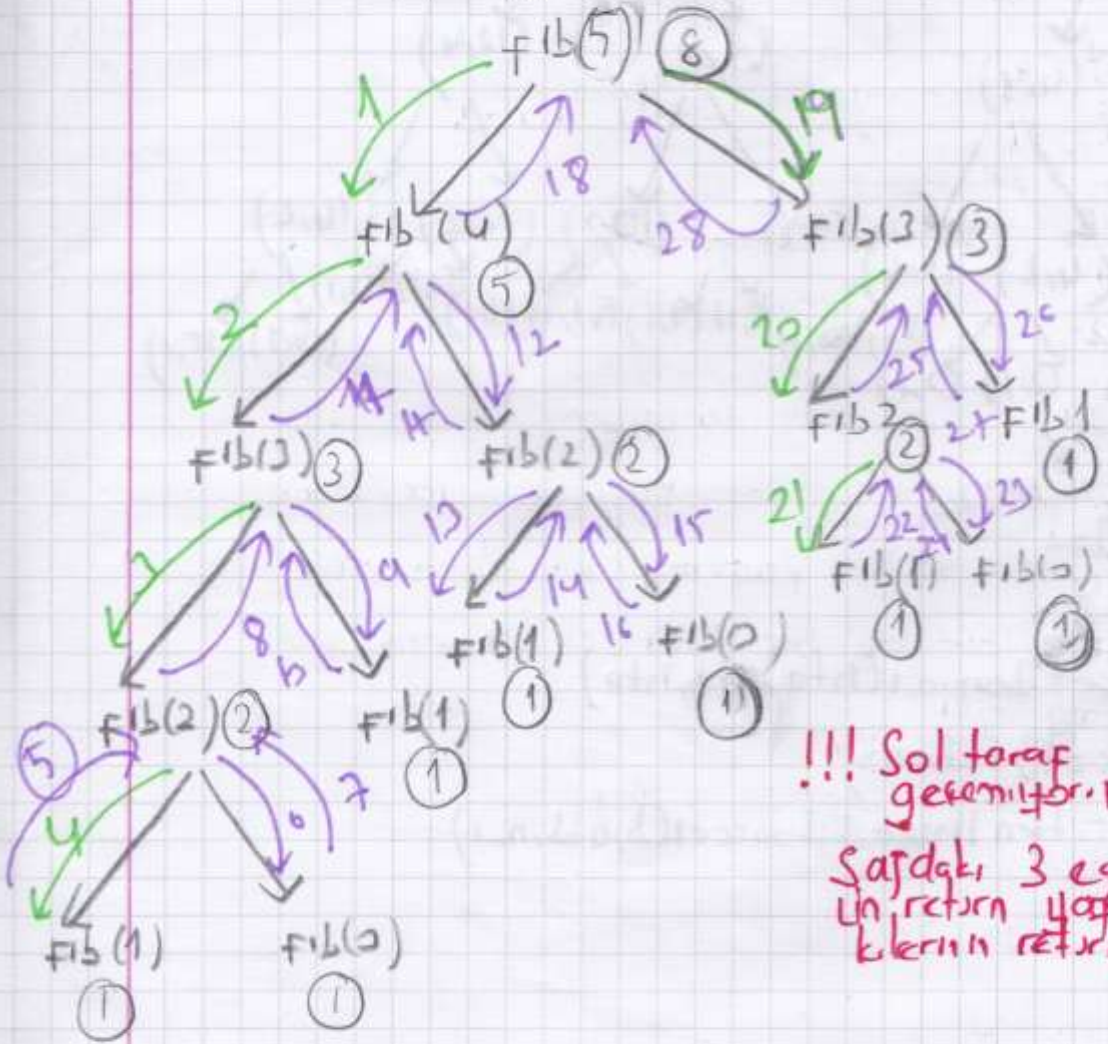
!! Tail Recursion: Recursif fonksiyonu iteratif haline dönüştürülmüş

Lineer recursion fonk iteratife çevrilebilir Recursif fonksiyonu iteratife çevrilmesi için çağrının en dipte olması gerekir. Ve return'le birlikte bir değeri ne çarpcaz ne topluycaz. reverse array gibi...

Örn reverse array'in iteratife çevrilmiş hali

```
while (i<j)   --> (1) if döngüsünü while yapıyoruz
int temp = A[j];
A[j] = A[i];
A[i] = temp;
i = i+1;   } (2) parametre olarak yolladığımızı
j = -1     } bu şekilde yapıyoruz
```

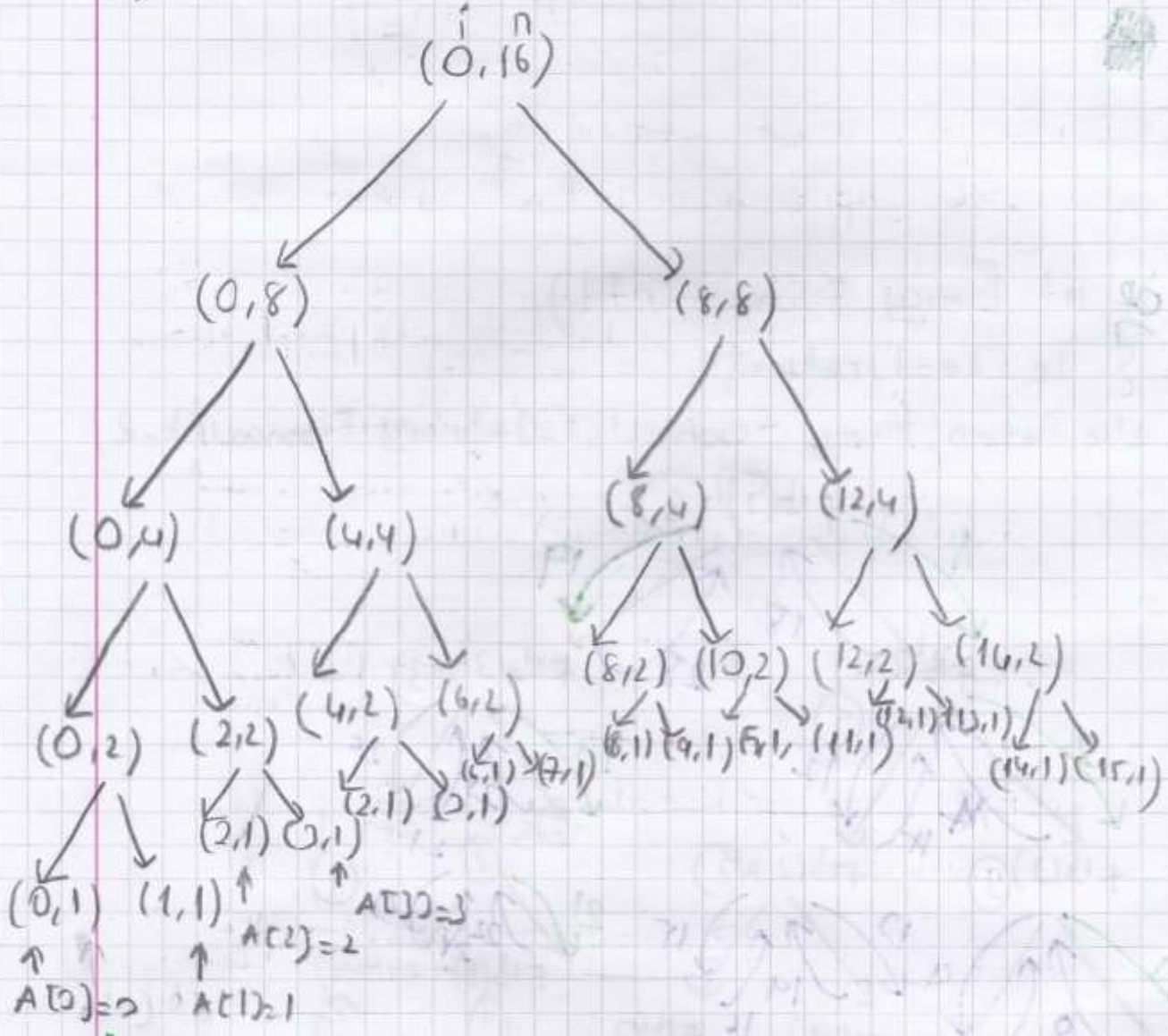
Ör int binary Fibonacci (int k)
{ if (k==1) return 1;
else return binary Fibonacci (k-1) + binary Fibonacci (k-2)



!!! Sol taraf bitmeden sağ taraf gerçekleşmiyor.

Sağdaki 3 gerçekleşmesi için 4 'ün return yapabilmesi ona altta-kilerin return yapması gerekiyor

Q. int binarysum(int A[], int i, int n)
{ if (n==1) return A[i] A={0,1,2 — 16}
else return binarysum(A, i, n/2)+binarysum(A, i+n/2, n/2)
}



Q. int linearFibonacci(int a, int b, int n)
{ if (n<=1) return b;
else return linearFibonacci(b, a+b, n-1)
}

1. Vize

1) header->next->next->prev=header; (1)

trailer->prev->next = header next; (2)

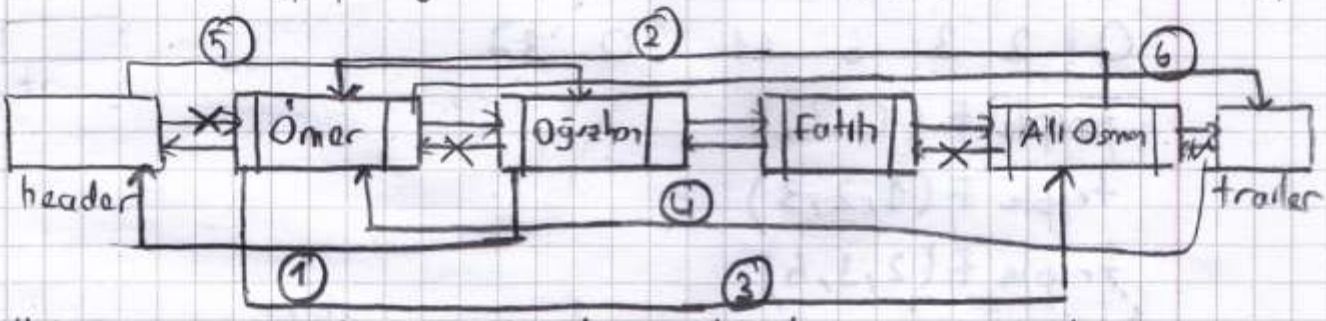
header->next->prev = trailer->prev; (3)

trailer->prev = header next; (4)

header->next = header->next->next; (5)

trailer->prev->next = trailer (6)

add back yapildigindan listenin sonundan eleman eklenliyor.



!! ilk durumda headerin nexti Ömer, trailerin prev Ali Osman

(5) trailerin prev artik Ömer

2) Recursif yapilmis listenin son elemanina kadar gidiyor. Son elemanini basiliyor.

mavi(header)

↓

node=header (headeri node olarak aliyoruz)

if calismaz direk else

mavi(header) node->header

↓

mavi(Ömer) node->Ömer

↓

mavi(Öğretmen) node->Öğretmen

↓

mavi(Fatih) node->Fatih

↓

mavi(Ali Osman) node->Ali Osman

cout<<node->elem

↓

3) linear fib

0 1 1 2 3 5 8 13 21 24

linear F(0,1)

linear F(1,1)

linear F(1,2)

(2,3)

(b, a+b)

0 1 2 3 6 11 20 37

triple F(0,1,2)

triple F(1,2,3)

triple F(2,3,6)

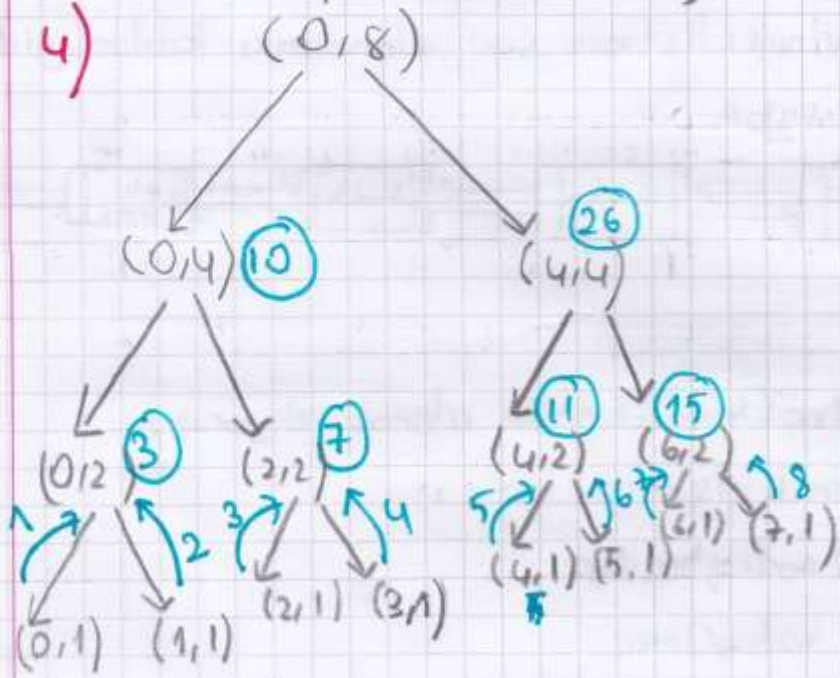
triple F(3,6,11)

triple F(6,11,20)

triple F(11,20,37)

triple F(b,c, a+b+c)

4)



sum=3
sum=7
sum=10
sum=11
sum=15
sum=26
sum=36

sum = binary sum(A, i, n/2) + binary sum(A, i+n/2, n/2)

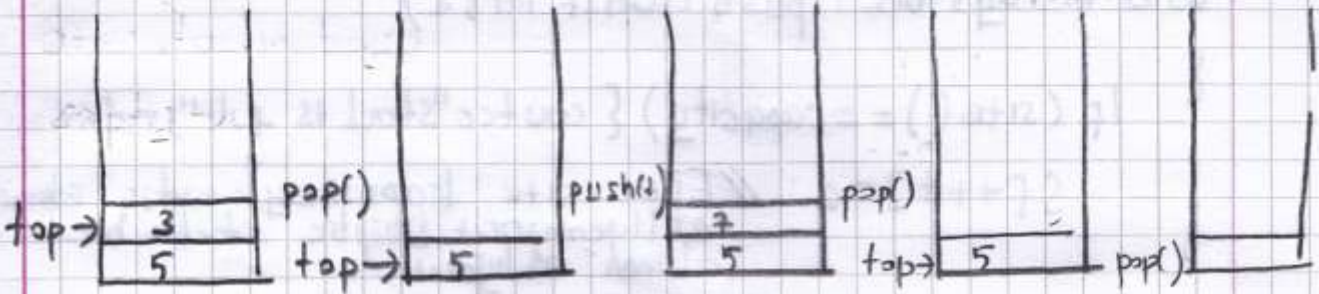
Chapter 5

stacks-last in first out (Son giren ilk çıkar) LIFO

2 işlem vardır push() → itiyor

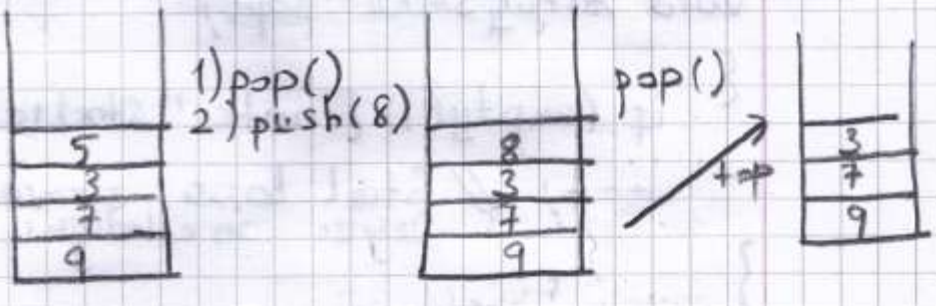
pop() → stacden eleman çekiyoruz.

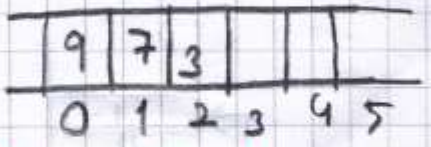
top() → stackin tepesini görüyoruz.



SINAV

1) push(9)
2) push(7)
3) push(3)
4) push(5)





Array Stack → Yığın

- pop --t t'yi bir azaltıyor
- push ++t t'yi bir arttırıyor.

Dizilere Göre Avantajı

Bağlı liste dinamik yapıdadır. Program kaynaklarken eleman ekleyip silebiliyoruz Dizilerde bu yok çünkü başta eleman sayısını verip sabitliyoruz.

Kod!

```
ArrayStack::ArrayStack(int cap)
  :S(new int[cap]), capacity(cap, (t-1) //constructor
int ArrayStack::size() const
{
  return t+1; //Yığındaki öğe sayısını gösteriyor
}
bool ArrayStack::empty()const
{
  return (t<0)   // Başlangıçta t=-1 di. İlk elemanı ek-
                    lerken t=0 oluyor Eğer t<0 sa
```

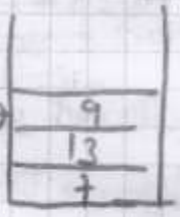
```
const int& ArrayStack::top()
{ if (empty()) { cout<<"Stack is empty"; return 0;}
   return s[t]      // Eğer stack boşsa stackin boş
                    olduğunu belirtsin değilse stackin
                    basındaki elemanı basacak
}
void ArrayStack::push(const int& e)
{
   if (size()==capacity) { cout<<"Stack is full"; return;}
      s[++t]=e   // Eğer size kapasiteye eşitse eleman
                    ekliyemiyoruz. Değilse stackin basına
                    eleman ekliyoruz.
}
void ArrayStack::pop()
{
   if (empty()) {cout<<"Stack is empty"; return;}
    --t; // Stack boşsa, eleman silemiyoruz
            Değilse son eklediğimiz elemanı siliyoruz
}
```

Uygulamasında:

```
A.push(7);
A.push(13);
A.push(1)
cout<<A.top()<<endl;
```

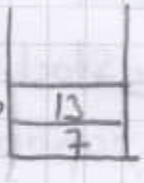
1

```
A.pop();
A.push(9);
cout<<A.top()<<endl;
```



9

```
A.pop()
cout<<A.top()<<endl;
  getchar();
}
```



13

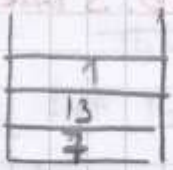
LINKED STACK
Bağlantılı

Stackleri tek yönlü bağlı listede gösterebiliyoruz.
Dizi yerine bağlantılı liste var. n bağlantılı liste eleman sayısı n=0 sa stack boş
Top fonksiyonu dizinin tepesindeki elemanı gösteriyordu. Burda da stackin başını (en son eklediğimiz elemanı yani head'i) gösteriyo

Kod

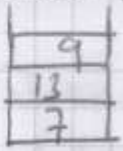
```
LinkedStack::LinkedStack() : s(), n(0)
{// constructor
}
int LinkedStack::size()const
{
  return n;  // n 0 dan başlamıştı yığındaki öğe sayısını gösteriyor.
}
bool LinkedStack::empty()const
{  return n==0;  // n=0 sa stack boş
}
const int8 LinkedStack::top()const
{ if (empty()) { cout<<"Top is empty stack"; return -1;}
   return S.front();  // Yığının başındaki elemanı (en son eklediğimiz elemanı gösteriyor, yığın boşsa boş olduğunu ekrana basıyor
}
void LinkedStack::push(const int8 e)
{  ++n;                 // ? n başlangıçta 0 dı 1 e mi eşitliyor
   S.Addfront(e);  // Stackin başına elemanı ekliyor.
}
void LinkedStack::pop()
{ if(empty()) { cout<<"Pop from empty stack"; return;}
   --n;
   S.removeFront();  // Stackin başındaki elemanı siliyor
}
```

Uygulamasında; LStack.push(7);
LStack.push(13);
LStack.push(1);



cout<< "Top=" << LStack.top() << endl; 1

LStack.pop();

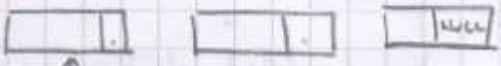


LStack.push(9);
cout<< "Top=" << LStack.top() << endl; 9

LStack.pop();

cout<< "Top=" << LStack.top() << endl; 13
::getchar();
}

TekBaşına Bağlantılı Liste (Singly Linked List)




```
void SinglyLinkedList::removeFront()
{ SinglyNode* old=head;
  head=old->next;
  delete old;
}
void SinglyLinkedList::addFront(const int& e){
 SinglyNode* v=new SinglyNode;
   v->elem=e;
   v->next=head;
    head=v;
}
SinglyLinkedList::SinglyLinkedList() //constructor
:head(NULL){}
SinglyLinkedList::~SinglyLinkedList()
{ while(!empty()) removeFront(); // destructor
}
bool SinglyLinkedList::empty()const
{ return head==NULL; //bağlantılı liste boş mu
}
```

```
const int& SinglyLinkedList::front() const
{ return head->elem;
}
```

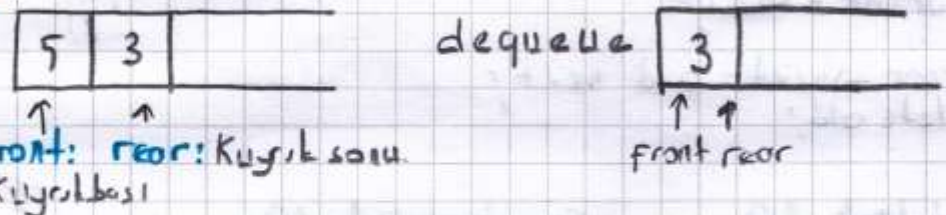
BAĞLANTILI KUYRUK YAPISI

Queues: Kuyruk FIFO (first in first out) mantığına göre çalışır. Kuyruğa veri ekleyen kuyruk sonundan ekliyoruz, veri silerken kuyruğun başından siliyoruz. Banka kuyruğu gibi düşün.

enqueue: Kuyruğa veri ekleme

dequeue: Kuyruktan veri silme

ÖR



Burdada dairesel bağlı liste kullanıyoruz.

Dairesel bağlı listede cursor listenin başını işaret ediyordu. Burda kuyruğun sonunu işaret ediyor

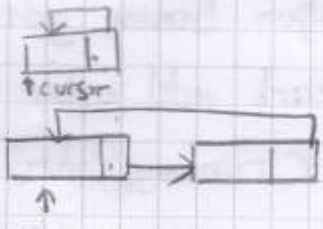
Son elemanın nexti kuyruğun başı (Eleman silerken kuyruğun başından siliyoduk)

TEK FARK!! Dairesel bağlı listeye eleman eklerken cursordan sonra ekliyodu. Cursor ilk eklediğimiz eleman oluyodu. Hep sabitti. Burda cursor eleman ekledikçe değişiyor, Ve son eleman oluyor. Silerkende cursorun nextini siliyodu. Burdada aynı cursorun nexti kuyruğun başını gösterdiğinden kuyruğun başını silmiş oluyoruz. Hani tek fark add e bir ADVANCE ekleniyor

KOD

```
bool CircularlyLinkedQueue::empty() const
{   return cursor == NULL // kuyruk boş mu?
}
const string& CircularlyLinkedQueue::back() const
{ return cursor->elem; // ilk cursor a elem değeri veriyor.
}
```

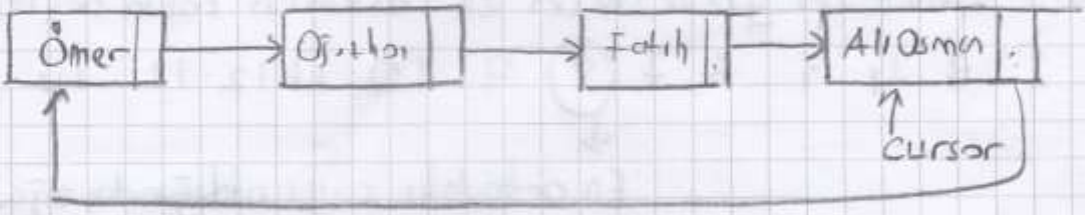
```
const string& CircularlyLinkedQueue::front() const
{
  return cursor->next->elem; // Sonra cursorun nextine elem
}                              degeri veriliyor.
void CircularlyLinkedQueue::advance()
{
  cursor = cursor->next;
}
void CircularlyLinkedQueue::remove()
  CircularyNode* old=cursor->next;
   if(old==cursor)
      cursor = NULL;
   else
     cursor->next = old next;
     delete old;
}
CircularlyLinkedQueue::CircularlyLinkedQueue()
  :cursor(NULL) { } // constructor
CircularlyLinkedQueue::~CircularlyLinkedQueue(){
 while (!empty()) remove(); }
void CircularlyLinkedQueue::add(const string& e)
{ CircularlyNode *v = new CircularyNode;
    v->elem=e;
  if(cursor==NULL)
  {
     v->next=v;
     cursor=v
  }
  else
{
  v->next=cursor->next
  cursor->next=v;
}
   advance(); → !! dairesel bagli listeyle farki
}
void CircularlyLinkedQueue::print()
{ CircularlyNode* first=cursor->next // cursorun next'i diyerek
```



```
while (!(first==cursor))
{ cout<<first->elem<<endl;
   first = first->next;
}
  cout<<first->elem<<endl;
{
```

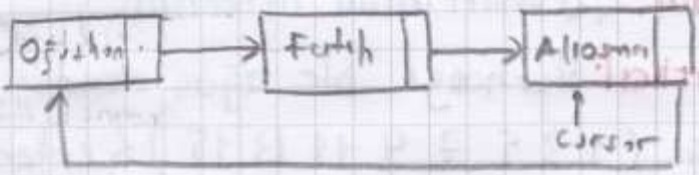
Uygulaması

```
//FIFO mantığına göre çalışır.
//Ekleme kuyruk sonuna(rear), silme kuyruk başından(front) yapılır.
list.add("Ömer");
list.add("Oğuzhan");
list.add("fatih");
list.add("Ali Osman");
list.print();
```



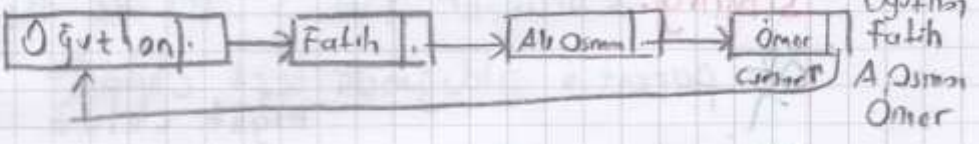
Ömer
Oğuzhan
Fatih
Ali Osman

```
list.remove()
list.print();
```



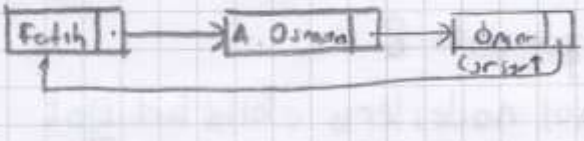
Oğuzhan
Fatih
Ali Osman

```
list.add("Ömer");
list.print();
```



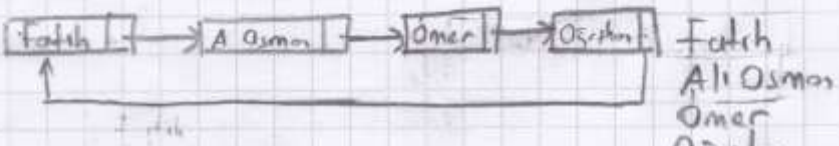
Oğuzhan
Fatih
Ali Osman
Ömer

```
list.remove();
list.print();
```



fatih
Ali Osman
Ömer

```
list.add("Oğuzhan");
list.print();
```



Fatih
Ali Osman
Ömer

BİNARY TREES: İki tane aşağı doğru link çıktığında binary

8 → root (kök) Ağacın en tepesindeki eleman

4 !!! parametre olarak verdiğimiz elemandan küçük olabilir sola büyük olabilir sağa

12

2

6

10

14

1

3

5

7

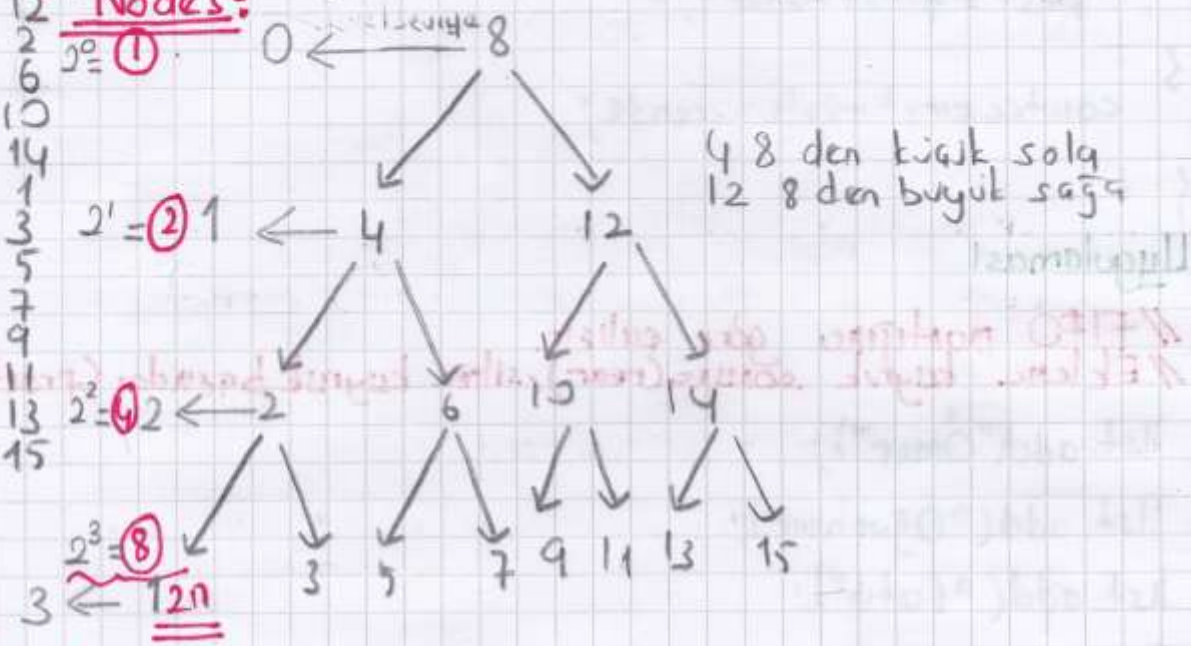
9

11

13

15

Nodes:



level seviye: rootun olduğu seviye

nodes: Ağacın her elemanı

Ağaca elemanları yerleştirken en önemlisi kökün seçilmiştir.

1 2 3 4 5 6 7 (8) 9 10 11 12 13 14 15

En ortadaki sayı olduğundan ağaç dengeli

parent: 8 parent olduğunda

child: 4 ve 12 childleri (4 left child 12 right child)

internal: Çocukları olan internal

external: Herhangi bir ağaç elemanının left ve right elemanı yoksa (1 3 5 7 9 11 13 15) } external nodes. Hem childleri null

sibling: Kardeşler 1 ve 3 10 ve 14 gibi...

ÖR parent 6 olduğunda left child 5
right child 7

ÖR rootun parenti null
4 ün parenti 8

!!! external nodeların childleri yok childleri null

int elt;

↳ 1 den 15 e kadar olan rakamlar elt değişkeninde tutuluyor.

KOD

```
LinkedBinaryTree::LinkedBinaryTree()
    :_root(NULL), n=0 { } // constructor

int LinkedBinaryTree::size()const
{
    return n        // Ağacın kaçıncı elemanı
}

bool LinkedBinaryTree::addRoot()
{
    _root = new Node        // root ağacın kökünü elemanı
      n=1                   ekliyor.
}

void LinkedBinaryTree::expandExternal(Node *v)
{ v->left = new Node;   } ① v nin sola eleman eklemek için v yi parametre verdik
  v->left->par = v;     }
  v->right = new Node;  } ② v nin sağına eleman eklemek için v yi parametre verdik.
  v->right->par = v;    }
  n+=2;
}
```

!! Hangi elemanın sağına ve soluna eleman ekliyecekse parametre olarak onu verecek.

Uygulamasında; LinkedBinaryTree nuTree;

① nuTree.addRoot();
nuTree._root->elt=8;

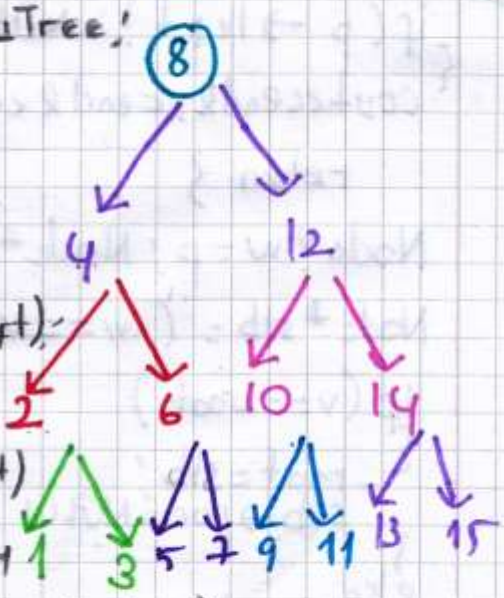
② nuTree.expandExternal(nuTree._root)
nuTree._root->left->elt=4;
nuTree._root->right->elt=12;

③ nuTree.expandExternal(nuTree._root->left);
nuTree._root->left->left->elt=2;
nuTree._root->left->right->elt=6;

④ nuTree.expandExternal(nuTree._root->right)
nuTree._root->right->left->elt=10;
nuTree._root->right->right->elt=14;

⑤ nuTree.expandExternal(nuTree._root->left->left)
nuTree._root->left->left->left->elt=1;
nuTree._root->left->left->right->elt=3;

⑥ nuTree.expandExternal(nuTree._root->left->right)



(7) nuTree.expandExternal(nuTree._root→right→left);

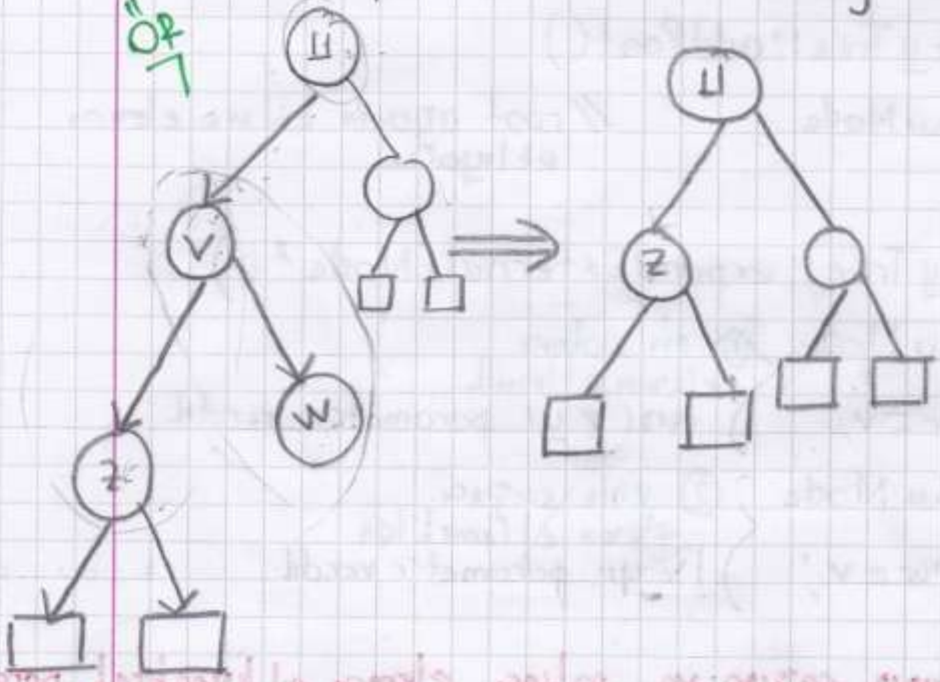
nuTree._root→right→left→left→elt=9;
nuTree._root→right→left→righ→elt=11;

(8) nuTree.expandExternal(nuTree._root→right→right);

nuTree._root→right→right→left→elt=13;
nuTree._root→right→right→right→elt=15;

remove yaparken: Parametre olarak external vermek zorundayız. (Yani left ve right childı olmayan ağaç elemanı) silerkende parametre olarak verdiğimizi ve babasını siliyor.

ÖR 7



KOD
```
void LinkedBinaryTree::removeAboveExternal(Node *p)
if(p→left != NULL || p→right != NULL) // Çocuğu var mı yani external olmasına bakıyor
{ cout<<endl<<endl<<"This node is not external!";
    return }
Node *w = p; Node *v = w→par; // silinecek elemanları kendisini ve parentını belirttik.
Node *sib = (w == v→left ? v→right : v→left) // w v nin lefti ise w nın kardeşi v nin righti değilse lefti
if (v == _root)
{ _root = sib;
  sib→par = NULL; // kardeşini root yapıyor
}
else
{   Node *gpar = v→par; // v nin parent w's grandparenti
    if (v == gpar→left) // v, w grandparentinin lefti ise
    gpar→left = sib;     grandparentın lefti w sibling
    → v == gpar→right
```

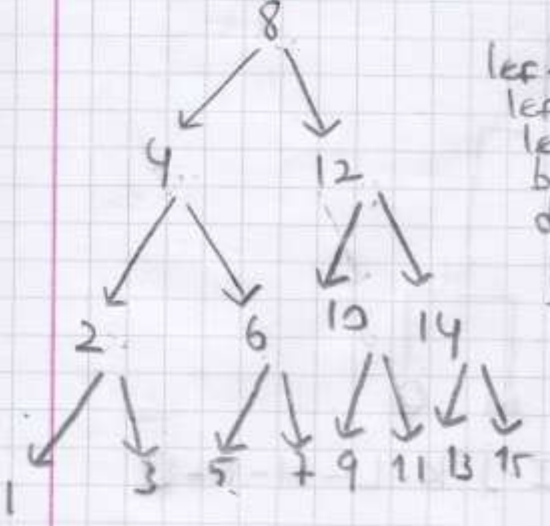
```
slb -> par = gpar; // w nın kardeşlerinin parenti w nın grand
                      parenti old.
}
delete w; delete v;
n = -2;
}
```

InOrder: L P R → Left Parent Right

preorder: P . L R → Parent Left Right

postorder: L R P → Left Right Parent

PREORDER: (PLR) 8 4 2 1 3 6 5 7 12 10 9 11 14 13 15



(8) parenti parenti bastı sonra parentin leftini (4) o aynı zamanda parent, parentın lefti (2), o aynı zamanda parent, parentın leftini (1), parentın rightını (3), parentı, leftini bastı, parentın rightını basacak (6) o aynı zamanda parent parentın leftini basacak (5), parentı rightını basacak (7), parentı bastı (8) leftini bastı (4) parentın rightını basacak 12 o aynı zamanda parent, parentın leftini basacak 10 o aynı zamanda parent, parentın leftini basacak 9, parentın rightını basacak 11, parentı bastı (12) parentın leftini bastı 10 rightını basacak (14) o aynı zamanda parent, parentın leftini basacak 13 rightını basacak 15

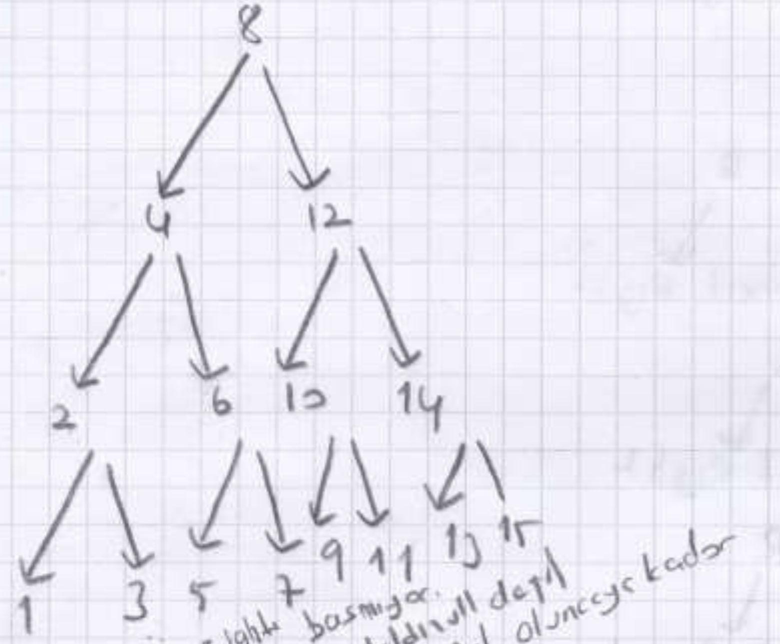
8-4-2-1-3-6-5-7-12-10-9-11-14-13-15

INORDER: (LPR) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

leftin child null olana kadar

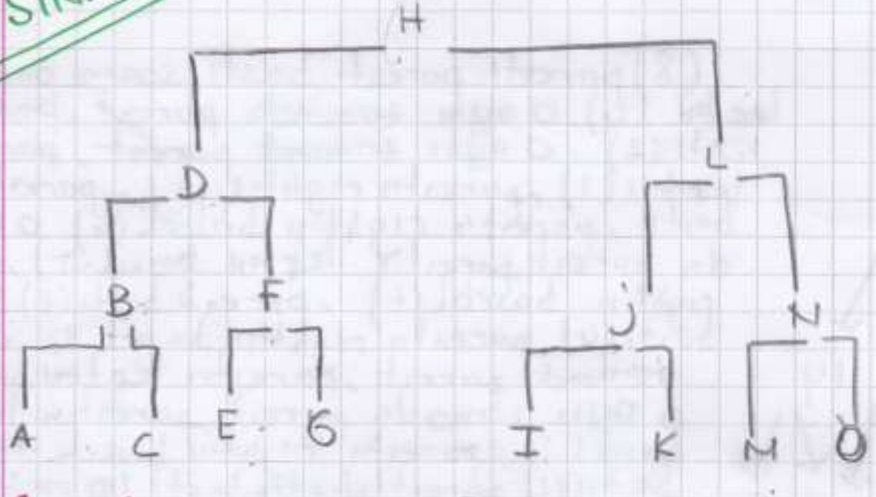
ilk en son lefte kadar gidecek (1) i basacak sonra 1 in parentını (2) yi basacak sonra sonra righti (3) ü basacak, lefti bastı parentı basacak (4) sonra righti basacak (6) o aynı zamanda parent parentı rightını basacak 7 lefti bastı parentı basacak (8), en sona kadar inecek (9) basacak parentını basacak (10) rightını basacak (11), lefti bastı parentını basacak 12 righti basacak en dibe gelecek 13 basacak parentı basacak 14 righti basacak 15

Postorder:
(Left, Right, Parent)



1 - 3 - 2 - 5 - 7 - 6 - 4 - 9 - 11 - 10 - 13 - 15 - 14 - 12 - 8

SINAV

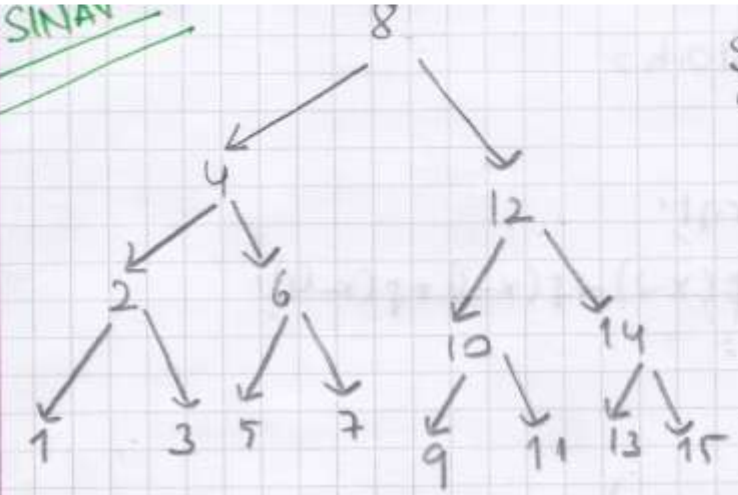


Inorder: A - B - C - D - E - F - G - I - J - K - L - M - N - O
(LPR)

Preorder: H - D - B - A - C - F - E - G - L - J - I - K - N - M - O
(PLR)

Postorder: A - C - B - E - G - F - D - I - K - J - M - O - N - L - H
(LRP)

SINAV



Silinecek eleman CHILD
CHILDIN babası PARENT

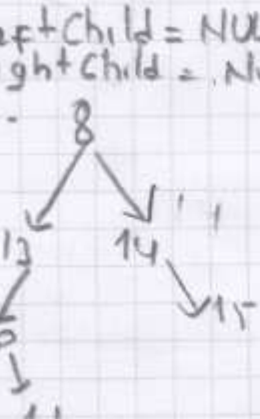
12 değerine sahip eleman bu koda göre silindiğinde ağacın son hali ne olur.

```
if( CHILD ->leftChild != NULL && CHILD->rightChild != NULL
{
  if (PARENT ->leftChild == CHILD)
  { temp = CHILD ->leftChild;
    PARENT ->leftChild = CHILD->leftChild;
    while (temp ->rightChild != NULL)
        temp = temp ->rightChild;
    temp ->rightChild = CHILD->rightChild;
     CHILD ->leftChild = NULL;
     CHILD ->rightChild = NULL;
  }
  else
     temp = CHILD ->rightChild;
     PARENT ->rightChild == CHILD ->rightchild
     while (temp ->leftchild != NULL)
          temp = temp ->left child;
      temp ->leftChild = CHILD ->leftChild;
          CHILD ->leftChild = NULL;
          CHILD ->rightChild = NULL;
```

12 PARENTın right childı olduğundan buraya girmez

(Parentın right childı 14 olur)

⟹



(A)

Priority Queues

Öncelik

Yine bir kuyruk var yanlız kuyruktan batılarının önceliği var

Priority Queues

enqueue: Eleman eklemek (Kuyruk sonuna eklenecek) — Farklı (uygun yer neresiyse)

dequeue: Eleman silmek (Kuyruk başından siliyoruz) — Aynı

Kuyruk elemanlarını sıralı tutuyoruz. Eklerken kuyrukta o sırada uygun yer neresiyse oraya ekliyoruz.

Verileri sıralı tutarken sürekli karşılaşma yapacak. Bu karşılaşmayı yapmalıyız **isless** fonksiyonunu kullanıyoruz.

insert: islessi çağırarak veri eklemesini sağlıyor.

std::list<int>::iterator p; Liste elemanlarını indisliyor (çift yönlü bağlı liste) (Doubly Linked List) Bir pointer — p yi arttırarak o dizideki elemanlara erişebiliyoruz.

std::list<int>L; diye çift yönlü bağlı liste üretiyor !!!!

insert: Elemanları eklerken sıralı ekliyor. Parametre olarak verdiğimiz indisten önce ekliyor.

remove min: remove front gibi çalışacak. Kuyruğun ilk elemanını silecek (Dequeue)

KOD

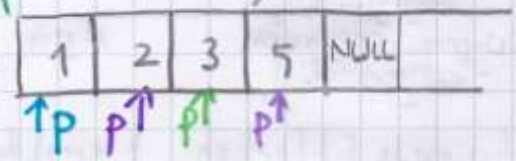
```
int ListPriorityQueue::size()const
{
    return L.size();
}
bool ListPriorityQueue::empty()const
{
    return L.empty()
}
bool ListPriorityQueue::isless(const int& e, const int& f) const{
    if(e<f)return true;
    else return false;
}
void ListPriorityQueue::Insert(const int& e)
{ std::list<int>::iterator p;
  p=L.begin(); //her seferinde dizinin başından başlasın
```

(mevcut listeye eklemek istediğimiz eleman)

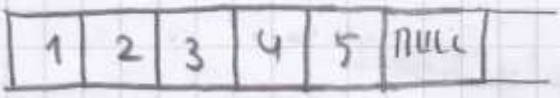
while(p!=L.end()&&!sless(e,*p)) ++p;
↓
dizinin son elemanı olmadığı sürece

L.insert(p,e);}

Or insert 4;



- p pointeri 1 i gösteriyor. 4 : 1 den küçük değil false döndürecek değil true ve 1 Listenin son elemanı değil p yi 1 artıracak
- p pointeri 2 yi gösteriyor 4 2 den küçük değil false döndürecek değil true ve 2 listenin son elemanı değil p yi 1 artıracak
- p pointeri 3 ü gösteriyor 4 3 ten küçük değil false döndürecek değil true ve 3 listenin son elemanı değil p yi 1 arttıracak
- p pointeri 5 i gösteriyor 4 < 5 den ve while döndürücü false olacak. Elemanı p pointerının gösterdiği yere ekliyecek ve p pointerının gösterdiği elemanı bir sonraya kaydırıyor.



SONUÇ: Basit bir sıralama algoritması

SONUÇ: INSERT listeye sıralı eleman ekliyor.

```
const int& ListPriorityQueue::min()const
{
    return L.front(); // Listenin minimum elemanı listenin başında bulunuyor
}
void ListPriorityQueue::removeMin()
{
    L.pop_front(); // Diziden eleman silerken Listenin başından (dequeue) siliyoruz. Listenin başında minimum eleman bulunduğunda minimum elemanı silmiş
}
```

```
void ListPriorty::print()
{  std::list<int> M=L;
   while(!M.empty)
{
   cout<<M.front()<<" ";
    M.pop-front();
  }
}
```

// Listenin elemanlarını basarken en baştan itibaren (yani minimum) elemandan başlayarak ekrana basıyor

HEAPS

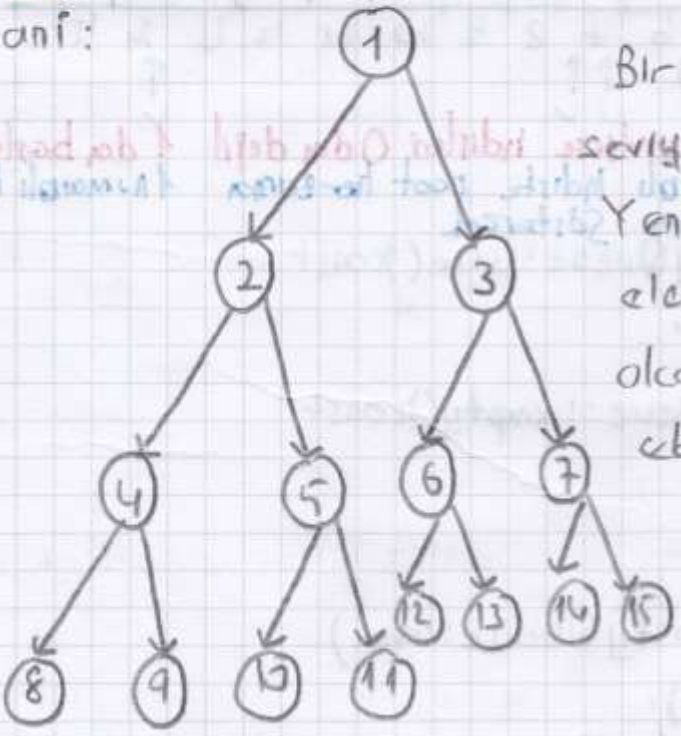
İşimiz kuyruk kuyruğu ağaç gibi yorumlıycaz. Kuyruktan eleman silerken, ağaçtan silcez, eleman ekterken ağaca ekliyormuşuz gibi (Kuyruk - vektördüzü - ağaç bir arada)

Ağaç sıralı, eleman silerken min elemanı silcez. Ağaçta en küçük elemanın root'u göstercek. Dolayısıyla remove min yaptığımızda en küçük elemanı silmiş olcaz.

Ağaca verileri ekterken ağacın dengesiz olmaması için root'un altındaki tüm elemanların ondan büyük olması lazım.

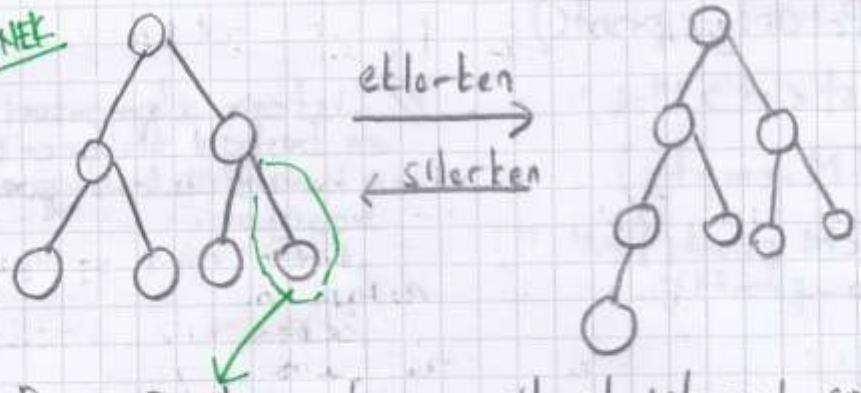
Kökten aşağıya doğru daha küçük sonra büyük olcak şekilde ekliyoruz.

Yani:



Bir seviye tam bitmeden aşağıki seviyeye gitmeye izin vermiycez. Yeni seviyede ilk ekleyeceğimiz eleman yeni seviyede en solda olcak şekilde soldan sağa doğru ekliycez.

ÖRNEK



Bu ağaçtan eleman silmek istersek en sağdakini silcek.

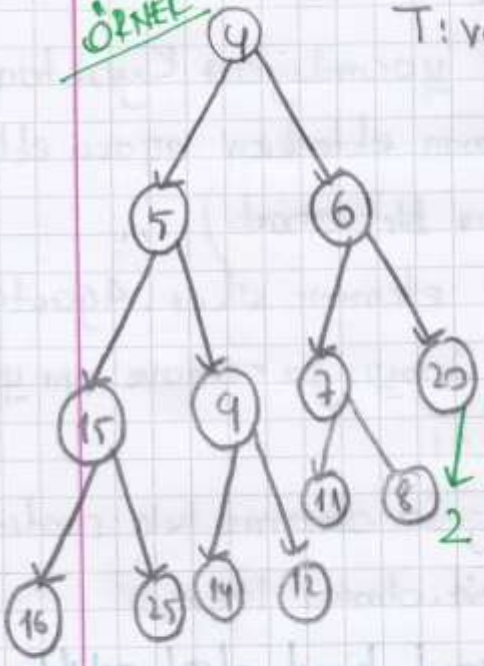
T: vektör değişkeni

Örneğin 2 yi ekliyelim. İlk olarak sonseviye deki en sola ekliycak. Sonra ağacın sıralı olup olmadığını kontrol etcek swaplarla 2 uygun yere getirilcek

!! x ile u swap yapılırken v ile u değil içerikleri swap yapılıyor

v=u atama değil (v u'nun gösterdiği adrese gidiyor)

ÖRNEK



T=0

0	4	5	6	15	9	7	20	16	25	14	12	11	8	2	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

1'ilk eleman root leftroot rightroot

u v

14

1 kullanmıyoruz. Sınavda soru verilirse indisleri 0'dan değil 1'den başlatcaz.

root her zaman 1 numaralı indiste ve en küçük elemanı göstercek

root her zaman 1 numaralı indiste

KOD

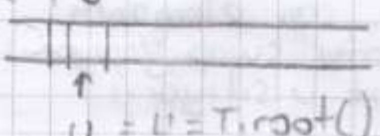
```
int HeapPriorityQueue::size()const
{
    return T.size();
}
bool HeapPriorityQueue::empty()const
{
    return size()==0
}
const Int8 HeapPriorityQueue::min()
{ return *(T.root());
}
bool HeapPriorityQueue::isless(const Int8&,const Int8&) const
{
    if (e<f) return true;
```

```
void HeapPriorityQueue::Insert(const int &e)
{   T.addLast(e);
    Position v = T.Last()
    while(! T.isRoot(v))
    {   Position u = T.parent(v)
        if(!isLess(*v, *u)) break;
        T.swap(v,u)
        v=u;
    }
}
```

// Listenin sonuna eleman ekliyoruz. Onun yerinde değil mi diye kontrol etcez. Değilse yukarı doğru swap yapcaz. Eklediğimizin babasına bakıyoruz değilse swap yapıyoruz. Ağaç doğgun olana kadar (babal çocukta küçük olacak şekilde) swap yapıyoruz

*v → gönderdiğimiz eleman
*u → gönderdiğimiz elemanın parentı

remove min: Rootu almadan önce rootla en son elemanı swap yapıyoruz



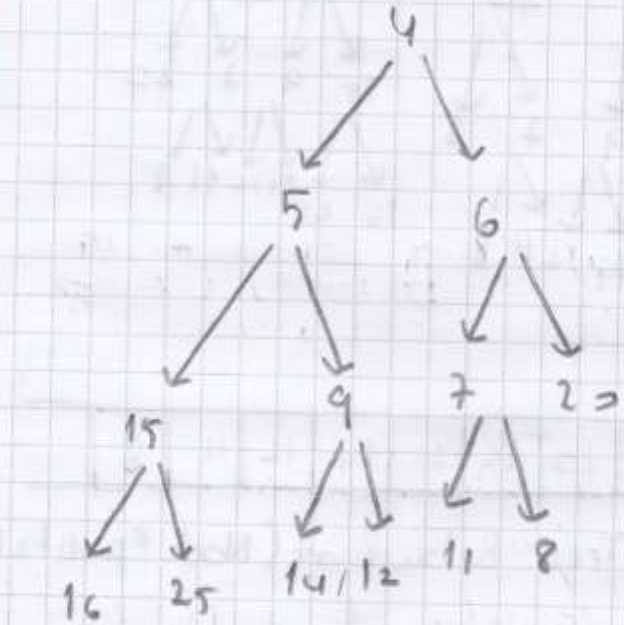
Roottaki eleman minimum mu değilse yukarıdan aşağıya doğru swap yapcaz. Swap yaparken önce sola sonra sağa bakıyoruz min olan hangisiyse onu yapcaz

```
T.hasleft(u) → u nun sol cocuğu varmı?
position parent(const Position &p) return pos(idx(p)/2);
bool has left(const Position &p)const return 2*idx(p)<=size();
bool has right(const Position &p)const return 2*idx(p)+1<=size();
void addlast(const int &e) V.push_back(e);
void removeLast()          V.popback;
void HeapPriorityQueue::removeMin()
{ if(size()==1)
    T.removeLast();
  else
  { Position u=T.root();   // en son elemanla
    T.swap(u, T.last());     root u swap yapıyorum
    T.removeLast();          root u siliyorum
  while(T.hasLeft(u)); u nun sol cocuğu varken
    Position v= T.left(u);
    if(T.hasRight(u) && isless(*(T.right(u)),*v)
    v=T.right(u);
   if(isless(*v,*u)
    { T.swap(u,v)
        u=v;
     }else break
   }
}
}
void HeapPriorityQueue::print()
{ std::vector<int> M=T.v;
  int i=0;
  while(i<=T.size())
  { cout<<M.at(i)<<" ";
}}
```

Uygulaması

```
Heap.insert(4);
Heap.insert(5);
Heap.insert(6);
Heap.insert(15);
Heap.insert(9);
Heap.insert(7);
Heap.insert(20);
Heap.insert(16);
Heap.insert(25);
Heap.insert(14);
Heap.insert(12);
Heap.insert(11);
Heap.insert(8);
```

!!!Baba çocuktan küçük olacak
Ağaca veri eklerken bir seviyede soldan sağa doğru ekliyoruz.

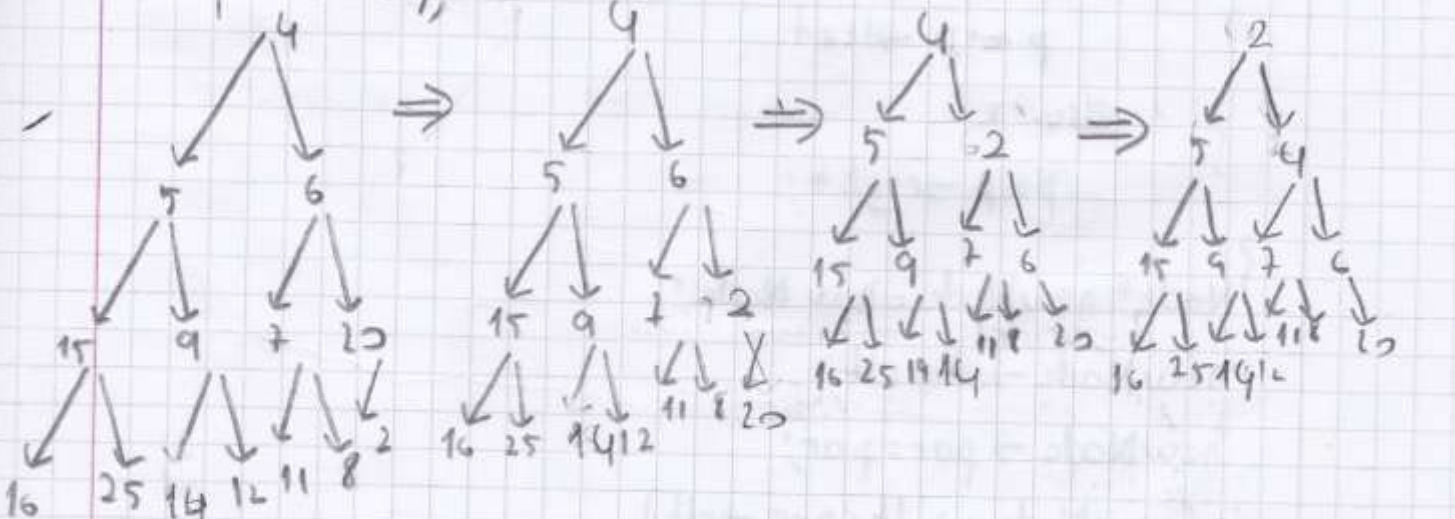


Heap.print

∅	4	5	6	15	9	7	20	16	25	14	12	11	8
0	1	2	3	4	5	6							

Heap.min() 4 (Ağaç dizginse root minimumdur)

Heap.insert(2);



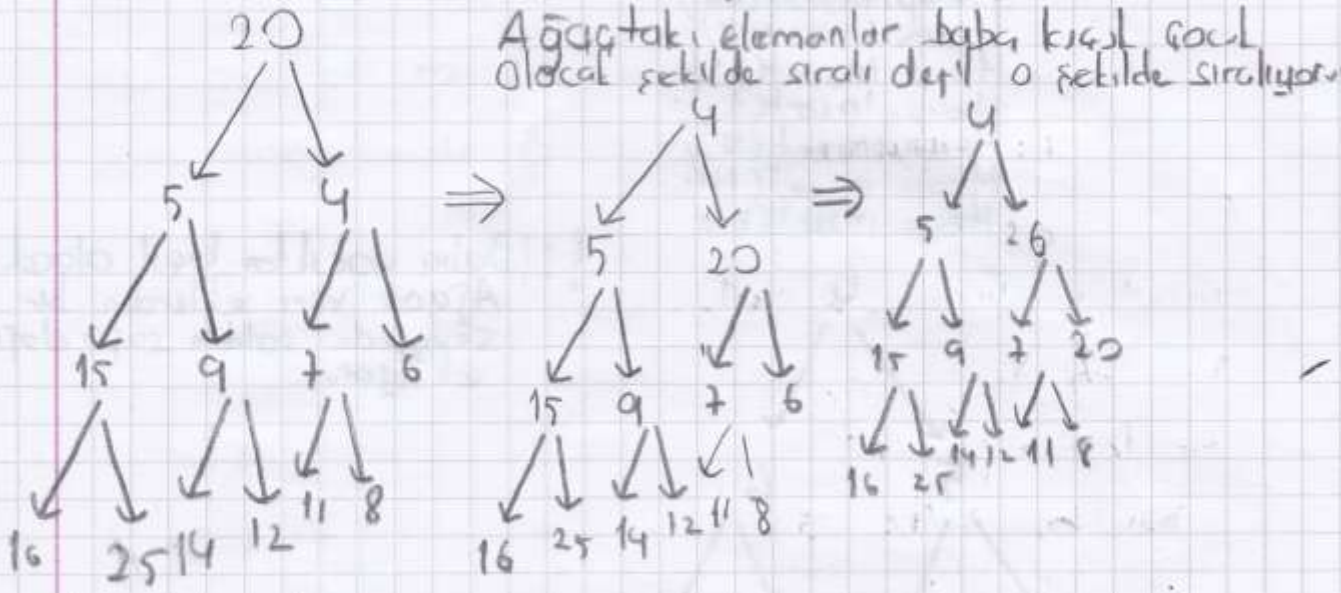
Heap.print

0	2	5	4	15	9	7	6	16	25	14	12	11	8	20

Heap min() 2

Heap.removeMin()

roottaki en sondaki elemanı yer değiştirip rootu sildik



Heap.print()

0	4	5	6	15	9	7	20	16	25	14	12	11	8

Serbest ödev

```
void LinkedBinaryTree::BelowRoot(Node *p, int elt)
{ Node *par;
  while (p != NULL)
  { par = p
    if(p->elt > elt)
      p = p->left
    else
      p = p->right;
  }
Node* newNode = new Node;
newNode->elt = elt;
newNode->par = par;
if(newNode->elt < par->elt)
  par->left = newNode;
else
  par->right = newNode;
n++;
```

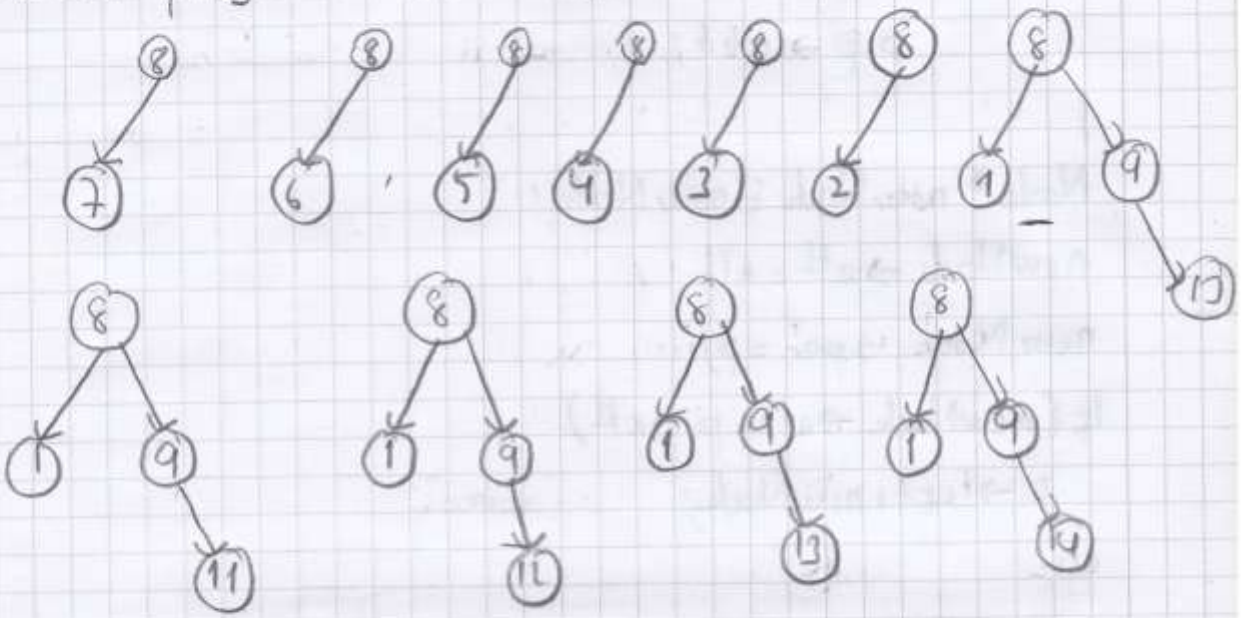
// Eleman eklerken eleman eklediğimiz yerin değerinden küçükse sağa büyükse sola ekliyor.

2. vize

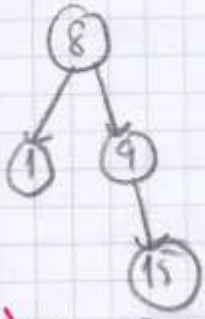
```
void addBelowRootModified(Node*p, int elt)
  while(p->left!=NULL && p->right!=NULL)
  { if (p->elt>elt)
      p=p->left;
    else
      p=p->right;
  }
  Node* newNode = new Node;
  newNode->elt=elt;
  newNode->par=p;
  if(newNode->elt<p->elt)
    p->left=newNode;
  else
    p->right=newNode;
  n+=1;
}
Void main()
{ Linked BinaryTree nuTree;
  nuTree.addRoot();
  nuTree root->elt=8;
nuTree. addBelowRootModified (nuTree._root, 7)
                            , (nuTree._root, 6)
                            (nuTree._root, 5)
                            (nuTree._root, 4)
                            (nuTree._root, 3)
                            (nuTree._root, 2)
                            (nuTree._root, 1)
                            (nuTree._root, 9)
                            (nuTree._root, 10)
```

(nLTree.root, 12)
(nLTree.root, 13)
(nLTree.root, 14)
(nLtree.root, 15)

Yukaridaki programin çiktisi nedir.

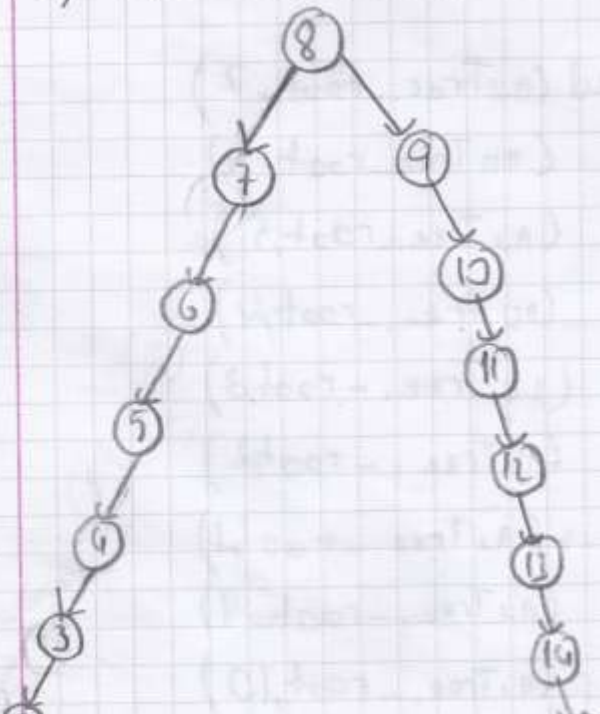


çiktisi : 8 1 9 15



2) 8 7 6 5 4 3 2 1 9 10 11 12 13 14 15

a) Yukaridaki verileri ikili agaca yerlestirmis.



inorder: L P R

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

preorder: P L R

8 7 6 5 4 3 2 1 9 10 11 12 13 14 15

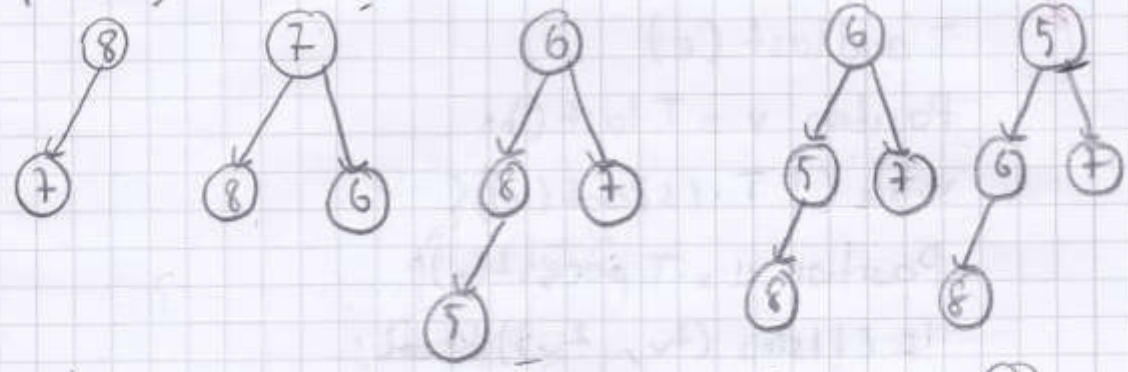
postorder: L R P

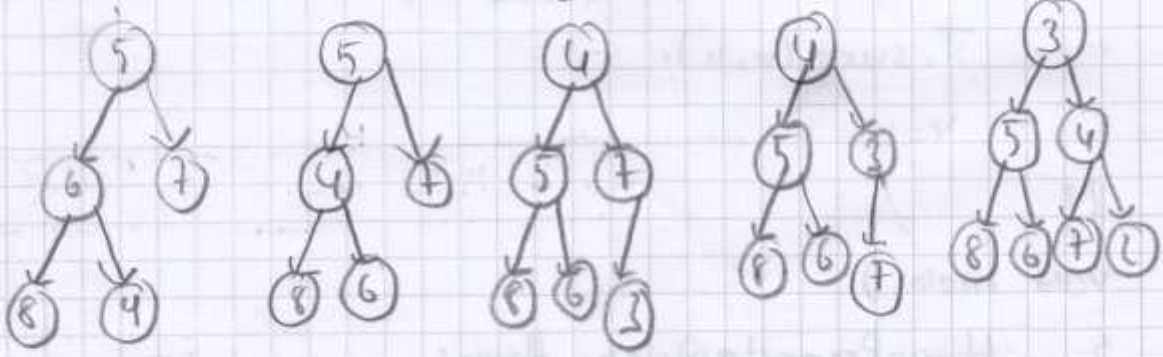
1 2 3 4 5 6 7 15 14 13 12 11 10 9 8

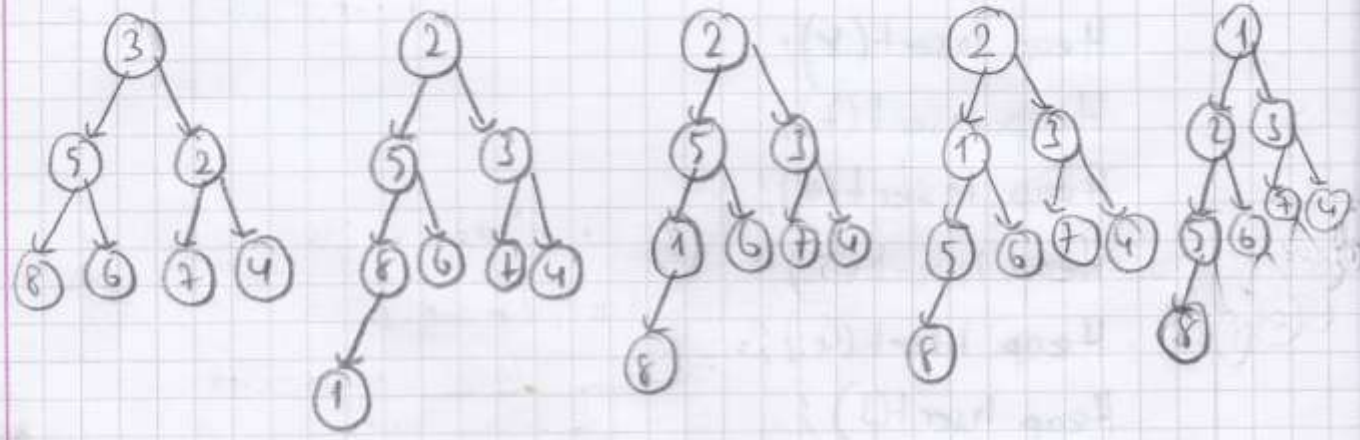
3) void HeapPriorityQueue::Insert(const Int& e)

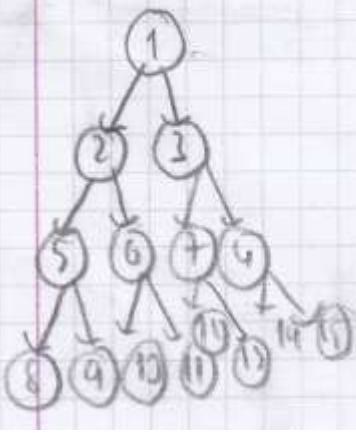
```
{   T.addlast(e)
    Position v = T.last();
    while(!T.isRoot(v)){
    Position u = T.parent(v);
    if(!isles(kv, ku)) break;
    T.swap(v,u);
    v=u;
}}
void main()
{   HeapPriorityQueue Heap;
   Heap.Insert(8);
   Heap.Insert(7);
   Heap.Insert(6);
   Heap.Insert(5);
   Heap.Insert(4);
   Heap.Insert(3);
   Heap.Insert(2);
   Heap.Insert(1);
   Heap.Insert(9);
   Heap.Insert(10);
   Heap.Insert(11);
   Heap.Insert(12);
```

```
Heap.Insert(14);
Heap.Insert(15);
cout<<"Heap Elements after Insertions:";
Heap.print();
Heap.removeMin();
cout<<"Heap Elements after removeMin:";
Heap.print();
```



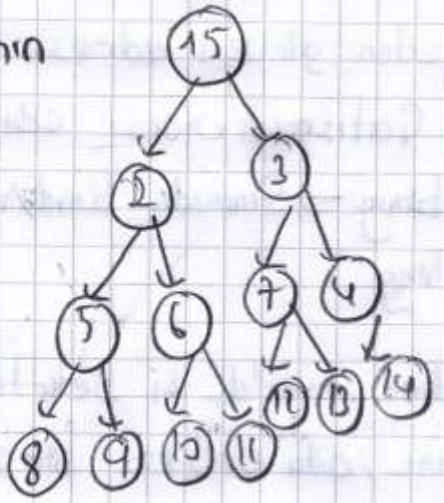


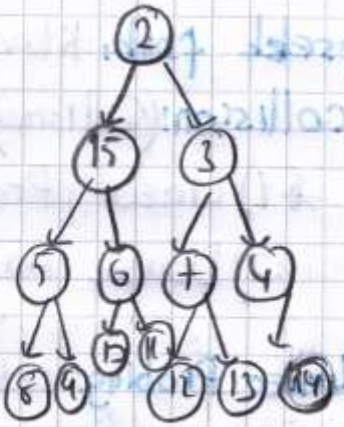


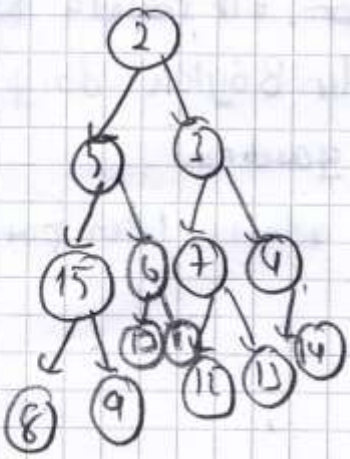


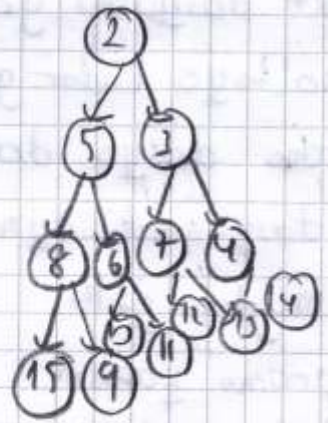
0	1	2	3	5	6	7	4	8	9	10	11	12	13	14	15

after remove min









0	2	5	3	8	6	7	4	15	9	10	11	12	13	14

HASHING

Dosya organizasyonu

Bağıl Dosya Organizasyon: Sözlük uygulaması klasiyeden bir kelime girildiğinde. Türkçe karşılığını bulup

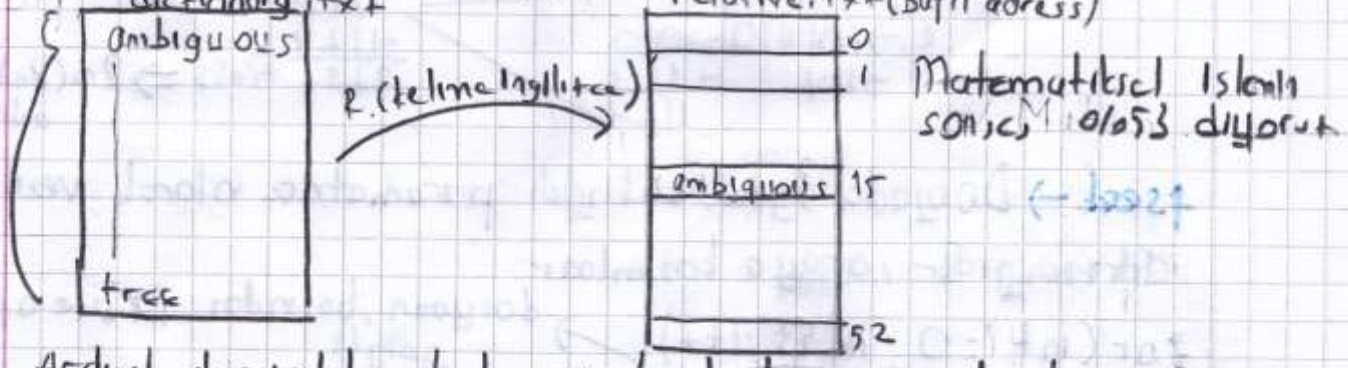
R (key) → Adres

Hash fonksiyonu

key: kelime, İngilizce

Hashing: Kelimeyi alıp karakterlerin ASCII karşılığına göre matematiksel işlem yapıp adres döndürüyor. Bağıl (relative) adress

Ardışıl (sequential) dosya



31

Ardışıl dosyadaki kelimeyi bağıl dosyaya adresleyip arama yapacak. Bağıl dosyayı çakışma fazla olmasın diye büyük seçiyoruz.

fseek fonk: Klavyeden girilen adrese direk gidiyor

collision: (Çakışma) Çakışma olursa çözüm yöntemleri

*1) Linear Probing → Sınavda bundan sorumluyuz

2) Double Hashing

Linear Probing: relative.txt'de ki belirtilen adrese gider. Orası dolu ise bir sonrakine, oda dolu ise bir sonrakine boş yer bulmaya kadar tüm txt dosyasını gezer. En sonunda boş yer bulamazsa başladığı noktaya kadar gelir. Böylece dosyanın dolu olduğu anlaşılır. Ekrana dosya dolu yazdırır.

→ Çakışma: Relative dosyasında aynı adrese 1'den fazla veri gönderilirse çakışma meydana gelir.

Or overloading için arama yapalım

```
int Hash(char *key)
{
 int sum=0
 for(j=0;j<4;j+=2)
 sum=(sum+10*key[j]+key[j+1]
 sum=sum %953;
   return sum;
}
```

Over
↑ ↑
j j+1

ASCII kod
O → 111 x 10 = 1110
V → 118 + 118
sum 1228

e → 101 * 10 = 1010
r → 114 + 114
1124

1228
+1124
2352 %953 ⇒ 20 (Relative adress)

fseek → Dosyada byte cinsinden parametre olarak verilen adrese gider, oraya konumlanır.

```
for(int i=0;i<53;i++)   → dosyanın başından i'nci site of kadar ileri
{
   fseek(rel,i * site of (keline)*20)
                ↓
          hashin döndürdüğü değer
```

vehkle (notsume)
adres = (adres+1)%53} // Kelime ararken ilk gönderilen adreste yoksa bir sonraki adrese sonra bir sonraki adrese bakar (Söylediği adreste çakışma varsa o adreste olmaz Bir sonraki adreste olur)

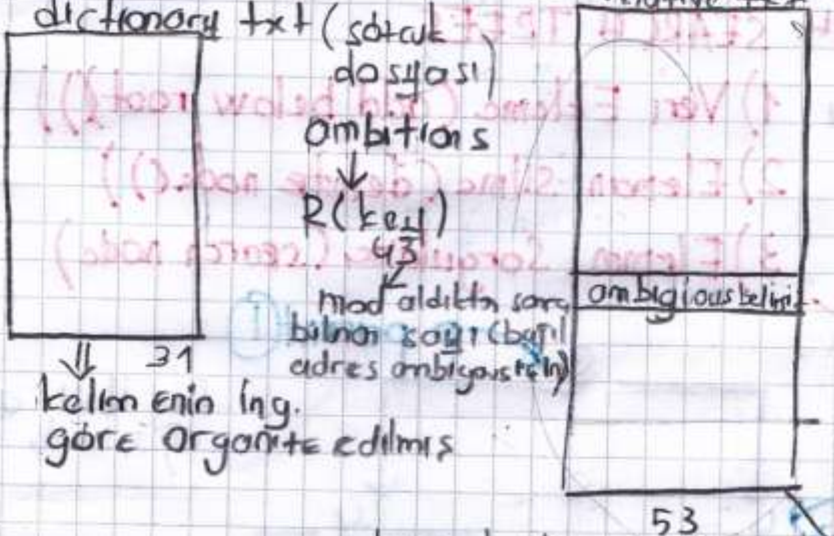
İkinci ket anlatıldı.

DOSYA ORGAZİSYONU (HASHING)

Sözcük uygulaması. Klavyeden ingilizce bir kelime giriliyor. Program sözcük dosyasından bu kelimenin Türkçesini söylüyor.

Hashing: Ayrı bir dosya daha oluşturuyor. Hash fonk aldığı kelimenin ASCI karşılığına göre işlem yapıyor.

Ör



key = kelime. ingilizce

Çakışmayı engellemek için relative txt daha büyük tutuyoruz.

Ör | am | bi | gious |

```
int sum=0
for (int j=0; j<4; j+=2)
  sum=(sum+10*key[j]+key[j+1]);
  sum=sum%53
  return sum;
```

10×97 = 970
+109
1079

10×98 = 980
+105
1085 = 43

43

Sorgulama yaparken kelimeyi kaydederken yaptığımız işlemin aynısını yapıyoruz.

collision (çakışma): Bütün kelimeler için farklı adres üretemeyebilir. Dolayısıyla farklı kelime için aynı adresi üretebilir. Buna collision (çakışma) denir.

Linear probing: Verilen adres doluysa bir sonrakine oda doluysa bir sonrakine bakıyor Başladığı yere gelinceye dek kelime için yer arıyor Sorgularkende linear probing yapıp başta adreslere bakıcaz

Hiçbir kayıt yapmadan — Öncelikle bütün kayıtlar için * sembolü koyuyoruz * sembolü dosyanın boş olduğunu gösteriyor.

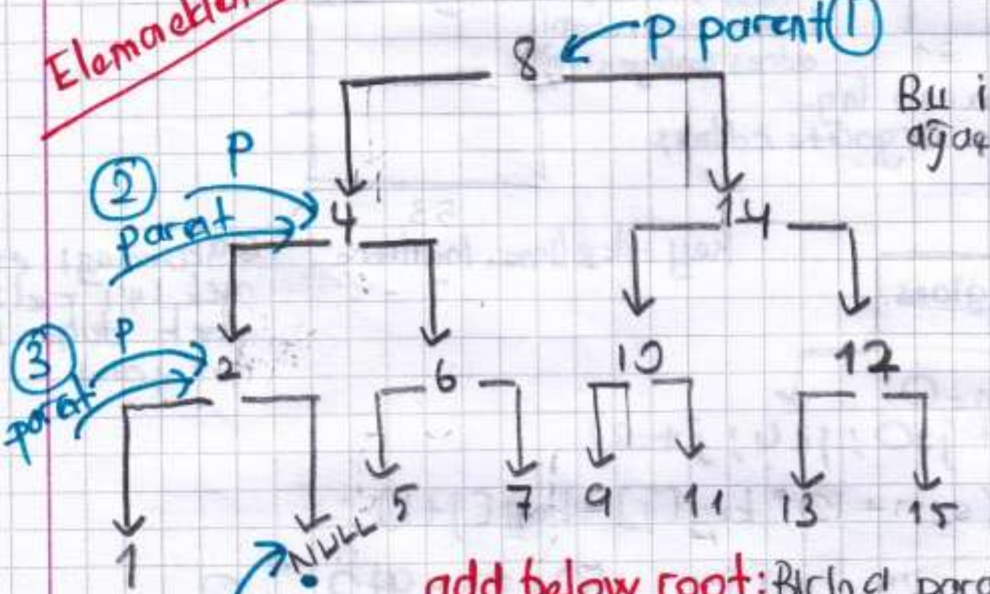
Adres = (Adres+1)%53 → Sona geldiğinde tekrar başa gelmesi için mod 53 alıyoruz.

Hocanın çalışma sorularına bak!!

BİNARY SEARCH TREES

İkili ağaca
1) Veri Ekleme (add below root())
2) Eleman silme (delete node())
3) Eleman Sorgulama (search node)

Eleman ekleme

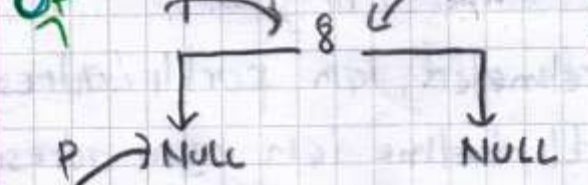


Bu ikili ağaçtan farklı ağaçlarda var

add below root: Birinci parametre root
İkinci parametre ekliyeceğimiz eleman elt=3

!!! parent hala 2
2'ye parent üzerinden erişiyoruz null da parent bağı yok. Null'da parent fonksiyonuna o yüzden ihtiyaç var.

ÖR

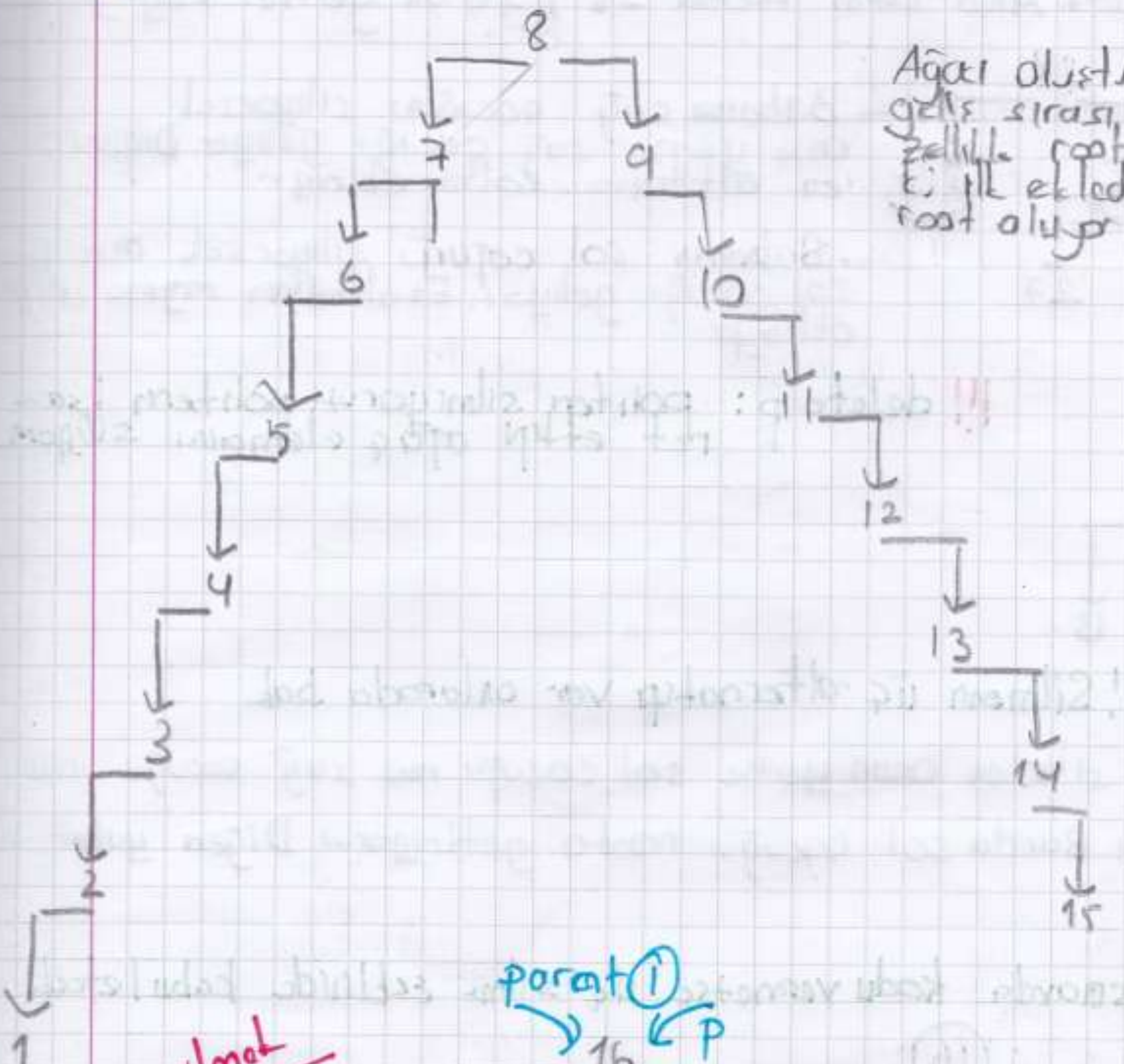


ÖR
p=parent ikiside adres
ikiside aynı adresi gösteriyor

P Null olduğu anda ağaçtan kopuyor. O yüzden parent pointer

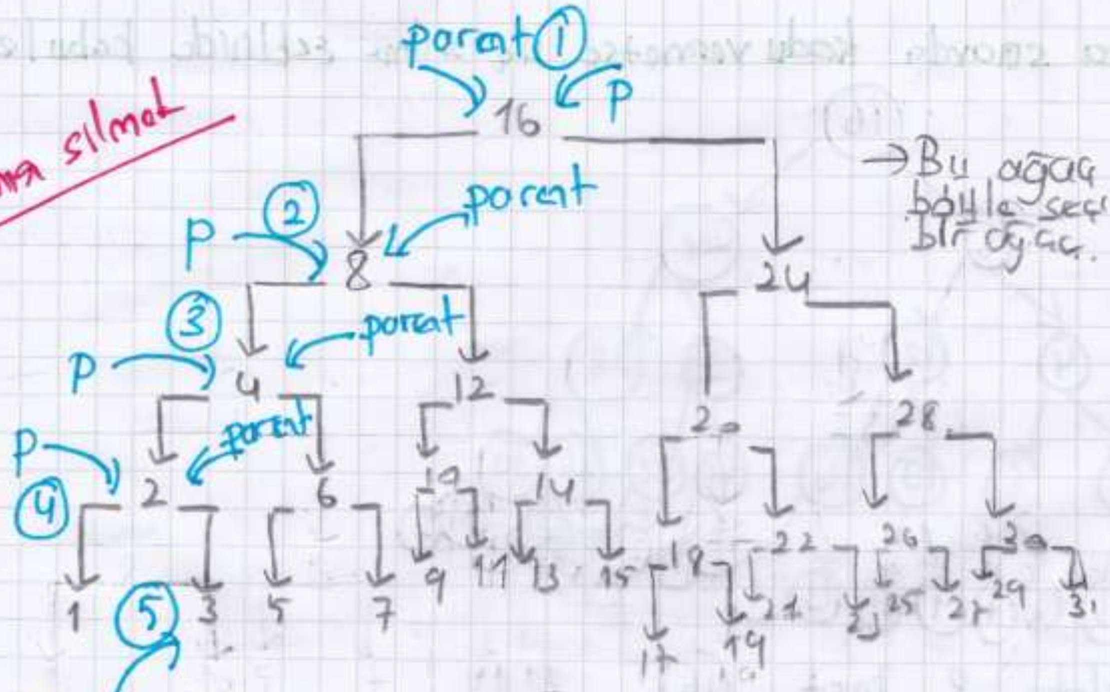
SINAV

8 7 6 5 4 3 2 1 9 10 11 12 13 14 15



Ağacı oluştururken verilerin geliş sırası çok önemli ö-zellikle root çok önemli çün-kü ilk eklediğimiz eleman root oluyor

Eleman silmek



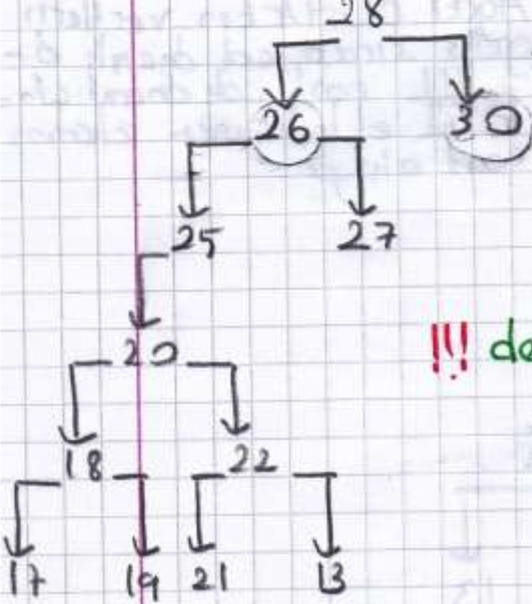
→ Bu ağaç özellikle böyle seçilmiş dengeli bir ağaç.

- p 3'ü silmeye çalışıyoruz

silceğimiz elemanın yerini göstermek için p kullanılıp- While döngüsünden çıktığımızda p silincek Parentte onun babasını göster-cek.

right ve lefti null yada herhangi biri nullsa kolay.nulldır

Ör: 24 ü silelim. 24 ü silerken 20 yi yada 28 i 24 ün yerine koyabiliriz. Ama bizim kodda 28 i 24 ün yerine koyuyoruz.



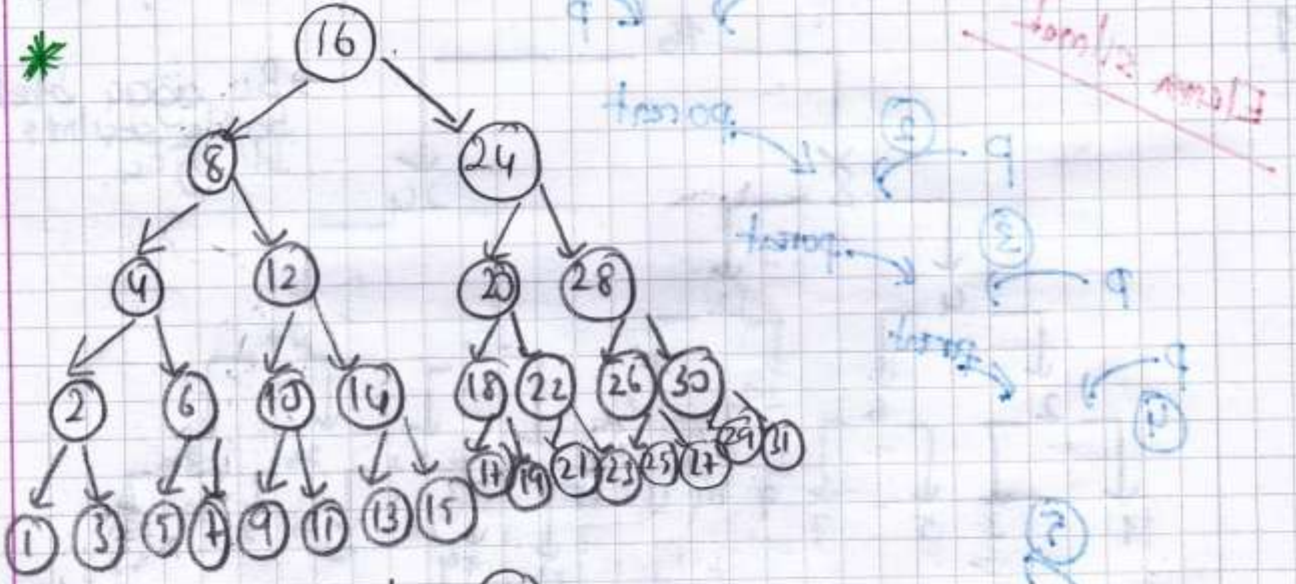
- Babanın sağ çocuğunu siliyorsak onun yerine sağ çocuğu geliyor. Diğeri en alttakinin soluna ekleniyor.
- Babanın sol çocuğunu siliyorsak onun yerine sol çocuğu geliyor. En alttakinin sağına diğeri ekleniyor.

!!! delete p: pointeri silmiyoruz pointerin işaret ettiği ağaç elemanını siliyoruz.

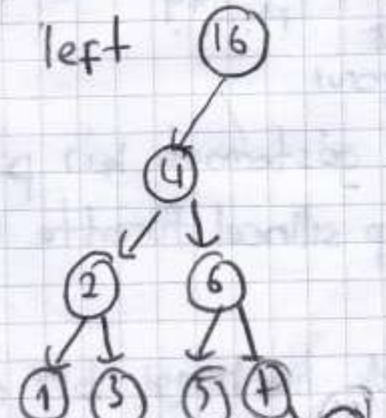
!!! Silmenin üç alternatif var onlarada bak

Rootu silerken onun yerine sol çocuğu mu sağ çocuğu mu getircez. Burda sol çocuğu roota getiriyoruz. Diğeri yukarıdaki gibi.

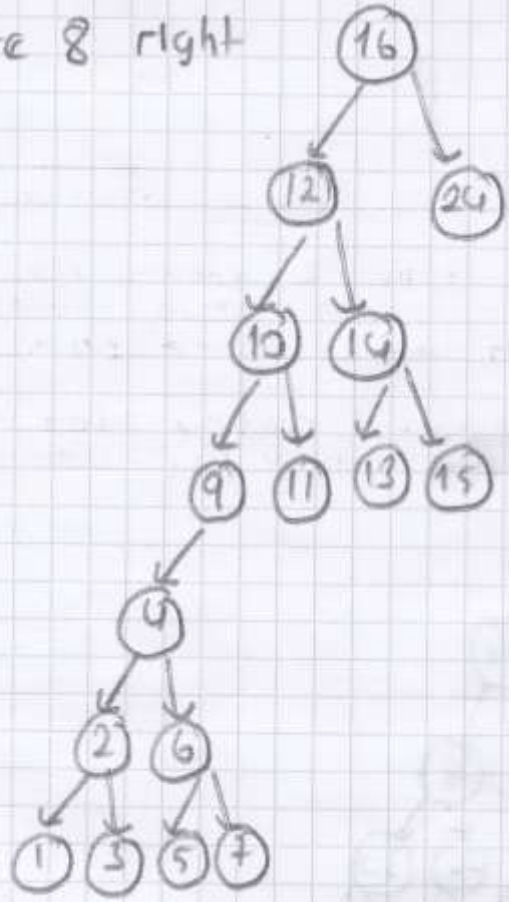
Hoca sınavda kodu vermezse üç silme eklerinde kabul edecek.



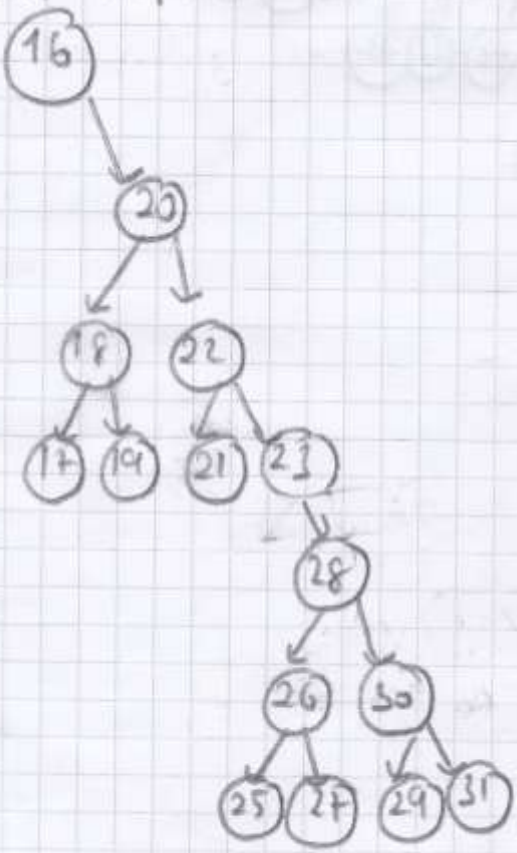
Delete 8 left



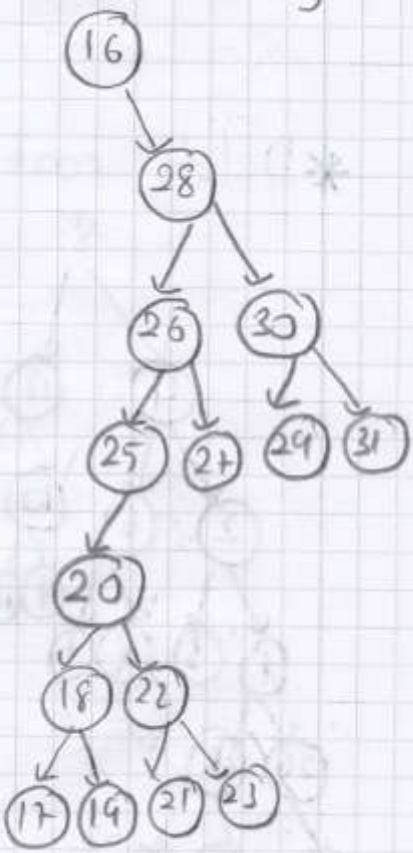
* Delete 8 right



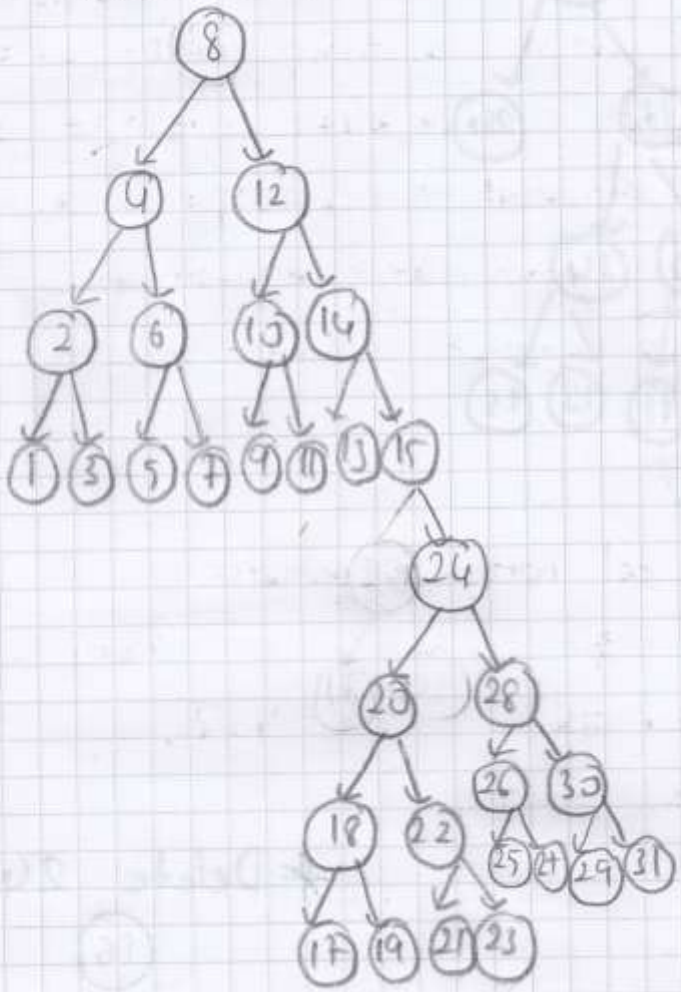
* Delete 24 left



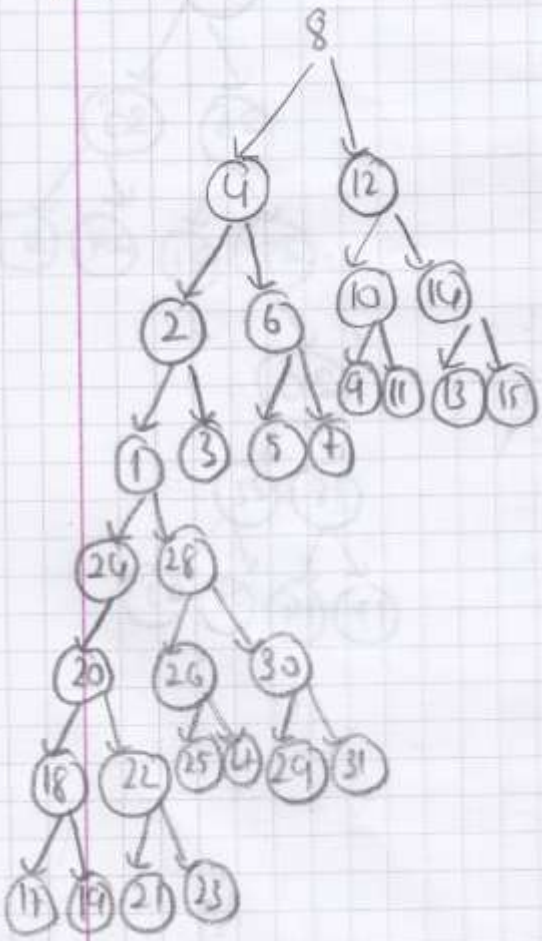
* Delete 24 right



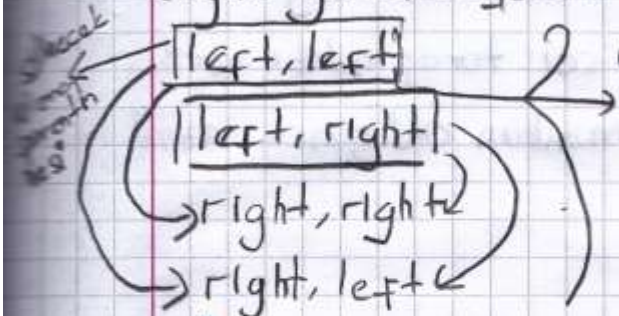
* Delete root , left



* Delete root right



Bizim gördüğümüz silme işlemi parentin sol çocuğunu siliyorsak onun yerine onun sol çocuğu sağ çocuğu siliyorsa onun yerine onun sağ çocuğunu yerleştiriyoruz. Rootu siliyorsak yerine sol çocuğunu yerleştiriyoruz. Ama diğer alternatiflerde var.



strcpy: Sağdakini sol tarafa kopyalıyor.

while döngüsüyle diğerini nereye yerleştireceğimizi buluyoruz.

```
if (p->left! = NULL && p->right! =NULL)
{  if (parent->left==p)
 {
   ≡
 }
 else (parent->right==p)
{
  ≡
}
```

BINARY SEARCH: Herhangi bir elemanı arama

İkili ağaca veri ekleme veri silmekte hiçbir fark yok. Add below root'tan farkı dosyadan okuması

list

client

binary

array

ambiguous

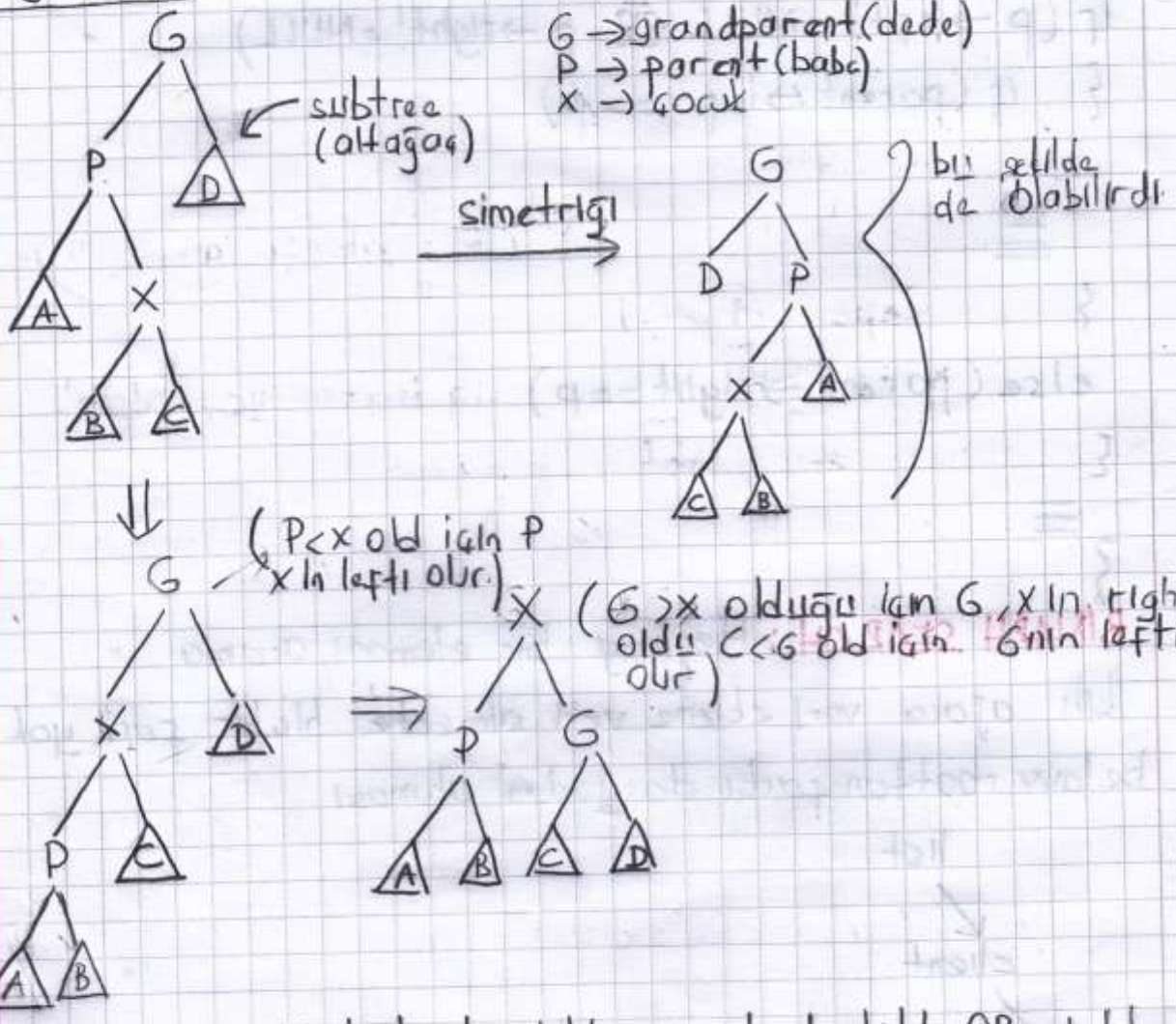
SPLAY TREES

* Splay Tree'ye veri eklendikten sonra 3 tane işlem vardır Zig-Zag, Zig-Zig, Zig Uygun olanı seçilerek eklenen veri yukarı doğru çekilir. Bu üç işlemin amacı ağacın dengeli olmasını sağlamaktır. Ağacın sağ ve sol tarafındaki veriler yaklaşık eşit olmalıdır. Seviye mümkün olduğunca azaltılmalıdır.

1) ZIG-ZAG

X is left child of a left child OR right child of a right child

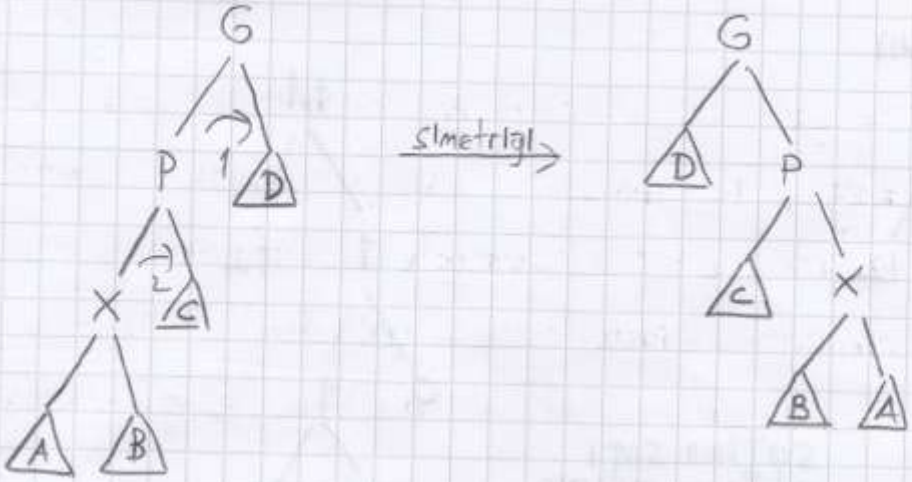
I II



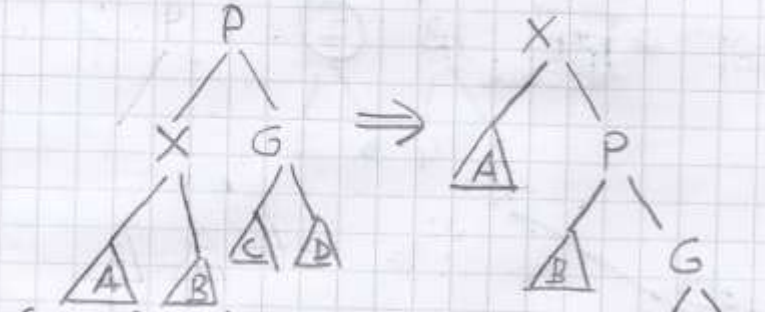
2) ZIG-ZIG X is left child of a left child, OR right child of a right child.

I II

* Ağacı oluştururken eklenen her eleman child olarak eklenir



(Önce baba ile dede yer değiştiriyor)

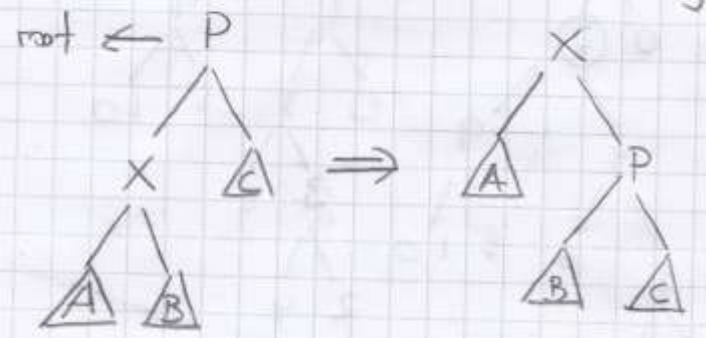


(C < G old. için G'nin left çocuğu olur)

(B < P olduğu için B, P'nin left child'ı olur)

3) ZIG : Repeat (1) & (2)

X is child of root (Zig işlemini yapabilmek için X'in baba olması lazım) root'un çocuğu.

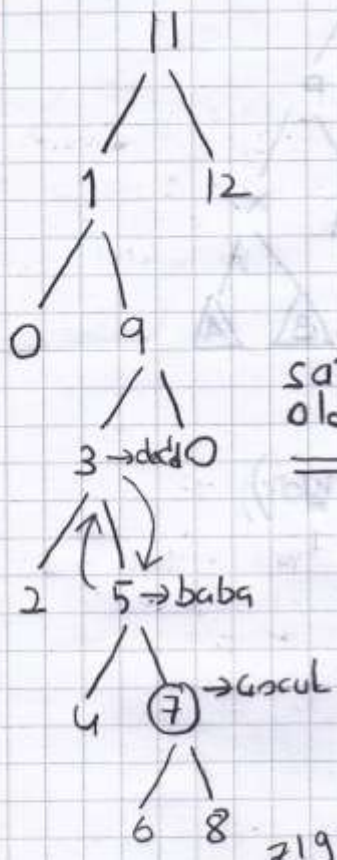


X < P old. için P, X'in right child'ı olur.

B < P old. için P'nin soluna yatılır.

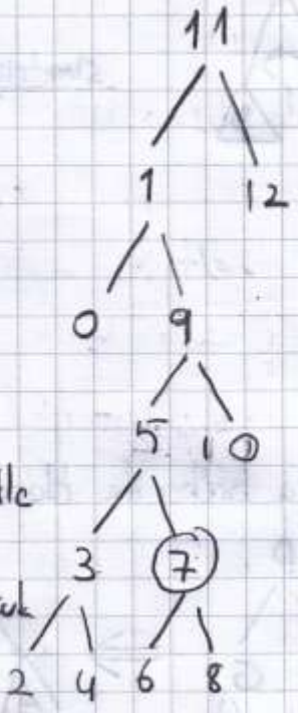
(sağ çocuğuda olabilir)

Ör x=7 için

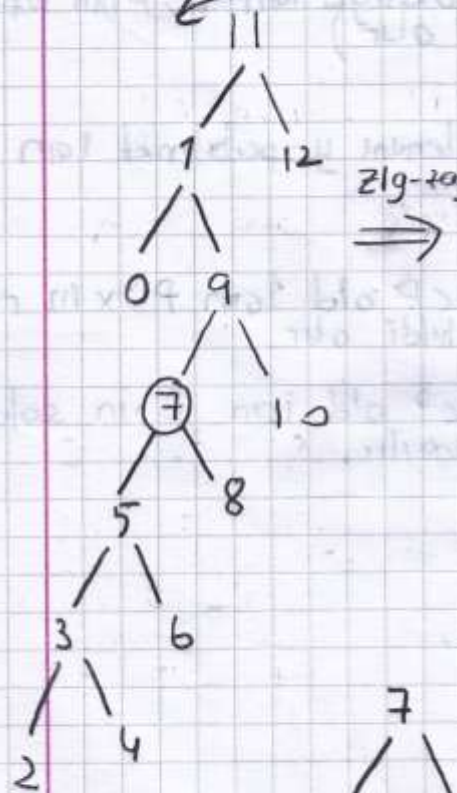


sağının sağı olduğu için zig zig

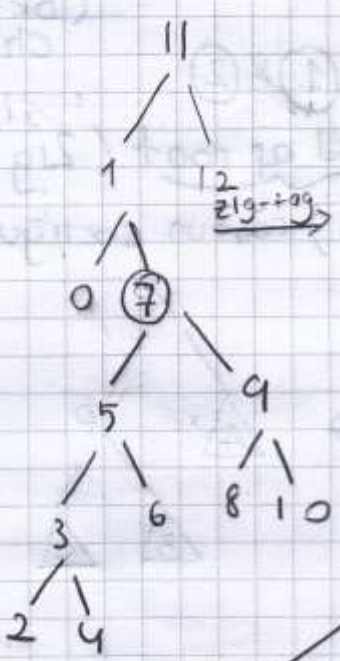
1) dedelle baba
2) baba ile çocuk



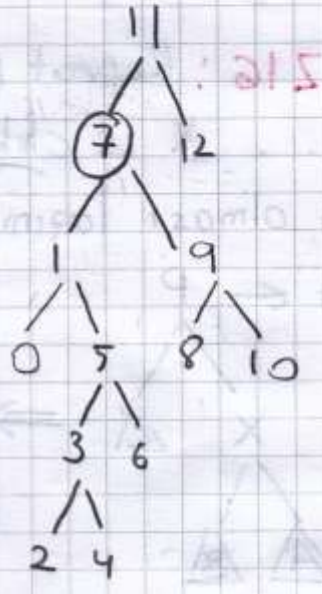
zig-zig



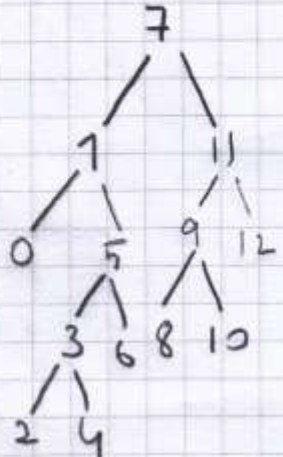
zig-zag



zig-zag



zig

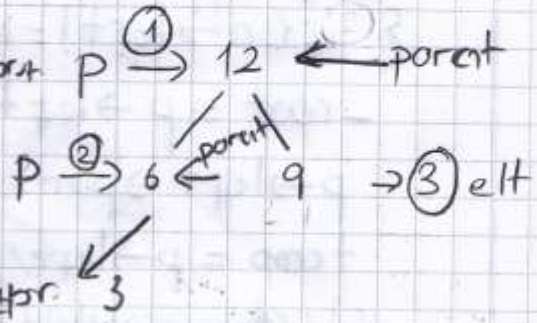


Sevilla sayısı atıldı ağaç daha dengeli oldu.

BİNARY TREE

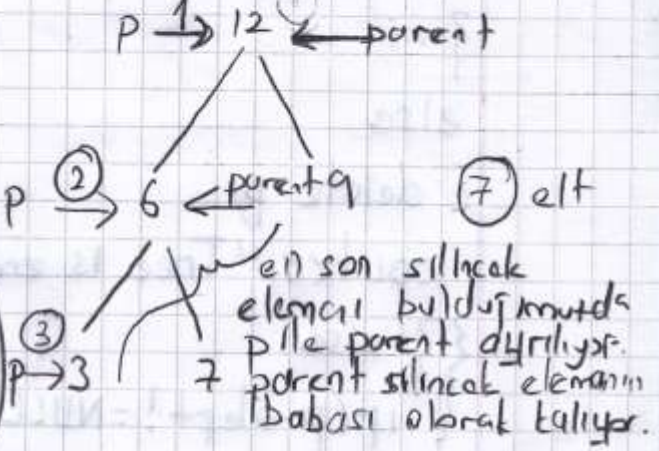
```
void LinkedBinaryTree::addBelowRoot(Node*p,int elt)
{Node*parent;
 while(p!=NULL)
 {  parent=p
     if(p->elt>elt)
        p=p->left;
     else  p=p->right
 }
 Node*newNode=new Node
 newNode->elt=elt;
 newNode->par=parent;
 if(newNode->elt<parent->elt)
    parent->left=newNode
 else  parent->right=newNode;
 n+=1;
}
void LinkedBinaryTree::deleteNode(Node*p,int e)
{ Node*temp;
  Node*parent;
 while(p!=NULL)
 {  if(p->elt==e) break;
  else {
        parent=p
        if(p->elt>e)
           p=p->left;
        else p=p->right;
 }}
 if(p==NULL)
 { cout<<"The node does not exists"<<endl
}
```

eklenecek yeri buluyor. p de parent'ı o anki yeri gösteriyor

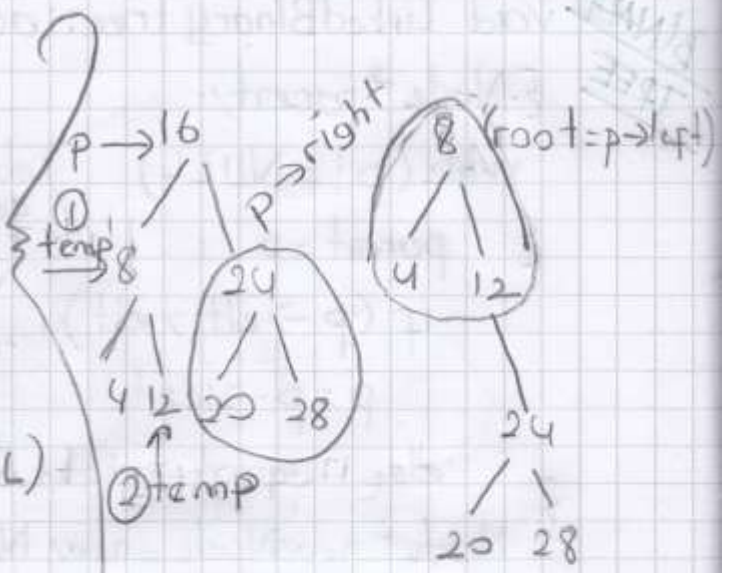


eklenecek yerin sağına mı solunu mu eklicağımızı karar veriyoruz.

silincek elemanı buluyoruz

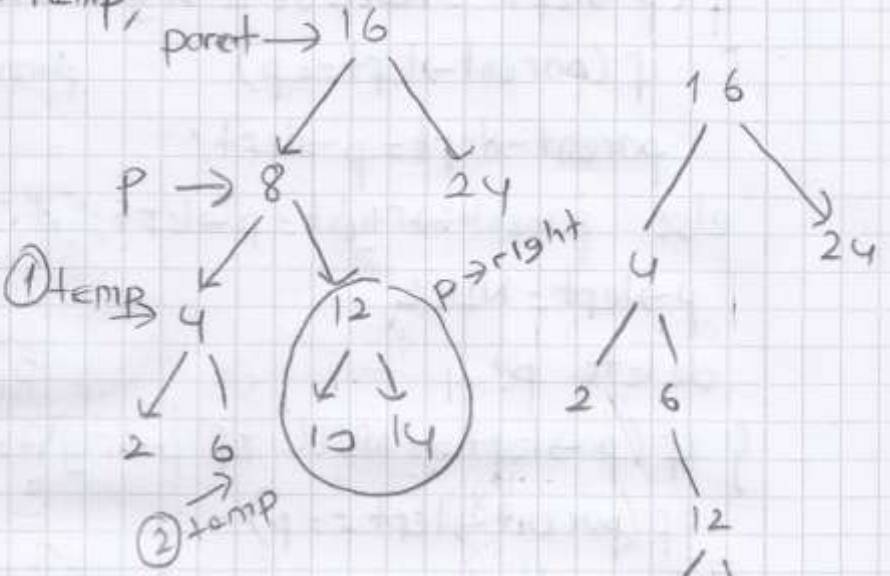


```
else
{  if(p==root)
    { if(p->left!=NULL)
      -root = p->left;
       p->left->par = NULL;
       temp = p->left;
    while(temp->right!=NULL)
      temp= temp->right;
    temp->right = p->right;
    temp->right->par = temp;
     p->left = NULL;
     p->right= NULL;
   delete p;
}
 else if(p->right!=NULL)
{ -root = p->right;
   p->right->par= NULL
  delete p;
}
else
{ delete p;
  cout<< "Tree is empty!" <<endl;
}} else
 { if(p->left!=NULL && p->right!=NULL)
 { if(parent->left==p)
   parent->left = p->left
  p->left->par=parent;
   temp = p->left
 while(temp->right!=NULL)
 temp=temp->right;
```

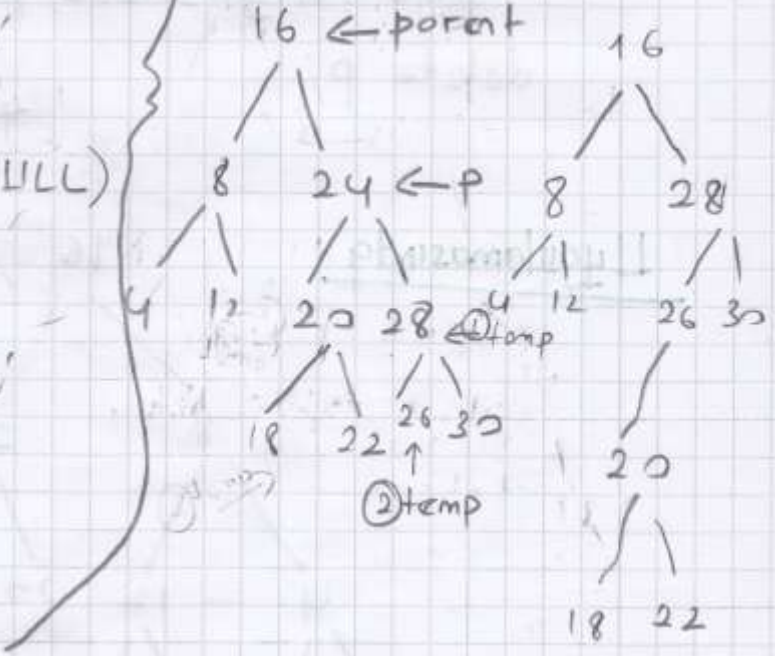


root silerken root için ilk önce aşağıdaki sola bakıyor sol'da eleman yoksa root yerine o geliyor eğer yoksa root yerine sağındaki geliyor.

```
temp->right->par = temp;
p->left = NULL;
p->right = NULL;
}
```

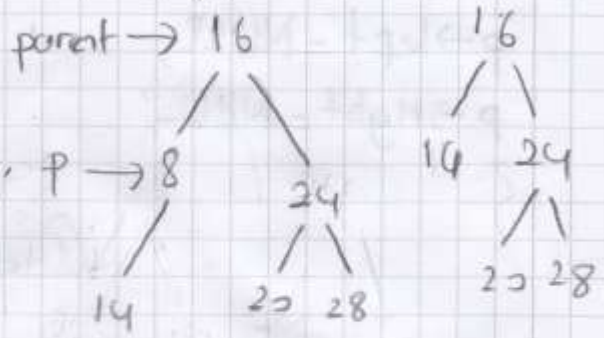


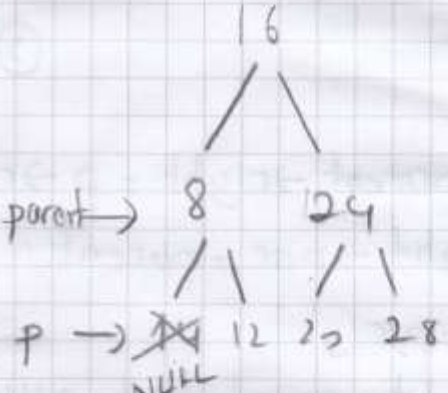

```
else { parent->right = p->right;
p->right->par = parent;
temp = p->right
while (temp->left != NULL)
  temp = temp->left;
temp->left = p->left;
temp->left->par = temp;
p->left = NULL;
p->right = NULL;
}
delete p
}
```



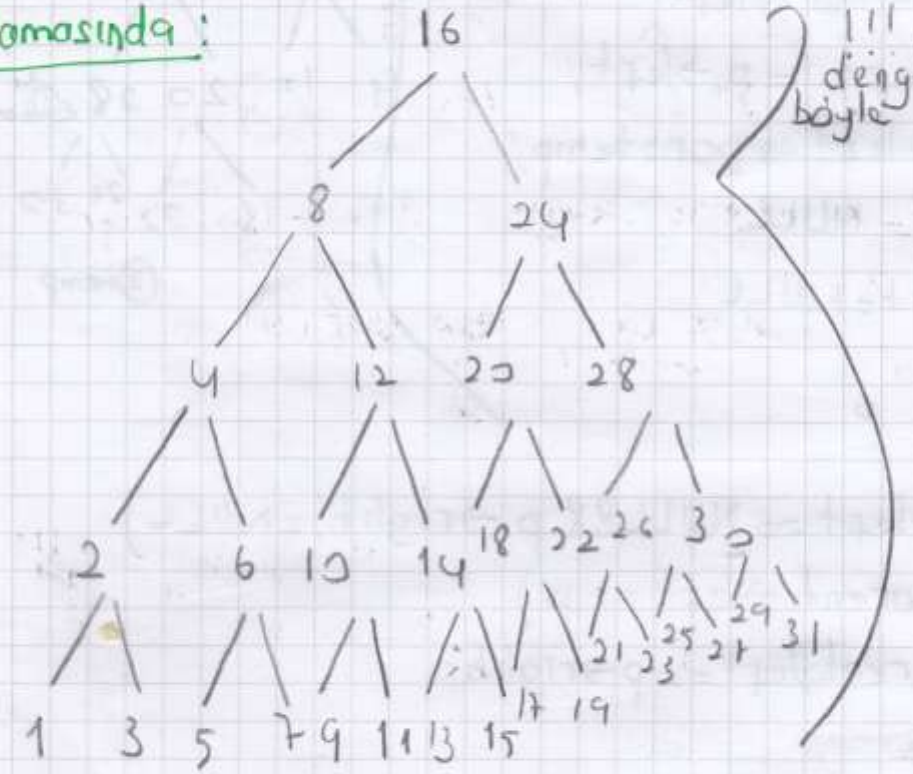

```
if (p->left == NULL && p->right != NULL)
{ if (parent->left == p)
    parent->left == p->right
  else
    parent->right = p->right;
    p->right = NULL
    delete p
}
```

```
if (p->left != NULL && p->right == NULL)
{  if (parent->left == p)
     parent->left = p->left;
   else parent->right = p->left;
   p->left = NULL
   delete p;
} if (p->left == NULL && p->right == NULL)
   if (parent->left == p)
     parent->left = NULL;
   else
     parent->right = NULL;
   delete p;
```





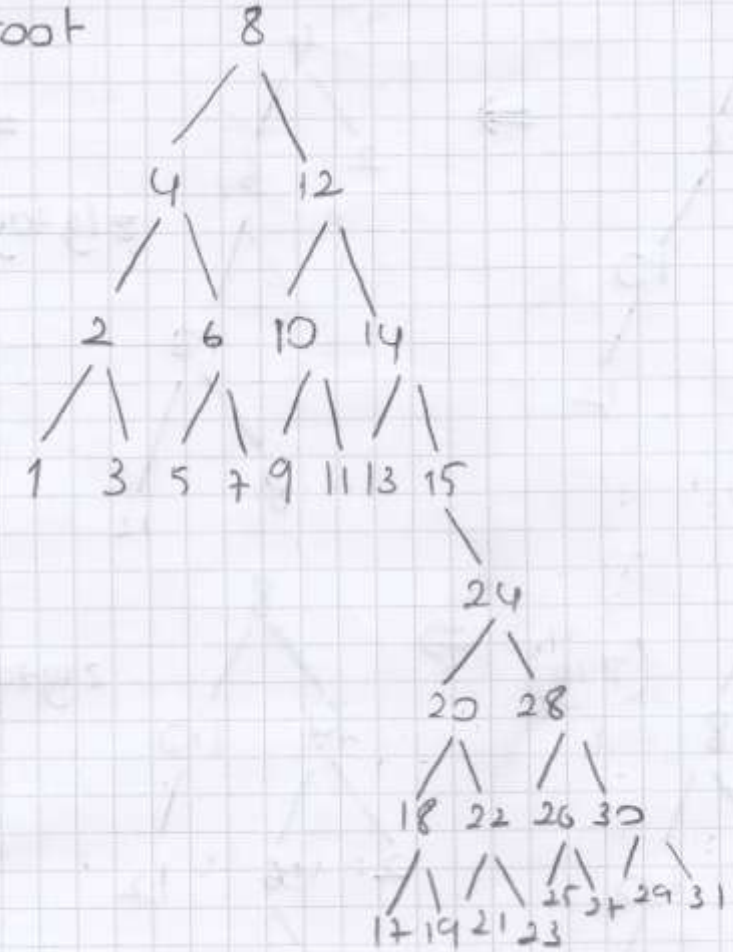
Uygulamasında:



!!! ağaç dengeli verilmiş böyle olmayabilirdi.

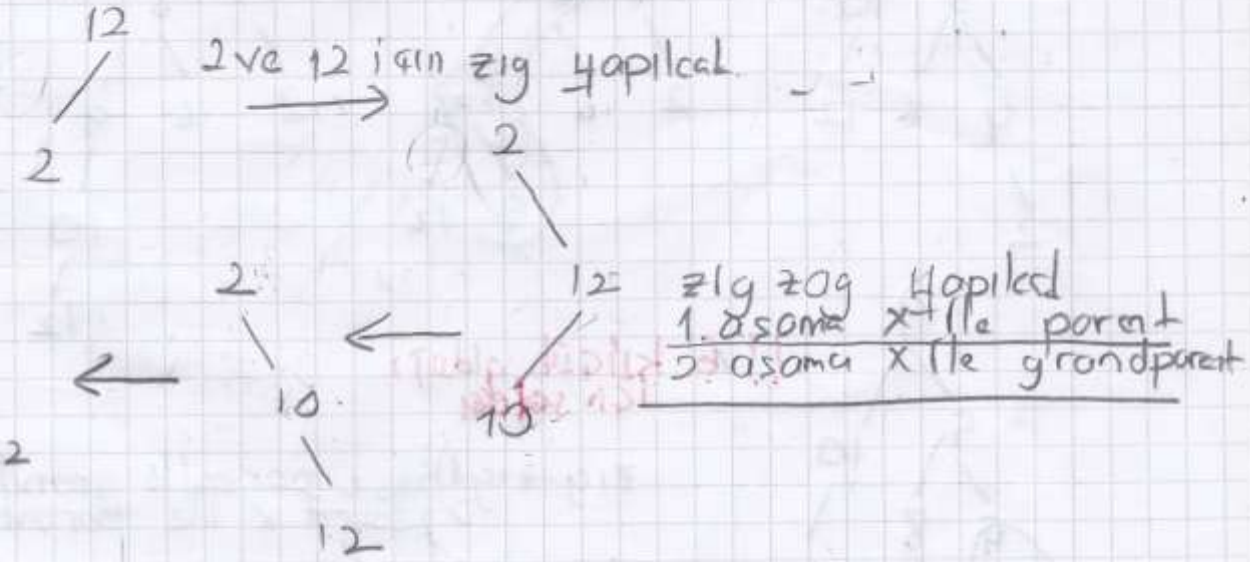
inorder: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
LPR 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31

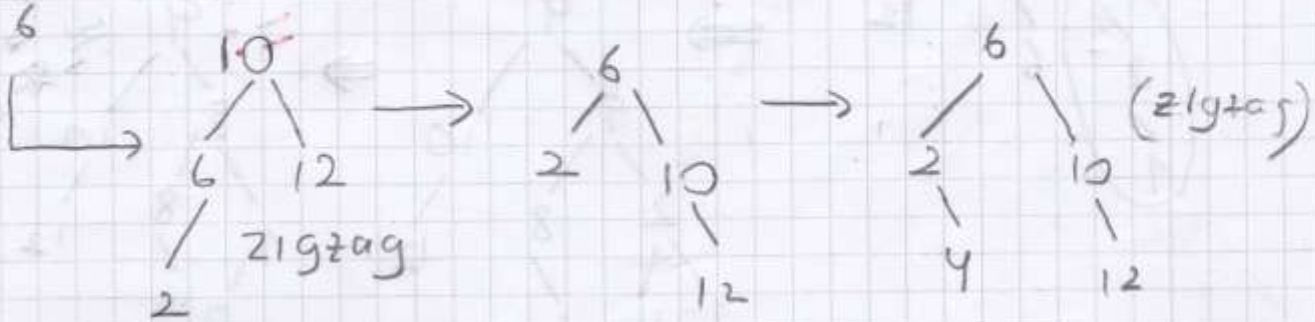
delete root

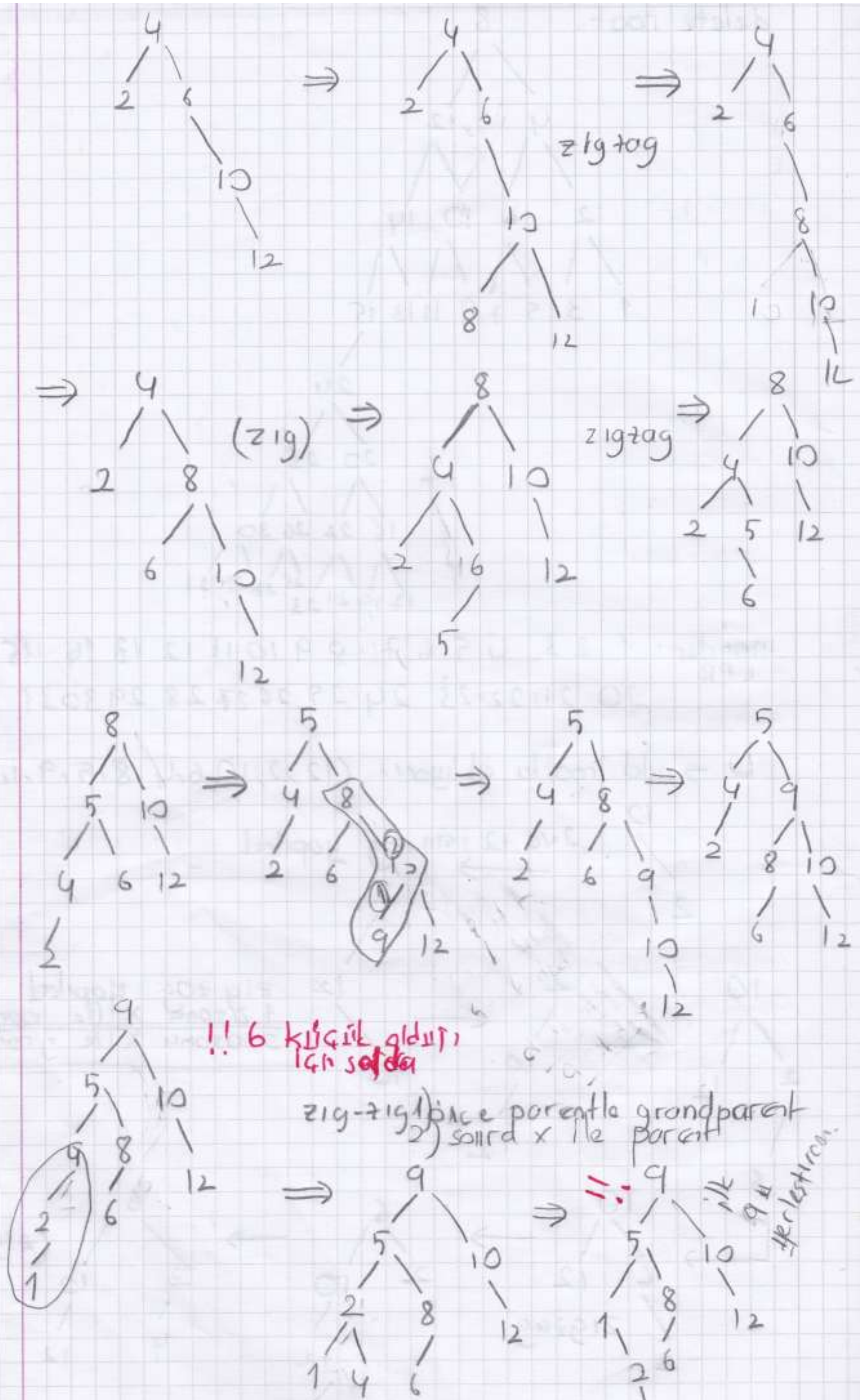


inorder: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 17 18 19
LPR
20 21 22 23 24 25 26 27 28 29 30 31

12 → add rootla ekliyoruz. (12, 2, 10, 6, 4, 8, 5, 9, 1, 13)

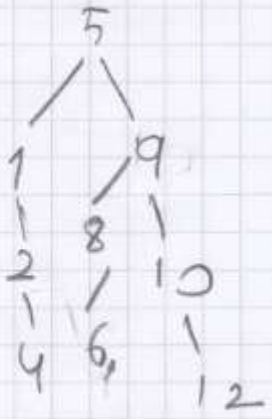




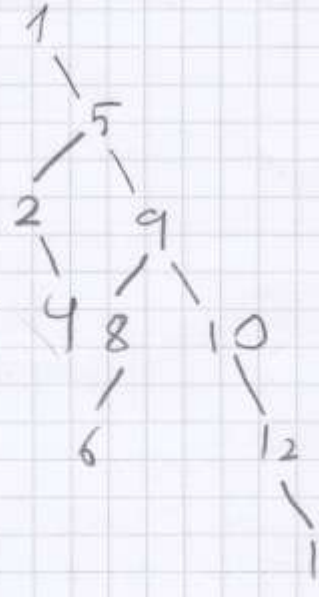


zigzag
(zig)
zigzag
!! 6 küçük olduğu için solda
zig-zig 1) önce parentla grandparent
2) sonra x ile parent

tekrar zig-zig

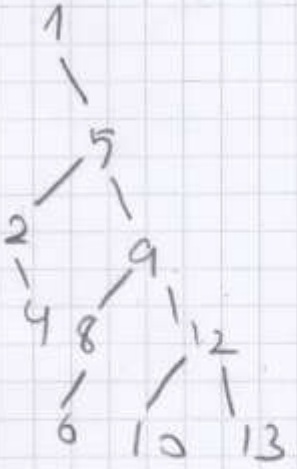


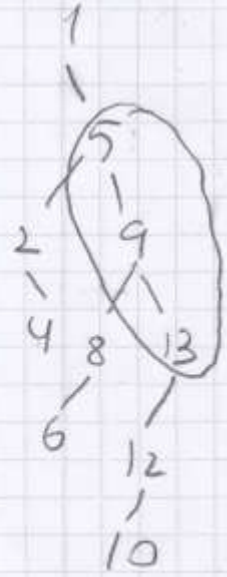
5'in sol çocuğu 9'un sol çocuğu olcak.

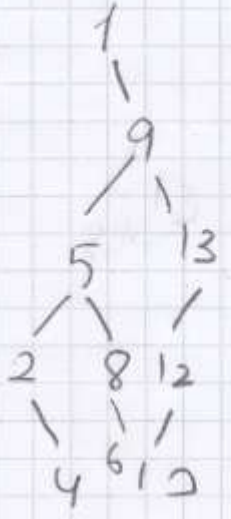


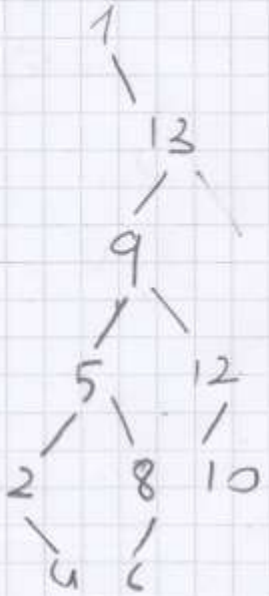
12'ile 10 yerdeştirecek

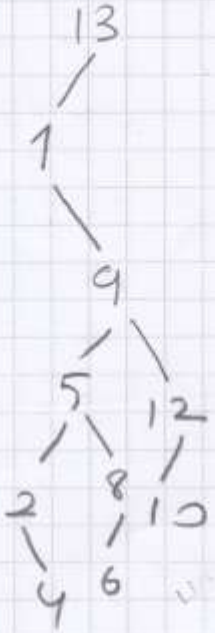
12 10 un yerine gelcek diğeri uygun yere



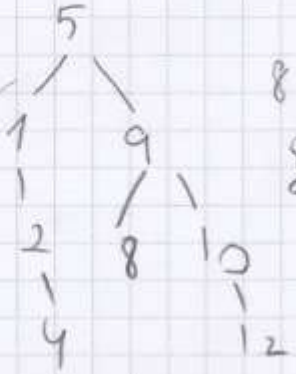








⇒ !!!



8 e yer arıyoruz

8 5 den büyük sağa

9 dan küçük sola