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

#include <assert.h>

#include <stdlib.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 \*/

};

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

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 cap elements

post: v->data is not null

\*/

// Creates the internal array to the struct DynArray

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.

\*/

// ENTIRE STRUCT DYNAMICALLY ALLOCATED

DynArr\* createDynArr(int cap)

{

assert(cap > 0);

DynArr \*r = (DynArr \*)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: none

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)

{

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: none

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

post: the memory used by d is freed

\*/

void deleteDynArr(DynArr \*v)

{

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

\*/

// Resize to the NEW CAP size argument

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

{

int tempSize = v -> size;

// 1) Create the new array using member function

assert(newCap > 0);

assert(v != 0);

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

assert(tempV != 0);

// 2) Move the values from current to new

int i;

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

{

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

}

// 3) Free old array but not the pointer itself

freeDynArr(v);

// 4) Point at the new array

v -> data = tempV;

// 5) Set the new cap and size

v -> capacity = newCap;

v -> size = tempSize;

}

/\* 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)

{

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

\*/

// Puts the new new element in size location

void addDynArr(DynArr \*v, TYPE val)

{

/\* 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

\*/

// The position is passed as the actual array pos so can be zero

TYPE getDynArr(DynArr \*v, int pos)

{

// Assert the preconditions

assert(v != 0 && v -> size > 0 && pos >= 0 && pos < v -> size);

// Return the required index if the asserts dont bomb out

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 the preconditions

assert(v != 0 && v -> size > 0 && pos >= 0 && pos < v -> size);

// Stick the new value into the location

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)

{

// Temp holder for the swap

TYPE tTemp;

// Assert the preconditions

assert(v != 0 && v -> size > 0 && i >= 0 && i < v -> size && j >= 0 && j < v -> size);

// Do the swapperooo

tTemp = v -> data[i];

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

v -> data[j] = tTemp;

}

/\* 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)

{

// Assert the preconditions

assert(v != 0 && v -> size > 0 && idx >= 0 && idx < v -> size);

// Bump all the values down one spot starting at the

// index one greater than the index being removed

int i;

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

{

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

}

// Dec the size

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: the dynArr is not null

post: none

ret: 1 if empty, otherwise 0

\*/

int isEmptyDynArr(DynArr \*v)

{

// Check if the pointer is null

assert(v != 0);

if(v -> size > 0)

return 0;

else

return 1;

}

/\* 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

\*/

// TOP of stack is index size - 1

void pushDynArr(DynArr \*v, TYPE val)

{

// Check if the pointer is null

assert(v != 0);

// This function already puts element on the top

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)

{

// Check if the pointer is null or empty

assert(v != 0 && isEmptyDynArr(v) == 0);

// Return the last item size - 1

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)

{

// Check if the pointer is null or empty

assert(v != 0 && isEmptyDynArr(v) == 0);

// Remove the top element

removeAtDynArr(v, v -> size - 1);

}

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

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)

{

// Index variable

int i;

// Check pre conditions with assertions

assert(v != 0 && isEmptyDynArr(v) == 0);

// Loop through and check

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

{

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

return 1;

}

// If it makes here the value is not present

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)

{

// Index variable

int i;

int iLocation = -99;

// Check pre conditions with assertions

assert(v != 0 && isEmptyDynArr(v) == 0);

// Loop and find the first occurrence

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

{

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

{

iLocation = i;

break;

}

}

// If location is greater than 0 then bump

// everything down

if(iLocation >= 0)

{

removeAtDynArr(v, iLocation);

}

}