INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 10

Instructor: James Pope

Email: jpope8@gmu.edu

Today

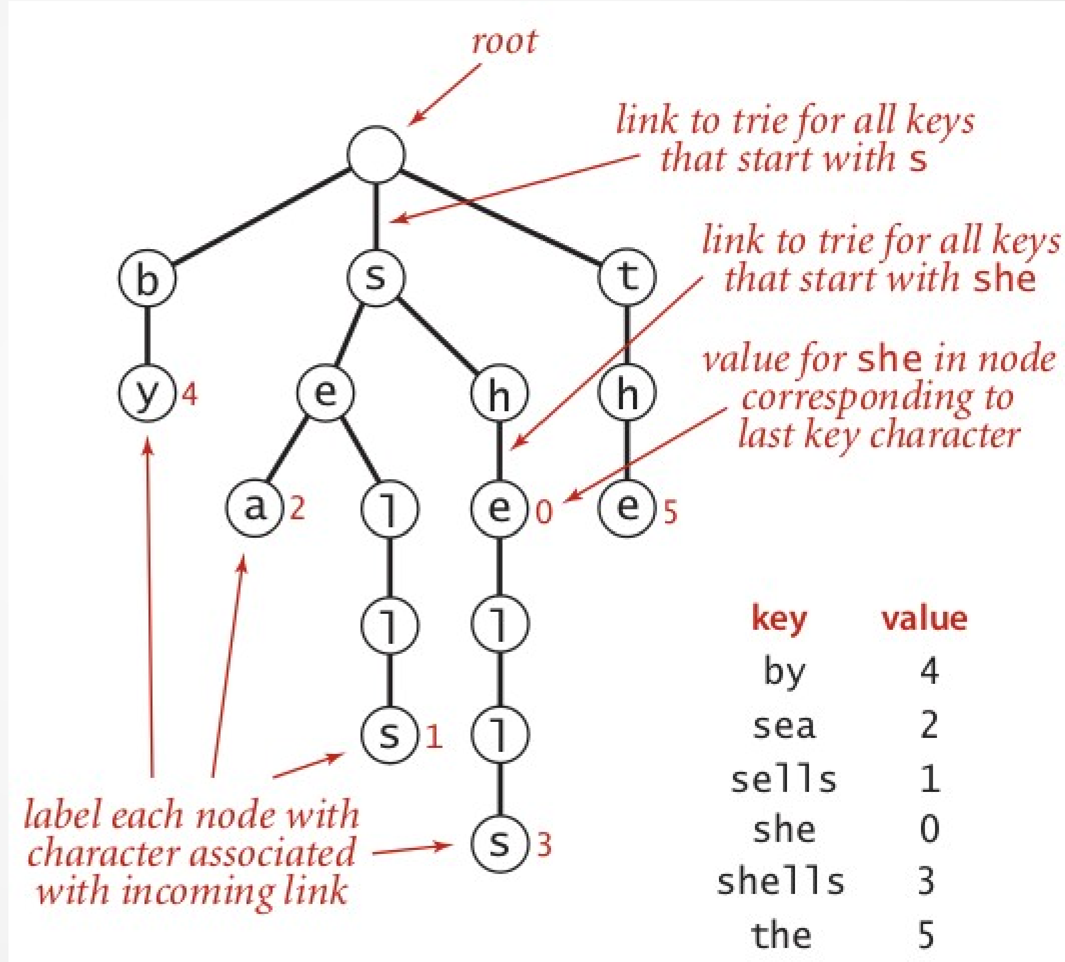
- Review Last Class
  - Trie
  - Huffman Coding
  - Hashing
- Schedule
  - More hashing
  - Linear Probing
  - Separate Chaining

Trie (pronounced “try”, from retrieval)

- Can specialize symbol table for Keys that are Strings
  - Strings made of characters
  - Allows new operations, e.g. keysWithPrefix(String s)
- Linked data structure, consists of nodes with links to other nodes.
  - Each node is pointed to once (just one parent)
  - Each node has R links, R is the alphabet size
- Though more generally used as a specialized symbol table, more concerned about binary trie for compressing coding tables

Example Trie Searches

- Hits
  - get(“shells”)
  - get(“she)
- Misses
  - get(“shell”)
  - get(“shore”)



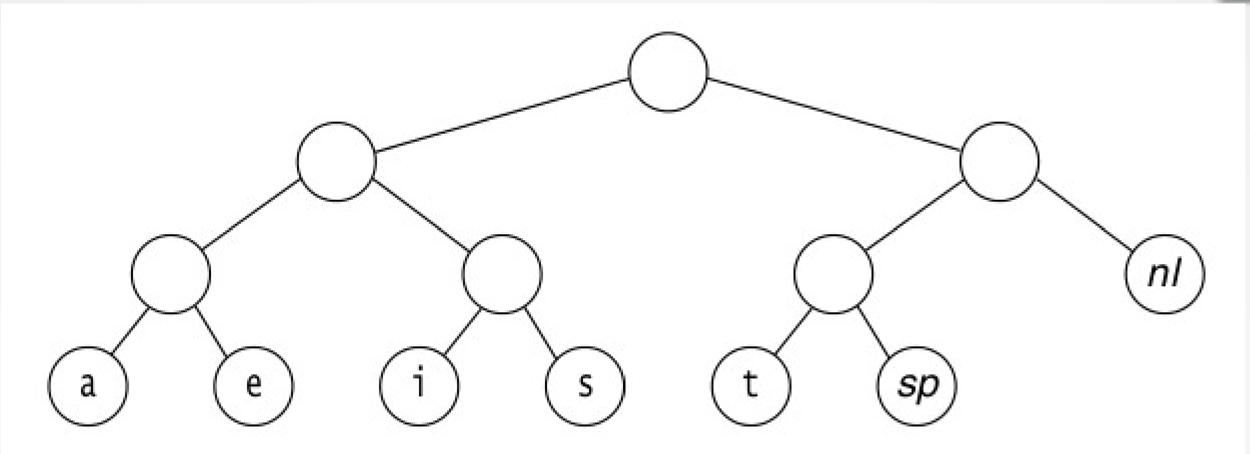
Anatomy of a trie

Compression using Prefix Codes

- General approach is to use a code to represent some symbol. To be effective for compression, the code is ideally smaller than the symbol representation.
  - Fixed length prefix codes
  - Variable length prefix codes
- Have to communicate the coding table to receiver so that they can properly decode (i.e. decompress)
- Other approaches
  - Run-length encoding

    e.g. 000000111111100 110011110100

Decoding Prefix Codes



- Suppose you receive the following and “a priori” know the coding table

  01001111000101100010000111

- What was sent? Note: No ambiguity.

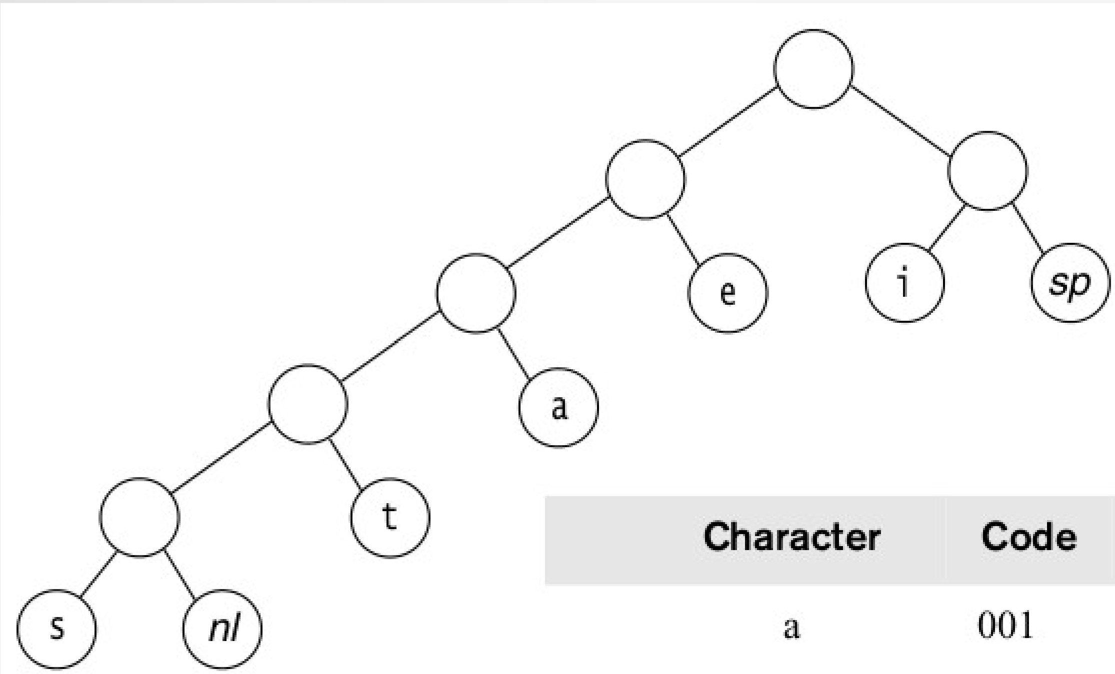
  010 011 11 000 101 100 010 001 11

Optimal Prefix Code

- Can we find an optimal (fixed or variable) prefix code? Yes, using the frequencies and variable length prefix codes.
  - Take the symbol that occurs the most and allocate the fewest number of bits per code
  - Repeat, possibly allocating more bits per code each time until the most infrequent maps to the most bits per code
- Consequences
  - Need to know frequencies of symbols in text

Example Optimal Prefix Code

Weiss Figures 12.4 and 12.5



Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
Total			146

Weighted External Path Length

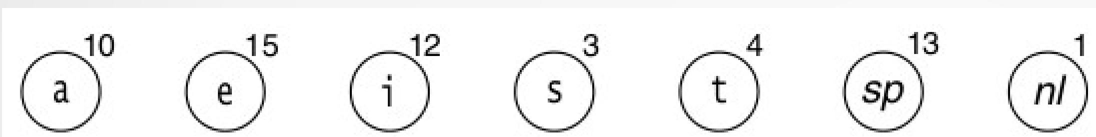
Sedgewick 5.5

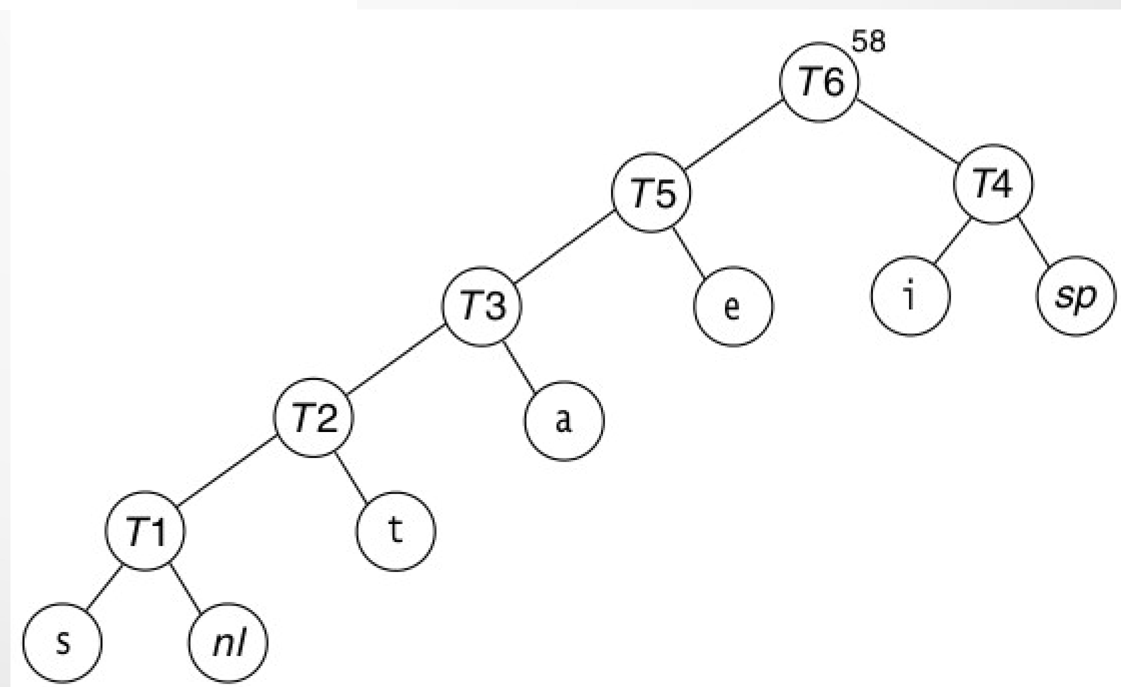
- Assume we are given a trie representing a prefix code.
- The weighted external path length is the sum of the depth of each leaf symbol multiplied by its frequency
  - W(T) = 3*10 + 2*15 + 2*12 + 5*3 + 4*4 + 2*13 + 5*1
  - Note: W(T) is the length of the encoded bit string
- Optimal: min W(T*)
- What if all frequencies are the same?
  - Balanced tree is generated, essentially fixed-length
- Other techniques (based on linear algebra) are used for images (typically have few repeated symbols).

Huffman's Algorithm

Weiss 12.1.2

- Repeatedly merges two minimum weight trees
  - Ties broken arbitrarily
  - New tree root is sum of merged subtrees





Compression using Huffman's Algorithm

- Compression steps (using alphabet of 256 symbols)
  1. Read the input text
  2. Determine frequency of each symbol (i.e. char)
  3. Build Huffman encoding trie using frequencies
  4. Build coding table from trie
  5. Write trie as a bitstring
  6. Write count of symbols in the input text
  7. Write the text as a bitstring using the coding table

Decompression using Huffman's Algorithm

- Decompression steps
  1. Read the trie (should be at beginning of bitstream)
  2. Read count of symbols encoded
  3. Use the trie to decode the bitstream

Huffman's Algorithm Summary

- Can create an optimal prefix code using symbol frequencies and generating a trie bottom up
- Uses other data structures
  - Minimum priority queue
  - Binary trie
  - Symbol table (in this case, naive implementation)
- Non-trivial combination of data structures, produces a very efficient approach for compression files
- What is compression running time? O(N + R lg(R))
  - Need to generate frequencies O(N)
  - Need binary heap O(R lg(R))

Questions?

Hashing

- Symbol Table Implementations
  - Several balanced tree implementations
    - Ordered operations
    - Guaranteed performance O(lg(N))
- Can get constant O(1) for seaching?
  - Yes, in the average case, using hash table schemes. If we give up ordered operations and somewhat performance guarantees
  - As we shall see, employs classic memory for performance trade off

Hashing

- Use the simpler SymbolTableAPI
  - Keys now have to properly implement hashCode and equals methods
- Recall symbol table motivation, want get and put to act like arrays. Idea: use an array (we'll denote as table)
- The put(Key key, Value value) operation
  - Similar to table[key] = value
- The Value get(Key key) operation
  - Similar to Value value = table[key]

Naive Hashing

- Consider all keys to be integers in range [0,65535]
  - Assume Key has hashCode operation that returns an integer in this range
- Operation put(Key key, Value value)
  - Implement table[key.hashCode()] = value
- Operation Value get(Key key)
  - Implement return table[key.hashCode()]
- Issues
  1. Can all keys be represented as integers?
  2. Key range is 32 bits, then we need table[4^32]!

Hexadecimal Notation

- All computer scientists must know how to convert between decimal, hexadecimal, and binary

Binary (base 2)	Dec (base 10)	Hex (base 16)
0100 0001	65	41
0100 0010	66	42
0100 0011	67	43
0100 0100	68	44

Binary (base 2)	Dec (base 10)	Hex (base 16)
1010 0101	?	?
1111 1111	?	?
0000 1100	?	?
1111 0100	?	?

Issue 2 Integers in range of table size

- The modulo operator as a hash function.
  - Given non-negative integer x, then x % 65536, produces number between [0,65535] regardless of how large x is.
- The modulo operator works well as a hash function provided the integer x provided is uniformly distributed.
  - Introduces new issue. When we go from a larger set A to a smaller set B via function *f*, we will have multiple elements in A mapped to same element in B.
  - In hashing terms, when multiple integer hash codes map to the same index position, a collision occurs.

Strings Hash Code

Weiss 20.2, Lafore 11.2

- Need the hashCode for String to ideally produce unique integer for each unique String. Can treat the characters as a digit in a polynomial

  $$\ldots A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0$$

  $$\ldots (((A_3X) + A_2)X + A_1)X + A_0$$ Horner's Method

- For example, “cats” is 99, 97, 116, 115
  - $99*(128^3) + 97*(128^2) + 116*(128^1) + 115*(128^0)$
  - 209,222,259
- Horner mitigates, but still have overflow issues that can produce negative values. Use a method to compute.

String's Hash Method 1

Based on Weiss Figure 20.2

```
// Note that M is the table.length
// Horner's method with 128 replaced by 37
public static int hash( String key, int M )
{
    int hashCode = 0;
    for( int i = 0; i < key.length; i++ )
    {
        hashCode = 37 * hashCode + key.charAt(i);
    }

    int hashIndex = hashCode % M;
    if( hashIndex < 0 )
        hashIndex = hashIndex + M;

    return hashIndex;
}
```

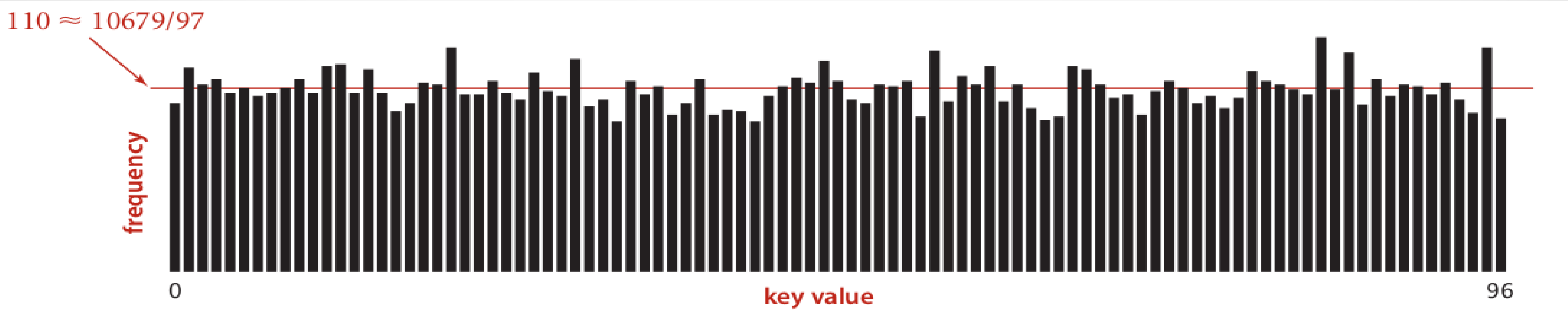
Hash Function Requirements

- Requirements for a good hash function
  1. Should be consistent, equal keys must produce same hash value
  2. Computed easily (i.e. fast)
  3. Uniformly distribute the generated integer so that index is more evenly distributed in the table
- Typical for the hashCode to produce a huge number and the modulo hash function to reduce to table index

  hugeNumber % table.length = valid index
- Other approaches use ^ (“xor”) and shifting (<<,>>) operations.

Example String (Method 2) Hash Distribution

Sedgewick 3.4



Hash value frequencies for words in *Tale of Two Cities* (10,679 keys, $M = 97$)

Object Hash Code

- Assume String, Integer, Double, etc., have evenly distributed hashCode. Composite hashCode from attributes for objects.

```
// Composite hashCode from all attributes
public class Student
{
    private String name;
    private int age;
    private double grade;
    ...
    // Caller determines M, computes index hashCode() % M
    // Note: Default is memory address, not proper hashCode!!!
    public int hashCode( )
    {
        int hash = 17; // pick prime constants
        hash = 31 * hash + name.hashCode();
        hash = 31 * hash + ((Integer)age).hashCode();
        hash = 31 * hash + ((Double)grade).hashCode();
        return hash;
    }
}
```

Hash Function Summary

- Assuming all objects have a good hashCode function.
  - Good? Users primarily define hashCode.
- Operation put(Key key, Value value)
  - table[key.hashCode() & 0x7fffffff%table.length] = value
- Operation Value get(Key key)
  - return table[key.hashCode() & 0x7fffffff%table.length]
- The hashCode method produces a huge number from attributes of the object
- The hash function takes a huge number and always produces a valid index position within the table.

Questions?

Collision Resolution

- Several ways to address, most common are:
  - Separate Chaining
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
- Commonly refer to index positions as cells. Number of keys in table is N, number of cells (i.e. array.length) is M.
- Separate Chaining
  - Linked list attached to each cell in hash table
- Open addressing
  - Use empty cells to resolve collision

Equals and HashCode Methods

- Requirement for equals and hashCode to be consistent:

  If two objects are equal

  Then must generate same hash code
- Reverse is not true, two objects with same hashCode may not be equal (i.e. they will collide when put into a hash table)
- Always override hashCode when you override the equals method (otherwise they are most certainly not consistent).

Example Equals Method

```java
// Proper equals derived from attributes, ideally immutable
public class Student
{
    private String name;
    private int age;
    private double grade;
    ...
    public boolean equals( Object obj )
    {
        // 1. performance trick, typical to check super.equals also
        if(obj == this) return true;
        // 2. type check, handles null
        if(!(obj instanceof Student)) return false;
        // 3. safe cast so can check each attribute
        Student that = (Student)obj;
        // 4. check each attribute for equality, should handle nulls
        return (
            this.name.equals(that.name) &&
            this.age   == that.age &&
            this.grade == that.grade;
    }
    ...
}
```

Linear Probing Put Operation

Weiss 20.3

- When a collision occurs, sequentially search from collision index to next index to find empty cell
  - If at end of array, wrap around to beginning
  - Care taken to not fill array full, otherwise infinite loop
- Naive analysis
  - Worse case insert would be N
  - Average case would be 1/2 N
- Fortunately, in practice
  - Performs much better

Linear Probing

Figure 20.5

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear Probing Get Operation

Weiss 20.3

- Get (find) follows similar approach
  - If collision, search sequentially until item found
  - Naive analysis similar to put
- Results in cluster of entries in sequential/contiguous order
- Direct implementation of delete not possible
  - Need the contiguous cluster, otherwise break implicit list and cannot access later items
- Delete operation workaround
  - Deleted items are marked, perhaps later removed during a rehash

Linear Probing Put Naive Analysis Issue

Weiss 20.3.1

- Naive analysis assumes the following
  1. Hash table is large
  2. Each probe is independent of the previous probe
- Second assumption is not correct
  - Load factor λ: fraction of the table that is full, N / M
  - For linear probing, 0.0 empty, 1.0 completely full
  - Note: other approaches allow load factor beyond 1.0
- Early inserts can assume probability of empty cell.
  - Probability cell is empty is $1 - \lambda$

Linear Probing Put Non-clustering Analysis

Weiss 20.3.2

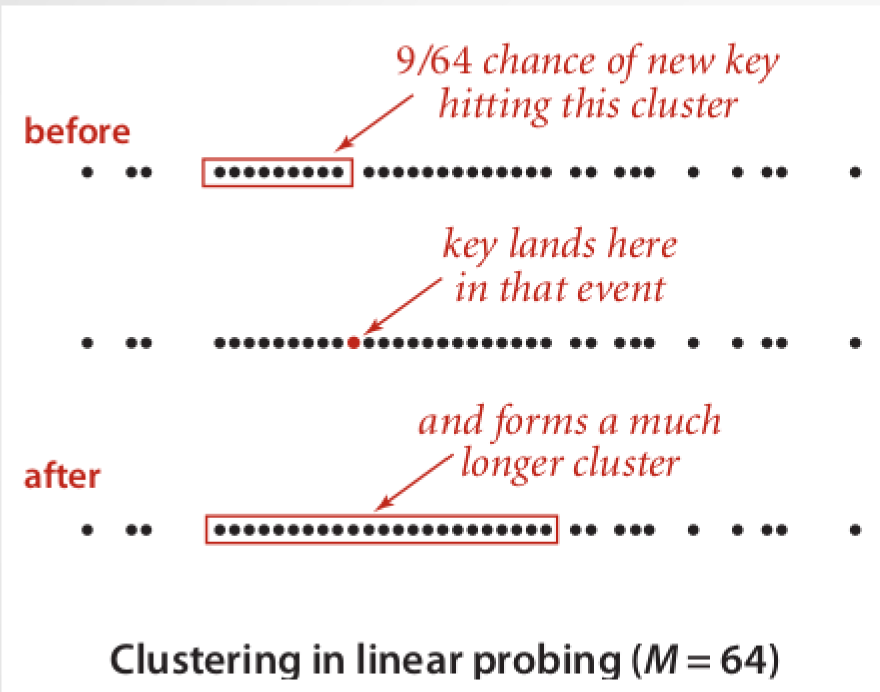
- Ignoring clustering, number of probes before finding an empty cell

  $1 / (1 - \lambda)$, e.g. $\lambda = 0.5$, then 2 probes expected

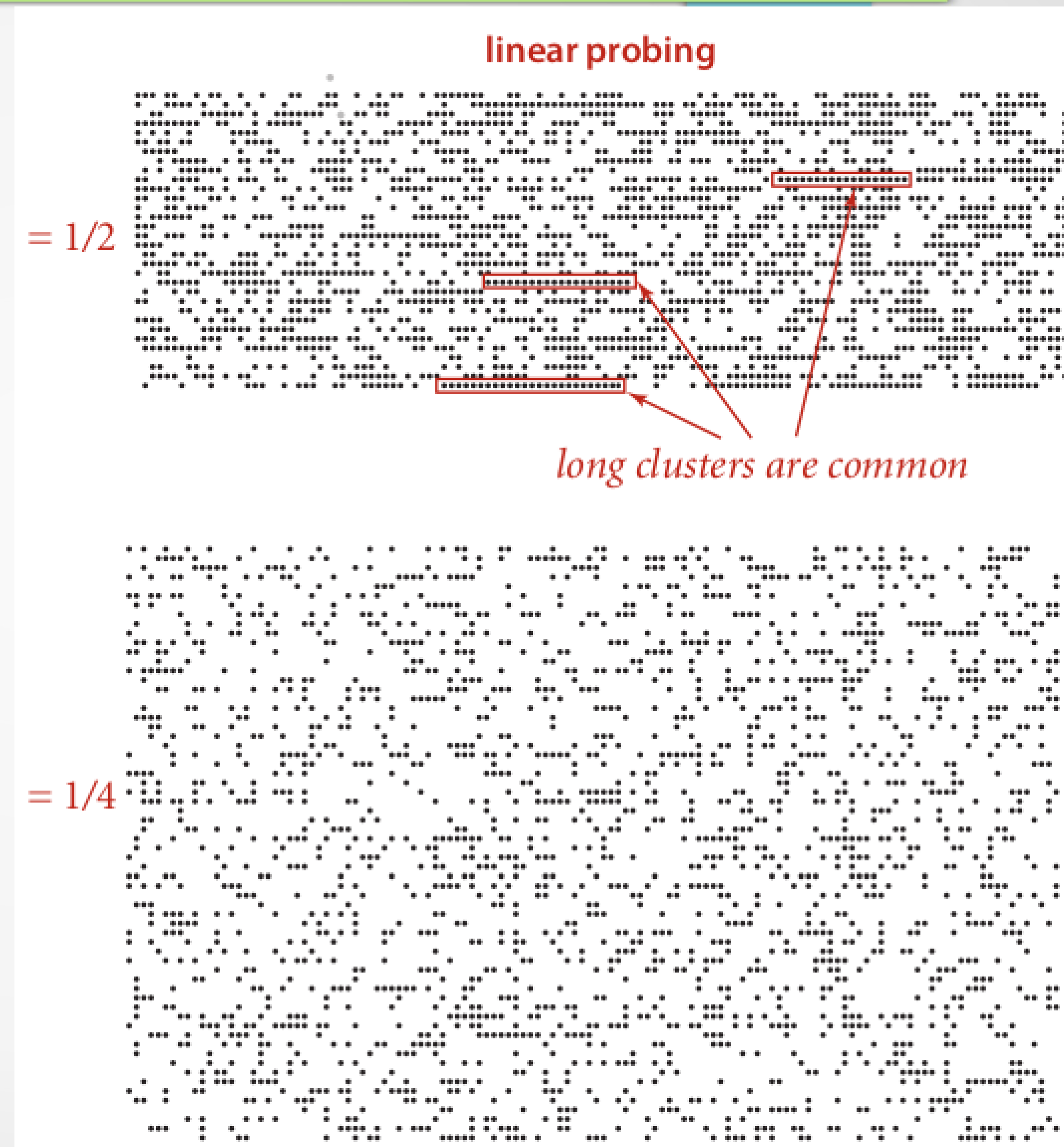
- Unfortunately, this is still incorrect
  - Primary clustering where large blocks of occupied cells are formed.
  - Subsequent collisions add to size of cluster.
  - Put operations are not independent of each other.

Linear Probing Primary Clustering

Sedgewick 3.4



Clustering in linear probing ($M = 64$)



Linear Probing Put Clustering Analysis

Weiss 20.3.2

- Clustering analysis very difficult, fortunately Knuth solved

  $1/2 \left(1 + 1 / (1 - \lambda)^2 \right)$

- Keeping the table's load factor around 0.5 is good
- As the load factor increases, expected number of probes increases dramatically. High load factor results in O(N) for both put and get operations.

Linear Probing Put Analysis Comparison

Weiss 20.3.2

- Cluster and non-cluster analysis roughly agree when λ is below 0.5

λ	Cluster #probes	Non-cluster #probes
0.10	1.12	1.11
0.20	1.28	1.25
0.30	1.52	1.43
0.40	1.89	1.67
0.50	2.50	2.00
0.60	3.63	2.50
0.70	6.06	3.33
0.80	13.00	5.00
0.90	50.50	10.00
0.95	200.50	20.00

Linear Probing Get Analysis

Weiss 20.3.3

- Get (find) operation analysis is also difficult
  - Early entries when λ is small are found in a small number of probes. Those inserted later will require a larger number of probes
  - The cost of a successful search for an entry is equal to the cost at the time the entry was inserted
- Using the put analysis, the expected number of probes for a find operation is

  $1/2\ (1 + 1 / (1 - \lambda))$

Linear Probing Rehashing

- For linear probing, low load factor is desirable. Load factor increases when new entries are inserted.
- When the hash table is created created, we generally do not know how many entries might be inserted.
  - Guess at M? Make arbitrarily large?
- How can we solve this?
  - Rehashing: Dynamically expand the hash table increasing M thereby reducing the load factor λ
  - Have to use put in new hash table because the hash function has changed
  - Classic trade off of using more memory to get better performance

Quadratic Probing

Weiss 20.4

- Quadratic probing avoids primary clustering by inserting at points away from the initial hashed index.
  - Increases quadratically, say we hash to index H, then if a collision occurs, we examine $H + i^2$, for increasing i
  - The first few cells examines, other than the initial cell, would be 1, 4, 9, 16, 25, 36, 49, etc.
- Is it possible to probe a cell twice? Is it possible to never insert even when cells are empty?
  - Yes and yes. Consider M=4 with items at 0,1. Insert an item that hashes to 0
  - Can solve by making M prime

Quadratic Probing Example

Weiss Figure 20.7

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Quadratic Probing Guarantee

Weiss Theorem 20.4

- Theorem 20.4: Using quadratic probing, if table is at most half full **and** the M is prime, then a new element can always be inserted.
  - No cell is probed twice
- To keep these guarantees, we need to dynamically expand the table when λ exceeds 0.5.
  - We cannot simply double the size, won't be prime
  - Can use function to calculate the next prime number from the current prime number efficiently (Figure 20.8)

Quadratic Probing Analysis

Weiss Theorem 20.4

- Rehash to keep table half empty $\lambda <= 0.5$
- Use nextPrime method to keep M a prime number
  - The next prime can be found efficiently $O(N^{1/2} \lg N)$
- The analysis of quadratic probing is unknown
  - Shown to perform better than linear probing in practice
  - Harder to implement? Wasted space?
- Secondary clustering
- Use double hashing scheme for collision resolution
  - Can be used to address secondary clustering

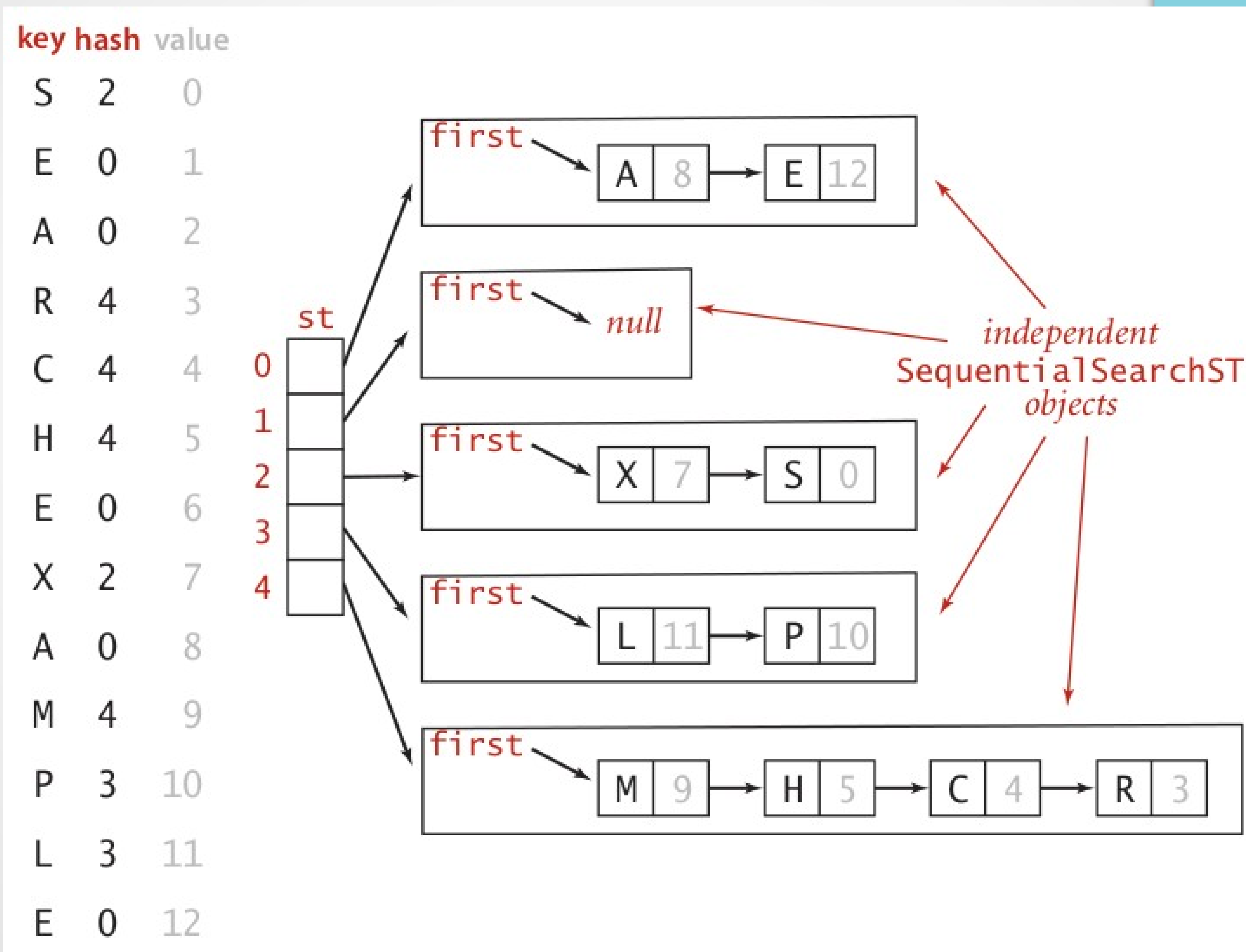
Separate Chaining

Weiss 20.5

- Space efficient alternative to quadratic probing.
- Each cell maintains linked list
  - Less sensitive to high load factors
  - Load factor λ allowed to increase beyond 1.0
- Now searching a linked list
  - This was bad? O(N)
  - Yes. However, can control length of the linked list
- Assuming uniform hashing function
  - Each list expected to have N / M entries, i.e. λ

Separate Chaining Example

Sedgewick 3.4



Separate Chaining Worse Case Analysis

Sedgewick 3.4

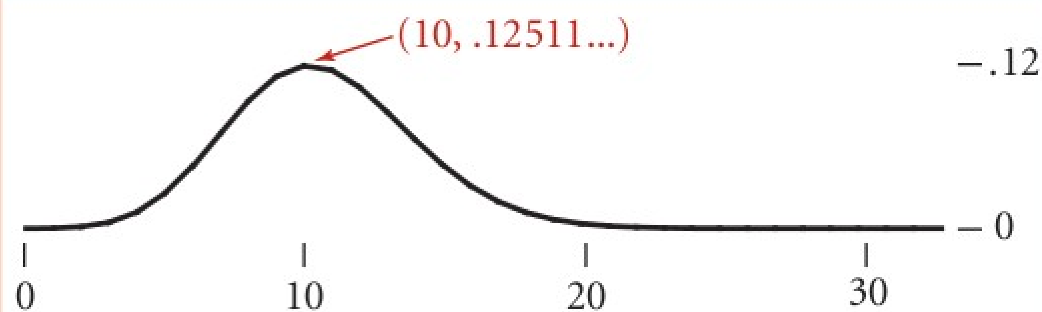
- In the worse case, the length of a list could be N
  - Put and get are N
- What about average case?
- Uniform hashing assumption
  - Uniformly and independently distribute keys between 0 and M-1 cells
- Given this, we expect each list to have N / M entries
  - How far from this expected list length?

Separate Chaining Average List Length

Sedgewick 3.4

- In separate chaining with uniform hashing assumption, the probability that the number of entries in a list is within a small constant factor of N/M is extremely close to 1.0
- Probability that hashed to list contains exactly k keys can be determined using the binomial distribution

$$\binom{N}{k}\left(\frac{1}{M}\right)^{k}\left(\frac{M-1}{M}\right)^{N-k}$$



Binomial distribution ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

Separate Chaining Average Analysis

Sedgewick 3.4

- To find an entry (get operation), expect λ / 2
  - For a small λ (e.g. 1.0) unsuccessful search is more efficient than a successful search
- To insert an entry (put operation)
  - Can insert into front of hashed list O(1)
- However, need to prevent duplicates
  - Workaround, insert at front and mask other entries in list, later during get operation can “garbage collect”
  - Pushes onus onto get. Put of get more often?
- To maintain invariant of no duplicates
  - Insert first checks list to ensure no duplicate O(N/M)

Separate Chaining Table Size M

- What table size should we use? When to rehash?
  - Low load factor does not necessarily increase performance
  - High load factor acceptable (even about 1.0) can can save memory
- Java collections uses load factor of 0.75 and will rehash
  - “Offers a good trade off between time and space costs” (HashMap)

Hash Table Summary

- Open Addressing
  - Delete not easily supported
  - Linear Probing analysis well understood, primary clustering a problem.
  - Quadratic probing solves primary clustering. Harder to implement, requires M prime.
- Separate Chaining
  - Generally more memory efficient
  - Easy to implement, including delete
- In all cases, can adjust hash table size (rehashing) to get average case O(1) for put and get operations.
  - For separate chaining, make M close to N

Symbol Table Summary

- Rule of thumb: Generally use hash table unless guaranteed performance or need ordered operations

Implementation	Worse-Case		Average-Case		Order Ops	remarks
	Search	Insert	Search	Insert		
Unordered List	N	N	N	N	No	
Ordered Array	lg N	N	lg N	N	Yes	
BST	N	N	lg N	lg N	Yes	Easy
AVL	lg N	lg N	lg N	lg N	Yes	Easy
Red-Black	lg N	lg N	lg N	lg N	Yes	Often Used*
HT Chaining	**N**	**N**	**N / M**	**N / M**	**No**	Often Used*
HT Probing	**N**	**N**	**1**	**1**	**No**	

* Good constants and relatively easy to implement, used in many libraries

Balanced Trees vs Hash Tables

- Balanced Trees
  - Necessary to handle sorted input
  - Harder to implement
  - Support ordered operations, find minimum, find all keys within some range, etc.
  - Good worse case performance guarantee
- Hash tables
  - Generally easier to implement (separate chaining)
  - Worse case poor but can trade memory to get O(1)
  - If ordered operations not needed, good choice
- Not a huge difference between O(lg N) and O(1)

Questions?

PA9

- Implement BasicSymbolTable using Separate Chaining

Free Question Time!