Faculté des sciences appliquées

Structures de données et algorithmes de P. Gueurts

Théorie

Antoine Wehenkel

Année académique 2014-2015

26 février 2015

Table des matières

Chapitre 1

Introduction et récursivité

Chapitre 2

Outils d'analyse

Chapitre 3

Tri

3.1 Algorithmes

3.1.1 Tri rapide(QuickSort)

Pseudo-code

PARTITION(A, p, r)
1 $x = A[r]$
2 $i = p - 1$
3 **for** $j = p$ **to** $r - 1$
4 **if** $A[j] \leq x$
5 $i = i + 1$
6 $swap(A[i], A[j])$
7 $swap(A[i + 1], A[r])$
8 **return** $i + 1$

QUICKSORT(A, p, r)
1 **if** $p < r$
2 $q =$ PARTITION(A, p, r)
3 QUICKSORT$(A, p, q - 1)$
4 QUICKSORT$(A, q + 1, r)$

Appel initial : QUICKSORT(A,1,A.length)

Complexité en temps

Complexité de partition : $T(n) = \Theta(n)$

Pire cas q = p ou q = r.
On a alors $T(n) = \Theta(n^2)$

Meilleur cas q = $\lfloor n/2 \rfloor$
On a alors $T(n) = \Theta(nlog(n))$

Cas moyen En moyenne cet algorithme est $\Theta(nlog(n))$

Complexité en espace

Complexité en espace $O(log(n))$ si bien implémenté(récursif terminal, en développant d'abord la partition la plus petite)

Propriétés

Itératif	Récursif	En place	Stable
Oui	Oui	Oui	Non

3.1.2 Construction d'un tas(Build-Max-Heap et tri par tas

Pseudo-code

MAX-HEAPIFY(A, i)

```
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size ∧ A[l] > A[i]
 4        largest = l
 5  else largest = i
 6  if r ≤ A.heap-size ∧ A[r] > A[largest]
 7        largest = r
 8  if largest ≠ i
 9        swap(A[i], A[largest])
10        MAX-HEAPIFY(A, largest)
```

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3        MAX-HEAPIFY(A, i)
```

HEAP-SORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3        swap(A[i], A[1])
4        A.heap-size = A.heap-size − 1
5        MAX-HEAPIFY(A, 1)
```

Complexité

MAX-HEAPIFY a une complexité au pire égale à la hauteur du noeud :$T(n) = O(log(n)) = O(h)$. On a donc comme complexité dans le pire cas les cas pour BUILD-MAX-HEAP de $T(n) = \Theta(n)$, cf. slides 154-158. On a pour le tri par tas une complexité dans tous les cas une complexité $\Theta(nlog(n))$.

Propriétés

Itératif	Récursif	En place	Stable
Oui	Oui	Oui	Non

Les points forts du tri par tas sont son efficacité et sa faible consommation de mémoire.

3.2 Questions

3.2.1 Qu'est-ce que le problème de tri ? Qu'est-ce qu'un algorithme de tri en place/itératif/ récursif/stable/comparatif ? Donnez à chaque fois un exemple.

- Le problème de tri :
  - Entrée : une séquence de n nombres $\langle a_1, a_2, \ldots, a_n \rangle$
  - Sortie : une permutation de la séquence de départ $\langle a'_1, a'_2, \ldots, a'_n \rangle$ telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- *Tri en place* : modifie directement la structure qu'il est en train de trier. Ne nécessite qu'une quantité de mémoire supplémentaire.
  Exemple : Insertion-Sort
  Contre-exemple : Merge-Sort
- *Tri itératif* : basé sur un ou plusieurs parcours itératifs du tableau.
  Exemple : Insertion-Sort
- *Tri récursif* : basé sur une procédure récursive.
  Exemple : Merge-Sort
- *Tri stable* : conserve l'ordre relatif des éléments égaux(au sens de la méthode de comparaison)
  Exemple : Insertion-Sort
  Contre-exemple : Quicksort
- *Tri comparatif* : basé sur la comparaison entre les éléments(clés)
  Exemple : Insertion-Sort

3.2.2 Qu'est-ce qu'un arbre ? Qu'est-ce qu'un arbre binaire/ordonné/binaire entier/binaire parfait/binaire complet ? Qu'est-ce que la hauteur d'un arbre ?

- Définition : Un arbre [1] (*tree*) T est un graphe dirigé (N, E), où :
  - N est un ensemble de nœuds, et
  - $E \subset N \times N$ est un ensemble d'arcs,

  possédant les propriétés suivantes :
  - T est connexe et acyclique
  - Si T n'est pas vide, alors il possède un nœud distingué appelé racine (*root node*). Cette racine est unique.
  - Pour tout arc $(n_1, n_2) \in E$, le nœud n_1 est le parent de n_2.
    - La racine de T ne possède pas de parent.
    - Les autres nœuds de T possèdent un et un seul parent.
- Un arbre *binaire* est un arbre ordonné possédant les propriétés suivantes :

1. En théorie des graphes, on parlera d'un arbre enraciné.

  - Chacun de ses noeuds possède au plus deux fils.
  - Chaque noeud fisl est soit un fils gauche, soit un fils droit.
  - Le fils gauche prédède le fils droit dans l'ordre des fils d'un noeud.

- Un arbre *ordonné* est un arbre dans lequel les ensembles de fils de chacun de ses nœuds sont ordonnés.
- Un arbre *binaire entier* ou propre (*full* or *proper*) est un arbre binaire dans lequel tous les nœuds internes possèdent exactement deux fils.
- Un arbre *binaire parfait* est un arbre binaire entier dans lequel toutes les feuilles sont à la même profondeur.
- Un arbre *binaire complet* est un arbre binaire tel que :
  - Si h dénote la hauteur de l'arbre :
    * Pour tout $i \in [0, h-1]$, il y a exactement 2^i nœuds à la profondeur i.
    * Une feuille a une profondeur h ou $h-1$.
    * Les feuilles de profondeur maximale (h) sont "tassées" sur la gauche.
- La *hauteur* (*height*) d'un nœud n est le nombre d'arcs d'un plus long chemin de ce nœud vers une feuille. La *hauteur de l'arbre* est la hauteur de sa racine.

3.2.3 Etablissez en utilisant les notations asymptotiques le lien qui existe entre la hauteur d'un arbre binaire (entier ou non) et le nombre de nœuds. Justifiez.

Dans n'importe quel arbre binaire, on a que la hauteur de l'arbre est $\leq$ au nombre de noeuds internes [2]. En effet si h est la hauteur de l'arbre celà veut dire qu'il existe un chemin qui contient h branches dans l'arbre, ce chemin est constitué de h + 1 noeuds, seul le dernier noeud est externe, le chemin contient donc h noeuds internes. L'arbre contient donc au moins h noeuds internes. Sachant celà on en déduit que $n \in \Omega(h)$, en effet $n > n_{interne}$ et $n_{interne} \geqslant h$ donc $n > h$. Le nombre de noeuds d'un arbre binaire est maximum quand celui-ci est parfait, dans ce cas $n = 2^{h+1} - 1$, n'importe quel arbre binaire a donc un nombre de noeuds $n < 2.2^h$ et donc $n \in O(2^h)$.

3.2.4 Qu'est-ce qu'un tas binaire ? Comment implémente-t-on un tas dans un vecteur ?

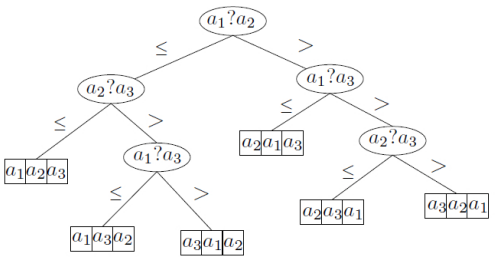
- Un *tas binaire* (binary heap) est un arbre binaire complet tel que :
  - Chacun de ses nœuds est associé à une clé.
  - La clé de chaque nœud est supérieure ou égale à celle de ses fils (*propriété d'ordre du tas*).
- Un tas peut être représenté de manière compacte à l'aide d'un tableau A.
  - La racine de l'arbre est le premier élément du tableau.
  - $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
  - $\textsc{Left}(i) = 2i$
  - $\textsc{Right}(i) = 2i + 1$

  Propriété d'ordre du tas : $\forall i, A[\textsc{Parent}(i)] \geq A[i]$

2. Un noeud qui possède au moins un fils est un noeud interne.

3.2.5 Construisez l'arbre de décision correspondant à un algorithme de tri X sur un tableau de taille Y .

Un algorithme de tri = un arbre binaire de décision (entier)



(arbre de décision pour le tri par insertion du tableau $[e_0, e_1, e_2]$)

Exercice : construire l'arbre pour le tri par fusion

Chapitre 4

Structure de données élémentaire

4.1 Algorithme

4.1.1 Insertion dans un tas(Heap-Insert)

Cf. section 4.7.2

4.2 Pile

4.2.1 Description

- Ensemble dynamique d'objets accessibles selon une discipline LIFO ("Last-in first-out").
- Interface
  - STACK-EMPTY(S) renvoie vrai si et seulement si la pile est vide
  - PUSH(S, x) pousse la valeur x sur la pile S
  - POP(S) extrait et renvoie la valeur sur le sommet de la pile S
- Applications :
  - Option 'undo' dans un traitement de texte
  - Langage postscript
  - Appel de fonctions dans un compilateur
  - Vérification de l'appariement de parenthèses
- Implémentations :
  - avec un tableau (taille fixée a priori)
  - au moyen d'une liste liée (allouée de manière dynamique)
  - ...

4.2.2 Implémentation

- S est un tableau qui contient les éléments de la pile
- $S.top$ est la position courante de l'élément au sommet de S

PUSH(S, x)
1 **if** $S.top == S.length$
2 **error** "overflow"
3 $S.top = S.top + 1$
4 $S[S.top] = x$

STACK-EMPTY(S)
1 **return** $S.top == 0$

POP(S)
1 **if** STACK-EMPTY(S)
2 **error** "underflow"
3 **else** $S.top = S.top - 1$
4 **return** $S[S.top + 1]$

- Complexité en temps *et en espace* : $O(1)$
(Inconvénient : L'espace occupé ne dépend pas du nombre d'objets)

4.3 File simple

4.3.1 Description

- Ensemble dynamique d'objets accessibles selon une discipline FIFO ("First-in first-out").
- Interface
  - ENQUEUE(Q, s) ajoute l'élément x à la fin de la file Q
  - DEQUEUE(Q) retire l'élément à la tête de la file Q
- Implémentation
  - A l'aide d'un tableau circulaire
  - A l'aide d'une liste liée

4.3.2 Implémentation à l'aide d'une liste liée

- Q est une liste simplement liée
- $Q.head$ (resp. $Q.tail$) pointe vers la tête (resp. la queue) de la liste

```
ENQUEUE(Q,x)
1  x.next = NIL
2  if Q.head == NIL
3        Q.head = x
4  else Q.tail.next = x
5  Q.tail = x
```

```
DEQUEUE(Q)
1  if Q.head == NIL
2        error "underflow"
3  x = Q.head
4  Q.head = Q.head.next
5  if Q.head == NIL
6        Q.tail = NIL
7  return x
```

- Complexité en temps $O(1)$, complexité en espace $O(n)$ pour n opérations

4.4 File double

4.4.1 Description

Combine accès LIFO et FIFO.Double ended-queue (dequeue).

- Généralisation de la pile et de la file
- Collection ordonnée d'objets offrant la possibilité
  - d'insérer un nouvel objet avant le premier ou après le dernier
  - d'extraire le premier ou le dernier objet
- Interface :
  - INSERT-FIRST(Q, x) : ajoute x au début de la file double
  - INSERT-LAST(Q, x) : ajoute x à la fin de la file double
  - REMOVE-FIRST(Q) : extrait l'objet situé en première position
  - REMOVE-LAST(Q) : extrait l'objet situé en dernière position
  - ...
- Application : équilibrage de la charge d'un serveur

4.4.2 Implémentation à l'aide d'une liste doublement liée

Soit la file double Q :

- Q.head pointe vers un élément sentinelle en début de liste
- Q.tail pointe vers un élément sentinelle en fin de liste
- Q.size est la taille courante de la liste

INSERT-FIRST(Q, x)

```
1  x.prev = Q.head
2  x.next = Q.head.next
3  Q.head.next.prev = x
4  Q.head.next = x
5  Q.size = Q.size + 1
```

INSERT-LAST(Q, x)

```
1  x.prev = Q.tail.prev
2  x.next = Q.tail
3  Q.tail.prev.next = x
4  Q.head.prev = x
5  Q.size = Q.size + 1
```

REMOVE-FIRST(Q)

```
1  if (Q.size == 0)
2        error
3  x = Q.head.next
4  Q.head.next = Q.head.next.next
5  Q.head.next.prev = Q.head
6  Q.size = Q.size − 1
7  return x
```

REMOVE-LAST(Q)

```
1  if (Q.size == 0)
2        error
3  x = Q.tail.prev
4  Q.tail.prev = Q.tail.prev.prev
5  Q.tail.prev.next = Q.head
6  Q.size = Q.size − 1
7  return x
```

Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

4.5 Liste

4.5.1 Description

- Ensemble dynamique d'objets ordonnés accessibles relativement les uns aux autres, sur base de leur position
- Généralise toutes les structures vues précédemment
- Interface :
  - Les fonctions d'une liste double (insertion et retrait en début et fin de liste)
  - INSERT-BEFORE(L, p, x) : insére x avant p dans la liste
  - INSERT-AFTER(L, p, x) : insère x après p dans la liste
  - REMOVE(L, p) : retire l'élement à la position p
  - REPLACE(L, p, x) : remplace par l'objet x l'objet situé à la position p
  - FIRST(L), LAST(L) : renvoie la première, resp. dernière position dans la liste
  - PREV(L, p), NEXT(L, p) : renvoie la position précédant (resp. suivant) p dans la liste
- Implémentation similaire à la file double, à l'aide d'une liste doublement liée (avec sentinelles)

4.5.2 Implémentation à l'aide d'une liste doublement liée

INSERT-BEFORE(L, p, x)

```
1  x.prev = p.prev
2  x.next = p
3  p.prev.next = x
4  p.prev = x
5  L.size = L.size + 1
```

REMOVE(L, p)

```
1  p.prev.next = p.next
2  p.next.prev = p.prev
3  L.size = L.size − 1
4  return p
```

INSERT-AFTER(L, p, x)

```
1  x.prev = p
2  x.next = p.next
3  p.next.prev = x
4  p.next = x
5  L.size = L.size + 1
```

Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

4.6 Vecteur

4.6.1 Description

- Ensemble dynamique d'objets occupant des rangs entiers successifs, permettant la consultation, le remplacement, l'insertion et la suppression d'éléments à des rangs arbitraires
- Interface
  - ELEM-AT-RANK(V, r) retourne l'élément au rang r dans V.
  - REPLACE-AT-RANK(V, r, x) remplace l'élément situé au rang r par x et retourne cet objet.
  - INSERT-AT-RANK(V, r, x) insère l'élément x au rang r, en augmentant le rang des objets suivants.
  - REMOVE-AT-RANK(V, r) extrait l'élément situé au rang r et le retire de r, en diminuant le rang des objets suivants.
  - VECTOR-SIZE(V) renvoie la taille du vecteur.
- Applications : tableau dynamique, gestion des éléments d'un menu,...
- Implémentation : liste liée, tableau extensible...

4.6.2 Implémentation par un tableau extensible

- Les éléments sont stockés dans un tableau extensible $V.A$ de taille initiale $V.c$.
- En cas de dépassement, la capacité du tableau est doublée.
- $V.n$ retient le nombre de composantes.
- Insertion et suppression :

INSERT-AT-RANK(V, r, x)

```
1  if V.n == V.c
2        V.c = 2 · V.c
3        W = "Tableau de taille V.c"
4        for i = 1 to n
5              W[i] = V.A[i]
6        V.A = W
7  for i = V.n downto r
8        V.A[i + 1] = V.A[i]
9  V.A[r] = x
10 V.n = V.n + 1
```

REMOVE-AT-RANK(V, r)

```
1  tmp = V.A[r]
2  for i = r to V.n − 1
3        V.A[i] = V.A[i + 1]
4  V.n = V.n − 1
5  return tmp
```

Complexité en temps

- INSERT-AT-RANK :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations d'insertion en début de vecteur -> coût amorti par opération est $\Theta(n)$
  - $\Theta(n)$ pour n opérations d'insertion en fin de vecteur -> coût amorti par opération est $\Theta(1)$
- REMOVE-AT-RANK :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations de retrait en début de vecteur -> coût amorti par opération est $\Theta(n)$
  - $\Theta(n)$ pour n opérations de retrait en fin de vecteur -> coût amorti par opération est $\Theta(1)$

Complexité en espace $Theta(n)$

4.7 File à priorité

4.7.1 Description

- Ensemble dynamique d'objets classés par ordre de priorité
  - Permet d'extraire un objet possédant la plus grande priorité
  - En pratique, on représente les priorités par les clés
  - Suppose un ordre total défini sur les clés
- Interface :
  - INSERT(S, x) : insère l'élément x dans S.
  - MAXIMUM(S) : renvoie l'élément de S avec la plus grande clé.
  - EXTRACT-MAX(S) : supprime et renvoie l'élément de S avec la plus grande clé.
- Remarques :
  - Extraire l'élément de clé maximale ou minimale sont des problèmes équivalents
  - La file FIFO est une file à priorité où la clé correspond à l'ordre d'arrivée des élements.
- Application : gestion des jobs sur un ordinateur partagé

4.7.2 Implémentation à l'aide d'un tas

- Accès et extraction du maximum :

```
HEAP-MAXIMUM(A)
1  return A[1]
```

HEAP-EXTRACT-MAX(A)
1 **if** $A.heap\text{-}size < 1$
2 **error** "heap underflow"
3 $max = A[1]$
4 $A[1] = A[A.heap\text{-}size]$
5 $A.heap\text{-}size = A.heap\text{-}size - 1$
6 MAX-HEAPIFY$(A, 1)$ // reconstruit le tas
7 **return** max

- Complexité : $\Theta(1)$ et $O(\log n)$ respectivement (voir chapitre 3)
- INCREASE-KEY(S, x, k) augmente la valeur de la clé de x à k (on suppose que $k \geq$ à la valeur courante de la clé de x).

HEAP-INCREASE-KEY(A, i, key)
1 **if** $key < A[i]$
2 **error** "new key is smaller than current key"
3 $A[i] = key$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5 $swap(A[i], A[\text{PARENT}(i)])$
6 $i = \text{PARENT}(i)$

- Complexité : $O(\log n)$ (la longueur de la branche de la racine à i étant $O(\log n)$ pour un tas de taille n).
- Pour insérer un élément avec une clé key :
  - l'insérer à la dernière position sur le tas avec une clé $-\infty$,
  - augmenter sa clé de $-\infty$ à key en utilisant la procédure précédente

HEAP-INSERT(A, key)
1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

- Complexité : $O(\log n)$.

$\Rightarrow$ Implémentation d'une file à priorité par un tas : $O(\log n)$ pour l'extraction et l'insertion.

Chapitre 5

Dictionnaires

5.1 Algorithmes

5.1.1 Recherche dichotomique

Peudo-code

BINARY-SEARCH$(V, k, low, high)$

```
1  if low > high
2        return NIL
3  mid = ⌊(low + high)/2⌋
4  x = ELEM-AT-RANK(V, mid)
5  if k == x.key
6        return x
7  elseif k > x.key
8        return BINARY-SEARCH(V, k, mid + 1, high)
9  else return BINARY-SEARCH(V, k, low, mid − 1)
```

Complexité et comparaison avec les autres algorithme

	Pire cas	*En moyenne*
Implémentation	SEARCH	
Liste	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(\log n)$
ABR	$\Theta(n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$
Table de hachage	$\Theta(n)$	$\Theta(1)$

5.1.2 Parcours d'arbre

Tous ces algorithme ont une complexité en temps dans tous les cas $\Theta(n)$.

Parcours infixe

Parcours infixe (en ordre) : Chaque nœud est visité après son fils gauche et avant son fils droit.

```
INORDER-TREE-WALK(T, x)
1  if hasLeft(T, x)
2        INORDER-TREE-WALK(T, LEFT(x))
3  print GETDATA(T, x)
4  if hasRight(T, x)
5        INORDER-TREE-WALK(T, RIGHT(x))
```

Parcous préfixe

Parcours préfixe (en préordre) : chaque nœud est visité avant ses fils.

```
PREORDER-TREE-WALK(T, x)
1  print GETDATA(T, x)
2  if hasLeft(T, x)
3        PREORDER-TREE-WALK(T, LEFT(x))
4  if hasRight(T, x)
5        PREORDER-TREE-WALK(T, RIGHT(x))
```

Parcours postfixe

Parcours postfixe (en postordre) : chaque nœud est visité après ses fils

```
POSTORDER-TREE-WALK(T, x)
1  if hasLeft(T, x)
2        POSTORDER-TREE-WALK(T, LEFT(x))
3  if hasRight(T, x)
4        POSTORDER-TREE-WALK(T, RIGHT(x))
5  print GETDATA(T, x)
```

Parcours en largeur

Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite des nœuds de profondeur 1, puis 2, ...

```
BREADTH-TREE-WALK(T)
 1  Q ="Empty queue"
 2  if not isEMPTY(T)
 3        ENQUEUE(Q, ROOT(T))
 4  while not QUEUE-EMPTY(Q)
 5        y = DEQUEUE(Q)
 6        print GETDATA(T, y)
 7        if hasLeft(T, y)
 8              ENQUEUE(Q, LEFT(y))
 9        if hasRight(T, y)
10              ENQUEUE(Q, RIGHT(y))
```

5.1.3 Arbre binaire de rechercher

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Table de hachage	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Recherche

- Recherche binaire

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2        return x
3  if k < x.key
4        return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Appel initial (à partir d'un arbre T)
TREE-SEARCH($T.root, k$)

- Complexité ? $T(n) \in O(h)$, où h est la hauteur de l'arbre
- Pire cas : $h = n$

Successeur

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.parent
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.parent
7  return y
```

Complexité : $O(h)$, où h est la hauteur de l'arbre

Insertion

TREE-INSERT(T, z)

```
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.parent = y
 9  if y == NIL
10        // Tree T was empty
11        T.root = z
12  elseif z.key < y.key
13        y.left = z
14  else y.right = z
```

Complexité : $O(h)$ où h est la hauteur de l'arbre

Suppression

TREE-DELETE(T, z)

```
 1  if z.left == NIL
 2        TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSPLANT(T, z, z.left)
 5  else // z has two children
 6        y = TREE-SUCCESSOR(z)
 7        if y.parent ≠ z
 8              TRANSPLANT(T, y, y.right)
 9              y.right = z.right
10              y.right.parent = y
11        // Replace z by y
12        TRANSPLANT(T, z, y)
13        y.left = z.left
14        y.left.parent = y
```

TRANSPLANT(T, u, v)

```
1  if u.parent == NIL
2        T.root = v
3  elseif u == u.parent.left
4        u.parent.left = v
5  else u.parent.right = v
6  if v ≠ NIL
7        v.parent = u.parent
```

Complexité : $O(h)$ pour un arbre de hauteur h
(Tout est $O(1)$ sauf l'appel à TREE-SUCCESSOR).

5.2 Questions

5.2.1 Qu'est-ce qu'un dictionnaire (principe, interface, exemples d'application) ? Enumérez au moins 4 manières de l'implémenter en précisant la complexité des opérations principales.

- Définition : un dictionnaire est un ensemble dynamique d'objets avec des clés comparables qui supportent les opérations suivantes :
  - SEARCH(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.
  - INSERT(S, x) insère l'élément x dans le dictionnaire S. Si un élément de même clé se trouve déjà dans le dictionnaire, on met à jour sa valeur
  - DELETE(S, x) retire l'élément x de S. Ne fait rien si l'élément n'est pas dans le dictionnaire.
- Deux objectifs en général :
  - minimiser le coût pour l'insertion et l'accès au données
  - minimiser l'espace mémoire pour le stockage des données
- Exemples d'applications :
  - Table de symboles dans un compilateur
  - Table de routage d'un DNS
  - ...
- Beaucoup d'implémentations possibles :

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

5.2.2 Comment peut-on implémenter un dictionnaire à l'aide d'un vecteur ?

- On suppose qu'il existe un ordre total sur les clés
- On stocke les éléments dans un vecteur qu'on maintient trié
- Recherche dichotomique (approche "diviser-pour-régner")

  BINARY-SEARCH$(V, k, low, high)$

```
1  if low > high
2      return NIL
3  mid = ⌊(low + high)/2⌋
4  x = Elem-At-Rank(V, mid)
5  if k == x.key
6      return x
7  elseif k > x.key
8      return Binary-Search(V, k, mid + 1, high)
9  else return Binary-Search(V, k, low, mid − 1)
```

Complexité au pire cas : $\Theta(\log n)$

- Insertion : recherche de la position par Binary-Search puis insertion dans le vecteur par Insert-At-Rank (=décalage des éléments vers la droite).
- Suppression : recherche puis suppression par Remove-At-Rank (=décalage des éléments vers la gauche).

5.2.3 Qu'est-ce qu'une structure de données de type arbre ? Précisez deux manières de l'implémenter.

- Principe :
  - Des données sont associées aux noeuds d'un arbre.
  - Les noeuds sont accessibles les uns par rapport aux autres selon leur position dans l'abre.
- Implémentation
  1. Structure liée
     - Comme pour un arbre binaire
     - Mais on remplace $n.left$ et $n.right$ par un pointeur $n.children$ vers un ensemble dynamique.
     - Le type d'ensemble dynamique (vecteur, liste, ...) dépendra des opérations devant être effectuées
  2. Représenter l'arbre quelconque par un arbre binaire
     - Si le nœud n possède les fils $n_1, n_2, \ldots, n_p$, avec $p > 2$, alors le sous-arbre issu de n est représenté par une arbre binaire dont :
       * n est la racine
       * Le fils gauche est la racine du sous-arbre issu de n_1
       * Le fils droit est associé à une valeur distinguée "$", et représente un arbre dont la racine possède les fils $n_2, n_3, \ldots, n_p$.

5.2.4 Montrez que les parcours infixe, préfixe et postfixe sont $\Theta(n)$.

- Soit $T(n)$ le nombre d'opérations pour un arbre avec n nœuds
- On a $T(n) = \Omega(n)$ (on doit au moins parcourir chaque nœud).
- Etant donné la récurrence, on a :

$$T(n) \leq T(n_L) + T(n - n_L - 1) + d$$

où n_L est le nombre de nœuds du sous-arbre à gauche et d une constante

- On peut prouver par induction que $T(n) \leq (c + d)n + c$ où $c = T(0)$ [1].
- $T(n) = \Omega(n)$ et $T(n) = O(n) \Rightarrow T(n) = \Theta(n)$

1. Cas de base : $n = 0$ ok car $T(0) = c \leq c$
Cas inductif : Pour tout $m \leq n : T(m) \leq (c + d)m + c \Rightarrow T(n) \leq (c + d)n + c$
Demo : $T(n) = T(n_L) + T(n - n_L - 1) \leq (c + d)n_L + c + (c + d)(n - n_L - 1) + c + d \leq (c + d)n + c$

5.2.5 Qu'est-ce qu'un arbre binaire de recherche ?

- Une structure d'arbre binaire implémentant un dictionnaire, avec des opérations en $O(h)$ où h est la hauteur de l'arbre
- Chaque nœud de l'arbre binaire est associé à une clé
- L'arbre satisfait à la propriété d'arbre binaire de recherche
  - Soient deux nœuds x et y.
  - Si y est dans le sous-arbre de gauche de x, alors $y.key < x.key$
  - Si y est dans le sous-arbre de droite de x, alors $y.key \geq x.key$

5.2.6 Comment peut-on trier avec un arbre binaire de recherche ? Comparez cet algorithme aux autres algorithmes de tri vu au cours.

```
BINARY-SEARCH-TREE-SORT(A)
1  T = "Empty binary search tree"
2  for i = 1 to n
3        TREE-INSERT(T, A[i])
4  INORDER-TREE-WALK(T.root)
```

- Complexité en temps identique au quicksort
  - Insertion : en moyenne, $n \cdot O(\log n) = O(n \log n)$, pire cas : $\Theta(n^2)$
  - Parcours de l'arbre en ordre : $\Theta(n)$
  - Total : $\Theta(n \log n)$ en moyenne, $\Theta(n^2)$ pour le pire cas
- Complexité en espace cependant plus importante, pour le stockage de la structure d'arbres.

5.2.7 Qu'est-ce qu'une table de hachage ? Décrivez les opérations d'insertion et de suppression et donnez leur complexité.

- Inventée en 1953 par Luhn
- Idée :
  - Utiliser une table T de taille $m \ll |U|$
  - stocker x à la position $h(x.key)$, où h est une fonction de hachage :

$$h : U \to \{0, \ldots, m-1\}$$

```
CHAINED-HASH-INSERT(T, x)
1  LIST-INSERT(T[h(x.key)], x)

CHAINED-HASH-DELETE(T, x)
1  LIST-DELETE(T[h(x.key)], x)
```

- Complexité :
  - Insertion : $O(1)$
  - Suppression : $O(1)$ si liste doublement liée, $O(n)$ pour une liste de taille n si liste simplement liée.

Dans le cas de résolution des collisions par chainage.

5.2.8 Qu'est-ce qu'une collision ? Qu'est-ce que le facteur de charge ? Expliquez le principe des deux méthodes principales de gestion des collisions, donnez leurs complexités (sans preuve) et comparez les.

- Collision : Lorsque deux clés distinctes k_1 et k_2 sont telles que $h(k_1) = h(k_2)$
- Le facteur de charge d'une table de hachage est donné par $\alpha = \frac{n}{m}$ où :
  - n est le nombre d'éléments deans la table
  - m est la taille de la table(c'est-à-dire, le nombre de listes liées)
- Il existe deux grandes façons de gèrer les collisions :
  1. Le chaînage : mettre les éléments qui sont hachés vers la même position dans une liste liée(simple ou double)
     - Insertion : $\Theta(1)$ dans tous les cas
     - Suppression : $\Theta(1)$ en moyenne, $O(n)$ dans le pire cas.
     - Recherche : $\Theta(1)$ en moyenne, $O(n)$ dans le pire cas.
  2. Adressage ouvert : Stocke tous les éléments dans le tableau, lorsqu'il y a collision on sonde le tableau pour trouver une case vide. On fera en sorte de toujours avoir α inférieur à 1.
     - Insertion : $\Theta(1)$ si α est constant.
     - Suppression : $\Theta(1)$ si α est constant.
     - Recherche : $\Theta(1)$ si α est constant.
- Adressage ouvert versus chaînage
  - Chaînage :
    - Peut gérer un nombre illimité d'éléments et de collisions
    - Performances plus stables
    - Surcoût lié à la gestion et le stockage en mémoire des listes liées
  - Adressage ouvert :
    - Rapide et peu gourmand en mémoire
    - Choix de la fonction de hachage plus difficile (pour éviter les grappes)
    - On ne peut pas avoir $n > m$
    - Suppression problématique

5.2.9 Décrivez deux fonctions de sondages.

1. Sondage linéaire
$$h(k, i) = (h'(k) + i) \bmod m,$$
où $h'(k)$ est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$.

   Propriétés :
   - très facile à implémenter
   - effet de grappe fort : création de longues suites de cellules occupées
     - La probabilité de remplir une cellule vide est $\frac{i+1}{m}$ où i est le nombre de cellules pleines précédant la cellule vide
   - pas très uniforme

2. Sondage quadratique

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

où h' est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$, c_1 et c_2 sont deux constantes non nulles.

Propriétés :

- nécessité de bien choisir les constantes c_1 et c_2 (pour avoir une permutation de $\langle 0, 1, \ldots, m-1 \rangle$)
- effet de grappe plus faible mais tout de même existant :
  - Deux clés de même valeur de hachage suivront le même chemin

$$h(k,0) = h(k',0) \Rightarrow h(k,i) = h(k',i)$$

- meilleur que le sondage linéaire

5.2.10 Qu'est-ce qu'une fonction de hachage ? Quelles sont les caractéristiques d'une bonne fonction de hachage ? Comment traiter des clés non numériques ?

- Fonction de hachage : fonction d'une clé qui renvoit une position dans le tableau pour cette clé.
- Fonction de hachage idéale :
  - Doit être facile à calculer.
  - Doit satisfaire l'hypothèse de hachage uniforme simple.
- Pour traiter des clé non numérique on utilise une fonction de codage.

5.2.11 Décrivez deux exemples de familles de fonctions de hachage

1. Méthode de division
   - La fonction de hachage calcule le reste de la division entière de la clé par la taille de la table

$$h(k) = k \bmod m.$$

   Exemple : $m = 20$ et $k = 91 \Rightarrow h(k) = 11$.

   Avantages : simple et rapide (juste une opération de division)

   Inconvénients : Le choix de m est très sensible et certaines valeurs doivent être évitées

   - Si la fonction de hachage produit des séquences périodiques, il vaut mieux choisir m premier
   - En effet, si m est premier avec b, on a :

$$\{(a + b \cdot i) \bmod m | i = 0, 1, 2, \ldots\} = \{0, 1, 2, \ldots, m-1\}$$

   $\Rightarrow$ Bonne valeur de m : un nombre premier pas trop près d'une puissance exacte de 2

2. Méthode de multiplication
   - Fonction de hachage :

$$h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$$

   où

  - A est une constante telle que $0 < A < 1$.
  - $kA \bmod 1 = kA - \lfloor kA \rfloor$ est la partie fractionnaire de kA.
- Inconvénient : plus lente que la méthode de division
- Avantage : la valeur de m n'est plus critique
- La méthode marche mieux pour certaines valeurs de A. Par exemple :

$$A = \frac{\sqrt{5} - 1}{2}$$

5.2.12 Comparez les arbres binaires de recherche et les tables de hachage en énumérant leurs avantages et défauts respectifs.

Tables de hachage :

- Faciles à implémenter
- Seule solution pour des clés non ordonnées
- Accès et insertion très rapides en moyenne (pour des clés simples)
- Espace gaspillé lorsque α est petit
- Pas de garantie au pire cas (performances "instables")

Arbres binaire de recherche (équilibrés) :

- Performance garantie dans tous les cas (stabilité)
- Taille de structure s'adapte à la taille des données
- Supportent des opérations supplémentaires lorsque les clés sont ordonnées (parcours en ordre, successeur, prédécesseur, etc.)
- Accès et insertion plus lente en moyenne

Chapitre 6

Résolution de problèmes