Preconditions: `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. Neither `i` nor `j` are iterators into `a`.

Effects: Equivalent to `a.insert(t)` for each element in `[i,j)`.

Complexity: Average case $\mathcal{O}(N)$, where N is `distance(i, j)`, worst case $\mathcal{O}(N(\texttt{a.size()}+1))$.

`a.insert_range(rg)`

Result: `void`

Preconditions: `value_type` is *Cpp17EmplaceConstructible* into `X` from `*ranges::begin(rg)`. `rg` and `a` do not overlap.

Effects: Equivalent to `a.insert(t)` for each element `t` in `rg`.

Complexity: Average case $\mathcal{O}(N)$, where N is `ranges::distance(rg)`, worst case $\mathcal{O}(N(\texttt{a.size()}+1))$.

`a.insert(il)`

Effects: Equivalent to `a.insert(il.begin(), il.end())`.

`a_uniq.insert(nh)`

Result: `insert_return_type`

Preconditions: `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()` is `true`.

Effects: If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`.

Postconditions: If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a\_uniq.size()})$.

`a_eq.insert(nh)`

Result: `iterator`

Preconditions: `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()` is `true`.

Effects: If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element.

Postconditions: `nh` is empty.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a\_eq.size()})$.

`a.insert(q, nh)`

Result: `iterator`

Preconditions: `nh` is empty or `a.get_allocator() == nh.get_allocator()` is `true`.

Effects: If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.

Postconditions: `nh` is empty if insertion succeeds, unchanged if insertion fails.

Returns: An iterator pointing to the element with key equivalent to `nh.key()`.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a.size()})$.

`a.extract(k)`

Result: `node_type`

Effects: Removes an element in the container with key equivalent to `k`.

Returns: A `node_type` owning the element if found, otherwise an empty `node_type`.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a.size()})$.

Dxxxx

Preconditions: `value_type` is *Cpp17EmplaceConstructible* into `X` from `*i`. Neither `i` nor `j` are iterators into `a`.

Effects: Equivalent to `a.insert(t)` for each element in `[i,j)`.

Complexity: Average case $\mathcal{O}(N)$, where N is `distance(i, j)`, worst case $\mathcal{O}(N(\texttt{a.size()}+1))$.

`a.insert_range(rg)`

Result: `void`

Preconditions: `value_type` is *Cpp17EmplaceConstructible* into `X` from `*ranges::begin(rg)`. `rg` and `a` do not overlap.

Effects: Equivalent to `a.insert(t)` for each element `t` in `rg`.

Complexity: Average case $\mathcal{O}(N)$, where N is `ranges::distance(rg)`, worst case $\mathcal{O}(N(\texttt{a.size()}+1))$.

`a.insert(il)`

Effects: Equivalent to `a.insert(il.begin(), il.end())`.

`a_uniq.insert(nh)`

Result: `insert_return_type`

Preconditions: `nh` is empty or `a_uniq.get_allocator() == nh.get_allocator()` is `true`.

Effects: If `nh` is empty, has no effect. Otherwise, inserts the element owned by `nh` if and only if there is no element in the container with a key equivalent to `nh.key()`.

Postconditions: If `nh` is empty, `inserted` is `false`, `position` is `end()`, and `node` is empty. Otherwise if the insertion took place, `inserted` is `true`, `position` points to the inserted element, and `node` is empty; if the insertion failed, `inserted` is `false`, `node` has the previous value of `nh`, and `position` points to an element with a key equivalent to `nh.key()`.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a\_uniq.size()})$.

`a_eq.insert(nh)`

Result: `iterator`

Preconditions: `nh` is empty or `a_eq.get_allocator() == nh.get_allocator()` is `true`.

Effects: If `nh` is empty, has no effect and returns `a_eq.end()`. Otherwise, inserts the element owned by `nh` and returns an iterator pointing to the newly inserted element.

Postconditions: `nh` is empty.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a\_eq.size()})$.

`a.insert(q, nh)`

Result: `iterator`

Preconditions: `nh` is empty or `a.get_allocator() == nh.get_allocator()` is `true`.

Effects: If `nh` is empty, has no effect and returns `a.end()`. Otherwise, inserts the element owned by `nh` if and only if there is no element with key equivalent to `nh.key()` in containers with unique keys; always inserts the element owned by `nh` in containers with equivalent keys. The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.

Postconditions: `nh` is empty if insertion succeeds, unchanged if insertion fails.

Returns: An iterator pointing to the element with key equivalent to `nh.key()`.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a.size()})$.

`a.extract(k)`

Result: `node_type`

Effects: Removes an element in the container with key equivalent to `k`.

Returns: A `node_type` owning the element if found, otherwise an empty `node_type`.

Complexity: Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(\texttt{a.size()})$.

§ 23.2.8.1 ©ISO/IEC 857

Table 82 — Character traits requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::compare(p,q,n)`	`int`	*Returns*: `0` if for each `i` in `[0,n)`, `X::eq(p[i],q[i])` is `true`; else, a negative value if, for some `j` in `[0,n)`, `X::lt(p[j],q[j])` is `true` and for each `i` in `[0,j)` `X::eq(p[i],q[i])` is `true`; else a positive value.	linear
`X::length(p)`	`size_t`	*Returns*: the smallest `i` such that `X::eq(p[i],charT())` is `true`.	linear
`X::find(p,n,c)`	`const X::char_type*`	*Returns*: the smallest `q` in `[p,p+n)` such that `X::eq(*q,c)` is `true`, `nullptr` otherwise.	linear
`X::move(s,p,n)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`. Copies correctly even where the ranges `[p,p+n)` and `[s,s+n)` overlap. *Returns*: `s`.	linear
`X::copy(s,p,n)`	`X::char_type*`	*Preconditions*: The ranges `[p,p+n)` and `[s,s+n)` do not overlap. *Returns*: `s`. for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`.	linear
`X::assign(r,d)`	(not used)	assigns `r=d`.	constant
`X::assign(s,n,c)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],c)`. *Returns*: `s`.	linear
`X::not_eof(e)`	`int_type`	*Returns*: `e` if `X::eq_int_type(e,X::eof())` is `false`, otherwise a value `f` such that `X::eq_int_type(f,X::eof())` is `false`.	constant
`X::to_char_type(e)`	`X::char_type`	*Returns*: if for some `c`, `X::eq_int_type(e,X::to_int_type(c))` is `true`, `c`; else some unspecified value.	constant
`X::to_int_type(c)`	`X::int_type`	*Returns*: some value `e`, constrained by the definitions of `to_char_type` and `eq_int_type`.	constant
`X::eq_int_type(e,f)`	`bool`	*Returns*: for all `c` and `d`, `X::eq(c,d)` is equal to `X::eq_int_type(X::to_int_type(c), X::to_int_type(d))`; otherwise, yields `true` if `e` and `f` are both copies of `X::eof()`; otherwise, yields `false` if one of `e` and `f` is a copy of `X::eof()` and the other is not; otherwise the value is unspecified.	constant

Dxxxx

Table 82 — Character traits requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::compare(p,q,n)`	`int`	*Returns*: `0` if for each `i` in `[0,n)`, `X::eq(p[i],q[i])` is `true`; else, a negative value if, for some `j` in `[0,n)`, `X::lt(p[j],q[j])` is `true` and for each `i` in `[0,j)` `X::eq(p[i],q[i])` is `true`; else a positive value.	linear
`X::length(p)`	`size_t`	*Returns*: the smallest `i` such that `X::eq(p[i],charT())` is `true`.	linear
`X::find(p,n,c)`	`const X::char_type*`	*Returns*: the smallest `q` in `[p,p+n)` such that `X::eq(*q,c)` is `true`, `nullptr` otherwise.	linear
`X::move(s,p,n)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`. Copies correctly even where the ranges `[p,p+n)` and `[s,s+n)` overlap. *Returns*: `s`.	linear
`X::copy(s,p,n)`	`X::char_type*`	*Preconditions*: The ranges `[p,p+n)` and `[s,s+n)` do not overlap. *Returns*: `s`. for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`.	linear
`X::assign(r,d)`	(not used)	assigns `r=d`.	constant
`X::assign(s,n,c)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],c)`. *Returns*: `s`.	linear
`X::not_eof(e)`	`int_type`	*Returns*: `e` if `X::eq_int_type(e,X::eof())` is `false`, otherwise a value `f` such that `X::eq_int_type(f,X::eof())` is `false`.	constant
`X::to_char_type(e)`	`X::char_type`	*Returns*: if for some `c`, `X::eq_int_type(e,X::to_int_type(c))` is `true`, `c`; else some unspecified value.	constant
`X::to_int_type(c)`	`X::int_type`	*Returns*: some value `e`, constrained by the definitions of `to_char_type` and `eq_int_type`.	constant
`X::eq_int_type(e,f)`	`bool`	*Returns*: for all `c` and `d`, `X::eq(c,d)` is equal to `X::eq_int_type(X::to_int_type(c), X::to_int_type(d))`; otherwise, yields `true` if `e` and `f` are both copies of `X::eof()`; otherwise, yields `false` if one of `e` and `f` is a copy of `X::eof()` and the other is not; otherwise the value is unspecified.	constant

§ 27.2.2 ©ISO/IEC 1398