BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
SECOND SEMESTER 2013-2014
Data Structures and Algorithms

PART-I (Implementing a dynamically sized array in C)

We are going to see a quick implementation of a dynamically sized array in C.

Basic C arrays- Here's a quick refresher on what a basic C array looks like, allocated on the stack:

```
int main() {
  // Declare an array of 3000 integers
  int my_array[3000];
}
```

This bit of code does the following:

- **Allocates memory on the stack1**. Specifically it allocates 3000 * sizeof(int) bytes on the stack frame of the main function call. Memory allocated in this way is automatically freed when execution reaches the end of the current block (in this case, the end of main).
- **Makes a pointer to the allocated memory available in the variable my_array**. We can index into my_array with subscripts e.g. my_array[271] would give us the 272nd element. This by the way, would do exactly the same thing as *(my_array + 271).

There's no easy way to automatically resize an array as its contents expand. You can call realloc on an array you've allocated on the heap to make it larger (more on this later) but ideally the resizing of the array would be automatic.

You can index out of array bounds. There is no bounds checking in C arrays. In other words, there's nothing stopping you from reading a value out of my_array[5000]. Since all this code does is read the memory 5000 elements [**By elements we mean the size of the type of the array. If an array holds elements of type int which each occupy four bytes, then integer_array[5000] means the location in memory of integer_array offset by (5000 * 4) bytes**.] after the location of the pointer my_array, indexing out of an arrays bounds will simply retrieve some as-yet unallocated location in virtual memory. This is most likely not what you intended, and can have potentially dangerous consequences.

When we can afford a few extra clock cycles and the requisite space, it would be nice to abstract these problems away in some sort of underlying data structure. A dynamic array a.k.a a vector does exactly that.

A vector doesn't solve all of the problems that we have when dealing with collections. It's quite good for lists which are append-only,

but if insertion/deletion is a common operation then it's possible that a linked list is a better choice.

Defining a Vector interface

In this example we're just going to create a dynamically sized array of integers. Here's what the definition of a vector interface might look like:

```
// vector.h

#define VECTOR_INITIAL_CAPACITY 100

// Define a vector type
typedef struct {
  int size;      // slots used so far
  int capacity;  // total available slots
  int *data;     // array of integers we're storing
} Vector;

void vector_init(Vector *vector);
void vector_free(Vector *vector);
int  vector_get(Vector *vector, int index);
void vector_append(Vector *vector, int value);
void vector_double_capacity_if_full(Vector *vector);
void vector_set(Vector *vector, int index, int value);
```

Description:

- ✓ **#define VECTOR_INITIAL_CAPACITY 100** is a constant we'll use later to set a starting capacity of our vectors internal array. 100 is an arbitrary value, and if we were writing this for real we might think a little harder about what an optimal initial capacity might be.
- ✓ **typedef struct { ... } Vector**; is the definition of our Vector type.
- ✓ **size** is the current size of the vector,
- ✓ **capacity** is the total size of the underlying array
- ✓ **data** is a pointer to the first element of the underlying array itself.
- ✓ **vector_init()** is a function that initializes a vector struct. It sets size to 0, capacity to VECTOR_INITIAL_CAPACITY and allocates an appropriate amount of memory (vector->capacity * sizeof(int)) for the underlying data array.
- ✓ **vector_append()** appends the given value to the vector. If the underlying data array is full, then calling this function should cause vector->data to expand to accept this value. Increments vector->size.
- ✓ **vector_get()** returns a value out of a vector at the given index. If the index is below 0 or greater than vector->size - 1, this function should complain about the index being out of bounds.
- ✓ **vector_set()** sets the value at the given index to the given value. If the index is greater than the vector->size, this function should expand the vector until it is big enough to contain the

index and set the value at that index. It should zero-fill all values in between. vector->size should be incremented accordingly.

- ✓ **vector_double_capacity_if_full()** doubles the underlying data array capacity if **vector->size >= vector->capacity**. We'll find out later that changing the size of the array is expensive, so in order to minimize the number of times we need to resize, we double the capacity each time.
- ✓ **vector_free()** frees the memory allocated for the data array. We leave freeing of the Vector struct itself to client code (so they can use any sort of pointer they like, be it stack or heap, and then clean up after themselves).

Implementing the Vector

Here's what an implementation of the interface we defined above might look like:

```
// vector.c

#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

void vector_init(Vector *vector) {
  // initialize size and capacity
  vector->size = 0;
  vector->capacity = VECTOR_INITIAL_CAPACITY;

  // allocate memory for vector->data
  vector->data = malloc(sizeof(int) * vector->capacity);
}

void vector_append(Vector *vector, int value) {
  // make sure there's room to expand into
  vector_double_capacity_if_full(vector);

  // append the value and increment vector->size
  vector->data[vector->size++] = value;
}

int vector_get(Vector *vector, int index) {
  if (index >= vector->size || index < 0) {
    printf("Index %d out of bounds for vector of size %d\n",
                                          index, vector->size);
    exit(1);
  }
  return vector->data[index];
}



void vector_set(Vector *vector, int index, int value) {
  // zero fill the vector up to the desired index
  while (index >= vector->size) {
```

```
    vector_append(vector, 0);
  }

  // set the value at the desired index
  vector->data[index] = value;
}

void vector_double_capacity_if_full(Vector *vector) {
  if (vector->size >= vector->capacity) {
    // double vector->capacity and resize the
    // allocated memory accordingly
    vector->capacity *= 2;
    vector->data =
           realloc(vector->data, sizeof(int) * vector->capacity);
  }
}

void vector_free(Vector *vector) {
  free(vector->data);
}
```

Using our Vector

In this usage example, we keep things simple and just pass around the pointer to a variable called vector allocated on the stack:

```
// vector-usage.c

#include <stdio.h>
#include "vector.h"

int main() {
  // declare and initialize a new vector
  Vector vector;
  vector_init(&vector);

  // fill it up with 150 arbitrary values
  // this should expand capacity up to 200
  int i;
  for (i = 200; i > -50; i--) {  vector_append(&vector, i); }

  // set a value at an arbitrary index
  // this will expand and zero-fill the vector to fit
  vector_set(&vector, 4452, 21312984);

  // print out an arbitrary value in the vector
  printf("Heres the value at 27: %d\n", vector_get(&vector, 27));

  // we're all done playing with our vector,
  // so free its underlying data array
  vector_free(&vector);
}
```

Tradeoffs

Working at this low a level helps to highlight a number of tradeoffs that we don't necessarily see when coding at break-neck speed in high-level languages like Ruby. For example:

Resizing the vector (specifically, calling realloc()) might be expensive. (From the man page)

The realloc() function tries to change the size of the allocation pointed to by ptr to size, and returns ptr. If there is not enough room to enlarge the memory allocation pointed to by ptr, realloc() creates a new allocation, copies as much of the old data pointed to by ptr as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory.

If we ever find ourselves in the situation where there's no room for the data array to expand into, we may be hit with the expensive copy operation. In order to avoid excessive calls to realloc then, we double the capacity of the array each time we expand it. This strategy has the downside of potentially allocating space we never use, so there's a trade-off to be made between wasted space and performance overhead.

Also, **Vector only ever holds integers**. If we were to hold an array of void pointers instead of integers, our vector could be used with values of arbitrary types. This would of course incur the overhead of dereferencing a pointer every time we wanted to read a value out of the array, so there's a trade-off there between flexibility and an (albeit small) performance hit.

******* End of Part-I *******

Important Programming Instructions for PART-II

1. All inputs to the program must be either (a) command line arguments (b) or read from a file (other than stdin). DO NOT READ anything from stdin and DO NOT USE PROMPTS like “Please enter a number …”
2. You are required to write the output to a file (other than stdout) and errors if any to a different output channel (stderr or another file)
3. Use standard C coding conventions for multi-file programs. Separate the following: interfaces of functions (use a “.h” file), data type definitions (use another “.h” file), ADT / algorithm implementation (use a “.c” file), and driver/test code (use another “.c” code)
4. You should write a makefile for your lab implementation.

PART-II Objective

A phone book in a mobile phone is always kept sorted name wise, where in you cannot have duplicate names. A typical record in a phone book

contains information like name, mobile number, email, date of birth etc.

You have to implement the phone book keeping in mind that the maximum size of the phone book is limited (you can assume the maximum size). Memory is to be allocated for each new contact element dynamically.

Description of the type definitions

1. Define an **ADT PhoneBook** and implement the same in C. Use a circular linked list for implementation.
2. Create a type definition for a **Contact** element containing name and mobile number. Note that a node in a list should also contain a pointer to the next node in the list. In case of a circular linked list the pointer of the last node points to the head/first node.
3. Create a type definition **PHBOOK** as a pointer to the head/first node of the PhoneBook.

(10 Min)

Description of operations supported _

1. createAddressBook()

Description : Creates an empty phone book
Parameters : None
Return : Pointer to the start of address book
Pre-conditions : None
Post-conditions : None

(10 Min)

2. isNotFull()

Description : Check for enough room in the phone book
Parameters : pointer to the phone book
Return : -1 if the phone book is full
Number of contact elements otherwise
Pre-conditions : The phone book is created
Post-conditions : None

(10 Min)

3. isEmpty()

Description : Checks if the phone book is not empty
Parameters : pointer to the phone book
Return : -1 if the phone book is empty
Number of contact elements otherwise
Pre-conditions : The phone book is created
Post-conditions : None

(10 Min)

4. printAddressBook()

Description : Prints the existing phone book to the given output file
Parameters : pointer to the phone book and a FILE pointer
Return : void

Pre-conditions : The phone book is created & is not empty
Post-conditions : None

(15 Min)

5. searchByName()
Description : Given a name search for a contact
Parameters : pointer to the phone book, contact to be searched for
Return : pointer to the contact element, if found NULL otherwise
Pre-conditions : The phone book is created & is not empty
Post-conditions : None

(15 Min)

6. insertAddress()
Description : Insert a new contact in the phone book
Parameters : pointer to the first node of the phone book, contact element to be inserted
Return : pointer to the first node of the phone book
Pre-conditions :

- The phone book is created
- If the contact doesn't exist but the phone book is full delete the next contact and add this contact.

Post-conditions :

- The phone book is sorted
- The phone book is maintained as a circular list

(30 Min)

7. deleteAddress ()
Description : Delete an existing contact
Parameters : pointer to the first node of the phone book, contact element to be deleted
Return : pointer to the first node of the phone book
Pre-conditions : The phone book is created & is not empty
Post-conditions :

- The phone book is sorted
- The phone book is maintained as a circular list

(10 Min)

8. deleteAllAddresses()
Description : Delete entire phone book
Parameters : pointer to the first node of the phone book
Return : A pointer to an empty phone book
Pre-conditions : The phone book is created & is not empty
Post-conditions : All contact elements are deleted

(10 Min)

The following procedure should go in the driver file:

9. uploadAddresses()
Description : Creates the phone book by reading the contact elements from a text file (filename is a parameter to this procedure, and should be a command line argument)

Parameters : Pointer to the phone book, the name of a file
Return : Pointer to the phone book (into which contacts have been added)
Pre-conditions : Text file contains zero or more contacts separated by an empty line. Each contact contains a name, and a (mobile) phone number
Post-condition :

- The returned phone book includes all the contact entries in the file.
- The phone book is maintained as a circular list

(30 Min)