ASSIGN 1

```
//assignment 1
#include<stdio.h>
#include<stdlib.h>
typedef struct
{
	int rpart,ipart;
}cplx;
typedef void*(*fun)(void*,void*);
void* cplx_sum(void*,void*);
void* int_sum(void*,void*);
void* float_sum(void*,void*);
void* sum_two_nos(void*,void*,fun);
fun getfun(int);
void* cplx_sum(void*p1,void*p2)
{
	cplx*pc1=(cplx*)p1;
	cplx*pc2=(cplx*)p2;
   cplx*pc3=(cplx*)malloc(sizeof(cplx));
   pc3->rpart=pc1->rpart+pc2->rpart;
   pc3->ipart=pc1->ipart+pc2->ipart;
   return pc3;
}
void* int_sum(void*p1,void*p2)
{
	int *pi1=(int*)p1;
	int *pi2=(int*)p2;
	int *pi3=(int*)malloc(sizeof(int));
	*pi3=*pi1+*pi2;
	return pi3;
```

```
}
void* float_sum(void*p1,void*p2)
{
	float*pf1=(float*)p1;
	float*pf2=(float*)p2;
	float*pf3=(float*)malloc(sizeof(float));
	*pf3=*pf1+*pf2;
	return pf3;
}
fun getfun(int choice)
{
	switch(choice)
	{
		case 1:return(&cplx_sum);
			break;
		case 2:return(&int_sum);
			break;
		case 3:return(&float_sum);
			break;
		default:printf("fp=NULL");
	}
	return NULL;
}
void* sum_two_nos(void*op1,void*op2,fun fp)
{
	return (fp(op1,op2));
}
int main()
{
	fun fp;
	cplx*pc1;
```

```
	cplx*pc2;
	cplx*pcresult;
	int *pi1,*pi2,*piresult;
	float *pf1,*pf2,*pfresult;
	int choice;
	printf("enter your choice\n");
	printf("enter 1.complex number addition\t 2.integer addition 3.float addition ");
	scanf("%d",&choice);
	fp=getfun(choice);
   switch(choice)
   {
		case 1:printf("enter real and imaginary part of complex number 1\n");
			pc1=(cplx*)malloc(sizeof(cplx));
			scanf("%d%d",&(pc1->rpart),&(pc1->ipart));
			printf("enter real and imaginary part of complex number 2\n");
			 pc2=(cplx*)malloc(sizeof(cplx));
			scanf("%d%d",&(pc2->rpart),&(pc2->ipart));
			pcresult=(cplx*)sum_two_nos(pc1,pc2,fp);
			printf("real part =%d\t   imaginary part=%d\n",pcresult->rpart,pcresult->ipart);
			break;
	case 2:printf("enter two integers \n");
		pi1=(int*)malloc(sizeof(int));
		scanf("%d",pi1);
		pi2=(int*)malloc(sizeof(int));
		scanf("%d",pi2);
		piresult=(int*)sum_two_nos(pi1,pi2,fp);
		printf("sum of %d and %d is %d\n",*pi1,*pi2,*piresult);
		break;
	case 3:printf("enter two numbers\n");
		pf1=(float*)malloc(sizeof(float));
```

```
                scanf("%f",pf1);
                pf2=(float*)malloc(sizeof(float));
                scanf("%f",pf2);
                pfresult=(float*)sum_two_nos(pf1,pf2,fp);
                printf("sum = %f\n",*pfresult);
                break;
        }

                return 0;
}
```

ASSIGN 2

//Recursion assignment 2

```
#include<stdio.h>
int sum(int);
int fib(int);
int dec(int);
int main()
{
        int n,a,c,ch;
        while(1)
        {
        printf("enter choice-1-sum of n numbers\t 2-fibonacci \t 3-decimal to binary\n");
        scanf("%d",&ch);
        switch(ch)
        {case 1:printf("enter n value\n");
                scanf("%d",&n);
          a=sum(n);
                printf("the sum of 1st %d numbers is %d\n",n,a);
                break;
```

```
        case 2:printf("enter n value\n");
                scanf("%d",&n);
                printf("the  first %d fibanocci series is : \n",n);
                for(int i=0;i<n;i++)
                {
                                        printf("%d\t",fib(i));
                                }
                                printf("\n");
                break;
        case 3:printf("enter n value\n");
                scanf("%d",&n);
                c=dec(n);
                printf("the binary of %d is %d\n",n,c);
                break;
        default:return 0;
        break;
}
}
}
int sum(int n)
{
        if(n==0)
        return 0;
        else
        return (n+sum(n-1));
}
int fib(int n)
{
        if(n==0||n==1)
        return n;
        else
```

```
	return(fib(n-1)+fib(n-2));
}
int dec(int n)
{
	if(n==0)
	return 0;
	else
	return (n%2)+10*(dec(n/2));

}
```

ASSIGN 3

//stack ADT

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node_info
{
	void* data;
	struct node_info* next;
}node;
typedef struct
{
	int count;
	node* top;
}stack;
stack* create_stack()
{
	stack* sp;
	sp=(stack*)malloc(sizeof(stack));
	if(sp)
```

```
	{
		sp->top=NULL;
		sp->count=0;
	}
	return sp;
}
int push_stack(stack*ps,void*pele)
{
	node* temp;
	temp=(node*)malloc(sizeof(node));
	if(temp)
	{
		temp->data=pele;
		temp->next=ps->top;
		ps->top=temp;
		(ps->count)++;
		return 1;
	}
	else
	return 0;
}
void* pop_stack(stack*ps)
{
	node* temp;
	void* dout;
	if(ps->count==0)
	{
		return NULL;
	}
	else
	{
```

```
                    temp=ps->top;
                    dout=temp->data;
                    ps->top=temp->next;
                    (ps->count)--;
      free(temp);
      return dout;
          }
}
void* stack_top(stack*ps)
{
          void*dout;
          if(ps->count==0)
          return NULL;
          else
          {
                    dout=ps->top->data;
                    return dout;
   }
}
int stack_count(stack*ps)
{
          return(ps->count);
}
int stack_empty(stack*ps)
{
          if(ps->count==0)
          return 1;
          else
          return 0;
}
int stack_full(stack*ps)
```

```
{
	node* temp;
	temp=(node*)malloc(sizeof(node));
	if(!temp)
	return 1;
	else
	free(temp);
	return 0;
}

stack* destroy_stack(stack*ps)
{
	node*temp;
	if(ps)
	{
		while(ps->top)
		{
			temp=ps->top;
			ps->top=temp->next;
			free(temp->data);
			free(temp);
		}
		free(ps);
	}
	return NULL;
}
```

//assignment 3

//stack ADT integer implementation

```
#include"stackadt.c"
```

```
void display_stack(stack* ps)
{
	node*temp;
	int*pele;
	temp=(node*)malloc(sizeof(node));
	temp=ps->top;
	printf("stack elements are\n");
	while(temp)
	{
		pele=(int*)temp->data;
		printf("%d\n",*pele);
		temp=temp->next;
	}
}
int main()
{
	int choice,*pele;
	stack* sp;
	sp=create_stack();
	while(1)
	{
		printf("enter choice 1-push\t2-pop\t3-top\t4-count\t5-empty\t6-fullstack\t7-
display\t8->destroy\n");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:printf("enter element to be pushed\n");
				pele=(int*)malloc(sizeof(int));
				scanf("%d",pele);
				if(push_stack(sp,pele))
				{
```

```
                                printf("element %d is pushed
successfully\n",*pele);
                        }
                        else
                        {
                                printf("element %d is not pushed
successfully\n",*pele);
                        }
                        break;

        case 2:pele=(int*)pop_stack(sp);
            if(pele)
            printf("popped element is %d\n",*pele);
            else
            printf("stack is empty\n");
            break;

        case 3:pele=(int*)stack_top(sp);
            if(pele)
            printf("top element is %d\n",*pele);
            else
            printf("stack is empty\n");
            break;

        case 4:printf("number of elements in stack is %d\n",stack_count(sp));
            break;

        case 5:if(stack_empty(sp))
            printf("stack is empty\n");
            else
            printf("stack is not empty\n");
```

```
                    break;

                case 6:if(stack_full(sp))
                    printf("stack is full/n");
                    else
                    printf("stack is not full\n");
                    break;

                case 7:display_stack(sp);
                    break;

                case 8:if(destroy_stack(sp)==NULL)
                    printf("stack is destroyed\n");
                    break;
                default: return 0;
                }
        }
}
```

ASSIGN 4

//stack array ADT

```
#include<stdio.h>
#include<stdlib.h>
typedef struct
{
        int count,maxsize,top;
        void** starr;
}stack;
stack* create_stack(int size)
{
```

```
	stack* sp;
	sp=(stack*)malloc(sizeof(stack));
	if(!sp)
	{
		return NULL;
	}
	else
	{
		sp->count=0;
		sp->maxsize=size;
		sp->top=-1;
		sp->starr=(void**)calloc(size,sizeof(void*));
		if(!(sp->starr))
		{
			free(sp);
			return NULL;
		}
		return sp;
	}
}
int push_stack(stack* sp,void* pdata)
{
	if(sp->count==sp->maxsize)
	return 0;
	(sp->top)++;
	sp->starr[sp->top]=pdata;
	(sp->count)++;
	return 1;
}
void* pop_stack(stack* sp)
{
```

```
	void* dptr;
	if(sp->count==0)
	return NULL;
	else
	{
		dptr=(sp->starr[sp->top]);
		(sp->top)--;
		(sp->count)--;
		return dptr;
	}
}
void* stack_top(stack* sp)
{
	void* dptr;
	if(sp->count==0)
	return NULL;
	else

	{
		dptr=(sp->starr[sp->top]);
		return dptr;
	}
}
int stack_count(stack* sp)
{
	return(sp->count);
}
int stack_empty(stack*sp)
{
	if(!(sp->count))
	return 1;
```

```
	else
	return 0;
}
int stack_full(stack*sp)
{
	if(sp->count==sp->maxsize)
	return 1;
	else
	return 0;
}
stack* destroy_stack(stack* sp)
{
	if(sp)
	{
		for(int i=(sp->top);i>=0;i--)
		{
			free(sp->starr[i]);
		}
			free(sp->starr);
			free(sp);
	}
	return NULL;
}
```

//stack array implementation of integers

```
#include"stackarradt.c"
void display_stack(stack*sp)
{
	if(sp)
	{
```

```
                int* dout;
                printf("stack elements are\n");
                for(int i=(sp->top);i>=0;i--)
                {
                  dout=(int*)(sp->starr[i]);
                        printf("%d\n",*dout);
                }
        }
        else
        printf("stack is empty\n");
}
int main()
{
        stack*sp;
        int size,*pele,choice;
        printf("enter number of  elements\n");
        scanf("%d",&size);
        sp=create_stack(size);
        while(1)
        {
                printf("enter choice 1-push\t 2-pop\t 3-top\t 4-count\t 5-empty\t 6-full\t 7-
display\t 8-destroy\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:printf("enter element to be pushed\n");
                            pele=(int*)malloc(sizeof(int));
                            scanf("%d",pele);
                            if(push_stack(sp,pele))
                            printf("%d is pushed successfully\n",*pele);
                            else
```

```
            printf("%d is not pushed successfully\n",*pele);
            break;
case 2:pele=(int*)pop_stack(sp);
      if(pele)
      printf("popped element is %d\n",*pele);
      else
      printf("stack is empty\n");
      break;
case 3:pele=(int*)stack_top(sp);
      if(pele)
      printf("top element is %d\n",*pele);
      else
      printf("stack is empty\n");
      break;
 case 4:printf("number of elements are: %d\n",stack_count(sp));
       break;
 case 5:if(stack_empty(sp))
        printf("stack is empty\n");
        else
        printf("stack is not empty\n");
        break;
  case 6:if(stack_full(sp))
         printf("stack is full\n");
         else
         printf("stack is not full\n");
         break;
  case 7:display_stack(sp);
         break;
  case 8: destroy_stack(sp);
        break;
  default:return 0;
```

```
                }

            }
}
```

ASSIGN 5

//queue ADT

```
#include<stdio.h>
#include<stdlib.h>
typedef struct qnode_info
{
        void* dptr;
        struct qnode_info*next;
}qnode;
typedef struct
{
        int count;
        qnode *front,*rear;
}queue;
queue* create_queue()
{
        queue* qp;
        qp=(queue*)malloc(sizeof(queue));
        if(qp)
        {
                qp->front=NULL;
                qp->rear=NULL;
                qp->count=0;
        }
        return qp;
```

```
}
int  en_queue(queue* qp,void* dp)
{
        qnode*temp;
        temp=(qnode*)malloc(sizeof(qnode));
        if(!temp)
        return 0;
        if(temp)
        {
                temp->dptr=dp;
                temp->next=NULL;
                if(qp->count==0)
                {
                        qp->front=temp;
                }
                else
                {
                        qp->rear->next=temp;
                }
                   qp->rear=temp;
                   (qp->count)++;

         }
         return 1;
}
int de_queue(queue*qp,void**dp)
{
        qnode*temp;
        if(qp->count==0)
        return 0;
        temp=qp->front;
```

```
		*dp=temp->dptr;
		qp->front=temp->next;
		if(qp->count==1)
		{
			qp->rear=NULL;
		}
		free(temp);
		(qp->count)--;
		return 1;
}
void* front_queue(queue*qp)
{
		if(!(qp->count))
		return NULL;
		else
		return(qp->front->dptr);
}
void* rare_queue(queue*qp)
{
		if(!(qp->count))
		return  NULL;
		else
		return(qp->rear->dptr);
}
int qcount(queue*qp)
{
		return(qp->count);
}
int qempty(queue*qp)
{
		if(qp->count==0)
```

```
        return 1;
        return 0;

}
int qfull(queue*qp)
{
        qnode* temp;
        temp=(qnode*)malloc(sizeof(qnode));
        if(!temp)
        return 1;
        else
        {
                free(temp);
           return 0;
        }
}

queue* destroy_queue(queue*qp)
{
        qnode *temp,*deleteptr;
        if(qp)
        {
                temp=qp->front;
                while(temp)
                {
                        deleteptr=temp;
                        temp=temp->next;
                        free(deleteptr->dptr);
                        free(deleteptr);
                }
                free(qp);
```

```
	}
	return NULL;
}
```

//q_ ADT implementation

```
#include"queueadt.c"
void display_queue(queue*qp)
{
	qnode* temp;
	int* pele;
	if(!qp->count)
	printf("queue is empty\n");
	else
	{
		temp=qp->front;
	   while(temp)
	  {
		   pele=(int*)temp->dptr;
	      printf("%d\n",*pele);
	      temp=temp->next;
		 }
	}
}
int main()
{
	int choice,*pele;
	queue* qp;
	qp=create_queue();
```

```
while(1)
{
        printf("enter choice 1-enqueue\t2-dequeue\t 3-count\t 4-frontq\t 5-rareq\t
6-qempty\t 7-fullq\t 8-display\t 9-destroy\n");
        scanf("%d",&choice);
        switch(choice)
        {
                case 1:printf("enter element to be insert\n");
                        pele=(int*)malloc(sizeof(int));
                        scanf("%d",pele);
                        if(en_queue(qp,pele))
        printf("%d is inserted\n",*pele);
        else
        printf("%d is not inserted\n",*pele);
        break;
case 2:if(de_queue(qp,(void**)&pele))
        { pele=(int*)pele;
        printf("%d is deleted\n",*pele);
                                }
        else
        printf("queue is empty\n");
        break;
case 3:printf("number of elements are %d\n",qcount(qp));
        break;
case 4:if(front_queue(qp))
        { pele=(int*)front_queue(qp);
         printf("front element is %d\n",*pele);
                                }
         else
        { printf("queue is empty\n");}
         break;
```

```
}

        case 5:if(rare_queue(qp))
                { pele=(int*)rare_queue(qp);
                 printf("rare element is %d\n",*pele);
                                                    }
                 else
                { printf("queue is empty\n");}
                 break;

        case 6:if(qempty(qp))
                 printf("queue is empty\n");
                 else
                 printf("queue is not empty\n");
                 break;
        case 7:if(qfull(qp))
                 printf("queue is full\n");
                 else
                 printf("queue is not full\n");
                 break;
        case 8:display_queue(qp);
                break;
        case 9:if(!destroy_queue(qp))
                 printf("queue is destroyed\n");
                 else
                 printf("queue not destroyed\n");
                 break;
        default : return 0;
                  }
        }
}
```

ASSIGN 6

//queue array ADT

```
#include<stdio.h>
#include<stdlib.h>
typedef struct
{
	int front,count,rare;
	int maxsize;
	void** qarray;
}queue;
queue* create_queue(int size)
{
	queue* qp;
	qp=(queue*)malloc(sizeof(queue));
	if(!qp)
	return NULL;
	else
	{
		qp->front=-1;
		qp->rare=-1;
		qp->count=0;
		qp->maxsize=size;
	}
	qp->qarray=(void**)calloc(size,sizeof(void*));
	if(!qp->qarray)
	{
		free(qp);
		return NULL;
	}
	else
```

```
  {
        return qp;
  }
}
int en_queue(queue*qp,void*dp)
{
      if(qp->count==qp->maxsize)
      return 0;
      else
      {
            (qp->rare)++;
            if(qp->rare==qp->maxsize)
            {
                  qp->rare=0;
            }
            qp->qarray[qp->rare]=dp;
            if(!qp->count)
            qp->front=0;
            (qp->count)++;
            return 1;
      }
}
void* de_queue(queue*qp)
{
      void* dp;
      if(!qp->count)
      return NULL;
      else
      {
            dp=qp->qarray[qp->front];
            (qp->front)++;
```

```
                if((qp->front)==(qp->maxsize))
            {
                  qp->front=0;
                }
           if(qp->count==1)
          {
                qp->front=-1;
                qp->rare=-1;
          }
          (qp->count)--;
  }
  return dp;
}
int qcount(queue*qp)
{
        return(qp->count);
}
void* qfront(queue*qp)
{
        if(!qp->count)
        return NULL;
        else
        return(qp->qarray[qp->front]);
}
void* qrare(queue*qp)
{
        if(!qp->count)
        return NULL;
        else
        return(qp->qarray[qp->rare]);
}
```

```
int qfull(queue*qp)
{
        if(qp->count==qp->maxsize)
        return 1;
        else
        return 0;
}
int qempty(queue*qp)
{
        if(!(qp->count))
        return 1;
        else
        return 0;
}
queue* qdestroy(queue*qp)
{
        int i;
        if(qp)
        {
                if(qp->count>0)
                {
                        i=qp->front;
                        while(i!=qp->rare)
                        {
                                free(qp->qarray[i]);
          i=i+1;
          if(i==qp->maxsize)
          i=0;
                        }
                        free(qp->qarray[qp->rare]);
                }
```

```
                free(qp->qarray);
                free(qp);
        }
        return NULL;
}
```

//q_array ADT implementation

```
#include"qarrayadt.c"
void qdisplay(queue*qp)
{
        int i;
        if(qp)
        {
                if(qp->count>0)
                {
                        i=qp->front;
                        while(i!=qp->rare)
                        {
                                printf("%d\n",*((int*)qp->qarray[i]));
                                i=i+1;
                                if(i==qp->maxsize)
                                i=0;
                        }
                        printf("%d\n",*(int*)qp->qarray[i]);
                }
                else
                {
                        printf("queue is empty\n");
                }
  }
```

```
}
int main()
{
	queue* qp;
	int size,*pele,choice;
	printf("enter number of  elements\n");
	scanf("%d",&size);
	qp=create_queue(size);
	while(1)
	{
		printf("enter choice 1-enqueue\t 2-dequeue\t 3-qcount\t 4-qfront\t 5-qrare\t
6-qfull\t 7-qempty\t8-qdisplay\t 9-qdestroy\n");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1:printf("enter element to be pushed\n");
			       pele=(int*)malloc(sizeof(int));
			       scanf("%d",pele);
			       if(en_queue(qp,pele))
			       printf("%d is inserted successfully\n",*pele);
			       else
			       printf("%d is not inserted \n",*pele);
			       break;
		   case 2:pele=(int*)de_queue(qp);
			   if(pele)
			   printf("popped element is %d\n",*pele);
			   else
			   printf("queue is empty\n");
			   break;
		    case 3:printf("number of elements in queue are %d\n",qcount(qp));
			   break;
```

```
        case 4:if(qfront(qp))
    { pele=(int*)qfront(qp);
     printf("front element is %d\n",*pele);
                                        }
     else
    { printf("queue is empty\n");
                                                 }
     break;
        case 5:if(qrare(qp))
    { pele=(int*)qrare(qp);
     printf("rare element is %d\n",*pele);
                                        }
     else
    { printf("queue is empty\n");
                                                 }
     break;
case 6:if(qfull(qp))
     printf("queue is full\n");
     else
     printf("queue is not full\n");
     break;
case 7:if(qempty(qp))
      printf("queue is empty\n");
      else
      printf("queue is not empty\n");
      break;
case 8:qdisplay(qp);
     break;
case 9:qdestroy(qp);
     break;
       default: return 0;
```

```
                break;
            }
        }
}
```

ASSIGN 7

//Linked lists

```
#include<stdio.h>
#include<stdlib.h>
typedef int(*comparedata)(void*,void*);
typedef struct node_info
{
        void* data;
        struct node_info* next;
}node;
typedef struct
{
        int count;
        node *head,*rare,*pos;
        comparedata comp;
}list;
int add_node(list*lp,void*pdata);
int search(list* lp,node**prev,node**curr,void*key);
int insert(list* lp,node*prev,void*data);
int remove_node(list* lp,void*key,void**dout);
int search_list(list*lp,void*key,void**dout);
void delete(list*lp,node*prev,node*curr,void**dout);
//create
list* create_list(comparedata cdata)
{
```

```
        list* lp;
        lp=(list*)malloc(sizeof(list));
        if(lp)
        {
                lp->count=0;
                lp->pos=NULL;
                lp->head=NULL;
                lp->rare=NULL;
                lp->comp=cdata;
        }
        return lp;
}
//insert
int add_node(list* lp,void* pdata)
{
        int f,i;
        node *prev,*curr;
        f=search(lp,&prev,&curr,pdata);
        if(f)
        return 0;
        i=insert(lp,prev,pdata);
        if(!i)
        return -1;
        else
        return 1;
}
int insert(list*lp,node*prev,void*pdata)
{
        node* temp;
        temp=(node*)malloc(sizeof(node));
   if(!temp)
```

```
   return 0;
   else
   {
               temp->data=pdata;
               temp->next=NULL;
               if(!prev)
               {
                      temp->next=lp->head;
                      lp->head=temp;
                      if(!lp->count)
                      lp->rare=temp;
               }
               else
               {
                      temp->next=prev->next;
                      prev->next=temp;
                      if(!temp->next)
                      {
                             lp->rare=temp;
                      }
               }
               (lp->count)++;
               return 1;
        }
}
//delete
int remove_node(list* lp,void* key,void**dout)
{
        int f;
        node *prev,*curr;
        f=search(lp,&prev,&curr,key);
```

```
        if(f)
    delete(lp,prev,curr,dout);
    return f;
}
void delete(list* lp,node* prev,node* curr,void** dout)
{
        *dout=curr->data;
        if(!prev)
        {
                lp->head=curr->next;
        }
        else
        {
                prev->next=curr->next;
            if(prev->next==NULL)
        {
                lp->rare=prev;
        }
    }
        free(curr);
        (lp->count)--;
}
//search
int search_list(list* lp,void*key,void**dout)
{
        int f;
        node *prev,*curr;
        f=search(lp,&prev,&curr,key);
        if(f)
        {
                *dout=curr->data;
```

```
	}
	else
	{
		*dout=NULL;
	}
	return f;
}
int  search(list*lp,node** prev,node** curr,void* key)
{
	int result;
	*prev=NULL;
	*curr=lp->head;
	if(lp->count==0)
	{
		return 0;
	}
	result=lp->comp(key,lp->rare->data);
	if(result==1)
	{
		*prev=lp->rare;
		*curr=NULL;
		return 0;
	}
	while((result=(lp->comp(key,(*curr)->data)))>0)
	{
		*prev=*curr;
		*curr=(*curr)->next;
	}
	if(result==0)
	return 1;
	else
```

```
	return 0;
}
//retrieve node
int retrieve_node(list*lp,void*pkey,void**dout)
{
	int f;
	node *prev,*curr;
	f=search(lp,&prev,&curr,pkey);
	if(!f)
	{
		*dout=NULL;
		return 0;
	}
	else
	{
		*dout=curr->data;
		return 1;
	}
}
//empty
int empty_list(list*lp)
{
	if(lp->count==0)
	return 1;
	else
	return 0;
}//full
int full_list(list*lp)
{
	node* temp;
	temp=(node*)malloc(sizeof(node));
```

```
	if(temp)
	{
		free(temp);
	   return 0;
	}
	else
	return 1;
}
//count
int list_count(list*lp)
{
	return(lp->count);
}
//traverse
int  traverse(list* lp,int fw,void**dout)
{
	if(empty_list(lp)==1)
	{
		return 0;
	}
	if(fw==0)
	{
		lp->pos=lp->head;
		lp->pos->data=*dout;
		return 1;
	}
	else
	{
		if(lp->pos->next==NULL)
		{
			return 0;
```

```
		}
		lp->pos=lp->pos->next;
		*dout=lp->pos->data;
		return 1;
	}
}
//desroy list
list* destroy_list(list*lp)
{
	node *temp;
	if(lp)
	{
		temp=lp->head;
		while(temp)
		{
			temp=temp->next;
			free(temp->data);
			free(temp);
			temp=lp->head;
		}
		free(lp);
	}
		return NULL;
}
```

//Implementation OF LINKLIST

#include"linkedlist.h"

```
int comp(void*pd1,void*pd2)
{
```

```
	int*p1=(int*)pd1;
	int*p2=(int*)pd2;
	if(*p1>*p2)
	return 1;
	else if(*p1==*p2)
	return 0;
	else
	return -1;
}
void display_list(list*lp)
{
	node* temp;
	temp=lp->head;
	while(temp!=NULL)
	{
		printf("%d\n",*((int*)temp->data));
		temp=temp->next;
	}
}
int main()
{
	list*l;
	int *a;
	int ch,f,tf;
	l=create_list(&comp);
	while(1)
	{
		printf("enter choice: 1-addnode\t 2-remove\t 3-search\t 4-retrieve\t 5-
empty\t 6-fulllist\t 7-count\t 8-traverse\t 9-display\t10-destroy\n");
		scanf("%d",&ch);
		switch(ch)
```

```
{
        case 1:a=(int*)malloc(sizeof(int));
            printf("enter element to be added\n");
            scanf("%d",a);
            f=add_node(l,a);
            if(f==-1)
            printf("overflow\n");
            else if(f==1)
            printf("%d is inserted\n",*a);
            else
            printf("%d is duplicate\n",*a);
            break;
        case 2:printf("enter element to be removed\n");
            int *d;
            a=(int*)malloc(sizeof(int));
            scanf("%d",a);
            f=remove_node(l,a,(void**)&d);
            if(f)
            printf("%d is deleted\n",*d);
            else
            printf("%d is not exists\n",*a);
            break;
        case 3:printf("enter element to be searched\n");
            a=(int*)malloc(sizeof(int));
            scanf("%d",a);
            f=search_list(l,a,(void**)&d);
            if(f==1)
            {
                        printf("%d exists\n",*d);
                    }
                    else
```

```
                                                  printf("%d not exists\n",*a);
                                                  break;
                    case 4:printf("enter key element\n");
                           a=(int*)malloc(sizeof(int));
                           scanf("%d",a);
                           f=retrieve_node(l,a,(void**)&d);
                           if(f==1)
                           {
                                                          printf("address of retrieve node having key %d
is:%p\n",*d,d);
                                                  }
                                                  else
                                                  {
                                                          printf("%d is not exist\n",*a);
                                                  }
                                                  break;
                    case 5:f=empty_list(l);
                          if(f==1)
                          printf("list is empty\n");
                          else
                          printf("list is not empty\n");
                          break;
                   case 6:if(full_list(l))
                          printf("list is full\n");
                          else
                          printf("list is not full\n");
                          break;
                    case 7:printf("number of elements %d\n",list_count(l));
                          break;
                    case 8:tf=0;
                           f=traverse(l,tf,(void**)&d);
```

```
                    tf=1;
                    break;
                case 9:printf("list contents are\n");
                    display_list(l);
                    break;
                case 10:destroy_list(l);
                     break;
                default:return 0;
            }
        }
}
```

ASSIGN 8

//ORDINARY binary tree

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node_info
{
   int data;
   struct node_info *l,*r;
}node;
void createtree(node**pr);
int insert_node(node**proot,int n,char*p);
node* getnode(int n);
void preorder(node*root);
void postorder(node*root);
void inorder(node*root);
int search_node(node*,int);
int ele_occ_count(node*,int);
int height_count(node*root);
```

```
node*copy_tree(node*root);
int sum_node(node*root);
int node_count(node*root);
int leaf_count(node*root);
int isBalanced(node* root);
int inter_count(node*root);
node * getParent(node *root, int key);
int main()
{
	node*root=NULL;
	node*copy;
	int ch=1,key;
	while(ch)
	{
		printf("Enter your choice\n1.Create tree\n2.Display tree\n3.Search
element\n4.Occurance count\n5.Height of tree\n6.Copy tree\n7.Node sum\n8.Node
count\n9.Leaf count\n10.Balanced\n11.Intermediate node count\n12.Parent of key\n");
		scanf("%d",&ch);
		switch (ch)
		{
			case 1: createtree(&root);
			break;
			case 2: inorder(root);
					printf("\n");
			break;
			case 3: printf("Enter the key element to be searched\n");
					scanf("%d",&key);
					if(search_node(root,key)) printf("Key element %d
exist\n",key);
					else printf("Key element does not exist\n");
			break;
			case 4: printf("Enter the key element\n");
```

```
                        scanf("%d",&key);
                        printf("The occurance count of the key element
is:%d\n",ele_occ_count(root,key));
                break;
                case 5: printf("The height of the tree is:%d\n",height_count(root));
                break;
                case 6: copy=copy_tree(root);
                        printf("The copied tree is\n");
                        inorder(copy);
                        printf("\n");
                break;
                case 7: printf("The sum of nodes of tree is:%d\n",sum_node(root));
                break;
                case 8: printf("The node count is:%d\n",node_count(root));
                break;
                case 9: printf("The leaf count of tree is:%d\n",leaf_count(root));
                break;
                case 10: if(isBalanced(root))
                        printf("Tree is balanced\n");
                        else
                        printf("Tree is not balanced\n");
                break;
                case 11: printf("The intermediate node count
is:%d\n",inter_count(root));
                break;
                case 12: printf("Enter key element\n");
                        scanf("%d",&key);
                        if(getParent(root,key)==NULL) printf("Element not
found\n");
                        else {
                        int *x=(int*)getParent(root,key);
                        printf("The parent of the key is:%d\n",*x);
```

```
                                    }
                        break;
                        default: printf("Enter valid choice\n");
                        return 0;
                }
        }
}

void createtree(node**pr)
{
  int n;char pos[30];
  int insert_node(node**,int,char*);
  printf("Enter root element\n");
  scanf("%d",&n);
  *pr=getnode(n);
  printf("Enter the tree elements\n");
  while(scanf("%d",&n)!=EOF)
  {
    printf("Enter the position string :");
    scanf("%s",pos);
    if(!insert_node(pr,n,pos))
                        printf("Invalid position string or node already exists\n");
  }
}
int insert_node(node**proot,int n,char*p)
{
  node *temp,*t1=*proot,*t2=NULL;
  int i;
  temp=getnode(n);
  for(i=0;*(p+i)!='\0';i++)
  {
```

```
        if(t1==NULL)
            break;
        t2=t1;
        if(*(p+i)=='l')
            t1=t1->l;
        else
            t1=t1->r;
    }
    if(*(p+i)=='\0'&&t1==NULL)
    {
        if(p[i-1]=='l')
            t2->l=temp;
        else
            t2->r=temp;
    }
    else
        return 0;
    return 1;
}
node* getnode(int n)
{
    node*temp;
    temp=(node*)malloc(sizeof(node));
    if(temp)
    {
        temp->data=n;
        temp->l=NULL;
        temp->r=NULL;
    }
    return temp;
}
```

```
void preorder(node*root)
{
        if(root!=NULL){
                printf("%d\t",root->data);
   preorder(root->l);
   preorder(root->r);
        }
}
void postorder(node*root)
{
        if(root!=NULL){
   postorder(root->l);
   postorder(root->r);
   printf("%d\t",root->data);
        }
}
void inorder(node*root)
{
        if(root!=NULL){
   inorder(root->l);
   printf("%d\t",root->data);
   inorder(root->r);
        }
}
int search_node(node*root,int key){
        if(root==NULL)
                return 0;
        else if(root->data==key)
                return 1;
        return (search_node(root->l,key)||search_node(root->r,key));
}
```

```
int ele_occ_count(node*root,int key)
{
	if(!root) return 0;
	if(root->data==key)return(1+ele_occ_count(root->l,key)+ele_occ_count(root->r,key));
	return(0+ele_occ_count(root->l,key)+ele_occ_count(root->r,key));
}
int height_count(node*root)
{
	if(!root) return 0;
	int lh=height_count(root->l);
	int rh=height_count(root->r);
	if(lh>rh) return (lh+1);
	return(rh+1);
}
node*copy_tree(node*root)
{
	node*temp;
	if(!root) return NULL;
	temp=(node*)malloc(sizeof(node));
	temp->data=root->data;
	temp->l=copy_tree(root->l);
	temp->r=copy_tree(root->r);
	return temp;
}
int sum_node(node*root)
{
	if(!root)return 0;
	return(root->data+sum_node(root->l)+sum_node(root->r));
}
int node_count(node*root)
{
```

```
		if(!root)return 0;
		return(1+node_count(root->l)+node_count(root->r));
}
int leaf_count(node*root)
{
		if(!root)return 0;
		if(!root->l && !root->r) return 1;
		return(0+leaf_count(root->l)+leaf_count(root->r));
}
int isBalanced(node* root)
 {
   int lh, rh;
   if(!root) return 1;
   lh = height_count(root->l);
   rh = height_count(root->r);
   if (abs(lh - rh) <= 1 && isBalanced(root->l) && isBalanced(root->r))
      return 1;
   return 0;
}
int inter_count(node*root)
{
		if(!root || (!root->l && !root->r)) return 0;
		return(1+inter_count(root->l)+inter_count(root->r));
}
//Parent key for bt
node * getParent(node *root, int key)
{
   if (root == NULL) return NULL;
   if ((root->l && root->l->data == key) || (root->r && root->r->data == key)) return root;
   node *left = getParent(root->l, key);
   if (left != NULL) return left;
```

```
   node *right = getParent(root->r, key);
   return right;
}
```

ASSIGN 9

//Binary Search Tree

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node_info
{
        int data;
        struct node_info *l,*r;
}node;
void insertnode(node**proot,int e);
node*getnode(int n);
void createbst(node** proot)
{
        int ele;
        void insertnode(node**,int);
        printf("Enter root element\n");
        scanf("%d",&ele);
        *proot=getnode(ele);
        printf("Enter the tree elements\n");


        while(scanf("%d",&ele)!=EOF)
                insertnode(proot,ele);
        return ;
}
```

```
void insertnode(node**proot,int e)
{
	node *t1=*proot,*t2=NULL;
	node*temp;
	temp=getnode(e);
	while(t1)
	{
		t2=t1;
		if(t1->data<=temp->data)
			t1=t1->r;
		else
			t1=t1->l;
	}
	if(temp->data<t2->data)
		t2->l=temp;
	else
		t2->r=temp;
}
node*getnode(int n)
{
	node*temp;
	temp=(node*)malloc(sizeof(node));
	temp->data=n;
	temp->l=NULL;
	temp->r=NULL;
	return temp;
}
void inorder(node*proot)//ascending
{
	if(proot)
	{
```

```
		inorder(proot->l);
		printf("%d\t",proot->data);
		inorder(proot->r);
	}
}
void inorder2(node*proot)//descending
{
	if(proot)
	{
		inorder2(proot->r);
		printf("%d\t",proot->data);
		inorder2(proot->l);
	}
}
void preorder(node*proot)
{
	if(proot)
	{
		printf("%d\t",proot->data);
		preorder(proot->l);
		preorder(proot->r);
	}
}
void postorder(node*proot)
{
	if(proot)
	{
		postorder(proot->l);
		postorder(proot->r);
		printf("%d\t",proot->data);
	}
```

```
}
int search_node(node*root,int key)
{
	if(!root) return 0;
	if(root->data==key) return 1;
	return(search_node(root->l,key) || search_node(root->r,key));
}
int ele_occ_count(node*root,int key)
{
	if(!root)return 0;
	if(root->data==key)
	return(1+ele_occ_count(root->l,key)+ele_occ_count(root->r,key));
	return(0+ele_occ_count(root->l,key)+ele_occ_count(root->r,key));
}
int height_count(node*root)
{
	if(!root)return 0;
	int lh=height_count(root->l);
	int rh=height_count(root->r);
	if(lh>rh)return(lh+1);
	else return (rh+1);
}
node* copy_tree(node*root)
{
	node*temp;
	if(!root)return NULL;
	temp=(node*)malloc(sizeof(node));
	temp->data=root->data;
	temp->l=copy_tree(root->l);
	temp->r=copy_tree(root->r);
	return temp;
```

```
}
int sum_nodes(node*root)
{
	if(!root)return 0;
	return(root->data+sum_nodes(root->l)+sum_nodes(root->r));
}
int node_count(node*root)
{
	if(!root) return 0;
	return(1+node_count(root->l)+node_count(root->r));
}
int leaf_count(node*root)
{
	if(!root)return 0;
	if(!root->l && !root->r) return 1;
	return(0+leaf_count(root->l)+leaf_count(root->r));
}
int balanced(node*root)
{
	if(!root)return 1;
	int lh=height_count(root->l);
	int rh=height_count(root->r);
	if(abs(lh-rh)<=1 && balanced(root->r) && balanced(root->r))
	return 1;
	return 0;
}
int inter_count(node*root)
{
	if(!root || (!root->l && !root->r))return 0;
	return(1+inter_count(root->l)+inter_count(root->r));
}
```

```
//parent key for bst
node * getParent(node *root, int key)
{
   if (root == NULL) return NULL;
   else if ((root->r && root->r->data == key) || (root->l && root->l->data == key))
    return root;
   else if (root->data > key)
    return (getParent(root->l, key));
   return (getParent(root->r, key));
   return root;
}

int main()
{
	node*root=NULL;
	node*copy;
	int ch=1,key;
	while(ch)
	{
		printf("Enter your choice\n1.Create tree\n2.Display tree\n3.Search element\n4.Occurance count\n5.Height of tree\n6.Copy tree\n7.Node sum\n8.Node count\n9.Leaf count\n10.Balanced\n11.Intermediate node count\n12.Parent of key\n");
		scanf("%d",&ch);
		switch (ch)
		{
			case 1: createbst(&root);
			break;
			case 2: printf("Elements in ascending order\n");
					inorder(root);
					printf("\n");
			   printf("Elements in descending order\n");
```

```
				inorder2(root);
				printf("\n");
			break;
			case 3: printf("Enter the key element to be searched\n");
					scanf("%d",&key);
					if(search_node(root,key)) printf("Key element %d 
exist\n",key);
					else printf("Key element does not exist\n");
			break;
			case 4: printf("Enter the key element\n");
					scanf("%d",&key);
					printf("The occurance count of the key element 
is:%d\n",ele_occ_count(root,key));
			break;
			case 5: printf("The height of the tree is:%d\n",height_count(root));
			break;
			case 6: copy=copy_tree(root);
					printf("The copied tree is\n");
					inorder(copy);
					printf("\n");
			break;
			case 7: printf("The sum of nodes of tree is:%d\n",sum_nodes(root));
			break;
			case 8: printf("The node count is:%d\n",node_count(root));
			break;
			case 9: printf("The leaf count of tree is:%d\n",leaf_count(root));
			break;
			case 10: if(balanced(root))
					printf("Tree is balanced\n");
					else
					printf("Tree is not balanced\n");
```

```
                        break;
                        case 11: printf("The intermediate node count 
is:%d\n",inter_count(root));
                        break;
                        case 12: printf("Enter the key element to be searched\n");
                                 scanf("%d",&key);
                                 int *x=(int*)getParent(root,key);
                                 printf("Parent element of %d is %d\n",key,*x);
                        break;
                        default: printf("Enter valid choice\n");
                        return 0;
                }
        }
        return 0;
}
```