

Malloc, Structs, Queue Lab

void Pointer

- “A pointer that has **no associated data type** with it.”
- Can point to the address of a variable of any data type and can be type-casted to the any type

Example

```
int main()
{
    int a = 10;
    char b = 'A';

    void *p = &a;
    p = &b;
    return 0;
}
```



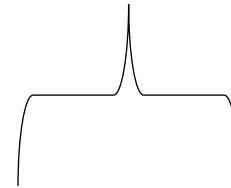
This is allowed. Code will compile and run successfully!

Exercise 1 – What is the output?

```
int main()
{
    int a = 10;
    char b = 'A';

    void *p = &a;
    p = &b;
    printf("Address : %p\n", p);
    return 0;
}
```

Address of variable b



//Address : 0x56F.....

Exercise 2 – What is the output?

```
int main()
{
    int a = 10;
    char b = 'A';

    void *p = &a;
    p = &b;
    printf("Value : %c\n", *p); //Compilation error
    return 0;
}
```

Exercise 2 - Solution

- **Typecast** the void pointer to the type of pointer whose value you are trying to print.

```
int main()
{
    int a = 10;
    char b = 'A';

    void *p = &a;
    p = &b;
    printf("Value : %c\n", *(char*)p); //Value : A
    return 0;
}
```

Why do we need malloc()?

- To be able to allocate space in runtime (or dynamically).
- Example: You do NOT know the size of an array and require user input to determine it.
 - This means size of the array is determined at run time
 - Space has to be allocated in run time (or dynamically!)
 - Time to use malloc()

malloc(size in Bytes)

Why is the return type of malloc() void*??

```
int* p = (int*) malloc (n * sizeof(int));
```

Return type of malloc() is **void***, so we need to typecast it to the correct type (int* in this case)

Stores either:

- 1) Address of the 1st byte of the allocated space in memory, or
- 2) **NULL**, if the allocation fails.

Size in bytes of n integers

Revisiting Strings

- Dynamically allocate space for a string containing 'n' characters
 - Initialize the string to "CS449"
 - Modify the string to "CS44_"

```
char *p = (char*) malloc (sizeof(char) * (n + 1));
```

```
strcpy(p, "CS449");
```

```
p[4] = '_';
```

Why did we use strcpy() to initialize?

```
char *p = (char*) malloc (sizeof(char) * (n + 1));
```

```
p = "CS449";
```

```
p[4] = '_';
```

```
printf("%s", p);
```



Undefined behavior. Will result in segmentation fault most likely.

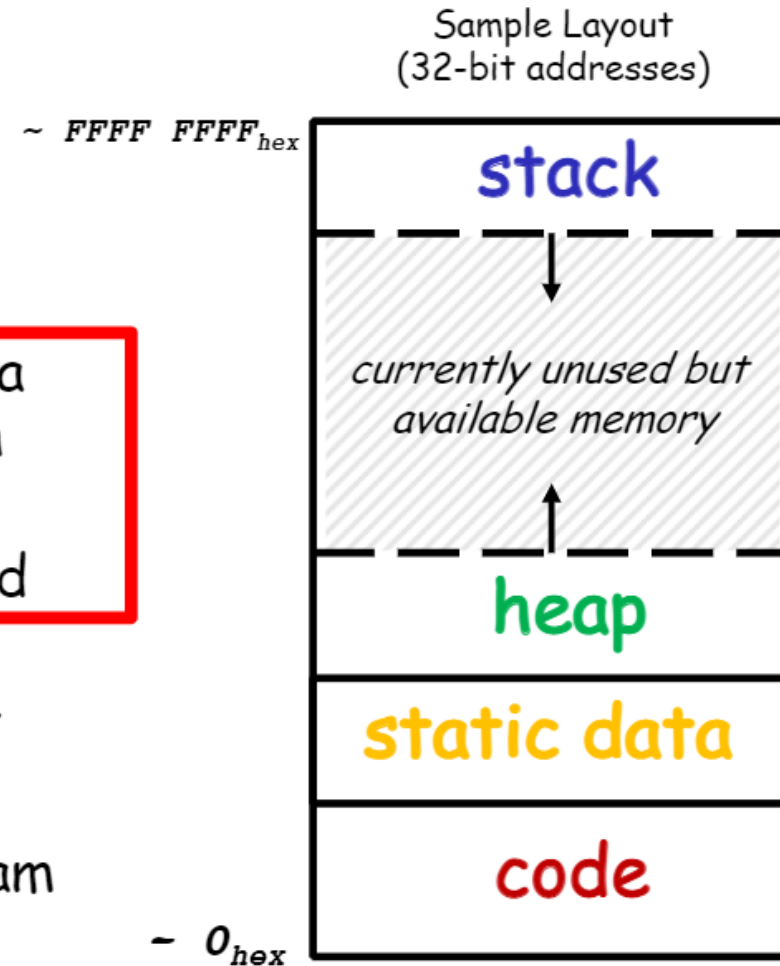
Use **strcpy()** to INITIALIZE a dynamically allocated string

Some Important Points

- malloc() does NOT **initialize** the dynamically allocated space.
 - calloc() initializes the allocated memory space to 0.
- Make sure to do conditional check on whether malloc returned **NULL**
- After you have finished using the dynamically allocated space using malloc, make sure that you **free** the space.

C Memory Layout

- Program's *address space* contains 4 regions:
 - Stack**: local variables, grows downward
 - Heap**: space requested via `malloc()` and used with pointers; resizes dynamically, grows upward
 - Static Data**: global and static variables, does not grow or shrink
 - Code**: loaded when program starts, does not change



Why do we need to free the allocated memory?

```
int *p = malloc(...)
```

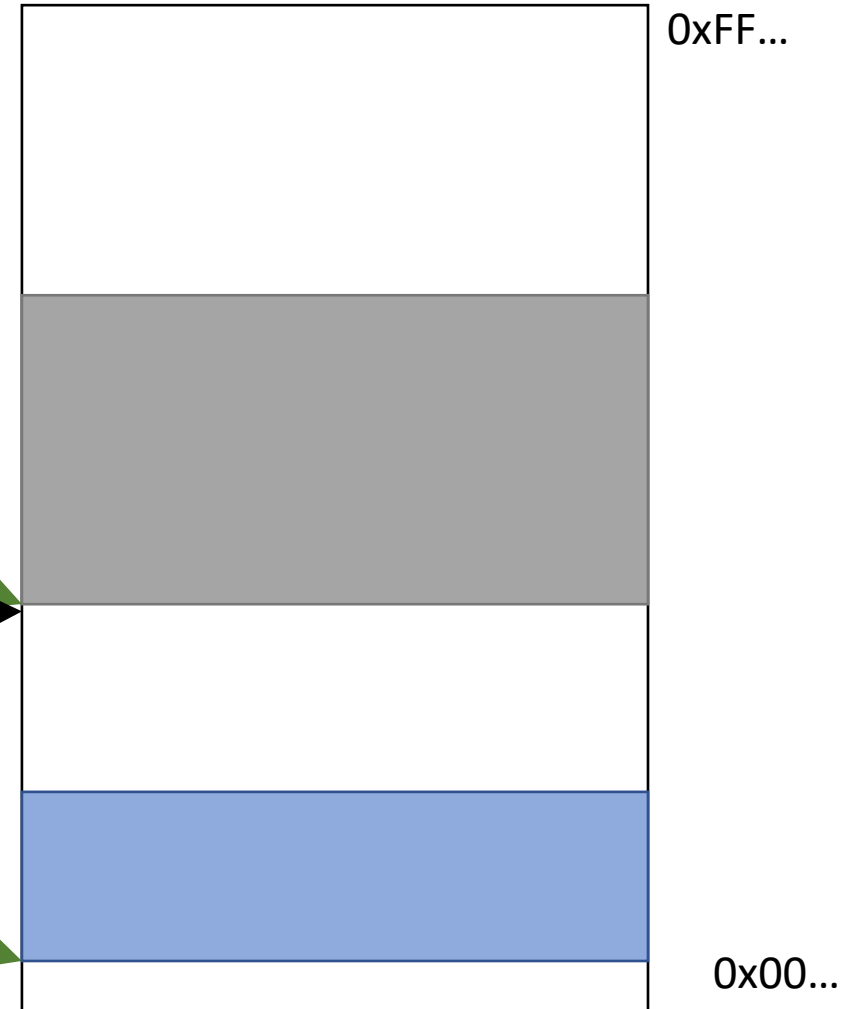
```
..
```

```
..
```

```
//work with the allocated space
```

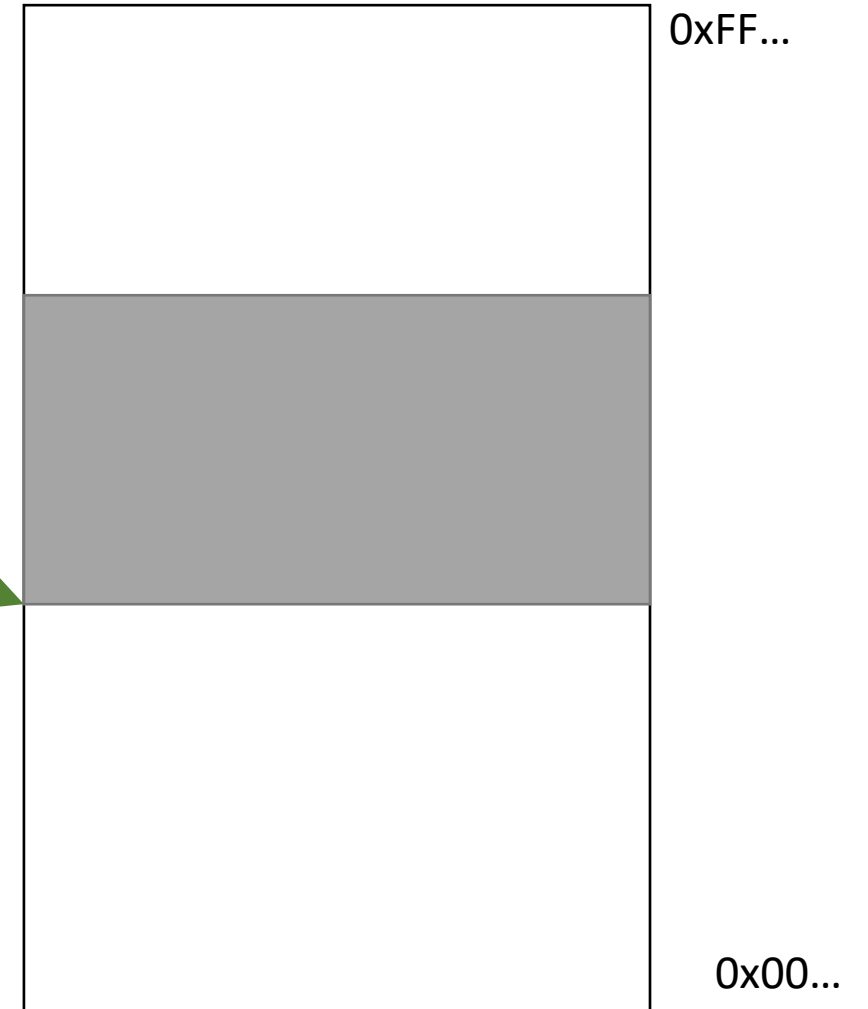
```
..
```

```
int *q = malloc(...)
```



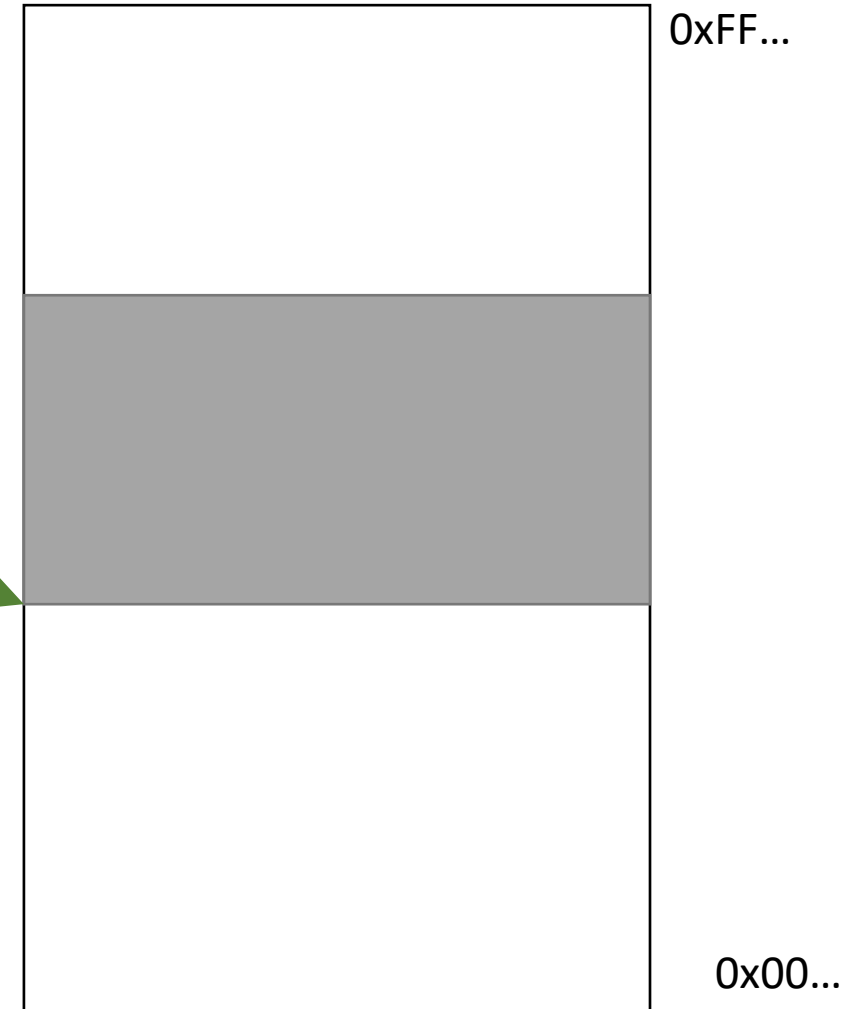
Why do we need to free the allocated memory?

```
int *p = malloc(...)  
..  
//work with the allocated space  
..
```



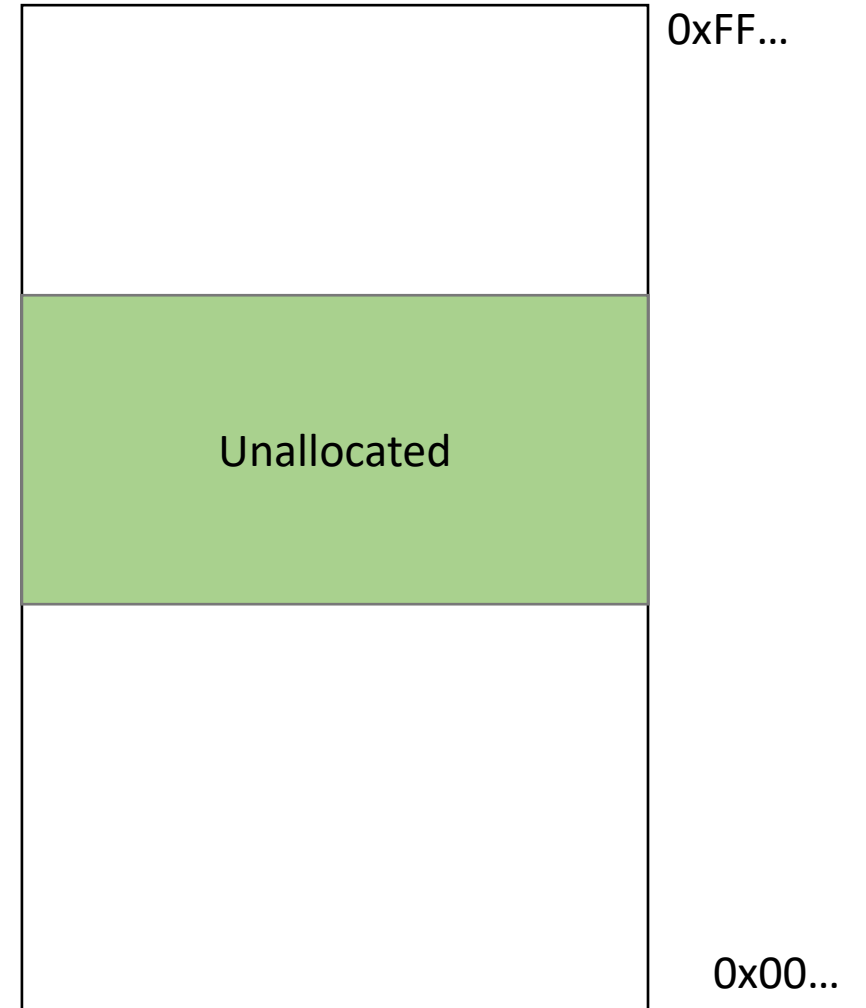
Why do we need to free the allocated memory?

```
int *p = malloc(...)  
..  
//work with the allocated space  
..  
free(p);
```



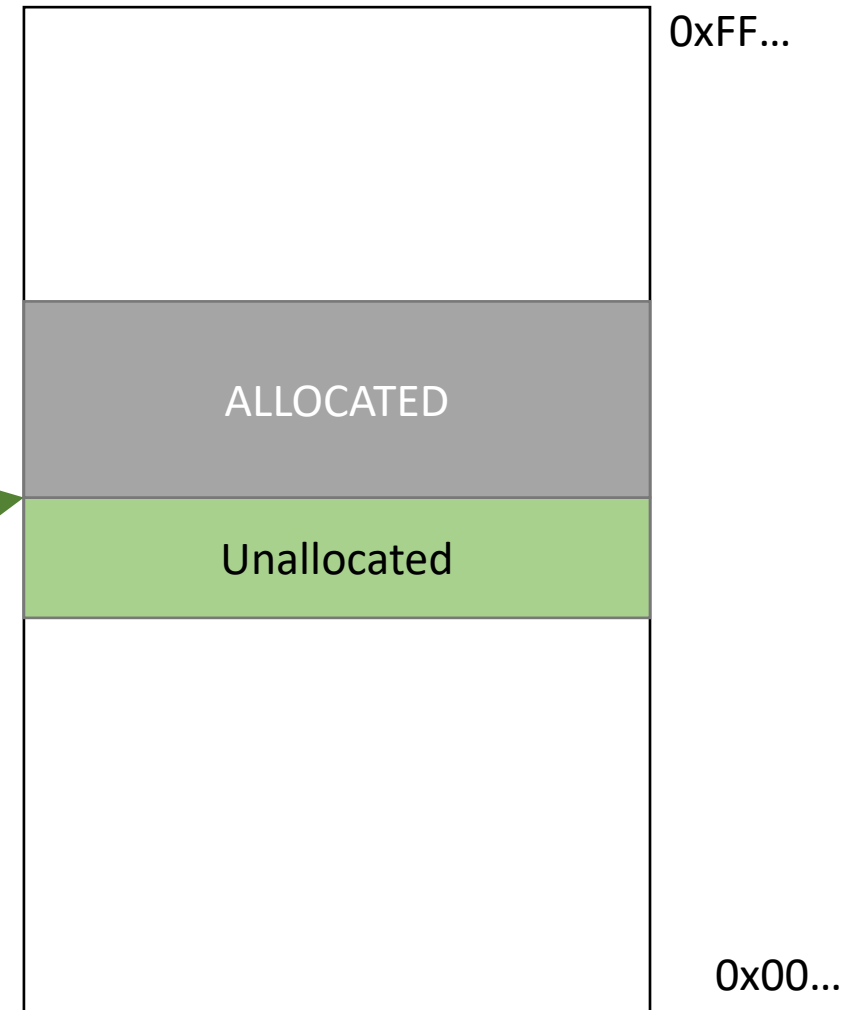
Why do we need to free the allocated memory?

```
int *p = malloc(...)  
..  
//work with the allocated space  
..  
free(p);
```



Why do we need to free the allocated memory?

```
int *p = malloc(...)  
..  
//work with the allocated space  
..  
free(p);  
int *q = malloc(...)
```

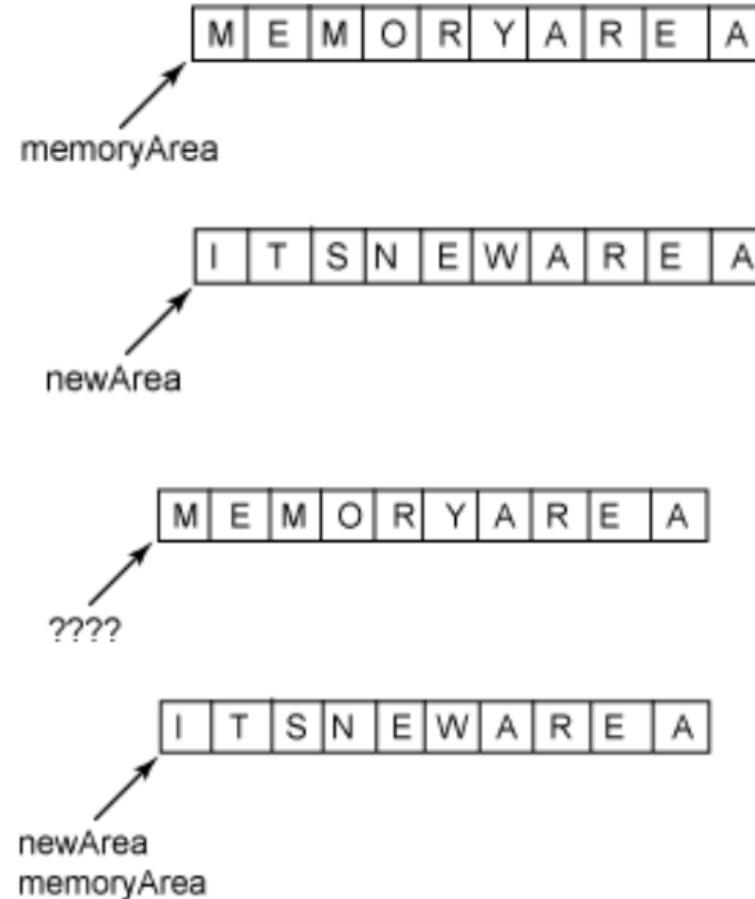


Correct Utilization of free()

- Call free with the pointer that points to the starting address of the allocated space. Otherwise, there will likely be memory leak.

```
char *memoryArea = malloc(10);  
char *newArea = malloc(10);
```

```
memoryArea = newArea;
```



struct

- User defined data type
 - Can hold several data items of different types
- C is not object oriented like Java but structs can be utilized the same way you can use java classes
- C struct vs JAVA class
 - C struct can hold only data items
 - Java class can hold both data items and functions

C struct and Java class

Java

```
public class MyJavaClass
{
    private int x;
    public int getX() {
        return x;
    }
    public int setX(int
value){
        x = value;
    }
}
```



C

```
struct MyJavaClass {
    int x;
};

int MyJavaClass_getX(struct MyJavaClass*
this)
{
    return this->x;
}

void MyJavaClass_setX(struct MyJavaClass*
this, int value)
{
    this->x = value;
}
```

Example

- Use struct to create the profile of **two students** in Pitt
- The attributes of a student are:
 - First name
 - Last Name
 - Peoplesoft #

Design the struct

```
typedef struct Student
{
    char * first_name;
    char * last_name;
    int Peoplesoft;
}Student;
```

Initialize the
credentials of the
1st student

```
int main()
{
    Student *s1 = (Student*)malloc (sizeof(Student));

    s1->first_name = (char*)malloc(25 * sizeof(char));
    strcpy(s1->first_name, "Harry");

    s1->last_name = (char*)malloc(25 * sizeof(char));
    strcpy(s1->first_name, "Potter");

    s1->Peoplesoft = 123456;

}
```

Initialize the
credentials of the
2nd student

```
int main()
{
    ...
    ...
    Student *s2 = (Student*)malloc (sizeof(Student));

    s2->first_name = (char*)malloc(25 * sizeof(char));
    strcpy(s2->first_name, "Hermione");

    s2->last_name = (char*)malloc(25 * sizeof(char));
    strcpy(s2->first_name, "Granger");

    s1->Peoplesoft = 654321;

}
```

Free all the
allocated space

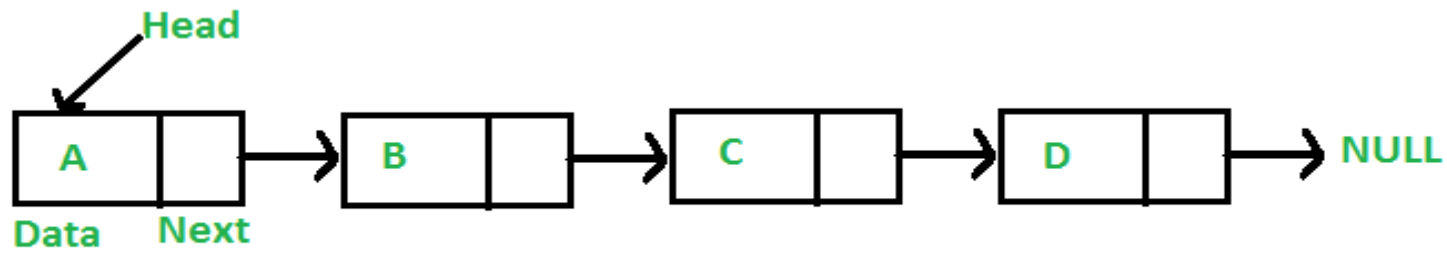
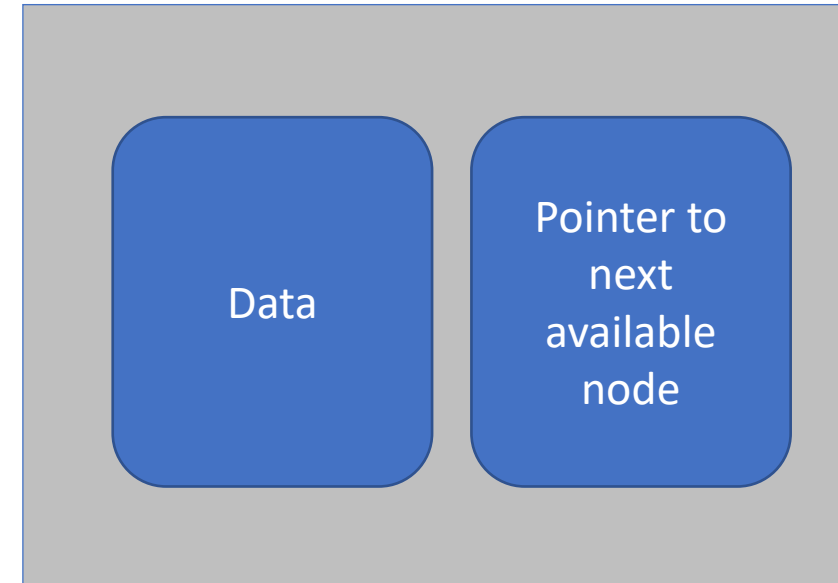
```
int main()
{
    ...
    ...
    free(s1->first_name);
    free(s1->last_name);
    free(s2->first_name);
    free(s2->last_name);

    free(s1);
    free(s2);
}
```

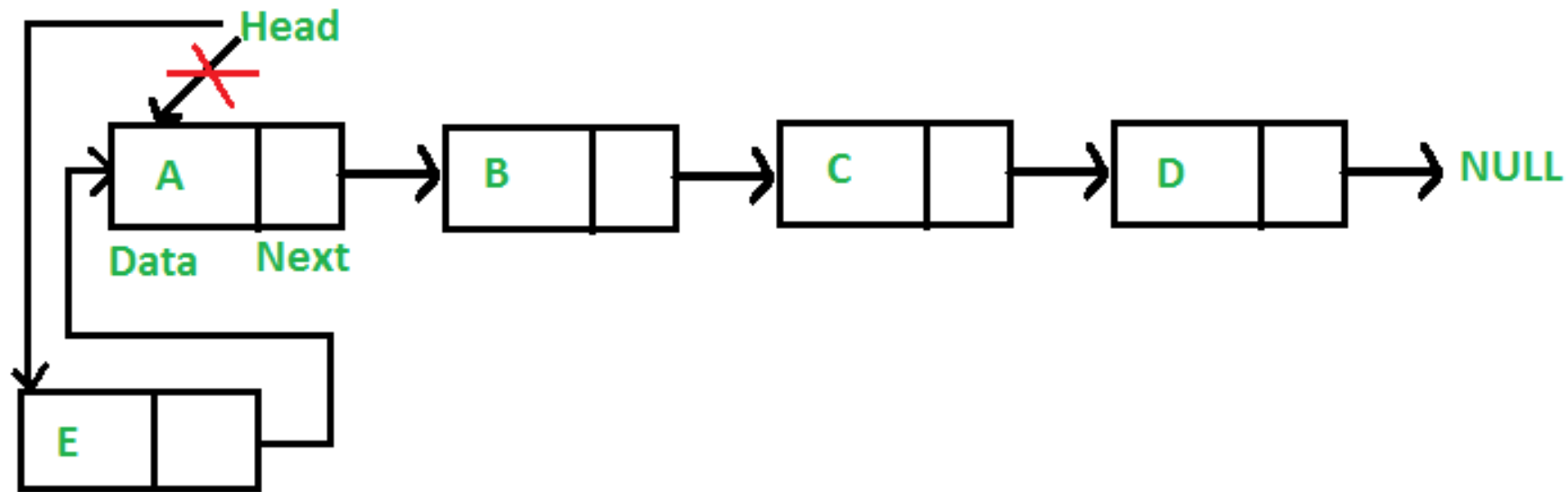

Linked List

```
// A linked list node
struct Node
{
    int data;
    struct Node *next;
};
```

Node



Insertion – At the front



Insertion – At the front

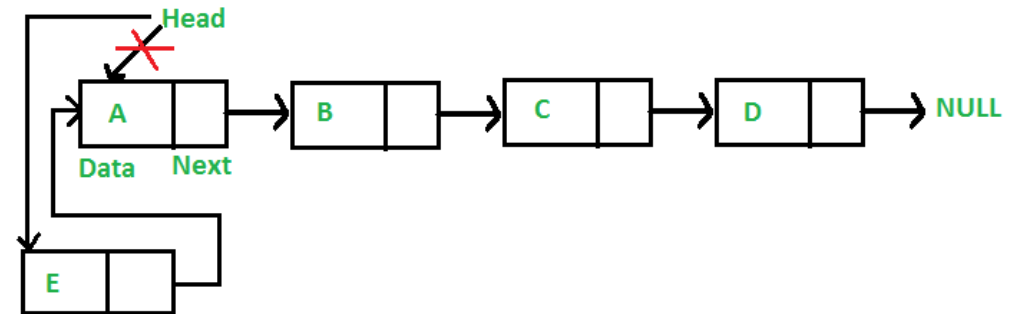
Check the top answer(answer with most votes) in this [LINK](#) to see why head is passed as a double pointer.

```
void insert_front(struct Node** head, int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head);

    /* 4. move the head to point to the new node */
    (*head) = new_node;
}
```

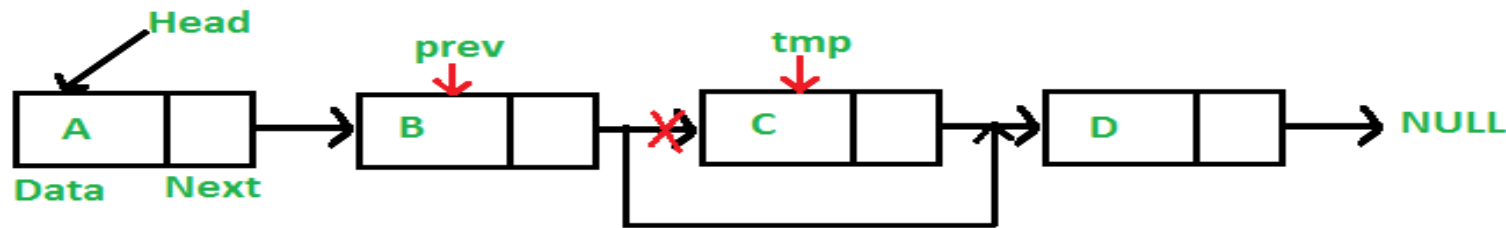


Deleting a node

Given a 'key', delete the first occurrence of this key in linked list.

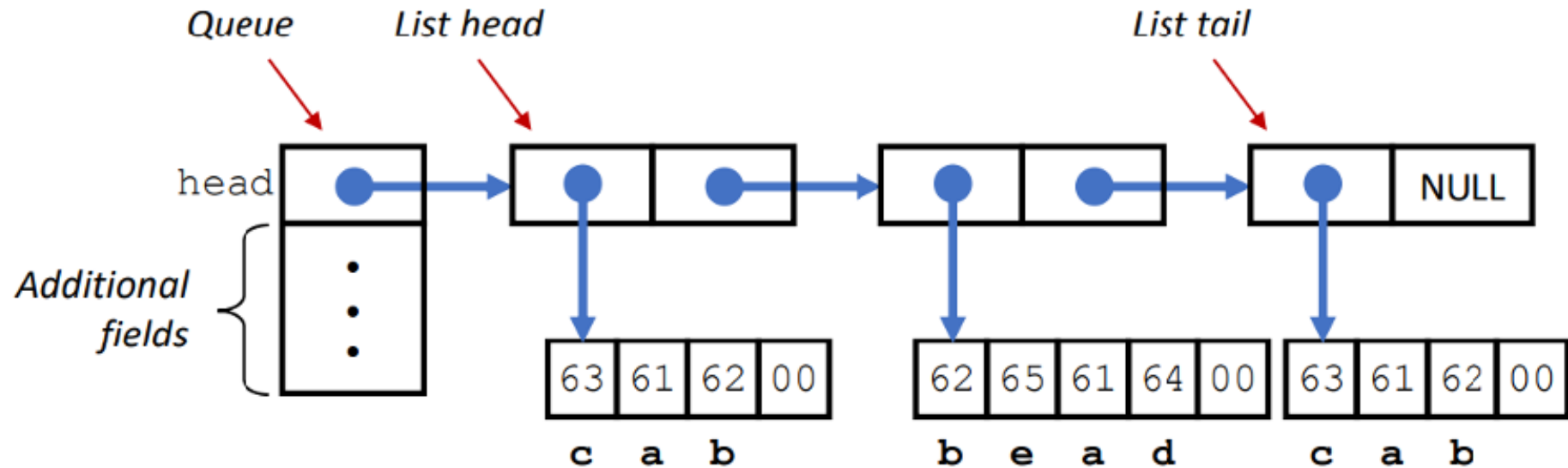
To delete a node from linked list, we need to do following steps.

- 1) Find previous node of the node to be deleted.
- 2) Change the next of previous node.
- 3) Free memory for the node to be deleted.

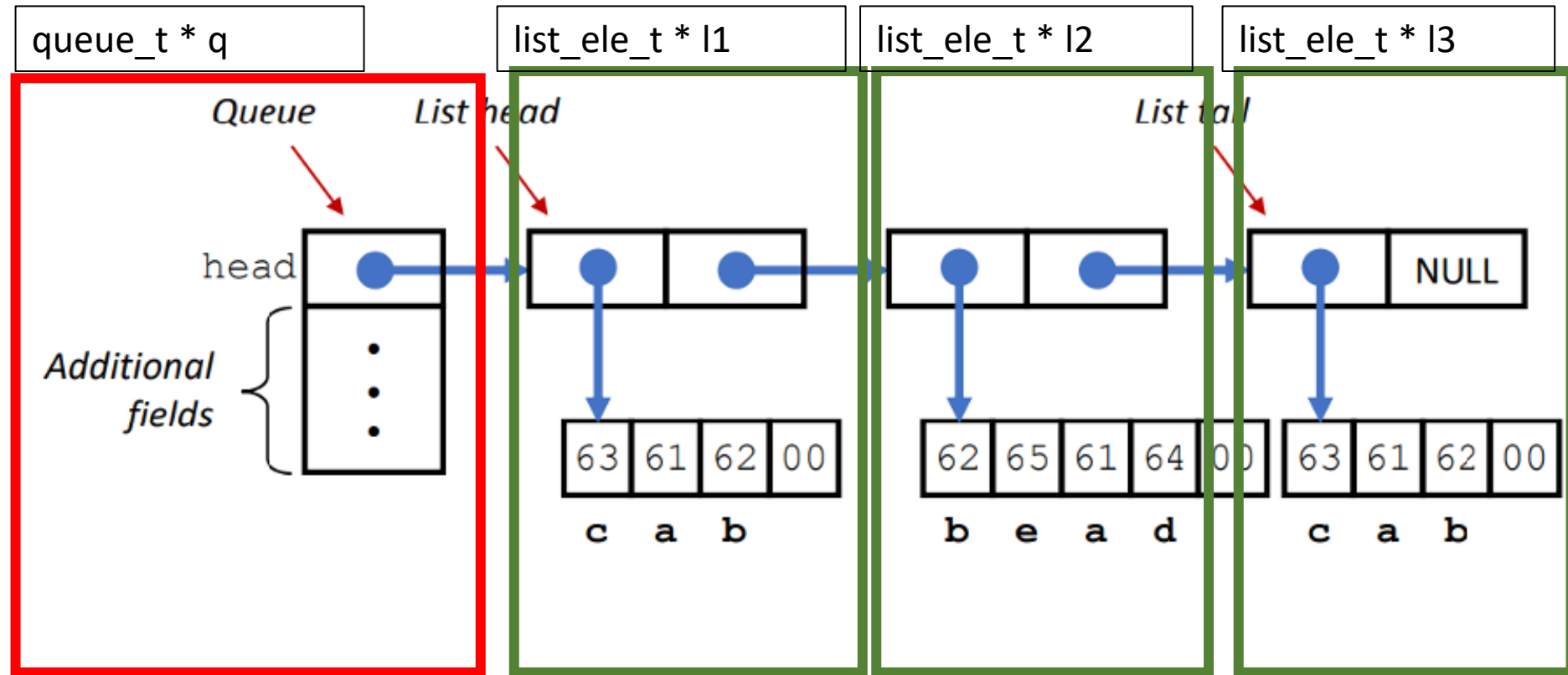


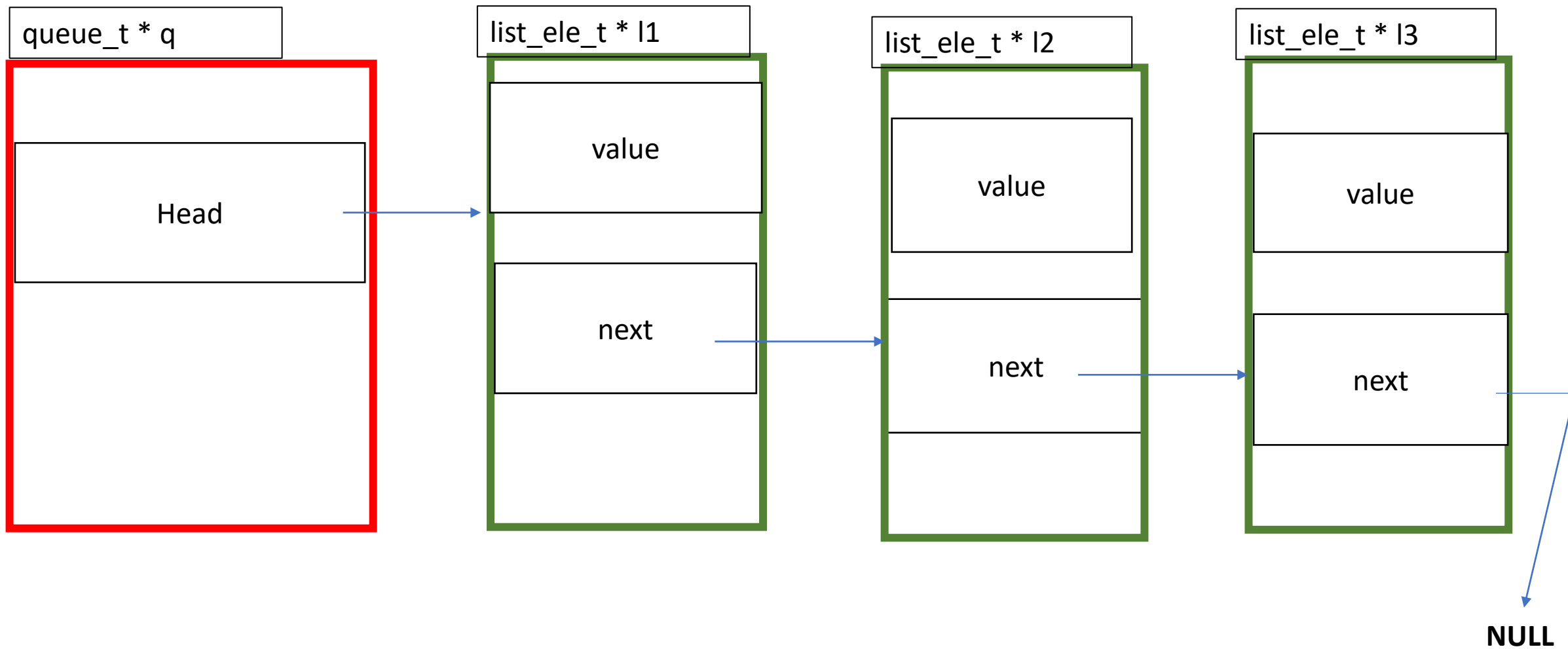
Queue Lab

```
/* Linked list element */ typedef struct ELE {  
    char *value;  
    struct ELE *next;  
} list_ele_t;  
  
/* Queue structure */ typedef struct {  
    list_ele_t *head; /* Linked list of elements */  
} queue_t;
```



Queue Lab





Now, do the linked list implementations

