ALGORITHMS AND DATA STRUCTURES LECTURE 5 – HASH TABLE AND BST

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

ASTANA IT UNIVERSITY

CONTENT

1. Hashing
2. Hash Table
3. Binary Search Tree
4. BST: Inorder Traversal

HASHING

Hashing means using some function or algorithm to map object data to some representative integer value

This so-called hash code (or simply hash) can then be used as a way to narrow down our search when looking for the item in the set

Object class contains hashCode() method with its default implementation

Recommended: Each class provides its own implementation of hashCode()

HASHING: SIMPLE EXAMPLE

Sum of ASCII values of each char **mod** no. of available slots

Mia	M	77	i	105	a	97	279 mod 11 =	4
Tim	T	84	i	105	m	109	298 mod 11 =	1
							the same for others..	
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10

Bea	Tim	Leo	Sam	Mia	Zoe	Jan	Lou	Max	Ada	Ted
0	1	2	3	4	5	6	7	8	9	10

HASH TABLE: LINEAR PROBING

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array(picture on previous slide) by looking into the next cell until it finds an empty cell. This technique is called **linear probing**.

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Sue	S	83	u	117	e	101	301	4

Bea	Tim			Mia	Zoe	Sue				
0	1	2	3	4	5	6	7	8	9	10

HASHING: STRING EXAMPLE

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i^{th} character of s

```
String s = "call";
int code = s.hashCode();
```

$$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$$
$$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$$

(Horner's method)

Horner's method to hash string of length L: L multiplies/adds.

Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$

HASHING:‘STANDARD’ RECIPE

Combine each **significant** field using the 31x + y rule.

If field is a primitive type, use wrapper type hashCode()

If field is null, return 0

If field is a reference type, use hashCode()

If field is an array, apply to each entry

```
public final class Transaction implements Comparable<Transaction>
{
   private final String  who;
   private final Date    when;
   private final double  amount;

   ...

   public int hashCode()                              nonzero constant
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}                              typically a small prime
```

HASH TABLE(CHAINING: CLOSED ADDRESSING)

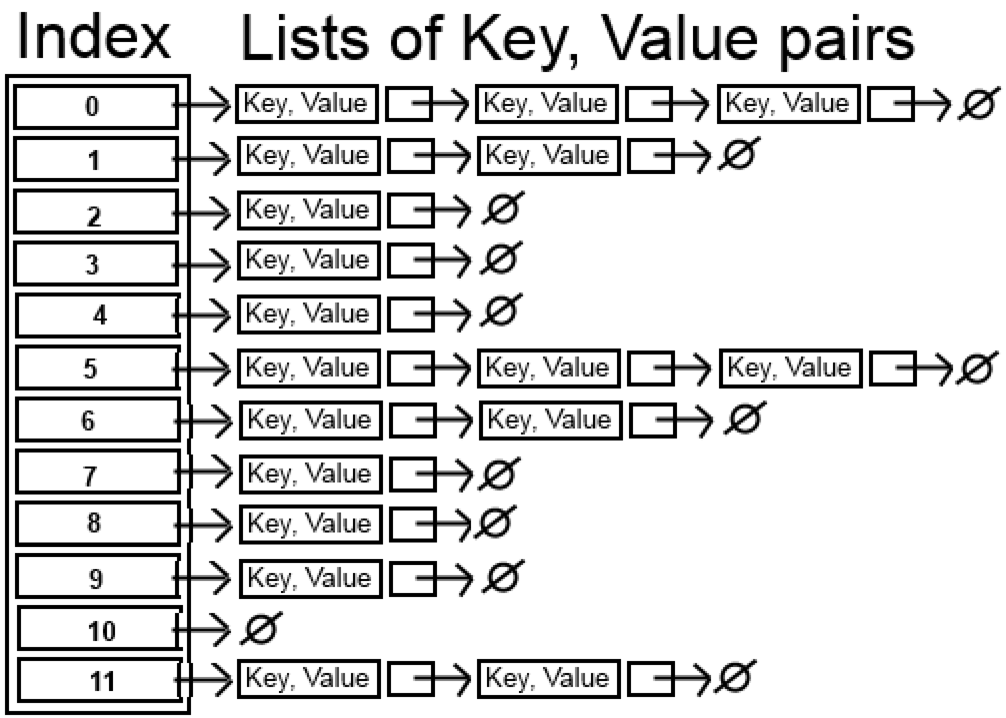
Hash table maps keys to values. Any non-null object can be used as a key or as a value

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

It looks like “an array of singly-linked lists (**chains**)”

Each linked list is accepted as **bucket**

Array size indicates number of buckets



Average case:

- Insertion = deletion = retrieving = searching = **O(1)**

HASH TABLE

The **capacity** (number of buckets - **M**) and **load factor** are parameters that affect to its performance

The **load factor** is a measure of how full the hash table is allowed to get before its capacity is automatically increased (LF should be around 0.75)

Load factor = (no. of elements) / (no. of table slots)

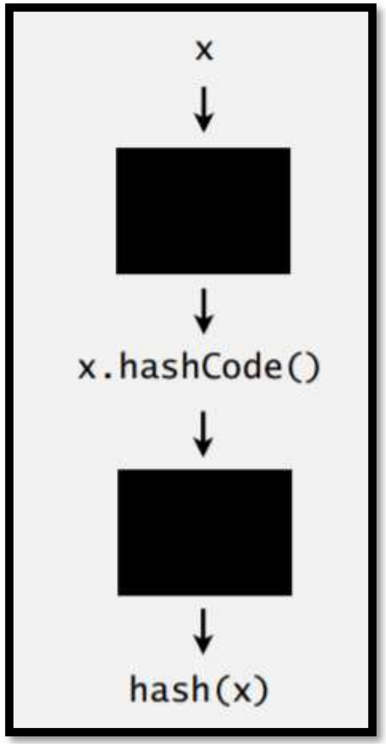
The hashCode is used to get an index of *chain* by **hash()** method (**Modular hashing**)

```
private int hash(Key key)
{   return Math.abs(key.hashCode()) % M;   }
```

1-in-a-billion bug

```
private int hash(Key key)
{   return (key.hashCode() & 0x7fffffff) % M;   }
```

correct



HASH TABLE

Collision – having same index for several nodes (cannot be avoided)

- A new node should be added to the same chain (bucket)

Challenge: Deal with collisions efficiently

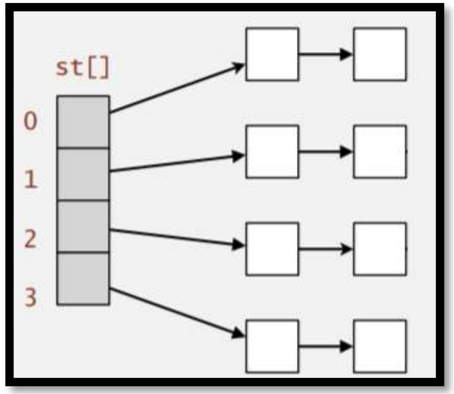
Target: Uniform distribution

Analysis: Number of probes for search/insert is proportional to N/M

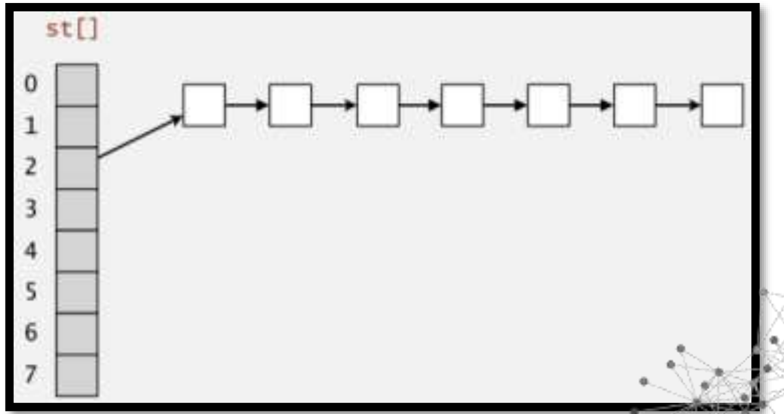
- M too large ⇒ too many empty chains
- M too small ⇒ chains too long
- Typical choice: M ~ N / 4 ⇒ constant-time ops

Once a hash table has passed its load factor - it has to rehash [create a new bigger table, and re-insert each element to the table]

Best case



Worst case



HASH TABLE: EXAMPLE

```java
public class MyHashTable<K, V> {

    private class HashNode<K, V> {...}

    private HashNode<K, V>[] chainArray; // or Object[]
    private int M = 11; // default number of chains
    private int size;

    public MyHashTable() {...}

    public MyHashTable(int M) {...}

    private int hash(K key) {...}

    public void put(K key, V value) {...}

    public V get(K key) {...}

    public V remove(K key) {...}

    public boolean contains(V value) {...}

    public K getKey(V value) {...}
}
```

```java
private class HashNode<K, V> {
    private K key;
    private V value;
    private HashNode<K, V> next;

    public HashNode(K key, V value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public String toString() {
        return "{" + key + " " + value + "}";
    }
}
```

OBJECTIVES OF HASH FUNCTION

- Minimize collisions
- Uniform distribution of hash values
- Easy to calculate
- Resolve any collisions

BINARY SEARCH TREE

A BST is a binary tree in symmetric order.

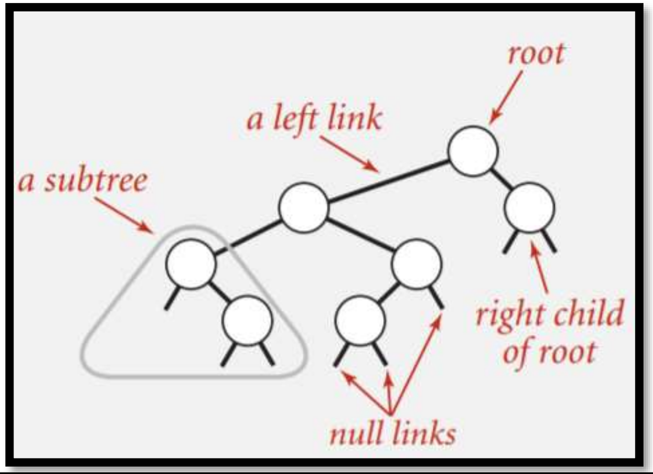
Each node has two references to left and right nodes

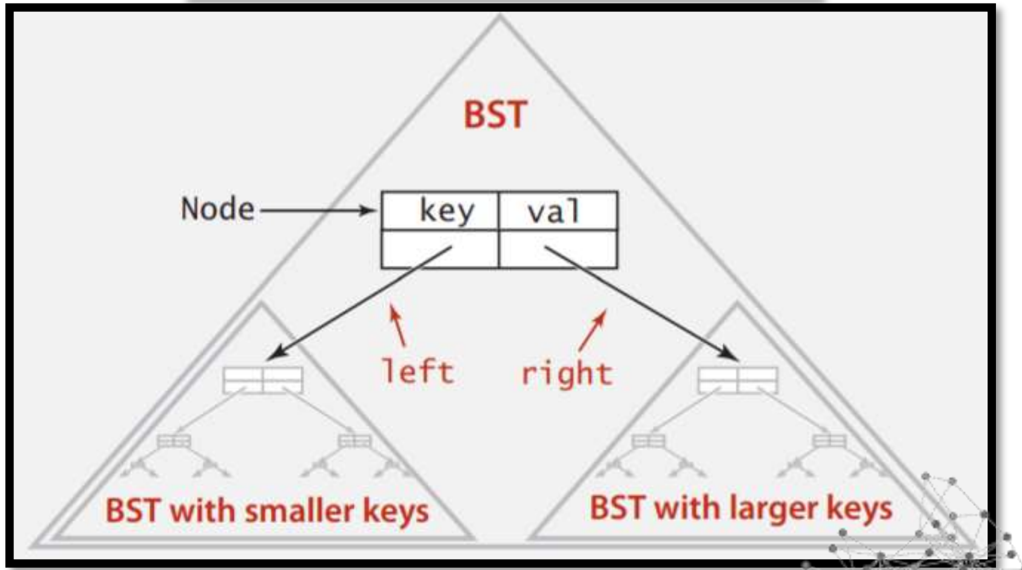
Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree
- Smaller than all keys in its right subtree

A Node is composed of four fields

- Key and Value
- Left and right subtree references





BINARY SEARCH TREE

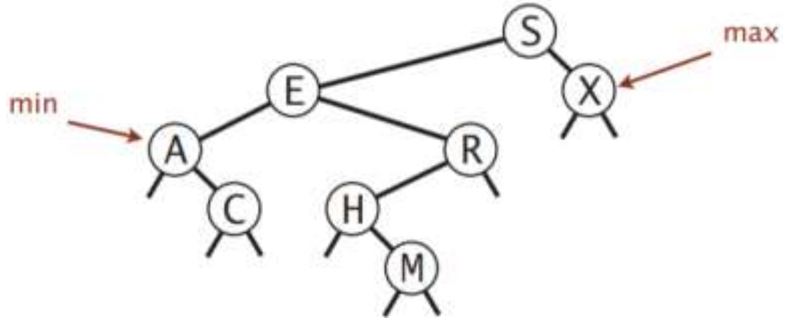
A BST uses O(log(N)) for most manipulations

Search: If less, go left; if greater, go right; if equal, search hit

Insert: If less, go left; if greater, go right; if null, insert

GetMin(): Most left node

GetMax(): Most right node




```java
public class BST<K extends Comparable<K>, V> {
    private Node root;
    private class Node
    {
        private K key;
        private V val;
        private Node left, right;
        public Node(K key, V val)
        {
            this.key = key;
            this.val = val;
        }
    }
    public void put(K key, V val) {...}

    public V get(K key) {...}

    public void delete(K key) {...}

    public Iterable<K> iterator() {...}
}
```

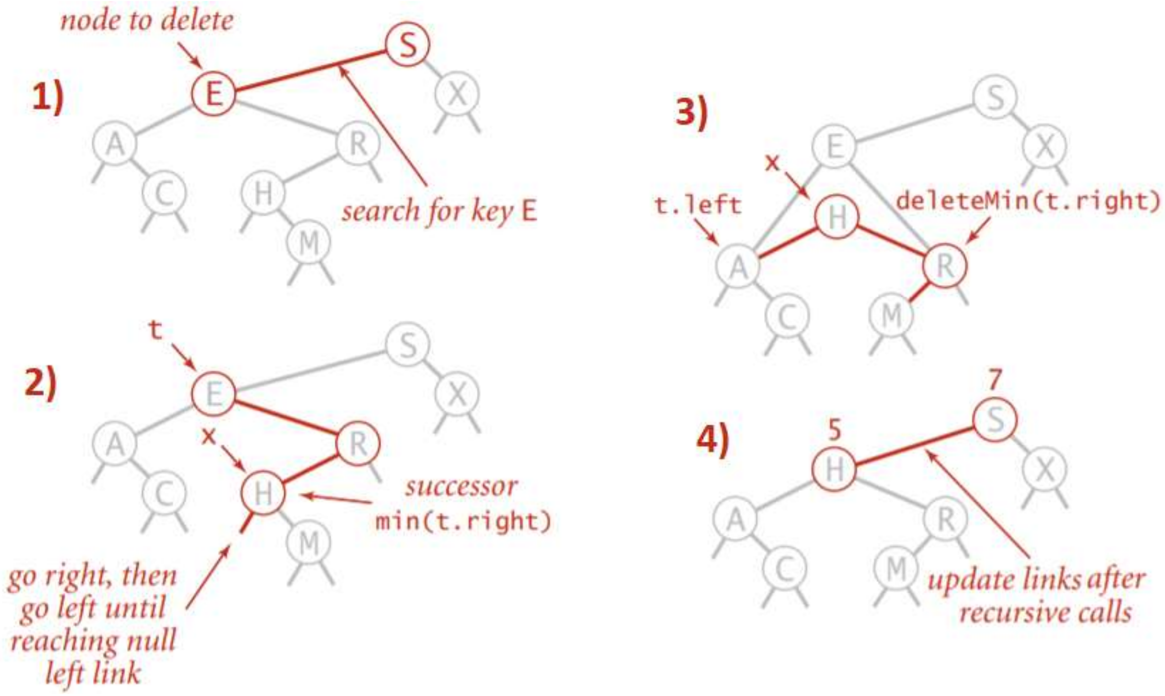
BINARY SEARCH TREE: DELETE

To delete a node with key k: search for node t containing key k

Case 1 (1 child): Delete t by replacing parent link

Case 2 (2 children):

- Find successor x of t
- Delete the minimum in t's right subtree
- Put x in t's spot



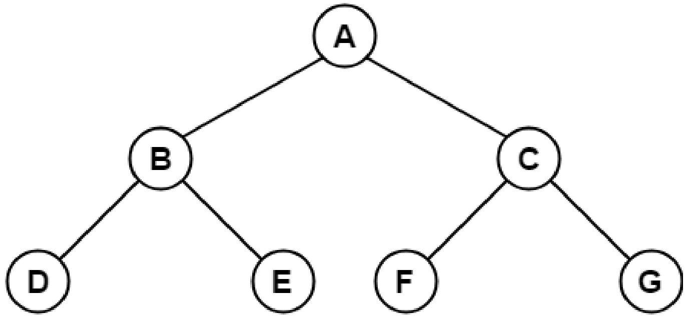
guarantee			average case			ordered ops?	operations on keys
search	insert	delete	search hit	insert	delete		
N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	compareTo()

BST: INORDER TRAVERSAL

In BST, **inorder traversal** is used to get nodes in increasing order (Left-Root-Right)

Ordered iteration

- Traverse left subtree
- Enqueue key
- Traverse right subtree



Inorder Traversal : D , B , E , A , F , C , G

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
   if (x == null) return;
   inorder(x.left, q);
   q.enqueue(x.key);
   inorder(x.right, q);
}
```

LITERATURE

Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapters 3.2, 3.4

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapter 5

GOOD LUCK!