One Dimensional Arrays (array)

Definition

An instance *A* of the parameterized data type *array*<*E*> is a mapping from an interval *I* = [*a*..*b*] of integers, the index set of *A*, to the set of variables of data type *E*, the element type of *A*. *A*(*i*) is called the element at position *i*.

#include < LEDA/array.h >

Types

array<E>::item	the item type.
array<E>::value_type	the value type.

Creation

array<E>	A(int a, int b)	creates an instance *A* of type *array*<*E*> with index set [*a*..*b*].
array<E>	A(int n)	creates an instance *A* of type *array*<*E*> with index set [0..*n* - 1].
array<E>	A	creates an instance *A* of type *array*<*E*> with empty index set.

Special Constructors

array<E>	A(int low, E x, E y)	creates an instance *A* of type *array*<*E*> with index set [*low*, *low* + 1] initialized to [*x*, *y*].
array<E>	A(int low, E x, E y, E w)	creates an instance *A* of type *array*<*E*> with index set [*low*, *low* + 2] initialized to [*x*, *y*, *w*].
array<E>	A(int low, E x, E y, E z, E w)	creates an instance *A* of type *array*<*E*> with index set [*low*, *low* + 3] initialized to [*x*, *y*, *z*, *w*].

Operations

Basic Operations

E&	A[int x]	returns *A*(*x*). **Precondition** *a* < = *x* < = *b*.
int	A.low()	returns the minimal index *a* of *A*.
int	A.high()	returns the maximal index *b* of *A*.
int	A.size()	returns the size (*b* - *a* + 1) of *A*.

Iteration STL compatible iterators are provided when compiled with `-DLEDA_STL_ITERATORS` (see *LEDAROOT/demo/stl/array.c* for an example).

Implementation

Arrays are implemented by C++vectors. The access operation takes time *O*(1), the sorting is realized by quicksort (time *O*(*n*log *n*)) and the binary_search operation takes time *O*(log *n*), where *n* = *b* - *a* + 1. The space requirement is *O*(*I***sizeof* (*E*)).

Linear Lists (list)

Definition

An instance *L* of the parameterized data type *list<E>* is a sequence of items (*list_item*). Each item in *L* contains an element of data type *E*, called the element type of *L*. The number of items in *L* is called the length of *L*. If *L* has length zero it is called the empty list. In the sequel < *x* > is used to denote a list item containing the element *x* and *L*[*i*] is used to denote the contents of list item *i* in *L*.

#include < LEDA/list.h >

Types

list<E>::item	the item type.
list<E>::value_type	the value type.

Creation

list<E>	L	creates an instance *L* of type *list<E>* and initializes it to the empty list.

Operations

Access Operations

int	L.length()	returns the length of *L*.
int	L.size()	returns *L*.length().
bool	L.empty()	returns true if *L* is empty, false otherwise.
list_item	L.first()	returns the first item of *L* (nil if *L* is empty).
list_item	L.last()	returns the last item of *L*. (nil if *L* is empty)
list_item	L.succ(list_item it)	returns the successor item of item *it*, nil if *it* = *L*.last(). **Precondition** *it* is an item in *L*.
list_item	L.pred(list_item it)	returns the predecessor item of item *it*, nil if *it* = *L*.first(). **Precondition** *it* is an item in *L*.
E	L.inf(list_item it)	returns *L*.contents(*it*).
E	L.front()	returns the first element of *L*, i.e. the contents of *L*.first(). **Precondition** *L* is not empty.
E	L.head()	same as *L.front*().
E	L.back()	returns the last element of *L*, i.e. the contents of *L*.last(). **Precondition** *L* is not empty.
E	L.tail()	same as *L.back*().

Update Operations

list_item	L.push(E x)	adds a new item < *x* > at the front of *L* and returns it (*L.insert*(*x*,*L.first*(),*LEDA*::*before*)).
list_item	L.push_front(E x)	same as *L.push*(*x*).
list_item	L.append(E x)	appends a new item < *x* > to *L* and returns it

		(*L.insert*(*x*,*L.last*(),*LEDA*::*after*)).
list_item	L.push_back(E x)	same as *L.append*(*x*).
E	L.pop()	deletes the first item from *L* and returns its contents. **Precondition** *L* is not empty.
E	L.pop_front()	same as *L.pop*().
E	L.Pop()	deletes the last item from *L* and returns its contents. **Precondition** *L* is not empty.
E	L.pop_back()	same as *L.Pop*().
E	L.del_item(list_item it)	deletes the item *it* from *L* and returns its contents *L*[*it*]. **Precondition** *it* is an item in *L*.
E	L.del(list_item it)	same as *L.del_item*(*it*).
void	L.erase(list_item it)	deletes the item *it* from *L*. **Precondition** *it* is an item in *L*.
void	L.clear()	makes *L* the empty list.

Operators

E&	L[list_item it]	returns a reference to the contents of *it*.
list_item	L += E x	same as *L.append*(*x*); returns the new item.
ostream&	ostream& out << L	same as *L.print*(*out*); returns *out*.
istream&	istream& in >> list<E>& L	same as *L.read*(*in*)); returns *in*.

Iteration

forall_items(*it*, *L*) { ``the items of *L* are successively assigned to *it*'' }

forall(*x*, *L*) { ``the elements of *L* are successively assigned to *x*'' }

STL compatible iterators are provided when compiled with `-DLEDA_STL_ITERATORS` (see *LEDAROOT*/*demo*/*stl*/*list.c* for an example).

Implementation

The data type list is realized by doubly linked linear lists. All operations take constant time except for the following operations: search and rank take linear time $O(n)$, item(i) takes time $O(i)$, bucket_sort takes time $O(n + j - i)$ and sort takes time $O(n*c*\log n)$ where c is the time complexity of the compare function. n is always the current length of the list.

Graphs (graph)

Definition

An instance *G* of the data type *graph* consists of a list *V* of nodes and a list *E* of edges (*node* and *edge* are item types). Distinct graphs have disjoint node and edge lists. The value of a variable of type node is either the node of some graph, or the special value *nil* (which is distinct from all nodes), or is undefined (before the first assignment to the variable). A corresponding statement is true for the variables of type edge.

#include < LEDA/graph.h >

Creation

graph G creates an object *G* of type *graph* and initializes it to the empty directed graph.

Operations

a) Access operations

int	G.outdeg(node v)	returns the number of edges adjacent to node *v* (| *adj_edges*(*v*)|).
int	G.indeg(node v)	returns the number of edges ending at *v* (| *in_edges*(*v*)|) if *G* is directed and zero if *G* is undirected).
int	G.degree(node v)	returns *outdeg*(*v*) + *indeg*(*v*).
node	G.source(edge e)	returns the source node of edge *e*.
node	G.target(edge e)	returns the target node of edge *e*.
node	G.opposite(node v, edge e)	
		returns *target*(*e*) if *v* = *source*(*e*) and *source*(*e*) otherwise.
int	G.number_of_nodes()	returns the number of nodes in *G*.
int	G.number_of_edges()	returns the number of edges in *G*.
list<node>	G.all_nodes()	returns the list *V* of all nodes of *G*.
list<edge>	G.all_edges()	returns the list *E* of all edges of *G*.
node	G.first_node()	returns the first node in *V*.
node	G.last_node()	returns the last node in *V*.
node	G.choose_node()	returns a random node of *G* (nil if *G* is empty).
edge	G.first_edge()	returns the first edge in *E*.
edge	G.last_edge()	returns the last edge in *E*.
edge	G.choose_edge()	returns a random edge of *G* (nil if *G* is empty).
bool	G.is_directed()	returns true iff *G* is directed.
bool	G.is_undirected()	returns true iff *G* is undirected.
bool	G.empty()	returns true iff *G* is empty.

b) Update operations

node	G.new_node()	adds a new node to *G* and returns it.
edge	G.new_edge(node v, node w)	

		adds a new edge (v, w) to G by appending it to $adj\_edges(v)$ and to $in\_edges(w)$ (if G is directed) or $adj\_edges(w)$ (if G is undirected), and returns it.
void	G.hide_edge(edge e)	removes edge e temporarily from G until restored by $G.restore\_edge(e)$.
bool	G.is_hidden(edge e)	returns *true* if e is hidden and *false* otherwise.
list<edge>	G.hidden_edges()	returns the list of all hidden edges of G.
void	G.restore_edge(edge e)	restores e by appending it to $adj\_edges(source(e))$ and to $in\_edges(target(e))$ ($adj\_edges(target(e))$ if G is undirected). **Precondition** e is hidden and neither $source(e)$ nor $target(e)$ is hidden.
void	G.restore_all_edges()	restores all hidden edges.
void	G.hide_node(node v)	removes node v temporarily from G until restored by $G.restore\_node(v)$. All non-hidden edges in $adj\_edges(v)$ and $in\_edges(v)$ are hidden too.
void	G.hide_node(node v, list<edge>& h_edges)	
		as above, in addition, the list of leaving or entering edges which are hidden as a result of hiding v are appended to *h_edges*.
bool	G.is_hidden(node v)	returns *true* if v is hidden and *false* otherwise.
list<node>	G.hidden_nodes()	returns the list of all hidden nodes of G.
void	G.restore_node(node v)	restores v by appending it to the list of all nodes. Note that no edge adjacent to v that was hidden by $G.hide\_node(v)$ is restored by this operation.
void	G.restore_all_nodes()	restores all hidden nodes.
void	G.del_node(node v)	deletes v and all edges incident to v from G.
void	G.del_edge(edge e)	deletes the edge e from G.
void	G.del_all_nodes()	deletes all nodes from G.
void	G.del_all_edges()	deletes all edges from G.
void	G.sort_nodes(node_array<T> A)	
		the nodes of G are sorted according to the entries of node_array A (cf. section Node Arrays). **Precondition** T must be numerical.
void	G.sort_edges(edge_array<T> A)	
		the edges of G are sorted according to the entries of edge_array A (cf. section Edge Arrays). **Precondition** T must be numerical.
void	G.make_undirected()	makes G undirected by appending $in\_edges(v)$ to $adj\_edges(v)$ for all nodes v.
void	G.make_directed()	makes G directed by splitting $adj\_edges(v)$ into $out\_edges(v)$ and $in\_edges(v)$.
void	G.clear()	makes G the empty graph.

f) I/O Operations

void G.print_node(node v, ostream& O = cout)

prints node v on the output stream O.

void G.print_edge(edge e, ostream& O = cout)

prints edge e on the output stream O. If G is directed e is represented by an arrow pointing from source to target. If *G* is undirected *e* is printed as an undirected line segment.

void G.print(string s, ostream& O = cout)

prints *G* with header line *s* on the output stream *O*.

void G.print(ostream& O = cout)

prints *G* on the output stream *O*.

g) Non-Member Functions

node	source(edge e)	returns the source node of edge *e*.
node	target(edge e)	returns the target node of edge *e*.
graph*	graph_of(node v)	returns a pointer to the graph that *v* belongs to.
graph*	graph_of(edge e)	returns a pointer to the graph that *e* belongs to.

h) Iteration

All iteration macros listed in this section traverse the corresponding node and edge lists of the graph, i.e. they visit nodes and edges in the order in which they are stored in these lists.

forall_nodes(*v*, *G*)
{ ``the nodes of *G* are successively assigned to *v*'' }

forall_edges(*e*, *G*)
{ ``the edges of *G* are successively assigned to *e*'' }

forall_rev_nodes(*v*, *G*)
{ ``the nodes of *G* are successively assigned to *v* in reverse order'' }

forall_rev_edges(*e*, *G*)
{ ``the edges of *G* are successively assigned to *e* in reverse order'' }

forall_adj_edges(*e*, *w*)
{ ``the edges adjacent to node *w* are successively assigned to *e*'' }

forall_out_edges(*e*, *w*) a faster version of **forall_adj_edges** for directed graphs.

forall_in_edges(*e*, *w*)
{ ``the edges of $in\_edges(w)$ are successively assigned to *e*'' }

forall_inout_edges(*e*, *w*)
{ ``the edges of $adj\_edges(w)$ and $in\_edges(w)$ are successively assigned to *e*'' }

forall_adj_nodes(*v*, *w*)
{ ``the nodes adjacent to node *w* are successively assigned to v'' }

Implementation

Graphs are implemented by doubly linked lists of nodes and edges. Most operations take constant time, except for all_nodes, all_edges, del_all_nodes, del_all_edges, make_map, make_planar_map, compute_faces, all_faces, make_map, clear, write, and read which take time $O(n + m)$, and adj_edges, adj_nodes, out_edges, in_edges, and adj_faces which take time $O(output\ size)$ where n is the current number of nodes and m is the current number of edges. The space requirement is $O(n + m)$.

Parameterized Graphs (GRAPH)

Definition

A parameterized graph *G* is a graph whose nodes and edges contain additional (user defined) data. Every node contains an element of a data type *vtype*, called the node type of *G* and every edge contains an element of a data type *etype* called the edge type of *G*. We use $< v, w, y >$ to denote an edge (v, w) with information *y* and $< x >$ to denote a node with information *x*.

#include < LEDA/graph.h >

Creation

GRAPH<vtype,etype>	G	creates an instance *G* of type *GRAPH<vtype,etype>* and initializes it to the empty graph.

Operations

vtype	G.inf(node v)	returns the information of node *v*.
const vtype&	G[node v]	returns a reference to *G*.inf(*v*).
etype	G.inf(edge e)	returns the information of edge *e*.
const etype&	G[edge e]	returns a reference to *G*.inf(*e*).
node	G.new_node(vtype x)	adds a new node $< x >$ to *G* and returns it.
edge	G.new_edge(node v, node w, etype x)	adds a new edge $< v, w, x >$ to *G* by appending it to *adj_edges*(*v*) and to *in_edges*(*w*) and returns it.
void	G.sort_nodes(list<node> vl)	makes *vl* the node list of *G*. **Precondition** *vl* contains exactly the nodes of *G*.
void	G.sort_edges(list<edge> el)	makes *el* the edge list of *G*. **Precondition** *el* contains exactly the edges of *G*.
void	G.sort_nodes()	the nodes of *G* are sorted increasingly according to their contents. **Precondition** *vtype* is linearly ordered.
void	G.sort_edges()	the edges of *G* are sorted increasingly according to their contents. **Precondition** *etype* is linearly ordered.

Implementation

Parameterized graphs are derived from directed graphs. All additional operations for manipulating the node and edge entries take constant time.

Undirected Graphs (ugraph)

Definition

An instance *U* of the data type *ugraph* is an undirected graph as defined in section Graphs.

#include < LEDA/ugraph.h >

Creation

ugraph	U	creates an instance *U* of type *ugraph* and initializes it to the empty undirected graph.
ugraph	U(graph G)	creates an instance *U* of type *ugraph* and initializes it with an undirected copy of *G*.

Operations

see section Graphs.

Implementation

see section Graphs.

Parameterized Ugraphs (UGRAPH)

Definition

A parameterized undirected graph *G* is an undirected graph whose nodes and contain additional (user defined) data (cf. Parameterized Graphs). Every node contains an element of a data type *vtype*, called the node type of *G* and every edge contains an element of a data type *etype* called the edge type of *G*.

#include < LEDA/ugraph.h >

UGRAPH<vtype,etype>	U	creates an instance *U* of type *ugraph* and initializes it to the empty undirected graph.

Operations

see section Parameterized Graphs.

Implementation

see section Parameterized Graphs.

Node Arrays (node_array)

Definition

An instance *A* of the parameterized data type *node_array<E>* is a partial mapping from the node set of a graph *G* to the set of variables of type *E*, called the element type of the array. The domain *I* of *A* is called the index set of *A* and *A*(*v*) is called the element at position *v*. *A* is said to be valid for all nodes in *I*.

#include < LEDA/node_array.h >

Creation

node_array<E>	A	creates an instance *A* of type *node_array<E>* with empty index set.
node_array<E>	A(graph G)	creates an instance *A* of type *node_array<E>* and initializes the index set of *A* to the current node set of graph *G*.
node_array<E>	A(graph G, E x)	creates an instance *A* of type *node_array<E>*, sets the index set of *A* to the current node set of graph *G* and initializes *A*(*v*) with *x* for all nodes *v* of *G*.

Operations

E&	A[node v]	returns the variable *A*(*v*). **Precondition** *A* must be valid for *v*.
void	A.init(graph G)	sets the index set *I* of *A* to the node set of *G*, i.e., makes *A* valid for all nodes of *G*.
void	A.init(graph G, E x)	makes *A* valid for all nodes of *G* and sets *A*(*v*) = *x* for all nodes *v* of *G*.

Implementation

Node arrays for a graph *G* are implemented by C++vectors and an internal numbering of the nodes and edges of *G*. The access operation takes constant time, *init* takes time *O*(*n*), where *n* is the number of nodes in *G*. The space requirement is *O*(*n*).

Remark: A node array is only valid for a bounded number of nodes of *G*. This number is either the number of nodes of *G* at the moment of creation of the array or it is explicitely set by the user. Dynamic node arrays can be realized by node maps (cf. section Node Maps).

Edge Arrays (edge_array)

Definition

An instance *A* of the parameterized data type *edge_array*<*E*> is a partial mapping from the edge set of a graph *G* to the set of variables of type *E*, called the element type of the array. The domain *I* of *A* is called the index set of *A* and *A*(*e*) is called the element at position *e*. *A* is said to be valid for all edges in *I*.

#include < LEDA/edge_array.h >

Creation

edge_array<E>	A	creates an instance *A* of type *edge_array*<*E*> with empty index set.
edge_array<E>	A(graph G)	creates an instance *A* of type *edge_array*<*E*> and initializes the index set of *A* to be the current edge set of graph *G*.
edge_array<E>	A(graph G, E x)	creates an instance *A* of type *edge_array*<*E*>, sets the index set of *A* to the current edge set of graph *G* and initializes *A*(*v*) with *x* for all edges *v* of *G*.
edge_array<E>	A(graph G, int n, E x)	creates an instance *A* of type *edge_array*<*E*> valid for up to *n* edges of graph *G* and initializes *A*(*e*) with *x* for all edges *e* of *G*. **Precondition** *n* > = | *E*|. *A* is also valid for the next *n* - | *E*| edges added to *G*.

Operations

E&	A[edge e]	returns the variable *A*(*e*). **Precondition** *A* must be valid for *e*.
void	A.init(graph G)	sets the index set *I* of *A* to the edge set of *G*, i.e., makes *A* valid for all edges of *G*.
void	A.init(graph G, E x)	makes *A* valid for all edges of *G* and sets *A*(*e*) = *x* for all edges *e* of *G*.

Implementation

Edge arrays for a graph *G* are implemented by C++vectors and an internal numbering of the nodes and edges of *G*. The access operation takes constant time, *init* takes time *O*(*n*), where *n* is the number of edges in *G*. The space requirement is *O*(*n*).

Remark: An edge array is only valid for a bounded number of edges of *G*. This number is either the number of edges of *G* at the moment of creation of the array or it is explicitely set by the user. Dynamic edge arrays can be realized by edge maps (cf. section Edge Maps).

Graph Windows (GraphWin)

Definition

GraphWin combines the two types *graph* and *window* and forms a bridge between the graph data types and algorithms and the graphics interface of LEDA. *GraphWin* can easily be used in LEDA programs for constructing, displaying and manipulating graphs and for animating and debugging graph algorithms.

#include < LEDA/graphwin.h >

Creation

GraphWin	gw(graph& G, const char* win_label="")	creates a graph window for graph *G* with a display window of default size and frame label *win_label*.
GraphWin	gw(window& W)	as above, but *W* is used as display window.

Operations

a) Window Operations

void	gw.display()	displays gw at default position.
bool	gw.edit()	enters the edit mode of *GraphWin* that allows to change the graph interactively by operations associated with certain mouse events or by choosing operations from the windows menu bar (see section edit-mode for a description of the available commands and operations). Edit mode is terminated by either pressing the *done* button or by selecting *exit* from the file menu. In the first case the result of the edit operation is *true* and in the latter case the result is *false*.
bool	gw.open()	as above, but displays the window at default position.
void	gw.close()	closes the window.
void	gw.message(const char* msg)	displays the message *msg* at the top of the window.
string	gw.get_message()	returns the current messsage string.
void	gw.del_message()	deletes a previously written message.
double	gw.get_xmin()	returns the minimal x-coordinate of the window.
double	gw.get_ymin()	returns the minimal y-coordinate of the window.
double	gw.get_xmax()	returns the maximal x-coordinate of the window.
double	gw.get_ymax()	returns the maximal y-coordinate of the window.

b) Graph Operations

void	gw.clear_graph()	deletes all nodes and egdes.
graph&	gw.get_graph()	returns a reference of the graph of *gw*.
void	gw.update_graph()	this operation has to be called after any update operation that has been performed directly (not by Graph Win) on the underlying graph, e.g., deleting or inserting nodes or edges.

I) Miscellaneous

void	gw.set_graph(graph& G)	makes *G* the graph of *gw*.
bool	gw.wait()	waits until the done button is pressed (*true* returned) or exit is selected from the file menu (*false* returned).
bool	gw.wait(const char* msg)	displays *msg* and waits until the done button is pressed (*true* returned) or exit is selected from the file menu (*false* returned).
bool	gw.wait(float sec, const char* msg="")	
		as above but waits no longer than *sec* seconds returns ?? if neither button was pressed within this time interval.
void	gw.acknowledge(string s)	displays string *s* and asks for acknowledgement.
node	gw.ask_node()	asks the user to select a node with the left mouse button. If a node is selected it is returned otherwise nil is returned.
edge	gw.ask_edge()	asks the user to select an edge with the left mouse button. If an edge is selected it is returned otherwise nil is returned.