

Structures de données et Algorithmes

Phelma
PET 1A
Semestre2

Fonctionnement



Plan

- Analyse d'un problème
- Types abstraits
 - Notion de type abstrait
 - Liste implantation statique & dynamique
 - Pile
 - File, File circulaire
- Adressage associatif : tables de hashage
 - Principe & Exemples
 - Notion & gestion de collisions
- Arbres
 - Notion d'arbres, Exemples
 - Terminologie
 - Opérations de base
 - Parcours de base (profondeur, largeur, meilleur)
 - Arbres particuliers
 - Arbre Binaire de Recherche
 - Tas
- Graphes
 - Graphes & Exemples : optimisation & itinéraire, planéité, ordonnancement, capacité des tuyaux
 - Terminologie
 - Représentations

Rappel sur l'analyse



Rappel sur l'analyse

- 1- identifier les objets du problèmes (variables)
- 2- identifier les actions que l'on peut faire avec ses objets (fonctions)
- 3- identifier les étapes de résolutions (boucles, algorithme)
- 4- mettre en œuvre (coder en C ou autre)

Exemple : bataille

- ✓ La bataille se joue avec un jeu ordinaire de 52 cartes, à deux joueurs.
- ✓ Au départ, les cartes sont placées faces cachées en 2 paquets de 26 cartes, 1 paquet pour chaque joueur.
- ✓ Chaque joueur prend la première carte sur le dessus de son paquet, la retourne, et la pose à côté, sur la pile de bataille.
- ✓ Si les deux cartes posées par chaque joueur sont identiques, les deux joueurs prennent une carte qu'ils mettent face cachée sur le dessus de la pile de bataille, puis une seconde, qu'ils posent face visible sur cette pile.
- ✓ Les joueurs recommencent tant que les cartes visibles sur la pile de bataille sont égales.
- ✓ Lorsque ces 2 cartes ne sont plus égales, le joueur qui a la plus forte carte visible remporte les cartes des 2 piles de bataille. Il les met sous son paquet de cartes

6

Exemple

```
Disposer ou distribuer les cartes en 2 paquets
Tant que les paquets 1 et 2 ne sont pas vides
    Prendre la carte du dessus paquet1 (2) et la retourner
    Mettre cette carte sur le dessus de la pile de bataille1 (2)
/* Gestion de la bataille */
Tant que les 2 cartes du dessus des piles de bataille sont égales
    Prendre la carte du paquet1 (2)
    Mettre cette carte sur le dessus de la pile de bataille1 (2)
    Prendre la carte du paquet1 (2) et la retourner
    Mettre cette carte sur le dessus de la pile de bataille1 (2)
/* Gain des cartes */
Tant que les piles de bataille ne sont pas vides
    Prendre la carte sur le dessus de la pile de bataille1 (2)
    La glisser sous le paquet du gagnant
```

Exemple

- **Données et leur description utile**
 - Carte
 - face : visible ou cachée
 - valeur : As, 2, .., 10, .roi
 - Couleur : carreau,trèfle,...
 - Paquet : contient plusieurs cartes, un paquet par joueur
 - suite des cartes
 - Jeu : 52 cartes non triées
 - **Paquet :**
 - Il faut pouvoir glisser dessous et prendre par dessus.
 - **Pile de bataille :**
 - Il faut pouvoir mettre et prendre par dessus.
- ```
struct carte {
 char face; /* Cachée ou non */
 int val;
 char couleur; /* trefle, carreau...*/
}

typedef struct carte CARTE;
```

# Exemple

## Opérations sur ces paquets

- | Prendre .. dessus
  - | Paramètres :
    - | Entrée : paquet,
  - | Valeur de retour :
    - | carte
  - | Rôle
    - | modifie un paquet en supprimant la carte du dessus
    - | Renvoie une carte.
- | Glisser..sous
  - | Paramètres :
    - | Entrée : carte, paquet,
  - | Valeur de retour :
    - | paquet
  - | Rôle,
    - | Ajoute la carte sous le paquet
    - | Retourne le paquet

## Opérations sur ces piles

- | Prendre .. dessus
  - | Paramètres :
    - | Entrée : pile,
  - | Valeur de retour :
    - | carte
  - | Rôle
    - | Enlève la carte du dessus
    - | renvoie cette carte.
- | Mettre dessus
  - | Paramètres :
    - | Entrée : carte, pile
    - | Valeur de retour :
      - | pile
  - | Rôle
    - | ajoute la carte sur le dessus de la pile
    - | renvoie une pile

9

# Exemple

```
#include <stdio.h>
#include "cartes.h"
main(){
 CARTE C1,C2;
 PAQUET J1, J2;
 PILE P1, P2;

 /*creation,initialisation du jeu */
 Distribuer(&J1, &J2);

 do {
 C1=PrendreDessus(&J1);
 Retourner(&C1);
 P1=MettreDessus(P1,C1);
 C2=PrendreDessus(&J2);
 Retourner(&C2);
 P2=MettreDessus(P2,C2);
 }
 if(compare(C1,C2)>0){/* J1 gagnant */
 while(Pilenonvide(P1)) {
 C1=PrendreDessus(&P1);
 P1=MettreDessus(P1,C1);
 C2=PrendreDessus(&P2);
 P2=MettreDessus(P2,C2);
 P1=MettreDessous(P1,C2);
 }
 } else /* J2 est le gagnant */
 } while (nonvide(J1) && nonvide(J2));
}
```

10

# Type Abstrait & structures de données



## Liste, files, piles



## Quelques types abstraits

### Pile



### File



### Liste



12

# Notion de type abstrait

- Structures de données
  - Ensemble fini d'éléments dont le nombre n'est pas fixé à l'avance
- Opérations de base:
  - ajouter un élément x à l'ensemble.
  - rechercher si un élément x est élément de l'ensemble.
  - supprimer un élément de l'ensemble.
  - tester si l'ensemble est vide
  - afficher les différents éléments de l'ensemble
- Type abstrait de données
  - séparer la vision abstraite qu'on a des objets de la représentation machine.
  - définir une sémantique indépendamment de la représentation

- Pour l'utilisateur
  - Connaître les propriétés sans savoir comment il est implanté
  - Ne pas être encombré par les détails
  - Ne pas être influencé par des contraintes secondaires
  - Repousser la représentation physique en machine le plus loin possible

13

# Opérations de base

- Opérations de base:
  - initialisation : () → liste
    - création d'une liste vide.
  - ajouter en tête = (élément x liste) → liste
    - tri, plusieurs occurrences ?
  - ajouter : (élément x liste) → liste
    - tri, plusieurs occurrences?
  - rechercher: (élément x liste) → élément
  - est\_vide: (liste) → booléen
  - supprimer: (élément x liste) → liste
- Autres opérations possibles
  - longueur, premier

15

# Listes



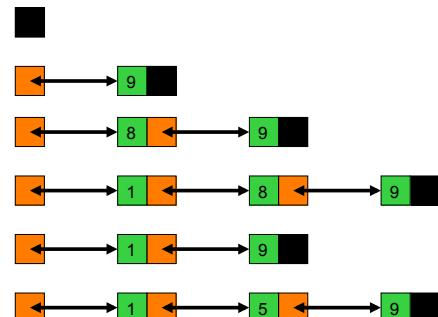
- Ensemble fini mais variable et séquentiel d'éléments
  - chaque élément contient
    - Une valeur
    - L'adresse de l'élément suivant
  - On ne connaît que la tête de liste

- Que veut dire séquentiel ?
  - L'accès à un élément se fait à partir du début (tête), en parcourant les éléments les uns après les autres
  - Comment retrouver l'élément 8 de la liste ci dessous ?



# Listes : Exemple

```
/* listel.c */
main() {
 Liste l;
 l=creerliste();
 visualiser(l);
 l=ajout_tete(9,l);
 visualiser(l);
 l=ajout_tete(8,l);
 visualiser(l);
 l=ajout_tete(1,l);
 visualiser(l);
 l=supprimer(8,l);
 visualiser(l);
 l=ajouter(5,l);
 visualiser(l);
}
```



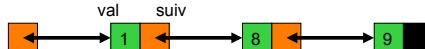
16

# Liste : Implémentation

## 2 Implémentations : comment représenter une adresse

- Pointeur
- Indice dans un tableau

### dynamique par chaînage



### contiguë par chaînage explicite

début

| tab | 0 | 1  | 2 | 3 | 4 | 5 |
|-----|---|----|---|---|---|---|
| 4   |   |    |   |   |   |   |
| 0   | 9 | -1 |   |   |   |   |
| 1   |   |    |   |   |   |   |
| 2   |   |    |   |   |   |   |
| 3   |   |    |   |   |   |   |
| 4   | 1 | 5  |   |   |   |   |
| 5   | 8 | 1  |   |   |   |   |

Tableau de structure avec la valeur (en vert) et l'indice de l'élément suivant dans le tableau.

ET

L'indice de début de la liste

### Quelle implémentation choisir ? Critères de choix

- Coût de l'allocation à chaque insertion
- Nombre d'éléments maximum fixé à l'avance

17

# Rappel sur les structures

## Un chaînon ou maillon d'une liste comporte 2 parties

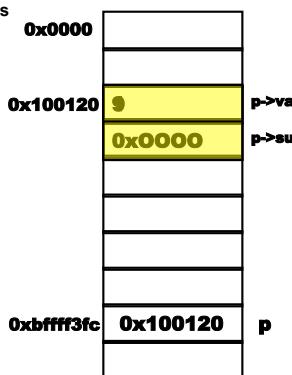
- ce qui doit être stocké dans la liste
- Une information pour indiquer où se trouve le suivant : pointeur ou indice
- On utilise donc une structure pour stocker les 2 infos

```
struct cellule { /* liste0.c */
 double val;
 struct cellule * suiv; } ;
typedef struct cellule Maillon;
typedef Maillon* Liste;

main () {
 Liste p=NULL; /* ou Maillon* p; */

 /* p est un pointeur, pas un maillon ! */
 /* On alloue la mémoire pour un maillon */
 p = malloc(1,sizeof(*p));

 p->val=1; /* On met 1 dans le chainon */
 p-> suiv=NULL; /* NULL dans le suivant */
 p->val=9; /* On change val */
}
```



19

# Listes

## Implémentation dynamique

# Listes dynamiques (1)

### ELEMENT permet de ne pas savoir sur quoi on travaille

### Type à définir :

```
typedef double ELEMENT;
struct cellule {
 ELEMENT val;
 struct cellule * suiv; } ;
typedef struct cellule* Liste;
```

### Utile à redéfinir

#### comparaison de deux ELEMENTS

```
int compare(ELEMENT* e1, ELEMENT* e2);
/* Compare *e1 et *e2 : retourne -1 si
 *e1>*e2, 0 si *e1==*e2, +1 sinon
 */
Affichage d'un ELEMENT
void affiche (ELEMENT* ae);
```

### Fonctions liées au type réel de ELEMENT

```
int compare(ELEMENT* ae1, ELEMENT* ae2)
{ return (*ae1-*ae2); }
void affiche (ELEMENT* ae)
{ printf("%lf ",*ae); }
```

20

# Listes

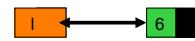
- Chaque maillon de la liste contient 1 valeur et 1 pointeur qui est l'adresse du suivant et tous les maillons ou chaînons sont du même type
- On ne connaît que le début de la liste
- On accède aux autres éléments séquentiellement, en partant du premier, puis en passant au second, puis au troisième, etc..

- Exemple :

- la liste vide



- Une liste avec 1 élément



- Une liste avec 2 éléments



- Une liste vide est représentée un pointeur vide (ou NULL) au départ

- On définit une variable *l*, de type Liste

- On l'initialise avec NULL, qui indique qu'il n'y a pas de suivant

- C'est le pointeur vide qui indique la fin de la liste

21

# Création d'une liste

- Création de liste

```
Liste creer_liste(void)
{ return NULL; /* la liste vide est représentée par NULL */
}
```

- est\_vide :

```
int est_vide(Liste L)
{ return !L; /* la liste vide est représentée par NULL */
}
```

- Exemple

```
main()
{
 Liste L;
 L = creer_liste();
 if (est_vide(L)) printf("la liste est vide\n");
 else printf("la liste n'est pas vide\n");
}
```

22

# Listes : ajout en tête(1)



- Pour ajouter le réel 1 au début de la liste contenant 9, il faut donc

- Créer un nouveau maillon

- Mettre 1 dedans

- Insérer ce nouveau maillon au bon endroit devant tous les autres, c'est à dire

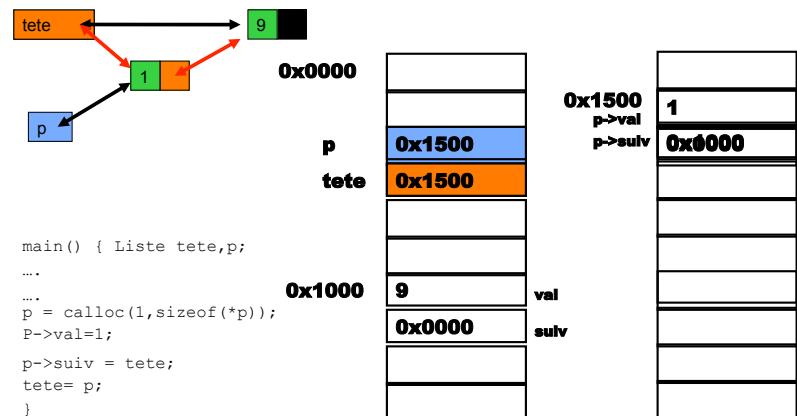
- Indiquer à ce nouveau maillon que son suivant est celui qui contient 9

- Indiquer que ce nouveau maillon est la tête de liste

- ATTENTION à L'ordre des deux dernières étapes

23

# Ajout en tête (2)

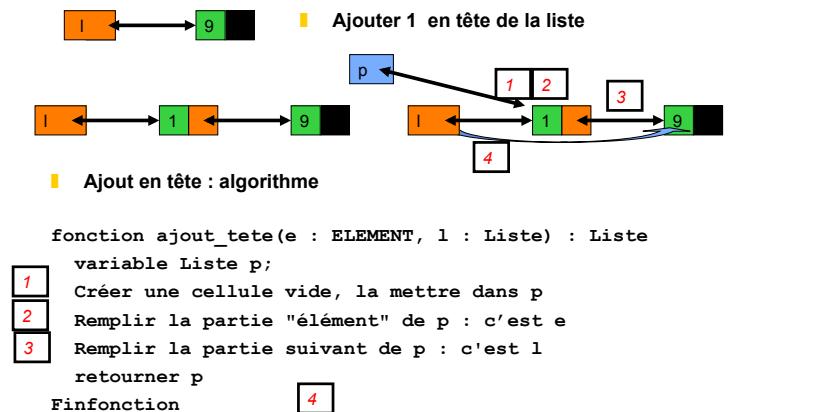


```
main() { Liste tete,p;
...
p = calloc(1,sizeof(*p));
p->val=1;
p->suiv = tete;
tete= p;
}
```

Que se passe t il si vous inversez l'ordre de : tete= p; p->suiv = tete;

24

## Listes : ajout en tête(3)



25

## Listes : ajout en tête(4)

### Ajout en tête : liste3.c

```

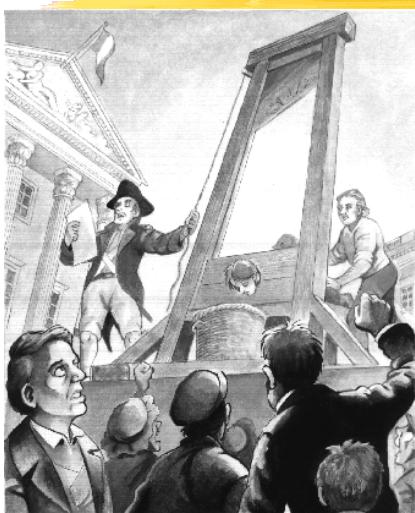
Liste ajout_tete(ELEMENT e, Liste L)
 1 Liste p=calloc(1,sizeof(*p));
 if (p==NULL) return NULL;
 2 p->val=e;
 p->suiv=L; 3
 return p;
}
main() { Liste l;
 l = creer_liste();
 l = ajout_tete(9,l); visualiser(l);
 4 l = ajout_tete(1,l); visualiser(l);

 /* Que fait la ligne suivante ? */
 ajout_tete(0,l); visualiser(l);
}

```

26

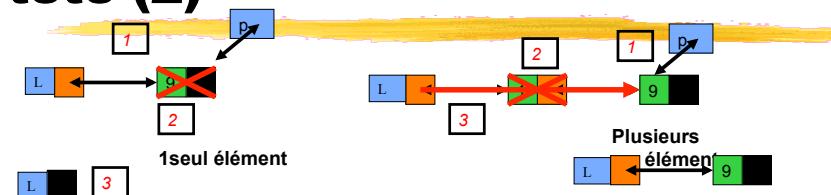
## Suppression en tête (1)



Suppression en tête :  
ne pas perdre la tête  
de liste

27

## Listes : suppression en tête (2)



### Suppression en tête : Algorithme

### libère la mémoire et retourne la tête

```

Fonction supp_tete(Liste L) : Liste
variable p : Liste
 1 sauver dans p la fin de liste
 Libérer la mémoire occupée par L 2
 3 retourner la nouvelle tête p;
}

```

Attention : ne pas libérer la mémoire  
AVANT de sauver le suivant

28

# Listes : suppression en tête (3)

## ■ Suppression en tête : liste5.c

```
Liste supp_tete(Liste L)
{ if (!est_vide(L)) {Liste p;
 p=L->suiv;
 free(L);
 return p;
}
else return NULL;
}

main() { Liste l; l=creerliste();
l=ajout_tete(9,l); l=ajout_tete(8,l);
l=ajout_tete(1,l); l=ajouter(5,l);
visualiser(l);
l = supp_tete(l); visualiser(l);
l = supp_tete(l); visualiser(l);
l = supp_tete(l); visualiser(l);
}
```

```
Terminal — bash — 81x7
5.c
Ordinateur-Desvignes:~/Documents/ens/1AERG/2007/ex1/listes_it desvignes$./a.out
1.000000 5.000000 8.000000 9.000000
5.000000 8.000000 9.000000
8.000000 9.000000
9.000000
Ordinateur-Desvignes:~/Documents/ens/1AERG/2007/ex1/listes_it desvignes$
```

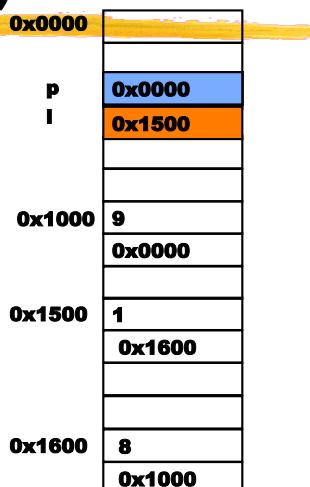
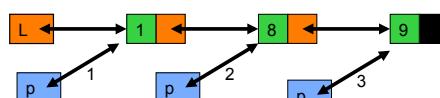
29

# Parcours itératif (1)

## ■ Visualiser itératif

```
void visualiser(Liste L)
{ Liste p;
p=L;
while (!est_vide(p)) {
 affiche(&p->val);
 p=p->suiv;
}
}
```

Affichage écran : 1 8 9



31

# Parcours itératif (1)

## ■ Parcours d'une liste pour visualiser tous les éléments

- Comme pour les tableaux, il faut un indice pour aller de maillon en maillon,
- ici, l'indice est l'adresse de l'élément suivant, i.e. un pointeur p
- Point essentiel : pour passer à l'élément suivant, il suffit que p prenne la valeur suivante  
p = p->suiv

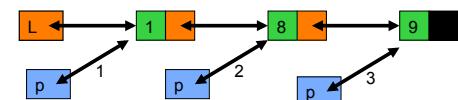
## ■ Algorithme : affiche tous les éléments de la liste

```
procédure visualiser(Entrée l : Liste)
variable p : Liste
p ← l
tant que p n'est pas le dernier
faire
 afficher(p→val)
 p ← élémentsuivant(p)
fintantque
finprocédure
```

## ■ Visualiser itératif

```
void visualiser(Liste L)
{ Liste p;
for (p=L; !est_vide(p); p=p->suiv)
 affiche(&p->val);
}
```

Affiche 1 8 9



30

# Listes triées

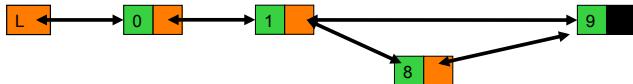
✓ Listes dont les éléments sont en ordre croissant

✓ Ordre établi à l'insertion

✓ Point essentiel : attention à l'ordre dans lequel est fait le chaînage

# Ajouter (1)

- Ajouter le réel 8 à la liste l contenant déjà les réels 0, 1 et 9 dans cet ordre

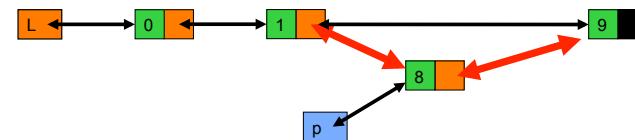


33

# Ajouter (1)

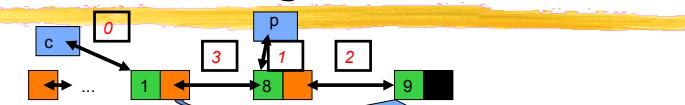
- Pour ajouter 8 à la liste contenant 0,1 et 9, il faut donc

- Créer un nouveau maillon
- Mettre 8 dedans
- Insérer ce nouveau maillon au bon endroit (entre les maillons contenant 1 et 9) : faire les chainages corrects
  - Indiquer à ce nouveau maillon que son suivant est celui qui contient 9
  - Indiquer au maillon contenant 1 que son suivant est le nouveau maillon
  - Ici, la tête de liste l n'est pas modifiée



34

# Listes triées : Ajout



## Algorithme itératif

```

fonction ajouter(element e, Liste L) :
 Liste
 variable c,p : Liste;
 0 Se positionner avec c juste avant le
 point d'insertion ;
 1 Créer un chainon p
 Stocker la valeur e dans la partie
 correspondante de p
 2 Remplir l'adresse du chainon suivant
 p (c'était la partie suivant de c)
 3 Modifier l'adresse du chainon suivant
 c, car c'est 8 qui suit 1 maintenant
 }

Liste ajouter(ELEMENT e, Liste L)
{
 Liste p,c;
 if (est_vide(L) ||
 compare(&e,&L->val)<0)
 return ajout_tete(e,L);
 for(c=L; !est_vide(c->suiv) &&
 compare(&e,&c->suiv->val)>0;
 c=c->suiv) ;
 p= calloc(1,sizeof(Cell));
 p->val=e;
 p->suiv= c->suiv;
 c->suiv=p;
 return L;
}

```

35

# Ajout itératif

## Algorithme itératif

```
#include "liste.h"
```

```

main() {
 Liste l;
 l=creer_liste(); visualiser(l);
 l=ajouter(9,l); visualiser(l);
 l=ajouter(8,l); visualiser(l);
 l=ajouter(1,l); visualiser(l);
 l=ajouter(5,l); visualiser(l);
 l=ajouter(15,l); visualiser(l);
}

```

```

Ordinateur-Desvignes:~/Documents/ens/1AERG/2007/ex1/listes_it desvignes$./a.out
9.000000
8.000000 9.000000
1.000000 8.000000 9.000000
1.000000 5.000000 8.000000 9.000000
1.000000 5.000000 8.000000 9.000000 15.000000
Ordinateur-Desvignes:~/Documents/ens/1AERG/2007/ex1/listes_it desvignes$

```

36

# Listes triées : Ajout récursif

- Algorithme récursif :
  - La fonction ajouter ajoute l'élément e à la liste L et retourne la nouvelle liste ainsi formée
- ```
fonction ajouter(element e, Liste L) : Liste
Si c'est la bonne position (e plus petit que la partie valeur de L)
    créer une nouvelle cellule c
    Stocker la valeur e dans la partie correspondante de c
    Remplir la partie suivant de c (c'est L)
    retourner la nouvelle cellule c
Sinon
    il faut rappeler la fonction ajouter sur la suite de la liste.
    cette fonction retourne la liste formée par l'ajout de e à la suite.
    Cette nouvelle liste est la nouveau suivant de L
    ie Mettre dans la partie suivant de L la liste retournée par
        l'insertion de e dans la suite de la liste
    retourner L, puisque ajouter doit retourner la nouvelle liste formée
    par l'ajout de e à L
```

37

Listes triées : Ajout (2)

- Ajouter récursif :
- ```
Liste ajouter(ELEMENT e, Liste L)
{ if (est_vide(L) || compare(&e, &L->val)<0)
 return ajout_tete(e,L);
else {
 /*ajouter qqpart apres : la fonction nous renvoie la
 liste obtenue en ajoutant e dans le reste de la liste */
 L->suiv=ajouter(e,L->suiv);
 return L;
}
main() { Liste l;
l=creerliste();
l = ajout_tete(9,l);
l = ajout_tete(1,l);
l = ajouter(8,l);
}
```

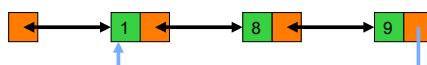
38

# Autres listes

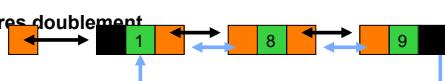
- Listes doublent chaînées
  - permettent le retour en arrière
  - la valeur de l'élément
  - l'adresse du suivant
  - l'adresse du précédent



- Listes circulaires
  - la fin de liste pointe sur le début de la liste



- Listes circulaires doublent chaînées



39

# Listes : Exercices

- Faire une fonction qui recherche si un élément (un réel) est présent dans une liste : cette fonction retourne le maillon de la liste contenant la valeur si elle est présente ou NULL sinon
- Faire une fonction qui fusionne deux listes
- Faire une fonction qui insère en tête dans une liste doublement chainée
- Faire une fonction qui insère en queue dans une liste doublement chainée
- Faire une fonction qui insère en tête dans une liste circulaire simplement chainée
- Faire une fonction qui ajoute un réel dans une liste triée et doublement chainée de réels.
- Faire une fonction qui prend une liste en paramètre, et crée une nouvelle liste contenant les éléments en ordre inverse. A t on besoin de connaître la nature des éléments de la liste ?

40

## Pour en savoir plus

- Vision récursive des listes, piles et files
- Parcours génériques

41

## Parcours récursif

- Visualiser récursif : on considère une liste l comme :

- Un premier élément (pointé par l)
- Suivi d'une liste plus courte de un élément pointée par l->suiv

- Pour visualiser toute la liste

- Cas général :

- afficher le contenu du maillon
- Recommencer sur le reste de la liste

- Cas particulier

- La liste l est vide

```
procédure visualiser(Entrée l : Liste)
```

```
 si l n'est pas vide faire
```

```
 afficher(l->val)
```

```
 visualiser le reste de la liste qui est donné par l->suiv
```

```
finsi
```

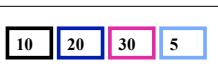
```
finprocédure
```

42

## Parcours récursif (2)

### Visualiser récursif

```
void visualiser(Liste L){
 if(!est_vide(L)){affiche(L->val); visualiser(L->suiv);} }
main() { Liste L;
L=creer_liste(); L=ajouter(5,L); L=ajouter(30,L); ... ; visualiser(L); }
```



## Parcours et action

- void action\_liste(Liste L) : réalise une action sur tous les éléments de la liste L, sans modifier la structure de la liste

- Exemple d'action : afficher, recherche

### Version itérative

```
void action_liste_it(Liste L) {
 for(p=L; !est_vide(p); p=p->suiv)
 action(p);
}
```

### Version récursive

```
void action_liste_rec(Liste L) {
 if(!est_vide(L)) {
 action(p);
 action_liste_rec(L->suiv);
 }
}
```

44

# Parcours et modification

- Liste action\_liste(Liste L) : réalise une action sur tous les éléments de la liste L, sans modifier la structure de la liste
- Exemple : ajouter, supprimer

## Version itérative

```
Liste action_liste(Liste L) {
 for(p=L; !est_vide(p); p=p->suiv) action(p);
 return L; /* Ou la nouvelle tête de liste si action l'a modifiée
}
```

## Version récursive

```
Liste action_liste(Liste L) {
 if(!est_vide(L)) { action(p); L->suiv = action_liste(L->suiv); }
 return L;
}
```

45

# Piles

## Analogie : pile d'assiettes

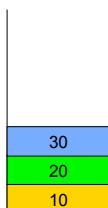
- Seul le sommet de la pile est accessible
- 2 opérations : empiler ou dépiler
- LIFO : Last In First Out

## Propriété

- Le dernier arrivé est le premier servi !!

## Exemple:

- Empiler 10
- Empiler 20
- Empiler 30
- Dépiler



# Piles

# Piles

```
#include "pile.h" »

main() { Pile p;
ELEMENT e;
p=creer_pile();
p=empiler(10,p); visualiser(p);
p=empiler(20,p); visualiser(p);
p=empiler(30,p); visualiser(p);
e= depiler(&p);
printf("Depiler: valeur de l'element depile %lf\n",e);
visualiser(p);
}
```



47

48

# Piles

## Opérations

- **creer\_pile** : → pile
  - initialisation.
- **empiler**: élément x pile → pile
  - ajout d'un élément au sommet i.e. en tête
- **sommet** : pile → élément
  - valeur de l'élément de tête
- **supprimer** : pile → pile
  - suppression de l'élément situé en tête.
- **pile\_vide** : pile → booléen
- **dépiler** : pile → élément x pile
  - Retourne la valeur en tête et suppression de cette valeur de la liste

49

# Piles : représentation chaînée

```
typedef double ELEMENT;
typedef struct cellule {
 ELEMENT val;
 struct cellule * suiv;} Maillon, * Pile;

/* declarations */
/* ATTENTION: plusieurs de ces fonctions n'ont pas de sens avec une pile
vide */

Pile creer_pile(void);
int pile_vide(Pile p);
Pile empiler(ELEMENT,Pile);
Pile supprimer(Pile);
ELEMENT sommet(Pile);
ELEMENT depiler(Pile*);

/* ATTENTION : depiler doit réaliser 2 actions : retourner l'élément
au sommet de pile et modifier donc le sommet de pile. C'est pour cela
que f est la pile est passée par adresse.
De plus, il est obligatoire que cette pile ne soit pas vide
*/
```

50

# Files

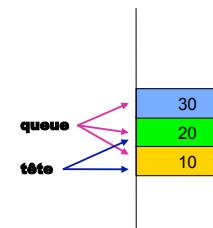
# Files

- Analogie : file d'attente du RU (mais pas celui de minatec !!)
  - insertion des éléments en queue
  - retrait des éléments en tête

- Terminologie :
  - Enfiler 10
  - Enfiler 20
  - Enfiler 30
  - defiler

- Propriété
  - First In First Out
  - Premier arrivé, premier servi

- Terminologie :
  - tête de file
  - queue de file



52

# Files

## Opérations

- file\_vide : file -> file
  - Création d'une file vide
- enfiler : élément x file -> file
  - ajout en queue dans une liste
- tête : file -> élément
  - valeur de la tête de la liste
- supprimer : file -> file
  - supprime "en tête" de la liste.
- est\_vide : file -> booléen
- défiler : file -> élément x file
  - retourne l'élément en tête puis suppression de l'élément.

53

# Files : représentation chaînée

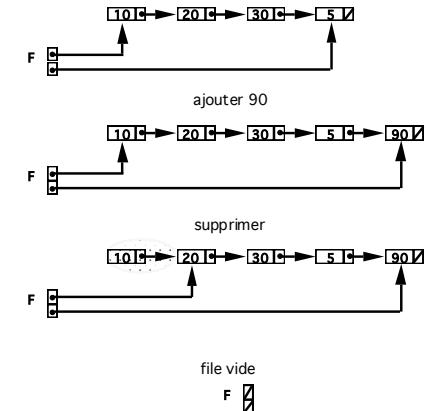
## doublet tête-queue de pointeurs vers une cellule.

- adresse de la tête de la file
- adresse de la queue de la file.

## File vide = doublet NULL-NUL

## Liste classique pour le reste

```
typedef int ELEMENT;
typedef struct cellule {
 ELEMENT val;
 struct cellule * suiv; }* Liste;
typedef struct {
 Liste tete, queue; } File;
```



54

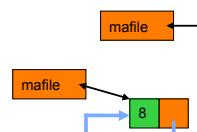
# Files : représentation chaînée astucieuse

- File : définie par un seul pointeur sur la QUEUE avec une Liste CIRCULAIRE
  - A la fin de la liste, on ajoute un lien qui ramène au début de la liste
- La tête est définie par l'element qui suit la queue : tête = liste-> suiv
- La queue est définie par la liste: queue= liste

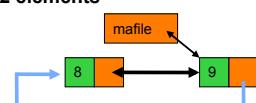
## File vide : file sans aucun élément

```
typedef int ELEMENT;
typedef struct cellule {
 ELEMENT val;
 struct cellule * suiv; } Maillon;
typedef Maillon* File;
File mafile;
```

## File avec 1 élément



## File avec 2 éléments



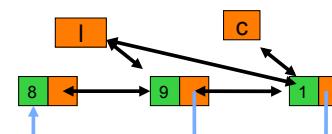
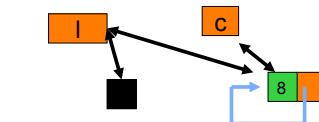
55

# Files : représentation chaînée astucieuse

```
File enfiler(ELEMENT e, File f) {
 File c = malloc(1,sizeof(*c));
 c->val = e;
 if (est_vide(f)) c-> suiv = c;
 else {
 c-> suiv = f-> suiv; f-> suiv=c;
 }
 return c;
}
```

```
main() { File l;
 l= creer_file();
 /* Enfiler 8 sur une file vide */
 l=enfiler(8,l);

 l=enfiler(9,l);
 l=enfiler(1,l);
}
```



56

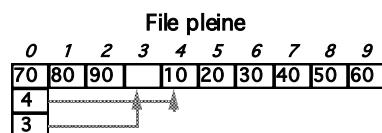
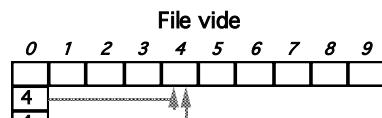
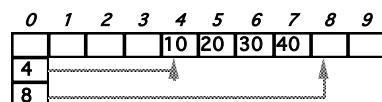
# Représentation contiguë des files et des piles

- structure:
  - un tableau représente les valeurs
  - un entier représente la position de la tête de la File
  - un entier représente la position de la queue de la File

- le tableau est exploité de façon circulaire

```
#define LONG_FILE 10
```

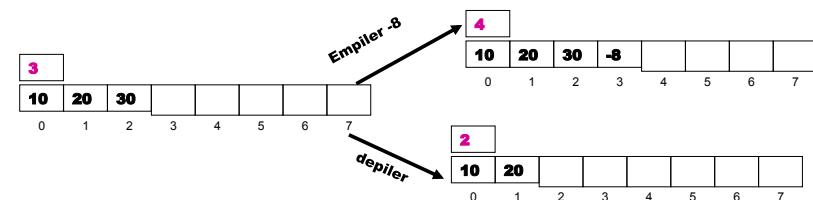
```
typedef int ELEMENT;
typedef struct {
 int tete;
 int queue;
 ELEMENT tab[LONG_FILE];}* File;
```



# Piles : représentation contiguë

- Pile représentée par
  - un tableau
  - un indice représentant le sommet
- Les deux éléments sont inclus dans une structure
- Attention à la capacité du tableau
- Pile initiale

```
#define LONGPILE 25
typedef double ELEMENT;
typedef struct { int nbre;
ELEMENT tab[LONGPILE];}* Pile;
```



# Files : représentation contiguë

```
/* les dépassements de capacité ne sont pas traités */
File creer_file(void) { File f=calloc(1,siozeof(*f));
 f->tete=f->queue=0;
 return f;
}

File enfileur(ELEMENT e,File f) {
 if (!(file_pleine(f))) {
 f->tete=f->queue=e;
 f->queue=(f->queue+1)%LONG_FILE;
 }
 else /* erreur : dépassement de capacité */
 return f;
}

File supprimer(File p) { p->tete=(p->tete+1)%LONG_FILE; return p; }
```

# Files : représentation contiguë

```
ELEMENT tete(File p) { return p->tab[p->tete]; }

ELEMENT defiler(File f) { ELEMENT e;
 if (!file_vide(f)) {
 e=f->tab[f->tete]; f=supprimer(f);
 }
 return e;
}

int file_pleine(File f) {return (f->queue+1)%LONG_FILE==f->tete; }

void visualiser(File f) { int i;
 for (i=f->tete; i!=f->queue; i=(i+1)%LONG_FILE)
 printf("%lf ",f->tab[i]);
}
```

61

# Files, Piles : exercices

- Ex 1 : Piles par listes chaînées : Écrire les fonctions empiler, depiler.
- Ex 2 : Piles par listes chaînées : Comment écrit on les fonctions empiler, depiler en utilisant les fonctions ajout\_tete et supp\_tete sur les listes chaînées.
- Ex 3 : Files par listes chaînées avec pointeur tête-queue : Écrire les fonctions enfiler, défiler
- Ex 4 : Files par listes chaînées circulaires : Écrire la fonction défiler
- Ex 5: Comparer la complexité de l'ajout/suppression d'un élément dans une file dans le cas d'une représentation par liste classique et dans le cas d'une liste circulaire. Dans, ce dernier cas, pourquoi l'entrée se situe-t-elle à la fin de la file et non au début ?
- Ex 5 : Piles par tableau : Écrire les fonctions empiler et depiler.

62

# Dictionnaires



# Dictionnaire ou table

- Un dictionnaire est une structure de données gardant en mémoire des éléments de la forme (clé, valeur)
  - Un dictionnaire de français contient des mots (les clés) et leurs définitions (les valeurs)
- Le but d'un dictionnaire est d'être capable d'accéder rapidement à une valeur, en donnant sa clé.
- Exemple de problèmes posés :
  - Vérifier l'orthographe d'un texte, Trouver la définition ou la traduction d'un mot
    - Clé : un mot – Valeur : présence, définition, traduction
    - Dictionnaire des mots connus dans une langue dans un fichier (N mots, N grand)
    - Vérifier un mot = rechercher s'il est présent dans le dictionnaire
  - Annuaire inversé : trouver le nom d'un individu à partir de son numéro de téléphone
    - Clé : un numéro (un entier ?) – Valeur : nom et adresse
    - Numéro à 10 chiffres (M=10<sup>10</sup> numéros. Peut-on faire un tableau de M positions ?

64

# Dictionnaire

- Quelles opérations peut on faire sur un dictionnaire ?
  - Insérer un élément dans le dictionnaire
  - Supprimer un élément dans le dictionnaire
  - Rechercher un élément dans le dictionnaire
  - Et aussi : énumérer les éléments, fusionner 2 dictionnaires
  
- Quelles structures de données peut on utiliser ?
  - Tableaux et tableaux triés
  - Listes
  - Table de hachage
  - Arbres binaires
  - Arbres lexicographiques

65

# Solution séquentielle

- Solution séquentielle
  - Charger le fichier en mémoire dans un tableau ou une liste de N éléments (des mots en TD)
  - Comparer la clé de l'élément à celle du premier élément du dictionnaire.
  - Si elles sont identiques, arrêt de la vérification
  - Sinon on passe à l'élément suivant
  
- Nombre de comparaisons effectuées pour rechercher un élément dans
  - Le pire des cas (le mot n'existe pas dans le dictionnaire) ?
  - En moyenne (en supposant une équi-repartition des mots dans la langue) ?

66

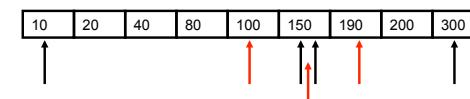
# Dichotomie

- Principe : dans un tableau trié, tous les éléments avant le milieu du tableau sont plus petit que l'élément du milieu, ceux placés ensuite sont plus grands
  - Charger le fichier en mémoire dans un tableau de N éléments
  - Trier le tableau selon leur clé.
  
- Le tableau contient des éléments triés selon les clés entre les indices  $i_1=0$  et  $i_2=N-1$
- Comparer la clé de l'élément à celle de l'élément qui est au milieu du dictionnaire, ie en  $i_m=(i_1+i_2)/2$
- S'il est identique, arrêt de la vérification
- Sinon deux cas
  - Soit la clé de l'élément recherché est plus grand que celle de l'élément en  $i_m$ 
    - Alors on recommence la même chose entre les indices  $i_1=i_m+1$  et  $i_2=i_2$
    - Sinon on recommence la même chose entre les indices  $i_1=i_1$  et  $i_2=i_m-1$
  
- Remarque : dans les exemples ci dessous, les éléments ne contiennent que la clé qui est un entier. Pour un dictionnaire classique, la clé est un mot

67

# Dichotomie

- Tab = 10 20 40 80 100 150 190 200 300,
- Recherche de 150
- Initialement  $i_1=0$ ,  $i_2=8$ ;  $i_m=4$

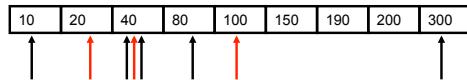


- $i_m=(0+8)/2=4$  : Comparaison du nb recherché 150 et de tab[im] (100)  
150 > 100 donc, on recommence avec  $i_1=im+1=5$ ,  $i_2=8$ ;
- $i_m=(5+8)/2=6$  : Comparaison du nb recherché 150 et de tab[im] (190)  
150 < 190 donc, on recommence avec  $i_1=5$ ,  $i_2=im-1=5$
- $i_m=(5+5)/2=5$  : Comparaison du nb recherché 150 et de tab[im] (150)  
150 == 150 donc, on a trouvé 150 dans le tableau

68

## Dichotomie

- Tab = 10 20 40 80 100 150 190 200 300, recherche de 30
- Initialement i1=0, i2=8; im=4



- Comparaison du nb recherché 30 et de tab[im] (100)  
30 < 100 donc, on recommence avec i1=0, i2=im-1=3;
- im vaut 1; Comparaison du nb recherché 30 et de tab[im] (20)  
30 > 20 donc, on recommence avec i1=im+1=2, i2=3;
- im vaut 2 : Comparaison du nb recherché 30 et de tab[im] (40)  
30 < 40 donc, on recommence avec i1=2, i2=im-1=2;
- Mais comme l'intervalle entre i1 et i2 est nul, cela veut dire qu'il faut s'arrêter

### Nombre de comparaisons effectuées dans

- Le pire des cas (le nombre ou le mot n'existe pas dans le dictionnaire) ?
- En moyenne (en supposant une équpartition des mots dans la langue) ?

69

70

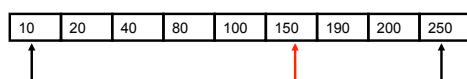
## Recherche par interpolation

- Principe: on suppose qu'il existe une relation entre la valeur d'un élément et l'endroit où il est rangé en mémoire

- Charger le fichier en mémoire dans un tableau de N éléments
- Trier le tableau.
- Le tableau contient des éléments triés par clés entre les indices  $i_1=0$  et  $i_2=N-1$
- Comparer la clé X de l'élément recherché à celle de l'élément qui en  $i_m=i_1 + (X-tableau[i_1]) / (tableau[i_2]-tableau[i_1]) * (i_2-i_1)$
- S'il est identique, arrêt de la vérification
- Si non deux cas
  - Soit la clé de l'élément recherché est plus grande que celle de l'élément en  $i_m$ 
    - Alors on recommence la même chose entre les indices  $i_1=i_m+1$  et  $i_2=i_2$
    - Sinon on recommence la même chose entre les indices  $i_1=i_1$  et  $i_2=i_m-1$

## Interpolation

- Tab = 10 20 40 80 100 150 190 200 300,
- Recherche de 150
- Initialement i1=0, i2=8; im=0 + (150-10)/(250-10)\*(8-0) = 4,66

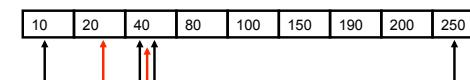


- Comparaison du nb recherché 150 et de tab[im] avec im=5  
150 == 100 donc, on a trouvé 150 dans le tableau

71

## Interpolation

- Tab = 10 20 40 80 100 150 190 200 300, recherche de 30
- Initialement i1=0, i2=8; im=0 + (30-10)/(250-10)\*(8-0) = 0,66



- Comparaison du nb recherché 30 et de tab[im] = 20  
30 > 20 donc, on recommence avec i1=im+1=2, i2=8;
- im = 2+(30-40)/(250-40)\*(8-2)=1,7 : Comparaison du nb recherché 30 et de tab[im] (40)  
30 < 40, on recommence avec i1=2, i2=im-1=2 : l'intervalle est nul
- Mais comme l'intervalle entre i1 et i2 est nul, cela veut dire qu'il faut s'arrêter

### Nombre de comparaisons effectuées dans

- Le pire des cas (le nombre ou le mot n'existe pas dans le dictionnaire) ?
- En moyenne (en supposant une équpartition des mots dans la langue) ?

72

## Hachage



## Table de hachage (2)

|      |      |                              |
|------|------|------------------------------|
| 0    |      |                              |
| 1    |      |                              |
| .... |      |                              |
| 6    | bac  | examen passé en terminale    |
| ...  |      |                              |
| 31   | lame | partie coupante d'un couteau |
| ...  |      |                              |
| 99   |      |                              |

- Utiliser le dictionnaire : recherche du mot xxxx dans le dico
  - Calculer hash(xxxx) qui donne l'indice n dans le tableau où doit se trouver le mot xxxx
  - Si tableau[n] contient xxxx, alors le mot (et sa définition) est trouvé
  - Sinon le mot n'existe pas dans le dictionnaire

## Table de hachage

### Solution : l'adressage calculé ou hachage

- accès à l'élément à l'aide d'une clé : dans le cas du dictionnaire, la clé est le mot dont on recherche la définition.
- La clé est utilisée pour calculer un indice entier, qui donne la position du mot dans le tableau
- Ce calcul est fait par une fonction de hachage, application de l'ensemble des clés vers les entiers
- accès en temps constant, ne dépend pas du nombre d'éléments stockés

### Exemple : Construire le dictionnaire dans un tableau de N cases

- Clés : mots i.e. une chaîne de caractères
- fonction de hachage :hash(mot) : somme des lettres de la chaîne modulo N
- Clé : Lame ; Définition : partie coupante d'un couteau  
$$\text{hash(lame)} = (12 + 1 + 13 + 5) \% N = 31 \% N = 31 \quad (N=100).$$

On mettra lame et sa définition à l'indice 31 de notre tableau
- Clé : Bac ; Définition : examen passé en terminale  
$$\text{hash(bac)} = (2+1+3)\%N = 6$$

On mettra bac et sa définition à l'indice 6 de notre tableau

74

## Table de hachage (3)

### Problème : les collisions

- Où mettre le mot male avec notre fonction de hachage ?  
$$\text{hash(lame)} = \text{hash(male)}$$
- Une collision survient lorsque 2 clés ont la même valeur de hachage

### Gestion des collisions 2 solutions :

- Utiliser des listes pour stocker les mots qui collisionnent
  - Dynamique avec pointeurs et allocation : hachage externe
  - Statique dans le tableau : hachage interne coalescent (par chaînage)
- Utiliser un deuxième calcul :
  - Prendre la case qui suit si elle est libre : hachage linéaire
  - Utiliser une deuxième fonction de hachage différente de la première qui doit séparer et donner des indices différents : il est peu probable que deux mots aient deux valeurs de hachage identiques par deux fonctions distinctes: hachage double

76

# Les collisions existent elles ?

- Avec une distribution uniforme, il y a 95% de chances d'avoir une collision dans une table de taille 1000 avant qu'elle ne contienne 77 éléments
- On est presque sur qu'il y aura eu une collision après l'introduction du 77ième élément
  
- C'est le paradoxe des anniversaires : combien mettre de personnes dans une pièce sans que 2 d'entre elle ne soient nées le même jour ?
  - Pour que personne n'ait le même jour d'anniversaire, il y a  $365 \times 364 \times \dots \times (365-n+1)$
  - Le nombre total de possibilités est  $365^n$
  - Donc la probabilité de n'avoir aucune personne avec la même date est  $A_{365}^n / 365^n$
  - et celle que nous ayons 2 personnes avec la même date est :  $1 - A_{365}^n / 365^n$
  - Si on fixe la probabilité  $p$ , le nombre  $n$  de personnes est donné par l'approximation

$$n(p) \approx \sqrt{2 \cdot 365 \ln \left( \frac{1}{1-p} \right)}$$

77

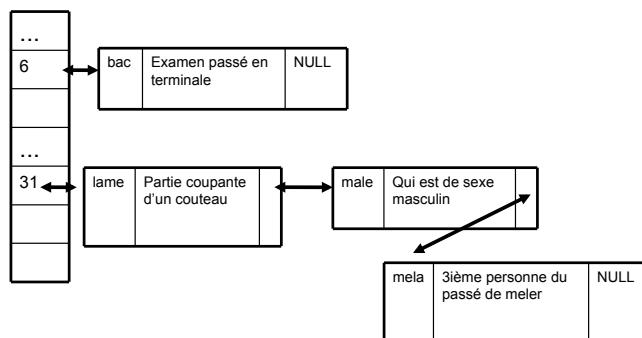
# Fonctions de hachage

- Une bonne fonction de hachage
  - Évite un grand nombre de collisions
  - remplissage uniforme de la table
  - rapide à calculer
  
- Considérons une table de taille N (de préférence premier) et un nombre d'entrées M
  - Clés numériques
    - Entiers :  $\text{hash}(k) = k \bmod N$
    - réels :  $\text{hash}(k) = \text{Partie entière de } k/N$
  - Clés alphanumériques ou chaîne de caractères :  $c_0c_1c_2c_3\dots c_k$ 
    - Première fonction
      - $\text{hash}(k) = (c_0 + a*c_1 + a^2*c_2 + a^3*c_3\dots + a^k*c_k) \bmod N$
      - $a$  est quelconque, éviter les puissances de 2 , la table de hachage de JAVA prend  $a = 31$
    - Fonction universelle
      - $a = 31415; b=27183$  ( $a$  et  $b$  premier entre eux)
      - $\text{hash}(k) = c_0 + (a*b)\%(N-1)*c_1 + (a^2*b^2)\%(N-1)*c_2 + (a^3*b^3)\%(N-1)*c_3\dots + (a^k*b^k)\%(N-1)*c_k \bmod N$
      - probabilité de conflits en  $1/N$

78

# Hachage par listes

- Listes chaînées (à l'extérieur du tableau)
  - toutes les clés ayant même hachage sont chaînées entre elles
  - nombre moyen d'accès en recherche :  $1+N/M$



79

# Hachage coalescent

- Principe : le tableau est coupé en deux :
  - Une partie initiale, où on met les premières clés et leurs définitions
  - Une partie réserve, qui sert à chaîner les collisions
  - Chaque élément du tableau contient la clé, l'information utile, et l'indice du suivant
- Construction et recherche
  - Si la fonction de hachage donne un indice où il n'y a pas d'élément, on le stocke à cet endroit
  - S'il y a déjà un élément, on cherche la première case libre dans la zone réserve.

- Exemple avec réserve pour les collisions à partir de l'indice 80 dans un tableau de 100 éléments

- ajout dans l'ordre
  - bac : h=6
  - lame : h = 31
  - male : h = 31
  - cab : h=6
  - mela : h = 31
- Recherche de male
- Recherche de elam

|    |      |    |
|----|------|----|
| 0  |      | -1 |
| .. |      | -1 |
| 6  | bac  | 98 |
| .. |      | -1 |
| 31 | lame | 99 |
| .. |      | -1 |
| 97 | mela | -1 |
| 98 | cab  | -1 |
| 99 | male | 97 |

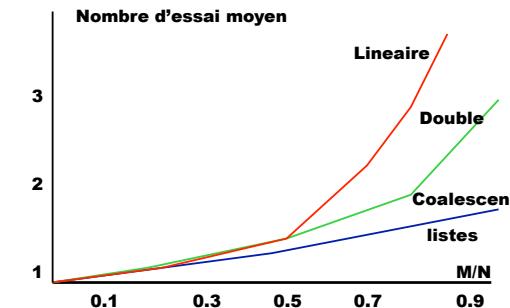
80

## Collisions par recalcul

- Principe : On calcule une nouvelle position par une nouvelle fonction de hachage
- Hachage linéaire
  - on prend le suivant, puis le suivant, puis....
  - La fonction est donc :  $f(\text{mot}) = \text{hash}(\text{mot}) + (i) \% N$   
où  $i$  est le nombre de fois qu'il y a des collisions pour mot
- Double hachage :
  - on prend une autre fonction de hachage  $d(c)$  :  $f(\text{mot}) = \text{hash}(\text{mot}) + d(\text{mot}) * (i) \% N$

81

## Comparaison



82

## Comparaison

- Hachage par listes
  - simple
  - allocation de mémoire dynamique pour la gestion des collisions
  - liens occupant de la mémoire : longueur moyenne d'une liste  $M/N$
  - suppressions simples
  - intéressant pour  $M \gg N$  : plus de clés et d'éléments que d'entrées dans la table de hachage
- Hachage ouvert par listes internes
  - $N$  et  $M$  proches,  $N$  et  $M$  grand,  $N > M$ ,  $N/M \sim 1$
  - recherche en moyenne:
    - 1,8 essais si la clé est présente,
    - 2,1 essais si la clé est absente
- Hachage fermé
  - $N$  et  $M$  proches,  $N$  et  $M$  grand,  $N > M$ ,  $N/M \sim 0,5$
  - recherche en moyenne:
    - 1,5 essais si la clé est présente,
    - 2,5 essais si la clé est absente

83

## Complément sur les pointeurs

### Les pointeurs de fonctions

# Pointeurs de fonctions

- Nom de la fonction
  - adresse de la fonction : on peut donc obtenir l'adresse d'une fonction
- Pointeur de fonction
  - variable contenant l'adresse d'une fonction
- Déclarer un pointeur de fonction
  - type\_de\_retour (\*id\_fonct) (déclaration paramètres);
- Executer la fonction pointée : 2 écritures
  - (\*id\_fonct) (paramètres);  
id\_fonct (paramètres);

85

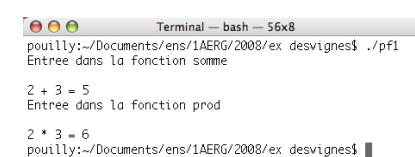
# Pointeurs de fonctions

```
Exemple : pf1.c
/* fonction qui affiche et retourne la
somme de 2 entiers */
int somme (int a,int b){
 printf("Somme %d %d\n",a,b);
 return(a+b);
}

/* fonction qui affiche et retourne le
produit de 2 entiers */
int prod (int a,int b){
 printf("Produit %d %d\n",a,b);
 return(a*b);
}

/* On y met l'adresse de la fonction
somme */
f=somme;
printf("On fait la somme\n");
/* Appel de somme par (*f) */
k = (*f)(i,j); /* ou f(i,j); */
printf("%d + %d = %d\n",i,j,k);

/* On y met l'adresse de la fonction
produit*/
printf("On fait le produit\n");
f=prod;
k = (*f)(i,j);
printf("%d * %d = %d\n",i,j,k);
```



86

# Pointeurs de fonctions (2)

```
Fonctions C standard : TRI : qsort, heapsort, RECHERCHE:bsearch
/* Qsort est définie en standart dans la libC */
qsort(char * base, int nel, int width, int(*comp)(char* x, char* y));
base: Tableau à trier nel: Nombre d'elements à trier
width: Taille des éléments
compar : pointeur vers la fonction qui compare deux éléments de base et
retourne <0, =0, ou >0

int comparaison(float* x, float* y) {return(*x - *y);}

#define N 20
main () {float tab[N]; int i;
for (i=0; i<N; i++) tab[i]=rand()/(double)rand();
for (i=0; i<N; i++) printf("%f ",tab[i]); puts("");
qsort(tab,N,sizeof(float),comparaison);
for (i=0; i<N; i++) printf("%f ",tab[i]); puts("");}
```



# Exercices

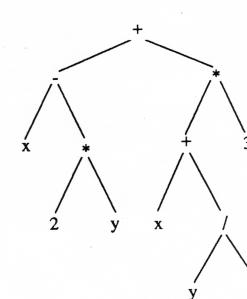
- Ex1 : Ecrire une fonction de hachage pour une clé alphanumérique pour une table de N éléments.
- Ex2 : Ecrire une fonction effectuant la recherche dichotomique d'une chaîne de caractères dans un tableau de chaînes de caractères.
- Ex3 : Ecrire un programme utilisant la fonction bsearch pour rechercher une chaîne de caractères dans un tableau de chaînes de caractères.

88

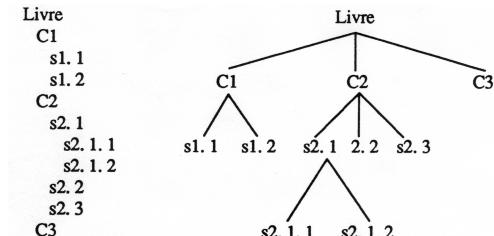
## Arbres

# Arbres explicites : représentation de données

$$(x - (2 * y)) + ((x + (y/z)) * 3)$$

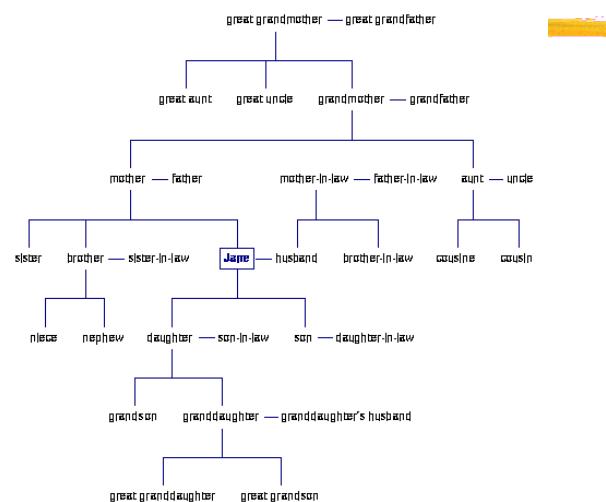


Livre  
C1  
s1.1  
s1.2  
C2  
s2.1  
s2.1.1  
s2.1.2  
s2.2  
s2.3  
C3



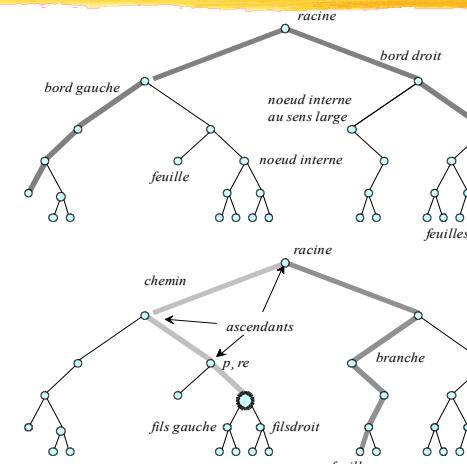
90

## Arbres explicites (2)



91

## Arbres : terminologie



92

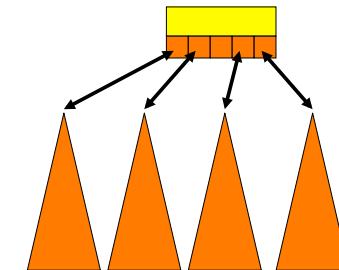
## Arbres : terminologie (2)

- nœud
  - | racine
  - | feuille (=nœud terminal) : aucun fils
  - | nœud interne : au moins un fils
- chemin menant à un noeud
  - | suite des noeuds menant de la racine au noeud concerné
  - | branche -- chemin de la racine à une feuille
- père
  - | les fils
  - | le fils et ses frères
- étiquette d'un nœud
- Propriétés essentielles
  - | racine = aucun père
  - | autres nœuds = un seul père.
  - | *chemin unique conduisant de la racine à un nœud*
- Mesures
  - | taille : nombre de nœuds
  - | degré d'un nœud : nombre de fils
  - | longueur d'une branche : nombre de noeud sur la branche
  - | hauteur / profondeur / niveau d'un nœud : nombre de noeud sur le chemin
    - | racine : 0
  - | hauteur / profondeur d'un arbre : hauteur de la plus grande branche
  - | génération : noeuds du même niveau

93

## Arbres : modèle abstrait

- Ensemble fini de nœuds organisés de façon hiérarchique à partir d'un nœud particulier: la racine
  - | soit vide,
  - | soit formé d'un nœud et d'une suite d'arbres : ses fils
- Opérations
  - | Créer un arbre
  - | Arbre vide ?
  - | Ajouter un élément
  - | Enlever un élément
- Parcours et utilisation
  - | Parcours en profondeur:
    - | on visite les descendants d'un nœud avant de visiter ses frères
  - | Parcours en largeur:
    - | on visite les frères d'un nœud avant de visiter ses fils



94

## AB: modèle abstrait (1)

- Arbre binaire
  - | soit vide
  - | soit un triplet  $\langle o, B1, B2 \rangle$  où  $B1$  et  $B2$  sont deux arbres binaires disjoints.
  - | chaque nœud a, au plus, 2 fils
- Arbre binaire de recherche
  - | étiquettes sont ordonnées:
    - | Les étiquettes du sous-arbre gauche sont inférieures à l'étiquette de la racine
    - | les étiquettes du sous-arbre droit sont supérieures à l'étiquette de la racine
- AB Parfait : arbre dont tous les niveaux sont complètement remplis sauf le dernier.
- AB Complet : AB Parfait + dernier niveau rempli
  - | → Représentation par tableau facile : Chaque noeud a des fils aux indices  $2i$  et  $2i+1$ .

95

## AB: modèle abstrait (2)

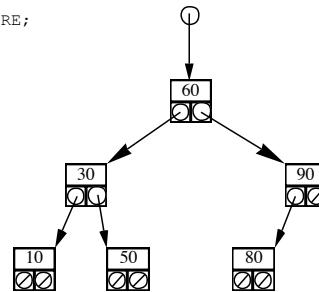
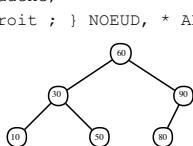
- arbre\_vide :  $\rightarrow$  arbre
  - | retourne un arbre vide.
- ajouter: élément  $x$  arbre  $\rightarrow$  arbre
  - | ajout d'une feuille ou à la racine.
- supprimer: élément  $x$  arbre  $\rightarrow$  arbre
- rechercher: élément  $x$  arbre  $\rightarrow$  arbre
  - | retourne le sous-arbre dont la racine contient l'élément
- Visualisation
  - | parcours: arbre  $\rightarrow$
  - | visualise les étiquettes dans l'ordre

96

## AB: représentation

- Représentation chaînée :

```
typedef int etiquette;
typedef struct noeud {
 etiquette val;
 struct noeud * gauche;
 struct noeud * droit ; } NOEUD, * ARBRE;
```



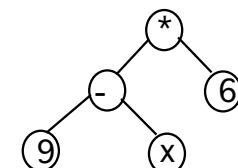
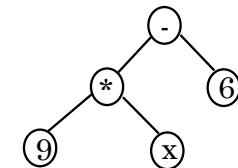
97

## AB: Exemple

- Expressions arithmétiques binaires

  - On utilise la notation préfixée :  
opérateur opérande1 opérande2

  - $9*x*6$
  - $(9-x)*6$

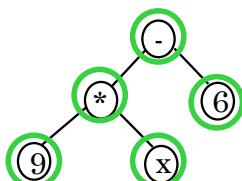


98

## AB: Parcours sans modifications de l'arbre

- Parcours en profondeur

```
préfixé ou RGD : Racine Gauche Droite
void parcours_RGD(NOEUD *p){
 if (p){ action(p);
 parcours_RGD(p->gauche);
 parcours_RGD(p->droit);
 }
}
void affiche_RGD(NOEUD *p){
 if (p){ printf(" %c ",p->val);
 affiche_RGD(p->gauche);
 affiche_RGD(p->droit);
 }
}
```



- \* 9 x 6

```
affiche_RGD('-')
printf('-');
affiche_RGD('*'); // Affiche le fils gauche de -
printf('*');
affiche_RGD('9'); // Affiche le fils gauche de *
printf('9');
pas de fils, donc fin de affiche_RGD('9');
affiche_RGD('X'); // Affiche le fils droit de *
affiche_RGD('6'); // Affiche le fils droit de -
```

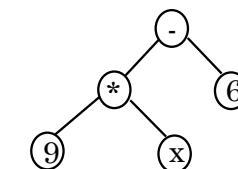
99

## AB: Parcours sans modifications de l'arbre

- Parcours en profondeur

```
infixé ou GRD : Gauche Racine Droite
void parcours_GRD(NOEUD *p){
 if (p) {
 parcours_GRD(p->gauche);
 action(p);
 parcours_GRD(p->droit);
 }
}
```

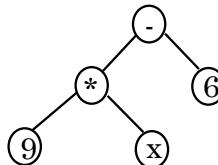
```
void affiche_GRD(NOEUD *p) {
 if (p) {
 affiche_GRD(p->gauche);
 printf(" %c ",p->val);
 affiche_GRD(p->droit);
 }
}
```



100

# AB: Parcours sans modifications de l'arbre

- Parcours en profondeur
  - postfixé ou GDR : Gauche Droite Racine
- void parcours\_GDR(NOEUD \*p)  
{if (p)  
 {parcours\_GDR(p->gauche);  
 parcours\_GDR(p->droit);  
 action(p);  
 }  
}
- Parcours en largeur
  - frères, fils des frères, fils des fils ...
  - utilise une file pour être réalisé
  - Algorithme
    - ajouter la racine de l'arbre dans la file
    - tant que la file n'est pas vide faire:
      - défiler
      - action à réaliser
      - enfiler le fils gauche
      - enfiler le fils droit
    - fin tant que



101

# AB: Parcours avec modification de l'arbre

- Attention, si on doit modifier l'arbre, il faut, comme pour les listes écrire :

```
ARBRE parcours_et_modification(ARBRE p) {
 if (p) {
 /* Je fais ce que je dois faire sur la valeur de ma racine */
 p->val=action(p);

 /* Si besoin, Je fais ce que je dois faire sur le fils gauche :
 Je récupère le nouvel arbre et je change le fils gauche avec
 le nouvel arbre */
 p->gauche=parcours_et_modification (p->gauche);

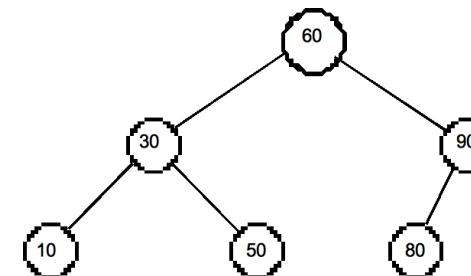
 /* Idem pour le droit */
 p->droit=parcours_et_modification (p->droit);
 }
 else {
 /* Cas où l'arbre est vide : Peut être y a t il qqchose à faire ici*/
 }
 return p;
}
```

102

# Arbres binaires de recherche

## ABR

- ABR : Arbre binaire de Recherche : noeuds ordonnés
  - Les étiquettes du sous-arbre gauche sont inférieures à l'étiquette de la racine
  - les étiquettes du sous-arbre droit sont supérieures à l'étiquette de la racine



104

# ABR

## Arbre vide ?

```
int arbre_vide(arbre a){ return a==NULL; }
```

## Création d'un arbre

```
arbre creer_arbre(void){ return NULL; }
```

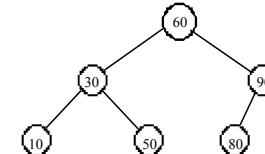
## Rechercher : quelle est la complexité ?

```
arbre rechercher(etiquette val, arbre a)
/* retourne le nœud qui contient la valeur recherchée */
/* retourne un arbre vide si l'élément est absent */
/* en cas de synonymes, retourne la position de la 1ere occurrence */
{
 if (arbre_vide(a)) return NULL; /* l'arbre est vide */
 else if (val==a->val) return a; /* trouvé : a contient val */
 /* sinon essayer à gauche à droite selon le cas */
 else if (val<a->val) return rechercher(val, a->gauche);
 else return rechercher(val, a->droit);
}
```

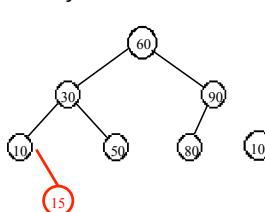
105

# ABR Ajouter

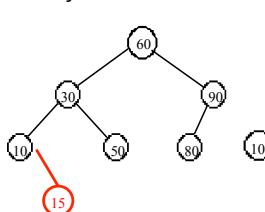
## Ajout de 5



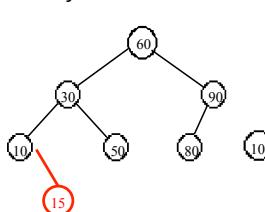
## Ajout de 15



## Ajout de 70



## Ajout de 100



106

# ABR Ajouter

## Algo : ajouter un nœud à un arbre; Quelle est la complexité ?

- | si l'arbre est vide alors
  - | créer un nœud et le retourner ce nœud
- | sinon
  - | si la valeur stockée sur la racine est plus grande que la valeur à insérer alors ajouter ce nœud dans le sous arbre gauche et retourner le nouvel arbre
  - | sinon ajouter ce nœud dans le sous arbre droit et retourner le nouvel arbre

```
ARBRE ajouter(etiquette nouveau, ARBRE a) {
 if (!arbre_vide(a)) { /* a!=arbre_vide() */
 if (nouveau< a->val)
 a->gauche=ajouter(nouveau,a->gauche);
 else
 a->droit=ajouter(nouveau,a->droit);
 return a;
 }
 else { ARBRE b=(arbre)malloc(1,sizeof(noeud));
 b->val=nouveau;
 return b;
 }
}
```

107

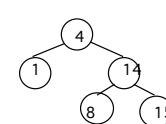
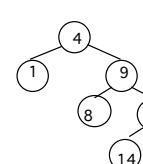
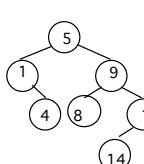
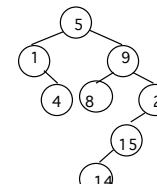
# ABR supprimer

## 3 cas

- | nœud sans fils : évident
- | nœud avec 1 seul fils : le nœud est remplacé par son fils
- | nœud avec 2 fils : remplacé par le plus petit élément du sous arbre droit ou le plus grand du sous arbre gauche pour respecter l'ordre

## Exemple :

- | Arbre
- | Suppression de 20      Suppression de 5      Suppression de 9



108

## ABR supprimer

```

ARBRE supprimer(etiquette val, ARBRE a){ /* val à supprimer */
 if (arbre_vide(a)) return a; /* arbre vide : pas de suppression */
 else if (val < a->val) { a->gauche=supprimer(val, a->gauche); return a; }
 else if (val > a->val) { a->droit =supprimer(val, a->droit); return a; }
 else{
 if (!a->gauche){ /* c'est le nœud à supprimer */
 if (!a->droit) { free(a); return(NULL); } /* Aucun fils */
 else {ARBRE q=a; a=a->droit; free(q); return a;} /*1 fils droit*/
 }
 else if(!a->droit)(ARBRE q=a; a=a->gauche; free(q); return a;) /*1 fils*/
 else { a->gauche= SupMax(a,a->gauche) ; return a; } /*2 fils*/
 }
}
/* dep : noeud à supprimer, recherche la valeur la plus grande du SAG (=Max) recopie dans le noeud *dep ce Max, détruit le noeud où se trouve ce Max retourne le nouvel SAD ainsi modifié*/
ARBRE SupMax(ARBRE dep, ARBRE p){
 if (!p->droit) { /* Max trouve */
 ARBRE q = p->gauche ; dep->val=p->val ; /* recopie de la valeur Max */
 free(p); return q; /* nouvel SAG */
 }
 else { p->droit = SupMax(dep,p->droit); return p ;}
}

```

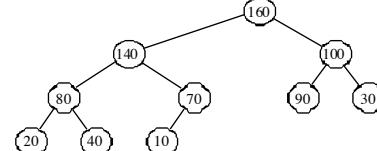
109

## TAS

## TAS ou HEAP

- ABR parfait partiellement ordonné
  - nœud avec 2 fils : arbre rempli par niveau
  - La racine est supérieure aux deux fils
  - deux fils non ordonnés

### Exemple

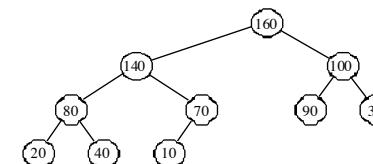


- Propriété essentielle
  - racine = plus grand élément

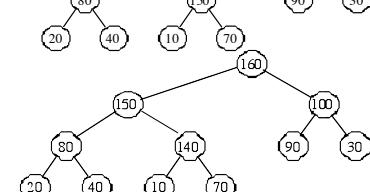
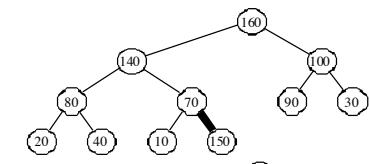
- Opérations
  - ajouter : élément x arbre -> arbre
  - max : arbre -> élément
    - retourne la racine
  - sup\_max: arbre -> arbre
    - retourne l'arbre privé de sa racine

111

## TAS : ajouter

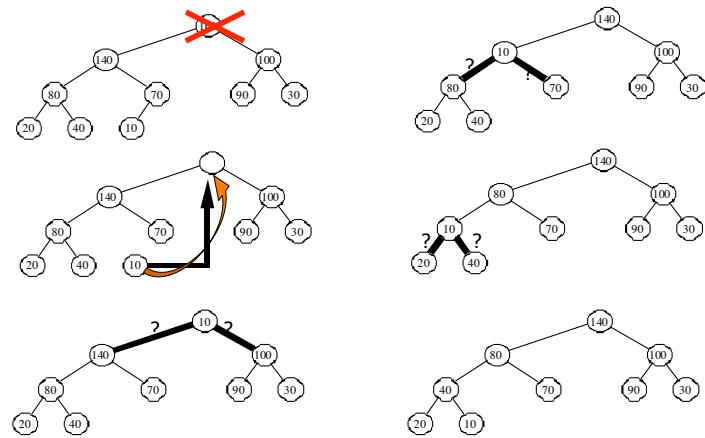


■ Ajout de 150



112

## TAS : suppression



113

## TAS : représentation

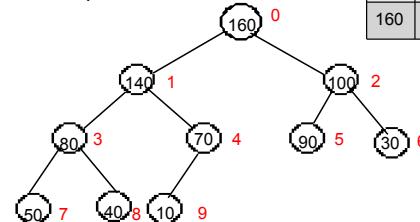
- numérotation des nœuds en ordre hiérarchique

- propriété:

- le père du noeud de numéro  $i$  porte le numéro  $(i - 1)/2$
- le fils gauche du noeud de numéro  $i$  porte le numéro  $i * 2 + 1$
- le fils droit du noeud de numéro  $i$  porte le numéro  $i * 2 + 2$

- Représentation en tableau

|     |     |     |    |    |    |    |    |    |    |
|-----|-----|-----|----|----|----|----|----|----|----|
| 0   | 1   | 2   | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 160 | 140 | 100 | 80 | 70 | 90 | 30 | 50 | 40 | 10 |

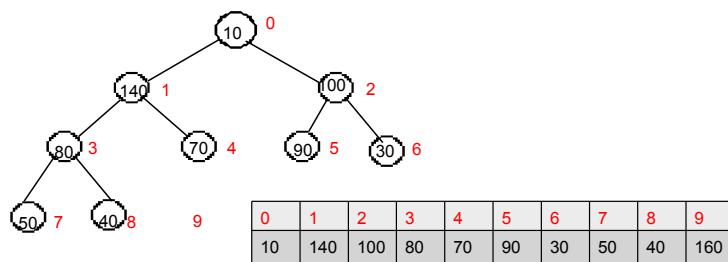


114

## Tri par TAS

- Prendre un tas.
- Etape 1 : Supprimer le maximum en l'échangeant avec le dernier élément du tableau
- Etape 2 : réorganiser le tas, en descendant la nouvelle racine (échange avec le plus grand des deux fils pour respecter l'ordre partiel)
- recommencer, avec un tas "diminué"
- Quelle est la complexité ?

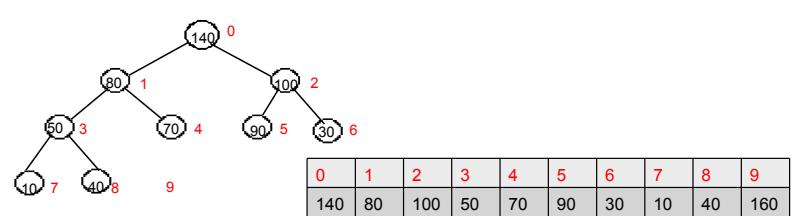
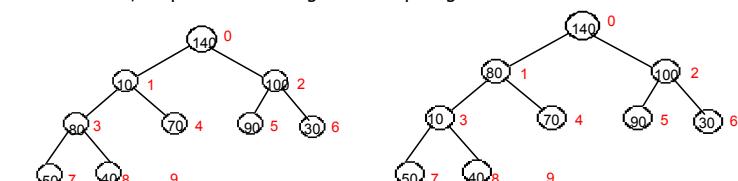
- Etape 1 : on échange 10 et 160



115

## Tri par TAS

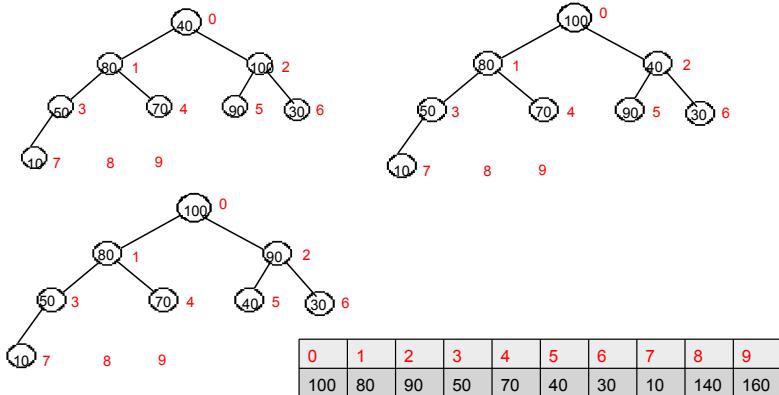
- Iteration 1, Etape 2 : on échange 10 et le plus grand de ses fils si besoin



116

## Tri par TAS

- Iteration 2, Etape 2 : on échange 40 et le plus grand de ses fils si besoin

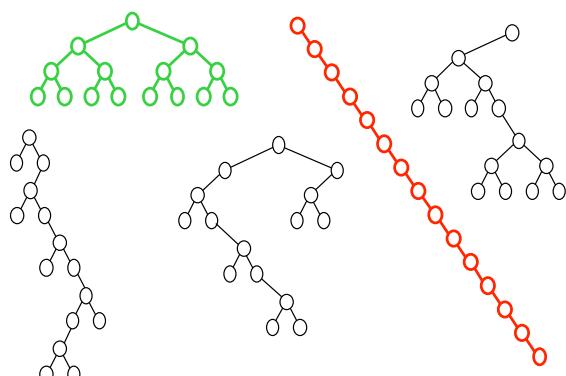


117

## Autres arbres

## Arbre : la structure parfaite ?

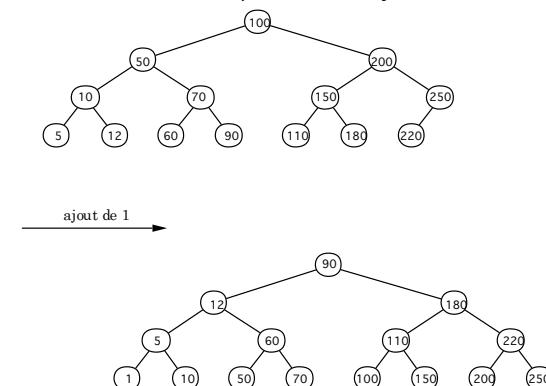
- Problème : nombre d'opérations à réaliser lors des ajouts/suppressions des éléments
  - dépend de la forme de l'arbre



119

## Arbre : maintenir la perfection ?

- Arbre parfait : coût des opérations uniforme
- Problème : maintien de l'équilibre trop lourd en recopie/déplacement d'éléments
- Exemple : tous les éléments sont déplacés lors de l'ajout de 1



120



## Arbres n-aires (2)

### ■ Représentation

```
typedef struct noeud { char val;
 struct noeud *fils, *frere; } NOEUD, * ARBRE;

■ Fonction d'ajout d'un mot à l'arbre : on ajoute le mot s à l'arbre
ARBRE ajouter(unsigned char *s, ARBRE r){
 if (s==NULL) return NULL; // Rien a ajouter
 if (!r || *s<r->val) { ARBRE p; // Arbre vide ou s plus petit
 p=(ARBRE) calloc(1,sizeof(NOEUD));
 p->val=*s;
 p->frere= r;
 if (*s!=0) p->fils=ajouter(s+1, NULL); //Pas en fin de mot ?
 return p;
 }
 else if (*s==r->val) { r->fils=ajouter(s+1,r->fils); return r;}
 else {r->frere=ajouter(s,r->frere); return r;}
}
```

125

## Arbres binaires : Exercices

- Faire une fonction qui teste l'égalité de deux arbres binaires, elle retourne 1 si les deux arbres sont égaux, 0 sinon
- Faire une fonction qui retourne la taille d'un arbre passé en paramètre
- Faire une fonction affiche les nœuds d'un ABR par ordre croissant
- Faire une fonction qui retourne le nombre de feuilles d'un arbre passé en paramètre
- Faire une fonction qui teste si un arbre binaire est complet.

126