Academic excellence for
business and the professions

IN2002 Data Structures and Algorithms

Lecture 6 – Hash Tables

Aravin Naren
Semester 1, 2018/19

Learning Objectives

- Understand and be able to use the data structures set, tables and hash tables

- Be able to understand, apply and develop algorithms to handle the data structures above. Including:
  - Store key-value pairs
  - Retrieve values by key
  - Delete key-value pairs

Sets

Sets

Sets in maths are a collection of things, no matter in which order.
Sets are the simplest case of a table, in which the key is the value

- The set stores values
- The set can receive new values and store them
- The set allows checking whether a value is stored
- Values can be deleted from the set

There are many ways of implementing sets

An abstract data type set

```
public interface Set {

    // Add the key to the set
    void insert(int key);

    // Is the key in the set?
    boolean search(int key);

    // Delete the key from the set, if present
    void delete(int key);
}
```

Known set implementations

Unordererd array:

- insert at the end: O(1)
- (sequential) search: O(n)
- delete and swap: O(1)
- search and delete: O(n)

Ordered array:

- insert in proper place: O(n)
- binary search: O(log n)
- delete and shift: O(n)
- search and delete: O(n)

Unordered linked list:

- insert at head: O(1)
- (sequential) search: O(n)
- delete: O(1)
- search and delete: O(n)

How can we do it faster ?

Main problem: searching the right index.

Takes long when using comparisons

Idea: use **Hashing**

- calculate the index from the key: gives constant time for finding the address.

Some more implementations of sets

Times for various implementations:

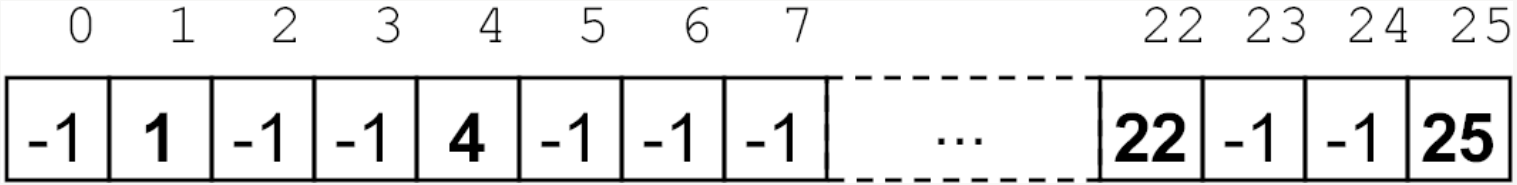
	insert	**search**	**delete**
unordered array	O(1)	O(n)	O(n)
ordered array	O(n)	O(log n)	O(n)
linked list	O(1)	O(n)	O(n)
simple tree (best)	O(log n)	O(log n)	O(log n)
(worst)	O(n)	O(n)	O(n)
balanced search tree	O(log n)	O(log n)	O(log n)
hash table	**O(1)**	**O(1)**	**O(1)**

How to obtain O(1) in a set of integers?

First approach:

- take an array `int array[max+1];` where `max` is the highest value to store
- initialise the values in the array to `-1`
- store the values in slot `index = key`.

  E.g., `array[24] = 24;`



First approach...

insertion of key takes O(1):

```
void insert(int key){array1[key] = key;}
```

search for key takes O(1):

```
boolean search(int key) {
    if( array1[key] != -1) return true;
    else  return false;
}
```

deletion of key takes O(1):

```
void delete(int key) { array1[key] = -1;}
```

Example

Store 24, 31, 42 and 81 in set

We get O(1), but:

- the maximum value (81) must be known when creating the array
- terribly wasteful (array of size 82 for 4 values!)

How to use less space

Keep the index inside the array bounds.

Remember the modulo operator ?

index <- (key mod length) + 1

makes sure the key is inside the array
(Java, 'mod' is '%' and '+1' is not needed)

Ideal case

Store 24, 31, 42 and 81 in set:

- Ideally, we could create `int array2[4];` and
  store value in `array2[value % 4];` still O(1)
- To search value, check at `array2[key % 4];` still O(1)
- To delete value, delete `array2[key % 4]`

This is a **hash table**

Tables (maps)

Tables (maps)

In sets, the keys were the data.

More often we have indexed data:

- There are other data associated to the keys
- Examples: dictionaries, phone books, price lists, tables of contents

It must be possible to add/remove elements

Examples:

- files/databases in general
- files in a drive
- variables in memory
- commands in a language

What is done in a table

- Data is stored as key-value pairs
- Data is retrieved by key

Implementations:

- ordered and unordered arrays
- lists
- hash tables
- search trees

An abstract data type table

```
public interface Table {
    // Set the value associated with the key
    void store(Key key, Data value);

    // Search for the key, returning true if found
    boolean search(Key key);

    // If the key exists, return the value
    // associated with it
    int getValue(Key key);

    // Delete the key-value, if present
    void delete(Key key);
}
```

The same example revisited

Store 24, 31, 42 and 81 in set:
Ideally, we could create `int array2[4];` and store value in `array2[key % 4];` still O(1)

What if we want to store more values?

- There is no more room in the array

What if we want to store 82 instead of 81?

- The fact that 82>81 is not a problem
- However, 82 mod 4 is 2, and `array2[2]` already contains a value: 42 (this is called a **collision**)

Handling collisions

Solution to the problems above:

- produce an array that has a size equal or reasonably greater than the number of elements you will want to store

- handle collision using some suitable strategy

Hashing

Hashing

A **hash function** calculates the *index* for a *key*

index <- *hash(key)* e.g., *key mod len*

Ideally there should be one unique index for each key (**perfect hash function**)

- Pro: there would be no collisions
- Con: we may not have enough space

Avoiding collisions

General strategy:

- Avoid collisions by distributing the keys uniformly over the addresses, even when there are patterns in the keys occurrence.

Pattern examples:

- more small than large numbers
- more even than odd numbers (or divisible by 4,10, ...)

For modulo function: choose *len* prime

What when collisions occur?

In case of collision, the possible strategies are to:

- store the value with the currently stored ones (buckets, closed hashing)
- look for another empty space and store the value there (open addressing, open hashing)
- modify the hashing function

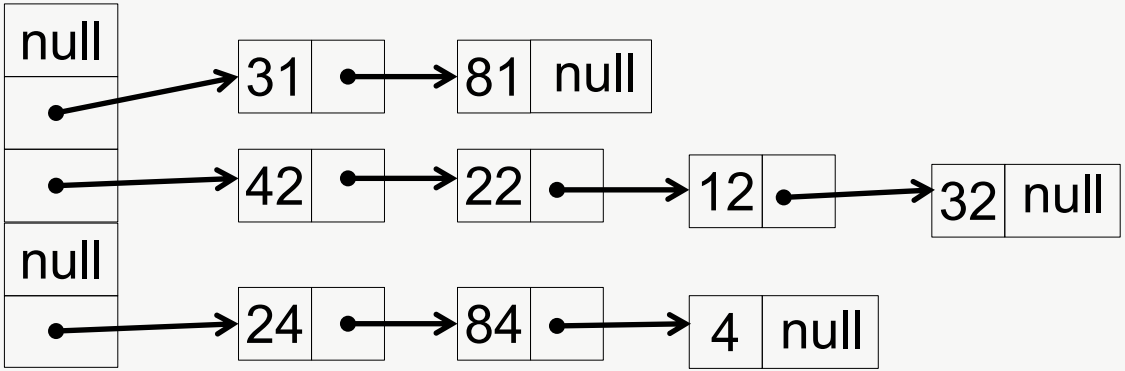
Buckets

The idea is to store duplicates together

- this could be done by using an array with pointers to arrays
- address may be assigned to several keys
- table of fixed size will overflow

A hash table with chaining

- Define the table as an array of objects of type `Node`,
- and create linked lists when collisions occur



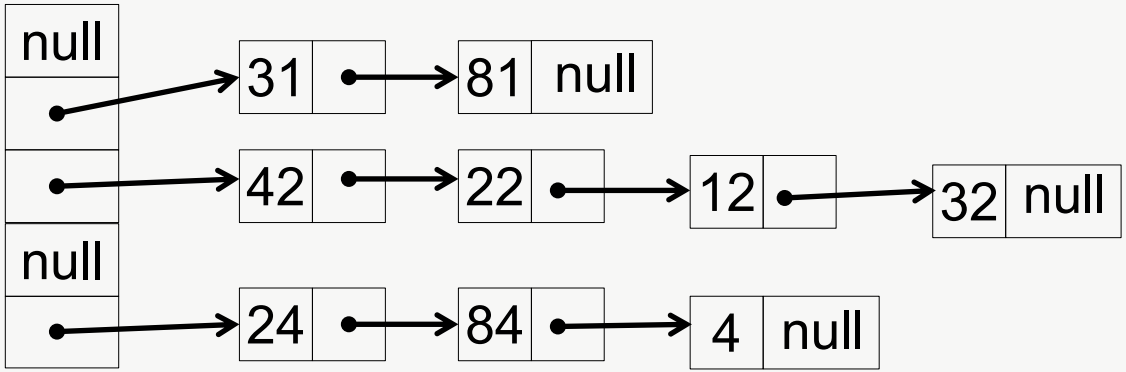
A hash table class in Java

```
class HashTable implements Set {
    private Node[] table;

    public HashTable(int size) {
        table = new Node[size];
    }
    private int hash(int key) {
        return key%table.length;
    }
    public void insert int key { ... }
    public boolean search(int key) { ...}
    public void delete(int key) { ... }
}
```

Searching

To search *key*, sequential search is performed in the linked list *table[hash(key)]*



Search for 22, 6, 100

Searching

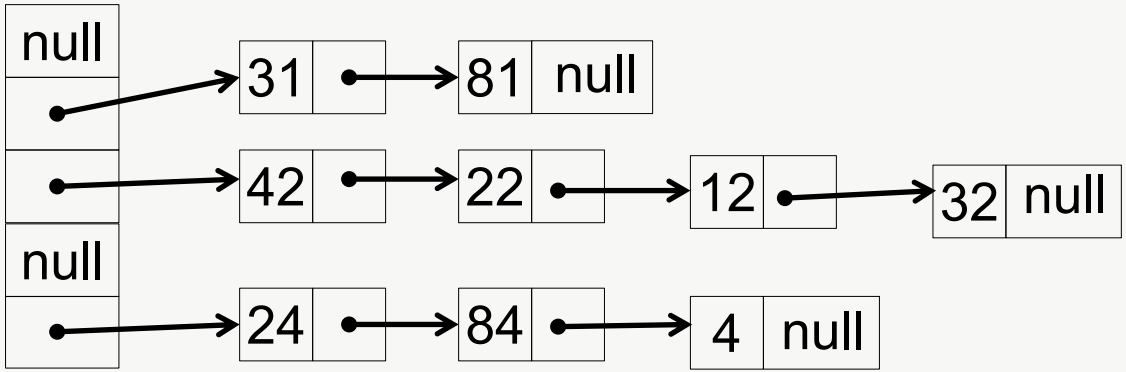
To search *key*, sequential search is performed in the linked list *table[hash(key)]*

Function search (key) :
p ← table[hash(key)]
WHILE (p ≠ null)
    IF value of p = key THEN
        Return key
    p ← next of p
Return null

Insert and delete also work as with linked lists

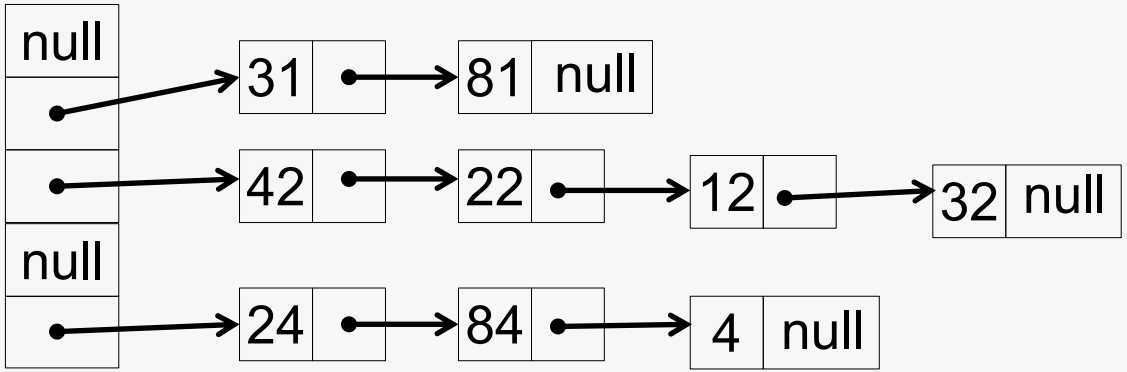
Inserting

Insert 16, 25



Deleting

Delete 81, 22, 31



Analysis of hash table operations

The cost of operations is proportional to

load factor = number of elements / size of table

- If we use a fixed table size, the cost of operations will be O(n), though with a very small constant factor
- If we keep the **load factor bounded**, the cost of operations will be **O(1)**. For this, the table would need to be extended when adding elements (remember extensible arrays).

Extending the hash table

- Simple scheme: create a new table, insert all the keys into it and discard the old one
- Cost is proportional to the size of the table, say *cn*, so analysis is similar to extensible arrays
- If we **extend the table by doubling its size**, the cost of n insertions is at most
  $cn + cn/2 + cn/2^2 + \ldots + c = c\,(2n - 1)$ $\quad O(n)$
- This yields an average, **amortised time of O(1)** per insertion
- However, this cost is unevenly distributed

Hash functions

- Java has a method `hashCode` for every Object
- A **uniform distribution of keys** is essential
- Division modulo a prime number (or a number with no small factors) is simple and often works well
- More complex transformations may be more uniform
- For strings, combine the characters in some way
  - For example, method `hashCode` for `String s` computes

$$31^0\, s[n-1] + 31^1\, s[n-2] + \ldots + 31^{n-2}\, s[1] + 31^{n-1}\, s[0]$$

Open addressing

An alternative: Open Addressing

Idea: to **save the space used by the pointers**, put the overflowing entries (collisions) into the same array.

Need to define **probing** method for where to put and search elements:

- **Linear probing**: place the key in the next free slot
- **Quadratic probing**
- **Double hashing**

Linear probing

When inserting a key k:

- First try $h(k)$
- If that is used, try $(h(k) + 1) \bmod n$,

then $(h(k) + 2) \bmod n$, and so on

On finding a free slot, place the key in it

- All locations in the array can be tried if necessary
- Simple, but subject to clustering

Linear probing – Insertion

5 slots

Hash function h(n) = n%5

Insert 26, 3, 11

Hashing with linear probing

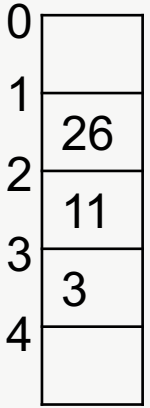
```
public class Set {
    private static String[] table;
    public Set (int size){ table=new String[size];}
    public boolean search(String key) {
        for(int i = key.hashCode()%table.length;
                table[i] != null; i=(i+1)%table.length){
            if (table[i].equals(key))
                return true;
        }
        return false;
    }
}
```

Hashing with linear probing – Search

5 slots

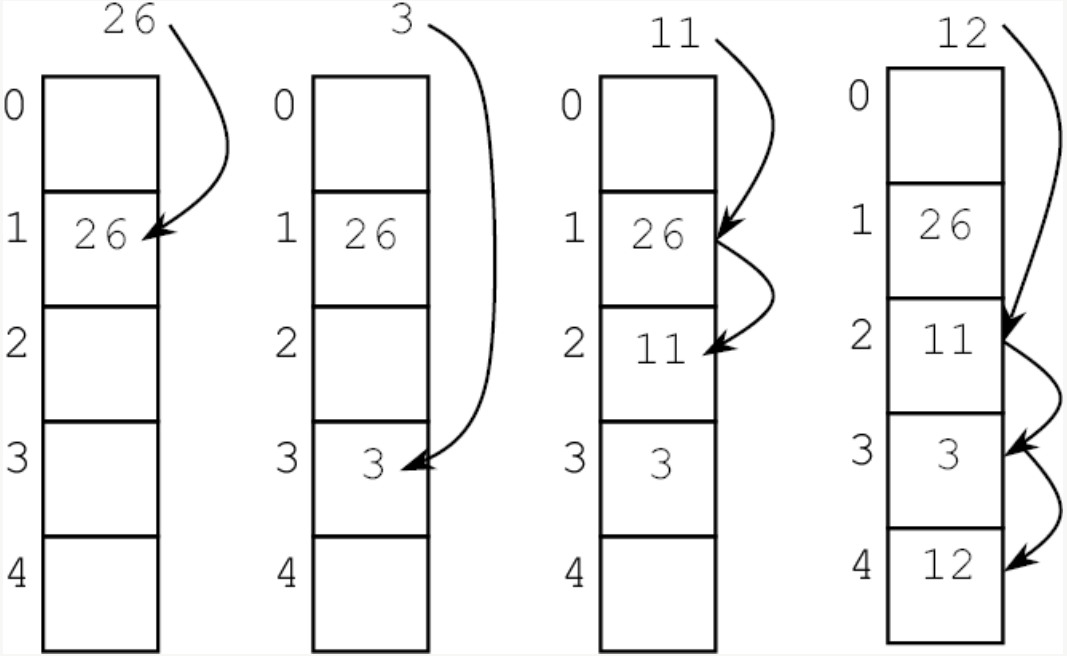
Hash function h(n) = n%5

Search 26, 11, 5



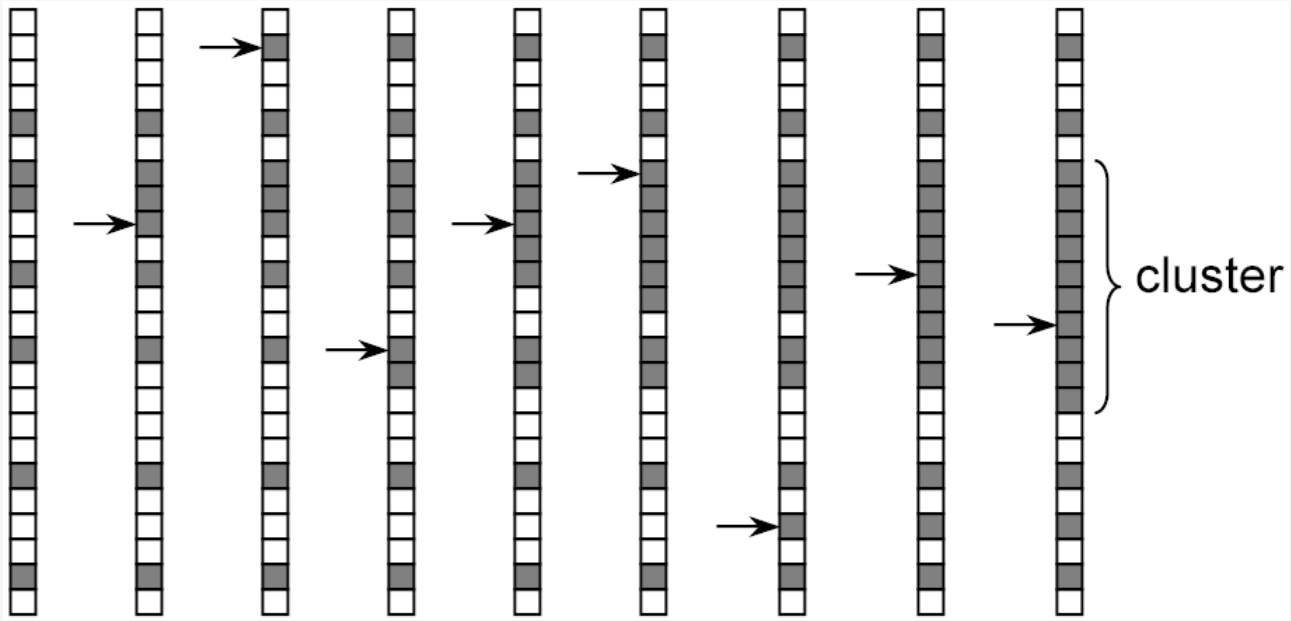
Nearly full tables

When the table is nearly full, performance degrades sharply



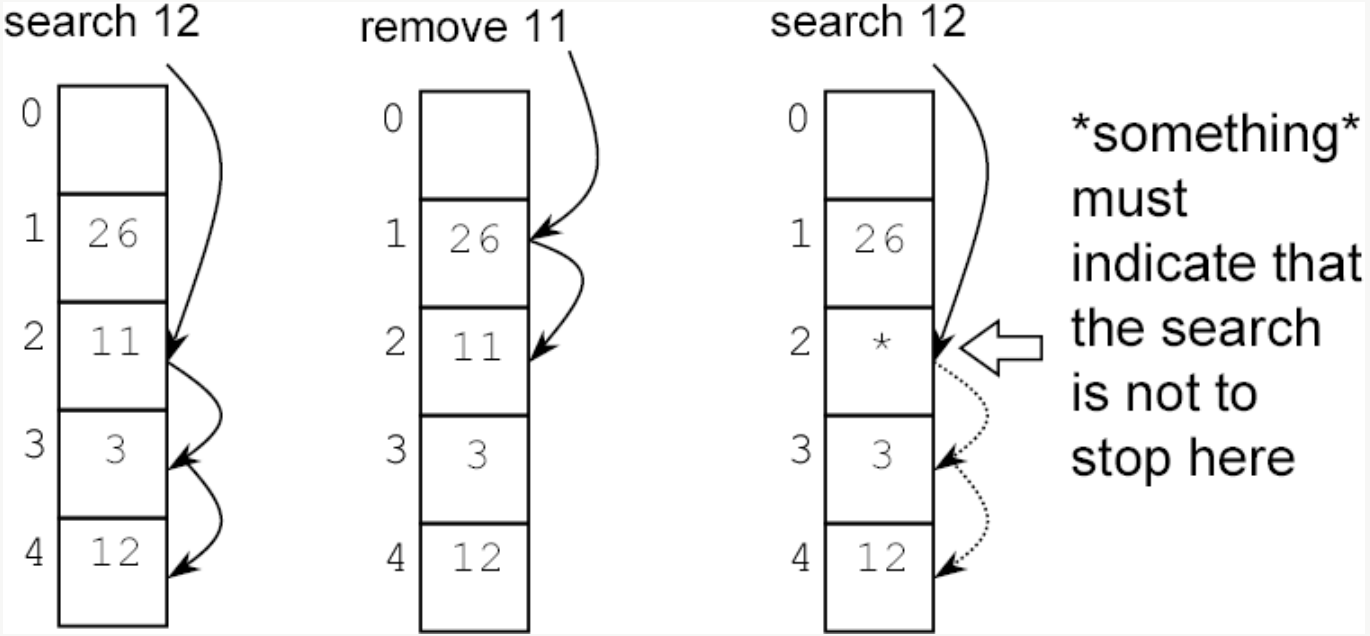
Clustering caused by linear probing

Overflowing addresses tend to group in a region of the array



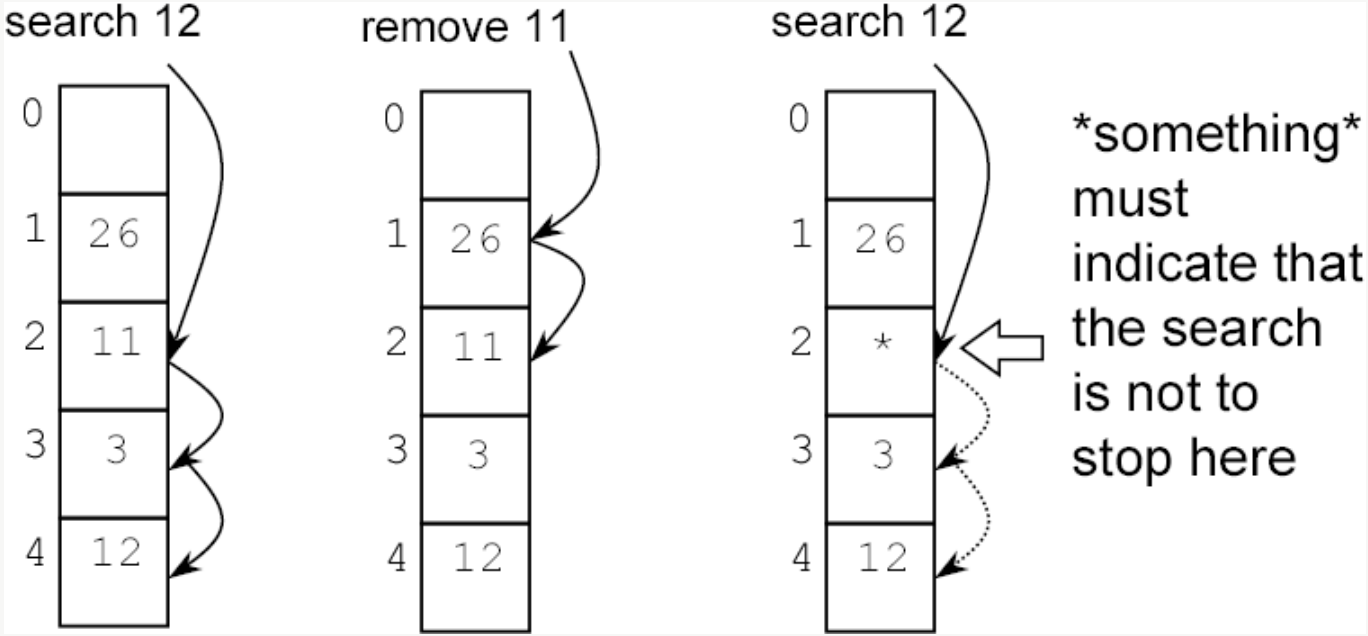
Deleting elements

Deletion requires creating a marker of deleted elements



Inserting after deleting elements…

After the deletion above, insert 6



Quadratic probing

Try *h(k) mod n*, *(h(k)* ***+ 1²****) mod n*, *(h(k)* ***+ 2²****) mod n*, *(h(k)* ***+ 3²****) mod n*, and so on until a free slot is found

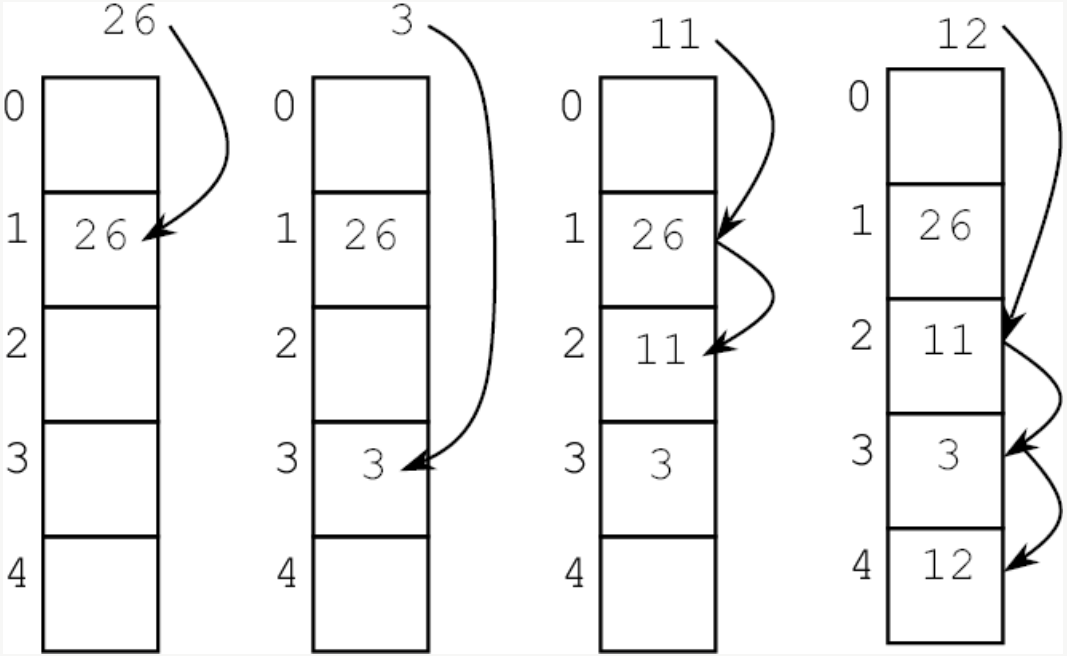
Probes i and j coincide if *n* divides $i^2 - j^2 = (i - j)(i + j)$:

- If *n* is prime, this means *n* divides i - j or i + j, so i = j or i + j = n; so the sequence visits at least (n+1)/2 locations.
- $(h(k) - 1^2)$ mod n, $(h(k) - 2^2)$ mod n, and so on cover some of the other half (under some conditions all).

So it works better to try *h(k) mod n*, *(h(k)* ***+ 1²****) mod n*, *(h(k)* ***-1²****) mod n*, *(h(k)* ***+ 2²****) mod n*, *(h(k)* ***- 2²****) mod n*, and so on.

Quadratic probing

Now insert 6



Analysing quadratic probing

- **Keys that hash to different addresses probe different sequences of addresses, eliminating primary clustering**
- Keys that hash to the same address probe the same sequence of locations, so there is still secondary clustering, but this is less serious as it is more distributed
- Ideally, to avoid secondary clustering, each probe should be different from those that came before
- An approximation to this is to use a second, independent hash function (double hashing)

Double hashing

We need **two hash functions** *h1* and *h2*, such that *h2(k)* is never divisible by *n* nor equal to 0

- Try locations *h1(k) mod n*, *(h1(k) + h2(k)) mod n*, *(h1(k) + 2 h2(k)) mod n*, and so on
- This eliminates both primary and secondary clustering, but hashing twice costs some time
- We could also use a quadratic version:
  - *h1(k) mod n*, *(h1(k) + h2(k)1^2) mod n*, *(h1(k)+h2(k)2^2) mod n*, and so on

Double hashing example

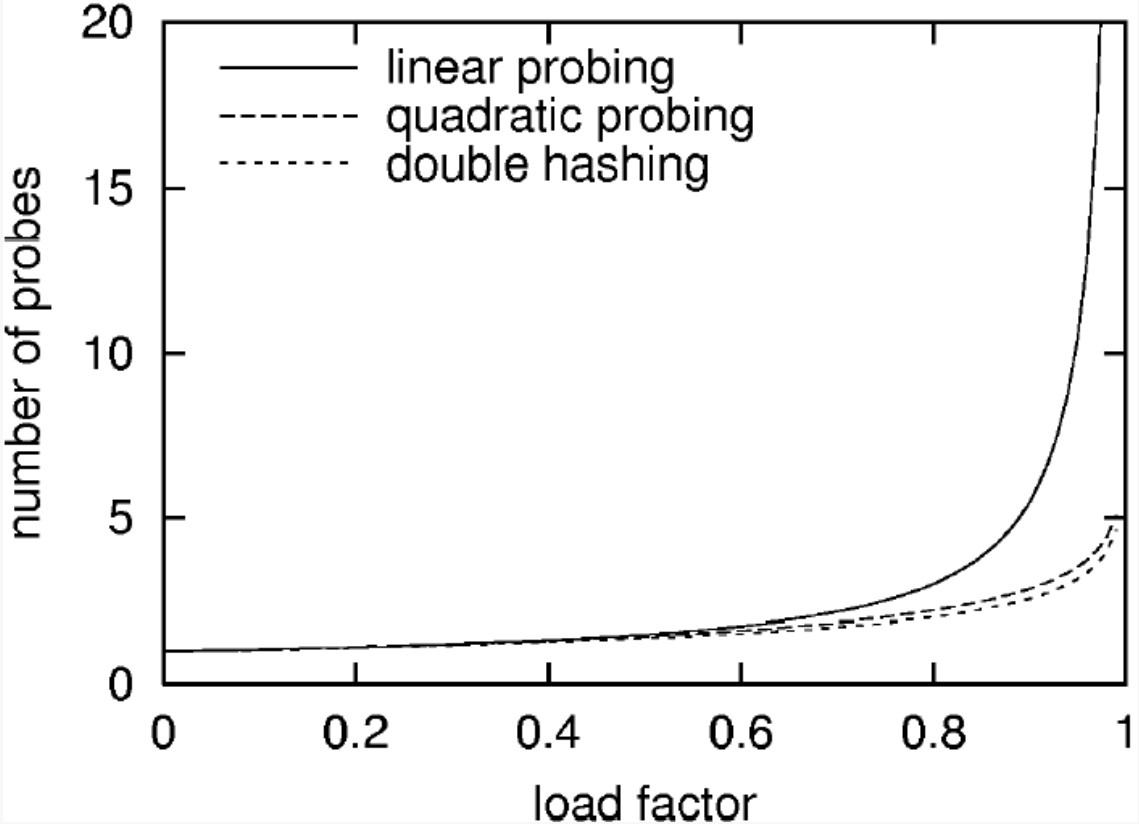
A good choice is to choose a prime p < size and
h2(k) = p - (k % p) (while h1(k) = k % n)

7 slots; h1(k) = k%7; p=5; h2(k) = 5-(k%5)

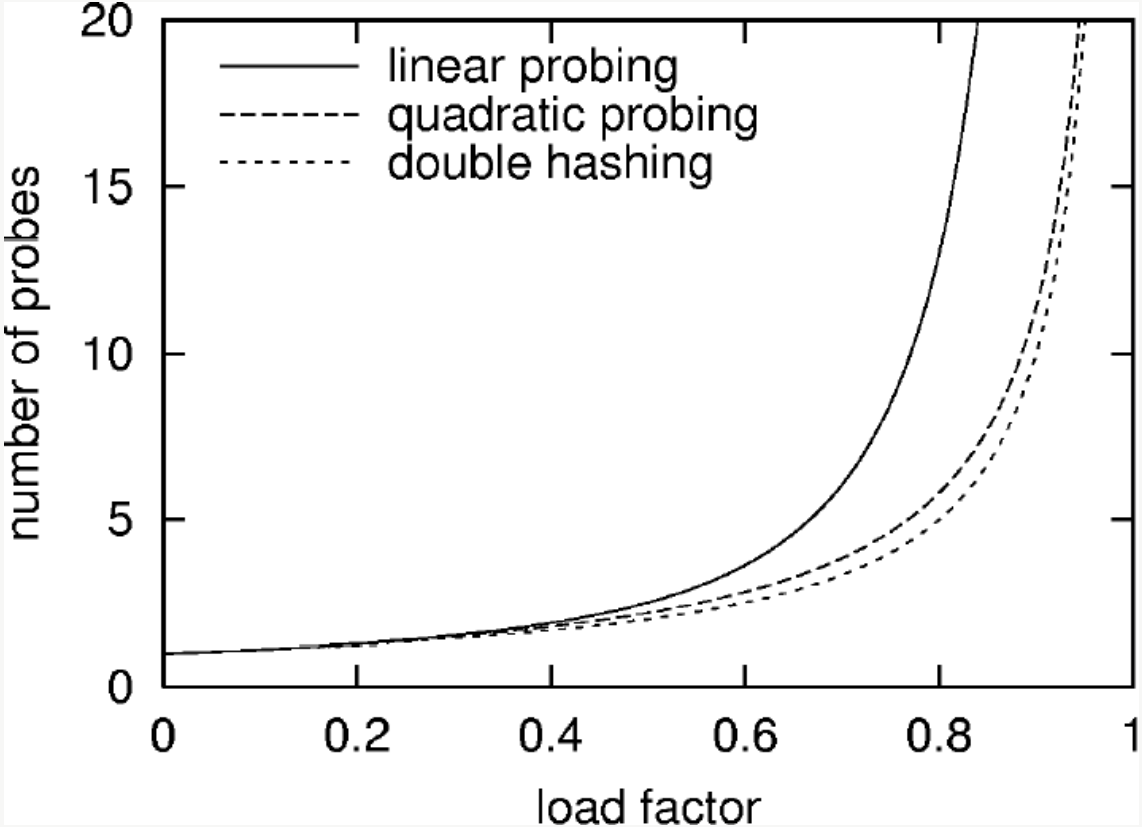
Insert 11, 18, 4

Remember try:
h1(k) mod n; (h1(k) + h2(k)) mod n; (h1(k) + 2h2(k)) mod n

Times for successful search



Times for unsuccessful search



Table/set operations

Search:

hash to initial index and then follow probe sequence until

- the key is found, success,
- an empty location is reached, failure

Insert is like search, but put the key in the empty location

Delete is harder, as many probe sequences pass through a given location.
Solution: indicate deletion with a special value

Sets in Java

There is a number of relevant interfaces and classes in the package **java.util**

Interface **Set** defines a set ADT (extensive interface)

Implementations:

- **HashSet** (using a hash table)
- **LinkedHashSet** (remembers insertion order)

Interface **SortedSet** extends set ADT

Implementations:

- **TreeSet** (keeps keys sorted)

Tables and Hashing in Java

There is a number of relevant interfaces and classes in the package **java.util**

Interface **Map** defines a table ADT (extensive interface)

Implementations:

- **HashMap** (**Hashtable** old sync. version)
- **IdentityHashMap** (test for reference-identical keys)

Interface **SortedMap** extends a map ADT

Implementations:

- **TreeMap** (keeps keys sorted)

When to use Hashtables ?

- Hash tables have relatively low access times for large numbers of elements
- Hashing may be too expensive
- For small collections, simple arrays can be better
- May have high computing overhead when growing
- Good when used for access by name (methods in compiler, telephone numbers in DB)
- HT do not know anything about order (except SortedMap/Set)

Reading

- Weiss: Chapter 19 (Hash tables)

- Drozdek: Chapter 10 (Hashing)

Next session: Trees

Drozdek: Sections 6.1 to 6.6 OR Weiss: Chapters 17 and 18.1-18.3

City, University of London
Northampton Square
London
EC1V 0HB
United Kingdom

T: +44 (0)20 7040 8406
E: ug-compsci@city.ac.uk
www.city.ac.uk/department-computer-science