Lecture 5
Data Structures (DAT037)

Ramona Enache
(with slides from Nick Smallbone)

Hash Tables

A **hash table** implements a set or map

The plan:
- take an array of size k
- define a hash function that maps values to indices in the range {0,...,k-1}

Hash Tables – Take 1

- Example:

    if the values are integers, hash function might be

`h(n)=n mod k`

- To find, insert or remove a value x, put it in index h(x) of the array

Hash Tables – Take 1

- Implementing a set of integers, suppose we take a hash table of size 5 and a hash function `h(n) = n mod 5`

The table contains 5, 17 and 8 already

0	1	2	3	4
5		17	8	

Inserting 14 gives:

0	1	2	3	4
5		17	8	**14**

Hash Tables – Take 1

- This naïve idea doesn't work.
  What if we want to insert 12 into the set?
- We should store 12 at index 2, but there's already something there!
- This is called a **collision**

Problems with Hash Tables – Take 1

1. Sometimes two values have the same hash – this is called a collision

Two ways of avoiding collisions, chaining and probing – we will see them later

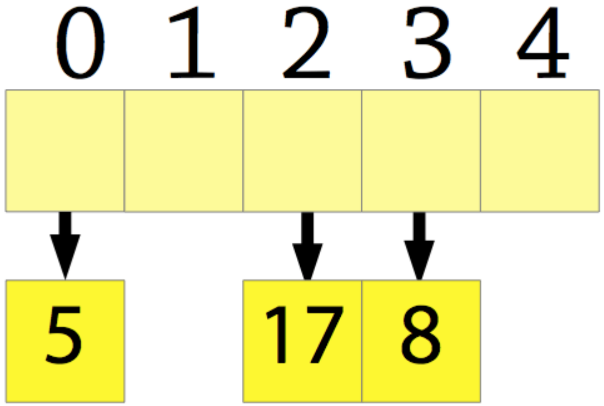
2. The hash function is specific to a particular size of array

Allow the hash function to return an arbitrary integer and then take it modulo the array size:

```
h(x) = x.hashCode() mod array.size
```

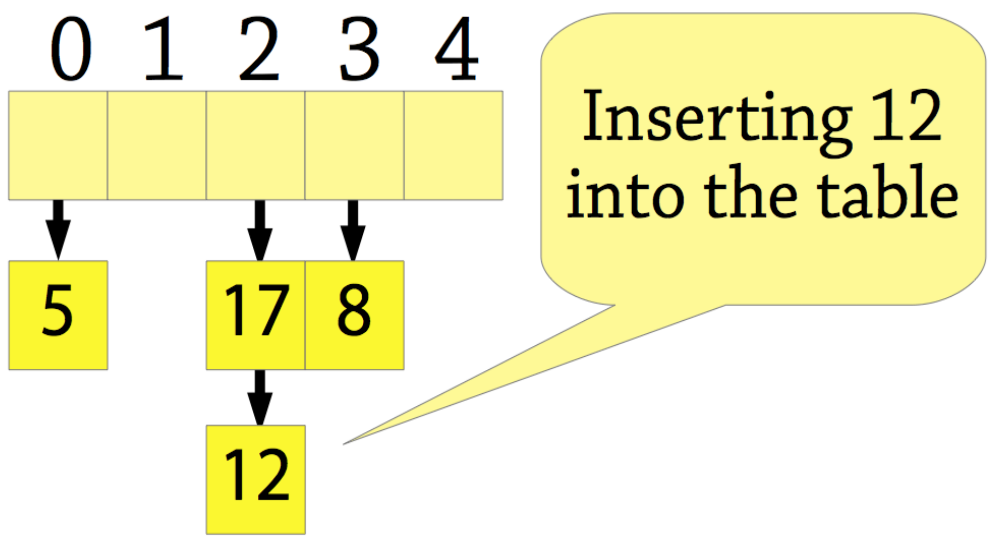
Chaining: Hash Tables – Take 2

- Instead of an array of elements, have an array of linked lists
- To add an element, calculate its hash and insert it into the list at that index



Chaining: Hash Tables – Take 2

- Instead of an array of elements, have an array of linked lists
- To add an element, calculate its hash and insert it into the list at that index



Chaining: Hash Tables – Take 2

- Performance
- If the linked lists are small, chained hash tables are fast
- If the size is bounded, operations are O(1) time

But if they get big, everything gets slow ☹

- Observation: the array must be big enough
- If the hash table gets too full (a high load factor), allocate a new array of about twice the size (rehashing)

Chaining: Hash Tables – Take 2

- Consider a hash table of size 2^16 and the sequence 1, 10, 100…10^n. The hash function is h(x) = x mod 2^16.

What is the problem with this ?

1. The odd positions will never be filled
2. The function is not efficiently computable
3. One of the cells will contain most of the elements for N large enough
4. All of the above

govote.at
Code 161121

Hash Functions

- Observation 2: the hash function must evenly distribute the elements!
- If everything has the same hash code, all operations are **O(n)**

Hash Functions

- What is wrong with the following hash function on strings?

Add together the character code of each character in the string (character code of a=97,b=98,c=99etc.)

Hash Functions

- Maps e.g. bass and bart to the same hash code! (s + s = r + t)
- Maps creative and reactive to the same hash code (anagrams)
- Similar strings will be mapped to nearby hash codes – does not distribute strings evenly !

- Even though collisions are hard to avoid, we want, either a good distribution or to have a better control over what gets to have the same hash code!

Hash Functions

- An idea: map strings to integers as follows:

`s0 · 128^n-1 + s1 · 128^n-2 + ... + sn-1`

- where si is the code of the character at index i
- If all characters are ASCII (character code 0 – 127), each string is mapped to a different integer!
- Similar to representation of integers in binary!

Hash Functions

- In many languages, when calculating

`s0 ·128^n-1 +s1 ·128^n-2 +...+sn-1,`

- the calculation happens modulo 2^32 (integer overflow)
- So the hash will only use the last few characters!
- Solution: replace 128 with 37 s0 ·37^n-1 +s1 ·37^n-2 +...+sn-1
- Use a prime number to get a good distribution This is what Java uses for strings

Linear Probing: Hash Tables – Take 3

Another way of dealing with collisions is **linear probing**

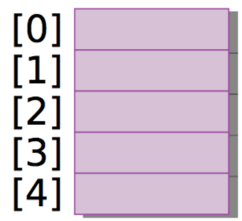
- Uses an array of values, like in the naïve hash table
- If you want to store a value at index i but it's full, store it in index i+1 instead!
- If that's full, try i+2, and so on
- ...if you get to the end of the array, wrap around to 0

Linear Probing: Hash Tables – Take 3

Question

What happens if we use linear probing for the following example ?

Tom Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

1. Dick, Sam, Pete, Harry, Tom
2. Sam, Pete, Dick, Harry, Tom
3. Dick, Pete, Harry, Sam, Tom
4. Tom, Dick, Sam, Harry, Dick

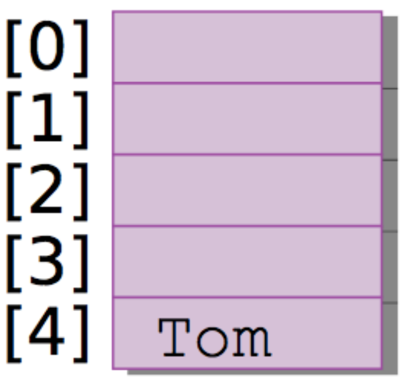
govote.at

Code 28871

Linear Probing: Hash Tables – Take 3

Answer:

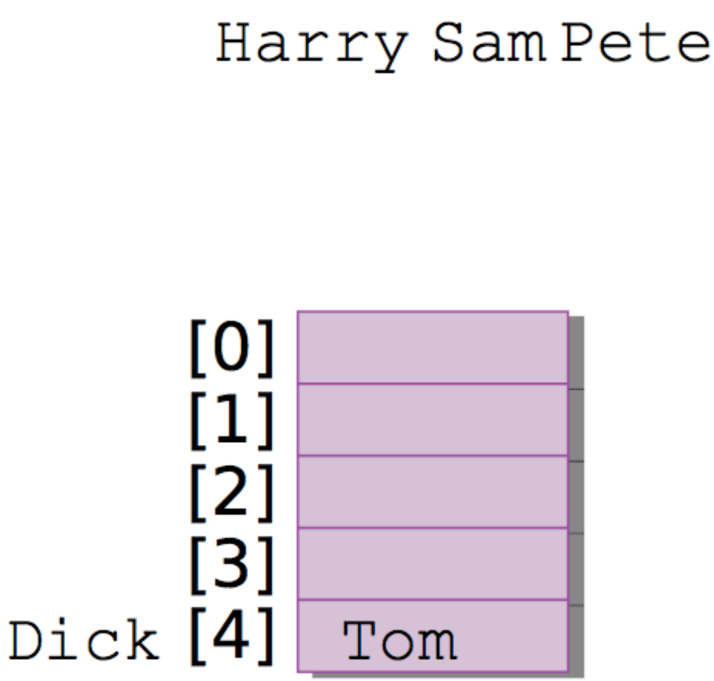
Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

Answer:

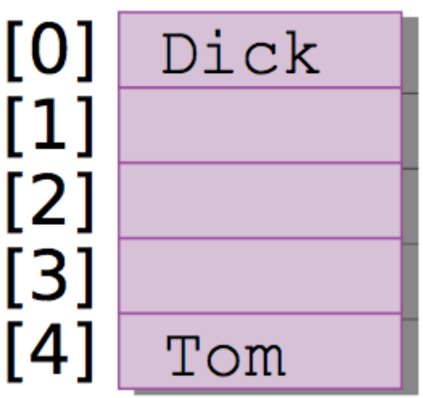


Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

Answer:

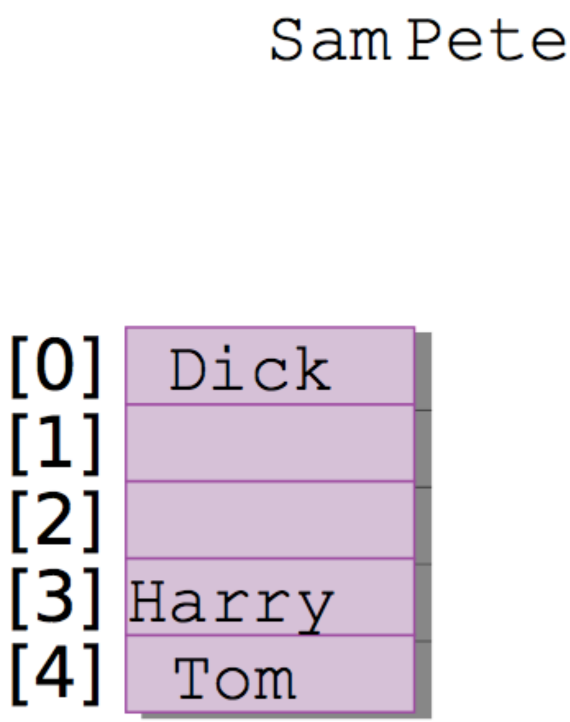
Harry Sam Pete



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

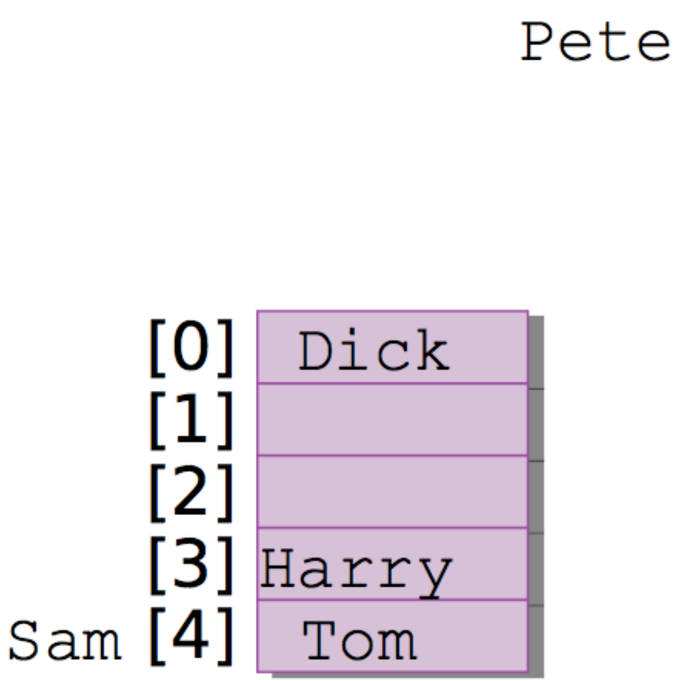
Answer:



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

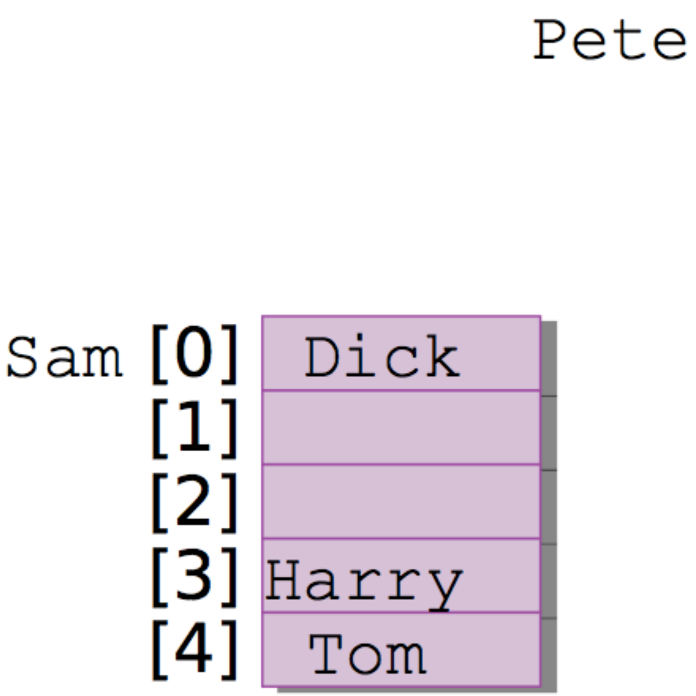
Answer:



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

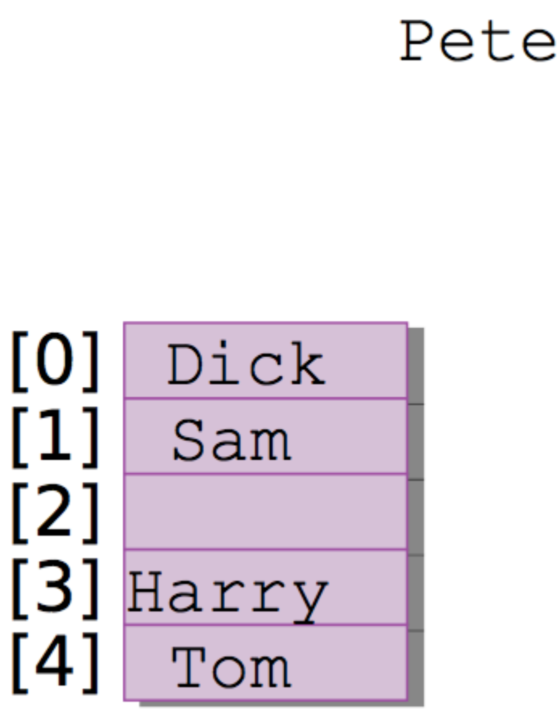
Answer:



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

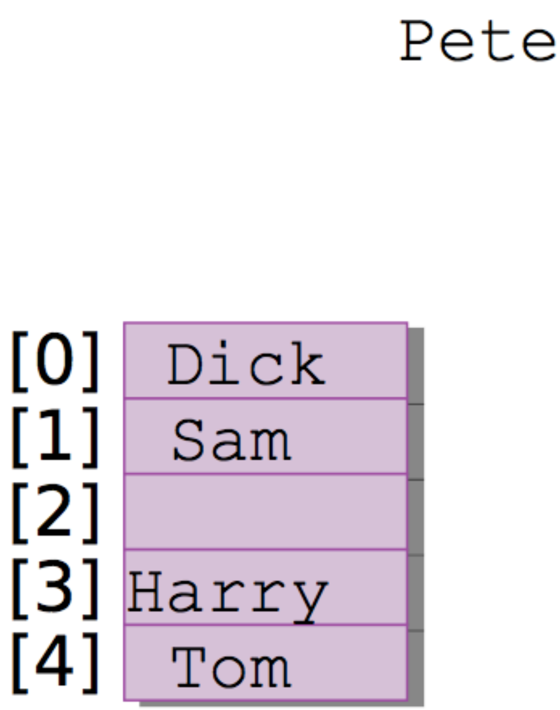
Answer:



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

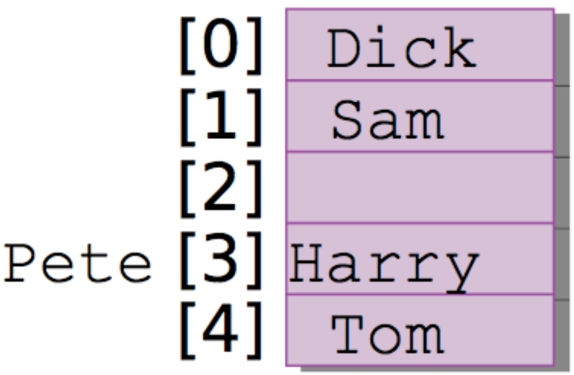
Answer:



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

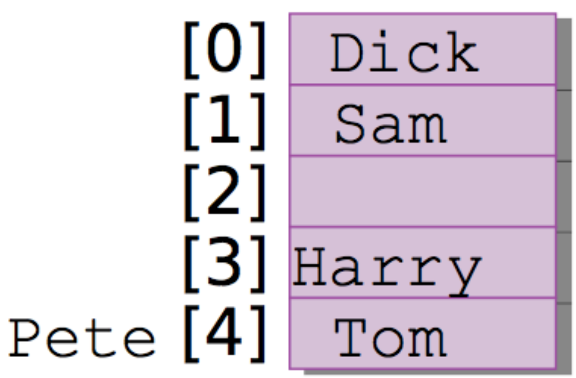
Answer:



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

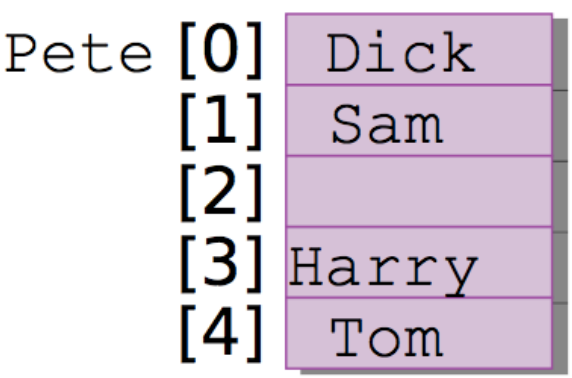
Answer:



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

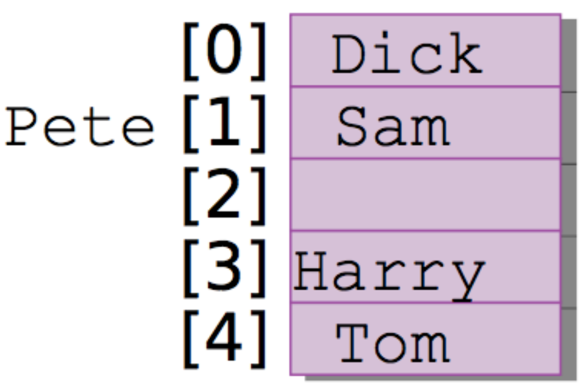
Answer:



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

Answer:



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Linear Probing: Hash Tables – Take 3

Answer:

[0]	Dick
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	[illegible]	[illegible]

To find "Pete" (hash 3), you must start at index 3 and work your way all the way around to index 2

Linear Probing: Hash Tables – Take 3

Similar things will happen with our previous example 10^n with hash code 0 !

Linear Probing: Hash Tables – Take 3

- To find an element under linear probing:

  - Calculate the hash of the element, i
  - Look at array[i]
  - If it's the right element, return it!
  - If there's no element there, fail
  - If there's a different element there, search again at index

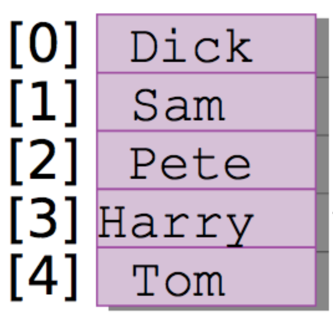
    `(i+1) % array.size`
- We call a group of adjacent non-empty indices a **cluster**

Linear Probing: Hash Tables – Take 3

- Deleting with linear probing

- we can’t just delete an element ☹

Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

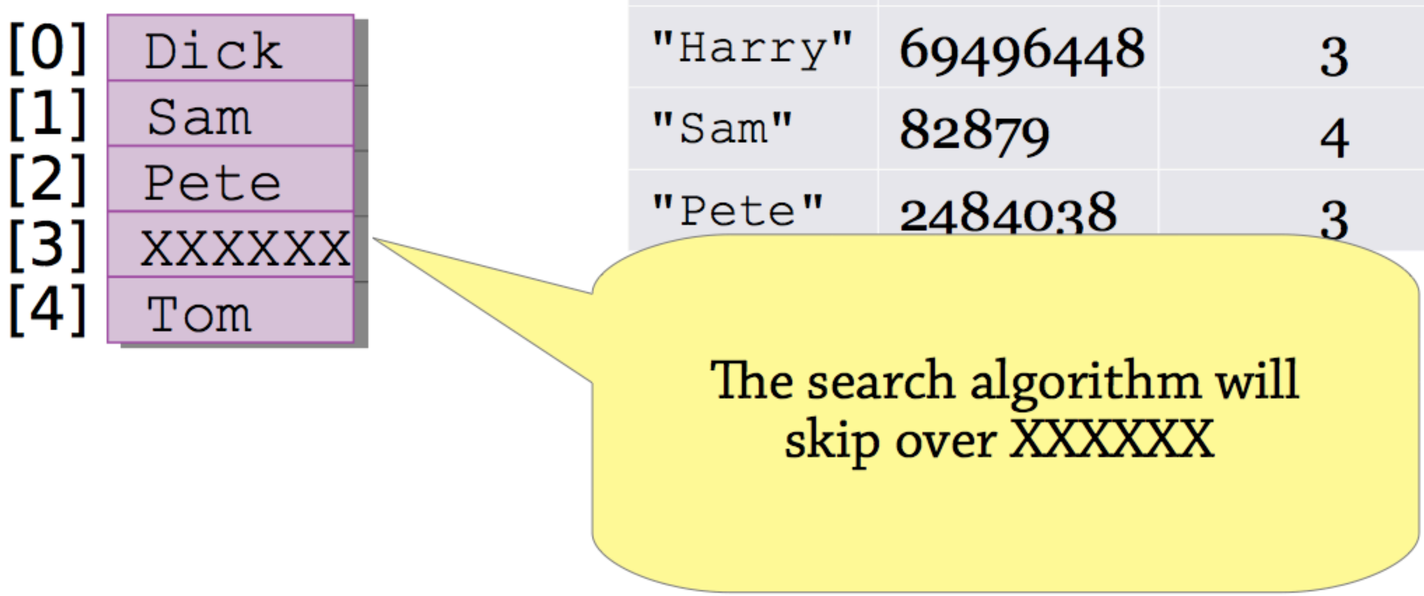


If we remove Harry,
Pete will be in the wrong cluster
and we won't be able to find him

Linear Probing: Hash Tables – Take 3

Instead we mark it as deleted (lazy deletion)

Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



Linear Probing: Hash Tables – Take 3

- It's useful to think of the **invariant** here:
- *Linear chaining*: each element is found at the index given by its hash code
- *Linear probing*: each element is found at the index given by its hash code, or a later index in the same cluster
- Naïve deletion will split a cluster in two, which may break the invariant
- Hence the need for an empty value that does not mark the end of a cluster

Linear Probing: Hash Tables – Take 3

Linear probing performance

- To insert or find an element under linear probing, you might have to look through a whole cluster of elements
- Performance depends on the size of these clusters:
  - Small clusters – expected O(1) performance
  - Almost-full array – O(n) performance
  - If the array is full, you can't insert anything!
- Thus you need:
  - to expand the array and **rehash** when it starts getting full
  - a hash function that distributes elements evenly
- Same situation as with linear chaining!

Linear Probing vs Linear Chaining

- In *linear chaining*, if you insert several elements with the same hash i, those elements become slower to find
- In *linear probing*, elements with hash i+1, i+2, etc., will belong to the same cluster as element i, and will also get slower to find
- If the load factor is too high, this tends to result in very long clusters in the hash table – a phenomenon called primary clustering

Linear Probing vs Linear Chaining

- Linear probing is more sensitive to high load
- On the other hand, linear probing uses less memory for a given load factor, so you can use a bigger array than you would with chaining

load factor (#elements / array size)	**#comparisons (linear probing)**	**#comparisons (linear chaining)**
0 %	1.00	1.00
25 %	1.17	1.13
50 %	1.50	1.25
75 %	**2.50**	1.38
85 %	3.83	1.43
90 %	5.50	1.45
95 %	10.50	1.48
100 %	—	1.50
200 %	—	2.00
300 %	—	**2.50**

Hash Table Design

- Several details to consider:
  - Rehashing: resize the array when the load factor is too high
  - A good hash function: need an even distribution
  - Collisions: either chaining or probing
- Hash tables have expected (average) O(1) performance if the hash function is random (there are no patterns) – but it's normally not!
- Nevertheless, performance is O(1) in practice with decent hash functions.
- So – theoretical foundations a little shaky, but very good practical performance ☺

Implementing Hash Tables in Java

- **Hashtable**
  - synchronized
  - more control over hashing process
  - legacy class – use ConcurrentHashMap instead!

- **HashMap**
  - unsynchronized
  - automatic rehashing

- java.util.Objects.hash.

```
public int hashCode() {
  return Objects.hash(field1, field2, field3); }
```

Implementing Hash Tables in Haskell

- **Data.Hashtable**
  - uses linear probing
  - IO monad – not pure
  - mutable data structure – insert/deletion

- Not a hash table – **Data.Map**
  - based on balanced binary trees
  - persistent data structure

Later

Hash Functions in the Real World

- MurmurHash http://en.wikipedia.org/wiki/MurmurHash
- CityHash http://en.wikipedia.org/wiki/CityHash
- Various hash functions from Google:

com.google.common.hash.

To Do

Read from the book

+ 5.1 – 5.6

Extra reading:

+ not in exam/dugga: Perfect Hashing (5.7.1), Universal Hashing (5.8)

Implement:

+ hash tables in your favourite programming language