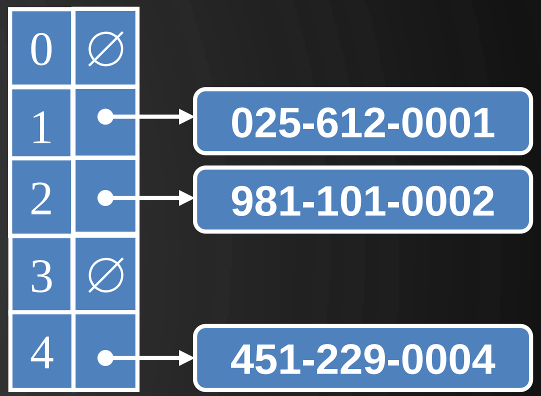


CHAPTER 10
MAPS, HASH TABLES, AND SKIP LISTS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

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
- Often called **associative containers**

THE MAP ADT

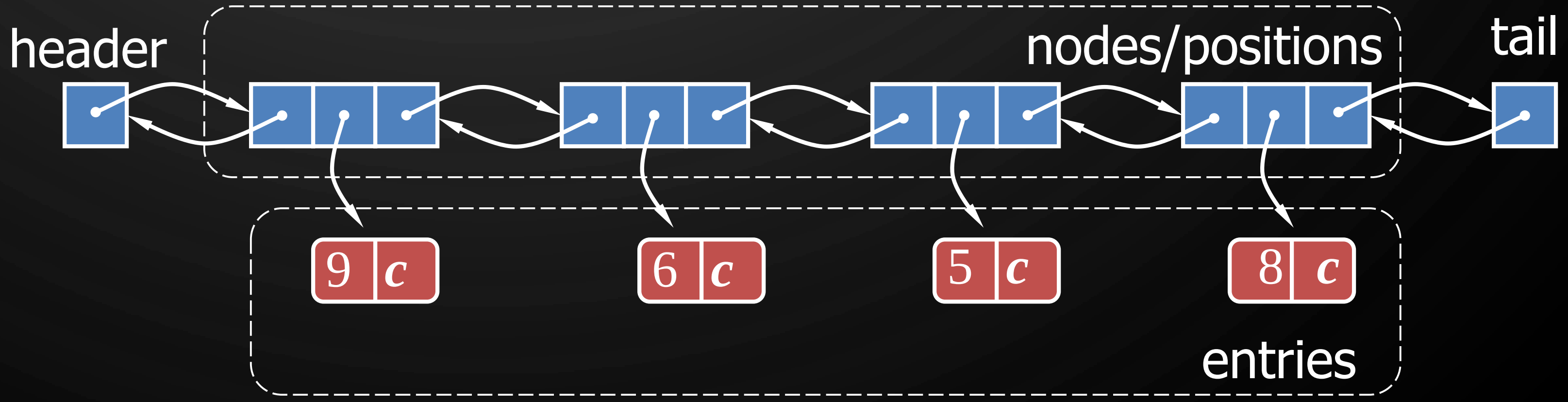
- `get(k)`: if the map M has an entry with key k, return its associated value; else, return null
- `put(k, v)`: insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- `remove(k)`: if the map M has an entry with key k, remove it from 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

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)`

LIST-BASED MAP

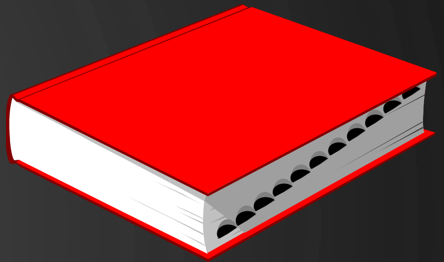
- We can implement a map with an unsorted list
  - Store the entries in arbitrary order
- Complexity of get, put, remove?
  - $O(n)$ on put, get, and remove



DIRECT ADDRESS TABLE MAP IMPLEMENTATION

- A direct address table is a map in which
  - The keys are in the range $[0, N]$
  - Stored in an array T of size N
  - Entry with key k stored in $T[k]$
- Performance:
  - `put(k, v)`, `get(k)`, and `remove(k)` all take $O(1)$ time
  - Space - requires space $O(N)$, independent of n, the number of entries stored in the map
- The direct address table is not space efficient unless the range of the keys is close to the number of elements to be stored in the map, i.e., unless n is close to N.

SORTED MAP

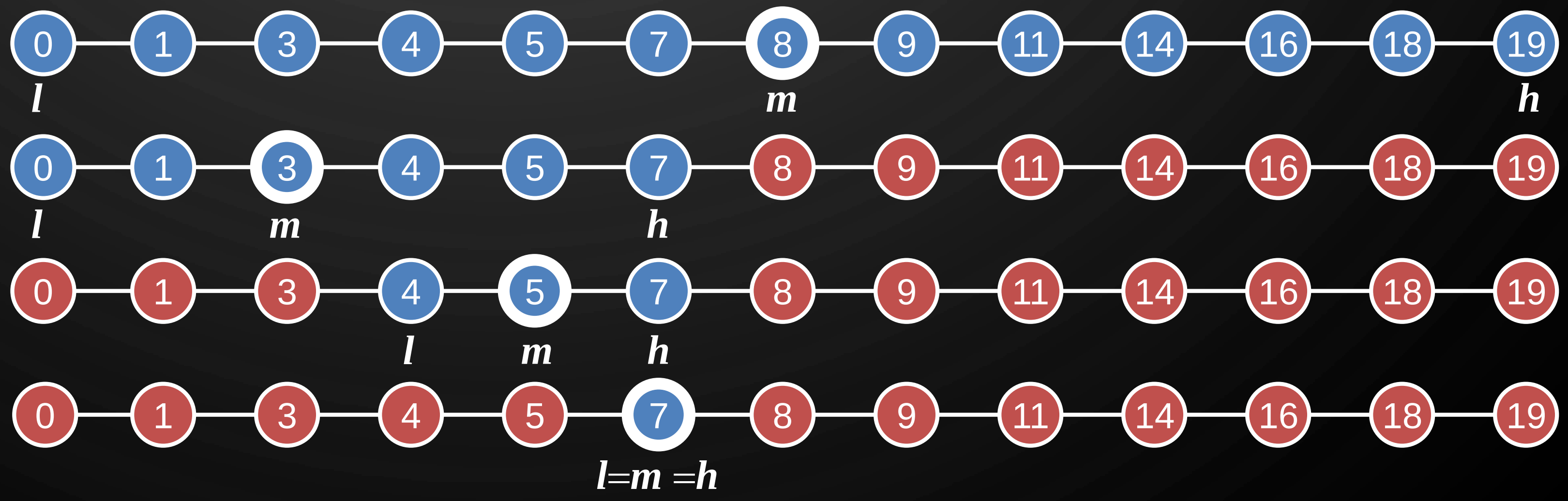


- A Sorted Map supports the usual map operations, but also maintains an order relation for the keys.
- Naturally supports
  - Sorted search tables - store dictionary in an array by non-decreasing order of the keys
  - Utilizes binary search
- Sorted Map ADT adds the following functionality to a map
  - `firstEntry()`, `lastEntry()` – return iterators to entries with the smallest and largest keys, respectively
  - `ceilingEntry(k)`, `floorEntry(k)` – return an iterator to the least/greatest key value greater than/less than or equal to k
  - `lowerEntry(k)`, `higherEntry(k)` – return an iterator to the greatest/least key value less than/greater than k
  - etc

EXAMPLE OF ORDERED MAP: BINARY SEARCH

- Binary search performs operation `get(k)` on an ordered search table
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
- Example

  `get(7)`



SIMPLE MAP IMPLEMENTATION SUMMARY

	`put(k, v)`	`get(k)`	Space
Unsorted list	$O(n)$	$O(n)$	$O(n)$
Direct Address Table	$O(1)$	$O(1)$	$O(N)$
Sorted Search Table (Naturally supported Sorted Map)	$O(n)$	$O(\log n)$	$O(n)$

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** (also known as a dictionary) 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.

SET ADT

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:

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

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 $S - T$.

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)
Input: Sets A,B as sorted lists
Output: Set S
S ← ∅
while ¬A.isEmpty() ∧ ¬B.isEmpty() do
    a ← A.first(); b ← B.first()
    if a < b
        aIsLess(a, S) //generic action
        A.removeFirst();
    else if b < a
        bIsLess(b, S) //generic action
        B.removeFirst()
    else //a = b
        bothAreEqual(a, b, S) //generic action
        A.removeFirst(); B.removeFirst()
while ¬A.isEmpty() do
    aIsLess(A.first(), S); A.eraseFront()
while ¬B.isEmpty() do
    bIsLess(B.first(), S); B.removeFirst()
return S
```

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

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 $(k, E(k))$

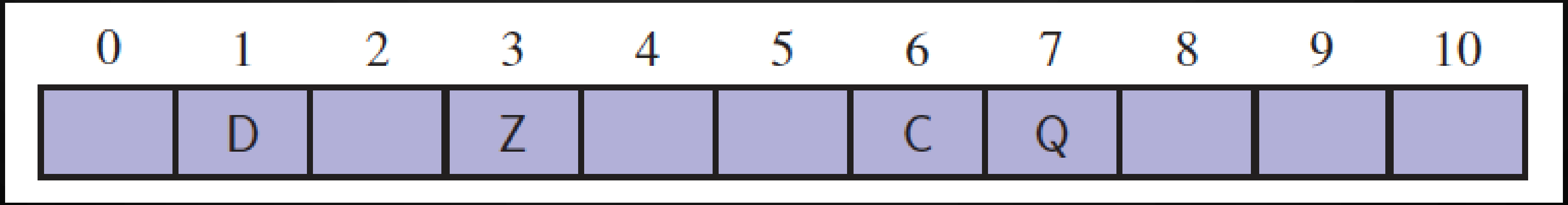
MULITMAPS

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.

HASH TABLES

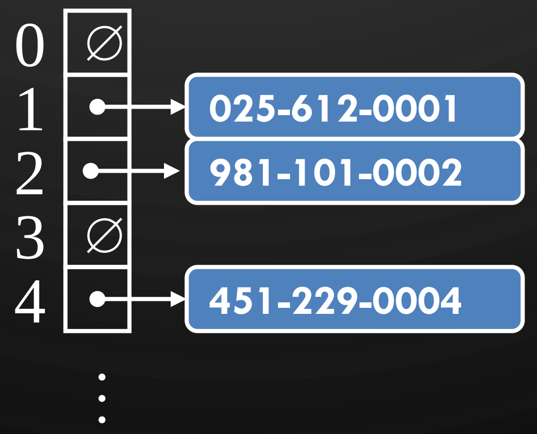
INTUITIVE NOTION OF A MAP

- 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 $N \geq n$.



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.



HASH TABLES

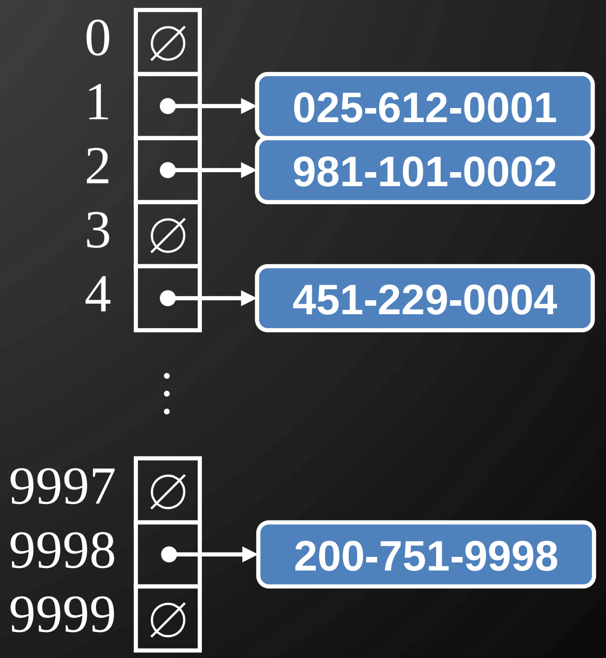
- A Hash function $h(k) \rightarrow [0, N-1]$
  - The integer $h(k)$ is referred to as the hash value of key k
  - Example - $h(k) = k \bmod N$ could be a hash function for integers
- **Hash tables** consist of
  - A hash function h
  - Array A of size N (either to an element itself or to a “bucket”)
- Goal is to store elements (k, v) at index $i = h(k)$

ISSUES WITH HASH TABLES

- Issues
  - Collisions - some keys will map to the same index of H (otherwise we have a Direct Address Table).
    - Chaining - put values that hash to same location in a linked list (or a “bucket”)
    - Open addressing - if a collision occurs, have a method to select another location in the table.
  - Load factor
  - Rehashing

EXAMPLE

- We design a hash table for a Map storing items (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 $h(k) = \text{last four digits of } k$



HASH FUNCTIONS

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

  h_1: keys $\rightarrow$ integers
- **Compression function**:

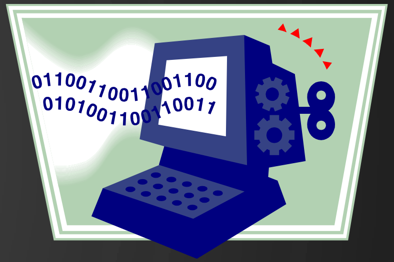
  h_2: integers $\rightarrow [0, N-1]$

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

$$h(k) = h_2\big(h_1(k)\big)$$

- 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
  - 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 C++)
- 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 C++)

HASH CODES

- 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 $p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}$ 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)
- Cyclic Shift:
  - Like polynomial accumulation except use bit shifts instead of multiplications and bitwise or instead of addition
- Can be used on floating point numbers as well by converting the number to an array of characters

COMPRESSION FUNCTIONS

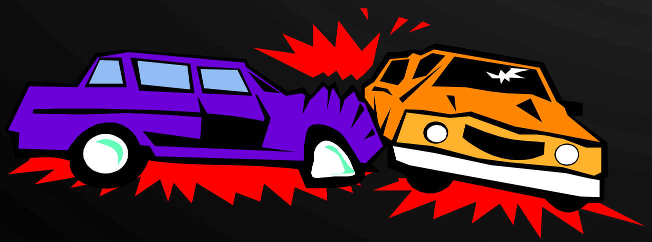
- Division:
  - $h_2(k) = k \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

- Multiply, Add and Divide (MAD):
  - $h_2(k) = (ak + b) \bmod N$
  - a and b are nonnegative integers such that

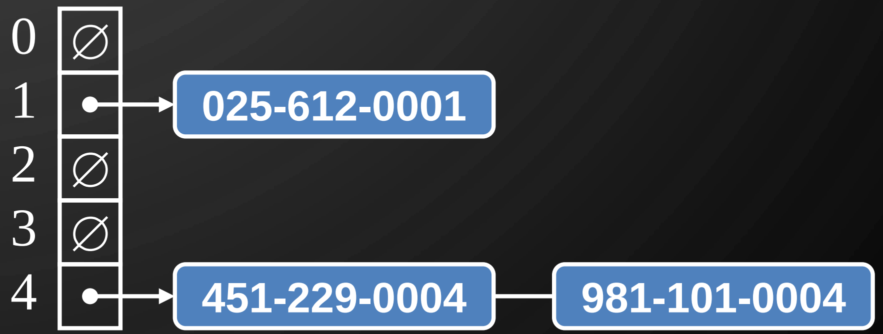
    $$a \bmod N \neq 0$$
  - Otherwise, every integer would map to the same value b

COLLISION RESOLUTION WITH
SEPARATE CHAINING

- 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



- Chaining is simple, but requires additional memory outside the table



EXERCISE
SEPARATE CHAINING

- Assume you have a hash table H with $N = 9$ slots ($A[0-8]$) and let the hash function be $h(k) = k \bmod N$
- Demonstrate (by picture) the insertion of the following keys into a hash table with collisions resolved by chaining
  - 5, 28, 19, 15, 20, 33, 12, 17, 10

COLLISION RESOLUTION WITH
OPEN ADDRESSING - LINEAR PROBING

- In 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. So the ith cell checked is:

$$h(k,i) = |h(k) + i| \bmod N$$

- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer probe sequence

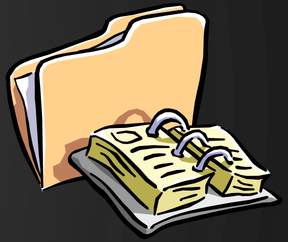
- Example:
  - $h(k) = k \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

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

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

<u>Algorithm get(k)</u>

$i \leftarrow h(k)$
$p \leftarrow 0$
repeat
    $c \leftarrow A[i]$
    if $c \neq \emptyset$
        return *null*
    else if $c.\mathrm{key}() = k$
        return c
    else
        $i \leftarrow (i + 1) \bmod N$
        $p \leftarrow p + 1$
until $p = N$
return *null*

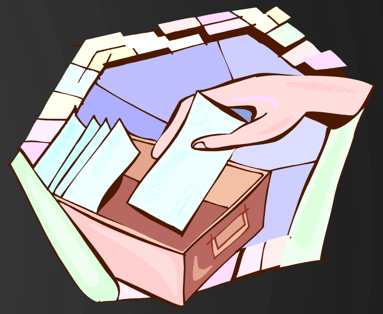
UPDATES WITH LINEAR PROBING

- To handle insertions and deletions, we introduce a special object, called DEFUNCT, which replaces deleted elements
- `remove(k)`
  - We search for an item with key k
  - If such an item (k, v) is found, we replace it with the special item DEFUNCT
  - Else, we return null
- `put(k, v)`
  - 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

EXERCISE
OPEN ADDRESSING – LINEAR PROBING

- Assume you have a hash table H with $N = 11$ slots ($A[0-10]$) and let the hash function be $h(k) = k \bmod N$
- Demonstrate (by picture) the insertion of the following keys into a hash table with collisions resolved by linear probing.
  - 10, 22, 31, 4, 15, 28, 17, 88, 59

COLLISION RESOLUTION WITH
OPEN ADDRESSING – QUADRATIC PROBING



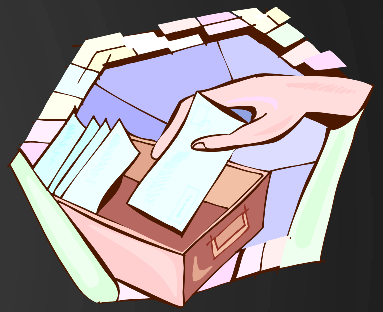
- Linear probing has an issue with clustering
- Another strategy called quadratic probing uses a hash function

$$h(k, i) = (h(k) + i^2) \bmod N$$

  for $i = 0, 1, \ldots, N - 1$

- This can still cause secondary clustering

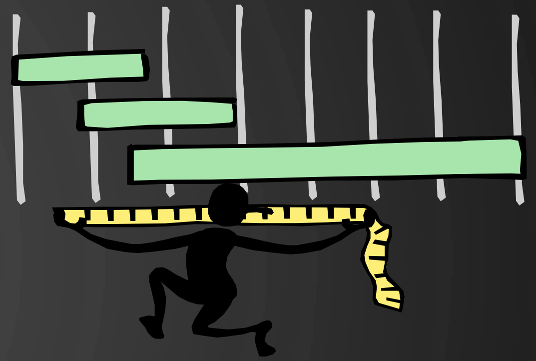
COLLISION RESOLUTION WITH
OPEN ADDRESSING - DOUBLE HASHING



- Double hashing uses a secondary hash function $h_2(k)$ and handles collisions by placing an item in the first available cell of the series
  $$h(k,i) = \left(h_1(k) + ih_2(k)\right) \bmod N$$
  for $i = 0, 1, \ldots, N-1$
- The secondary hash function $h_2(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression map for the secondary hash function:
  $$h_2(k) = q - (k \bmod q)$$
  where
  - $q < N$
  - q is a prime
- The possible values for $h_2(k)$ are $1, 2, \ldots, q$

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 $\alpha = \frac{n}{N}$ 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

$$\frac{1}{1-\alpha} = \frac{1}{1-{}^{n}/_{N}} = \frac{1}{N-{}^{n}/_{N}} = \frac{N}{N-n}$$

- The expected running time of all the Map 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

UNIFORM HASHING ASSUMPTION

- The probe sequence of a key k is the sequence of slots probed when looking for k
  - In open addressing, the probe sequence is $h(k,0), h(k,1), \dots, h(k,N-1)$
- Uniform Hashing Assumption
  - Each key is equally likely to have any one of the $N!$ permutations of $\{0,1,\dots,N-1\}$ as is probe sequence
  - Note: Linear probing and double hashing are far from achieving Uniform Hashing
    - Linear probing: N distinct probe sequences
    - Double Hashing: N^2 distinct probe sequences

PERFORMANCE OF UNIFORM HASHING

- Theorem: Assuming uniform hashing and an open-address hash table with load factor $\alpha = \frac{n}{N} < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$.
- Exercise: compute the expected number of probes in an unsuccessful search in an open address hash table with $\alpha = \frac{1}{2}$, $\alpha = \frac{3}{4}$, and $\alpha = \frac{99}{100}$.

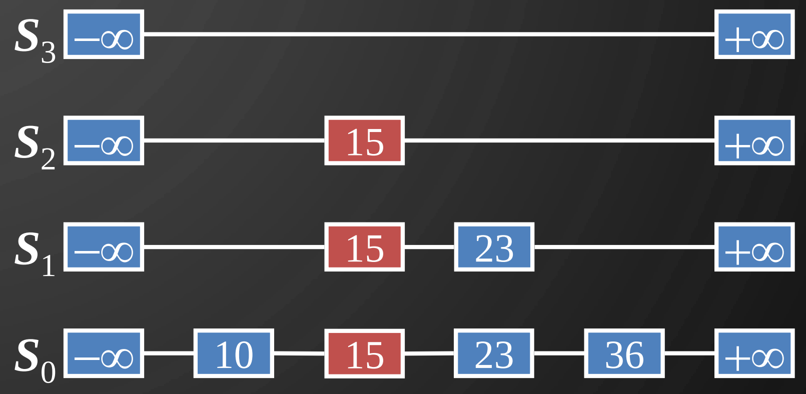
ON REHASHING

- Keeping the load factor low is vital for performance
- When resizing the table:
  - Reallocate space for the array (of size that is a prime)
  - Design a new hash function (new parameters) for the new array size (practically, change the mod)
  - For each item you reinsert into the table rehash

SUMMARY MAPS (SO FAR)

	`put(k, v)`	`get(k)`	Space
Unsorted list	$O(n)$	$O(n)$	$O(n)$
Direct Address Table	$O(1)$	$O(1)$	$O(N)$
Sorted Search Table (Naturally supported Sorted Map)	$O(n)$	$O(\log n)$	$O(n)$
Hashing (chaining)	$O\left(\frac{n}{N}\right)$	$O\left(\frac{n}{N}\right)$	$O(n+N)$
Hashing (open addressing)	$O\left(\frac{1}{1-\frac{n}{N}}\right)$	$O\left(\frac{1}{1-\frac{n}{N}}\right)$	$O(N)$

SKIP LISTS



RANDOMIZED ALGORITHMS

- A randomized algorithm controls its execution through random selection (e.g., coin tosses)
- It contains statements like:

  $b \leftarrow \text{randomBit}()$

  if $b = 0$

    do something…

  else $//b = 1$

    do something else…

- Its running time depends on the outcomes of the "coin tosses"

- Through probabilistic analysis we can derive the expected running time of a randomized algorithm
- We make the following assumptions in the analysis:
  - the coins are unbiased
  - 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 to insert in expected $O(\log n)$–time
- When randomization is used in data structures they are referred to as probabilistic data structures

WHAT IS A SKIP LIST?

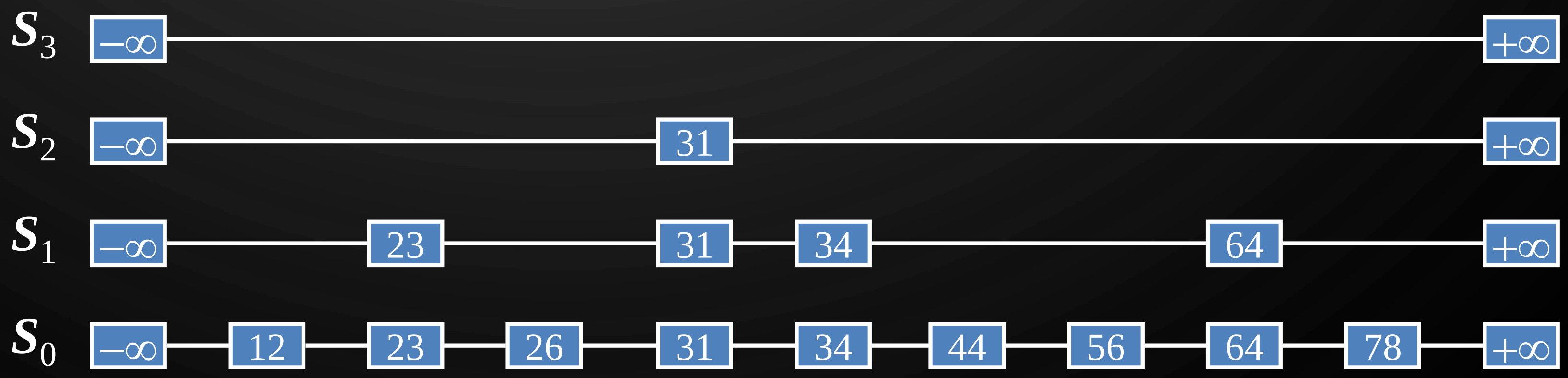
- A **skip list** for a set S of distinct (key, element) items is a series of lists

$$S_0, S_1, \dots, S_h$$

  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non-decreasing order
  - Each list is a subsequence of the previous one, i.e.,

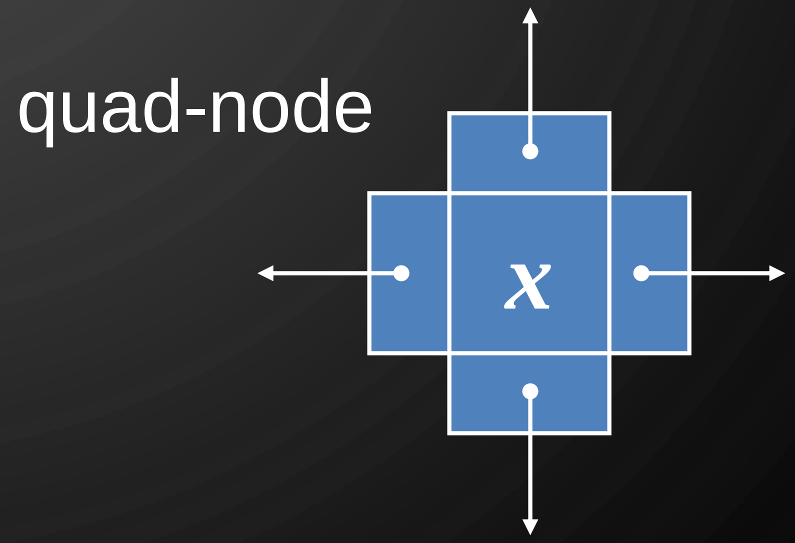
$$S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$$

  - List S_h contains only the two special keys
- Skip lists are one way to implement the Ordered Map ADT
- [Java applet]



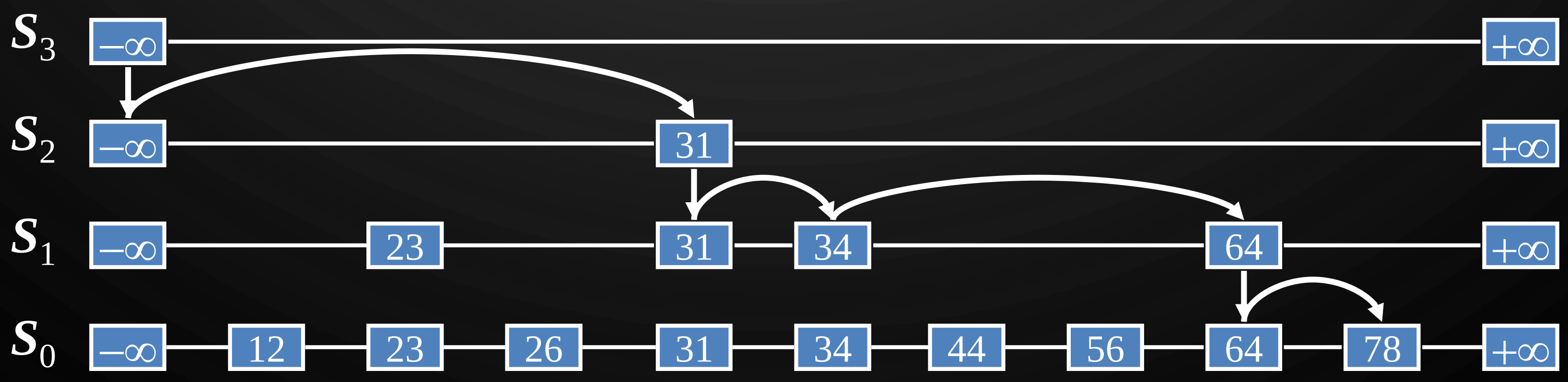
IMPLEMENTATION

- We can implement a skip list with quad-nodes
- A quad-node stores:
  - (Key, Value)
  - links to the nodes before, after, below, and above
- Also, we define special keys $+\infty$ and $-\infty$, and we modify the key comparator to handle them



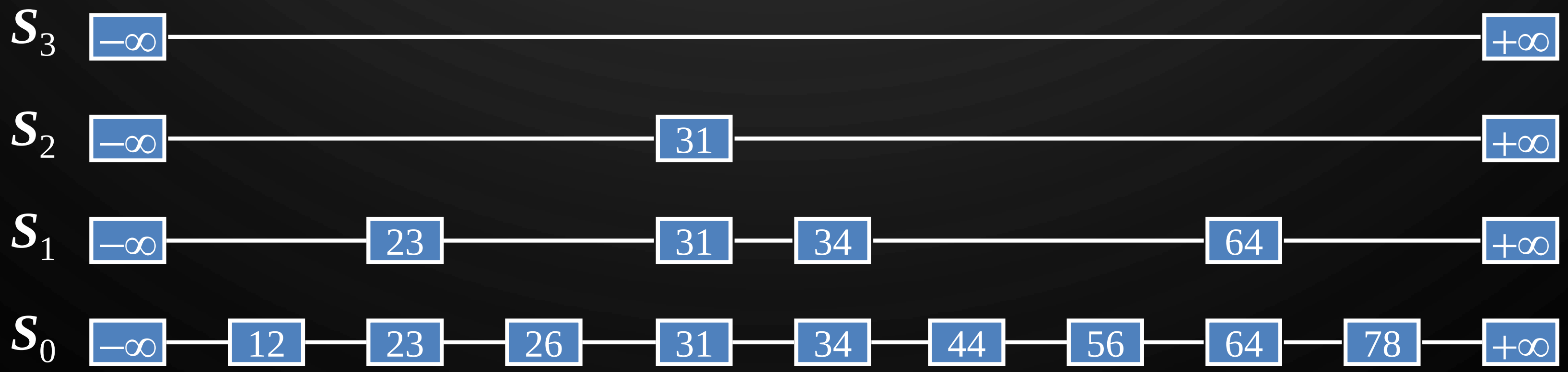
SEARCH - GET (K)

- We search for a key k in a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare k with $y \leftarrow p.\mathrm{next}().\mathrm{key}()$
    $x = y$: we return $p.\mathrm{next}().\mathrm{value}()$
    $x > y$: 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



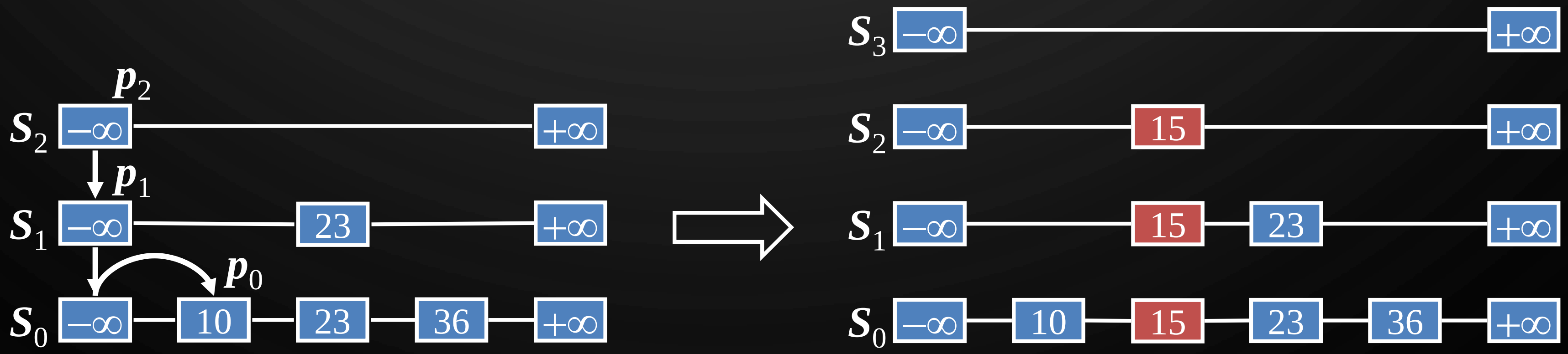
EXERCISE
SEARCH

- We search for a key k in a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare k with $y \leftarrow p.\text{next}().\text{key}()$
    $x = y$: we return $p.\text{next}().\text{value}()$
    $x > y$: we scan forward
    $x < y$: we drop down
  - If we try to drop down past the bottom list, we return ***NO_SUCH_KEY***
- Ex 1: search for 64: list the (S_i, node) pairs visited and the return value
- Ex 2: search for 27: list the (S_i, node) pairs visited and the return value



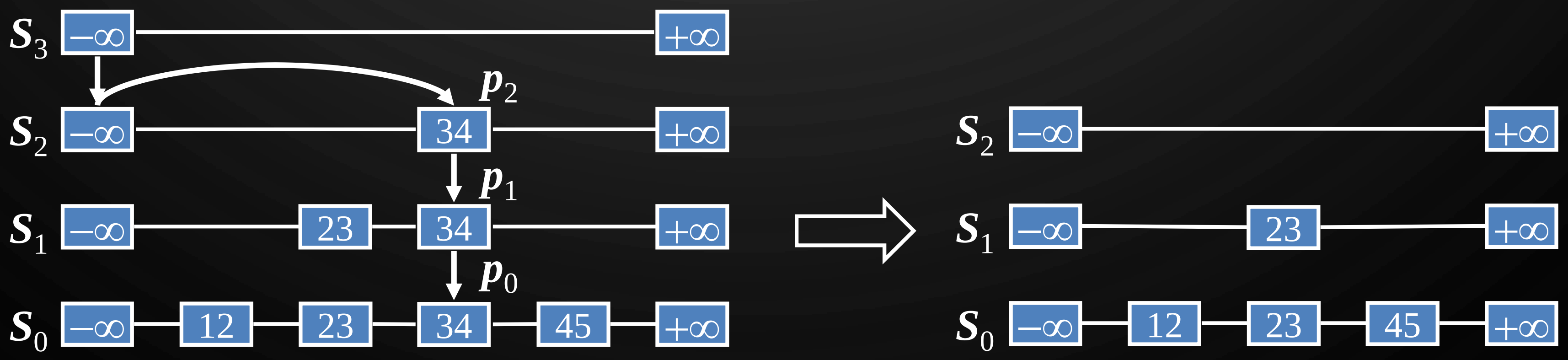
INSERTION - `PUT(K, V)`

- To insert an item (k, v) 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 $S_{h+1}, \dots, S_{i+1}$ each containing only the two special keys
  - We search for k in the skip list and find the positions $p_0, p_1, \dots, p_i$ of the items with largest key less than k in each list $S_0, S_1, \dots, S_i$
  - For $i \leftarrow 0, \dots, i$, we insert item (k, v) into list S_i after position p_i
- Example: insert key 15, with $i = 2$



DELETION - REMOVE(K)

- To remove an item with key k from a skip list, we proceed as follows:
  - We search for k in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key k, where position p_i is in list S_i
  - 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



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 $\frac{1}{2^i}$
  - Fact 2: If each of n items is present in a set with probability p, the expected size of the set is np
- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $\frac{1}{2^i}$
  - By Fact 2, the expected size of list S_i is $\frac{n}{2^i}$
- 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 is $O(2n)$

HEIGHT

- The running time of $\mathrm{find}(k)$, $\mathrm{put}(k, v)$, and $\mathrm{erase}(k)$ operations are 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

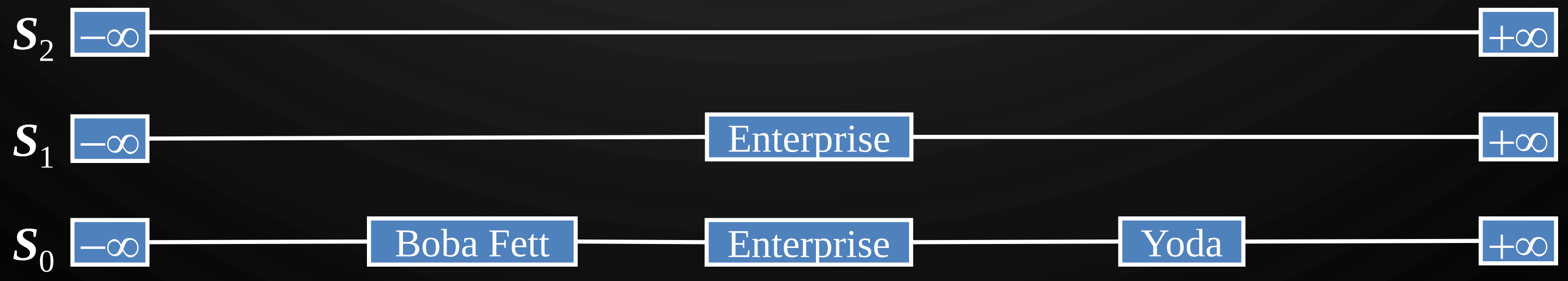
- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $\frac{1}{2^i}$
  - By Fact 3, the probability that list S_i has at least one item is at most $\frac{n}{2^i}$
- By picking $i = 3 \log n$, we have that the probability that $S_{3 \log n}$ has at least one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

SEARCH AND UPDATE TIMES

- The search time in a skip list is proportional to
  - the number of drop-down steps
  - the number of scan-forward steps
- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ expected time
- 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

EXERCISE

- You are working for ObscureDictionaries.com a new online start-up which specializes in sci-fi languages. The CEO wants your team to describe a data structure which will efficiently allow for searching, inserting, and deleting new entries. You believe a skip list is a good idea, but need to convince the CEO. Perform the following:
  - Illustrate insertion of “X-wing” into this skip list. Randomly generated (1, 1, 1, 0).
  - Illustrate deletion of an incorrect entry “Enterprise”
  - Argue the complexity of deleting from a skip list



SUMMARY

- A skip list is a data structure for dictionaries that uses a randomized insertion algorithm
- In a skip list with n items
  - 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