UIN SYARIF HIDAYATULLAH JAKARTA

TREES

ALGORITMA DAN STRUKTUR DATA

Oleh : Syahrul Akbar Ramdhani (11230940000027)

Dosen Pengampu: M. Irvan Septiar Musti, M.Si

8.1 GENERAL TREES

Para ahli produktivitas sering menyebut bahwa terobosan muncul dari pemikiran "nonlinier". Dalam konteks ini, struktur data pohon (tree) adalah salah satu bentuk pemikiran nonlinier yang penting dalam ilmu komputer. Struktur pohon memungkinkan kita untuk menjalankan berbagai algoritma dengan lebih cepat dibandingkan struktur linier seperti array atau linked list.

Pohon juga mencerminkan cara alami dalam mengorganisasi data, sehingga banyak digunakan dalam sistem file, antarmuka grafis, basis data, situs web, dan sistem komputer lainnya.

Saat disebut "nonlinier", yang dimaksud adalah bahwa pohon tidak hanya mengandalkan urutan "sebelum" dan "sesudah" seperti pada array, tetapi menggunakan hubungan hierarkis, di mana beberapa elemen berada "di atas" atau "di bawah" elemen lainnya. Istilah-istilah dalam pohon seperti parent, child, ancestor, dan descendant diambil dari struktur pohon keluarga.

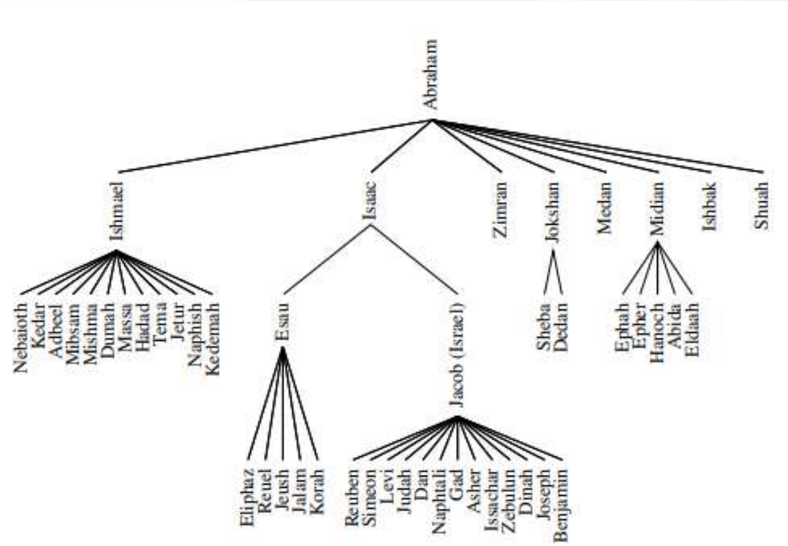


Figure 8.1: A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

8.1.1 Tree Definitions and Properties

Tree (pohon) adalah tipe data abstrak yang menyimpan elemen secara hierarkis. Kecuali elemen paling atas (yang disebut root), setiap elemen memiliki induk (parent) dan nol atau lebih anak (children).

Pohon biasanya divisualisasikan dengan elemen-elemen dalam bentuk oval atau persegi, dan garis lurus digunakan untuk menghubungkan antara elemen induk dan anak. Meskipun disebut root (akar), elemen paling atas ini digambar di bagian atas, sedangkan elemen lainnya tersusun di bawahnya, berlawanan dengan pohon dalam dunia nyata (botani).

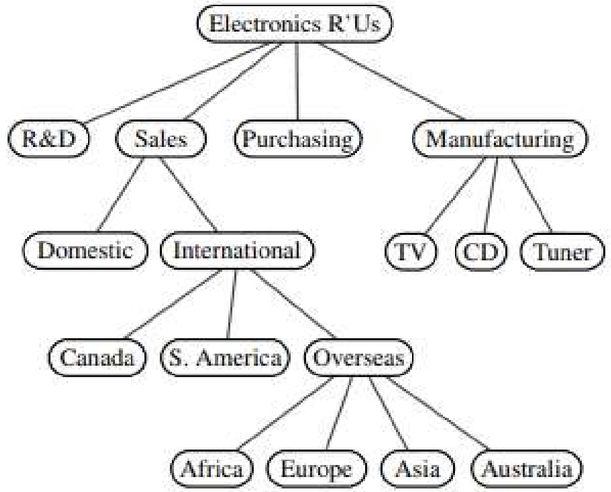


Figure 8.2: A tree with 17 nodes representing the organization of a fictitious corporation. The root stores *Electronics R'Us*. The children of the root store *R&D*, *Sales*, *Purchasing*, and *Manufacturing*. The internal nodes store *Sales*, *International*, *Overseas*, *Electronics R'Us*, and *Manufacturing*.

Formal Tree Definition

Secara formal, pohon (tree) T adalah sekumpulan simpul (node) yang menyimpan elemen dan memiliki hubungan induk-anak (parent-child) yang memenuhi sifat-sifat berikut:

- Jika T tidak kosong, maka T memiliki simpul khusus yang disebut akar (root), yang tidak memiliki induk.
- Setiap simpul v (selain akar) memiliki satu induk unik yaitu simpul w. Semua simpul yang memiliki induk w disebut anak-anak dari w.

Perlu dicatat bahwa berdasarkan definisi ini, pohon bisa kosong, artinya tidak memiliki simpul sama sekali. Konvensi ini memungkinkan kita untuk mendefinisikan pohon secara rekursif, yaitu, Sebuah pohon T adalah kosong, atau terdiri dari satu simpul akar (r), dan sekelompok (mungkin kosong) subpohon yang akarnya adalah anak-anak dari r.

Other Node Relationships

Dua simpul yang merupakan anak dari induk yang sama disebut saudara (siblings). Sebuah simpul v disebut eksternal jika tidak memiliki anak. Sebaliknya, simpul v disebut internal jika memiliki satu atau lebih anak. Simpul eksternal juga dikenal sebagai daun (leaves).

Contoh 8.1:

Pada bagian 4.1.4 sebelumnya, telah dibahas hubungan hierarkis antara file dan direktori dalam sistem file komputer, meskipun saat itu belum disebut secara eksplisit bahwa struktur tersebut adalah pohon.

Pada Gambar 8.3, contoh itu ditampilkan kembali. Kita bisa melihat bahwa, simpul internal dalam pohon mewakili direktori, dan daun (leaves) atau simpul eksternal mewakili file biasa.

Dalam sistem operasi UNIX dan Linux, akar pohon disebut sebagai “root directory”, yang disimbolkan dengan “/”.

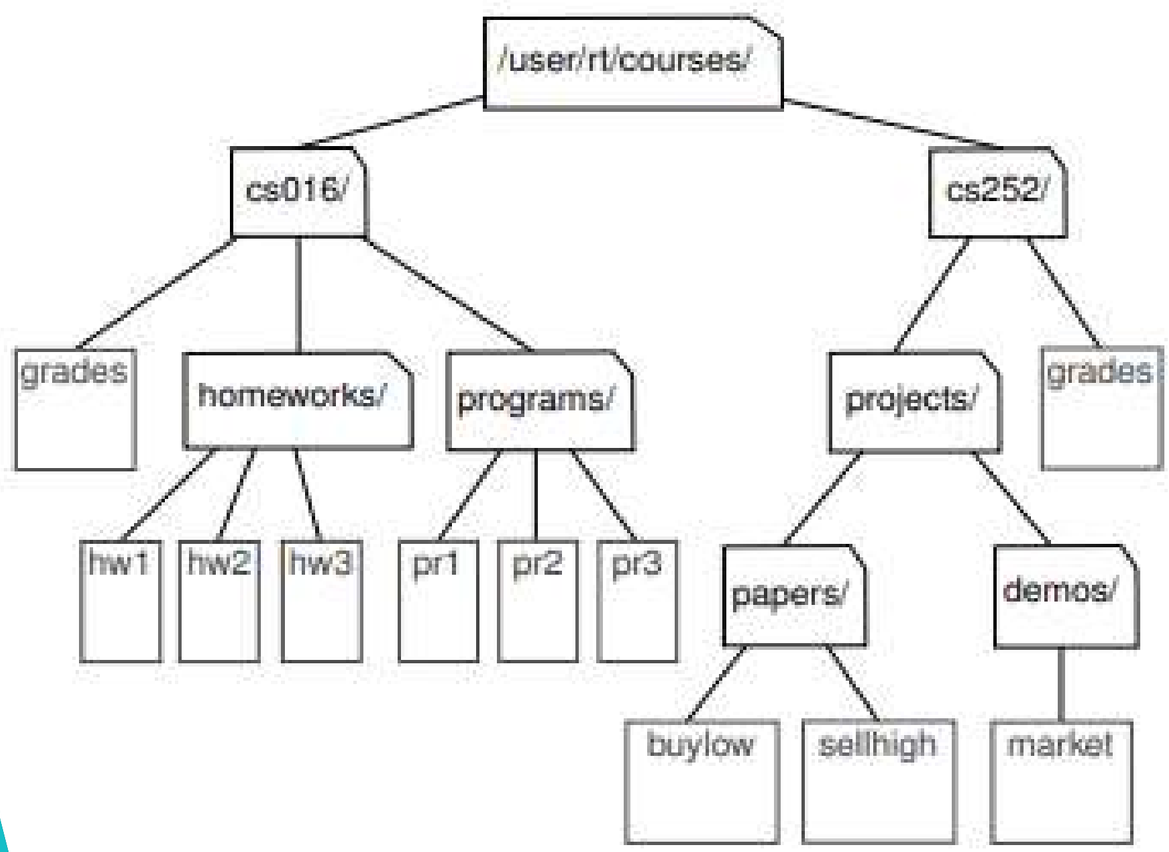


Figure 8.3: Tree representing a portion of a file system.

Hubungan Kekerabatan dalam Trees

- Sebuah node u disebut nenek moyang (ancestor) dari node v jika:
  - u = v, atau
  - u adalah ancestor dari induk v.
- Sebaliknya, node v disebut keturunan (descendant) dari node u jika u adalah ancestor dari v.

Contoh:

- Dalam Gambar 8.3, direktori cs252/ adalah ancestor dari direktori papers/.
- File pr3 adalah descendant dari direktori cs016/.

Subtree (Subpohon):

- Subtree dari pohon T yang berakar di simpul v adalah pohon yang terdiri dari:
  - Semua keturunan v dalam T, termasuk v itu sendiri.

Contoh Subtree:

- Subtree yang berakar di direktori cs016/ berisi:
  - Direktori: cs016/, homeworks/, programs/
  - File: grades, hw1, hw2, hw3, pr1, pr2, pr3

Edges and Paths in Trees

Edge (sisi) dari pohon T adalah pasangan simpul (u, v) di mana u adalah induk dari v, atau sebaliknya (v adalah anak dari u). Path (lintasan) dalam pohon T adalah urutan simpul di mana setiap dua simpul yang berurutan dalam urutan tersebut terhubung oleh sebuah edge. Contoh, dalam Gambar 8.3, terdapat path (cs252/, projects/, demos/, market), artinya, mulai dari direktori cs252/ menuju projects/, lalu ke demos/, dan terakhir ke market, semua simpul tersebut terhubung secara hierarkis melalui edge-edge.

Hubungan pewarisan (inheritance) antar kelas dalam program Python membentuk sebuah pohon jika hanya menggunakan pewarisan tunggal (single inheritance).
Sebagai contoh, pada Bagian 2.4, telah dijelaskan hierarki tipe-tipe exception dalam Python (ditampilkan pada Gambar 8.4, yang sebelumnya adalah Gambar 2.5). Kelas BaseException adalah akar (root) dari hierarki tersebut. Semua kelas exception buatan pengguna (user-defined exception) secara konvensional sebaiknya diturunkan dari kelas Exception, yang merupakan turunan dari BaseException.
Contohnya, kelas Empty yang diperkenalkan dalam Code Fragment 6.1 di Bab 6 adalah turunan dari Exception.

Edges and Paths in Trees

Di Python, semua kelas tersusun dalam satu hierarki tunggal, karena terdapat kelas bawaan yang disebut object, yang menjadi kelas dasar utama (ultimate base class). Kelas object adalah induk langsung atau tidak langsung dari semua tipe lainnya di Python, bahkan jika tidak dinyatakan secara eksplisit saat membuat kelas baru.

Oleh karena itu, hierarki yang ditampilkan pada Gambar 8.4 hanyalah sebagian kecil dari keseluruhan hierarki kelas Python.

Sebagai gambaran awal untuk materi berikutnya dalam bab ini, Gambar 8.5 menunjukkan hierarki kelas buatan sendiri yang akan digunakan untuk merepresentasikan berbagai bentuk dari struktur pohon.

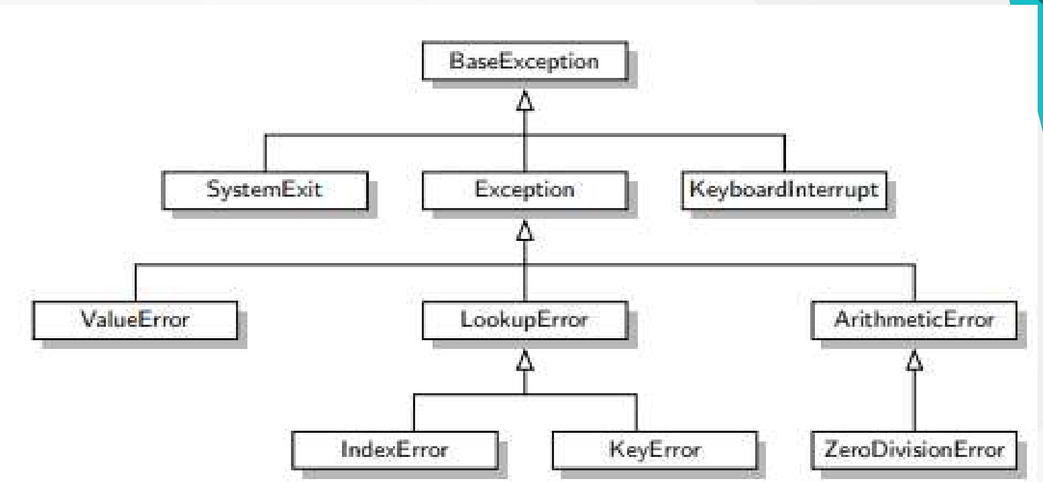
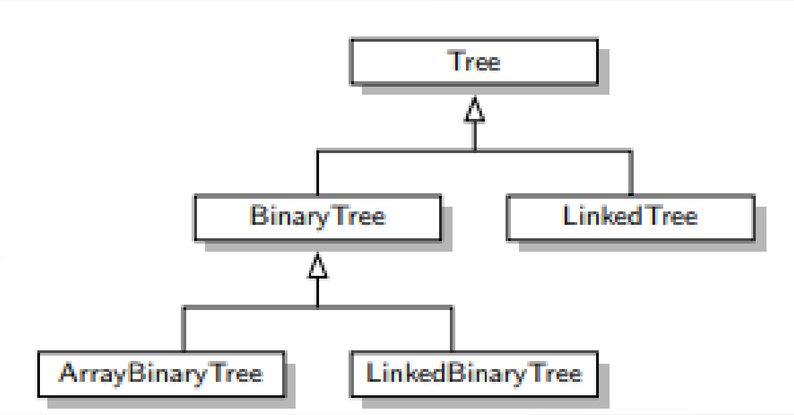


Figure 8.4: A portion of Python's hierarchy of exception types.

Gambar 8.5 memperlihatkan hierarki pewarisan (inheritance hierarchy) yang kita buat sendiri untuk memodelkan berbagai abstraksi dan implementasi struktur data pohon.

Dalam sisa bab ini, akan dijelaskan, implementasi dari kelas, Tree (pohon umum/abstrak), BinaryTree (pohon biner), LinkedBinaryTree (pohon biner dengan representasi berantai, serta gambaran umum (sketsa tingkat tinggi) untuk desain kelas, LinkedTree (pohon umum berbasis linked structure), ArrayBinaryTree (pohon biner berbasis array)



Ordered Trees

Sebuah pohon dikatakan terurut jika terdapat urutan linier yang bermakna di antara anak-anak dari setiap simpul. Artinya, kita secara sengaja mengidentifikasi anak-anak tersebut sebagai anak pertama, kedua, ketiga, dan seterusnya.
Visualisasi dari pohon terurut biasanya dilakukan dengan menyusun simpul-simpul saudara (siblings) dari kiri ke kanan, sesuai dengan urutan tersebut.

ontoh 8.3: Komponen dari sebuah dokumen terstruktur, seperti buku, tersusun secara hierarkis dalam bentuk pohon, di mana, simpul internal mewakili bagian (part), bab (chapter), dan subbagian (section). Daun (leaves) mewakili paragraf, tabel, gambar, dan elemen konten lainnya. (liat gambar 8.6). Akar dari pohon ini merepresentasikan buku itu sendiri. Pohon ini bahkan bisa diperluas lebih dalam, misalnya paragraf terdiri dari kalimat, kalimat terdiri dari kata, kata terdiri dari karakter. struktur ini merupakan contoh dari pohon terurut (ordered tree) karena terdapat urutan yang jelas di antara anak-anak setiap simpul, misalnya urutan paragraf dalam bab, atau urutan kata dalam kalimat.

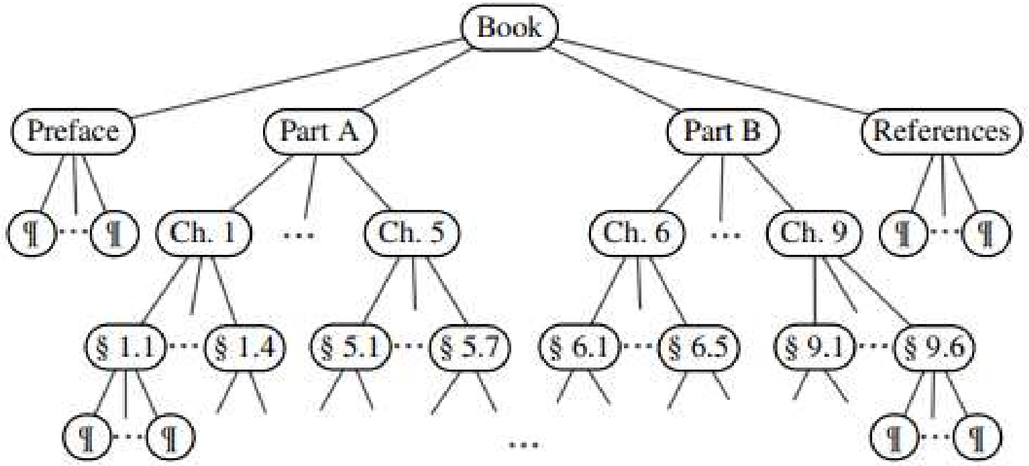


Figure 8.6: An ordered tree associated with a book.

Apakah Urutan Anak Penting dalam Pohon?

Mari kita lihat kembali contoh-contoh pohon yang telah dibahas sebelumnya dan pertimbangkan apakah urutan anak pada tiap simpul bermakna:

- Pohon keluarga (seperti pada Gambar 8.1):
- Biasanya dianggap sebagai pohon terurut, karena urutan saudara mencerminkan urutan kelahiran.
- Bagan organisasi perusahaan (Gambar 8.2):
- Umumnya dianggap sebagai pohon tak terurut, karena urutan antar posisi/jabatan tidak penting secara struktural.
- Hierarki pewarisan kelas dalam pemrograman (Gambar 8.4):
- Juga merupakan pohon tak terurut, karena urutan subclass tidak memiliki arti khusus.
- Struktur file komputer (Gambar 8.3):
- Meskipun sistem operasi bisa menampilkan isi direktori dalam urutan tertentu (misalnya alfabetis atau berdasarkan waktu), urutan tersebut tidak melekat dalam struktur pohon pada level sistem file. Jadi, ini biasanya dianggap sebagai pohon tak terurut juga.

8.1.2 The Tree Abstract Data Type

Seperti yang kita lakukan pada positional list di Bagian 7.4, kita mendefinisikan tree ADT menggunakan konsep posisi sebagai abstraksi untuk simpul dari sebuah pohon. Sebuah elemen disimpan di setiap posisi, dan posisi-posisi ini memenuhi hubungan induk-anak yang mendefinisikan struktur pohon.

Sebuah objek posisi pada pohon mendukung metode berikut:

- p.element(): Mengembalikan elemen yang disimpan pada posisi p.

ADT pohon kemudian mendukung metode-metode akses (accessor) berikut, yang memungkinkan pengguna menavigasi berbagai posisi dalam pohon:

- T.root(): Mengembalikan posisi akar dari pohon T, atau None jika T kosong.
- T.is_root(p): Mengembalikan True jika posisi p adalah akar dari pohon T.
- T.parent(p): Mengembalikan posisi induk dari posisi p, atau None jika p adalah akar dari T.
- T.num_children(p): Mengembalikan jumlah anak dari posisi p.
- T.children(p): Menghasilkan iterasi dari anak-anak posisi p.
- T.is_leaf(p): Mengembalikan True jika posisi p tidak memiliki anak.
- len(T): Mengembalikan jumlah posisi (dan juga elemen) yang terdapat dalam pohon T.
- T.is_empty(): Mengembalikan True jika pohon T tidak memiliki posisi apa pun.
- T.positions(): Menghasilkan iterasi dari semua posisi dalam pohon T.
- iter(T): Menghasilkan iterasi dari semua elemen yang disimpan dalam pohon T.

8.1.2 The Tree Abstract Data Type

Setiap metode di atas yang menerima posisi sebagai argumen harus menghasilkan ValueError jika posisi tersebut tidak valid untuk T.

Jika sebuah pohon T adalah terurut, maka T.children(p) akan mengembalikan anak-anak dari p dalam urutan yang alami. Jika p adalah sebuah leaf, maka T.children(p) akan menghasilkan iterasi kosong. Demikian pula, jika pohon T kosong, maka baik T.positions() maupun iter(T) akan menghasilkan iterasi kosong.

Kita belum mendefinisikan metode apa pun untuk membuat atau memodifikasi pohon pada titik ini. Kita akan menjelaskan berbagai metode pembaruan pohon bersama dengan implementasi khusus dari antarmuka pohon dan aplikasi-aplikasi pohon tertentu.

A Tree Abstract Base Class in Python

Dalam Python, abstraksi ADT (Abstract Data Type) sering diterapkan lewat duck typing, seperti yang dilakukan pada berbagai implementasi queue (ArrayQueue, LinkedQueue, CircularQueue) yang memiliki antarmuka serupa tanpa definisi formal.

Untuk pendekatan yang lebih formal, digunakan abstract base class melalui inheritance, seperti kelas Tree dalam Code Fragment 8.1, yang mewakili ADT pohon.

Kelas Tree ini mendefinisikan:

- Kelas bersarang abstrak Position, dan
- Deklarasi metode-metode akses pohon.

Namun, Tree tidak menyimpan struktur data secara langsung, dan lima metode utama (root, parent, num_children, children, __len__) masih abstrak (menghasilkan NotImplementedError).

Implementasi konkrit akan dibuat oleh subclass, dengan menyusun representasi internal masing-masing.

Meskipun Tree adalah kelas abstrak, ia juga menyediakan beberapa metode konkrit seperti is_root, is_leaf, dan is_empty (lihat Code Fragment 8.2), yang mengandalkan metode abstrak dan akan diwarisi oleh semua subclass, mendorong reusabilitas kode.

Nantinya, metode seperti positions() dan __iter__() juga akan diimplementasikan berbasis algoritma penelusuran pohon yang dibahas di Bagian 8.4.

Karena Tree bersifat abstrak, tidak perlu dibuat instance langsung dari kelas ini. Ia hanya berfungsi sebagai dasar pewarisan untuk kelas-kelas konkret.

IMPLEMENTATION IN PYTHHON

```
class Tree:
  """Abstract base class representing a tree structure."""

  #-------------------- nested Position class --------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)              # opposite of __eq__

  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
    """Return the number of children that Position p has."""
    raise NotImplementedError('must be implemented by subclass')

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    raise NotImplementedError('must be implemented by subclass')

  def __len__(self):
    """Return the total number of elements in the tree."""
    raise NotImplementedError('must be implemented by subclass')
```

Code Fragment 8.1: A portion of our Tree abstract base class (continued in Code Fragment 8.2).

```
  # ---------- concrete methods implemented in this class ----------
  def is_root(self, p):
    """Return True if Position p represents the root of the tree."""
    return self.root( ) == p

  def is_leaf(self, p):
    """Return True if Position p does not have any children."""
    return self.num_children(p) == 0

  def is_empty(self):
    """Return True if the tree is empty."""
    return len(self) == 0
```

Code Fragment 8.2: Some concrete methods of our Tree abstract base class.

8.1.3 Computing Depth and Height

Misalkan p adalah posisi dari suatu simpul dalam pohon T. Maka, kedalaman (depth) dari p adalah jumlah nenek moyang (ancestor) dari p, tidak termasuk p itu sendiri Sebagai contoh, dalam pohon pada Gambar 8.2, simpul yang menyimpan "International" memiliki kedalaman 2. Definisi ini juga menyiratkan bahwa akar (root) dari T memiliki kedalaman 0. Kedalaman dari p juga dapat didefinisikan secara rekursif sebagai berikut:

- Jika p adalah akar, maka kedalamannya adalah 0.
- Jika tidak, maka kedalamannya adalah satu ditambah kedalaman dari induk p.

Berdasarkan definisi ini, kita dapat membuat algoritma rekursif sederhana, bernama depth (lihat Code Fragment 8.3), untuk menghitung kedalaman dari posisi p dalam pohon T.

Metode ini memanggil dirinya sendiri secara rekursif pada induk dari p, lalu menambahkan 1 terhadap nilai yang dikembalikan.

Waktu eksekusi dari T.depth(p) untuk posisi p adalah O(dp + 1), di mana dp adalah kedalaman posisi p dalam pohon T. Hal ini karena algoritma melakukan langkah rekursif berdurasi konstan untuk setiap nenek moyang (ancestor) dari p. Jadi, algoritma T.depth(p) memiliki kompleksitas waktu terburuk O(n), dengan n adalah jumlah total posisi dalam T, karena dalam kasus ekstrem, semua simpul membentuk satu cabang dan sebuah posisi bisa memiliki kedalaman hingga n - 1. Namun, meskipun kompleksitas ini bergantung pada ukuran input n, akan lebih informatif jika waktu eksekusi dinyatakan sebagai fungsi dari parameter dp, karena dp sering kali jauh lebih kecil daripada n dalam banyak kasus nyata.

```
52  def depth(self, p):
53    """Return the number of levels separating Position p from the root."""
54    if self.is_root(p):
55      return 0
56    else:
57      return 1 + self.depth(self.parent(p))
```

Code Fragment 8.3: Method depth of the Tree class.

8.1.3 Computing Depth and Height

Tinggi suatu posisi dalam pohon didefinisikan secara rekursif: jika posisi tersebut adalah daun (leaf), maka tingginya 0; jika bukan, tingginya adalah satu ditambah tinggi maksimum dari anak-anaknya. Tinggi pohon yang tidak kosong adalah tinggi dari akarnya. Menurut Proposisi 8.4, tinggi pohon juga dapat ditentukan sebagai kedalaman maksimum dari semua posisi daun. Berdasarkan proposisi ini, algoritma height1 disusun sebagai metode nonpublik dalam kelas Tree, dengan memanfaatkan algoritma depth untuk menghitung tinggi pohon secara tidak langsung dari kedalaman semua daunnya.

```
58  def _height1(self):                    # works, but O(n^2) worst-case time
59    """Return the height of the tree."""
60    return max(self.depth(p) for p in self.positions( ) if self.is_leaf(p))
```

Code Fragment 8.4: Method _height1 of the Tree class. Note that this method calls the depth method.

Sayangnya, algoritma height1 tidak efisien. Meskipun metode positions() nantinya dapat diimplementasikan dengan waktu O(n), algoritma height1 memanggil depth(p) untuk setiap daun pohon, sehingga waktu eksekusinya menjadi $O(n + \sum_{p \in L}(d_p + 1))$, dengan L adalah himpunan daun dan dp adalah kedalaman daun p. Dalam kasus terburuk, jumlah tersebut bisa sebanding dengan $O(n^2)$, sehingga height1 memiliki kompleksitas waktu terburuk $O(n^2)$. Untuk mengatasinya, kita bisa menggunakan definisi rekursif asli dari tinggi simpul, dengan menghitung tinggi dari subtree yang berakar di suatu posisi. Pendekatan ini diimplementasikan dalam algoritma height2 (lihat Code Fragment 8.5), sebagai metode nonpublik yang mampu menghitung tinggi pohon dalam waktu terburuk O(n).

```
61  def _height2(self, p):                 # time is linear in size of subtree
62    """Return the height of the subtree rooted at Position p."""
63    if self.is_leaf(p):
64      return 0
65    else:
66      return 1 + max(self._height2(c) for c in self.children(p))
```

Code Fragment 8.5: Method _height2 for computing the height of a subtree rooted at a position *p* of a Tree.

8.1.3 Computing Depth and Height

Penting untuk memahami mengapa algoritma height2 lebih efisien dibandingkan height1. Algoritma ini bersifat rekursif dan berjalan dari atas ke bawah (top-down), dimulai dari akar dan kemudian memanggil dirinya sendiri untuk setiap anak secara berurutan hingga seluruh posisi dalam pohon dikunjungi satu kali. Kompleksitas waktu dihitung dari total kerja non-rekursif di setiap posisi, yang terdiri dari operasi konstan dan pencarian nilai maksimum dari anak-anaknya. Jika iterasi children(p) membutuhkan waktu O(cp + 1), maka waktu totalnya adalah O(∑p(cp + 1)) = O(n + ∑p cp). Berdasarkan Proposisi 8.5, jumlah total anak dari semua simpul (∑p cp) dalam pohon dengan n posisi adalah n − 1, sehingga algoritma height2 memiliki kompleksitas waktu O(n). Untuk kenyamanan pengguna, kita dapat membungkus height2 (yang bersifat nonpublik) dalam metode publik height() sehingga pengguna cukup memanggil T.height() tanpa harus menyebutkan akar pohon secara eksplisit, seperti yang ditunjukkan pada Code Fragment 8.6.

```
67    def height(self, p=None):
68      """Return the height of the subtree rooted at Position p.
69
70      If p is None, return the height of the entire tree.
71      """
72      if p is None:
73        p = self.root( )
74      return self._height2(p)          # start _height2 recursion
```

Code Fragment 8.6: Public method Tree.height that computes the height of the entire tree by default, or a subtree rooted at given position, if specified.

8.2 BINARY TREES

Pohon biner adalah pohon terurut (ordered tree) yang memiliki tiga ciri utama: (1) setiap simpul memiliki maksimal dua anak, (2) masing-masing anak diberi label sebagai anak kiri atau anak kanan, dan (3) anak kiri selalu didahulukan dari anak kanan dalam urutan anak. Subpohon yang berakar pada anak kiri atau kanan disebut subpohon kiri dan subpohon kanan. Sebuah pohon biner disebut pohon biner proper (atau full binary tree) jika setiap simpul memiliki tepat dua anak atau tidak memiliki anak sama sekali; jika tidak, maka disebut improper.

Salah satu contoh penting dari pohon biner adalah pohon keputusan (decision tree), yang digunakan untuk merepresentasikan berbagai kemungkinan hasil dari rangkaian pertanyaan ya/tidak. Dalam pohon keputusan, setiap simpul internal adalah sebuah pertanyaan, dan cabang kiri atau kanan diikuti tergantung jawabannya (“Ya” atau “Tidak”), hingga mencapai daun yang berisi keputusan. Pohon keputusan ini merupakan contoh pohon biner yang proper. Gambar 8.7 menggambarkan contoh pohon keputusan untuk memberikan saran kepada calon investor.

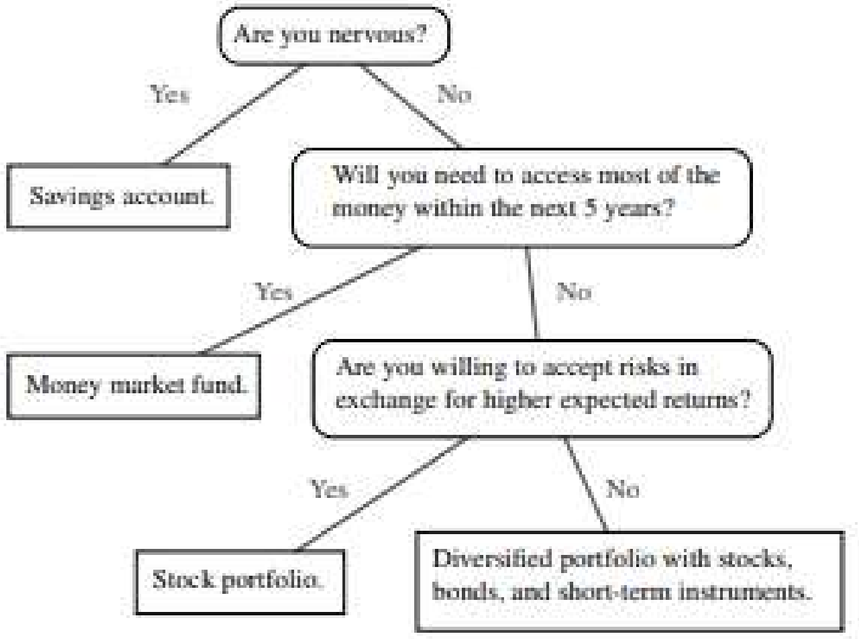


Figure 8.7: A decision tree providing investment advice.

8.2 BINARY TREES

Contoh 8.7: Suatu ekspresi aritmetika dapat direpresentasikan dalam bentuk pohon biner, di mana daun-daunnya berisi variabel atau konstanta, dan simpul-simpul internalnya berisi salah satu operator aritmetika: +, −, ×, atau / (lihat Gambar 8.8). Setiap simpul dalam pohon ini memiliki nilai yang ditentukan sebagai berikut:

- Jika simpul tersebut adalah daun, maka nilainya adalah nilai dari variabel atau konstanta yang dikandungnya.
- Jika simpul tersebut adalah simpul internal, maka nilainya dihitung dengan menerapkan operator pada nilai dari kedua anaknya.

Pohon ekspresi aritmetika seperti ini merupakan pohon biner proper, karena setiap operator biner (seperti +, −, ×, /) membutuhkan tepat dua operand. Namun, jika kita mengizinkan operator unari seperti negasi (−), misalnya pada ekspresi “−x”, maka pohonnya bisa menjadi pohon biner yang tidak proper (improper).

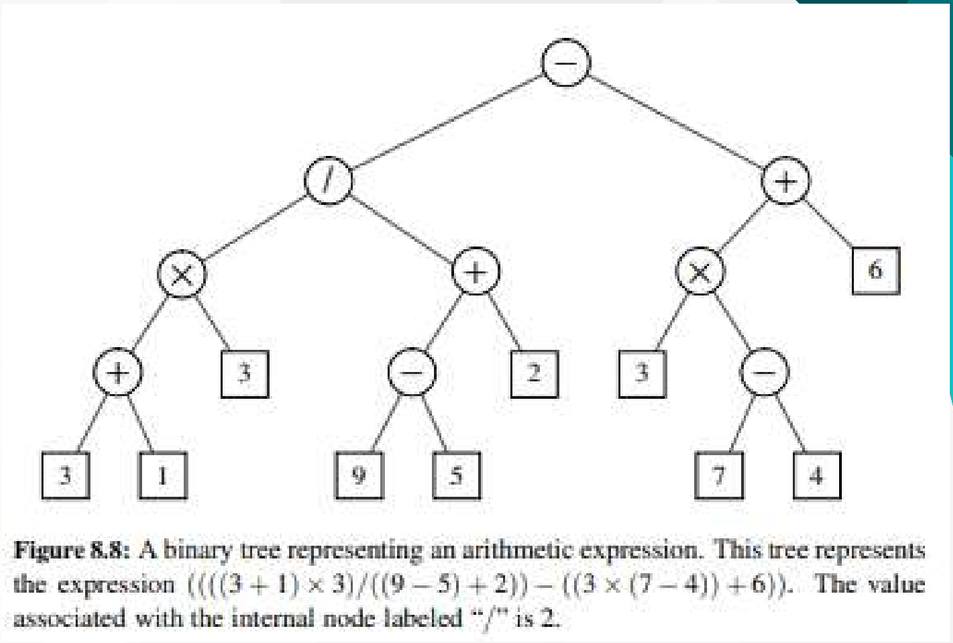


Figure 8.8: A binary tree representing an arithmetic expression. This tree represents the expression $((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$. The value associated with the internal node labeled “/” is 2.

Pohon biner juga dapat didefinisikan secara rekursif sebagai berikut:

Pohon biner adalah struktur yang entah kosong, atau terdiri dari:

- Sebuah simpul r yang disebut akar (root) dari T, yang menyimpan sebuah elemen,
- Sebuah pohon biner kiri (bisa kosong), yang disebut subpohon kiri dari T,
- Sebuah pohon biner kanan (bisa kosong), yang disebut subpohon kanan dari T.

Dengan definisi ini, pohon biner dibangun secara berlapis dari simpul-simpul dan dua subpohon, menjadikannya sangat cocok untuk diproses menggunakan algoritma rekursif.

8.2.1 The Binary Tree Abstract Data Type

Sebagai abstract data type (ADT), pohon biner merupakan spesialisasi dari pohon umum yang menyediakan tiga metode akses tambahan, yaitu:

- T.left(p): Mengembalikan posisi yang merepresentasikan anak kiri dari p, atau None jika tidak ada anak kiri.
- T.right(p): Mengembalikan posisi yang merepresentasikan anak kanan dari p, atau None jika tidak ada anak kanan.
- T.sibling(p): Mengembalikan posisi yang merepresentasikan saudara kandung (sibling) dari p, atau None jika tidak ada saudara kandung.

Sama seperti pada pembahasan ADT pohon di Bagian 8.1.2, di sini kita belum mendefinisikan metode pembaruan (update methods) khusus untuk pohon biner. Sebagai gantinya, kita akan membahas kemungkinan metode pembaruan tersebut saat menjelaskan implementasi dan aplikasi spesifik dari pohon biner nantinya.

8.2.1 The Binary Tree Abstract Data Type

Sama seperti kelas Tree yang didefinisikan sebagai abstract base class pada Bagian 8.1.2, kita juga mendefinisikan kelas BinaryTree untuk merepresentasikan ADT pohon biner. Kelas ini dibuat melalui pewarisan (inheritance) dari kelas Tree, sehingga mewarisi seluruh fungsionalitas pohon umum seperti parent, is_leaf, root, dan juga kelas bersarang Position. Namun, BinaryTree tetap bersifat abstrak karena belum menetapkan bagaimana struktur internalnya akan diimplementasikan, serta belum menyediakan seluruh perilaku secara lengkap.

Dalam Code Fragment 8.7, implementasi Python dari BinaryTree ditampilkan. Kelas ini mendeklarasikan metode-metode abstrak tambahan, yakni left(p) dan right(p), yang harus diimplementasikan oleh subclass konkret. Selain itu, terdapat dua metode konkret:

- sibling(p), yang menentukan saudara dari p dengan memanfaatkan parent, left, dan right. Jika p adalah akar (root) atau anak tunggal, maka sibling(p) mengembalikan None.
- children(p), yang menghasilkan iterator terhadap anak-anak p secara berurutan (kiri lalu kanan), dengan menggunakan perilaku yang diharapkan dari metode left dan right.

Dengan demikian, meskipun representasi internal simpul belum ditentukan, kelas ini menyediakan antarmuka dasar dan beberapa perilaku umum untuk digunakan dan dikembangkan lebih lanjut dalam subclass konkret.

IMPLEMENTATION IN PYTHON

```
class BinaryTree(Tree):
  """Abstract base class representing a binary tree structure."""

  # --------------- additional abstract methods ---------------
  def left(self, p):
    """Return a Position representing p's left child.

    Return None if p does not have a left child.
    """
    raise NotImplementedError('must be implemented by subclass')

  def right(self, p):
    """Return a Position representing p's right child.

    Return None if p does not have a right child.
    """
    raise NotImplementedError('must be implemented by subclass')

  # ---------- concrete methods implemented in this class ----------
  def sibling(self, p):
    """Return a Position representing p's sibling (or None if no sibling)."""
    parent = self.parent(p)
    if parent is None:                  # p must be the root
      return None                       # root has no sibling
    else:
      if p == self.left(parent):
        return self.right(parent)       # possibly None
      else:
        return self.left(parent)        # possibly None

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    if self.left(p) is not None:
      yield self.left(p)
    if self.right(p) is not None:
      yield self.right(p)
```

Code Fragment 8.7: A BinaryTree abstract base class that extends the existing Tree abstract base class from Code Fragments 8.1 and 8.2.

8.2.1 The Binary Tree Abstract Data Type

Pohon biner memiliki beberapa sifat menarik yang berkaitan dengan hubungan antara tinggi pohon dan jumlah simpul. Dalam hal ini, kita menyebut himpunan semua simpul dalam pohon T yang berada pada kedalaman (depth) d sebagai level d dari T.
Dalam pohon biner:

- Level 0 paling banyak memiliki 1 simpul (yaitu akar),
- Level 1 paling banyak memiliki 2 simpul (anak-anak dari akar),
- Level 2 paling banyak memiliki 4 simpul, dan seterusnya.

Secara umum, level ke-d dalam pohon biner dapat memiliki maksimal 2^d simpul.
Pola ini menggambarkan bagaimana jumlah simpul bertambah secara eksponensial seiring bertambahnya kedalaman pohon (lihat Gambar 8.9).

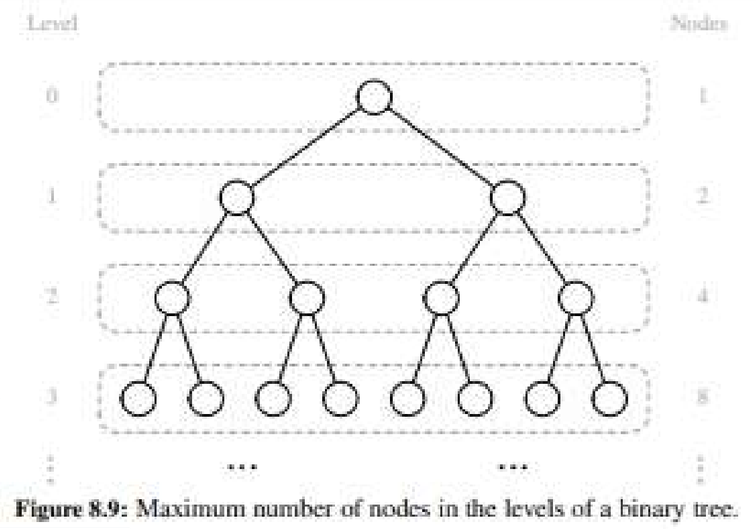


Figure 8.9: Maximum number of nodes in the levels of a binary tree.

8.2.1 The Binary Tree Abstract Data Type

Dari pola pertumbuhan jumlah simpul di setiap level pohon biner yang bersifat eksponensial, kita dapat menyimpulkan sejumlah proposisi matematis penting. Misalkan T adalah pohon biner yang tidak kosong, dengan:

- n = jumlah total simpul,
- nE = jumlah simpul eksternal (daun),
- nI = jumlah simpul internal,
- h = tinggi pohon.

Maka, pohon T memenuhi sifat-sifat berikut:

1. Jumlah simpul total: $h + 1 \leq n \leq 2^{(h+1)} - 1$
2. Jumlah simpul eksternal (daun): $1 \leq nE \leq 2^h$
3. Jumlah simpul internal: $h \leq nI \leq 2^h - 1$
4. Batas bawah dan atas tinggi pohon: $\log_2(n + 1) - 1 \leq h \leq n - 1$

Jika T adalah pohon biner proper (setiap simpul internal punya tepat dua anak), maka berlaku sifat-sifat khusus berikut:

1. Jumlah simpul total: $2h + 1 \leq n \leq 2^{(h+1)} - 1$
2. Jumlah simpul eksternal (daun): $h + 1 \leq nE \leq 2^h$
3. Jumlah simpul internal: $h \leq nI \leq 2^h - 1$
4. Batas tinggi pohon: $\log_2(n + 1) - 1 \leq h \leq (n - 1) / 2$

Sifat-sifat ini memberikan pemahaman mendalam tentang efisiensi struktur pohon, dan digunakan untuk menganalisis performa algoritma berbasis pohon, seperti pencarian, penyisipan, dan penyeimbangan pohon.

Relating Internal Nodes to External Nodes in a Proper Binary Tree

Dalam pohon biner proper yang tidak kosong, jumlah simpul eksternal nE selalu satu lebih banyak dari jumlah simpul internal nI, atau nE = nI + 1. Untuk membuktikannya, bayangkan proses pengurangan pohon secara bertahap dengan memindahkan simpul ke dua tumpukan: satu untuk simpul internal dan satu untuk eksternal.

Case 1: Jika pohon hanya memiliki satu simpul, simpul tersebut adalah akar sekaligus daun. Maka, simpul itu masuk ke tumpukan eksternal, dan tumpukan internal tetap kosong. Hasilnya, nE = 1, nI = 0, sehingga nE = nI + 1.

Case 2: Jika pohon memiliki lebih dari satu simpul, kita hapus sepasang simpul: satu simpul eksternal w dan induknya v (simpul internal). w masuk ke tumpukan eksternal, dan v ke tumpukan internal. Jika v punya induk u, maka anak v yang tersisa disambungkan kembali ke u, agar struktur pohon tetap proper. Proses ini diulang hingga tersisa satu simpul terakhir (daun), yang ditambahkan ke tumpukan eksternal. Dengan demikian, total simpul eksternal akan selalu satu lebih banyak dari simpul internal.

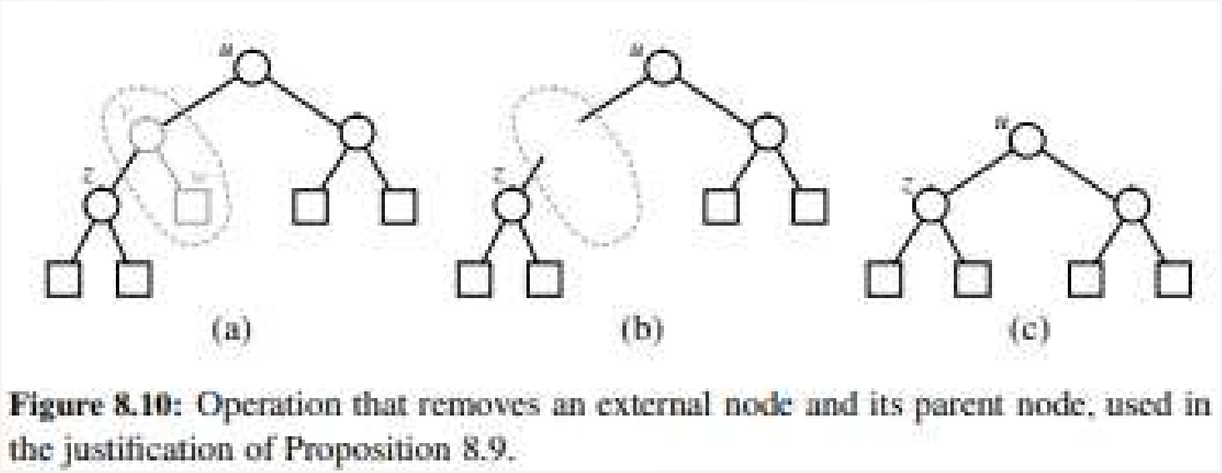


Figure 8.10: Operation that removes an external node and its parent node, used in the justification of Proposition 8.9.

8.3 IMPLEMENTING TREES

Kelas Tree dan BinaryTree yang telah didefinisikan sebelumnya dalam bab ini merupakan abstract base class secara formal. Meskipun sudah menyediakan banyak dukungan fungsional, keduanya tidak dapat diinstansiasi secara langsung karena belum memiliki detail implementasi tentang bagaimana pohon direpresentasikan secara internal dan bagaimana cara berpindah antara simpul induk dan anak. Agar menjadi implementasi konkret, sebuah kelas pohon harus menyertakan definisi metode seperti root, parent, num_children, children, __len__, dan khusus untuk BinaryTree, juga metode left dan right.

Terdapat beberapa pilihan untuk representasi internal pohon, dan bagian ini membahas representasi yang paling umum digunakan, dimulai dari pohon biner karena bentuk strukturnya yang lebih terdefinisi dengan jelas.

8.3.1 Linked Structure for Binary Trees

Cara yang alami untuk mengimplementasikan pohon biner T adalah dengan menggunakan struktur tertaut (linked structure). Setiap simpul menyimpan referensi ke elemen yang berada pada posisi p, serta referensi ke simpul anak kiri, anak kanan, dan induk dari p (lihat Gambar 8.11a). Jika p adalah akar dari T, maka referensi induknya adalah None. Demikian pula, jika p tidak memiliki anak kiri (atau anak kanan), maka referensi ke anak tersebut juga bernilai None.

Pohon itu sendiri menyimpan referensi ke simpul akar (jika ada), serta sebuah variabel bernama size yang menunjukkan jumlah total simpul dalam T. Representasi struktur tertaut untuk pohon biner ini ditunjukkan dalam Gambar 8.11b.

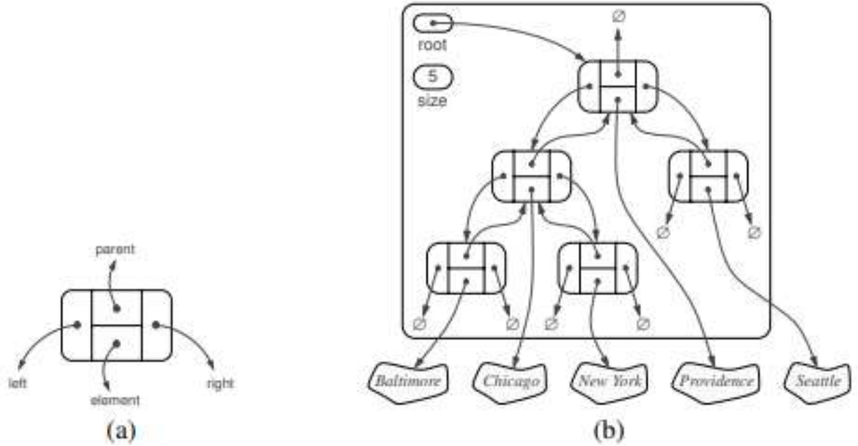


Figure 8.11: A linked structure for representing: (a) a single node; (b) a binary tree.

Python Implementation of a Linked Binary Tree Structure

Di bagian ini, kita membuat kelas LinkedBinaryTree sebagai versi nyata (konkret) dari pohon biner, dengan mewarisi dari kelas BinaryTree. Cara kerjanya mirip seperti PositionalList sebelumnya: kita punya kelas kecil bernama Node (untuk menyimpan data dan hubungan antar simpul), dan Position (untuk membungkus simpul dan berinteraksi dengan pengguna).

Agar aman saat mengakses simpul, kita pakai dua fungsi bantu:

- validate untuk memeriksa apakah posisi yang diberikan valid,
- make_position untuk mengubah simpul jadi posisi yang bisa dikembalikan ke pengguna.

Kelas Position ini juga mendukung perbandingan seperti p != q, karena diturunkan dari kelas posisi sebelumnya.

Selanjutnya, kita mendefinisikan konstruktor yang membuat pohon kosong (root = None, size = 0) dan mengisi metode-metode penting seperti root, parent, left, right, dan lainnya. Semua implementasi dilakukan dengan hati-hati agar tetap aman dan efisien.

Operations for Updating a Linked Binary Tree

Sejauh ini, kita baru menyediakan fungsi untuk memeriksa struktur pohon biner yang sudah ada. Namun, konstruktor LinkedBinaryTree hanya menghasilkan pohon kosong, dan kita belum menyediakan cara untuk mengubah struktur atau isi pohon tersebut.

Ada dua alasan mengapa metode pembaruan (update methods) tidak kita cantumkan di kelas abstrak Tree atau BinaryTree:

1. Efisiensi tergantung implementasi: Walaupun prinsip encapsulation menyarankan bahwa perilaku luar suatu kelas tidak perlu bergantung pada representasi internalnya, efisiensi operasi sangat bergantung pada cara struktur pohon diimplementasikan. Karena itu, lebih baik jika setiap kelas konkret menentukan sendiri metode pembaruannya agar paling sesuai dengan strukturnya.
2. Tidak semua aplikasi memerlukan metode pembaruan: Tidak semua jenis pohon atau penggunaannya boleh diubah sembarangan. Misalnya, jika kita menambahkan metode seperti T.replace(p, e)—yang mengganti elemen di posisi p dengan elemen baru e—maka metode ini tidak cocok untuk pohon ekspresi aritmatika, karena kita ingin memastikan bahwa simpul internal hanya menyimpan operator, bukan sembarang elemen.

Dengan kata lain, metode pembaruan lebih baik hanya disediakan pada kelas konkret dan spesifik, bukan diwariskan secara umum melalui kelas dasar.

Untuk pohon biner berbasis linked (tautan simpul), terdapat enam metode pembaruan (update methods) yang umum dan praktis untuk disediakan, yaitu:

1. add_root(e): Membuat akar pohon dengan elemen e (hanya jika pohon masih kosong).
2. add_left(p, e): Menambahkan anak kiri pada posisi p dengan elemen e, jika belum ada anak kiri.
3. add_right(p, e): Menambahkan anak kanan pada posisi p dengan elemen e, jika belum ada anak kanan.
4. replace(p, e): Mengganti elemen di posisi p dengan e, mengembalikan elemen lama.
5. delete(p): Menghapus simpul di posisi p hanya jika ia punya paling banyak satu anak, lalu menyambung anaknya (jika ada) ke atas.
6. attach(p, T1, T2): Menyambungkan pohon T1 dan T2 sebagai anak kiri dan kanan dari posisi daun p, lalu mengosongkan T1 dan T2.

Semua metode ini bisa diimplementasikan dalam waktu konstan O(1) dengan representasi linked tree karena hanya perlu manipulasi pointer antar simpul.

Namun, karena ada aplikasi tertentu yang tidak cocok jika sembarang orang bisa mengubah strukturnya (contohnya ekspresi aritmatika), maka metode-metode ini tidak dibuat publik secara langsung. Sebagai gantinya, versi nonpublik (misalnya _delete) disediakan, dan hanya bisa diakses dari dalam kelas atau subclass.

Jika suatu aplikasi memang butuh akses publik terhadap metode-metode ini, kita bisa membuat subclass seperti MutableLinkedBinaryTree yang secara eksplisit membuka akses publik terhadap enam metode update ini dengan membungkus versi nonpubliknya.

IMPLEMENTATION IN PYTHON

```
1  class LinkedBinaryTree(BinaryTree):
2    """Linked representation of a binary tree structure."""
3
4    class _Node:          # Lightweight, nonpublic class for storing a node.
5      __slots__ = '_element', '_parent', '_left', '_right'
6      def __init__(self, element, parent=None, left=None, right=None):
7        self._element = element
8        self._parent = parent
9        self._left = left
10       self._right = right
11
12   class Position(BinaryTree.Position):
13     """An abstraction representing the location of a single element."""
14
15     def __init__(self, container, node):
16       """Constructor should not be invoked by user."""
17       self._container = container
18       self._node = node
19
20     def element(self):
21       """Return the element stored at this Position."""
22       return self._node._element
```

```
23
24     def __eq__(self, other):
25       """Return True if other is a Position representing the same location."""
26       return type(other) is type(self) and other._node is self._node
27
28   def _validate(self, p):
29     """Return associated node, if position is valid."""
30     if not isinstance(p, self.Position):
31       raise TypeError('p must be proper Position type')
32     if p._container is not self:
33       raise ValueError('p does not belong to this container')
34     if p._node._parent is p._node:          # convention for deprecated nodes
35       raise ValueError('p is no longer valid')
36     return p._node
37
38   def _make_position(self, node):
39     """Return Position instance for given node (or None if no node)."""
40     return self.Position(self, node) if node is not None else None
```

Code Fragment 8.8: The beginning of our LinkedBinaryTree class (continued in Code Fragments 8.9 through 8.11).

IMPLEMENTATION IN PYTHON

```
41    #---------- binary tree constructor ----------
42    def __init__(self):
43      """Create an initially empty binary tree."""
44      self._root = None
45      self._size = 0
46
47    #---------- public accessors ----------
48    def __len__(self):
49      """Return the total number of elements in the tree."""
50      return self._size
51
52    def root(self):
53      """Return the root Position of the tree (or None if tree is empty)."""
54      return self._make_position(self._root)
55
56    def parent(self, p):
57      """Return the Position of p's parent (or None if p is root)."""
58      node = self._validate(p)
59      return self._make_position(node._parent)
60
```

```
61    def left(self, p):
62      """Return the Position of p's left child (or None if no left child)."""
63      node = self._validate(p)
64      return self._make_position(node._left)
65
66    def right(self, p):
67      """Return the Position of p's right child (or None if no right child)."""
68      node = self._validate(p)
69      return self._make_position(node._right)
70
71    def num_children(self, p):
72      """Return the number of children of Position p."""
73      node = self._validate(p)
74      count = 0
75      if node._left is not None:          # left child exists
76        count += 1
77      if node._right is not None:         # right child exists
78        count += 1
79      return count
```

Code Fragment 8.9: Public accessors for our LinkedBinaryTree class. The class begins in Code Fragment 8.8 and continues in Code Fragments 8.10 and 8.11.

IMPLEMENTATION IN PYTHON

```
80    def _add_root(self, e):
81      """Place element e at the root of an empty tree and return new Position.
82
83      Raise ValueError if tree nonempty.
84      """
85      if self._root is not None: raise ValueError('Root exists')
86      self._size = 1
87      self._root = self._Node(e)
88      return self._make_position(self._root)
89
90    def _add_left(self, p, e):
91      """Create a new left child for Position p, storing element e.
92
93      Return the Position of new node.
94      Raise ValueError if Position p is invalid or p already has a left child.
95      """
96      node = self._validate(p)
97      if node._left is not None: raise ValueError('Left child exists')
98      self._size += 1
99      node._left = self._Node(e, node)                    # node is its parent
100     return self._make_position(node._left)
```

```
101
102   def _add_right(self, p, e):
103     """Create a new right child for Position p, storing element e.
104
105     Return the Position of new node.
106     Raise ValueError if Position p is invalid or p already has a right child.
107     """
108     node = self._validate(p)
109     if node._right is not None: raise ValueError('Right child exists')
110     self._size += 1
111     node._right = self._Node(e, node)                   # node is its parent
112     return self._make_position(node._right)
113
114   def _replace(self, p, e):
115     """Replace the element at position p with e, and return old element."""
116     node = self._validate(p)
117     old = node._element
118     node._element = e
119     return old
```

Code Fragment 8.10: Nonpublic update methods for the LinkedBinaryTree class (continued in Code Fragment 8.11).

IMPLEMENTATION IN PYTHON

```
120   def _delete(self, p):
121     """Delete the node at Position p, and replace it with its child, if any.
122
123     Return the element that had been stored at Position p.
124     Raise ValueError if Position p is invalid or p has two children.
125     """
126     node = self._validate(p)
127     if self.num_children(p) == 2: raise ValueError('p has two children')
128     child = node._left if node._left else node._right        # might be None
129     if child is not None:
130       child._parent = node._parent      # child's grandparent becomes parent
131     if node is self._root:
132       self._root = child                # child becomes root
133     else:
134       parent = node._parent
135       if node is parent._left:
136         parent._left = child
137       else:
138         parent._right = child
139     self._size -= 1
140     node._parent = node                 # convention for deprecated node
141     return node._element
```

```
142
143   def _attach(self, p, t1, t2):
144     """Attach trees t1 and t2 as left and right subtrees of external p."""
145     node = self._validate(p)
146     if not self.is_leaf(p): raise ValueError('position must be leaf')
147     if not type(self) is type(t1) is type(t2):   # all 3 trees must be same type
148       raise TypeError('Tree types must match')
149     self._size += len(t1) + len(t2)
150     if not t1.is_empty( ):                       # attached t1 as left subtree of node
151       t1._root._parent = node
152       node._left = t1._root
153       t1._root = None                            # set t1 instance to empty
154       t1._size = 0
155     if not t2.is_empty( ):                       # attached t2 as right subtree of node
156       t2._root._parent = node
157       node._right = t2._root
158       t2._root = None                            # set t2 instance to empty
159       t2._size = 0
```

Code Fragment 8.11: Nonpublic update methods for the LinkedBinaryTree class (continued from Code Fragment 8.10).

Performance of the Linked Binary Tree Implementation

Berikut adalah ringkasan efisiensi dari representasi struktur tautan, berdasarkan analisis waktu eksekusi metode-metode dalam kelas LinkedBinaryTree, termasuk metode-metode turunan dari kelas Tree dan BinaryTree:

- Metode __len__, yang diimplementasikan dalam LinkedBinaryTree, menggunakan variabel instans yang menyimpan jumlah simpul dalam T dan memerlukan waktu O(1). Metode is_empty, yang diturunkan dari Tree, hanya memanggil len, sehingga juga memerlukan waktu O(1).
- Metode-metode akses seperti root, left, right, parent, dan num_children diimplementasikan langsung dalam LinkedBinaryTree dan semuanya berjalan dalam waktu O(1). Metode sibling dan children, yang diturunkan dari BinaryTree, bergantung pada sejumlah pemanggilan metode akses tersebut, sehingga juga berjalan dalam waktu O(1).
- Metode is_root dan is_leaf, yang berasal dari kelas Tree, keduanya berjalan dalam waktu O(1); is_root memanggil root dan menggunakan perbandingan posisi, sedangkan is_leaf memanggil left dan right lalu memeriksa apakah hasilnya adalah None.
- Metode depth dan height telah dianalisis dalam Bagian 8.1.3. Metode depth pada posisi p berjalan dalam waktu O(dp + 1), di mana dp adalah kedalaman posisi tersebut; sedangkan metode height pada akar pohon berjalan dalam waktu O(n), dengan n adalah jumlah total simpul.
- Metode pembaruan seperti add_root, add_left, add_right, replace, delete, dan attach (versi nonpublik) masing-masing berjalan dalam waktu O(1), karena hanya melibatkan perubahan hubungan antar simpul dalam jumlah konstan.

Tabel 8.1 merangkum performa implementasi struktur tautan untuk pohon biner.

Operation	Running Time
len, is_empty	$O(1)$
root, parent, left, right, sibling, children, num_children	$O(1)$
is_root, is_leaf	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$
add_root, add_left, add_right, replace, delete, attach	$O(1)$

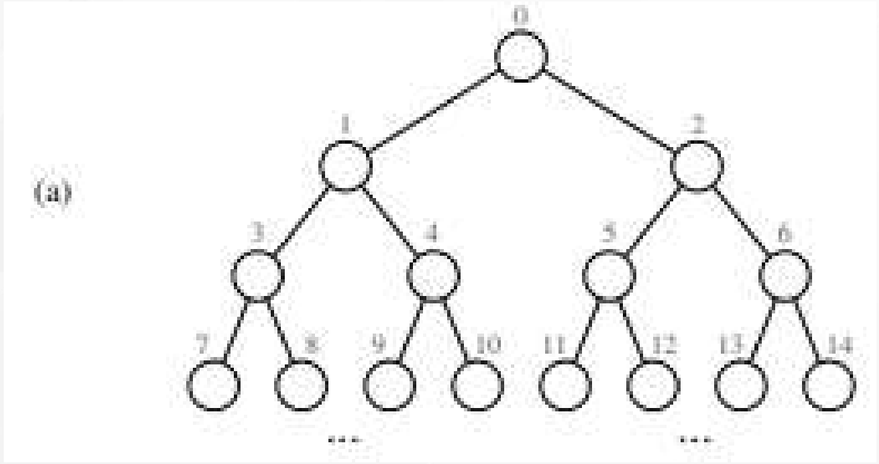
8.3.2 Array-Based Representation of a Binary Tree

Representasi alternatif dari pohon biner T dapat didasarkan pada cara memberikan nomor pada setiap posisi dalam T. Untuk setiap posisi p dalam T, diberikan fungsi bilangan bulat f(p) yang didefinisikan sebagai berikut:

- Jika p adalah akar dari T, maka f(p) = 0.
- Jika p adalah anak kiri dari posisi q, maka f(p) = 2·f(q) + 1.
- Jika p adalah anak kanan dari posisi q, maka f(p) = 2·f(q) + 2.

Fungsi penomoran f ini dikenal sebagai penomoran level (level numbering) dari posisi-posisi dalam pohon biner T, karena ia memberi nomor pada posisi dalam setiap level secara berurutan dari kiri ke kanan. (Lihat Gambar 8.12.)

Perlu dicatat bahwa penomoran level ini didasarkan pada posisi potensial dalam pohon, bukan hanya pada simpul yang benar-benar ada. Oleh karena itu, nomor yang dihasilkan tidak selalu berurutan. Misalnya, dalam Gambar 8.12(b), tidak ada simpul dengan penomoran level 13 atau 14 karena simpul bernomor 6 tidak memiliki anak.



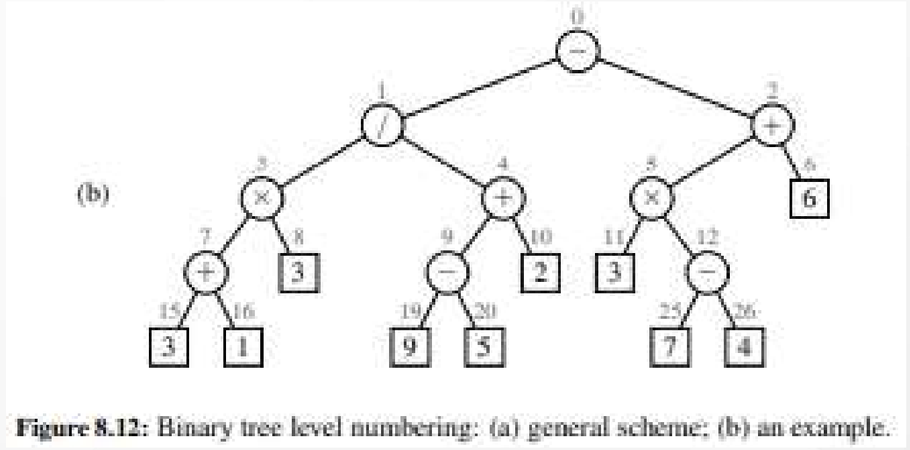
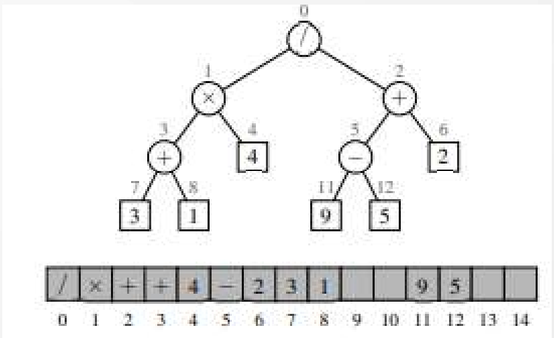


Figure 8.12: Binary tree level numbering: (a) general scheme; (b) an example.

8.3.2 Array-Based Representation of a Binary Tree

Fungsi penomoran level f memberikan ide untuk merepresentasikan pohon biner T dengan menggunakan struktur berbasis array (seperti list pada Python), di mana elemen pada posisi p dalam pohon disimpan pada indeks f(p) dalam array tersebut. Artinya, setiap posisi dalam pohon diberi nomor menggunakan aturan fungsi f, lalu elemen dari posisi itu diletakkan di indeks yang sesuai dalam array. Contoh representasi pohon biner berbasis array ini ditunjukkan pada Gambar 8.13.

Representasi pohon biner dengan array punya keuntungan karena posisi simpul bisa ditentukan hanya dengan satu angka, yaitu f(p). Berkat rumus f(p), kita bisa dengan mudah menemukan anak kiri, anak kanan, dan induk simpul dengan operasi matematika sederhana.
Namun, kelemahannya adalah penggunaan ruang bisa menjadi sangat boros jika bentuk pohon tidak seimbang. Misalnya, untuk menyimpan n simpul, bisa saja kita butuh array sepanjang 2n−1, karena banyak slot di array yang kosong. Ini membuat representasi ini tidak cocok untuk semua jenis pohon.
Meskipun begitu, ada kasus tertentu seperti heap di mana representasi array sangat efisien karena semua elemen terisi rapat. Tapi untuk pohon biner umum, ini kurang efisien, terutama untuk operasi seperti menghapus simpul dan menaikkan anaknya, yang bisa memakan waktu O(n) karena banyak elemen harus dipindahkan.



8.3.3 Linked Structure for General Trees

Dalam representasi pohon biner menggunakan struktur linked, setiap simpul secara eksplisit memiliki atribut left dan right sebagai referensi ke anak-anaknya. Namun, untuk pohon umum (general tree), jumlah anak dari satu simpul bisa tidak terbatas. Oleh karena itu, cara yang alami untuk merepresentasikan pohon umum T dengan struktur linked adalah dengan menyimpan satu kontainer (misalnya, sebuah list di Python) yang berisi referensi ke semua anak dari suatu simpul. Misalnya, sebuah simpul bisa memiliki atribut children yang berisi daftar referensi ke anak-anaknya (jika ada). Representasi ini ditunjukkan secara skematis dalam Gambar 8.14.

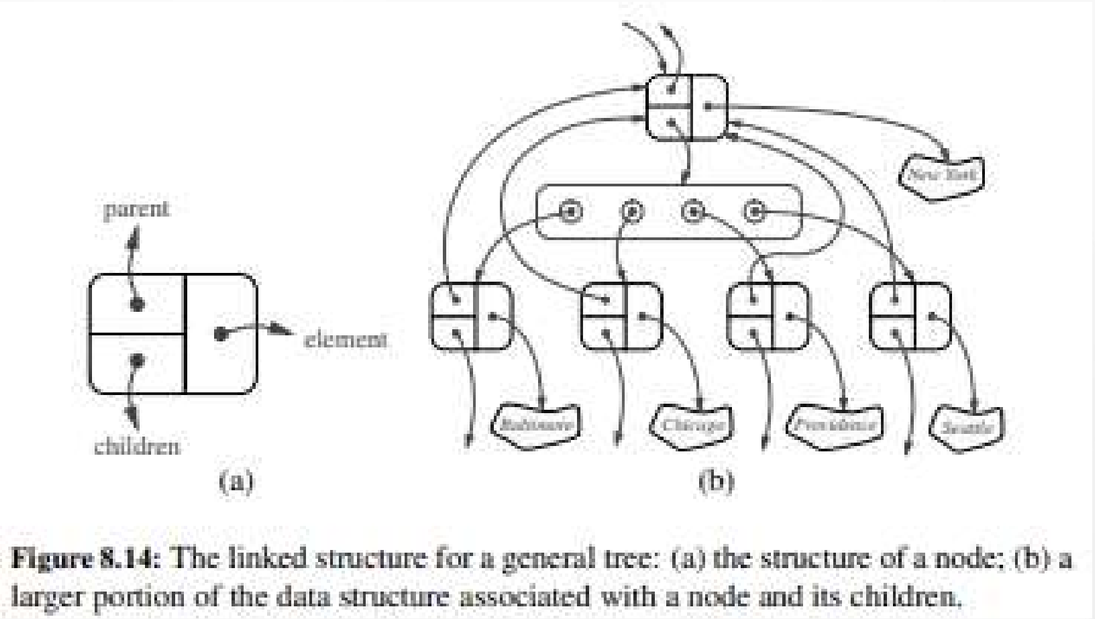


Figure 8.14: The linked structure for a general tree: (a) the structure of a node; (b) a larger portion of the data structure associated with a node and its children.

8.3.3 Linked Structure for General Trees

Dalam representasi pohon biner menggunakan struktur linked, setiap simpul secara eksplisit memiliki atribut left dan right sebagai referensi ke anak-anaknya. Namun, untuk pohon umum (general tree), jumlah anak dari satu simpul bisa tidak terbatas. Oleh karena itu, cara yang alami untuk merepresentasikan pohon umum T dengan struktur linked adalah dengan menyimpan satu kontainer (misalnya, sebuah list di Python) yang berisi referensi ke semua anak dari suatu simpul. Misalnya, sebuah simpul bisa memiliki atribut children yang berisi daftar referensi ke anak-anaknya (jika ada). Representasi ini ditunjukkan secara skematis dalam Gambar 8.14.

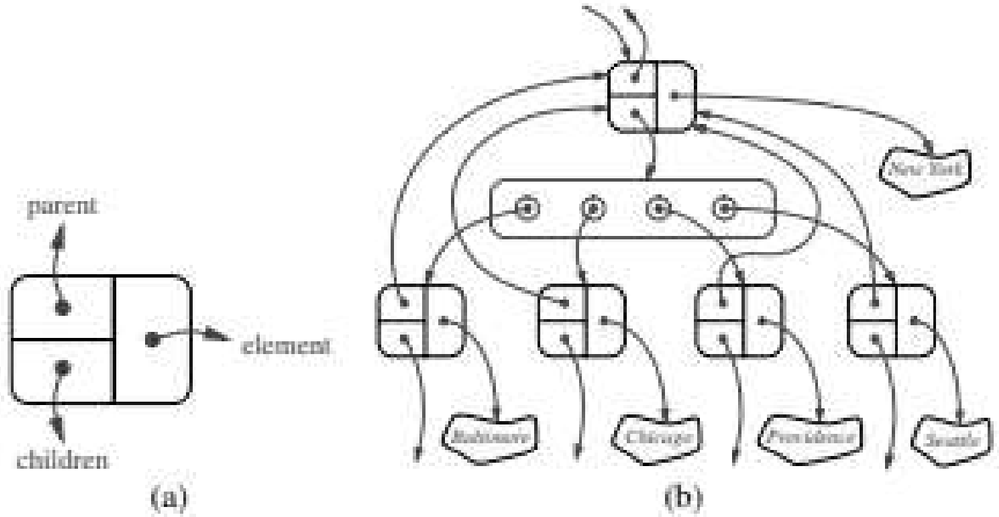


Figure 8.14: The linked structure for a general tree: (a) the structure of a node; (b) a larger portion of the data structure associated with a node and its children.

8.3.3 Linked Structure for General Trees

Tabel 8.2 merangkum performa implementasi pohon umum (general tree) dengan menggunakan struktur linked. Meskipun analisis rinci diserahkan sebagai latihan (R-8.14), perlu dicatat bahwa dengan menyimpan anak-anak dari setiap posisi p dalam suatu koleksi (seperti list), maka metode children(p) dapat diimplementasikan cukup dengan melakukan iterasi atas koleksi tersebut. Ini membuat operasi tersebut efisien dan langsung, karena tidak memerlukan pencarian tambahan.

Operation	Running Time
len, is_empty	$O(1)$
root, parent, is_root, is_leaf	$O(1)$
children(p)	$O(c_p + 1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Table 8.2: Running times of the accessor methods of an n-node general tree implemented with a linked structure. We let c_p denote the number of children of a position p. The space usage is $O(n)$.

8.4 TREE TRAVERSAL ALGORITHMS

Penelusuran (traversal) pohon T adalah cara sistematis untuk mengakses atau "mengunjungi" semua posisi dalam pohon tersebut. Tindakan spesifik saat "mengunjungi" suatu posisi p bergantung pada tujuan atau aplikasi penelusuran tersebut—bisa berupa sesuatu yang sederhana seperti menambah nilai penghitung, hingga melakukan komputasi kompleks pada p. Dalam bagian ini, akan dibahas beberapa skema penelusuran pohon yang umum digunakan, implementasinya dalam konteks kelas-kelas pohon yang telah kita definisikan sebelumnya, serta berbagai aplikasi umum dari penelusuran pohon.

8.4.1 Preorder and Postorder Traversals of General Trees

Dalam penelusuran preorder pada pohon T, simpul akar dari T dikunjungi terlebih dahulu, lalu penelusuran dilakukan secara rekursif ke setiap subpohon yang berakar pada anak-anaknya. Jika pohon memiliki urutan anak yang ditentukan (ordered tree), maka subpohon akan ditelusuri sesuai urutan tersebut.

```
Algorithm preorder(T, p):
    perform the "visit" action for position p
    for each child c in T.children(p) do
        preorder(T, c)              {recursively traverse the subtree rooted at c}
```

Code Fragment 8.12: Algorithm preorder for performing the preorder traversal of a subtree rooted at position p of a tree T.

Kode ini memastikan bahwa setiap simpul dikunjungi sebelum anak-anaknya. Gambar 8.15 dalam buku menunjukkan urutan kunjungan posisi dalam suatu contoh pohon saat algoritma preorder ini dijalankan.

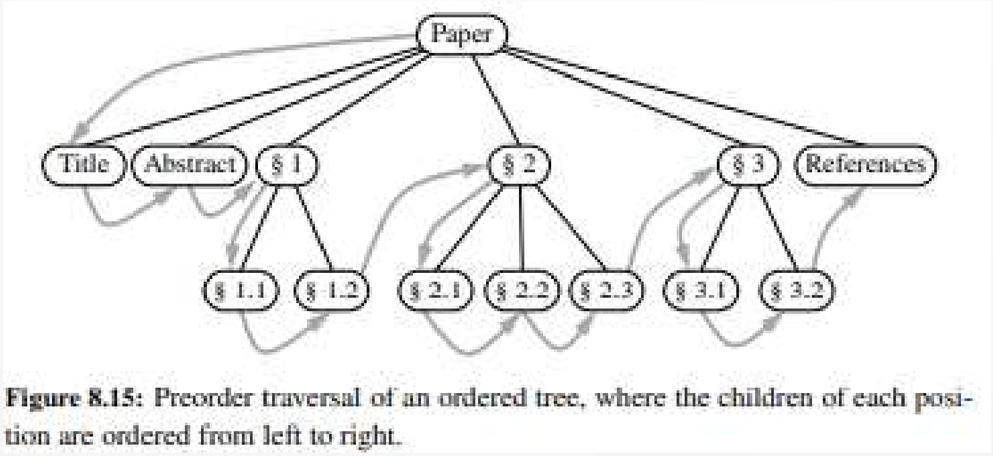


Figure 8.15: Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

Postorder Traversal

Penelusuran postorder adalah algoritma penting lainnya dalam traversal pohon. Berbeda dengan preorder yang mengunjungi simpul sebelum anak-anaknya, postorder terlebih dahulu menelusuri semua subpohon dari anak-anak suatu simpul secara rekursif, dan baru kemudian mengunjungi simpul itu sendiri. Karena itu, penelusuran ini disebut postorder — kunjungan dilakukan setelah semua anak selesai ditelusuri.

Pseudokode untuk algoritma postorder ditampilkan dalam Code Fragment 8.13, dan contoh urutan kunjungannya digambarkan dalam Gambar 8.16. Traversal ini sangat berguna dalam aplikasi seperti evaluasi ekspresi aritmatika, di mana kita perlu menyelesaikan perhitungan sub-bagian terlebih dahulu sebelum operator utama.

```
Algorithm postorder(T, p):
  for each child c in T.children(p) do
    postorder(T, c)                    {recursively traverse the subtree rooted at c}
  perform the "visit" action for position p
```

Code Fragment 8.13: Algorithm postorder for performing the postorder traversal of a subtree rooted at position p of a tree T.

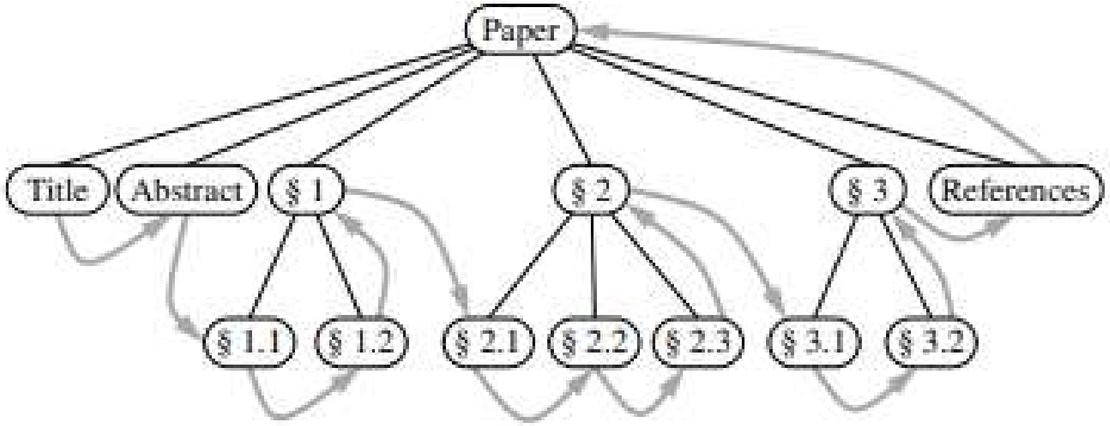


Figure 8.16: Postorder traversal of the ordered tree of Figure 8.15.

Running-Time Analysis

Baik preorder maupun postorder traversal merupakan cara yang efisien untuk mengunjungi seluruh posisi dalam sebuah pohon. Analisis kompleksitas waktunya serupa dengan algoritma height2 yang dijelaskan sebelumnya: pada setiap simpul p, bagian non-rekursif dari algoritma membutuhkan waktu O(cp + 1), di mana cp adalah jumlah anak dari p, dengan asumsi bahwa proses “kunjungan” memakan waktu O(1).

Berdasarkan Proposisi 8.5, jumlah total operasi selama traversal terhadap pohon T adalah O(n), di mana n adalah jumlah simpul dalam pohon. Ini adalah efisiensi terbaik yang mungkin (optimal secara asimtotik), karena semua simpul memang harus dikunjungi satu per satu selama traversal.

8.4.2 Breadth-First Tree Traversal

Selain preorder dan postorder, traversal pohon juga bisa dilakukan dengan breadth-first traversal atau penelusuran melebar, yaitu mengunjungi semua simpul pada kedalaman tertentu sebelum pindah ke kedalaman berikutnya. Metode ini sering digunakan pada game seperti Tic-Tac-Toe, di mana kita ingin mengevaluasi semua langkah awal terlebih dahulu, lalu semua kemungkinan balasannya, dan seterusnya.

Traversal ini menggunakan queue (antrian) agar simpul dikunjungi secara urut (FIFO). Setiap simpul yang dikunjungi, anak-anaknya dimasukkan ke antrian. Karena setiap simpul hanya diproses satu kali, waktu eksekusinya adalah O(n), dengan n jumlah simpul.

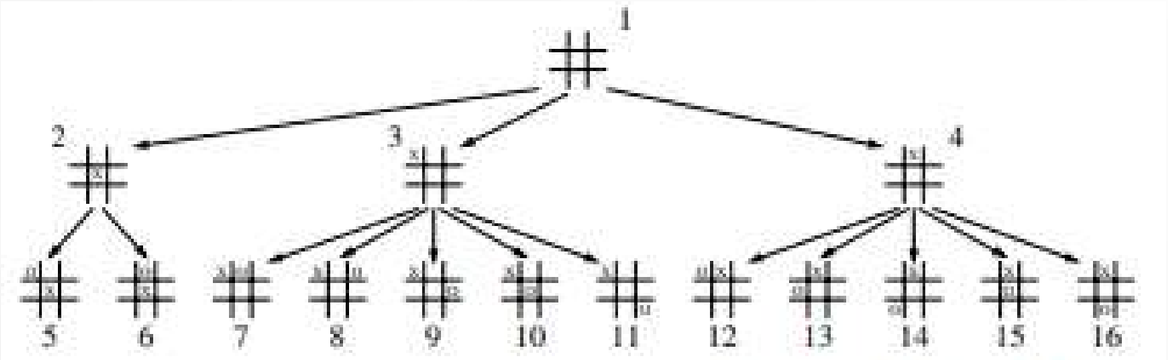


Figure 8.17: Partial game tree for Tic-Tac-Toe, with annotations displaying the order in which positions are visited in a breadth-first traversal.

```
Algorithm breadthfirst(T):
  Initialize queue Q to contain T.root( )
  while Q not empty do
    p = Q.dequeue( )                    {p is the oldest entry in the queue}
    perform the "visit" action for position p
    for each child c in T.children(p) do
      Q.enqueue(c)      {add p's children to the end of the queue for later visits}
```

Code Fragment 8.14: Algorithm for performing a breadth-first traversal of a tree.

8.4.3 Inorder Traversal of a Binary Tree

Traversal inorder adalah metode penelusuran pohon biner yang mengunjungi simpul dalam urutan: subpohon kiri → simpul saat ini → subpohon kanan. Metode ini menelusuri pohon dari kiri ke kanan, sehingga cocok untuk menyusun data secara berurutan, misalnya dalam binary search tree (BST).

```
Algorithm inorder(p):
  if p has a left child lc then
    inorder(lc)                    {recursively traverse the left subtree of p}
  perform the "visit" action for position p
  if p has a right child rc then
    inorder(rc)                    {recursively traverse the right subtree of p}
```

Code Fragment 8.15: Algorithm inorder for performing an inorder traversal of a subtree rooted at position p of a binary tree.

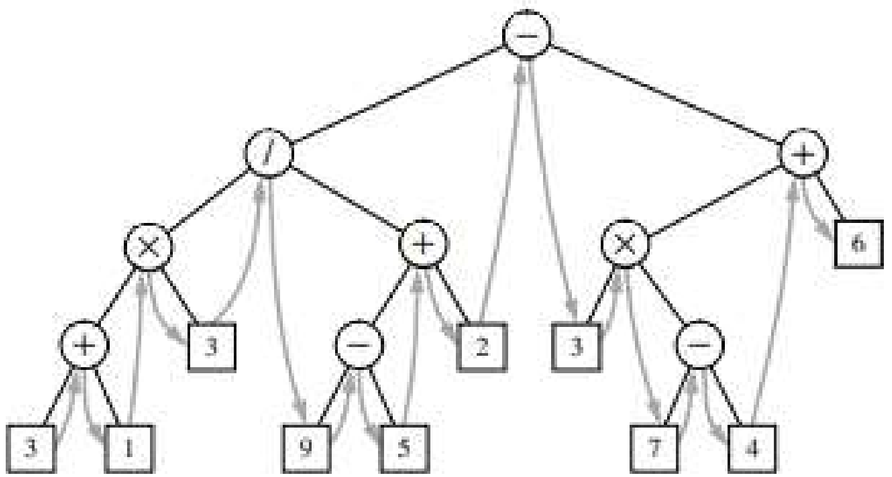


Figure 8.18: Inorder traversal of a binary tree.

Traversal ini juga berguna dalam pohon ekspresi aritmetika, karena menghasilkan urutan operasi seperti bentuk ekspresi matematika standar (misalnya: 3 + 1 × 3 ÷ 9 − 5 + 2).

Binary Search Trees

Salah satu penggunaan penting dari traversal inorder adalah untuk menyimpan urutan elemen yang terurut dalam struktur pohon yang disebut binary search tree (BST). Misalnya, jika kita punya kumpulan bilangan (himpunan S) yang bisa diurutkan, maka:

- Setiap posisi p pada BST menyimpan satu elemen e(p) dari himpunan.
- Semua elemen di subpohon kiri lebih kecil dari e(p).
- Semua elemen di subpohon kanan lebih besar dari e(p).

Karena aturan ini, traversal inorder pada BST akan menghasilkan elemen-elemen dalam urutan menaik (terurut). Untuk mencari sebuah nilai v dalam BST, kita mulai dari akar:

- Jika v < e(p), maka kita lanjut ke subpohon kiri.
- Jika v = e(p), maka pencarian berhasil.
- Jika v > e(p), maka kita lanjut ke subpohon kanan.
- Jika sampai pada simpul kosong, maka elemen tidak ditemukan.

BST bekerja seperti pohon keputusan (decision tree) yang mengevaluasi apakah nilai yang dicari lebih kecil, sama, atau lebih besar dari setiap simpul yang dilalui. Efisiensi pencarian dalam BST bergantung pada tinggi pohon:

- Kalau pohon seimbang, tinggi pohon mendekati log(n) → pencarian cepat.
- Kalau pohon tidak seimbang (mirip linked list), tingginya bisa mendekati n → pencarian jadi lambat.

Singkatnya, BST adalah cara cerdas menyimpan data terurut dalam struktur pohon, yang bisa membuat pencarian jadi efisien jika tinggi pohonnya kecil.

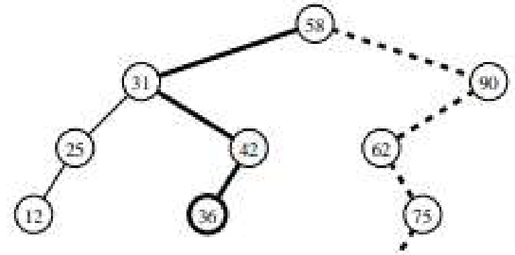


Figure 8.19: A binary search tree storing integers. The solid path is traversed when searching (successfully) for 36. The dashed path is traversed when searching (unsuccessfully) for 70.

8.4.4 Implementing Tree Traversals in Python

Saat pertama kali mendefinisikan ADT pohon (Tree ADT) di Bagian 8.1.2, disebutkan bahwa pohon T harus mendukung dua metode berikut:

- T.positions() : Menghasilkan iterasi dari semua posisi dalam pohon T.
- iter(T) : Menghasilkan iterasi dari semua elemen yang disimpan dalam pohon T.

Saat itu, belum ditentukan urutan apa yang digunakan saat iterasi dilakukan. Pada bagian ini, akan ditunjukkan bahwa semua traversal pohon yang telah dipelajari (seperti preorder, postorder, dll.) dapat digunakan untuk menghasilkan iterasi tersebut.

Untuk memulai, perlu diketahui bahwa kita bisa dengan mudah membuat iterasi atas semua elemen dalam pohon, jika sudah ada iterasi atas semua posisi. Oleh karena itu, dukungan terhadap sintaks iter(T) dapat dibuat secara formal dalam implementasi konkrit dari metode khusus __iter__ di kelas abstrak Tree.

Kita bisa menggunakan sintaks generator Python untuk menghasilkan iterasi. Contoh implementasinya dapat dilihat pada Code Fragment 8.16 berikut:

```
75    def __iter__(self):
76      """Generate an iteration of the tree's elements."""
77      for p in self.positions( ):          # use same order as positions()
78        yield p.element( )                 # but yield each element
```

Code Fragment 8.16: Iterating all elements of a Tree instance, based upon an iteration of the positions of the tree. This code should be included in the body of the Tree class.

Untuk mengimplementasikan metode positions(), kita bisa memilih traversal pohon mana yang digunakan. Karena setiap jenis traversal memiliki keuntungan tersendiri, kita bisa menyediakan implementasi traversal yang berbeda agar dapat dipanggil langsung oleh pengguna. Setelah itu, kita bisa memilih salah satu dari traversal tersebut sebagai urutan default untuk metode positions() dalam ADT pohon.

Preorder Traversal

Kita mulai dengan mempertimbangkan algoritma traversal preorder. Kita akan menyediakan sebuah metode publik dengan nama T.preorder() untuk pohon T, yang menghasilkan iterasi preorder dari semua posisi dalam pohon.
Namun, algoritma preorder traversal seperti pada Code Fragment 8.12 bersifat rekursif dan memerlukan posisi awal, yaitu akar subtree yang ditelusuri. Karena itu, kita akan:

- Membuat metode bantu nonpublik dengan parameter posisi.
- Metode publik preorder() akan memanggil metode bantu tersebut, dimulai dari akar pohon.

```
79   def preorder(self):
80     """Generate a preorder iteration of positions in the tree."""
81     if not self.is_empty( ):
82       for p in self._subtree_preorder(self.root( )):       # start recursion
83         yield p
84
85   def _subtree_preorder(self, p):
86     """Generate a preorder iteration of positions in subtree rooted at p."""
87     yield p                                  # visit p before its subtrees
88     for c in self.children(p):               # for each child c
89       for other in self._subtree_preorder(c):    # do preorder of c's subtree
90         yield other                          # yielding each to our caller
```

Code Fragment 8.17: Support for performing a preorder traversal of a tree. This code should be included in the body of the Tree class.

Karena kita menggunakan generator, traversal dilakukan dengan yield, bukan dengan aksi langsung. Pengguna bebas menentukan aksi saat posisi di-yield.

Preorder Traversal

Fungsi subtree_preorder() bersifat rekursif dan bekerja seperti ini:

- Pertama mengunjungi posisi p.
- Lalu menelusuri semua anak p satu per satu dengan rekursi.
- Jika p adalah daun, maka for c in self.children(p) akan dilewati (kasus dasar).

Dengan dukungan ini, pengguna dapat menulis:

```
for p in T.preorder( ):
  # "visit" position p
```

Selain itu, ADT pohon mensyaratkan metode:
T.positions() # menghasilkan iterasi seluruh posisi dalam pohon

```
91    def positions(self):
92      """Generate an iteration of the tree's positions."""
93      return self.preorder( )              # return entire preorder iteration
```

Code Fragment 8.18: An implementation of the positions method for the Tree class that relies on a preorder traversal to generate the results.

Dengan ini, traversal preorder sudah terintegrasi sepenuhnya ke dalam Tree ADT, dan dapat digunakan sebagai dasar dari iterasi dalam pohon.

Postorder Traversal

Kita dapat mengimplementasikan traversal postorder dengan teknik yang sangat mirip seperti pada traversal preorder. Perbedaannya adalah bahwa dalam metode rekursif utilitas untuk postorder, kita menunggu hingga seluruh posisi dalam subtree dari suatu simpul dikunjungi terlebih dahulu sebelum akhirnya mengunjungi simpul tersebut. Implementasinya ditunjukkan dalam Cuplikan Kode 8.19.

```
94    def postorder(self):
95      """Generate a postorder iteration of positions in the tree."""
96      if not self.is_empty( ):
97        for p in self._subtree_postorder(self.root( )):          # start recursion
98          yield p
99
100   def _subtree_postorder(self, p):
101     """Generate a postorder iteration of positions in subtree rooted at p."""
102     for c in self.children(p):                                  # for each child c
103       for other in self._subtree_postorder(c):                  # do postorder of c's subtree
104         yield other                                             # yielding each to our caller
105     yield p                                                     # visit p after its subtrees
```

Code Fragment 8.19: Support for performing a postorder traversal of a tree. This code should be included in the body of the Tree class.

Breadth-First Traversal

Dalam Cuplikan Kode 8.20, kami menyajikan implementasi algoritma traversal breadth-first dalam konteks kelas Tree. Ingat bahwa algoritma traversal breadth-first tidak bersifat rekursif; algoritma ini menggunakan antrean (queue) dari posisi-posisi untuk mengelola proses traversalnya. Implementasi ini menggunakan kelas LinkedQueue dari Bagian 7.1.2, meskipun implementasi antrean lainnya juga bisa digunakan.

```
106   def breadthfirst(self):
107     """Generate a breadth-first iteration of the positions of the tree."""
108     if not self.is_empty( ):
109       fringe = LinkedQueue( )                  # known positions not yet yielded
110       fringe.enqueue(self.root( ))             # starting with the root
111       while not fringe.is_empty( ):
112         p = fringe.dequeue( )                  # remove from front of the queue
113         yield p                                # report this position
114         for c in self.children(p):
115           fringe.enqueue(c)                    # add children to back of queue
```

Code Fragment 8.20: An implementation of a breadth-first traversal of a tree. This code should be included in the body of the Tree class.

Inorder Traversal for Binary Trees

Traversal preorder, postorder, dan breadth-first berlaku untuk semua jenis pohon, sehingga diimplementasikan dalam kelas abstrak Tree dan diwarisi oleh BinaryTree, LinkedBinaryTree, dan kelas turunannya.
Sementara itu, traversal inorder hanya berlaku untuk pohon biner karena bergantung pada anak kiri dan kanan. Oleh sebab itu, implementasinya disertakan khusus dalam kelas BinaryTree, menggunakan teknik serupa dengan preorder dan postorder.

```
37    def inorder(self):
38      """Generate an inorder iteration of positions in the tree."""
39      if not self.is_empty( ):
40        for p in self._subtree_inorder(self.root( )):
41          yield p
42
43    def _subtree_inorder(self, p):
44      """Generate an inorder iteration of positions in subtree rooted at p."""
45      if self.left(p) is not None:          # if left child exists, traverse its subtree
46        for other in self._subtree_inorder(self.left(p)):
47          yield other
48      yield p                               # visit p between its subtrees
49      if self.right(p) is not None:         # if right child exists, traverse its subtree
50        for other in self._subtree_inorder(self.right(p)):
51          yield other
```

Code Fragment 8.21: Support for performing an inorder traversal of a binary tree. This code should be included in the BinaryTree class (given in Code Fragment 8.7).

Untuk banyak aplikasi pohon biner, traversal inorder memberikan cara iterasi yang alami. Oleh karena itu, kita dapat menjadikannya sebagai metode default untuk kelas BinaryTree dengan mengganti (override) metode positions yang diwarisi dari kelas Tree (lihat Cuplikan Kode 8.22).

```
52    # override inherited version to make inorder the default
53    def positions(self):
54      """Generate an iteration of the tree's positions."""
55      return self.inorder( )                # make inorder the default
```

Code Fragment 8.22: Defining the BinaryTree.position method so that positions are reported using inorder traversal.

8.4.5 Applications of Tree Traversals

Pada bagian ini, kita menunjukkan beberapa contoh aplikasi traversal pohon, termasuk beberapa penyesuaian terhadap algoritma traversal standar.

Table of Contents

Ketika menggunakan pohon untuk merepresentasikan struktur hierarkis suatu dokumen, traversal preorder sangat cocok untuk menghasilkan daftar isi dokumen tersebut.

Contoh:

- Gambar 8.15 menunjukkan struktur pohon dari dokumen.
- Gambar 8.20 memperlihatkan daftar isinya dalam dua versi:
  - (a) Tanpa indentasi (satu elemen per baris)
  - (b) Dengan indentasi, disesuaikan dengan kedalaman elemen dalam pohon

Struktur seperti ini juga bisa digunakan untuk menampilkan isi sistem file komputer, karena direktori dan folder disusun dalam bentuk pohon.

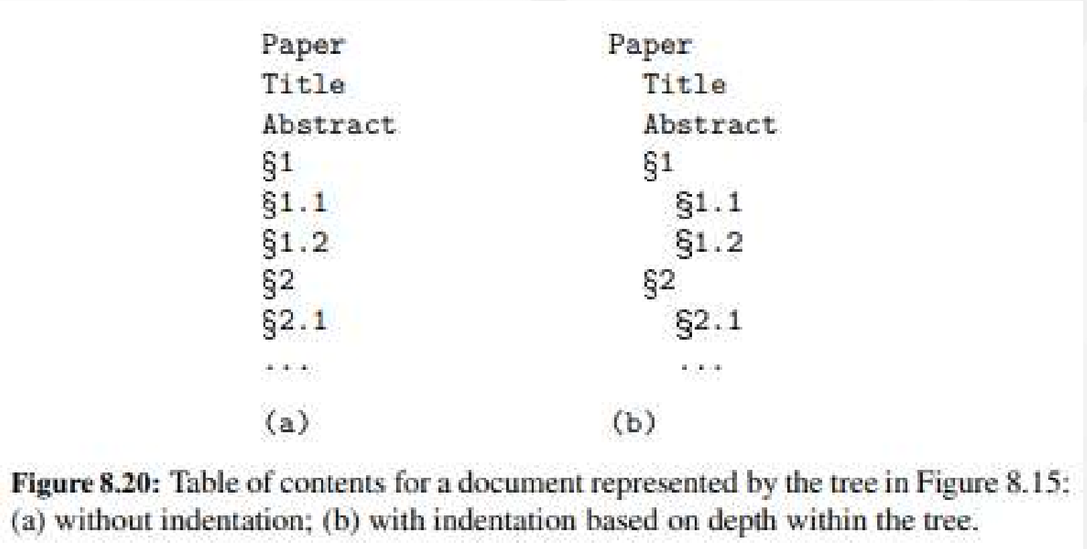


Figure 8.20: Table of contents for a document represented by the tree in Figure 8.15: (a) without indentation; (b) with indentation based on depth within the tree.

Table of Contents

Versi Tanpa Indentasi (Gambar 8.20a)
Jika kita punya pohon T, kita bisa cetak daftar isi tanpa indentasi seperti ini:

```
for p in T.preorder():
    print(p.element())
```

Versi Dengan Indentasi Sesuai Kedalaman (Gambar 8.20b)
Untuk membuat tampilan yang lebih rapi, kita bisa menambahkan indentasi dua spasi sesuai dengan kedalaman tiap elemen:

```
for p in T.preorder():
    print('  ' * T.depth(p) + str(p.element()))
```

Namun, pendekatan ini tidak efisien, karena fungsi T.depth(p) dipanggil untuk setiap posisi dalam pohon. Hal ini bisa menyebabkan waktu eksekusi menjadi $O(n^2)$ pada kasus terburuk, meskipun traversal preorder-nya sendiri berjalan dalam O(n) waktu (lihat analisis algoritma height1 di Bagian 8.1.3).
Untuk efisiensi yang lebih baik, sebaiknya kita menghitung kedalaman sekali, lalu meneruskannya secara rekursif saat traversal.

Table of Contents

Untuk menghasilkan daftar isi berindentasi dengan lebih efisien, kita tidak lagi memanggil fungsi depth berulang kali. Sebagai gantinya, kita mendesain rekursi top-down yang menyertakan kedalaman sebagai parameter tambahan. Implementasi ini ditunjukkan pada Code Fragment 8.23:

```
1  def preorder_indent(T, p, d):
2    """Print preorder representation of subtree of T rooted at p at depth d."""
3    print(2*d*' ' + str(p.element( )))              # use depth for indentation
4    for c in T.children(p):
5      preorder_indent(T, c, d+1)                   # child depth is d+1
```

Code Fragment 8.23: Efficient recursion for printing indented version of a preorder traversal. On a complete tree T, the recursion should be started with form preorder_indent(T, T.root( ), 0).

Table of Contents

Pada contoh Gambar 8.20, penomoran sudah ada di elemen-elemen pohon. Namun, pada kasus umum, kita ingin mencetak pohon dengan penomoran eksplisit, berdasarkan urutan dari akar ke simpul saat ini, seperti ini:

```
Electronics R'Us
  1 R&D
  2 Sales
    2.1 Domestic
    2.2 International
      2.2.1 Canada
      2.2.2 S. America
```

Penomoran ini tidak tersimpan secara langsung dalam pohon, melainkan tergantung pada urutan anak di setiap level. Solusi:

- Gunakan list untuk merepresentasikan jalur indeks dari akar ke posisi saat ini.
- Jangan duplikasi list; gunakan satu list yang sama di seluruh proses rekursi.
- Tambahkan satu indeks ke list sebelum rekursi ke anak, lalu hapus lagi setelahnya agar tidak meninggalkan jejak.

Pendekatan efisien ini ditunjukkan di Code Fragment 8.24 (yang akan diberikan di bagian berikutnya). Teknik ini mempertahankan efisiensi dan memungkinkan pembuatan struktur pohon yang rapi dan bernomor.

Berikut ini adalah fungsi preorder_label untuk mencetak representasi preorder dengan penomoran dan indentasi, seperti ditunjukkan dalam Code Fragment 8.24:

```
1  def preorder_label(T, p, d, path):
2    """Print labeled representation of subtree of T rooted at p at depth d."""
3    label = '.'.join(str(j+1) for j in path)    # displayed labels are one-indexed
4    print(2*d*'  ' + label, p.element( ))
5    path.append(0)                              # path entries are zero-indexed
6    for c in T.children(p):
7      preorder_label(T, c, d+1, path)           # child depth is d+1
8      path[-1] += 1
9    path.pop( )
```

Code Fragment 8.24: Efficient recursion for printing an indented and *labeled* presentation of a preorder traversal.

Parenthetic Representations of a Tree

Terkadang kita ingin merepresentasikan struktur pohon dalam bentuk string ringkas yang mudah diproses oleh komputer, bukan hanya untuk manusia. Salah satu cara adalah representasi parentetik.

Jika pohon T hanya terdiri dari satu posisi p:

$$P(T) = \text{str}(p.\text{element}()).$$

Jika p memiliki anak-anak T1, T2, ..., Tk, maka:

$$P(T) = \text{str}(p.\text{element}()) + \text{'('} + P(T_1) + \text{', '} + \cdots + \text{', '} + P(T_k) + \text{')'}$$

Misalnya, pohon dari Gambar 8.2 akan ditampilkan sebagai:

```
Electronics R'Us (R&D, Sales (Domestic, International (Canada,
S. America, Overseas (Africa, Europe, Asia, Australia))),
Purchasing, Manufacturing (TV, CD, Tuner))
```

Ini seperti traversal preorder, tetapi:

- Kita harus mencetak tanda kurung buka tepat sebelum mengeksplor anak-anak.
- Cetak koma di antara anak-anak.
- Cetak kurung tutup setelah semua anak dicetak.

Karena itu, implementasi preorder biasa (seperti di Code Fragment 8.17) tidak bisa langsung digunakan. Maka dibuat fungsi khusus bernama parenthesize, yang akan dijelaskan di Code Fragment 8.25.

```
1  def parenthesize(T, p):
2    """Print parenthesized representation of subtree of T rooted at p."""
3    print(p.element( ), end='')          # use of end avoids trailing newline
4    if not T.is_leaf(p):
5      first_time = True
6      for c in T.children(p):
7        sep = ' (' if first_time else ', '          # determine proper separator
8        print(sep, end='')
9        first_time = False                # any future passes will not be the first
10       parenthesize(T, c)                # recur on child
11     print(')', end='')                  # include closing parenthesis
```

Code Fragment 8.25: Function that prints parenthetic string representation of a tree.

Computing Disk Space

Struktur pohon bisa digunakan untuk memodelkan sistem file:

- Direktori = simpul internal
- File = simpul daun

Untuk menghitung total ruang yang dipakai suatu direktori dan seluruh isinya, kita menggunakan rekursi dengan pendekatan postorder traversal, karena, kita harus tahu dulu ukuran semua anak (subdirektori dan file) sebelum menghitung ukuran direktori induk.

Masalah dengan postorder biasa, metode postorder() hanya mengunjungi simpul, tanpa mengembalikan nilai seperti ukuran file. Jadi tidak cocok untuk menjumlahkan ukuran file ke induknya.

Solusi:

Gunakan rekursi yang mengembalikan nilai total ke atas, seperti ini:

```
1  def disk_space(T, p):
2    """Return total disk space for subtree of T rooted at p."""
3    subtotal = p.element( ).space( )              # space used at position p
4    for c in T.children(p):
5      subtotal += disk_space(T, c)                # add child's space to subtotal
6    return subtotal
```

Code Fragment 8.26: Recursive computation of disk space for a tree. We assume that a space( ) method of each tree element reports the local space used at that position.

Inti Logika:

- Ambil ukuran dari simpul saat ini
- Tambahkan hasil rekursi dari semua anak
- Kembalikan total ukuran

Kode ini mengasumsikan setiap elemen pohon punya metode .space() untuk mengembalikan ukuran file/direktori.

8.4.6 Euler Tours and the Template Method Pattern

Berbagai aplikasi dalam Section 8.4.5 menunjukkan kekuatan traversal pohon secara rekursif. Namun, implementasi traversal standar seperti preorder, postorder, dan inorder tidak cukup fleksibel untuk semua kebutuhan. Dalam beberapa kasus:

- Kita perlu pekerjaan awal sebelum rekursi dilakukan ke subtree,
- Pekerjaan tambahan setelah rekursi selesai,
- Untuk pohon biner, terkadang diperlukan aksi di antara dua rekursi (kiri dan kanan),
- Kita perlu tahu kedalaman simpul, jalur dari akar, atau mengembalikan informasi dari anak ke induk.

Meskipun solusi khusus bisa dibuat untuk setiap kasus, prinsip object-oriented programming menekankan pentingnya fleksibilitas dan keterpakaiulangan (reusability).

Euler Tour Traversal

Sebagai solusi umum, kita perkenalkan Euler tour traversal.
Traversal ini dapat dibayangkan sebagai jalan mengelilingi pohon T dari akar ke anak paling kiri, dengan anggapan setiap tepi pohon adalah "dinding" yang selalu kita jaga di sisi kiri.
(Lihat Gambar .21).

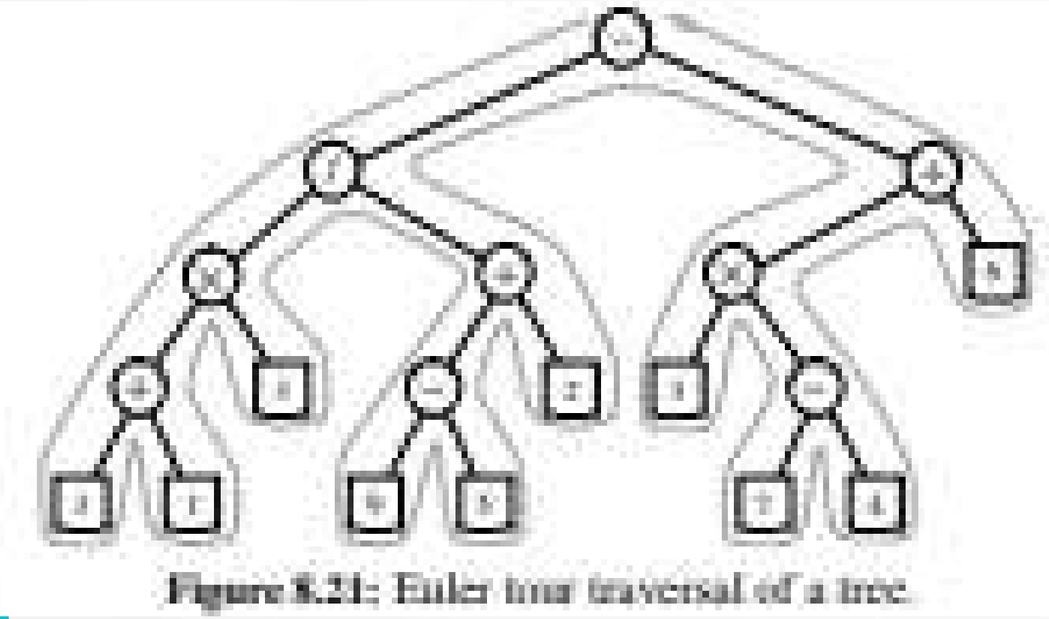


Figure 8.21: Euler tour traversal of a tree

Kompleksitas waktu:
O(n), karena setiap dari (n−1) sisi dilewati dua kali:

1. Sekali saat turun,
2. Sekali lagi saat naik.

Untuk menyatukan konsep preorder dan postorder, kita anggap bahwa setiap simpul p dikunjungi dua kali:

- Pre-visit: saat pertama kali mencapai simpul (dari sisi kiri),
- Post-visit: saat kita naik meninggalkan simpul (melewati sisi kanan).

Euler Tour Traversal

Traversal Euler Tour dapat dibayangkan sebagai proses rekursif, karena untuk setiap simpul (node) p:

1. Dilakukan aksi pre-visit saat pertama kali mengunjungi simpul p.
2. Kemudian dilakukan traversal secara rekursif ke setiap anak dari p.
3. Setelah semua anak selesai, dilakukan aksi post-visit pada p.

📌 Contoh dari Gambar 8.21:

- Misalkan simpul dengan elemen "/" memiliki dua anak.
- Maka akan dilakukan:
  - Euler tour untuk anak kiri
  - Euler tour untuk anak kanan
  - Semuanya terjadi di antara pre-visit dan post-visit simpul "/".

Pseudo-code Euler Tour (Fragment 8.27)

```
Algorithm eulertour(T, p):
    perform the "pre visit" action for position p
    for each child c in T.children(p) do
        eulertour(T, c)          {recursively tour the subtree rooted at c}
    perform the "post visit" action for position p
```

Code Fragment 8.27: Algorithm eulertour for performing an Euler tour traversal of a subtree rooted at position p of a tree.

Template Method Pattern

Template Method Pattern adalah pola desain dalam pemrograman berorientasi objek yang menyediakan kerangka kerja algoritma dasar, namun tetap fleksibel karena bisa disesuaikan di bagian-bagian tertentu. Penyesuaian ini dilakukan lewat fungsi tambahan yang disebut hook.

Dalam Euler Tour Traversal, digunakan dua hook, yaitu previsit (dijalankan sebelum menelusuri anak-anak suatu node) dan postvisit (dijalankan setelah semua anak selesai ditelusuri). Dengan cara ini, kita bisa menambahkan logika tertentu tanpa harus mengubah struktur utama dari traversal.

Struktur umum ini biasanya dituliskan dalam kelas EulerTour. Untuk menyesuaikan perilaku traversal, kita cukup membuat subclass dari EulerTour dan mengganti isi previsit atau postvisit sesuai kebutuhan. Cara ini memudahkan kita membuat aplikasi traversal seperti menghitung ukuran folder, menampilkan isi pohon dengan indentasi, atau menghitung ekspresi matematika, tanpa harus mengulang kode dasar traversal.

Python Implementation

Implementasi EulerTour pada Python mencakup proses rekursif utama yang ditangani oleh metode non-publik bernama tour. Untuk menggunakannya, kita cukup membuat objek dari kelas EulerTour dengan memberikan pohon sebagai parameter. Setelah itu, proses traversal dimulai dengan memanggil metode publik execute. Metode ini menjalankan seluruh perjalanan Euler dari akar pohon dan mengembalikan hasil akhir dari proses komputasi yang dilakukan selama traversal.

```
1  class EulerTour:
2    """Abstract base class for performing Euler tour of a tree.
3
4    _hook_previsit and _hook_postvisit may be overridden by subclasses.
5    """
6    def __init__(self, tree):
7      """Prepare an Euler tour template for given tree."""
8      self._tree = tree
9
10   def tree(self):
11     """Return reference to the tree being traversed."""
12     return self._tree
13
14   def execute(self):
15     """Perform the tour and return any result from post visit of root."""
16     if len(self._tree) > 0:
17       return self._tour(self._tree.root( ), 0, [ ])          # start the recursion
18
19   def _tour(self, p, d, path):
20     """Perform tour of subtree rooted at Position p.
21
22       p           Position of current node being visited
23       d           depth of p in the tree
24       path        list of indices of children on path from root to p
25       """
26     self._hook_previsit(p, d, path)                          # "pre visit" p
27     results = [ ]
28     path.append(0)              # add new index to end of path before recursion
29     for c in self._tree.children(p):
30       results.append(self._tour(c, d+1, path))      # recur on child's subtree
31       path[-1] += 1               # increment index
32     path.pop( )                   # remove extraneous index from end of path
33     answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
34     return answer
35
36   def _hook_previsit(self, p, d, path):                      # can be overridden
37     pass
38
39   def _hook_postvisit(self, p, d, path, results):            # can be overridden
40     pass
```

Code Fragment 8.28: An EulerTour base class providing a framework for performing Euler tour traversals of a tree.

Fitur Tambahan di EulerTour Traversal

Untuk mendukung traversal kompleks, EulerTour ditingkatkan dengan:
1. Pelacakan Kedalaman dan Jalur
Traversal kini mencatat:
- depth: kedalaman node saat ini,
- path: indeks jalur dari root ke node (seperti pada Code Fragment 8.24).

2. Pengembalian Nilai dari Anak ke Induk
Masing-masing postvisit anak bisa mengembalikan nilai yang akan digunakan induknya.

Dua Hook yang Bisa Dikustomisasi

- previsit(p, d, path)
- → Dipanggil sebelum anak-anak ditelusuri.
- → Tidak mengembalikan nilai.
- postvisit(p, d, path, results)
- → Dipanggil setelah semua anak ditelusuri.
- → Bisa mengembalikan nilai ke induk.

Tambahan

Subclass dapat menyimpan state lewat variabel instansi jika dibutuhkan untuk tugas tertentu.

Using the Euler Tour Framework

Contoh Penggunaan Subclass EulerTour

Berikut adalah beberapa contoh aplikasi traversal menggunakan turunan (subclass) dari kerangka kerja EulerTour.

Preorder Indented Traversal

(Mirip Code Fragment 8.23)

Menampilkan elemen dengan indentasi berdasarkan kedalaman:

```
1  class PreorderPrintIndentedTour(EulerTour):
2    def _hook_previsit(self, p, d, path):
3      print(2*d*' ' + str(p.element( )))
```

Code Fragment 8.29: A subclass of EulerTour that produces an indented preorder list of a tree's elements.

Pemanggilan:

```
tour = PreorderPrintIndentedTour(T)
tour.execute( )
```

Preorder Indented + Labeled Traversal

(Mirip Code Fragment 8.24)

Menambahkan label berdasarkan urutan anak:

```
1  class PreorderPrintIndentedLabeledTour(EulerTour):
2    def _hook_previsit(self, p, d, path):
3      label = '.'.join(str(j+1) for j in path)    # labels are one-indexed
4      print(2*d*' ' + label, p.element( ))
```

Code Fragment 8.30: A subclass of EulerTour that produces a labeled and indented, preorder list of a tree's elements.

Using the Euler Tour Framework

Parenthetic Representation
(Mirip Code Fragment 8.25)
Menampilkan representasi string pohon dengan tanda kurung:

```
1  class ParenthesizeTour(EulerTour):
2    def _hook_previsit(self, p, d, path):
3      if path and path[-1] > 0:                # p follows a sibling
4        print(', ', end='')                    # so preface with comma
5      print(p.element( ), end='')              # then print element
6      if not self.tree( ).is_leaf(p):          # if p has children
7        print(' (', end='')                    # print opening parenthesis
8
9    def _hook_postvisit(self, p, d, path, results):
10     if not self.tree( ).is_leaf(p):          # if p has children
11       print(')', end='')                     # print closing parenthesis
```

Code Fragment 8.31: A subclass of EulerTour that prints a parenthetic string representation of a tree.

Menghitung Ukuran Disk (Disk Space)
(Mirip Code Fragment 8.26)
Menjumlahkan ukuran node dan semua anaknya:

```
1  class DiskSpaceTour(EulerTour):
2    def _hook_postvisit(self, p, d, path, results):
3      # we simply add space associated with p to that of its subtrees
4      return p.element( ).space( ) + sum(results)
```

Code Fragment 8.32: A subclass of EulerTour that computes disk space for a tree.

```
1  class BinaryEulerTour(EulerTour):
2    """Abstract base class for performing Euler tour of a binary tree.
3
4    This version includes an additional _hook_invisit that is called after the tour
5    of the left subtree (if any), yet before the tour of the right subtree (if any).
6
7    Note: Right child is always assigned index 1 in path, even if no left sibling.
8    """
9    def _tour(self, p, d, path):
10     results = [None, None]          # will update with results of recursions
11     self._hook_previsit(p, d, path)                  # "pre visit" for p
12     if self._tree.left(p) is not None:               # consider left child
13       path.append(0)
14       results[0] = self._tour(self._tree.left(p), d+1, path)
15       path.pop( )
16     self._hook_invisit(p, d, path)                   # "in visit" for p
17     if self._tree.right(p) is not None:              # consider right child
18       path.append(1)
19       results[1] = self._tour(self._tree.right(p), d+1, path)
20       path.pop( )
21     answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
22     return answer
23
24   def _hook_invisit(self, p, d, path): pass          # can be overridden
```

Code Fragment 8.33: A BinaryEulerTour base class providing a specialized tour for binary trees. The original EulerTour base class was given in Code Fragment 8.28.

THE EULER TOUR TRAVERSAL OF A BINARY TREE

Pada Section 8.4.6, konsep Euler tour traversal dikembangkan untuk pohon biner dengan membuat subclass bernama BinaryEulerTour. Berbeda dari versi umum (EulerTour), subclass ini menambahkan satu hook method tambahan, yaitu hook_invisit, yang dipanggil di antara traversal anak kiri dan kanan. Ini berguna untuk mendukung traversal seperti inorder yang khas pada pohon biner. Jika sebuah node hanya memiliki satu anak, invisit tetap dipanggil pada posisi yang sesuai. Untuk node daun, ketiga hook (previsit, invisit, postvisit) akan dijalankan secara berurutan. Pendekatan ini memberikan cara fleksibel dan terstruktur untuk menyesuaikan proses traversal sesuai kebutuhan pohon biner.

The Euler Tour Traversal of a Binary Tree

Untuk menunjukkan fleksibilitas BinaryEulerTour, dibuat subclass bernama BinaryLayout yang menghitung koordinat (x, y) untuk setiap node dalam pohon biner, seperti ilustrasi pada Gambar 8.22. Tujuannya adalah mengatur posisi visual tiap simpul agar bisa digambar di layar.
Aturannya sebagai berikut:

- x(p) = jumlah node yang sudah dikunjungi sebelum posisi p dalam traversal inorder (menentukan posisi horizontal).
- y(p) = kedalaman (depth) dari node p (menentukan posisi vertikal).

Dalam konteks grafis komputer:

- sumbu x meningkat dari kiri ke kanan,
- sumbu y meningkat dari atas ke bawah (asal koordinat di pojok kiri atas layar).

Kelas BinaryLayout meng-override metode hook_invisit, karena koordinat x ditentukan saat inorder traversal. Ia juga menyimpan variabel instansi count untuk menghitung berapa banyak node yang sudah dikunjungi.
Berikut bagian penting implementasinya:

```
1  class BinaryLayout(BinaryEulerTour):
2    """Class for computing (x,y) coordinates for each node of a binary tree."""
3    def __init__(self, tree):
4      super().__init__(tree)                  # must call the parent constructor
5      self._count = 0                         # initialize count of processed nodes
6
7    def _hook_invisit(self, p, d, path):
8      p.element().setX(self._count)           # x-coordinate serialized by count
9      p.element().setY(d)                     # y-coordinate is depth
10     self._count += 1                        # advance count of processed nodes
```

Code Fragment 8.34: A BinaryLayout class that computes coordinates at which to draw positions of a binary tree. We assume that the element type for the original tree supports setX and setY methods.

8.5 CASE STUDY: AN EXPRESSION TREE

Pada bagian ini, diperkenalkan kelas ExpressionTree, sebuah subclass dari LinkedBinaryTree, untuk merepresentasikan ekspresi aritmatika menggunakan pohon biner. Setiap node internal menyimpan operator (seperti +, -, ×, /), dan setiap daun menyimpan nilai numerik. Kelas ini mendukung dua bentuk inisialisasi: (1) ExpressionTree(value) untuk membuat pohon dengan satu nilai, dan (2) ExpressionTree(op, E1, E2) untuk membentuk pohon baru dengan op sebagai akar dan dua subtree E1 dan E2.

Composing a Parenthesized String Representation

Untuk menampilkan ekspresi seperti (((3+1)×4)/((9−5)+2)), digunakan traversal inorder dengan tambahan tanda kurung: kurung buka dicetak sebelum menelusuri anak kiri (preorder step), dan kurung tutup setelah menelusuri anak kanan (postorder step). Hal ini diimplementasikan dalam metode __str__() dengan bantuan metode rekursif parenthesize_recur() yang membangun representasi string secara efisien. Pendekatan ini memungkinkan kita membuat, menampilkan, dan mengevaluasi ekspresi matematika kompleks secara terstruktur.

PYTHON IMPLEMENTATION

```
1  class ExpressionTree(LinkedBinaryTree):
2    """An arithmetic expression tree."""
3
4    def __init__(self, token, left=None, right=None):
5      """Create an expression tree.
6
7      In a single parameter form, token should be a leaf value (e.g., '42'),
8      and the expression tree will have that value at an isolated node.
9
10     In a three-parameter version, token should be an operator,
11     and left and right should be existing ExpressionTree instances
12     that become the operands for the binary operator.
13     """
14     super().__init__()                          # LinkedBinaryTree initialization
15     if not isinstance(token, str):
16       raise TypeError('Token must be a string')
17     self._add_root(token)                       # use inherited, nonpublic method
18     if left is not None:                        # presumably three-parameter form
19       if token not in '+-*x/':
20         raise ValueError('token must be valid operator')
21       self._attach(self.root(), left, right)   # use inherited, nonpublic method
22
23   def __str__(self):
24     """Return string representation of the expression."""
25     pieces = []                                 # sequence of piecewise strings to compose
26     self._parenthesize_recur(self.root(), pieces)
27     return ''.join(pieces)
28
29   def _parenthesize_recur(self, p, result):
30     """Append piecewise representation of p's subtree to resulting list."""
31     if self.is_leaf(p):
32       result.append(str(p.element()))           # leaf value as a string
33     else:
34       result.append('(')                        # opening parenthesis
35       self._parenthesize_recur(self.left(p), result)  # left subtree
36       result.append(p.element())                # operator
37       self._parenthesize_recur(self.right(p), result) # right subtree
38       result.append(')')                        # closing parenthesis
```

Code Fragment 8.35: The beginning of an ExpressionTree class.

Evaluasi Pohon Ekspresi (Expression Tree Evaluation)

Evaluasi nilai numerik dari sebuah Expression Tree dilakukan dengan postorder traversal. Artinya, kita menghitung nilai dari subtree kiri dan kanan terlebih dahulu, lalu menerapkan operator yang ada di node induk.

Logika Evaluasi (Pseudo-code):

```
Algorithm evaluate_recur(p):
  if p is a leaf then
    return the value stored at p
  else
    let ○ be the operator stored at p
    x ← evaluate_recur(left(p))
    y ← evaluate_recur(right(p))
    return x ○ y
```

Code Fragment 8.36: Algorithm evaluate_recur for evaluating the expression represented by a subtree of an arithmetic expression tree rooted at position p.

Implementasi di Python:

```
39  def evaluate(self):
40    """Return the numeric result of the expression."""
41    return self._evaluate_recur(self.root( ))
42
43  def _evaluate_recur(self, p):
44    """Return the numeric result of subtree rooted at p."""
45    if self.is_leaf(p):
46      return float(p.element( ))          # we assume element is numeric
47    else:
48      op = p.element( )
49      left_val = self._evaluate_recur(self.left(p))
50      right_val = self._evaluate_recur(self.right(p))
51      if op == '+': return left_val + right_val
52      elif op == '-': return left_val - right_val
53      elif op == '/': return left_val / right_val
54      else: return left_val * right_val       # treat '*' or 'x' as multiplication
```

Code Fragment 8.37: Support for evaluating an ExpressionTree instance.

Metode evaluate() akan mengembalikan nilai akhir dari seluruh ekspresi, sementara _evaluate_recur(p) bekerja secara rekursif dari bawah ke atas.

Building an Expression Tree

Konstruktor ExpressionTree dapat digunakan untuk menyusun ekspresi aritmatika secara bertahap, namun untuk membangun pohon ekspresi dari string ekspresi lengkap seperti (((3+1)x4)/((9-5)+2)), kita gunakan algoritma bottom-up berbasis stack. Kita asumsikan ekspresi telah ditokenisasi (misalnya ['(', '(', '3', '+', '1', ')', 'x', '4', ')', '/', '(', '(', '9', '-', '5', ')', '+', '2', ')']), dan telah diparentesis penuh.

Logika Algoritma:

1. Operator (misal +, -, x, /) → dimasukkan ke stack.
2. Nilai literal (misal 3, 9, 2) → dibuatkan pohon tunggal, lalu dimasukkan ke stack.
3. Tanda kurung tutup) → kita ambil tiga elemen terakhir dari stack:
   - Subtree kanan
   - Operator
   - Subtree kiri
   - Kemudian dibuat pohon baru dan dimasukkan kembali ke stack.
4. Tanda kurung buka (→ diabaikan.

Setelah semua token diproses, elemen terakhir pada stack adalah pohon ekspresi lengkap.

Contoh Implementasi Python:

```
1  def build_expression_tree(tokens):
2    """Returns an ExpressionTree based upon by a tokenized expression."""
3    S = []                                    # we use Python list as stack
4    for t in tokens:
5      if t in '+-x*/':                        # t is an operator symbol
6        S.append(t)                           # push the operator symbol
7      elif t not in '()':                     # consider t to be a literal
8        S.append(ExpressionTree(t))           # push trivial tree storing value
9      elif t == ')':          # compose a new tree from three constituent parts
10       right = S.pop( )                      # right subtree as per LIFO
11       op = S.pop( )                         # operator symbol
12       left = S.pop( )                       # left subtree
13       S.append(ExpressionTree(op, left, right)) # repush tree
14     # we ignore a left parenthesis
15   return S.pop( )
```

Code Fragment 8.38: Implementation of a build_expression_tree that produces an ExpressionTree from a sequence of tokens representing an arithmetic expression.

TERIMAKASIH

ATAS PERHATIANNYA