平成 29 年 5 月 23 日

知的システム構成特論

Data Structures and Algorithms, Chapter 12 (pp.379-410)

メモリ管理

西田研究室 17344219 二宮 悠二

12.4 不同サイズブロックの記憶割り当て

この章の最初で話したヒープの管理について考える．ブロックの割り当てにはポインタの集まりを用いる．ブロックには何らかの型のデータを入れる．1 節で用いたデータは文字列であった．ヒープに入れるデータは文字列でなくても良いが，ここではデータの中にヒープ内の番地へのポインタは含まれていないものとする．

ヒープ管理の利点としては，前節で扱った同一サイズのセルのリスト構造の保守と比べると，使用中のブロックに印をつける作業が非再帰的に行えるということが挙げられる．また，外部からヒープへのポインタを見て，指されているブロックに印をつければ良いので連結構造を深さ優先で探索したり，する必要はない．

一方で，使用可能スペースリストの管理が比較的難しい．

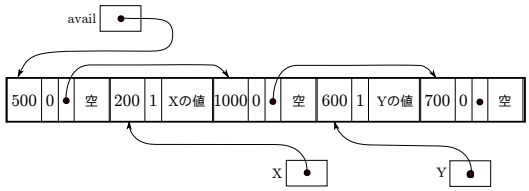

Fig. 1: ヒープによる管理

新しいデータを入れるための空ブロックを探したり，不要になったデータが入っているブロックを再利用したりできるように本節ではブロックに関して次のように仮定する．

(1) 各ブロックは次のデータを格納するだけの大きさがあるものとする．

- **(a)** ブロックの大きさ
- **(b)** 使用可能なブロックをつなぐポインタ
- **(c)** ブロックが空かどうかを示すビット

(2) 空ブロックは左から，ブロックの長さ，0 の Full/Empty ビット，次の使用可能ブロックへのポインタ，空領域という形をしている．

(3) データを保有しているブロックは左から，長さ，1 の Full/Empty ビット，データ自身という形をしている．

こうした仮定から，使用中はデータを入れることができ，使用してないときは同じ場所にポイントを入れる必要がある．

12.4.1 断片化と空ブロックの詰め直し

ヒープ管理で生ずる特別な問題を調べるため，**Fig.**1 で変数 Y の値が変わり，それを表していたブロックを使用可能スペースにしたいときを考える．このブロックは，使用可能スペースリストの先頭に入れるのが最も楽である．

断片化とは，使用可能リスト上の大きな領域が小さなブロックの集まりになってしまう傾向をいう．こうして空になったブロックを隣接する空ブロックとまとめることで断片化をおさえるには次の 3 つが挙げられる．

(1) ブロックが空くたびに隣接空領域をまとめる．

(2) ブロックが空くたびに，双方向の使用可能スペースリストに登録する．

(3) 隣接空領域をまとめずにそのままにしておく．

12.4.2 使用可能ブロックの選択

新しいデータを格納するには，新たにブロックを提供する必要がある．そのためには，使用可能なブロックから 1 つを選び，その一部，または全部に新しいデータを入れる必要がある．これらの選択に際して，データを入れるブロックをなるべく早く，かつ断片化が進まないようにしたい．そのために以下の 2 つの方法がある．

(1) 初適合法：最初に見つけたブロックを選択する．

(2) 最良適合法：ブロックの中で最小のブロックを選択する．

(1) の方法は (2) の方法に比べ素早く決められる．(2) の方法は処理時間が圧倒的に遅いが，断片化を抑えられる．断片の数はあまり変わらないが，占める領域は小さい．ただし，これらの方法は中程度の大きさの断片をあまり作らないため，(1) なら応じられるものでも (2) では応じられないという傾向がある．その逆もしかりである．

12.5　バディシステム

断片化を防ぎながら，空ブロックの大きさの分布がひどくならないようにするデータを配置する方法であり，隣接する空ブロックを短時間でまとめられる．一方で，ブロックの大きさが限られるため，必要以上の大きさのブロックを必要としてしまうことからスペースの無駄遣いが生じる．

バディシステムの考え方は，ブロックの大きさをいくつかの種類に制限することである．

$s_1 < s_2 < s_3 < \cdots < s_k$ という k 通りの大きさのブロックだけを使うものとする．この中から新しいデータが入る最小のブロックを使う．このときの大きさを s_i とすると，この大きさの空ブロックがない場合は大きさ s_{i+1} のブロックを探し，これを s_i と $s_{i+1} - s_i$ の 2 つのブロックに分割する．ある $k \geq 0$ に対して $j = i - k$ であるとすると，$s_{i+1} - s_i = s_{i-1}$ より

$$s_{i+1} = s_i + s_{i-k}$$

が得られる．これを k 次のバディシステムという．特に，$k = 0$ のときを指数型バディシステム，$k = 1$ のときをフィボナッチバディシステムと呼ぶ．

12.5.1　ブロックの分布

k 次のバディシステムでは，大きさ s_{i+1} のブロックは，s_i と s_{i-k} の 2 つのブロックからできていると考えられる．ここで，大きさ s_i のブロックが左にあり，大きさ s_{i-k} が右側にあるとすると，ヒープ全体がある大きな n に対する 1 つのブロック s_n であると考えられ，大きさ s_i のブロックが始まる位置が決定される．

12.5.2　ブロックの割り当て

大きさ n のブロックが必要になった場合，使用可能スペースリストから，$n \leq s_i$ かつ $i = 1$ または $s_{i-1} < n$ を満たす満たす大きさ s_i のブロックをどれか 1 つ選び使用する．このような大きさのブロックがなければ，大きさ s_{i+1} か s_{i+k+1} のブロックを選んで分割すると，大きさ s_i のブロックができる．そのようなブロックもなければ，$s = i + 1$ の大きさについてこの分割法を再帰的に適用すればよい．

12.5.3　ブロックの再利用

ブロックが再度使用できるようになったとき，新たに利用できるようになったブロックのバディも使用可能ならば，この 2 つをまとめることにより断片化を防ぐことができる．こうしてできたブロックがまた使用可能ならば，この 2 つもさらにまとめられ，この手順を繰り返すことができる．

指数型のバディシステムでは，バディの位置が簡単に分かるが，$k \geq 1$ の場合は容易ではない．そのため，各ブロックに以下の情報を格納しておく．

(1) Full/Empty ビット

(2) サイズインデックス：ブロックの大きさが s_i であることを示す．

(3) 左のバディカウント：そのブロックが何回連続で左側バディになるかを示す．

12.6　記憶の詰め直し

空ブロックをまとめたとしても，新たな要求に応じられない場合がある．この問題の解決策として次の 2 つが挙げられる．

(1) いくつかの空ブロックをポインタでつないで使用する．

(2) ヒープ内でデータを移動して使用中のブロックを左側に集め，右側に使用可能ブロックを 1 つにする．

(1) の方法はブロックをつなぐポインタ分にスペースを取られてしまうため，スペースの無駄遣いが起こりやすい．一方で，比較的簡単に実現できるため，ほとんどのファイルシステムはこの方法を採用している．(2) の方法は，効果は高いが，使用可能スペースをまとめるためにポインタの更新が必要であり多くの処理時間がかかる．また，使用中のブロックの移動にはそのヒープの大きさに比例した時間がかかり，詰め直しのコストの中で最も大きな部分を占める．

12.6.1　モリスのアルゴリズム

ブロック内に移動先用のスペースを取らずにヒープを詰め直す方法である．使用中の各ブロック内の一定位置から始まるポインタの連鎖を作り，そのブロックを指しているポインタをすべてその連鎖につなげる．