INFS 519 – Fall 2015

Program Design and Data Structures

Lecture 9

Instructor: James Pope

Email: jpope8@gmu.edu

Today

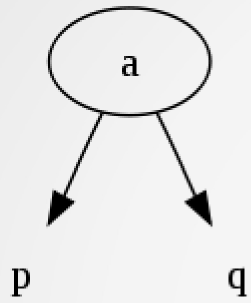
- Review Last Class
  - 2-3 Trees, Red-Black Trees, B/B+ Trees
- Schedule
  - Review Trees
  - Trie
  - Huffman Coding
  - Hashing

2-3 Tree Properties

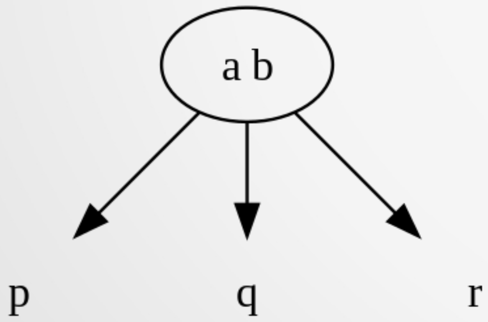
Sedgewick/Wayne 3.3

- Every node of 2-3 tree is a 2 or 3 node
  - every node has between 1 and 2 keys
  - values are stored in sorted order
  - between 2 and 3 children
  - including null links (leaves only)
- 2-3 Trees are perfectly balanced – all null links (those of the leaves) are equal distance to the root

2-3 Trees: Order Property



- 2-node
  - 1 value (a), 2 subtrees (p, q)
  - p < a < q



- 3-node
  - 2 values (a, b), 3 subtrees (p, q, r)
  - p < a < q < b < r

images: http://en.wikipedia.org/wiki/2%E2%80%933_tree

2-3 Tree Insert (Put) at Root

Sedgewick/Wayne 3.3

- Case1 Insert into a 2-node (no parent)
  - Simply add key to make it a 3-node
- Case2 Insert into a 3-node (no parent)
  - Temporarily add the key (in order) to make a 4-node
  - Take middle value, create the higher key node
  - Create two new nodes, one with the left key and one with the right key
  - Point the left child of the higher node to left key node and right child to right key node

2-3 Tree Insert (Put) with Parent

Sedgewick/Wayne 3.3

- Case3 Insert into a 3-node with 2-node parent
  - Similar to Case2, push middle key to parent
- Case4 Insert into a 3-node with 3-node parent
  - Temporarily create 4-node, split as in Case2
  - Push middle up to parent, parent now 4-node
  - Push middle of parent up, split (harder)
  - Repeat (recursion to root if necessary)

Red-Black Tree (RBT) Intuition

- BST easy to implement, 2-3 tree is balanced
  - Can we combine to get best of both?
  - "Yes", store 2-3 tree in a BST structure
- Keep 1-to-1 correspondence between the implemented BST and the logically represented 2-3 tree
  - Put operation involves several different cases
  - Remove/delete also maintain invariant
- Debatable whether RBT is easy, though it is easier than 2-3 tree and performs well

Red-Black Tree as a 2-3 Tree

Sedgewick 3.3

- Sedgewick presents RBT using 2-3 approach
- Invariants using 2-3 approach
    1) Red links lean left ("left leaning RBT")
    2) No node has two red links connected to it
    3) Tree has perfect black balance: every path from the root to a null link has the same number of black links
- Also involves several rotation cases. We will use this approach for RBT.

Red-Black Tree as a 2-3 Tree

Sedgewick 3.3

- Sedgewick presents RBT using 2-3 approach
- Invariants using 2-3 approach

    1) Red links lean left ("left leaning RBT")

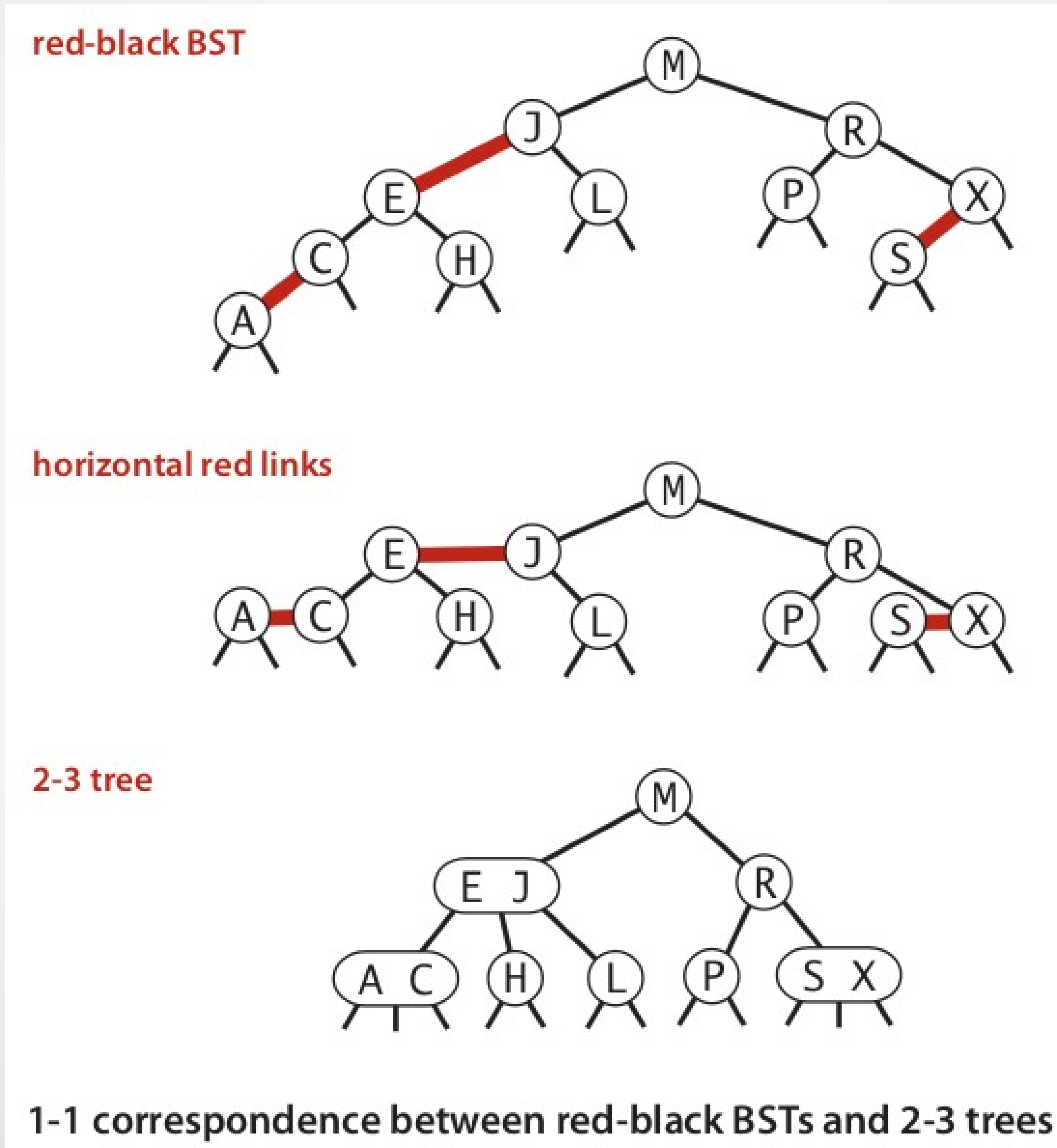
    2) No node has two red links connected to it

    3) Tree has perfect black balance: every path from the root to a null link has the same number of black links

- Also involves several rotation cases. We will use this approach for RBT.

Red-Black as 2-3 Tree and BST

Sedgewick/Wayne 3.3



Symbol Table Summary

- Generally use hash table unless guaranteed performance or need ordered operations

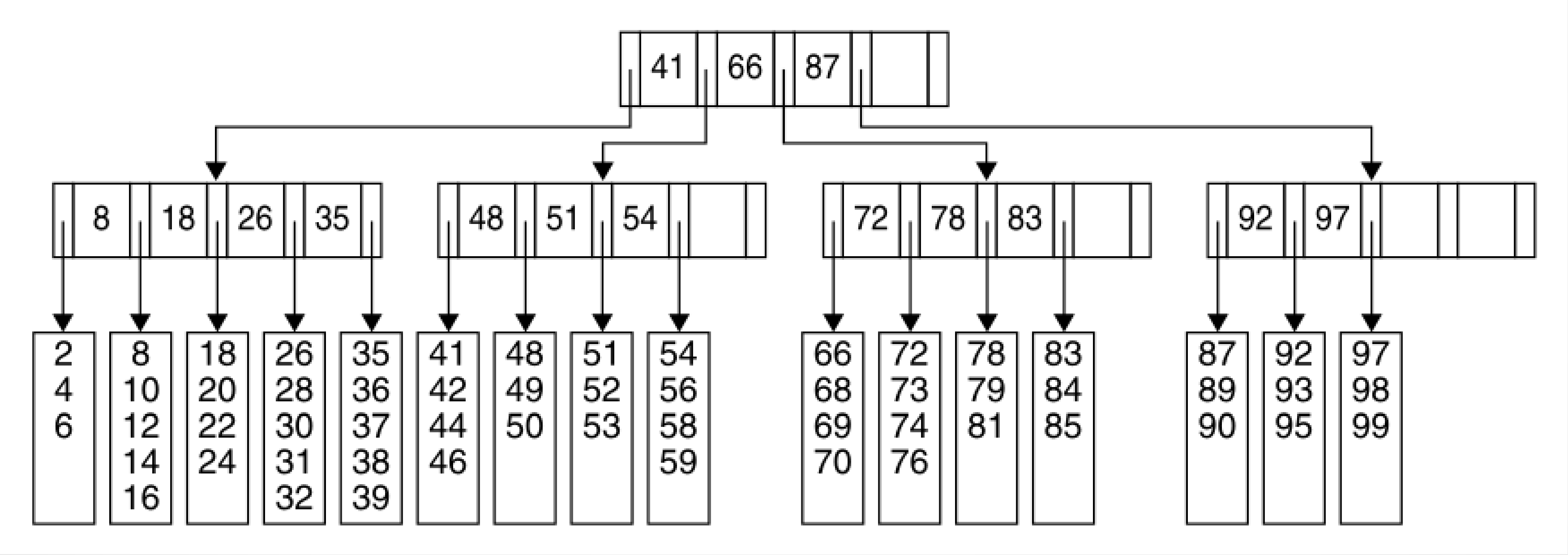
Implementation	Worse-Case		Average-Case		Order Ops	remarks
	Search	Insert	Search	Insert		
Unordered List	N	N	N	N	No	
Ordered Array	lg N	N	lg N	N	Yes	
BST	N	N	lg N	lg N	Yes	Easy
AVL	lg N	lg N	lg N	lg N	Yes	Easy
Red-Black	**lg N**	**lg N**	**lg N**	**lg N**	**Yes**	**Often Used**
HT Chaining	N	N	N / M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

* Depending on variant, will assume O(lg(N)) ~ O(1)

Huge Data Sets

- Up to now entire data structure fits in memory
  - If it's too big it won't fit into memory...
  - ... so use a tree to break it up, store on disk
- Now, we have to perform in-memory instructions intermixed with disk accesses
  - Can ~25 million instructions in one second
  - Can ~ 6 disk accesses in one second
- Given 1 million records, assuming disk access required, balanced BST
  - Requires 20 accesses, about 3.5 seconds

2 for 1: It's a list! In a tree!



B Tree Properties

- B tree of order M is an M-ary tree, invariants:

  1) Data Items are stored as leaves

  2) Non-leaf node store M-1 keys to guide search

  3) Root is leaf or between 2 and M children

  4) Non-leaf nodes (other than root) have between ceil(M/2) and M children

  5) All leaves are at same depth and have ceil(L/2) and L data items

- Note: Invariants keep tree from becoming degenerate

B Tree vs other Balanced Trees

- B Tree reduces height of tree and therefore potentially number of disk accesses
- However, if all data can fit in-memory, then other balanced trees (e.g. RBT) should be used
  - Generally constants to traverse/insert into a RBT are better than B Tree

Questions?

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

Java R-way Trie Node Representation

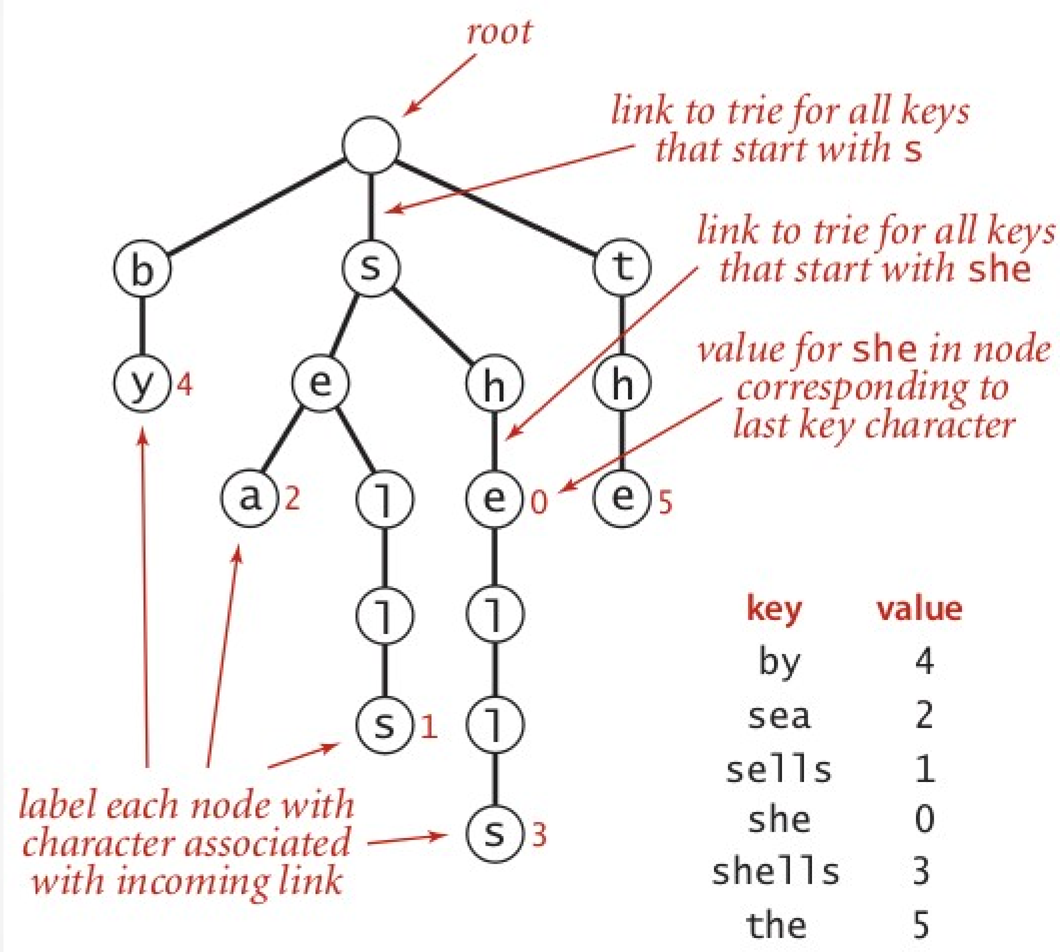
- Alphabet is a set of symbols
- Radix is the number of characters in an alphabet
- Key and Characters are implicitly represented
  - Null links exist, use memory
  - Visually drawn without null links

```
private static class Node<Value>
{
    // number of characters in the alphabet, e.g. extended ascii
    public static final int RADIX = 256;

    private Value value;
    private Node next = new Node[RADIX];
}
```

Example Trie Searches

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



Anatomy of a trie

Java Binary Trie Node Representation

- Alphabet is {0,1}, radix is two
  - Easiest to use left for 0 and right for 1
  - Value is the symbol represented (in this case a char)
- Special binary trie used to compress text
  - Symbols have frequency of occurrence in some text

```java
private static class Node<Value>
{
    private char symbol;
    private Node left;     // link implies 0
    private Node right;    // link implies 1
    private int frequency; // number symbol occurs in text
}
```

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

Compression using Prefix Codes

Weiss 12.1

- Standard Fixed Length Encoding
  - ASCII alphabet, 256 symbols, requires 8 bits per code to represent each symbol
  - Alphabet {a,e,i,s,t,sp,nl}, 7 symbols, requires 3 bits per code to represent each symbol
- In general: For standard fixed length prefix codes, requires the following bits per code to represent symbol

$$\lceil lg(radix) \rceil$$

Example Standard Coding Scheme

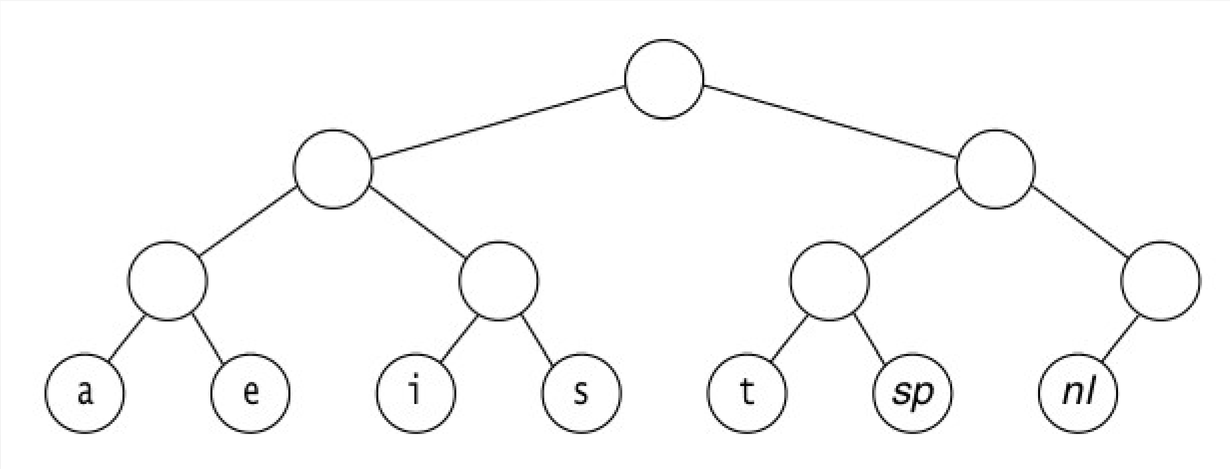
- Can save space (or transmission) using codes
  - Compress symbols ("Character") to codes
  - Decompress codes to symbols
- Example coding table alphabet size of 7

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
Total			**174**

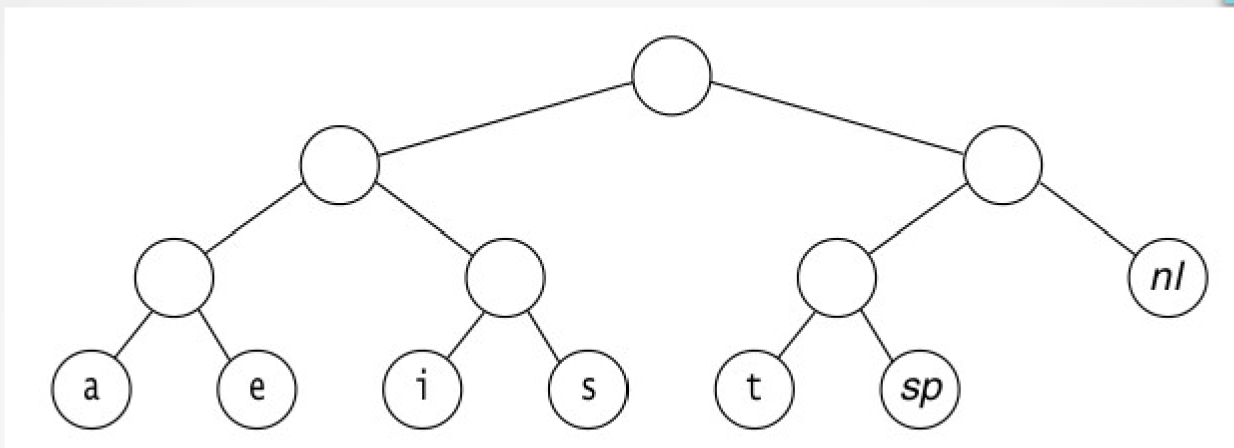
Weiss, Figure 12.1

Example Fixed-Length Prefix Code

- The coding table can be represented as a binary trie!
  - Left branch represents a 0
  - Right branch represents a 1
- Is this “optimal”? Can we find a better tree?
- Even if so, what about the frequency of the symbols, can we use to our advantage?



Decoding Prefix (Free) Code



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

  0100111100010110001000111

- What was sent? Note: No ambiguity.

  010 011 11 000 101 100 010 001 11

Variable Length Prefix Code

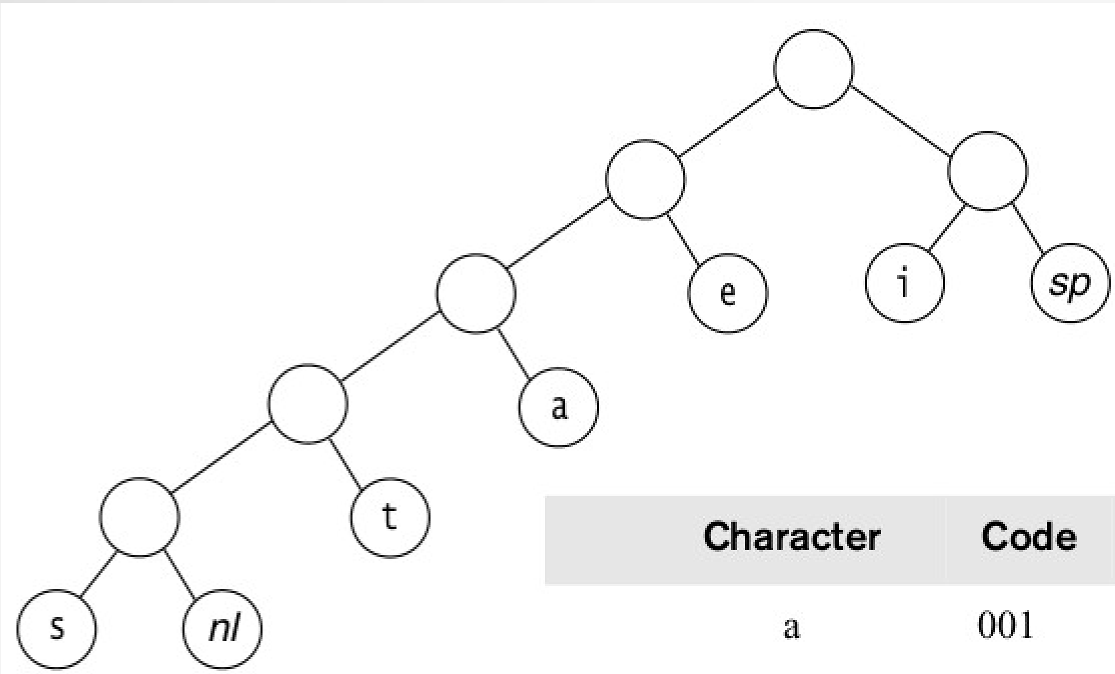
- Variable length prefix code
  - Symbols converted into codes that may have different lengths, e.g. a -> 001, e -> 01
- Desire a prefix code, i.e. each code cannot be a prefix of another code
  - This is true whenever the symbols are leaves in the trie.
- As with fixed length prefix codes, decoding is simple with no ambiguity

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

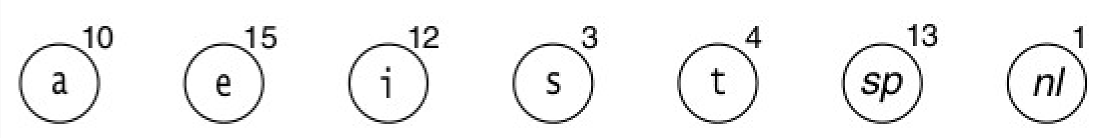


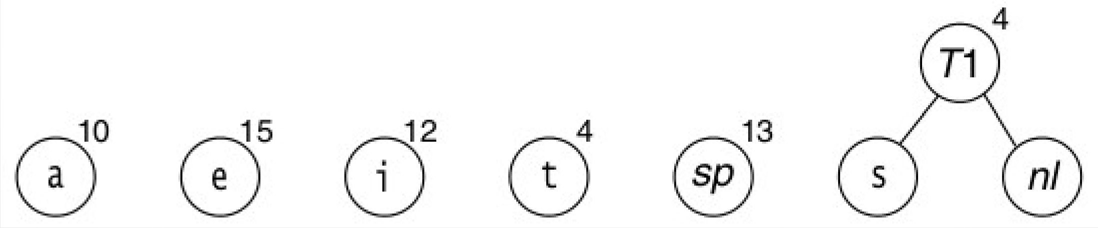
Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
Total			146

Huffman's Algorithm

Weiss 12.1.2

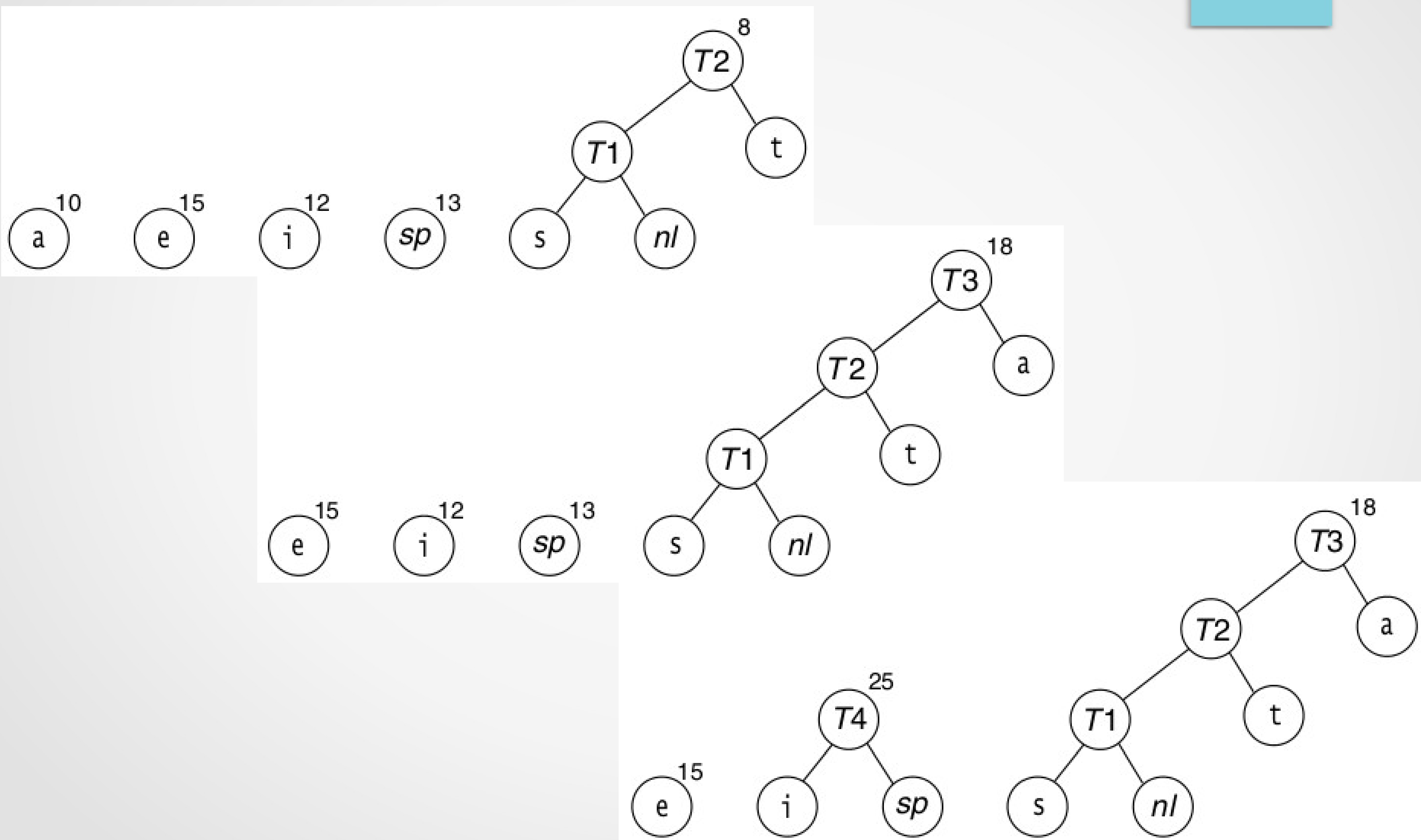
- How to we generate an optimal, prefix free, coding table?
  - Huffman's algorithm
- Repeatedly merges two minimum weight trees
  - Ties broken arbitrarily
  - New tree root is sum of merged subtrees





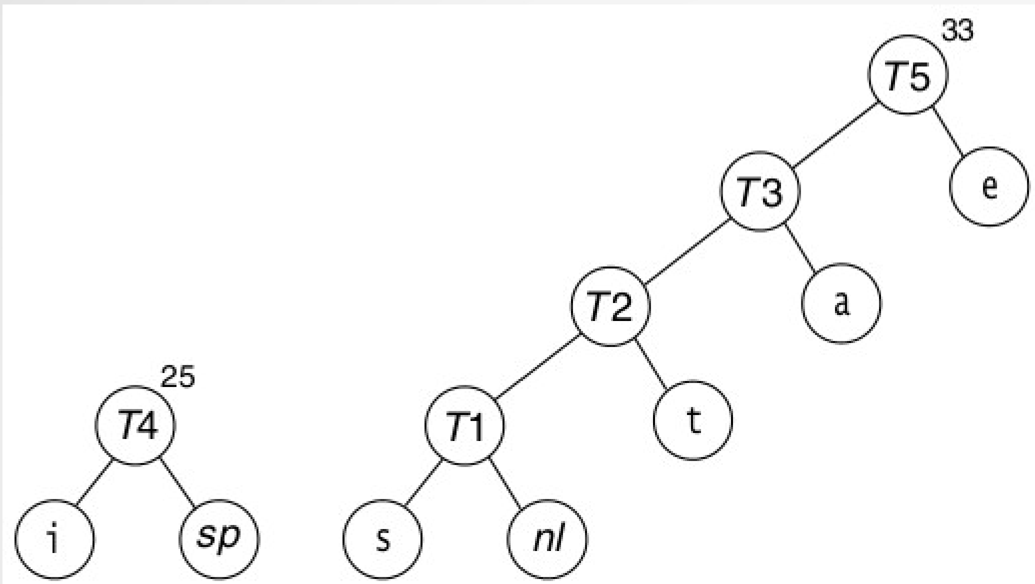
Huffman's Algorithm Example 1/2

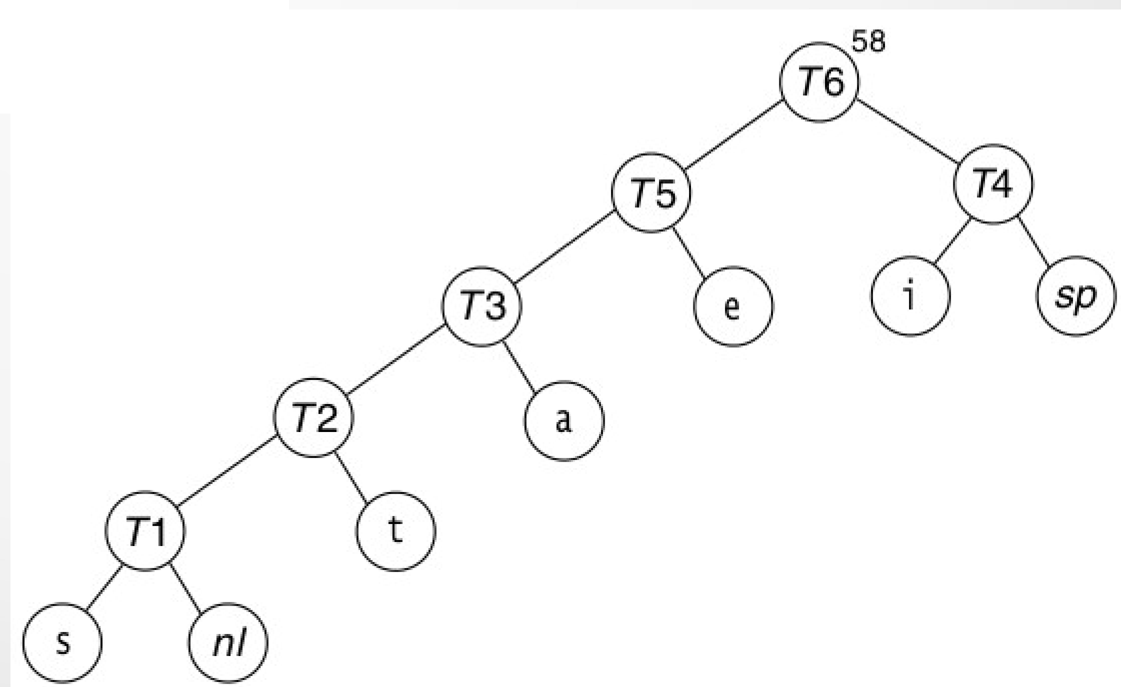
Weiss 12.1.2



Huffman's Algorithm Example 2/2

Weiss 12.1.2





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. Build coding table from trie
  3. Read count of symbols encoded
  4. Use the coding table to decode the bitstream

Build Huffman Encoding Trie

```java
// Using huffman approach, make prefix-free code
private static Node makeTrie( int[] freq )
{
    MinHeap<Node> pq = new MinHeap<Node>(freq.length);
    for( char i = 0; i < freq.length; i++ )
    {
        // Add node for each non-zero frequency to priority queue
    }

    // Special handling if only one "tree"

    // Merge all the sub-trees into one rooted tree
    while( pq.size() > 1 )
    {
        // Remove two smallest
        // Create new node with sum of their frequencies
        // Add new node back to priority queue
    }

    // last one is root of trie
    return pq.delMin();
}
```

Build Coding Table from Trie

```
// Creates coding table from the given trie
private static void makeCodingTable(
    String[] table,
    Node x, String code )
{
    // Base case
    ...

    makeCodingTable( table, x.left,  code+'0' );
    makeCodingTable( table, x.right, code+'1' );
}
```

Write Trie

```
private static void writeTrie(Node x, BitStreamOutput out)
{
    // Use preorder traversal to encode the trie
    if (x.isLeaf())
    {
        out.writeBit(true);
        out.writeBits(x.symbol, 8);
        return;
    }
    out.writeBit(false);
    writeTrie(x.left,  out);
    writeTrie(x.right, out);
}
```

Read Trie

```
private static Node readTrie( BitStreamInput in )
{
    boolean bit = in.readBit();
    if( bit )
    {
        char symbol = (char)in.readBits(8);
        return new Node(symbol, 0);
    }
    Node internalNode = new Node('\0', 0);
    internalNode.left  = readTrie( in );
    internalNode.right = readTrie( in );

    return internalNode;
}
```

Compression Code Outline

```
// 1. Calling function main reads in the input text
public static byte[] compress( char[] text )
{
    // 2. Determine the frequencies, use zero for none found
    ...
    // 3. Create the trie
    ...
    // 4. Create table code lookup using trie
    String[] table = new String[RADIX];
    makeCodingTable( table, trie, "" );

    // 5. Write the trie for the decoder and number of symbols
    BitStreamOutput out = new BitStreamOutput();
    writeTrie(trie, out);
    // 6. Write number of characters in text
    out.writeBits(text.length, 31);

    // 7. Write out text using coding table
    for( int i = 0; i < text.length; i++ )
    {
        ...
    }
    out.flush(); // Important!!! May miss last byte if not called
    return out.toArray();
}
```

Decompression Code Outline

```
public static char[] decompress(BitStreamInput in)
{
    // 1. Read in the trie
    ...

    // 2. Create table code lookup using trie
    String[] table = new String[RADIX];
    makeCodingTable( table, trie, "" );

    // 3. Read in number of symbols in original text
    int n = in.readBits(31);
    char[] decompressedText = new char[n];

    // 4. Decode remaining bitstream using coding table
    for (int i = 0; i < n; i++)
    {
        ...
    }

    return decompressedText;
}
```

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 1 Number Representation

- All information in a computer is a number which is an ordered set of 0's and 1's, e.g. integers, longs, shorts, byte, float, double, boolean, char.
- What about Strings? An array of chars.

Binary (base 2)	Dec (base 10)	Hex (base 16)	Letter (ASCII)
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
...			
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d

UTF/ASCII Coding

http://www.asciitable.com/

- Letters are encoded using some sequence of bits. ASCII is subsumed by UTF.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Issue 1 How to combine numbers

- Objects are composite of primitive variables which are themselves numbers. Need to convert an Object into a number within the range of the table.
- Consider a Student object with a String name, int age, double grade.
  - Sum the chars in name + age + grade. Range?
  - Subtract, multiply, divide have similar issues
- Solution: Hash Functions takes a large value (that would require a huge array) and maps to a smaller value in the range of the table size.

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
  - $A_3X^3 + A_2X^2 + A_1X^1 + A_0X^0$
  - $(((A_3X) + A_2)X + A_1)X + A_0$ 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

```java
// 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;
}
```

String's Hash Method 2

Based on Sedgewick 3.4

```
// Generating hashCode for a String,
// Base-R is some small prime, e.g. 31
public static int hashCode(String key, int M)
{
    int hashVal = 0;
    for( int i = 0; i < key.length; i++ )
    {
        hashVal = R * hashVal + key.charAt(i) % M;
    }
    return hashVal;
}

// General hashIndex method for all keys, including Strings
public int hashIndex( Key key )
{
    return (key.hashCode() & 0x7fffffff) % table.length;
}
```

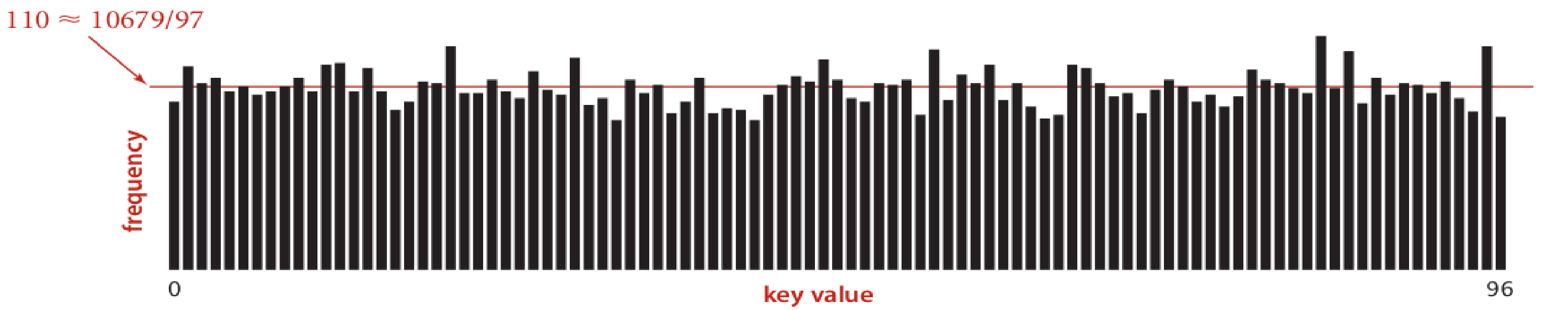
Hash Function Requirements

- 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, $M = 97$)

String HashCode Caching

Weiss Figure 20.4

- Most hashCode methods meet the speed requirement. However, long Strings would become problematic.
- Can solve by using more memory to store hashCode. Possible because Strings are immutable.

```
public final class String
{
    private int hash = 0;
    ...
    public int hashCode()
    {
        if( hash != 0 ) return hash;

        for( int i = 0; i < this.length(); i++ )
        {
            hash = 31 * hash + (int)this.charAt(i);
        }
        return hash;
    }
}
```

Object Hash Code

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

```java
// 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.

Collisions

- Several ways to address collisions, will discuss
  - Separate Chaining
  - Linear Probing
- Will discuss and analyze next week

Questions?

PA9

- Implement Huffman encoding and decoding

Free Question Time!