BSP1 Listenbenutzung

⟨*t1.c*⟩≡

```
#include <LEDA/list.h>
#include <LEDA/point.h>
#include <LEDA/tuple.h>
#include <LEDA/stream.h>

typedef two_tuple<int,point> my_list_info;

int main() {
  list<my_list_info> L;
  point p(1,1);
  my_list_info t1, t2(11,p);
  L.append(t1);
  L.append(t2);

  cout << L << endl;

  ofstream O("tmp");
  if (!O) error_handler(1,"ostream error.");
  O << L << endl;
  O.close();

  ifstream I("tmp");
  if (!I) error_handler(1,"istream error.");
  I >> L;
  I.close();

  cout << L << endl;

  return 0;
}
```

BSP2 Graphenparametrisierung

⟨*t2.c*⟩≡

```
#include <LEDA/ugraph.h>
#include <LEDA/point.h>

class my_node_info {
  int   z;
  point p;
public:
  my_node_info() : z(0),p() {}

  my_node_info(int z_init, point p_init)
  { z = z_init; p = p_init; }

  my_node_info(const my_node_info& t)
  { z=t.z; p=t.p; }

  my_node_info& operator=(const my_node_info& t)
  { z=t.z; p=t.p; return *this; }

  int   number()   const { return z; }
  point position() const { return p; }

friend int compare(const my_node_info& t1, const my_node_info& t2)
{ return (t1.z-t2.z); }

friend ostream& operator<<(ostream& O, const my_node_info& t)
{ O << t.z << " " << t.p;  return O; }

friend istream& operator>>(istream& I, my_node_info& t)
{ int z; point p;
  I >> z >> p;
  t=my_node_info(z,p);
  return I;
}

};

inline const char* leda_tname(const my_node_info*) { return "my_node_info"; }
// WICHTIG BEIM EINLESEN !!!

typedef int my_edge_info;
// leda_tname fuer int schon definiert!!!
```

⟨*t2.c*⟩+≡

```
int main()
{
  UGRAPH<my_node_info,my_edge_info> U;
  node v1 = U.new_node( my_node_info(1,point(0,0)) );
  node v2 = U.new_node( my_node_info(2,point(10,0)) );
  node v3 = U.new_node( my_node_info(3,point(0,10)) );
  node v4 = U.new_node( my_node_info(4,point(10,10)) );

  edge e1 = U.new_edge (v1,v2,1);
  edge e2 = U.new_edge (v1,v3,2);
  edge e3 = U.new_edge (v1,v4,3);
  edge e4 = U.new_edge (v2,v4,4);

/* UGRAPH is

  v3    v4
  |   / |
  e2 e3 e4
  | /   |
  v1-e1-v2

*/

  cout << "\nObjekt Iteration:\n";
  node v; edge e;
  forall_nodes(v,U) cout << U[v] << endl;
  forall_edges(e,U) cout << U[e] << endl;

  cout << "\nOutput Operator:\n";
  cout << U << endl;

  U.write("mygraph.ug");

  UGRAPH<my_node_info,my_edge_info> U_r;
  U_r.read("mygraph.ug");

  cout << "\nAfter O/I:\n";
  cout << U_r << endl;

  return 0;
}
```

BSP3 Windowbenutzung

⟨*t3.c*⟩≡

```
#include <LEDA/window.h>

static list<point> PL;

void draw_points(window* pW)
{ point p;
  forall(p,PL) pW->draw_filled_node(p);
}

int main()
{
  window W(600,600);
  W.init(0,100,0,5);
  W.set_show_coordinates(true);
  W.set_node_width(4);
  W.set_redraw(draw_points);
  int EXIT = W.button("Exit","exits the program");
  // simple button "Exit"

  menu MM("My Menu");
  MM.button("First Action",11);
  MM.button("Second Action",12);
  int ACTION = W.button("Actions",MM);
  // pull down menu "Actions"

  W.display();

  point p;
  while (true) {
    int mb = W.read_mouse(p);
    if (mb == MOUSE_BUTTON(3) || mb == EXIT) break;
    switch (mb) {
      case 11:  W.acknowledge("Action 1"); break;
      case 12:  W.acknowledge("Action 2"); break;
    }
    PL.append(p);
    draw_points(&W);
  }
  return 0;
}
```

BSP4 LEDA DFS und Graphwin

$\langle t4.c\rangle\equiv$

```
#include <LEDA/ugraph.h>
#include <LEDA/graph_gen.h>
#include <LEDA/basic_graph_alg.h>
#include <LEDA/graphwin.h>

int main()
{
  ugraph U; node v;
  random_simple_undirected_graph(U,20,30);

  node_array<bool> visited(U,false);
  list<node> L = DFS(U,U.first_node(),visited);

  GraphWin GW(U);
  GW.display();

  cout << "DFS Durchmusterung:\n";
  forall(v,L) cout << index(v) << " " << endl;

  GW.edit();
  return 0;
}
```

BSP5 Delaunay Triangulierungen

$\langle t5.c \rangle \equiv$

```
#include <LEDA/delaunay.h>

int main()
{
  float_delaunay_triang DT;
  point p;
  for(int i=0; i<5; i++) {
    cout << "Bitte Punkt eingeben:";
    cin >> p;
    DT.insert(p);
  }
  cout << "Bitte Standort eingeben:";
  cin >> p;
  node vn = DT.nearest_neighbor(p);
  cout << "Naechster Punkt is " << DT.pos(vn) << endl;
}
```