Implementation of the Dynamic Connectivity Algorithm by Monika Rauch Henzinger and Valerie King

David Alberts[1]

B 95–10
June 1995

[1] Freie Universität Berlin, Institut für Informatik, Takustr. 9, D-14195 Berlin, Germany, E-mail: alberts@inf.fu-berlin.de, Phone/Fax: +49 30 838 75 164/75 109. Graduiertenkolleg "Algorithmische Diskrete Mathematik", supported by the Deutsche Forschungsgemeinschaft, grant We 1265/2-1.

Contents

Chapter 1

Introduction

M. Rauch Henzinger and V. King [10] recently developed a new dynamic connectivity algorithm which for the first time achieved a polylogarithmic update time. We implement this algorithm in `C++` using LEDA [11, 12]. The previous best bound was $O(\sqrt{n})$ [4, 5] where n is as usual the number of nodes in the graph for general graphs. For plane graphs [7] and for planar graphs [6] polylogaritmic algorithms were known.

We are given the following problem. Let G be a graph on a fixed node set of size n. We want to answer quickly queries of the type "Are the nodes u and v connected in the current graph?". These queries are intermixed with edge updates of the graph, i.e., edges are inserted or deleted. We are thus looking for a data structure that allows three kinds of operations.

`ins(node u, node v)`	Insert an edge connecting `u` and `v`.
`del(edge e)`	Delete the edge `e`.
`connected(node u, node v)`	Return "True" if there is a path connecting the nodes `u` and `v` in the current graph.

The data structure proposed by Henzinger and King achieves $O(\log n)$ worst case query time and $O(\log^3 n)$ amortized expected update time. Here the expectation is over random choices in the update algorithm and holds for any input. However, the amortization holds only if there are at least $\Omega(m_0)$ updates where m_0 is the number of edges in the initial graph. We base our implementation of the data structure on randomized search trees [2], so we achieve only $O(\log n)$ expected query time. Choosing a different variant of balanced binary trees would yield $O(\log n)$ worst case query time, but in practice we expect a better behavior of randomized search trees because they are comparatively easy to implement.

In all dynamic connectivity algorithms the basic idea is to maintain a spanning forest of the current graph. If there are only insertions of edges, a situation often called *semi-dynamic*, then the trees in the forest are implicitly represented by a UNION-FIND data structure [3]. Queries are answered by comparing the representatives of the two given nodes. When an edge is inserted, it is first checked whether its vertices are already connected. If this is the case there is no modification of the data structure necessary. If they were not connected before the insertion their connected components are UNIONed together.

If deletions are also possible, it might occur that a forest edge is deleted. Then there are two cases possible. The deleted edge is either a bridge, then one connected component is split into two or it is not a bridge, then there is a replacement edge among those edges which were no forest edges before. So in contrast to the easy semi-dynamic setting we also have to keep track of edges which do not immediately become forest edges at the time of their insertion in order to find replacement edges fast or to see that there are no such edges. This is the main difficulty in the setting with edge insertions and deletions, which is also called the *fully-dynamic* setting.

The previous best algorithm [4, 5] is based on an earlier approach by Frederickson [8]. Frederickson uses a sophisticated vertex decomposition scheme of the graph to achieve a bound of $O(\sqrt{m})$ per update and a constant query time where m is the number of edges in the current graph. The algorithm in [4] uses an additional edge decomposition scheme and achieves a bound of $O(\sqrt{n})$ and constant query time. These algorithms are deterministic. The algorithm by Henzinger and King is randomized and uses a novel decomposition scheme of the graph.

This report is structured as follows. In the next Chapter we describe the main ideas used in the Henzinger and King algorithm. Randomized balanced binary trees form the core of the data structure. They are implemented in Chapter 3 and in a variant with node weights in Chapter 4. In Chapter 5 we describe and implement `et_trees`. `ed_trees` are described and implemented in Chapter 6. These two structures are derived form weighted balanced binary trees and they are the basis of the dynamic connectivity data structure. A detailed presentation of the algorithm is given in Chapter 7. It is followed by Chapter 8 on the implementation of the dynamic connectivity algorithm.

 The sources are available via anonymous `ftp` from `ftp.inf.fu_berlin.de:/pub/misc/dyn_con/dyn_con-1.0.tar.gz`.

Version[1]**M** $\equiv$

```
{// Version 1.0                                          //}
```

This macro is invoked in definitions 3, 4, 19, 21, 34, 41, 45, 49, and 55.

LEGAL NOTE[2]**M** $\equiv$

```
{// Copyright (C) 1995 by David Alberts                   //
 // FU Berlin, Inst. f. Informatik, Takustr. 9, 14195 Berlin, FRG //
 // All rights reserved.                                  //
 // There is ABSOLUTELY NO WARRANTY.                      //}
```

This macro is invoked in definitions 3, 4, 19, 21, 34, 41, 45, 49, and 55.

This document was prepared using `funnelweb 3.0AC` by R. Williams and A. B. Coates.

Chapter 2

Main Ideas of the Algorithm

Like all previous algorithms the algorithm by Henzinger and King also maintains a spanning forest. All trees in the spanning forest are maintained in a data structure which allows logarithmic updates and queries within the forest. All we have to do is to keep it spanning, so the crucial case is again the deletion of a forest edge. One main idea is to use random sampling among the edges incident to the tree T containing the forest edge e to be deleted in order to find a replacement edge fast. The goal is a polylogarithmic update time, so the number of sampled edges is polylogarithmic. However, the set of possible edges reconnecting the two parts of T, which is called the *candidate set of* e in the following, might only be a small fraction of all non-tree edges which are adjacent to T. In this case it is unlikely to find a replacement edge for e among the sampled edges. If there is no candidate among the sampled edges the algorithm checks all adjacent edges of T. Otherwise it would not be guaranteed to provide correct answers to the queries. Since there might be a lot of edges which are adjacent to T this could be an expensive operation, so it should be a low probability event. This is not yet true, since deleting all edges in a relatively small candidate set, reinserting them, deleting them again, and so on will almost surely produce many of those events.

The second main idea prevents this undesirable behavior. The algorithm maintains an edge decomposition of the current graph G into $O(\log n)$ edge disjoint subgraphs $G_i = (V, E_i)$. These subgraphs are hierarchically ordered. Each i corresponds to a *level*. For each level i there is a forest F_i of G_i such that the union $\cup_{i \leq k} F_i$ is a spanning forest of $\cup_{i \leq k} G_i$; in particular the union F of all F_i is a spanning forest of G. A *spanning tree on level* i is a tree in $\cup_{j \leq i} F_j$. The *weight* $w(T)$ of a spanning tree T at level i is the number non-tree edges in G_i with at least one endpoint in T. If T_1 and T_2 are the two trees resulting from the deletion of e, we sample edges adjacent to the tree with the smaller weight. If sampling is unsuccessful due to a candidate set which is non-empty but relatively small, then the two pieces of the tree which was split are reconnected on the next higher level using one candidate, and all other candidate edges are copied to that level. The idea is to have sparse cuts on high levels and dense cuts on low levels. Non-tree edges always belong to the lowest level where their endpoints are connected or a higher level, and we always start sampling at the level of the deleted tree edge. After moving the candidates one level up they are normally no longer a small fraction of all adjacent non-tree edges at the new level. If the candidate set on one level is empty, we try to sample on the next higher level. There is one more case to mention: if sampling was unsuccessful despite the fact that the candidate set was big enough, which means that we had bad luck, we do not move the candidates to the next level, since this event has a small probability and does not happen very frequently.

If there are only deletions of edges a bound of $O(\log n)$ for the number of levels is guaranteed. If there are also insertions there have to be periodical rebuilds of parts of the data structure – i.e., we have to move some edges down again – to achieve a logarithmic number of levels, too.

The spanning trees at each level are represented by a data structure called `et_tree` ("et" means Euler tour in this context). It allows joining and splitting, `find_root` queries, and – in conjunction with a secondary data structure for storing non-tree edges adjacent to a node at a certain level, called `ed_tree` – finding a random adjacent edge in expected logarithmic time.

Chapter 3

Randomized Balanced Binary Trees

3.1 Introduction

A *binary* tree is a rooted tree such that every node has at most two children. Each child is either a *left child* or a *right child*. Each node has at most one left and at most one right child. In the following we deal only with binary trees. A tree on n nodes is *balanced* if its height is $O(\log n)$. There is a canonical ordering on a tree. It is called the *In-order*. In this order a node is bigger than all the nodes in its left subtree and smaller than all the nodes in its right subtree. We often identify the In-order of a tree with the sequence of nodes of the tree in In-order.

A given binary tree can be balanced by a sequence of *rotations*. A rotation changes the parent-child relation of a constant number of nodes without destroying In-order, i.e., the In-order of the nodes in the tree resulting from a rotation is the same as the In-order of the initial tree (see Figure 3.1).

Normally, balanced binary trees are used in a dynamic setting. This means that we can *join* two trees and *split* a tree at a node. Joining two balanced binary trees T_1 and T_2 results in a balanced binary tree T with an In-order which is the concatenation of the In-orders for T_1 and T_2. The split operation is in some sense the dual operation of join, but there are two possibilities. If we split a balanced binary tree T at a node u then this results in two balanced binary trees T_1 and T_2, such that the concatenation of their In-orders is again the In-order of T. One possibility is that u is the last vertex in the In-order of T_1, the other possibility is that u is the first vertex in the In-order of T_2. This has to be specified as a parameter of the split operation. Using join and split we can also insert or delete a node in a balanced binary tree.

A lot of useful queries can already be formulated in this basic setting. We describe four of the more important ones. The first one is *find_root*. It takes a node of some balanced binary tree and returns the root of this tree. The second is *pred* which returns the predecessor of a node with respect to In-order, if it exists. The third one is *succ*, which returns the successor of a given node. The fourth one takes two nodes and returns true if the first is smaller than the second with respect to In-order.

The worst case running time of all of these operations is linear in the height of the involved tree(s), so it is $O(\log n)$ if they are balanced. There are several methods for keeping the balance of binary trees, e.g., [1, 9, 13]. For the implementation we decided to take the simplest one which was given by Aragon and Seidel [2]. It is easy to implement and a single operation has small constants hidden in the O notation. However, the bound is only on the expected time, *not* on the worst case time. The expectation is taken over the random choices of the algorithm so it holds for any input and update sequence.

The randomized scheme for balancing binary trees in [2] works by giving each node in the tree a random priority and by maintaining the *Heap-order* of the nodes. A tree is in Heap-order with respect to some priorities of its nodes if the children of each node have smaller priorities than their parent. While there are usually several trees with the same In-order, for given distinct priorities there is exactly one tree which additionally is in Heap-order. Aragon and Seidel have shown that this tree is balanced with high probability if the priorities are random. In the implementation we do not care about the distinctness of the priorities. This is no problem as long as the set of possible priorities is much larger then the size of the trees, which is a realistic assumption for our application.

3.2 Overview

We implement balanced binary trees with the randomized balancing scheme by Aragon and Seidel [2] as a class in `C++`. There are two files: the header file `rnb_tree.h` which declares the class and its methods, and the file `rnb_tree.c` which implements the methods.

Actually, we rather define a class containing the information of one tree node, called `rnb_node_struct`. A `rnb_node` is pointer to an instance of class `rnb_node_struct`. A randomized balanced binary tree eventually is a `rnb_node` with no parent. Note that we cannot express this in `C++`, so the types `rnb_node` and `rnb_tree` are the same and `rnb_tree` was only introduced to clarify some definitions.

At each node we store (pointers to) its parent and children and its priority for balancing. All methods will be described in detail later. There are two output methods, namely `print` and `traverse`, for testing.

rnb_tree.h[3] ≡

```
{// ------------------------------------------------------------ //
 // rnb_tree.h: header file for rnb_trees                        //
 //                                                              //
 // comment: rnb_tree is an implementation of balanced binary    //
 //          trees with a randomized balancing scheme.           //
 //          See also the documentation in dyn_con.ps.           //
 //                                                              //
 Version[1]
 //                                                              //
 LEGAL NOTE[2]
 // ------------------------------------------------------------ //

 // RCS ID //
 /* $Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $ */

 #ifndef RNB_TREE  // avoid multiple inclusion
 #define RNB_TREE

 #include<iostream.h>
 #include<stdlib.h>

 // we define left and right
 enum rnb_dir {rnb_left=0, rnb_right=1};

 // we define a null pointer
 #ifndef nil
 #define nil 0
 #endif

 // we define true and false
 #ifndef true
 #define true 1
 #endif

 #ifndef false
 #define false 0
 #endif

 class   rnb_node_struct;
 typedef rnb_node_struct* rnb_node;
 typedef rnb_node rnb_tree;

 class rnb_node_struct{

 public:

   rnb_node_struct() {par = child[0] = child[1] = nil; prio = random();}
   // construct a new tree containing just one element

   virtual ~rnb_node_struct() { isolate(); }
   // virtual destructor in order to deallocate the right amount of storage
   // even in derived classes

   rnb_node find_root();
```

```
  // returns the root of the tree containing this node.
  // Prec.: this != nil

  rnb_node sub_pred();
  // returns the predecessor of this node in the (sub)tree rooted at this node
  // or nil if it does not exist
  // Prec.: this != nil

  rnb_node sub_succ();
  // returns the successor of this node in the (sub)tree rooted at this node
  // or nil if it does not exist
  // Prec.: this != nil

  rnb_node pred();
  // returns the predecessor of this node or nil if it does not exist
  // Prec.: this != nil

  rnb_node succ();
  // returns the successor of this node or nil if it does not exist
  // Prec.: this != nil

  rnb_node cyclic_pred() { return (this == first()) ? last() : pred(); }
  // return the cyclic predecessor of this node (or nil)
  // Prec.: this != nil

  rnb_node cyclic_succ() { return (this == last()) ? first() : succ(); }
  // return the cyclic successor of this node (or nil)
  // Prec.: this != nil

  rnb_node first();
  // Return the first node in In-order in the tree rooted at this node.

  rnb_node last();
  // Return the last node of this tree.

  friend int smaller(rnb_node u, rnb_node v);
  // returns true iff u is smaller than v

  friend rnb_tree rnb_join(rnb_tree t1, rnb_tree t2, rnb_node dummy);
  // join t1 and t2 and return the resulting rnb_tree

  friend void split(rnb_node at, int where, rnb_tree& t1, rnb_tree& t2,
                    rnb_node dummy);
  // split the rnb_tree containing the node at before or after at
  // depending on where. If where == rnb_left we split before at,
  // else we split after at. The resulting trees are stored in t1 and t2.
  // If at == nil, we store nil in t1 and t2.

  virtual void print();
  // prints the contents of this node to stdout for testing

  friend void traverse(rnb_tree t);
  // traverses the tree t and prints each node to stdout for testing

protected:

  rnb_node  par;                // parent node
  rnb_node  child[2];           // children
  long      prio;               // priority for balancing

  friend void rotate(rnb_node rot_child, rnb_node rot_parent);
  // Rotate such that rot_child becomes the parent of rot_parent.
  // Prec.: rot_child is a child of rot_parent

  virtual void after_rot() { }
  // This method is called for rot_parent after each rotation in order
  // to fix additional information at the nodes in derived classes.

  virtual void init() { }
  // This method is used to initialize the dummy node in join and split
  // after linking it to the tree(s).
  // Prec.: this != nil

  virtual void isolate();
  // Make this node an isolated node.
  // Prec.: this != nil
};

#endif
}
```

This macro is attached to an output file.

rnb_tree.c[4] ≡

```
{// ---------------------------------------------------------------- //
```

```
// rnb_tree.c: implementation of rnb_trees                          //
//                                                                  //
// comment: rnb_tree is an implementation of balanced binary        //
//          trees with a randomized balancing scheme.               //
//          See also the documentation in dyn_con.ps.               //
//                                                                  //
```
Version[1]
```
//                                                                  //
```
LEGAL NOTE[2]
```
// ---------------------------------------------------------------- //

// RCS ID //
static char rcs[]="$Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $";

#include"rnb_tree.h"
```
rnb_tree methods[5]
}

This macro is attached to an output file.

3.3 Implementation

3.3.1 `rotate`

The method **`rotate`** takes two nodes. The first one has to be the child of the second one. The effect of this method is best illustrated by a picture.

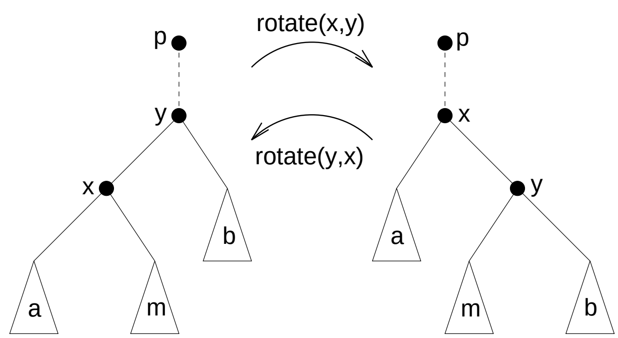

Figure 3.1: The Effect of a Rotation

There are two types of rotations depending on whether the first argument is the left child of the second argument or the right one. Let us look at `rotate(x,y)`. This is called a *right rotation*. The edges in the tree are directed since they are represented by parent and child pointers. The following edges disappear: `(m,x)`, `(x,y)`, `(y,p)`. The following edges are created: `(m,y)`, `(y,x)`, `(x,p)`. Note that the subtree `m` "changes sides". `rotate(y,x)` is a *left rotation*.

rnb_tree methods[5] $+\equiv$

{
```
inline void rotate(rnb_node  rot_child, rnb_node rot_parent)
// Rotate such that rot_child becomes the parent of rot_parent.
// Prec.: rot_child is a child of rot_parent.
{
  // determine the direction dir of the rotation
  int dir = (rot_parent->child[rnb_left] == rot_child) ? rnb_right : rnb_left;

   // subtree which changes sides
  rnb_tree middle = rot_child->child[dir];

  // fix middle tree
  rot_parent->child[1-dir] = middle;
```

```
    if(middle) middle->par = rot_parent;

    // fix parent field of rot_child
    rot_child->par = rot_parent->par;
    if(rot_child->par)
      if(rot_child->par->child[rnb_left] == rot_parent)
         rot_child->par->child[rnb_left]  = rot_child;
      else
         rot_child->par->child[rnb_right] = rot_child;

    // fix parent field of rot_parent
    rot_child->child[dir] = rot_parent;
    rot_parent->par = rot_child;

    // fix additional information in derived classes
    rot_parent->after_rot();
  }
  }
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.2 isolate

This method is called to safely isolate a node even in derived classes. It takes care of parent pointers of the children of the node and the appropriate child pointer of its parent. Derived classes may store additional information at the nodes. The additional information of other nodes in the tree may be affected by the removal of this node. Since this is a virtual function the appropriate steps can be added to the implementation in each derived class.

rnb_tree methods[6] $+\equiv$

```
{
  void rnb_node_struct::isolate()
  // Make this node an isolated node.
  // Prec.: this != nil
  {
    // adjust child pointer of parent if it exists
    if(par)
      if(par->child[rnb_left] == this)
        par->child[rnb_left] = nil;
      else
        par->child[rnb_right] = nil;

    // adjust parent pointers of children if they exist
    if(child[rnb_left]) child[rnb_left]->par = nil;
    if(child[rnb_right]) child[rnb_right]->par = nil;
  }
  }
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.3 find_root

We simply follow the parent pointers as long as possible.

rnb_tree methods[7] $+\equiv$

```
{
  rnb_tree rnb_node_struct::find_root()
  // returns the root of the tree containing this node.
  // Prec.: this != nil
  {
    for(rnb_node aux = this; aux->par; aux = aux->par);
    return aux;
  }
  }
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.4 sub_pred and sub_succ

The predecessor of a node u in the subtree rooted at u in In-order is the biggest node in its left subtree. If u has a left subtree we follow the chain of right children of the left child of u as long as possible. We end up at the desired node.

rnb_tree methods[8] + ≡

```
{
rnb_node rnb_node_struct::sub_pred()
// returns the predecessor of this node in the subtree rooted at this node
// or nil if it does not exist
// Prec.: this != nil
{
  // handle the nil case first
  if(!child[rnb_left]) return nil;

  // find the last node with no right child in the left subtree of u
  for(rnb_node aux = child[rnb_left]; aux->child[rnb_right];
              aux = aux->child[rnb_right]);
  return aux;
}
}
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

We do the same for the successor...

rnb_tree methods[9] + ≡

```
{
rnb_node rnb_node_struct::sub_succ()
// returns the successor of this node in the subtree rooted at this node
// or nil if it does not exist
// Prec.: this != nil
{
  // handle the nil case first
  if(!child[rnb_right]) return nil;

  // find the first node with no left child in the right subtree of u
  for(rnb_node aux = child[rnb_right]; aux->child[rnb_left];
              aux = aux->child[rnb_left]);
  return aux;
}
}
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.5 pred and succ

In contrast to the method sub_pred we now also may have to look at the ancestors of the given node u in order to find its predecessor. If there is no left subtree but there is a parent p of u such that u is the right child of p then p is the predecessor of u. If u is the left child of p, then we follow the chain of parents of p until we arrive at the root or at a node which is the right child of its parent q. In the former case there is no predecessor. In the latter case q is the predecessor.

rnb_tree methods[10] + ≡

```
{
rnb_node rnb_node_struct::pred()
// returns the predecessor of this node or nil if it does not exist
// Prec.: this != nil
{
  // search for predecessor in the subtree of this node first
  rnb_node sub_pr = sub_pred();
  // if it exists we can return it
  if(sub_pr) return sub_pr;

  // otherwise we have to look for the ancestors of this node
  if(par)       // if there is a parent
    if(this == par->child[rnb_right])
```

```
    // this is a right child
      return par;
    else
    // this is a left child
    {
      for(rnb_node aux = par; aux->par; aux = aux->par)
        if(aux == aux->par->child[rnb_right]) return aux->par;
    }

  // there is no predecessor
  return nil;
}
}
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

rnb_tree methods[11] $+\equiv$

```
{
rnb_node rnb_node_struct::succ()
// returns the successor of this node or nil if it does not exist
// Prec.: this != nil
{
  // search for successor in the subtree of this node first
  rnb_node sub_s = sub_succ();
  // if it exists we can return it
  if(sub_s) return sub_s;

  // otherwise we have to look for the ancestors of this node
  if(par)        // if there is a parent of u
    if(this == par->child[rnb_left])
    // this node is a left child
      return par;
    else
    // this node is a right child
    {
      for(rnb_node aux = par; aux->par; aux = aux->par)
        if(aux == aux->par->child[rnb_left]) return aux->par;
    }

  // there is no predecessor
  return nil;
}
}
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.6 `first` and `last`

We simply walk as long to the left as possible starting at this node.

rnb_tree methods[12] $+\equiv$

```
{
rnb_node rnb_node_struct::first()
// Return the first node in In-order in the tree rooted at this node.
{
  // remember one node before current node
  rnb_node last = nil;
  for(rnb_node current = this; current; current = current->child[rnb_left])
    last = current;

  return last;
}
}
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

We simply walk as long to the right as possible starting at this node.

rnb_tree methods[13] $+\equiv$

```
{
rnb_node rnb_node_struct::last()
// Return the last node of this tree.
{
```

```
    // remember one node before current node
    rnb_node last = nil;
    for(rnb_node current = this; current; current = current->child[rnb_right])
      last = current;

    return last;
  }
  }
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.7 smaller

The method `smaller` takes two nodes `u` and `v` as its arguments and returns true iff the first node is smaller than the second one with respect to In-order. If `u` and `v` are incomparable, i.e., in different trees, we return false.

We first check whether `u` and `v` have the same root. If this is the case then the path from the root to `u` and the path from the root to `v` are recorded as sequences of left and right moves in two arrays. We decide whether `u` is smaller than `v` by comparing these sequences.

rnb_tree methods[14] + ≡

```
  {
  int smaller(rnb_node u, rnb_node v)
  // returns true iff u is smaller than v
  {
    if(!u || !v) return false;
    if(u == v) return false;

    // determine the height of u and v
    rnb_node aux_u = u;
    for(int u_height = 0; aux_u->par; aux_u = aux_u->par, u_height++);
    rnb_node aux_v = v;
    for(int v_height = 0; aux_v->par; aux_v = aux_v->par, v_height++);

    // if u and v have different roots they are incomparable and we return false
    if(aux_u != aux_v) return false;

    // we represent the paths from u and v to their roots by arrays
    // create arrays
    int *u_path = new int[u_height];
    int *v_path = new int[v_height];

    // insert left and right moves
    int u_i = u_height - 1;
    for(aux_u = u; aux_u->par; aux_u = aux_u->par, u_i--)
    {
      if(aux_u->par->child[rnb_left] == aux_u) u_path[u_i] = rnb_left;
      else                                     u_path[u_i] = rnb_right;
    }

    int v_i = v_height - 1;
    for(aux_v = v; aux_v->par; aux_v = aux_v->par, v_i--)
    {
      if(aux_v->par->child[rnb_left] == aux_v) v_path[v_i] = rnb_left;
      else                                     v_path[v_i] = rnb_right;
    }

    // compare the paths
    // skip identical prefix
    for(int i = 0; ((i<u_height) && (i<v_height)) && (u_path[i] == v_path[i]);
            i++);

    // at least one path is not completely scanned because u!=v
    // but u->find_root() == v->find_root()
    // at i they are different
    int result;
    if( (i<u_height) && (u_path[i] == rnb_left) )
      result = true;
    else
      if( (i<v_height) && (v_path[i] == rnb_right) )
        result = true;
      else
        result = false;

    // delete the paths
```

```
    delete[] u_path;
    delete[] v_path;

    return result;
  }
}
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.8 rnb_join

We implement the **rnb_join** method by creating a dummy node. Its children become the two trees to be joined. Note that the resulting tree already has the right In-order. We only have to remove the dummy node. The dummy node is trickled down the tree using rotations that do not change the In-order. We only have to take care of the additional Heap-order. This can be done by always choosing the child with biggest priority for the next rotation. When the dummy node eventually is a leaf of the tree we can simply remove it.

rnb_tree methods[15] $+\equiv$

```
{
  rnb_tree rnb_join(rnb_tree t1, rnb_tree t2, rnb_node dummy)
  // join t1 and t2 and return the resulting rnb_tree
  {
    // handle the trivial t1 == nil || t2 == nil case
    if(!t1 || !t2)
    {
      if(t1) return t1;
      if(t2) return t2;
      return nil;
    }

    dummy->par = nil;
    dummy->child[rnb_left] = t1;
    dummy->child[rnb_right] = t2;
    t1->par = dummy;
    t2->par = dummy;
    // fix additional information in derived classes
    dummy->init();

    // trickle dummy down
    while( (dummy->child[rnb_left]) || (dummy->child[rnb_right]) )
    // while there is at least one child
    {
      // rotate with child with biggest priority

      // find child with biggest priority...
      rnb_node bigger = dummy->child[rnb_left];
      if(dummy->child[rnb_right])
      {
        if(dummy->child[rnb_left])
        {
          if(dummy->child[rnb_right]->prio > dummy->child[rnb_left]->prio)
            bigger = dummy->child[rnb_right];
        }
        else bigger = dummy->child[rnb_right];
      }

      // ...and rotate with it
      rotate(bigger,dummy);
    }

    // disconnect dummy from the new tree
    dummy->isolate();

    // return root of the new tree
    if(t2->par) return t1;
    else        return t2;
  }
}
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.9 split

The effect of the operation split(at,where,t1,t2) is the following. The tree T containing at is split into two trees which are assigned to t1 and t2. t1 contains the prefix of the In-order of T up to but including or excluding at. t2 contains the rest of the In-order of T. Whether t1 contains at depends on where. If where has a value of rnb_left then the T is split left of at, so t1 does not contain at. If where is set to rnb_right then T is split right of at.

This is implemented by inserting a dummy node dummy immediately before or after at with respect to In-order. This node is rotated up until it becomes the root of T. Then the left subtree of dummy is t1 and the right subtree of dummy is t2.

rnb_tree methods[16] + ≡

```
{
 void split(rnb_node at, int where, rnb_tree& t1, rnb_tree& t2, rnb_node dummy)
 // split the rnb_tree containing the node at before or after at
 // depending on where. If where == rnb_left we split before at,
 // else we split after at. The resulting trees are stored in t1 and t2.
 // If at == nil, we store nil in t1 and t2.
 {
   // handle the trivial at == nil case first
   if(!at)
   {
     t1 = nil;
     t2 = nil;
     return;
   }

   dummy->child[rnb_left] = nil;
   dummy->child[rnb_right] = nil;

   // insert dummy in the right place (w.r.t. In-order)
   // where == rnb_left => split before at
   // where != rnb_left => split after at
   if(where != rnb_left) // split after at
   {
     // store dummy as left child of the subtree successor of at
     // or as right child of at if there is no subtree successor
     rnb_node s = at->sub_succ();
     if(!s)
     {
       at->child[rnb_right] = dummy;
       dummy->par = at;
     }
     else
     {
       s->child[rnb_left] = dummy;
       dummy->par = s;
     }
   }
   else     // split before at
   {
     // store dummy as right child of the subtree predecessor of at
     // or as left child of at if there is no subtree predecessor
     rnb_node p = at->sub_pred();
     if(!p)
     {
       at->child[rnb_left] = dummy;
       dummy->par = at;
     }
     else
     {
       p->child[rnb_right] = dummy;
       dummy->par = p;
     }
   }
   // fix additional information in derived classes
   dummy->init();


   // rotate dummy up until it becomes the root
   for(rnb_node u = dummy->par; u; u = dummy->par) rotate(dummy,u);

   // store the subtrees of dummy in t1 and t2
   t1 = dummy->child[rnb_left];
   t2 = dummy->child[rnb_right];

   // disconnect dummy
```

```
    dummy->isolate();
  }
  }
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

3.3.10 print and traverse

rnb_tree methods[17] + ≡

```
{
  void rnb_node_struct::print()
  // prints the contents of this node to stdout for testing
  // Prec.: this != nil
  {
    cout << "node at " << this << ":\n";
    cout << "  parent:      " << par << "\n";
    cout << "  left child:  " << child[rnb_left] << "\n";
    cout << "  right child: " << child[rnb_right] << "\n";
    cout << "  priority:    " << prio << "\n";
  }
  }
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

rnb_tree methods[18] + ≡

```
{
  void traverse(rnb_tree t)
  // traverses the tree and outputs each node to stdout for testing
  {
    if(t)
    {
      t->print();
      traverse(t->child[rnb_left]);
      traverse(t->child[rnb_right]);
    }
  }
  }
```

This macro is defined in definitions 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, and 18.
This macro is invoked in definition 4.

This method traverses the given tree in Pre-order, i.e., root first, then left subtree, then right subtree. It `prints` each encountered node.

Chapter 4

Randomized Balanced Binary Trees with Node Weights

In this chapter we describe a randomized balanced tree as above with additional non-negative integer weights at the nodes. The sum of the weights in the subtree rooted at a certain node is maintained and it is possible to locate the node which "represents" a certain integer in the "weight range" of such a tree.

In addition to the methods implemented for randomized balanced binary trees the following methods are supported by the weighted version.

- A node weight or the weight of the subtree rooted at a certain node is returned in constant time.
- A node weight can be updated in logarithmic expected time.
- For a node u let $T(u)$ be the subtree rooted at u. Let $weight(u)$ be the weight associated to the node u, and let $weight(T)$ be the sum of weights of nodes in T. Given a node u and a weight w with $0 < w \leq weight(T(u))$, a node v with the following properties is returned by the **`rnbw_locate`** function.
  - v is a member of $T(u)$.
  - Let $\{u_1, \ldots, u_k, u_{k+1} = v, u_{k+2}, \ldots, u_r\}$ be the In-order of $T(u)$. Then the following holds.

$$\sum_{i=1}^{k} weight(u_i) < w \leq \sum_{i=1}^{k+1} weight(u_i)$$

We have to augment the implementation of the **`rotate`** method for randomized balanced binary trees in order to maintain the sums of weights.

4.1 Overview

There are two files: the header file `rnbw_tree.h` which declares the class and its methods, and the file `rnbw_tree.c` which implements the methods.

The class `rnbw_node_struct` is derived from the base class `rnb_node_struct`. The weight of a node is stored in its **`weight`** field. There is an additional field **`sub_weight`** which stores the weight of the subtree rooted at this node. The **`weight`** field is not really necessary, but it simplifies the implementation of some methods.

There are methods to get or set the weight of a node and to get the weight of a subtree. There is a function which locates the node which represents a certain weight as described in the Introduction. In order to handle the **`sub_weight`** field correctly some virtual functions of the base class have to be redefined.

In order to get return values of the right type some conversion functions are defined.

rnbw_tree.h[19] ≡

```
{// ------------------------------------------------------------ //
 // rnbw_tree.h: header file for the rnbw_trees                   //
 //                                                               //
 // comment: rnbw_trees are derived from rnb_trees. They have an  //
 //          additional non-negative weight at each node and      //
 //          subtree weights.                                     //
 //          See also the documentation in dyn_con.ps.            //
 //                                                               //
 Version[1]
 //                                                               //
 LEGAL NOTE[2]
 // ------------------------------------------------------------ //

 // RCS ID //
 /* $Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $ */

 #ifndef RNBW_TREE
 #define RNBW_TREE

 #include"rnb_tree.h"

 class rnbw_node_struct;
 typedef rnbw_node_struct* rnbw_node;
 typedef rnbw_node rnbw_tree;


 class rnbw_node_struct : public rnb_node_struct {

 public:

   rnbw_node_struct(int w = 1) : rnb_node_struct() { weight = sub_weight = w; }
   // construct a new tree containing just one node with weight w.
   // By default each node gets a weight of one.
   // Prec.: w >= 0

   int get_weight() { return weight; }
   // returns the weight of this node

   int get_subtree_weight() { return sub_weight; }
   // returns the weight of the subtree rooted at this node

   void set_weight(int w);
   // sets the weight of this node to w
   // Prec.: w >= 0

   void add_weight(int a);
   // adds a to the weight of this node
   // Prec.: a >= -(weight of this node)

   friend rnbw_node rnbw_locate(rnbw_tree t, int w, int& offset);
   // returns the node in the tree rooted at t which corresponds to
   // w with respect to In-order.
   // Prec.: 0 < w <= weight of tree rooted at t

   Conversion Functions for rnbw_trees[20]

   virtual void print();
   // we redefine print in order to output also the additional fields

 protected:

   void after_rot();
   // We fix the weight fields after a rotation. This is called as
   // a virtual function in the base class rnb_tree.

   void init();
   // This method is used to initialize the dummy node in join and split
   // after linking it to the tree(s). It is a virtual function in the base
   // class.

   virtual void isolate();
   // We fix the sub_weight fields of the ancestors of this node.

 private:

   int weight;      // stores the weight of this node
   int sub_weight;  // stores the weight of the subtree rooted at this node
 };

 #endif
 }
```

This macro is attached to an output file.

Using the methods from the base class as they are can sometimes lead to type conflicts. A cast of a pointer to an instance of a derived class to a pointer to an instance of its base class is done automatically by the compiler whereas a cast in the other direction has to be explicit.

As an example for the undesirable consequences of this problem consider the following example. If we simply `join` to `rnbw_trees` using the `rnb_join` method from the base class `rnb_node_struct` the return value is a `rnb_tree` and not a `rnbw_tree`. So we cannot assign the resulting tree as it is to another `rnbw_tree`. There has to be an explicit cast. However, there are also methods which do not pose this problem like, e.g., the `smaller` method.

In order to provide a consistent interface to the class `rnbw_node_struct` we define new methods corresponding to the critical ones in the base class which have the desired type. We also define methods returning the parent and the children of a node with the correct type.

Conversion Functions for rnbw_trees[20] ≡

```
{// --- Conversion Functions ---
 rnbw_node parent() { return (rnbw_node) par; }
 rnbw_node left_child() { return (rnbw_node) child[rnb_left]; }
 rnbw_node right_child() { return (rnbw_node) child[rnb_right]; }
 rnbw_tree find_root() {return (rnbw_tree) rnb_node_struct::find_root(); }
 rnbw_node pred() { return (rnbw_node) rnb_node_struct::pred(); }
 rnbw_node succ() { return (rnbw_node) rnb_node_struct::succ(); }
 friend inline rnbw_tree rnbw_join(rnbw_tree t1, rnbw_tree t2, rnbw_node dummy)

 { return (rnbw_tree) rnb_join(t1,t2,dummy); }}
```

This macro is invoked in definition 19.

rnbw_tree.c[21] ≡

```
{// ------------------------------------------------------------ //
 // rnbw_tree.c: implements rnbw_trees                           //
 //                                                              //
 // comment: rnbw_trees are derived from rnb_trees. They have an //
 //          additional non-negative weight at each node and     //
 //          subtree weights.                                    //
 //          See also the documentation in dyn_con.ps.           //
 //                                                              //
 Version[1]
 //                                                              //
 LEGAL NOTE[2]
 // ------------------------------------------------------------ //

 // RCS ID //
 static char rcs[]="$Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $";

 #include"rnbw_tree.h"

 rnbw_tree methods[22]
 }
```

This macro is attached to an output file.

4.2 Implementation

4.2.1 `after_rot`, `init`, and `isolate`

rnbw_tree methods[22] + ≡

```
{
 void rnbw_node_struct::after_rot()
 {
   // the parent gets the sub_weight of this node
   parent()->sub_weight = sub_weight;

   // recalculate the sub_weight field of this node
   sub_weight = weight;
   if(left_child()) sub_weight += left_child()->sub_weight;
   if(right_child()) sub_weight += right_child()->sub_weight;
 }
 }
```

This macro is defined in definitions 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 21.

A rotation affects the subtree weights so we have to update some `sub_weight` fields, namely those of this node and its parent, since this method is invoked for the node who was the parent of its new parent after the rotation.

rnbw_tree methods[23] + ≡
```
{
 void rnbw_node_struct::init()
 {
   // initialize the sub_weight field of this node
   sub_weight = weight;
   if(left_child()) sub_weight += left_child()->sub_weight;
   if(right_child()) sub_weight += right_child()->sub_weight;
 }
}
```
This macro is defined in definitions 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 21.

In the `join` and `split` methods of the base class `rnb_node_struct` a dummy node is created and linked to the tree(s). Its `sub_weight` field has to be initialized at least for `join`.

rnbw_tree methods[24] + ≡
```
{
 void rnbw_node_struct::isolate()
 // We fix the sub_weight fields of the ancestors of this node.
 {
   // fix sub_weight fields
   for(rnbw_node aux = parent(); aux; aux = aux->parent())
     aux->sub_weight -= sub_weight;

   // fix base class
   rnb_node_struct::isolate();
 }
}
```
This macro is defined in definitions 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 21.

In addition to fixing parent and child pointers pointing to this node which is done in the the base class `rnb_node_struct` we also have to fix the `sub_weight` fields of the ancestors if they exist.

4.2.2 `set_weight` and `add_weight`

If the `weight` field of a node is changed this has an influence on its `sub_weight` field and the `sub_weight` fields of its ancestors.

rnbw_tree methods[25] + ≡
```
{
 void rnbw_node_struct::set_weight(int w)
 // sets the weight of this node to w
 // Prec.: w >= 0
 {
   // remember the difference between the new and the old weight
   int w_diff = w - weight;

   // update the weight and subweight fields of this node
   sub_weight += w_diff;
   weight = w;

   // update the sub_weight fields of the ancestors of this node
   for(rnbw_node aux = parent(); aux; aux = aux->parent())
     aux->sub_weight += w_diff;
 }
}
```
This macro is defined in definitions 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 21.

rnbw_tree methods[26] $+\equiv$

```
{
 void rnbw_node_struct::add_weight(int a)
 // adds a to the weight of this node
 // Prec.: a >= -(weight of this node)
 {
   // update the weight and subweight fields of this node
   sub_weight += a;
   weight += a;

   // update the sub_weight fields of the ancestors of this node
   for(rnbw_node aux = parent(); aux; aux = aux->parent())
     aux->sub_weight += a;
 }
 }
```

This macro is defined in definitions 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 21.

4.2.3 `rnbw_locate`

See the introduction for a description of the effect of this method. It is implemented by following a way down in the tree guided by two values `lower` and `upper`. In `lower` we maintain the sum of weights of nodes in In-order up to but excluding the current node. In `upper` we maintain `lower` plus the weight of the current node. The interval of weights represented by the current node is $(lower, upper]$. Depending on whether the given weight `w` is left or right of this interval for the current node, we proceed with the left or right child. Eventually `w`$\in (lower, upper]$. We return the corresponding `rnbw_node` and store `w`$-lower$ in `offset`.

rnbw_tree methods[27] $+\equiv$

```
{
 rnbw_node rnbw_locate(rnbw_tree t, int w, int& offset)
 // returns the node in the tree rooted at t which corresponds to
 // w with respect to In-order.
 // Prec.: 0 < w <= weight of tree rooted at t
 {
   // current node
   rnbw_node curr_node = t;
   // sum of weights up to but excluding current node
   int lower = curr_node->left_child() ?
               curr_node->left_child()->sub_weight : 0;
   // sum of weights up to and including current node
   int upper  = lower + curr_node->weight;

   while(w <= lower || w > upper)
   // weight w not represented at current node
   // so we have to proceed at a child of the current node
   {
     if(w <= lower)
     // proceed at left child
     {
       curr_node = curr_node->left_child();
       // update lower
       lower -= curr_node->sub_weight;
       if(curr_node->left_child())
         lower += curr_node->left_child()->sub_weight;
       // update upper
       upper = lower + curr_node->weight;
     }
     else
     // proceed at right child
     {
       curr_node = curr_node->right_child();
       // update lower
       lower = upper + curr_node->sub_weight - curr_node->weight;
       if(curr_node->right_child())
         lower -= curr_node->right_child()->sub_weight;
       // update upper
       upper = lower + curr_node->weight;
     }
   }

   // store offset of w from lower
   offset = w - lower;

   // return the node representing w
```

```
    return curr_node;
  }
  }
```

This macro is defined in definitions 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 21.

4.2.4 print

rnbw_tree methods[28] $+ \equiv$

```
{
 void rnbw_node_struct::print()
 // we redefine print in order to output also the additional fields
 {
   // output base fields
   rnb_node_struct::print();

   // output new fields
   cout << "  weight:      " << weight << "\n";
   cout << "  sub_weight:  " << sub_weight << "\n";
 }
 }
```

This macro is defined in definitions 22, 23, 24, 25, 26, 27, and 28.
This macro is invoked in definition 21.

Chapter 5

Euler Tour Trees

5.1 Introduction

In the dynamic connectivity algorithm various spanning trees are maintained. They are represented implicitly by some encoding. The encoding of a spanning tree T at level i is a sequence $ET(T)$ of the vertices of T in the order in which they are encountered during a traversal of T starting at an arbitrarily selected vertex r, the root of T. The traversal is an Euler tour of a modified T where each edge is doubled. Thus, each node v occurs exactly $d(v)$ times except for the root which appears $d(r)+1$ times. In total the sequence has $2k-1$ occurrences where k is the number of nodes in T.

The tree T is subject to changes. It may be split by removing an edge or joined with another tree by inserting an edge. We quote the description of the counterparts of these operations for Euler tour sequences from [10].

Procedures for modifying encodings

1. **To delete edge $\{a,b\}$ from T:** Let T_1 and T_2 be the two trees which result, where $a \in T_1$ and $b \in T_2$. Let $o_{a_1}, o_{a_2}, o_{b_1}, o_{b_2}$ represent the occurrences encountered in the two traversals of $\{a,b\}$. If $o_{a_1} < o_{b_1}$ and $o_{b_1} < o_{b_2}$ then $o_{a_1} < o_{b_1} < o_{b_2} < o_{a_2}$. Thus $ET(T_2)$ is given by the interval of $ET(T)$ $o_{b_1}, \ldots, o_{b_2}$ and $ET(T_2)$ is given by splicing out of $ET(T)$ the sequence $o_{b_1}, \ldots, o_{a_2}$.
2. **To change the root of T from r to s:** Let o_s denote any occurrence of s. Splice out the first part of the sequence ending with the occurrence before o_s, remove its first occurrence o_r, and tack this on to the end of the sequence which now begins with o_s. Add a new occurrence o_s to the end.
3. **To join two rooted trees T and T' by edge e:** Let $e = \{a,b\}$ with $a \in T$ and $b \in T'$. Given any occurrences o_a and o_b, reroot T' at b, create a new occurrence o_{a_n} and splice the sequence $ET(T')o_{a_n}$ into $ET(T)$ immediately after o_a.

See Figure 5.1 and Figure 5.1.

We maintain the Euler tour sequence $ET(T)$ of a tree T at level i implicitly by storing the occurrences at the nodes of a balanced binary tree which is called the `et_tree` for T. `et_trees` are derived from `rnbw_trees`, randomized balanced binary trees with node weights. For each node v of T one of the occurrences of v in $ET(T)$ is arbitrarily selected to be the *active occurrence* of v. The active occurrence of v gets a node weight equal to the number of non-tree edges adjacent to v at level i. Nodes which are not active get a weight of zero. At each node of an `et_tree` the weight of its subtree is maintained. In particular we can determine the weight of a tree by looking at the subtree weight of the root of its corresponding `et_tree`. At each node v of G we store an array `act_occ` of pointers to its active occurrences at each level.

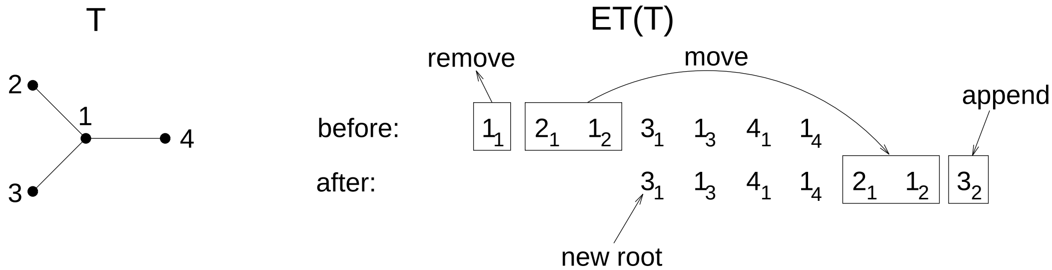

Figure 5.1: The Effect of Changing the Root

This figure illustrates the effect of changing the root of `T` from `1` to `3` on the Euler tour sequence of `T`.

Each tree edge e on level i is represented by three or four occurrences of its nodes in some `et_tree et` at level i corresponding to its traversal in the Euler tour sequence represented by `et` (see Figure 5.1). There are only three occurrences in the case that one of the nodes of e is a leaf. At each tree edge e there is an array `tree_occ` of pointers to four occurrences per level. Consider level i. If e belongs to a higher level, i.e., $e \in G_j$ for $j > i$ then these pointers are all `nil`. Otherwise those three or four occurrences representing e at level i are stored there, such that `tree_occ[i][0]` and `tree_occ[i][1]` belong to one node of e and `tree_occ[i][2]` and `tree_occ[i][3]` belong to the other node. One of these four occurrences may be `nil`.

The modification of T by inserting an edge connecting it to another tree or by deleting one of its edges may result in the deletion of an occurrence of $ET(T)$. This means that we have to check whether a deleted occurrence o was the active occurrence of its node v. If this is the case, we pass the activity to another occurrence of v at the same level. The modification of the Euler tour sequence in general has an influence on the representation of some tree edges. In order to do the necessary updates we keep at each occurrence o pointers to the edges represented by o and its left and right neighbor, respectively. These issues are described in more detail below.

5.2 Interface

In this section we present the interface of the data structure. It is part of the header file `dyn_con.h`.

Declaration of et_trees[29] $\equiv$

```
{// ------------ et_trees ----------------- //
 #include"rnbw_tree.h"

 // some forward definitions
 class dyn_con;

 class   et_node_struct;
 typedef et_node_struct* et_node;
 typedef et_node et_tree;
```

Declaration of et_node_struct[30]}

This macro is invoked in definition 49.

We begin by defining the nodes of the `et_tree`. They are pointers to structures containing the information stored at the node. An `et_tree` is just an `et_node` with no parent.

There are five new fields in an `et_node_struct ens` with respect to a `rnbw_node_struct`. There is a pointer to the instance of class `dyn_con` to which the node represented by `ens` belongs. It is used to access global information. There is the `corr_node` field which points to the node u such that this `et_node_struct` is an occurrence of u. The level of the tree T represented by the `et_tree ens` belongs to is stored at `level`. There is the `active` field which is `true` if `ens` is an active occurrence and `false` otherwise. The field `edge_occ[0]`

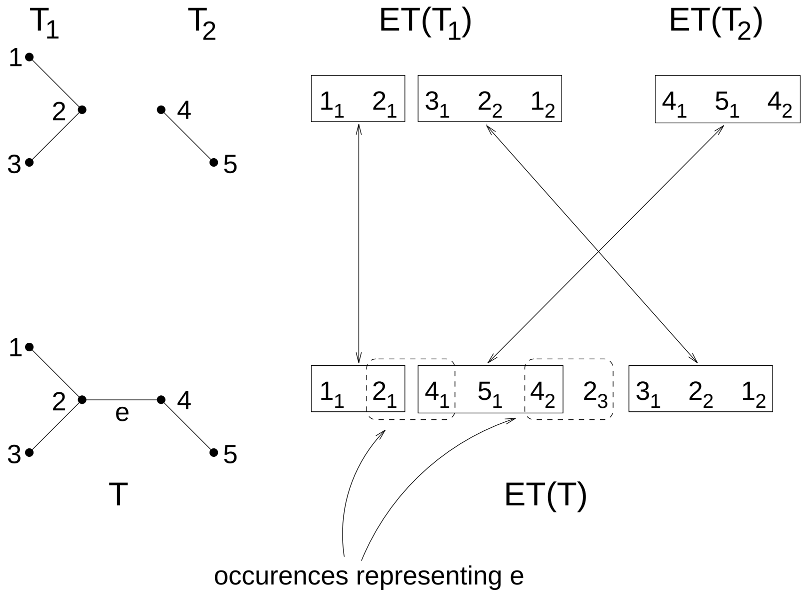

Figure 5.2: The Effects of Inserting and Deleting a Tree Edge

This figure illustrates the effect of inserting a new tree edge between 2 and 4. For convenience T_2 is already rooted at 4. Otherwise there had to be a `change_root` operation first. This figure also illustrates the effect of deleting the edge `e` in `T` when viewed from bottom to top.

contains the edge represented in part by the predecessor of `ens` and `ens` itself in $ET(T)$. We call it the *left edge* of `ens`. Similarly, `edge_occ[1]` contains the right edge of `ens`.

Declaration of et_node_struct[30] ≡

```
{
class et_node_struct : public rnbw_node_struct {

public:

  Interface of et_trees[31]

protected:

  dyn_con*  dc;          // the dynamic connectivity data structure this node
                         // belongs to
  node      corr_node;   // corresponding node in G
  int       level;       // the level of this node
  int       active;      // true iff active occurrence
  edge      edge_occ[2]; // the at most two tree edges represented
                         // also by this node. ordered left and right

  Declaration of Protected Operations on et_trees[33]
};
}
```

This macro is invoked in definition 29.

Interface of et_trees[31] ≡

```
{// Constructors
et_node_struct(dyn_con* dcp, node v, int my_level = -1, int activate = false);
// Create a new et_node_struct at level my_level for v.
// Activate it if activate is true.

et_node_struct(et_node en);
// Create a new et_node_struct which is an inactive copy of en.

Conversion Functions for et_trees[32]
```

```
node get_corr_node() { return corr_node; }
// This et_node is an occurrence of the returned graph node.

int is_active() { return active; }
// true <=> active occ.

friend et_tree et_link(node u, node v, edge e, int i, dyn_con* dc);
// Modify the et_trees of dc at level i corresponding to the insertion of
// the edge (u,v) into F_i.
// Prec.: u and v belong to dc, and they are not connected at the
//        valid level i.

friend void et_cut(edge e, int i, dyn_con* dc);
// Update the et_trees at level i corresponding to the removal of
// the tree edge e.
// Prec.: e actually is a tree edge at level i.

void print();
// Print this node to cout for testing.}
```

This macro is invoked in definition 30.

Conversion Functions for et_trees[32] ≡

```
{// --- Conversion Functions ---
et_node parent() { return (et_node) rnbw_node_struct::parent(); }
et_node left_child() { return (et_node) child[rnb_left]; }
et_node right_child() { return (et_node) child[rnb_right]; }
et_tree find_root() { return (et_tree) rnb_node_struct::find_root(); }
et_node first() { return (et_node) rnb_node_struct::first(); }
et_node last() { return (et_node) rnb_node_struct::last(); }
et_node cyclic_succ() { return (et_node) rnb_node_struct::cyclic_succ(); }
et_node cyclic_pred() { return (et_node) rnb_node_struct::cyclic_pred(); }
friend inline et_tree et_join(et_tree t1, et_tree t2, et_node dummy)
{ return (et_tree) rnb_join(t1,t2,dummy); }
friend inline et_node et_locate(et_tree et, int w, int& offset)
{ return (et_node) rnbw_locate(et,w,offset); }}
```

This macro is invoked in definition 31.

Declaration of Protected Operations on et_trees[33] ≡

```
{void pass_activity(et_node to);
// Make this node inactive and pass its activity to to.
// Prec.: This node is active, to represents the same vertex and is on the
//        same level.

friend void change_root(et_tree& et, et_node en, int i, dyn_con* dc);
// Change the root of the tree T represented by the et_tree et to the
// vertex represented by the et_node en. The new tree is stored at et.
// Prec.: The et_node en is in the et_tree et.}
```

This macro is invoked in definition 30.

5.3 Implementation

The following macro contains the file `et_tree.c` which implements the `et_tree` procedures. In the following subsections we discuss and implement these procedures.

et_tree.c[34] ≡

```
{// ------------------------------------------------------------- //
// et_tree.c: implementation of et_trees.                         //
//                                                                //
Version[1]
//                                                                //
LEGAL NOTE[2]
// ------------------------------------------------------------- //

// RCS Id //
static char rcs[]="$Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $";

#include"dyn_con.h"

Operations on et_trees[35]
}
```

This macro is attached to an output file.

5.3.1 Constructors

There are two constructors. One is for creating a new node from scratch. The other one is for cloning an existing node. In the latter case the copy is always inactive.

Operations on et_trees[35] $+\equiv$

```
{
 et_node_struct::et_node_struct
 (dyn_con* dcp, node v, int my_level, int activate)
 : rnbw_node_struct(0)
 {
   dc = dcp;
   corr_node = v;
   level = my_level;
   active = activate;
   if(activate) dc->Gp->inf(v)->act_occ[level] = this;
   edge_occ[0] = edge_occ[1] = nil;
 }

 et_node_struct::et_node_struct(et_node en)
 : rnbw_node_struct(0)
 {
   dc = en->dc;
   corr_node = en->corr_node;
   level = en->level;
   active = false;
   edge_occ[0] = edge_occ[1] = nil;
 }
 }
```

This macro is defined in definitions 35, 36, 37, 38, 39, and 40.
This macro is invoked in definition 34.

5.3.2 `pass_activity`

Operations on et_trees[36] $+\equiv$

```
{
 void et_node_struct::pass_activity(et_node to)
 // Make this node inactive and pass its activity to to.
 // Prec.: this node is active, to represents the same vertex and is on the
 //        same level.
 {
   active = false;
   to->active = true;
   to->set_weight(get_weight());
   set_weight(0);
   dc->Gp->inf(corr_node)->act_occ[level] = to;
 }
 }
```

This macro is defined in definitions 35, 36, 37, 38, 39, and 40.
This macro is invoked in definition 34.

5.3.3 `change_root`

We implement the procedure described in the Introduction of this chapter. The main difficulty lies in updating the `tree_occs` of up to two edges.

Operations on et_trees[37] $+\equiv$

```
{
 void change_root(et_tree& et, et_node en, int i, dyn_con* dc)
 // Change the root of the tree T represented by the et_tree et to the
 // vertex represented by the et_node en. The new tree is stored at et.
 // Prec.: The et_node en is in the et_tree et.
 {
   et_node first_nd = et->first();
```

```
    // if en is already the first node do nothing
    if(en == first_nd) return;

    // create a new occurrence for the new root
    et_node new_occ = new et_node_struct(en);

    // --- update active occurrences --- //
    et_node last_nd = et->last();
    // if the first node is active, pass activity to last node, since
    // the first node will be deleted
    if(first_nd->active) first_nd->pass_activity(last_nd);

    // --- update tree_occs --- //
    if(en->edge_occ[rnb_left] == en->edge_occ[rnb_right])
    {
      // replace the nil pointer in tree_occs of en->edge_occ[rnb_left]
      // by the new occurrence
      for(int k=0; nil != dc->Gp->inf(en->edge_occ[rnb_left])->tree_occ[i][k];
          k++);
      dc->Gp->inf(en->edge_occ[rnb_left])->tree_occ[i][k] = new_occ;
    }
    else
    {
      // replace en by the new occurrence
      for(int k=0; en != dc->Gp->inf(en->edge_occ[rnb_left])->tree_occ[i][k];
          k++);
      dc->Gp->inf(en->edge_occ[rnb_left])->tree_occ[i][k] = new_occ;
    }

    edge first_edge = first_nd->edge_occ[rnb_right];
    if((first_edge != last_nd->edge_occ[rnb_left]) ||
       (en == last_nd))
    {
      // replace first_nd by last_nd in the tree_occs of first_edge
      for(int k=0; first_nd != dc->Gp->inf(first_edge)->tree_occ[i][k];
          k++);
      dc->Gp->inf(first_edge)->tree_occ[i][k] = last_nd;
    }
    else
    {
      // replace first_nd by nil in the tree_occs of first_edge
      for(int k=0; first_nd != dc->Gp->inf(first_edge)->tree_occ[i][k];
          k++);
      dc->Gp->inf(first_edge)->tree_occ[i][k] = nil;
    }

    // --- update edge_occs --- //
    // right edge of first_nd becomes right edge of last node
    last_nd->edge_occ[rnb_right] = first_edge;

    // left edge of en becomes left edge of new_occ
    new_occ->edge_occ[rnb_left] = en->edge_occ[rnb_left];
    en->edge_occ[rnb_left] = nil;

    // --- update the et_tree --- //
    // split off first occurrence and delete it
    et_tree s1, s2;
    split(first_nd,rnb_right,s1,s2,dc->et_dummy);
    delete first_nd;

    // split immediately before en
    split(en,rnb_left,s1,s2,dc->et_dummy);

    // join the pieces
    et = et_join(s2,et_join(s1,new_occ,dc->et_dummy),dc->et_dummy);
  }
  }
```

This macro is defined in definitions 35, 36, 37, 38, 39, and 40.
This macro is invoked in definition 34.

5.3.4 et_link

Let `u` and `v` be two nodes of the graph G. The operation `et_link(u,v,e,i,dc)` links the two `et_trees` T_1 and T_2 on level `i` containing the active occurrences `u_act` and `v_act` of `u` and `v` on level `i`. `e` is the edge consisting of `u` and `v`. `dc` is a pointer to the dynamic connectivity data structure `u` and `v` belong to. Of course, we require `u` and `v` to be disconnected at level `i`.

This is implemented as follows.

1. We look up the active occurrences of `u` and `v` on level `i`, `u_act` and `v_act`. We create a new occurrence for `u`. We reroot the tree containing `v_act` at `v_act`.
2. We initialize the `tree_occ[i]` array of `e`.
3. We update the `tree_occ[i]` array of the edge following `e` in the Euler tour sense (if it exists).
4. We update the `edge_occs` of the involved occurrences.
5. We join the `et_trees` corresponding to the change in the Euler tour sequences as described in the Introduction of this Chapter.

Operations on et_trees[38] $+\equiv$

```
{
et_tree et_link(node u, node v, edge e, int i, dyn_con* dc)
// Modify the et_trees at level i corresponding to the insertion of
// the edge e = (u,v) into F_i.
// Prec.: u and v belong to dc, and they are not connected at the
//        valid level i.
{
  // get active occurrences of u and v, create a new occurrence of u
  et_node u_act = dc->Gp->inf(u)->act_occ[i];
  et_node v_act = dc->Gp->inf(v)->act_occ[i];
  et_node new_u_occ = new et_node_struct(u_act);

  // find the tree etv containing v_act and reroot it at v_act
  et_tree etv = v_act->find_root();
  change_root(etv,v_act,i,dc);

  // --- initialize tree_occs of e ---
  // u_act and new_u_occ become the first two tree_occs of e
  dc->Gp->inf(e)->tree_occ[i][0] = u_act;
  dc->Gp->inf(e)->tree_occ[i][1] = new_u_occ;

  // the first and the last node of etv are tree_occ[i][2 and 3] if
  // they are different otherwise tree_occ[i][2] is nil
  et_node etv_last = etv->last();
  dc->Gp->inf(e)->tree_occ[i][3] = etv_last;
  if(etv_last != v_act) dc->Gp->inf(e)->tree_occ[i][2] = v_act;
  else                  dc->Gp->inf(e)->tree_occ[i][2] = nil;

  // --- update tree_occs of the edge following e if it exists ---
  edge after_e = u_act->edge_occ[rnb_right];
  if(after_e)
  {
    if(u_act->edge_occ[rnb_left] != after_e)
    {
      // replace u_act by new_u_occ
      for(int k=0; u_act != dc->Gp->inf(after_e)->tree_occ[i][k]; k++);
      dc->Gp->inf(after_e)->tree_occ[i][k] = new_u_occ;
    }
    else
    {
      // replace nil pointer by new_u_occ
      for(int k=0; nil != dc->Gp->inf(after_e)->tree_occ[i][k]; k++);
      dc->Gp->inf(after_e)->tree_occ[i][k] = new_u_occ;
    }
  }

  // --- update edge_occs ---
  new_u_occ->edge_occ[rnb_right] = u_act->edge_occ[rnb_right];
  new_u_occ->edge_occ[rnb_left] = e;
  u_act->edge_occ[rnb_right] = e;
  v_act->edge_occ[rnb_left] = e;
  etv_last->edge_occ[rnb_right] = e;

  // --- update et_trees ---
  // concatenate etv and the new occurrence
  etv = et_join(etv,new_u_occ,dc->et_dummy);

  // split the et_tree containing u_act after u_act
  et_tree s1, s2;
  split(u_act,rnb_right,s1,s2,dc->et_dummy);

  // concatenate the pieces
```

```
    return et_join(s1,et_join(etv,s2,dc->et_dummy),dc->et_dummy);
  }
  }
```

This macro is defined in definitions 35, 36, 37, 38, 39, and 40.
This macro is invoked in definition 34.

5.3.5 et_cut

This is the opposite operation of the previous one. An `et_tree` representing a tree T at level `i` is split by deleting one of its edges `e`. This edge `e` is represented by 3 or 4 occurrences. One pair of occurrences originates from the first traversal of `e` in $ET(T)$ and the other pair originates from the second traversal of `e`. The second occurrence of the first traversal is also the first occurrence of the second traversal if the second node of the edge is a leaf. Exactly in this case the edge is represented only by 3 occurrences. The operation is implemented as follows.

1. Let `ea1` and `ea2` be the first two `tree_occs` of `e`, and let `eb1` and `eb2` be the second two `tree_occs` of `e`. We insure that `ea1` $<$ `eb1` $<$ `eb2` $<$ `ea2` in the search tree order.
2. We set the `tree_occs` of `e` to `nil`.
3. One of the resulting two `et_trees` is given by the subtree represented by the subsequence starting at `eb1` and ending with `eb2`. The other `et_tree` is given by deleting the subsequence starting at `eb1` and ending with `ea2` from the sequence for T.
4. `ea2` will be deleted, so we pass its activity to `ea1` if `ea2` is actually active.
5. We update the `tree_occs` of the edge following `e` in the Euler tour sense, if it exists.
6. We update the `edge_occs` of `ea1`, `eb1` and `eb2`.
7. We delete `ea2`.

Operations on et_trees[39] $+\equiv$

```
{
 void et_cut(edge e, int i, dyn_con* dc)
 // Update the et_trees at level corresponding to the removal of
 // the tree edge e.
 // Prec.: e actually is a tree edge at level i.
 {
   // get the et_nodes representing e at level i
   et_node ea1 = dc->Gp->inf(e)->tree_occ[i][0];
   et_node ea2 = dc->Gp->inf(e)->tree_occ[i][1];
   et_node eb1 = dc->Gp->inf(e)->tree_occ[i][2];
   et_node eb2 = dc->Gp->inf(e)->tree_occ[i][3];

   // set the tree_occ to nil;
   dc->Gp->inf(e)->tree_occ[i][0] = nil;
   dc->Gp->inf(e)->tree_occ[i][1] = nil;
   dc->Gp->inf(e)->tree_occ[i][2] = nil;
   dc->Gp->inf(e)->tree_occ[i][3] = nil;

   // sort ea1, ea2, eb1, and eb2, such that
   // ea1 < eb1 < eb2 < ea2 in In-order if they are not nil
   // eb1 may be nil
   et_node aux;
   if(ea1 && ea2)
   {
     if(smaller(ea2,ea1))
     {
       aux = ea1; ea1 = ea2; ea2 = aux;
     }
   }
   else      // either ea1 or ea2 is nil...
   {
     if(ea1) // ...it is ea2
     {
       ea2 = ea1; ea1 = nil;
     }
```

```
    }

    if(eb1 && eb2)
    {
      if(smaller(eb2,eb1))
      {
        aux = eb1; eb1 = eb2; eb2 = aux;
      }
    }
    else      // either eb1 or eb2 is nil...
    {
      if(eb1) // ...it is eb2
      {
        eb2 = eb1; eb1 = nil;
      }
    }

    // now ea2 and eb2 are non-nil
    if(smaller(ea2,eb2))
    {
      aux = eb1; eb1 = ea1; ea1 = aux;
      aux = eb2; eb2 = ea2; ea2 = aux;
    }

    // --- update et_trees ---
    // compute s1, s2 and s3
    et_tree s1, s2, s3;
    split(ea1,rnb_right,s1,s2,dc->et_dummy);
    split(ea2,rnb_right,s2,s3,dc->et_dummy);

    // compute the first of the two resulting trees
    et_join(s1,s3,dc->et_dummy);

    // split off ea2 from s2 giving the second tree
    split(eb2,rnb_right,s1,s2,dc->et_dummy);

    // --- update active occurrences ---
    if(ea2->active) ea2->pass_activity(ea1);

    // --- update tree_occs ---
    // update tree_occs of the edge following e if it exists
    edge after_e = ea2->edge_occ[rnb_right];
    if(after_e)
    {
      if(ea1->edge_occ[rnb_left] != after_e)
      {
        // replace ea2 by ea1
        for(int k=0; ea2 != dc->Gp->inf(after_e)->tree_occ[i][k]; k++);
        dc->Gp->inf(after_e)->tree_occ[i][k] = ea1;
      }
      else
      {
        // replace ea2 by nil
        for(int k=0; ea2 != dc->Gp->inf(after_e)->tree_occ[i][k]; k++);
        dc->Gp->inf(after_e)->tree_occ[i][k] = nil;
      }
    }

    // --- update edge_occs --- //
    ea1->edge_occ[rnb_right] = ea2->edge_occ[rnb_right];
    if(eb1) eb1->edge_occ[rnb_left] = nil;
    else    eb2->edge_occ[rnb_left] = nil;
    eb2->edge_occ[rnb_right] = nil;

    delete ea2;
  }
  }
```

This macro is defined in definitions 35, 36, 37, 38, 39, and 40.
This macro is invoked in definition 34.

5.3.6 print

For better readability we do not print the memory locations contained in `corr_node` and `edge_occ[i]`, but the corresponding node indices.

Operations on et_trees[40] $+\equiv$

```
{
 void et_node_struct::print()
 // we redefine print in order to output also the additional fields
```

```
{
  // output base fields
  rnbw_node_struct::print();

  // output new fields
  cout << "  dc:          " << dc << "\n";
  cout << "  corr_node:   " << index(corr_node) << "\n";
  cout << "  level:       " << level << "\n";
  cout << "  active:      " << active << "\n";
  if(edge_occ[0])
  {
    cout << "  edge_occ[0]: " << "(" << index(source(edge_occ[0]));
    cout << "," << index(target(edge_occ[0])) << ")\n";
  }
  else cout << "  edge_occ[0]: nil\n";
  if(edge_occ[1])
  {
    cout << "  edge_occ[1]: " << "(" << index(source(edge_occ[1]));
    cout << "," << index(target(edge_occ[1])) << ")\n";
  }
  else cout << "  edge_occ[1]: nil\n";
}
}
```

This macro is defined in definitions 35, 36, 37, 38, 39, and 40.
This macro is invoked in definition 34.

Chapter 6

Adjacency Trees

The non-tree edges at level i incident to the node u are stored in a balanced binary tree, called `ed_tree`, in order to permit efficient random sampling. The data structure for these trees is again derived from the `rnbw_tree` structure.

`ed_trees` implement an unordered list of edges with the possibility to insert an element, to delete an element, and to access the k^{th} element in logarithmic time. This is useful for random sampling in the replacement procedure for deleted tree edges.

6.1 Interface

In this section we present the interface of the data structure. It is contained the header file `ed_tree.h`.

ed_tree.h[41] $\equiv$

```
{// ------------------------------------------------------------ //
 // ed_tree.h: declaration of ed_trees. An ed_tree stores the     //
 //            non-tree edges adjacent to a node at a certain     //
 //            level in the dynamic connectivity algorithm by     //
 //            M. Rauch Henzinger and V. King                     //
 //                                                               //
 Version[1]
 //                                                               //
 LEGAL NOTE[2]
 // ------------------------------------------------------------ //

 // RCS Id //
 /* $Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $ */

 #ifndef ED_TREE
 #define ED_TREE

 #include<LEDA/graph.h>
 #include"rnbw_tree.h"

 Declaration of ed_trees[42]

 #endif
}
```

This macro is attached to an output file.

We begin by defining the nodes of an `ed_tree`. They are pointers to structures containing the information stored at the node. An `ed_tree` is just an `ed_node` with no parent.

There is just one new fields in an `ed_node_struct` with respect to a `rnbw_node_struct`. It is the `ed_edge` field which points to the corresponding non-tree edge.

Declaration of ed_trees[42] ≡

```
{
class   ed_node_struct;
typedef ed_node_struct* ed_node;
typedef ed_node ed_tree;

class ed_node_struct : public rnbw_node_struct {

public:

  Interface of ed_trees[43]

private:

  edge ed_edge;    // corresponding edge in G
};
}
```

This macro is invoked in definition 41.

Interface of ed_trees[43] ≡

```
{ed_node_struct(edge e) : rnbw_node_struct(1)   // constructor
{                         // each node contains exactly one edge
  ed_edge = e;
}

Conversion Functions for ed_trees[44]

edge get_corr_edge() { return ed_edge; }
// this ed_node corresponds to the returned edge of the graph

friend ed_node ed_insert(ed_tree& edt, edge e, ed_node dummy);
// create a new node for e and insert it into the tree edt
// the new root of the tree is stored in edt
// the new node is returned

friend void ed_delete(ed_tree& edt, ed_node edn, ed_node dummy);
// delete the node edn in the ed_tree edt
// the new root is stored in edt

void print();
// prints this node to the screen, for testing}
```

This macro is invoked in definition 42.

Conversion Functions for ed_trees[44] ≡

```
{// --- Conversion Functions ---
ed_node left_child() { return (ed_node) child[rnb_left]; }
ed_node right_child() { return (ed_node) child[rnb_right]; }
friend inline ed_tree ed_join(ed_tree t1, ed_tree t2, ed_node dummy)
{ return (ed_tree) rnb_join(t1,t2,dummy); }
friend inline ed_node ed_locate(ed_tree edt, int w, int& offset)
{ return (ed_node) rnbw_locate(edt,w,offset); }}
```

This macro is invoked in definition 43.

6.2 Implementation

The `ed_tree` procedures are contained in the file `ed_tree.c`. In the following subsections we implement these procedures.

ed_tree.c[45] ≡

```
{// ------------------------------------------------------------- //
// ed_tree.c: implementation of ed_trees.                        //
//                                                               //
Version[1]
//                                                               //
LEGAL NOTE[2]
// ------------------------------------------------------------- //

// RCS Id //
static char rcs[]="$Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $";
```

```
#include"ed_tree.h"
```
Operations on ed_trees[46]
}

This macro is attached to an output file.

6.2.1 ed_insert

Operations on ed_trees[46] $+\equiv$

{
```
ed_node ed_insert(ed_tree& edt, edge e, ed_node dummy)
// create a new node for e and insert it into the tree edt
// the new root of the tree is stored at edt
// the new node is returned
{
  ed_tree aux = new ed_node_struct(e);
  edt = ed_join(edt,aux,dummy);
  return aux;
}
```
}

This macro is defined in definitions 46, 47, and 48.
This macro is invoked in definition 45.

6.2.2 ed_delete

Operations on ed_trees[47] $+\equiv$

{
```
void ed_delete(ed_tree& edt, ed_node edn, ed_node dummy)
// delete the node edn in the ed_tree edt
// the new root is stored in edt
{
  // split off edn
  ed_tree t1,t2,t3;
  split(edn,rnb_left,t1,t2,dummy);
  split(edn,rnb_right,t3,t2,dummy);

  // now t3 contains just edn so we can safely delete edn
  delete edn;

  // merge the remaining pieces together again
  edt = ed_join(t1,t2,dummy);
}
```
}

This macro is defined in definitions 46, 47, and 48.
This macro is invoked in definition 45.

6.2.3 print

Operations on ed_trees[48] $+\equiv$

{
```
void ed_node_struct::print()
// we redefine print in order to output also the additional fields
{
  // output base fields
  rnbw_node_struct::print();

  // output new fields
  cout << "  ed_edge:     " << ed_edge << "\n";
}
```
}

This macro is defined in definitions 46, 47, and 48.
This macro is invoked in definition 45.

Chapter 7

Algorithm

In this Chapter we describe the algorithm which will be implemented in the following Chapter. The algorithm differs in some details from the one described in [10]. We will point out these differences.

7.1 Data Structure

As already described in Chapter 2 the current graph G is always partitioned in edge disjoint subgraphs $G_0, \ldots, G_l$. l is called `max_level` in the implementation. In each G_i we maintain a forest F_i such that $\cup_{i \leq k} F_i$ is a spanning forest of $\cup_{i \leq k} G_i$ and $F := \cup_i F_i$ is a spanning forest of G. A spanning tree at level i is a tree in $\cup_{j \leq i} F_j$.

- There is an array of edge lists, called `tree_edges`, for the tree edges at each level. There is also an array of edge lists, called `non_tree_edges`, for the non-tree edges at each level.
- Each spanning tree at each level is maintained as an `et_tree`.
- The graph G is represented by a parameterized LEDA graph which allows storing some additional information at the nodes and edges.
- At each node v we keep an array `act_occ` for the active occurrences of v at each level.
- We also keep an array of `ed_trees`, called `adj_edges`, at v containing the non-tree edges adjacent to v at each level.
- At each edge we keep its level.
- If an edge e is a non-tree edge we keep a pointer at e pointing to its occurrence in the list of non-tree edges at its level. We also keep two pointers to the two occurrences of e in the `ed_trees` of its endpoints for its level.
- If e is a tree edge we store at e a pointer to its occurrence in the list of tree edges at its level. We also store at e an array of pointers to those `et_nodes` which represent the traversal of e. These are 3 or 4 occurrences per level starting at the level of e and above.
- The two arrays `added_edges` and `rebuild_bound` are used by the rebuild procedure. They are explained later on.
- There are some quasi-constants depending only on the number of nodes of the graph which is an invariant. These are `max_level`, the maximum level, `edges_to_sample`, the number of edges to sample in order to find a replacement edge in the tree edge replacement procedure, `small_set`, a bound used

in the replacement algorithm in order to decide if some cut is sparse, and `small_weight`, a bound also used in `replace` in order to decide whether it is better to sample or to look at all edges, if there are only few of them.

- There are two dummy nodes needed for the join and split operations on `et_trees` and `ed_trees`.

There are some differences to [10]. We do not implement the variant of the algorithm which uses $(\log n)$-ary trees for the `et_trees` on the highest level and the `ed_trees` on each level in order to shave off a $O(\log \log n)$ factor, since we think on the one hand that there is no practical relevance to it in terms of performance, and on the other hand it would lead to a noticeably more complicated implementation. `rebuilds` are handled somewhat different, see the next section. We store non-tree edges at level i adjacent to a node v at v instead of storing them at the active occurrence of v at level i. At a non-tree edge e we store pointers to the two `ed_nodes` containing e instead of "pointers to the two leaves of the $ET - tree$ in which it is stored" [10].

7.2 Internal Functions

We use the following internal functions in order to realize the interface operations `ins`, `del` and `connected`.

- `bool connected(node x, node y, int i)` Return `true` iff the nodes `x` and `y` are connected at level `i`.
- `bool tree_edge(edge e)` Return `true` iff `e` is a tree edge.
- `int level(edge e)` Return the level of `e`.
- `void insert_tree(edge e, int i, bool create_tree_occs)` Insert the already existing edge `e` as a tree edge at level `i`. `create_tree_occs` is `false` by default, if it is true then the `tree_occ` array for `e` is created.
- `void delete_tree(edge e)` Delete the tree edge e in the data structure, but not in the graph.
- `void replace(node u, node v, int i)` Try to replace the deleted level `i` tree edge which connected the nodes `u` and `v`.
- `edge sample_and_test(et_tree T, int i)` Randomly select a non-tree edge e adjacent to the tree `T` at level `i`, where an edge with both endpoints in `T` is picked with probability $2/w(\mathtt{T})$ and an edge with exactly one endpoint in `T` is picked with probability $1/w(\mathtt{T})$. If exactly one endpoint of e is in `T` return e, else return `nil`.
- `get_cut_edges(et_tree T, int i, list<edge>& cut_edges)` Return the list of all all non-tree edges at level `i` with exactly one endpoint in the tree `T` at level `i`. Normally, this list would be empty, but the function is called by `replace` after a tree edge deletion. It uses an auxiliary function called `traverse_edges`.
- `void insert_non_tree(edge e, int i)` Insert the edge `e` as a non-tree edge at level `i`.
- `void delete_non_tree(edge e)` Delete the non-tree edge `e` in the data structure, but not in the graph.
- `void rebuild(int i)` Rebuild level `i` if necessary. This is explained below.
- `void move_edges(int i)` Move all edges in levels $j \geq i$ to level $i - 1$.

Unlike [10] `insert_tree` and `delete_tree` work on all necessary levels, i.e., a tree edge `e` is deleted at all levels with a call to `delete_tree(e)`. There is no function `tree` in our implementation. We use the `connected` function and the `find_root` function of `et_trees` depending on the situation, instead. The function `get_cut_edges` replaces `nontree_edges` in [10] which returned all adjacent non-tree edges.

The procedure `rebuild` is necessary in order to bound the number of levels, since edges may be moved up during tree edge deletions. If there are no insertions, this is no problem, but when insertions are also allowed, then from time to time some edges have to be moved down again. This is done by `move_edges`. `rebuild` merely checks whether it is necessary to call `move_edges` at a certain level. In order to describe the condition for the necessity to move edges down, we introduce the notion of edge additions. An edge is *added to level* `k` if it is either newly inserted by `ins` at level `k` or moved up from level `k-1` by a `replace`. `rebuild(i)` checks whether the sum of added edges in all levels `j≥i` reaches `rebuild_bound[i]`. If this is the case then `move_edges(i)` is called. The value `rebuild_bound[max_level]` is some constant which may be given by the user. For all `i<max_level` we have `rebuild_bound[i] =2 * rebuild_bound[i+1]`. A `rebuild` is usually an expensive operation. By supplying a high `rebuild_bound[max_level]` the number of `rebuilds` drops, but each update which does not cause a rebuild might take longer, and the fewer `rebuilds` are also more expensive, because there are more edges involved. By supplying a low `rebuild_bound[max_level]` there are more `rebuilds`, but each of them deals with fewer edges, and an update which does not cause a `rebuild` might be faster.

Since `replace` is a very important function, we present it in pseudocode.

replace(u, v, i)

1. **Let** T_u and T_v be the spanning trees at level i containing u and v, respectively. **Let** T be the tree with smaller weight among T_u and T_v. Ties are broken arbitrarily.

2. **If** $w(t) \geq \log^2 n$ **then**

   (a) **Repeat** sample_and_test(T) for at most $32 \log^2 n$ times. Stop if a replacement edge e is found.

   (b) **If** a replacement edge e is found **then do** delete_non_tree(e), insert_tree(e, i), and **return**.

3. (a) **Let** S be the set of edges with exactly one endpoint in T.

   (b) **If** $|S| \geq w(T)/(16 \log n)$ **then**
   Select one $e \in S$, delete_non_tree(e), and insert_tree(e, i).

   (c) **Elsif** $0 < |S| < w(T)/(16 \log n)$ **then**
   Delete one edge e from S, delete_non_tree(e), and insert_tree($e, i + 1$).
   Forall $e' \in S$ **do** delete_non_tree(e') and insert_non_tree($e', i + 1$).
   Update added_edges[$i + 1$] and rebuild level $i + 1$ if necessary.

   (d) **Else if** $i < l$ **then** replace($u, v, i + 1$).

The "constants" depending only on n – $\log^2 n, 32 \log^2 n, 16 \log n$ – can be changed in the implementation. See the next Chapter. In the implementation $\log^2 n$ is called `small_weight`, $32 \log n$ is called `edges_to_sample`, and $16 \log n$ is called `small_set`.

`sample_and_test(T,i)` is realized as follows. We determine the weight w of `T`, i.e., the number of adjacent non-tree edges, and select a random index between 1 and w. We `et_locate` the active occurrence o in `T` which represents this index in search tree order. This yields also an offset. o corresponds to a node v of the graph. We `ed_locate` the edge $e = (u, v)$ corresponding to the offset in the `ed_tree` at level i of v. We return e if `connected(u,v,i)` is `false`.

7.3 Interface Functions and Initialization

Using the above described internal functions the interface functions are realized as follows.

- `connected(u,v)` is simply done by a call to `connected(u,v,max_level)`.

- When `ins(u,v)` is called we first generate the new edge `e=(u,v)` in the graph and check whether it becomes a tree edge or not using `connected`. If `e` becomes a tree edge then it is inserted on the highest level by `insert_tree(e,max_level)`. If `e` becomes a non-tree edge – i.e., `u` and `v` are already connected at level `k` and above – we compute `k` by means of a binary search, and insert `e` there by `insert_non_tree(e,k)`. If `e` was inserted into level `r` then we increment the count of added edges for level `r`, and check whether a rebuild of level `r` is necessary.

- The deletion of an edge `e=(u,v)` by `del(e)` is done as follows. We first check whether `e` is a tree edge or a non-tree edge. If it is a non-tree edge, we simply delete it by `delete_non_tree(e)`. If it is a tree edge at level `i`, we delete it by `delete_tree(e)`, we call `replace(u,v,i)`, and we do a rebuild if necessary, i.e., if edges were moved during `replace` and pushed the number of added edges beyond the bound at some level `j>i`.

The data structure is initialized by inserting all edges into level 0. There are some possibilities to adapt the data structure to a special input situation. This can be done by assigning values to the optional parameters of the constructor. We describe the two most important ones here. The first one allows us to prescribe the bound for rebuilds on the highest level, i.e., the value of `rebuild_bound[max_level]`. Its default is 5000. The second one determines the number of levels if set. If it is not set, then the number of the number of levels depends on the value of `rebuild_bound[max_level]`. In [10] there are $6 \log n$ levels, and there is an implicit `rebuild_bound[max_level]` of 4. We skip approximately as many levels as necessary in order to get to `rebuild_bound[max_level]`, e.g., with `rebuild_bound[max_level] = 16` we would use two levels less. Note that prescribing a certain number of levels can invalidate the analysis of the running time in [10].

Chapter 8

Implementation

8.1 Overview

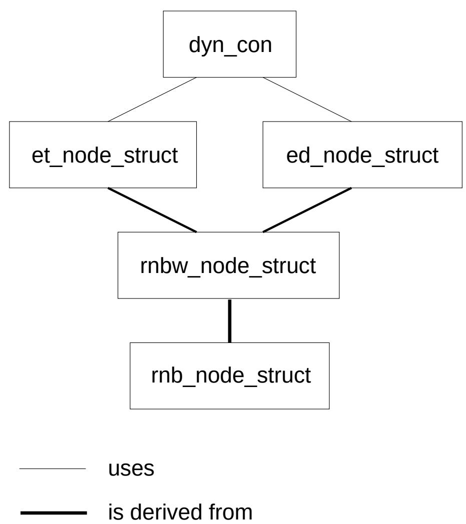

Figure 8.1: The Classes used in the Implementation

The class `rnb_node_struct` implements a node in a randomized balanced binary tree. The derived class `rnbw_node_struct` implements a node in a randomized balanced binary tree with a weight. Weights of subtrees are maintained. The class `et_node_struct` implements one node in an `et_tree`. `et_trees` are used to represent the various trees in the algorithm. The class `ed_node_struct` implements one node in an `ed_tree`. `ed_trees` store the non-tree edges adjacent to a node of the graph at a certain level.

We implement the algorithm as a `C++` class `dyn_con`. All data which is global to the update algorithm like the lists of tree and non-tree edges is encapsulated in this class. The constructor of this class is the initialization function for the data structure. It takes an initial graph as its argument. The public methods are the query `connected` and the update operations `ins` and `del`. In addition there are the destructor and `print_statistics` which prints a summary of the operations performed so far to a specified output stream. The interface is given in the file `dyn_con.h`.

dyn_con.h[49] $\equiv$

```
{// ------------------------------------------------------------- //
```

```
// dyn_con.h: This header file contains the interface to a C++   //
//            implementation of the polylogarithmic dynamic      //
//            connectivity alg. by Monika Rauch Henzinger and    //
//            Valerie King (STOC 95).                             //
//                                                                //
//            See also the documentation in dyn_con.ps.          //
//                                                                //
```

Version[1]

```
//                                                                //
```

LEGAL NOTE[2]

```
// -------------------------------------------------------------- //

// RCS ID //
/* $Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $ */

#ifndef DYN_CON
#define DYN_CON

#include"ed_tree.h"
#include<LEDA/graph.h>
#include<LEDA/list.h>
```

Declaration of et_trees[29]

Declaration of dc_graph[52]

Declaration of class dyn_con[50]

```
#endif
```

}

This macro is attached to an output file.

Like in [10] we do not implement the public methods directly, but by means of some internal functions. These are the private methods of the class `dyn_con`. Since there is a strong interaction between `et_trees` and their `dyn_con` structure, `et_node_struct` is a friend class of `dyn_con` and the friends of `et_node_struct` are also friends of `dyn_con`.

Declaration of class dyn_con[50] ≡

```
{class dyn_con{

public:

  dyn_con(dc_graph& G, int ml_reb_bound = -1, int n_levels = -1,
          int edges_to_samp = -1, int small_w = -1, int small_s = -1);
  // constructor, initializes the dynamic connectivity data structure
  // if ml_reb_bound >= 1 it specifies rebuild_bound[max_level] (default 5000)
  // if n_levels > 0 then it specifies the number of levels (default O(log n))
  // if edges_to_samp >= 0 then it specifies edges_to_sample
  // (default 32 log^2 n)
  // if small_w >= 0 then it specifies small_weight (default log^2 n)
  // if small_s >= 1 then it specifies small_set (default 16 log n)

  ~dyn_con();
  // destructor

  edge ins(node u, node v);
  // create an edge connecting u and v and return it

  void del(edge e);
  // delete the edge e

  bool connected(node u, node v);
  // return true if u and v are connected in the current graph
  // and false otherwise

  void print_statistics(ostream& out);
  // prints some statistics to the output stream out

private:
```

dyn_con Data[53]

Private dyn_con Methods[54]

```
  friend class et_node_struct;
  friend void change_root(et_tree& et, et_node en, int i, dyn_con* dc);
  friend et_tree et_link(node u, node v, edge e, int i, dyn_con* dc);
  friend void et_cut(edge e, int i, dyn_con* dc);
};}
```

This macro is invoked in definition 49.

8.2 Usage

We present a simple program which illustrates how to use the implementation of the dynamic connectivity data structure.

foo.c[51] ≡

```
{// ---------------------------------------------------- //
// foo.c: Source code of a stupid example program using  //
//        the dynamic connectivity data structure.       //
// ---------------------------------------------------- //

#include"dyn_con.h"
#include<LEDA/graph.h>

main()
{
  // the graph has to be a dc_graph
  dc_graph G;

  // build initial graph (a circle)
  node nodes[100];
  for(int i=0; i<100; i++) nodes[i] = G.new_node(nil);
  for(i=0; i<99; i++) G.new_edge(nodes[i],nodes[i+1],nil);
  G.new_edge(nodes[99],nodes[0],nil);

  // initialize the data structure
  dyn_con dc(G);
  // insert an edge
  edge e = dc.ins(nodes[0],nodes[10]);
  // ask a query
  if( dc.connected( nodes[17] , nodes[42] ) )
  {
    cout << "This is the right answer.\n";
  }
  else
  {
    cout << "This should not happen!\n";
    cout << "Did you change the source code?\n";
  }
  // delete an edge
  dc.del(e);
  // print statistics
  dc.print_statistics(cout);
}
}
```

This macro is attached to an output file.

Comments:

1. Include the header file `dyn_con.h` in your application.

2. The graph for which the dynamic connectivity data structure is maintained has to be of type `dc_graph` (defined in `dyn_con.h`). You cannot use a graph of another type, e.g., a graph parameterized with your own data structures for the nodes and edges. This is due to the design of LEDA, see next item.

3. You must not change the information which is maintained at the nodes and edges of the graph by the dynamic data structure. Unfortunately in LEDA there is currently no possibility to store information associated with a dynamically changing edge set in a safe and efficient way. We chose to use a parameterized graph. This is efficient, but unsafe, i.e., if you change the information at a node or an edge, something unpredictable might happen. Moreover, it might be inconvenient for the application to use a `dc_graph` instead of a parameterized graph of a different kind.

4. Link the program with `libdc.a` and the LEDA libraries `libG.a` and `libL.a`, e.g., `g++ foo.c -o foo -I. -L. -ldc -lG -lL` assuming that the libraries and the header files are in some standard directory or the current directory.

5. The library `libdc.a` can be made by using the supplied `Makefile`, e.g., type `make lib`.
6. Statistics are only maintained if `libdc.a` is compiled with `-DSTATISTICS`. If this is not the case then `print_statistics` only prints a short message that there are no statistics available.
7. If the library was compiled with `-DDEBUG` then it prints a lot of messages telling you which internal functions are executed.
8. The implementation also works with multigraphs, i.e., there can be more than one edge connecting one pair of nodes, and edges connecting a node to itself are also allowed.
9. A study on the performance comparing this imlementation with static algorithms and other dynamic algorithms will be given in a future paper.

8.3 Data

Some of the data needed in the algorithm is stored at the nodes and edges of the graph G. We use a parameterized LEDA GRAPH (and define this type to be a `dc_graph`) and store pointers to auxiliary structures at each node and edge. We declare the type of these structures below.

Declaration of dc_graph[52] $\equiv$

```
{// ---------- dc_graph -------------- //
class dc_node_struct{
public:
  et_node *act_occ;    // array of active occurrences
  ed_tree *adj_edges;  // array of ed_trees storing adjacent non-tree edges

  dc_node_struct()
  { act_occ = nil; adj_edges = nil; }
};

class dc_edge_struct{
public:
  int        level;             // this edge belongs to G_level
  list_item  non_tree_item;     // non-tree edge: list_item in
                                // non_tree_edges[level]
                                // tree edge: nil
  list_item  tree_item;         // tree edge: list_item in tree_edges[level]
                                // non-tree edge: nil
  ed_node    non_tree_occ[2];   // non-tree edge: pointer to the two
                                // corresponding ed_nodes
                                // tree edge: both are nil
  et_node  **tree_occ;          // tree edge: array for each level the
                                // 4 pointers to occurrences repr. this edge
                                // non-tree edge: nil

  dc_edge_struct()
  { non_tree_item = nil; tree_item = nil;
    non_tree_occ[0] = non_tree_occ[1] = nil; tree_occ = nil; }
};

typedef dc_node_struct* dc_node_inf;
typedef dc_edge_struct* dc_edge_inf;

// the type of the input graph

typedef GRAPH<dc_node_inf,dc_edge_inf> dc_graph;}
```

This macro is invoked in definition 49.

There is also some global information which is neither associated with a particular node nor with a particular edge. This is stored as private data of the class `dyn_con`.

dyn_con Data[53] $\equiv$

```
{dc_graph    *Gp;              // pointer to the graph
 int          max_level;       // the maximum level
 list<edge>  *tree_edges;      // a list of tree edges for each level
 list<edge>  *non_tree_edges;  // a list of non-tree edges for each level

 int         *added_edges;     // array of #edges added to each level
```

```
                                    // since last rebuild at lower level
int          *rebuild_bound;        // array, s.t. sum for j>=i of added_edges[j]
                                    // > rebuild_bound[i] <=> rebuild at level i
                                    // necessary

int          small_weight;          // bound used in replace
int          edges_to_sample;       // sample at most this many edges while
                                    // searching for a replacement edge for a
                                    // deleted tree edge
int          small_set;             // used in the replacement algorithm, too

et_node      et_dummy;              // dummy nodes for splitting and joining trees
ed_node      ed_dummy;

// some statistics - these counters are only maintained with
// the -DSTATISTICS compile option
int          n_ins;                 // number of ins operations
int          n_del;                 // number of del operations
int          n_query;               // number of user supplied connected queries
int          n_connected;           // number of conecteds
int          n_ins_tree;            // number of insert_trees
int          n_del_tree;            // number of delete_trees
int          n_replace;             // number of replaces
int          rep_small_weight;  // w(T_1) small in replace
int          rep_succ;              // successful sampling
int          rep_big_cut;           // big cut
int          rep_sparse_cut;        // sparse cut
int          rep_empty_cut;         // empty cut
int          n_sample_and_test; // number of sample_and_test
int          n_get_cut_edges; // number of invocations of get_cut_edges in
                                    // replace (get_cut_edges is recursive)
int          n_ins_non_tree;        // number of insert_non_tree
int          n_del_non_tree;        // number of delete_non_tree
int          n_move_edges;          // number of move_edges
int          edges_moved_up;        // number of edges moved up (during replace)
int          edges_moved_down;  // number of edges moved down
}
```

This macro is invoked in definition 50.

8.4 Private Methods

We represent the internal functions of the dynamic connectivity algorithm as private methods of the class `dyn_con`.

Private dyn_con Methods[54] ≡

```
{// --- Internal Functions of the dynamic connectivity data structure --- //
 // Let G_i be the subgraph of G on level i, let F_i be the
 // forest in G_i, and let F be the spanning forest of G.
 // Let T be a spanning tree on level i.

 bool connected(node x, node y, int i);
 // Return true if x and y are connected on level i. Otherwise
 // return false.

 bool tree_edge(edge e);
 // Return true if e is an edge in F, false otherwise.

 int level(edge e);
 // Return i such that e is in G_i.

 void insert_tree(edge e, int i, bool create_tree_occs = false);
 // Insert e into F_i. If create_tree_occs is true the storage for the
 // tree_occ array for e is allocated.

 void delete_tree(edge e);
 // Remove the tree edge e from F.

 void replace(node u, node v, int i);
 // Replace the deleted tree edge (u,v) at level i.

 edge sample_and_test(et_tree T, int i);
 // Randomly select a non-tree edge of G_i that has at least one
 // endpoint in T, where an edge with both endpoints in T is picked
 // with 2/w(T) and an edge with exactly one endpoint in T is picked
 // with probability 1/w(T).
```

```
// Test if exactly one endpoint is in T, and if so, return the edge.
// Otherwise return nil.

void get_cut_edges(et_node et, int i, list<edge>& cut_edges);
// Return all non-tree edges with exactly one endpoint in the tree
// T at level i in cut_edges.

void traverse_edges(ed_node ed, list<edge>& cut_edges);
// Append edges with exactly one endpoint in the subtree rooted at ed
// to edge_list. This is an auxiliary function called by get_cut_edges.

void insert_non_tree(edge e, int i);
// Insert the non-tree edge e into G_i.

void delete_non_tree(edge e);
// Delete the non-tree edge e.

void rebuild(int i);
// Rebuild level i if necessary.

void move_edges(int i);
// For j>=i, insert all edges of F_j into F_{i-1}, and all
// non-tree edges of G_j into G_{i-1}.}
```

This macro is invoked in definition 50.

In the next Section we implement the public methods except for the constructor and the destructor. In Section 8.6 we will implement the constructor, the destructor, and the internal functions. The implementation takes place in the file `dyn_con.c`.

dyn_con.c[55] $\equiv$

```
{// ------------------------------------------------------------ //
 // dyn_con.c: implementation of internal functions and user      //
 //            interface functions for the implementation of      //
 //            the dynamic connectivity algorithm by              //
 //            M. Rauch Henzinger and V. King.                    //
 //                                                               //
 //            See also the documentation in dyn_con.ps.          //
 //                                                               //
 Version[1]
 //                                                               //
 LEGAL NOTE[2]
 // ------------------------------------------------------------ //

 // RCS Id //
 static char rcs[]="$Id: dyn_con.fw,v 1.13 1995/06/15 11:58:41 alberts Exp $";

 #include"dyn_con.h"
 #include<LEDA/queue.h>
 #include<sys/time.h>

 Interface Functions[56]

 Internal Functions[60]}
```

This macro is attached to an output file.

8.5 Interface

In this section we describe the implementation of the interface operations `ins`, `del`, and `connected` using the internal operations.

8.5.1 ins

There are two cases possible: either the new edge $e = (u, v)$ connects two components of the former graph and thus becomes a tree edge, or u and v were already connected. We check which case applies by using `connected`. In the former case we insert e as a tree edge on the highest level. In the latter case we search the lowest level l such that u and v are connected by a binary search using `connected` and insert e as a non-tree edge on level l.

Interface Functions[56] + ≡

```
{edge dyn_con::ins(node u, node v)
 // create an edge connecting u and v and return it
 {
 #ifdef STATISTICS
   n_ins++;
 #endif
   // create the new edge
   edge e = Gp->new_edge(u,v);
   (*Gp)[e] = new dc_edge_struct();

   // test whether u and v are already connected
   if(!connected(u,v,max_level))
   // they are not, so e becomes a forest edge at level max_level
   {
     insert_tree(e,max_level,true);
     added_edges[max_level]++;
     rebuild(max_level);
   }
   else
   // u and v are already connected, find lowest such level
   {
     // current level
     int curr_level = max_level/2;
     // lower bound and upper bound
     int lower = 0;
     int upper = max_level;
     while(curr_level != lower)
     {
       if(connected(u,v,curr_level))
       // search below current level
       {
         upper = curr_level;
         curr_level = (lower + curr_level)/2;
       }
       else
       // search above current level
       {
         lower = curr_level;
         curr_level = (upper + curr_level)/2;
       }
     }
     // Now depending on parity either
     // a) connected(u,v,lower-1) == false && connected(u,v,lower) == true or
     // b) connected(u,v,lower) == false && connected(u,v,lower+1) == true holds.

     // handle case a),
     if(!connected(u,v,lower)) lower++;

     // insert e at appropriate level,
     insert_non_tree(e,lower);
     added_edges[lower]++;
     rebuild(lower);
   }
   return e;
 }
 }
```

This macro is defined in definitions 56, 57, 58, and 59.
This macro is invoked in definition 55.

8.5.2 del

The edge e to be deleted is either a tree edge or not. We delete it using the appropriate method (either `delete_non_tree` or `delete_tree`). Moreover, we have to deallocate the additional information stored with the edge.

Interface Functions[57] + ≡

```
{
 void dyn_con::del(edge e)
 // delete the edge e
 {
 #ifdef STATISTICS
   n_del++;
 #endif

   // if e is not an edge in F
   if(!tree_edge(e)) delete_non_tree(e);
   else
```

```
    // e is a tree edge
    {
      // remember e
      int e_level = level(e);
      node u = source(e);
      node v = target(e);

      // remove e
      delete_tree(e);

      // delete specific information for tree edges stored at e
      for(int j=0; j<=max_level; j++) delete[] (*Gp)[e]->tree_occ[j];
      delete[] (*Gp)[e]->tree_occ;

      // look for a replacement edge
      replace(u,v,e_level);
    }

    // delete information stored at e
    delete (*Gp)[e];
    (*Gp)[e] = nil;
    Gp->del_edge(e);
  }
  }
```

This macro is defined in definitions 56, 57, 58, and 59.
This macro is invoked in definition 55.

8.5.3 connected

Interface Functions[58] + ≡

```
{
bool dyn_con::connected(node u, node v)
// return true if u and v are connected in the current graph
// and false otherwise
{
#ifdef STATISTICS
  n_query++;
#endif

  return connected(u,v,max_level);
}
}
```

This macro is defined in definitions 56, 57, 58, and 59.
This macro is invoked in definition 55.

8.5.4 print_statistics

Interface Functions[59] + ≡

```
{
void dyn_con::print_statistics(ostream& out)
{
#ifdef STATISTICS
  out << "\nStatistics\n==========\n";
  out << "number of nodes: " << Gp->number_of_nodes();
  out << " final number of edges: " << Gp->number_of_edges() << "\n\n";
  out << "user supplied operations\n";
  out << "number of ins operations: " << n_ins << "\n";
  out << "number of del operations: " << n_del << "\n";
  out << "number of connected operations: " << n_query << "\n\n";
  out << "internal variables\n";
  out << "number of levels: " << max_level+1 << "\n";
  out << "bound for rebuilds on highest level: " << rebuild_bound[max_level];
  out << "\nsmall_weight: " << small_weight << "\n";
  out << "maximum number of edges to sample: " << edges_to_sample << "\n";
  out << "small_set: " << small_set << "\n\n";
  out << "internal functions\n";
  out << "number of connected operations: " << n_connected << "\n";
  out << "number of insert_tree operations: " << n_ins_tree << "\n";
  out << "number of delete_tree operations: " << n_del_tree << "\n";
  out << "number of replace operations: " << n_replace << "\n";
  out << "  weight of T_1 too small: " << rep_small_weight << "\n";
  out << "  case 2.(b): " << rep_succ << "\n";
  out << "  case 3.(b): " << rep_big_cut << "\n";
```

```
    out << "  case 3.(c): " << rep_sparse_cut << "\n";
    out << "  case 3.(d): " << rep_empty_cut << "\n";
    out << "number of sample_and_test operations: " << n_sample_and_test << "\n";
    out << "number of get_cut_edges operations without recursive calls: ";
    out << n_get_cut_edges << "\n";
    out << "number of insert_non_tree operations: " << n_ins_non_tree << "\n";
    out << "number of delete_non_tree operations: " << n_del_non_tree << "\n";
    out << "number of move_edges: " << n_move_edges << "\n";
    out << "number of edges moved up: " << edges_moved_up << "\n";
    out << "number of edges moved down: " << edges_moved_down << "\n";
  #else
    out << "\ndyn_con::print_statistics: sorry, no statistics available\n";
    out << "  compile libdc.a with -DSTATISTICS to get statistics\n\n";
  #endif
  }
  }
```

This macro is defined in definitions 56, 57, 58, and 59.
This macro is invoked in definition 55.

8.6 Internal Functions

8.6.1 connected, tree_edge and level

Internal Functions[60] + ≡

```
{
  bool dyn_con::connected(node x, node y, int i)
  // Return true if x and y are connected on level i. Otherwise
  // return false.
  {
  #ifdef STATISTICS
    n_connected++;
  #endif

    // get the active occurrences of x and y at level i
    et_node x_act_occ = (*Gp)[x]->act_occ[i];
    et_node y_act_occ = (*Gp)[y]->act_occ[i];

    // return whether they belong to the same tree at level i
    return (x_act_occ->find_root() == y_act_occ->find_root());
  }
  }
```

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

We maintain the invariant that `(*Gp)[e]->tree_occ` is `nil` if and only if `e` is a non-tree edge. This leads to a trivial test, whether a given edge is a tree edge or not.

Internal Functions[61] + ≡

```
{
  inline bool dyn_con::tree_edge(edge e)
  // Return true if e is an edge in F, false otherwise.
  {
    return ((*Gp)[e]->tree_occ != nil);
  }
  }
```

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

Internal Functions[62] + ≡

```
{
  inline int dyn_con::level(edge e)
  // Return i such that e is in G_i.
  {
    return (*Gp)[e]->level;
  }
  }
```

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.2 insert_tree

Internal Functions[63] + ≡

{

```
void dyn_con::insert_tree(edge e, int i, bool create_tree_occs)
// Insert e into F_i. If create_tree_occs is true the storage for the
// tree_occ array for e is allocated.
{
#ifdef STATISTICS
  n_ins_tree++;
#endif

  // find the endpoints of e
  node u = source(e);
  node v = target(e);

#ifdef DEBUG
  cout << "(" << index(u) << "," << index(v) << ") tree ins at level ";
  cout << i << "\n";
#endif

  // enter level of e
  (*Gp)[e]->level = i;

  // create tree_occ array for e if requested
  if(create_tree_occs)
  {
    (*Gp)[e]->tree_occ = new et_node*[max_level+1];
    for(int lev=0; lev<=max_level; lev++)
    {
      (*Gp)[e]->tree_occ[lev] = new et_node[4];
      for(int j=0; j<4; j++) (*Gp)[e]->tree_occ[lev][j] = nil;
    }
  }

  // link the et_trees containing the active occurrences of u and v
  for(int j=i; j<=max_level; j++) et_link(u,v,e,j,this);

  // append e to the list of tree edges at level i
  (*Gp)[e]->tree_item = tree_edges[i].append(e);
}
```

}

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.3 delete_tree

Internal Functions[64] + ≡

{

```
void dyn_con::delete_tree(edge e)
// Remove the tree edge e from F.
{
#ifdef STATISTICS
  n_del_tree++;
#endif

  // get the level of e
  int i = level(e);

#ifdef DEBUG
  cout << "(" << index(source(e)) << "," << index(target(e)) << ") tree del ";
  cout << "at level " << i << "\n";
#endif

  // cut the spanning trees
  for(int j=i; j<=max_level; j++) et_cut(e,j,this);

  // remove e from the list of tree edges at level i
  tree_edges[i].del_item((*Gp)[e]->tree_item);

  // set tree_item of e to nil
  (*Gp)[e]->tree_item = nil;
}
```

}

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.4 replace

Internal Functions[65] $+\equiv$

```
{
 void dyn_con::replace(node u, node v, int i)
 // try to reconnect the trees on level i containing u and v.
 // if not possible try to recurse on higher level
 {
 #ifdef STATISTICS
   n_replace++;
 #endif

   // determine the level i trees containing u and v
   et_tree t1 = (*Gp)[u]->act_occ[i]->find_root();
   et_tree t2 = (*Gp)[v]->act_occ[i]->find_root();

   // let t1 be the smaller tree
   if(t1->get_subtree_weight() > t2->get_subtree_weight()) t1 = t2;

   int handle_cut = false;
   if(t1->get_subtree_weight() > small_weight)
   {
     // sample randomly at most edges_to_sample edges
     int not_done = true;
     edge e;
     for(int j=0; not_done && (j<edges_to_sample); j++)
     {
       e = sample_and_test(t1,i);
       if(e) not_done = false;
     }

     if(e)
     {
       // sampling was successful, insert e as a tree edge at level i
       delete_non_tree(e);
       insert_tree(e,i,true);
 #ifdef STATISTICS
       rep_succ++;
 #endif
     }
     else // sampling not successful
     {
       handle_cut = true;
     }

   }
   else // weight of T_1 too small (too few adjacent edges)
   {
     handle_cut = true;
 #ifdef STATISTICS
     rep_small_weight++;
 #endif
   }

   if(handle_cut)
   // sampling was unsuccessful or too few edges
   {
     // determine all edges crossing the cut
     list<edge> cut_edges;
     if(t1->get_subtree_weight() > 0)
     {
       get_cut_edges(t1,i,cut_edges);
 #ifdef STATISTICS
       n_get_cut_edges++;
 #endif
     }

     if(cut_edges.size() == 0)
     // no replacement edge on this level, recurse on higher level if possible
     {
 #ifdef STATISTICS
       rep_empty_cut++;
 #endif
       if(i<max_level) replace(u,v,i+1);
     }
     else  // cut_edges.size() > 0
     {
       if(cut_edges.size() >= t1->get_subtree_weight()/small_set)
       // if cut_edges is big enough we reconnect t1 and t2 on level i
       {
 #ifdef STATISTICS
         rep_big_cut++;
 #endif
         edge reconnect = cut_edges.pop();
         delete_non_tree(reconnect);
```

```
        insert_tree(reconnect,i,true);
      }
      else
      {
        // 0 < cut_edges.size() < t1->get_subtree_weight()/small_set
        // there are too few edges crossing the cut
#ifdef STATISTICS
        rep_sparse_cut++;
#endif
        edge reconnect = cut_edges.pop();
        delete_non_tree(reconnect);

        if(i<max_level)
        {
          // move cut edges one level up
          insert_tree(reconnect,i+1,true);
          added_edges[i+1]++;
          edge e;
          forall(e,cut_edges)
          {
            delete_non_tree(e);
            insert_non_tree(e,i+1);
            added_edges[i+1]++;
          }
#ifdef STATISTICS
          edges_moved_up += cut_edges.size() + 1;
#endif
          rebuild(i+1);
        }
        else     // on top level, no moving of edges
        {
          insert_tree(reconnect,i,true);
        }
      }
    }
  }
}
}
```

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.5 sample_and_test

Internal Functions[66] + ≡

```
{
edge dyn_con::sample_and_test(et_tree T, int i)
// Randomly select a non-tree edge of G_i that has at least one
// endpoint in T, where an edge with both endpoints in T is picked
// with 2/w(T) and an edge with exactly one endpoint in T is picked
// with probability 1/w(T).
// Test if exactly one endpoint is in T, and if so, return the edge.
// Otherwise return nil.
{
#ifdef STATISTICS
  n_sample_and_test++;
#endif

  // get the number of adjacencies
  int no_of_adj = T->get_subtree_weight();

  // pick a random one
  int rnd_adj = 1 + (random() % no_of_adj);

  // locate the et_node representing this adjacency and get the corr. node
  int offset;
  et_node et_repr = et_locate(T,rnd_adj,offset);
  node u = et_repr->get_corr_node();

  // locate the edge corresp. to offset adjacent to u at level i
  ed_node en = ed_locate((*Gp)[u]->adj_edges[i],offset,offset);
  edge e = en->get_corr_edge();

  // get the second node of e
  node v = (source(e) == u) ? target(e) : source(e);

  // if v is in a different tree at level i then return e else nil
  if(connected(u,v,i)) return nil;
  else                 return e;
}
}
```

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.6 get_cut_edges

Internal Functions[67] + ≡

{
```
void dyn_con::traverse_edges(ed_node ed, list<edge>& edge_list)
// append edges with exactly one endpoint in subtree rooted at ed to edge_list
// auxiliary function called by get_cut_edges
{
  if(ed)
  {
    edge e = ed->get_corr_edge();
    if(!connected(source(e),target(e),level(e)))
    {
      // only one endpoint of e in current spanning tree -> append edge
      edge_list.append(e);
    }

    traverse_edges(ed->left_child(),edge_list);
    traverse_edges(ed->right_child(),edge_list);
  }
}

void dyn_con::get_cut_edges(et_node u, int level, list<edge>& result)
// Return the edges with exactly one endpoint in the et_tree rooted at u
// in result.
{
  if(u && u->get_subtree_weight())
  {
    node v = u->get_corr_node();
    if(u->is_active()) traverse_edges((*Gp)[v]->adj_edges[level],result);
    get_cut_edges(u->left_child(),level,result);
    get_cut_edges(u->right_child(),level,result);
  }
}
```
}

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.7 insert_non_tree

Internal Functions[68] + ≡

{
```
void dyn_con::insert_non_tree(edge e, int i)
// Insert the non-tree edge e into G_i.
{
#ifdef STATISTICS
  n_ins_non_tree++;
#endif

#ifdef DEBUG
  cout << "(" << index(source(e)) << "," << index(target(e));
  cout << ") non-tree ins at level " << i << "\n";
#endif

  (*Gp)[e]->level = i;
  node u = source(e);
  node v = target(e);

  // insert e in the adjacency trees of its endpoints at level i
  (*Gp)[e]->non_tree_occ[0] =
                  ed_insert((*Gp)[u]->adj_edges[i],e,ed_dummy);
  (*Gp)[e]->non_tree_occ[1] =
                  ed_insert((*Gp)[v]->adj_edges[i],e,ed_dummy);

  // update non_tree_edges[i]
  (*Gp)[e]->non_tree_item = non_tree_edges[i].append(e);

  // increase the weight of the active occurrences of u and v at level i
  (*Gp)[u]->act_occ[i]->add_weight(1);
  (*Gp)[v]->act_occ[i]->add_weight(1);
}
```

}

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.8 delete_non_tree

Internal Functions[69] + ≡

{
```
void dyn_con::delete_non_tree(edge e)
// Delete the non-tree edge e.
{
#ifdef STATISTICS
  n_del_non_tree++;
#endif

  // find the endpoints and the level of e
  node u = source(e);
  node v = target(e);
  int i = level(e);

#ifdef DEBUG
  cout << "(" << index(source(e)) << "," << index(target(e));
  cout << ") non-tree del at level " << i << "\n";
#endif

  // remove e from the ed_trees of u and v at level i
  ed_delete((*Gp)[u]->adj_edges[i],(*Gp)[e]->non_tree_occ[0],ed_dummy);
  (*Gp)[e]->non_tree_occ[0] = nil;
  ed_delete((*Gp)[v]->adj_edges[i],(*Gp)[e]->non_tree_occ[1],ed_dummy);
  (*Gp)[e]->non_tree_occ[1] = nil;

  // remove e from the list of non-tree edges at level i
  non_tree_edges[i].del_item((*Gp)[e]->non_tree_item);
  (*Gp)[e]->non_tree_item = nil;

  // decrease the weights of the active occurrences of u and v if they exist
  (*Gp)[u]->act_occ[i]->add_weight(-1);
  (*Gp)[v]->act_occ[i]->add_weight(-1);
}
```
}

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.9 rebuild

Internal Functions[70] + ≡

{
```
void dyn_con::rebuild(int i)
// does a rebuild at level i if necessary
{
  // rebuilds take place only at level 3 and higher
  if(i<3) return;

  // count added edges at level j>=i
  int sum_added_edges = 0;
  for(int j=i; j<=max_level; j++) sum_added_edges += added_edges[j];

  if(sum_added_edges > rebuild_bound[i])
  {
#ifdef DEBUG
    cout << "rebuild(" << i << ")\n";
#endif
    // move edges down
    move_edges(i);
    for(j=i; j<=max_level; j++) added_edges[j] = 0;
  }
}
```
}

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.10 move_edges

The purpose of move_edges(i) is to move all edges at each level j with j≥i to level i-1. We can easily access the edges at level j by means of the lists non_tree_edges[j] and tree_edges[j]. We move a non-tree edge e by a pair of delete_non_tree(e) and insert_non_tree(e,i-1) calls.

In principle we could also move the tree edges in the same manner by using delete_tree and insert_tree. However, if i is close to j and both are relatively small compared to max_level, this would be a waste of time, since it means also splitting all corresponding et_trees at levels j to max_level and then joining them again. We just fix the lists of tree edges and et_join the affected trees on levels i-1 to j-1, instead.

Internal Functions[71] + ≡

{
```
void dyn_con::move_edges(int i)
// For j>=i, insert all edges of F_j into F_{i-1}, and all
// non-tree edges of G_j into G_{i-1}.
{
#ifdef STATISTICS
  n_move_edges++;
#endif

  // for each level starting at max_level and ending at i...
  for(int j=max_level; j>=i; j--)
  {
#ifdef STATISTICS
    edges_moved_down += non_tree_edges[j].size() + tree_edges[j].size();
#endif
    // move non-tree edges
    while(non_tree_edges[j].size())
    {
      edge e = non_tree_edges[j].head();
      // delete non-tree edge at level j ...
      delete_non_tree(e);
      // ... and insert it into level i-1
      insert_non_tree(e,i-1);
    }

    // move tree edges
    while(tree_edges[j].size())
    {
      edge e = tree_edges[j].head();

      // update tree_edges[j], tree_edges[i-1], tree_item and level
      tree_edges[j].del_item(Gp->inf(e)->tree_item);
      Gp->inf(e)->tree_item = tree_edges[i-1].append(e);
      Gp->inf(e)->level = i-1;

      // link the corresponding et_trees from level i-1 to j-1
      for(int k=i-1; k<j; k++)
      {
        et_link(source(e),target(e),e,k,this);
      }
    }
  }
}
```
}

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.11 Constructor

There are five optional parameters which can be given to the constructor in order to adapt the data structure to a special input situation. These are ml_rebuild_bound which specifies the bound for newly inserted edges on the highest level before that level is rebuilt, and n_levels which specifies the number of levels. Moreover the user can prescribe the different bounds which are used in the replacement algorithm for a deleted forest edge. Note that the asymptotic analysis may no longer be valid if you choose your own number of levels or bounds.

Internal Functions[72] + ≡

```
{
 dyn_con::dyn_con(dc_graph& G, int ml_reb_bound, int n_levels,
                  int edges_to_samp, int small_w, int small_s)
 // constructor, initializes the dynamic connectivity data structure
 // if ml_reb_bound >= 1 it specifies rebuild_bound[max_level] (default is 5000)
 // if n_levels > 0 then it specifies the number of levels (default O(log n))
 // if edges_to_samp >= 0 then it specifies edges_to_sample (default 32 log^2 n)
 // if small_w >= 0 then it specifies small_weight (default log^2 n)
 // if small_s >= 1 then it specifies small_set (default 16 log n)
 {
   // --- initialize random numbers ---
   struct timeval dummy1;
   struct timezone dummy2;
   gettimeofday(&dummy1,&dummy2);
   srandom(dummy1.tv_sec+dummy1.tv_usec);

   // --- initialize the constants ---
   Gp = &G;
   int log_n = 0;
   for(int i = G.number_of_nodes(); i; i /= 2) log_n++;

   if(small_w>=0) small_weight = small_w;
   else           small_weight = log_n * log_n;

   if(edges_to_samp>=0) edges_to_sample = edges_to_samp;
   else                 edges_to_sample = 32 * log_n * log_n;

   if(small_s>=1) small_set = small_s;
   else           small_set = 16 * log_n;

   if(n_levels > 0) max_level = n_levels - 1;
   else
   {
     max_level = 6 * log_n;
     for(int k=4; (k<ml_reb_bound) && (max_level>=2); k *= 2, max_level--);
   }

 #ifdef DEBUG
   cout << "|V(G)|          = " << G.number_of_nodes() << "\n";
   cout << "max_level       = " << max_level << "\n";
   cout << "edges_to_sample = " << edges_to_sample << "\n";
   cout << "small_set       = " << small_set << "\n\n";
 #endif

   // --- initialize dummy nodes ---
   et_dummy = new et_node_struct(this,nil);
   ed_dummy = new ed_node_struct(nil);

   // --- initialize the edge lists ---
   non_tree_edges = new list<edge> [max_level+1];
   tree_edges = new list<edge> [max_level+1];

   // --- initialize added_edges ---
   added_edges = new int[max_level+1];
   for(i=0; i<=max_level; i++) added_edges[i] = 0;

   // --- initialize rebuild_bound ---
   rebuild_bound = new int[max_level+1];
   int bound;
   if(ml_reb_bound>=1) bound = ml_reb_bound;
   else                bound = 5000;
   for(int k=max_level; k>=0; k--)
   {
     rebuild_bound[k] = bound;
     if(bound < MAXINT/2) bound *= 2;  // double the bound if possible
   }

   // --- initialize the nodes ---
   node u;
   forall_nodes(u,G)
   {
     G[u] = new dc_node_struct();
     G[u]->act_occ = new et_node[max_level+1];
     G[u]->adj_edges = new ed_tree[max_level+1];
     for(i=0; i<=max_level; i++)
     {
       G[u]->act_occ[i] = new et_node_struct(this,u,i,true);
       G[u]->adj_edges[i] = nil;
     }
   }

   // --- initialize the edges ---
   edge e;
   forall_edges(e,G)
   {
     G[e] = new dc_edge_struct();
```

```
      if(!connected(source(e),target(e),0)) insert_tree(e,0,true);
      else                                  insert_non_tree(e,0);
    }

  #ifdef STATISTICS
    // --- initialize statistics ---
    n_ins = 0;
    n_del = 0;
    n_query = 0;
    n_connected = 0;
    n_ins_tree = 0;
    n_del_tree = 0;
    n_replace = 0;
    rep_succ = 0;
    rep_big_cut = 0;
    rep_sparse_cut = 0;
    rep_empty_cut = 0;
    rep_small_weight = 0;
    n_sample_and_test = 0;
    n_get_cut_edges = 0;
    n_ins_non_tree = 0;
    n_del_non_tree = 0;
    n_move_edges = 0;
    edges_moved_up = 0;
    edges_moved_down = 0;
  #endif
  }
  }
```

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

8.6.12 Destructor

The destructor deallocates all additional memory used for the dynamic connectivity data structure, but it does not delete the nodes and edges of the graph.

Internal Functions[73] $+\equiv$

```
 {
  dyn_con::~dyn_con()
  {
    // first delete all edges in the data structure (not in G)
    edge e;
    forall_edges(e,*Gp)
    {
      if(tree_edge(e))
      {
        delete_tree(e);
        for(int j=0; j<=max_level; j++) delete[] (*Gp)[e]->tree_occ[j];
        delete[] (*Gp)[e]->tree_occ;
      }
      else delete_non_tree(e);

      delete (*Gp)[e];
      (*Gp)[e] = nil;
    }

    // delete fields (edge lists are empty, no need to clear() them)
    delete[] non_tree_edges;
    delete[] tree_edges;
    delete[] added_edges;
    delete[] rebuild_bound;

    // delete the et_nodes and the information at the nodes of G
    node v;
    forall_nodes(v,*Gp)
    {
      // per node of G only its active occurrence at each level is left
      for(int i=0; i<=max_level; i++) delete (*Gp)[v]->act_occ[i];

      delete[] (*Gp)[v]->act_occ;
      delete[] (*Gp)[v]->adj_edges;

      delete (*Gp)[v];
      (*Gp)[v] = nil;
    }
  }
  }
```

This macro is defined in definitions 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, and 73.
This macro is invoked in definition 55.

Bibliography

[1] G. M. Adel'son-Velskii and Y. M. Landis. An algorithm for the organization of information. *Soviet. Math. Dokl.*, 3:1259 – 1262, 1962.

[2] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. 30th Symp. on Foundations of Computer Science*, pages 540 – 545, 1989.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[4] D. Eppstein, Z. Galil, and G. F. Italiano. Improved sparsification. Technical Report 93-20, Dept. of Inf. and Comp. Sc., Univ. of Calif., Irvine, CA 92717, 1993.

[5] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. In *Proc. 33rd Symp. on Foundations of Computer Science*, pages 60 – 69, 1992.

[6] D. Eppstein, Z. Galil, G. F. Italiano, and T. H. Spencer. Seperator based sparsification for dynamic planar graph algorithms. In *Proc. 25th Symp. on Theory of Computing*, pages 208 – 217, 1993.

[7] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13:33 – 54, 1992.

[8] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14:781 – 798, 1985.

[9] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Symp. on Foundations of Computer Science*, pages 8 – 21, 1978.

[10] M. Rauch Henzinger and V. King. Randomized dynamic algorithms with polylogarithmic time per operation. To appear in *Proc. 27th Symp. on Theory of Computing*, 1995.

[11] K. Mehlhorn and S. Näher. Leda, a library of efficient data types and algorithms. Technical Report TR A 04/89, Universität des Saarlandes, FB 10, 1989.

[12] S. Näher. *The LEDA User Manual, Version 3.1.* Max Planck Institute for Computer Science, 66123 Saarbrücken, Germany, 1995.

[13] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *JACM*, 32:652 – 686, 1985.