プログラム設計とアルゴリズム
第5回 (10/25)

早稲田大学高等研究所　講師
福永津嵩

(前回の復習) 二分探索法

- 探索範囲を半減させていくことで解を発見する手法

問題:
初対面のAさんの年齢を当てたいと考えます。
Aさんの年齢が20歳以上36歳未満であるとわかっているとします。

Aさんに「Yes/Noで答えられる質問」を4回まで出来るとした時、
あなたはこの年齢当てゲームで勝つ事が出来るでしょうか？

(前回の復習) ペア和のK以上の中での最小値

問題:
N個の整数 a_i (i = 0, 1, ⋯ N−1)と、N個の整数b_i (i = 0, 1, ⋯ N−1)
が与えられ、２個の整数列から1つずつ整数を選んで和を計算する
ものとする。その和の値のうち、K以上の範囲での最小値を求めなさい。

- 以前全探索で扱った問題を再考する。

  全探索では$O(N^2)$であったが、二分探索を活用することで
  実は$O(N \log N)$で解く事が出来る。

(前回の復習) ペア和のK以上の中での最小値

- まず、bをソートする。
  第7回で講義するが、この計算量はO(N log N)となる。

- あるa_iが固定されていると考えると、問題は次のようになる。

> 問題:
> N個の整数b_i ($i = 0, 1, \cdots N-1$)が与えられる。そのうち、
> $K-a_i$以上の範囲での最小値を求めなさい。

これは、lower_bound()関数を用いる事でO(log N)で解く事が出来る。

- aの値はN個あるので、計算量は全部でO(N log N)となる。

(前回の復習) 貪欲法

- 今後のことは考えず、目の前にある選択肢の中から最も良さそうなものを選択していくことを繰り返す手法を貪欲法と呼ぶ。

例題:
500円玉、100円玉、50円玉、10円玉、5円玉、1円玉がそれぞれa_0、a_1、a_2、a_3、a_4、a_5枚あったとする。X円を支払う時、最も支払うコインの枚数が少なくなるような組み合わせを求めよ。

- この問題に対する貪欲法は、最も額が大きいコインから払えるだけ払っていくことである。例えばX = 1299円で各コインは十分にあるとすると、
1299 = 500*2 + 100*2 + 50*1 + 10*4 + 5*1 + 1*4となり、14枚となる。
実際にこれが最適である。

(前回の復習) 区間スケジューリング問題

問題:
N個の仕事があり、各仕事は時刻s_iに始まりt_iに終わる。この時、最大何個の仕事をする事が出来るかを求めなさい。

図7.2

(前回の復習) 区間スケジューリング問題

- 「開始時間が早い順番に仕事を選んでいく」という貪欲法が考えられるが、「最初に始まるが最後に終わる仕事」などを考えると明らかに正しい方法ではない。

- 「終了時間が早い順番に仕事を選んでいく」という貪欲法が適切。すなわち、

  1. 終了時刻に基づいて仕事をソートする。
  2. 一番終了時間の速い仕事を選ぶ。その仕事と重なっている他の仕事は受けられないので削除する。
  3. 残っているうち、一番終了時間の速い仕事を選ぶ。これを繰り返す。

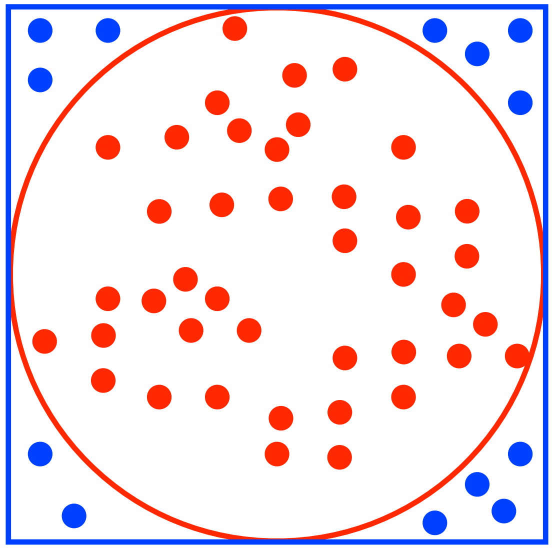
(前回の復習) 乱数と乱択アルゴリズム

- 乱択アルゴリズムとは、乱数を利用して確率的な挙動を示すアルゴリズムのことである。特に、正しい解(あるいはその近似解)を高い確率で得られるが、間違える可能性があるアルゴリズムをモンテカルロ法と呼ぶ。

- 乱数列とは、乱数の列のことであり、平たく言えば次に来る数字がランダムであり全く予測できない数列の事である。

- しかしコンピュータは、決定的な計算だけしか行う事ができないので全く規則のない数列を作る事はできない。よって真の乱数列は作れない。そのため、擬似乱数列を作って乱数とみなしている。

(前回の復習) 円周率の計算



- [-1, 1]から乱数を2つ取り、
  それをある点のx座標、y座標とみなす。
  このような点をN個取る。

- N個の点のうち、原点を中心とする半径が1の円
  の中に何個の点が入っているのかを調べる。(原点との距離が
  1以下であるかを調べる。) 入っていた個数をM個とする。

- 面積比を考えると、Nが十分に大きい時に$M/N = \pi/4$に従う。
  この事で円周率を計算できる。

(前回の復習) 行列積の計算

- N*Nの行列A, B, Cがあった時に、AB = Cであるかを判定したい。
  普通に行列積を計算した場合、$O(N^3)$かかるが、乱拓アルゴリズムによって、$O(N^2)$で計算する事が可能である。

- 要素がランダムに0または1である、長さNのベクトルxを考える。
  そしてABxとCxを計算し、これが同じ値になるかを考える。
  ABx = A(Bx)より、この計算は$O(N^2)$で行う事が出来る。

- AB = Cの時は、同じ答えになる。
  AB ≠ Cの時は、高々1/2の確率で同じ答えになる。
  少なくとも、違う答えであれば、AB ≠ Cである。

第八章
データ構造(1):
配列、リスト、ハッシュテーブル

データ構造とは

- プログラムの中で、データを保存しておく持ち方のこと

- データ構造が異なると、様々な処理において必要な計算時間が異なってくる。

- まず、1. データへのアクセス、2. データの削除、3. データの挿入、4. データの検索という処理について、配列、連結リスト、ハッシュテーブルといったデータ構造の計算時間を見てみる。

データ構造とは

表8.1

- データ構造によって、得意(可能)な処理が異なる。
- よってプログラム設計においては、どのような処理を行うかで用いるデータ構造を使い分ける必要がある。

データ構造その１：配列

- 要素を順番に並べたデータ構造。
- C++でいうvectorであり、プログラミング入門で習った配列そのもの
- a= (4, 3, 12, 7, 11, 1, 9, 8, 14, 6)とすると、
  a[0] = 4, a[1] = 3, a[2] = 12として要素にアクセス可能である。
- C/C++/Pythonの場合は、要素がメモリ上で連続に並んでいる。

図8.1

配列での要素へのアクセス

- つまり(簡略化して言うならば、)
  a[0]がメモリ上で100番目にあるなら、a[1]は101番目にある。
  よって、a[i]は100+i番目の位置にある。

- a[i]にアクセスしようと思った場合は、

  1. a[0]のメモリ上の位置を入手する。
  2. a[0]+iを計算する
  3. a[0]+i番目のメモリに存在している要素にアクセスする

  という3つのプロセスで行うことが出来る。よってその計算量はO(1)

配列での要素の削除/挿入

図8.2および図8.3

配列での要素の削除/挿入

- a = (4, 3, 12, 7, 11, 1, 9, 8, 14, 6)から7を消したいとする。
つまり、a = (4, 3, 12, 11, 1, 9, 8, 14, 6)を作りたい。

- 先ほどと同様a[0]が100番目にあるとする。7を消すとは、103番目の要素を消すことである。

- 新しい配列では、a[3] = 11となっているが、これはつまり元の配列では104番目のものを103番目に移動する必要があることを意味する。
その後ろの要素も同様に、1つずつ前の位置に移動しないければいけない。

- 配列要素数をN個とすると、その最悪計算量はO(N)となる。
特定の要素直後への挿入も同様なので、その最悪計算量はO(N)

配列での要素の検索

- a = (4, 3, 12, 7, 11, 1, 9, 8, 14, 6)の中に5があるかどうかを検索する。
- 配列は、要素がソートされて並んでいるわけではないので、二分探索などの方法は使えず、全部調べるしかない。
- そのため、検索にかかる計算量はO(N)である。(ソートしてから二分探索するとO(NlogN)であり、むしろ計算量がかかる)

データ構造その２：連結リスト

- 連結リストは配列とは違い、要素の挿入・削除に強いデータ構造である。
- リストでは、要素間の前後関係はポインタという矢印で繋がれている。
- ここで、「要素」と「次の要素を指し示すポインタ」の組をノードと呼ぶ。

図8.4

データ構造その２：連結リスト

- ポインタは、そのノードが存在するメモリ上の位置して実装されうる

- 例)

メモリ位置	要素	ポインタ
10	佐藤	30
20	高橋	40
30	鈴木	20
40	伊藤	nil

- 最初が佐藤であるとすると、この連結リストは
  佐藤→鈴木→高橋→伊藤を意味する。

連結リストでの要素へのアクセス

- 要素がメモリ上で連続して並んでいるわけではないので、
  たとえば8番目の要素にアクセスしようと思った場合には、

  0→1→2→3→・・・→7→8

  と先頭から一つずつポインタを辿っていかなければならない。

- よってその最悪計算量はO(N)である。

連結リストでの要素の挿入

図8.5

連結リストでの要素の挿入

- 例)

メモリ位置	要素	ポインタ
10	佐藤	30
20	高橋	40
30	鈴木	20
40	伊藤	nil

- 2番目に田中を入れることを考えると、次のようにすれば良い

メモリ位置	要素	ポインタ
10	佐藤	50
20	高橋	40
30	鈴木	20
40	伊藤	nil
50	田中	30

連結リストでの要素の削除

図8.6

連結リストでの要素の削除

- 例)

メモリ位置	要素	ポインタ
10	佐藤	30
20	高橋	40
30	鈴木	20
40	伊藤	nil

- 2番目の要素の削除は、次のようにすれば良い

メモリ位置	要素	ポインタ
10	佐藤	20
20	高橋	40
40	伊藤	nil

連結リストでの要素の挿入／削除／検索

- 連結リストでは、要素の挿入／削除はポインタを繋ぎかえるだけで良いため、その計算時間はO(1)となる。

- ただし、要素へのアクセスはO(N)であるため、
「i番目の要素を削除／その後に挿入」といった操作は
結局O(N)の計算時間がかかる。

- また、要素の検索は明らかにO(N)となり、配列と同等である。

- よって配列の方が利便性が高いことが多いが、連結リストが
非常に力を発揮する場面も存在する。

データ構造その3：ハッシュテーブル

- 要素の検索もO(1)で行うためのデータ構造

- まず、格納する要素がM未満の整数に限られる場合を考える。

ハッシュテーブルのアイデアを示す配列

- この方法では、まず全ての値をfalseで初期化した後、
  挿入: T[x]にtrueを代入
  削除: T[x]にfalseを代入
  検索: T[x]の値を調べる

ことで、O(1)で挿入・削除・検索を行うことが可能となる。
(ただし、要素を順番に調べると言ったようなことは苦手である)

データ構造その3：ハッシュテーブル

- 先ほどの方法を、要素がどのようなデータであっても対応できるよう拡張したものがハッシュテーブルである。

- 格納したい要素xに対して、何らかの関数h(x)を定義する。ただし、0 <= h(x) < Mを満たすものとする。

- このh(x)をハッシュ関数と呼び、xをキー、得られたh(x)の値をハッシュ値と呼ぶ。

- 異なるキーxに対して、ハッシュ値h(x)が必ず異なる値になるハッシュ関数を完全ハッシュ関数と呼ぶ。

データ構造その3：ハッシュテーブル

図8.10と表8.4

ハッシュの衝突

- 現実的なケースにおいては、完全ハッシュ関数の設計は困難であり、
  異なるキーに対して同じハッシュ値が割り当てられることがある。
  これをハッシュの衝突と呼ぶ。

- 最悪のケースは、全てのkeyが同じハッシュ値を持つときである。

- ハッシュ値が特定の値をとる確率がランダムに1/Mであり、
  そのため任意の2つのキーが特定の値をとる確率が1/Mとなるとき、
  そのようなハッシュを単純一様ハッシュと呼ぶ。

ハッシュ関数の例

- 文字列についてハッシュ関数を設計するとする。

- 最悪のハッシュ関数は定数関数h(x)=0などである。

- l(x)を文字列xの長さを得る関数として、h(x) = l(x) % Mとすると
  少しマシだがが、文字列長は短いことが多いので、
  ハッシュ値に偏りが生じていると言える。

- 次のローリングハッシュなどはよく利用されている。
  文字列 $x = c_1c_2\cdots c_n$としたとき、
  $h(x) = (c_1a^{m-1}+c_2a^{m-2}+\cdots+c_ma^0)$ % M

ハッシュの衝突対策(1):連鎖法

- 衝突した場合データを捨てるのは困るので、何らかの対策が必要になる
- 連鎖法は、ハッシュとリストを兼ね備えたデータ構造
(true/falseだけでなく要素も格納する)

図8.11

ハッシュの衝突対策(1):連鎖法

- 同一のハッシュ値を持つデータをリストとして保管する。
- ハッシュテーブルには、リストの先頭要素を指し示すポインタを格納する。
- データの検索・挿入・削除はリストでの操作と同じように行うことが可能である。

ハッシュの衝突対策(2):オープンアドレス法

- 追加のデータ構造を用いず、ハッシュテーブルの中だけで完結させる方法。
  クローズドハッシュ法とも言う。

- ハッシュに衝突が起きた場合に再ハッシュを行って、
  新たなハッシュ値の計算を行いそこに要素が存在しなければ
  データを格納する方法をいmする

- たとえばハッシュ値を(h(x)+i)%Mなどとする。(iは再ハッシュの回数)
  このような再ハッシュは線形探査法と呼ばれる。

- 他にも、もう一つハッシュ関数g(x)を用意して、
  ハッシュ値を(h(x)+i*g(x))%Mとする方法もある(二重ハッシュ法)。

オープンアドレス法での挿入

- 例) 以下、全て線形探査法で再ハッシュする。

  新たに田中を右のハッシュテーブルに追加する
  h(田中) = 3であれば、鈴木と衝突する。

  そこで再ハッシュを行う。
  (h(田中)+1)%5 = 4であり、そこに
  要素は存在しないので4番目に田中を入れる。

index	要素
0	nil
1	nil
2	nil
3	鈴木
4	nil

↓

index	要素
0	nil
1	nil
2	nil
3	鈴木
4	田中

オープンアドレス法での挿入

- 例)

新たに伊藤を右のハッシュテーブルに追加する
h(伊藤) = 3であれば、鈴木と衝突する。

そこで再ハッシュを行う。
(h(伊藤)+1)%5 = 4であるが、そうすると
今度は田中と衝突する。

そのため、２回目の再ハッシュを行う
(h(伊藤)+2)%5 = 0となり、そこはnilのため
0番目に伊藤を入れる

index	要素
0	nil
1	nil
2	nil
3	鈴木
4	田中

↓

index	要素
0	伊藤
1	nil
2	nil
3	鈴木
4	田中

オープンアドレス法での検索

- 例)
  ハッシュテーブルに田中が存在するか検索する。
  h(田中) = 3であるが、3番目は伊藤であり
  田中ではない。

  しかし存在しないと判定してはならない。

index	要素
0	伊藤
1	nil
2	nil
3	鈴木
4	田中

- 再ハッシュし続けた値がnilに到達するまで、再ハッシュを繰り返し
  存在するかどうかを確認する。
  (h(田中)+1)%5 = 4であり、4番目は田中であり、存在することが
  わかる。

オープンアドレス法での検索

- 例)
  ハッシュテーブルに山田が存在するか検索する。
  h(山田) = 0であるとする。
  0番目は伊藤であって山田ではない。よって
  検索を続ける。

index	要素
0	伊藤
1	nil
2	nil
3	鈴木
4	田中

- (h(山田)+1)%5 = 1である。
  1番目はnilである。よってハッシュテーブルには山田は存在しない。

オープンアドレス法での削除

- 例)
  要素を検索して発見した後に、それを取り除けば良い。しかし、nilにしてはいけない！

- たとえば、田中を削除してnilにしたとする。(誤り)

index	要素
0	伊藤
1	nil
2	nil
3	鈴木
4	田中

↓

index	要素
0	伊藤
1	nil
2	nil
3	鈴木
4	nil

オープンアドレス法での削除

- 例)
  田中を削除した後に、伊藤が存在するかを探索する。

- h(伊藤)=3であったが、鈴木と衝突するので再ハッシュする。

- (h(伊藤)+1)%5 = 4を見ると、そこはnilである。
  よってハッシュテーブルに伊藤は存在しない！

- オープンアドレス法でnilにすると、要素の検索においてこのような誤りが生じる。

index	要素
0	伊藤
1	nil
2	nil
3	鈴木
4	nil

オープンアドレス法での削除

- 例)
  よって、nilではなく別の記号を利用しなければならない(“deleted”など)

index	要素
0	伊藤
1	nil
2	nil
3	鈴木
4	deleted

- 探索の際にdeletedにぶつかった際には、そこで探索を終了するのではなく探索を先に進めなければならない。

- 要素の挿入においてdeletedにぶつかった場合には、そこに要素を挿入して問題ない。

第九章
データ構造(2):
スタックとキュー

スタックとキュー

- スタックとキューはどちらも、要素の追加と取り出しをサポートするデータ構造である(要素の検索などをサポートする必要はない)。

- スタックでは、要素の取り出しにおいて、最後に追加された要素が取り出される。LIFO(last-in first-out)またはFILO(first-in last-out)と呼ばれる。

- キューでは、要素の取り出しにおいて、最初に追加された要素が取り出される。FIFO(first-in first-out)と呼ばれる。

スタックの挙動

図9.3

- スタックでは要素の追加／取り出しはpush／popと呼ばれる。

キューの挙動

図9.4

- キューでは要素の追加／取り出しはenqueue /dequeueと呼ばれる。

スタックとキューの操作例

- 空のスタックと空のキューがあり、それぞれに要素A, B, Cをこの順番で追加した。この時、スタックとキューの状態は次のように書くとする。

  スタック：ABC
  キュー　：ABC

- この状態において、スタックから要素を取り出し、取り出した要素をキューに追加した。その後、キューから要素を取り出し、取り出した要素をスタックに追加した。

- 再び、スタックから要素を取り出し、取り出した要素をキューに追加し、またキューから要素を取り出し、取り出した要素をスタックに追加した。この時、スタックとキューの状態はどのようになっているか？

スタックとキューの操作例

- スタック：ABC
  キュー　：ABC

- スタックからの要素を取り出すとCなので、それをキューに追加すると

  スタック：AB
  キュー　：ABCC

- キューから要素を取り出すとAなので、それをスタックに追加すると

  スタック：ABA
  キュー　：BCC

スタックとキューの操作例

- スタック：ABA
  キュー　：BCC

- スタックからの要素を取り出すとAなので、それをキューに追加すると

  スタック：AB
  キュー　：BCCA

- キューから要素を取り出すとBなので、それをスタックに追加すると

  スタック：ABB
  キュー　：CCA

スタックの実装

```cpp
const int MAX = 100000; // スタック配列の最大サイズ

int st[MAX]; // スタックを表す配列
int top = 0; // スタックの先頭を表す添字

// スタックを初期化する
void init() {
    top = 0; // スタックの添字を初期位置に
}

// スタックが空かどうかを判定する
bool isEmpty() {
    return (top == 0); // スタックサイズが 0 かどうか
}

// スタックが満杯かどうかを判定する
bool isFull() {
    return (top == MAX); // スタックサイズが MAX かどうか
}

// push
void push(int x) {
    if (isFull()) {
        cout << "error: stack is full." << endl;
        return;
    }
    st[top] = x; // x を格納して
    ++top; // top を進める
}
```

スタックの実装

```
// pop
int pop() {
    if (isEmpty()) {
        cout << "error: stack is empty." << endl;
        return -1;
    }
    --top; // top をデクリメントして
    return st[top]; // top の位置にある要素を返す
}

int main() {
    init(); // スタックを初期化

    push(3); // スタックに 3 を挿入する {} -> {3}
    push(5); // スタックに 5 を挿入する {3} -> {3, 5}
    push(7); // スタックに 7 を挿入する {3, 5} -> {3, 5, 7}

    cout << pop() << endl; // {3, 5, 7} -> {3, 5} で 7 を出力
    cout << pop() << endl; // {3, 5} -> {3} で 5 を出力

    push(9); // 新たに 9 を挿入する {3} -> {3, 9}
}
```

キューの実装の注意点

- スタックの実装に必要な変数はtopのみで、先ほどの例で言えば、
  要素の追加／取り出しに応じて、データ構造の右端のみが変化する。

- 一方で、キューの実装に必要な変数はheadとtailの2つがあり、
  要素の追加／取り出しに応じて、データ構造の両端が変化する。

- 配列で実装することを考えると、追加／取り出しを繰り返すと
  headもtailも右側にずれていってしまい、格納サイズに応じて
  不必要なメモリサイズになる。

- これを解決する方法が、リングバッファである。
  または、配列ではなくリストを使って管理しても良い。

リングバッファ

図9.5

- 配列の先頭と終端は隣接していると考えて実装する

キューの実装

```
const int MAX = 100000; // キュー配列の最大サイズ

int qu[MAX]; // キューを表す配列
int tail = 0, head = 0; // キューの要素区間を表す変数

// キューを初期化する
void init() {
    head = tail = 0;
}

// キューが空かどうかを判定する
bool isEmpty() {
    return (head == tail);
}

// キューが満杯かどうかを判定する
bool isFull() {
    return (head == (tail + 1) % MAX);
}

// enqueue
void enqueue(int x) {
    if (isFull()) {
        cout << "error: queue is full." << endl;
        return;
    }
    qu[tail] = x;
    ++tail;
    if (tail == MAX) tail = 0; // リングバッファの終端に来たら 0 に
}

```

キューの実装

```
// dequeue
int dequeue() {
    if (isEmpty()) {
        cout << "error: queue is empty." << endl;
        return -1;
    }
    int res = qu[head];
    ++head;
    if (head == MAX) head = 0; // リングバッファの終端に来たら 0 に
    return res;
}

int main() {
    init(); // キューを初期化

    enqueue(3); // キューに 3 を挿入する {} -> {3}
    enqueue(5); // キューに 5 を挿入する {3} -> {3, 5}
    enqueue(7); // キューに 7 を挿入する {3, 5} -> {3, 5, 7}

    cout << dequeue() << endl; // {3, 5, 7} -> {5, 7} で 3 を出力
    cout << dequeue() << endl; // {5, 7} -> {7} で 5 を出力

    enqueue(9); // 新たに 9 を挿入する {7} -> {7, 9}
}
```

まとめ

- 基本的なデータ構造として、配列／連結リスト／ハッシュテーブル／スタック／キューを紹介した。
- データ構造は、要素の挿入／削除／検索などにおいて、得意不得意があり、用途に応じて適切なデータ構造を利用することが必要である。
- ハッシュテーブルは挿入／削除／検索全てにおいてO(1)となるデータ構造だが、ハッシュ値の衝突を考慮しなければならない。
その方法として、連鎖法／オープンアドレス法などがある。