Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.170 Problem Set 1 Cover Sheet — Due: Monday, September 18, 1989

Name: Marc Horowitz

Date: Monday, September 18, 1989

TA:

___ Earl Waldin
_✓_ Russell Finn
___ Cailein Barclay

Time Estimates: (Estimate time *before* doing actual work.)

	Estimated	Actual
Design	15 minutes	20 minutes
Implementation	30 minutes	30 minutes
Testing	30 minutes	120 minutes
Documentation	30 minutes	60 minutes

For TA use only!	
Grading	
Design	__ (out of 3)
Implementation	__ (out of 8)
Testing	__ (out of 4)
Documentation	__ (out of 5)
Late penalty	__
Total Score	__ (out of 20)

TA comments:

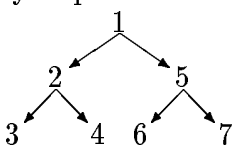
Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.170 Handed Out: Tuesday, September 12, 1989
Problem Set 1— Handout 4 Due: Monday, September 18, 1989

Using Data Abstractions

1 Problem Specification Amendments

The specifications given in the book do not give a specific form for the outputted breadth-first lists. In my implementation, I chose to print each level of the tree on a separate line.



For example, would be printed as

```
1
2 5
3 4 6 7 .
```

2 Design Documentation

2.1 Overview

The two programs are quite simple, designed to test the breadth-first routines. Each program has a `start_up` procedure which calls the list routine several times with several test trees or digraphs.

Originally, I designed my code to print the entire list on one line. After printing several tests, I decided that this was both aesthetically unpleasing and not very useful. This code was written to deal with only trees at first.

In the next implementation, I used two queues to print each depth on a separate line. I created a queue `thisline` and added the node number `n` which was passed in to the procedure. The main loop checks to see if `thisline` is empty. If it is, the procedure ends. If not, the queue `lastline` is created and set equal to `thisline`. The inner loop prints each member of `lastline` and adds its children to `thisline`. The loop is repeated until the new line contains no elements.

To write `breadth_first_graph`, all that was necessary was to maintain an intset of visited nodes. The initial node is added to the intset of visited nodes. Each node is added to the intset when it is displayed. In the inner loop, a node is added to `thisline` only if it had not been visited before.

2.2 Module Specifications

2.2.1 `breadth_first_tree`

putint = **proc** (i:int) **returns**()
effects: outputs the integer i followed by a space to the primary output stream

putstr = **proc** (s:string) **returns**()
effects: outputs the string s followed by a newline to the primary output stream

putnl = **proc** () **returns**()
effects: outputs a newline to the primary output stream

breadth_first_tree = **proc** (t: tree, n:int) **returns**()
effects: prints breadth-first list of t starting at node n.
Each successive level of nodes is printed on a separate line

start_up = **proc** () **returns**()
effects: tests breadth_first_tree with four different cases

test_tree_1 = **proc** () **returns** (tree)
effects: returns a tree with 1 node

test_tree_2 = **proc** () **returns** (tree)
effects: returns a simple binary tree

test_tree_3 = **proc** () **returns** (tree)
effects: returns a tree with nodes having only one child each

test_tree_4 = **proc** () **returns** (tree, int)
effects: returns a reasonably complex tree and the label of
a node for testing

2.2.2 `breadth_first_graph`

putint = **proc** (i:int) **returns**()
effects: outputs the integer i followed by a space to the primary
output stream

putstr = **proc** (s:string) **returns**()
effects: outputs the string s followed by a newline to the primary
output stream

putnl = **proc** () **returns**()
effects: outputs a newline to the primary output stream

breadth_first_graph = **proc** (d: digraph, n:int) **returns**()
effects: prints breadth-first list of d starting with node n.
Each successive level of nodes is printed on a separate line

start_up = **proc** () **returns**()
effects: tests breadth_first_graph with four different digraphs

test_graph_1 = **proc** () **returns** (digraph)
effects: returns a digraph with 1 node

test_graph_2 = **proc** () **returns** (digraph)
effects: returns a reasonably complex digraph with no loops

test_graph_3 = **proc** () **returns** (digraph,int)
effects: returns a small digraph and a node label for testing

test_graph_4 = **proc** () **returns** (digraph,int,int,int,int)
effects: returns a big, ugly, circular, non-connected digraph and
a bunch of labels to test with

3 Program Listings

3.1 breadth_first_tree

```
si=sequence[int]

putint = proc (i:int)
      % effects: outputs the integer i followed by a space to the primary
      % output stream

      stdout:stream := stream$primary_output()

      stream$puts(stdout,int$unparse(i))
      stream$putc(stdout,' ')
      end putint

putstr = proc (s:string)
      % effects: outputs the string s followed by a newline to the primary
      % output stream

      stdout:stream := stream$primary_output()

      stream$putl(stdout,s)
      end putstr

putnl = proc ()
      % effects: outputs a newline to the primary output stream

      stdout:stream := stream$primary_output()

      stream$putc(stdout,'\n')
      end putnl

breadth_first_tree = proc (t: tree, n:int)
      % effects: prints breadth-first list of t starting at node n.
      % Each successive level of nodes is printed on a separate line

      thisline:queue := queue$create()       % initalize by creating thisline
      queue$enq(thisline,n)                      % and enqueuing the index of the top
                                                 % of the tree

      while ~queue$is_empty(thisline) do
            lastline:queue := thisline
            thisline := queue$create()
            while ~queue$is_empty(lastline) do
                  i:int := queue$deq(lastline)
                  children:si := tree$children(t,i)

                  putint(i)
                  for j:int in si$elements(children) do
                        queue$enq(thisline,j)
                        end
                  end
            putnl()
```

```
            end
        end breadth_first_tree

%
% test trees for the debugger
%

start_up = proc ()
        % effects: tests breadth_first_tree with four different cases

        putstr("Testing test_tree_1")
        breadth_first_tree(test_tree_1(),1)
        putnl()

        putstr("Testing test_tree_2")
        breadth_first_tree(test_tree_2(),1)
        putnl()

        putstr("Testing test_tree_3")
        breadth_first_tree(test_tree_3(),1)
        putnl()

        t:tree,n:int := test_tree_4()
        putstr("Testing test_tree_4 from the top")
        breadth_first_tree(t,1)
        putnl()
        putstr("Testing test_tree_4 from the middle")
        breadth_first_tree(t,n)
        putnl()

        end start_up

test_tree_1 = proc () returns (tree)
        % effects: returns a tree with 1 node

        t:tree := tree$create()

        return(t)
        end test_tree_1

test_tree_2 = proc () returns (tree)
        % effects: returns a simple binary tree

        t:tree := tree$create()
        n1,n2,i:int

        t, n1 := tree$add_node_and_edge(t,1)
        t, n2 := tree$add_node_and_edge(t,1)

        t, i := tree$add_node_and_edge(t,n1)
        t, i := tree$add_node_and_edge(t,n1)

        t, i := tree$add_node_and_edge(t,n2)
        t, i := tree$add_node_and_edge(t,n2)
```

```
        return(t)
        end test_tree_2

test_tree_3 = proc () returns (tree)
        % effects: returns a tree with nodes having only one child each

        t:tree := tree$create()
        child:int

        t,child := tree$add_node_and_edge(t,1)
        t,child := tree$add_node_and_edge(t,child)
        t,child := tree$add_node_and_edge(t,child)
        t,child := tree$add_node_and_edge(t,child)
        t,child := tree$add_node_and_edge(t,child)

        return(t)
        end test_tree_3

test_tree_4 = proc () returns (tree, int)
        % effects: returns a reasonably complex tree and the label of
        % a node for testing

        t:tree := tree$create()
        n1,n2,n3,n4,i:int

        t, n1 := tree$add_node_and_edge(t,1)
        t, n2 := tree$add_node_and_edge(t,1)
        t, n3 := tree$add_node_and_edge(t,1)

        t, i := tree$add_node_and_edge(t,n1)
        t, i := tree$add_node_and_edge(t,i)
        t, i := tree$add_node_and_edge(t,i)

        t, i := tree$add_node_and_edge(t,n2)
        t, i := tree$add_node_and_edge(t,n2)

        t, n4 := tree$add_node_and_edge(t,n3)

        t, i := tree$add_node_and_edge(t,n4)
        t, i := tree$add_node_and_edge(t,n4)

        return(t,n4)
        end test_tree_4
```

3.2 breadth_first_graph

```
si=sequence[int]

putint = proc (i:int)
        % effects: outputs the integer i followed by a space to the primary
        % output stream

        stdout:stream := stream$primary_output()
```

```
        stream$puts(stdout,int$unparse(i))
        stream$putc(stdout,' ')
        end putint

putstr = proc (s:string)
        % effects: outputs the string s followed by a newline to the primary
        % output stream

        stdout:stream := stream$primary_output()

        stream$putl(stdout,s)
        end putstr

putnl = proc ()
        % effects: outputs a newline to the primary output stream

        stdout:stream := stream$primary_output()

        stream$putc(stdout,'\n')
        end putnl

breadth_first_graph = proc (d: digraph, n:int)
        % effects: prints breadth-first list of d starting with node n.
        % Each successive level of nodes is printed on a separate line

        visited:intset := intset$create()      % keep record of visited nodes
        thisline:queue := queue$create()       % initalize by creating thisline
        queue$enq(thisline,n)                      % and enqueuing the index of the top
        intset$insert(visited,n)                   % of the tree

        while ~queue$is_empty(thisline) do
                lastline:queue := thisline
                thisline := queue$create()
                while ~queue$is_empty(lastline) do
                        i:int := queue$deq(lastline)
                        children:si := digraph$successors(d,i)

                        putint(i)
                        for j:int in si$elements(children) do % add children to queue
                                if ~intset$member(visited,j) then % for next line if not
                                        queue$enq(thisline,j)             % if not already visited
                                        intset$insert(visited,j)
                                        end
                                end
                        end
                putnl()
                end
        end breadth_first_graph

%
% test trees for the debugger
%

start_up = proc ()
        % effects: tests breadth_first_graph with four different digraphs
```

```
        d:digraph
        n1,n2,n3,n4:int

        putstr("Testing test_graph_1")
        breadth_first_graph(test_graph_1(),1)
        putnl()

        putstr("Testing test_graph_2")
        breadth_first_graph(test_graph_2(),1)
        putnl()

        d,n1 := test_graph_3()

        putstr("Testing test_graph_3 one")
        breadth_first_graph(d,1)
        putnl()
        putstr("Testing test_graph_3 two")
        breadth_first_graph(d,n1)
        putnl()

        d,n1,n2,n3,n4 := test_graph_4();

        putstr("Testing test_graph_4 one")
        breadth_first_graph(d,1)
        putnl()
        putstr("Testing test_graph_4 two")
        breadth_first_graph(d,n1)
        putnl()
        putstr("Testing test_graph_4 three")
        breadth_first_graph(d,n2)
        putnl()
        putstr("Testing test_graph_4 four")
        breadth_first_graph(d,n3)
        putnl()
        putstr("Testing test_graph_4 five")
        breadth_first_graph(d,n4)
        putnl()
        end start_up

test_graph_1 = proc () returns (digraph)
        % effects: returns a digraph with 1 node

        d:digraph := digraph$create()

        return(d)
        end test_graph_1

test_graph_2 = proc () returns (digraph)
        % effects: returns a reasonably complex digraph with no loops

        d:digraph := digraph$create()
        n1,n2,n3,n4,i:int
```

```
        d, n1 := digraph$add_node_and_edge(d,1)
        d, n2 := digraph$add_node_and_edge(d,1)
        d, n3 := digraph$add_node_and_edge(d,1)

        d, i := digraph$add_node_and_edge(d,n1)
        d, i := digraph$add_node_and_edge(d,i)
        d, i := digraph$add_node_and_edge(d,i)

        d, i := digraph$add_node_and_edge(d,n2)
        d, i := digraph$add_node_and_edge(d,n2)

        d, n4 := digraph$add_node_and_edge(d,n3)

        d, i := digraph$add_node_and_edge(d,n4)
        d, i := digraph$add_node_and_edge(d,n4)

        return(d)
        end test_graph_2

test_graph_3 = proc () returns (digraph,int)
        % effects: returns a small digraph and a node label for testing

        d:digraph := digraph$create()
        n1,n2,n3,n4,i:int

        d, n1 := digraph$add_node_and_edge(d,1)
        d, n2 := digraph$add_node_and_edge(d,n1)
        d, n3 := digraph$add_node_and_edge(d,n2)
        d, n4 := digraph$add_node_and_edge(d,n3)
        d := digraph$add_edge(d,n4,1)

        return(d,n3)
        end test_graph_3

test_graph_4 = proc () returns (digraph,int,int,int,int)
        % effects: returns a big, ugly, circular, non-connected digraph and
        %                 a bunch of labels to test with

        d:digraph := digraph$create()
        n1,n2,n3,n4,n5,n6,i:int
        m1,m2,m3,m4,m5,m6,m7:int

        % The first piece

        d, n1 := digraph$add_node_and_edge(d,1)
        d, n2 := digraph$add_node_and_edge(d,1)
        d, n3 := digraph$add_node_and_edge(d,1)

        d, n4 := digraph$add_node_and_edge(d,n2)
        d, n5 := digraph$add_node_and_edge(d,n4)

        d := digraph$add_edge(d,n1,n4)
        d := digraph$add_edge(d,n1,n5)
        d := digraph$add_edge(d,n4,n2)
```

```
    d, n6 := digraph$add_node_and_edge(d,n3)

    d, i := digraph$add_node_and_edge(d,n6)
    d, i := digraph$add_node_and_edge(d,n6)

    % The second piece

    d, m1 := digraph$add_node(d)
    d, m2 := digraph$add_node_and_edge(d,m1)
    d, m3 := digraph$add_node_and_edge(d,m2)
    d, i := digraph$add_node_and_edge(d,m2)

    d, m4 := digraph$add_node(d)
    d, m5 := digraph$add_node_and_edge(d,m4)

    d, m6 := digraph$add_node(d)
    d, m7 := digraph$add_node_and_edge(d,m6)
    d, i := digraph$add_node_and_edge(d,m6)

    d := digraph$add_edge(d,m4,m2)
    d := digraph$add_edge(d,m5,m3)
    d := digraph$add_edge(d,m7,m5)

    return(d,n1,m1,m4,m6)
    end test_graph_4
```

4 Testing Description and Output

The trees and graphs I built for testing are intended to test boundary cases, average cases, and complex cases.

4.1 Description

For `breadth_first_tree`, I used four test trees. The first consists of just one element. The second is a simple binary tree. The third is a straight line of nodes with one child each. The fourth is complex, including nodes with anywhere from zero to three children at several depths. I printed lists of all of these trees from the root, and printed a list of the fourth tree from a node not the root. The first and third cases are are intended to test boundary cases of no children or just one. The second case is a normal case, and the fourth is supposed to be big and heinous.

For `breadth_first_graph`, I used four graphs. The first, again, is just one node. The second is the same as the most complex tree from the previous set of tests. The third is a ring of five nodes, which I printed from two different points. The fourth was a complex graph, with two disjoint sets of nodes, each pointing back upon itself several times. This was printed from several different places. The first case is a boundary condition. The second and is simple normal input. The third will test routines which do not handle remembering visited nodes correctly. (This test caught an error in my code, actually.) The fourth case tests everything, containing two disjoint pieces, several interconnected nodes, and nodes unreachable one direction, but reachable the other.

4.2 Output

4.2.1 `breadth_first_tree`

```
Testing test_tree_1
1

Testing test_tree_2
1
2 3
4 5 6 7

Testing test_tree_3
1
2
3
4
5
6

Testing test_tree_4 from the top
1
2 3 4
5 8 9 10
6 11 12
```

```
7

Testing test_tree_4 from the middle
10
11 12
```

4.2.2 breadth_first_graph

```
Testing test_graph_1
1

Testing test_graph_2
1
3 5 7
9 15 17 19
11 21 23
13

Testing test_graph_3 one
1
3
5
7
9

Testing test_graph_3 two
7
9
1
3
5

Testing test_graph_4 one
1
3 5 7
9 11 13
15 17

Testing test_graph_4 two
3
9 11
5

Testing test_graph_4 three
19
21
23 25

Testing test_graph_4 four
27
29 21
23 25

Testing test_graph_4 five
31
```

```
33 35
29
23
```