2022年度「プログラミング言語」配布資料 (0)

五十嵐 淳

2022年10月02日

1　2分探索木

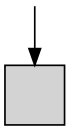
この講義を通じて使う例題として **2分探索木 (binary search tree)** をとりあげる．2分探索木はデータ構造の表現，操作の実現方法に様々なバリエーションが考えられるため，複数の言語間の比較だけでなく，ひとつの言語内での機能の比較にもよい例題になる．まずは，2分探索木について抽象的な (プログラミング言語に依存しない) アルゴリズムのレベルでの復習から始めよう．

1.1　2分木

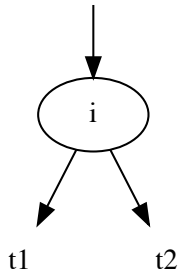
> 以下の規則で構成されるものを **2分木 (binary tree)** という．* `leaf` は2分木である．* 2分木 t_1, t_2 と整数 i に対し `branch(`t_1`,`i`,`t_2`)` は2分木である．

`leaf` と `branch(`t_1`,`i`,`t_2`)` はそれぞれ以下のように図示する．

`leaf`

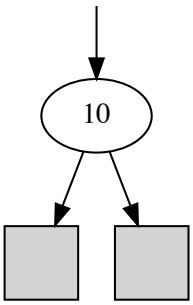


`branch(`t_1`,`i`,`t_2`)`

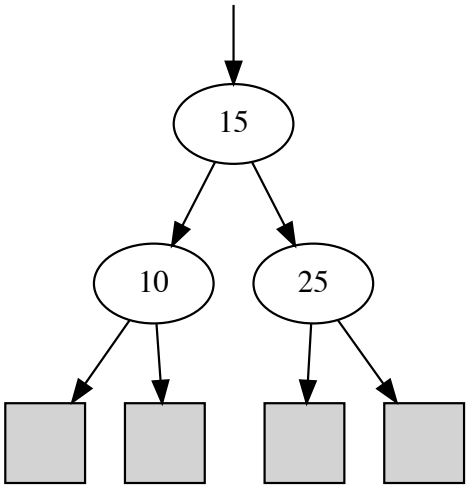


1.1.1 2 分木の例

$t_a =$ `branch(leaf, 10, leaf)`

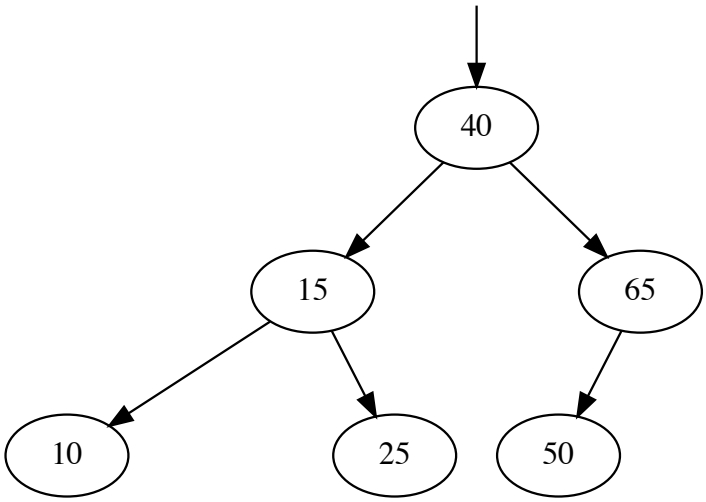


$t_b =$ `branch(`t_a`, 15, branch(leaf, 25, leaf))` $(=$ `branch(branch(leaf, 10, leaf), 15, branch(leaf, 25, leaf))`$)$



また，図を書く時には `leaf` の部分をしばしば省略する．

$t_c =$ `branch(`t_b`, 40, branch(branch(leaf, 50 leaf), 65, leaf))`



1.1.2 補足

`leaf` (葉) は子を持たないノード，`branch` は整数を格納したふたつの子 (部分木) を持つノードを表している．一番外側の `branch` が根に相当する．2 分木はグラフの特別な場合として定義されることも多い (グラフ理論ではむしろその方が標準的な定義である) が，実質，等価な定義になっていることを確認してもらいたい．(`leaf` の存在を無視してしまうのが重要である．)

また，この定義は再帰的である．すなわち，2 分木の定義中に再び 2 分木が言及されている．しかし，`leaf` の項だけみると，ここで 2 分木は言及されていないので，これを種にしてより大きな 2 分木を構成していくことができる．また，2 分木の構成法は**上に書かれた二種類しかない** ので，以下のようなことがいえる．

> 任意の 2 分木 t について，* t = `leaf` または * ある 2 分木 t_1,t_2 と整数 i が存在して t = `branch(`t_1`,`i`,`t_2`)`
>
> が成立する．

このことは，当たり前のようだが，*2* **分木に対する処理を考える時には，基本的には** *leaf* **と** *branch* **のふたつの場合さえ考えればよい**，という意味で重要である．

1.1.3 2 分木上の関数

2 分木上の関数の例として，木のサイズ (`branch` の数) を表す $size(t)$ と，深さ (根から各葉までに通過する `branch` の数の最大) を表す $depth(t)$ を考えてみよう．

1.1.3.1 具体例

$$\begin{array}{ccc} size(t_a)=1 & size(t_b)=3 & size(t_c)=6 \\ depth(t_a)=1 & depth(t_b)=2 & depth(t_c)=3 \end{array}$$

1.1.3.2 $size$, $depth$ の (再帰的) 定義

$$\begin{array}{lcl} size(\texttt{leaf}) & = & 0 \\ size(\texttt{branch(}t_1\texttt{,}i\texttt{,}t_2\texttt{)}) & = & size(t_1)+size(t_2)+1 \\ depth(\texttt{leaf}) & = & \boxed{??} \\ depth(\texttt{branch(}t_1\texttt{,}i\texttt{,}t_2\texttt{)}) & = & \boxed{??} \end{array}$$

2 分木が再帰的に定義されているため，2 分木上の関数は **自然に**再帰的に定義されることになる[1]．

1.2 2 分探索木

2 分探索木 (binary search tree) は，大小関係があるようなデータ (正確には全順序集合 (totally ordered set) の元) の集まりを表すためのデータ構造で，データの **探索 (find)** ，**追加 (insert)** ，**削除 (delete)** 操作を行うことができる．2 分探索木は，節点に格納されたデータ (ここでは整数) が以下の条件 (2 分探索木の**不変条件 (invariant condition)** と呼ぶ) を満たすような特殊な 2 分木である．

> 任意の `branch(`t_1`,`i`,`t_2`)` について，部分木 t_1 に現れる任意の整数 j について $j < i$ かつ，部分木 t_2 に現れる任意の整数 k について $i < k$ が成立する．

[1] 数学で「自然に」(naturally) といった場合，「当然」というニュアンスもある．自然の摂理として当然，という感じでしょうか．

この条件のおかげで，木がバランスしている場合は探索を効率よく行うことができる．一方で，追加・削除にあたってはこの条件を崩さないように木の形を維持する必要があるので少しややこしい．(さらに，バランスを保つように挿入・削除のアルゴリズムを工夫したものが AVL 木や赤黒木といったデータ構造であるが，この講義では扱わない．)

ここでは，2 分探索木は 2 分木の特殊な場合として与えているが，2 分木を定義した時と同様に，その構成規則を与えることで定義することもできる[2]．

1.2.1 2 分探索木の例

既に例でみた t_a, t_b, t_c はみな 2 分探索木になっている．

1.2.2 探索

探索は以下のような簡単な再帰手続きで与えることができる．

```
find(t, i)  // t is a tree and i is an integer; find returns a Boolean.
  if t is leaf then return false
  else if t is branch(t1, j, t2) then
    if i = j then return true
    else if i < j then return find(t1, i)
    else return find(t2, i)
```

2 分探索木の条件より，根に格納された整数と探索する整数の大小関係によって，どちらかの部分木は探索対象から外すことができるというのがミソである．

1.2.3 挿入

挿入に際しては，探索と同様な要領で木にもぐっていき，葉に辿りついたらそれを新しい節点で置き換えることになる．

```
insert(t, i)  // t is a tree and i is an integer
  if t is leaf then replace t with branch(leaf, i, leaf) and return
  else if t is branch(t1, j, t2) then
    if i = j then return
    else if i < j then insert(t1, i)
    else insert(t2, i)
```

上の擬似コードでは `replace` などとさらっと書いているが，後述するように，この部分を実現するのは少し面倒なことが多い．

[2]少し工夫が必要である．いきなり 2 分探索木全体を定義せずに，「格納された任意のデータが区間 $[i, j]$ に入っているような 2 分探索木」を作るための規則を考えるのがよい．

1.2.4 削除

削除については，まず探索と同様の要領で削除対象の節点を見つけるが，その節点が子をいくつ持つかで処理が変わってくる．

1. 子を持たない節点 branch(leaf,j, leaf) の場合は，これを葉に置き換える．
2. 子がひとつの節点 branch(t',j, leaf) もしくは branch(leaf,j,t') の場合は，この節点を t' で置き換える．
3. 子がふたつの場合，右の部分木 (この要素はみな j より大きい) の最小値 k で j を置き換えるとともに，右の部分木から k を (再帰的に) 削除する．

```
delete(t, i)  // t is a tree and i is an integer
  if t is leaf then return // do nothing
  else if t is branch(t1, j, t2) then
    if i = j then
      if both t1 and t2 are leaves then replace t with leaf // case 1
      else if t1 is leaf then replace t with t2  // case 2-1
      else if t2 is leaf then replace t with t1  // case 2-2
      // case 3
      else let k = min(t2);
           replace j in t with k;
           delete(t2, k);
    else if i < j then delete(t1, i)
    else delete(t2, i)
```

1.2.5 練習問題: 探索・挿入・削除の再帰的定義

以下の定義を完成させよ．

$$
\begin{aligned}
find(\texttt{leaf}, i) &= false \\
find(\texttt{branch}(t_1,j,t_2), i) &= \begin{cases} true & (\text{if } i = j) \\ find(\boxed{\ ??\ }, i) & (\text{if } i > j) \\ find(\boxed{\ ??\ }, i) & (\text{if } i < j) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
insert(\texttt{leaf}, i) &= \texttt{branch(leaf,}i\texttt{,leaf)} \\
insert(\texttt{branch}(t_1,j,t_2), i) &= \begin{cases} \boxed{\ ??\ } & (\text{if } i = j) \\ ...insert(t_2, i)... & (\text{if } i > j) \\ ...insert(t_1, i)... & (\text{if } i < j) \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
delete(\texttt{leaf}, i) &= \texttt{leaf} \\
delete(\texttt{branch}(t_1,j,t_2), i) &= \begin{cases} \boxed{\ ??\ } & (\text{if } i = j) \\ ...delete(t_2, i)... & (\text{if } i > j) \\ ...delete(t_1, i)... & (\text{if } i < j) \end{cases}
\end{aligned}
$$