```
/
    *---------------------------------------------------------------------------
    --------------
 * CS 261 Assignment 5 - heaps
 * Name: Jacob Karcz
 * Date: 11.07.2016
 -------------------*/

/
    ***************************************************************************
    **************

 ******************                 File: dynamicArray.h
     *****************


     **************************************************************************
     **************/


#ifndef DYNAMIC_ARRAY_H
#define DYNAMIC_ARRAY_H

#define TYPE void*

typedef struct DynamicArray DynamicArray;
typedef int (*compareFunction)(TYPE, TYPE);
typedef void (*printFunction)(TYPE);

struct DynamicArray;

DynamicArray* dyNew(int capacity);
void dyDelete(DynamicArray* array);

// Dynamic array
void dyAdd(DynamicArray* array, TYPE value);
void dyAddAt(DynamicArray* array, TYPE value, int position);
void dyPut(DynamicArray* array, TYPE value, int position);
void dyRemoveAt(DynamicArray* array, int position);
TYPE dyGet(DynamicArray* array, int position);
int dySize(DynamicArray* array);
void dySwap(DynamicArray* array, int position1, int position2);

// Stack
void dyStackPush(DynamicArray* stack, TYPE value);
void dyStackPop(DynamicArray* stack);
TYPE dyStackTop(DynamicArray* stack);
int dyStackIsEmpty(DynamicArray* stack);

// Bag
void dyBagAdd(DynamicArray* bag, TYPE value);
void dyBagRemove(DynamicArray* bag, TYPE value, compareFunction compare);
int dyBagContains(DynamicArray* bag, TYPE value, compareFunction compare);

// Ordered bag
void dyOrderedAdd(DynamicArray* bag, TYPE value, compareFunction compare);
void dyOrderedRemove(DynamicArray* bag, TYPE value, compareFunction compare);
int dyOrderedContains(DynamicArray* bag, TYPE value, compareFunction compare);

// Heap
void dyHeapAdd(DynamicArray* heap, TYPE value, compareFunction compare);
void dyHeapRemoveMin(DynamicArray* heap, compareFunction compare);
TYPE dyHeapGetMin(DynamicArray* heap);
void dyHeapSort(DynamicArray* heap, compareFunction compare);

// Iterator
typedef struct DynamicArrayIterator DynamicArrayIterator;

struct DynamicArrayIterator
{
    DynamicArray* array;
    int current;
};

DynamicArrayIterator* dyIteratorNew(DynamicArray* array);
void dyIteratorDelete(DynamicArrayIterator* iterator);
int dyIteratorHasNext(DynamicArrayIterator* iterator);
TYPE dyIteratorNext(DynamicArrayIterator* iterator);
void dyIteratorRemove(DynamicArrayIterator* iterator);

// Utility
/**
 * Prints the size, capacity, and elements of array, calling the print
 * function on each element.
 * @paremeter array
 * @paremeter print
 */
void dyPrint(DynamicArray* array, printFunction print);
void dyCopy(DynamicArray* source, DynamicArray* destination);

#endif






/
    ***************************************************************************
    **************

 ******************                 File: dynamicArray.c
     *****************


     ***************************************************************************
     **************/

#include "dynamicArray.h"
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
```

```c
#define TESTING

#ifndef TESTING
static void adjustHeap(DynamicArray* heap, int last, int position,
    compareFunction compare);
static void buildHeap(DynamicArray* heap, compareFunction compare);
#endif

struct DynamicArray {
    TYPE* data;
    int size;
    int capacity;
};

// --- Dynamic array ---

static void setCapacity(DynamicArray* array, int capacity)
{
    TYPE* data = malloc(sizeof(TYPE) * capacity);
    for (int i = 0; i < array->size; i++)
    {
        data[i] = array->data[i];
    }
    free(array->data);
    array->data = data;
    array->capacity = capacity;
}

static void init(DynamicArray* array, int capacity)
{
    assert(capacity > 0);
    array->data = NULL;
    array->size = 0;
    setCapacity(array, capacity);
}

DynamicArray* dyNew(int capacity)
{
    DynamicArray* array = malloc(sizeof(DynamicArray));
    init(array, capacity);
    return array;
}

void dyDelete(DynamicArray* array)
{
    free(array->data);
    free(array);
}

void dyAdd(DynamicArray* array, TYPE value)
{
    if (array->size >= array->capacity)
    {
        setCapacity(array, 2 * array->capacity);
    }
    array->data[array->size] = value;
    array->size++;
}

void dyAddAt(DynamicArray* array, TYPE value, int position)
{
    assert(position <= array->size);
    dyAdd(array, value);
    for (int i = array->size - 1; i > position; i--)
    {
        dySwap(array, i, i - 1);
    }
}

void dyPut(DynamicArray* array, TYPE value, int position)
{
    assert(position < array->size);
    array->data[position] = value;
}

void dyRemoveAt(DynamicArray* array, int position) {
    assert(position < array->size);
    for (int i = position; i < array->size - 1; i++) {
        array->data[position] = array->data[position + 1];
    }
    array->size--;
}

TYPE dyGet(DynamicArray* array, int position)
{
    //printf("arrayPosition = %d, arraySize = %d\n", position, array->size);

    assert(position <= array->size);
    return array->data[position];
}

int dySize(DynamicArray* array)
{
    return array->size;
}

void dySwap(DynamicArray* array, int position1, int position2)
{
    assert(position1 < array->size);
    assert(position2 < array->size);
    TYPE temp = array->data[position1];
    array->data[position1] = array->data[position2];
    array->data[position2] = temp;
}

// --- Stack ---

void dyStackPush(DynamicArray* stack, TYPE value)
{
    dyAdd(stack, value);
}
```

```
void dyStackPop(DynamicArray* stack)
{
    dyRemoveAt(stack, stack->size - 1);
}

TYPE dyStackTop(DynamicArray* stack)
{
    return dyGet(stack, stack->size - 1);
}

int dyStackIsEmpty(DynamicArray* stack)
{
    return stack->size == 0;
}

// --- Bag ---

static int findFirst(DynamicArray* array, TYPE value, compareFunction compare)
{
    for (int i = 0; i < array->size; i++)
    {
        if (compare(value, array->data[i]) == 0)
        {
            return i;
        }
    }
    return -1;
}

void dyBagAdd(DynamicArray* bag, TYPE value)
{
    dyAdd(bag, value);
}

void dyBagRemove(DynamicArray* bag, TYPE value, compareFunction compare)
{
    int position = findFirst(bag, value, compare);
    if (position != -1)
    {
        dyRemoveAt(bag, position);
    }
}

int dyBagContains(DynamicArray* bag, TYPE value, compareFunction compare)
{
    return findFirst(bag, value, compare) != -1;
}

// --- Ordered bag ---

static int binarySearch(DynamicArray* array, TYPE value, compareFunction
    compare)
{
    int low = 0;
    int high = array->size - 1;
    while (low <= high)
    {
        int middle = (low + high) / 2;
        if (compare(value, array->data[middle]) < 0)
        {
            high = middle - 1;
        }
        else if (compare(value, array->data[middle]) > 0)
        {
            low = middle + 1;
        }
        else
        {
            return middle;
        }
    }
    return low;
}

void dyOrderedAdd(DynamicArray* bag, TYPE value, compareFunction compare)
{
    int position = binarySearch(bag, value, compare);
    dyAddAt(bag,value, position);
}

void dyOrderedRemove(DynamicArray* bag, TYPE value, compareFunction compare)
{
    int position = binarySearch(bag, value, compare);
    if (compare(value, bag->data[position]) == 0)
    {
        dyRemoveAt(bag, position);
    }
}

int dyOrderedContains(DynamicArray* bag, TYPE value, compareFunction compare)
{
    int position = binarySearch(bag, value, compare);
    return compare(value, dyGet(bag, position)) == 0;
}
/
    ***************************************************************************
    **************
*******************************************************************************
    ***********
************************************ //--- Heap ---\
    \*************************************
*******************************************************************************
    ***********
*******************************************************************************
    **********/


/
    ***************************************************************************
    **************
 * Adjusts heap to maintain the heap property.
 *      @paremeter heap
 *      @paremeter last  index to adjust up to.
 *      @paremeter position  index where adjustment starts.
 *      @paremeter compare  pointer to compare function.

     **************************************************************************
     **************/
void adjustHeap(DynamicArray* heap, int last, int position, compareFunction
    compare) {
    // FIXME: implement
        <-----------------------------------------------------------------+
    int left = position * 2 + 1;
    int right = position * 2 + 2;
    int min; //max == last


    if (right < last) {                     //if right index w/in array, must
        be L&R nodes

        //min = indexSmallest (heap, left, right);   //calculate smallest child
        if ( compare(dyGet(heap, left), dyGet(heap, right)) == -1) //left <
            right
            min = left;
        else // ( compare(dyGet(heap, left), dyGet(heap, right)) == 1)
            min = right;

        //if min < pos, swap; adjust starting at min
        if (compare (dyGet(heap, min), dyGet(heap, position) ) == -1) {
            dySwap (heap, position, min);
            adjustHeap (heap, last, min, compare);
        }
    }
    if (left < last) {
        if (compare (dyGet(heap, left), dyGet(heap, position) ) == -1) {
            dySwap (heap, position, left);
            adjustHeap (heap, last, left, compare);
        }
    }
    else
        return;                                 //reached the bottom, done
            adjusting.
}

/
    ***************************************************************************
    **************
 * Builds a valid heap from an arbitrary array.
 * @param heap  array with elements in any order.
 * @param compare  pointer to compare function.

     **************************************************************************
     **************/
void buildHeap(DynamicArray* heap, compareFunction compare) {
    // FIXME: implement
        <-----------------------------------------------------------------+
    int last = dySize(heap);
    int i;

    for (i = (last/2)-1 ; i >= 0; i--)
        adjustHeap(heap, last, i, compare);
}

/
    ***************************************************************************
    **************
 * Adds an element to the heap.
 * @paremeter heap
 * @paremeter value  value to be added to heap.
 * @paremeter compare  pointer to compare function.

     **************************************************************************
     **************/
void dyHeapAdd(DynamicArray* heap, TYPE value, compareFunction compare) {
    // FIXME: implement
        <----------------------------------------------------------------+

    int parent;
    int position = dySize(heap);    // pos of new value
    dyAdd(heap, value);             //add new value

    while (position > 0) {
        parent = (position - 1)/2;
        if (compare(dyGet(heap, position), dyGet(heap, parent)) == -1) {
            dySwap(heap, parent, position);     //if pos < parent, swap
            position = parent;                      //percolate up
        }
        else
            return;     //if no swap, the tree is "balanced"
    }

}

/
    ***************************************************************************
    **************
 * Removes the first element, which has the min priority, form the heap.
 * @paremeter heap
 * @paremeter compare  pointer to the compare function.

     **************************************************************************
     **************/
void dyHeapRemoveMin(DynamicArray* heap, compareFunction compare) {
    // FIXME: implement
        <----------------------------------------------------------------+
    int last = dySize(heap) -1 ;
//    assert (last != 0);
//    printf("last: %d\n", last);

    dyPut(heap, dyGet(heap, last), 0);      //put last node at root
//    dySwap(heap, 0, last);                 //swap last and root
    dyRemoveAt(heap, last);                 //remove Last
    adjustHeap(heap, last, 0, compare);     //balance it out

}

/
    ***************************************************************************
    **************
 * Returns the first element, which has the min priority, from the heap.
 * @paremeter heap
 * @return  Element at the top of the heap.

     **************************************************************************
     **************/
TYPE dyHeapGetMin(DynamicArray* heap) {
    // FIXME: implement
        <-----------------------------------------------------------------+

    assert (dySize(heap) > 0);

    TYPE highPri = dyGet(heap, 0);
    return highPri;
}

/
    ***************************************************************************
    **************
 * Sorts arbitrary array in-place.
 * @paremeter heap  array with elements in arbitrary order.
 * @paremeter compare  pointer to the compare function.

     **************************************************************************
     **************/
void dyHeapSort(DynamicArray* heap, compareFunction compare) {
    // FIXME: implement
        <-----------------------------------------------------------------+

    // 1. build heap
    buildHeap(heap, compare);

    // 2. sort
    int heapSize = dySize(heap);
    int i;
    for (i = heapSize -1; i > 0; i--) {
        dySwap(heap, 0, i);
        adjustHeap(heap, i, 0, compare);
    }
}

// --- Iterator ---

DynamicArrayIterator* dyIteratorNew(DynamicArray* array)
{
    DynamicArrayIterator* iterator = malloc(sizeof(DynamicArrayIterator));
    iterator->array = array;
    iterator->current = 0;
    return iterator;
```

```c
}

void dyIteratorDelete(DynamicArrayIterator* iterator)
{
    free(iterator);
}

int dyIteratorHasNext(DynamicArrayIterator* iterator)
{
    return iterator->current < iterator->array->size;
}

TYPE dyIteratorNext(DynamicArrayIterator* iterator)
{
    TYPE value = dyGet(iterator->array, iterator->current);
    iterator->current++;
    return value;
}

void dyIteratorRemove(DynamicArrayIterator* iterator)
{
    iterator->current--;
    dyRemoveAt(iterator->array, iterator->current);
}

// --- Utility ---

void dyPrint(DynamicArray* array, printFunction print)
{
    printf("\nsize: %d\ncapacity: %d\n[\n", array->size, array->capacity);
    for (int i = 0; i < array->size; i++)
    {
        printf("%d : ", i);
        print(array->data[i]);
        printf("\n");
    }
    printf("]\n");
}

void dyCopy(DynamicArray* source, DynamicArray* destination)
{
    free(destination->data);
    init(destination, source->capacity);
    for (int i = 0; i < source->size; i++)
    {
        destination->data[i] = source->data[i];
    }
    destination->size = source->size;
}
```

```
/
    ***************************************************************************
    **************

 *********************                  File: task.h
     ********************


     **************************************************************************
     **************/

#ifndef TASK_H
#define TASK_H

#define TASK_NAME_SIZE 128

typedef struct Task Task;

struct Task {
    int priority;
    char name[TASK_NAME_SIZE];
};

Task* taskNew(int priority, char* name);
void taskDelete(Task* task);
int taskCompare(void* left, void* right);
void taskPrint(void* value);

#endif /* TASK_H */



/
    ***************************************************************************
    **************

 *********************                  File: task.c
     ********************


     **************************************************************************
     **************/
```

```
#include "task.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>


/**
 * Creates a new task with the given priority and name.
 * @parameter priority
 * @parameter name
 * @return  The new task.
 */
Task* taskNew(int priority, char* name) {
    // FIXME: implement
        <------------------------------------------------------------------
    struct Task *newTask;
    newTask = malloc(sizeof(struct Task));
    assert (newTask != NULL);

    newTask->priority = priority;
    strcpy(newTask->name, name);
    /*
     int i;
     int j = strlen(name);
     for (i = 0; i < j; i++)
     newTask->name[i] = name[i];
     for(int k = i; k < 128; k++)
     newTask->name[k] = ' ';
     */

    return newTask;
}

/**
 * Frees a dynamically allocated task.
 * @parameter task
 */
void taskDelete(Task* task) {
    free(task);
}

/**
 * Casts left and right to Task pointers and returns
 * -1 if left's priority < right's priority,
 *  1 if left's priority > right's priority,
 *  0 if left's priority == right's priority.
 * @param left  Task pointer.
 * @param right  Task pointer.
 * @returning
 */
int taskCompare(void *left, void *right) {
    // FIXME: implement
        <------------------------------------------------------------------

    /*
```

```
     assert (left != NULL);
     assert (right != NULL);
     */
    if (left == NULL && right != NULL)
        return -1;
    if (left != NULL && right == NULL)
        return 1;
    if (left == NULL && right == NULL)
        return 0;

    //typeCast TYPE void * as TYPE data
    struct Task *taskLeft;
    struct Task *taskRight;
    taskLeft = (struct Task *) left;
    taskRight = (struct Task *) right;

    //compare the values of the data structs
    if (taskLeft->priority < taskRight->priority)
        return -1;
    else if (taskLeft->priority > taskRight->priority)
        return 1;
    else
        return 0;

}



/**
 * Prints a task as a (priority, name) pair.
 * @param value  Task pointer.
 */
void taskPrint(void* value) {
    Task* task = (Task*)value;
    printf("(%d, %s)", task->priority, task->name);
}
```

```
/
    ***************************************************************************
    **************

 *********************                  File: toDo.c
     ********************


     **************************************************************************
     **************/

/
    ***************************************************************************
    **************
 * CS 261 Assignment 5
 * Name: Jacob Karcz
 * Date: 11.07.2016

     **************************************************************************
     **************/

#include "dynamicArray.h"
#include "task.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

/**
 * Loads into heap a list from a file with lines formatted like
 * "priority, name".
 * @parameter heap
 * @parameter file
 */
void listLoad(DynamicArray* heap, FILE* file) {
    const int FORMAT_LENGTH = 256;
    char format[FORMAT_LENGTH];
    snprintf(format, FORMAT_LENGTH, "%%d, %%%d[^\n]", TASK_NAME_SIZE);

    Task temp;
    while (fscanf(file, format, &temp.priority, &temp.name) == 2) {
        dyHeapAdd(heap, taskNew(temp.priority, temp.name), taskCompare);
    }
    //correct for unsorted list loading (w/o access to adjustHeap or buildHeap)

}

/**
 * Writes to a file all tasks in heap with lines formatted like
 * "priority, name".
 * @parameter heap
 * @parameter file
 */
void listSave(DynamicArray* heap, FILE* file) {
    for (int i = 0; i < dySize(heap); i++) {
        Task* task = dyGet(heap, i);
        fprintf(file, "%d, %s\n", task->priority, task->name);
    }
}

/**
 * Prints every task in heap.
 * @parameter heap
 */
void listPrint(DynamicArray* heap) {
    DynamicArray* temp = dyNew(1);
    dyCopy(heap, temp);
    while (dySize(temp) > 0) {
        Task* task = dyHeapGetMin(temp);
        printf("\n");
        taskPrint(task);
        printf("\n");
        dyHeapRemoveMin(temp, taskCompare);
    }
    dyDelete(temp);
}

/**
 * Handles the given command.
 * @parameter list
 * @parameter command
 */
void handleCommand(DynamicArray* list, char command) {
    // FIXME: Implement
        <-----------------------------------------------------------------
    char fileName[128];
    char *newLine;
    char taskName[128];
    FILE *filePointer;
    Task* newTask;
    Task* firstTask;
    int priority;
    DynamicArray *toDoneList = list;


    switch (command) {
        case 'l': // load to-do list from a file
            printf("Enter a file name to create a To-Do list: ");
            //fgets(fileName, 128, stdin);
            //if ((filePointer = fopen(fileName, "r")) == NULL)
            //    printf("Error in fopen: Error #%i\n", errno);

            // get filename from user input (from keyboard)
            if (fgets(fileName, sizeof(fileName), stdin) != NULL)
            {
                // remove trailing newline character
                newLine = strchr(fileName, '\n');
                if (newLine)
                    *newLine = '\0';
            }
            // open the file
            filePointer = fopen(fileName, "r"); // "r" = read
            if (filePointer == NULL) {
                fprintf(stderr, "Cannot open %s\n", fileName);
                break;
            }
            // load the list from the file
            listLoad(toDoneList, filePointer);
            // close the file
            fclose(filePointer);

            //organize list
            //            buildHeap(toDoneList, taskCompare);

            printf("To-Do list successfully loaded from file.\n\n");

            break;

        case 's': // save to-do list to a file
            if (dySize(toDoneList) > 0) {
                // get filename from user input (from keyboard)
                printf("Enter a filename to save to: ");
                if (fgets(fileName, sizeof(fileName), stdin) != NULL) {
                    // remove trailing newline character
                    newLine = strchr(fileName, '\n');
                    if (newLine)
                        *newLine = '\0';
                }
                // open the file
                filePointer = fopen(fileName, "w"); // "w" == write
                if (filePointer == NULL) {
                    fprintf(stderr, "Cannot open %s\n", fileName);
                    break;
                }
                // save the list to the file
                listSave(toDoneList, filePointer);
                // close the file
                fclose(filePointer);
                printf("To-Do list successfully saved to file.\n\n");
            }
            else
                printf("List cannot be saved, empty to-do list.\n\n");

            break;

        case 'a': // add a new task
            printf("Enter task name: ");
            // get task description from user input
            if (fgets(taskName, sizeof(taskName), stdin) != NULL) {
                // remove trailing newline character
                newLine = strchr(taskName, '\n');
                if (newLine)
                    *newLine = '\0';
            }
```

```
                do {// get task priority from user input
                    printf("Enter task's priority level {0 (high) - 1000 (low)}: ")
                        ;
                    scanf("%d", &priority);
                } while(!(priority >= 0 && priority <= 1000));

                // clear the trailing newline character
                while (getchar() != '\n');

                // create task and add the task to the heap
                newTask = taskNew(priority, taskName);
                dyHeapAdd(toDoneList, newTask, taskCompare);
                printf("Added task: '%s'\n\n", taskName);

                break;

            case 'g': // get the first task
                if (dySize(toDoneList) > 0) {
                    firstTask = (Task *)dyHeapGetMin(toDoneList);
                    printf("First task is: %s\n\n", firstTask->name);
                }
                else
                    printf("This list is empty.\n\n");

                break;

            case 'r': // remove the first task
                if (dySize(toDoneList) > 0) {
                    firstTask = (Task *)dyHeapGetMin(toDoneList);

                    //taskDelete(toDoneList->data[toDoneList->size]);
                    printf("Removed first task: '%s,'\n\n", firstTask->name);
                    taskDelete (dyHeapGetMin(list));
                    dyHeapRemoveMin(toDoneList, taskCompare);
                }
                else
                    printf("This list is already empty.\n\n");

                break;

            case 'p': //print the list
                if (dySize(toDoneList) > 0) {
                    listPrint(toDoneList);
                    printf("\n");
                }
                else
                    printf("List is empty, nothing to print.\n\n");
                break;

            case 'e': //exit the program
                printf("To-do list program terminated\n\n");

                break;
        }

    }
```

```c
int main() {
    // Implement
    printf("\n\n** TO-DO LIST APPLICATION **\n\n");
    DynamicArray* list = dyNew(8);
    char command = ' ';
    do {
        printf("Press:\n"
               "'l' to load to-do list from a file\n"
               "'s' to save to-do list to a file\n"
               "'a' to add a new task\n"
               "'g' to get the first task\n"
               "'r' to remove the first task\n"
               "'p' to print the list\n"
               "'e' to exit the program\n\n"
               );

        command = getchar();
        printf("\n");

        // Eat newlines
        while (getchar() != '\n');

        handleCommand(list, command);

    } while (command != 'e');

    while (dySize(list) > 0) {
        taskDelete (dyHeapGetMin(list));
        dyHeapRemoveMin(list, taskCompare);
    }

    dyDelete(list);

    return 0;
}
// valgrind --tool=memcheck --leak-check=yes toDo
```