CSCI 2270: Data Structures

Recitation #6 (Section 101)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours - 10am to 2pm on Mondays in ECAE 128**
  - In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.
  - Also, you can attend any TA's office hours. Timings are available on moodle in calendar.

Logistics

- Midterm 1 on **Feb 21, 2020 from 5pm to 7pm**
  - Details about location available on Moodle
  - For Section 100 (Instructor Zagrodzki) : Probably **DUAN G1B20**
- Special Accommodation
  - On Feb 21, 2020 from 5pm onwards
  - Location - TBD

Logistics

Make Up Assignment

- You have an opportunity to redo one of the assignments in which you have scored less points and get 100% in that assignment.

- You can choose any assignment from Assignment 1 to Assignment 4.

- Make sure the assignment runs perfectly fine on your local system.

- In Interview Grading, I will ask basic questions to check that you understood what you have done and to ensure you have done that yourself.
    - When? - Office Hours. Send me an email to reserve a time slot for 10-15 mins during my office hours.
    - Needs to be done by the end of next week (February 24th to February 28th)

Logistics

Midterm Format

- 70% weightage for 2 coding questions
  - Q1 is mandatory to do.
  - Can opt to do either Q2 or Q3 (your choice)
  - No coding question on stacks or queues
  - IN MY OPINION, Linked Lists and Array Doubling are important topics for this section.

- 30% weightage for MCQs
  - 6 MCQs and 5 points for each MCQ.
  - IN MY OPINION, Static and Dynamic Memory allocation, Arrays, Linked Lists, Stacks and Queues are important topics for this section.

Logistics

Midterm Format

- Don’t worry about including header files or writing “using namespace std”
- You are generally given a skeleton code and you just have to implement the required function.
- One page cheat sheet is allowed (front and back).
    - Note down things that you think are important.

Any questions on Logistics?

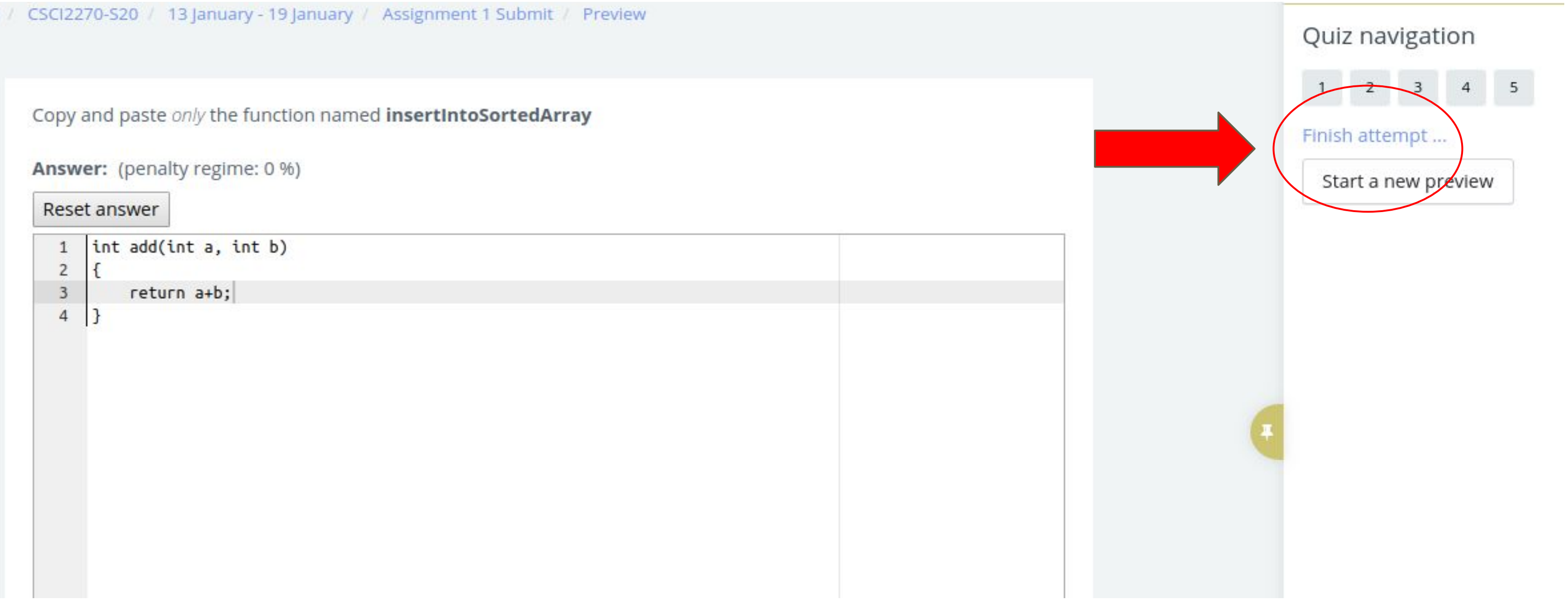
Logistics

Thank You for filling out FCQs

Logistics

- **Assignment 5 is due on Sunday, March 1 2020, 11:59 PM. GOOD LUCK!**
  - Any questions on that?

Please click on “Finish Attempt” after you are done!



Today’s Agenda

- Review (40 ~ 45 mins)
  - Practice Midterm discussion
  - Static and Dynamic Memory Allocation
  - Pass By Value,Pointers and Reference
  - Array Doubling
  - Linked List
  - Stacks
  - Queues
  - Few practice questions from my side

- Your questions

Practice Midterm

Practice Midterm

Q1.

How many of you have actually looked at it and tried to solve it?

Practice Midterm

Q1.

How many of you have actually looked at it and tried to solve it?

Solutions to the practice midterm are available on Moodle now.

Practice Midterm - MCQs

Question 1

Which one of the following statements is <u>false</u>?

a. Accessing an element in an array is generally slower than in a linked list

b. Linked lists generally occupy more space than arrays

c. Adding an element in the middle of a medium-sized array is generally slower than a linked list

d. Inserting an element in the middle of a linked list is generally faster than an array

Practice Midterm - MCQs

Question 1

Which one of the following statements is false?

a. Accessing an element in an array is generally slower than in a linked list

b. Linked lists generally occupy more space than arrays

c. Adding an element in the middle of a medium-sized array is generally slower than a linked list

d. Inserting an element in the middle of a linked list is generally faster than an array

Practice Midterm - MCQs

Question 1

Which one of the following statements is false?

a. Accessing an element in an array is generally slower than in a linked list

Explanation :

You can access individual elements in an array by just using its index, A[i]
It takes just O(1) time.
However, for linked lists you need to traverse the linked list, starting from head to that element which takes O(n) time.

Practice Midterm - MCQs

Question 1

Which one of the following statements is false?

b. Linked lists generally occupy more space than arrays

Explanation :

Well, in an array you just need to store the elements that you care about. In a linked list, you need to store both the element and a pointer to the next node in the linked list. That extra space for pointers is not required by arrays.

Practice Midterm - MCQs

Question 1

Which one of the following statements is false?

c. Adding an element in the middle of a medium-sized array is generally slower than a linked list

Explanation :

For inserting in the middle of a linked list, you traverse to the middle of a linked list and just insert your node. For inserting in an array, you have to move all the elements from the middle position to the last position by one position which will take time. That's why inserting at the middle position is slower in an array.

Practice Midterm - MCQs

Question 1

Which one of the following statements is false?

d. Inserting an element in the middle of a linked list is generally faster than an array

Explanation :

It’s the same as option “c” but it in different words.

Practice Midterm - MCQs

Question 2

Which of these data structures would be the most useful when storing information on all eight planets in our solar system, knowing that the program will need to access this information many times in no particular order?

a. Stack

b. Queue

c. Dynamically doubling array

d. Fixed size array

Practice Midterm - MCQs

Question 2

Which of these data structures would be the most useful when storing information on all eight planets in our solar system, knowing that the program will need to access this information many times in no particular order?

a. Stack

b. Queue

c. Dynamically doubling array

d. Fixed size array

Practice Midterm - MCQs

Question 2

Which of these data structures would be the most useful when storing information on all eight planets in our solar system, knowing that the program will need to **access this information many times in no particular order**?

a. Stack

Explanation :

With Stack, you can only access elements in LIFO order and so random access of information is not possible.

Practice Midterm - MCQs

Question 2

Which of these data structures would be the most useful when storing information on all eight planets in our solar system, knowing that the program will need to **access this information many times in no particular order**?

b. Queue

Explanation :

With Queue, you can only access elements in FIFO order and so once again random access of information is not possible.

Practice Midterm - MCQs

Question 2

Which of these data structures would be the most useful **when storing information on all eight planets in our solar system**, knowing that the program will need to access this information many times in no particular order?

c. Dynamically doubling array

Explanation :

We just need to store information for 8 planets. The number of planets is fixed. We don't need a dynamically doubling array in such a case.

Practice Midterm - MCQs

Question 2

Which of these data structures would be the most useful when storing information on all eight planets in our solar system, knowing that the program will need to access this information many times in no particular order?

d. Fixed size array

Explanation :

An array of size 8 can be used to store information of all 8 planets and access that information randomly in no particular order.

Practice Midterm - MCQs

Question 3

Using a dynamically doubling array with an initial capacity of 20, how many resizing operations would be required to accommodate 1000 elements?

a. 4

b. 5

c. 6

d. 7

Practice Midterm - MCQs

Question 3

Using a dynamically doubling array with an initial capacity of 20, how many resizing operations would be required to accommodate 1000 elements?

a. 4

b. 5

c. 6

d. 7

Practice Midterm - MCQs

Question 3

Using a dynamically doubling array with an initial capacity of 20, how many resizing operations would be required to accommodate 1000 elements?

a. 4

b. 5

c. 6

d. 7

Doubling the array first time	New Array size = 20*2=40
Doubling the array second time	New Array size = 40*2=80
Doubling the array third time	New Array size = 80*2=160
Doubling the array fourth time	New Array size = 160*2=320
Doubling the array fifth time	New Array size = 320*2=640
Doubling the array sixth time	New Array size = 640*2=1280

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

a. 488

b. 487

c. 617

d. 611

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

a. 4 8 8

b. 4 8 7

c. 6 1 7

d. 6 1 1

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

Queue is empty.
front and rear both are currently NULL

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

1

front and rear both currently point to 1

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

1	6

front currently points to 1
rear currently points to 6

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

1	6	4

front currently points to 1
rear currently points to 4

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

1	6	4	8

front currently points to 1
rear currently points to 8

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

6	4	8

front currently points to 6
rear currently points to 8

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

4	8

front currently points to 4
rear currently points to 8

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

4	8

front currently points to 4
rear currently points to 8

Prints 4 on the output screen

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

8

front currently points to 8
rear currently points to 8

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

8

front currently points to 8
rear currently points to 8

Prints 8 on the output screen

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

8	7

front currently points to 8
rear currently points to 7

Practice Midterm - MCQs

Question 4

What will be printed when the following code is run?

```
Queue q; // initialized to be empty
q.enqueue(1);
q.enqueue(6);
q.enqueue(4);
q.enqueue(8);
q.dequeue();
q.dequeue();
cout << q.peek_front();
q.dequeue();
cout << q.peek_front();
q.enqueue(7);
cout << q.peek_front();
```

8	7

front currently points to 8
rear currently points to 7

Prints 8 on the output screen

Practice Midterm - MCQs

Question 5

Which of the following is <u>not</u> a difference between static and dynamic memory?

a. Dynamic memory doesn’t obey normal scoping rules

b. Static variables have names, while dynamic variables do not

c. Pointers can only be allocated dynamically

d. Dynamic memory is allocated using the new keyword, while static memory is not

Practice Midterm - MCQs

Question 5

Which of the following is not a difference between static and dynamic memory?

a. Dynamic memory doesn’t obey normal scoping rules

b. Static variables have names, while dynamic variables do not

c. Pointers can only be allocated dynamically

d. Dynamic memory is allocated using the new keyword, while static memory is not

Practice Midterm - MCQs

Question 5

Which of the following is not a difference between static and dynamic memory?

a. Dynamic memory doesn’t obey normal scoping rules

Explanation :

- What do we mean by scoping rules?
  - The scope rules of a language decide in which part(s) of the program a particular piece of code or data item can be accessed.
- Dynamically allocated memory doesn’t follow the same scoping rules as static memory.
  - For example: if we allocate some heap memory using new operator inside a function, then that memory is not deallocated when the function ends. You have to delete it yourself using the “delete” operator or else it leads to MEMORY LEAK.

Practice Midterm - MCQs

Question 5

Which of the following is not a difference between static and dynamic memory?

b. Static variables have names, while dynamic variables do not

Explanation :

This is true. The memory that we allocate from heap doesn't get any name. That's why we use a pointer to access it. (We discussed this in detail during Recitation 3. Refer to those slides for further information)

Practice Midterm - MCQs

Question 5

Which of the following is not a difference between static and dynamic memory?

c. Pointers can only be allocated dynamically

Explanation :

Well, that's clearly wrong.

Int *p = &a;
// This is a valid c++ statement and the pointer is allocated memory statically.

Practice Midterm - MCQs

Question 5

Which of the following is not a difference between static and dynamic memory?

d. Dynamic memory is allocated using the new keyword, while static memory is not

Explanation :

Well, that's the definition. Can't argue with that!

Any Questions?

Practice Midterm - Coding Question

Q1. Write a function to check if the length of the linked list is even or not.

What's your approach?

Practice Midterm - Coding Question

Q1. Write a function to check if the length of the linked list is even or not.

```
Pseudocode:

    1. SET: temp pointer to head
    2. SET: num_nodes = 0
    3. LOOP-WHILE: temp is not NULL
            num_nodes += 1
            temp = temp->next
    4. IF num_nodes % 2 == 0: return True
    5. RETURN False
```

Practice Midterm - Coding Question

Q2. Write a function to delete all the nodes with negative values from a linked list.

What's your approach?

What are the possible edge cases?

Practice Midterm - Coding Question

Q2. Write a function to delete all the nodes with negative values from a linked list.

```
1. SET: curr pointer to head and prev pointer to NULL
2. LOOP-WHILE: curr is not NULL
        IF curr>data < 0:
            IF curr == head:
                head = head->next;
                delete curr;
                curr = head;
            ELSE:
                prev->next = curr->next;
                delete curr;
                curr = prev->next;
        ELSE:
            prev = curr;
            curr = curr->next;
3. RETURN head;
```

Practice Midterm - Coding Question

Q3. Write down the code for a dynamic array implementation of stack that halves its size when the extra space isn't needed.

What's your approach?

What are the possible edge cases?

Any Questions?

Static Memory and Dynamic Memory

- Static Memory Allocation
  - Memory for named variables is allocated by the compiler at compile time.
  - Exact size and type of storage must be known at compile time.
  - For standard array declarations, this is why the size has to be constant.
  - Performed using Stack Memory.
- Dynamic Memory Allocation
  - Memory is allocated to variables “on the fly” during run time.
  - Performed using Heap Memory (Done using “new” operator)

Static Memory Allocation (using Stacks)

- Stack memory keeps track of all the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution, and handles allocation of all function parameters and local variables.

- For example:

```
int add(int a, intb){
        return a+b;
}
int main() {
        int x =5, y=15;
        int z;
        z = add(x,y);
        return 0;
}
```

Dynamic Memory Allocation (Using Heap)

- In order to use the heap memory in C++ we use the **“new”** and **“delete”** keywords.
    - “new” for allocating memory on the heap
    - “delete” for deallocating/freeing memory from the heap

- Creating variables on the heap is very different from creating variables on the stack. On stack memory, variables get actual names and addresses. On heap memory, they don’t get names, JUST ADDRESSES.

- Instead, we use a pointer to allocate the memory for the variable.

Dynamic Memory Allocation

- "new" operator

int* p1; ⟶ Stored on the stack

STACK	Address
int* p1	0x7ffd9181ddec

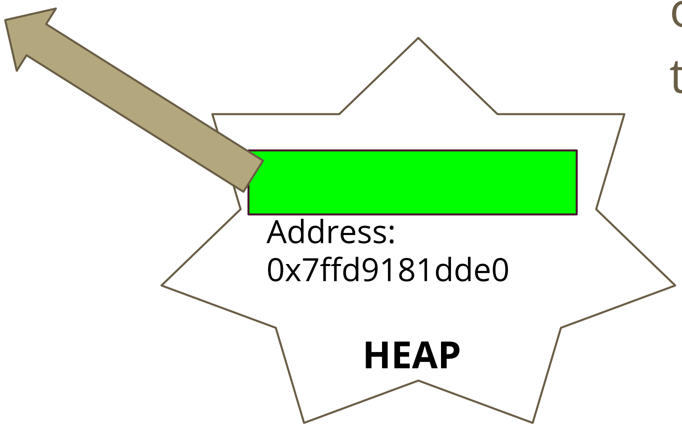
Dynamic Memory Allocation

- **“new”** operator

int* p1; → Stored on the stack

STACK	Address
int* p1	0x7ffd9181ddec

p1 = new int; → Allocates a memory block on Heap of size int and stores the address of that block in pointer p1



cout<<p1;

Output: 0x7ffd9181dde0

Dynamic Memory Allocation

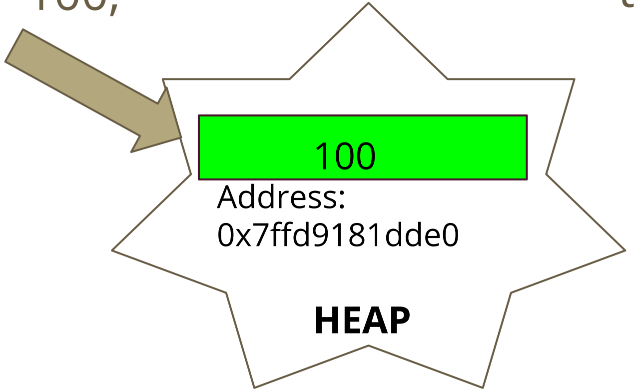
- **“new”** operator

STACK	Address
int* p1	0x7ffd9181ddec

int* p1; Stored on the stack

p1 = new int; Allocates a memory block on Heap of size int and stores the address of that block in pointer p1

*p1 = 100;



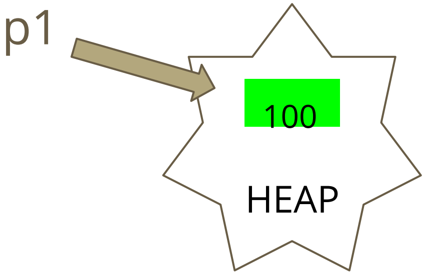
cout<<p1<< “ “ << *p1;

Output: 0x7ffd9181dde0 100

Dynamic Memory Deallocation

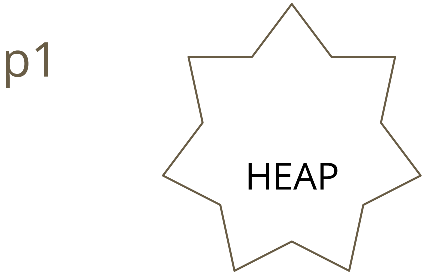
- **“delete”** operator (Visualization)
  int* p1; // pointer p1 stored on the STACK
  p1 = new int; // pointer p1 stores the address of a memory block on HEAP
  *p1 = 100; // modifies the value pointed by p1 to 100

STACK	ADDRESS
int* p1	0x7ffd9181ddec



delete p1;

STACK	ADDRESS
int* p1	0x7ffd9181ddec



Any Questions?

Alright, I have a question for you guys

```
# include<iostream>
using namespace std;

void fun(int *a)
{
    a = new int;
    *a = 5;
}

int main()
{
    int *p;
    fun(p);
    *p = 6;
    cout<<p;
    return 0;
}
```

What happens when you run it?

a) Segmentation Fault
b) Prints 6
c) Prints 5
d) Compilation Error

Who gets confused with the changeValue function?

```
#include <iostream>
using namespace std;
int gobal_var = 42;

void changeValue(int* &some_pointer) {
	some_pointer = &gobal_var;
}

int main() {
	int var = 23;
	int* ptr_to_var = &var;
	cout << "Before :" << *ptr_to_var << endl;
	changeValue(ptr_to_var);
	cout << "After :" << *ptr_to_var << endl;
	return 0;
}
```

Quick Review: Pass-by-Value

- Creates a local copy of variables

```
void add2 (int num)
{
    num = num + 2;
}

int main ()
{
    int a = 10;
    add2(a);
    cout << a;
}
```

Address	Stack
0x7ffeeacf8e2c	int a = 10
	...
SOME ADDRESS	int num = 12

Terminal

```
10
```

Quick Review: Pass-by-Pointers

- What happens in this case?

```
void add2 (int * num)
{
    *num = *num + 2;
}

int main ()
{
    int a = 10;
    add2( &a );
    cout << a;
}
```

Address | **Stack**

Address	Stack
0x7ffeeacf8e2c	int a = 12
	…
	num = 0x7ffeeacf8e2c

Terminal

12

Quick Review: Pass-by-Reference

- What happens in this case?

```
void add2 (int &num)
{
    num = num + 2 ;
}

int main ()
{
    int a = 10 ;
    add2( a );
    cout << a;
}
```

Address | **Stack**

Address	Stack
0x7ffeeacf8e2c	int a = 12
	...

Terminal

```
12
```

Who gets confused with the changeValue function?

```
void changeValue(int* &some_pointer) {
        some_pointer = &gobal_var;
}
```

- Generally, & operator is used to access a variable's address.

- BUT when & operator is used with a parameter in a function, it just means that the variable is being passed by reference. That means no local copy of that variable will be created on our stack memory.
  Thus, if the variable is modified inside the function, its value gets modified outside the function as well.

Who gets confused with the changeValue function?

```
void changeValue(int* &some_pointer) {
        some_pointer = &gobal_var;
}
```

- To understand what's going on, I generally remove the & operator to see what's left.
  void changeValue(int* some_pointer)

  Okay, so now the parameter that is being passed is a pointer.

- Since, it was passed with & operator, that just means it is being passed by reference. No local copy of that variable will be created on my stack memory.

Who gets confused with the changeValue function?

```
void changeValue(int* &some_pointer) {
        some_pointer = &gobal_var;
}
```

- So, this just means that I am passing a pointer variable as a parameter to this function and I am passing that pointer by reference.

- If the pointer is modified inside the function, the changes will be reflected outside the function as well.

Any Questions?

Array Doubling

- This was Recitation 3's exercise.

- You implemented the resize function, which when called, creates a new array whose size is double of the current array's size and copies all elements from the current array to the new array. At the end it deletes the current array and returns a pointer to the new array.

Array Doubling

- This is what it looks like

```
int* resize(int* arrayPtr, int &capacity)
{
    // Implement resizing logic
    int newCapacity = capacity * 2;
    cout<<"Resizing from "<<capacity<<" to "<<newCapacity<<endl;
    int *newArray = new int[newCapacity];
    // copying the data
    for(int i = 0; i < capacity; i++){
        newArray[i] = arrayPtr[i];
    }

    delete arrayPtr;
    //arrayPtr = newArray;
    capacity = newCapacity;

    return newArray;
}
```

Array Doubling

Few additional tips:

- DO NOT FORGET to delete the old array.
- Do not delete the same array multiple times. It leads to runtime error.
- What's the opposite of array doubling?
  - Array Halving. Check out Q3 of Practice Midterm for that.

Question

Assume the dynamic array is full and you wish to insert a new element into it. The time complexity of inserting an element at the end of such a dynamic array is __________

a) O (n)

b) O (n^(½))

c) O (log n)

d) O (1)

Question

Assume the dynamic array is full and you wish to insert a new element into it. The time complexity of inserting an element at the end of such a dynamic array is __________

a) O (n)

b) O (n^(½))

c) O (log n)

d) O (1)

Linked List

- "head" pointer should always point to the first node of the linked list and the last node of the linked list should always point to "NULL".

- How to check if linked list is empty?
  - If (head == NULL)

- A special edge case while traversing the LL (and looking for a value):

```
Node * temp=head;
while(temp->next->key != value  && temp!=NULL)  {
Blah Blah Blah; }
```

  What's the issue with this?

Insertion at the beginning of the linked list

```
//Create a new node
Node* newNode = new Node;
newNode->key = newKey;
//Make it point to the current head of the LL
newNode->next = head;
//Make your head point to the new node
head = newNode;
```

Insertion in the middle of a linked list

```
//Create a new node
Node* newNode = new Node;
newNode->key = newKey;
//Assume who have the pointer pointing to the previous node. Call it prev. If you don't have it, you traverse the linked list and find it.
//Make your new node point to the prev pointer's next
newNode->next = prev->next;
//Make your prev point to the new node
prev->next = newNode;
```

Insertion at the end of a linked list

```
//Create a new node
Node* newNode = new Node;
newNode->key = newKey;
//Traverse the linked list to reach the last node in the list
Node* tmp = head;
while( tmp->next != NULL){
        tmp = tmp->next;
}
//temp pointer now points to the last node in the LL
//Make your last node point to the new node
tmp->next = newNode;
//Make your newNode point to NULL because the last element in a LL
always points to NULL
newNode->next = NULL;
```

Deletion in a Linked List

- First the position at which we want to delete must be found or the node that we want to delete must be found. Call it **“to_be_deleted”**

- After that,
    a. find the node just before the node that we wish to delete. Call it **“prev”**
    b. Modify “prev->next” to point to the node right after the node that you are deleting. (This step is performed to ensure that there is no break in your linked list)
    prev->next = to_be_deleted->next;

- Delete the memory used by the node you want to delete.
  delete to_be_deleted;

Deletion at the beginning of a linked list

```
//Create a new pointer and make it point to the first node in the LL
Node* to_be_deleted = head;
//Make your head point to next node of the LL (this is the 2nd node in LL )
head = head->next;
//delete your to_be_deleted node
delete to_be_deleted;
```

Deletion in the middle of a linked list

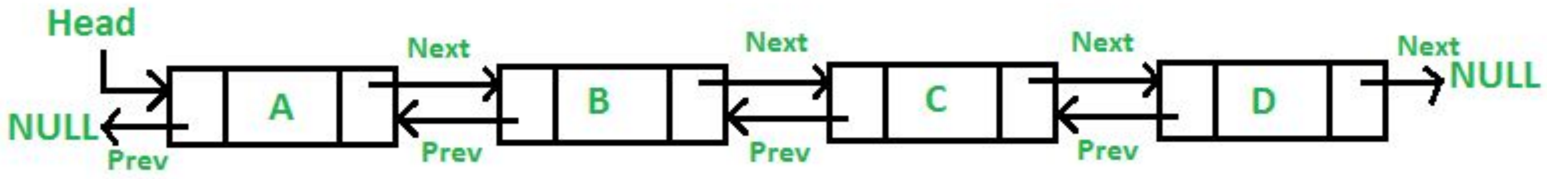
```
//Create a new pointer and make it point to the first node in the LL
Node* to_be_deleted = head;
//pointer to access the node previous to node that will be deleted
Node* prev = NULL;
//Suppose you have been asked to delete the node at index n
//Traverse the linked list till you reach the nth node in the LL
int index=0;  //to check if you have reached the correct index in the LL
while(index!=n && to_be_deleted->next !=NULL){
        prev = to_be_deleted;
        to_be_deleted = to_be_deleted->next;
        index++;
}
//Make your prev point to the node right after your "to_be_deleted" node
prev->next = to_be_deleted->next;
//Delete your "to_be_deleted" node
delete to_be_deleted;
```

Deletion at the end of a Linked List

```
//Create a new pointer and make it point to the first node in the LL
Node* to_be_deleted = head;
//pointer to access the node previous to node that will be deleted
Node* prev = NULL;
//Traverse the linked list till you reach the last node in the LL
while( to_be_deleted->next !=NULL){
        prev = to_be_deleted;
        to_be_deleted = to_be_deleted->next;
}
//Make your prev point to NULL
prev->next = NULL;
//Delete your “to_be_deleted” node
delete to_be_deleted;
```

Doubly Linked List

Has two pointers, one points to the next node in the linked list while the other one points to the previous node in the linked list. One special advantage of doubly linked list is that it allows us to travel back while singly linked list doesn't.



Any questions?

Question

Write a function to find the sum of alternate elements of a linked list.

Question

Write a function to find the sum of alternate elements of a linked list.

Spend next 3-5 mins writing this function.

Function Skeleton -

```
int SumAlternateNode(struct Node* head)
```

Returns the sum of the alternate node’s data.

Question

Write a function to find the sum of alternate elements of a linked list.

```
int SumAlternateNode(struct Node* head)  {
	int count = 0;
	int sum = 0;
	Node *temp = head;
	while (temp != NULL) {
		// Add every other node’s data to sum starting from the “head” node.
		if (count % 2 == 0)
			sum += temp->data;
		count++;			// count the number of nodes visited so far
		temp = temp>next;		// move on to the next node in the LL
		}
	return sum;
}
```

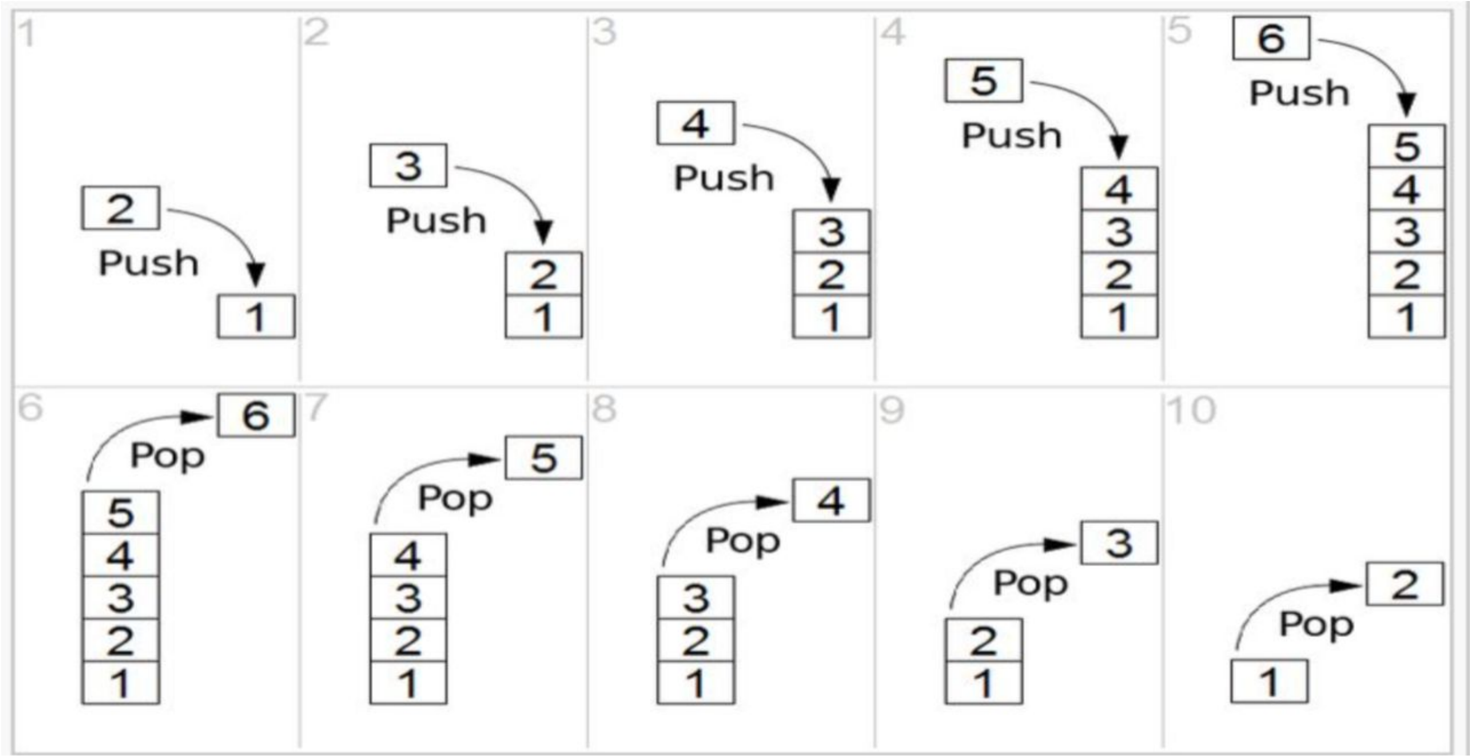
Stacks

- Formal definition - It is a linear data structure which follows a particular order in which the elements are inserted or removed.
    - Can only perform operations from one end. Generally called “TOP”

- LIFO - Last In First Out
    - The element that was inserted last in a stack will be removed first from the stack.
    - This also implies that the element that was inserted first in a stack will be removed last from the stack.

- Typical stack operations:
    - Push - Insert an element into your stack
    - Pop - Remove an element into your stack
    - isFull - Is the stack full?
    - isEmpty - Is the stack empty?
    - Peek - What’s the most recent value that was entered?

Stacks



1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

Array Implementation of a Stack

- Elements are stored in an array but while performing push and pop operations, ensure LIFO order is followed.

- Initialize variable **top** to 0

- **top** of the stack refers to the index of the array where the next element will be added. All the insert/delete operations occur using **top**
  - When stack is empty, top = 0
  - When stack is full, top = maximum size of the array
  - Overflow error when top> maximum size of the array

Linked List Implementation of a Stack

- Why is there a linked list implementation?
  - because now the size can grow and shrink according to the needs at runtime. (unlike arrays of constant size)

- **top** here is a pointer that always points to the head of the linked list.

- Push() in the Linked list implementation of a stack:
  Every new element is inserted at the head of the list. So, every new element is pointed by the top pointer.

- Pop() in the Linked list implementation of a stack:
  Every element is removed from the head of the linked list (head is referred to as top here). To pop an element, simply delete the node pointed by the top pointer, and make top point to the next node in the list.

Queue

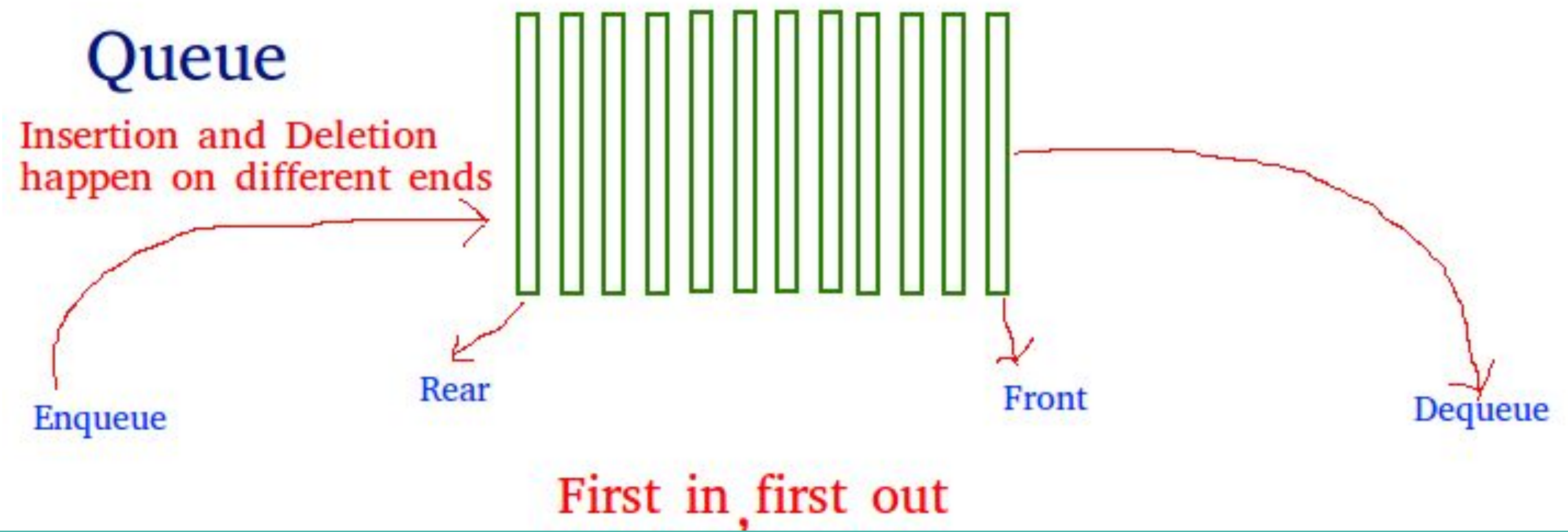
- Formal definition - It is a linear data structure which follows a particular order in which the elements are inserted or removed.
  - Can perform operations from two ends.
  - Generally called (“FRONT” and “REAR”) or (“FRONT” and “END”) or (“HEAD” and “TAIL”)

- FIFO - First In First Out
  - The element that was inserted first in a queue will be removed first from the queue.
  - This also implies that the element that was inserted last in a queue will be removed last from the queue.

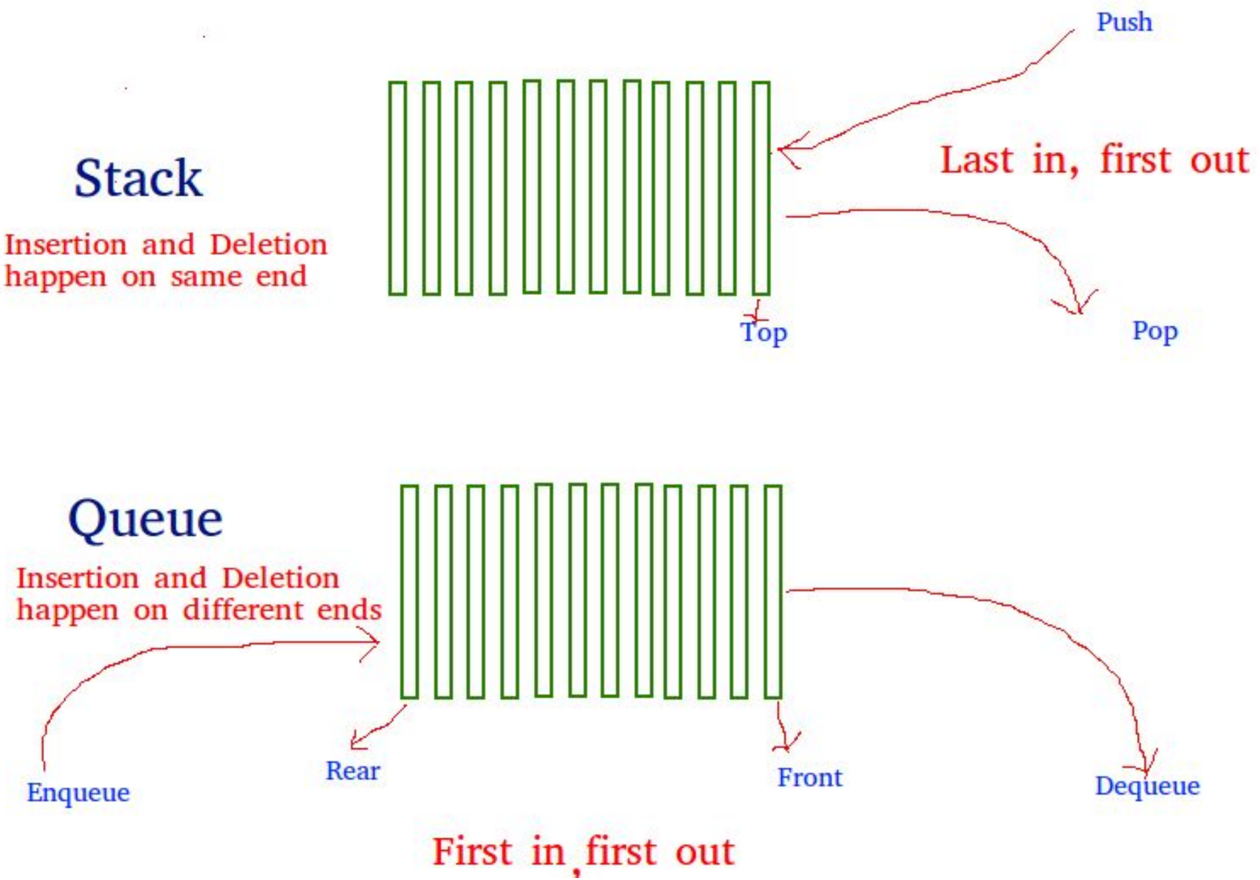
- Typical queue operations:
  - Enqueue - Insert an element in your queue
  - Dequeue - Remove an element from your queue
  - Peek - Returns the front element present in the queue without dequeuing it.
  - isFull - Is the queue full?
  - isEmpty - Is the queue empty?

Queue

- Insertion occurs at the Rear end and Deletion occurs at the Front end
- You remove from the front (head) of the queue.
- You insert at the rear (tail) of the queue.



Stack vs Queue



Queue ADT: Enqueue

Pseudocode: Enqueue

```
Queue.enqueue("D");
```

Queue ADT: Enqueue

Pseudocode: Enqueue

```
Queue.enqueue("D");
```

Queue ADT: Dequeue

Pseudocode: Dequeue

```
Queue.dequeue();
```

Queue ADT: Dequeue

Pseudocode: Dequeue

```
Queue.dequeue();
```

Queue ADT: Peek

Pseudocode: peek

```
var = Queue.peek();

print var;

Output: B
```

Array implementation of a Queue

- Elements are stored in an array but while performing enqueue and dequeue operations, ensure FIFO order is followed.

- Initialize variable **front/head** to -1 and **rear/tail** to -1

- All the enqueue operations occur using **tail** and all the dequeue operations occur using **head**

Queue: Array Implementation

[Different from Recitation Writeup]

Pro: No Pointers!

Con: Not dynamic. Cannot change size at runtime.

Initialize `HEAD = -1, TAIL = -1.`

HEAD			TAIL		
10	11	12	13		
0	1	2	3	4	5

QueueArray: Enqueue

HEAD			TAIL		
10	11	12	13		
0	1	2	3	4	5

0. Check if there is space in the array

```
if (TAIL + 1 == array_size) { return OVERFLOW; }
```

QueueArray: Enqueue

1. Else, Insert the element in the queue

```
Queue[TAIL + 1] = 14;
```

QueueArray: Enqueue

HEAD			TAIL		
10	11	12	13	14	
0	1	2	3	4	5

1. Else, Insert the element in the queue

```
Queue[TAIL + 1] = 14;
```

QueueArray: Enqueue

HEAD			TAIL		
10	11	12	13	14	
0	1	2	3	4	5

2. Update TAIL

```
TAIL = TAIL + 1;
```

QueueArray: Enqueue

HEAD				TAIL	
10	11	12	13	14	
0	1	2	3	4	5

2. Update TAIL

```
TAIL = TAIL + 1;
```

QueueArray: Enqueue

HEAD				TAIL	
10	11	12	13	14	
0	1	2	3	4	5

3. Check if HEAD needs to be updated [Edge Case]

```
if (HEAD == -1) { HEAD = HEAD + 1; }
```

QueueArray: Dequeue

HEAD				TAIL	
10	11	12	13	14	
0	1	2	3	4	5

0. Check HEAD is a valid index [Edge Case]

```
if (HEAD < 0) { return UNDERFLOW;}
```

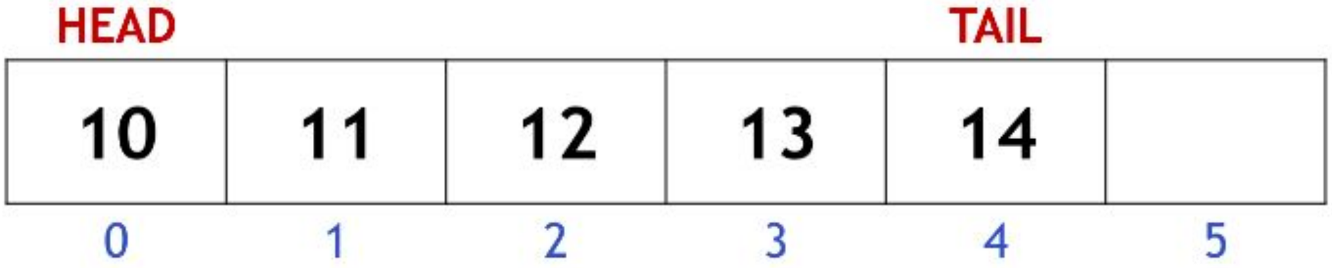
QueueArray: Dequeue

HEAD				TAIL	
10	11	12	13	14	
0	1	2	3	4	5

1. Check if there is only one element in queue [Edge Case]

```
if (HEAD == TAIL) { HEAD = -1; TAIL = -1;}
```

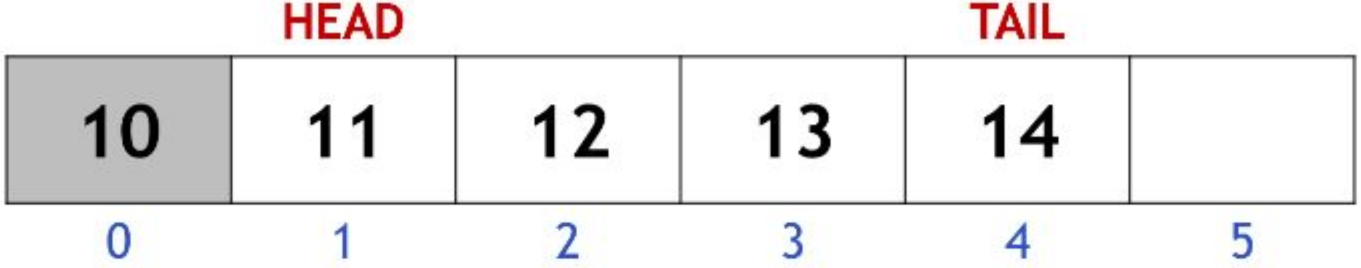
QueueArray: Dequeue



2. Else just update HEAD

```
else { HEAD = HEAD + 1;}
```

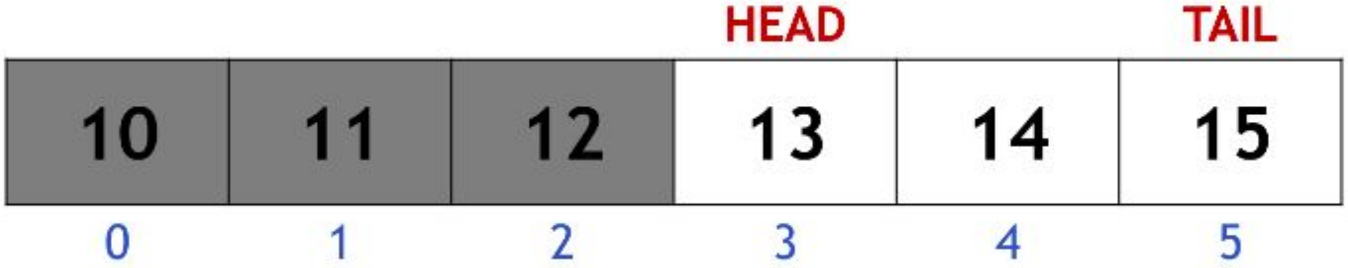
QueueArray: Dequeue



2. Else just update HEAD

```
else { HEAD = HEAD + 1;}
```

Queue: Circular Array Implementation



QueueArray seems to waste a lot of space.

Q. Why was this not an issue in StackArray?

Queue: Circular Array Implementation

Utilize space that is wasted on elements outside the queue

Q. What is `(TAIL + 1) % array_size`***?***

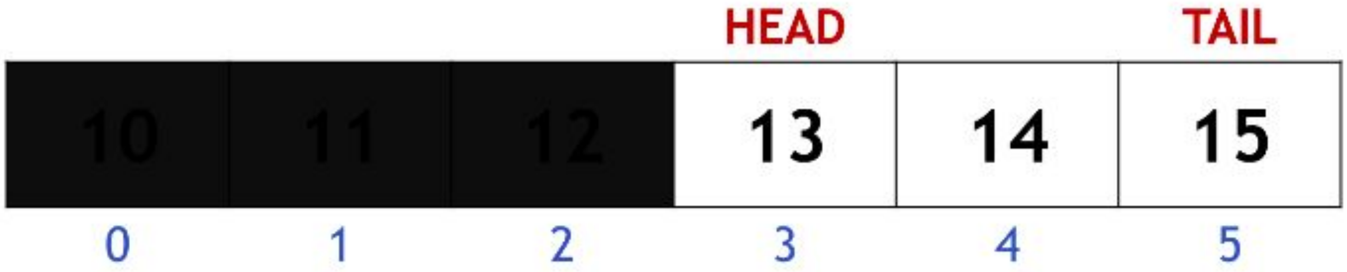
QueueCircular: Enqueue

		TAIL	HEAD		
10	11	12	13	14	15
0	1	2	3	4	5

0. Check if there is enough space [Edge Case]

```
if (HEAD == (TAIL + 1) % array_size) { OVERFLOW; }
```

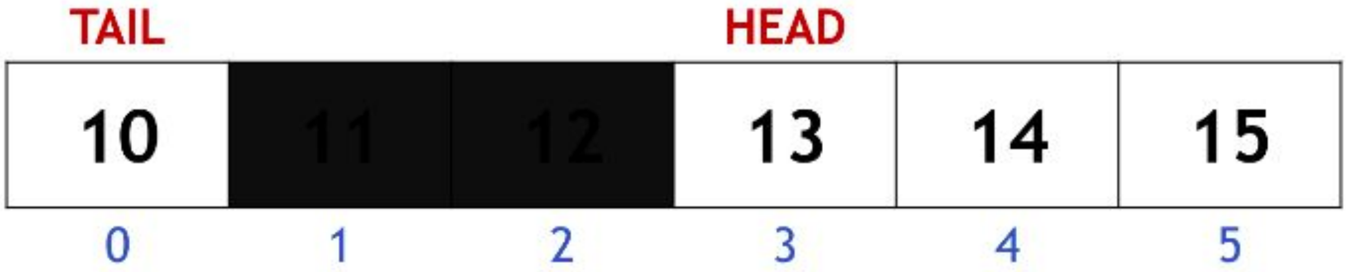
QueueCircular: Enqueue



1. Else, Update TAIL with mod increment

```
TAIL = (TAIL + 1) % array_size;
```

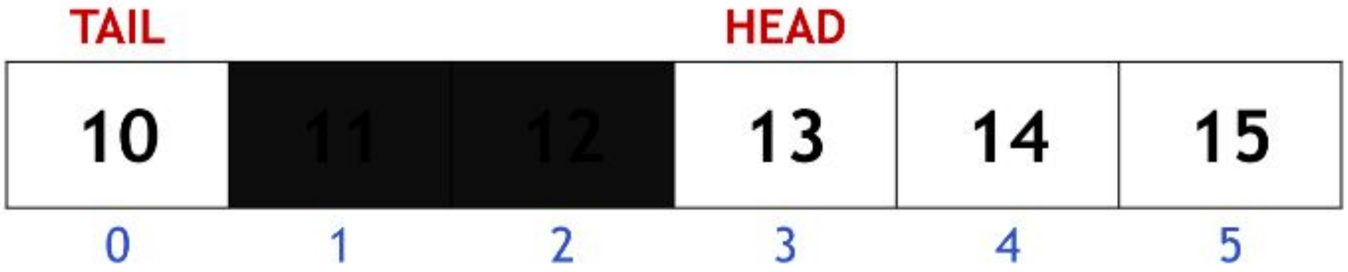
QueueCircular: Enqueue



1. Else, Update TAIL with mod increment

```
TAIL = (TAIL + 1) % array_size;
```

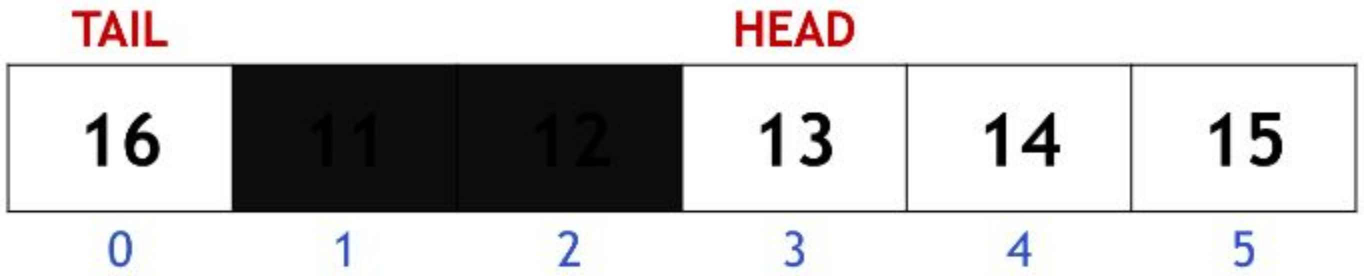
QueueCircular: Enqueue



2. Insert element at TAIL

```
Queue[TAIL] = 16;
```

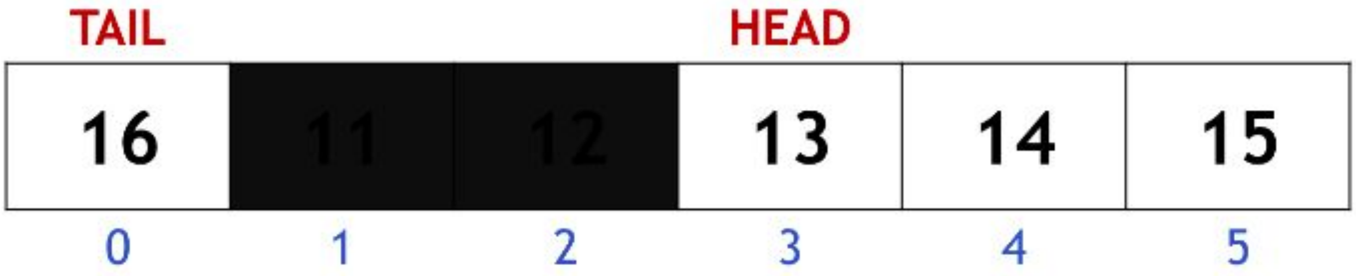
QueueCircular: Enqueue



2. Insert element at TAIL

```
Queue[TAIL] = 16;
```

QueueCircular: Enqueue



3. Check if HEAD needs to be updated [Edge Case]

```
if (HEAD == -1) { HEAD = HEAD + 1; }
```

QueueCircular: Enqueue

TAIL			HEAD		
16	[redacted]	[redacted]	13	14	15
0	1	2	3	4	5

Any Questions?

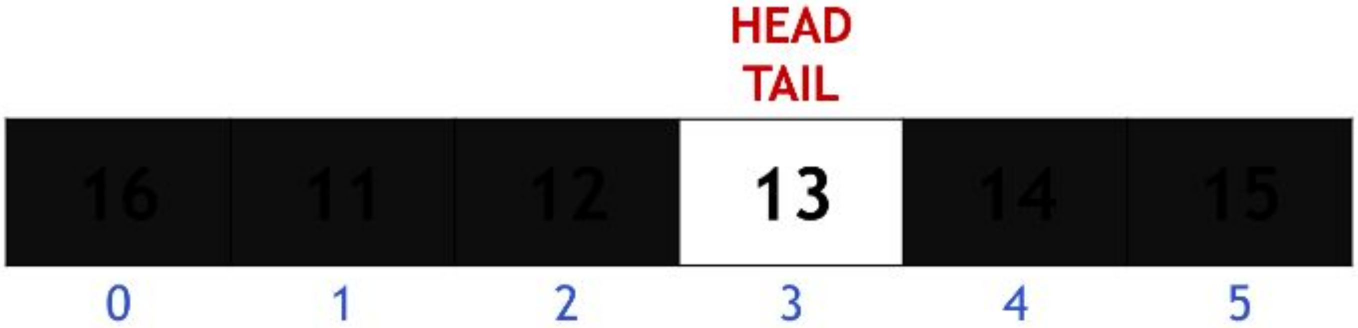
QueueCircular: Dequeue

TAIL			HEAD		
16	[redacted]	[redacted]	13	14	15
0	1	2	3	4	5

0. Check HEAD is a valid index [Edge Case]

```
if (HEAD < 0) { return UNDERFLOW;}
```

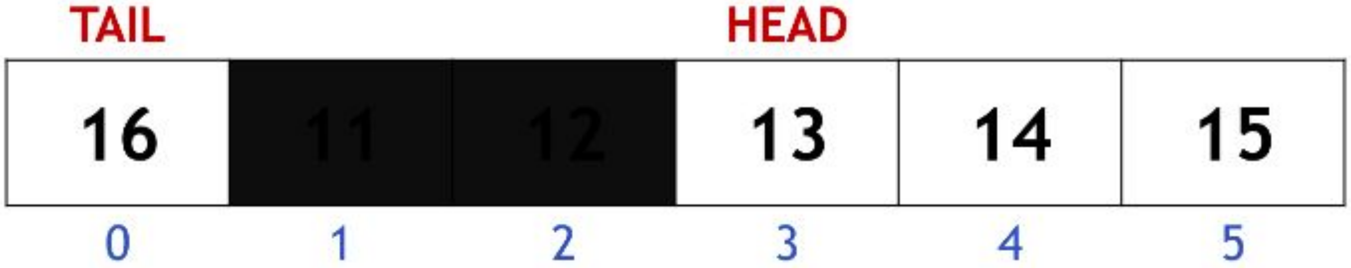
QueueCircular: Dequeue



1. Check if HEAD == TAIL [Edge Case]

```
if (HEAD == TAIL) { HEAD = -1; TAIL = -1; }
```

QueueCircular: Dequeue



2. Else, Update HEAD with mod increment

```
else { HEAD = (HEAD + 1) % array_size; }
```

QueueCircular: Dequeue

TAIL				HEAD	
16	[redacted]	[redacted]	[redacted]	14	15
0	1	2	3	4	5

2. Else, Update HEAD with mod increment

```
else { HEAD = (HEAD + 1) % array_size; }
```

Any Questions?

Linked List Implementation of a Queue

- **front/head** here is a pointer that always points to the head of the linked list.
- **rear/tail** here is a pointer that always points to the last node of the linked list.
- Enqueue() in the Linked list implementation of a queue:
  Every new element is inserted at the end of the linked list. So, every new element is pointed by the **rear** pointer.
- Dequeue() in the Linked list implementation of a queue:
  Element is removed from the beginning of the linked list. To remove an element, simply remove the node pointed by the **front** pointer, and make **front** point to the next node in the list.

Question

Consider the following pseudocode that uses a stack

```
while ( there are more characters in the word to read ){
    read a character
    push the character on the stack
}
while ( the stack is not empty ){
    pop a character off the stack
    write the character to the screen
}
```

What is output for input "geeksquiz"?

a) geeksquizgeeksquiz
b) ziuqskeeg
c) geeksquiz
d) ziuqskeegziuqskeeg

Question

Assume the following circular queue can accommodate maximum six elements with the following be the current status of the queue.

front = 2 rear = 4
queue = ___, L, M, N, ___, ___

What will happen after enqueue(“O”) operation takes place?

a) front = 2 rear = 5
queue = __, L, M, N, O, ___

b) front = 3 rear = 5
queue = ___, L, M, N, O, ___

c) front = 3 rear = 4
queue = __, L, M, N, O, ___

d) front = 2 rear = 4
queue = __, L, M, N, O, __

GOOD LUCK FOR THE MIDTERM