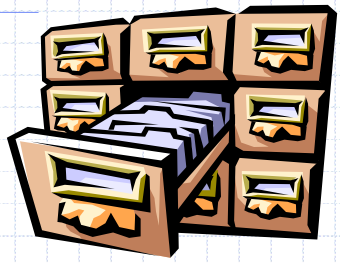
Maps



1

1

Non altrettanto efficiente per ricerca, ma efficiente nell'inserimento.
Nelle mappe no orchinamento → deve scorrere tutto. Ma perché? Perché inserimento su fa in tempo costante.

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database

2

Utile per esempio per i log di un sistema.

NON SET?

2 Key: Non deve esistere duplicazione di chiavi

Ricerca + sostituzione

The Map ADT

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Coppie Chiave-Valore

3

3

Example

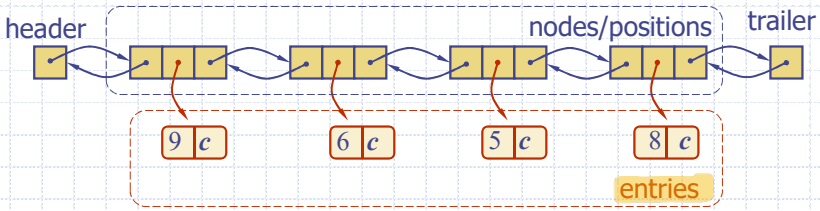
Operation	*Output*	*Map*
isEmpty()	**true**	Ø
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

4

4

A Simple List-Based Map

- We can implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly linked list), in arbitrary order



5

5

The get(k) Algorithm

```
Algorithm get(k):
    B = S.positions() {B is an iterator of the positions in S}
    while B.hasNext() do
        p = B.next()  { the next position in B }
        if p.element().getKey() = k then
            return p.element().getValue()
    return null {there is no entry with key equal to k}
```

6

6

Worst Case O(n)

Questa ricerca non in tempo costante: $O(n)$, va bene ma non è altrettanto

The put(k,v) Algorithm

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
        t = p.element().getValue()
        S.set(p,(k,v))
        return t        {return the old value}
S.addLast((k,v))
n = n + 1       {increment variable storing number of entries}
return null     { there was no entry with key equal to k }
```

7

7

The remove(k) Algorithm

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
        t = p.element().getValue()
        S.remove(p)
        n = n – 1       {decrement number of entries}
        return t        {return the removed value}
return null             {there is no entry with key equal to k}
```

8

8

Anche se non ce ne importasse, abbiamo doppia chiave e le funzioni si scrivono. Ma come sta implementato sopra non supporta copie!

Performance of a List-Based Map

- Performance:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

con implementato: inserisco inizio o fine senza problemi

9

9

Implementation (1)

```
public class UnsortedTableMap_mod<K,V> implements Map<K,V> {

    /** Underlying storage for the map of entries. */
    private ArrayList<UnsortedTableMap_mod.MapEntry<K,V>> table = new ArrayList<>();

    /** Constructs an initially empty map. */
    public UnsortedTableMap_mod() { }

    //---------- nested MapEntry class ----------
    /**
     * A concrete implementation of the Entry interface to be used
     * within a Map implementation.
     */
    protected static class MapEntry<K,V> implements Entry<K,V> {
        private K k;  // key
        private V v;  // value

        public MapEntry(K key, V value) {
            k = key;
            v = value;
        }

        // public methods of the Entry interface
        public K getKey() { return k; }
        public V getValue() { return v; }

        // utilities not exposed as part of the Entry interface
        protected void setKey(K key) { k = key; }
        protected V setValue(V value) {
            V old = v;
            v = value;
            return old;
        }

        /** Returns string representation (for debugging only) */
        public String toString() { return "<" + k + ", " + v + ">"; }
    } //---------- end of nested MapEntry class ----------
```

10

10

Implementation (2)

```
/**
 * Returns the number of entries in the map.
 * @return number of entries in the map
 */
@Override
public int size() { return table.size(); }

/**
 * Tests whether the map is empty.
 * @return true if the map is empty, false otherwise
 */
@Override
public boolean isEmpty() { return size() == 0; }

// Provides support for keySet() and values() methods, based upon
// the entrySet() method that must be provided by subclasses

//---------------- nested KeyIterator class ----------------
private class KeyIterator implements Iterator<K> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator();  // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public K next() { return entries.next().getKey(); }            // return key!
    public void remove() { throw new UnsupportedOperationException("remove not supported"); }
} //----------- end of nested KeyIterator class -----------

//---------------- nested KeyIterable class ----------------
private class KeyIterable implements Iterable<K> {
    public Iterator<K> iterator() { return new UnsortedTableMap_mod.KeyIterator(); }
} //----------- end of nested KeyIterable class -----------

/**
 * Returns an iterable collection of the keys contained in the map.
 *
 * @return iterable collection of the map's keys
 */
@Override
public Iterable<K> keySet() { return new UnsortedTableMap_mod.KeyIterable(); }
```

11

11

Implementation (3)

```
//---------------- nested ValueIterator class ----------------
private class ValueIterator implements Iterator<V> {
    private Iterator<Entry<K,V>> entries = entrySet().iterator();  // reuse entrySet
    public boolean hasNext() { return entries.hasNext(); }
    public V next() { return entries.next().getValue(); }          // return value!
    public void remove() { throw new UnsupportedOperationException("remove not supported"); }
} //----------- end of nested ValueIterator class -----------

//---------------- nested ValueIterable class ----------------
private class ValueIterable implements Iterable<V> {
    public Iterator<V> iterator() { return new UnsortedTableMap_mod.ValueIterator(); }
} //----------- end of nested ValueIterable class -----------

/**
 * Returns an iterable collection of the values contained in the map.
 * Note that the same value will be given multiple times in the result
 * if it is associated with multiple keys.
 *
 * @return iterable collection of the map's values
 */
@Override
public Iterable<V> values() { return new UnsortedTableMap_mod.ValueIterable(); }

// private utility
/** Returns the index of an entry with equal key, or -1 if none found. */
private int findIndex(K key) {
    int n = table.size();
    for (int j=0; j < n; j++)
        if (table.get(j).getKey().equals(key))
            return j;
    return -1;                                  // special value denotes that key was not found
}
```

12

12

Implementation (4)

```
/**
 * Returns the value associated with the specified key, or null if no such entry exists.
 * @param key  the key whose associated value is to be returned
 * @return the associated value, or null if no such entry exists
 */
@Override
public V get(K key) {
    int j = findIndex(key);
    if (j == -1) return null;                    // not found
    return table.get(j).getValue();
}

/**
 * Associates the given value with the given key. If an entry with
 * the key was already in the map, this replaced the previous value
 * with the new one and returns the old value. Otherwise, a new
 * entry is added and null is returned.
 * @param key    key with which the specified value is to be associated
 * @param value  value to be associated with the specified key
 * @return the previous value associated with the key (or null, if no such entry)
 */
@Override
public V put(K key, V value) {
    int j = findIndex(key);
    if (j == -1) {
        table.add(new UnsortedTableMap_mod.MapEntry<>(key, value));      // add new entry
        return null;
    } else                                       // key already exists
        return table.get(j).setValue(value);       // replaced value is returned
}
```

Implementazione in $O(n)$

13

13

Implementation (5)

```
/**
 * Removes the entry with the specified key, if present, and returns its value.
 * Otherwise does nothing and returns null.
 * @param key  the key whose entry is to be removed from the map
 * @return the previous value associated with the removed key, or null if no such entry exists
 */
@Override
public V remove(K key) {
    int j = findIndex(key);
    int n = size();
    if (j == -1) return null;                    // not found
    V answer = table.get(j).getValue();
    if (j != n - 1)
        table.set(j, table.get(n-1));              // relocate last entry to 'hole' created by removal
    table.remove( index: n-1);                     // remove last entry of table
    return answer;
}

//---------------- nested EntryIterator class ----------------
private class EntryIterator implements Iterator<Entry<K,V>> {
    private int j=0;
    public boolean hasNext() { return j < table.size(); }
    public Entry<K,V> next() {
        if (j == table.size()) throw new NoSuchElementException("No further entries");
        return table.get(j++);
    }
    public void remove() { throw new UnsupportedOperationException("remove not supported"); }
} //---------- end of nested EntryIterator class ----------

//---------------- nested EntryIterable class ----------------
private class EntryIterable implements Iterable<Entry<K,V>> {
    public Iterator<Entry<K,V>> iterator() { return new UnsortedTableMap_mod.EntryIterator(); }
} //---------- end of nested EntryIterable class ----------

/**
 * Returns an iterable collection of all key-value entries of the map.
 *
 * @return iterable collection of the map's entries
 */
@Override
public Iterable<Entry<K,V>> entrySet() { return new UnsortedTableMap_mod.EntryIterable(); }
```

14

14

Usage

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello Maps!");

        UnsortedTableMap_mod<String, String> mappaAeroporti = new UnsortedTableMap_mod<String, String>();

        print("isEmpty(): " + mappaAeroporti.isEmpty());
        print("put():" + mappaAeroporti.put("NAP", "Napoli Capodichino"));
        print("isEmpty(): " + mappaAeroporti.isEmpty());
        print("size():" + mappaAeroporti.size());

        print("put():" + mappaAeroporti.put("FCO", "Roma"));
        print("put():" + mappaAeroporti.put("FCO", "Roma Fiumicino"));
        print("size():" + mappaAeroporti.size());

        print ("get(): " + mappaAeroporti.get("NAP"));
        mappaAeroporti.remove( key: "LHR");   // Heathrow: non c'è!

        mappaAeroporti.remove( key: "FCO");   // Fiumicino: non c'è!
        print("size():" + mappaAeroporti.size());

    }

    private static void print(String s) { System.out.println(s); }
}
```

Output

```
"C:\Program Files\Java\jdk1.8.0_162\bin\java.exe" ...
Hello World!
isEmpty(): true
put():null
isEmpty(): false
size():1
put():null
put():Roma
size():2

Process finished with exit code 0
```

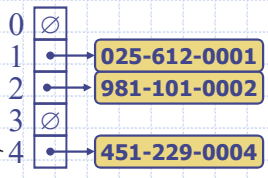
15

15

Hash Tables

Mappe di prima: Prob. della duplicaz. delle chiavi. Devo fare in modo che chiave generata sempre univoca.

Chiave: un dato numerico.

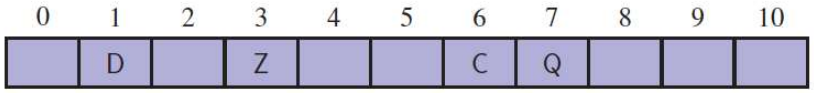


16

16

Intuitive Notion of a Map

- Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].
- As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to N − 1, for some N ≥ n.

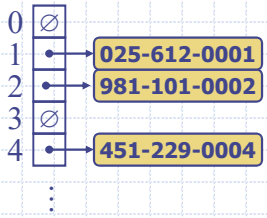


18

18 Mappa è limitata di solito. Le chiavi sono limitate.
I valori sono meno limitati delle chiavi: la funzione di Hash deve assicurare che K ∈ I definito.

More General Kinds of Keys

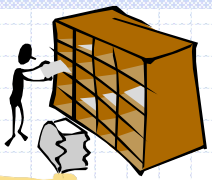
- But what should we do if our keys are not integers in the range from 0 to N − 1?
  - Use a **hash function** to map general keys to corresponding indices in a table.
  - For instance, the last four digits of a Social Security number.



19

19 Nota: Valore chiave non è detto che sia numerico

Hash Functions and Hash Tables



- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:
  $h(x) = x \bmod N$
  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

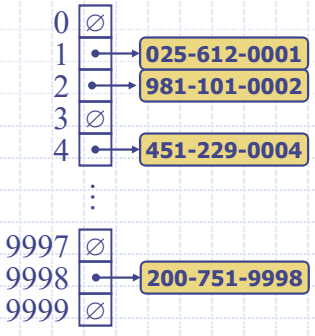
20

20

Perché array? Perché è posizionale.
↳ Oggetto inserito nell'indice sulla base di h(x)

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x) =$ last four digits of x



21

21

Perché voglio che elementi siano più lontani possibile l'uno dall'altro, in modo da abbassare la prob.

Hash Functions

Non è funzione singola

- A hash function is usually specified as the composition of two functions:
  - Hash code: da chiave genera intero
    h_1: keys $\rightarrow$ integers
  - Compression function:
    h_2: integers $\rightarrow [0, N-1]$
- The hash code is applied first, and the compression function is applied next on the result, i.e.,
  $h(x) = h_2(h_1(x))$
- The goal of the hash function is to "disperse" the keys in an apparently random way

22

22

un intervallo non prestabilito
fa lo riporto nell'intervallo specificato

PRE COMPRESSION Hash Codes

Funziona bene per oggetti, ma se ho passo oggetto numerico è piccolo in bit, ma locato in memoria è un numero enorme

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

Utile se oggetto è piccolo: numero di bit che lo rappresentano sono limitati.

- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

1001 0110 0111
+ + =key

Ottimo quando chiave ha lunghezza maggiore dell'intero

23

23

Ci va bene perché comunque non è mai troppo alto: O(33)!

Hash Codes (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $a_0 a_1 \dots a_{n-1}$
  - We evaluate the polynomial
    $p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $n = 33$ gives at most 6 collisions on a set of 50,000 English words)
- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $p_0(z) = a_{n-1}$
    $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$
    $(i = 1, 2, \dots, n-1)$
- We have $p(z) = p_{n-1}(z)$

grado della potenza

24

24

Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

    ↳ Minor numero di collisioni possibili.
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

25

25

Abstract Hash Map in Java

```
1  public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
2    protected int n = 0;                    // number of entries in the dictionary
3    protected int capacity;                 // length of the table
4    private int prime;                      // prime factor
5    private long scale, shift;              // the shift and scaling factors
6    public AbstractHashMap(int cap, int p) {
7      prime = p;
8      capacity = cap;
9      Random rand = new Random();
10     scale = rand.nextInt(prime–1) + 1;
11     shift = rand.nextInt(prime);
12     createTable();
13   }
14   public AbstractHashMap(int cap) { this(cap, 109345121); }  // default prime
15   public AbstractHashMap() { this(17); }                     // default capacity
16   // public methods
17   public int size() { return n; }
18   public V get(K key) { return bucketGet(hashValue(key), key); }
19   public V remove(K key) { return bucketRemove(hashValue(key), key); }
20   public V put(K key, V value) {
21     V answer = bucketPut(hashValue(key), key, value);
22     if (n > capacity / 2)                 // keep load factor <= 0.5
23       resize(2 * capacity – 1);           // (or find a nearby prime)
24     return answer;
25   }
```

Servono per calcolo dirind

↳ Nel caso ci scordiamo valore primo

Secchio: presuppone che ogg. numero è mod lista.

↳ Se n raggiunge metà capacità, raddoppio la dim. della tabella.

26

26

FATTORE DI CARICO = $\frac{\text{Maggy}}{\text{Ampiezza array}}$. Se $F_c < 0.5$, numero di collisioni è molto basso.

altrimenti sì vola.

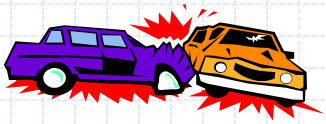
Abstract Hash Map in Java, 2

```
26   // private utilities
27   private int hashValue(K key) {
28     return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
29   }
30   private void resize(int newCap) {
31     ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
32     for (Entry<K,V> e : entrySet())
33       buffer.add(e);
34     capacity = newCap;
35     createTable();                        // based on updated capacity
36     n = 0;                                // will be recomputed while reinserting entries
37     for (Entry<K,V> e : buffer)
38       put(e.getKey(), e.getValue());
39   }
40   // protected abstract methods to be implemented by subclasses
41   protected abstract void createTable();
42   protected abstract V bucketGet(int h, K k);
43   protected abstract V bucketPut(int h, K k, V v);
44   protected abstract V bucketRemove(int h, K k);
45 }
```

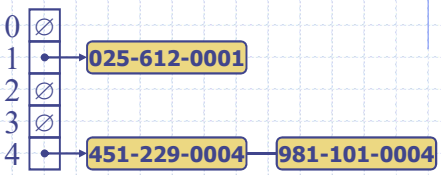
27

27

Collision Handling



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there



Metodo più EZ!

DAMN DANIEL!

- Separate chaining is simple, but requires additional memory outside the table

28

28

↳ INSERIMENTO RIMANE UNITARIO → La get deve controllare se chiave c'è già però!

RICERCA É LINEARE

↳ (Unlink + Scorrere la lista delle entries)

Map with Separate Chaining

Put sostituisce valore già presente?

Delegate operations to a list-based map at each cell:

```
Algorithm get(k):
return A[h(k)].get(k)

Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = null then
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ null then
    n = n - 1
return t
```

Mi trovo lui i

k nuova chiave {k is a new key}

→ Size

→ La metto nell'array

{k was found}

29

29 Nota: Se non ho mai usato posto, non rinizializzo linked list.

Hash Table with Chaining

```
1  public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
2    // a fixed capacity array of UnsortedTableMap that serve as buckets
3    private UnsortedTableMap<K,V>[ ] table;   // initialized within createTable
4    public ChainHashMap( ) { super( ); }
5    public ChainHashMap(int cap) { super(cap); }
6    public ChainHashMap(int cap, int p) { super(cap, p); }
7    /** Creates an empty table having length equal to current capacity. */
8    protected void createTable( ) {
9      table = (UnsortedTableMap<K,V>[ ]) new UnsortedTableMap[capacity];
10   }
11   /** Returns value associated with key k in bucket with hash value h, or else null. */
12   protected V bucketGet(int h, K k) {
13     UnsortedTableMap<K,V> bucket = table[h];
14     if (bucket == null) return null;
15     return bucket.get(k);
16   }
17   /** Associates key k with value v in bucket with hash value h; returns old value. */
18   protected V bucketPut(int h, K k, V v) {
19     UnsortedTableMap<K,V> bucket = table[h];
20     if (bucket == null)
21       bucket = table[h] = new UnsortedTableMap<>( );
22     int oldSize = bucket.size( );
23     V answer = bucket.put(k,v);
24     n += (bucket.size( ) – oldSize);     // size may have increased
25     return answer;
26   }
```

Controllo sempre se è nullo o no

Numero di bucket = numero di chiavi

30

30

Hash Table with Chaining, 2

```
27   /** Removes entry having key k from bucket with hash value h (if any). */
28   protected V bucketRemove(int h, K k) {
29     UnsortedTableMap<K,V> bucket = table[h];
30     if (bucket == null) return null;
31     int oldSize = bucket.size( );
32     V answer = bucket.remove(k);
33     n –= (oldSize – bucket.size( ));     // size may have decreased
34     return answer;
35   }
36   /** Returns an iterable collection of all key-value entries of the map. */
37   public Iterable<Entry<K,V>> entrySet( ) {
38     ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
39     for (int h=0; h < capacity; h++)
40       if (table[h] != null)
41         for (Entry<K,V> entry : table[h].entrySet( ))
42           buffer.add(entry);
43     return buffer;
44   }
45 }
```

Tutti i bucket e li metto nell'iterabile

31

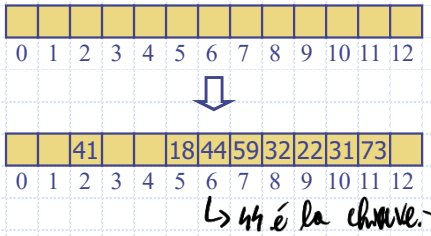
31

Ho spazio tra una chiave e l'altra → Trovo collisione → Salvo nella casella successiva libera.

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



↳ 44 è la chiave.

Se dovesse trovare a pos. 5 e non c'è, vaa avanti.

32

32

Ho uno spazio molto occupato. Ma per evitare numero spazio di collisione.

Search with Linear Probing

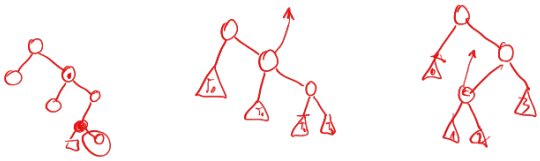
- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Perché N?

```
Algorithm get(k)
  i ← h(k)
  p ← 0
  repeat
    c ← A[i]
    if c = ∅
      return null
    else if c.getKey() = k
      return c.getValue()
    else
      i ← (i + 1) mod N
      p ← p + 1
  until p = N
  return null
```

33

33



Se faccio put con chiave già inserita sostituisce? E se sì, ogni volta che si sposta dovrebbe controllare ogni valore se è da sostituire

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called *DEFUNCT*, which replaces deleted elements
- remove(k)
  - We search for an entry with key k
  - If such an entry (k, o) is found, we replace it with the special item *DEFUNCT* and we return element o
  - Else, we return *null*
- put(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores *DEFUNCT*, or
    - N cells have been unsuccessfully probed
  - We store (k, o) in cell i

34

34

dunque problema con linear probing

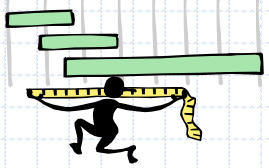
Perché non sta scritto che deve controllare anche se trova quello giusto?

Cerco K: STOP! E ORA?

PEROTTA: ORDINAMENTO

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1 / (1 - \alpha)$



- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

40

Impossibile

Nota: se aggiungi comunque alla vecchia chiave non ci arrivi mai.

40 Se valore occupa indice di un'altra chiave => quindi devo inserire in quella posiz. vuota avanti.