Linked list as a stack - pop an element

```
int pop(struct node** headRef);
```

Note the **headRef indicating a pointer to a pointer, this is needed so you can change the head to point to the second element after the first is popped.

Return the data of the element you popped.

Remember to free the memory of the node you just popped.

POP THE FIRST ELEMENT IN THE LINKED LIST

```
int pop(struct node** headRef) {
  assert(headRef != NULL);

  if (*headRef != NULL) {
    printf("Freeing: %d", (*headRef)->data);
    struct node* next = (*headRef)->next;

    int data = (*headRef)->data;
    (*headRef)->next = NULL;
    free(*headRef);

    *headRef = next;

    return data;
  }

  assert(0);
}
```

FAIL IF HEADREF IS NULL, THIS IS PRESENT IN THE ASSERT.H LIBRARY

IF THERE IS A HEAD ELEMENT ONLY THEN POP IT, OTHERWISE ASSERT ERROR OUTSIDE THE IF-BLOCK

STORE THE NEXT POINTER SO WE HAVE A REFERENCE TO THE NEW HEAD

STORE THE DATA SO WE CAN RETURN IT

CLEAR UP THE MEMORY OF THE ELEMENT WE JUST POPPED

Delete all the elements in the linked list

```
void delete_list(struct node** headRef);
```

Note the **headRef indicating a pointer to a pointer, this is needed so you can change the head to point to null once the list is deleted

Remember to free the memory of all the nodes

DELETE ALL ELEMENTS IN THE LINKED LIST

```
void delete_list(struct node** headRef) {
  assert(headRef != NULL);

  struct node* head = *headRef;
  struct node* next = NULL;

  while (head != NULL) {
    printf("Freeing: %d", head->data);
    next = head->next;
    head->next = NULL;
    free(head);

    head = next;
  }
}
```

FAIL IF HEADREF IS NULL, THIS IS PRESENT IN THE ASSERT.H LIBRARY

THIS KEEPS TRACK OF THE NEXT ELEMENT AS WE FREE THE CURRENT ONE

STORE THE NEXT POINTER SO WE HAVE A REFERENCE TO WALK THE LIST

CLEAR UP THE MEMORY OF THE ELEMENT WE DELETE

MOVE THE HEAD POINTER TO THE NEXT ELEMENT

Insert an element at the nth position in a list

```
void insert_nth(struct node** headRef, int data, int n);
```

Note the **headRef indicating a pointer to a pointer, this is needed if you insert at the 0th element and need to update the head

Allocate memory for a new node to hold the data

If n > length of the list, append the element at the end of the list

INSERT ELEMENT AT THE nTH POSITION IN THE LIST

```
void insert_nth(struct node** headRef, int data, int n) {
  assert(headRef != NULL);
  assert(n >= 0);

  if (n == 0 || *headRef == NULL) {
    struct node* headNode = create_node(data);
    headNode->next = *headRef;
    *headRef = headNode;
    return;
  }

  int index = 0;
  struct node* head = *headRef;
  struct node* prev = *headRef;

  while (head != NULL && index < n) {
    prev = head;
    head = head->next;
    index++;
  }

  if (prev != NULL) {
    prev->next = create_node(data);
    prev->next->next = head;
  }
}
```

FAIL FOR THE CASES THAT YOU DO NOT HANDLE

INSERTING AT THE 0TH POSITION INVOLVES MOVING THE HEAD POINTER

CREATE_NODE IS A HELPER METHOD WHICH ALLOCATES MEMORY FOR A NODE, INITIALIZES ITS DATA AND RETURNS IT

WALK THE LIST TILL THE RIGHT INDEX POSITION IS FOUND, PREV KEEPS TRACK OF THE PREVIOUS POSITION TO SET UP THE LINKS CORRECTLY

CREATE A NODE AND INSERT BETWEEN PREV AND HEAD

Insert an element at the right position in a sorted

```
void sorted_insert(struct node** headRef, int data);
```

Note the **headRef indicating a pointer to a pointer, you might need to change the reference of the head of the list if the element belongs at the beginning.

Assume the list is sorted in ascending order.

Example:

If the list looks like this 1->3->5->7->NULL and you insert 4, then the list should become 1->3->4->5->7->NULL

INSERT ELEMENT AT THE RIGHT POSITION IN A SORTED LIST

```
void sorted_insert(struct node** headRef, int data) {
  assert(headRef != NULL);

  struct node* head = *headRef;
  struct node* prev = NULL;

  while (head != NULL && data > head->data) {
    prev = head;
    head = head->next;
  }

  struct node* newNode = create_node(data);

  if (prev != NULL) {
    prev->next = newNode;
  } else {
    *headRef = newNode;
  }

  newNode->next = head;
}
```

SET UP TWO POINTERS WHICH TRAVERSE THE LIST ONE BEHIND THE OTHER, PREV IS ALWAYS EXACTLY ONE ELEMENT BEHIND HEAD

ALONG WITH THE USUAL NULL CHECK, CHECK THAT THE DATA TO BE INSERTED IS LARGER THAN THE CURRENT ELEMENT, KEEP WALKING THE LIST TILL THE RIGHT POSITION IS FOUND

PREV POINTS TO THE ELEMENT JUST BEFORE WHERE THE NEW NODE IS TO BE INSERTED

PREV BEING NULL MEANS THE LIST IS EMPTY OR THE NODE BELONGS AT THE BEGINNING OF THE LIST

INSERT THE NODE BETWEEN PREV AND HEAD