Fundamentals of C required to study Data Structure

Dr. Rahul Das Gupta

1. Basics of Pointer :

(i) Two different meaning of *.

Case-1: * is associated to *data type* in case of any *declaration statement*.

```
int *ip;
```

/*ip is a pointer to an integer or address of an integer.*/

```
float *fp;
```

/*fp is a pointer to a float or address of a float.*/

```
char *cp;
```

/*cp is a pointer to a character or address of a character.*/

Variable	Data Type
ip	int *(pointer to an integer or address of an integer)
fp	float *(pointer to a float or address of a float)
cp	char * (pointer to a character or address of a character)

Case-2: * is a *'content of'* operator in case of any *non-declaration statement*. Here * is associated with variable.

```
int *ip;
int n=2;
ip=&n; /* Here, & is a 'address of operator'.*/
printf("\n Content at ip =%d", *ip);
*ip=4;
/* Meaning of *ip : Content at the address ip
Where ip is a pointer to an integer or address of an integer. */
printf("\n Value of  n =%d", n);
```

```
int *ip;
int n=2;
ip=&n
printf ("\n Content at ip =%d", *ip);
*ip=4;
printf("\n Value of  n =%d", n);
```

n | *ip = ~~2~~ 4

ip= &n = 1677

Content at ip = 2

Value of n = 4

Output:

Content at ip = 2
Value of n = 4

(ii) **Actual value of (p+i) = p + i*sizeof(*data_type*)**
Actual value of (p-i) = p - i*sizeof(*data_type*)

```
int A[10];
int *p;
p=A;
```

$\boxed{p+7=??}$

p = A=1074	A[0]
A+1=1076	A[1]
A+2=1078	A[2]
A+3=1080	A[3]
A+4=1082	A[4]
A+5=1084	A[5]
A+6=1086	A[6]
A+7=1088	A[7]
A+8=1090	A[8]
A+9=1092	A[9]

Wrong Answer

~~p+7 = 1074 + 7 = 1081~~

Correct Answer

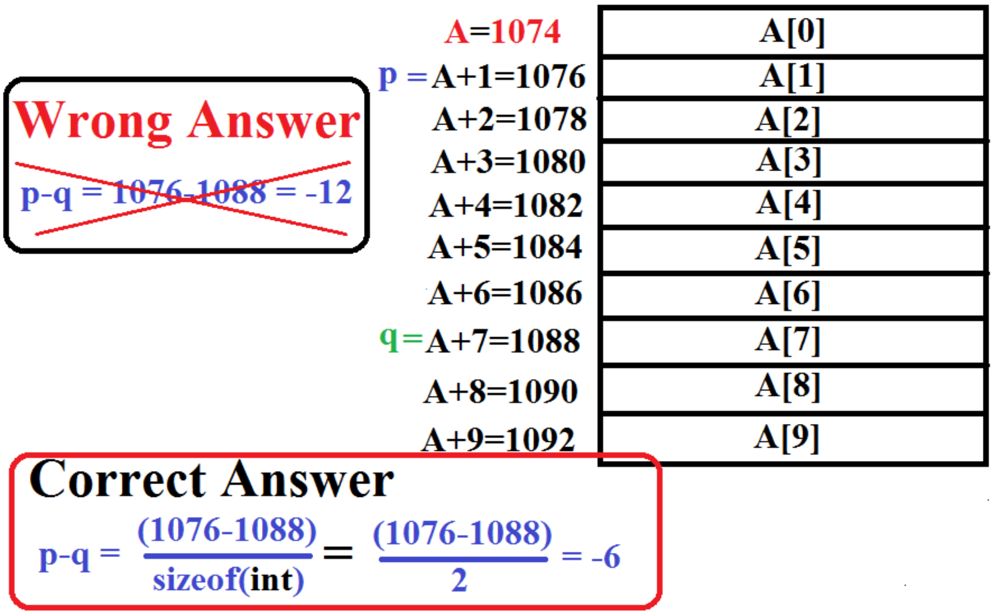
p+7 = 1074 + 7*sizeof(int) =1074 +7*2 = 1088

(iii) **Actual value of (p-q) = (p–q)/sizeof(*data_type*)**

Example:

```
int A[10];
int *p,*q;
p=&A[1];
q=&A[7];
printf("\n %d",p-q);
```

Output:



(iv) **p[i]=*(p+i)=*(i+p)=i[p]**

(v) **p+i=&(p[i])=&(i[p])=i+p**

2. To change the value of a variable through a function, the address of the variable must be passed as an input argument to the function.

Example:

```
#include<stdio.h>

void swapping1 (int, int);
void swapping2(int *,int *);

void main ( )
{
  int  m=2, n=3;

   swaping1 (m, n);

   /* Swapping  will not  be effective.*/

   printf ("\n m = %d, n = %d ",  m, n);
```

```
   swapping2 (&m, &n);

  /* Actual swapping.  */

   printf ("\n m = %d, n = %d ",  m, n);
}







void swapping1 (int x, int y)
{
   int  t=x;
   x=y;
   y=t;
}


void swapping2(int *p, int *q)
{
    int  t=*p;
    *p=*q;
    *q=t;
}
```

Output

m=2, n=3
m=3, n=2

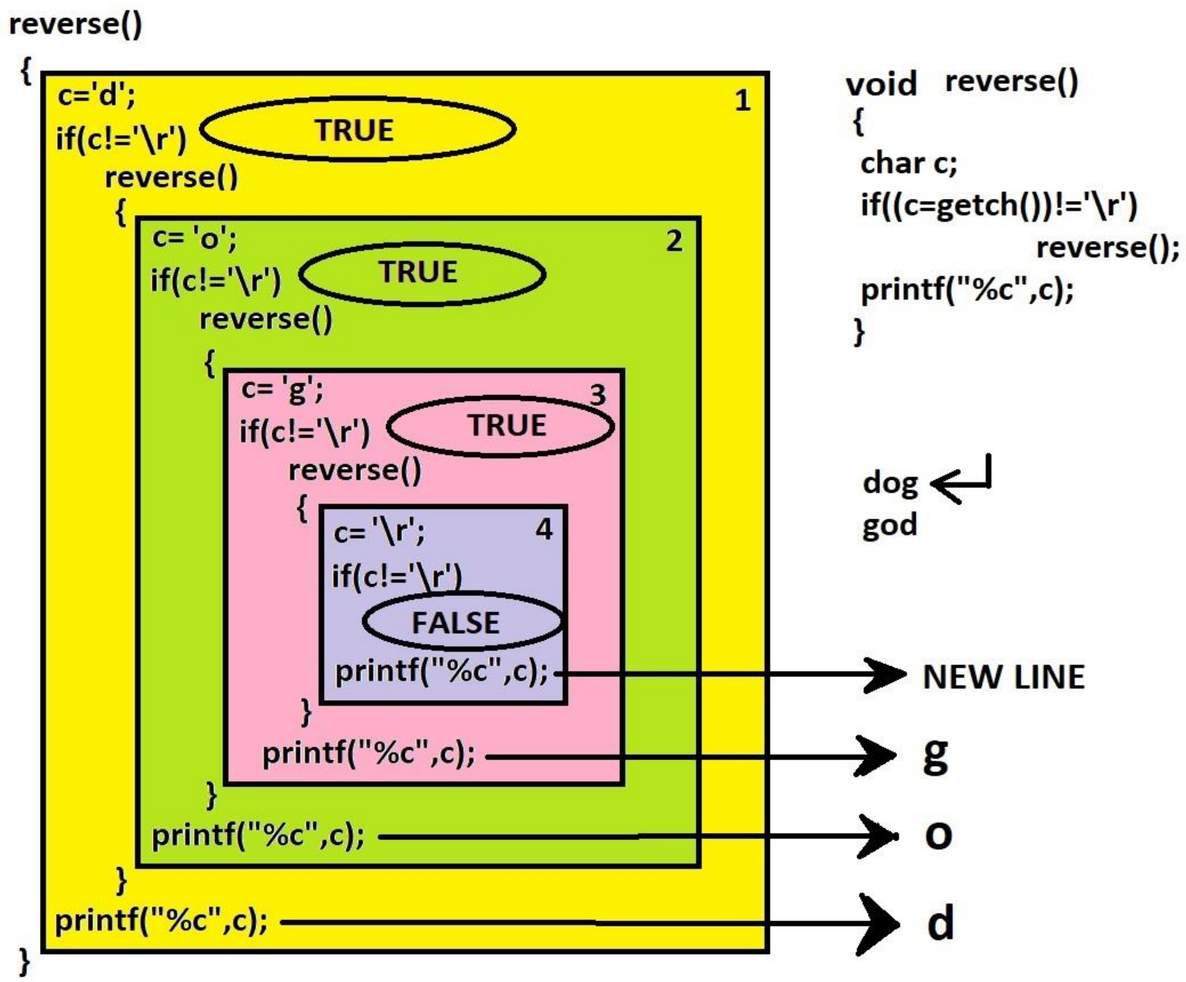
Item to be Modified	Prototype of the Function Modifying the Item
int float	void change(int *); void change(float *);
LINKED_LIST * (address of the Head Node)	void change(LINKED_LIST **); (Input Argument : Address holding the address of the Head Node)
BS_TREE * (address of the Root Node)	void change(BS_TREE **); (Input Argument : Address holding the address of the Root Node)

3. (i) Array is a contiguous location of homogeneous data.
(ii) Array name is the starting address of the contiguous allocation.
(iii) Array name is a constant pointer.

```
int x;
int a[10];
a=NULL;  /*We can not do it. */
a=&x;  /*We can not do it.*/
```

4. Recursive Function

A function that calls itself repetitively until any terminating condition is achieved.



’\r’: carriage return character

INPUT: dog’\r’

OUTPUT AT NEW LINE: god

5. Structure

```
struct employee
{
    char id[10];
    char name[40];
    char sex;
    int age;
    float sal;
};
struct employee e;
```

Type Definition of Structure

```
typedef struct
{
    char id[10];
    char name[40];
    char sex;
    int age;
    float sal;
}EMPLOYEE;

EMPLOYEE e;
```

Member Accessing Operators (. and ->)

(i) Ordinary Structure Variable (Non-pointer Variable)

EMPLOYEE e;

Members are the following:

e.id
e.name
e.sex
e.age
e.sal

Member accessing operator : . (Dot)

(ii) Pointer to a Structure (Pointer Variable)

EMPLOYEE * p;

Allocation of space at the address is a mandatory requirement

p = (EMPLOYEE *) malloc (sizeof (EMPLOYEE));

Members are the following:

p->id
p->name
p->sex
p->age
p->sal

Member accessing operator : -> (Arrow)

Important Points to be Noted

p->id = (*p).id
p->name = (*p).name
p->sex = (*p).sex
p->age = (*p).age
p->sal = (*p).sal

6. Self-Referential Structure

A structure element which contains at least one pointer pointing to a structure of similar type.

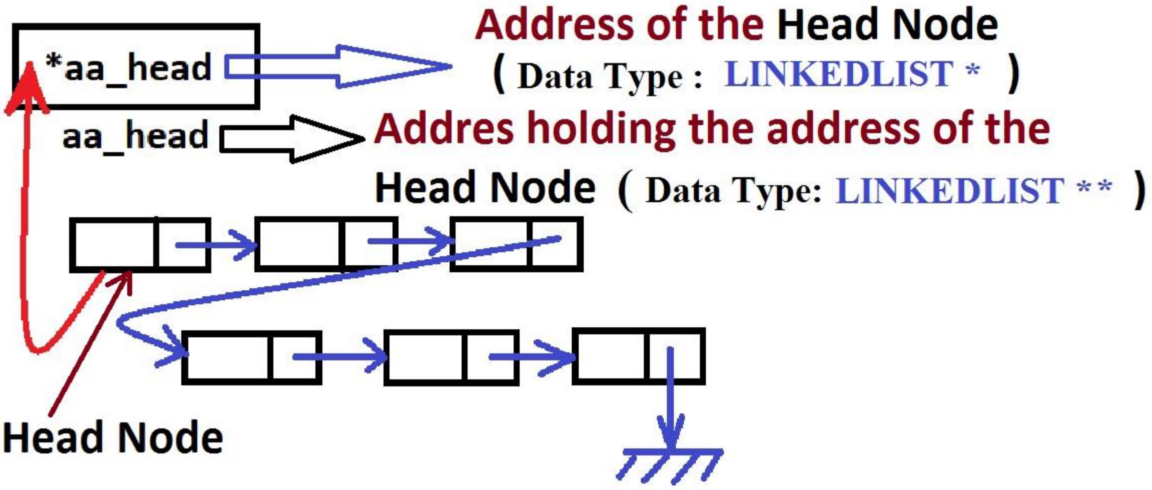
Examples:

(i) Linked List

Recursive Definition of Structure (Self Referential Structure)

```
typedef struct linked_list
{
    int data;
    struct linked_list  *nxt;
}LINKEDLIST;
```

The name LINKEDLIST is equivalent to struct linked_list



(ii) Binary Tree

```
typedef struct bs_tree
{
    int data;
    struct bs_tree  *left, *right;
}BS_TREE;
```

The name BS_TREE is equivalent to struct bs_tree

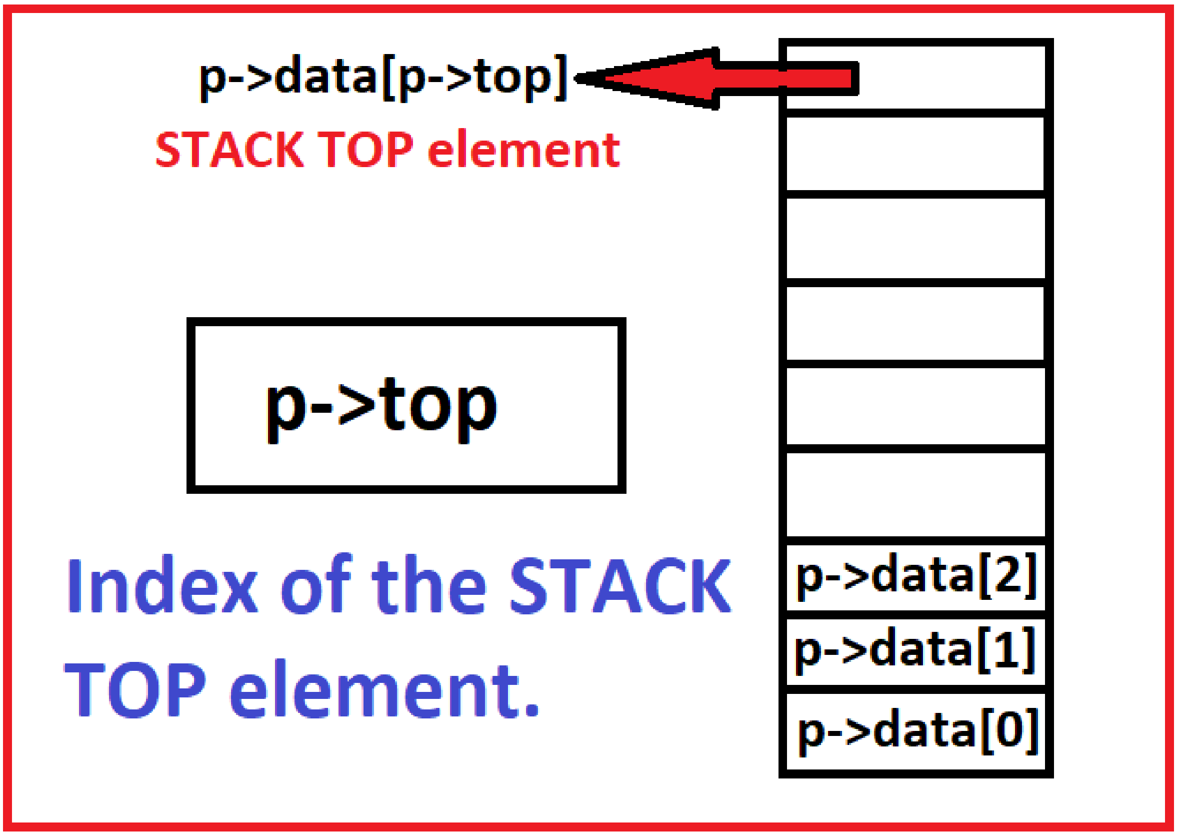
STACK

Dr. Rahul Das Gupta

Implementation of Stack Using Array

```
#include<stdio.h>
#include<stdlib.h>
#define MAX  100     /*PREPROCESSOR DIRECTIVE*/

/* ============      STRUCTURE DEFINITION    ============*/
typedef struct
{
  int data[MAX];
  int  top;
}STACK;

/* ============= DYNAMIC STACK=====================
typedef struct
{
  int *data;
  int  top;
}STACK;
=====================================================*/

/* ========    END OF STRUCTURE DEFINITION ===========*/
```

```
/* ==============   PROTOTYPE DECLERATION      =========*/
void initialisation (STACK *);
void initialisation_dynamic(STACK  *, int );
int is_empty (STACK);
int is_full (STACK);
void push (STACK *, int );
int pop (STACK * );
/* ==========   END OF PROTOTYPE DECLERATION    ========*/
```



p = Address where the STACK is allocated.
[Data Type : STACK *]

```
void initialisation (STACK  *p)
{
        p->top=-1;
}
/*========= Initialisation for Dynamic Stack =====================
```

```
void initialisation_dynamic(STACK  *p, int stack_size)
{
          p->data=(int *) malloc(sizeof(int)*stack_size);
          p->top= -1;
}
==============================================================*/

int is_empty (STACK  p)
{
         return (p.top = = -1);
}


int is_full (STACK  p)
{
        return (p.top==MAX-1);
}

void push (STACK *p,  int n)
{
  if (p->top!=MAX-1)
       p->data[++(p->top)]=n;
 else
      printf ("\n Stack Over Flow…");
}


int pop (STACK  *p)
{
  if (p->top!=-1)
     return( p->data[(p->top)--]);
  else
     {
        printf("\n Empty Stack…");
        exit(1);
     }
}

/*============================================================
```

IMPORTANT POINTS:

(i) **Three functions: initialisation, push and pop causing structural change to the STACK . Hence, in these functions one of the**

argument must be pointer to STACK to make those change effective.

(ii) Two other functions is_empty and is_full are just reading the STACK, performing no structural change to the STACK . Hence, in these functions there is no pointer to STACK as an argument.

```
=====================================================
*/

void main ( )
{
    STACK  s,*pt;
    int size;
   % Case I:  Implemented using a structure variable %

    initialisation(&s);

    push(&s,4);
    push(&s,7);
    push(&s,9);
    push(&s,14);
    push(&s,16);
    push(&s,18);
    push(&s,24);

   printf("\n %d",pop(&s));
   printf("\n %d",pop(&s));
   printf("\n %d",pop(&s));
   printf("\n %d",pop(&s));
   printf("\n %d",pop(&s));
   printf("\n %d",pop(&s));
   printf("\n %d",pop(&s));
   printf("\n %d",pop(&s));

  % Case II:    Using a pointer to STACK %

  % === Allocation of memory for the structure STACK ===%
  pt=(STACK *)malloc(sizeof(STACK));
  % *** Without this allocation it will not work  *****%

   printf("\n Enter the stack size:");

   scanf("%d", &size);

   initialisation_dynamic(pt, size);
```

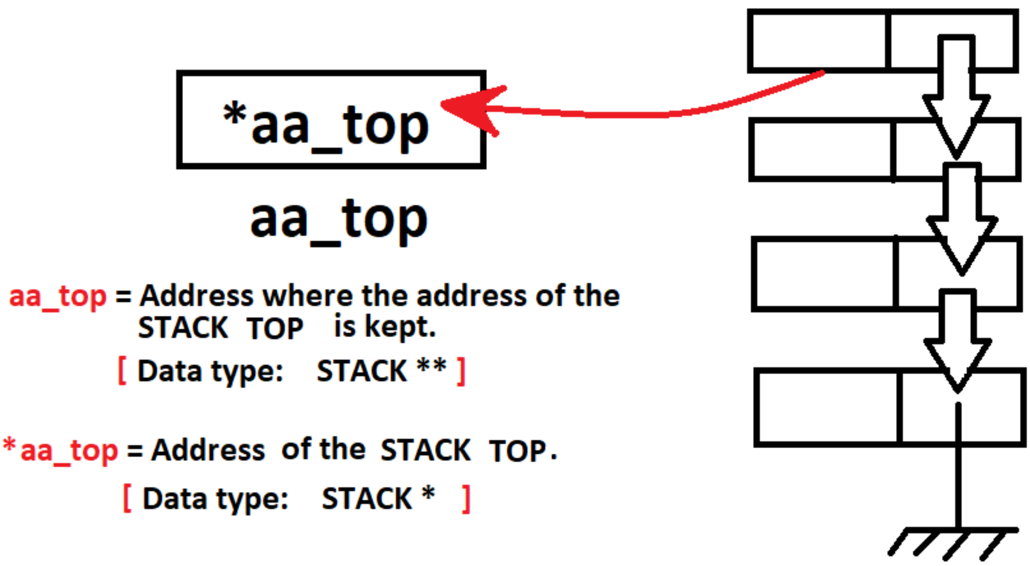
```
push(pt,4);
push(pt,7);
push(pt,9);
push(pt,14);
push(pt,16);
push(pt,18);
push(pt,24);

printf("\n %d",pop(pt));
printf("\n %d",pop(pt));
printf("\n %d",pop(pt));
printf("\n %d",pop(pt));
printf("\n %d",pop(pt));
printf("\n %d",pop(pt));
printf("\n %d",pop(pt));
printf("\n %d",pop(pt));

}
```

Implementation of Stack Using Linked List

Dr. Rahul Das Gupta

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

/* ==DEFINITION  OF SELF-REFERENTIAL  STRUCTURE ==*/
typedef struct linked_list
{
  int data;
  struct linked_list *nxt;
}STACK;
/* ========    END OF STRUCTURE DEFINITION ===========*/


/* ==============   PROTOTYPE DECLERATION      =========*/
void initialisation (STACK **);
```

```
void push (STACK **, int);
int  pop (STACK **);
int  is_empty (STACK *);
/* ==========   END OF PROTOTYPE DECLERATION    ========*/

void initialisation (STACK **aa_top)
{
        *aa_top=NULL; /* Data Type of *aa_top : STACK*    */
}

/*=====================================================
IMPORTANT POINTS:
    Whenever an item is pushed or popped the address of the STACK TOP
is changed.  Hence a function that attempts to change the address of the
STACK TOP must contains address of the address of the STACK TOP as
an argument.
     Functions of this kind are initialisation, push and pop. All these
functions contain address of the address of the STACK TOP as an
argument.
=====================================================
*/
```



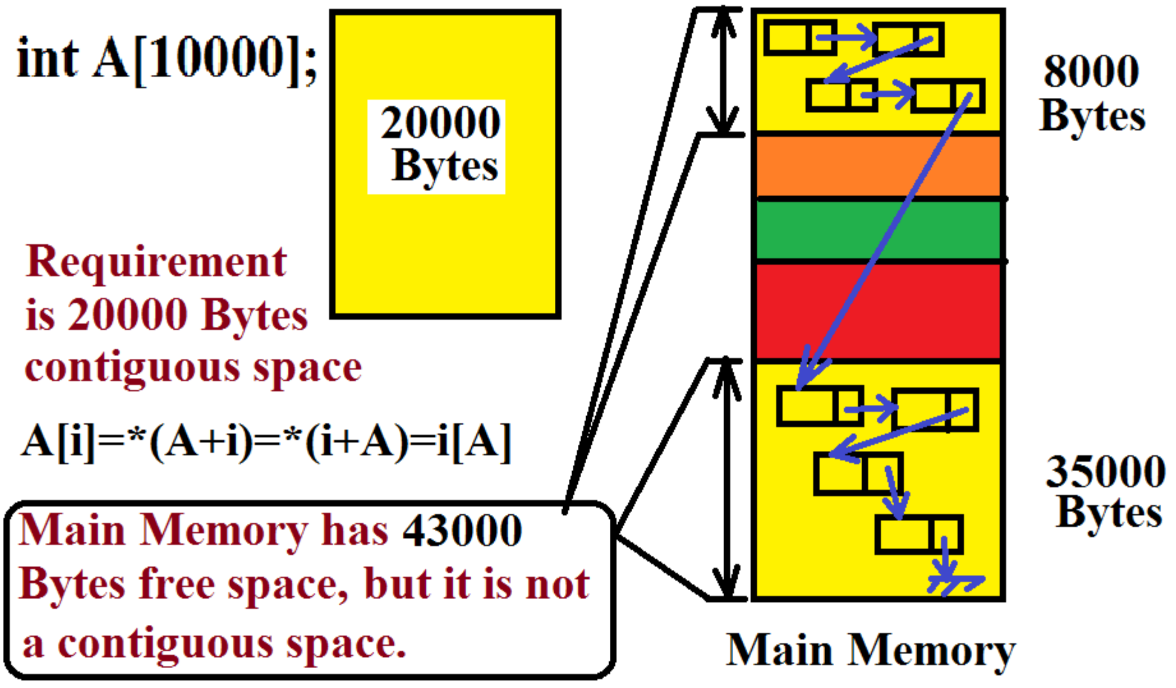

```
void push (STACK **aa_top, int n)
{
  STACK *t;
  t=(STACK *)malloc(sizeof(STACK));
  t->data=n;
```

```
  t->nxt=*aa_top;
 /* *aa_top= Address of the Stack Top (Data Type: STACK* )  */
  *aa_top=t;
}

int pop (STACK **aa_top)
{
     int n;
     STACK * t;
     if (*aa_top!=NULL)
     {
           t=*aa_top;
           n=t->data;
           *aa_top=t->nxt;
           free(t);
           return(n);
     }
    else
     {
        printf("\n Empty Stack...");
        exit(1);
     }
}


int  is_empty(STACK *aa_top)
{
      return (aa_top==NULL);
}

void  main( )
{
      STACK *s=NULL;

       initialisation (&s);

       push(&s,2);
       push(&s,3);
       push(&s,5);
       push(&s,6);
       push(&s,7);
```

```
    push(&s,8);
    push(&s,9);
    push(&s,10);

    printf("\n %d",pop(&s));
    printf("\n %d",pop(&s));
    printf("\n %d",pop(&s));
    printf("\n %d",pop(&s));
    printf("\n %d",pop(&s));
    printf("\n %d",pop(&s));
    printf("\n %d",pop(&s));
    printf("\n %d",pop(&s));

}
```

Single Pointer Linear Linked List

Dr. Rahul Das Gupta



Linked lists are preferable over arrays when:

1. you have huge free memory space available compare to required memory space, but it is not contiguous.
2. you need constant-time insertions/deletions from the list (such as in real-time computing where time predictability is absolutely critical)
3. you don't know how many items will be in the list. With arrays, you may need to re-declare and copy memory if the array grows too big

4. you don't need random access to any elements
5. you want to be able to insert items in the middle of the list (such as a priority queue)

Arrays are preferable when:

1. you need indexed/random access to elements
2. you know the number of elements in the array ahead of time so that you can allocate the correct amount of memory for the array
3. you need speed when iterating through all the elements in sequence. You can use pointer math on the array to access each element, whereas you need to lookup the node based on the pointer for each element in linked list, which may result in page faults which may result in performance hits.
4. memory is a concern. Filled arrays take up less memory than linked lists. Each element in the array is just the data. Each linked list node requires the data as well as one (or more) pointers to the other elements in the linked list.

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

/* ==DEFINITION  OF SELF-REFERENTIAL  STRUCTURE ==*/
typedef struct s_linked_list
{
  int data;
  struct s_linked_list *nxt;
} S_LINKED_LIST;
/* ========    END OF STRUCTURE DEFINITION ===========*/

/* =============   PROTOTYPE DECLERATION     =========*/
void initialisation (S_LINKED_LIST **);

void insertion_sorted_order (S_LINKED_LIST **,  int);
void insertion_serial_order (S_LINKED_LIST **,  int);

void recursive_insertion_serial_order (S_LINKED_LIST **,  int);
void recursive_insertion_sorted_order (S_LINKED_LIST **,  int);

void reverse (S_LINKED_LIST **);
void recursive_reverse (S_LINKED_LIST **);

int count (S_LINKED_LIST *);
int recursive_count (S_LINKED_LIST *);
```

void display (S_LINKED_LIST *);
void recursive_display (S_LINKED_LIST *);

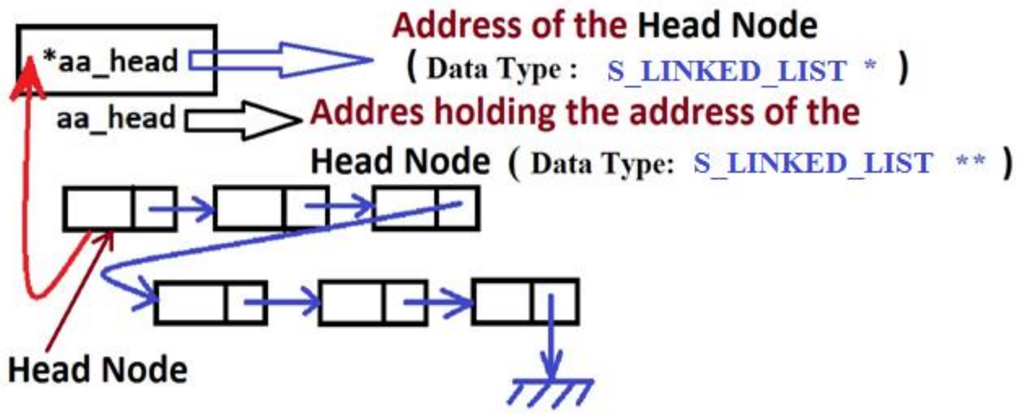
void deletion (S_LINKED_LIST **, int);

void split_and_display (S_LINKED_LIST *, S_LINKED_LIST **, S_LINKED_LIST **);

void merge_and_display(S_LINKED_LIST *, S_LINKED_LIST *, S_LINKED_LIST **);

void alternative_merge_and_display(S_LINKED_LIST *, S_LINKED_LIST *, S_LINKED_LIST **);

/* ==========   END OF PROTOTYPE DECLERATION   ========*/
/*==
IMPORTANT POINTS:
Whenever an item is inserted or deleted the address of the HEAD NODE may (not always) changed. Hence a function that attempts to change the address of the HEAD NODE must contains address holding the address of the HEAD NODE as an argument.
Functions of this kind are initialisation, insertion, reverse and deletion. All these functions contain address holding the address of the HEAD NODE as an argument.
==
*/



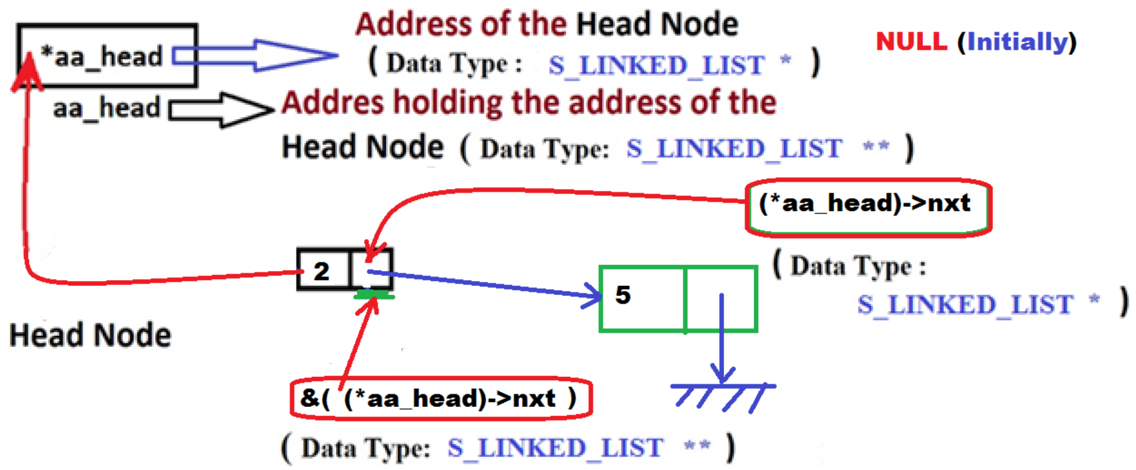
/*

aa_head = Address holding the address of the HEAD NODE.
(Data Type of aa_head : S_LINKED_LIST **)

***aa_head = Address of the HEAD NODE.**

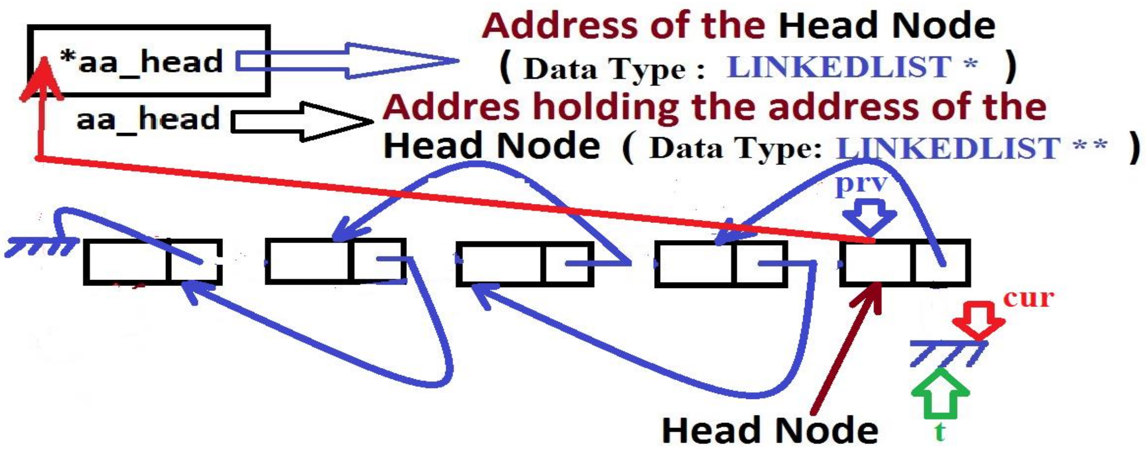
```
              (Data Type  of  *aa_head : S_LINKED_LIST *)
==============================================================
*/


void initialisation (S_LINKED_LIST **aa_head)
{
     *aa_head=NULL;
}


void  insertion_ sorted_order (S_LINKED_LIST **aa_head, int n)
{
    S_LINKED_LIST *t, *prv, *cur;
    t=(S_LINKED_LIST *) malloc (sizeof(S_LINKED_LIST));
    t->data=n;
    for(cur =*aa_head, prv=NULL; cur && n>cur->data  ; cur=cur->nxt)
        prv=cur;
    t->nxt=cur;
    if (prv!=NULL)
      prv->nxt=t;
    else
      *aa_head=t;
     /*The newly created node must be inserted at the beginning as head node.*/
}

void  insertion_serial_order (S_LINKED_LIST **aa_head, int n)
{
    S_LINKED_LIST *t, *prv, *cur;
    t=( S_LINKED_LIST *) malloc(sizeof(S_LINKED_LIST));
    t->data=n;
    for(cur=*aa_head, prv=NULL; cur ; cur=cur->nxt)
        prv=cur;
    t->nxt=cur;
    if (prv)
      prv->nxt=t;
    else
      *aa_head=t;
}


void recursive_insertion_serial_order (S_LINKED_LIST **aa_head, int n)
{
```

```
    if (*aa_head= =NULL)
    {
        *aa_head =( S_LINKED_LIST *) malloc(sizeof(S_LINKED_LIST));
        (*aa_head)->data=n;
        (*aa_head)->nxt=NULL;
    }
   else
      recursive_insertion_serial_order (&(*aa_head)->nxt, n);
}
```

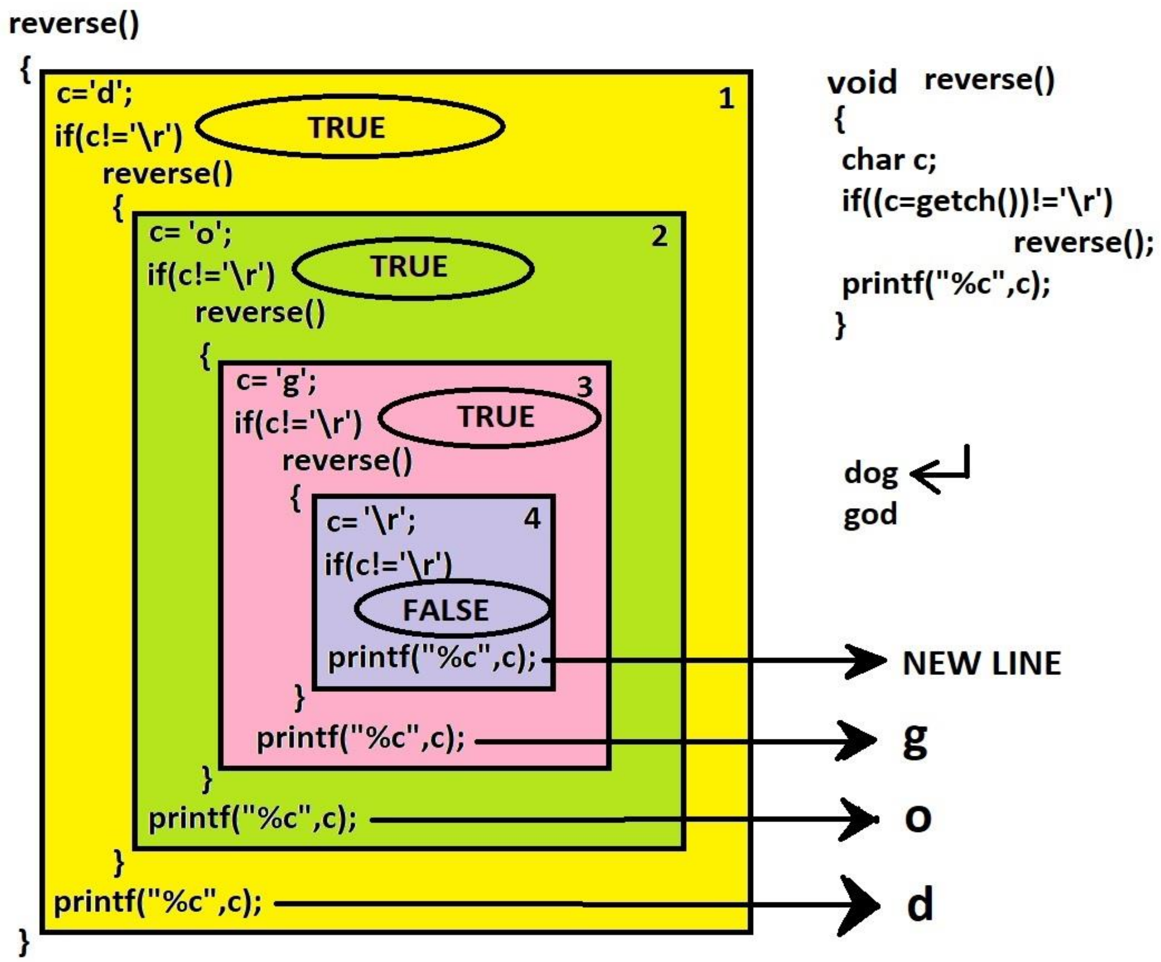



```
void recursive_insertion_sorted_order (S_LINKED_LIST **aa_head
, int n)
{
     S_LINKED_LIST *t;
     t=( S_LINKED_LIST *) malloc(sizeof(S_LINKED_LIST));
     t->data=n;
     t->nxt=NULL;
    if (*aa_head==NULL)
          *aa_head =t;
    else if (n < (*aa_head)->data)
    {
          /* next part of the new node will point to the existing head nde.*/
```

```
            t->nxt=*aa_head;
            *aa_head =t; /* New node will become head node. */
     }
    else
        recursive_insertion_sorted_order (&(*aa_head)->nxt, n);

}
```




```
void reverse (S_LINKED_LIST  **aa_head)
{
        S_LINKED_LIST  *prv , *cur, *t;
        for (cur=*aa_head, prv=NULL; cur; cur=t)
          {
               t=cur->nxt; /*keep the address of the head node at the pointer t */
               cur->nxt=prv;
               prv=cur;
          }
        *aa_head=prv;
}
```




```
void recursive_reverse (S_LINKED_LIST  *a_head )
{
    if (a_head->nxt)
      recursive_reverse (a_head ->nxt);
    printf("\t %d", a_head->data);
}
```

```
int count (S_LINKED_LIST *a_head);
{
    int count=0;
    for( ; a_head; a_head = a_head->nxt)
        count++;
    return (count);
}
```

```
int recursive_count (S_LINKED_LIST  * a_head)
{
   if (a_head = = NULL)
      return 0;
   else
      return (recursive_count (a_head->nxt) + 1);
}

void display (S_LINKED_LIST  *a_head)
{
     for( ; a_head ; a_head = a_head->nxt)
            printf("\t %d", a_head->data);
}


void recursive_display (S_LINKED_LIST  *a_head)
{
       if (a_head)
       {
            printf("\t %d", a_head->data);
            recursive_display (a_head->nxt);
        }
       else
            printf("\t END. ");
}

void   deletion (S_LINKED_LIST  **aa_head,  int  n)
{
      S_LINKED_LIST  *prv, *cur;
      int found=0;
      for (cur = *aa_head, prv=NULL; cur; cur=cur->nxt)
      {
           if (prv != NULL  && n = = cur->data)
           {
               found=1;
               prv->nxt = cur->nxt;
               free(cur);
               return;
           }
          else if (prv = = NULL  && n = = cur->data)
          {
               found=1;
               *aa_head = cur->nxt;
```

```
                free(cur);
                return;
           }
          else
              prv=cur;
      }
     if (!found)
           printf ("\n Data not found…");
}

void split_and_display (S_LINKED_LIST *a_head,
S_LINKED_LIST **aa_head_odd, S_LINKED_LIST **aa_head_even)
{
         S_LINKED_LIST *cur;
        initialisation (aa_head_even);
        initialisation (aa_head_odd);
        for (cur = a_head; cur; cur=cur->nxt)
        {
            if(cur->data%2==0)
                  insertion_serial_order (aa_head_even, cur->data);
            else
                  insertion_serial_order (aa_head_odd, cur->data);
        }
        display(*aa_head_even);
        display(*aa_head_odd);

}


void merge_and_display(S_LINKED_LIST *a_head1,
S_LINKED_LIST *a_head2, S_LINKED_LIST **aa_head_m)
{
        S_LINKED_LIST *cur;
        initialisation (aa_head_m);
        for (cur=a_head1; cur ;cur=cur->nxt)
              insertion_ sorted_order (aa_head_m, cur->data);
        for (cur=a_head2; cur ;cur=cur->nxt)
              insertion_ sorted_order (aa_head_m, cur->data);
        display(*aa_head_m);
}
```

```
void alternative_merge_and_display(S_LINKED_LIST *a_head1,
S_LINKED_LIST *a_head2, S_LINKED_LIST **aa_head_m)
{
        S_LINKED_LIST *cur1,*cur2;
        initialisation (aa_head_m);
        for (cur1=a_head1, cur2=a_head2; cur1 && cur2; )
        {
          if (cur1->data < cur2->data)
          {
             insertion_serial_order (aa_head_m, cur1->data);
             cur1=cur1->nxt;
          }
          else
          {
             insertion_serial_order (aa_head_m, cur2->data);
             cur2=cur2->nxt;
          }
        }
        for ( ;cur1; cur1=cur1->nxt)
              insertion_serial _order (aa_head_m, cur1->data);
        for ( ; cur2; cur2=cur2->nxt)
              insertion_serial _order (aa_head_m, cur2->data);

        display(*aa_head_m);
}




void main( )
{
  S_LINKED_LIST   *L=NULL;

  initialisation (&L);

  insertion_sorted_order (&L,  12);
  insertion_sorted_order (&L,  4);
  insertion_sorted_order (&L,  13);
  insertion_sorted_order (&L,  6);
  insertion_sorted_order (&L,  2);
  insertion_sorted_order (&L,  5);
```

```
    insertion_sorted_order (&L,  1);
    insertion_sorted_order (&L,  11);
    insertion_sorted_order (&L,  8);
    insertion_sorted_order (&L,  6);
    insertion_sorted_order (&L,  3);
    insertion_sorted_order (&L,  9);
    insertion_sorted_order (&L,  15);



    display(L);

    deletion(&L, 2);
    deletion(&L, 6);
    deletion(&L, 8);
    deletion(&L, 9):

    display(L);
}
```