Stack - Linked implementation

```
typedef char StackEntry;
typedef struct stacknode{
      StackEntry entry;
      struct stacknode *next;
}StackNode;

typedef struct stack{
      StackNode *top;
}Stack;

void Push(StackEntry item, Stack *ps){
      StackNode *p;
      p=(StackNode *)malloc(sizeof(StackNode));
      p->entry=item;
      p->next=ps->top;
      ps->top=p;
}

void Pop(StackEntry *pitem, Stack *ps){
      StackNode *p;
      *pitem=ps->top->entry;
      p=ps->top;
      ps->top=ps->top->next;
      free(p);
}

int StackEmpty(Stack ps){
      return ps.top==NULL;
}

int StackFull(Stack ps){
      return 0;
}

void CreateStack (Stack *ps){
      ps->top=NULL;
}

void ClearStack(Stack *ps){
      StackNode *p=ps->top;
      while(p){
            p=p->next;
            free(ps->top);
            ps->top=p;
      }
}
```

Stack - Array-based implementation

```
#define MAXSTACK 10
typedef char StackEntry;
typedef struct stack{
      int top;
      StackEntry entry[MAXSTACK];
} Stack;

void Push(StackEntry item, Stack *ps){
      ps->entry[ps->top++] = item;
}

void Pop(StackEntry *item, Stack *ps){
      *item = ps->entry[--ps->top];
}

int StackEmpty(Stack ps){
     return (ps.top == 0)
}

int StackFull(Stack ps){
      return ps.top==MAXSTACK;
}

void CreateStack(Stack *ps){
     ps->top = 0;
}

void ClearStack(Stack *ps){
     ps->top = 0;
}
```

Queue - Array-based implementation

```
#define MAXQUEUE 10
typedef char QueueEntry;

typedef struct queue{
   int front;
   int rear;
   int size;
   QueueEntry entry[MAXQUEUE];
}Queue;

void CreateQueue(Queue *pq){
      pq->front= 0;
      pq->rear = -1;
      pq->size = 0;
}

void Append(QueueEntry e, Queue* pq){
  pq->rear = (pq->rear + 1) % MAXQUEUE;
  pq->entry[pq->rear] = e;
  pq->size++;
}

void Serve(QueueEntry *pe, Queue* pq){
  *pe = pq->entry[pq->front];
  pq->front = (pq->front + 1) % MAXQUEUE;
  pq->size--;
}

int QueueEmpty(Queue pq){
      return !pq.size;
}

int QueueFull(Queue pq){
      return (pq.size == MAXQUEUE);
}

int QueueSize(Queue pq){
      return pq.size;
}

void ClearQueue(Queue* pq){
      pq->front = 0;
      pq->rear  = -1;
      pq->size  = 0;
}
```

Queue - Linked implementation

```
typedef char QueueEntry;

typedef struct queuenode{
      QueueEntry entry;
      struct queuenode *next;
}QueueNode;

typedef struct queue{
      QueueNode *front;
      QueueNode *rear;
      int   size;
}Queue;

void CreateQueue(Queue *pq){
   pq->front=NULL;
   pq->rear=NULL;
   pq->size=0;}

void Append(QueueEntry e, Queue* pq){
  QueueNode*pn=(QueueNode*)malloc(sizeof(QueueNode));
  pn->next=NULL;
  pn->entry=e;
  if (!pq->rear)
    pq->front=pn;
  else
    pq->rear->next=pn;//run time error for empty queue
  pq->rear=pn;
  pq->size++;}

void Serve(QueueEntry *pe, Queue* pq){
      QueueNode *pn=pq->front;
      *pe=pn->entry;
      pq->front=pn->next;
      free(pn);
      if (!pq->front)
            pq->rear=NULL;
      pq->size--;}

int QueueEmpty(Queue pq){
      return !pq.front;}

int QueueFull(Queue pq){
      return 0;}

int QueueSize(Queue pq){
      return pq.size;}

void ClearQueue(Queue* pq){
      while(pq->front){
            pq->rear=pq->front->next;
            free(pq->front);
            pq->front=pq->rear;
      }
      pq->size  = 0; }
```

List - Array-based implementation

```
typedef struct list{
      ListEntry entry[MAXLIST];
      int size;
}List;

void CreateList(List *pl){
      pl->size=0;
}

int ListEmpty(List pl){
      return !pl.size;
}

int ListFull(List pl){
      return pl.size==MAXLIST;
}

int ListSize(List pl){
      return pl.size;
}

void DestroyList(List *pl){
      pl->size=0;
}

void InsertList(int p, ListEntry e, List *pl){ /*0 <= p <= size*/
      int i;
      /*The loop shifts up all the elements in      the range [p,
      size-1] to free the p$^{th}$      location*/
      for(i=pl->size-1; i>=p; i--)
            pl->entry[i+1]=pl->entry[i];
      pl->entry[p]=e;
      pl->size++;
}

void DeleteList(int p, ListEntry *pe, List *pl){
      /*0<= p <= size-1 and List not empty*/
      int i;
      *pe=pl->entry[p];
      /*The loop shifts down all the elements in    the range
      [p+1, size-1] to free the p$^{th}$      location*/
      for(i=p+1; i<=pl->size-1; i++)
            pl->entry[i-1]=pl->entry[i];
      pl->size--;
}
```

List - Linked implementation

```
typedef struct listnode{
      ListEntry entry;
      struct listnode *next;
}ListNode;

typedef struct list{
      ListNode   *head;
      int        size;
}List;

void CreateList(List *pl){
      pl->head=NULL;
      pl->size=0;
}

int ListEmpty(List pl){
      return (pl.size==0);
      //or return !pl.head
}

int ListFull(List pl){
      return 0;
}

int ListSize(List pl){
      return pl.size;
}

void DestroyList(List *pl){
      ListNode *q;
      while(pl->head){
            q=pl->head->next;
            free(pl->head);
            pl->head=q;
      }
      pl->size=0;
}

void InsertList(int pos, ListEntry e, List *pl){
   ListNode *p, *q;
   int i;
   p=(ListNode *)malloc(sizeof(ListNode));
   p->entry=e;
   p->next=NULL;
   if (pos==0){//will work also for head equals NULL
      p->next=pl->head;
      pl->head=p;
   }
   else{
      for(q=pl->head, i=0; i<pos-1; i++)
            q=q->next;
      p->next=q->next;
      q->next=p;
   }
   pl->size++;
}
```

```
void DeleteList(int pos, ListEntry *pe, List *pl){
   int i;
   ListNode *q, *tmp;

   if (pos==0){
      *pe=pl->head->entry;
      tmp=pl->head->next;
      free(pl->head);
      pl->head=tmp;
   }// it works also for one node
   else{
      for(q=pl->head, i=0; i<pos-1; i++)
            q=q->next;
      *pe=q->next->entry;
      tmp=q->next->next;
      free(q->next);
      q->next=tmp;
   }// check for pos=size-1 (tmp will be NULL)
   pl->size--;
}
```

Tree implementation

```
typedef struct treenode{
      TreeEntry entry;
      struct treenode *left, *right;
}TreeNode;
typedef TreeNode * Tree;

void CreateTree(Tree *pt){
      *pt=NULL;
}

int TreeEmpty(Tree pt){
      return (!pt);
}

int TreeFull(Tree pt){
      return 0;
}

void Inorder(Tree *pt){
    if (*pt){
      Inorder((*pt)->left);
      Visit(*pt)
      Inorder((*pt)->right);
    }
}

/*An iterative version of the function*/
void Inorder(Tree *pt){
   Stack s;
   TreeNode *p=pt;
   if(p){
      CreateStack(&s);
      do{
         while(p){
           Push(p, &s);
           p=p->left;
        }
         Pop(&p, &s);
         Visit(p);
         p=p->right);
        }
      }while(!StackEmpty(&s) || p);
   }
}

void ClearTree(Tree *pt){
      if (*pt){
            ClearTree(&(*pt)->left);
            ClearTree(&(*pt)->right);
            free(*pt);
            *pt=NULL;
      }
}
```

```
int TreeSize(Tree pt){
   if (!pt)
      return 0;
   return (1+TreeSize(pt->left)+
            TreeSize(pt->right));
}

int TreeDepth(Tree pt){
   if (!pt)
      return 0;
    int a=TreeDepth(pt->left);
    int b=TreeDepth(pt->right);
    return (a>b)? 1+a : 1+b;
}

void InsertTree(Tree *pt, TreeEntry pe){
      if (!*pt){
            *pt=(Tree)malloc(sizeof(TreeNode));
            (*pt)->entry=pe;
            (*pt)->left=NULL;
            (*pt)->right=NULL;
      }else if (pe<(*pt)->entry))
            InsertTree(&(*pt)->left, pe);
      else
            InsertTree(&(*pt)->right, pe);
}

/*An iterative version of the function*/
void InsertTree(Tree *pt, TreeEntry pe){
   TreeNode *p, *prev, *curr;
   p=(TreeNode *)malloc(sizeof(TreeNode));
   p->entry=pe;  p->left=NULL;          p->right=NULL;
   if (!(*pt))
      (*pt) =p;
   else{
      curr=(*pt);
      while(curr){
         prev=curr;
         if(pe<curr->entry))
            curr=curr->left;
         else
            curr=curr->right;
      }
      if(pe<prev->entry))
         prev->left=p;
      else
         prev->right=p;
   }
}
```

```
int DeleteItemTree(Tree *pt, TreeEntry k){
   int found=0;        TreeNode *q=*pt, *r=NULL;
   while(q && !(found=(k==q->entry))){
      r=q;
      if(k<q->entry)
         q=q->left;
      else
         q=q->right;
   }
   if (found){
      *pe=q->entry;
      if(!r)//Case of deleting the root
         DeleteNodeTree(pt);
      else if(k< (r->entry))
         DeleteNodeTree(&r->left);
      else
         DeleteNodeTree(&r->right);
   }
   return found;
}

void DeleteNodeTree(Tree *pt){//Very inefficient version
   TreeNode *q=*pt, *r;
   if(!(*pt)->left)//First case
      *pt=(*pt)->right;
   else if(!(*pt)->right)//Second case
      *pt=(*pt)->left;//Also, both account for a leaf node
   else{//third case
         for(r=q->right; r->left; r=r->left);
         r->left=q->left;
         *pt=(*pt)->right;
   }
   free(q);
}

void DeleteNodeTree(Tree *pt){ //an efficient version
   TreeNode *q=*pt, *r=Null;
   if(!(*pt)->left)//First case
      *pt=(*pt)->right;
   else if(!(*pt)->right)//Second case
      *pt=(*pt)->left;//Also, both account for a leaf node
   else{//third case
      q=(*pt)->left;
      while(q->right){
            r=q;
            q=q->right;
         }
      (*pt)->entry=q->entry;
      if(!r){
         (*pt)->left=q->left;
      }
      else{
         r->right=q->left;
      }
   }
   free(q);
}
```