Data Structures and Algorithms

Maps and Dictionaries

Outline

- Maps
- Hash tables
- Dictionaries
- Skip Lists

Maps

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

The Map ADT

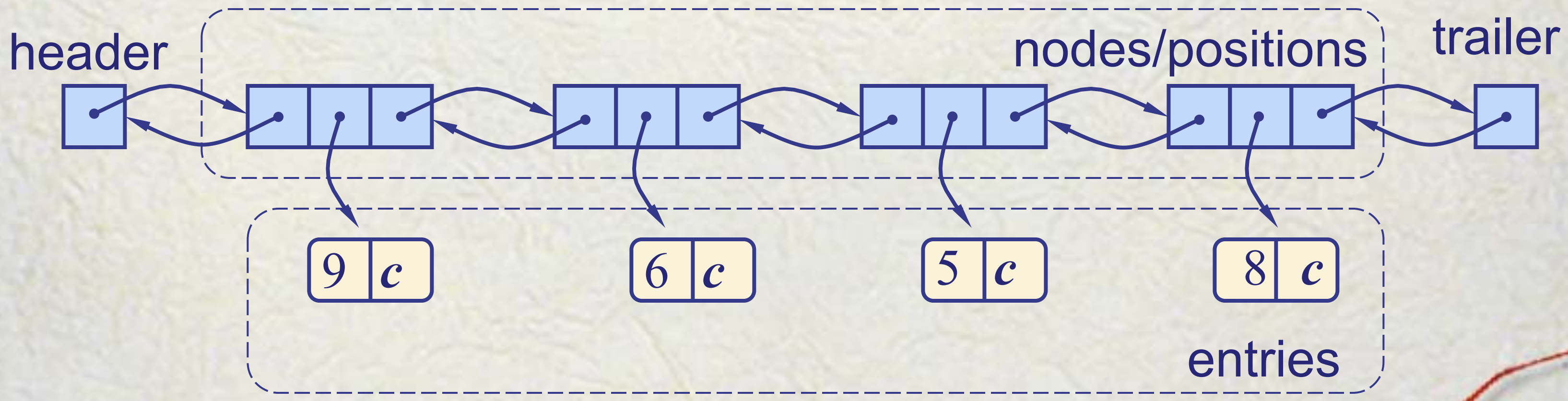
- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with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

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*)

A Simple List-Based Map

- We can efficiently 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



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().key() = $k$   then
            return $p$.element().value()
    return null {there is no entry with key equal to $k$}
```

The put(k,v) Algorithm

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

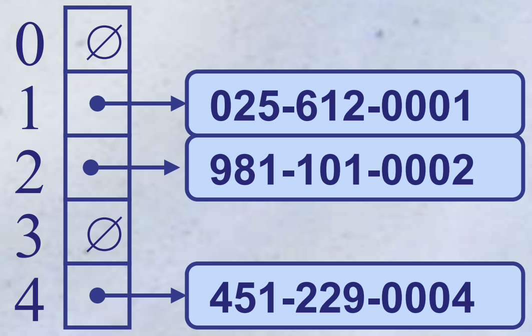
The remove(k) Algorithm

```
**Algorithm** remove($k$):
$B$ =$S$.positions()
while $B$.hasNext() do
    $p$ = $B$.next()
    if $p$.element().key() = $k$ then
        $t$ = $p$.element().value()
        $S$.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 $k$}
```

Performance of a List-Based Map

- Performance:
  - put takes $O(n)$ time since we have to search the sequence to check if the given key exists ($O(1)$ if keys are always unique) .
  - 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 with unique keys (known beforehand and simplified put method), while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Tables

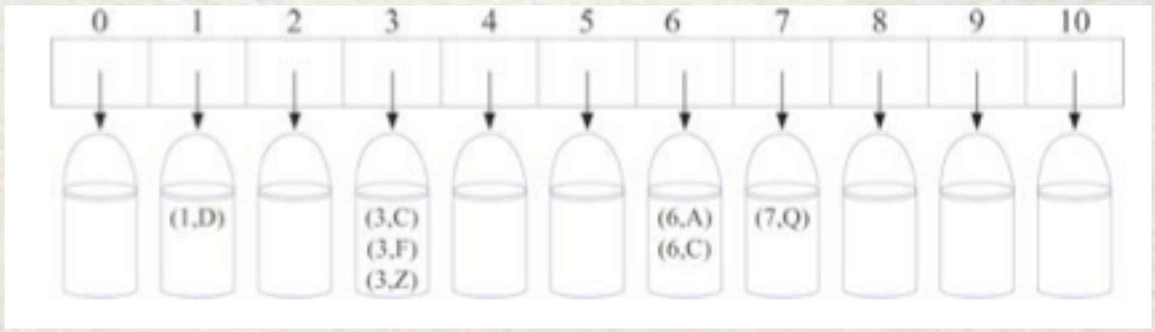


Recall the Map ADT

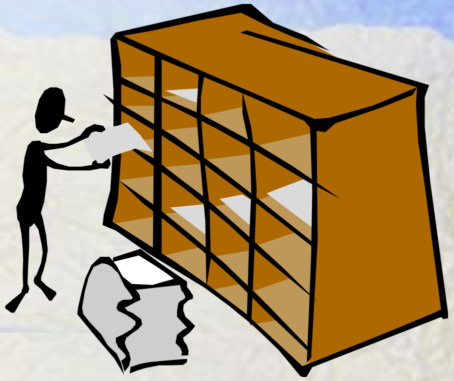
- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its associated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with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

Hash table

- Expected time: O(1)
- Bucket array
- Hash function



Hash Functions and Hash Tables



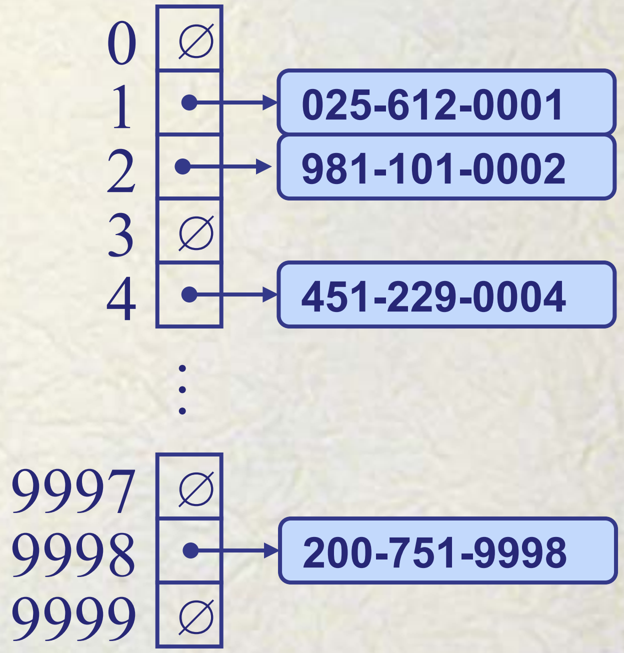
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

  $h(x) = x \bmod N$

  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 $i = h(k)$

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 $\boldsymbol{h}(\boldsymbol{x})$ = last four digits of $\boldsymbol{x}$



Drawbacks

- Space is proportional to N:
  - Waste of space if N >> n
- Keys are required to be integers in the range [0, N-1] -> need “good” hashing function:
  - Minimize collision
  - Fast and easy to compute

Hash Functions

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

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

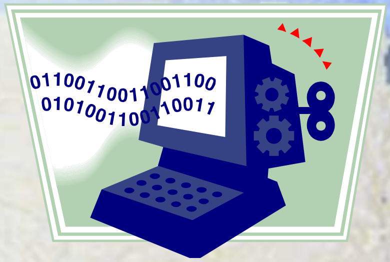
  h_2: integers $\rightarrow$ $[0, N-1]$

- 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

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 (same key should have the same hash code)
- 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)

Hash Codes (cont.)

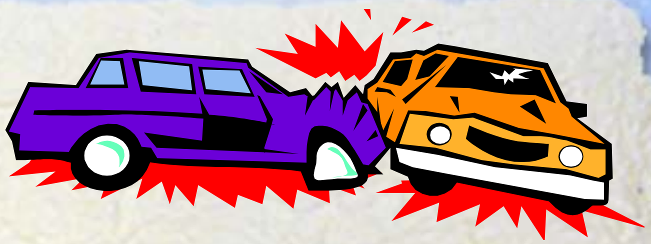
- Polynomial accumulation:
  - Order is important
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $$a_0\, a_1\, \ldots\, a_{n-1}$$
  - We evaluate the polynomial
    $$p(z) = a_{n-1} + a_{n-2}z + a_{n-3}z^2 + \ldots$$
    $$\ldots + a_0 z^{n-1}$$
    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
    $$p_0(z) = a_{n-1}$$
    $$p_i(z) = a_{n-i-1} + zp_{i-1}(z)$$
    $$(i = 1, 2, \ldots, n-1)$$
- We have $p(z) = p_{n-1}(z)$

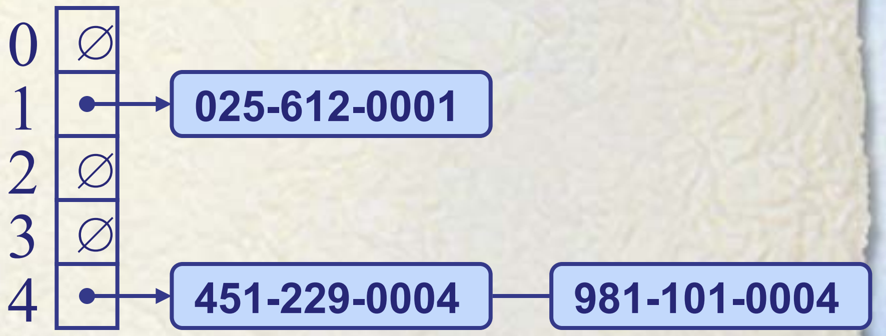
Compression Functions

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - {200, 205, 210, 215,.., 600}: 6 collisions with N=100, No collision with N=101
  - Not enough with repeated patterns of hash codes pN+q for different values of p
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - N is prime, a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

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
- Load factor: n/N < 1



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

Map Methods with Separate Chaining used for Collisions

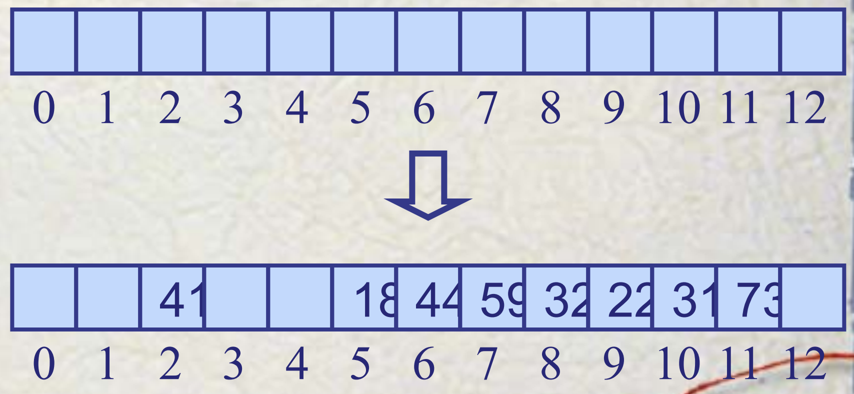
- Delegate operations to a list-based map at each cell:

```
Algorithm get(k):
Output: The value associated with the key k in the map, or null if there is no
    entry with key equal to k in the map
return A[h(k)].get(k)     {delegate the get to the list-based map at A[h(k)]}

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we
    return its value (replacing it with v); otherwise, we return null
t = A[h(k)].put(k,v)      {delegate the put to the list-based map at A[h(k)]}
if t = null then                          {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or null if there
    is no entry with key equal to k in the map
t = A[h(k)].remove(k)     {delegate the remove to the list-based map at A[h(k)]}
if t ≠ null then                     {k was found}
    n = n - 1
return t
```

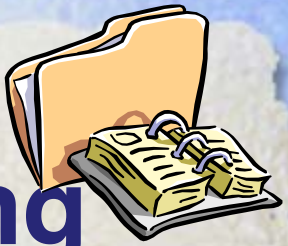
Linear Probing

- 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



Search with Linear Probing



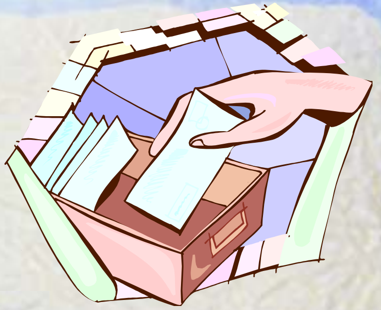
- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

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

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called *AVAILABLE*, 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 *AVAILABLE* 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 *AVAILABLE*, or
    - N cells have been unsuccessfully probed
  - We store entry (k, o) in cell i

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 $j = 0, 1, \ldots, N - 1$
- 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:

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

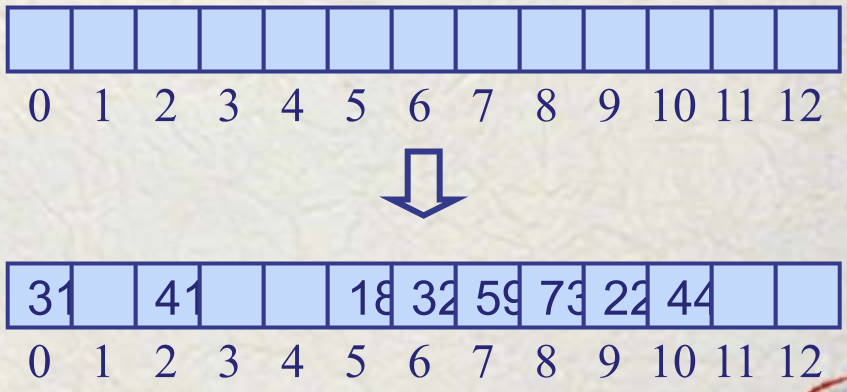
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are

  $1, 2, \ldots, q$

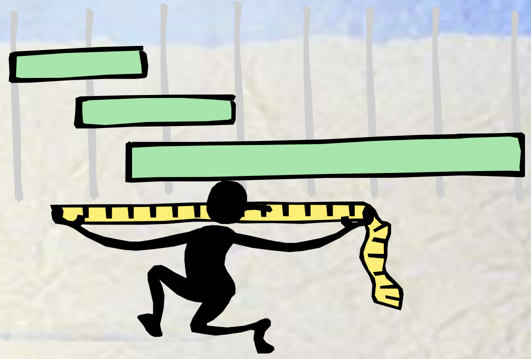
Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- 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		



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

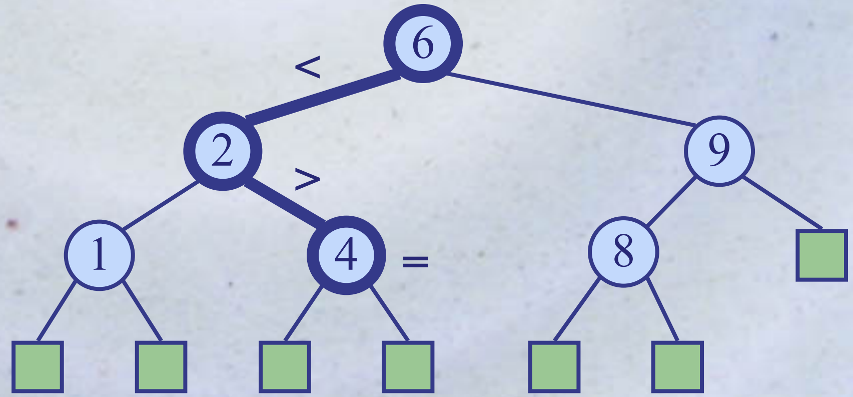
  $1 / (1 - \alpha)$

- 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
- Open addressing is not faster than chaining method if space is an issue.

Example

- Counting Word Frequencies.

Dictionaries



Dictionary ADT

- The dictionary ADT models a searchable collection of key-element entries: ordered and unordered.
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key **are** allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

- Dictionary ADT methods:
  - find(k): if the dictionary has an entry with key k, returns it, else, returns null
  - findAll(k): returns an iterator of all entries with key k
  - insert(k, o): inserts and returns the entry (k, o)
  - remove(e): remove the entry e from the dictionary
  - entries(): returns an iterator of the entries in the dictionary
  - size(), isEmpty()

Example

Operation	***Output***	***Dictionary***
insert(5,*A*)	(5,*A*)	(5,*A*)
insert(7,*B*)	(7,*B*)	(5,*A*),(7,*B*)
insert(2,*C*)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*)
insert(8,*D*)	(8,*D*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
insert(2,*E*)	(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(7)	(7,*B*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(4)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(2)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
findAll(2)	(2,*C*),(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
size()	5	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
remove(find(5))	(5,*A*)	(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(5)	**null**	(7,*B*),(2,*C*),(8,*D*),(2,*E*)

A List-Based Dictionary

- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- Performance:
  - insert takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - find 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 log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

The findAll(k) Algorithm

```
**Algorithm** findAll(k):
Input: A key k
Output: An iterator of entries with key equal to k

Create an initially-empty list L
B = D.entries()
while B.hasNext() do
    e = B.next()
    if e.key() = k then
        L.insertLast(e)
return L.elements()
```

The insert and remove Methods

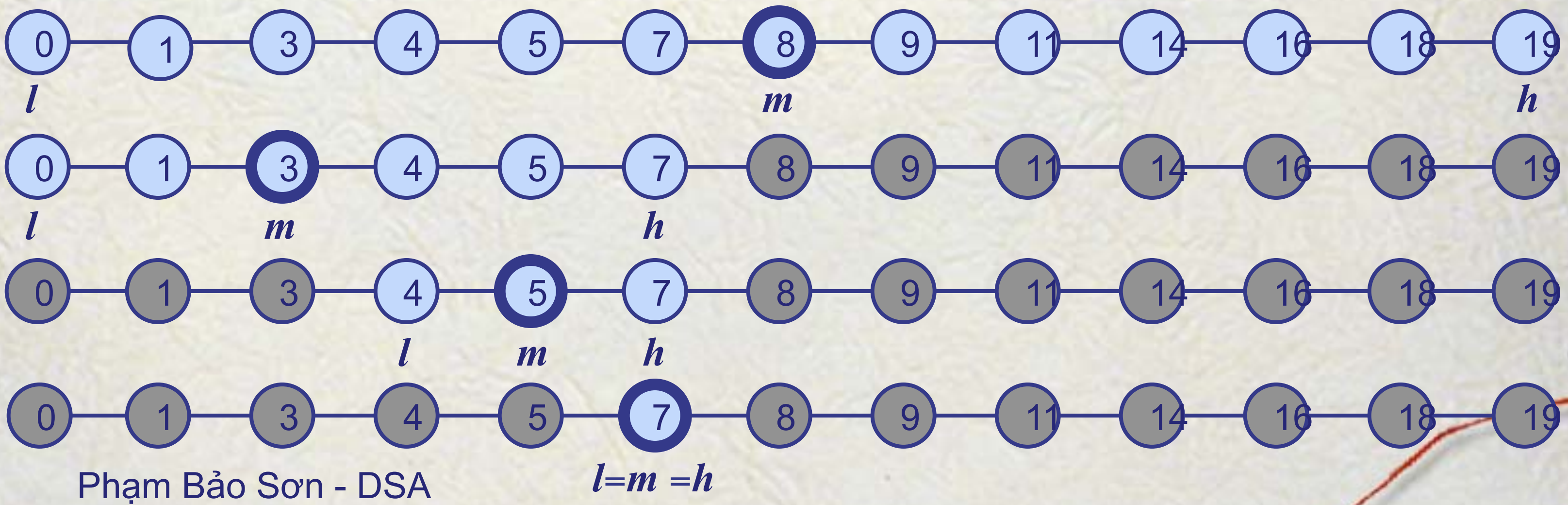
```
**Algorithm** insert(k,v):
Input: A key k and value v
Output: The entry (k,v) added to D
Create a new entry e = (k,v)
S.insertLast(e)        {S is unordered}
return e

Algorithm remove(e):
Input: An entry e
Output: The removed entry e or null if e was not in D
{We don't assume here that e stores its location in S}
B = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element() = e then
        S.remove(p)
        return e
return null            {there is no entry e in D}
```

Hash Table Implementation

- Unordered dictionaries.
- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

Binary Search

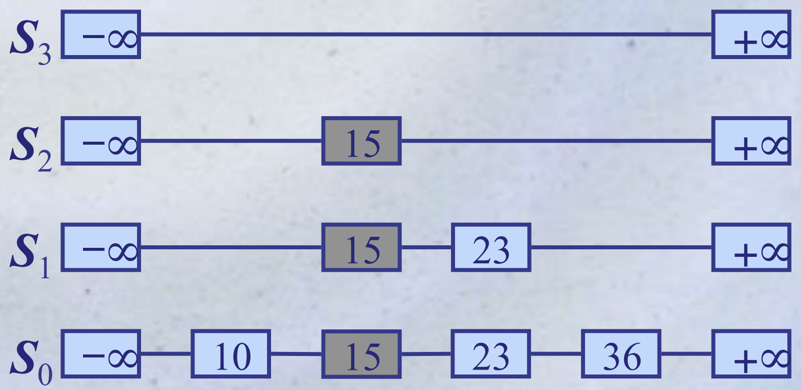
- Ordered dictionaries.
- Binary search performs operation find(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
- Example: find(7)



Search Table

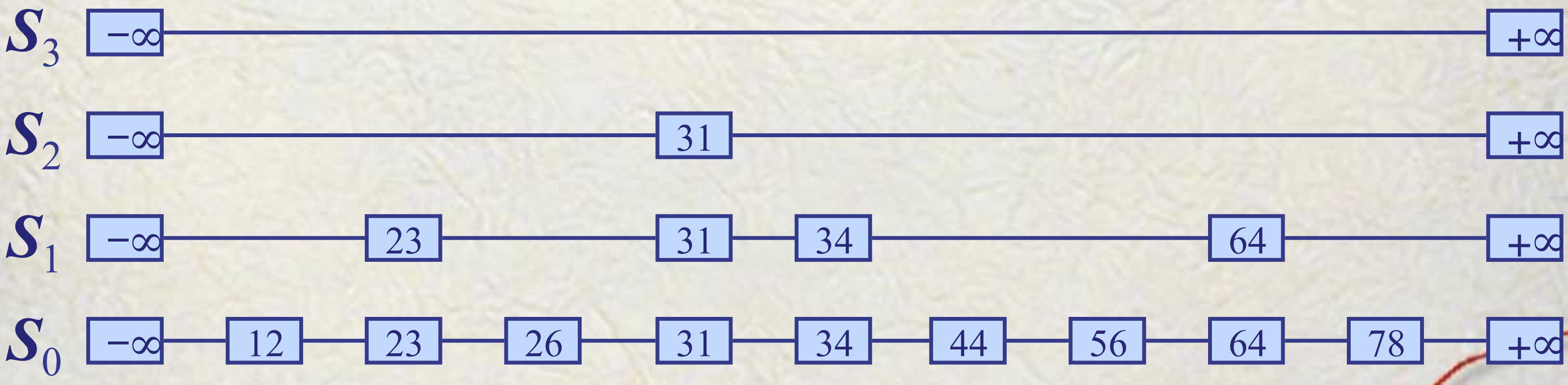
- A search table is a dictionary implemented by means of a sorted array
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - find takes $O(\log n)$ time, using binary search
  - insert takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - remove takes $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Skip Lists



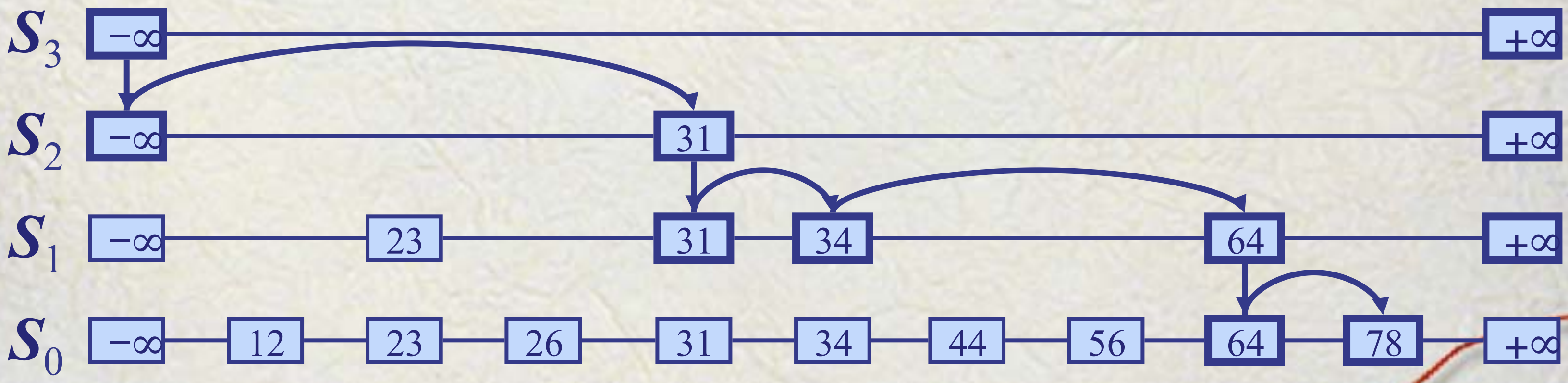
What is a Skip List

- 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.,
    $$S_0 \subseteq S_1 \subseteq \ldots \subseteq S_h$$
  - List S_h contains only the two special keys
- We show how to use a skip list to implement the dictionary ADT



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



Randomized Algorithms

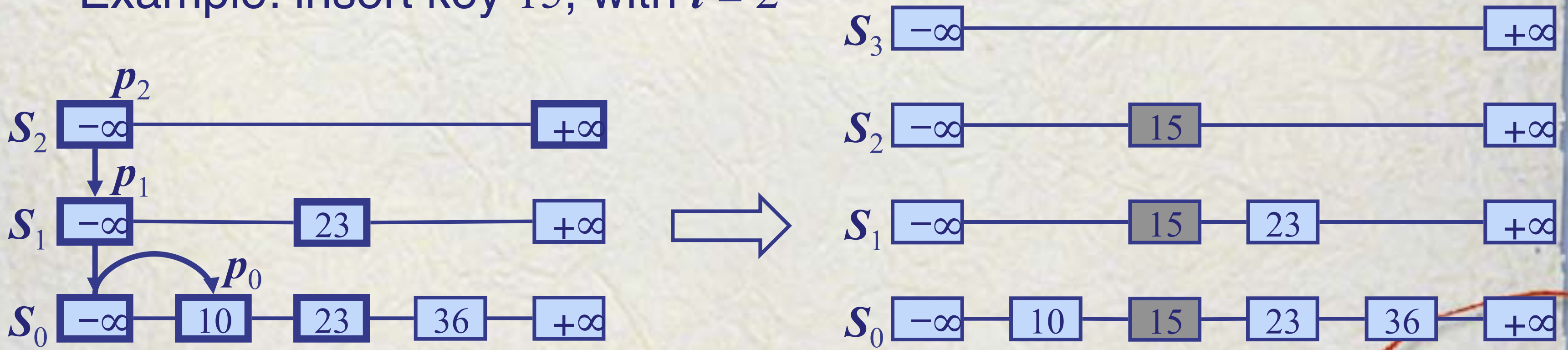
- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type

```
$b \leftarrow random()$
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

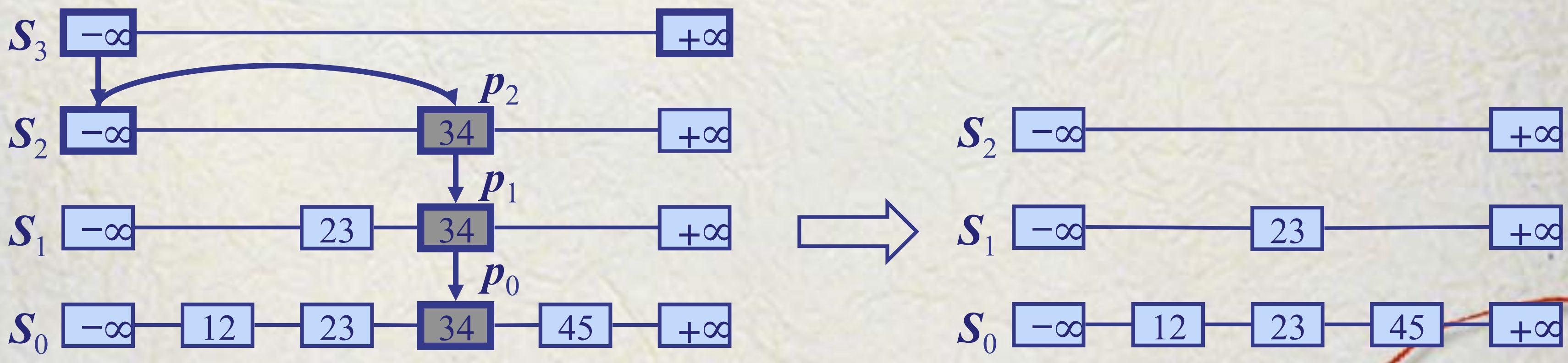
Insertion

- 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 $i = 2$



Deletion

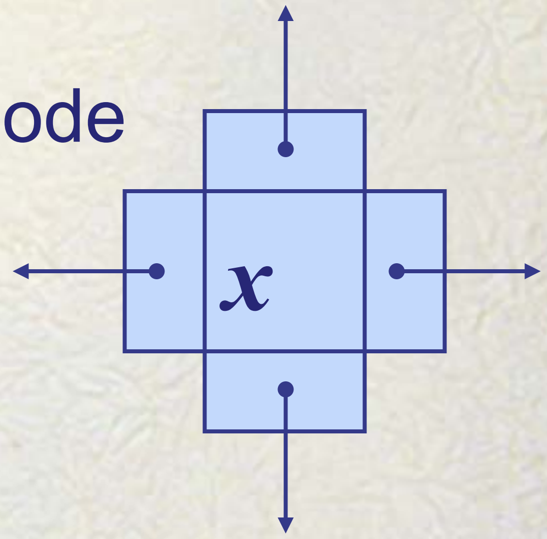
- To remove an entry with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys
- Example: remove key 34



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

quad-node



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 $\boldsymbol{i}$ consecutive heads when flipping a coin is $1/2^{\boldsymbol{i}}$

  Fact 2: If each of $\boldsymbol{n}$ entries is present in a set with probability $\boldsymbol{p}$, the expected size of the set is $\boldsymbol{np}$

- Consider a skip list with $\boldsymbol{n}$ entries
  - By Fact 1, we insert an entry in list $\boldsymbol{S_i}$ with probability $1/2^{\boldsymbol{i}}$
  - By Fact 2, the expected size of list $\boldsymbol{S_i}$ is $\boldsymbol{n}/2^{\boldsymbol{i}}$
- 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 $\boldsymbol{n}$ items is $\boldsymbol{O}(\boldsymbol{n})$

Height

- The running time of the search an insertion algorithms is affected by the height h of the skip list
- We show that with high probability, a skip list with n items has height $O(\log n)$
- We use the following additional probabilistic fact:

  Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most np

- Consider a skip list with n entires
  - By Fact 1, we insert an entry in list S_i with probability $1/2^i$
  - By Fact 3, the probability that list S_i has at least one item is at most $n/2^i$
- By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one entry is at most

  $$n/2^{3\log n} = n/n^3 = 1/n^2$$

- Thus a skip list with n entries has height at most $3\log n$ with probability at least $1 - 1/n^2$

Search and Update Time

- 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

Summary

- A skip list is a data structure for dictionaries 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