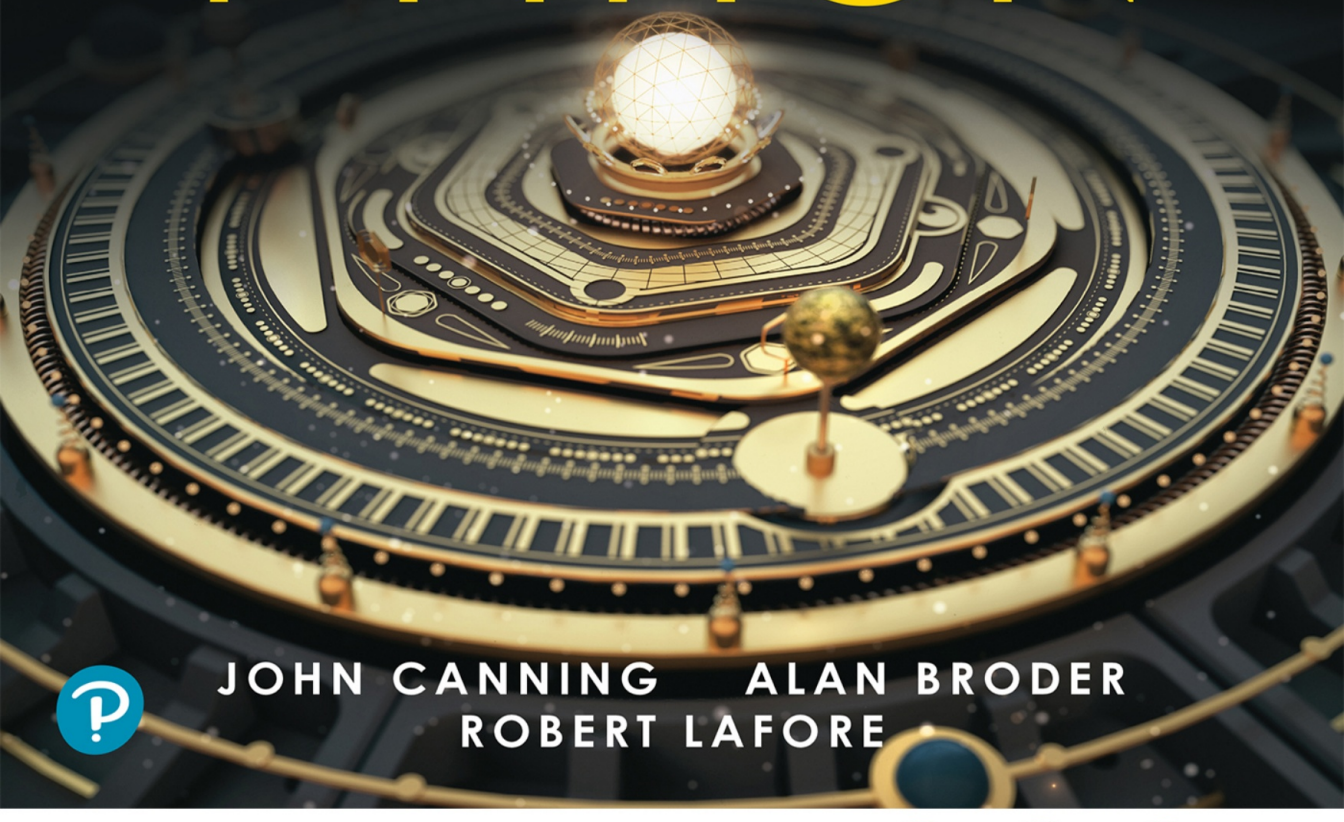
DATA STRUCTURES & ALGORITHMS in PYTHON



JOHN CANNING ALAN BRODER
ROBERT LAFORE

جان کیننگ
ایلن بروڈر
رابرٹ لافور

ڈیٹا ڈھانچے اور ازگر میں الگورتھم

Addison-Wesley

بوسٹن •کولمبس •نیویارک •سان فرانسسکو •ایمسٹرڈیم •کیپ ٹاؤن
دبئی •لندن •میڈرڈ •میلان •میونخ •پیرس •مونٹریال •ٹورنٹو •دہلی •میکسیکو سٹی
ساؤ پالو •سڈنی •ہانگ کانگ •سیول •سنگاپور •تائپے •ٹوکیو

چیف ایڈیٹر
مارک ٹاؤب

ڈائریکٹر، آئی ٹی پی پروڈکٹ
انتظام
بریٹ بارٹو

حصول ایڈیٹر
کم اسپینسلی

ترقیاتی ایڈیٹر
کرس زبن

مینیجنگ ایڈیٹر
سینڈرا شروڈر

پروجیکٹ ایڈیٹر
مینڈی فرینک

کاپی ایڈیٹر
چک ہچنسن

اشاریہ ساز
چیرل لینسر

پروف ریڈر
باربرا میک

ادارتی معاون
سنڈی ٹیٹرز

ڈیزائنر
چوٹی پرسرتستھ

کمپوزیٹر کوڈ منتر

مینوفیکچررز اور بیچنے والے اپنی مصنوعات کو الگ کرنے کے لیے استعمال کیے گئے بہت سے عہدوں کا دعویٰ بطور ٹریڈ مارک کیا جاتا ہے۔ جہاں وہ عہدہ اس کتاب میں نظر آتا ہے، اور پبلشر کو ٹریڈ مارک کے دعوے کا علم تھا، وہ عہدہ ابتدائی بڑے حروف کے ساتھ یا تمام بڑے حروف میں پرنٹ کیا گیا ہے۔

مصنفین اور پبلشر نے اس کتاب کی تیاری میں احتیاط برتی ہے، لیکن کسی قسم کی کوئی ظاہری یا مضمر وارنٹی نہیں دیں گے اور غلطیوں یا کوتاہی کے لیے کوئی ذمہ داری قبول نہیں کریں گے۔ یہاں موجود معلومات یا پروگراموں کے استعمال سے یا اس کے نتیجے میں ہونے والے واقعاتی یا نتیجہ خیز نقصانات کے لیے کوئی ذمہ داری قبول نہیں کی جاتی ہے۔

اس ٹائٹل کو بڑی مقدار میں خریدنے کے بارے میں معلومات کے لیے، یا خاص فروخت کے مواقع کے لیے (جس میں الیکٹرانک ورژن شامل ہو سکتے ہیں؛ حسب ضرورت کور ڈیزائن؛ اور آپ کے کاروبار کے لیے مواد، تربیتی اہداف، مارکیٹنگ کی توجہ، یا برانڈنگ کی دلچسپیاں)، براہ کرم ہمارے کارپوریٹ سے رابطہ کریں۔ corpsales@pearsoned.com یا 382-3419 (800)پر سیلز ڈیپارٹمنٹ۔

سرکاری فروخت سے متعلق پوچھ گچھ کے لیے، براہ کرم governmentsales@pearsoned.comسے رابطہ کریں۔

امریکہ سے باہر فروخت کے بارے میں سوالات کے لیے، براہ کرم intlcs@pearson.comپر رابطہ کریں۔

ہمیں ویب پر دیکھیں: informit.com/awلائبریری آف کانگریس کنٹرول نمبر: Copyright © 2023 Pearson Education, Inc.
2022910068

جملہ حقوق محفوظ ہیں. یہ اشاعت کاپی رائٹ کے ذریعہ محفوظ ہے، اور کسی بھی ممنوعہ تولید، بازیافت کے نظام میں ذخیرہ کرنے، یا کسی بھی شکل میں یا کسی بھی ذریعہ، الیکٹرانک، میکینیکل، تصویری کاپی، ریکارڈنگ، یا اسی طرح سے نشر کرنے سے پہلے ناشر سے اجازت حاصل کی جانی چاہیے۔ Pearson Education Global Rights & Permissions Departmentکے اندر اجازتوں، درخواست فارموں اور مناسب رابطوں کے بارے میں معلومات کے لیے، براہ کرم www.pearson.com/permissionsپر جائیں۔

یہاں موجود معلومات کے استعمال کے سلسلے میں کوئی پیٹنٹ ذمہ داری قبول نہیں کی جاتی ہے۔ اگرچہ اس کتاب کی تیاری میں تمام احتیاط برتی گئی ہے، لیکن ناشر اور مصنف غلطیوں یا کوتاہی کی ذمہ داری قبول نہیں کرتے۔ نہ ہی یہاں موجود معلومات کے استعمال کے نتیجے میں ہونے والے نقصانات کے لیے کوئی ذمہ داری قبول نہیں کی جاتی۔

ISBN-13: 978-0-13-485568-4
ISBN-10: 0-13-485568-X
سکاؤٹ آٹومیٹڈ پرنٹ کوڈ

ایک نظر میں مشمولات

فہرست کا خانہ

Machine Translated by Google

تنوع کے لیے پیئرسن کا عزم، ایکویٹی، اور شمولیت

Pearsonتعصب سے پاک مواد تخلیق کرنے کے لیے وقف ہے جو تمام سیکھنے والوں کے تنوع کی عکاسی کرتا ہے۔
ہم تنوع کی بہت سی جہتوں کو قبول کرتے ہیں، جن میں نسل، نسل، جنس، سماجی اقتصادی حیثیت، قابلیت، عمر، جنسی رجحان، اور مذہبی یا سیاسی عقائد شامل ہیں لیکن ان تک محدود نہیں۔

تعلیم ہماری دنیا میں مساوات اور تبدیلی کے لیے ایک طاقتور قوت ہے۔ اس میں ایسے مواقع فراہم کرنے کی صلاحیت ہے جو زندگیوں کو بہتر بناتے ہیں اور معاشی نقل و حرکت کو قابل بناتے ہیں۔ جیسا کہ ہم ہر پروڈکٹ اور سروس کے لیے مواد تخلیق کرنے کے لیے مصنفین کے ساتھ کام کرتے ہیں، ہم شمولیت کا مظاہرہ کرنے اور متنوع اسکالرشپ کو شامل کرنے کی اپنی ذمہ داری کو تسلیم کرتے ہیں تاکہ ہر کوئی سیکھنے کے ذریعے اپنی صلاحیتوں کو حاصل کر سکے۔ دنیا کی سرکردہ لرننگ کمپنی کے طور پر، ہمارا فرض ہے کہ ہم تبدیلی کو آگے بڑھانے میں مدد کریں اور اپنے مقصد کے مطابق زندگی گزاریں تاکہ زیادہ سے زیادہ لوگوں کو اپنے لیے ایک بہتر زندگی بنانے اور ایک بہتر دنیا بنانے میں مدد ملے۔

ہماری عزائم جان بوجھ کر ایک ایسی دنیا میں تعاون کرنا ہے جہاں

- ہر کسی کے پاس سیکھنے کے ذریعے کامیاب ہونے کا مساوی اور زندگی بھر کا موقع ہے۔
- ہماری تعلیمی مصنوعات اور خدمات جامع ہیں اور بھرپور تنوع کی نمائندگی کرتی ہیں۔
  سیکھنے والوں کی
- ہمارا تعلیمی مواد درست طریقے سے ان سیکھنے والوں کی تاریخوں اور تجربات کی عکاسی کرتا ہے جن کی ہم خدمت کرتے ہیں
- ہمارا تعلیمی مواد سیکھنے والوں کے ساتھ گہری بات چیت کا اشارہ کرتا ہے اور انہیں اپنی تعلیم (اور عالمی نظریہ) کو بڑھانے کے لیے تحریک دیتا ہے۔

جب کہ ہم غیرجانبدارانہ مواد پیش کرنے کے لیے سخت محنت کرتے ہیں، ہم اس پیئرسن پروڈکٹ سے متعلق کسی بھی قسم کے خدشات یا ضروریات کے بارے میں آپ سے سننا چاہتے ہیں تاکہ ہم ان کی چھان بین اور ان کا ازالہ کر سکیں۔

براہ کرم کسی بھی ممکنہ تعصب کے بارے میں خدشات کے ساتھ https://www.pearson.com/report-bias.htmlپر رابطہ کریں ۔

میری ماں کو، جس نے مجھے علم کی پیاس دی،

میرے والد کو، جنہوں نے مجھے انجینئرنگ کی خوشیاں سکھائیں،

اور جون تک، جس نے دونوں کا پیچھا کرنا ممکن بنایا۔

جان کیننگ

میرے والد سول بروڈر کے لیے، کمپیوٹر سائنس کے علمبردار، راہنمائی
کے لیے۔
میری والدہ مارلن بروڈر کو، ماسٹر ایجوکیٹر، مجھے پڑھانے کی
ترغیب دینے کے لیے۔
فران کو، میری زندگی کو مکمل کرنے کے لیے۔

ایلن بروڈر

ڈاؤن لوڈز، اپ ڈیٹس، اور تصحیحات کے دستیاب ہوتے ہی ان تک رسائی حاصل کرنے کے لیے Pythonمیں ڈیٹا اسٹرکچرز اور الگورتھم کی اپنی کاپی informit.comپر رجسٹر کریں۔ رجسٹریشن کا عمل شروع کرنے کے لیے، informit.com/registerپر جائیں اور لاگ ان کریں یا اکاؤنٹ بنائیں۔ پروڈکٹ ISBN 9780134855684درج کریں اور جمع کروائیں پر کلک کریں۔ عمل مکمل ہونے کے بعد، آپ کو "رجسٹرڈ پروڈکٹس" کے تحت کوئی بھی دستیاب بونس مواد مل جائے گا۔

کتاب کے بارے میں دوسرے قارئین کے ساتھ گفتگو میں شامل ہونے کے لیے کتاب کی ویب سائٹ https://datastructures.live ملاحظہ کریں ، اور ان تصورات کے بارے میں مزید جانیں جو ڈیٹا کے ڈھانچے کو زندہ کرتے ہیں۔

اعترافات

جان کیننگ اور ایلن بروڈر سے

اس کتاب کا رابرٹ لافور کا جاوا پر مبنی ورژن کئی سالوں سے دنیا بھر میں ڈیٹا سٹرکچر کورسز اور پیشہ ور افراد کے حوالہ جات میں ایک اہم مقام رہا ہے۔ جب یشیوا یونیورسٹی کے اسٹرن کالج فار ویمن میں ایلن کا ڈیٹا سٹرکچر کورس پائتھون پر چلا گیا تو کورس میں لافور کی کتاب کا استعمال نہ کرنا ایک حقیقی نقصان تھا۔ اس طرح ہم خاص طور پر اس نئے اور نظر ثانی شدہ ایڈیشن کو Python پروگرامرز اور طالب علموں کی دنیا میں لانے میں خوش ہیں۔

ہم سٹرن کے بہت سے طلباء کا شکریہ ادا کرنا چاہتے ہیں جنہوں نے پچھلے کئی سالوں میں براہ راست یا بالواسطہ اس کتاب میں تعاون کیا۔ Laforeکے جاوا کے نفاذ کے ابتدائی Pythonورژن ایلن کے Pythonپر مبنی کورسز میں مرکزی حیثیت رکھتے تھے، اور Sternطالب علم کے تاثرات نے کوڈ کی وضاحت کو بہتر بنانے، اس کی کارکردگی کو بہتر بنانے، اور بعض اوقات کیڑے کی نشاندہی کرنے اور درست کرنے میں بھی مدد کی!

اس نئے ایڈیشن کے ابتدائی مسودوں پر ان کی قیمتی آراء اور سفارشات کے لیے، ہم ایلن کے ڈیٹا سٹرکچر کورسز میں بہت سے طلباء کے شکر گزار ہیں، جن میں ایسٹی بروکس، اڈینا بروس، جولیا چیس، بانا فشر، لیمور کوہنم، ایلیشیوا کوہن، شیرا اورلین، شیرا پہمر شامل ہیں۔ , Jennie Peled, Alexandra Roffe, Avigail Royzenberg, Batia Segal, Penina Waghalter, and Esther Werblowsky. اگر ہم نے کسی کا نام چھوڑ دیا ہے تو معذرت خواہ ہیں۔

اس کتاب کے آپ کے مطالعہ کو بڑھانے کے لیے ڈیٹا سٹرکچر ویژولائزیشن کا ایک اوپن سورس پیکیج دستیاب ہے، اور اسٹرن طلباء نے بصری سازی سافٹ ویئر کی ترقی میں ایک فعال کردار ادا کیا۔ جان اور ایلن اس پراجیکٹ کے اسٹرن اسٹوڈنٹ کے علمبرداروں اور رہنماؤں کا بہت شکریہ ادا کرتے ہیں، بشمول الانا ریڈنسکی، ایلانا اپفیلبام، ایلیانا ٹیٹیلبام، اور للی پولونٹسکی، نیز مندرجہ ذیل ماضی اور حال کے اسٹرن طلباء کے تعاون کرنے والوں اور سرپرستوں: زو ایبودی، آیلیٹ ابرون ، لارا امر، نطانیہ برنبام، اڈینا بروس، چانی ڈوبن، سارہ اینجل، سارہ گراف، ایویگیل ہیلمین، میکل کافمین، سرینا کوفمین، ریچل لیزر، تالیہ لیٹنر، شانی لیوس، رینا میلنکوف، اٹارا نیوگروشل، شیرا پاہمر، میریم، میریم۔ ریپ، شیرا ساسون، شیرا شنائیڈر، مزال شوئن والڈ، شیرا اسمتھ، ریوا ٹراپ، الیگزینڈرا وولچیک، اور ایستھر وربلوسکی۔ اس کے علاوہ، اسٹرن فیکلٹی کا بہت شکریہ جنہوں نے طلباء کے شرکاء کی رہنمائی کی: پروفیسر ایری شماش، پروفیسر لارنس ٹائٹل مین، اور پروفیسر جوشوا ویکسمین۔ اگر ہم نے اس فہرست میں سے کسی کو چھوڑ دیا ہے تو ہم معذرت خواہ ہیں۔

یونیورسٹی آف پنسلوانیا کے پروفیسر ڈیوڈ میٹسزیک کا بہت شکریہ ان کے خیالات اور پاورپوائنٹ سلائیڈز کی ابتدائی شراکت کے لیے جب ایلن نے پہلی بار سٹرن میں ڈیٹا سٹرکچرز پڑھانا شروع کیا۔ انسٹرکٹرز ریسورسز کے سیکشن میں دستیاب بہت سی سلائیڈز کی اصل اس کی واضح اور اچھی طرح سے ڈیزائن کردہ سلائیڈوں میں ہے۔ نیز، ہم یشیوا یونیورسٹی کے شعبہ ریاضی کے پروفیسر ماریان گیڈیا کے شکر گزار ہیں کہ انہوں نے کروی مثلثیات میں بصیرت کی۔

آخر میں، ہم پیئرسن کے باصلاحیت ایڈیٹرز کا بہت بڑا قرض دار ہیں جنہوں نے اس کتاب کو حقیقت بنایا: مارک ٹیبر، کم اسپینسلی، مینڈی فرینک، چوٹی پراسرتستھ، اور کرس زہن۔
ان کی بہت سی صلاحیتوں اور مریض کی مدد کے بغیر، یہ پروجیکٹ ٹیکسٹ فائلوں، ڈرائنگ اور سورس کوڈ کا محض ایک عجیب مجموعہ ہوگا۔

پہلے ایڈیشن کا اعتراف، ڈیٹا کے ڈھانچے اور الگورتھم جاوا

اس مختصر اعتراف میں درج ذیل لوگوں (اور بہت سے دوسرے) کے لیے میرا شکریہ ادا نہیں کیا جا سکتا۔ ہمیشہ کی طرح، مچ ویٹ نے جاوا کی چیز کسی اور سے پہلے سمجھ لی تھی۔ اس نے مجھے ایپلٹس کو اس وقت تک اچھالنے دیا جب تک کہ وہ کام نہ کر لیں، اور قیاس آرائیوں کی بدحالی سے اس منصوبے کی مجموعی شکل نکال لی۔ میرے ایڈیٹر، کرٹ سٹیفن، نے بہت اچھے مبصرین کو تلاش کیا، اس بات کو یقینی بنایا کہ ہر کوئی ایک ہی صفحے پر ہے، گیند کو رول کرتا رہے، اور نرمی سے لیکن مضبوطی سے اس بات کو یقینی بنایا کہ میں نے وہی کیا جو مجھے کرنا تھا۔ ہیری ہینڈرسن نے بہت سے قیمتی تجاویز کے ساتھ پہلے مسودے کا ماہرانہ جائزہ فراہم کیا۔ رچرڈ ایس رائٹ، جونیئر، تکنیکی ایڈیٹر کے طور پر، تفصیل کے لیے اپنی گہری نظر سے متعدد مسائل کو درست کیا۔ ،Jaime Niñoنیو اورلینز یونیورسٹی کے پی ایچ ڈی، نے مجھے اپنے آپ سے بچانے کی کوشش کی اور کبھی کبھار کامیاب ہو گئے، لیکن میرے نقطہ نظر یا کوڈنگ کی تفصیلات کی کوئی ذمہ داری نہیں اٹھانی چاہیے۔ سوسن والٹن اس منصوبے کے جوہر کو غیر تکنیکی تک پہنچانے میں مدد کرنے میں ایک مضبوط اور قابل تعریف حامی رہی ہے۔ کارمیلا کارواجل علمی دنیا کے ساتھ اپنے روابط بڑھانے میں انمول تھی۔ ڈین شیرف نے نہ صرف CD-ROMکو اکٹھا کیا، بلکہ تیزی سے تیار ہونے والی سافٹ ویئر کی تبدیلیوں پر مجھے اپ ٹو ڈیٹ رکھنے میں انتھک محنت کی۔ آخر میں، سیسیل کافمین نے کتاب کو تدوین سے لے کر پیداواری عمل تک منتقلی کے ذریعے اس کی بھرپور طریقے سے مدد کی۔

دوم کو تسلیمات ایڈیشن

میں سامس پبلشنگ میں درج ذیل لوگوں کا شکریہ ادا کرتا ہوں کہ ان کی قابلیت، کوشش اور صبر کے ساتھ اس دوسرے ایڈیشن کی تیاری میں۔ ایکوزیشن ایڈیٹر کیرول ایکرمین اور ڈیولپمنٹ ایڈیٹر سونگلن کیو نے پیچیدہ پیداواری عمل کے ذریعے اس ایڈیشن کی بھرپور رہنمائی کی۔ پروجیکٹ ایڈیٹر Matt Purcellنے نیم لامحدود گرام کی غلطیوں کو درست کیا اور اس بات کو یقینی بنایا کہ ہر چیز صحیح ہے۔ ٹیک ایڈیٹر مائیک کوپک نے پروگراموں کا جائزہ لیا اور مجھے کئی مسائل سے بچایا۔ آخری لیکن کم از کم، ڈین شیرف، پچھلے دور کا ایک پرانا دوست، سامس ویب سائٹ پر میرے کوڈ اور ایپلٹس کا ہنر مند انتظام فراہم کرتا ہے۔

مصنفین کے بارے میں

ڈاکٹر جان کیننگ ایک انجینئر، کمپیوٹر سائنسدان اور محقق ہیں۔ انہوں نے میساچوسٹس انسٹی ٹیوٹ آف ٹیکنالوجی سے الیکٹریکل انجینئرنگ میں ایس بی کی ڈگری حاصل کی اور پی ایچ ڈی کی۔ کالج پارک میں یونیورسٹی آف میری لینڈ سے کمپیوٹر سائنس میں۔ ان کے مختلف پیشوں میں کمپیوٹر سائنس کا پروفیسر، صنعت میں ایک محقق اور سافٹ ویئر انجینئر، اور کمپنی کا نائب صدر شامل ہے۔ اب وہ شکمنت سافٹ ویئر کے صدر ہیں۔

ایلن بروڈر کلینیکل پروفیسر ہیں اور نیو یارک شہر میں یشیوا یونیورسٹی کے اسٹرن کالج برائے خواتین میں کمپیوٹر سائنس کے شعبہ کی چیئر ہیں۔ وہ Pythonپروگرامنگ، ڈیٹا ڈھانچے، اور ڈیٹا سائنس میں تعارفی اور جدید کورسز پڑھاتا ہے۔ سٹرن کالج میں شامل ہونے سے پہلے، وہ ایک سافٹ ویئر انجینئر تھا، بڑے پیمانے پر ڈیٹا کے تجزیہ کے نظام کو ڈیزائن اور بنا رہا تھا۔ اس نے وائٹ اوک ٹیکنالوجیز، انکارپوریٹڈ کو اس کے سی ای او کے طور پر قائم کیا اور اس کی قیادت کی، اور بعد میں فیئر فیکس، ورجینیا میں اس کی جانشین کمپنی، نوویٹا کے چیئرمین اور ساتھی کے طور پر خدمات انجام دیں۔

رابرٹ لافور نے الیکٹریکل انجینئرنگ اور ریاضی میں ڈگریاں حاصل کی ہیں، لارنس برکلے لیبارٹری کے لیے ایک نظام تجزیہ کار کے طور پر کام کیا ہے، اپنی سافٹ ویئر کمپنی کی بنیاد رکھی ہے، اور کمپیوٹر پروگرامنگ کے شعبے میں سب سے زیادہ فروخت ہونے والے مصنف ہیں۔ اس کے کچھ عنوانات C++میں آبجیکٹ اورینٹڈ پروگرامنگ اور جاوا میں ڈیٹا سٹرکچرز اور الگورتھم ہیں۔

تعارف

اس کتاب میں کیا ہے؟ یہ کتاب ان طلباء کے لیے ڈیٹا ڈھانچے اور الگورتھم کے عملی تعارف کے لیے بنائی گئی ہے جنہوں نے ابھی کمپیوٹر پروگرام لکھنا شروع کیا ہے۔ یہ تعارفی عبارت آپ کو کتاب کے بارے میں مزید بتائے گی، یہ کیسے ترتیب دی گئی ہے، ہم امید کرتے ہیں کہ کتاب شروع کرنے سے پہلے قارئین کو کیا تجربہ حاصل ہوگا، اور اسے پڑھنے اور مشقیں کرنے سے آپ کو کیا علم حاصل ہوگا۔

یہ کتاب کس کے لیے ہے۔

ڈیٹا ڈھانچے اور الگورتھم کمپیوٹر سائنس کا مرکز ہیں۔ اگر آپ نے کبھی یہ سمجھنا چاہا ہے کہ کمپیوٹر کیا کر سکتے ہیں، وہ کیسے کرتے ہیں، اور وہ کیا نہیں کر سکتے، تو آپ کو دونوں کے بارے میں گہری سمجھ کی ضرورت ہے (شاید یہ کہنا بہتر ہے کہ "کمپیوٹر کو کیا کرنے میں دشواری ہوتی ہے" کے بجائے وہ نہیں کر سکتے)۔ اس کتاب کو ڈیٹا سٹرکچرز اور/یا الگورتھم کورس میں بطور متن استعمال کیا جا سکتا ہے، جو کثرت سے یونیورسٹی کے کمپیوٹر سائنس کے نصاب کے دوسرے سال میں پڑھایا جاتا ہے۔ تاہم، متن پیشہ ور پروگرامرز، ہائی اسکول کے طلباء، اور کسی اور کے لیے بھی ڈیزائن کیا گیا ہے جسے محض پروگرامنگ زبان جاننے سے اگلا قدم اٹھانے کی ضرورت ہے۔ چونکہ یہ سمجھنا آسان ہے، اس لیے یہ زیادہ رسمی کورس کے ضمنی متن کے طور پر بھی مناسب ہے۔ یہ مثالوں، مشقوں، اور اضافی مواد سے بھری ہوئی ہے، لہذا اسے کلاس روم کی ترتیب سے باہر خود مطالعہ کے لیے استعمال کیا جا سکتا ہے۔

اس کتاب کو لکھنے میں ہمارا نقطہ نظر یہ ہے کہ قارئین کے لیے یہ سمجھنا آسان ہو کہ ڈیٹا ڈھانچے کیسے کام کرتے ہیں اور انہیں عملی طور پر کیسے لاگو کیا جائے۔ یہ کچھ دوسرے متن سے مختلف ہے جو ریاضی کے نظریہ پر زور دیتے ہیں، یا ان ڈھانچے کو کسی خاص زبان یا سافٹ ویئر لائبریری میں کیسے لاگو کیا جاتا ہے۔ ہم نے حقیقی دنیا کے ایپلی کیشنز کے ساتھ مثالوں کا انتخاب کیا ہے اور صرف ریاضی یا غیر واضح مثالوں کے استعمال سے گریز کیا ہے۔ ہم اہم خیالات کو بات چیت کرنے میں مدد کے لیے اعداد و شمار اور تصوراتی پروگرام استعمال کرتے ہیں۔ ہم اب بھی الگورتھم کی پیچیدگی کا احاطہ کرتے ہیں اور یہ بتانے کے لیے ریاضی کی ضرورت ہے کہ کس طرح پیچیدگی کارکردگی کو متاثر کرتی ہے۔

اس کتاب کو پڑھنے سے پہلے آپ کو کیا جاننے کی ضرورت ہے۔

اس کتاب کو استعمال کرنے کے لیے ضروری شرائط ہیں: کچھ پروگرامنگ زبان اور کچھ ریاضی کا علم۔ اگرچہ نمونہ کا کوڈ Pythonمیں لکھا گیا ہے، لیکن آپ کو Pythonکو جاننے کی ضرورت نہیں ہے کہ کیا ہو رہا ہے۔ اگر آپ نے کچھ طریقہ کار اور/یا آبجیکٹ پر مبنی پروگرامنگ کی ہے تو ازگر کو سمجھنا مشکل نہیں ہے۔ ہم نے مثالوں میں نحو کو ہر ممکن حد تک عام رکھا ہے،

مزید خاص طور پر، ہم Pythonورژن 3نحو استعمال کرتے ہیں۔ یہ ورژن Python 2سے کچھ مختلف ہے، لیکن بہت زیادہ نہیں۔ Pythonایک بھرپور زبان ہے جس میں بہت سے بلٹ ان ڈیٹا کی اقسام اور لائبریری ہیں جو اس کی صلاحیتوں کو بڑھاتی ہیں۔ تاہم، ہماری مثالیں دو وجوہات کی بنا پر زیادہ بنیادی تعمیرات کا استعمال کرتی ہیں: یہ دوسری زبانوں سے واقف پروگرامرز کے لیے ان کو سمجھنا آسان بناتا ہے، اور یہ ڈیٹا ڈھانچے کی تفصیلات کو زیادہ واضح طور پر واضح کرتا ہے۔ بعد کے ابواب میں، ہم Pythonکی کچھ خصوصیات کا استعمال کرتے ہیں جو دوسری زبانوں میں نہیں ملتی ہیں جیسے جنریٹر اور فہرست کی تفہیم۔ ہم وضاحت کرتے ہیں کہ یہ کیا ہیں اور ان سے پروگرامر کو کیسے فائدہ ہوتا ہے۔

یقیناً، اگر آپ ازگر (ورژن 2یا 3سے پہلے ہی واقف ہیں تو اس سے مدد ملے گی۔ شاید آپ نے Pythonکے بہت سے ڈیٹا ڈھانچے میں سے کچھ استعمال کیے ہوں گے اور آپ کو اس بارے میں تجسس ہے کہ ان کو کیسے لاگو کیا جاتا ہے۔ ہم باب 1میں Pythonنحو کا جائزہ لیتے ہیں، "جائزہ" ان لوگوں کے لیے جنہیں تعارف یا ریفریشر کی ضرورت ہے۔ اگر آپ نے جاوا، C++، C#، JavaScriptیا پرل جیسی زبانوں میں پروگرام کیا ہے، تو بہت سی تعمیرات سے واقف ہونا چاہیے۔ اگر آپ نے صرف فنکشنل یا ڈومین کے لیے مخصوص زبانوں کا استعمال کرتے ہوئے پروگرام کیا ہے، تو آپ کو ازگر کے بنیادی عناصر سے واقف ہونے کے لیے زیادہ وقت گزارنا پڑ سکتا ہے۔ اس متن کے علاوہ، نوآموز Pythonپروگرامرز کے لیے بہت سے وسائل دستیاب ہیں، بشمول انٹرنیٹ پر بہت سے ٹیوٹوریلز۔

پروگرامنگ زبان کے علاوہ، ہر پروگرامر کو کیا جاننا چاہیے؟ ریاضی سے الجبرا کے ذریعے ریاضی کا ایک اچھا علم ضروری ہے۔ کمپیوٹر پروگرامنگ علامتی ہیرا پھیری ہے۔ بالکل الجبرا کی طرح، اصطلاحات کو دوبارہ ترتیب دینے، انہیں مختلف شکلوں میں رکھنے، اور کچھ حصوں کو زیادہ نمایاں کرنے کے لیے اظہار کو تبدیل کرنے کے طریقے ہیں، یہ سب کچھ ایک ہی معنی کو برقرار رکھتے ہوئے ہے۔ ریاضی میں استفادہ کو سمجھنا بھی ضروری ہے۔ کمپیوٹر سائنس کا زیادہ تر یہ جاننے پر مبنی ہے کہ ایک نمبر کو دوسرے نمبر کی طاقت تک بڑھانے کا کیا مطلب ہے۔ ریاضی کے علاوہ، تنظیم کا ایک اچھا احساس بھی تمام پروگرامنگ کے لیے فائدہ مند ہے۔ یہ جاننا کہ آئٹمز کو مختلف طریقوں سے کیسے ترتیب دیا جائے (وقت کے لحاظ سے، فنکشن کے لحاظ سے، سائز کے لحاظ سے، پیچیدگی کے لحاظ سے، وغیرہ) پروگراموں کو موثر اور برقرار رکھنے کے لیے بہت ضروری ہے۔ جب ہم کارکردگی اور برقرار رکھنے کے بارے میں بات کرتے ہیں، تو کمپیوٹر سائنس میں ان کے خاص معنی ہوتے ہیں۔ کارکردگی زیادہ تر اس بات پر ہوتی ہے کہ چیزوں کی گنتی میں کتنا وقت لگتا ہے لیکن اس میں لگنے والی جگہ کی مقدار بھی ہو سکتی ہے۔ مینٹینیبلٹی سے مراد دوسرے پروگرامرز کے ساتھ ساتھ آپ کے پروگراموں کو سمجھنے اور ان میں ترمیم کرنے میں آسانی ہے۔

آپ کو انٹرنیٹ پر چیزوں کو تلاش کرنے، سافٹ ویئر ڈاؤن لوڈ اور انسٹال کرنے اور انہیں کمپیوٹر پر چلانے کے بارے میں بھی علم کی ضرورت ہوگی۔ ویژولائزیشن پروگراموں کو ڈاؤن لوڈ کرنے اور چلانے کی ہدایات اس کتاب کے ضمیمہ Aمیں دیکھی جا سکتی ہیں۔ انٹرنیٹ نے پروگرامنگ اور کمپیوٹر سائنس سیکھنے کے ٹولز سمیت ٹولز کے کارنوکوپیا تک رسائی کو بہت آسان بنا دیا ہے۔ ہم توقع کرتے ہیں کہ قارئین پہلے ہی جان لیں گے کہ مفید وسائل کیسے تلاش کیے جائیں اور ایسے ذرائع سے بچیں جو نقصان دہ سافٹ ویئر فراہم کر سکتے ہیں۔

آپ اس کتاب سے کیا سیکھ سکتے ہیں۔

جیسا کہ آپ اس کے عنوان سے توقع کر سکتے ہیں، یہ کتاب آپ کو یہ سکھا سکتی ہے کہ ڈیٹا سٹرکچر کس طرح پروگراموں (اور پروگرامرز) کو اپنے کام میں زیادہ موثر بناتے ہیں۔ آپ جان سکتے ہیں کہ کس طرح ڈیٹا آرگنائزیشن اور مناسب الگورتھم کے ساتھ اس کا جوڑا بہت زیادہ اثر انداز ہوتا ہے کہ کمپیوٹنگ وسائل کی دی گئی مقدار کے ساتھ کیا شمار کیا جا سکتا ہے۔ یہ کتاب آپ کو اعداد و شمار کے ڈھانچے کو لاگو کرنے کے طریقے کے بارے میں مکمل طور پر وضاحت دے سکتی ہے، اور یہ آپ کو کسی بھی پروگرامنگ زبان میں ان کو نافذ کرنے کے قابل بنائے گی۔ آپ یہ فیصلہ کرنے کا عمل سیکھ سکتے ہیں کہ کسی خاص پروگرامنگ کی درخواست کو پورا کرنے کے لیے کون سا ڈیٹا ڈھانچہ اور الگورتھم سب سے زیادہ موزوں ہیں۔ شاید سب سے اہم بات، آپ یہ جان سکتے ہیں کہ ایک الگورتھم اور/یا ڈیٹا کا ڈھانچہ کسی مخصوص استعمال کے معاملے میں کب ناکام ہو جائے گا۔ ڈیٹا ڈھانچے اور الگورتھم کو سمجھنا کمپیوٹر سائنس کا بنیادی حصہ ہے، جو کہ ایک Python(یا دوسری زبان) پروگرامر ہونے سے مختلف ہے۔

کتاب بنیادی ڈیٹا ڈھانچے کو سکھاتی ہے جو ہر پروگرامر کو جاننا چاہئے۔
قارئین کو سمجھ لینا چاہیے کہ اور بھی بہت ہیں۔ ڈیٹا کے یہ بنیادی ڈھانچے مختلف قسم کے حالات میں کام کرتے ہیں۔ اس کتاب میں آپ جو مہارتیں تیار کرتے ہیں، آپ کو قابل ہونا چاہیے۔

کسی دوسرے ڈیٹا ڈھانچے یا الگورتھم کی تفصیل کو پڑھنے کے لیے اور تجزیہ کرنا شروع کریں کہ آیا یہ اس سے بہتر کارکردگی کا مظاہرہ کرے گا یا اس سے بدتر کارکردگی کا مظاہرہ کرے گا جو آپ نے پہلے سے ہی مخصوص استعمال کے معاملات میں سیکھا ہے۔

یہ کتاب Pythonکے کچھ نحو اور ساخت کی وضاحت کرتی ہے، لیکن یہ آپ کو اس کی تمام صلاحیتیں نہیں سکھائے گی۔ کتاب میں Pythonکی مکمل صلاحیتوں کے ذیلی سیٹ کا استعمال کیا گیا ہے تاکہ یہ واضح کیا جا سکے کہ ڈیٹا کے مزید پیچیدہ ڈھانچے کس طرح آسان تعمیرات سے بنائے جاتے ہیں۔ یہ کسی ایسے شخص کو پروگرامنگ کی بنیادی باتیں سکھانے کے لیے نہیں بنایا گیا ہے جس نے کبھی پروگرام نہیں کیا ہے۔ ازگر ایک بہت ہی اعلیٰ سطحی زبان ہے جس میں بہت سے بلٹ ان ڈیٹا ڈھانچے ہیں۔ کچھ زیادہ ابتدائی اقسام کا استعمال کرنا جیسے کہ انٹیجرز کی صفوں یا ریکارڈ ڈھانچے کا استعمال کرنا، جیسا کہ آپ کو Cیا ++Cمیں مل سکتا ہے، ازگر میں کچھ زیادہ مشکل ہے۔ چونکہ کتاب کا فوکس ڈیٹا ڈھانچے کا نفاذ اور تجزیہ ہے، اس لیے ہماری مثالیں ان قدیم قسموں کے لیے قربت کا استعمال کرتی ہیں۔

کچھ Pythonپروگرامرز معیاری لائبریریوں میں زبان کے ساتھ فراہم کردہ مزید خوبصورت تعمیرات کے بارے میں جانتے ہوئے، یہ مثالیں غیر ضروری طور پر پیچیدہ محسوس کر سکتے ہیں۔ اگر آپ کمپیوٹر سائنس، اور خاص طور پر الگورتھم کی پیچیدگی کو سمجھنا چاہتے ہیں، تو آپ کو پرائمیٹوز پر بنیادی کارروائیوں کو سمجھنا چاہیے۔ جب آپ کسی پروگرامنگ لینگویج میں فراہم کردہ ڈیٹا سٹرکچر یا اس کے کسی ایڈ آن ماڈیول سے استعمال کرتے ہیں، تو آپ کو اکثر یہ جاننے کے لیے اس کی پیچیدگی کو جاننا پڑے گا کہ آیا یہ آپ کے استعمال کے معاملے میں اچھی طرح سے کام کرے گا۔ ڈیٹا کے بنیادی ڈھانچے، ان کی پیچیدگیوں، اور تجارتی معاہدوں کو سمجھنے سے آپ کو ان کے اوپر بنائے گئے ڈھانچے کو سمجھنے میں مدد ملے گی۔

تمام ڈیٹا ڈھانچے کو آبجیکٹ اورینٹڈ پروگرامنگ (OOP)کا استعمال کرتے ہوئے تیار کیا گیا ہے۔ اگر یہ آپ کے لیے ایک نیا تصور ہے تو، باب 1کا جائزہ اس بات کا کہ Pythonمیں کلاسز کی تعریف اور استعمال کیسے کیا جاتا ہے OOPکا بنیادی تعارف فراہم کرتا ہے۔ آپ کو اس متن سے OOPکی مکمل طاقت اور فوائد سیکھنے کی امید نہیں رکھنی چاہیے۔ اس کے بجائے، آپ ہر ڈیٹا ڈھانچے کو بطور کلاس لاگو کرنا سیکھیں گے۔ یہ کلاسز OOPمیں اشیاء کی قسمیں ہیں اور یہ سافٹ ویئر تیار کرنا آسان بناتی ہیں جنہیں بہت سے مختلف ایپلی کیشنز قابل اعتماد طریقے سے دوبارہ استعمال کر سکتے ہیں۔

کتاب میں بہت سی مثالیں استعمال کی گئی ہیں، لیکن یہ کمپیوٹر سائنس کے کسی خاص اطلاقی علاقے جیسے ڈیٹا بیس، یوزر انٹرفیس، یا مصنوعی ذہانت کے بارے میں کتاب نہیں ہے۔ مثالوں کا انتخاب پروگراموں کی عام ایپلی کیشنز کو واضح کرنے کے لیے کیا جاتا ہے، لیکن تمام پروگرام ایک خاص سیاق و سباق میں لکھے جاتے ہیں، اور یہ وقت کے ساتھ ساتھ تبدیل ہوتے رہتے ہیں۔ 1970میں لکھا گیا ڈیٹا بیس پروگرام اس وقت بہت ترقی یافتہ دکھائی دے سکتا ہے، لیکن آج یہ بہت معمولی لگتا ہے۔ اس متن میں پیش کردہ امتحانات کو یہ سکھانے کے لیے ڈیزائن کیا گیا ہے کہ ڈیٹا ڈھانچے کو کس طرح لاگو کیا جاتا ہے، وہ کیسے کارکردگی کا مظاہرہ کرتے ہیں، اور نیا پروگرام ڈیزائن کرتے وقت ان کا موازنہ کیسے کیا جاتا ہے۔ امتحانات کو ہر ڈیٹا ڈھانچے کے سب سے زیادہ جامع یا بہترین نفاذ کے طور پر نہیں لیا جانا چاہیے، اور نہ ہی ان تمام ممکنہ ڈیٹا ڈھانچے کے مکمل جائزہ کے طور پر لیا جانا چاہیے جو کسی مخصوص درخواست کے علاقے کے لیے موزوں ہو سکتے ہیں۔

ساخت

ہر باب ڈیٹا ڈھانچے اور متعلقہ الگورتھم کا ایک خاص گروپ پیش کرتا ہے۔
ابواب کے اختتام پر، ہم باب کے اہم نکات اور بعض اوقات پچھلے ابواب کے ساتھ تعلقات کا احاطہ کرنے والے جائزہ سوالات فراہم کرتے ہیں۔ ان کے جوابات ضمیمہ سی، "سوالات کے جوابات" میں مل سکتے ہیں۔ ان سوالات کا مقصد قارئین کے لیے خود امتحان ہے، تاکہ یہ یقینی بنایا جا سکے کہ آپ تمام مواد کو سمجھ گئے ہیں۔

بہت سے ابواب قارئین کو آزمانے کے لیے تجربات تجویز کرتے ہیں۔ یہ انفرادی سوچ کے تجربات، ٹیم سائنمنٹس، یا کتاب کے ساتھ فراہم کردہ سافٹ ویئر ٹولز کے ساتھ مشقیں ہو سکتی ہیں۔ یہ ابھی سیکھے گئے علم کو کسی اور شعبے میں لاگو کرنے اور آپ کی سمجھ کو گہرا کرنے میں مدد کرنے کے لیے بنائے گئے ہیں۔

پروگرامنگ کے منصوبے طویل، زیادہ چیلنجنگ پروگرامنگ مشقیں ہیں۔ ہم مشکل کی مختلف سطحوں کے پروجیکٹس فراہم کرتے ہیں۔ ان منصوبوں کو کلاس روم کی ترتیبات میں ہوم ورک اسائنمنٹ کے طور پر استعمال کیا جا سکتا ہے۔ پروگرامنگ پروجیکٹس کے نمونے حل پبلشر اور ویب سائٹ https://datastructures.liveسے اہل اساتذہ کے لیے دستیاب ہیں۔

تاریخ مچل ویٹ اور رابرٹ لافور نے اس کتاب کا پہلا ورژن تیار کیا اور اسے جاوا میں ڈیٹا سٹرکچرز اور الگورتھم کا عنوان دیا۔ پہلا ایڈیشن 1998میں شائع ہوا تھا، اور دوسرا ایڈیشن، رابرٹ کا، 2002میں سامنے آیا تھا۔ جان کیننگ اور ایلن بروڈر نے اس ورژن کو ایجوکیشن اور تجارتی اور غیر تجارتی سوفٹ ویئر کی ترقی میں مقبولیت کی وجہ سے پائتھون کا استعمال کرتے ہوئے تیار کیا۔ جاوا بڑے پیمانے پر استعمال کیا جاتا ہے اور کمپیوٹر سائنس دانوں کے لیے جاننے کے لیے ایک اہم زبان ہے۔ بہت سے اسکولوں نے Pythonکو پہلی پروگرامنگ لینگویج کے طور پر اپنانے کے ساتھ، نصابی کتب کی ضرورت جو پہلے سے مانوس زبان میں نئے تصورات متعارف کراتے ہیں، اس کتاب کی ترقی کا باعث بنے۔ ہم نے ڈیٹا ڈھانچے کی کوریج کو بڑھایا اور بہت سی مثالوں کو اپ ڈیٹ کیا۔

ہم نے سیکھنے کے عمل کو ہر ممکن حد تک تکلیف دہ بنانے کی کوشش کی ہے۔ ہم امید کرتے ہیں کہ یہ متن بنیادی، اور واضح طور پر، کمپیوٹر سائنس کی خوبصورتی کو سب کے لیے قابل رسائی بنائے گا۔ کھڑے ہونے کے علاوہ، ہم امید کرتے ہیں کہ آپ کو ان خیالات کو سیکھنے میں مزہ آئے گا۔ اپنے آپ سے لطف اندوز!

فگر کریڈٹس

کریڈٹ/انتساب	اعداد و شمار
شکمنت سافٹ ویئر	اعداد و شمار ،5.4 ،5.3 ،4.11 ،4.10 ،4.6 ،4.2 ،3.9 ،3.7 ،3.5 ،2.6 ،2.4–2.1 ،5.9 ،6.10 ،6.18 ،7.3 ،7.4 ،7.6 ،7.10 ،7.11 ،8.5 ،8.8 ،8.9 ،8.11 ،8.14 ،8.15 ،9.7–9.9 ،10.09-10.11(ac) ،10.14–10.16 ،10.29(ab) ،11.06–11.10 ،11.14 ،11.15 ،11.18 ،11.19 ،12.24 ،13.7 ،13.8 ،13.9 ،13.13 ،14.6 ،14.12 ،14.17 ،15.02 ،15.9 ،15.14–15.18 A.01-A.04
بشکریہ Droste BV	شکل 6.1
www.wordclouds.comکے ذریعہ تیار کردہ لفظ کلاؤڈ۔	شکل 13.15
آرکی /13شٹر اسٹاک	ڈھانپنا

باب 8

بائنری درخت

اس باب میں

اس باب کو ہم الگورتھم سے تبدیل کرتے ہیں، باب 7 کا فوکس، "اعلی درجے کی ترتیب،" ڈیٹا ڈھانچے پر۔ بائنری درخت پروگرامنگ میں استعمال ہونے والے بنیادی ڈیٹا اسٹوریج ڈھانچے میں سے ایک ہیں۔ وہ ایسے فوائد فراہم کرتے ہیں جو آپ نے اب تک دیکھے ہوئے ڈیٹا ڈھانچے نہیں کر سکتے۔ اس باب میں آپ سیکھتے ہیں کہ آپ درختوں کو کیوں استعمال کرنا چاہیں گے، وہ کیسے کام کرتے ہیں، اور انہیں کیسے بنانا ہے۔

بائنری درخت کیوں استعمال کریں؟

آپ درخت کیوں استعمال کرنا چاہتے ہیں؟ عام طور پر، کیونکہ یہ com دو دیگر ڈھانچے کے فوائد کو جوڑتا ہے: ایک ترتیب شدہ صف اور ایک منسلک فہرست۔ آپ ایک درخت کو تیزی سے تلاش کر سکتے ہیں، جیسا کہ آپ آرڈر کی گئی صف کر سکتے ہیں، اور آپ آئٹمز کو جلدی سے داخل اور حذف بھی کر سکتے ہیں، جیسا کہ آپ لنک کردہ فہرست کے ساتھ کر سکتے ہیں۔ آئیے درختوں کی تفصیلات میں جانے سے پہلے ان موضوعات کو تھوڑا سا دریافت کرتے ہیں۔

ایک ترتیب شدہ صف میں آہستہ اندراج ایک ایسی صف کا تصور کریں جس میں تمام عناصر کو ترتیب سے ترتیب دیا گیا ہو —یعنی ایک ترتیب شدہ صف —جیسا کہ آپ نے باب 2 میں دیکھا تھا۔ جیسا کہ آپ نے سیکھا، آپ بائنری تلاش کا استعمال کرتے ہوئے کسی خاص قدر کے لیے اس طرح کی صف کو تیزی سے تلاش کر سکتے ہیں۔

آپ صف کے مرکز میں چیک کریں؛ اگر آپ جس چیز کی تلاش کر رہے ہیں وہ اس سے زیادہ ہے جو آپ کو وہاں ملتی ہے، تو آپ اپنی تلاش کو صف کے اوپری نصف تک محدود کر دیتے ہیں۔ اگر یہ کم ہے تو، آپ اپنی تلاش کو نیچے کے نصف تک محدود کر دیتے ہیں۔ اس عمل کو بار بار لگانے سے O(log N) وقت میں آبجیکٹ مل جاتا ہے۔ آپ ترتیب شدہ ترتیب میں ہر آبجیکٹ کا دورہ کرتے ہوئے، ترتیب دی گئی صف کو بھی تیزی سے عبور کر سکتے ہیں۔

دوسری طرف، اگر آپ کسی نئے آبجیکٹ کو ترتیب شدہ صف میں داخل کرنا چاہتے ہیں، تو آپ کو پہلے یہ معلوم کرنا ہوگا کہ آبجیکٹ کہاں جائے گا اور پھر اس کے لیے جگہ بنانے کے لیے تمام اشیاء کو زیادہ کلیدوں کے ساتھ صف میں ایک جگہ پر منتقل کریں۔ یہ متعدد

چالیں وقت طلب ہیں، اوسطاً، نصف اشیاء کو منتقل کرنے کی ضرورت ہوتی ہے N/2)حرکتیں)۔
حذف کرنے میں ایک ہی متعدد حرکتیں شامل ہیں اور اس طرح اتنی ہی سست ہے۔

اگر آپ بہت زیادہ اندراجات اور حذف کرنے جا رہے ہیں، تو آرڈر شدہ صف ایک برا انتخاب ہے۔

لنکڈ لسٹ میں سست تلاش جیسا کہ آپ نے باب 5میں دیکھا، "لنکڈ لسٹ"، آپ لنکڈ لسٹ پر تیزی سے اندراج اور حذف کر سکتے ہیں۔
آپ صرف چند حوالوں کو تبدیل کرکے ان کارروائیوں کو پورا کرسکتے ہیں۔

ان دو آپریشنز کے لیے O(1)وقت (تیز ترین بگ Oٹائم) درکار ہوتا ہے۔

بدقسمتی سے، منسلک فہرست میں ایک مخصوص عنصر تلاش کرنا اتنا تیز نہیں ہے۔ آپ کو فہرست کے آغاز سے شروع کرنا چاہیے اور ہر
ایک عنصر کو اس وقت تک ملاحظہ کرنا چاہیے جب تک کہ آپ کو وہ چیز نہ مل جائے جس کی آپ تلاش کر رہے ہیں۔ اس طرح، آپ کو اوسطاً
N/2اشیاء کا دورہ کرنے کی ضرورت ہے، ہر ایک کی کلید کا مطلوبہ قدر کے ساتھ موازنہ کرنا۔ یہ عمل سست ہے، O(N)وقت درکار ہے۔ (نوٹ
کریں کہ ایک ترتیب کے لیے تیز سمجھے جانے والے اوقات داخل کرنے، حذف کرنے اور تلاش کے بنیادی ڈیٹا ڈھانچے کے کاموں کے لیے سست
ہیں۔)

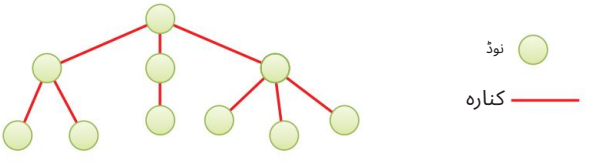
آپ سوچ سکتے ہیں کہ آپ آرڈر شدہ لنکڈ لسٹ کا استعمال کرکے چیزوں کو تیز کرسکتے ہیں، جس میں عناصر کو ترتیب سے ترتیب دیا گیا
ہے، لیکن اس سے کوئی فائدہ نہیں ہوتا ہے۔ آپ کو اب بھی شروع سے شروع کرنا چاہیے اور عناصر کو ترتیب سے دیکھنا چاہیے کیونکہ کسی
بھی عنصر کے حوالہ جات کی پیروی کیے بغیر اس تک رسائی کا کوئی طریقہ نہیں ہے۔ آپ ترتیب شدہ ترتیب میں ایک خلا کو تلاش کرنے کے
بعد عنصر کی تلاش کو ترک کر سکتے ہیں جہاں اسے ہونا چاہیے تھا، لہذا اس سے گمشدہ اشیاء کی شناخت میں تھوڑا وقت بچ جائے گا۔ آرڈر
شدہ فہرست کا استعمال صرف نوڈس کو تیز تر ترتیب دینے میں مدد کرتا ہے اور کسی صوابدیدی چیز کو تلاش کرنے میں مدد نہیں کرتا ہے۔

بچاؤ کے لیے درخت

یہ اچھا ہو گا کہ اگر کسی لنک شدہ فہرست کے فوری اندراج اور حذف کرنے کے ساتھ ایک ترتیب شدہ صف کی فوری تلاش کے ساتھ ڈیٹا کا
ڈھانچہ موجود ہو۔ درخت ان دونوں خصوصیات کو فراہم کرتے ہیں اور ڈیٹا کے سب سے دلچسپ ڈھانچے میں سے ایک ہیں۔

درخت کیا ہے؟

ایک درخت کناروں سے جڑے ہوئے نوڈس پر مشتمل ہوتا ہے۔ شکل 1-8ایک درخت کو دکھاتی ہے۔ درخت کی ایسی تصویر میں نوڈس کو
دائروں کے طور پر اور کناروں کو دائروں کو جوڑنے والی لکیروں کے طور پر دکھایا جاتا ہے۔



شکل 1-8ایک عام (غیر بائنری) درخت

درختوں کا بڑے پیمانے پر تجریدی ریاضیاتی اداروں کے طور پر مطالعہ کیا گیا ہے، لہذا ان کے بارے میں نظریاتی علم کی ایک بڑی مقدار موجود ہے۔ ایک درخت درحقیقت ایک زیادہ عمومی زمرہ کی مثال ہے جسے گراف کہتے ہیں۔ نوڈس کو جوڑنے والے کناروں کی اقسام اور ترتیب درختوں اور گرافوں میں فرق کرتے ہیں، لیکن آپ کو گرافس کے اضافی مسائل کے بارے میں فکر کرنے کی ضرورت نہیں ہے۔ ہم باب 14، "گرافس،" اور باب 15، "وزن والے گرافس" میں گرافس پر بحث کرتے ہیں۔

کمپیوٹر پروگراموں میں، نوڈس اکثر اداروں کی نمائندگی کرتے ہیں جیسے فائل فولڈرز، فائلز، ڈیپارٹمنٹس، لوگ وغیرہ۔ دوسرے لفظوں میں، کسی بھی قسم کے ڈیٹا ڈھانچے میں محفوظ کردہ عام ریکارڈ اور آئٹمز۔ آبجیکٹ اورینٹڈ پروگرامنگ لینگویج میں، نوڈس ایسی اشیاء ہوتی ہیں جو کبھی کبھی حقیقی دنیا میں ہستیوں کی نمائندگی کرتی ہیں۔

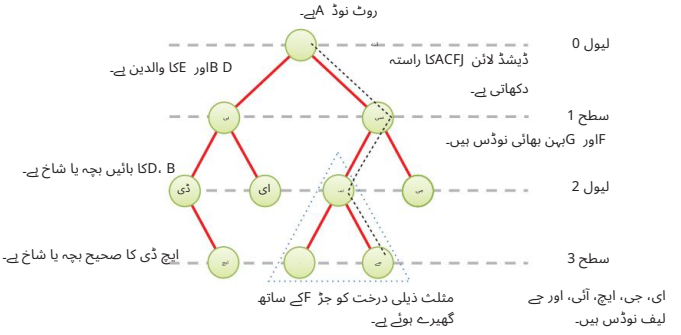
نوڈس کے درمیان لکیریں (کنارے) نوڈس کے متعلق ہونے کے طریقے کی نمائندگی کرتی ہیں۔ موٹے طور پر، لکیریں سہولت کی نمائندگی کرتی ہیں: ایک پروگرام کے لیے ایک نوڈ سے دوسرے نوڈ تک جانا آسان (اور تیز) ہے اگر کوئی لائن ان کو جوڑتی ہے۔ درحقیقت، نوڈ سے نوڈ تک جانے کا واحد راستہ لائنوں کے ساتھ ایک راستے پر چلنا ہے۔ یہ بنیادی طور پر وہی ہیں جو آپ نے منسلک فہرستوں میں دیکھے تھے۔ ہر نوڈ میں دوسرے نوڈس کے کچھ حوالہ جات ہو سکتے ہیں۔ الگورتھم کناروں کے ساتھ ایک سمت جانے تک محدود ہیں: نوڈ سے کسی دوسرے نوڈ کے حوالے سے۔ دوہرے منسلک نوڈس کا استعمال بعض اوقات دونوں سمتوں میں جانے کے لیے کیا جاتا ہے۔

عام طور پر، ایک نوڈ کو درخت کی جڑ کے طور پر نامزد کیا جاتا ہے۔ بالکل لنکڈ لسٹ کے ہیڈ کی طرح، باقی تمام نوڈس جڑ سے کناروں کی پیروی کرتے ہوئے پہنچ جاتے ہیں۔ روٹ نوڈ ڈائیگرام کے اوپری حصے میں ٹائپی کالی تیار کیا گیا ہے، جیسا کہ شکل 8-1 میں ہے۔ دوسرے نوڈس اس کے نیچے دکھائے گئے ہیں، اور خاکہ میں جتنا نیچے، دوسرے نوڈ تک جانے کے لیے اتنے ہی کناروں کو فالو کرنے کی ضرورت ہے۔ اس طرح، درختوں کے خاکے اوپر سے چھوٹے اور نیچے بڑے ہوتے ہیں۔ یہ ترتیب حقیقی درختوں کے مقابلے میں الٹا لگ سکتی ہے، کم از کم زمین کے اوپر حقیقی درختوں کے حصوں کے مقابلے میں۔ خاکے بصری معنوں میں درختوں کے جڑ کے نظام کی طرح ہیں۔ یہ ترتیب انہیں چارٹس کی طرح بناتا ہے جو خاندانی درختوں کو اوپر والے آباؤ اجداد کے ساتھ اور نیچے کی اولاد کو دکھانے کے لیے استعمال کیا جاتا ہے۔ عام طور پر، پروگرام درخت کے چھوٹے حصے، جڑ سے آپریشن شروع کرتے ہیں، اور کناروں کو وسیع تر کنارے تک لے جاتے ہیں۔ اوپر سے نیچے تک جانے کے بارے میں سوچنا زیادہ فطری ہے، جیسا کہ متن کو پڑھتے ہیں، اس لیے جڑ کے نیچے دوسرے نوڈس رکھنے سے نوڈس کی نسبتی ترتیب کو ظاہر کرنے میں مدد ملتی ہے۔

درختوں کی مختلف قسمیں ہیں، کناروں کی تعداد اور قسم سے ممتاز ہیں۔ شکل 8-1 میں دکھائے گئے درخت میں فی نوڈ دو سے زیادہ بچے ہیں۔ (ہم ایک لمحے میں بتاتے ہیں کہ "بچوں" کا کیا مطلب ہے۔) اس باب میں ہم درخت کی ایک مخصوص شکل پر بحث کرتے ہیں جسے بائنری ٹری کہتے ہیں۔ بائنری درخت میں ہر نوڈ میں زیادہ سے زیادہ دو بچے ہوتے ہیں۔ زیادہ عام درخت، جن میں نوڈس میں دو سے زیادہ بچے ہو سکتے ہیں، ملٹی وے ٹری کہلاتے ہیں۔ ہم باب 4-3-2، "درخت اور بیرونی ذخیرہ" 9، میں ملٹی وے درختوں کی مثالیں دکھاتے ہیں۔

درخت کی اصطلاحات

درختوں کے مخصوص پہلوؤں کو بیان کرنے کے لیے بہت سی اصطلاحات استعمال کی جاتی ہیں۔ آپ کو ان کو جاننے کی ضرورت ہے تاکہ یہ بحث قابل فہم ہو۔ خوش قسمتی سے، ان میں سے زیادہ تر اصطلاحات حقیقی دنیا کے درختوں یا خاندانی تعلقات سے متعلق ہیں، اس لیے انہیں یاد رکھنا مشکل نہیں ہے۔ شکل 8-2 ان میں سے بہت سی اصطلاحات کو دکھاتا ہے جو بائنری درخت پر لاگو ہوتے ہیں۔

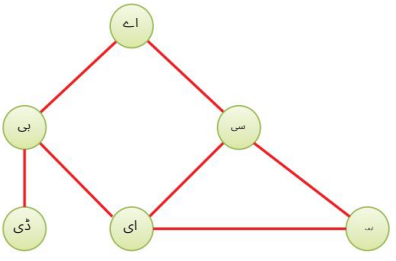


شکل 8-2درخت کی اصطلاحات

جڑ درخت کے اوپری حصے میں موجود نوڈ کو جڑ کہا جاتا ہے۔ ایک درخت میں صرف ایک جڑ ہے، جس پر تصویر میں Aکا لیبل لگا ہوا ہے۔

راستہ کسی ایسے شخص کے بارے میں سوچیں جو ان کو جوڑتے ہوئے کناروں کے ساتھ نوڈ سے نوڈ تک چلتے ہیں۔ نوڈس کے نتیجے میں ترتیب کو راستہ کہا جاتا ہے۔ نوڈس اور کناروں کے مجموعے کو درخت کے طور پر بیان کرنے کے لیے، جڑ سے کسی دوسرے نوڈ تک ایک (اور صرف ایک!) راستہ ہونا چاہیے۔

شکل 8-3ایک غیر درخت کو ظاہر کرتی ہے۔ آپ دیکھ سکتے ہیں کہ یہ اس اصول کی خلاف ورزی کرتا ہے کیونکہ Aسے نوڈس Eاور Fتک متعدد راستے ہیں۔ یہ گراف کی ایک مثال ہے جو درخت نہیں ہے۔



تصویر 8-3ایک غیر درخت

والدین

کسی بھی نوڈ (جڑ کے علاوہ) کا بالکل ایک کنارہ ہوتا ہے جو اسے اوپر والے نوڈ سے جوڑتا ہے۔ اس کے اوپر والے نوڈ کو نوڈ کا پیرنٹ کہا جاتا ہے۔ روٹ نوڈ کا پیرنٹ نہیں ہونا چاہیے۔

بچہ

کسی بھی نوڈ میں ایک یا زیادہ کنارے ہو سکتے ہیں جو اسے نیچے نوڈس سے جوڑتے ہیں۔ دیئے گئے نوڈ کے نیچے ان نوڈس کو اس کے بچے، یا بعض اوقات شاخیں کہتے ہیں۔

Siblingروٹ نوڈ کے علاوہ کسی بھی نوڈ میں بہن بھائی نوڈس ہو سکتے ہیں۔ ان نوڈس میں مشترکہ پیرنٹ نوڈ ہوتا ہے۔

لیف ایک نوڈ جس کی کوئی اولاد نہیں ہوتی اسے لیف نوڈ یا محض ایک پتی کہا جاتا ہے۔ ایک درخت میں صرف ایک جڑ ہوسکتی ہے، لیکن بہت سے پتے ہوسکتے ہیں. اس کے برعکس، ایک نوڈ جس میں بچے ہوتے ہیں ایک اندرونی نوڈ ہوتا ہے۔

سب ٹری کسی بھی نوڈ (جڑ کے علاوہ) کو ذیلی درخت کی جڑ سمجھا جا سکتا ہے ، جس میں اس کے بچے، اور اس کے بچوں کے بچے وغیرہ شامل ہیں۔ اگر آپ خاندانی جھوٹ کے لحاظ سے سوچتے ہیں تو، نوڈ کے ذیلی درخت میں اس کی تمام اولادیں ہوتی ہیں۔

نوڈ کا دورہ اس وقت کیا جاتا ہے جب پروگرام کنٹرول نوڈ پر آتا ہے، عام طور پر نوڈ پر کچھ آپریشن کرنے کے مقصد کے لیے، جیسے کہ اس کے ڈیٹا فیلڈ میں سے کسی کی قدر کی جانچ کرنا یا اسے ڈسپلے کرنا۔ ایک نوڈ سے دوسرے نوڈ کے راستے پر محض ایک نوڈ کے اوپر سے گزرنا نوڈ پر جانا نہیں سمجھا جاتا ہے۔

ایک درخت کو عبور کرنے کا مطلب ہے تمام نوڈس کو کچھ مخصوص ترتیب میں جانا۔ مثال کے طور پر، آپ چڑھتے ہوئے کلیدی قدر کی ترتیب میں تمام نوڈس ملاحظہ کر سکتے ہیں۔ درخت کو عبور کرنے کے اور بھی طریقے ہیں، جیسا کہ ہم بعد میں بیان کریں گے۔

سطحیں کسی خاص نوڈ کی سطح سے مراد یہ ہے کہ نوڈ جڑ سے کتنی نسلوں پر مشتمل ہے۔

اگر آپ فرض کریں کہ جڑ لیول 0ہے، تو اس کے بچے لیول 1پر ہیں، اس کے پوتے پوتیاں لیول 2پر ہیں، وغیرہ۔ اسے بعض اوقات نوڈ کی گہرائی بھی کہا جاتا ہے۔

کیز آپ نے دیکھا ہے کہ کسی چیز میں ایک ڈیٹا فیلڈ کو عام طور پر کلیدی قدر کے طور پر نامزد کیا جاتا ہے، یا ایک کلید کو سم پلائی کیا جاتا ہے۔ یہ قدر آئٹم کو تلاش کرنے یا اس پر دیگر کارروائیاں کرنے کے لیے استعمال ہوتی ہے۔ درختوں کے خاکوں میں، جب ایک دائرہ ڈیٹا آئٹم کے حامل نوڈ کی نمائندگی کرتا ہے، تو آئٹم کی کلیدی قدر عام طور پر دائرے میں دکھائی جاتی ہے۔

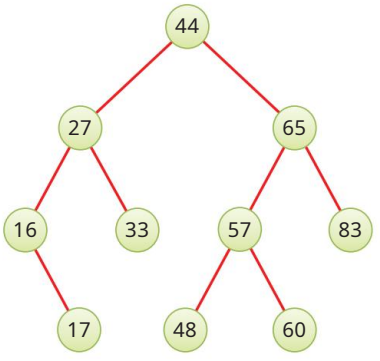
Binary Treesاگر درخت کے ہر نوڈ میں زیادہ سے زیادہ دو بچے ہوں تو درخت کو بائنری ٹری کہا جاتا ہے۔ اس باب میں ہم بائنری درختوں پر توجہ مرکوز کرتے ہیں کیونکہ وہ سب سے آسان اور عام ہیں۔

بائنری ٹری میں ہر نوڈ کے دو بچوں کو لیفٹ چائلڈ اور رائٹ چائلڈ کہا جاتا ہے، ان کی پوزیشنوں کے مطابق جب آپ درخت کی تصویر کھینچتے ہیں، جیسا کہ شکل 2-8میں دکھایا گیا ہے۔ بائنری ٹری میں ایک نوڈ ضروری نہیں کہ زیادہ سے زیادہ دو ہوں۔

بچے؛ اس کا صرف بائیں بچہ ہو سکتا ہے یا صرف دائیں بچہ ہو سکتا ہے، یا اس کا کوئی بچہ نہیں ہو سکتا (اس صورت میں یہ ایک پتی ہے)۔

بائنری سرچ ٹری جس قسم کے بائنری ٹری پر ہم اس باب کے شروع میں بحث کر رہے ہیں اسے تکنیکی طور پر بائنری سرچ ٹری کہا جاتا ہے۔ نوڈس کی چابیاں تلاش کے درختوں میں ایک خاص ترتیب رکھتی ہیں۔

شکل 8-4ایک بائنری تلاش کا درخت دکھاتا ہے۔



تصویر 8-4ایک بائنری تلاش کا درخت

نوٹ

بائنری سرچ ٹری کی وضاحتی خصوصیت یہ ہے: نوڈ کے بائیں بچے کی کلید اس کے والدین کی کلید سے کم ہونی چاہیے، اور نوڈ کے دائیں بچے کے پاس اس کے والدین کی کلید سے زیادہ یا اس کے برابر کلید ہونی چاہیے۔

ایک تشبیہ

ایک عام درخت کا سامنا ڈیسک ٹاپ کمپیوٹرز پر درجہ بندی کا فائل سسٹم ہے۔
یہ سسٹم بیسویں صدی میں کاروباری اداروں کے ذریعہ استعمال ہونے والی دستاویزی ذخیرہ کرنے کی مروجہ ٹیکنالوجی پر وضع کیا گیا تھا: فائلنگ کیبنٹ جس میں فولڈرز ہوتے ہیں جن میں ذیلی فولڈر ہوتے ہیں، انفرادی دستاویزات تک۔ کمپیوٹر آپریٹنگ سسٹم فائلوں کو درجہ بندی میں محفوظ کرکے اس کی نقل کرتے ہیں۔ درجہ بندی کے اوپری حصے میں روٹ ڈائرکٹری ہے۔ اس ڈائرکٹری میں "فولڈرز" شامل ہیں جو کہ ذیلی ڈائرکٹریز ہیں، اور فائلیں، جو کاغذی دستاویزات کی طرح ہیں۔ ہر ذیلی ڈائرکٹری میں اس کی اپنی اور مزید فائلوں کی سب ڈائرکٹریاں ہوسکتی ہیں۔ ان سب کی درخت میں مشابہتیں ہیں: روٹ ڈائرکٹری روٹ نوڈ ہے، سب ڈائرکٹریاں بچوں کے ساتھ نوڈز ہیں، اور فائلیں لیف نوڈ ہیں۔

فائل سسٹم میں کسی خاص فائل کی وضاحت کرنے کے لیے، آپ روٹ ڈائرکٹری سے فائل تک مکمل راستہ استعمال کرتے ہیں۔ یہ درخت کے نوڈ کے راستے کے برابر ہے۔ یونیفارم ریسورس لوکا ٹورز (URLs)انٹرنیٹ پر کسی وسائل کا راستہ دکھانے کے لیے اسی طرح کی تعمیر کا استعمال کرتے ہیں۔ فائل سسٹم کے پاتھ نام اور یو آر ایل دونوں ذیلی ڈائرکٹریز کی کئی سطحوں کی اجازت دیتے ہیں۔ فائل سسٹم پاتھ میں آخری نام یا تو سب ڈائرکٹری یا فائل ہے۔ فائلیں پتوں کی نمائندگی کرتی ہیں۔ ان کی اپنی کوئی اولاد نہیں ہے۔

واضح طور پر، ایک درجہ بندی کا فائل سسٹم بائنری ٹری نہیں ہے کیونکہ ایک ڈائریکٹری میں بہت سے بچے ہوسکتے ہیں۔ ایک درجہ بندی کا فائل سسٹم ان درختوں سے ایک اور اہم طریقے سے مختلف ہے جن پر ہم یہاں بحث کرتے ہیں۔ فائل سسٹم میں، سب ڈائرکٹریاں ان کے نام جیسی صفات کے علاوہ کوئی ڈیٹا نہیں رکھتی ہیں۔ ان میں صرف دیگر ذیلی ڈائریکٹریوں یا فائلوں کے حوالے ہوتے ہیں۔ صرف فائلوں میں ڈیٹا ہوتا ہے۔

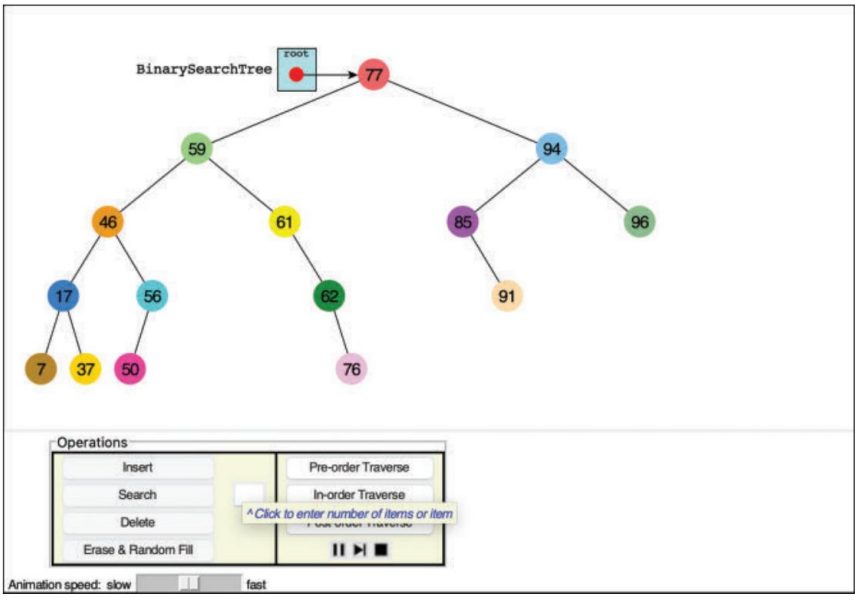
ایک درخت میں، ہر نوڈ میں ڈیٹا ہوتا ہے۔ ڈیٹا کی صحیح قسم کا انحصار اس بات پر ہوتا ہے کہ کیا بھیجا جا رہا ہے: اہلکاروں کے بارے میں ریکارڈ، گاڑی کی تعمیر میں استعمال ہونے والے اجزاء کے بارے میں ریکارڈ، وغیرہ۔ اعداد و شمار کے علاوہ، پتیوں کے علاوہ تمام نوڈس دیگر نوڈس کے حوالے پر مشتمل ہوتے ہیں۔

درجہ بندی کے فائل سسٹم دوسرے پہلوؤں میں بھی بائنری سرچ ٹری سے مختلف ہیں۔ فائل سسٹم کا مقصد فائلوں کو منظم کرنا ہے۔ بائنری سرچ ٹری کا مقصد زیادہ عمومی اور خلاصہ ہے۔ یہ ایک ڈیٹا ڈھانچہ ہے جو آئٹمز کے مجموعے پر داخل کرنے، حذف کرنے، تلاش کرنے، اور ٹراورسل کی عام کارروائیاں فراہم کرتا ہے، ان کو اپنی کلیدوں کے ذریعے منظم کرکے کارروائیوں کو تیز کرتا ہے۔ دونوں کے درمیان مشابہت کا مقصد ایک اور مانوس نظام کو ظاہر کرنا ہے جو کچھ اہم خصوصیات کا اشتراک کرتا ہے، لیکن تمام نہیں۔

بائنری تلاش کے درخت کیسے کام کرتے ہیں؟

آئیے دیکھتے ہیں کہ دی گئی کلید کے ساتھ نوڈ تلاش کرنے، نیا نوڈ ڈالنے، درخت کو عبور کرنے اور نوڈ کو حذف کرنے کے عام بائنری ٹری آپریشنز کو کیسے انجام دیا جائے۔ ان میں سے ہر ایک آپریشن کے لیے، ہم سب سے پہلے یہ دکھاتے ہیں کہ اسے انجام دینے کے لیے بائنری سرچ ٹری ویژولائزیشن ٹول کا استعمال کیسے کیا جائے؛ پھر ہم متعلقہ Pythonکوڈ کو دیکھتے ہیں۔

بائنری سرچ ٹری ویژولائزیشن ٹول اس مثال کے لیے، بائنری سرچ ٹری ویژولائزیشن ٹول شروع کریں (پروگرام کو BinaryTree.pyکہا جاتا ہے)۔ آپ کو ایک ایسی سکرین نظر آنی چاہیے جو کہ شکل 5-8میں دکھائی گئی ہے۔



تصویر 5-8بائنری سرچ ٹری ویژولائزیشن ٹول

ویژولائزیشن ٹول کا استعمال نوڈس میں دکھائی جانے والی کلیدی اقدار 0سے 99تک ہوتی ہیں۔ بلاشبہ، ایک حقیقی درخت میں، کلیدی اقدار کی ایک بڑی رینج ہو سکتی ہے۔ مثال کے طور پر، اگر ٹیلی فون نمبرز کو کلیدی اقدار کے لیے استعمال کیا جاتا ہے، تو وہ 999,999,999,999,999تک ہو سکتے ہیں 15)ہندسے بشمول بین الاقوامی ٹیلی کمیونیکیشن یونین کے معیار میں ملک کے کوڈز)۔ ہم ممکنہ چابیاں کے ایک آسان سیٹ پر توجہ مرکوز کرتے ہیں۔

ویژولائزیشن ٹول اور حقیقی درخت کے درمیان ایک اور فرق یہ ہے کہ ٹول اپنے درخت کو پانچ کی گہرائی تک محدود کرتا ہے۔ یعنی، جڑ سے نیچے تک پانچ سے زیادہ سطحیں نہیں ہو سکتیں (لیول 0سے لیول -(4یہ پابندی یقینی بناتی ہے کہ درخت میں موجود تمام نوڈس اسکرین پر نظر آئیں گے۔ ایک حقیقی درخت میں سطحوں کی تعداد لامحدود ہوتی ہے (جب تک کہ کام پیوٹر میموری ختم نہ ہوجائے)۔

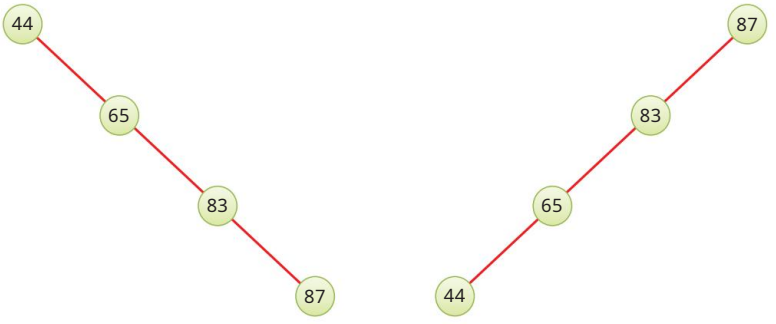
ویژولائزیشن ٹول کا استعمال کرتے ہوئے، آپ جب چاہیں ایک نیا درخت بنا سکتے ہیں۔ ایسا کرنے کے لیے، متعدد آئٹمز درج کریں اور Fill Ease & Randomبٹن پر کلک کریں۔ آپ 0سے 99آئٹمز بھرنے کے لیے کہہ سکتے ہیں۔ اگر آپ 0کا انتخاب کرتے ہیں، تو آپ ایک خالی درخت بنائیں گے۔ بڑی تعداد کے استعمال سے مزید نوڈس بھر جائیں گے، لیکن کچھ درخواست کردہ نوڈس ظاہر نہیں ہو سکتے ہیں۔ یہ درخت کی گہرائی کی حد اور آئٹمز کو داخل کیے جانے کے بے ترتیب ترتیب کی وجہ سے ہے۔ آپ بے ترتیب ترتیب سے نکلنے والے درختوں کی مختلف قسموں کو دیکھنے کے لیے نوڈس کی مختلف تعداد کے ساتھ درخت بنا کر تجربہ کر سکتے ہیں۔

نوڈس مختلف رنگوں کے ساتھ بنائے گئے ہیں۔ رنگ کلید کے ساتھ ذخیرہ شدہ ڈیٹا کی نمائندگی کرتا ہے۔ ہم تھوڑی دیر بعد دکھاتے ہیں کہ کچھ آپریشنز میں اس ڈیٹا کو کیسے اپ ڈیٹ کیا جاتا ہے۔

درختوں کی تعمیر جیسا کہ ویژولائزیشن ٹول میں دکھایا گیا ہے، درخت کی شکل دونوں چیزوں پر منحصر ہوتی ہے جو اس میں شامل ہیں اور ساتھ ہی ان اشیاء کو درخت میں کس ترتیب سے داخل کیا جاتا ہے۔ یہ پہلے تو عجیب لگ سکتا ہے۔ اگر آئٹمز کو ترتیب شدہ صف میں داخل کیا جاتا ہے، تو وہ ہمیشہ ایک ہی ترتیب میں ختم ہوتے ہیں، قطع نظر ان کی ترتیب سے۔ بائنری تلاش کے درخت مختلف کیوں ہیں؟

بائنری سرچ ٹری کی ایک اہم خصوصیت یہ ہے کہ اس میں آئٹمز کو داخل ہونے کے ساتھ ہی انہیں مکمل طور پر آرڈر کرنے کی ضرورت نہیں ہے۔ جب یہ کسی موجودہ درخت میں کوئی نئی چیز شامل کرتا ہے، تو یہ فیصلہ کرتا ہے کہ نئے لیف نوڈ کو کہاں رکھنا ہے اس کی کلید کا درخت میں پہلے سے ذخیرہ شدہ نوڈس سے موازنہ کر کے۔ یہ جڑ سے ایک گمشدہ بچے تک جانے والے راستے کی پیروی کرتا ہے جہاں نیا نوڈ "تعلق رکھتا ہے۔" بائیں چائلڈ کو منتخب کرنے سے جب نئے نوڈ کی کلید اندرونی نوڈ کی کلید سے کم ہو اور دوسری اقدار کے لیے صحیح چائلڈ، ہمیشہ نئے نوڈ کے لیے ایک منفرد راستہ ہوگا۔ اس منفرد راستے کا مطلب ہے کہ آپ اس نوڈ کو بعد میں اس کی کلید کے ذریعے آسانی سے تلاش کر سکتے ہیں، لیکن اس کا مطلب یہ بھی ہے کہ پہلے داخل کردہ آئٹمز کسی بھی نئی آئٹم کے راستے کو متاثر کرتی ہیں۔

مثال کے طور پر، اگر آپ خالی بائنری سرچ ٹری سے شروع کرتے ہیں اور کلیدی ترتیب میں اضافہ کرتے ہوئے نوڈس داخل کرتے ہیں، تو ہر ایک کے لیے منفرد راستہ ہمیشہ صحیح ترین راستہ ہوگا۔ ہر اندراج نیچے دائیں جانب ایک اور نوڈ کا اضافہ کرتا ہے۔ اگر آپ نوڈس کی ترتیب کو ریورس کرتے ہیں اور انہیں خالی درخت میں داخل کرتے ہیں، تو ہر اندراج نیچے بائیں طرف نوڈ کو جوڑ دے گا کیونکہ کلید اب تک درخت میں کسی دوسرے سے کم ہے۔ شکل 6-8دکھاتا ہے کہ اگر آپ ،83 ،65 44اور 87کلیدوں کے ساتھ نوڈس کو آگے یا ریورس ترتیب میں داخل کرتے ہیں تو کیا ہوتا ہے۔

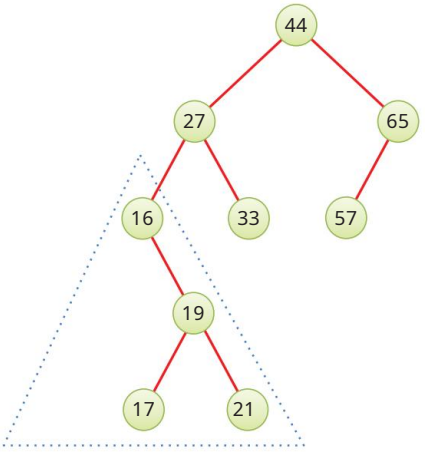


شکل 8-6درخت ترتیب دیے گئے نوڈس ڈال کر بنائے گئے ہیں۔

غیر متوازن درخت وہ درخت جو شکل 8-6میں دکھائے گئے ہیں، درختوں کی طرح نظر نہیں آتے۔ درحقیقت، وہ لنکڈ فہرستوں کی طرح نظر آتے ہیں۔

بائنری سرچ ٹری کے اہداف میں سے ایک خاص نوڈ کی تلاش کو تیز کرنا ہے، لہذا نوڈ کو تلاش کرنے کے لیے منسلک فہرست کے ذریعے قدم اٹھانا کوئی بہتری نہیں ہوگی۔ درخت کا فائدہ حاصل کرنے کے لیے، جڑ کے دونوں طرف نوڈس کو تقریباً متوازن ہونا چاہیے۔ مثالی طور پر، نوڈ کو تلاش کرنے کے راستے پر ہر قدم کو تلاش کرنے کے لیے نوڈس کی تعداد کو آدھے حصے میں کاٹنا چاہیے، بالکل اسی طرح جیسے صفوں کی بائنری تلاشوں اور باب 2میں بیان کردہ guess-a-numberگیم میں۔

ہم کچھ درختوں کو غیر متوازن کہہ سکتے ہیں ۔ یعنی، ان کے زیادہ تر نوڈس جڑ کے ایک طرف یا دوسری طرف ہوتے ہیں، جیسا کہ شکل 8-7 میں دکھایا گیا ہے۔ کوئی بھی ذیلی درخت بھی غیر متوازن ہو سکتا ہے جیسا کہ اعداد و شمار کے نچلے بائیں طرف بیان کردہ۔ بلاشبہ، صرف ایک مکمل متوازن درخت کے بائیں اور دائیں ذیلی درختوں پر مساوی تعداد میں نوڈس ہوں گے (اور مکمل طور پر متوازن ہونے کی وجہ سے، ہر نوڈ کا ذیلی درخت بھی مکمل طور پر متوازن ہوگا)۔ تصادفی طور پر منتخب کردہ آئٹمز پر ایک وقت میں ایک نوڈس داخل کرنے کا مطلب ہے کہ زیادہ تر درخت ایک یا زیادہ نوڈس سے غیر متوازن ہوں گے کیونکہ وہ تعمیر ہوتے ہیں، لہذا آپ عام طور پر مکمل طور پر متوازن درخت تلاش کرنے کی توقع نہیں کر سکتے۔ مندرجہ ذیل ابواب میں، ہم ان میں توازن پیدا کرنے کے طریقوں کو زیادہ احتیاط سے دیکھتے ہیں کیونکہ نوڈس ڈالے اور حذف کیے جاتے ہیں۔



تصویر 8-7ایک غیر متوازن درخت (ایک غیر متوازن ذیلی درخت کے ساتھ)

درخت اس ترتیب کی وجہ سے غیر متوازن ہو جاتے ہیں جس میں ڈیٹا آئٹمز ڈالے جاتے ہیں۔ اگر ان کلیدی اقدار کو تصادفی طور پر داخل کیا جائے تو درخت کم و بیش متوازن ہو جائے گا۔ جب ایک صعودی ترتیب (جیسے 11، 18، 33، 42، 65) یا نزولی ترتیب کا سامنا ہوتا ہے، تو تمام اقدار دائیں بچے (اگر چڑھتے ہیں) یا بائیں بچے (اگر اترتے ہیں) ہوں گے، اور درخت غیر متوازن ہوگا۔ ویژولائزیشن ٹول میں کلیدی قدریں تصادفی طور پر پیدا ہوتی ہیں، لیکن یقیناً کچھ مختصر صعودی یا نزولی ترتیب بہرحال تخلیق کی جائے گی، جو مقامی عدم توازن کا باعث بنے گی۔

اگر ایک درخت ڈیٹا آئٹمز کے ذریعہ بنایا گیا ہے جس کی کلیدی قدریں بے ترتیب ترتیب میں آتی ہیں، تو غیر متوازن درختوں کا مسئلہ بڑے درختوں کے لیے بہت زیادہ مسئلہ نہیں ہوسکتا ہے کیونکہ ایک ترتیب میں نمبروں کے طویل چلنے کے امکانات کم ہیں۔ بعض اوقات، تاہم، کلیدی قدریں سخت ترتیب میں آئیں گی۔ مثال کے طور پر، جب کوئی ڈیٹا انٹری کرنے والا ڈیٹا داخل کرنے سے پہلے فارموں کے اسٹیک کو حروف تہجی کے حساب سے ترتیب دیتا ہے۔ جب ایسا ہوتا ہے تو، درخت کی کارکردگی کو سنجیدگی سے گرایا جا سکتا ہے۔ ہم ابواب 9اور 10میں غیر متوازن درختوں اور ان کے بارے میں کیا کرنے کے بارے میں بات کرتے ہیں۔

پائتھون کوڈ میں درخت کی نمائندگی آئیے ازگر میں بائنری سرچ ٹری کو لاگو کرنا شروع کریں۔ دوسرے ڈیٹا ڈھانچے کی طرح، کمپیوٹر کی میموری میں درخت کی نمائندگی کرنے کے کئی طریقے ہیں۔ سب سے عام یہ ہے کہ نوڈس کو میموری میں (غیر متعلقہ) جگہوں پر اسٹور کیا جائے اور ہر نوڈ میں حوالہ جات کا استعمال کرتے ہوئے ان کو جوڑ دیا جائے جو اس کے بچوں کی طرف اشارہ کرتے ہیں۔

آپ میموری میں ایک درخت کی بھی ایک صف کے طور پر نمائندگی کر سکتے ہیں، مخصوص پوزیشنوں میں نوڈس کے ساتھ صف میں متعلقہ پوزیشنوں میں محفوظ ہیں۔ ہم اس باب کے آخر میں اس امکان کی طرف لوٹتے ہیں۔ ہمارے نمونے کے Pythonکوڈ کے لیے ہم حوالہ جات کا استعمال کرتے ہوئے نوڈس کو جوڑنے کا طریقہ استعمال کریں گے، جیسا کہ باب 5میں منسلک فہرستوں کو لاگو کیا گیا تھا۔

BinarySearchTreeکلاس ہمیں ٹری آبجیکٹ کے لیے ایک کلاس کی ضرورت ہے: وہ آبجیکٹ جو تمام نوڈس کو رکھتا ہے، یا کم از کم اس کی طرف لے جاتا ہے۔ ہم اس کلاس کو BinarySearchTreeکہیں گے۔ اس میں صرف ایک فیلڈ ہے، __root، جو روٹ نوڈ کا حوالہ رکھتا ہے، جیسا کہ فہرست 1-8میں دکھایا گیا ہے۔ یہ لنکڈ لسٹ کلاس سے بہت ملتا جلتا ہے جو باب 5میں منسلک فہرستوں کی نمائندگی کے لیے استعمال کیا گیا تھا۔ BinarySearchTreeکلاس کو دوسرے نوڈس کے لیے فیلڈز کی ضرورت نہیں ہے کیونکہ وہ سبھی روٹ نوڈ سے دوسرے حوالوں کی پیروی کرتے ہوئے حاصل کیے جاتے ہیں ۔

فہرست سازی BinarySearchTree 1-8کلاس کے لیے کنسٹرکٹر

```
کلاس بائنری سرچ ٹری (آبجیکٹ):  #ایک بائنری سرچ ٹری کلاس

    #درخت اپنی  #چابیاں کے ذریعہ نوڈس کو منظم کرتا ہے۔ ابتدائی طور
                                                    پر، یہ خالی ہے۔
    def __init__(self):
        self.__root = کوئی نہیں۔
```

کنسٹرکٹر روٹ نوڈ کے حوالے کو شروع کرتا ہے جیسے خالی درخت سے شروع کرنے کے لیے کوئی نہیں ۔ جب پہلا نوڈ داخل کیا جاتا ہے، تو __rootاس کی طرف اشارہ کرے گا جیسا کہ شکل 5-8کے ویژولائزیشن ٹول کی مثال میں دکھایا گیا ہے۔ یقیناً بہت سے طریقے ہیں جو BinarySearchTreeآبجیکٹ پر کام کرتے ہیں، لیکن پہلے، آپ کو ان کے اندر موجود نوڈس کی وضاحت کرنے کی ضرورت ہے۔

__نوڈ کلاس درخت کے نوڈس میں ذخیرہ کی جانے والی اشیاء کی نمائندگی کرنے والا ڈیٹا ہوتا ہے (مثال کے طور پر ایڈریس بک میں رابطے کی معلومات)، ان اشیاء کی شناخت کے لیے ایک کلید (اور انہیں ترتیب دینے کے لیے)، اور ہر نوڈ کے دو کے حوالے بچے. ہمیں یاد دلانے کے لیے کہ BinarySearchTreeآبجیکٹ بنانے والے کالرز کو براہ راست نوڈس کو تبدیل کرنے کی اجازت نہیں ہونی چاہیے، ہم اس کلاس کے اندر ایک نجی Node__کلاس بناتے ہیں۔ 8-2کی فہرست بتاتی ہے کہ BinarySearchTreeکلاس کے اندر اندرونی کلاس کی وضاحت کیسے کی جا سکتی ہے -

فہرست سازی 8-2 __Node اور BinarySearchTreeکلاسز کے کنسٹرکٹرز

```
کلاس بائنری سرچ ٹری (آبجیکٹ):  #ایک بائنری سرچ ٹری کلاس
...

کلاس __نوڈ(آبجیکٹ): self.key = key              #بائنری سرچ ٹری میں ایک نوڈ
data, left=None, right=None):                    #کنسٹرکٹر ایک کلید لیتا ہے جو  #تلاش کے درخت کے اندر پوزیشن
def __init__(self, key,                          کا تعین کرنے کے لیے استعمال کیا جاتا ہے،  #کلید سے وابستہ ڈیٹا  #
                                                  اور بائیں اور دائیں چائلڈ نوڈ

                                                  #اگر معلوم ہو۔
                                                  #مثال کے طور پر پیرامیٹرز کاپی کریں  #آبجیکٹ کی
self.data = ڈیٹا                                  خصوصیات
self.leftChild = بائیں
self.rightChild = حق

def__rts__(خود):                                 #نوڈ کو سٹرنگ ریٹرن کے طور پر پیش کریں "}" + str(self.data) +
                                                  " {" + str(self.key) + ","

def __init__(self):                              #درخت اپنی  #چابیاں کے ذریعہ نوڈس کو منظم کرتا ہے۔ ابتدائی طور
self.__root = کوئی نہیں۔                          پر، یہ خالی ہے.

def isEmpty(self):                                #چیک کریں کہ آیا درخت خالی ہے۔
خود کو واپس کریں ۔ __جڑ کوئی نہیں ہے۔

def root(self): if self.isEmpty():                #روٹ نوڈ کا ڈیٹا اور کلید حاصل کریں۔
                                                  #اگر درخت خالی ہے تو استثنیٰ بڑھائیں استثنیٰ ("خالی درخت میں کوئی جڑ نوڈ
                                                  نہیں")
واپسی (                                           #بصورت دیگر روٹ ڈیٹا اور اس کی کلید واپس کریں۔
self.__root.data, self.__root.key)
```

Node__آبجیکٹ BinarySearchTreeکے طریقوں سے بنائے جاتے ہیں اور جوڑ توڑ کرتے ہیں ۔ Node__کے اندر فیلڈز کو عوامی اوصاف کے طور پر شروع کیا جا سکتا ہے کیونکہ BinarySearchTreeکے طریقے اس بات کا خیال رکھتے ہیں کہ کبھی بھی Node__آبجیکٹ کو واپس نہ کریں۔ یہ اعلامیہ ()getKeyیا ()setDataجیسے رسائی کے طریقے بنائے بغیر براہ راست پڑھنے اور لکھنے کی اجازت دیتا ہے ۔ __نوڈ کنسٹرکٹر صرف فراہم کردہ دلائل سے فیلڈز کو آباد کرتا ہے۔ اگر چائلڈ نوڈس فراہم نہیں کیے جاتے ہیں، تو ان کے حوالہ جات کے لیے فیلڈز Noneسے بھرے جاتے ہیں۔

ہم ڈیبگ کرتے وقت مواد کو ظاہر کرنے میں مدد کے لیے Node__آبجیکٹ کے لیے ایک ()__str__طریقہ شامل کرتے ہیں۔ خاص طور پر، یہ چائلڈ نوڈس نہیں دکھاتا ہے۔ ہم تھوڑی دیر بعد مکمل درختوں کو ظاہر کرنے کے طریقہ پر تبادلہ خیال کرتے ہیں۔ یہ وہ تمام طریقے ہیں جن کی ضرورت Node__آبجیکٹ کے لیے ہے۔ باقی تمام طریقے جن کی آپ وضاحت کرتے ہیں وہ BinarySearchTreeآبجیکٹ کے لیے ہیں۔

8-2کی فہرست BinarySearchTreeآبجیکٹ کے لیے ()isEmptyطریقہ دکھاتی ہے جو یہ چیک کرتی ہے کہ آیا درخت میں کوئی نوڈس ہیں یا نہیں۔ روٹ ()طریقہ روٹ نوڈ کا ڈیٹا اور کلید نکالتا ہے۔ یہ قطار کے لیے ()peekکی طرح ہے اور اگر درخت خالی ہے تو استثناء پیدا کرتا ہے۔

کچھ پروگرامرز Node__کلاس میں نوڈ کے والدین کا حوالہ بھی شامل کرتے ہیں ۔ ایسا کرنا کچھ کاموں کو آسان بناتا ہے لیکن دوسروں کو پیچیدہ بناتا ہے، لہذا ہم اسے یہاں شامل نہیں کرتے ہیں۔ والدین کا حوالہ شامل کرنے سے باب 5میں بیان کردہ DoublyLinkedList کلاس کی طرح کچھ حاصل ہوتا ہے ، "لنکڈ فہرستیں"؛ یہ بعض سیاق و سباق میں مفید ہے لیکن پیچیدگی کا اضافہ کرتا ہے۔

ہم نے ہر نوڈ کے لیے کلید کو اس کے اپنے فیلڈ میں اسٹور کرکے ایک اور ڈیزائن کا انتخاب کیا ہے۔ صفوں پر مبنی ڈیٹا سٹرکچرز کے لیے، ہم نے ایک کلیدی فنکشن استعمال کرنے کا انتخاب کیا ہے جو ہر صف کے آئٹم سے کلید نکالتا ہے۔ یہ نقطہ نظر صفوں کے لیے زیادہ آسان تھا کیونکہ کیز کو ڈیٹا سے الگ ذخیرہ کرنے کے لیے ڈیٹا سرنی کے ساتھ کلیدی سرنی کے برابر کی ضرورت ہوگی۔ نامزد فیلڈز کے ساتھ نوڈ کلاس کے معاملے میں، ایک کلیدی فیلڈ شامل کرنے سے کوڈ کو شاید زیادہ پڑھنے کے قابل اور کچھ فنکشن کالز سے گریز کرکے کچھ زیادہ موثر بناتا ہے۔ یہ کلید کو ڈیٹا سے زیادہ آزاد بھی بناتا ہے، جس سے لچک میں اضافہ ہوتا ہے اور ڈیٹا میں تبدیلی کے باوجود بھی غیر تبدیل شدہ کلیدوں جیسی رکاوٹوں کو نافذ کرنے کے لیے استعمال کیا جا سکتا ہے۔

BinarySearchTreeکلاس کے کئی طریقے ہیں۔ وہ نوڈس کو تلاش کرنے، داخل کرنے، حذف کرنے اور عبور کرنے کے لیے استعمال ہوتے ہیں۔ اور درخت کی نمائش کے لیے۔ ہم ان میں سے ہر ایک کی الگ الگ تحقیقات کرتے ہیں۔

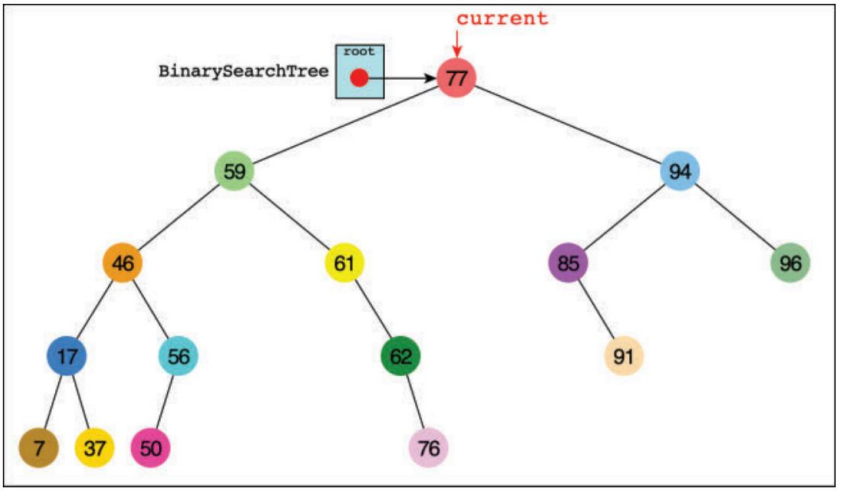
نوڈ تلاش کرنا

ایک مخصوص کلید کے ساتھ نوڈ تلاش کرنا درخت کے بڑے آپریشنز میں سب سے آسان ہے۔ یہ سب سے اہم بھی ہے کیونکہ یہ بائنری سرچ ٹری کے مقصد کے لیے ضروری ہے۔

ویژولائزیشن ٹول ہر نوڈ کے لیے صرف کلید اور اس کے ڈیٹا کے لیے ایک رنگ دکھاتا ہے۔ ذہن میں رکھیں کہ ڈیٹا ڈھانچہ کا مقصد صرف کلید یا سادہ رنگ نہیں بلکہ ریکارڈز کا مجموعہ ذخیرہ کرنا ہے۔ چابیاں سادہ عدد سے زیادہ ہو سکتی ہیں۔ کسی بھی ڈیٹا کی قسم جس کا آرڈر دیا جا سکتا ہے استعمال کیا جا سکتا ہے۔ یہاں دکھایا گیا تصور اور مثالیں اختصار کے لیے عدد کا استعمال کرتی ہیں۔ ایک نوڈ کو اس کی کلید سے دریافت کرنے کے بعد، یہ وہ ڈیٹا ہے جو کال کرنے والے کو واپس کیا جاتا ہے، خود نوڈ کو نہیں۔

نوڈ تلاش کرنے کے لیے ویژولائزیشن ٹول کا استعمال کرتے ہوئے ویژولائزیشن ٹول کو دیکھیں اور ایک نوڈ چنیں، ترجیحاً درخت کے نیچے (جت تک ممکن ہو جڑ سے)۔ اس نوڈ میں دکھایا گیا نمبر اس کی کلیدی قدر ہے۔ ہم یہ ظاہر کرنے جا رہے ہیں کہ کلیدی قدر کے پیش نظر ویژولائزیشن ٹول نوڈ کو کیسے تلاش کرتا ہے۔

اس بحث کے مقاصد کے لیے، ہم کلیدی قدر 50کے ساتھ آئٹم کے حامل نوڈ کو تلاش کرنے کا انتخاب کرتے ہیں، جیسا کہ شکل 8-8میں دکھایا گیا ہے۔ بلاشبہ، جب آپ ویژولائزیشن ٹول چلاتے ہیں، تو آپ کو ایک مختلف درخت مل سکتا ہے اور آپ کو ایک مختلف کلیدی قیمت لینے کی ضرورت پڑ سکتی ہے۔

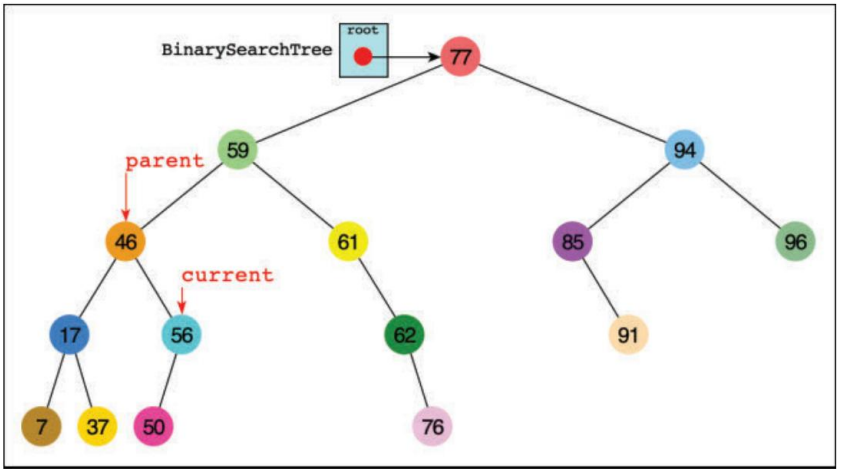


تصویر 8-8کلید 50کے ساتھ نوڈ تلاش کرنا

ٹیکسٹ انٹری باکس میں کلیدی قدر درج کریں، شفٹ کی کو دبائے رکھیں، اور سرچ بٹن کو منتخب کریں، اور پھر اسٹیپ بٹن۔ اسٹیپ بٹن کو بار بار دبانے سے، آپ کلید 50کو تلاش کرنے کے لیے اٹھائے گئے تمام انفرادی اقدامات کو دیکھ سکتے ہیں۔ دوسری پریس پر، موجودہ پوائنٹر درخت کی جڑ میں ظاہر ہوتا ہے، جیسا کہ شکل 8-8میں دیکھا گیا ہے۔ اگلے کلک پر، ایک پیرنٹ پوائنٹر ظاہر ہوتا ہے جو موجودہ پوائنٹر کی پیروی کرے گا۔ ایک لمحے کے لیے اس پوائنٹر اور کوڈ ڈسپلے کو نظر انداز کریں؛ ہم ان کو جلد ہی تفصیل سے بیان کرتے ہیں۔

جیسا کہ ویژولائزیشن ٹول مخصوص نوڈ کو تلاش کرتا ہے، یہ موجودہ نوڈ پر فیصلہ کرتا ہے ۔ یہ مطلوبہ کلید کا موازنہ موجودہ نوڈ پر موجود کلید سے کرتا ہے۔ اگر یہ ایک جیسا ہے تو اسے مطلوبہ نوڈ مل گیا ہے اور وہ چھوڑ سکتا ہے۔ اگر نہیں، تو اسے فیصلہ کرنا ہوگا کہ آگے کہاں دیکھنا ہے۔

شکل 8-8میں موجودہ تیر جڑ سے شروع ہوتا ہے۔ پروگرام گول کلید کی قدر 50کا جڑ کی قدر سے موازنہ کرتا ہے، جو کہ 77ہے۔ گول کلید کم ہے، اس لیے پروگرام جانتا ہے کہ مطلوبہ نوڈ درخت کے بائیں جانب ہونا چاہیے —یا تو جڑ کے بائیں بچے یا ان میں سے ایک اس بچے کی اولاد روٹ کے بائیں چائلڈ کی ویلیو 59ہے، لہذا 50اور 59کا comپیریزن ظاہر کرے گا کہ مطلوبہ نوڈ 59کے بائیں ذیلی درخت میں ہے۔ کرنٹ کرنٹ ایرو 46پر جاتا ہے، اس ذیلی درخت کی جڑ۔ اس بار، 46 50نوڈ سے بڑا ہے، تو یہ دائیں طرف جاتا ہے، نوڈ 56پر، جیسا کہ شکل 8-9 میں دکھایا گیا ہے۔ چند قدم بعد، 50کا 56سے موازنہ پروگرام کو بائیں بچے کی طرف لے جاتا ہے۔ اس لیف نوڈ کا موازنہ ظاہر کرتا ہے کہ 50 نوڈ کی کلیدی قدر کے برابر ہے، اس لیے اسے وہ نوڈ مل گیا ہے جس کی ہم نے تلاش کی تھی۔



تصویر 9-8کلید 50کو تلاش کرنے کا دوسرا آخری مرحلہ

مطلوبہ نوڈ تلاش کرنے کے بعد ویژولائزیشن ٹول تھوڑا بدل جاتا ہے۔ موجودہ تیر نوڈ تیر میں بدل جاتا ہے (اور پیرنٹ پی میں تبدیل ہوتا ہے) ۔ یہ اس وجہ سے ہے کہ کوڈ میں variablesکا نام رکھا گیا ہے، جسے ہم اگلے حصے میں دکھاتے ہیں۔ ٹول نوڈ کو ڈھونڈنے کے بعد اس کے ساتھ کچھ نہیں کرتا، سوائے اس کے گھیرے میں لینے اور ایک پیغام ظاہر کرنے کے کہ یہ مل گیا ہے۔ ایک سنجیدہ پروگرام پایا نوڈ پر کچھ آپریشن کرے گا، جیسے کہ اس کے مواد کو ظاہر کرنا یا اس کے کسی ایک فیلڈ کو تبدیل کرنا۔

نوڈ لسٹنگ 3-8تلاش کرنے کے لیے ازگر کوڈ ()find__اور ()searchطریقوں کا کوڈ دکھاتا ہے ۔ () find__طریقہ نجی ہے کیونکہ یہ نوڈ آبجیکٹ کو واپس کرسکتا ہے۔ BinarySearchTreeکلاس کے کال کرنے والے ایک نوڈ میں ذخیرہ شدہ ڈیٹا حاصل کرنے کے لیے ()searchطریقہ استعمال کرتے ہیں۔

فہرست 3-8اس کی کلید کی بنیاد پر بائنری سرچ ٹری نوڈ تلاش کرنے کے طریقے

کلاس بائنری سرچ ٹری (آبجیکٹ):

...

```
    def __find(خود، مقصد):
        جڑ = خود.__موجودہ
        والدین = خود جبکہ (موجودہ اور
                  موجودہ.کلید != مقصد):
            والدین = موجودہ
            اگر مقصد < موجودہ.کلید:
                موجودہ = موجودہ.بائیں بچہ
            اور:
                موجودہ = موجودہ.دائیں بچہ
        واپسی موجودہ، والدین

    def تلاش (خود، مقصد):
        نوڈ، _ = خود.__find(مقصد)
        اگر نوڈ:
            پرنٹ ("مل گیا!")
        اور:
            پرنٹ ("نہیں ملا")
```

#ایک اندرونی نوڈ تلاش کریں جس کی کلید #گول اور اس کے والدین سے مماثل ہو۔
#روٹ سے شروع کریں اور موجودہ نوڈ کے والدین کو ٹریک کریں #جب کہ دریافت
کرنے کے لیے ایک درخت باقی ہے اور موجودہ کلید مقصد نہیں ہے #ایک سطح نیچے جانے
کے لیے تیار ہوں
گول ! :(والدین =current.key =موجودہ موجودہ current.leftChild) =اگر گول # current.key else <گول
موجودہ ہے۔ رائٹ چائلڈ)
#ایڈوانس کرنٹ کو بائیں سب ٹری جب

#موجودہ کلید سے کم، ورنہ ٹھیک ہے۔

#اگر لوپ کسی نوڈ پر ختم ہوتا ہے، تو اس میں گول کلید کی واپسی ہونی چاہیے (موجودہ، والدین)
#نوڈ یا کوئی نہیں اور والدین کو واپس کریں۔

```
def search(self, goal): # مقصد کی کلید کے ساتھ منسلک ڈیٹا حاصل کرنے کا عوامی طریقہ
   node, p = self.__find(goal) # سب سے پہلے، نوڈ تلاش کریں
   return node.data if node else None # w/ واپس گول اور کوئی
```

find()__کی واحد دلیل گول ہے ، کلیدی قدر جس کو تلاش کرنا ہے۔ یہ معمول متغیر کرنٹ بناتا ہے تاکہ اس نوڈ کو بولڈ کیا جا سکے جس کی فی الحال جانچ کی جا رہی ہے۔ روٹین جڑ سے شروع ہوتی ہے –واحد نوڈ جس تک یہ براہ راست رسائی حاصل کر سکتا ہے۔ یعنی یہ روٹ پر کرنٹ سیٹ کرتا ہے۔ یہ خود کے لیے ایک متغیر متغیر بھی متعین کرتا ہے، جو کہ درخت کی چیز ہے۔ ویژولائزیشن ٹول میں، پیرنٹ ٹری آبجیکٹ کی طرف اشارہ کرنا شروع کرتا ہے۔ چونکہ پیرنٹ لنکس نوڈس میں محفوظ نہیں ہوتے ہیں، اس لیے find()__طریقہ کرنٹ کے پیرنٹ نوڈ کو ٹریک کرتا ہے تاکہ وہ اسے گول نوڈ کے ساتھ کالر کو واپس کر سکے۔ یہ صلاحیت دوسرے طریقوں میں بہت کارآمد ثابت ہوگی۔ پیرنٹ متغیر ہمیشہ یا تو BinarySearchTreeکو تلاش کیا جاتا ہے یا اس کے Node__آبجیکٹ میں سے ایک ہوتا ہے۔

whileلوپ میں ، find()__پہلے تصدیق کرتا ہے کہ کرنٹ Noneنہیں ہے اور کچھ موجودہ نوڈ کا حوالہ دیتا ہے۔ اگر ایسا نہیں ہوتا ہے تو، تلاش لیف نوڈ سے آگے نکل گئی ہے (یا خالی درخت سے شروع ہوئی ہے)، اور گول نوڈ درخت میں نہیں ہے۔ while test comکا دوسرا حصہ موجودہ نوڈ کے کلیدی فیلڈ کی قدر کے ساتھ ، تلاش کی جانے والی قدر ، گول کا موازنہ کرتا ہے۔ اگر کلید ملتی ہے، تو لوپ کیا جاتا ہے. اگر ایسا نہیں ہوتا ہے، تو موجودہ کو مناسب ذیلی درخت کی طرف بڑھنے کی ضرورت ہے۔ سب سے پہلے، یہ پیرنٹ کو موجودہ نوڈ بننے کے لیے اپ ڈیٹ کرتا ہے اور پھر کرنٹ کو اپ ڈیٹ کرتا ہے۔ اگر ہدف کرنٹ کی کلید سے کم ہے تو کرنٹ اپنے بائیں بچے کی طرف بڑھتا ہے۔ اگر ہدف کرنٹ کی کلید سے بڑا ہے ، تو کرنٹ اپنے صحیح بچے کی طرف بڑھتا ہے۔

نوڈ نہیں مل سکتا

اگر کرنٹ Noneکے برابر ہو جاتا ہے ، تو آپ جس نوڈ کو تلاش کر رہے تھے اسے تلاش کیے بغیر لائن کے آخر تک پہنچ گئے ہیں، لہذا یہ درخت میں نہیں ہو سکتا۔ ایسا ہو سکتا ہے اگر روٹ نوڈ کوئی نہ ہو یا اگر چائلڈ لنکس کو فالو کرنے سے بچے کے بغیر نوڈ بن جائے (اس طرف جہاں گول کی جائے گی)۔ موجودہ نوڈ (کوئی نہیں) اور اس کے والدین دونوں کو کال کرنے والے کو واپس کر دیا جاتا ہے تاکہ نتیجہ کی نشاندہی کی جا سکے۔ ویژولائزیشن ٹول میں، ایک کلید داخل کرنے کی کوشش کریں جو درخت میں ظاہر نہیں ہوتی ہے اور تلاش کو منتخب کریں۔ اس کے بعد آپ دیکھیں گے کہ موجودہ پوائنٹر موجودہ نوڈس کے ذریعے اترتا ہے اور اس جگہ پر اترتا ہے جہاں کلید ملنی چاہیے لیکن کوئی نوڈ موجود نہیں ہے۔

"خالی جگہ" کی طرف اشارہ کرتا ہے کہ متغیر کی قدر کوئی نہیں ہے۔

نوڈ ملا

اگر موجودہ وقت کے دوران لوپ کی حالت مطمئن نہیں ہوتی ہے جبکہ موجودہ درخت میں کچھ نوڈ کا حوالہ دیتا ہے، تو لوپ باہر نکل جاتا ہے، اور موجودہ کلید کا مقصد ہونا چاہیے۔ یعنی، اس نے نوڈ کو تلاش کیا ہے اور اس کا موجودہ حوالہ دیا ہے۔ یہ پیرنٹ ریفرنس کے ساتھ نوڈ کا حوالہ واپس کرتا ہے تاکہ روٹین جسے find()__کہا جاتا ہے نوڈ (یا اس کے والدین کے) ڈیٹا میں سے کسی تک رسائی حاصل کر سکے۔ نوٹ کریں کہ یہ کلید تلاش کرنے میں کامیابی اور ناکامی دونوں کے لیے کرنٹ کی قدر لوٹاتا ہے ۔ جب مقصد نہیں ملتا ہے تو یہ کوئی نہیں ہے۔

سرچ ()طریقہ اپنے نوڈ اور پیرنٹ (p)متغیرات کو سیٹ کرنے کے لیے find()__طریقہ کو کال کرتا ہے۔ find()__طریقہ واپس آنے کے بعد آپ ویژولائزیشن ٹول میں یہی دیکھتے ہیں ۔ اگر کوئی غیر کا حوالہ نہیں ملا تو تلاش() اس نوڈ کا ڈیٹا لوٹاتا ہے۔ اس صورت میں، طریقہ یہ مانتا ہے کہ نوڈس میں ذخیرہ شدہ ڈیٹا آئٹمز کبھی بھی کوئی نہیں ہو سکتے۔ بصورت دیگر، کال کرنے والا ان میں تمیز نہیں کر سکے گا۔

درخت کی کارکردگی جیسا کہ آپ دیکھ سکتے ہیں، نوڈ کو تلاش کرنے کے لیے درکار وقت کا انحصار درخت میں اس کی گہرائی، جڑ سے نیچے کی سطحوں کی تعداد پر ہوتا ہے۔ اگر درخت متوازن ہے، تو یہ O(log N)وقت ہے، یا اس سے زیادہ خاص طور پر O(log2 N)وقت ہے، لاگرتھم سے بیس 2، جہاں N نوڈس کی تعداد ہے۔

یہ بالکل بائنری تلاش کی طرح ہے جو صفوں میں کی جاتی ہے جہاں ہر موازنہ کے بعد آدھے نوڈس کو ختم کردیا جاتا ہے۔ ایک مکمل متوازن درخت بہترین صورت ہے۔ بدترین صورت میں، درخت مکمل طور پر غیر متوازن ہے، جیسا کہ شکل 8-6میں دکھایا گیا ہے، اور مطلوبہ وقت O(N)ہے۔

ہم اس باب کے اختتام کی طرف find()__اور دیگر کارروائیوں کی کارکردگی پر تبادلہ خیال کرتے ہیں ۔

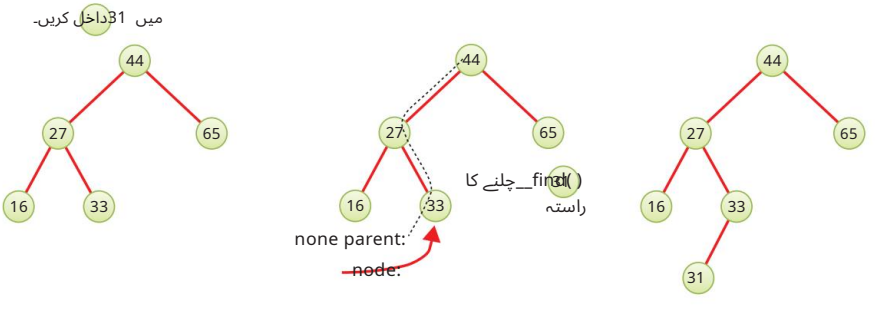
نوڈ داخل کرنا

نوڈ داخل کرنے کے لیے، آپ کو پہلے اسے داخل کرنے کے لیے جگہ تلاش کرنا ہوگی۔ یہ وہی عمل ہے جو کسی ایسے نوڈ کو تلاش کرنے کی کوشش کرتا ہے جو کہ موجود نہیں ہے، جیسا کہ پہلے "نوڈ کو تلاش نہیں کر سکتے" سیکشن میں بیان کیا گیا ہے۔ آپ جڑ سے مناسب نوڈ کی طرف راستے پر چلتے ہیں۔ یہ یا تو ایک نوڈ ہے جس کی کلید نوڈ ڈالی جائے گی یا کوئی نہیں، اگر یہ نئی کلید ہے۔ اگر یہ ایک ہی کلید ہے، تو آپ اسے صحیح ذیلی درخت میں ڈالنے کی کوشش کر سکتے ہیں، لیکن ایسا کرنے سے کچھ پیچیدگیاں بڑھ جاتی ہیں۔ دوسرا آپشن اس نوڈ کے ڈیٹا کو نئے ڈیٹا سے تبدیل کرنا ہے۔ ابھی کے لیے، ہم صرف منفرد کلیدوں کو داخل کرنے کی اجازت دیتے ہیں۔ ہم بعد میں ڈپلیکیٹ کیز پر بات کرتے ہیں۔

اگر داخل کرنے کی کلید درخت میں نہیں ہے، تو پھر find()__نوڈ کے حوالے کے لیے پیرنٹ ریفرنس کے ساتھ Noneواپس کرتا ہے ۔ نیا نوڈ والدین کے بائیں یا دائیں بچے کے طور پر جڑا ہوا ہے، اس بات پر منحصر ہے کہ آیا نئے نوڈ کی کلید والدین کی کلید سے کم ہے یا زیادہ۔ اگر find()__کے ذریعہ واپس کیا گیا پیرنٹ حوالہ خود ہے ، خود ، BinarySearchTreeتو نوڈ روٹ نوڈ بن جاتا ہے۔

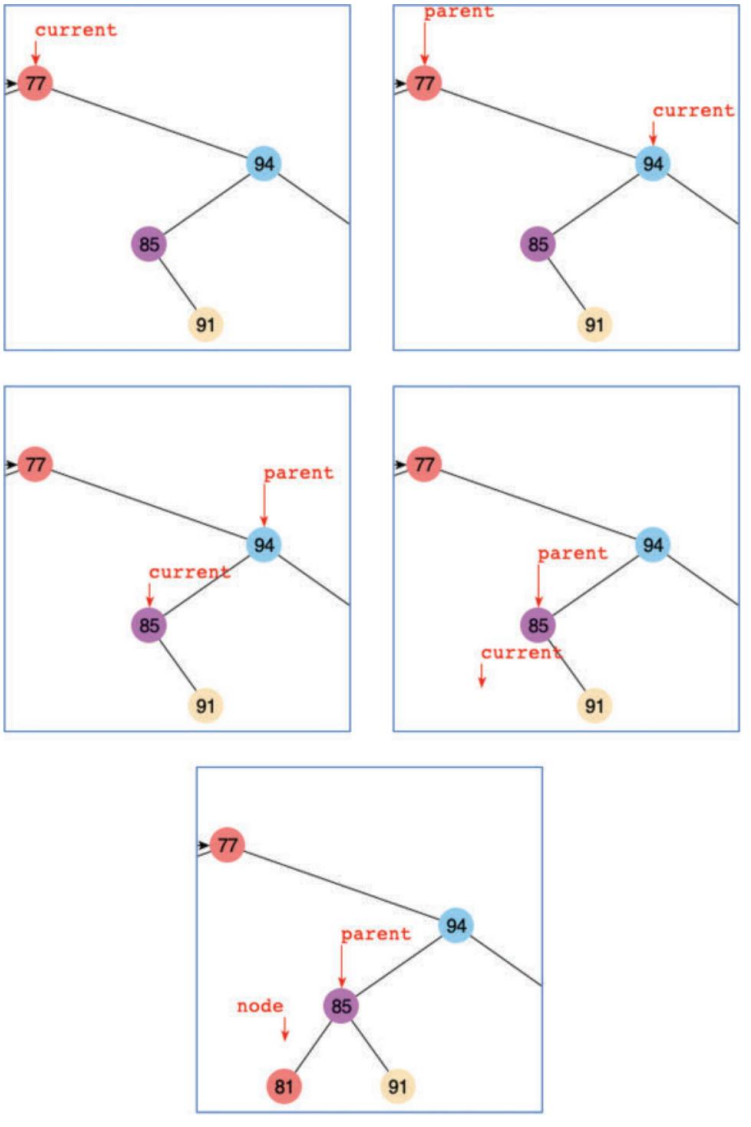
شکل 8-10ایک نوڈ داخل کرنے کے عمل کی وضاحت کرتا ہے، کلید 31کے ساتھ، درخت میں۔ find (31)__طریقہ روٹ نوڈ سے راستے پر چلنا شروع کرتا ہے۔ کیونکہ 31روٹ نوڈ کلید 44سے کم ہے، یہ بائیں چائلڈ لنک کی پیروی کرتا ہے۔ اس بچے کی کلید 27ہے، لہذا یہ اس بچے کے دائیں چائلڈ لنک کی پیروی کرتی ہے۔ وہاں یہ کلید 33کا سامنا کرتا ہے، لہذا یہ دوبارہ بائیں چائلڈ لنک کی پیروی کرتا ہے۔

یہ کوئی نہیں ہے، لہذا find(31)__کلید 33کے ساتھ لیف نوڈ کی طرف اشارہ کرنے والے پیرنٹ کے ساتھ رک جاتا ہے۔ کلید 31کے ساتھ نیا لیف نوڈ بن جاتا ہے، اور والدین کے بائیں چائلڈ لنک کو اس کا حوالہ دینے کے لیے سیٹ کیا جاتا ہے۔



تصویر 8-10بائنری سرچ ٹری میں نوڈ ڈالنا

نوڈ داخل کرنے کے لیے ویژولائزیشن ٹول کا استعمال کرتے ہوئے ویژولائزیشن ٹول کے ساتھ نیا نوڈ داخل کرنے کے لیے، ایک کلیدی قدر درج کریں جو درخت میں نہیں ہے اور داخل کریں بٹن کو منتخب کریں۔ پروگرام کے لیے پہلا قدم یہ معلوم کرنا ہے کہ اسے کہاں داخل کیا جانا چاہیے۔ مثال کے طور پر، پہلے کی مثال سے درخت میں 81ڈالنا ()find__فہرست 3-8کا طریقہ کہتا ہے، جس کی وجہ سے تلاش تصویر 8-11میں دکھائے گئے راستے پر چلتی ہے۔



تصویر 8-11ویژولائزیشن ٹول کا استعمال کرتے ہوئے کلید 81کے ساتھ نوڈ داخل کرنے کے اقدامات

موجودہ پوائنٹر روٹ نوڈ سے کلید 77سے شروع ہوتا ہے۔ 81کو بڑا ہونے کی تلاش میں، یہ دائیں چائلڈ، نوڈ 94پر جاتا ہے ۔

نوڈ 85تک۔ پیرنٹ پوائنٹر ان میں سے ہر ایک مرحلے پر موجودہ پوائنٹر کی پیروی کرتا ہے۔ جب کرنٹ نوڈ 85تک پہنچتا ہے، تو یہ اپنے بائیں بچے کے پاس جاتا ہے اور اسے غائب پایا جاتا ہے۔ ()find__پر کال کوئی نہیں اور پیرنٹ پوائنٹر واپس کرتی ہے ۔

خالی بچے کے ساتھ پیرنٹ نوڈ کا پتہ لگانے کے بعد جہاں نئی کلید کو جانا چاہیے، ویژولائزیشن ٹول کلید 81کے ساتھ ایک نیا نوڈ بناتا ہے، جس میں کچھ ڈیٹا رنگ سے ظاہر ہوتا ہے، اور نوڈ 85کے بائیں چائلڈ پوائنٹر، پیرنٹ، کو پوائنٹ کرنے کے لیے سیٹ کرتا ہے۔ اس پر. ()find__کے ذریعہ لوٹا ہوا نوڈ پوائنٹر ہٹا دیا گیا ہے کیونکہ یہ اب بھی کوئی نہیں ہے۔

نوڈ داخل کرنے کے لیے پائتھون کوڈ ()insertطریقہ کلید اور ڈیٹا داخل کرنے کے لیے پیرامیٹرز لیتا ہے، جیسا کہ فہرست 8-4میں دکھایا گیا ہے۔ یہ نئے نوڈ کی کلید کے ساتھ ()find__طریقہ کو کال کرتا ہے تاکہ یہ معلوم کیا جا سکے کہ آیا وہ کلید پہلے سے موجود ہے اور اس کا پیرنٹ نوڈ کہاں ہونا چاہیے۔ یہ نفاذ درخت میں صرف منفرد کلیدوں کی اجازت دیتا ہے، لہذا اگر اسے اسی کلید کے ساتھ کوئی نوڈ ملتا ہے، تو ()insertاس کلید کے لیے ڈیٹا کو اپ ڈیٹ کرتا ہے اور Falseلوٹاتا ہے تاکہ یہ ظاہر کیا جا سکے کہ کوئی نیا نوڈ نہیں بنایا گیا تھا۔

فہرست BinarySearchTree 4-8کا ()insertطریقہ

```
کلاس BinarySearchTree(آبجیکٹ):
...
    def insert(self, key, data): node, parent = self.__find(key) if node:
        #بائنری میں ایک نیا نوڈ داخل کریں #تلاش کے درخت کو تلاش کریں کہ
        اس کی کلید اسے کہاں رکھتی ہے اور اس کا ڈیٹا اسٹور کرتی ہے

        #درخت میں کلید تلاش کرنے کی کوشش کریں #اور اس کا بنیادی نوڈ
        حاصل کریں۔
        #اگر ہمیں اس کلید کے ساتھ کوئی نوڈ ملتا ہے، #پھر نوڈ کے ڈیٹا کو اپ
        ڈیٹ کریں #اور بغیر اندراج کے جھنڈا واپس کریں۔
        node.data = ڈیٹا
        غلط واپسی

    #خالی درختوں کے لیے، نیا نوڈ داخل کریں۔
    self.__root = self.__Node(key, data) #اگر نئی کلید والدین کی کلید سے کم ہے تو (data, right=node)
    parent.leftChild = self.__Node( key, data, right=node) #اگر والدین خود ہیں : parent.key: #
    #بائیں کے طور پر نیا نوڈ داخل کریں۔

    #والدین کا بچہ
    #بصورت دیگر والدین کے دائیں #بچے کے طور پر نیا نوڈ داخل کریں۔
    دوسری:
        parent.rightChild = self.__Node( key, data, right=node)

    #درست اندراج کے لیے جھنڈا واپس کریں۔
    واپس سچ
```

اگر کوئی مماثل نوڈ نہیں ملا، تو اندراج اس بات پر منحصر ہے کہ ()find__سے کس قسم کا پیرنٹ حوالہ واپس کیا گیا تھا ۔ اگر یہ خود BinarySearchTree، خالی ہونا ضروری ہے، لہذا نیا نوڈ درخت کا جڑ نوڈ بن جاتا ہے ۔ بصورت دیگر، والدین ایک نوڈ ہے، لہذا ()insert نئے نوڈ کی کلید کا والدین کی کلید سے موازنہ کرکے فیصلہ کرتا ہے کہ کون سا بچہ نیا نوڈ حاصل کرے گا۔ اگر نئی کلید کم ہے، تو نیا نوڈ بائیں چائلڈ بن جاتا ہے۔ دوسری صورت میں، یہ صحیح بچہ بن جاتا ہے۔ آخر میں، ()insertواپسی Trueاس بات کی نشاندہی کرنے کے لیے کہ داخل کرنا کامیاب ہو گیا۔

جب ()insertنیا نوڈ بناتا ہے، تو یہ نئے نوڈ کے رائٹ چائلڈ کو ()find__سے واپس آنے والے نوڈ پر سیٹ کرتا ہے۔ آپ حیران ہوں گے کہ یہ وہاں کیوں ہے، خاص طور پر اس لیے کہ اس مقام پر نوڈ صرف Noneہو سکتا ہے (اگر یہ Noneنہ ہوتا تو ()insertاس مقام تک پہنچنے سے پہلے Falseواپس کر دیتا)۔ وجہ ڈپلیکیٹ چابیاں کے ساتھ کیا کرنا ہے واپس چلا جاتا ہے. اگر آپ ڈپلیکیٹ چابیاں والے نوڈس کی اجازت دیتے ہیں، تو آپ کو انہیں کہیں رکھنا چاہیے۔ بائنری سرچ ٹری ڈیف انیشن کا کہنا ہے کہ نوڈ کی کلید اس کے دائیں بچے کی کلید سے کم یا اس کے برابر ہوتی ہے۔ لہذا، اگر آپ ڈپلیکیٹ چابیاں کی اجازت دیتے ہیں، تو ڈپلیکیٹ نوڈ بائیں بچے میں نہیں جا سکتا۔ نئے نوڈ کے رائٹ چائلڈ کے طور پر Noneکے علاوہ کسی اور چیز کی وضاحت کرکے ، اسی کلید کے ساتھ دوسرے نوڈس کو برقرار رکھا جا سکتا ہے۔ ہم ایک مشق کے طور پر چھوڑتے ہیں کہ ڈپلیکیٹ کیز کے ساتھ نوڈس کیسے داخل کریں (اور حذف کریں)۔

درخت کو عبور کرنا

درخت کو عبور کرنے کا مطلب ہے ہر نوڈ کو ایک مخصوص ترتیب میں جانا۔ درخت کو عبور کرنا کچھ حالات میں مفید ہے جیسے کہ تمام ریکارڈز کو تلاش کرنے کے لیے ان کو تلاش کرنا جن کے لیے کچھ کارروائی کی ضرورت ہے (مثال کے طور پر، گاڑی کے پرزے جو کسی خاص ملک سے حاصل کیے گئے ہوں)۔

یہ عمل عام طور پر نوڈس کو تلاش کرنے، داخل کرنے اور حذف کرنے کے طور پر استعمال نہیں ہوسکتا ہے لیکن اس کے باوجود یہ اہم ہے۔

آپ ایک درخت کو تین آسان طریقوں سے عبور کر سکتے ہیں۔ انہیں پری آرڈر، ان آرڈر اور پوسٹ آرڈر کہا جاتا ہے۔ بائنری سرچ ٹریز کے لیے سب سے زیادہ استعمال ہونے والا آرڈر ان آرڈر ہے، تو آئیے پہلے اسے دیکھتے ہیں اور پھر دوسرے دو پر مختصر طور پر واپس آتے ہیں۔

ان آرڈر ٹراورسل

بائنری سرچ ٹری کا ان آرڈر ٹراورسل تمام نوڈس کو ان کی کلیدی اقدار کے صعودی ترتیب میں دیکھنے کا سبب بنتا ہے ۔ اگر آپ بائنری ٹری میں ڈیٹا کی ایک فہرست بنانا چاہتے ہیں جو ان کی چابیاں کے مطابق ترتیب دی گئی ہے، تو یہ ایسا کرنے کا ایک طریقہ ہے۔

ٹراورسل کرنے کا سب سے آسان طریقہ تکرار کا استعمال ہے (جس پر باب 6میں بحث کی گئی ہے)۔
پورے درخت کو عبور کرنے کے لئے ایک تکراری طریقہ کو نوڈ کے ساتھ دلیل کے طور پر کہا جاتا ہے۔
ابتدائی طور پر، یہ نوڈ جڑ ہے. طریقہ کار کو صرف تین چیزیں کرنے کی ضرورت ہے:

1.نوڈ کے بائیں سب ٹری کو عبور کرنے کے لیے خود کو کال کریں۔

2.نوڈ پر جائیں۔

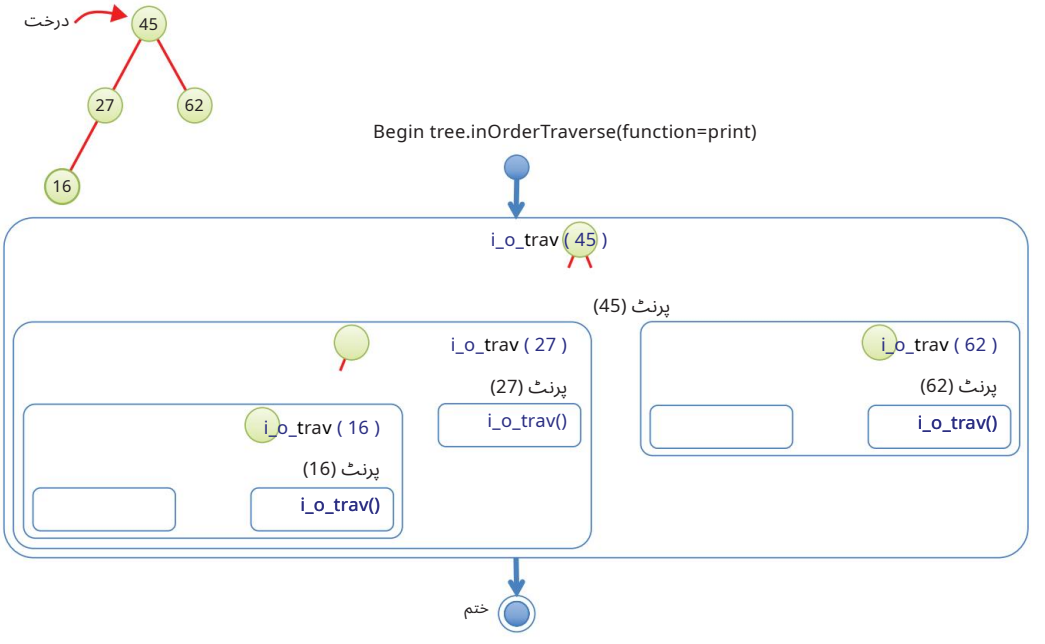
3.نوڈ کے دائیں ذیلی درخت کو عبور کرنے کے لیے خود کو کال کریں۔

یاد رکھیں کہ نوڈ پر جانے کا مطلب ہے اس کے لیے کچھ کرنا: اسے ڈسپلے کرنا، کسی فیلڈ کو اپ ڈیٹ کرنا، اسے قطار میں شامل کرنا، اسے فائل میں لکھنا، یا کچھ بھی۔

تین ٹراورسل آرڈرز کسی بھی بائنری ٹری کے ساتھ کام کرتے ہیں، نہ صرف بائنری سرچ ٹری کے ساتھ۔
ٹراورسل میکانزم نوڈس کی کلیدی اقدار پر کوئی توجہ نہیں دیتا ہے۔ یہ صرف نوڈ کے بچوں اور ڈیٹا سے متعلق ہے۔ دوسرے لفظوں میں، ان آرڈر ٹراورسل کا مطلب ہے "کلیدی اقدار کو بڑھانے کی ترتیب میں" صرف اس صورت میں جب بائنری سرچ ٹری کے معیار کو درخت میں نوڈس رکھنے کے لیے استعمال کیا جاتا ہے۔ ان آف آرڈر سے مراد بائیں اور دائیں ذیلی درختوں کے درمیان ایک نوڈ کا دورہ کیا جاتا ہے۔ پری آرڈر کا مطلب ہے بچوں سے ملنے سے پہلے نوڈ کا دورہ کرنا، اور پوسٹ آرڈر بچوں سے ملنے کے بعد نوڈ کا دورہ کرنا۔ یہ فرق ایسا ہے۔

باب 4میں بیان کردہ ریاضی کے تاثرات کے لیے انفکس اور پوسٹ فکس اشارے کے درمیان فرق ، "اسٹیک اور قطاریں۔"

یہ دیکھنے کے لیے کہ یہ بار بار چلنے والا ٹراورسل کیسے کام کرتا ہے، شکل 12-8ان کالوں کو دکھاتی ہے جو ایک چھوٹے بائنری ٹری کے ان آرڈر ٹراورسل کے دوران ہوتی ہیں۔ درخت کا متغیر چار نوڈ بائنری سرچ ٹری کا حوالہ دیتا ہے۔ اعداد و شمار درخت پر ایک ()inOrderTraverseطریقہ کی درخواست کو ظاہر کرتا ہے جو اس کے ہر نوڈس پر پرنٹ فنکشن کو کال کرے گا۔



شکل 12-8ایک چھوٹے سے درخت کی ترتیب سے گزرنا

نیلے گول مستطیل ہر ذیلی درخت پر دوبارہ آنے والی کالیں دکھاتے ہیں۔ اعداد و شمار میں تمام کالوں کو فٹ کرنے کے لئے تکراری طریقہ کا نام ()i_o_travکے طور پر مختص کیا گیا ہے۔ پہلی (باہر ترین) کال روٹ نوڈ (کلید 45)پر ہوتی ہے۔ ہر تکراری کال پہلے بیان کردہ تین مراحل کو انجام دیتی ہے۔ سب سے پہلے، یہ بائیں ذیلی درخت پر ایک تکراری کال کرتا ہے، جس کی جڑ کلید 27پر ہوتی ہے۔ یہ شکل کے بائیں جانب ایک اور نیلے گول مستطیل کے طور پر ظاہر ہوتا ہے۔

کلید 27پر جڑی ہوئی سب ٹری کو پروسیس کرنا اس کے بائیں سب ٹری پر ایک بار بار کال کرنے سے شروع ہوتا ہے، جو کلید 16پر جڑی ہوئی ہے۔ ایک اور مستطیل اس کال کو نیچے بائیں طرف دکھاتا ہے۔ پہلے کی طرح، اس کا پہلا کام اس کے بائیں سب ٹری پر بار بار آنے والی کال کرنا ہے۔ وہ ذیلی درخت خالی ہے کیونکہ یہ ایک لیف نوڈ ہے اور تصویر میں بغیر کسی دلیل کے ()i_o_travکو کال کے طور پر دکھایا گیا ہے۔ کیونکہ سب ٹری خالی ہے، کچھ نہیں ہوتا اور بار بار آنے والی کال واپس آتی ہے۔

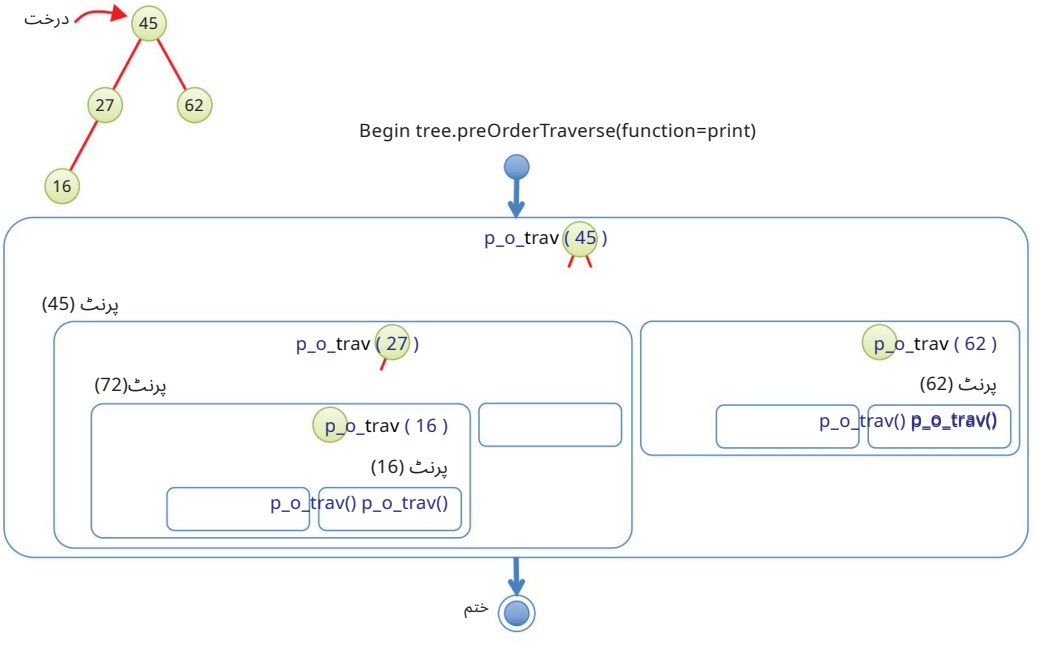
(16)i_o_travکو کال میں واپس ، یہ اب مرحلہ 2تک پہنچ جاتا ہے اور نوڈ پر ہی فنکشن، پرنٹ کو انجام دے کر نوڈ کو "وزٹ" کرتا ہے۔ یہ تصویر میں بطور پرنٹ (16)سیاہ میں دکھایا گیا ہے۔
عام طور پر، نوڈ پر جانا صرف نوڈ کی کلید پرنٹ کرنے سے زیادہ کام کرتا ہے۔ یہ نوڈ میں ذخیرہ شدہ ڈیٹا پر کچھ کارروائی کرے گا۔ اعداد و شمار اس عمل کو نہیں دکھاتا ہے، لیکن یہ اس وقت ہوگا جب پرنٹ (16)کو انجام دیا جائے گا۔

کلید 16کے ساتھ نوڈ پر جانے کے بعد، یہ مرحلہ 3کا وقت ہے: خود کو دائیں ذیلی درخت پر کال کریں۔
کلید 16والے نوڈ کا کوئی صحیح بچہ نہیں ہے، جو سب سے چھوٹے سائز کے مستطیل کے طور پر ظاہر ہوتا ہے کیونکہ یہ خالی ذیلی درخت پر کال ہے۔ یہ کلید 16پر جڑے ہوئے ذیلی درخت کے لیے عمل کو مکمل کرتا ہے۔ کنٹرول واپس کالر کے پاس جاتا ہے، سب ٹری پر کال کلید 27پر جڑ جاتی ہے۔

باقی پروسیسنگ اسی طرح ترقی کرتی ہے۔ کلید 27کے ساتھ نوڈ کا دورہ پرنٹ (27)کو انجام دیتا ہے اور پھر اس کے خالی دائیں ذیلی درخت پر کال کرتا ہے۔ اس سے نوڈ 27پر کال ختم ہو جاتی ہے اور کنٹرول درخت کی جڑ، نوڈ 45پر کال پر واپس چلا جاتا ہے۔ پرنٹ (45) پر عمل کرنے کے بعد، یہ اپنے دائیں (غیر خالی) ذیلی درخت کو عبور کرنے کے لیے کال کرتا ہے۔ یہ درخت کا چوتھا اور آخری نوڈ ہے، نوڈ 62۔ یہ اپنے خالی بائیں سب ٹری پر کال کرتا ہے، پرنٹ (62)پر عمل کرتا ہے، اور اپنے خالی دائیں سب ٹری پر کال کے ساتھ ختم کرتا ہے۔ کنٹرول روٹ نوڈ، 45پر کال کے ذریعے بیک اپ جاتا ہے، اور اس سے درخت کا مکمل راستہ ختم ہو جاتا ہے۔

پری آرڈر اور پوسٹ آرڈر ٹراورسلز

دوسرے دو ٹراورسل آرڈرز ایک جیسے ہیں: صرف نوڈ پر جانے کا سلسلہ تبدیل ہوتا ہے۔
پری آرڈر ٹراورسل کے لیے، نوڈ کو پہلے وزٹ کیا جاتا ہے، اور پوسٹ آرڈر کے لیے، اسے آخری بار دیکھا جاتا ہے۔ دو ذیلی درختوں کو ہمیشہ ایک ہی ترتیب میں دیکھا جاتا ہے: بائیں اور پھر دائیں شکل 13-8اسی چار نوڈ درخت پر پری آرڈر ٹراورسل کے عمل کو ظاہر کرتا ہے جیسا کہ شکل 12-8میں ہے۔ پرنٹ ()فنکشن کا عمل دو ذیلی درختوں کو دیکھنے سے پہلے ہوتا ہے۔ اس کا مطلب ہے کہ پری آرڈر ٹراورسل 62 45، 27، 16کے مقابلے میں 62، 16، 27، 45پرنٹ کرے گا۔ بڑا ہے۔



شکل 13-8ایک چھوٹے سے درخت کا پری آرڈر کریں۔

پیتھون کوڈ فار ٹراورسنگ آئیے اب ان آرڈر ٹراورسل کو لاگو کرنے کا ایک آسان طریقہ دیکھتے ہیں۔ جیسا کہ آپ نے ڈھیروں، قطاروں، منسلک فہرستوں، اور دیگر ڈیٹا ڈھانچے میں دیکھا ہے، tra versalکی وضاحت ایک دلیل کے طور پر پاس کردہ فنکشن کا استعمال کرتے ہوئے کرنا سیدھا ہے جو ڈھانچے میں محفوظ ہر آئٹم پر لاگو ہوتا ہے۔ درختوں کے ساتھ دلچسپ فرق یہ ہے کہ تکرار اسے بہت کمپیکٹ بناتی ہے۔

چونکہ ان درختوں کی نمائندگی دو کلاسز، BinarySearchTreeاور Node__کے ذریعے کی گئی ہے، آپ کو ایسے طریقوں کی ضرورت ہے جو دونوں قسم کی اشیاء پر کام کر سکیں۔ فہرست 5-8میں، ()inOrderTraverseطریقہ BinarySearchTreeآبجیکٹ پر ٹراورسل کو ہینڈل کرتا ہے ۔ یہ ٹراورسل کے لیے عوامی انٹرفیس کے طور پر کام کرتا ہے اور ذیلی درختوں پر اصل کام کرنے کے لیے ایک نجی طریقہ ()inOrderTraverse__کو کال کرتا ہے۔ یہ روٹ نوڈ کو پرائیویٹ طریقہ میں منتقل کرتا ہے اور واپس آتا ہے۔

فہرست سازی () inOrderTraverse 5-8کا بار بار عمل درآمد

```
...

#درخت کے تمام نوڈس کو ترتیب سے دیکھیں
#اور ہر نوڈ پر ایک فنکشن لاگو کریں #خود سے شروع ہونے والے تکراری ورژن کو کال
کریں۔root، function=function) # root node.__

#بار بار ایک ذیلی درخت کو ترتیب میں دیکھیں
#ذیلی درخت کی جڑ نوڈ ہے۔
#چیک کریں کہ یہ اصلی ذیلی درخت ہے۔
#بائیں ذیلی درخت کو عبور کریں۔

#اس نوڈ پر جائیں۔
#دائیں ذیلی درخت کو عبور کریں۔

def inOrderTraverse(
خود، فنکشن =پرنٹ):
self.__inOrderTraverse(

def __inOrderTraverse(خود، نوڈ، فنکشن):
اگر نوڈ:
self.__inOrderTraverse(
function)
self.__inOrderTraverse(node.rightChild,
function) function(node)
node.leftChild,
```

نجی طریقہ اپنے نوڈ پیرامیٹر کے لیے ایک Node__آبجیکٹ (یا Noneکی توقع کرتا ہے اور نوڈ پر جڑی ذیلی درخت پر تین مراحل انجام دیتا ہے۔ سب سے پہلے، یہ نوڈ کو چیک کرتا ہے اور فوری طور پر واپس آتا ہے اگر یہ Noneنہیں ہے کیونکہ یہ خالی ذیلی درخت کی نشاندہی کرتا ہے۔ جائز نوڈس کے لیے، یہ نوڈ کے بائیں چائلڈ پر کارروائی کرنے کے لیے سب سے پہلے اپنے آپ کو ایک بار بار کال کرتا ہے۔ دوسرا، یہ نوڈ پر فنکشن کو پکار کر اس کا دورہ کرتا ہے۔ تیسرا، یہ نوڈ کے دائیں بچے پر کارروائی کرنے کے لیے ایک بار بار کال کرتا ہے۔ بس اتنا ہی ہے۔

ٹراورسل کے لیے جنریٹر کا استعمال () inOrderTraverseطریقہ سیدھا ہے، لیکن اس میں کم از کم تین خامیاں ہیں۔

سب سے پہلے، دوسرے آرڈرنگ کو لاگو کرنے کے لیے، آپ کو یا تو مزید طریقے لکھنے ہوں گے یا ایک ایسا پیرامیٹر شامل کرنے کی ضرورت ہوگی جو انجام دینے کے لیے ترتیب کی وضاحت کرے۔

دوسرا، ہر نوڈ کو "وزٹ" کرنے کے لیے ایک دلیل کے طور پر منظور ہونے والے فنکشن کو Node__آبجیکٹ کو بطور دلیل لینے کی ضرورت ہے۔ یہ BinarySearchTreeکے اندر ایک نجی کلاس ہے جو نوڈس کو کال کرنے والے کے ذریعے ہیرا پھیری سے بچاتی ہے۔ ایک متبادل جو Node__آبجیکٹ کا حوالہ دینے سے گریز کرتا ہے وہ صرف ڈیٹا فیلڈ (اور شاید کلیدی فیلڈ) میں گزرنا ہوگا۔

وزٹ فنکشن کے دلائل کے طور پر ہر نوڈ کا۔ اس نقطہ نظر سے یہ کم ہو جائے گا کہ کال کرنے والا نوڈ کے ساتھ کیا کر سکتا ہے اور اسے دوسرے نوڈ حوالوں کو تبدیل کرنے سے روک دے گا۔

تیسرا، ہر نوڈ پر انجام دینے کے عمل کو بیان کرنے کے لیے فنکشن کا استعمال کرنے کی اپنی حدود ہیں۔
عام طور پر، فنکشنز ہر بار ایک ہی آپریشن کرتے ہیں جب ان کو پکارا جاتا ہے اور پچھلی کالوں کی تاریخ کے بارے میں نہیں جانتے ہیں۔
درخت کی طرح ڈیٹا سٹرکچر کے ٹراورسل کے دوران، پچھلے نوڈ وزٹ کے نتائج کو استعمال کرنے کے قابل ہونا ممکنہ کارروائیوں کو ڈرامائی طور پر بڑھا دیتا ہے۔ یہاں کچھ مثالیں ہیں جو آپ کرنا چاہتے ہیں:

ہر نوڈ پر ایک مخصوص فیلڈ میں تمام اقدار کو شامل کریں۔

ہر نوڈ سے فیلڈ میں تمام منفرد تاروں کی فہرست حاصل کریں۔

نوڈ کی کلید کو کچھ فہرست میں شامل کریں اگر پہلے ملاحظہ کیے گئے نوڈس میں سے کسی کی بھی کسی فیلڈ میں قدر زیادہ نہیں ہے۔

ان تمام ٹراورسلز میں، ٹراورسل کے دوران کہیں بھی نتائج جمع کرنے کے قابل ہونا بہت آسان ہے۔ فنکشنز کے ساتھ ایسا کرنا ممکن ہے، لیکن جنریٹر اسے آسان بناتے ہیں۔ ہم نے باب 5میں جنریٹرز متعارف کرائے ہیں، اور چونکہ درخت ان ڈھانچوں کے ساتھ بہت سی مماثلت رکھتے ہیں، اس لیے وہ درختوں کو عبور کرنے کے لیے بہت مفید ہیں۔

ہم ان کوتاہیوں کو ٹراورس میتھڈ، ()traverse_recکے ایک بار بار آنے والے جنریٹر ورژن میں دور کرتے ہیں، جو فہرست 6-8میں دکھایا گیا ہے۔ یہ ورژن کوڈ میں کچھ پیچیدگی کا اضافہ کرتا ہے لیکن ٹراورسل کا استعمال بہت آسان بنا دیتا ہے۔ سب سے پہلے، ہم ایک پیرامیٹر، traverse_rec، ()traverseTypeطریقہ میں شامل کرتے ہیں تاکہ ہمیں تین الگ الگ ٹراورس روٹینز کی ضرورت نہ ہو۔ پہلا اگر بیان اس بات کی تصدیق کرتا ہے کہ یہ پیرامیٹر تین معاون ترتیبوں میں سے ایک ہے: پری، ان، اور پوسٹ۔ اگر نہیں، تو یہ ایک استثناء پیدا کرتا ہے۔ بصورت دیگر، یہ روٹ نوڈ سے شروع ہونے والے ریکسریو پرائیویٹ طریقہ، ()traverse__کا آغاز کرتا ہے، بالکل اسی طرح جیسے ()inOrderTraverseکرتا ہے۔

()traverse__طریقہ کو کال کرنے میں ایک اہم لیکن لطیف نکتہ نوٹ کرنا ہے ۔ عوامی ()traverse_recطریقہ نجی ()traverse__طریقہ کو کال کرنے کا نتیجہ واپس کرتا ہے اور اسے صرف ذیلی روٹین کے طور پر نہیں کہتا ہے۔ وجہ یہ ہے کہ ()traverseطریقہ خود جنریٹر نہیں ہے۔ اس میں کوئی پیداواری بیانات نہیں ہیں۔ اسے ()traverse__پر کال کے ذریعہ تیار کردہ ایٹریٹر کو واپس کرنا ہوگا ، جسے ()traverse_recکالر نوڈس پر اعادہ کرنے کے لئے استعمال کرے گا۔

()traverse__طریقہ کے اندر ، اگر بیانات کا ایک سلسلہ ہے ۔ پہلا بیس کیس کی جانچ کرتا ہے۔ اگر نوڈ کوئی نہیں ہے ، تو یہ ایک خالی درخت (یا ذیلی درخت) ہے۔ یہ اس بات کی نشاندہی کرنے کے لئے واپس آتا ہے کہ تکرار کرنے والا اختتام کو پہنچ گیا ہے (جسے ازگر StopIterationاستثناء میں تبدیل کرتا ہے)۔ اگلا اگر بیان چیک کرتا ہے کہ آیا ٹراورسل قسم پہلے سے آرڈر ہے، اور اگر یہ ہے، تو اس سے نوڈ کی کلید اور ڈیٹا حاصل ہوتا ہے۔ یاد رکھیں کہ اٹیریٹر کو اس وقت روک دیا جائے گا جب کہ کنٹرول واپس اس کے کالر کے پاس جاتا ہے۔ اسی جگہ نوڈ کو "وزٹ کیا جائے گا۔" پروسیسنگ مکمل ہونے کے بعد، کالر کا لوپ اگلا نوڈ حاصل کرنے کے لیے اس تکرار کرنے والے کو طلب کرتا ہے۔ تکرار کنندہ تمام سیاق و سباق کو یاد رکھتے ہوئے، پیداوار کے بیان کے فوراً بعد دوبارہ کارروائی شروع کرتا ہے۔

جب تکرار کرنے والا دوبارہ شروع ہوتا ہے (یا اگر آرڈر پری آرڈر کے علاوہ کچھ تھا) تو اگلا مرحلہ لوپ کے لیے ہوتا ہے۔ یہ بائیں سب ٹری کے ٹراورسل کو انجام دینے کے لیے ایک بار بار چلنے والا جنریٹر ہے۔
یہ اسی traverseTypeکا استعمال کرتے ہوئے نوڈ کے لیفٹ چائلڈ پر ()traverse__طریقہ کو کال کرتا ہے۔
یہ اس ذیلی درخت میں نوڈس پر کارروائی کرنے کے لیے اپنا ایٹریٹر بناتا ہے۔ جیسا کہ نوڈس حاصل کیے جاتے ہیں

واپس کلید کے طور پر، ڈیٹا کے جوڑے، یہ اعلیٰ سطحی تکرار کرنے والا انہیں واپس اپنے کالر کو دیتا ہے۔ یہ لوپ کنسٹرکشن ایٹریٹروں کا ایک نیسٹڈ اسٹیک تیار کرتا ہے، جیسا کہ شکل 12-8میں دکھایا گیا ()_o_travکے نیسٹڈ انوکیشنز کی طرح ہے ۔ جب ہر تکرار کرنے والا اپنے کام کے اختتام پر واپس آتا ہے، تو یہ ایک StopIterationپیدا کرتا ہے۔ منسلک تکرار کرنے والا ہر استثنا کو پکڑتا ہے، لہذا مختلف سطحیں ایک دوسرے میں مداخلت نہیں کرتی ہیں۔

فہرست سازی 6-8ٹراورسل کے لیے تکراری جنریٹر

```
#بائنری سرچ ٹری کلاس                                کلاس BinaryTreeIterable(آبجیکٹ):
...
    #پری، ان، یا پوسٹ آرڈر میں بار بار درخت کو عبور کریں۔   def traverse_rec(self, traverseType="in"):  اگر traverseType in [
                                                                'پری'، 'ان'، 'پوسٹ']: خود واپسی
    #تصدیق کی قسم ایک قبول شدہ قدر ہے اور #درخت پر چلنے کے لیے جنریٹر کا
    استعمال کریں #حاصل کرنے والے (کلید، ڈیٹا) جوڑے #جڑ سے شروع

                                                                "
                                                    + str(traverseType))  ValueErrorبڑھائیں ("نامعلوم ٹراورسل قسم:
    #بار بار چلنے والا جنریٹر عبور کرنے کے لیے #ذیلی درخت کی جڑیں نوڈ پر پری، ان، یا #   def __traverse(self, node, traverseType):  اگر نوڈ کوئی
    پوسٹ آرڈر میں                                                                        نہیں ہے:
    #اگر ذیلی درخت خالی ہے تو #ٹراورسل کیا جاتا ہے۔
                                                                                        واپسی
    #پری آرڈر کے لیے، موجودہ #نوڈ کو پہلے "پری" کریں۔ if traverseType == "pre":
                                                                                        پیداوار (node.key، node.data)
                                                                                        چائلڈ کی کے لیے ، خود میں چائلڈ ڈیٹا
    #اس کے نوڈس حاصل کرنا                                  بائیں ذیلی درخت کو عبور کریں۔ node.leftChild، traverseType): #
    #اگر ترتیب میں ہے، تو اب موجودہ #نوڈ حاصل کریں۔          traverseType == "in":  اگر yield (childKey، childData)
                                                                                        "ان": پیداوار (node.key، node.data)
                                                                                        چائلڈ کی کے لیے ، خود میں چائلڈ ڈیٹا
    #نوڈ حاصل کریں                                         اگر پوسٹ آرڈر کریں تو باقی سب کے بعد موجودہ node.rightChild, traverseType):# traverse right subtree # yielding its nodes #
                                                                                        پیداوار (چائلڈ کی، چائلڈ ڈیٹا)
                                                                                        traverseType == "post":  اگر
                                                                                        پیداوار (node.key، node.data)
```

باقی ()__traverseطریقہ سیدھا ہے۔ بائیں سب ٹری میں تمام نوڈس پر لوپ ختم کرنے کے بعد، اگلا اگر اسٹیٹمنٹ ان آرڈر ٹراورسل قسم کی جانچ کرتا ہے اور نوڈ کی کلید اور ڈیٹا حاصل کرتا ہے، اگر یہ ترتیب ہے۔ نوڈ کو ان آرڈر ٹراورسل کے لیے بائیں اور دائیں ذیلی درختوں کے درمیان پروسیس کیا جاتا ہے۔ اس کے بعد، دائیں سب ٹری کو اس کے اپنے لوپ میں پروسیس کیا جاتا ہے، جس سے ہر وزٹ کیے گئے نوڈس واپس اس کے کالر کو ملتے ہیں۔ صحیح ذیلی درخت کے مکمل ہونے کے بعد، پوسٹ آرڈر ٹراورسل کے لیے ایک چیک اس بات کا تعین کرتا ہے کہ آیا اس مرحلے پر نوڈ کو حاصل کیا جانا چاہیے یا نہیں۔ اس کے بعد، ()__traverseجنریٹر ہو جاتا ہے، اس کے کالر کا لوپ ختم ہوتا ہے۔

جنریٹر کو موثر بنانا

تکراری جنریٹر میں ساختی سادگی کا فائدہ ہے۔ بیس کیسز اور بار بار آنے والی کالیں درخت کے نوڈ اور چائلڈ سٹرکچر کی پیروی کرتی ہیں۔ پروٹو ٹائپ تیار کرنا اور اس ڈھانچے سے قدرتی طور پر اس کے صحیح رویے کے بہاؤ کو ثابت کرنا۔

جنریٹر، تاہم، عملدرآمد میں کچھ ناکارہ ہے. ()traverse__طریقہ کی ہر درخواست میں دو لوپس شامل ہوتے ہیں: ایک بائیں کے لیے اور ایک دائیں بچے کے لیے۔ ان میں سے ہر ایک لوپ ایک نیا ایٹریٹر بناتا ہے تاکہ آئٹمز کو ان کے ذیلی درختوں سے واپس حاصل کرنے کے لیے ()traverse__طریقہ کی اس درخواست کے ذریعے تخلیق کیا گیا ہو۔ تکرار کرنے والوں کی یہ تہہ جڑ سے نیچے ہر لیف نوڈ تک پھیلی ہوئی ہے۔

درخت میں Nاشیاء کو عبور کرنے میں O (N) Oوقت لگنا چاہئے، لیکن ہر پتے تک جڑ سے نیچے تک تکرار کرنے والوں کا ایک ڈھیر بنانا پیچیدگی کو بڑھاتا ہے جو پتوں کی گہرائی کے متناسب ہے۔ پتے O(log N)کی گہرائی میں ہیں، بہترین صورت میں۔ اس کا مطلب ہے کہ Nاشیاء کے مجموعی طور پر گزرنے میں O(N×log N)وقت لگے گا۔

O(N)وقت حاصل کرنے کے لیے، آپ کو باب 6کے آخر میں زیر بحث طریقہ کو لاگو کرنے کی ضرورت ہے اور کارروائی کی جا رہی اشیاء کو رکھنے کے لیے اسٹیک کا استعمال کرنا ہوگا۔ آئٹمز میں نوڈ کے ڈھانچے اور (کلید، ڈیٹا) جوڑے شامل ہوتے ہیں جو نوڈس پر محفوظ کیے جاتے ہیں تاکہ ایک خاص ترتیب میں گزرے جائیں۔
7-8کی فہرست کوڈ دکھاتا ہے۔

غیر تکراری طریقہ تکراری نقطہ نظر کے دو حصوں کو ایک واحد ()traverseطریقہ میں جوڑتا ہے۔ ٹراورسل قسم کی درستگی کے لیے وہی چیک شروع میں ہوتا ہے۔ اگلا مرحلہ باب LinkStack) 5ماڈیول میں بیان کردہ) سے منسلک فہرست پر بنائے گئے Stackکلاس کا استعمال کرتے ہوئے ایک اسٹیک بناتا ہے۔

ابتدائی طور پر، طریقہ درخت کے جڑ نوڈ کو اسٹیک پر دھکیلتا ہے۔ اس کا مطلب ہے کہ باقی کام پورے درخت کو جڑ سے شروع کرنا ہے۔ اس کے بعد آنے والا whileلوپ باقی کام کے ذریعے اس وقت تک کام کرتا ہے جب تک کہ اسٹیک خالی نہ ہو جائے ۔

جب لوپ سے گزرتے ہیں تو ، اسٹیک کی سب سے اوپر والی چیز پاپ آف ہوجاتی ہے۔ اسٹیک پر تین قسم کی اشیاء ہوسکتی ہیں: ایک نوڈ آئٹم، ایک (کلید، ڈیٹا) ٹوپل، یا کوئی نہیں۔ مؤخر الذکر اس صورت میں ہوتا ہے جب درخت خالی ہو اور جب وہ لیف نوڈس پر کارروائی کرتا ہے (اور ان کے بچوں کو کوئی نہیں ہوتا ہے)۔

اگر اسٹیک کا اوپری حصہ نوڈ آئٹم ہے تو، ()traverseطریقہ اس بات کا تعین کرتا ہے کہ درخواست کردہ ٹراورسل آرڈر کی بنیاد پر نوڈ کے ڈیٹا اور اس کے بچوں کو کیسے پروسیس کیا جائے۔ یہ آئٹمز کو اسٹیک پر دھکیلتا ہے تاکہ اس کے بعد کے گزرنے والے گزرگاہوں پر عمل کیا جا سکے ۔ چونکہ آئٹمز اسٹیک سے الٹے آرڈر میں پاپ ہو جائیں گے جس طرح سے انہیں اس پر دھکیل دیا گیا تھا، یہ پوسٹ آرڈر ٹراورسل کے کیس کو سنبھالنے سے شروع ہوتا ہے۔

پوسٹ آرڈر میں، پہلا آئٹم جو دھکیلا گیا ہے وہ نوڈز (کلید، ڈیٹا) ٹیپل ہے۔ چونکہ اسے پہلے دھکیل دیا گیا ہے، اس پر مجموعی طور پر آخری کارروائی ہوگی۔ اگلی چیز جو دھکیل دی گئی وہ نوڈ کا دائیں بچہ ہے۔ پوسٹ آرڈر میں، یہ نوڈ کے ڈیٹا پر کارروائی کرنے سے پہلے گزر جاتا ہے۔ دوسرے آرڈرز کے لیے، صحیح بچہ ہمیشہ آخری نوڈ ہوتا ہے۔

دائیں بچے کو دبانے کے بعد، اگلا اگر بیان چیک کرتا ہے کہ آیا ان آرڈر ٹرا ورسل کی درخواست کی گئی تھی۔ اگر ایسا ہے تو، یہ دو چائلڈ نوڈس کے درمیان پروسیسڈ ہونے کے لیے اسٹیک پر نوڈز (کی، ڈیٹا) ٹیپل کو دھکیل دیتا ہے۔ اس کے بعد بائیں بچے کو پروسیسنگ کے لیے اسٹیک پر دھکیلنا ہے۔

فہرست سازی 7-8غیر تکراری ٹراورس() جنریٹر

LinkStackدرآمد سے *

#بائنری سرچ ٹری کلاس — کلاس BeerThcraeSyraniB(آبجیکٹ):

#درخت کو عبور کرنے کے لیے #غیر تکراری جنریٹر پہلے، اندر یا پوسٹ آرڈر میں — def traverse (self, traverseType='in'): اگر traverseType ['pre', 'in', 'post'] میں نہیں ہے : ValueError کو بڑھاو

#تصدیق کریں کہ ٹراورسل قسم ایک #قبول شدہ قدر ہے۔

"نامعلوم ٹراورسل قسم: " + str(traverseType))

#ایک اسٹیک بنائیں — stack = Stack() stack.push(self.__root)

#روٹ نوڈ کو اسٹیک میں رکھیں

#جبکہ اسٹیک میں کام ہے۔ — جبکہ stack.isEmpty():

#اگلا آئٹم حاصل کریں۔ — item = stack.pop() if isinstance(item, self.__Node): # اگر یہ ٹری نوڈ ہے

if traverseType == 'post': # پوسٹ آرڈر کے لیے، اسے آخری رکھیں

stack.push((item.key, item.data))

stack.push(item.rightChild) # traverse right child if traverseType == 'in': # پری آرڈر کے لیے، آئٹم ((item.data

2nd stack.push((item.key,

stack.push(item.leftChild) # بائیں بچے کو عبور کریں اگر traverseType == 'pre': #

#پری آرڈر کے لیے، آئٹم ((item.key, item.data) 1st stack.push رکھیں

#ہر دوسری غیر کوئی چیز ایک #(کلیدی، قدر) جوڑی ہے جسے حاصل کیا جائے گا۔ — ایلیف آئٹم:

پیداوار آئٹم

آخر میں، آخری if اسٹیٹمنٹ چیک کرتا ہے کہ آیا پری آرڈر ٹراورسل کی درخواست کی گئی تھی اور پھر نوڈ کے ڈیٹا کو اسٹیک پر بائیں اور دائیں بچوں سے پہلے پروسیسنگ کے لیے دھکیلتا ہے۔
یہ وائل لوپ کے ذریعے اگلے پاس کے دوران پاپ آف ہو جائے گا ۔ یہ نوڈ آئٹم کے تمام کام کو مکمل کرتا ہے ۔

حتمی ایلیف سٹیٹمنٹ اسٹیک پر ایک غیر None آئٹم کے لیے چیک کرتا ہے، جو ایک (کلیدی، ڈیٹا) ٹیپل ہونا چاہیے۔ جب لوپ کو ایسا ٹپل مل جاتا ہے، تو یہ اسے کال کرنے والے کو واپس دیتا ہے۔ پیداوار کا بیان اس بات کو یقینی بناتا ہے کہ traverse()طریقہ جنریٹر بن جائے، فنکشن نہیں۔

لوپ میں Noneقدروں کا کوئی واضح ہینڈلنگ نہیں ہے جو خالی روٹ اور چائلڈ لنکس کے لیے اسٹیک پر دھکیل دیا جاتا ہے۔ وجہ یہ ہے کہ ان کے لیے کرنے کے لیے کچھ نہیں ہے: بس انہیں اسٹیک سے باہر کریں اور باقی کام پر جاری رکھیں۔

اسٹیک کا استعمال کرتے ہوئے، آپ نے اب ایک O(N) جنریٹر بنایا ہے۔ درخت کے ہر نوڈ کو بالکل ایک بار دیکھا جاتا ہے، اسٹیک پر دھکیل دیا جاتا ہے، اور بعد میں پاپ آف ہوجاتا ہے۔ اس کے کلیدی ڈیٹا کے جوڑے اور چائلڈ لنکس کو بھی دھکیل دیا جاتا ہے اور بالکل ایک بار پاپ آف کیا جاتا ہے۔ نوڈ وزٹ اور چائلڈ لنکس کی ترتیب درخواست کردہ ٹراورسل آرڈرنگ کی پیروی کرتی ہے۔
اسٹیک کا استعمال کرتے ہوئے اور احتیاط سے ریورس کرنا

اس پر ڈالی گئی اشیاء کوڈ کو سمجھنے کے لیے قدرے پیچیدہ بناتی ہیں لیکن کارکردگی کو بہتر بناتی ہیں۔

ٹراورسنگ کے لیے جنریٹر کا استعمال

جنریٹر اپروچ (دونوں تکراری اور اسٹیک پر مبنی) کالر کے لوپس کو آسان بناتا ہے۔ مثال کے طور پر، اگر آپ کسی درخت میں موجود تمام اشیاء کو جمع کرنا چاہتے ہیں جن کا ڈیٹا اوسط ڈیٹا ویلیو سے کم ہے، تو آپ دو لوپس استعمال کر سکتے ہیں:

```
کل، شمار ،0 =کلید کے لیے ('pre')random_tree.traverse ،0میں ڈیٹا :

    کل =+ڈیٹا
    شمار 1 =+

اوسط =کل / گنتی [] = below_averageکلید کے لیے ، ڈیٹا :('in')random_tree.traverse اگر ڈیٹا =<اوسط: below_average.append((key, data))
```

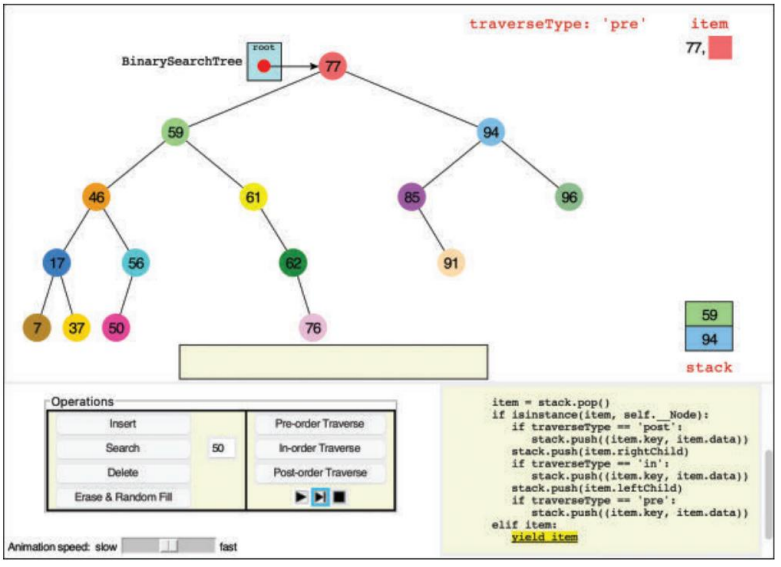
پہلا لوپ random_treeمیں آئٹمز کی تعداد کو شمار کرتا ہے اور ان کے ڈیٹا کی قدروں کو جمع کرتا ہے۔ دوسرا لوپ ان تمام آئٹمز کو تلاش کرتا ہے جن کا ڈیٹا اوسط سے کم ہے اور کلید اور ڈیٹا کے جوڑے کو نیچے کی اوسط فہرست میں شامل کرتا ہے۔ چونکہ دوسرا لوپ ان آرڈر میں ہوتا ہے، نیچے_اوسط میں کیز صعودی ترتیب میں ہوتی ہیں۔ کسی فنکشن باڈی سے باہر کچھ عالمی (یا غیر مقامی) متغیرات کی وضاحت کیے بغیر نتائج جمع کرنے والے متغیرات کا حوالہ دینے کے قابل ہونا—کل، شمار، اور نیچے_اوسط —جنریٹر کا استعمال ٹراورسل کے لیے بہت آسان بناتا ہے۔

بائنری سرچ ٹری ویژولائزیشن ٹول کے ساتھ ٹریورسنگ آپ کو جنریٹرز کا استعمال کرتے ہوئے ٹراورسل کی تفصیلات دریافت کرنے کی اجازت دیتا ہے۔ آپ پری آرڈر ٹراورس، ان آرڈر ٹریورس، یا پوسٹ آرڈر ٹریورس بٹن کو منتخب کر کے تین قسم کے ٹراورسلز میں سے کوئی بھی لانچ کر سکتے ہیں۔ ہر صورت میں، ٹول فارم کا ایک سادہ لوپ چلاتا ہے۔

```
کلید کے لیے ، ٹری ٹراورس میں ڈیٹا ("پری"): پرنٹ (کلید)
```

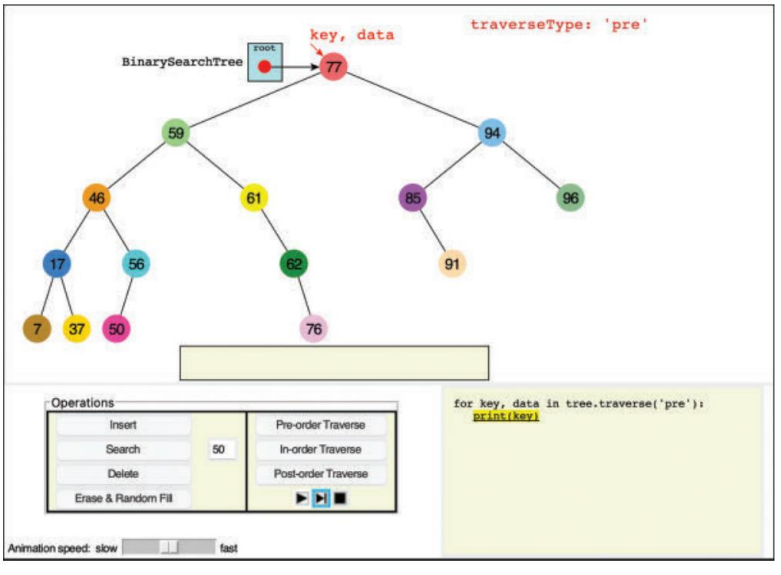
تفصیلات دیکھنے کے لیے، اسٹیپ بٹن کا استعمال کریں (بٹن کو منتخب کرتے وقت آپ شفٹ کی کو دبا کر اسٹیپ موڈ میں آپریشن شروع کر سکتے ہیں)۔ کوڈ ونڈو میں، آپ کو پہلے مختصر ٹراورسل لوپ نظر آتا ہے۔ مثال traverse()طریقہ کو کال کرتی ہے تاکہ ایک لوپ میں تمام کیز اور ڈیٹا کو دیکھیں جیسے کہ پری جیسے آرڈرز میں سے ایک۔

شکل 14-8پری آرڈر ٹراورسل کے آغاز کے قریب ایک سنیپ شاٹ دکھاتا ہے۔ traverse()طریقہ کا کوڈ نیچے دائیں طرف ظاہر ہوتا ہے۔ کوڈ کے اوپر درخت کے دائیں طرف، اسٹیک دکھایا گیا ہے۔ 59اور 94کیز پر مشتمل نوڈس اسٹیک پر ہیں۔ اسٹیک کے اوپری حصے کو پہلے ہی پاپ آف کر دیا گیا تھا اور آئٹم لیبل کے نیچے اوپر دائیں طرف منتقل کر دیا گیا تھا ۔ یہ کلید، 77کوما کے ساتھ اس کے رنگین مستطیل سے الگ کرتا ہے (کلید، ڈیٹا) ٹیپل کی نمائندگی کرنے کے لیے جسے اسٹیک پر دھکیل دیا گیا تھا۔ پیداوار کے بیان پر روشنی ڈالی گئی ہے، جس سے ظاہر ہوتا ہے کہ traverse() iteratorکلید اور ڈیٹا کالر کو واپس دینے والا ہے۔ لوپ جسے traverse()کہا جاتا ہے کوڈ ڈسپلے کو بند کر دیا گیا ہے لیکن اگلے مرحلے پر دکھایا جائے گا۔



شکل iterator 8-14 traverse()کا استعمال کرتے ہوئے پری آرڈر میں درخت کو عبور کرنا

جب کنٹرول کالنگ لوپ پر واپس آجاتا ہے، iterator traverse()کوڈ ونڈو سے غائب ہوجاتا ہے اور اسی طرح اسٹیک بھی، جیسا کہ شکل 8-15میں دکھایا گیا ہے۔ کلید اور ڈیٹا متغیر اب 77اور روٹ نوڈ کے ڈیٹا کے پابند ہیں ۔ پرنٹ اسٹیٹمنٹ کو ہائی لائٹ کیا گیا ہے کیونکہ پروگرام درخت کے نیچے والے آؤٹ پٹ باکس میں کلید کو پرنٹ کرنے والا ہے ۔ اگلا مرحلہ ظاہر کرتا ہے کہ کلید 77کو آؤٹ پٹ باکس میں کاپی کیا جا رہا ہے۔



تصویر 8-15لوپ جو ٹراورس() ایٹریٹر کو کال کرتا ہے۔

پرنٹ کرنے کے بعد، ٹری. ٹریورس ('پری') لوپ میں ڈیٹا فار کلید پر کنٹرول واپس آجاتا ہے ۔ یہ traverse() iteratorکو کوڈ ڈسپلے پر واپس دھکیلتا ہے، اس کے ساتھ اس کے اسٹیک بھی شکل 8-14سے ملتا جلتا ہے۔ iteratorمیں while loopسے پتہ چلتا ہے کہ اسٹیک خالی نہیں ہے، اس لیے یہ ٹاپ آئٹم کو پاپ کرتا ہے ۔ وہ آئٹم نوڈ 59ہے، نوڈ 77کا بائیں چائلڈ۔ یہ عمل نوڈ 59کے بچوں اور نوڈ کی کلید، اسٹیک پر ڈیٹا پیئر کو دبانے سے دہرایا جاتا ہے۔ اگلے لوپ کی تکرار پر، وہ ٹوپل اسٹیک سے باہر نکل جاتا ہے، اور اسے پرنٹ لوپ پر واپس لایا جاتا ہے۔

تکرار کرنے والوں کی پروسیسنگ بیان کرنے کے لیے پیچیدہ ہے، اور ویژولائزیشن ٹول تحریری تفصیل کو پڑھنے کے بجائے مختلف سطحوں اور مراحل کی پیروی کرنا آسان بنا دیتا ہے۔ متعدد نوڈس کی پروسیسنگ کے ذریعے قدم اٹھانے کی کوشش کریں، بشمول جب تکرار کنندہ کسی لیف نوڈ تک پہنچتا ہے اور اسٹیک پر Noneکو دھکیلتا ہے۔ اسٹیک تکرار کرنے والے کو نوڈس پر واپس جانے کے لیے رہنمائی کرتا ہے جن پر کارروائی ہونا باقی ہے۔

ٹراورسل آرڈر

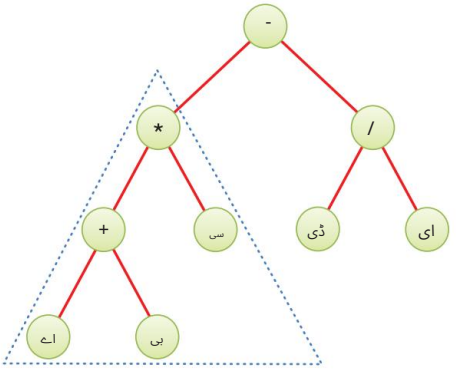
تین ٹراورسل آرڈرز کا کیا فائدہ؟ ایک فائدہ یہ ہے کہ ان آرڈر ٹراورسل بائنری تلاش کے درختوں میں چابیاں کے صعودی ترتیب کی ضمانت دیتا ہے۔ آرڈر سے پہلے اور بعد کے سفر کے لیے الگ الگ موٹی ویشن ہے۔ یہ بہت مفید ہیں اگر آپ ایسے پروگرام لکھ رہے ہیں جو الجبری تاثرات کو پارس یا تجزیہ کرتے ہیں۔ آئیے دیکھتے ہیں کہ ایسا کیوں ہے۔

ایک بائنری ٹری (بائنری سرچ ٹری نہیں) کو ایک الجبری اظہار کی نمائندگی کرنے کے لیے استعمال کیا جا سکتا ہے جس میں بائنری ریاضی کے آپریٹرز جیسے ، /، –، +اور *شامل ہوتے ہیں۔ روٹ نوڈ اور ہر نون لیف نوڈ میں ایک آپریٹر ہوتا ہے۔ لیف نوڈس یا تو متغیر نام (جیسے A، B، یا C) یا نمبر رکھتے ہیں۔ ہر ذیلی درخت ایک درست الجبری اظہار ہے۔

مثال کے طور پر، شکل 8-16میں دکھایا گیا بائنری درخت الجبری اظہار کی نمائندگی کرتا ہے۔

(A+B) * C □ D/E

اسے انفکس نوٹیشن کہتے ہیں۔ یہ وہ اشارے ہے جو عام طور پر الجبرا میں استعمال ہوتا ہے۔ (انفکس اور پوسٹ فکس کے بارے میں مزید جاننے کے لیے باب 4میں سیکشن "آرتھمیٹک ایکسپریشنز کو پارس کرنا" دیکھیں۔) درخت کو ترتیب سے ٹریور کرنے سے ترتیب میں درست ترتیب A+B*C–D/Eپیدا ہوتی ہے، لیکن آپ کو داخل کرنے کی ضرورت ہے۔ آپریشن کی متوقع ترتیب حاصل کرنے کے لیے اپنے آپ کو قوسین کرتا ہے۔ نوٹ کریں کہ ذیلی درخت اپنے ذیلی تاثرات بناتے ہیں جیسے (A+B) * Cشکل میں بیان کیا گیا ہے۔



تصویر 8-16بائنری ٹری ایک الجبری اظہار کی نمائندگی کرتا ہے۔

اس سب کا پری آرڈر اور پوسٹ آرڈر ٹراورسلز سے کیا تعلق ہے؟ آئیے دیکھتے ہیں کہ پری آرڈر ٹراورسل انجام دینے میں کیا شامل ہے۔ اقدامات ہیں۔

.1نوڈ پر جائیں۔

.2نوڈ کے بائیں سب ٹری کو عبور کرنے کے لیے خود کو کال کریں۔

.3نوڈ کے دائیں ذیلی درخت کو عبور کرنے کے لیے خود کو کال کریں۔

پری آرڈر کا استعمال کرتے ہوئے شکل 16-8میں دکھائے گئے درخت کو عبور کرنا اور نوڈ کی قدر کو پرنٹ کرنے سے اظہار پیدا ہوگا۔

*+ABC/DE

اسے پریفکس نوٹیشن کہتے ہیں۔ پہلی بار جب آپ اس کا سامنا کرتے ہیں تو یہ عجیب لگ سکتا ہے، لیکن اس کی اچھی خصوصیات میں سے ایک یہ ہے کہ قوسین کی کبھی ضرورت نہیں ہوتی ہے۔ اظہار ان کے بغیر غیر مبہم ہے۔ بائیں طرف سے شروع کرتے ہوئے، ہر آپریٹر کو اظہار میں اس کے دائیں جانب اگلی دو چیزوں پر لاگو کیا جاتا ہے، جسے آپرینڈز کہتے ہیں۔ پہلے آپریٹر کے لیے، ،-یہ دو چیزیں پروڈکٹ [illegible] میں اظہار A+B ہوگا۔ آخر میں، چوتھا آپریٹر، /، دو متغیرات Dاور Eپر کام کرتا ہے۔

تیسری قسم کا ٹراورسل، پوسٹ آرڈر، تین مراحل پر مشتمل ہے جو ایک اور طریقے سے ترتیب دیے گئے ہیں:

.1نوڈ کے بائیں سب ٹری کو عبور کرنے کے لیے خود کو کال کریں۔

.2نوڈ کے دائیں ذیلی درخت کو عبور کرنے کے لیے خود کو کال کریں۔

.3نوڈ پر جائیں۔

شکل 16-8میں درخت کے لیے، پوسٹ آرڈر ٹراورسل کے ساتھ نوڈس کا دورہ اظہار پیدا کرتا ہے

AB+C*DE/-

اسے پوسٹ فکس نوٹیشن کہتے ہیں۔ اس کا مطلب ہے "اظہار میں آخری آپریٹر کو اس کے بائیں طرف فوری طور پر دو چیزوں پر لگائیں۔" پہلی چیز AB+C* ہے، اور دوسری چیز DE/ ہے۔ پہلی چیز کا تجزیہ کرتے ہوئے، AB+C*، اس کا مطلب ظاہر کرتا ہے کہ "آپریٹر کو فوری طور پر اس کے بائیں جانب دو چیزوں پر لگائیں، AB+ اور C۔" اس اظہار کی پہلی چیز کا تجزیہ کرنے سے، AB+، اس کا مطلب ظاہر کرتا ہے کہ "+ آپریٹر کو اس کے بائیں جانب دو چیزوں پر فوری طور پر لاگو کریں، A اور B۔" ابتدائی طور پر یہ دیکھنا مشکل ہے، لیکن "چیزیں" ہمیشہ تین قسموں میں سے ایک ہوتی ہیں: ایک واحد متغیر، ایک عدد، یا بائنری آپریٹر پر ختم ہونے والا اظہار۔

پوسٹ فکس ایکسپریشن کے معنی پر کارروائی کرنے کے لیے، آپ دائیں طرف کے آخری حرف سے شروع کرتے ہیں اور اس کی مندرجہ ذیل تشریح کرتے ہیں۔ اگر یہ بائنری آپریٹر ہے، تو آپ اس عمل کو دہرائیں گے۔

اس کے بائیں جانب دو ذیلی اظہارات کی تشریح کریں، جو آپریٹر کے آپرینڈ بن جاتے ہیں۔ اگر یہ ایک حرف ہے، تو یہ ایک سادہ متغیر ہے، اور اگر یہ ایک عدد ہے، تو یہ ایک مستقل ہے۔ متغیر ایبلز اور نمبرز دونوں کے لیے، آپ انہیں اظہار کے دائیں جانب سے "پاپ" کرتے ہیں اور انہیں انکلوژنگ ایکسپریشن کے عمل میں واپس کردیتے ہیں۔

ہم یہاں تفصیلات نہیں دکھاتے ہیں، لیکن آپ تصویر 16-8میں پوسٹ فکس ایکسپریشن کو بطور ان پٹ استعمال کرکے آسانی سے درخت بنا سکتے ہیں۔ نقطہ نظر پوسٹ فکس ایکسپریشن کا جائزہ لینے کے مترادف ہے، جسے آپ نے باب 4میں PostfixTranslate.pyپروگرام اور اس سے متعلقہ InfixCalculator Visualizationٹول میں دیکھا ہے۔ اسٹیک پر آپرینڈز کو ذخیرہ کرنے کے بجائے، تاہم، آپ پورے ذیلی درختوں کو ذخیرہ کرتے ہیں۔ آپ پوسٹ فکس سٹرنگ کے ساتھ بائیں سے دائیں پڑھتے ہیں جیسا کہ آپ نے PostfixEvaluate()طریقہ میں کیا تھا۔ یہاں وہ اقدامات ہیں جب آپ کو آپرینڈ (متغیر یا نمبر) کا سامنا کرنا پڑتا ہے:

1.ایک نوڈ کے ساتھ ایک درخت بنائیں جس میں آپرینڈ ہو۔

2.اس درخت کو اسٹیک پر دھکیلیں۔

جب آپ کسی آپریٹر کا سامنا کرتے ہیں تو یہ اقدامات ہیں، :O

1.دو اوپرینڈ درخت Rاور Lکو اسٹیک سے ہٹا دیں (اسٹیک کے اوپری حصے میں سب سے دائیں طرف ہے آپرینڈ، آر)۔

2.آپریٹر، Oکے ساتھ اس کی جڑ میں ایک نیا درخت Tبنائیں۔

3. Tکے صحیح بچے کے طور پر Rکو جوڑیں۔

4. Tکے بائیں بچے کے طور پر Lکو جوڑیں۔

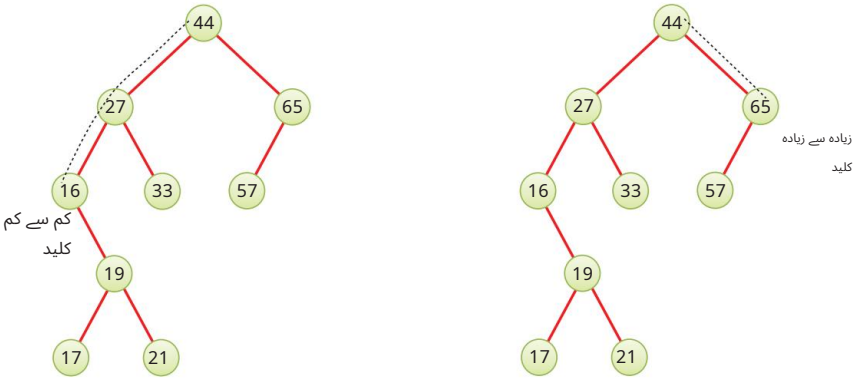
5.نتیجے میں آنے والے درخت، Tکو اسٹیک پر واپس دھکیلیں۔

جب آپ پوسٹ فکس سٹرنگ کی جانچ کر لیتے ہیں، تو آپ ایک باقی آئٹم کو اسٹیک سے باہر کر دیتے ہیں۔ کسی حد تک حیرت انگیز طور پر، یہ آئٹم ایک مکمل درخت ہے جو الجبری اظہار کو ظاہر کرتا ہے۔ اس کے بعد آپ ہمارے بیان کردہ تین ترتیبوں میں سے کسی ایک میں درخت کو عبور کرتے ہوئے اصل پوسٹ فکس اشارے (اور پوسٹ فکس اظہار کی بازیافت) کے سابقہ اور انفکس کی نمائندگی دیکھ سکتے ہیں۔ ہم اس عمل کے نفاذ کو ایک مشق کے طور پر چھوڑتے ہیں۔

کم از کم اور زیادہ سے زیادہ کلیدی اقدار تلاش کرنا

اتفاق سے، آپ کو نوٹ کرنا چاہیے کہ بائنری سرچ ٹری میں کم از کم اور زیادہ سے زیادہ کلیدی اقدار کو تلاش کرنا کتنا آسان ہے۔ درحقیقت، یہ عمل اتنا آسان ہے کہ ہم اسے ویژولائزیشن ٹول میں بطور آپشن شامل نہیں کرتے ہیں۔ پھر بھی، یہ سمجھنا ضروری ہے کہ یہ کیسے کام کرتا ہے۔

کم از کم، جڑ کے بائیں بچے پر جائیں؛ پھر اس بچے کے بائیں بچے کے پاس جائیں، اور اسی طرح، جب تک کہ آپ اس نوڈ پر نہ آجائیں جس میں کوئی بچہ نہیں ہے۔ یہ نوڈ کم از کم ہے۔
اسی طرح، زیادہ سے زیادہ، جڑ سے شروع کریں اور صحیح چائلڈ لنکس کی پیروی کریں جب تک کہ وہ ختم نہ ہوں۔ یہ درخت میں زیادہ سے زیادہ کلید ہوگی، جیسا کہ شکل 17-8میں دکھایا گیا ہے۔



تصویر 17-8بائنری سرچ ٹری کی کم از کم اور زیادہ سے زیادہ کلیدی اقدار

یہاں کچھ کوڈ ہے جو کم از کم نوڈ کا ڈیٹا اور کلیدی اقدار واپس کرتا ہے:

```
#کم از کم کلید کے نوڈ کا ڈیٹا تلاش کریں اور واپس کریں۔
def minNode(self):
#اگر درخت خالی ہے تو استثنیٰ بڑھائیں استثنیٰ ("خالی درخت میں کوئی کم از کم
نوڈ نہیں")
اگر :self.isEmpty()
#جڑ سے شروع کریں۔
node = self.__root
#جبکہ نوڈ میں بائیں بچہ ہے،
جبکہ :node.leftChild
#بچے کے بائیں حوالہ کی پیروی کریں۔
node = node.leftChild
واپسی # (node.key، node.data)واپسی حتمی نوڈ کلید اور ڈیٹا
```

زیادہ سے زیادہ تلاش کرنا اسی طرح ہے؛ صرف بائیں بچے کے لئے دائیں کو تبدیل کریں۔ آپ نوڈس کو حذف کرنے کے بارے میں اگلے حصے میں کم از کم قدر تلاش کرنے کے ایک اہم استعمال کے بارے میں سیکھتے ہیں۔

نوڈ کو حذف کرنا

نوڈ کو حذف کرنا بائنری تلاش کے درختوں کے لیے درکار سب سے پیچیدہ کام ہے۔ تاہم، حذف کرنے کے بنیادی عمل کو نظر انداز نہیں کیا جا سکتا، اور تفصیلات کا مطالعہ کردار کی تعمیر کرتا ہے۔ اگر آپ کردار سازی کے موڈ میں نہیں ہیں، تو بلا جھجھک Binary Search Trees سیکشن کی Efficiencyپر جائیں۔

آپ اس بات کی تصدیق کرتے ہوئے شروع کرتے ہیں کہ درخت خالی نہیں ہے اور پھر جس نوڈ کو آپ حذف کرنا چاہتے ہیں اسے ڈھونڈتے ہیں، اسی نقطہ نظر کو استعمال کرتے ہوئے جو آپ نے __find()اور insert()میں دیکھا تھا۔ اگر نوڈ نہیں ملا، تو آپ کا کام ہو گیا۔ جب آپ کو نوڈ اور اس کے والدین مل جاتے ہیں، تو غور کرنے کے لیے تین صورتیں ہیں:

1.جس نوڈ کو حذف کیا جانا ہے وہ ایک پتی ہے (اس کی کوئی اولاد نہیں ہے)۔

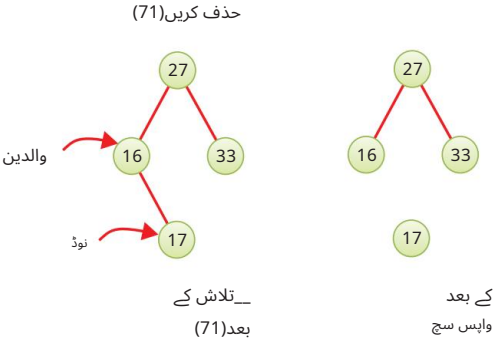
2.حذف کیے جانے والے نوڈ کا ایک بچہ ہے۔

3.حذف کیے جانے والے نوڈ کے دو بچے ہیں۔

آئیے ان تینوں صورتوں کو باری باری دیکھتے ہیں۔ پہلا آسان ہے؛ دوسرا، تقریباً اتنا ہی آسان؛ اور تیسرا، کافی پیچیدہ۔

کیس :1جس نوڈ کو حذف کیا جائے اس کا کوئی بچہ نہیں ہے۔

لیف نوڈ کو حذف کرنے کے لیے، آپ نوڈ کے والدین میں مناسب چائلڈ فیلڈ کو نوڈ کے بجائے Noneمیں تبدیل کر دیتے ہیں۔ نوڈ آبجیکٹ اب بھی موجود ہے، لیکن اب یہ درخت کا حصہ نہیں ہے، جیسا کہ شکل 18-8میں نوڈ 17کو حذف کرتے وقت دکھایا گیا ہے۔



تصویر 18-8بغیر بچوں کے نوڈ کو حذف کرنا

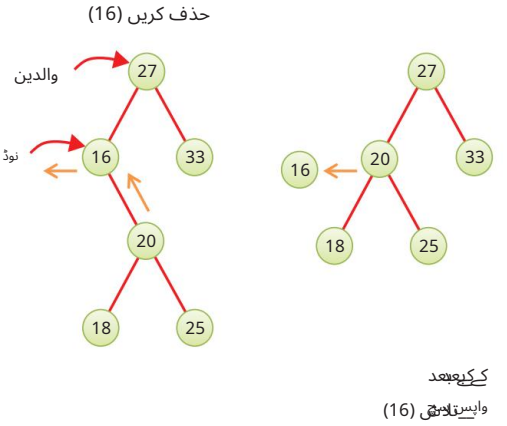
اگر آپ ازگر یا جاوا جیسی زبان استعمال کر رہے ہیں جس میں کوڑا کرکٹ جمع ہے، تو حذف شدہ نوڈ کی میموری کو دوسرے استعمال کے لیے دوبارہ حاصل کیا جائے گا (اگر آپ پروگرام میں اس کے تمام حوالوں کو ختم کر دیتے ہیں)۔ ایسی زبانوں میں جن کے لیے میموری کی واضح مختص اور ڈی ایلوکیشن کی ضرورت ہوتی ہے، حذف شدہ نوڈ کو دوبارہ استعمال کے لیے جاری کیا جانا چاہیے۔

بغیر بچوں کے نوڈ کو حذف کرنے کے لیے ویژولائزیشن ٹول کا استعمال بائنری سرچ ٹری ویژولائزیشن ٹول کا استعمال کرتے ہوئے لیف نوڈ کو حذف کرنے کی کوشش کریں۔ آپ ٹیکسٹ انٹری باکس میں نوڈ کی کلید ٹائپ کر سکتے ہیں یا اپنے پوائنٹر ڈیوائس کے ساتھ ایک لیف منتخب کر سکتے ہیں اور پھر ڈیلیٹ کو منتخب کر سکتے ہیں۔ آپ دیکھتے ہیں کہ پروگرام find()__کو اس کی کلید کے ذریعے نوڈ کو تلاش کرنے کے لیے استعمال کرتا ہے، اسے ایک عارضی متغیر میں کاپی کرتا ہے، پیرنٹ لنک کو Noneپر سیٹ کرتا ہے، اور پھر حذف شدہ کلید اور ڈیٹا کو (اس کے رنگین پس منظر کی شکل میں) "واپس" کرتا ہے۔

کیس :2حذف کیے جانے والے نوڈ میں ایک بچہ ہے۔

یہ دوسرا معاملہ بھی زیادہ مشکل نہیں ہے۔ نوڈ کے صرف دو کنارے ہیں: ایک اس کے والدین کی طرف اور دوسرا اس کے اکلوتے بچے کی طرف۔ آپ نوڈ کو اس کے والدین کو براہ راست اس کے بچے سے جوڑ کر اس ترتیب سے "کاٹ" کرنا چاہتے ہیں۔ اس عمل میں حذف شدہ نوڈ کے بچے کی طرف اشارہ کرنے کے لیے والدین (بائیں چائلڈ یا رائٹ چائلڈ یا __روٹ) میں مناسب حوالہ تبدیل کرنا شامل ہے۔

شکل 19-8میں نوڈ 16کو حذف کیا گیا ہے، جس کا صرف ایک بچہ ہے۔



تصویر 8-19ایک بچے کے ساتھ نوڈ کو حذف کرنا

نوڈ اور اس کے والدین کو تلاش کرنے کے بعد، حذف کرنے کے طریقہ کار کو صرف ایک حوالہ تبدیل کرنا ہوگا۔
حذف شدہ نوڈ، تصویر میں کلید ،16درخت سے منقطع ہو جاتا ہے (حالانکہ اس میں اب بھی اس نوڈ کے لیے چائلڈ پوائنٹر ہو سکتا ہے جس کو پروموٹ کیا گیا تھا (کلید ۔(20کوڑا اٹھانے والے یہ جاننے کے لیے کافی نفیس ہیں کہ وہ حذف کیے گئے نوڈ کا دوبارہ دعویٰ کر سکتے ہیں۔ دوسرے نوڈس کے لنکس کی پیروی کیے بغیر جن کی اب بھی ضرورت ہوسکتی ہے۔

اب آئیے بغیر بچوں کے نوڈ کو حذف کرنے کے معاملے پر واپس جائیں۔ اس صورت میں، حذف کرنے کے طریقہ کار نے والدین کے چائلڈ پوائنٹر میں سے ایک کو تبدیل کرنے کے لیے ایک تبدیلی بھی کی ہے۔
اس پوائنٹر کو Noneپر سیٹ کیا گیا تھا کیونکہ کوئی متبادل چائلڈ نوڈ نہیں تھا۔ یہ کیس 2کا سمی لار آپریشن ہے، لہذا آپ یہ کہہ کر کیس 1اور کیس 2کا ایک ساتھ علاج کر سکتے ہیں، "اگر نوڈ کو حذف کرنا ہے، ،Dکے 0یا 1بچے ہیں، تو اس کے والدین میں مناسب لنک کو بائیں بچے کے ساتھ تبدیل کریں۔ Dکا، اگر یہ خالی نہیں ہے، یا Dکا صحیح بچہ ہے۔" اگر Dسے دونوں چائلڈ لنکس کوئی نہیں ہیں، تو آپ نے کیس 1کا احاطہ کیا ہے۔ اگر Dکے چائلڈ لنکس میں سے صرف ایک ہی غیر کوئی نہیں ہے، تو مناسب بچے کو والدین کے نئے بچے کے طور پر منتخب کیا جائے گا، جس میں کیس 2کا احاطہ کیا جائے گا۔

آپ والدین کے بچے (یا ممکنہ طور پر (root__حوالہ میں واحد بچے یا Noneکو فروغ دیتے ہیں۔

ایک بچے کے ساتھ نوڈ کو حذف کرنے کے لیے ویژولائزیشن ٹول کا استعمال آئیے فرض کریں کہ آپ شکل 5-8میں درخت پر ویژولائزیشن ٹول استعمال کر رہے ہیں اور نوڈ 61کو حذف کر رہے ہیں، جس کا دائیں بچہ ہے لیکن بائیں بچہ نہیں۔ نوڈ 61پر کلک کریں اور ڈیلیٹ بٹن کو فعال کرتے ہوئے کلید ٹیکسٹ انٹری ایریا میں ظاہر ہونی چاہیے۔ بٹن کو منتخب کرنے سے (find__پر ایک اور کال شروع ہوتی ہے جو کرنٹ کی طرف نوڈ اور پیرنٹ پوائنٹ نوڈ 59کی طرف اشارہ کرنے کے ساتھ رک جاتی ہے۔

نوڈ 61کی ایک کاپی بنانے کے بعد، اینیمیشن دکھاتی ہے کہ نوڈ 59سے صحیح چائلڈ لنک نوڈ 61کے رائٹ چائلڈ، نوڈ 62پر سیٹ کیا جا رہا ہے۔ نوڈ 61کی اصل کاپی ختم ہو جاتی ہے، اور درخت کو ایڈجسٹ کیا جاتا ہے تاکہ نوڈ پر جڑی ہوئی ذیلی درخت رکھی جا سکے۔ 62اپنی نئی پوزیشن میں۔ آخر میں، نوڈ 61کی کاپی نیچے والے آؤٹ پٹ باکس میں منتقل ہو جاتی ہے۔

Machine Translated by Google

سنگل چائلڈ نوڈس کے ساتھ نئے درخت بنانے کے لیے ویژولائزیشن ٹول کا استعمال کریں اور دیکھیں کہ جب آپ انہیں حذف کرتے ہیں تو کیا ہوتا ہے۔ ذیلی درخت کو تلاش کریں جس کی جڑ حذف شدہ نوڈ کا بچہ ہے۔ اس سے کوئی فرق نہیں پڑتا ہے کہ یہ ذیلی درخت کتنا ہی پیچیدہ ہے، اسے آسانی سے اوپر منتقل کیا جاتا ہے اور حذف شدہ نوڈ کے والدین کے نئے بچے کے طور پر پلگ ان ہوتا ہے۔

نوڈ کو حذف کرنے کے لیے پائتھون کوڈ اب کم از کم کیسز 1اور 2کے کوڈ کو دیکھتے ہیں۔ 8-8کی فہرست ڈیلیٹ() طریقہ کار کے لیے کوڈ کو ظاہر کرتی ہے ، جس میں ایک دلیل لی جاتی ہے، نوڈ کو حذف کرنے کی کلید۔ یہ یا تو نوڈ کا ڈیٹا لوٹاتا ہے جسے حذف کیا گیا تھا یا کوئی نہیں، اس بات کی نشاندہی کرنے کے لیے کہ نوڈ نہیں ملا تھا۔

اس سے یہ کچھ اس طرح برتاؤ کرتا ہے جیسے کسی آئٹم کو اسٹیک سے ہٹانے یا قطار سے کسی آئٹم کو حذف کرنے کے طریقوں کی طرح۔ فرق یہ ہے کہ اعداد و شمار کے ڈھانچے میں معلوم مقام پر ہونے کے بجائے نوڈ کو درخت کے اندر پایا جانا چاہیے۔

فہرست 8-8بائنری سرچ ٹری کا ڈیلیٹ ()طریقہ

```
کلاس BinarySearchTree(آبجیکٹ): #بائنری سرچ ٹری کلاس
...
def حذف (خود، مقصد): #ایک نوڈ کو حذف کریں جس کی کلید گول سے مماثل ہو۔
node, parent = self.__find(goal) # گول اور اس کے والدین کو تلاش کریں اگر نوڈ کوئی نہیں ہے:
#اگر نوڈ پایا گیا تھا، #پھر پیرنٹ کے نیچے نوڈ #پر حذف کریں۔
خود کو واپس کریں ۔__حذف کریں(والدین، نوڈ)

def __delete(خود، والدین، نوڈ): #درخت میں مخصوص نوڈ کو حذف کریں #پیرنٹ نوڈ/ٹری میں ترمیم کرنا

حذف شدہ = node.data #وہ ڈیٹا محفوظ کریں جسے حذف کرنا ہے۔
اگر node.leftChild: #ذیلی درختوں کی تعداد کا تعین کریں۔
if node.rightChild: self.__promote_successor(node) #اگر دونوں ذیلی درخت موجود ہیں، #پھر جانشین کو #ڈیلیٹ شدہ نوڈ کو تبدیل کرنے کے لیے پروموٹ کریں۔
دوسری: #اگر صحیح بچہ نہیں ہے تو بائیں بچے کو اوپر لے جائیں۔
اگر والدین خود ہیں: #اگر والدین پورا درخت ہے self.__root = node.leftChild #اپ ڈیٹ روٹ elif parent.leftChild نوڈ ہے : #اگر نوڈ والدین کا بائیں
ہے، # parent.leftChild = node.leftChild بچہ، باقی بائیں کو اپ ڈیٹ کریں: else #دائیں بچے کو اپ ڈیٹ کریں

parent.rightChild = node.leftChild
دوسری: #کوئی بچہ نہیں چھوڑا؛ لہذا صحیح بچے کو فروغ دیں
اگر والدین خود ہیں : #اگر ماں باپ پورا درخت ہے،
self.__root = node.rightChild # update root elif parent.leftChild نوڈ ہے : #اگر نوڈ والدین کا بائیں ہے

parent.leftChild = node.rightChild # بچہ، پھر اپ ڈیٹ کریں : #بائیں چائلڈ لنک اور اپ ڈیٹ

parent.rightChild = node.rightChild # right child # حذف شدہ نوڈ کا ڈیٹا واپس کریں
واپسی کو حذف کر دیا گیا۔
```

بالکل اسی طرح جیسے داخل کرنے کے لیے، پہلا قدم حذف کرنے کے لیے نوڈ اور اس کے والدین کو تلاش کرنا ہے۔ اگر اس تلاش کو گول نوڈ نہیں ملتا ہے ، تو درخت سے حذف کرنے کے لیے کچھ نہیں ہے، اور حذف ()کوئی نہیں لوٹاتا ہے ۔ اگر حذف کرنے کے لیے نوڈ مل جاتا ہے، تو نوڈ اور اس کے والدین کو درخت میں موجود نوڈس میں ترمیم کرنے کے لیے نجی ()delete__طریقہ کار میں منتقل کیا جاتا ہے۔

()delete__طریقہ کے اندر ، پہلا قدم یہ ہے کہ حذف کیے جانے والے نوڈ ڈیٹا کا حوالہ محفوظ کیا جائے ۔ یہ مرحلہ درخت سے حوالہ جات ہٹانے کے بعد نوڈ کے ڈیٹا کی بازیافت کے قابل بناتا ہے۔ اگلا مرحلہ چیک کرتا ہے کہ نوڈ میں کتنے ذیلی درخت ہیں۔ یہ طے کرتا ہے کہ کس کیس پر کارروائی ہو رہی ہے۔ اگر بائیں اور دائیں دونوں بچے موجود ہیں، تو یہ کیس 3ہے، اور یہ حذف کرنے کو ایک اور نجی طریقہ، ()promote_successor__کے حوالے کرتا ہے ، جسے ہم تھوڑی دیر بعد بیان کرتے ہیں۔

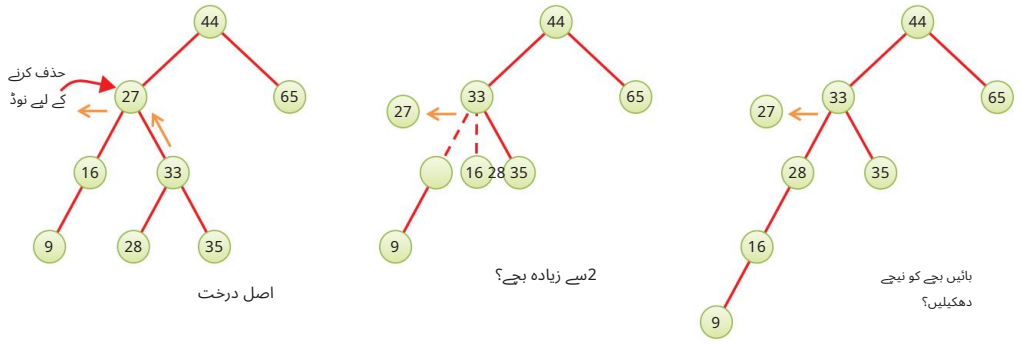
اگر حذف کرنے کے لیے نوڈ کا صرف ایک بائیں ذیلی درخت ہے، تو اگلی چیز اس کا پیرنٹ نوڈ ہے۔ اگر والدین BinarySearchTreeآبجیکٹ (self)ہے، تو حذف کرنے کے لیے نوڈ کا روٹ نوڈ ہونا چاہیے، اس لیے بائیں بچے کو روٹ نوڈ سلاٹ میں ترقی دی جاتی ہے۔ اگر پیرنٹ کا بایاں بچہ حذف کرنے کے لیے نوڈ ہے، تو نوڈ کو ہٹانے کے لیے والدین کے بائیں چائلڈ کے لنک کو نوڈ کے بائیں چائلڈ سے بدل دیا جاتا ہے۔ بصورت دیگر، نوڈ کو ہٹانے کے لیے والدین کے دائیں چائلڈ لنک کو اپ ڈیٹ کیا جاتا ہے۔

یاد رکھیں کہ حوالہ جات کے ساتھ کام کرنے سے پورے ذیلی درخت کو منتقل کرنا آسان ہوجاتا ہے۔ جب نوڈ کے لیے والدین کا حوالہ اپ ڈیٹ کیا جاتا ہے، تو ترقی پانے والا بچہ واحد نوڈ یا بہت بڑا ذیلی درخت ہوسکتا ہے۔ صرف ایک حوالہ تبدیل کرنے کی ضرورت ہے۔ اگرچہ ذیلی درخت میں بہت سے نوڈس ہو سکتے ہیں، آپ کو ان کو انفرادی اتحادی میں منتقل کرنے کے بارے میں فکر کرنے کی ضرورت نہیں ہے۔ درحقیقت، وہ صرف دوسرے نوڈس کے مقابلے میں تصوراتی طور پر مختلف پوزیشنوں میں ہونے کے معنی میں "حرکت" کرتے ہیں۔ جہاں تک پروگرام کا تعلق ہے، سب ٹری کی جڑ کے لیے صرف والدین کا حوالہ تبدیل ہوا ہے، اور میموری میں باقی مواد وہی رہتا ہے۔

()delete__طریقہ کار کی حتمی دوسری شق اس معاملے سے نمٹتی ہے جب نوڈ میں کوئی بچہ نہیں ہوتا ہے۔ چاہے نوڈ کا صحیح بچہ ہے یا نہیں ، ()delete__کو نوڈ کے دائیں بچے کی طرف اشارہ کرنے کے لیے صرف والدین کے حوالے کو اپ ڈیٹ کرنے کی ضرورت ہے۔ یہ کیس 1اور کیس 2دونوں کو ہینڈل کرتا ہے۔ اسے اب بھی یہ طے کرنا ہوگا کہ پیرنٹ آبجیکٹ کے کون سے فیلڈ کو نوڈ کے دائیں بچے کا حوالہ ملتا ہے، بالکل اسی طرح جیسے پہلے لائنوں میں جب صرف بائیں بچہ موجود تھا۔ اس کے مطابق، یہ node.rightChildکو والدین کے root__، leftChild، یا rightChildفیلڈ میں رکھتا ہے۔ آخر میں، یہ نوڈ کا ڈیٹا لوٹاتا ہے جسے حذف کر دیا گیا تھا۔

کیس :3حذف کیے جانے والے نوڈ کے دو بچے ہیں۔

اب مزہ شروع ہوتا ہے۔ اگر حذف شدہ نوڈ کے دو بچے ہیں، تو آپ اسے صرف ان بچوں میں سے کسی ایک سے نہیں بدل سکتے، کم از کم اگر اس بچے کے اپنے (بڑے) بچے ہوں۔ کیوں نہیں؟ شکل 20-8میں جانچیں اور تصور کریں کہ نوڈ 27کو حذف کریں اور اسے اس کے دائیں ذیلی درخت سے تبدیل کریں، جس کی جڑ 33ہے۔ آپ صحیح ذیلی درخت کو فروغ دے رہے ہیں، لیکن اس کے اپنے بچے ہیں۔ نوڈ 33کی نئی پوزیشن میں کون سا بائیں بچہ ہوگا، حذف شدہ نوڈ کا بایاں بچہ، 16یا نوڈ 33کا بائیں بچہ، 28؟اور آپ دوسرے بچے کے ساتھ کیا کرتے ہیں؟ آپ اسے صرف پھینک نہیں سکتے۔

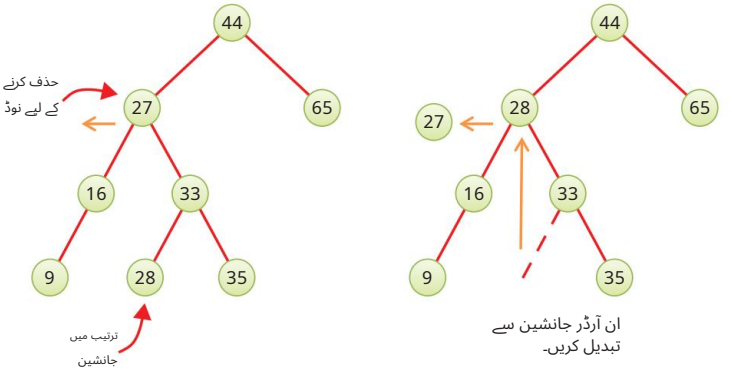


شکل 8-20دو ذیلی درختوں کے ساتھ نوڈ کو حذف کرنے کے اختیارات

شکل 8-20میں درمیانی آپشن تین بچوں کو ممکنہ طور پر اجازت دیتا ہے۔ اس سے بہت ساری دیگر پریشانیاں پیدا ہوں گی کیونکہ درخت اب بائنری نہیں ہے (اس خیال پر مزید کے لیے باب 9دیکھیں)۔ اعداد و شمار میں دائیں ہاتھ کا اختیار حذف شدہ نوڈ کے بائیں بچے، 16کو نیچے دھکیلتے ہوئے اور نئے نوڈ کے بائیں بچے، 28کو اس کے اوپر دھکیلتے ہوئے دکھاتا ہے۔ یہ نقطہ نظر قابل فہم لگتا ہے۔ درخت اب بھی ایک بائنری تلاش درخت ہے، کم از کم. مسئلہ، تاہم، یہ ہے کہ کیا کرنا ہے اگر ترقی یافتہ نوڈ کے بائیں بچے کا اپنا ایک پیچیدہ ذیلی درخت ہے (مثال کے طور پر، اگر اعداد و شمار میں نوڈ 28کے نیچے ایک پورا ذیلی درخت تھا)۔ اس کا مطلب یہ ہو سکتا ہے کہ بائیں ذیلی درختوں کو آپس میں کہاں تقسیم کرنا ہے یہ جاننے کے لیے ایک لمبا راستہ چھوڑنا ہے۔

ہمیں ایک اور نقطہ نظر کی ضرورت ہے۔ اچھی خبر یہ ہے کہ ایک چال ہے۔ بری خبر یہ ہے کہ، یہاں تک کہ چال کے ساتھ، غور کرنے کے لئے خاص معاملات ہیں. یاد رکھیں، بائنری سرچ ٹری میں، نوڈس کو چڑھتی ہوئی چابیاں کی ترتیب سے ترتیب دیا جاتا ہے۔ ہر نوڈ کے لیے، اگلی اعلی ترین کلید کے ساتھ نوڈ کو اس کا ان آرڈر جانشین، یا محض اس کا جانشین کہا جاتا ہے۔ شکل 8-20کے اصل درخت میں، نوڈ 28نوڈ 27کا ان آرڈر جانشین ہے۔

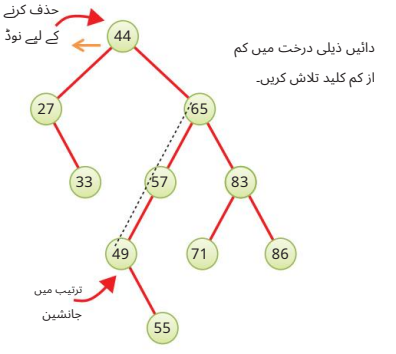
چال یہ ہے: دو بچوں والے نوڈ کو حذف کرنے کے لیے، نوڈ کو اس کے ان آرڈر جانشین سے بدل دیں۔
شکل 8-21ایک حذف شدہ نوڈ کو اس کے جانشین سے تبدیل کر رہا ہے۔ نوٹ کریں کہ نوڈس اب بھی ترتیب میں ہیں۔ اس میں صرف ایک سادہ متبادل تھا۔ اگر جانشین کے خود بچے ہوں تو یہ تھوڑا زیادہ پیچیدہ ہو جائے گا۔ ہم ایک لمحے میں اس امکان کو دیکھتے ہیں۔



تصویر 8-21نوڈ کو اس کے جانشین نے تبدیل کیا ہے۔

جانشین کی تلاش آپ نوڈ کے جانشین کو کیسے تلاش کرتے ہیں؟ انسان یہ کام جلدی کر سکتا ہے (چھوٹے درختوں کے لیے، ویسے بھی)۔ بس درخت پر ایک سرسری نظر ڈالیں اور حذف کیے جانے والے نوڈ کی کلید کو کم کرنے والی اگلی سب سے بڑی تعداد تلاش کریں۔ تصویر 8-21میں یہ دیکھنے میں زیادہ وقت نہیں لگتا کہ 27کا جانشین 28ہے، یا یہ کہ 35کا جانشین 44ہے۔ تاہم، کمپیوٹر "ایک نظر میں" چیزیں نہیں کر سکتا۔ اسے الگورتھم کی ضرورت ہے۔

کم سے کم یا زیادہ سے زیادہ کلید کے ساتھ نوڈ کو تلاش کرنا یاد ہے؟ اس صورت میں آپ حذف کی جانے والی کلید سے بڑی کم از کم کلید تلاش کر رہے ہیں ۔ جس نوڈ کو حذف کیا جانا ہے اس میں بائیں اور دائیں دونوں ذیلی درخت ہیں کیونکہ آپ کیس 3پر کام کر رہے ہیں۔ لہذا، آپ صرف دائیں ذیلی درخت میں کم از کم کلید تلاش کر سکتے ہیں، جیسا کہ شکل 22-8میں دکھایا گیا ہے۔ آپ کو صرف اس وقت تک بائیں چائلڈ لنکس پر عمل کرنے کی ضرورت ہے جب تک کہ آپ کو کوئی نوڈ نہ ملے جس میں بائیں بچے کے ساتھ کوئی بچہ نہ ہو۔

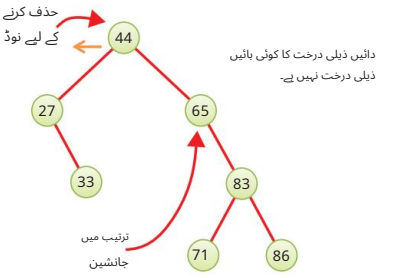


تصویر 22-8جانشین کی تلاش

حذف کیے جانے والے نوڈ کے اوپر جڑے درختوں میں ممکنہ نوڈس کے بارے میں کیا خیال ہے؟ کیا جانشین اس میں کہیں نہیں ہو سکتا؟ آئیے اس کے ذریعے سوچتے ہیں۔ تصور کریں کہ آپ شکل 22-8میں نوڈ 27کے کامیاب سیسر کو تلاش کر رہے ہیں۔ جانشین کی عمر 27سے زیادہ اور 33سال سے کم ہونی چاہیے، جو اس کے صحیح بچے کی کلید ہے۔ ان دونوں اقدار کے درمیان کلید کے ساتھ کوئی بھی نوڈ نوڈ 33کے بائیں ذیلی درخت میں کہیں داخل کیا جائے گا۔ یاد رکھیں کہ آپ ہمیشہ بائنری سرچ ٹری کو تلاش کرتے ہیں اور درخت میں پہلے سے موجود کلیدوں کے لیے کلید کے رشتہ دار ترتیب کی بنیاد پر راستہ منتخب کرتے ہیں۔ مزید برآں، نوڈ 33کو نوڈ 27کے دائیں چائلڈ کے طور پر رکھا گیا تھا کیونکہ یہ روٹ نوڈ، 44سے کم تھا۔ کسی بھی نوڈ کی رائٹ چائلڈ کلید اس کے والدین کی کلید سے کم ہونی چاہیے اگر یہ اس والدین کا بائیں بچہ ہے۔ لہذا والدین، دادا دادی، یا اس سے آگے جانا (بائیں چائلڈ لنکس کے بعد) صرف بڑی کلیدوں کی طرف جاتا ہے، اور وہ چابیاں نہیں ہوسکتی ہیں

جانشین

جانشین کے بارے میں نوٹ کرنے کے لئے ایک دو چیزیں ہیں. اگر حذف کرنے کے لیے اصل نوڈ کے دائیں بچے کے بائیں بچے نہیں ہیں، تو یہ دائیں بچہ خود جانشین ہے، جیسا کہ شکل 23-8کی مثال میں دکھایا گیا ہے۔ چونکہ جانشین کے پاس ہمیشہ ایک خالی بائیں چائلڈ لنک ہوتا ہے، اس لیے اس میں زیادہ سے زیادہ ایک بچہ ہوتا ہے۔



تصویر 23-8صحیح بچہ جانشین ہے۔

جانشین کے ساتھ تبدیل کرنا جانشین کو تلاش کرنے کے بعد، آپ آسانی سے اس کی کلید اور ڈیٹا کی قدروں کو حذف کرنے کے لیے نوڈ میں کاپی کر سکتے ہیں، لیکن جانشین نوڈ پر جڑی ذیلی درخت کے ساتھ آپ کیا کریں گے؟ آپ جانشین نوڈ کی کاپی وہاں درخت میں نہیں چھوڑ سکتے کیونکہ ڈیٹا کو دو جگہوں پر ذخیرہ کیا جائے گا، ڈپلیکیٹ کیز بنائیں گے، اور جانشین کو حذف کرنے کو مستقبل میں ایک مسئلہ بنا دیں گے۔

تو، اسے درخت سے نکالنے کا سب سے آسان طریقہ کیا ہے؟

امید ہے کہ، باب 6پڑھنے سے جواب درست ہو جائے گا۔ اب آپ بار بار آنے والی کال کا استعمال کرکے جانشین کو درخت سے حذف کر سکتے ہیں۔ آپ جانشین پر وہی آپریشن کرنا چاہتے ہیں جو آپ اصل نوڈ پر حذف کرنے کے لیے کر رہے ہیں —جس میں گول کلید ہے۔ کیا فرق ہے کہ آپ کو صرف ایک چھوٹے درخت میں حذف کرنے کی ضرورت ہے، صحیح ذیلی درخت جہاں آپ کو جانشین ملا ہے۔ اگر آپ نے گول نوڈ کو تبدیل کرنے کے بعد درخت کی جڑ سے شروع کرنے کی کوشش کی تو، ()find__طریقہ اسی راستے پر چلے گا اور اس نوڈ پر ختم ہوگا جسے آپ نے ابھی تبدیل کیا ہے۔ جانشین کو حذف کرنے تک آپ کلید کو تبدیل کرنے میں تاخیر کرکے اس مسئلے کو حل کرسکتے ہیں، لیکن یہ بہت آسان ہے —اور زیادہ اہم بات، تیز —اگر آپ صحیح ذیلی درخت میں ایک نیا حذف آپریشن شروع کرتے ہیں۔ تلاش کرنے کے لیے بہت کم درخت ہوں گے، اور آپ غلطی سے پچھلے گول نوڈ پر نہیں پہنچ سکتے ۔

درحقیقت، جب آپ نے جانشین کو تلاش کیا، تو آپ نے راستے کا تعین کرنے کے لیے چائلڈ لنکس کی پیروی کی، اور اس سے آپ کو جانشین اور جانشین کا پیرنٹ نوڈ دونوں مل گئے۔ ان دو حوالوں کی دستیابی کے ساتھ، اب آپ کے پاس پرائیویٹ ()delete__طریقہ کو کال کرنے کے لیے درکار ہر چیز موجود ہے جو فہرست 8-8میں دکھائے گئے ہیں۔ اب آپ ()promote_successor__طریقہ کی وضاحت کر سکتے ہیں، جیسا کہ فہرست 9-8میں دکھایا گیا ہے۔ یاد رکھیں، یہ وہ طریقہ ہے جو کیس 3کو ہینڈل کرنے کے لیے استعمال کیا جاتا ہے —جب حذف کرنے والے نوڈ کے دو بچے ہوں۔

()promote_successor__طریقہ اپنے واحد پیرامیٹر کے طور پر نوڈ کو حذف کرتا ہے۔
چونکہ یہ اس نوڈ کے ڈیٹا اور کلید کو تبدیل کرنے جا رہا ہے اور پھر جانشین کو حذف کر دے گا، اس تناظر میں اسے تبدیل کرنے کے لیے نوڈ کے طور پر حوالہ دینا آسان ہے۔ شروع کرنے کے لیے، یہ تبدیل کیے جانے والے نوڈ کے دائیں چائلڈ پر ایک succes sorمتغیر کی نشاندہی کرتا ہے۔ بالکل ()find__طریقہ کی طرح، یہ جانشین نوڈ کے پیرنٹ کو ٹریک کرتا ہے، جسے تبدیل کرنے کے لیے نوڈ بننے کے لیے شروع کیا جاتا ہے۔

پھر یہ ()minNodeطریقہ کی طرح کام کرتا ہے، اگر بائیں بچہ ہے تو اس کے بائیں بچے کے ساتھ جانشین کو اپ ڈیٹ کرنے کے لیے ایک whileلوپ کا استعمال کرتا ہے۔ جب لوپ باہر نکلتا ہے، جانشین نوڈ اور پیرنٹ کو اس کے پیرنٹ نوڈ کی طرف اشارہ کرتا ہے۔

Machine Translated by Google

فہرست سازی 8-9 BinarySearchTree کا () promote_successor__طریقہ

کلاس بائنری سرچ ٹری (آبجیکٹ): BinarySearchTree

...

```
def __promote_successor(خود، نوڈ):            #دونوں ذیلی درختوں کے ساتھ نوڈ کو حذف کرتے وقت، #دائیں سب ٹری پر جانشین تلاش کریں،
                                             #اس کا ڈیٹا اس نوڈ میں ڈالیں، اور #جانشین کو دائیں سب ٹری سے حذف کریں۔
    جانشین = node.rightChild                  #میں جانشین کی تلاش شروع کریں۔
    والدین = نوڈ جبکہ successor.leftChild:    #دائیں ذیلی درخت اور اس کے والدین کو ٹریک کریں۔
                                             #نیچے بائیں چائلڈ لنکس تک
        والدین = جانشین جانشین = successor.leftChild  #مزید بائیں لنک نہیں ملتے
    node.key = successor.key node.data = successor.data  #جانشین کی کلید اور ڈیٹا کو حذف کرنے کے لیے نوڈ میں منتقل کریں
    self.__delete(والدین، جانشین)              #جانشین نوڈ کو ہٹا دیں۔
```

بس اتنا کرنا باقی ہے کہ نوڈ کی کلید اور ڈیٹا کو اپ ڈیٹ کرنا ہے جسے تبدیل کیا جانا ہے اور ()delete__پر دوبارہ آنے والی کال کا استعمال کرتے ہوئے جانشین نوڈ کو حذف کرنا ہے۔ آپ کے دیکھے ہوئے پچھلے تکراری طریقوں کے برعکس، یہ اسی روٹین کے لیے کال نہیں ہے جہاں کال ہوتی ہے۔ اس صورت میں، ()promote_successorطریقہ ()delete__کو کال کرتا ہے، جو بدلے میں ()promote_suc cessor__ کو کال کرسکتا ہے۔ اسے باہمی تکرار کہتے ہیں —جہاں دو یا زیادہ معمولات ایک دوسرے کو کال کرتے ہیں۔

آپ کے حواس اب کانپ رہے ہوں گے۔ آپ کیسے جانتے ہیں کہ یہ باہمی تکرار ختم ہو جائے گی؟ بنیادی کیس کہاں ہے جو آپ نے "سادہ" تکرار کے معمولات کے ساتھ دیکھا؟ کیا آپ باہمی تکراری کالوں کے لامحدود لوپ میں داخل ہو سکتے ہیں؟ اس کے بارے میں فکر کرنا اچھی بات ہے، لیکن یہاں ایسا نہیں ہونے والا ہے۔ یاد رکھیں کہ نوڈ کو حذف کرنا تین صورتوں میں ٹوٹ جاتا ہے۔ کیس 1اور 2ایک بچے کے ساتھ لیف نوڈس اور نوڈس کو حذف کرنے کے تھے۔ ان دو کیسز سے ()promote_successor__کالز نہیں ہوئیں ، اس لیے وہ بنیادی کیسز ہیں۔ جب آپ کیس 3کے لیے ()promote_successor__کو کال کرتے ہیں ، تو یہ حذف کرنے کے لیے نوڈ پر جڑی ذیلی درخت پر کام کرتا ہے، اس لیے واحد موقع یہ ہے کہ درخت کو بار بار پروسیس کیا جا رہا ہے اصل سے چھوٹا نہیں ہے اگر حذف کرنے والا نوڈ روٹ نوڈ ہے۔ . تاہم، کلینچر یہ ہے کہ ()delete__ ()mote_successor pro__کو صرف جانشین نوڈس پر کال کرتا ہے —ایسے نوڈس جن پر زیادہ سے زیادہ ایک بچہ ہونے کی ضمانت دی جاتی ہے اور درخت میں کم از کم ایک سطح اس نوڈ سے کم ہوتی ہے جس پر انہوں نے شروع کیا تھا۔ وہ ہمیشہ ایک بنیادی کیس کی طرف لے جاتے ہیں اور کبھی بھی لامحدود تکرار نہیں کرتے ہیں۔

دو بچوں کے ساتھ نوڈ کو حذف کرنے کے لیے ویژولائزیشن ٹول کا استعمال کرتے ہوئے ویژولائزیشن ٹول کے ساتھ ایک درخت بنائیں اور دو بچوں کے ساتھ ایک نوڈ چنیں۔ اب مرد یہ پتہ لگاتے ہیں کہ کون سا نوڈ اس کا جانشین ہے، اس کے دائیں بچے کے پاس جا کر اور پھر اس دائیں بچے کے بائیں بچوں کی لائن پر عمل کرتے ہوئے (اگر کوئی ہے)۔ اپنی پہلی کوشش کے لیے، آپ یہ یقینی بنانا چاہیں گے کہ جانشین کی اپنی کوئی اولاد نہیں ہے۔ بعد کی کوششوں پر، زیادہ پیچیدہ صورتحال کو دیکھنے کی کوشش کریں جہاں ایک نوڈ کے بجائے جانشین کے پورے ذیلی درختوں کو ادھر ادھر منتقل کیا جاتا ہے۔

حذف کرنے کے لیے نوڈ کا انتخاب کرنے کے بعد، حذف کریں بٹن پر کلک کریں۔ آپ انفرادی مراحل کو ٹریک کرنے کے لیے قدم یا توقف/پلے بٹن استعمال کرنا چاہتے ہیں۔ ہم نے جو طریقہ بیان کیا ہے ان میں سے ہر ایک کوڈ ونڈو میں ظاہر ہوگا، لہذا آپ دیکھ سکتے ہیں کہ یہ کس طرح نوڈ کو حذف کرنے کا فیصلہ کرتا ہے جس کے دو بچے ہیں، جانشین کا پتہ لگاتا ہے، جانشین کلید اور ڈیٹا کو کاپی کرتا ہے، اور پھر جانشین نوڈ کو حذف کرتا ہے۔

کیا حذف کرنا ضروری ہے؟

اگر آپ یہاں تک پہنچے ہیں، تو آپ دیکھ سکتے ہیں کہ حذف کرنا کافی حد تک شامل ہے۔ درحقیقت، یہ اتنا پیچیدہ ہے کہ کچھ پروگرامرز اسے مکمل طور پر چھوڑنے کی کوشش کرتے ہیں۔ وہ Node__کلاس میں ایک نیا بولین فیلڈ شامل کرتے ہیں ، جسے isDeletedکہا جاتا ہے۔ نوڈ کو حذف کرنے کے لیے، انہوں نے صرف اس فیلڈ کو Trueپر سیٹ کیا۔ یہ ایک طرح کا "نرم" حذف ہے، جیسے کسی فائل کو صحیح معنوں میں حذف کیے بغیر کوڑے دان کے فولڈر میں منتقل کرنا۔ پھر دوسرے آپریشنز، جیسے ,()find__اس فیلڈ کو چیک کریں تاکہ یہ یقینی بنایا جا سکے کہ نوڈ کے ساتھ کام کرنے سے پہلے اسے حذف شدہ کے طور پر نشان زد نہیں کیا گیا ہے۔ اس طرح، نوڈ کو حذف کرنے سے درخت کی ساخت تبدیل نہیں ہوتی ہے۔ یقیناً، اس کا مطلب یہ بھی ہے کہ میموری پہلے "حذف شدہ" نوڈس سے بھر سکتی ہے۔

یہ نقطہ نظر تھوڑا سا پولیس آؤٹ ہے، لیکن یہ مناسب ہوسکتا ہے جہاں درخت میں بہت زیادہ حذف نہیں ہوں گے۔ بہت محتاط رہیں۔ اس طرح کے مفروضے آپ کو پریشان کرنے کے لئے واپس آتے ہیں۔ مثال کے طور پر، یہ فرض کرنا کہ کمپنی کے عملے کے ریکارڈ کے لیے حذف کرنا متواتر نہیں ہو سکتا ہے، پروگرامر کو isDeletedفیلڈ استعمال کرنے کی ترغیب دے سکتا ہے۔ اگر کمپنی سیکڑوں سالوں تک قائم رہتی ہے تو مستقبل میں کسی وقت فعال ملازمین سے زیادہ حذف ہونے کا امکان ہے۔ یہی بات درست ہے اگر کمپنی کو ٹرن اوور کی اعلی شرحوں کا سامنا کرنا پڑتا ہے، یہاں تک کہ مختصر وقت کے اندر بھی۔ یہ درختوں کی کارروائیوں کی کارکردگی کو نمایاں طور پر متاثر کرے گا۔

ثنائی تلاش کے درختوں کی کارکردگی جیسا کہ آپ نے دیکھا ہے، درختوں کے ساتھ زیادہ تر کارروائیوں میں کسی خاص نوڈ کو تلاش کرنے کے لیے درخت کو سطح سے دوسری سطح تک اتارنا شامل ہے۔ اس آپریشن میں کتنا وقت لگتا ہے؟ ہم نے پہلے ذکر کیا ہے کہ نوڈ کو تلاش کرنے کی کارکردگی O(log N)سے O(N)تک ہو سکتی ہے، لیکن آئیے تفصیلات دیکھیں۔

ایک مکمل، متوازن درخت میں، تقریباً نصف نوڈس نیچے کی سطح پر ہوتے ہیں۔ زیادہ درست طور پر، ایک مکمل، متوازن درخت میں، باقی درخت کے مقابلے میں نیچے کی قطار میں بالکل ایک اور نوڈ ہوتا ہے۔ اس طرح، تمام تلاشوں یا اندراجات یا حذفوں میں سے تقریباً نصف کے لیے نچلی سطح پر نوڈ تلاش کرنے کی ضرورت ہوتی ہے۔ (تمام تلاشی کارروائیوں کے ایک چوتھائی کے لیے اگلے سے نچلی سطح پر نوڈ تلاش کرنے کی ضرورت ہوتی ہے، وغیرہ۔)

تلاش کے دوران، آپ کو ہر سطح پر ایک نوڈ کا دورہ کرنے کی ضرورت ہے۔ آپ یہ جان کر اچھی طرح اندازہ لگا سکتے ہیں کہ ان کارروائیوں کو انجام دینے میں کتنا وقت لگتا ہے یہ جان کر کہ کتنے درجے ہیں۔ ایک مکمل، متوازن درخت کو فرض کریں، جدول 1-8دکھاتا ہے کہ دی گئی تعداد میں نوڈس کو رکھنے کے لیے کتنی سطحیں ضروری ہیں۔

نمبر بہت زیادہ ایسے ہی ہیں جیسے باب ٹیر 2میں زیر بحث ترتیب شدہ سرنی کو تلاش کرنے کے لیے۔ اس صورت میں، بائنری تلاش کے لیے موازنہ کی تعداد تقریباً ارے میں خلیوں کی تعداد کے بیس 2لاگرتھم کے برابر تھی۔ یہاں، اگر آپ پہلے کالم Nمیں نوڈس کی تعداد، اور دوسرے کالم Lمیں لیولز کی تعداد کو کال کرتے ہیں، تو آپ کہہ سکتے ہیں کہ Nپاور Lمیں 2سے 1کم ہے، یا

$$N = 2L - 1$$

مساوات کے دونوں اطراف میں 1کا اضافہ کرنا، آپ کے پاس ہے۔

$$N + 1 = 2L$$

باب 2میں جو کچھ آپ نے سیکھا ہے اس کا استعمال کرتے ہوئے لاگرتھم ایک عدد کو طاقت تک بڑھانے کے الٹا ہونے کے بارے میں، آپ دونوں اطراف کا لوگارتھم لے سکتے ہیں اور شرائط کو دوبارہ ترتیب دے سکتے ہیں۔

$$\log_2 (N + 1) = \log_2 (2L) = L$$

$$L = \log_2 (N + 1)$$

اس طرح، عام درختوں کی کارروائیوں کو انجام دینے کے لیے درکار وقت Nکے بیس 2لاگ کے متناسب ہے۔ بگ Oاشارے میں، آپ کہتے ہیں کہ اس طرح کی کارروائیوں میں O(log N)وقت لگتا ہے۔

جدول 8-1نوڈس کی مخصوص تعداد کے لیے سطحوں کی تعداد

نوٹسطحوں کی تعداد		نوڈس کی تعداد
1	1	1
2	2	3
3	3	7
4	4	15
5	5	31
...	...	
10	10	1,023
...	...	...
15	15	32,767
...	...	...
20	20	1,048,575
...	...	...
25	25	33,554,431
...	...	...
30	30	1,073,741,823

اگر درخت مکمل یا متوازن نہیں ہے، تو تجزیہ مشکل ہے۔ آپ کہہ سکتے ہیں کہ درجات کی دی گئی تعداد والے درخت کے لیے، غیر مکمل درخت کے لیے اوسط تلاش کا وقت پورے درخت کے مقابلے میں کم ہوگا کیونکہ کم تلاشیں نچلی سطح پر جائیں گی۔

درخت کا موازنہ دوسرے ڈیٹا اسٹوریج ڈھانچے سے کریں جن پر ہم نے اب تک بات کی ہے۔ 1,000,000آئٹمز پر مشتمل ایک غیر یا ڈیرڈ صف میں یا ایک لنک شدہ فہرست میں، جس چیز کو آپ چاہتے ہیں اسے تلاش کرنے کے لیے، اوسطاً، 500،000موازنہ، بنیادی طور پر 1,000,000 ۔O(N)اشیاء کے متوازن درخت میں، صرف 20(یا اس سے کم) موازنہ کی ضرورت ہے کیونکہ یہ O(log N)ہے۔

ترتیب شدہ صف میں، آپ اتنی ہی تیزی سے ایک آئٹم تلاش کر سکتے ہیں، لیکن کسی آئٹم کو داخل کرنے کے لیے، اوسطاً، 500,000 آئٹمز کو منتقل کرنے کی ضرورت ہوتی ہے۔ 1,000,000آئٹمز والے درخت میں کسی آئٹم کو داخل کرنے کے لیے 20یا اس سے کم موازنے درکار ہوتے ہیں، نیز آئٹم کو جوڑنے کے لیے تھوڑا سا وقت درکار ہوتا ہے۔ اضافی وقت مستقل ہے اور اشیاء کی تعداد پر منحصر نہیں ہے۔

اسی طرح، -1,000,000آئٹم سرنی سے کسی آئٹم کو حذف کرنے کے لیے اوسطاً 500,000آئٹمز کو منتقل کرنے کی ضرورت ہوتی ہے، جب کہ -1,000,000نوڈ ٹری سے کسی آئٹم کو حذف کرنے کے لیے 20یا اس سے کم کی ضرورت ہوتی ہے۔

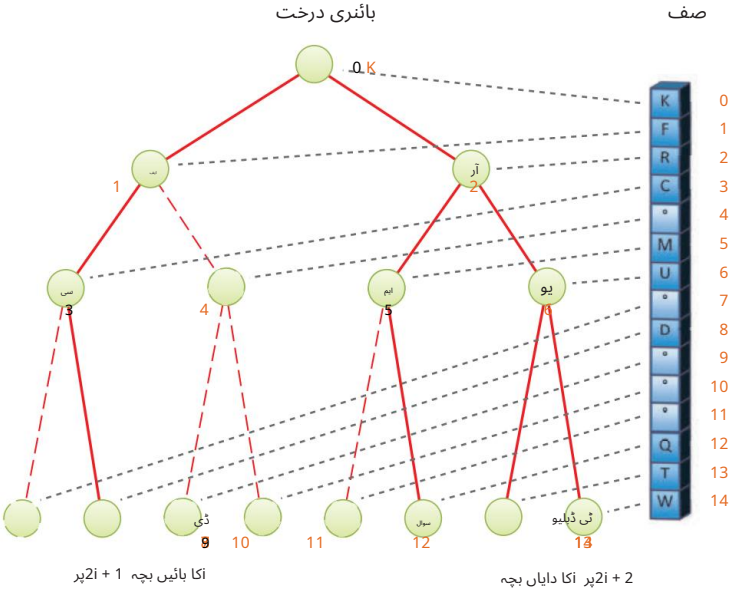
آئٹم کو تلاش کرنے کے لیے موازنہ، نیز اس کے جانشین کو تلاش کرنے کے لیے کچھ اور موازنہ، نیز اس چیز کو منقطع کرنے اور اس کے جانشین کو جوڑنے کے لیے مختصر وقت۔ چونکہ جانشین حذف کرنے کے لیے نوڈ کے مقابلے میں درخت میں کہیں کم ہے، نوڈ اور اس کے جانشین دونوں کو تلاش کرنے کے لیے موازنہ کی کل تعداد 20یا اس سے کم ہوگی۔

اس طرح، ایک درخت ڈیٹا ذخیرہ کرنے کے تمام کاموں کے لیے اعلیٰ کارکردگی فراہم کرتا ہے: تلاش، اندراج، اور حذف کرنا۔ ٹراورسنگ دیگر آپریشنز کی طرح تیز نہیں ہے، لیکن تعریف کے مطابق، تمام Nآئٹمز کا احاطہ کرنے کے لیے اسے O(N)ہونا چاہیے۔ آپ نے جو بھی ڈیٹا ڈھانچہ دیکھا ہے، ان میں یہ O(N)رہا ہے، لیکن ہم بعد میں کچھ دوسرے ڈیٹا ڈھانچے دکھاتے ہیں جہاں یہ زیادہ ہو سکتا ہے۔ صفوں یا فہرستوں کے مقابلے درخت کو عبور کرنے کے لیے تھوڑی زیادہ میموری کی ضرورت ہوتی ہے کیونکہ آپ کو بار بار آنے والی کالز کو اسٹور کرنے یا اسٹیک استعمال کرنے کی ضرورت ہوتی ہے۔ وہ میموری O(log N)ہوگی۔ یہ ان صفوں اور فہرستوں سے متصادم ہے جنہیں ٹراورسل کے دوران صرف O(1)میموری کی ضرورت ہوتی ہے۔

درختوں کو صفوں کے طور پر دکھایا گیا ہے۔

اب تک، ہم نے بائیں اور دائیں بچوں کے حوالہ جات کے ساتھ آبجیکٹ کا استعمال کرتے ہوئے بائنری ٹری نوڈس کی نمائندگی کی ہے۔ درخت کی نمائندگی کرنے کا ایک بالکل مختلف طریقہ ہے: ایک صف کے ساتھ۔

صف کے نقطہ نظر میں، نوڈس ایک صف میں محفوظ ہوتے ہیں اور حوالہ جات سے منسلک نہیں ہوتے ہیں۔ صف میں نوڈ کی پوزیشن درخت میں اس کی پوزیشن کے مساوی ہے۔ ہم روٹ نوڈ کو انڈیکس 0پر رکھتے ہیں۔ جڑ کے بائیں بچے کو انڈیکس 1پر اور اس کے دائیں بچے کو انڈیکس 2پر رکھا جاتا ہے، اور اسی طرح درخت کی ہر سطح کے ساتھ ساتھ بائیں سے دائیں آگے بڑھتا ہے۔ یہ نقطہ نظر تصویر 24-8میں دکھایا گیا ہے، جو ایک بائنری سرچ ٹری ہے جس میں کلید کے حروف ہیں۔



تصویر 24-8ایک بائنری درخت جس کی نمائندگی ایک صف سے ہوتی ہے۔

درخت کی ہر پوزیشن، چاہے وہ موجودہ نوڈ کی نمائندگی کرے یا نہ کرے، صف میں موجود سیل سے مساوی ہے۔ درخت میں دی گئی پوزیشن پر نوڈ کو شامل کرنے کا مطلب ہے نوڈ کو صف میں موجود مساوی سیل میں داخل کرنا۔ بغیر نوڈس کے درخت کی پوزیشنوں کی نمائندگی کرنے والے سیل ،None ،0یا کچھ اور خاص قدر سے بھرے ہوتے ہیں جنہیں نوڈ کے ساتھ الجھایا نہیں جا سکتا۔

تصویر میں، خالی نوڈس کے لیے صف میں °علامت استعمال کی گئی ہے۔

اس اسکیم کے ساتھ، ایک نوڈ کے بچوں اور والدین کو صف میں نوڈ کے انڈیکس نمبر پر کچھ آسان ریاضی لگا کر تلاش کیا جا سکتا ہے۔ اگر نوڈ کا انڈیکس نمبر انڈیکس ہے، تو اس نوڈ کا بایاں بچہ ہے۔

2انڈیکس 1 +

اس کا صحیح بچہ ہے

2انڈیکس 2 +

اور اس کے والدین ہیں

)انڈیکس 2 // (1

(جہاں // بغیر کسی باقی کے عددی تقسیم کی نشاندہی کرتا ہے)۔ آپ شکل 24-8میں انڈیکس کو دیکھ کر ان فارمولوں کے کام کی تصدیق کر سکتے ہیں۔ کوئی بھی الگورتھم جو نوڈس کے درمیان روابط کی پیروی کرتا ہے آسانی سے اس بات کا تعین کر سکتا ہے کہ اگلے نوڈ کو کہاں چیک کرنا ہے۔ یہ اسکیم کسی بھی بائنری درخت کے لیے کام کرتی ہے، نہ کہ صرف بائنری تلاش کے درختوں کے لیے۔ اس میں اچھی خصوصیت ہے کہ نوڈس کے درمیان کناروں / لنکس کا سفر کرنا اتنا ہی آسان ہے جتنا کہ وہ نیچے جا رہے ہیں (فہرستوں کے لئے ڈبل لنکنگ کی ضرورت کے بغیر)۔ اس سے بھی بہتر، یہ کسی بھی درخت پر عام کیا جا سکتا ہے جس میں بچوں کی ایک مقررہ تعداد ہے۔

تاہم، زیادہ تر حالات میں، ایک صف کے ساتھ درخت کی نمائندگی کرنا زیادہ کارگر نہیں ہوتا ہے۔ بھرے ہوئے نوڈس صف میں سوراخ چھوڑ دیتے ہیں، میموری کو ضائع کر دیتے ہیں۔ اس سے بھی بدتر، جب نوڈ کو حذف کرنے میں ذیلی درختوں کو منتقل کرنا شامل ہوتا ہے، سب ٹری میں موجود ہر نوڈ کو صف میں اس کے نئے مقام پر منتقل کیا جانا چاہیے، جو بڑے درختوں میں وقت طلب ہے۔ ان انسریشنز کے لیے جو نوڈس کو درخت کی موجودہ زیادہ سے زیادہ گہرائی سے باہر داخل کرتے ہیں، سرنی کا سائز تبدیل کرنے کی ضرورت پڑ سکتی ہے۔

اگر حذف کرنے کی اجازت نہیں ہے یا بہت نایاب ہیں اور درخت کی زیادہ سے زیادہ گہرائی کا اندازہ لگایا جا سکتا ہے، تو صف کی نمائندگی مفید ہو سکتی ہے، خاص طور پر اگر ہر نوڈ کے لیے متحرک طور پر میموری حاصل کرنا، کسی وجہ سے، بہت زیادہ وقت طلب ہے۔ ایسا ہو سکتا ہے جب اسمبلی لینگویج میں پروگرامنگ ہو یا ایک بہت ہی محدود آپریٹنگ سسٹم، یا ایسا سسٹم جس میں کوڑا کرکٹ جمع نہ ہو۔

درخت کی سطح اور سائز

جب درختوں کو صفوں کے طور پر پیش کیا جاتا ہے تو، زیادہ سے زیادہ سطح اور نوڈس کی تعداد سرنی کے سائز کے لحاظ سے تنگ ہوتی ہے۔ منسلک درختوں کے لیے، کوئی خاص زیادہ سے زیادہ نہیں ہے۔ دونوں نمائندگیوں کے لیے، موجودہ زیادہ سے زیادہ سطح اور نوڈس کی تعداد کا تعین صرف درخت سے گزر کر کیا جا سکتا ہے۔ اگر ان میٹرکس کی درخواست کرنے کے لیے متواتر کالیں آتی ہیں، تو Binary SearchTree آبجیکٹ ان کے لیے اقدار کو برقرار رکھ سکتا ہے، لیکن ()insertاور ()deleteطریقوں کو قدروں کو اپ ڈیٹ کرنے کے لیے تبدیل کیا جانا چاہیے کیونکہ نوڈس کو شامل اور ہٹا دیا جاتا ہے۔

منسلک درخت میں نوڈس کو گننے کے لیے، آپ تمام نوڈس پر اعادہ کرنے کے لیے ()traversal طریقہ استعمال کر سکتے ہیں اور گنتی میں اضافہ کر سکتے ہیں، جیسا کہ پہلے مثال میں اوسط کلیدی قدر تلاش کرنے کے لیے اور دوبارہ فہرست 8 کے ()nodes طریقہ میں دکھایا گیا ہے۔ 10-زیادہ سے زیادہ سطح تلاش کرنے کے لیے، آپ ایک ہی تکنیک کا استعمال نہیں کر سکتے ہیں کیونکہ ٹراورسل کے دوران ہر نوڈ کی سطح فراہم نہیں کی جاتی ہے (حالانکہ اسے جنریٹر میں ترمیم کرکے شامل کیا جا سکتا ہے)۔ اس کے بجائے، لسٹنگ 10-8 میں دکھائی گئی تکراری تعریف کام کو کوڈ کی چند سطروں میں انجام دیتی ہے۔

فہرست سازی BinarySearchTree 10-8 کے درجے ()اور نوڈس() طریقے

```
...

    #ایٹریٹر کا استعمال کرتے ہوئے ٹری نوڈس کو شمار کریں۔
    #ایک خالی درخت فرض کریں۔
    کلید کے لیے ، self.traverse میں ڈیٹا # :()کسی بھی میں تمام کلیدوں پر اعادہ کریں۔
    #آرڈر اور انکریمنٹ شمار
        شمار 1 =+
    واپسی کی گنتی

#بار بار ذیلی درخت میں سطحوں کی گنتی کریں۔
#اگر نوڈ فراہم کیا جاتا ہے، تو لیول 1 ہے۔
    واپسی # max child self.__levels(node.leftChild), # self.__levels(node.rightChild)) + 1 خالی ذیلی درخت کی
    کوئی سطح نہیں ہے

    دوسری: واپسی 0

def __levels(self, node): if node:
    ڈیف نوڈس (خود):
    شمار 0 =
    واپسی self.__levels(self.__root) # جڑ سے شروع ہونے والی گنتی
```

ذیلی درخت کی سطحوں کو شمار کرنا اس سے کچھ مختلف ہے جو آپ نے پہلے دیکھا ہے کہ ہر نوڈ اپنے ہر ذیلی درخت کی زیادہ سے زیادہ سطح لیتا ہے اور نوڈ کے لیے اس میں ایک کا اضافہ کرتا ہے۔ ایسا لگتا ہے جیسے کم سے کم یا زیادہ سے زیادہ کلید کی گہرائی کو دیکھ کر کوئی شارٹ کٹ ہونا چاہئے تاکہ آپ کو ہر نوڈ پر جانے کی ضرورت نہ ہو۔ اگر آپ اس کے بارے میں سوچتے ہیں، تاہم، یہاں تک کہ کم سے کم اور زیادہ سے زیادہ چابیاں تلاش کرنے سے بھی درخت کے بائیں اور دائیں "فلانکس" پر گہرائی ظاہر ہوتی ہے۔ درمیان میں کہیں طویل راستے ہوسکتے ہیں، اور انہیں تلاش کرنے کا واحد طریقہ تمام نوڈس کا دورہ کرنا ہے۔

پرنٹنگ درخت

آپ نے دیکھا ہے کہ درختوں کو مختلف ترتیبوں میں کیسے عبور کرنا ہے۔ آپ درخت کے تمام نوڈس کو پرنٹ کرنے کے لیے ہمیشہ ٹراورسل طریقہ استعمال کر سکتے ہیں، جیسا کہ ویژولائزیشن ٹول میں دکھایا گیا ہے۔ ان آرڈر ٹراورسل کا استعمال اشیاء کو ان کی چابیاں کے بڑھتے ہوئے ترتیب میں دکھائے گا۔ دو جہتی آؤٹ پٹ پر، آپ افقی محور کے ساتھ نوڈس اور ہر نوڈ کی سطح کو اس کی عمودی پوزیشن کا تعین کرنے کے لیے ترتیب ترتیب کا استعمال کر سکتے ہیں۔ یہ پچھلے اعداد و شمار میں دکھائے گئے درختوں کے خاکے تیار کر سکتا ہے۔

ایک سادہ کمانڈ لائن آؤٹ پٹ پر، فی لائن ایک نوڈ پرنٹ کرنا آسان ہے۔ پھر مسئلہ درخت کی شکل کی نشاندہی کرنے کے لیے لائن پر نوڈ کی پوزیشننگ بن جاتا ہے۔ اگر آپ سب سے اوپر روٹ نوڈ چاہتے ہیں، تو آپ کو پورے درخت اور جگہ کی چوڑائی کا حساب لگانا ہوگا۔

وہ نوڈ پوری چوڑائی کے وسط میں ہے۔ زیادہ درست طریقے سے، آپ کو بائیں اور دائیں ذیلی درختوں کی چوڑائی کا حساب لگانا ہوگا اور متوازن اور غیر متوازن درختوں کو درست طریقے سے دکھانے کے لیے جڑ کو پوزیشن میں رکھنے کے لیے اس کا استعمال کرنا ہوگا۔

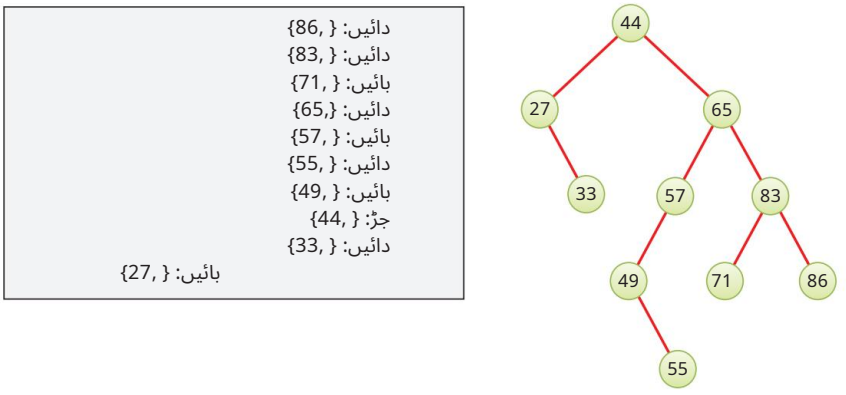
دوسری طرف، اگر آپ روٹ کو آؤٹ پٹ لائن کے بائیں جانب رکھتے ہیں اور نوڈس کی سطح کو سب سے بائیں کالم سے انڈینٹیشن کے طور پر دکھاتے ہیں، تو درخت کو ٹرمینل پر پرنٹ کرنا آسان ہے۔ ایسا کرنا بنیادی طور پر درخت کو 90° بائیں گھماتا ہے۔ درخت کا ہر نوڈ آؤٹ پٹ کی اپنی لائن پر ظاہر ہوتا ہے۔ یہ آپ کو ذیلی درختوں کی چوڑائی کا تعین کرنے کے بارے میں بھولنے اور ایک سادہ تکراری طریقہ لکھنے کی اجازت دیتا ہے، جیسا کہ فہرست 11-8میں دکھایا گیا ہے۔

ایک نوڈ فی لائن کے ساتھ درختوں کو پرنٹ کرنے کے 11-8طریقوں کی فہرست

```
...
    def pTree(self, node, nodeType, indent, indentBy=4):
        #نوڈ ٹائپ اپنے #والدین سے تعلق کو ظاہر کرتا ہے اور انڈینٹ اس کی سطح کو ظاہر کرتا ہے

        #ذیلی درختوں کے لیے انڈینٹ لیول میں اضافہ کریں۔
        #صرف اس صورت میں پرنٹ کریں جب کوئی نوڈ ہو۔
        if node != None:
            print(indent + nodeType + ":", node) # اگر نوڈ:
            #اس نوڈ کو خود پرنٹ کریں۔ # "  " * indentBy,
            #بائیں حاشیہ پرنٹ کریں صحیح حاشیہ + indentBy) # subtree
            self.__pTree(node.leftChild, "LEFT:", # پرنٹ کریں +
                         "  " * indentBy, indentBy) # ذیلی درخت
            self.__pTree(node.rightChild, "RIGHT:", #
                         indentBy) # subtree print(indent + nodeType, node) #
```

پبلک پرنٹ ()طریقہ روٹ نوڈ سے شروع ہونے والے نوڈس کو بار بار پرنٹ کرنے کے لیے نجی pTree()__طریقہ کو کہتے ہیں۔ یہ ایک پیرامیٹر لیتا ہے، indentBy، یہ کنٹرول کرنے کے لیے کہ درخت کی ہر سطح کو انڈینٹ کرنے کے لیے کتنی جگہیں استعمال کی جاتی ہیں۔ یہ نوڈس پر لیبل لگاتا ہے تاکہ ان کے والدین کے ساتھ ان کا رشتہ ظاہر کیا جا سکے (اگر یہ ان کے انڈینٹیشن اور رشتہ دار پوزیشن سے پہلے ہی واضح نہیں تھا)۔ تکراری طریقہ پر عمل درآمد بیس کیس، ایک خالی نوڈ کو چیک کرکے شروع ہوتا ہے، اس صورت میں کچھ بھی پرنٹ کرنے کی ضرورت نہیں ہے۔ ہر دوسرے نوڈ کے لیے، یہ سب سے پہلے دائیں سب ٹری کو بار بار پرنٹ کرتا ہے کیونکہ یہ پرنٹ شدہ ورژن کا سب سے اوپر ہے۔ یہ انڈینٹ میں خالی جگہیں شامل کرتا ہے تاکہ ذیلی درخت کو مزید دائیں طرف پرنٹ کیا جائے۔ پھر یہ موجودہ نوڈ کو اس کے انڈینٹیشن اور نوڈ ٹائپ لیبل کے ساتھ پہلے سے پرنٹ کرتا ہے۔ آخر میں، یہ توسیع شدہ انڈینٹیشن کے ساتھ بائیں ذیلی درخت کو چھپی ہوئی چھاپتا ہے۔ یہ ایک آؤٹ پٹ تیار کرتا ہے جیسا کہ شکل 8-25میں دکھایا گیا ہے۔ نوڈس کو {key, data}جوڑوں کے بطور پرنٹ کیا جاتا ہے اور اعداد و شمار کی مثال میں اس کے ساتھ کوئی ڈیٹا محفوظ نہیں ہوتا ہے۔



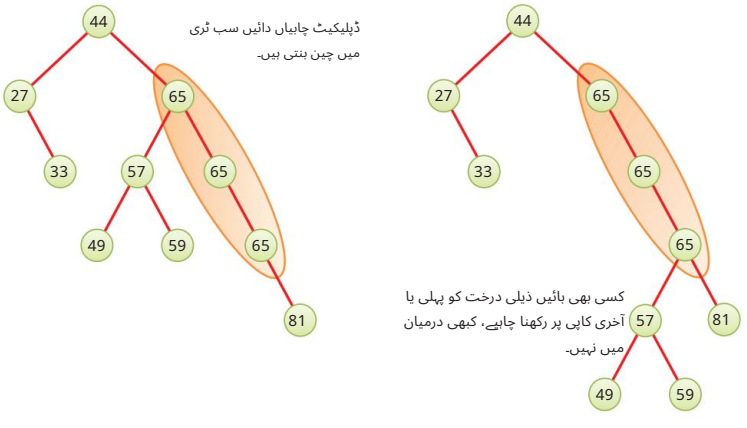
شکل 25-8درخت نوڈ کی گہرائی کے لیے انڈینٹیشن کے ساتھ پرنٹ کیا گیا ہے۔

درخت کو اس طرح پرنٹ کرنے میں، آپ تین معیاری ترتیب سے مختلف ٹراورسل آرڈر استعمال کرتے ہیں۔ پرنٹ آرڈر درخت کے الٹ ان آرڈر ٹراورسل کا استعمال کرتا ہے۔

ڈپلیکیٹ چابیاں

دوسرے ڈیٹا ڈھانچے کی طرح، ڈپلیکیٹ کیز کے مسئلے کو حل کرنا ضروری ہے۔ ()insertکے لیے دکھائے گئے کوڈ میں اور ویژولائزیشن ٹول میں، ڈپلیکیٹ کلید والا نوڈ داخل نہیں کیا جائے گا۔ ٹول نوڈ کو بھرنے کے لیے ایک نئے رنگ کے دائرے کو حرکت دے کر اپ ڈیٹ کیے جانے والے نوڈ کا ڈیٹا دکھاتا ہے۔

ڈپلیکیٹ کلیدوں کی اجازت دینے کے لیے، آپ کو کئی انتخاب کرنا ہوں گے۔ ڈپلیکیٹس بنیادی بائنری سرچ ٹری اصول کی بنیاد پر صحیح ذیلی درخت میں جاتے ہیں۔ وہ صرف صحیح چائلڈ لنکس کے ساتھ نوڈس کا ایک سلسلہ بناتے ہیں، جیسا کہ شکل 26-8میں دکھایا گیا ہے۔ ڈیزائن کے انتخاب میں سے ایک یہ ہے کہ کسی بھی بائیں چائلڈ لنک کو کہاں رکھنا ہے۔ اسے صرف سلسلہ میں پہلے یا آخری ڈپلیکیٹ پر جانا چاہئے تاکہ الگورتھم کو معلوم ہو کہ اسے کہاں تلاش کرنا ہے۔ اعداد و شمار دو انتخاب کی وضاحت کرتا ہے۔ نئی ڈپلیکیٹ چابیاں سلسلہ کے مخالف سرے پر ڈالی جانی چاہئیں۔



تصویر 26-8بائنری سرچ ٹریز میں ڈپلیکیٹ کیز

ایک اور انتخاب یہ ہے کہ ڈپلیکیٹس والی کلید کے لیے ()find__اور ()searchطریقوں سے کیا لوٹنا ہے ۔ کیا اسے پہلی یا آخری واپس کرنی چاہئے؟ انتخاب اس کے مطابق بھی ہونا چاہئے کہ کس نوڈ کو حذف کیا جاتا ہے اور حذف ()طریقہ سے واپس کیا جاتا ہے۔ اگر انہیں پہلے داخل کیا جاتا ہے اور پہلے سے ہٹا دیا جاتا ہے، تو حذف() ڈوپلی کیٹ نوڈس کے لیے منی اسٹیک کی طرح کام کرے گا۔

حذف کرنے کا عمل اس حقیقت سے پیچیدہ ہے کہ ہر ڈپلیکیٹ نوڈس پر مختلف ڈیٹا ویلیوز کو محفوظ کیا جا سکتا ہے۔ کال کرنے والے کو ڈپلیکیٹ کلید والے کسی نوڈ کے بجائے مخصوص ڈیٹا والے نوڈ کو حذف کرنے کی ضرورت پڑسکتی ہے۔ جو بھی اسکیم منتخب کی گئی ہے، حذف کرنے کے معمول کو اس بات کو یقینی بنانے کی ضرورت ہوگی کہ بائیں ذیلی درخت، اگر کوئی ہے، مناسب جگہ سے منسلک رہے۔

کسی بھی قسم کی ڈپلیکیٹ چابیاں کے ساتھ، درخت کو متوازن کرنا مشکل یا ناممکن ہو جاتا ہے۔ ڈپلیکیٹس کی زنجیریں اضافی سطحیں شامل کرتی ہیں جنہیں توازن میں مدد کے لیے دوبارہ ترتیب نہیں دیا جا سکتا۔ اس کا مطلب ہے کہ آئٹم کو تلاش کرنے کی کارکردگی O(log N)کے بہترین کیس سے O(N)کی طرف بڑھ جاتی ہے۔

جیسا کہ آپ دیکھ سکتے ہیں، ڈپلیکیٹ کیز کو اجازت دینا ڈیٹا ڈھانچے میں کوئی آسان اضافہ نہیں ہے۔
ڈیٹا کے دوسرے ڈھانچے میں، ڈپلیکیٹ کیز چیلنجز پیش کرتی ہیں، لیکن یہ سب بائنری سرچ ٹری کی طرح مشکل نہیں ہیں۔

BinarySearchTreeTester.pyپروگرام

یہ ہمیشہ ایک اچھا خیال ہے کہ کوڈ ماڈیول کے کام کاج کو ٹیسٹ لکھ کر جانچیں جو ہر آپریشن کو استعمال کرتے ہیں۔ ٹیسٹوں کا ایک جامع سیٹ لکھنا اپنے آپ میں ایک فن ہے۔ ایک اور کارآمد حکمت عملی ایک انٹرایکٹو ٹیسٹ پروگرام لکھنا ہے جو آپ کو مختلف ترتیبوں اور مختلف دلائل کے ساتھ آپریشنز کی ایک سیریز کو آزمانے کی اجازت دیتا ہے۔ دکھائے گئے تمام BinarySearchTreeکلاس میتھ odsکو جانچنے کے لیے، آپ BinarySearchTreeTester.pyجیسا پروگرام استعمال کر سکتے ہیں جو فہرست 12-8میں دکھایا گیا ہے۔

فہرست سازی BinarySearchTreeTester.py 12-8پروگرام

```
# BinarySearchTree امپورٹ سے انٹرایکٹو طریقے سے BinarySearchTreeکلاس کی جانچ کریں *

theTree = BinarySearchTree()                    #خالی درخت سے شروع کریں۔

"2006 2") theTree.insert("Tim", "2016 1")       #کچھ ڈیٹا داخل کریں۔
        "2006 1") theTree.insert("Vint",
      " 2002 3") theTree.insert("Fran",
      "2002 3") theTree.insert("Ron",
      "1996 1") theTree.insert("Adi",
    "1994 1") theTree.insert("Amir",
    "1988 1" ") theTree.insert("Raj",
    "1979 1") theTree.insert("Ivan",
      "1975 2") theTree.insert("Ken",
    "1974 1") theTree.insert("Herb",
        theTree.insert("Don",

def print_commands(نام): پرنٹ ('ممکنہ کمانڈز ہیں'، نام)   #ممکنہ کمانڈز کی فہرست پرنٹ کریں۔
```

#درخت کے تمام نوڈس کو ہٹا دیں۔ def clearTree():

جبکہ theTree.isEmpty(): نہیں

ڈیٹا، کلید theTree.delete(key) = theTree.root()

def traverseTree(traverseType="in"): # Tree.traverse(traverseType)میں کلید، ڈیٹا کے لیے تمام نوڈس کو عبور اور پرنٹ کریں :

پرنٹ کریں

کمانڈز # [ ] =کمانڈ کے نام، افعال، اور ان کے پیرامیٹرز

['print', theTree.print, [], ['insert', ('key', 'data')], ['delete', theTree.delete, ('key',)],
['quit', none, [], ,'تلاش'][], ['help', print_commands, []], ['?', print_commands,
clearTree, []], ['traverse', traverseTree, ('type',)], ['clear',
theTree.search, ('key',)],

]

#ایک فہرست میں کمانڈ کے تمام نام جمع کریں۔

command_names = ".".join(c[0] for c in commands) for i in range(len(commands)):

#دلیل کی فہرست میں کمانڈ کے نام رکھیں

اگر کمانڈز[i] == [1]پرنٹ_کمانڈز: پرنٹ_کمانڈز کا #

کمانڈز[i][2] = [command_names]

#کمانڈ کے نام کے پہلے حرف کو #کمانڈ کی تفصیلات (نام، فنکشن، پیرامیٹرز/آرگس) کمانڈ_ڈکٹ dict((c[0][0], c) =میں cکے لیے ایک لغت بنائیں

#انٹرایکٹو لوپ کے لیے معلومات پرنٹ کریں۔

print_commands(command_names)
theTree.print()
' '
جواب =

#صارف سے کمانڈ حاصل کرنے کے لیے لوپ کریں اور اس پر عمل کرتے ہوئے جواب دیں :'q' != [0]

پرنٹ ('درخت ہے', theTree.nodes(), 'نوڈس پار'،
theTree.levels(), 'levels')
if len(ans) == 0: ()rewol.("کمانڈ کا پہلا حرف درج کریں:")tupnians =

' '
جواب =

اگر جواب[0] کمانڈ_ڈکٹ میں :

نام، فنکشن، پیرامیٹرز =کمانڈ_ڈکٹ[جواب[0]]

اگر فنکشن کوئی نہیں ہے:

```
= :(پیرامیٹر، فہرست)ecnatsnisiprint(name) if
                                    پیرامیٹرزarguments

دوسری:

                    param پیرامیٹرز میںarguments = []
                                  کے لیے : (arg)arguments.append
                    arg = input("Enter " "command:")

کوشش کریں: نتیجہ =فنکشن (*دلائل) پرنٹ ('نتیجہ:'،
نتیجہ)

استثناء کے علاوہ بطور ای: پرنٹ ('استثنیٰ واقع ہوا') پرنٹ
(ای)

دوسری:

پرنٹ ("غلط کمانڈ: '"جواب، (""
```

یہ پروگرام صارفین کو ٹرمینل انٹرفیس میں ٹائپ کرکے کمانڈ داخل کرنے کی اجازت دیتا ہے۔ یہ سب سے پہلے BinarySearchTree ماڈیول درآمد کرتا ہے اور اس کے ساتھ ایک خالی درخت بناتا ہے۔ پھر یہ کچھ تاروں کے ساتھ ناموں کو جوڑنے کے لیے ()insertکا استعمال کرتے ہوئے اس میں کچھ ڈیٹا رکھتا ہے ۔ نام وہ چابیاں ہیں جو درخت کے اندر نوڈس رکھنے کے لیے استعمال ہوتی ہیں۔

ٹیسٹر تمام ممکنہ کمانڈز کو پرنٹ کرنے، درخت سے تمام نوڈس کو صاف کرنے، اور ہر نوڈ کو پرنٹ کرنے کے لیے درخت کو عبور کرنے کے لیے متعدد افادیت کے افعال کی وضاحت کرتا ہے۔ یہ فنکشن نیچے کمانڈ لوپ میں کمانڈز کو ہینڈل کرتے ہیں۔

ٹیسٹر پروگرام کا اگلا حصہ کمانڈز کی فہرست کی وضاحت کرتا ہے۔ ہر ایک کے لیے، اس کا ایک نام، کمانڈ پر عمل کرنے کے لیے ایک فنکشن، اور دلائل یا پیرامیٹرز کی فہرست یا ٹوپل ہے۔
یہ ہم نے اب تک دکھایا ہے اس سے کہیں زیادہ جدید Pythonکوڈ ہے، لہذا یہ تھوڑا سا عجیب لگ سکتا ہے۔ نام وہ ہیں جو صارف ٹائپ کرے گا (یا کم از کم ان کا پہلا حرف)، اور فنکشنز یا تو درخت کے طریقے ہیں یا ٹیسٹر میں بیان کردہ یوٹیلیٹی فنکشنز۔ صارف کے کمانڈ کا انتخاب کرنے کے بعد دلائل اور پیرامیٹرز پر کارروائی کی جائے گی۔

تھوڑی کمانڈ لائن مدد فراہم کرنے کے لیے، ٹیسٹر کمانڈ کے ناموں کی فہرست کو ایک سٹرنگ میں جوڑتا ہے، انہیں کوما سے الگ کرتا ہے۔ یہ آپریشن جوائن ()سٹرنگ کے طریقہ سے مکمل کیا جاتا ہے۔ ہر کمانڈ کے نام کے درمیان جو متن رکھنا ہے وہ سٹرنگ ہے (کوما اور اسپیس)، اور جوائن کرنے کی دلیل ناموں کی فہرست ہے۔ پروگرام کمانڈز میں کمانڈ کی وضاحتوں کے ذریعے اعادہ کرنے کے لیے فہرست کی سمجھ کا استعمال کرتا ہے اور پہلا عنصر نکالتا ہے، جو کہ کمانڈ کا نام ہے: _(".join(c[0] for c in commands)، "نتیجہ کمانڈ_نام متغیر میں محفوظ ہے۔

پھر کمانڈ کے ناموں کی مربوط سٹرنگ کو print_commandsفنکشن کے لیے دلیل کی فہرست میں داخل کرنے کی ضرورت ہے ۔ یہ لوپ کے لئے میں کیا گیا ہے . دو اندراجات میں print_commandsفنکشن ہے: مدد اور ؟ احکامات

کمانڈ لوپ کے لیے تیاری کا آخری حصہ ایک لغت، کمانڈ_ڈکٹ بناتا ہے، جو ہر کمانڈ کے پہلے کردار کو کمانڈ کی تفصیلات میں نقشہ بناتا ہے۔ آپ کے پاس نہیں ہے۔

ابھی تک اس Pythonڈیٹا کا ڈھانچہ استعمال کیا ہے۔ باب 11میں، "بیش ٹیبلز،" آپ دیکھتے ہیں کہ وہ کیسے کام کرتے ہیں، لہذا اگر آپ ان سے واقف نہیں ہیں، تو ان کے بارے میں سوچیں کہ انٹیجر کے بجائے ایک سٹرنگ کے ذریعہ ترتیب کردہ ایک سرنی۔ آپ صف میں قدریں تفویض کر سکتے ہیں اور پھر انہیں تیزی سے تلاش کر سکتے ہیں۔ ٹیسٹر پروگرام میں، ['p']command_dictکا جائزہ لینے سے پرنٹ کمانڈ، یعنی [] ['print', theTree.print,]کے لیے تفصیلات واپس آ جائیں گی ۔ وہ تصریحات کومپیکٹ (لیکن خفیہ) فہم کا استعمال کرتے ہوئے لغت میں محفوظ ہو جاتی ہیں: dict((c[0][0], c) for c in command_dict.values())کمانڈز میں cکے لیے)۔

باقی ٹیسٹر کمانڈ لوپ کو نافذ کرتا ہے۔ یہ پہلے درخت کو ٹر منل پر پرنٹ کرتا ہے، اس کے بعد کمانڈز کی فہرست۔ جواب متغیر صارف کے ذریعہ ٹائپ کردہ ان پٹ رکھتا ہے۔ یہ ایک اسپیس میں شروع ہو جاتا ہے تاکہ کمانڈ لوپ شروع ہو اور ایک نئی کمانڈ کا اشارہ کرے۔

کمانڈ لوپ اس وقت تک جاری رہتا ہے جب تک کہ صارف quitکمانڈ کو طلب نہیں کرتا، جو qسے شروع ہوتا ہے۔ لوپ باڈی کے اندر، درخت میں نوڈس اور لیولز کی تعداد پرنٹ کی جاتی ہے، اور پھر صارف سے کمانڈ کے لیے کہا جاتا ہے۔ ()inputکے ذریعے واپس آنے والی سٹرنگ کو کمانڈ کی تلاش کو آسان بنانے کے لیے چھوٹے حروف میں تبدیل کیا جاتا ہے۔ اگر صارف نے صرف واپسی کو دبایا، تو اسٹرنگ میں کوئی پہلا حرف نہیں ہوگا، لہذا آپ ایک کو بھریں گے ؟ پہلے سے طے شدہ جواب بنانے کے لیے تمام کمانڈ کے ناموں کو دوبارہ پرنٹ کرنا ہے۔

اگلے بیان میں — if ans[0] in command_dict: —ٹیسٹر چیک کرتا ہے کہ آیا صارف کے جواب میں پہلا کردار معلوم کمانڈز میں سے ایک ہے۔ اگر کریکٹر کو recognizedکیا جاتا ہے، تو یہ کمانڈ_ڈکٹ میں محفوظ کردہ تفصیلات سے نام، فنکشن اور پیرامیٹرز نکالتا ہے۔ اگر عمل کرنے کے لیے کوئی فنکشن ہے، تو اس پر کارروائی کی جائے گی۔ اگر نہیں، تو صارف نے چھوڑنے کو کہا، اور whileلوپ باہر نکل جائے گا۔ جب صارف کے جواب کا پہلا کردار کسی کمانڈ سے میل نہیں کھاتا ہے، تو ایک ایرر میسج پرنٹ کیا جاتا ہے، اور لوپ ایک نئی کمانڈ کا اشارہ کرتا ہے۔

کمانڈ کی تفصیلات ملنے کے بعد، اسے یا تو صارف کو فنکشن کو کال کرتے وقت استعمال کرنے کے لیے دلائل کے لیے اشارہ کرنے کی ضرورت ہوتی ہے یا انھیں تفصیلات سے حاصل کرنا ہوتا ہے۔ یہ انتخاب اس بات پر مبنی ہے کہ آیا پیرامیٹرز کو Python tupleیا فہرست کے طور پر بیان کیا گیا تھا۔ اگر یہ ٹوپل ہے تو، ٹیپل کے عناصر پیرامیٹرز کے نام ہیں ۔ اگر یہ فہرست ہے، تو فہرست میں فنکشن کے دلائل ہوتے ہیں۔ ٹیپلز کے لیے، صارف کو ہر دلیل کو نام کے ساتھ درج کرنے کے لیے کہا جاتا ہے، اور جوابات دلائل کی فہرست میں محفوظ کیے جاتے ہیں۔ دلائل کے تعین کے بعد ، کمانڈ لوپ رزلٹ =فنکشن(*دلائل) کا استعمال کرتے ہوئے آرگیومینٹس کی فہرست کے ساتھ فنکشن کو کال کرنے کی کوشش کرتا ہے ۔ دلائل سے پہلے ستارہ (*)ضرب کیشن آپریٹر نہیں ہے۔ اس کا مطلب ہے کہ دلائل کی فہرست کو فنکشن کے لیے پوزیشنی دلائل کی فہرست کے طور پر استعمال کیا جانا چاہیے۔ اگر فنکشن کوئی مستثنیات اٹھاتا ہے، تو وہ پکڑے جاتے ہیں اور دکھائے جاتے ہیں۔ دوسری صورت میں، فنکشن کا نتیجہ دوسری کمانڈ حاصل کرنے کے لئے لوپ کرنے سے پہلے پرنٹ کیا جاتا ہے۔

ٹیسٹر کو استعمال کرنے کی کوشش کریں چار اہم آپریشنز: تلاش کریں، داخل کریں، عبور کریں، اور حذف کریں۔
حذف کرنے کے لیے، اثر دیکھنے کے لیے 0، 1، اور 2چائلڈ نوڈس والے نوڈس کو حذف کرنے کی کوشش کریں۔ جب آپ 2بچوں کے ساتھ نوڈ کو حذف کرتے ہیں، تو اندازہ لگائیں کہ کون سا جانشین نوڈ حذف شدہ نوڈ کی جگہ لے گا اور دیکھیں کہ آیا آپ صحیح ہیں۔

ہف مین کوڈ

آپ کو یہ خیال نہیں آنا چاہئے کہ بائنری درخت ہمیشہ تلاش کے درخت ہوتے ہیں۔ بہت سے بائنری درخت دوسرے طریقوں سے استعمال ہوتے ہیں۔
شکل 16-8ایک مثال دکھاتی ہے جہاں ایک بائنری درخت ایک الجی بریک اظہار کی نمائندگی کرتا ہے۔ اب ہم ایک الگورتھم پر بات کرتے ہیں جو ڈیٹا
کو کمپریس کرنے کے لیے حیران کن انداز میں بائنری ٹری کا استعمال کرتا ہے۔ اسے ہف مین کوڈ کہتے ہیں، ڈیوڈ ہف مین کے بعد، جس نے اسے
1952میں دریافت کیا تھا۔ ڈیٹا کمپریشن بہت سے حالات میں اہم ہوتا ہے۔ ایک مثال انٹرنیٹ پر یا ڈیجیٹل نشریات کے ذریعے ڈیٹا بھیجنا ہے،
جہاں معلومات کو اس کی مختصر ترین شکل میں بھیجنا ضروری ہے۔ ڈیٹا کو کمپریس کرنے کا مطلب ہے کہ ایک ہی وقت میں بینڈوڈتھ کی حد
کے تحت مزید ڈیٹا بھیجا جا سکتا ہے۔

کریکٹر کوڈز ایک غیر کمپریسڈ ٹیکسٹ فائل میں ہر کریکٹر کو کمپیوٹر میں ایک سے چار بائٹس کے ذریعے دکھایا جاتا ہے، اس پر منحصر ہے کہ

حروف کو انکوڈ کیا جاتا ہے۔ قابل احترام ASCIIکوڈ کے لیے، صرف ایک بائٹ استعمال کیا جاتا ہے، لیکن یہ حروف کی حد کو محدود کر دیتا ہے جن
کا اظہار 128سے کم ہو سکتا ہے۔ دنیا کی تمام زبانوں کے علاوہ دیگر علامتوں جیسے ایموجیز کا حساب کتاب کرنے کے لیے، مختلف یونی کوڈ
معیارات تک استعمال کرتے ہیں ۔ چار بائٹس فی کریکٹر۔ اس بحث کے لیے، ہم فرض کرتے ہیں کہ صرف ASCIIحروف کی ضرورت ہے، اور ہر کردار
ایک بائٹ (یا آٹھ بٹس) لیتا ہے۔

جدول 8-2دکھاتا ہے کہ ASCIIکوڈ کا استعمال کرتے ہوئے بائنری میں کچھ حروف کی نمائندگی کیسے کی جاتی ہے۔

جدول 8-2کچھ ASCIIکوڈز

بائنری	کوشاریہ	
01000000	64	@
01000001	65	اے
01000010	66	بی
...	...	...
01011001	89	Y
01011010	90	Z
...	...	...
01100001	97	a
01100010	98	ب

ڈیٹا کو کمپریس کرنے کے کئی طریقے ہیں۔ متن کے لیے، سب سے عام نقطہ نظر بٹس کی تعداد کو کم کرنا ہے جو سب سے زیادہ استعمال ہونے والے
حروف کی نمائندگی کرتے ہیں۔ نتیجے کے طور پر، ہر کردار بٹس کے "سٹریم" میں بٹس کی متغیر تعداد لیتا ہے جو مکمل متن کی نمائندگی کرتا ہے۔

انگریزی میں، Eاور Tبہت عام حروف ہیں، جب نثر اور دوسرے شخص سے فرد کے مواصلات کا جائزہ لیتے ہیں اور خالی جگہوں اور اوقاف جیسی چیزوں کو نظر انداز کرتے ہیں۔ اگر آپ ایسی اسکیم کا انتخاب کرتے ہیں جس میں E، Tاور دوسرے عام حروف کو لکھنے کے لیے صرف چند بٹس استعمال کیے جاتے ہیں ، تو یہ اس سے زیادہ کمپیکٹ ہونا چاہیے اگر آپ ہر حرف کے لیے یکساں بٹس استعمال کرتے ہیں۔ سپیکٹرم کے دوسرے سرے پر، Qاور Zشاذ و نادر ہی ظاہر ہوتے ہیں، اس لیے ان حروف کے لیے کبھی کبھار بڑی تعداد میں بٹس استعمال کرنا اتنا برا نہیں ہے۔

فرض کریں کہ آپ Eکے لیے صرف دو بٹس استعمال کرتے ہیں —کہئے ۔01آپ انگریزی حروف تہجی کے ہر حرف کو دو بٹس میں انکوڈ نہیں کر سکتے کیونکہ صرف چار 2بٹ کے امتزاج ہیں: ،10 ،01 ،00اور ۔11
کیا آپ ان چار مجموعوں کو سب سے زیادہ استعمال ہونے والے چار حروف کے لیے استعمال کر سکتے ہیں؟ ٹھیک ہے، اگر آپ نے ایسا کیا، اور آپ اب بھی کم استعمال شدہ حروف کے لیے کچھ انکوڈنگ کرنا چاہتے ہیں، تو آپ کو پریشانی ہوگی۔ بٹس کی تشریح کرنے والے الگورتھم کو کسی نہ کسی طرح یہ اندازہ لگانا ہوگا کہ بٹس کا جوڑا ایک کریکٹر ہے یا کچھ لمبے کریکٹر کوڈ کا حصہ ہے۔

انکوڈنگ میں اہم خیالات میں سے ایک یہ ہے کہ ہمیں کوڈ کی قدروں میں سے کچھ کو اشارے کے طور پر ایک طرف رکھ دینا چاہیے جو کم استعمال شدہ کردار کو انکوڈ کرنے کے لیے تھوڑی لمبی سٹرنگ کی پیروی کرتی ہے۔ الگورتھم کو کسی خاص لمبائی کی تھوڑی سی تار کو دیکھنے اور اس بات کا تعین کرنے کے لیے ایک طریقہ کی ضرورت ہوتی ہے کہ آیا یہ حروف میں سے کسی ایک کے لیے مکمل کوڈ ہے یا لمبے کوڈ کی قدر کے لیے صرف ایک سابقہ ہے۔ آپ کو محتاط رہنا چاہیے کہ کسی بھی کردار کی نمائندگی اسی بٹ کے امتزاج سے نہیں ہوتی ہے جو کسی دوسرے کردار کے لیے استعمال ہونے والے لمبے کوڈ کے شروع میں ظاہر ہوتا ہے۔ مثال کے طور پر، اگر 01 Eہے، اور 01011000 Zہے، تو 01011000کو ڈی کوڈنگ کرنے والا الگورتھم یہ نہیں جانتا کہ ابتدائی 01 Eکی نمائندگی کرتا ہے یا Zکا آغاز۔ یہ ایک اصول کی طرف جاتا ہے: کوئی کوڈ کسی کا سابقہ نہیں ہو سکتا۔ دوسرے کوڈ۔

اس بات پر بھی غور کریں کہ کچھ پیغامات میں، ممکن ہے Eسب سے زیادہ استعمال ہونے والا حرف نہ ہو۔ اگر متن ایک پروگرام سورس فائل ہے، مثال کے طور پر، رموز اوقاف کے حروف جیسے بڑی آنت ،(:)سیمی کالون ،(؛)اور انڈر سکور E (_)کی نسبت زیادہ کثرت سے ظاہر ہو سکتے ہیں۔ اس مسئلے کا حل یہ ہے: ہر پیغام کے لیے، آپ اس مخصوص پیغام کے مطابق ایک نیا کوڈ بناتے ہیں۔ فرض کریں کہ آپ SPAM EGG + SPAM SPAMپیغام بھیجنا چاہتے ہیں ۔ حرف Sبہت زیادہ ظاہر ہوتا ہے، اور اسی طرح خلائی کردار بھی۔ ہو سکتا ہے آپ ایک ٹیبل بنانا چاہیں جس میں دکھایا جائے کہ ہر حرف کتنی بار ظاہر ہوتا ہے۔ اسے فریکوئنسی ٹیبل کہا جاتا ہے، جیسا کہ جدول 3-8میں دکھایا گیا ہے۔

سپیم پیغام کے لیے جدول 3-8فریکوئنسی ٹیبل

شمار	کردار			
4	E			
4	A			
5	SPACE			
1	+			

سب سے زیادہ گنتی والے حروف کو تھوڑی تعداد میں بٹس کے ساتھ کوڈ کیا جانا چاہیے۔
جدول 4-8ایک طریقہ دکھاتا ہے کہ آپ سپام پیغام میں حروف کو کیسے انکوڈ کر سکتے ہیں۔

آپ اسپیس کے لیے 01استعمال کر سکتے ہیں کیونکہ یہ سب سے زیادہ بار بار آتا ہے۔ اگلے سب سے زیادہ متواتر کردار ادا کرنے والے A، P، Sاور Mہیں، ہر ایک چار بار ظاہر ہوتا ہے۔ آپ آخری کوڈ 00استعمال کرتے ہیں، ۔Mباقی کوڈز 00یا 01سے شروع نہیں ہو سکتے کیونکہ اس سے اصول ٹوٹ جائے گا۔

کہ کوئی کوڈ کسی دوسرے کوڈ کا سابقہ نہیں ہو سکتا۔ یہ 10اور 11کو دوسرے حروف کے لیے بطور سابقہ استعمال کرنے کے لیے چھوڑ دیتا ہے۔

سپیم پیغام کے لیے جدول 4-8ہف مین کوڈ

کوڈ	شمار	کردار	کوڈ			
110	4	پی	11			
101	4	ایس		10000	1	ای
01	5	خلا		1001	2	جی
10001	1	+		00	4	ایم

3بٹ کوڈ کے امتزاج کے بارے میں کیا خیال ہے؟ آٹھ امکانات ہیں: ،110 ،101 ،100 ،011 ،010 ،001 ،000اور ،111لیکن آپ پہلے ہی جانتے ہیں کہ آپ 00یا 01سے شروع ہونے والی کوئی بھی چیز استعمال نہیں کر سکتے۔ اس سے چار امکانات ختم ہو جاتے ہیں۔ آپ ان میں سے کچھ 3بٹ کوڈز کو اگلے اکثر آنے والے حروف کو تفویض کر سکتے ہیں، Sبطور 101، Pبطور 110، Aاور بطور ۔111یہ باقی حروف کے لیے استعمال کرنے کے لیے سابقہ 100چھوڑ دیتا ہے۔ آپ 4بٹ کوڈ، 1001اگلے سب سے زیادہ بار بار آنے والے کردار، جی کے لیے استعمال کرتے ہیں، جو دو بار ظاہر ہوتا ہے۔ دو حروف ہیں جو صرف ایک بار ظاہر ہوتے ہیں، Eاور ۔+ان کو 5بٹ کوڈز، 10000اور 10001کے ساتھ انکوڈ کیا گیا ہے۔

اس طرح، پورے پیغام کو بطور کوڈ کیا جاتا ہے۔

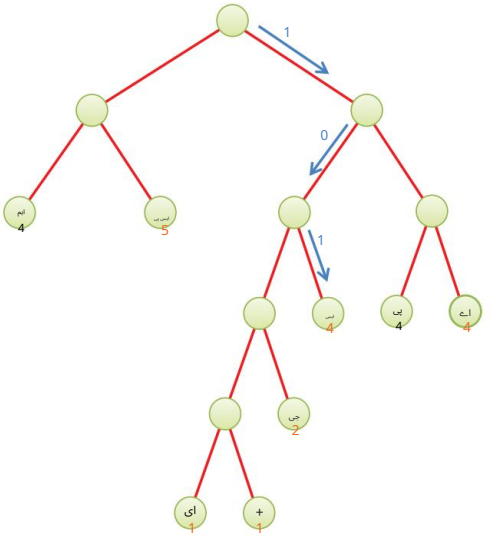
101 110 111 00 01 101 110 111 00 01 10000 1001 1001 01 10001 01
101 110 111 00

واضح ہونے کے لیے، ہم اس پیغام کو انفرادی حروف کے کوڈز میں توڑ کر دکھاتے ہیں۔ یقینا، تمام بٹس ایک ساتھ چلیں گے کیونکہ بائنری میس سیج میں کوئی اسپیس کریکٹر نہیں ہے، صرف 0sاور ۔1sاس سے یہ معلوم کرنا زیادہ مشکل ہو جاتا ہے کہ کون سے بٹس کسی کردار سے مطابقت رکھتے ہیں۔ تاہم، اہم نکتہ یہ ہے کہ ان پٹ پیغام میں 25حروف، جو عام طور پر میموری میں 200بٹس (25 × 8)میں محفوظ ہوں گے، ہف مین کوڈنگ میں صرف 72بٹس کی ضرورت ہوتی ہے۔

ہف مین ٹری کے ساتھ ڈی کوڈنگ ہم بعد میں دکھاتے ہیں کہ ہف مین کوڈز کیسے بنتے ہیں۔ پہلے، آئیے ڈی کوڈنگ کے کسی حد تک آسان پروسیس کا جائزہ لیں۔ فرض کریں کہ آپ کو پچھلے حصے میں دکھائے گئے بٹس کی تار موصول ہوئی ہے۔

آپ اسے دوبارہ کرداروں میں کیسے تبدیل کریں گے؟ آپ ایک قسم کا بائنری درخت استعمال کر سکتے ہیں جسے ہف مین ٹری کہتے ہیں۔ تصویر SPAM 27-8پیغام کے لیے ہف مین ٹری کو دکھاتا ہے جس پر ابھی ابھی بات ہوئی ہے۔

پیغام کے حروف درخت میں لیف نوڈس کے طور پر ظاہر ہوتے ہیں۔ پیغام میں ان کی فریکوئنسی جتنی زیادہ ہوگی، درخت میں وہ اتنے ہی اوپر دکھائی دیں گے۔ ہر لیف نوڈ کے باہر کا نمبر اس کی فریکوئنسی ہے۔ یہ خلائی کردار (sp)کو دوسرے درجے پر رکھتا ہے، اور S، P، Aاور Mحروف کو دوسرے یا تیسرے درجے پر رکھتا ہے۔ سب سے کم بار بار، Eاور ،+سب سے نچلی سطح پر ہیں، ۔5



پیغام: SPAM SPAM SPAM EGG + SPAM

کریکٹر	کاؤنٹ	کوڈ
پی		4 110
س		4 101
خلا		5 01
+		1 10001

اے		4 111
ای		10000
جی		2 1001
ایم		4 00

تصویر 27-8سپیم پیغام کے لیے ہف مین ٹری

پیغام کو ڈی کوڈ کرنے کے لیے آپ اس درخت کو کیسے استعمال کرتے ہیں؟ آپ پیغام کے پہلے بٹ کو دیکھ کر شروع کرتے ہیں اور درخت کے جڑ نوڈ پر ایک پوائنٹر سیٹ کرتے ہیں۔ اگر آپ کو 0بٹ نظر آتا ہے، تو آپ پوائنٹر کو نوڈ کے بائیں چائلڈ کی طرف لے جاتے ہیں، اور اگر آپ کو 1بٹ نظر آتا ہے، تو آپ اسے دائیں منتقل کرتے ہیں۔ اگر identi fiedنوڈ میں کوئی منسلک کریکٹر نہیں ہے، تو آپ پیغام میں اگلے بٹ پر جائیں گے۔ اسے Sکے کوڈ کے ساتھ آزمائیں ، جو کہ 101ہے۔ آپ دائیں، بائیں، پھر دائیں پھر جائیں، اور ،voilaآپ اپنے آپ کو Sنوڈ پر پائیں گے۔ یہ شکل 27-8میں نیلے تیروں سے دکھایا گیا ہے۔

آپ دوسرے کرداروں کے ساتھ بھی ایسا ہی کر سکتے ہیں۔ لیف نوڈ پر پہنچنے کے بعد، آپ اس کے کریکٹر کو ڈی کوڈ شدہ سٹرنگ میں شامل کر سکتے ہیں اور پوائنٹر کو روٹ نوڈ پر واپس لے جا سکتے ہیں۔
اگر آپ میں صبر ہے تو، آپ اس طرح پوری بٹ سٹرنگ کو ڈی کوڈ کر سکتے ہیں۔

ہف مین ٹری بنانا آپ نے دیکھا ہے کہ ڈی کوڈنگ کے لیے ہف مین ٹری کا استعمال کیسے کیا جاتا ہے، لیکن آپ اس درخت کو کیسے بناتے ہیں؟

اس مسئلے سے نمٹنے کے بہت سے طریقے ہیں۔ آپ کو ہف مین ٹری آبجیکٹ کی ضرورت ہے، اور یہ کچھ اس طرح ہے جیسا کہ پہلے بیان کیا گیا ہے کہ اس میں نوڈس ہیں جن میں دو چائلڈ نوڈس ہیں ۔ تاہم، یہ بالکل مختلف ہے، کیونکہ معمولات جو تلاش کے درختوں میں کلیدوں کے لیے مخصوص ہیں، جیسے ,()insert ,()findاور ,()deleteمتعلقہ نہیں ہیں۔ یہ پابندی کہ نوڈ کی کلید اس کے بائیں بچے کی کسی بھی کلید سے بڑی ہو اور اس کے دائیں بچے کی کسی بھی کلید کے برابر یا اس سے کم ہو، ہفمین درخت پر لاگو نہیں ہوتا ہے۔ آئیے نئی کلاس HuffmanTreeکو کال کریں، اور سرچ ٹری کی طرح، ہر نوڈ پر ایک کلید اور ایک قدر ذخیرہ کریں۔ کلید ڈی کوڈ شدہ میسج کریکٹر جیسے Sیا Gکو رکھے گی۔ یہ اسپیس کریکٹر ہو سکتا ہے، جیسا کہ آپ نے دیکھا ہے، اور اسے "کوئی کریکٹر" کے لیے ایک خاص قدر کی ضرورت ہے۔

میسج سٹرنگ سے ہف مین ٹری بنانے کا الگورتھم یہ ہے:

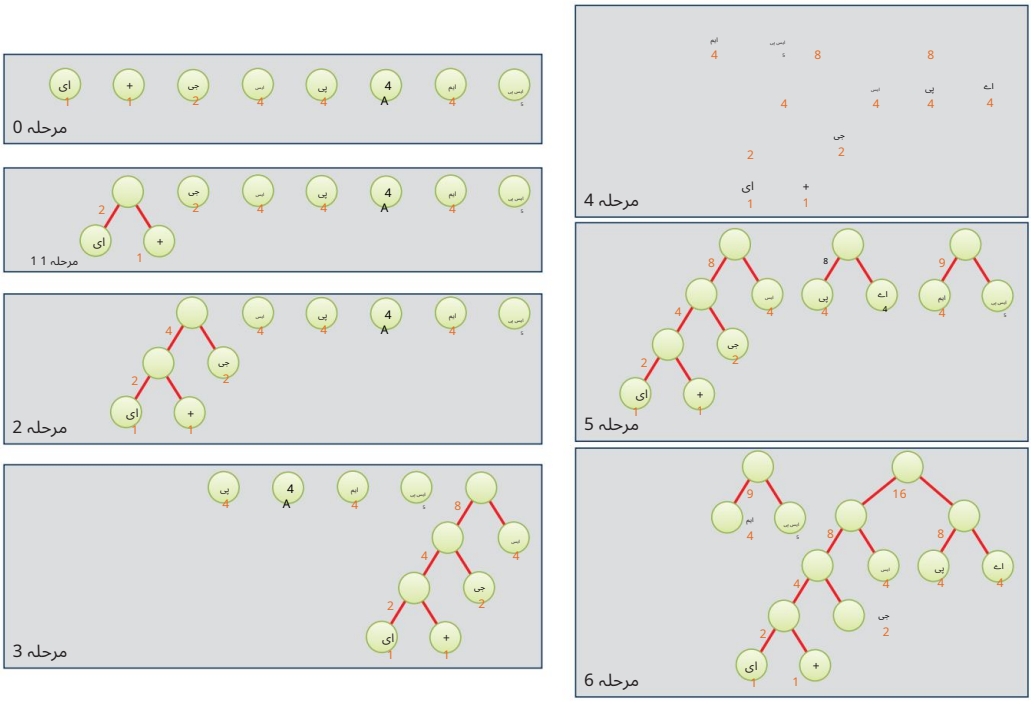
تیاری

.1شمار کریں کہ پیغام کی تار میں ہر حرف کتنی بار ظاہر ہوتا ہے۔

.2پیغام میں استعمال ہونے والے ہر کردار کے لیے HuffmanTreeآبجیکٹ بنائیں ۔ سپیم کے لیے
پیغام کی مثال، وہ آٹھ درخت ہوں گے۔ ہر درخت میں ایک نوڈ ہوتا ہے جس کی کلید ایک کریکٹر ہے اور جس کی قیمت پیغام میں
اس کریکٹر کی فریکوئنسی ہے۔ وہ valuesٹیبل 3-8یا ٹیبل 4-8میں اسپام پیغام کے لیے مل سکتے ہیں۔

.3ان درختوں کو ترجیحی قطار میں داخل کریں (جیسا کہ باب 4میں بیان کیا گیا ہے)۔ انہیں فریکوئنسی (ہر روٹ نوڈ کی قدر کے طور
پر ذخیرہ کیا جاتا ہے) اور درخت میں سطحوں کی تعداد کے حساب سے ترتیب دیا جاتا ہے۔ سب سے چھوٹی فریکوئنسی والا درخت
سب سے زیادہ ترجیح رکھتا ہے۔ مساوی تعدد والے درختوں میں، زیادہ درجے والے درخت کو سب سے زیادہ ترجیح دی جاتی ہے۔
دوسرے لفظوں میں، جب آپ ترجیحی قطار سے کسی درخت کو ہٹاتے ہیں، تو یہ ہمیشہ سب سے کم استعمال شدہ کردار کا سب
سے گہرا درخت ہوتا ہے۔ (درخت کی گہرائی کا استعمال کرتے ہوئے تعلقات کو توڑنا، حتمی Huffmanدرخت کے توازن کو بہتر بناتا
ہے۔)

یہ تیاری مکمل کرتا ہے، جیسا کہ شکل 28-8کے مرحلہ 0میں دکھایا گیا ہے۔ ہر ایک نوڈ ہف مین ٹریز میں نوڈ کے بیچ میں ایک کریکٹر
دکھایا گیا ہے اور نوڈ کے نیچے اور بائیں طرف ایک فریکوئنسی ویلیو دکھائی گئی ہے۔



تصویر 28-8ہف مین درخت کو اگانا، پہلے چھ مراحل

پھر درج ذیل کریں:

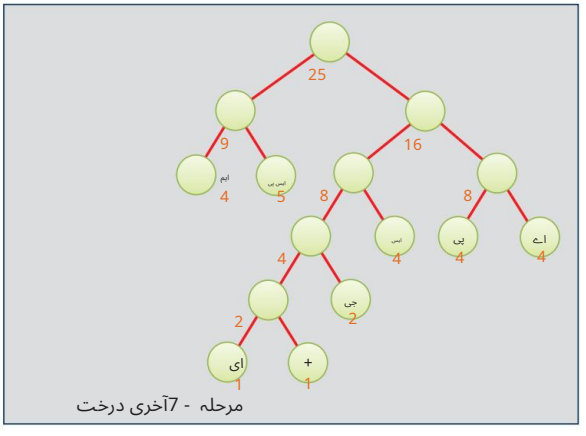
درخت کا استحکام

.1ترجیحی قطار سے دو درختوں کو ہٹا دیں اور انہیں نئے بچوں میں بنائیں
نوڈ نئے نوڈ کی فریکوئنسی ویلیو ہے جو بچوں کے فریکوئنسی کا مجموعہ ہے۔ اس کی کریکٹر کلید کو خالی چھوڑا جا سکتا ہے (کسی کریکٹر کے لیے
خصوصی قدر، نہ کہ اسپیس کریکٹر)۔

.2اس نئے، گہرے درخت کو دوبارہ ترجیحی قطار میں داخل کریں۔

.3مرحلہ 1اور 2کو دہراتے رہیں۔ درخت بڑے اور بڑے ہوتے جائیں گے، اور وہاں ہوں گے۔
ان میں سے کم اور کم. جب ترجیحی قطار میں صرف ایک درخت باقی رہ جاتا ہے، تو وہ ہے ہفمین درخت اور آپ کا کام ہو گیا۔

شکل 28-8اور شکل 29-8دکھاتے ہیں کہ سپیم پیغام کے لیے ہف مین ٹری کیسے بنایا گیا ہے۔



تصویر 29-8ہف مین درخت کو اگانا، آخری مرحلہ

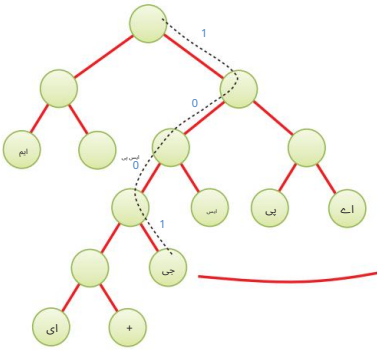
پیغام کو کوڈ کرنا اب جب کہ آپ کے پاس ہف مین ٹری ہے، آپ پیغام کو کیسے انکوڈ کرتے ہیں؟ آپ ایک کوڈ ٹیبل بنا کر شروع کرتے ہیں، جس میں ہر کردار کے
ساتھ ہف مین کوڈ درج ہوتا ہے۔ بحث کو آسان بنانے کے لیے، ہم یہ فرض کرتے رہتے ہیں کہ صرف ASCIIحروف ہی ممکن ہیں، اس لیے ہمیں 128خلیات کے
ساتھ ایک میز کی ضرورت ہے۔ ہر سیل کا اشاریہ ASCIIکریکٹر کی عددی قدر ہو گی: Aکے لیے 65، B کے لیے 66، وغیرہ۔ سیل کے مشمولات متعلقہ کردار کے
لیے ہف مین کوڈ ہوں گے۔ ابتدائی طور پر، آپ "کوئی کوڈ نہیں" کی نشاندہی کرنے کے لیے کچھ خاص قدر بھر سکتے ہیں جیسے کہ کوئی نہیں یا Pythonمیں
خالی سٹرنگ ان خامیوں کی جانچ پڑتال کے لیے جہاں آپ کسی کردار کے لیے کوڈ بنانے میں ناکام رہے۔

اس طرح کا کوڈ ٹیبل کوڈ شدہ پیغام کو تیار کرنا آسان بناتا ہے: اصل پیغام میں ہر ایک کردار کے لیے، آپ اس کے کوڈ کو کوڈ ٹیبل میں بطور انڈیکس استعمال
کرتے ہیں۔ اس کے بعد آپ بار بار ہف مین کوڈز کو کوڈ شدہ پیغام کے آخر میں شامل کرتے ہیں جب تک کہ یہ مکمل نہ ہو جائے۔

ٹیبل میں کوڈز کو بھرنے کے لیے، آپ ہف مین ٹری کو عبور کرتے ہیں، ہر نوڈ کے راستے پر نظر رکھتے ہوئے جب اس کا دورہ کیا جاتا ہے۔ جب آپ کسی لیف نوڈ پر جاتے ہیں، تو آپ اس نوڈ کی کلید کو ٹیبل کے اشاریہ کے طور پر استعمال کرتے ہیں اور سیل کی قدر میں بائنری سٹرنگ کے طور پر راستہ داخل کرتے ہیں۔ ہر سیل میں کوڈ نہیں ہوتا ہے -صرف وہی جو پیغام میں ظاہر ہوتے ہیں۔ شکل 30-8دکھاتا ہے کہ یہ سپیم پیغام کے لیے کیسا لگتا ہے۔ ٹیبل کو صرف اہم قطاروں کو دکھانے کے لیے مختصر کیا گیا ہے۔

کریکٹر Gکے لیے لیف نوڈ کا راستہ دکھایا گیا ہے کیونکہ درخت کو عبور کیا جا رہا ہے۔

مکمل کوڈ ٹیبل ایک ایسے طریقہ کو کال کرکے بنایا جا سکتا ہے جو جڑ سے شروع ہوتا ہے اور پھر ہر بچے کے لیے بار بار کال کرتا ہے۔ آخر کار، تمام لیف نوڈس کے راستے تلاش کیے جائیں گے، اور کوڈ ٹیبل مکمل ہو جائے گا۔



	انڈیکس کریکٹر کوڈ
32	خلا 01
43	+ 10001
65	اے 111
69	ای 10000
70	
71	جی 1001
77	ایم 00
78	ن
79	اے
80	ص 110
81	
82	آر
83	ایس 101

تصویر 30-8کوڈ ٹیبل بنانا

غور کرنے کے لیے ایک اور چیز: اگر آپ کو ایک بائنری پیغام موصول ہوتا ہے جسے ہف مین کوڈ کے ساتھ کمپریس کیا گیا ہے، تو آپ کو کیسے معلوم ہوگا کہ اسے ڈی کوڈ کرنے کے لیے ہف مین ٹری کو کس طرح استعمال کرنا ہے؟ جواب یہ ہے کہ ہف مین ٹری کو پہلے، بائنری پیغام سے پہلے، کسی ایسے فارمیٹ میں بھیجا جانا چاہیے جس کے لیے پیغام کے مواد کے علم کی ضرورت نہیں ہے۔ یاد رکھیں کہ ہف مین کوڈز ڈیٹا کو کمپریس کرنے کے لیے ہیں، اسے خفیہ کرنے کے لیے نہیں۔ ہف مین ٹری کی مختصر تفصیل بھیجنے کے بعد لمبے پیغام کا کمپریسڈ ورژن بہت سے بٹس بچاتا ہے۔

خلاصہ

□□درخت کناروں سے جڑے ہوئے نوڈس پر مشتمل ہوتے ہیں۔

□□جڑ درخت میں سب سے اوپر کی نوڈ ہے؛ اس کا کوئی والدین نہیں ہے۔

□□تمام نوڈس کے علاوہ درخت میں جڑ کے بالکل ایک پیرنٹ ہوتے ہیں۔

□□ایک بائنری درخت میں، ایک نوڈ کے زیادہ سے زیادہ دو بچے ہوتے ہیں۔

□□درخت میں لیف نوڈس میں چائلڈ نوڈس نہیں ہوتے اور جڑ تک بالکل ایک راستہ ہوتا ہے۔

□□غیر متوازن درخت وہ ہوتا ہے جس کی جڑ میں دائیں سے زیادہ بائیں نسل کی اولاد ہوتی ہے۔
اولاد، یا اس کے برعکس۔

□□درخت کا ہر نوڈ کچھ ڈیٹا محفوظ کرتا ہے۔ ڈیٹا میں عام طور پر ایک کلیدی قدر ہوتی ہے جسے شناخت کرنے کے لیے استعمال کیا جاتا ہے۔
اسے بتانا۔

□□کناروں کو عام طور پر نوڈ کے بچوں کے حوالے سے ظاہر کیا جاتا ہے۔ کم com
monایک نوڈ سے اس کے والدین کے حوالہ جات ہیں۔

□□درخت کو عبور کرنے کا مطلب ہے اس کے تمام نوڈس کو کچھ پہلے سے طے شدہ ترتیب میں دیکھنا۔

□□سب سے آسان ٹراورسلز پری آرڈر، ان آرڈر اور پوسٹ آرڈر ہیں۔

□□پری آرڈر اور پوسٹ آرڈر ٹراورسلز الجبری تاثرات کو پارس کرنے کے لیے مفید ہیں۔

□□بائنری تلاش کے درخت

□□ایک بائنری سرچ ٹری میں، تمام نوڈس جو نوڈ Aکی اولاد چھوڑے گئے ہیں
کلیدی قدریں Aسے کم؛ تمام نوڈس جو Aکی صحیح اولاد ہیں ان کی کلیدی قدریں Aسے زیادہ (یا اس کے برابر) ہیں۔

□□ثنائی تلاش کے درخت O(log N)میں تلاش، اندراج، اور حذف کرتے ہیں
وقت

□□بائنری سرچ ٹری میں نوڈ کی تلاش میں ایک نوڈ کی کلیدی قدر سے ملنے والی گول کلید کا موازنہ کرنا اور اگر گول کی کم ہے تو
اس نوڈ کے بائیں چائلڈ کے پاس جانا یا اگر گول کلید ہے تو نوڈ کے دائیں بچے کی طرف جانا شامل ہے۔ زیادہ

□□داخل کرنے میں نیا نوڈ داخل کرنے کے لیے جگہ تلاش کرنا اور پھر اس کا حوالہ دینے کے لیے چائلڈ فیلڈ کو اس کے نئے پیرنٹ (یا
درخت کی جڑ) میں تبدیل کرنا شامل ہے۔

□□ایک ان آرڈر ٹراورسل وزٹ نوڈس کو چڑھنے والی کلیدوں کی ترتیب میں۔

□□جب نوڈ کے کوئی بچے نہیں ہوتے ہیں، تو آپ چائلڈ فیلڈ کو صاف کر کے اسے حذف کر سکتے ہیں۔
اس کے والدین (مثال کے طور پر، اسے Pythonمیں Noneپر سیٹ کرنا)۔

□□جب نوڈ میں ایک بچہ ہوتا ہے، تو آپ اس میں چائلڈ فیلڈ سیٹ کرکے اسے حذف کرسکتے ہیں۔
والدین اپنے بچے کی طرف اشارہ کریں۔

□□جب کسی نوڈ کے دو بچے ہوتے ہیں، تو آپ اسے اس کے successorسے تبدیل کر کے اور subtreeسے جانشین کو حذف
کر کے اسے حذف کر سکتے ہیں۔

□□آپ Aمیں کم از کم نوڈ تلاش کرکے نوڈ Aکے جانشین کو تلاش کرسکتے ہیں۔
دائیں ذیلی درخت۔

□□ڈپلیکیٹ کلیدی اقدار والے نوڈس کو اضافی کوڈنگ کی ضرورت ہوتی ہے کیونکہ عام طور پر صرف
ان میں سے ایک (پہلا) ایک تلاش میں پایا جاتا ہے، اور اپنے بچوں کو منظم کرنے سے اندراج اور حذف کرنے کی سازش ہوتی ہے۔

□□درختوں کو کمپیوٹر کی میموری میں ایک صف کے طور پر پیش کیا جا سکتا ہے، حالانکہ حوالہ پر مبنی نقطہ نظر زیادہ عام اور میموری کو موثر ہے۔

□□ہف مین ٹری ایک بائنری ٹری ہے (لیکن سرچ ٹری نہیں) جسے ڈیٹا کمپریشن الگورتھم میں استعمال کیا جاتا ہے جسے ہف مین کوڈنگ کہتے ہیں۔

□□ہف مین کوڈ میں، جو حروف اکثر نظر آتے ہیں ان کو سب سے کم بٹس کے ساتھ کوڈ کیا جاتا ہے، اور جو شاذ و نادر ہی نظر آتے ہیں ان کو سب سے زیادہ بٹس کے ساتھ کوڈ کیا جاتا ہے۔

□□ہف مین ٹری میں راستے ہر ایک لیف نوڈس کے لیے کوڈ فراہم کرتے ہیں۔

□□لیف نوڈ کی سطح اس کی کلید کے لیے کوڈ میں استعمال ہونے والے بٹس کی تعداد کی نشاندہی کرتی ہے۔

□□ہف مین کوڈڈ پیغام میں کم سے کم کثرت سے ظاہر ہونے والے حروف کو ہف مین درخت کی گہری سطح پر لیف نوڈس میں رکھا جاتا ہے۔

سوالات ان سوالات کا مقصد قارئین کے لیے خود امتحان ہے۔ جوابات ضمیمہ سی میں مل سکتے ہیں۔

.1بائنری سرچ ٹری میں اندراج اور حذف کرنے کے لیے کیا بگ Oوقت درکار ہوتا ہے؟

.2ایک بائنری درخت ایک تلاش درخت ہے اگر

aہر نان لیف نوڈ میں ایسے بچے ہوتے ہیں جن کی کلیدی قدریں اس سے کم یا برابر ہوتی ہیں۔
والدین

ب ہر نون لیف نوڈ کی کلیدی اقدار کی کلیدوں کا مجموعہ یا مجموعہ ہے۔
اس کے بچے

cہر بائیں بچے کی کلید اس کے والدین سے کم ہوتی ہے اور ہر دائیں بچے کے پاس اس کے والدین سے بڑی یا اس کے برابر کی ہوتی ہے۔

dجڑ سے لے کر ہر لیف نوڈ تک کے راستے میں، ہر نوڈ کی کلید اس سے بڑی ہوتی ہے۔
یا اس کے والدین کی کلید کے برابر۔

.3صحیح یا غلط: اگر آپ درخت کو عبور کرتے ہیں اور ہر نوڈ کے راستے کو ایک سلسلہ کے طور پر پرنٹ کرتے ہیں
حروف Lاور Rاس بات کے لیے کہ آیا راستہ ہر قدم پر بائیں یا دائیں بچے کی پیروی کرتا ہے، کچھ ڈپلیکیٹ راستے ہو سکتے ہیں۔

.4جب ترتیب شدہ صف میں ڈیٹا کو ذخیرہ کرنے کے مقابلے میں، اس میں ذخیرہ کرنے کا بنیادی فائدہ
ایک بائنری تلاش درخت ہے

aبگ Oنوٹیشن میں ٹراورسل ٹائم جیسا ہی سرچ ٹائم ہونا۔

ب آئٹمز ڈالنے یا حذف کرتے وقت ڈیٹا کاپی کرنے کی ضرورت نہیں ہے۔

c O(log N)وقت میں کسی چیز کو تلاش کرنے کے قابل ہونا۔

dایک کلید ہونا جو کلید کے ذریعہ شناخت شدہ قدر سے الگ ہے۔

20 .5نوڈس کے ساتھ ایک مکمل، متوازن بائنری درخت میں، اور جڑ کو سمجھا جاتا ہے۔
سطح ،0سطح 4پر کتنے نوڈس ہیں؟

.6بائنری درخت کے ذیلی درخت میں ہمیشہ ہوتا ہے۔

aایک جڑ جو اہم درخت کی جڑ کا بچہ ہے۔

ب ایک جڑ جو مرکزی درخت کی جڑ سے جڑی ہوئی ہے۔

cمرکزی درخت سے کم نوڈس۔

dنوڈس کی مساوی یا بڑی تعداد کے ساتھ ایک بہن بھائی۔

اوعام طور پر sepaہیں _____ _____ .7درختوں کو بطور آبجیکٹ لاگو کرتے وقت، درجہ بندی کی جاتی ہے۔

.8بائنری سرچ ٹری میں نوڈ تلاش کرنے میں نوڈ سے نوڈ تک جانا، پوچھنا شامل ہے

aتلاش کی کلید کے سلسلے میں نوڈ کی کلید کتنی بڑی ہے۔

ب نوڈ کی کلید اس کے دائیں یا بائیں بچے کی کلید سے کتنی بڑی ہے۔

cآپ کس لیف نوڈ تک پہنچنا چاہتے ہیں۔

dچاہے آپ جس سطح پر ہیں وہ سرچ کلید کے اوپر ہے یا نیچے۔

.9ایک غیر متوازن درخت ایک ہے۔

aجس میں زیادہ تر کیز کی قدریں اوسط سے زیادہ ہوتی ہیں۔

ب جہاں مرکزی نوڈ کے اوپر نیچے سے زیادہ نوڈس ہیں۔

cجہاں لیف نوڈس اپنے برابر کے بائیں بچے کے طور پر زیادہ کثرت سے ظاہر ہوتے ہیں۔
صحیح بچے کے طور پر، یا اس کے برعکس.

dجس میں روٹ یا کسی اور نوڈ میں دائیں اولاد کے مقابلے میں بہت زیادہ بائیں اولاد ہوتی ہے، یا اس کے برعکس۔

.10صحیح یا غلط: ایک درجہ بندی کا فائل سسٹم بنیادی طور پر ایک بائنری سرچ ٹری ہے، حالانکہ یہ
غیر متوازن ہو سکتا ہے۔

.11نوڈ داخل کرنا انہی مراحل سے شروع ہوتا ہے ایک نوڈ _____

.12درختوں کے ڈیٹا ڈھانچے کو عبور کرنا

aمختلف ٹراورسل آرڈرز کو سنبھالنے کے لیے متعدد طریقوں کی ضرورت ہوتی ہے۔

ب تکراری افعال یا جنریٹرز کا استعمال کرتے ہوئے لاگو کیا جا سکتا ہے۔

cسرنی ڈیٹا ڈھانچے کو عبور کرنے سے کہیں زیادہ تیز ہے۔

dاشیاء کے نرم حذف کو عملی بنانے کا ایک طریقہ ہے۔

ڈیٹا _____ .13جب ایک درخت انتہائی غیر متوازن ہوتا ہے تو وہ اس طرح برتاؤ کرنے لگتا ہے۔

ساخت

14.فرض کریں کہ ایک نوڈ Aکے پاس بائنری سرچ ٹری میں جانشین نوڈ Sہے جس کا کوئی ڈپلیکیٹ نہیں ہے لیکن اس سے چھوٹا یا اس کے برابر ہے۔ _____ چابیاں پھر Sکے پاس ایک کلید ہونی چاہیے جو اس سے بڑی ہو۔

_____

15.بائنری سرچ ٹری میں نوڈس کو حذف کرنا پیچیدہ ہے کیونکہ

aجانشین کے نیچے ذیلی درختوں کو کاپی کرنے کے لیے ایک اور ٹراورسل کی ضرورت ہوتی ہے۔

ب جانشین تلاش کرنا مشکل ہے، خاص طور پر جب درخت غیر متوازن ہو۔

cدرخت ایک سے زیادہ درختوں، ایک جنگل میں تقسیم ہو سکتا ہے، اگر یہ صحیح طریقے سے نہیں کیا جاتا ہے۔

dنوڈ کے حذف کیے جانے والے چائلڈ نوڈس کی مختلف تعداد کے لیے آپریشن بہت مختلف ہے، 1، 0یا 2۔

16.ایک بائنری درخت میں جو ریاضی کے اظہار کی نمائندگی کرتا ہے،

aآپریٹر نوڈ کے دونوں بچے آپرینڈز ہونے چاہئیں۔

ب پوسٹ آرڈر ٹراورسل کے بعد، قوسین کو شامل کرنا ضروری ہے۔

cپری آرڈر ٹراورسل کے بعد، قوسین کو شامل کرنا ضروری ہے۔

dپری آرڈر ٹراورسل میں، ایک نوڈ کو اس کے بچوں میں سے کسی سے پہلے دیکھا جاتا ہے۔

17.جب ایک درخت کو ایک صف سے ظاہر کیا جاتا ہے، انڈیکس nمیں نوڈ کے دائیں بچے کے پاس ایک ہوتا ہے _____ کی انڈیکس

18.صحیح یا غلط: بائنری سرچ ٹری سے ایک بچے کے ساتھ نوڈ کو حذف کرنے میں اس نوڈ کے جانشین کو تلاش کرنا شامل ہے۔

19.ایک Huffmanدرخت عام طور پر استعمال کیا جاتا ہے ڈیکوڈ ڈیٹا۔ _____

20.مندرجہ ذیل میں سے کون سا ہف مین درخت کے بارے میں درست نہیں ہے؟

aسب سے زیادہ استعمال ہونے والے حروف ہمیشہ درخت کی چوٹی کے قریب ظاہر ہوتے ہیں۔

ب عام طور پر، کسی پیغام کو ڈی کوڈ کرنے میں بار بار جڑ سے کسی راستے پر چلنا شامل ہوتا ہے۔
ایک پتی کو.

cایک کردار کو کوڈنگ میں، آپ عام طور پر ایک پتی سے شروع کرتے ہیں اور اوپر کی طرف کام کرتے ہیں۔

dدرخت کو ترجیحی بنیاد پر ہٹانے اور داخل کرنے کی کارروائیوں سے پیدا کیا جا سکتا ہے۔
چھوٹے درختوں کی قطار۔

تجربات

ان تجربات کو انجام دینے سے باب میں شامل موضوعات کے بارے میں بصیرت فراہم کرنے میں مدد ملے گی۔ کوئی پروگرامنگ شامل نہیں ہے۔

A-8بائنری سرچ ٹری ویژولائزیشن ٹول کا استعمال کرکے 20بے ترتیب درختوں کو 20اشیاء کی مطلوبہ تعداد کے طور پر استعمال کریں۔ آپ کتنے فیصد کو سنجیدگی سے غیر متوازن کہیں گے؟

8-B فہرست 12-8 اور پرو میں دکھایا گیا BinarySearchTreeTester.py پروگرام استعمال کریں
درج ذیل تجربات کرنے کے لیے پبلشر کی ویب سائٹ سے کوڈ کی مثالوں کے ساتھ :vided

a ایک نوڈ کو حذف کریں جس کے کوئی بچے نہیں ہیں۔

ب ایک نوڈ کو حذف کریں جس میں 1 چائلڈ نوڈ ہو۔

c ایک نوڈ کو حذف کریں جس میں 2 چائلڈ نوڈس ہوں۔

d داخل کرنے کے لیے نئے نوڈ کے لیے ایک کلید چنیں۔ اس بات کا تعین کریں کہ آپ کے خیال میں اسے درخت میں کہاں ڈالا جائے گا، اور پھر اسے پروگرام کے ساتھ داخل کریں۔ کیا یہ طے کرنا آسان ہے کہ یہ کہاں جائے گا؟

e پچھلے مرحلے کو کسی اور کلید کے ساتھ دہرائیں لیکن اسے دوسرے بچے میں ڈالنے کی کوشش کریں۔
شاخ مثال کے طور پر، اگر آپ کا پہلا نوڈ بائیں بچے کے طور پر ڈالا گیا تھا، تو کوشش کریں کہ ایک کو دائیں بچے کے طور پر یا دائیں ذیلی درخت میں ڈالیں۔

8-C فہرست 12-8 میں دکھایا گیا BinarySearchTreeTester.py پروگرام آئٹمز کے داخل کرنے کے آرڈر کی بنیاد پر 7 سطحوں پر 11 نوڈس کے ابتدائی درخت کو پرنٹ کرتا ہے۔ درخت کے مکمل طور پر متوازن ورژن میں 4 سطحوں پر ایک جیسے نوڈس محفوظ ہوں گے۔ درخت کو صاف کرنے کے لیے پروگرام کا استعمال کریں، اور پھر اس بات کا تعین کریں کہ بال اینسڈ ٹری بنانے کے لیے وہی چابیاں کس ترتیب میں ڈالنی ہیں۔ اپنا آرڈر کرنے کی کوشش کریں اور دیکھیں کہ کیا درخت متوازن طور پر نکلتا ہے۔ اگر نہیں، تو ایک اور آرڈر کرنے کی کوشش کریں۔ کیا آپ چند جملوں میں داخل کرنے کی ترتیب کو بیان کر سکتے ہیں جو ہمیشہ چابیاں کے مخصوص سیٹ سے ایک متوازن بائنری سرچ ٹری بنائے گا؟

8-D بائنری سرچ ٹری ویژولائزیشن ٹول استعمال کریں تاکہ ہر ممکن طریقے سے نوڈ کو حذف کیا جا سکے۔

صورت حال

پروگرامنگ پروجیکٹس

پروگرامنگ پروجیکٹس کو حل کرنے کے لیے پروگرام لکھنا مواد کے بارے میں آپ کی سمجھ کو مستحکم کرنے میں مدد کرتا ہے اور یہ ظاہر کرتا ہے کہ باب کے تصورات کو کس طرح لاگو کیا جاتا ہے۔ (جیسا کہ تعارف میں بتایا گیا ہے، اہل اساتذہ پبلشر کی ویب سائٹ پر پروگرامنگ پروجیکٹس کے مکمل حل حاصل کر سکتے ہیں۔)

8.1 اس باب میں بیان کردہ BinarySearchTree کلاس کو تبدیل کریں تاکہ ڈوپلی کیٹ کیز کے ساتھ نوڈس کی اجازت دی جا سکے۔ تین طریقے متاثر ہوئے ہیں: insert(), find()__ اور delete()۔ نئے بائیں بچوں کو مساوی کلیدوں کے درمیان سب سے کم سطح پر داخل کرنے کا انتخاب کریں، جیسا کہ شکل 8-26 کے بائیں جانب دکھایا گیا ہے، اور ہمیشہ برابر کیز کے درمیان سب سے گہری تلاش کریں اور حذف کریں۔ مزید خاص طور پر، find()__ اور search() طریقوں کو مساوی کلیدوں کے درمیان گہری est کو واپس کرنا چاہئے جس کا سامنا کرنا پڑتا ہے لیکن ایک اختیاری پیرامیٹر کو سب سے کم تلاش کرنے کی وضاحت کرنے کی اجازت دینی چاہئے۔ insert() طریقہ کار کو اس صورت میں ہینڈل کرنا چاہیے جب داخل کی جانے والی آئٹم موجودہ نوڈ کو ڈپلیکیٹ بناتی ہے، ایک نیا نوڈ ڈال کر گہری ڈپلیکیٹ کلید کے نیچے خالی بائیں بچے کے ساتھ۔ ڈیلیٹ ()طریقہ کو ڈپلیکیٹ کیز کے درمیان سب سے گہرے نوڈ کو حذف کرنا چاہیے، اس طرح ڈپلیکیٹ کیز کے درمیان LIFO یا اسٹیک جیسا برتاؤ فراہم کرتا ہے۔ حذف کرنے کے معاملات کے بارے میں احتیاط سے سوچیں اور آیا

جانشین نوڈس کا انتخاب مختلف اقدار سے وابستہ کئی ڈپلیکیٹ کلیدیں داخل کرنے والے درخت پر آپ کا نفاذ کیسے کام کرتا ہے اس کا مظاہرہ کریں۔ پھر ان کیز کو حذف کریں اور ان کی قدریں دکھائیں تاکہ یہ واضح ہو جائے کہ آخری ڈپلیکیٹ داخل کیا گیا پہلا ڈپلیکیٹ حذف کیا گیا ہے۔

8.2ایک پروگرام لکھیں جو پوسٹ فکس ایکسپریشن پر مشتمل سٹرنگ لے اور الجبری ایکسپریشن کی نمائندگی کرنے کے لیے ایک بائنری ٹری بنائے جیسا کہ شکل 16-8میں دکھایا گیا ہے۔
آپ کو BinaryTreeکلاس کی ضرورت ہے ، جیسے BinarySearchTreeکی ، لیکن بغیر کسی کلید یا نوڈس کی ترتیب کے۔ ,(),insert ,find() اور ()deleteطریقوں کی بجائے ، آپ کو سنگل نوڈ BinaryTreesبنانے کی اہلیت کی ضرورت ہے جس میں ایک واحد آپرینڈ ہو اور دو بائنری ٹریوں کو ملا کر تیسرا آپریٹر کے ساتھ روٹ نوڈ کے طور پر بنایا جائے۔ آپریٹرز اور آپرینڈز کا نحو وہی ہے جو باب 4کے PostfixTranslate.py ماڈیول میں استعمال کیا گیا تھا۔ آپ ان پٹ سٹرنگ کو آپریٹر اور آپرینڈ ٹوکنز میں پارس کرنے کے لیے اس ماڈیول میں ()NextTokenفنکشن کا استعمال کر سکتے ہیں۔ آپ کو حد بندی کے طور پر قوسین کی ضرورت نہیں ہے کیونکہ پوسٹ فکس اظہار ان کا استعمال نہیں کرتے ہیں۔

تصدیق کریں کہ ان پٹ ایکسپریشن ایک ہی الجبری ایکسپریشن تیار کرتا ہے اور اگر ایسا نہیں ہوتا ہے تو ایک استثناء بڑھائیں۔ درست الجبری بائنری درختوں کے لیے، آؤٹ پٹ فارمز میں ان پٹ کا ترجمہ کرنے کے لیے درخت کے پری، ان- اور پوسٹ آرڈر ٹراورسلز کا استعمال کریں۔ آؤٹ پٹ ترجمہ میں آپریٹر کی ترجیح کو واضح کرنے کے لیے ان آرڈر ٹراورسل کے لیے قوسین کو شامل کریں۔ اپنے پروگرام کو کم از کم درج ذیل تاثرات پر چلائیں:

a 91 95 + 15 + 19 + 4 *

ب بی بیٹ *اے سی * 4

c 42

A 57 dاس میں ایک استثنا پیدا ہونا چاہئے۔

/ + eاس سے ایک استثنا پیدا ہونا چاہئے۔

8.3ہف مین کوڈنگ اور ڈی کوڈنگ کو لاگو کرنے کے لیے ایک پروگرام لکھیں۔ اسے کرنا چاہیے۔
درج ذیل:

- ٹیکسٹ میسج (سٹرنگ) کو قبول کریں۔
- اس پیغام کے لیے ہف مین ٹری بنائیں۔
- کوڈ ٹیبل بنائیں۔
- ٹیکسٹ میسج کو بائنری میں انکوڈ کریں۔
- بائنری پیغام کو واپس متن میں ڈی کوڈ کریں۔
- بائنری پیغام میں بٹس کی تعداد اور ان پٹ پیغام میں حروف کی تعداد دکھائیں۔

اگر پیغام مختصر ہے، تو پروگرام کو تخلیق کرنے کے بعد ہف مین ٹری کو ظاہر کرنے کے قابل ہونا چاہیے۔ آپ بائنری پیغامات کو ذخیرہ کرنے کے لیے Pythonسٹرنگ متغیرات کو حروف 1اور 0کی ترتیب کے طور پر استعمال کر سکتے ہیں۔ بائٹ رے کا استعمال کرتے ہوئے اصل بٹ ہیرا پھیری کرنے کی فکر نہ کریں جب تک کہ آپ واقعی ایسا نہ کریں۔ کوڈ ٹیبل بنانے کا آسان ترین طریقہ

Pythonمیں ڈکشنری (dict)ڈیٹا ٹائپ استعمال کرنا ہے۔ اگر یہ ناواقف ہے تو، یہ بنیادی طور پر ایک صف ہے جسے سٹرنگ یا ایک کریکٹر کے ذریعے ترتیب دیا جا سکتا ہے۔ یہ BinarySearchTreeTester.pyماڈیول میں استعمال ہوتا ہے جو لسٹنگ 12-8میں نقشہ کمانڈ کرنے کے لیے دکھائے گئے ہیں۔ اگر آپ انٹیجر انڈیکسڈ صف کو استعمال کرنے کا انتخاب کرتے ہیں، تو آپ ایک حرف کو عدد میں تبدیل کرنے کے لیے Pythonکے ()ordفنکشن کا استعمال کر سکتے ہیں لیکن آگاہ رہیں کہ اگر آپ پیغام میں صوابدیدی یونیکوڈ حروف جیسے ایموجیز ()کی اجازت دیتے ہیں تو آپ کو ایک بڑی صف کی ضرورت ہوگی۔ .

8.4درخت کے توازن کی پیمائش مشکل ہو سکتی ہے۔ آپ دو آسان اقدامات کر سکتے ہیں: نوڈ بیلنس اور لیول (یا اونچائی) بیلنس۔ جیسا کہ پہلے ذکر کیا گیا ہے، متوازن درختوں کے بائیں اور دائیں ذیلی درختوں میں تقریباً مساوی تعداد میں نوڈس ہوتے ہیں۔ اسی طرح، بائیں اور دائیں ذیلی درختوں میں سطحوں (یا اونچائی) کی تقریبا برابر تعداد ہونی چاہیے۔ درج ذیل طریقے لکھ کر BinarySearchTreeکلاس کو بڑھائیں :

aنوڈ بیلنس - ()دائیں ذیلی درخت میں نوڈس کی تعداد کو بائیں ذیلی درخت میں نوڈس کی تعداد کو کم کرتا ہے

ب لیول بیلنس() —دائیں سب ٹری میں لیولز کی تعداد کا حساب لگاتا ہے مائنس بائیں سب ٹری میں لیولز کی تعداد

cغیر متوازن نوڈس (بائی — (1 =نوڈ کیز کی ایک فہرست لوٹاتا ہے جہاں بیلنس میٹرکس میں سے کسی ایک کی مطلق قدر حد سے تجاوز کر جاتی ہے ، جو پہلے سے طے شدہ 1ہوتی ہے۔

ان تینوں طریقوں کے لیے (بار بار آنے والے) مددگار طریقوں کی ضرورت ہوتی ہے جو درخت کے اندر نوڈس پر جڑے ذیلی درختوں کو عبور کرتے ہیں۔ ایک متوازن درخت میں، غیر متوازن نوڈس کی فہرست خالی ہوگی۔ ایک خالی BinarySearchTreeمیں کلیدوں کی درج ذیل چار فہرستیں ڈال کر (ترتیب میں، بائیں سے دائیں)، نتیجے میں آنے والے -15نوڈ ٹری کو پرنٹ کرکے، نتیجے میں آنے والے روٹ نوڈ کے نوڈ اور لیول بیلنس کو پرنٹ کرکے، اور پھر فہرست کو پرنٹ کرکے اپنے اقدامات کو آزمائیں۔ by=1اور by=2کے ساتھ غیر متوازن کلیدوں کا ۔

13، 15]، [8، 4] ، 2، 3، 1، 6، 5، 7، 12، 10، 9، 11، 14، 13، 8.5]
9، 11، 14، 13، 15]، [8، 4، 2، 3، 1، 6، 5، 7، 12، 10، 9، 11، 14،
8، 12، 10، 9، 11، 14، 13، 15]، [8، 4، 5، 6، 7، 3، 2، 1، 12] ، 10،
[7، 6، 5، 4، 3، 2، 1،

8.5ہر بائنری درخت کو ایک صف کے طور پر دکھایا جا سکتا ہے، جیسا کہ عنوان والے حصے میں بیان کیا گیا ہے۔
"درختوں کو صفوں کے طور پر پیش کیا گیا ہے۔" ایک صف کو درخت کے طور پر پیش کرنے کا الٹ، تاہم، صرف کچھ صفوں کے لیے کام کرتا ہے۔ درخت کے گم شدہ نوڈس کو صف کے خلیوں میں کچھ پہلے سے طے شدہ قدر کے طور پر پیش کیا جاتا ہے —جیسے کہ کوئی نہیں —جو کہ درخت کے نوڈ میں ذخیرہ شدہ قدر نہیں ہو سکتی۔ اگر صف میں روٹ نوڈ غائب ہے، تو متعلقہ درخت نہیں بنایا جا سکتا۔ ایک فنکشن لکھیں جو ایک صف کو بطور ان پٹ لیتا ہے اور اس کے مواد سے بائنری ٹری بنانے کی کوشش کرتا ہے۔ ہر وہ سیل جو Noneنہیں ہے ایک ٹری نوڈ پر ذخیرہ کرنے کی قدر ہے۔

جب آپ پیرنٹ نوڈ کے بغیر کسی نوڈ کے سامنے آتے ہیں (روٹ نوڈ کے علاوہ)، فنکشن کو ایک استثناء اٹھانا چاہئے جس سے یہ ظاہر ہوتا ہے کہ درخت نہیں بنایا جا سکتا۔ نوٹ کریں کہ نتیجہ ضروری نہیں کہ بائنری سرچ ٹری ہو، صرف ایک بائنری ٹری۔ اشارہ: لیف نوڈس سے جڑ تک کام کرنا آسان ہے، ہر اس سیل کے لیے نوڈس بنانا جو کوئی نہیں ہے اور نتیجے میں آنے والے نوڈ کو دوبارہ ان پٹ ارے کے اسی سیل میں اسٹور کرنا

بازیافت جب اسے کسی اور سطح پر نوڈ کے ذیلی درخت کے طور پر استعمال کیا جاتا ہے۔ فنکشن کو چلانے کا نتیجہ درج ذیل صفوں پر پرنٹ کریں جہاں n = None۔ صف میں موجود اقدار کو یا تو کلید یا نوڈ کی قدر کے طور پر ذخیرہ کیا جا سکتا ہے کیونکہ درخت کو بائنری سرچ ٹری سے تعبیر نہیں کیا جائے گا۔

[]،[ن، ن، ن]، [55، 12، [71 ،12 ،55]ن، [55، 12، [4ن، ،4ن، ن، ن، ن، ،8ن، این ، این ، این ، این ، این ،

این ، ، 6این ، این] ، ، 12 ، 55]این ، این ، این ، این ، این ، ، 4این ، ، 8این ، این ، این ، این ، این ، این ، این ،

این ، این ، این ، این ، این ، [n، n، 6، n

انڈیکس

نمبرز

ای

آر

یو

وی