Biomedical Engineering Institute, Bogazici University Instructor: Ahmet Ademoglu, PhD

BM 593 Numerical Methods & C Programming

2nd week Basic Syntax of C Language

Arithmetic Operators

```
int x,y;


x=1;
x+=3;
x=y-3;
x=x*y;
x/y;
x%y;
++x;
x++;
--x;
--y;
y=-x;
```

Relational and Logical Operators

```
(x>y) /* 1 if x greater than y else 0 */
(x>=y) /* 1 if x greater than or equal to y else 0 */
(x<y) /* 1 if x less than y else 0 */
(x<=y) /* 1 if x less than or equal to y else 0 */
(x0=y) /* 1 if x equals y else 0 */
(x!=y) /* 1 if x is not equal y else 0 */
(!x)   /* 1 if x is  equal 0 else 0 */
(x&&y) /* 1 if both x and y are 1 else 0 */
(x||y) /* 1 if either x or y or both are 1 else 0 */


z ? x:y /** If z is 1 than evaluate x else evaluate y  */


Ex: if (x>y) ? (max=x) : (max=y); /* Finds whichever is greater between x and y  */


(type) x /* changes the value of x to type */


Ex: int x=2; float y; y = (float) x /* Finds whichever is greater between x and y  */
```

Bitwise Operators

```
~x /* Complements all bits */
x&y /* bitwise AND */
x|y /* bitwise OR */
x^y /* bitwise EXCLUSIVE OR */
x>>y /* shift to right by y bits or divide by 2^y */
x<<y /* shift to left by y bits or multiply by 2^y */
```

Command Line Arguments

```
void main(int argc, char *argv[]){
  /* File Copy */
  char c;
  FILE *f1,*f2;

  if (argc == 3) {
    f1=fopen(*++argv,"r");
    f2=fopen(*++argv,"w");
    c=getc(f1);
    while (c ==!EOF){
      putc(c,f2);
      c=getc(f1);
    }
    fclose(f1);
    fclose(f2);
  }
  else
    printf("Usage: copy inputfile outputfile\n");
}
```

Macro Substitution for fast exection

```
#define max(A,B) ((A)>(B) ? (A) : (B))

#define SQR(A) ((A)*(A))
#define EUCLID(A,B) ( sqrt( (SQR(A))+(SQR(B)) ) )

#define SIGN(A,B) ( (B) > 0 ? fabs(A) : -fabs(A))

#define SWAP(A,B) {float temp=(A);(A)=(B);(B)=temp;}

#define IGREG (15+31L*(12L+1582))
```

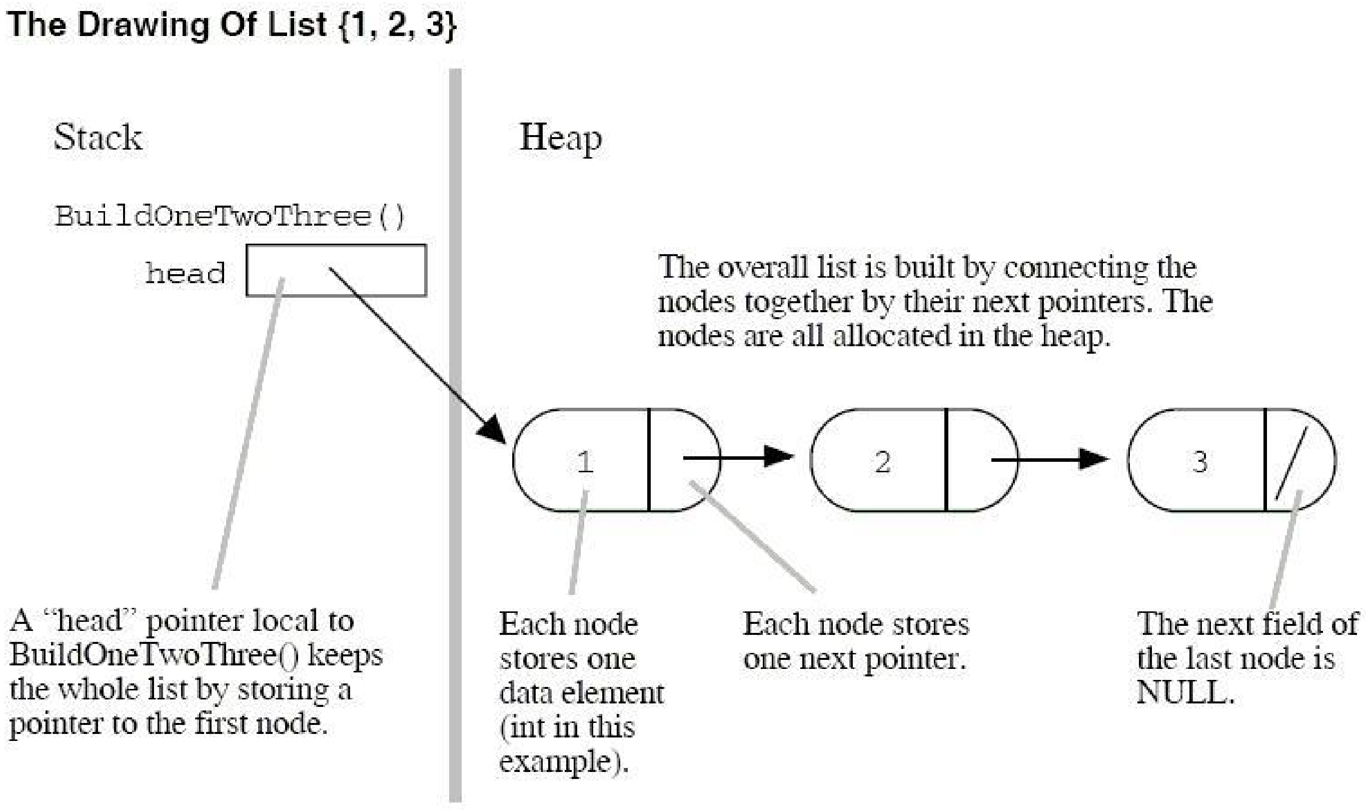
Function Pointers

```
#include <stdio.h>


int trapezoid (*func)(int), int, int);
int myfunction (int);
int yourfunction (int);


void main(int argc, char *argv[]){
  int x=1;
  int y=2;


  printf ("My trapezoidal value is %d\n",trapezoid(myfunction,x,y));
  printf ("Your trapezoidal value is %d\n",trapezoid(yourfunction,x,y));
}


int trapezoid(int (*func)(int), int a, int b){
  return ( (a+b)*(*func)(3)) ;
}


int myfunction(int x){
  return (1+x)
}


int yourfunction(int x){
  return (1+3*x)
}


#include <stdio.h>
#include <stdlib.h>  /* qsort is defined in stdlib.h */
#define IMAXVALUES 10


int icompare_func(const void *, const void *);
int (*ifunct_ptr)(const void *, const void *);


void main(){
  int i;
  int iarray[IMAXVALUES]={0,5,3,2,8,7,9,1,4,6};
  ifunct_ptr=icompare_funct;
  qsort(iarray,IMAXVALUES,sizeof(int),ifunct_ptr);
  for (i=0; i<IMAXVALUES;i++)
    printf("%d ",iarray[i]);
}

int icompare_funct(const void *iresult_a, const void *iresult_b){
  return ((*(int *)iresult_a) - (*(int *) iresult_b) );
}

int *(*(*ifunct_ptr)(int))[5];
/* a function pointer to a function that passed an integer argument and returns a pointer to
array of 5 int pointers*/
float (*(*ffunct_ptr)(int,int))(float);
/* a function pointer to a function that takes two arguments and returns a pointer to
a function taking a float argument and returning a float*/
typedef double (*(*(*dfunct_ptr)())[5])();
/* dfunct_ptr() is defined as a pointer to a function that is passed nothing and returns
a pointer to an array of 5 pointers that point to functions that are
passed nothing and return  double */
  dfunct_ptr A_dfunct_ptr;
int (*(*function_ary_ptrs())[5])();
/* a function declaration function_ary_ptrs() that takes no arguments and returns a
pointer to an array of five pointers that point to functions
taking no arguments and returning integers */
```

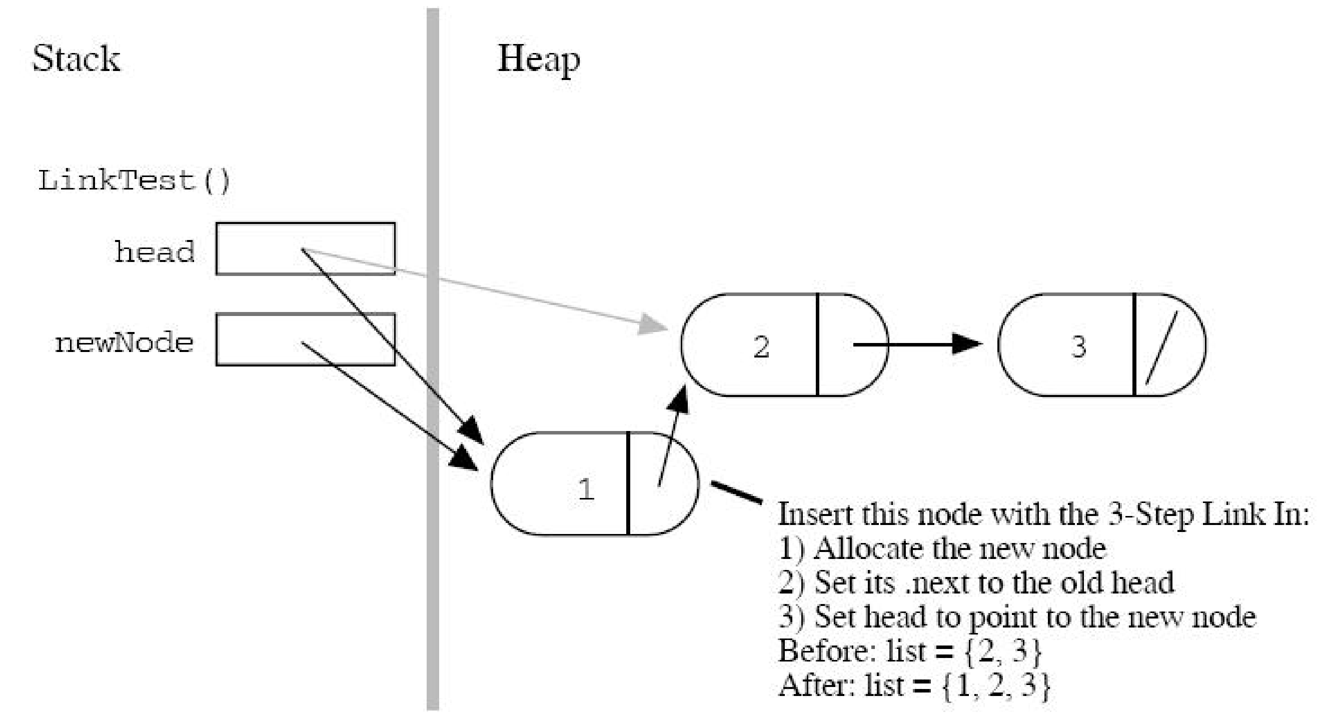
Linked Lists

```
struct node {
  int data;
  struct node* next;
}
/*
Build the list {1, 2, 3} in the heap and store
its head pointer in a local stack variable.
Returns the head pointer to the caller.
*/
struct node* BuildOneTwoThree() {
struct node* head = NULL;
struct node* second = NULL;
struct node* third = NULL;
head = malloc(sizeof(struct node)); // allocate 3 nodes in the heap
second = malloc(sizeof(struct node));
third = malloc(sizeof(struct node));
head->data = 1; // setup first node
head->next = second; // note: pointer assignment rule
second->data = 2; // setup second node
second->next = third;
third->data = 3; // setup third link
third->next = NULL;
// At this point, the linked list referenced by "head"
// matches the list in the drawing.
return head;
}
```

Exercise Q: Write the code with the smallest number of assignments (=) which will build the above memory structure. A: It requires 3 calls to malloc(). 3 int assignments (=) to setup the ints. 4 pointer assignments to setup head and the 3 next fields. With a little cleverness and knowledge of the C language, this can all be done with 7 assignment operations (=). /* How can you write the code with smallest number of assignments (=)? */


The Drawing Of List {1, 2, 3}
Stack
Heap
BuildOneTwoThree()
head
The overall list is built by connecting the nodes together by their next pointers. The nodes are all allocated in the heap.
1
2
3
A "head" pointer local to BuildOneTwoThree() keeps the whole list by storing a pointer to the first node.
Each node stores one data element (int in this example).
Each node stores one next pointer.
The next field of the last node is NULL.

```
void LinkTest() {
struct node* head = BuildTwoThree(); // suppose this builds the {2, 3} list
struct node* newNode;
newNode= malloc(sizeof(struct node)); // allocate
newNode->data = 1;
newNode->next = head; // link next
head = newNode; // link head
// now head points to the list {1, 2, 3}
}
```


Stack
Heap
LinkTest()
head
newNode
2
3
1
Insert this node with the 3-Step Link In:
1) Allocate the new node
2) Set its .next to the old head
3) Set head to point to the new node
Before: list = {2, 3}
After: list = {1, 2, 3}

```
void WrongPush(struct node* head, int data) {
struct node* newNode = malloc(sizeof(struct node));
newNode->data = data;
newNode->next = head;
head = newNode; // NO this line does not work!
}
void WrongPushTest() {
List head = BuildTwoThree();
WrongPush(head, 1); // try to push a 1 on front -- doesn't work
}

/* Correct PushTest
```

```
Takes a list and a data value.
Creates a new link with the given data and pushes
it onto the front of the list.
The list is not passed in by its head pointer.
Instead the list is passed in as a "reference" pointer
to the head pointer -- this allows us
to modify the caller's memory.
*/
void Push(struct node** headRef, int data) {
struct node* newNode = malloc(sizeof(struct node));
newNode->data = data;
newNode->next = *headRef; /* The '*' to dereferences back to the real head  */
*headRef = newNode;
}
void PushTest() {
struct node* head = BuildTwoThree();/* suppose this returns the list {2, 3} */
Push(&head, 1); /* note the & */
Push(&head, 13);
/* head is now the list {13, 1, 2, 3} */
}
```

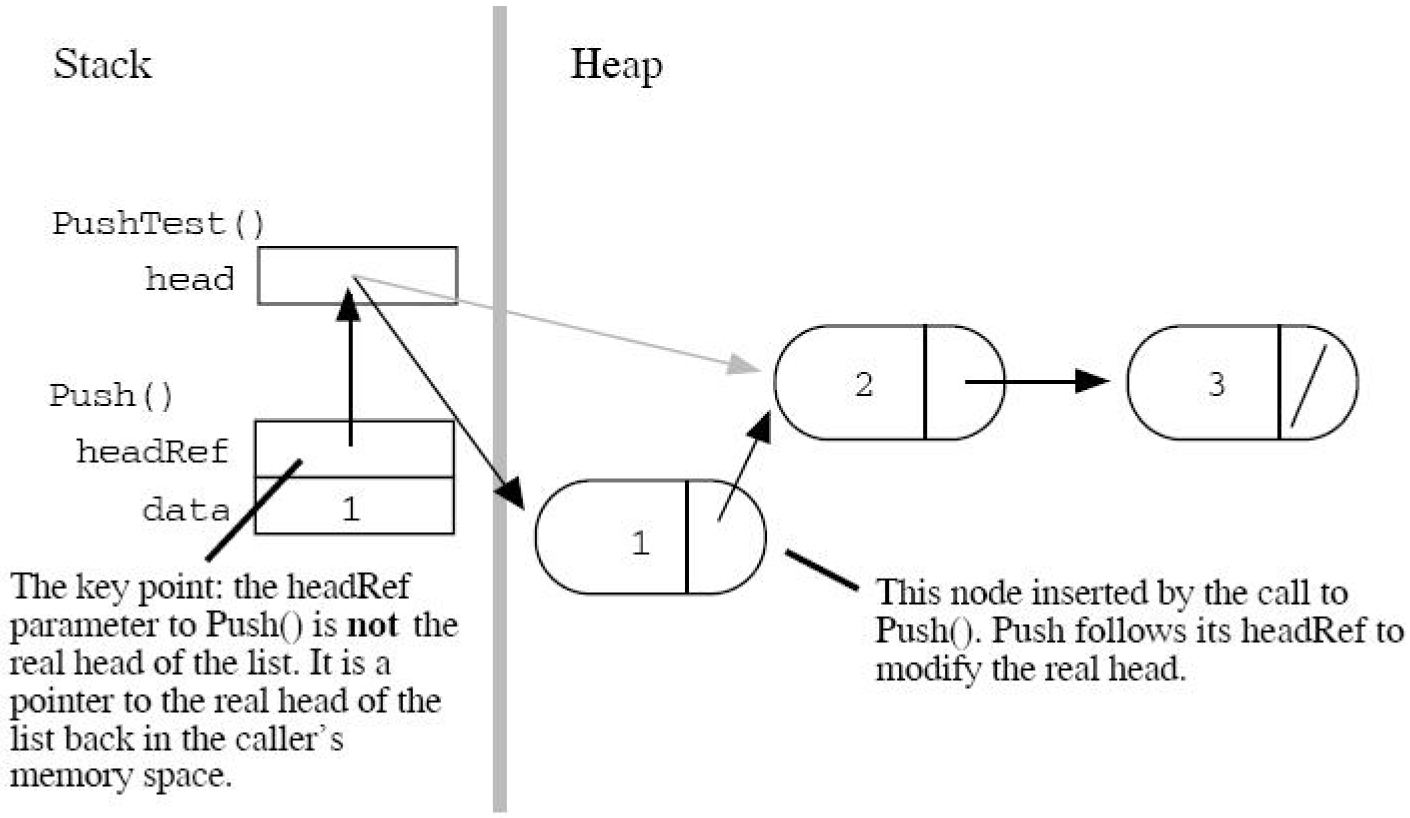
Exercise

The above drawing shows the state of memory at the end of the first call to `Push()` in `PushTest()`. Extend the drawing to trace through the second call to `Push()`. The result should be that the list is left with elements {13, 1, 2, 3}.

Exercise

The following function correctly builds a three element list using nothing but `Push()`. Make the memory drawing to trace its execution and show the final state of its list. This will also demonstrate that `Push()` works correctly for the empty list case.

```
void PushTest2() {
struct node* head = NULL; // make a list with no elements
Push(&head, 1);
Push(&head, 2);
Push(&head, 3);
/* head now points to the list {3, 2, 1} */
}


struct node* AppendNode(struct node** headRef, int num) {
```


Stack
Heap
PushTest()
head
Push()
headRef
data
1
1
2
3
The key point: the headRef parameter to Push() is **not** the real head of the list. It is a pointer to the real head of the list back in the caller's memory space.
This node inserted by the call to Push(). Push follows its headRef to modify the real head.

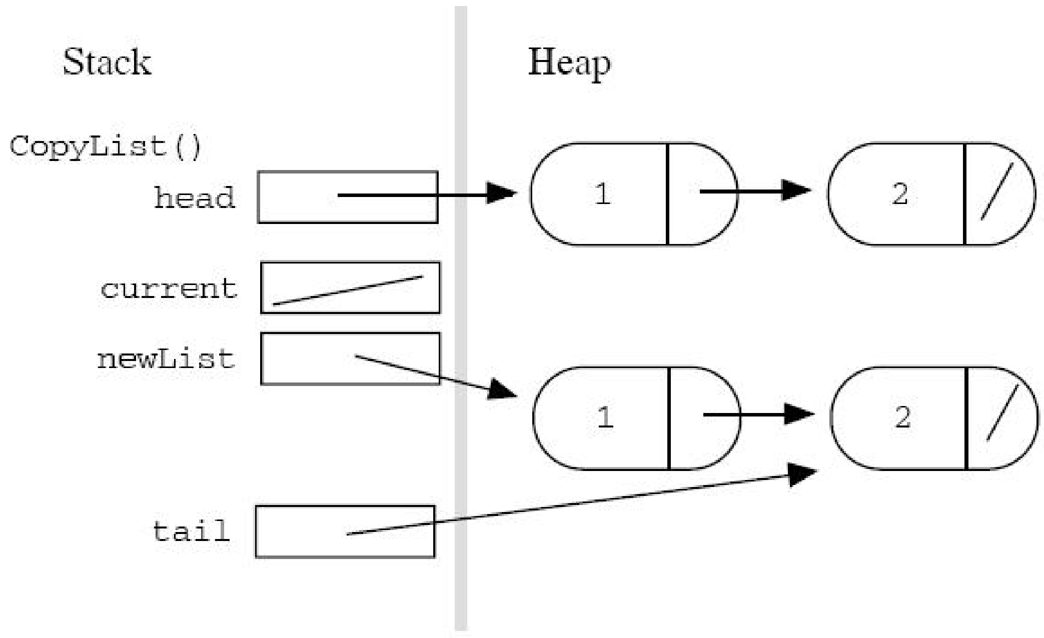
```
struct node* current = *headRef;
struct node* newNode;
newNode = malloc(sizeof(struct node));
newNode->data = num;
newNode->next = NULL;
/* special case for length 0 */
if (current == NULL) {
*headRef = newNode;
}
else {
/* Locate the last node */
while (current->next != NULL) {
current = current->next;
}
current->next = newNode;
}
}
```

```
struct node* AppendNode(struct node** headRef, int num) {
struct node* current = *headRef;
/* special case for the empty list */
if (current == NULL) {
Push(headRef, num);
} else {
/* Locate the last node */
while (current->next != NULL) {
current = current->next;
}
/* Build the node after the last node */
Push(&(current->next), num);
}
}
```

```
struct node* CopyList(struct node* head) {
struct node* current = head; /*used to iterate over the original list*/
struct node* newList = NULL; /* head of the new list */
struct node* tail = NULL; /* kept pointing to the last node in the new list */
while (current != NULL) {
if (newList == NULL) { /* special case for the first new node */
newList = malloc(sizeof(struct node));
newList->data = current->data;
newList->next = NULL;
tail = newList;
}
else {
tail->next = malloc(sizeof(struct node));
tail = tail->next;
tail->data = current->data;
tail->next = NULL;
}
current = current->next;
}
return(newList);
}
```

`CopyList()` With `Push()` Exercise

The above implementation is a little unsatisfying because the 3-step-link-in is repeated once for the first node and



once for all the other nodes. Write a `CopyList2()` which uses Push() to take care of allocating and inserting the new nodes, and so avoids repeating that code.

`CopyList()` With `Push()` Answer

```
/* Variant of CopyList() that uses Push() */
struct node* CopyList2(struct node* head) {
struct node* current = head; /* used to iterate over the original list */
struct node* newList = NULL; /* head of the new list */
struct node* tail = NULL; /* kept pointing to the last node in the new list */
while (current != NULL) {
if (newList == NULL) { /* special case for the first new node */
Push(&newList, current->data);
tail = newList;
}
else {
Push(&(tail->next), current->data); // add each node at the tail
tail = tail->next; // advance the tail to the new last node
}
current = current->next;
}
return(newList);
}
```

```
/* Dummy node variant */
struct node* CopyList(struct node* head) {
struct node* current = head; /* used to iterate over the original list */
struct node* tail; /* kept pointing to the last node in the new list */
struct node dummy; /* build the new list off this dummy node */
dummy.next = NULL;
tail = &dummy; /* start the tail pointing at the dummy */
while (current != NULL) {
Push(&(tail->next), current->data); /* add each node at the tail */
tail = tail->next; /* advance the tail to the new last node */
}
current = current->next;
}
return(dummy.next);
}
```

```
/* Local reference variant */
struct node* CopyList(struct node* head) {
struct node* current = head; /* used to iterate over the original list */
struct node* newList = NULL;
struct node** lastPtr;
lastPtr = &newList; /* start off pointing to the head itself */
while (current != NULL) {
Push(lastPtr, current->data); /* add each node at the lastPtr */
lastPtr = &((*lastPtr)->next); /* advance lastPtr */
current = current->next;
}
return(newList);
}
```

```
/* Recursive variant */
struct node* CopyList(struct node* head) {
if (head == NULL) return NULL;
else {
struct node* newList = malloc(sizeof(struct node)); /* make the one node */
newList->data = current->data;
newList->next = CopyList(current->next); /* recur for the rest */
return(newList);
}
}


#include <stdlib.h>
#include <stdio.h>
struct tree_el{
  int val;
  struct tree_el *right,*left;
}
typedef struct tree_el node;
void insert(node **tree, node *item){
  if (!(*tree)){
    *tree=item;
    return;
  }
if (item->val < (*tree)->val)
  insert(&(*tree)->left,item);
else if (item->val > (*tree)->val)
   insert(&(*tree)->right,item);
}
void printout(node *tree){
 if (tree->left) printout(tree->left);
 printf("%d\n",tree->val);
 if (tree->right) printout(tree->right);
}
void main(){
  node *curr, *root;
  int i;
  root = NULL;
  for (i=1;i<=10;i++){
    curr=(node*)malloc(sizeof(node));
    curr->left=curr->right=NULL;
    curr->val=rand();
    insert(&root,curr);
  }
  printout(root);
}
```