More on linked lists

Use of "dummy" nodes to demark head and tail
-- avoids special cases of head / tail insertion and deletion

(see sentinel.cc)

Use of linked list for stack----avoids limitation of fixed sized arrays
-- all insertions, deletions happen at the front

(see stack.cc)

Heap--- data structure used to maintain minimum (or maximum)
value in a group
-- minimum found always at the "top" of the heap
-- small number of operations needed to add item, or delete minimum

A node has
-- parent (node "above" it in the data structure)
-- children (left and right--- for which it is the parent)
   -- may have null children

typedef struct nodeStruct node;
struct nodeStruct {
    char attribute[50];
    node *up, *left, *right;
};

Node with no children is called a "leaf"
Node with no parent is called the "root"

Heap property: value of parent is no larger than value of either of its children

Inserting a new node----
a) find a node with an empty child point (left or right)
   How?
   --- start at root.
   case --- one of children pointers == NULL, done
   else -- descend left or right, "randomly" and repeat

b) link new node in as a child
c) "walk" the heap backwards towards root
   swapping child and parent if heap property at
   parent is not maintained.

What's involved in swapping?
Heaps and recursion go hand-in-hand

Example - heap property at a subtree rooted in a node
   a) node's value no greater than maximum of its children's values, AND
   b) heap property is satisfied at the subtrees rooted in its left child, and
   c) trivially if node is NULL

Example -- height of a heap --- the longest path from any leaf to the root
     (path length is number of nodes on path)

     height of subtree rooted in a node is
       1 + maximum { height of subtree rooted in left child, height of subtree rooted
     in right child}
       0 if node is NULL

See heap.cc
/* sentinel.cc shows linked lists insert, delete, list using "sentinel" nodes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

typedef struct listNode node;
struct listNode {
   int val;
   node *next;
};

node headNode;
node tailNode;

/* insert in increasing order */
int Insert (int val) {
    node *n = (node *)malloc(sizeof(node));
    if( n == NULL ) return -1;
    n->val = val;

    node *here = &headNode;
    while( here->next->val < n->val ) {
        here = here->next;
    }

    n->next = here->next;
    here->next = n;
    return 1;
}

/* remove least item */
void DeleteFront() {
    node *ptr = headNode.next;

    /* make sure we aren't trying to delete from an empty list */
    if( headNode.next != &tailNode ) {
        headNode.next = headNode.next->next;
        free(ptr);
    }
}

/* list elements in list (excluding sentinels) */
void List() {
    node *ptr = headNode.next;
    printf("-----------------------------\n");
    while( ptr->next ) {
        printf("%p : (%d, %p)\n", ptr, ptr->val, (ptr->next==&tailNode)?NULL:ptr->next);
        ptr = ptr->next;
    }
}
printf("-----------------------------------\n");

int main() {

  /* initialize head and tail sentinel nodes */
  headNode.next = &tailNode;
  headNode.val  = -INT_MAX;
  tailNode.val  = INT_MAX;
  tailNode.next = NULL;

  char cmd[20];

  while(1) {
    printf("(insert N, del, list) > ");
    scanf("%s", cmd);

    if( strcmp(cmd, "insert")==0 || strcmp(cmd, "Insert")==0) {
      int val;
      /* get the next (integer) argument. Remember that scanf
        returns the number of matches */
      if( scanf("%d", &val) == 1) {
        /* return code indicates whether
           operation was successful */
        if( Insert(val) < 0 )
          printf("Unable to insert value %d\n", val);
        continue;
      }
    }

    if( strcmp(cmd, "del")==0 || strcmp(cmd, "Del")==0) {
      DeleteFront();
      continue;
    }

    if( strcmp(cmd, "list")==0 || strcmp(cmd, "List")==0) {
      List();
      continue;
    }
  }
}
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

enum OpCodes {ERR = -1, PLUS=0, MINUS=1, TIMES=2, DIV=3};

typedef struct nodeStruct node;
struct nodeStruct {
    float number;
    node *next;
};

/* transform non-numeric input into an opcode */
OpCodes InterpretType(char * op) {
    if ( strcmp(op, "+" ) == 0 ) return PLUS;
    if ( strcmp(op, "-" ) == 0 ) return MINUS;
    if ( strcmp(op, "*" ) == 0 ) return TIMES;
    if ( strcmp(op, "/" ) == 0 ) return DIV;
    return ERR;
}

node *Stack;

int Push(char *); int Pop(); int Top(); int Exec(char *); void List();

int main() {
char cmd[20];
char arg[20];
int ecode;

while(1) {
    printf("(push N, pop, list, top, exec op) > ");
    scanf("%s", cmd);

    if( strcmp(cmd, "push")==0 || strcmp(cmd, "Push")==0) {
        /* get second argument */
        scanf("%s", arg);

        if ( Push(arg) < 0 )
            printf("%s is not recognized as a number, or heap is exhausted\n", arg);
            continue;
    }

    if( strcmp(cmd, "pop")==0 || strcmp(cmd, "Pop")==0) {
        if ( Pop() < 0 )
            printf("Cannot pop an empty stack!\n");
            continue;
    }

    if( strcmp(cmd, "list")==0 || strcmp(cmd, "List")==0) {
        List();
        continue;
    }
}
if( strcmp(cmd, "top")==0 || strcmp(cmd, "Top")==0) {
    if ( Top() < 0 )
        printf("list is empty\n");
    continue;
}

if( strcmp(cmd, "exec")==0 || strcmp(cmd, "Exec")==0) {

    /* get second argument */
    scanf("%s", arg);
    ecode = Exec(arg);
    if( ecode == -1 )
        printf("%s is not recognized as a legitimate operator\n", arg);
    if( ecode  == -2 )
        printf("Insufficient number of arguments for Exec %s\n", arg);
    continue;
}
}

/* push a value onto the stack */
int Push(char * val) {
    float a;

    /* parse string using sscanf, if format is wrong then sscanf return is 0 */
    if ( sscanf(val, "%f", &a) != 1 ) return -1;

    /* the argument is legit.  Create a node */
    node *newnode = (node *)malloc( sizeof(node) );

    /* bail if malloc could not do it */
    if( newnode == NULL) return -1;

    /* fill in the node information */
    newnode->number = a;

    /* put the new node at the front */
    newnode->next = Stack;
    Stack = newnode;
    return 1;
}

/* pop the top of the stack */
int Pop() {
    node *ptr = Stack;

    /* test for empty stack */
    if( ptr == NULL) return -1;

    /* change the Stack pointer and release node */
    Stack = Stack->next;
    free( ptr );
    return 1;
}

/* print the value at the top of the stack */
int Top() {
    if( Stack == NULL) return -1;
printf("Top of stack has value %f\n", Stack->number);
return 1;
}

/* if there are two elements on the stack, First->Second, remove them, 
   compute First op Second, and put the result on the stack 
*/

int Exec(char *op) {

    /* is this a recognized operation? */
    OpCode X = InterpretType(op);

    /* type is ERR if it was not recognized */
    if( X == ERR) return -1;

    /* must have two arguments to work with */
    if( Stack == NULL || Stack->next == NULL ) return -2;

    float first, second, result;
    first  = Stack->number;
    second = Stack->next->number;

    switch (X) {
        case PLUS  : result = first+second; break;
        case MINUS : result = first-second; break;
        case TIMES : result = first*second; break;
        case DIV   : result = first/second; break;
        default     : printf("something is very wrong. Should never execute this line\n");
                      return -1;
    }

    /* remove top of stack */
    node *ptr = Stack;
    Stack = Stack->next;
    free(ptr);

    /* change the stored value at the top of the stack */
    Stack->number = result;

    return 1;
}

void List() {
    node *ptr = Stack;
    while( ptr ) {

        /* note use of %p to format a pointer for printing */
        printf("%p : (%f, %p)\n", ptr, ptr->number, ptr->next);
        ptr = ptr->next;
    }
}
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct nodeStruct node;
struct nodeStruct {
    char attribute[50];
    node *up, *left, *right;
};

int insertions = 0;
node *Heap;

/* find a node where a new item can be inserted */
node *FindLeaf( node *here) {
    /* push further if this node has left AND right subtrees */
    if( here->left != NULL && here->right != NULL) {
        /* if the number of insertions so far is even, go left */
        if( !insertions%2 ) return FindLeaf( here->left );
        return FindLeaf( here->right );
    }
    /* one or the other of the subtree links is empty, so we can insert here */
    return here;
}

/* swap the position of a parent and child in the heap */
void Swap( node *parent, node *child) {
    if( child->right ) child->right->up = parent;
    if( child->left ) child->left->up = parent;

    if( child == parent->left ) {
        node *tmp = child->right;
        child->right = parent->right;
        parent->right = tmp;
        parent->left = child->left;
        child->left = parent;
    } else {
        node *tmp = child->left;
        child->left= parent->left;
        parent->left= tmp;
        parent->right= child->right;
        child->right = parent;
    }
    child->up = parent->up;
    parent->up = child;

    /* the "out" links from the child and parent are correct now. Make sure the "in" links to them are also */
    if( child->up ) {
        if( child->up->right == parent) child->up->right = child;
        else child->up->left = child;
    }
    if( child->left ) child->left->up = child;
if (child->right) child->right->up = child;
if (parent->left) parent->left->up = parent;
if (parent->right) parent->right->up = parent;
}

/* find a node where a new item can be attached, then heapify */
void InsertNode(node *n) {
    n->up = NULL;
    n->left = NULL;
    n->right = NULL;

    if (Heap == NULL) {
        Heap = n;
        return;
    }

    /* find a first parent */
    node *here = FindLeaf(Heap);

    /* build a code that describes parent's subtree states */
    int code = 0;
    code |= (here->left == NULL) << 1;
    code |= (here->right == NULL);

    switch(code) {
        case 1: /* right subtree is null */
            here->right = n;
            break;
        case 2: /* left subtree is null */
            here->left = n;
            break;
        case 3: /* both are null, pick one */
            if (insertions%2 == 0) here->right = n;
            else here->left = n;
            break;
        default: printf("shouldn't get here!\n");
            exit(-1);
    }

    /* link to parent */
    n->up = here;

    /* heapify, i.e., walk up towards root, swapping child and parent until child
        is less than the parent */
    while (here && strcmp(here->attribute, n->attribute) > 0) {
        /* Swap n and here. */
        node *save_parent = here->up;
        Swap(here, n);
        here = save_parent;
    }

    /* if "here" is null then the Heap pointer changes to point to n */
    if (here == NULL) Heap = n;
}

/* heap property is satisfied at a node if it is satisfied
   in both subtrees, and the "here" node val is no larger
   than either subtree values */
int HeapProperty(node *here) {
    int pLeft;
    int pRight;

    /* base case of recursion */
    if( here == NULL) return 1;

    /* check property on left subtree */
    pLeft = HeapProperty( here->left);
    if(! pLeft ) return 0;

    /* check property on right subtree */
    pRight = HeapProperty( here->right);
    if(! pRight ) return 0;

    int code = (here->left == NULL)<<1;
    code |= (here->right == NULL);
    switch(code) {
        case 0: return ( ! (strcmp(here->attribute, here->left->attribute) > 0 )
                    && ! (strcmp(here->attribute, here->right->attribute) > 0));
                    break;
        case 1: return ! (strcmp(here->attribute,here->left->attribute) > 0);
                    break;
        case 2: return ! (strcmp(here->attribute,here->right->attribute) > 0);
                    break;
        default: return 1;
                    break;
    }
}

int AddElement(char *);
void TopElement();
void List( node *);
int Height( node *);

int main() {

    char cmd[20];
    char arg[20];
    int ecode;

    while(1) {
        printf("(add val, top, height, list, heap)>");
        scanf("%s", cmd);

        if( strcmp(cmd, "add")==0 ) {
            scanf("%s", arg);
            if( AddElement(arg) < 0 )
                printf("unable to add element %s\n", arg);
            continue;
        }

        if( strcmp(cmd, "top") == 0 ) {
            TopElement();
            continue;
        }
    }
if ( strcmp(cmd, "list") == 0 ) {
    List( Heap );
    continue;
}

if ( strcmp(cmd, "height") == 0 ) {
    printf("heap height is %d\n", Height(Heap));
    continue;
}

if ( strcmp(cmd, "heap") == 0 ) {
    printf("Heap property is %s satisfied\n", HeapProperty(Heap)?"":"not");
    continue;
}

int AddElement(char *attr) {
    /* get the space */
    node *ptr = (node *)malloc(sizeof(node));
    if( ptr == NULL ) return -1;

    /* copy in the attribute */
    strcpy(ptr->attribute, attr);
    InsertNode( ptr );
    insertions++;
    return 1;
}

void TopElement() {
    if( Heap == NULL ) {
        printf("heap is empty\n");
        return;
    }
    printf("Attribute of top element in heap is %s\n", Heap->attribute);
    return;
}

void List(node *here) {
    if( here == NULL ) return;

    printf("Attribute %s has parent %s, left child %s, right child %s\n", 
        here->attribute, (here->up?here->up->attribute:"none"), 
        (here->left?here->left->attribute:"none"), 
        (here->right?here->right->attribute:"none"));

    List(here->right);
    List(here->left);
    return;
}

int Height(node *here) {
    if( here == NULL ) return 0;
    int leftHeight = Height(here->left);
    int rightHeight = Height(here->right);
    return 1 + (leftHeight<rightHeight?rightHeight:leftHeight);
}