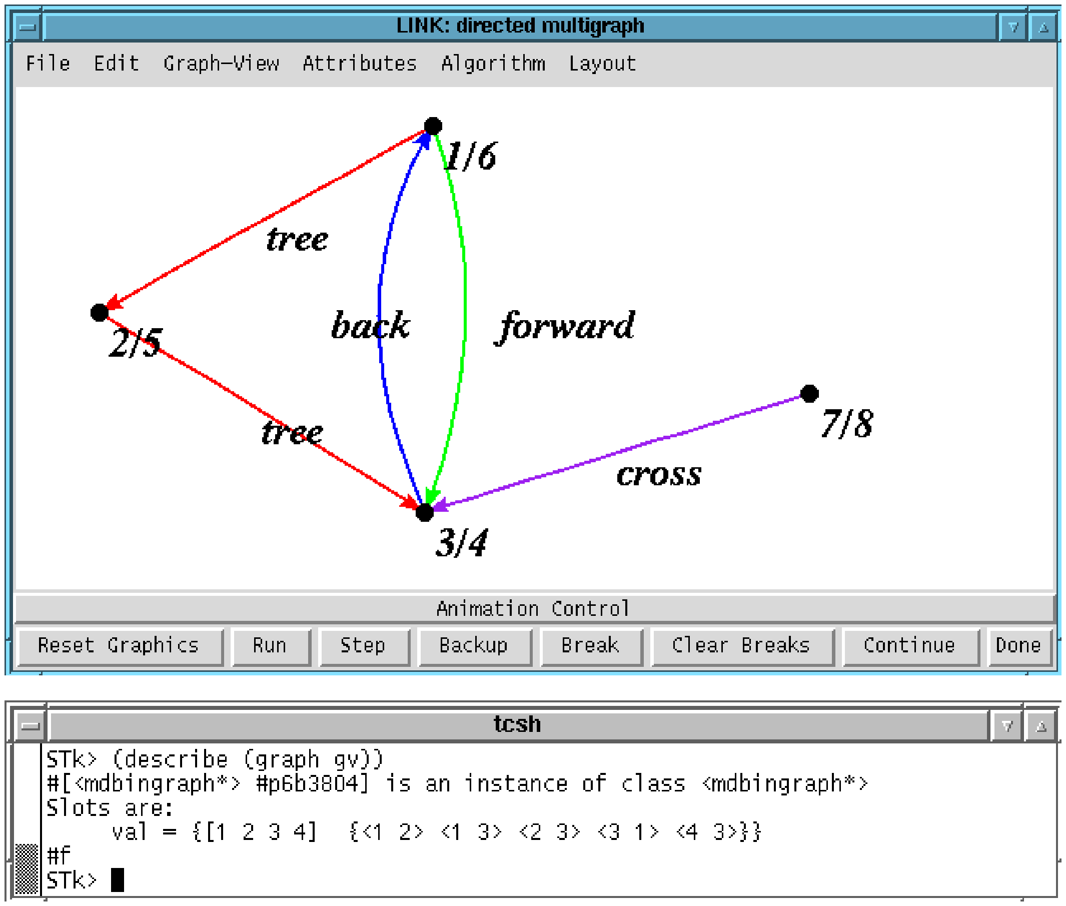
Link 1.2

Jonathan W. Berry
Elon College, N.C.
berryj@numen.elon.edu

January 3, 1998


LINK: directed multigraph
File Edit Graph–View Attributes Algorithm Layout
1/6
tree
2/5
back
forward
7/8
tree
cross
3/4
Animation Control
Reset Graphics
Run
Step
Backup
Break
Clear Breaks
Continue
Done
tcsh
STk> (describe (graph gv))
#[<mdbingraph*> #p6b3804] is an instance of class <mdbingraph*>
Slots are:
val = {[1 2 3 4] {<1 2> <1 3> <2 3> <3 1> <4 3>}}
#f
STk>

Contents

List of Figures

Part I
Getting Started with Link

1 Initial Remark

This manual accompanies the initial release of the LINK system, an environment for combinatorial computing. The system is useful not only as a collection of C++ libraries, but as an computing environment for manipulating and visualizing graphs and hypergraphs. It is menu driven for accessibility and includes a Scheme command-line interface for power and flexibility. This Scheme interface is based on Erick Gallesio's STk package [6], so LINK inherits all of STk's functionality. Some examples of this include regular expression processing and socket objects; however, no STk functionality will be discussed in this manual. See the STk manual [6] for details.

All of the coding examples in this manual were included via automatic mechanisms which run the code and produce LaTeXoutput. This first release includes many examples and lists of methods without much textual description. The manual will continue to fill out during the first few years of public LINK usage, and updates will be included with subsequent releases.

2 Forward

The high-level plan for LINK was conceived discussed by the four primary investigators: *Nathanial Dean*, *Mark Goldberg*, *Greg Shannon*, and *Steven Skiena*. They established guidelines describing the system's capabilities [2], obtained the necessary funding and assembled the teams responsible for the implementation. The coordinator of the project, and one of the primary designers and developers is Dr. Jonathan Berry.

This version of LINK is not in the public domain, but can be used freely for research and teaching. Users should be aware of the copyright agreements of the various software packages which it uses.

This version of LINK has been partially funded by NSF grant CCR-9214487 and ONR award 400x116yip01.

Questions and comments should be addressed to Jonathan W. Berry, the project coordinator, at berryj@dimacs.rutgers.edu

LINK is a cooperative project of DIMACS, Los Alamos National Laboratory, Indiana University, Rensselaer Polytechnic Institute, and The State

University of New York at Stony Brook.

The many people who have contributed to the LINK effort are listed on page 198.

3 Downloading the System

0.1 Getting Link

LINK is maintained at The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) at Rutgers University. The system and this manual are available on the World-Wide Web at `http://dimacs.rutgers.edu` and via anonymous ftp at /ftp://dimacs.rutgers.edu/pub/berryj. For example, the following commands will obtain it:

```
%ftp dimacs.rutgers.edu  (log in as ''anonymous'')
>cd pub/berryj
>get link.tar.gz
%gunzip link.tar
%tar xf link.tar
```

The result is a directory hierarchy containing all the essential elements of LINK.

0.2 Installing Link

LINK depends on the following software:

- **g++2.6.3/libg++2.6.2 (or later)** The system has been tested with this version of the gnu compiler (Note: for versions 2.6.X, compiler patch is required in order to instantiate templates. The patch is found at: *ftp.cygnus.com:pub/g++/gcc-2.6.3-template-fix*. Your systems administrator may have it already installed). LINK has been designed to be portable to other compilers, but the actual ports have not been tried, so there is probably more work in store before it builds with any other compiler.

- **flex and bison** These should be available on your system.

- **Scheme/Tk (STk)** This free software is available from `ftp://kaolin.unice.fr/pub` or `ftp://cs.nyu.edu/pub/local/fox/stk`, and runs on many Unix systems. Ports are being made to various Microsoft Windows versions, and Link will follow eventually.

There is a file called INSTALL in the root directory of the LINK distribution giving the installation details. Often the process will be as simple as the following:

```
%./configure
%make
%make libs
```

However, small changes to a file called *Config.mk* may be necessary. NOTE: Link uses some large template instantiation files. This puts a strain on PC's running linux which will probably require the user or administrator to increase the swap space. The compilation may take many hours on machines with 16MB or less. The setup will be improved to better accomodate PC installations, possibly in the near future.

The disk space used by the system varies. If debugging information is requested, all object files and libraries are left to sit around, and the executable is not stripped, the distribution can approach 100MB. However, if -O2 is used during compilation and the executable is stripped, the system can take approximately 27MB. From there, the removal of object files and libraries drastically reduces the space requirements.

0.3 Mailing List

There is a mailing list for LINK users. This list will be useful for reporting bugs and sharing tips. Mail should be sent to

`link-list@dimacs.rutgers.edu+.`

To subscribe (unsubscribe), send mail to:

`link-list-request@dimacs.rutgers.edu+.`

with "subscribe" ("unsubscribe") on the subject line.

1 Introduction

Over the last 10 years there have been numerous attempts to develop software for discrete mathematics that would benefit pure research, application-driven research, and/or pedagogic concerns. This software has focused on combinatorics, graph theory, computational geometry, or some combination of these areas. However, each package has had shortcomings which has prevented it from becoming a generally useful tool. Some packages are too specialized, some are not robust, some are inefficient, and some are hard to learn. Nevertheless, the fact that these systems continue to be designed in spite of previous efforts indicates that there is a real need for a comprehensive and robust system which can serve a variety of users, from applications to research, to education. The magnitude of such a project exceeds the efforts made so far by the researchers and research groups responsible for the existing systems. A good discussion of software packages for discrete math is found in [4, 14, 9].

From the standpoint of a theoretician or an applications researcher, there are several features which characterize a useful software tool for their work. First of all, it is essential to have a set of libraries that are efficient, easy to use, robust, and extensible. Secondly, a sophisticated GUI and command-line interface are vital for interaction, visualization, testing, and experimentation. Having a large set of standard algorithms and data structures is also important. Educators trying to produce the next generation of researchers would also find all of these features very useful. But even with the improvements in object-oriented design and GUI technology these goals present an enormous challenge. The LINK project is an attempt to balance the trade-offs inherent in these varied goals and produce a usable and reliable tool for theorists, applied researchers and educators.

In July 1991, four researchers met at the SIAM conference for Applied Mathematics in Washington D.C. and discussed the possibility of creating a large software package for discrete mathematics. Each had already directed similar projects. Greg Shannon of Indiana University had organized the development of GraphLab [12], Nate Dean was largely responsible for the package NETPAD at Bellcore [10] , Steve Skiena at State University at New York at Stony Brook had produced Combinatorica on top of Mathematica [13], and Mark Goldberg and his students at Rensselaer Polytechnic Institute had implemented SetPlayer [1]. Though these packages have many useful features, some overlapping and some not, none was complete or without significant drawbacks depending on the user and the application. These researchers wanted a more comprehensive package that would add new features and retain the positive aspects of their previous efforts. They wanted a package that could easily be used for intensive research and applications, and yet have the capability to be used for pedagogic purposes. There would be numerous tradeoffs in trying to satisfy these somewhat competing goals. The hope was that they could find a satisfactory balance that would fulfill each of these goals in turn.

Later that year and into the next they, along with their students, did further research and wrote a proposal for a research grant to develop such a package. The grant stipulated that they would receive organizational support from DIMACS (Center for Discrete Mathematics and Theoretical Computer Science). The late Daniel Gorenstein joined the project as the principal investigator with co-PI's Nate Dean, Mark Goldberg, Greg Shannon and Steve Skiena.

Independently, at Los Alamos National Laboratory (LANL) there were several projects where such a package would be of considerable use, and researchers and programmers at LANL began implementing many components similar to software that the PI's had developed. Mark Goldberg was on sabbatical at LANL and was involved in one of these projects. He and the other PI's convinced Vance Faber (the group leader of CIC-3, the Computer Research and Applications Group of the Communications, Information, and Computing Division) that a more general tool would better fulfill the needs of these and similar projects at LANL. The grant was accepted by the National Science Foundation in the fall of 1992, and CIC-3 provided staff support and development resources in the form of support for the PIs' students to work on the project at LANL. In addition, Greg Shannon took a leave of absence from IU and worked on LINK at LANL as a contractor.

Initial work on LINK began with meetings in December 1992 between representatives from Indiana University, RPI, Stony Brook and LANL. Many months were spent in designing the graph hierarchy, and how that hierarchy would be represented to the user, and in the implementation of basic data structure classes. Algorithms were implemented using these designs in order to test the ease of programming and the efficiency of the design. Much time was spent dealing with the intricacies of C++ and the immaturity of the compilers available at that time.

In the summer of 1993, students from all the representative institutions worked together with staff from LANL at Los Alamos. During this time, the current *Graph* and *Collection* hierarchies were developed and much of the work on the Tcl interface was done. This command-line interface provided a means of testing the various generators, layouts and algorithms that had been implemented during the spring and summer months.

In November, 1994, Dr. Jonathan Berry, a student of Mark Goldberg's at the time and the designer of the *Collection*hierarchy and co-designer of the *Graph*hierarchy, took over as project coordinator. Plans were made for Dr. Berry to spend a year at DIMACS reworking the LINK interface to be more friendly to students and researchers and preparing the system for its initial July, 1996 release. Dr. Berry will remain project coordinator and primary contact for matters concerning the LINK system at Elon College in North Carolina, though the `berryj@dimacs.rutgers.edu` email address will still be valid.

Part II
Graphical User Interface

1 General

LINK's graphical user interface (GUI) has been written in such a way that extensions and changes are much easier than would be possible with an XWindows application. Erick Gallesio's object-oriented Scheme interface (STk) [6] to John Ousterhout's Tk graphics package [11] makes this possible. It is not necessary to know any further details of these packages in order to use LINK effectively, but the LINK User's Manual (see Page **??** describes how to make effective use of some power features of the system. For example, it is possible to program series of computations such as groups of experiments, and systematic evaluation and/or setting of attributes by writing very small Scheme programs.

The GUI is intended to be usable without documentation. However, we will briefly describe its features below. In our descriptions, we will use the following notation to describe events:

Notation	Expanation
Main Window	The small window that comes up when LINK is run
Graph View	A window in which graphs can be drawn.
Graph Menu	The menu at the top of a graph window.
Menu-option *Menu1-Menu2-option* ...	Select *option* from *Menu* Selecte the *Menu2* option from *Menu1*, then select *option* from *Menu2*. and so on.
$< 1st >$	Click with the first (leftmost) mouse button
$< 2nd >$	Click with the second (middle) mouse button
$< 3rd >$	Click with the third (rightmost) mouse button
$< Shift - 1st >$	Click the first mouse button while holding the shift key down

2 Graph Creation, Copying, and File Operations

2.1 Defining Graphs

2.1.1 Graph Creation and Copying

Graphs of all of the various types supported by LINK can be created from the GUI. LINK's main window has an option called *Graph Windows* explicitly for this purpose. Graph types are selected by proceding through the cascading menus associated with that option. The user's selections provide answers to the three questions listed below, then a window containing a graph of the appropriate type appears.

- graph or hypergraph?
- directed undirected, or mixed?
- multiple edges allowed or not?

It is also possible to redefine the graph associated with a graph view. The *File* menu has the following menu options to change the graph being viewed or bring up new graph views.

- New This option brings up the same cascade of menus used in creating graphs from the main window. The user selects a graph type, and a new empty graph of the appropriate type replaces the the current graph (which is lost).
- *Generate* It is often desirable to call algorithms which generate graphs rather than creating graphs by hand. The *Generate* menu current supports the generation of cycles, complete graphs, grids, and uniform hypergraphs, but adding new generators is straightforward (though as yet undocumented 7/19/96).
- *Clone* Often one wants to compare a slight variation of a graph G with G itself. The *Clone* option will bring up an exact copy of a graph, which can then be modified without affecting the original.
- *Multi-Graph Clone* All graphs in LINK either allow multiple edges or do not. In certain cases, such a collapsing groups of vertices into subgraphs, it is necessary to allow multiple edges. If such an operation is needed for a simple graph, this option will bring up a clone of the graph, except of the corresponding type allowing multiple edges.
- *Simple Graph Clone* Similar to Multi-Graph Clone, except that the new copy of the graph is always a simple graph.

2.2 Loading and Saving Graphs

The *Load* and *Save* options of the graph menu accomplish these functions. Graphs may be expressed in the input graph language with or without their attributes, so saved graphs, when reloaded, retain the same vertex location, color, etc. The graph input language is described in Appendix 2.7.2, but the simple examples follow:

- *Undirected Triangle*

$$\{[123]\{\{12\}\{13\}\{23\}\}\}$$

- *Directed Triangle*

$$\{[123]\{< 12 >< 13 >< 23 >\}\}$$

3 Graph Editing

3.1 Creating Vertices and Edges

Graphs can also be created using the mouse. Clicking the $< 3rd >$ button on the canvas creates vertices, and clicking the same button on two vertices in sequence creates the appropriate edge. If the graph is a variety of hypergraph, then edges containing more than two vertices. This is accomplished by clicking $< 3rd >$ on the first vertex of the edge, $< Shift - 3rd >$ on all intermediate vertices, and finally clicking $< 3rd >$ on the last vertex.

3.2 Moving Vertices and Edges

The $< 2nd >$ button is used to move vertices and edges around. To move an edge, make sure that the edge label is visible. This label is also the control point of a cubic spline, so that when you drag it around, the edge is pulled into an arc. This is necessary for visualizing multigraphs, and generally useful for improving the appearance of drawings.

3.3 Selecting Vertices and Edges

Vertices and edges are selected with the $< 1st >$ button, and their appearance changes upon selection. It is possible to select all of the objects in a rectangular region by clicking $< 1st >$ in an open area of the canvas, then dragging the mouse while holding the button. It is also possible to add to the selection with $< Shift - 1st >$. SHIFT-1ST SHOULD BE A TOGGLE OPERATION, BUT IT IS NOT YET. NEEDS TO BE FIXED.

3.4 Deleting Vertices and Edges

In order to delete a set of objects, simply select them and then select the *Delete* option from the *Edit* menu. An alternative to pulling down the method is simply to type the character "x" on the keyboard.

3.5 Changing Attributes

The *Attributes* menu on the graph menubar allows the GUI user to modify the attributes of the graph. It will be an easy change to allow the user to choose which attribute is displayed as the vertex or edge label, but this is not done as of 7/19/96. However, it is straightforward to modify the label, color, size, and/or width of vertices using the options of this menu.

4 Graph Layout

4.1 Layout of Graphs

Layout algorithms assign coordinate in the world coordinate system (0.0, 0.0) to (1.0, 1.0). The following are selectable from the *Layout* menu of each graph view:

- *Circular* The vertices are laid out in a circle.
- *Component*The user is prompted for two layout algorithms (chosen from this list minus *Component*): an algorithm with which to layout the *component graph*, i.e. the graph obtained by collapsing each component into a single vertex, and an algorithm with which to layout each component.
- *Grid* The user is prompted for the numbers of rows, and columns, then the graph is laid out in a rectangular grid.
- *Random* The vertices are given random locations (NOTE: STk uses Unix rand(), which is not very random (7/19/96)).
- *Spring* The vertices are laid out according to forces calculated so that edges act like rubber bands. CITE?

5 Graph Algorithms

5.1 Algorithm Specifications

The following list enumerates the algorithms which are currently callable from the GUI interface, and those which we anticipate will be included in the July 31st relase of the system. The algorithm animations may be called and viewed, as in any animation system, but the results (in the form of the return value and any attributes set) can be obtained for further use in the Scheme interface. For example, selecting *Kruskal* from the *Algorithm* menu runs Kruskal's minimum spanning tree animation, but the return value can be saved and used by calling the same algorithm from the command-line interface (see the User's Manual on Page **??**). For example, a command to bring up a the spanning tree of a graph g is:

```
(show-labeled-graph (graph (vertices g) (kruskal g))).
```

Each algorithm below is listed by its Scheme command name (and alias if appropriate). The return values can be retrieved by calling the algorithms from Scheme (as above), and the values of the attributes set by algorithms can also be retrieved using Scheme. See the User's Manual for a discussion of this (NOT WRITTEN YET 7/19/96). It would be straightforward to incorporate the retrieval and setting of all fundamental attributes through the graph menu, but this has not been done for all of them yet. However, color, vertex size, edge width, and vertex and edge labels *are* easily changed from the GUI interface.

5.1.1 Searches and Sorts

command name	depth-first-search
alias	dfs
status	in use, animated
input	<graph*>
output	Sequence<Vertex*>
attributes used	starttime finishtime back low pred
attributes set	starttime finishtime back low pred

command name	topological-sort
alias	
status	in use, animated
input	directed, acyclic graph
output	Sequence<Vertex*>
attributes used	finishtime
attributes set	

command name	breadth-first-search
alias	bfs
status	in use, animated
input	<graph*>
output	Sequence<Vertex*>
attributes used	starttime finishtime pred
attributes set	starttime finishtime pred

5.1.2 Components

command name	strongly-connected-components
alias	scc
status	in use, component layout exists
input	directed graph
output	Set<Set<Vertex*>>
attributes used	finishtime pred
attributes set	

command name	biconnected-components
alias	bcc
status	under construction
input	undirected graph
output	Set<Set<Vertex*>>
attributes used	low
	starttime
attributes set	

5.1.3 Shortest Paths

command name	dijkstra
alias	
status	under construction
input	<graph*>
output	Sequence<double>
attributes used	weight distance pred
attributes set	weight distance

command name	floyd-warshall
alias	
status	in use, not animated
input	<graph*>
output	Sequence<double>
attributes used	weight distance pred
attributes set	weight distance

5.1.4 Minimum Spanning Tree

command name	kruskal
alias	
status	in use, animated
input	<graph*>
output	Set<Edge*>
attributes used	weight mark DJSet (disjoint set)
attributes set	DJSet mark (indicates tree edge)

command name	prim
alias	
status	under construction
input	<graph*>
output	Set<Edge*>
attributes used	weight key pred (disjoint set)
attributes set	pred

5.1.5 Network Flows and Matching

command name	ford-fulkerson
alias	
status	drawing board
input	directed graph, source, sink
output	double (the maximum flow)
attributes used	flow capacity partition (disjoint set)
attributes set	flow capacity partition (disjoint set)

command name	goldberg-tarjan
alias	
status	implemented, but not available yet
input	directed graph, source, sink
output	double (the maximum flow)
attributes used	flow capacity partition (disjoint set)
attributes set	flow capacity partition (disjoint set)

command name	bipartite-matching
alias	
status	drawing board, (but simply calls flow alg.)
input	<graph*>
output	Set<Edge*>
attributes used	flow capacity mark
attributes set	flow capacity mark (indicates matching edge)

6 Graph Operations

6.1 Graph Operations

Several primitive operations on graphs (both unary and binary) are available or imminently available from the GUI. Unary operations are selected from the graph menu and are listed below:

6.2 Unary Operations

- *Complement* Trivial, but not available yet (7/19/96).
- *Edge Contraction* The *collapse selected* option of the *Edit* menu replaces all selected items with a single "super vertex" sv. Then (u, sv) is an edge if and only if u was not selected, and there exists a vertex v in the subgraph represented by sv such that (u, v) was an edge. A similar argument holds for (sv, u).

  In particular, an edge is contracted by selecting its endpoints with $< 1st >$, then selecting the *Collapse Subgraph* option or typing s from the keyboard. Super vertices are replaced with the subgraphs they represent by selecting them, then selecting the *expand-selected* option from the *Edit* menu or typing S.
- *Induced Subgraphs* It is very useful sometimes to extract parts of a graph and treat them is graphs themselves. The *induced subgraph* option of the *Edit* menu allows the user to extract the subgraph induced by all selected vertices. This operation can also be invoked by selecting the vertices and then typing i on the keyboard.
- *Edge-Induced Subgraphs* This feature is not implemented yet (7/19/96), but will be used in the same manner as *induced-subgraph*.
- *Line Graph* Not available yet (7/19/96).

6.2.1 Binary Operations

Binary operations are selected from the main window under the *Graph Operations* menu.

- *Graph Isomorphism* Graphs can be tested for isomorphism by selecting the *isomorphism* option, then clicking on two graphs (as instructed by a message window). A window appears indicating whether the graphs are isomorphic or not. The actual isomorphism testing is performed by Brendan McKay's *nauty* system [8].

- *Graph Sum*
- *Graph Product*
- *Lexicographic Product*

Part III

User's Manual

7 Introduction

Let's begin with an example. Assuming that LINK has been properly installed on your system, you can start it by typing `Link` at the command line prompt of your operating system (or by double-clicking the appropriate icon). You may now enter commands and start building graphs. This tutorial will give you several examples so that you can get started right away. When you need to learn the system more thoroughly, the later chapters will provide much more detail.

Our first example is shown in Figure **??**. This example may not be very

```
STk>   (define g (graph))
#[undefined]

STk>   (describe g)
#[<ubingraph*> 40243e34] is an instance of class <ubingraph*>
Slots are:
     val = {[]  {}}
#f
```

Figure 7.1: Our first graph: the empty graph

exciting, but don't worry; things will get more interesting shortly! Even though our graph is not very interesting, there are many things to note about this example. First of all, note that the commands we input are enclosed in parentheses. This is a characteristic of

Scheme, the computer language used by Link. Scheme is a *functional language*. Programs in this type of programming language are generally constructed by *composing* functions together rather than by stringing sequences of statments together. To compose two functions means to use the output of one function as input to the other. For example, in Figure **??**, the function which creates the graph is *graph*, and its output is then passed to the function *define* as an argument. The result of the command is that the symbol *g* new refers to the newly created graph. There are many other ways to create graphs, but we will describe those later. This example shows the simplest way.

A graph without any vertices isn't much fun, so let's add some vertices.

```
STk>   (define h (graph '(1 2 3) '((1) (2 3) (1 2 3))))
#[undefined]

STk>   (describe h)
#[<uhypergraph*> 4023f874] is an instance of class <uhypergraph*>
Slots are:
     val = {[1 2 3]  {{1} {1 2 3} {2 3}}}
#f
```

Figure 7.2: Creating an undirected hypergraph

Link allows you to name your vertices with integers, symbols, or character strings. The example in Figure 7.4 shows the addition of three vertices, one with each type of name. Of course, it might be quite inconvenient to have to add each vertex individually if we want to build a big graph. One solution to this problem is to use the `add-vertices!` command shown in Figure **??**. This command allows you to specify a range of integers, then creates a vertex for each integer in the range.

Another solution to this problem, and one which will be the solution to many problems, is to use the Scheme command language. We can write a short program to add our vertices for us. This program uses a Scheme *do* loop, which will not be described in detail here. There is a tutorial on programming in Scheme in Section **??**. Also, the STk Reference Manual contains more complete information.

Now that we have given our graph some vertices, it is time to add edges. An edge is a collection of vertices. The only type of edge we will discuss here has exactly two vertices, and there is no ordering of the vertices. Such an edge is usually drawn as a line connecting the two vertices. Our next task will be to construct some edges from the vertex set.

First, we must obtain the vertex set from the graph so that we can pick out the vertices we want for our edges. The vertex set of a graph is returned by the *vertices* function. This is shown in Figure **??**. The figure also shows the construction of $\binom{V}{2}$, which is a set containing every subset of the vertex set V which has size 2. This set represents all possible lines connecting vertices in our graph.

```
STk>   (define g (graph '(1 2 3) '((1 2) (2 3) (3 1))))
#[undefined]

STk>   (define h (graph '(1 2 3) '((1) (2 3) (1 2 3))))
#[undefined]

STk>   (define h2 (graph (+ (vertices g) (vertices h))
                  (+ (edges g) (edges h))))
#[undefined]

STk>   (describe h2)
#[<muhypergraph*> 4023bd88] is an instance of class <muhypergraph*>
Slots are:
     val = {[1 2 3]  {{1} {1 2} {1 2 3} {1 3} {2 3} {2 3}}}
#f
```

Figure 7.3: Computing the sum of two graphs

8 Basic Objects

The foundation of LINK is a group of *basic* templated C++ objects. These objects are easily accessible from the Scheme interface, as we will see below.

```
STk> (add-vertex! 1 g)
#[<vertex*> 4023f778]

STk> (add-vertex! 'apple g)
#[<vertex*> 4023f43c]

STk> (add-vertex! "orange" g)
#[<vertex*> 4023f190]

STk> (describe g)
#[<ubingraph*> 40243e34] is an instance of class <ubingraph*>
Slots are:
     val = {[1 apple orange]  {}}
#f
```

Figure 7.4: Adding Vertices One by One

```
STk> (define g (graph))
#[undefined]

STk> (add-vertices! 1 4 g)
#[<set<vertex*>> 4023f76c]

STk> (describe g)
#[<ubingraph*> 40243e34] is an instance of class <ubingraph*>
Slots are:
     val = {[1 2 3 4]  {}}
#f
```

Figure 7.5: Adding Ranges of Vertices

```
STk>  (define g (graph))
#[undefined]

STk>  (do ((i 1 (+ i 1)))
    ((= i 4) g)
    (add-vertex! i g))
#[<ubingraph*> 40243e34]

STk>  (describe g)
#[<ubingraph*> 40243e34] is an instance of class <ubingraph*>
Slots are:
     val = {[1 2 3]  {}}
#f
```

Figure 7.6: A Short Program to Add Vertices

8.1 *Collections*

All of the following methods are generated automatically for each type of collectionn. The instantiation of new types requires recompiling, so the process will eventually be described in the programmer's manual. However, most users should be satisfied with the types currently available, which are the same types as those enumerate for typed lists (Page 90).

(mset-*type*)

Creates an empty C++ multiset to hold elements of type *type*. $O(1)$

(set-*type*)

Creates an empty C++ set to hold elements of type *type*. $O(1)$

(sequence-*type*)

Creates an empty C++ sequence to hold elements of type *type*. $O(1)$

(mset-*type* i_1 i_2 $\ldots$ i_k)

Creates a C++ multiset containing the specified elements. Note: for C++ types which have close Scheme analogs, such as `int`, the elements i_1 i_2 $\ldots$ i_k may be appropriately-typed Scheme objects. $O(n)$

(set-*type* i_1 i_2 $\ldots$ i_k)

Creates a C++ set containing the specified elements. (same note) $O(n)$

(sequence-*type* i_1 i_2 … i_k)

Creates a C++ sequence containing the specified elements. (same note) $O(n)$

(list->mset-*type* *list*)
(list->set-*type* *list*) (list->sequence-*type* *list*)

Given a Scheme list, create a C++ collection containing the elements of the list. $O(n)$

(mset-*type*->list *mset*)
(set-*type*->list *set*) (sequence-*type*->list *sequence*)

Create and return a Scheme list containing the elements of the collection argument. $O(n)$

(collection-int? *col*)

Return #t if *col* is a collection of `int` objects. $O(1)$

(size *col*)

Return the number of elements in *col*. $O(n)$

(null? *col*)

Return #t if *col* has 0 elements. $O(1)$

(member? *value* *col*)

Return *value* if it is in the colleciton *col*; otherwise return #f. $O(n)$

(insert! *value* *col*)

Add the element *value* to colleciton *col*. If *col* does not allow multiple elements, requests for multiple insertion of an element are ignored. Time complexity depends on collection implementation.

(remove! *value* *col*)

Remove the element *value* to colleciton *col*. If *value* occurred k times before the call, it will occur $k-1$ times afterwards. Time complexity depends on collection implementation.

(clear! *col*)

Remove all of the elements from colleciton *col*. $O(n)$

(occurrences *value* *col*)

Remove number of times element *value* occurs in collection *col*. $O(n)$

(eq? *col* col_2 … col_k)

Return true if all argments are the same object. $O(n)$

(eqv? *col* col_2 … col_k)

Return true if all argments are collections with the same elements. $O(n)$

(< *col* col_2 … col_k)

Return true if $col_i < col_j$ whenever $i < j$. Ordering is lexicographic. $O(n)$

($<=$ *col* col_2 ... col_k)

Return true if col_i <= col_j whenever i<= j. Ordering is lexicographic. $O(n)$

($>$ *col* col_2 ... col_k)

Return true if col_i > col_j whenever i> j. Ordering is lexicographic. $O(n)$

($>=$ *col* col_2 ... col_k)

Return true if col_i >= col_j whenever i>= j. Ordering is lexicographic. $O(n)$

(+ *col* col_2 ... col_k)

Return a multiset containing the union of *col* ... col_k, i.e. each occurrence of each element of collection col_i is added to the result. Time complexity depends on the types of the col_i.

(^*col* col_2 ... col_k)

Return a multiset containing the intersection of *col* ... col_k. The number of occurrences of element i in the result is the minimum number of occurrences of i in any col_i. Time complexity depends on the types of the col_i.

(- *col* col_2 ... col_k)

Return a multiset containing the difference of *col* ... col_k. If $r_j(i)$ denotes the number of occurrences of element i in col_j, then the number of occurrences of i in the result is the maximum of 0 and the difference between $r_1(i)$and the sum from 2 to k of $r_j(i)$. Time complexity depends on the types of the col_i.

(subset *col* col_2 ... col_k)

Return #t if *col* $\subseteq col_2 \subseteq col_k$. Time complexity depends on the types of the col_i.

(proper-subset *col* col_2 ... col_k)

Return #t if *col* $\subset col_2 \subset col_k$, where $col_i \subset col_j$ if and only if for each element e in the $col_i \cup col_j$, $r\_i(e)$¡$r\_j(e)$. Time complexity depends on the types of the col_i.

(cartesian-product *set* 1 *set* 2n)

Return the set of all ordered pairs in the cartesian product of *set* 1 and *set* 2n. $O(n^2)$.

(choose *set* k_n)

Return the set of all k-subsets of *set*, where *set* is a LINK *set* class object (not a multiset or sequence). $O(2^n)$.

(power-set *set*)

Return the set of all subsets of *set*, where *set* is a LINK *set* class object (not a multiset or sequence). $O(2^n)$.

The *Collection* hierarchy, documented in the Programmer's Manual, provides LINK with a flexible core of set and sequence classes that interact with each other easily, facilitating the representation of a rich variety of graphs.

8.2 Typed Vectors

All of the following methods are generated automatically for each type of vector. The instantiation of new types requires recompiling, so the process will eventually be described in the programmer's manual. However, most users should be satisfied with the types currently available: `int, char, Vertex, Graph, Edge, Attribute`.

```
STk> (define ms (mset-int 3 2 2 1))
#[undefined]

STk> (describe ms)
#[<mset<int*>> 4023e1d0] is an instance of class <mset<int*>>
Slots are:
     val = {1 2 2 3}
#f

STk> (define s (set-int ms))
#[undefined]

STk> (describe s)
#[<set<int*>> 402123a4] is an instance of class <set<int*>>
Slots are:
     val = {1 2 3}
#f
```

Figure 8.1: Definition of sets and multisets

(vector-*type* *size*)

Creates an empty dynamic array with *size* C++ elements of type *type*. $O(1)$

(vector-*type*->list *vec*)

Create a Scheme list containing the elements of *vec* . Note: for C++ types which have close Scheme analogs, such as `int`, the resulting list elements may be appropriately-typed Scheme objects. $O(n)$

```
STk>   (define ms (mset-int 3 2 2 1))
#[undefined]

STk>   (define sq (sequence-int 3 2 2 1))
#[undefined]

STk>   (describe sq)
#[<sequence<int*>> 4023b248] is an instance of class <sequence<int*>>
Slots are:
     val = <3 2 2 1>
#f

STk>   (define sq2 (sequence-int ms))
#[undefined]

STk>   (describe sq2)
#[<sequence<int*>> 4020ce3c] is an instance of class <sequence<int*>>
Slots are:
     val = <1 2 2 3>
#f
```

Figure 8.2: Definition of sequences

(vector-*type*? *vec*)

Return #t if *vec* is a vector of C++ objects of type *type*. $O(1)$

(vector-ref *vec index*)

Return the element of *vec* at location $O(1)$

(vector-fill! *vec value*)

Fill all of the entries of *vec* with *value*. $O(n)$

(vector-copy *vec*)

Create and return a new copy of *vec* . $O(n)$

(vector-set! *vec* *value*)

Assign *value* to $vect[index]$. $O(1)$

(vector-length *vec*)

Return the length of the longest initialized segment of *vec* . $O(n)$

(eq? vec_1 vec_2 $\ldots$ vec_k)

Return true if all argments are the same vector. $O(n)$

(eqv? vec_1 vec_2 $\ldots$ vec_k)

Return true if all argments are vectors with the same elements. $O(n)$

(< vec_1 vec_2 $\ldots$ vec_k)

Return true if $vec_i < vec_j$ whenever $i< j$. Ordering is lexicographic. $O(n)$

(<= vec_1 vec_2 $\ldots$ vec_k)

Return true if $vec_i <= vec_j$ whenever $i<= j$. Ordering is lexicographic. $O(n)$

(> vec_1 vec_2 $\ldots$ vec_k)

Return true if $vec_i > vec_j$ whenever $i> j$. Ordering is lexicographic. $O(n)$

($>=$ vec_1 vec_2 $\ldots$ vec_k)

Return true if vec_i $>=$ vec_j whenever $i>= j$. Ordering is lexicographic. $O(n)$

(vector-resize *vec* *newsize*)

Change the number of memory locations allocated to *vec* . $O(n + newsize)$

```
STk>   (define m (list-¿mset-int '(1 2 2 3)))
#[undefined]

STk>   (describe m)
#[<mset<int*>> 4023e8fc] is an instance of class <mset<int*>>
Slots are:
     val = {1 2 2 3}
#f

STk>   (define s (list-¿set-int (mset-int-¿list m)))
#[undefined]

STk>   (describe s)
#[<set<int*>> 40211354] is an instance of class <set<int*>>
Slots are:
     val = {1 2 3}
#f

STk>   (define sq (list-¿sequence-int '(3 1)))
#[undefined]

STk>   (describe sq)
#[<sequence<int*>> 4020d0e8] is an instance of class <sequence<int*>>
Slots are:
     val = <3 1>
#f
```

Figure 8.3: Conversions between collections and Scheme lists

```
STk> (define ms (mset-int 1 2 2 3))
#[undefined]

STk> (if (member 2 ms)
      (remove! 2 ms))
#[undefined]

STk> (describe ms)
#[<mset<int*>> 4023e1d0] is an instance of class <mset<int*>>
Slots are:
     val = {1 2 3}
#f
```

Figure 8.4: Membership testing in collections

```
STk>  (define ms (mset-int 1 2 2 3))
#[undefined]

STk>  (define sq (sequence-int 1 2 2 3))
#[undefined]

STk>  (null? ms)
#f

STk>  (null? sq)
#f

STk>  (eq? sq ms)
#f

STk>  (eqv? sq ms)
#t
```

Figure 8.5: Example of overloaded method: null?

```
STk>  (define-generic f)
#[undefined]

STk>  (define-method f ((a ¡collection¡int*¿¿)) 'f)
f

STk>  (f (mset-int 1 2))
f

STk>  (f (set-int 1 2))
f

STk>  (f (sequence-int 1 2))
f

STk>  (f (list-int 1 2))

*** Error:
No applicable method for #[<generic> f]
in call (f #[<list<int*>> 4022e340])
```

Figure 8.6: Defining generic functions which take collections as arguments

```
STk>   (define ms (mset-int 3 2 2 1))
#[undefined]

STk>   (define sq (sequence-int 3 2 2 1))
#[undefined]

STk>   (cond ((¡ ms sq) 'seq-greater)
     (#t        'mset-lesseq))
seq-greater
```

Figure 8.7: Comparing collections

```
STk>  (define ms (mset-int 3 2 2 1))
#[undefined]

STk>  (define s (set-int ms))
#[undefined]

STk>  (define sq (sequence-int ms))
#[undefined]

STk>  (define ms2 (+ ms s sq))
#[undefined]

STk>  (define ms3 (^ ms s sq))
#[undefined]

STk>  (define ms4 (- ms2 ms3))
#[undefined]

STk>  (describe ms2)
#[<mset<int*>> 40239a9c] is an instance of class <mset<int*>>
Slots are:
     val = {1 1 1 2 2 2 2 2 3 3 3}
#f

STk>  (describe ms3)
#[<mset<int*>> 40232a38] is an instance of class <mset<int*>>
Slots are:
     val = {1 2 3}
#f

STk>  (describe ms4)
#[<mset<int*>> 40230a88] is an instance of class <mset<int*>>
Slots are:
     val = {1 1 2 2 2 2 3 3}
#f
```

Figure 8.8: The set primitive operations

```
STk> (define s (set-int 1 2 3))
#[undefined]

STk> (define bin-edges (choose s 2))
#[undefined]

STk> (define hyper-edges (power-set s))
#[undefined]

STk> (describe bin-edges)
#[<set<set<int*>>> 4023d5d0] is an instance of class <set<set<int*>>>
Slots are:
     val = {{1 2} {1 3} {2 3}}
#f

STk> (describe hyper-edges)
#[<set<set<int*>>> 4023c5d4] is an instance of class <set<set<int*>>>
Slots are:
     val = {{} {1} {1 2} {1 2 3} {1 3} {2} {2 3} {3}}
#f
```

Figure 8.9: k-sets and power sets of a set

```
STk> (define v (vector-int 10))
#[undefined]

STk> (describe v)
#[<array<int*>> 402442d8] is an instance of class <array<int*>>
Slots are:
     val = []
#f

STk> (vector-fill! v 0)

*** Error:
vector-fill!: not a vector: #[<array<int*>> 402442d8]

STk> (vector-set! v 1 10)

*** Error:
vector-set!: not a vector: #[<array<int*>> 402442d8]

STk> (describe v)
#[<array<int*>> 402442d8] is an instance of class <array<int*>>
Slots are:
     val = []
#f
```

Figure 8.10: Creating and manipulating vectors

```
STk> (vector-int? v)
#t

STk> (define v2 (vector 10))
#[undefined]

STk> (vector? v2)
#t

STk> (vector-int? v2)
#f
```

Figure 8.11: Typed vs. untyped vectors

```
STk> (describe v)
#[<array<int*>> 402442d8] is an instance of class <array<int*>>
Slots are:
     val = []
#f

STk> (define l (vector-int-¿list v))
#[undefined]

STk> l
()
```

Figure 8.12: Conversion of typed vector to Scheme list

```
STk>  (define v3 (vector-copy v))

*** Error:
vector-copy: bad vector: #[<array<int*>> 402442d8]

STk>  (describe v3)

*** Error:
unbound variable: v3

STk>  (define v3 (vector-int v))
#[undefined]

STk>  (describe v3)
#[<array<int*>> 4029ac50] is an instance of class <array<int*>>
Slots are:
     val = []
#f
```

Figure 8.13: Copying of typed vectors

```
STk>   (describe v)
#[<array<int*>> 402442d8] is an instance of class <array<int*>>
Slots are:
     val = []
#f

STk>   (define v3 (vector-int v))
#[undefined]

STk>   (¡= v v3)

*** Error:
Bad number in a comparison: (a b)

STk>   (vector-set! v 0 1)

*** Error:
vector-set!: not a vector: #[<array<int*>> 402442d8]

STk>   (describe v)
#[<array<int*>> 402442d8] is an instance of class <array<int*>>
Slots are:
     val = []
#f

STk>   (¡= v v3)

*** Error:
Bad number in a comparison: (a b)
```

Figure 8.14: Comparison of typed vectors

8.3 Typed Lists

Generally, algorithms which return collections of data will use objects from the *Collection* hierarchy. However, it is possible for algorithms to return *List*<type> objects, and the latter can be manipulated in a limited way in Scheme. The only necessary manipulations are simply translations to and from standard Scheme lists.

(list->list-*type* *list*)

Given a Scheme list containing the elements of type *type*, create and return a C++ list containing those elements. $O(n)$

(list-*type* ->list *type-list*)

Create a Scheme list containing the elements of *type-list*. $O(n)$

(list-int? *list*)

Return #t if *list* is a C++ list of `int` objects. $O(1)$

9 Graph Objects

9.1 *Graphs*

9.1.1 Graph Creation

(graph)

Create and return an undirected, binary, simple graph.

(graph *vlist elist*)

If *vlist* is a Scheme list containing integer, string, and/or symbol Scheme objects, and *elist* is a scheme list containing sublists of these elements, Create an undirected graph or hypergraph of the appropriate type.

(graph *set* <*vertex* *> *mset* <*edge* *>)

Create an undirected graph or hypergraph of the appripriate type with the given vertex and edge sets.

(graph *set* <*vertex* *> *set* <*set* <*vertex* *>>

Create an undirected graph or hypergraph of the appripriate type with the given vertex and edge sets. NOTE: the edge set is not a multiset since the output of (choose s k) is a set of sets. CAN BE FIXED.

(digraph)

Create and return an undirected, binary, simple graph.

(digraph *vlist elist*)

If *vlist* is a Scheme list containing integer, string, and/or symbol Scheme objects, and *elist* is a scheme list containing sublists of these elements, Create a directed graph or hypergraph of the appropriate type.

(digraph *set* <*vertex* *> *mset* <*edge* *>)

NOT IMPLEMENTED YET.

(digraph *set* <*vertex* *> *set* <*sequence* <*vertex* *>>)

Create a directed graph or hypergraph of the appripriate type with the given vertex and edge sets. NOTE: the edge set is not a multiset since the output of (choose s k) is a set of sets. CAN BE FIXED.

(mbingraph)
(mubingraph)
(mdbingraph)
(mhypergraph)
(muhypergraph)
(mdhypergraph)
(bingraph)
(ubingraph)
(dbingraph)
(hypergraph)
(uhypergraph)
(dhypergraph)

Creates an empty graph of the appropriate type.

(directed-graph *g*)

Return *g* if it is not a directed graph (it is undirected or mixed), otherwise create and return a directed graph which is isomorphic to *g* except that each edge is directed according to lexicographic ordering of its vertices.

(multi-graph g)

Return g if it is a multigraph, otherwise create and return a simple graph which is isomorphic to g except that no edge is duplicated.

(simple-graph g)

Return g if it is not a multigraph, otherwise create and return a multigraph of the corresponding type and isomorphic to g.

(undirected-graph g)

Return g if it is not an undirected graph, otherwise create and return an undirected graph which is isomorphic to $g$when the directions of the edges are ignored.

9.1.2 Graph Modification

(add-vertex! *vertex graph*)

(add-vertices! *lower_bound upper_bound graph*)

(add-edge! *set <vertex *> graph*)

(add-edges! *mset <set <vertex *>> graph*)

(add-edges! *mset* <*sequence* <*vertex* *>> *graph*)

(remove-vertex! *vertex graph*)

(remove-edge! *edge graph*)

(remove-edges! <list of edges> *graph*)

9.1.3 Graph Information

(edges *graph*)

(order *graph*)

(size *graph*)

(vertices *graph*)

(vertex-vector *graph*)

(vertex-ref *graph integer*)

(vertex-rank *vertex graph*)

9.1.4 Graph Vertex Information

(adjacent *vertex vertex*

NOT IMPLEMENTED YET.

(adjacent *vertex vertex graph*

(degree *vertex graph*)

(edge? *edge*

(incident-edges *vertex graph*)

(in-degree *vertex graph*)

(in-incident-edges *vertex graph*)

(out-degree *vertex* *graph*)

(out-incident-edges *vertex* *graph*)

(vertex? *vertex*

9.1.5 Graph Queries

(binary? *graph*)

Returns #t if *graph* is not a hypergraph.

(multigraph? *graph*)

Returns #t if *graph* is of a type allowing multiple edges.

(<mbingraph*>? *graph*)
(<mubingraph*>? *graph*) (<mdbingraph*>? *graph*)
(<mhypergraph*>? *graph*) (<muhypergraph*>? *graph*)
(<mdhypergraph*>? *graph*) (<bingraph*>? *graph*)
(<ubingraph*>? *graph*) (<dbingraph*>? *graph*)
(<hypergraph*>? *graph*) (<uhypergraph*>? *graph*)
(<dhypergraph*>? *graph*)

Returns #t if *graph* is of the appropriate type.

```
STk>   (define g (graph '(1 2 3) '((1 2) (1 3) (3 1))))
#[undefined]

STk>   (define v (vertices g))
#[undefined]

STk>   (describe v)
#[<set<vertex*>> 401ff080] is an instance of class <set<vertex*>>
Slots are:
     val = {1 2 3}
#f

STk>   (define n (neighbors (ref v 0)))
#[undefined]

STk>   (describe n)
#[<set<vertex*>> 402ca6a4] is an instance of class <set<vertex*>>
Slots are:
     val = {2 3}
#f
```

Figure 9.1: Vertex neighbors

9.2 Graph Views

(make <graph-view> :graph *graph* :xpixels *xpixels* :ypixels *ypixels* :layout *layout*)

Create and return a toplevel window containing a view of graph. The keyword arguments are optional, where *graph* is a <graph*> object, *xpixels* and *ypixels* are integer values, and *layout* is a symbol naming the layout algorithm. At this time, the layouts available are: 'circular, 'random, 'spring, and 'bipartite. A copy of *graph* is made and retained in the <graph-view> object.

(name *gv*)

(slot) Holds the name of the graph, which corresponds to the *name* attribute of the *Graph*class in C++.

(graph *gv*)

(slot) Holds the associated <graph*> object, which *name* attribute of the *Graph*class in C++.

(xpixels *gv*)

(slot) Defines the width of the graph's canvas widget. Changing its value causes the graph to be redrawn. Note that the window can be resized using the mouse. Such an operation does not automatically cause the graph to be redrawn (that would be too slow). Instead, there is a "redraw" option in the "Graph-View" menu which modifies xpixels and ypixels.

(ypixels *gv*)

(slot) Defines the height of the graph's canvas widget. The discussion of xpixels pertains to this slot also.

(title *gv*

(slot) Stores the title displayed in the toplevel window of the <graph-view> object (not necessarily the name of the graph).

(graph-toplevel *gv*)

(slot) Holds the Tk toplevel window in which the graph window's menu and canvas are packed.

(graph-canvas *gv*)

(slot) Holds the Tk canvas widget in which the graph's vertices and edges are drawn.

(vertex-table *gv*)

(slot) Holds the STk hash table which allows the user or programmer to search, given a <vertex*> object, for the associated <vertex-item>.

(edge-table *gv*)

(slot) Holds the STk hash table which allows the user or programmer to search, given a <edge*> object, for the associated <edge-item>.

(segment-table *gv*)

(slot) Holds the STk hash table which, allows the user or programmer to search, given a the integer tag of a <line> or <text-time> return the associated <edge-segment-item> object. This is used by the selection and dragging mechanism and may not be of any interest to the user.

(vertex-tag-table *gv*)

(slot) Holds the STk hash table which allows the user or programmer to search, given the tag of an <oval> or <text-item> for the associated <vertex-item> object. This is used by the selection and dragging mechanism and may not be of any interest to the user.

(edge-tag-table *gv*)

(slot) Holds the STk hash table which allows the user or programmer to search, given the tag of an <oval> or <text-item> for the associated <edge-item> object. This is used by the selection and dragging mechanism and may not be of any interest to the user.

(vertices *gv*)

(method) Returns a list of all <vertex-item> objects in *gv* .

(edges *gv*)

(method) Returns a list of all <edge-item> objects in *gv* .

(update-graph-view *gv*)

(method) Updates the attributes of the <graph-view> and its associated <graph*>.

(path-draw *gv*)

(method) Changes the visual appearance of each <edge-item> in the graph to be "path-draw," i.e., a single path of line segments through the member vertices. This is the default edge drawing style.

(star-draw *gv*)

(method) Changes the visual appearance of each <edge-item> in the graph to be "star-draw," i.e., a central edge label with line segments to each member vertex.

(layout-edge *ei* *gv*)

(method) Is used primarily to view hyperedges. It changes the drawing mode of <edge-item> *ei* to be "star-draw," and updates the location of the central edge label.

(layout-edges *gv*)

(method) Calls layout-edge for each <edge-item> in *gv*.

(hide-label *vi*)

(method) Makes the label associated with <vertex-item> *vi* invisible.

(show-label *vi*)

(method) Makes the label associated with <vertex-item> *vi* visible.

(hide-labels *gv*)
(show-labels *gv*)

These methods hide and show all labels associated with the vertices and edges of *gv*.

(hide-vertex-labels *gv*)
(show-vertex-labels *gv*)

(method) Hides all labels associated with the vertices of *gv*.

(hide-edge-labels *gv*)
(show-edge-labels *gv*)

(method) Hides all labels associated with the edges of *gv*.

(reset-label *vi*)
(reset-label *ei*)

(method) Resets the label slots of the given <vertex-item> or <edge-item> to be the name of the associated <vertex*> or <edge*> C++ object.

(reset-labels *gv*)

(method) Calls reset-label for each <vertex-item> and <edge-item> of *gv*.

(reset-colors *gv*)

(method) Resets the colors of each <vertex-item> and <edge-item> to their default values.

(reset-graphics *gv*)

(method) Calls reset-colors and reset-labels.

(vertex-item *vertex gv*)

(method) Returns the <vertex-item> object associated with <vertex*> *vertex* in <graph-view> *gv* .

(edge-item *edge gv*)

(method) Returns the <edge-item> object associated with <edge*> *edge* in <graph-view> *gv*.

(vertex-items *vlist gv*)
(vertex-items $mset - vertex$ *gv*) (vertex-items $set - vertex$ *gv*)

(method) Returns the list of <vertex-item> objects of <graph-view> *gv* associated with the <vertex*> objects in list *vlist* , <mset<vertex*>> $mset - vertex$, or <set<vertex*>> $set - vertex$.

(edge-items *elist* *gv*)
(edge-items $mset - edge$ *gv*) (edge-items $set - edge$ *gv*)

(method) Returns the list of <edge-item> objects of <graph-view> *gv* associated with the <edge*> objects in list *elist* , <mset<vertex*>> $mset - edge$, or <set<vertex*>> $set - edge$.

(graph-view? *gv*)

(method) Returns #t if *gv* is a <graph-view> object.

(replace-graph *g* *gv*)

The graph displayed in graph-view *gv* is replaced by *g*. The former is deleted.

(collapse-subgraph <list-of-vertex-items> *gv*)

The vertices in <list-of-vertex-items> are identified into a subgraph, which can later be expanded. The latter is represented by a "super-vertex," which is returned.

(expand-subgraph *vg* *gv*)

If <vertex-item> *vg* is a super-vertex, replace it with the subgraph is represents. The return value is the set of vertices which comprised the subgraph (or *vg* if it is not a super-vertex.

9.3 Vertex Graphics

(oval-item vg)

(slot) Holds an object of class <guarded-oval>, where the latter prevents the user from changing the "fill" slot, since "color" is the slot which controls the color of the vertex. NOT SETTLED YET - MUST OTHER ATTRIBUTES BE GUARDED TOO?

(txt-item vg)

(slot) Holds the STk <txt-item> which holds the the vertex label. Note that vertex-label and vertex-name are two different concepts. The name is constant, while the label can change to display various information.

(vertex vg)

(slot) Holds the <vertex*> object contained within vg.

(coords vg)

(slot) The viewing coordinates of <vertex-item> vg.

(world-coords vg)

(slot) The world coordinates of <vertex-item> vg.

(size vg)

(slot) The size in pixels of the oval depicting the vertex.

(width vg)

(slot)The width of the outline encircling the vertex-item. The default is 1.

(outline vg)

(slot) The color of the outline encircling the vertex-item.

(color vg)

(slot) The color of the oval depicting the vertex.

(text-color vg)

(slot) The color of the vertex label.

(font vg)

(slot) The font of the vertex label.

(stipple vg)

(slot) The stipple of the oval depicting the vertex. For example, “” is the default value, and ”gray-50” gives the oval a sparser background. See the Tk documentation for other options.

(tags vg)

(slot) The STk tags associated with the vertex-item. This information is contained in a list, and some common values include 'vertex, which is always present in vertices, and 'link:selected-item,' which is added if the user selects the vertex.

(degree vg)

(method) returns the degree of the vertex contained within vertex-item vg.

(random-neighbor *vg*)

(method) returns a random neighbor (out-neighbor) of vertex-item *vg*.

(vertex-item? *vg*)

(method) Returns #t if *vg* is a vertex-item object.

9.4 Edge Graphics

(seg-list *eg*)

(slot) Holds a list of <edge-segment-item> objects containing segments of the edge and associated labels. For binary graphs, the segment list has length 1.

(edge *eg*)

(slot) Holds the <edge*> object contained within *eg*.

(width *eg*)

(slot) The width of the line segments in <edge-item>. The default is 1.

(color *eg*)

(slot) The color of the line segments in <edge-item>.

(text-color *eg*)

(slot) The color of the edge label(s).

(font *eg*)

(slot) The font of the edge label(s).

(tags *eg*)

(slot) The STk tags associated with the edge-item. This information is contained in a list, and some common values include 'edge, which is always present in vertices, and 'link:selected-item,' which is added if the user selects the edge.

(lower-edge *eg*)

(method) Lower the priority of the edge in the STk <canvas>'s list of displayed items, moving it below all other items.

(raise-edge *eg*)

(method) Raise the priority of the edge in the STk <canvas>'s list of displayed items, moving it above all other items.

(random-edge-weights *g*)

(method) Assign random values to the weight attribute of the <graph*> object. OLD, UNSATISFACTORY. USES STk's RANDOM, WHICH CALLS RAND(). VERY BAD.

(show-edge-weights *g*)

(method) Produce a list of the weight attributes of the edges of <graph*> *g*.

```
STk>  (define dg (digraph '(1 2 3 4) '((1 2 3) (2 3 4))))
#[undefined]

STk>  (define n (neighbors (vertex-ref dg 1)))
#[undefined]

STk>  (describe (vertex-ref dg 1))
#[<vertex*> 402cb118] is an instance of class <vertex*>
Slots are:
     val = 2
#f

STk>  (describe n)
#[<set<vertex*>> 401fe8c4] is an instance of class <set<vertex*>>
Slots are:
     val = {3 4}
#f

STk>  (define in (in-neighbors (vertex-ref dg 1)))
#[undefined]

STk>  (describe in)
#[<set<vertex*>> 402c9de0] is an instance of class <set<vertex*>>
Slots are:
     val = {1}
#f

STk>  (define out (out-neighbors (vertex-ref dg 1)))
#[undefined]

STk>  (describe out)
#[<set<vertex*>> 402c92d0] is an instance of class <set<vertex*>>
Slots are:
     val = {3 4}
#f
```

Figure 9.2: Vertex neighbors, directed graphs

```
STk>    (define g (graph '(1 2 3) '((1 2) (1 3) (3 1))))
#[undefined]

STk>    (map (lambda (x)
           (find-string-attribute 'color x))
         (set-vertex-¿list (vertices g)))
("black" "black" "black")
```

Figure 9.3: Finding attributes

```
STk>    (define g (graph '(1 2 3) '((1 2) (1 3) (3 1))))
#[undefined]

STk>    (set-string-attribute! 'color "red" (vertex-ref g 1))
#[undefined]

STk>    (map (lambda (x) (find-string-attribute 'color x))
(set-vertex-¿list (vertices g)))
("black" "red" "black")
```

Figure 9.4: Manipulating attributes

```
STk>    (define g (graph '(1 2 3 4) '((1 2) (1 3) (1 4))))
#[undefined]

STk>    (define (max-vert l)
       (cond ((null? l) 0)
             (#t (max (degree (car l)) (max-vert (cdr l))))))
#[undefined]

STk>    (define-generic delta)
#[undefined]

STk>    (define-method delta ((g ¡graph*¿))
       (let ((v (set-vertex-¿list (vertices g))))
               (max-vert v)))
delta

STk>    (delta g)
3
```

Figure 9.5: Defining graph functions: $\Delta(g)$

Part IV

Programmer's Manual

2 Extending the System

Assuming that LINK has been installed properly, the procedure for customizing the system to meet individual needs is relatively straightforward. This first release of manual will serve as an introduction to this process, but, unfortunately, is not yet a complete reference guide. This section of the manual will grow from release to release of LINK, and will give references to examples in the code when the current text is inadequate. The process of extending the STk interpreter is a derivative of the process described by Erick Gallesio in the STk documentation. If you plan to add several commands to the system, the document entitled *Extending the STk Interpreter* and found in the STk distribution is a valuable reference. However, the LINK system does include many helpful routines which help in the extension of the interpreter in a systematic way. In particular, new STk primitives for *Collection*, *List*, and *Array* methods are generated automatically by a process described below. To begin, let us consider a simple example. We assume is that the following environment variables have been set:

Environment Variable	Value
LINK_BASE	full pathname of link's root directory
STK_LOAD_PATH	$(LINK_BASE)/stklos
STK_LIBRARY	depends on the location of your STk distribution
LINK_LIBRARY	the location of *LINK*'s libraries once installed (see the file $LINK_BASE/INSTALL)

2.1 Compiling Standalone Programs using *LINK*'s Libraries

The file *Makefile* in the LINK_BASE directory is useful not only as the root makefile of the system, but also as a makefile for stand-alone user applications which use LINK's libraries. The programmer may link his or her C/C++ code with the LINK libraries to produce a C++ program, or to produce a new version of the whole system with new features. In order to use the makefile to compile your Link applications, you must add your source files to the USER_SOURCE list in the Makefile (see Figure 2.2, making sure to end each line except the last one with a backslack. There is a demonstration application in $LINK_BASE/standalone, which the Makefile is set up to build. To build and run this example application, Copy all of

```
###################################################################
#USER_SOURCE = \
#               demo.cpp \
#               linkUserInit.cpp # must edit this file and include it
#                               # in this list to augment the GUI
#
USER_SOURCE = \
                demoNoGUI.cpp  # demoNoGUI.cpp is a sample program
                               # using LINK objects, but not the GUI.
###################################################################
###################################################################
## Name your executable file below.
###################################################################
#EXECUTABLE = demo
EXECUTABLE = demoNoGUI
###################################################################
```

Figure 2.2: Makefile macros intended to be modified by the LINK library programmer.

the files in that directory and the LINK Makefile to an an empty directory. Then you can type

make demoNoGUI

to build the first demo. The result is a program called *demo* which computes a simple function using LINK objects. The function computed by the demo program can also be included into the LINK interface so that users may experiment with it interactively. To build a version that includes the GUI, reverse the comments in USER_SOURCE and EXECUTABLE so that demo.cpp and linkUserInit.cpp are compiled. A new version of the system which includes new commands is then built with the command:

make -D STK_GUI=yes demo

NOTE: The LINK Makefile links your program with all of the Link libraries (layouts, generators, algorithms, graph, and basic). If executable size is a problem for you, edit the LINK_LIBS macro in the makefile so that only the appropriate libraries are included. Note also that since *LINK* uses manual template instantiation with g++, new types of objects (e.g. Set<NewObject>)

need special treatment. *LINK* employs some mechanisms which automatically generate template instantiations. [1] The programs which accomplish this are *instantiate* and *genMethods*, located in the directory $LINK_BASE/autowrap. These programs are currently undocumented, but dozens of examples of their usage to generate template instantiations and STk wrappings for various Collection and Container objects such as sets and lists are found in the *GENERATION_COMMANDS.** files located in the directories *linkTemplates*, *STkTemplates*, and *autogenSTkMethods* under the *src/basic*, *src/graph*, and *src/userobj* directories. When the requirements for instantiation new templates exceed the capabilities of *instantiate* and *genMethods*, manually-generated template instantiations are put into the files *basicTemplate.cpp*, *graphTemplate.cpp*, and *userobjTemplate.cpp*. Note that $LINK_BASE/bin contains some useful utilities for quick editing and searching in the *LINK* file hierarchy.

2.2 Adding a Menu Option

Any algorithm you wrap into the user interface can easily be added to the menu of each graph window. Simply make your own copy the file *$LINK_BASE/stklos/graph-menu.stklos* and edit it. It will probably not be necessary to learn any Scheme, since you should be able to copy the menu entry of an existing algorithm and change the name. If it becomes necessary to make more interesting changes, see the STk manual and the GUI part of the Link manual. If you put your copy of `graph-menu.stklos` (and any other interface files you wish to customize) in the current directory, they will be loaded automatically when the application starts. If, however, you wish to keep your customized interface files in a separate directory, modify the STK_LOAD_PATH environment variable to include that directory before $LINK_BASE/stklos. For an example of this customization, consider the menu option for the *spring-layout* command was created by adding the following line to the "Layouts" sublist of the menu string contained in `graph-menu.stklos`:

```
("Spring" ,(lambda ()(spring-layout* (slot-ref gv 'graph) gv)))
```

The first element in the list is the text of the menu item, while the second element is the function which is to be called if the user selects this menu item. The comma is used since this line is used to force evaluation of the

[1] This is, they automatically generate files which perform manual instantiation from g++'s point of view. Manual instantiation reduces the burden placed on the compiler and linker during compilation.

```
//**************************************************************************
PRIMITIVE evenOddRanksWrapper(SCM g)
{
        Graph *gr;
        if (!(gr = getGraph(g)))   // see src/stkWrapper/stkUtils.cpp
                Err("even-odd-ranks: expected <graph*> ", g);
        Set<Set<Vertex*> > result = evenOddRanks(gr);
        Set<Set<Vertex*> >* new_copy = new Set<Set<Vertex*> >(result);
        return STk_make_CXXwrapper(
                Wrapper<Set<Set<Vertex*> > >::type,     // stkWrapper/Wrapper.h
                Wrapper<Set<Set<Vertex*> > >::name,
                (void *) new_copy,
                LINK_DYNAMIC);          // tells STk's garbage collector
                                        // to delete new_copy when the user
                                        // is finished with it
}
//
// This is what must go into userinit.cpp
//
void STk_evenOddRanks_Init()
{
        STk_add_new_cpp_primitive("even-odd-ranks",     // cmd name
                                  tc_subr_1,            // # parms
                                  (PRIMITIVE (*)(...))evenOddRanksWrapper);
}
//***********************************************************************
```

Figure 2.3: An example wrapper routine

expression (this line is found within a back-quoted list in the file. See the Scheme standard for an explanation of Scheme quoting).

2.3 An Example Wrapper

A *wrapper* routine is one which translates its input into a new format, calls a routine which requires this format, and then translates the return value back into the format of the calling environment. In *LINK*, wrappers extract C++ data from STk structures, call a *LINK* method, and store the return value in a new Scheme object. As stated above, Erick

Gallesio's document *Extending the STk Interpreter* will be a helpful guide to extending the *LINK* system, though there are dozens small wrapper programs in *LINK* which can serve as examples. In particular, the file *graph_Graph_Methods.cpp* in `src/graph/manualSTkMethods` contains wrappers for the many methods of the *Graph*class, while the files in `src/algorithm/manualSTkMethods` contain individual wrapper programs for the library algorithms. Let us consider the example wrapper in Figure 2.3. This is the wrapper associated with the example program `demoNoGUI.cpp`. The input is an STk "cons" element, which is simple a structure pointer. STk provides many helpful macros for extracting useful information from this structure (see stk.h). However, *LINK* provides several shortcut functions to abstract some of the detail away.

2.4 Adding a Graph Generator

In order to add a graph generating algorithm to the library, use the following procedure:

- Add the file(s) containing the code to the `$LINK\_BASE/src/generator` directory. A generator of random graphs is shown in Figure 2.4.
- Once the code has been written and included, edit the file `Make.generator` in the same directory and include the filename in the SOURCES.cpp macro so that the file will be compiled by the make process.
- Edit the file `Generators.h` in the directory `$LINK\_BASE/include/LINK/generator/` to include a prototype for the new generator routine.
- Create a wrapper routine for the new generator algorithm in the `$LINK\_BASE/src/generator/manualSTkWrappers` directory. This wrapper may be virtually identical to most other wrappers in that directory, which therefore serve as a good starting point. Figure 2.5 contains an example wrapper code.
- Edit the file `Make.generatorWrappers` in the same directory, and add the wrapper filename to the SOURCES macro.
- Edit the file `generatorSTkInit.cpp` in the directory `$LINK\_BASE/src/wrappers`. Include the prototype for the new wrapper at the top of the file and a call to it in the STk_generator_init() function.
- Edit the file `graph-menu.stklos` in the directory `$LINK\_BASE/stklos` and add an option under the "Generate" menu for the new generation routine. This will consist of adding a new line to the ....

```
UBinGraph*
GenRandom(int n, int p)

    UBinGraph*  graph = new UBinGraph();
    graph->addVertices(n);
    const SortedArray<Vertex*>& vertices = graph->vertices();
    int num_vertices = graph->order();
    if (p < 0.0 || p > 1.0)
        warning("GenRandom(): bad edge probability, using 0.0");
        p = 0.0;

    Set<Vertex*> s;
    for (int i=1; i < num_vertices; i++)
        for (int j=i; j < num_vertices; j++)
                if (LINK_randomDouble0_1() < p)
                        s.insert(vertices[i]);
                        s.insert(vertices[j]);
                        Edge* edge = graph->addEdge(s);
                        s.clear();


    return graph;
```

Figure 2.4: An example graph generating algorithm

2.5 Adding New Algorithms to the *LINK* Libraries

Typically, to customize the system, *LINK*'s libraries should be used as they are (as described in the previous section). Remaking the libraries themselves is certainly possible, but requires more knowledge of the system's organization. Below, we will describe the process of recompiling the libraries to include a new algorithm. Suppose that we have been developing a *LINK* algorithm, and are satisfied that it belongs in *LINK*'s libraries for other people to use. Suppose further that our algorithm has its own associated header file and two associated source files. The following sections will describe the integration process with a definite example: the Kernighan-Lin partitioning algorithm [7, 5]. Our algorithm's header file is called `Partitioning.h`, and its source

```
static PRIMITIVE GenRandomCmd(SCM n, SCM p)

        Graph *gr;
        if (!INTEGERP(n))
                Err("gen-random-graph: bad arg 1, <integer> expected", n);
        if (!IFLONUMP(p))
                Err("gen-random-graph: bad arg 2, <number> expected", n);
        int sz = INTEGER(n);
        double prob = FLONUM(n);
        Graph *g = GenRandom(sz, prob);
        Register<Graph>::createEntry(Wrapper<UBinGraph>::type,
                                     Wrapper<UBinGraph>::name, g);
        Graph **gp = new Graph*(g);
        return STk_make_CXXwrapper(Wrapper<UBinGraph*>::type,
                Wrapper<UBinGraph*>::name, (void *) gp, LINK_DYNAMIC);

void gen_Random_Wrapper()

        STk_add_new_cpp_primitive("gen-random-graph", tc_subr_2,
                                  (PRIMITIVE (*)(...))GenRandomGraphCmd);
```

Figure 2.5: An example wrapper for a graph generating algorithm

files are called `Partitioning.cpp` and `KLFM.cpp`. The partitioning files are separated since they will be of use to potentially many similar algorithms.

2.5.1 Integrating the Header Files

The file `Partitioning.h` belongs in the following directory: `$LINK_BASE/include/LINK/algorithm`. Note that if this were a graph generating algorithm, its files would belong in the `generator` subdirectory instead of `algorithm` (and similarly, graph layout algorithm files belong in the `layout` subdirectory). Once the header file has been moved to the appropriate place, the file `Make.inclalgorithm` should be updated to include `Partitioning.h` in the "FILES.h" macro therein to keep a record of its existance. *While in the algorithms directory, we must add a prototype for our new algorithm to the file* `Algorithms.h`.

2.5.2 Integrating the Source Files

The source files should be copied to the following directory:

`$LINK\_BASE/src/algorithm`.

Again, note that the procedure for including new generator and layout algorithms is similar. To update the compilation process, it is next necessary to edit the file `Make.algorithm` in the same directory by including the new source files in the "SOURCES.cpp" marcro. The system is now ready for recompilation, after which our new algorithm will be available in the `libalg.a` library.

2.5.3 Wrapping the New Algorithm into the Interface

Once the algorithm has been written and integrated into *LINK*'s C++ libraries, programmers will be able to use it to build new C++ programs; however, the real power of *LINK* is its computing environment, so new algorithms should be wrapped into the latter so that they can be called from the command-line and graphical interfaces. Example wrappers are shown above in Figures 2.3 and 2.5, and a similar wrapper should be written for our new algorithm. Since we are adding a fundamental graph algorithm (as opposed to a graph generating or layout algorithm), its wrapper belongs in the

`$LINK\_BASE/src/algorithm/manualSTkWrappers`

directory. Once the file is there, we must update the `Make.algorithmWrappers` file to include our new wrapper file in the "SOURCES.cpp" macro. The wrapper file must contain a function to create the new STk commands which will call our algorithm (for example, see the function called *gen_Random_Wrapper()* in Figure 2.5. We must now arrange for this function to be called. This is done by editing the file `$LINK\_BASE/src/wrappers/algorithmSTkInit.cpp` to make the appropriate call. This file simply contains prototypes for and calls to each wrapper generating function. Figure 2.6 shows the contents of this file at the time of this writing. The functions *STk_algorithm_init()* and *STk_algorithm_cleanup* are called at the beginning and ending, respectively of a *LINK* session.

3 Algorithms

3.1 Algorithm Specifications

The starred algorithms are finished, and the other algorithms are ranked in order of urgency. Hopefully these rankings will not be necessary after July 31st, 1996.

Priority	Algorithm/Property	input	output	attr's & algs used
2	Biconnected Components	Undir. G	Set<Set<Vertex*> >	DFS - back DFS - low DFS - discovery_time
5	Bipartite Matching	Undir. G	Set<Edge*>	weight Ford-Fulkerson - flow
*	Breadth-First Search	G,v	Sequence<Vertex*>	color distance ???
*	Depth-First Search	G	Sequence<Vertex*>	color starttime finishtime back low pred
3	Dijkstra's Single Source Shortest Paths	G, v	Set<double>	weight distance pred
4	Ford-Fulkerson Maximum Flow	Network s, t	double	flow capacity partition
7*	Floyd-Warshall All Pairs Shortest Paths	G	Matrix<double>	weight distance pred
*	Kruskal's MST	G	Set<Edge*>	weight mark parent pred
6	Prim's MST	G	Set<Edge*>	weight key pred
1	Strongly-Connected Components	Digraph	Set<Set<Vertex*> >	DFS - mark DFS - finishtime
*	TopSort	DAG	Sequence<Vertex*>	DFS - finishtime

4 Basic Objects

The foundation of LINK is a group of templated C++ objects. These objects are obtained by including their header files from the `LINK/basic` directory (see Section 1). There are many examples of programs using the LINK basic objects in the following sections.

2 *Containers*

Container is an abstract base class which is the parent of the following data structure classes: *Array*, *BinaryHeap*, *BinarySearchTree*, *DList*, *List*, and *RedBlackTree*. This hierarchy of containers is useful because certain common operations such as iteration, comparison, and display can be presented to the programmer with a common interface, even though the underlying mechanisms may differ. In fact, the *Iterator* objects described on page 89 can operate seamlessly on both *Containers* and *Collections* (see 3). There are two basic varieties of *Container*. The first variety is templated only by a key field, and currently contains *Array*, *List*, *DList*, *SortedArray*, and *SortedList*. The second variety, dictionaries, are templated by key and information fields. This set of classes currently includes *BinaryHeap*, *BinomialHeap*, *BinarySearchTree*, and *RedBlackTree*. *These dictionary structures have not been tested.*

2.1 Adding New *Containers*

It is straightforward to add new simple *Container* objects to LINK. The new classes must inherit from class *SimpleContainer* and must be templated with a single *Item* argument. *Item* will be assumed to have the following operators defined: `== <= != <<`. In the methods of the new data structure class, comparisons and output of data items should be carried out using the following methods of the class *ElementOps*, which is the base class for both *Container* and *Collection*:

- *int compareItems(Item, Item)*
- *int displayItem(ostream&, Item).*

These methods should be used instead of the comparison and output operators because certain very important element types such as *Vertex** and *Edge** must be treated as exceptions by the *Container* and *Collection* hierarchies. Mere pointer comparisons are unsatisfactory in these cases, so specialized methods for these are prototyped in ElementOps.h and given definitions in graphTemplate.cpp. *Containers* and *Collections* may be compared using the overloaded C++ operators: `==`, `!=`, `<=`, `<`, `>=`, `>$`. The comparison is based on lexicographic ordering of the *Container/Collection* elements. In general, *if two objects of class Collection or Container contain the same elements in the same order, then they are considered equal.* Inequalities are decided based upon the lexicographic ordering of the individual elements. An example involving *List* comparisons is given in Section 2.3. Some examples to follow when implementing new *SimpleContainer* objects are found in List.h, List.cpp, Array.h, and Array.cpp in the `/include/LINK/basic` directory. The following virtual functions define the interface of the *Container* class. Those that are pure virtual must be given definitions in the new class, while those with definitions may or may not be overridden. For example, *Container¡Item¿::min()* and *Container¡Item¿::max()* are implemented by iterating through all elements. If the data structure class stores data in sorted order, these should be overridden. **2. Operations**

virtual ˜*Container*<item>()

The virtual destructor in the *Container* class ensures that the proper subclass destructor will be called when the object referenced or pointed to is deleted. The new data structure must define its own destructor (or inherit one from a parent *Container*).

Item info(ContainerNode *cn) const;

Return the data value stored in node *cn*.

int size() const = 0;

Return the number of stored elements.

Bool emptyQ() const = 0;

Return TRUE if there are no stored elements.

Bool fullQ() const = 0;

Return TRUE if there are no space left for additional elements.

Bool memberQ(const Item&e) const = 0;

Return TRUE if e is stored.

Item min() const;

Return the minimum data value stored ($O(n)$). Note that this method is virtual and should be redefined in any descendant class which can implement it more efficiently.

Item max() const;

Return the maximum data value stored ($O(n)$). Like *min()*, this should be overridden in any descendant class in which a more efficient implementation is possible.

DataType type() const = 0;

Return the type of this structure (*DataType* is an enumerated type defined in general.h).

Item get() = 0;

Remove and return the first element in iteration order.

void remove(Item e) = 0;

Find and remove e from the structure. If this is a linked structure, delete the *ContainerNode* containing e. Do not delete e.

void clear() = 0;

Remove all data from the structure, deleting *ContainerNodes* but not actual elements.

int iterate(Iterator<Item>& it, Item& e) const = 0;

This method is used by *Iterator* objects to obtain the successor of e. Good examples of different implementations of this method are found in Array.h and List.h. This method is declared to be `public` in this class to allow access through *Container* pointers. However, it should be declared private in the descendant classes. The only access should be with *Iterator* objects.

ostream& display(ostream&os) = 0;

This will be called when *operator<<(ostream&,Container<Item>&)* is invoked.

Bool operator==(const Collection<Item>&) const = 0;
Bool operator<=(const Collection<Item>&) const = 0;
Bool operator< (const Collection<Item>&) const = 0;
Bool operator>=(const Collection<Item>&) const = 0;
Bool operator> (const Collection<Item>&) const = 0;
Bool operator!=(const Collection<Item>&) const = 0;
Bool operator==(const Container<Item>&) const = 0;
Bool operator<=(const Container<Item>&) const = 0;
Bool operator< (const Container<Item>&) const = 0;
Bool operator>=(const Container<Item>&) const = 0;
Bool operator> (const Container<Item>&) const = 0;
Bool operator!=(const Container<Item>&) const = 0;

Compare the *Container* with a *Collection* or *Container* argument. These methods are defined in Container.cpp and are common to all *Container* objects.

2.2 Iterators

Iteration through *Containers* and *Collections* is performed by *Iterator* objects. These class instances are constructed with the address of a *Container* (or *Collection*). Elements are retrieved subsequently using the () operator, which returns 1 while elements remain, then returns 0. For example, consider Figure 2.3. More than one iterator may operate on a given *Container* at once; however, removal of elements from within the iteration loop can corrupt the iteration scheme. Note that *Iterators* make the definition of copy constructor member functions almost trivial. Any new data structure class may define a constructor which takes an argument of type (*Container*<Item>&). The body of the constructor simply iterates through the elements of the passed *Container* and inserts them into the structure under construction.

2.2.1 *ContainerNode*

ContainerNode is an abstract base class which serves as parent to any node types used by container descendants. Two examples are the *ListNode* class defined in List.h and the *BinaryHeapNode* class defined in BinaryHeap.h.
1. Creation

There are no public constructors defined for ContainerNode. They can only be created by inserting items into a container. **2. Operations** void* info()

Return the address of the information stored in *ContainerNode*. This is useful in creating a consistent interface between *Containers* and algorithms. For example, consider the following code:

```
Container<int> *cp = createMysteryLinkedIntContainer();
ContainerNode<int> *cn = cp->insert(3);

const int i = cp->info(cn);
```

Regardless of the underlying representation, the information stored in a node is available. When using linked container structures such as lists or trees, it is sometimes useful to store *ContainerNode* pointers for future quick reference without having to search for items. However, If the underlying container is an array, there is no *ContainerNode* structure.

2.3 *List*

The *List* class is the most basic linked data structure class in the system. It is templated by element type and it stores its elements in *ListNodes*, which can be referenced through *ContainerNode* pointers. Figure 2.4 contains a coding sample showing the construction of empty *Lists* and some operations with elements and *ContainerNodes*. Note that receiveing *ContainerNode* pointers as return values is useful since they providing entry points into the list and enable operations such as *predecessor()* and *successor()* to run in constant time. When we consider the doubly-linked list (*DList*) class, this feature will also permit constant time linking and unlinking of list nodes. It is possible to construct new *List* objects from existing ones using both `X(const X&)` constructors and the assignment operator, as shown in Figure 2.5 below. These operations iterate through the source *List*, inserting elements in the same order into the destination *List*. Note that if this presents an efficiency problem, *Collections* should be used. These implement reference counting (see Section 3). *Lists* are the default implementation for *Sequences*, which will be discussed in Section 3.4. The default implementation for *Sets* and *MSets* (multisets) is *SortedList*, a child class of *List*. *When the LINK interface switches to STk,* Lists *will have special significance since they will be* *implemented using the same underlying memory structure as STk objects.*

1. Creation

List<Item> l;

Construct a singly-linked list l.

List<Item> l2(c);

Construct a singly-linked list $l2$ and initialize it with the elements of *Container* c such that the elements are in the same order.

2. Operations

List<Item>& operator=(List<Item>& list)

If the current list is different from the passed list, then remove any elements in the current list and iterate through the elements of the passed list, inserting them into the current list. $O(n)$

int size()

Return the number of items stored in the current list. In order to allow a smooth interface with Scheme, no element count is maintained. $O(n)$

Bool memberQ(Item& item)

Indicate whether item is in list or not. $O(n)$

Bool sortedQ() const

Return FALSE. $O(1)$.

ContainerNode* search(Item& item)

Search for item and return a pointer to the node which contains it. $O(n)$

Bool emptyQ()

Return true if list is empty. $O(1)$

Item get()

Return first element in the of list and remove it. $O(1)$ *Caution:* If the list is empty, this routine returns an uninitialized *Item* object.

Item first()

Return the first item. $O(1)$ *Caution:* If the list is empty, this routine returns an uninitialized *Item* object.

Item last()

Return the last item. $O(n)$ *Caution* If the list is empty, this routine returns an uninitialized *Item* object.

ContainerNode* prepend(Item item)

Insert the item at the beginning of list. $O(1)$

ContainerNode* insert(Item item)

Insert the item at the beginning of the list. $O(1)$ The redundancy in naming is to maintain naming consistency with *Collection* member functions.

ContainerNode* append(Item item)

Append the item to the back of list. $O(n)$

ContainerNode* insertBefore(Item item, ContainerNode *n)

Insert the item before node n. $O(1)$

ContainerNode* insertAfter(Item item, ContainerNode *n)

Insert the item after node n. $O(1)$

void remove(Item item)

Remove the item from list and delete the *ListNode* which contained it. $O(n)$

void clear()

Remove all items from the list and delete their associated *ListNodes*. $O(n)$ *Note:* This will become $O(1)$ with the Scheme interface, since periodic garbage collection cleans up memory

ContainerNode* predecessor(const ContainerNode *n)

Return the node preceding n in list order. $O(1)$

ContainerNode* successor(const ContainerNode *n) const

Return the node succeeding n in list order. $O(1)$

ostream& display(ostream& os)

Output the list to the output stream *os*. Uses *Container::displayItem(...)* to display each element. $O(n)$

int   rank(Item& item)

Return the position of the item in the list. $O(n)$

2.4 *SortedList*

The *SortedList* class is a straightforward adaptation of *List* in which elements are inserted in order and searches are peformed more efficiently. *SortedList* is currently the default set implementation (see Page 143). As stated above in Section 2, the copy construction and assignment operations work the same way with any combination of *SimpleContainer* objects. The interaction between *List* and *SortedList* in Figure 2.6 provides our first example of this. Note that the invariant of lexicographically-ordered storage of elements in *SortedList* is never violated, despite the unordered source *List*. Like assignment and initialization, comparison of *Containers* is consistent despite *Container* type. Unlike assignment and initialization, though, comparison is consistent accross both *SimpleContainer* and *Dictionary* objects. For an example, consider the code given in Figure 2.7.

```
#include<LINK/wrappers/algorithmSTkInit.h>

void gen_Animation_Wrapper();
void gen_BiconnectedComponents_Wrapper();
void gen_BreadthFirstSearch_Wrapper();
void gen_DepthFirstSearch_Wrapper();
void gen_GoldbergTarjan_Wrapper();
void gen_HamiltonianCycles_Wrapper();
void gen_KLFM_Wrapper();
void gen_Kruskal_Wrapper();
void gen_MaximalClique_Wrapper();
void gen_Prim_Wrapper();
void gen_StronglyConnectedComponents_Wrapper();
void gen_TopologicalSort_Wrapper();
void gen_Triangles_Wrapper();

void STk_algorithm_init(void)

        gen_Animation_Wrapper();
        gen_BiconnectedComponents_Wrapper();
        gen_BreadthFirstSearch_Wrapper();
        gen_DepthFirstSearch_Wrapper();
        gen_GoldbergTarjan_Wrapper();
        gen_HamiltonianCycles_Wrapper();
        gen_Kruskal_Wrapper();
        gen_KLFM_Wrapper();
        gen_MaximalClique_Wrapper();
        gen_Prim_Wrapper();
        gen_StronglyConnectedComponents_Wrapper();
        gen_TopologicalSort_Wrapper();
        gen_Triangles_Wrapper();


void STk_algorithm_cleanup(void)
```

Figure 2.6: algorithmSTkInit.cpp

```
#include<LINK/basic/Sequence.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/graph/Attribute.h>
#include<LINK/algorithm/Algorithms.h>

main()
{
        DBinGraph g;
        Sequence<Vertex*>  s, postorder;
        Vertex *parent, *child1, *child2;

        child1 = g.addVertex("child1");
        parent = g.addVertex("parent");
        child2 = g.addVertex("child2");
        s.insert(parent); s.append(child1);
        g.addEdge(s);
        s.clear();
        s.insert(parent); s.append(child2);
        g.addEdge(s);

        postorder = DepthFirstSearch(&g);
        cout << postorder << endl;
}
```

```
<child1 child2 parent>
```

Figure 3.7: Depth-First Search

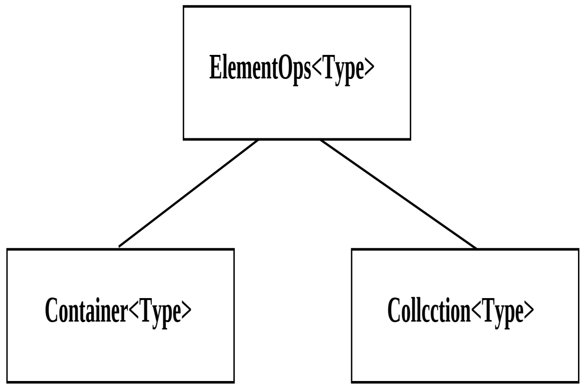


Figure 2.2: The *ElementOps* Class

```
#include<iostream.h>
#include<LINK/basic/List.h>

main()
{
        List<int> l;

        l.insert(1);
        l.insert(2);
        l.insert(3);
        l.insert(4);

        Iterator<int> get_next(&l);
        int i;
        while (get_next(i))
                cout << "next element: " << i << endl;
}
```

```
next element: 4
next element: 3
next element: 2
next element: 1
```

Figure 2.3: Use of *Iterator*

```
#include<iostream.h>
#include<LINK/basic/List.h>

main()
{
	List<int> l;

	ContainerNode *cn = l.insert(3);
	l.insertAfter(4, cn);
	cout << l << endl;

	cn = l.search(3);
	ContainerNode *cn2 = l.successor(cn);
	cout << "value: " << l.info(cn2) << endl;
}
```

```
(3 4)
value: 4
```

Figure 2.4: *List* objects

```
#include<iostream.h>
#include<LINK/basic/List.h>

main()
{
        List<int> list1;

        for (int i=0; i<3; i++)
                list1.append(i);

        List<int> list2(list1);
        list2.append(9);

        List<int> list3;
        list3 = list2;
        list3.insert(12);

        cout << list1 << endl;
        cout << list2 << endl;
        cout << list3 << endl;
}
```

```
(0 1 2)
(0 1 2 9)
(12 0 1 2 9)
```

Figure 2.5: *List* constructors

```
#include<iostream.h>
#include<LINK/basic/List.h>

main()
{
        List<int> list;

        for (int i=0; i<3; i++)
                list.insert(2 * i);

        SortedList<int> slist(list);
        slist.insert(3);

        List<int> list2(slist);

        cout << list << endl;
        cout << slist << endl;
        cout << list2 << endl;
}
```

```
(4 2 0)
(0 2 3 4)
(0 2 3 4)
```

Figure 2.6: *List* and *SortedList*

```
#include<iostream.h>
#include<LINK/basic/List.h>
main()
{
        List<int> list1;
        SortedList<int> list2;
        list1.append(2); list2.append(2);
        list1.append(1); list2.append(1);

        List<int> list3(list1);
        list3.append(3);

        List<List<int> > llist;
        llist.append(list1); llist.append(list2); llist.append(list3);
        Iterator<List<int> > it1(&llist);
        Iterator<List<int> > it2(&llist);
        List<int> temp1, temp2;
        while (it1(temp1)) {
            while (it2(temp2)) {
                if (temp1 == temp2)
                        cout <<"("<< temp1 << " == " << temp2 << ")\t" ;
                if (temp1 != temp2)
                        cout <<"(" << temp1 << " != " << temp2 <<  ")\t" ;
                if (temp1 <= temp2)
                        cout <<"(" << temp1 << " <= " << temp2 <<  ")\t" ;
                if (temp1 < temp2)
                        cout <<"(" << temp1 << " < " << temp2 <<  ")\t" ;
                if (temp1 >= temp2)
                        cout <<"(" << temp1 << " >= " << temp2 <<  ")\t" ;
                if (temp1 > temp2)
                        cout <<"(" << temp1 << " > " << temp2 <<  ")\t" ;
                cout << endl;
            }
            it2.reset();
        }
}
```

```
((2 1) == (2 1))        ((2 1) <= (2 1))        ((2 1) >= (2 1))
((2 1) != (1 2))        ((2 1) >= (1 2))        ((2 1) > (1 2))
((2 1) != (2 1 3))      ((2 1) <= (2 1 3))      ((2 1) < (2 1 3))
((1 2) != (2 1))        ((1 2) <= (2 1))        ((1 2) < (2 1))
((1 2) == (1 2))        ((1 2) <= (1 2))        ((1 2) >= (1 2))
((1 2) != (2 1 3))      ((1 2) <= (2 1 3))      ((1 2) < (2 1 3))
((2 1 3) != (2 1))      ((2 1 3) >= (2 1))      ((2 1 3) > (2 1))
((2 1 3) != (1 2))      ((2 1 3) >= (1 2))      ((2 1 3) > (1 2))
((2 1 3) == (2 1 3))    ((2 1 3) <= (2 1 3))    ((2 1 3) >= (2 1 3))
```

Figure 2.7: Comparisons of *Containers*

The preceding example is interesting because it illustrates comparisons between elements which happen to be *List* and *SortedList* objects. Remember that two objects of class *Container* or *Collection* are considered to be equivalent if they contain equivalent objects in equivalent order. Nested iteration is used above to consider all possible comparison between three given lists. In order for the iteration scheme to work, *Container* pointers must have access through the virtual function mechanism to the *iterate()* method of each class in the *Container* hierarchy. This means that *SortedList* must declare *List* as a `public` base class even though private inheritance would be more natural. Therefore, *SortedList* must redefine certain inherited member functions to maintain the invariant that elements are stored in lexicographic order. The *List* methods listed in Section 2.3 are inherited by *SortedList*; only those redefined are listed below. **1. Creation**

SortedList<Item> l;

Construct a singly-linked sorted list l.

SortedList<Item> l2(l);

Construct a singly-linked list $l2$ and initialize it with the elements of l such that the elements of the new list are in lexicographic order. If l is an instance of a container other than a *SortedList*, then a sorted insertion routine must be used to load the elements, leading to $O(n^2)$ time. If l is a *SortedList*, then elements are appended to the end of $l2$, yielding $O(n)$ time.

2. Operations

List<Item>& operator=(Container<Item>& list)

If the current list is different from the passed list, then remove any elements in the current list and iterate through the elements of the passed list, inserting them in the lexicographic order. $O(n^2)$

List<Item>&   operator=(SortedList<Item>& list)

If the current list is different from the passed list, then remove any elements in the current list and iterate through the elements of the passed list, appending them in the same order onto the end of the current list. $O(n)$

Bool   memberQ(Item& item)

Indicate whether item is in list or not. Do not proceed if *item* is less than the current item $O(n)$

Bool   sortedQ() const

Return TRUE. $O(1)$.

ContainerNode*   search(Item& item)

Search for item and return a pointer to the node which contains it. Do not proceed if *item* is less than the current item. $O(n)$

ContainerNode*   prepend(Item item)

Insert the item at the beginning of list if no order violation would occur. Otherwise warn of an attempted order violation and return NULL. $O(1)$

ContainerNode*   insert(Item item)

Insert the item in lexicographically sorted order. $O(n)$

ContainerNode*   append(Item item)

Append the item to the back of list if no order violation would occur. $O(n)$

ContainerNode* insertBefore(Item item, ContainerNode *n)

Insert the item before node n if no order violation would occur. Otherwise warn the user and return. $O(1)$

ContainerNode* insertAfter(Item item, ContainerNode *n)

Insert the item after node n if no order violation would occur. Otherwise warn the user and return. $O(1)$

2.5 *DList*

The *DList*, or doubly-linked list, class is useful for implementing algorithms in which it is essential that constant-time splicing and unsplicing of list elements be supported. The node type used with this data structure differs from that of *List* since it contains a predecessor pointer. However, the *Container* hierarchy ensures that objects of class *List* and *DList* can interact easily. Copy construction, inter-assignment, and comparison of *Container* objects are still automatic. Code using *DList* objects is similar to that using *List* objects; most methods are the same (the exceptions are enumerated below). As with the singly-linked *List* object, *ContainerNode* pointers provide are used by the programmer to keep track of individual nodes, as shown in Figure 2.8. Some important applications such as bucket sorting require that nodes be spliced in and out of various doubly-linked lists very frequently. Such applications cannot be implemented efficiently using singly-linked lists due to the search overhead required to find predecessors of moving nodes. The *DList* class has additional methods which allow insertion and removal of *ContainerNodes* from the *List* (as opposed to elements), illustrated in Figure 2.9. Once again, comparison and display of *SimpleContainer* objects is consistent. The example in Figure 2.10 provides another illustration. *DList* objects have the same methods as *List* objects with the following exceptions:

1. Creation

DList<Item> l;

Construct a doubly-linked list l.

```
#include<iostream.h>
#include<LINK/basic/DList.h>

main()
{
        DList<int> l;

        ContainerNode *cn = l.insert(3);
        l.insertBefore(4, cn);
        cout << l << endl;

        cout << "value: " << l.info(l.predecessor(cn)) << endl;
}
```

```
(4 3)
value: 4
```

Figure 2.8: *DList*

List<Item> l2(l);

Construct a doubly-linked list $l2$ and initialize it with the elements of *Container* l such that the elements are in the same order.

2. Operations

List<Item>& operator=(List<Item>& list)

If the current list is different from the passed list, then remove any elements in the current list and iterate through the elements of the passed list, inserting them into the current list. $O(n)$

```
#include<iostream.h>

#include<LINK/basic/DList.h>
#include<LINK/basic/List.h>

main()
{
        DList<int> dl;

        dl.insert(1);
        ContainerNode *cn = dl.insert(2);

        DList<int> dl2(dl);

        dl.unlink(cn);
        dl2.insert(cn);


        cout << dl << endl;
        cout << dl2 << endl;
}
```

```
(1)
(2 1 2)
```

Figure 2.9: Special *DList* operations

```
#include<iostream.h>
#include<LINK/basic/List.h>

#include<LINK/basic/DList.h>

main()
{
        List<int> l;

        l.insert(5);
        l.insert(4);
        l.insert(3);

        DList<int> dl(l);

        if (dl == l)
           cout << "they have the same elements in the same order" << endl;
        dl.append(7);

        cout << dl << endl;
        cout << l << endl;
}
```

```
they have the same elements in the same order
(3 4 5 7)
(3 4 5)
```

Figure 2.10: Comparisons of *List* and *DList*

ContainerNode* insert(ContainerNode *cn)

Insert the item stored in node *cn* at the beginning of the list. $O(1)$. This enables users who have saved pointers to certain elements to move them between *DLists* efficiently. $O(1)$

ContainerNode* unlink(ContainerNode *cn)

Remove a node from the list. $O(1)$

2.6 *Deque*

The *Queue* and *Stack* classes are derived from the abstract base class *Deque*, which inherits privately from *List* to screen out all operations that allow access to elements other than the first element. *Deque* pointers can be used to point to either *Stack* or *Queue* objects. This flexibility is helpful in algorithms such as maximum flow, in which the algorithm user might wish to select at run time which storage protocal to use. Several *List* operations are allowed through the filter for use by *Stacks* and *Queues*. Those accessible through *Deque* pointers are listed below. Note that the `X(const X&)` constructors for *Stacks* and *Queues* are more restricted than those of other *Container* classes. Copies are constructed only from like objects, not arbitrary *Containers*.

Item front()

Returns the head of the *Deque* without removing it. $O(1)$

void insert(Item& e) = 0;

Inserts e at the front of the list if a *Stack* is referenced ($O(1)$) and at the end of the list if a *Queue* is ($O(n)$).

Item get()

Removes the element at the front of the list and returns it. $O(1)$

Bool emptyQ()

Is the *Deque* empty? $O(1)$

Bool sortedQ() const

Return FALSE. $O(1)$.

void clear()

Remove all the elements from the *Deque*. $O(n)$

void display()

Produce a human-readable display of the data structure. $O(n)$

2.6.1 *Queue*

Instances of the *Queue* class are singly-linked lists restricted to FIFO ("first in, first out") access. The methods differing from those of *Deque* (which are inherited) are listed below. **1. Creation**

Queue<Item> q;

Create an empty *Queue*. $O(1)$

Queue<Item> q2(q);

Create an empty queue and initialize it with the elements of q such that the elements of each *Queue* are in the same order. $O(n)$

2. Operations

Item front()

Returns the head of the *Queue*, without removing it.

Item dequeue()

Returns the head of the *Queue* and removes it.

void enqueue(Item& passed_item)

Appends *passed_item* to the end of *Queue*.

void insert(Item& item)

Calls *enqueue()*. This method is overrides the virtual method of the same name in *Deque*.

2.6.2 Stack

Like *Queue*, the *Stack* class is derived from *Deque* and redefines certain methods. The elements of the stack are stored using a LIFO ("last in, first out") protocol. **1. Creation**

Stack<Item> s;

Create an empty *Stack*. $O(1)$

Stack<Item> s2(s);

Create an empty *Stack* and initialize it with the elements of s such that the elements will be popped in the same order. $O(n)$

2. Operations

Item top()

Returns the item on top of the *Stack*, without removing it. $O(1)$

Item pop()

Returns the item on top of *Stack*and removes it. $O(1)$

void push(Item& passed_item)

Pushes *passed_item* onto the top of the *Stack*. $O(1)$

void insert(Item& passed_item)

Pushes *passed_item* onto the top of the *Stack*. This method overrides the virtual function of the same name in *Deque*.

Bool emptyQ()

Is the *Stack*empty? $O(1)$

void clear()

Remove all the elements from the *Stack*. $O(n)$

2.7 Arrays

Array-based data structures are fast for some applications, but they are relatively inflexible since the size of the array must be specified when it is allocated. The *Array* class solves this problem by allocating a fixed size initial array, and doubling it whenever it gets full. To cost of recopying the current contents is amortized over the size of the array. *Array* objects offer the ability to circumvent the standard *Container* iteration scheme, though the latter still works. Since the elements are stored in a contiguous block, it is much more efficient to iterate through them using pointer arithmetic. There is no need to package the data elements in structures from the *ContainerNode* hierarchy. The example in Figure 2.11 illustrates both forms of iteration on the elements of an *Array* object: **1. Creation**

Array<Item> a;

Creates an empty dynamic array. A default size of `sizeof(Item)` * DEFAULT_SIZE (see general.h) is used. $O(1)$

Array<Item> a((int) sz);

Creates an dynamic array with a block of space large enough to store *sz* `Item`s. $O(1)$

Array<Item> a((Array<Item>&) a);

Creates an dynamic array with a block of space as large as that of a and initializes it with the elements of a. $O(n)$

Array<Item> a((Container<Item>&) c);

Creates an dynamic array with storage of default size and initializes it with the elements of c. Note that the storage may be reallocated during this process. $O(n)$

2. Operations

```
#include<iostream.h>
#include<LINK/basic/Array.h>

main()
{
	Array<int> l;

	l.insert(3);
	l[2] = 4;
	cout << l << endl;
	int i, count = l.size();
	for (i=0; i<count; i++)
		cout << l[i] << " ";
	cout << endl;

	Iterator<int> it(&l);
	int element;
	while (it(element))
		cout << element << " ";
	cout << endl;
}
```

```
[3 0 4]
3 0 4
3 0 4
```

Figure 2.11: Iteration and *Array* objects

Array<Item>& operator=(Array<Item>& a)

Allocate space large enough to hold the elements of a, then copy its contents.

ContainerNode * insert(Item item)

Insert *item* into the 0'th cell of *Array*, shifting one cell forward any contiguous block of elements starting with that cell. If this operation would extend the active range of cells beyond the current allocation, reallocate the *Array* and recopy the elements. Note: The return type is *ContainerNode** so that this method may be called from *Container* pointers or references. However, since there is no packaging node structure in *Array*, a NULL pointer is returned. $O(n)$

ContainerNode * append(Item item)

The same as insert(Item item), except that the new item is added to the empty cell adjacent to the cell in the array's active range of largest index.

void remove(Item item)

Remove the item from *Array* if it is there. Shift the elements of larger index to fill in the vacancy left by the removal. $O(n)$

Item start()

Returns the address of the 0'th element of the *Array*. $O(1)$

Item get()

Return the last element in the active range of the array and remove it. $O(1)$ *Caution:* If the *Array*stores no elements, this routine returns an uninitialized *Item* object.

void clear()

Remove all items from the active range of the *Array*. $O(n)$

Bool memberQ(Item& item)

Is item in the array or not? $O(n)$

Bool sortedQ() const

Return FALSE. $O(1)$.

int order()

Return the number of cells allocated in the array. $O(1)$

int size()

Return the active range of elements, defined to be $(m - 0$, where m is the largest index of a referenced cell. $O(1)$

Item& operator[] (int ix)

Return item of index ix. Bounds checking is done, and the element count is updated if the cell referenced has a larger index than that of any previous reference. $O(1)$

int search(Item)

Search for an item in the active range of the array and return its index if it is found. Return -1 otherwise $O(n)$.

Item min()

The *Container* method is redefined since faster iteration is possible in this class. $O(n)$

Item   max()

The *Container* method is redefined since faster iteration is possible in this class. $O(n)$

ostream&   display(ostream& os)

Output the list to the output stream *os*. Uses *Container::displayItem(...)* to display each element. $O(n)$

2.8 *SortedArray*

When elements must be stored in order for efficient searching and convenient output, and the number of elements is likely to remain relatively stable, *SortedArray* storage is a good option. *SortedArray* is a class derived from *Array* that restricts insertion to ensure that the elements be stored in order. Binary search then reduces search time to $O(\log n)$. *SortedArray* is the storage medium used for the vertex set of LINK's graphs. The methods of *SortedArray* are listed below. Any *Array* method not listed or redefined is inherited. **1. Creation**

SortedArray<Item>   a;

Creates an empty dynamic sorted array. A default size of `sizeof(Item)` * DEFAULT_SIZE (see general.h) is used. $O(1)$

SortedArray<Item>   a((int) sz);

Creates an dynamic sorted array with a block of space large enough to store *sz* `Items`. $O(1)$

SortedArray<Item>   a((SortedArray<Item>&) a);

Creates an dynamic sorted array with a block of space as large as that of *a* and initializes it with the elements of *a* such that order is preserved. $O(n)$

```
#include<iostream.h>
#include<LINK/basic/List.h>
#include<LINK/basic/Array.h>

main()
{
        Array<int> array;
        List<int> list;

        for (int i=0; i<3; i++) {
                array.insert(2 * i);
                list.insert(2 * i);
        }

        SortedList<int> slist(array);
        SortedArray<int> sarray(list);
        slist.insert(3);
        sarray.insert(-4);

        cout << array << endl;
        cout << list  << endl;
        cout << slist << endl;
        cout << sarray<< endl;
}
```

```
[4 2 0]
(4 2 0)
(0 2 3 4)
[-4 0 2 4]
```

Figure 2.12: *Arrays* and *SortedArrays*

SortedArray<Item>   a((Container<Item>&) c);

Creates an dynamic array with storage of default size and initializes it with the elements of c. Note that the elements of c may not be stored in order, leading to a worst case running time of $O(n^2)$.

2. Operations

SortedArray<Item>&   operator=(SortedArray<Item>& a)

Allocate space large enough to hold the elements of a, then copy its contents such that order is preserved.

ContainerNode *   insert(Item item)

Insert *item* into the *SortedArray* such that order is preserved. If this operation would extend the active range of cells beyond the current allocation, reallocate the *Array* and recopy the elements. Note: The return type is *ContainerNode** so that this method may be called from *Container* pointers or references. However, since there is no packaging node structure in *Array*, a NULL pointer is returned. $O(n)$

ContainerNode *   append(Item item)

The same as insert(Item item), except that the new item is added to the empty cell adjacent to the cell in the array's active range of largest index *if the operation does not violate the invariant that the elements be stored in order. Otherwise, the new item is inserted in order* $O(1)$

Bool   sortedQ() const

Return TRUE. $O(1)$.

int   search(Item)

Search for an item in the active range of the array using binary search and return its index if it is found. Return -1 otherwise $O(\log n)$.

Item   min()

The *Container* method is redefined to take advantage of the ordered elements. $O(1)$

Item   max()

The *Container* method is redefined to take advantage of the ordered elements. $O(1)$

```
#include<iostream.h>
#include<LINK/basic/BinaryHeap.h>

main()
{
        BinaryHeap<int, char> heap; // default size = BINHEAPSIZE (general.h)
        ContainerNode *cn;

        heap.insert(1, 'a');
        heap.insert(2, 'b');
        cn = heap.insert(3, 'c');

        cout << heap << endl;

        heap.decreaseKey(cn, -4);
        BinaryHeapNode<int,char> *next;

        for (int i=0; i<3; i++) {
                next = heap.extractMin();
                cout << next->item << " ";
        }
        cout << endl;
}
```

```
[a b c]
c a b
```

Figure 2.13: *BinaryHeap*

2.9 Priority Queue Dictionary Structures

2.9.1 *BinaryHeap*

Priority queues are data structures which support fast insertion and extract-minimum operations. They are templated on both key and information fields, with the key field defining where the information sits in the heap.

1. Creation

```
#include<iostream.h>
#include<LINK/basic/BinaryHeap.h>
#include<LINK/basic/List.h>

main()
{
        BinaryHeap<int, char> heap(10);
        ContainerNode *cn;

        heap.insert(1, 'a');
        heap.insert(2, 'b');

        List<char> list(heap);  // uses Iterator to copy
        list.insert('d');

        cout << heap << endl;
        cout << list << endl;

        //BinaryHeap<int, char> heap2(list);
        //error: dictionaries cannot be constructed using
        //       other container types.
}
```

```
[a b]
(d a b)
```

Figure 2.14: Another *BinaryHeap* example

BinaryHeap<Key,Item> bh;

Create a binary heap of size BINHEAPSIZE (defined in general.h), with an array-based implementation. This is a fast priority queue if the number of elements is predetermined and unions are infrequent.

BinaryHeap<Key,Item> binary_heap_name((int) n);

Create a binary heap of size n.

2. Operations

BinaryHeapNode<Key,Item>* insert(Key k, Item i)

Insert item i with key k into the heap, returning a pointer to the node containing it. $O(\log n)$

void remove(Item i)

This does nothing. It is defined with an empty body to provide a definition for the *Container::remove(Item)* method.

Item extractMin()

Removes the node with minimum key from the heap and returns the information stored within. $O(\log n)$

Item get()

Calls *extractMin()*. $O(\log n)$

BinaryHeapNode<Key,Item>* minimum()

Return a pointer to the node containing the minimum item in the heap, but do not remove it. $O(1)$

Item min()

Call *minimum()*, then extract the Item from the returned node. $O(1)$

int size()

Return the number of items stored in the heap. An element count is always stored. $O(1)$

Bool emptyQ()

Is the heap empty? $O(1)$

Bool fullQ()

Is the heap full? $O(1)$

Bool memberQ(const Item&)

Is the item in the heap? $O(n)$

Bool sortedQ() const

Return FALSE. $O(1)$.

void clear()

Remove all the elements from the priority queue. $O(n)$

void decreaseKey(BinaryHeapNode<Key,Item>* node, Item k)

Decreases the value of the key in *node* to the new key value k. $O(\log n)$

void   deleteKey(ContainerNode* node)

Remove the entry in *noda* from the heap. $O(\log n)$

ostream&   display(ostream&)

Produce a human-readable display of the data structure. $O(n)$

friend ostream&   operator<<(ostream& s, Heap<Item>& h)

Output h to output stream s. This is the inverse operation of >>.

ObjectType   type()

What type/implementation of priority queue is this?

2.9.2 ***BinomialHeap***

1. Creation

BinomialHeap<Key,Item>   bnh;

Create a binomial heap, which may contain an arbitrary number of items, and supports fast unions.

2. Operations

BinomialHeapNode<Key,Item>*   insert(Key k, Item i)

Insert item i with key k into the heap, returning a pointer to the HeapNode containing it. $O(\log n)$

void search(const Key&)

Searches the heap for the passed key. $O(n)$

void searchItem(const Item&)

Searches the heap for the passed item. $O(n)$

void remove(BinomialHeapNode<Key,Item>* n)

Removes n from the heap. $O(\log n)$

void remove(Item i)

Removes n from the heap by first searching for i, then removing the node containing it. $O(n)$

Item extractMin()

Removes the node with minimum key from the heap and returns the information stored within. $O(\log n)$

Item get()

Calls *extractMin()*. $O(\log n)$

BinomialHeapNode<Key,Item>* minimum()

Return a pointer to the node containing the minimum item in the heap, but do not remove it. $O(1)$

Item min()

Call *minimum()*, then extract the Item from the returned node. $O(1)$

int size()

Return the number of items stored in the heap. An element count is always stored. $O(1)$

Bool emptyQ()

Is the heap empty? $O(1)$

Bool fullQ()

Is the heap full? $O(1)$

Bool memberQ(const Item&)

Is the item in the heap? $O(n)$

Bool sortedQ()

Return FALSE. $O(1)$

void clear()

Remove all the elements from the priority queue. $O(n)$

void changeKey(HeapNode* node, Key k)

Change the value of the key in *node* to the new key value k. $O(\log n)$

void decreaseKey(BinomialHeapNode<Key,Item>* node, Item k)

Decreases the value of the key in *node* to the new key value k. $O(\log n)$

void   merge(BinomialHeap<Key,Item>& H)

Merges the contents of H into the current heap. $O(\log n)$

void   scramble()

Randomly permute the contents of the data structure without losing the priority queue property. Useful for testing. $O(n)$

void   deleteKey(ContainerNode* node)

Remove the entry in *noda* from the heap. $O(\log n)$

ostream&   display(ostream&)

Produce a human-readable display of the data structure. $O(n)$

friend ostream&   operator<<(ostream& s, Heap<Item>& h)

Output h to output stream s. This is the inverse operation of >>.

ObjectType   type()

What type/implementation of priority queue is this?

2.10 General Dictionary Structures

The general dictionary structures allow random access like *Containers* but allow distinct lookup keys like the priority queue classes. Theses classes now include *BinarySearchTree* and *RedBlackTree*, though neither has been well tested or documented. The examples in Figures 2.15 and 2.16 illustrate the use of general dictionary classes. However, these classes should be used only with caution and the expectation that bugs may exist. As with any LINK problem, please report these problems to Jonathan Berry at `berryj@dimacs.rutgers.edu`.

```
#include<iostream.h>
#include<LINK/basic/BinarySearchTree.h>

main()
{
        BinarySearchTree<int, char> bntree;
        ContainerNode *cn;

        bntree.insert(1, 'b');
        bntree.insert(-1, 'a');
        bntree.insert(2, 'c');

        cout << bntree << endl;

        bntree.preorderWalk(cout);
        bntree.inorderWalk(cout);
        bntree.postorderWalk(cout);
}
```

```
<(-1,a)(1,b)(2,c)>

<(1,b)(-1,a)(2,c)>
<(-1,a)(1,b)(2,c)>
<(-1,a)(2,c)(1,b)>
```

Figure 2.15: *BinarySearchTree*

```
#include<iostream.h>
#include<LINK/basic/RedBlackTree.h>

main()
{
        RedBlackTree<int, char> rbtree;
        ContainerNode *cn;

        rbtree.insert(1, 'b');
        rbtree.insert(-1, 'a');
        rbtree.insert(2, 'c');

        cout << rbtree << endl;

        rbtree.preorderWalk(cout);
        rbtree.inorderWalk(cout);
        rbtree.postorderWalk(cout);
}
```

```
<(-1,a)(1,b)(2,c)>

<(1,b)(-1,a)(2,c)>
<(-1,a)(1,b)(2,c)>
<(-1,a)(2,c)(1,b)>
```

Figure 2.16: *RedBlackTree*

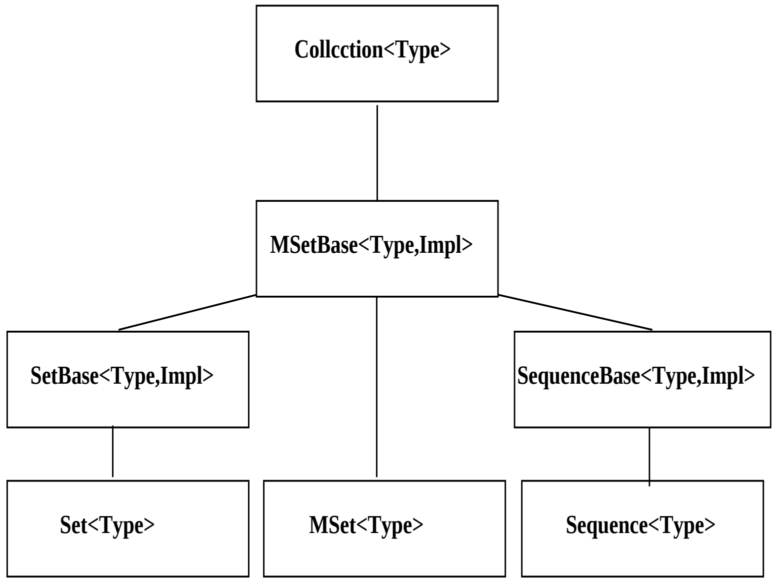


Figure 3.17: The *Collection* Hierarchy

3 *Collections*

The *Collection* hierarchy, shown in Figure 3.17, provides LINK with a flexible core of set and sequence classes that interact with each other easily, facilitating the representation of a rich variety of graphs.

3.1 Discussion

Objects of classes in the *Collection* hierarchy can be thought of as "shells" surrounding a storage medium from the *Container* class hierarchy. These shells filter the messages sent to the underlying *Container* instance, thus providing a small, clean interface to the *Container* hierarchy. Set primitive operations are defined for any two objects of the *Collection* hierarchy.

3.1.1 Organization

Class *Collection* itself is an abstract base class, meaning that there are no true "*Collection*" objects. However, pointers and references to sets and sequences of the *Collection* hierarchy have access to a consistent interface, consisting of a set of member function prototypes and a set of comparison operations. The term *collection objects* is used to refer to instances of classes in the hierarchy. The elements stored in collection objects are considered to come from a totally ordered universe. In order to compile an instantiation of a collection object with a new element type A, the following operations must be defined:

- A::int operator==(const A&) const;
- A::int operator<(const A&) const;
- friend ostream& operator<<(ostream&, const A&);

The collection objects available to the programmer are listed below and defined completely in the subsections to follow.

- MSetBase<Item, Impl> : Multiset (the implementation can be chosen by the programmer).
- SetBase<Item, Impl> : Set (the implementation can be chosen by the programmer).
- SequenceBase<Item, Impl> : Sequence (the implementation can be chosen by the programmer).
- MSet<Iteml>: Multiset which uses the DEFAULT_SET_IMPL defined in MSet.h.
- Set<Iteml>: Multiset which uses the DEFAULT_SET_IMPL defined in MSet.h.
- Set<Iteml>: Multiset which uses the DEFAULT_SEQ_IMPL defined in Sequence.h.

The comparison methods of set objects (*MSetBase*, *MSet*, *SetBase*, *Set*) work under the assumption that the elements are stored in lexicographic order. Therefore, in order for these operations to work correctly, the underlying *Container* implementation must store the elements in sorted order, or at least be capable of iterating through the elements in sorted order. With this restriction in mind, it is possible to introduce new *Container* objects into LINK and use them as *Collection* implementations.

3.1.2 Reference Counting

Aside from implementing the core set and sequence operations, the *Collection* hierarchy offers the programmer a major advantage when compared to the *Container* hierarchy.

> The copying of *Collections* is implemented *efficiently* if possible, i.e., if the source object is a descendant of the destination object, using a reference counting scheme. This is of great importance when dealing with frequently passed groups of objects such as the neighbors of a vertex. In contrast, the initialization of one *Container* with another always accomplished by iteration through all of the elements.

All class derivation in the *Collection* hierarchy is public. In general, if a child class is derived publically from a parent class, it is convenient to say that a child class instance *is a* parent class instance. For example, a sequence *is a* multiset in which the the elements can appear in any specified order. This rule can be used to determine whether or not a given copy operation will involve an increment in reference count or the iteration through all of the elements. Reference counting occurs when the object being copied *is an* instance of the destination object. Otherwise the receiving object is incompatible with the source object and each element must be copied. For example, consider the code fragment in Figure 3.18. The variable *array_set* is a multiset object which uses a *SortedArray* to store its elements. On the other hand, *list_set1* and *list_set2* use *SortedList* objects to store their elements. The reference counting scheme, which uses pointer aliasing to avoid unnecessary copying, will not work with incompatible containers. Under reference counting, it is possible for many *Collection* objects to have pointers to the same *Container* object. As long as no changes are made to this underlying implementation, such aliasing is all right. However, once a *Collection* attempts to insert or remove elements, it must be given its own copy of the elements. For examples, consider the code in Figure 3.19. When *mset2* attempts to append an additional element, sharing a common *Container* with *mset1* is no longer an option. A full copy of the elements is made before the insertion. In this manner, the integrity of *mset1* is preserved. Since `X(const X&)` constructors are called automatically upon argument pass by value and function return by value (and since in the *Collection* hierarchy they control the reference counting along with the assignment operators), *Collection* objects can be passed around efficiently. This makes the efficient programmer's memory management job much easier. For example, consider the code in Figure 3.20. The return of a temporary object cannot be accomplished by reference, so without reference counting, an efficient return would require dynamic memory allocation. But this is not necessary; the return of *Collection* objects is an efficient operation.

```
#include<iostream.h>
#include<LINK/basic/Array.h>
#include<LINK/basic/List.h>
#include<LINK/basic/MSet.h>

main()
{
        MSet<int> list_set1;

        for (int i=0; i<3; i++)
                list_set1.append(i);

        //*************reference count is incremented*********
        MSet<int> list_set2(list_set1);

        //*************complete copy is made*****************
        MSetBase<int, SortedArray<int> > array_set(list_set1);

        cout << list_set1 << endl;
        cout << list_set2 << endl;
        cout << array_set << endl;
}
```

```
{0 1 2}
{0 1 2}
{0 1 2}
```

Figure 3.18: Reference Counting in *Collections*

```
#include<iostream.h>
#include<LINK/basic/MSet.h>

main()
{
        MSet<int> mset1, mset2;

        for (int i=0; i<3; i++)
                mset1.append(i);

        mset1.insert(0);
        mset2 = mset1;          // reference count increment
        mset2.append(0);        // copy

        cout << mset1 << endl;
        cout << mset2 << endl;
}
```

```
{0 0 1 2}
{0 0 0 1 2}
```

Figure 3.19: Reference Counting and Modifications in *Collections*

```
#include<iostream.h>
#include<LINK/basic/MSet.h>

MSet<int> f()
{
        MSet<int> mset;
        mset.insert(9);
        mset.insert(10);
        mset.append(11);
        return mset;
}

main()
{
        MSet<int> mset0;
        MSet<int> mset1;
        mset0.insert(1);
        mset0.insert(2);
        mset0.append(3);

        mset1 = mset0;
        MSet<int> mset2(mset1);

        mset0 = f();            // function return takes O(1) (reference
                                // counting means that {9 10 11} needn't
                                // be recopied)
        mset1.insert(9);
        cout << mset0<< endl;
        cout << mset1 << endl;
        cout << mset2 << endl;
}
```

```
{9 10 11}
{1 2 3 9}
{1 2 3}
```

Figure 3.20: Passing and Returning Collection Objects

```
#include<iostream.h>
#include<LINK/basic/MSet.h>

main()
{
        MSet<int> mset1;
        SortedList<int> slist;

        for (int i=0; i<3; i++) {
                mset1.insert(i);
                slist.insert(i);
        }


        //List<int> list1(mset1);     // illegal - Containers must be
                                      // constructed with other Containers

        if (mset1 == slist)
                cout << "(" << mset1 <<" == " << slist << ")" << endl;
}
```

```
({0 1 2} == (0 1 2))
```

Figure 3.21: Comparisons Between *Collections* and *Containers*

3.1.3 Comparisons

Collection and *Container* objects can be compared easily using the operations defined in the *Container* class on Page 88. For example the code in Figure 3.21 illustrates a comparison between *MSet* and *SortedList* objects. *Collection* objects are compared with each other in the same way, as shown in Figure 3.22.

3.1.4 Set Primitive Operations

The three set primitive operations (union, intersection, and difference) are binary operations which take arbitrary *Collection* objects as arguments, treat them as multisets, and return the result as a multiset.

```
#include<iostream.h>
#include<LINK/basic/MSet.h>
#include<LINK/basic/Set.h>

main()
{
        MSet<int> mset1;

        for (int i=0; i<3; i++)
                mset1.insert(i);
        mset1.insert(0);

        Set<int> set1(mset1);

        if (mset1 != set1)
                cout << "(" << mset1 <<" != " << set1 << ")" << endl;

        if (mset1 <= set1)
                cout << "(" << mset1 <<" <= " << set1 << ")" << endl;

        if (mset1 < set1)
                cout << "(" << mset1 <<" < " << set1 << ")" << endl;
}
```

```
({0 0 1 2} != {0 1 2})
({0 0 1 2} <= {0 1 2})
({0 0 1 2} < {0 1 2})
```

Figure 3.22: Comparisons between *Collections*

```
#include<iostream.h>
#include<LINK/basic/MSet.h>
#include<LINK/basic/Set.h>

main()
{
        MSet<int> mset1, mset2;

        for (int i=0; i<3; i++) {
                mset1.insert(i);
                mset1.insert(i);
                mset2.insert(2*i);
                mset2.insert(2*i);
        }

        (mset1 + mset2);
        cout << mset1 << " + " << mset2 <<" = " << (mset1 + mset2) << endl;
        (mset1 ^ mset2);
        cout << mset1 << " ^ " << mset2 <<" = " << (mset1 ^ mset2) << endl;
        (mset1 - mset2);
        cout << mset1 << " - " << mset2 <<" = " << mset1-mset2 << endl;
}
```

```
{0 0 1 1 2 2} + {0 0 2 2 4 4} = {0 0 0 0 1 1 2 2 2 2 4 4}
{0 0 1 1 2 2} ^ {0 0 2 2 4 4} = {0 0 2 2}
{0 0 1 1 2 2} - {0 0 2 2 4 4} = {1 1}
```

Figure 3.23: The Set Primitive Operations

```
#include<iostream.h>
#include<LINK/basic/Set.h>      // includes MSet.h
#include<LINK/basic/Sequence.h>

main()
{
        MSet<int> mset1;
        Set<int> set;
        Sequence<int> seq1;

        for (int i=0; i<3; i++) {
                mset1.insert(i);
                mset1.insert(i);
                seq1.insert(2*i);
                seq1.insert(2*i);
        }
        mset1 ^seq1;
        cout << "MSet receives return value" << endl;
        cout << mset1 << " + " <<seq1 <<" = " << mset1+seq1 << endl;
        cout <<seq1  << " + " << mset1<<" = " <<seq1+mset1 << endl;
        cout << mset1 << " ^ " <<seq1 <<" = " << (mset1^seq1) << endl;
        cout << seq1 << " ^ " << mset1<<" = " << (seq1^mset1) << endl;
        cout << mset1 << " - " <<seq1 <<" = " << mset1-seq1 << endl;
        cout << seq1 << " - " << mset1<<" = " <<seq1-mset1 << endl;
        cout << "Set receives return value" << endl;
        cout << mset1 << " + " <<seq1 <<" = " << (set=mset1+seq1) << endl;
        cout << mset1 << " ^ " <<seq1 <<" = " << (set=(mset1^seq1)) << endl;
        cout << mset1 << " - " <<seq1 <<" = " << (set=mset1-seq1) << endl;
}
```

```
MSet receives return value
{0 0 1 1 2 2} + <4 4 2 2 0 0> = {0 0 0 0 1 1 2 2 2 2 4 4}
<4 4 2 2 0 0> + {0 0 1 1 2 2} = {0 0 0 0 1 1 2 2 2 2 4 4}
{0 0 1 1 2 2} ^ <4 4 2 2 0 0> = {0 0 2 2}
<4 4 2 2 0 0> ^ {0 0 1 1 2 2} = {0 0 2 2}
{0 0 1 1 2 2} - <4 4 2 2 0 0> = {1 1}
<4 4 2 2 0 0> - {0 0 1 1 2 2} = {4 4}
Set receives return value
{0 0 1 1 2 2} + <4 4 2 2 0 0> = {0 1 2 4}
{0 0 1 1 2 2} ^ <4 4 2 2 0 0> = {0 2}
{0 0 1 1 2 2} - <4 4 2 2 0 0> = {1}
```


Figure 3.24: Set Primitives with Set and Sequence Operands

The example in Figure 3.23 illustrates the three set primitive operations applied to multisets, while the more extensive example shown in Figure 3.24 shows the treatment of sequences as sets during these operations. Note that the type of the object receiving the return value of each operation determines whether the result is a set or a multiset.

3.1.5 Interface

The following methods are available to pointers and references to *Collection* objects. Those that are pure virtual are defined in class *MSet*. virtual ˜*Collection*<item>()

The virtual destructor in the *Collection* class ensures that the proper subclass destructor will be called when the object referenced or pointed to is deleted. The new data structure must define its own destructor (or inherit one from a parent *Collection*).

int size() const = 0;

Return the number of stored elements.

Bool emptyQ() const = 0;

Return TRUE if there are no stored elements.

Bool fullQ() const = 0;

Return TRUE if there are no space left for additional elements.

Bool memberQ(const Item&e) const = 0;

Return TRUE if e is stored.

Bool sortedQ() const = 0;

Does the *Container* used to store the elements store them in sorted order?

DataType type() const = 0;

Return the type of this structure (*DataType* is an enumerated type defined in general.h).

Item first() const = 0;

Return the first element in the *Container* in which the *Collection* stores its elements.

Item last() const = 0;

Return the last element in the *Container* in which the *Collection* stores its elements.

void insert(Item e) = 0;

Insert *e* into the *Container* in which the *Collection* stores its elements.

void append(Item e) = 0;

Insert *e* as the last element of the *Container* which stores the elements if this is possible. Otherwise, i.e. the *Container* is sorted, just call insert()

void remove(Item e) = 0;

Remove *e* from the *Container* in which the *Collection* stores its elements.

void clear() = 0;

Remove all elements from the *Container* used to store the elements.

int iterate(Iterator<Item>& it, Item& e) const = 0;

This method is used by *Iterator* objects to obtain the successor of e. Typically, a *Collection* object will implement this by calling its *Container*'s iterate() method.

Item ref(int i) = 0;

Return the element with index i.

int rank(Item it) = 0;

Return the index of item it.

int occurrences(Item) = 0;

Returns the number of occurrences of the passed item.

Bool subsetQ(Collection<Item>& c) = 0;

Return TRUE if every occurrence of an element e in the current collection is matched with an occurrence of the same element in c.

Bool properSubsetQ(Collection<Item>& c) = 0;

Return TRUE is the current collection is a subset of c and has fewer elements.

int permutationQ();

Returns FALSE if duplicate elements exist.

void concatenate(const Collection<Item>& = 0);

Add the elements of the passed *Collection* by iterating through them and using append().

```
ostream&    display(ostream&os) = 0;
```

This will be called when *operator<<(ostream&,Collection<Item>&)* is invoked.

```
Collection<Item>*    newEmpty() const = 0;
```

Dynamically allocates a new *Collection* of the same type as the current one and returns it. This is useful in copying graphs and other situations, where the only access to the *Collection* is through a *Collection* pointer (i.e. the type of *Collection* is not known). $O(1)$

```
Bool operator==(const Collection<Item>&) const = 0;
Bool operator<=(const Collection<Item>&) const = 0;
Bool operator< (const Collection<Item>&) const = 0;
Bool operator>=(const Collection<Item>&) const = 0;
Bool operator> (const Collection<Item>&) const = 0;
Bool operator!=(const Collection<Item>&) const = 0;
Bool operator==(const Container<Item>&) const = 0;
Bool operator<=(const Container<Item>&) const = 0;
Bool operator< (const Container<Item>&) const = 0;
Bool operator>=(const Container<Item>&) const = 0;
Bool operator> (const Container<Item>&) const = 0;
Bool operator!=(const Container<Item>&) const = 0;
```

Compare the *Collection* with a *Collection* or *Container* argument (by calling `Container::operator...()`)

3.2 *MSetBase* and *MSet*

Most of the methods declared pure virtual in the *Collection* class are given definitions in class *MSetBase*. The latter is the only non-abstract class from which all other *Collection* objects inherit. It is templated both by element type and by implementation type in order to allow the programmer the flexibility of customizing sets and sequences to for various applications. The descendant classes of *MSetBase* inherit most of their functionality from it, and redefine only those methods which distinguish them. The comparison, set primitive, subset, and output operations are inherited and used by all

```
#include<iostream.h>
#include<LINK/basic/Array.h>
#include<LINK/basic/MSet.h>
#include<LINK/basic/Set.h>

main()
{
        MSetBase<int, SortedArray<int> > mset1;
        MSet<int> mset2;                        // default impl:SortedList

        for (int i=0; i<3; i++) {
                mset1.insert(i);
                mset2.insert(i);
        }

        mset2.insert(0);
        cout<<"there are "<<mset2.occurrences(0)<<" zeros in "<< mset2<<endl;

        if (mset1.properSubsetQ(mset2))
                cout << mset1 << " is a proper subset of " << mset2 << endl;
        if (mset1.subsetQ(mset2))
                cout << mset1 << " is a subset of " << mset2 << endl;

        mset2.remove(0);

        if (mset1.properSubsetQ(mset2))
                cout << mset1 << " is a proper subset of " << mset2 << endl;
        if (mset1.subsetQ(mset2))
                cout << mset1 << " is a subset of " << mset2 << endl;

}
```

```
there are 2 zeros in {0 0 1 2}
{0 1 2} is a proper subset of {0 0 1 2}
{0 1 2} is a subset of {0 0 1 2}
{0 1 2} is a subset of {0 1 2}
```

Figure 3.25: *MSetBase* and *MSet*

other *Collections*. This allows various sets and sequences to be used together very naturally, a property on which the graph hierarchy depends. Two fundamental assumptions about *Containers* used as *Collection* implementations apply:

1. Multiset and Set objects should be stored in sorted *Containers*.
2. Sequence objects *must* be stored in unsorted *Containers*.

If two *Collections* are stored in *Containers* which store elements in sorted order, then the comparison, set primitive, and subset operations take linear time in the worst case. Otherwise the respective time complexities worsen, and output will not be as neat. However, *MSetBase* objects using unsorted *Containers* as implementations are permitted. On the other hand, sequences must be implemented using unsorted *Containers*. Otherwise, the only possible ordering of n elements is lexicographic order. *MSet* objects are simply *MSetBase* objects in which the default set implementation *Container* (DEFAULT_SET_IMPL defined in general.h) is used to store the elements. **1. Creation**

MSetBase<Item,Impl> ms;

Create an empty multiset using the *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation.

MSetBase<Item,Impl> ms((Collection<Item>&) c);

Create a multiset using the *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation. Initialize the collection with the elements of c. $O(n)$

MSetBase<Item,Impl> ms((MSetBase<Item,Impl>&) m);

Create a multiset using the *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation. Use reference counting to share m's implementation. $O(1)$

MSet<Item> ms;

Create an empty multiset using the implementation specified by DEFAULT_SET_IMPL in MSet.h. Initialize it with the elements of c. $O(n)$

MSet<Item> ms((Collection<Item>&) c);

Create a multiset using the implementation specified by DEFAULT_SET_IMPL in MSet.h. Initialize it with the elements of c. $O(n)$

MSet<Item> ms((MSet<Item>&) m);

Create a multiset using the default *Container* class as an implementation. Use reference counting to share m's implementation. $O(1)$

2. Operations

virtual ˜*MSetBase*<Item,Impl>()

Decrement the reference count of the *Container* which stores the elements (this container object is called `store`). $O(1)$. If the count reaches 0, delete the object. $O(n)$

int size() const;

Return the number of stored elements by calling *store→size()*.

Bool emptyQ() const;

Return TRUE if there are no stored elements (call *store→emptyQ()*).

Bool   fullQ() const;

Return TRUE if there are no space left for additional elements (call *store→fullQ()*).

Bool   memberQ(const Item&e) const;

Return TRUE if e is stored (call *store→memberQ()*).

Bool   sortedQ(const Item&e) const;

Return TRUE if `store` is stored (call *store→sortexQ()*).

DataType   type() const;

Return the type of this structure (*DataType* is an enumerated type defined in general.h).

Item   first() const;

Return the first element in `store` (call *store→first()*).

Item   last() const;

Return the last element in `store` (call *store→last()*).

void   insert(Item e);

If the reference count of `store` is greater than 1, then other collection objects are also using it as their implementation. Rather than changing its contents, a copy will be made ($O(n)$), and the reference counts will be updated. Finally, Call *store→insert(e).* to insert e.

void append(Item e);

After considering reference counts as above, call *store→append(e).* to insert e at the end, if sorted order permits (see *SortedList*).

void remove(Item e);

After considering reference counts as above, remove e from `store`.

void clear();

After considering reference counts as above, remove all elements from `store`.

int iterate(Iterator<Item>& it, Item& e) const;

Return *store→iterate(it,e).* Note that some typecasting is required, as *iterate()* is not a publically-accessible member of any of the descendants of *Container*. The *iterate()* method needn't ever be called by the programmer; use *Iterator* objects.

Item ref(int i);

Return the element with index i. $O(n)$

int rank(Item it);

Return the index of item it. $O(n)$

int occurrences(Item);

Returns the number of occurrences of the passed item by iterating through all of the elements. $O(n)$

int permutationQ();

Returns FALSE if duplicate elements exist. $O(n)$ if `store` is a sorted *Container*. $O(n^2)$ otherwise.

void concatenate(const Collection<Item>&);

Add the elements of the passed *Collection* by iterating through them and using append().

ostream& display(ostream&os);

This will be called when *operator<<(ostream&,Collection<Item>&)* is invoked.

Bool operator==(const Collection<Item>&) const;
Bool operator<=(const Collection<Item>&) const;
Bool operator< (const Collection<Item>&) const;
Bool operator>=(const Collection<Item>&) const;
Bool operator> (const Collection<Item>&) const;
Bool operator!=(const Collection<Item>&) const;
Bool operator==(const Container<Item>&) const;
Bool operator<=(const Container<Item>&) const;
Bool operator< (const Container<Item>&) const;
Bool operator>=(const Container<Item>&) const;
Bool operator> (const Container<Item>&) const;
Bool operator!=(const Container<Item>&) const;

Return *store→operator...(..)*.

Bool eq(const MSetBase<Item,Impl>&s);

This alternative to `operator==(...)` is provided for access from the TCL interface.

Bool neq(const MSetBase<Item,Impl>&s);

This alternative to `operator!=(...)` is provided for access from the TCL interface.

MSetBase<Item,Impl> unions(Collection<Item>& c);
MSetBase<Item,Impl> operator+(Collection<Item>& c);

Construct the union of the current multiset and c as a *MSetBase<Item,Impl>* object. The number of occurrences of an element in the union is the sum of the numbers of occurrences in the two original multisets. Note however that if the object receiving the return value of this method is a *SetBase* or *Set* object, duplicates are removed. $O(n^2)$ if either of the *Collections*are unsorted, $O(n)$ otherwise. (the $O(n^2)$ could be improved with a more involved implementation which builds frequency tables.

MSetBase<Item,Impl> intersection(Collection<Item>& c);
MSetBase<Item,Impl> operatorˆ(Collection<Item>& c);

Construct the intersection of the current multiset and c as a *MSetBase<Item,Impl>* object. The number of occurrences of an element e in the result is the minimum of the numbers of occurrences of e in the two original mutlisets. $O(n^2)$ if either of the *Collections*are unsorted, $O(n)$ otherwise. (the $O(n^2)$ could be improved with a more involved implementation which builds frequency tables.

MSetBase<Item,Impl> difference(Collection<Item>& c);
MSetBase<Item,Impl> operator-(Collection<Item>& c);

Construct the difference between the current multiset and c as a *MSetBase<Item,Impl>* object. The number of occurrences of an element e in the result $(n\_r(e))$ is defined as $(n\_\mathrm{this}(e) - n\_c(e)))$. $O(n^2)$ if either of the *Collections*are unsorted, $O(n)$ otherwise. (the $O(n^2)$ could be improved with a more involved implementation which builds frequency tables.

Bool subsetQ(Collection<Item>& c);

Return TRUE if every occurrence of an element e in the current multiset is matched with an occurrence of the same element in c. $O(n)$ if both *Collections* are sorted, otherwise $O(nq)$ where q is the complexity of *store→memberQ(e)*.

Bool properSubsetQ(Collection<Item>& c);

Return TRUE is the current multiset is a subset of c and has fewer elements. $O(n)$ if both *Collections* are sorted, otherwise $O(nq)$ where q is the complexity of *store→memberQ(e)*.

int occurrences(Item item);

Iterate through the list and return the number of occurrences of *item*. $O(n)$

ostream& display(ostream& os) const

Display the multiset on the output stream *os*. $O(n)$

Collection<Item>* newEmpty() const;

Dynamically allocates a new *MSetBase* or *MSet* object of the same type as the current one and returns it. This is useful in copying structures when the only access to a *Collection* is through a *Collection* pointer (i.e. the type of *Collection* is not known). $O(1)$

3.3 *SetBase* and *Set*

Sets which contain no duplicate elements are obtained by defining *SetBase* and *Set* objects, which inherit very heavily from class *MSetBase*. The only difference is that insertions are guarded by a membership test to maintain the invariant the no duplicate elements ever exist in a set. The methods which differ from those of *MSetBase* are listed below. **1. Creation**

SetBase<Item,Impl> s;

Create an empty set using the *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation.

SetBase<Item,Impl> s((Collection<Item>&) c);

Create a set using the *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation. Initialize the collection with the elements of c. Note that although c may be a multiset, no duplicate elements are inserted into s. $O(nq)$, where q is the complexity of the *Container*'s `memberQ()` method.

SetBase<Item,Impl> s((SetBase<Item,Impl>&) ss);

Create an empty multiset using the *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation. Use reference counting to share the elements of ss. $O(1)$

Set<Item> s((Collection<Item>&) c);

Create a set using the implementation specified by DEFAULT_SET_IMPL in MSet.h. Initialize it with the elements of c. $O(nq)$, where q is the complexity of the *Container*'s `memberQ()` method.

Set<Item> s((Set<Item>&) ss);

Create an empty multiset using the implementation specified by DEFAULT_SET_IMPL in MSet.h. Use reference counting to share the elements of ss. $O(1)$

2. Operations

void insert(Item e);

If the reference count of `store` is greater than 1, then other collection objects are also using it as their implementation. Rather than changing its contents, a copy will be made ($O(n)$), and the reference counts will be updated. Finally, check to ensure that e is not currently a member, then call *store→insert(e).* to insert e.

void append(Item e);

After considering reference counts as above, check to ensure that e is not a member, then call *store→append(e).*

void remove(Item e);

After considering reference counts as above, remove e from `store`.

int occurrences(Item);

Return 1 or 0. $O(1)$.

int permutationQ();

Always returns TRUE. $O(1)$.

3.4 *SequenceBase* and *Sequence*

*Collections*capable of storing data elements in any order are *SequenceBase* and *Sequence* classes. Like *SetBase* and *Set* these classes inherit the vast majority of their functionality from *MSetBase*. Unlike set objects, however, sequence objects *must* use unsorted *Container* objects as storage media. Error checks in the *SequenceBase* and *Sequence* constructors prevent the use of any *Container* object which returns TRUE when given the `sortedQ()` query. *Sequence* The main advantage of derivation from *MSetBase* is illustrated in Figures 3.26 and 3.27. Since a sequence *is a* multiset which happens to

```
#include<iostream.h>
#include<LINK/basic/MSet.h>
#include<LINK/basic/Sequence.h>

main()
{
        MSet<int> mset1;
        Sequence<int> seq;
        seq.insert(1);
        seq.insert(2);
        seq.append(3);
        cout << seq << endl;

        for (int i=0; i<3; i++) {
                mset1.insert(i);
        }
        Sequence<int> seq1(mset1);

        cout << mset1 << endl;
        cout << seq1 << endl;
        if (mset1 == seq1)
                cout << "(" << mset1 <<" == " << seq1 << ")" << endl;
        seq1.append(-2);
        if (mset1 <= seq1)
                cout << "(" << mset1 <<" <= " << seq1 << ")" << endl;
}
```

```
<2 1 3>
{0 1 2}
<0 1 2>
({0 1 2} == <0 1 2>)
({0 1 2} <= <0 1 2 -2>)
```

Figure 3.26: *Sequence*

be stored in an unsorted *Container* object, it can appear as an operand on either side of the set comparison, set primitive, and set construction methods. The methods which differ from those of *MSetBase* are listed below. **1. Creation**

SequenceBase<Item,Impl> s;

Create an empty sequence using the unsorted *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation.

SequenceBase<Item,Impl> s((Collection<Item>&) c);

Create a sequence using the unsorted *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation. Initialize the collection with the elements of c such that the same order is maintained. $O(n)$

SequenceBase<Item,Impl> s((SequenceBase<Item,Impl>&) ss);

Create an empty sequence using the unsorted *Container* class *Impl* (where *Impl* contains elements of type *Item*) as an implementation. Use reference counting to share the elements of ss. $O(1)$

Sequence<Item> s((Collection<Item>&) c);

Create a set using the implementation specified by DEFAULT_SET_IMPL in MSet.h. Initialize it with the elements of c such that the same order is maintained. $O(n)$

Sequence<Item> s((Sequence<Item>&) ss);

Create an empty sequence using the implementation specified by DEFAULT_SET_IMPL in MSet.h. Use reference counting to share the elements of ss. $O(1)$

```
#include<iostream.h>
#include<LINK/basic/MSet.h>
#include<LINK/basic/Sequence.h>

MSet<int> ms()
{
        MSet<int> mset;
        mset.insert(12); mset.insert(13);
        return mset;
}

Sequence<int> sq()
{
        Sequence<int> seq;
        seq.insert(9); seq.append(10);
        return seq;
}

main()
{
        Sequence<int> seq0(ms());       // copy is made
        Sequence<int> seq1(sq());       // reference counting eliminates
                                        // need for copy
        cout << seq0 << endl;
        cout << seq1 << endl;
}
```

```
<12 13>
<9 10>
```

Figure 3.27: Another *Sequence* Example

void insertAfter(Item e, Item after_item);

If the reference count of `store` is greater than 1, then other collection objects are also using it as their implementation. Rather than changing its contents, a copy will be made ($O(n)$), and the reference counts will be updated. Finally, insert e after the first occurrence of `after\_item`. If there is no such occurrence, a NULL pointer is passed to the *Container*'s *insertAfter(Item, ContainerNode*)* method. Complexity depends on the storage medium.

void insertBefore(Item e, Item before_item);

Similar to *insertAfter()* except that e is inserted before the first occurrence of `before\_item`.

void prepend(Item e);

After considering reference counts as above, call *store→prepend(e)*, which should insert e at the beginning. Note that this usually has the same result as calling *insert*.

Item element(int i);

Return the ith element, or the last element if i¿this->size() $O(n)$.

SequenceBase<Item,Impl> subsequence(int i, int j);

Construct a new sequence and initialize it with $s[i] \ldots s[j]$, where $s[p]$ is the pth element of the current sequence. $O(n)$

4 Combinatorial Objects

Permutations, subsets, and set partitions are combinatorial objects are both useful in describing properties of graphs and objects with interesting properties of their own. One distinguishing property of combinatorial objects is that there are a finite number of them of any given size, and that a lexicographic order can be defined on them. Thus they can be ranked and unranked, and constructed at random.

4.1 Commonalities Among Combinatorial Objects

Permutations and sets of subsets are grouped into a class called *SetFuncs*. This is useful to help instantiate templates in an orderly manner, though eventually compilers should reach a state where this is no longer a concern. The combinatorial objects described below are represented with *Collections*, *Collections*of *Collections*, and operations on these. In this way, all of the commonalities between *Collections*(copying, comparisons, set primitives, etc.) apply.

4.2 Permutations

A permutation of a set of objects is a particular arrangement of them. Permutations can be throught of as either arrangements, or an operation which rearranges a set of objects. Permutations of the numbers $1, \ldots, n$ can describe a particular arrangment of arbitrary objects (and hence are an operation). The *Permutation* object describes such an arrangement, and can be applied to a *Sequence*of typed objects to obtain a permutation of the latter.

1. Creation

Permutation permutation_name(int n)

Create the identity permutation $1, \ldots, n$.

Permutation<Item> permutation_name(Set s)
Permutation<Item,Structure> permutation_name(Set s)

Create a permutation whose elements are the elements of set s. The initial permutation corresponds to the sorted order of these element Structure is the Container implementation used to maintain the permutation's local copy of s.

2. Operations

Permutation& random()

Rearrange the permutation in random order.

int rank()

Return the rank of the permutation, ie. the position of the permutation in lexicographic order. The ranks run from 0 to $n! - 1$.

Permutation& unrank(int m)

Unrank the mth permutation, ie. change the current permutation to the mth permutation in lexicographic order.

Permutation& next()

Change the permutation to its successor in lexicographic order.

Permutation& previous()

Change the permutation its predecessor in lexicographic order.

Permutation& first()

Change the permutation to the first permutation in lexicographic order, the identity permutation.

Permutation& last()

Change the permutation to the last permutation in lexicographic order, the reverse of the identity permutation.

void display()

Display the permutation in human-readable form.

Bool operator==(Permutation& p)
Bool operator>(Permutation& p)
Bool operator>=(Permutation& p)
Bool operator<(Permutation& p)
Bool operator<=(Permutation& p)
Bool operator!=(Permutation& p)

Compare two permutations according to lexicographic order.

friend istream& operator>>(istream& s,Permutation p)

Input p from input stream s.

friend ostream& operator<<(ostream& s,Permutation p)

Output p to output stream s.

Permutation& operator=(Permutation p)
Permutation& copy(Permutation p)

Creates a copy of permutation p.

Type operator[] (int i)

Return the value of the ith element of the permutation. If the permutation was constructed from the element of the set, an element is returned, otherwise it is an integer from 1 to n.

int size()

Return size of the permutation

Permutation objects can be used to obtain permutation of typed objects like vertices and edges as follows. The class *SetFuncs*, provides the necessary interface. Note that several of the methods in this class produces results of exponential in the size of the input. Set<Sequence<Item> > SetFuncs<Item>::cartesianProduct(const Set<Item>&s, const Set<Item>&s2);

Constructs and returns the set of all ordered pairs in the cartesian product of s and s2.

Set<Sequence<Item> > SetFuncs<Item>::cartesianProduct(const Set<Item>&s1, const Set<Item>&s2)

Constructs and returns the set of all k-subsets of s.

Set<Set<Item> > SetFuncs<Item>::choose(const Set<Item>&s, int k)

Constructs and returns the set of all k-subsets of s.

Set<Set<Item> > SetFuncs<Item>::powerSet(const Set<Item>&s)

Constructs and returns the set of all possible subsets s.

Sequence<Item> SetFuncs<Item>::permutation(const Set<Item>&s, const Permutation& p);

Permutes the elements of s into the order specified by permutation p.

Sequence<Item> SetFuncs<Item>::permutation(const Array<Item>&a, const Permutation& p);

Permutes the elements of a into the order specified by permutation p.

Set<Sequence<Item> > SetFuncs<Item>::permutations(const Set<Item>&a);

Returns a set of all permutations of s.

2 *Graph* and *GraphObject*

The *Graph* class and its descendants provide the programmer with 12 different types of graph and hypergraph objects. The following three types of restriction of a graph's edge set determine the graph classes.

1. *Multiset or not?* This determines whether the graph is a multigraph or a simple graph.

2. *Cardinality of each edge equals 2?* This determines whether the graph is a graph (also called *binary graph*) or a hypergraph.

3. *Directed, Undirected, or Mixed?* In directed graphs, every edge must consist of a *Sequence* of vertices. In undirected graphs, each edge must consist of a *Set* of vertices. LINK also offers the option of defining *mixed graphs*, in which both directed and undirected edges may occur in the same graph.

Restrictions 1 and 3 are implemented using specific objects of the *Collection* hierarchy as the *Vertex* sets of *Edges*. Restriction 2 is implemented with simple test. Each of the classes determined by these criteria is described in detail in Section 3. *Graphs*, *Vertices*, and *Edges* have some commonality in that they each have character string names and may have other attributes, and in that they each are "owned" by a *Graph* object. *Graph* objects may be owned by themselves, or by a parent *Graph* through the subgraph mechanism (see Section 2.7.1). The *GraphObject* class exists as an abstract parent to all three classes to avoid redefinition of common operations. It contains methods which the *Graph* classes call access and modify attributes, but there is a more natural for the LINK programmer to use an alternate method, which will be described in Section 2.2.

2.1 *Graph* Representation

Simple graphs are commonly represented using adjacency lists, in which each vertex contains a pointer to a list of its neighbors. There is typically no explicit representation of edges; v_i is said to be adjacent to v_j if v_j is in the adjacency list of v_i. In LINK, however, this representation is not sufficient. A hypergraph G is a pair (V, E), where $V = \{v_0, v_1, \ldots, v_{n-1}\}$ is a set of vertices and $E = \{e_0, e_2, \ldots, e_{m-1}\}$ is a collection of finite subsets of V. Since edges may have cardinalities not equal to 2, adjacency list membership is not sufficient to determine the neighbors of a vertex. In LINK, hypergraphs are represented by a *SortedArray* of *Vertex* pointers and an *MSet* of *Edge* pointers. The *Container* object *SortedArray* is used instead of *Set< Vertex*, SortedArray< Vertex*>* > in order to allow the programmer to circumvent the *Iterator* mechanism and iteration through the *Vertex* set using fast pointer arithmetic. Membership checks prevent duplicate *Vertices* from occurring. Each *Edge* object contains a *Collection* pointer which locates the set or sequence storing the *Vertices* in that *Edge*. The virtual functions of the *Collection* class can allow a single algorithm code to work on many different classes of *Graph* objects with no modification. The bottleneck in accomplishing this in general is being able to define an algorithm's performance on various classes of *Graph* rather than language or system limitations.

2.2 *GraphObject* Attributes

Although there is an *Attribute* class in the *graph* directory of LINK's file hierarchy, objects of this class are typically not instantiated by the programmer. The reason for this is that *Attribute* objects, though owned by a specific *Graph* object G, are accessible to any *GraphObject* (*Vertex*, *Edge*, or *Subgraph*) which belongs to G. The programmer's interface to the attribute mechanism consists of the following three templated functions. This is one of the very very instances of the use of non-member functions in LINK programming.

```
template <class Item>
int  newAttribute(Graph*, String name, const Item&);
template <class Item>
int  setAttribute(GraphObject*, String name, const Item&);
template <class Item>
int  getAttribute(GraphObject*, String name, Item&);
```

An example illustrating the use of these functions is presented in Figure 2.2. The *newAttribute(...)* function dynamically allocates an *Attribute¡Item¿* object and associates it with the passed *Graph*. The default value of the attribute is stored in this object and accessed with the *setAttribute(...)* function. The latter may take as its first argument not only a pointer to a *Graph* object, but also a pointer to any instance of the *GraphObject* hierarchy. Thus, even though the actual *Attribute* object is "owned" by the graph, any of its vertices and edges may access its default value. In order to set differing attribute values for individual vertices and edges of a graph, the

```
#include<iostream.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/graph/Vertex.h>
#include<LINK/graph/Attribute.h>

void main()
{
        int i;
        DBinGraph g1;
        if (newAttribute((Graph*)&g1, "test_attr", 1) != LINK_OK)
                error("Attr.cpp: error creating test_attr");

        g1.addVertices(4);
        Vertex** vertices = g1.vertexStart();
        int num_vertices = g1.order();

        for (i=2; i<num_vertices; i++) {
                setAttribute((GraphObject*)vertices[i], "test_attr", i);
        }
        int value;
        for (i=0; i<num_vertices; i++) {
                getAttribute((GraphObject*)vertices[i], "test_attr", value);
                cout << "v: "<<*vertices[i]<<" test_attr: "<<value<<endl;
        }
}
```

```
v: 0 test_attr: 1
v: 1 test_attr: 1
v: 2 test_attr: 2
v: 3 test_attr: 3
```

Figure 2.2: Graph *Attributes*

setAttribute(...) function is used. The call format is the same as in the other two. Recall that the *Attribute* object itself is owned by the *Graph* object. When a call to *setAttribute(...)* asks for some *Vertex* or *Edge* to replace the default value with a new value, a *copy* of the *Attribute* object is obtained, associated with the *Vertex* or *Edge*, and given the new value. Subsequent calls to *getAttribute(...)* for the affected *Vertex* or *Edge* return the local attribute value rather than the graph's default value. Note that this implies that even if an attribute is intended to apply only to a graph's vertices, it does exist in the graph and can be given values for the graphs edges also. For example, it is unnecessary to employ two separated attributes *vertex_weight* and *edge_weight*. One *weight* attribute is sufficient. The *Attribute* class imposes only one restriction on a data type A used as an attribute value: that the operator function `ostream& operator<<(ostream&, const A&)` be defined. Subject to this requirement, any data type may be used as a graph attribute. However, there are restrictions in the types of attributes that can be stored in a file with the graph (See Section 2.7.2). **Warning:** Since the three attribute functions described above are templated functions, the actual parameters in the a function call must match the formal parameters exactly. Otherwise, a "*type unification failed*" will result (g++). The first argument of each function is an abstract base class pointer, so it may seem natural to pass in addresses of descendent objects. This must be avoided. Note that the calls in Figure 2.2 use explicit typecasts to match the function template.

2.3 The Default Graph Object Attributes

- *back* [int] Set by Depth-first search for each vertex v, this attribute stores the discovery time of the earliest ancestor which v is adjacent to via a back edge.
- *color* [char *] This attribute, commonly associated with both vertices and edges, takes on string values representing Tk colors. See the Tk manual for details, but the values are typically very straightforward, e.g. "blue" or "green." The STk interface uses this value to determine the actual color of each vertex or edge (and can modify it).
- *direction* [int] This outdated attribute stores information indicating whether an edge object is directed or undirected. A more natural way to tell this is: *e-¿vertices()-¿sortedQ()*, where an answer of "yes" means undirected.

- *distance* [double] Distance calculations such as shortest paths use and modify this attribute, usually associated with vertices.
- *name* [char *] This attribute stores the name by which this graph or graph object is identified. This attribute should never change.
- *finishtime* [int] Depth-first search sets this attribute as it finishes searching the neighbors of each vertex. Ordering the vertices by *finishtime* yields a postorder walk.
- *label* [char *] The *label* of a vertex or edge is a string of text currently associated with the object. Users and algorithms may change this from the interface and from C++ programs.
- *low* [int] Set by Depth-first search for each vertex v, this attribute stores the discovery time of the earliest proper ancestor to which any child of v is adjacent to via a back edge.
- *mark* [int] This general purpose attribute is provided to call attention to special vertices or edges. For example, the strongly connected components algorithm sets this mark for each component leader.
- *partition* [int] Various graph algorithms partition the vertex and/or edge sets of a graph into two or more parts. This attribute is used to store the locations of graph objects.
- *pred* [Vertex*] Graph searches induce a tree which is very useful for constructing more advanced algorithms. This tree is stored in the form of *predecessor* information for each vertex. The root of the search tree is found by following *pred* pointers as far as possible.
- *size* [int] This attribute is usually associated only with Vertex objects, and specifies the size in pixels of the vertex on the screen.
- *starttime* [int] Depth-first search sets this attribute as it discovers each vertex. Ordering the vertices by *starttime* yields a preorder walk.
- *type* [int] This is another general purpose attribute. One examples usage is to store the classification of an edge e during and after a depth-first search.
- *width* [int] The *width* of an edge is the screen width in pixels. Common values range from 1 to 4.

- *weight* [double] Graphs with vertex and edge weights arise in many, many applications. This double- precision attribute stores the weight of a Vertex or Edge object.
- x [double] This is the "x" coordinate of a Vertex object or the label (and control point) of an Edge object. The value is expected to be in world coordinates, where the world ranges from 0 to 1 in each direction.
- y [double] Similar to "x."

2.4 Building Graphs

Graph* bipartiteRepresentation(Graph *g);

Primarily useful for obtaining alternate representations for hypergraphs, this function creates a new binary graph whose vertex set is the union of the vertex and edge sets of g. A Vertex v representing edge e of g is adjacent to the vertices representing vertices of g which are members of e.

int compareVertexNames(char *, char *);

This is the default method for comparing vertices. Vertex names are character strings, but those strings most often contain digits. Standard lexicographic ordering of strings is therefore not very convenient most of the time. This routines determines if the strings represent integers, and if so, compares their integer values.

Graph* makeGraph(Set<Vertex*>, MSet<Edge*>);
Graph* makeGraph(MSet<Vertex*> vset, MSet<Set<Vertex*> > eset);
Graph* makeGraph(MSet<Vertex*> vset, MSet<Sequence<Vertex*> > eset);
Graph* makeGraph(MSet<char*> vset, MSet<Set<char*> > eset);
Graph* makeGraph(Set<char*> vseq, MSet<Sequence<char*> > eseq);
Graph* makeGraph(Set<int*> vset, MSet<Set<int*> > eset);

These routines dynamically allocate and construct a new graph of the appropriate type given the passed vertex and edge sets.

Graph* directedGraph(const Graph* g);

Dynamically allocate a new empty graph of the same type as g, except if g is not directed. In that case, dynamically allocate a directed graph of the corresponding type.

Graph* multiGraph(const Graph* g);

Dynamically allocate a new empty graph of the same type as g, except if g is a simple graph. In that case, dynamically allocate a multigraph of the corresponding type.

graph* nautyGraph(const Graph* g);

Given a LINK graph object, return a graph in the format of *nauty*, the system due to Brendan McKay for identifying automorphism groups and testing isomorphism.

Graph* newEmpty()onst Graph* g);

Dynamically allocate a new empty graph of the same type as g.

Graph* simpleGraph(const Graph* g);

Dynamically allocate a new empty graph of the same type as g, except if g is a simple graph. In that case, dynamically allocate a multigraph of the corresponding type.

Graph* undirectedGraph(const Graph* g);

Dynamically allocate a new empty graph of the same type as g, except if g is not undirected. In that case, dynamically allocate an undirected graph of the corresponding type.

2.5 *Vertex*Methods

Each *Vertex* object contains an *MSet* of *Edge* pointers representing its incident edges. Note that the storage medium is a multiset rather than a set because, in a multigraph, an *Edge* might be incident to the same *Vertex* twice. *Vertex* objects are never constructed explicitly by the programmer. See the *Graph* methods for adding vertices. **2. Operations**

~*Vertex*()

Clears the edge set before deleting the current *Vertex* object.

void addEdge(Edge* passed_edge)

Add *passed_edge*, which has already been constructed, to the edge set. NOTE: THIS METHOD SHOULD BE PRIVATE, BUT IT CANNOT BE RIGHT NOW SINCE THE FILE FORMAT PARSER CALLS IT (THE REASON FOR THE EXISTENCE OF THIS METHOD IS TO MANIPULATE SUBGRAPHS, WHICH MAY BE STORED. THE SUBGRAPH PROCESSING IN THE GRAPH CLASSES CAN BE ACCOMPLISHED BY MAKING THEM FRIENDS, BUT THE PARSER IS NOT A CLASS). PROGRAMMERS SHOULD NEVER CALL THIS. IT WILL BE MADE PRIVATE EVENTUALLY.

void removeEdge(Edge* passed_edge)

Remove *passed_edge* from the edge set. NOTE: THE COMMENT ABOVE APPLIES TO THIS METHOD ALSO.

char* name() const

Retrieve and return the *name* attribute of the vertex.

DataType type() const

Return VERTEX. (see general.h)

MSet<Edge*> incidentEdges() const

Return the edge set. Note that this is an $O(1)$ operation due to the reference counting of the *Collection* hierarchy (Returning a constant reference to the *MSet* is another alternative, but g++ 2.6.3 does not protect the set in that case; it only generates a warning).

Set<Vertex*> neighbors()

If the graph is undirected, return all of the vertices which share an edge with this *Vertex* object (not including itself), otherwise return outNeighbors(). $O(ds)$, where d is the maximum degree of a vertex in G, and s is the maximum cardinality of an edge.

Set<Vertex*> inNeighbors()

For each incident edge, if the edge is directed, find all vertices which occur *before* the current vertex in the sequence. If the edge is undirected, return all verticies in the edge other than the current vertex. $O(ds)$, where d and s are defined as above.

Set<Vertex*> outNeighbors()

For each incident edge, if the edge is directed, find all vertices which occur *after* the current vertex in the sequence. If the edge is undirected, return all verticies in the edge other than the current vertex. $O(ds)$, where d and s are defined as above.

Vertex* randomNeighbor()

Randomly select and return a neighbor (an out-neighbor if the graph is directed. $O(1)$

Bool operator==(const Vertex&) const
Bool operator<=(const Vertex&) const
Bool operator< (const Vertex&) const
Bool operator>=(const Vertex&) const
Bool operator> (const Vertex&) const
Bool operator!=(const Vertex&) const

Compare the current vertex with another based on lexicographic ordering of vertex names.

void saveToFile(ofstream& fout, int indent)

This method outputs the infomation associated with the current vertex, including its attributes. It is not generally called by the programmer; *MHyperGraph::saveToFile()* calls it while writing out a graph.

friend ostream& operator<< (ostream& stream, const Vertex& v)

Output the name of the vertex to a stream.

2.6 *Edge*

Edge objects use the flexibility of the *Collection* hierarchy to allow any given edge to be directed or undirected. A pointer to a *Collection* of *Vertex* pointers is stored in the *Edge* object. If this pointer points to a sequence, then the edge takes on properties of a directed edge. Otherwise the edge behaves as an undirected edge. Thus, *directed hyperedges* are simply defined to be sequences of vertices of varying length. As described briefly in the *Vertex* methods above, the *inNeighbors()* of a *Vertex* object v with respect to a directed *Edge* object e are those *Vertices* which occur *before* v in the e's vertex sequence. If, on the other hand, *Edge* e is undirected, the *inNeighbors()* of v with respect to e are defined to be all $\{w : (w \in e) \wedge (w \neq v)\}$. The *outNeighbors()* of v with respect to e are defined similarly, except that in the directed case, the neighboring vertices are those found *after* v in e. Like *Vertex* objects, *Edge* objects are never constructed explicitly by the programmer. See the *Graph* methods for adding vertices. **2. Operations**

~*Edge*()

The destructor for the vertex *Collection* object is called.

Bool hasVertex(Vertex* v)

Returns TRUE is v is a member of the vertex *Collection*.

Vertex* sourceVertex()

Returns a pointer to the first *Vertex* object in the vertex *Collection*. Note that this implies that it returns the smallest *Vertex* object in lexicographic order if the *Collection* is a *Set*. The time complexity depends on the *Container* used to store the elements.

Vertex* sinkVertex()

Returns a pointer to the *last Vertex* object in the vertex *Collection*. Note that this implies that it returns the largest *Vertex* object in lexicographic order if the *Collection* is a *Set*. The time complexity depends on the *Container* used to store the elements.

Sequence<Vertex> otherVertices(Vertex* v)

Return a *Sequence* object containing all vertices in the edge except v (in the same order in which they appear in the edge). $O(s)$, where s is the cardinality of the largest edge.

void replace(Vertex *old_v, Vertex *new_v)

This method is used by the *MHyperGraph* methods to manipulate subgraphs. It should never be called by the programmer. NEEDS TO BE MADE PRIVATE, BUT THE PROBLEM IS THAT THE PARSER NEEDS ACCESS.

Collection<Vertex*>* vertices() const

Return a pointer to the *Collection* of vertices. $O(1)$

DataType type() const

Return either EDGEUND or EDGEDIR depending on the type of *Collection* used to store vertices. (see general.h) It shouldn't normally be necessary to call this.

Bool operator==(const Vertex&) const
Bool operator<=(const Vertex&) const
Bool operator< (const Vertex&) const
Bool operator>=(const Vertex&) const
Bool operator> (const Vertex&) const
Bool operator!=(const Vertex&) const

Compare the current edge with another based on lexicographic ordering of edge names.

void saveToFile(ofstream& fout, int indent)

This method outputs the infomation associated with the current edge, including its attributes. It is not generally called by the programmer; *MHyperGraph::saveToFile()* calls it while writing out a graph.

friend ostream& operator<< (ostream& stream, const Edge& v)

Output the *Collection* containing the vertices to a stream.

2.7 *Graph*

The *Graph* class is an abstract class from which all graph types in LINK are derived. Like the *Collection* class the *Graph* class provides declarations for the core methods of the associated hierarchy of objects (this graph hierarchy is described in detail in Section 3). Unlike class *Collection*, however, the *Graph* class also provides several method definitions. It can do this since

```
#include<iostream.h>
#include<LINK/basic/Array.h>
#include<LINK/basic/Set.h>
#include<LINK/graph/BinGraph.h>


main()
{
        BinGraph g;
        MSet<Vertex*> s;
        Set<Vertex*>  s2;
        Vertex *v1, *v2;

        v1 = g.addVertex("v1");
        v2 = g.addVertex("v2");
        s.insert(v2); s.insert(v1); s.insert(v1);
        s2 = s;
        g.addEdge(s);
        g.addEdge(s2);
        cout << g << endl;
}
```

```
{[v1 v2]  {{v1 v2}}}
```

Figure 2.3: “Mixed” Binary Graph

the graph representation objects (see Page 163) are available in the class (The *Collection* implementation is not available in class *Collection*; it is defined in *MSetBase*).

2.7.1 Subgraphs

Before describing the *Graph* methods, it is necessary to give a description of the *subgraph* mechanism used in LINK. In applications which manipulate extremely large graphs, it is often a practical necessity to present the graph as a small, “coarse” graph in which certain areas of interest to the end user can be expanded to reveal finer detail. LINK's subgraph mechanism allows

```
#include<iostream.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/basic/Set.h>
#include<LINK/basic/Sequence.h>

main()
{
        DBinGraph g1;

        g1.addVertices(4);
        Vertex** vertices = g1.vertexStart();
        int num_vertices = g1.order();

        for (int i=1; i<num_vertices; i++) {
                Sequence<Vertex*> s;
                s.append(vertices[i]);
                s.append(vertices[i-1]);
                g1.addEdge(s);
        }
        cout << g1 << endl;
}
```

```
{[0 1 2 3]  {<1 0> <2 1> <3 2>}}
```

Figure 2.4: Directed Binary Graph

```
#include<iostream.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/basic/Sequence.h>


main()
{
        MDBinGraph mg;
        DBinGraph g;
        Sequence<Vertex*> s;
        Vertex *v1, *v2;

        v1 = mg.addVertex("v1");
        v2 = mg.addVertex("v2");
        s.append(v1); s.append(v2);
        mg.addEdge(s); mg.addEdge(s);
        cout << mg << endl;

        s.clear();

        v1 = g.addVertex("v1");
        v2 = g.addVertex("v2");
        s.append(v1); s.append(v2);
        g.addEdge(s); g.addEdge(s);
        cout << g << endl;
}
```

```
{[v1 v2]  {<v1 v2> <v1 v2>}}
{[v1 v2]  {<v1 v2>}}
```

Figure 2.5: Directed Binary Multigraph

```
#include<iostream.h>
#include<LINK/graph/UBinGraph.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/basic/Sequence.h>


main()
{
        UBinGraph ug;
        DBinGraph dg;
        Sequence<Vertex*> sq;
        Set<Vertex*>       s;
        Vertex *v1, *v2;

        v1 = ug.addVertex("v1");
        v2 = ug.addVertex("v2");
        s.append(v2); s.append(v1);   // must obey sorted order
        ug.addEdge(s);
        cout << ug << endl;

        v1 = dg.addVertex("v1");
        v2 = dg.addVertex("v2");
        sq.append(v2); sq.append(v1);
        dg.addEdge(sq);
        cout << dg << endl;
}
```

```
{[v1 v2]  {{v1 v2}}}
{[v1 v2]  {<v2 v1>}}
```

Figure 2.6: Undirected Binary Multigraph

the user or programmer to identify the vertices of any *induced subgraph* of G [2] and replaced it with a single vertex, called a *supervertex.* This supervertex can later be expanded to reveal the hidden detail. This subgraph contraction in LINK is hierarchical, i.e., the programmer or user can create arbitrary numbers of subgraph layers (subgraphs contracted within subgraphs). An induced subgraph of a graph $G = (V, E)$ is contracted by selecting a subset of vertices V' and calling the *addSubgraph(...)* method defined below. When contracted, the vertices of V' and the edges of E' are (apparently to the user or programmer) removed from the graph and replaced by a supervertex, which behaves as a normal vertex when processed by a graph algorithm. However, the subgraph information is maintained as an attribute of the supervertex. This attribute is itself an object from the *Graph* hierarchy of the same type as its parent graph. An important restriction to keep in mind when creating subgraphs is the following:

> Every vertex which is put into a subgraph must have the same parent.

In other words, two vertices which reside on different levels of the implicit subgraph hierarchy cannot be placed together into a new subgraph. One might ask how this situation could ever arise in the first place, since the user would not be able to see vertices at different levels simultaneously. It *is* possible for this to happen, however. Once created, subgraphs may be "opened" without being "dissolved." The user may want to view the expanded graph momentarily, then hide the subgraph again. The methods implementing this functionality are described in Section 2.7.3 below.

2.7.2 Input Graph Language

It is possible to store and retrieve LINK graphs from disk along with certain associated attributes using the *Graph::saveToFile(...)* method and the *parseGraph(...)* function. Although the input graph language allows for the saving of attributes, it reduces to the format produced by the *Graph::display(...)* method if no attributes need to be saved. The system currently supports the saving and loading of *int, float, double,* and *String (char*)* attributes. Future modifications to the parser may allow the storage of general type information, allowing the programmer to add support a specific complex or pointer typed attributes. In the meantime, programmers who want to group

[2] An induced subgraph of a graph $G = (V, E)$ is another graph $G' = (V', E')$, where $V' \subseteq V$ and E' consists of those edges in E whose vertex sets are entirely contained in V'.

all attribute information into a single class attribute to save calls to *setAttribute(...)* and *getAttribute(...)* are encouraged to write transformation routines which extract individual attribute fields and make them independent primitive attributes prior to storage on disk (and vice versa). Stored graphs are loaded into LINK using a parser which uses the grammar listed in Appendix A. Several examples of graph input files are also included in Appendix A.3. As stated previously, attributes of *GraphObjects* can be stored in the graph input file. The parser will recognize and build attributes of the following types, which conform to the C language standard with stated exceptions:

- *int*: Signed integers, with leading 0's allowed.
- *double*: Double precision floating point constants, expressed exactly as in the C standard.
- *float*: Single precision floating point constants, expressed as in the C standard except that a trailing *f* or *F* to designate the constant as `float` and not the default `double` representation.
- *String*: Valid C string constants (enclosed in double quotes).

The graph input file format itself begins with an optional preamble which identifies the graph type and declares *all* attributes which should be associated with the graph and its *Vertex* and *Edge* objects. If the preamble is missing, the graph type is assumed to be the most specific type matching the edge set. For example, *DBinGraph* (directed binary graph) is more specific than *DHyperGraph* (directed hypergraph) or *BinGraph* (binary graph where each edge may be either directed or undirected). A *DBinGraph is a DHyperGraph* and a *BinGraph* at the same time. See Section 3 for a discussion. The parenthesized attribute declarations list consists of 0 or more attribute declarations of the following format (the names used here are not the actual names of nonterminals in the input language grammar):

<attr_name> (<attr_value>)

The <attr_name> is a C language identifier (not a string constant). The <attr_value> may be any of the constants listed above (int, float, double, string constant), unless the given <attr_name> is one of the reserved words listed below, in which case the type must conform to the following table:

reserved attribute name	required type of constant
graph_name	<string constant>
name	<string constant>
location	(<numeric constant>,<numeric constant>)
vcolor	<int constant>
ecolor	<int constant>
direction	<int constant>
weight	<int constant>
open	<int constant>

These attributes are reserved since they are fairly standard and are assumed to be the stated type by the underlying system. New reserved attributes may be added by adding reserved words to `/src/interface/scanner.l`. The parser need not be changed unless the new reserved attribute must be of a type which is not currently supported. If the <attr_name> is not reserved, then the type of the <attr_value> determines the type of *Attribute* object which will be created and associated with the graph. After the preamble, there is a square-bracket enclosed list of vertex names with optional associated attributes. Each vertex specification consists of an identifier, integer constant, or string constant naming the vertex and a square bracket enclosed list of attributes of the same format as that discussed above. The only difference is that *it is not permitted to declare new attributes outside of the preamble.* Any unrecognized attribute name outside of the preamble will be flagged as an undeclared attribute and the parse will be aborted. The final section of the graph input file identifies the *Edges*. Undirected edges are specified with curly-brace enclosed lists of identifiers, integer constants, or string constants identifying vertices. There is no delimiter other than white space separating the vertex names in this list. *If any edge contains a vertex name which did not appear in the* Vertices *section, that vertex is reported to be undeclared and the parse fails.* Support for further attribute types can be obtained by studying and specializing the *AttributeElementOps¡T¿* class in Attribute.h (for correct output) and the graph language grammar listed in Appendix A, which is automatically generated from parser.y in the `/src/interface` directory. In order to allow arbitrary structure or pointer types to be stored and reloaded successfully, the graph input language would have to be augmented with more type recognizing power. Recall from Section 2.7.1 that a *Subgraph* is actually maintained as an *Attribute* of a special vertex. The parser is designed so that it is possible to save and reload graphs with layers of subgraphs. However, this mechanism is not complete and bugs remain. An example of the process of saving a LINK graph is depicted in Figure 2.7.

Now give an example with different types

2.7.3 *Graph* Methods

The *Graph* class is abstract, so there are no objects of class *Graph* to create. The methods used by objects of the derived graph hierarchy are listed below. Those ending in `= 0` lack definitions, which are generally provided by class *MHyperGraph* (see Section 3). Graph* newGraph()raph* g) = 0;

Dynamically allocate a new empty graph of the same type as g.

Graph* copyGraph(Graph* g) = 0;

Dynamically allocate a new empty graph of the same type as g, and initialize it with g's contents. This is useful when the only access to g is through a *Graph* pointer. If the actual graph object is available, copy constructors can be used.

Graph* newEmpty() const;

Return a dynamically allocated graph of the same type.

ASymMatrix<int> adjacencyMatrix();

Return an adjacency matrix representation of the graph. If the graph is a hypergraph, it will not be reconstructable from the matrix, but the adjacency information will still be correct.

```
#include<fstream.h>
#include<LINK/graph/UBinGraph.h>
#include<LINK/graph/Attribute.h>

void main()
{
        UBinGraph g1;
        if (newAttribute((Graph*)&g1, "test_attr", 1.0) != LINK_OK)
                error("Attr.cpp: error creating test_attr");
        g1.addVertices(2);
        Vertex** vertices = g1.vertexStart();
        setAttribute((GraphObject*)vertices[1], "test_attr", 8.0);
        ofstream fout;
        fout.open("g1.g", ios::out);
        g1.saveToFile(&fout);
        ifstream fin;
        fin.open("g1.g", ios::in);
        char ch;
        while (fin.get(ch))   cout.put(ch);
}
```

```
 { Graph UBinGraph
 (
  graph_name("Graph0")
  name("")
  direction(0)
  width(1)
  size(10)
  weight(1.00000)
  x(0.500000)
  y(0.500000)
  color("black")
  label("")
  mark(0)
  type(0)
  starttime(0)
  finishtime(0)
  back(0)
  low(0)
  partition(0)
  distance(0.00000)
  pred(0)
  forefather(0)
  open(1)
  test_attr(1.00000)
 )
 Vertices [
   0 ( name("0") )
   1 ( name("1") test_attr(8.00000) )
 ]
 Edges {
 }
 }
```

Figure 2.7: Saving a LINK Graph

```
#include<iostream.h>
#include<fstream.h>
#include<LINK/graph/UBinGraph.h>

Graph  *LoadGraph(String);      // use parser of input graph lang.

void main()
{
        ifstream fin;
        fin.open("Progg0.g", ios::in);
        char ch;
        while (fin.get(ch))   cout.put(ch);
        fin.close();
        cout << "----------------------------" << endl;
        Graph *gp= LoadGraph("Progg0.g");
        cout << *gp << endl;
}
```

```
 { Graph UBinGraph
 (
  graph_name("Graph1")
  name("")
  direction(0)
  weight(1)
  color(1)
  color(4)
  open(1)
  test_attr(1)
 )
 Vertices [
   v1 ( name("v1") test_attr(8) )
   v2 ( name("v2") )
   v3 ( name("v3") )
 ]
 Edges {
{ v1 v2 } ( name("e0") )
 }
 }
----------------------------
Parse failed: the returned graph is empty
```

Figure 2.8: Loading a LINK Graph from Disk

```
#include<iostream.h>
#include<LINK/graph/UBinGraph.h>

main()
{
	UBinGraph g;
	Set<Vertex*>	s;
	Vertex *v1, *v2;

	v1 = g.addVertex("v1");
	v2 = g.addVertex("v2");
	s.append( v2); s.append( v1);   // must obey sorted order
	g.addEdge(s);
	cout << g << endl;

	UBinGraph g2(g);	// graph attributes other than "name" are
				// not copied.
	g2.addVertex("v3");
	cout << g2 << endl;
}
```

```
{[v1 v2]  {{v1 v2}}}
{[v1 v2 v3]  {{v1 v2}}}
```

Figure 2.9: Copying a Graph without its Attributes

Bool isomorphicQ(const Graph* *graph*);
Bool isomorphicQ(const Graph& *graph*);

Return TRUE if the current graph is isomorphic to *graph* . The nauty system is used for isomorphism testing, and the symbolic constant MAXN determines the maximum allowable graph size. As a default, MAXN is set to 30, since nauty performs major code optimizations if the graphs are smaller than the word size. MAXN should be reset if bigger graphs need to be compared.

Bool binaryQ();

Return TRUE if the every edge has size 2.

Bool simpleQ();

Return TRUE if the graph type is one of: M_MIXEDHYPERGRAPH, M_UNDHYPERGRAPH, M_DIRHYPERGRAPH, M_MIXEDBINARYGRAPH, M_UNDBINARYGRAPH, M_DIRBINARYGRAPH

Bool directedQ();

Return TRUE if the graph is directed (not undirected or mixed.

Vertex* addVertex(String vname)

Creates and inserts a new vertex.

Edge* addEdge(Sequence<Vertex*> vs) = 0;

Creates a new (directed) edge with a pointer to a copy of *vs* and inserts the edge.

Edge* addEdge(Collection<Vertex*>& vs) = 0;

Creates a new (undirected) edge with a pointer to a copy of *vs* and inserts the edge. Note that *vs* is copied into a *Set* object, so even though multisets of vertices may be passed in, no duplicate vertices will exist in the edge (the *Collection* reference argument allows sets using representations other than DEFAULT_SET_IMPL to be passed).

Edge* addEdge(Edge *e) = 0;

Used by the subgraph mechanism to move edges between parent graphs and subgraphs. Should never be called by the programmer. SHOULD BE PRIVATE, BUT ACCESS PROBLEMS WITH THE PARSER.

void removeVertex(Vertex *v)

Used by the subgraph mechanism to move vertices between parent graphs and subgraphs. Should never be called by the programmer. SHOULD BE PRIVATE, BUT ACCESS PROBLEMS WITH THE PARSER.

void removeEdge(Edge *v)

Used by the subgraph mechanism to move edges between parent graphs and subgraphs. Should never be called by the programmer. SHOULD BE PRIVATE, BUT ACCESS PROBLEMS WITH THE PARSER.

void deleteVertex(Vertex *v) = 0;

Remove v from the vertex set of each of its incident edges, then delete v and any newly-empty edges.

void deleteEdge(Edge *e) = 0;

Remove e from the incident edge sets of each of its vertices, then delete it.

void clear()

Remove all edges and vertices from the graph.

int rank(Vertex* passed_vertex)

Where does this vertex rank in the current order of vertices $(0 \ldots n-1)$? $O(1)$

int rank(Edge* passed_edge)

Where does this edge rank in the current order of edges $(0 \ldots m-1)$? $O(1)$

Vertex* vertex(int i)

Return a pointer to the ith vertex. $O(1)$.

Vertex* randomVertex()

Return a pointer to a randomly-selected vertex. $O(1)$.

Edge* edge(int i)

Return a pointer to the ith edge. $O(1)$.

friend ostream& operator<<(ostream& os, const Graph& g)

Output graph g to output stream os.

ostream& display(ostream& os)

Output graph g on stream *os*.

void display(int indent)

This is called by the saveToFile() method to accommodate subgraphs.

void saveToFile(ofstream* fout) = 0;
void saveToFile(ofstream* fout, int indent) = 0;

Save the graph (including all of its subgraphs) to a file in the format described in Section A.

Vertex* findVertexByName(String name)

Perform a binary search through the vertex array for a vertex with the given name. $O(l \log n)$, where l is the length of the attribute list.

Vertex* findVertexWithEdge(String name)

This method is used primary by the subgraph mechanism. IT IS BUGGED, AND IT, ALONG WITH A PORTION OF THE SUBGRAPH MECHANISM, REQUIRES ATTENTION.

GraphType type()

Return the graph type (enumerated in general.h). $O(1)$

GraphType name()

Fetch and return the graph's *name* attribute. $O(l)$, where l is the length of the attribute list.

Bool adjacentQ(Vertex* v1, Vertex* v2)

Return TRUE if the two vertices $v1$ and $v2$ are adjacent. This is the case if there is an edge which contains both $v1$ and $v2$.

Vertex** vertexStart()

Return a pointer to the first *Vertex* pointer in the graph's vertex array. Iteration through the vertices is almost always begun this way (and carried out by a for loop). $O(1)$

const SortedArray<Vertex*>& vertices()

Return a reference to the graphs array of vertices $O(1)$. Note: in order for this operation to be efficient, the receiving lvalue must be a *reference* to a *SortedArray*. Otherwise the copy constructor will be invoked and the operation will take time $= O(n)$.

const MSet<Edge*>& edges()

Return the *MSet* containing the graph's edges. Note that receiving the return value with object is still efficient to due reference counting in the *Collection* hierarchy. $O(1)$

MSet<Edge*> incidentEdges(Vertex* v) const

Return the edges incident on *passed_vertex*. $O(1)$

MSet<Edge*> inIncidentEdges(Vertex* v) const

Return the edges in which v is a "sink." For directed binary graphs, this is a straightforward operation explained simply as "all edges pointing to v. For all hypergraphs and undirected graphs, though, the more abstract explanation is "all edges in which v is not listed first." $O(ds)$, where d is the maximum degree of the graph and s is the maximum size of an edge.

MSet<Edge*> outIncidentEdges(Vertex* v) const

Return the edges in which v is a "source." For directed binary graphs, this is a straightforward operation explained simply as "all edges pointing away from v. For all hypergraphs and undirected graphs, though, the more abstract explanation is "all edges in which v *is* listed first." $O(ds)$, where d is the maximum degree of the graph and s is the maximum size of an edge.

int order()

Return number of vertices in graph. $O(1)$

int size()

Return number of edges in graph. Time complexity depends on the *Container* used to implement *MSet*. This is specified by DEFAULT_SET_IMPL in MSet.h. With the current *SortedList* default, the complexity is $O(n)$ (no *last* pointer with STk interface)

int degree(Vertex* passed_vertex)

Return the degree of vertex in the graph.

int inDegree()ertex* passed_vertex)

Return the indegree of passed_vertex (see inIncidentEdges()).

int outDegree(Vertex* passed_vertex)

Return the outdegree of passed_vertex (see outIncidentEdges()).

int maxDegree() const

Return the maximum degree of any vertex in the graph. $O(n)$

int minDegree() const

Return the minimum degree of any vertex in the graph. $O(n)$

Sequence<Edge*> pathEdges(Sequence<Vertex*>)

Given a sequence of vertices, return a sequence of binary edges that connect the vertices if one exists, otherwise return an empty sequence. $O(n)$

Sequence<Vertex*> pathVertices(Sequence<Edge*>)

Given a sequence of binary edges, return the sequence of vertices that connect the vertices if one exists, otherwise return an empty sequence. $O(n)$

Sequence<Edge*> treePath()ertex* descendent, Vertex* ancestor)

Assuming that the pred attribute has been set by a search or shortest paths algorithm, retrive a path of edges from the descendent to the ancestor via predecessor pointers. If the vertices do not lie on the same branch of the depth-first or shortest paths tree, return an empty sequence. $O(n)$

Vertex* addSubgraph(Graph *owner, List<Vertex*>* l) = 0;

Given the passed list of vertices l, this method creates a graph object g of the same type as the current graph and moves the vertices of l and all induced edges into g. A new "supervertex," which contains a pointer to g as an attribute, is then created and inserted into the current graph, then a pointer to it is returned. The supervertex is adjacent to a vertex v in the current graph if and only if there exists $w \in V(g)$ such that w was adjacent to v in the original graph. Supervertices are treated as a normal vertices and can become members of additional subgraphs.

Set<Vertex*> dissolveSubgraph(Vertex *sv) = 0;

If sv is a supervertex, then it hides a subgraph. This method extracts the vertices and edges from the subgraph and adds them back to the current graph such that the latter is restored to the same adjacency structure it had before the subgraph was created. The return value is the set of subgraph vertices.

2.7.4 Adjacency Matrices

AsymMatrix<int> AdjacencyMatrix(const Graph *);

Given a graph, return its adjacency matrix. The class AsymMatrix<T> is described with the Matrix<T> class.

UBinGraph AdjacencyMatrix2Graph(const Matrix<int>&);

Given an adjacency matrix, return a binary graph object realizing the same adjacency relationship. Note that the input matrix must be symmetric.

DBinGraph AdjacencyMatrix2Digraph(const Matrix<int>&);

Given an adjacency matrix, return a directed binary graph object realizing the same adjacency relationship.

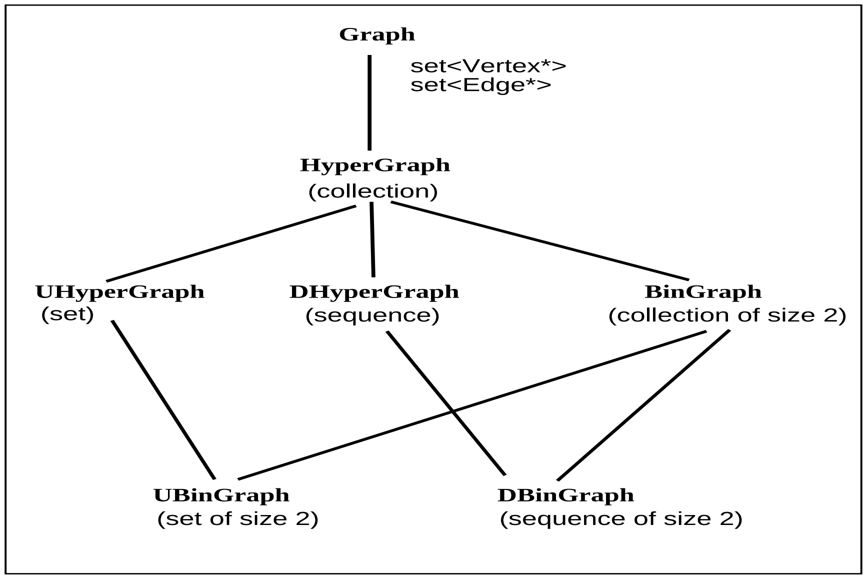


Figure 3.10: The *Graph*Hierarchy

AsymMatrix<int> IncidenceMatrix(const Graph *);

Given a graph, return its incidences matrix..

UBinGraph IncidenceMatrix2Graph(const Matrix<int>&);

Given an incidence matrix, return a binary graph object realizing the same adjacency relationship.

AsymMatrix<int> LaplacianMatrix(Matrix<int>& m);

Returns the product of matrix m and its transpose.

3 The Graph Hierarchy

4 Graph Operations

The unary and binary graph operations are listed below. For a discussion of the meaning of each operation, see [3] (for example). Note that each operation returns *MHyperGraph*. This allows the programmer maximum flexibility with respect to graph type. The *Graph*class constructors make it easy to convert the result into a more specific form. For example, suppose the programmer knows that the result should be treated as an undirected, binary graph. The following line of code produces a copy of the g's complement which is of the appropriate type:

UBinGraph ug = g-¿complement();

MBinGraph complement();

Return the complement graph. The result is a mixed binary graph containing only undirected edges.

MBinGraph intersection(const Graph& g);

Return the graph intersection of the current graph and g. Edges retain their directions, so the resulting graph may have undirected edges, directed edges, or both.

MBinGraph lexicographicProduct(const Graph& g);

Return the lexicographic product (an undirected graph) not implemented yet.

MBinGraph lineGraph();

Return the line graph. Edges of the result are always undirected.

MBinGraph product(const Graph& g);

Return the graph product of the current graph and g. The result is an undirected graph with x and y coordinates assigned to each vertex in such a way that the copies of the current graph is arranged in a circle, and each copy is itself arranged in a smaller circle.

MBinGraph sum(const Graph& g);

Return the graph sum of the current graph and g. Edges retain their directions, so the resulting graph may have undirected edges, directed edges, or both.

MBinGraph operator+(const Graph& g);

Return the result of calling sum(g).

MBinGraph operator*(const Graph& g);

Return the result of calling product(g).

5 Algorithms

2 Algorithms

2.1 Algorithm Specifications

The starred algorithms are finished, and the other algorithms are ranked in order of urgency. Hopefully these rankings will not be necessary after July 31st, 1996.

```
#include<LINK/basic/Sequence.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/graph/Attribute.h>
#include<LINK/algorithm/Algorithms.h>

main()
{
        DBinGraph g;
        Sequence<Vertex*>  s, postorder;
        Vertex *parent, *child1, *child2;

        child1 = g.addVertex("child1");
        parent = g.addVertex("parent");
        child2 = g.addVertex("child2");
        s.insert(parent); s.append(child1);
        g.addEdge(s);
        s.clear();
        s.insert(parent); s.append(child2);
        g.addEdge(s);

        postorder = DepthFirstSearch(&g);
        cout << postorder << endl;
}
```

```
<child1 child2 parent>
```

Figure 2.2: Depth-First Search

Priority	Algorithm/Property	input	output	attr's & algs used
2	Biconnected Components	Undir. G	Set<Set<Vertex*> >	DFS - back DFS - low DFS - discovery_time
5	Bipartite Matching	Undir. G	Set<Edge*>	weight Ford-Fulkerson - flow
*	Breadth-First Search	G,v	Sequence<Vertex*>	color distance ???
*	Depth-First Search	G	Sequence<Vertex*>	color starttime finishtime back low pred
3	Dijkstra's Single Source Shortest Paths	G, v	Set<double>	weight distance pred
4	Ford-Fulkerson Maximum Flow	Network s, t	double	flow capacity partition
7*	Floyd-Warshall All Pairs Shortest Paths	G	Matrix<double>	weight distance pred
*	Kruskal's MST	G	Set<Edge*>	weight mark parent pred
6	Prim's MST	G	Set<Edge*>	weight key pred
1	Strongly-Connected Components	Digraph	Set<Set<Vertex*> >	DFS - mark DFS - finishtime
*	TopSort	DAG	Sequence<Vertex*>	DFS - finishtime

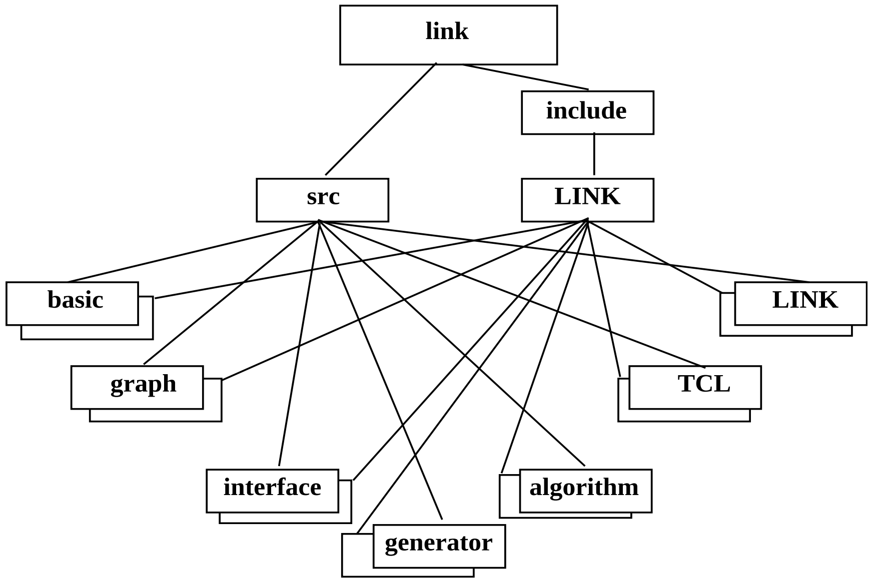


Figure 1.1: The LINK Directory Structure

Part V
Internals

1 The Directory Structure

Part VI
Acknowledgements

The original implementation of LINK was directed by *Patricia K. Fasel* of Los Alamos National Laboratories, and much of the early design and coding is due to her. Jonathan Berry designed the Collection hierarchy and co-designed the graph hierarcy as a graduate research assistant at Los Alamos National Laboratory, then took over as project coordinator in November, 1994, made fundamental changes to all aspects of the system, and designed the STk interface while visiting DIMACS. Many other people have contributed; they are listed below in alphabetical order along with their contributions and affiliations during their participation:

- *Chris Burrows* (Trenton State College) Implemented several algorithms and designed animations as part of a Research Experience for Undergraduates appointment at DIMACS.

- *John Chen* (SUNY Stony Brook) Responsible, with Yin-yi Juan, for the TCL handler generator which gives LINK its TCL command language interface.

- *Michael Dineen* (Los Alamos National Laboratory) During the course of his use of LINK in applications, suggested and implemented several improvements to the *Array* class and implemented the spring layout algorithm.

- *Tong (Sue) Gao* (SUNY Stony Brook) Revised the *Container* classes.

- *Elizabeth Johnson* (Indiana University and Los Alamos National Laboratory) Designed the inital TCL interface, predating the STk interface. Has used LINK in several applications and has made many improvements in the form of bug fixes and improving and adding algorithms.

- *Yin-yi Juan* (SUNY Stony Brook) Responsible, with John Chen, for the TCL handler generator which gives LINK its TCL command language interface.

- *Vassili Leonov* (SUNY Stony Brook) Developed the ttool regression test environment and contributed with systems issues.

- *Darren Lim* (Rensselaer Polytechnic Institute) Explored the addition of mutlisets to the *Collection* hierarchy. Also extracted a symbolic set object from Mark Goldberg's *SetPlayer* system for the purpose of inclusion in LINK.

- *John MacCuish* (Los Alamos National Laboratory) Responsible for the library of graph generators and several graph algorithms, including network flow and several layout algorithms.

- *Michael Murphy* (SUNY Stony Brook and Los Alamos National Laboratory) Responsible for the implementation of the priority queue and matrix objects and several algorithms.

- *Stephen Schmitz* (SUNY Stony Brook) Implemented initial combinatorial objects, including Permutation and subset. Responsible for preliminary work on the revised TCL interface.

- *Gregg Steubeng* (Rensselaer Polytechnic Institute) Worked on a prototype implementation of a permutation ranking algorithm with Mark Goldberg.
- *Xuesong Ran* (SUNY Stony Brook) Revised the priority queue objects.
- *David Wagner* (SUNY Stony Brook) Revised the *Collection* classes and helped with testing the library of graph algorithms.

Additional work on the LINK project was done by Hector Beltran, Christian Joita, Jim Klosowski, Galina Leonova, and Chris Smith of SUNY Stony Brook, and Dan Liles of Los Alamos National Laboratory.

A Graph File Format

A.1 Lexical Tokens

Token ID	Token String
INT_CONST	([+\-])?[0-9][0-9]*
DBL_CONST	([+\-])?[0-9]+(\.[0-9]*)([Ee][+\-]?[0-9]+)?([lL])?
DBL_CONST	([+\-])?[0-9]*(\.[0-9]+)([Ee][+\-]?[0-9]+)?([lL])?
DBL_CONST	([+\-])?[0-9]+(\.[0-9]*)?([Ee][+\-]?[0-9]+)([lL])?
FLT_CONST	([+\-])?[0-9]+(\.[0-9]*)([Ee][+\-]?[0-9]+)?[fF]
FLT_CONST	([+\-])?[0-9]*(\.[0-9]+)([Ee][+\-]?[0-9]+)?[fF]
FLT_CONST	([+\-])?[0-9]+(\.[0-9]*)?([Ee][+\-]?[0-9]+)[fF]
LBRACE	"{"
RBRACE	"}"
LBRACKET	"["
RBRACKET	"]"
LDIRECT	"<"
RDIRECT	">"
COMMA	","
ABSTRACTGRAPH	Graph
MHYPERGRAPH	MHyperGraph
HYPERGRAPH	HyperGraph
MUHYPERGRAPH	MUHyperGraph
UHYPERGRAPH	UHyperGraph
MDHYPERGRAPH	MDHyperGraph
DHYPERGRAPH	DHyperGraph
MBINGRAPH	MBinGraph
BINGRAPH	BinGraph
MUBINGRAPH	MUBinGraph
UBINGRAPH	UBinGraph
MDBINGRAPH	MDBinGraph
DBINGRAPH	DBinGraph
VERTICES	Vertices
EDGES	Edges
LOCATION	location
SUBGRAPH	x_subgraph
INT_ATTR	back
STR_ATTR	color
INT_ATTR	direction
DBL_ATTR	distance
INT_ATTR	finish_time
STR_ATTR	graph_name
STR_ATTR	label
INT_ATTR	low
INT_ATTR	mark
STR_ATTR	name
INT_ATTR	open
VPTR_ATTR	pred
VPTR_ATTR	forefather
INT_ATTR	start_time
INT_ATTR	size
DBL_ATTR	weight
INT_ATTR	width
DBL_ATTR	x
DBL_ATTR	y
IDENT	[a-zA-Z_][a-zA-Z0-9_]*
STR_CONST	\"([^"\n]|\\\"|\n)*\"

A.2 Language Grammar

graph_main	→	*graph_decl*
graph_decl	→	**LBRACE** *graph_desc graph_deflist* **RBRACE**
graph_decl	→	**LBRACE** *graph_deflist* **RBRACE**
graph_desc	→	*graph_type* **(** *graph_attrs* **)**
graph_desc	→	*graph_type* **(** **)**
graph_desc	→	*graph_type*
graph_desc	→	**(** *graph_attrs* **)**
graph_deflist	→	*vertices edges*
graph_type	→	**ABSTRACTGRAPH**
graph_type	→	**ABSTRACTGRAPH MHYPERGRAPH**
graph_type	→	**ABSTRACTGRAPH HYPERGRAPH**
graph_type	→	**ABSTRACTGRAPH MUHYPERGRAPH**
graph_type	→	**ABSTRACTGRAPH UHYPERGRAPH**
graph_type	→	**ABSTRACTGRAPH MDHYPERGRAPH**
graph_type	→	**ABSTRACTGRAPH DHYPERGRAPH**
graph_type	→	**ABSTRACTGRAPH MBINGRAPH**
graph_type	→	**ABSTRACTGRAPH BINGRAPH**
graph_type	→	**ABSTRACTGRAPH MUBINGRAPH**
graph_type	→	**ABSTRACTGRAPH UBINGRAPH**
graph_type	→	**ABSTRACTGRAPH MDBINGRAPH**
graph_type	→	**ABSTRACTGRAPH DBINGRAPH**
graph_attrs	→	*graph_attr*
graph_attrs	→	*graph_attrs graph_attr*
graph_attr	→	*rstr_g_attr*
graph_attr	→	*rint_g_attr*
graph_attr	→	*rdbl_g_attr*
graph_attr	→	*rvptr_g_attr*
graph_attr	→	*reptr_g_attr*
graph_attr	→	*int_g_attr*
graph_attr	→	*flt_g_attr*
graph_attr	→	*dbl_g_attr*
graph_attr	→	*str_g_attr*
graph_attr	→	**LOCATION (** *number* **COMMA** *number* **)**
rstr_g_attr	→	**STR_ATTR (STR_CONST)**
str_g_attr	→	**IDENT (STR_CONST)**
rint_g_attr	→	**INT_ATTR (INT_CONST)**
int_g_attr	→	**IDENT (INT_CONST)**
flt_g_attr	→	**IDENT (FLT_CONST)**

rvptr_g_attr → **VPTR_ATTR** (*obj_name*)
reptr_g_attr → **EPTR_ATTR** (*obj_name*)
rdbl_g_attr → **DBL_ATTR** (*number*)
dbl_g_attr → **IDENT** (**DBL_CONST**)
vertices → *vertex_hdr* *vertex_list* **RBRACKET**
vertices → *vertex_hdr* **RBRACKET**
vertex_hdr → *v_hdr*
vertex_hdr → **VERTICES** *v_hdr*
v_hdr → **LBRACKET**
vertex_list → *vertex*
vertex_list → *vertex_list* *vertex*
vertex → *vertex_name* (*vertex_attrs*)
vertex → *vertex_name* ( )
vertex → *vertex_name*
vertex_name → *obj_name*
obj_name → **STR_CONST**
obj_name → **IDENT**
obj_name → **INT_CONST**
vertex_attrs → *vertex_attr*
vertex_attrs → *vertex_attrs* *vertex_attr*
number → **INT_CONST**
number → **FLT_CONST**
number → **DBL_CONST**
vertex_attr → *rstr_v_attr*
vertex_attr → *rint_v_attr*
vertex_attr → *rdbl_v_attr*
vertex_attr → *rvptr_v_attr*
vertex_attr → *reptr_v_attr*
vertex_attr → *int_v_attr*
vertex_attr → *flt_v_attr*
vertex_attr → *dbl_v_attr*
vertex_attr → *str_v_attr*
vertex_attr → **LOCATION** (*number* **COMMA** *number*)
vertex_attr → *subgrph_attr* (*graph_decl*)
subgrph_attr → **SUBGRAPH**
rint_v_attr → **INT_ATTR** (**INT_CONST**)
int_v_attr → **IDENT** (**INT_CONST**)
flt_v_attr → **IDENT** (**FLT_CONST**)
rdbl_v_attr → **DBL_ATTR** (*number*)
rvptr_v_attr → **VPTR_ATTR** (*obj_name*)

reptr_v_attr	→	**EPTR_ATTR** (*obj_name*)
dbl_v_attr	→	**IDENT** (**DBL_CONST**)
rstr_v_attr	→	**STR_ATTR** (**STR_CONST**)
str_v_attr	→	**IDENT** (**STR_CONST**)
edges	→	*edge_hdr edge_list* **RBRACE**
edges	→	*edge_hdr* **RBRACE**
edge_hdr	→	*e_hdr*
edge_hdr	→	**EDGES** *e_hdr*
e_hdr	→	**LBRACE**
edge_list	→	*edge*
edge_list	→	*edge_list edge*
edge	→	*edge_name edge_def* (*edge_attrlist*)
edge	→	*edge_def* (*edge_attrlist*)
edge	→	*edge_name edge_def* ( )
edge	→	*edge_def* ( )
edge	→	*edge_name edge_def*
edge	→	*edge_def*
edge_name	→	*obj_name*
edge_def	→	*dir_edge*
edge_def	→	*und_edge*
edge_attrlist	→	*edge_attr*
edge_attrlist	→	*edge_attrlist edge_attr*
edge_attr	→	*rstr_e_attr*
edge_attr	→	*rint_e_attr*
edge_attr	→	*rdbl_e_attr*
edge_attr	→	*rvptr_e_attr*
edge_attr	→	*reptr_e_attr*
edge_attr	→	*str_e_attr*
edge_attr	→	*int_e_attr*
edge_attr	→	*flt_e_attr*
edge_attr	→	*dbl_e_attr*
edge_attr	→	**LOCATION** (*number* **COMMA** *number*)
rint_e_attr	→	**INT_ATTR** (**INT_CONST**)
int_e_attr	→	**IDENT** (**INT_CONST**)
flt_e_attr	→	**IDENT** (**FLT_CONST**)
rdbl_e_attr	→	**DBL_ATTR** (*number*)
dbl_e_attr	→	**IDENT** (**DBL_CONST**)
rvptr_e_attr	→	**VPTR_ATTR** (*obj_name*)
reptr_e_attr	→	**EPTR_ATTR** (*obj_name*)
rstr_e_attr	→	**STR_ATTR** (**STR_CONST**)

str_e_attr	→	**IDENT** (**STR_CONST**)
und_edge	→	**LBRACE** *vertex_set* **RBRACE**
dir_edge	→	**LDIRECT** *vertex_seq* **RDIRECT**
vertex_set	→	*vertex_und*
vertex_set	→	*vertex_set vertex_und*
vertex_und	→	*obj_name*
vertex_und	→	*obj_name* **LBRACKET** *obj_name* **RBRACKET**
vertex_seq	→	*vertex_dir*
vertex_seq	→	*vertex_seq vertex_dir*
vertex_dir	→	*obj_name*
vertex_dir	→	*obj_name* **LBRACKET** *obj_name* **RBRACKET**

A.3 Language Grammar

The following are two different input files for the same graph: the Petersen Graph.

```
------------------------------------------------------------------------
 { [ 0 1 2 3 4 5 6 7 8 9 ]
   { { 0 1 } { 0 4 } { 0 5 } { 1 2 } { 1 6 } { 2 3 } { 2 7 } { 3 4 } { 3 8 }
     { 4 9 } { 5 7 } { 5 8 } { 6 8 } { 6 9 } { 7 9 }
   }
}
------------------------------------------------------------------------
```

```
{ Graph UBinGraph
 (
  graph_name("Graph1")
  name("")
  direction(0)
  ewidth(1)
  vsize(10)
  weight(1.00000)
  location(0.5, 0.5)
  vcolor("blue")
  ecolor("black")
  label("")
  mark(0)
  time(0)
  pred(0)
  open(1)
 )
Vertices [
   1 ( name("1") location(0.603333,0.876667) vcolor("blue") vsize(10) )
   2 ( name("2") location(0.296667,0.570000) vcolor("blue") vsize(10) )
   3 ( name("3") location(0.943333,0.576667) vcolor("blue") vsize(10) )
   4 ( name("4") location(0.506667,0.206667) vcolor("blue") vsize(10) )
   5 ( name("5") location(0.806667,0.206667) vcolor("blue") vsize(10) )
   6 ( name("6") location(0.593333,0.656667) vcolor("blue") vsize(10) )
   7 ( name("7") location(0.570000,0.363333) vcolor("blue") vsize(10) )
   8 ( name("8") location(0.740000,0.370000) vcolor("blue") vsize(10) )
   9 ( name("9") location(0.736667,0.536667) vcolor("blue") vsize(10) )
   10 ( name("10") location(0.526667,0.540000) vcolor("blue") vsize(10) )
 ]
 Edges {
{ 1 2 } ( name("e2") label("e2") ewidth(2) )
{ 1 3 } ( name("e3") label("e3") ewidth(2) )
{ 1 6 } ( name("e13") label("e13") ewidth(2) )
{ 2 4 } ( name("e6") label("e6") ewidth(2) )
{ 2 10 } ( name("e12") label("e12") ewidth(2) )
{ 3 5 } ( name("e4") label("e4") ewidth(2) )
{ 3 9 } ( name("e14") label("e14") ewidth(2) )
{ 4 5 } ( name("e5") label("e5") ewidth(2) )
{ 4 7 } ( name("e16") label("e16") ewidth(2) )
{ 5 8 } ( name("e15") label("e15") ewidth(2) )
{ 6 7 } ( name("e7") label("e7") ewidth(2) )
{ 6 8 } ( name("e8") label("e8") ewidth(2) )
{ 7 9 } ( name("e11") label("e11") ewidth(2) )
{ 8 10 } ( name("e9") label("e9") ewidth(2) )
{ 9 10 } ( name("e10") label("e10") ewidth(2) )
}}
```

References

[1] D. Berque, R. Cecchini, M. Goldberg, and R. Rivenburgh. The setplayer system for symbolic computation on power sets. *Journal of Symbolic Computation*, 14:645–662, 1992.

[2] J. Berry, N. Dean, P. Fasel, M. Goldberg, E. Johnson, J. MacCuish, G. Shannon, and S. Skiena. Link: A combinatorics and graph theory workbench for applications and research. Technical Report 95-15, Center for Discrete Mathematics and Theoretical Computer Science (see also: http://dimacs.rutgers.edu), Piscataway, NJ, 1995.

[3] J. Bondy and U. Murty. *Graph Theory with Applications.* North-Holland, 1976.

[4] N. Dean and G. E. Shannon, editors. *Computational Support for Discrete Mathematics.* American Mathematical Society, 1993. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 15.

[5] Charles M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[6] E. Gallesio. The stk reference manual. Technical Report RT 95-31a, I3S CNRS, Université de Nice - Sophia Antipolis, France, 1995.

[7] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[8] B. McKay. nauty user's guide (version 1.5). Technical Report TR-CS-90-02, Department of Computer Science, Australian National University, Australia, 1990.

[9] K. Mehlhorn and S. Nähger. Leda: A platform for combinatorial and geometric computing. *CACM*, 38(1):96–102, Jan 1995.

[10] M. Mevenkamp, N. Dean, and C. Monma. NETPAD user's guide and reference guide, 1990.

[11] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[12] G. Shannon, L. Meeden, and D. Friedman. SchemeGraphs: An object-oriented environment for manipulating graphs, 1990. Software and documentation.

[13] S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, 1990.

[14] R. Tamassia and I. G. Tollis, editors. *Proceedings of the DIMACS International Workshop on Graph Drawing*, 1994. Lecture Notes in Computer Science, 894.

Index