Structures de données avancées

N. TSOPZE

Dictionnaires (rappel)

Dictionnaires

- Dictionnaire : ensemble dynamique d'objets composés de couple *(clé, valeur)* des clés comparables supportant les opérations suivantes :
  - Rechercher (S, k) retourne un pointeur x vers un élément dans S tel que
    x.key = k, ou NIL si un tel élément n'appartient pas à S.
  - Inserer(S, x) insère l'élément x dans le dictionnaire S. Si un élément de même clé se trouve dans le dictionnaire, sa valeur est mise à jour en tenant compte de l'ordre entre les lés
  - Supprimer(S, x) supprime l'élément x de S.

Pour faciliter la recherche, on peut supposer que les clés sont totalement ordonnés.

Dictionnaires (autres opérations)

- MINIMUM(*S*) : retourne l'élément de *S* ayant la plus petite clé.
- MAXIMUM(*S*) : retourne l'élément de *S* ayant la plus grande clé.
- SUCCESSEUR(*S, x*) : étant donné un élément *x* dont la clé appartient à un ensemble *S*, retourne le prochain élément de *S* qui est plus grand que *x*, ou NIL si *x* est l'élément maximal.
- PRÉDÉCESSEUR(*S, x*) : étant donné un élément *x* dont la clé appartient à un ensemble *S* totalement ordonné, retourne le prochain élément de *S* qui est plus petit que *x*, ou NIL si *x* est l'élément minimal

Arbres et ABR (rappel)

Définitions

- *Graphe connexe*
- *Nœud particulier: racine*
- *Chemin unique allant de la racine à n'importe quel nœud*
- pas de cycle
- nœud est représenté par un objet;
- Chaque nœud contient un champ *clé*
- pointeurs sur les autres nœuds (fils);
  - Nombre variable selon le type d'arborescence

Les structures arborescentes permettent une amélioration globale des accès aux informations

Définitions

- **Arbre** = graphe purement hiérarchique
- **Arbre binaire** = tout nœud a au plus deux fils
- **Liste** = arbre dégénéré
- **Forêt** = ensemble d 'arbres

- Définition récursive d'un arbre :
  - vide
  - constitué d'un élément
  - Constitué de plusieurs arbres

Définitions

- **nœud** : caractérisé par une valeur + un nombre fini de fils, possède un unique père
- **feuille** : nœud sans fils
- **nœud interne** : nœud qui n'est pas une feuille
- **arité** d'un nœud n : nombre de fils du nœud n
- **arité** d'un arbre a : nombre maximal de fils d'un nœud de a
- **racine** d'un arbre a : c'est le seul nœud sans père
- **profondeur** d'un nœud n : nombre de nœuds sur la branche entre la racine et le nœud n exclu
- **hauteur** d'un arbre a : c'est le nombre de nœuds sur la branche qui va de la racine de a à la feuille de profondeur maximale

Structure de données

Pointeur=^noeud

noeud = Enregistrement

  fils1: pointeur;

  …

  info:type_élément;

  fils:pointeur;

fin;

Parcours de l'arbre

- préfixé (préordre): on traite la racine, puis les sous-arbres gauches, enfin les sous-arbres droits
- infixé (projectif ou symétrique) : on traite les sous-arbres gauches, puis la racine, et enfin les sous-arbres droits
- postfixé (ordre terminal) : on traite les sous-arbres gauche, puis les sous-arbres droits, enfin la racine

Arbres binaires

Arbres binaires

- chaque noeud a au plus 2 fils,
- vide ou composé d'un élément auquel sont rattachés un sous-arbre gauche et un sous-arbre droit
  - valeur d'un noeud,
  - fils gauche d'un noeud
  - fils droit d'un noeud.

Arbre binaire complet

- Chaque noeud non terminal a exactement deux fils
- parfait :
  - avant-dernier niveau complet
  - les feuilles du dernier niveau groupées le plus à gauche possible

Arbre binaire Ordonné

- La chaîne infixée des valeurs est ordonnée
- Tous les éléments dans le sous-arbre gauche d'un noeud sont inférieurs à l'élément racine
- Tous les éléments dans son sous-arbre droit sont supérieurs à l'élément racine

Arbre binaire - parcours

- préfixé (préordre): on traite la racine, puis le sous-arbre gauche, puis le sous-arbre droit
- infixé (projectif ou symétrique) : on traite le sous-arbre gauche, puis la racine, puis le sous-arbre droit
- postfixé (ordre terminal) : on traite le sous-arbre gauche, le sous-arbre droit, puis la racine

Arbre binaire - parcours

procédure parcoursprefixe(racine: noeud)

si racine < > nil alors

1. traiter(racine);
2. parcoursprefixe(racine^.gauche);
3. parcoursprefixe(racine↑.droite);

finsi

fin

Arbre binaire - parcours

procédure parcours_infixe(racine: noeud)

si racine < > nil alors

1. Parcours_infixe(racine^.gauche);
2. traiter(racine);
3. Parcours_infixe(racine^.droite);

finsi

fin

Arbre binaire - parcours

procédure postfixé(racine:noeud);

si racine <> nil alors

1. postfixé(racine^.gauche);
2. postfixé(racine^.droite);
3. traiter(racine);

finsi

Arbre binaire

1. **Calcul de la taille d'un arbre binaire**
2. **Nombre de feuilles d'un arbre binaire**
3. **Vérifier qu'un arbre n'est pas dégénéré**
4. **Recherche une valeur dans un arbre binaire**
5. Longueur de cheminement de l'arbre = somme des longueurs de tous les chemins issus de la racine : LC

Arbre binaire

1. Longueur de cheminement externe = somme des longueurs de toutes les branches issues de la racine : LCE
2. Profondeur moyenne (d'un noeud) de l 'arbre = moyenne des hauteurs de tous les noeuds : LC/taille
3. Profondeur moyenne externe (d'une feuille) de l'arbre = moyenne des longueurs de toutes les branches :LCE/nbfeuilles

Arbres Binaires de Recherche

ABR

- opérations d'ensemble dynamique: RECHERCHER, MINIMUM, MAXIMUM,PRÉDÉCESSEUR, SUCCESSEUR, INSÉRER et SUPPRIMER
- arbre binaire complet à n noeuds, opérations en $\Theta(\lg n)$ dans le cas le plus défavorable.
- arbre dégénéré à n noeuds, opérations en $\Theta(n)$
- hauteur attendue d'un arbre binaire de recherche construit aléatoirement est $O(\lg n)$, ce opérations de base en $\Theta(\lg n)$ en moyenne

ABR - éléments

- champ *clé ou info*
- son enfant de gauche,
- son enfant de droite
- son parent

ABR

propriété d'arbre binaire de recherche:

- Si y est un noeud du sous-arbre de gauche de x, alors $clé[y] \leq clé[x]$.
- Si y est un noeud du sous-arbre de droite de x, alors $clé[x] \leq clé[y]$.

ABR

ARBRE-RECHERCHER(x, k)

si x = NIL ou k = *clé*[x] **alors**

retourner x

si k < *clé*[x] **alors retourner**

ARBRE-RECHERCHER(*gauche*[x], k)

sinon

retourner ARBRE-RECHERCHER(*droite*[x], k)

ABR

Fonction minimum

Fonction Maximum

Fonction Successeur

Fonction prédécesseur

ABR - INSERTION ET SUPPRESSION

- Modification de la structure de l'arbre binaire de recherche.
- conservation la propriété d'arbre binaire de recherche

arbres rouges et noirs

Définition

arbre binaire de recherche est un arbre rouge-noir si :

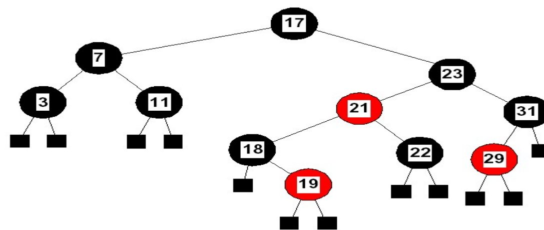
- chaque nœud est soit rouge, soit noir ;
- la racine est noire ;
- chaque sous-arbre vide est noir ;
- si un nœud est rouge, alors ses deux enfants sont noirs ;
- pour chaque nœud, tous les chemins reliant le nœud à une feuille contiennent le même nombre de nœuds noirs (ce nombre est appelé hauteur noire).

Définition

- **ABR** où chaque nœud est de couleur rouge ou noire de telle que:
  - Feuilles sont nulles, (sentinelles, nœuds externes)
  - les feuilles sont noires,
  - les fils d'un nœud rouge sont noirs,
  - le nombre de nœuds noirs le long d'une branche de la racine à une feuille est indépendant de la branche.

  les chemins du nœud vers les feuilles qui en dépendent ont le même nombre de nœuds noirs.

Définition



Hauteur - Proposition

Soit un arbre rouge-noir de hauteur h et possédant n nœuds vérifie :

$$h \leq 2 \log_2(n+1)$$

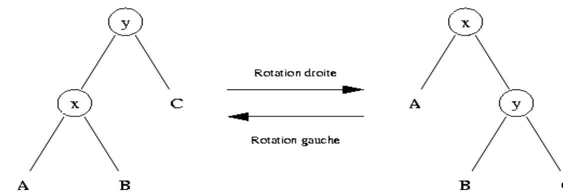
hauteur noire hn(x) : nombre de nœuds internes noirs le long d'une branche de la racine *x* à une feuille

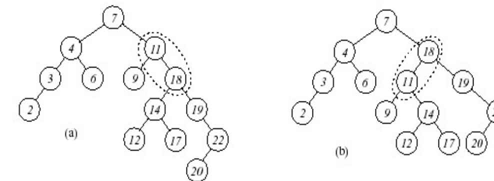
Rotations

échanger un nœud avec l'un de ses fils:

1. rotation droite: parent devient fils droit de son fils (ancien) gauche.
2. rotation gauche: parent devient le fils gauche de son fils (ancien) droit.
3. rotations gauche et droite sont inverses l'une de l'autre

Rotations





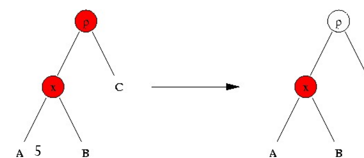
Insertion

- Pareil que dans le cas de l'arbre binaire de recherche
- Couleur du nouveau nœud: rouge
- Problème: possibilité d'avoir deux rouges successifs: père rouge et fils rouge.

Modifier l'arbre pour rééquilibrer

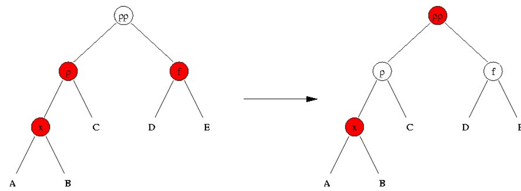
Cas 1: le père est la racine

- Changement de couleur du père
- Augmentation de la hauteur noire



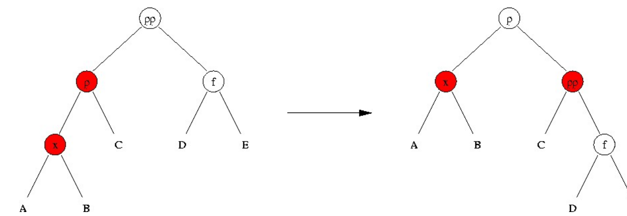
Cas 1: frère du père est rouge

- Le père et son frère deviennent noirs et leur père (grand-père) devient rouge
- Possibilité d'avoir deux rouges consécutifs



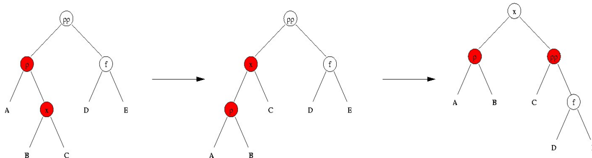
Cas 1: frère du père est noir et insertion à gauche

- une rotation droite entre père et grand-père.
- le père devient noir et le grand-père rouge



Cas 1: frère du père est noir et insertion à droit

- rotation gauche entre le fils et le père de sorte que le père p devienne le fils gauche du fils.
- rotation droite entre le fils et le grand-père.
- le fils devient noir et le grand-père rouge

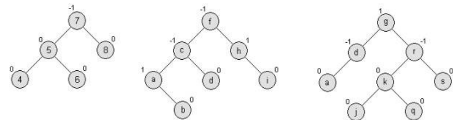


arbres AVL

G.M. Adelson-Velsky et E.M.Landis (1982)

Arbres AVL

- arbre binaire tel que
  - différence de hauteur entre le sous arbre gauche et le sous arbre droit d'un nœud diffère d'au plus 1.
  - les arbres gauches et droits d'un sommet sont des arbres AVL
  - éviter d'avoir des situations où la recherche est longue

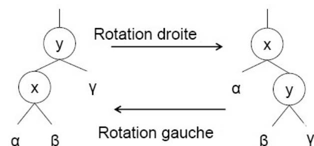


Arbres AVL (équilibrage)

- *facteur d'équilibrage* : entier ∈ {-1,0,1} (détenu par chaque nœud) permettant de déterminer si il est nécessaire de rééquilibrer l'arbre
- facteur d'équilibrage $eq(s)= h(D_s)-h(G_s)$
  - $h(NIL)=0$
  - A est un arbre $h(A)=hauteur(A)+1$
- Après insertion d'un nouvel élément, eq(s) peut passer à -2 ou à 2, il faut donc faire des rotations

Arbres AVL (Rotation)

- rotation droite autour du sommet *y* d'un arbre binaire de recherche : faire descendre le sommet *y* et faire remonter son fils gauche *x* sans invalider l'ordre des éléments.
- rotation gauche autour du sommet *y* d'un arbre binaire de recherche : faire remonter le sommet *y* et faire descendre son parent x (et x devient son fils gauche) sans invalider l'ordre des éléments.



Arbres AVL (Rotation)

- Après rotation droite autour de y:
  - eq'(X)=eq(X)+1+max(eq'(Y),0)
  - eq'(Y)=eq(Y)+1-min(eq(X),0)
- Après rotation gauche autour de X:
  - eq'(X)=eq(X)-1-max(eq(Y),0)
  - eq'(Y)=eq(Y)-1+min(eq'(X),0)

eq'(X) et eq'(Y) les facteurs d'équilibrage après rotation

Arbres AVL (insertion)

- Pareilles que celles d'un arbre binaire de recherche
- Ajout de la gestion du facteur d'équilibrage
- Insertion:
  - création d'une feuille f avec y le premier ancêtre de cette feuille qui viole la condition AVL (*eq(y)=-2* ou *eq(y)=2)* et x le fils gauche de y *(eq(y)=-2)*
  - *Si eq(x)=-1* , on effectue une rotation droite,
  - *eq(x)=1*, alors *x* a un sous arbre droit de racine *z*, qui a deux sous arbres, on effectue une rotation gauche autour de *x* puis une rotation droite autour de *y*

Arbres AVL (suppression)

- Pareilles que celles d'un arbre binaire de recherche
- Ajout de la gestion du facteur d'équilibrage
- Suppression:
  - Suppression d'une feuille avec y le premier ancêtre de cette feuille qui viole la condition AVL (*eq(y)=-2* ou *eq(y)=2)* , x le fils gauche de y et t le fils droit et la feuille supprimée est de racine t
    - *Si eq(x)=-1 ou eq(x)=0* , on effectue une rotation droite,
    - *eq(x)=1*, alors *x* a un sous arbre droit de racine *z*, qui a deux sous arbres, on effectue une rotation gauche autour de *x* puis une rotation droite autour de *y*

B-arbres

B-arbres

- arbres de recherche équilibrés conçus pour être efficaces sur des disques magnétiques ou autres unités de stockage secondaires à accès direct
  $\Rightarrow$minimiser les entrées-sorties disque
- Possibilité d'avoir de nombreux enfants (milliers).
- facteur de ramification (#enfants d'un nœud) : déterminé par les caractéristiques de l'unité de disque utilisée
- tout B-arbre à n noeuds a une hauteur $O(\lg n)$

B-arbres (propriétés)

Un ***B-arbre*** *T :* arborescence (de racine *racine*[*T*]).

- Chaque noeud *x* contient les champs :
  - $n[x]$, le nombre de clés conservées par le noeud *x*,
  - les $n[x]$ clés elles-mêmes, stockées par ordre non décroissant : $clé_1[x]$ $clé_2[x]$ · · · $clé_{n[x]}[x]$,
  - *feuille*[*x*], une valeur booléenne qui vaut VRAI si *x* est une feuille et FAUX si *x* est un noeud interne.

B-arbres (propriétés)

Un ***B-arbre*** *T :* arborescence (de racine *racine*[*T*]).

1. Chaque noeud interne *x* contient également $n[x] + 1$ pointeurs $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ vers ses enfant. Les feuilles n'ont pas d'enfants, et leurs champs *ci* ne sont donc pas définis.
2. Les clés $clé_i[x]$ déterminent les intervalles de clés stockés dans chaque sous-arbre : si k_i est une clé stockée dans le sous-arbre de racine $c_i[x]$, alors :

$$k_1 \leq clé_1[x] \leq k_2 \leq clé_2[x] \leq \ldots \leq clé_{n[x]}[x] \leq k_{n[x]+1} .$$

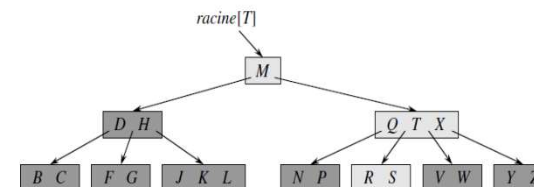
1. Toutes les feuilles ont la même profondeur, qui est la

B-arbres (propriétés)

Un ***B-arbre*** *T :* arborescence (de racine *racine*[*T*]).

- Il existe un majorant et un minorant pour le nombre de clés pouvant être contenues par un noeud. Ces bornes peuvent être exprimées en fonction d'un entier fixé $t \geq 2$, appelé le ***degré minimal*** du B-arbre :
  a. Tout noeud autre que la racine doit contenir au moins $t - 1$ clés. Tout nœud interne autre que la racine possède donc au moins *t* enfant. Si l'arbre n'est pas vide, la racine doit posséder au moins une clé.
  b. Tout noeud peut contenir au plus $2t-1$ clés. Un noeud interne peut donc posséder au plus $2t$ enfants. On dit qu'un noeud est ***complet*** s'il contient exactement $2t - 1$ clés

Tables à accès direct



Source: Cormen et al.

un noeud interne *x* qui contient $n[x]$ clés possède $n[x] + 1$ enfants.
Ces clés *x* sont utilisées comme points de séparation de l'intervalle des clés gérées par *x* en $n[x] + 1$ sous-intervalles, chacun étant pris en charge par un enfant de *x*

B-arbres (Opérations)

- RECHERCHER-B-ARBRE : vérifie si un élément est dans le B-arbre, prise de décision parmi $(n[x] + 1)$ options possibles.
- CRÉER-B-ARBRE : créer un noeud racine vide ; ensuite, utiliser INSÉRER-B-ARBRE pour ajouter de nouvelles clés
- INSÉRER-B-ARBRE : ***partager*** un noeud y plein (ayant $2t - 1$ clés) autour de sa ***clé médiane*** $clé_t[y]$, pour en faire deux nœuds ayant chacun $t-1$ clés. Remonter la clé de cette médiane dans le parent de y pour identifier le point de partage entre les deux nouveaux arbres.

Tables de hachage

Définition

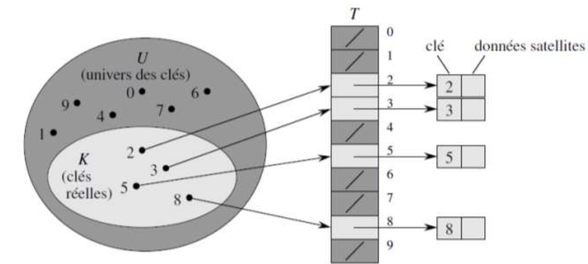
- structure de données permettant d'implémenter efficacement des dictionnaires
  - recherche d'un élément en $\Theta(n)$ (cas le plus défavorable),
  - Sous certaines hypothèses raisonnables, le temps moyen de recherche d'un élément dans une table de hachage est $O(1)$
- généralisation de la notion de tableau ordinaire.
  - adressage direct dans un tableau ordinaire utilise efficacement la possibilité d'examiner une position arbitraire dans un tableau en temps $O(1)$. Utiliser l'adressage direct lorsqu'on est en mesure d'allouer un tableau qui possède une position pour chaque clé possible

Tables à accès direct

Tables à accès direct

- besoin d'un ensemble dynamique dans lequel chaque élément possède une clé prise dans l'univers $U = \{0, 1, \ldots, m-1\}$, avec m n'est pas trop grand et deux éléments distincts n'ont pas la même clé
- Utilisation d'un tableau, ***table à adressage direct***, $T[0 \,..\, m-1]$ pour représenter l'ensemble dynamique, dans lequel chaque position (***alvéole***), correspond à une clé de l'univers U.
- Chaque **alvéole** k pointe vers un élément de l'ensemble ayant pour clé k, *ou NIL si* l'ensemble ne contient aucun élément de clé k, ie $T[k]$ = NIL.

Tables à accès direct



Source: Cormen et al.

Tables à accès direct (op. du dico)

- Recherche

  RECHERCHER-ADRESSAGE-DIRECT(*T, k*)

  retourner $T[k]$
- Insertion

  INSÉRER-ADRESSAGE-DIRECT(T, x)

  T[clé[x]] ← x
- Suppression

  SUPPRIMER-ADRESSAGE-DIRECT(T, x)

  T[clé[x]] ← NIL
- Opérations en O(1)

Tables de hachage

Tables de hachage

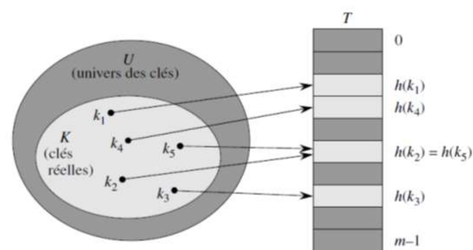
- si l'univers U est grand, alors il devient difficile de gérer une table T de taille $|U|$ compte tenu de l'espace mémoire
- l'ensemble K des clés *réellement conservées* peut être trop petit comparé à U ie la majeure partie de l'espace alloué pour T est gaspillé
- la table de hachage requiert moins de place de stockage qu'une table à adressage direct si l'ensemble K des clés stockées dans un dictionnaire est beaucoup plus petit que l'univers U de toutes les clés possibles
  - espace de stockage réduit à $\Theta(|K|)$,
  - recherche d'un élément dans la table de hachage en $O(1)$.

Tables de hachage

- adressage direct $\Rightarrow$ conservation de l'élément de clé k dans l'alvéole k
- Hachage $\Rightarrow$ conservation de l'élément de clé k dans l'alvéole $h(k)$,
  - Utilisation d'une ***fonction de hachage*** h pour calculer l'alvéole à partir de la clé k, réduire l'intervalle des indices de $|U|$ à m
  - $h : U \longrightarrow \{0, 1, \ldots, m-1\}$

  correspondance entre l'univers U des clés et les alvéoles d'une ***table de hachage*** $T[0 \,..\, m-1]$
  - élément de clé k : ***haché*** dans l'alvéole $h(k)$;
  - $h(k)$: ***valeur de hachage*** de la clé k

Tables de hachage



Source: Cormen et al.

Problème:
Collision : deux clés peuvent être hachées vers la même alvéole (ex. k_2 et k_5)

Tables de hachage (résolution de collisions par chainage)

- placer dans une liste chaînée tous les éléments hachés vers la même alvéole
  - alvéole j contient un pointeur vers la tête de liste de tous les éléments hachés vers j ; si aucun n'élément n'est présent, l'alvéole j contient NIL.
- opérations de dictionnaire sur une table de hachage T .
  - Insertion

    INSÉRER-HACHAGE-CHAÎNÉE(T, x)

    insère x en tête de la liste $T[h(clé[x])]$
  - recherche

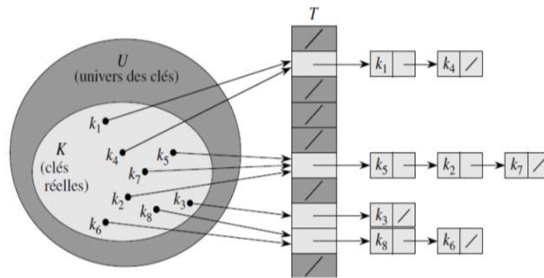
    RECHERCHER-HACHAGE-CHAÎNÉE(T, k)

    recherche un élément de clé k dans la liste $T[h(k)]$
  - suppression

    SUPPRIMER-HACHAGE-CHAÎNÉE(T, x)

    supprime x de la liste $T[h(clé[x])]$

Tables de hachage (résolution de collisions par chainage)



Source: Cormen et al.

Fonctions de hachage

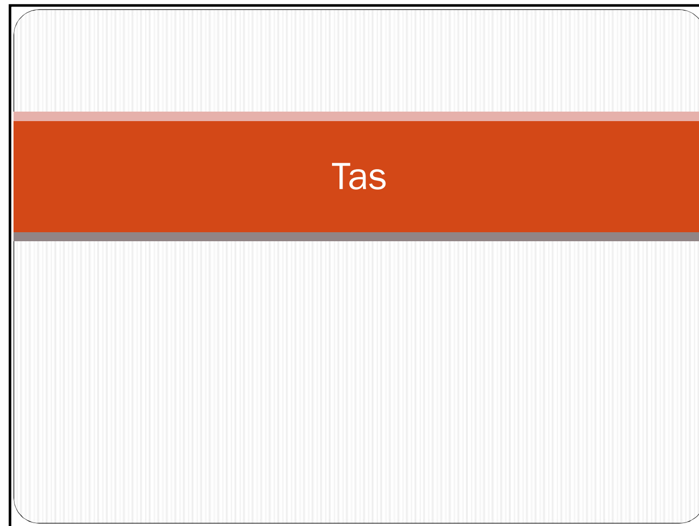
Tables de hachage (fonction de hachage)

- Supposer que les clés sont des entiers naturels
- ***méthode de la division*** : faire correspondre une clé k avec l'une des m alvéoles en prenant le reste de la division de k par m
  - $h(k) = k \bmod m$.
  - si $m = 2^p$, $h(k)$ est constitué des p bits de poids faible de k, *donc* permutation des caractères de k ne modifie pas sa valeur de hachage or il vaut mieux faire dépendre la fonction de hachage de tous les bits de la clé

Tables de hachage (fonction de hachage)

- Supposer que les clés sont des entiers naturels
- ***méthode de la multiplication*** :
  - multiplier la clé k par une constante A de l'intervalle $0 < A < 1$ et extraire la partie décimale de kA.
  - multiplier cette valeur par m et prendre la partie entière du résultat.
  - fonction de hachage : $h(k) = \lfloor m(kA \bmod 1) \rfloor$,

où « $kA \bmod 1$ » représente la partie décimale de kA: $kA - \lfloor kA \rfloor$



Structure de tas

- tableau qui peut être vu comme un arbre binaire presque complet
  - Chaque nœud de l'arbre correspond à un élément du tableau qui contient la valeur du noeud.
  - L'arbre est complètement rempli à tous les niveaux, sauf éventuellement au niveau le plus bas
- Un tableau *A* représentant un tas est un objet ayant deux attributs :
  - *longueur*[*A*], nombre d'éléments du tableau,
  - *taille*[*A*], nombre d'éléments du tas rangés dans le tableau *A*.
  - *A*[1 . . *longueur*[*A*]] contient des nombres valides

Structure de tas

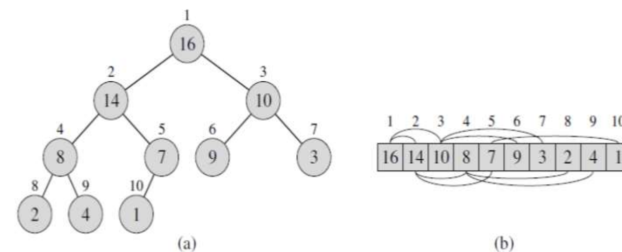
- racine de l'arbre : *A*[1],
- étant donné l'indice *i* d'un noeud, les indices de son parent PARENT(*i*), de son enfant de gauche GAUCHE(*i*) et de son enfant de droite DROITE(*i*) peuvent être facilement calculés :
  - PARENT(*i*)

  retourner *i* / 2
  - GAUCHE(*i*)

  retourner 2*i*
  - DROITE(*i*)

  retourner 2*i* + 1

Structure de tas



(a) (b)

Sortes de tas

- Tas min

pour chaque noeud i autre que la racine,

  - $A[\text{PARENT}(i)] \leq A[i]$.
  - Le plus petit élément d'un tas min est à la racine.

- Tas max

pour chaque noeud i autre que la racine,

  - $A[\text{PARENT}(i)] \geq A[i]$,
  - Le plus grand élément d'un tas max est stocké dans la racine,

Tas max (entasser)

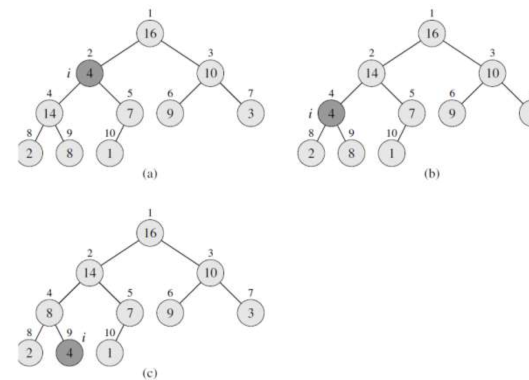
- Rôle: faire « descendre » la valeur de $A[i]$ dans le tas max de manière que le sous-arbre enraciné en i devienne un tas max.
- entrée : un tableau A et un indice i.
- Quand ENTASSER-MAX est appelée, on suppose que les arbres binaires enracinés en GAUCHE(i) et DROITE(i) sont des tas max, mais que $A[i]$ peut être plus petit que ses enfants, violant ainsi la propriété de tas max.

Tas max (entasser)

ENTASSER-MAX(A, i)

1 $l \leftarrow$ GAUCHE(i)
2 $r \leftarrow$ DROITE(i)
3 **si** $l \leq taille[A]$ et $A[l] > A[i]$
4 **alors** $max \leftarrow l$
5 **sinon** $max \leftarrow i$
6 **si** $r \leq taille[A]$ et $A[r] > A[max]$
7 **alors** $max \leftarrow r$
8 **si** $max \neq i$
9 **alors** échanger $A[i] \leftrightarrow A[max]$
10 ENTASSER-MAX(A, max)

Tas max (entasser)



Ta max (Construire)

- CONSTRUIRE-TAS-MAX(A)
  - sous-tableau $A[(n/2+1)..n]$: feuilles de l'arbre
  - parcourir les autres noeuds de l'arbre et appeler ENTASSER-MAX pour chacun

1 *taille*[A] ← *longueur*[A]
2 **pour** i ← ⌊*longueur*[A]/2⌋ **jusqu'à** 1
3 **faire** ENTASSER-MAX(A, i)