91

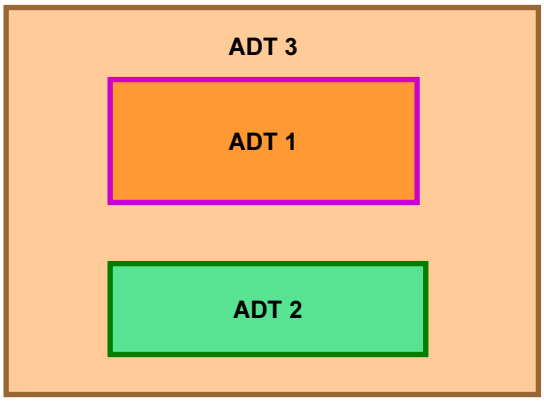
# Chapter 3

## Nested ADTs

92

## Nested ADT's

A high-level ADT may be built on top of (many) lower-level ADT's.

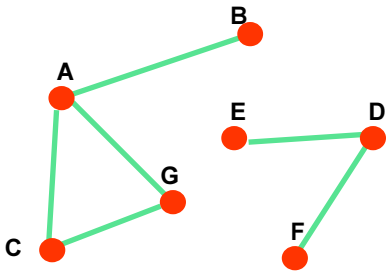


93

## Example: GRAPH

GRAPH = SET of vertices + SET of edges

edge = pair of vertices.




V = { A, B, C, D, E, F,G }

E = { {A,B}, {A,G}, {C,G}, {A,C}, {E,D}, {D,F} }

neighbors(G) = {A,C}

connected-component(G) = {G,C,A,B}

94

## Other Possible GRAPH Implementations

- SET of "extended" vertices.
  - An "extended" vertex contains a list of neighboring vertices.
- SET of vertices + adjacency matrix.
  - How is this similar to the *bit array* implementation of SET ?

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| A | | 1 | 1 | | | | 1 |
| B | 1 | | | | | | |
| C | 1 | | | | | | 1 |
| D | | | | | 1 | 1 | |
| E | | | | 1 | | | |
| F | | | | 1 | | | |
| G | 1 | | 1 | | | | |

95

## GRAPH interface

- *Create* graph.
- *Destroy* graph.
- *Add vertex* to graph.
- *Add edge* to graph.
- *Remove vertex* from graph.
- *Remove edge* from graph.
- Find *neighbors* of a vertex in a graph.
- Find *connected component* of a vertex in a graph.
- *Are* two vertices in a graph *connected* ?
- *Print* graph.

**Other possible interface functions:**

*Copy* graph.
Vertex *iterator*.
Edge *iterator*.

96

## graph.h - ADT interface

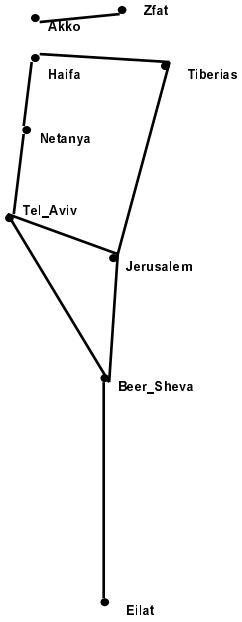
Note that the contents of the `graph_rec` structure are still unknown.

```
#ifndef GRAPH_HDR_ /* '{' */
#define GRAPH_HDR_
#include "set.h"
#define VERTEX_LABEL_SIZE 30
typedef struct graph_rec*  Graph;
typedef char* Label;
Graph     GraphCreate              (void);
void      GraphDestroy             (Graph);
Result    GraphAddVertex           (Graph,Label);
Result    GraphAddEdge             (Graph,Label,Label);
Result    GraphRemoveVertex        (Graph,Label);
Result    GraphRemoveEdge          (Graph,Label,Label);
Set       GraphNeighbors           (Graph,Label);
Set       GraphConnectedComponent  (Graph,Label);
bool      GraphTwoConnected        (Graph,Label,Label);
void      GraphPrint               (Graph);
#endif /* GRAPH_HDR_ '}' */
```

97

## A graph as a model for a road network

Cities are vertices.
Roads are edges.




98

## graph_app.c

```
#include <stdio.h>
#include "graph.h"

Graph BuildVertices(Graph road)
{ GraphAddVertex(road, "Jerusalem");
  GraphAddVertex(road, "Tel_Aviv");
  GraphAddVertex(road, "Haifa");
  GraphAddVertex(road, "Tiberias");
  GraphAddVertex(road, "Akko");
  GraphAddVertex(road, "Zfat");
  GraphAddVertex(road, "Netanya");
  GraphAddVertex(road, "Beer_Sheva");
  GraphAddVertex(road, "Eilat");
  return road;
}
```

Sorry, MS

We do not test returned values :-(

99

## graph_app.c

Neither do we here :-(

```
Graph BuildEdges(Graph road)
{ GraphAddEdge(road,  "Jerusalem", "Tel_Aviv");
  GraphAddEdge(road,  "Akko",      "Haifa");
  GraphAddEdge(road,  "Jerusalem", "Beer_Sheva");
  GraphAddEdge(road,  "Jerusalem", "Tiberias");
  GraphAddEdge(road,  "Tel_Aviv",  "Netanya");
  GraphAddEdge(road,  "Tel_Aviv",  "Beer_Sheva");
  GraphAddEdge(road,  "Netanya",   "Haifa");
  GraphAddEdge(road,  "Haifa",     "Tiberias");
  GraphAddEdge(road,  "Beer_Sheva","Eilat");
  GraphAddEdge(road,  "Akko",      "Zfat");
  GraphAddEdge(road,  "Tel_Aviv",  "Jerusalem");

  GraphRemoveEdge(road, "Haifa",   "Akko");
  return road;
}
```

100

## graph_app.c

```
Graph
BuildGraph(void)
{ Graph g = GraphCreate();
  if (!g) return NULL;
  return BuildEdges(BuildVertices(g));
}
```

Using the returned value

```
void
CheckConnected(Graph roads, char *town1, char *town2)
{ char const * const ans =
      GraphTwoConnected(roads,town1,town2) ? "" : "not ";
  printf("%s is %sconnected to %s\n", town1, ans, town2);
  return;
}
```

101

## graph_app.c

```
int main(void)
{ Graph road_net = BuildGraph();
  if (!road_net) return 1;

  GraphPrint(road_net);
  SetPrint(GraphNeighbors(road_net, "Jerusalem"));
  SetPrint(GraphConnectedComponent(road_net,"Jerusalem"));

  check_connected(road_net,"Tiberias", "Eilat");
  check_connected(road_net,"Akko", "Jerusalem");

  GraphDestroy(road_net);
  return 0;
}
```

102

## Makefile

```
graph.o: graph.h graph.c set.h set.o
        gcc -c graph.c -o tmp.o
        ld tmp.o set.o -o graph.o
        rm tmp.o

graph_app: graph_app.c graph.h graph.o
        gcc graph_app.c graph.o -o graph_app
```

This incorporates `set.o` into `graph.o`.



103

## graph_app output
## לחץ כדי להוסיף כותרת

This slide was viciously destroyed by M$-PowerPoint and could not be restored from any back-up file except one that was more than two years old and was based on an older version of PowerPoint

• לחץ כדי להוסיף טקסט

VERTICES:
Jerusalem Tel_Aviv Haifa Tiberias Akko Zfat Netanya Beer_Sheva Eilat

EDGES:
Jerusalem--Tel_Aviv Jerusalem--Beer_Sheva Jerusalem--Tiberias
Tel_Aviv--Netanya Tel_Aviv--Beer_Sheva Netanya--Haifa
Haifa--Tiberias Beer_Sheva--Eilat Akko--Zfat

Tel_Aviv Beer_Sheva Tiberias

Jerusalem Tel_Aviv Netanya Haifa Tiberias Beer_Sheva Eilat

Tiberias is connected to Eilat
Akko is not connected to Jerusalem

104

## graph.c - implementation

```
#include <stdio.h>
#include <stdlib.h>
#include "graph.h"
typedef struct {
   char label[VERTEX_LABEL_SIZE + 1];
} vertex_rec, *Vertex;
typedef struct {
   Vertex vertex1, vertex2;
} edge_rec, *Edge;
struct graph_rec {
   Set vertices, edges;
};
#define ALLOCATE(var,type,num)   \
           {if((var = (type*)malloc((num)*sizeof(type))) == NULL) \
              { fprintf(stderr,"Cannot allocate\n"); exit(1); }}
#define EMPTY_VERTEX_SET()       \
           SetCreate(vertex_cmp,vertex_cpy,vertex_lbl,vertex_fre)
#define EMPTY_EDGE_SET()         \
           SetCreate(edge_cmp,edge_cpy,edge_lbl,edge_fre)
#define LABEL(v) (((Vertex)(v))->label)
```

105

## graph.c - vertex functions

```
static bool vertex_cmp(Element e1, Element e2)
{ return (bool)!strcmp(LABEL(e1), LABEL(e2));
}

static Element vertex_cpy(Element e)
{  Vertex v;
   ALLOCATE(v, vertex_rec, 1);
   *v = *(Vertex)e;
   return v;
}

static char* vertex_lbl(Element e)
{  char *s;
   ALLOCATE(s, char, VERTEX_LABEL_SIZE + 1);
   return strcpy(s,LABEL(e));
}

static void vertex_fre(Element e)
{ free(e);  /* free() is NULL pointer protected */
  return;
}
```

Why don't we use `free()` directly ?

106

## graph.c - edge functions

```
static bool edge_cmp(Element e1, Element e2)
{  Edge edge1 = e1;    /* Saves casting and     */
   Edge edge2 = e2;    /* easier understanding  */

   return (bool)((vertex_cmp(edge1->vertex1,edge2->vertex1)
                  && vertex_cmp(edge1->vertex2,edge2->vertex2))
              || (vertex_cmp(edge1->vertex2,edge2->vertex1)
                  && vertex_cmp(edge1->vertex1,edge2->vertex2)));
}


static Element edge_cpy(Element e)
{  Edge edge;
   ALLOCATE(edge, edge_rec, 1);
   edge->vertex1 = vertex_cpy(((Edge)e)->vertex1);
   edge->vertex2 = vertex_cpy(((Edge)e)->vertex2);
   return edge;
}
```

107

## graph.c - edge functions

```
static char* edge_lbl(Element e)
{ Edge  edge = e;
  char* s;
  ALLOCATE(s, char, 2*VERTEX_LABEL_SIZE+3);
  sprintf(s,"%s--%s", LABEL(edge->vertex1),
                      LABEL(edge->vertex2));
  return s;
}


static void edge_fre(Element e)
{ Edge edge = e;
  if (edge == NULL) return;
  vertex_fre(edge->vertex1);
  vertex_fre(edge->vertex2);
  free(edge);
  return;
}
```

108

## graph.c - implementation

```
static Vertex create_vertex(Label l)
{  Vertex v;
   ALLOCATE(v, vertex_rec, 1);
   strcpy(v->label, l);
   return v;
}

static Edge create_edge(Label l1, Label l2)
{  Edge e;
   ALLOCATE(e, edge_rec, 1);
   e->vertex1 = create_vertex(l1);
   e->vertex2 = create_vertex(l2);
   return e;
}
```



109

## graph.c - implementation

```
Graph GraphCreate(void)
{  Graph g;
   ALLOCATE(g, struct graph_rec, 1);
   g->vertices = EMPTY_VERTEX_SET();
   g->edges    = EMPTY_EDGE_SET();
   if (g->vertices && g->edges) return g;
   GraphDestroy(g);
   return NULL;
}

void GraphDestroy(Graph g)
{ if (g == NULL) return;
  SetDestroy(g->vertices);
  SetDestroy(g->edges);
  free(g);
  return;
}
```

110

## graph.c

Memory Leak :-(

```
Result GraphAddVertex(Graph g, Label l)
{ return SetAdd(g->vertices, create_vertex(l)); }

Result GraphAddEdge(Graph g, Label l1, Label l2)
{ Vertex v1 = create_vertex(l1),
         v2 = create_vertex(l2);
  bool   b  = SetIsIn(g->vertices,v1)
              && SetIsIn(g->vertices,v2);
  vertex_fre(v1);  vertex_fre(v2);
  if (b) {
    Edge   e   = create_edge(l1, l2);
    Result res = SetAdd(g->edges, e);
    edge_fre(e);
    return res;
  }
  return FAILURE;
}
```

No Memory Leak :-)



111

## graph.c

```
Result GraphRemoveEdge(Graph g, Label l1, Label l2)
{  Edge   e   = create_edge(l1, l2);
   Result res = SetRemove(g->edges, e);
   edge_fre(e);
   return res;
}


void GraphPrint(Graph g)
{  printf("VERTICES:\n");
   SetPrint(g->vertices);
   printf("\nEDGES:\n");
   SetPrint(g->edges);
}
```

112

## graph.c

A helper function

Why don't we use the one in set.c ?

```
static Set addOrDestroy(Set s, Element e)
{ if (SetAdd(s,e) != FAILURE) return s;
  SetDestroy(s);
  return NULL;
}
```

See an older version in graph.c (old)

```
static Set incident_edges(Graph g,Label l)
{  Edge e;
   Set inc = EMPTY_EDGE_SET();
   if (inc != NULL)
     SET_FOREACH(e, g->edges)
       if (   !strcmp(LABEL(e->vertex1),l)
           || !strcmp(LABEL(e->vertex2),l))
          if (!addOrDestroy(inc,e)) return NULL;
   return inc;
}
```



113

## graph.c

```
Result GraphRemoveVertex(Graph g, Label l)
{ Vertex v   = create_vertex(l);
  Result res = SetRemove(g->vertices,v);
  vertex_fre(v);
  if (res != FAILURE) {
    Edge e;  /* local variables */
    Set inc = incident_edges(g, l);
    if (inc == NULL) return FAILURE;
    SET_FOREACH(e, inc) {
      GraphRemoveEdge(g,LABEL(e->vertex1),LABEL(e->vertex2));
              /* The above should NEVER fail */
    }
    SetDestroy(inc);
  }
  return res;
}
```

114

## graph.c

See an older version in graph.c (old)

```
Set GraphNeighbors(Graph g, Label l)
{ Vertex  v = create_vertex(l);
  Set neigh = EMPTY_VERTEX_SET();
  if (neigh != NULL) {
    Edge e;
    SET_FOREACH(e,g->edges) {
      if (vertex_cmp(e->vertex1,v)) {
         if (!addOrDestroy(neigh,e->vertex2)) break;
      } else if (vertex_cmp(e->vertex2,v))
         if (!addOrDestroy(neigh,e->vertex1)) break;
    }
    if (e != NULL) neigh = NULL;
  }
  vertex_fre(v);
  return neigh;
}
```

There was a break



115

## graph.c

See an older version in graph.c (old)

```
static Result
depth_first_traversal(Graph g, Vertex v, Set component)
{ Result  r = SetAdd(component, v);
  Set neigh = GraphNeighbors(g, LABEL(v));
  if (neigh == NULL)  return FAILURE;
  if (r != FAILURE) {
    Vertex vert;
    SET_FOREACH(vert, neigh) {
      if (!SetIsIn(component, vert))
        if ((r = depth_first_traversal(g, vert, component))
              == FAILURE)   break;
    }
  }
  SetDestroy(neigh);
  return r;
}
```

116

## graph.c

```
Set GraphConnectedComponent(Graph g, Label l)
{ Set component = EMPTY_VERTEX_SET();
  if (component == NULL) return NULL;
  if (depth_first_traversal(g,create_vertex(l),component) == SUCCESS)
    return component;
  SetDestroy(component);
  return NULL;
}

bool GraphTwoConnected(Graph g, Label l1, Label l2)
{ Set cc = GraphConnectedComponent(g,l1);
  if (cc != NULL) {
    Vertex  v = create_vertex(l2);
    bool    b = SetIsIn(cc,v);
    SetDestroy(cc);  vertex_fre(v);
    return b;
  }
  return FALSE;
}
```

See an older version in graph.c (old)

Do you see a memory leak here ? ;-)

See an older version in graph.c (old)



117

## graph.c (old version)

```
static Set incident_edges(Graph g,Label l)
{
   Edge e;
   Set inc = EMPTY_EDGE_SET();

   SET_FOREACH(e, g->edges)
     if (   !strcmp(LABEL(e->vertex1),l)
         || !strcmp(LABEL(e->vertex2),l))
        if (SetAdd(inc,e)==FAILURE) {
           SetDestroy(inc);
           return NULL;
        }
   return inc;
}
```

118

## graph.c (old version)

```
Set GraphNeighbors(Graph g, Label l)
{  Set neigh = EMPTY_VERTEX_SET();
   Vertex v = create_vertex(l);
   Edge e;
   SET_FOREACH(e,g->edges) {
      if (vertex_cmp(e->vertex1,v))
         if (SetAdd(neigh,e->vertex2)==FAILURE) {
            SetDestroy(neigh);  vertex_fre(v);
            return NULL;
         }
      if (vertex_cmp(e->vertex2,v))
         if (SetAdd(neigh,e->vertex1)==FAILURE) {
            SetDestroy(neigh);  vertex_fre(v);
            return NULL;
         }
   }
   vertex_fre(v);  return neigh;
}
```

119

## graph.c (old version)

```
static Set glob_comp;  /* Ooops ... a global variable */

static Result depth_first_traversal(Graph g, Vertex v)
{  Set neigh;
   Vertex vert;
   if (SetAdd(glob_comp, v) == FAILURE) return FAILURE;
   if ((neigh = GraphNeighbors(g, LABEL(v))) == NULL)
      return FAILURE;
   SET_FOREACH(vert, neigh)
      if (!SetIsIn(glob_comp, vert))
         if (depth_first_traversal(g, vert) == FAILURE) {
            SetDestroy(neigh);
            return FAILURE;
         }
   SetDestroy(neigh);
   return SUCCESS;
}
```

120

## graph.c (old version)

```
Set GraphConnectedComponent(Graph g, Label l)
{
  Set component;
  SetDestroy(glob_comp); /* If it was used before */
  component = glob_comp = EMPTY_VERTEX_SET();

  if (depth_first_traversal(g,create_vertex(l)) == FAILURE){
    SetDestroy(glob_comp);
    glob_comp = NULL;
    return NULL;
  }
  glob_comp = NULL;
  return component;
}
```

? Do you see a memory leak here



121

## graph.c (old version)

```
bool GraphTwoConnected(Graph g, Label l1, Label l2)
{
   Vertex v;
   bool   res;
   Set cc = GraphConnectedComponent(g,l1);

   if (cc == NULL)  return FALSE;
   v   = create_vertex(l2);
   res = SetIsIn(cc,v);
   SetDestroy(cc);
   vertex_fre(v);
   return res;
}
```

122