The **Swan** User's Manual

Version 1.1

Jun Yang   Clifford A. Shaffer   Lenwood S. Heath
Department of Computer Science
Virginia Tech
Blacksburg, Va. 24061

May 1995

Contents

1 Introduction

Swan is a data structure visualization system. Its main purpose is to allow the user to visualize the data structures used in a C/C++ program. **Swan** is specially designed to support visualization of programs implementing various graph algorithms. Throughout this manual, the **annotator** is the person who annotates a C/C++ program with **Swan**'s library of visualization functions. The **viewer** is the person who runs the annotated program using **Swan**'s Viewer Interface (**SVI**).

In **Swan**, **visualization** means a graphical representation of the data structure and the abstraction represented by the data structure. These are intended to help the viewer understand the algorithm implemented in the program.

To use **Swan**, a program must first be *annotated*, then compiled and linked with the **Swan** Annotation Interface Library (**SAIL**). The viewer can then run the program.

To annotate a program, the annotator should have a clear understanding of its data structure. Then different views (i.e. graphs) of the data structure can be constructed by calling **SAIL** functions. The annotator is not required to control the graphical display of these graphs, but he has full power to decide most graphical attributes of the graphs if he wants. **Swan** does not assume any responsibility to analyze or understand the specific data structure of the annotated program.

To run an annotated program, the viewer simply starts the executable file of the program and investigates the views rendered in the **Swan** display window. The viewer has the capability to modify not only the graphical attributes of the graphs, but also the logical structure of the graphs if this is allowed by the annotator. Thus, the visualization process in **Swan** can be considered as a two way communication process between the annotator and the viewer. The annotator builds different views for the viewer and receives the viewer's requests to modify the views. On the other hand, the viewer explores the views constructed by the annotator and sends requests to modify the graphs. The protocols for this communication process are actually the main components of **Swan**: the **Swan** Annotation Interface Library (**SAIL**) and the **Swan** Viewer's Interface (**SVI**).

SAIL is a library of functions which can be added in a C/C++ program by using any text editor. **SVI** is a window environment in which the viewer can see the graphs and modify their graphical attributes. He can also control the running process and modify the logical topology of those graphs if allowed by the annotator.

The details of design and implementation of **Swan** can be found in [8]. Several references on graph drawing algorithms are listed at the end of the manual.

Typographic Conventions. The following typographic conventions are observed in this manual:

Italic Font is used for formal parameter names, emphasis, and to introduce new terms.

`Teletype Font` is used for actual parameter names, file excerpts, file names, and function prototypes.

Bold Font is used for proper titles.

2 Swan Viewer Interface (SVI)

Windows **SVI** provides the viewer's interface with an annotated program. It contains a **main window** entitled "**Swan**". The main window has a control panel which is a set of buttons with different functionalities. It also has three child windows: the **display window**, the **I/O window** and the **location window**. The display window is the place for the graphs created in **Swan** to be displayed. The I/O window is used by the annotator and **Swan** system to display one-line messages and get input from the viewer. The coordinates of the cursor in the display window are shown in the location window (Figure 1).

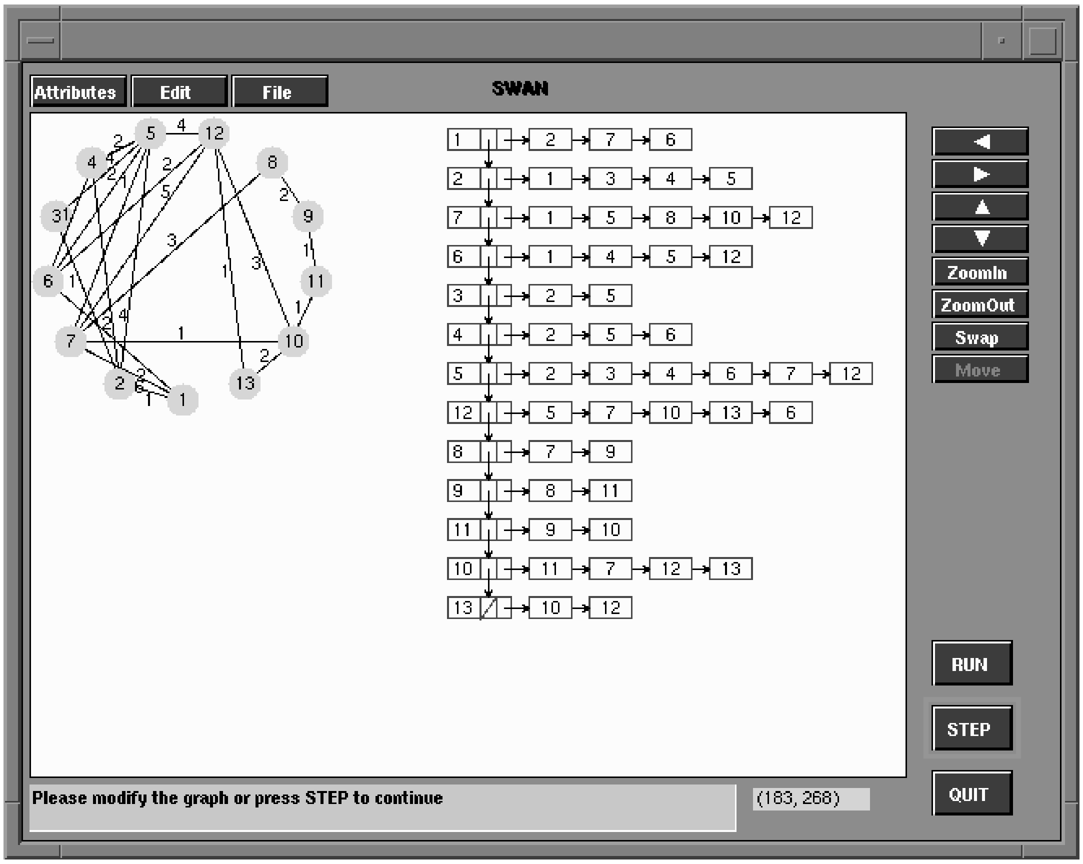


Figure 1: Two views of a graph created in an annotated minimum spanning tree algorithm. The display window is the big rectangular area in which the two views are shown. The I/O window is the long grey box at the bottom with a message displayed in it. The location window is the small box next to the I/O window which contains the coordinates of the cursor.

Picking The viewer can pick a node or an edge in **Swan** display window to get more information about it.

To pick a node, click on the node. A popup window will appear with the ID and label of the node displayed in it. The popup window has to be closed by clicking it again before any other action can be taken.

To pick an edge, click on the edge. A popup window will appear to display the ID's of the edge's two end nodes and its label. The popup window has to be closed by clicking it again before any other action can be taken.

Panning and Zooming There are eight buttons in the upper right hand corner of the main window, Six of them are used to pan and zoom the graphs in the display window. The first four buttons are used to move the viewed area in four directions indicated by the corresponding arrows. The next two buttons are used to zoom in or out.

The buttons **Swap** and **Move** are used to swap or move the nodes in a graph. These two functions are only allowed on some specific graph layouts, such as `KKNET` layout of a general undirected graph (Section 3.3.1). To swap physical positions of two nodes in one graph, the viewer picks two nodes in turn. Their positions will be swapped automatically by **Swan**. To move a node, the viewer picks a node and selects a destination position. **Swan** will move the node to that position.

Process Control The three buttons in the lower right hand corner of the main window are used to control the running process of the annotated program. Clicking button **RUN** will make the program run in continuous mode. Click button **STEP** will make the program to run in step mode. When the program is running in step mode, it will stop at any break point set by the annotator, waiting for the viewer to click button **STEP** to continue running in step mode or button **RUN** to run in continuous mode. When in continuous mode, the annotator's break points will be ignored. The annotator can disable these two buttons. A disabled button has no effect on the program if it is clicked.

QUIT can be clicked to leave **Swan**. The annotator cannot disable it.

Graphical Attributes Modification Every graph, node and edge created in **Swan** has a set of graphical attributes. For a graph, these attributes include default graphical attributes for its nodes and edges and its layout method. For a node, these attributes include type, color, size, and line thickness. For an edge, these attributes include color and line thickness.

To modify graphical attributes, the viewer can click the **Attributes** button on the top left corner of the **Swan** main window. A popup menu will appear. It has four items: **Graph Attributes**, **Node Attributes**, **Edge Attributes** and **Global Attributes**. The viewer can select any one of these items to modify its graphical attributes accordingly. For example, if the viewer selects **Graph Attributess**, **Swan** will display a message in the I/O window which asks the viewer to pick a graph from the graphs in the **Swan** display window. The viewer can pick a graph by clicking in the area it covers in **Swan** display window. A popup window which contains all the modifiable graphical attributes of the picked graph will appear. The buttons in the **Node Attributes** box are used to change the default attributes of nodes in the graph, including:

`Type -` change the shape of the nodes. It can be `Box` or `Circle`.

`Color -` change the color of the nodes.

`Filled` - draw the nodes in their framed shape or in filled areas.

`Thickness` - change the thickness of the lines.

`Width, Height` - control the sizes of the nodes.

The buttons on the right side of the window are used to change the default attributes of edges in the graph, including:

`Color` - change the color of the edges.

`Thickness` - change the thickness of the lines used to draw the edges.

`Length` - control the minimum length of the edges.

`Label` - determine whethe to show the edge label or not.

The buttons on the lower left corner are used to control the layout of the graph, including:

`Layout` - select different layout algorithm for the graph.

`Mode` - select layout mode between `AUTO` and `MANU`. If it is in `AUTO` mode, the position of the graph will be decided by **Swan**. If it is in `MANU` mode, the position of the graph can be adjusted manually by the viewer.

`Relayout` - turn on or off the automatic relayout switch. If it is `ON`, any modification of the physical attributes of the graph will cause **Swan** to redraw the the graph. Otherwise, the graph will only be redrawn until the annotator requests.

The viewer can change the attributes to the ones he prefers. Finally, the button **OK** in the popup window needs to be clicked to confirm all the modifications and the graph will be redrawn with the new set of attributes. Otherwise, if the button **Cancel** is clicked, all the modifications will be ignored and graph will be the same as before. Graphical attributes of nodes and edges can be modified in a similar way.

Graph Editing The logical structure of an annotated graph can be modified interactively by the viewer through insertion or deletion of nodes and edges. The annotator can enable or disable any of these functions. An editing function is effective only when it is enabled.

To edit a graph, click the button **Edit** on the upper left corner of the main window. A popup menu will appear. It has four items: **Insert Node**, **Delete Node**, **Insert Edge**, and **Delete Edge**. They correspond to the four graph editing functions mentioned above. Select one of these items to initiate the corresponding function.

Because the annotator has control of the logical structure of the graph, any modification on the logical structure eventually must be performed by the annotator. Therefore, the annotator actually determines the semantics of the editing functions. In most cases, the annotator will provide functions corresponding to the labels of those menu items, however, nothing stops the annotator from providing functions which may have nothing to do with insertion or deletion of nodes and edges. The viewer needs to be careful while editing to follow the instructions provided by the annotator in the **Swan** I/O window.

Graph Layout Saving and Restoring During execution of the annotated program, the viewer may want to save a particular graph layout. This is supported in **Swan**. Saving a graph layout means to save the physical layout of the graph and all the graphical attributes associated with it so that when the layout is restored, the graph will look exactly like when it was saved. However, the restored graph is an image which cannot be used directly as input data to the annotated program.

Click the button **File** to reach the menu items **Save** and **Load**. To save a graph layout, select **Save**, pick a graph, and specify a file name in the I/O window. To load a graph layout, select **Load**, and give the file name which contains the layout to be restored. **Swan** will let the viewer know whether the action is successful or any error occurs.

Error Log All the errors occurring during a **Swan** session are recorded in the file `error.log` for future reference. A **Swan** user may wish to examine this file or send a copy to the program annotator if they suspect a problem in the annotation.

3 Swan Annotation Interface Library (SAIL)

3.1 Introduction

The **Swan** Annotation Interface Library (**SAIL**) is a set of easy to use functions for annotating a program so that its significant data structure and execution process can be visualized. Given an appropriate description of the data structure used in a program, **Swan** is able to display it using graphical elements as specified by the annotator. Proper use of **SAIL** functions should provide a more intuitive understanding of the data structures and the manner which the data structures change in a program.

Graphs are used to represent the actual data structures in the program or the abstractions represented by those data structures. For example, consider a C program to find a minimum spanning tree in a graph G. The graph is stored in the program using an *adjacency list*. To annotate this program, two views can be constructed. A view of the adjacency list is a direct representation of the physical data structure used in the program. A view of an undirected graph represents the logical topology of the graph G (Figure 1).

The annotator should have a good understanding of the data structures in a program before designing views of the data structure. In **Swan**, a **view** is a **graph** which consists of a set of nodes and a set of edges. The semantics of nodes and edges in the graph are decided by the annotator, i.e. the annotator decides what structure these nodes and edges represent.

Every **Swan** graph has a logical topology and a physical representation (i.e. **layout**). The logical topology of the graph is determined by the nodes and edges in the graph. The annotator decides what nodes and edges should be in this graph and what their meanings are. Under certain circumstances, the viewer is allowed by the annotator to modify the logical structure of the graph.

The layout of a graph is a drawing of the graph on a 2-dimensional surface. Specifically, it is an assignment of Euclidean coordinates to the nodes and edges on the X-Y plane. A graph may have infinitely many different layouts. A layout with good readability can help the viewer's understanding of the graph. There are numerous graph drawing algorithms. Several algorithms are implemented in **Swan** to draw arrays, linked lists, binary trees, general rooted trees, and general undirected and directed graphs.

The annotator also has control of several buttons in the **Swan** main window. He can decide whether the viewer is allowed to modify the logical topology of the graph by enabling or disabling the items in the **Edit** menu. He can also enable/disable buttons **RUN** and **STEP** to allow or disallow the viewer's control of the running process.

3.2 Usage

Currently **SAIL** supports annotations of C or C++ programs on UNIX with the X Window System installed.

To annotate a C program, the header file `sail.h` must be included. Then **SAIL** function calls can be added to annotate the program. After the annotation is finished, compile the program using a C compiler (e.g. `gcc`) and link it with the C version of **SAIL** in `libsail.a` to generate the executable file.

To annotate a C++ program, compile the annotated program with a C++ compiler (e.g. `g++`) and link it with C++ version of **SAIL** `libsail++.a`.

For example, if `mst.c` is an annotated C program, the following command will generate the executable file assuming the **SAIL** library `libsail.a` and the header file `sail.h` are in your working directory:

```
gcc -o mst mst.c -L. -lsail -lXt -lX11 -lm -L/usr/local/lib -lg++
```

If `mst.c` is an annotated C++ program, the command is similar, assuming the **SAIL** library `libsail++.a` and the header file `sail.h` are in your working directory:

```
g++ -o mst mst.c -L. -lsail++ -lXt -lX11 -lm
```

Before you run `mst`, please make sure the **Swan** interface file `swan.inf` is in your working directory.

3.3 SAIL Basics

3.3.1 Basic Elements

SAIL provides a small set of elements to be used by the annotator to construct different views of a data structure. These elements not only have logical meanings, but also have graphical attributes since they can be displayed in the **Swan** display window. These elements include:

graph - a generalized graph whose definition can be found in any data structure textbook. It can be either *undirected* or *directed*. Every graph has a unique ID in **Swan**.

node - any node in a graph. Every node has an ID which must be unique within a graph. The same ID can be used for nodes in different graphs.

edge - any edge in a graph. An edge connects two nodes (e.g. node `s` and `t`) in a graph. If the graph is directed, the edge has a direction which is *from* node `s` *to* node `t`. If the graph is undirected, the order of the two nodes makes no difference.

LC - *Layout Component(LC)* is a mechanism used by the annotator to provide graphical layout hints to **Swan**. It does not carry any logical information.

Valid LC's are:

`ARRAYACROSS` - a horizontal array.

`ARRAYDOWN` - a vertical array.

`LISTACROSS` - a horizontal linked list.

`LISTDOWN` - a vertical linked list.

`CIRCLENET` - nodes will be evenly distributed on a circle and edges are straight lines forming chords of the circle.

`BINTREE` - The graph will be laid out as a binary tree.

`TREE` - The graph will be laid out as a rooted tree.

`KKNET` - The general undirected graph will be laid out using the Kamada and Kawai's algorithm in [5].

`HIERARCHY` - The general directed graph will be laid out hierarchically.

`MANUAL` - The positions of nodes in the LC must be specified by the annotator directly. Then edges will be drawn as straight lines connecting the nodes.

3.3.2 Basic Operations

Following are the operations that can be performed on the basic elements.

Graph

A graph is created by `sw_newgraph`. A unique ID should be provided by the annotator. If the annotator does not want to specify the ID, `NULLGRAPHID` can be used as the corresponding argument.

Then the function will return an ID generated automatically by the system. This ID can be used later to refer to this graph. The graph can be created as a *directed* graph or an *undirected* graph. The default display type of nodes in this graph must also be declared (e.g. `BOX` or `CIRCLE`).

A graph is deleted by `sw_deletegraph`. If the graph is displayed, it will be removed from the window. The ID of the graph will become invalid.

A graph can be displayed by `sw_displaygraph`. `sw_displayallgraphs` is used to display all the valid graphs in **Swan**.

A graph contains a set of nodes and edges. The default graphical attributes of nodes and edges in a graph can be set by the function `sw_setgraphattr`.

Layout Component(LC)

A graph in **Swan** consists of a set of LC's. Each LC has a set of nodes and edges. When the graph is displayed, the layout of nodes and edges will be determined by the type of the LC they belong to.

LC's in a graph are organized as a rooted tree. The graph has a *root* LC. Every other LC has a parent LC and a set of children LC. The parent-child relation between two LC's are established by inserting an edge which connects two nodes in these two LC's.

Currently, only `LISTDOWN` and `LISTACROSS` LC's can have children LC's of the type `LISTDOWN` or `LISTACROSS`. The restriction is one node in the parent LC can be connected with at most one child LC. LC's of other types *cannot* have children LC's.

Each graph has a *current* LC. All the insertions of nodes and edges in the graph will happen in this current LC. Therefore, at least one LC has to be created in each graph so that nodes and edges can be inserted. However, in most cases, only one LC will be created for a graph.

An LC is created by `sw_newlc`. It is deleted by `sw_deletelc`. The current LC can be changed by `sw_setcurlc`. The default graphical attributes of nodes and edges in an LC can be set by the function `sw_setlcattr`. These attributes override the default attributes of the graph.

Node

A node is inserted into a graph by calling function `sw_insertnode`. The ID of the node is given by the annotator which can be anything castable to `NODEID`, a **SAIL** data type.

This node is *logically* inserted in the graph and *physically* inserted in the current LC of the graph, i.e. the graphical attributes of the node are inherited from the current LC.

The graphical attributes for a specified node can be modified by the function `sw_setnodeattr`. This function does not affect settings of other nodes' graphical attributes.

A node can be deleted from a graph by the function `sw_deletenode`. It will be deleted both logically and physically, i.e. if the node is already displayed, it will be removed from **Swan** display window. Further, all the edges incident on the node will be deleted.

Edge

An edge is inserted in a graph by functions `sw_insertedge` or `sw_insertnodeedge`. The first function will insert an edge between `node1` and `node2`. If either `node1` or `node2` is not in the graph, the edge cannot be inserted successfully. Obviously it's cumbersome to insert the two end nodes of an edge every time before the edge can be inserted. The second function is more powerful from this point of view. Its main purpose is to insert an edge along with either or both of the nodes on which the edge is incident, as necessary.

`sw_insertbinedge` is used to insert edges into a binary tree. Identifying whether a node is its parent's left or right child makes edge insertion in a binary tree special.

In a *directed* graph, each edge will have a direction from `node1` to `node2`. In an *undirected* graph, the order of the edge's two nodes makes no difference.

If the edge's two nodes are in the same LC, the edge will also belong to this LC. Otherwise it connects two different LC's. In this case, the nearest common ancestor of the two LC's in the LC tree is found. Then the edge is assigned to that LC.

An edge can be deleted from a graph by the function `sw_deleteedge`. The graphical attributes of an edge can be modified by the function `sw_setedgeattr`. This function does not affect settings of graphical attributes of other edges.

3.3.3 Process Control

Swan provides several process control resources to allow the annotated program to be the main controller of the visualization. These resources include two control buttons, one graph edit menu and a set of process control functions. The viewer of the visualization has limited control over the running of the process.

RUN and STEP

Buttons **RUN** and **STEP** in the lower right hand corner of the **Swan** main window are used to control the process of the annotated program. Button **RUN** makes the program run in continuous mode. Button **STEP** makes the program run in step mode. When the program is running in step mode, it stops at any break point set by the annotator, waiting for the viewer to click the **STEP** button to run in step mode or the **RUN** button to run in continuous mode (break points are ignored in continuous mode). A green frame will appear around the button when the process is running in its corresponding mode. The annotator can enable or disable these two buttons by using `sw_enablebuttons` or `sw_disablebuttons`.

Graph Edit Menu

The edit button is contained in the upper left corner of the **Swan** main window. There are four menu items in its associated popup window. Each of these is a graph editing action, i.e. *insert a node*, *delete a node*, *insert an edge*, and *delete an edge*. After the viewer selects one of these items, the annotated program will be notified if it is waiting for any of these actions to be taken. The annotator can decide what to do according to the menu item selected. Thus, the semantics of these menu items are decided by the annotator, which is not necessarily the same as what the label of the item implies. The annotator can enable or disable any of these menu items by calling function `sw_enablemenuitem` or `sw_disablemenuitem`.

Process Control Functions

There are a few functions in **SAIL** which are used to control the process. Several other functions have side effects on the running process. A brief introduction of their usage is given here. Please refer to Section 3.5 for details.

Function `sw_init` initializes the **Swan** system. It must be the first **SAIL** function call in the annotated program. After **Swan** is initialized, the **Swan** main window is displayed and it is ready to receive **SAIL** function calls from the annotated program.

Function `sw_quit` should be called to quit **Swan**. This function informs **Swan** to delete all the elements it created and close all **Swan** windows.

Within the annotated program, `sw_run` and `sw_step` can be used to set the current mode for the running process. `sw_run` makes the program run in continuous mode in which the break points set in the annotated program are ignored. `sw_step` makes the program run in step mode so that the program will stop at any break point set by the annotator.

Break points in the annotated program are set by **SAIL** function `sw_break`. If this function is called within the program, the process will stop and wait for the viewer to click either **RUN** or **STEP** to resume running in the corresponding mode. Therefore, if `sw_break` is used in the program, be careful not to disable the **RUN** and **STEP** buttons at the same time.

Function `sw_wait` will make the program stop running and wait for a *signal*. Signals sent to the program by **Swan** to indicate which control button is clicked or which graph editing function is selected. Once the function receives one of the signals it is waiting for, it returns with the ID of that signal.

There is a group of **SAIL** functions to get viewer's input, either a string of characters or a sequence of mouse operations. Once the input is received, they return and the program continues. These functions include `sw_getstr`, `sw_pickgraph`, `sw_picknode`, `sw_pickedge` and `sw_pickpos`.

3.3.4 Errors

Errors could occur when **SAIL** functions are called with wrong arguments (i.e., out of range), functions are called in an inappropriate order, and under other circumstances. **SAIL** has an internal variable to keep the ID of the most recent error. When an error occurs, **SAIL** function `sw_errno` can be used to get the ID of the error. Function `sw_errmsg` can be used to get a brief description of an error. In addition, all the errors occurring during a **Swan** session are recorded in the file `error.log` for future reference.

3.4 Data Types and Constants

In addition to standard data types and constants in C or C++, several data types and constants are defined in **SAIL** for use in **SAIL** function calls.

The data types in **SAIL** for the annotator to use are as followings:

- GRAPHID
- LCID
- LCTYPE
- NODEID
- NODETYPE
- LABEL
- BOOLEAN
- INDEX
- IVALUE

The possible values of these types to be used in **SAIL** functions are defined as constants in header file `sail.h` which should be included in every annotated program. These constants are introduced with the data types they are associated with.

GRAPHID

The ID of a graph given by **SAIL** as the return value from function `sw_newgraph`. The annotator needs to declare a variable of this type for each graph.

`NULLGHAPHID` can be used as one of the arguments to `sw_newgraph` which indicates the annotator would like **SAIL** to generate an ID for the graph automatically.

LCID

The ID of a Layout Component (LC) given by **SAIL** as the return value from function `sw_newlc`. The annotator needs to declare a variable of this type for each LC.

`NULLLCID` can be used as one of the arguments to `sw_newlc` which indicates the annotator would like **SAIL** to generate an ID for the LC automatically.

LCTYPE

The type of a Layout Component. The valid types are:

- `ARRAYACROSS`

- `ARRAYDOWN`
- `LISTACROSS`
- `LISTDOWN`
- `CIRCLENET`
- `KKNET`
- `TREE`
- `BINTREE`
- `HIERARCHY`
- `MANUAL`

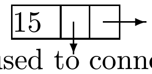
NODEID

The ID of a node. It can be anything castable to `NODEID`, such as `int, *int, long, *long,` etc. The ID of a node is determined by the annotator so that he can associate the node created in **Swan** with a certain data structure in the annotated program.

NODETYPE

The type of a node. The valid types are:

- `BOX`
- `TABOXL`
- `CIRCLE`

A node of type `TABOXL` looks like: 

The left box is used to display the label of the node. The middle and right boxes are used to connect two neighbor nodes if any.

LABEL

LABEL is a string type and is equivalent to `char*`. The labels of nodes and edges are always declared to be of type `LABEL`.

BOOLEAN

It is a Boolean type which only has two values `TRUE` or `FALSE`.

INDEX and IVALUE

These two types are closely related to the settings of graphical attributes of graphs, LC's, nodes, edges and some settings of **Swan**'s global environment.

Each value of `INDEX` represents a graphical attribute or a system switch. `IVALUE` is the actual value of that attribute to be set. The following list is used by **SAIL** function `sw_setgraphattr`, `sw_setlcattr`, `sw_setnodeattr`, `sw_setedgeattr`, and `sw_setcurrentattr`. The indices and their legal values are:

1. `GDIRECTED` - whether the graph is *directed* or *undirected*. Use `DIRECTED` and `UNDIRECTED` for the two types of graph respectively.
2. `GNEWLAYOUT` - a flag. It can be `TRUE` or `FALSE`. If it is `TRUE`, the layout of the graph will be updated when it is displayed again. Otherwise, not.
3. `GAUTORELAYOUT` - a flag. It can be `TRUE` or `FALSE`. If it is `TRUE`, whenever there is a modification which may cause the layout of the graph to change, the flag `GNEWLAYOUT` of the graph will be set as `TRUE`. Otherwise, the system will not change `GNEWLAYOUT`.
4. `GLAYOUTMODE` - a flag which can be either `MANULAYOUT` or `AUTOLAYOUT`. If it is `AUTOLAYOUT`, the position of the graph is determined by **Swan**. If it is `MANULAYOUT`, the viewer can change the position of the graph manually.
5. `GNODETYPE` - type of the node. It could be:
   - `BOX`
   - `TABOXL`
   - `CIRCLE`
6. `GNODECOLOR` - color of the node. The list of **Swan**'s colors is:
   - `BLACK`
   - `DGRAY`
   - `MGRAY`
   - `LGRAY`
   - `BLUEGRAY`
   - `LBLUE`
   - `PEACH`
   - `LCYAN`
   - `MCYAN`
   - `GRAY`
   - `MYELLOW`
   - `LYELLOW`
   - `MAGENTA`
   - `DGREEN`
   - `PASGREEN`
7. `GNODEFILLED` - whether the boxes or circles representing the node are filled or not. `TRUE` for filled. `FALSE` for unfilled.

8. GNODEWIDTH - width of the bounding box of the node. The value is an integer.
9. GNODEHEIGHT - height of the bounding box of the node. The value is an integer.
10. GNODELINETH - thickness of the line to draw the node. It can be either THICKLINE or THINLINE.
11. GNODEPOS - position of the node. Its value is a pair of coordinates (x, y) which is a relative position of the node within its layout component.
12. GEDGELENGTH - minimum length of the edge in a graph. The value is an integer.
13. GEDGELINETH - thickness of the line to draw the edge. It can be either THICKLINE or THINLINE.
14. GEDGELABEL - a flag to indicate whether the label of the edge should be displayed. It can be either TRUE, to display, or FALSE, not to display.
15. GEDGECOLOR - color of the edge. For valid color values see the list under GNODECOLOR.
16. GAUTOREDRAW - a flag to indicate whether **Swan** should redraw a graph automatically after the graph is modified. TRUE to redraw, FALSE not to redraw. This is a system-wide switch.

3.5 SAIL Function Library

3.5.1 Classification

Functions in **SAIL** are classified as follows.

1. *Functions for constructing and modifying graphs*
   - sw_newgraph
   - sw_deletegraph
   - sw_insertnode
   - sw_deletenode
   - sw_insertedge
   - sw_insertnodeedge
   - sw_insertbinedge
   - sw_deleteedge
2. *Functions for displaying graphs*
   - sw_displaygraph
   - sw_displayallgraphs
3. *Functions for specifying layout components*
   - sw_newlc

- sw_deletelc
- sw_setcurlc

4. *Functions for specifying graphic attributes*
    - sw_setcurrentattr
    - sw_setgraphattr
    - sw_getgraphattr
    - sw_setlcattr
    - sw_setnodeattr
    - sw_getnodeattr
    - sw_setedgeattr
    - sw_getedgeattr
    - sw_setgraphpos
    - sw_getgraphpos

5. *Functions for program status control*
    - sw_init
    - sw_quit
    - sw_clear
    - sw_run
    - sw_step
    - sw_break
    - sw_wait
    - sw_enablebuttons
    - sw_disablebuttons
    - sw_enablemenuitem
    - sw_disablemenuitem
    - sw_errmsg
    - sw_errno

6. *Input/Output*
    - sw_print
    - sw_getstr
    - sw_pickgraph
    - sw_picknode
    - sw_pickedge
    - sw_pickpos
    - sw_getpickedgraph

3.5.2 List of the functions

Following is a complete list of the C prototypes of **SAIL**'s functions in alphabetic order.

```
void sw_break(void) ;

void sw_clear(void) ;

BOOLEAN sw_deleteedge(GRAPHID, NODEID, NODEID) ;

BOOLEAN sw_deletelc(GRAPHID, LCID) ;

BOOLEAN sw_deletegraph(GRAPHID) ;

BOOLEAN sw_deletenode(GRAPHID, NODEID) ;

void sw_disablebuttons(int) ;

BOOLEAN sw_disablemenuitem(int, int) ;

BOOLEAN sw_displaygraph(GRAPHID) ;

BOOLEAN sw_displayallgraphs(void) ;

void sw_enablebuttons(int) ;

BOOLEAN sw_enablemenuitem(int, int) ;

BOOLEAN sw_errmsg(int, char*) ;

int sw_errno(void) ;

sw_getedgeattr(GRAPHID, NODEID, NODEID, INDEX, ...)  ;

sw_getgraphattr(GRAPHID, INDEX, IVALUE) ;

sw_getgraphpos(GRAPHID, int*, int*) ;

BOOLEAN sw_getnodeattr(GRAPHID, NODEID, INDEX n, ...)  ;

BOOLEAN sw_getpickedgraph(GRAPHID*) ;

void sw_getstr(char*) ;

BOOLEAN sw_init(void) ;

BOOLEAN sw_insertbinedge(GRAPHID, NODEID, NODEID, LABEL, BOOLEAN) ;

BOOLEAN sw_insertedge(GRAPHID, NODEID, NODEID, LABEL) ;

BOOLEAN sw_insertnode(GRAPHID, NODEID, LABEL) ;

BOOLEAN sw_insertnodeedge(GRAPHID, NODEID, LABEL, NODEID, LABEL, LABEL) ;

LCID sw_newlc(GRAPHID, LCID, LCTYPE) ;

GRAPHID sw_newgraph(GRAPHID, NODETYPE, FLAG) ;
```

```
BOOLEAN sw_pickedge(NODEID*, NODEID*) ;

BOOLEAN sw_pickgraph(GRAPHID*) ;

BOOLEAN sw_picknode(NODEID*) ;

BOOLEAN sw_pickpos(int*, int*) ;

void sw_print(LABEL) ;

void sw_quit(void) ;

void sw_run(void) ;

BOOLEAN sw_setcurlc(GRAPHID, LCID) ;

BOOLEAN sw_setcurrentattr(INDEX, IVALUE) ;

BOOLEAN sw_setedgeattr(GRAPHID, NODEID, NODEID, INDEX, ...)  ;

BOOLEAN sw_setgraphattr(GRAPHID, INDEX, IVALUE) ;

BOOLEAN sw_setgraphpos(GRAPHID, int, int) ;

BOOLEAN sw_setlcattr(LCID, INDEX, IVALUE) ;

BOOLEAN sw_setnodeattr(GRAPHID, NODEID, INDEX, IVALUE) ;

void sw_step(void) ;

int sw_wait(int) ;
```

3.5.3 Specifications

---

sw_break
syntax:

```
void sw_break(void)
```

parameters:

None.

return:

None.

description:

Set a break point in the annotated program. If the program runs in step mode, it stops at this break point and waits for the viewer to click button **Step** or **Run** to continue. If the program runs in continuous mode, the break point is ignored.

---

sw_clear
syntax:

```
void sw_clear(void)
```

parameters:

None.

return:

None.

description:

Delete all the graphs created in **Swan**. The **Swan** display window will also be cleared.

---

sw_deleteedge
syntax:

```
BOOLEAN sw_deleteedge(GRAPHID g, NODEID sn, NODEID en)
```

parameters:

`g` - ID of the graph from which the edge will be deleted.

`sn, en` - ID's of the two end nodes of the edge.

return:

`TRUE` if the edge is successfully deleted from graph `G`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Delete an edge between node `sn` and `en` in graph `G`. If graph `g` is undirected, the order of the two nodes makes no difference. If graph `g` is directed, the edge from `sn` to `en` is deleted. If the edge is displayed in **Swan** display window, it will be erased.

---

sw_deletegraph
syntax:

```
BOOLEAN sw_deletegraph(GRAPHID g)
```

parameters:

`g` - ID of the graph to be deleted.

return:

`TRUE` if the graph is successfully deleted.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Delete a graph `g`. All the nodes and edges in the graph will be deleted. If graph `g` is already displayed, it will be removed from the **Swan** display window.

---

sw_deletelc
syntax:

```
BOOLEAN sw_deletelc(GRAPHID g, LCID lc)
```

parameters:

`g` - ID of the graph from which the layout component will be deleted.

`lc` - ID of the layout component.

return:

`TRUE` if the layout component is successfully deleted from graph `g`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Delete a layout component from graph `g`. All the nodes and edges in this layout component will be removed from the window if this LC is currently displayed.

---

sw_deletenode
syntax:

```
BOOLEAN sw_deletenode(GRAPHID g, NODEID n)
```

parameters:

`g` - ID of the graph from which the node will be deleted.

`n` - ID of the node to be deleted.

return:

`TRUE` if the node is successfully deleted from graph `g`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Delete a node `n` from graph `g`. All the edges in graph `g` incident on `n` will be deleted too.

---

sw_disablebuttons
syntax:

```
void sw_disablebuttons(int buttons)
```

parameters:

`buttons` - ID's of buttons to be disabled. These ID's can be combined together using `or` operation. Valid button ID's are:

- `RUN`
- `STEP`

return:

None.

description:

Disable process control buttons. There is no effect if the viewer clicks a disabled button. Disabled buttons can be enabled by `sw_enablebuttons`.

---

sw_disablemenuitem
syntax:

```
BOOLEAN sw_disablemenuitem(int menu, int item)
```

parameters:

`menu` - ID of the menu in which the `item` will be disabled.

`item` - index of the item in the `menu`. Valid menu ID's and their item indices are:

- `EDITMENU`
  - `ITEMINSNODE`
  - `ITEMDELNODE`
  - `ITEMINSEDGE`
  - `ITEMDELEDGE`

return:

`TRUE` if the menu item is disabled.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Disable an `item` in the `menu` so that it cannot be selected by the viewer. Disabled menu items can be enabled by `sw_enablemenuitem`.

---

sw_displaygraph
syntax:

```
BOOLEAN sw_displaygraph(GRAPHID g)
```

parameters:

`g` - ID of the graph which will be displayed.

return:

`TRUE` if the graph is successfully displayed.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Display graph `g` in the **Swan** display window. If the layout mode is `AUTOLAYOUT` (the default value), the position of the graph is determined by **Swan** automatically. If the layout mode is `MANULAYOUT`, the position of the graph can be set by using the function `sw_setgraphpos`, and can also be changed by the viewer during run time.

---

sw_displayallgraphs
syntax:

```
void sw_displayallgraphs(void)
```

parameters:

None.

return:

None.

description:

Display all the graphs currently existing in **Swan** into the **Swan** display window. Positions of graphs are determined by the layout mode of each graph individually.

---

sw_enablebuttons
syntax:

```
void sw_enablebuttons(int buttons)
```

parameters:

`buttons` - ID's of buttons to be enabled. These ID's can be combined together using the `or` operation. Valid button ID's are:

- `RUN`
- `STEP`

return:

None.

description:

Enable process control buttons so that they can be used by the viewer. Enabled buttons can be disabled by `sw_disablebuttons`.

---

sw_errmsg
syntax:

```
BOOLEAN sw_errmsg(int errno, char* str)
```

parameters:

`errno` - the ID of an error.

`str` - a string to hold the message copied from **SAIL**.

return:

`TRUE` if the error number is valid and a message about this error is successfully copied to `str`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Gives a brief description of the error with ID `errno`.

---

sw_errno
syntax:

```
int sw_errno(void)
```

parameters:

None.

return:

The ID of the current system error.

description:

Check the type of the most recent error during a **Swan** session.

---

sw_enablemenuitem
syntax:

```
BOOLEAN sw_enablemenuitem(int menu, int item)
```

parameters:

`menu` - ID of the menu in which `item` will be disabled.

`item` - index of the item in `menu`. Please refer to descriptions of `sw_disablemenuitem` for valid menu ID's and their item indices.

return:

`TRUE` if the menu item is enabled.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Enable `item` in `menu` so that it can be selected. Items can be disabled by `sw_disablemenuitem`.

---

sw_getedgeattr
syntax:

```
BOOLEAN sw_getedgeattr(GRAPHID g, NODEID sn, NODEID en, INDEX n, ...)
```

parameters:

`g` - ID of the graph to which the edge belongs.

`sn, en` - ID's of the two end nodes of the edge.

`n` - the index to the attribute table. It specifies which attribute of the edge to retrieve.

`...` - the address of the memory to keep the retrieved value of the attribute. The value can be of different types depending on the attribute.

return:

`TRUE` if the value of the attribute is successfully retrieved.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Retrieve the value of a specific graphic attribute for an edge. Attribute `n` can be:

- `GEDGELENGTH`
- `GEDGELINETH`
- `GEDGECOLOR`

Valid values of these attributes can be found in Section 3.4.

---

sw_getgraphattr
syntax:

```
BOOLEAN sw_getgraphattr(GRAPHID g, INDEX n, IVALUE *value)
```

parameters:

`g` - ID of the graph whose attribute will be modified.

`n` - index to the attribute table. It specifies which attribute of the graph to set.

`value` - address of the memory holding the retrieved value of the attribute.

return:

`TRUE` if the value of the attribute is successfully modified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

This function retrieves the value of a specific attribute of a graph. Attribute `n` can be an one of

- `GDIRECTED`
- `GNEWLAYOUT`
- `GAUTORELAYOUT`
- `GLAYOUTMODE`
- `GNODETYPE`
- `GNODECOLOR`
- `GNODEFILLED`
- `GNODEWIDTH`
- `GNODEHEIGHT`
- `GNODELINETH`
- `GEDGELENGTH`
- `GEDGELINETH`
- `GEDGECOLOR`
- `GEDGELABEL`

Valid values of these attributes can be found in Section 3.4.

sw_getgraphpos

syntax:

```
BOOLEAN sw_getgraphpos(GRAPHID g_id, int *x, int *y)
```

parameters:

`g_id` - ID of the graph.

`x, y` - address holding the position of the graph to be retrieved.

return:

`TRUE` if a graph is successfully set at the position as specified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Retrieve the position of graph `g_id`.

---

sw_getnodeattr

syntax:

```
BOOLEAN sw_getnodeattr(GRAPHID g, NODEID n_id, INDEX n, ...)
```

parameters:

`g` - ID of the graph which contains the node `n_id`.

`n_id` - ID of the node from which the attribute will be retrieved.

`n` - index to the attribute table. It specifies which attribute of the node to get.

`...` - address for the value(s) of the attribute. There could be one or two addresses depending on the attribute.

return:

`TRUE` if the node's attribute is successfully retrieved.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Retrieve the value of a specific graphic attribute of a node. Attribute `n` can be:

- `GNODETYPE`
- `GNODECOLOR`
- `GNODEFILLED`
- `GNODEWIDTH`
- `GNODEHEIGHT`
- `GNODELINETH`
- `GNODELABEL`
- `GNODEPOS ---` *Two parameters follow: x and y.*

Valid values of these attributes can be found in Section 3.4.

---

sw_getpickedgraph
syntax:

```
BOOLEAN sw_getpickedgraph(GRAPHID* g_ptr)
```

parameters:

`g_ptr` - address storing the ID of the picked graph.

return:

`TRUE` if there is a picked graph. The ID of the picked graph will be copied to the place referenced by g_ptr.

`FALSE` if there is no picked graph. The value pointed at by `g_ptr` is not changed.

description:

Get the ID of the graph which is picked by the viewer. Note that if the viewer picks a node or an edge, the graph which the node or the edge belongs to is considered as the picked graph.

---

sw_getstr
syntax:

```
void sw_getstr(char* str)
```

parameters:

`str` - pointer to a character string.

return:

None.

description:

Get a string which is entered by the viewer on the keyboard. The `Return` key must be the last character entered to end the string. The annotator can use this function to get different kinds of input from the viewer by analyzing the string `str` with standard C function `sscanf`.

---

sw_init
syntax:

```
BOOLEAN sw_init(void)
```

parameters:

None.

return:

`TRUE` if **Swan** is successfully initialized.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Establish a connection between the annotated program and the **Swan** environment. If this function returns `TRUE`, all subsequent **SAIL** function calls in the annotated program can be accepted. This function should always be used as the first **SAIL** function call in any annotated program. If **Swan** is successfully initialized, the **Swan** main window will appear.

---

sw_insertbinedge
syntax:

```
BOOLEAN sw_insertbinedge(GRAPHID g, NODEID sn, NODEID en, LABEL str, BOOLEAN child)
```

parameters:

`g` - ID of the graph to which the edge will be inserted.

`sn, en` - ID's of the two end nodes of the edge.

`str` - label of the edge.

`child` - a flag to indicate whether `en` is the left or the right child of `sn` in the binary tree.

return:

`TRUE` if the edge is successfully inserted into graph `g`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Insert an edge from `sn` to `en` in graph `g`. Note `en` will be considered as either left or right child of node `sn` according to the flag `child`. This function can only be used to insert an edge into a layout component of type `BINTREE`.

---

sw_insertedge
syntax:

```
BOOLEAN sw_insertedge(GRAPHID g, NODEID sn, NODEID en, LABEL str)
```

parameters:

`g` - ID of the graph to which the edge will be inserted.

`sn, en` - ID's of the two end nodes of the edge.

`str` - label of the edge.

return:

`TRUE` if the edge is successfully inserted into graph `g`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Insert an edge between node `sn` and `en` in graph `g`. If graph `g` is undirected, the order the two nodes makes no difference. If graph `g` is directed, the edge will have a direction that is from node `sn` to `en`. If graph `g` is displayed, the edge will also be displayed in **Swan** display window. If any of the two end nodes does not exist in graph `g`, the edge will *not* be inserted.

---

sw_insertnode
syntax:

```
BOOLEAN sw_insertnode(GRAPHID g, NODEID n_id, LABEL str)
```

parameters:

`g` - ID of the graph which contains the node `n_id`.

`n_id` - ID of the node which will be inserted into graph `g`.

`str` - label of the node.

return:

`TRUE` if the node is successfully inserted into graph `g`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Insert node `n_id` into graph `g`. If graph `g` is already displayed, node `n_id` will also be displayed in **Swan** display window.

---

sw_insertnodeedge
syntax:

```
BOOLEAN sw_insertnodeedge(GRAPHID g, NODEID sn, LABEL str1, NODEID en,
   LABEL str2, LABEL str3)
```

parameters:

`g` - ID of the graph.

`sn, en` - ID's of the two end nodes of the edge.

`str1` - label of node `sn`.

`str2` - label of the node `en`.

`str3` - label of the edge.

return:

`TRUE` if the edge is successfully inserted into graph `g`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Insert an edge connecting nodes `sn` and `en` into graph `g`. If either node `sn` or `en` does not exist, it will be created and inserted into graph `g`. If graph `g` is already displayed, the edge will also be displayed in **Swan** display window.

---

sw_newlc
syntax:

```
LCID sw_newlc(GRAPHID g, LCID lc, LCTYPE lc_type)
```

parameters:

`g` - ID of the graph to which the created layout component will belong.

`lc` - ID of the layout component to be created.

`lc_type` - type of the layout component to be created.

return:

ID of the created layout component.

`NULLLCID` if errors occur. Use `sw_errno` to check the type of the error.

description:

Create a new layout component in graph `g` with ID `lc`. If `lc` is `NULLLCID`, **Swan** will generate an ID for this LC automatically.

---

sw_newgraph
syntax:

```
GRAPHID sw_newgraph(GRAPHID g, NODETYPE n_type, BOOLEAN f)
```

parameters:

`g` - ID of the graph to be created.

`n_type` - type of the nodes in the graph to be created.

`f` - a flag to indicate whether the graph will be *directed* or *undirected*. `f` can be set to `DIRECTED` for directed graph or `UNDIRECTED` for undirected graph.

return:

A graph ID if the graph is successfully created.

`NULLGRAPHID` if errors occur. Use `sw_errno` to check the type of the error.

description:

Create a new graph with the specified ID and node type. If the specified ID is `NULLGRAPHID`, **Swan** will generate an ID for the graph automatically. Otherwise, the function will return the specified ID. All nodes inserted into this graph will have the specified type unless explicitly modified by other functions.

---

sw_pickedge
syntax:

```
BOOLEAN sw_pickedge(NODEID* sn, NODEID* en)
```

parameters:

`sn, en` - addresses for the edge's nodes to be stored.

return:

`TRUE` if an edge is successfully picked by the viewer. The ID's of the edge's two nodes will be stored in the place referenced by `sn` and `en`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Request the viewer to pick an edge from the graphs currently displayed. The function will return after an edge is successfully picked or errors occur. If an edge is picked, the graph which the edge belongs to will be regarded as the currently picked graph whose ID can be retrieved by `sw_getpickedgraph`.

---

sw_pickgraph
syntax:

```
BOOLEAN sw_pickgraph(GRAPHID* g_ptr)
```

parameters:

`g_ptr` - addresses for the ID of the graph to be stored.

return:

`TRUE` if a graph is successfully picked by the viewer. The ID of the graph will be stored in the place referenced by `g_ptr`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Request the viewer to pick one graph from the graphs currently displayed. The function will return after a graph is successfully picked or errors occur.

---

sw_picknode
syntax:

```
BOOLEAN sw_picknode(NODEID* node_id)
```

parameters:

`node_id` - address for the ID of the node to be stored.

return:

`TRUE` if a node is successfully picked by the viewer. The ID of the node will be stored in the place referenced by `node_id`.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Request the viewer to pick a node from the graphs currently displayed. The function will return after a node is successfully picked or errors occur. If a node is picked, the graph which the node belongs to will be regarded as the currently picked graph whose ID can be retrieved by `sw_getpickedgraph`.

---

sw_pickpos
syntax:

```
BOOLEAN sw_pickpos(int* x_ptr, int* y_ptr)
```

parameters:

`x_ptr, y_ptr` - address for the position picked.

return:

`TRUE` if a position is successfully picked by the viewer. The X and Y coordinate of the position is stored in `x_ptr` and `y_ptr`, respectively.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Request the viewer to pick a position in the **Swan** display window. The picked position is represented by the pair of coordinates in **Swan** display plane.

---

sw_print
syntax:

```
void sw_print(LABEL str)
```

parameters:

`str` - a string of characters.

return:

None.

description:

Display `str` in the **Swan** I/O window.

---

sw_quit
syntax:

```
void sw_quit(void)
```

parameters:

None.

return:

None.

description:

End the **Swan** session. The **Swan** system window will be closed. All subsequent **SAIL** function calls in the annotated program will not be accepted unless the connection is re-established by another `sw_init` call.

---

sw_run
syntax:

```
void sw_run(void)
```

parameters:

None.

return:

None.

description:

Change the running mode of the annotated program from **step** to **run**.

---

sw_setcurlc
syntax:

```
BOOLEAN sw_setcurlc(GRAPHID g, LCID lc)
```

parameters:

`g` - ID of the graph to which the layout component `lc` belongs.

`lc` - ID of the layout component.

return:

`TRUE` if the current layout component is successfully set.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Set the current layout component of graph `g` to `lc`.

sw_setcurrentattr
syntax:

```
BOOLEAN sw_setcurrentattr(INDEX n, IVALUE value)
```

parameters:

`n` - index to the attribute table. It specifies which attribute to set.

`value` - new value of the attribute.

return:

`TRUE` if the value of the attribute is successfully modified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Modify the value of a default global graphical attribute in **Swan**. The valid attribute `n` can be:

- `GDIRECTED`
- `GNODETYPE`
- `GNODECOLOR`
- `GNODEFILLED`
- `GNODEWIDTH`
- `GNODEHEIGHT`
- `GNODELINETH`
- `GEDGELENGTH`
- `GEDGELINETH`
- `GEDGECOLOR`
- `GEDGELABEL`
- `GAUTOREDRAW`

Valid values for these attributes can be found in Section 3.4.

---

sw_setedgeattr

syntax:

```
BOOLEAN sw_setedgeattr(GRAPHID g, NODEID sn, NODEID en, INDEX n, ...)
```

parameters:

`g` - ID of the graph to which the edge belongs.

`sn, en` - ID's of the two end nodes of the edge.

`n` - index to the attribute table. It specifies which attribute of the edge to set.

`...` - new value of the attribute. The value can be of different types depending on the attribute.

return:

`TRUE` if the value of the attribute is successfully modified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Modify the value of a specific graphic attribute of an edge. The attribute `n` can be:

- `GEDGELENGTH`
- `GEDGELINETH`
- `GEDGECOLOR`

Valid values for these attributes can be found in Section 3.4.

---

sw_setgraphattr
syntax:

```
BOOLEAN sw_setgraphattr(GRAPHID g, INDEX n, IVALUE value)
```

parameters:

`g` - ID of the graph whose attribute will be modified.

`n` - index to the attribute table. It specifies which attribute of the graph to set.

`value` - new value of the attribute.

return:

`TRUE` if the value of the attribute is successfully modified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Modify the value of a specific attribute of a graph. The valid attribute `n` can be:

- GDIRECTED
- GNEWLAYOUT
- GAUTORELAYOUT
- GLAYOUTMODE
- GNODETYPE
- GNODECOLOR
- GNODEFILLED
- GNODEWIDTH
- GNODEHEIGHT
- GNODELINETH
- GEDGELENGTH
- GEDGELINETH
- GEDGECOLOR
- GEDGELABEL

Valid values for these attributes can be found in Section 3.4.

sw_setgraphpos

syntax:

```
BOOLEAN sw_setgraphpos(GRAPHID g_id, int x, int y)
```

parameters:

`g_id` - ID of the graph.

`x, y` - position of the graph to be set.

return:

`TRUE` if a graph is successfully set at the position as specified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Set the graph at a specific position on the **Swan** display plane.

---

sw_setlcattr
syntax:

```
BOOLEAN sw_setlcattr(GRAPHID g_id, LCID lc, INDEX n, IVALUE value)
```

parameters:

`g_id` - ID of the graph to which `lc` belongs.

`lc` - ID of the layout component.

`n` - index to the attribute table. It specifies which attribute of the layout component to set.

`value` - new value of the attribute.

return:

`TRUE` if the value of the attribute is successfully modified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Modify the value of a specific graphic attribute of a layout component. Attribute `n` can be:

- GNODETYPE
- GNODECOLOR
- GNODEFILLED
- GNODEWIDTH
- GNODEHEIGHT
- GNODELINETH
- GEDGELENGTH
- GEDGELINETH
- GEDGECOLOR

Valid values for these attributes can be found in Section 3.4.

sw_setnodeattr

syntax:

```
BOOLEAN sw_setnodeattr(GRAPHID g, NODEID n_id, INDEX n, ...)
```

parameters:

`g` - ID of the graph which contains the node `n_id`.

`n_id` - ID of the node to be modified.

`n` - index to the attribute table. It specifies which attribute of the node to set.

`...` - new value(s) of the attribute. There could be one or two values depending on the attribute.

return:

`TRUE` if the node's attribute is successfully modified.

`FALSE` if errors occur. Use `sw_errno` to check the type of the error.

description:

Modify the value of a specific graphic attribute of a node. The valid attribute `n` can be:

- `GNODETYPE`
- `GNODECOLOR`
- `GNODEFILLED`
- `GNODEWIDTH`
- `GNODEHEIGHT`
- `GNODELINETH`
- `GNODELABEL`
- `GNODEPOS ---` *Two parameters follow: x and y.*

Valid values for these attributes can be found in Section 3.4.

---

sw_step
syntax:

```
void sw_step(void)
```

parameters:

None.

return:

None.

description:

Change running mode of the annotated program from **run** to **step**.

---

sw_wait
syntax:

```
int sw_wait(int signals)
```

parameters:

`signals` - signals the process is waiting for. These signals can be any one of the following valid signals or their combinations by using `or` operation. Valid signals are:

- `RUN`
- `STEP`
- `INSNODE`
- `DELNODE`
- `INSEDGE`
- `DELEDGE`

return:

The signal received.

description:

Suspend running of the annotated program until any signal in the waiting list `signals` generated. Signals `RUN` and `STEP` generated when the corresponding button in the **Swan** window clicked. Signals `INSNODE`, `DELNODE`, `INSEDGE` and `DELEDGE` are generated when the corresponding menu item in graph editing menu is selected.

3.6 An Example

3.6.1 Annotation Techniques

Before a concrete example is introduced, some useful techniques to annotate a program are summarized here.

An annotation template

The following is a template which shows the basic procedures to annotate a program using **SAIL**:

```
#include "sail.h"    /* The header file for SAIL which must be
                        included in every annotated program.  */
GRAPHID g_id;        /* ID of the graph to be created in Swan.  */
LCID lc_id;          /* ID of the layout component to be created in Swan.
                        Every graph created in Swan must have at least one LC.
                        Here lc_id is the LC in the graph g_id.  */
main()
{
    ...
    sw_init();           /* Initialize Swan. This function must be the first
                            SAIL function call in the annotated program.  */
    ...
    create_a_graph();    /* Create a graph in Swan. Details are shown below.  */
    ...
    sw_displayallgraphs();  /* Draw all the graphs created in the Swan
                               display window.  */
    ...
    sw_quit();           /* Quit Swan. All the graphs created are deleted.
                            The Swan display window is cleared.  */
    ...
}

create_a_graph()
{
    ...
    /* Create an undirected graph in Swan. The shapes of nodes in the graph
       are circles. The graph ID returned from sw_newgraph will be used in
       following SAIL function calls to refer to this graph.  */
    g_id = sw_newgraph(NULLGRAPHID, CIRCLE, UNDIRECTED);

    /* sw_setgraphattr is used to set graphical attributes of the graph.
       Here, for example, the node color is light yellow, and both the width
       and height of the node are 20 pixels.  */
    sw_setgraphattr(g_id, GNODECOLOR, LYELLOW);
    sw_setgraphattr(g_id, GNODEWIDTH, 20);
```

```
        sw_setgraphattr(g_id, GNODEHEIGHT, 20);
        ...
        /* The LC is created for graph g_id. It informs Swan to draw the
           graph as a circle.  */
        lc_id = sw_newlc(g_id, NULLLCID, CIRCLENET);

        /* Set current LC as lc_id so that nodes or edges will be inserted
           into it if any insertion operations are executed.  */
        sw_setcurlc(g_id, lc_id);
        ...
        /* Insert a node into graph g_id. It will be inserted in the LC lc_id.  */
        sw_insertnode(g_id, (NODEID)node_id, node_label);
        ...
        /* Insert an edge into graph g_id. It will be inserted in the LC lc_id.  */
        sw_insertedge(g_id, (NODEID)node_id1, (NODEID)node_id2, edge_label);
        ...
    }
```

Run the program repeatedly

The viewer may want to run the annotated program several times without quitting **Swan**. The annotator can use the following mechanism to make it possible. First, rename the `main` function in the original C program (e.g., to `algo_1`), and then make the `main` function in the annotated program to contain an infinite loop which repeats the following steps:

- Initialize **Swan** once, then wait for the viewer to click **RUN** or **STEP** to start;
- Execute the function `algo_1`; and
- Wait for the viewer clicking **RUN** or **STEP** to run again.

Following is a code segment to describe this strategy:

```
main()
{
    ...
    sw_init();
    while (TRUE) {
        algo_1(...);
        sw_print("Press RUN or STEP to run again") ;
        sw_wait(RUN | STEP) ;
    }
    ...
}
```

The viewer can click the button **QUIT** to stop running the program.

Re-build a graph

SAIL provides several functions to create a graph. If an existing graph topology is modified during the running of the annotated program, **Swan** has to be notified about this modification in order to make future operations on this graph correct.

Usually, there are two approaches to communicate this information. One is to use available **SAIL** functions to do the modification directly. For example, if a node needs to be inserted, call `sw_insertnode`. If an edge needs to be deleted, call `sw_deleteedge` to delete it from the graph. However, the annotator also has to modify the data structure used by the annotated program accordingly to make the program really run on a modified graph. If the data structure is complicated, the modification will be a non-trivial task for the annotator.

The other method is to prepare a graph building function for each graph. Whenever this graph's topology is changed, this function is called. The main operations in this function are to delete the existing graph in **Swan** and build a new graph to replace the old one according to the graph's current structure. This method makes graph modification much easier. The main disadvantages are the possible inefficiency when the graph being modified is large and the change of graph layout and identity which may cause inconvenience or inconsistency.

Different methods can be chosen for different graphs to achieve better performance and make annotation easier.

Input and output

The **Swan** message window is divided into two parts: the top line is used by the annotator to display a one-line message, while the bottom line is used by the viewer to enter any data required by the annotator.

The annotator can use **SAIL** function `sw_print` to display a character string in the message window. Function `sw_getstr` is used to get input from the viewer.

Communicate with the viewer

The viewer of **Swan** applications has certain capabilities to control the running of the annotated program and modify logical structures of graphs under the annotator's permission.

The annotator can allow the viewer to control the running process by enabling the RUN or STEP buttons. He can insert function `sw_break` anywhere in the annotated program so that the process can stop at interesting points when the viewer steps through the program.

The annotator has to consider carefully whether he allows the viewer to modify the graphs generated by the annotated program. If modifications are allowed, facilities need to be built in the annotated program to support these modifications.

The annotator can create a function `modify_graph` to be inserted at certain places in the annotated program when he/she allows the viewer to modify one or more graphs.

The following code segment is an illustration of the function `modify_graph`.

```
modify_graph()
{
    ...
    sw_print("Please modify the graph or press STEP to continue") ;

    /* sflag is TRUE after RUN or STEP is clicked. Otherwise, FALSE. */
    sflag = FALSE ;

    /* Enable the control buttons and EDITMENU to allow graph modification.   */
    sw_enablebuttons(RUN|STEP) ;
    sw_enablemenuitem(EDITMENU, ITEMINSNODE) ;
    sw_enablemenuitem(EDITMENU, ITEMDELNODE) ;
    sw_enablemenuitem(EDITMENU, ITEMINSEDGE) ;
    sw_enablemenuitem(EDITMENU, ITEMDELEDGE) ;

    /* Start a loop. If any edit menu item is selected, take actions correspondingly.
       If RUN or STEP is clicked, exit the loop.   */
    while (TRUE) {
        switch (sw_wait(RUN|STEP|INSNODE|DELNODE|INSEDGE|DELEDGE)) {
        case STEP:
        case RUN:
            sflag = TRUE ;
            break ;
        case INSNODE:
            _insertnode() ;
            break ;
        case DELNODE:
            _deletenode() ;
            break ;
        case INSEDGE:
            _insertedge() ;
            break ;
        case DELEDGE:
            _deleteedge() ;
            break ;
        }
        if (sflag) break ;
    }

    /* Disable EDITMENU to disallow graph modifications.   */
    sw_disablemenuitem(EDITMENU, ITEMINSNODE) ;
    sw_disablemenuitem(EDITMENU, ITEMDELNODE) ;
    sw_disablemenuitem(EDITMENU, ITEMINSEDGE) ;
    sw_disablemenuitem(EDITMENU, ITEMDELEDGE) ;
```

```
    ...
}
```

_insertnode, _deletenode, _insertedge, and _deleteedge are functions provided by the annotator to actually carry out the modifications of the graphs.

Basically, modify_graph sets up a communication channel between the annotator and the viewer. When the annotator allows the viewer to modify the graph, he enables the graph editing menu items. Otherwise, he disables those menu items. If these menu items are enabled, **Swan** will inform the annotated program when the viewer chooses any one of them.

3.6.2 An example: bst.c

This is an example using **SAIL** to annotate a C++ program bst.c. The purpose of this program is to build and maintain a **binary search tree**. The program is annotated to show the structure of the binary search tree graphically. This program can be further modified to take advantage of **Swan**'s graph editing capability. Then the testing statements in the original program would not be necessary. With the function modify_graph, the viewer can modify the binary search tree interactively. From another point of view, this is also a simple example to show **Swan**'s potential as a debugging tool.

To annotate this program, the annotator creates a graph g. It has one layout component (LC) of the type BINTREE. The function rebuild_tree is called whenever the structure of the tree is changed. The function _tree_insert is recursively called to traverse the binary search tree and insert nodes and edges into graph g properly. rebuild_tree is inserted in main after each tree.print() statement to reflect the changes of the tree.

As mentioned above, a function modify_tree can be built to allow the viewer to modify the tree interactively. It can be inserted at some places in main where the annotator allows the tree to be modified.

The source code of the annotated program bst.c can be found in Appendix B.

References

[1] P. Eades and K. Sugiyama, "How to Draw a Directed Graph", *Journal of Information Processing*, Vol. 13, No. 4, 1990, pp. 424-437,

[2] T.M.J. Fruchterman and E.M. Reingold, "Graph Drawing by Force-directed Placement", *Software — Practice and Experience*, Vol. 21(11), November 1991, pp. 1129-1164.

[3] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo, "A Technique for Drawing Directed Graphs", *IEEE Transactions on Software Engineering*, Vol. 19, No. 3, March 1993, pp. 214-230.

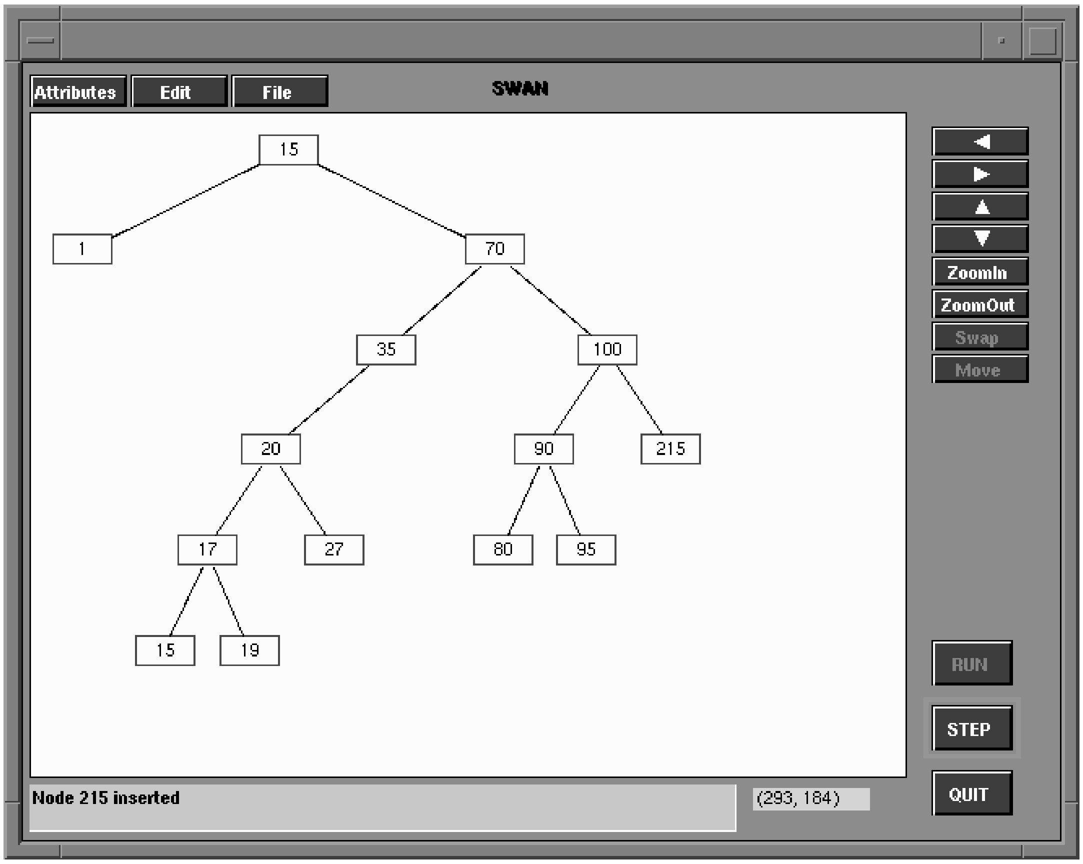


Figure 2: A binary search tree

[4] E.R. Gansner, S.C. North and K.-P. Vo, "DAG — A Program that Draws Directed Graphs", *Software — Practice and Experience*, Vol. 18(1), November 1988, pp. 1047-1062.

[5] T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs", *Information Processing Letters*, Vol. 31, April 1989, pp. 7-15.

[6] L.A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Sprirakis and A. Tuan, "A Browser for Directed Graphs", *Software — Practice and Experience*, Vol. 17(1), January 1987, pp. 61-76.

[7] C. Wetherell and A. Shannon, "Tidy Drawings of Trees", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, September 1979, pp. 514-520.

[8] J. Yang, "Swan — A Data Structure Visualization System", MS Thesis, Department of Computer Science, Viginia Tech, Blacksburg, VA, May 1995.

A The makefile for *bst*

```
#-------------------------------------------------------------------
#  This is the makefile for installation of the annotated Binary
#  Search Tree algorithm.
#  To create the executble file:
# make
#  To run the demo:
# bst
#
#  Note: Assume g++ library is installed on your local host.
#  If you use other compilers, please make necessary change.
#-------------------------------------------------------------------
PROG=bst
CC=g++
INCLUDES = -I../../include/
SRC = $(PROG).c
OBJ = $(PROG).o
TARGET = $(PROG)
LIBS = -L../../lib -lsail++ -lutils -lXt -lX11 -lm

$(TARGET): $(SRC) ../../lib/libsail++.a ../../lib/libutils.a

$(CC) -o $(TARGET) $(SRC) -DSWAN $(INCLUDES) $(LIBS)
#-------------------------------------------------------------------
```

B Source Code of *bst.c*

```
/************************************************************
 *  bst.c -- a C++ program to construct and maintain a  binary *
 *          search tree annotated with SAIL function calls    *
 ************************************************************/

/*
   To get the executable file without SWAN annotation:

        g++ -o bst bst.c

   To get the executable file with SWAN annotation:

        g++ -o bst -DSWAN bst.c -L. -lsail++ -lXt -lX11 -lm

   assuming sail.h and libsail++.a are in current directory.
*/

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <assert.h>

#ifdef SWAN
#include "sail.h"
#endif

#define FALSE 0
#define TRUE 1
#define DEFAULT_SIZE 10  // size for lists if no size is given.

template <class T> class BinNode {
  public:
    T element;          // The node's value
    BinNode<T>* left;   // Pointer to left child
    BinNode<T>* right;  // Pointer to right child

    BinNode() { left = right = NULL; }
    BinNode(T& e, BinNode<T>* l =NULL, BinNode<T>* r =NULL)
        { element = e; left = l; right = r; }
    ~BinNode() { }
    BinNode<T>* leftchild() { return left; }
    BinNode<T>* rightchild() { return right; }
    T& value() { return element; }
    BinNode<T>* setValue(T& val) { element = val; return this;}
    bool isLeaf() { return (left == NULL) && (right == NULL); }
};

template <class T> class BST {
  private:
    BinNode<T>* root;
    void clearhelp(BinNode<T>*);
    void inserthelp(BinNode<T>*&, T&);
```

```
    void removehelp(BinNode<T>*&, T&);
    BinNode<T>* findhelp(BinNode<T>*, T&);
    void printhelp(BinNode<T>*, int) const;
  public:
    BST() { root = NULL; }
    ~BST() { clearhelp(root); }
    BST<T>& clear()
        { clearhelp(root); root = NULL; return *this; }
    BST<T>& insert(T& val)
        { inserthelp(root, val); return *this; }
    BST<T>& remove(T& val)
        { removehelp(root, val); return *this; }
    BinNode<T>* deletemin(BinNode<T>*&);
    BinNode<T>* find(T& val)
        { return findhelp(root, val); }
    bool isEmpty() const
        { return root == NULL; }
    void print() const {
        if (root == NULL) cout << "The BST is empty.\n";
        else printhelp(root, 0);
    }
    BinNode<T>* getRoot(void) { return root ; }
} ;

template <class T>
void BST<T>::clearhelp(BinNode<T>* rt) {
    if (rt == NULL) return;
    clearhelp(rt->leftchild());
    clearhelp(rt->rightchild());
    delete rt;
}

template <class T>
void BST<T>::removehelp(BinNode<T>*& rt, T& val) {
    if (rt == NULL) cout << val << " is not in the tree.\n";
    else if (val < rt->value()) removehelp(rt->left, val);
    else if (val > rt->value()) removehelp(rt->right, val);
    else {  // Found it -- now delete it
        BinNode<T>* temp = rt;
        if (rt->left == NULL) rt = rt->right;
        else if (rt->right == NULL) rt = rt->left;
        else {  // Both children are non-empty
            temp = deletemin(rt->right);
           rt->setValue(temp->value());
        }
        delete temp;
    }
}

template <class T>
BinNode<T>* BST<T>::deletemin(BinNode<T>*& rt) {
    BinNode<T>* temp;
    assert(rt != NULL);
    if (rt->left != NULL) return deletemin(rt->left);
    else { temp = rt; rt = rt->right; return temp; }
}
```

```
template <class T>
BinNode<T>* BST<T>::findhelp(BinNode<T>* rt, T& val) {
    if (rt == NULL) return NULL;
    else if (val < rt->value())
        return findhelp(rt->leftchild(), val);
    else if (val == rt->value()) return rt;
    else return findhelp(rt->rightchild(), val);
}

template <class T>
void BST<T>::inserthelp(BinNode<T>*& rt, T& val) {
  if (rt == NULL) rt = new BinNode<T>(val, NULL, NULL);
  else if (val < rt->value()) inserthelp(rt->left, val);
  else inserthelp(rt->right, val);
}

template <class T>
void BST<T>::printhelp(BinNode<T> *rt, int level) const {
    int i;
    if (rt == NULL) return;
    printhelp(rt->leftchild(), level+1);
    for (i=0; i<level; i++) cout << "  ";  // indent
        cout << rt->value() << "\n";
    printhelp(rt->rightchild(), level+1);
}

// new functions added for the annotation purpose
#ifdef SWAN
rebuild_tree(BST<int>&) ;
_tree_insert(BinNode<int>*) ;
modify_tree(BST<int>&) ;
_insertnode(BST<int>&) ;
_deletenode(BST<int>&) ;

GRAPHID g = NULLGRAPHID ; // graph ID
LCID lc_handle = NULLLCID ;  // LC ID
#endif

// main() is to test whether the binary search tree is correctly
// constructed and maintained

int main()
{
    BST<int> tree;

#ifdef SWAN
    sw_init() ;                 // Swan initialization
    sw_disablebuttons(RUN) ;    // force viewer step through
    sw_step() ;                 // change mode to STEP
    sw_print("Press STEP to start...") ;
    sw_break() ;
#endif

    cout << "IsEmpty: " << tree.isEmpty() << "\n";
    tree.insert(10);
    tree.print();
```

```
#ifdef SWAN
    rebuild_tree(tree) ;
    sw_break() ;
#endif

    cout << "IsEmpty: " << tree.isEmpty() << "\n";
    tree.remove(10);
    tree.print();

#ifdef SWAN
    rebuild_tree(tree) ;
    sw_break() ;
#endif

    cout << "IsEmpty: " << tree.isEmpty() << "\n";
    tree.clear();
    cout << "IsEmpty: " << tree.isEmpty() << "\n";
    tree.insert(15);
    cout << "IsEmpty: " << tree.isEmpty() << "\n";
    tree.find(20);
    tree.find(15);
    tree.remove(20);
    tree.insert(20);
    tree.print();

#ifdef SWAN
    rebuild_tree(tree) ;
    sw_break() ;
#endif

    tree.remove(20);
    tree.print();

#ifdef SWAN
    rebuild_tree(tree) ;
    sw_break() ;
#endif

    tree.insert(70);
    cout << "IsEmpty: " << tree.isEmpty() << "\n";
    tree.insert(35);
    tree.insert(20);
    tree.insert(17);
    tree.insert(15);
    tree.insert(19);
    tree.insert(100);
    tree.insert(90);
    tree.insert(95);
    tree.insert(1);
    tree.print();

#ifdef SWAN
    rebuild_tree(tree) ;
    modify_tree(tree) ;
    sw_break() ;
#endif
```

```
    tree.find(100);
    tree.find(99);
    tree.find(20);
    cout << "Need to do some delete tests.\n";
    tree.remove(15);
    tree.print();

#ifdef SWAN
    rebuild_tree(tree) ;
    sw_break() ;
#endif

    tree.remove(15);
    tree.print();

#ifdef SWAN
    rebuild_tree(tree) ;
    sw_break() ;
#endif

    tree.clear();
    tree.print();

#ifdef SWAN
    rebuild_tree(tree) ;
    sw_break() ;
#endif

    cout << "IsEmpty: " << tree.isEmpty() << "\n";

#ifdef SWAN
    sw_quit() ;
#endif

    return(0);
}

#ifdef SWAN
rebuild_tree(BST<int>& tree_ptr)
{
    if (g != NULLGRAPHID)
        sw_deletegraph(g) ;

    g = sw_newgraph(NULLGRAPHID, BOX, UNDIRECTED) ;
    if (g == NULL) {
        cout << "g cannot be created\n" ;
         exit(0) ;
    }
    sw_setgraphattr(g, GNODECOLOR, DGREEN) ;
    sw_setgraphattr(g, GEDGELABEL, OFF) ;
    lc_handle = sw_newlc(g, NULLLCID, BINTREE) ;
    sw_setcurlc(g, lc_handle) ;
    _tree_insert(tree_ptr.getRoot()) ;
    sw_displayallgraphs() ;
}

modify_tree(BST<int>& tree)
```

```
{
    BOOLEAN    temp_flag = FALSE ;

    sw_print("Please modify the graph or STEP to continue...") ;
    sw_enablemenuitem(EDITMENU, ITEMINSNODE) ;
    sw_enablemenuitem(EDITMENU, ITEMDELNODE) ;
    while (TRUE) {
        switch (sw_wait(STEP|INSNODE|DELNODE)) {
        case STEP:
            temp_flag = TRUE ;
            break ;

        case INSNODE:
            _insertnode(tree) ;
            break ;

        case DELNODE:
            _deletenode(tree) ;
            break ;

        default:
            break ;
        }
        if (temp_flag) break ;
    }
    sw_disablemenuitem(EDITMENU, ITEMINSNODE) ;
    sw_disablemenuitem(EDITMENU, ITEMDELNODE) ;
}

_tree_insert(BinNode<int>* node_ptr)
{
    BinNode<int>    *left_child ;
    BinNode<int>    *right_child ;

    if (node_ptr == NULL)
        return TRUE ;
    sw_insertnode(g,(NODEID)node_ptr,inttostr((*node_ptr).value()));

    if ((left_child = (*node_ptr).leftchild()) != NULL) {
        _tree_insert(left_child) ;
        sw_insertbinedge(g, (NODEID)node_ptr, (NODEID)left_child,
         NULL, LEFTCHILD) ;
    }
    if ((right_child = (*node_ptr).rightchild()) != NULL) {
        _tree_insert(right_child) ;
        sw_insertbinedge(g, (NODEID)node_ptr, (NODEID)right_child,
         NULL, RIGHTCHILD) ;
    }
}

_insertnode(BST<int>& tree)
{
    char    temp_str[80] ;
    int     val ;

    sw_print("Please enter a node value:") ;
    sw_getstr(temp_str) ;
```

```
    sscanf(temp_str, "%d", &val) ;
    tree.insert(val) ;
    rebuild_tree(tree) ;
    sprintf(temp_str, "Node %d inserted", val) ;
    sw_print(temp_str) ;
}

_deletenode(BST<int>& tree)
{
    char      temp_str[80] ;
    LABEL     temp_label ;
    int       val ;
    NODEID    node_id ;

    sw_print("Please pick a node:") ;
    sw_picknode(&node_id) ;
    sw_getnodeattr(g, node_id, GNODELABEL, &temp_label) ;
    sscanf(temp_label, "%d", &val) ;
    tree.remove(val) ;
    rebuild_tree(tree) ;
    sprintf(temp_str, "Node %d removed", val) ;
    sw_print(temp_str) ;
}
#endif

/**************************************************************
 * End of bst.c                                               *
 **************************************************************/
```