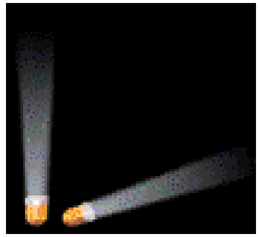
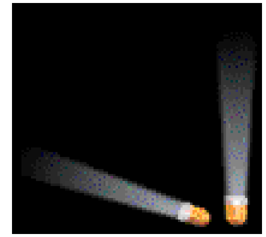
Dictionaries

Data structures

Fall 2018

Dictionary





- Collection of pairs.
  - (key, element)
  - Pairs have different keys (keys are unique).
- Operations.
  - get(theKey)
  - put(theKey, theElement)
  - remove(theKey)

Application

- Collection of student records in this class.
  - (key, element) = (student id, linear list of assignment and exam scores)
  - All keys are distinct.
- Get the element whose key is 2008-12345.
- Update the element whose key is 2007-54321.
  - put() implemented as update when there is already a pair with the given key.
  - remove() followed by put().

Dictionary With Duplicates

- Keys are not required to be distinct.
- Student records as a multiset.
  - Pairs are of the form (student id, assg. number, marks).
  - May have two or more entries for the same key.
    - (2008-12345, 1, 36)
    - (2007-54321, 1, 44)
    - (2008-12345, 2, 43)
    - (2006-34251, 2, 41)
    - (2007-54321, 2, 35)
    - etc.

Can also be interpreted as a dictionary with (student id, assg. number) as the key

Dictionary ADT

AbstractDataType *Dictionary* {

instances

collection of elements with distinct keys

operations

get(k) : return the element with key k;

put(k, x) : put the element x whose key is k into the dictionary and return the old element associated with k;

remove(k) : remove the element with key k and return it;

}

Represent As A Linear List

- $L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})$
- Each e_i is a pair (key, element).
- 5-pair dictionary D = (a, b, c, d, e).
  - a = (aKey, aElement), b = (bKey, bElement), etc.
- Array or linked representation.

Array Representation

a	b	c	d	e										

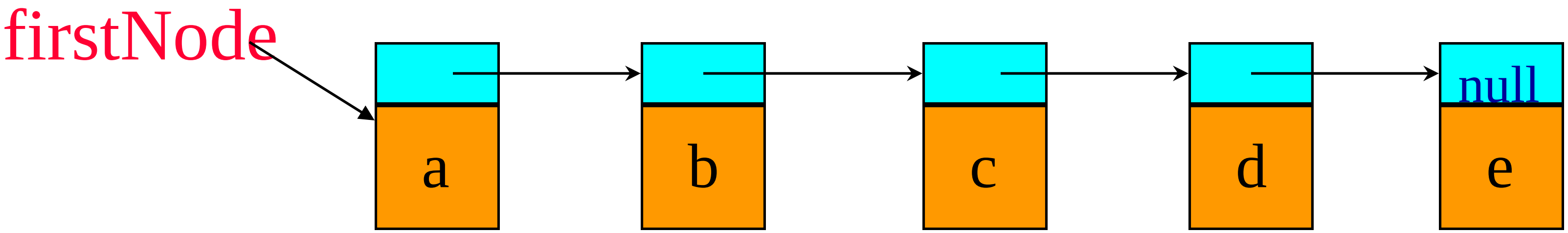
- get(theKey)
  - O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to add at right end.
- remove(theKey)
  - O(size) time.

Sorted Array

A	B	C	D	E										

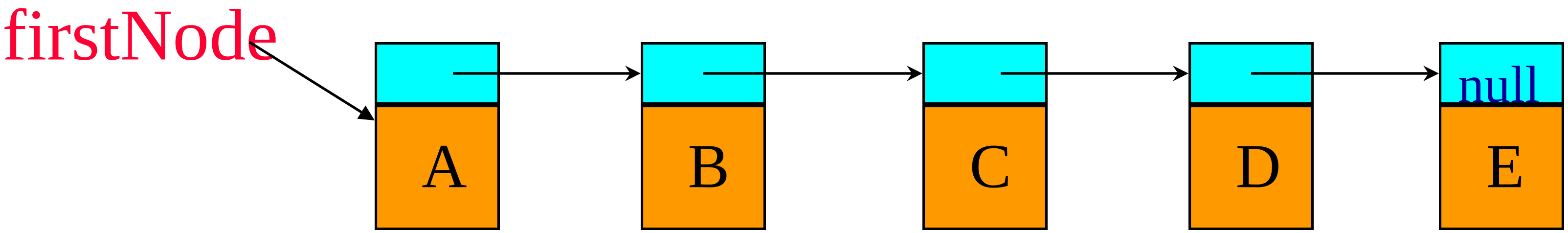
- elements are in ascending order of key.
- get(theKey)
  - O(log size) time
- put(theKey, theElement)
  - O(log size) time to verify duplicate, O(size) to add.
- remove(theKey)
  - O(size) time.

Unsorted Chain



- get(theKey)
  - O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to add at left end.
- remove(theKey)
  - O(size) time.

Sorted Chain



- Elements are in ascending order of Key.
- get(theKey)
  - O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to put at proper place.
- remove(theKey)
  - O(size) time.

Dictionary implementations

Complexities of dictionary operations in various dictionary implementations (n = size):

Implementation	Worst Case			Excepted		
	Search	Insert	Remove	Search	Insert	Remove
Sorted array	$\theta(\log n)$	$\theta(n)$	$\theta(n)$	$\theta(\log n)$	$\theta(n)$	$\theta(n)$
Sorted chain	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
Skip lists	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$
Hash tables	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(1)$
BBST*	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$

*BBST – Balanced Binary Search Tree

Skip lists and BBSTs are better than hashing when we need to output all elements in sorted order or search by element rank.

Skip Lists

Skip Lists

- Worst-case time for get, put, and remove is O(size).
- Expected time is O(log (size)).
- Simple randomized data structure; easy to implement.

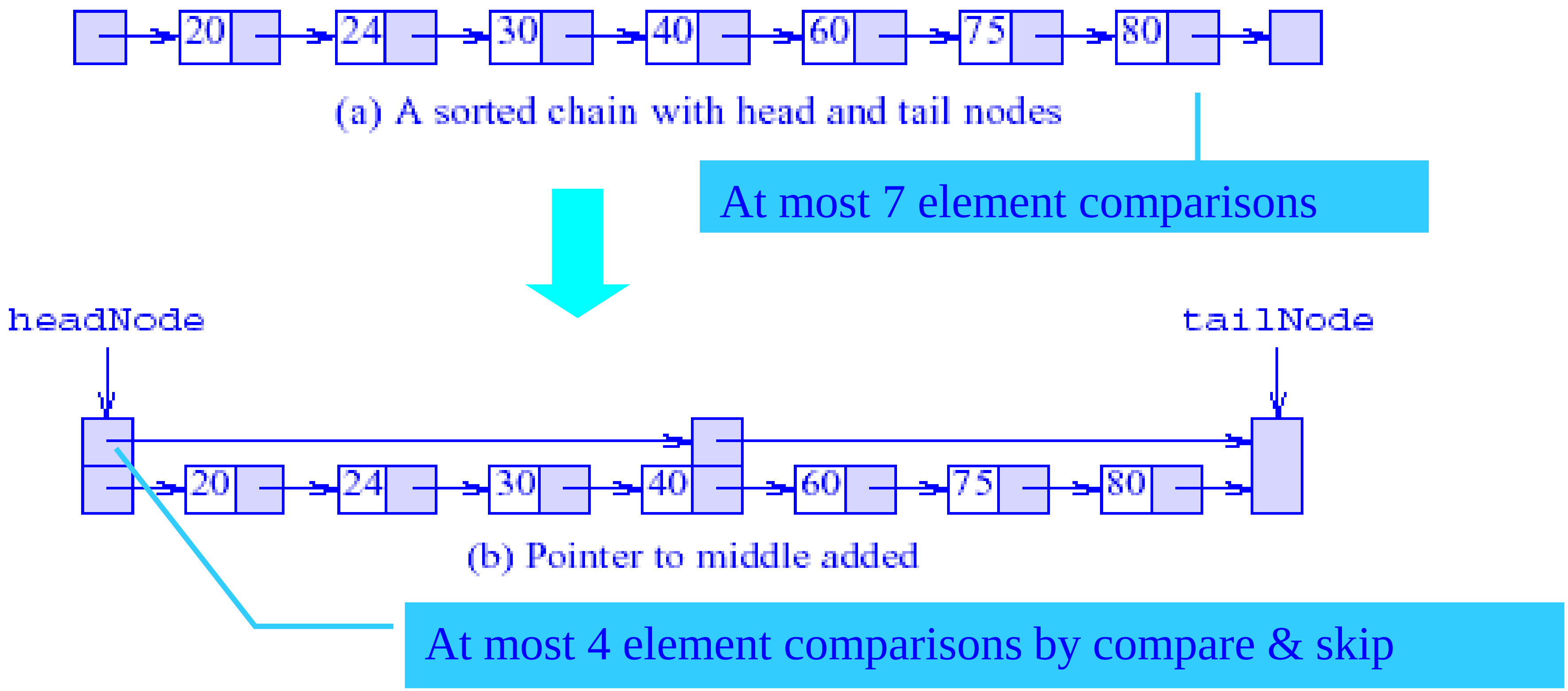
Skip Lists

- In a sorted chain with n elements, to search for an arbitrary element e_i
  - n element comparisons are needed in the worst case
  - The number of comparisons can be reduced to n/2 + 1 by storing a pointer to the middle element in the chain
    - Compare with the middle point
    - If $e_i <$ middle point, search only the left half
    - Else, search only the right half

- Adding more pointers, we can reduce the number of comparisons further
  - We can perform a binary search in a sorted chain by storing extra pointers into the chain

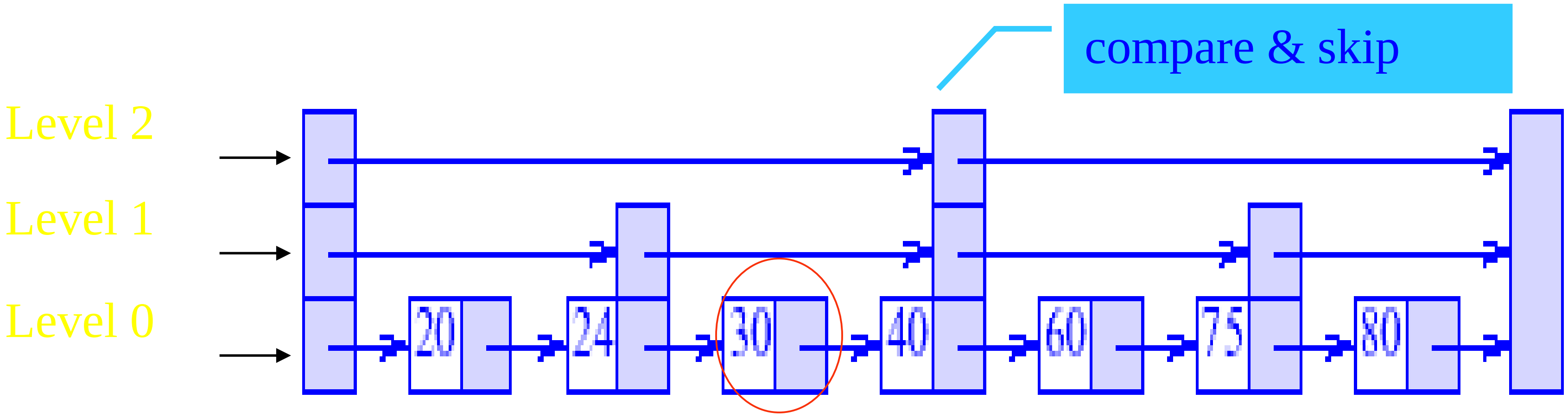
Skip Lists (2)

- Example : Consider the seven-element sorted chain



Skip Lists (3)

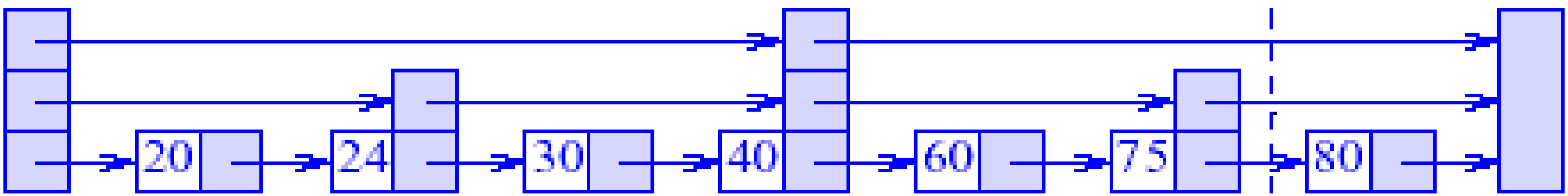
- Example(Cont.)
  - By keeping pointers to the middle elements of each half, we can reduce the number of element comparisons further



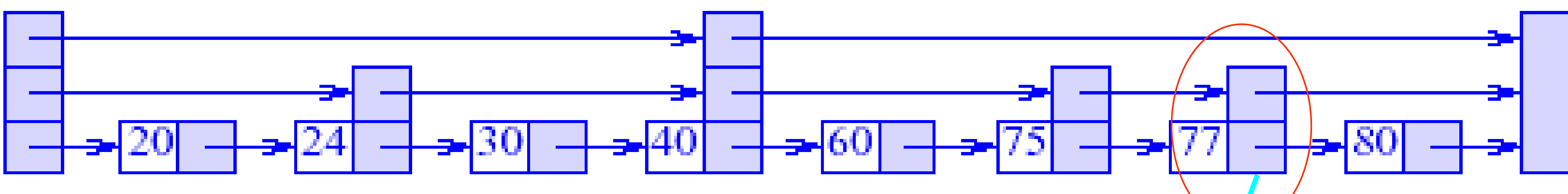
(c) Pointers to every second node added

Skip Lists – put()

- Example(Cont.) : Consider inserting element 77



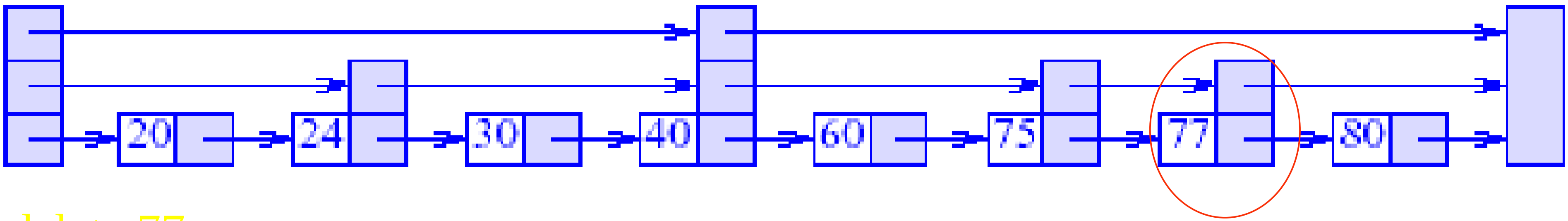
(d) Last pointers encountered when searching for 77



(e) 77 inserted

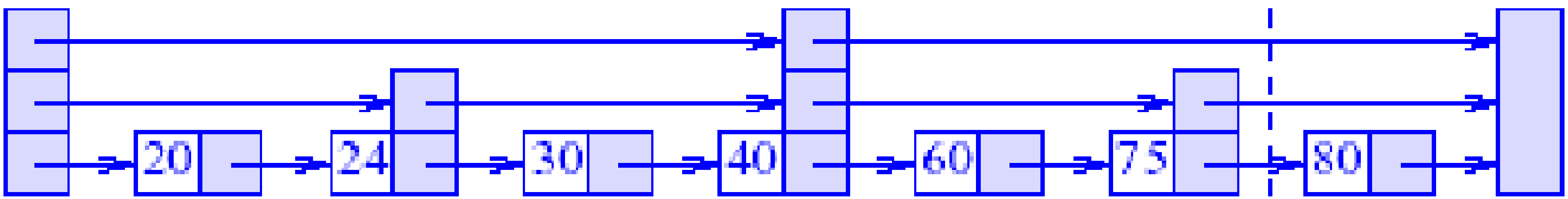
The element 77 may be in level 0 or level 1 or level 2

Skip Lists – remove()



delete 77

1. Search for 77
2. The encountered pointers are the level 2 in “40” and the level 1,0 in “75”
3. Level 0,1 pointers are to be changed to point to the element after 77



Asymptotic performance

- Complexity
  - get(), put(), remove():
    - O(n + maxLevel) worst case, where n is the number of elements
    - O(maxLevel) expected (maxLevel = O(log n))
  - Space:
    - Worst case space: O(n * MaxLevel) for pointers
    - Expected number of pointers = O(n)

Hashing

Hash Tables

- Worst-case time for get, put, and remove is O(size).
- Expected time is O(1).
- Space: O(n)

Ideal Hashing

- Uses a 1D array (or table) table[0:b-1].
  - Each position of this array is called a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
  - f(k) is the home bucket for key k.
- Every dictionary pair (key, element) is stored in its home bucket table[f[key]].

Ideal Hashing Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is: f(key) = key/11.
- Pairs are stored in table as below:

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- get, put, and remove take O(1) time.

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Where does (26,g) go?
- Keys that have the same home bucket are called synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The home bucket for (26,g) is already occupied.

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- A collision occurs when the home bucket for a new pair is occupied by a pair with a different key.
- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.
- Need a method to handle overflows.

Hash Table Issues

- Choice of hash function.
- Overflow handling method.
- Size (number of buckets) of hash table.

Hash Functions

- Two parts:
  - Convert key into an integer in case the key is not an integer.
    - Done by the method hashCode().
  - Map an integer into a home bucket.
    - f(key.hashCode()) is an integer in the range [0, b-1], where b is the number of buckets in the table.

String To Integer

- Each Java character is 2 bytes long.
- An int is 4 bytes.
- A 2 character string s may be converted into a unique 4 byte int using the code:

```
int answer = s.charAt(0);
answer = (answer << 16) + s.charAt(1);
```

- Strings that are longer than 2 characters do not have a unique int representation.

String To Non-negative Integer

```
public static int integer(String s)
{
  int length = s.length();
     // number of characters in s
  int answer = 0;
  if (length % 2 == 1)
  {// length is odd
    answer = s.charAt(length - 1);
    length--;
   }
```

String To Non-negative Integer

```
  // length is now even
  for (int i = 0; i < length; i += 2)
  {// process two characters at a time
     answer += s.charAt(i);
     answer += ((int) s.charAt(i + 1)) << 16;
  }
  return (answer < 0) ? -answer : answer;
}
```

Map Into A Home Bucket

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Most common method is by division.

homeBucket =

Math.abs(theKey.hashCode()) % divisor;

- divisor equals number of buckets b.
- 0 <= homeBucket < divisor = b

Uniform Hash Function

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Let keySpace be the set of all possible keys.
- A uniform hash function maps the keys in keySpace into buckets such that approximately the same number of keys get mapped into each bucket.

Uniform Hash Function

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Equivalently, the probability that a randomly selected key has bucket i as its home bucket is $1/b$, $0 <= i < b$.

- A uniform hash function minimizes the likelihood of an overflow when keys are selected at random.

Hashing By Division

- keySpace = all ints.
- For every b, the number of ints that get mapped (hashed) into bucket i is approximately $2^{32}/b$.
- Therefore, the division method results in a uniform hash function when keySpace = all ints.
- In practice, keys tend to be correlated.
- So, the choice of the divisor b affects the distribution of home buckets.

Selecting The Divisor

- Because of this correlation, applications tend to have a bias towards keys that map into odd integers (or into even ones).
- When the divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - 20%14 = 6, 30%14 = 2, 8%14 = 8
  - 15%14 = 1, 3%14 = 3, 23%14 = 9
- The bias in the keys results in a bias toward either the odd or even home buckets.

Selecting The Divisor

- When the divisor is an odd number, odd (even) integers may hash into any home.
  - 20%15 = 5, 30%15 = 0, 8%15 = 8
  - 15%15 = 0, 3%15 = 3, 23%15 = 8
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
- Better chance of uniformly distributed home buckets.
- So do not use an even divisor.

Selecting The Divisor

- Similar biased distribution of home buckets is seen, in practice, when the divisor is a multiple of prime numbers such as 3, 5, 7, …
- The effect of each prime divisor p of b decreases as p gets larger.
- Ideally, choose b so that it is a prime number.
- Alternatively, choose b so that it has no prime factor smaller than 20.

Java.util.HashTable

- Simply uses a divisor that is an odd number.
- This simplifies implementation because we must be able to resize the hash table as more pairs are put into the dictionary.
  - Array doubling, for example, requires you to go from a 1D array table whose length is b (which is odd) to an array whose length is 2b+1 (which is also odd).

Overflow Handling

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- We may handle overflows by:
  - Search the hash table in some systematic fashion for a bucket that is not full.
    - Linear probing (linear open addressing).
    - Quadratic probing.
    - Random probing.
  - Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.
    - Array linear list.
    - Chain.

Linear Probing – Get And Put

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing – Remove

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- remove(0)

0				4				8				12				16
34		45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	45					6	23	7			28	12	29	11	30	33

Linear Probing – remove(34)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
	0	45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
0		45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
0	45					6	23	7			28	12	29	11	30	33

Linear Probing – remove(29)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
34	0	45				6	23	7			28	12		11	30	33

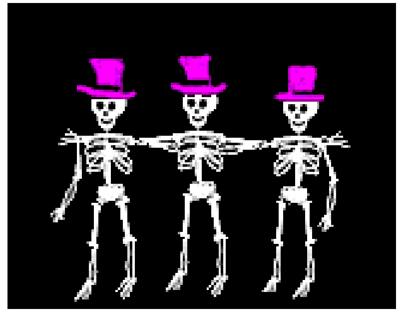
- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	0	45				6	23	7			28	12	11		30	33

0				4				8				12				16
34	0	45				6	23	7			28	12	11	30		33

0				4				8				12				16
34	0					6	23	7			28	12	11	30	45	33

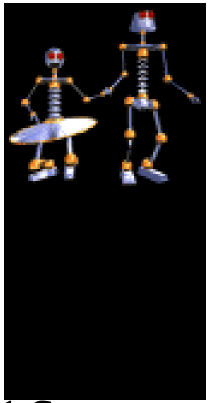
Performance Of Linear Probing



0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Worst-case get/put/remove time is Theta(n), where n is the number of pairs in the table.
- This happens when all pairs are in the same cluster.

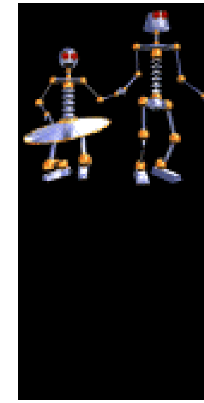
Expected Performance



0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- alpha = loading density = (number of pairs)/b.
  - alpha = 12/17.
- S_n = expected number of buckets examined in a successful search when n is large
- U_n = expected number of buckets examined in a unsuccessful search when n is large
- Time to put and remove governed by U_n.

Expected Performance



- $S_n \sim ½(1 + 1/(1 – alpha))$
- $U_n \sim ½(1 + 1/(1 – alpha)^2)$
- Note that 0 <= alpha <= 1.

alpha	S_n	U_n
0.50	1.5	2.5
0.75	2.5	8.5
0.90	5.5	50.5

Alpha <= 0.75 is recommended.

Hash Table Design

- Performance requirements are given, determine maximum permissible loading density.
- We want a successful search to make no more than 10 compares (expected).
  - $S_n \sim ½(1 + 1/(1 – alpha))$
  - alpha <= 18/19
- We want an unsuccessful search to make no more than 13 compares (expected).
  - $U_n \sim ½(1 + 1/(1 – alpha)^2)$
  - alpha <= 4/5
- So alpha <= min{18/19, 4/5} = 4/5.

Hash Table Design

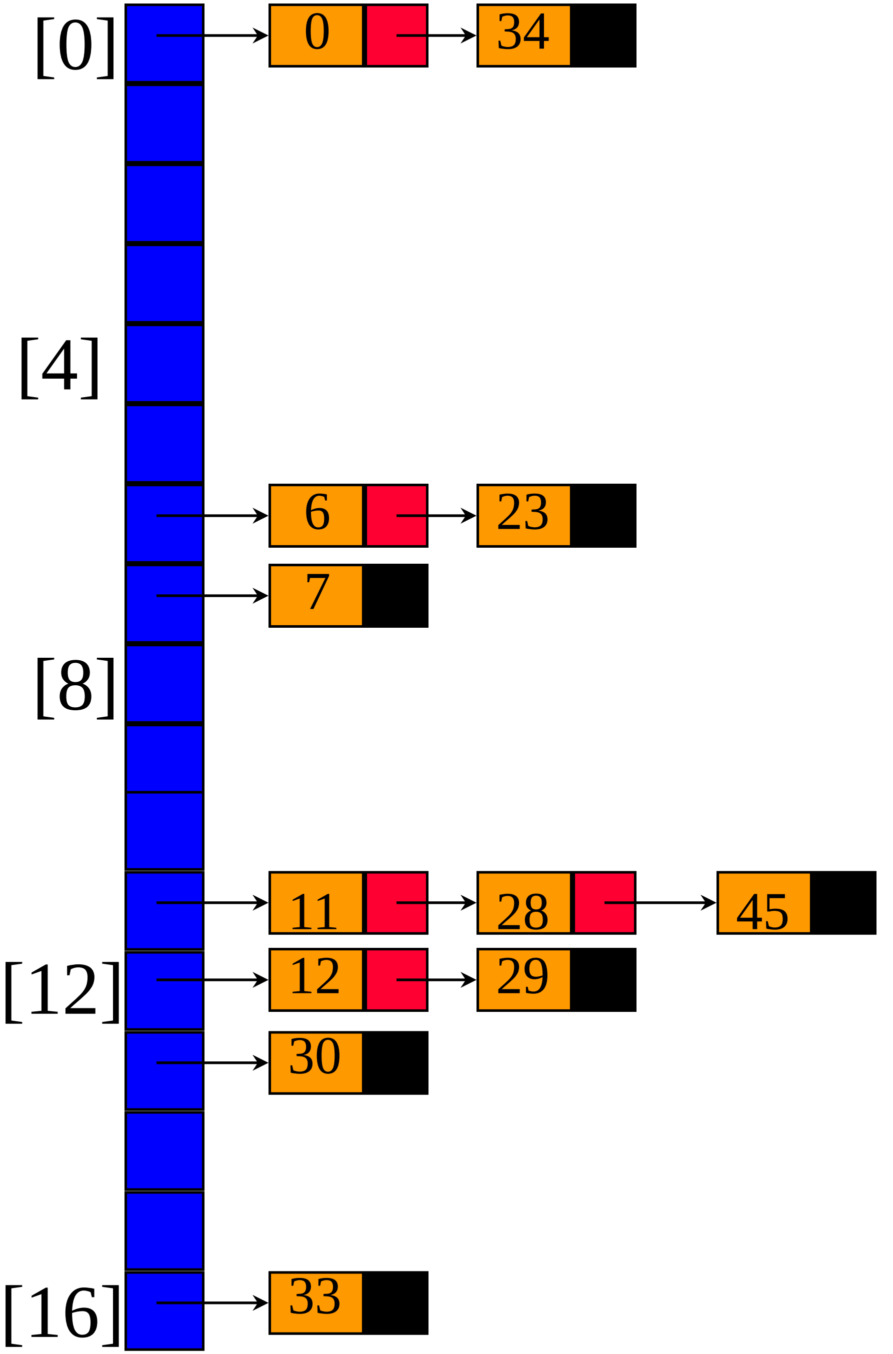
- Dynamic resizing of table.
  - Whenever loading density exceeds threshold (4/5 in our example), rehash into a table of approximately twice the current size.
- Fixed table size.
  - Know maximum number of pairs.
  - No more than 1000 pairs.
  - Loading density <= 4/5 => b >= 5/4*1000 = 1250.
  - Pick b (equal to divisor) to be a prime number or an odd number with no prime divisors smaller than 20.

Linear List Of Synonyms

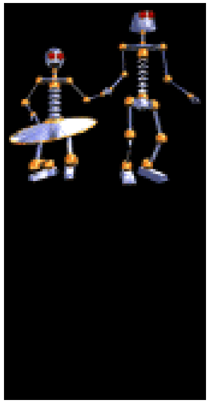
- Each bucket keeps a linear list of all pairs for which it is the home bucket.
- The linear list may or may not be sorted by key.
- The linear list may be an array linear list or a chain.

Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Home bucket = key % 17.



Expected Performance



- Note that alpha >= 0.
- Expected chain length is alpha.
- S_n ~ 1 + alpha/2.
- U_n <= alpha, when alpha < 1.
- U_n ~ 1 + alpha/2, when alpha >= 1.

java.util.Hashtable

- Unsorted chains.
- Default initial b = divisor = 101
- Default alpha <= 0.75
- When loading density exceeds max permissible density, rehash with newB = 2b+1.