Université Claude Bernard - Lyon 1 — Licence Sciences et Technologies - L3
Algorithmique, Programmation et Complexité - LIFAPC — 2022 - 2023
http : //liris.cnrs.fr/ ∼ rchaine/COURS/LIFAP6/

TP 4

Dans le cadre de l'UE, vous êtes amenés à programmer en **C++** en utilisant les références pour réaliser vos passages de paramètres, les constructeurs, les destructeurs et les surcharges de l'opérateur d'affectation. Néanmoins, il sera important que vous sachiez toujours basculer dans un autre langage de programmation. Pour cela, il est important que vous établissiez votre raisonnement au niveau du pseudo-langage algorithmique, en utilisant les spécificités du langage utilisé seulement quand vous effectuez la mise en œuvre.

1 Analyse de performances

Le but de ce TP est d'effectuer un enregistrement des performances de vos procédures de recherche et d'insertion dans une `Skip-List` en fonction du nombre `n` d'éléments dans la liste au moment de la requête.

- Pour cela vous créerez un nombre significatif NB_SKIP_LIST de `Skip-List`s de taille n initialisée à 0 (par exemple $NB\_SKIP\_LIST = 100$).
- Vous ferez ensuite une boucle itérative dans laquelle n sera incrémenté à chaque passage, suite à l'insertion d'un élément dans chacune des NB_SKIP_LIST `Skip-List`s distinctes. Il y a donc NB_SKIP_LIST insertions mais dans des `Skip-List`s différentes. A chaque passage dans la boucle, l'élément qui est inséré dans chacune des NB_SKIP_LIST `Skip-List`s est un élément différent, choisi aléatoirement. En calculant la moyenne du temps d'insertion sur les NB_SKIP_LIST `Skip-List`s de même taille `n`, vous obtiendrez un temps moyen d'insertion dans une liste contenant un nombre fixe `n` d'éléments. Pour obtenir le profil du temps d'insertion en fonction de `n`, il est suffisant de calculer puis d'enregistrer ce temps moyen toutes les `B` insertions.
- Les NB_SKIP_LIST `Skip-List`s ne contiennent pas les mêmes éléments mais elles ont une taille commune n qui évolue au fur et à mesure des insertions.
- Après chaque insertion de `B` éléments dans chacune des `Skip-List`s, vous effectuerez également une recherche, dans chaque `Skip-List`, d'un élément choisi aléatoirement, ce qui vous permettra de calculer la moyenne du temps requis pour faire la recherche dans chacune des `Skip-List`s (moyenne calculée sur les NB_SKIP_LIST `Skip-List`s de même taille `n`).

Vos performances (moyenne du temps nécessaire à l'insertion resp. la recherche d'un élément dans une `Skip-List` contenant `n=mB` éléments) seront enregistrées dans des fichiers `performance1.txt` et `performance2.txt` (des rappels concernant les entrées-sorties sur fichiers sont placés sur la page web du cours).

Exemple de fichier `performance.txt` :

```
# "nb element" "Temps"
1000 4.5
2000 5.1
3000 8.9
5000 10.7
```

Les fichiers ainsi écrits pourront être lus avec `gnuplot` de manière à visualiser les temps d'exécution sur un diagramme. Vous pouvez également recourir à un autre visualiseur de votre choix.

```
Tapez la commande gnuplot dans le shell
Vous obtenez l' ''invit'' suivante :
gnuplot>
```

```
Tapez-y la commande
gnuplot> plot ‘‘performance.txt’’
et vous obtiendrez un diagramme (plus d’info avec man gnuplot)
```

Exemple d’un programme utilisant la classe `std::chrono` pour faire des mesures de temps :

```
#include <chrono> //std::chrono::time_point
#include <cstdlib> /*atoi*/
#include <cstdio>

int main(int argc, char ** argv)
{
  int i,nbiter;
  if(argc!=2)
    {
      std::printf("Usage : \n %s suivi d’un entier \n
         (nombre de passages dans une boucle d’incrementation
          d’un int)\n", argv[0]);
      exit(1);
    }
  nbiter=std::atoi(argv[1]);

  std::chrono::time_point<std::chrono::system_clock> start, end;
  start = std::chrono::system_clock::now();
  for (i=1;i<=nbiter;++i) std::printf("%d %d\n",i, i*i);
  end = std::chrono::system_clock::now();
  int elapsed_microseconds
   = std::chrono::duration_cast<std::chrono::microseconds>(end-start).count();

  std::printf("Temps requis pour calculer %d carres : %d \n",
         nbiter, elapsed_microseconds);
  return 0;
}
```

2 Pousser plus loin les performances !

Si vous avez terminé d’enregistrer le profil des performances de vos `Skip-lists` (pour des valeurs de p variées), vous êtes armés pour les comparer à celles d’autres structures de données permettant de stocker une séquence triée.

2.1 Listes triées

Si vous ne l’avez pas encore terminé au TP2, mettez au point le module `ListeTriée` en modifiant le module liste chaînée du premier TP de manière à modifier son interface. L’idée était d’utiliser les listes chaînées pour y stocker des séquences triées. Du coup, les procédures d’ajout en queue et d’ajout en tête disparaissaient au profit d’une procédure d’insertion dans une liste triée. Enregistrez les performances de recherche et d’insertion d’un élément dans une `ListeTriée` de la même manière que vous l’avez fait avec les `Skip-lists`.

2.2 Comparaison des performances

Les profils que vous avez obtenus sont-ils cohérents avec les résultats vus en cours ? Vous pouvez s’il vous reste du temps établir des profils de temps d’insertion et de recherche dans un tableau dynamique trié (classe `std::vector` de la `STL`).