Reference Manual for Generic Interval Tree Data Structures

Vikas Kapoor *　　Alexander Wolff *

Revision: 1.1, Date: 1998/01/22

*Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Takustraße 9, 14195 Berlin, Germany, {kapoor | awolff}@inf.fu-berlin.de

Introduction

The interval tree stores intervals in such a way that those intersecting a query interval can be reported efficiently. One of the applications of the interval tree is the rectangle intersection problem, i.e. given n axis-parallel rectangles in the plane, report for each rectangle all rectangles which intersect this rectangle. An $\mathcal{O}(k + n \log n)$ algorithm for this problem can be sketched as follows: initialize the interval tree with the intervals (on the y-axis) to which the left sides of the rectangles correspond. Then sweep horizontally across the plane. When the sweep line hits a rectangle, query with the corresponding interval and insert it into the interval tree. An interval is deleted when the sweep line leaves the corresponding rectangle.

Here we implement a static version which must be initialized with the endpoints of all intervals it is supposed to contain afterwards. The preprocessing, namely sorting the points and building up an empty balanced tree, takes $\mathcal{O}(n \log n)$ time, where n is the total number of intervals. Inserting intervals can then be done in $\mathcal{O}(\log n)$ time, deleting in constant time, while a query needs $\mathcal{O}(k + \log n)$ steps, where k is the number of intervals reported. Since every interval is stored once in the interval tree, only linear storage is needed.

Chapter 1

Interval Tree

Definition

Class *GenIntervalTree<...>* stores intervals in such a way that those intersecting a query interval can be reported efficiently.

#include <gen_interval_tree.h>

Template Parameters

The class *GenIntervalTree<...>* is templated with the following parameters:

parameter	*task*	*requirement*
`ForwardIt`	access to input intervals	STL forward iterator
`LeftDA`	access to left endpoint of an interval	read data accessors
`RightDA`	access to right endpoint of an interval	read data accessors

Table 1.1: template parameters of the class `GenIntervalTree`

Inherits From

GenIntervalTree<...> inherits from the base class *GenIntervalTree_user_intersection<...>* which differs from *GenIntervalTree<...>* only in that it needs a further template parameters to define an intersection between two intervals, see Chapter 2. This class hierarchy is necessary due to the fact that most compilers cannot handle default template parameters yet.

The constructor of *GenIntervalTree<...>* calls the constructor of the base class by plugging data accessors into the template class which implements the compare function object described in Chapter 3.

Types

GenIntervalTree<...>:: LeftDA::value_type value_type;

Type of the endpoints of the intervals to be stored in the interval tree.

GenIntervalTree<...>:: Interval_position

Position of an interval in the interval tree needed to delete it in constant time.

GenIntervalTree<...>:: Iterator

Type of the iterator that allows to traverse the intervals intersecting a given interval.

Creation

GenIntervalTree<...> *interval_tree(ForwardIt beg,*
ForwardIt end,
LeftDA lda = LeftDA(),
RightDA rda = RightDA())

creates the object *interval_tree* of type *GenIntervalTree<...>*.

Modifiers

Interval_position

interval_tree.insert(ForwardIt interval)

inserts the given interval in the *interval_tree*.

bool *interval_tree.erase(Interval_position& interval)*

deletes the given interval from *interval_tree*.

Access Functions

Iterator *interval_tree.begin(ForwardIt& interval)*

returns an iterator referring to the first interval intersecting the given interval in *interval_tree*.

Iterator *interval_tree.end()* returns an iterator which marks the end of a sequence of intervals intersecting a given interval.

Chapter 2

Interval Tree with User-Supplied Intersection Definition

Definition

The class *GenIntervalTree_user_intersection*<...> is similar to the class *GenIntervalTree*<...> with the only difference that the user is expected to provide a further template parameter which defines an intersection between two intervals.

#include <*gen_interval_tree.h*>

Template Parameters

The class *GenIntervalTree_user_intersection*<...> is templated with the following parameters:

parameter	*task*	*requirement*
`ForwardIterator`	access to input intervals	STL forward iterator
`LeftDA`	access to left endpoint of an interval	read data accessors
`RightDA`	access to right endpoint of an interval	read data accessors
`Compare`	compare function object for the endpoints of the intervals	function object

Table 2.1: template parameters of the class `GenIntervalTree_user_intersection`

Creation

GenIntervalTree_user_intersection<...> *interval_tree_user_inters*(*ForwardIt beg,*
ForwardIt end,
LeftDA lda = LeftDA(),
RightDA rda = RightDA(),
Compare com = Compare())

creates the object *interval_tree_user_inters* of type *GenIntervalTree_user_intersection*<...>.

Chapter 3

Comparing the Endpoints of the Intervals

Definition

Given the endpoints x and y of one of the ends of two intervals, a function object of type *Comp<DA>* returns *true* iff $x < y$.

#include <gen_interval_tree.h>

Template Parameters

The class *Comp<DA>* is templated with a data accessor *DA* whose *value_type* must be that of the endpoints of the intervals to be compared.

Types

Comp<DA>:: typedef DA::value_type value_type;

Type of the endpoints of the input intervals.

Creation

Comp<DA> *comp*;

creates the object *comp* of type *Comp<DA>*.

Operations

bool *comp(value_type x, value_type y)*

returns *true* iff $x < y$.

Chapter 4

Example

The following example illustrates the use of interval tree.

```
#include <iostream.h>
#include <vector.h>
#include <gen_interval_tree.h>

/* type of an interval */
struct Interval
{
  double left, right;
  Interval(double l, double r):  left(l), right(r)
  {}
  Interval()
  {}
};

/* input operator for an interval: */
istream &operator>> (istream &in, Interval& interval)
{
  in >> interval.left >> interval.right;
  return in;
}

/* output operator for an interval: */
ostream &operator<< (ostream &out, Interval const &interval)
{
  out << "(" << interval.left << ", " << interval.right << ")";
  return out;
}

/* data accessors for the left and right endpoints of the intervals */
struct LeftDA { typedef double value_type; };
struct RightDA { typedef double value_type; };

/* the corresponding data accessing functions */
template <class It>
inline double get(LeftDA da, It const &it) { return (*it).left; }
```

```
template <class It>
inline double get(RightDA da, It const &it) { return (*it).right; }

main()
{
  cout << endl << "         Testing the interval tree" << endl << endl;

  /* type definitions */
  typedef vector<Interval> Interval_vector;
  typedef typename Interval_vector::iterator Interval_vector_it;
  typedef GenIntervalTree<Interval_vector_it, LeftDA, RightDA> Interval_tree;
  typedef typename Interval_tree::Interval_position ID;
  typedef typename Interval_tree::Iterator TreeIt;

  /* read the input intervals */
  Interval_vector interval_ls;
  Interval interval;
  cout << "Enter the endpoints of the intervals: " << endl <<
    "(enter an interval of length zero to mark the end of the input" <<
    " intervals)"<< endl;
  while (cin >> interval && (interval.left != interval.right))
    interval_ls.push_back(interval);

  /* output the read intervals */
  cout << "  Interval No.    " << "Interval" << endl;
  int i = 0;
  Interval_vector_it intervals_end = interval_ls.end();
  for (Interval_vector_it intervals_beg = interval_ls.begin();
       intervals_beg != intervals_end; intervals_beg++)
    cout << "      " << i++ << "           " << *intervals_beg << endl;
  cout << endl;

  /* declare an object of type interval tree */
  Interval_tree interval_tree(interval_ls.begin(), interval_ls.end());
  ID *del_info = new ID[interval_ls.size()];

  /* update on and report from the interval tree */
  cout << "i-insert, d-delete, r-report and number of interval:" << endl;
  char c;
  while (cin >> c)
  {
    cin >> i;
    switch (c)
      {
      case 'i':
        {
          del_info[i] = interval_tree.insert(&interval_ls[i]);
        }
        break;
      case 'd':
        {
          interval_tree.erase(del_info[i]);
        }
        break;
```

```
    case 'r':
      {
        TreeIt end = interval_tree.end();
        TreeIt begin = interval_tree.begin(&interval_ls[i]);
        for(; !(begin == end); ++begin)
          cout << *(*begin) << endl;
      }
      break;
    }
  }
  delete[] del_info;
}
```