Technical Reports

Proposed Update
Unicode Technical Standard #10

UNICODE COLLATION ALGORITHM

Version	9
Authors	Mark Davis (mark.davis@us.ibm.com), Ken Whistler (ken@unicode.org)
Date	2001-04-24
This Version	http://www.unicode.org/reports/tr10/tr10-9.html
Previous Version	http://www.unicode.org/reports/tr10/tr10-8.html
Latest Version	http://www.unicode.org/reports/tr10/
Base Unicode Version	Unicode 3.1 with Corrigendum 3

Summary

This report provides the specification of the Unicode Collation Algorithm, which provides a specification for how to compare two Unicode strings while remaining conformant to the requirements of The Unicode Standard, Version 3.0.

Status

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 This is a draft document which may be updated, replaced, or superseded by other documents at any time.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

Feedback is requested, particularly on the new section, §8 Searching and Matching, and the conformance test [Test].

A list of current Unicode Technical Reports is found on [Reports]. For more information about versions of the Unicode Standard, see [Versions]. Please mail corrigenda and other comments to the author(s).

Contents

1 Scope

The Unicode Collation Algorithm (UCA) provides a specification for how to compare two Unicode strings while remaining conformant to the requirements of *The Unicode Standard, Version 3.0* . The UCA also supplies the Default Unicode Collation Element Table as the data specifying the default collation order.

Readers should be familiar with *Section 5.17 Sorting and Searching* of the Unicode Standard before proceeding with the rest of this document, since that section introduces the basic concepts involved in collation.

1.1 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*) for creating 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, both in terms of performance and in terms of memory requirements.

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

> **Note**: The Default Unicode Collation Element Table is currently limited to the Unicode 3.0 repertoire. However, the algorithm is well defined over *all* Unicode code points. See §7.1.2 Legal code points.
>
> **Note**: When the table is updated to the Unicode 3.1 repertoire the ordering of a few Unicode 3.0 characters will change for consistency with UAX #15: Unicode Normalization Forms.

1.2 Non-Goals

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

1. *reversibility:* from a Collation Element you are not guaranteed that you can 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:* for example, removing articles such as *a* and *the* during bibliographic sorting is not provided.
5. *Stability of binary sort key values between versions:* (However, this will be addressed in a future version of this document.)

1.3 Summary

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 proposed International String Ordering standard [SoStd].

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 for tie-breaking (semi-stability) may be used for tie-breaking between strings not otherwise distinguished.

This design allows implementations to produce culturally acceptable collation, while putting the least burden on implementations in terms of memory requirements and performance. In particular, Collation Element Tables only require storage of 32 bits of collation data per significant character.

However, implementations of the Unicode Collation Algorithm are not limited to supporting only 3 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 3 levels *and* must have those specials collate in non-Unicode order (as, 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.

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. 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 so long as any two strings compared by the implementation are ordered the same as they would be by the algorithm. They are also free to use a different format for the data in the Collation Element Table. The sort key is also a *logical* intermediate object: so long as an implementation produces the same results in comparison of strings, the sort keys can differ in format from what is specified here. (See §6 Implementation Notes.)

The 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 Unicode strings as those produced by §4 Main Algorithm.*

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

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 backward levels, variable weighting, semi-stability or rearrangement 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, since this is required for French.

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

The precise values of the collation elements for the characters may change over time as new characters are added to the Unicode Standard. Because canonical equivalence depends on the version of the Unicode Standard, that version must also be 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 16-bit weights. (All code points not explicitly mentioned in the mapping are given an implicit weight: see §7 Weight Derivation).

Note: Implementations can produce the same result without using 16-bit weights — see §6 Implementation Notes.

The first weight is called the *Level 1* weight (or *primary* weight), the second is called the *Level 2* weight (*secondary* weight), the third is called the *Level 3* weight (*tertiary* weight), the fourth is called the *Level 4* weight (*quarternary* weight), and so on. For a collation element X, these can be abbreviated as X_1, X_2, X_3, X_4, etc. Given two collation elements X and Y, we will use 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*	$X =_1 Y$ and $X_2 = Y_2$
$X =_3 Y$	*X is tertiary equal to Y*	$X =_2 Y$ and $X_3 = Y_3$
$X =_4 Y$	*X is quarternary equal to Y*	$X =_3 Y$ and $X_4 = Y_4$

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 quarternary 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 these. 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 $Y \leq_n X$

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

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

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 Table, we write $A \equiv B$. If they are bit-for-bit identical, we write 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 ignorable (or primary ignorable)* is a collation element that is ignorable at Level 1, but not at Level 2;
- a *Level 2 ignorable (or secondary ignorable)* is ignorable at Levels 1 and 2, but not Level 3;
- a *Level 3 ignorable (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.

The following are sample collation elements that are used in the examples illustrating the algorithm. Unless otherwise noted, all weights are in hexadecimal format.

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

Linguistic requirements of collation are covered in more detail in *The Unicode Standard, Version 3.0, Section 5.17 Sorting and Searching*.

3.1.1 Multiple Mappings

The mapping from characters to collation elements may not be a simple mapping from one character to one collation element: in general, it may map from one to many, from many to one, or from many to many. For example:

3.1.1.1 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*.

3.1.1.2 Contractions

Similarly, where *ch* is treated as a single letter as 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*. The 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.

3.1.1.3 Other Multiple Mappings

Certain characters may both expand and contract: see *Section 5.17 Sorting and Searching.*

3.1.2 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.

3.1.3 Rearrangement

Certain characters are not coded in logical order. In Unicode 3.0, these are the Thai vowels `0E40` through `0E44` and the Lao vowels `0EC0` through `0EC4` (this list may be extended in the future, and so is included in the Default Unicode Collation Element Table). For collation, they are rearranged by swapping with the following character before further processing, since logically they belong afterwards. For example, here is a string processed by rearrangement:

input string:	`0E01` **`0E40`** `0E02 0E03`
normalized string:	`0E01 0E02` **`0E40`** `0E03`

3.1.4 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 (e.g. $a_1 = A_1$ and $a_2 = A_2$), but have different tertiary weights (e.g. $a_3 < A_2$). The unmarked characters will have tertiary weights (such as a_3) equal to MIN_3.

However, a well-formed Unicode Collation Element Table *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.

3.1.5 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 UTR #18: Unicode Regular Expression Guidelines.

3.2 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 weighted. Any code points that are not explicitly mentioned in this table are given a derived collation element, as described in §7 Weight Derivation. There are three types mappings:

- **Normal.** One Unicode character maps to a 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.
  - These are provided for those instances where a canonical decomposable character had to be given a distinct primary weight in the main weight table, which implied that the canonically equivalent combining character sequences should also be given the same weights. These currently include Indic two-part vowels and with some Cyrillic accented characters, to match the expected collating behavior for those scripts.

This table is constructed to be consistent with the Unicode Canonical Equivalence algorithm, and to respect the Unicode character properties. It is not, however, merely algorithmically derivable from those data, since the assignment of levels does take into account characteristics of particular scripts. For example, in general the combining marks are Level 1 ignorables; however, the Indic combining vowels are given non-zero Level 1 weights, since 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 §7 Weight Derivation.

For many languages, some degree of tailoring is required to match user expectations.

3.2.1 File Format

Each of the files consists of a version line followed by an optional variable-weight line, optional rearrangement lines, optional backwards lines, and a series of entries, all separated by newlines. A '#' 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>*
                           <rearrangement>*
                           <entry>+
```

The version line is of the form:

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

The rearrangement line is of the following form. It specifies the characters that are to be rearranged in collation, as discussed above in §3.1.3 Rearrangement.

```
<rearrangement> := '@rearrange' <charList> <eol>

<charList>      := <item> ("," <item> )*
```

The rearrangement lines are determined by the version of the Unicode Standard, using the Logical_Order_Exception property in the UCD.

The variable-weight line has three possible values that may change the weights of collation elements in processing (see §3.2.2 Variable Collation Elements). 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: CANONSEQ usually indicates a canonical equivalence, and COMPATSEQ usually indicates a compatibility equivalence. (There are cases where characters do not actually have the associated equivalences, but behave as if they do.)

Here are some selected entries:

```
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. 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. For more information, see §3.4 Stability.

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

3.2.2 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 are also affected.

There are four possible options for variable weighted characters, with the default being *Shifted:*

- **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 ignorable, and 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 set to ignorable at levels one through three. In addition, a new final-level weight is appended, whose value depends on the type:

Type	L4	Examples
Completely Ignorable	0000	*NULL* `[.0000.0000.0000.0000]`
Ignorable (L1, L2) after 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 ignorables are reset 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.

- **Shift-Trimmed**: the same as **Shifted**, except that all trailing FFFFs are trimmed from the sort key. This option is designed to emulate POSIX behavior.

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

The following gives an example of 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 *hyphen* are separated by *deluge,* since an *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. In this case, it is because they are ignorable, but their fourth level differences are according to the original primary order, which is more intuitive than Unicode order.

Blanked	Non-ignorable	Shift	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.

3.3 Well-Formed Collation Element Tables

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

1. Except in special cases detailed in §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.
   - The reason for this will be explained under Step 4 of the main algorithm.
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
     - Then 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.

3.4 Stability

One very common confusion in terms of collation centers around the notion of *stability* in sorting.

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* 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 comparison, *not* 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, 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 step 3.10 below.

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. 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 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. Since they were assigned characters in Unicode 3.2, their sorting semantics and respective sorting weights will change.
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 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 weights for code points, and as such, it is 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 each input string

Step 1. Produce a normalized form of each input string, applying the following two steps:

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 UTR #15: Unicode Normalization Forms).

- Conformant implementations may skip this step in certain circumstances: see §7 Weight Derivation for more information.

S1.2 If any character is marked as *rearranging* (see §3.1.3 Rearrangement), swap it and the succeeding character (if there is one). In practice, rearranging characters should never appear adjacent to one another. If for some reason they do, then successive pairs in the sequence will be swapped.

For example, if digits 1-4 were *rearranging*, then "1x...12...123y" would result in "x1...21...21y3"

S1.3 If any character is marked as ignorable at all levels, remove it from the string.

For example, if a Control-A (U+0001) has a collation element [0000.0000.0000], then it will be removed.

Example:

input string:	cáb<control-A>
normalized string:	ca´b

4.2 Produce an array of collation elements for each string

Step 2. First we define a combining mark in a string to be *blocked* if there is another combining mark of the same canonical combining class or zero between it and its base character.

The collation element array is built by sequencing through the normalized form as follows:

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 combining marks following S, process each combining mark C.
>
> **S2.1.2** If C is not blocked, find if S + C has a match in the table.
>
> **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 §7.1 Derived Collation Elements

S2.3 Process collation elements according to the variable-weight setting, as described in §3.2.2 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.

Conformant implementations may skip steps 2.1.1 through 2.1.3 if their repertoire of supported character sequences does not require this level of processing.

> **Note**: The reason for considering the extra combining marks C is that 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 didn't have the step of considering the extra combining marks, 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 combining marks are matched. For example, *<a, combining_macron, combining_ring>* would compare as after *a-macron*, and not after *z*. As in the previous note, additional 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 a sort key for each string

Step 3. The sort key is formed by successively appending 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.)

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*
   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.

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 the sort keys

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:

String	Sort Key
cab	**0706** 06D9 06EE 0000 0020 0020 **0020** 0000 **0002** 0002 0002
Cab	**0706** 06D9 06EE 0000 0020 0020 **0020** 0000 **0008** 0002 0002
cáb	**0706** 06D9 06EE 0000 0020 0020 **0021** 0020 0000 0002 0002 0002 0002
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, the first difference is in `0002` vs. `0008` (Level 3)
- For the middle two strings the first difference is in `0020` vs. `0021` (Level 2)
- For the last two strings, the first difference is in `0706` vs. `0712` (Level 1).

Note: At this point we can explain the reason for disallowing ill-formed weights. If ill-formed weights were allowed, the ordering of elements can 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, *a1 = A1.* Then the following incorrect keys would be generated:

1. *"áe" = <a, acute, e> => [a1 e1 0000 acute2 0000 **a3** acute3 e3...]*
2. *"Aé" = <A, e, acute> => [a1 e1 0000 acute2 0000 **A3** acute3 e3...]*

Since 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* vs *a*. With well-formed weights, this does not happen, and you get the following correct ordering:

1. *"Aé" = <A, e, acute> => [a1 e1 0000 a2 **e2** acute2 0000 a3 acute3 e3...]*
2. *"áe" = <a, acute, e> => [a1 e1 0000 a2 **acute2** e2 0000 A3 acute3 e3...]*

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 §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. 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 reordering can represent a Level 1 difference, Level 2 difference, Level 3 difference, or identity (in levels 1 to 3). Since such reordering includes sequences, arbitrary multiple mappings can be specified.

2. Setting levels to be backwards (French) or forwards (normal). Typically this is only needed for the secondary level.
3. Set variable weighting options.
4. Customizing the exact list of rearranging characters.
5. Customizing the exact list of variable collation elements.

5.1 Preprocessing

In addition to tailoring, some implementation 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
- padding digits with zeros to approximate numeric order
- 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 so 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	Sort Key
càb (0)	`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	Sort Key
càb (0)	`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	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 don't 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 quarternary 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. You'll use these new unassigned values 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

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, since 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: define a value BOUND between MINTOP and MAXBOTTOM (the exact value can be chosen based on the expected frequency of synthetic low weights vs. 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).

The result of this process are keys that are never greater 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

```
[(X1+1).0000.0000], [yyyy.zzzz.wwww]
```

They 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. Since 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 §7 Weight Derivation. However, these values can be compacted to a range of values that don't 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 §7 Weight Derivation.

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 stores 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 2000 weights at L2, then 248 characters can be given single keys, while 1792 are given 2 collation keys of the form [yyyy.00zz.00ww] [0000.00nn.0000]. (The 248 can be chosen to be the higher frequency characters!)

6.3.3 Leveraging Unicode Tables

Since 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.

Since 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 §7 Weight Derivation.

This means that only a fraction of the total number of Unicode characters needs to have an explicit collation element associated with them. 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 §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 Section 5.7 of The Unicode Standard, Version 2.0, the mapping table from characters to collation elements can both fast and small. For an example of how this can be done, see §6.11 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 §7 Weight Derivation. However, this does required 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.

6.6 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 §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 be compared once and if they are generally dissimilar, since 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, since 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 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]; for more powerful features, see [ICUCollator]).

1. Java doesn't use a default table in the Unicode Collation Element format: instead it always uses a tailoring syntax. Here is a description of the entries:

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 can be more than one character, to handle contractions and expansions. NULL is completely ignorable, so by using the above operations, various levels of ignorable characters can be specified.

2. Entries 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 rules of the form "x < y & y < z", you can omit "& y", leaving just "x < y < z".

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

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`

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 (not that we have any) 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.

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

	short backwardsOffset;	In case we are going backwards, offset to true contractions 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;* 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.	

6.11.1 Collation Element Format

- 'real' collationElement
  - 16 bits primary (FFE0..FFFF not allowed)
  - 8 bits secondary
  - 8 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 be to have the offsets be not indexes into the arrays, but byte offsets from the start of the table. That would limit the size of the table, but use fewer machine instructions.

6.11.2 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, and has not been tested.

> There is also sample Java code and a demonstration applet available at [Sample]. (That code is not designed to use a flat-file format.)

```
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.
* Since the list is sorted, we can exit if our value is too high.<p>
* Since we have a length, we could implement this as a
* binary search, although we don't 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.
* Don't use 00000000 as a terminator, since that may be a valid CE.
* These elements don't 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>
* Since we have a length, we could implement this as a
* binary search, although we don't 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 on UnicodeData.txt (and documented in UnicodeData.html).

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 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 codepoints are illegal in a data stream. These include non-characters (codepoints with hex values ending in FFFF or FFFE), unpaired surrogates (codepoints between D800 and DFFF), 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 codepoint that is not explicitly mentioned in the table is mapped a sequence of two collation elements as described in 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 (e.g. 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 codepoint CP is separated into two parts, chosen for the correct numerical properties. First, separate off the top 6 bits of the codepoint. Since codepoints can go from 0 to 10FFFF, this will have values from 0 to 21_{16} (= 33_{10}). Add this to the special value BASE.

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

Now take the bottom 15 bits of the code point. Turn the top bit on, so that the value is non-zero.

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

The mapping given to CP is then given by:

```
CP => [.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, etc. in the first collation element and zero in the second. (Since all distinct codepoints have different **AAAA/BBBB** combination, the exact values.)

The value for BASE depends on the type of character:

FB40	CJK Ideograph
FB80	CJK Ideograph Extension A/B
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.

> **Note**: The range of primary weights from FC00 to FFFF is used for trailing weights 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 weighting these characters in this range, the units are ordered independent of subsequent characters with higher weights. Otherwise problems occur, such as in the following example.

Case 1

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

Case 2

2	{G}{A}{K}囗
1	{G}{A}囗

> 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+56D7 (囗).

7.2 Canonical Decompositions

Characters with canonical decompositions do not require mappings to collation elements, because Step 1.1 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 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, since *cedilla* is an L1-ignorable character, and there is some character, e.g. *a-grave*, that decomposes to a sequence containing the same base letter *a.* Note that although strings in Normalization Form C generally don't 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.*

7.3 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 decomposition. e.g.

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

2. Get the CE for each character in the decomposition.

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

3. Set the first L3 to be lookup(L3), where the lookup uses the table in §7.3.1 Tertiary Weight Table. Set the remaining L3 values to MAX (which in the default table is 001F):

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

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

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

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

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

7.3.1 Tertiary Weight Table

Characters are given tertiary weights according to the following table. The Decomposition Type is from the Unicode Character Database. 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 from MIN except for the katakana/hiragana values.

Decomp. Type	Condition	Weight	Samples

NONE		0x0002	i	ب	)	mw	1⁄2	X
<wide>		0x0003	ｉ					
<compat>		0x0004	ⅰ					
<font>		0x0005						
<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>	Uppercase	0x001D				㎿		
<fraction>		0x001E					½	
n/a	(MAX value)	0x001F						

8 Searching and Matching (Informative)

The collation elements can also be used for matching string and for searching for strings, so that a proper native-language match is produced. For example, "ß" will properly match against "ss". Users of search algorithms should be allowed to modify the comparison strength, thus excluding differences at less significant levels. This is especially useful for searching, but can also apply to comparison.

Excluding differences at Level 3 has the effect of ignoring case and compatibility format distinctions between letters when searching. Excluding differences at Level 2 has the effect of ignoring accentual distinctions when searching.

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), the position where a string matches may be indefinite, 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-", etc.
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...". However, this may also depend on language and user customization.
5. The above two conditions can be considered part of a general condition: "Whole Grapheme Clusters Only"; though probably expressed in user interfaces with more natural wording as "Whole Characters Only". This is very similar to the common "Whole Words Only" checkbox that is included in most search dialog boxes. (For more information on grapheme clusters, see UTR #18: Unicode Regular Expression Guidelines)
6. Certain Thai and Lao vowels are swapped with the preceding character. For example, the text string “A\u0E02\u0E40” is modified internally in collation to “A\u0E40\u0E02”. This may mean that a string logically matches a discontiguous section of another string. If, however, the vowels are considered to be part of a grapheme cluster, then this situation is handled by the "whole grapheme clusters only" option.
7. If the matching is does not check for "Whole Grapheme Clusters 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 since collation is defined to observe canonical equivalence.)

The following definitions come into play:

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".

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

DS2. There is a *match* according to C for P within Q[*s,e*] if and only if C generates the same sort key for P as for Q[s,e].

DS3. There is a *canonical match* according to C for P within Q[*s,e*] if and only if there is some Q', canonically equivalent to Q[*s,e*], and some *s'* and *e'* such that P matches within Q[*s',e'*].

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

- By using minimal matches, the issue with ignorables is avoided.

DS5. The match is *grapheme-complete* if *s* and *e* are both at grapheme cluster boundaries.

- By using grapheme-complete matches, contractions and combining sequences are not interrupted.

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 *e* less than or equal to *b* such that for some *s*, P matches within Q[s,e].

- Forward and backward matches can be narrowed to be minimal or grapheme-complete, or broadened to be canonical, or use any mixture of these.

Acknowledgements

Thanks to Karlsson Kent, Vladimir Weinstein, and Richard Gillam for their feedback on previous versions of this document, and Cathy Wissink for her contributions to the text.

References

[AllKeys]	The latest version of this file is found on: http://www.unicode.org/reports/tr10/allkeys.txt
[CanStd]	CAN/CSA 2243.4.1
[FAQ]	Unicode Frequently Asked Questions http://www.unicode.org/faq/ *For answers to common questions on technical issues.*
[Glossary]	Unicode Glossary http://www.unicode.org/glossary/ *For explanations of terminology used in this and other documents.*
[ICUCollator]	http://oss.software.ibm.com/icu/userguide/Collate_Intro.html
[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
[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/
[SoStd]	ISO/IEC 14651
[Test]	http://www.unicode.org/Public/BETA/UCA/CollationTest.html
[Versions]	Versions of the Unicode Standard http://www.unicode.org/versions/ *For details on the precise contents of each version of the Unicode Standard, and how to cite them.*

Modifications

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

9
- Added C4
- Added more conditions in 3.3 Well-Formed Collation Element Tables
- Added S1.3
- Added treatment of ignorables after variables in 3.2.2 Variable Weighting
- Added 3.4 Stability
- Modified and reorganized 7 Weight Derivation. In particular, CJK characters and unassigned characters are given different weights. Added MAX to 7.3.
- Added references
- Minor editing