Technical Reports

L2/10-188

Proposed Update Unicode Technical Standard #10

UNICODE COLLATION ALGORITHM

Version	6.0.0 draft 3
Authors	Mark Davis (markdavis@google.com), Ken Whistler (ken@unicode.org)
Date	2010-04-28
This Version	http://www.unicode.org/reports/tr10/tr10-21.html
Previous Version	http://www.unicode.org/reports/tr10/tr10-20.html
Latest Version	http://www.unicode.org/reports/tr10/
Latest Proposed Update	http://www.unicode.org/reports/tr10/proposed.html
Revision	21

Summary

This report ~~provides~~ is the specification of the Unicode Collation Algorithm (UCA), which ~~provides a specification for~~ details how to compare two Unicode strings while remaining conformant to the requirements of the Unicode Standard. The UCA also supplies the Default Unicode Collation Element Table (DUCET) as the data specifying the default collation order for all Unicode characters.

Status

*This is a **draft** document which may be updated, replaced, or superseded by other documents at any time. Publication does not imply endorsement by the Unicode Consortium. This is not a stable document; it is inappropriate to cite this document as other than a work in progress.*

> ***A Unicode Technical Standard (UTS)** is an independent specification. Conformance to the Unicode Standard does not imply conformance to any UTS.*

Please submit corrigenda and other comments with the online reporting form [Feedback]. Related information that is useful in understanding this document is found in the References. For the latest version of the Unicode Standard see [Unicode]. For a list of current Unicode Technical Reports see [Reports]. For more information about versions of the Unicode Standard, see [Versions].

Contents

1 Introduction

Collation is the general term for the process and function of determining the sorting order of strings of characters. It is a key function in computer systems; whenever a list of strings is presented to users, they are likely to want it in a sorted order so that they can easily and reliably find individual strings. Thus it is widely used in user interfaces. It is also crucial for databases, both in sorting records and in selecting sets of records with fields within given bounds.

Collation varies according to language and culture: Germans, French and Swedes sort the same characters differently. It may also vary by specific application: even within the same language, dictionaries may sort differently than phonebooks or book indices. For non-alphabetic scripts such as East Asian ideographs, collation can be either phonetic or based on the appearance of the character. Collation can also be customized according to user preference, such as ignoring punctuation or not, putting uppercase before lowercase (or vice versa), and so on. Linguistically correct *searching* needs to use the same mechanisms: just as "v" and "w" sort as if they were the same base letter in Swedish, a loose search should pick up words with either one of them.

Collation implementations must deal with the complex linguistic conventions for ordering text in specific languages, and provide for common customizations based on user preferences. Furthermore, algorithms that allow for good performance are crucial for any collation mechanisms to be accepted in the marketplace.

The following table shows some examples of cases where sort order differs by language, usage, or another customization.

Example Differences

Language	Swedish:	z < ö
	German:	ö < z
Usage	German Dictionary:	of < öf
	German Telephone:	öf < of
Customizations	Upper-First	A < a
	Lower-First	a < A

Languages vary regarding which types of comparisons to use (and in which order they are to be applied), and in what constitutes a fundamental element for sorting. For example, Swedish treats *ä* as an individual letter, sorting it after *z* in the alphabet; German, however, sorts it either like *ae* or like other accented forms of *a*, thus following *a*. In Slovak, the digraph *ch* sorts as if it were a separate letter after *h*. Examples from other languages and scripts abound. Languages whose writing systems use uppercase and lowercase typically ignore the differences in case, unless there are no other differences in the text.

It is important to ensure that collation meets user expectations as fully as possible. For example, in the majority of Latin languages, ø sorts as an accented variant of o, meaning that most users would expect ø alongside o. However, a few languages, such as Norwegian and Danish, sort ø as a unique element after z. Sorting "Søren" after "Sylt" in a long list, as would be expected in Norwegian or Danish, will cause problems if the user expects ø as a variant of o. A user will look for "Søren" between "Sorem" and "Soret", not see it in the selection, and assume the string is missing, confused because it was sorted in a completely different location. In matching, the same can occur, which can cause significant problems for software customers; for example, in a database selection the user may not realize what records are missing. See *Section 1.5, Other Applications of Collation*.

With Unicode applications widely deployed, multilingual data is the rule, not the exception. Furthermore, it is increasingly common to see users with many different sorting expectations accessing the data. For example, a French company with customers all over Europe will include names from many different languages. If a German employee at this French company accesses the data, the customer names need to show up in the order that meets this employee's expectations—that is, in a German order—even though there will be many different accented characters that do not normally appear in German text.

For scripts and characters not used in a particular language, explicit rules may not exist. For example, Swedish and French have clearly specified, distinct rules for sorting ä (either after z or as an accented character with a secondary difference from a), but neither defines the ordering of characters such as Ж, ש, ♫, ∞, ◊, or ⌂.

1.1 Multi-Level Comparison

To address the complexities of language-sensitive sorting, a *multilevel* comparison algorithm is employed. In comparing two words, the most important feature is the identity of the base letters—for example, the difference between an *A* and a *B*. Accent differences are typically ignored, if the base letters differ. Case differences (uppercase versus lowercase), are typically ignored, if the base letters or their accents differ. Treatment of punctuation varies. In some situations a punctuation character is treated like a base letter. In other situations, it should be ignored if there are any base, accent, or case differences. There may also be a final, tie-breaking level, whereby if there are no other differences at all in the string, the (normalized) code point order is used.

Comparison Levels

Level	Description*	Examples
L1	Base characters	role < roles < rule
L2	Accents	role < rôle < roles
L3	Case	role < Role < rôle
L4	Punctuation	role < “role” < Role
Ln	Tie-Breaker	role < ro□le < “role”

These examples are in English; the levels may correspond to different writing system features in other languages. In each example for levels L2 through Ln, the differences on that level (indicated by the underlined characters) are swamped by the stronger-level differences (indicated by the blue text). For example, the L2 example shows that difference between an o and an accented ô is swamped by an L1 difference (the presence or absence of an *s*). In the last example, the □ represents a format character, which is otherwise completely ignorable.

The primary level (L1) is for the basic sorting of the text, and the non-primary levels (L2..Ln) are for adjusting string weights for other linguistic elements in the writing system that are important to users in ordering, but less important than the order of the basic sorting. In practice, fewer levels may be needed, depending on user preferences or customizations.

1.1.1 Collation Order and Code Chart Order

Many people expect the characters in their language to be in the "correct" order in the Unicode code charts. Because collation varies by language and not just by script, it is not possible to arrange the encoding for characters so that simple binary string comparison produces the desired collation order for all languages. Because multi-level sorting is a requirement, it is not even possible to arrange the encoding for characters so that simple binary string comparison produces the desired collation order for any particular language. Separate data tables are required for correct sorting order. For more information on tailorings for different languages, see [CLDR].

The basic principle to remember is: ***The position of characters in the Unicode code charts does not specify their sorting weight.***

1.2 Canonical Equivalence

There are many cases in Unicode where two sequences of characters are canonically equivalent: the sequences represent essentially the same text, but with different actual sequences. For more information, see [UAX15].

Sequences that are canonically equivalent must sort the same. The table below gives some examples of canonically equivalent sequences. For example, the *angstrom sign* was encoded for compatibility, and is canonically equivalent to an *A-ring*. The latter is also equivalent to the decomposed sequence of *A* plus the *combining ring* character. The order of certain combining marks is also irrelevant in many cases, so such sequences must also be sorted the same, as shown in the second example. The third example shows a composed character that can be decomposed in four different ways, all of which are canonically equivalent.

Canonical Equivalence

1	Å	U+212B ANGSTROM SIGN
	Å	U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
	A ◌̊	U+0041 LATIN CAPITAL LETTER A, U+030A COMBINING RING ABOVE
2	x ◌̛ ◌̣	U+0078 LATIN SMALL LETTER X, U+031B COMBINING HORN, U+0323 COMBINING DOT BELOW
	x ◌̣ ◌̛	U+0078 LATIN SMALL LETTER X, U+0323 COMBINING DOT BELOW, U+031B COMBINING HORN
3	ự	U+1EF1 LATIN SMALL LETTER U WITH HORN AND DOT BELOW
	ụ◌̛	U+1EE5 LATIN SMALL LETTER U WITH DOT BELOW, U+031B COMBINING HORN
	u ◌̛ ◌̣	U+0075 LATIN SMALL LETTER U, U+031B COMBINING HORN, U+0323 COMBINING DOT BELOW
	ư ◌̣	U+01B0 LATIN SMALL LETTER U WITH HORN, U+0323 COMBINING DOT BELOW
	u ◌̣ ◌̛	U+0075 LATIN SMALL LETTER U, U+0323 COMBINING DOT BELOW, U+031B COMBINING HORN

1.3 Contextual Sensitivity

There are additional complications in certain languages, where the comparison is context sensitive and depends on more than just single characters compared directly against one another.

The first example of such a complication consists of ***contractions***, where two (or more) characters sort as if they were a single base letter. In the table below, *CH* acts like a single letter sorted after *C*.

The second example consists of ***expansions***, where a single character sorts as if it were a sequence of two (or more) characters. In the table below, an *Œ* ligature sorts as if it were the sequence of *O* + *E*.

Both contractions and expansions can be combined: that is, two (or more) characters may sort as if they were a different sequence of two (or more) characters. In the third example in the table below, for Japanese, a length mark sorts like the vowel of the previous syllable: as an *A after KA* and as an *I* after *KI*.

Context Sensitivity

Contractions	H < Z*, but* CH > CZ
Expansions	OE < Œ < OF
Both	カー < カイ*, but* キー > キイ

Some languages have additional oddities in the way they sort. Normally, all differences in sorting are assessed from the start to the end of the string. If all of the base letters are the same, the first accent difference determines the final order. In row 1 of the following examples, the first accent difference is on the *o*, so that is what determines the order. In French and a few other languages, however, it is the *last* accent difference that determines the order, as shown in row 2.

French Ordering

Normal Accent Ordering	cote < coté < côte < côté
French Accent Ordering	cote < côte < coté < côté

1.4 Customization

In practice, there are additional features of collation that users need to control. These are expressed in user-interfaces and eventually in APIs. Other customizations or user preferences include the following:

- *Language.* This is the most important feature, because it is crucial that the collation match the expectations of users of the target language community.
- *Strength.* This refers to the number of levels that are to be considered in comparison, and is another important feature. Most of the time a three-level strength is needed for comparison of strings. In some cases, a larger number of levels will be needed, while in others—especially in searching—fewer levels will be desired.
- *Case Ordering.* Some dictionaries and authors collate uppercase before lowercase while others use the reverse, so that preference needs to be customizable. Sometimes the case ordering is mandated by the government, as in Denmark. Often it is simply a customization or user preference.

- *Punctuation.* Another common option is whether to treat punctuation (including spaces) as base characters or treat such characters as only making a level 4 difference.
- *User-Defined Rules.* Such rules provide specified results for given combinations of letters. For example, in an index, an author may wish to have symbols sorted as if they were spelled out; thus "?" may sort as if it were the string "question mark".
- *Merged Tailorings.* An option may allow the merging of sets of rules for different languages. For example, someone may want Latin characters sorted as in French, and Arabic characters sorted as in Persian. In such an approach, generally one of the tailorings is designated the “master” in cases of conflicting weights for a given character.
- *Script Order.* A user may wish to specify which scripts come first. For example, in a book index an author may want index entries in the predominant script that the book itself is written in to come ahead of entries for any other script. For example:

  b < ב < β < б [Latin < Hebrew < Greek < Cyrillic] *versus*
  β < b < б < ב [Greek < Latin < Cyrillic < Hebrew]

  Attempting to achieve this effect by introducing an extra strength level before the first (primary) level would give incorrect ordering results for strings which mix characters of more than one script.

- *Numbers.* A customization may be desired to allow sorting numbers in numeric order. If strings including numbers are merely sorted alphabetically, the string “A-10” comes before the string “A-2”, which is often not desired. This behavior can be customized, but it is complicated by ambiguities in recognizing numbers within strings (because they may be formatted according to different language conventions). Once each number is recognized, it can be preprocessed to convert it into a format that allows for correct numeric sorting, such as a textual version of the IEEE numeric format.

Phonetic sorting of Han characters requires use of either a lookup dictionary of words or, more typically, special construction of programs or databases to maintain an associated phonetic spelling for the words in the text.

1.5 Other Applications of Collation

The same principles about collation behavior apply to realms beyond sorting. In particular, searching should behave consistently with sorting. For example, if *v* and *w* are treated as identical base letters in Swedish sorting, then they should also be treated the same for searching. The ability to set the maximal strength level is very important for searching.

Selection is the process of using the comparisons between the endpoints of a range, as when using a SELECT command in a database query. It is crucial that the range returned be correct according to the user's expectations. For example, if a German businessman making a database selection to sum up revenue in each of of the cities from *O...* to *P...* for planning purposes does not realize that all cities starting with *Ö* were excluded because the query selection was using a Swedish collation, he will be one very unhappy customer.

A sequence of characters considered a unit in collation, such as *ch* in Slovak, represents a *tailored* grapheme cluster. For applications of this concept, see UTS #18: Unicode Regular Expressions [UTS18]. For more information on grapheme clusters, see UAX #29, Unicode Text Segmentation [UAX29].

1.6 Interleaved Levels

Levels may also need to be interleaved. For example, the simplest way to sort a database according to two fields is to sort field by field, sequentially. This gives the results in column one in the example below. All the levels in Field 1 are compared first, and then all the levels in Field 2. The problem with this approach is that high-level differences in the second field are swamped by minute differences in the first field, which results in unexpected ordering for the first names.

Merged Fields

Sequential		Weak First		Merged	
F1L1, F1L2, F1L3, F2L1, F2L2, F2L3		F1L1, F2L1, F2L2, F2L3		F1L1, F2L1, F1L2, F2L2, F1L3, F2L3	
di Silva	John	disílva	Fred	di Silva	Fred
di Silva	Fred	diSilva	Fred	diSilva	Fred
diSilva	John	di Silva	Fred	disílva	Fred
diSilva	Fred	di Silva	John	di Silva	John
disílva	John	diSilva	John	diSilva	John
disílva	Fred	disílva	John	disílva	John

A second way to do the sorting is to ignore all but base-level differences in the sorting of the first field. This gives the results in the second column. The first names are all in the right order, but the problem is now that the first field is not correctly ordered except by the base character level.

The correct way to sort two fields is to merge the fields, as shown in the "Merged" column. Using this technique, all differences in the fields are taken into account, and the levels are considered uniformly. Accents in all fields are ignored if there are any base character differences in any of the field, and case in all fields is ignored if there are accent or base character differences in any of the fields.

1.7 Performance

Collation is one of the most performance-critical features in a system. Consider the number of comparison operations that are involved in sorting or searching large databases, for example. Most production implementations will use a number of optimizations to speed up string comparison.

Strings are often preprocessed into sort keys, so that multiple comparisons operations are much faster. With this mechanism, a collation engine generates a *sort key* from any given string. The binary comparison of two sort keys yields the same result (less, equal, or greater) as the collation engine would return for a comparison of the original strings. Thus, for a given collation C and any two strings A and B:

A ≤ B according to C if and only if sortkey(C, A) ≤ sortkey(C, B)

However, simple string comparison is faster for any individual comparison, because the generation of a sort key requires processing an entire string, while differences in most string comparisons are found before all the characters are processed. Typically, there is a considerable difference in performance, with simple string comparison being about 5 to 10 times faster than generating sort keys and then using a binary comparison.

Sort keys, on the other hand, can be much faster for multiple comparisons. Because binary comparison is much faster than string comparison, it is faster to use sort keys whenever there will be more than about 10 comparisons per string, if the system can afford the storage.

1.8 Common Misperceptions

There are a number of common misperceptions about collation. This section corrects those misperceptions.

Collation is not aligned with character sets or repertoires of characters.

Swedish and German share most of the same characters, for example, but have very different sorting orders.

Collation is not code point (binary) order.

A simple example of this is the fact that capital Z comes before lowercase a in the code charts. As noted earlier, beginners may complain that a particular Unicode character is “not in the right place in the code chart.” That is a misunderstanding of the role of the character encoding in collation. While the Unicode Standard does not gratuitously place characters such that the binary ordering is odd, the only way to get the linguistically-correct order is to use a language-sensitive collation, not a binary ordering.

Collation is not a property of strings.

In a list of cities, with each city correctly tagged with its language, a German user will expect to see all of the cities sorted according to German order, and will not expect to see a word with *ö* appear after *z*, simply because the city has a Swedish name. As in the earlier example, it is crucially important that if a German businessman makes a database selection, such as to sum up revenue in each of of the cities from *O...* to *P...* for planning purposes, cities starting with *Ö not* be excluded.

Collation order is not preserved under concatenation or substring operations, in general.

For example, the fact that x is less than y does not mean that x + z is less than y + z, because characters may form contractions across the substring or concatenation boundaries. In summary:~~, the following shows which implications~~

~~*not* to expect.~~

x < y does not imply that xz < yz
x < y does not imply that zx < zy
xz < yz does not imply that x < y
zx < zy does not imply that x < y

Collation order is not preserved when comparing sort keys generated from different collation sequences.

Remember that sort keys are a preprocessing of strings according to a given set of collation features. Different features result in different binary sequences. For example, if there are two collations, F and G, where F is a French collation (with accents compared from the end), and G is a German phonebook ordering, then:

- A ≤ B according to F if and only if sortkey(F, A) ≤ sortkey(F, B), *and*
- A ≤ B according to G if and only if sortkey(G, A) ≤ sortkey(G, B)
- The relation between sortkey(F, A) and sortkey(G, B) says nothing about whether A ≤ B according to F, or whether A ≤ B according to G.

Collation order is not a stable sort.

Stability is a property of a sort algorithm, not of a collation sequence. For more information, see *Section 3.8, Stability*.

Collation order is not fixed.

Over time, collation order will vary: there may be fixes needed as more information becomes available about languages; there may be new government or industry standards for the language that require changes; and finally, new characters added to the Unicode Standard will interleave with the previously-defined ones. This means that collations must be carefully versioned.

1.9 The Unicode Collation Algorithm

The Unicode Collation Algorithm (UCA) details how to compare two Unicode strings while remaining conformant to the requirements of the Unicode Standard. The UCA also supplies the Default Unicode Collation Element Table (DUCET), which is data specifying the default collation order for all Unicode characters. This table is designed so that it can be *tailored* to meet the requirements of different languages and customizations.

Briefly stated, the Unicode Collation Algorithm takes an input Unicode string and a Collation Element Table, containing mapping data for characters. It produces a sort key, which is an array of unsigned 16-bit integers. Two or more sort keys so produced can then be binary-compared to give the correct comparison between the strings for which they were generated.

The Unicode Collation Algorithm assumes multiple-level key weighting, along the lines widely implemented in IBM technology, and as described in the Canadian sorting standard [CanStd] and the International String Ordering standard [ISO14651].

By default, the algorithm makes use of three fully-customizable levels. For the Latin script, these levels correspond roughly to:

1. alphabetic ordering
2. diacritic ordering
3. case ordering.

A final level may be used for tie-breaking between strings not otherwise distinguished.

This design allows implementations to produce culturally acceptable collation, with a minimal burden on memory requirements and performance. In particular, Collation Element Tables only require storage of 32 bits of collation data per significant character.

Implementations of the Unicode Collation Algorithm are not limited to supporting only three levels. They are free to support a fully customizable 4th level (or more levels), as long as they can produce the same results as the basic algorithm, given the right Collation Element Tables. For example, an application which uses the algorithm, but which must treat some collection of special characters as ignorable at the first three levels *and* must have those specials collate in non-Unicode order (for example to emulate an existing EBCDIC-based collation), may choose to have a fully customizable 4th level. The downside of this choice is that such an application will require more storage, both for the Collation Element Table and in constructed sort keys.

The Collation Element Table may be tailored to produce particular culturally required orderings for different languages or locales. As in the algorithm itself, the tailoring can provide full customization for three (or more) levels.

1.9.1 Goals

The algorithm is designed to satisfy the following goals:

1. A complete, unambiguous, specified ordering for all characters in Unicode.
2. A complete resolution of the handling of canonical and compatibility equivalences as relates to the default ordering.
3. A complete specification of the meaning and assignment of collation levels, including whether a character is ignorable by default in collation.
4. A complete specification of the rules for using the level weights to determine the default collation order of strings of arbitrary length.
5. Allowance for override mechanisms (*tailoring*) to create language-specific orderings. Tailoring can be provided by any well-defined syntax that takes the default ordering and produces another well-formed ordering.
6. An algorithm that can be efficiently implemented, in terms of both performance and memory requirements.

Given the standard ordering and the tailoring for any particular language, any two companies or individuals—with their own proprietary implementations—can take any arbitrary Unicode input and produce exactly the same ordering of two strings. In addition, when given a tailoring specifying French accents this algorithm passes the Canadian and ISO 14651 benchmarks ([CanStd], [ISO14651]).

> **Note:** The Default Unicode Collation Element Table does not explicitly list weights for all assigned Unicode characters. However, the algorithm is well defined over *all* Unicode code points. See *Section 7.1.2, Legal Code Points*.

1.9.2 Non-Goals

The Default Unicode Collation Element Table (DUCET) explicitly does not provide for the following features:

1. *Reversibility:* from a Collation Element one is not guaranteed to be able to recover the original character.
2. *Numeric formatting:* numbers composed of a string of digits or other numerics will not necessarily sort in *numerical order.*
3. *API:* no particular API is specified or required for the algorithm.
4. *Title sorting:* removing articles such as *a* and *the* during bibliographic sorting is not provided.
5. *Stability of binary sort key values between versions:* weights in the DUCET may change between versions. For more information, see *Section 3.8, Stability*.
6. *Linguistic applicability:* to meet most user expectations, a linguistic tailoring is needed. For more information, see *Section 5, Tailoring*.

The feature of linguistic applicability deserves further discussion. DUCET does not and cannot actually provide linguistically correct sorting for every language without further tailoring. That would be impossible, due to conflicting requirements for ordering different languages that share the same script. It is not even possible in the specialized cases where a script may be predominantly used by a single language, because of the limitations of the DUCET table design and because of the requirement to minimize implementation overhead for all users of DUCET.

Instead, the goal of DUCET is to provide a reasonable default ordering for all scripts that are *not* tailored. Any characters used in the language of primary interest for collation are expected to be tailored to meet all the appropriate linguistic requirements for that language. For example, for a user interested primarily in the Malayalam language, DUCET would be tailored to get all details correct for the expected Malayalam collation order, while leaving other characters (Greek, Cyrillic, Han, and so forth) in the default order, because the order of those other characters is not of primary concern. Conversely, a user interested primarily in the Greek language would use a Greek-specific tailoring, while leaving the Malayalam (and other) characters in their default order in the table.

2 Conformance

~~There are many different ways to compare strings, and the Unicode Standard does not restrict the ways in which implementations can do this.~~ The Unicode Collation Algorithm does not restrict the many different ways in which implementations can compare strings. However, any Unicode-conformant implementation that purports to implement the Unicode Collation Algorithm must do so as described in this document.

~~**Note:**~~ A conformance test for the UCA is available in [Test].

The algorithm is a *logical* specification~~, designed to be straightforward to describe~~. ~~Actual~~ Implementations ~~of the algorithm~~ are free to change any part of the algorithm as long as any two strings compared by the implementation are ordered the same as they would be by the algorithm as specified. ~~They are also free to~~ Implementations may also use a different format for the data in the Collation Element Table. The sort key is ~~also~~ a logical intermediate object: ~~as long as~~ if an implementation produces the same results in comparison of strings, the sort keys can differ in format from what is specified ~~here~~ in this document. (See *Section 6, Implementation Notes*.)

The conformance requirements ~~for conformance on implementations~~ of the Unicode Collation Algorithm are as follows:

C1 *Given a well-formed Unicode Collation Element Table, a conformant implementation shall replicate the same comparisons of strings as those produced by Section 4, Main Algorithm.*

In particular, a conformant implementation must be able to compare any two canonical~~ly~~-equivalent strings as being equal, for all Unicode characters supported by that implementation.

If a conformant implementation compares strings in a legacy character set, it must provide the same results as if those strings had been transcoded to Unicode.

C2 *A conformant implementation shall support at least three levels of collation.*

A conformant implementation is only required to implement three levels. However, it may implement four (or more) levels if desired.

C3 *A conformant implementation that supports any of the following features: backward levels, variable weighting, and semi-stability, shall do so in accordance with this specification.*

A conformant implementation is not required to support these features; however, if it does~~ so~~, it must interpret them properly. Unless they are functioning in a very restricted domain, it is strongly recommended that implementations support a backwards secondary level, because this is required for French.

C4 *A conformant implementation must specify the version number of this Unicode Technical Standard.*

The precise values of the collation elements for the characters may change over time as new characters are added to the Unicode Standard.

The version number of this document is synchronized with the version of the Unicode Standard ~~for~~ which ~~it~~ specifies the repertoire covered.

C5 *An implementation claiming conformance to Matching and Searching according to UTS #10, shall meet the requirements described in Section 8, Searching and Matching.*

C6 *An implementation claiming conformance to standard UCA parametric tailoring shall do so in accordance with the specifications Section 5, Tailoring.*

An implementation claiming such conformance does not have to support all of the parameter attributes and values; ~~the only requirement is that those that it does claim~~ however, those it claims to support must behave as specified.

3 Collation Element Table

A Collation Element Table contains a mapping from one (or more) characters to one (or more) *collation elements*, where a collation element is an ordered list of three or more 16-bit weights. (All code points not explicitly mentioned in the mapping are given an implicit weight: see *Section 7,* *Weight Derivation*).

> **Note:** Implementations can produce the same result without using 16-bit weights. See *Section 6,* *Implementation Notes*.

3.1 Weight Levels and Notation

The first weight is called the *Level 1* or *primary* weight; the second is called the *Level 2* or *secondary* weight; the third is called the *Level 3* or *tertiary* weight; the fourth is called the *Level 4* or *quaternary* weight, and so on. For a collation element X, these can be abbreviated as X_1, X_2, X_3, X_4, and so on.

Given two collation elements X and Y, ~~we will use~~ this document uses the following notation:

Equals Notation

Notation	Reading	Meaning
$X =_1 Y$	*X is primary equal to Y*	$X_1 = Y_1$
$X =_2 Y$	*X is secondary equal to Y*	$X_2 = Y_2$ and $X =_1 Y$
$X =_3 Y$	*X is tertiary equal to Y*	$X_3 = Y_3$ and $X =_2 Y$
$X =_4 Y$	*X is quaternary equal to Y*	$X_4 = Y_4$ and $X =_3 Y$

Less Than Notation

Notation	Reading	Meaning

$X <_1 Y$	*X is primary less than Y*	$X_1 < Y_1$
$X <_2 Y$	*X is secondary less than Y*	$X <_1 Y$ or ($X =_1 Y$ and $X_2 < Y_2$)
$X <_3 Y$	*X is tertiary less than Y*	$X <_2 Y$ or ($X =_2 Y$ and $X_3 < Y_3$)
$X <_4 Y$	*X is quaternary less than Y*	$X <_3 Y$ or ($X =_3 Y$ and $X_4 < Y_4$)

Other operations are given their customary definitions in terms of the above. That is:

- $X \leq_n Y$ if and only if $X <_n Y$ or $X =_n Y$
- $X >_n Y$ if and only if $Y <_n X$
- $X \geq_n Y$ if and only if $Y \leq_n X$

~~The collation algorithm results in a similar ordering among characters and strings, so that for two strings A and B we can write $A <_2 B$, meaning that A is less than B and there is a primary or secondary difference between them. If $A <_2 B$ but $A=_1 B$, we say that there is *only* a secondary difference between them. If two strings are equivalent (equal at all levels) according to a given Collation Element Table, we write $A \equiv B$. If they are bit-for-bit identical, we write $A = B$.~~

This notation for collation elements is also adapted to refer to ordering between strings, as shown in the following table, where A and B refer to two strings.

Notation for String Ordering

Notation	Meaning
$A <_2 B$	A is less than B, and there is a primary or secondary difference between them
$A <_2 B$ and $A=_1 B$	A is less than B, but there is *only* a secondary difference between them
$A \equiv B$	A and B are equivalent (equal at all levels) according to a given Collation Element Table
$A = B$	A and B are bit-for-bit identical

~~**Note:**~~ Where only plain text ASCII characters are available the following fallback notation can be used:

Fallback Notation

Notation	Fallback
$X <_n Y$	X <[n] Y
X_n	X[n]
$X \leq_n Y$	X <=[n] Y
$A \equiv B$	A =[a] B

If a weight is 0000, then that collation element is *ignorable* at that level: the weight at that level is not taken into account in sorting. A *Level N ignorable* is a collation element that is ignorable at level N but not at level N+1. Thus:

- A *Level 1* or *primary ignorable* is a collation element that is ignorable at Level 1, but not at Level 2.
- A *Level 2* or *secondary ignorable* is ignorable at Levels 1 and 2, but not Level 3.
- A *Level 3* or *tertiary ignorable* is ignorable at Levels 1, 2, and 3 but not Level 4.

In addition:

- A collation element that is not ignorable at any level is called a *non-ignorable*.
- A collation element with zeros at every level is called *completely ignorable.*

For a given Collation Element Table, MIN_n is the least weight in any collation element at level *n*, and MAX_n is the maximum weight in any collation element at level *n*.

3.2 Simple Mappings

Most of the mappings in a collation element table are simple: they consist of the mapping of a single character to a single collation element.

The following ~~are sample collation elements~~ list shows several simple mappings that are used in the examples illustrating the algorithm. Unless otherwise noted, all weights used in the collation elements are in hexadecimal format. The specific weight values shown are illustrative only; they may not match the weights in the latest Default Unicode Collation Element Table [Allkeys].

~~Sample Table~~

Character	Collation Element	Name
0300 "`"	[0000.0021.0002]	COMBINING GRAVE ACCENT
0061 "a"	[06D9.0020.0002]	LATIN SMALL LETTER A
0062 "b"	[06EE.0020.0002]	LATIN SMALL LETTER B
0063 "c"	[0706.0020.0002]	LATIN SMALL LETTER C
0043 "C"	[0706.0020.0008]	LATIN CAPITAL LETTER C
0064 "d"	[0712.0020.0002]	LATIN SMALL LETTER D

> ~~**Note:** Weights in all examples are illustrative, and may not match what is in the latest Default Unicode Collation Element Table.~~

~~3.1 Linguistic Features~~

~~The following section describes the implications of the features discussed in *Section 1, Introduction*.~~

3.3 Multiple Mappings

The mapping from characters to collation elements may not always be a simple mapping from one character to one collation element~~: in general, it may map~~. In general, the mapping may be from one to many, from many to one, or from many to many. ~~The following sections illustrate this.~~

3.3.1 Expansions

The Latin letter *æ* is treated as an independent letter by default. Collations such as English, which may require treating it as equivalent to an <*a e*> sequence, can tailor the letter to map to a sequence of more than one collation elements, such as in the following example:

Character	Collation Element	Name
00E6	[06D9.0020.0002], [073A.0020.0002]	LATIN SMALL LETTER AE; "æ"

In this example, the collation element `[06D9.0020.0002]` gives the weight values for *a*, and the collation element `[073A.0020.0002]` gives the weight values for *e*.

3.3.2 Contractions

Similarly, where *ch* is treated as a single letter, as for instance in traditional Spanish, it is represented as a mapping from two characters to a single collation element, such as in the following example:

Character	Collation Element	Name
0063 0068	[0707.0020.0002]	LATIN SMALL LETTER C, LATIN SMALL LETTER H; "ch"

In this example, the collation element `[0707.0020.0002]` has a primary value one greater than the primary value for the letter *c* by itself, so that the sequence *ch* will collate after *c* and before *d*. This~~e above~~ example shows the result of a tailoring of collation elements to weight sequences of letters as a single unit.

Any character (such as *soft hyphen*) that is not completely ignorable between two characters of a contraction will cause them to sort as separate characters. Thus a soft hyphen can be used to separate and cause distinct weighting of sequences such as Slovak *ch* or Danish *aa* that would normally be weighted as units.

Contractions that end with *non-starter* characters are known as *discontiguous contractions.* For example, suppose that there is a contraction of <***a, combining ring above***>, as in Danish where this sorts as after "z". If the input text contains the sequence <***a****, combining dot below,* ***combining ring above***>, then the contraction still needs to be detected. This is required because the rearrangement of the combining marks is canonically equivalent:

<***a****, combining dot below,* ***combining ring above***>
≡
<***a****,* ***combining ring above****, combining dot below*>.

That is, discontiguous contractions must be detected in input text whenever the final sequence of non-starter characters could be rearranged so as to make a contiguous matching sequence that is canonically equivalent. In the formal algorithm this is handled by rule Rule S2.1. For information on non-starters, see [UAX15].

3.3.3 Many-to-Many Mappings

In some cases a sequence of two or more characters is mapped to a sequence of two or more collation elements. For example, this technique is used in the Default Unicode Collation Element Table [Allkeys] to handle weighting of rearranged sequences of Thai or Lao left-side-vowel + consonant. See *Section 3.5, Rearrangement*.

Both many-to-many mappings and many-to-one mappings are referred to as *contractions* in the discussion of the Unicode Collation Algorithm, even though many-to-many mappings often do not actually shorten anything. The key issue for implementations is that for both many-to-one mappings and many-to-many mappings, the weighting algorithm must first identify a sequence of characters in the input string and "contract" them together as a unit for weight lookup in the table. The identified unit may then be mapped to any number of collation elements. Contractions pose particular issues for implementations, because all eligible contraction targets must be identified first, before the application of simple mappings, so that processing for simple mappings does not bleed away the context needed to correctly identify the contractions.

3.3.4 Other Multiple Mappings

Certain characters may both expand and contract. See *Section 1.3, Contextual Sensitivity.*

3.4 French Accents

In some languages (notably French), accents are sorted from the back of the string to the front of the string. This behavior is not marked in the Default Unicode Collation Element Table, but may occur in tailored tables. In such a case, the collation elements for the accents and their base characters are marked as being *backwards* at Level 2.

3.5 Rearrangement

Certain characters~~ are not coded in logical order~~, such as the Thai vowels เ through ไ ~~and the Lao vowels ເ through ໄ~~ (and related vowels in the Lao and Tai Viet scripts of Southeast Asia), are not represented in strings in logical order. ~~(this list is indicated~~ The exact list of such characters is given by the Logical_Order_Exception property in the Unicode Character Database [UAX44]~~)~~. For collation, they are rearranged by swapping them with the following character before further processing, because logically they belong afterward~~s~~. This is done by providing these sequences as ~~contractions~~ many-to-many mappings in the Collation Element Table.

3.6 Default Unicode Collation Element Table

The Default Unicode Collation Element Table is provided in [AllKeys]. This table provides a mapping from characters to collation elements for all the explicitly weighted characters. The mapping lists characters in the order that they ~~would be~~ are weighted. Any code points that are not explicitly mentioned in this table are given a derived collation element, as described in *Section 7, Weight Derivation*. There are three types of mappings:

- **~~Normal~~Simple.** One Unicode character maps to one collation element.
- **Expansions.** One Unicode character maps to a sequence of collation elements.
- **Contractions.** A sequence of Unicode characters maps to a sequence of (one or more) collation elements.

The Default Unicode Collation Element Table does not aim to provide precisely correct ordering for each language and script; tailoring is required for correct language handling in almost all cases. The goal is instead to have all the *other* characters, those that are not tailored, show up in a reasonable order. ~~In particular,~~ This is particularly true for contractions, because ~~the use of~~ contractions can result in larger tables and significant performance degradation. ~~While~~ Contractions are required in tailorings, but their use is kept to a minimum in the Default Unicode Collation Element Table ~~their use is kept to the bare minimum to avoid such problems~~ to enhance performance.

In the Default Unicode Collation Element Table, contractions are ~~required in those instances~~ necessary where a canonical~~ly~~ decomposable character requires a distinct primary weight in the table, so that the canonical~~ly~~-equivalent character sequences are ~~also~~ given the same weights. For example, Indic two-part vowels have primary weights as units, and their canonical~~ly~~-equivalent sequence of vowel parts must be given the same primary weight by means of a contraction entry in the table. The same applies to a number of precomposed Cyrillic characters with diacritic marks and to a small number of Arabic letters with *madda* or *hamza* marks.

Contractions are also entered in the table for Thai, ~~and~~ Lao, and Tai Viet logical order exception vowels. Because ~~both Thai and Lao both~~ these scripts all have five vowels that are represented in strings in visual order, ~~instead of logical order,~~ the~~y~~ vowels cannot simply be weighted by their representation order in strings. One option is to ~~require~~ preprocess~~ing of Thai and Lao strings,~~ Thai, Lao, and Tai Viet strings to identify and reorder all logical order exception vowels around the following consonant. That approach was used in Version 4.0 ~~(~~and earlier~~)~~ of the UCA. Starting with Version 4.1 of the UCA, contractions for the relevant combinations of Thai and Lao vowel+consonant have been entered in the Default Unicode Collation Element Table instead.

Those are the only ~~two~~ classes of contractions allowed in the Default Unicode Collation Element Table. Generic contractions of the sort needed~~, for example,~~ to handle digraphs such as "ch" in Spanish or Czech sorting, should be dealt with ~~instead~~ in tailorings to the default table~~ -- in part~~ —because they often vary in ordering from language to language, and ~~in part~~ because every contraction entered into the default table has a significant implementation cost for all applications of the default table, even those which may not be particularly concerned with the affected script. See the Unicode Common Locale Data Repository ~~(CLDR)~~ [CLDR] for extensive tailorings of the DUCET for various languages, including those requiring contractions.

Th~~is table~~e Default Unicode Collation Element Table is constructed to be consistent with the Unicode ~~Canonical Equivalence~~ Normalization algorithm, and to respect the Unicode character properties. It is not, however, merely algorithmically derivable ~~from those data~~ based on considerations of canonical equivalence and an inspection of character properties, because the assignment of levels ~~does take~~ also takes into account characteristics of particular scripts. For example, ~~in general~~ the combining marks are generally Level 1 ignorables; however, the Indic combining vowels are given non-zero Level 1 weights, because they are as significant in sorting as the consonants.

Any character may have variant forms or applied accents which affect collation. Thus, for FULL STOP there are three compatibility variants: a fullwidth form, a compatibility form, and a small form. These get different tertiary weights, accordingly. For more information on how the table was constructed, see ~~*Section 7, Weight Derivation*~~ *Section 7.2, Tertiary Weight Table*.

The following table ~~shows the layout~~ summarizes the overall ordering of the collation elements in the Default Unicode Collation Element Table~~, ordered by primary weight:~~. The collation elements are ordered by primary, secondary, tertiary, and Unicode value weights, with primary, secondary, and tertiary weights for variables blanked (replaced by "0000"). Entries in the table which contain a sequence of collation elements have a multi-level ordering applied: comparing the primary weights first, then the secondary weights, and so on. This construction of the table makes it easy to see the order in which characters would be collated.

DUCET ~~Layout~~ Ordering

Values	~~Range~~Type	~~Types~~Examples of Characters
X_1, X_2, $X_3 = 0$	tertiary ignorables	– Control Codes – Format Characters – Hebrew Points – Tibetan Signs ...
X_1, $X_2 = 0$; $X_3 \neq 0$	secondary ignorables	*None in DUCET; could be in tailorings*
$X_1 = 0$; X_2, $X_3 \neq 0$	primary ignorable	– Most nonspacing marks
X_1, X_2, $X_3 \neq 0$	variable	– Whitespace – Punctuation – Symbols
	regular	– Small number of ~~exceptional~~ extender symbols (for example, U+02D0 (ː) MODIFIER LETTER TRIANGULAR COLON) – Numbers – Latin – Greek ...
	implicit	– CJK ~~&~~ Unified Ideographs from the URO and CJK Compatibility blocks ~~(those not decomposed)~~ – CJK Extensions A, ~~&~~ B, C, ... – Unassigned and others given implicit weights
	trailing	*None in DUCET; could be in tailorings*

For most languages, some degree of tailoring is required to match user expectations. For more information, see *Section 5, Tailoring*.

3.6.1 File Format

Each of the files consists of a version line followed by an optional variable-weight line, optional backwards lines, and a series of entries, all separated by newlines. A '#' or '%' and any following characters on a line are comments. Whitespace between literals is ignored. The following is an extended BNF description of the format, where "*x*+" indicates one or more *x*'s, "*x**" indicates zero or more *x*'s, "*x?*" indicates zero or one *x*, and <char> is a hexadecimal Unicode code value.

```
<collationElementTable> := <version>
                           <variable>?
                           <backwards>*
                           <entry>+
```

The version line is of the form:

```
@<version> := <major>.<minor>.<variant> <eol>
```

The variable-weight line has three possible values that may change the weights of collation elements in processing (see *Section 3.6.2, Variable Weighting*). The default is `shifted`.

```
<variable>       := '@variable ' <variableChoice> <eol>
<variableChoice> := 'blanked' | 'non-ignorable' | 'shifted'
```

A backwards line lists a level that is to be processed in reverse order. A forwards line does the reverse. The default is for lines to be forwards.

```
<backwards> := ('@backwards ' | '@forwards ') <levelNumber> <eol>
```

Each entry is a mapping from character(s) to collation element(s), and is of the following form:

```
<entry>       := <charList> ';' <collElement>+ <eol>
<collElement> := "[" <alt> <char> "." <char> "." <char> ("." <char>)* "]"
<alt>         := "*" | "."
```

In the Default Unicode Collation Element Table, the comment may contain informative tags.

Here are some selected entries taken from a particular version of the data file. (It may not match the actual values in the current data file.)

```
0020 ; [*0209.0020.0002.0020] % SPACE
02DA ; [*0209.002B.0002.02DA] % RING ABOVE; COMPATSEQ
0041 ; [.06D9.0020.0008.0041] % LATIN CAPITAL LETTER A
3373 ; [.06D9.0020.0017.0041] [.08C0.0020.0017.0055] % SQUARE AU; COMPATSEQ
00C5 ; [.06D9.002B.0008.00C5] % LATIN CAPITAL LETTER A WITH RING ABOVE; CANONSEQ
212B ; [.06D9.002B.0008.212B] % ANGSTROM SIGN; CANONSEQ
0042 ; [.06EE.0020.0008.0042] % LATIN CAPITAL LETTER B
0043 ; [.0706.0020.0008.0043] % LATIN CAPITAL LETTER C
0106 ; [.0706.0022.0008.0106] % LATIN CAPITAL LETTER C WITH ACUTE; CANONSEQ
0044 ; [.0712.0020.0008.0044] % LATIN CAPITAL LETTER D
```

~~The entries in each file are ordered by collation element, not by character, using a SHIFTED comparison. This makes it easy to see the order in which characters would be collated.~~

Although this document describes collation elements as three levels, the file contains a fourth level (as in `[.0712.0020.0008.`**`0044`**`]`), which is computable. In most cases the fourth level is ~~simply~~ equal to the code point itself. For composite characters which have have collation weights using a sequence of collation elements, the fourth level for each collation element is based on the decomposition of the character. For completely ignorable collation elements, the fourth level is set to zero. For more information on the use of the fourth level and stable sorts, see *Section 3.8, Stability*.

Implementations can also add more customizable levels, as discussed ~~above under conformance~~ in *Section 2, Conformance*. For example, an implementation might want to ~~be capable not only of~~ handl~~ing~~e the standard Unicode Collation, but also be capable of emulating an EBCDIC multi-level ordering (having a fourth-level EBCDIC binary order).

3.6.2 Variable Weighting

Collation elements that are marked with an asterisk in a Unicode Collation Element Table are known as *variable collation elements.*

Character	Collation Element	Name
0020 " "	[*0209.0020.0002]	SPACE

Based on the setting of the variable weighting tag, collation elements can be either treated as ignorables or not. When they are treated as ignorables, ~~then~~ any sequence of ignorable characters that immediately follows the variable collation element is also affected.

There are ~~three~~ four possible options for variable weighted characters:

- **Blanked:** Variable collation elements and any subsequent ignorables are reset so that their weights at levels one through three are zero. For example,
    - *SPACE* would have the value `[.0000.0000.0000]`
    - A combining grave accent after a space would have the value `[.0000.0000.0000]`
    - *Capital A* would be unchanged, with the value `[.06D9.0020.0008]`
    - A combining grave accent after a *Capital A* would be unchanged
- **Non-ignorable:** Variable collation elements are not reset to be ignorable, but get the weights explicitly mentioned in the file.
    - *SPACE* would have the value `[.0209.0020.0002]`
    - *Capital A* would be unchanged, with the value `[.06D9.0020.0008]`
    - Ignorables are unchanged.
- **Shifted:** Variable collation elements are reset to zero at levels one through three. In addition, a new fourth-level weight is appended, whose value depends on the

type, as shown in the following table.

L4 Weights for Shifted Variables

Type	L4	Examples
Tertiary Ignorable	0000	NULL `[.0000.0000.0000.0000]`
Primary or Secondary Ignorable, following a Variable	0000	COMBINING GRAVE `[.0000.0000.0000.0000]`
Variable	old L1	SPACE `[.0000.0000.0000.0209]`
None of the above	FFFF	*Capital A* `[.06D9.0020.0008.FFFF]`

Any subsequent primary or secondary ignorables following a variable are reset so that their weights at levels one through four are zero.

- A combining grave accent after a space would have the value `[.0000.0000.0000.0000]`.
- A combining grave accent after a *Capital A* would be unchanged.

The *shifted* option provides for improved orderings when the variable collation elements are ignorable, while still only requiring three fields to be stored in memory for each collation element. It ~~does~~ results in somewhat longer sort keys, although they can be compressed (see *Section 6.1, Reducing Sort Key Lengths* and Section 6.3, *Reducing Table Sizes)*.

- ~~A fourth possible option would be~~ **Shift-Trimmed:** This option is the same as **Shifted**, except that all trailing FFFFs are trimmed from the sort key. This could be used to emulate POSIX behavior.

The following ~~gives an~~ example ~~of~~ shows the differences between orderings using the different options for variable collation elements. In this example, sample strings differ by the third character: a letter, *space,* '-' *hyphen-minus (002D)*, or '‐' *hyphen (2010);* followed by an uppercase/lowercase distinction. In the first column ~~below~~, the words with *hyphen-minus* ~~and~~ or *hyphen* are separated by "deluge", because ~~an l~~ the letter "l" comes between them in Unicode code order. In the second column, they are grouped together but before all letters in the third position. This is because they are no longer ignorable, and have primary values that differ from the letters. In the third column, the *hyphen-minus* and *hyphen* are grouped together, and their differences are less significant than ~~between the deluge~~ the casing differences in the letter "l". ~~In this case, it is because~~ This grouping results from the fact that they are ignorable, but their fourth level differences are according to the original primary order, which is more intuitive than Unicode order.

Comparison of Variable Ordering

Blanked	Non-ignorable	Shifted	Shift-Trimmed

death	de luge	death	death
de luge	de Luge	de luge	deluge
de-luge	de-luge	de-luge	de luge
deluge	de-Luge	de-luge	de-luge
de-luge	de-luge	deluge	de-luge
de Luge	de-Luge	de Luge	deLuge
de-Luge	death	de-Luge	de Luge
deLuge	deluge	de-Luge	de-Luge
de-Luge	deLuge	deLuge	de-Luge
demark	demark	demark	demark

Primaries for variable collation elements are not *interleaved* with other primary weights. This allows for more compact storage of memory tables. Rather than using a bit per collation element to determine whether the collation element is variable, the implementation only needs to store the maximum primary value for all the variable elements. All collation elements with primary weights from 1 to that maximum are variables; all other collation elements are not.

3.6.3 Default Values

~~Both~~ In the Default Unicode Collation Element Table and in typical tailorings, most unaccented letters differ in the primary weights, but have secondary weights (such as a_1) equal to MIN_2. The primary ignorables will have secondary weights greater than MIN_2. Characters that are compatibility or case variants will have equal primary and secondary weights (for example, $a_1 = A_1$ and $a_2 = A_2$), but have different tertiary weights (for example, $a_3 < A_3$). The unmarked characters will have tertiary weights (such as a_3) equal to MIN_3.

~~However, a well-formed Unicode Collation Element Table~~ This use of secondary and tertiary weights does not guarantee that the meaning of a secondary or tertiary weight is uniform across tables. For example, a *capital A* and *katakana ta* could both have a tertiary weight of 3.

3.7 Well-Formed Collation Element Tables

A well-formed Collation Element Table meets the following conditions:

1. Except in special cases detailed in *Section 6.2, Large Weight Values*, no collation element can have a zero weight at Level N and a non-zero weight at Level N-1.

   For example, the secondary can only be ignorable if the primary is ignorable. ~~The reason for this will be explained under~~ See Step 4 of the main algorithm for an explanation.

2. All Level N weights in Level N-1 ignorables must be strictly less than all weights in Level N-2 ignorables.

   For example, secondaries in non-ignorables must be strictly less than those in primary ignorables: Given collation elements [C, D, E] and [0, A, B], where C ≠ 0 and A ≠ 0, D *must be* less than A.

3. No variable collation element has an ignorable primary.
4. For all variable collation elements U, V, if there is a collation element W such that $U_1 \leq W_1$ and $W_1 \leq V_1$, then W is also variable.

   This provision prevents interleaving~~, mentioned above~~.

3.8 Stability

The notion of *stability* in sorting often causes confusion when discussing collation.

A *stable sort* is one where two records with a field that compares as equal will retain their order if sorted according to that field. This is a property of the sorting algorithm, *not* of the comparison mechanism. For example, a bubble sort is stable, while a quick sort is not. This is a useful property, but cannot be accomplished by modifications to the comparison mechanism or tailorings.

A *semi-stable collation* is different. It is a collation where strings that are not canonical equivalents will not be judged to be equal. This is a property of the comparison, not of the sorting algorithm. ~~In general~~ This is not a particularly useful property—its implementation also ~~typically~~ requires extra processing in string comparison or an extra level in sort keys, and thus may degrade performance to little purpose. However, if a semi-stable collation is required, the specified mechanism is to append the NFD form of the original string after the sort key, in *Section 4.3, Form Sort Key*. See also *Appendix A, Deterministic Sorting*.

The fourth-level weights in the Default Collation Element Table can be used to provide an approximation of a semi-stable collation.

Neither ~~one~~ of the above refers to the stability of the Default Collation Element Table itself. ~~For any particular version of the UCA,~~ The contents of that table will remain unchanged in any particular version of the UCA. However, the contents may~~, however,~~ change between successive versions of the UCA as new characters are added, or ~~as~~ more information is obtained about existing characters.

Implementers should be aware that using different versions of the UCA~~, as well as~~ or different versions of the Unicode Standard~~,~~ could result in different collation results of their data. There are numerous ways collation data could vary across versions, for example:

1. Code points that were unassigned in a previous version of the Unicode Standard are now assigned in the current version, and ~~as such,~~ will have a sorting semantic appropriate to the repertoire to which they belong. For example, the code points U+103D0..U+103DF were undefined in Unicode 3.1. Because they were assigned characters in Unicode 3.2, their sorting semantics and respective sorting weights ~~will~~ changed as of that version.
2. Certain semantics of the Unicode standard could change between versions, such that code points are treated in a manner different than previous versions of the standard~~ (for example, normalization errata)~~.

3. More information is gathered about a particular script, and ~~in order~~ the weight of a code point may need to be adjusted to provide a more linguistically accurate sort~~, the weight of a code point may need to be adjusted~~.

Any of these reasons could necessitate a change between versions with regards to ~~sort~~ collation weights for code points~~, and as such, it is~~. It is therefore important that the implementers specify the version of the UCA, as well as the version of the Unicode Standard under which their data is sorted.

4 Main Algorithm

The main algorithm has four steps. First is to normalize each input string, second is to produce an array of collation elements for each string, and third is to produce a sort key for each string from the collation elements. Two sort keys can then be compared with a binary comparison; the result is the ordering for the original strings.

4.1 Normalize

Step 1. Produce a normalized form of each input string, applying S1.1.

S1.1 Use the Unicode canonical algorithm to decompose characters according to the canonical mappings. That is, put the string into Normalization Form D (see [UAX15]).

- Conformant implementations may skip this step *in certain circumstances:* see ~~*Section 7, Weight Derivation*~~ *Section 6.5, Avoiding Normalization* for more information.

4.2 Produce Array

Step 2. The collation element array is built by sequencing through the normalized form ~~as follows:~~, applying S2.1 through S2.6

> ~~**Note:** A non-starter in a string is called *blocked* if there is another non-starter of the same canonical combining class or zero between it and the last character of canonical combining class 0.~~

String to Collation Element Array

Normalized String	Collation Element Array
ca´b	[0706.0020.0002], [06D9.0020.0002], [0000.0021.0002], [06EE.0020.0002]

S2.1 Find the longest initial substring S at each point that has a match in the table.

S2.1.1 If there are any non-starters following S, process each non-starter C.

S2.1.2 If C is not blocked from S, find if S + C has a match in the table.

> **Note:** A non-starter in a string is called *blocked* if there is another non-starter of the same canonical combining class or zero between it and

the last character of canonical combining class 0.

S2.1.3 If there is a match, replace S by S + C, and remove C.

S2.2 Fetch the corresponding collation element(s) from the table if there is a match. If there is no match, synthesize a weight as described in *Section 7.1, Derived Collation Elements*.

S2.3 Process collation elements according to the variable-weight setting, as described in *Section 3.6.2, Variable Weighting*.

S2.4 Append the collation element(s) to the collation element array.

S2.5 Proceed to the next point in the string (past S).

S2.6 Loop until the end of the string is reached.

> **Note:** ~~The reason for considering~~ The extra non-starter C ~~is that~~ needs to be considered in Step 2.1.1 because otherwise irrelevant characters could interfere with matches in the table. For example, suppose that the contraction <*a, combining_ring*> (= *å*) is ordered after *z*. If a string consists of the three characters <*a, combining_ring, combining_cedilla*>, then the normalized form is <*a, combining_cedilla, combining_ring*>, which separates the *a* from the *combining_ring*. ~~If we did not have the step of~~ Without considering the extra non-starter, this string would compare incorrectly as after *a* and not after *z*.
>
> If the desired ordering treats <*a, combining_cedilla*> as a contraction which should take precedence over <*a, combining_ring*>*,* then an additional mapping for the combination <*a, combining_ring, combining_cedilla*> can be introduced to produce this effect.
>
> ~~**Note:**~~ For conformance to Unicode canonical equivalence, only unblocked non-starters are matched in Step 2.1.2. For example, <*a, combining_macron, combining_ring*> would compare as after *a-macron*, and not after *z*. A~~s in the previous note, a~~dditional mappings can be added to customize behavior.

Example:

normalized string:	ca´b
collation element array:	`[0706.0020.0002], [06D9.0020.0002], [0000.0021.0002], [06EE.0020.0002]`

4.3 Form Sort Key

Step 3. The sort key is formed by successively appending all non-zero weights from the collation element array. The weights are appended from each level in turn, from 1 to 3. (Backwards weights are inserted in reverse order.)

Collation Element Array to Sort Key

Collation Element Array	Sort Key
[0706.0020.0002], [06D9.0020.0002], [0000.0021.0002], [06EE.0020.0002]	0706 06D9 06EE 0000 0020 0020 0021 0020 0000 0002 0002 0002 0002

An implementation may allow the maximum level to be set to a smaller level than the available levels in the collation element array. For example, if the maximum level is set to 2, then level 3 and higher weights are not appended to the sort key. Thus any differences at levels 3 and higher will be ignored, leveling any such differences in string comparison.

Here is a more detailed statement of the algorithm:

S3.1 For each weight level L in the collation element array from 1 to the maximum level,

S3.2 If L is not 1, append a _level separator_*

Note: The level separator is zero (0000), which is guaranteed to be lower than any weight in the resulting sort key. This guarantees that when two strings of unequal length are compared, where the shorter string is a prefix of the longer string, the longer string is always sorted after the shorter—in the absence of special features like contractions. For example: "abc" < "abcX" where "X" can be any character(s).

S3.3 If the collation element table is forwards at level L,

S3.4 For each collation element CE in the array

S3.5 Append CE_L to the sort key if CE_L is non-zero.

S3.6 Else the collation table is backwards at level L, so

S3.7 Form a list of all the non-zero CE_L values.

S3.8 Reverse that list

S3.9 Append the CE_L values from that list to the sort key.

~~***** The level separator is zero (0000), which is guaranteed to be lower than any weight in the resulting sort key. This guarantees that when two strings of unequal length are compared, where the shorter string is a prefix of the longer string, the longer string is always sorted after the shorter (in the absence of special features like contractions). For example:~~

~~"abc" < "abcX" where "X" can be any character(s)~~

S3.10 If a semi-stable sort is required, then after all the level weights have been added, append a copy of the NFD version of the original string. This strength level is called "identical". (See also _Appendix A: Deterministic_Sorting_.)

Example:

collation element array:	`[0706.0020.0002], [06D9.0020.0002], [0000.0021.0002], [06EE.0020.0002]`
sort key:	`0706 06D9 06EE 0000 0020 0020 0021 0020 0000 0002 0002 0002 0002`

4.4 Compare

Step 4. Compare the sort keys for each of the input strings, using a binary comparison. This means that:

- Level 3 differences are ignored if there are any Level 1 or 2 differences.
- Level 2 differences are ignored if there are any Level 1 differences.
- Level 1 differences are never ignored.

~~*Example:*~~Comparison of Sort Keys

	String	Sort Key
1	cab	**0706** 06D9 06EE 0000 0020 0020 **0020** 0000 **0002** 0002 0002
2	Cab	**0706** 06D9 06EE 0000 0020 0020 **0020** 0000 **0008** 0002 0002
3	cáb	**0706** 06D9 06EE 0000 0020 0020 **0021** 0020 0000 0002 0002 0002 0002
4	dab	**0712** 06D9 06EE 0000 0020 0020 0020 0000 0002 0002 0002

In this example, "cab" $<_3$ "Cab" $<_2$ "cáb" $<_1$ "dab". The differences that produce the ordering are shown by the **bold underlined** items:

- For ~~the first two~~ strings 1 and 2, the first difference is in **`0002`** versus **`0008`** (Level 3).
- For ~~the middle two~~ strings 2 and 3, the first difference is in **`0020`** versus **`0021`** (Level 2).
- For ~~the last two~~ strings 3 and 4, the first difference is in **`0706`** versus **`0712`** (Level 1).

~~**Note:** At this point we can explain the reason for only allowing~~ Only well-formed weights~~.~~ are allowed because if ill-formed weights were allowed, the ordering of elements ~~can~~ could be incorrectly reflected in the sort key. For example, suppose the secondary weights of the Latin characters were zero (ignorable) and that (as normal) the primary weights of case-variants are equal: that is, $a_1 = A_1$. Then the following incorrect keys would be generated:

1. ~~*"áe" = <a, acute, e> => [a$_1$ e$_1$ 0000 acute$_2$ 0000 **a$_3$** acute$_3$ e$_3$...]*~~
2. ~~*"Aé" = <A, e, acute> => [a$_1$ e$_1$ 0000 acute$_2$ 0000 **A$_3$** acute$_3$ e$_3$...]*~~

Order	String	Normalized	Sort Key
1	"áe"	a, acute, e	a$_1$ e$_1$ 0000 acute$_2$ 0000 **a$_3$** acute$_3$ e$_3$...
2	"Aé"	A, e, acute	a$_1$ e$_1$ 0000 acute$_2$ 0000 **A$_3$** acute$_3$ e$_3$...

Because the secondary weights for *a, A,* and *e* are lost in forming the sort key, the relative order of the acute is also lost, resulting in an incorrect ordering based solely on the case of *A* versus *a*. With well-formed weights, this does not happen, and ~~you get~~ the following correct ordering is obtained:

1. ~~*"Aé" = <A, e, acute> => [a_1 e_1 0000 a_2 **e_2** $acute_2$ 0000 a_3 $acute_3$ e_3...]*~~
2. ~~*"áe" = <a, acute, e> => [a_1 e_1 0000 a_2 **$acute_2$** e_2 0000 A_3 $acute_3$ e_3...]*~~

Order	String	Normalized	Sort Key
1	"Aé"	A, e, acute	a_1 e_1 0000 a_2 **e_2** $acute_2$ 0000 a_3 $acute_3$ e_3...
2	"áe"	a, acute, e	a_1 e_1 0000 a_2 **$acute_2$** e_2 0000 A_3 $acute_3$ e_3...

However, there are circumstances—typically in expansions—where higher-level weights in collation elements can be zeroed (resulting in ill-formed collation elements) without consequence (see *Section 6.2, Large Weight Values*). Implementations are free to do this as long as they produce the same result as with well-formed tables.

5 Tailoring

Tailoring is any well-defined syntax that takes the Default Unicode Collation Element Table and produces another well-formed Unicode Collation Element Table. This syntax can provide linguistically-accurate collation, if desired. Such syntax will usually allow ~~for~~ the following~~ capabilities~~:

1. Reordering any character (or contraction) with respect to others in the standard ordering. ~~Such a~~ The reordering can represent a Level 1 difference, Level 2 difference, Level 3 difference, or identity (in levels 1 to 3). Because such reordering includes sequences, arbitrary multiple mappings can be specified.

2. Setting the secondary level to be backwards (French) or forwards (normal).

3. Set variable weighting options.

4. Customizing the exact list of variable collation elements.

For best interoperability, it is recommended that tailorings for particular locales (or languages) make use of the tables provided in the Unicode Common Locale Data Repository [CLDR].

For an examples of a tailoring syntax, see *Section 5.2, Tailoring Example: Java*.

For a discussion of more powerful tailoring features, see [ICUCollator]. For details on a common XML format for tailorings, see [LDML].

5.1 Parametric Tailoring

Parametric tailoring, if supported, is specified using a set of attribute-value pairs that specify a particular kind of behavior relative to the UCA. The standard parameter names (attributes) and their posible values are listed in the table **Collation Parameters**, and follow those defined in the Unicode *Locale Data Markup Language* [LDML] in the table *Collation Settings* (*Section 5.1~~3~~4.3, Setting Options*). The bold values are the defaults for the UCA.

Collation Parameters

Attribute	Options	Description
locale (or language)	*locale_id*	Specifies the tailoring rules for the language and/or variant. The locale_id for locale or language uses the syntax from [LDML], *Section 3, Unicode Language and Locale Identifiers*. Unless otherwise specified, tailoring by locale uses the tables from the Unicode Common Locale Data Repository [CLDR]. ~~Such a~~ This choice may override the defaults for the attributes given below. ~~The default if nothing is specified is the~~ **UCA behavior** is the default if nothing is specified.
strength	primary (1) secondary (2) **tertiary** (3) quaternary (4) identical (5)	Sets the default strength for comparison, as described in the UCA. The parenthesized numbers are alternate forms.
alternate	non-ignorable **shifted** *blanked*	Sets alternate handling for variable weights, as described in Section 3.6.2, Variable Weighting. Note that in [LDML], *blanked* is not supported, and **shifted** is the default.
backwards	on **off**	Sets the comparison for the second level *only* to be backwards ("French"), as described in 3.2 French Accents and specified in S3.3–S3.6. The default is **on** for the French locales and **off** for others.
normalization	**on** off	Conformant implementations may skip **[S1.1]** in certain circumstances. If *on*, then the normal UCA algorithm is used. If *off*, all strings that normalized will sort correctly, but others won't necessarily sort correctly. So it should only be set *off* if the the strings to be compared are normalized. (It is recommended that implementations correctly sort all strings that are in the format known as "Fast C or D form" (FCD) even if normalization is *off*. For more information on FCD, see [UTN5].)
caseLevel	on **off**	If set to *on,* a level consisting only of case characteristics will be inserted in front of tertiary level. To ignore accents but take cases into account, set strength to primary and case level to *on*.

caseFirst	upper lower **off**	If set to *upper*, causes uppercase to sort before lowercase. If set to *lower*, lowercase will sort before uppercase. Useful for locales that have already supported ordering but require different order of cases. Affects case and tertiary levels.
hiraganaQuaternary	on **off**	Controls special treatment of Hiragana code points on quaternary level. If turned *on*, Hiragana codepoints will get lower values than all the other non-variable code points. The strength must be greater or equal than quaternary ~~if you want~~ for this attribute to take effect. The default is **on** for the Japanese locales and **off** for others.
numeric	on **off**	If set to *on*, any sequence of Decimal Digits (General_Category = Nd in the Unicode Character Database [UAX44]) is sorted at a primary level with its numeric value. For example, "A-21" < "A-123".
variableTop	*uXXuYYYY*	The parameter value is an encoded Unicode string, with code points in hex, leading zeros removed, and "u" inserted between successive elements. Sets the default value for the variable top. All the code points with primary strengths less than variable top will be considered variable, and thus affected by the alternate handling. If not specified, then the default is the value in DUCET.
match-boundaries:	none whole-character whole-word	Defined in *Section 8, Searching and Matching.*
match-style	minimal medial maximal	Defined in *Section 8, Searching and Matching.*

5.2 Tailoring Example: Java

Java 2 implements a number of the tailoring features described in this document. The following summarizes these features. (For more information, see ~~Collator on~~ [JavaCollator].)~~.~~

~~1.~~ Java does not use a default table in the Unicode Collation Element format; instead it always uses a tailoring syntax. Here is a description of the entries:

Java Tailoring Syntax

~~Java~~ Syntax	Description
& y < x	Make x primary-greater than y
& y ; x	Make x secondary-greater than y
& y , x	Make x tertiary-greater than y
& y = x	Make x equal to y

Either *x* or *y* in this syntax can ~~be~~ represent more than one character, to handle contractions and expansions. The character U+0000 NULL is completely ignorable, so by using the above operations, various levels of ignorable characters can be specified.

~~2.~~ Entries for tailoring can be abbreviated in a number of ways:

- They do not need to be separated by newlines.
- Characters can be specified directly, instead of using their hexadecimal Unicode values.
- ~~Wherever you have~~ In rules of the form "x < y & y < z", ~~you can omit~~ "& y" can be omitted, leaving just "x < y < z".

These abbreviations can be ~~done~~ applied successively, so the following are equivalent in ordering.

Equivalent Tailorings

Java	Unicode Collation Element Table

a, A ; à, À < b, B	`0061 ; [.0001.0001.0001] % a`
	`0040 ; [.0001.0001.0002] % A`
	`00E0 ; [.0001.0002.0001] % à`
	`00C0 ; [.0001.0002.0002] % à`
	`0042 ; [.0002.0001.0001] % b`
	`0062 ; [.0002.0001.0002] % B`

5.3 Use of Combining Grapheme Joiner

The Unicode Collation Algorithm involves the normalization of Unicode text strings before collation weighting. ~~The~~ U+034F COMBINING GRAPHEME JOINER (CGJ) is ordinarily ignored in collation key weighting in the UCA, but it can be used to block the reordering of combining marks in a string as described in [Unicode]. In that case, its effect can be to invert the order of secondary key weights associated with those combining marks. Because of this, the two strings would have distinct keys, making it possible to treat them distinctly in searching and sorting without having to further tailor either the combining grapheme joiner or the combining marks ~~themselves~~.

The CGJ can also be used to prevent the formation of contractions in the Unicode Collation Algorithm. Thus, for example, while *ch* is sorted as a single unit in a tailored Slovak collation, the sequence <*c*, CGJ, *h*> will sort as a *c* followed by an *h*. This can also be used in German, for example, to force *ü* to be sorted as *u* + *umlaut* (thus $u <_2$ *ü*), even where a dictionary sort is being used (which would sort $ue <_3$ *ü)*. This happens without having to further tailor either the combining grapheme joiner or the sequence.

> **Note:** As in a few other cases in Unicode, ~~such as U+200B ZERO WIDTH SPACE (which is not a white space character),~~ the name of the CGJ ~~is~~ can be misleading: the usage above is in some sense the inverse of "joining".

Sequences of characters which include the combining grapheme joiner or other completely ignorable characters may also be given tailored weights. Thus the sequence <c, CGJ, h> could be weighted completely differently from ~~the~~ either the contraction "ch" or ~~how~~ the sequence "c" ~~and~~ followed by "h" ~~would have sorted~~ without the contraction. However, this application of CGJ is not recommended, because it would produce effects much different than the normal usage above, which is to simply interrupt contractions.

5.4 Preprocessing

In addition to tailoring, some implementations may choose to preprocess the text for special purposes. Once such preprocessing is done, the standard algorithm can be applied.

Examples include:

- mapping "McBeth" to "MacBeth"
- mapping "St." to "Street" or "Saint", depending on the context
- dropping articles, such as "a" or "the"
- using extra information, such as pronunciation data for Han characters

Such preprocessing is outside of the scope of this document.

6 Implementation Notes

As noted above for efficiency, implementations may vary from this logical algorithm as long as they produce the same result. The following items discuss various techniques that can be used for reducing sort key length, reducing table sizes, customizing for additional environments, searching, and other topics.

6.1 Reducing Sort Key Lengths

The following discuss methods of reducing sort key lengths. If these methods are applied to all of the sort keys produced by an implementation, they can result in significantly shorter and more efficient sort keys while retaining the same ordering.

6.1.1 Eliminating Level Separators

Level separators are not needed between two levels in the sort key, if the weights are properly chosen. For example, if all L3 weights are less than all L2 weights, then no level separator is needed between them. If there is a fourth level, then the separator before it needs to be retained.

For example, here is a sort key with these level separators removed.

String	Technique(s) Applied	Sort Key

càb	none	0706 06D9 06EE **0000** 0020 0020 0021 0020 **0000** 0002 0002 0002 0002
càb	1	0706 06D9 06EE 0020 0020 0021 0020 0002 0002 0002 0002

While this technique is relatively easy to implement, it can interfere with other compression methods.

6.1.2 L2/L3 in 8 Bits

The L2 and L3 weights commonly are small values. Where that condition occurs for all possible values, they can then be represented as single 8-bit quantities.

Here is the above example with both these changes (and grouping by bytes). Note that the separator has to remain after the primary weight when combining these techniques. If any separators are retained (such as before the fourth level), they need to have the same width as the previous level.

String	Technique(s) Applied	Sort Key
càb	none	07 06 06 D9 06 EE **00** **00** 00 20 00 20 00 21 00 20 **00** **00** 00 02 00 02 00 02 00 02
càb	1, 2	07 06 06 D9 06 EE **00** **00** 20 20 21 20 02 02 02 02

6.1.3 Machine Words

The sort key can be represented as an array of different quantities depending on the machine architecture. For example, comparisons as arrays of 32-bit quantities may be much faster on some machines. If this is done, the original is to be padded with trailing (not leading) zeros as necessary.

String	Technique(s) Applied	Sort Key
càb	1, 2	07 06 06 D9 06 EE 00 00 20 20 21 20 02 02 02 02
càb	1, 2, 3	070606D9 06EE0000 20202120 02020202

6.1.4 Run-Length Compression

Generally sort keys do not differ much in the secondary or tertiary weights, so you tend to end up with keys with a lot of repetition. This also occurs with quaternary weights generated with the shifted parameter. By the structure of the collation element tables, there are also many weights that are never assigned at a given level in the sort key. You can take advantage of these regularities in these sequences to compact the length — while retaining the same sort sequence — by using the following technique. (There are other techniques that can also be used.)

This is a logical statement of the process: the actual implementation can be much faster and performed as the sort key is being generated.

- For each level ***n,*** find the most common value COMMON produced at that level by the collation element table for typical strings. For example, for the Default Unicode Collation Element Table, this is:
  - 0020 for the secondaries (corresponding to unaccented characters)
  - 0002 for tertiaries (corresponding to lowercase or unmarked letters)
  - FFFF for quaternaries (corresponding to non-ignorables with the shifted parameter)
- Reassign the weights in the collation element table at level ***n*** to create a gap of size GAP above COMMON. Typically for secondaries or tertiaries this is done after the values have been reduced to a byte range by the above methods. Here is a mapping that moves weights up or down to create a gap in a byte range.

  ```
  w -> w + 01 - MIN, for MIN <= w < COMMON
  w -> w + FF - MAX, for COMMON < w <= MAX
  ```
- At this point, weights go from 1 to MINTOP, and from MAXBOTTOM to MAX. These new unassigned values are used to run-length encode sequences of COMMON weights.
- When generating a sort key, look for maximal sequences of **m** COMMON values in a row. Let W be the weight right after the sequence.
  - If W < COMMON (or there is no W), replace the sequence by a synthetic low weight equal to (MINTOP + m).
  - If W > COMMON, replace the sequence by a synthetic high weight equal to (MAXBOTTOM - m).

  In the following example, the low weights are 01, 02; the high weights are FE, FF; and the common weight is 77.

  ~~Examples~~**Run-Length Compression**

Original Weights	Compressed Weights
01	01
02	02
77 01	03 01
77 02	03 02
77 77 01	04 01
77 77 02	04 02
77 77 77 01	05 01
77 77 77 02	05 02
...	...
77 77 77 FE	FB FE
77 77 77 FF	FB FF
77 77 FE	FC FE
77 77 FF	FC FF
77 FE	FD FE
77 FF	FD FF
FE	FE
FF	FF

- The last step is a bit too simple, because we have to keep the synthetic weights from colliding with other values with long strings of COMMON weights. This is done by using a sequence of synthetic weights, absorbing as much length into each one as possible. This is done by defining a value BOUND between MINTOP and MAXBOTTOM. The exact value for BOUND can be chosen based on the expected frequency of synthetic low weights versus high weights for the particular collation element table.

  - If a synthetic low weight would not be less than BOUND, use a sequence of low weights of the form (BOUND-1)..(BOUND-1)(MINTOP + remainder) to express the length of the sequence.
  - Similarly, if a synthetic high weight would be less than BOUND, use a sequence of high weights of the form (BOUND)..(BOUND)(MAXBOTTOM - remainder).

This process results in keys that are never longer than the original, are generally much shorter, and result in the same comparisons.

6.2 Large Weight Values

If a collation sequence requires more than 65,535 weight values (or 65,024 values where zero bytes are avoided), this can still be accommodated by using multiple collation elements for a single character. For example, suppose that 50,000 UTF-16 supplementary characters are assigned in a particular implementation, and that these are to be sorted after X. Simply assign them all dual collation elements of the form

$$[(X_1+1).0000.0000], [yyyy.zzzz.wwww]$$

If there was an element with the primary weight (X_1+1), then it also needs to be converted into a dual collation element.

The characters will then sort properly with respect to each other and to the rest of the characters. The first collation element is one of the instances where ill-formed collation elements are allowed. Because the second collation element is well-formed and the first element will only occur in combination, ordering is preserved.

6.3 Reducing Table Sizes

The data tables required for full Unicode sorting can be quite sizable. This section discusses ways to significantly reduce the table size in memory. These have very important implications for implementations.

6.3.1 Contiguous Weight Ranges

The Default Unicode Collation Element Table has secondary weights that are greater than 00FF. This is the result of the derivation described in *Section 7, Weight Derivation*. However, these values can be compacted to a range of values that do not exceed 00FF. Whenever collation elements have different primary weights, the ordering of their secondary weights is immaterial. Thus all of the secondaries that share a single primary can be renumbered to a contiguous range without affecting the resulting order. Composite characters still need to be handled correctly if normalization is avoided as discussed in ~~*Section 7, Weight Derivation*~~ *Section 6.5, Avoiding Normalization*.

For example, for the primary value 0820 (for the letter O), there are 31 distinct secondary values ranging from 0020 to 012D. These can be renumbered to the contiguous range from 0020 to 003F, which is less than 00FF.

6.3.2 Escape Hatch

Although the secondary and tertiary weights for the Default Unicode Collation Element Table can both fit within one byte, of course, any particular tailored table could conceivably end up with secondary or tertiary weights that exceed what can be contained in a single byte. However, the same technique used for large weight values can also be used for implementations that do not want to handle more than 00FF values for a particular weight.

For example, the Java collation implementation only stored 8-bit quantities in level 2 and level 3. However, characters can be given L2 or L3 weights with greater values by using a series of two collation elements. For example, with characters requiring 2,000 weights at L2, then 248 characters can be given single keys, while 1,752 (2000 - 248) are given two collation keys of the form [yyyy.00zz.00ww] [0000.00nn.0001].

The 248 can be chosen to be the higher frequency characters, while there would need to be eight distinct zz values to cover the remaining characters. These zz values must only be used with dual collation elements.

6.3.3 Leveraging Unicode Tables

Because all canonically decomposable characters are decomposed in Step 1.1, no collation elements need to be supplied for them. This includes a very large number of characters, not only a large number of Latin and Greek characters, but also the very large number of Hangul Syllables.

Because most compatibility decomposable characters in the default table can be algorithmically generated from the decomposition, no collation elements need to be stored for those decomposable characters: the collation elements can be generated on the fly with only a few exceptions entered in the table. The collation elements for the Han characters (unless tailored) are algorithmically derived; no collation elements need to be stored for them either. For more information, see *Section 7, Weight Derivation*.

This means that only a small fraction of the total number of Unicode characters need to have an explicit collation element. This can cut down the memory storage considerably.

6.3.4 Reducing the Repertoire

If characters are not fully supported by an implementation, then their code points can be treated as if they were unassigned. This allows them to be algorithmically constructed from code point values instead of including them in a table. This can significantly reduce the size of the required tables. See *Section 7.1, Derived Collation Elements* for more information.

6.3.5 Memory Table Size

Applying the above techniques, an implementation can thus safely pack all of the data for a collation element into a single 32-bit quantity: 16 for the primary, 8 for the secondary and 8 for the tertiary. Then applying techniques such as the Two-Stage table approach described in *"Multistage Tables"* in *Section 5.1, Transcoding to Other Standards* of [Unicode], the mapping table from characters to collation elements can both fast and small. For an example of how this can be done, see *Section 6.10, Flat File Example*.

6.4 Avoiding Zero Bytes

If the resulting sort key is to be a C-string, then zero bytes must be avoided. This can be done by:

- using the value 0101_{16} for the level separator instead of 0000.
- preprocessing the weight values to avoid zero bytes, such as remapping as follows:
  - x => 0101_{16} + (x / 255)*256 + (x % 255)
- Where the values are limited to 8-bit quantities (as discussed above), zero bytes are even more easily avoided by just using 01 as the level separator (where one is necessary), and mapping weights by
  - x => 01 + x.

6.5 Avoiding Normalization

Implementations that do not handle separate combining marks can map decomposable characters (such as "à") to single collation elements with different Level 2 weights for the different accents. ~~For more information, see *Section 7, Weight Derivation*.~~ However, this does require including the mappings for these characters in the collation table, which will increase the size substantially unless the collation elements for the Hangul Syllables are computed algorithmically.

6.5.1 Canonical Decompositions

Characters with canonical decompositions do not require mappings to collation elements, because ~~Step 1.1~~ S1.1 maps them to collation elements based upon their decompositions. However, they may be given mappings to collation elements anyway. The weights in those collation elements must be computed in such a way they will sort in the same relative location as if the characters were decomposed using Normalization Form D. By including these mappings, this allows an implementation handling a restricted repertoire of supported characters to compare strings correctly without performing the normalization in ~~Step 1.1~~ S1.1 of the algorithm.

A combining character sequence is called *impeding* if it contains any conjoining Jamo, or if it contains an L1-ignorable combining mark and there is some character that canonically decomposes to a sequence containing the same base character. For example, the sequence <a, cedilla> is an impediment, because *cedilla* is an L1-ignorable character, and there is some character (for example, *a-grave*) that decomposes to a sequence containing the same base letter *a*. Note that although strings in Normalization Form C generally do not contain impeding sequences, there is nothing prohibiting them from containing them.

> **Note:** Conformant implementations that do not support impeding character sequences as part of their repertoire can avoid performing Normalization Form D processing as part of collation.

6.5.2 Compatibility Decompositions

~~As remarked above,~~ Most characters with compatibility decompositions can have collation elements computed at runtime to save space, duplicating the work that was done to compute the Default Unicode Collation Element Table. This can be an important savings in memory space. The process works as follows.

1. Derive the compatibility decomposition. For example,

```
2475 PARENTHESIZED DIGIT TWO => 0028, 0032, 0029
```

2. Look up the collation element for each character in the decomposition. For example,

```
0028 [*023D.0020.0002] % LEFT PARENTHESIS
0032 [.06C8.0020.0002] % DIGIT TWO
0029 [*023E.0020.0002] % RIGHT PARENTHESIS
```

3. Set the first two L3 values to be lookup (L3), where the lookup function uses the table in *Section 7.2, Tertiary Weight Table*. Set the remaining L3 values to MAX (which in the default table is 001F). For example,

```
0028 [*023D.0020.0004] % LEFT PARENTHESIS
0032 [.06C8.0020.0004] % DIGIT TWO
0029 [*023E.0020.001F] % RIGHT PARENTHESIS
```

4. Concatenate the result to produce the sequence of collation elements that the character maps to. For example,

```
2475 [*023D.0020.0004] [.06C8.0020.0004] [*023E.0020.001F]
```

Some characters cannot be computed in this way. They must be filtered out of the default table and given specific values. For example,

```
017F [.085D.00FD.0004.017F] % LATIN SMALL LETTER LONG S; COMPAT
```

6.6 Case Comparisons

In some languages, it is common to sort lowercase before uppercase; in other languages this is reversed. Often this is more dependent on the individual concerned, and is not standard across a single language. It is strongly recommended that implementations provide parameterization that allow uppercase to be sorted before lowercase, and provide information as to the standard (if any) for particular countries. This can easily be done to the Default Unicode Collation Element Table before tailoring by remapping the L3 weights (see *Section 7, Weight Derivation*). It can be done after tailoring by finding the case pairs and swapping the collation elements.

6.7 Incremental Comparison

Implementations do not actually have to produce full sort keys. Collation elements can be incrementally generated as needed from two strings, and compared with an algorithm that produces the same results as sort keys would have. The choice of which algorithm to use depends on the number of comparisons between the same strings.

- Generally incremental comparison is *more* efficient than producing full sort keys if strings are only to be compared once and if they are generally dissimilar, because differences are caught in the first few characters without having to process the entire string.
- Generally incremental comparison is *less* efficient than producing full sort keys if items are to be compared multiple times.

However, it is very tricky to produce an incremental comparison that produces correct results. For example, some implementations have not even been transitive! Be sure to test any code for incremental comparison thoroughly.

6.8 Catching Mismatches

Sort keys from two different tailored collations cannot be compared, because the weights may end up being rearranged arbitrarily. To catch this case, implementations can produce a hash value from the collation data, and prepend it to the sort key. Except in extremely rare circumstances, this will distinguish the sort keys. The implementation then has the opportunity to signal an error.

6.9 Handling Collation Graphemes

A collation ordering determines a *collation grapheme cluster* (also known as a collation grapheme or collation character), which is a sequence of characters that is treated as a primary unit by the ordering. For example, *ch* is a collation grapheme for a traditional Spanish ordering. These are generally contractions, but may include additional ignorable characters. To determine the boundaries for a collation grapheme starting at a given position, use the following process:

1. Set `oldPosition` to be equal to `position`.
2. If `position` is at the end of the string, return it.
3. Fetch the next collation element(s) mapped to by the character(s) at `position`.
4. If the collation element(s) contain a non-ignorable and `position` is not equal to `oldPosition`, return `position`.
5. Otherwise set `position` to be the end of the characters mapped.
6. Loop back to step 2.

For information on the use of collation graphemes, see [UTS18].

6.10 Flat File Example

The following is a sample flat-file binary layout and sample code for collation data. It is included only for illustration. The table is used to generate collation elements from characters, either going forwards or backwards, and detect the start of a contraction. The backwards generation is for searching backwards or Boyer-Moore-style searching; the contraction detection is for random access.

In the file representation, ints are 32 bit values, shorts are 16, bytes are 8 bits. Negatives are two's-complement. For alignment, the ends of all arrays are padded out to multiples of 32 bits. The signature determines endianness. The locale uses an ASCII representation for the Java locale: a 2 byte ISO language code, optionally followed by '_' and 2 byte ISO country code, followed optionally by a series of variant tags separated by '_'; any unused bytes are zero.

Flat File Format

<table>
<tr><th>Data</th><th colspan="2">Comment</th></tr>
<tr><td>int signature;</td><td colspan="2">Constant <code>0x636F6C74</code>, used also for big-endian detection</td></tr>
<tr><td>int tableVersion;</td><td colspan="2">Version of the table format</td></tr>
<tr><td>int dataVersion;</td><td colspan="2">Version of the table data</td></tr>
<tr><td>byte[32] locale;</td><td colspan="2">Target locale (if any)</td></tr>
<tr><td>int flags;</td><td colspan="2">Bit01 = 1 if French secondary. Other values are reserved.</td></tr>
<tr><td>int limitVariable;</td><td colspan="2">Every CE below this value that has a non-zero primary is variable. Because variables are not interleaved, this does not need to be stored on a per-character basis.</td></tr>
<tr><td>int maxCharsPerCE;</td><td colspan="2">Maximum number of characters that are part of a contraction</td></tr>
<tr><td>int maxCEsPerChar;</td><td colspan="2">Maximum number of collation elements that are generated by an expansion</td></tr>
<tr><td>int indexOffset;</td><td colspan="2">Offset to index table</td></tr>
<tr><td>int collationElementsOffset;</td><td colspan="2">Offset to main data table</td></tr>
<tr><td>int expansionsOffset;</td><td colspan="2">Offset to expansion table</td></tr>
<tr><td>int contractionMatchOffset;</td><td colspan="2">Offset to contraction match table</td></tr>
<tr><td>int contractionResultOffset;</td><td colspan="2">Offset to contraction values table</td></tr>
<tr><td>int nonInitialsOffset;</td><td colspan="2">Offset to non-initials table. These are used for random access.</td></tr>
<tr><td>int[10] reserved;</td><td colspan="2">Reserved</td></tr>
<tr><td>int indexLength;</td><td colspan="2">Length of following table</td></tr>
<tr><td>int[] index;</td><td colspan="2">Index for high-byte (trie) table. Contains offsets into Collation Elements. Data is accessed by:
<code>ce = collationElements[index[char>>8]+char&0xFF]</code></td></tr>
<tr><td>int collationElementsLength;</td><td colspan="2">Length of following table</td></tr>
<tr><td>int[] collationElements;</td><td colspan="2">Each element is either a real collation element, an expansionsOffset, or an contractionsOffset. See below for more information.</td></tr>
<tr><td>int expansionsLength;</td><td colspan="2">Length of following table</td></tr>
<tr><td>int[] expansions;</td><td colspan="2">The expansionOffsets in the collationElements table point into sublists in this table. Each list is terminated by FFFFFFFF.</td></tr>
<tr><td>int contractionMatchesLength;</td><td colspan="2">Length of following table</td></tr>
<tr><td rowspan="2">short[] contractionMatches;</td><td colspan="2">The contractionOffsets in the collationElements table point into sublists in this table. Each sublist is of the following format:</td></tr>
<tr><td>short backwardsOffset;</td><td>When processing backwards, offset to true contractions table.</td></tr>
</table>

	short length;	Number of chars in list to search
	short[] charsToMatch;	Characters in sorted order.
int contractionCEsLength;	Length of following table	
int[] contractionCEs;	List of CEs. Each corresponds to a position in the contractionChars table. The one corresponding to the length in a sublist is the *bail-out;* this specifies what to do if a match is not found.	
int nonInitialsLength;	Length of following table	
short[] nonInitials;	List of characters (in sorted order) that can be non-initials in contractions. That is, if "ch" is a contraction, then "h" is in this list. If "abcd" is a contraction, then "b", "c", and "d" are in the list.	

6.10.1 Collation Element Format

- 'real' collationElement
  - 16 bits primary (FFE0..FFFF not allowed)
  - 10 bits secondary
  - 6 bits tertiary
- expansionsOffset
  - 12 bits = FFF
  - 20 bits = offset (allows for 1,048,576 items)
- contractionsOffset
  - 12 bits = FFE
  - 20 bits = offset (allows for 1,048,576 items)

An alternative structure would have the offsets be byte offsets from the start of the table, instead of indexes into the arrays. That would limit the size of the table, but use fewer machine instructions.

6.10.2 Sample Code

The following is a pseudo code using this table for the required operations. Although using Java syntax in general, the code example uses arrays so as to be more familiar to users of C and C++. *The code is presented for illustration only; it is not a complete statement of the algorithm.*

```
char[] input;   // input buffer (i)
int inputPos;   // position in input buffer (io)
int[] output;   // output buffer (o)
int outputPos;  // position in output buffer (io)
boolean forwards;   // 0 for forwards, 1 for backwards (i)

/**
* Reads characters from input, writes collation elements in output
```

```
*/
void getCollationElements() {
    char c = input[inputPos++];
    int ce = collationElements[index[c>>8] + c&0xFF];
    processCE(ce);
}

/**
* Normally just returns ce. However, special forms indicate that
* the ce is actually an expansion, or that we have to search
* to see if the character was part of a contraction.
* Expansions use
*/
void processCE(int ce) {
    if (ce < 0xFFF00000) {
        output[outputPos++] = ce;
    } else if (ce >= 0xFFE00000) {
        copyExpansions(ce & 0x7FFFFF);
    } else {
        searchContractions(ce & 0x7FFFFF);
    }
}

/**
* Search through a contraction sublist to see if there is a match.
* Because the list is sorted, we can exit if our value is too high.<p>
* Because we have a length, we could implement this as a
* binary search, although we do not right now.<p>
* If we do find a match, we need to recurse. That's how "abc" would
* be handled.<p>
* If we fail, we return the non-matching case. That can be an expansion
* itself (it would never be a contraction).
*/
void searchContractions(int offset) {
    if (forwards) inputPos++;
    else offset += input[inputPos++];
    short goal = (short)input[inputPos++];
    int limit = offset + contractionMatches[offset];
    for (int i = offset; i < limit; ++i) {
        short cc = contractionMatches[i];
        if (cc > goal) { // definitely failed
            processCE(contractionCEs[offset]);
            break;
        } else if (cc == goal) { // found match
            processCE(contractionCEs[i]);
            break;
        }
    }
}

/**
* Copy the expansion collation elements up to the terminator.
* Do not use 00000000 as a terminator, because that may be a valid CE.
* These elements do not recurse.
*/
void copyExpansions (int offset) {
    int ce = expansions[offset++];
    while (ce != 0xFFFFFFFF) {
        output[outputPos++] = ce;
        ce = expansions[offset++];
    }
}

/**
* For random access, gets the start of a collation element.
* Any non-initial characters are in a sorted list, so
* we just check that list.<p>
* Because we have a length, we could implement this as a
* binary search, although we do not right now.
*/
int getCollationElementStart(char[] buffer, int offset) {
    int i;
    main:
    for (i = offset; i > 0; --i) {
        char c = buffer[i];
        for (int j = 0; j < nonInitialsLength; ++j) {
            char n = nonInitials[j];
            if (c == n) continue main;
            if (c > n) break main;
        }
        break;
    }
    return i;
}
```

7 Weight Derivation

This section describes the generation of the Unicode Default Unicode Collation Element Table, and the assignment of weights to code points that are not explicitly mentioned in a Collation Element Table. This uses information from the Unicode Character Database [UAX44].

7.1 Derived Collation Elements

CJK Ideographs and Hangul Syllables are not explicitly mentioned in the default table. CJK ideographs are mapped to collation elements that are derived from their Unicode code point value as described in *Section 7.1.3, Implicit Weights*.

The collation algorithm requires that Hangul Syllables be decomposed. However, if the table is tailored so that the primary weights for Hangul Jamo (and all related characters) are adjusted, then the Hangul Syllables can be left as single code points and treated in the same way as CJK ideographs. That will provide a collation which is approximately the same as UCA, and may be sufficient in environments where individual jamo are not expected.

The adjustment is to move each initial jamo (and related characters) to have a primary weight corresponding to the first syllables starting with that jamo, and make all non-initial jamo (and related characters) be ignorable at a primary level.

7.1.1 Illegal Code Points

Certain code points are illegal in a data stream. These include noncharacters (code points with the Noncharacter_Code_Point property in the Unicode Character Database [UAX44]), unpaired surrogates (code points with the General_Category property Cs), and out-of-range values (< 0 or > 10FFFF). Implementations may also choose to treat these as error conditions and respond appropriately, such as by throwing an exception.

If they are not treated as an error condition, they must be mapped to [.0000.0000.0000.], and thus ignored.

7.1.2 Legal Code Points

Any other legal code point that is not explicitly mentioned in the table is mapped a sequence of two collation elements as described in *Section 7.1.3, Implicit Weights*.

7.1.3 Implicit Weights

A character is mapped to an implicit weight in the following way. The result of this process consists of collation elements that are sorted in code point order, that do not collide with any explicit values in the table, and that can be placed anywhere (for example, at BASE) with respect to the explicit collation element mappings (by default, they go after all explicit collation elements).

To derive the collation elements, the value of the code point is used to calculate two numbers, by bit shifting and bit masking. The bit operations are chosen so that the resultant numbers have the desired ranges for constructing implicit weights. The first number is calculated by taking the code point expressed as a 32-bit binary integer CP and bit shifting it right by 15 bits. Because code points range from U+0000 to U+10FFFF, the result will be a number in the range 0 to 21_{16} (= 33_{10}). This number is then added to the special value BASE.

```
AAAA = BASE + (CP >> 15);
```

Now mask off the bottom 15 bits of CP. OR a 1 into bit 15, so that the resultant value is non-zero.

```
BBBB = (CP & 0x7FFF) | 0x8000;
```

AAAA and BBBB are interpreted as unsigned 16-bit integers. The implicit weight mapping given to the code point is then constructed as:

```
[.AAAA.0020.0002.][.BBBB.0000.0000.]
```

If a fourth or higher weights are used, then the same pattern is used: they are set to a non-zero value, and so on in the first collation element and zero in the second. (Because all distinct code points have different **AAAA/BBBB** combination, the exact non-zero value does not matter.)

The value for BASE depends on the type of character. The first BASE value is for the core Han Unified Ideographs. The second BASE value is for all other Unified Han ideographs. In both of these cases, compatibility decomposibles are excluded, since they are otherwise handled in the UCA. Unassigned codepoints are also excluded from these first two BASE values. The final BASE value is for all other code points, including unassigned code points.

Values for Base

Base	Applicable Ranges
FB40	Unified_Ideograph=True **AND** ((Block=CJK_Unified_Ideograph) OR (Block=CJK_Compatibility_Ideographs))

	In regex notation: [\p{unified_ideograph}&[\p{Block=CJK_Unified_Ideographs}\p{Block=CJK_Compatibility_Ideographs}]]
FB80	Unified_Ideograph=True **AND NOT** ((Block=CJK_Unified_Ideograph) OR (Block=CJK_Compatibility_Ideographs)) In regex notation: [\p{unified ideograph}-[\p{Block=CJK_Unified_Ideographs}\p{Block=CJK_Compatibility_Ideographs}]]
FBC0	Any other code point

These results make AAAA (in each case) larger than any explicit primary weight; thus the implicit weights will not collide with explicit weights. It is not generally necessary to tailor these values to be within the range of explicit weights. However if this is done, the explicit primary weights must be shifted so that none are between each of the BASE values and BASE + 34.

7.1.4 Trailing Weights

This section on trailing weights needs to be generalized and rewritten, so that weighting of final jamos is just one example.

The range of primary weights from FC00 to FFFF is available for use as trailing weights especially for the case of Hangul Syllables. These syllables can be of the form LL*VV*T*: that is, one or more Lead jamo, followed by one or more Vowel jamo, followed optional by any number of Trail jamos. For more information, see *Section 3.12, Conjoining Jamo Behavior* in [Unicode].

Trailing weights are for characters that are given primary weights, but grouped as a unit together with a previous character, such as U+1160 HANGUL JUNGSEONG FILLER through U+11F9 HANGUL JONGSEONG YEORINHIEUH. By tailoring these characters in this range, the units are ordered independently of subsequent characters with higher weights. Otherwise problems may occur, such as in the following example.

Case 1

1	{G}{A}
2	{G}{A}{K}

Case 2

2	{G}{A}{K}事
1	{G}{A}事

In this example, the symbols {G}, {A}, and {K} represent letters in a script where syllables (or other sequences of characters) are sorted as units. By proper choice of weights for the individual letters, the syllables can be ordered correctly. But the weights of the following letters may cause syllables of different lengths to change order. Thus {G}{A}{K} comes after GA in Case 1. But in Case 2, it comes *before*. That is, the order of these two syllables would be reversed when each is followed by a CJK character: in this case, U+4E8B (事).

7.1.5 Hangul ~~Trailing Weights~~ Collation

Move this section to new location, as it is really about Hangul collation, and not about trailing weights per se.

The Hangul script is in a rather unique position, because of the large number of the precomposed characters, and because those precomposed characters are the normal (NFC) form of interchanged text. For Hangul syllables to sort correctly, either the UCA data must be tailored or the UCA algorithm (and data) must be tailored. The following are possible solutions:

Data Method

1. Tailor the Vs and Ts to be Trailing Weights, with the ordering T < V
2. Tailor each sequence of multiple L's that occurs in the repertoire as a contraction, with an independent primary weight after any prefix's weight
    - This means that if L_1 has a primary weight of 555, and L_2 has 559, then L_1L_2 would have to be given a weight from 556 to 558.

Terminator Method

1. Add an internal terminator primary weight (Ⓣ).
2. Tailor all Jamo so that Ⓣ < T < V < L
3. Algorithmically add the terminator primary weight (Ⓣ) to the end of every standard Hangul syllable.
    - This is done by adding the terminator between any pairs of characters that are not kept together according to the rules of *Section 3.12, Conjoining Jamo Behavior* of [Unicode]

Interleaving Method

1. Generate a modified weight table:
    a. Assign a weight to each precomposed Hangul Syllable character, with a 1-weight gap between each one. (see *Section 6.2, Large Weight Values*)
    b. Give each Jamo a 1-byte internal weight. Also add an internal terminator 1-byte weight (Ⓣ). These are assigned so that all Ⓣ < T < V < L.
        - These weights are separate from the default weights, and are just used internally.
2. When any string of Jamo and/or Hangul Syllables is encountered, break it into syllables according to the rules of *Section 3.12, Conjoining Jamo Behavior* of [Unicode]. Process each syllable separately:
    a. If a syllable is canonically equivalent to one of the precomposed Hangul Syllables, then just assign the weight as above
    b. If not, then find the greatest syllable that it is greater than; call that the base syllable. Generate a weight sequence corresponding to the following gap weight, followed by all the Jamo weight bytes, followed by the terminator

byte.

Each of these methods can correctly represent the ordering of all modern and ancient Hangul Syllables, but there are implementation trade-offs between them. These trade-offs can have a significant impact on the acceptability of the implementation, because substantially longer sort keys will cause significant performance degradations and database index bloat.

> **Note:** If the repertoire of supported Hangul syllables is limited to modern syllables (those of the form LV or LVT), then all of these become simpler.

The Data method provides for the following order of weights, where the X_b are all the scripts sorted before Hangul, and the X_a are all those sorted after.

X_b	L	X_a	T	V

This ordering gives the right results among the following:

Chars	Weights				Comments
$L_1V_1X_a$	W_{L1}	W_{V1}	W_{Xa}		
L_1V_1L...	W_{L1}	W_{V1}	W_{Ln}	...	
$L_1V_1X_b$	W_{L1}	W_{V1}	W_{Xb}		
$L_1V_1T_1$	W_{L1}	W_{V1}	W_{T1}		Works because W_T > all W_X and W_L
$L_1V_1V_2$	W_{L1}	W_{V1}	W_{V2}		Works because W_V > all W_T
$L_1L_2V_1$	W_{L1L2}	W_{V1}			Works *if* L_1L_2 is a contraction.

The disadvantages of the Data method are that the weights for T and V are separated from those of L, which can cause problems for sort-key compression, and that a combination of LL that is outside the contraction table will not sort properly.

The Terminator method would assign the following weights:

Ⓣ	X_b	T	V	L	X_a

This ordering gives the right results among the following:

Chars	Weights					Comments
$L_1V_1X_a$	W_{L1}	W_{V1}	Ⓣ	W_{Xa}		
$L_1V_1L_n$...	W_{L1}	W_{V1}	Ⓣ	W_{Ln}	...	
$L_1V_1X_b$	W_{L1}	W_{V1}	Ⓣ	W_{Xb}		
$L_1V_1T_1$	W_{L1}	W_{V1}	W_{T1}	Ⓣ		Works because W_T > all W_X and Ⓣ
$L_1V_1V_2$	W_{L1}	W_{V1}	W_{V2}	Ⓣ		Works because W_V > all W_T
$L_1L_2V_1$	W_{L1}	W_{L2}	W_{V1}	Ⓣ		Works because W_L > all W_V

The disadvantages of the Terminator method are that an extra weight is added to all

Hangul syllables, increasing the length of sort keys by roughly 40%, and the fact that the terminator weight is non-contiguous can disable sort-key compression.

The Interleaving method provides for the following assignment of weights. W_n represents the weight of a Hangul Syllable, and $W_{n'}$ is the weight of the gap right after it. The L, V, T weights will only occur after a W, and thus can be considered part of an entire weight.

X_b	W	X_a

byte weights:

Ⓣ	T	V	L

This ordering gives the right results among the following:

Chars	Weights			Comments
$L_1V_1X_a$	W_n	X_a		
$L_1V_1L_n$...	W_n	W_k	...	The L_n will start another syllable
$L_1V_1X_b$	W_n	X_b		
$L_1V_1T_1$	W_m			Works because $W_m > W_n$
$L_1V_1V_2$	W_m'$L_1V_1V_2$Ⓣ			Works because W_m'>W_m
$L_1L_2V_1$	W_m'$L_1L_2V_1$Ⓣ			Works because the byte weight for L_2 > all V

The Interleaving method is somewhat more complex than the others, but produces the shortest sort keys for all of the precomposed Hangul Syllables, so for normal text it will have the shortest sort keys. If there were a large percentage of ancient Hangul Syllables, the sort keys would be longer than other methods.

> **Note:** The Unicode Consortium recognizes that one of these solutions should be implemented in the standard UCA algorithm and tables, but is attempting to work out a common approach to the problem with the ISO SC2 OWG-SORT group, which takes considerable time. In the meantime, one of these approaches can be used for correct ordering.

7.2 Tertiary Weight Table

Characters are given tertiary weights according to the following table. The Decomposition Type is from the Unicode Character Database [UAX44]. The Condition is either based on the General Category or on a specific list of characters. The weights are from MIN = 2 to MAX = $1F_{16}$, excluding 7, which is not used for historical reasons. The Samples show some minimal values that are distinguished by the different weights. All values are distinguished The samples have empty cells when there are no (visible) values showing a distinction.

Tertiary Weight Assignments

Decomposition Type	~~Condition~~Case or Kana Subtype	Weight	Samples					
NONE		0x0002	i	ب	)	mw	1⁄2	X
<wide>		0x0003	ｉ					
<compat>		0x0004	ⅰ					
<font>		0x0005	𝐢					
<circle>		0x0006	ⓘ					
!unused!		0x0007						
NONE	Uppercase	0x0008	I			MW		
<wide>	Uppercase	0x0009	Ｉ		）			
<compat>	Uppercase	0x000A	Ⅰ					
<font>	Uppercase	0x000B	ℑ					
<circle>	Uppercase	0x000C	Ⓘ					
<small>	small hiragana (3041, 3043, ...)	0x000D						ぁ
NONE	normal hiragana (3042, 3044, ...)	0x000E						あ
<small>	small katakana (30A1, 30A3, ...)	0x000F			﹚			ァ
<narrow>	small narrow katakana (FF67..FF6F)	0x0010						ｧ
NONE	normal katakana (30A2, 30A4, ...)	0x0011						ア
<narrow>	narrow katakana (FF71..FF9D), narrow hangul (FFA0..FFDF)	0x0012						ｱ
<circle>	circled katakana (32D0..32FE)	0x0013						㋐
<super>		0x0014			⁾			
<sub>		0x0015			₎			
<vertical>		0x0016			︶			
<initial>		0x0017		ﺑ				
<medial>		0x0018		ﺒ				
<final>		0x0019		ﺐ				
<isolated>		0x001A		ﺏ				
<noBreak>		0x001B						
<square>		0x001C				㎽		
<square>, <super>, <sub>	Uppercase	0x001D				㎿		
<fraction>		0x001E					½	
n/a	(MAX value)	0x001F						

8 Searching and Matching

Language-sensitive searching and matching are closely related to collation. Strings that compare as equal at some strength level are those that should be matched when doing language-sensitive matching. For example, at a primary strength, "ß" would match against "ss" according to the UCA, and "aa" would match "å" in a Danish tailoring of the UCA. The main difference from the collation comparison operation is that the ordering is not important. Thus for matching it doesn't matter that "å" would sort after "z" in a Danish tailoring: the only relevant information is that they don't match.

The basic operation is matching: determining whether string X matches string Y. Other operations are built on this:

- Y contains X when there is some substring of Y that matches X
- A search for a string X in a string Y succeeds if Y contains X.
- Y starts with X when some initial substring of Y matches X
- Y ends with X when some final substring of Y matches X

The collation settings determine the results of the matching operation (see *Section 5.1, Parametric Tailoring*). Thus users of searching and matching need to be able to modify parameters such as locale or comparison strength. For example, setting the strength to exclude differences at Level 3 has the effect of ignoring case and compatibility format distinctions between letters when matching. Excluding differences at Level 2 has the effect of also ignoring accentual distinctions when matching.

Conceptually, a string matches some target where a substring of the target has the same sort key. But there are a number of complications:

1. The lengths of matching strings may differ: "aa" and "å" would match in Danish.
2. Because of ignorables (at different levels), there are different possible positions where a string matches, depending on the attribute settings of the collation. For example, if hyphens are ignorable for a certain collation, then "abc" will match "abc", "abc-", "-abc-", and so on.
3. Suppose that the collator has contractions, and that a contraction spans the boundary of the match. Whether or not it is considered a match may depend on user settings, just as users are given a "Whole Words" option in searching. So in a language where "ch" is a contraction, "bac" would not match in "bach" (given the proper user setting).
4. Similarly, combining character sequences may need to be taken into account. Users may not want a search for "abc" to match in "...abç..." (with a cedilla on the c). However, this may also depend on language and user customization.
5. The above two conditions can be considered part of a general condition: "Whole Characters Only"; very similar to the common "Whole Words Only" checkbox that is included in most search dialog boxes. (For more information on grapheme clusters, see [UTS18].
6. If the matching does not check for "Whole Characters Only", then some other complications may occur. For example, suppose that P is "x^", and Q is "x ^¸". Because the cedilla and circumflex can be written in arbitrary order and still be

> equivalent, one would expect to find a match for P in Q. A canonically-equivalent matching process requires special processing at the boundaries to check for situations like this. (It does not require such special processing within the P or the substring of Q because collation is defined to observe canonical equivalence.)

The following are used to provide a clear definition of searching and matching that deal with the above complications:

DS1. Define *S[start,end]* to be the substring of S that includes the character after the offset *start* up to the character before offset *end*. For example, if S is "abcd", then S[1,3] is "bc". Thus S = S[0,length(S)].

DS1a. A boundary condition is a test imposed on an offset within a string. An example includes Whole Word Search, as defined in UAX #29.

The tailoring parameter *match-boundaries* specifies constraints on matching (see *Section 5.1, Parametric Tailoring*). The parameter *match-boundaries=whole-character* requires that the start and end of a match each be on a grapheme boundary. The value *match-boundaries=whole-character* further requires that the start and end of a match each be on a word boundary as well. For more information on the specification of these boundaries, see [UAX29].

By using grapheme-complete conditions, contractions and combining sequences are not interrupted. This also avoids the need to present visually discontiguous selections to the user (except for BIDI text).

Suppose there is a collation C, a pattern string P and a target string Q, and a boundary condition B. C has some particular set of attributes, such as a strength setting, and choice of variable weighting.

DS2. The pattern string P *has a match at Q[s,e] according to collation C* if C generates the same sort key for P as for Q[s,e], and the offsets *s* and *e* meet the boundary condition B. We also say P has a match in Q according to C.

DS3. The pattern string P has a *canonical* match at Q[s,e] according to collation C if there is some Q' that is canonically equivalent to Q[*s,e*], and P has a match in Q'.

> For example, suppose that P is "Å", and Q is "...Ḁ̊...". There would not be a match for P in Q, but there would be a canonical match, because P does have a match in "Ḁ̊", which is canonically equivalent to "Ḁ̊". However, it is not commonly necessary to use canonical matches, so this definition is only supplied for completeness.

Each of the following are qualifications of DS2 or DS3.

DS3a. The match is *grapheme-complete* if B requires that the offset be at a grapheme cluster boundary. Note that Whole Word Search as defined in [UAX29] is grapheme complete.

DS4. The match is *minimal* if there is no match at Q[*s+i,e-j*] for any *i* and *j such that i ≥ 0, j* ≥ 0, and *i* + *j* > 0. In such a case, we also say that P has a *minimal* match *at* Q[*s,e*].

DS4a. The match is *medial* when it is contained in the maximal match, contains the minimal match, and is extended beyond the minimal match whenever there is a successive binary match between the extra characters in pattern and target.

DS4b. The match is *maximal* if there is no match at Q[*s-i,e+j*] for any *i* and *j such that i* ≥ *0, j* ≥ 0, and *i* + *j* > 0. In such a case, we also say that P has a *maximal* match *at* Q[*s,e*].

As an example of the differences between these, consider the following case, where the collation strength is set to ignore punctuation and case, and **format** indicates the match.

Minimal, Medial, and Maximal Matches

	Text	Description
Pattern	*!abc!*	Notice that the *! and !* are ignored in matching.
Target Text	def$!Abc%$ghi	
Minimal Match	def$!**Abc**%$ghi	The minimal match is the tightest one, because $! and %$ are ignored in the target.
Medial Match	def$**!Abc**%$ghi	The medial one includes those characters that are binary equal.
Maximal Match	def**$!Abc%$**ghi	The maximal match is the loosest one, including the surrounding ignored characters.

By using minimal, maximal, or medial matches, the issue with ignorables is avoided. Medial matches tend to match user expectations the best.

When an additional condition is set on the match, the types (minimal, maximal, medial) are based on the matches *that meet that condition.* Consider the following:

Alternate End Points for Matches

	Value	Notes
Pattern~~:~~	abc	
Strength~~:~~	*primary*	thus ignoring combining marks, punctuation
Text~~:~~	abc ¸ −°d	two combining marks, cedilla and ring
Matches~~:~~	|abc| ¸ |−|°|d	four possible end points, indicated by |

~~Thus ~~If, for example, the condition is Whole Grapheme, then the matches are restricted to "abc¸|-°|d", thus discarding match positions that would not be on a grapheme cluster boundary. ~~Thus~~ In this case the minimal match would be "abc¸|-°d"

DS6. The *first forward match* for P in Q starting at *b* is the least offset *s* greater than or equal to *b* such that for some *e*, P matches within Q[s,e].

DS7. The *first backward match* for P in Q starting at *b* is the greatest offset *s* less than or equal to *b* such that for some *e*, P matches within Q[s,e].

In DS6 and DS7, matches can be minimal, medial, or maximal; the only requirement is that the combination in use in DS6 and DS7 be specified. Of course, a possible match can also be rejected on the basis of other conditions, such as being grapheme-complete or applying Whole Word Search, as described in [UAX29]).

The choice of medial or minimal matches for the "starts with" or "ends with" operations only affects the positioning information for the end of the match or start of the match, respectively.

Special Cases. Ideally, the UCA at a secondary level would be compatible with the standard Unicode case folding and removal of compatibility differences, especially for the purpose of matching. For the vast majority of characters, it is compatible, but there are the following exceptions:

1. The UCA maintains compatibility with the DIN standard for sorting German by having the German *sharp-s* (U+00DF (ß) LATIN SMALL LETTER SHARP S) sort as a secondary difference with "SS", instead of having ß and SS match at the secondary level.
2. Compatibility normalization (NFKC) folds stand-alone accents to a combination of space + combining accent. This was not the best approach, but for backwards compatibility cannot be changed in NFKC. UCA takes a better approach to weighting stand-alone accents, but as a result does not weight them exactly the same as their compatibility decompositions.
3. Case-Folding maps *iota-subscript* (U+0345 (ͅ) COMBINING GREEK YPOGEGRAMMENI) to an iota, due to the special behavior of iota-subscript, while the UCA treats *iota-subscript* as a regular combining mark (secondary ignorable).
4. When compared to their case and compatibility folded values, UCA compares the following as different at a secondary level, whereas other compatibility differences are at a tertiary level.
    - U+017F (ſ) LATIN SMALL LETTER LONG S (and precomposed characters containing it)
    - U+1D4C (ᵌ) MODIFIER LETTER SMALL TURNED OPEN E
    - U+2D6F (ⵯ) TIFINAGH MODIFIER LETTER LABIALIZATION MARK

In practice, most of these differences are not important for modern text, with one exception: the German ß. Implementations should consider tailoring ß to have a tertiary difference from SS, at least when collation tables are used for matching. Where full compatibility with case and compatibility folding are required, either the text can be preprocessed, or the UCA tables can be tailored to handle the outlying cases.

8.1 Collation Folding

Matching can be done by using the collation elements, directly, as discussed above. However, since matching doesn't use any of the ordering information, the same result can be achieved by a folding. That is, two strings would fold to the same string if and only if they would match according to the (tailored) collation. For example, a folding for a Danish collation would map both "Gård" and "gaard" to the same value. A folding for a primary-strength folding would map "Resume" and "résumé" to the same value. That folded value is typically a lowercase string, such as "resume".

A comparison between folded strings cannot be used for an ordering of strings, but it can be applied to searching and matching quite effectively. The data for the folding can be smaller, because the ordering information does not need to be included. The folded strings are typically much shorter than a sort key, and are human-readable, unlike the sort key. The processing necessary to produce the folding string can also be faster than that used to create the sort key.

The following is an example of the mappings used for such a folding using to the [CLDR] tailoring of UCA:

Parameters:

{locale=da_DK, strength=secondary, alternate=shifted}

Mapping:

...

ª	→	a	Map compatibility (tertiary) equivalents, such as full-width and superscript characters, to representative character(s)
ａ	→	a	
A	→	a	
Ａ	→	a	
ᵃ	→	a	

...

å	→	aa	Map contractions (a + ring above) to equivalent values
Å	→	aa	

...

Once the table of such mappings is generated, the folding process is a simple longest-first match-and-replace: a string to be folded is first converted to NFD, then at each point in the string, the longest match from the table is replaced by the corresponding result.

However, ignorable characters need special handling. Characters that are fully ignorable at a given strength level level normally map to the empty string. For example, at *strength=quaternary*, most controls and format characters map to the empty string; at *strength=primary*, most combining marks also map to the empty string. In some contexts, however, fully ignorable characters may have an effect on comparison, or characters that aren't ignorable at the given strength level are treated as ignorable.

1. Any discontiguous contractions need to be detected in the process of folding and handled according to Rule S2.1. For more information about discontiguous contractions, see *Section 3.3.2, Contractions.*
2. An ignorable character may interrupt what would otherwise be a contraction. For example, suppose that "ch" is a contraction sorting after "h", as in Slovak. In the absence of special tailoring, a CGJ or SHY between the "c" and the "h" prevents the contraction from being formed, and causes "c<CGJ>h" to not compare as equal to "ch". If the CGJ is simply folded away, they would incorrectly compare

as equal. See also *Section 5.3, Combining Grapheme Joiner*.

3. With the parameter values *alternate=shifted* or *alternate=blanked*, any (partially) ignorable characters after Variables have their weights reset to zero at levels 1 to 3, and may thus become fully ignorable. In that context, they would also be mapped to the empty string. For more information, see *Section 3.6.2, Variable Weighting*.

Appendix A: Deterministic Sorting

There is often a good deal of confusion about what is meant by the terms "stable" or "deterministic" when applied to sorting or comparison. This confusion in terms often leads people to make mistakes in their software architecture, or make choices of language-sensitive comparison options that have significant impact in terms of performance and footprint, and yet do not give the results that users expect.

A.1 Stable Sort

A stable sort is one where two records will retain their order when sorted according to a particular field, even when the two fields have the same contents. Thus those two records come out in the same relative order that they were in before sorting, although their positions relative to other records may change. Importantly, this is a property of the sort~~ing~~ algorithm, *not* the comparison mechanism.

Two examples of differing sort~~ing~~ algorithms are Quick sort and Merge sort. Quick sort is not stable while Merge sort is stable. (A Bubble sort, as typically implemented, is also stable.)

- For background on the names and characteristics of different sorting methods, see [SortAlg]
- For a definition of stable sorting, see [Unstable]

~~Suppose that you have~~ Assume the following records~~ and you sort on the Last_Name field only~~:

Original Records

Record	Last_Name	First_Name
1	Davis	John
2	Davis	Mark
3	Curtner	Fred

The results of a Merge sort on the Last_Name field only are:

Merge Sort Results

Record	Last_Name	First_Name
3	Curtner	Fred
1	Davis	John

2	Davis	Mark

The results of a Quicksort on the Last_Name field only are:

Quick sort Results

Record	Last_Name	First_Name
3	Curtner	Fred
2	Davis	Mark
1	Davis	John

As ~~you see from the above~~ is apparent, the Quicksort algorithm is not stable; ~~the~~ records 1 and 2 are not in the same order they were in before sorting.

A stable sort is often desir~~ed~~able; ~~—~~for one thing, it allows records to be successively sort~~ing~~ed according to different fields, and to retain the correct lexicographic order. Thus, with a stable sort, one could sort all the records by First_Name, and then sort ~~all the records~~ them again by Last_Name, giving the desired results: that all records would be ~~sorted~~ ordered by Last_Name, and in the case where the Last_Name values are the same, ~~sorted~~ be further subordered by First_Name.

A.2 Deterministic Sort

A *deterministic* sort is a very different beast. This is a sort algorithm that returns the same results each time. On the face of it, it would seem odd for any sort algorithm to *not* be deterministic, but there are examples of real-world sort algorithms that aren't.

The key concept is that these sort algorithms *are* deterministic when two records have unequal fields, but they may return different results at different times when two records have equal fields.

For example, a classic Quick sort algorithm works recursively on ranges of records. For any given range of records, it takes the first element as the *pivot element*. ~~We're not going to explain the details here, but~~ However, that algorithm performs badly with input data that happens to be already sorted (or mostly sorted). A randomized Quick sort, which picks a random element as the pivot, can on average be faster. The results it gets are not deterministic; from *exactly* the same input, different results can be output:

Enhanced Quick sort Results

Record	Last_Name	First_Name
3	Curtner	Fred
2	Davis	John
1	Davis	Mark

or

Record	Last_Name	First_Name
3	Curtner	Fred
1	Davis	Mark
2	Davis	John

As another example, multiprocessor sort~~ing~~ algorithms can be non-deterministic. The work of sorting different blocks of data are farmed out to different processors and then merged back together. The ordering of records with equal fields might be different according to when different processors finish different tasks.

Note that a deterministic sort is weaker than a stable sort. A stable sort is always deterministic, but not vice versa. And typically when people say they want a deterministic sort, they really mean that they want a stable sort.

A.3 Deterministic Comparison

A *deterministic comparison* is different than either of the above; it is a property of a comparison function, *not* a sort~~ing~~ algorithm. This is a comparison where strings that do not have identical binary contents (optionally, after some process of normalization) will compare as unequal. A deterministic comparison is sometimes called a *stable* (or *semi-stable*) *comparison*.

There are many people who confuse a deterministic comparison with a deterministic (or stable) sort, but once again, these are *very* different creatures. A comparison is used by a sort~~ing~~ algorithm to determine the relative ordering of two fields, such as strings. Using a deterministic comparison *cannot* cause a sort to be deterministic, nor to be stable. Whether a sort is deterministic or stable is a property of the sort algorithm, *not* the comparison function. If you look at the examples above, this is clear.

A.3.1 Forcing Deterministic Comparisons

One can produce a deterministic comparison function from a non-deterministic one, in the following way (in pseudo-code):

```
int new_compare (String a, String b) {
  int result = old_compare(a, b);
  if (result == 0) {
    result = binary_compare(a, b);
  }
  return result;
}
```

Programs typically also provide the facility to generate a *sort key*, which is a sequences of bytes generated from a string in alignment with a comparison function. Two sort keys will binary-compare in the same order as their original strings. To create a deterministic sort key that aligns with the above `new_compare`, the simplest means is to append a copy of the original string to the sort key. This will force the comparison to be deterministic.

```
byteSequence new_sort_key (String a) {
  return old_sort_key(a) + SEPARATOR + toByteSequence(a);
}
```

Because sort keys and comparisons must be aligned, a sort key generator is deterministic if and only if a comparison is.

A.3.2 Best Practice

~~However, ~~A deterministic comparison is generally not best practice. First, it has a certain performance cost in comparison, and a quite substantial impact on sort key size. (For example, ICU language-sensitive sort keys are generally about the size of the original string, so appending a copy of the original string generally *doubles* the size of the sort key.) A database using these sort keys can have substantially increased disk footprint and memory footprint, and consequently reduced performance.

More importantly, a deterministic comparison function does not actually achieve the effect people think it will have. Look at the sorted examples above. Whether a deterministic comparison is used or not, there will be ***no*** effect on the Quick sort example, since the two records in question have identical Last_Name fields. It does not make a non-deterministic sort into a deterministic one, nor does it make a non-stable sort into a stable one.

Thirdly, a deterministic comparison is often not what is wanted, when people look closely at the implications. Look at ~~our above~~ the example again, and suppose that this time the user is sorting first by last name, then by first name.

Original Records

Record	Last_Name	First_Name
1	Davis	John
2	Davis	Mark
3	Curtner	Fred

The desired results are the following, which should result whether the sort~~ing~~ algorithm is stable or not, since we are using both fields.

First then Last

Record	Last_Name	First_Name
3	Curtner	Fred
1	Davis	John
2	Davis	Mark

Now suppose that in record 2, the source for the data caused the last name to contain a format control character, such as a ZWJ (used to request ligatures on display). There is in this case no visible distinction in the forms, since the font doesn't have any ligatures for these ~~values~~ sequences of Latin letters. The ~~standard~~ default UCA collation ~~sequence~~ weighting causes ~~that character~~ the ZWJ to be—correctly—ignored in comparison, since it should only affect rendering. However, if that comparison is changed to be deterministic (by appending the binary values for the original string), then unexpected results will occur.

First then Last (Deterministic)

Record	Last_Name	First_Name
3	Curtner	Fred
2	Davis	Mark

1	Davis	John

A.3.3 Forcing Stable Sorts

Typically, what people really want when they say they want a deterministic comparison is actually a stable sort.

One *can* force a non-stable sort~~ing~~ algorithm to produce stable results by how one does the comparison. However, ~~it~~ this has literally nothing to do with making the comparison ~~be~~ deterministic or not. ~~Instead, it~~ Forcing stable results can be done by appending the *current record number* to the strings to be compared. (The implementation may not actually append the number; it may use some other mechanism, but the effect would be the same.)

If such a modified ~~sort~~ comparison is used, for example, ~~then~~ it forces Quick sort to get the same results as a Merge sort. And in that case, the irrelevant character ZWJ does not affect the outcome.~~, as illustrated below, and~~ The correct results occur, as illustrated below.

First then Last (Forced Stable Results)

Record	Last_Name	First_Name
3	Curtner	Fred
1	Davis	John
2	Davis	Mark

If anything, this then is what users want when they say they want a deterministic comparison. See also *Section 1.6, Interleaved Levels*.

A.4 Stable and Portable Comparison

There are a few other terms worth mentioning, simply because they are also subject to considerable confusion. Any or all of the following terms may be easily confused with the discussion above.

A *stable comparison* is one that does not change over successive software versions. That is, as one uses successive versions of an API, with the same "settings" (such as locale), one gets the same results.

A *stable sort key generator* is one that generates the same binary sequence over successive software versions.

> **Warning:** if the sort key generator is stable, then the associated comparison will perforce be. However, the reverse is not guaranteed. To take a trivial example, suppose the new version of the software always adds an 0xFF byte at the front of every sort key. The results of any comparison of any two *new* keys would be identical to the results of the comparison of any two corresponding *old* keys. But the bytes have changed, and the comparison of old and new keys would give different results. Thus one can have a stable comparison, yet an associated non-stable sort key generator.

A *portable comparison* is where corresponding APIs for comparison produce the same results across different platforms. That is, if one uses the same "settings" (such as locale), one gets the same results.

A *portable sort key generator* is where corresponding sort key APIs produce exactly the same sequence of bytes across different platforms.

> **Warning:** as above, a comparison may be portable without the associated sort key generator being portable.

Ideally, all products would have the same string comparison and sort key generation for, say Swedish, and thus be portable. For historical reasons, this is not the case. Even if the main letters sort the same, there will be differences in the handling of other letters, or of symbols, punctuation, and other characters. There are some libraries that offer portable comparison, such as [ICUCollator], but in general the results of comparison or sort key generation may vary significantly between different platforms.

In a closed system, or in simple scenarios, portability may not matter. Where someone has a given set of data to present to a user, and just wants the output to be reasonably appropriate for Swedish, ~~then~~ the exact order on the screen may not matter.

In other circumstances, ~~it~~ differences can lead to data corruption. For example, suppose that two implementations do a database SELECT for records between a pair of strings. If the collation is different in the least way, they can get different data results. Financial data might be different, for example, if a city is included in one SELECT on one platform and excluded from the same SELECT on another platform.

Acknowledgements

Thanks to Bernard Desgraupes, Richard Gillam, Kent Karlsson, York Karsunke, Michael Kay, Åke Persson, Roozbeh Pournader, Markus Scherer, Javier Sola, Otto Stolz, Ienup Sung, Yoshito Umaoka, Andrea Vine, and Vladimir Weinstein for their feedback on previous versions of this document, to Jianping Yang and Claire Ho for their contributions on matching, and to Cathy Wissink for her many contributions to the text.

References

[AllKeys]	The latest approved version of this file is: http://www.unicode.org/Public/UCA/latest/allkeys.txt The version at the time of this publication is: http://www.unicode.org/Public/UCA/6.0.0/allkeys.txt ~~*Note:*~~ ftp access is available, starting at: ftp://www.unicode.org/Public/UCA/
[CanStd]	CAN/CSA Z243.4.1

[CLDR] *Common Locale Data Repository*
http://unicode.org/cldr/

[FAQ] Unicode Frequently Asked Questions
http://www.unicode.org/faq/
For answers to common questions on technical issues.

[Feedback] Reporting Errors and Requesting Information Online
http://www.unicode.org/reporting.html

[Glossary] Unicode Glossary
http://www.unicode.org/glossary/
For explanations of terminology used in this and other documents.

[ICUCollator] ICU User Guide: Collation Introduction
~~http://icu.sourceforge.net/userguide/Collate_Intro.html~~
http://userguide.icu-project.org/collation

[ISO14651] International Organization for Standardization. *Information Technology--International String ordering and comparison--Method for comparing character strings and description of the common template tailorable ordering.* (ISO/IEC 14651:2001). *For availability see http://www.iso.org*

[JavaCollator] http://java.sun.com/j2se/1.4/docs/api/java/text/Collator.html,
http://java.sun.com/j2se/1.4/docs/api/java/text/RuleBasedCollator.html

[LDML] UTS #35: *Locale Data Markup Language (LDML)*
http://www.unicode.org/reports/tr35/

[Reports] Unicode Technical Reports
http://www.unicode.org/reports/
For information on the status and development process for technical reports, and for a list of technical reports.

[Sample] http://www.unicode.org/reports/tr10/Sample/

[Test] The latest approved versions of these files are:
http://www.unicode.org/Public/UCA/latest/CollationTest.html
http://www.unicode.org/Public/UCA/latest/CollationTest.zip
The versions at the time of this publication are:
http://www.unicode.org/Public/UCA/6.0.0/CollationTest.html
http://www.unicode.org/Public/UCA/6.0.0/CollationTest.zip
Note: ftp access is available, starting at:
ftp://www.unicode.org/Public/UCA/

Modifications

The following summarizes modifications from the previous revisions of this document.

Revision 21 [KW]

- Proposed update for Unicode 6.0.0.
- Updated text of Summary at top of document.
- Reorganized and renumbered several sections for better text flow.

- Provided anchors for tables, and updated table and caption formats to match current Technical Report style. Added captions for tables or figures that did not have them. Removed unneeded color backgrounds from tables.
- Updated several obsolete links.
- Significant editorial corrections throughout.

Revision 20

- Updated for Unicode 5.2.0.
- In *Section 7.1.3* *Implicit Weights*, clarified the calculation of implicit weights.
- Made it clear that the BASE value does not include unassigned code points.
- Clarified why some sample cells are empty in the first table.
- General: updated references to UAX/UTS's
- Removed reference to UTR#30
- Better aligned the options with the 3 values for variableChoice.
- Clarified the computation of the fourth level in *Section 3.2.1, File Format*. [KW]
- Changed bit layout in *Section 6.10.1* *Collation Element Format* for a real collation element, to account for the fact that the DUCET secondary values number more than 255, so no longer fit in 8 bits. [KW]
- Made small editorial clarifications regarding variable weighting in *Section 3.2.2, Variable Weighting*. [KW]
- Updated reference to SC22 WG20 to SC2 OWG-SORT in *Section 7.1.4.1*. [KW]
- Made a minor wording clarification in *Section 7.3* *Compatibility Decompositions*. [KW]
- Small editorial updates through for formatting consistency. [KW]
- Updated Modifications section to current conventions for handling proposed update drafts. [KW]

Revision 20 being a proposed update, only changes between revisions 20 and 18 are noted here.

Revision 18

- Updated for Unicode 5.1.0.
- Disallowed skipping 2.1.1 through 2.1.3 (*Section 4.2, Produce Array*).
- Clarified use of contractions in the DUCET in *Section 3.2, Default Unicode Collation Element Table* and *Section 3.1.1.2, Contractions*.
- Added information about the use of parameterization (*Section 5.1, Parametric Tailoring*) and a new conformance clause C6.
- In *Section 8, Searching and Matching*, added new introduction and explained special cases; clarified language in definitions.
- Added *Section 8.1, Collation Folding*.
- Fixed a number of reported typos.

Revision 17 being a proposed update, only changes between revisions 18 and 16 are noted here.

Revision 16

- Updated for Unicode 5.0.0.
- Replaced "combining mark" by "non-starter" where necessary.
- Updated reference to Unicode 5.0 with the ISBN number.
- Added UTN#9 text in informative appendix as *Appendix A: Deterministic_Sorting*.

Revision 15 being a proposed update, only changes between revisions 16 and 14 are noted here.

Revision 14

- Updated for Unicode 4.1.0.
- Expanded use of 0x1D in *Section 7.3.1, Tertiary Weight Table*.
- Removed DS5, added DS1a, DS2a, explanations of interactions with other conditions, such as Whole Word or Whole Grapheme.
- Added conformance clause C5 for searching and matching.
- Many minor edits.
- Removed S1.3, so that fully ignorable characters will interrupt contractions (that do not explicitly contain them).
- Added related *Section 3.1.6, Combining Grapheme Joiner*.
- Removed S1.2 for Thai, and a paragraph in 1.3.
- Added more detail about Hangul to *Section 7.1.4, Trailing Weights*, including a description of the Interleaving method.
- Fixed dangling reference to base standard in C4.
- Added definitions and clarifications to *Section 8, Searching and Matching*.
- Added more information on user expectations to *Section 1, Introduction*.

Data tables for 4.1.0 contain the following changes:

1. The additions of weights for all the new Unicode 4.1.0 characters.
2. The change of weights for characters Æ, Ǽ, Ǣ; Đ, Ð; Ħ; Ł, Ŀ; and Ø, Ǿ (and their lowercase and accented forms) to have secondary (accent) differences from AE; D; H; L; and O, respectively. This is to provide a much better default for languages in which those characters are not tailored. See also the section on user expectations.
3. Change in weights for U+0600 ARABIC NUMBER SIGN and U+2062 INVISIBLE TIMES and like characters (U+0600..U+0603, U+06DD, U+2061..U+2063) to be not completely ignorable, because their effect on the interpretation of the text can be substantial.
4. The addition of about 150 contractions for Thai. This is synchronized with the removal of S1.2. The result produces the same results for well-formed Thai data, while substantially reducing the complexity of implementations in searching and matching. Other changes for Thai include:
    a. After U+0E44 ไ THAI CHARACTER SARA AI MAIMALAI
       Insertion of the character U+0E45 ๅ THAI CHARACTER LAKKHANGYAO
    b. Before U+0E47 ็ THAI CHARACTER MAITAIKHU

   Insertion of the character U+0E4E ๎ THAI CHARACTER YAMAKKAN

   c. After U+0E4B ๋ THAI CHARACTER MAI CHATTAWA
   Insertion of the character U+0E4C ์ THAI CHARACTER THANTHAKHAT
   Then the character U+0E4D ํ THAI CHARACTER NIKHAHIT

5. Changed the ordering of U+03FA GREEK CAPITAL LETTER SAN and U+03FB GREEK SMALL LETTER SAN.

Revisions 12 and 13 being proposed updates, only changes between revisions 14 and 11 are noted here.

Revision 11

- Changed the version to synchronize with versions of the Unicode Standard, so that the repertoire of characters is the same. This affects the header and C4. This revision is synchronized with Unicode 4.0.0.
- Location of data files changed to http://www.unicode.org/Public/UCA/
- Added new Introduction. This covers concepts in *Section 5.17*, "Sorting and Searching", in *The Unicode Standard, Version 3.0*, but is completely reworked. The Scope section has been recast and is now at the end of the introduction.
- In *Section 6.9, Tailoring Example: Java*, added informative reference to LDML; moved informative reference to ICU.
- Added explanation of different ways that the Hangul problem can be solved in *Section 7.1.4, Trailing Weights*.
- Copied sentence from Scope up to Summary, for more visibility.

Revision 10 being a proposed update, only changes between revisions 11 and 9 are noted here.

Revision 9

- Added C4.
- Added more conditions in *Section 3.3, Well-Formed Collation Element Tables*.
- Added S1.3.
- Added treatment of ignorables after variables in *Section 3.2.2, Variable Weighting*.
- Added *Section 3.4, Stability*.
- Modified and reorganized *Section 7, Weight Derivation*. In particular, CJK characters and unassigned characters are given different weights. Added MAX to *Section 7.3*.
- Added references.
- Minor editing.
- Clarified noncharacter code points in *Section 7.1.1, Illegal code points*.
- Modified S1.2 and *Section 3.1.3, Rearrangement* to use the Logical_Order_Exception property, and removed *rearrange* from the file syntax in *Section 3.2.1, File Format*, and from *Section 5, Tailoring*.
- Incorporated Cathy Wissink's notes on linguistic applicability.
- Updated links for [Test].