24a Container Library

24.12 Summary table for 2022

The various facilities in the vectors, lists, maps, sets, and tree containers in Ada 2022 are summarized in Table 24.2. The queue containers do not fit the same pattern so are excluded. The holder container is also omitted because it is so simple.

In order to save space the following abbreviations are used in the table:

T	container type eg Map	E_T	Element_Type
C: T	Container: container type	K_T	Key_Type
P: C	Position: Cursor	X	Key or Element
L, R	Left, Right	Ex_Ind	Extended_Index
C_T	Count_Type	B	Boolean

+ Index means that another subprogram exists with similar parameters except that the first parameters are of type Vector and Index_Type (or Extended_Index) rather than those involving cursors.

+ Key and + Element similarly apply to maps and sets respectively.

In the case of the new package Stable which is the final item in the individual containers, the aspects such as Nonblocking and Post are given for the various subprograms and types in order to illustrate the technique used in the *ARM*.

Table 24.2 Summary of containers.

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
with Ada.Iterator_Interfaces; **generic**	Y	Y	Y	Y	Y	Y	Y
type Index_Type is range <>;	Y						
type Key_Type is private;			Y	Y			
type Element_Type is private;	Y	Y	Y	Y	Y	Y	Y
with function Hash(X: X_Type) return Hash_Type;			Key		Element		
with function Equivalent_Xs(L, R: X_Type) return B;			Key		Element		
with function "<" (L, R: X_Type) return Boolean is <>;				Key		Element	
with function "=" (L, R: Element_T) return Boolean is <>;	Y	Y	Y	Y	Y	Y	Y
package Ada.Containers. ... with Preelaborate, Remote_Types Nonblocking, Global => in out synchronized is	Vectors	Doubly_ Linked_ Lists	Hashed_ Maps	Ordered_ Maps	Hashed_ Sets	Ordered_ Sets	Multiway_ Trees
function Equivalent_Xs(L, R: X_Type) return Boolean;				Key		Element	
subtype Extended_Index is ... No_Index: constant Ex_Ind := Ex_Ind'First;	Y						
type T is tagged private with Constant_Indexing => Constant_Reference, Variable_Indexing => Reference, Default_Iterator => Iterate, Iterator_Element => Element_Type, Iterator_View => Stable.T, Aggregate => (Empty => Empty, Add_ Stable_Properties => (No tampering with Default_Initial_Condition => Preelaborable_Initialization;	Vector Empty, Named =>Append + indexed), Len, Cap crsrs + els Length =0	List Empty, Unnamed =>Append), Length crsrs + els Length =0	Map Empty, Named =>Insert), Length crsrs + els Length =0	Map Empty, Named =>Insert), Length crsrs + els Length =0	Set no variable indexing Empty, Unnamed =>Include), Length cursors Length =0	Set no variable indexing Empty Unnamed =>Include), Length cursors Length =0	Tree no aggregate Node_Cnt crsrs + els Nd_Cnt=1
type Cursor is private with Preelaborable_Initialization;	Y	Y	Y	Y	Y	Y	Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
Empty_T: constant T;	Vector	List	Map	Map	Set	Set	Tree
No_Element: constant Cursor;	Y	Y	Y	Y	Y	Y	Y
function_Equal_Element(L, R: Element_Type) return B;							Y
function Has_Element(P: C) return Boolean; function Has_Element(C: T; P: C) return Boolean	Y	Y	Y	Y	Y	Y	Y
package T_Iterator_Interfaces is new Ada.Iterator_Interfaces(Cursor, Has_Element);	Vector	List	Map	Map	Set	Set	Tree
function Equal_Subtree(L_Position, R_Position: C) return B;							Y
function "=" (Left, Right: T) return Boolean;	Y	Y	Y	Y	Y	Y	Y
function Equivalent_Sets(L, R: Set) return Boolean; function To_Set(New_Item: E_T) return Set;					Y	Y	
unction Tampering_With_Cursors_Prohibited(C: T) return B;	Y	Y	Y	Y	Y	Y	Y
function Tampering_With_Elements_Prohibited(C: T) return B;	Y	Y	Y	Y			Y
function Maximum_Length return Count_Type;	Y						
function Empty(Capacity: Count_Type := *imp-def*) return T;	Y		Y		Y		
function Empty return T;		Y		Y		Y	Y

function To_Vector(Length: C_T) return Vector;
function To_Vector(New_Item: E_T;
Length: C_T) return Vector;
	Y						

function New_Vector(First, Last: Index_Type) return Vector;	Y						
function "&" (L, R: Vector) return Vector; function "&" (L: Vector; R: E_T) return Vector; function "&" (L: E_T; R: Vector) return Vector; function "&" (L, R: E_T) return Vector;	Y						
function Capacity(C: T) return C_T; procedure Reserve_Capacity(C: in out T; Capacity: C_T);	Y		Y		Y		
function Length(C: T) return Count_Type;	Y	Y	Y	Y	Y	Y	
procedure Set_Length(C: in out Vector; Length: in C_T);	Y						
function Is_Empty(C: T) return Boolean; procedure Clear(C: in out T);	Y	Y	Y	Y	Y	Y	Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees

function Node_Count(C: Tree) return C_T;
function Subtree_Node_Count(P: C) return C_T;
function Subtree_Node_Count(C: Tree; P: C) return C_T;
function Depth(P: C) return C_T:
function Depth(C: Tree; P: C) return C_T;
function Is_Root(P: C) return Boolean;
function Is_Root(C: Tree; P: C) return Boolean;
function Is_Leaf(P: C) return Boolean;
function Is_Leaf(C: Tree; P: C) return Boolean;
function Is_Ancestor_Of(C: Tree; Parent: Cursor;
Position: Cursor) return Boolean;
function Root(C: Tree) return Cursor;
function Meaningful_For(C: Tree; P: C) return Boolean;
							Y

function To_Cursor(C: Vector; Index: Ex_Ind) return Cursor;
function To_Index(P: C) return Ex_Ind;
function To_Index(C: Vector; P: C) return Ex_Ind;
	Y						

function Key(P: C) return K_T; function Key(C: T; P: C) return K_T			Y	Y			
function Element(P: C) return E_T; function Element(C: T; P: C) return E_T;	Y + Index	Y	Y	Y	Y	Y	Y
procedure Replace_Element(C: in out T; P: C; New_Item: E_T);	Y + Index	Y	Y	Y	Y	Y	Y

procedure Query_Element(P: C;
Process: not null access proc(Element: E_T));
procedure Query_Element(C: T; P: C;
Process: not null access proc(Element: E_T));
	Y
	+ Index
		Y			Y	Y	Y

procedure Query_Element(P: C;
Process: not null access proc(Key: K_T; Element: E_T));
procedure Query_Element(C: T; P: C;
Process: not null access proc(Key: K_T; Element: E_T));
			Y	Y			

procedure Update_Element(C: in out T; P: C; Process: not null access proc(Element: in out E_T));	Y + Index	Y					Y
procedure Update_Element(C: in out T; P: C; Process: not null access proc(Key: K_T; Element: in out E_T));			Y	Y			

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
type Constant_Reference_Type (Element: not null access constant E_T) is private with Implicit_Dereference => Element;	Y	Y	Y	Y	Y	Y	Y
type Reference_Type (Element: not null access E_T) is private with Implicit_Dereference => Element;	Y	Y	Y	Y			Y
function Constant Reference(C: aliased in T; P: C) return Constant_Reference_Type;	Y + Index	Y	Y + Key	Y + Key	Y	Y	Y
function Reference(C: aliased in out T; P: C) return Reference_Type;	Y + Index	Y	Y + Key	Y + Key			Y

procedure Assign(Target: in out T; Source: in T);
function Copy(Source: T; Capacity: C_T := 0) return T;
procedure Move(Target, Source: in out T);
	Y	Y
		no Cap
			Y	Y
				no Cap
					Y	Y
						no Cap
							Y
							no Cap

procedure Delete_Leaf(C: in out Tree; P: in out Cursor);
procedure Delete_Subtree(C: in out Tree; P: in out Cursor);
function Child_Count(Parent: Cursor) return C_T;
function Child_Count(C: Tree; Parent: Cursor) return C_T;
function Child_Depth(Parent, Child: Cursor) return C_T;
function Child_Depth(C: Tree;
Parent, Child: Cursor) return C_T;
							Y

procedure Insert_Vector(C: in out Vector; Before: Ex_Ind;
New_Item: Vector);
procedure Insert_Vector(C: in out Vector; Before: Cursor;
New_Item: Vector);
procedure Insert_Vector(C: in out Vector; Before: Cursor;
New_Item: Vector; Position: out Cursor);
	Y						

procedure Insert(C: in out T; Before: Cursor; New_Item: E_T; Count: C_T := 1);	Y + Index	Y					
procedure Insert(C: in out T; Before: Cursor; New_Item: E_T; Position: out Cursor; Count: C_T := 1);	Y	Y					
procedure Insert(C: in out T; Before: Cursor; Position: out Cursor; Count: C_T := 1); new item has default value	Y + Index	Y					

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
procedure Insert_Child(C: in out Tree; Parent, Before: Cursor; New_Item: E_T; Count: C_T := 1);							Y
procedure Insert_Child(C: in out Tree; Parent, Before: Cursor; New_Item: E_T; Position: out Cursor; Count: C_T := 1);							Y
procedure Insert_Child(C: in out Tree; Parent, Before: Cursor; Position: out Cursor; Count: C_T := 1); new item has default value							Y
procedure Insert(C: in out T; Key: K_T; New_Item: E_T; Position: out Cursor; Inserted: out B);			Y	Y	Y no Key	Y no Key	
procedure Insert(C: in out Map; Key: K_T; Position: out Cursor; Inserted: out B); new item has default value			Y	Y			
procedure Insert(C: in out T; Key: K_T; New_Item: E_T);			Y	Y	Y no Key	Y no Key	
procedure Prepend_Vector(C: in out Vector; New_Item: Vector);	Y						
procedure Prepend(C: in out T; New_Item: E_T; Count: C_T := 1);	Y	Y					
procedure Prepend_Child(C: in out Tree; Parent: Cursor; New_Item: E_T; Count: C_T := 1);							Y
procedure Append_Vector(C: in out Vector; New_Item: Vector);	Y						
procedure Append(C: in out T; New_Item: E_T; Count: C_T); procedure Append(C: in out T; New_Item: E_T);	Y	Y					
procedure Append_Child(C: in out Tree; Parent: Cursor; New_Item: E_T; Count: C_T);							Y
procedure Insert_Space(C: in out V; Before: Cursor; Position: out Cursor; Count: C_T := 1);	Y + Index						
procedure Include(C: in out T; Key: Key_Type; New_Item: E_T);			Y	Y	Y no Key	Y no Key	

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
procedure Replace(C: in out T; Key: Key_Type; New_Item: E_T);			Y	Y	Y no Key	Y no Key	
procedure Exclude(C: in out T; Key: Key_Type);			Y	Y	Y Item not Key	Y Item not Key	
procedure Delete(C: in out T; P: in out C; Count: C_T := 1);	Y + Index	Y	Y no Count + Key	Y no Count + Key	Y no Count + Elem	Y no Count + Elem	
procedure Delete_First(C: in out T; Count: C_T := 1); procedure Delete_Last(C: in out T; Count: C_T := 1);	Y	Y		Y no Count		Y no Count	

procedure Delete_Children(C: in out Tree; Parent: Cursor);
procedure Copy_Subtree(Target: in out Tree;
Parent, Before, Source: Cursor);
procedure Copy_Local_Subtree(Target: in out Tree;
Parent, Before, Source: Cursor);
procedure Copy_Subtree(Target: in out Tree;
Parent, Before: Cursor; Source: Tree; Subtree: Cursor);
							Y

procedure Reverse_Elements(C: in out T);	Y	Y					
procedure Swap(C: in out T; I, J: in Cursor);	Y + Index	Y					Y
procedure Swap_Links(C: in out List; I, J: Cursor);		Y					

procedure Splice(Target: in out List; Before: Cursor;
Source: in out List);
procedure Splice(Target: in out List; Before: Cursor;
Source: in out List; Position: in out Cursor);
procedure Splice(Container: in out List; Before: Cursor;
Position: in Cursor);
		Y					

procedure Splice_Subtree(Target: in out Tree;
Parent, Before: Cursor; Source: in out Tree; P: C);
procedure Splice_Subtree(Container: in out Tree;
Parent, Before: Cursor; Position: Cursor);
							Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees

procedure Splice_Children(Target: in out Tree;
Target_Parent, Before: Cursor;
Source: in out Tree; Source_Parent: Cursor);
procedure Splice_Children(Container: in out Tree;
Target_Parent, Before: Cursor;
Source_Parent: Cursor);
							Y

procedure Union(Target: in out Set; Source: Set);
function Union(L, R: Set) return Set;
function "or" (L, R: Set) return Set renames Union;
					Y	Y	

procedure Intersection(Target: in out Set;
Source: Set);
function Intersection(L, R: Set) return Set;
function "and" (L, R: Set) return Set renames Intersection;
					Y	Y	

procedure Difference(Target: in out Set; Source: Set);
function Difference(L, R: Set) return Set;
function "–" (L, R: Set) return Set renames Difference;
					Y	Y	

procedure Symmetric_Difference(Target: in out Set; Source: Set); function Symmetric_Difference (L, R: Set) return Set; function "xor" (L, R: Set) return Set renames Symmetric_Difference;					Y	Y	
function Overlap(L, R: Set) return Boolean; function Is_Subset(Subset: Set; Of_Set: Set) return B;					Y	Y	
function Parent(Position: Cursor) return Cursor; function Parent(C: Tree; Position: Cursor) return Cursor;							Y
function First_Index(C: Vector) return Index_Type;	Y						
function First(C: T) return Cursor;	Y	Y	Y	Y	Y	Y	

function First_Child(Parent: Cursor) return Cursor;
function First_Child(C: Tree; Parent: Cursor) return Cursor;
function First_Child_Element(Parent: Cursor) return E_T;
function First_Child_Element(C: Tree;
Parent: Cursor) return E_T;
							Y

function First_Element(C: T) return E_T;	Y	Y		Y		Y	

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
function First_Key(C: Map) return K_T;				Y			
function Last_Index(C: Vector) return Ex_Ind;	Y						
function Last(C: T) return Cursor;	Y	Y		Y		Y	

function Last_Child(Parent: Cursor) return Cursor;
function Last_Child(C: Tree; Parent: Cursor) return Cursor;
function Last_Child_Element(Parent: Cursor) return E_T;
function Last_Child_Element(C: Tree;
Parent: Cursor) return E_T;
							Y

function Last_Element(C: T) return E_T;	Y	Y		Y		Y	
function Last_Key(C: Map) return K_T;				Y			

function Next(P: C) return Cursor;
function Next(C: T; P: C) return Cursor;
procedure Next(P: in out C);
procedure Next(C: T; P: in out C);
	Y	Y	Y	Y	Y	Y	

function Next_Sibling(P: C) return Cursor;
function Next_Sibling(C: Tree; P: C) return Cursor;
procedure Next_Sibling(P: in out C);
procedure Next_Sibling(C: Tree; P: in out C);
							Y

function Previous(P: C) return Cursor;
function Previous(C: T; P: C) return Cursor;
procedure Previous(P: in out C);
procedure Previous(C: T; P: in out C);
	Y	Y		Y		Y	

function Previous_Sibling(P: C) return Cursor;
function Previous_Sibling(C: Tree; P: C) return Cursor;
procedure Previous_Sibling(P: in out C);
procedure Previous_Sibling(C: Tree; P: in out C);
							Y

function Find_Index(C: T; Item: E_T; Index: I_T := I_T'First) return Ex_Ind;	Y						
function Find(C: T; Element: E_T; P: C := No_Element) return Cursor;	Y	Y			Y no Posn	Y no Posn	Y no Posn
function Find(C: T; Key: K_T) return Cursor;			Y	Y			

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees

function Find_In_Subtree(P: C; Item: E_T) return Cursor;
function Find_In_Subtree(C: Tree;
P: C; Item: E_T) return Cursor;
function Ancestor_Find(P: C; Item: E_T) return Cursor;
function Ancestor_Find(C: Tree;
P: C; Item: E_T) return Cursor;
							Y

function Element(C: T; Key: K_T) return E_T;			Y	Y			
function Reverse_Find_Index(C: T; Item: E_T; Index: I_T := I_T'First) return Ex_Ind;	Y						
function Reverse_Find(C: T; Item: E_T; P: C := No_Element) return Cursor;	Y	Y					
function Floor(C: T; ...) return Cursor; function Ceiling(C: T; ...) return Cursor;				Key: K_T		Item: E_T	
function Contains(C: T; Item: Element_Type) return B;	Y	Y			Y	Y	Y
function Contains(C: T; Key: Key_Type) return Boolean;			Y	Y			
function Equivalent_... (L, R: Cursor) return Boolean; function Equivalent_... (L: Cursor; R: ...) return Boolean; function Equivalent_... (L: ... ; R: Cursor) return Boolean;			Keys K_T		Elements E_T		
function "<" (L, R: Cursor) return Boolean; function ">" (L, R: Cursor) return Boolean; function "<" (L, Cursor; R: ...) return Boolean; function ">" (L, Cursor; R: ...) return Boolean; function "<" (L: ... ; R: Cursor) return Boolean; function ">" (L: ... ; R: Cursor) return Boolean;				Key_T		Element_T	
procedure Iterate(C: in T; Process: not null access proc (P: C));	Y	Y	Y	Y	Y	Y	Y
procedure Reverse_Iterate(C: in T; Process: not null access proc (P: C));	Y	Y		Y		Y	

procedure Iterate_Subtree(P: in C;
Process: not null access proc (P: C));
procedure Iterate_Subtree(C: in Tree: P: in C;
Process: not null access proc (P: C));
							Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
function Iterate(C: in T) return T_Iterator_Interfaces.Reversible_Iterator'Class;	Y	Y	Y Forward	Y	Y Forward	Y	Y Forward
function Iterate(C: in T; Start: in Cursor) return T_Iterator_Interfaces.Reversible_Iterator'Class;	Y	Y		Y		Y	
function Iterate_Subtree(P: in C) return Tree_Iterator_Interfaces.Forward_Iterator'Class; function Iterate_Subtree(C: in Tree; P: in C) return Tree_Iterator_Interfaces.Forward_Iterator'Class;							Y
procedure Iterate_Children(Parent: Cursor; Process: not null access proc (P: C)); procedure Iterate_Children(C: Tree; Parent: Cursor Process: not null access proc (P: C)); procedure Reverse_ Iterate_Children(Parent: Cursor; Process: not null access proc (P: C)) procedure Reverse_ Iterate_Children(C: Tree; Parent: Cursor; Process: not null access proc (P: C))							Y
function Iterate_Children(C: in Tree; Parent: Cursor) return Tree_Iterator_Interfaces.Reversible_Iterator'Class;							Y

generic
with function "<" (Left, Right: E_T) return B is <>;
package Generic_Sorting
with Nonblocking, Global => null is
function Is_Sorted(C: T) return Boolean;
procedure Sort(C: in out T);
procedure Merge(Target, Source: in out T);
end Generic_Sorting;
	Y	Y					

generic type Key_Type (<>) is private;					Y	Y	
with function Key(Element: E_T) return Key_Type;					Y	Y	
with function Hash(Key: K_T) return Hash_Type;					Y		
with function Equivalent_Keys (L, R: Key_Type) return Boolean;					Y		
with function "<" (L, R: Key_Type) return B is <>;						Y	

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
package Generic_Keys with Nonblocking, Global => null is					Y	Y	
function Equivalent_Keys(L, R: Key_Type) return Boolean;						Y	
function Key(P: C) return Key_Type; function Key(C: Set; P: C) return Key_Type;					Y	Y	
function Element(C: Set; Key: K_T) return Element_Type;					Y	Y	

procedure Replace(C: in out Set; Key: Key_Type;
New_Item: E_T);
procedure Exclude(C: in out Set; Key: Key_Type);
procedure Delete(C: in out Set; Key: Key_Type);
					Y	Y	

function Find(C: Set; Key: K_T) return Cursor;					Y	Y	
function Floor(C: Set; Key: K_T) return Cursor; function Ceiling(C: Set; Key: K_T) return Cursor;						Y	
function Contains(C: Set; Key: K_T) return Boolean;					Y	Y	
procedure Update_Element_Preserving_Key (C: in out Set; P: C; Process: not null access proc (Element: in out E_T));					Y	Y	
type Reference_Type(E: not null access E_T) is private with Implicit_Dereference => Element;					Y	Y	
function Reference_Preserving_Key(C: aliased in out Set; P: C) return Reference_Type;					Y	Y	
function Constant_Reference(C: aliased in Set; Key: Key_Type) return Constant_Reference_Type;					Y	Y	
function Reference_Preserving_Key(C: aliased in out Set; Key: Key_Type) return Reference_Type;					Y	Y	
end Generic_Keys;					Y	Y	

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
package Stable is type T (Base: not null access ....T is tagged limited private with Constant_Indexing => Constant_Reference, Variable_Indexing => Reference, Default_Iterator => Iterate, Iterator_Element => Element_Type, Stable_Properties => (Global => null, Default_Initial_Condition => Preelaborable_Initialization; type Cursor is private with Preelaborable_Initialization;	Vectors Len, Cap Length =0	Doubly_ Linked_ Lists Length Length =0	Hashed_ Maps Length Length =0	Ordered_ Maps Length Length =0	Hashed_ Sets Length Length =0	Ordered_ Sets Length Length =0	Multiway_ Trees Node_Cnt Nd_Cnt=1
Empty_T: constant T;	Vector	List	Map	Map	Set	Set	Tree
No_Element: constant Cursor;	Y	Y	Y	Y	Y	Y	Y
function Has_Element(P: C) return Boolean with Nonblocking, Global => in all, Use_Formal => null;	Y	Y	Y	Y	Y	Y	Y
package T_Iterator_Interfaces is new Ada.Iterator_Interfaces(Cursor, Has_Element);	Vector	List	Map	Map	Set	Set	Tree
procedure Assign(Target: in out ...; Source: in ...) with Post => Length(Source) = Length(Target); also Capacity (Target) >= Length(Target) for Vector only	Vs.Vector Vector	DLLs.List List	HMs.Map Map	OMs.Map Map	HSs.Set Set	OSs.Set Set	

procedure Assign(Target: in out Multiway_Trees.Tree;
Source: in Tree)
with Post => Node_Count(Source) = Node_Count(Target);
							Y

function Copy(Source: ...) return ... with Post => Length(Copy'Result) = Length(Source);	Vs.Vector Vector	DLLs.List List	HMs.Map Map	OMs.Map Map	HSs.Set Set	OSs.Set Set	
function Copy(Source: Multiway_Trees.Tree) return Tree with Post => Node_Count(Copy'Result) = Node_Count(Source);							Y

	vectors	lists	hashed maps	ordered maps	hashed sets	ordered sets	multiway trees
type Constant_Reference_Type (Element: not null access constant E_T) is private with Implicit_Dereference => Element, Nonblocking, Global => null, Use_Formal => null, Default_Initial_Condition => (raise Program_Error);	Y	Y	Y	Y	Y	Y	Y no Use_ Formal
type Reference_Type (Element: not null access E_T) is private with Implicit_Dereference => Element; Nonblocking, Global => null, Use_Formal => null, Default_Initial_Condition => (raise Program_Error);	Y	Y	Y	Y			Y no Use_ Formal
... -- *additional subprograms as described in the ARM* ... -- *are declared here* **private** ... -- *not specified by the language* **end** Stable;	Y	Y	Y	Y	Y	Y	Y
private ... -- *not specified by the language* **end** Ada.Containers. ... ;	Y	Y	Y	Y	Y	Y	Y