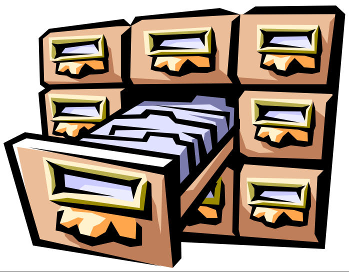


Maps

1

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
  - A university's information system relies on some form of a student ID as a key that is mapped to that student's associated record (such as the student's name, address, and course grades) serving as the value.
  - The domain-name system (DNS)maps a host name, such as www.wiley.com, to an Internet-Protocol (IP) address, such as 208.215.179.146

2

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

3

java.util.map Example

```java
// Java program to demonstrate
// the working of Map interface

import java.util.*;
class HashMapDemo {
    public static void main(String args[])
    {
        Map<String, Integer> hm
            = new HashMap<String, Integer>();

        hm.put("a", new Integer(100));
        hm.put("b", new Integer(200));
        hm.put("c", new Integer(300));
        hm.put("d", new Integer(400));

        // Traversing through the map
        for (Map.Entry<String, Integer> me : hm.entrySet()) {
            System.out.print(me.getKey() + ":");
            System.out.println(me.getValue());
        }
    }
}
```

4

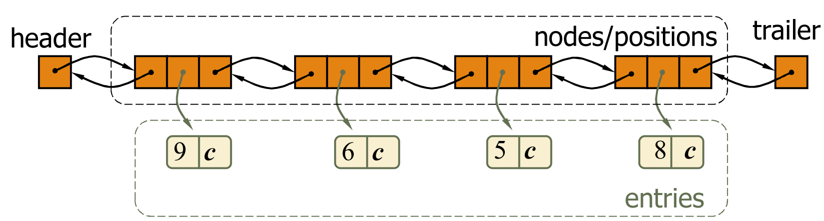
Example

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*)

5

A Simple List-Based Unsorted Map

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



6

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
```

7

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
```

8

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
```

9

Performance of a List-Based Map

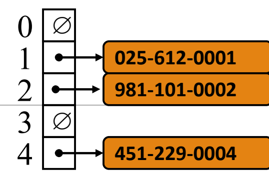
Performance:

- put **may** take $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence. Previous implementation takes $O(n)$ time.
- 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)

10

Hash Tables

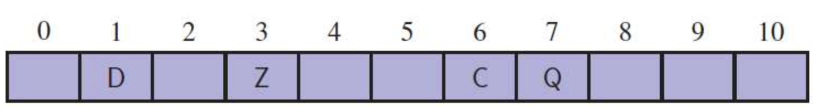


11

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**.

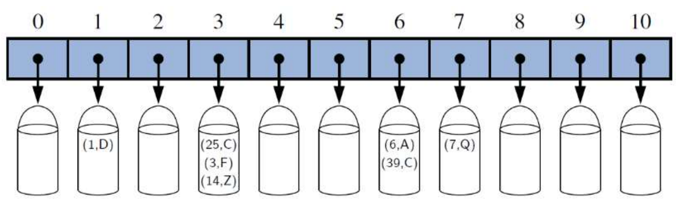


12

Limitations

There are two challenges in extending this framework to the more general setting of a map.

- First, not wish to devote an array of length N if it is the case that $N \gg n$.
- Second, in general it is not required that a map's keys be integers.
- Would like to be able to store more than one entry in one map. (Bucket array)



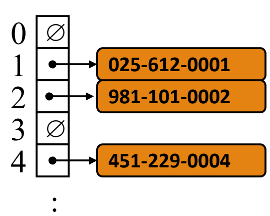
A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

13

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.



14

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 $i = h(k)$

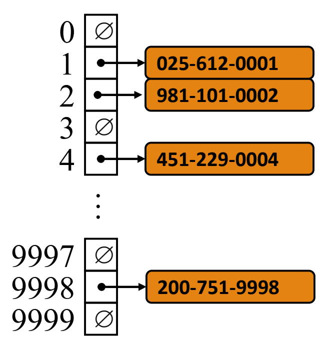
15

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



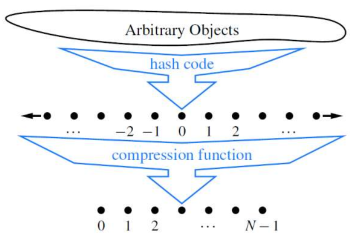
16

Hash Functions

A hash function is usually specified as the composition of two functions:

Hash code:
h_1: keys $\rightarrow$ integers

Compression function:
h_2: integers $\rightarrow [0, N-1]$



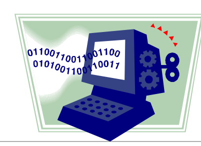
The hash code is applied first, and the compression function is applied next on the result, i.e.,

$$h(x) = h_2(h_1(x))$$

The goal of the hash function is to "disperse" the keys in an apparently random way

17

Hash Codes



- 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)

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)

18

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)
  $$a_0\, a_1 \,\ldots\, a_{n-1}$$
- We evaluate the polynomial
  $$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots$$
  $$\ldots + a_{n-1} z^{n-1}$$
  at a fixed value z, ignoring overflows
- Especially suitable for strings (e.g., the choice $z = 33$ 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
  $$p_0(z) = a_{n-1}$$
  $$p_i(z) = a_{n-i-1} + z p_{i-1}(z)$$
  $$(i = 1, 2, \ldots, n-1)$$

We have $p(z) = p_{n-1}(z)$

19

Hash Codes (cont.)

Cyclic-Shift hash codes:

- A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits.
- For example, a 5-bit cyclic shift of the 32-bit value 00111101100101101010100010101000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 10110010110101010001010100000111

```
static int hashCode(String s)
{
int h=0;
for (int i=0; i<s.length( ); i++) {
   h = (h << 5) | (h >>> 27);
  // 5-bit cyclic shift of the running sum
   h += (int) s.charAt(i);
// add in next character
}
return h;
}
```

Shift	Collisions Total	Collisions Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

20

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

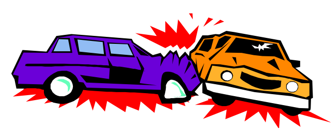
Multiply, Add and Divide (MAD):

- $h_2(y) = (ay + b) \bmod N$
- a and b are nonnegative integers such that

  $a \bmod N \neq 0$
- Otherwise, every integer would map to the same value b

If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is 1/N.

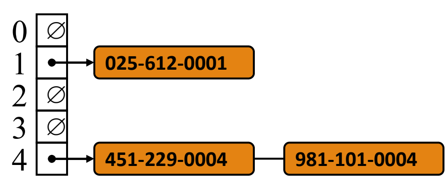
21



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



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

Assuming we use a good hash function to index the n entries of our map in a bucket array of capacity N, the expected size of a bucket is n/N. The ratio lamda = n/N, called the load factor of the hash table, should be bounded by a small constant, preferably below 1. As long as lamda is O(1), the core operations on the hash table run in O(1) expected time.

22

Open Addressing

The separate chaining rule requires the use of an auxiliary data structure to hold entries with colliding keys.

If space is at a premium (for example, if we are writing a program for a small handheld device), then we can use the alternative approach of storing each entry directly in a table slot.

This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to properly handle collisions.

There are several variants of this approach, collectively referred to as open addressing schemes.

Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself.

26

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

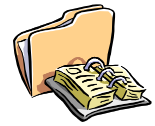
Example:
- $h(x) = x \bmod 13$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

27



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

Algorithm $get(k)$
  $i \leftarrow h(k)$
  $p \leftarrow 0$
  repeat
    $c \leftarrow A[i]$
    if $c = \varnothing$
      return $null$
    else if $c.getKey() = k$
      return $c.getValue()$
    else
      $i \leftarrow (i + 1) \bmod N$
      $p \leftarrow p + 1$
  until $p = N$
  return $null$

28

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

29

```
public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
  private MapEntry<K,V>[ ] table;        // a fixed array of entries (all initially null)
  private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);   //sentinel
  public ProbeHashMap( ) { super( ); }
  public ProbeHashMap(int cap) { super(cap); }
  public ProbeHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable( ) {
    table = (MapEntry<K,V>[ ]) new MapEntry[capacity];   // safe cast
  }
  /** Returns true if location is either empty or the "defunct" sentinel. */
  private boolean isAvailable(int j) {
    return (table[j] == null || table[j] == DEFUNCT);
  }
```

Probe Hash Map in Java

30

```
  /** Returns index with key k, or −(a+1) such that k could be added at index a. */
  private int findSlot(int h, K k) {
    int avail = −1;                          // no slot available (thus far)
    int j = h;                               // index while scanning table
    do {
      if (isAvailable(j)) {                  // may be either empty or defunct
        if (avail == −1) avail = j;          // this is the first available slot!
        if (table[j] == null) break;         // if empty, search fails immediately
      } else if (table[j].getKey( ).equals(k))
        return j;                            // successful match
      j = (j+1) % capacity;                  // keep looking (cyclically)
    } while (j != h);                        // stop if we return to the start
    return −(avail + 1);                     // search has failed
  }
  /** Returns value associated with key k in bucket with hash value h, or else null. */
  protected V bucketGet(int h, K k) {
    int j = findSlot(h, k);
    if (j < 0) return null;                  // no match found
    return table[j].getValue( );
  }
```

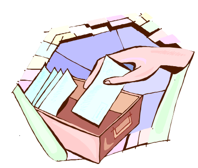
Probe Hash Map in Java, 2

31

```java
/** Associates key k with value v in bucket with hash value h; returns old value. */
protected V bucketPut(int h, K k, V v) {
  int j = findSlot(h, k);
  if (j >= 0)                                   // this key has an existing entry
    return table[j].setValue(v);
  table[−(j+1)] = new MapEntry<>(k, v);         // convert to proper index
  n++;
  return null;
}
/** Removes entry having key k from bucket with hash value h (if any). */
protected V bucketRemove(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                       // nothing to remove
  V answer = table[j].getValue( );
  table[j] = DEFUNCT;                           // mark this slot as deactivated
  n−−;
  return answer;
}
/** Returns an iterable collection of all key-value entries of the map. */
public Iterable<Entry<K,V>> entrySet( ) {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>( );
  for (int h=0; h < capacity; h++)
    if (!isAvailable(h)) buffer.add(table[h]);
  return buffer;
}
}
```

Probe Hash Map in Java, 3

32



Double Hashing

Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + jd(k)) \bmod N$$

for $j = 0, 1, \dots, N-1$

The secondary hash function $d(k)$ cannot have zero values

The table size N must be a prime to allow probing of all the cells

Common choice of compression function for the secondary hash function:

$$d_2(k) = q - k \bmod q$$

where

- $q < N$
- q is a prime

The possible values for $d_2(k)$ are

$$1, 2, \dots, q$$

33

Example of Double Hashing

Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$
- **A[(h(k)+ i*d (k)) mod N] next, for i = 1,2,3, . .**

Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

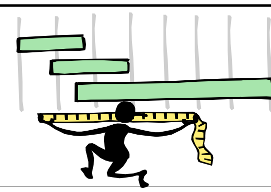
0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

34

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

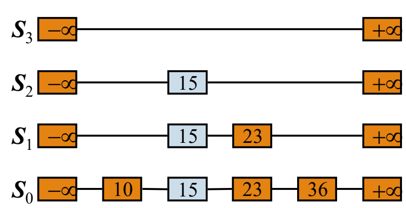
35

Ex: Counting Word Frequencies

```
/** A program that counts words in a document, printing the most frequent. */
public class WordCount {
    public static void main(String[] args) {
        Map<String,Integer> freq = new ChainHashMap<>( ); // or any concrete
        map
        // scan input for words, using all nonletters as delimiters
        Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
        while (doc.hasNext( )) {
            String word = doc.next( ).toLowerCase( );//convert next word to
            lowercase
            Integer count = freq.get(word); //get the previous count for
            this word
            if (count == null)
                count = 0; // if not in map, previous count is zero
            freq.put(word, 1 + count); // (re)assign new count for this word
        }
        int maxCount = 0;
        String maxWord = "no word";
        for (Entry<String,Integer> ent : freq.entrySet( )) // find max-count
        word
            if (ent.getValue( ) > maxCount) {
                maxWord = ent.getKey( );
                maxCount = ent.getValue( );
            }
        System.out.print("The most frequent word is '" + maxWord);
        System.out.println("' with " + maxCount + " occurrences.");
    }
}
```

38

Skip Lists



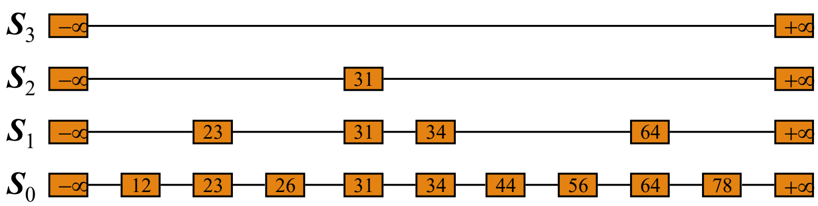
39

What is a Skip List

A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that

- Each list S_i contains the special keys $+\infty$ and $-\infty$
- List S_0 contains the keys of S in nondecreasing order
- Each list is a subsequence of the previous one, i.e.,
  $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
- List S_h contains only the two special keys

We show how to use a skip list to implement the map ADT



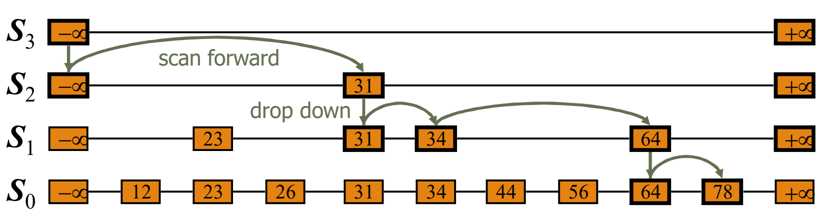
40

Search

We search for a key x in a a skip list as follows:

- We start at the first position of the top list
- At the current position p, we compare x with $y \leftarrow key(next(p))$
  - $x = y$: we return $element(next(p))$
  - $x > y$: we "scan forward"
  - $x < y$: we "drop down"
- If we try to drop down past the bottom list, we return ***null***

Example: search for 78



41

Randomized Algorithms

A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution

It contains statements of the type

$b \leftarrow random()$

if $b = 0$

    do A …

else $\{ b = 1\}$

    do B …

Its running time depends on the outcomes of the coin tosses

We analyze the expected running time of a randomized algorithm under the following assumptions

- the coins are unbiased, and
- the coin tosses are independent

The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads")

We use a randomized algorithm to insert items into a skip list

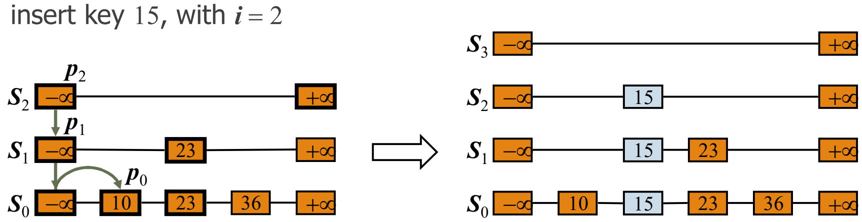
42

Insertion

To insert an entry (x, o) into a skip list, we use a randomized algorithm:

- We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
- If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
- We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
- For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j

Example: insert key 15, with $i = 2$



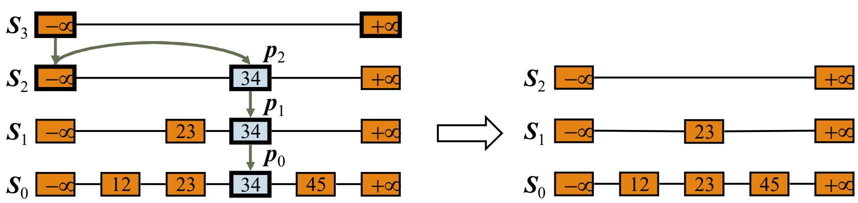
43

Deletion

To remove an entry with key x from a skip list, we proceed as follows:

- We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
- We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
- We remove all but one list containing only the two special keys

Example: remove key 34



44

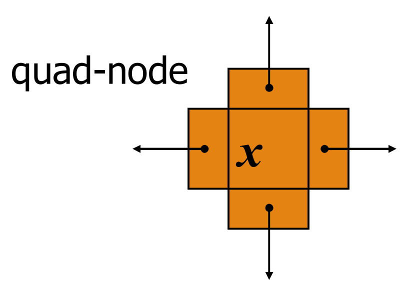
Implementation

We can implement a skip list with quad-nodes

A quad-node stores:

- entry
- link to the node prev
- link to the node next
- link to the node below
- link to the node above

Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them



45

Space Usage

The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm

We use the following two basic probabilistic facts:

- Fact 1: The probability of getting i consecutive heads when flipping a coin is $1/2^i$
- Fact 2: If each of n entries is present in a set with probability p, the expected size of the set is np

Consider a skip list with n entries

- By Fact 1, we insert an entry in list S_i with probability $1/2^i$
- By Fact 2, the expected size of list S_i is $n/2^i$

The expected number of nodes used by the skip list is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- Thus, the expected space usage of a skip list with n items is $O(n)$

46

Search and Update Times

The search time in a skip list is proportional to

- the number of drop-down steps, plus
- the number of scan-forward steps

The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability

To analyze the scan-forward steps, we use yet another probabilistic fact:

- Fact 4: The expected number of coin tosses required in order to get tails is 2

When we scan forward in a list, the destination key does not belong to a higher list

- A scan-forward step is associated with a former coin toss that gave tails

By Fact 4, in each list the expected number of scan-forward steps is 2

Thus, the expected number of scan-forward steps is $O(\log n)$

We conclude that a search in a skip list takes $O(\log n)$ expected time

The analysis of insertion and deletion gives similar results

48

Summary

A skip list is a data structure for maps that uses a randomized insertion algorithm

In a skip list with n entries
- The expected space used is $O(n)$
- The expected search, insertion and deletion time is $O(\log n)$

Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability

Skip lists are fast and simple to implement in practice

49

Multisets and Multimaps

50

Definitions

A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.

- Elements of a set are like keys of a map, but without any auxiliary values.

A **multiset** (also known as a **bag**) is a set-like container that allows duplicates.

A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values.

- For example, the index of a book maps a given term to one or more locations at which the term occurs.

51

add(e): Adds the element e to S (if not already present).

remove(e): Removes the element e from S (if it is present).

contains(e): Returns whether e is an element of S.

iterator( ): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$\begin{aligned} S \cup T &= \{e\colon\ e \text{ is in } S \text{ or } e \text{ is in } T\}, \\ S \cap T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is in } T\}, \\ S - T &= \{e\colon\ e \text{ is in } S \text{ and } e \text{ is not in } T\}. \end{aligned}$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.

retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.

removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Set ADT

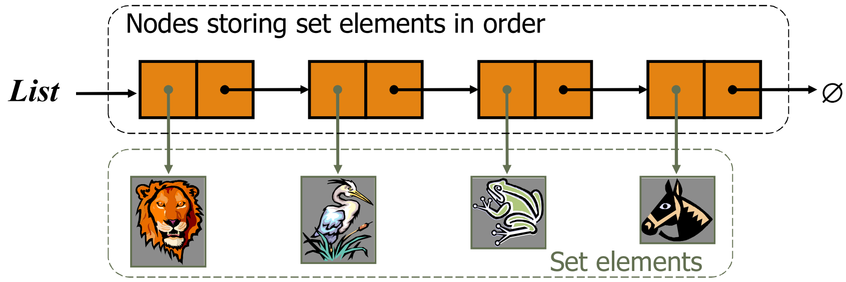
52

Storing a Set in a List

We can implement a set with a list

Elements are stored sorted according to some canonical ordering

The space used is $O(n)$



53

Generic Merging

Generalized merge of two sorted lists A and B

Template method genericMerge

Auxiliary methods
- aIsLess
- bIsLess
- bothAreEqual

Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

Algorithm *genericMerge*(A, B)
    $S \leftarrow$ empty sequence
    while $\neg A.isEmpty() \land \neg B.isEmpty()$
        $a \leftarrow A.first().element()$; $b \leftarrow B.first().element()$
        if $a < b$
            $aIsLess(a, S)$; $A.remove(A.first())$
        else if $b < a$
            $bIsLess(b, S)$; $B.remove(B.first())$
        else $\{ b = a \}$
            $bothAreEqual(a, b, S)$
            $A.remove(A.first())$; $B.remove(B.first())$
    while $\neg A.isEmpty()$
        $aIsLess(a, S)$; $A.remove(A.first())$
    while $\neg B.isEmpty()$
        $bIsLess(b, S)$; $B.remove(B.first())$
    return S

54

Using Generic Merge for Set Operations

Any of the set operations can be implemented using a generic merge

For example:

- For intersection: only copy elements that are duplicated in both list
- For union: copy every element from both lists except for the duplicates

All methods run in linear time

55

Multimap

A **multimap** is similar to a map, except that it can store multiple entries with the same key

We can implement a multimap M by means of a map M’

- For every key k in M, let E(k) be the list of entries of M with key k
- The entries of M’ are the pairs (k, E(k))

56

get(k): Returns a collection of all values associated with key k in the multimap.

put(k, v): Adds a new entry to the multimap associating key k with value v, without overwriting any existing mappings for key k.

remove(k, v): Removes an entry mapping key k to value v from the multimap (if one exists).

removeAll(k): Removes all entries having key equal to k from the multimap.

size( ): Returns the number of entries of the multiset (including multiple associations).

entries( ): Returns a collection of all entries in the multimap.

keys( ): Returns a collection of keys for all entries in the multimap (including duplicates for keys with multiple bindings).

keySet( ): Returns a nonduplicative collection of keys in the multimap.

values( ): Returns a collection of values for all entries in the multimap.

Multimaps

57

```java
public class HashMultimap<K,V> {
  Map<K,List<V>> map = new HashMap<>();   // the primary map
  int total = 0;                           // total number of entries in the multimap
  /** Constructs an empty multimap. */
  public HashMultimap() { }
  /** Returns the total number of entries in the multimap. */
  public int size() { return total; }
  /** Returns whether the multimap is empty. */
  public boolean isEmpty() { return (total == 0); }
  /** Returns a (possibly empty) iteration of all values associated with the key. */
  Iterable<V> get(K key) {
    List<V> secondary = map.get(key);
    if (secondary != null)
      return secondary;
    return new ArrayList<>();              // return an empty list of values
  }
```

Java Implementation

58

```java
/** Adds a new entry associating key with value. */
void put(K key, V value) {
  List<V> secondary = map.get(key);
  if (secondary == null) {
    secondary = new ArrayList<>();
    map.put(key, secondary);       // begin using new list as secondary structure
  }
  secondary.add(value);
  total++;
}
/** Removes the (key,value) entry, if it exists. */
boolean remove(K key, V value) {
  boolean wasRemoved = false;
  List<V> secondary = map.get(key);
  if (secondary != null) {
    wasRemoved = secondary.remove(value);
    if (wasRemoved) {
      total−−;
      if (secondary.isEmpty())
        map.remove(key);         // remove secondary structure from primary map
    }
  }
  return wasRemoved;
}
```

Java Implementation, 2

59

```java
  /** Removes all entries with the given key. */
  Iterable<V> removeAll(K key) {
    List<V> secondary = map.get(key);
    if (secondary != null) {
      total −= secondary.size();
      map.remove(key);
    } else
      secondary = new ArrayList<>();       // return empty list of removed values
    return secondary;
  }
  /** Returns an iteration of all entries in the multimap. */
  Iterable<Map.Entry<K,V>> entries() {
    List<Map.Entry<K,V>> result = new ArrayList<>();
    for (Map.Entry<K,List<V>> secondary : map.entrySet()) {
      K key = secondary.getKey();
      for (V value : secondary.getValue())
        result.add(new AbstractMap.SimpleEntry<K,V>(key,value));
    }
    return result;
  }
}
```

Java Implementation, 3

60