Dynamic Data Structure : linked list

We have discussed dynamic memory management using malloc, free and other functions in the last notes. We will continue from where we left and we will discuss some examples of use of dynamic memory management.
In these examples, models or diagrams are very much necessary to assist us in understanding the concepts. Please draw the diagrams for yourself while reading these notes and understand how these programs work. It is extremely important to understand where the pointers point to and how they get changed.
Let us examine the code in 1_ex.c.
struct X is a type. struct Y has a variable of struct X within it. It is a classical example of structure within a structure.
There is an issue with struct Z.
When the compiler compiles the struct definition, it decides the binary layout of the structure. It decides the relative position or offset of each field and also decides the total size of the structure – with or without padding based on the alignment criteria.

```
struct Z
{
        int c;
        struct Z z; // infinitely recursively included

};
```

We cannot find the size of struct Z as we cannot find the size of its field z which itself is a structure of the same kind. This results in a compile time error.

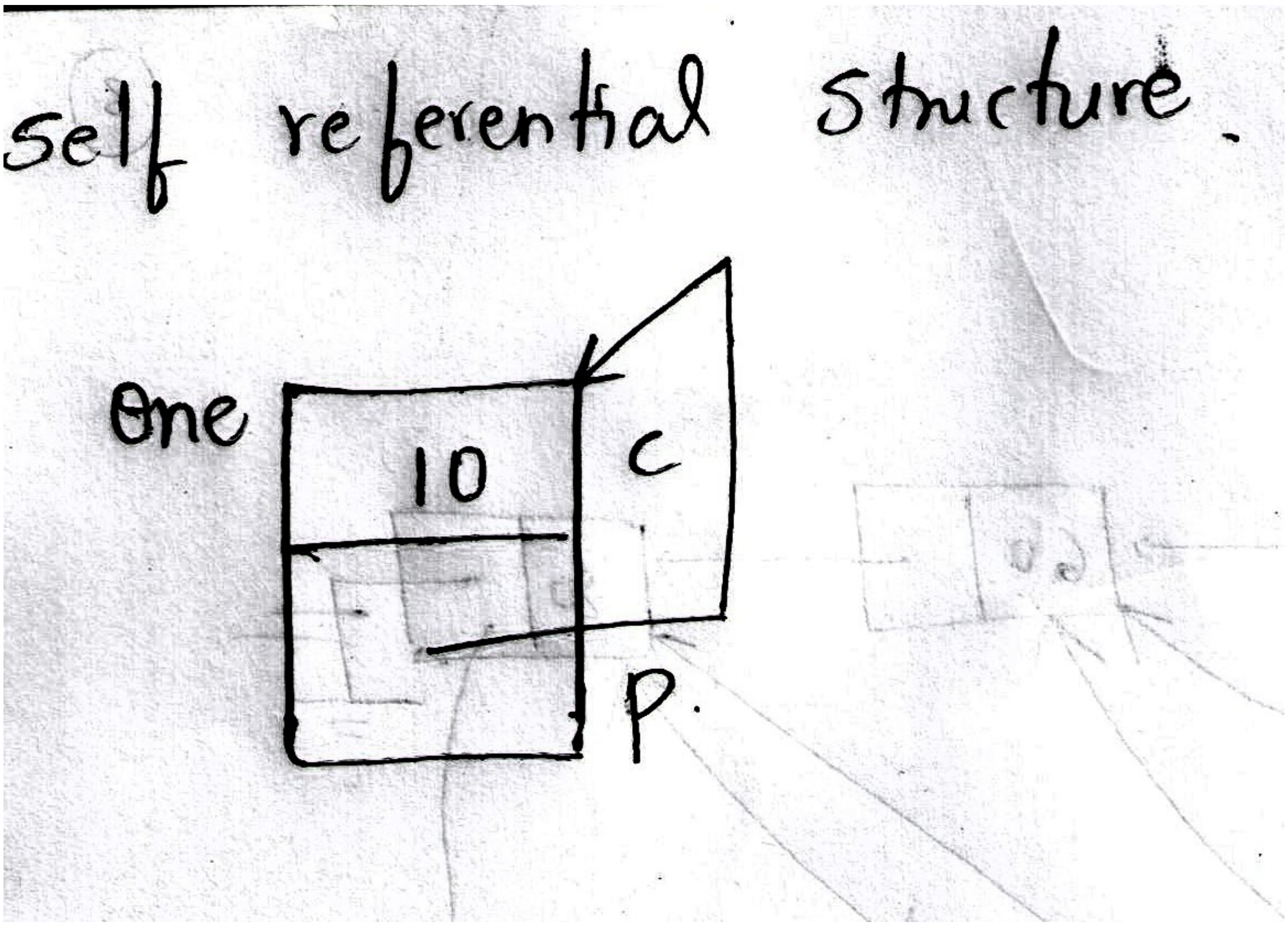
Struct Z is perfectly alright. The size of pointer to any type on a given implementation is fixed. So size of the structure can be computed by the compiler at compile time.
A structure which has a pointer to itself as a field is called a self referential structure.

```
struct A
{
	int c;
	struct A *p;
};

// 1_ex.c
#include <stdio.h>
#include <stdlib.h>
struct X
{
	int a;
};
struct Y
{
	int b;
	struct X x;
};
#if 0
struct Z
{
	int c;
	struct Z z; // infinitely recursively included

};
#endif

// self referential structure
struct A
{
	int c;
	struct A *p;
};

int main()
{
	printf("size : %lu\n", sizeof(struct X));
```

```
        printf("size : %lu\n", sizeof(struct Y));
        //printf("size : %lu\n", sizeof(struct Z));
        printf("size : %lu\n", sizeof(struct A));

        struct A one;
        one.c = 10;
        one.p = &one;
        printf("%d %d %d\n", one.c, one.p->c, one.p->p->c);
}
```

In this example, the pointer p of the variable one points to variable one.

So, one→p is same as one→p→p !!.



Linked List: a simple example:

A List is a data structure like an array. It has the following characteristics.

- Has zero or more components

- Each component has an ordinal position. We can refer to an element by its position. You may recall that set of Python does not support this concept.
- We can access element one after another. There is no way to access the element directly as in the case of arrays. Arrays have random or direct access whereas lists have sequential access.
- Insertion and deletion in a list at a given position requires a very few operations and requires no shifting.
- A Linked list is an implementation of list using self referential structures,

Let us discuss the following piece of code.

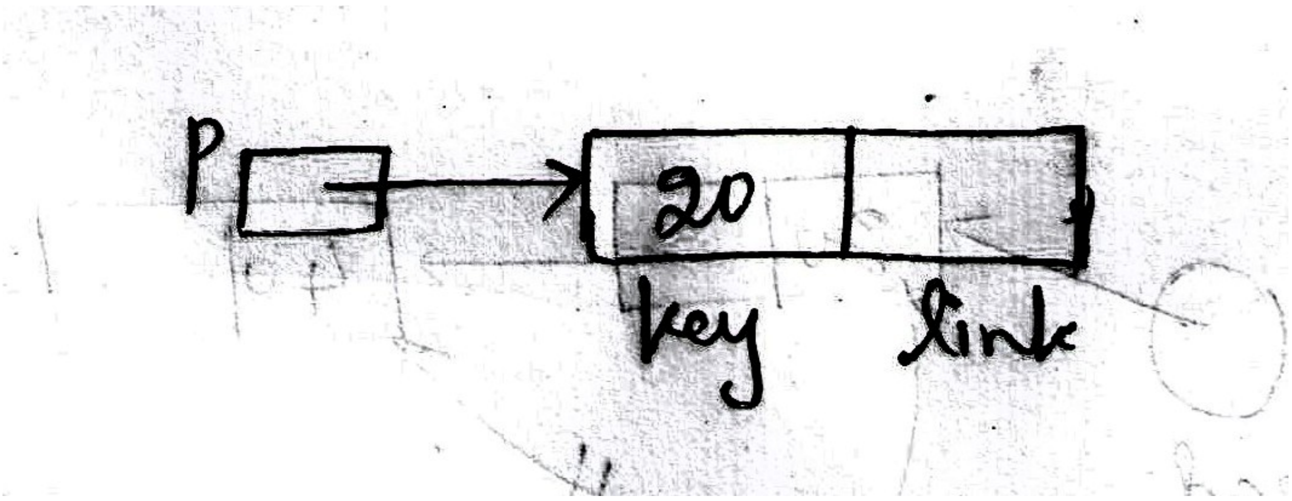
```
struct node
{
        int key;
        struct node *link;
};
typedef struct node node_t;
```

So, struct node has a field called link whose type is a pointer to the same structure. struct node is a classical example of self referential structure.

p is a pointer to structure.

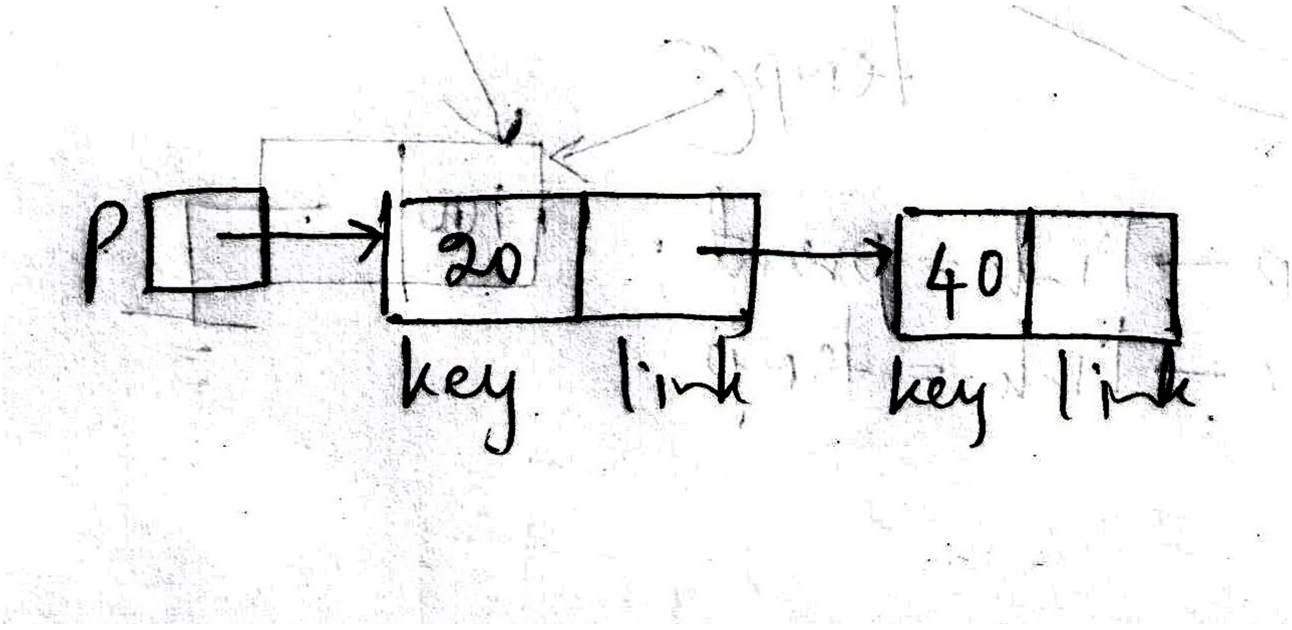
```
        node_t *p;
        p = (node_t*)malloc(sizeof(node_t));
        p->key = 20;
```

p points to a structure dynamically created. p→key has been assigned a value.



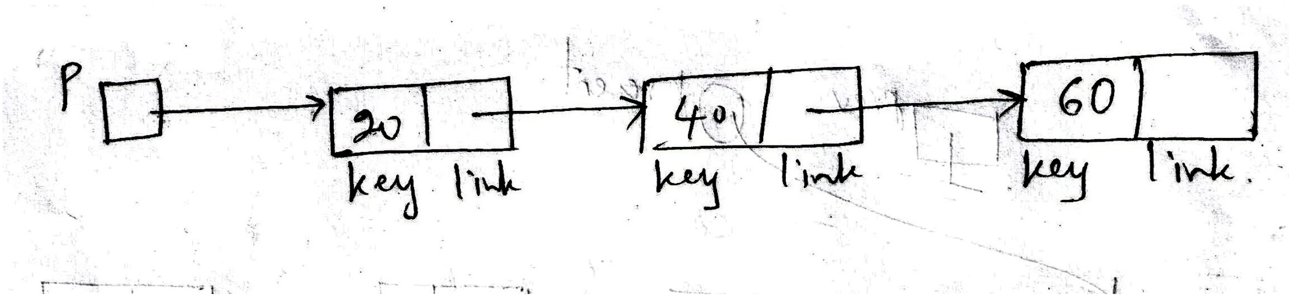
One more structure is created and p→link points to it. Key field in this new structure is assigned 40. Check the diagram.

```
p->link = (node_t*)malloc(sizeof(node_t));
p->link->key = 40;
```



```
p->link->link = (node_t*)malloc(sizeof(node_t));
p->link->link->key = 60;
p->link->link->link = NULL;
```

One more structure is created. Link field of the earlier structure points to this. Link field of the new structure is grounded (assigned NULL). So, the list ends here.

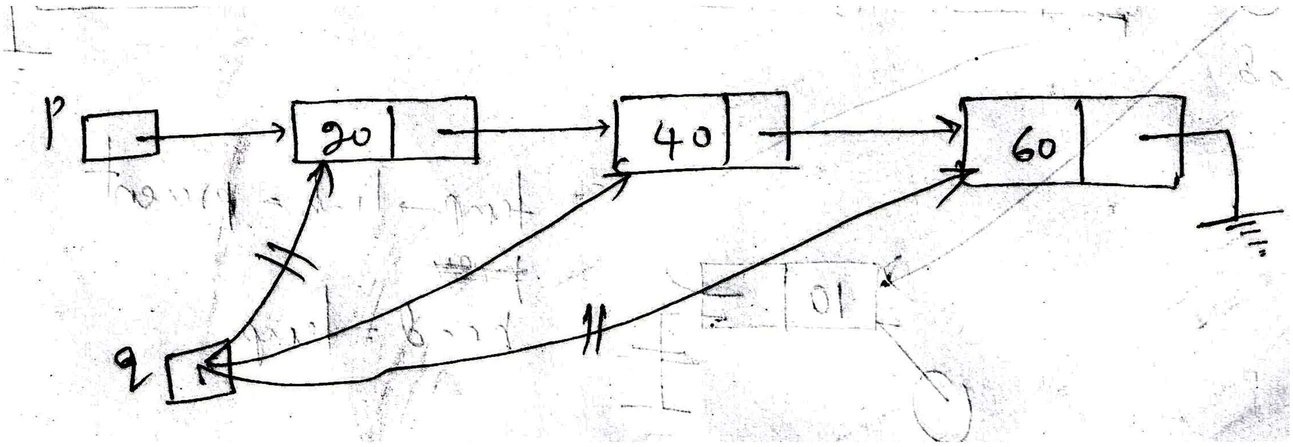


```
// 2_ex.c
#include <stdio.h>
```

```
#include <stdlib.h>

struct node
{
	int key;
	struct node *link;
};
typedef struct node node_t;

void disp(node_t* q)
{
	while(q != NULL)
	{
		printf("%d ", q->key);
		q = q->link;
	}
}
void freelist(node_t* q)
{
	node_t* r;
	while(q != NULL)
	{
		r = q->link;
		free(q);
		q = r;
	}

}
int main()
{
	node_t *p;
	p = (node_t*)malloc(sizeof(node_t));
	p->key = 20;
	p->link = (node_t*)malloc(sizeof(node_t));
	p->link->key = 40;
```

```
        p->link->link = (node_t*)malloc(sizeof(node_t));
        p->link->link->key = 60;
        p->link->link->link = NULL;
        disp(p);
        freelist(p);
}
```

p points to the beginning of the list - points to the first node of the list. p is referred as the head of the list. Given the head of the list, we can traverse or walk through the list. Let us examine the disp function. The parameter q gets a copy of p. q points to the first node in the list.
printf("%d ", q→key);
q→key displays the key field in the first node.

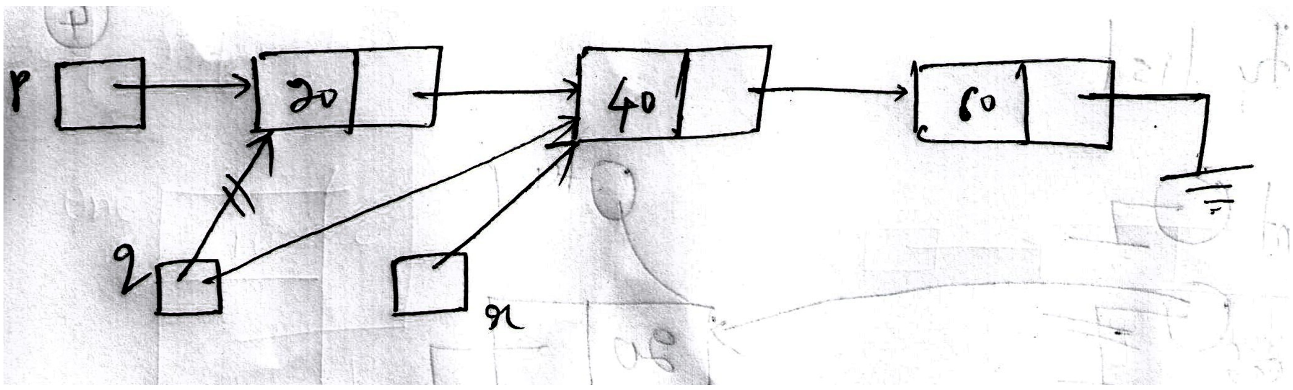


We should move to the next node. We cannot do q++ as the nodes are not necessarily allocated contiguous memory unlike arrays. The link field in the node to which q points to, points to the next node. So, we copy the link field of q to q. The statement below is the most important statement in data structures.
q = q→link;
we repeat this until q becomes NULL signifying that we have come to the end of the list.
The function freelist releases the memory occupied by the linked list.
Observe that we should not say

free(p);

This would release the first node and rest of the list becomes garbage.

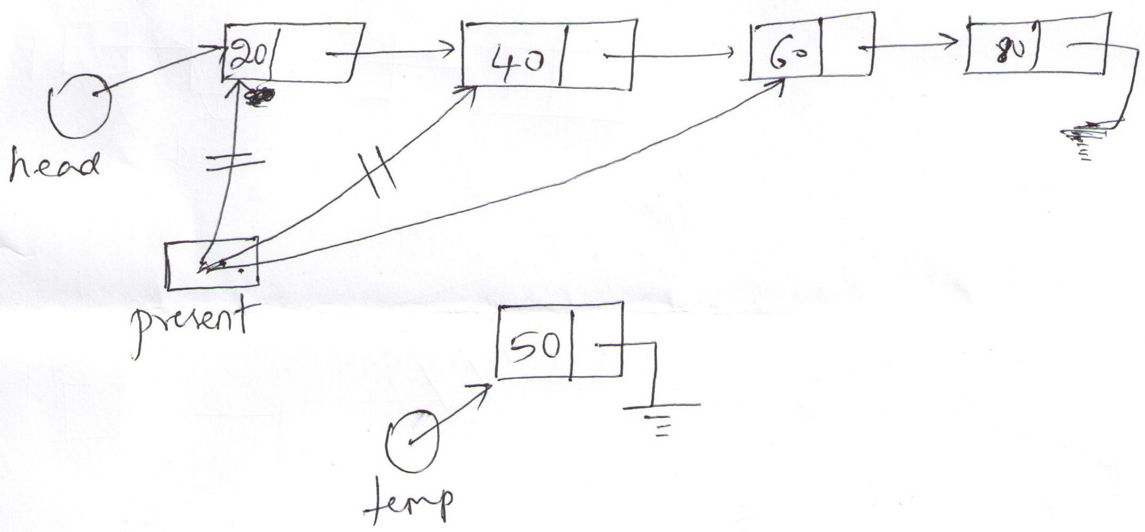
```
r = q->link;
free(q);
q = r;
```



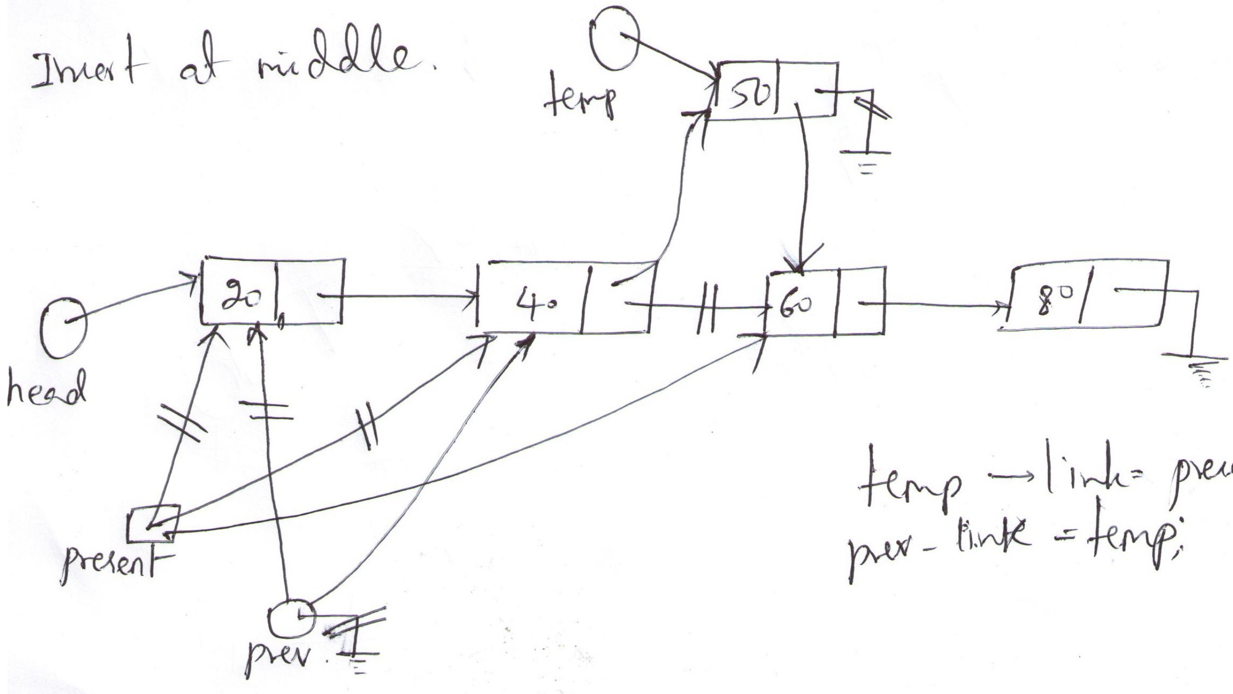
Observe that before removing what q points to, we should copy the link field so that we can change q to point to the next node after deleting what q pointed to before. The order of statements is very important. This technique of storing a pointer value temporarily before restoring it into some variable is very commonly used in data structures.

Ordered list:

Ordered List

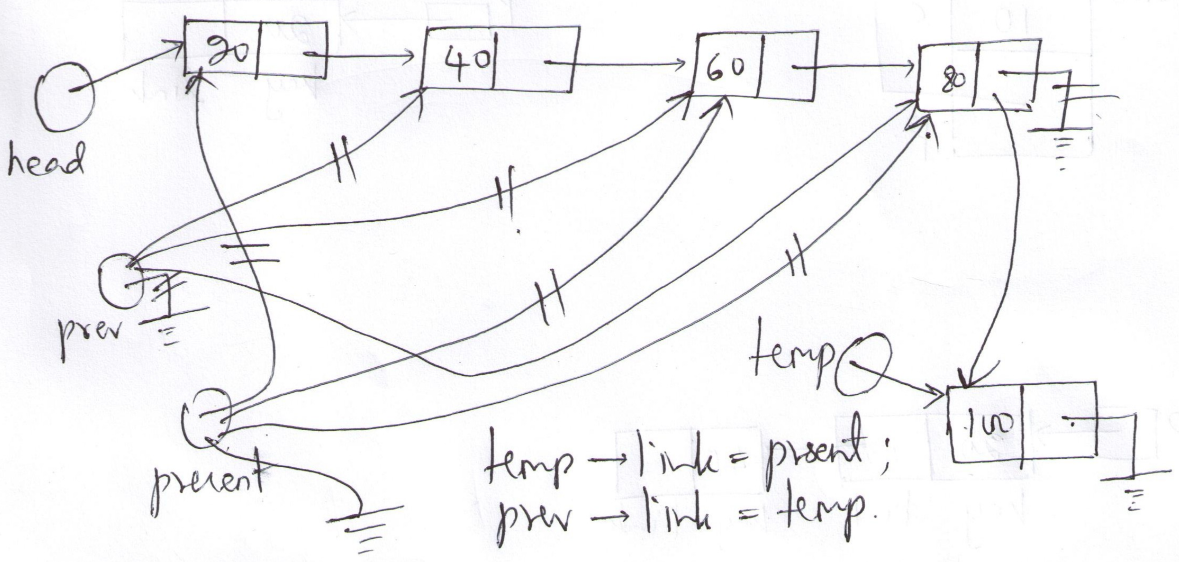


Insert at middle.



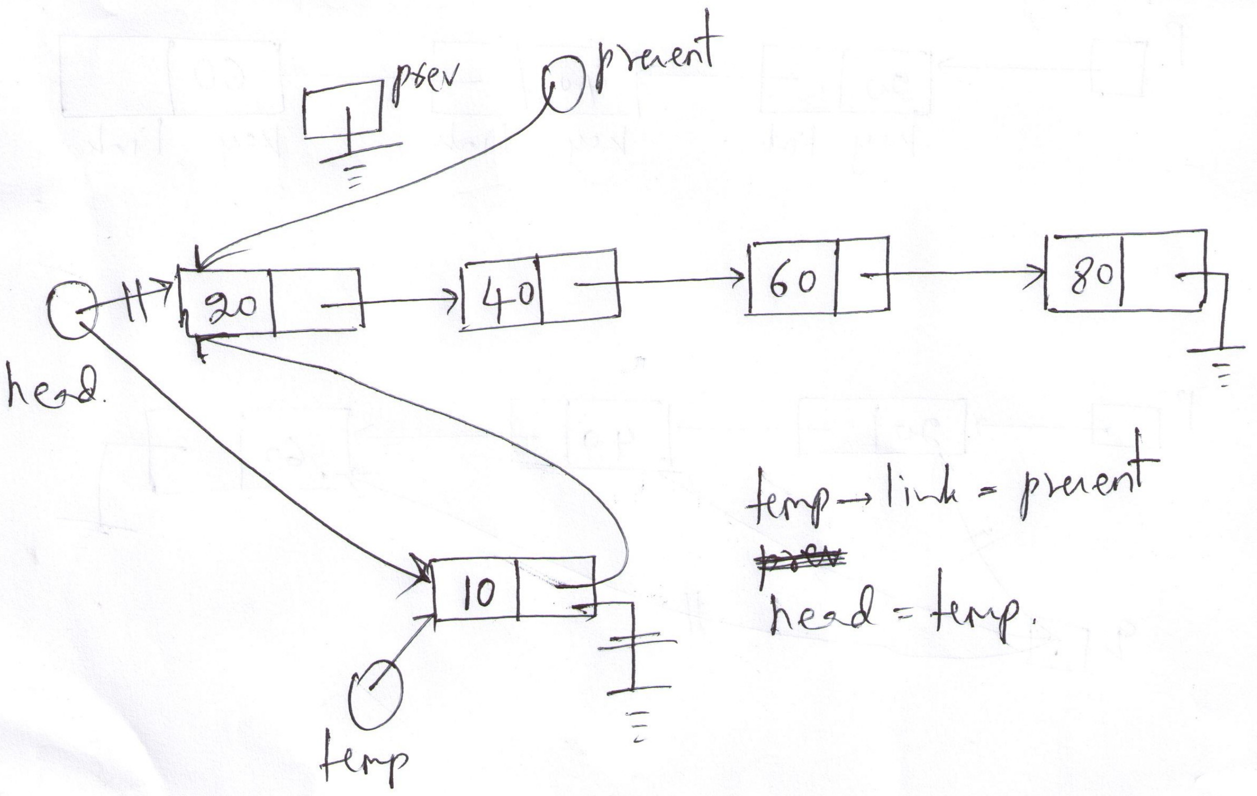
temp → link = present;
prev - link = temp;

insert at the end.

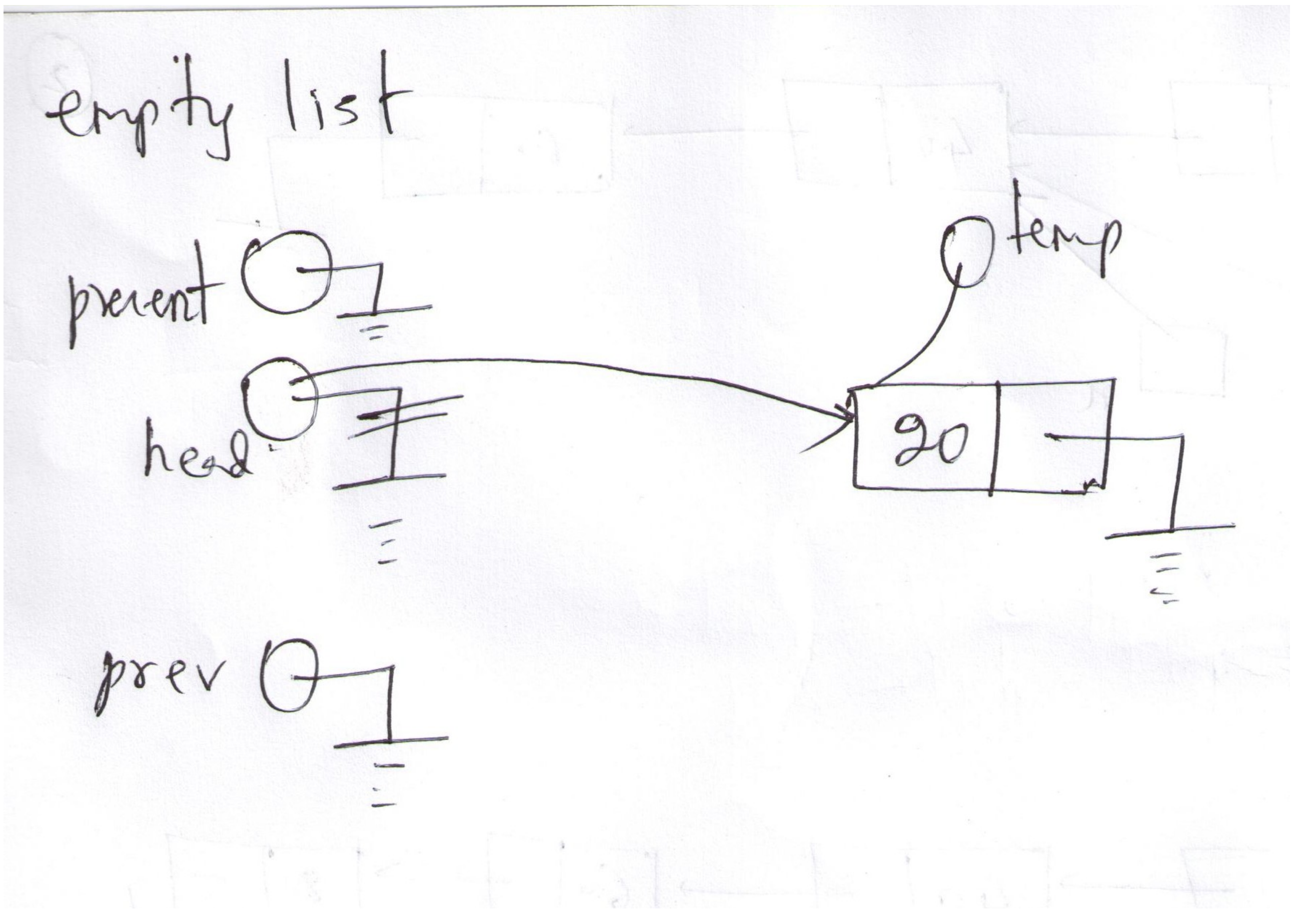


temp → link = present;
prev → link = temp.

insert at the begging.



temp → link = present
~~prev~~
head = temp.



// client.c

An ordered list has nodes arranged in the order of key based on some predicate. Let us try to create a list of nodes in the non-decreasing order of keys.

The list structure is opaque for the user. The user should not directly access any thing within this structure as this could change.

Our implementation of list structure has the head of the list within it. Our implementation uses self referential structure node_t.

The client code initializes the list. The empty list normally grounds the head – makes it NULL.

The client code inserts elements into the list by calling the function insert_list.

The insert_list function creates a node, fills the key field, grounds the link field and then inserts the node in the right position in the list.

There are four cases to consider.

a) empty list

b) insert in the middle

c) insert at the end

d) insert in the beginning

```
if(ptr_list->head == NULL)
        {
                ptr_list->head = temp;
                temp->link = NULL;
        }
```

The code above takes care of the empty list case. The head being NULL indicates that the list is empty. Change the head to point to the new node created and ground the pointer of the new node.

To insert in a non-empty list, we have to find the position to insert. We make a pointer pres point to the first node of the list. Compare the key in node pointed to by pres to the key in new node pointed to by temp. if the former is less than the latter, we move to the next node by changing pres. pres = pres→link. We realize that we should have inserted only after we cross the point. It is like traveling in one way traffic. We cannot move back.

We can overcome this problem by having a prev pointer tail before the pres as shown in the code. The initial value of prev is NULL.

```
node_t* prev = NULL;
node_t* pres = ptr_list->head;
while(pres != NULL && pres->key < temp->key )
{
        prev = pres;
        pres = pres->link;
}
```

We exit the loop in two ways. Either we have found a position or the we have reached the end of the list.

In case of insertion in the beginning of the list, head has to be changed. We make out that the insertion has to be in the beginning, if the prev is still NULL after the loop is exited.

```
if(prev == NULL) // insert in the beginning
                {
                        temp->link = pres;
                        ptr_list->head = temp;
```

```
		}
		else // middle or end
		{
			temp->link = pres;
			prev->link = temp;
		}
```

We also note that the code for insertion at the end or in the middle is same. We insert between the nodes pointed to by prev and pres. Link of prev points to what temp points to and link of temp points to what pres points to.

The rest of the functions are self explanatory.

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

int main()
{
	mylist_t l;
	init_list(&l);
	int a[] = { 20, 10, 50, 30, 40};
	int n = 5;
	for(int i = 0; i < n; ++i)
	{
		insert_list(&l, a[i]);
	}
	disp_list(&l);
	free_list(&l);
}

//  mylist.h
#ifndef MYLIST_H
#define MYLIST_H
struct node
{
	int key;
	struct node *link;
};
typedef struct node node_t;



struct mylist
{
	node_t *head;
};
typedef struct mylist mylist_t;

void init_list(mylist_t*);
void insert_list(mylist_t*, int key);
void disp_list(mylist_t*);
void free_list(mylist_t*);

#endif

// mylist.c
#include <stdio.h>
#include "mylist.h"
#include <stdlib.h>

void init_list(mylist_t* ptr_list)
{
	ptr_list->head = NULL;
}
// 1. empty list
// 2. insert in the middle
// 3. insert in the beginning
// 4. insert at the end
void insert_list(mylist_t* ptr_list, int key)
{
	// create the node
	node_t* temp = (node_t*)malloc(sizeof(node_t));
```

```
        temp->key = key; temp->link = NULL;

        // 1.empty list
        if(ptr_list->head == NULL)
        {
                ptr_list->head = temp;
                temp->link = NULL;
        }
        else
        {
                // traverse
                node_t* prev = NULL;
                node_t* pres = ptr_list->head;
                while(pres != NULL && pres->key < temp->key )
                {
                        prev = pres;
                        pres = pres->link;
                }
                if(prev == NULL) // insert in the beginning
                {
                        temp->link = pres;
                        ptr_list->head = temp;
                }
                else // middle or end
                {
                        temp->link = pres;
                        prev->link = temp;
                }

        }


}

void disp_(node_t* q)
```

```
{
	while(q != NULL)
	{
		printf("%d ", q->key);
		q = q->link;
	}
}

void disp_list(mylist_t* ptr_list)
{
	disp_(ptr_list->head);
}
void free_list_(node_t* q)
{
	node_t* r;
	while(q != NULL)
	{
		r = q->link;
		free(q);
		q = r;
	}
}
void free_list(mylist_t* ptr_list)
{
	free_list_(ptr_list->head);
}
```