Structures de données récursives en C
Code

Denis Monasse

JI3-Agadir Mai 2011
Document à usage exclusif pendant les JI3

Chapitre 1

Pointeurs et allocation mémoire

1.1 Pointeurs, variables, adresses

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

void parDix(int *i){
    *i=(*i)*10;
}

main(){
    int i=1,*j,k;
    parDix(&i);
    printf("%d\n",i);
    j=&i;
    *j=(*j)*2;
    printf("%d\n",i);
}
```

1.2 Allocation mémoire

```
/* pointeur vers entier */
int *a = malloc(sizeof(*a));
int *b = malloc(sizeof(int));
/* tableau de 10 entiers */
int *c = malloc(10*sizeof(int));
/* matrice 10x7 d'entiers */
int **m = malloc(10*sizeof(int*));
for(i=0; i<10; i++) m[i]=malloc(7*sizeof(int));
```

1.3 Tableaux et pointeurs

```
main(){
    int a[10],*b;
    *a=2011;
    printf("%d\n",a[0]);
    *(a+2)=75005;
    printf("%d\n",a[2]);
    b=malloc(5*sizeof(int));
    b[3]=1515;
    printf("%d\n",b[3]);
}
```

Chapitre 2

Listes en C

2.1 Le type des listes

Une liste est un pointeur sur un noeud. Un noeud est un enregsistrement à deux champs : la valeur et un pointeur vers le noeud suivant.

```
#define LIST_TYPE int  /* le type des elements */

typedef struct NODE {
     struct NODE *suivant;
     LIST_TYPE value;
 } Node;

typedef Node *Liste;
```

2.2 Les méthodes de base : accesseurs

Tester si une liste est vide, retourner la tête d'une liste, retourner la queue d'une liste.

```
int empty(Liste l){
     return l==NULL;
 }

 LIST_TYPE car(Liste l){
     return l->value;
 }

 Liste cdr(Liste l){
     return l->suivant;
 }
```

2.3 Les méthodes de base : constructeurs

Construire une liste vide, construire une liste à l'aide de sa tête et de sa queue.

```
Liste newliste(){
    return NULL;
}

Liste cons(LIST_TYPE n, Liste l){
    Node *pNode = malloc(sizeof(Node));
    pNode->value = n;
    pNode->suivant = l;
    return pNode;
}
```

2.4 Manipulations de listes : versions récursives

Longueur d'une liste, concaténation de deux listes, déversement d'une liste dans une autre, miroir d'une liste.

```
int list_length(Liste l){
    if(empty(l)) return 0;
    else return list_length(cdr(l))+1;
}

Liste concat(Liste l1, Liste l2){
     if(empty(l1)) return l2;
     else return cons(car(l1),concat(cdr(l1),l2));
}

Liste deverse(Liste l1, Liste l2){
     if(empty(l1)) return l2;
     else return deverse(cdr(l1),cons(car(l1),l2));
}

Liste miroir(Liste l){
     return deverse(l,newliste());
}
```

2.5 Manipulations de listes : versions itératives

Longueur d'une liste, concaténation de deux listes, déversement d'une liste dans une autre, miroir d'une liste.

```
int list_length(Liste l){
    int n=0;
    for( ;!empty(l); n++, l=cdr(l));
    return n;
}

Liste deverse(Liste l1, Liste l2){
    Liste l = l2;
    for( ; !empty(l1); l=cons(car(l1),l),l1=cdr(l1));
```

```
    return l;
}

Liste miroir(Liste l){
     return deverse(l,newliste());
}

Liste concat(Liste l1, Liste l2){
    return deverse(miroir(l1),l2);
}
```

2.6 Opérations ensemblistes : versions récursives

Appartenance, adjonction d'un élément, réunion, intersection, suppression d'un élément.

```
int appartient(LIST_TYPE x,Liste l){
    if(empty(l)){ return 0; }
    else if(car(l)==x){ return 1; }
    else { return appartient(x,cdr(l)); }
}

Liste ajoute_elt(LIST_TYPE x, Liste l){
    if(appartient(x,l)) return l;
    else return cons(x,l);
}

Liste reunit(Liste l1, Liste l2){
    if(empty(l1)) return l2;
    else return ajoute_elt(car(l1),reunit(cdr(l1),l2));
}

Liste intersecte(Liste l1,Liste l2){
    if(empty(l1)) return newliste();
    else if(appartient(car(l1),l2))
            return cons(car(l1),intersecte(cdr(l1),l2));
    else return intersecte(cdr(l1),l2);
}

Liste supprime(LIST_TYPE x, Liste l){
    if(empty(l)) return l;
    else if(car(l)==x) return cdr(l);
    else return cons(car(l),supprime(x,cdr(l)));
}
```

2.7 Opérations ensemblistes : versions itératives

Appartenance, adjonction d'un élément, réunion, intersection, suppression d'un élément.

```
int appartient(LIST_TYPE x,Liste l){
     Liste l1;
     for(l1=l; !empty(l1); l1=cdr(l1))
          if(car(l1)==x){ return 1; }
     return 0;
 }

 Liste ajoute_elt(LIST_TYPE x, Liste l){
     if(appartient(x,l)) return l;
     else return cons(x,l);
 }

 Liste reunit(Liste l1, Liste l2){
     Liste l, res=l2;
     for(l=l1; !empty(l); l=cdr(l), res=ajoute_elt(car(l),res));
     return res;
  }

  Liste intersecte(Liste l1,Liste l2){
     Liste l, res=newliste();
     for(l=l1; !empty(l); l=cdr(l))
        if(appartient(car(l),l2)) res=cons(car(l),res);
     return res;
  }

Liste supprime(LIST_TYPE x, Liste l){
     Liste l1, res=newliste();
     for(l1=l; !empty(l1); l1=cdr(l1))
       if(x!=car(l1)) res=cons(car(l1),res);
     return res;
 }
```

2.8 Tri d'une liste par insertion

```
Liste insere(LIST_TYPE x, Liste triee){
     if(empty(triee)) return cons(x,triee);
     else if(x<=car(triee)) return cons(x,triee);
     else return cons(car(triee),insere(x,cdr(triee)));
 }

 Liste tri_insertion(Liste l){
     if(empty(l)) return l;
     else return insere(car(l),tri_insertion(cdr(l)));
 }
```

2.9 Tri d'une liste par insertion

```
typedef struct MINRESTE {
     LIST_TYPE minimum;
     Liste liste;
 } MinReste;

 MinReste extrait_minimum(Liste l){
     MinReste c;
     if(empty(l)){ c.minimum=10000; c.liste = NULL; return c;}
     else {
         LIST_TYPE h=car(l);
         c=extrait_minimum(cdr(l));
         if(h>c.minimum) { c.liste=cons(h,c.liste); return c;}
         else { c.minimum=h; c.liste=cdr(l); return c;}
     }
 }

 Liste tri_selection(Liste l){
     MinReste c=extrait_minimum(l);
     return cons(c.minimum,tri_insertion(c.liste));
 }
```

2.10 Les tas

Un tas est une liste modifiable. Trois méthodes : test de vacuité, empilement, dépilement LIFO. Un seul constructeur.

```
typedef struct STACK {
     Node *premier;
     } *Stack;

Stack new_stack(){
     Stack s;
     s=malloc(sizeof(struct STACK));
     s->premier=NULL;
     return s;
}

int empty_stack(Stack s){
    return s->premier==NULL;
}

void push_stack(LIST_TYPE x, Stack s){
    s->premier = cons(x, s->premier);
}

LIST_TYPE pop_stack(Stack s){
    LIST_TYPE x = car(s->premier);
    s->premier = cdr(s->premier);
```

```
    return x;
}
```

2.11 Les files d'attente ou queues

Liste chaînée avec un pointeur vers le début, un pointeur vers la fin. Trois méthodes : test de vacuité, empilement, dépilement FIFO. Un seul constructeur.

```
typedef struct QUEUE {
     Node *premier, *dernier;
 } *Queue;

 Queue new_queue(){
     Queue pQueue;
     pQueue=malloc(sizeof(struct QUEUE));
     pQueue->premier=NULL;
     pQueue->dernier=NULL;
     return pQueue;
 }

 int empty_queue(Queue q){
     return q->premier==NULL;
 }

 void push_queue(LIST_TYPE x, Queue q){
     Node *pNode = malloc(sizeof(Node));
     pNode->value = x;
     pNode->suivant = NULL;
     q->dernier->suivant = pNode;
     q->dernier=pNode;
     (* attention à ne pas oublier *)
     if(q->premier=NULL) q->premier=pNode;
 }

 LIST_TYPE pop_queue(Queue q){
     LIST_TYPE x = q->premier->value;
     q->premier=q->premier->suivant;
     (* attention à ne pas oublier *)
     if(q->premier==NULL) q->dernier=NULL;
     return x;
 }
```

2.12 Tri d'une liste par fusion

```
typedef struct DECOUPE {
     Liste fst, snd;
} Decoupe;

Decoupe coupe_en_deux(Liste l){
```

```
    Liste l1=l;
    Decoupe dec;
    if(empty(l)){
        dec.fst=newliste(); dec.snd=newliste();
    } else if(empty(cdr(l))){
        dec.fst=cons(car(l),newliste()); dec.snd=newliste();
    } else {
        dec=coupe_en_deux(cdr(cdr(l)));
        dec.fst=cons(car(l),dec.fst);
        dec.snd=cons(car(cdr(l)),dec.snd);
    }
    return dec;
}

Liste fusionne(Liste l1, Liste l2){
     if(empty(l1)) return l2;
     if(empty(l2)) return l1;
     if(car(l1)<car(l2)) return cons(car(l1),fusionne(cdr(l1),l2));
     return cons(car(l2), fusionne(l1,cdr(l2)));
}

Liste tri_fusion(Liste l){
     Decoupe dec = coupe_en_deux(l);
     return fusionne(tri_fusion(dec.fst),tri_fusion(dec.snd));
}
```

2.13 Tri rapide d'une liste

```
Decoupe coupe_pivot(Liste l, LIST_TYPE x){
    Decoupe dec;
    if(empty(l)){
        dec.fst=newliste(); dec.snd=newliste();
    } else {
        dec=coupe_pivot(cdr(l),x);
        if(car(l)<x) dec.fst=cons(car(l),dec.fst);
        else dec.snd=cons(car(l),dec.snd);
    }
    return dec;
}

Liste tri_rapide(Liste l){
     Decoupe dec = coupe_pivot(cdr(l),car(l));
     return concat(tri_rapide(dec.fst),cons(car(l),tri_rapide(dec.snd)));
}
```

2.14 Listes d'association

Définition dynamique d'une fonction par son graphe mathématique.

```
#define NON_TROUVE -1
#define DEPART_TYPE char  /* le type des elements */
#define ARRIVEE_TYPE int  /* le type des elements */

typedef struct ASSOCNODE {
     ASSOCNODE *suivant;
     DEPART_TYPE depart;
     ARRIVEE_TYPE arrive;
 } AssocNode;

typedef AssocNode *Assoc;

Assoc newAssoc(){
     return NULL;
 }

Assoc ajoute(Assoc graphe, DEPART_TYPE x, ARRIVEE_TYPE y){
     AssocNode *pNode = malloc(sizeof(AssocNode));
     pNode->depart = x;
     pNode->arrive = y;
     pNode->suivant = graphe;
     return pNode;
 }

ARRIVEE_TYPE assoc(Assoc graphe, DEPART_TYPE x){
     if(graphe==NULL) return NON_TROUVE;
     else if(graphe->depart==x) return graphe->arrive;
     else return assoc(graphe->suivant, x);
 }
```

Chapitre 3

Listes en C++

3.1 La classe liste

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

#define LIST_TYPE int  /* le type des elements */

typedef struct NODE {
     struct NODE *suivant;
     LIST_TYPE value;
 } Node;

 class Liste {
     struct NODE *first;
     public:
     // le constructeur
     Liste();
     // les méthodes
     int isEmpty();
     LIST_TYPE car();
     class Liste cdr();
     Liste cons(LIST_TYPE x);
     int length();
     Liste concat(Liste l);
     Liste deverse(Liste l);
     Liste miroir();
 }
```

3.2 Les méthodes de base : constructeurs

```
Liste::Liste() {
     first=NULL;
 }
```

```
 Liste Liste::cons(LIST_TYPE x){
     Node *pNode = new Node;
     Liste *res = new Liste();
     pNode->value = x;
     pNode->suivant = first;
     res->first=pNode;
     return *res;
 }
```

3.3 Les méthodes de base : accesseurs

Tester si une liste est vide, retourner la tête d'une liste, retourner la queue d'une liste.

```
int Liste::isEmpty(){
     return first==NULL;
 }

 LIST_TYPE Liste::car(){
     return first->value;
 }

 Liste Liste::cdr(){
     Liste *res=new Liste();
     res->first=first->suivant;
     return *res;
 }
```

3.4 Manipulations de listes : versions récursives

Longueur d'une liste, concaténation de deux listes, déversement d'une liste dans une autre, miroir d'une liste.

```
int Liste::length(){
     Node *pNode=first;
     int l=0;
     while(pNode!=NULL){
         l++;
         pNode=pNode->suivant;
     }
     return l;
 }

 Liste Liste::concat(Liste l){
     if(isEmpty()) return l;
     return cdr().concat(l).cons(car());
 }

 Liste Liste::deverse(Liste l){
```

```
    if(isEmpty()) return l;
    return l.cons(car()).deverse(cdr());
}

Liste Liste::miroir(){
    Liste *res = new Liste();
    return deverse(*res);
}
```

3.5 La classe Stack

```
class Stack{
    Liste laliste;
    public:
    // le constructeur
    Stack();
    // les méthodes
    int isEmpty();
    LIST_TYPE pop();
    void push(LIST_TYPE x);
}

Stack::Stack(){
    laliste=Liste();
}

int Stack::isEmpty(){
    return laliste.isEmpty();
}

LIST_TYPE Stack::pop(){
    LIST_TYPE x=laliste.car();
    laliste=laliste.cdr();
    return x;
}

void Stack::push(LIST_TYPE x){
    laliste=laliste.cons(x);
}
```

Chapitre 4

Arbres binaires hétérogènes

4.1 Le type des arbres hétérogènes

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

#define FEUILLES_TYPE int  /* le type des feuilles */
#define NOEUDS_TYPE char  /* le type des noeuds */

enum {FEUILLE,NOEUD};

typedef struct TREE {
    int type;
    union {
        FEUILLES_TYPE feuille;
        struct{
            NOEUDS_TYPE c;
            TREE *gauche,*droite;
        } noeud;
    } contenu;
} *Tree;
```

4.2 Les constructeurs d'arbres

Construire une feuille à partir de son étiquette, construire un noeud à partir de son étiquette et des deux branches.

```
Tree new_feuille(FEUILLES_TYPE x){
    Tree pTree=malloc(sizeof(TREE));
    pTree->type=FEUILLE;
    pTree->contenu.feuille=x;
    return pTree;
}

Tree new_noeud(NOEUDS_TYPE s, Tree g, Tree d){
```

```
    Tree pTree=malloc(sizeof(TREE));
    pTree->type=NOEUD;
    pTree->contenu.noeud.c=s;
    pTree->contenu.noeud.gauche=g;
    pTree->contenu.noeud.droite=d;
    return pTree;
}
```

4.3 Les méthodes des arbres : accesseurs

Branche gauche, branche droite, étiquette.

```
Tree gauche(Tree t){
    return t->contenu.noeud.gauche;
}

Tree droite(Tree t){
    return t->contenu.noeud.droite;
}

FEUILLES_TYPE etiquette_feuille(Tree t){
    return t->contenu.feuille;
}

NOEUDS_TYPE etiquette_noeud(Tree t){
    return t->contenu.noeud.c;
}
```

4.4 Les méthodes classiques

Nombre de feuilles, nombre de noeuds (internes), profondeur.

```
int nb_feuilles(Tree t){
    if(t->type==FEUILLE) return 1;
    else return nb_feuilles(gauche(t))+nb_feuilles(droite(t));
}

int nb_noeuds(Tree t){
    if(t->type==FEUILLE) return 0;
    else return nb_noeuds(gauche(t))+nb_noeuds(droite(t))+1;
}

int profondeur(Tree t){
    if(t->type==FEUILLE) return 0;
    else return max(profondeur(gauche(t)),profondeur(droite(t)))+1;
}
```

4.5 Parcours d'arbres

Parcours prefixe, infixe, postfixe. Utilise une fonction de traitement des feuilles et une fonction de traitement des noeuds.

```
void parcours_prefixe(Tree t,void (*f)(FEUILLES_TYPE), void (*n)(NOEUDS_TYPE)){
    if(t->type==FEUILLE) f(etiquette_feuille(t));
    else {
        n(etiquette_noeud(t));
        parcours_prefixe(gauche(t),f,n);
        parcours_prefixe(droite(t),f,n);
    }
}

void parcours_infixe(Tree t,void (*f)(FEUILLES_TYPE), void (*n)(NOEUDS_TYPE)){
    if(t->type==FEUILLE) f(etiquette_feuille(t));
    else {
        parcours_infixe(gauche(t),f,n);
        n(etiquette_noeud(t));
        parcours_infixe(droite(t),f,n);
    }
}

void parcours_postfixe(Tree t,void (*f)(FEUILLES_TYPE), void (*n)(NOEUDS_TYPE)){
    if(t->type==FEUILLE) f(etiquette_feuille(t));
    else {
        parcours_postfixe(gauche(t),f,n);
        parcours_postfixe(droite(t),f,n);
        n(etiquette_noeud(t));
    }
}
```

4.6 Reconstitution à partir d'un parcours

4.6.1 Les éléments de parcours

Un élément de parcours sera soit une feuille, soit un noeud (dont nous négligerons les branches). Un parcours sera un tableau d'éléments de parcours.

```
typedef struct TREE parcours_elt;
typedef parcours_elt *parcours;
```

4.6.2 Reconstitution préfixe

```
Tree reconstitue_prefixe_aux(parcours_elt *parcours,int *i0,int n){
    parcours_elt elt=parcours[*i0];
    if(elt.type==FEUILLE){
        (*i0)++;
        return new_feuille(elt.contenu.feuille);
    } else {
```

```
        (*i0)++;
        Tree g=reconstitue_prefixe_aux(parcours,i0,n);
        Tree d=reconstitue_prefixe_aux(parcours,i0,n);
        return new_noeud(elt.contenu.noeud.c,g,d);
    }
}

Tree reconstitue_prefixe(parcours_elt *parcours){
    int i=0, n=sizeof(parcours)/sizeof(parcours_elt);
    Tree theTree=reconstitue_prefixe_aux(parcours,&i,n);
    if(i==n) return theTree; else return NULL;
}
```

4.6.3 Reconstitution postfixe

```
#define PILE_TYPE Tree
#include "pile.c"


Tree reconstitue_postfixe(parcours_elt *parcours){
    pile laPile=new_pile();
    int i=0, n=sizeof(parcours)/sizeof(parcours_elt);
    while(i<n){
        parcours_elt elt=parcours[i];
        if(elt.type==FEUILLE){
            i++;
            push(laPile,new_feuille(elt.contenu.feuille));
        } else {
            i++;
            Tree d=pop(laPile);
            Tree g=pop(laPile);
            push(laPile,new_noeud(elt.contenu.noeud.c,g,d));
        }
    }
    Tree theTree=pop(laPile);
    return theTree;
}
```

Chapitre 5

Arbres binaires homogènes

5.1 Le type des arbres binaires

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

#define TREE_TYPE int  /* le type des elements */

typedef struct BINTREE {
    TREE_TYPE racine;
    BINTREE *gauche, *droite;
} *BinTree;
```

5.2 Arbres binaires : constructeur

```
BinTree new_bintree(TREE_TYPE r, BinTree g, BinTree d){
    BinTree b = malloc(sizeof(struct BINTREE));
    b->racine=r;
    b->gauche=g;
    b->droite=d;
    return b;
}
```

5.3 Arbres binaires : méthodes classiques

```
int empty(BinTree t){
    return t==NULL;
}

int nb_noeuds(BinTree t){
    if(empty(t)) return 0;
    else return 1+nb_noeuds(t->gauche)+nb_noeuds(t->droite);
}
```

```
int hauteur(BinTree t){
    if(empty(t)) return 0;
    else return 1+max(nb_noeuds(t->gauche),nb_noeuds(t->droite));
}
```

5.4 Arbres binaires de recherche

5.4.1 Test d'un arbre binaire de recherche

```
void cherche_min_max(BinTree t,TREE_TYPE *mint, TREE_TYPE *maxt){
    TREE_TYPE ming, mind, maxg, maxd;
    if(empty(t)){ *mint = 10000; *maxt = -10000;
    } else {
        cherche_min_max(t->gauche,&ming,&maxg);
        cherche_min_max(t->droite,&mind,&maxd);
        *mint=min(ming,mind,t->racine);
        *maxt=max(maxg,maxd);
    }
}

void teste_recherche_aux(BinTree t,int *mint, int *maxt, int *test){
    TREE_TYPE ming, mind, maxg, maxd;
    int testg, testd;
    if(empty(t)){ *mint = 10000; *maxt = -10000; *test=1;
    } else {
        TREE_TYPE n = t->racine;
        teste_recherche_aux(t->gauche,&ming,&maxg,&testg);
        teste_recherche_aux(t->droite,&mind,&maxd,&testd);
        *test=testg && testd && (n >= maxg) && (n < mind);
        *mint=min(ming,mind,n);
        *maxt=max(maxg,maxd,n);
    }
}

int teste_recherche(BinTree t){
    TREE_TYPE mint,maxt;
    int test;
    teste_recherche_aux(t,&mint,&maxt,&test);
    return test;
}
```

5.4.2 Recherche dans un arbre binaire de recherche

```
int figure_dans_arbre(TREE_TYPE x, BinTree t){
    if(empty(t)) return 0;
    else if(t->racine==x) return 1;
    else if(t->racine<x) return figure_dans_arbre(x,t->gauche);
    else return figure_dans_arbre(x,t->droite);
}
```

5.4.3 Insertion aux feuilles

```
void insere_feuille(TREE_TYPE x, BinTree *t){
    if(empty(*t)) *t=new_bintree(x,NULL,NULL);
    else if(x<=(*t)->racine) insere_feuille(x,(*t)->gauche);
    else insere_feuille(x,(*t)->droite);
}
```

5.4.4 Insertion à la racine

```
void coupe(TREE_TYPE x, BinTree t, BinTree *petit, BinTree *grand){
    if(empty(t)) {*petit=NULL; *grand=NULL; }
    else {
        if(x<=t->racine){
            BinTree petitg, grandg;
            coupe(x,t->gauche,&petitg,&grandg);
            *petit=petitg;
            *grand=new_bintree(t->racine,grandg,t->droite);
        } else {
            BinTree petitd, grandd;
            coupe(x,t->droite,&petitd,&grandd);
            *grand=grandd;
            *petit=new_bintree(t->racine,t->gauche,petitd);
        }
    }
}

void insere_racine(TREE_TYPE x, BinTree *t){
    if(empty(*t)) *t=new_bintree(x,NULL,NULL);
    else {
        BinTree petit, grand;
        coupe(x,t,&petit,&grand);
        *t=new_bintree(x,petit,grand);
    }
}
```

5.4.5 Suppression dans un arbre de recherche

Pour supprimer un élément, on cherche cet élément, puis on supprime la racine de la branche dont l'arbre est la racine. Pour supprimer une racine, on cherche l'élément le plus à droite de la branche gauche, on le supprime de la branche gauche et on en fait la nouvelle racine.

```
int plus_a_droite(BinTree *t){
    if((*t)->droite==NULL){
        *t = (*t)->gauche;
        return (*t)->racine;}
    else{
        return plus_a_droite(&((*t)->droite));
    }
}
```

```
void supprime_racine(BinTree *t){
    if((*t)->gauche==NULL){
        *t = (*t)->droite;
        return;
    }
    int new_rac = plus_a_droite(t);
    *t= new_bintree(new_rac,(*t)->gauche,(*t)->droite);
}

void supprime(TREE_TYPE x, BinTree t){
    if(t==NULL) return;
    if(t->racine==x) supprime_racine(&t);
    else if(t->racine>x) supprime(x,&(t->gauche));
    else supprime(x,&(t->droite));
}
```

5.5 Files de priorité

5.5.1 Test d'une file de priorité

```
int estPriorite(BinTree t){
    if(t==NULL) return 1;
    int testg=(t->gauche==NULL) || (t->racine >= t->gauche->racine);
    int testd=(t->droite==NULL) || (t->racine >= t->droite->racine);
    return testg && testd && estPriorite(t->gauche) && estPriorite(t->droite);
}
```

5.5.2 Insertion dans une file de priorité

```
void inserePriorite(TREE_TYPE x,BinTree *t){
   if(*t==NULL) {*t=new_bintree(x,NULL,NULL); return; }
   if((*t)->racine<x) {
        inserePriorite((*t)->racine,(*t)->gauche);
        (*t)->racine = x;
   } else {
        inserePriorite(x, (*t)->droite);
   }
}
```

5.5.3 Dépilement dans une file de priorité

```
void supprimePriorite(BinTree *t){
    if(*t==NULL) return;
    if((*t)->gauche==NULL) *t=(*t)->droite;
    else if((*t)->droite==NULL) *t=(*t)->gauche;
    else if((*t)->gauche->racine>(*t)->droite->racine){
        (*t)->racine = (*t)->gauche->racine;
        supprimePriorite((*t)->gauche);
    } else {
```

```
        (*t)->racine = (*t)->droite->racine;
        supprimePriorite((*t)->droite);
    }
}
```

Chapitre 6

Expressions algébriques

6.1 Expressions algébriques totalement parenthésées

6.1.1 Le type des EATP (syntaxe concrète)

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

enum elt_type {NB, VAR, FCT, OP, DELIM};

typedef struct {
     elt_type type;
     union {
        float x;
        char* var;
        char* fct;
        char op;
        char delim;
     } c;
} elt;

/* fonction polymorphe de création d'élément */
elt newElt(elt_type type,...){
    va_list ap;
    elt e;
    va_start(ap,1);
    e.type=type;
    switch(type){
       case NB:
           e.c.x=va_arg(ap,float);
           break;
       case VAR:
           e.c.var=va_arg(ap,char*);
           break;
       case FCT:
```

```
            e.c.fct=va_arg(ap,char*);
            break;
        case OP:
            e.c.op=va_arg(ap,char);
            break;
        case DELIM:
            e.c.delim=va_arg(ap,char);
   }
   return e;
}
```

6.1.2 Arbres d'expressions algébriques (syntaxe abstraite)

```
#define LIST_TYPE elt
#include "listeslib.c"
typedef Liste EATP;
typedef struct ND{
    elt_type type;
    char *name;
    union {
        float x;
        char *var;
        ND *nd1;
        struct {ND* gauche; ND* droite;} nd2;
    } c;
} noeud;
/* fonction polymorphe de création de noeud */
noeud newNoeud(elt_type type,...){
    va_list ap;
    noeud n;
    va_start(ap,1);
    n.type=type;
    switch(type){
        case NB:
            n.c.x=va_arg(ap,float);
            break;
        case VAR:
            n.c.var=va_arg(ap,char*);
            break;
        case FCT:
            n.c.nd1=va_arg(ap,ND*);
            break;
        case OP:
            n.c.nd2.gauche=va_arg(ap,ND*);
            n.c.nd2.droite=va_arg(ap,ND*);
            break;
   }
   va_end(ap);
   return n;
}
```

6.2 Arbre en eatp

```
EATP arbre_en_eatp(noeud n){
    switch(n.type){
        case NB:
            return cons(newElt(NB,n.c.x),NULL);
            break;
        case VAR:
            return cons(newElt(VAR,n.c.var),NULL);
            break;
        case FCT:
            return cons(newElt(FCT,n.name),arbre_en_eatp(*(n.c.nd1)));
            break;
        case OP:
            EATP e;
            e=cons(newElt(DELIM,')'),NULL);
            e=concat(arbre_en_eatp(*(n.c.nd2.droite)),e);
            e=cons(newElt(OP,n.name),e);
            e=concat(arbre_en_eatp(*(n.c.nd2.gauche)),e);
            return cons(newElt(DELIM,'('),e);
    }
}
```

6.3 Eatp en arbre

Ebauche de fonction à complêter.

```
noeud eatp_en_arbre(EATP *eatp){
     elt e=car(*eatp);
     switch(e.type){
         case NB:
             *eatp=cdr(*eatp);
             return newNoeud(NB,e.c.x);
         case VAR:
             *eatp=cdr(*eatp);
             return newNoeud(VAR,e.c.var);
         case FCT:
             *eatp=cdr(*eatp);
             return newNoeud(FCT,car(*eatp).c.fct,eatp_en_arbre(*eatp));
         case DELIM:
             *eatp=cdr(*eatp); /* supprime la parenthèse ouvrante */
             noeud n1=eatp_en_arbre(*eatp);
             e=car(*eatp);
             noeud n2=eatp_en_arbre(*eatp);
             *eatp=cdr(*eatp);  /* supprime la parenthèse fermante */
             return newNoeud(OP,e.c.op,n1,n2);
     }
}
```

Chapitre 7

Graphes

7.1 Structure de données

```
#include <stdlib.h>
#include <stdio.h>
#define maxV 100

/* les arêtes */
typedef struct { int v; int w; } Edge;
Edge new_Edge(int, int);

/* les noeuds */
typedef struct node
  { int v; node* next;
  } *Node;

typedef struct graph *Graph;
struct graph { int V; int E; Node *adj; };
```

7.2 Constructeurs

```
Node new_Node(int x, Node t)
      {   Node n = malloc(sizeof(node));
          n->v = x; n->next = t;
          return n;
      }

Edge new_Edge(int v, int w){
    Edge* e=malloc(sizeof(Edge));
    e->v = v; e->w=w;
    return *e;
}

/* initialise un graphe à V noeuds, sans arêtes */
Graph GRAPHinit(int V)
```

```
  { int v;
    Graph G = malloc(sizeof *G);
    G->V = V; G->E = 0;
    G->adj = malloc(V*sizeof(Node));
    for (v = 0; v < V; v++) G->adj[v] = NULL;
    return G;
  }

/* insère une arête e dans le graphe G */
void GRAPHinsertE(Graph G, Edge e)
  { int v = e.v, w = e.w;
    G->adj[v] = new_Node(w, G->adj[v]);
    G->adj[w] = new_Node(v, G->adj[w]);
    G->E++;
  }
```

7.3 Conversions matrices - listes d'adjacence

```
/* listes d'adjacence -> matrice d'adjacence */
int** AdjToMat(Node* adj, int V){
    int** mat = malloc(V*sizeof(int*));
    int i,j; Node n;
    for(i=0; i<V; i++){
         mat[i]=malloc(V*sizeof(int));
         for(j=0;j<V;j++) mat[i][j]=0;
         for(n=adj[i]; n!= NULL; n=n->next) mat[i][n->v]=1;
    }
    return mat;
}

/* matrice d'adjacence -> listes d'adjacence */
Node* MatToAdj(int** mat,int V){
    Node* adj = malloc(V*sizeof(Node));
    int i,j;
    for(i=0;i<V;i++){
        adj[i]=NULL;
        for(j=0;j<V;j++)
            if(mat[i][j]!=0) adj[i]=new_Node(j,adj[i]);
    }
    return adj;
}
```

7.4 Visualisation

```
/* construit un tableau des arêtes du graphe */
int GRAPHedges(Edge a[], Graph G)
```

```
  { int v, E = 0; Node t;
    for (v = 0; v < G->V; v++)
      for (t = G->adj[v]; t != NULL; t = t->next)
        if (v < t->v) a[E++] = new_Edge(v, t->v);
    return E;
  }

  void GRAPHshow(Graph G)
  { int i; Node n;
    printf("%d vertices, %d edges\n", G->V, G->E);
    for (i = 0; i < G->V; i++)
      {
        printf("%2d:", i);
        for (n=G->adj[i]; n!=NULL; n=n->next)
            if(i<n->v) printf(" %2d", n-> v);
        printf("\n");
      }
  }
```

7.5 Graphes aléatoires

```
/* construction de graphe aléatoire */
int randV(Graph G)
  { return G->V * (rand() / (RAND_MAX + 1.0)); }

Graph GRAPHrand(int V, int E)
  { Graph G = GRAPHinit(V);
    while (G->E < E)
      GRAPHinsertE(G, new_Edge(randV(G), randV(G)));
    return G;
  }
```

7.6 Recherche de chemin

```
/* recherche d'existence de chemin dans un graphe */
int pathR(Graph G, int v, int w, int* visited)
  { Node n;
    if (v == w) return 1;
    visited[v] = 1;
    for(n=G->adj[v]; n!=NULL; n=n->next)
      if (visited[n->v] == 0)
          if (pathR(G, n->v, w)) return 1;
    return 0;
  }
int GRAPHpath(Graph G, int v, int w)
  { int t;
    int visited[maxV];
    for (t = 0; t < G->V; t++) visited[t] = 0;
```

```
    return pathR(G, v, w, visited);
  }
```

7.7 Composantes connexes

```
/* Recherche des composantes connexes d'un graphe par parcours en profondeur
   */
 void dfsRcc(Graph G, int v, int id, int* cc)
  { Node t;
    cc[v] = id;
    for (t = G->adj[v]; t != NULL; t = t->next)
      if (cc[t->v] == -1) dfsRcc(G, t->v, id,cc);
  }

int* GRAPHcc(Graph G)
  { int v, id = 0;
    int* cc = malloc(G->V * sizeof(int));
    for (v = 0; v < G->V; v++)
      cc[v] = -1;
    for (v = 0; v < G->V; v++)
      if (cc[v] == -1) dfsRcc(G, v, id++,cc);
    return cc;
  }
```

7.8 Tour d'Euler non orienté

```
/* Euler tour, chaque arête est parcourue deux fois */
  int pre[maxV];
  int cnt;

  void dfsReuler(Graph G, Edge e)
  { Node  t;
    printf("-%d", e.w);
    pre[e.w] = cnt++;
    for (t = G->adj[e.w]; t != NULL; t = t->next)
      if (pre[t->v] == -1)
        dfsReuler(G, new_Edge(e.w, t->v));
      else if (pre[t->v] < pre[e.v])
        printf("-%d-%d", t->v, e.w);
    if (e.v != e.w)
         printf("-%d", e.v);
    else printf("\n");
  }
```