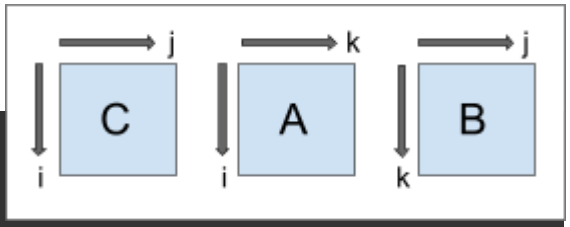
STEP Week2 宿題

2021.5.20

宿題1

行列積を求めるプログラムを書いて、行列のサイズNと実行時間の関係を調べてみよう



時間を計測したプログラムはこちら↓

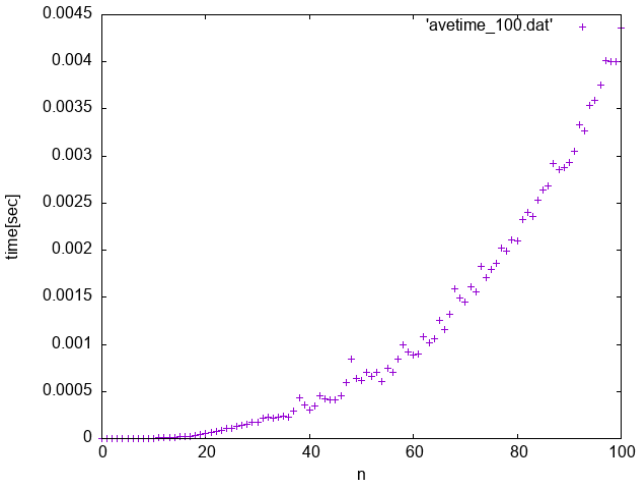
```
for(i = 0; i < n ; i++){
   for (j = 0; j < n; j++){
      for(k = 0; k < n; k++){
        c[i * n + j] += a[i * n + k] * b[k * n + j];
      }
   }
}
```

それぞれn回のfor文が入れ子になり、3重ループになっていることから、計算量はO(n^3)であると考えられる。

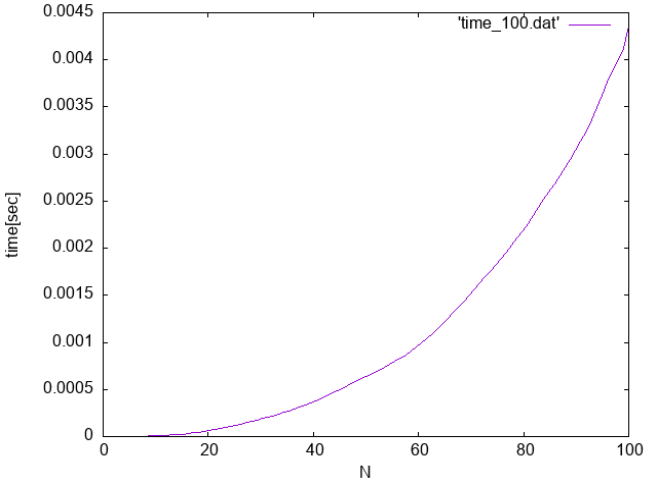
計測結果

N=100までと、N=1000までのグラフを描画した。実行時間は、10回計測した平均をとっている。

N=100までNを1ずつ大きくしていったときの実行時間との関係

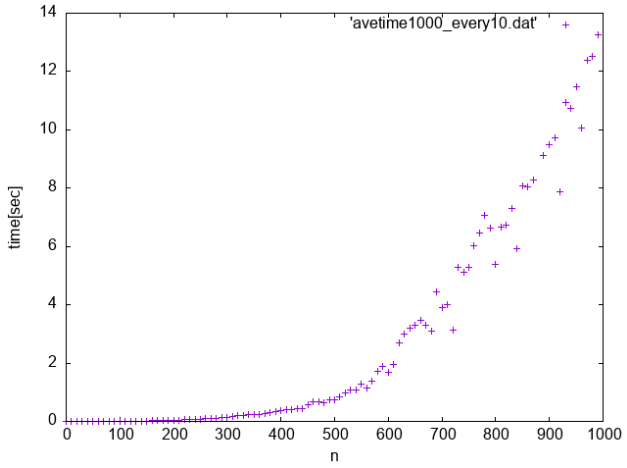


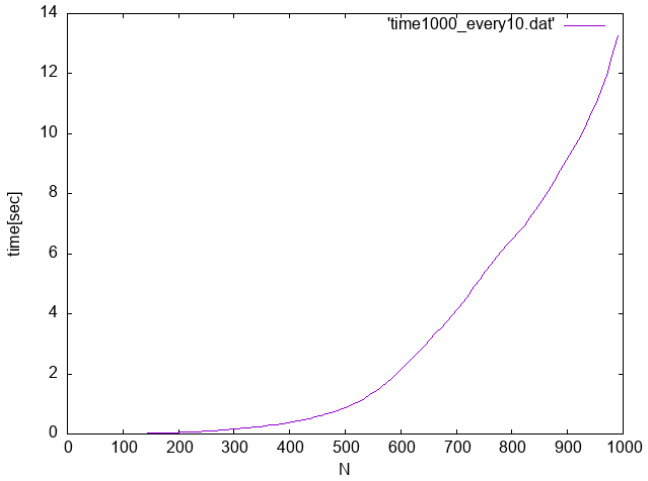
データをそのまま点でプロット



ベジェ曲線で補間

N=1000までNを10ずつ大きくしていったときの実行時間との関係





データをそのまま点でプロット

ベジェ曲線で補間

上のグラフから、おおよそO(n^3)で実行時間が推移していることがわかった。N=500くらいまではきれいに実行時間が増加していくが、500あたりからはかなり実行時間にばらつきがあり、なかなかきれいな曲線にならなかった。(どうしてでしょうか…)

宿題2

木構造を使えばO(log N)、ハッシュテーブルを使えばほぼO(1)で 検索・追加・削除を実現することができて、これだけ見ればハッシュテーブルのほうがはるかに優れているように見える。ところが、現実の大規模なデータベースでは、ハッシュテーブルではなく木構造が使われることが多い。その理由を考えよ。

- メモリ確保
  ハッシュテーブルは、ハッシュ値による分類のために常にX個のテーブルのメモリを確保しなければならない。入ってくるデータがあるテーブルに偏り、空のテーブルができてしまうと確保しているメモリが無駄になってしまうことがある。また、重複した際のメモリも確保しておかなければならない。木構造は、1つデータが入ってきたら1つ分のメモリを新たに確保していけば良いので無駄が少ない。

- データの分類
  ハッシュテーブルでは、データがざっくばらんに入ってしまい、データに順序をつけ

ることができない。
木構造は大小の情報が入っているため、情報抽出がしやすい。似たような性質を持ったデータで木を分類したら、分析もしやすそう。

- データの探索
  ハッシュテーブルでは最悪の場合一つのテーブルに集中してしまって計算量がO(n)になってしまうことがある。(なかなかそんなことはないと思うが...)
  木構造だと、(二分木の場合)バランスされていればO(logn)での探索が保証されている。

- 実装の手間
  ハッシュは、ハッシュ値の作成方法や衝突したときの処理について考えなくてはならない点で木よりも実装が大変だと思う。
  また、データの数が大きくなってハッシュテーブルを増やす、となったときに再構築が大変。

- 人間にやさしい
  木構造だと、探したいデータまでの道を辿ることができる。そしてそれを文字列に起こすことができる。
  (ディレクトリやファイルの位置を示すパスや、URLなど)

宿題3
キャッシュの管理をほぼO(1)で実現できるデータ構造を考えよ

キャッシュを双方向リストを組み合わせたようなデータ構造を考えた。
(追加を削除はO(1)でできそうなのですが、挿入の方法はまだ考え中です)

ハッシュの値を以下のように管理し、

H[ハッシュ値] = {
　　value : 値(今回はwebページ)
　　prev : 1つ前の順位のwebページのハッシュ値
　　next : 次の順位のwebページのハッシュ値
}

グローバル変数で

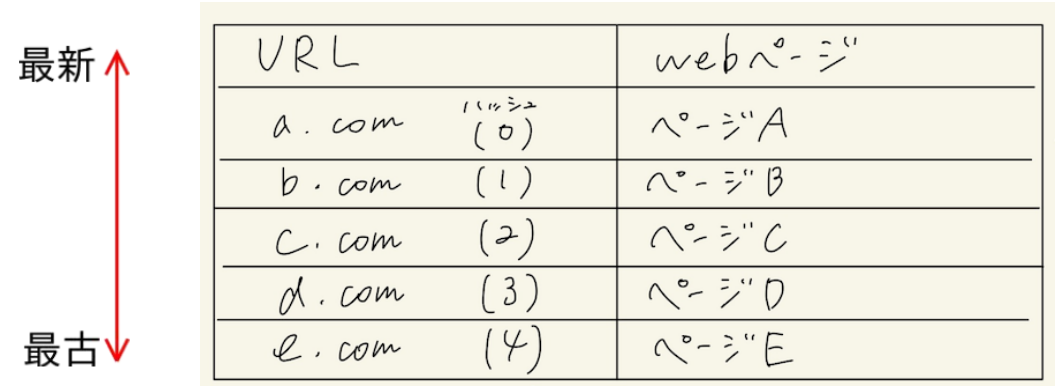
new : 最新のwebページのハッシュ値
old : 最古のwebページのハッシュ値

を記憶しておく。

ハッシュ値はURLの頭文字とする。0~25までの数字を順番に振り分ける。
a -> 0
b -> 1
c -> 2
　.
　.
z -> 25

例では、わかりやすくするためアルファベットそのままで
H[a]と表示しています

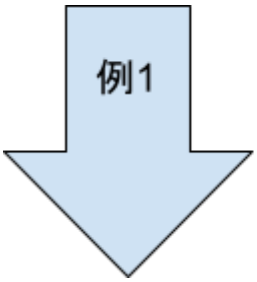
説明のため例を挙げる。
以下のようなハッシュテーブルがあるとする。aが最新、eが最古とする。



このとき、
H[a] = {value = ページA, prev = None, next = b}
H[b] = {value = ページB, prev = a, next = c}
H[c] = {value = ページC, prev = b, next = d}
H[d] = {value = ページD, prev = c, next = e}
H[e] = {value = ページE, prev = d, next = None}

new = a,
old = e

となっている。

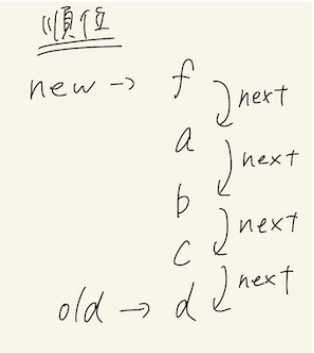


ハッシュテーブルにない
<f.com, ページF>
がアクセスされた！

ハッシュテーブルにないものがアクセスされたとき、最古のデータを消して、最新を更新したい。

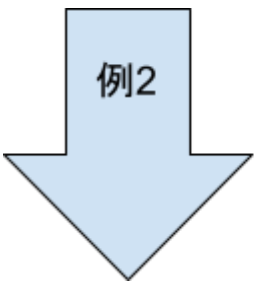
① H[f]が存在しないことを確認
② 最古のページであるH[old] (H[e])を削除する。
　・H[H[old].prev].next (H[d].next) = Noneにする。
　・old = H[old].prev (d) に更新
　・H[old] (H[e])の値を削除
③ 新しいデータを追加
　H[f] = {value = ページF, prev = None, next = a}を追加
④ H[new].prev (H[a].prev) = f に更新
⑤ new = f に更新

このとき、
H[a] = {value = ページA, prev =f, next = b}
H[b] = {value = ページB, prev = a, next = c}
H[c] = {value = ページC, prev = b, next = d}
H[d] = {value = ページD, prev = c, next = None}
H[f] = {value = ページF, prev = None, next = a} 追加
~~H[e] = {value = ページE, prev = d, next = None}~~ 削除



new = f, old = d

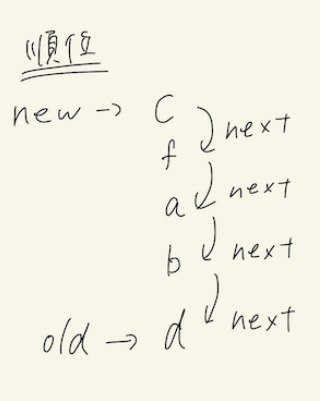
となる。



<c.com, ページC>
が再びアクセスされた！

最新のデータを更新したい。

① H[c]がハッシュテーブルに存在し、new = c でないことを確認
　(newならハッシュテーブル変更なし)
② H[H[c].prev].next (H[b].next) を H[c].next (d) に変更
③ H[H[c].next].prev (H[b].prev) を H[c].prev (b) に変更
④ H[new].prev (H[f].prev) を c に変更
⑤ H[c] = {value = ページC, prev = None, next = f}に更新
⑥ new = c に変更



このとき、
H[a] = {value = ページA, prev = f, next = b}
H[b] = {value = ページB, prev = a, next = d}
H[c] = {value = ページC, prev = None, next = f}
H[d] = {value = ページD, prev = b, next = None}
H[f] = {value = ページF, prev = c, next = a}

new = c, old = d

となる。
これで、追加と削除はO(1)でできると考えた。

挿入の問題点

挿入をO(1)でする方法が思いつかなかった。
上位X個のデータが保存できるX個に保たれたハッシュテーブルを作りたいので、削除されたデータがあればそこに新しいデータが入ってくれたらいいなと思ったが、ハッシュ値をうまく作る方法が浮かばなかった。

上の例だと、ハッシュテーブルは5個で、最初a=0, b=1, c=2, d=3, e=4というハッシュ値で管理している。仮に(頭文字の数字mod5)でハッシュ値を設定するとする。eを削除してfを入れるとき、4というハッシュ値のところが空になるが、fのハッシュ値は(5mod5=0)となり、aと衝突する。
衝突したハッシュ値のところでリストを追加したり、新たにハッシュを追加することも考えたが、それだとprevやnextに入れる情報を考え直さなければならない。(リストだとアクセスするのに線形時間かかる...?)

入るデータの数がわかっているときに(今回はX個)、ハッシュ値が衝突した際の対処法としてオープンアドレス法というものを見つけた。何か使えそうな気がしたので、もう少し挿入がO(1)になるような構造を考えていきたい。

C++のmapやdictionaryを使えばできるかもしれない？