C から入る C++

担当: 金丸隆志

第 12 回 (最終回)

1 はじめに

今回は「**二分木**」を中心とした演習を行う. 二分木でのポインタの使い方は今までとは異なるように思えるかもしれないが, 重要な見方であるので, なるべく理解するようにしたい.

2 「関連」を表すポインタ

前回, 論理回路シミュレータが, 基本素子 (の基底クラス) である Node および Node をつなぐ Edge と Pin からできていることを学んだ.

このように「複数のオブジェクト (素子) をつないで大きなオブジェクト (ネットワーク)」を作る, という手法は大規模なプログラミングをする上でしばしば用いられる.

このとき, オブジェクトと他のオブジェクトの**関連**を定義するにはポインタを用いることが多い [1]. 本演習では, ポインタは「配列の先頭を指すもの」という用途で用いることが多かったが, ここではポインタを本来の定義である「オブジェクトを指し示すもの」として捉えることが必要になる.

ポインタのそのような利用例として代表的なものに, 「**線形リスト**」と「**二分木**」がある.

線形リストは図 1 (a) のように一つのポインタでオブジェクトをつないで行くデータ構造である [2]. 線形リストの各 Node に数値を割りあてると, 線形リストを配列のように利用できる. 線形リストには配列に対して, 以下のようなメリットがある.

- 要素数が固定されておらず, メモリの許す限り値を追加することができる.
- ポインタをつなぎ変えることで, 要素と要素の間に値を挿入することができる.

線形リストは, C++ 標準ライブラリの list コンテナとして実現されている [3]. また, 「配列のソートの C++ 標準ライブラリ版」を取り扱った際, 配列の代わりに「int 型の vector コンテナ」を用い, コンテナの末尾に乱数を次々に追加していったのを覚えているだろうか. vector コンテナの実装は線形リストとは異なるが, 上記と同様のメリットを持っていることに注意しておく.

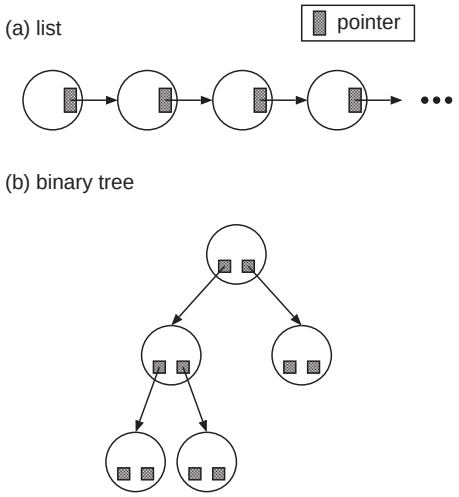

図 1: 線形リストと二分木

一方, 二分木 (binary tree) は図 1 (b) のように, 二つのポインタで Tree のように Node をつないでゆくデータ構造である [4]. 二分木の利点と実装法などを以下の章で詳しく取り扱う.

なお, 二分木は C++ 標準ライブラリの map コンテナなどとして実装されており, 論理回路シミュレータ内でも利用されていることに注意しておく. それゆえ, 今回の演習の目的は「ポインタを用いてオブジェクトをつなぐという手法の理解」および「標準ライブラリ内のコンテナがどのように実現されているかの理解」ということになるだろう.

3　二分木とは

3.1　二分木の利用例

ある英文から, 単語の出現回数をカウントすることを考える. 例えば, “How does the living organism avoid decay to the equilibrium?” (「どのようにして生体は (熱) 平衡へおちいるのを防ぐのだろうか?」) [8] という文を考えよう.

基本的には, 単語 “How” から始め, 新たな単語が出て来るたびにデータを追加してゆけばよい. これ

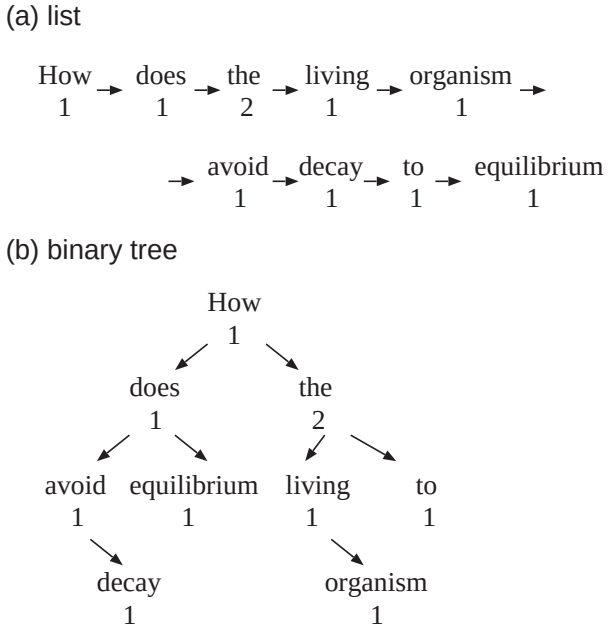

図 2: 線形リストと二分木による単語出現回数のカウント.

を線形リストと二分木を用いて実現することを考えてみよう. そのためには, リストと二分木の各 Node のデータメンバに, 単語をあらわす key (char 型) と出現回数をあらわす data (int 型) があれば良いことがわかるだろう.

線形リストを用いて単語の出現回数をカウントすると図 2 (a) のようになる. 新たな単語を加える場合, リストの先頭 (“How”) からスキャンし, 単語が既に存在すれば, data を一増やし, 存在しなければリストの末尾に新たな Node を加えれば良い.

一方, 二分木を用いて単語の出現回数をカウントすると図 2 (b) のようになる. 二分木の基本的なアイディアは, 「key の大小関係によって, 登録先 (左か, 右か) を決める」ということである. 単語の例の場合, 「辞書順で先に来る単語を左下に追加する」ということである. 例えば図 2 (b) の二分木に “is” という単語を追加することを考える. 「How より右」, 「the より左」と木を辿って行き, 最終的に「living の左下」に “is” という key を持つ Node を付け加えることになる.

[**課題**]
“How” からはじめ, 図 2 (b) の木の図を描いてみよ.

3.2　二分木の利点

前節の例で, すでに N 個の単語が線形リストまたは二分木に登録されているとする.

ここで新たな単語を登録することを考えよう. 単語を登録するには, すでにその単語がリストまたは木に含まれているかどうかを調べなければならなかった. その探索の回数はリストの場合, 平均的に N 回のオーダーである. 一方, 二分木の場合, 全ての単語を探索する必要はなく, 平均的に $\log_2 N$ のオーダーで済む.

図 2 のように単語数が少ないときは N と $\log_2 N$ の差は大きくないが, 「新聞に含まれる単語の出現回数」や「本一冊の単語の出現回数」などを考えると, N と $\log_2 N$ の差は非常に大きくなる (選択ソートとクイックソートの処理時間の差を思いだそう).

このように, 登録や探索にかかる時間が非常に短くて済むというのが二分木の利点である. (もちろん, 線形リストには線形リストの利点があり, 目的に応じて使い分ければ良い)

3.3　イテレータ (反復子)

前節での例のように, 二分木は登録されたデータの探索に向いていることがわかった. しかし, 「登録された全てのデータを取り出す」ことについてはどうだろうか.

線形リストであれば, ポインタをたどって全てのデータを取り出すことは簡単にできる. しかし二分木の場合, ポインタが 2 つあるので, 木を単純にたどってゆくのは難しそうである.

このように, 登録された任意のデータにアクセスするために, C++ 標準ライブラリには**イテレータ** (iterator, または**反復子**) というクラスが用意されている. イテレータは木の中の Node を指すように

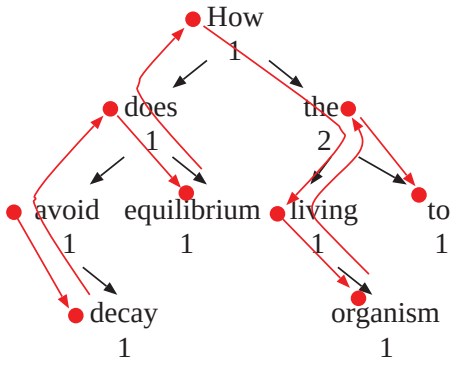

図 3: 二分木におけるイテレータの動作.

なっており [9], 順に木をたどって行く. イテレータの動作例は図 3 のようになる.

本演習でもこのイテレータを模倣した振舞いをする機構を組み入れるが, その実装は難しいので, こちらで提示する.

3.4 若干の注意

“A big cat is sleeping.” という文に対して二分木を作ると, 図 4 のようになる. 図からわかるように,

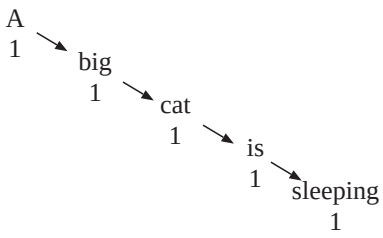

図 4: バランスの悪い二分木の例.

この場合は探索にかかる時間は線形リストと同じになり, 二分木のメリットは失われる.

ただし, 探索にかかる時間を短縮する必要があまりない場合は, 図 4 のように二分木を線形リストとして用いることもできる. 実は論理回路シミュレータで用いられている map (C++ 標準ライブラリの二分木) はそのような使い方になっている.

3.5 二分木の他の例

二分木の例として,「単語の出現回数のカウント」を取り上げた. 他にも,「チェスの次の手の格納 [6]」,「遺伝的プログラミング [10]」など, 多くの例で二分木に類似したデータ構造が用いられている.

4 二分木の実装

4.1 満たすべき性質

前章の解説に基いて, 以下のような二分木クラスを作成してみよう.

- Node を表す BNode クラス, および二分木を表す BTree クラスを持つ.
- BNode クラスは左右の Node を指すポインタ leftNode および rightNode をもつ. さらに, 上位の Node をたどれるように, ポインタ parentNode も持つ. さらに, BNode クラスはデータメンバ「int key」および「int data」を持つ. 前章の解説では key は文字列 (char* または myString) であったが, 簡単のために int にした [11].
- (key, data) のデータの組を持つ Node を追加するため, BNode クラスは insert(key, data) 関数を持つ. それを二分木クラスから呼び出すため, BTree クラスにも insert(key,data) 関数を持たせる.
- (key, data) のデータを持つ Node が存在するとき, key から data を取り出す (探索する) ために, BNode および BTree クラスに find(key) 関数を持たせる. find(key) は data を返す.

プログラムの枠組みをダウンロードできるようになっているので, 以下はファイルをダウンロードし, ファイルを参照しながら読み進めて欲しい.

4.2 データの追加 (insert 関数)

今, (key, data) なる値を持つ Node があるとし, そこに (k, d) なる値を持つ Node を追加する関数 insert(k, d) を考えよう (図 5). このとき, ポイントになるのは key と k の値の大小関係である.

まず「key=k」の時, すなわち既に同じ key が存在する場合を考える. 前章の例では data に 1 を加える (単語の出現回数を増やす) ところであるが, 後の演習のことを考え, ここでは「d で data を上書きする」という仕様にする.

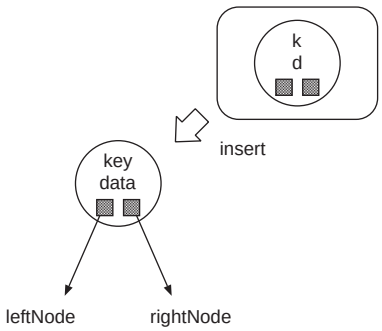

図 5: BNode の追加.

つぎに「k<key」の時を考える. このとき, (k,d) の値を持つ Node は leftNode 側に接続されるはずである. しかし, 実際に leftNode に直接接続されるかどうかは leftNode が指す Node が存在するかどうかに依存する. 存在しない場合は leftNode に (k, d) の Node を接続すれば良い. 一方, leftNode が指す Node が存在する場合は, lefNode よりも下位のどこかに (k,d) の Node を接続することになる.

このとき, leftNode に対してさらに insert 関数を呼び出せばよい. このように入れ子になった関数呼び出しを**再帰的** (recursive) な関数呼び出しという. これにより, 下位の Node をたどって Node を追加することが可能になる.

4.3 データの探索 (find 関数)

key を手がかりに, (key, data) なる値を持つ Node を探索することを考えよう. 考え方は insert 関数とほとんど同じで, ここでも再帰的な関数呼び出しがポイントとなる.

「k=key」のときは, その Node が求めるものであるから, そのまま data を返せば良い.

「k<key」の時は, leftNode 側の Node のどこかに求める Node がある可能性があるので, leftNode の指す Node に対して再帰的に探索を試みればよい.

もちろん, k なる key をもつ Node が存在しない場合もありうる. そのときは 0 を返すように find 関数を定義しよう.

4.4 BTree クラス

BNode クラスをまとめるクラスとして二分木 (BTree) クラス があるわけだが, BTree クラスは最上位の Node (root) とイテレータ (iterator) だけをデータメンバに持てばよい (図 6).

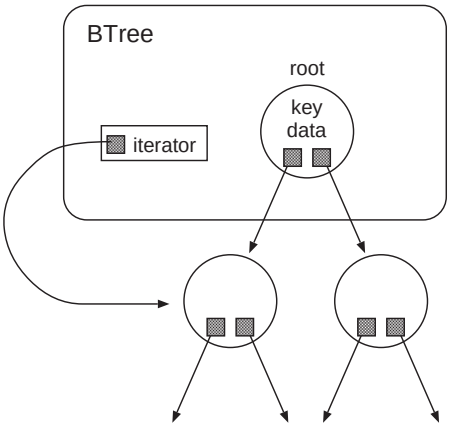

図 6: BTree クラスの模式図.

BTree に対して BNode を追加したり, 探索したりするには, root の Node に対して, insert 関数と find 関数を呼び出せばよい. これにより再帰的に下位の Node まで探索される.

なお, BNode クラスと BTree クラスでそれぞれ insert 関数という同じ名前の関数を持つが, これらはそれぞれ BNode::insert(k,d), BTree::insert(k,d) であり, 実体は別であることに注意しておく (find 関数も同様).

4.5 BTree クラスの使用例 (btree_test.cpp)

btree_test.cpp では, 図 7 のような二分木を作成している. (ただし, 選ばれている key, data にはそれほど意味はない. 一応, data は挿入される順番を表すようにした.)

ただし, BTree.cpp において, **「右側に Node を追加 (探索) する」機能が実装されていない**ので, 図 7 の二分木は一部しか作成されない.

[**課題**]

(1) 「右側に Node を追加 (探索) する」機能を実装し, 二分木クラスを完成させよ.
BTree.cpp を「注」という文字列で検索すれば, 記述すべき場所がわかる (二箇所ある).

(2) 論理回路シミュレータを思いだそう. この中で, Edge の集合を C++ 標準ライブラリの map

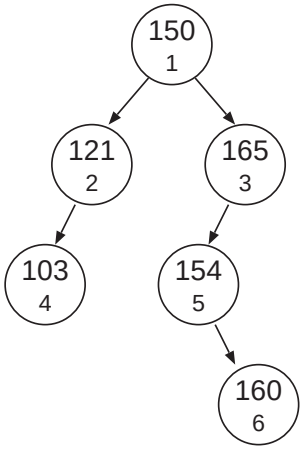

図 7: btree_test.cpp で作成する二分木.

クラスで実現している. この EdgeMap クラスを, ここで作成した BTree クラスで置き換えよ (詳細は別途指示する).

参考文献

[1] Tucker!, “憂鬱なプログラマのためのオブジェクト指向開発講座,” 翔泳社 (1998).

[2] 線形リストの実現法ついては, 文献 [6] の 20.3 章, 文献 [7] の 3-5 章などが参考になるだろう.

[3] Nicolai M. Josuttis, “C++ 標準ライブラリチュートリアル & リファレンス,” アスキー (2001).

[4] 二分木の実現法については, 文献 [5] の 6.5 章 (ただし, C 言語の教本なので構造体による記述), 文献 [6] の 20.6 章, 文献 [7] の 5-4 章などが参考になるだろう.

[5] B.W. カーニハン/D.M リッチー, “プログラミング言語 C (第 2 版),” 共立出版株式会社 (1989).

[6] Steve Oualline, “C++ 実践プログラミング,” オライリー・ジャパン (1996).

[7] 柴田望洋, “C プログラマのための C++ 入門,” ソフトバンク (1992).

[8] Erwin Schrödinger, “What is life?,” (1944).

[9] 正確には, Node と Node の間を指すようになっている.

[10] 遺伝的プログラミング (Genetic Programming: GP) は, 遺伝的アルゴリズム (Genetic Algorithm: GA) の一分野で, プログラムを遺伝的アルゴリズムを用いて自動生成しよう, という研究分野である. その際, 進化の対象となるプログラムは二分木で表現される.

たとえば, 「遺伝的プログラミング Notebook」
http://www.geneticprogramming.com/
japanese_frames/
の「遺伝的プログラミングチュートリアル」でその図をみることができる.

[11] char∗ 型の key を持つ二分木を作るには文献 [5], [6], [7] などを参照.