COMP 3760: Algorithm Analysis and Design

Lesson 9: Maps, Sets, Hashing Functions

Rob Neilson

rneilson@bcit.ca

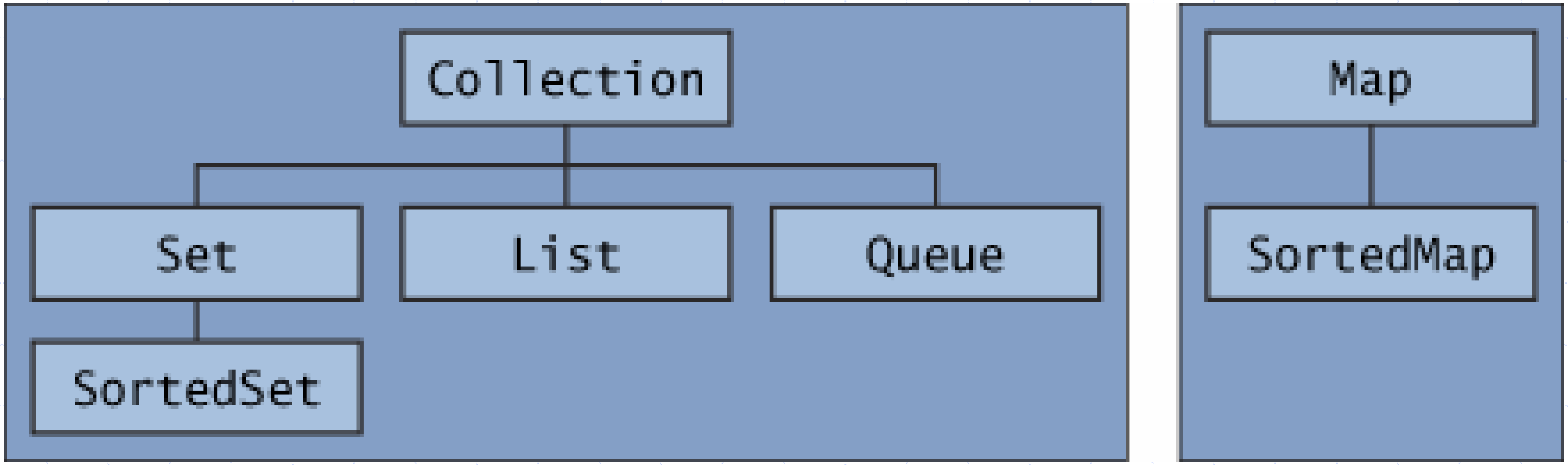
Homework (due 8:30 Monday next week)

- Reading for next week:
  - read chapters 6.4
- Exercises:
  - there is one question that you must sketch a solution for
  - you are expected to explain exactly how to solve the problem, providing pseudocode as appropriate
  - the actual question/problem is available as Homework 5 in webct
- Important:
  - for this week only all students in all sets are required to submit their solution to webct, *PRIOR TO 8:30AM MONDAY OCT 6*
  - late submissions will not be accepted; no exceptions

Java Collections Framework

- set of classes and interfaces that implement commonly used data structures and algorithms
- save programmers a lot of work as you don't have to re-invent the wheel
- very useful for solving most common programming problems

Interfaces



Java Collections Framework (2)

Java provides the following general-purpose implementations of the interfaces in the Java Collections Framework.

Note: there are many special purpose implementations … these are just the most commonly used ones that you ought to be aware of …

Set:	HashSet	TreeSet	LinkedHashSet
Map:	HashMap	TreeMap	LinkedHashMap
List:	LinkedList	ArrayList	
Queue:	PriorityQueue	LinkedBlockingQueue	

There are other useful implementations of sub-interfaces, such as:

	ArrayDeque	Stack

Java Collections Framework (3)

Algorithms

sort(List)

- Sorts a list using a merge sort algorithm with guaranteed O(n*log n) performance.

binarySearch(List, Object)

- Searches for an element in an ordered list.

reverse(List)

- Reverses the order of the elements in the a list.

shuffle(List)

- Randomly permutes the elements in a list.

fill(List, Object)

- Overwrites every element in a list with the specified value.

copy(List dest, List src)

- Copies the source list into the destination list.

min(Collection)

- Returns the minimum element in a collection.

max(Collection)

- Returns the maximum element in a collection.

Java Collections Framework (4)

replaceAll(List list, Object oldVal, Object newVal)

- Replaces all occurrences of one specified value with another.

swap(List, int, int)

- Swaps the elements at the specified positions in the specified list.

frequency(Collection, Object)

- Counts the number of times the specified element occurs in the collection.

disjoint(Collection, Collection)

- Determines whether two collections are disjoint, in other words, whether they contain no elements in common.

addAll(Collection<? super T>, T...)

- Adds all of the elements in the specified array to the specified collection.

newSetFromMap(Map)

- Creates a general purpose Set implementation from a general purpose Map implementation.

Maps and Sets

- Sets are ***unordered collections of objects***
  - They are just containers that you can use to throw a bunch of related objects into
- Maps are like “dictionaries” or “associative arrays”. They map keys to specific values.
- Both maps and sets can be efficiently implemented using hashing
  - maps and sets can also be implemented using structures other than hash tables
- Why would we want to use hashing?

Sets: HashSet

HashSet is the fastest implementation, but it is unordered

- because it is a *set*, it has the properties of a set, such as
  - no implied order
  - no duplicate entries
- the drawback of a set is that it is not sorted; even worse, there is *no deterministic order* for its elements
- since it is implemented using a hash table …
  - … we get ***fast*** insert, delete, find (all are ***O(1)***)
- What about iteration? How fast can we iterate over a HashSet?
  - we don't know where the elements are stored (which buckets), so we have to check every bucket
  - this means the efficiency is proportional not only to the number of stored elements (n), but also to the length of the hashtable (h)
  - it turns out that iteration is ***O(n+h)***

Sets: TreeSet

TreeSet is slower, but maintains a sorted order

- TreeSet is also a set, but it is maintained in sorted order
  - note that objects must implement comparable
- the set is implemented using a red-black tree
  - a type of balanced binary tree; see section 6.3 of the text
- since the set is maintained in sorted order, there is a performance decrease for all operations
  - most operations are ***O(logn)*** ... or ... the height of the backing red-black tree

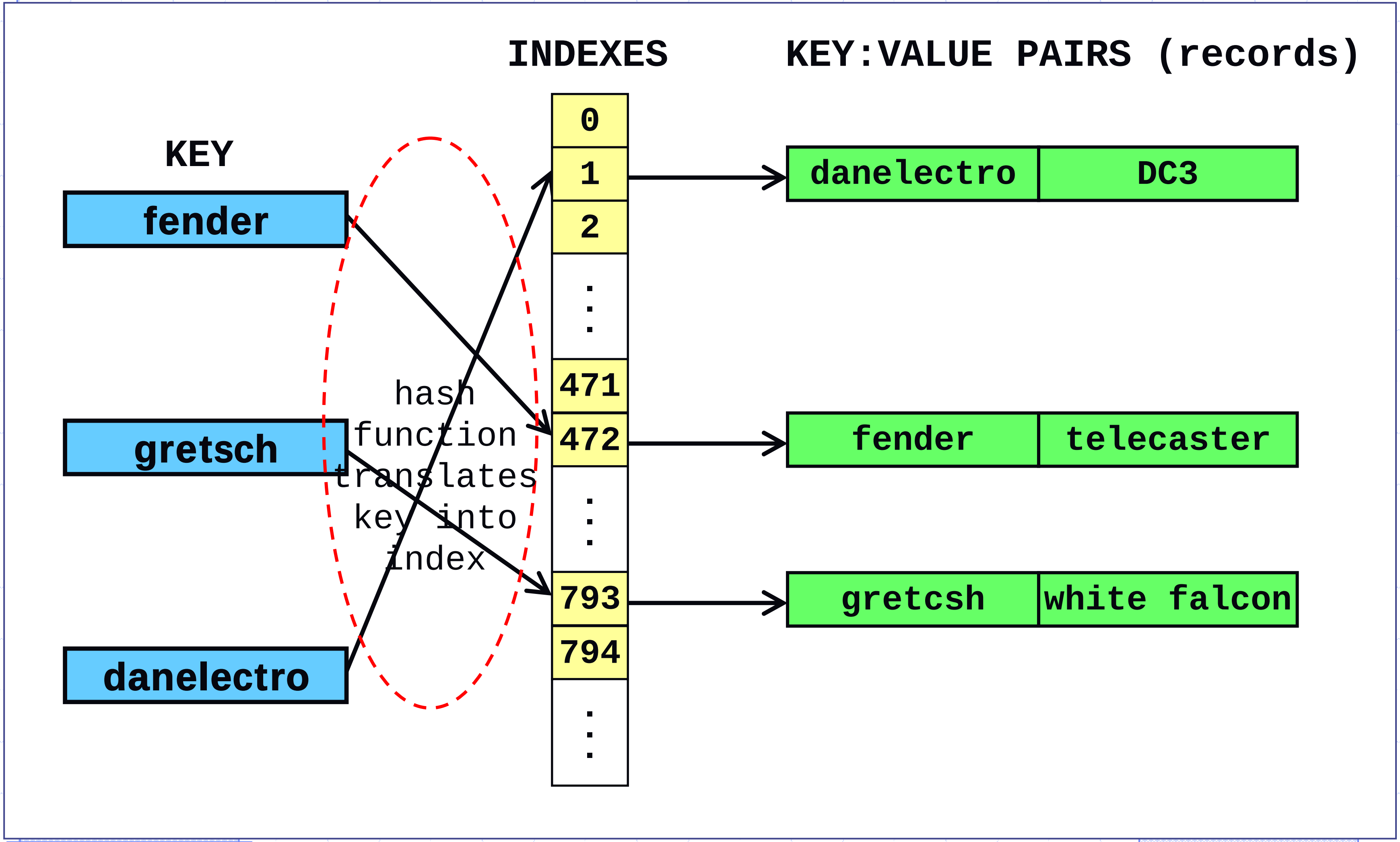
Sets: LinkedHashSet

- *LinkedHashSet* maintains the objects in "insertion order"
  - it is a HashSet with a double linked list running through it
    - new elements are added to the end of the list when they are inserted into the set
    - the hash is used for Add, Contains, Remove, but the list is used for iteration
    - this guarantees an order in the iteration, and makes iteration faster
- *Summary:*
  - *almost the same performance as HashSet, but faster for traversal*

Operation	Method	HashSet	TreeSet	LinkedHashSet
Insert	**Set.add(Object)**	**O(1)**	**O(logn)**	**O(1)**
Traverse	**Iterator**	**O(n+h)**	**O(n)**	**O(n)**
Find	**Set.contains(Object)**	**O(1)**	**O(logn)**	**O(1)**
Delete	**Set.remove(Object)**	**O(1)**	**O(logn)**	**O(1)**

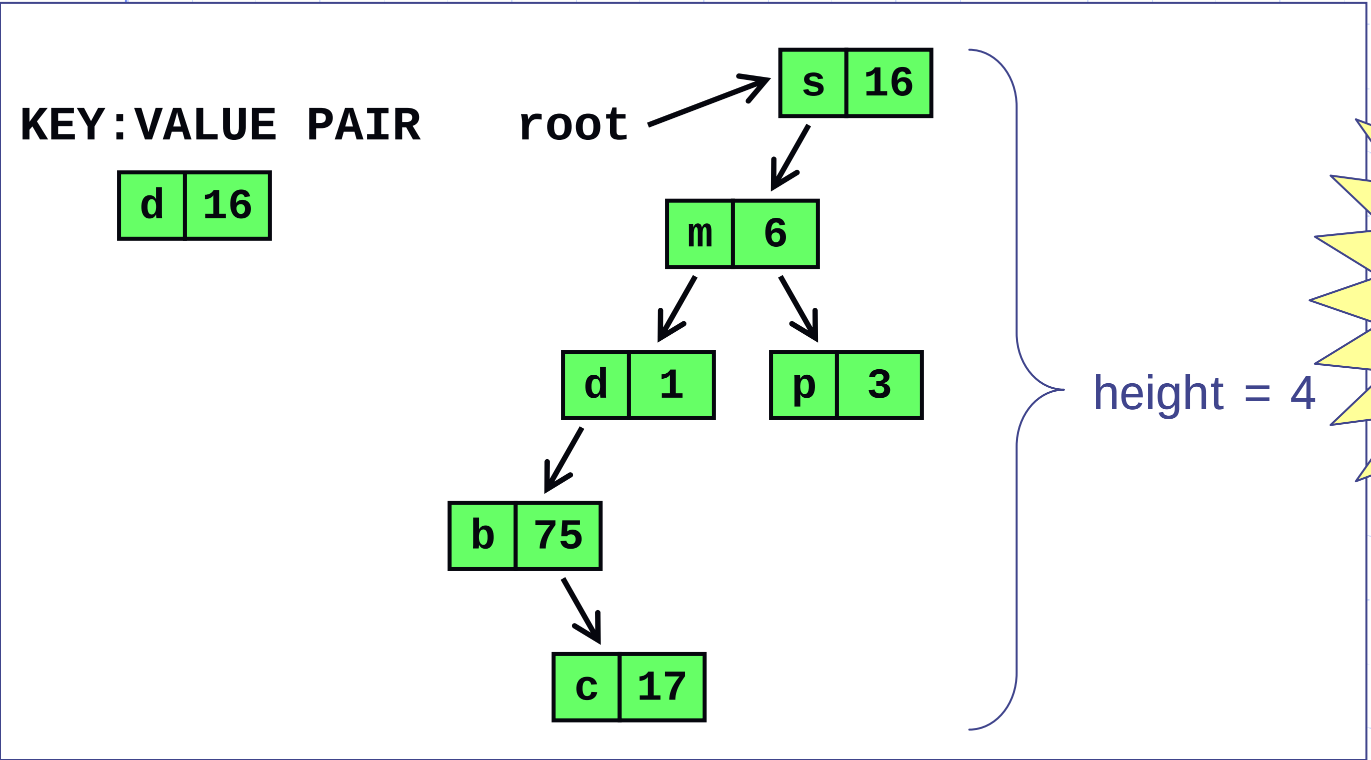
Map (as a hash table)

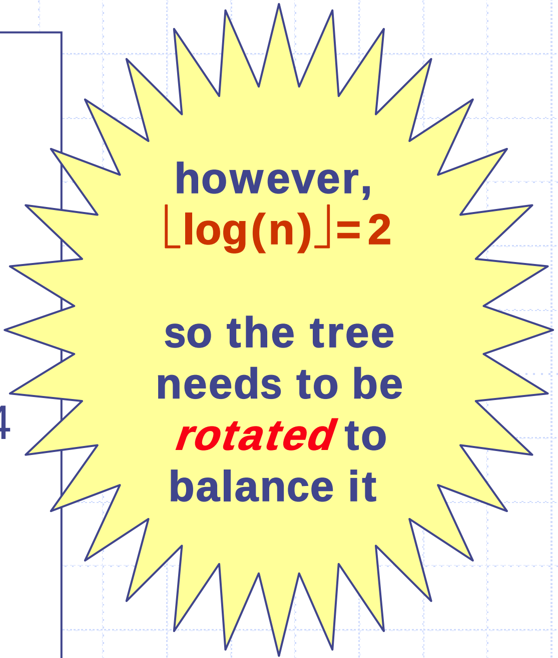
- a ***Map*** is a lookup table that takes a ***key*** and return a ***value***
  - the most common implementation is as a hashtable (hashmap)



Map (as a balanced tree)

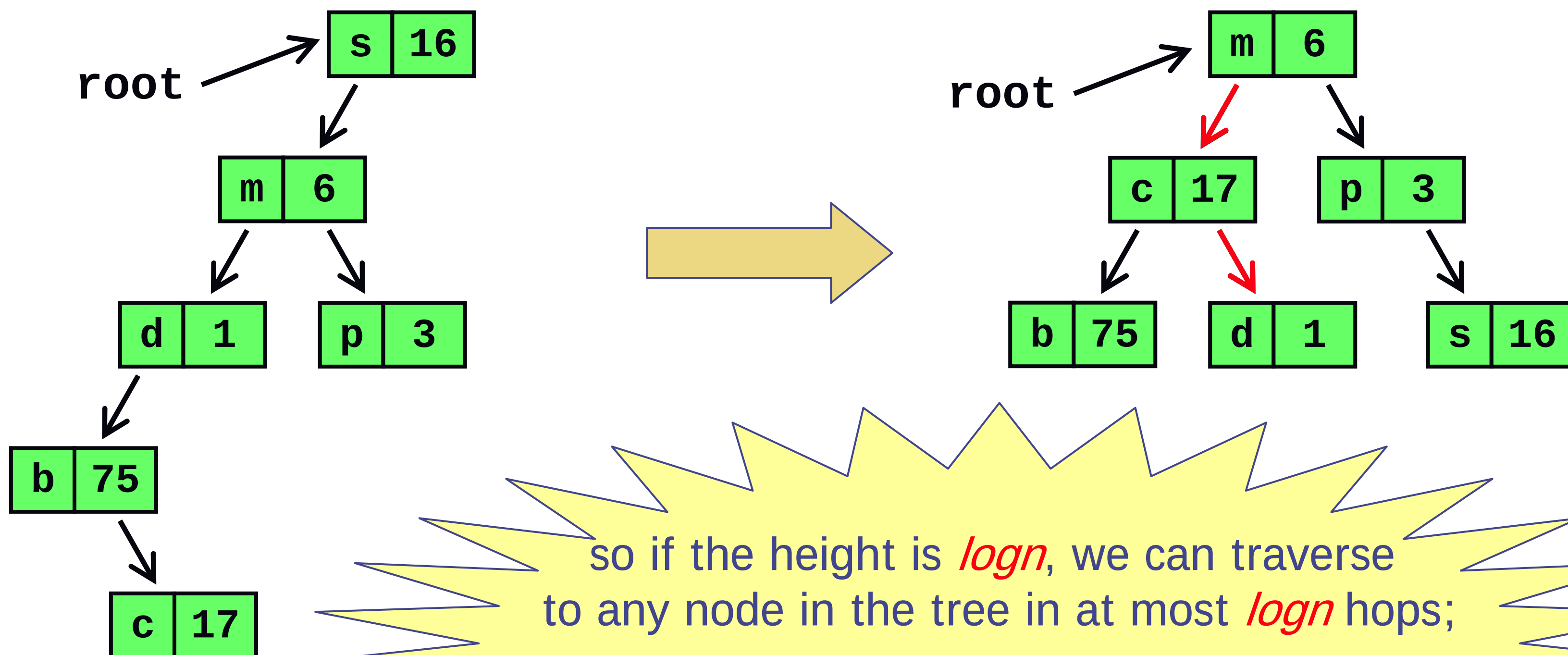
- we can also implement a map as a balanced binary tree
  - "***binary***" trees have two children per node
  - "***balanced***" trees use algorithms to maintain the height of the tree at or near ⌊log(n)⌋, where n is the number of elements in the tree
  - when we refer to a balanced binary tree we are actually talking about "***balanced binary search trees***" – which maintain a sorted order of elements





Map (as a balanced tree - 2)

- tree implementations, such as AVL trees, red-black trees, or 2-3 trees use various algorithms to “rotate” the nodes to keep balanced

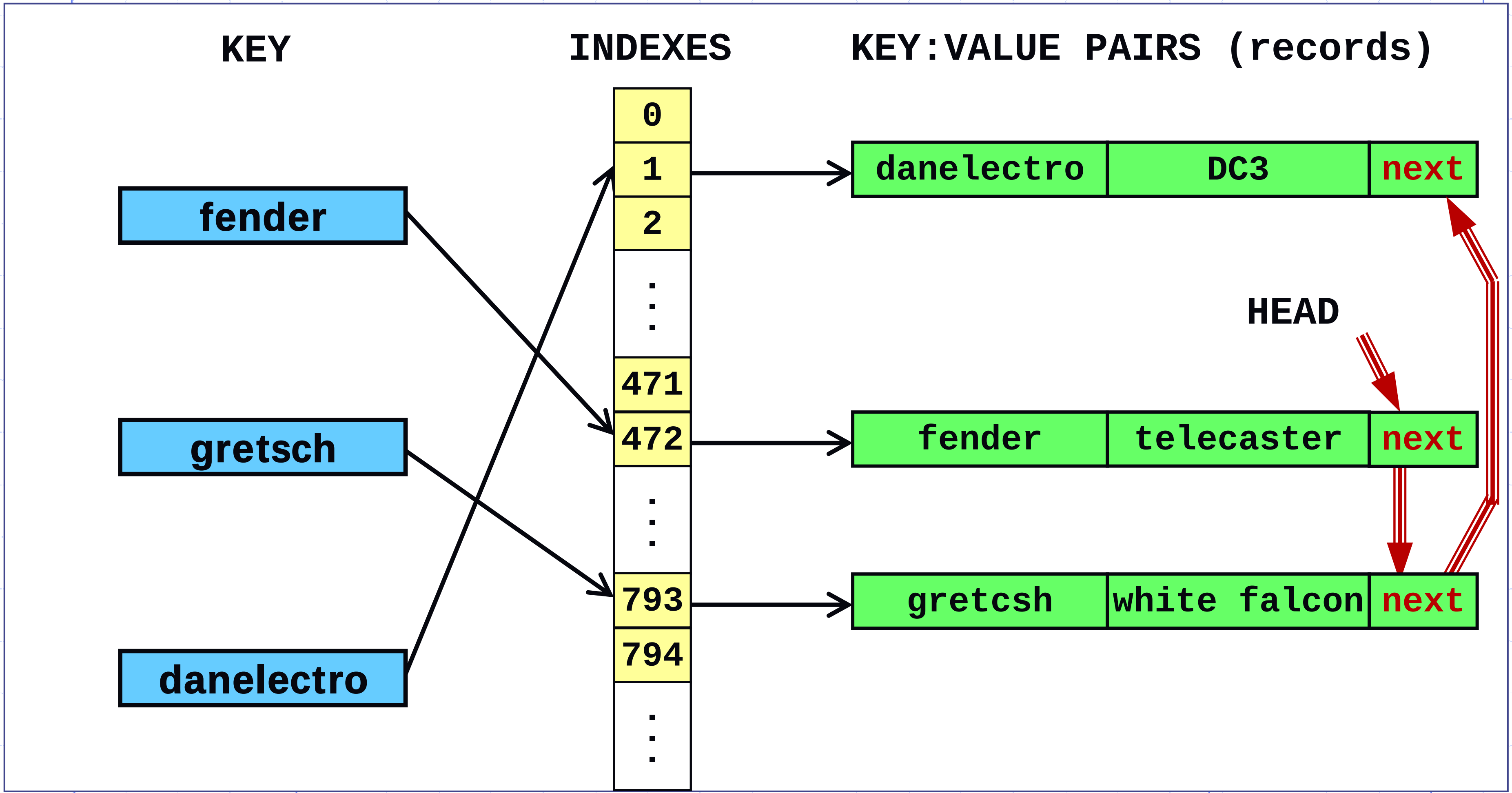


so if the height is *logn*, we can traverse to any node in the tree in at most *logn* hops;

for example, to get to any of the leaves (b,d,s) it will take at most 2 hops from the root

Map (as a linked hash table - 3)

- with a linked hash table, two data structures are created and maintained as nodes are added and deleted …



Maps: HashMap / TreeMap / LinkedHashMap

Java provides all three implementations (hash, tree, linkedhash)

- Java maps have a maps have a few methods that sets do not:

containsKey(Object key)

- **Returns true if this map contains a mapping for the specified key.**

containsValue(Object value)

- **Returns true if this map maps one or more keys to the specified value**

entrySet()

- **Returns a Set view of the mappings contained in this map.**

equals(Object o)

- **Compares the specified object with this map for equality.**

keySet()

- **Returns a Set view of the keys contained in this map.**

important: you cannot iterate over a map directly,
instead you iterate over its entryset or keyset

Java Map Efficiency

<u>*Summary of efficiency (Java maps / sets) ...*</u>

Operation	Method	HashSet	TreeSet	LinkedHashSet
Insert	**Set.add(Object)**	**O(1)**	**O(logn)**	**O(1)**
Traverse	**Set.Iterator**	**O(n+h)**	**O(n)**	**O(n)**
Find	**Set.contains(Object)**	**O(1)**	**O(logn)**	**O(1)**
Delete	**Set.remove(Object)**	**O(1)**	**O(logn)**	**O(1)**

<u>*Summary of efficiency (Arrays and Linked Lists)*</u>

Operation	Method	ArrayList	LinkedList
Insert	**List.add(Object)**	**O(1)**	**O(1)**
	List.add(Index, Object)	**O(n)**	**O(n)**
Traverse	**List.Iterator**	**O(n)**	**O(n)**
Find	**List.contains(Object)**	**O(n)**	**O(n)**
Delete	**List.remove(Index)**	**O(n)**	**O(n)**

Primitive Array	Sorted Array
n/a	**O(logn)**
O(1)	**n/a**
O(n)	**O(n)**
O(n)	**O(logn)**
O(1)	**O(n)**

Applications of Hash Tables

- Set Membership

  Problem: determine if an item is the member of some group of items

  - examples:
    - check if name (computer, or network, or whatever) is valid
    - check if a word in a computer program is a known symbol

  - want fast insert and/or lookup of keys

  - use a HashSet (ie: no stored values, just keys)

Dictionary (Map) Lookups

Problem: given a known key, look up an associated value

- use a HashMap (ie: store a key:value pair)
  - use when we want fast insert/lookup, and sort order is not important
- typical use is to store a record that can be accessed via a unique key
- examples:
  - a userid / password
  - destination / next hop

Simple Translations

Problem: we have a finite set of values that need to be translated to some other values

- use a hashMap, keys are the things we want to translate, values are the translation

- Note: only use a map when you cannot translate by a simple algorithm

- examples:
  - cryptography (change one char sequence to another)
  - auto-correction of typing

Counting / Enumerating

Problem:

- we need to count items that have non-integer names (and we don't want to have to search for the item being counted each time its value needs to be incremented)
- probably we need to be able to look up the counts quickly at some time in the future
- use a map where the key is the thing to be counted, and the value is the count
- examples:
  - number of words in a text
  - number of times a pattern occurs

Sorting / Grouping

Problem: we have input with a known uniform distribution and we want to put items into sorted order

- need a hash function that *maintains an ordering* when it hashes the key, such as

$$\mathbf{h(K) = \lfloor K/Kmax \rfloor * m \quad \Rightarrow \quad 0 < K/Kmax < 1}$$

where:

K	is the key
m	is number of buckets
Kmax	is the maximum value for a key

this function returns a value between 0 and 1

- examples:
  - sorting the index of mp3 songs on a website

since the ratio K/Kmax is always increasing as K increases, the buckets will always be in ascending order

therefore we can use this to sort in O(n) -> called a ***bucket sort***

The End