Structures de données et algorithmes

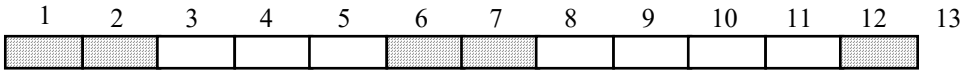
Projet : File de priorité de trous fusionnables

1. Présentation du problème

En cours nous avons étudié des ensembles/sacs dans lesquels les éléments — définis sur un ensemble doté d'une structure d'ordre total — sont indépendants : l'insertion d'un nouvel élément n'a pas d'influence sur l'existence ou les propriétés des éléments déjà présents. Ce caractère d'indépendance n'est pas toujours vérifié. Considérons en effet le cas d'une mémoire segmentée pour laquelle on répertorie les emplacements libres (trous) et telle que la suppression porte sur le plus grand élément (file de priorité). Dans une telle structure de données l'adjonction d'un nouvel intervalle peut avoir des conséquences sur les trous[1] existants puisqu'elle peut provoquer des fusions.

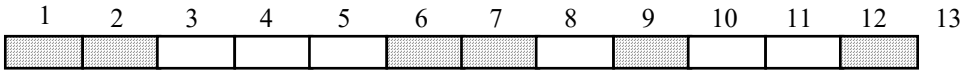
Exemple

Considérons la configuration mémoire suivante :



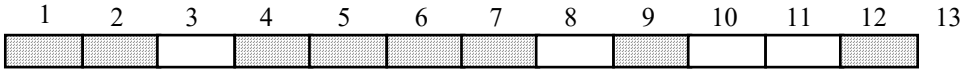
On a l'ensemble de trous : {(1,3), (6,8), (12,13)}.

1. On ajoute l'intervalle (9,10) :

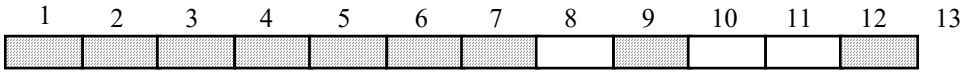
   On a l'ensemble de trous : {(1,3), (6,8), (9,10), (12,13)}.

2. On ajoute l'intervalle (4,6) :

   L'intervalle (4,6) et le trou (6,8) fusionnent pour produire le trou (4,8). On a donc l'ensemble de trous : {(1,3), (4,8), (9,10), (12,13)}.

3. On ajoute l'intervalle (3,4) :

   L'intervalle (3,4) et les trous (1,3) et (4,8) fusionnent pour produire le trou (1,8). On a donc l'ensemble de trous : {(1,8), (9,10), (12,13)}.

On note que dans une telle structure de données, deux trous quelconques sont toujours disjoints, (ils ne peuvent même être contigus). Cette propriété dote l'ensemble des trous d'une structure d'ordre total. On

[1]Par convention on réserve le terme de *trou* aux éléments *présents* dans la structure de données et d'*intervalle* aux éléments *à introduire* dans la structure de données.

suppose en outre que lorsque l'on demande l'adjonction d'un intervalle, celui-ci est compatible avec la structure de données existante (l'intersection avec tous les trous est vide).

L'interface du type considéré se présente sous la forme :

unité interface sdd_fdp_trou **c'est**

type fdp_trou;

procedure CREER (**var** fdp_trou E);
procedure AJOUT (**var** fdp_trou E; ent D, F);
procedure SUPPR (fdp_trou E; **var** ent LG);
function EST_VIDE (fdp_trou E) : bool;
function LG_MAX (fdp_trou E) : ent;
procédure AFF_ARBRE (fdp-trou E; ...);

La procédure CREER produit une file de priorité vide.

La procédure AJOUT va introduire un nouvel intervalle dans la file de priorité. L'ajout d'un intervalle va dans certains cas exiger de restructurer la file de priorité existante en fusionnant cet intervalle avec les éventuels trous *contigus* situés de part et d'autre de l'intervalle considéré.

La fonction EST_VIDE permet de savoir si une file de priorité est vide, la fonction LG_MAX délivre la longueur du plus grand trou existant dans la file de priorité. La procédure SUPPR supprime le trou de plus grande taille et fournit sa longueur.

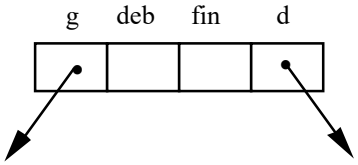
La procédure AFF_ARBRE affiche l'arbre (voir ci-dessous) correspondant à la file de priorité passée en paramètre. Pour une visualisation aisée, on pourra ajouter un paramètre (indentation) à cette procédure.

2. Représentations

On décide d'adopter une représentation en arbre binaire de recherche (sur le champ deb comme sur le champ fin) et en maximier sur la longueur[2] des trous.

Définition d'un maximier :

- un arbre vide est un maximier.
- un arbre binaire quelconque est un maximier si :
  - tous les noeuds sont ≤ à la racine
  - le sous-arbre gauche et le sous-arbre droit sont des maximiers.



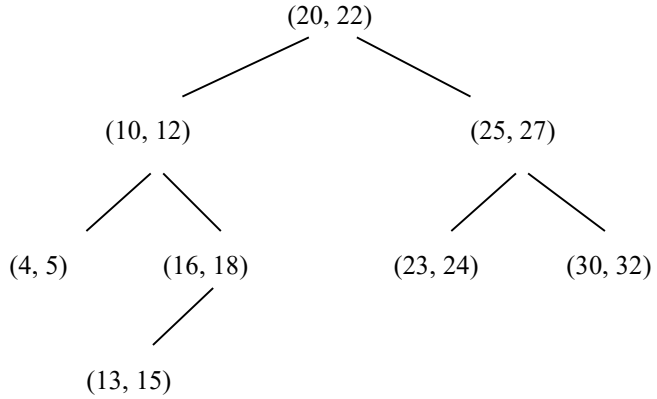
g : pointeur vers le sous-arbre gauche,
d : pointeur vers le sous-arbre droit
deb : coordonnée du début du trou,
fin : coordonnée de la fin du trou.

3. Indication sur les opérations

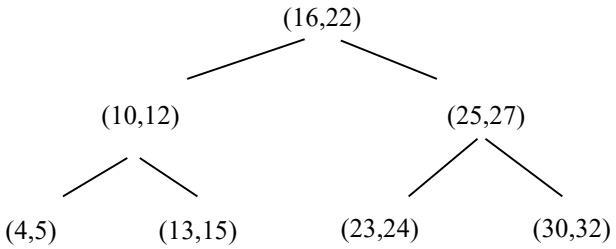
Lors de l'insertion d'un intervalle dans un tel arbre deux cas se présentent. (1) l'intervalle à insérer est situé strictement avant ou après le trou situé à la racine. (2) l'intervalle à insérer est contigu (à gauche pour fixer les idées) au trou. Dans le premier cas il faut effectuer l'insertion dans le sous-arbre gauche ou droit selon la position relative de l'intervalle et du trou à la racine. Dans le second cas il faut élargir le

[2]La taille d'un trou ne sera pas explicitement représentée dans un nœud. Elle se calcule par l'expression fin-deb quand nécessaire.

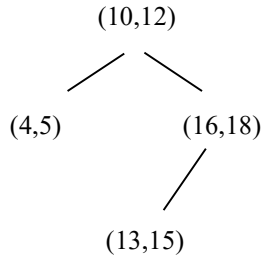
trou considéré et peut-être le fusionner avec le trou précédent[3] si son indice de fin est égal à l'indice de début de l'intervalle à insérer. Ainsi, insérer l'intervalle (18, 20) dans l'arbre (a) :



va produire l'arbre (b) suivant puisque les éléments (16,18), (18,20), (20,22) fusionnent :



Pour cela il faut que l'on puisse supprimer le trou le plus à droite (celui qui a les coordonnées les plus grandes) d'un arbre quand sa coordonnée de fin coïncide avec la coordonnée de début de l'intervalle à insérer. Ainsi dans l'arbre (a) ci-dessus, lors de l'insertion de l'intervalle (18,20), il faut supprimer le nœud (16,18) dans :



de façon à produire (c) :

L'arbre de (c) devient alors le sous-arbre gauche de la racine dans (b), tandis que la valeur 16 devient coordonnée gauche du trou à la racine.

On a pour cela besoin des procédures SUPMAX et SUPMIN (respectivement pour les cas gauche et droit).

procedure SUPMAX (**var** fdp_trou E; ent DEB; **var** ent NOUV_DEB; **var** bool RES);

---

[3]dans l'ordre des coordonnées croissantes.

qui, si DEB correspond à la coordonnée droite la plus grande dans l'arbre E, supprime le nœud correspondant et fournit en outre (NOUV_DEB) la coordonnée gauche du nœud supprimé. Le paramètre de sortie RES signale si on a supprimé un nœud (et donc si le paramètre NOUV_DEB est significatif).

Après l'ajout d'un intervalle, un nouveau nœud peut se révéler plus grand en taille que ses ancêtres. Pour conserver la propriété de maximier il va falloir “remonter” ce nœud. On propose d'opérer par rotations simples.

4. Travail à réaliser

Mettre en oeuvre l’unité sdd_fdp_trou.
Ecrire un programme utilisant cette unité et permettant de tester son bon fonctionnement.

A. Consignes à lire attentivement

A.1. Généralités

- N’utiliser **aucune variable globale** dans les procédures.
- Etre **homogène** dans l’écriture : les constantes en majuscules, les variables en minuscules, les types préfixés.
- Ne pas avoir plusieurs blocs similaires dans le programme (pas de copier/coller); **faire des procédures** (sous-problèmes).
- Utiliser des **noms explicites** de variables et de procédures.
- **Indenter** correctement le programme pour mettre en évidence la hiérarchie des blocs.
- Expliquer les algorithmes (vos méthodes pour traiter les différents problèmes) et **commenter** correctement le programme.

A.2. Dossier à remettre

Le dossier à remettre est composé de trois parties :

1. Analyse concise et structuré du programme :
   - une brève introduction présentant le contexte (sans aller jusqu’à recopier l’énoncé!);
   - un corps décrivant vos choix algorithmiques, vos décompositions en procédures
   - une conclusion qui discute des atouts de votre programme, des limites du système, des difficultés rencontrées, des optimisations éventuelles, etc…
2. Le code bien commenté.
3. Un jeu d’essais.

A.3. Programme source

- En plus de la version papier, l’ensemble de votre programme source (non compilé) ainsi qu’un fichier makefile (ou équivalent) sera envoyé en fichier texte attaché à l’enseignant de votre groupe qui le compilera et l’exécutera.
- Les programmes en plusieurs exemplaires (erreurs, oublis, modifications), ou en morceaux dans plusieurs messages iront directement à la corbeille, donc préparez bien votre mail.