KARADENİZ TEKNİK ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

BIL 205 VERİ YAPILARI DERS NOTLARI

2014-2015 Güz Dönemi

19.09.2014
CUMA

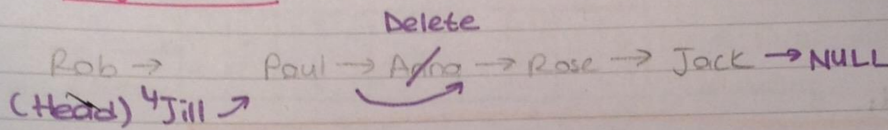
VERİ YAPILARI (Data Structures)

$n! = n.(n-1)!$
$f(n) = n.f(n-1)$

```
int f (int n)
{
  f (n==0) return 1
  else return n*f(n-1)
}
```

* Hiçbir şeye işaret etmeyen pointer NULL'a eşitlenir. 106
Boşluk oluştur nextler birbirini göstersin.

Bağlı Liste:



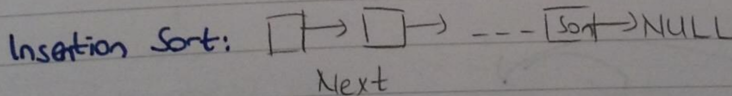
(Singly Linked list)   head = head→next

** Jonathan Shewchuk
* Jrs @ congieres

CHAPTER 3:

Name	Mike	Rob	Paul	Anna	Rose	Jack	
Score	1105	750	720	660	590	510	

Singly Node: Bağlı liste elemanlarının her birini temsil eder.

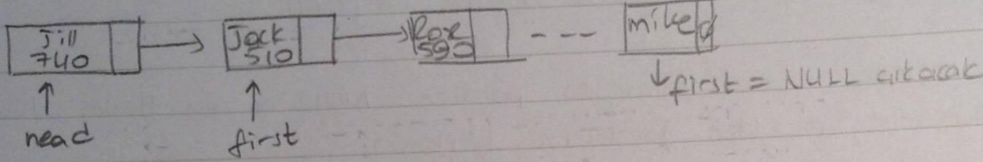
Insertion Sort: 

Singly linked list: Tek yönlü bağlı liste.

FABER-CASTELL

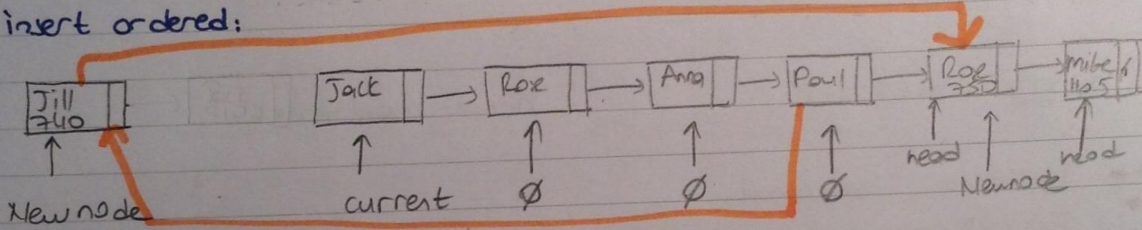
19.09.14

- Insert order: sıralı ekler
- addfront: Başına ekler
- remove front: siler. → parametre almıyor.



→ sağdaki pointer hangi nesneyi işaret ediyorsa soldaki de aynısını işaret eder.

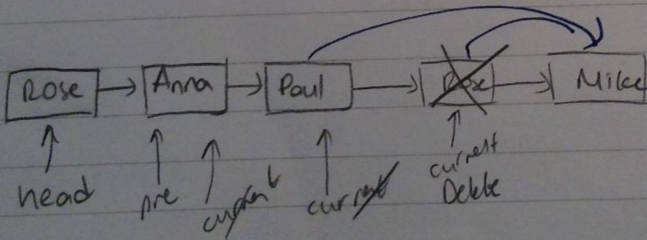
+ insert ordered:



- Head kime işaret ediyorsa current de ona işaret eder.
- Tek yönlü head pointer → NULL set.

Remove ordered:

head = NULL ise boş bir listeden bir şey silemezsin. Jack silinirse başlangıç Rose olur.



Tek başına previous?
Ne bu.

* 3,12 Insertion Sort'dan sorumlu değiliz. FIFO (First In First Out)

Big-O Notasyonu

19.09.2014

C: Statik dizilerin dezavantajları, teker teker kaydırmak (fiziksel sıralama olduğundan)

Doubly Linked Lists; (Çift yönlü bağlı liste)

Insertion → Jill'i eklemek için bir indis sağa eklemek.

new → yeni structlar oluşturmak ve bellekte yer açmak.
bellek alanı tahsis edilir.

```
GameEntry *entries = new GameEntry[maxEntries]
GameEntry entries[max Entries]
```
↳ C, izin vermezdi.

entries → | GE | GE | – | – | – | | | | | GE |
0 1 2 3 4 5 6 7 8 9

* Dizi doluysa bile yeni eleman alabiliriz.

i → sağ
sol ← j

3.2. Tek Yönlü Bağlı Liste

1) Singly Linked (Tek yönlü Bağlı Liste)
2) Double Linked List (Çift " " ")
3) Circular Linked List (Dairesel " " ")

Singly Linked List

Oğuzhan 1500 → next → Ömer 1000 → next → Fatih 1250 | Ø → NULL

head

node = düğüm

```
#include <iostream>
#include <string>
using namespace std;
struct GameEntry
  {
    string name ;
    int score ;
  };
void main()
{
    GameEntry gEntry;
    gEntry.name= "Omer";
    gEntry.score =1000;
  cout<< gEndry. name<< "\t" << gEntry.score<< endl;
      Game Entry* pEntry= &gEntry;
      getchar();
}
```

```
pEntry -> name
 *p= ^
```

Insert Order → küçükten büyüğe sıralama

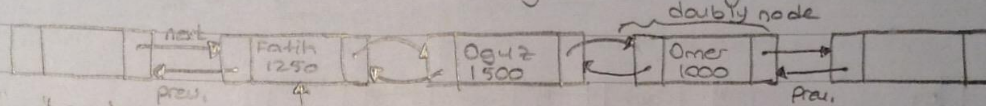
#VERIYAPILARI

26.09.14
CUMA

Çift yönlü Bağlı Liste

* hem "next" hem de "previous"u var (önceki ve sonraki)

Hem başını hem de sonunu gösteren düğüm (pointer değil),



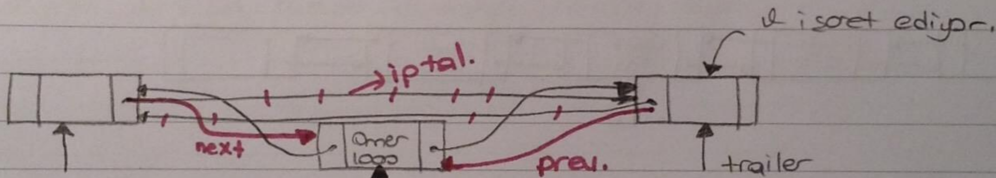
('header' da bir (head) listenin başındaki düğüme işaret eden pointer. '

trailer → pointer
↘ listenin sonundaki düğüme işaret eden pointer

* header'ın next'i işaret ettiği düğüm
* pointer sayısı ikiye çıktı.

→ DoublyNode* header;
DoublyNode* trailer;

v işaret ediyor.

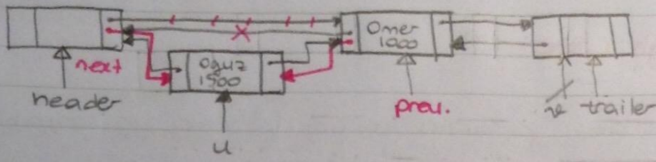


header'ın next'ini parametre olarak yolluyor.

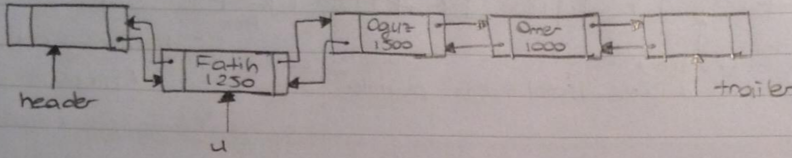
u

* add fonksiyonu v'nin işaret ettiği v=header→next düğümden önce düğüm ekler. Yeni ekleyeceğimiz elemana işaret eder "u" pointerı.

26.09.14

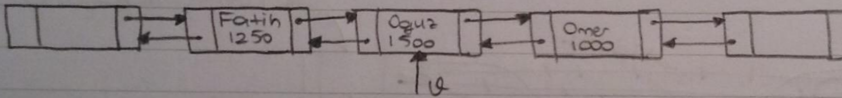


58) v → prev → next   bu üç pointer'dan en son yazılan "next"i setler.
59)



header'ın next'i;

add Front		Addback'in trailer'ı
Fatih 1250	Omer 1000	
Oguz 1500	Oguz 1500	✓
Omer 1000	Fatih 1250	



Addback ile neden trailer'ı çağırıyoruz? (da) trailer'ın previousunu çağırmıyoruz? sınavda vardı. Add'lerin çağrılması farklı → Addback'e bak.

Remove Fonksiyonu

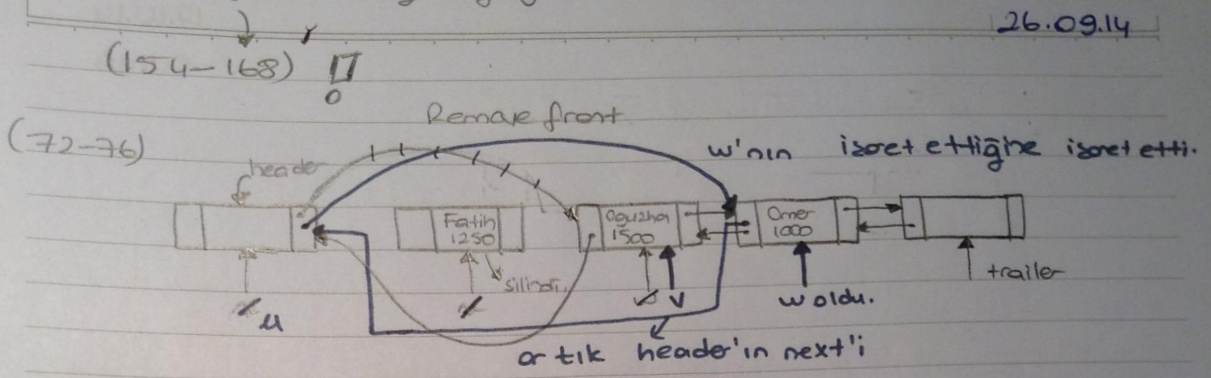
Remove From → Parametre olarak verilen pointer'ın işaret ettiği düğümü siler.
Remove Back → Listenin en sonunda işaret edileni siler.

(62-67.) satırdaki remove'un içindeki neyi işaret ederse o silinir.

```
    remove (header → next);
}
void DoublyLinkedList :: removeBack()
    remove (trailer → prev);
}
```

F11 → kodunun olduğu yere
F10 → fonksiyonun çağrıldığı yere

26.09.14



Fatih 1250 | Oguzhan 1500 | Omer 1000

header

trailer

list.sumScores();

list.maxScore();

list.minScore

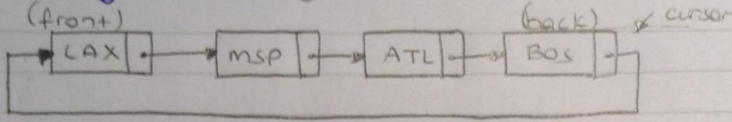
class nesne alırken çalışır → constructure

front → listenin başı gibi işaret ettiği düğüm back.

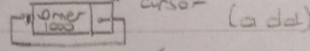
17.10.14
CUMA

advanced cursor cursor→next

Dairesel - Bağlı Liste (sy129)



front(); Return the element referenced by the cursor; on error results.



(add)

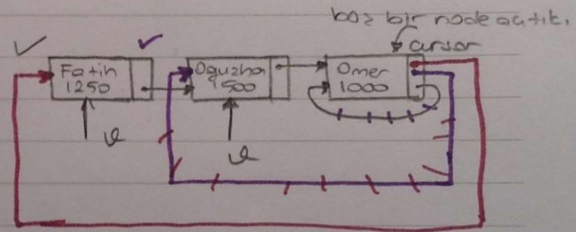
add() → yeni bir liste eklemek için (liste boş ise cursor kendine işaret eder, next'i de kendisine işaret eder.)

remove → eleman silmek için

* cursor'ın next'ini ekliyor (tek yönlüdeki addfront gibi). cursor'ın next'i ne (front)

* düğüm sayısı tek ise kendini değilse next'ini siler. Son elemanı da sildikten sonra "NULL"a setler.

list.add ("Omer", 1000);
("Oguzhan", 1500);
("Fatih", 1250);

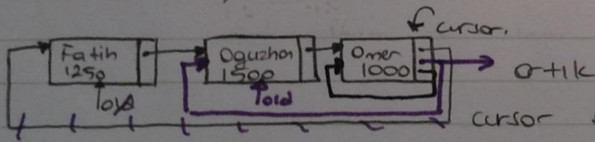


otomatik olarak

* Remove parametre almaz (cursor'ın next'ini siler)

Remove için;

list.remove();
();
();



cursor olarak 'Fatih'e erişemiyoruz. Bu yüzden 'old' la ulaşıyoruz.

İlk sileceği eleman "Fatih" olur (next'i olduğu için).

Son eleman silinirken old ve cursor aynı

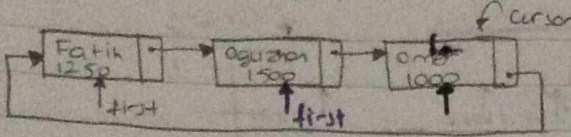
True, döndürün

cursor = NULL olur ve Omer de silinir.

FABER-CASTELL

Queue → enqueue → add
→ kuyruk veri yapısı
→ dequeue (remove)

17.10.14
CUMA



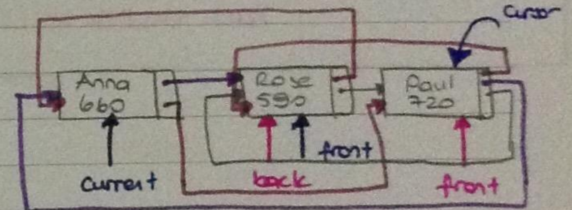
75) first cursor temsil etmediği sürece
first = first -> next;

Fatih 1250
Oguzhan 1500
(first cursor'a eşit olduğundan çıktık) Omer 1000

remove'dan sonra print yazarsak;
Oguzhan
Omer
"Omer"i de silersek;
"list is empty!" der.

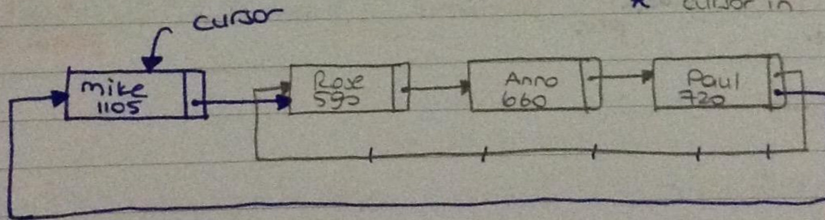
97'den önce ekliyoruz. Ardından uygun yeri bulduktan sonra o yere taşıyoruz.

	e	i	
list.InsertOrdered	("Paul"	, 720)	✓
	("Rose"	, 590)	✓ 97.
	("Anna"	, 660)	
	("mike"	, 1105)	
	("Rob"	, 750)	
	("Jack"	, 510);	
	("Jill"	, 740);	



sırayı bozdu; False döndürür.

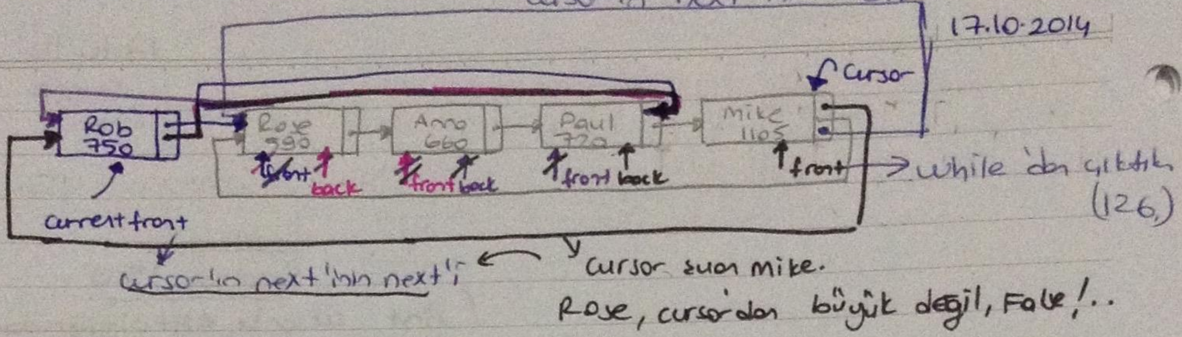
97. satır
* "cursor'ın" "next'inin" "next'i.
119 → False



* Add fonksiyonu; "Mike"'ı ekledi. düzenlenmiş hâli

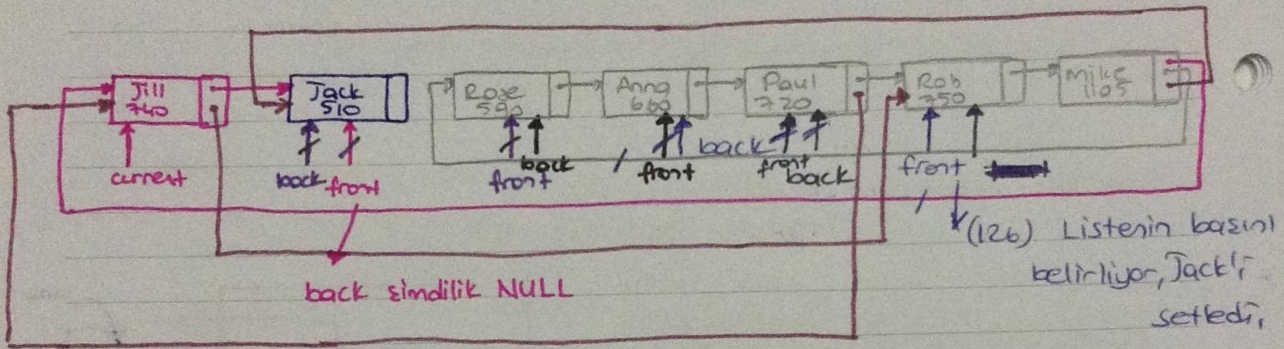
bu aslında şu demek; "Rose" un 'next'i "Anna";
'cursor'ın "next" inin 'next'i

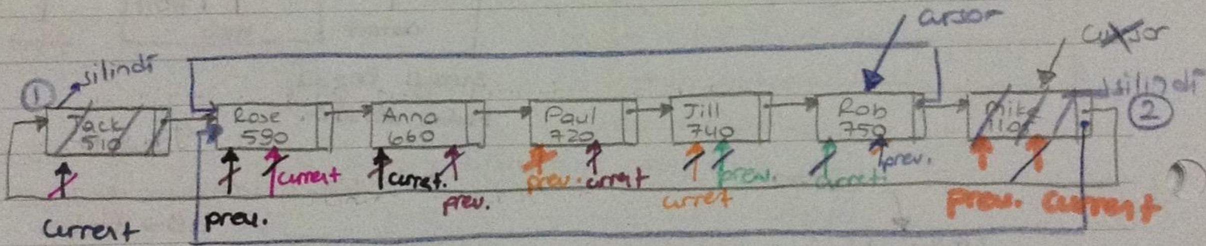
17.10.2014



Rose, cursor'dan büyük değil, False!..

Rob'un peşine Mike.
Paul'un next'i Rob.
Rob'un next'i Mike.
Mike'ın next'i Rose.





current
↳ 153'te getirdik.

silme işlevi için; iki yardımcı "pointer"ımız var.

list. remove Order ("Jack", 510);
("Mike", 1105);
("Paul", 720);

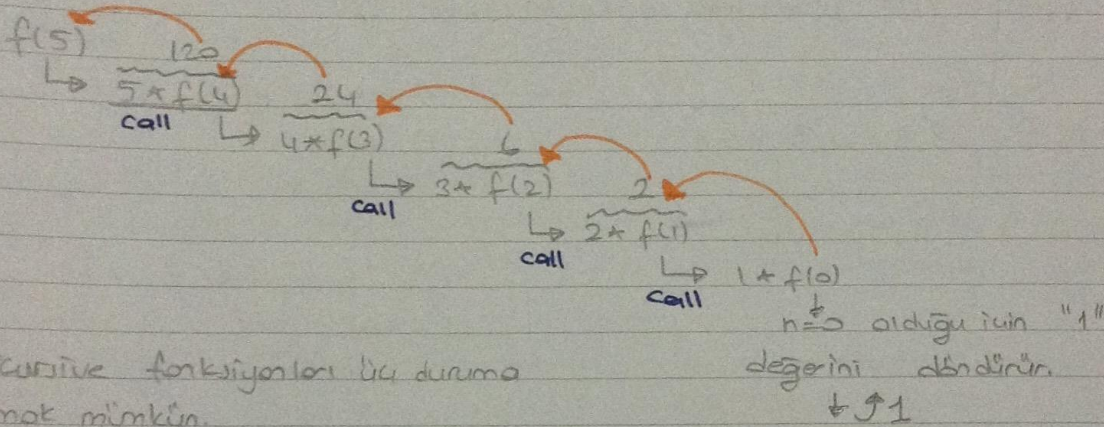
FABER-CASTELL

31.10.2014
CUMA

Recursion (Öz Yineleme)

Fonksiyon kendi içerisinde farklı bir parametreyle kendini çağırır.
Recursion'ın "Hello world!"ü ↷
(factorial)(n) = n * factorial (n-1) (recursiona uygun bir hesap)

```
int factorial (int n)
{
    if(n==0) return 1  → base case ← yoksa sonsuz döngüye girer. recursive kısımları sonlandıran çağrı
    else return n * factorial (n-1);  // recursive case
}
```



Recursive fonksiyonları üç duruma ayırmak mümkün.

1) **Linear Recursion** → Fonksiyon kendisini bir noktadan recursive çağırır.
2) **Binary Recursion** (iki noktada kendini recursive olarak çağırır).
3) **Multiple Recursion** n sayıdaki noktadan kendisini recursive çağırır.

Linear Sum

A[] = {1, 2, 3, 4, 5, 6, 7, 8}
n 0 1 2 3 4 5 6 7

* A dizisini alıyor; dizinin elemanlarının toplamını döndürüyor.

$\frac{8*9}{2} = 36$

Sum(A, 8)
↳ 8 + Sum(A, 7)
↳ 7 + Sum(A, 6)
↳ 6 + Sum(A, 5)
↳ 5 + Sum(A, 4)
↳ 4 + Sum(A, 3)
↳ 3 + Sum(A, 2)
↳ 2 + Sum(A, 1)
↳ 1

1'i döndürür.

FABER-CASTELL

dizi elemanlarını büyükten küçüğe sıralıyor → "reverse" yapıyor 31.10.2014

Reversing an Array by Recursion → "Lineer Recursion" örneği

A[] = {1,2,3,4,5,6,7,8}

```
if (i<j)
{
   int temp = A[j];
   A[j] = A[i];
   A[i] = temp;   → 8'in yedeğini tutuyor. 8'i yeniden yazmamız için.
   return reverseArray (A, i+1, j-1);
}
   for (int i=0; i<16; i++) cout<< A[i] << " ";
}
```

A[] = {8,7,3,4,5,6,2,1}

temp = 6
A[2]=3
A[6]=5

A[] = {8,7,6,5,4,3,2,1}

temp=5
A[3]=5
A[4]=4

* Dokuz elemanlı bir dizide ortadaki dizi sabit kalır; kaydırma olmaz.

Tail Recursion → Lineer Recursion'ın özel bir hâli.

if(i<j) → while döngüsüne dönüştürülür.

Recursive çağrı fonksiyonun son tarafında yapılırsa; tail recursion olur.

```
j = j-1
i = i+1   → şeklinde kod satırı olarak ekler.
return reverseArray (A, i+1, j-1)
```

(i+1, j-1 → silinip → dışarıdan)

FABER-CASTELL

31.10.2014

while döngüsüne çevrilir;

```
while (i<j)
{
   int temp= A[j];
   A[j] = A[i];
   j = j-1
   i = i+1
```

* (+)'dan önceki recursive koşar. İlk önce return"ün peşine gelen çağrılır.

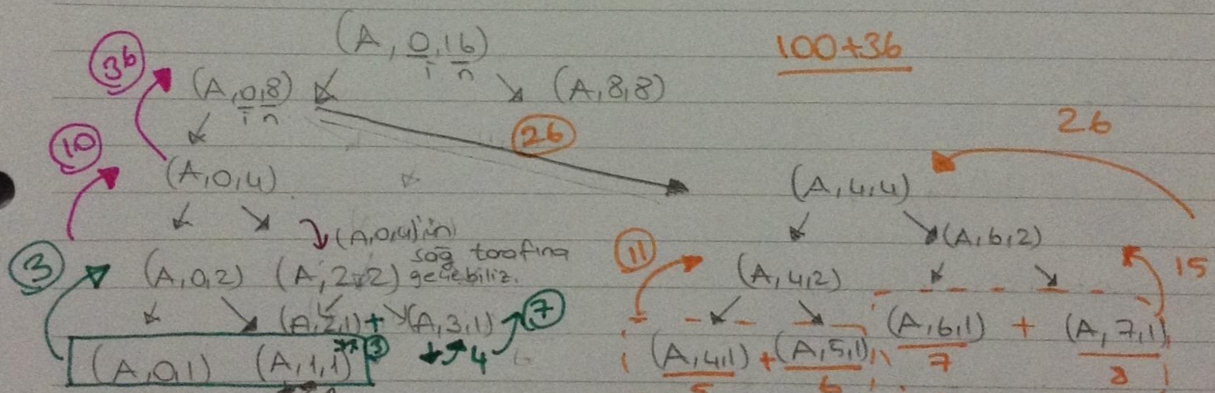
Binary Recursion (iki noktadan kendini çağırır) * linear sum'ın binary versiyonu

* içerdeki return'lerde de (+) dan önceki sonrası olacak.

A[]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}

(A,0,16)

```
int binarySum(int A[] ,int i ,int n)
{
   if(n==1) return A[i];
   else return binarySum (A,i,n/2) +
   binarySum (A, i+N/2, n/2);
}
```

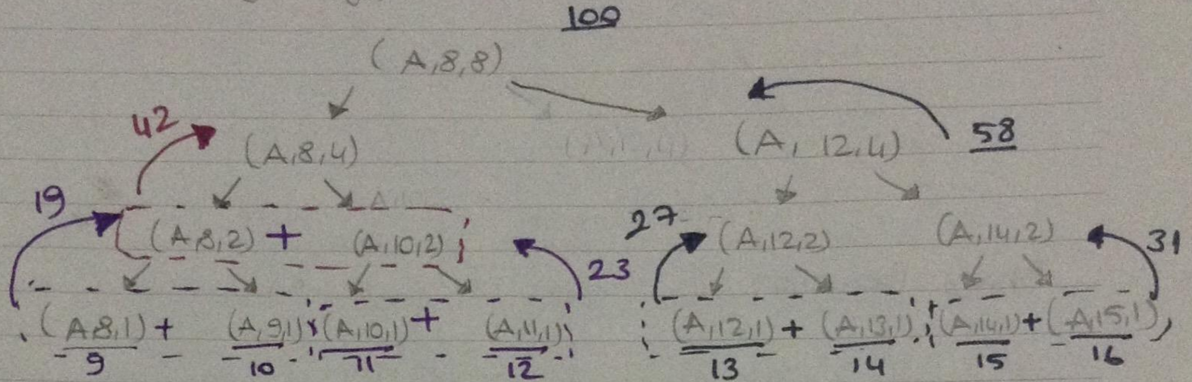


n==1 olduğu için A[0] ! (A[i] return yapar).

↙ Artık A'nın sağ tarafına geçebilir.

31.10.2014
CUMA

(A,0,16)'nın sağ tarafı;



3
7
10
11
15
26
36
} sınav sorusuydu.

n=0 için

fib(0) = 0
fib(1) = 1
fib(2) = fib(1) + fib(0)
fib(3) = fib(2) + fib(1)
fib(4) = fib(3) + fib(2)

0 1 1 2 3 5 8 13 21

f(10) = 55

1'den 10'a kadar olan sayıların toplamı = 55.

31.10.2014

a b n
f(0, 1, 10)

→ içeri girdiğimizde ikinci terim 1. terim oluyor ve n'i 1 eksiltiyoruz.

fibonacci sayıları

f(1,1,9)

f(1,2,8) → o anki fibonacci değeri.

f(2,3,7)

f(3,5,6)

↳ n, 2 olacak.

Böyle ilginç bir yapıya sahip işte... Mukadderat. :)

f(5,8,5)

f(8,13,4)

f(13,21,3)

f(21,34,2)

f(34,55,1/n)

*"return yapılan değerler" bir değişkene atmıp toplanmıyor.

31.10.2014
CUMA

(A, 0, 27) 378

(A, 18, 9)

(A, 24, 3) 78
(A, 26, 1) + (A, 25, 1) + (A, 24, 1)
27 26 25

(A, 21, 3) 69
(A, 23, 1) + (A, 22, 1) + (A, 21, 1)
24 23 22

(A, 18, 3)
(A, 20, 1) (A, 19, 1) (A, 18, 1)

(A, 9)

(A, 15, 3) 51
(A, 17, 1) + (A, 16, 1) + (A, 15, 1)
18 17 16

(A, 12, 3) 42
(A, 14, 1) + (A, 13, 1) + (A, 12, 1)
15 14 13

(A, 9, 3) 33
(A, 11, 1) + (A, 10, 1) + (A, 9, 1)
12 11 10

1. 78
2. 69
⋮ 60
207
51
42
33
126
24
15
6
\+ 45
378

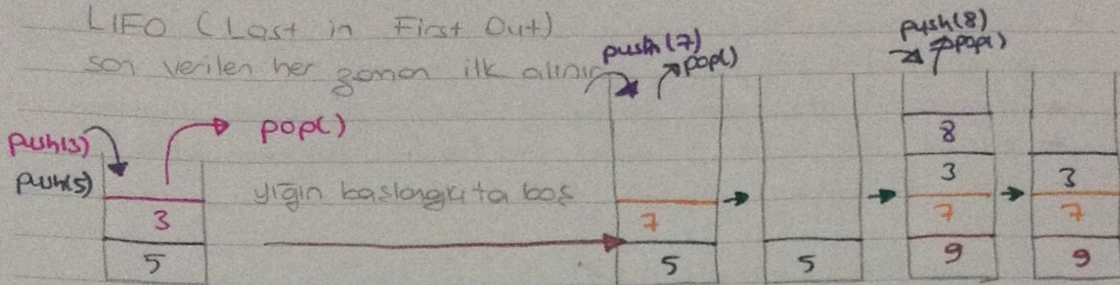
STL → Standart Template Library
Standart Şablon Kütüphanesi

Big O Notation

STL kütüphanesi 'C++' ın içinde. 07.11.2014 CUMA

Stacks

LIFO (Last in First Out)
son verilen her zaman ilk alınır.



* Pop yığından elemanı çeker, parametresizdir.
* Pop elemanı yığından eleman çeker, parametresizdir.
* tepedeki elemanı siler.
* Top; yığının tepesindeki elemanı "return" yapar.
* Dört elemanı olduğu için size "4" değerini döndürür.

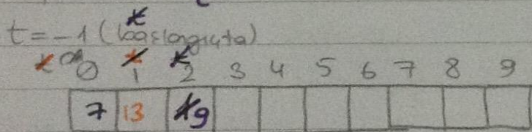
top() ⭑ boş elemanı çekmek hata döndürür. Yani 5

5.1.4 A simple Array-Based Stack

basit, tek boyutlu dizideki yığın veri yapısıdır.

int *S; → tek boyutlu diziyi adresler.
int capacity; → dizinin max. eleman sayısını tutacak.
int t; → yığının tepesine işaret edecek (index of the top of the stack)

t = -1 (başlangıçta)

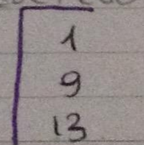


S[++] ↳ t'yi ilk önce artırır. "t=0" olur.

t; diziye eklenen en son elemanı işaret edecek.

```
A.push(7);  ✓
A.push(13); ✓
A.push(1);  ✓
cout << A.top() << endl;
```
↳ 22. satırda (1 değerini döndürür. yığının tepesini işaret eder.)

```
A.pop(); → t'yi 1 eksiltir.
A.push(9); ✓ t'yi 1 artırır tekrar.
cout << A.top() << endl;
A.pop()
cout << A.top()
```



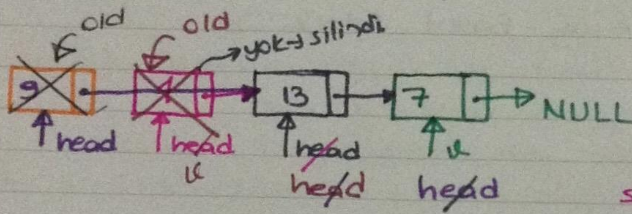
Program çıktısı

FABER-CASTELL

07.11.2014
CUMA

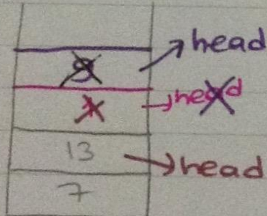
★ Aslında çok kullanışlı bir durum değil.
★ Onun yerine tek yönlü bağlı liste kullanacağız.
★ Singly-Node bir araya getirerek bağlı listeler oluşturuyoruz (Singly Linked List).
★ push(7) yaparken arka planda "add" işlemi uygulanıyor.
★ Front fonksiyonu "head"in işaret ettiğini döndürür. Listedeki ilk eleman yığının tepesi olur.

```
LStack.push(7); ✓
      .push(13); ✓
      .push(1); ✓
cout<< LStack.top() ✓
      .pop()
      .push(9)
cout<< .top()
      .pop()
cout << .top()
```



s.front → head pointer'ı işaret ettiği elemanı döndürür. (listenin ilk elemanı, yığının tepesi)

≡



1
9
13

ekran çıktısı

push() = addFront()
pop() = removeFront()

07.11.2014

CUMA

```
# include <stack>
using std::stack; // make stack acce
Stack <int, myStack ; // a stack of integer
```

↳ "int" kullanacağımız için "int" yazdık.

STL'deki yığın veri yapısı içerisinde sadece int değil başka türden yığınlar da vardır. "class"'tan türetilmiş nesneler de içerir.

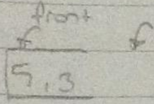
I. VİZE SONRASI

21.11.2014
CUMA

Queues (Kuyruklar)

First in, First out mantığına göre çalışır (FIFO).
Ekleme silme işlemleri olarak düşünürsek; ekleme işlemi sonda, silme işlemine baştan başlıyoruz.

enqueue : ekleme
dequeue : silme (boşsa hata mesajı verilir)
empty : "Kuyruk boş mu değil mi?"



dequeue yapılınca "5" silinir 3 baş eleman olur.

Example 5.4.
dairesel bağlı liste kullanılacak.

Yeni eklenen eleman listenin son elemanı olunca cursor'ın next'ine eleman ekliyoruz ve cursor'ı bir adım ilerletiyoruz. Ve artık yeni elemanı cursor göstermiş bulunuyor (Kuyruk veri yapısında).

in Figre 5.5

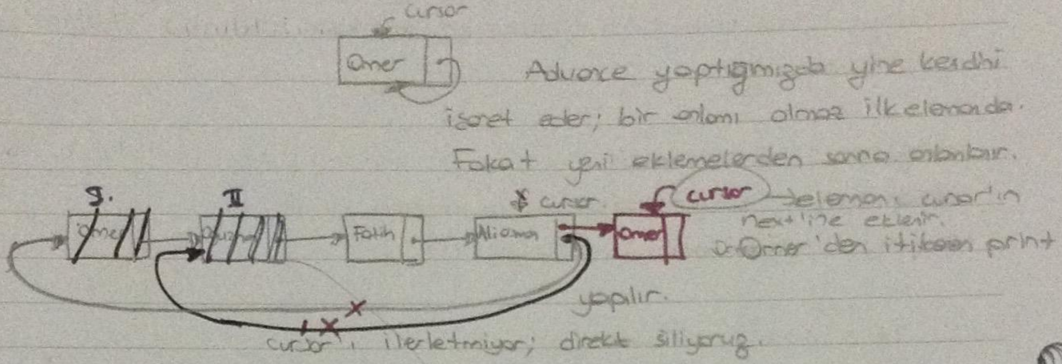
```
void CircularlyLinkedQueue::enqueue(const string& e)
{
    C.add(e);
    C.advance( );
    n++;
}
```

Advance: cursor'ı ilerletme.

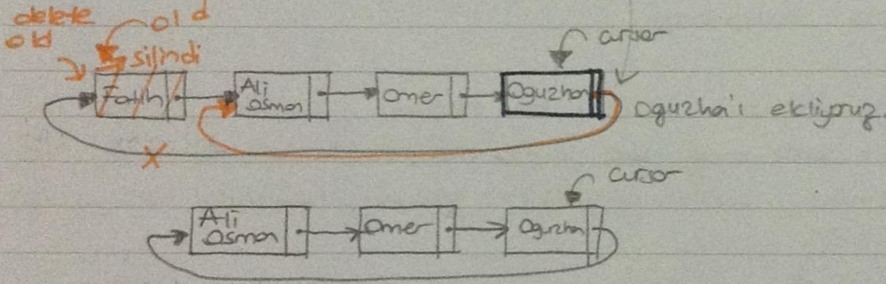
21.11.2014
CUMA

Dequeue:

Dairesel bağlı listenin remove'u kullanılır. Cursor'ın next'i silinir. Cursor'ın next'i ilk elemandır (ilk eleman silineceği için).



Dairesel bağlı listede listenin başına ekliyor; başından siliyorduk. Kuyruk veri yapısında ise başına ekliyor; sonundan siliyoruz.



*Kuyruğun sonuna geldik.

21.11.2014

Double Ended Queues → Türkiye'deki kuyruklara güzel bir örnek.

Kuyruğun hem başından hem de sonundan silmeyi destekler. Karıştırılmaması için önündeki ue'ler silinmiş :). Sonuna veya başına ekleme işlemi de destekleniyor.

*Çift yönlü bağlı liste kullanılabilir. Çift yönlü bağlı listeye ait kodlar kullanılabilir (node, doublynode).

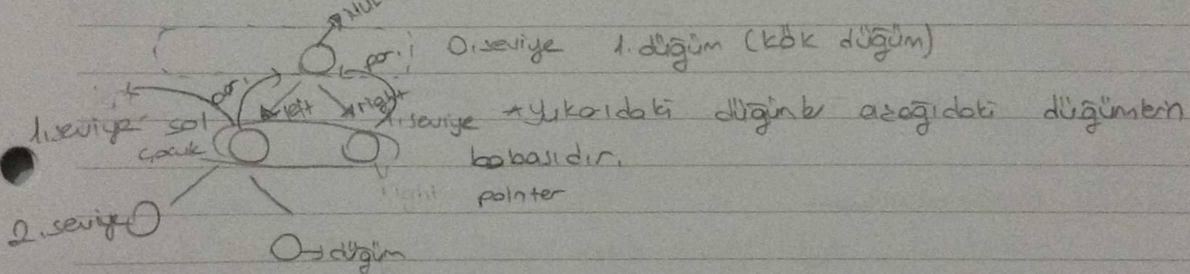
— CHAPTER 6 GEÇİLDİ —

CHAPTER 7

— Trees —

Ağaçlar (Binary Trees)

Her bir düğümün iki tane çocuğu var (hiç çocuğu da olmayabilir).



Maksimum düğüm sayısı = 2^n

n = seviye sayısı

addRoot → sadece kök düğümü oluşturacak.

expandExternal → oluştururken kullanılacak.

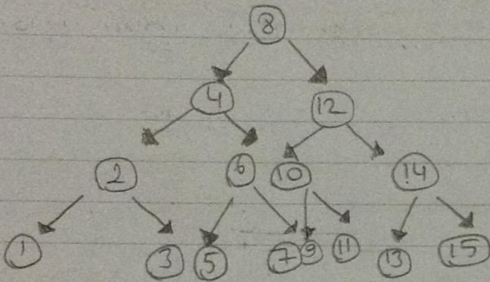
Chp. 10'da düğüm silme konusundan bahsedilecek.

21.11.14

preorder } recursive fonksiyonlar
inorder } gezinerek ağaç üzerinde, ağaçta gezinme
postorder } yapılacak.

ilk okunan sayı ağacın kökü olacak
küçükse sola
büyükse sağa
kuralını biz belirliyoruz

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



Not; Read me!

```
void LinkedBinaryTree:: addRoot ()
{
    root = new Node;
    n=1;
}
```

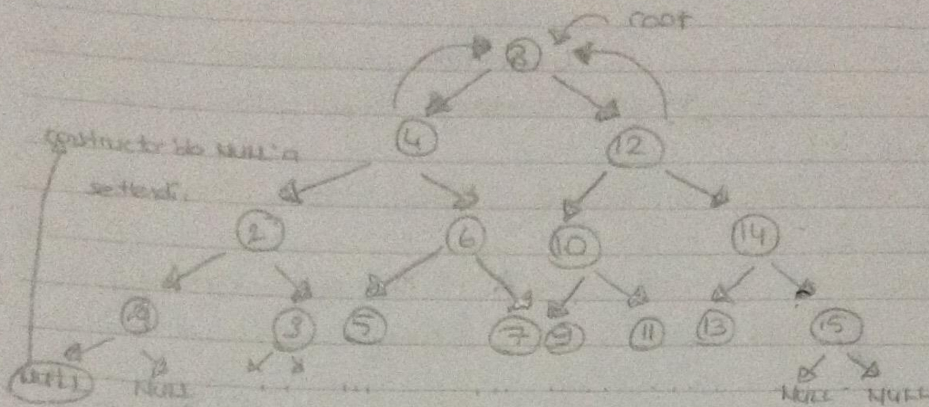
root → tek yönlü bağlı listedeki headpointer gibi

```
void LinkedBinaryTree:: expandExternal (Node* v)
{
    v->left   = new Node
    v->left->par
```

FABER-CASTELL

LRP → post order
LPR → Inorder traversal
PLR → preorder traversal

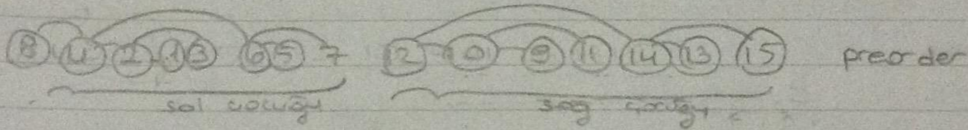
21.11.2014

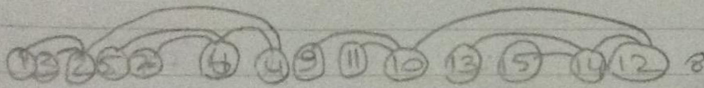


* 4'ü silip yerine 6'yı yerleştirdiğimizde 2 5'in sol çocuğu; 6 değil de 2'yi yerleştirdiğimiz zaman ise 6 3'ün sağ çocuğu olur.

Inorder ağaç elemanlarını küçükten büyüğe doğru sıralar (küçükse sola; büyükse sağa).

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





28.11.2014
CUMA

Heap & Priority Queues

Öncelik STL'deki bağlı listeler kullanılmaz kitaptaki kodlarda.
Bağlı listedeki InsertOrder'ın tıpa tıp aynısı.

removeMin = dequeue
Insert = enqueue → verileri sıralı işleyeceği için içinde sıralama algoritması var ve karşılaştırma yapması gerek.

```
bool ListPriorityQueue::isLess(const int& e, const int& f) const
{
(55) if (e<f) return true;    → 2<3 mü? e=2 f=3 iken
     else return false;
}
void ListPriorityQueue::insert(const int& e)
```

dögünün iterate'i olur; döner.
iterate= ilerlemek

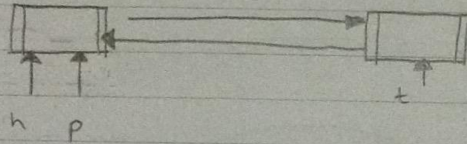
28.11.2014

```
{
std::list<int>::iterator p   → pointer değişkeni (bağlı liste elemanlarına işaret eden)
p= L.begin ();
while( p!= L.end() && !isless (c , *p) ) ++p;
  L.insert (p,e);
}
```

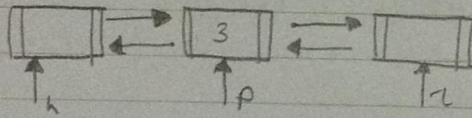
! liste boşken while çalışmaz.

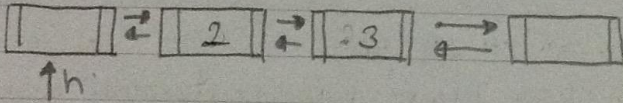
Priority Queue .insert(3); → 3'ü insert yaparken;
Priority Queue . insert(2);
Priority Queue . insert(1);

Çift yönlü bağlı liste olarak düşünürsek;



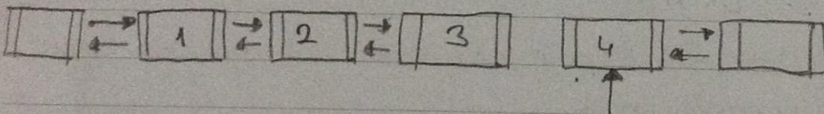
* 2'yi eklerken while'a gireriz.





* Tek yönlü bağlı liste de kullanılabilir.
* Çift yönlü bağlı listedeki insert order'ın aynısı.

e=4
*p=



* 4' 1'den küçük değil False; tersini döndüreceğimizden True olur.

* Insert işleminde farklılık var. (remove kısmı aynı)
Dequeue işlemi kuyruğun başından yapılıyordu (bağlı listedeki removeFront gibi)
Equeue farklı döndürme algoritmasından mütevellit.

FABER-CASTELL

minHeap → küçükse swap yapılır.
maxHeap → büyükse swap yapılır.

28.11.2014

void PriorityQueue :: print()
{
// yaz ve ilerlet; gayet kolay. :)

* Yığın veri yapısı ctrl-z yapılırken kullanılabilir.

İlk yapılan isteği ilk; son yapılan isteği son gerçekleşebilmesi için ala planda mesaj kuyruğu var oraya atılıyor.

8.3 Heaps → Kuyruk veri yapı ile ilgili.

Kuyruğu liste değil; ağaçta tutacağız. İşlemi hızlandırmak amacıyla. Minimum değeri ilk indiste toplamak için priority kullanılır. Ağaçtaki minimum eleman root'ta toplayabilmek için kullanılır silme işleminde. RemoveMin root'taki elemanı siler. Öncelikli kuyruğun ağaç kullanılarak gerçeklenmiş hâli.

→ minimum değer root'ta

* Soldan sağa doğru düğümlere bakılır ve ilk boş kısma eklenir. Öncelik sol kısımdır.

root'a çıkabilir mi?

* Babasından küçükse swap yapılır. 2<20 ile swap olur.

* Sıfırdan ağaç oluştururken de bu adımlar kullanılır.

* Ekleme sırası. Babadan küçükse swap'lar yapılır.

if (!less ---
↓
ünlem kaldırılırsa maxHeap olur

removeMin() → root silinir.

minHeap → küçükse swap yapılır.
maxHeap → büyükse swap yapılır.

28.11.2014

void PriorityQueue :: print()
{ yaz ve ilerlet; gayet kolay. :)

* Yığın veri yapısını ctrl-z yapılırken kullanılabilir.

İlk yapılan isteği ilk; son yapılan isteği son gerçekleyebilmesi için ortamda mesaj kuyruğu var oraya atılıyor.

8.3 Heaps → Kuyruk veri yapısı ile ilgili.

Kuyruğu liste değil; ağaçta tutacağız. İşlemi hızlandırmak amacıyla. Minimum değeri ilk indiste toplamak için priority kullanılır. Ağaçtaki minimum elemanı root'ta toplayabilmek için kullanılır silme işleminde. RemoveMin root'taki elemanı siler. Öncelikli kuyruğun ağaç kullanılarak gerçeklenmiş hâli.

→ minimum değer root'ta

root'a çıkabilir mi?
* Babasından küçükse swap yapılır. 2<20 ile swap olur.

* Soldan sağa doğru düğümlere bakılır ve ilk boş kısma eklenir. Öncelik sol kısımdır.

* Eklenme sırası. Babadan küçükse swap'la yapılır.

★ Sıfırdan ağaç oluştururken de bu adımlar kullanılır.

if (!) less - - -
ünlem kaldırılırsa maxHeap olur

removeMin() → root silinir.

* Vektör'ü kullanarak ağaç veri yapısını gerçekleriz. 28.11.2014-CUMA

13 7'nin sağ çocuğuydu 4 silindi, 13 geçti. Ama hala roottaki eleman minimum değil. Onun için 13'ün sol çocuğu olan 5 ile 13 yer değiştirir.

BABA ÇOCUK İLİŞKİSİ

```
struct VectorCompleteTree                    std::vector<int> V;
{
    typedef std::vector<int>::iterator Position;
    VectorCompleteTree() : V(1) {}
    int size const              {return V.size() - 1;}
    Position left (const Position& p)   {return pos(2*idx(p));}
    Position right (const Position& p)  {return pos(2*idx(p)+1);}
    Position parent (const Position& p) {return pos(idx(p)/2);}
    bool hasLeft(const Position& p) const {return 2* idx(p) <= size();}
```

hasLeft → sol çocuğu var mı?

pushback ≃ Removeback gibi.

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

*v < *u

* 0'dan da başlayabilirdik; ozaman 0'ı da başardı.

— o — ikiyi eklerken;

e=2

```
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;
}
}
```

sona ekledi 2'yi.

while'dan çıktık.

* Insert işlemi 2'yi yukarı çıkarır.

28.11.2014

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

* Kodlarda ağaç yok yerine vektör mevcut. Vektörün ağaç versiyonudur.

12<14
12<13

6<5'in işaret ettiği

05.12.2014
CUMA

HASHING (CHPT 9) → relative addresing
(bağıl adresleme)

Sözlük

strcmp(str1,str2) iki string'i karşılaştırmada kullanabiliriz.

Verilerin aritmetik sıralamaya göre kullanılması pek kullanışlı değil.

dictionary.txt

key

(key) fseek

relative.txt

* Sorgulama çok hızlı. Aramayı İngilizce kelime olarak yapıyoruz ve hashing'te parametre olarak gönderiyoruz. ASCII karakterlerini de kullanıyoruz.

fseek() = hash fonksiyonun döndürdüğü adres. Direkt adresten kelimeyi okuyup cevap verebiliyoruz.

FABER-CASTELL

05.12.2014
CUMA

* Adresi yazmıyoruz; adrese gidiyor ve kelimeyi buluyoruz.
Çakışma Çözümleme:

1) Lineer Probing
2) Double Hashing'ten sonra yeniden Lineer Probing yapabiliriz.

* relative.txt oluşturduktan sonra bir kereye mahsus kullanıyoruz sadece o da dictionary.txt'tekileri yazmak. dictionary.txt ile daha işimiz kalmıyor. Çakışma ihtimalini düşürmek maksadıyla relative.txt dictionary.txt'e göre daha geniş tutmamız önerilir.

```
20  struct word
    {
      char ingilizce[12];
      char turkce[14];
    } kelime;
    int Hash (char *key)
    {
      int sum=0;
      for( int j=0; j<4; j+2=1
          sum= (sum+10*key[j] +key[j+1]);
          sum = sum % nRel;
          return sum;
```

% → genelde asal sayı alır. 40'tan büyük ilk asal sayı 41

ilk olarak 78. satırdaki

```
void RelativeOlustur()
{
   int collisions=0;
   dic= fopen ("dictionary.txt", "r");
   rel= fopen(" relative.txt", "wt");
   for (int i=0; i<nRel; i++)
   {
      fseek (rel, i*sizeof(kelime), 0);
      fputc ('*', rel);
   }
```

nRel → kapasitesi

* özel bir karakter basıyor (*) adresin boş olup olmadığını sorguluyor. "*" varsa boş olduğunu anlayacağız.

* Dosyanın başından "i* sizeof"un döndürdüğü byte kadar ilerle.

ambiguous = belirsiz – Otomata

mod izlemi listenin başına götürür.

05.12.2014

CUMA

26 byte ta bir "" karakteri basar.

ASCII j=0

97 a sum = 0*10 * 97 + 109 = 1079

b j=2

c sum = 1079 * 10 + 98 + 105

d sum = 2164

e

f 2464 = 141 mod 32

g

h

i

j

kelimenin İngilizcesini alır ve bir adres üretir.

1079
+ 1085

Bağıl adres

32 = 2164 % 41

a m b i g u o u s
0 1 2 3 4 5 6 7 8

```
Temp = Adres;   -> bütün kayıtlara baktık.
c = fgetc(rel);
while (c != '*')   -> ise doludur.
{
printf ("%d ADRESINE %s\t\t ...
collisions ++;
Adres = (Adres+1) % nRel;
if (Adress == Temp)
{
printf ("DOSYA DOLU ! \n");
return;
}
```

'*' karakteri ile karşılaştırırsak ise Lineer probing yapılır ya da dolu ise while'dan çıkabiliriz.

int sizeofRel -> boyutu, relative.txt'in

Performans Artırma

1) Bucket Addressing
2) Synonym Chaining

Bucket

Dezavantajı çok fazla bellek kullanması gerekmesi.

0	*	*	*	*
1	*	*	*	*
2	*	*	*	*
⋮				
32	/ / /			
⋮				
40	*	*	*	*

```
88. bucketsize[Adres]r = 1;
 ⋮
95. if (bucket size[i] > oBucket ) o ...
```

↑ maksimum

Bucket'ın kayıt tutma kapasitesi

* Synonym Chaining

Çakışan kayıtlar bağlı listede tutulur.

32. adresteki ambiguous > list > nuts > object > NULL gibi klasik hashing için. Çakışma olmayanları tutmak zorunda değiliz. relative.txt'ye yine yazacağız. Çakışanlar için bağlı liste kuracağız; bağlı listeyi dosyaya yazmayacağız bellekte olacak. Dosyada sorgulanmayacağız. → Lineer probing kullanacağız ona bağlı listede sorgulayacağız verinin kendisini. Eklemede yine çakışma olabilir. → bağlı liste.

12.12.2014
CUMA

CHP 10

Binary Search Tree

Ağaç Üzerinde Arama İşlemi

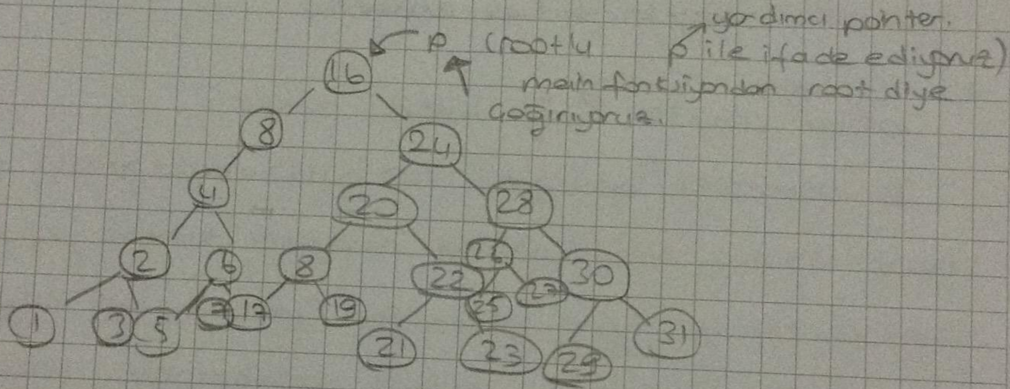
Ağaçta int verilerini tuttuğumuzu varsayıyoruz ama class'tan türetilmiş nesneler de tutabilirdik.

preorder, postorder → ağaç üzerinde gezinmeyi sağlayan recursive fonksiyonlar.

addBelowRoot → ağaca düğüm eklerken root'tan itibaren düğümün nereye eklenmesi gerektiğini biliyor ve oraya ekliyor (küçükse sola büyükse sağa)
→ Binary Search Tree'deki önemli fonksiyonlar.
deleteNode →

İlk oluşturulan düğüm Root düğümü.

newNode → elt=e; → içini dolduruyoruz.



22'nin eklenmesi

addBelowRoot (root, 22)

* Her bir düğümün parent düğümü var.

insert Order → yeni eleman ekliyor.

22, 16'dan küçük mü? değil.

12.12.2014

```
void LinkedBinaryTree::addBelowRoot(Node* p, int e)
{
Node* parent;
while (p!=NULL)
{
parent = p;
 if(e<p->elt)
    p=p->left
 else
       p=p->right;
}

Node* newNode=newNode;
new Node-> e; ← içini dolduruyoruz.
newNode->par=parent;

if( newNode->elt <parent->elt)
    parent -> left=newNode;
 else
     parent->right=newNode;
     n+=1
}
```

—— o ——

```
deleteNode( root, 8)      ← O düğümü silebilmek için ilk önce
deleteNode( root, 7)        bulmamız gerek → bi'nevi "search işlevi"
deleteNode( root, 16)
```

—— o ——

```
void LinkedBinaryTree::deleteNode(Node* p, int e)
{
 Node* temp;
 while(p!=NULL)
 {                          ↗ gelmek istediğimiz düğüme geldik.
  if(p->elt==e)   break; → ilk önce false döner (8'i silmek istiyorsak)
  else                  8'i görünce while'dan çıkarız.
  {                     8'in yerine kim gelecek?
   if(e<p->elt)         - Kendisinden küçük en büyük düğümü kendi yerine koy
       p=p->left;       - Kendisinden büyük en küçük düğümü kendi yerine koy.
   else
      p=p->right;
   }
 }
  if (p==NULL) → silmek istediğimiz düğümü bulamadık.
  {             ağacın son düğümüne kadar gideriz p==NULL'da çıkarız.
   cout << "The node doesn't exist" << endl;
   //getchar();                    böyle bir düğüm yoktur.
   return;
  }
  else
  {
  if (p==root)
```

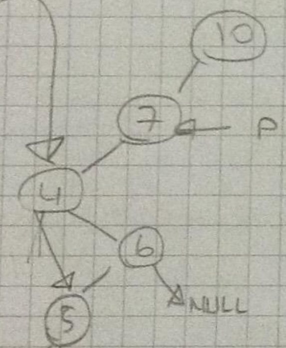
FABER-CASTELL

* ikisi de aynı anda olamaz. ikisi de mantıklı.

```
}
1)  // Kendisinden küçük en büyük düğümü kendi yerine koy   (8'in yerine 7)

    if (p->left != NULL)
    {                       yardımcı pointer
        temp = p->left;   -> önce bir kere sola gitmeliyiz. Kendisinden küçük dediği için.
        while (temp->right != NULL) temp = temp->right;
        root->elt = temp->elt;   -> 6 - 7 while'dan çıktık (7'nin sağ çocuğu NULL)
        if (temp->left != NULL)   -> sol çocuğu yok 7'nin (7'nin sol çocuğu var mı?)
        {
            temp->par->right = temp->left;
            temp->left->par = temp->par;
        }
        else
        {
            temp->par->right = NULL;
        }
```

* temp ile işaret etti.
* en büyüğü bulmak lazım. sürekli sağa gitmeli.
* sağ çocuğu sileceğimizi biliyoruz.
* 7'den küçük en büyük 6. 6'nın sağ çocuğu NULL.

p -> 7'yi işaret ediyor. P, 7'yi işaret ettiğinde break ile çıkar. Silinecek düğümü işaret eder p. sol çocuk NULL'dan farklı mı? -> 4 => farklı.



-> sahipsiz kalmadı. 5'in parent'ı 4 oldu.

temp'in left'i NULL'dan farklı mı? -> farklı. => 7 silindi.

parent artık 16

FABER-CASTELL

12.12.14

2) // Kendisinden büyük en küçük düğümü kendi yerine koy. 8'i silip 9'u.

↳ root'lu silerken 16'nın yerine 25

Bir kere sağa gidip hep sola gideceğiz

```
if(p->right != NULL)
      ↑ büyük dediği için right.
{
  temp = p->right;
  while(temp->left != NULL) temp = temp->left;
  root->elt = temp->elt;

  if ( temp->right != NULL)
  {
     temp->par->left = temp->right;
     temp->right->par = temp->par;
  }
  else
  {
     temp->par->left = NULL;
  }
     delete temp;
     return;
}
   else if (p->right != NULL)
   {
      root = p->right;
      delete p;
      return;
   }
   else
   {
     root = NULL;
     return;
   }
}
else
{
```

sözlükte önce geliyorsa sola; sonra ise sağa.

12.12.14

```
if( !:strcmp(newNode->kelime.ingilizce, parent->kelime.ingilizce) == -1)
parent->left = newNode;
else
  parent->right = newNode;
  n += 1;
}
```

önce gelirse -1 döndürür. Dolayısıyla sola gidiyoruz. Else ise sağa gideriz.

```
void LinkedBinaryTree::deleteNode* p, char* sorgu)
{
   Node* temp;
   Node* parent;

   while (p != NULL) → silinecek elemanı ağacı içerisinde arıyor.

     if( ::strcmp (sorgu, p->kelime.ingilizce) == 0) break;

   else
   {
    parent = p;
   if (::strcmp (sorgu, p->kelime.ingilizce) == -1)

     p = p->left;
   else
     p = p->right;
   }
}
```

p silinecek elemana eşitse 0 döndürür. Yiki string eşitse 0 döndürür.

Ağaçtaki veri sayısı n olsun.

Karşılaştırma sayısı da en dipte ise (en kötü ihtimal) $\log_2 n$ olur.

FABER-CASTELL

12.12.2014

```
# include "LinkedBinaryTree.h"

void main()
{
    LinkedBinaryTree splayTree;

    splayTree.addRoot();
    splayTree.root -> elt = 12;

    splayTree.addBelowRoot(splayTree.root, 2);
    splayTree.addBelowRoot(splayTree.root, 10);
    splayTree.addBelowRoot(splayTree.root, 6);
    splayTree.addBelowRoot(splayTree.root, 4);
    splayTree.addBelowRoot(splayTree.root, 8);
    splayTree.addBelowRoot(splayTree.root, 5);
    splayTree.addBelowRoot(splayTree.root, 9);
    splayTree.addBelowRoot(splayTree.root, 1);
    splayTree.addBelowRoot(splayTree.root, 13);
    splayTree.addBelowRoot(splayTree.root, 7);   // (12.01.2012 Final)
```

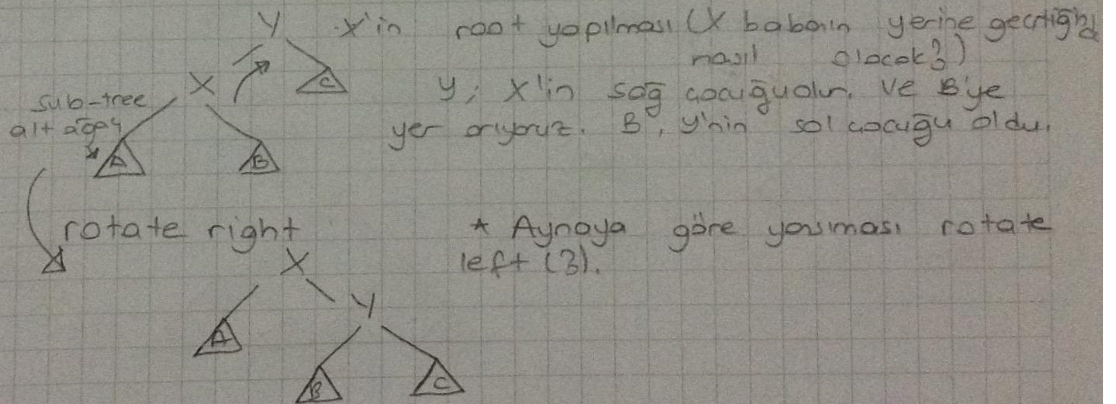
→ tüm çizdiklerimizi bu kodlar oluşturuyor.

Ø'dan Splay Tree → X'i hep yukarı çıkarıyoruz.

Her bir veriyi eklediğimizde sona ekleme yaparız. Ve ardından roota ekleriz. Bu aynı zamanda balance işlemi de görür. Silme için aynısı geçerli değil.

Tree Rotations

Splay trees are kept balanced with rotation.



X'in root yapılması (X babanın yerine geçtiğinde nasıl olacak?)

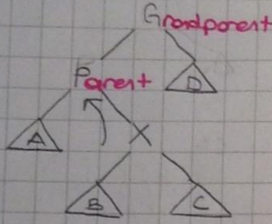
Y; X'in sağ çocuğu olur. Ve B'ye yer arıyoruz. B, Y'nin sol çocuğu oldu.

* Aynaya göre yansıması rotate left (3).

12.12.14

ZIG-ZAG

① X is left child of a right child OR right child of a left child.
↓ X'in kendisi

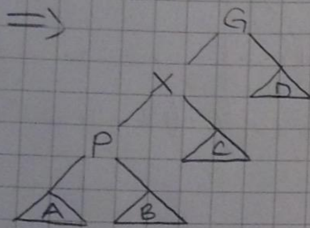


Dededen X'e gelirken zig-zag oluşur.

X, önce babanın yerine geçiyor. Sonra da dedenin yerine geçecek.

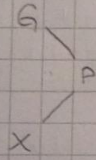
İkinci adımda dedenin yerine geçecek.

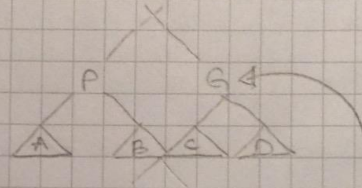
⟹



1. adım

⟹





G; X'ten büyük olduğu için

zig-zag

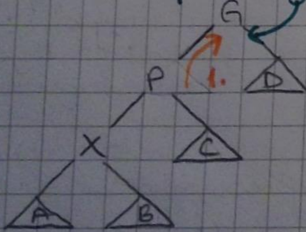
2. adım

Keep your eyes open :)

② ZIG-ZIG

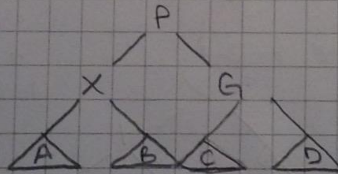
X is left child of a left child OR right child of a right child. Hep aynı doğrultuda. → Tek bir doğrultuda gidiyor.

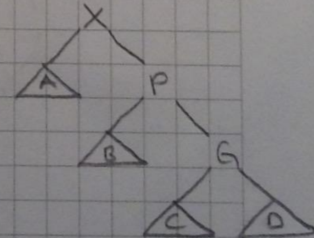
parent yukarı çıkacak



Önce baba dedenin yerine geçecek sonra X babanın yerine geçecek.

⟹





denge = root'un sağındaki düğüm sayısı solundaki düğüm sayısına eşit olma durumu.

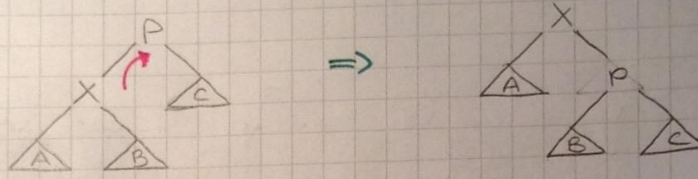
FABER-CASTELL

12.12.14

* 1 ve 2 işlemini X root'un çocuğuna kadar çıkmışsa sürekli gerçekleştiriyoruz.

③ ZIG

* X is child of the root.



∅ dan Splay Tree

* İkili ağaç mantığına göre root'u ekliyoruz. küçükse sola, büyükse sağa.

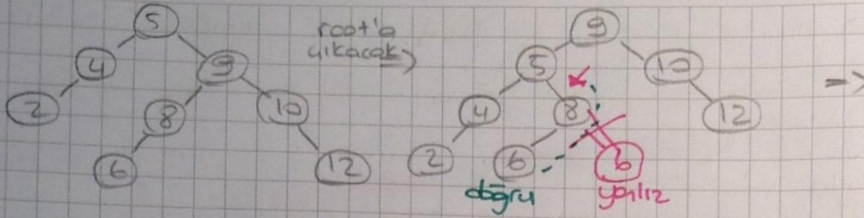
12 → 2 zig => 2 G, 12 P, 10 X => 2, 10, 12 => 10 G zig-zag, 2 P, 12, 6 X =>

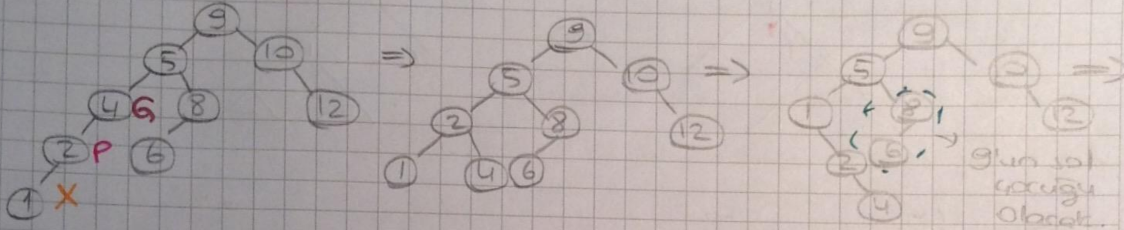
10, 6, 12, 2 => G 6, P 2, 10, X 4, 2 => 6, 4, 10, 2, 12 => 4, 2, 6, 10 G P, 8 X, 12 =>

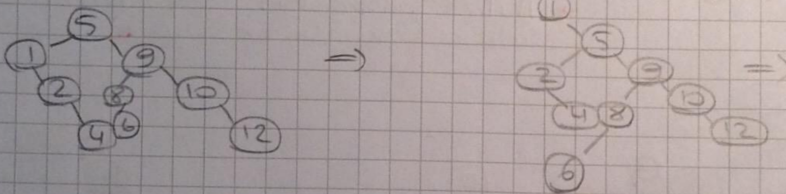
4, 2, 6, 8, 10, 12 => 4, 2, 8, 6, 10, 12 => 8, 4, 10, 2, 6, 12 => 8, 4 G, 10, 2, 6 P, 12, 5 X =>

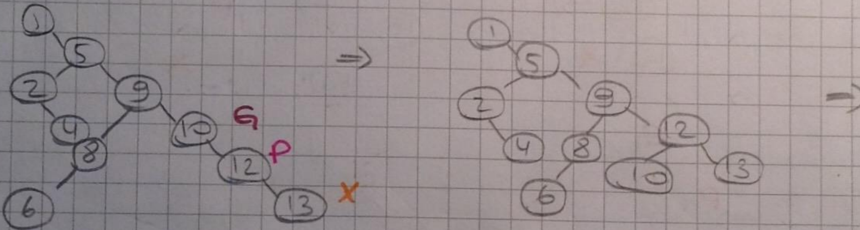
8, 5, 10, 4, 6, 12, 2 => 5, 4, 8 G, 2, 6, 10 P, 9 X, 12 => 5, 4, 8, 2, 6, 9, 10, 12 =>

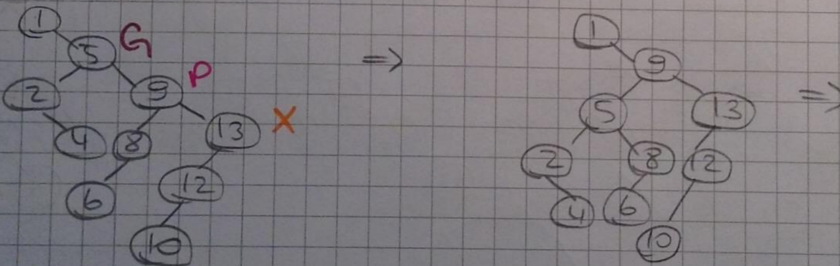
12.12.14

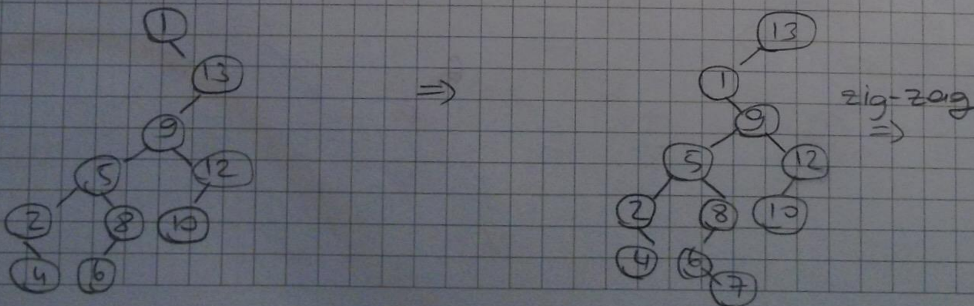




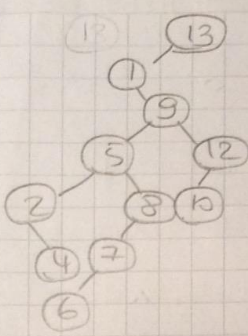




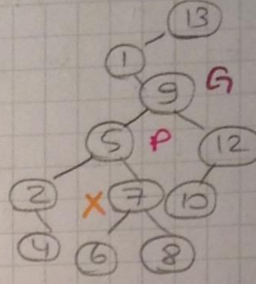




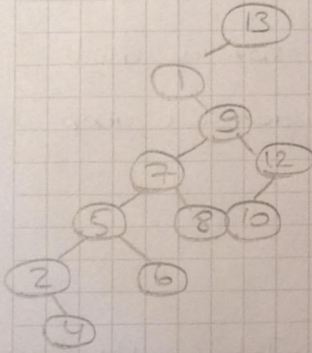
12.12.2014



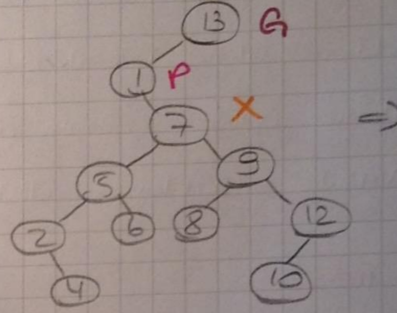
=>



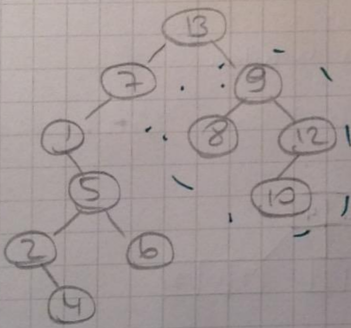
=>



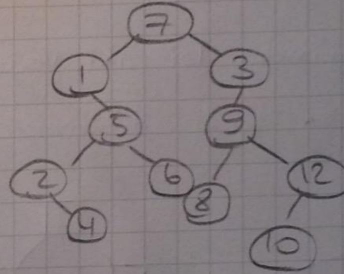
=>



=>



=>



* (7)'nin solunda ve sağında beş düğüm var.

"Hashing.cpp" synony chaining ödevi 19.12.2014

* boş bir adresi bulup yazmanın yanı sıra bağlı listelerde tutuyoruz.
çakışma var ise: relative txt'de bir kayıt var. Aynı zamanda bağlı listeye yazmıyoruz.
hash fonksiyonunun ürettiği adres

temp = ilk hesaplanan adres (while'a girmeden önce) tutuluyor.

synonym chaining'de sorgulama daha hızlı; relative txt'de yoksa devam edilmiyor doğrudan o adres değerine sahip bağlı listenin head pointer'ına atıyoruz (Node pointer'ı). Addback fonksiyonunu kullanırız.

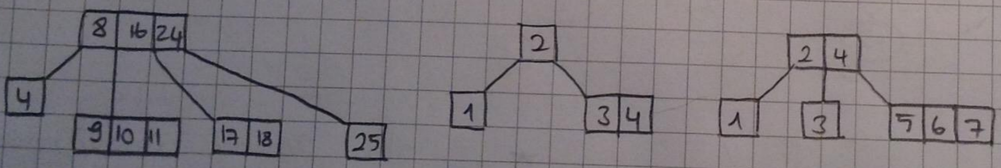
kelime ekleme silme işlemlerinde hem dosyada hem de bağlı listede güncelleme yapılıyor. Synonims diye bir dosyaya yazmazsak uçup gider çünkü RAM'de saklanıyor diskte değil.

2-3-4 Tree

* Her bir düğümün iki, üç veya dört çocuğu vardır. Tek çocuklu düğüme izin yok.

* Her bir düğümde bir, iki veya üç eleman tutulur.

Binary tree'den farkı; düğümden birden fazla eleman tutuluyor olması.



* düğümdeki eleman sayısının bir fazlası kadar çocuğu olmak zorunda.

Düğüm içindeki elemanlar küçükten büyüğe sıralı tutulur.

hem düğümdeki elemanlar hem de çocuklardaki pointerlar değişken.

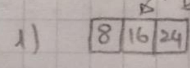
Çift yönlü listenin modifiye edilmiş hâli.

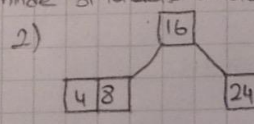
19.12.2014 CUMA

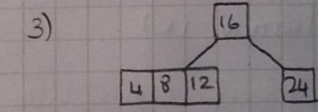
Veriyapıları 3- devam

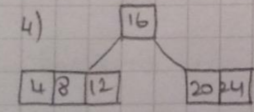
```
#include "LinkedBinaryTree.h"

void main()
{
  LinkedBinaryTree tree_234;

  tree_234.insert(16); → ilk eklenir.
  tree_234.insert(8);
  tree_234.insert(24); 'ün next'inin down'ı 26-28
  tree_234.insert(4);
  tree_234.insert(12); → 16'dan küçük (ikili ağaç mantığına göre)
  tree_234.insert(20);
  tree_234.insert(28); next'inin down'ı 29-30
  tree_234.insert(2);
  tree_234.insert(6);
  tree_234.insert(10);
  tree_234.insert(14);
  tree_234.insert(18);
  tree_234.insert(22);
  ⋮
```

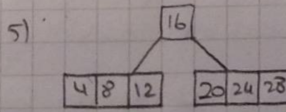
+ Üç elemanlı düğüme rastlarsak kesinlikle parçalanıyor.
↳ bir sonraki ekleme işleminde ortadaki baba olur.

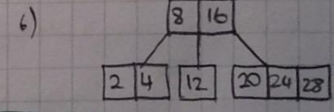
1) 

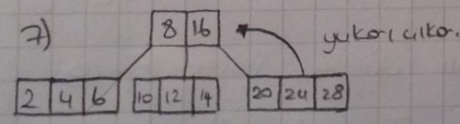
2) 

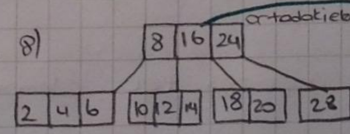
3) 

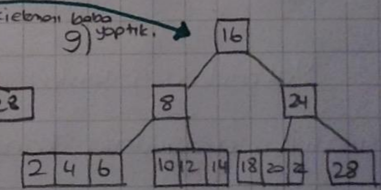
4) 

5) 

6) 

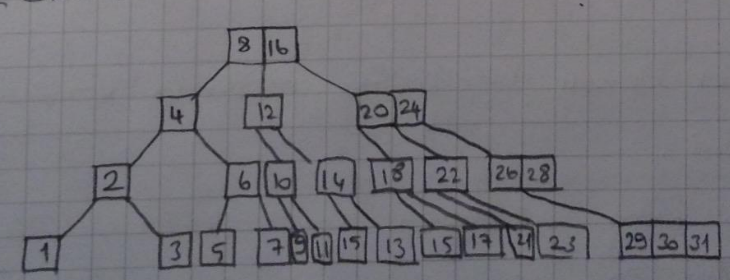
7) 

8) 

9) 

★ Üçlü olan root'u parçalamak daha mantıklı.
Çocuklar baba olabilsin diye.

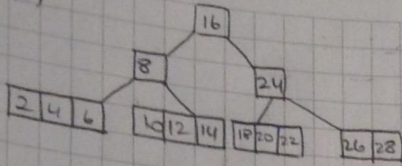
—— o ——



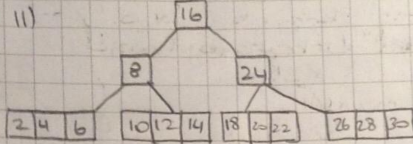
8.0 vs 8.5
very balanced tree...

FABER-CASTELL

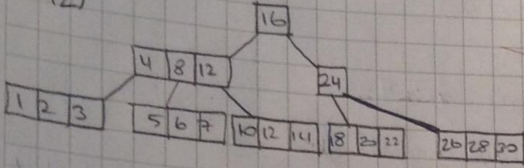
10)

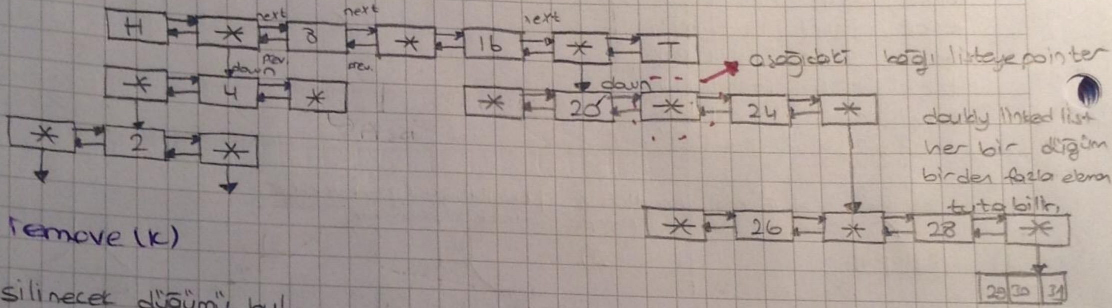


11)



12)





* remove(k)
* silinecek düğümü bul.
* Eğer yapraksa (çocuğu yoksa, externalsa) sil.
* Eğer çocuğu varsa (internal) kendisinden büyük en küçük elemanı onun yerine yaz.
  ↳ tam tersi de olabilir.

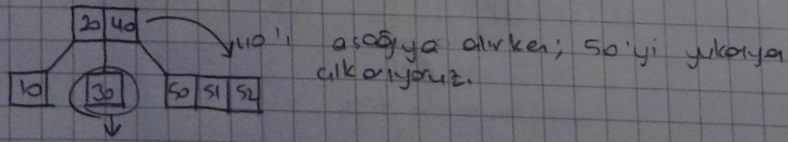
Yukarıdaki işlemleri yaparken tek elemanlı düğüme rastlandığında sırasıyla aşağıdaki kurallardan gerekeni uygula. Tek düğümlü elemanları komşudan alarak bir şekilde birleştirerek hareket etmeliyiz.

Rule-1

Eğer (root dışında) tek elemanlı bir düğüme rastlarsa komşuların birinden ödünç almaya çalış.

Tries to steal key from an adjacent.

remove(30)



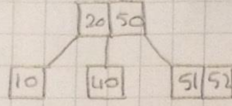
40'ı aşağıya alırken; 50'yi yukarıya çıkarıyoruz.

yaprak fakat Rule-1'e uymuyor. 30'u silersek bulunduğu pointer boş kalır. Mutlaka komşudan ödünç almalıyız.

FABER-CASTELL

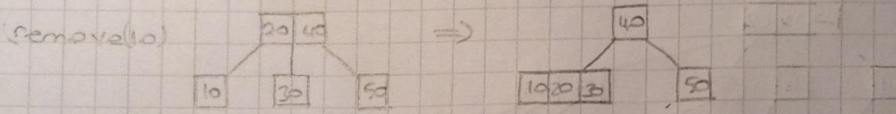
19.12.2014
CUMA

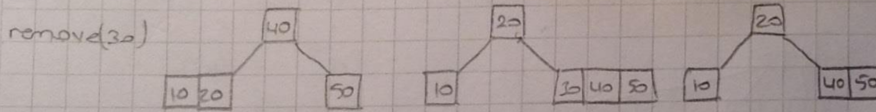
★ 2-3-4 ağacı da Splay Tree gibi dengeli olarak büyür. Dengeleme olduğu için de sorgulama hızı o kadar iyi. Seviye miktarından mütevellit.



Rule-2

Eğer bir elemanlı düğümün komşularının hiç birinde birden fazla yoksa (hepsi tek elemanlıysa) babadan bir eleman alıp (babası, komşu, kendisi) olmak üzere üç elemanlı bir düğüm oluştur.

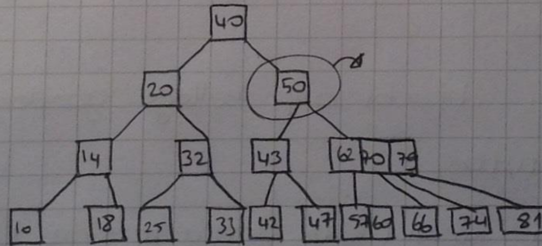




Rule-3

Eğer baba kök(root) ve komşular birer elemana sahipse üçünü tek düğümde birleştir. Bu üç elemanlı düğüm yeni root olur. Bu durumda ağacın seviyesi bir azalır.

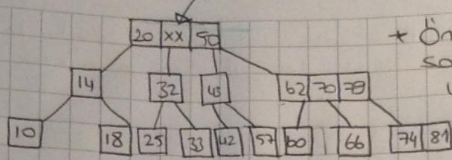
remove (40)



40, en alt seviyede (yaprak olmadığı için) kendinden büyük en küçük olan 42'yi arıyoruz.

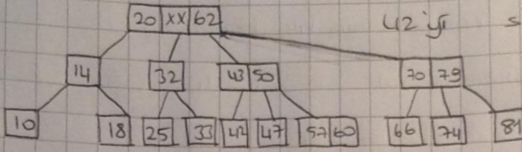
19.12.2014

40 idi silindi.

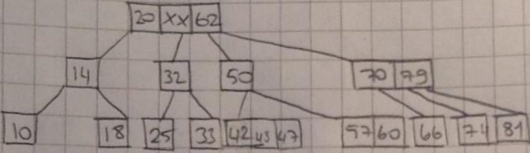


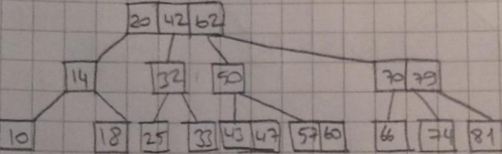
* Önce bir kez sağa sonra hep sola gidecek. 43'e geldi; 43 uymuyor fakat ödünç alabiliyoruz.

* Ekleme ve silme işlemleri 2-3-4 ağacında dengeleme işlevi de görüyor.



42'yi silebilmemiz için 43-42 47'yi birleştirmeliyiz.





O(n) Notasyonu - Order N Notasyonu

$O(n^2)$ Algoritma karışıklığı nasıl ifade edilir fonksiyonda?

Asymptotic Analysis

Bellekte tutulan diziyi dosyadan alıp belleğe gönderiyoruz.

10,000 ms saniye tutar.

* Her bir verinin sıralama süresi (uygun yere yerleştirme) 10 ms olsun.

$T(n) = 10{,}000 + 10 * n$ → n bağlı listeye eklenecek elemanların sayısı olabilir.

→ n değeri arttığında algoritmanın koşma süresini büyük ölçüde etkileyen kısım!!!

Sadeleştirirken big Oh notasyonu devreye girecek. baskın kısmı etkili olacak (n→∞ giderken vb).

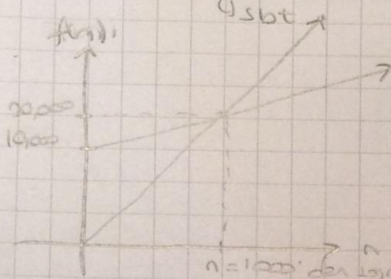
FABER-CASTELL

19.12.2014
CUMA

Big-Oh Notation (upper bound) → üst sınır
↳ ikinci fonksiyonu belirtmek önemli, order fonksiyonu burada devreye girecektir.

$T(n) \in O(f(n))$

$T(n) \leq c * f(n)$
sbt



$Tn = 10{,}000 + 10n$
$f(n) = 20 * n$
c, 20 alınırsa istediğimiz sayıyı verebiliriz.

n=1000'den sonra $f(n) = 20n$ kesinlikle T(n)'den büyük.

$f(n) = n$ → $O(f(n)) = O(n)$

peş peşe for → n
iç içe for $\xrightarrow{\text{ikiise}}$ n^2
$\xrightarrow{\text{üç}}$ n^3
⋮

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2+n}{2} \qquad O(n^2)$$

$$T(n) = \frac{1}{2}n^2 + \frac{1}{2}n$$

$$T(n) = n^3 + n^2 + n + 1000$$

— O —