5.3.2 Representing a Stack using a Linked List

A stack represented using a linked list is also known as *linked stack*. The array based representation of stacks suffers from following limitations

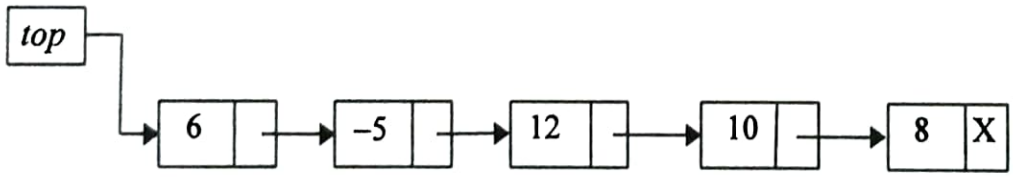
- Size of the stack must be known in advance.
- We may come across situations when an attempt to push an element causes overflow. However, stack as an abstract data structure can not be full. Hence, abstractly, it is always possible to push an element onto stack. Therefore, representing stack as an array prohibits the growth of stack beyond the finite number of elements.

The linked list representation allow a stack to grow to a limit of the computer's memory. The following are the necessary declarations

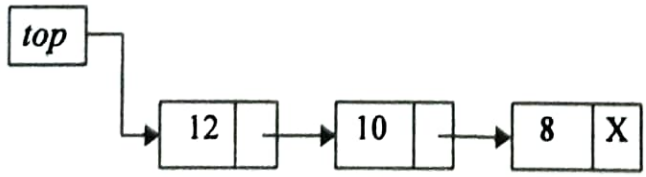
```
typedef struct node
{
    int  info;
    struct node *next;
 } stack;

stack *top;
```

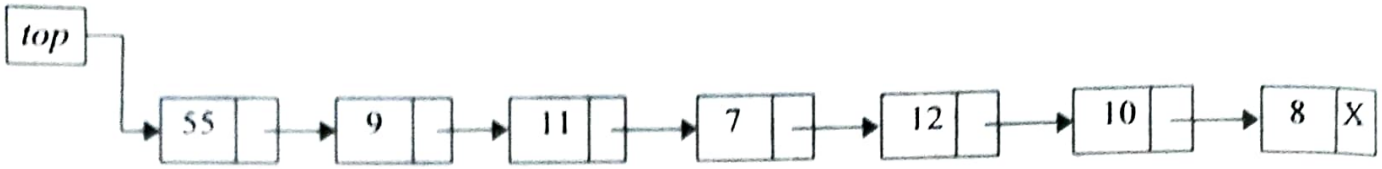
We have defined our own data type named *stack*, which is a self-referential structure and whose first element *info* hold the element of the stack and the second element *next* holds the address of the element under it in the stack. The last line declares a pointer variable *top* of type *stack*.



(*a*) Representation of stack of Figure 5.1(*a*) in memory



(*b*) Representation of stack of Figure 5.1(*b*) in memory



(*c*) Representation of stack of Figure 5.1(*c*) in memory

Figure 5.3 Linked representation of stack of Figure 5.1

With these declarations, we will write functions for various operation to be performed on a stack represented using linked list.

5.3.2.1 Creating an Empty Stack

Before we can use a stack, it is to be initialized. To initialize a stack, we will create an empty linked list. The empty linked list is created by setting pointer variable *top* to value NULL.

This simple task can be accomplished by the following function.

Listing 5.12

```
void CreateStack( stack **top )
{
    *top = ( stack * ) NULL;
}
```

5.3.2.2 Testing Stack for Underflow

The stack is tested for underflow condition by checking whether the linked list is empty. The empty status of the linked lists will be indicated by the NULL value of pointer variable *top*. This is shown in the following function.

Listing 5.13

```
boolean IsEmpty( stack *top )
{
   if ( top == ( stack * ) NULL )
     return true;
   else
     return false;
}
```

The above function returns *boolean* value *true* if the test condition is satisfied; otherwise it returns value false.

The *IsEmpty()* function can also be written using conditional operator (?:) as

Listing 5.14

```
boolean IsEmpty( stack *top )
{
   return ( top == ( stack * ) NULL ? true : false );
}
```

5.3.2.3 Testing Stack for Overflow

Since a stack represented using a linked list can grow to a limit of a computer's memory, there overflow condition never occurs. Hence, this operations is not implemented for linked stacks.

5.3.2.4 Push Operation

To push a new element onto the stack, the element is inserted in the beginning of the linked list. This task is accomplished as shown in the following function.

Listing 5.15

```
void Push( stack **top, int value )
{
   stack *ptr;
   ptr = ( stack * ) malloc ( sizeof ( stack ) );
   ptr->info = value;
   ptr->next = *top;
   *top = ptr;
}
```

5.3.2.5 Pop Operation

To pop an element from the stack, the element is removed from the beginning of the linked list. This task is accomplished as shown in the following function.

Listing 5.16

```
int Pop( stack **top )
{
   int temp;
   stack *ptr;
```

```
temp = *top->data;
ptr = *top
*top = *top->next;
free ptr;
return temp;
```

The element in the beginning of the linked list is assigned to a local variable temp, which later on will be returned via the return statement. And the first node from the linked list is removed.

5.3.2.6 Accessing Top Element

The top element is accessed as shown in the following function.

Listing 5.17

```
int Peek( stack *top )
{
   return ( top->data );
}
```

5.3.2.7 Dispose a Stack

Because stack is implemented using linked lists, therefore it is programmers job to write the code to release the memory occupied by the stack. The following listing shows the different steps to be performed.

Listing 5.18

```
void DisposeStack( stack **top )
{
   stack *ptr;
   while ( *top != stack * NULL )
   {
      ptr = *top;
      *top = (*top)->next;
      free ( ptr );
   }
}
```

This dispose operations is O(n) time.