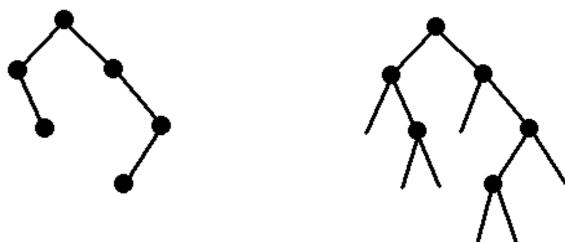
Arbre binaire

Par définition un arbre binaire est un arbre avec une racine, et où chacun des nœuds possède :

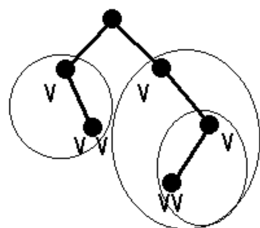
- soit aucun successeur,
- soit un successeur, à gauche ou à droite,
- soit deux successeurs.

Voici un exemple d'arbre binaire, dessiné avec sa racine en haut et ses branches en descente, à l'inverse d'un arbre dans la nature :



On peut aussi dessiner cet arbre (*à droite ci-dessus*) en considérant que chaque nœud est muni de deux branches pendantes auxquelles viennent s'accrocher, ou pas, d'autres nœuds.

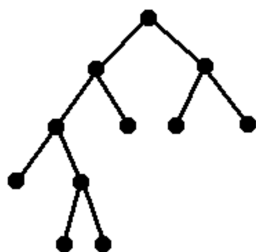
Un arbre binaire peut aussi être réduit à l'arbre vide. Pour comprendre l'importance de l'arbre vide, voici comment on peut donner une définition récursive tout à fait cohérente d'un arbre binaire : il s'agit soit de l'arbre vide, soit d'un nœud racine auquel sont accrochés un arbre à gauche et un arbre à droite aussi. Si l'on oubliait le cas de l'arbre vide, la définition perdrait toute signification : notamment l'arbre ne pourrait avoir aucune feuille, c'est-à-dire un nœud auxquels sont accrochés deux arbres vides.



L'arbre avec ses sous-arbres, notamment ses arbres vides *V*

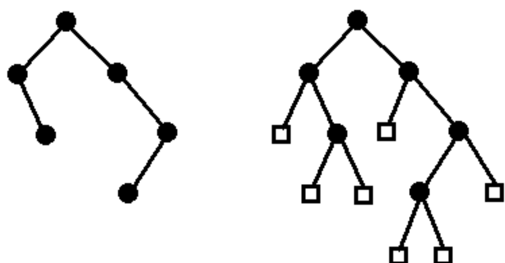
On définit aussi ce que l'on appelle un arbre binaire plein. Outre l'arbre vide, il s'agit d'un arbre avec une racine, et tel que chaque nœud possède soit aucun, soit deux successeurs, l'un à gauche et l'autre à droite. On distingue alors deux types de nœuds : les nœuds internes, qui ont deux successeurs, et les nœuds terminaux, sans successeurs. On vérifie qu'il y a toujours un nœud terminal de plus que de nœuds internes [1].

[1] Démontrons cette propriété. Appelons n le nombre de nœuds internes. Lorsque l'arbre est tel que $n=1$, il y a deux nœuds terminaux, et la propriété est vraie pour $n=1$. Supposons que la propriété soit vraie pour un certain nombre n de nœuds internes. Et montrons qu'elle le reste pour $n+1$ nœuds, à savoir qu'il y a un nœud terminal de plus. En effet le passage de n nœuds internes à $n+1$ nœuds internes se fait en transformant un nœud terminal en nœud interne avec ses deux nœuds terminaux, ce qui maintient la différence de 1 entre le nombre des nœuds terminaux et celui des nœuds internes.



Un arbre binaire rempli avec 11 nœuds, dont 5 internes et 6 terminaux

Il existe un lien étroit entre un arbre binaire et un arbre binaire plein. On peut en effet associer à tout arbre binaire, avec ses n nœuds, un arbre binaire rempli avec les mêmes n nœuds qui deviennent maintenant des nœuds internes, et n+1 nœuds terminaux ajoutés au-dessous d'eux. Ce processus est réversible: il suffit d'enlever les nœuds terminaux ou de les rajouter pour passer d'un type d'arbre à l'autre.



Un arbre binaire et l'arbre binaire plein correspondant

1. Nombre d'arbres binaires à *n* nœuds

Appelons $c(n)$ le nombre d'arbres binaires à n nœuds (ou encore avec n nœuds internes et n+1 nœuds terminaux, en prenant les arbres pleins). A cause de l'arbre vide, on a $c(0)=1$. Donnons-nous $n>0$. Un nœud est la racine. Il reste à placer les n-1 nœuds restants dans le sous-arbre gauche et dans le sous-arbre droit, de toutes les façons possibles. Soit on ne place aucun nœud à gauche (sous-arbre vide), ce qui fait $c(0)=1$ façon, et n-1 nœuds dans le sous-arbre droit, avec $c(n-1)$ possibilités, ce qui fait au total $c(0)\ c(n-1)$ cas. Soit on place un nœud à gauche, et n-2 nœuds à droite, ce qui fait $c(1)\ c(n-2)$ cas. Plus généralement, on place k nœuds à gauche, d'où $c(k)$ sous-arbres gauches, et n-1-k nœuds à droite, d'où $c(n-1-k)$ sous-arbres droits, ce qui fait au total $c(k)\ c(n-1-k)$ arbres de la sorte. Cela vaut pour toutes les valeurs de k de 0 à n-1. Finalement le nombre d'arbres ayant n nœuds est :

$$c(n) = c(0)\ c(n-1) + c(1)\ c(n-2) + \ldots + c(k)\ c(n-1-k) + \ldots + c(n-1)\ c(0).$$

Cette formule de récurrence, jointe à la condition initiale $c(0)=1$, donne les nombres $c(n)$ de proche en proche pour tout n. On appelle les nombres $c(n)$ nombres de Catalan.

A partir de $c(0)$ on trouve la succession 1, 1, 2, 5, 14, 42, 132, …

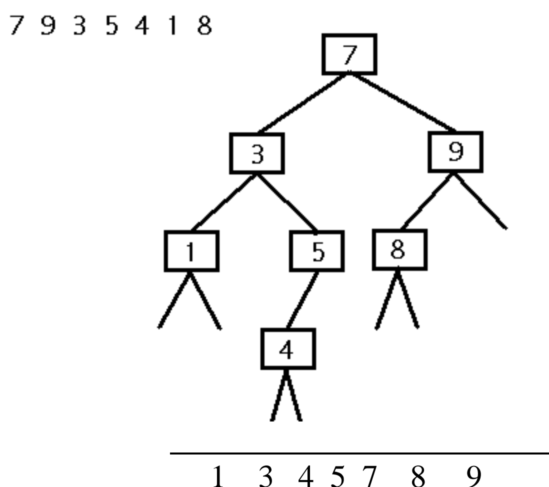
Programme permettant de calculer les nombres de Catalan de $c(0)$ à $c(N)$

On se donne le nombre N. Les nombres de Catalan vont être placés l'un après l''autre dans un tableau $C[N+1]$ vide au départ.

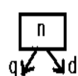
```
C[0]=1 ;  afficher
for(n=1 ; n<=N ; n++)
  { cumul=0 ;
    for(k=0 ;k<n ; k++) cumul += C[k]*C[n-1-k] ;
    C[n]=cumul ; afficher
  }
```

2. Le tri par arbre binaire ou tri des bijoutiers

Il s'agit là d'un exemple où l'arbre binaire doit être fabriqué nœud après nœud, par le programme lui-même. Au départ il s'agit d'un banal problème de tri, plus précisément le tri des bijoutiers, appelé aussi tri rapide, ou *quick sort* en anglais. Pour trier un tas de diamants, le bijoutier utilise un tamis qui permet de séparer le tas initial en deux tas. Et il recommence avec chacun des deux tas. La bonne qualité du tri dépend alors des tamis utilisés à chaque étape, de façon que chaque tas soit séparé en deux parties à peu près égales. Si les mailles des tamis sont trop grandes ou trop petites, le tri n'aura rien de rapide. En termes informatiques, cela se traduit de la façon suivante. On part d'une liste de nombres à trier. On prend le premier nombre, qui constitue la racine de l'arbre que l'on va créer. Ce nœud, comme tous ceux qui vont suivre, est dessiné avec deux branches pendantes. Puis on prend le deuxième nombre. Selon qu'il est plus petit ou plus grand que le premier, on l'accroche sous la racine à gauche ou à droite. Et l'on continue de la même façon. Chaque nombre est descendu dans l'arbre, en allant à gauche s'il est plus petit que le nombre du nœud où il passe, et à droite s'il est plus grand. Il prend finalement la première place vide sur laquelle il tombe. Par exemple :



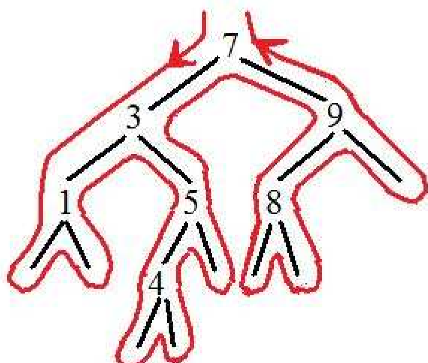
Il s'agit bien du tri des bijoutiers puisqu'à gauche de chaque nœud de l'arbre se trouvent les nombres plus petits, et à sa droite les nombres plus grands. L'arbre est construit peu à peu par adjonctions de briques élémentaires ainsi constituées :



Sous réserve que l'arbre soit dessiné comme ci-dessus (en diminuant l'angle entre les branches pendantes lors de la descente) on constate qu'n projetant l'arbre sur une ligne horizontale, comme si on l'écrasait sur le sol, les nombres sont dans l'ordre croissant : 1 3 4 5 7 8 9. Mais comme cette opération de projection n'est pas simple à réaliser sur ordinateur, on va chercher une lecture des nombres en ordre croissant par un certain parcours de l'arbre.

2.1. Les trois parcours d'un arbre

Pour parcourir un arbre de la racine jusqu'au retour à la racine, on dispose du chemin indiqué ci-dessous :



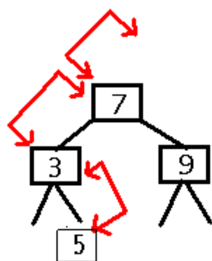
On constate que ce chemin passe trois fois le long de chaque nœud, soit en descente, soit au-dessous, soit lors de la remontée. Si l'on note les nombres situés dans les nœuds lors de la descente, on obtient le parcours dit préfixé : 7 3 1 5 4 9 8 avec la racine en premier, puis la racine du sous-arbre gauche, etc. Lorsqu'on note les nombres quand on passe en-dessous des nœuds, on a le parcours dit infixé : 1 3 4 5 7 8 9 avec la racine 7 au centre, ayant à sa gauche le sous-arbre gauche et à droite le sous-arbre droit. C'est ce parcours qui donne les nombres en ordre croissant. Enfin si l'on note les nombres lors de la remontée on a le parcours dit postfixé : 1 4 5 3 8 9 7 avec la racine en dernier. Nous verrons que la programmation de ces parcours est simple.

2.2. Comment fabriquer l'arbre ?

Cela se fait en deux étapes. D'abord on met en place la racine. Puis on insère les autres cellules une à une, en utilisant un crochet que l'on fait descendre dans l'arbre, en allant à gauche ou à droite selon que le nombre à placer est plus petit ou plus grand que les nombres déjà placés dans les cellules. Quand le crochet a son extrémité basse qui tombe sur une place vide, c'est là que va être placée la nouvelle cellule avec son nombre. D'où le programme :

```
/* les  nombres à trier sont supposés placés dans un tableau a[N] */
/* déclaration de la cellule de base */
struct cell { int n ; struct cell * g ; struct cell * d ;} ;

/* mise en place de la racine */
racine= (struct cell *) malloc (sizeof (struct cell)) ; racine->n=a[0]; racine->g=NULL; racine->d=NULL;
/* insertion progressive des nombres dans l'arbre */
for(i=1 ;i<N ;i++)
  { e1= NULL; e2=racine; /* mise en place du crochet (e1 e2) en haut de l'arbre */
    /* la nouvelle cellule */
    ptr==(struct cell*)malloc (sizeof (struct cell)) ; ptr->n= a[i]; ptr->g=NULL; ptr->d=NULL;
        while (e2!=NULL)     /* on descend le crochet dans l'arbre */
            { e1=e2;  if (a[i]<e2->n) e2=e2->g;  else e2=e2->d;  }
    if (a[i]<e1->n) e1->g = ptr; else e1->d = ptr;  /* on accroche la cellule pour de bon */
  }
```



Insertion de la cellule 5 dans l’arbre en construction

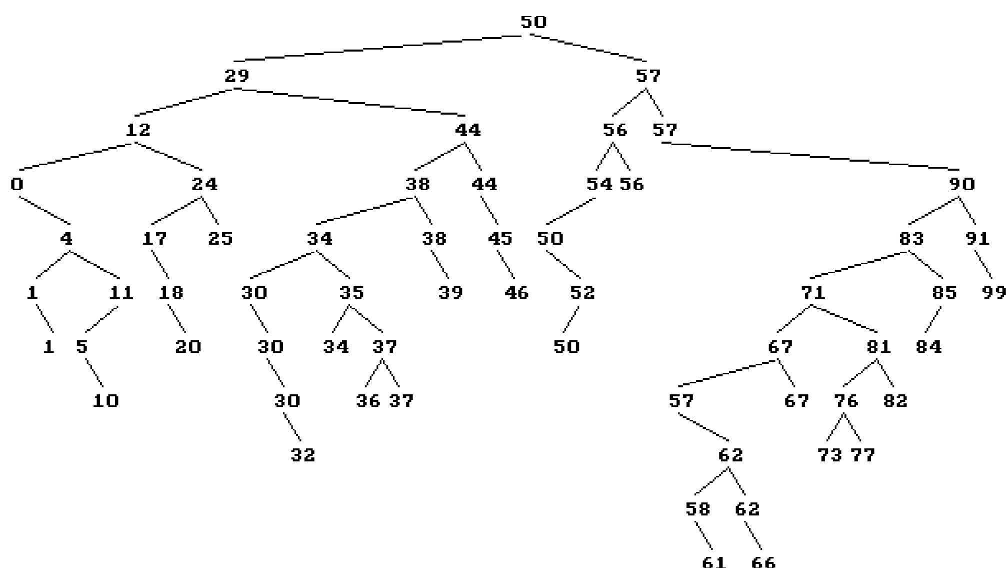
Il reste maintenant à afficher le résultat, à savoir les nombres triés. Pour cela on utilise une fonction de parcours de l’arbre, que l’on lance à partir de la racine, soit *parcours*(racine). Cette fonction se rappelle sur chacun des nœuds *r* de l’arbre :

```
void parcours( struct cell * r)
{  if (r !=NULL)  { parcours (r->g); printf(“%d”,  r->n);  parcours(r->d); }
}
```

Les nœuds sont affichés entre les parcours des sous-arbres gauche et droite. Il s’agit du parcours dit infixé de l’arbre, qui donne les nombres triés du plus petit au plus grand. Remarquons que si l’on plaçait l’affichage non plus entre les deux parcours des sous-arbres gauche et droit, mais avant, on aurait le parcours préfixé, et si on le plaçait à la fin on aurait le parcours postfixé.

2.3. Dessin de l’arbre

On peut aussi afficher l’arbre sur l’écran en gérant les coordonnées de chaque cellule. A chaque étape du parcours de l’arbre, lorsque se produit l’affichage du nombre *r->n*, il suffit d’augmenter l’abscisse d’une quantité constante (notée *pasx*). Cela correspond à la projection de l’arbre sur le sol comme nous l’avions dit auparavant. Et à chaque descente dans l’arbre lors de sa fabrication, l’ordonnée augmente aussi d’une quantité constante (*pasy*). Il convient aussi de conserver le prédécesseur de chaque nœud afin de pouvoir tracer les branches de jonction.



La cellule de base correspondant à un noeud contient maintenant des composantes supplémentaires: ses coordonnées *abs* et *ord*, ainsi qu'un pointeur *pred* vers son predecesseur.

```
struct cell { int n; struct cell * g; struct cell * d; int abs; int ord; struct cell * pred;};

main()
{ x=10;
  randomize();    for(i=0;i<N;i++) a[i]=random(100);  /* tableau de nombres pris au hasard */
 for(i=0;i<N;i++)
   {  ptr=(struct cell *) malloc(sizeof(struct cell));    ptr->n=a[i]; ptr->g=NULL; ptr->d=NULL;
     if (i==0) {racine=ptr;racine->pred=NULL; racine->ord=10;}
     else { e1=NULL; e2=racine; y=10;
            while (e2!=NULL) { e1=e2; if (a[i]<e2->n) e2=e2->g; else e2=e2->d; y+=pasy; }
            if (a[i]<e1->n) e1->g=ptr; else e1->d=ptr;
            ptr->pred=e1; ptr->ord=y;
          }
   }
calculabs(racine);   parcours(racine);
}

void parcours(struct cell * r)
{ if (r!=NULL)
   { parcours (r->g);
     afficher r->n  en  (r->abs, r->ord);
     if (r!=racine)   line(r->abs,r->ord,r->pred->abs,r->pred->ord);
     parcours (r->d);
   }
}

void calculabs(struct cell * r)
{    if (r!=NULL)  { calculabs(r->g); x+=pasx;r->abs=x; calculabs(r->d); }   }
```

Notons que nous avons fait deux parcours de l'arbre, dont le premier appelé *calculabs* détermine seulement les abscisses des nœuds. En effet si l'on se contenant du seul *parcours* de l'arbre en y intégrant le calcul des abscisses, tout en dessinant le lien entre le nœud concerné et son prédécesseur, on n'aurait pas forcément à sa disposition l'abscisse du prédécesseur, car celle-ci est parfois calculée plus tard dans la fonction *parcours*.

3. Exercice : Insertion, recherche, et suppression dans un arbre binaire

A partir de nombres que l'on se donne, on va construire l'arbre binaire correspondant, comme auparavant, mais en mettant en place les fonctions classiques associées, celles qui permettent d'insérer un nombre dans l'arbre, de supprimer un nombre en cas de besoin, et de chercher si un nombre est présent ou non dans l'arbre.

Dans un contexte plus réaliste, on serait amené à entrer les nombres au coup par coup dans l'arbre. Ici, on va simplement se donner un tableau *a*[*N*] de *N* nombres, puis insérer dans l'arbre les nombres successifs de ce tableau.

1) Construire une liste de N nombres pris au hasard, et tous différents

Pour remplir la case numéro *i* du tableau *a*[*N*], on tire un nombre *h* au hasard, et on recommence le tirage tant que ce nombre *h* est déjà placé dans la partie du tableau précédemment remplie (de la case 0 à la case *i* – 1).

```
void creationtableau(void)
 {  int i,j,h,dejafait;
    srand(time(NULL));
   for(i=0;i<N; i++)
    { do
       { dejafait=NON;       h=rand()%(3*N);   /* un nombre entre 0 et 3N – 1 */
         for(j=0;j<i;j++) if(a[j]==h) {dejafait=OUI; break;}
       }
      while( dejafait==OUI);
       a[i]=h;
     }
   for(i=0;i<N;i++) printf("%d ",a[i]);
 }
```

2) Fabriquer la fonction permettant d'insérer un nombre k dans l'arbre

Il existe deux cas, soit l'arbre est vide et l'on crée la racine de l'arbre pour y placer le nombre *k*, soit l'arbre n'est pas vide, et l'on descend dans l'arbre : en passant d'un nœud à son fils gauche si le nombre *k* est inférieur au nombre situé dans le nœud) ou sinon à droite, jusqu'à trouver une place libre pour la cellule contenant le nombre *k*. On reprend pour cela le programme donné au début de ce chapitre (*comment fabriquer l'arbre*), en utilisant une sorte de crochet d'extrémités haute et basse *e*1 et *e*2, descendant dans l'arbre. Quand le pointeur *e*2 tombe sur une place vide (*NULL*), on accroche la cellule contenant *k* au bout de la branche pendante sous la cellule où pointe *e*1, celle de droite ou celle de gauche selon la valeur de *k*.

```
void inserer(int k)
{
   if(racine==NULL) {racine=( struct cell * )malloc(sizeof(struct cell));
                     racine->n=k; racine->g=NULL; racine ->d=NULL;}
   else
     {  e1=NULL; e2=racine;
        newcell=( struct cell * )malloc(sizeof(struct cell));
        newcell->n=k; newcell->g=NULL; newcell->d=NULL;
        while(e2!=NULL)  { e1=e2; if (k<e2->n) e2=e2->g; else e2=e2->d; }
        if (k<e1->n) e1->g=newcell; else e1->d=newcell;
      }
}
```

3) Fabriquer une fonction de recherche d'un élément donné k à l'intérieur de l'arbre

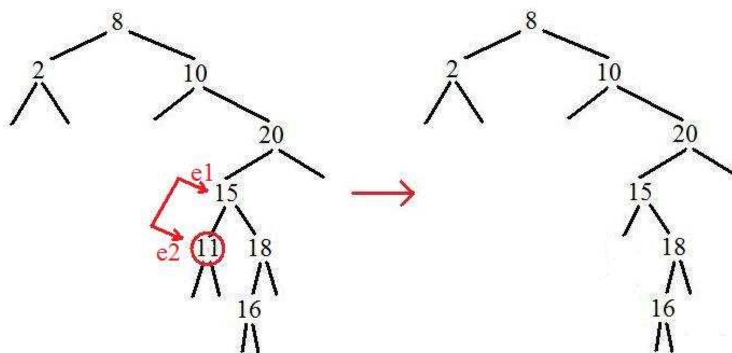
Quand l'arbre est construit, chaque cellule des nœuds contient un nombre *n*, et ce nombre est tel que tous les nombres qui sont dans le sous-arbre gauche du nœud sont inférieur à *n*, et que ceux situés dans son sous-arbre droit sont supérieurs. La recherche de la présence ou non d'un nombre donné *k* peut se faire de façon dichotomique, et récursive. Si le nombre *k* est inférieur à celui de la racine, on réduit la recherche en la relançant à partir du fils gauche, et sinon à partir du fils droit. Le nombre *k* étant donné, la fonction *chercher*(*k*, *r*) est appelée sur la racine, soit *chercher* (*k*, *racine*) dans le programme principal, puis elle se rappelle sur les nœuds *r* en descente.

```
void chercher(int k, struct cell * r)
{  if (r==NULL) printf("\n%d n'est pas present",k);
   else if (r->n==k) printf("\n%d est present", k);
   else
     { if (k<r->n) chercher(k,r->g);
       else if (k>r->n) chercher(k,r->d);
     }
}
```

4) Construire la fonction de suppression d'un élément k dans l'arbre

On suppose que le nombre *k* est bien situé dans l'arbre, quitte à lancer auparavant la fonction de recherche pour le savoir. La cellule contenant le nombre *k* est trouvée en faisant descendre le crochet *e*1 *e*2, jusqu'à ce que *e*2 pointe sur cette cellule. Plusieurs cas se présentent :

- La cellule de *k* est une feuille, c'est-à-dire qu'elle possède deux branches pendantes, pointant vers *NULL*. Si elle est à gauche au-dessous de *e*1, on met le fils gauche de *e*1 à *NULL*, et sinon le fils droit. La cellule de *k* est ainsi décrochée de l'arbre, et l'on peut ensuite libérer la place en mémoire jusque là occupée par cette cellule (*free*(*e*2)).

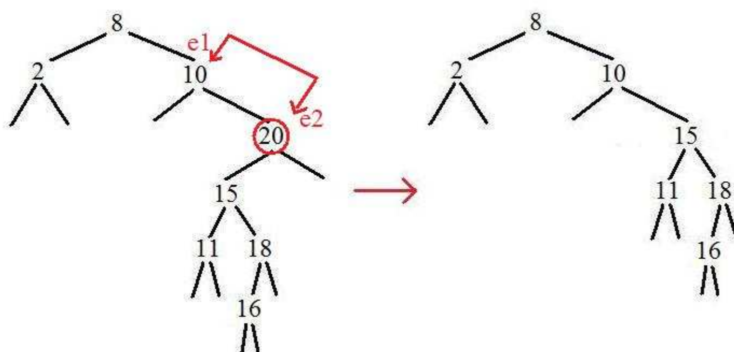


suppression de la cellule contenant 11

- La cellule contenant *k* possède une et une seule branche vide, celle de droite. Par exemple, si c'est la branche droite qui pointe sur *NULL*, et pas la gauche, on commence par regarder si la cellule de *k* est à gauche ou à droite de son « père » *e*1.

Si elle est à droite, comme sur le dessin ci-dessous, on accroche au fils droit de *e*1 le fils gauche de *e*2, ce qui met le sous-arbre gauche de *e*2 comme sous arbre droit de *e*1, tous ses éléments étant inférieurs à *k* mais supérieurs au nombre de la cellule sur lequel pointe *e*1.

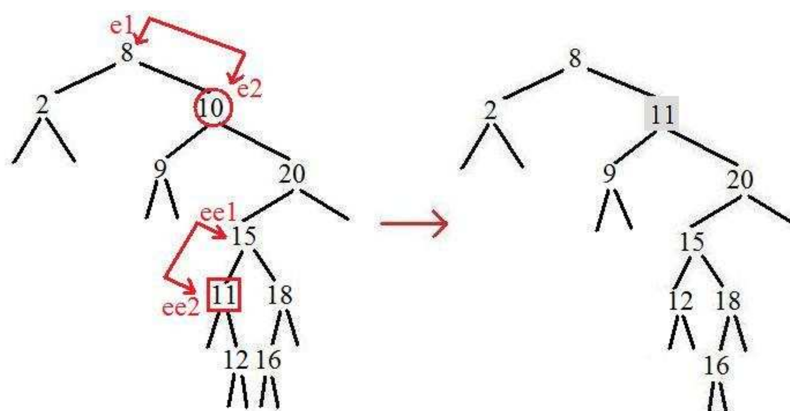
Si elle est à gauche, on accroche le sous-arbre gauche de *e*2 au fils gauche de *e*1.



suppression de la cellule contenant 20

- La cellule a comme branche vide unique celle située à sa gauche. On fait comme dans le cas précédent.

- La cellule n'a aucune branche vide. Dans ce cas, on commence par chercher la cellule contenant le plus petit nombre supérieur à k. Pour cela il faut d'abord emprunter le branche droite de la cellule de k, puis ensuite descendre toujours à gauche jusqu'à tomber sur une branche gauche pendante (vide). On utilise un nouveau crochet *ee*1 *ee*2, placé au départ avec *ee*1 sur *e*2 et *ee*2 sur le fils droit de *e*1. Puis on le descend, toujours à gauche, jusqu'à ce que *ee*2 pointe sur la cellule voulue (*ee*2->*g* == *NULL*), et *ee*1 sur son père. Dans l'exemple ci-dessous, à partir de la cellule contenant 10 à supprimer, on tombe sur la cellule contenant 11. C'est ce nombre qui va remplacer le 10 dans la cellule où pointe *e*2, et celle-ci conserve ces deux branches à droite et à gauche. Mais cela ne suffit pas. Il faut aussi supprimer la cellule contenant 11 (où pointe *ee*2) puisque la cellule sur laquelle pointe *e*2 la remplace, tout en conservant sa branche de droite (*ee*2 ->*d*), que l'on accroche maintenant à la branche gauche de son père *ee*1.



suppression du nombre 10, sa cellule étant conservée en y mettant 11 à la place

Mais il y a un cas particulier. On a vu que pour arriver à la position finale de *ee*2, on faisait d'abord un démarrage à droite, suivi de descentes à gauche. S'il y a au moins une descente à gauche, ce qui correspond au dessin ci-dessus, ce que l'on vient de faire est valable. Mais il peut arriver qu'il n'y ait aucune descente à gauche, auquel cas le crochet *ee*1 *ee*2 reste dans sa position initiale, avec *ee*2 à droite sous *ee*1 (qui est aussi *e*2). Dans ce cas il faut accrocher le sous-arbre droit de *ee*2 au sous-arbre droit de *e*2 (ou *ee*1), et non pas à celui de gauche comme auparavant.

La fonction de suppression s'en déduit :

```
void supprimer(int k)
{  struct cell *ee1,*ee2;
   e1=NULL; e2=racine;   /* recherché de la cellule contenant k */
   while(e2->n!=k)
     {  e1=e2;
        if (k<e2->n) e2=e2->g;    else e2=e2->d;
     }
   if (e2->g==NULL && e2->d==NULL)   /* la cellule est une feuille */
     {  if (e1->g==e2) e1->g=NULL; else e1->d =NULL;
        free(e2);
     }
   else if ( e2->g==NULL)   /* la cellule a sa branche gauche vide, mais pas l'autre */
```

```
        {  if(e1->g==e2)  e1->g=e2->d;  else e1->d=e2->d; free(e2);}
    else if ( e2->d==NULL)   /* la cellule a sa branche droite vide, mais pas l'autre  */
         { if(e1->g==e2)  e1->g=e2->g;  else e1->d=e2->g; free(e2);}
    else          /* la cellule a deux sous-arbres non vides accrochés à gauche et à droite  */
        {    ee2=e2->d; ee1=e2;
            while(ee2->g!=NULL)   /* ee2 va pointer sur le nombre qui remplace k dans sa cellule  */
              {    ee1=ee2; ee2=ee1->g;   }
            if(ee2==ee1->g)   ee1->g= ee2->d;  /* ee2 est à gauche de son père */
            else ee1->d=ee2->d;  /* cas particulier: pas de descente à gauche pour avoir ee2  */
            e2->n=ee2->n;
           free(ee2);
        }
}
```

5) Faire le programme principal utilisant les fonctions précédentes

Aux fonctions précédentes, on doit ajouter la fonction de parcours de l'arbre (*voir plus haut*) qui donne les *N* nombres une fois triés, ce qui permet de vérifier le bon fonctionnement des fonctions. Il reste le programme principal et les déclarations préliminaires :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 20
#define OUI 1
#define NON 0
int a[N];
struct cell { int n; struct cell * g; struct cell *d;};
struct cell * racine=NULL;  struct cell * newcell, *e1,*e2;
void creationtableau(void);
void inserer(int k);
void parcours(struct cell * r);
void chercher(int k, struct cell * r);
void supprimer(int k);

int main()
{  int i,j,hh,s;

   creationtableau();
   for(i=0;i<N;i++) inserer(a[i]);
   printf("\n\n");  parcours(racine);
   printf("\n"); chercher(20,racine);  /*  20 est-il présent?  */
   hh=rand()%N; s=a[hh]; printf("\ns= %d",s); /* on prend un élément au hasard à supprimer */
   supprimer(s);
   printf("\n\n"); parcours(racine);  /* pour constater la suppression de l'élément */
   getchar();return 0;
}
```