TD9 : Tableau dynamique

Exercice 1 : Complexité de l'extension et notion de coût amorti

En cours, nous avons fait le choix de doubler la capacité du tableau à chaque fois qu'une extension s'avère nécessaire. Aurait-on eu la même performance si l'on avait choisi d'augmenter la capacité du tableau de 10 emplacements au lieu de la doubler ?

Rappelons que dans le cas d'un doublement, le coût total d'une séquence de n insertions est de l'ordre de 3n. Ce qui nous donne un coût amorti (moyen) de 3 opérations par insertion (cf. CM).
Calculons le coût total d'une séquence de n insertions dans le cas d'une augmentation de 10 emplacements. Un ajout sur 10 va conduire à une extension et donc une recopie des k éléments déjà stockés dans le tableau, coût = k+1 affectations d'éléments (le +1 étant l'élément à ajouter), tandis que les 9 suivants se feront simplement (une seule affectation pour l'élément à ajouter).
Si on part avec un tableau de taille 1 comme précédemment, alors on réalisera une extension pour faire le 2ème ajout (passage de 1 à 11 emplacements), le 12ème ajout (passage de 11 à 21 emplacements), le 22ème ajout (passage de 21 à 31 emplacements), etc.

Nb d'élts placés	Extensions aux indices	Nb extension i	Nb opérations
1	-	0	1 affectation
2	1→10	1	1 copie + 1 aff.
3	1→10	1	1 aff.
…	…	…	…
11	1→10	1	1 aff.
12	1→10 et 11→ 20	2	11 copies + 1 aff.
13	1→10 et 11→ 20	2	1 aff.
…	…	…	…
21	1→10 et 11→ 20	2	1 aff.
22	1→10 et 11→ 20 et 21→30	3	21 copies + 1 aff.
23	1→10 et 11→ 20 et 21→30	3	1 aff.
…	…	…	…
n		i = E((n-2)/10) + 1	

Soit q=E((n-2)/10) où E() correspond à la partie entière.
Lorsque l'on a besoin d'une nouvelle extension i, on fait la recopie de 10(i-1)+1 éléments et 1 affectation (+ allocation de 10 éléments, non comptabilisée ici). Le nombre d'extensions nécessaires est q+1. Lorsque l'on n'a pas besoin d'extension, on fait 1 affectation. Donc on fait au total n affectations et q+1 copies.

$C = n + \left[\sum_{i=1}^{q+1}(10(i-1)+1)\right] = n + 10\left[\sum_{i=1}^{q+1} i\right] - 9(q+1) = n + 10\frac{(q+1)(q+2)}{2} - 9(q+1)$
$C = 5(q+1)(q+2) - 9(q+1) + n = (q+1)(5(q+2)-9) + n$
$C = (q+1)(5q+1) + n$ où $q = \left\lfloor \frac{n-2}{10} \right\rfloor$
$C = \left(\frac{n-2}{10} + 1\right) \times \left(\frac{5(n-2)}{10} + 1\right) + n$ où / représente la division entière.
$C = \frac{1}{20}n^2 + \frac{7}{5}n$

On obtient donc un coût total en $O(n^2)$ pour les n insertions, donc un coût amorti par insertion en $O(n)$. Le temps d'exécution des n ajouts serait donc moins bon que dans le cas précédent. Ainsi, lorsqu'une extension est nécessaire, il est préférable de doubler la capacité du tableau plutôt que de l'augmenter de 10 emplacements.

Exercice 2 : Crible d'Eratosthène

a. On se propose de calculer tous les nombres premiers plus petits qu'un entier $n > 1$ donné. La méthode consiste à calculer pas à pas ces nombres en utilisant la règle suivante : si un entier k n'est divisible par aucun nombre premier plus petit que k alors il est lui-même premier. Quelles sont les structures de données qu'on peut utiliser pour résoudre ce problème ? Quelle est la plus efficace ?

On peut soit réserver un tableau de taille n, soit utiliser un tableau dynamique avec la méthode du doublement de capacité vue en cours.

Pour se convaincre qu'il vaut mieux utiliser un tableau dynamique, voici un tableau représentant différentes valeurs de n, le cardinal $P(n)$ des nombres premiers plus petits que n et le ratio $P(n)/n$:

n	$P(n)$	ratio
10	4	40%
10^2	25	25%
10^3	168	16,8%
10^4	1229	12,29%
10^5	9592	9,59%
10^6	78498	7,84%
10^7	664579	6,64%

On gaspillerait donc beaucoup de mémoire en utilisant un tableau de taille fixe égale à n. Mieux vaut utiliser une structure de données extensible automatiquement en fonction des besoins : elle s'agrandira automatiquement au fur et à mesure que l'on calculera les nombres premiers.

b. Ecrire en C++ la procédure **eratosthene** qui calcule les nombres premiers plus petits que n passé en paramètre.

```
/* Préconditions : t est vide et n > 1
   Postconditions : t contient les nombres premiers <= n */
void eratosthene(TableauDynamique & t, unsigned int n) {
   int i, k;
   bool divisible;
   t.ajouterElement(2);
   for (i=3; i<=n; i++) {
      k = 0;
      divisible = false;
      while ((k < t.taille_utilisee) && !divisible) {
         if ((i % t.valeurIemeElement(k)) == 0)/* i divisible par k => pas premier*/
            divisible = true;
         else k++;
      }
      if (!divisible) /* i n'est divisible par aucun nb premier < i */
         t.ajouterElement(i);
   }
}
```

Exercice 3 : Recherche dichotomique dans un tableau initialement trié

a. Ecrire en C++ la fonction membre qui renvoie l'indice, de l'élément e passé en paramètre, dans un tableau trié. Cette recherche se fera de façon dichotomique.

```
int TableauDynamique::rechercheDicho(ElementTD e) const {
  return rechercheDansBornes(e,0,taille_utilisee-1);
}

int TableauDynamique::rechercheDansBornes(ElementTD e, int idep, int ifin) const {
   if (ifin < idep) return -1;
   int ind_milieu = (ifin+idep)/2;
   if (ad[ind_milieu] == e) return ind_milieu;
   if (e < ad[ind_milieu]) return rechercheDansBornes(e,idep,ind_milieu-1);
   return rechercheDansBornes(e,ind_milieu+1,ifin);
}
```

b. Quel est le coût d'une telle recherche ?

Au pire, on va trouver l'élément lorsque l'intervalle de recherche est de longueur 1, c'est-à-dire au bout de d subdivisions, avec $2^d = n$. C'est-à-dire $d=\log_2(n)$. Ce qui nous donne un coût en $O(\log_2 n)$.