1 Listes chaînées

```
boolean est_vide(LISTE p)
{ return (p==NULL);}

boolean enlevepremier(LISTE *l,int *x){
  LISTE temp;
  if (*l==NULL)
    return (false);
  else {
    temp=*l;
    *x=(*l)->valeur;
    *l=(*l)->suivant;
    free(temp);
    return (true);
  }}

void inserepremier(LISTE *lp, int x)
{ LISTE tmp;
  tmp=(LISTE) malloc(sizeof(LISTE));
  tmp->valeur=x;
  tmp->suivant=*lp;
  *lp=tmp;
}

void afficheRec(LISTE l)
{
  if (est_vide(l))
    printf("\n");
  else
    {
    printf("%d  ",l->valeur);
    afficheRec(l->suivant);
    }}

void afficheIt(LISTE l){
  while (!est_vide(l)){
    printf("%d ",l->valeur);
    l=l->suivant;
    }
  printf("\n");
}

int longueur(LISTE l){
  if (l==NULL){
    return(0);
  }
  else {
    return(1+longueur(l->suivant));
      }}

void concatene(LISTE *l1, LISTE l2){
  LISTE tmp;
  if (*l1==NULL) {
    *l1=l2;
       }
  else {
    tmp=*l1;
    while (tmp->suivant != NULL){
      tmp=tmp->suivant;
    }
    tmp->suivant = l2;
  }}

void miroirIt(LISTE * l){
  LISTE reste, miroir, tmp;
  reste=*l;
  miroir=NULL;
  while (reste!=NULL){
      tmp=reste;
      reste=reste->suivant;
      tmp->suivant=miroir;
      miroir=tmp;
  }
  *l=miroir;
}

void miroirRec(LISTE *l){
  LISTE finp;
  void local(LISTE *l, LISTE *queuep){
    if ((*l)->suivant != NULL) {
      local(&((*l)->suivant),queuep);
      (*queuep)->suivant=*l;
      *l=(*l)->suivant;
      *queuep=(*queuep)->suivant;
      (*queuep)->suivant=NULL;
    }
  }

  if (*l != NULL) {
    queueIteratif(*l, &finp);
    local(l,&finp);
  }}
```

2 Polynômes

```
/** degre = -1 pour polynome nul**/
int degre(POLY_PTR  p){
  int degresofar=-1, position=0;
  while (p !=NULL) {
     if (p->coeff != 0) {
       degresofar = position;
      }
      p=p->suivant;
      position++;
    }
  return(degresofar);
}

int degreRec(POLY_PTR  p){
```

```
  int d;
  if (p==NULL) {
    return(-1);
  }
  else {
    d=degreRec(p->suivant);
    if ((p->coeff==0)&&(d==-1))
        return(-1);
      else
        return(1+d);
  }}

int evalue ( POLY_PTR p, int n) {
  if (p==NULL) {
    return(0);
  }
  else {
   return(p->coeff+n*evalue(p->suivant,n));
  }}

void multiplieparX( POLY_PTR * p){
  POLY_PTR temp;
  temp=(POLY_PTR)malloc(sizeof(POLY));
  temp->coeff=0;
  temp->suivant=*p;
  *p=temp;
}

void multiplieparc( POLY_PTR p, int c){
  while (p!=NULL){
   p->coeff=c*(p->coeff);
   p=p->suivant;
  }}

POLY_PTR copie(POLY_PTR p){
  POLY_PTR temp;
  if (p==NULL){
    return(NULL);
  }
  else {
    temp=(POLY_PTR)malloc(sizeof(POLY));
    temp->coeff=p->coeff;
    temp->suivant=copie(p->suivant);
    return(temp);
 }}

POLY_PTR somme(POLY_PTR p, POLY_PTR q){
  POLY_PTR temp;
  if (p==NULL) {
    return ( copie(q));
  }
  else {
    if (q==NULL) {
      return (copie(p));
    }
    else {
      temp= (POLY_PTR)malloc(sizeof(POLY));
      temp->coeff = p->coeff + q->coeff;
      temp->suivant=somme(p->suivant,
                          q->suivant);
      return(temp);
    }}}

void libere (POLY_PTR * p){
  POLY_PTR temp;
  if (*p!=NULL){
    temp=*p;
    libere(&(*p)->suivant);
    free(temp);
  }}

POLY_PTR multiplie(POLY_PTR p,POLY_PTR q){
  POLY_PTR debut,reste,resultat;
  if (p==NULL){
    return(NULL);
  }
  else {
    debut=copie(q);
    multiplieparc( debut,  p->coeff);
    reste=multiplie(p->suivant,q);
    multiplieparX(&reste);
    resultat=somme(debut,reste);
    libere(&debut);
    libere(&reste);
    return(resultat);
  }}
```

3 Arbres binaires de recherche

```
struct tmp_abr
  {
  int valeur;
  struct tmp_abr *fg, *fd;
};
typedef struct tmp_abr rec_arbre;
typedef rec_arbre * arbre;

void parcoursPrefixe(arbre a){
  if (a!=NULL){
    printf("%d ",a->valeur);
    parcoursPrefixe(a->fg);
    parcoursPrefixe(a->fd);
  }}

void parcoursInfixe(arbre a){
  if (a!=NULL){
    parcoursInfixe(a->fg);
    printf("%d ",a->valeur);
```

```
    parcoursInfixe(a->fd);
  }}

void parcoursSuffixe(arbre a){
  if (a!=NULL){
    parcoursSuffixe(a->fg);
    parcoursSuffixe(a->fd);
    printf("%d ",a->valeur);
  }}

boolean recherche(arbre a, int x){
  if (a==NULL)
    return(false);
  else {
    if (a->valeur ==x)
      return(true);
    else
      if (a->valeur <x)
        return(recherche(a->fd,x));
      else
      return(recherche(a->fg,x));
  }}

void insere(arbre *a, int x){
  if (*a == NULL){
    *a=(arbre)malloc(sizeof(rec_arbre));
    (*a)->valeur=x;
    (*a)->fg=NULL;
    (*a)->fd=NULL;
  }
  else {
    if ((*a)->valeur >x)
      insere(&((*a)->fg),x);
    else
      insere(&((*a)->fd),x);
  }}

/** Premice : l'arbre est non vide **/
void supprimeracine(arbre *a){
  arbre tmp;
  arbre b=(*a)->fd;
  if (b==NULL) {
    tmp=*a;
    *a=(*a)->fg;
    free(tmp);
  }
  else
    if (b->fg==NULL){
      b->fg=(*a)->fg;
      free(*a);
      *a=b;
    }
    else {
      while (b->fg->fg!=NULL)
b=b->fg;
      tmp=b->fg;
      tmp->fg=(*a)->fg;
      b->fg=tmp->fd;
      tmp->fd=(*a)->fd;
      *a=tmp;
    }}

boolean supprime(arbre *a, int x){
  if (*a==NULL)
    return(false);
  else {
    if ((*a)->valeur ==x)
      supprimeracine(a);
    else
      if ((*a)->valeur <x)
        supprime(&((*a)->fd),x);
      else
        supprime(&((*a)->fg),x);
  }}
```

4 Corrigé devoir 1

```
int MaxShifted(int t[], int taille,
                           int *nbcomp_ptr){
  int g=0, d=taille-1, x;
  while (d>g)
    {
      x=(g+d+1)/2;
      (*nbcomp_ptr)++;
      if (t[g]<t[x])
              g=x;
      else
              d=x-1;
    }
  return (t[d]);
}

main()
{
  int t[N];
  int i,taille,valeur,resultat;
  int nbcomp=0;
  scanf("%d", &taille);
  for (i=0; i<taille; i++)
    {
    scanf("%d", &t[i]);
    }
  resultat=MaxShifted(t,taille,&nbcomp);
  printf("%d %d \n", resultat, nbcomp);
}
```

5 Corrigé devoir 2

```
boolean recherche1 (int x, LISTE *l,
```

```
                    int *position_ptr){
  LISTE courant, precedent;
  int fini = false, pos = 1;
  if (est_vide (*l))
    return(false);
  else {
    if ((*l)->valeur == x) {
      *position_ptr = pos;
      return(true);
    }
    else {
      precedent = *l;
      courant = (*l)->suivant;
      fini=false; succes=false;
      while (!fini) {
        if (courant->suivant == NULL)
          fini = true;
        else {
          if (courant->valeur == x) {
            succes = true; fini = true;
            *position_ptr = pos;
            precedent->suivant = courant->suivant;
            courant->suivant = *l;
            *l = courant;
          }
          else {
            precedent = courant;
            courant = courant->suivant;
            pos++;
          }}
      }
      return(succes);
    }}}

boolean recherche2 (int x, LISTE *l,
                    int *position_ptr){

  boolean locale(int x, LISTE *l,
                 int *position_ptr){
    boolean b;
    LISTE tmp;
    if (est_vide((*l)->suivant))
      return(false);
    else if ((*l)->suivant!=x){
           b=locale(x,&((*l)->suivant),position_ptr);
          (*position_ptr)++;
          return(b);
          }
         else {
          *position_ptr=2;
          tmp = (*l)->suivant;
          (*l)->suivant = (*l)->suivant->suivant;
          tmp->suivant = *l;
          *l = tmp;
          return(true);
         }
     }/*fin de la procedure locale*/

  if (est_vide (*l))
    return(false);
  else {
    if ((*l)->valeur == x) {
      *position_ptr = 1;
      return(true);
    }
    else return(locale(x,l,position_ptr));
}}
```