, 22, 23

CS 240 - Data Structures and Data Management **Spring 2017**

Lecture 21, 22, 23: July 18 - 25, 2017

Lecturer: Taylor Smith *Notes By: Harsh Mistry*

21.1 Compression

The problem : How to store and transmit data?

Judging Encoding Schemes

Encoding schemes that try to minimize $\mid C \mid$, the size of the coded text, perform data compression. We will measure the compression ratio:

$$\frac{\mid C \mid \cdot \log \mid \sum_C \mid}{\mid S \mid \cdot \log \mid \sum_S \mid}$$

Types of Data Compression

Logical vs. Physical

- Logical Compression uses the meaning of the data and only applies to a certain domain (e.g. sound recordings)
- Physical Compression only knows the physical bits in the data, not the meaning behind them

Lossy vs. Lossless

- Lossy Compression achieves better compression ratios, but the decoding is approximate; the exact source text S is not recoverable
- Lossless Compression always decodes S exactly

Encodings

Definition 21.1 ***Character Encodings:*** *Standard character encodings provide a matching from the source alphabet $\sum_S$ (sometimes called a charset) to binary strings.*

Definition 21.2 ***Variable-Length Codes:*** *Different key strings have different lengths.*

Decoding

We need a decoding algorithm mapping $\sum_C^* \to \sum_S^*$. A prefix-free code will have a fixed dictionary to support such a mapping.

Huffman Coding

- Source alphabet is arbitrary, coded alphabet is $\{0, 1\}$
- We build a binary trie to store the decoding dictionary D
- Each character of $\sum$ is a leaf of the trie

Building the best trie

1. Determine the frequency of each character $c \in \sum$ in S
2. Make $| \sum |$ height- 0 tries holding each character $c \in \sum$. Assign a "weight" to each trie: sum of frequencies of all letters in trie (initially, these are just the character frequencies)
3. Merge two tries with the least weights, new weight is their sum (corresponds to adding one bit to the encoding of each character)
4. Repeat Step 3 until there is only 1 trie left; this is D.

The tries should be stored in a min-ordered heap.

Huffman Summary

- Encoder must do lots of work
  - Building decoding trie $O(| S | + | \sum | \log | \sum |)$
  - Construct encoding dictionary mapping
  - Encode $S \to C$
- Decoding trie must be transmitted along with the coded text C
- Decoding is faster; this is an asymmetric scheme.
- The constructed trie is an optimal one that will give the shortest C (we will not go through the proof)
- Huffman is the best we can do for encoding one character at a time.

Run-Length Encoding

- Variable-length code with a
  xed decoding dictionary,but one which is not explicitly stored.
- Not a character-encoding (multiple characters represented by one dictionary-entry)
- The source alphabet and coded alphabet are both binary: f0; 1g.

Prefix-free Integer Encoding

The encoding of run-length k must be pre

x-free, because the decoder has to know when to stop reading k. We will encode the binary length of k in unary, followed by the actual value of k in binary.

The binary length of k is $len(k) = floor(\log k) + 1$ Since $k \geq 1$ we will encode $len(k) - 1$ which is at least 0/

The prefix-free encoding of the positive integer k is in two parts:

- $\log k$ copies of 0, followed by
- The binary representation of k

RLE Properties

- Compression ratio could be smaller than 1Usually, we are not that lucky:
- Method can be adapted to larger alphabet sizes

Adaptive Dictionaries

In Huffman, the dictionary is not fixed, but it is static: the dictionary is the same for the entire encoding/decoding.

Properties of adaptive encoding:

- There is an initial dictionary D_0. Usually this is fixed.
- For $i \geq 0$, D_i is used to determine the i th output character
- After writing the i'th character to output, both encoder and decoder update D_i to D_{i+1}

Note that both encoder and decoder must have the same information. Usually encoding and decoding algorithms will have the same cost.

Lempel-Ziv

Lempel-Ziv is a family of adaptive compression algorithms.

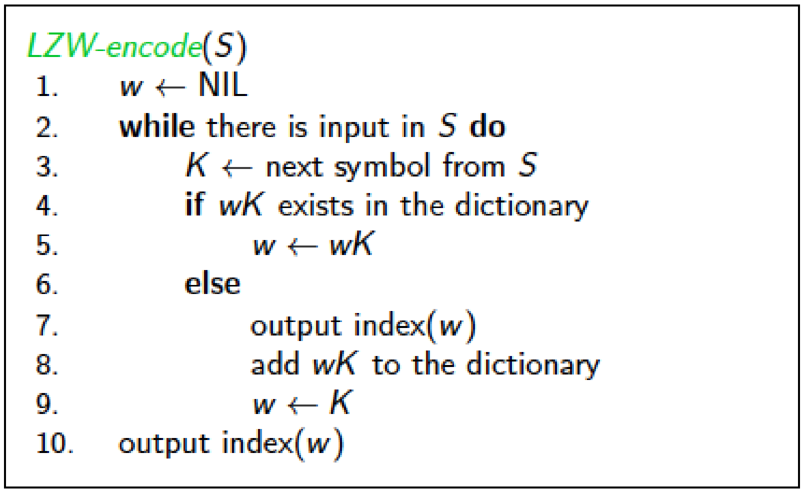
Main Idea : Each character in the coded text C either refers to a single character in $\sum_S$, or a substring of S that both encoder and decoder have already seen.

LZW Overview

- Fixed-width encoding using k bits (e.g. k = 12). Store decoding dictionary with 2k entries.
- First $| \sum_S |$ entries are for single characters, remaining entries involve multiple characters
- Encoding: after encoding a substring x of S, add xc to D where c is the character that follows x

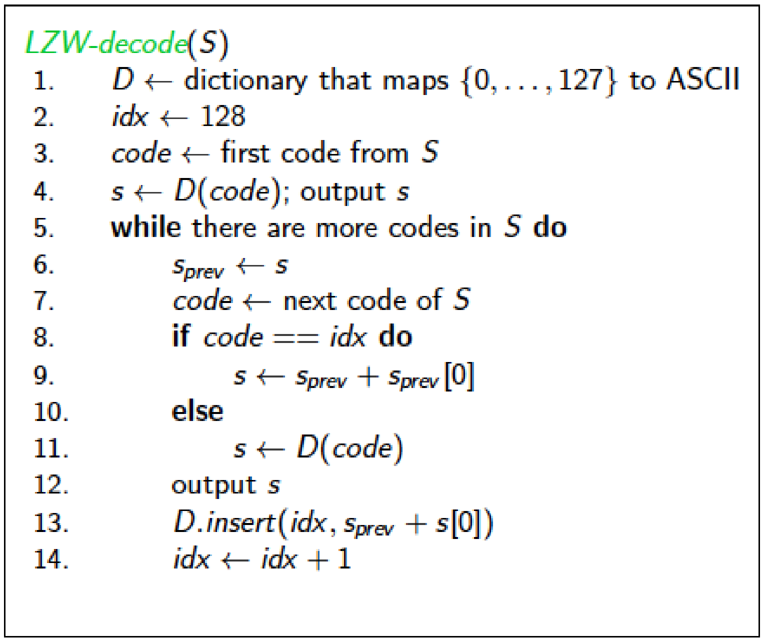
- Decoding: after decoding a substring y of S, add xc to D, where x is previously encoded/decoded substring of S, c is the first character of y
  Note: start adding to D after second substring of S is decoded

LZW encoding



```
LZW-encode($S$)
1.  $w \leftarrow$ NIL
2.  while there is input in $S$ do
3.      $K \leftarrow$ next symbol from $S$
4.      if $wK$ exists in the dictionary
5.          $w \leftarrow wK$
6.      else
7.          output index($w$)
8.          add $wK$ to the dictionary
9.          $w \leftarrow K$
10. output index($w$)
```

LZW decoding



```
LZW-decode($S$)
1.  $D \leftarrow$ dictionary that maps $\{0, \ldots, 127\}$ to ASCII
2.  $idx \leftarrow 128$
3.  $code \leftarrow$ first code from $S$
4.  $s \leftarrow D(code)$; output $s$
5.  while there are more codes in $S$ do
6.      $s_{prev} \leftarrow s$
7.      $code \leftarrow$ next code of $S$
8.      if $code == idx$ do
9.          $s \leftarrow s_{prev} + s_{prev}[0]$
10.     else
11.         $s \leftarrow D(code)$
12.     output $s$
13.     $D.insert(idx, s_{prev} + s[0])$
14.     $idx \leftarrow idx + 1$
```

Compression summary

Huffman	run-length encoding	Lempel-Ziv-Welch
variable-length	variable-length	fixed-length
single-character	multi-character	multi-character
2-pass	1-pass	1-pass
60% compression on English text	bad on text	45% compression on English text
optimal 01-prefix-code	good on long runs (e.g., pictures)	good on English text
must send dictionary	can be worse than ASCII	can be worse than ASCII
rarely used directly (part of pkzip, JPEG, MP3)	rarely used directly (fax machines, old picture-formats)	frequently used (GIF, some variants of PDF)

Text transformations

For efficient compression, we need frequently repeating characters and/or frequently repeating substrings.

Move-to-Front Encoding/Decoding

```
MTF-encode($S$)
1.  $L \leftarrow$ linked list with $\Sigma_S$ in some pre-agreed, fixed order
2.  while $S$ has more characters do
3.      $c \leftarrow$ next character of $S$
4.      output index $i$ such that $L[i] = c$
5.      Move $c$ to position $L[0]$
```

Decoding works in *exactly* the same way:

```
MTF-decode($C$)
1.  $L \leftarrow$ linked list with $\Sigma_S$ in some pre-agreed, fixed order
2.  while $C$ has more characters do
3.      $i \leftarrow$ next integer from $C$
4.      output $L[i]$
5.      Move $L[i]$ to position $L[0]$
```

Burrows-Wheeler Transform

The Burrows-Wheeler Transform is a sophisticated compression technique

- Transforms source text to a coded text with the same letters, just in a different order

- The coded text will be more easily compressible with MTF
- Compression algorithm does not make just a few ”passes” over S. BWT is a block compression method.
- Decoding is more efficient than encoding, so BWT is an asymmetric scheme.

BWT Encoding

A cyclic shift of a string X of length n is the concatenation of $X[i+1,\dots,n-1]$ and $X[0,\dots 1]$ for $0 \leq i \leq n$.

For Burrows-Wheeler, we assume the source text S ends with a special end-of-word character $ that occurs nowhere else in S.

The Burrows-Wheeler Transform proceeds in three steps:

1. Place all cyclic shifts of S in a list L
2. Sort the strings in L lexicographically
3. C is the list of trailing characters of each string in L

BWT Decoding

View the coded text C as an array of characters

- Make array of A of tuples (C[i]; i)
- Sort A by the characters, record integers in array N (Note: C[N[i]] follows C[i] in S, for all $0 \leq i \leq n$)
- Set j to index of $ in C and S to empty string
- Set j N[j] and append C[j] to S
- Repeat Step 4 until C[j] = $

BWT Overview

Encoding Cost : $O(n^2)$ (using radix sort)

- Sorting cyclic shifts is equivalent to sorting suffixes
- This can be done by traversing suffix trie
- Possible in O(n) time

Decoding cost : $O(n)$ (faster than encoding)

Encoding and decoding both use $O(n)$

21.2 External Memory

Different levels of memory

- registers
- cache L1, L2
- main memory
- external memory

Dictionaries in external memory

Tree-based data structures have poor memory locality: If an operation accesses m nodes, then it must access m spaced-out memory locations.

- In an AVL tree, $\theta(\log n)$ pages are loaded in the worst case
- Better solution : B-trees

2-3 Trees

A 2-3 tree is like a BST with additional structural properties

- Every internal node either contains one KVP and two children, or two KVPs and three children.
- The leaves are NULL (do not store keys)
- All the leaves are at the same level.

Searching through a 1-node is just like in a BST. For a 2-node, we must examine both keys and follow the appropriate path.

Insertion in a 2-3 Tree

First, we search to find the lowest internal node where the new key belongs.

If the node has only 1 KVP, just add the new one to make a 2-node.

Otherwise, otherwise order the three keys as $a < b < c$.
split the node into two 1-nodes, containing a and c, and recursively insert b into the parent along with the new link.

Deletion from a 2-3 Tree

As with BSTs and AVL trees, we first swap the KVP with its successor, so that we always delete from a leaf.

Say we're deleting KVP x from a node V:

- If V is a 2-node, just delete x.
- ElseIf V has a 2-node immediate sibling U, perform a transfer : Put the "intermediate" KVP in the parent between V and U into V, and replace it with the adjacent KVP from U.
- Otherwise, we merge V and a 1-node sibling U: Remove V and (recursively) delete the "intermediate" KVP from the parent, adding it to U.

B-trees

The 2-3 Tree is a specific type of (a,b)-tree :

An **(a,b)-tree of order M** is a search tree satisfying

- Each internal node has at least a children, unless it is the root. The root has at least 2 children.
- Each internal node has at most b children.
- If a node has k children, then it stores k 1 key-value pairs (KVPs).
- Leaves store no keys and are at the same level.

A B-tree of order M is a $(ceiling(M/2), M)$ - tree.
A 2-3 tree has a M = 3.

Height of a B-tree

$$\text{Total : } n \geq 1 + 2\sum_{i=0}^{h-1}(M/2)^i(M/2-1) = 2(M/2)^h - 1$$

Therefore height of tree with n nodes is $\theta((\log n)/(\log M))$

Analysis of B-tree operations

Assume each node stores its KVPs and child-pointers in a dictionary that supports $O(\log M)$ search, insert, and delete.

Then search, insert, and delete work just like for 2-3 trees, and each require $\theta(height)$ node operations.

$$\text{Total cost is } O\left(\frac{\log n}{\log M}\cdot(\log M)\right) = O(\log n)$$

B-tree variations

Other strategies: insert and delete without backtracking via pre-emptive splitting and pre-emptive merging.
Red-black trees: identical to a B-tree with minsize 1 and maxsize 3, but each 2-node or 3-node is represented by 2 or 3 binary nodes, and each nodes holds a "color" value of red or black.
B+ trees : All KVPs are stored at the leaves (interior nodes just have keys) and the leaves are linked sequentially.

Hashing in External Memory

Compared to Linear probing, **Extendible Hashing** which is similar to a B-tree with height 1 and max size S at the leaves is a more efficient approach.

Extendible Hashing Overview

- The **directory** (similar to root node) is stored in **internal memory** . Contains array of size 2^d, where $d \leq L$ is called the order .
- Each directory entry points to a **block** stored in external memory. Each block contains at most S items.
- To look up a key k in the directory use the d leading bits of h(k).

Extendible Hashing Details

Blocks are shared by entries in a specific manner

- Every block B stores a **local depth** $k_B \leq d$.
- Hash values in B agree on leading k_B bits
- All directory entries with the same k_B leading bits point to B
- So 2^{d-k_N} directory entries point to a block B

Searching in Extendible Hashing

Searching is done in the directory, then in a block:

- Given a key k, compute h(k).
- Leading d digits of h(k) give index in directory.
- Load block B at this index into main memory.
- Perform a search in B for all items with hash value h(k).
- Search among them for the one with key k.

Cost:

- CPU time : depends on how the blocks are organized
- Disk transfers : 1 (directory resides in internal memory)

Insertion in Extendible Hashing

insert(k, v) is done as follows:

- Search for $h(k)$ to find the proper block B for insertion
- If the B has space, then put (k, v) there.
- ElseIf the block is full and $k_B < d$, perform a *block split*:
  - Split B into two blocks B_0 and B_1.
  - Separate items according to the $(k_B + 1)$-th bit.
  - Set local depth in B_0 and B_1 to $k_B + 1$
  - Update references in the directory
  - Try again to insert
- ElseIf the block is full and $k_B = d$, perform a *directory grow*:
  - Double the size of the directory ($d \leftarrow d + 1$)
  - Update references appropriately.
  - Then split block B (which is now possible).

Extendible hashing conclusion

delete(k) is performed in a reverse manner to *insert*:

- Search for block B and remove k from it
- If block becomes too empty, then we perform a *block merge*
- If every block B has local depth $k_B \leq d - 1$, perform a *directory shrink*

But most likely just do *lazy deletion*.

Cost of *insert* and *delete*:

CPU time: Search in a block depends on the implementation
$\Theta(S)$ to do/undo one split
Directory grow/shrink costs $\Theta(2^d)$ (but very rare).

Disk transfers: 1 when no split

Summary of Extendible Hashing

- Directory is much smaller than total number of stored keys and should fit in main memory.
- To make more space, we only add one block. Rarely do we have to change the size of the directory. Never do we have to move all items in the dictionary (in contrast to normal hashing).
- Space usage is not too inefficient: can be shown that under uniform hashing, each block is expected to be 69
- Potentially extra CPU cost