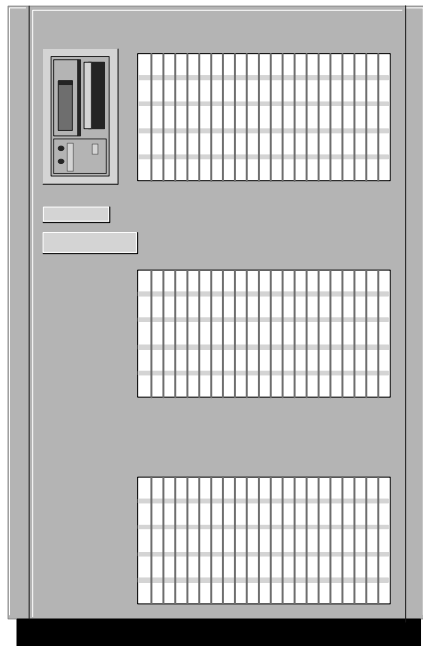


The Theory Behind the z/Architecture Sort Assist Instructions

SHARE in San Jose
August 10 - 15, 2008
Session 8121



Michael Stack
NEON Enterprise
Software, Inc.

Outline

- A Brief Overview of Sorting
- Tournament Tree Selection Sort with Replacement
- A Note on Binary Tree Implementation
- Offset Value Codes
- Tournament Tree Replacement / Selection Sort with Offset Value Codes

Outline (cont'd)

- The Hardware Assist Instructions
- What Was Omitted?
- References
- Appendix: Proofs of the Unequal Code Theorem and the Equal Code Theorem

Consumer Warning No. 1

- While the operation of CFC and UPT is not difficult to memorize, learning why they work (*this* session) and how to use them (*next* session) can be a major challenge (it was for me, anyway)
- These two sessions should help you get started, but don't expect to fully understand the details on first encounter

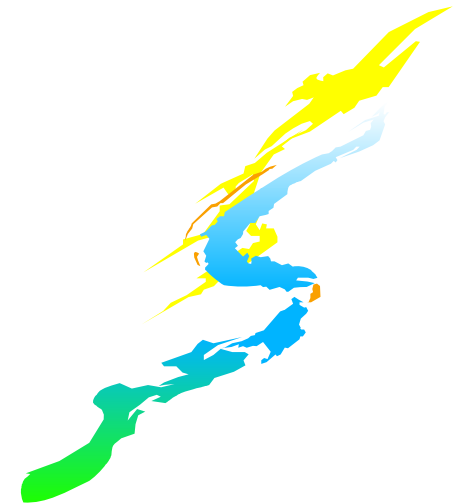
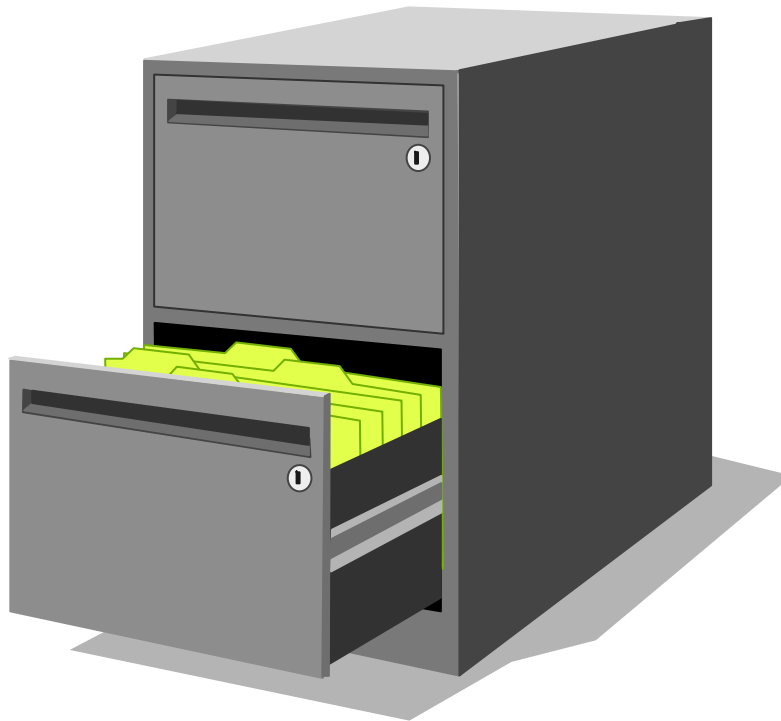
Consumer Warning No. 2

- There are two theorems in this session: the **Unequal Code Theorem** and the **Equal Code Theorem**
- Studying the logic of the proofs is probably the best way to learn how and why **Offset Value Codes** work
- So, the proofs are included as an appendix for later study

Consumer Warning No. 3

- The sort terminology in this presentation follows both Knuth⁴ and Iyer³ (see References)
- This presentation is based mostly on the paper by Iyer³, without which it could not have been prepared

A Brief Overview of Sorting



Overview of Sorting

- Multiple sort methods are used for sorting in a DBMS
 - Slow sorts are $O(N^2)$
 - Fast sorts are $O(N \lg_2 N)$
 - Fastest sorts - $O(N)$ - are Distribution Sorts, such as radix sort (good if keys are not too long)
- For more about "Big O" notation, see <http://www.nist.gov/dads/HTML/bigOnotation.html>

Overview of Sorting

- Knuth⁴ identifies five sort categories
 - Insertion (Straight, Shellsort)
 - Exchange (Bubble, Quicksort)
 - Selection (Straight, Tree, Heapsort)
 - Merge (Straight, Two-Way, List)
 - Distribution (Radix List)
- Our focus will be on a variation of selection sort called **tournament tree based replacement/selection sort**

Overview of Sorting

- In what follows, it is assumed WLOG that we are sorting in *ascending* sequence
- This means a key of lower value "wins" over a key of higher value
- For *descending* sequence, some changes must be made
- Also, we assume no duplicate keys



Tournament Tree Selection Sort with Replacement



The Theory Behind UPT



Tournament / Selection Sort

Introduction

- In the examples in this section, we will use 16 numbers chosen at random by Knuth on March 19, 1963:
 - 503 087 512 061 908 170 897 275
653 426 154 509 612 677 765 703
- Our first example will show Straight Selection
- Then we will show Quadratic Selection, an easy improvement

Tournament / Selection Sort: Straight Selection

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

503 **087** 512 **061** 908 170 897 275 653 426 154 509 612 677 765 703

061 **087** 512 503 908 170 897 275 653 426 154 509 612 677 765 703

061 087 **512** **503** 908 **170** 897 275 653 426 **154** 509 612 677 765 703

061 087 154 **503** 908 **170** 897 275 653 426 512 509 612 677 765 703

061 087 154 170 **908** **503** 897 **275** 653 426 512 509 612 677 765 703

061 087 154 170 275 **503** 897 908 653 **426** 512 509 612 677 765 703

...

For each key, K_i , we scan right for smaller keys

After comparing with all other keys, we exchange with smallest

We repeat this for each key from K_1 to K_{N-1}

Tournament / Selection Sort: Straight Selection

- With all those comparisons, is it any wonder that Straight Selection takes $2.5N^2 + 3N\lg_2 N$ units of running time (according to Knuth⁴)?
- In fact, every algorithm for finding the maximum of N elements, based on comparing pairs of elements, must make at least $N-1$ comparisons
- Happily, that rule applies only to the first step (that's important!)

Tournament / Selection Sort: Quadratic Selection

- We can improve on this by "remembering" the comparisons
- For example, we can first group the N keys into \sqrt{N} groups of \sqrt{N} elements

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

- Then we need only compare the "winners" from each group, picking a new winner at each pass

Tournament / Selection Sort: Quadratic Selection

- After a winner is chosen, we replace its value with a very large number - in this case, INF = $+\infty$) which can never "win"
- A very important point: at each level we are dealing with pointers to records to be sorted, not the records themselves

Tournament / Selection Sort: Quadratic Selection

061

170

154

612

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

(Here, $\text{sqrt}(N) = 4$ so 4 groups of 4 keys each)

Tournament / Selection Sort: Quadratic Selection

061

170

154

612

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

087

170

154

612

503 087 512 INF 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Quadratic Selection

061 170 154 612
503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

087 170 154 612
503 087 512 INF 908 170 897 275 653 426 154 509 612 677 765 703

503 170 154 612
503 INF 512 INF 908 170 897 275 653 426 154 509 612 677 765 703

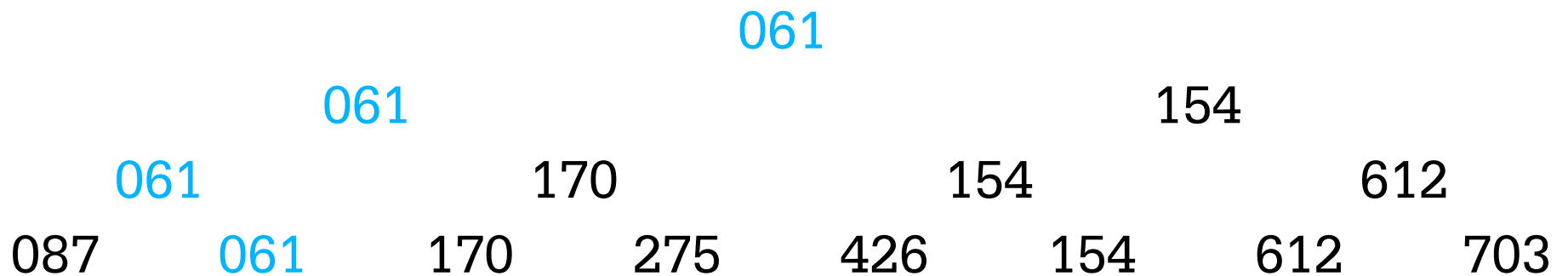
503 170 426 612
503 INF 512 INF 908 170 897 275 653 426 INF 509 612 677 765 703

Tournament / Selection Sort: Quadratic Selection

- The advantage of quadratic selection is that only the group from which the previous winner was taken needs to be re-checked
- This can be extended to cubic and quartic selection
- The ultimate is "tree selection"

Tournament / Selection Sort: Tree Selection - "Winner Tree"

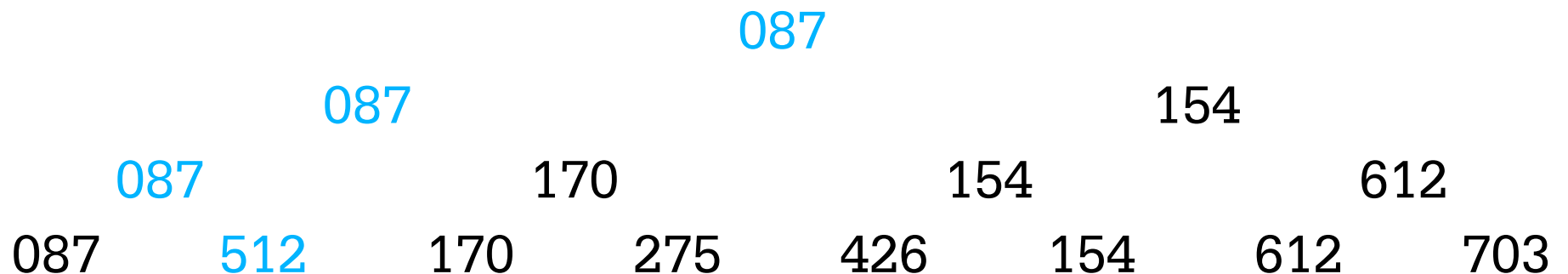
- Here is an example of tree selection showing a "winner" tree and path
 - Only the leaf nodes have keys; the internal (upper) nodes are just pointers
 - The dashed line separates leaf nodes from internal nodes



503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Winner Tree"

- When the "winner" is removed, it is replaced by a "large key" (INF here)
- At each level, only one comparison is needed to select a new winner, so $\lg_2 N$ comparisons for each key, and $N \lg_2 N$ comparisons, all told



503 087 512 **INF** 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- More useful will be a "loser tree"
- Internal nodes point to the loser of the comparison at the level below
- Why is the "loser tree" so important?
(and it is very important)
 1. Because the next winner will come from the previous winner's path!
 2. And comparisons are now along the path rather than between siblings

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- Let's construct a loser tree starting with the original keys (we call this "priming" the tree)
- We create the first internal level of the tree by comparing the keys of each pair of leaf nodes; we will show the winners in blue

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- The "loser" (highest) is pointed to by the node above the pair
- We then compare the winners at the previous level and point to the loser at the next level up

503 512 908 897 653 509 677 765

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- Now compare the previous winner nodes to get the losers at the next level
- For example, comparing the winners at level 1 (087 and 061) we have the first node (loser) at the next level

087	275	426	703
503 512	908 897	653 509	677 765

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- Finally, we have the primed loser tree and we have the winning (lowest) key (which we save in node 0)

061

154

170

612

087

275

426

703

503

512

908

897

653

509

677

765

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- The winner's path to the root is shown here and is important because it contains the next winner (*why?*)

061

154

170

612

087

275

426

703

503

512

908

897

653

509

677

765

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- Now we replace the winner (061) with a very large key (INF) so that it will always lose in future compares
- We then "pick up" INF and start up the path of 061, the previous winner
 - If the next key up is larger, we ignore it, as it cannot be a winner
 - If the next key up is smaller, we swap the bigger key we have with that smaller key

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- So, we start by replacing the first winner with INF, then start comparing and swapping up the path to the root

061

154

170

612

087

275

426

703

503

512

908

897

653

509

677

765

503 087 512 **INF** 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- Since the next winner is in this path, when we reach the root of the tree we will have the pointer to the next winner

087

154

170

612

512

275

426

703

503

INF

908

897

653

509

677

765

503 087 512 INF 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- We repeat this process until the last key is selected (all keys on the bottom row are INF)

154

170

503

612

512

275

426

703

INF

INF

908

897

653

509

677

765

503 INF 512 INF 908 170 897 275 653 426 154 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

- *You should verify that at each stage, the tree continues to be a "loser tree"*

170

426

503

612

512

275

509

703

INF

INF

908

897

653

INF

677

765

503 INF 512 INF 908 170 897 275 653 426 INF 509 612 677 765 703

Tournament / Selection Sort: Tree Selection - "Loser Tree"

■ So, we now have 061, 087, 154, 170, 275 as the first five winners

275

426

503

612

512

908

509

703

INF

INF

INF

897

653

INF

677

765

503 INF 512 INF 908 INF 897 275 653 426 INF 509 612 677 765 703

Tournament / Selection Sort: Tree Selection with Replacement

- Of course, we will probably have many more than sixteen keys to sort
- When a key is emitted, we will replace it with a new key
- When a new key is introduced that is less than the last key emitted, mark it for the next run
 - Pre-pend a sequence number on each key to identify its run

Tournament / Selection Sort: Tree Selection with Replacement

503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703

<u>Leaf Node Keys</u>				<u>Output</u>
503	087	512	061	061
503	087	512	<u>908</u>	087
503	<u>170</u>	512	908	170
503	<u>897</u>	512	908	503
<u>275</u>	897	512	908	512 (new key < previous winner)
<u>275</u>	897	<u>653</u>	908	653
<u>275</u>	<u>897</u>	<u>426</u>	908	897 (new key < previous winner)
<u>275</u>	<u>154</u>	<u>426</u>	908	908 (etc.)
<u>275</u>	<u>154</u>	<u>426</u>	<u>509</u>	(end of run)
275	154	426	509	154
275	<u>612</u>	426	509	275

etc.

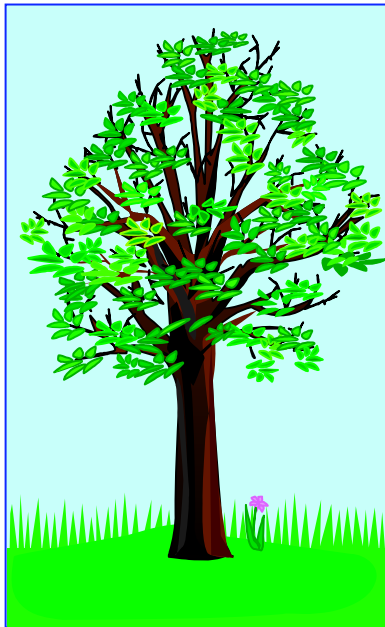
Tournament / Selection Sort: Tree Selection with Replacement

- If there are P *leaf* nodes in the tree, average run size will be $2P$ (see Knuth⁴)
- Longer sort runs reduce the number of runs which must be merged at the end
- When no more input, insert large "infinity" keys to flush remaining keys
- Note that this is really a merge operation (important for later)



A Note on Binary Tree Implementation

How Is It Possible to
Climb a Binary Tree?



Binary Tree Implementation

- In an arbitrary non-empty binary tree, each node has 0, 1 or 2 subtrees
- This requires that each node have two pointers, one for each of its possible subtrees
- But the binary trees created for Tournament / Selection sort are *complete*

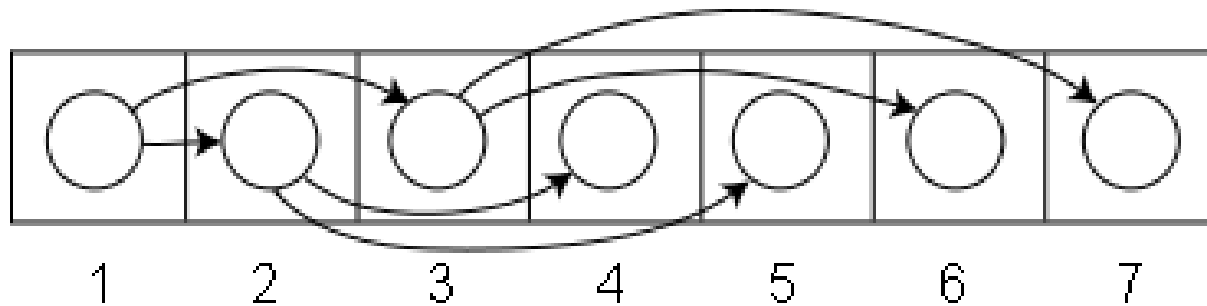
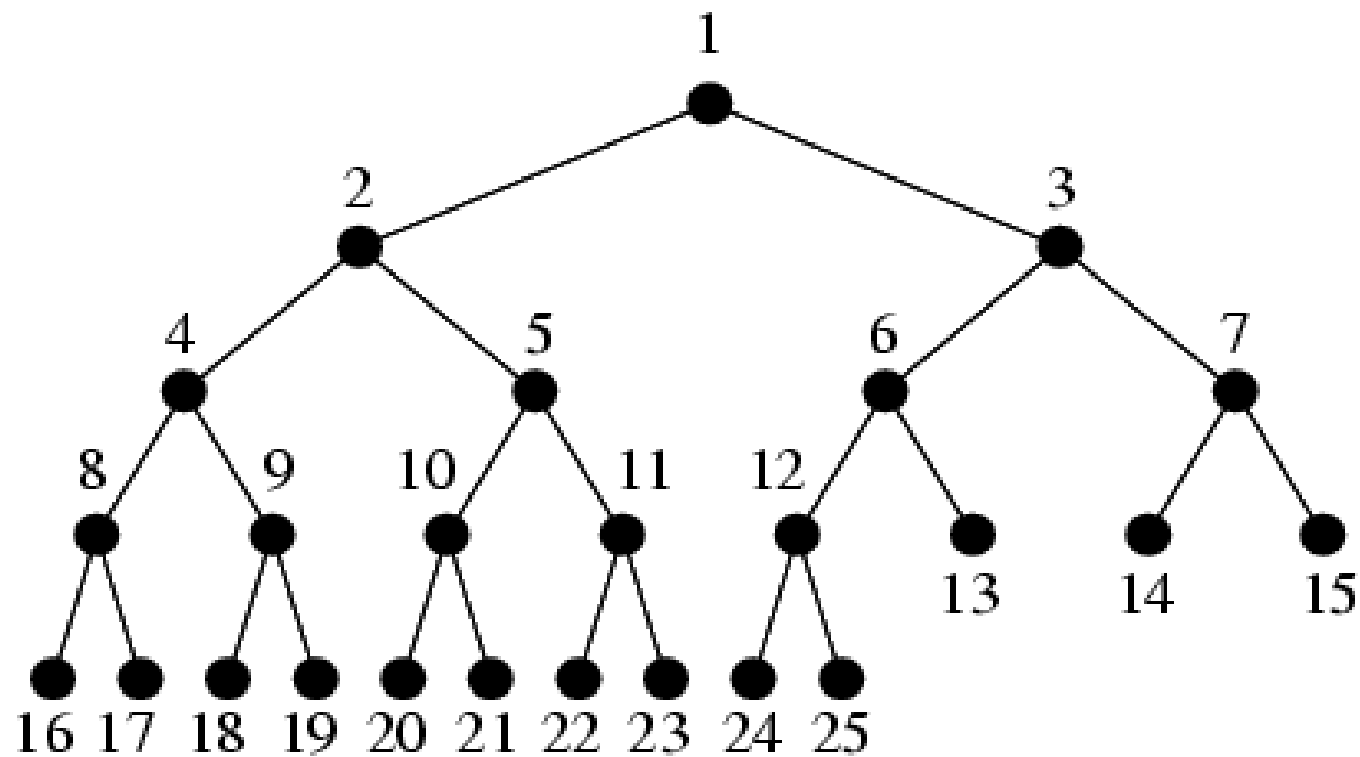
Binary Tree Implementation

- This means that each row is full, with the possible exception of the leaf (bottom) row, filled from the left, with INF's filling the rest of the row
- Since there are no omitted subtrees before the bottom row, pointers are not needed
- Instead, the location of each subtree and each parent can be computed

Binary Tree Implementation

- Such a tree can be implemented as an array, without pointers
- Each node is identified by its index $I = 1, 2, \dots, N$ for a binary tree with N nodes (root node has index 1)
- Then, given the index I of a node:
 - $\text{Parent}(I) = \text{floor}(I/2)$
 - $\text{Left child}(I) = 2I$
 - $\text{Right child}(I) = 2I+1$

Binary Tree Implementation



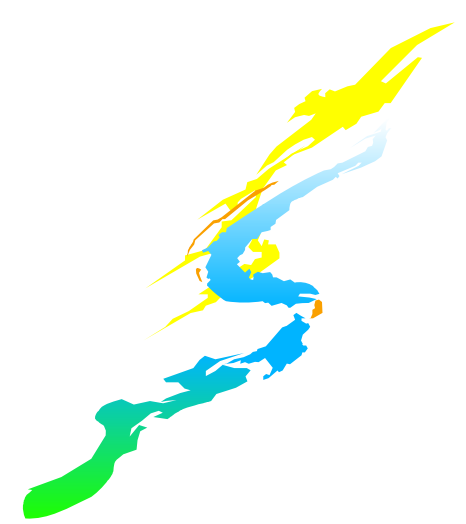
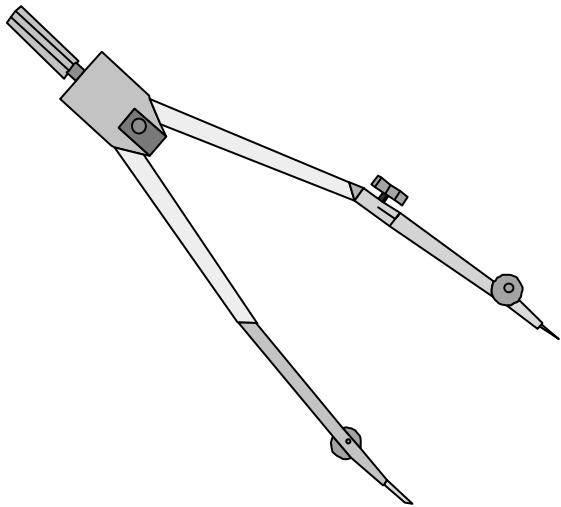
Binary Tree Implementation

- The binary tree which is the target of UPT (Update Tree) is implemented as just this kind of array
- And the UPT instruction execution follows the upward path seen in the loser tree, finding Parent nodes by taking the Floor of the current node's offset (rather than index)



Offset Value Codes

The Theory Behind CFC



Offset Value Codes

- Sort complexity of a "good" sort is $O(K N \lg_2 N)$ where K is the average key length
- Thus, long keys can have a significant impact on sort time
- Offset Value Coding attacks this cost by using cheaper (shorter) comparisons
- Invented / discovered by W M Conner¹ in 1977

Offset Value Codes: Formal Definition

- Let key A be a concatenation of characters $A(1)A(2)\dots A(p)\dots A(K)$ where K is the key length
 - $A(1)$ is the most significant position
- Let B and C also be keys of length K
- An offset value code is defined for a pair of *non-equal* keys
 - [Remember that we are defining for *ascending* sort]

Offset Value Codes:

Formal Definition

- The Offset Value Code of key B wrt key A, $OVC(B,A)$, is the concatenation of the pair offset $O(B,A)$ and value code $VC(B,A)$:
 $(O(B,A), VC(B,A))$
- $O(B,A)$ is the smallest integer $p \geq 1$ for which $B(p) \neq A(p)$
[the place where B and A first differ]
- $VC(B,A)$ is the complement of B(p)
[for ascending sorts]

Offset Value Codes: Arithmetic Examples

- Assuming that each position is taken from the domain of single digit non-negative integers
 - $OVC(426,154) = (1,5)$
 - $OVC(087,061) = (2,1)$
 - $OVC(509,503) = (3,0)$
- Key characters may be bytes, double bytes or any character set for which order and complement are well defined

Offset Value Codes: Unequal Code Theorem

■ Unequal Code Theorem for Offset Value Codes:

Given keys $B > A$ and $C > A$, and that $OVC(B,A) > OVC(C,A)$, then:

$$C > B \quad \text{and} \quad OVC(C,B) = OVC(C,A)$$

[The importance of this very powerful result cannot be overemphasized; it is used repeatedly in what follows]

Offset Value Codes: Unequal Code Theorem

- What this means is that, given two keys (B & C) which compare greater than a common third, smaller key (A):
 1. The two keys compare inversely as their OVCs against the common smaller key
 2. The OVC of the larger key (C) against the smaller key (B) is exactly the same as the OVC of the larger key (C) against the common third, smaller key (A)

Offset Value Codes: Unequal Code Theorem

- The second property will be used to avoid computing the OVC whenever possible during sorting, thus avoiding the key comparisons needed to do so
- However, two keys could compare differently against a common smaller key but their OVCs may be the same
 - The next theorem deals with that

Offset Value Codes: Equal Code Theorem

- Equal Code Theorem for Offset Value Codes:

Given keys $B > A$ and $C > A$, and

$OVC(B,A) = OVC(C,A)$, then

$O(B,C) > O(B,A)$

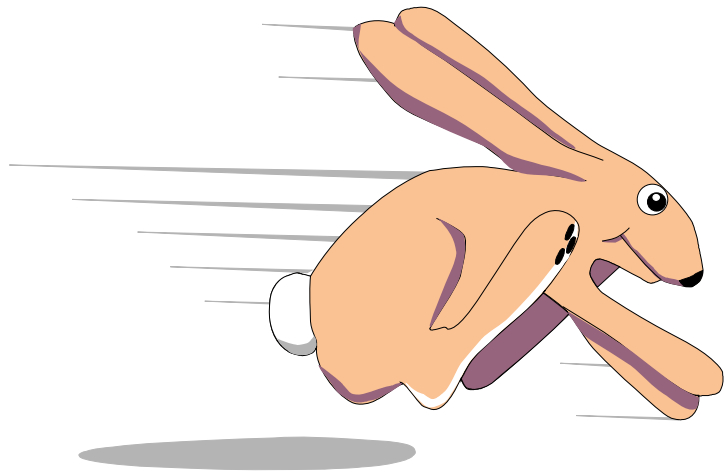
[This is a much weaker result than the one given by the Unequal Code Theorem]

Offset Value Codes: Equal Code Theorem

- This Theorem says that when the OVCs of two keys against a common smaller key are equal, we cannot use these to determine the result of comparing the two keys
- We instead need to examine less significant positions (but not the higher order positions, as they are identical)

Tournament Tree Replacement / Selection Sort with Offset Value Codes

Combining CFC & UPT



Tournament Tree Sorting with OVC

- We will now see how to save time by comparing OVCs instead of keys

- The sort process has three phases
 1. Initialization of the tree with key values and offset value codes ("priming")
 2. Run formation
 3. Tree flush

- After this, a final merge of the runs will be needed

Tournament Tree Sorting with OVC

- Whenever two OVCs against the most recent winner (smaller key) are unequal, we do not have to compare keys
- When the two OVCs are equal, we need to compare the two keys to compute the OVC of the larger key against the smaller one (but we do not need to compare the high order part that matched)

Tournament Tree Sorting with OVC: Priming the Tree

- First, we create OVCs for the leaf nodes by comparing their keys to 000

503	087	512	061	908	170	897	275
(1,4)	(2,1)	(1,4)	(2,3)	(1,0)	(1,8)	(1,1)	(1,7)

- For example, $OVC(503,000) = (1,4)$
- Then we compare OVCs pairwise to create the next level

Tournament Tree Sorting with OVC: Priming the Tree

503

(1,4)

OVC(503,087)
OVC(503,000)

512

(1,4)

OVC(512,061)
OVC(512,000)

908

(1,0)

OVC(908,170)
OVC(908,000)

897

(1,1)

OVC(897,275)
OVC(897,000)

503

087

512

061

908

170

897

275

(1,4)

(2,1)

(1,4)

(2,3)

(1,0)

(1,8)

(1,1)

(1,7)

OVC(503,000)

OVC(087,000)

OVC(512,000)

OVC(061,000)

OVC(908,000)

OVC(170,000)

OVC(897,000)

OVC(275,000)

The Unequal Code Theorem tells us that since

$OVC(087,000) = (2,1) > (1,4) = OVC(503,000)$

then $503 > 087$ (so 503 is the loser) and

$OVC(503,087) = OVC(503,000)$

Tournament Tree Sorting with OVC: Priming the Tree - Final

170

(1,8)

OVC(170,061)

OVC(170,000)

087

(2,1)

OVC(087,061)

OVC(087,000)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

512

(1,4)

OVC(512,061)

OVC(512,000)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

087

(2,1)

OVC(087,000)

512

(1,4)

OVC(512,000)

061

(2,3)

OVC(061,000)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

- We now emit 061 as the first winner and replace it with the next key, 653
- We calculate an OVC against the previous winner, $OVC(653,061)=(1,3)$, and place this OVC in the leaf
- We then move up the winner path, comparing OVCs and swapping when we find a larger OVC [OVCs *are* comparable, all against key 061]

Tournament Tree Sorting with OVC: Run Formation

170

(1,8)

OVC(170,061)
OVC(170,000)

087

(2,1)

OVC(087,061)
OVC(087,000)

275

(1,7)

OVC(275,170)
OVC(275,000)

503

(1,4)

OVC(503,087)
OVC(503,000)

512

(1,4)

OVC(512,061)
OVC(512,000)

908

(1,0)

OVC(908,170)
OVC(908,000)

897

(1,1)

OVC(897,275)
OVC(897,000)

503

(1,4)

OVC(503,000)

087

(2,1)

OVC(087,000)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

170

(1,8)

OVC(170,061)

OVC(170,000)

087

(2,1)

OVC(087,061)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

087

(2,1)

OVC(087,000)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

170

(1,8)

OVC(170,061)

OVC(170,000)

512

(1,4)

OVC(512,061)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

087

(2,1)

OVC(087,000)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

170

(1,8)

OVC(170,061)

OVC(170,000)

512

(1,4)

OVC(512,061)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

087

(2,1)

OVC(087,000)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

- We now emit 087 as the second winner and replace it with the next key, 426
- We calculate an OVC against the previous winner, $OVC(426,087) = (1,5)$, and place this OVC in the leaf
- We then move up the winner path, comparing OVCs and swapping when we find a larger OVC

Tournament Tree Sorting with OVC: Run Formation

- But we can see the path has OVCs calculated against both 061 and 087
- So, the question is whether these OVCs are, in fact, comparable
- We have already seen that $OVC(087,061) > OVC(512,061)$
- So the Unequal Code Theorem says $OVC(512,087) = OVC(512,061)$

Tournament Tree Sorting with OVC: Run Formation

- In fact, the Theorem says that unequal codes against the previous winner will always be comparable
- And whenever codes are equal, the Equal Code Theorem is applied to get an OVC against the previous winner
- So as we follow the previous winner's path, we will always have comparable OVCs

Tournament Tree Sorting with OVC: Run Formation

170

(1,8)

OVC(170,087)

OVC(170,061)

512

(1,4)

OVC(512,087)

OVC(512,061)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

426

(1,5)

OVC(426,087)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

170

(1,8)

OVC(170,087)

OVC(170,061)

512

(1,4)

OVC(512,087)

OVC(512,061)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

426

(1,5)

OVC(426,087)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

170

(1,8)

OVC(170,087)

OVC(170,061)

512

(1,4)

OVC(512,087)

OVC(512,061)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

426

(1,5)

OVC(426,087)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

426

(1,5)

OVC(426,087)

512

(1,4)

OVC(512,087)

OVC(512,061)

275

(1,7)

OVC(275,170)

OVC(275,000)

503

(1,4)

OVC(503,087)

OVC(503,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,170)

OVC(908,000)

897

(1,1)

OVC(897,275)

OVC(897,000)

503

(1,4)

OVC(503,000)

426

(1,5)

OVC(426,087)

512

(1,4)

OVC(512,000)

653

(1,3)

OVC(653,061)

908

(1,0)

OVC(908,000)

170

(1,8)

OVC(170,000)

897

(1,1)

OVC(897,000)

275

(1,7)

OVC(275,000)

Tournament Tree Sorting with OVC: Run Formation

- We now emit 170 as the third winner and replace it with the next key, 154, which forces us to start a new run
- In order to start a new run, we need a run code pre-pended to the keys
- With run code 0, the key values remain the same, but the OVCs are consistently greater

Tournament Tree Sorting with OVC: Run Formation - With Run Code

0426
(2,5)

0512
(2,4)

0275
(2,7)

0503
(2,4)

0653
(2,3)

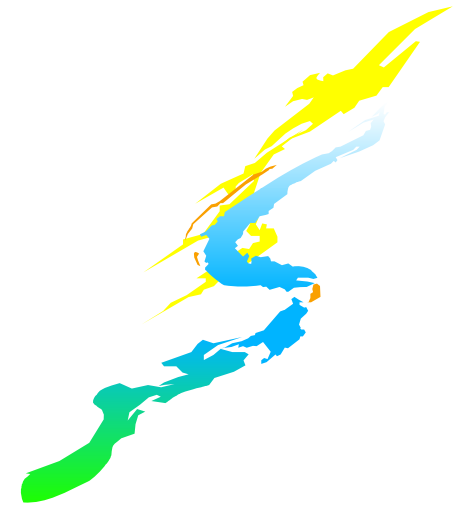
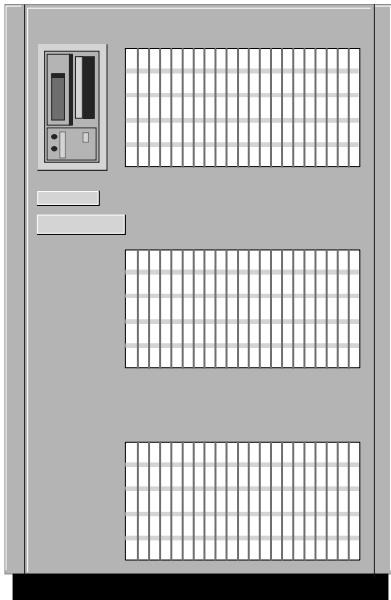
0908
(2,0)

0897
(2,1)

0503	0426	0512	0653	0908	1154	0897	0275
(2,4)	(2,5)	(2,4)	(2,3)	(2,0)	(1,9)	(2,1)	(2,7)

We can see that on the path to the root, (2,7) wins as expected

Hardware Assist Instructions



Hardware Assist Instructions

- CFC - Compare and Form Codeword
 - Compares two keys in 2- or 6-byte chunks (amode-dependent)
 - Creates a 4- or 8-byte codeword (amode-dep)
 - 2-byte offset
 - 2- or 6-byte value code (amode-dependent)
 - ▶ Value code is complemented if ascending sort
 - Offset is for unit after the one in which comparison failed
 - Easy to resume comparing after OVCs compare equal

Hardware Assist Instructions

■ UPT - Update Tree

- Program places starting OVC in register
- UPT starts at leaf, heads toward root
- UPT compares reg OVC to node OVC
 - If reg OVC < node OVC, swap OVCs
 - If reg OVC > node OVC, leave unchanged
 - If reg OVC = node OVC, stop (Equal Codes)
 - ▶ Re-compare keys w/CFC
 - ▶ Re-issue UPT at this node
- At end, register has next winner

Compare and Form Codeword

■ **label** **CFC** **$D_2(B_2)$** **[S]**

■ **Operand contents**

- $c(R1)$ = storage address of first key
- $c(R3)$ = storage address of second key
- $c(R2)$ = offset to start comparing keys
- $D_2(B_2)$ = limiting value for offset
 - bit 63 = 0 -> ascending sort
 - bit 63 = 1 -> descending sort

Compare and Form Codeword

■ Result

- Codeword formed in R2

■ Condition Codes

- 0 - Operands equal
- 1 - Operand 1 winner
- 2 - Operand 3 winner
- 3 - --

Update Tree

■ label UPT [E]

■ Operand contents

- $c(R0,R1)$ = a tree node consisting of an OVC and a leaf pointer
- $c(R2,R3)$ = a tree node where OVCs compared equal
- $c(R4)$ = A(node 0 of tree)
- $c(R5)$ = offset to node whose OVC is to be compared to OVC in R0

Update Tree

■ Condition Codes

- 0 - Equal node found in path
- 1 - No equal compare values in path
- 2 - --
- 3 - $c(R0) < 0$ and $c(R5)$ non-zero

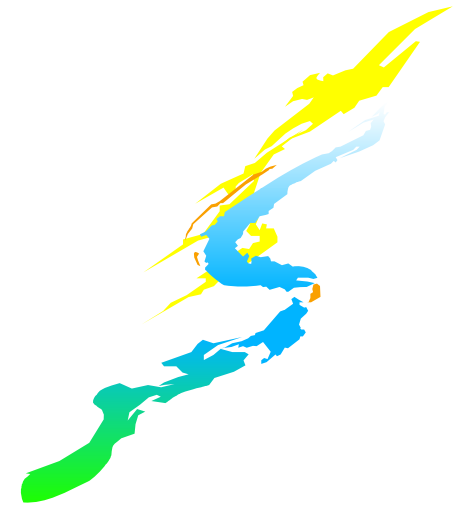
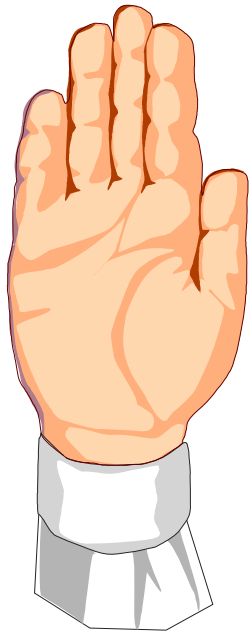
■ Result

- If $CC = 0$, equal node in (R2,R3)
- If $CC = 1$, max OVC node in (R0,R1)

Final Merge

- Once the sort is complete, the sorted runs must be merged
- This can be accomplished by using CFC and UPT once again, this time with each input sequence sorted, so no separate runs are created
- This final merge will produce the final sorted file

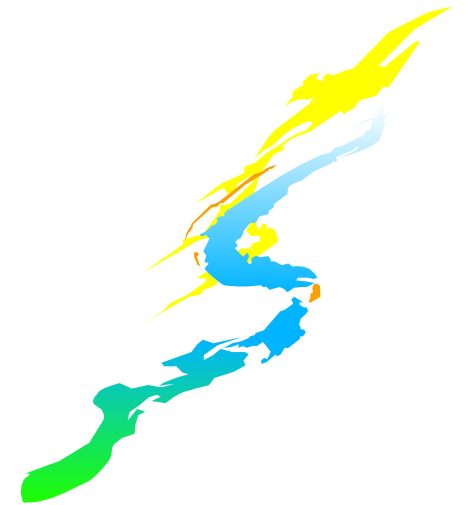
What Was Omitted?



What Was Omitted?

- Sorting in descending order
 - 2nd part of OVC is not complemented
- An example of Equal Code Theorem
 - Compare keys from position in OVC
 - Lower key is declared winner
 - OVC(loser, winner) left in node
 - This is done automatically by CFC
- Handling Duplicate Keys
 - Stable sorts are always preferred

References




References

1. Conner, W. M., Offset Value Coding, IBM Technical Disclosure Bulletin, 12-77, December 1977, pp. 2832-37, IBM Corp. [\[not seen\]](#)
2. IBM: z/Architecture Principles of Operation, SA22-7832-06, February 2008, Seventh Edition

References

3. Iyer, Balakrishna R., "Hardware Assisted Sorting in IBM's DB2 DBMS," International Conference on Management of Data COMAD 2005b, Hyderabad, India, December 20-22, 2005

4. Knuth, D. E., The Art of Computer Programming, Vol. 3, Sorting and Searching, 2nd Ed., 1998, Addison-Wesley, Reading, MA



**Appendix:
Proofs of the Unequal
Code Theorem and the
Equal Code Theorem**

Finally, Some Homework



Why bother studying the proofs?

- The best way to learn how to use the sort assist instructions is to first learn why they work
- There are some surprises, and why miss out on the fun?
- Yes, it takes some work, so don't get impatient and quit; the effort pays off in understanding

Why bother studying the proofs?

- Even more fun: maybe there's a mistake in one of the proofs
 - Could you find one if it's there?
- The proofs of these two theorems did not appear in the original Technical Disclosure Bulletin¹ and are due to Dr Balakrishna Iyer³
 - All I did was reorganize a bit, try to explain some of the more difficult steps, and add some emphasis

Proof of Unequal Code Theorem

- Theorem: Given keys $B > A$ and $C > A$, and $OVC(B,A) > OVC(C,A)$, then $B < C$ and $OVC(C,B) = OVC(C,A)$
 - Proof: Since $OVC(B,A) > OVC(C,A)$ is given, we have (by definition of OVC)
 $((O(B,A), VC(B,A)) > ((O(C,A), VC(C,A)))$
- This can happen in only two ways:
- 1) $O(B,A) > O(C,A)$
 - 2) $O(B,A) = O(C,A)$ and $VC(B,A) > VC(C,A)$

Proof of Unequal Code Theorem

Case 1

■ Given $O(B,A) > O(C,A)$

1. In positions 1 through $O(C,A)-1$, A, B, and C are identical
2. In position $O(C,A)$ A & B are identical (they first differ beyond $O(C,A)$)
3. In position $O(C,A)$ A & C are different
4. $C > A$ is given, so in position $O(C,A)$, C has a higher value than both A and B
5. Hence $C > B$, one of the parts to be proved

Proof of Unequal Code Theorem

Case 1

■ Given $O(B,A) > O(C,A)$ [cont'd]

6. In position $O(C,A)$, $C > B$

7. Hence $OVC(C,B)$

= $(O(C,B), \text{complement}(C(O(C,B))))$

= $(O(C,A), \text{complement}(C(O(C,A))))$

[since A & B are identical at $O(C,A)$]

= $OVC(C,A)$

which is the other part to be proved
for this case

Proof of Unequal Code Theorem

Case 2

■ Given $O(B,A) = O(C,A)$ and
 $VC(B,A) > VC(C,A)$

1. A, B, and C are identical in positions 1 through $O(B,A)-1$ [and $O(C,A)-1$]

2. $VC(B,A) = \text{compl}(B(O(B,A)))$

3. $VC(C,A) = \text{compl}(C(O(C,A)))$
 $= \text{compl}(C(O(B,A)))$

4. $VC(B,A) > VC(C,A)$ is given

5. So, $\text{compl}(B(O(B,A)))$
 $> \text{compl}(C(O(B,A)))$

Proof of Unequal Code Theorem

Case 2

- Given $O(B,A) = O(C,A)$ and
 $VC(B,A) > VC(C,A)$ [cont'd]
- 6. Thus, $B(O(B,A)) < C(O(B,A))$
- 7. At the highest order position where B and C differ from A, the value in B is smaller than the value in C.
- 8. Hence, $B < C$, which is one of the parts to be proved in this case
- 9. This also shows that B and C are different at the highest position where A and B differ from C

Proof of Unequal Code Theorem

Case 2

■ Given $O(B,A) = O(C,A)$ and
 $VC(B,A) > VC(C,A)$ [cont'd]

10. In other words,

$$O(C,B) = O(C,A) = O(B,A)$$

11. So $OVC(C,A)$

$$= (O(C,A), \text{compl}(C(O(C,A))))$$

$$= (O(C,B), \text{compl}(C(O(C,B))))$$

$$= OVC(C,B)$$

■ QED

Proof of Equal Code Theorem

■ Given keys $B > A$ and $C > A$, and $OVC(B,A) = OVC(C,A)$, then

$$O(B,C) > O(B,A)$$

1. Given that $OVC(B,A) = OVC(C,A)$
2. Therefore, $O(B,A) = O(C,A)$
3. Hence, A , B & C are identical from positions 1 through $O(B,A)-1$ [also through $O(C,A)-1$, trivially]
4. The value components are also equal:
 $VC(B,A) = VC(C,A)$

Proof of Equal Code Theorem

■ [cont'd]

5. Hence B and C are identical in position $O(B,A)$ as well

6. If they differ, they must differ first in a lower order position

7. Thus $O(B,C) > O(B,A)$

■ QED