Synchronisation fine

— application : ramasse-miettes à la volé —

Vocabulaire

Ramasse-miettes (*garbage collection*) = action de récupérer les emplacements en mémoire qui ne sont plus utilisés par le système, donc une mise à jour des tables d'allocation de mémoire.

Composantes du système incluant le GC :

- *Mutator*= processus qui alloue et puis change la mémoire
- *Collector*= processus qui récupère la mémoire non-utilisée
- *root*= racine de la mémoire utilisée
- *free*= racine de la mémoire libre

Types de GC (selon le moment de leur action) :

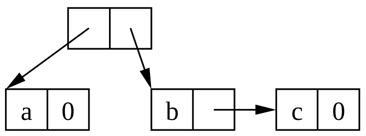
- **séquentiel** : *Collector* agit quand il n'y a plus de mémoire libre pour le *Mutator*
- **parallèle** (à la volé) : *Collector* agit en même temps que *Mutator*

Problème du ramasse-miettes à la volé (1/3)

Mémoire dynamique

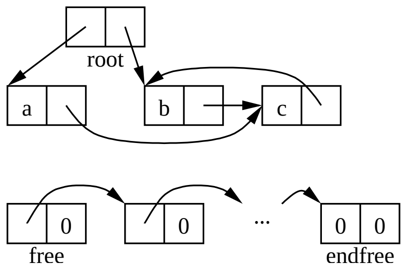
- (unique) structure de donnée dynamique = liste

  `( (a), (b, c) )`



- représentation de la mémoire dynamique :

```
int left[N+1], right[N+1];
int root, free, endfree;
```

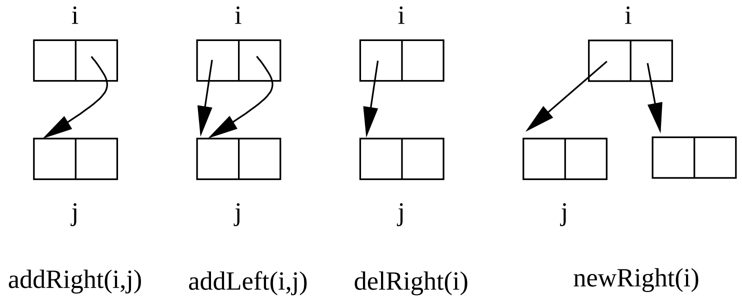


Problème du ramasse-miettes à la volé (2/3)

Actions du *Mutator*

- *addLeft(i,j)* : affecte j à la partie gauche de i
- *delLeft(i)* : affecte à 0 la partie gauche de i
- *newLeft(i)* : affecte à un nouveau noeud la partie gauche de i

et de manière similaire *addRight*, *delRight*, *newRight*.




```
active proctype Mutator () {
...
  addRight(i, j); addLeft(i, j);
  delRight(i); newRight(i)
...
}
```

Problème du ramasse-miettes à la volé (3/3)

Actions du *Collector*

1. colore tous les noeuds accessibles
2. ajoute les noeuds non-accessibles (blancs) à la liste *free*
3. remet en blancs tous les noeuds colorés

```
mtype = { white, black }; /* suffit ? */
mtype color[N+1];
active proctype Collector () {
  do :: true -> /* phase marquage */
                /* phase ramassage */
                /* phase demarquage */
  od
}
```

Choix simplificateur

FREE : left[*free*] $\neq 0$

Invariant global

GC : le noeud *endfree* n'est pas accessible du noeud *root*

Vers une solution du GC parallèle

Etat initial

```
left[0] = 0; right[0] = 0;

root = 1;
left[root] = 0; right[root] = 0;
color[root] = white;

free = 2;
i = 2;
do :: i < N ->
          left[i] = i+1; right[i] = 0;
          color[i] = white
od

endfree = N;
left[N] = 0; right[N] = 0;
color[N] = white
```

Étapes

1. Ecrire les phases du *Collector* sans se soucier de la synchronisation avec le *Mutator*, mais en assurant le prédicat de marquage correct :

$$\forall i : \quad 1 \leq i \leq N : (color[i] = black) \Rightarrow (color[left[i]] = black \ \wedge \ color[right[i]] = black)$$

2. Pareil pour les actions du *Mutator*.
3. Identifier les points d'interférence entre *Mutator* et *Collector*. (Ind. interférence entre *Collector* et l'action *addLeft* du *Mutator*.)
4. Introduire de la synchronisation (et une couleur) pour assurer le comportement correct.
5. Argumenter que les programmes écrits satisfont les invariants FREE et GC.
6. Indiquer comment la synchronisation logicielle utilisée peut être implémentée efficacement.