The Fifth DIMACS Implementation Challenge: Data Type Definitions and Specifications

C.C.McGeoch

June 4, 1996

1 Last Modified 4 June 1996

The following lists changes to previous versions of this document:

- 4/6. Notice to dictionary implementors/users: Section 4 contains a new discussion of assumptions about the valid lifetime of an item reference. Read the paragraph about **Persistence of item references.**
- 4/6. The description of the *lkp* line for the dictionary driver is modified.
- 3/6. Fixed typos. In a few places *pr_type* was supposed to be *in_type*, etc. The priority type for DIMACS tests is *double*, not *real*.
- 3/6. Section 5 new, correct, information on obtaining files.
- 3/6. Functions *infoval*, *prioval*, and *keyval* each have a new parameter pointing to the data structure. For example, **ky_type keyval (it_type item)** has been changed to **ky_type keyval (dict_ptr D, it_type item)**.
- 3/6. The function *infoval* now has a declaration like **in_type *infoval (dict_ptr D, it_type item)** (with the star added), since *in_type* is assumed to be a string for the tests.
- 3/6. The sample priority queue driver provided by DIMACs has an extra field in the *pqh* line. The specs haven't changed — this is the kind of minor variation from specs that we believe will be inevitable. Implementors are responsible for documenting such changes and making suitable modifications to the test set.

- 3/6. The implementations of priority queue and dictionary test drivers provided by DIMACS cover more operations than those previously listed.
- 3/6. The test drivers provided by DIMACS can handle an empty third field in *ins* lines, rather than an explicit "0" value, when the item name is not being used. Either way is OK.

2 Introduction

The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) invites participation in an Implementation Challenge to compare implementations of three basic data structures: *Priority Queues*, *Dictionaries*, and *Multi-Dimensional Point Sets*. Two kinds of point sets, static and dynamic, are described.

This document contains specifications of each abstract data type, as well as format specifications for a set of testbed problems for each data structure to be announced later. Implementors of data structures are expected to develop independent research projects which are presumably more extensive than the testbed problems.

Who Can Participate. We expect that most participants will be professional computer scientists and graduate students with expertise in algorithms, data structures, and implementation. However the Challenge is open to any interested group or individual. We especially invite participation from groups involved in application projects that use one or more of the data structures.

The Timeline. Individuals or groups who wish to participate in the Challenge should send an email message to `challenge5@dimacs.rutgers.edu` to register and get on the mailing list. We request that each participant submit a brief (one-paragraph) Project Proposal sometime before **January 31, 1996** (although proposals will be accepted any time). The steering committee will review each proposal, make suggestions, and notify participants when redundant projects are planned. In **April 1996** we will request progress reports from participants to help in planning the testbed specifications. By **May 1996** the testbed suite will be announced, and implementors will be expected to carry out the specified trials. Ten-page abstracts describing the research results will be submitted for review sometime around **August 1996**; accepted papers will be presented at a DIMACS workshop held in **October 1996** and full papers will subsequently be published by the American Mathematical Society.

The next three sections define the abstract data types and describe standard interfaces for comparing and testing implementations of each data structure. Note that although a C language syntax is used here for convenience, the Implementation Challenge does not specify a particular programming language. Participants who implement in C should obey the syntax conventions here. Participants are encouraged to implement in C++ and to follow LEDA specifications which are reproduced in the Appendix. Implementors using other languages should submit specifications to DIMACS early so that they can be made available to others. Particular application studies may require extentions or semantic variations on the basic operation sets given here; please notify the Challenge committee of any such changes so that they may be passed on to other participants.

In these C specifications, some common names are used in different data structures. If an application program requires different instantiations of these data structure schemas, a prefix should be added to all required variables, types, and functions to distinguish one instantiation from another. The prefixes should be of the form *pq.*, *dc.*, *ps.*, or *ss.* (for static point sets) with additional one-digit identifiers when necessary. For example, an application program that uses a priority queue, a dictionary with integer keys, and a dictionary with character keys would contain variables *pq.item*, *dc1.item*, and *dc2.item*, type declarations for *pq.it_type*, *dc1.it_type*, and *dc2.it_type*, and call to functions *pq.insert*, *dc1.insert*, and *dc2.insert*.

3 Priority Queues

A C language implementation of a Priority Queue requires the following variables and types.

- An instance of a priority queue is called Q and has type *pq_ptr*, which is a C pointer type. The variable Q contains the address of the priority queue. A priority queue contains items.
- The type of an item in the priority queue is *it_type*. The *item* variable is assumed to behave like a pointer or index, and to have some pre-defined *NULL_ITEM* value. However, the underlying type can be defined by the implementor.
- Each item being referenced by *item* has two parts. The first part is a priority *prio* which is a linearly-ordered type called *pr_type*. The second part is an information element *info*, which is of type *in_type*.

.

General assumptions and preconditions. The operations supported by the Priority Queue are listed below. It is assumed that one call to *construct_pq* occurs before before all other calls, and that the pointer Q subsequently contains the address of the priority queue. It is assumed that a call to *destruct_pq* occurs after all references to Q in the program. Also, in all cases where *item* is passed as a parameter, it is assumed that *item* is defined and refers to an item in Q.

Note that the C specifications require that the pointer Q be passed by value, not by reference. If a function implementation changes the address of the priority queue data structure, a second level of indirection will be required. That is, Q should point to a pointer containing a modifiable address of the priority queue.

pq_ptr construct_pq (). Create an instance of the priority queue and initialize it to the empty priority queue. Return a pointer to the priority queue.

it_type insert (pq_ptr Q, pr_type prio, in_type info). Insert a new item having the specified priority and information, and return the item reference.

it_type find_min (pq_ptr Q). Return an item reference for an item having minimal priority in Q, or return *NULL_ITEM* if Q is empty. The item is not removed from Q.

pr_type del_min (pq_ptr Q). Remove an item having minimal priority in Q, and return the priority of the item. Although ties may be broken arbitrarily, the item that this function chooses to delete must be the same one *find_min* would return. *Precondition*: Q is not the empty queue.

void decrease_p (pq_ptr Q, it_type item, pr_type prio). Make *prio* the new priority of *item*. **Precondition**: Priority *prio* is not larger than the current priority of *item*.

pr_type prioval (pq_ptr Q, it_type item). Return the priority of *item*.

in_type infoval (pq_ptr Q, it_type item). Return the information part of *item*. (When *in_type* is a string, use *infoval.)

int size (pq_ptr Q). Return the number of items in Q.

void clear (pq_ptr Q). Make Q the empty priority queue by removing all items.

pq_ptr destruct_pq (pq_ptr Q). De-allocate all memory associated with Q, and return the pointer value *null*. This function contains an invocation of *clear*, which removes all items from the priority queue. If the implementation has dynamic memory elements that remain after the call to *clear*, this function de-allocates them; otherwise nothing further happens.

3.1 Comparing Priority Queue Implementations

Participants are encouraged to develop application programs that require priority queues, and to evaluate different implementation strategies on these applications.

In order to provide a standard interface for exchanging and comparing implementations across different languages, we describe a "Priority Queue Test Driver" and an input file format for the driver. Applications programs should be able to generate procedure call traces in the file format. Data structure implementations are called by drivers that read the files and perform the specified invocations. A C language driver will be available at DIMACS; participants using other languages are invited to submit drivers for redistribution.

Data types. For the purposes of comparative testing, item priorities will be non-negative double-precision reals: that is, $pr\_type = double$. The info part of an item is assumed to be a string containing (at most) k characters, with particular string lengths to be named later. For example, tests may be requested with $in\_type = str128$ and with $in\_type = str4$.

Unique priorities. The Priority Queue specification allows priorities to be non–unique. However in these comparative tests we wish to avoid situations where a procedure-call trace produced by an application program is incorrect for some implementations because of the way ties are broken. (For example a *decrease_p* may specify an item that has been removed in one implementation but not in another.) Therefore for the tests we require that all items in the priority queue have distinct priorities.

In application programs that generate function-call traces, integer priorities can be made distinct by letting the decimal part of a real priority correspond to a time-stamp recording the insertion time of the item. For example, two items both having integer priority 3 in the real application might be recorded as having priorities 3.114 and 3.005 in the trace.

Item names. A call to the *decrease_p* operation requires an item reference as a parameter. For the comparative tests, items may be given *index* or *name* values which are integers in the range $1 \ldots MAXNAMES$. An item name is associated with an item at insert time, and the association remains valid throughout the lifetime of the item. The driver program uses an array to convert index values to item references. Applications programs should generate integer "names" for items where necessary.

Input File Specifications. An input files for a Priority Queue test contains a sequence of lines each terminated by an *eoln* character. Each line begins with a three-character line code, specifying either a file header, a comment, or a type of procedure call. Other values may appear on the line, as specified below. Example test files will be available online from DIMACS; see Section 6 for more information.

pqh The priority queue header line is the first line in the file. It begins with the three-character code and contains an integer N describing the maximum index value used in the file. You can set this to 0 if the test doesn't ever use the **dcr** command.

(Note: the driver implementation provided by DIMACS also reads an integer M giving the max number of items in the priority queue. This is not part of the specifications, but is typical of the small changes that might be required by specific implementations. Implementors are asked to clearly document such changes from the specifications.)

com Comment lines begin with the three-character string *com* and then contain at most 100 characters of text. These lines are inserted for human readibility and are ignored by the driver.

ins An insert line causes an invocation of the *insert* function. Besides the three-character code this line contains a real priority *pri* and an optional field for integer index *inx*. A missing index value, or one set to 0, is ignored by the driver, and can be used for an item that never appears in a subsequent *decrease_p* operation.

fmn A find-min line contains only the code *fmn*. It causes an invocation of the *find_min* function.

dmn A delete-min line causes an invocation of the *del_min* function.

dcr The decrease-priority line contains the *dcr* code, a real priority *pri*, and an index *inx*. This line causes an access of the index array to get an *item* value, followed by a call to *decrease_p* using *pri* and *item*.

prv The prioval line contains the three character code and an integer item name. The *prioval* function is invoked.

inv The infoval line contains the three character code and an integer item name. The *infoval* function is invoked.

siz This line causes a call to *size*.

clr This line causes a call to *clear*.

The Driver. A pseudo-code description of the driver program appears below. A C version of this program is available from DIMACS: see Section 6 for information on obtaining the file. Participants using other languages are invited to develop similar drivers and submit them for public distribution.

```
/*---Constants and Variables ---*/
 MAXNAMES:  constant giving maximum index value in array
       N:  integer giving max index used in input
itemindex:  array [1..MAXNAMES] of type it_type
     inx:  integer index to itemindex array
       Q:  pointer to the priority queue, of type pq_ptr
     cmd:  three-character command code
     pri:  priority of type real
    item:  item of type it_type
   dummy:  a dummy character string of type strk

/*---Code---*/
Q = construct_pq( );
while (not end of file) {
       read (cmd);
       case cmd of {
         "pqh" : { read(N);
                  if (N > MAXNAMES) report array bound error; exit; }
         "com" : { do nothing; }
         "ins" : { read(pri, inx);
                  (append inx to end of dummy);
                  item = insert (Q, pri, dummy);
                  if (inx != 0) itemindex[inx] = item; }
         "fmn" : { item = find_min (Q); }
         "dmn" : { pri = delete_min (Q);}
         "dcr" : { read(pri, inx);
                  item = itemindex[inx];
                  decrease_p (Q, pri, item); }
         "prv" : { read(inx);
                  prioval(Q, itemindex[inx]);}
         "inv" : { read (inx);
                  infoval (Q, itemindex[inx]); }
         "siz" : { size(Q); }
         "clr" : { clear(Q); }
       } /* end case; */
```

```
    read input to next eoln;
}/*end while*/
Q = destruct_pq (Q);
```

The current DIMACS program prints a trace of procedure calls and their results. A later version of the driver will record counts and runtime statistics for key operations; participants will be asked to insert appropriate counting instructions in their data structure implementations.

4 Dictionaries

C languages implementations of Dictionaries require the following variables and types.

- The variable D contains the address of a dictionary. The type of D is *dict_ptr*, which is a C pointer type. A dictionary contains items.
- The variable *item* is of type *it_type*. The *item* variable behaves like a pointer or index, and is assumed to have a pre-defined *NULL_ITEM* value. However the choice of base type is left to the implementor.
- Each item contains two parts. The first is a *key* of some linearly-ordered type *ky_type*. The second is an information part *info* of type *in_type*.

General assumptions and preconditions. It is assumed that one call to *construct_dict* occurs before all other calls, and that D subsequently contains the address of the dictionary. It is assumed that one call to *destruct_dict* occurs after all references to the dictionary in a program. It is assumed that for each distinct *key* value there is at most one item in the dictionary. Also, in all cases where *item* is passed as a parameter, it is assumed that *item* is defined and refers to an item in the dictionary.

Note that in the C specifications given here, the variable D is passed by value, not by reference. If a function is implemented that changes the address of the dictionary, then a second level of indirection is required: that is, D should point to a modifiable pointer containing the address of the dictionary.

Persistence of item references. Both the *insert* and *lookup* operations return item references; the *delete_item* function deletes a dictionary element according to its item reference value. How long is an item reference assumed to be valid? We recognize two reasonable strategies for implementors and users of dictionaries.

1. A dictionary implementation may guarantee that item references are *persistent*: that is, the value returned by the *insert* function at insert-time remains correct throughout the lifetime of the item. A dictionary implementor who wishes to guarantee persistence must avoid copying any item from one "location" to another during insertion, deletion, and lookup operations. For example, a function *swap (itema, itemb)* could not use a standard three-way-copy because previously-assigned references to *itema* and *itemb* would become invalid.

2. A dictionary implementation may not support persistent item references; that is, the implementation is assumed to be correct only if every *delete_item* is immediately preceeded by a *lookup* of that item.

We request that dictionary implementors and test-data produces *clearly state* the assumptions their projects require. Dictionary implementors should mention, in comments, whether item references are guaranteed persistent; and test-data producers should mention whether *delete_items* are always preceeded by *lookups*.

dict_ptr construct_dict ( ). Create a dictionary and initialize it to the empty dictionary. Return the address of the dictionary.

it_type insert (dict_ptr D, ky_type key, in_type info). Insert the item with *key* and *info* values in the dictionary. If there is already an item having identical key value in the dictionary, then it is overwritten with this new item.

it_type lookup (dict_ptr D, ky_type key). Return the item having the given key value, or return *NULL_ITEM* if the item is not in the dictionary.

void delete (dict_ptr D, ky_type key). Delete the item having the given key value. If the item is not in the dictionary, then nothing happens.

void del_item (dict_ptr D, it_type item). Delete the specified item from the dictionary. This function can be used right after *lookup* to avoid a redundant search. *Precondition:* The item must be in the dictionary.

ky_type keyval (dict_ptr D, it_type item). Return the key value of the item. (When *ky_type* is a string, use *keyval).

in_type infoval (dict_ptr D, it_type item). Return the info value of the item. (When *in_type* is a string, use *infoval.)

int size (dict_ptr D). Return the number of items in the dictionary.

void clear (dict_ptr D). Make *D* the empty dictionary by removing all items.

dict_ptr destruct_dict (dict_ptr D) . De-allocate all memory associated with D, and return the pointer value *null*. This function contains an invocation of *clear*, which removes all items from the dictionary. If the implementation has dynamic memory elements that remain after the call to *clear*, this function de-allocates them; otherwise nothing further happens.

4.1 Comparing Dictionary Implementations

Participants are encouraged to develop application programs that use dictionaries, and to evaluate different implementation strategies for the applications. In order to provide a standard interface for comparing implementations in different languages, we describe a "Dictionary Test Driver" and an input file format for the driver. Applications programs should be able to generate function call traces in the format given here. Data structure implementations can be called by drivers that read the files and perform the specified sequence of procedure calls.

Data types. For the purposes of comparative testing, two kinds of key types will be used. In some tests, item keys are assumed to be non-negative integers; that is, *ky_type* = *int*. In other tests, an item key is assumed to be a string containing k characters: for example *ky_type* = *str128*. Specific values of k will be named later so that participants can study the effect of key size on performance (for example participants may be asked to run separate tests with $str4$ and $str128$.)

In all tests, the info type *in_type* is assumed to be a character string of length k, with specific values of k to be named later.

Item names. The *del_item* operation requires an item reference as a parameter. For the comparative tests, items are given *index* values which are integers in the range $1 \ldots$ *MAXNAMES*. The driver program uses an array to convert the index values to particular item references.

Input File Specifications Input files for Dictionary tests contain a sequence of lines, each terminated with an *eoln* characer. Each line begins with a three-character line code, and may contain other values depending on the code. The line codes have the following meanings.

dch The dictionary header line is the first line in the file. Besides the three-character code, the line contains a integer N giving the maximum index value used in the file.

com Comment lines begin with the three-character code and then contain text. These lines are inserted for human readability and are ignored by the driver.

ins An insert line causes an invocation of the *insert* function. Besides the three-character code, this line contains a non-negative integer key *key*, and an optional index value *inx*. An missing index value, or one set to 0, is ignored by the driver and can be used for items that never appear in subsequent *del_item* operations.

lkp A lookup line causes an invocation of the *lookup* function. This line contains an integer *key* and and optional index integer *inx*. If the index is missing or set to 0, then the item reference returned by lookup is ignored by the driver (fine for persistent dictionaries). If a nonzero index is present, the returned item reference is stored by the driver (suitable for non-persistent dictionaries).

dlk A delete line causes an invocation of the *delete* function. This line contains an integer *key* value.

dli A delete-item line causes an invocation of the *del_item* function, which takes an item reference rather than a key as a parameter. This line contains an index value **inx** that names an item already inserted in the dictionary.

kyv This line contains an index value and causes an invocation of *keyval*.

inv This line contains an index value and causes an invocation of *infoval*.

siz This causes a call to the *size* function.

clr This causes a call to *clear*.

The Driver. A pseudo-code description of the Dictionary Driver program appears below. A C version of this program is available DIMACS: see Section 6 for information on obtaining the file. Participants using other languages are invited to develop drivers that can read input files and generate procedure calls as specified here. DIMACS will make such drivers available as they arrive.

```
/*---Constants and Variables---*/
 MAXNAMES: constant giving maximum index in array
        N: integer giving maximum index in input
itemindex: array [1..MAXNAMES] of type it_type
      inx: integer index to itemindex array
        D: pointer to the dictionary, of type dict_ptr
      cmd: three-character command code
```

```
     key:  integer key
    item:  item of type it_type
   dummy:  a dummy character string of type strk

/*---Code---*/
Q = construct_dict ( );
while (not end of file) {
      read (cmd);
      case cmd of {
        "dch" : { read(N);
                  if (N > MAXNAMES) report array bound error; exit; }
        "com" : { do nothing;}
        "ins" : { read(key, inx);
                  item = insert (D, key, dummy);
                  if (inx != 0) itemindex[inx] = item; }
        "lkp" : { read ( key , inx);
                  item = lookup (D, key);
                  if (inx != 0) itemindex[inx] = item; }
        "dlk" : { read (key);
                  delete (D, key);}
        "dli" : { read (inx);
                  item = itemindex[inx];
                  del_item (D, item); }
        "kyv" : { read (inx);
                  keyval(D, inx); }
        "inv" : { read (inx);
                  infoval(D, ins); }
        "siz" : { size (D); }
        "clr" : { clear(D); }
       } /* end case; */
       read input to next eoln;
}/*while*/
Q = destruct_dict (Q);
```

A later version of the driver will will record counts and runtime statistics for key operations. Participants will be asked to insert appropriate counting instructions in their data structure implementations.

5 Multi-Dimensional Point Sets

This section contains a description of the abstract data type Three-Dimensional Point Set, followed by a Static Point Set variation, which does not support dynamic insertion of points. Participants are invited to develop implementations for either or both variations.

Participants who wish to implement higher-dimensional point sets may do so by modifying the definition of *pt_type*. Specifications of other variables and functions need not be modified for higher dimensions, although of course the code will change. Participants may develop implementations either for fixed dimensions or for dimensionalities specified by a compile-time parameter *DIM*.

In applications where different point sets of different dimensionalities are used, prefixes of the form *ps2_*, *ps4_* (the digit denoting the dimensionality) should be attached to all variables, types, and functions associated with the different data structures. Similarly, the prefix *ss3_* denotes a static point set of dimension three.

5.1 Dynamic Point Sets

The following variables and types are used in Dynamic Three-Dimensional Point Sets implemented in C.

- The variable S contains the address of the point set. The type of S is *ps_ptr*, which is a C pointer type. A point set contains items.
- The variable *item* is of type *it_type*. The variable *item* behaves like a pointer or index, and has some predefined *NULL_ITEM* value. However, the choice of base type is left to the implementor.
- Each item contains two parts. The first part is a three-dimensional point *point* of type *pt_type*, which is an array of 3 integers indexed $0, 1, 2$. The second part is an information unit *info* which is of type *in_type*. Points are unique: that is, for each distinct point value there is at most one item in the set S.

General assumptions and preconditions. It is assumed that a call to *construct_ps* occurs before all other calls, and that S subsequently contains the address of the point set. It is assumed that a call to *destruct_ps* occurs after all operations using the point set. In cases where *item* is passed as a parameter, it is assumed that *item* is defined and refers to an item in S.

Note that in the C specifications below, the pointer S is passed by value, not by reference. In implementations where a function changes the address of the point set, a second level of indirection is required: that is, S should point to a pointer containing a modifiable address of the point set. Point sets support the following operations.

ps_ptr construct_ps ( ). Create an empty point set. Return the address of the point set.

it_type insert (ps_ptr S, pt_type point, in_type info). Insert the point *point* and its associated *info* in the point set. If there is already an item having this value of *point*, the new information over-writes the old.

ps_item lookup (ps_ptr S, pt_type point). Return the item in the set having the given point value, or return *NULL_ITEM* if no such item exists.

void delete (ps_ptr S, pt_type point). Delete the item having the given point value. If the item is not in the set, nothing happens.

void del_item (ps_ptr S, it_type item). Delete the specified item from the set. This function can be used after a *lookup* to avoid extra searches. **Precondition**: The item must exist in the set.

it_type nearest (ps_ptr S, pt_type point). Return the item that is nearest to the specified point, breaking ties arbitrarily. A double-precision Euclidean distance computation should be used. If the point set is empty, return *NULL_ITEM*. Note that the specified point need not be an element of the set.

ps_ptr range_search (ps_ptr S, pt_type lo, pt_type hi). The point *lo* specifies a lower bound in each dimension. The point *hi* specifies an upper bound value in each dimension. This function calls *construct_ps* to create a new point set R, and then inserts into R all items in S having point values within the given bounds (including the bound values) in all dimensions. If no such points exist then R remains the empty set. The function returns the address of R.

pt_type pointval (it_type item). Return the point value of the item.

in_type infoval (it_type item). Return the info value of the item.

int size (ps_ptr S). Return the number of points in the point set.

void clear (ps_ptr S). Make S the empty set by removing all items.

ps_ptr destruct_ps (ps_ptr S). De-allocate all memory associated with S, and return the pointer value *null*. This procedure contains an invocation of *clear*, which removes all items from the point set. If the implementation has dynamic memory elements that remain after the call to *clear*, this function de-allocates them; otherwise nothing further happens.

5.2 Static Point Sets

The specification of the *Static Three-Dimensional Point Set* data type is identical to that above, except for the following changes.

- A *pointlist* is an array indexed from 0 to L-1, for some maximum list size L. The array type is *pl_type*. Each element of the array is a record containing a a *point* of type *pt_type* and an *info* of type *in_type*.
- The **ps_ptr construct_ss (pl_type pointlist, int n)** operation creates a point set containing pointlist elements indexed by 0 through $n-1$. The operation returns the address of the point set. This operation replaces the *construct_ps* operation defined earlier.
- The **insert** operation defined in the previous section is not supported by static point sets.
- The **delete** and **del_item** operations are optional for static point sets. Participants may choose whether to support these operations.
- Otherwise, the static point set data type should support all operations given in the previous section.

5.3 Comparing Point Set Implementations

Participants are encouraged to develop application programs that use multi-dimensional point sets and to evaluate different implementation strategies for the applications. In order to provide a standard interface for comparing implementations in different languages, we describe a "Point Set Test Driver" and an input file format for the driver. Applications programs should be able to generate procedure call traces in the format given here. Data structure implementations are called by drivers that read the files and perform the specified sequence of procedure calls.

For the purposes of comparative testing, the info part of an item is assumed to be a string containing k characters: that is *in_type* = *strk*. Specific values of k will be named later so that participants can study the effect of item record size on performance (for example participants may be asked to run separate tests with $str8$ and $str128$.)

The *del_item* operation requires an item reference as a parameter. For the comparative tests, items may be given *index* values which are integers in the range $1 \ldots MAXNAMES$. This corresponds to the *name* of the point, not to its location in D-space. The driver program uses an array to convert the index values to particular item references.

Input File Specifications Input files for Point Set tests contain a sequence of lines, each terminated with an *eoln* character. Each line begins with a three-character line code, and may contain other values depending on the code. The line codes have the following meanings.

psh The point set header line is the first line in the file. Besides the three-character code, the line contains a value N giving the maximum index used in the file, and a value D giving the dimensionality of the point set.

The driver is intended to be used for arbitrary dimension, specified by a compile-time parameter *DIM*. The input value D is compared to *DIM*. When $D < DIM$, the driver will pad with coordinate values of zero to produce higher-dimensional points. When $D > DIM$ only the first *DIM* coordinate values in the input will be used.

com Comment lines begin with the three-character code **com** and then contain text. These lines are inserted for human readibility and are ignored by the driver.

ins An insert line causes an invocation of the *insert* function (an alternative meaning is described below for static point sets). Besides the three-character code, this line contains D integers corresponding to coordinates of the point, followed by an integer index *inx*. An index value of 0 is ignored by the driver and can be used for items that never appear in a subsequent *del_item* operation.

lkp A lookup line causes an invocation of the *lookup* function. Besides the three-character code, this line contains D integer coordinate values.

dlp A delete-point line causes an invocation of the *delete* function. This line contains D integer coordinate values.

dli A delete-item line causes an invocation of the *del_item* function, which takes an item reference rather than a point as a parameter. This line contains an integer index *inx* that names an item already inserted in the point set.

nbr A nearest-neighbor line causes an invocation of the *nearest* function. This line contains D integers specifying a point.

rng A range-search line causes an invocation of the *range_search* function. This line contains D integers corresponding to the *lo* point, followed by D integers corresponding to the *hi* point.

The Driver. A pseudo-code description of the Point Set driver program appears below. A C version of this program will soon be available from DIMACS: see Section 6 for information on obtaining the file. Participants using other languages are invited to develop drivers that can read input files and generate procedure calls as specified here. DIMACS will make such drivers available as they arrive.

```
/*---Constants and Variables---*/
MAXNAMES:  constant giving range of index values
    DIM:  constant giving the dimensionality of the functions.
      N:  input integer giving the max index value in the input
      D:  input integer giving dimensionality of the input
      S:  pointer to the point set, of type ps_ptr
itemindex: array [0..MAXNAMES] of type it_type
    inx:  integer index to itemindex array
    cmd:  three-character command string
  point:  a point of type pt_type, which is an array (0..DIM-1) of integers.
   item:  an item of type it_type
  dummy:  a dummy character string of type strk, containing k characters

/*---Code---*/
pt_type function getpoint (int D, int DIM ) {
 Reads all D points, and pads or truncates to produce DIM points.
} /*end getpoint*/

Q = construct_ps ( );
while ( not end of file ) {
      read ( cmd );
      case cmd of {
         "psh" : { read( N, D );
                  if (N > MAXNAMES) report array bound error; exit; }
         "com" : { do nothing;}
         "ins" : { point = getpoint(D, DIM);
                  item = insert (S, point, dummy);
                  read (inx);
                  if (inx != 0) itemindex[inx] = item; }
         "lkp" : { point = getpoint(D, DIM);
                  item = lookup (S, point);}
         "dlk" : { point = getpoint(D, DIM);
                  delete (S, point);}
         "dli" : { read (inx);
```

```
                item = itemindex[inx];
                del_item (S, item); }
        "nbr" : { point = getpoint(D, DIM);
                item = nearest (S, point);
              }
        "rng" : { lo = getpoint(D, DIM);
                hi = getpoint(D, DIM);
                R = range_search (S, lo, hi);
                R = destruct_ps (R);
              }
      } /* end case; */
      read input to next eoln;
}/*while*/
S = destruct_ps ( S );
```

A later version of the driver will contain a data structure to record counts of key operations performed during function invocations. The driver will contain code to initialize, accumulate, and report these counts; participants will be asked to insert appropriate counting instructions in their data structure implementations.

Test of Static Point Set Implementations. The input file specifications for Static Point Sets are identical to those given above. The driver program is modified as follows

- The driver assumes that all **ins** lines in the input file come immediately after the **psh** line and before any other lines (ignoring comments). The driver first reads the **ins** lines and constructs a *pointlist* and an *itemindex*. When all the *ins* lines are read, the driver invokes *construct_ss* to initialize the point set.
- A compile-time switch determines whether the driver ignores the **dlk** and **dli** delete lines.

6 Obtaining Files from DIMACS

The web page **http://cs.amherst.edu/c̃cm/challenge5/index.html** contains pointers to documents and software relating to the Fifth DIMACS Challenge. Files are available through the web page or by anonymous ftp from **cs.amherst.edu**, in directory **pub/dimacs**.

For questions, register to participate, or to get on the mailing list, send email to **challenge5@dimacs.rutgers.edu**.

Appendix: LEDA Specifications

This Appendix reproduces excerpts from the *The LEDA User Manual, Variation R 3.3. beta*, by Stefan Näher and Christian Uhrig. The sections here give specifications for Priority Queues, Dictionaries, and Two-Dimensional Point Sets. (Because the excerpts do not modify the source text, section numbers do not match the original, and some references in the original are unresolved here.)

Full LEDA documentation is available by anonymous ftp from **ftp.mpi-sb.mpg.de**, in directory **pub/LEDA/articles**. For more information check the WWW page **http://www.mpi-sb.mpg.de/guide/staff/uhrig/leda.html** or the following reference: K. Mehlhorn and S. Näther. "LEDA: A platform for combinatorial and geometric computing." *Communications of the ACM*, (38)1, January 1995.

7 Priority Queues (p_queue)

1. Definition

An instance Q of the parameterized data type *p_queue*<P, I> is a collection of items (type *pq_item*). Every item contains a priority from a linearly ordered type P and an information from an arbitrary type I. P is called the priority type of Q and I is called the information type of Q. The number of items in Q is called the size of Q. If Q has size zero it is called the empty priority queue. We use <p, i> to denote a *pq_item* with priority p and information i.

2. Creation

p_queue<P, I> Q;

creates an instance Q of type *p_queue*<P, I> and initializes it with the empty priority queue.

3. Operations

P	Q.prio(*pq_item it*)	returns the priority of item *it*. *Precondition*: *it* is an item in Q.
I	Q.inf(*pq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in Q.

pq_item	Q.insert($P\ x,\ I\ i$)	adds a new item <x, i> to Q and returns it.
pq_item	Q.find_min()	returns an item with minimal priority (nil if Q is empty).
P	Q.del_min()	removes the item it = Q.find_min() from Q and returns the priority of it. *Precondition*: Q is not empty.
void	Q.del_item(*pq_item* it)	removes the item it from Q. *Precondition*: it is an item in Q.
void	Q.change_inf(*pq_item* it, $I\ i$)	makes i the new information of item it. *Precondition*: it is an item in Q.
void	Q.decrease_p(*pq_item* it, $P\ x$)	makes x the new priority of item it. *Precondition*: it is an item in Q and x is not larger then $prio(it)$.
int	Q.size()	returns the size of Q.
bool	Q.empty()	returns true, if Q is empty, false otherwise.
void	Q.clear()	makes Q the empty priority queue.

4. Implementation

Priority queues are implemented by Fibonacci heaps [?]. Operations insert, del_item, del_min take time $O(\log n)$, find_min, decrease_p, prio, inf, empty take time $O(1)$ and clear takes time $O(n)$, where n is the size of Q. The space requirement is $O(n)$.

5. Example

Dijkstra's Algorithm (cf. section **??**)

8 Priority Queues with Implementation Parameter (_p_queue)

1. Definition

An instance of type *_p_queue*<$P, I, impl$> is a priority queue implemented by data type *impl*. *impl* must be one of the priority queue implementations listed in section **??** or a user defined data structure fulfilling the specification given in section **??**. Note that the priority type P must linearly ordered.

9 Dictionaries (dictionary)

1. Definition

An instance D of the parameterized data type *dictionary*<K, I> is a collection of items (*dic_item*). Every item in D contains a key from the linearly ordered data type K, called the key type of D, and an information from the data type I, called the information type of D. The number of items in D is called the size of D. A dictionary of size zero is called the empty dictionary. We use <k, i> to denote an item with key k and information i (i is said to be the information associated with key k). For each $k \in K$ there is at most one $i \in I$ with <k, i> $\in D$.

2. Creation

dictionary<K, I> D;

creates an instance D of type *dictionary*<K, I> and initializes it with the empty dictionary.

3. Operations

K	D.key(*dic_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in D.
I	D.inf(*dic_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in D.
dic_item	D.insert(K k, I i)	associates the information i with the key k. If there is an item <k, j> in D then j is replaced by i, else a new item <k, i> is added to D. In both cases the item is returned.
dic_item	D.lookup(K k)	returns the item with key k (nil if no such item exists in D).

I	D.access(K k)	returns the information associated with key k. *Precondition*: there is an item with key k in D.
void	D.del(K k)	deletes the item with key k from D (null operation, if no such item exists).
void	D.del_item(*dic_item it*)	removes item *it* from D. *Precondition*: *it* is an item in D.
void	D.change_inf(*dic_item it*, I i)	makes i the information of item *it*. *Precondition*: *it* is an item in D.
void	D.clear()	makes D the empty dictionary.
int	D.size()	returns the size of D.
bool	D.empty()	returns true if D is empty, false otherwise.

4. Implementation

Dictionaries are implemented by randomized search trees [?]. Operations insert, lookup, del_item, del take time $O(\log n)$, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n)$. Here n is the current size of the dictionary. The space requirement is $O(n)$.

5. Example

We count the number of occurrences of each string in a sequence of strings.

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

main()

{ dictionary<string,int> D;

  string s;

  dic_item it;

  while (cin >> s)

  { it = D.lookup(s);

    if (it==nil) D.insert(s,1);

    else D.change_inf(it,D.inf(it)+1);
```

```
  }
  forall_items(it,D) cout << D.key(it) << " : " << D.inf(it) << endl;
}
```

10 Dictionaries with Implementation Parameter (_dictionary)

1. Definition

An instance of type *_dictionary*<$K, I, impl$> is a dictionary implemented by data type *impl*. *impl* must be one of the dictionary implementations listed in section **??** or a user defined data structure fulfilling the specification given in section **??**. Note that depending on the actual implementation *impl* the key type K must either be linearly ordered or hashed.

Example
Using a dictionary implemented by skiplists to count the number of occurrences of the elements in a sequence of strings.

```
#include <LEDA/_dictionary.h>
#include <LEDA/impl/skiplist.h>
main()
{
  _dictionary<string,int,skiplist> D;
  string s;
  dic_item it;
  while (cin >> s)
  { it = D.lookup(s);
    if (it==nil) D.insert(s,1);
    else D.change_inf(it,D.inf(it)+1);
  }
  forall_items(it,D) cout << D.key(it) << " : " << D.inf(it) << endl;
}
```

11 Two Dimensional Point Sets

1. Definition

An instance S of the parameterized data type *point_set<I>* is a collection of items (*ps_item*). Every item in S contains a two-dimensional point as key (data type *point*), and an information from data type I, called the information type of S. The number of items in S is called the size of S. A point set of size zero is said to be empty. We use $<p,i>$ to denote the item with point p, and information i. For each point p there is at most one item $<p,i> \in S$. Beside the normal dictionary operations, the data type *point_set* provides operations for rectangular range queries and nearest neighbor queries.

2. Creation

point_set<I> S;

creates an instance S of type *point_set<I>* and initializes S to the empty set.

3. Operations

point	S.key(*ps_item it*)	returns the point of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*ps_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
ps_item	S.insert(*point p*, *I i*)	associates the information i with point p. If there is an item $<p,j>$ in S then j is replaced by i, else a new item $<p,i>$ is added to S. In both cases the item is returned.
ps_item	S.lookup(*point p*)	returns the item with point p (nil if no such item exists in S).
ps_item	S.nearest_neighbor(*point q*)	returns the item $<p,i> \in S$ such that the distance between p and q is minimal.
list<ps_item>	S.range_search(*double* $x0$, *double* $x1$, *double* $y0$, *double* $y1$)	returns all items $<p,i> \in S$ with $x_0 \le p$.xcoord() $\le x_1$ and $y_0 \le p$.ycoord() $\le y_1$.

list<ps_item>	S.convex_hull()	returns the list of items containing all points of the convex hull of S in clockwise order.
void	S.del(*point* p)	deletes the item with point p from S.
void	S.del_item(*ps_item* it)	removes item it from S. *Precondition*: it is an item in S.
void	S.change_inf(*ps_item* it, I i)	makes i the information of item it. *Precondition*: it is an item in S.
list<ps_item>	S.all_items()	returns the list of all items in S.
list<point>	S.all_points()	returns the list of all points in S.
void	S.clear()	makes S the empty point_set.
bool	S.empty()	returns true iff S is empty.
int	S.size()	returns the size of S.

4. Implementation

Point sets are implemented by a combination of two-dimensional range trees [**?**, **?**] and Voronoi diagrams. Operations insert, lookup, del_item, del take time $O(\log^2 n)$, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n \log n)$. A range_search operation takes time $O(k+\log^2 n)$, where k is the size of the returned list. A nearest_neighbor query takes time $O(n^2)$, if it follows any update operation (insert or delete) and $O(\log n)$ otherwise. Here n is the current size of the point set. The space requirement is $O(n^2)$.