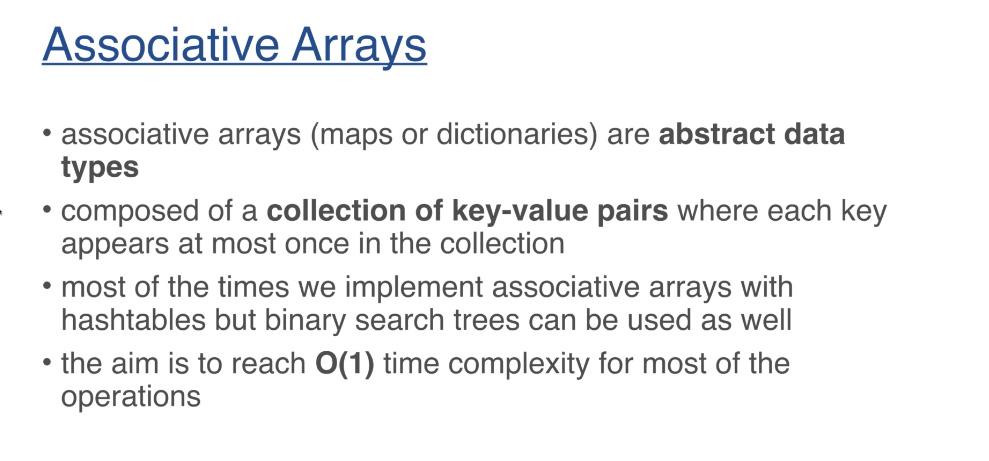
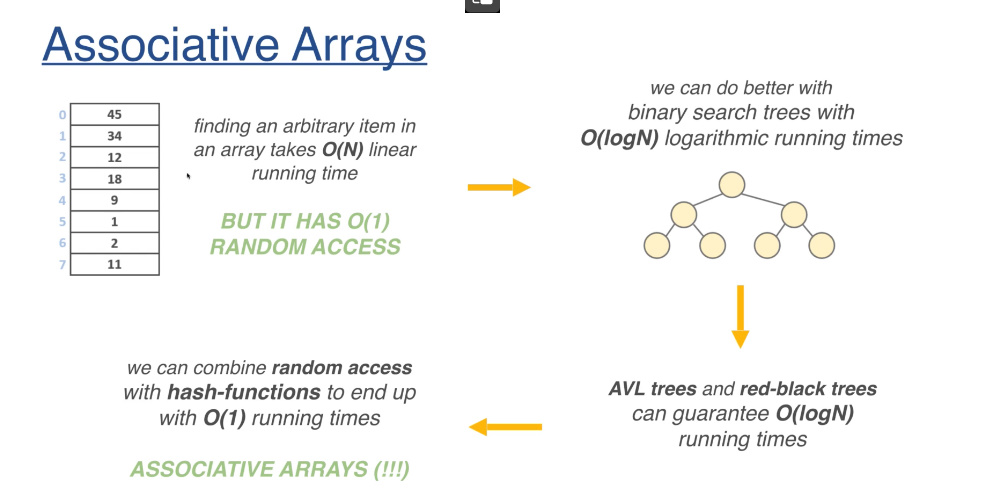
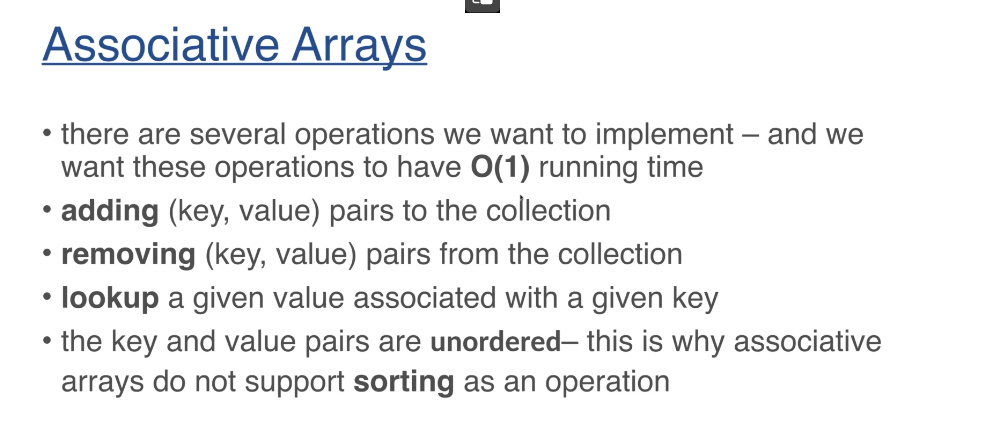
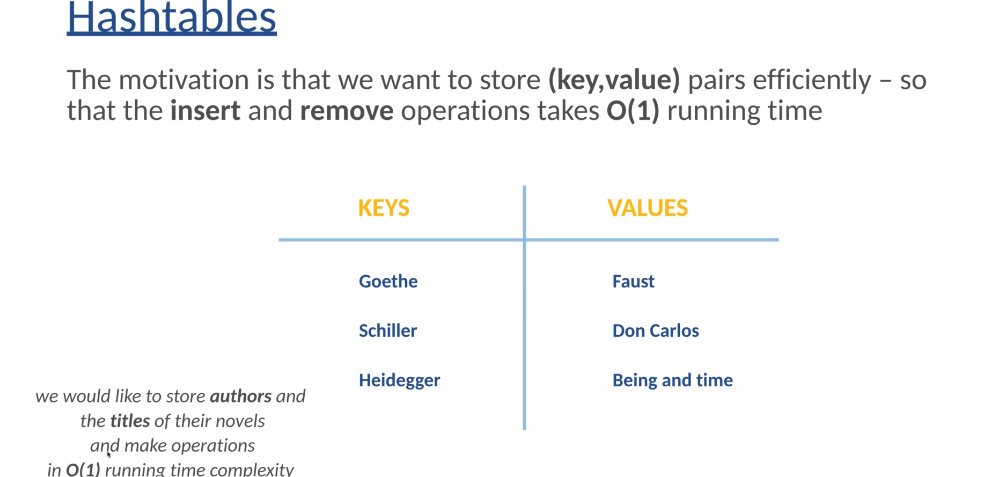
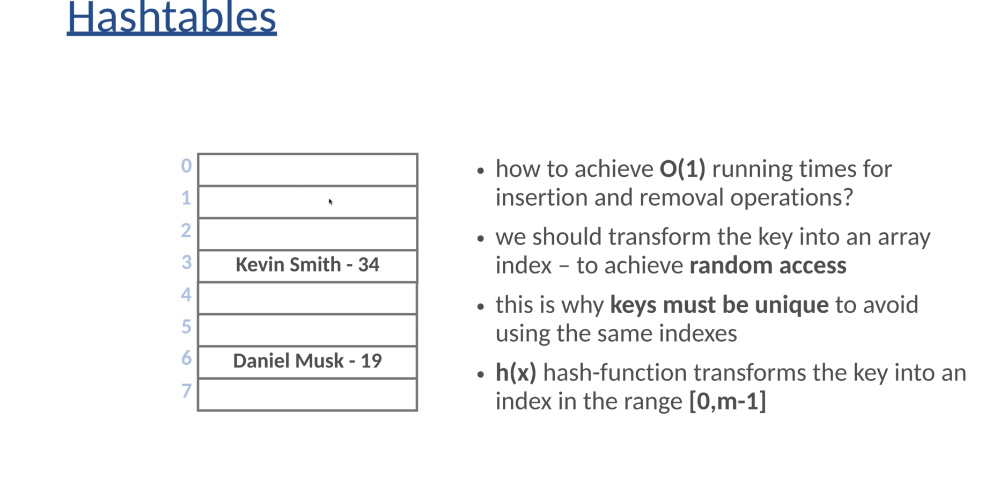
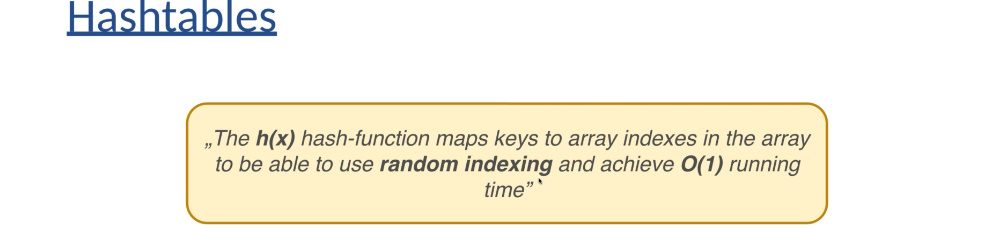
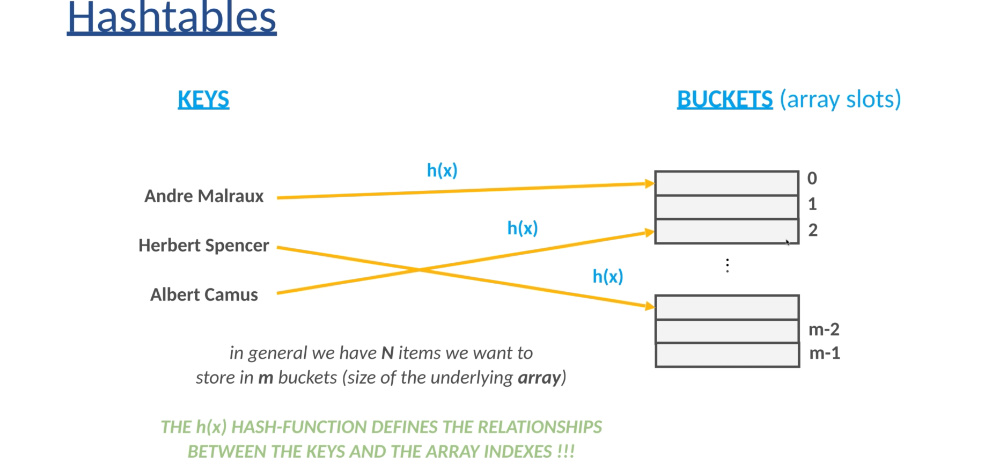
**Associative Arrays**

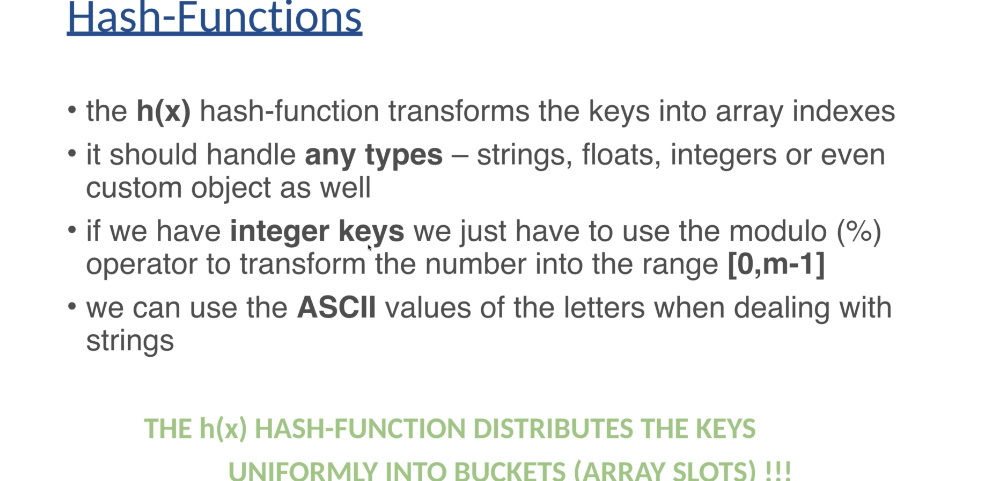
* 
* Associative – adica se asociaza unei key un index specific
* 
* Deci, combinarea accesarii unui item dupa array pe baza la hash functii face ca gasirea unui element sa fie O(1)
* Asta e cel mai rapid mod posibil de a gasi iteme
* 
* O problema la Associative Arrays e ca nu suporta sortarea

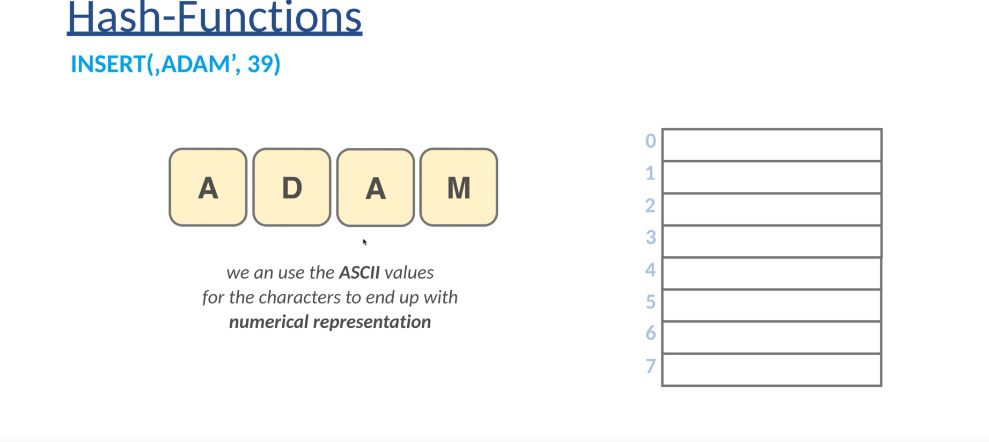
**Hash Tables**

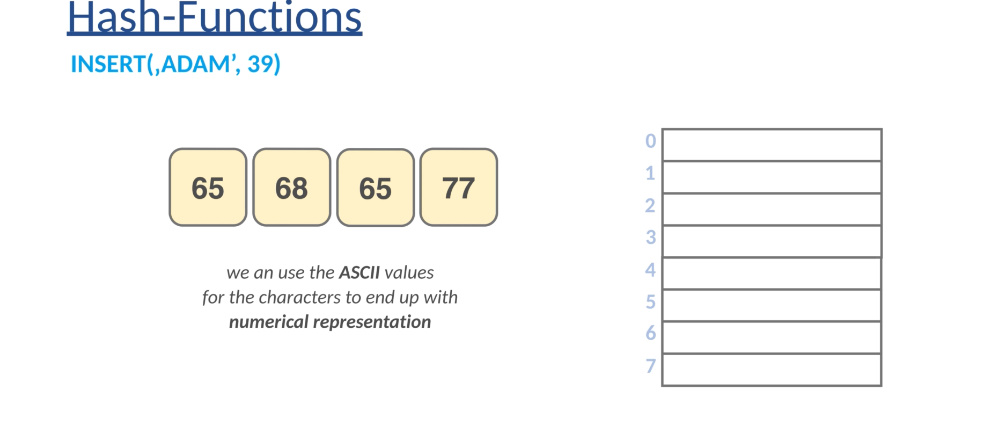
* 
* 

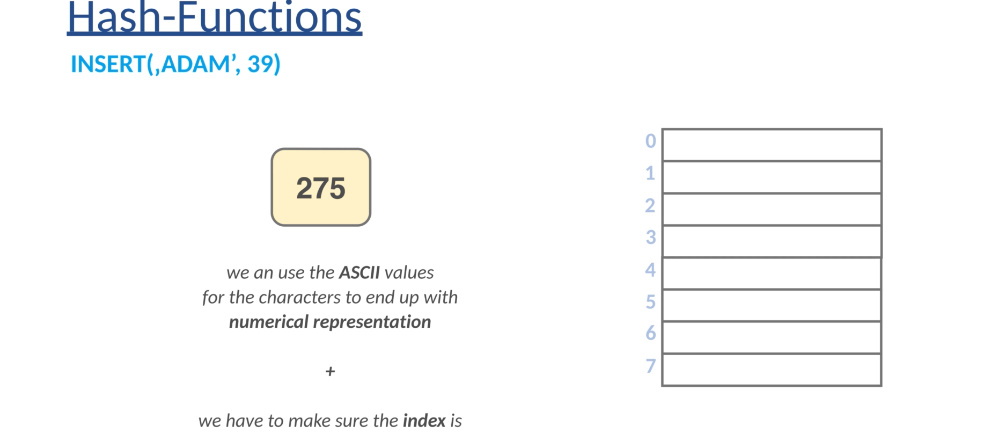


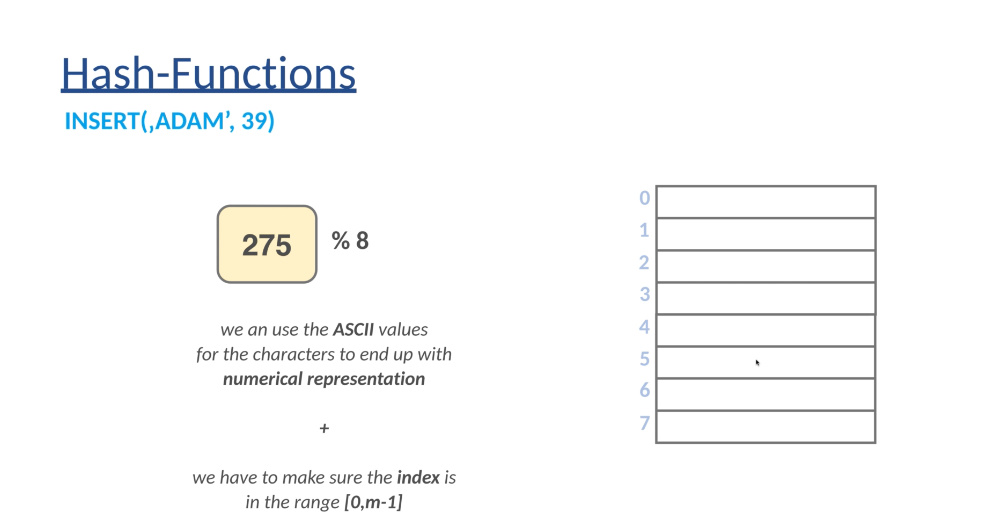
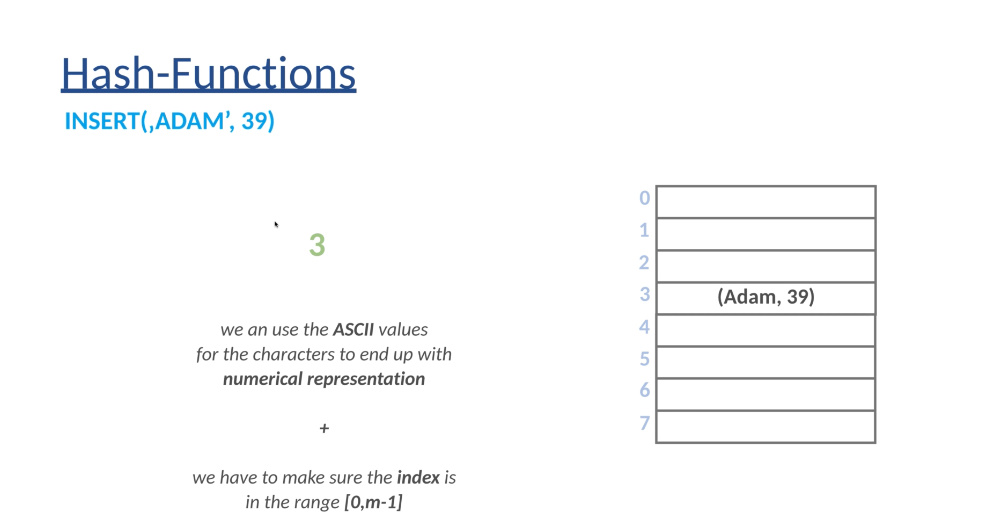






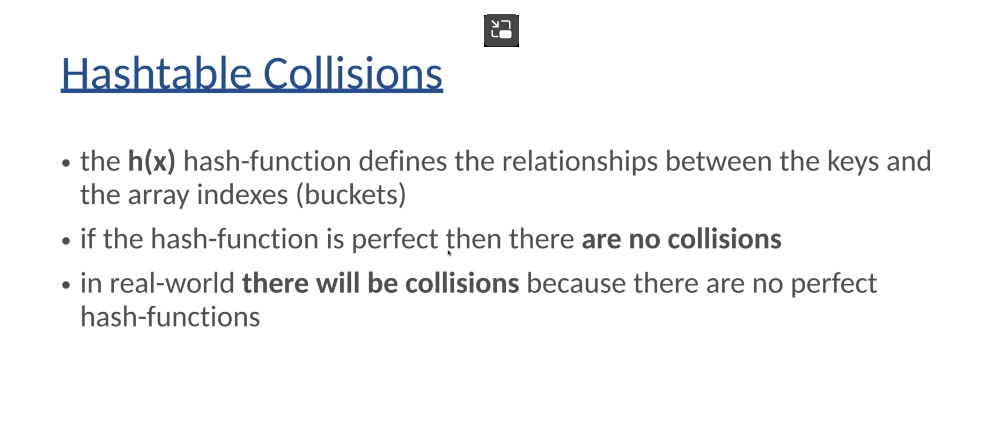


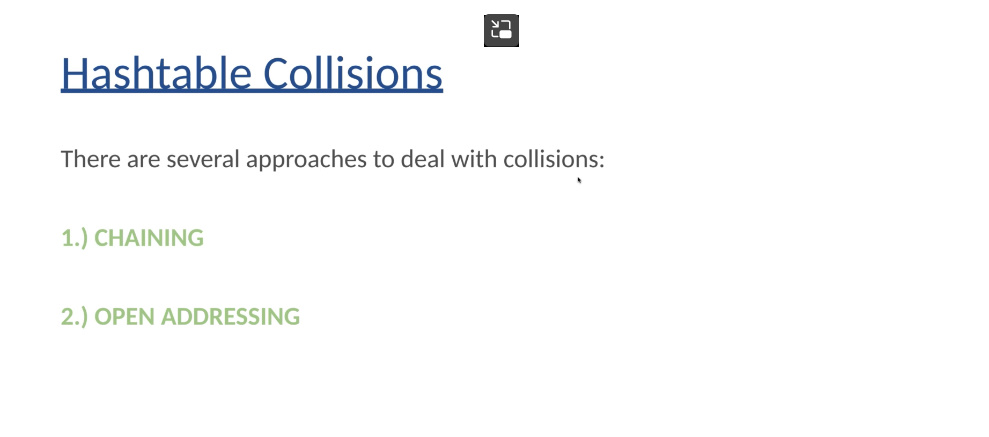


* 
* 
* Deci abordarea la HashTable e simpla. Se creaza un array de dimensiune constanta, apoi pentru fiecare item adaugat(key + value), pe baza la key, se va afla care este indexul la care acest item trebuie pus. De ex, am putea aduna codurile ASCII la fiecare caracter din key, sa facem % 8, daca array are maxim 8 elemente in el, si asa aflam indexul itemului

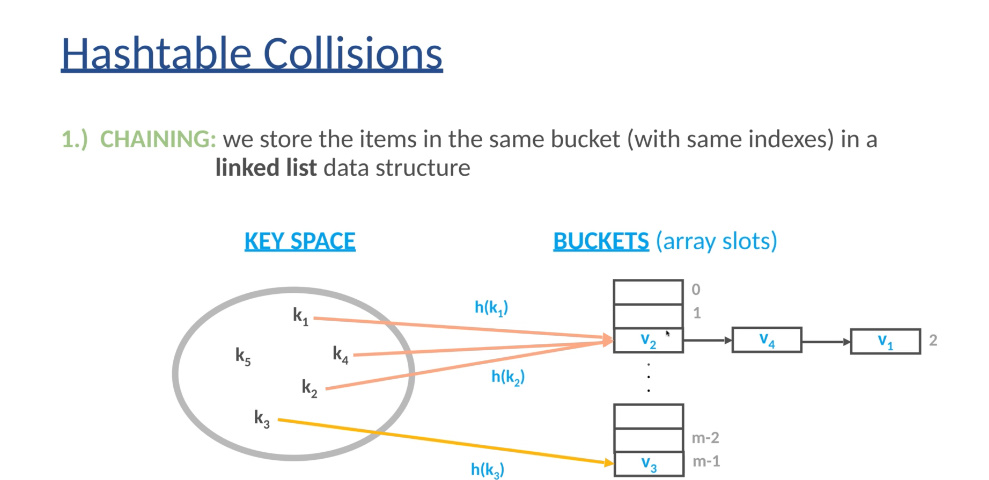
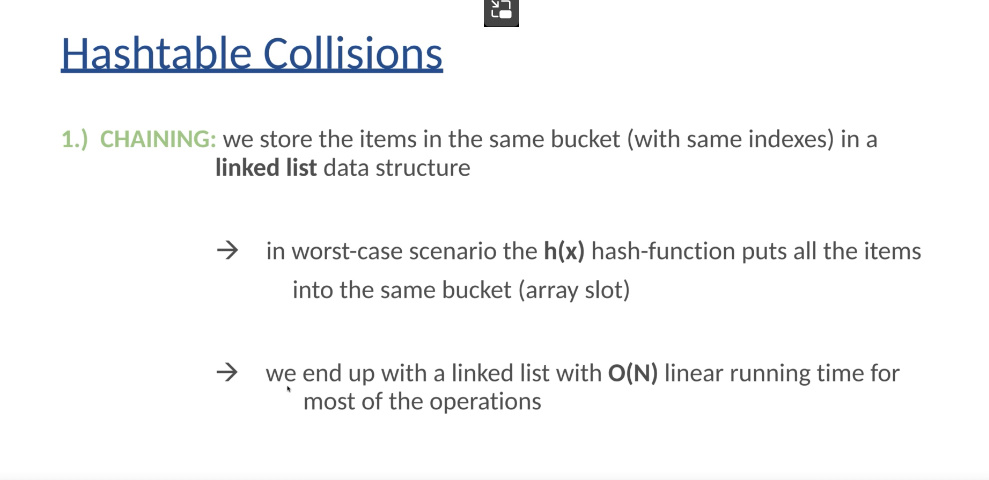
**Collisions**

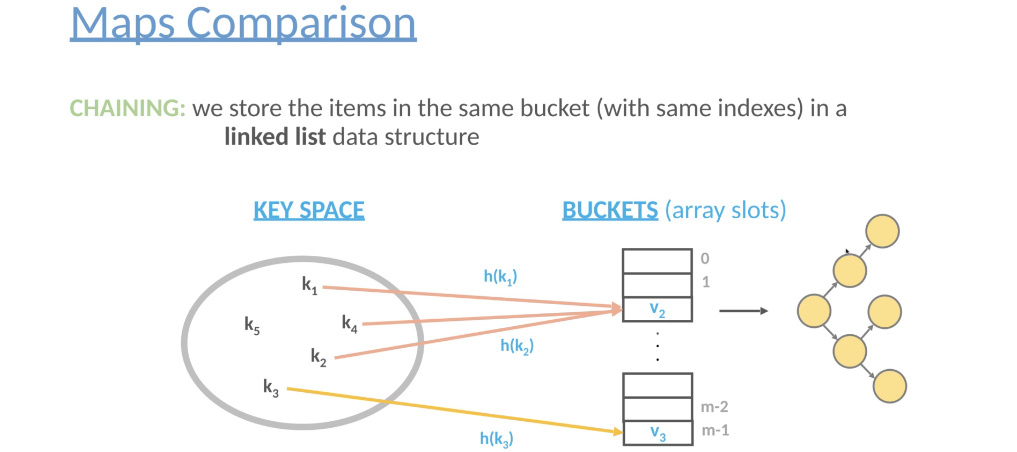
* Totusi, functiile Hash nu pot returna mereu un index diferit per item, mai ales ca avem un nr limitat de iteme in array.
* Asa dar, apare o coliziune cand un item trebuie pus la un index unde deja este un alt element





**Chaining**

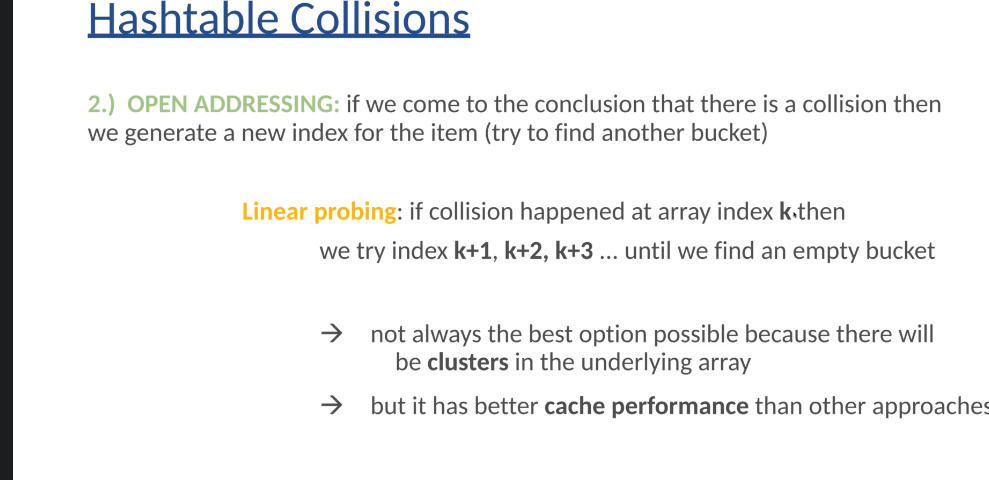
* Solutia data prevede ca sa stocam mai multe iteme in acelasi index cu un linked list. Adica, fiecare index va avea un linkedlist cu elemente in el. Nu propriu zis LinkedList<> din Collections, dar vom crea un Node class si fiecare element va fi adaugat in Node si fiecare node va avea referinta la urmatorul element.
* 
* 
* Deci, din cauza ca LinkedList au nevoie de memorie aditionala, si din cauza ca se poate intampla sa avem complexitatea chiar O(N) daca toate elementele au acelasi index, urmatoarea metoda de rezolvare a coliziunii e mai buna
* **Totusi, in Java, HashMap,HastTable ... foloseau Chaining pana la Java 8. Dupa Java 8, in loc de chaining se foloseste Red Black Tree pentru collisions**

****

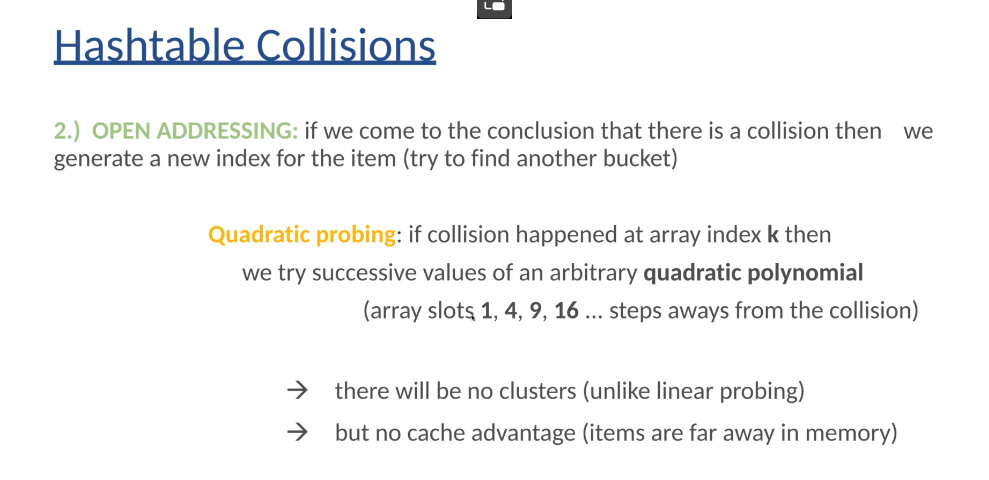
**Open Addressing**

* Metoda data se bazeaa pe aceea ca cautam un alt index daca un element este deja in acel index. Si metoda data mai are 3 metode:

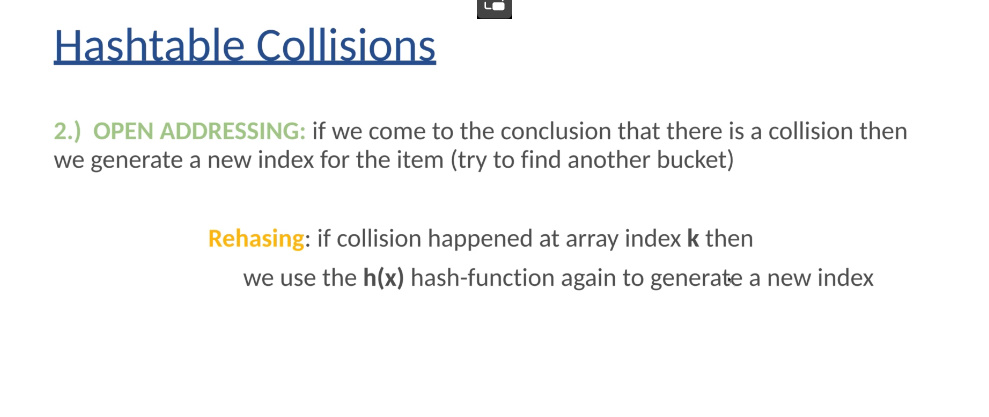
1. Linear probing

* 
* Deci, cand un index e ocupat, atunci incercam urmatorul index si apoi iar urmatorul daca nici acela nu e liber

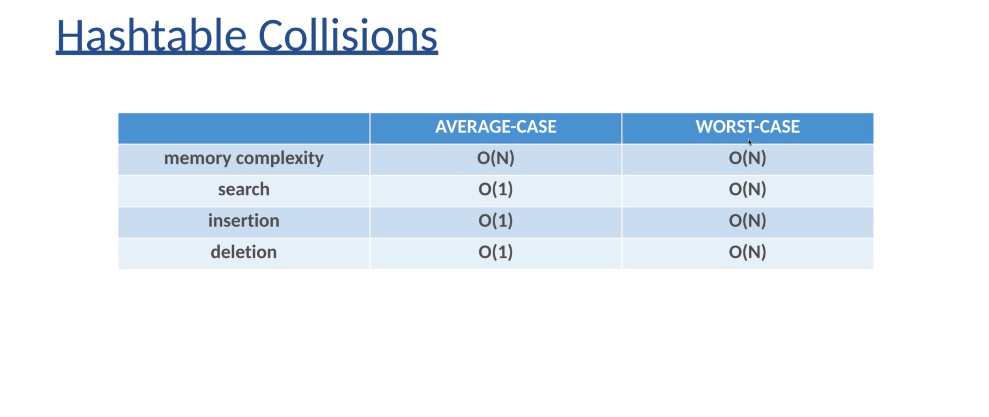
1. Quadratinc probing:



1. Rehasing

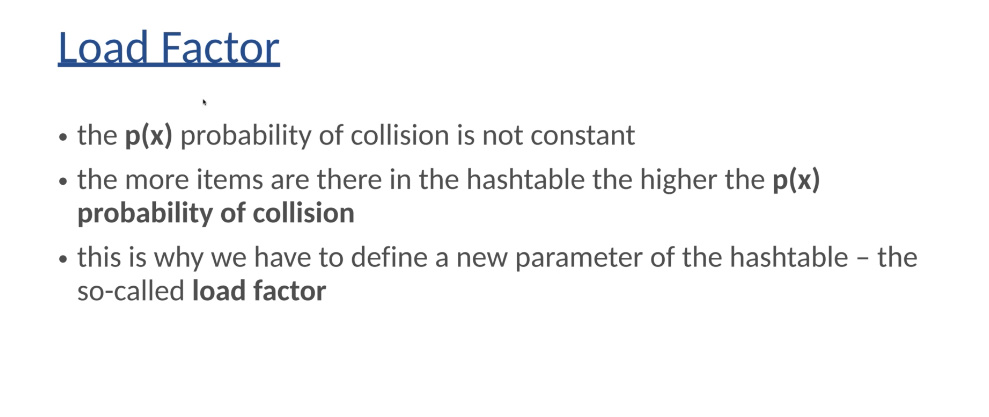
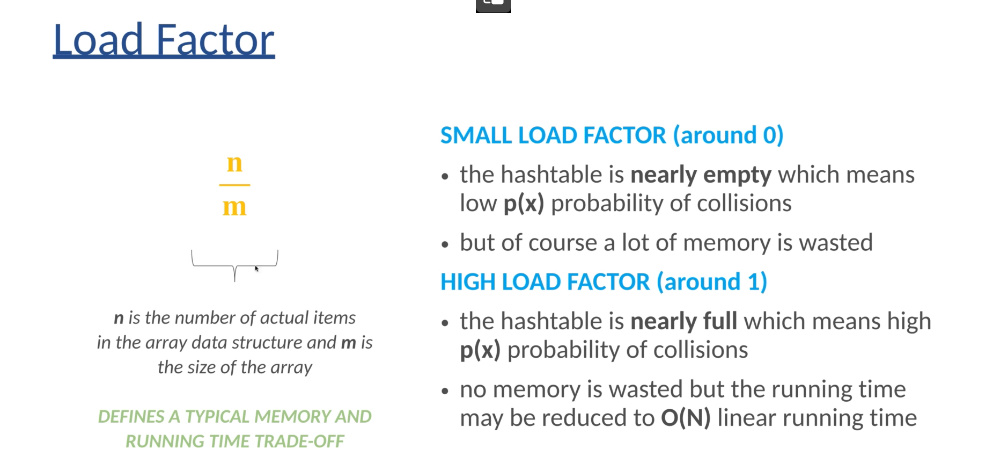
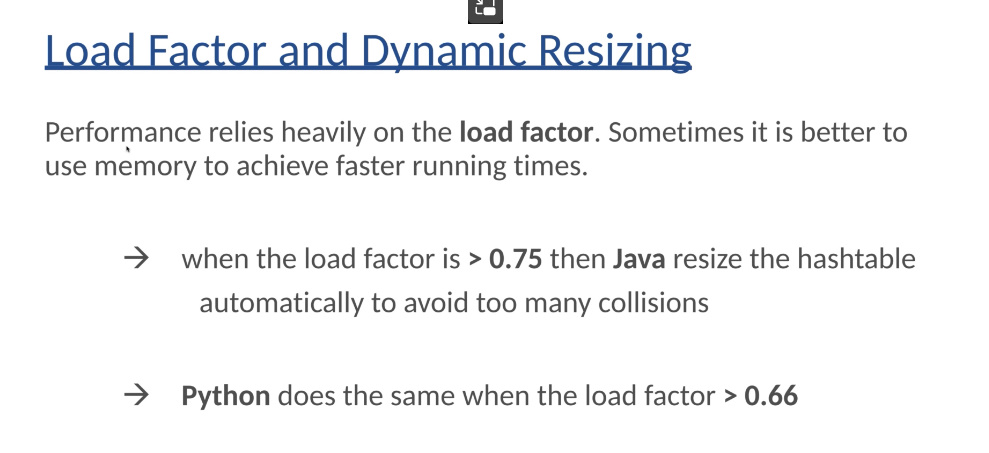
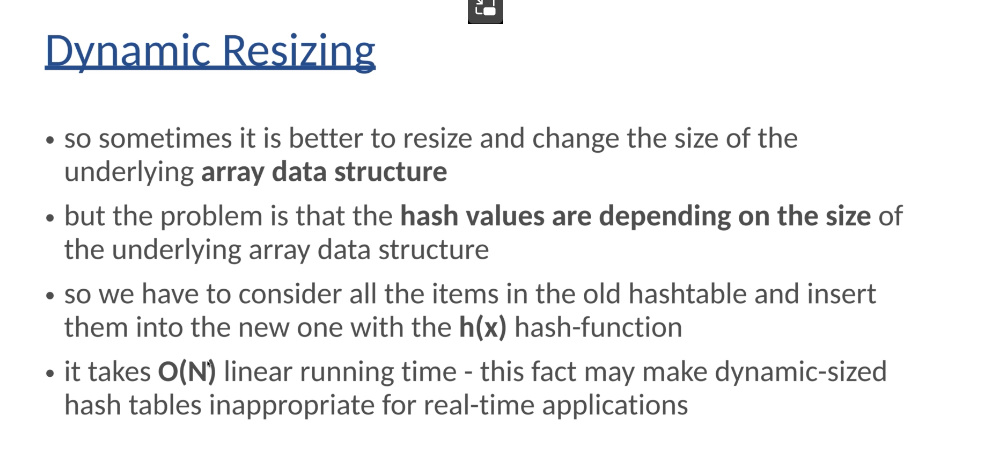


* Open Addressing e greu de implementat, de aceea nu prea e folosit



* Deci, HashTable pot ajunge sa fie la fel ca LinkedList si ArrayList la time complexity

**Load Factor and Dynamic Resizing**

* 
* 
* **Load Factor** – Number of items in array / size of array
* ****
* Deci, in Java load Factor este default 0.75.
* Daca raportul dintre itemele din array in care se pastreaza itemele si capacitatea la array ajunge la 0.75, Java creaza un nou array mai mare si copie in el toate elementele
* 
* Dar, aici apare o mare problema. Daca de ex avem un array de 8 elemente, toate itemele au obtinut indexul prin % 8, dar daca se va crea un nou array, pai itemele din vechiul array trebuie copiate in noul, dar nu se pun la indexii unde erau in vechiul array, dar deja trebuie pusi in noi indexi, conform deja noului array, si sa folosim % New Size
* deci, efectuarea acestor calcule si mutarea ia O(N), iata de ce in real app, HashTable nu prea sunt bune

**Hashtable in Java**

* Hashtable face parte si el din Collections, doar ca el a aparut pana la acest framework, dar a fost adaptat
* Este deja depreciat
* Foloseste key value items
* Impementeaza si el Map

***Map***

* Map interface este o mapping(reprezentare,schita) intre o valoare key si value
* Aceasta nu extinde Collection Interface
* **Metode:**
* clear()
* containsKey(Object)
* containsValue(Object) – verifica daca vreo key are o asa valoare
* entrySet() – creaza un set in care se contin keys si values
* equals(Map) – compara 2 maps sa vada daca au elementele comune
* get(key)
* hashCode() – creaza un hash code pentru map
* isEmpty()
* keySet() – returneaza un Set ce contine keys. Acest Set e strict legat de Map, deci orice modificare in Map se va vedea si in acest Set
* put(key,value) – adauga item. Daca o asa key deja exista, ea va fi inlocuita cu tot cu value
* putAll(Map) – copie toate elementele dintr-o mapa in alta
* remove(key)
* size()
* values() – creaza o colectie ce contine values
* putIfAbsent(key,value) – adauga perechea doar daca ea deja nu e prezenta
* replace(key,value) – modifica value pentru key
* replace(key,oldValue,newValue) – modifica value a lui key in newValue doar daca aceasta key are o valoare egala cu oldValue

**HashMap**

* Este practic identic cu HastTable la implementare
* Tot folosete hash function
* Nu e synchronized ca HashTable
* Nu extinde HashTable, ci doar Map
* Accepta null ca key
* Itemele nu sunt puse in ordinea in care le adaugam
* **Incepand cu Java 8, nu mai foloseste Chaining pentru a rezolva collisions, dar Red Black Tree acolo unde apar, si asa complexitatea garantata e mereu O(logN)**
* Orice clasa creata de noi va avea o metoda hasCode, care vine de la Object, si anume ea e facuta in asa fel ca sa calculeze HashCode folosind toate atributele din clasa
* HashMap mereu va folosi metoda hashCode() de la key object pentru a crea pozitia indexului
* Totusi, nu e prea bine de a suprascrie metoda hashCode
* Daca 2 obiecte sunt identice ca fields, ele trebuie sa aiba acelasi hashcode
* Totusi, 2 obiecte cu acelasi hashcode nu neaparat sunt egale ca fields
* Cand se aduna numarul de elemente egal cu Capacity\*Load Factor, HashMap dubleaza capacitatea sa
* In mod normal, adaugarea si accesarea itemelor e O(1), dar in medie e O(logN) deoarece incepand cu Java 8, coliziunile sunt rezolvate cu Red Black Tree
* **Key din HashMap trebuie sa implementeze Comparable, daca vrem sa fie aplicat Red Black Tree.**
* HashMap class contine o class Node cu urmatoarle fields:

int hash

K key

V value

Node next

Dar, deoarece din Java 8 deja foloseste Red Black Tree, mai are si ;

static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {  
 TreeNode<K,V> parent; // red-black tree links  
 TreeNode<K,V> left;  
 TreeNode<K,V> right;  
 TreeNode<K,V> prev; // needed to unlink next upon deletion  
 boolean red;

* **Constructori:**
* HashMap() – capacity = 16, loadfactory = 0.75f
* HashMap(int initialCapacity) – loadfactory = 0.75f
* HashMap(int initialCapacity, float loadFactor)
* **HashMap(Map map) – copie itemele din map**

**Entry and EntrySet**

* Interfata Map are si ea o Interfata in ea, numita Entry
* Map are si metoda entrySet()
* HashMap suprasctie metoda entrySet(). Ea returneaza un Set ce contine in el elementele la HashMap

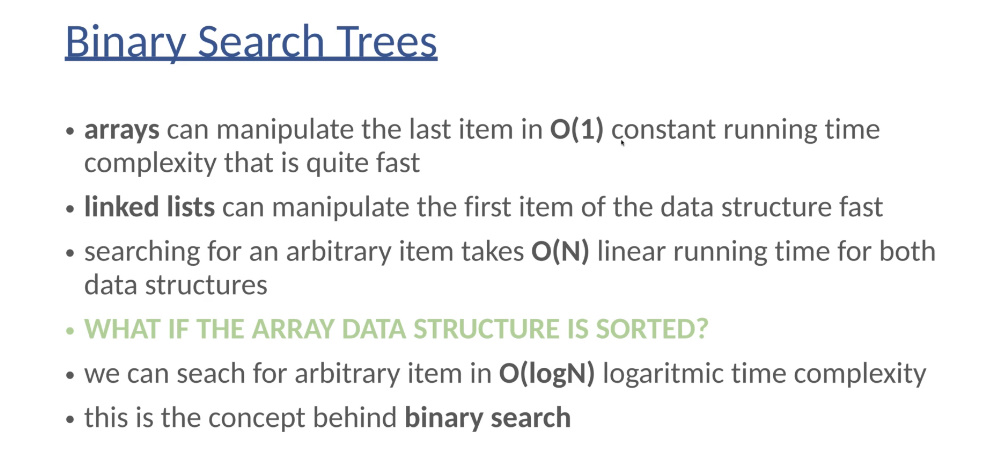
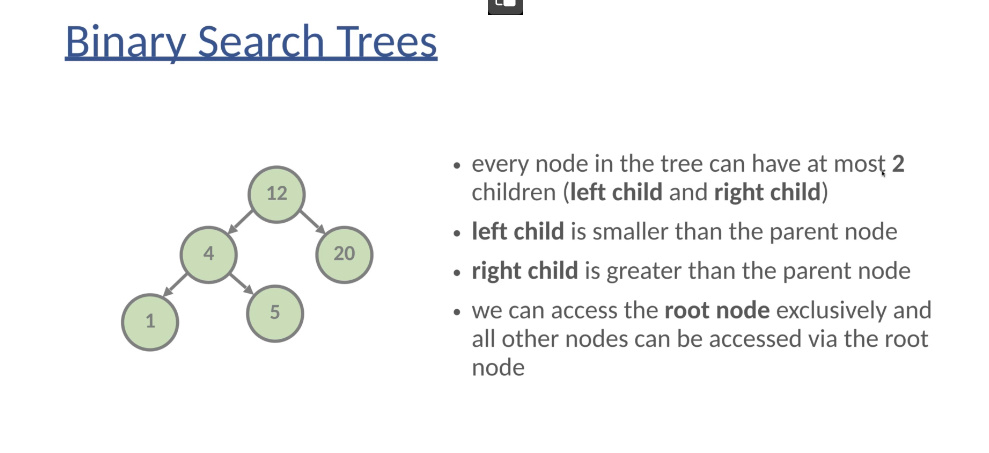
public Set<Map.Entry<K,V>> entrySet() {  
 Set<Map.Entry<K,V>> es;  
 return (es = entrySet) == null ? (entrySet = new EntrySet()) : es;  
}

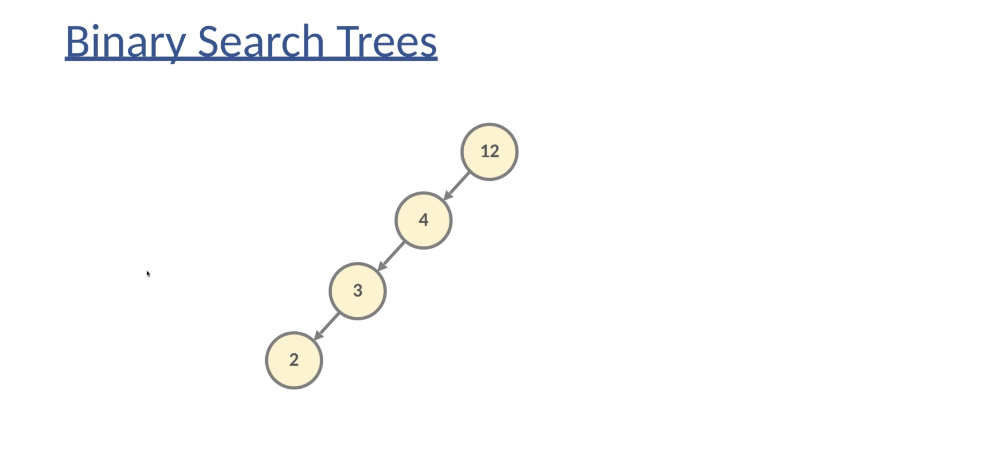
* HashMap are si un field entrySet

**LinkedHashMap**

* LinkedHashMap extinde HashMap
* HashMap, la adaugarea itemilor cu put(key,value), nu le adauga unul dupa altul, ci in dependenta de indexul determinat cu hash code.
* LinkedHashMap foloseste principiul la doubly linked list pentru a lega elementele adaugate intre ele in ordinea adaugarii lor.
* De asta, cand le vom accesa, desi ele nu sunt puse unul dupa altul in array, ele vor fi conectate prin pointeri
* In rest, LinkedHashMap e exact ca HashMap

**Tree Data Structures**

* 
* Deci, daca pastram itemele sortate, e mult mai rapida accesarea lor.
* 
* Totusi, BST are si el o problema, si anume ca s-ar putea sa aiba time complexity O(N) in situatii de genul:

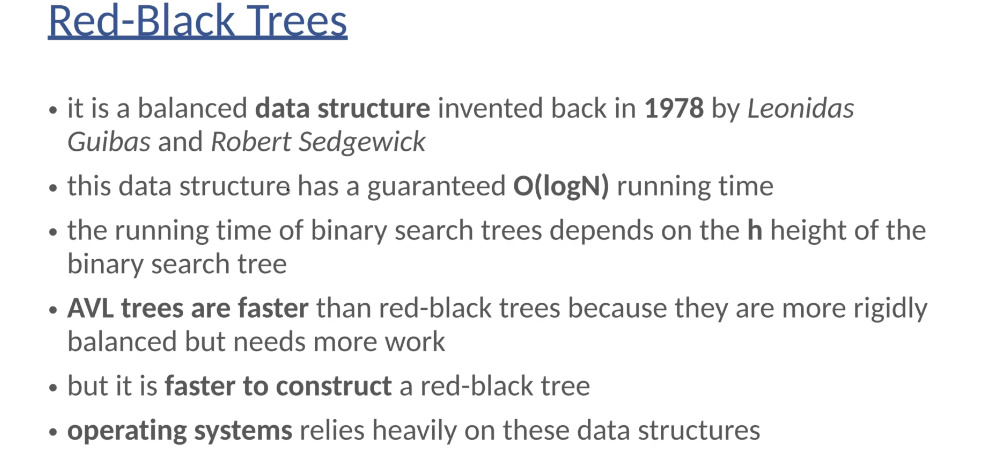
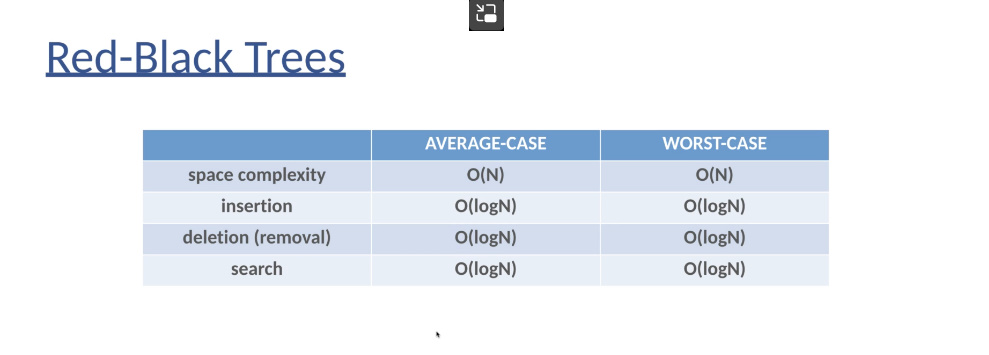


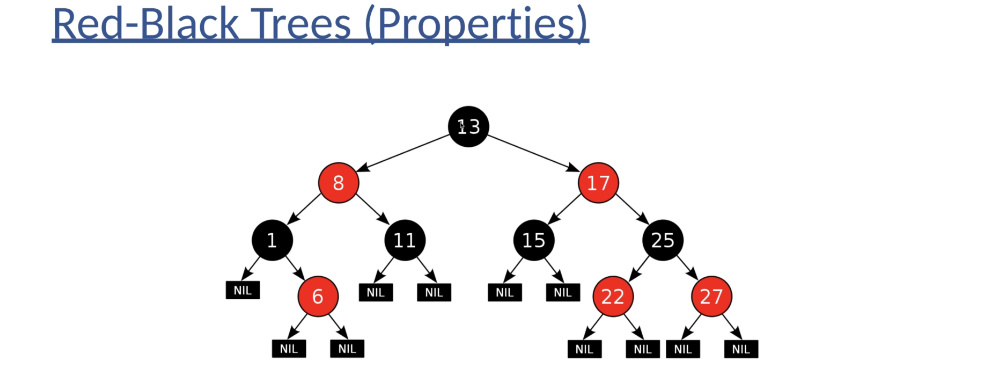
* Aici BST nu este balansat, dar exista solutii.
* Pentru a rezolva problema, se foloseste Balanced Binary Search Tree.
* Exista 2 tipuri de Balanced Binary Search Tree:

1. AVL
2. Red-Black Tree

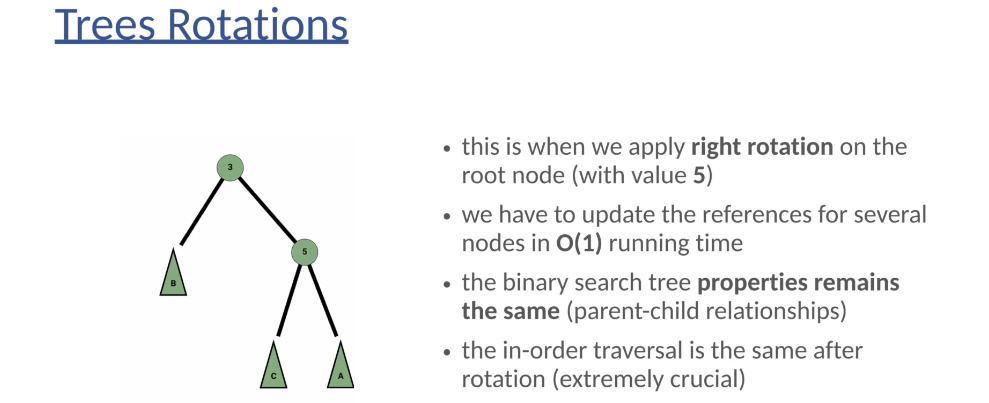
* Java foloseste Read-Black Tree in TreeMap si TreeSet

**Red-Black Tree**

* 
* 
* Fiecare Node in el are o culoare: Red sau Black



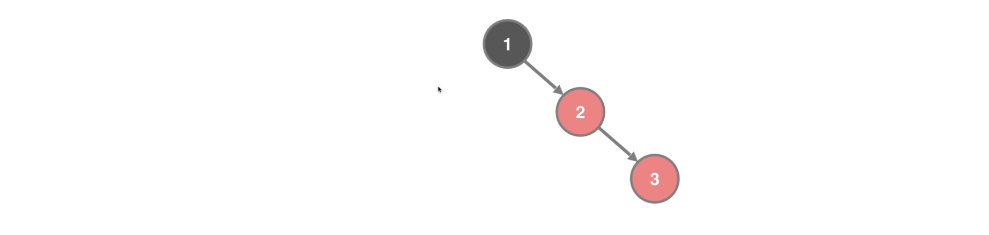
Null tot este Black

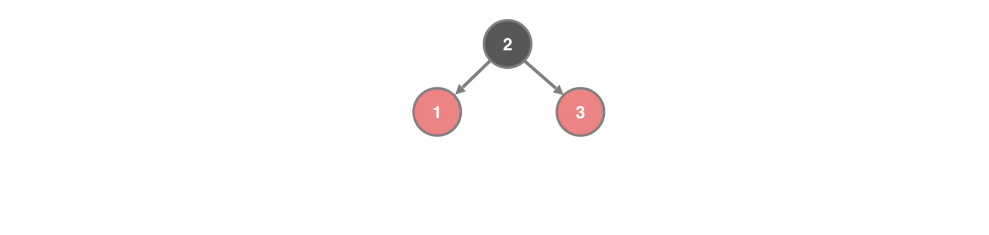
* O proprietate importanta este ca de la Root pana la orice Null(frunza) mereu e acelasi numar de noduri negre
* Pentru a asigura ca arborele e balansat, se fac diferite rotatii, asa incat root sa aiba un numar aproximativ egal de left childs si right childs
* 

Exemplu:

Insert(1)

Insert(2)

Insert(3)  
  
Se face rotatie si recoloring:

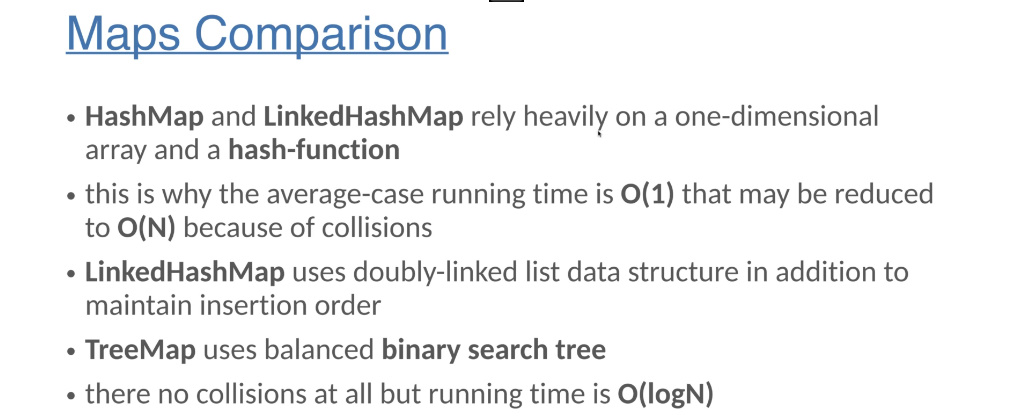


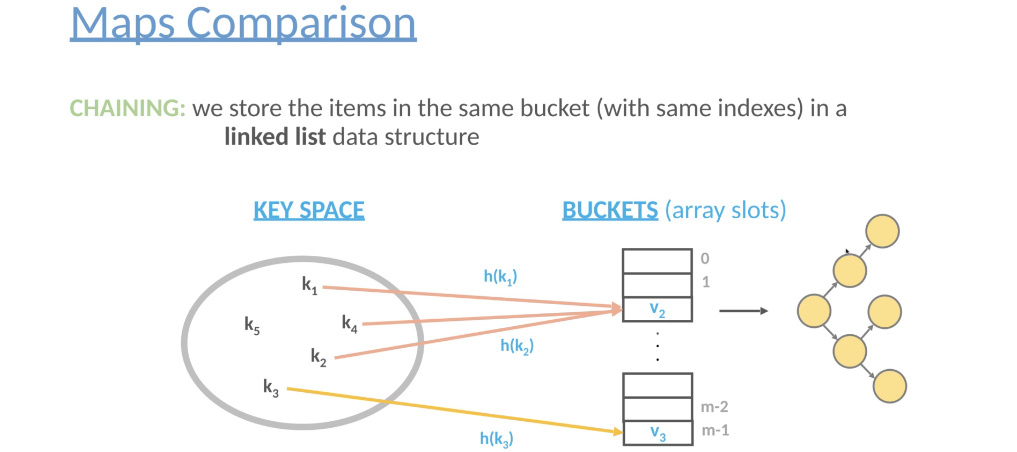
**Tree Map**

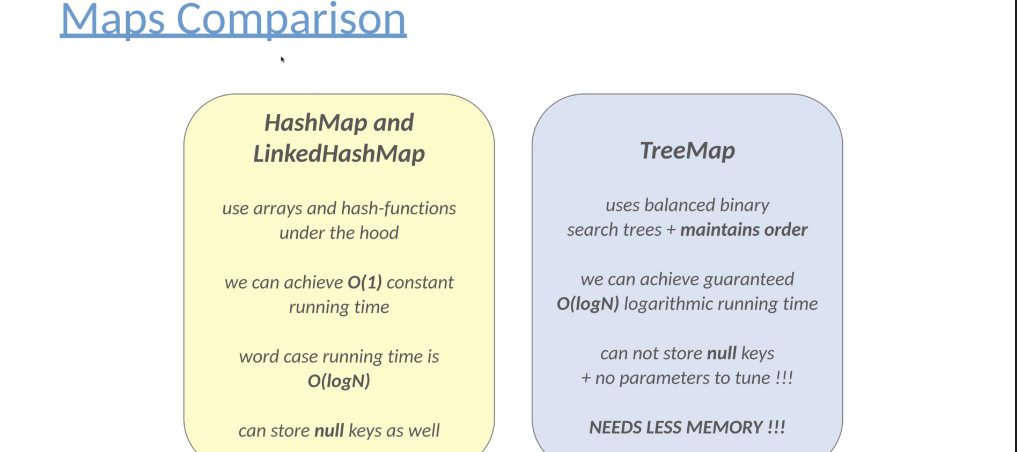
* TreeMap extinde clasa SortedMap si implementeaza interfata Map
* Pastreaza itemele in ordine crescatoare
* Cautarea, adaugarea, stergerea mereu e O(logN)
* Ordinea este definita fie de Interfata Comparable, pe care orice key item trebuie sa o implementeze, sau daca nu, atunci de un obiect care implementeaza interfata Comparator si care noi il specificam
* Nu accepta keya null
* Totusi, in unele cazuri, TreeMap ar putea fi lent, asa cum fiecare item inserat va face ca arborele sa se balanseze
* **In TreeMap nu se foloseste hashFunction!!!**
* **Constructori**:
* TreeMap()
* TreeMap(Comparator)
* TreeMap(Map)
* TreeMap(SortedMap)
* **Specific Methods:**
* firstKey() – returneaza cel mai mic element din TreeMap
* headMap(Object key) – returneaza un NavigableMap cu elementele care au key strict mai mic
* lastKey() – returneaza cel mai mare element
* subMap(key1,key2) – returneaza un NavigableMap ce contine elementele de la key1 pana la key2

....... Mai sunt si altele

**Maps Comparison**

* 
* Din Java 8, HashMap nu mai foloseste Chaining in caz de coliziuni, dar foloseste Red Black Tree pentru elementele din acelasi index din array:



* 
* HashMap este totusi mai rapida ca TreeMap, dar daca avem nevoie sa sortam mereu itemele, TreeMap e unicul ce ne poate ajuta