# CS 137: Assignment #10

Due on Friday, Dec 6, 2022, at 11:59 PM

No penalty for submitting late for up to 48 hours.

Submit all programs using the Marmoset Submission and Testing Server located at
https://marmoset.student.cs.uwaterloo.ca/

Victoria Sakhnini

Fall 2022

## Notes:

- Use the examples to guide your formatting for your output. Remember to terminate your output with a newline character.
- For this assignment, you may use any content covered until the end of Module 14.
- Your solution may not pass marmoset tests for some of the tests due to the wrong implementation or a memory leak. My advice is that even if you pass your own tests and are very positive about your implementation. Use Valgrind to detect memory leaks before submitting your solutions. gcc won't detect memory leaks, and maybe your local compiler won't catch them, either.

## Problem 1

Consider the following definition:

```
typedef struct dlnode {
        int data;
        struct dlnode *next;
         struct dlnode *prev;

} dlnode;

typedef struct ndlst {
        dlnode *head;  //points to the first element; otherwise, NULL
         dlnode *tail;  //points to the last element; otherwise, NULL
         int  len;      // number of elements in the list
} ndlst;
```

to define a doubly linked list where each node points to the previous and to the next nodes if exist; otherwise, it points to NULL. Also, the head points to the first element of the list, and the tail points to the last element in the list to make working with this list more efficient (such as when adding/removing from the front/end of the list).

Create a C program `doublyList.c` that completes the following functions (The first one was completed for you):

```
struct ndlst *dlistCreate(void){
     ndlst *ret = malloc(sizeof(ndlst));
     ret->head = NULL;
     ret->tail = NULL;
     ret->len = 0;
     return ret;
}

void dlistDestroy(ndlst *lst){ }
```

//returns the number of elements in the list
```
int dlistLength(ndlst *lst){  }
```

//Pre: `lst` is a valid list with a length of at least `n` where `n>=1`
//Post: `n`th item removed
```
void dlistRemoveElem(ndlst *lst, int n){  }
```

```
// lst is a valid list
void dlistAddToFront(ndlst *lst, int elem){  }

// lst is a valid list
void dlistAddToEnd(ndlst *lst, int elem){  }

//  lst  is a valid list, to be sorted in ascending order
void dlistSort(ndlst *lst){  }

// rotate the list  n  times to the left, assume lst not empty
void dlistRotateLeft(ndlst *lst, int n){


// rotate the list n times to the right, assume lst  not empty

void dlistRotateRight(ndlst *lst, int n)
```

You are to submit this file containing all implemented functions (that is, you must delete the test cases portion/`main` function). However, **you must keep the required included libraries and structure definitions.**

The sample code for testing is below. The **two** provided print functions are beneficial to check that you are building the list correctly with correct links for `next` and `prev` pointers.

```
void dlistPrint(ndlst *lst)  // I am using this function to print
                            // results but don't submit this function
{
    dlnode *node = lst->head;
    for (; node; node = node->next)
            printf("%d  ",node->data);
    printf("\n");
}

void dlistPrintReverse(ndlst *lst)   // I am using this function to
                        // print results but don't submit this function
{
    dlnode *node = lst->tail;
    for (; node; node = node->prev)
            printf("%d  ",node->data);
    printf("\n");
}

int main(void)
{
        ndlst *lst1 = dlistCreate();
        assert(dlistLength(lst1) == lst1->len);
        assert(dlistLength(lst1) == 0);
        dlistAddToEnd(lst1, 10);
        dlistAddToFront(lst1, -20);
```

```c
        dlistAddToFront(lst1, 0);
        dlistAddToEnd(lst1, 9);
        dlistAddToFront(lst1, -9);
        dlistAddToFront(lst1, 7);
        dlistAddToEnd(lst1, 40);
        assert(dlistLength(lst1) == lst1->len);
        assert(dlistLength(lst1) == 7);
        dlistPrint(lst1);
        dlistPrintReverse(lst1);
        dlistRemoveElem(lst1, 5);
        dlistRemoveElem(lst1, 1);
        dlistRemoveElem(lst1, 5);
        dlistAddToEnd(lst1, 100);
        assert(dlistLength(lst1) == lst1->len);
        assert(dlistLength(lst1) == 5);
        dlistPrint(lst1);
        dlistPrintReverse(lst1);
        dlistSort(lst1);
        dlistPrint(lst1);
        dlistRotateLeft(lst1, 2);
        dlistPrint(lst1);
        dlistRotateRight(lst1, 3);
        dlistPrint(lst1);
        dlistDestroy(lst1);
        return 0;
}

```

The output of the code above:

```
7   -9   0   -20   10   9   40

40   9   10   -20   0   -9   7

-9   0   -20   9   100

100   9   -20   0   -9

-20   -9   0   9   100

0   9   100   -20   -9

100   -20   -9   0   9
```

## Problem 2

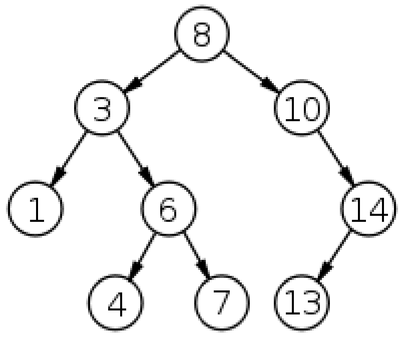
Consider the following definition:

```
struct bstnode {
      int item;
      struct bstnode *left;
      struct bstnode *right;
};

struct bst {
      struct bstnode *root;
      int count; // number of integers in the tree
};
```

**to define a BST (Binary Search Tree) of integers where the value of each node is bigger than all the integers on the left sub-tree (if exist) and smaller than all integers on the right sub-tree (if exist)**.

For example:



source: Wikipedia.

Create a C program `binaryTree.c` that completes the following functions (some of the functions were completed for you):

```
struct bst *bst_create(void)
{
      struct bst *t = malloc(sizeof(struct bst));
      t->root = NULL;
      t->count = 0;
      return t;
}
```

```
void destroy(struct bstnode *node)
{
     if (node != NULL)
     {
          destroy(node->left);
          destroy(node->right);
          free(node);
     }
}

void bst_destroy(struct bst *t)
{
    destroy(t->root);
    free(t);
}
```

// prints the integers in ascending order ending with `\n` character. There is a space after each printed number, Including the last one and before the `\n`. If the tree is empty, the function prints nothing.

```
void print(struct bstnode *node)
{
     if (node)
     {
          print(node->left);
          printf("%d ", node->item);
          print(node->right);
     }
}

void bst_print(struct bst *t)
{
     if (t->root)
     {
          print(t->root);
          printf("\n");
     }
}
```

// if num exists, don't add it, otherwise add it as a leaf. The result must be a BST tree

```
void bst_insert(int num, struct bst *t) {  }
```

// returns the minimal value in non empty BST tree.

```
int bst_min(struct bst *t){  }
```

// returns the maximal value in non empty BST tree.

```
int bst_max(struct bst *t){  }
```

// returns `1` if `val` in `t`, `0` otherwise

```
int bst_search(struct bst *t, int val) {  }
```

// returns the height of `t`. the height of the BST tree in the example above is 3. BST with one node has a height of 0. BST with no nodes has a height of -1.

```
int bst_height(struct bst *t){ }
```

You are to submit this file containing only the implemented functions (that is, you must delete the test cases portion). However, **<u>you must keep the required included libraries and the structure definitions.</u>**

Sample code for testing:

```
int main(void)
{
          struct bst *tree = bst_create();
          bst_insert(100, tree);
          bst_insert(150, tree);
          bst_insert(78, tree);
          bst_insert(88, tree);
          assert(bst_min(tree)==78);
          assert(bst_max(tree)==150);
          bst_print(tree);
          bst_insert(-130, tree);
          assert(tree->count==5);
          bst_insert(-130, tree);
          assert(tree->count == 5);
          bst_print(tree);
          assert(bst_search(tree, 100));
          assert(!bst_search(tree, 90));
          assert(bst_height(tree)==2);
          bst_destroy(tree);
 }
```

The output of the code above:

```
78 88 100 150
-130 78 88 100 150
```