

# Searching Lists

## Algorithms and Datastructures

Marjahan Begum and Anders Kalhauge



Spring 2017

## Symbol Tables continued

- Hashed

- Search Trees

- Balanced Search Trees

## Symbol Tables continued

- Hashed

- Search Trees

- Balanced Search Trees

## Problem

We want to access a symbol table with  $n$  keys, using a key  $k \in K$  as was it an index to an array with  $n$  elements.

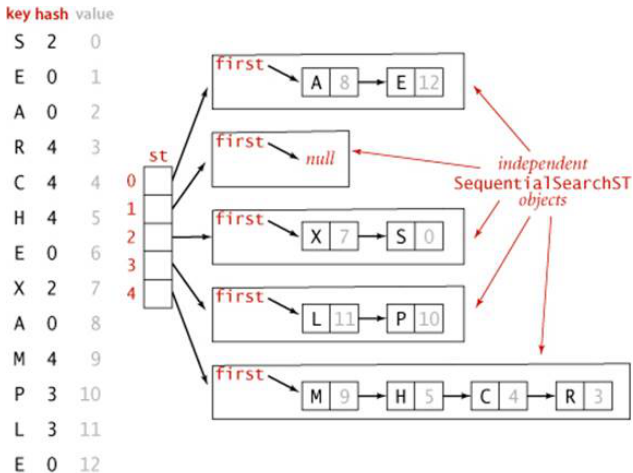
## Perfect solution

By **magic** we find a mapping from the keys  $K$  to  $\{0, \dots, n-1\}$ . We call it the hash function:  $h : K \rightarrow \{0, \dots, n-1\}$ .

## Realistic Solution

Find a hash function that maps the keys  $K$  uniformly over  $\{0, \dots, m-1\}$  where  $m \approx n$ .

## Chained hashing



Hashing with separate chaining for standard indexing client

## Linear probing

key	hash	value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0							S									
E	10	1							S				E					
A	4	2					A		S				E					
R	14	3				A	2		S				E				R	
C	5	4				A	2	C	4				E				R	
H	4	5				A	2	C	4	S	H		E				R	
E	10	6				A	2	C	4	S	H		E				R	
X	15	7				A	2	C	4	S	H		E				R	X
A	4	8				A	2	C	4	S	H		E				R	X
M	1	9		M		A	2	C	4	S	H		E				R	X
P	14	10	P	M		A	2	C	4	S	H		E				R	X
L	6	11	P	M		A	2	C	4	S	H	L					R	X
E	10	12	P	M		A	2	C	4	S	H	L	E				R	X

entries in red are new

entries in gray are untouched

keys in black are probes

probe sequence wraps to 0

keys[]

vals[]

Trace of linear-probing ST implementation for standard indexing client

With:

```
@FunctionalInterface
public interface HashFunction {
    int function(String key);
}
```

You can:

```
public static test(String[] keys, HashFunction hash) {
    for (String key : keys)
        System.out.println(hash.function(key));
}
public static void main(String... args) {
    String[] words =
        FileUtility.toStringArray("sp.txt", "[^A-Za-z]");
    test(words, k -> k.length());
}
```

Experiment with various hash implementation of hash functions. All functions should return a number between 0 and 31.

The signature of the hash function should be:

```
@FunctionalInterface
public interface HashFunction {
    int function(String key);
}
```

Create hash functions that uses:

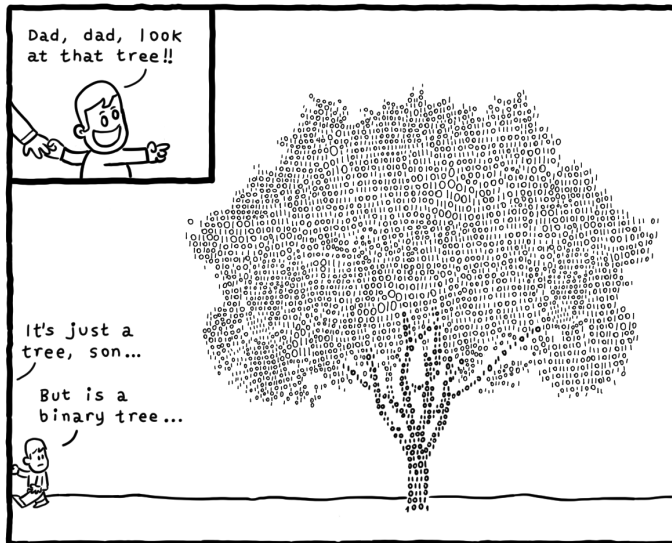
- ☐ the first character code
- ☐ the last character code
- ☐ the sum of character codes
- ☐ `Strings hashCode()` method

You can use  $n\%32$  to fit  $n$  into the range of 0...31

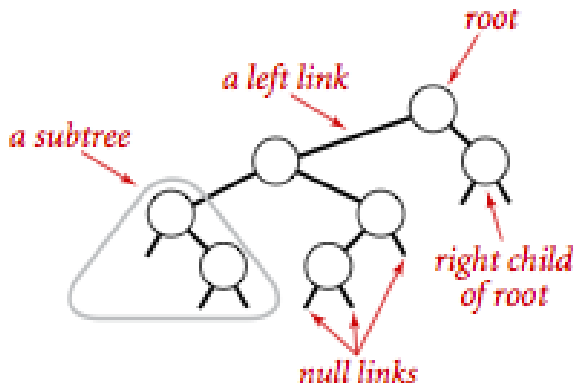
**Make a histogram of each hash function**



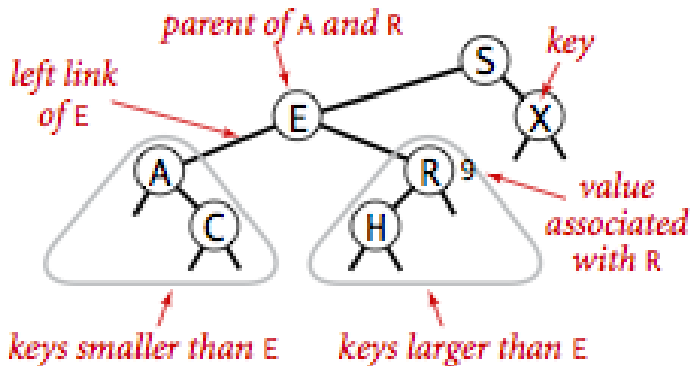
- Universal hashing
- Static tables
- Perfect hashing



Daniel Stori {turnoff.us}



**Anatomy of a binary tree**

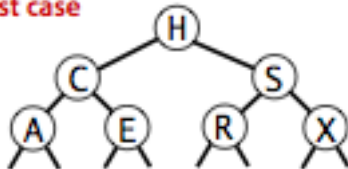


## Anatomy of a binary search tree

Implement the structure for a binary search tree

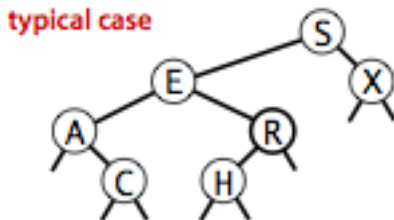
## Analysis

best case



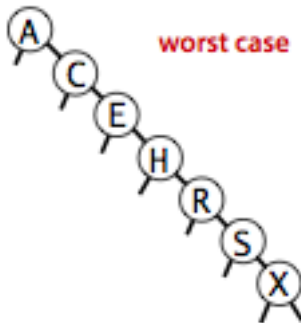
$$O(\log n)$$

## Analysis



$$O(\log n)$$

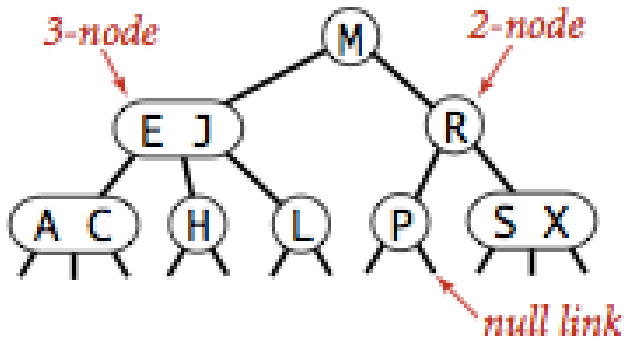
## Analysis



$$O(n)$$



### Anatomy



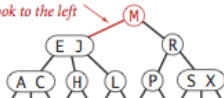
### Anatomy of a 2-3 search tree

Implement the structure for a binary search tree

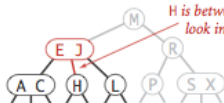
## Searching

### successful search for H

H is less than M so  
look to the left



H is between E and J so  
look in the middle

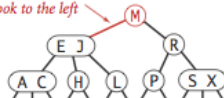


found H so return value (search hit)

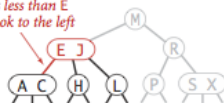


### unsuccessful search for B

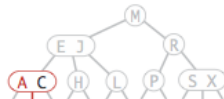
B is less than M so  
look to the left



B is less than E  
so look to the left

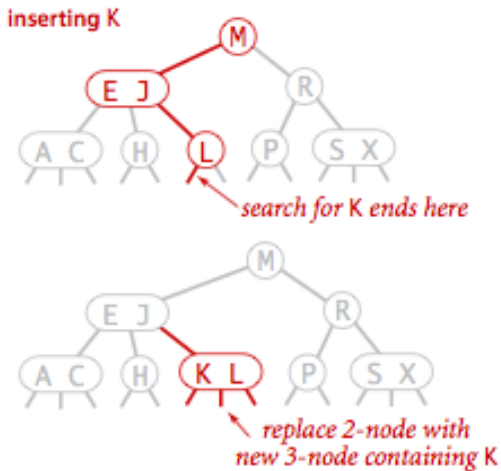


B is between A and C so look in the middle  
link is null so B is not in the tree (search miss)



Search hit (left) and search miss (right) in a 2-3 tree

### Inserting



**Insert into a 2-node**

### Inserting

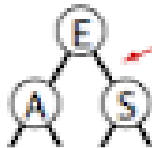
Inserting S



← no room for S



← make a 4-node

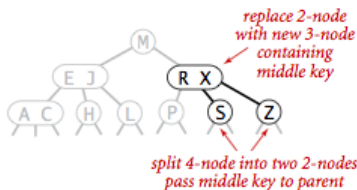
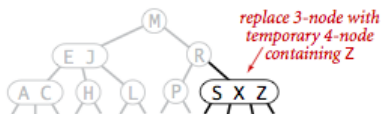
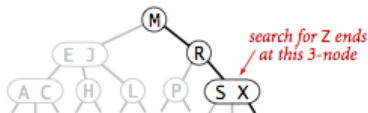


split 4-node into  
this 2-3 tree

Insert into a single 3-node

### Inserting

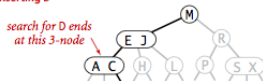
inserting Z



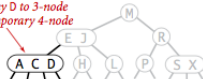
Insert into a 3-node whose parent is a 2-node

## Inserting

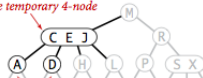
inserting D



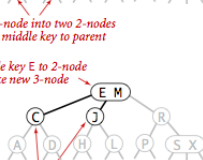
add new key D to 3-node to make temporary 4-node



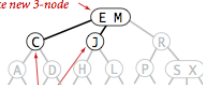
add middle key C to 3-node to make temporary 4-node



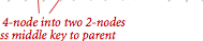
split 4-node into two 2-nodes pass middle key to parent



add middle key E to 2-node to make new 3-node



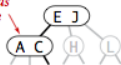
split 4-node into two 2-nodes pass middle key to parent



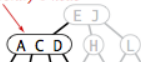
Insert into a 3-node whose parent is a 3-node

inserting D

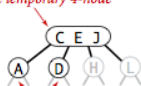
search for D ends at this 3-node



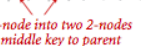
add new key D to 3-node to make temporary 4-node



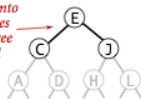
add middle key C to 3-node to make temporary 4-node



split 4-node into two 2-nodes pass middle key to parent



split 4-node into three 2-nodes increasing tree height by 1



Splitting the root

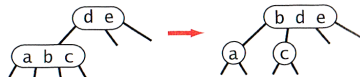
## Summing up

root



parent is a 3-node

left



parent is a 2-node

left



middle



right



right



Splitting a temporary 4-node in a 2-3 tree (summary)