**TODO LIST PROJECT** **IMPLEMENTATION in C (With BST and Dynamic Array)**

/\*

**File: bst.h**

Interface definition of the binary search tree data structure.

\*/

#ifndef \_\_BST\_H

#define \_\_BST\_H

/\* Defines the type to be stored in the data structure. These macros

\* are for convenience to avoid having to search and replace/dup code

\* when you want to build a structure of doubles as opposed to ints

\* for example.

\*/

# ifndef TYPE

# define TYPE void\*

# endif

/\* function used to compare two TYPE values to each other, define this in your compare.c file \*/

int compare(TYPE left, TYPE right);

/\* function used to print TYPE values, define this in your compare.c file \*/

void print\_type(TYPE curval);

struct BSTree;

/\* Declared in the c source file to hide the structure members from the user. \*/

/\* Initialize binary search tree structure. \*/

void initBSTree(struct BSTree \*tree);

/\* Alocate and initialize search tree structure. \*/

struct BSTree \*newBSTree();

/\* Deallocate nodes in BST. \*/

void clearBSTree(struct BSTree \*tree);

/\* Deallocate nodes in BST and deallocate the BST structure. \*/

void deleteBSTree(struct BSTree \*tree);

/\*-- BST Bag interface --\*/

int isEmptyBSTree(struct BSTree \*tree);

int sizeBSTree(struct BSTree \*tree);

void addBSTree(struct BSTree \*tree, TYPE val);

int containsBSTree(struct BSTree \*tree, TYPE val);

void removeBSTree(struct BSTree \*tree, TYPE val);

void printTree(struct BSTree \*tree);

# endif

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

/\*

**File: structs.h**

You can modify the structure to store whatever you'd like in your BST \*/

struct data {

int number;

char \*name;

};

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/\*

**File: compare.c**

\*/

#include <stdio.h>

#include <assert.h>

#include "bst.h"

#include "structs.h"

/\*----------------------------------------------------------------------------

very similar to the compareTo method in java or the strcmp function in c. it

returns an integer to tell you if the left value is greater then, less then, or

equal to the right value. you are comparing the number variable, letter is not

used in the comparison.

if left < right return -1

if left > right return 1

if left = right return 0

\*/

/\*Define this function, type casting the value of void \* to the desired type.

The current definition of TYPE in bst.h is void\*, which means that left and

right are void pointers. To compare left and right, you should first cast

left and right to the corresponding pointer type (struct data \*), and then

compare the values pointed by the casted pointers.

DO NOT compare the addresses pointed by left and right, i.e. "if (left < right)",

which is really wrong.

\*/

int compare(TYPE left, TYPE right)

{

struct data \*l, \*r;

l = (struct data \*) left;

r = (struct data \*) right;

/\* Comparing number part of the struct data \*/

if (l->number < r->number){

return -1;

} else if (l->number > r->number){

return 1;

} else {

return 0;

}

}

/\*Define this function, type casting the value of void \* to the desired type\*/

void print\_type(TYPE curval)

{

assert (curval != NULL);

struct data \*cur;

cur = (struct data \*) curval;

/\* Comparing by number part of struct data \*/

printf("%d", cur->number);

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/\* **dynArr.c: Dynamic Array implementation**. \*/

#include <assert.h>

#include <stdlib.h>

#include <stdio.h>

#include "dynamicArray.h"

struct DynArr

{

TYPE \*data; /\* pointer to the data array \*/

int size; /\* Number of elements in the array \*/

int capacity; /\* capacity ofthe array \*/

};

struct bag

{

struct DynArr \*dynArr;

};

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Dynamic Array Functions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* Initialize (including allocation of data array) dynamic array.

param: v pointer to the dynamic array

param: cap capacity of the dynamic array

pre: v is not null

post: internal data array can hold capacity elements

post: v->data is not null

\*/

void \_initDynArr(DynArr \*v, int capacity)

{

assert(capacity > 0);

assert(v!= 0);

v->data = (TYPE \*) malloc(sizeof(TYPE) \* capacity);

assert(v->data != 0);

v->size = 0;

v->capacity = capacity;

}

/\* Allocate and initialize dynamic array.

param: cap desired capacity for the dyn array

pre: none

post: none

ret: a non-null pointer to a dynArr of cap capacity

and 0 elements in it.

\*/

DynArr\* createDynArr(int cap)

{

DynArr \*r;

assert(cap > 0);

r = malloc(sizeof( DynArr));

assert(r != 0);

\_initDynArr(r,cap);

return r;

}

/\* Deallocate data array in dynamic array.

param: v pointer to the dynamic array

pre: v is not null

post: d.data points to null

post: size and capacity are 0

post: the memory used by v->data is freed

\*/

void freeDynArr(DynArr \*v)

{

assert(v!=0);

if(v->data != 0)

{

free(v->data); /\* free the space on the heap \*/

v->data = 0; /\* make it point to null \*/

}

v->size = 0;

v->capacity = 0;

}

/\* Deallocate data array and the dynamic array ure.

param: v pointer to the dynamic array

pre: v is not null

post: the memory used by v->data is freed

post: the memory used by d is freed

\*/

void deleteDynArr(DynArr \*v)

{

assert (v!= 0);

freeDynArr(v);

free(v);

}

/\* Resizes the underlying array to be the size cap

param: v pointer to the dynamic array

param: cap the new desired capacity

pre: v is not null

post: v has capacity newCap

\*/

void \_dynArrSetCapacity(DynArr \*v, int newCap)

{

int i;

TYPE \*oldData;

int oldSize = v->size;

oldData = v->data;

printf("========Resizing========\n");

/\* Create a new dyn array with larger underlying array \*/

\_initDynArr(v, newCap);

for(i = 0; i < oldSize; i++){

v->data[i] = oldData[i];

}

v->size = oldSize;

free(oldData);

#ifdef ALTERNATIVE

int i;

/\* Create a new underlying array\*/

TYPE \*newData = (TYPE\*)malloc(sizeof(TYPE)\*newCap);

assert(newData != 0);

/\* copy elements to it \*/

for(i = 0; i < v->size; i++)

{

newData[i] = v->data[i];

}

/\* Delete the oldunderlying array\*/

free(v->data);

/\* update capacity and size and data\*/

v->data = newData;

v->capacity = newCap;

#endif

}

/\* Get the size of the dynamic array

param: v pointer to the dynamic array

pre: v is not null

post: none

ret: the size of the dynamic array

\*/

int sizeDynArr(DynArr \*v)

{

assert(v!=0);

return v->size;

}

/\* Adds an element to the end of the dynamic array

param: v pointer to the dynamic array

param: val the value to add to the end of the dynamic array

pre: the dynArry is not null

post: size increases by 1

post: if reached capacity, capacity is doubled

post: val is in the last utilized position in the array

\*/

void addDynArr(DynArr \*v, TYPE val)

{

assert(v!=0);

/\* Check to see if a resize is necessary \*/

if(v->size >= v->capacity)

\_dynArrSetCapacity(v, 2 \* v->capacity);

v->data[v->size] = val;

v->size++;

}

/\* Get an element from the dynamic array from a specified position

param: v pointer to the dynamic array

param: pos integer index to get the element from

pre: v is not null

pre: v is not empty

pre: pos < size of the dyn array and >= 0

post: no changes to the dyn Array

ret: value stored at index pos

\*/

TYPE getDynArr(DynArr \*v, int pos)

{

assert(v!=0);

assert(pos <= v->size); // Updated = 11 Aug

assert(pos >= 0);

return v->data[pos];

}

/\* Put an item into the dynamic array at the specified location,

overwriting the element that was there

param: v pointer to the dynamic array

param: pos the index to put the value into

param: val the value to insert

pre: v is not null

pre: v is not empty

pre: pos >= 0 and pos < size of the array

post: index pos contains new value, val

\*/

void putDynArr(DynArr \*v, int pos, TYPE val)

{

assert(v!=0);

assert(pos < v->size);

assert(pos >= 0);

v->data[pos] = val;

}

/\* Swap two specified elements in the dynamic array

param: v pointer to the dynamic array

param: i,j the elements to be swapped

pre: v is not null

pre: v is not empty

pre: i, j >= 0 and i,j < size of the dynamic array

post: index i now holds the value at j and index j now holds the value at i

\*/

void swapDynArr(DynArr \*v, int i, int j)

{

TYPE temp;

assert(v!=0);

assert(i < v->size);

assert(j < v->size);

assert(i >= 0);

assert(j >= 0);

temp = v->data[i];

v->data[i] = v->data[j];

v->data[j] = temp;

}

/\* Remove the element at the specified location from the array,

shifts other elements back one to fill the gap

param: v pointer to the dynamic array

param: idx location of element to remove

pre: v is not null

pre: v is not empty

pre: idx < size and idx >= 0

post: the element at idx is removed

post: the elements past idx are moved back one

\*/

void removeAtDynArr(DynArr \*v, int idx){

int i;

assert(v!= 0);

assert(idx < v->size);

assert(idx >= 0);

//Move all elements up

for(i = idx; i < v->size-1; i++){

v->data[i] = v->data[i+1];

}

v->size--;

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Stack Interface Functions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* Returns boolean (encoded in an int) demonstrating whether or not the

dynamic array stack has an item on it.

param: v pointer to the dynamic array

pre: v is not null

post: none

ret: >0 if empty, otherwise 0

\*/

int isEmptyDynArr(DynArr \*v)

{

assert(v!= 0);

return !(v->size);

/\* alternatively:

if(v->size == 0)

return 1;

else return 0;

\*/

}

/\* Push an element onto the top of the stack

param: v pointer to the dynamic array

param: val the value to push onto the stack

pre: v is not null

post: size increases by 1

if reached capacity, capacity is doubled

val is on the top of the stack

\*/

void pushDynArr(DynArr \*v, TYPE val)

{

assert(v!=0);

addDynArr(v, val);

}

/\* Returns the element at the top of the stack

param: v pointer to the dynamic array

pre: v is not null

pre: v is not empty

post: no changes to the stack

\*/

TYPE topDynArr(DynArr \*v)

{

assert(v!=0);

assert(!isEmptyDynArr(v));

return v->data[v->size-1];

}

/\* Removes the element on top of the stack

param: v pointer to the dynamic array

pre: v is not null

pre: v is not empty

post: size is decremented by 1

the top has been removed

\*/

void popDynArr(DynArr \*v)

{

assert(v!=0);

assert(! isEmptyDynArr(v));

v->size--;

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bag Interface Functions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* Returns boolean (encoded as an int) demonstrating whether or not

the specified value is in the collection

true = 1

false = 0

param: v pointer to the dynamic array

param: val the value to look for in the bag

pre: v is not null

pre: v is not empty

post: no changes to the bag

\*/

int containsDynArr(DynArr \*v, TYPE val)

{

int i = 0;

assert(v!=0);

assert(!isEmptyDynArr(v));

for(i = 0; i < sizeDynArr(v); i++)

if(compare(v->data[i], val) == 0)

return 1;

return 0;

}

/\* Removes the first occurrence of the specified value from the collection

if it occurs

param: v pointer to the dynamic array

param: val the value to remove from the array

pre: v is not null

pre: v is not empty

post: val has been removed

post: size of the bag is reduced by 1

\*/

void removeDynArr(DynArr \*v, TYPE val)

{

int i = 0;

assert(v!=0);

assert(!isEmptyDynArr(v));

assert(containsDynArr(v,val));

for(i = 0; i < sizeDynArr(v); i++)

if(compare(v->data[i], val) == 0)

{

removeAtDynArr(v,i);

break;

}

}

/\* Copy elements from a dynamic array to another dynamic array

param: source pointer to the source dynamic array

param: destination pointer to the destination dynamic array

pre: s is not null and s is not empty

post: destination is initialized

post: the elements from source are copied to destination

\*/

void copyDynArr(DynArr \*source, DynArr \*destination)

{

int i;

assert(source->size > 0);

\_initDynArr(destination, source->capacity);

/\* copy elements to destination array \*/

for(i = 0; i < source->size; i++)

destination->data[i] = source->data[i];

destination->size = source->size;

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Heap-based Priority Queue Implementation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*/

/\* internal function prototypes \*/

int \_smallerIndexHeap(DynArr \*heap, int i, int j);

void \_adjustHeap(DynArr \*heap, int max, int pos);

/\* Get the index of the smaller node between two nodes in a heap

param: heap pointer to the heap

param: i index of one node

param: j index of other node

pre: i < size and j < size

ret: the index of the smaller node

\*/

int \_smallerIndexHeap(DynArr \*heap, int i, int j)

{

assert( heap != NULL );

assert( i < sizeDynArr( heap ) );

assert( j < sizeDynArr( heap ) );

int sml\_idx = -1;

if( compare( getDynArr( heap, i ), getDynArr( heap, j ) ) == -1 )

{

sml\_idx = i;

}

else

{

sml\_idx = j;

}

return( sml\_idx );

}

/\* Get the first node, which has the min priority, from the heap

param: heap pointer to the heap

pre: heap is not empty

ret: value of first node

\*/

TYPE getMinHeap(DynArr \*heap)

{

assert(sizeDynArr(heap) > 0);

return getDynArr(heap, 0);

}

/\* Add a node to the heap

param: heap pointer to the heap

param: node node to be added to the heap

pre: heap is not null

post: node is added to the heap

\*/

void addHeap(DynArr \*heap, TYPE val)

{

{

int parent;

int pos = sizeDynArr(heap);

addDynArr(heap, val);

while(pos != 0)

{

parent = (pos-1)/2;

if(compare(getDynArr(heap, pos), getDynArr(heap, parent)) == -1)

{

swapDynArr(heap, parent, pos);

pos = parent;

} else return;

}

}

}

/\* Adjust heap to maintain heap property

param: heap pointer to the heap

param: max index to adjust up to (but not included)

param: pos position index where the adjustment starts

pre: max <= size

post: heap property is maintained for nodes from index pos to index max-1 (ie. up to, but not including max)

\*/

void \_adjustHeap(DynArr \*heap, int max, int pos)

{

assert( heap != NULL );

assert( max <= sizeDynArr( heap ) );

int left\_child = pos \* 2 + 1;

int right\_child = pos \* 2 + 2;

int sml\_child = -1;

if( right\_child < max )

{

/\* Check for a second child \*/

if( left\_child < max )

{

/\* Determine the smallest child \*/

sml\_child = \_smallerIndexHeap( heap, left\_child, right\_child );

}

else

{

/\* Only one child, so smallest by default \*/

sml\_child = right\_child;

}

if( compare( getDynArr( heap, sml\_child ), getDynArr( heap, pos ) ) == -1 )

{

swapDynArr( heap, pos, sml\_child );

}

\_adjustHeap( heap, max, sml\_child );

}

/\* Only one child \*/

else if( left\_child < max )

{

if( compare( getDynArr( heap, left\_child ), getDynArr( heap, pos ) ) == -1 )

{

swapDynArr( heap, pos, left\_child );

}

\_adjustHeap( heap, max, left\_child );

}

}

/\* Remove the first node, which has the min priority, from the heap

param: heap pointer to the heap

pre: heap is not empty

post: the first node is removed from the heap

\*/

void removeMinHeap(DynArr \*heap)

{

assert( sizeDynArr( heap ) > 0 );

int last = sizeDynArr(heap)-1;

if (last != 0) /\* Copy the last element to the first \*/

putDynArr(heap, 0, getDynArr(heap, last)); /\* position \*/

removeAtDynArr(heap, last); /\* Remove last element. \*/

\_adjustHeap(heap, last , 0);/\* Rebuild heap \*/ // last -1 aug 15

}

/\* builds a heap from an arbitrary dynArray

param: v dynamicArray

pre: v is not empty

post: v is a proper heap

\*/

void \_buildHeap(DynArr \*heap)

{

assert( heap != NULL );

assert( sizeDynArr( heap ) > 0 );

int max = sizeDynArr(heap); int i;

/\* Start at the last non-leaf node \*/

for( i = ( max / 2 ) - 1; i >= 0; --i )

{

\_adjustHeap( heap, sizeDynArr( heap ), i );

}

}

/\*

In-place sort of the heap

param: heap pointer to the heap

pre: heap is not empty

post: the dynArr is in reverse sorted order

\*/

void sortHeap(DynArr \*heap)

{

assert( heap != NULL );

assert( sizeDynArr( heap ) > 0 );

int max = sizeDynArr(heap);

int i;

\_buildHeap( heap );

for( i = ( max - 1 ); i > 0; --i )

{

/\* Swap the outer two elements in the heap, then adjust it \*/

swapDynArr( heap, i, 0 );

\_adjustHeap( heap, i, 0 );

}

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/\* **dynArr.h : Dynamic Array Header declaration**. \*/

#ifndef DYNAMIC\_ARRAY\_INCLUDED

#define DYNAMIC\_ARRAY\_INCLUDED 1

#include "type.h"

# ifndef TYPE

# define TYPE void\*

# endif

/\* function used to compare two TYPE values to each other, define this in your compare.c file \*/

int compare(TYPE left, TYPE right);

/\* function used to print TYPE values, define this in your compare.c file \*/

void print\_type(TYPE curval);

typedef struct DynArr DynArr;

struct bag;

/\* Dynamic Array Functions \*/

DynArr \*createDynArr(int cap);

void deleteDynArr(DynArr \*v);

int sizeDynArr(DynArr \*v);

void addDynArr(DynArr \*v, TYPE val);

TYPE getDynArr(DynArr \*v, int pos);

void putDynArr(DynArr \*v, int pos, TYPE val);

void swapDynArr(DynArr \*v, int i, int j);

void removeAtDynArr(DynArr \*v, int idx);

/\* Stack interface. \*/

int isEmptyDynArr(DynArr \*v);

void pushDynArr(DynArr \*v, TYPE val);

TYPE topDynArr(DynArr \*v);

void popDynArr(DynArr \*v);

/\* Bag Interface \*/

int containsDynArr(DynArr \*v, TYPE val);

void removeDynArr(DynArr \*v, TYPE val);

/\* Heap-based Priority Queue Interface \*/

TYPE getMinHeap(DynArr \*heap);

void addHeap(DynArr \*heap, TYPE node);

void removeMinHeap(DynArr \*heap);

void sortHeap(DynArr \*heap);

/\* Utility function\*/

void copyDynArr(DynArr \*source, DynArr \*destination);

#endif

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

/\* **toDoList.c: List implementation**. \*/

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <string.h>

#include "toDoList.h"

/\* Create a task from the description and the priority

param: priority priority of the task

param: desc pointer to the description string

pre: none

post: none

ret: a task with description and priority

\*/

Task\* createTask (int priority, char \*desc)

{

char tmpdesc[128];

char \*p = &desc[0];

strncpy(tmpdesc,p,sizeof(tmpdesc));

Task\* task;

task = malloc(sizeof( Task));

strncpy(task->description,p,sizeof(task->description));

task->priority = priority;

return task;

}

/\* Save the list to a file

param: heap pointer to the list

param: filePtr pointer to the file to which the list is saved

pre: The list is not empty

post: The list is saved to the file in tab-delimited format.

Each line in the file stores a task, starting with the

task priority, followed by a tab character (\t), and

the task description.

The tasks are not necessarily stored in the file in

priority order.

\*/

void saveList(DynArr \*heap, FILE \*filePtr)

{

int i;

Task\* task;

assert(sizeDynArr(heap) > 0);

for(i = 0; i < sizeDynArr(heap); i++)

{

task = getDynArr(heap, i);

fprintf(filePtr, "%d\t%s\n", task->priority, task->description);

}

}

/\* Load the list from a file

param: heap pointer to the list

param: filePtr pointer to the file

pre: none

post: The tasks are retrieved from the file and are added to the list.

Refer to the saveList() function for the format of tasks in the file

\*/

void loadList(DynArr \*heap, FILE \*filePtr)

{

Task\* task;

char line[100]; /\* Assume lines < 100 \*/

char desc[TASK\_DESC\_SIZE], \*nlptr;

int priority;

/\* Read the priority first, then the description.

\* fgets() is used to read string with spaces

\*/

#ifdef NOTDEF

while (fscanf(filePtr, "%d\t", &priority) != EOF)

{

/\* fgets() stops reading at \n character \*/

fgets(desc, sizeof(desc), filePtr);

/\* remove trailing newline character \*/

nlptr = strchr(desc, '\n');

if (nlptr)

\*nlptr = '\0';

task = createTask(priority, desc);

addHeap(heap, task);

}

#endif

while(fgets(line, sizeof(line), filePtr) != 0)

{

sscanf(line, "%d\t%[^\n]", &priority, desc);

task = createTask(priority, desc);

addHeap(heap, task);

} /\* should use feof to make sure it found eof and not error\*/

}

/\* Print the list

param: heap pointer to the list

pre: the list is not empty

post: The tasks from the list are printed out in priority order.

The tasks are not removed from the list.

\*/

void printList(DynArr \*heap)

{

Task\* task;

DynArr \*temp;

task = (Task\*)heap;

temp = (DynArr\*)heap;

task = malloc(sizeof( Task));

temp = malloc(sizeof( Task));

assert(sizeDynArr(heap) > 0);

/\* copy the main list to a temp list

\* so that tasks can be printed out and removed.

\*/

copyDynArr(heap, temp);

while(sizeDynArr(temp) > 0)

{

/\* get the task \*/

task = getMinHeap(temp);

/\* print the task \*/

printf("%s\n\n", task->description);

/\* remove the task \*/

removeMinHeap(temp);

}

/\* free the temp list \*/

deleteList(temp);

}

/\* Delete the list

param: heap pointer to the list

post: The tasks from the list are removed and their occupied memories are freed

\*/

void deleteList(DynArr \*heap)

{

Task\* task;

while(sizeDynArr(heap) > 0)

{

/\* get the task \*/

task = getMinHeap(heap);

/\* remove the task \*/

removeMinHeap(heap);

/\* free the task \*/

free(task);

}

/\* free the heap \*/

deleteDynArr(heap);

}

/\* Compare two tasks by priority

param: left first task

param: right second task

pre: none

post: none

ret: -1 if priority of left < priority of right

1 if priority of left > priority of right

0 if priority of left = priority of right

\*/

/\*Define this function, type casting the value of void \* to the desired type.

The current definition of TYPE in dynamicArray.h is void\*, which means that left and

right are void pointers. To compare left and right, you should first cast

left and right to the corresponding pointer type (struct Task\*), and then

compare the values pointed by the casted pointers.

DO NOT compare the addresses pointed by left and right, i.e. "if (left < right)",

which is really wrong.

\*/

int compare(TYPE left, TYPE right)

{

Task \*l;

Task \*r;

l = malloc(sizeof(struct Task)); // added 15 aug

r = malloc(sizeof( struct Task));

l = (struct Task \*) left;

r = (struct Task \*) right;

/\* Comparing number part of the struct data \*/

if (l->priority < r->priority){

return -1;

} else if (l->priority > r->priority){

return 1;

} else {

return 0;

}

}

/\*Define this function, type casting the value of void \* to the desired type\*/

void print\_type(TYPE curval)

{

assert (curval != NULL);

Task\* data;

data = (Task\*)curval;

printf("%d\t%s\n", data->priority, data->description);

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

/\* **toDoList.h: List decleration**. \*/

#ifndef \_\_TODOLIST\_H

#define \_\_TODOLIST\_H

#include "dynamicArray.h"

#include "type.h"

Task\* createTask (int priority, char \*desc);

void saveList(DynArr \*heap, FILE \*filePtr);

void loadList(DynArr \*heap, FILE \*filePtr);

void printList(DynArr \*heap);

void deleteList(DynArr \*heap);

#endif

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/\*  **type.h** ; You can modify the structure to store whatever you'd like in your dynamic array.

We define struct Task for the to-do list application in this example \*/

#ifndef \_\_TYPE\_H

#define \_\_TYPE\_H

#define TASK\_DESC\_SIZE 128

struct Task {

char description[TASK\_DESC\_SIZE]; /\* description of the task \*/

int priority; /\* task priority \*/

};

typedef struct Task Task;

#endif

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

/\*  **main.c \*/**

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <string.h>

#include "toDoList.h"

int main (int argc, const char \* argv[])

{

Task\* newTask;

Task\* firstTask;

char desc[TASK\_DESC\_SIZE], filename[50], \*nlptr;

int priority;

char cmd = ' ';

FILE \*filePointer;

DynArr\* mainList = createDynArr(10);

printf("\n\n\*\* TO-DO LIST APPLICATION \*\*\n\n");

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"

);

/\* get input command (from the keyboard) \*/

cmd = getchar();

/\* clear the trailing newline character \*/

while (getchar() != '\n');

switch (cmd)

{

case 'a': /\* add new task \*/

printf("Please enter the task description: ");

/\* get task description from user input (from keyboard) \*/

if (fgets(desc, sizeof(desc), stdin) != NULL)

{

/\* remove trailing newline character \*/

nlptr = strchr(desc, '\n');

if (nlptr)

\*nlptr = '\0';

}

/\* get task priority from user input (from keyboard) \*/

do {

printf("Please enter the task priority (0-999): ");

scanf("%d", &priority);

} while(!(priority >= 0 && priority <= 999));

/\* clear the trailing newline character \*/

while (getchar() != '\n');

/\* create task and add the task to the heap \*/

newTask = createTask(priority, desc);

addHeap(mainList, newTask);

printf("The task '%s' has been added to your to-do list.\n\n", desc);

break;

case 'g': /\* get the first task \*/

if (sizeDynArr(mainList) > 0)

{

firstTask = (Task\*)getMinHeap(mainList);

printf("Your first task is: %s\n\n", firstTask->description);

}

else

printf("Your to-do list is empty!\n\n");

break;

case 'r': /\* remove the first task \*/

if (sizeDynArr(mainList) > 0)

{

firstTask = (Task\*)getMinHeap(mainList);

removeMinHeap(mainList);

printf("Your first task '%s' has been removed from the list.\n\n", firstTask->description);

/\* need to free up memory occupied by this task \*/

free(firstTask);

}

else

printf("Your to-do list is empty!\n\n");

break;

case 'p': /\* print the list \*/

if (sizeDynArr(mainList) > 0)

{

printList(mainList);

}

else

printf("Your to-do list is empty!\n\n");

break;

case 's': /\* save the list to file \*/

if (sizeDynArr(mainList) > 0)

{

/\* get filename from user input (from keyboard) \*/

printf("Please enter the filename: ");

if (fgets(filename, sizeof(filename), stdin) != NULL)

{

/\* remove trailing newline character \*/

nlptr = strchr(filename, '\n');

if (nlptr)

\*nlptr = '\0';

}

/\* open the file \*/

filePointer = fopen(filename, "w");

if (filePointer == NULL) {

fprintf(stderr, "Cannot open %s\n", filename);

break;

}

/\* save the list to the file \*/

saveList(mainList, filePointer);

/\* close the file \*/

fclose(filePointer);

printf("The list has been saved into the file successfully.\n\n");

}

else

printf("Your to-do list is empty!\n\n");

break;

case 'l': /\* load the list from the file \*/

printf("Please enter the filename: ");

/\* get filename from user input (from keyboard) \*/

if (fgets(filename, sizeof(filename), stdin) != NULL)

{

/\* remove trailing newline character \*/

nlptr = strchr(filename, '\n');

if (nlptr)

\*nlptr = '\0';

}

/\* open the file \*/

filePointer = fopen(filename, "r");

if (filePointer == NULL) {

fprintf(stderr, "Cannot open %s\n", filename);

break;

}

/\* load the list from the file \*/

loadList(mainList, filePointer);

/\* close the file \*/

fclose(filePointer);

printf("The list has been loaded from file successfully.\n\n");

break;

case 'e': /\* exit the program \*/

printf("Bye!\n\n");

break;

default:

printf("What is your command anyway?\n\n" );

break;

}

}

while(cmd != 'e');

/\* delete the list \*/

deleteList(mainList);

return 0;

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++c

**/\* todo.txt; Use your ways to change this file \* /**

0 take a nap

1 study heap-based priority queue

101 review trees for Midterm 2

3 do assignment 5

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**OPTIONAL FILES**

**/\* main2.c, used for Sorting list \*/**

#include <stdio.h>

#include <stdlib.h>

#include <assert.h>

#include <string.h>

#include "dynamicArray.h"

#include "toDoList.h"

#define TESTSORT

int main(int argc, const char \* argv[])

{

Task \*task1, \*task2, \*task3, \*task4, \*task5, \*task6, \*task7, \*task8, \*task9, \*task10;

DynArr \*mainList;

int i;

mainList = createDynArr(10);

/\* create tasks \*/

task1 = createTask(9, "task 1");

task2 = createTask(3, "task 2");

task3 = createTask(2, "task 3");

task4 = createTask(4, "task 4");

task5 = createTask(5, "task 5");

task6 = createTask(7, "task 6");

task7 = createTask(8, "task 7");

task8 = createTask(6, "task 8");

task9 = createTask(1, "task 9");

task10 = createTask(0, "task 10");

/\* add tasks to the dynamic array \*/

addHeap(mainList, task1);

addHeap(mainList, task2);

addHeap(mainList, task3);

addHeap(mainList, task4);

addHeap(mainList, task5);

addHeap(mainList, task6);

addHeap(mainList, task7);

addHeap(mainList, task8);

addHeap(mainList, task9);

addHeap(mainList, task10);

#ifdef TESTHEAP

for(i = 0; i < sizeDynArr(mainList);i++)

printf("DynArr[%d] = %d\n", i, getDynArr(mainList,i).priority);

while(!isEmptyDynArr(mainList))

{

TYPE v;

v = getMinHeap(mainList);

printf("Val = %s \_\_\_%d\n", v->description, v->priority);

removeMinHeap(mainList);

}

#endif

#ifdef TESTSORT

printf("Before Sort Called \n");

for(i = 0; i < sizeDynArr(mainList);i++)

printf("DynArr[%d] = %d\n", i, ((Task\*)getDynArr(mainList,i))->priority);

/\* sort tasks \*/

sortHeap(mainList);

printf("After Sort Called \n");

/\* print sorted tasks from the dynamic array \*/

for(i = 0; i < sizeDynArr(mainList);i++)

printf("DynArr[%d] = %d\n", i, ((Task\*)getDynArr(mainList,i))->priority);

return 0;

#endif

}

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Makefile. Txt**

all: prog1 prog2

prog1: dynamicArray.o main.o toDoList.o

gcc -g -Wall -std=c99 -o prog1 dynamicArray.o toDoList.o main.o

prog2: dynamicArray.o main2.o toDoList.o

gcc -g -Wall -std=c99 -o prog2 dynamicArray.o toDoList.o main2.o

main.o: main.c dynamicArray.h type.h toDoList.h

gcc -g -Wall -std=c99 -c main.c

main2.o: main2.c dynamicArray.h type.h toDoList.h

gcc -g -Wall -std=c99 -c main2.c

dynamicArray.o: dynamicArray.c dynamicArray.h

gcc -g -Wall -std=c99 -c dynamicArray.c

toDoList.o: toDoList.c toDoList.h

gcc -g -Wall -std=c99 -c toDoList.c

clean:

rm dynamicArray.o

rm toDoList.o

rm main.o

rm main2.o

rm prog1

rm prog2

++++++++++++++++++++++++++++++++++++++++++++++++++++++