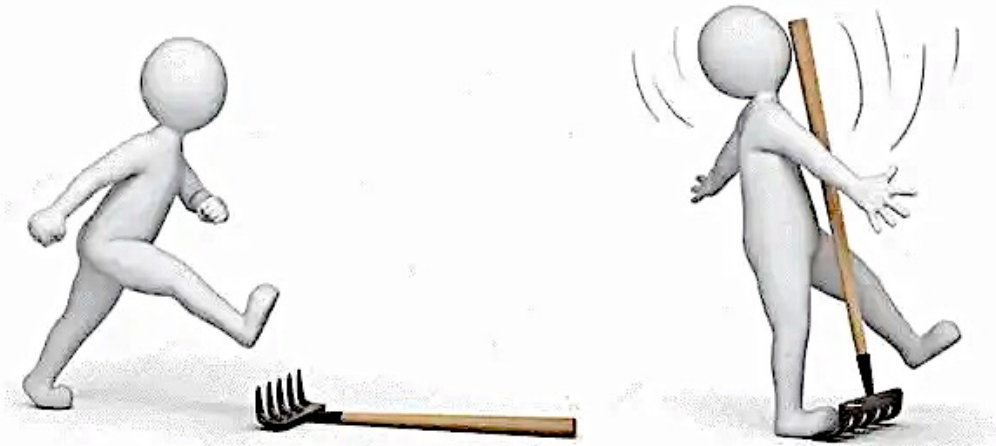
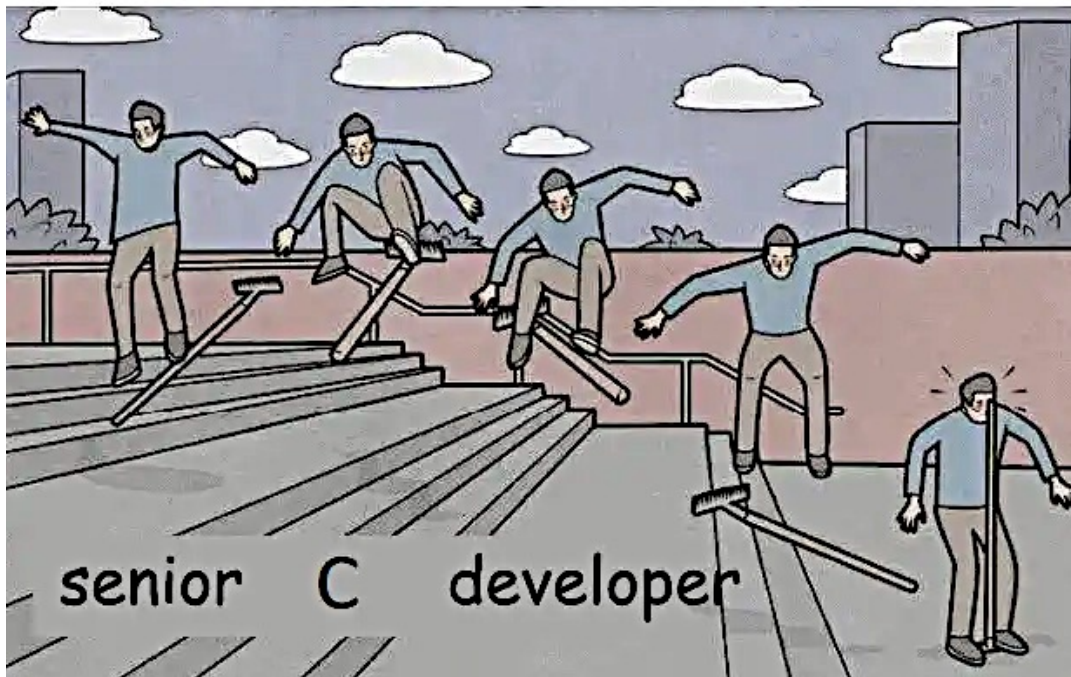
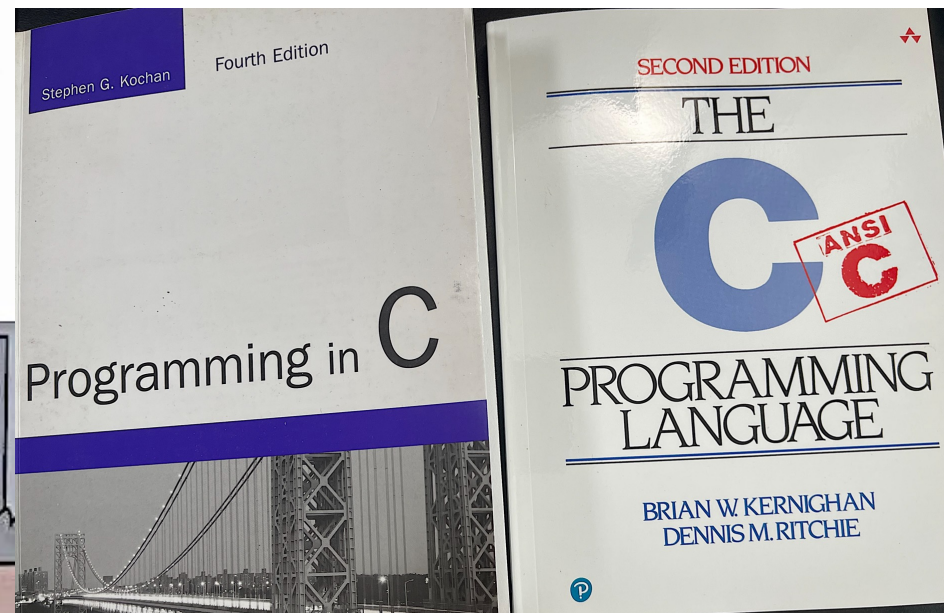


Lecture 13: C part VI



junior C developer

Suggested Literature



senior C developer

RECAP: Pointers

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

```
int z = 10;
```

```
void test(int** ptr)
```

```
{  
    *ptr = &z;  
}
```

The address of
the pointer to int
is passed by
value

```
int main ()
```

```
{  
    int x = 4;  
    int* y = &x;  
    printf("x = %d\n", x);  
    printf("&x = %p\n", (void*) &x);  
    printf(" y = %p\n", (void*) y);  
  
    test(&y);  
    printf("x = %d\n", x);  
    printf("&x = %p\n", (void*) &x);  
    printf(" y = %p\n", (void*) y);  
    printf("&z = %p\n", (void*) &z);  
}
```

```
whovian@phy504:~/host/Phy504Docker/C$ gcc poi
```

```
whovian@phy504:~/host/Phy504Docker/C$ ./point
```

```
x = 4
```

```
&x = 0x400180537c
```

```
y = 0x400180537c
```

```
x = 4
```

```
&x = 0x400180537c
```

```
y = 0x4000004010
```

```
&z = 0x4000004010
```



What if we need to
modify the pointer
function argument?
We need pointers of
pointers!

RECAP: Double Pointers

```
#include <stdio.h>

int main()
{
    int a = 5;
    int* ptr = &a;
    int** d_ptr = &ptr;

    printf("Size of normal Pointer: %li \n",
        sizeof(ptr));

    printf("Size of Double Pointer: %li \n",
        sizeof(d_ptr));

    return 0;
}
```

int* and **int****
have the same size;
they are both
addresses

```
whovian@phy504:~/host/C$ gcc pointers18.c -o pointers18
whovian@phy504:~/host/C$ ./pointers18 8 bytes = 64 bits =
Size of normal Pointer: 8 up to 16 digits in
Size of Double Pointer: 8 Hex
```

RECAP: **malloc** & **calloc** on **int****

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

int main ()
{
    const int nrows = 10;
    const int ncols = 10;
    double** x = (double**) malloc(sizeof(double*)*nrows);
    if (x == NULL)
    {
        printf("Error in memory: memory exhaustion \n");
        exit(1);
    }
    for (int i=0; i<nrows; i++)
    {
        x[i] = (double*) calloc(ncols, sizeof(double));
        if (x[i] == NULL)
        {
            printf("Error in memory: memory exhaustion \n");
            exit(1);
        }
    }
    for (int i=0; i<nrows; i++)
    {
        for(int j=0; j<ncols; j++)
        {
            printf("x[%d,%d] = %.2f", i, j, x[i][j]);
        }
        printf("\n");
    }
    for (int i=0; i<nrows; i++)
    {
        free(x[i]);
    }
    free(x);
    return 0;
}
```

In this common way: we need to **malloc** each dimension separately (with the **for loop**)

When deallocating memory with **free**, we also need to free each dimension separately

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

```
int main ()
```

```
{
```

```
    const int nrows = 3;
```

```
    const int ncols = 3;
```

```
    const int len = sizeof(double*)*nrows + sizeof(double)*ncols*nrows;
```

```
    double** x = (double**) malloc(len);
```

```
    if (x == NULL)
```

```
    {
```

```
        printf("Error in memory: memory exhaustion \n");
```

```
        exit(1);
```

```
    }
```

```
    for (int i=0; i<nrows; i++)
```

```
    { // for loop to point rows pointer to right loc in 2D array
```

```
        x[i] = ((double*)(x + nrows) + ncols*i);
```

```
    }
```

```
    for (int i=0; i<nrows; i++)
```

```
    {
```

```
        for(int j=0; j<ncols; j++)
```

```
        {
```

```
            x[i][j] = i + j;
```

```
            printf("x[%d,%d] = %.2f ", i, j, x[i][j]);
```

```
        }
```

```
        printf("\n");
```

```
    }
```

```
    free(x);
```

```
    return 0;
```

```
}
```

 **pointer
arithmetic**

**malloc &
calloc** on
double
pointers

Solution 2
(awesome)
Memory is
contiguous

```
whovian@phy504:~/host/Phy504Docker/C$ gcc pointers16.c -o pointers16
```

```
whovian@phy504:~/host/Phy504Docker/C$ ./pointers16
```

```
x[0,0] = 0.00 x[0,1] = 1.00 x[0,2] = 2.00
```

```
x[1,0] = 1.00 x[1,1] = 2.00 x[1,2] = 3.00
```

```
x[2,0] = 2.00 x[2,1] = 3.00 x[2,2] = 4.00
```

Pointer Arithmetic

```
#include <stdio.h>

// pointers are incremented and decremented
// by the size of the data type they point to
int main()
{
    int a = 22;
    int* p = &a;
    printf("p = %p\n", (void*) p);
    printf("p++ = %p\n\n", (void*) ++p); // +4bytes
                                         // = +16bits
                                         // = +0x4

    double b = 22.22;
    double* q = &b;
    printf("q = %p\n", (void*) q);
    printf("q++ = %p\n\n", (void*) ++q); // +8 bytes
                                         // = +32bits
                                         // = +0x8

    char c = 'a';
    char* r = &c;
    printf("r = %p\n", r);
    printf("r++ = %p\n", (void*) ++r); // +1 byte
                                         // = +4bits
                                         // = +0x1

    return 0;
}
```

Pointer arithmetic is super useful to speed-up access to elements stored on contiguous containers

```
whovian@phy504:~/hos
p = 0x4001805384
p++ = 0x4001805388

q = 0x4001805388
q++ = 0x4001805390

r = 0x4001805383
r++ = 0x4001805384
```

Source:

<https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/>

Pointer Arithmetic

```
#include <stdio.h>

int main()
{
    int x[10] = {1,4,6,8,10,12,14,16,18,20};
    int* ptr1 = &x[0]; // address of the first elem

    printf("x[%i] = %i \n", 4, *(ptr1+3)); // 4th elem

    int *ptr2 = &x[9];
    printf("proof x mem is contiguous: 0x%lx\n",
           (ptr2 - ptr1)*sizeof(int)); // 9 * 4 = 36 (dec)
                                       // = 24 in Hex

    printf("proof x mem is contiguous (dec): %ld\n",
           (ptr2 - ptr1)*sizeof(int)); // 9 * 4 = 36 (dec)
                                       // = 24 in Hex

    return 0;
}
```

```
whovian@phy504:~/host/C$ ./ptrarith2
x[4] = 8
proof x mem is contiguous: 0x24
proof x mem is contiguous (dec): 36
```

Pointer arithmetic is super useful to speed-up access to elements stored on contiguous containers

Source:

<https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/>

Function Pointers

```
#include <stdio.h>

// Source:
// www.geeksforgeeks.org/function-pointer-in-c/

void fun(int a)
{
    printf("Value of a is %d\n", a);
}

int main()
{
    // fun_ptr is a pointer to function fun()
    void (*fun_ptr)(int) = &fun;

    /* above line is equival of following two
    void (*fun_ptr)(int);
    fun_ptr = &fun;
    */

    (*fun_ptr)(10); // Invoking fun() using fun_ptr
    return 0;
}
```

Parentheses are mandatory



```
whovian@phy504:~/host/C$ gcc pointers19.c -o pointers19
whovian@phy504:~/host/C$ ./pointers19
Value of a is 10
```

- Like normal data pointers, we can have pointers to functions
- Unlike normal pointers, we do not allocate or deallocate memory using function pointers.
- Unlike normal pointers, a function pointer points to code, not data. Function pointer stores the starting memory of the executable code.

Source:

<https://www.geeksforgeeks.org/function-pointer-in-c/>

Function Pointers

Like normal pointers, we can have an array of function pointers.

```
#include <stdio.h>

void add(int a, int b)
{
    printf("Addition is %d\n", a+b);
}

void subtract(int a, int b)
{
    printf("Subtraction is %d\n", a-b);
}

void multiply(int a, int b)
{
    printf("Multiplication is %d\n", a*b);
}

int main()
{ // array of function pointers
    void (*fun_ptr_arr[])(int, int) =
        {add, subtract, multiply};

    int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for "
           "subtract and 2 for multiply\n");

    scanf("%i", &ch);
    if (ch > 2)
    {
        return 0;
    }
    (*fun_ptr_arr[ch])(a, b);
    return 0;
}
```



Source: <https://www.geeksforgeeks.org/function-pointer-in-c/>

```

#include <stdio.h>

void fun1()
{
    printf("Fun1\n");
}

void fun2()
{
    printf("Fun2\n");
}

// A function that receives a simple func
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
}

```

Function Pointers

Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

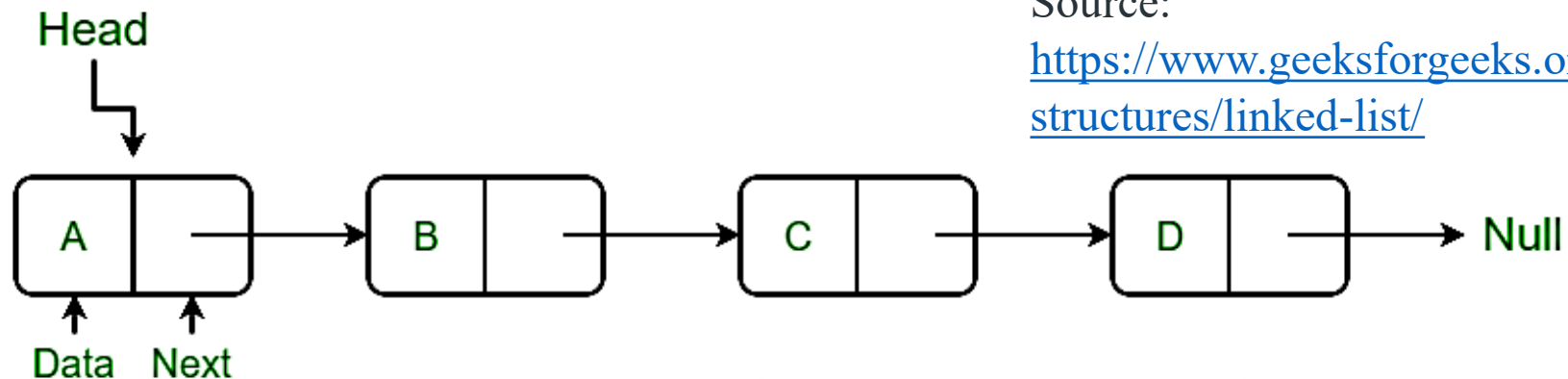
Source:

<https://www.geeksforgeeks.org/function-pointer-in-c/>

Linked List

```
// I wrote Node on the top as well so I can
// add a pointer to the same type (the next
// pointer). Without this, the compiler would have
// been lost
typedef struct Node
{ // A linked list node
  int data;
  // Here we need to use the word struct even
  // with typedef
  struct Node* next;
} Node;
```

A linked list is a linear data structure in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.



Source:

<https://www.geeksforgeeks.org/data-structures/linked-list/>

Linked List

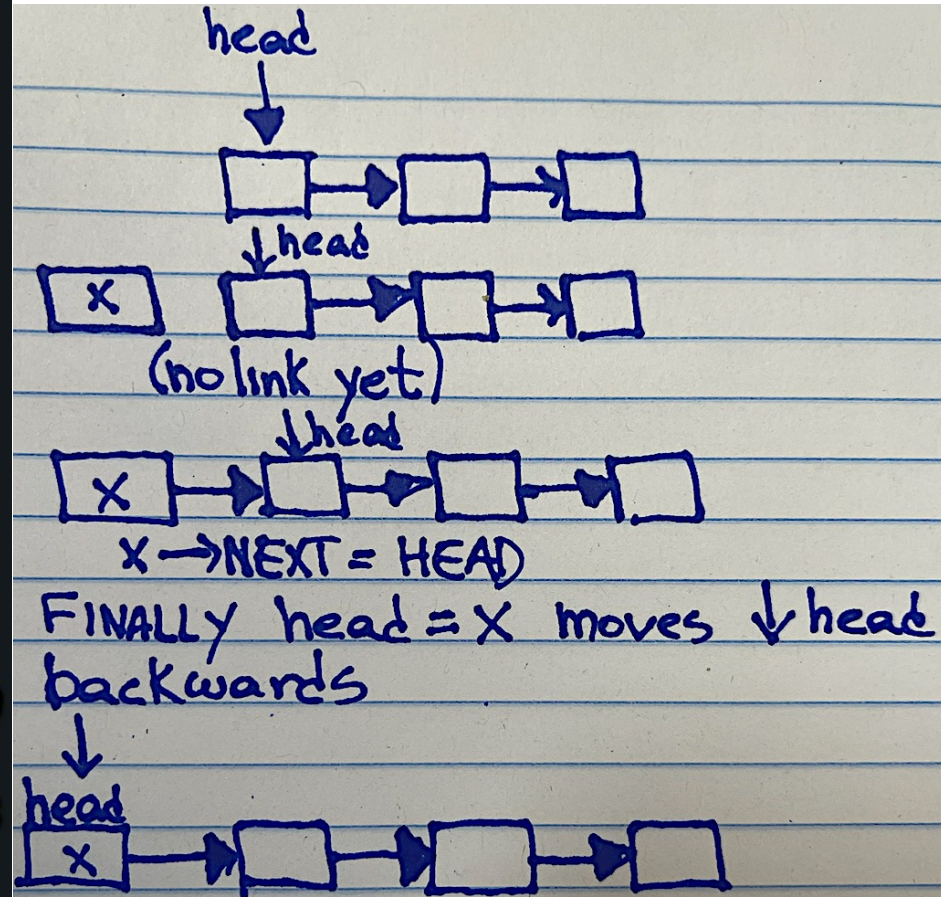
Adding element to the top of the LL

```
void push(Node** head_ref, int data)
{
    // 1. allocate node
    // Remember out class on pointers
    // Without malloc, Node x would be erased
    // when the function stack collapses.
    Node* x = (Node*) malloc(sizeof(Node));

    x->data = data; // 2. put in the data

    // 3. Next of x links to the original head
    x->next = (*head_ref);

    // 4. move the head to point to the new node
    // This is why we need pointer to pointer
    // We are modifying the original linked list
    (*head_ref) = x;
}
```



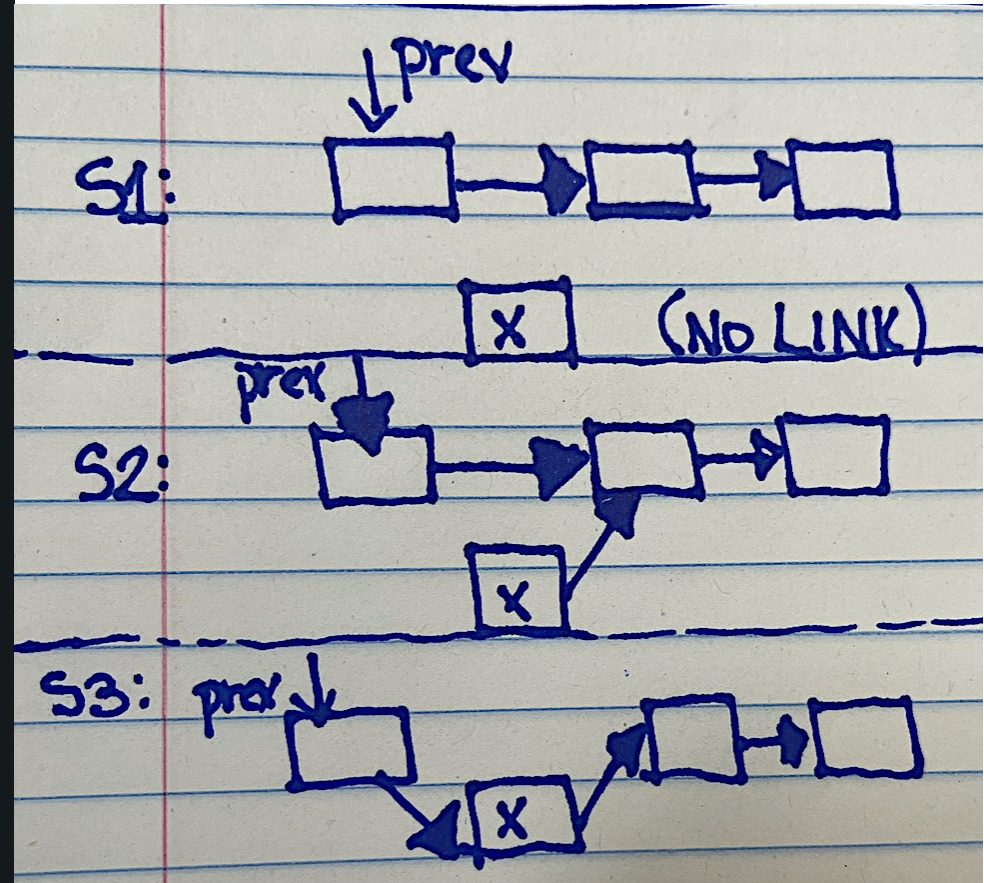
Linked List

Add a node after a given node

```
void InsertAfter(Node* prev_node, int data)
{
    if (prev_node == NULL)
    {
        printf("node cannot be NULL");
        return;
    }
    // allocate new node & write data
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;

    // Make next of new node as next of prev_node
    x->next = prev_node->next;

    // move the next of prev_node as new node
    prev_node->next = x;
}
```



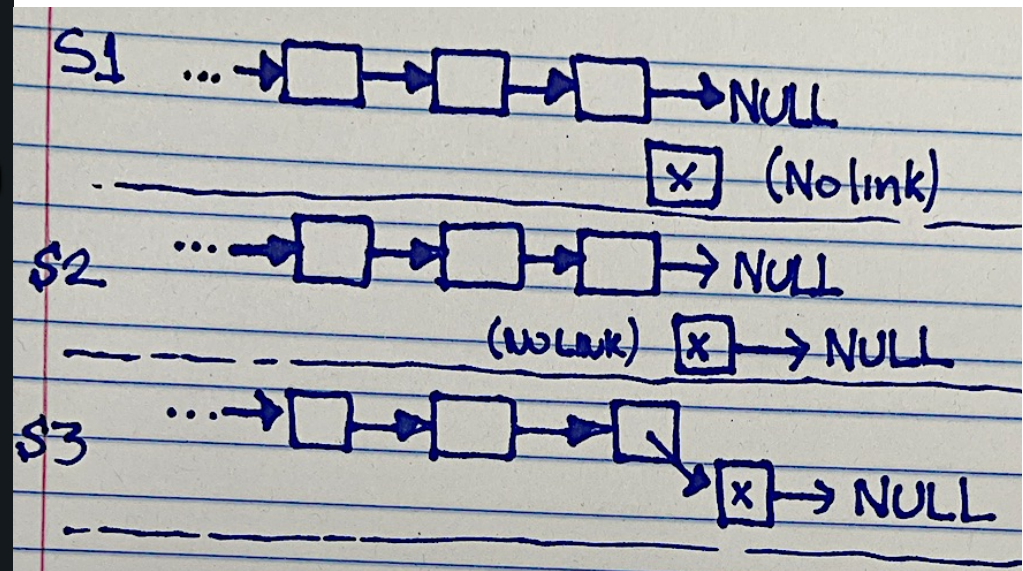
Linked List

Add a node at the end

```
void append(Node** head_ref, int data)
{
    // allocate node & write data to new node
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL; // this will be the last node

    if (*head_ref == NULL)
    { // if LL is empty, then make the new node as head
        *head_ref = x; // we modify the pointer that is
            // why the pointer to pointer arg
        return;
    }
    else
    {
        Node* last = *head_ref;

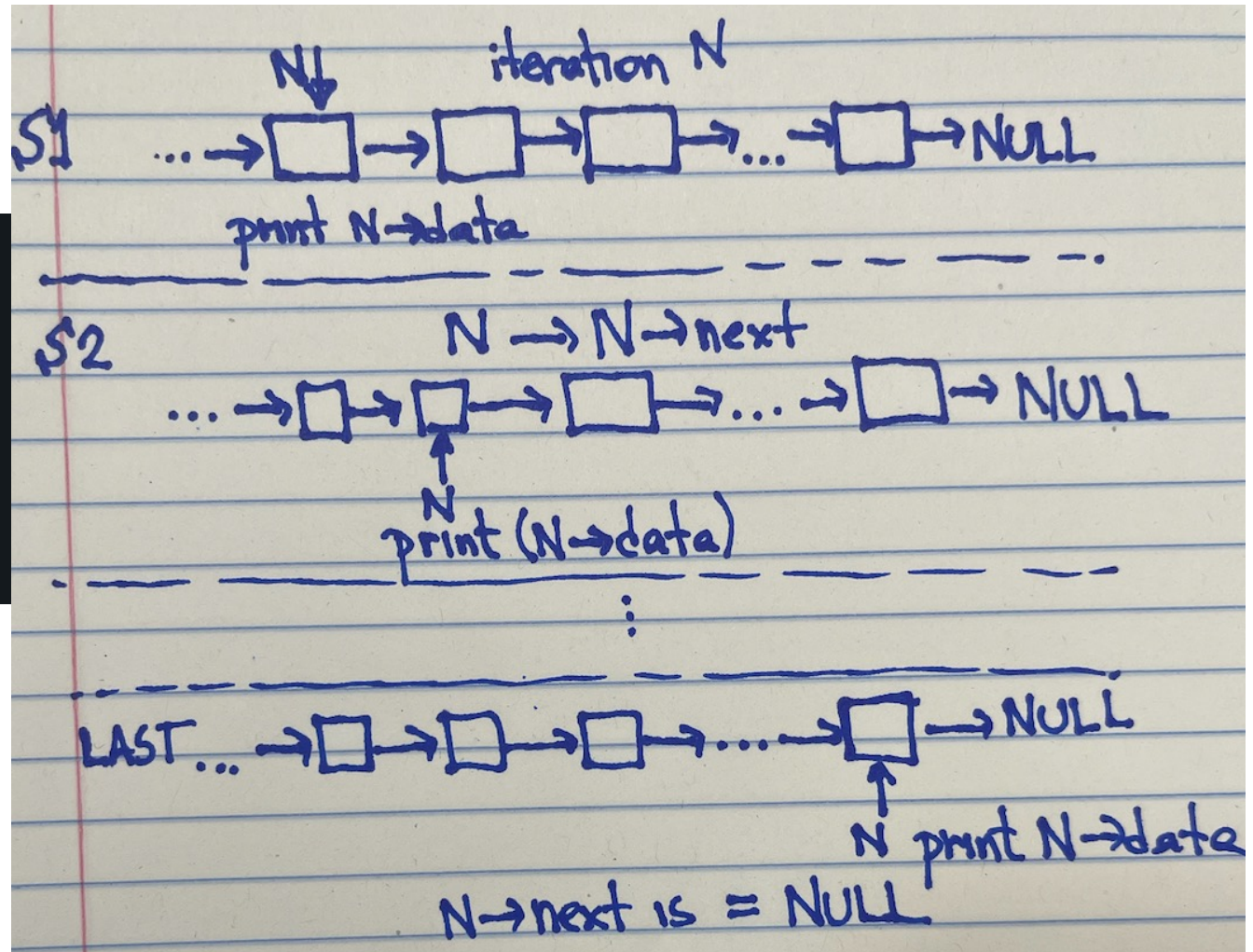
        while (last->next != NULL)
        { // traverse till the last node
            last = last->next;
        }
        last->next = x; // Change the next of last node
    }
}
```



Linked List

Showing all the functions that add elements in practice

```
void PrintList(Node* node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```



Linked List

Showing all the functions that add elements in practice

```
int main()
{
    // Start with the empty list
    Node* head = NULL;

    append(&head, 6); // linked list becomes 6->NULL
    push(&head, 7);   // linked list becomes 7->6->NULL
    push(&head, 1);   // linked list becomes 1->7->6->NULL
    append(&head, 4); // linked list becomes 1->7->6->4->NULL

    // linked list becomes 1->7->8->6->4->NULL
    // Remember that head->data = 1
    //                      head->next->data = 8
    InsertAfter(head->next, 8);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc linkedlist1.c -o linkedlist1
whovian@phy504:~/host/C$ ./linkedlist1
Created Linked list is: 1 7 8 6 4
```

Linked List

How to count the number of elements in a linked list

Because memory is non-contiguous, we can't access it via pointer arithmetic. To count elements, we must visit all nodes until we hit the **NULL** pointer.

```
int GetCount(Node* head)
{
    int count = 0;           // Initialize count
    Node* current = head;    // Initialize current

    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}
```

Source:

<https://www.geeksforgeeks.org/data-structures/linked-list/>

Linked List

```
bool search(Node* head, int x)
{
    Node* current = head;    // Init current

    while (current != NULL)
    {
        if (current->data == x)
        {
            return true;
        }
        current = current->next;
    }
    return false;
}
```

Check whether the value x is present in a linked list

Because memory is non-contiguous, we need to visit all nodes to check if the element exists

No pointer arithmetic possible to get where the next element will be

Source: <https://www.geeksforgeeks.org/data-structures/linked-list/>

Linked List

```
void lreverse(Node** head_ref)
{
    Node* prev = NULL;
    Node* current = *head_ref;
    Node* next = NULL;

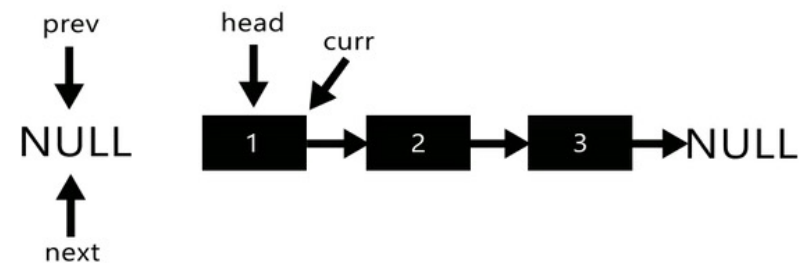
    while (current != NULL)
    {
        next = current->next; // store next
        // Reverse current node's pointer.
        // instead of pointing to the next node,
        // current will point to the prev node.
        // Next is not lost because we stored it.
        current->next = prev;

        // Move pointers one position ahead.
        prev = current;
        current = next;
    }
    *head_ref = prev; // current is null
}
```

How to reverse a linked list?

Source:

<https://www.geeksforgeeks.org/reverse-a-linked-list/>



```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

Linked List

```
int main()
{
    // Start with the empty list
    Node* head = NULL;

    append(&head, 6); // linked list becomes 6->NULL
    push(&head, 7);   // linked list becomes 7->6->NULL
    push(&head, 1);   // linked list becomes 1->7->6->NULL
    append(&head, 4); // linked list becomes 1->7->6->4->NULL

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    // now reverse it
    ltreverse(&head);
    printf("Reversed Linked list is: ");
    PrintList(head);
    printf("\n");

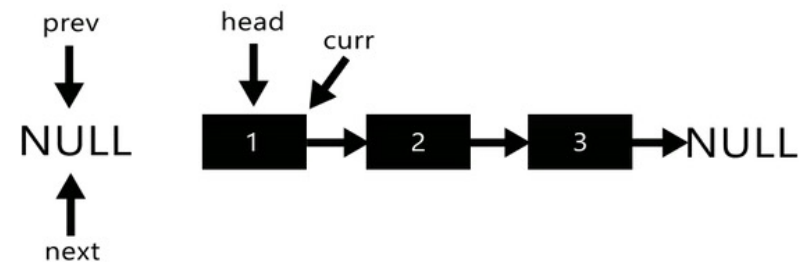
    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc linkedlist3.c -o linkedlist3
whovian@phy504:~/host/C$ ./linkedlist3
Created Linked list is: 1 7 6 4
Reversed Linked list is: 4 6 7 1
```

How to reverse a linked list?

Source:

<https://www.geeksforgeeks.org/reverse-a-linked-list/>

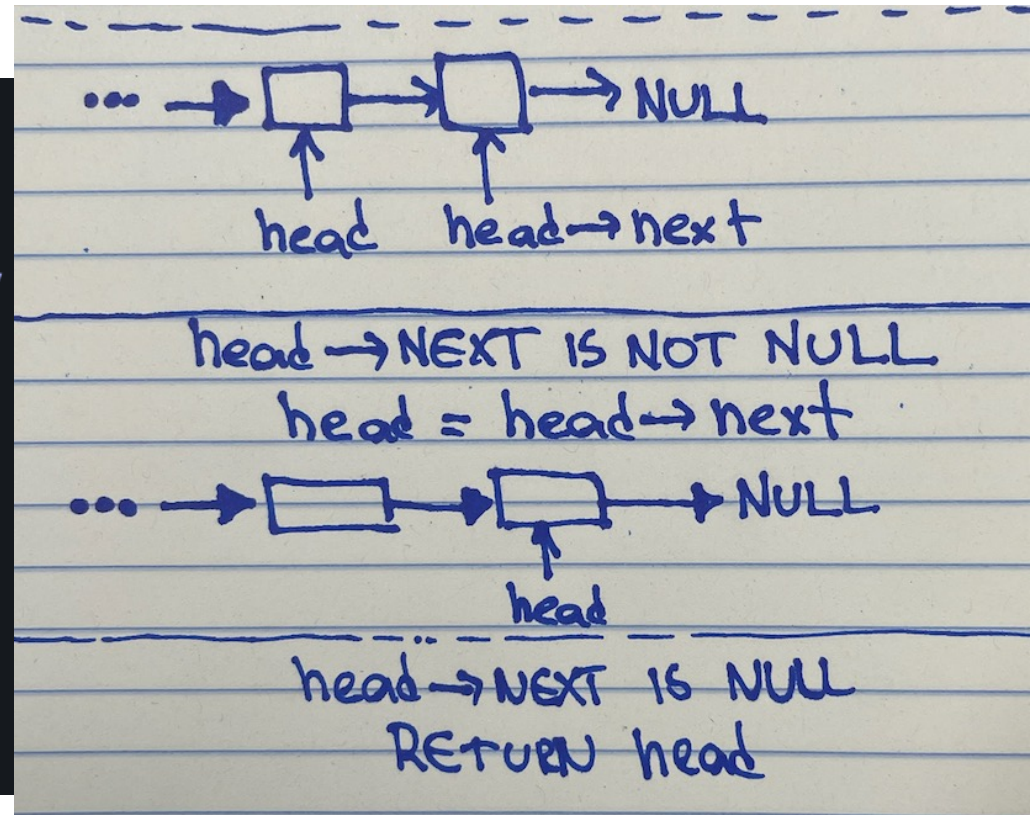


```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

Linked List

How to find last element a linked list?

```
Node* LastNode(Node* head)
{ // Finding last node of linked list
  if(head == NULL)
  { // in case the linked list is empty
    return head;
  }
  while (head->next != NULL)
  {
    head = head->next;
  }
  return head;
}
```



Source:

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>

Linked List

How do you swap consecutive elements on a linked list?

```
void swap(Node* a, Node* b)
{
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}
```

Source:

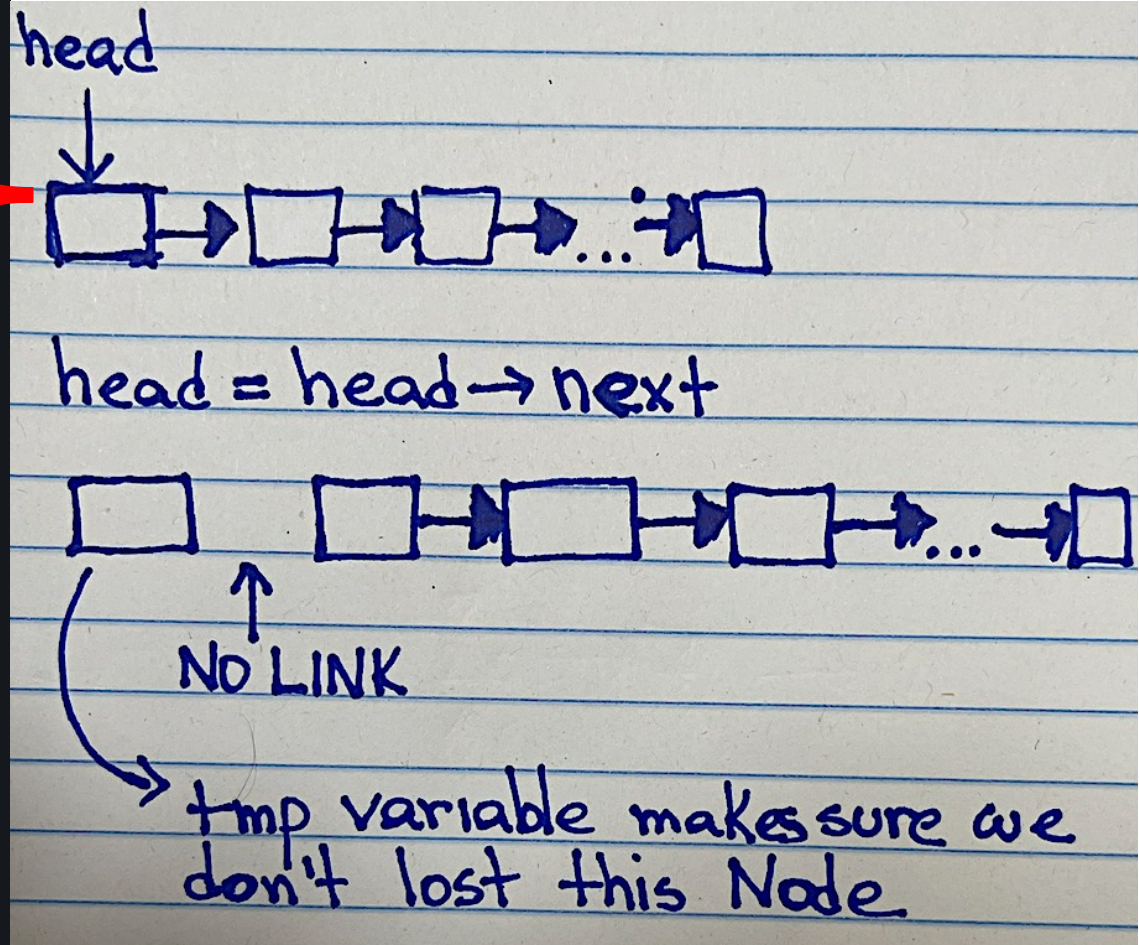
<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>

Linked List

Delete nodes in a linked tree?

```
void ldelete(Node** head_ref, int pos)
{
    assert(pos > 0); // post == 1, first pos
    pos--;
    Node* tmp = *head_ref;
    Node* prev = *head_ref;

    if (pos == 0)
    { // Del first element
        if(tmp != NULL)
        { // if num nodes == 0, nothing to del
            (*head_ref) = (*head_ref)->next;
            free(tmp);
        }
    }
    else
    { // pos > 1
        for (int i=0; i<pos; i++)
        {
            if (tmp == NULL)
            { // in case pos > number of nodes
                break;
            }
            prev = tmp;
            tmp = tmp->next;
        }
        if (tmp != NULL)
        {
            prev->next = tmp->next;
            free(tmp);
        }
    }
}
```



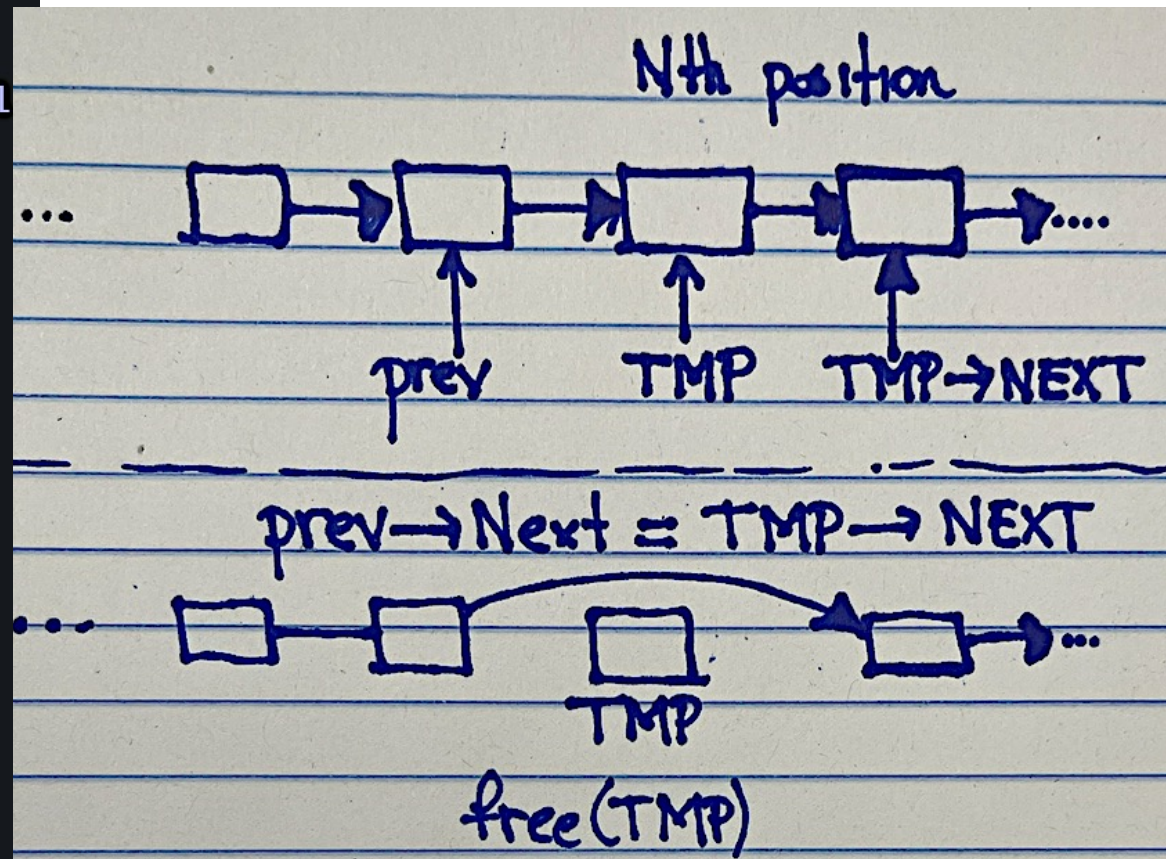
Source: <https://www.geeksforgeeks.org/deletion-in-linked-list/>

Linked List

Delete nodes in a linked tree?

```
void ldelete(Node** head_ref, int pos)
{
    assert(pos > 0); // post = 1, first pos
    pos--;
    Node* tmp = *head_ref;
    Node* prev = *head_ref;

    if (pos == 0)
    { // Del first element
        if(tmp != NULL)
        { // if num nodes == 0, nothing to del
            (*head_ref) = (*head_ref)->next;
            free(tmp);
        }
    }
    else
    { // pos > 1
        for (int i=0; i<pos; i++)
        {
            if (tmp == NULL)
            { // in case pos > number of nodes
                break;
            }
            prev = tmp;
            tmp = tmp->next;
        }
        if (tmp != NULL)
        {
            prev->next = tmp->next;
            free(tmp);
        }
    }
}
```



Source: <https://www.geeksforgeeks.org/deletion-in-linked-list/>

Linked List

How to delete nodes in a linked tree?

Source: <https://www.geeksforgeeks.org/deletion-in-linked-list/>

```
int main()
{
    Node* head = NULL;

    append(&head, 1);
    push(&head, 2);
    push(&head, 3);
    push(&head, 4);
    push(&head, 5);
    push(&head, 6);
    push(&head, 7);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    // now delete first element
    ldelete(&head, 1);
    PrintList(head);
    printf("\n");

    // now delete third element
    ldelete(&head, 3);
    PrintList(head);
    printf("\n");

    return 0;
}
```

```
whovian@phy504:~/host/C$ ./linkedlist6
Created Linked list is: 7 6 5 4 3 2 1
6 5 4 3 2 1
6 5 3 2 1
```

```

void RemoveDuplicates(Node* head)
{
    if (head == NULL)
    { // do nothing if the list is empty
        return;
    }
    while (head->next != NULL)
    {
        if (head->data == head->next->data)
        {
            Node* tmp = head->next;
            head->next = head->next->next;
            free(tmp);
        }
        else
        {
            head = head->next;
        }
    }
}

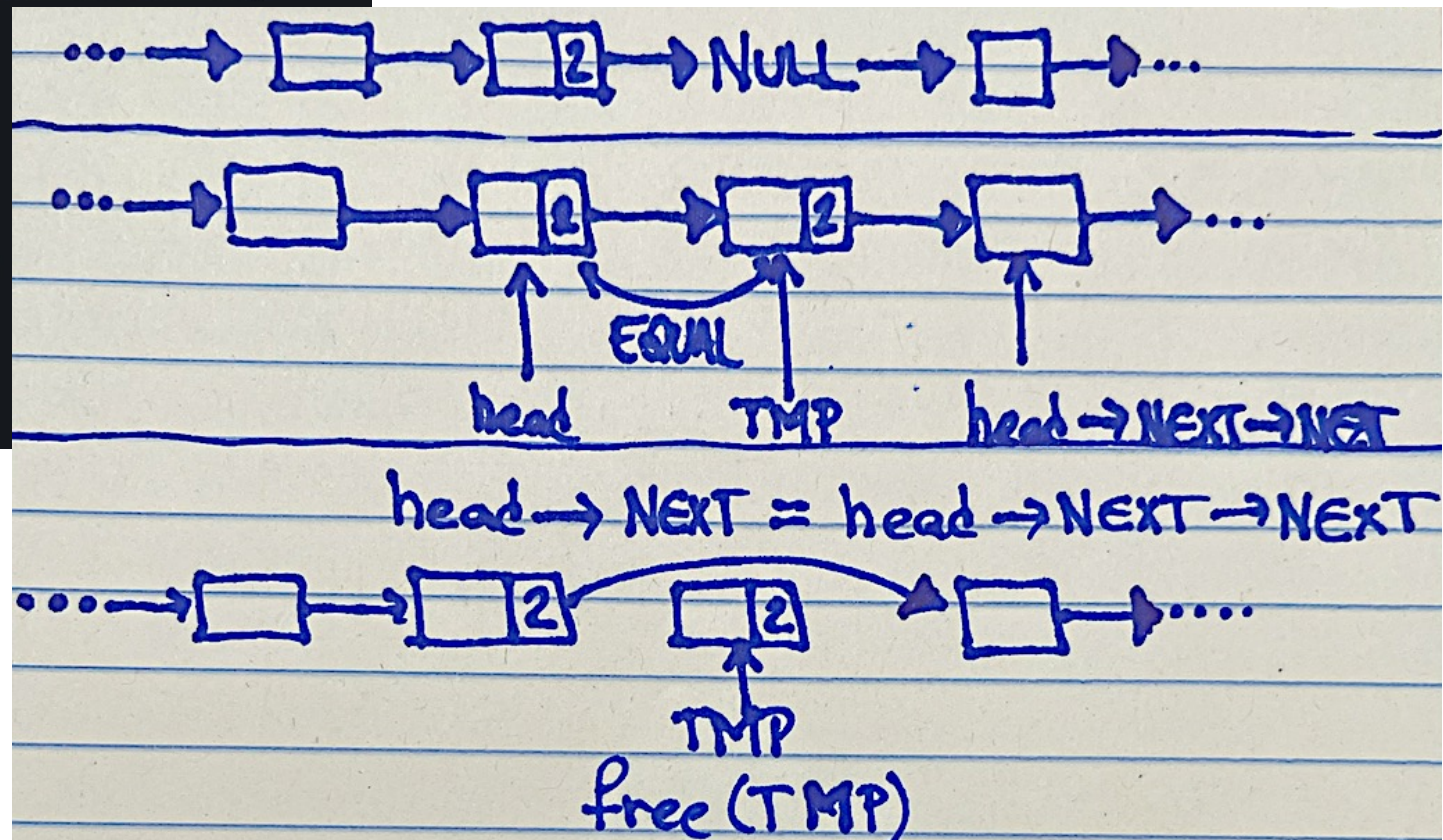
```

Linked List

How to remove duplicates
in a sorted linked tree?

Source:

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>



Linked List

How to remove duplicates in a sorted linked tree?

Source:

<https://www.geeksforgeeks.org/deletion-in-linked-list/>

```
int main()
{
    Node* head = NULL;

    append(&head, 6);
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);
    push(&head, 10);
    push(&head, 9);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    // now delete duplicates
    RemoveDuplicates(head);
    printf("Linked list w/ no duplicates: ");
    PrintList(head);
    printf("\n");

    return 0;
}
```

```
whovian@phy504:~/host/C$ gcc linkedlist7.c -o linkedlist7
whovian@phy504:~/host/C$ ./linkedlist7
Created Linked list is: 9 10 11 11 11 13 13 20 6
Linked list w/ no duplicates: 9 10 11 13 20 6
```

Linked Lists

How do you swap consecutive elements on a linked list?

```
void swap(Node* a, Node* b)
{
    int temp = a->data;
    a->data = b->data;
    b->data = temp;
}
```

Source:

<https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/>

Sort in Linked List

Repeat “Pass” until an entire iteration without any swaps

```
void sort(Node* head)
{
    if(head == NULL)
    {
        return;
    }

    int swapped;
    do
    {
        swapped = 0;
        Node* p0 = head;

        while (p0->next != NULL) Pass
        {
            if(p0->data > p0->next->data)
            {
                swap(p0, p0->next);
                swapped = 1;
            }
            p0 = p0->next;
        }
    } while (swapped == 1);
}
```

(5 1 4 2 8) -> (1 5 4 2 8), (1 4 2 5 8) -> (1 4 2 5 8)
(1 5 4 2 8) -> (1 4 5 2 8), (1 4 2 5 8) -> (1 2 4 5 8),
(1 4 5 2 8) -> (1 4 2 5 8), (1 2 4 5 8) -> (1 2 4 5 8)
(1 4 2 5 8) -> (1 4 2 5 8), (1 2 4 5 8) -> (1 2 4 5 8)

First Pass

Second Pass

(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)
(1 2 4 5 8) -> (1 2 4 5 8)

Third Pass

whovian@phy504:~/host/C\$./linkedlist8

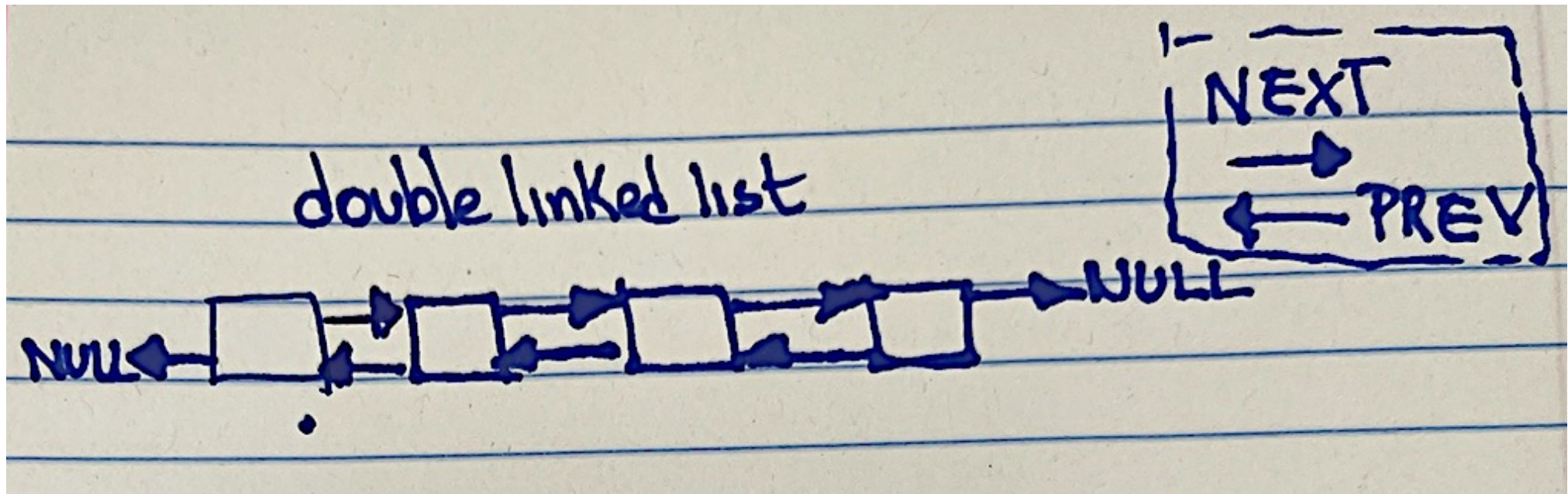
Created Linked list is: 6 10 2 5 7 50 3 1

Linked list w/ no duplicates: 1 2 3 5 6 7 10 50

Doubly Linked Lists

Double Linked list is a linear data structure in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers on both directions

Source: <https://www.geeksforgeeks.org/doubly-linked-list-tutorial/>



Doubly Linked Lists

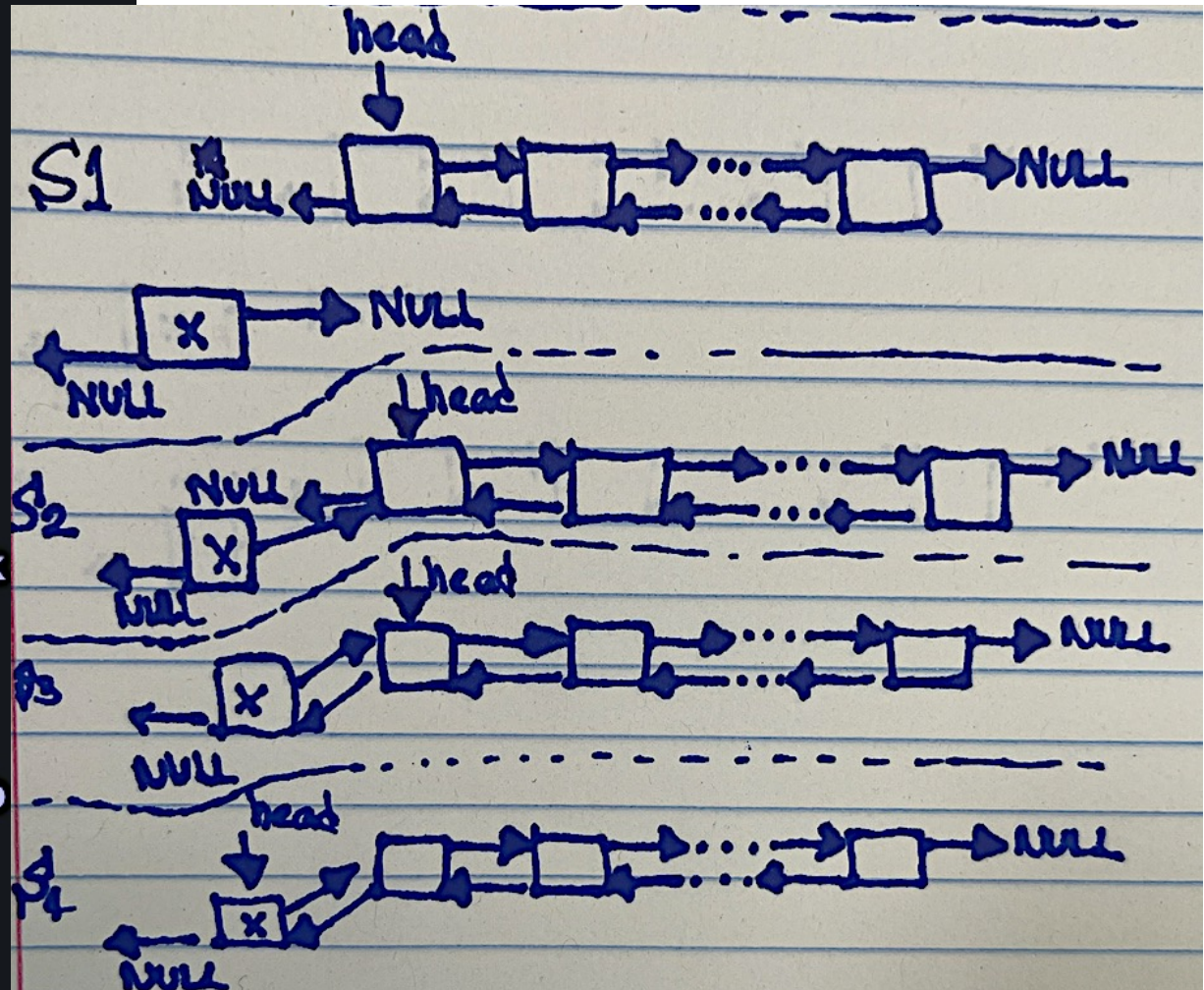
```
void push(Node** head_ref, int data)
{
    //1. create the new node
    Node* x =
        (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL;
    x->prev = NULL;

    //2. Next of x links to the
    //original head
    x->next = (*head_ref);

    if (*head_ref != NULL)
    {
        //3. prev of head links to x
        (*head_ref)->prev = x;
    }

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

Adding element to either at the top or end of the DLL



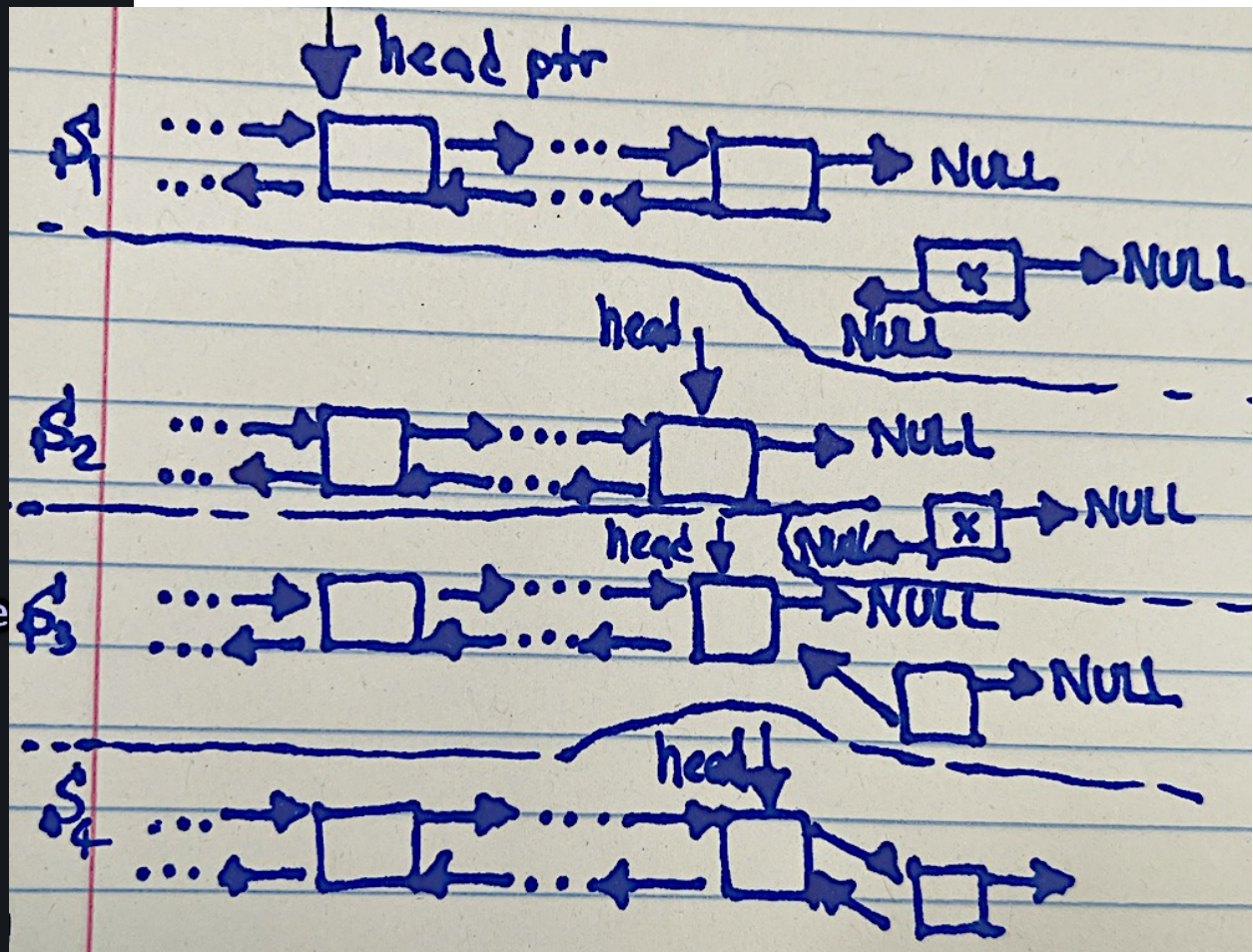
Doubly Linked Lists

Adding element to either at the top or end of the DLL

```
void append(Node** head_ref, int data)
{
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL;
    x->prev = NULL;

    if (*head_ref == NULL)
    {
        *head_ref = x;
        return;
    }
    else
    {
        Node* last = *head_ref;

        while (last->next != NULL)
        { // traverse till the last node
            last = last->next;
        }
        last->next = x;
        x->prev = last;
    }
}
```



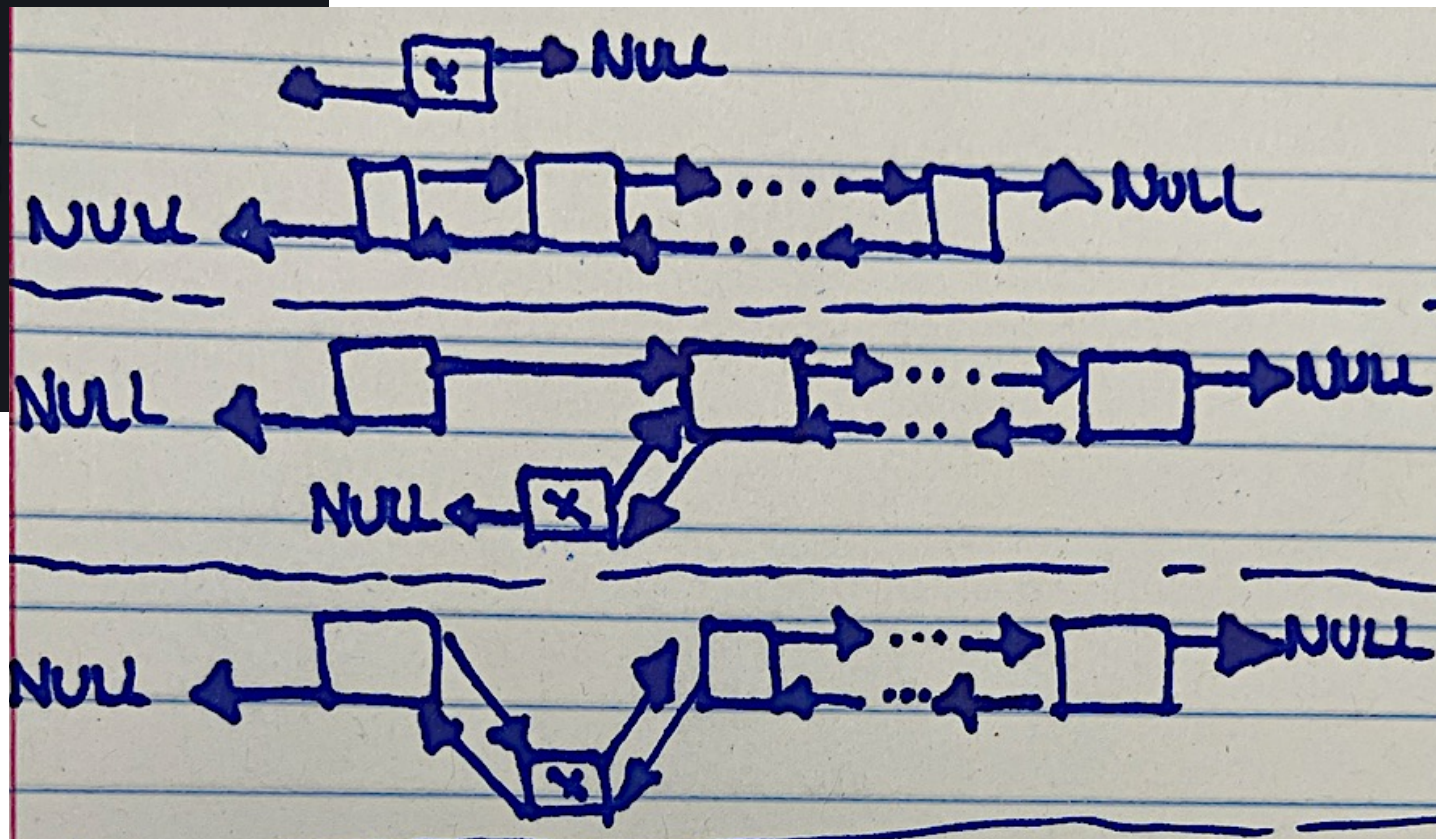
Doubly Linked Lists

```
void InsertAfter(Node* prev_node, int data)
{
    if (prev_node == NULL)
    {
        printf("node cannot be NULL");
        return;
    }
    Node* x = (Node*) malloc(sizeof(Node));
    x->data = data;
    x->next = NULL;
    x->prev = NULL;

    x->next = prev_node->next;
    prev_node->next->prev = x;

    prev_node->next = x;
    x->prev = prev_node;
}
```

Add a node after a given node



Doubly Linked Lists

Print all nodes with a possible rewind

```
void PrintList(Node* node)
{
    while (node != NULL &&
           node->prev != NULL)
    { // rewind list
        node = node->prev;
    }
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}
```

Doubly Linked Lists

```
int main()
{
    Node* head = NULL;

    append(&head, 6); // linked list becomes 6->NULL
    push(&head, 7);  // linked list becomes 7->6->NULL
    push(&head, 1);  // linked list becomes 1->7->6->NULL
    append(&head, 4); // linked list becomes 1->7->6->4->NULL

    // linked list becomes 1->7->8->6->4->NULL
    // Remember that head->data = 1
    //                      head->next->data = 8
    InsertAfter(head->next, 8);

    printf("Created Linked list is: ");
    PrintList(head);
    printf("\n");

    return 0;
}
```

All together

```
whovian@phy504:~/host/C$ gcc dbll1.c -o dbll1
whovian@phy504:~/host/C$ ./dbll1
Created Linked list is: 1 7 8 6 4
```

Homework I

- Write a program that demonstrates two facts:
 1. Arrays themselves are not passed to functions, but a pointer to the first element of the array is what is passed.
 2. Ints and floats are directly passed to functions.

Your output should be compelling evidence printed out to the terminal window (standard out). Note that you can use any type of array that you would like. Your program will compile with an appropriate warning if you're doing this correctly.

- According to the C standard, `arr[0]` is actually **syntactic sugar** for **`*(arr+0)`**. Write a program called `arraysugar.c` that prints all the elements of an integer array using this alternative notation.

Homework II

What is the output of the following program and why?

Does this program have a memory leak? Why?

```
#include <stdio.h>

int main()
{
    int *ptr, arr[] = {10, 20, 30, 40, 50};
    ptr = arr;
    *ptr = 3;
    ptr += 2;
    *ptr = 5;

    printf("Val = %d\n", arr[0]+arr[2]);

    return 0;
}
```

Homework III

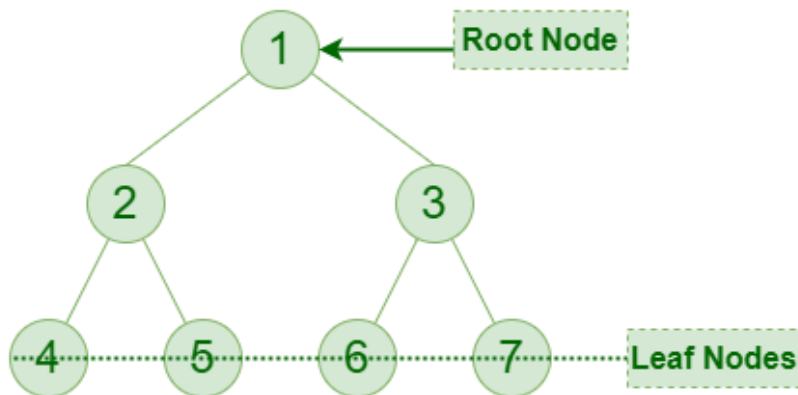
Write a C program that generalizes to a doubly linked list the following operations

- Delete nodes from a doubled linked list
- Reverse a double linked list
- Remove duplicates in a sorted double linked list
- Sort a double linked list

Homework IV

A binary tree is a tree data structure in which each node can have at most two children, which are referred to as the left child and the right child. The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves.

- Write the function **Node* NewNode(int data)** that creates a binary tree, allocates data to the node, and set left and right children to **NULL**



```
typedef struct Node {  
    int data;  
    struct Node* left;  
    struct Node* right;  
} Node;
```