9 Supplementary material

9.1 *A leaf-to-root pass in a tournament tree*

The following diagrams illustrate a leaf-to-root pass in a traditional tree-of-losers priority queue or tournament tree that merges 12 sorted runs. This tree-of-losers priority queue or tournament tree is adapted from Knuth's "The Art of Computer Programming" [K 98].

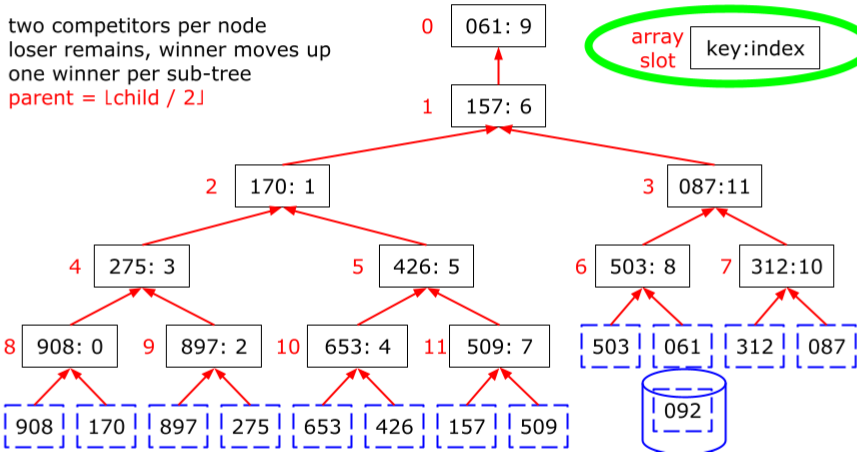

Figure 11. Initial tournament tree.

Figure 11 shows the initial tournament tree, perhaps immediately after initialization with the lowest key value in each of the 12 merge inputs. The dashed boxes along the bottom represent the current keys from the runs to be merged; the solid boxes are tree nodes in the tree-of-losers priority queue. The example node in the top-right corner explains what all the numbers mean. An index is here the run identifier within the merge logic, values 0-11. The root of this tournament tree is in array slot 0. It holds the overall smallest key value, 61, which came from merge input 9. The next key value in merge input 9 is also shown; this key value will start the next leaf-to-root pass. Notice that key value 157 is above key value 87. This is because 157 emerged as the winner from the left subtree and 87 is only the runner-up in the right subtree. (The runner-up can be anywhere along the leaf-to-root path of the overall winner.)

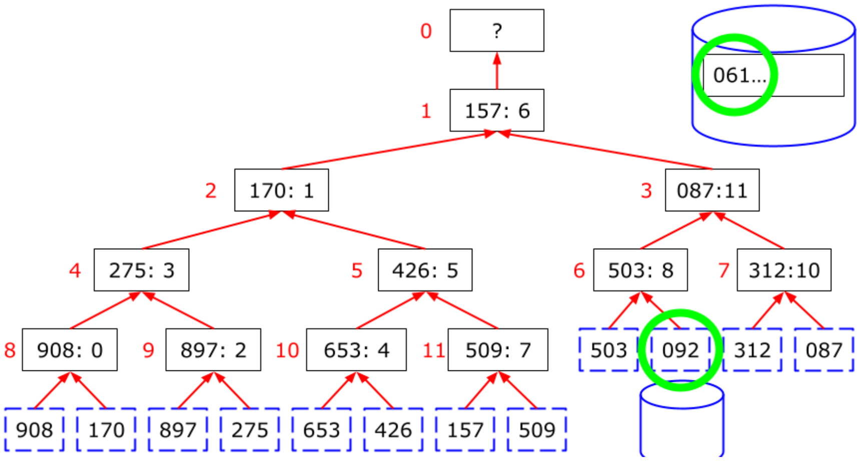

Figure 12. Tournament tree after moving key values.

Figure 12 shows the tournament tree after the merge logic moves key value 61 from the tree root to the merge output and obtains a new key value from input 9, the origin of key value 61. This successor key starts a new leaf-to-root pass at the same leaf as the prior winner it replaces in the tree-of-losers priority queue.

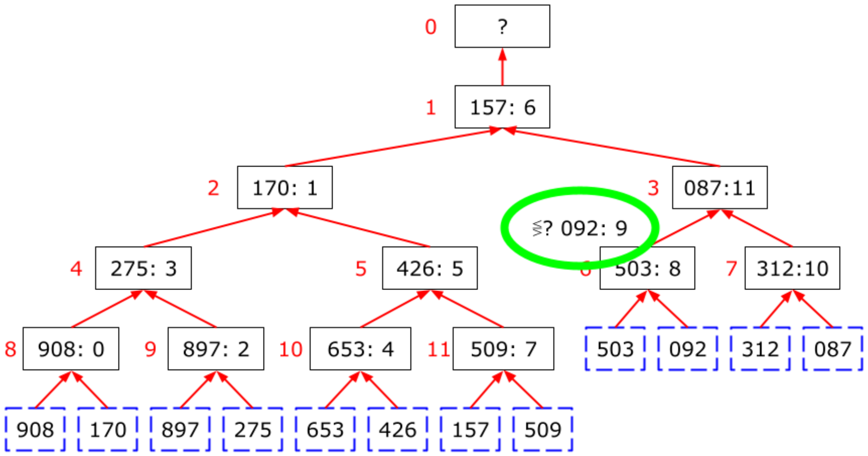

Figure 13. Tournament tree starting a leaf-to-root pass.

Figure 13 shows the first step in the leaf-to-root pass. This leaf-to-root pass bubbles the next lowest key value to the root in $\log_2(F)$ steps for merge fan-in and priority queue capacity F. In the leaf node, key value 92 wins against key value 503; therefore, key value 92 moves up to the parent node.

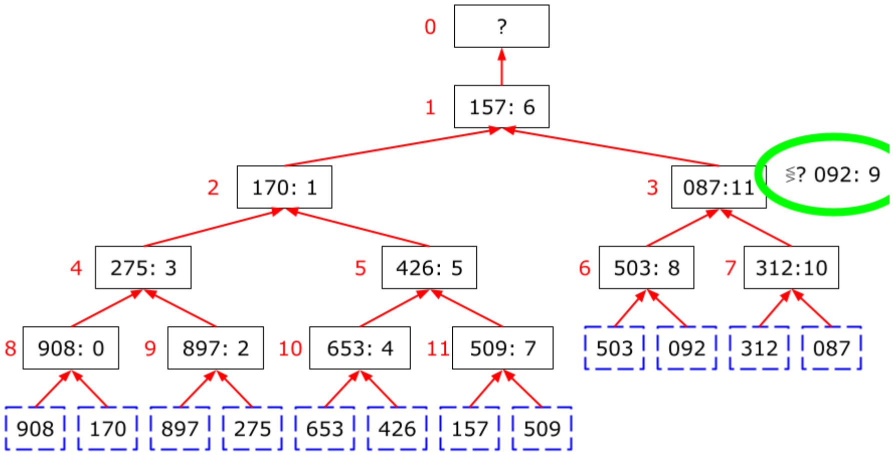

Figure 14. The comparison in the parent node.

Figure 14 illustrates the comparison in the parent node. Here, key value 92 loses to key value 87; therefore, key value 92 remains in this node and key value 87 moves up to compete at the grandparent node.

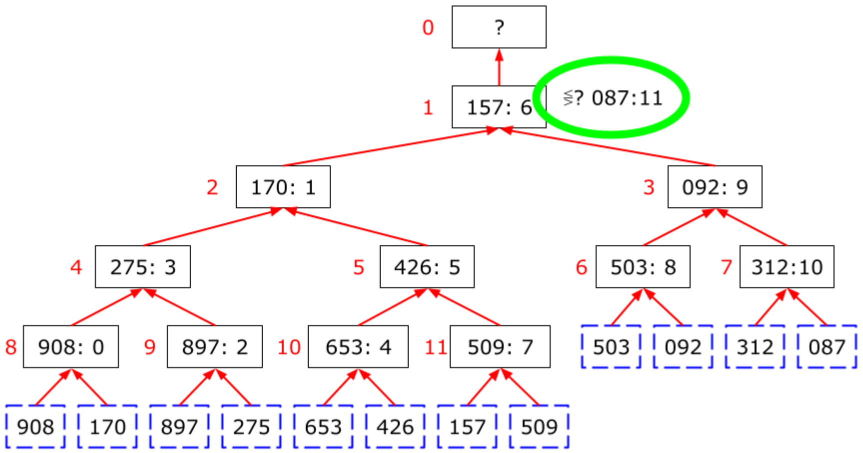

Figure 15. The comparison in the grandparent node.

Figure 15 shows the comparison in the grandparent node. Key value 87 wins against key value 157; therefore, key value 87 moves up to the root node.

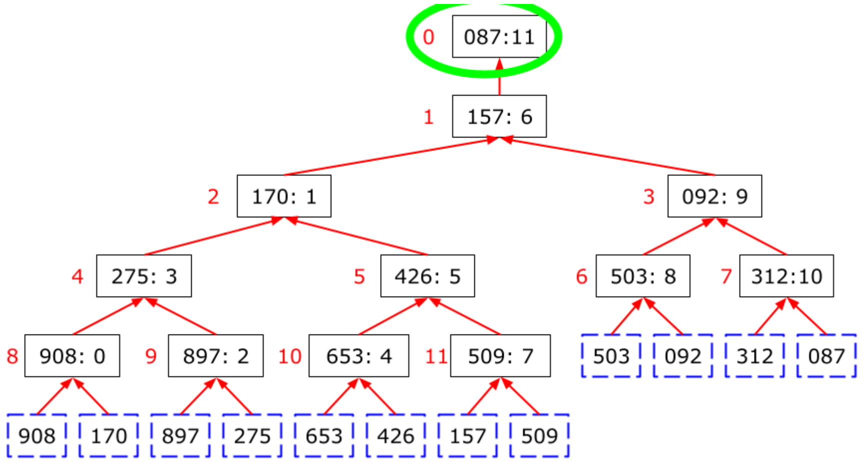

Figure 16. Final tournament tree.

Figure 16 shows the tournament tree after the leaf-to-root pass is complete. When a merge input ends, a plus infinity value must push valid values towards the root. The merge step is complete when plus infinity reaches the root.

9.2 Updates in an addressable priority queue

Section 3 and Figure 5 introduce the required logic as well as code to manage a tree-of-losers priority queue with large replacement key values, including logical deletion by a +∞ replacement key. The trees in the following figures illustrate examples of updates in an addressable tree-of-losers priority queue.

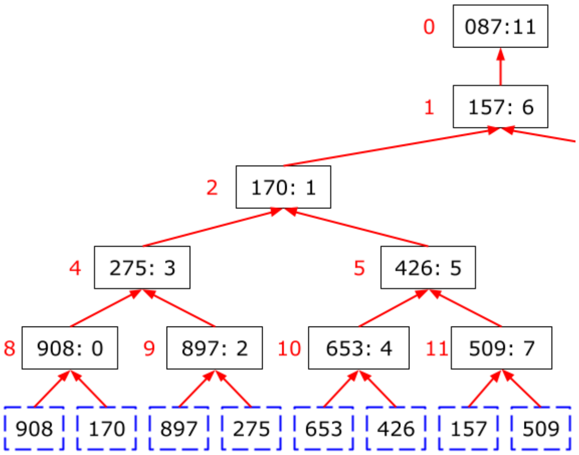

Figure 17. Initial tree-of-losers priority queue.

Figure 17 shows the starting point for this sequence of updates. The right half of the tree is cropped; its winner is input 11 with key value 087.

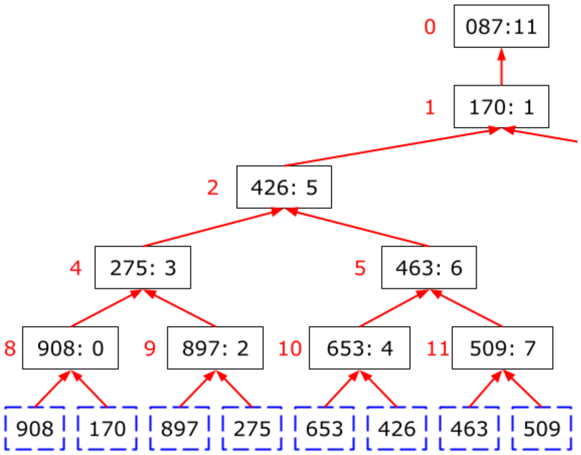

Figure 18. Tree-of-losers priority queue after increasing the key value for index 6.

Figure 18 shows the tree-of-losers priority queue after a leaf-to-root pass to replace key value 157 with 463 for index 6. This key value increase affects only the subtree within Figure 17 rooted by index 6 and the leaf-to-root pass ends there. New key value 463 wins and loses along the path such that key value 170 replaces the initial node for index 6.

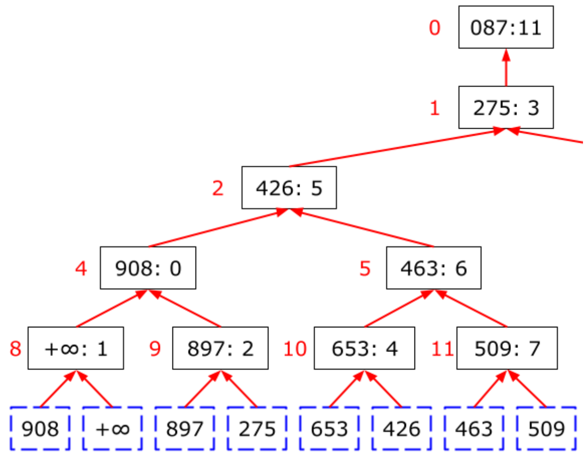

Figure 19. Tree-of-losers priority queue after logical deletion for index 1.

Figure 19 shows the same tree-of-losers priority queue after deletion of key value 170 for index 1. This replacement with late fence +∞ is rather similar to the prior update, with wins and losses, and with a termination of the leaf-to-root pass at index 1 in Figure 18.

9.3 *Non-monotone updates*

Section 4 introduces the required logic, and Figure 6 introduces new code, to manage a tree-of-losers priority queue with small replacement key values, including logical deletion by a −∞ replacement key. The trees in the following figures illustrate examples of a non-monotone sequence of new key values. These updates continue the sequence from Section 9.2.

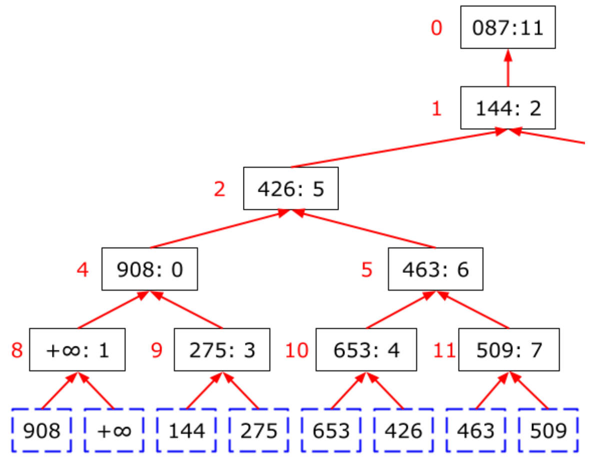

Figure 20. Tree-of-losers priority queue after decreasing the key value for index 2.

Figure 20 shows the same tree-of-losers priority queue after replacement of key value 897 with 144 for index 2, i.e., a lower replacement key value than the replaced key. Index 2 is found in a leaf node in Figure 19; the local prior winner (index 3) is found much further up. The new key value is lower (earlier in the sort order) than this prior winner, which therefore must move backward on its leaf-to-root path. The new key loses against the key value in the root; therefore, it ends up in the node of index 3 in Figure 19.

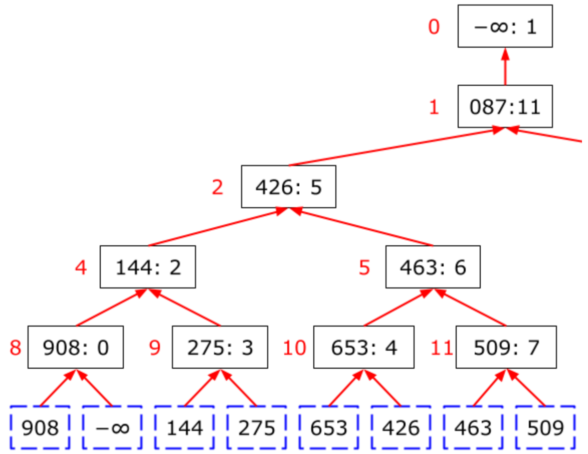

Figure 21. Tree-of-losers priority queue after replacing a late fence with an early fence.

Figure 21 shows the tree-of-losers priority queue after replacement of a late fence (key value +∞) with an early fence (key value −∞) for index 1. The prior winner in the leaf node (index 0) must move backward to the leaf, and in fact other prior winners must also move backward on their respective leaf-to-root paths, including the key value in the root of Figure 20. The early fence ends up in the root node.

9.4 A recent theorem with proof and corollaries

The following proposition and theorem relate the offset-value codes of three key values consecutive in a sorted stream, with examples from Figure 3. Two corollaries apply the theorem to comparisons decided by offset-value codes and extend the theorem to longer sorted streams, e.g., in database query processing.

Definitions: For key values A and B, let pre(A,B)≥0 be the length of their maximal shared prefix; let val(A,i) with i≥0 be the data at offset i within key value A; let val(A,B)=val(B, pre(A,B)) be the first difference in key value B relative to key value A; let A<B mean that A sorts lower (earlier) than B; and let ovc(A,B) with A≤B be the offset-value code of key value B relative to key value A, computed from pre(A,B) and val(A,B) as shown in Figure 3.

Proposition: For three key values A<B<C (with A≠B or B≠C or both), ovc(A,B) ≠ ovc(B,C).

Proof (by contradiction): ovc(A,B) = ovc(B,C) would imply that C has the same data value as B at the same offset, e.g., column index, but this would violate the definition of offset-value codes, which requires the maximal shared prefix.

Examples: In Figure 3, no two consecutive offset-value codes are equal.

Theorem: For three key values A<B<C, ovc(A,C) = max(ovc(A,B), ovc(B,C)) in ascending offset-value coding and ovc(A,C) = min(ovc(A,B), ovc(B,C)) in descending offset-value coding.

Proof (given here only for ascending offset-value codes in an ascending sort, in three cases by the lengths of maximal shared prefixes, with examples after the proof):

(i) If pre(A,B)>pre(B,C), then pre(A,C)=pre(B,C), val(A,C)=val(B,C), and ovc(A,C)=ovc(B,C). With ovc(A,B)<ovc(B,C), the theorem holds.

(ii) Otherwise, if pre(A,B)<pre(B,C), then pre(A,C)=pre(A,B), val(A,C)=val(A,B), and ovc(A,C)=ovc(A,B). With ovc(A,B)>ovc(B,C), the theorem holds.

(iii) Otherwise, pre(A,B)=pre(B,C) and, by the lengths of maximal shared prefixes, val(A,B)<val(B,C). Thus, pre(A,C)=pre(B,C), val(A,C)=val(B,C), and ovc(A,C)=ovc(B,C). With ovc(A,B)<ovc(B,C), the theorem holds.

Examples: Case (i) in the proof applies to the first three rows in Figure 3. If the second row were removed, then the offset-value codes of the third row would change in neither ascending nor descending offset-value coding. As an example of case (ii), if the second-to-last row were removed in Figure 3, the offset-value codes of the last row would be those of the removed row. As an example of case (iii), if the third row were removed in Figure 3, the offset-value codes of the fourth row would remain unchanged.

Corollary (Iyer's “unequal code theorem” [6] [I 05]): For three key values A<B<C, if ovc(A,B)<ovc(A,C), then ovc(B,C)=ovc(A,C).

Proof: The theorem implies ovc(A,C)=ovc(A,B) or ovc(A,C)=ovc(B,C), but the corollary’s precondition implies ovc(A,C)≠ovc(A,B).

Implication: If offset-value codes relative to base A decide the comparison between key values B and C, then the loser's offset-value code relative to the winner is the same as its offset-value code relative to the old base. There is no need to compute a new offset-value code for the loser relative to the winner.

Corollary: The theorem above extends to any number of intermediate key values, or if $X_0<X_1<\ldots<X_{n-1}<X_n$, then (in ascending offset-value coding) $\mathrm{ovc}(X_0,X_n) = \max_{i=1\ldots n}(\mathrm{ovc}(X_{i-1},X_i))$.

Proof (sketch): By induction using the theorem above.

Implication: If a filter or other operation removes rows from a sorted input, simple and efficient integer calculations can derive offset-value codes for the output from offset-value codes of the input.

[6] Iyer “equal code theorem” is about two key values B and C with equal offset-value codes relative to base A. It shows that B and C require data comparisons only past the shared offset and value.

9.5 Updates with offset-value coding

Section 6 introduces the logic and Figure 9 introduces the code required to manage a tree-of-losers priority queue with offset-value coding for strings of characters, symbols, bytes, fields, keys, or column values; with strings of column values better also known as database rows. The following figures illustrate a sequence of replacement key values with offset-value codes in the tournament tree but without the benefit of offset-value codes in replacement key values (i.e., an offset-value code relative to the replaced key value).

In these diagrams, the dashed boxes show key values for indexes 8-11, the solid boxes show tree nodes. The left part of the tree with indexes 0-7 has been cropped; only the winner of that sub-tree is shown: key value “157” from index 6. Each node contains a string value, a character value, its offset, and an index, e.g., a run number in a merge step or an array index during run generation. Ascending offset-value codes are shown above many nodes, derived from the offset and the character value. For example, ‘8’@1 means character ‘8' at offset 1 distinguishes this key value from its base key.

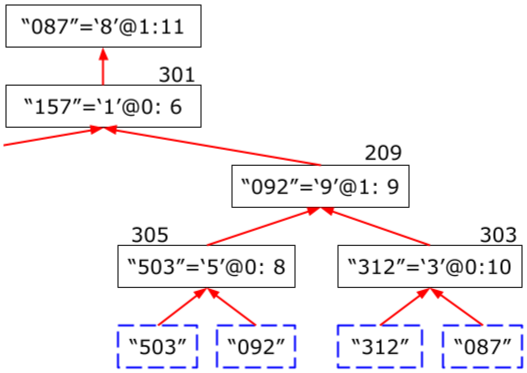

Figure 22. Initial tree-of-losers priority queue with offset-value codes.

Figure 22 shows the starting point of the example sequence of updates.

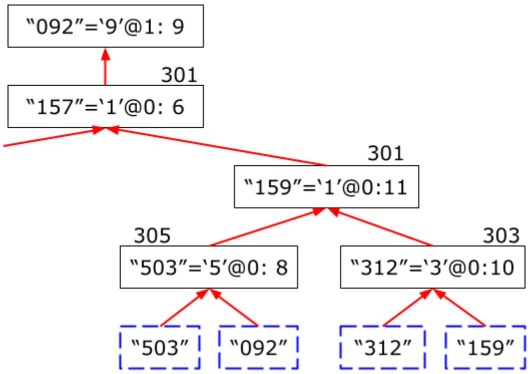

Figure 23. Priority queue with new key value “159” for index 11.

Figure 23 shows the tree-of-losers priority queue after a new key value “159” replaces the previous key value associated with index 11. The search loop, starting with the state shown in Figure 22 and invoking two full string comparisons without the benefit of offset-value codes, has key value “159” win against “312” but lose against “092". Thereafter, key value “092” goes to the root with one comparison of two offset-value codes relative to former root key value “087”.

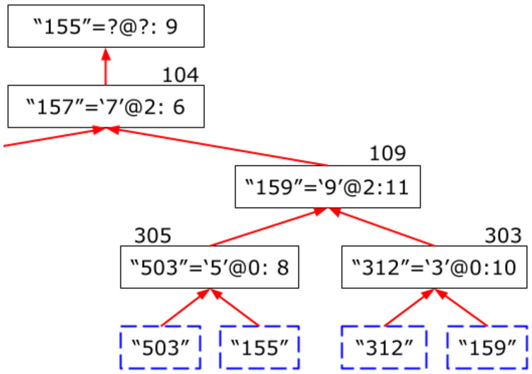

Figure 24. Priority queue with new key value “155” for index 9.

Figure 24 shows the tree-of-losers priority queue after a new key value “155” replaces the key value previously associated with index 9. The search loop (starting with the state shown in Figure 23) requires three full string comparisons in a full leaf-to-root pass for key value “155”. Two of these comparisons produce new offset-value codes (new offset 2) for pre-existing key values “157” and “159”.

After this update, there is no offset and no offset-value code in the root node. It may be assumed that the lack of offset-value codes in the input implies that the current application context is something different from merging sorted runs; if so, there is no need for a sorted output run with offset-value codes.

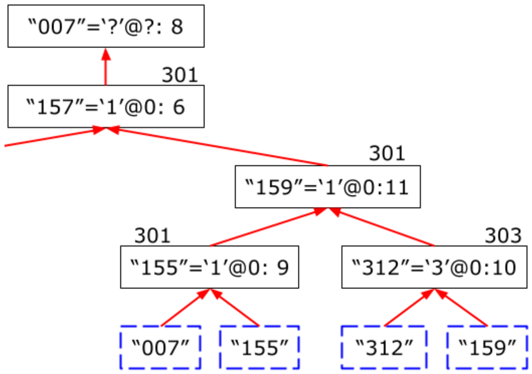

Figure 25. Priority queue with new key value “007” for index 8.

Figure 25 shows the tree-of-losers priority queue after key value “007” is assigned to index 8. The search loop finds the index value in the leaf node (see Figure 24) and the repair loop finds the leaf node’s local winner in the root node. The only comparison in this update is a full comparison at the root. The prior winner and its key value “155” lose against the new key value; therefore, the prior winner must move backward on its leaf-to-root path from the root to the leaf node while the new replacement key value “007” takes its place in the root node. Now two of the tree nodes on this leaf-to-root path have incorrect offset-value codes: the offset-value codes relative to prior local winner “155” must be replaced with new offset-value codes relative to the new local winner “007”. This is the task of the new loop introduced in Section 6 (lines 76 and 77 in Figure 9).

9.6 *Code samples for tree-of-losers priority queues*

Code samples and discussions in this document refer to methods for which some readers may appreciate additional sample code.

```
typedef unsigned int Index;
typedef unsigned int Key;
typedef char Level;

Level const height;
struct Node
{
  Index index;
  Key key;
  ...
} * const heap;

PQ::~PQ () { delete [] heap; }
PQ::PQ (Level const h)
  : height (h), heap (new Node [1 << h])
{
  ...
}

Index PQ::capacity () const { return Index (1 << height); }
Index PQ::root () const { return Index (0); }

void PQ::leaf (Index const index, Index & slot) const
    { slot = capacity() + index; }
void PQ::parent (Index & slot) const { slot /= 2; }

void PQ::leaf (Index const index, Index & slot, Level & level) const
    { level = 0; leaf (index, slot); }
void PQ::parent (Index & slot, Level & level) const
    { ++ level; parent (slot); }

Key PQ::early_fence (Index const index) const { return Key (index); }
Key PQ::late_fence (Index const index) const { return ~ Key (index);}

bool PQ::empty ()
{
  Node const & hr = heap [root ()];
  while (hr.key == early_fence (hr.index))
    pass (hr.index, late_fence (hr.index));
  return hr.key == late_fence (hr.index);
}

Index PQ::poptop (bool const invalidate)
{
  if (empty ()) return badIndex;
  if (invalidate)
    heap [root ()].key = early_fence (heap [root ()].index);
  return heap [root ()].index;
}
```

```
131.  Index PQ::top () { return poptop (false); }
132.  Index PQ::pop () { return poptop (true); }
133.  void PQ::push (Index const index, Key const key)
134.      { pass (index, early_fence (capacity ()) + key); }
135.
136.  void PQ::insert (Index const index, Key const key)
137.      { push (index, key); }
138.  void PQ::update (Index const index, Key const key)
139.      { push (index, key); }
140.  void PQ::delete (Index const index)
141.      { pass (index, late_fence (index)); }
```

Figure 26. Types and methods for tree-of-losers priority queues.

Figure 26 lists some type definitions and some methods for tree-of-losers priority queues. Highlighting in Figure 26 shows how much the principal methods for priority queues rely on direct and indirect invocations of the "pass" method discussed throughout this paper (lines 119, 134, and 141 as well as 125, 131, 132, and 139). The entire state of the priority queue is captured in two constants, height and heap (lines 85 and 91), initialized by the constructor (line 95). Line 97 stands for initialization of the tree-of-losers array; line 90 stands for definitions of methods "Node", "swap", and "sibling", not additional data. Note that the "push" method (line 133) modifies the key to skip over early fence keys (line 134) – in order to allow for early and late fences, permissible values for the "key" values to "push" is the full range of type "Key" (line 82) minus twice the capacity of the priority queue. Note also that the "empty" method (line 115) is not declared "const" as it might shift some heap entries from the root towards the leaf level by replacing early fences with late fences and invoking appropriate leaf-to-root passes (see lines 119, 125, 131, and 132). The "delete" method (line 140) does not include an optimization mentioned towards the end of Section 6, which would require code similar to line 127.

The most noteworthy observation in Figure 26 is that all methods are implemented using leaf-to-root passes – none requires a root-to-leaf pass. Compared to a leaf-to-root pass, a root-to-leaf pass would forgo efficient comparisons using offset-value codes and it would require twice as many comparisons per tree level and per pass.

9.7 Code samples for offset-value coding

The follow code sample illustrates comparisons with offset-value codes.

```
142. bool PQ::Node::less (Node & other, bool const full)
143. {
144.   Offset offset;
145.   if (full) offset = Offset (−1);
146.   else if (key != other.key) return (key < other.key);
147.   else offset = offsetFromKey (key);
148.
149.   bool const isLess = less (index, other.index, offset);
150.   Node & loser = (isLess ? other : * this);
151.   loser.key = keyFromOffset (loser.index, offset);
152.   return isLess;
153. }
154.
155. bool less (Index const left, Index const right, Offset & offset)
156. {
157.   while ( ++ offset < size)
158.     if (data [left] [offset] != data [right] [offset])
159.       return data [left] [offset] < data [right] [offset];
160.   return false;
161. }
```

Figure 27. Comparisons with offset-value coding.

Figure 27 illustrates comparisons with offset-value codes as well as full comparisons without them. The “Offset” type in lines 144 and 155 is some appropriate integer type. Unless line 145 forces full data comparisons (see Section 6 and Figure 9), line 146 decides a comparison entirely by offset-value codes, ideally the common case. Line 147 exploits earlier data comparisons captured in offsets. Line 149 invokes additional data comparison; the “offset” parameter is both input and output in the function starting in line 155. Lines 157 and 158 assume appropriate global constants or variables for key “size” and “data” arrays.

Line 150 identifies the loser and line 151 maps the deciding offset to a new offset-value code for the loser. This is the method’s side effect in addition to the result in line 152.

Line 147 and the side effect of line 157 show the crucial benefit of offset-value coding. They ensure a linear total cost for data comparisons in a merge sort, i.e., <N×K data comparisons for N rows with K keys. This bound applies to both internal and external merge sort.