BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
CS-F211: Data Structures and Algorithms
Lab 3: Heap Memory Estimation, Stacks and Queues

Introduction
Welcome to Week 3 of Data Structures and Algorithms! Over the last two weeks, we brushed up on our C programming and are now (hopefully) comfortable with pointers, structs, file I/O, and working on multi-file projects. We also looked at time measurement using the <sys/time.h> library. This week, we will get started with data structures.

Objectives
- Efficiency Estimation (Heap Space Measurement)
- Abstract Data Types (ADTs)
    - The Stack ADT
    - The Queue ADT

Efficiency Estimation (Heap Space Measurement)
While writing programs, we have two major resources - time and space. The space-time tradeoff is a common theme while dealing with performance optimisation. Oftentimes, improving the performance of one comes at the cost of the other. Typically, you can speed up algorithms by *remembering* more, i.e., by using more memory. If you have memory constraints, you can repeat computations, which slows down your program. Last week, we concluded by looking at how can we measure performance in terms of time taken by a program using the gettimeofday() function from the <sys/time.h> library. Now, we would be looking at a similar process of how we can compare different algorithms based on their memory requirements.

For now, we would assume that the bulk of our memory usage would be on the heap. This is a fair assumption to make when dealing with dynamic data structures. We can always convert significant stack usage into heap usage (replacing static arrays with dynamically allocated arrays).

While there are specialised tools that can monitor space usage like *valgrind*, we would be using a simple technique where we would add a wrapper around our memory management functions (malloc() and free()). Whenever this new version malloc() (say myalloc()) is called requesting a certain number of bytes (say n) of memory, myalloc() would now internally request (sizeof(int) + n) bytes of memory (using malloc()). Subsequently, myalloc() would store the integer n at the beginning of the block and return the pointer starting from just after the integer holding the size. It would also update a global variable holding the total size allocated.

Suppose we want to allocate 5 bytes to hold the character array {'h', 'e', 'l', 'l', 'o'}. When we use malloc, our call would have been:

```
char *p = malloc(5);
```

This would have allocated 5 bytes and returned a pointer to the beginning of this block, which can then be populated with the characters {'h', 'e', 'l', 'l', 'o'}.

Post this, the resulting memory layout is illustrated in Fig 1. A block of 5 bytes contains the required array. The arrow points to the location returned by malloc().

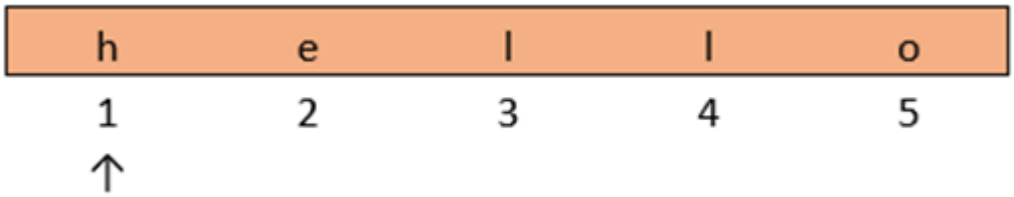


Fig 1: Memory layout when malloc(5) is used to store the character array

Now instead, if we would have used myalloc(5), the total memory allocated would be sizeof(int) (ie 4) + the requested size (ie 5) = 9 bytes of memory. The corresponding layout is illustrated in Fig 2.

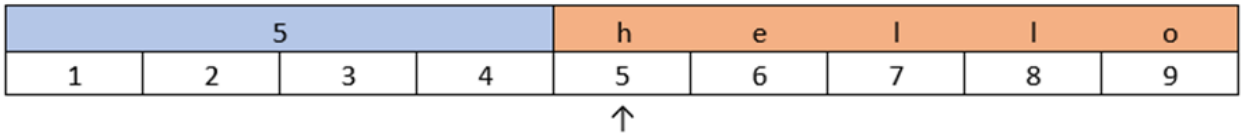


Fig 2: Memory layout when myalloc(5) is used to store the character array

As seen here, the memory requested is for 5+4 = 9 bytes. The first 4 bytes hold the integer 5. The block of size 5 starting after the first four is returned by the function. In this way, the calling function sees no change in the returned pointer, but in memory the size information has been stored. So now when this pointer would be passed to wrapper version of free() (say myfree()), it would look for an integer stored just before the start of the block pointed to by the pointer (by decrementing the void pointer by sizeof(int) (ie 4 bytes) and dereferencing it as an int called size), decrement this integer size from the global variable maintaining the total allocated size, and then free the entire block including the integer.

Let us see both these new wrapper functions in more detail. As we saw the last week, the signature for malloc() is:

```
void *malloc(size_t size);
```

Here, size_t is just an alias for unsigned int defined in the <stdlib.h> header file. Thus, the parameter for malloc() is the integer number of bytes that need to be allocated. It returns a ***void pointer*** to the block of the required size starting at that address. A **void pointer** is a pointer that has no associated data type. A void pointer can hold the address of any data type and can be typecasted to any other pointer type. However, since a void pointer does not know the size or type of the data it points to, it **cannot be dereferenced directly**. Therefore, a void pointer must be **explicitly cast to a specific pointer type** before dereferencing.
For example, if an integer value is stored at the memory location pointed to by a void pointer ptr, it can be accessed as follows:

```
*((int *)(ptr)) = 42;
```

or

```
int x = *((int *)(ptr));
```

Void pointers are primarily used to achieve **type-independent memory management**. They are commonly used in **dynamic memory allocation functions** such as malloc(), which return a void pointer because the allocated memory does not have an inherent data type. The programmer later assigns the appropriate type through casting. Void pointers are also widely used in **generic programming**, such as standard library functions like qsort() and bsearch(), and in **generic data structures** (linked lists, stacks, queues) that are designed to store and manipulate data of multiple types.

We would be ensuring that our wrappers (myalloc() and myfree()) have identical parameters and return types as malloc() and free() so that the user can replace all instances of malloc() and free() with their corresponding wrappers without worrying about the data types.

The code for an example using this technique is given below:

```
#include <stdio.h>
#include<stdlib.h>
```

```c
size_t heapMemoryAllocated = 0;
#define ADDITIONAL_MEMORY sizeof(int)

void *myalloc(size_t size)
{
    void *ptr = malloc(size + ADDITIONAL_MEMORY);
    if(ptr == NULL)
        return NULL;
    heapMemoryAllocated += size;
    *((int *)ptr) = size;
    return ptr + ADDITIONAL_MEMORY;
}

void myfree(void *ptr)
{
    int size = *((int *)(ptr - ADDITIONAL_MEMORY));
    heapMemoryAllocated -= size;
    free(ptr - ADDITIONAL_MEMORY);
}

int main()
{
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);

    int *ptr = myalloc(sizeof(int));
    *ptr = 10;
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);
    myfree(ptr);
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);

    float *ptr2 = myalloc(sizeof(float));
    *ptr2 = 10.0;
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);

    char *ptr3 = myalloc(sizeof(char));
    *ptr3 = 'e';
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);
    myfree(ptr3);
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);

    double *arr;
```

```
    arr = myalloc(sizeof(double) * 40);

    printf("Heap memory allocated: %d\n", heapMemoryAllocated);

    for (int i = 0; i < 40; i++)
    {
        arr[i] = i * 3.14;
    }
    myfree(arr);
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);

    myfree(ptr2);
    printf("Heap memory allocated: %d\n", heapMemoryAllocated);

    return 0;
}
```

Here, if you observe myalloc(), its argument is the integer number of bytes requested. It first requests malloc() to allocate ADDITIONAL_MEMORY (here sizeof(int)) plus the requested number of bytes. If the allocation was successful (ptr != NULL), the global variable heapMemoryAllocated is updated. Now, we wish to store the number of bytes requested for allocation, in the first ADDITIONAL_MEMORY bytes of the allocated block. For this, we simply cast the pointer (which is currently of type void *) to int * and store the size there. Now, the block starting at the end of this int value is returned to the user by:

```
return ptr + ADDITIONAL_MEMORY;
```

It is important to note that with void pointers when we perform ptr + ADDITIONAL_MEMORY, the pointer arithmetic directly returns the address ADDITIONAL_MEMORY bytes ahead of ptr (as against an int pointer which would have been sizeof(int) * ADDITIONAL_MEMORY bytes ahead. ie sizeof(void) is taken to be 1 in this computation).

Since the signature is identical to that of malloc(), the user can replace all malloc() calls with myalloc() calls without making any other changes in their code. myfree() works in a similar manner. You can find an implementation of myfree()in the above code. Thus, all heap memory management operations can be performed through this pair of functions.

Thus, using this method we can keep track of the heap space used by the program in a global variable heapMemoryAllocated, with negligible overhead.

Abstract Data Types (ADTs)

An abstract data type is a theoretical specification of a data structure and its operations. It is a powerful idea that allows us to separate how we use the data structure from the particular form of the data structure. An ADT is defined by just its operations. These operations are made available to the client, who can use them irrespective of how it is internally implemented.

Students familiar with Object Oriented Programming in Java might be reminded of interfaces, and for a good reason, as they are one way of implementing ADTs. We would be implementing ADTs in C using header files to specify the interface.

Formally, an ADT is a mathematically specified entity that defines a set of its instances, with:
1. **A specific interface**: This is a collection of ***signatures*** (or ***function declarations in C***) of operations that can be invoked on an instance. This might be provided as an interface in Java or a header file in C.
2. A set of **axioms** (**pre-conditions** and **post-conditions**) that define the semantics of the operations (i.e., what the operations do to instances of the ADT, but not how). These pre- and post-conditions would be typically expressed in some form of predicate logic expressions (don't worry if you haven't studied them).

This interface is provided to the client who can make use of it without worrying about the exact implementation. Thus, the way the data is represented and operations are implemented does not matter for an ADT. A single ADT (as we shall soon see) can have multiple implementations and each implementation may differ in its performance.

ADTs allow us to separate the issues of *correctness* and *efficiency.* As long as the set of axioms is satisfied by the implementation and the function calls are made as per the signatures provided in the ADT, correctness is ensured. And performance can be optimized by modifying the implementation of the ADT independently without worrying about the correctness being compromised. As long as the pre- and post-conditions continue to be met, modifying the implementation won't affect the correctness.

Let's understand ADTs better through some concrete examples.

The Stack ADT

A stack is a **Last In First Out (LIFO)** structure. We insert and remove elements from the same end (called the ***top***) of a stack. It is analogous to a stack of cleaned dishes in a kitchen. As the new

dish is cleaned, it is added to the top of the stack. Whenever someone needs a clean dish, they would pick it from the top.

In the stack data structure, the operation of inserting an element at the top is called ***push*ing** the element to the stack, and the operation of removing the element from the top is called ***pop*ping** the element off of the stack. These operations are illustrated in Fig 3.

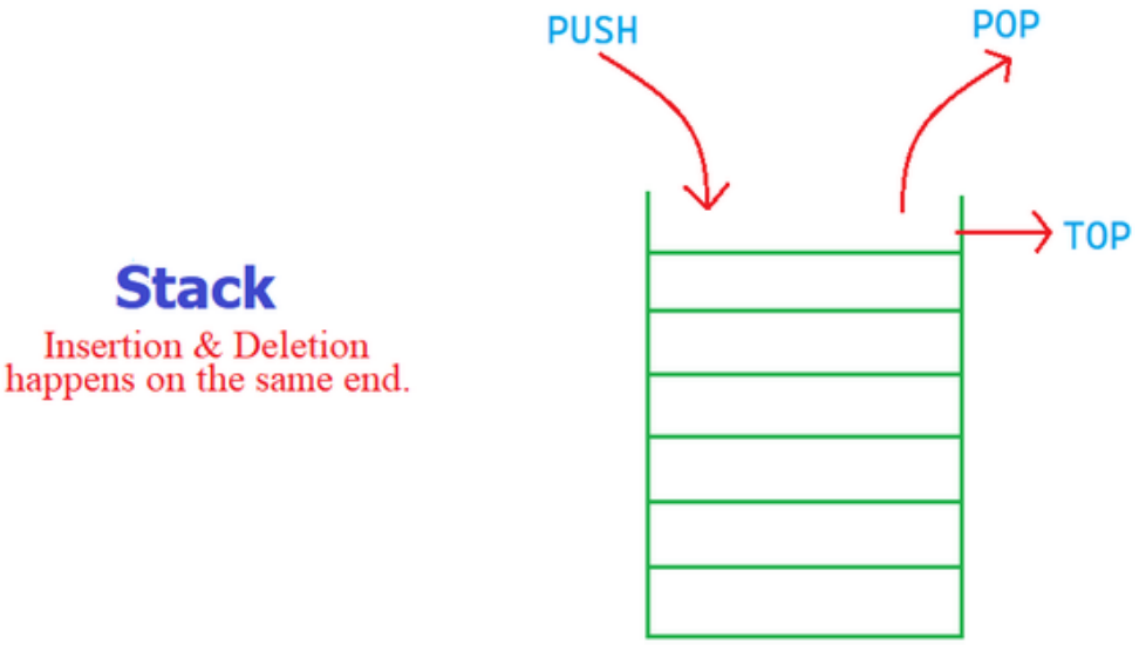


Fig 3: Illustration of stack operations.

One common application where one might choose to make use of a stack is when you might need to provide the user of your application with an *undo* facility (like the one provided by your favourite code editor). Here, you can store the history of changes made in a stack, pushing a new change onto the stack, and whenever the user wants to undo it, you can pop the last change off the stack and revert it.Let us consider a stack that would contain members of a user-defined type 'Element'. This data structure can now be represented as an ADT as follows:

Behaviour (Interface)

The following methods are included in the Stack ADT:

- `Element top(Stack s)` : Get the last element
- `Stack pop(Stack s)` : Remove the last element
- `Stack push(Stack s, Element e)` : Add a new element
- `Boolean isEmpty(Stack s)` : Find whether the list is empty

- `Stack newStack()` : Create a new stack

Properties (Axioms) (for unbounded Stack)

The following axioms must hold for the implementation to ensure correctness:

- `isEmpty(newStack()) == TRUE`
- `isEmpty(push(s,e)) == FALSE`
- `top(push(s,e)) == e`
- `pop(push(s,e)) == s`

Implementation

Let us begin implementing this ADT now. We would create a multi-file project for implementing our ADTs of this lab sheet. We have provided a few files to help you get started in the directory "Stack". Let us first create the structure for our custom data-type Element that would populate the stack in a new file named *element.h*. The file contents look like this:

```
#ifndef ELEMENT_H
#define ELEMENT_H
struct Element
{
    int int_value;
    float float_value;
};
typedef struct Element Element;
#endif
```

You would notice that our usual expected header file contents, namely the struct definition and the typedef statement, are wrapped in an #ifndef - #endif block. This is called an "include-guard". An include-guard has the syntax:

```
#ifndef <token>
#define <token>
header file contents…
#endif
```

This is used to prevent multiple inclusions of the file. Without include-guards, if the same header file gets included twice directly or indirectly in a file, it would lead to a compilation error. Include-guards prevent this multi-inclusion from happening. While this is not needed as such in our project as with a small number of files where we can manually keep track of which headers were

included in which file, it is a good practice to use them when dealing with multiple files. You can learn more about include-guards at https://en.wikipedia.org/wiki/Include_guard.

As we saw, some functions here require us to have a boolean datatype. While historically C did not have an explicit inbuilt boolean datatype, one was included in the header file <stdbool.h> in C99. By including this header file, we can directly use:

```
#include <stdbool.h>
bool isEmpty = true;
```

Alternatively, we can also define it using an enum as follows in a file called *bool.h*.

```
#ifndef BOOL_H
#define BOOL_H

typedef enum { false, true } bool;

#endif
```

Thus you now have a standard implementation of bool that can be used in any of our files by simply including this file. Since the members of the enum are in the order {false, true}, false would be implicitly equal to const int 0 and true would be equal to the const int 1, which is the convention followed in C.

Now, let us specify our method (or function) signatures of the stack ADT in a new file called *stack.h*.

```
#ifndef STACK_H
#define STACK_H

#include "element.h"
#include "bool.h"

typedef struct Stack Stack; // Stack is a alias to a struct stack

Stack *newStack();
// Returns a pointer to a new stack. Returns NULL if memory allocation fails

bool push(Stack *stack, Element element);
// Pushes element onto stack. Returns false if memory allocation fails

Element *top(Stack *stack);
```

```
// Returns a pointer to the top element. Returns NULL if stack is empty

bool pop(Stack *stack);
// Pops the top element and returns true. Returns false if stack is empty

bool isEmpty(Stack stack);
// Returns true if stack is empty. Returns false otherwise

void freeStack(Stack *stack);
// Frees all memory associated with stack

#endif
```

You would notice that the method signatures are slightly different from those mentioned in the ADT above. This has been done for ease of implementation.

This file (*stack.h*) would be included and made available to the client. The client can look at the method signatures and decide which calls to make.

Now let us implement the actual ADT. The stack ADT can be implemented in many different ways. We would be looking at two ways here, namely, using an array and using a linked list.

Using Arrays:

Now we begin implementing the stack ADT using an array. As one would expect, with an array we run into similar issues to those that arrays had. Namely, we cannot have dynamic-sized stacks. Thus, we would declare a max size for this declaration. Our push function would return *false* when the stack is completely filled.
Let us start implementing the stack ADT using an array in a new file *stack_array.c*.
The file would start looking like this:

```
#include "element.h"
#include "stack.h"
#include <stdlib.h>

#define STACK_SIZE 1000
struct Stack
{
    int top;
    Element data[STACK_SIZE];
```

```
};

Stack *newStack()
{
    Stack *s = (Stack *)malloc(sizeof(Stack));
    if(s != NULL)
        s->top = -1;
    return s;
}

bool push(Stack *s, Element e)
{
    if(s->top == STACK_SIZE - 1)
        return false;
    s->data[++(s->top)] = e;
    return true;
}
```

Here, the stack struct has first been defined (which was typedef'ed in the file stack.h). The array implementation has an array member of size STACK_SIZE and an integer top representing the current top index. In the function newStack(), we instantiate a stack dynamically, initialise the top member to -1 signifying an empty stack and return a pointer to the instantiated stack. In push(), an Element e is pushed onto a Stack s. The stack is passed by reference. The top variable is incremented. In case the stack is full and we are unable to push the element, the function returns false. Else the function returns true.

Task 1: After pasting the above code in the file *stack_array.c*, complete the implementation by adding definitions for the functions pop(), isEmpty(), top() and freeStack() in a similar fashion to the same file. You are obviously required to follow the signatures provided in the file *stack.h*.

Now we are ready with the implementation of a stack using an array. We can test out the implementation using a driver. A driver has been provided to you in a file called *stackDriver.c*.

Now, you can test out your implementation using the driver code. You can use the following makefile for your convenience:

```
runStackWithArray: stackDriver.o stack_array.o
    gcc -o runStackWithArray stackDriver.o stack_array.o
    ./runStackWithArray
```

```
stackDriver.o: stackDriver.c stack.h
    gcc -c stackDriver.c

stack_array.o: stack_array.c stack.h
    gcc -c stack_array.c

clean:
    rm -f *.o runStackWithArray
```

In case some part of the above make file does not make sense, do revisit the previous lab sheet to reacquaint yourself with the same. Thus, you can now test it out with a single command: `make runStackWithArray`

Home Exercise 1: Modify the driver (stackDriver.c) to ensure that all properties (axioms) of the stack ADT are satisfied by your implementation. Now, test it out and handle the corner cases in your implementation if needed.

Using Linked Lists:

We saw above that the implementation using arrays had the limitation that the size of the stack had to be bounded explicitly in the implementation. Also, the memory usage is constant. It is often wasteful as an array of length STACK_SIZE is always allocated irrespective of the actual number of elements present. To avoid these problems, we will now implement the same stack ADT using a linked list. A linked list would allow us to have dynamic length arrays and our stack size would be bound only by the total heap memory available to the program and not an arbitrary value specified by the implementer.

First, we need an implementation of linked lists having elements of type *Element*. The linked list must implement the signatures provided to you in the *linked_list.h* header file.

Task 2: Go through the *linked_list.h* header file and implement the functions in a new file called ***linked_list.c***. As you would notice in the file, the linked_list and node structures have been defined.

```
#include "element.h"

struct node
{
    Element data;
    struct node *next;
};
```

```
typedef struct node node;
typedef node * NODE;

struct linked_list
{
    int count;
    NODE head;
    // NODE tail; // Not required for stack. Required for Queue

};
typedef struct linked_list linked_list;
typedef linked_list * LIST;
```

We have 'typedef'ed the types linked_list, node, LIST, NODE to refer to the types struct linked_list, node, pointer to struct linked_list, pointer to node respectively. You need to implement the following functions:

1) LIST createNewList(): This function allocates memory for a new list and returns a pointer to it. The list is empty and the count is set to 0.
2) NODE createNewNode(element data): This function allocates memory for a new node and returns a pointer to it. The next pointer is set to NULL and the data is set to the value passed in.
3) void insertNodeIntoList(NODE node, LIST list): This function inserts a node at the beginning of the list.
4) void removeFirstNode(LIST list): This function removes the first node from the list.
5) void insertNodeAtEnd(NODE node, LIST list): This function inserts a node at the end of the list. (Optional for stack, but would be required when we implement queue).

So, now we have a linked list, but we are no closer to creating a stack … or are we? You might have realised that we are almost done. We just need to now simulate the stack using the above-defined linked list. Let us try to understand how we can go about this.

A stack, as we know, is characterised by its two characteristic operations: push and pop. How can these be carried out on a linked list? The stack is a LIFO data structure. A push is an insert operation. So, it would have to use one of the insert methods of the linked list. In our current implementation of the linked list, we only have the head pointer. Thus, it would be more efficient to use the head of the list as the top of the stack as push and pop would correspond directly to insertion and deletion at the head and would run in O(1) time.

Now, similarly, the other methods of the stack ADT can also be simulated by the methods of the linked list.

Task 3: Create a new file *stack_ll.c*. Include the files *linked_list.h* and *stack.h* in this file. Implement the methods of *stack.h* using *linked_list.h*. Extend the Makefile provided in the section on the implementation of a stack using arrays to include the implementation of the stack using a linked list as well. You can use the same driver (*stackDriver.c*) to test this code.

Task 4: Modify both implementations to include performance analysis. Compare the **time taken** and **the heap space utilized** by both implementations.

You have been provided with two files *heap_usage.h* and *heap_usage.c*. Go through them and understand how they work. You can use these functions to implement heap space measurement. Replace all calls to malloc(), calloc(), realloc() and free() in *stack_array.c* and *stack_ll.c* with the corresponding calls to myalloc() or myfree()...

You have been provided with three input files, namely, *small.csv*, *medium.csv* and *large.csv*. These files have different numbers of rows of data consisting of students' BITSAT score and CGPA at the end of the first year. For each of these files, find the time taken and heap space used in populating the stack and then emptying the stack. You need to push all elements from this file and and then pop all of them them. Print the maximum space utilised and the total time taken in populating and emptying the stack for each input file. ***Ensure that you do not include file or I/O operations in the time measurement***. That means you would have to keep adding the measured time for the push and pop operations to a counter and output the total time at the end.

Consider the file *cgStackDriver.c*. Here the code for reading the file has been provided to you. As you would notice, the method described in lab sheets 1-2 of reading a CSV file using fgets() and strtok() has been used here. fgets() reads a line from the file and stores it in the line variable. fgets() returns NULL when it reaches the end of the file. This condition is used to terminate the while loop. strtok() splits the line into tokens separated by the comma and returns the first token when called for the first time with the string line as the first argument and the delimiter "," as the second. When called successively with NULL as the first argument, strtok returns the subsequent tokens.

Modify the file to achieve the functionality required in this task as described in the comment blocks.. A helper function iftoe() has been provided to instantiate the Element struct which can be used if needed. iftoe() is a function that creates an Element from an integer and a float. It takes two arguments, an integer and a float, and returns an Element. The integer value of the

Element is set to the integer argument and the float value of the Element is set to the float argument.

The input file is to be provided to this main as a **command line argument**. You can run it with different files and thus compare the performance.
For example: if the executable is cgStack.exe, it can be executed with the input file small.csv by running the following command:

./cgStack.exe small.csv

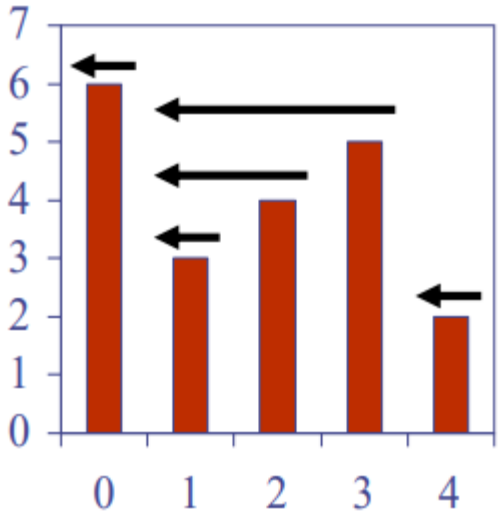
Also, find out what are the asymptotic complexities of the operations in your implementation theoretically. Observe whether your empirical results match the theoretical time and space complexities.

Home Exercise 2: Arithmetic expressions can have either infix, prefix or post-fix notation. We are used to the infix notation in general where the binary operator is present between the operands. For eg. "3+4", "1-(3-4*6)+4/2", etc. The same expressions in postfix notation would look like "3 4 +" and "1 3 4 6 * - - 4 2 / +" respectively. This may not look very intuitive to us but the postfix notation has certain advantages. One such advantage is that we would have needed parenthesis to specify the operator precedence overriding in infix notation, whereas it is implicit in the postfix notation. The postfix notation is also known as Reverse Polish Notation (RPN).

Your task is to write a program that would accept a string containing an RPN arithmetic expression as input, and print the result as the output. You can assume that the operators would be among +, -, * and /, your operands would be integer values, these would be separated by spaces (as given in the example below), and your result should be the floating-point result obtained on evaluating the expression. (Hint: You can use the stack you have implemented.)

For example: If the input is "1 3 4 6 * - - 4 2 / +", the result would be **24.0**.
If the input is "1 2 3 4 5 + * - -", the result would be **26.0**.

Home Exercise 3: Given an array X, the span S[i] of element X[i] is the maximum number of consecutive elements X[j] immediately preceding X[i] and such that X[j] <= X[i]. In other words, S[i] is the maximum k such that the elements X[i], X[i-1],..., X[i - k +1] are all less than or equal to X[i].



X	6	3	4	5	2
S	1	1	2	3	1

Fig 4. Example of spans. X is the input array and S is the spans array.

Consider the example in Fig 4. The span for element 5 is 3 since the elements 3, 4 and 5 are <= 5. You have been provided with the file incomplete file *computeSpan.c* in the "HW Exercises" directory. Complete the computeSpans() function using your implementation of stack such that the function has a complexity of O(n). [Hint: Look at what happens at the (S[i] - 1)th element.]

The Queue ADT

Queues differ from stacks in that queues follow the **first-in-first-out (FIFO)** principle. You can think of the queues of students during lunch hours at IC. As opposed to stacks, in a queue, insertion and deletion occur at different ends. The insertion (called the ***enqueue*** operation) is said to take place at the ***rear*** end of the queue, while the removal (called the ***dequeue*** operation) takes place at the ***front*** end of the queue. These operations are illustrated with an example in Fig 5.

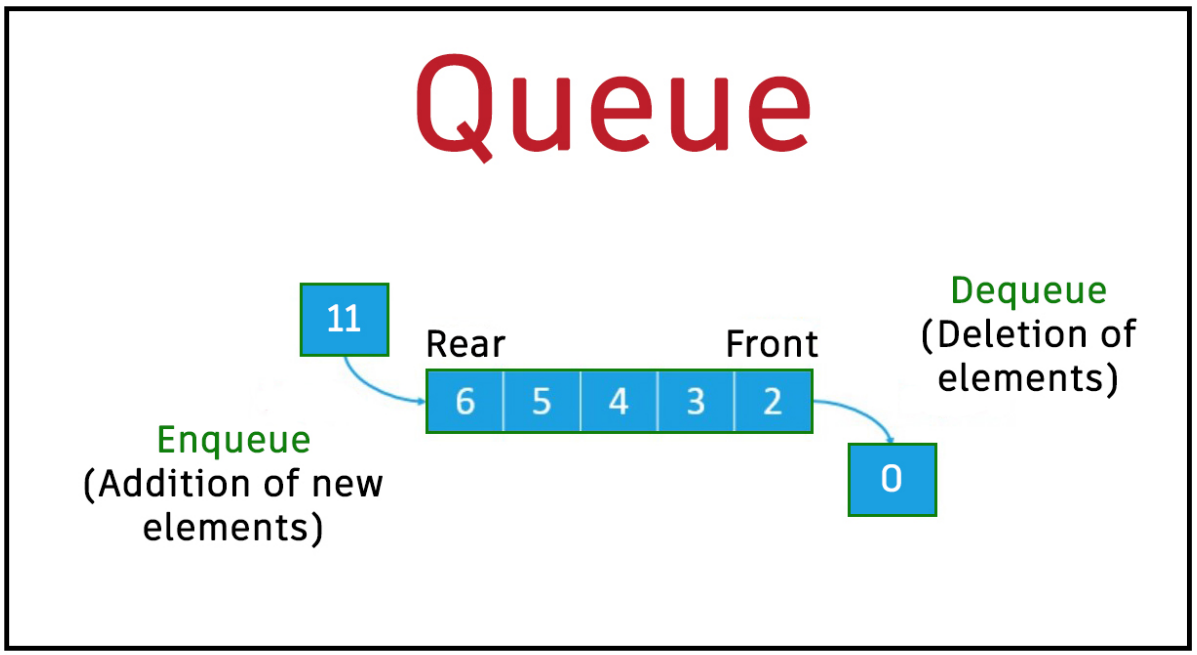


Fig 5. Illustration of Queue operations

Many scheduling algorithms make use of queues and their variations to ensure fairness and efficiency. For example, consider a workspace that has a shared printer between 10 users. It is possible that multiple users may submit print jobs simultaneously to the printer. If the printer services these jobs in an interleaved manner, it would result in garbage output. So to prevent this, the printer maintains a queue of jobs and services the jobs that arrive in a *FIFO* manner.

Let us now look at the Queue ADT.

Behaviour (Interface)

The following methods are included in the Queue ADT:

- `Queue createQueue()`: Create an empty queue
- `Queue enqueue(Queue q, Element o)`: Insert object **o** at the rear of the queue
- `Queue dequeue(Queue q)`: Remove the object from the front end of the queue; throw an error if queue is empty
- `Element front(Queue q)`: Return (and not remove) the element at the front end of the queue; throw an error if queue is empty
- `int size(Queue q)`: Return the number of elements in the queue
- `boolean empty(Queue q)`: Return **`TRUE`** if the queue is empty, **`FALSE`** otherwise

Properties (Axioms)

The following axioms must hold for the implementation to ensure correctness:

- `Front(Enqueue(createQueue(), v)) == v`
- `Dequeue(Enqueue(createQueue(), v)) == createQueue()`
- `Front(Enqueue(Enqueue(Q, w), v)) == Front(Enqueue(Q, w))`
- `Dequeue(Enqueue(Enqueue(Q, w), v)) == Enqueue(Dequeue(Enqueue(Q, w)), v)`

Implementation

As we did for the Stack ADT, we would begin implementing the queue ADT now. Again, we have provided a few files to help you get started in the directory "Queue". We would first be specifying the behaviour in a *queue.h* header file as follows:

```
#ifndef QUEUE_H
#define QUEUE_H

#include "element.h"
#include "bool.h"

typedef struct Queue Queue;

Queue *createQueue();
// createQueue() returns a pointer to a new Queue instance.

bool enqueue(Queue *queue, Element element);
// enqueue() returns true if the element was successfully added to the
// queue, false otherwise.

bool dequeue(Queue *queue);
// dequeue() returns true if the element was successfully removed from
// the queue, false otherwise.

Element *front(Queue *queue);
// front() returns a pointer to the element at the front of the queue,
// or NULL if the queue is empty.

int size(Queue *queue);
// size() returns the number of elements in the queue.

bool isEmpty(Queue *queue);
```

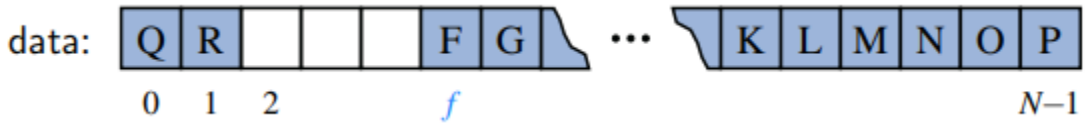
```
// isEmpty() returns true if the queue is empty, false otherwise.

void destroyQueue(Queue *queue);
// destroyQueue() frees all memory associated with the queue.
#endif
```

Again as we did in stacks, we have modified the method signatures here. As we pass the structure by reference, we can mutate the same structure; hence, our mutators need not return the modified structure. Instead, they now return a bool denoting whether the enqueue() or dequeue() operation was successful or not. Note that this is a design choice that we must always make – whether to have our functions return the modified structure, which then has to be assigned to the earlier structure, or to have them passed by reference and let the original structure itself be modified internally.

Using Arrays:

When we first think of using an array to simulate a queue, we might be tempted to simply start adding elements onto consecutive locations starting from the start of the array and dequeuing them from the front at the same time. We would require just an array and two integer variables for this implementation. The two variables would represent the front and rear of the queue respectively. As we enqueue() more elements onto the queue at the rear, the rear starts moving rightward (ie its index starts increasing). When we dequeue() elements, the front also starts moving rightward. However, this implementation has a drawback. The space freed up by a dequeued element cannot be reused for some other element.

This drawback can be resolved by using the array circularly. Here, we allow both the **front** and the **rear** to drift rightward, with the contents of the queue "wrapping around" the end of an array, as necessary. Assuming that the array has fixed length N, new elements are enqueued toward the "end" (rear pointer) of the current queue, progressing from the front to index N −1 and continuing at index 0, then 1. The following figure represents such a queue with first element F and last element R.



Use this logic now to implement the queue ADT. You can alternatively use **front** and **size** as the two variables as front, rear and size are linked to each other through the equation:

```
rear = (front + size - 1) % ARRAY_SIZE
```

Task 5: Implement the Queue ADT as specified by the signatures in the *queue.h* file in a new file called *queue_array.c*. Include the *heap_usage.h* file and make all the memory management calls through the wrapper functions. Test the correctness of your implementation using a driver similar to *stackDriver.c*.

Using Linked Lists:

As we did with the stack, queues can also be implemented using linked lists. However, unlike a stack, for the queue, insertions and deletions occur at different ends. So, with the linked list implementation that we currently have, we only have the head pointer (access to one end) in the linked list structure. So both enqueue() can dequeue() cannot be achieved in O(1). However, if we modify the linked list structure to also include a tail pointer, we would now have access to both ends of the list. Thus, we incurred the cost of marginal additional memory usage for achieving significantly better time complexity.

Task 6: Modify the linked list defined by in the *linked_list.h* file to include the tail pointer. Now also modify the function declarations in the *linked_list.c* file for maintaining the correct value at the tail pointer as you carry out different operations on the list.

You should have implementations (including the tail pointer) for all the functions defined in the linked_list.h file. Importantly, we should have the functionality to insert and remove elements from the head and insert elements from the tail. (Is it possible to have an O(1) function to remove the tail element? If yes, how? If not, why not?)

Task 7: Now implement the queue ADT in a new file *queue_ll.c* using the modified linked list as described above. Check its correctness and compare the performance of both implementations of queues in a similar manner to Task 6. You can refer to the driver provided in the same for the general flow and File IO template.

Home Exercise 4: The operating system is responsible for scheduling the different processes. There are various scheduling algorithms that can be used. The simplest one is the First-Come-First-Serve (FCFS) algorithm. In this algorithm, the processes are scheduled in the order in which they arrive. The next process is scheduled only after the current process has finished executing. This algorithm is very simple to implement and is used in batch systems.

The following is the structure that is used to represent a process:

```
typedef struct process
{
    int pid;
```

```
    int arrival_time;
    int burst_time;
} Process;
```

You are given an input file *fcfs_input.txt* in the "HW Exercises" directory that contains the details of the processes that arrive. The file contains the following information:
The first line contains the total number of processes n.
The next n lines contain the details of the processes in the following format:
pid arrival_time burst_time

You can assume that the processes arrive in the order in which they are given in the input file. The pid is a unique identifier for the process. The arrival_time is the time at which the process arrives. The burst_time is the time required by the process to finish executing.

You are required to modify your queue implementation to implement the FCFS algorithm. The queue should contain the processes that are waiting to be executed. The queue should be implemented using a linked list.

Every time unit, do the following:

1. If a process arrives at the current time unit, insert it into the queue.
2. If a process is currently executing, decrement its burst time by 1. If the burst time becomes 0, the process has finished executing. Print the process id and the time at which it finished executing.
3. If no process is currently executing, remove the first process from the queue and start executing it. Print the process id and the time at which it started executing.
4. If no process is currently executing and the queue is empty, do nothing.

An example input file is given below:

```
5
1 0 5
2 1 3
3 3 2
4 8 4
5 9 1
```

The output for the above input file is given below:

```
Process 1 started at time 0
Process 1 finished at time 5
Process 2 started at time 5
```

```
Process 2 finished at time 8
Process 3 started at time 8
Process 3 finished at time 10
Process 4 started at time 10
Process 4 finished at time 14
Process 5 started at time 14
Process 5 finished at time 15
```

[Note: The output for *fcfs_input.txt* would be different]