Heapsort for Equal Keys

Kai Schweinsberg[2], Jukka Teuhola[1], and Lutz Wegner[2]

[1] Department of Information Technology, University of Turku,
Joukahaisenkatu 3-5, FI-20014 Turku, Finland
teuhola@it.utu.fi

[2] Department of Electrical Engineering and Computer Science, University of Kassel
D-34121 Kassel, Germany
kai.schweinsberg@uni-kassel.de, lutzmwegner@gmail.com

Preprint September 2014

Abstract. Efficient duplicate detection and deletion is an algorithmic challenge both in practical terms and from a theoretical stand point. Duplicates may occur in database tables after a projection, in tracking web traffic, experimentation and statistics. To reduce these multisets to proper sets, the most common approach is to sort the file first and then – in an additional sweep – take one instance, say the first, from each multiplicity of keys. If done in place, ideally the front of the file contains afterwards the sorted subset of unique keys and the duplicates are in the back. Sorting methods which can be engineered to do early duplicate deletion may reduce the effort spent to $O(n \log k)$, where k is the number of distinct keys. General wisdom had it that this smooth behaviour wasn't achievable with heapsort unless the sort was totally redesigned in the style of Dijkstra's SMOOTHSORT. Here we show that this is a misperception and present DDHEAPSORT as a duplicate elimination method which achieves the lower bound of $O(n \log k)$ steps – both on the average and in the worst case – and requires $O(1)$ extra space. Empirical evidence suggests that DDHEAPSORT comes with very little penalty in the case of no duplicates when compared to a fast *heapsort* with subsequent duplicate detection sweep.

Keywords: Sorting, heapsort, multisets, duplicate deletion

1 Introduction

In May 1987, the last named author of this contribution was invited to hold a one week long series of lectures on *Sorting* at the University of Turku. The idea for this visit had come from the second named author. Emphasis was placed on novel methods dealing with multisets and presortedness. A revised edition of the presented algorithms was recently published as Lecture Note of the Turku Centre for Computer Science [13].

One open issue during the lectures concerned a possible modification of the classical *heapsort* [4, 5, 14] with the aim of elegantly eliminating duplicates, i.e. separating a given multiset s.t. the subset of records with pairwise distinct keys is afterwards in front and the duplicates at the back, all with only $O(1)$ extra space

and in at most $O(n \log n)$ time. Normally, the common approach to this task is to sort the file first and then – in an additional sweep – take one instance, say the first, from each multiplicity of keys. Alternatively, suitable forms of hashing can identify and separate the subset of pairwise distinct keys and their duplicates, as suggested by Teuhola and Wegner [11].

When the problem was presented in 1987, the participants agreed that it could be solved by intelligently dividing the file into three parts: a sorted prefix of records with unique keys, a sequence of duplicates, and a heap, respectively sequence of yet untreated records[3]. Extra key comparisons on equality were needed to identify a duplicate. Clearly they would occur when parent-child and sibling orderings take place in the *sift*-operations in *heapsort*.

When in building the heap and in the selection phase a duplicate is detected, it could be replaced by the rightmost leaf (or leftmost leaf in a min-heap with root to the right). Note that this node with new value might continue up or down the heap or be again a duplicate to parent, child or sibling. This led to the notion of a *locally duplicate-free heap*, or *df-heap* for short.

2 Locally duplicate-free heaps

As an example for a *df-heap*, consider a worst case heap[4] s.t. no two siblings are equal, respectively no parent is equal to one of its children. Figure 1 is a min-heap. The maximal values of n for $k = 1, 2, \ldots, 8$ in this example are 1, 2, 4, 6, 10, 14, 22, 30.

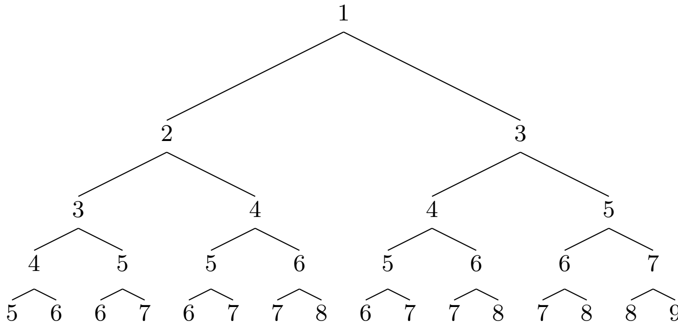


Fig. 1. A df-heap: no parent-child or sibling-nodes have equal keys.

Observation: A df-heap with k distinct values has at most $n = f(k) = 2^{k/2+1} - 2$ nodes, for k even, and $n = (f(k-1) + f(k+1))/2$ nodes for k odd.

[3] We often equate keys with records for shortness, although our sorting model is based on records having a key component and satellite information.

[4] Worst case in the sense that we want the largest df-heap with N nodes that can be build with k distinct keys.

At that point the hand-waving started in 1987 and the idea was never followed-up with a concise algorithm. Instead Teuhola and Wegner later presented a minimal space, average linear time duplicate deletion algorithm, termed DDT, which is based on hashing and was already mentioned above [11].

3 Where to search for duplicates?

The idea to design a duplicate deletion algorithm based on the principle of a df-heap is tempting. We remove the root (minimum) from a df-heap, place the leftmost leaf in its position, do a *sift*-operation which again brings the new minimum to the root and restores as postcondition the df-heap property.

However, as soon as one starts to implement this idea, doubts arise as to the usefulness of locally testing for a duplicate that could be anywhere in the heap. In particular, to establish the df-heap property in phase 1 (heap building bottom-up), the leaf layer of a heap has to be included. Normally, building starts at position $N/2$ working its way up to the root. To make sure that no two siblings are equal, we must compare nodes at position i and $i + 1$ which adds $N/4$ comparisons with little benefit.

Secondly, we would like to use the Floyd-improvement of letting a hole sink down first which saves parent-child comparisons and thus halfs the number of key comparisons. Overall it is known to improve the asymptotic speed from $16N \log N + 0.2N$ to $13N \log N + O(N)$. Now, if we detect a duplicate in that phase, what should we do? Turn it into yet another hole? This quickly creates a bookkeeping problem. Thus we abandoned the idea of maintaining a df-heap early on.

Rather we used the Floyd-improvement, but checked for duplicates in the *sift-up* phase only. If a duplicate is detected in this *sift-up* phase, it is turned into a hole and sinks down again, so there is no bookkeeping problem. It is known that keys which came from a leaf-position don't climb very high. The probabilities of an inserted leaf to climb up i levels are [6, p. 619, answer to ex. 18]:

$i = 0 : 0.848$
$i = 1 : 0.135$
$i = 2 : 0.016$

The rationale was that we invest little added overhead in searching for duplicates. Should, however, the input contain a very large number of duplicates, they have a chance to be detected and deleted in the first heap building phase which is known to be linear in time.

Unfortunately, a first implementation showed no real speed-up for duplicates, even though duplicates were detected in the *sift-up* phase and removed immediately. The reason for the poor speed-up is that the path-length for the *sift*-operations remains almost unchanged, even though a larger number of leaves are pruned from the tree.

Overall, searching for duplicates as part of the child-child and parent-child comparisons in the *sift*-phase isn't as good an idea as the concept of a df-heap

suggests. Rather, the root region of the heap is the only relevant area for duplicate handling. To understand this, consider the following Figure 2 for a min-heap with fixed multiplicities m and keys 1, 2, …, k.

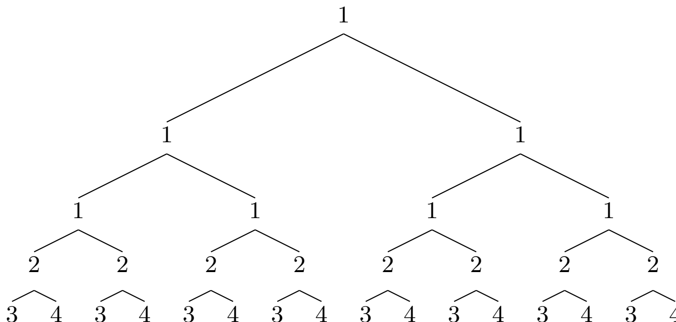


Fig. 2. Perfect min-heap for a multiset.

The example uses $k = 4$ and $m \leq 8$. Clearly, the neat level-wise ordering will rarely occur for input in random order but illustrates the point here. The point is that the m (or rather $m - 1$) occurrences of the minimum (here key value 1) form a tree of height $\log_2 m$, whereas the next m occurrences of the second minumum form a single layer of width m, the third and fourth minima occupy half a layer etc.

To gain a speed-up for multiset input it is thus necessary and sufficient to skip the unary root area of the heap thus shortening each *sift*-path by $O(log_2\ m)$ comparisons. Indeed this is what DDHeapsort does.[5]

4 Solution

The solution has two parts which must be explained. First, how to shorten the *sift*-path. Second, how to keep the sorted sequence of unique keys, the duplicates, and the remaining heap separate without additional space.

To shorten the *sift*-path, we do not extract the root from the heap, replacing it with the leftmost leaf, as done in heapsort. Rather we scan levelwise for duplicates of the root starting at the root. The first key not equal to the root stops the scan and the key before it, say at position r, which still is a duplicate, is removed from the heap. The leaf enters here at r and is sifted down (as a hole as in Floyd's improvement). This does not violate the heap property because everything above and including the node at r is a duplicate to the root node.

[5] Deapsort would have been a nicer name, but unfortunately there is a name clash with the data structure "deap" which is a special min-max-heap proposed by Svente Carlsson [1,2]. We stick to the name DDHeapsort for the moment.

Should the *sift*-routine bring another duplicate to the node at r, we repeat the process, otherwise r retreats stepwise to the root, deleting duplicates in each step. Note that we do not claim that this removes *all* duplicates in one scan, as Figure 2 might suggest. In fact, one non-duplicate might appear right under the root, stopping the scan, as shown in Figure 3.

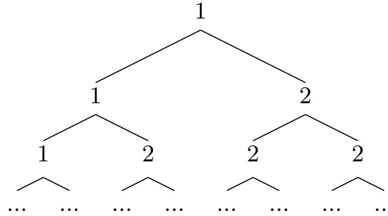


Fig. 3. Min-heap with skewed distribution for a multiset.

The second part, keeping track of duplicates and already sorted prefix, is easily explained through the following Figure 4. To have the duplicates afterwards at

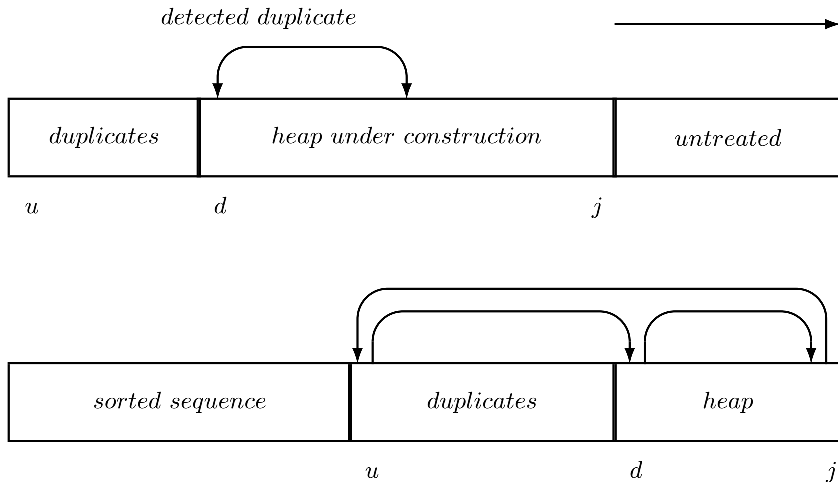


Fig. 4. Heap build-up and extraction of the minima

the right end of the file, a min-heap is constructed whose root is to the right. Everything is thus mirrored and the heap levels run from the right to the left. The formula for converting an index i in *heapsort* into j in DDHeapsort is

$j = n - i + 1$. The right child of a node j in DDHeapsort is in location $2*j-n-1$, the left child in $2*j-n-2$. Otherwise the algorithm follows closely the ordinary heapsort principle[6], i.e. in phase 1 a heap is constructed bottom up and phase 2 extracts continuously the minimum value (but not from the root position) which is thrown to the left. There it is compared to the previously extracted key in order to recognized it as a possible duplicate. Duplicates roll to the right in what we termed a *wheel* in [11].

Scanning keys in the root area for equality to the root node before extraction and checking for duplicates after removal from the heap is done for each key once and adds $2n$ key comparisons. Combining both into one check would be nice but seems complicated. Even without optimizations, DDHeapsort has some nasty special cases, e.g. when a duplicate is at the same time the leftmost leaf that should be swapped with a key from the root-area.

Listing 1.1. DDHeapsort – *heapsort* with on-the-fly duplicate deletion.

```
{DDHeapsort is a variant of heapsort which does}
{duplicate deletion on-the-fly. Its input is an}
{array a from 1 to n. It has the minimum at its}
{root and the root is to the right. Afterwards,}
{the sorted sequence of unique keys is to the  }
{left in a[1..u] and duplicates on the right in}
{a[u+1 .. n]. DDHeapsort returns the number of }
{pairwise distinct keys.}

function ddheapsort(var a: sequence; n: longint): longint;
var r, d, u: longint;
  t: item;

  procedure dsift(var a: sequence; i, k: longint);
  {for a newly placed key at a[k], dsift restores  }
  {the heap property. It works with the Floyd-      }
  {improvement for heaps (SiftUp), i.e. starts      }
  {with sinking a hole. No searching for duplicates}
  {inside dsift. The heap ends at a[i], i.e. k is  }
  {the right end (root), i the leftmost leaf, which}
  {is one past the last duplicate. The original     }
  {array stretched from 1 to n.}

  var start, j: longint;
    x: item;
  begin
    start := k;
    x := a[k];
```

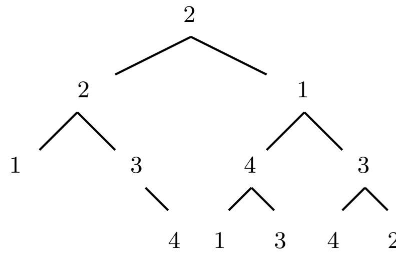
[6] The actual code was derived from a C-implementation of Floyd's improvement which the second author had published on the Internet [10].

```
    j := 2*k - n - 1; {j on right child}
    while j >= i do
    begin
      if j > i then
      begin
        if a[j].key > a[j-1].key then dec(j);
      end; {a[j] is smaller child}
      a[k] := a[j]; k := j; j := 2*k - n -1;
    end; {hole at a[k] has reached bottom}
    j := (k + n + 2) div 2; {j is father of k}
    while j <= start do
    begin
      if a[j].key > x.key then
      begin {father moves down, hole moves up}
        a[k] := a[j];
        k := j; j := (k + n + 2) div 2
      end
      else Break
    end;
    a[k] := x
  end {of dsift};

begin {ddheapsort}
  for r := n - (n div 2) + 1 to n do
    dsift(a, 1, r);
  {first key is never a duplicate; }
  {sort then by extracting continuously the}
  {minimum, but check for duplicates first.}
  t := a[n]; a[n] := a[1]; a[1] := t;
  u := 2; {u is one past the end of sorted prefix}
  d := 2; {d is one past the end of duplicates}
  while d <= n do
  begin
    dsift(a, d, n);
    r := n - 1;
    while (r >= d) and (a[r].key = a[n].key) do
      dec(r);
    {a[r] is rightmost non-duplicate to root}
    inc(r);
    while (r <= n) and (r >= d) do
    begin
      t := a[r]; a[r] := a[d]; a[d] := a[u]; a[u] := t;
      inc(d);
      dsift(a, d, r);
      if a[u].key <> a[u-1].key then inc(u);
```

```
      while (a[r].key = a[n].key) and (r >= d) do
        dec(r);
      inc(r)
    end
  end;
  ddheapsort := u - 1;
end {ddheapsort};
```

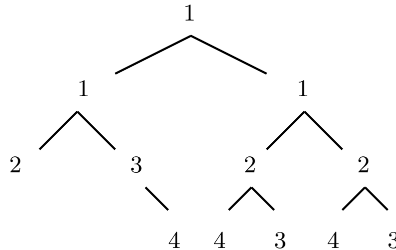
As an example consider an input of $n = 12$ records with $k = 4$ distinct values, each occuring $m = 3$ times (Figure 5). In the following Figure 6 we see the



4	1	3	4	2	1	3	4	3	2	1	2

Fig. 5. Start situation – no min-heap yet.

situation after phase 1. The min-heap is almost optimal with only one inversion on the second lowest level. Phase 1 does not search for duplicates.



4	4	3	4	3	2	3	2	2	1	1	1

Fig. 6. Situation at end of phase 1.

Next, we show the situation when all 1s and 2s have been removed from the heap (Figure 7) and four records have been eliminated as duplicates. Note that the boldface 4 denotes the leftmost leaf. The remaining nodes above it happen to form an optimal multiset heap.

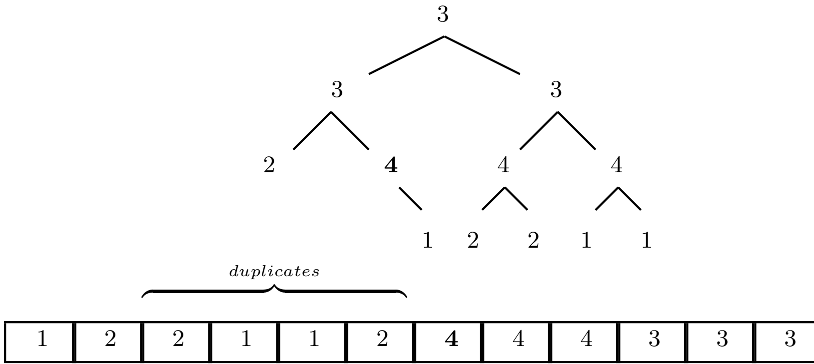


Fig. 7. Situation after removal of all 1s and 2s.

Finally, when sorting is completed, the duplicates are all in the back as shown in Figure 8. We should also emphasize that the algorithm is not *stable* in the sense defined by Knuth [7, p. 4], i.e. equal keys do not retain their relative order and thus the one we extract and attach to the list of unique keys is not guaranteed to be the first in input order. As can be seen from Figures 5 to 8, duplicates are shuffled heavily.

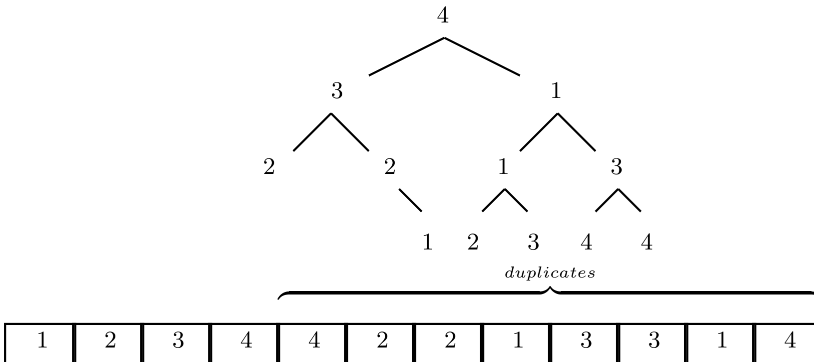


Fig. 8. Final outcome – unique keys are in front.

As another remark, the number of key comparisons observed for this particular input permutation is 69. Other permutations for this particular multiset produced 68 or 70 comarisons, so there seems little deviation from the average. This leads directly to the question of effectiveness in general.

5 Performance

When Dijkstra introduced SMOOTHSORT as a *heapsort*-variant for presorted input [3], he claimed a smooth transition from $O(n \log n)$ for no presortedness to $O(n)$ for input already in ascending order. Indeed we would like to have the same smooth transition for multiset input. In the case of *quicksort*, Sedgewick has proven a lower bound for partition-exchange programs of the *quicksort*-type when sorting *n-ary* input [9]. With `MIX`-comparisons the number of comparisons for fixed multiplicities M is $2(N+M)H_n - 3N - n$ which is $O(N \log n)$, $n = N/M$ the number of distinct keys. In [12] Wegner shows that several of his quicksort derivatives both for linked lists and arrays achieve this lower bound.

As it turns out, DDHEAPSORT shows this $O(n \log k)$ performance on the average. This comes from saving $\log_2\ m$ path-length on the *sift*-operations for each of the n records by skipping duplicates at the root. Given each key occurs m times, we have $O(n \log\ n - n \log\ m) = O(n \log\ k)$ key comparisons for $k = n/m$.

Figures 9 and 10 show the number of comparisons and running times for DDHEAPSORT and a *heapsort* with Floyd-improvement and an subsequent sweep to eliminate duplicates *in-situ*. Note that the x-axis for the multiplicities m has a logarithmic scale.

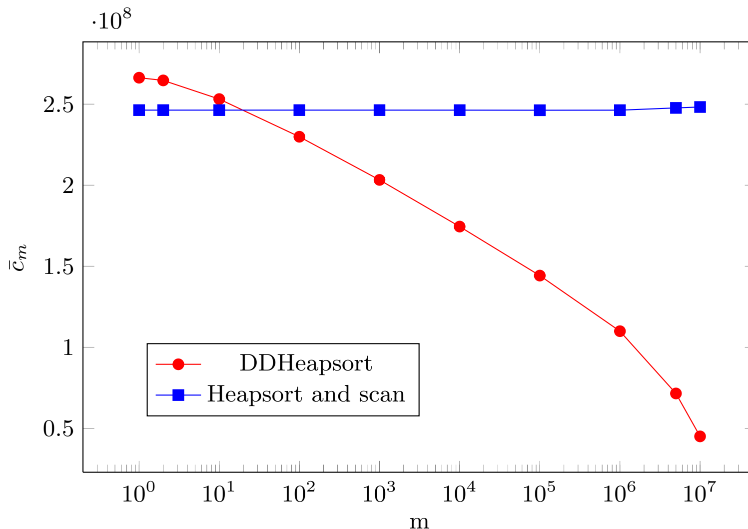


Fig. 9. Key comparissons on the average for $n = 10,000,000$

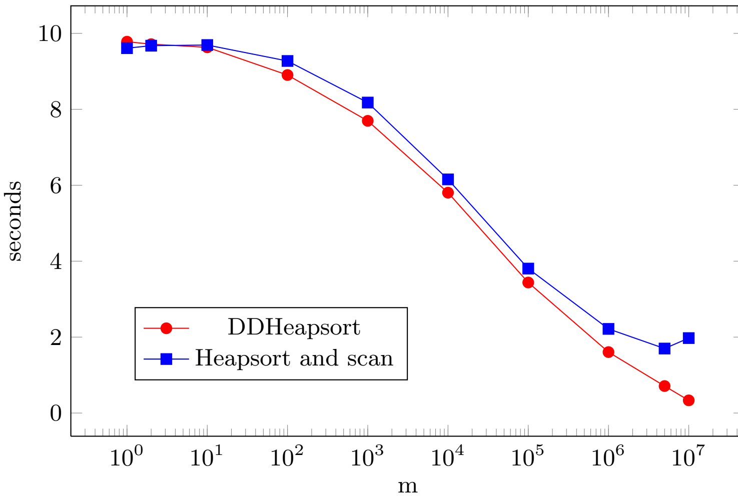


Fig. 10. Execution times on the average for $n = 10,000,000$

In summary, we have shown how to do smooth, on-the-fly duplicate deletion with a heapsort-variant and have proven that this comes with little penalty when competing with one of the better heapsorts with duplicate elimination afterwards. As compared to methods based on hashing, the records with unique keys are in ascending order afterwards. Also the first (and minimal) unique key is delivered in linear time as the building phase continues to be of order $O(n)$ as with heaps in general.

On the negative side, timings show no speed-up that would match the decrease in key comparisons for DDHeapsort. The reason seems to be that with multisets the ordinary *heapsort* can profit from caching effects as leaves sink along the same path all the time. Things would change completely when we started to prune the tree from the root as Smoothsort does, organizing its data as a forest of Leonardo-trees. However, Smoothsort is not competitive at all for ordinary input in random order.

6 What if ternary comparisons came for free?

As a last remark we would like to comment on the difference between binary and ternary key comparisons. When discussing solutions for sorting or duplicate deletion in connection with multisets, the difference is an issue. Consider the following code fragment (Listing 1.2).

Listing 1.2. Nested if-then-else.

```
if a[i].key < a[j].key
then begin ... end
else
```

if $a[i].key = a[j].key$
then begin ... **end**
else $\{a[i].key > a[j].key\}$ **begin** ... **end**;

A smart optimizing compiler might turn this three-way comparison into

```
CMP Op1, Op2
JLT Addr1
JEQ Addr2
#code for the case > a[j].key
...
```

As to our understanding of processor architectures, a `CMP Op1, Op2`-instruction (*compare Operand1, Operand2*) delivers a result to a flag register, from which the three orderings $<, =, >$ may be deduced. The comparison is followed by jump-instructions like a `JLE` or `BLE`, i.e. a *jump on less or equal, branch on less or equal.* They come with a jump target as operand from which execution continues when the condition is fulfilled, otherwise execution continues with the next instruction in sequence.

If a compiler recognizes the nature of the nested if-then-else construct above and generates the sequence of three maschine instructions shown, then this is hardly more expensive than a binary comparison and most likely faster than branching via jump tables. This would also explain why there aren't any conditional branches with *two* jump targets, say something like `J3LE R1, R2` with indirect jumps via two address registers R_1, R_2. If it existed, a given comparison result -1 made us jump to $[R_1]$, a 0 directed us to $[R_2]$ and with a +1 the instruction counter was incremented.

The way things are, programmers would be on the safe side if high-level programming languages would provide constructs for three-way comparisons. This is only the case with a few exotic languages. The C-language offers a built-in `strcmp`-function (string comparison) which returns a three-valued result. The result must then be further processed with, say, a *case*-instruction which is not what we have in mind.

If three-way comparisons came for free and algorithm designers were aware of the advantages, quite a few algorithms might look differently. One example would be three-way partitioning in quicksort (the Dutch national flag problem [8]). Another is DDHeapsort here, where we would test for equality all along the path of a *sift*-operation, deleting duplicates near the root of a heap much earlier.

References

1. Carlsson, S.: The deap - a double-ended heap to implement double-ended priority queues. Inf. Process. Lett. 26(1), 33–36 (1987)
2. Carlsson, S., Chen, J., Strothotte, T.: A note on the construction of data structure "deap". Inf. Process. Lett. 31(6), 315–317 (1989)
3. Dijkstra, E.W.: Smoothsort, an alternative for sorting in situ. Sci. Comput. Program. 1(3), 223–233 (1982)

4. Floyd, R.W.: Algorithm 113: Treesort. Commun. ACM 5(8), 434 (1962)
5. Floyd, R.W.: Algorithm 245: Treesort3. Commun. ACM 7(12), 701 (1964)
6. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)
7. Knuth, D.E.: The Art of Computer Programming. Vol. 3: Sorting and Searching, 2nd ed., vol. 3. Addison-Wesley (1998)
8. McMaster, C.L.: An analysis of algorithms for the dutch national flag problem. Commun. ACM 21(10), 842–846 (Oct 1978)
9. Sedgewick, R.: Quicksort with equal keys. SIAM J Comput 6, No. 2, 240–267 (1977)
10. Teuhola, J.: Bottom-up heapsort (1993), `www.diku.dk/hjemmesider/ansatte/jyrki/Experimentarium/in-place_linear_probing_sort/heapsort.bottom-up.cTeuholaheapsortfloyd`, [Online; accessed 17-June-2014]
11. Teuhola, J., Wegner, L.M.: Minimal space, average linear time duplicate deletion. Commun. ACM 34(3), 62–73 (1991)
12. Wegner, L.: Quicksort for equal keys. IEEE TC C-34 No. 4, 362–367 (April 1985)
13. Wegner, L.M.: Sorting - The Turku Lectures (2014)
14. Williams, J.: Algorithm 232 (Heapsort). CACM 7, 347–348 (1964)