CS507 - Data Structures and Analysis of Algorithms

SOLUTIONS: Midterm exam - Spring 1998

1. *Given a string representing an arithmetic expression composed of one-digit numbers, arithmetic operators (+,-,* and /) and left and right parenthesis, write an algorithm which will return TRUE if the parenthesis are properly nested and balanced, and FALSE otherwise. (25 points)*

   Examples.

   *- in the expression ((5-3)*4+2/3 the parenthesis are not properly balanced (there are two left and only one right parenthesis*

   - in the expression 5)3-4)+7 the parenthesis are not properly nested (the first right parenthesis does not correspond to any left parenthesis;*

   *- in the expression ((2-5)/3+(2*7+4))*6 the parenthesis are correctly nested and balanced.*

Solution.

Assume that the expression is stored in a (null character -ended) string and that the class Stack (of characters) is defined with the prototype given on page 106 of the text.

The following function will check the parenthesis in the expression:

```
int parenthesis(char *expr){
    Stack s;
    for ( ; *expr != '\0' ; expr++)
        switch (*expr){
            case '(':
                s.push(*expr);
                break;
            case ')':
                if (!s.isEmpty())
                    s.pop();
                else;
                    return FALSE; // assuming that FALSE was defined
                break;
            default:
                break;
        }
    if (s.isEmpty())
        return TRUE;
    else
        return FALSE;
}
```

2. *Write a recursive subroutine (function) that takes as input a binary tree and a value K and returns TRUE if the value is stored in the tree and FALSE otherwise. (25 points).*

- *How many comparisons does this search need in the worst case? (10 points)*
- *Modify the algorithm such that when K is found it also produces a specification of the position of the node that stores it. (15 points)*

Solution.

Assume that the class BinNode is defined like on page 128 of the text. The subroutine will take two arguments: a pointer to a BinNode (the root of the tree) and a value x. A possible implementation will be:

```
int BTSearch(BinNode* n, BELEM x){
    if (n->value == x)
        return TRUE;
    else
        if (BTSearch(n->leftchild(), x))
            return TRUE;
        else
            return BTSearch(n->rightchild(), x));
}
```

- The worst case happens when x is not in the tree. In this case x has to be compared with all the values stored in the tree. Thus, for a binary tree with n nodes the highest possible number of comparisons will be n.
- To keep track of the path to the current node we can use a stack (use class definition on page 106) of characters in which 'l'-s and 'r'-s will be stored:

```
int BTSearch(BinNode* n, BELEM x){
    static Stack s;

    if (n->value == x){
        print_reverse(s);
        return TRUE;
    }
    else{
        s.push('l');
        if (BTSearch(n->leftchild(), x)
            return TRUE;
        else{
            s.pop();
            s.push('r');
            return BTSearch(n->rightchild(), x));
        }
    }
}
```

- here print_reverse is a function that takes as argument a stack of characters and prints the values stored in it in reversed order (the path will start at the bottom of the stack).

3. Given a directed, weighted graph and a vertex in that graph, write an algorithm that calculates the shortest paths from each of the vertices of the graph to the given vertex (single-destination shortest paths). (25 points)

Solution.

- The most straightforward solution is to add two new methods to the graph class:

  1. first_incident(vertex v) - the first incident edge to vertex v

  2. next_incident(vertex v) - the next incident edge to vertex v, in the list of incident nodes to v.

  The only modifications to Dijkstra's algorithm will be using these methods instead of first and next and using v1 to get the starting vertex of an edge rather than v2, its ending vertex:

```
void Dijkstra(Graph& G, int s) {
    int D[G.n()];
    for (int i=0; i<G.n(); i++)
        D[i] = INFINITY;
    D[s] = 0;
    for (i=0; i<G.n(); i++) {
        int v = minVertex(G, D);
        G.Mark[v] = VISITED;
        if (D[v] == INFINITY) return;
        for (Edge w = G.first_incident(v);
                        G.isEdge(w);
                        w = G.nextincident(w))
            if (D[G.v1(w)] > (D[v] + G.weight(w)))
                D[G.v1(w)] = D[v] + G.weight(w);
    }
}

int minVertex(Graph& G, int* D) { // Find min cost vertex
    int v;
    for (int i=0; i<G.n(); i++)
        if (G.Mark[i] == UNVISITED) { v = i; break; }
    for (i=0; i<G.n(); i++)  // Now find smallest value
    if ((G.Mark[i] == UNVISITED) && (D[i] < D[v])) v = i;
        return v;
}
```