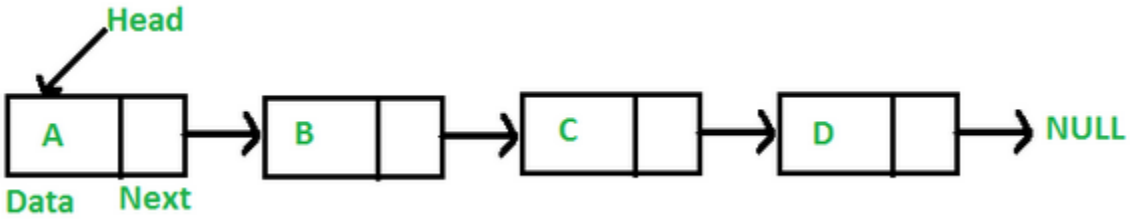
LINKED LIST

What is a linked list?

A linked list is an ordered collection of data in which each element contains the location of the next element or elements.

It is a linear collection of self-referential structures, called *nodes (a structure)*, connected by pointer links.



How to create a self-referential structure?

Self-referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

Example:

```
struct node{
        int data;
        struct node *next; /*points to a structure of type node*/
};
```

next is referred to as a link – i.e., can be used to "tie" a structure of type struct node to another structure of the same type.

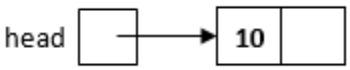
Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees.

How to allocate dynamic memory for the node?

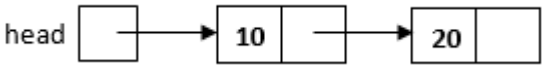
Assuming the following definition:

```
typedef struct node* Nodeptr;//to clean the code of the *
                             //use this when referring to the node pointer

typedef struct node{
    int data;
    Nodeptr next;
}Node;//use this name to refer to the node

Nodeptr head;//reference for the node
head = (Nodeptr) malloc(sizeof(Node));//dynamically allocate memory for the first node
head->data = 10;//store value to data
```

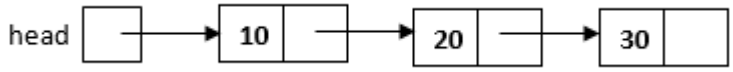
The first node is created and initialized and referred to by head. Here is the illustration:



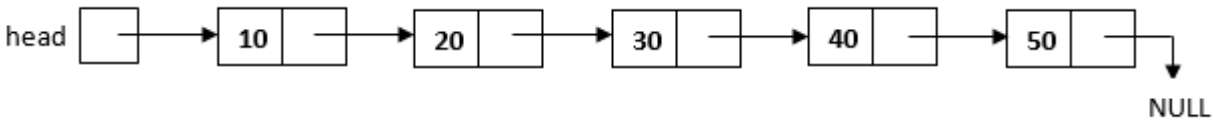

```
head->next = (Nodeptr) malloc(sizeof(Node));//creates the second node
head->next->data = 20;//stores value for the second node
```




```
head->next->next = (Nodeptr) malloc(sizeof(Node));//creates the third node
head->next->next->data = 30;//stores value for the third node
```




```
Nodeptr head;//reference for the node
head = (Nodeptr) malloc(sizeof(Node));//dynamically allocate memory for the first node
head->data = 10;//store value to data

head->next = (Nodeptr) malloc(sizeof(Node));//creates the second node
head->next->data = 20;//stores value for the second node

head->next->next = (Nodeptr) malloc(sizeof(Node));//creates the third node
head->next->next->data = 30;//stores value for the third node

head->next->next->next = (Nodeptr) malloc(sizeof(Node));//creates the fourth node
head->next->next->next->data = 40;//stores value for the fourth node

head->next->next->next->next = (Nodeptr) malloc(sizeof(Node));//creates the fifth node
head->next->next->next->next->data = 50;//stores value for the fifth node

head->next->next->next->next->next = NULL; //always set the last node pointer to NULL
```



As you can see, we now have a linked list.

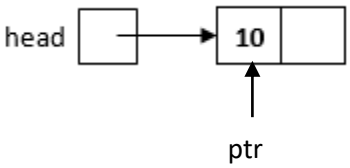
By convention, the link pointer in the last node of the list is set to NULL to mark the end of the list.

What consists a linked list?

1. root node (head), allocated on the stack (an ordinary C variable)
2. one or more records, allocated on the heap (by calling malloc)

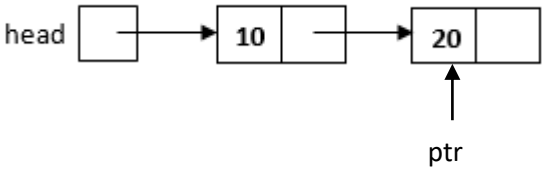
However, the problem with the above implementation is when you are going to create much more nodes for the list. This could be a bad implementation. To revise the code:

```
Nodeptr head;//reference to the first node; By convention, we always name it head
Nodeptr ptr; //use this as the node walker

head = (Nodeptr) malloc(sizeof(Node));//dynamically allocate memory for the first node
head->data = 10;//store value to data
ptr = head;//let ptr point to head; use head only to refer to the first node
```



Here, another pointer ptr is used. Ptr will be the walker that will walk from one node to the next. This is so since head should always be pointing to the first node.

```
ptr->next = (Nodeptr) malloc(sizeof(Node));//creates the second node; using ptr
ptr->next->data = 20;//stores value for the second node
ptr = ptr->next;//lets ptr move to the next node
```



After creating five nodes, here is now the code:

```
Nodeptr head;//reference to the first node; By convention, we always name it head
Nodeptr ptr; //use this as the node walker

head = (Nodeptr) malloc(sizeof(Node));//dynamically allocate memory for the first node
head->data = 10;//store value to data
ptr = head;//let ptr point to head; use head only to refer to the first node

ptr->next = (Nodeptr) malloc(sizeof(Node));//creates the second node; using ptr
ptr->next->data = 20;//stores value for the second node
ptr = ptr->next;//lets ptr move to the next node

ptr->next = (Nodeptr) malloc(sizeof(Node));//creates the third node
ptr->next->data = 30;//stores value for the third node
ptr = ptr->next;//lets ptr move to the next node

ptr->next = (Nodeptr) malloc(sizeof(Node));//creates the fourth node
ptr->next->data = 40;//stores value for the fourth node
ptr = ptr->next;//lets ptr move to the next node

ptr->next = (Nodeptr) malloc(sizeof(Node));//creates the fifth node
ptr->next->data = 50;//stores value for the fifth node
ptr = ptr->next;//lets ptr move to the next node

ptr->next = NULL; //always set the last node pointer to NULL
```

There are now repetitive statements. We can now modify the code with the use of loops.

Here is the simplified code using a loop:

```
int i;
Nodeptr head;
Nodeptr ptr;

head = (Nodeptr) malloc(sizeof(Node));
head->data = 10;
ptr = head;
for(i=2;i<=5;i++){
    ptr->next = (Nodeptr) malloc(sizeof(Node));
    ptr->next->data = i*10;
    ptr = ptr->next;
}
ptr->next = NULL;
```

To summarize, a linked list can be accessed via a pointer to the first node of the list; subsequent nodes are accessed via the link pointer member stored in each node.

Traverse a List

Since head always points to the first node, after creating the list, ptr is usually at the end. So to traverse, you initialize ptr back to where head is currently pointing.

Here is the code accessing the elements of the list.

```
ptr=head;
while(ptr!=NULL){
    printf("%d ",ptr->data);
    ptr = ptr->next;
}
```

Linked List to Functions

There is a little modification to the program. Another naming for the list so that to refer to the entire linked list, we use a different name from when naming the individual node pointer.

```
typedef Nodeptr List;//use List when creating the list
                     //use Nodeptr when referring to the individual node pointers

List head;//use List when referring to the entire list
Nodeptr ptr; //use Nodeptr for the individual node pointer
```

Here is the sample program:

list.h list.c

```c
typedef struct node* Nodeptr;//to clean the code of the *
                             //use this when referring to the node pointer

typedef struct node{
    int data;
    Nodeptr next;
}Node;//use this name to refer to the node

typedef Nodeptr List;//use List when creating the list
                     //use Nodeptr when referring to the individual node pointers

List createList();
void display(List head);
```

list.h list.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

List createList(){
    int i;
    List head;//use List when referring to the entire list
    Nodeptr ptr; //use Nodeptr for the individual node pointer
    head = (Nodeptr) malloc(sizeof(Node));
    head->data = 10;
    ptr = head;
    for(i=2;i<=10;i++){
        ptr->next = (Nodeptr) malloc(sizeof(Node));
        ptr->next->data = i*10;
        ptr = ptr->next;
    }
    ptr->next = NULL;
    return head;
}

void display(List head){
    Nodeptr ptr=head;
    while(ptr!=NULL){
        printf("%d ",ptr->data);
        ptr = ptr->next;
    }
}
```

list.h | list.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/* run this program using the console

int main(int argc, char *argv[]) {

    List head = createList();
    display(head);

    return 0;
}
```

More Traversal Examples

Let us define more examples for linked list traversal.

```c
int computeSum(List head);
int count(List head);
int countPos(List head);
int countNeg(List head);
int countEven(List head);
int countOdd(List head);

//computes the total of the elements
int computeSum(List head){
    int sum = 0;
    Nodeptr ptr=head;
    while(ptr!=NULL){
        sum+=ptr->data;
        ptr = ptr->next;
    }
    return sum;
}

//counts the number of elements in the list
int count(List head){
    int ctr = 0;
    Nodeptr ptr=head;
    while(ptr!=NULL){
        ctr++;
        ptr = ptr->next;
    }
    return ctr;
}
```

```c
//counts the number of positive elements
int countPos(List head){
    int ctr = 0;
    Nodeptr ptr=head;
    while(ptr!=NULL){
        if(ptr->data>=0)
            ctr++;
        ptr = ptr->next;
    }
    return ctr;
}

//counts the number of negative elements
int countNeg(List head){
    int ctr = 0;
    Nodeptr ptr=head;
    while(ptr!=NULL){
        if(ptr->data<0)
            ctr++;
        ptr = ptr->next;
    }
    return ctr;
}

//counts the number of even elements
int countEven(List head){
    int ctr = 0;
    Nodeptr ptr=head;
    while(ptr!=NULL){
        if(ptr->data%2==0)
            ctr++;
        ptr = ptr->next;
    }
    return ctr;
}

//counts the number of odd elements
int countOdd(List head){
    int ctr = 0;
    Nodeptr ptr=head;
    while(ptr!=NULL){
        if(ptr->data%2!=0)
            ctr++;
        ptr = ptr->next;
    }
    return ctr;
}
```

```
printf(" = %d\n",computeSum(head));
printf(" = %d\n",count(head));
printf(" = %d\n",countPos(head));
printf(" = %d\n",countNeg(head));
printf(" = %d\n",countEven(head));
printf(" = %d\n",countOdd(head));
```

Search Function Examples

Here are search functions. Let us define them.

```
int findItem(List head, int item);
int findMin(List head);
int findMax(List head);
```

```
//returns 1 if item is in the list; 0 if otherwise
int findItem(List head, int item){
    int found=0;
    Nodeptr ptr=head;
    while(ptr!=NULL){
        if(ptr->data==item){
            found=1;
            break;
        }
        ptr = ptr->next;
    }
    return found;
}
```

Practice Exercise (Ungraded)

Define the following functions:

```
int findMin(List head);//returns the smallest element
int findMax(List head);//returns the largest element
```

Inserting a Node

Here is the algorithm on how to insert a node in a list:

1. Allocate memory for the new node.
2. Store the data value in the newly created node.
3. Determine the insertion point – that is, the position within the list where the new data is to be placed. To identify the insertion point, we need to know only the new node's predecessor.
4. Point the new node to its successor.
5. Point the predecessor to the new node.

We always start with an empty list. Inserting a node depends on where to insert.

1. ***Insert at the front***

```
//Steps 1 and 2 of the algorithm
Nodeptr createNode(int item){
    Nodeptr ptr;
    ptr=(Nodeptr)malloc(sizeof(Node));
    ptr->data=item;
    ptr->next=NULL;
    return ptr;
}

//Adds item at the front of the list
List addFront(List head, int item){
    Nodeptr temp = createNode(item);//Steps 1 and 2
    //Skip step 3 - new node's predecessor is head
    temp->next = head;//Step 4
    head = temp;//Step 5
    return head;
}
```

2. ***Insert at the end***

```
//Adds item at the end of the list
List add(List head, int item){
    Nodeptr ptr;
    Nodeptr temp = createNode(item);//Steps 1 and 2
    //Step 3
    ptr=head;
    while(ptr->next!=NULL){//Just traverse the list until it stops at the last node
        ptr=ptr->next;
    }
    //Skip step 4 since the successor is NULL
    // temp->next already points to NULL as done already in createNode()
    //temp->next = ptr->next;//or do this
    ptr->next = temp;//Step 5
    return head;
}
```

3. ***Insert in the given position***

```
//Adds item at the given position
//generic insert algorithm
//Assuming pos is within the bounds of the list
List addAt(List head, int item, int pos){
    int i;
    Nodeptr ptr;
    Nodeptr temp = createNode(item);//Steps 1 and 2
    if(pos==1){
        temp->next=head;//Step 4
        head = temp;//Step 5
    }
    else{//Step 3
        ptr=head; i=1;
        while(i<pos-1){
            ptr = ptr->next;
            i++;
        }
        temp->next = ptr->next;//Step 4
        ptr->next=temp;//Step 5
    }
    return head;
}
```

Deleting a Node

Here is the algorithm on how to delete a node:

1. Determine the position of the node that you want to delete. Know only its predecessor.
2. Have a temporary pointer and let it point to the node that you want to delete.
3. Point its predecessor to its successor.
4. Point the node to be deleted to NULL.
5. Free the deleted node.

Similar to the insert algorithm, how to delete depends on the position of the node to be deleted.

1. ***Delete from the front***

```
//deletes the first element
List deleteFront(List head){
    //Skip step 1 since the predecessor is head
    Nodeptr temp = head;//Step 2
    head = head->next; // or head = temp->next; //Step 3
    temp->next = NULL;//Step 4
    free(temp);//Step 5
    return head;
}
```

2. ***Delete the end node***

```
List deleteEnd(List head){
    Nodeptr temp, ptr;
    //Step 1, loops until ptr is at second to the last node
    ptr = head;
    while(ptr->next->next!=NULL)
        ptr = ptr->next;
    temp = ptr->next;//Step 2
    ptr->next = NULL; //or ptr->next=temp->next; //Step3
    temp->next=NULL;//Step 4
    free(temp);//Step 5
    return head;
}
```

3. ***Delete node at pos***

Practice Exercise (Ungraded)

Define the function that would delete the node in a specified position. Note that the first node is marked as node 1. deleteAt() is the generic algorithm for delete. It can delete any node as long as it the value of pos is within the bounds of the list.

```
List deleteAt(List head, int pos);
```