関数型言語（OCaml 演習）(15)

ヒープ

浅井 健一

お茶の水女子大学

ヒープ

木を配列に埋め込んだデータ構造を考える。

<table>
<tr><td colspan="8">0</td></tr>
<tr><td colspan="4">1</td><td colspan="4">2</td></tr>
<tr><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">5</td><td colspan="2">6</td></tr>
<tr><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td></td><td></td><td></td></tr>
</table>

上の配列を a とすると

- 木の根は a.(0)
- a.(i) の左右の子は a.($2i+1$) と a.($2i+2$)
- a.(i) の親は a.($(i-1)/2$)

配列を使うと、親や子供に定数時間でアクセスできる。

ヒープの満たすべき性質

各頂点の値は、どちらの子供の頂点の値よりも小さい

<table>
<tr><td colspan="8">A:2</td></tr>
<tr><td colspan="4">B:3</td><td colspan="4">C:8</td></tr>
<tr><td colspan="2">D:11</td><td colspan="2">E:4</td><td colspan="2">F:10</td><td colspan="2">G:9</td></tr>
<tr><td>H:15</td><td>I:12</td><td>J:11</td><td>K:14</td><td>L:13</td><td></td><td></td><td></td></tr>
</table>

ヒープの特徴

- 最小の要素を取ってくるのは定数時間。
- 常にバランスしている。
- 値の挿入、削除、変更に $O(\log n)$ 時間。

ヒープへの挿入

1. 新しい要素を配列の最後に付け加える。
2. その要素とその親とを比べ、自分の方が小さかったら親と入れかえる。
3. これをヒープの条件が満たされるまで繰り返す。

<table>
<tr><td colspan="8">A:2</td></tr>
<tr><td colspan="4">B:3</td><td colspan="4">C:8</td></tr>
<tr><td colspan="2">D:11</td><td colspan="2">E:4</td><td colspan="2">F:10</td><td colspan="2">G:9</td></tr>
<tr><td>H:15</td><td>I:12</td><td>J:11</td><td>K:14</td><td>L:13</td><td>M:7</td><td></td><td></td></tr>
</table>

ヒープへの挿入

1. 新しい要素を配列の最後に付け加える。
2. その要素とその親とを比べ、自分の方が小さかったら親と入れかえる。
3. これをヒープの条件が満たされるまで繰り返す。

<table>
<tr><td colspan="8">A:2</td></tr>
<tr><td colspan="4">B:3</td><td colspan="4">C:8</td></tr>
<tr><td colspan="2">D:11</td><td colspan="2">E:4</td><td colspan="2">M:7</td><td colspan="2">G:9</td></tr>
<tr><td>H:15</td><td>I:12</td><td>J:11</td><td>K:14</td><td>L:13</td><td>F:10</td><td></td><td></td></tr>
</table>

ヒープへの挿入

1. 新しい要素を配列の最後に付け加える。
2. その要素とその親とを比べ、自分の方が小さかったら親と入れかえる。
3. これをヒープの条件が満たされるまで繰り返す。

<table>
<tr><td colspan="8">A:2</td></tr>
<tr><td colspan="4">B:3</td><td colspan="4">M:7</td></tr>
<tr><td colspan="2">D:11</td><td colspan="2">E:4</td><td colspan="2">C:8</td><td colspan="2">G:9</td></tr>
<tr><td>H:15</td><td>I:12</td><td>J:11</td><td>K:14</td><td>L:13</td><td>F:10</td><td></td><td></td></tr>
</table>

最大で木の深さ回（$O(\log n)$）で終了する。

最小の要素の削除

1. 配列の最初の要素（根の要素）を削除し、そこに配列の最後の要素を移動する。
2. その要素とその子を比べ、自分が最小でなかったら、小さい方の子と入れかえる。
3. これをヒープの条件が満たされるまで繰り返す。

<table>
<tr><td colspan="8">A:2</td></tr>
<tr><td colspan="4">B:3</td><td colspan="4">M:7</td></tr>
<tr><td colspan="2">D:11</td><td colspan="2">E:4</td><td colspan="2">C:8</td><td colspan="2">G:9</td></tr>
<tr><td>H:15</td><td>I:12</td><td>J:11</td><td>K:14</td><td>L:13</td><td>F:10</td><td></td><td></td></tr>
</table>

最小の要素の削除

1. 配列の最初の要素（根の要素）を削除し、そこに配列の最後の要素を移動する。
2. その要素とその子を比べ、自分が最小でなかったら、小さい方の子と入れかえる。
3. これをヒープの条件が満たされるまで繰り返す。

F:10							
B:3				M:7			
D:11		E:4		C:8		G:9	
H:15	I:12	J:11	K:14	L:13			

最小の要素の削除

1. 配列の最初の要素（根の要素）を削除し、そこに配列の最後の要素を移動する。
2. その要素とその子を比べ、自分が最小でなかったら、小さい方の子と入れかえる。
3. これをヒープの条件が満たされるまで繰り返す。

B:3							
F:10				M:7			
D:11		E:4		C:8		G:9	
H:15	I:12	J:11	K:14	L:13			

最小の要素の削除

1. 配列の最初の要素（根の要素）を削除し、そこに配列の最後の要素を移動する。
2. その要素とその子を比べ、自分が最小でなかったら、小さい方の子と入れかえる。
3. これをヒープの条件が満たされるまで繰り返す。

<table>
<tr><td colspan="8">B:3</td></tr>
<tr><td colspan="4">E:4</td><td colspan="4">M:7</td></tr>
<tr><td colspan="2">D:11</td><td colspan="2">F:10</td><td colspan="2">C:8</td><td colspan="2">G:9</td></tr>
<tr><td>H:15</td><td>I:12</td><td>J:11</td><td>K:14</td><td>L:13</td><td></td><td></td><td></td></tr>
</table>

ヒープの実装の概要

```
type ('a, 'b) t
  (* 最小値を求める値が 'a 型で
     そのほかの付加情報が 'b 型であるヒープの型 *)

val create : int -> 'a -> 'b -> ('a, 'b) t
  (* 使い方：create size key value *)
  (* ヒープのサイズとダミーの値を受け取ったら *)
  (* 空のヒープを返す *)

val insert : ('a, 'b) t -> 'a -> 'b ->
                                    index_t * ('a, 'b) t
  (* 使い方：insert heap key value *)
  (* ヒープに新しい要素を追加する *)
  (* ヒープは（破壊的に）書き変わる *)

type index_t  (* ヒープの添字の型 *)
```

ヒープの使用

```
# let heap1 = Heap.create 100 0 "" ;;
val heap1 : (int, string) Heap.t = <abstr>
# let (indexA, heap2) = Heap.insert heap1 2 "A" ;;
val indexA : Heap.index_t = <abstr>
val heap2 : (int, string) Heap.t = <abstr>
# let (indexB, heap3) = Heap.insert heap2 3 "B" ;;
val indexB : Heap.index_t = <abstr>
val heap3 : (int, string) Heap.t = <abstr>
```

ヒープが純粋なデータなら

- heap1 空のヒープ。
- heap2 A:2 のみが入ったヒープ。
- heap3 A:2, B:3 が入ったヒープ。

ヒープの使用

```
# let heap1 = Heap.create 100 0 "" ;;
val heap1 : (int, string) Heap.t = <abstr>
# let (indexA, heap2) = Heap.insert heap1 2 "A" ;;
val indexA : Heap.index_t = <abstr>
val heap2 : (int, string) Heap.t = <abstr>
# let (indexB, heap3) = Heap.insert heap2 3 "B" ;;
val indexB : Heap.index_t = <abstr>
val heap3 : (int, string) Heap.t = <abstr>
```

実際には、ヒープは内部で副作用を使っているので

- heap1 不明。
- heap2 不明。
- heap3 A:2, B:3 が入ったヒープ。

ヒープの使用

```
# let heap = Heap.create 100 0 "" ;;
val heap : (int, string) Heap.t = <abstr>
# let (indexA, heap) = Heap.insert heap 2 "A" ;;
val indexA : Heap.index_t = <abstr>
val heap : (int, string) Heap.t = <abstr>
# let (indexB, heap) = Heap.insert heap 3 "B" ;;
val indexB : Heap.index_t = <abstr>
val heap : (int, string) Heap.t = <abstr>
```

同じ名前（heap）を使うことで、常に最新のヒープのみを使うようにする。

まとめ

ヒープ

自分の値がどちらの子供の値よりも小さくなっているような木構造。配列に埋め込んで実現することが多い。

- 最小の要素を定数時間で取って来られる。
- 木は常に完全にバランスしている。
- 要素の追加、削除、変更が $O(\log n)$ でできる。

副作用を伴うデータ構造

- 副作用の使用をモジュール内のみに限定し、
- 常に最新のデータのみを使う

と、純粋なデータのように扱えることが多い。