KARADENİZ TEKNİK ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

BIL 2001 VERİ YAPILARI
DERS NOTLARI

MELTEM DOĞAN

2017-2018 Güz Dönemi

Veri Yapıları
Melike DOĞAN

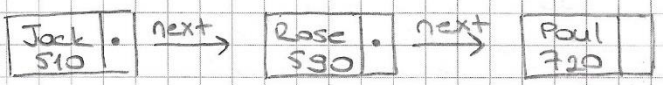
Ders kitabı : Data structures & Algorithms in C++

web sayfası : ceng2.ktu.edu.tr/ncakir/veri_yapileri.html

~ Veri yapıları verileri birbirine bağlayarak kurulur. Verileri birbirine pointerlarla bağlarız.

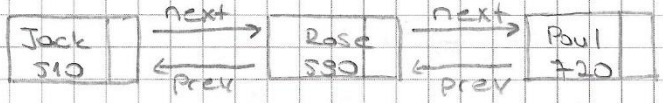
Singly linked list

```
struct SinglyNode
{
    string elem;
    int    score;
    SinglyNode* next;
};
```

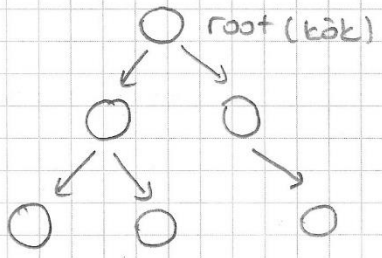


Double linked list

```
struct DoublyNode
{
    string elem;
    int    score;
    DoublyNode* next;
    DoublyNode* prev;
};
```



Binary Tree



1. struct ta düğümleri
2. struct ta fonksiyonları tutacağız. (Eleman ekleme, silme, merge etme, sıralama,.... gibi işlemler yapılacak.)

Örn (scores) /

```
struct GameEntry {

    String name;
    int    score; };

    void main ()

{   GameEntry gEntry;
    gEntry.name = "Omer";
    gEntry.score = "1000";

    cout << gEntry.name << "\t" << gEntry.score << endl;
```

printf gibi

Çıktısı : Omer 1000

```
    GameEntry * pEntry = &gEntry;

    pEntry -> name = "Oguzhan";

    pEntry -> score = 1500;

    cout <<  - - - - - - .
```

Çıktısı : Omer 1000
Oguzhan 1500

Insertion Sort (sıralama)

Dıs'daki for döngüsü 1 den başlıcak yani 1 eksiği kadar dönücek

```
#include <iostream>
using namespace std;
Void insertionSort(int A[], int n)
{
For (int i=1 ; i<n ; i++)
{
  int cur = A[i] ;
  int j   = i-1 ;
while ((j>=0) && (A[j]>cur))
{ A[j+1] = A[j] ;
  j-- ;
}
  A[j+1]=cur ;
}
}
Void main()
int A[8]= { 3,7,5,2,8,4,6,1 } ;
insertion sort (A,8) ;
cout << "Sorted array: " ;
for(int i=0; i<8 ; i++) cout<< A[i] << ' ' ;
:: getchar() ;
```

i=	0	1	2	3	4	5	6	7
	3	7	5	2	8	4	6	1

3 elemanlı bir dizide 5 elemanının yerini bulup kaydıcaz.

i=0 için 3

i=1 için cur=7 j=0 3>7 olmadığı için while girmedi ilk iki eleman sıralı

i=2 için

cur = A[i] = 5
j = 1
7 > 5 olduğu için

i=	0	1	2	3	4	5	6	7
	3	7	7	2	8	4	6	1

cur = 5
j = 0 (i-1)
3 > 5

i=	0	1	2	3	4	5	6	7
	3	5	7	2	8	4	6	1

i=3 için

cur = 2
j = 2
7 > 2

i=	0	1	2	3	4	5	6	7
	3	5	7	7	8	4	6	1

j = 1
5 > 2

i=	0	1	2	3	4	5	6	7
	3	5	5	7	8	4	6	1

j = 0
3 > 2

i=	0	1	2	3	4	5	6	7
	3	3	5	7	8	4	6	1

j = -1
cur = 2.

i=	0	1	2	3	4	5	6	7
	2	3	5	7	8	(4)	6	1

i=4 için

2 3 4 5 7 8 (6) 1

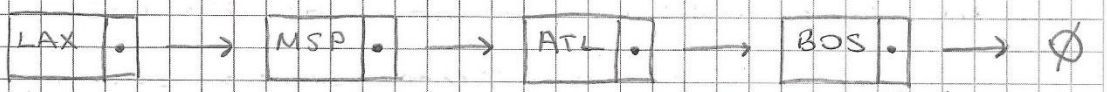
insert

2 3 4 5 6 7 8 1

shift

1 2 3 4 5 6 7 8

Singly Linked List (Tek yönlü Bağlı Liste)



head
↓
(Listenin ilk elemanını işaret eden pointer)

Tail
↓
(Listenin sonunu işaret eder.)

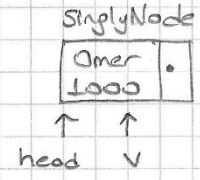
~ Son pointer hiçbir seyi işaret etmediği için null'a setleriz. Ordo pointer var ama boş. Son elemana geldiğini onlar döngüden çıkar.

⇒ addfront : sürekli listenin başına ekleyerek liste oluşturma

```
void SinglyLinkedList :: addfront (cont string & e, const int & i)
    SinglyNode* v = new SinglyNode; // Boş struct açıyor ve
                                    v pointerına işaret
                                    ediyor.
    v -> elem = e;
    v -> score = i;
    v -> next = head;
     head = v;
}
```

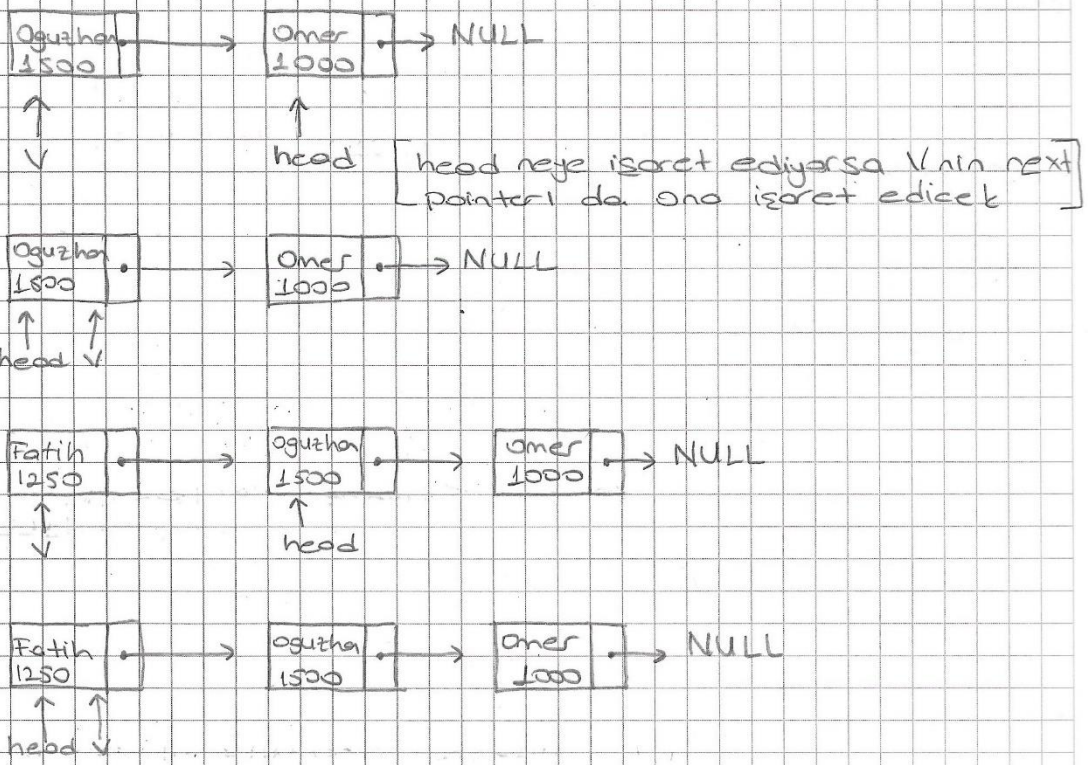
• Başlangıçta head = NULL

addFront ("Omer" "1000")
("Oguzhan" "1500")
("Fatih" "1250")



İki pointer birbirine eşitleniyorsa eşitliğin sağ tarafındaki pointer neye işaret ediyorsa soldaki de ona işaret eder.

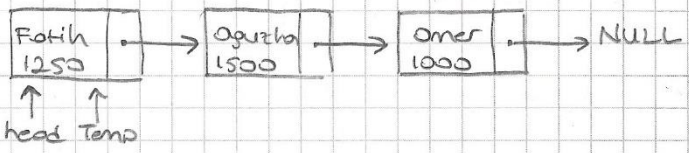
Oğuzhan 1500 ile çağırdığımızda;



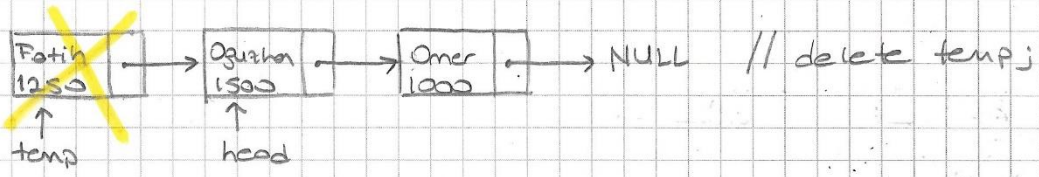
~ Print fonksiyonu liste elemanını baştan sona print eder.

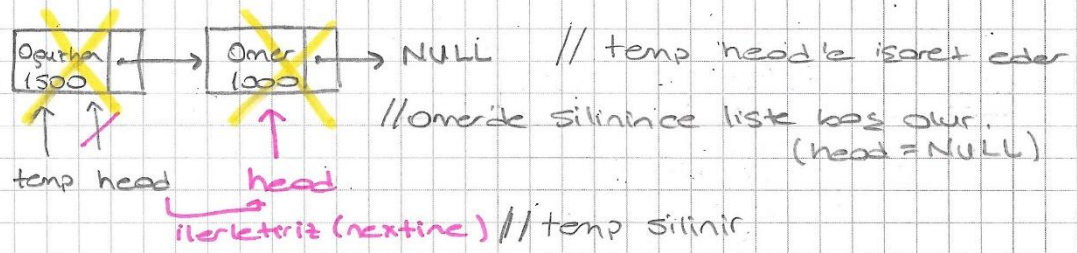
Remove Front: listenin ilk elemanını siler.

```
Void SinglyLinkedList::removeFront()
{
    SinglyNode *temp = head;
    head = head->next
    delete temp;   // head'i ilerlettiğimizde temp silinir.
}
```



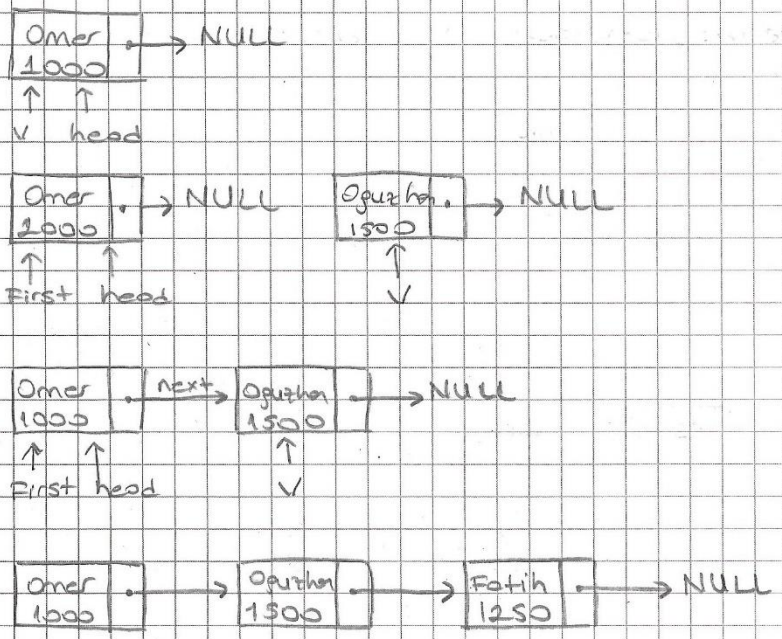
// head = head->next işaret ettiği için head'i ilerletiriz.





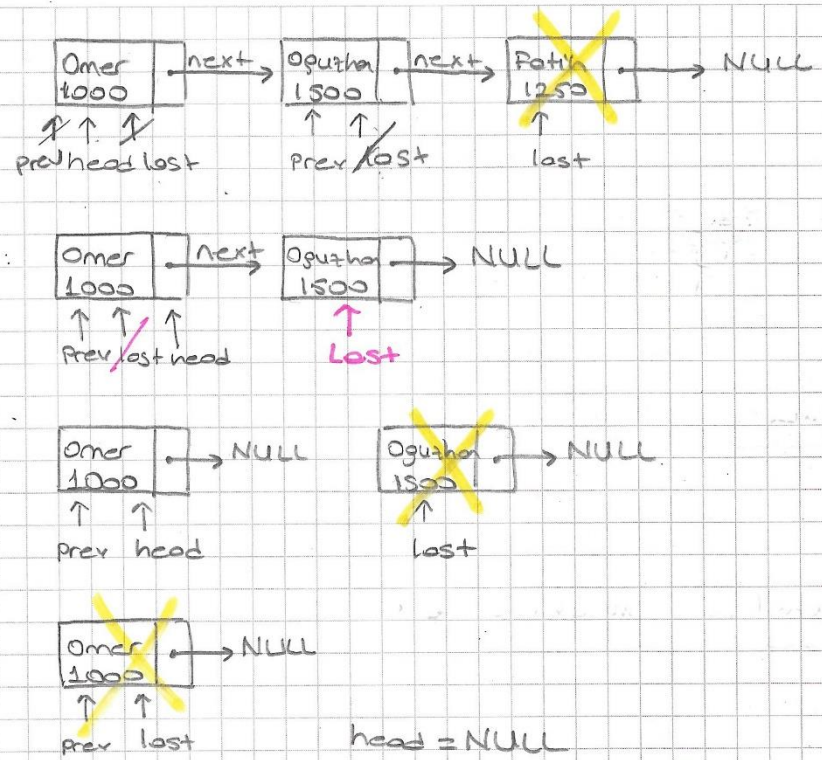
AddBack : Listenin sonuna ekleme

```
Void SinglyLinkedList :: addBack(const string &e, const int &i)
{
    SinglyNode * v = new SinglyNode;

    v -> elem = e;
    v -> score = i;
    v -> next = NULL;

    if (head == NULL) head = v;

    else
    {
        SinglyNode * first = head;

        while (first -> next != NULL) first = first -> next;

        first -> next = v;

    }

}
```



Remove Back : Listenin son elemanını siler

```
Void SinglyLinkedList :: removeBack()
{
    if(head == NULL)
    {
        cout << "list is empty!" << endl;
        return;
    }
    SinglyNode* last = head;
    SinglyNode* prev = head;
    While (last->next != NULL)
    {
        prev = last;
        last = last->next;
    }
    prev->next = NULL;
    if(last == head) head = NULL;
    delete last;
}
```

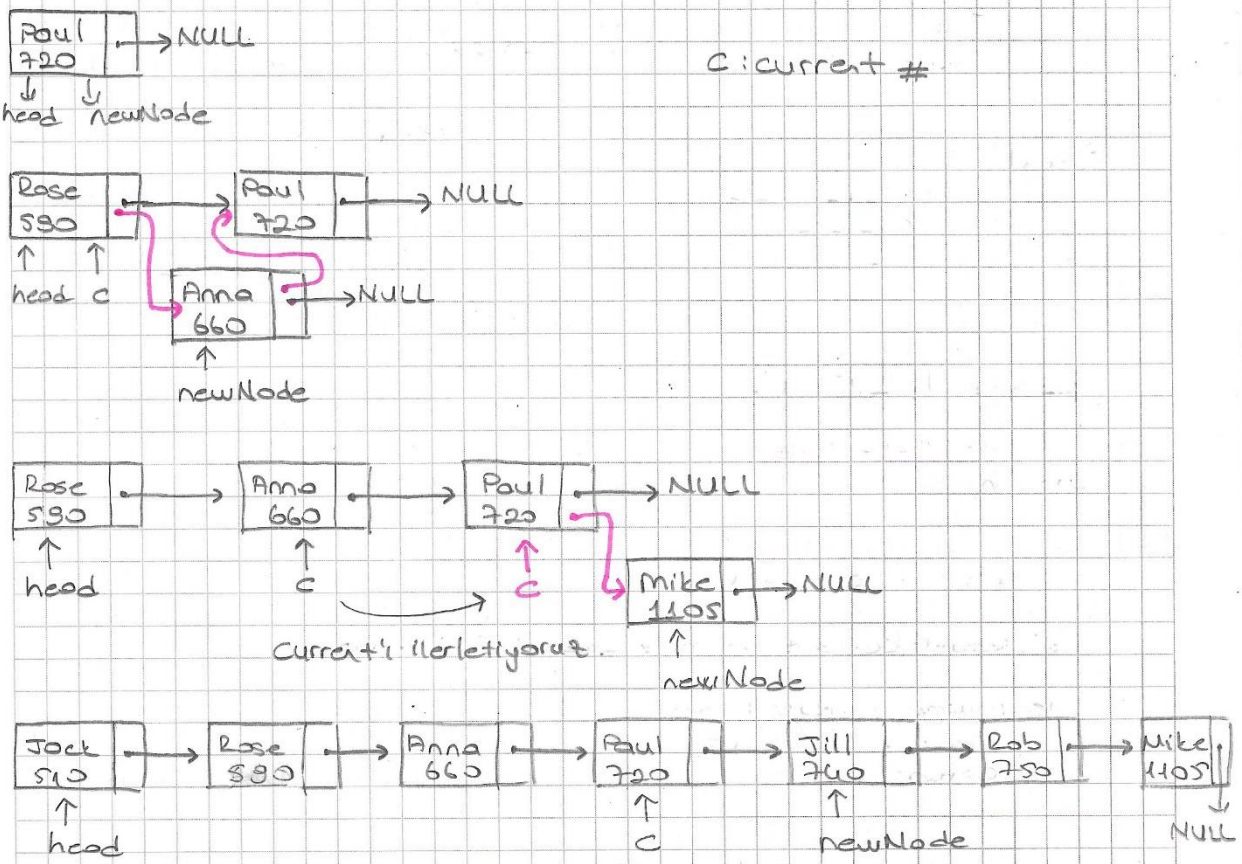


Insert Order : Parametre değerine göre sıralama. (Score değerine göre). Küçükten büyüğe sıralıyor.

Singly Linked list;

```
list. insertOrdered ("Paul", 720);
list. insertOrdered ("Rose", 580);
                    ("Anna", 660);
                    ("Mike", 1105);
                    ("Rob", 750);
                    ("Jack", 510);
                    ("Jill", 740);
```

```
Void SinglyLinkedList::insertOrdered(const string &e, const int &i)
{
  SinglyNode *newNode = new SinglyNode;

  newNode->elem = e;
  newNode->score = i;
  newNode->next = NULL;

  //Liste boş mu?
  if (head == NULL)
  {
    head = newNode;
    return;
  }

  //newNode listenin başına mı eklenecek?
  if (newNode->score < head->score)
  {
    newNode->next = head;
      head = newNode;
      return;
  }

  SinglyNode *current = head;
  while (current->next != NULL
  {
   if (newNode->score >= current->next->score)
    current = current->next;
   else
      break;
   }
    //newNode'u current'tan sonra ekle
      newNode->next = current->next;
       current->next = newNode;
   }
```



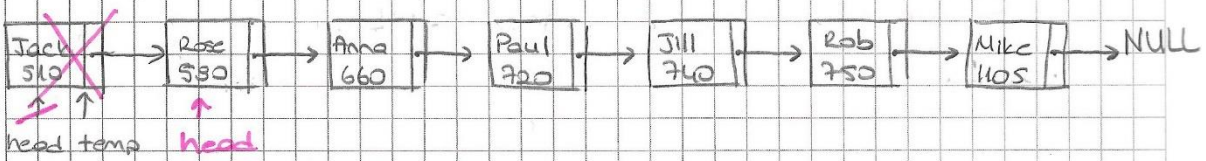
RemoveOrder : Sıralı listede parametre olarak gelen değeri arayıp bulup siliyor.

```
Void SinglyLinkedList::removeOrdered(const string &e, const int &i)
{
  // liste boş mu?
  if (head == NULL)
  {
    cout << "list is empty!" << endl;
    return;
  }
```

(Baştan silme)

```
// Listenin ilk elemanı mı silinicek ?
if ((e.compare (head->elem) == 0) && (head->score == i))
{
SinglyNode * temp = head;
             head = head->next;
             delete temp;
             return;
}
```

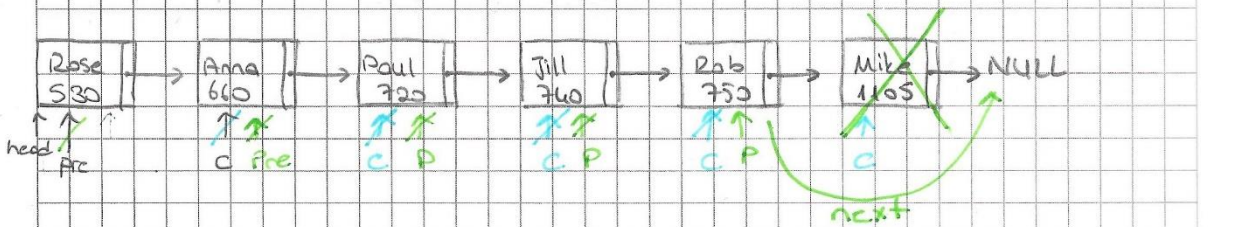
// Baştan silme removeFront ile aynı



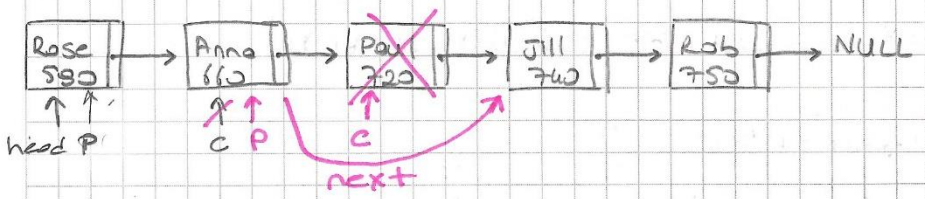
(ortadan ve sondan silme)

```
SinglyNode * previous = head;
SinglyNode* current  = head->next;
While (current != NULL)
{
? ⇒ if ((e.compare (current->elem) == 0 && (current->score == i))
    {
      previous->next = current->next;
      delete current;
      return;
    }
    previous = current;
    current  = current->next;
}
if (current == NULL) cout << "\n" << e << "is not found" << endl;
}
```

// Rob'u siliceğiz !



// Paulu silicez. (Ortadan silme)



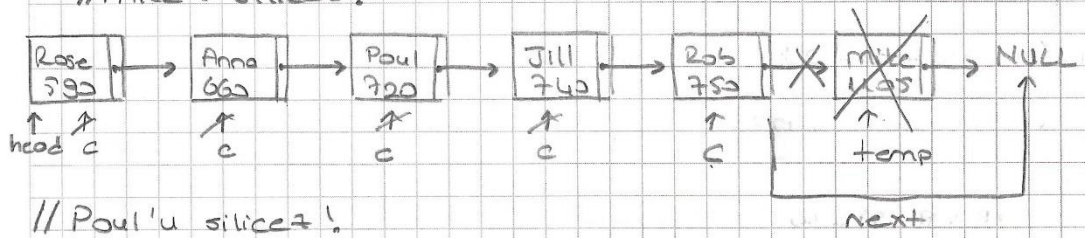
~ Silmek için current yeterli previous'a gerek yok.

```
{ SinglyNode * temp = current->next;
  current->next = current->next->next;
  delete temp;
  return;
}
current = current->next;
}
if (current->next = NULL) cout << "\n" << c << " is not found" endl;
```

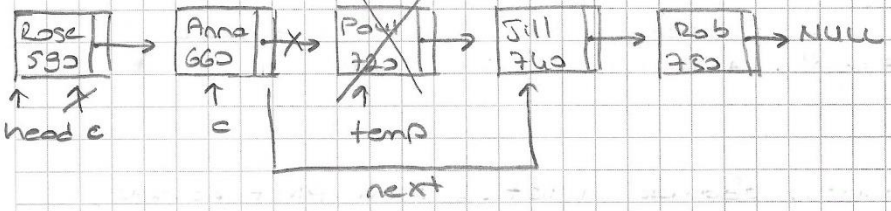
Kodun başında;
SinglyNode *current=head;
var.

▽ Tek başına current'la silmede, current'ın nextini silicez.

o //Mike'ı silicez!



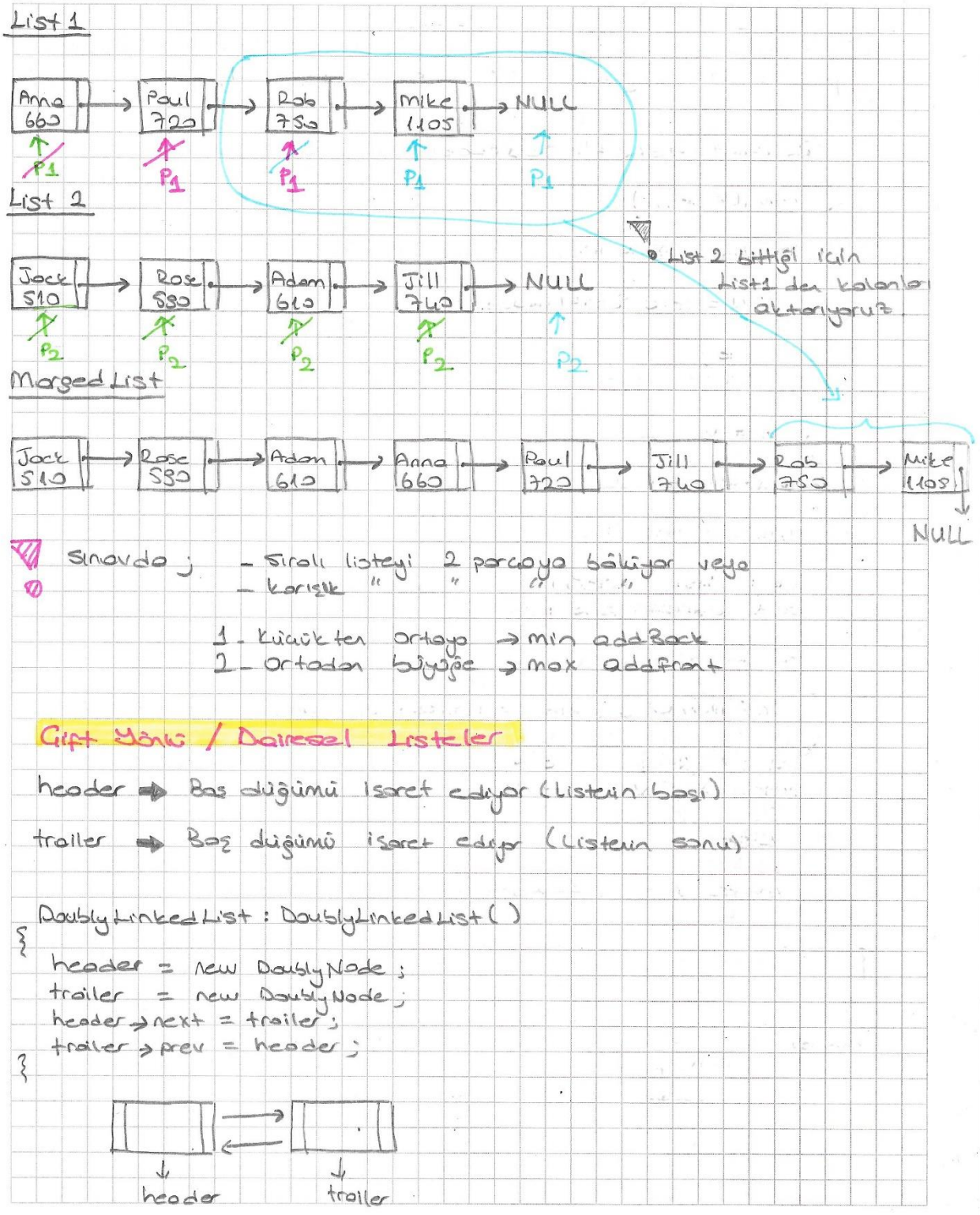
//Paul'u silicez!



Merge List = Sıralı 2 listeyi birleştirerek sıralı 3. Liste yapıyor.

```
SinglyLinkedList * SinglyLinkedList::mergeLists(SinglyLinkedList* list2)
{
  SinglyLinkedList* mergedList = new SinglyLinkedList();

    SinglyNode* plist1 = this->head;
    SinglyNode* plist2 = list2->head;

    while ((plist1 != NULL) || (plist2 != NULL))
    {
      if (plist1 == NULL)
      {
        while (plist2 != NULL)
        {
          mergedList->addBack(plist2->elem, plist2->score);
          plist2 = plist2->next;
        }
        return mergedList;
      }
      if (plist2 == NULL)
      {
      while (plist1 != NULL)
      {
        mergedList->addBack(plist1->elem, plist1->score);
        plist1 = plist1->next;
      }
      return mergedList;
      }
        if (plist1->score <= plist2->score)
      {
        mergedList->addBack(plist1->elem, plist->score);
        plist1 = plist1->next;
      }
        else
      {
        mergedList->addBack(plist2->elem, plist->score);
      }
    }
}
```

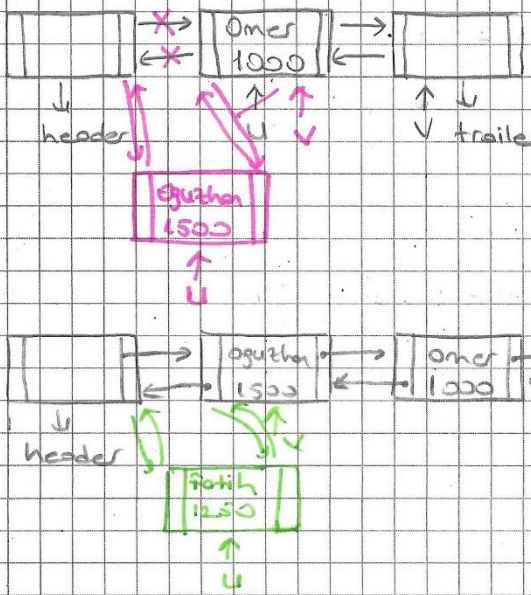
List 1		List 2		MergedList	
Anne	660	Jack	510	Jack	510
Paul	720	Rose	580	Rose	580
Rob	750	Adam	610	Adam	610
Mike	1105	Jill	740	Anne	660
				Paul	720
				Jill	740
				Rob	750
				Mike	1105


List 1
Anna 660
Paul 720
Rob 750
mike 1105
NULL
P1
List 2
Jack 510
Rose 580
Adam 610
Jill 740
NULL
P2
List 2 bittiği için List1 den kalanlar aktarıyoruz.
Merged List
Jack 510
Rose 580
Adam 610
Anna 660
Paul 720
Jill 740
Rob 750
Mike 1105
NULL
Sınavda ; - Sıralı listeyi 2 parçaya bölüyor veya
- karışık " " " "
1 - Küçükten ortaya → min addBack
2 - Ortadan büyüğe → max addFront
Çift Yönlü / Dairesel Listeler
header → Baş düğümü işaret ediyor (Listenin başı)
trailer → Boş düğümü işaret ediyor (Listenin sonu)
DoublyLinkedList : DoublyLinkedList ( )
{
header = new DoublyNode ;
trailer = new DoublyNode ;
header → next = trailer ;
trailer → prev = header ;
}
header
trailer

AddFront, AddBack :

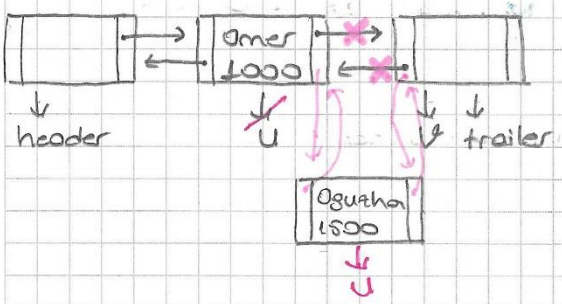
```
Void DoublyLinkedList :: addFront(const String &e, const int &i)
{
  add(header->next, e, i);
}
Void DoublyLinkedList :: addBack(const string &e, const int &i)
{
  add(trailer, e, i);
}
Void DoublyLinkedList :: add(DoublyNode* v, const string &e, const int &i)
{
DoublyNode* u = new DoublyNode;

  u->elem = e;
  u->score = i;

  u->next = v;
  u->prev = v->prev;
  v->prev->next = u;
  v->prev = u;
```

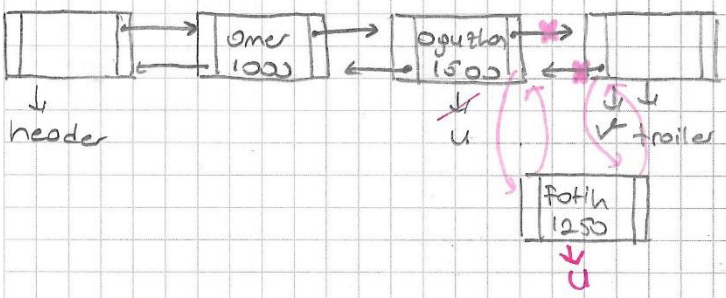
addFront ("Omer", 1000)
("Oguzhan", 1500)
("fatih", 1250)



addBack ("Omer", 1000)
("Oguzhan", 1500)
("Fatih", 1250)

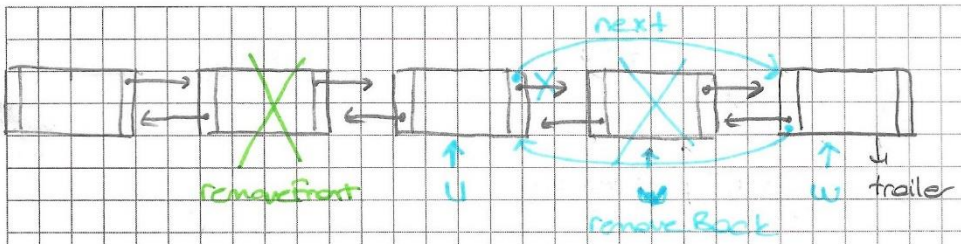
∇ AddBack'te v hep trailer





Remove : Parametre olarak verilen değerin işaret ettiğini siliyor.

```
Void DoublyLinkedList :: removeFront ( )
{
  remove (header → next);
}
Void DoublyLinkedList :: removeBack()
{
  remove (trailer → prev);
}
Void DoublyLinkedList::remove (DoublyNode* v)
{
  DoublyNode* u = v → prev;
  DoublyNode* w = v → next;

  u → next = w;
  w → prev = u;
  delete v;
```



MergeList : # Print H2T → bastan sona

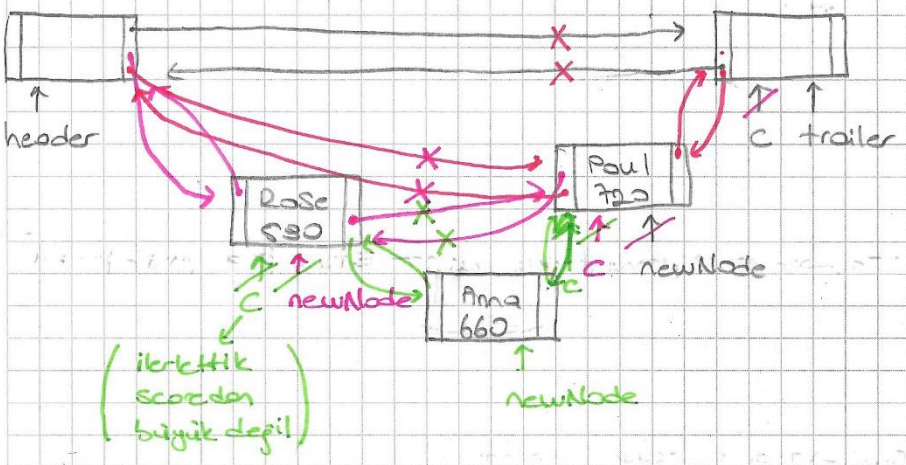
Print T2H → sondan basa printler.

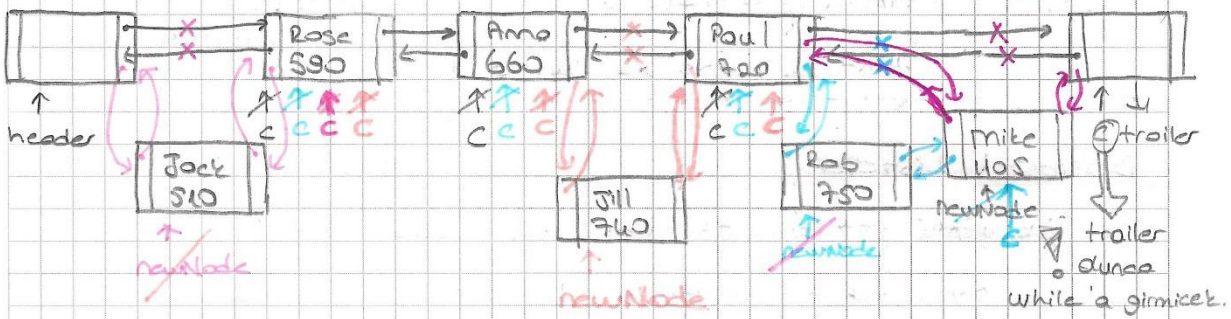
insertOrder :

```
Void DoublyLinkedList :: InsertOrdered (const string &e, const int &i)
{
  DoublyNode* newNode = new DoublyNode;
  newNode -> elem = e;
  newNode -> score = i;

  DoublyNode* current = header -> next;

    while (current != trailer)
    {
      if (newNode -> score >= current -> score)
          current = current -> next;
      else
          break;

  //newNode'u current'tan önce ekle (add() ile aynı)

      newNode -> next = current;
      newNode -> prev = current -> prev;
      current -> prev -> next = newNode;
      current -> prev = newNode;

  }
```

▽ current'ın score'u newNode'un score'undan büyük olana kadar current'ı (o'yu) ilerlet.

List.insertOrdered ("Paul" 720)
("Rose" 590)
("Amo" 660)
("mike" 1105)
("Rob" 750)
("Jack" 510)
("Jill" 740)

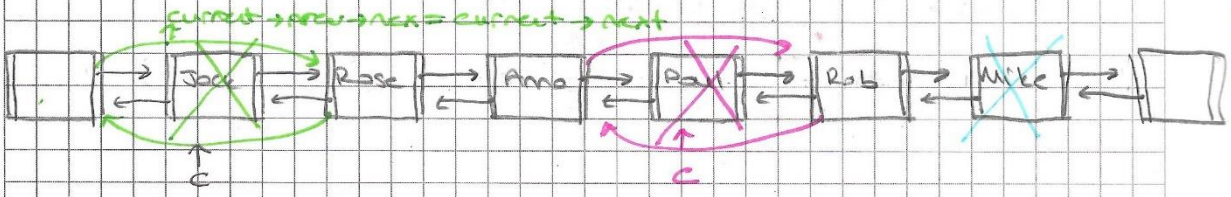




RemoveOrder : Baştan, ortan, sondan silme yapıcaz.

```
Void DoublyLinkedList :: removeOrdered(const string &e, const int &i)
{
// Liste boş mu?
    if (header->next == trailer)
    {
        cout << "list is empty!" << endl;
        return;
    }
```

```
DoublyNode* current = header->next;
While (current != trailer)
{
    if ((e.compare(current->elem) == 0) && (current->score == i))
    {
        current->prev->next = current->next;
        current->next->prev = current->prev;
        delete current;
        return;
    }
    current = current->next;
}
cout << "\n" << e << " is not found" << endl;
```



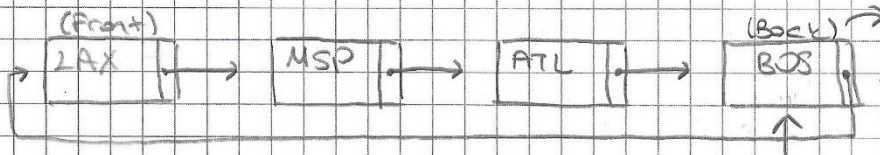
MergeList : null → trailer oluyor (SinglyList deki gibi)

First başlangıçta headera işaret ediyor.

Print T2H → Sondan başa

Dairesel Listeler

- Tek yönlü listelerin özel hali
- Tek farkı listenin son elemanı, ilk elemanını işaret ediyor.



cursor (Pointer)
(dairesel listenin son elemanına işaret ediyor.)

Cursor->next = ilk eleman

Front → cursorun nexti
Back →
Advance → Cursor'u ilerletiyor.
add → cursor'un peşine ekliyor (nextini)
remove → " " deki elemanı siliyor (nextini)

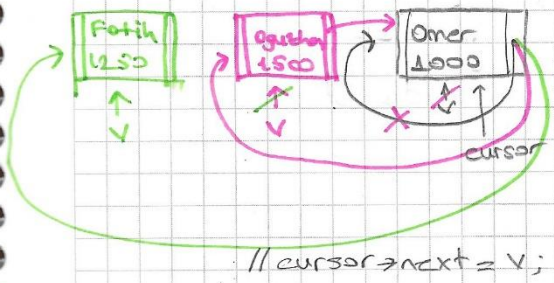
// add → addFront
// remove → removeFront yapıyor.

Add List.add ("Omer", 1000)
("Oguzhan", 1500)
("Fatih", 1250)

cursor = NULL //Başlangıçta

```
Void CircularlyLinkedList::add(const string &e, const int &i)
{
  CircularlyNode* v = new CircularlyNode;
  v->elem = e;
  v->score = i;

  if (cursor == NULL)
  {
     v->next = v;
     cursor = v;
  }
  else
     v->next = cursor->next;
     cursor->next = v;
  }
}
```

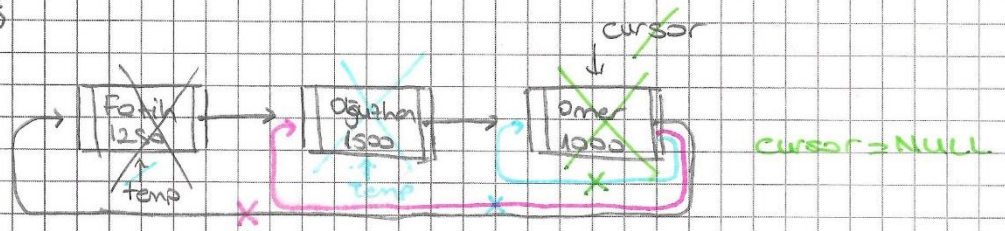
⚠ Add'de cursor'un yeri değişmiyor.



//cursor->next = v;

Remove : // cursor = NULL

```
Void CircularlyLinkedList :: remove ()
{
  if (empty ())
  {
    cout << "List is empty!" << endl;
    return;
  }
  CircularlyNode* temp = cursor->next;
  if (temp == cursor)
      cursor = NULL;
  else
      cursor->next = temp->next;
  delete temp;
}
```



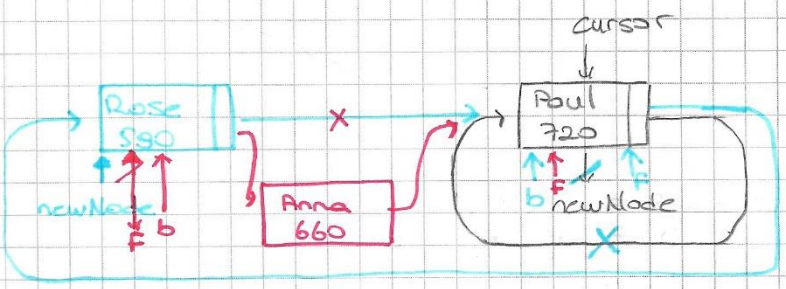
Print = First cursordan farklı olduğu sürecte print yapıyor.

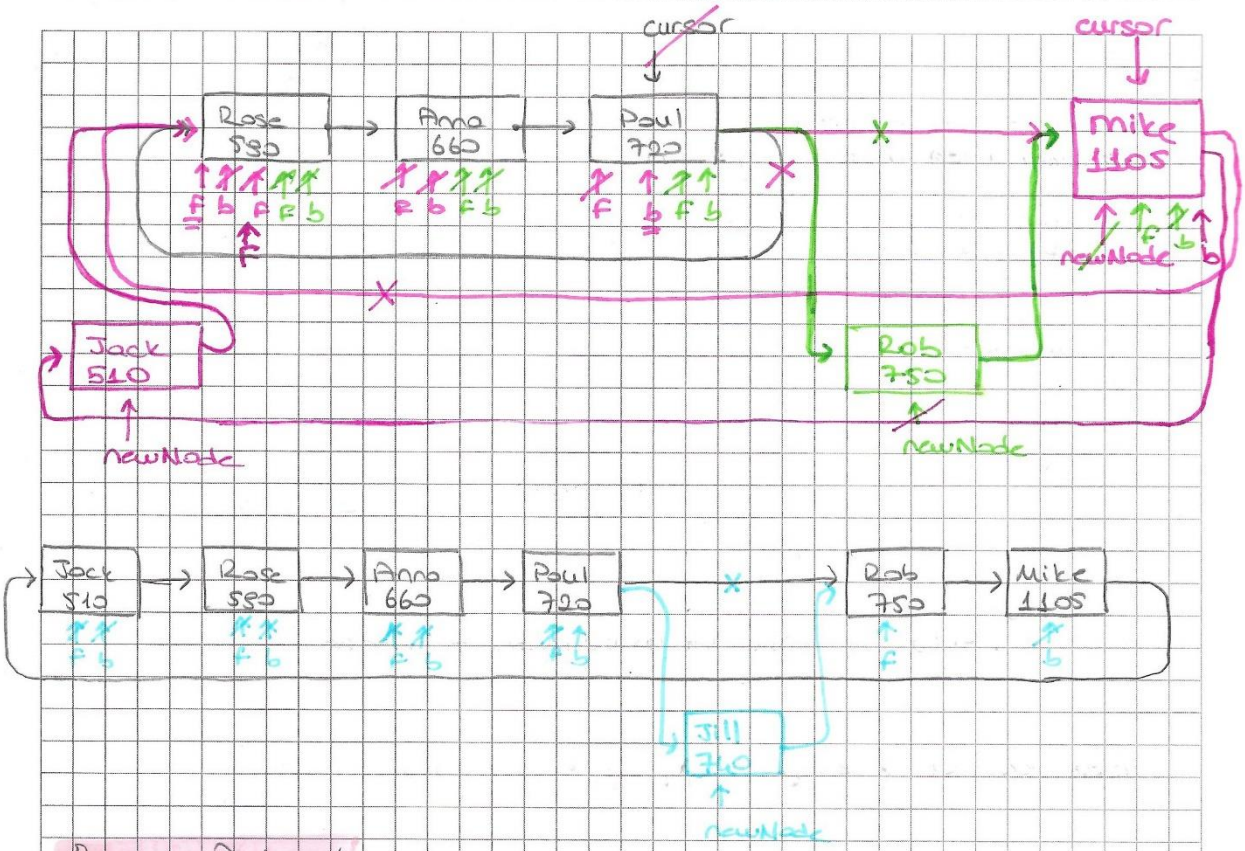
insertOrdered :

```
list.insertOrdered ("Paul", 720)
                   ("Rose", 590)
                   ("Anna", 660)
                   ("Mike", 1101)
                   ("Rob", 750)
                   ("Jack", 510)
                   ("Jill", 740)
```

İlk elemanı eklerken; cursor = NULL

```
void CircularlyLinkedList::insertOrdered(const string& e, const int& i)
{
  CircularlyNode* newNode = new CircularlyNode;
  newNode -> elem = e;
  newNode -> score = i;

  if (cursor == NULL)
  {
    newNode -> next = newNode;
    cursor = newNode
    return;
  }
  CircularlyNode* front = cursor -> next;
  CircularlyNode* back = cursor;

  while (newNode -> score > front -> score)
  {
    back = front;
    front = front -> next;

    if (back == cursor)
    {
    // Liste sonuna ekle

      cursor -> next = newNode;
      newNode -> next = front;
      cursor = cursor -> next;
      return;
    }
  }

  // newNode'u back ve front arasına ekle

    back -> next = newNode;
    newNode -> next = front;
}
```

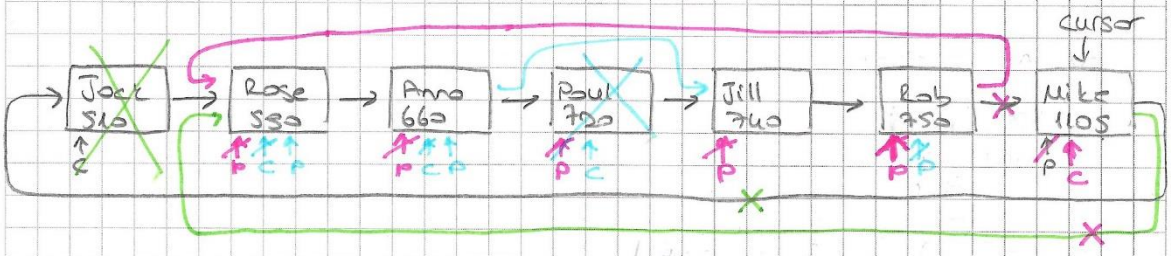




Remove Ordered

```
Void CircularlyLinkedList :: removeOrdered (const string &c, const &i)
{
   If (empty ())
{
   Cout << "list is empty!" << endl;
   return;
};
   // listede kalan tek eleman mı siliniyor?
   if (cursor->next = cursor)
{
   delete cursor;
   cursor = NULL;
   return;
}
   CircularlyNode* previous = cursor;
   CircularlyNode* current = cursor->next;
```

```
While (current != cursor)
{
if ((e.compare(current->elem)==0) && (current->score==i))
// silmek istediğim ia e değeri mi?
{
previous->next = current->next;
delete current;
return;
}

previous = current;
current = current->next;
}

//Listein sonundaki eleman mı silinicek?
if ((e.compare(current->elem)==0 && (current->score==i))
{
previous->next = current->next;
delete current;
cursor = previous; // yeni liste sonu elemanı artık öncekı
}
else

cout << "\n" << e << "is not found" << endl;
```



Recursion

Fonksiyonun kendisini, farklı parametre ile çağırmasına deir

1. Linear Recursion ⇒ fonksiyon kendisini 1 noktadan recursive çağırma
2. Binary Recursion ⇒ " " 2 " " "
3. Multiple Recursion ⇒ " " 2 den fazla noktadan recursive çağırma

2 ye örnek Fibonacci.

Recursive fonksiyon nereden çağrıldıysa oraya return yapacak.

```
int F(int n)
{
    if (n==0) return 1;
    else return n*F(n-1);
}
```

Linear Recursion

```
int linearsum (int A[], int n)
{
    if (n==1)
        return A[0];
    else
        return A[n-1] + linearsum(A, n-1);
}
int main()
{
    int A[] = {1,2,3,4,5,6,7,8}; // 1 den 8 e kadar olan sayıların toplamını hesaplayan program

int reverseArray (int *A, int i, int j)
{
    if (i<j)
    { int temp = A[j];
      A[j] = A[i];
      A[i] = temp;
      reverseArray(A, i+1, j-1); // son satırda recursive çağrı
    }
}
    int main()
    {
        int A[] = {1,... 8}; // Büyükten küçüğe sıralama yapar
```

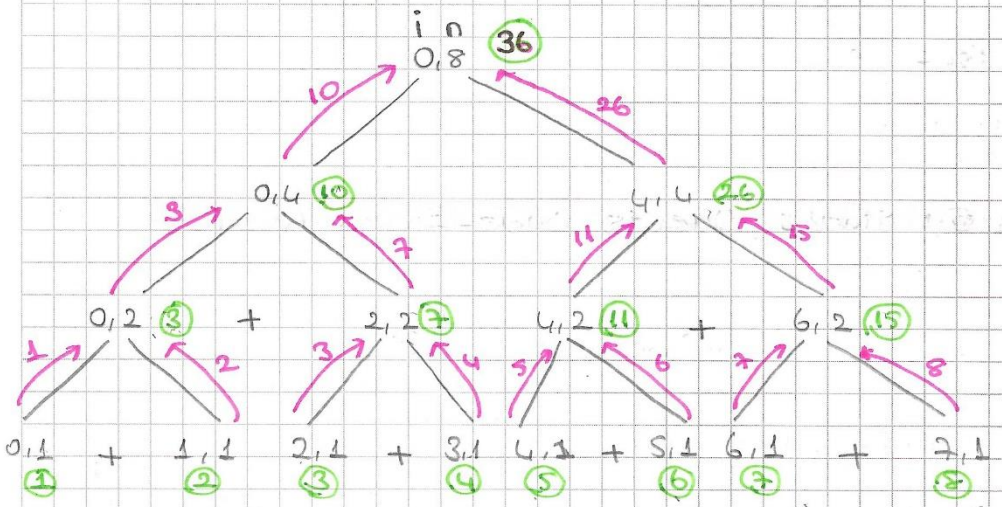
Tail Recursion : Linear rec. özel bir hali. Hem linear olucak hemde recursive çağrıların sonlara doğru yapılması.

Yukarıdaki kod buna örnektir.

Binary Recursion

```
int binarysum ( int A[], int i, int n )
{
   if ( n==0 return A[i];

   else
     return binarysum (A, i, n/2) + binarysum(A, i+n/2, n/2 );
}
```

Burayı değerlendirebilmek için öncesinin return yapılması lazım. Bir sayısal değer gerek




```
int binaryFibonacci (int k)
{
   if (k<=1)
     return k;
   else
     return binaryFibonacci (k-1) + binaryFibonacci (k-2);
}

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

```
int main()
{
```

(0, 1, 10)	F_1
1, 1, 9	F_2
1, 2, 8	F_3
2, 3, 7	F_4
3, 5, 6	F_5
5, 8, 5	F_6
8, 13, 4	F_7
13, 21, 3	F_8
21, 34, 2	F_9
34, 55, 1	F_{10}

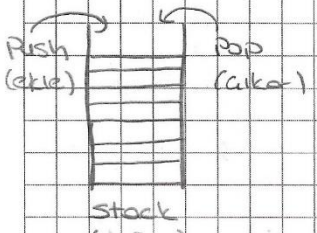
return 6

CHAPTER 5 : Stacks, Queues, Duble-Ended Queues

[SinglyList] → Kuyruk [CircularlyList] → Çift sonlu kuyruk [DoublyList]

STACKS

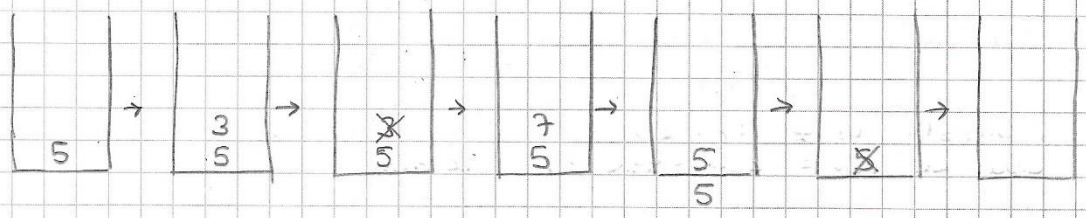
İnternette yada browserda bir önceki sayfaya gitmek için back tuşuna bastığımızda bir önceki sayfanın linkleri yığında tutulur. Bu yığın veri yapısıdır. LIFO [last in first out: son giren ilk çıkar.] prensibine göre çalışır.



- Yığının tepesinde en son ziyaret edilen sayfa tutulur.
- Gündel hayattan; silah şarjörü, dolmuşların bozuk para koydukları apparat LIFO mantığıyla çalışır.
- Yığın veri yapısının 3 temel fonksiyonu vardır:
  1. Top – Yığının tepesindekini return yapar
  2. Push – Yığına iter. Parametrelidir. [Ne ekleyeceğini parametre olarak veriyor]
  3. Pop – Tepedekini siler.

	Operation	output	Stock Constent
1.	push(5)	—	5
2.	push(3)	—	5,3
3.	pop()	—	5
4.	push(7)	—	5,7
5.	pop()	—	5
6.	top	5	5
7.	pop()	—	0
8.	pop()	error	0
9.	top()	error	0
10.	empty()	true	0
11.	push(9)	—	9
12.	push(7)	—	9,7
13.	push(3)	—	9,7,3
14.	push(5)	—	9,7,3,5
15.	size	4	9,7,3,5
16.	pop	—	9,7,3
17.	push(8)	—	9,7,3,8
18.	pop()	—	9,7,3
19.	top()	3	9,7,3

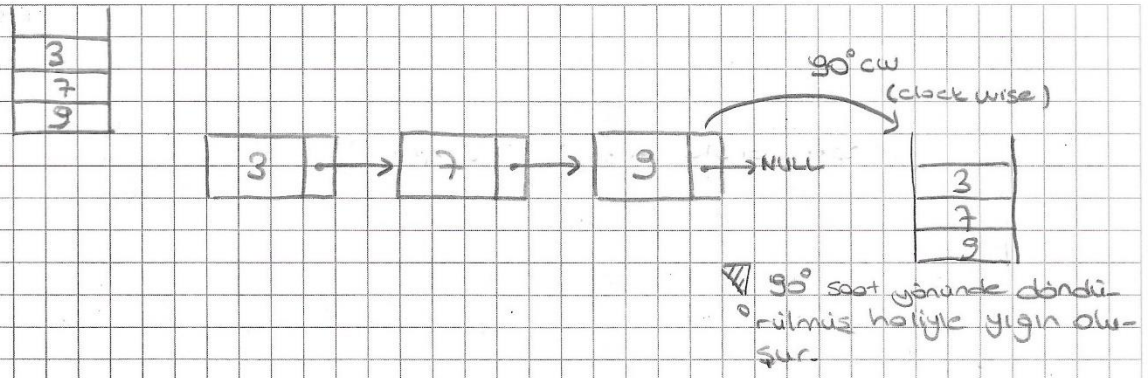
→ 5'i yığına ekle
→ 3'ü " "
→ 3'ü yığından sil



→ Yığın boş olduğu için 8. satırda birşey çıkaramayacağı için error çıkartır.

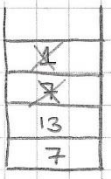
? • front → yığında top
• addFront → Push
• removeFront → Pop

⇒ STL (Standard Template Library) → Veri yapıları için kütüphane




```
int main ( )
{
LinkedStack LStack;

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

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

   LStack.pop( ) ;
   Lstack.push(9);

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

   LStack.pop ( );

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

   :: getchar( );
}

Const int & LinkedStack :: top( ) Const
{
if (empty( ) ) {Cout << "Top of empty stack!" << endl; return 0; }
return S.Front( );
}

Void LinkedStack :push ( const int & e)
{
   ++n ;
   S.addfront (e);
}
```

```
void LinkedStack :: pop ()
{
  if (empty())
  {
    cout << "Pop from empty stack!" << endl;
    return;
  }
  --n;
  S.removeFront();
}
```



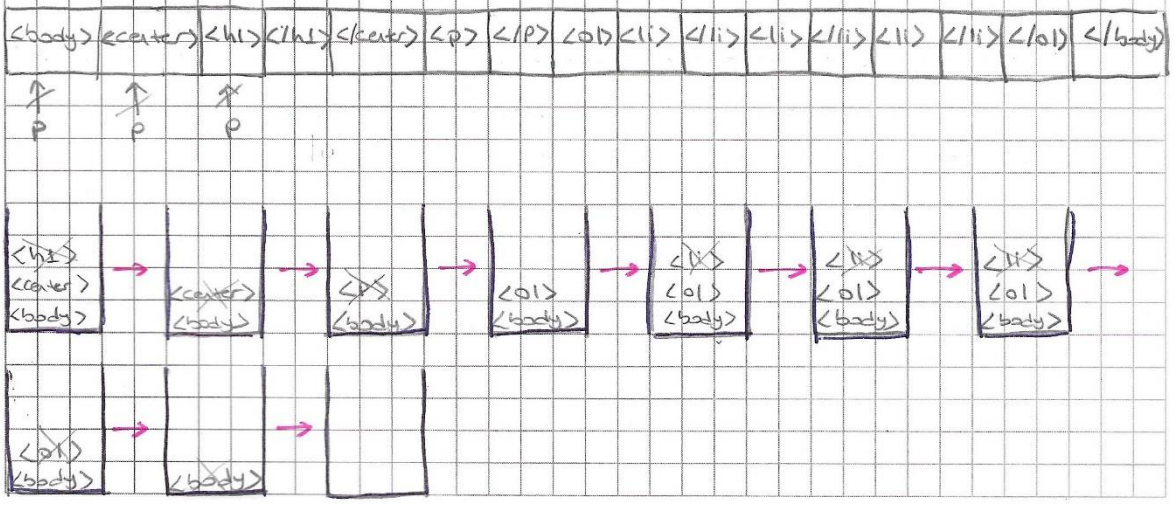
Top=1
Top=9
Top=13

// getHtmlTags html taglarını (etiketlerini) alıp vectorlerde tutuyor. Satır satır kopyalıyor, "<" sembolünü arıyor. "<" sembolünü bulduktan sonra ">" sembolünü arıyor, Tags vektorünü vectore push back yapıyor.

```
vector<string> getHtmlTags ()
{
  vector<string> tags;
  ifstream fin ("a.html");
  string line;
  while (getline(fin, line))
  {
    int pos = 0;
    int ts = line.find ("<", pos);
    while (ts != string::npos)
    {
      int te = line.find (">", ts+1);
      tags.push_back (line.substr (ts, te-ts+1));
      pos = te+1;
      ts = line.find ("<", pos);
    }
  }
  return tags;
}
```

```
bool isHtmlMatched (const vector<string> &tags)
{
    Linked.Stack S;
    typedef vector <string> :: const_iterator Itr;

    for (Itr p = tags.begin(); p != tags.end(); ++p)
    {
        if (p->at(1) != '/')        // '/' farklı ise push yap. Kapama
            S.push (*p)             taglerini S'ye push yap. (S=yığın)

        else
        {
            if (S.empty()) return false;
            string open = S.top().substr(1);
            string close = p-> substr(2);
            if (open.compare(close) != 0) return false;
            else S.pop();
        }
    }

    if (S.empty()) return true;
    else return false;
}
```

- tags.begin → Vectorün ilk elemanını işaret eder.
- tags.end → Son elemanına kadar dönsün
- p->at(1) → p'nin işaret ettiği tagın 1. karakteri (0. indisi)
- substr(1) → </h1>
  0 1 2 3 4



QUEUES

Ekleme kuyruğun sonuna, silme kuyruğun başından yapılır, Yığında S, kuyruk veri yapısında C kullanıyoruz. Kuyruk veri yapısı FIFO algoritmasıyla çalışıyor. [First in First out : ilk giren ilk çıkar.]

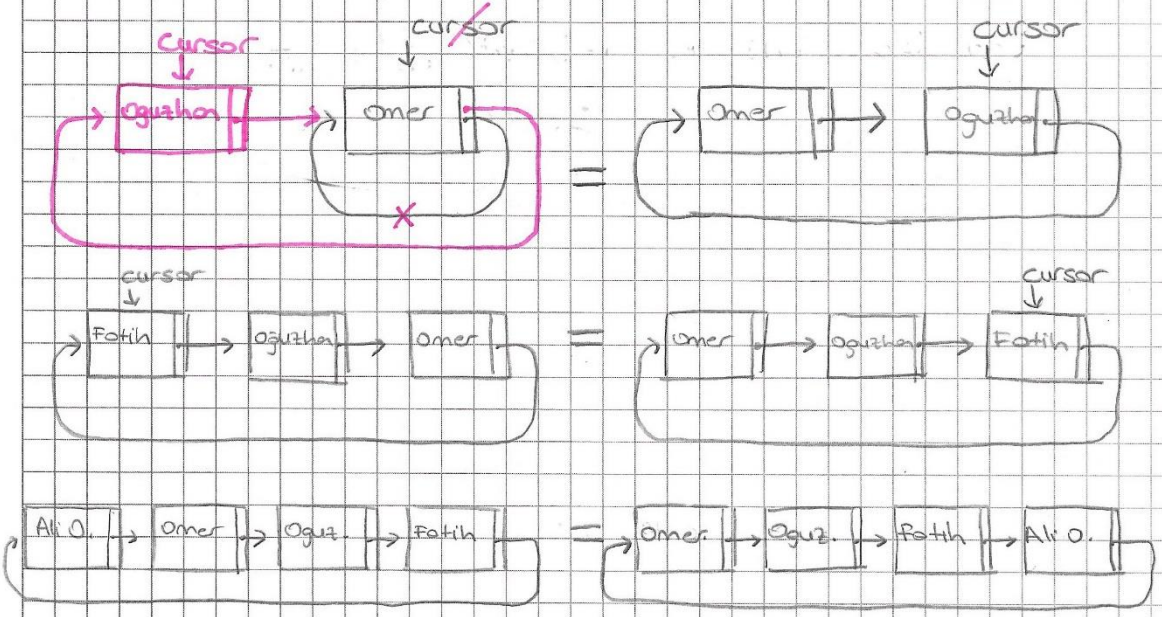
Enqueues → ekleme
Dequeues → silme

* Dairesel bağlı listelerle aynı çalışan add mi remove mu?

– Dairesel bağlı listelerde add , addfront olarak yapılıyordu yani listenin başına ekleme yapılıyor. Remove ise removefront yapılıyordu, yani listenin başından siliyordu. Queues te ise sondan ekleme baştan silme yapıldığı için remove aynı çalışır.

```
struct CircularlyLinkedQueue
{
  CircularlyLinkedList C;
  int n;

  CircularlyLinkedQueue ();
  int size () const;
  bool empty () const;
  const string & front ();
  void enqueue (const string & e);
  void dequeue ();
};
void CircularlyLinkedQueue :: dequeue ()
{
  if (empty())
  {
    cout << "dequeing of empty queue!" << endl;
    return;
  }
    C.remove ();
    n--;
}
```

```
int main()
{
  CircularlyLinkedQueue ....

    Queue.enqueue ("Omer");
                  ("Oguzhan");
                  ("Fatih");
                  ("Ali Osman");

    Queue.cPrint();
```



DOUBLE_ENDED QUEUES

Kuyruğun başından sonundan ekleme yapmak.

insertFront → D.oddfront(e,i);
n++;

insertBack → D.oddBack(e,i);
n++;

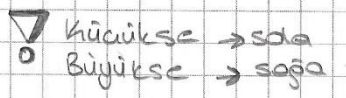
removeFront → D.remov-front();
n--;

removeBack → D.removeBack();
n--;

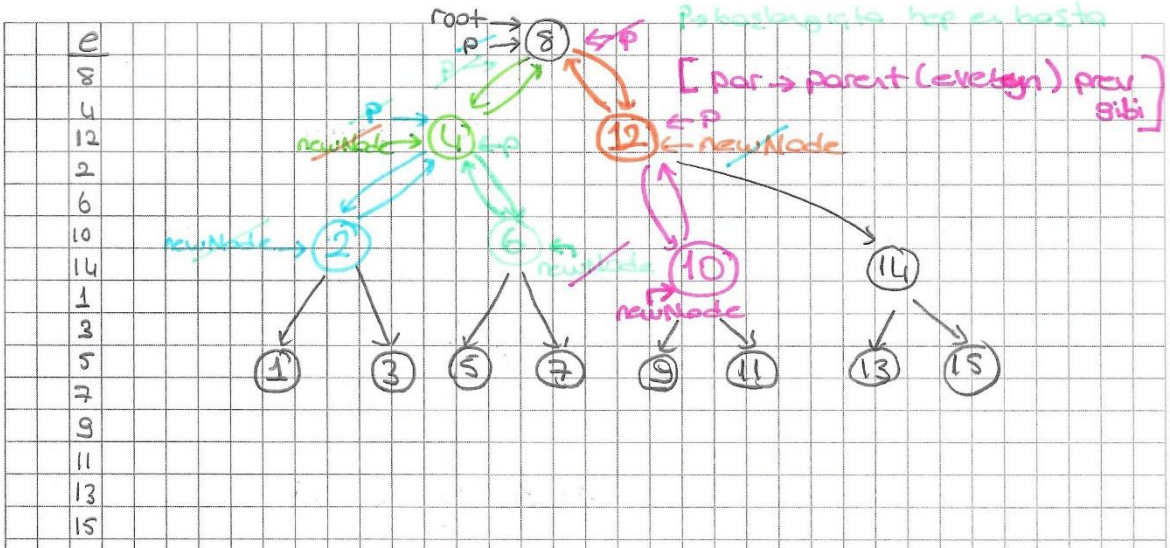
CHAPTER-7

Ağaç Veri Yapısı

addBelowRoot
addRoot → ağaçta düğüm oluşturmak için
deleteNode → silmek için



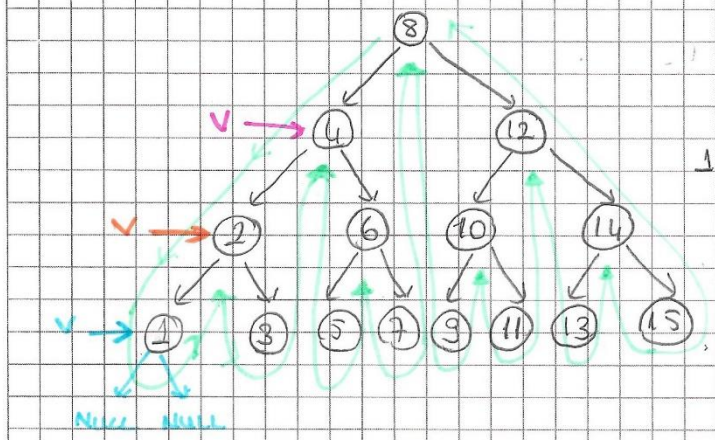

```
Void BinaryTree :: addBelowRoot (TreeNode * p, int e)

  TreeNode * newNode = new TreeNode;
  newNode -> elem = e;

While (true)

if (e < p -> elem)  // p başlangıçta ağacın rootu

if ( p -> left == NULL)

 p -> left = newNode;
newNode -> par = p;
break;

else  p = p -> left;

else

if (p -> right == NULL)

P -> right = newNode;
newNode -> par = p;
break;

else  p = p -> right;


n += 1;
```



Ağaçta gezinme için 4 tane fonksiyon var.
- inorder, preorder, postorder, Euler Tour

INORDER

```
Void BinaryTree :: inorder (TreeNode* v) const
{
  if (v->left != NULL) inorder (v->left);
    cout << v->elem << " ";
  if (v->right != NULL) inorder (v->right);
// Ağaçtaki düğümleri küçükten büyüğe kayıt yapacak
```



// sol - baba - sağ

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

≈ Çocukla işi bittiğinde çağırıldığı yere return yapıyor.

PREORDER

```
Void BinaryTree :: preorder (TreeNode* v) const
{
cout << v->elem << " "; //ilkini cout yap
  if (v->left != NULL) preorder (v->left); // sol cocugu var mı ?
  if (v->right != NULL) preorder (v->right);
```

// Baba - sol - sağ

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

1 → 2nin sol cocugundan cağırdık.
3 → return
5 → return
7 → return

POSTORDER

```
  Void BinaryTree :: postorder (TreeNode * v) const
{
  if (v-> left != NULL) postorder (v->left);
  if (v-> right != NULL) postorder (v->right);
  cout << v->elem << " ";
}
```

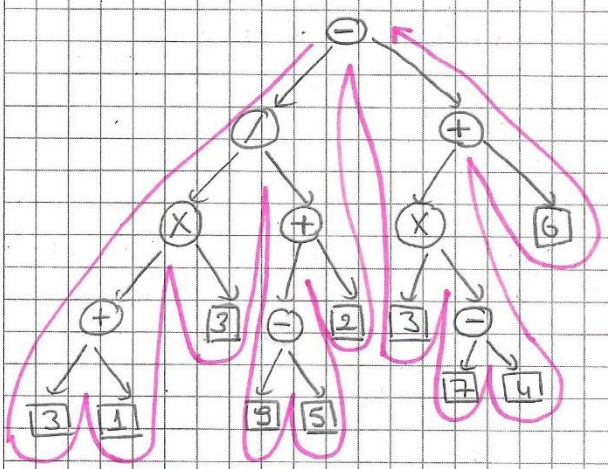
// sol - Sağ - Baba

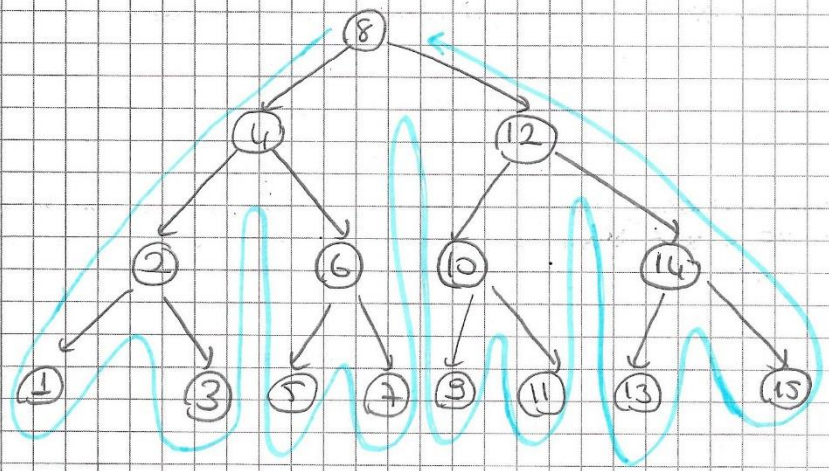
1 3 2 5 7 6 4 9 11 10 13 15 14 12 8

EULER TOUR

```
  Void BinaryTree :: eulerTour (TreeNode * v) const
{
cout << v->elem << " ";
if (v->left != NULL)
{
EulerTour (v->left);
cout << v->elem << " ";
}
```

```
    if (v->right != NULL)
{
    callerTour (v->right);
    cout << v->elem << " ";
```





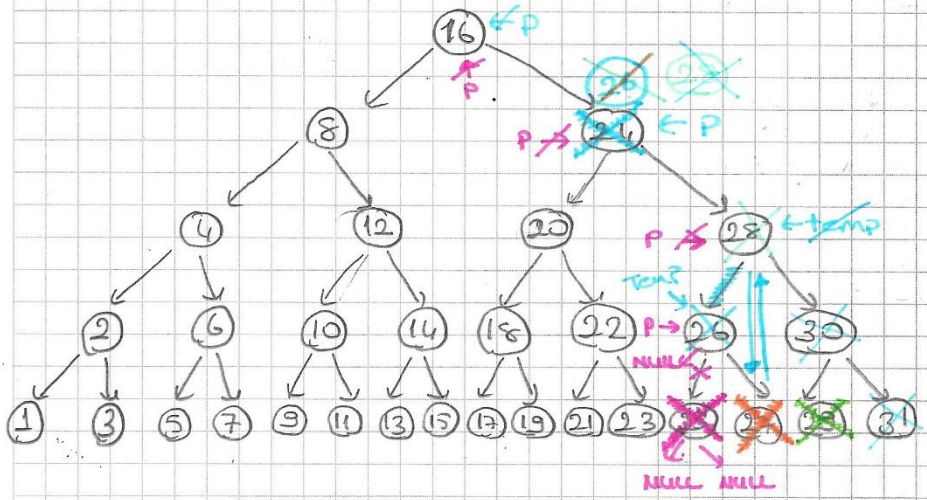
8 4 2 1 2 3 2 4 6 5 6 7 6 4 8 12 10 9 10

11 10 12 14 13 14 15 14 12 8

Düğüm Silmek

Çocuğu varsa çocuğu için yeni baba setlemeniz gerekebilir. Kendisinden büyük en küçük düğümü kendi yerine koy.

25 - 24 - 27 - 26 - 29 - 30 - 31 - 28 ⇒ sıra ile silicez.



Eğer çocuklu düğümü siliyor olsaydık kendisinden en yakın büyük düğümü yerine koyuyoruz yani 8'i silseydik yerine 9'u koyup çocuk olan 9'u silicektik.

VİZE SONRASI

STL

List / vector / stack / queue

```
# include <list>
# include <vector>
# include <stack>
# include <queue>
```

STL List Samples

```
DoublyNode Node;

list<DoubyNode> stl_list;

Node.elem = "Paul";
Node.score = 720;

stl_list.push_back <Node>;
```

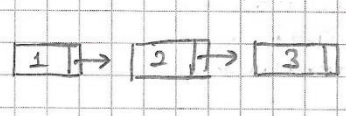
List Priority Queue
(Öncelikli kuyruk)

e = eklenecek elemanı
p = başlangıç elemanı

```
Struct List PriorityQueue
{ std :: list <int> L;

  int size () const;
  bool empty () const;
  void insert (const int & e);
  const int & min () const;
  void remove min ();
  bool isLess (const int & e, const int & f) const;
  void print ();
}

bool ListPriorityQueue :: isLess (const int &e, const int &f) const
{
  if (e<f) return true;
  else return false;
} // Öncelikli kuyruğa verileri sıralı eklerken
```

```
void ListPriorityQueue::insert(const int& e)
{
    std::list<int>::iterator p;
    p = L.begin();
    while (p != L.end() && !isLess(e, *p)) ++p;
    L.insert(p, e);
}
```

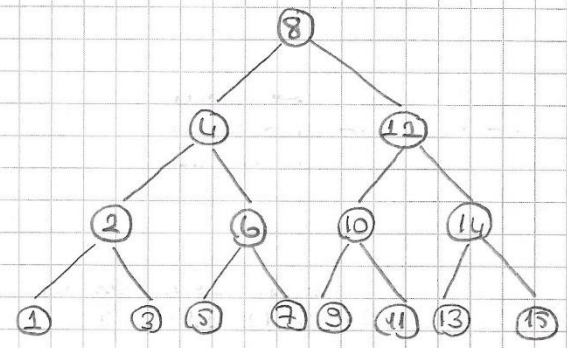
```
priorityQueue.insert(3);
(2);
(1);
```

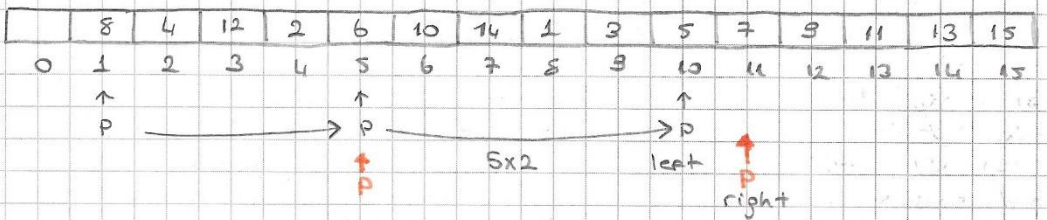



```
priorityQueue.print()

priorityQueu.min(); // 1
```

Heap Priority Queue

Heap, öncelikli kuyruğun ağaçta tutulmuş hali





```
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();
bool hasright(const position &p) const
    return 2*idx(p)+1 <= size();
bool isRoot(const position &p) const
    return idx(p) == 1;
position root() return pos(1);
position last() return pos(size());
void addlast(const int &e) V.push_back(e);
void removelast() V.pop_back();
```

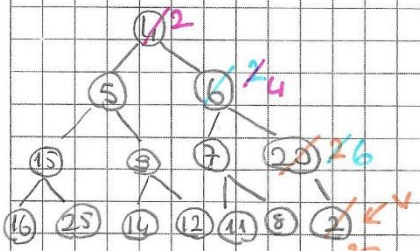
idx() → vectorun indisini geri döndürür.
pos → pointer döndürür.

left ➡ parametre olarak verilen p'nin sol çocuğu (6)
right ➡ parametre olarak verilen p'nin sağ çocuğu (7)
parent ➡ Baba

isRoot index değeri 1 mi? boolean değer döndürür

OR// Heap.insert 4,5,6,15,9,7,20,16,25,14,12,11,8,2

```
void __ ::insert(____)
{ 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;
  }
}
```



	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

4 için while 'a girmedi, çünkü root
5 in parent'ı 4 old. için u = 4
while 'a gir.
5 < 4 mü? (isLess)
v = 6 babası 4
6 < 4 mü? false → true (break yapar.)

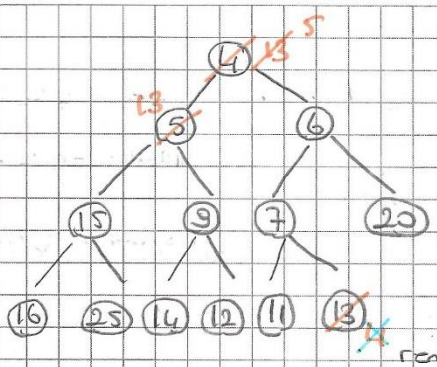
swap yeni eklenen elemanı babasından küçükse yukarı taşır.

size = 2
2 < 20 evet → hayır break yapmaz
swap yapar.

min eleman root ta tutulduğu için 2 yukarı kadar çıkar

```
void :: --- removeMin()
{
  if (size() == 1) T.removeLast();
  else
  { position u = T.root();
    T.swap(u, T.last());
    T.removeLast();
    while (T.hasLeft(u))
    { 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;
    }
  }
}
```

4 5 6 15 8 7 20 16 25 14 12 11 13
13 13 13 13 14
5 8 12



~ 4'ü silerken 4 ile 13 swap sonra 4'ü sil.

~ 13 ü çocuklar ile karşılaştırarak swap yapar. Küçük çocuk ile (5)

removeLast(4);

5 < 13 true / swap yap.
9 < 15
9 < 13 true / swap.

12 < 14 v=12
12 < 13 true / swap

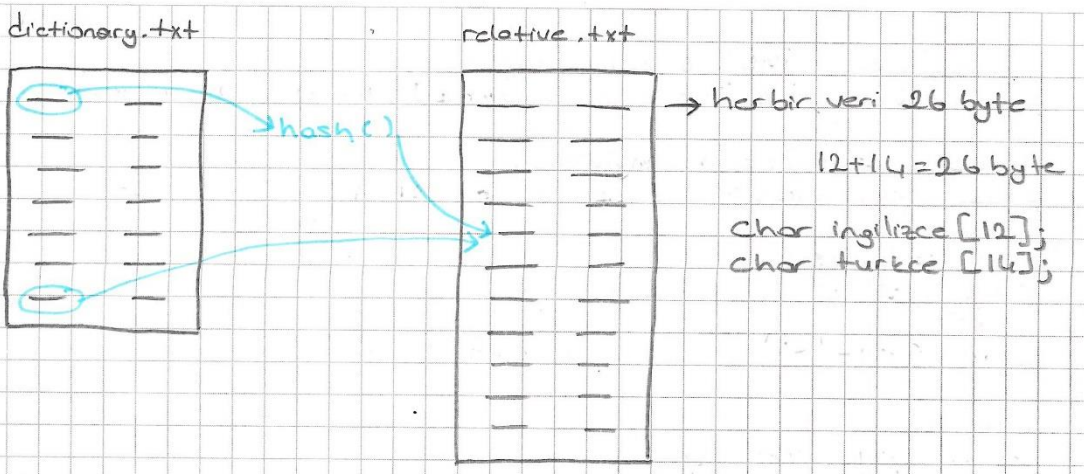
11.2 = 22 while'dan çık
size = 13

Hashing (Sözlük Uygulaması)

↳ İngilizce girilen kelimenin türkçesini veren bir uygulama.

Hashing → Kelimenin İngilizcesini parametre olarak alıyor, Hash Fonksiyonunu kullanarak kelimenin ASCII karşılığı üzerinden bir bağıl adres üretiyor. Bu bağıl adres ilgili kayıda dictionary.txt listten karşılığını yazıyor.

```
Hash Fonk;

int Hash (char * key ) // R(key) fonksiyonu
{
int sum= 0;
For ( int j=0 ; j<4 ; j+=2)

sum (sum +10 * key[j] + key[j+1] ) ;
sum = sum % nRel ; // nRel değeri genellikle asal sayı seçilir. Mod'dan dönen değerler daha iyi bir dağılım sergiliyor.
return Sum ;
}
```



→ Çakışma Çözümleme Yöntemleri

1) Lineer Probing : Bir sonraki adrese bakar, boş adres arar.

2) Double Hashing : Farklı Hash() fonksiyonları kullanılır.

Bağıl adres uzayı = 0 --- 40

```
Struct word
{
    char ingilizce[12];
    char turkce[14];
} kelime;

Void RelativeOlustur()
{
int collisions = 0;
dic = fopen("dictionary.txt", "r");
rel = fopen("relative.txt", "w+"); // w+ hem okuma hem yazma

For(int i=0; i<n Rel; i++) // Her bağıl adrese * basar. Boş olup
{                          olmadığını anlamak için
  FSeek(rel, i*sizeof(kelime), 0);
  Fputc('*', rel);
}
  printf("\n\n"); // i→ bağıl adresi temsil eder

while(!Eof(dic))
{
```

```
fscanf (dic, "%s" & kelime.ingilizce);
fscanf (dic, "%s" & kelime.turkce);

Adres = Hash (kelime.ingilizce);
fseek (rel, Adres * sizeof (kelime), 0);

Temp = Adres;
c = fgetc (rel);

while (c != '*')
{
//LINEAR PROBING

printf ("%4d ADRESINE %s\t\t CAKISMA YUZUNDEN LINEAR PROBING YAP \n",
        Adres, kelime.ingilizce);
collisions ++;

Adres = (Adres + 1) % nRel;
if (Adres == Temp)
{
printf (" DOSYA DOLU ! \n");
return ;
}

fseek (rel, Adres * sizeof (kelime), 0);
c = fgetc (rel);
}
printf ("%4d ADRESINE %s\t\t ADRES BOS OLDUGU ICIN YAZ \n",
        Adres, kelime.ingilizce);

fseek (rel, Adres * sizeof (kelime), 0);
fwrite (&kelime, sizeof (kelime), 1, rel);

strcpy (kelime.ingilizce, "          ");
strcpy (kelime.turkce, "          ");
}
fcloseall ();

printf ("_____");
printf ("_____");
printf ("_____");

getch ();
}
```

0	stack	—	10	class	—	20	signal	—	30	friend	—
1	*		11	gate	—	21	sorting	—	31	*	—
2	*		12	artificial	—	22	*		32	ambigous	belirsiz
3	*		13	external	—	23	*		33	list	—
4	*		14	linked	—	24	*		34	objeict	—
5	inheritance	—	15	overloading	—	25	array	—	35	client	—
6	sibling	—	16	recursion	—	26	*		36	*	—
7	compiler	—	17	data	—	27	binary	—	37	queue	—
8	server	—	18	priority	—	28	null	—	38	child	—
9	tree	—	19	root	—	29	order	—	39	circuit	—
									40	parent	—

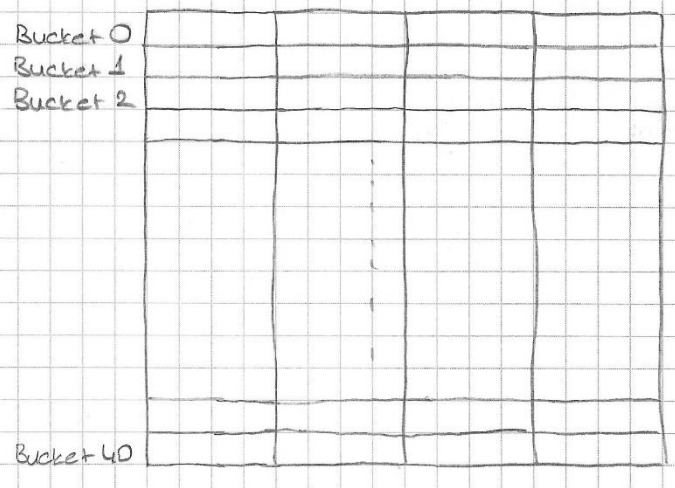
→ ilk başta hepsinde * var
→ kayıt atandıkca siliniyor.
→ *: o adresin boş olduğunu gösteriyor.

// Hashing with Bucket Adressing

Her boşil adreste birden fazla kayıt tutuyoruz. Her boşil adres birden fazla kayıt tutunca Linear Probingi hızlandırıyor. Her boşil adres 4 kapasiteli olucak. Birden fazla boşil adres kayıt tuttuğunda buna bucket deir.

Hash fonksiyonunun döndürdüğü değer → Bucket

- Her boşil adreste 1'den fazla paket tuttuğumuzu varsayalım.



~ Cash bellek mantığına benziyor. 0 adres için bucket içerisinde yer bulmak kolay. 2 inder Probing'i iyileştiriyor. Kayıt atamak kolay

~ Her kayıt için birden fazla yer ayırıyorsun. Bellek kullanımı n-1 kadar fazla.

! (n-1) boyut kadar fazla kayıt alanı

Avantaj = peşpeşe yerler bulabiliyorsun

Her bir bağıl adres max. kaç kere üretilebiliyor.

Optimum bucket size = 4

Bucket = 0 : * * * * [0]
Bucket = 1 : * * * * [0]
Bucket = 2 :
Bucket = 3 :
⋮
Bucket = 17 : data veri priority öncelik root kök signal sinyal [4]
Bucket = 18 : * root gök * * * [0]
⋮
Bucket = 25 : Array dizi * * * [1]

17 de root var. Tekrar root yazsaydık bir sonraki bucket'e ekler. root kök yerine gök olarak yazarsak

Bucket = 17 : data veri priority öncelik root kök signal sinyal
Bucket = 18 : root gök * * *

Size of Bucket = Bütün kelimeler için adres uzayı kadar dizi tutuyor. Bağıl adresin kaç kere üretildiğini hesaplıyor. Diziyi 1 arttırarak.

Hashinge çok benziyor farkı çarpma (*oBucket) var.

Bir Bucketten diğerine geçebilmek için oBucketle çarpmanız lazım.

_ kelime ekleme _

```
Fseek (rel, Bucket * sizeof(kelime) * oBucket , 0 );
Adres = Bucket * oBucket;
Temp = Bucket *
c = fgetc (rel);

While (c != '*')                    → 4 ile çarpılıyor
{
Adres = (Adres+1) % (nRel* oBucket);

if ( Adress == Temp)
{
   Printf ("Dosya Dolu! \n");
   return;
}
   fseek (rel, Adres*sizeof(kelime), 0);
   c= fgetc (rel);
}
   Printf ("Bucket = %d \t kelime = %s \n" Bucket, kelime.ingilizce);
```

Hash fonk. döndürdüğü değer oBucketle çarpılarak gidiyor. (Birinden diğerine atlarken veya Bucketin başına giderken 4 ile çarpıyoruz.)

_ Kelime Silme _

Kelime silme kelime sorgulamaya benziyor. Bucket üretilir, o bucketin başına gidilir, Bucket içerisinde aranır, yoksa diğer buckete gider.

2 tane root var soldan sağa doğru ilerlediği için ilk yazılı olan rootu siler.

Hashingden farkı; 2 veya daha fazla kayıt tutması.

// Hashing with Synonym Chaining

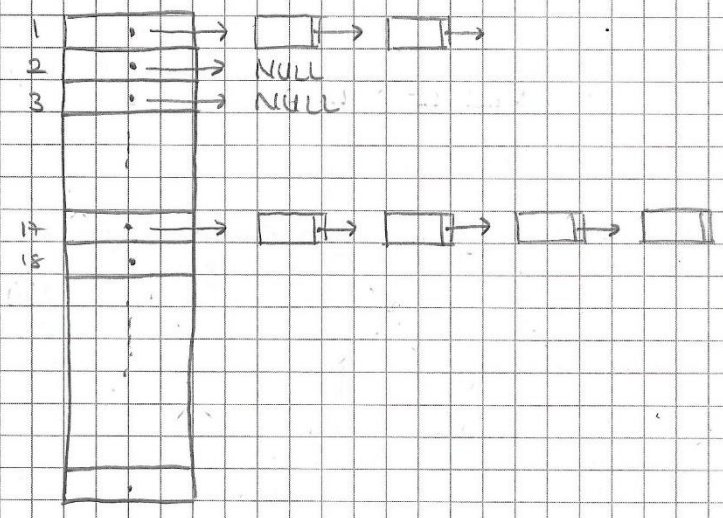
Çakışan kayıtlara synonym chaning diyoruz. Çakışan kayıtları synonym da tutuyoruz.

Avantajı: Hızlı sorgulama. Çakışan kayıtları listede tuttuğu için bucketten daha hızlı. Bucket dolusa çok aşağılara yazabilirsin synonym chaningde ayrı dosyaya yazılıyor.

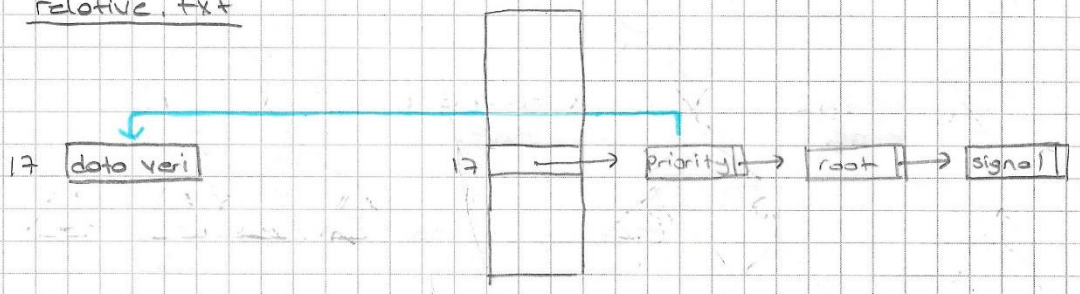
```
SinglyLinkedList * listofSynonyms;
int nlist = 0;

Void printSynonyms ();

int Hash (char* key) // R(key) Fonksiyonu
{
  int sum = 0;
  For (int j=0; j<4; j+=2)
     sum = (sum + 10 * key[j] + key[j+1]);

  Sum = sum % nrec;
  return sum;
}
```

İlgili listenin başını tutan pointerı gösteren dizi

*listsofSynonyms



relative.txt




```
struct SinglyLinkedList
{
    SinglyNode * head;
    SinglyLinkedList();
    bool empty() const;
    void print();
    void addBack(char* ing, char* tur, const int& i);
    void removeOrdered(char* ing, char* tur, const int& i);

    int n=0;
};
```

SPLAY TREES

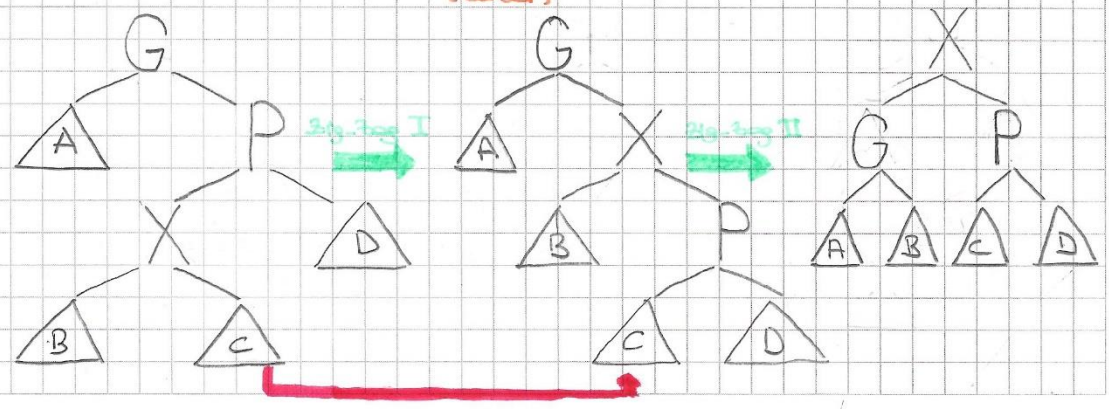
Her yeni eklediğimiz veriyi root'a kadar çıkarıyoruz.

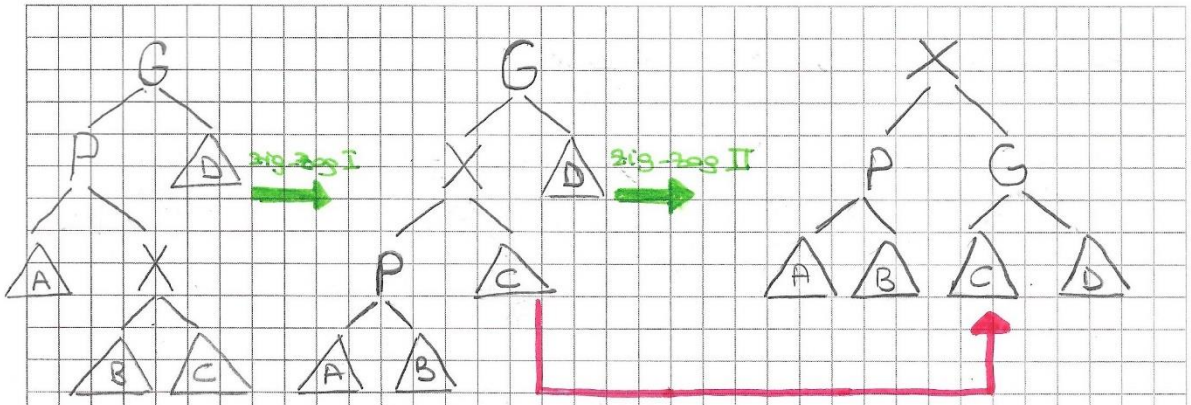
1-) Zig-Zag:

X is left child of a right child or right child of a left child

X

P (parent) (baba)



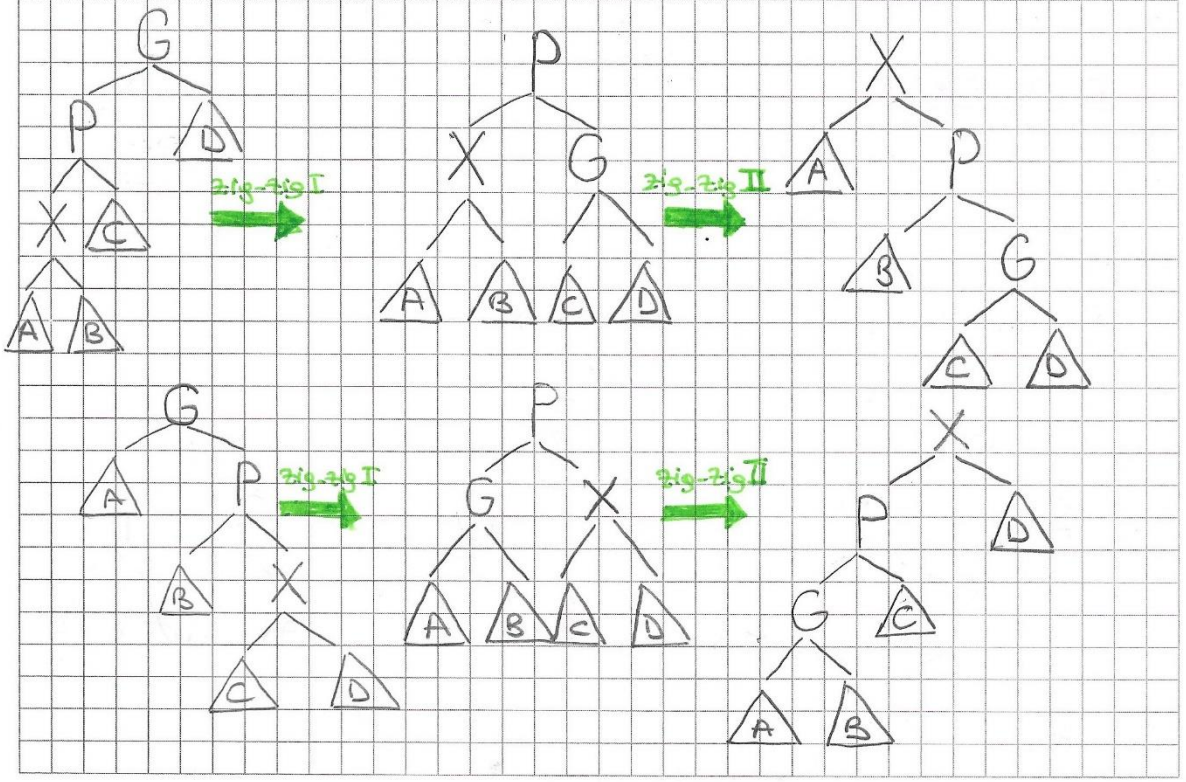


Amacımız ağacı dengelemek

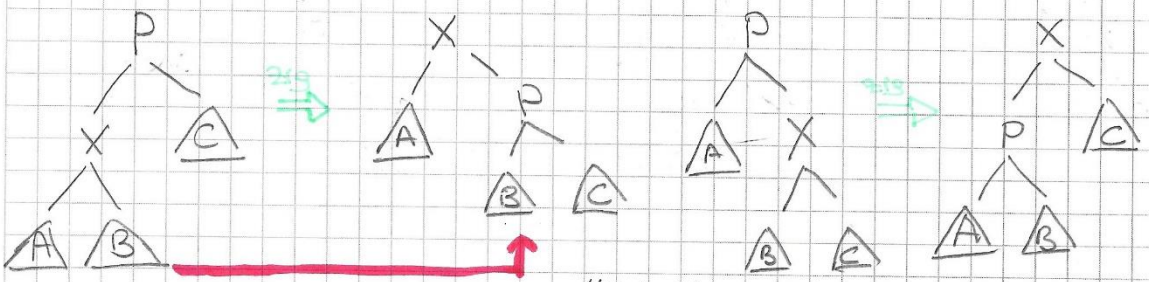
2_) Zig-Zig :

X is left child of a left child or right child of a right child

~ Zig-Zag 'dan farkı Önce baba (P) dedenin (G) seviyesine çıkıyor.



3. zig : X is child of the root

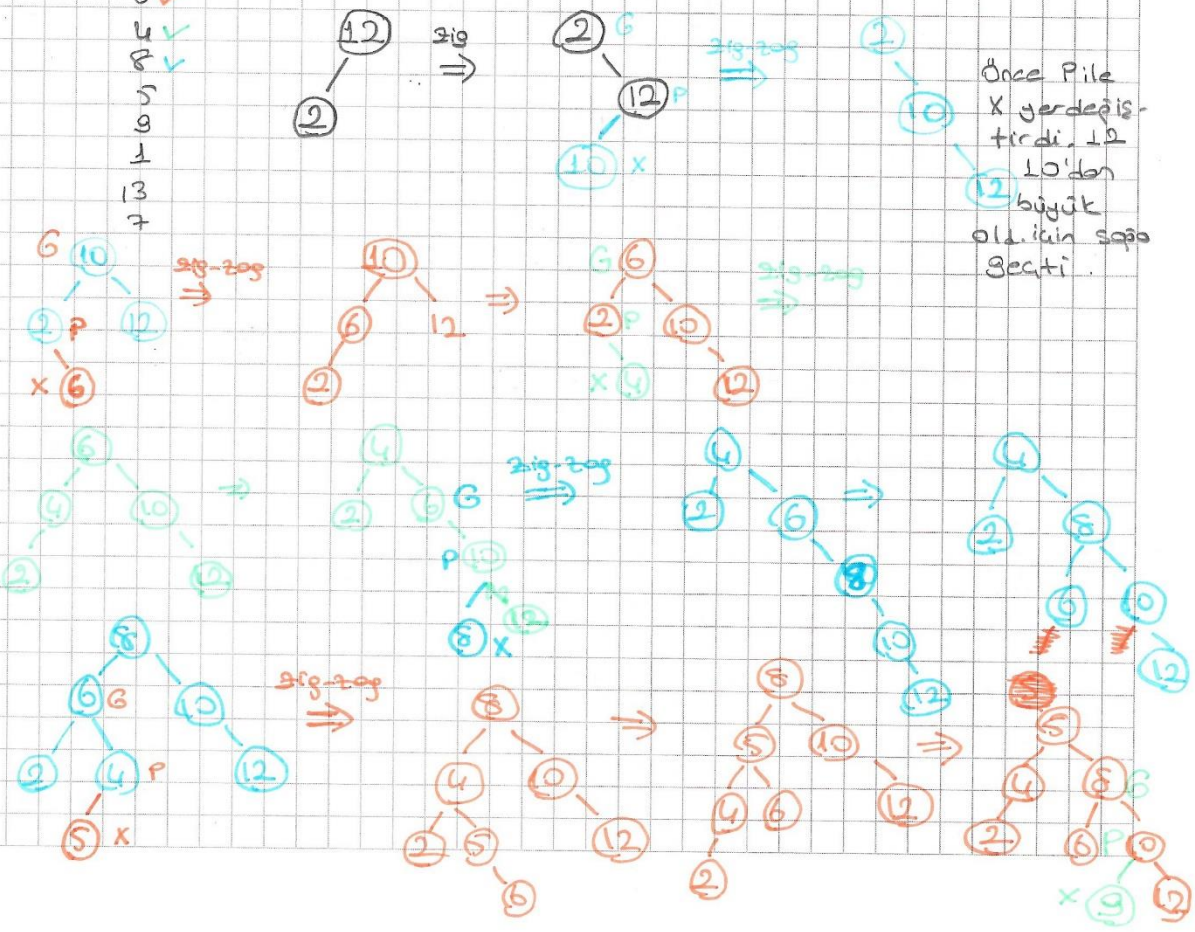


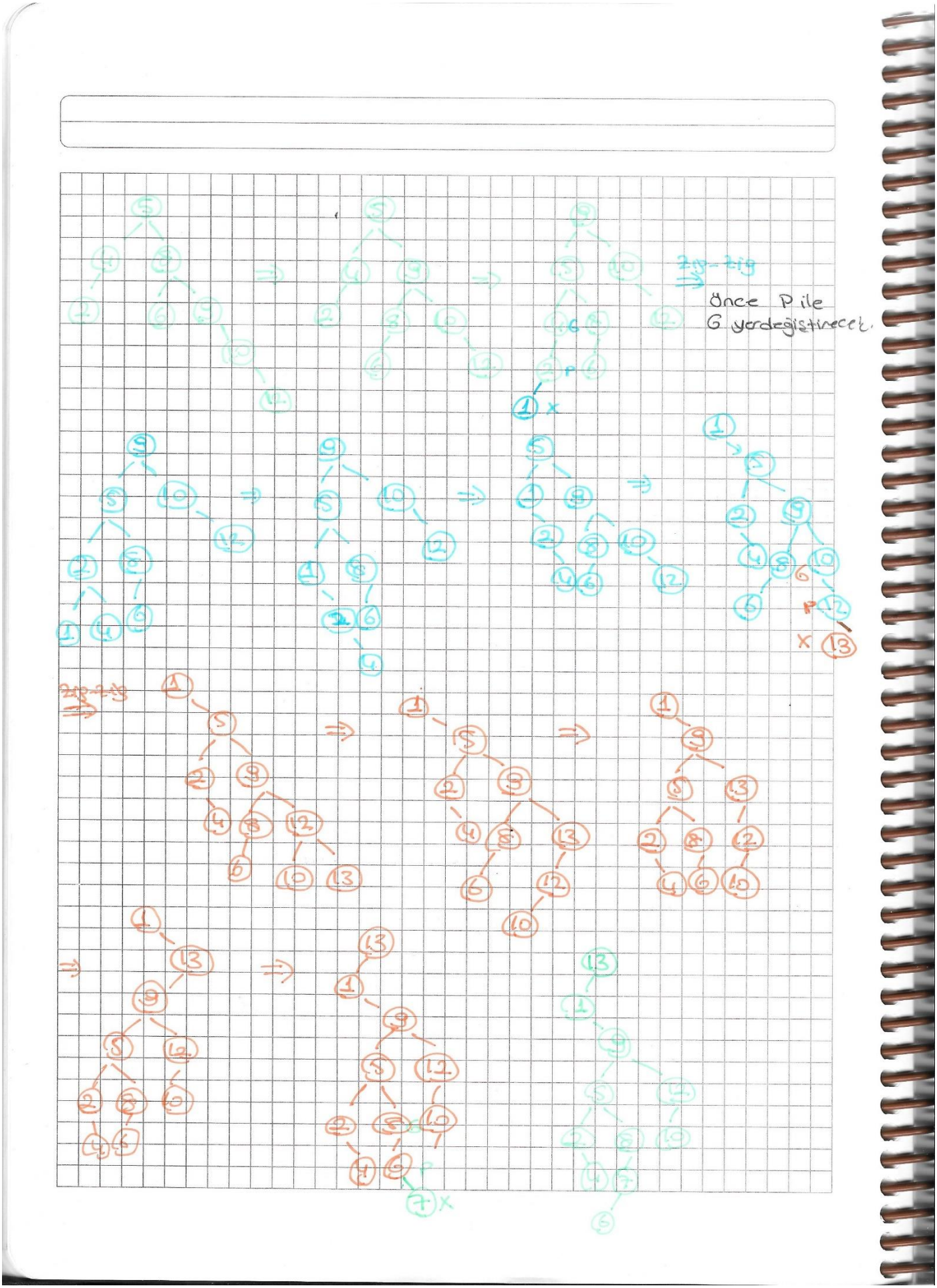
// Pile X in yerdeğiştirmesi

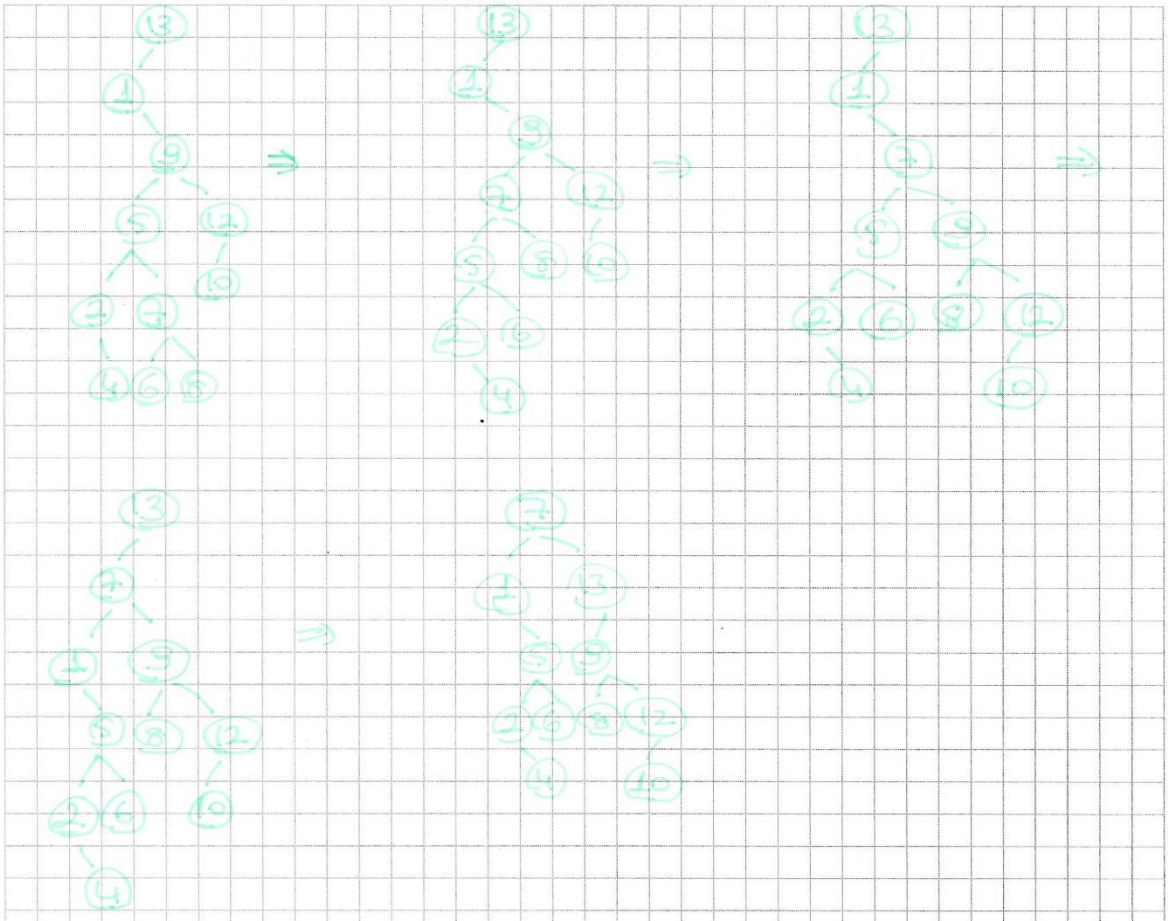
2012 Final
Sorusunun cevabı:

root 12 ✓
2 ✓
10 ✓
6 ✓
4 ✓
8 ✓
5
9
1
13
7

Yeni eklenen sayı büyükse sağa, küçükse sola.




Önce P ile
G yer değiştirecek.



```
// ZiG_ZAG (left_right)

  else if ((newNode == parent->right) && (parent == Gparent->left))
{
if(newNode->left != NULL) //newNode <-> parent I
  {
    parent->right = newNode->left;
    newNode->left->par = parent;
  }
  else parent->right = NULL;
    newNode->left = parent;
    parent->par   = newNode;
    Gparent->left = newNode;
    newNode->par  = Gparent;
```

```
    if (newNode->right != NULL;) //newNode <-> Gparent II
{
    Gparent->left = newNode->right;
    newNode->right->par = Gparent;
}
    else Gparent->left = NULL;

        newNode->par = Gparent->par;
        newNode->right = Gparent;
        Gparent->par = newNode;
}
//ZIG-ZAG (right-left)

else if ((newNode == parent->left) && (parent == Gparent->right))
{
    if (newNode->right != NULL) //newNode <-> parent I
    {
        Parent->left = newNode->right;
        newNode->right->parent = parent;
    }
        else parent->left = NULL;

        newNode->right = parent;
        parent->par = newNode;
        Gparent->right = newNode;
        newNode->par = Gparent;

        if (newNode->left != NULL) //newNode <-> Gparent II
{
        Gparent->right = newNode->left;
        newNode->left->par = Gparent;
}
        else Gparent->right = NULL;

        newNode->par = Gparent->par;
        newNode->left = Gparent;
        Gparent->par = newNode;
```

```
// zig - zig (left - left)

else if (newNode == parent->left) && (parent == Gparent->left))
{
  if (parent->right != NULL)  // Parent <-> Gparent I
  {
    Gparent->left = parent->right;
    Parent->right->par = Gparent;
  }
  else Gparent->left = NULL;
  newNode->par = Gparent->par;
  parent->right = Gparent;
  Gparent->par = parent;

  if (newNode->right != NULL) // newNode <-> Parent II
  {
  Parent->left = newNode->right;
  newNode->right->par = parent;
  }
  else  parent->left = NULL;
        newNode->right = parent;
        Parent->par = newNode;
}

// zig - zig (right - right)

else if ((newNode == parent->right) && (parent == Gparent->right))
{
  if (parent->left != NULL) // parent <-> Gparent I
  {
  Gparent->right = parent->left;
  Parent->left->par = Gparent;
  }
  else Gparent->right = NULL;
  newNode->par = Gparent->par;
  parent->left = Gparent;
  Gparent->par = parent;

  if (newNode->left != NULL) // newNode <-> parent II
  {
  parent->right = newNode->left;
  newNode->left->par = parent;
  }
```

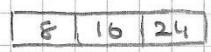
```
if (newNode->par = NULL)
    root = newNode;
else
{
    if (newNode->elem < newNode->par->elem)
        newNode->par->left = newNode;
    else
        newNode->par->right = newNode;
        parent = newNode->par;
        Gparent = parent->par;
}

// 216
    if (newNode->par == root)
    {
    if (newNode->elem < root->elem)
    {
    if(newNode->right != NULL)
    {
        root->left = newNode->right;
        newNode->right->par = root;
    {
        else  root->left = NULL;
              newNode->right = root;
              root->par = newNode;
    }
    else
    {
        if (newNode->left != NULL)
        {
            root->right = newNode->left;
            newNode->left->par = root;
        }
        else root->left = NULL;
        newNode->right = root;
        root->par = newNode;
    }
    else
    { if (newNode->left != NULL)
      {
        root->right = newNode->left;
        newNode->left->par = root;
      } else root->right = NULL;
        newNode->left = root;
        root->par = newNode;
    }
    root = newNode;
    newNode->par = NULL;
}
```

2_3_4 Tree

Herhangi bir düğümde max. 3 eleman tutulur. Solo küçük, sağa büyük gelir.

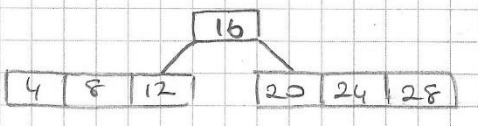
16, 8, 24, 4, 12, 20, 28, 2, 6, 10, 14, 18, 22, 26, 30, ...

insert → 8, 16, 24

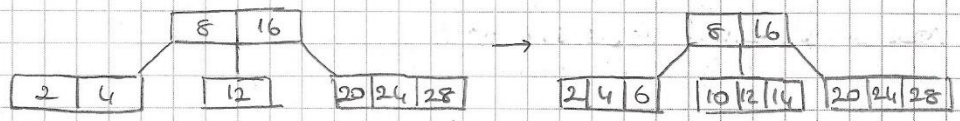


- İkiye parçalarken ortadakini yukarı çıkarıyoruz.
- Eleman eklerken üstten aşağı doğru bakılır

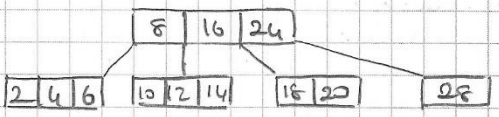
insert → 4, 12, 20, 28



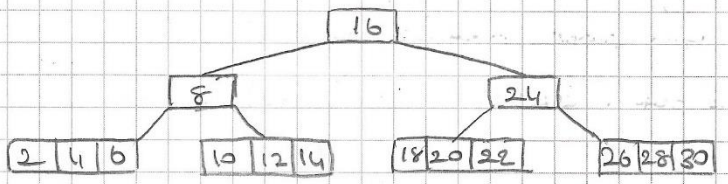
insert → 2, 6, 10, 14



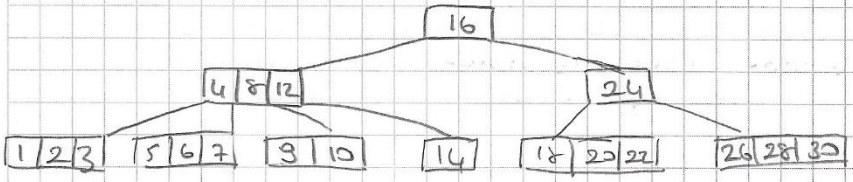
insert → 18 [16dan büyük sağa eklicez ama yer yok parçalıcaz]



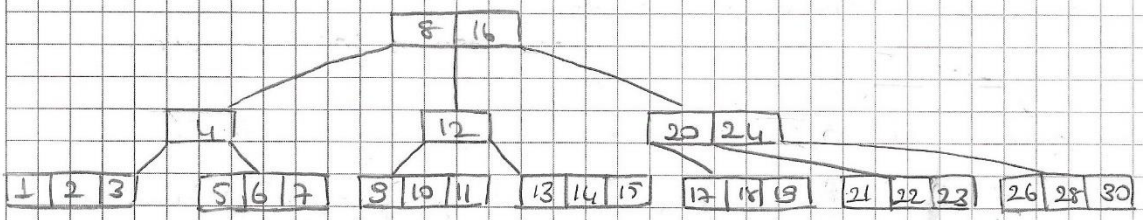
insert → 22, 26, 30



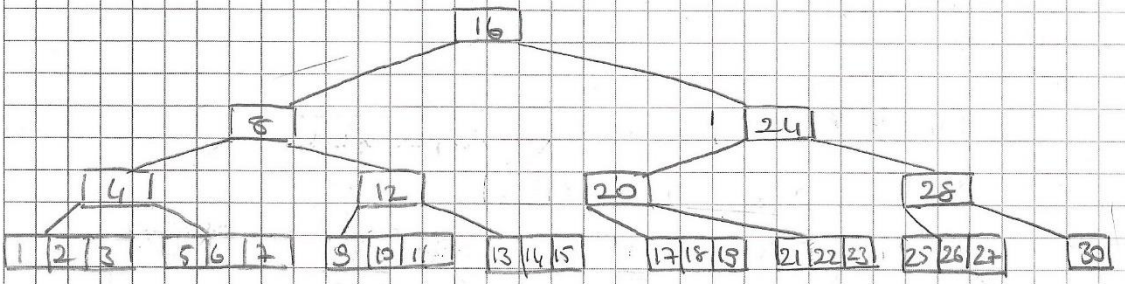
insert → 1, 3, 5, 7, 9



Insert → 11, 13, 15, 27, 18, 21, 23



Insert → 25



Remove ( ) :

~ External = En alt seviyede çocuğu olmayan düğümler.
~ Internal = Çocuğu olan düğüm.

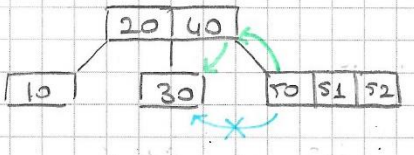
2-3-4 den düğüm silerken ;

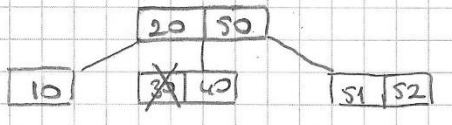
- External ise direkt sil
- Internal ise kendisinden büyük ya küçük düğümü yerine yaz ve sil.

// Bu iki durumda silinecek düğüm ararken (root dışında) tek elemanlı düğüme rastlarsan ;

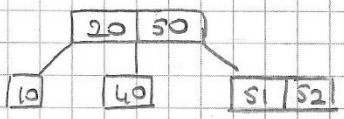
R1 : Kardeşten ödünç al.
R2 : Babadan ödünç al. (kardeş, baba, kendisi) 3 elemanlı düğüm oluştur.
R3 : Baba tek elemanlı, root ve kardeşte tek elemanlı ise (Kardeş, root, kendisi) 3 elemanlı bir root düğüm oluştur. Bu durumda ağacın seviyesi 1 azalır.

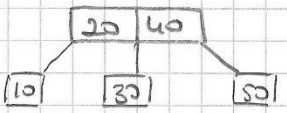
remove (30)





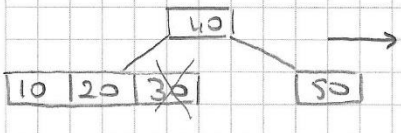
! Kardeşten ödünç al. (Rule 1)

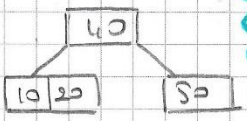




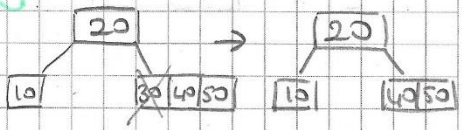
! Babadan ödünç al. (Babadan 20'yi de 40'da ödünç alabiliriz

~ 20'yi ödünç alırsak;

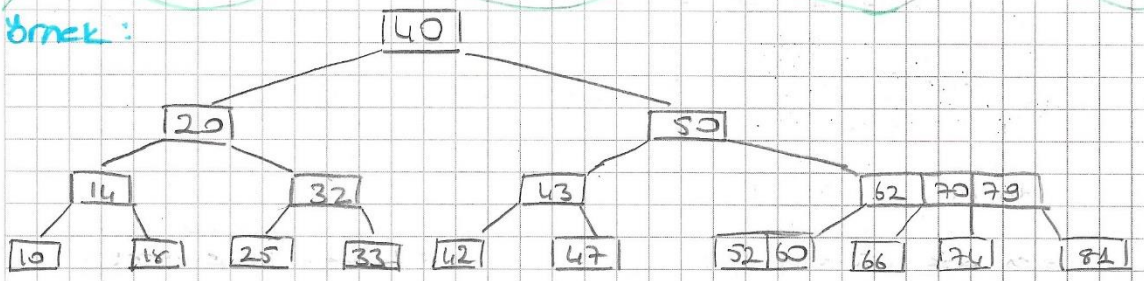




~ 40'ı ödünç alırsak

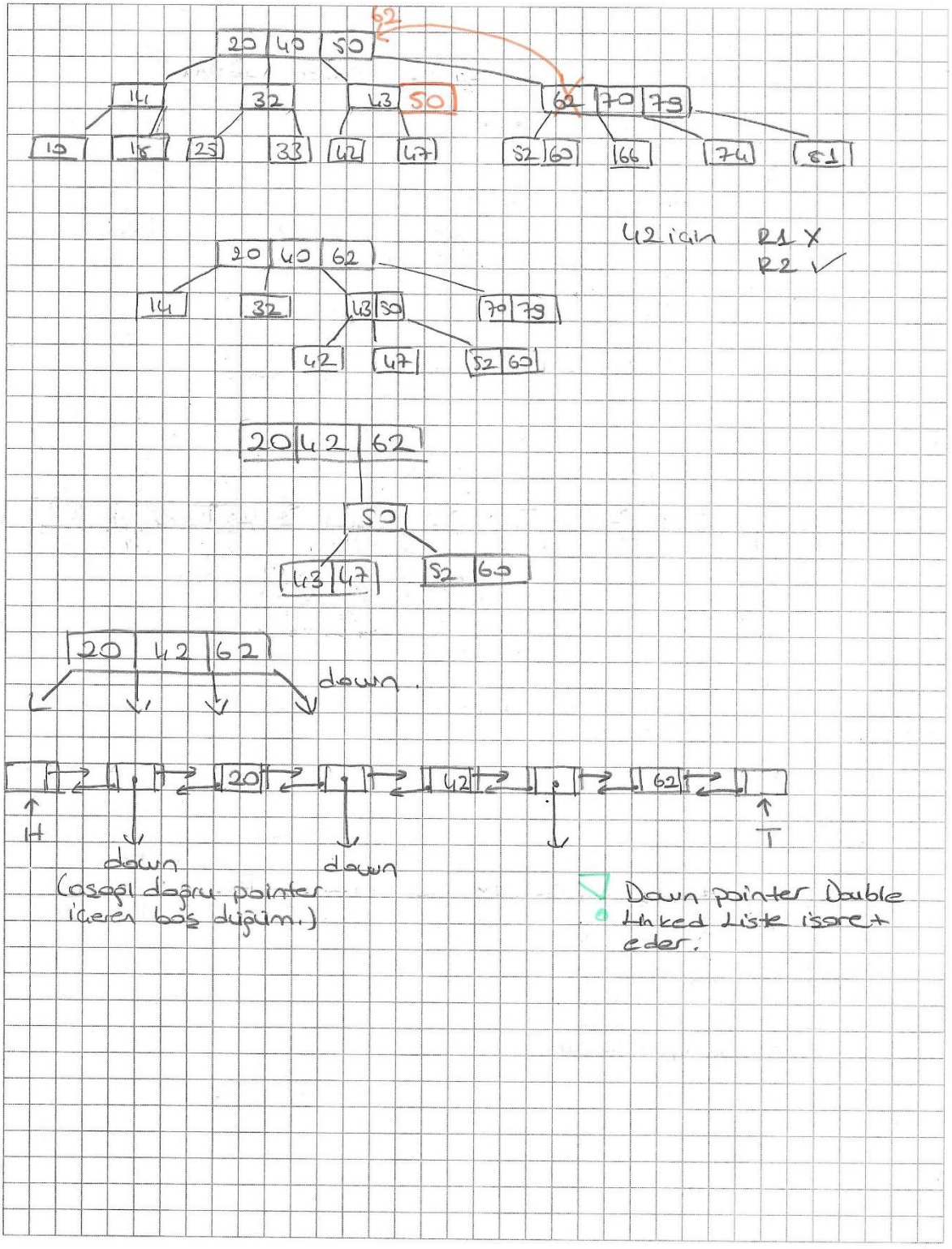


Örnek:

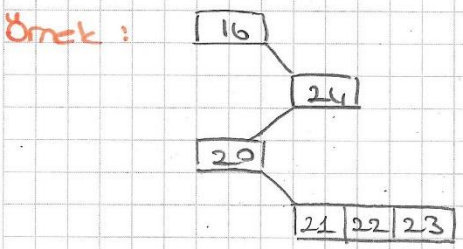


remove (40):
KBEK

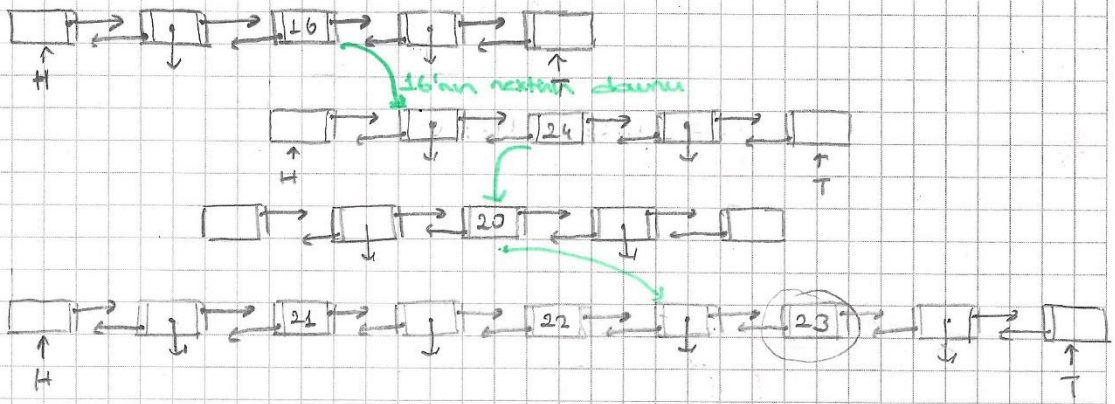
40'ın sağına bak [50] root değil → R3
R1 X
R2 X
↓
20-40-50 birleştir.


42
20 40 50
14 32 43 50
62 70 79
10 15 25 33 42 47 52 60 66 74 81
42 için R1 X
R2 ✓
20 40 62
14 32 43 50 70 79
42 47 52 60
20 42 62
50
43 47 52 60
20 42 62
down.
H
T
down
(aşağı doğru pointer içeren boş düğüm.)
down
Down pointer Double Linked List işaret eder.

Örnek :



16 dan 23e gidelim.



Örnek:

i
insert 16
8
24

```
*current = header -> next -> next;
while (current != trailer)
{   if
    else break;
}
```

234 ağacı kodu

- down pointer'ı başka bir düğüme işaret ediyor.
- insert order, double linked liste çok benziyor. Aradaki fark downlar.

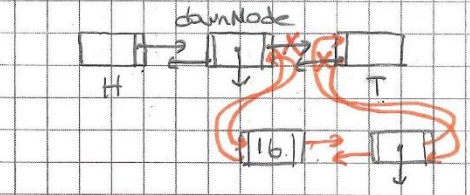
```
DoublyLinkedList :: DoubleLinkedList()
{
    header = newDoublyNode;
    trailer = newDoublyNode;
    header -> score = 0;
    trailer -> score = 0;

Doubly
    ;
    ;
}
```

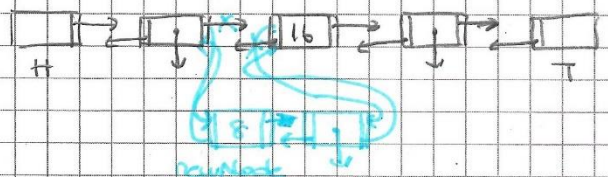
```
void DoublyLinkedList::insertOrdered(const int& i)
    DoublyNode *newNode = new DoublyNode;
        :
    else
        break;
}
    :
n++
```

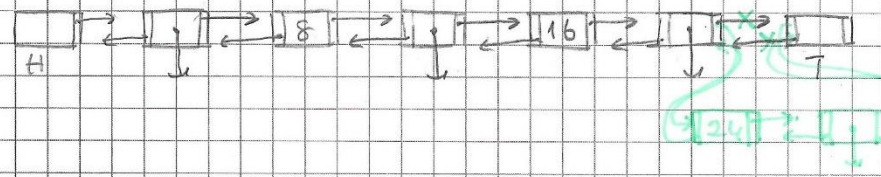
* 8-16-24 düğümüne insert(14) yapacağız.

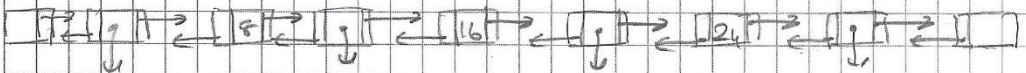
Önce parçalamaya ihtiyacın var.

p=root;









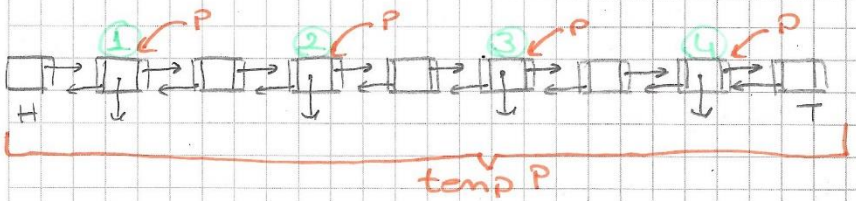
int DoublyLinkedList :: hNextScr (int e)

{

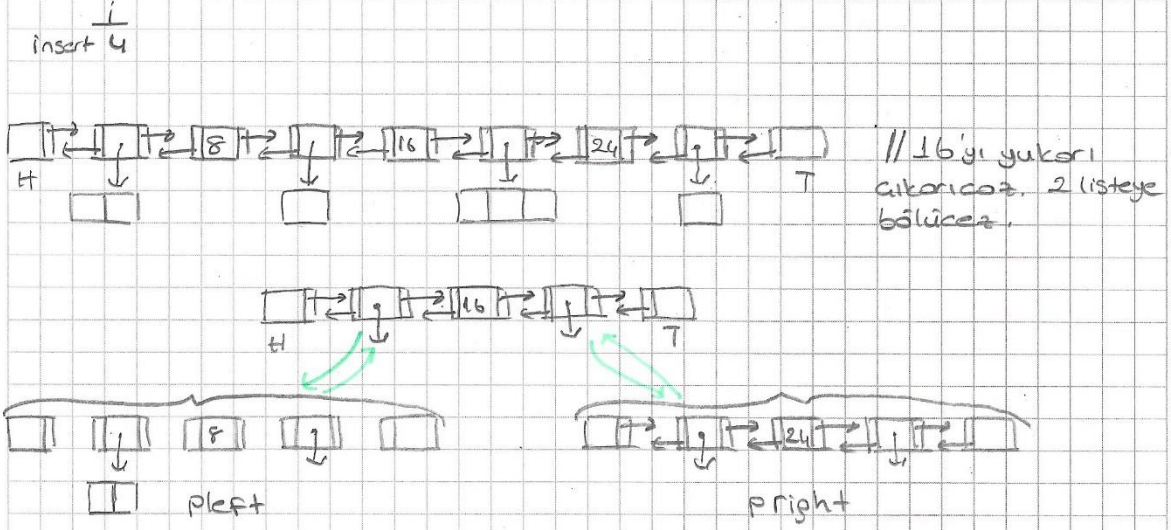
: 1. elemana erişmek için 2 next
2. " " " 4 next
3. " " " 6 next yapılıyor.

:: downptr (int e)

Down Pointera işaret eden elemanı döndürüyor.



Parent aşağıya doğru yani eklenicek elemanın bi alt seviyeye inip yanı orakın babasına işaret eder..



down pointerı = Listeye pointerı

Up Pointerı = düğüm'e pointerı

▽ root parçalıyorsan yeni liste oluşturuyorsun