**DYNAMIC ARRAY IMPLEMENTATION (With Stack & Bag functions)**

**/\*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 \*/

};

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

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 = 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.

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;

/\* Remember, init did not free the original data \*/

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);

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(EQ(v->data[i], val) )

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)); /\* Design decision: Error if they try to remove something not in there! \*/

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

if(EQ(v->data[i], val))

{

removeAtDynArr(v,i);

break;

}

}

/\* Utility function for debugging \*/

void \_printDynArr(struct DynArr \*da)

{

int i;

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

printf("DA[%d] == %d\n", i, da->data[i]);

}

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

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

#ifndef DYNAMIC\_ARRAY\_INCLUDED

#define DYNAMIC\_ARRAY\_INCLUDED 1

# ifndef TYPE

# define TYPE int

# define TYPE\_SIZE sizeof(int)

# endif

# ifndef LT

# define LT(A, B) ((A) < (B))

# endif

# ifndef EQ

# define EQ(A, B) ((A) == (B))

# endif

typedef struct DynArr DynArr;

/\* 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);

#endif

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

**/\* dynamicArrayMain.c: Dynamic Array MAIN implementation. \*/**

#include "dynamicArray.h"

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

/\* VISUAL STUDIO (VS) USERS: COMMENT OUT THE LINE BELOW

TO EXCLUDE THE MEMORY TEST CODE.

\*/

//#define MEMORY\_TEST\_INCLUDED

//#ifdef MEMORY\_TEST\_INCLUDED

/\* This header is needed for memory usage calculation. \*/

//#include <sys/resource.h>

/\* Function to get current memory usage in KB (Max Resident Set Size)

long getMemoryUsage() {

int who = RUSAGE\_SELF;

struct rusage usage;

int ret;

ret = getrusage(who, &usage);

return usage.ru\_maxrss;

}

#endif

\*/

/\*Function to get number of milliseconds elapsed since program started execution\*/

double getMilliseconds() {

return 1000.0 \* clock() / CLOCKS\_PER\_SEC;

}

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

DynArr\* b;

int n, i;

double t1, t2;

#ifdef MEMORY\_TEST\_INCLUDED

/\* variables to hold memory used before and after creating DynArr \*/

long m1, m2;

/\* memory used BEFORE creating DynArr \*/

m1 = getMemoryUsage();

#endif

if( argc != 2 ) return 0;

b = createDynArr(1000);

n = atoi(argv[1]); /\*number of elements to add\*/

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

addDynArr(b, (TYPE)i); /\*Add elements\*/

}

#ifdef MEMORY\_TEST\_INCLUDED

/\* memory used AFTER creating DynArr \*/

m2 = getMemoryUsage();

printf("Memory used by DynArr: %ld KB \n", m2-m1);

#endif

t1 = getMilliseconds();/\*Time before contains()\*/

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

containsDynArr(b, i);

}

t2 = getMilliseconds();/\*Time after contains()\*/

printf("Time for running contains() on %d elements: %g ms\n", n, t2-t1);

/\* delete DynArr \*/

deleteDynArr(b);

return 0;

}

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

**/ \*makefiledynamicArray.txt – Use if making use of manual linking \*/**

default: prog

dynamicArray.o: dynamicArray.c dynamicArray.h

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

prog: dynamicArray.o

gcc -Wall -std=c99 -o prog dynamicArray.o dynamicArrayMain.c

clean:

rm dynamicArray.o

cleanall: clean

rm prog

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